Eine Metrik-Suite zur Analyse des Einsatzes von Interfaces in Java

---

Bachelorarbeit

Philip Mayer

Vorgelegt zur Erlangung des Bachelor of Science in Angewandter Informatik

Institut für Informationssysteme, Wissensbasierte Systeme, Fachbereich Informatik
Universität Hannover

Abstract

Using interfaces in large programs has several advantages, among them being an increase in comprehensibility of the code and extended substitutability of classes. This is particularly true for partial interfaces, that is interfaces that cover only a subset of the total set of published methods of a class. However, analysis of large frameworks such as the Java API suggests that interfaces are only sparsely used.

In this thesis I present a metrics suite and additional methods to assist the developer in creating and using interfaces, both total and partial. Two goals for interface-based programming are presented – one based on the concept of the “usage context” – and metrics are derived which evaluate their fulfilment. This metrics suite is then compared with interface metrics already available and evaluated by means of empirical validation. Finally, an implementation of the metrics suite as a plug-in for an Integrated Development Environment (IDE) is presented.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, September 2003

_____

Philip Mayer

Eingegangen am (Datum/Stempel): _____

Ich möchte Herrn PD Dr. Friedrich Steimann für die stets hervorragende Betreuung meiner Arbeit danken.

Inhaltsverzeichnis

1 Einleitung

1.1 Problemstellung

Die Nutzung von Interfaces als Typen von Variablen ist immer wieder empfohlen worden (vgl. z.B. Gamma et al. 1995, Fowler 2000, Steimann 2001) und kann schon beinahe als Allgemeingut aufgefasst werden. In der Realität ist diese Nutzung jedoch noch nicht besonders weit verbreitet: Eine kürzlich veröffentlichte Studie (Steimann et al. 2003) zeigt, dass durchschnittlich nur eine von fünf Variablen in großen Java-Projekten mit Interfaces typisiert ist.

Spekulationen über mögliche Gründe schließen mit ein, dass die Einführung und Wartung von Interfaces für den Entwickler zusätzlichen Aufwand darstellen; dieser Aufwand kann jedoch durch geeignete Refactorings reduziert werden (vgl. Meißner 2003). Die Identifikation derjenigen Klassen, bei welchen Interfaces nicht oder nur wenig eingesetzt werden und sich folglich Refactorings anbieten, ist jedoch, insbesondere bei großen Projekten, ein nichttriviales Problem; hier können Metriken und Analysetools helfen. Diese sind allerdings nicht nur zur Auswahl von geeigneten Zielen für Refactorings geeignet, sondern ermöglichen dem Entwickler auch eine detaillierte Nutzungsanalyse seiner Klassen und helfen damit beim Verständnis des Codes.

Da Interfaces und ihre konzeptuelle Entsprechung, die Rollen, als grundlegende Elemente eines objektorientierten Systems angesehen werden können (Steimann 2000, Steimann 2001, Steimann et al. 2003), ist es von Interesse, die Programmierung im Interface- oder Rollenparadigma – das *interface-based programming* – einem möglichst breiten Feld von Entwicklern zugänglich zu machen. Interfaces sind jedoch nicht nur von der konzeptuellen Warte aus zu empfehlen; auch technische Gründe wie die steigende Modularität und Abstraktion des Codes, woraus eine erhöhte Wartbarkeit resultiert, sprechen für ihre Nutzung.

Aus den genannten Punkten leitet sich das Ziel dieser Arbeit ab: *Die Entwicklung und Implementation einer Metrik-Suite und weiterer Herangehensweisen zur Analyse des Einsatzes von Interfaces in Java – zum Zwecke der Unterstützung des vermehrten und besseren Einsatzes von Interfaces*.

1.2 Beitrag dieser Arbeit

Bevor mit der Entwicklung einer Metrik-Suite mit den oben genannten Eigenschaften begonnen werden kann, ist eine eingehende Beschäftigung sowohl einerseits mit Interfaces, aber auch andererseits mit Metriken erforderlich. So erfordert die Aufstellung von Zielen für die Programmierung unter intensiver Verwendung von Interfaces zunächst die Untersuchung des Konzepts des Interfaces an sich: Sowohl die historische Entwicklung als auch die unterschiedlichen Umsetzungen in den verschiedenen objektorientierten Programmiersprachen sollen hierbei genauer beleuchtet und Kategorisierungsmöglichkeiten für Interfaces aufgezeigt werden. Im Anschluss werden dann zwei klar definierte Ziele des Interfaceparadigmas formuliert.

Neben einer genaueren Betrachtung des Interfacekonzepts ist auch die Untersuchung der Ursprünge und Entwicklung sowie eine Kategorisierung von Metriken notwendig – vorhandene Metriken können direkt zur Untersuchung der Interfacenutzung eingesetzt werden oder entsprechend umdefiniert zum Einsatz kommen. Neben einer Darlegung verfügbarer Metriken im Allgemeinen werden in dieser Arbeit daher speziell Metriken und Metrik-Suiten mit Bezug zum Interfaceparadigma untersucht.

Die Einführung eigener Metriken ist, wie sich im Verlauf der Untersuchung vorhandener Metriken herausstellen wird, für eine vollständige Untersuchung des Interfaceparadigmas unabdingbar, da einige Aspekte der Nutzung von Interfaces bislang von keiner Metrik untersucht werden. Daher wird eine Metrik-Suite mit zwei bereits vorhandenen und fünf neuentwickelten Metriken definiert, welche direkt die Erreichung der zuvor formulierten Ziele des Interfaceparadigmas messen. Die Suite wird mittels einer bekannten Arbeitsmethode zur Definition von Metriken entwickelt – dem Goal-Question-Metric-Ansatz (Basili et al. 1994).

Um eine direkte Evaluation der Metriken zu erlauben, werden diese im Anschluss in einer Art Feldversuch auf vier Open-Source-Projekte angewandt. Die Ergebnisse können als Vergleichswerte und Anhaltspunkte für eigene Berechnungen dienen. Ebenso werden die neu definierten Metriken von bereits vorhandenen abgegrenzt.

Neben der Definition der Metrik-Suite sowie weiterer Analysemethoden in der Theorie werden in dieser Arbeit auch direkt für die Praxis relevante Ergebnisse erzielt: Die Metrik-Suite sowie weitere Analysemethoden werden als Plug-In einer integrierten Entwicklungsumgebung (IDE) realisiert, um eine möglichst direkte Verwendung durch den Softwareentwickler zu erlauben.

Die genaue Untersuchung des Pro und Contra der interfacebasierten Programmierung ist dagegen nicht Thema dieser Arbeit; viele andere Arbeiten haben sich bereits mit dieser Begründung befasst (z. B. Gamma et al. 1995, Steimann 2000, Steimann 2001 und andere). Ebenso wird auch in die Domäne der Metriken nur so weit eingeleitet, wie es für das Verständnis und die Einordnung der hier definierten Metrik-Suite erforderlich ist.

1.3 Gliederung

Wie bereits im vorigen Abschnitt erwähnt zieht sich die Entwicklung der Metrik-Suite als roter Faden durch die Kapitel:

- In Kapitel 2 wird das Interface-Konzept vorgestellt. Es werden historische Aspekte beleuchtet, mögliche Kategorisierungen vorgestellt und Ziele der Verwendung von Interfaces definiert.
- Kapitel 3 ist den Metriken gewidmet. Hier wird auf den Ursprung der Metriken eingegangen, ebenso werden Metriken und Metrik-Suiten mit Bezug zum Interfacekonzept untersucht und Evaluierungsmöglichkeiten für Metriken vorgestellt.
- In Kapitel 4 werden sowohl eine Metrik-Suite als auch weitere Analysemethoden definiert, welche die Entwicklung innerhalb des Interfaceparadigmas unterstützen. Die Metrik-Suite wird anhand von vier Open-Source-Projekten erprobt und von bestehenden Metrik-Suiten und Einzelmetriken abgegrenzt.
- Kapitel 5 liefert Details über die Implementation der Suite und weiterer Analysemethoden als Plug-Ins einer Entwicklungsumgebung.
- In Kapitel 6 und 7 finden sich Zusammenfassung, Ausblick und Schlussbemerkung.

2 Interfaces

Der Begriff Interface – zu Deutsch Schnittstelle – wird in der Informatik vielfältig genutzt. Bevor im Folgenden Interfaces in der Verwendung in modernen Programmiersprachen – insbesondere Java – untersucht werden, soll die ursprüngliche Bedeutung des Begriffs in Erinnerung gerufen werden.

Eine Schnittstelle bezeichnet im Allgemeinen eine Stelle in einem System, an welcher zwei oder mehr Komponenten aneinander gekoppelt werden. Die Encyclopedia of Software Engineering enthält folgenden Eintrag zu „Interface Definition“:

> *Die Konstruktion von Systemen erfordert notwendigerweise die bedachtsame Dekomposition des Gesamtsystems sowohl in die funktionale Dimension als auch die der Subsysteme […] Die so zerlegten Funktionen und Subsysteme müssen allerdings miteinander interagieren und interoperieren. Die Beschaffenheit dieser Interaktion und Interoperation wird normalerweise in einer Interfacedefinition explizit angezeigt und präzisiert. […]* (Marciniak 1994, S. 1316, eigene Übersetzung)

Das Konzept des Interfaces wurde im Zuge der Fokussierung auf die Strukturierung von Software entwickelt. Clements und Northrop sehen die Wurzeln der Softwarearchitektur – welche in der Softwarestrukturierung liegen – in einem im Jahr 1968 von Edsger Dijkstra veröffentlichten Artikel (Dijkstra 1968). Sie schreiben insbesondere: „[…]Dijkstra pointed out that it pays to be concerned with how software is partitioned and structured, as opposed to simply programming so as to produce a correct result“ (Clements/Northrop 1996). Dijkstra gliedert Software hierarchisch in unterschiedliche Abstraktionsstufen; jede Stufe spricht dabei nur mit direkt anliegenden Stufen, die jeweils als Interface für alle darunter- bzw. darüber liegenden Abstraktionsstufen dienen.

Parnas (Parnas 1972) strebt neben der Hierarchisierung die Modularisierung als eigenständiges Konzept der Systemgliederung an. Zwischen den einzelnen Modulen befinden sich dabei Interfaces, die möglichst abstrakt gehalten sein sollten; ebenso sollten sie – im Sinne des Geheimnisprinzips – so wenig wie möglich über den inneren Aufbau des jeweiligen Moduls verraten. Jedes Modul stellt Funktionen und/oder Daten zur Verfügung, welche abgerufen werden können. Die Summe dieser Funktionen und Daten stellt die Schnittstelle des Moduls dar – wird ein Modul genutzt, so erfolgt dies nach dem Geheimnisprinzip ausschließlich über diese Schnittstelle.

In diesen Arbeiten liegt der Ursprung des Interfaceparadigmas – also der Programmierweise unter intensiver Nutzung von Interfaces, das später in der von Gamma et al. getroffenen Aussage gipfelt: „Program to an interface, not to an implementation“ (Gamma et al. 1995, S.25). Dieses Paradigma stellt auch die Grundlage dieser Arbeit dar.

Um der Softwarekomplexität Herr zu werden ist es möglich, bestimmten Nutzern nur Teile der Funktionen und Daten eines Moduls zur Verfügung zu stellen. Damit ist die *partielle* Schnittstellendefinition geboren, also eine Auflistung aller einem Nutzer *zur Verfügung gestellten* Funktionen und Daten, welche nicht unbedingt *alle* Funktionen und Daten enthalten muss. Dies ist bereits in der Definition der ITU-T Recommendation X.902 enthalten:

> *Interface: An abstraction of the behaviour of an object that consists of a subset of the interactions of that object together with a set of constraints on when they may occur. Each interaction of an object belongs to a unique interface. Thus the interfaces of an object form a partition of the interactions of that object* (Isolec 1995).

Ein Interface kann folglich auch als Filter angesehen werden: Indem durch ein bestimmtes Interface auf ein Modul zugegriffen wird, kann auch nur das im Interface definierte Verhalten des Moduls erfasst und verwendet werden.

Bereits in der strukturierten Programmierung sind Interfaces verwendet worden, um die öffentlich zugänglichen Funktionen von Modulen zu spezifizieren – z.B. durch Header-Dateien in C. Das Konzept des Interfaces ist jedoch insbesondere in modernen, objektorientierten Programmiersprachen von Bedeutung und hat sich von einer einfachen Schnittstellendefinition zu einem Konzept der Abstraktion und Polymorphie[1] entwickelt. Interfaces haben sich über die Jahre emanzipiert; sie sind z.T. als besondere Ausprägungen des Klassenkonzeptes realisiert – z.B. durch spezielle, abstrakte Klassen in C++ – zum Teil sogar als eigene Kategorie von Typen[2], wie z.B. in Java.

Neben der Integration von Interfaces in einzelne Programmiersprachen existieren so genannte *Interface Definition Languages* (IDL), welche die Spezifikation von Interfaces für Systeme wie CORBA oder COM erlauben. Allerdings enthalten „die Interfacedefinitionen solcher Sprachen […] meist mehr Informationen als das Konstrukt der jeweilig verwendeten Programmiersprache selbst, da einem Protokoll für den entfernten Zugriff auf Objekte bekannt sein muss, welche Parameter gelesen und welche geschrieben werden sowie, welche weiteren Zugriffsvorrausetzungen bestehen“ (Willcock et al. 2001); derartige Definitionen sind beispielsweise in der CORBA IDL (CORBAIDL) oder der Microsoft IDL (Microsoft 2003) anzutreffen. Die in einer IDL beschriebenen Interfaces dienen dann als Templates für die Definition konkreter Interfaces in den jeweiligen Programmiersprachen. Da die IDLs von Programmiersprachen losgelöst arbeiten (und insbesondere auch keine von Außen erkennbare Unterscheidung zwischen partiellen und totalen Interfaces einer Klasse anbieten, siehe unten), werden sie hier jedoch nicht weiter vertieft.

Diese Arbeit untersucht konkret das Interface-Konstrukt von Java. Das englische Wort *Interface* bezeichnet daher im Folgenden stets einen Typ, wie er in Java durch das Schlüsselwort `interface` definiert wird. Das deutsche Wort Schnittstelle, welches zu *Interface* eigentlich sprachlich äquivalent ist, wird hier in seiner ursprünglichen Bedeutung genutzt (s.o.).

2.1 Das Interface-Konstrukt von Java

Mit dem Boom des Internet hat die Sprache Java, welche bereits seit Anfang der 1990er Jahre bei Sun durch eine von James Gosling geführte Entwicklergruppe entwickelt wurde, ihren Durchbruch erzielt. Die Definition von Interfaces in Java weist mehrere Besonderheiten auf. Während ein Interface in C++ und weiteren Sprachen lediglich eine besondere Nutzung des Klassenkonzepts darstellt, bietet Java Klassen und Interfaces als unterschiedliche syntaktische Konzepte an.

Java-Interfaces korrespondieren bis zu einem gewissen Grad mit Protokollen: Ein Protokoll definiert eine bestimmte Verhaltensweise; es ist eine Anleitung zur Kommunikation: Die Beherrschung eines Protokolls zeigt an, dass das jeweilige Objekt fähig ist, auf gewisse Nachrichten zu reagieren. Das eigentliche Objekt ist irrelevant, so lange bekannt ist, dass es das Protokoll beherrscht. Ein Protokoll kann – je nach Definition – auch bestimmte Verhaltensweisen sowie mögliche Zustände und Zustandswechsel beinhalten; diese können in Java Interfaces jedoch nicht ausgedrückt werden.

[1] Unter Polymorphie wird hier der Mechanismus verstanden, Methoden auf einer Variablen aufzurufen, die durch ihren Typ Objekte mehrerer Klassen enthalten kann; es wird dann jeweils diejenige Methode ausgeführt, die in der Klasse der momentan in der Variable befindlichen Objektreferenz definiert ist. Dies gilt sowohl für spätes Binden (Bestimmung der korrekten Methode erst zur Ausführungszeit) als auch für frühes Binden (Bestimmung der korrekten Methode bereits zur Übersetzungszeit).
[2] Aus diesem Grund steht der Begriff Typ im Folgenden stets für eine Klasse oder ein Interface.

Ein Interface in Java kann dennoch als die (entsprechend eingeschränkte) Definition eines Protokolls angesehen werden. Die Deklaration eines Interfaces erzeugt einen neuen Typ, welcher keine Implementation aufweist. Er kann (innere) Klassen und (innere) Interfaces, Konstanten und abstrakte Methoden enthalten; letztere definieren Verhalten im Sinne eines Protokolls. Jede Klasse, welche das Interface implementiert, muss eine Implementation der abstrakten Methoden anbieten (außer sie ist selbst abstrakt).

In Java existiert eine separate Klassen- und Interfacehierarchie; deren Schnittstelle stellt die Implementation von Interfaces durch Klassen dar. Beide Hierarchien erlauben Vererbung, wobei jedoch kategorisch zwischen Implementations- und Interfacevererbung unterschieden werden muss:

- In der *Klassenhierarchie* sind alle Klassen des Systems angeordnet. Eine Klasse kann dabei von maximal einer Klasse direkt *erben*, wobei alle Spezifikationen sowie Implementationen übernommen werden. Dies wird als Implementationsvererbung bezeichnet; aufgrund der Restriktion auf eine Superklasse ist diese Vererbung hier einfach.
- In der *Interfacehierarchie* sind alle Interfaces des Systems angeordnet. Ein Interface kann dabei als direkte Erweiterung eines oder mehrerer Interfaces deklariert werden; dabei übernimmt das erweiternde Interface alle Deklarationen aus dem erweiterten Interface. Dies wird als Interfacevererbung bezeichnet; aufgrund der mehreren möglichen Superinterfaces ist die Vererbung hier mehrfach.
- An der *Hierarchieschnittstelle* kann eine Klasse beliebig viele Interfaces *implementieren*; ebenso kann jedes Interface von beliebig vielen Klassen implementiert werden, die sich zudem nicht in einem durch Vererbung bestimmten Verwandtschaftsverhältnis befinden müssen.

Ein von einer Klasse implementiertes Interface wird als Supertyp der Klasse angesehen. Dies hat Auswirkungen auf die Polymorphie, wie die Java Language Specification verdeutlicht:

> *A variable whose declared type is an interface may have as its value a reference to any instance of a class which implements the specified interface* (Gosling et al. 2000, S. 199).

Die Implementation eines Interfaces durch eine Klasse muss explizit deklariert werden; Strukturkonformität wird nicht durch den Compiler erschlossen:

> *It is not sufficient that the class happen to implement all the abstract methods of the interface; the class or one of its superclasses must actually be declared to implement the interface, or else the class is not considered to implement the interface* [sic] (Gosling et al. 2000, S. 199).

Die Nutzung von Interfaces in Java hat insbesondere wegen der klaren syntaktischen Trennung von Klassen viele Vorteile, u.A. eine größere konzeptuelle Klarheit in der Entwicklung sowie eine erhöhte Abstraktion und Modularisierung des Codes. Auf diese Vorteile wird im Abschnitt 2.5.1 (Ziele der interfacebasierten Programmierung) genauer eingegangen.

Zusammenfassend lässt sich sagen, dass auch in Java die Trennung von Implementation und Spezifikation noch immer eines der Hauptziele von Interfaces darstellt, jedoch die Verwendungsmöglichkeit von Interfaces als Typen mit den Möglichkeiten der Mehrfachimplementation und Substituierbarkeit – und alle daraus resultierenden Vorteile der Polymorphie – einen enormen Zuwachs an Nützlichkeit bedeutet.

Ein Beispiel für die Definition eines Interfaces mit implementierender Klasse und verschiedenen Nutzungsmöglichkeiten zeigt folgender Code (Abbildung 2-1).

```
public interface Drawable {
      public void drawOnScreen();
}

public interface Externalizable {
      public void storeToDisk(File file);
      public void loadFromDisk (File file);

public class SomePicture implements Drawable, Externalizable {
      public void drawOnScreen() {
            // implementation
      }
      public void storeToDisk(File file) {
            // implementation
      }
      public void loadFromDisk (File file) {
            // implementation
      }
}

// Ebenso z.B.

public class SomeChart implements Drawable, Externalizable {…}

// oder

public class SomeAudio implements Externalizable {…}

// Nutzung

public void store(Externalizable externalizeable, File f) {
      externalizeable.storetoDisk(f);
}

…
Drawable sp = new SomePicture();
Drawable sc = new SomeChart();

// Direkter Aufruf:

sp.drawOnScreen();
sc.drawOnScreen();

// Über geeignete Methoden:

SomeChart someChart = new SomeChart();
store(someChart, new File("…");
…
```

Abbildung 2-1: Beispiel eines Interfaces und dessen Implementation in Java

2.2 Interfaces in anderen objektorientierten Programmiersprachen

In diesem Abschnitt soll die Umsetzung des Interface-Konzepts in weiteren objektorientierten Programmiersprachen untersucht werden. Zunächst folgt eine chronologisch geordnete Übersicht über einige interessante Programmiersprachen; die meisten werden im Anschluss dann näher beleuchtet.

Die erste objektorientierte Programmiersprache war SIMULA, welche zwischen 1962 und 1967 in Oslo entwickelt wurde (Holmevik 1994). SIMULA wurde nie in großem Umfang eingesetzt, hatte jedoch Einfluss auf die später entwickelten Sprachen. Bezeichnenderweise enthält SIMULA weder Mehrfachvererbung noch Interfaces.

Die Sprache Smalltalk, Anfang 1970 entwickelt (Kay 1993), stand Mitte der 90er Jahre kurz vor dem Durchbruch im kommerziellen Sektor (Shan 1995) und wird von vielen als bestes Beispiel einer „echt“ objektorientierten Sprache gesehen (Zanden et al. 2001). Implizit hat sie Interfaces bereits seit ihren Anfängen enthalten.

Die Objektorientierung erreichte ihren Durchbruch mit der Integration objektorientierter Features in C durch Bjarne Stroustrup Anfang der 80er Jahre. Die zunächst als „C with classes“ bekannte Sprache wurde 1983 C++ getauft. C++ war die erste objektorientierte Sprache, die in großem Umfang kommerziell eingesetzt wurde. Interfaces können in C++ jedoch nur simuliert werden.

Objective-C ist eine andere objektorientierte Variante von C, im Gegensatz zu C++ mit einer an Smalltalk angelehnten Nachrichtensyntax. Die Sprache wurde Anfang der 80er Jahre von Brad Cox und der StepStone Corporation mit dem Ziel entwickelt, die Hauptfeatures von Smalltalk-80 in C zu integrieren. Die Betrachtung der Sprache Objective-C ist insbesondere deswegen interessant, weil Java semantisch stark durch Objective-C beeinflusst ist (Naughton 2003). Die Sprache enthält das Konzept der Protokolle, die mit Java-Interfaces direkt vergleichbar sind.

Die syntaktisch ebenfalls an C angelehnte Sprache C# wurde von Microsoft mit dem Ziel entwickelt, eine „intuitive und gleichzeitig mächtige Programmiersprache für die Windows-Programmierung“ zu schaffen. C# ist ein Gemisch aus unterschiedlichen Sprachen (C, C++, Java, VB) und wurde 1999 von Microsoft vorgestellt; eine erste Entwicklungsumgebung war 2001 verfügbar. Die Sprache enthält ein Interfacekonstrukt, welches bis auf wenige Besonderheiten mit dem von Java identisch ist.

2.2.1 Interfaces in Smalltalk

Die Interfaces eines Objekts sind in Smalltalk nur implizit enthalten, sie sind Teil der Klassendefinition. Sadeh und Ducasse schreiben hierzu: „[…] since Smalltalk does not have interfaces as first class objects, they cannot be conversed with, referred to, or reflected upon“ (Sadeh/Ducasse 2002).

Damit sind die in Smalltalk angebotenen Interfaces in direkter Konsequenz nicht mit den in Java verfügbaren Interfaces vergleichbar. Cook sieht jedoch die Möglichkeit, die impliziten Interfacedefinitionen durch eine entsprechende Analyse der Methoden in eine Interfacehierarchie zu überführen (Cook 1992). Eine solche Überführung kann jedoch lediglich der Information dienen.

In Smalltalk-80 existieren außerdem so genannte *Protokolle* – Gruppierungen von Methoden mit ähnlichem Verhalten. Diese sind jedoch ebenfalls nicht mit den Interfaces von Java zu

verwechseln: Bei Protokollen in Smalltalk wird nicht gefordert, dass eine Klasse bestimmte Methoden mit bestimmten Signaturen implementieren muss, während Interfaces in Java Methodenmengen exakt bis zur Signatur vorschreiben und der Compiler deren Implementation auch statisch überprüft.

Sadeh und Ducasse erkennen das Konzept der Interfaces als zentralen Punkt der objektorientierten Programmierung an und stellen eine Erweiterung von Smalltalk vor, die so genannten *SmallInterfaces*, welche das Konzept des Interfaces in Smalltalk integrieren. Interessant an diesem Ansatz ist die dynamische Natur der so neu hinzugefügten Interfaces. Anders als in Java können Beziehungen zwischen Interfaces und Klassen dynamisch durch das System erkannt und gezogen werden. Implementiert eine Klasse also alle Methoden eines Interfaces, ist es automatisch konform zu diesem und kann auch unter den entsprechenden Typ subsumiert werden.

Wie diese und andere Besonderheiten der *SmallInterfaces* zeigen, weist die Implementation dieser wesentliche Unterschiede zu der in Java verwendeten auf. Sie sollen hier daher nicht weiter vertieft werden.

2.2.2 Interfaces in Objective-C

Die Sprache Objective-C hatte großen Einfluss auf die Entwicklung von Java (Naughton 2003) und ist heute u.A. in Mac OS X integriert – das dort verfügbare objektorientierte Framework namens Cocoa (ein Nachkomme von NeXTStep) ist in Objective-C realisiert (Apple 2003). Das Prinzip der Java-Interfaces ist in Objective-C bereits angelegt – sie tragen hier den Namen Protokoll (*protocol*).

Die Objective-C-FAQ beschreibt Protokolle wie folgt:

> *In short, a protocol is a set of method declarations. A protocol can inherit from multiple other protocols, and a class can inherit multiple protocols. The protocol hierarchy is unrelated to the class hierarchy. The type of a variable can be 'id <P>' to denote that the class of the object pointed to does not matter, as long as the class implements all methods prescribed by the protocol P. Protocols provide `multiple inheritance of interface'. They are matched by Java interfaces and C++ signatures.* (OCFAQ)

Objective-C verfügt, wie Java, nicht über mehrfache Implementationsvererbung. Genau wie in Java erlaubt die Protokollhierarchie jedoch mehrfache Interfacevererbung. Die Protokollhierarchie ist dabei, wie oben beschrieben, unabhängig von der Klassenhierarchie.

Die Objective-C-FAQ gibt folgendes Beispiel für die Definition und Implementation eines Protokolls, welches die Fähigkeit einer Klasse zur Serialisierung beschreibt. Weitere Informationen finden sich in der Objective-C Sprachreferenz (Apple 2000).

```
@protocol Archiving
-read: (Stream *) stream;
-write: (Stream *) stream;
@end
```

Abbildung 2-2: Definition eines Protokolls in Objective-C

```
/* MyClass inherits from Object and conforms to the
   Archiving protocol.  */

@interface MyClass: Object <Archiving>
@end
```

Abbildung 2-3: Implementation eines Protokolls in Objective-C

2.2.3 Interfaces in C++

Ein allein stehendes Konzept des Interfaces ist in Standard-C++ nicht existent. Es gibt keinen besonderen Typ, der ein Interface repräsentiert; ebenso existiert keine eigene Vererbungshierarchie.

Im Gegensatz zu den anderen hier beleuchteten Sprachen unterstützt C++ jedoch die Mehrfachvererbung in der normalen Klassenhierarchie. Wie in andere Sprachen auch gibt es die Möglichkeit, abstrakte Klassen zu definieren, und über diese ist es auch möglich, eine Art Interface zu deklarieren – eine abstrakte Klasse, welche lediglich abstrakte Methoden ohne jegliche Implementation enthält.

In C++ besteht die Möglichkeit, Methoden einer Klasse als `virtual` zu definieren. Das Schlüsselwort `virtual` bewirkt, dass bei einer Ausführung der Methode effektiv die am tiefsten in der Hierarchie angesiedelte Methode zur Ausführung gebracht wird. Um die Methode – wie in Java – ohne Defaultimplementation zu definieren, muss die Methode mit einer Nullzuweisung (`=0`) als abstrakt definiert werden (ein Schlüsselwort `abstract` für Methoden existiert nicht).

Ein Teil des Beispiels aus Abschnitt 2.2 ist in C++ wie folgt realisierbar:

```
class Drawable {
public:
      virtual void drawOnScreen() = 0;
}

class SomePicture : public Drawable {
public:
      void drawOnScreen() {
            // implementation
      }
}
```

Abbildung 2-4: Vollständig abstrakte Klasse und deren Implementation in C++

Zusätzlich zu dieser Möglichkeit existiert eine Erweiterung der Sprache C++, die *C++ Signatures,* welche jedoch nur im GNU Compiler realisiert sind (Baumgartner/Russo 1995). Signatures weisen eine gewisse Ähnlichkeit zu Java-Interfaces auf (u.A. stehen Signatures in einer eigenen Hierarchie – unabhängig von der Implementationshierarchie), jedoch nutzt Java Namenskonformität (d.h. eine Klasse muss genutzte Interfaces über das `implements`-Schlüsselwort angeben), während bei dem hier gewählte Ansatz Strukturkonformität durch den Compiler erschlossen wird (d.h. ein Interface wird von einer Klasse implementiert, sobald die Klasse alle im Interface definierten Methoden enthält).

Weiter wird die Nutzung von Templates als Interfaces diskutiert. Bis auf die oben vorgestellte Nutzung abstrakter Klassen sind die genannten Verfahren jedoch nicht standardisiert und/oder umstritten.

2.2.4 Interfaces in C#

Die Interfaces von C# sind weitgehend zu den in Java definierten äquivalent. Sie erlauben jedoch eine größere Flexibilität bei der Implementation (Albahari 2000).

Wie in Java ist die Klassenhierarchie in C# von der Interfacehierarchie getrennt. Ein Interface kann mehrere andere Interfaces erweitern; ebenso kann ein Klasse mehrere Interfaces implementieren; wie in Java unterstützt die Interfacehierarchie folglich mehrfache Interfacevererbung. Interfaces enthalten keine Implementationen.

Das erste zuvor in Java präsentierte Interface sowie seine Implementation sieht in C# wie folgt aus (es ist lediglich das Schlüsselwort `implements` durch den Doppelpunkt ersetzt worden):

```
public interface Drawable {
      void drawOnScreen();
}

public class SomePicture : Drawable {
      public void drawOnScreen() {
            // implementation
      }
}
```

Abbildung 2-5: Interface und Implementation in C#

C# bietet jedoch, anders als Java, die Möglichkeit zur expliziten Implementation der definierten Methoden eines der implementierten Interfaces. Albahari gibt hierzu folgendes Beispiel (Albahari 2000):

```
public interface ITeller
{
    void Next ();
}

public interface IIterator
{
    void Next ();
}

public class Clark : ITeller, IIterator
{
    void ITeller.Next () {}
    void IIterator.Next () {}
}
```

Abbildung 2-6: Explizite Implementation der Interfacemethoden in C#

Die so definierten Methoden können über explizite Typecasts aufgerufen werden; eine weitere Möglichkeit wäre die Definition von Variablen mit den jeweiligen Interfaces. Ersteres ist wie folgt möglich:

```
Clark clark = new Clark ();
((ITeller)clark).Next();
```

Abbildung 2-7: Aufruf explizit implementierter Interfacemethoden in C#

2.3 Kategorisierung von Interfaces

Die Interfaces von Java können auf vielfältige Art und Weise genutzt werden; es ist daher hilfreich, verschiedene Nutzungsklassen zu identifizieren. Zuvor sollen jedoch einige Begriffe geklärt werden, die sich zur Beschreibung der Nutzungsklassen eignen.

Zunächst einmal existieren verschiedene Konzepte von Interfaces. In strukturierten Programmiersprachen ist meist die schlichte Separierung von Spezifikation und Implementierung gefragt, bei objektorientierten kommt die Polymorphie durch die mögliche Typisierung hinzu. Damit lassen sich also zwei grundlegende Konzepte unterscheiden:

- *Interfaces zur Trennung von Implementation und Spezifikation*. Hierbei werden meist Teile des Implementationscodes – z.B. die Signatur einer Funktion – in einem separaten Bereich der Datei oder einer insgesamt separaten Datei nochmals wiederholt. Eine solche Funktionalität kann z.T. durch den Compiler bereitgestellt werden.
- *Interfaces als Typen*. Derartige Interfaces enthalten die Funktion der zuvor genannten Interfaces, sind jedoch als echte Typen in der Programmiersprache realisiert. Dadurch, dass die verschiedenen Klassen desselben Programms dasselbe Interface implementieren können (und somit demselben Typ entsprechen) wird die Entscheidung, welche Implementation verwendet wird, von der Übersetzungs- zur Laufzeit verlagert.

Die folgende Diskussion dreht sich um das zweite – in Java realisierte – Konzept der Interfaces. Daher sind die folgenden Definitionen nur für Interfaces als Typen relevant.

Steimann unterscheidet folgende Eigenschaften von Interfaces (Steimann 2003):

- *Partielles/totales Interface*. Unter einem partiellen Interface wird ein Interface einer Klasse verstanden, welches nicht alle (öffentlichen) Features der Klasse enthält, sondern nur eine Teilmenge dieser. Das Interface stellt also eine partielle Spezifikation der Klasse dar. Ein totales Interface enthält dagegen alle (öffentlichen) Features der implementierenden Klasse. Für die Unterscheidung zwischen partiellen und totalen Interfaces ist allein die Anbieterperspektive, also die implementierende Klasse, relevant. Ein Interface kann gleichzeitig partiell und total sein, wenn es von mehreren Klassen implementiert wird.
- *Kontextspezifisches/kontextübergreifendes Interface*. Ein Interface wird als kontextspezifisch bezeichnet, wenn das Interface einer implementierenden Klasse genau diejenigen Methoden enthält, die von einem Klienten der Klasse in einem gewissen Kontext[3] benötigt werden. Ein kontextübergreifendes Interface ist dagegen vom Kontext unabhängig und enthält Funktionen für alle möglichen Kontexte (die meisten Interfaces werden zwischen den Extremen liegen, siehe auch Kapitel 4). Die Unterscheidung zwischen kontextspezifischen und kontextübergreifenden Interfaces hängt also von der Nutzung der Klassen ab (Nutzerperspektive).

[3] Das Konzept des Kontexts wird unten näher erläutert.

- *Aktives/Passives Interface*. Wird ein Interface von einer Klasse implementiert, damit die Klasse von weiteren Klassen in einer Art und Weise eingebunden werden kann, von der sie selbst profitiert (z.B. `Serializable`, `Cloneable`, `Comparable` in der Java API), handelt es sich um ein passives Interface. Wird das Interface dagegen von einer Klasse implementiert, die durch dieses Interface Dienste anbietet, von denen andere Objekte abhängen, handelt es sich um ein aktives Interface.

Auf der Basis der obigen Eigenschaften können die folgenden vier Interfacetypen unterschieden werden (Steimann et al. 2003):

2.3.1 Family Interfaces

Family Interfaces werden, wie der Name bereits andeutet, für eine ganze Familie von Klassen definiert. Java erlaubt bei Klassen keine Mehrfachvererbung; diese kann durch geeignete Interfacehierarchien jedoch simuliert werden. Entsprechende (Super-)Interfaces, zu welchen – als Typ betrachtet – eine ganze Familie von Klassen zuweisungskompatibel ist, werden als Family Interfaces bezeichnet. Sie sind i.d.R. kontextunabhängig. Ein Beispiel hierfür ist das Interface `Set` der Java API.

Ebenso schaffen Family Interfaces Vergleichsmöglichkeiten zwischen Klassen: Gemeinsame Interfaces von zwei Klassen geben deren Gleichheit an, was eine bestimmte, aber kontextunabhängige Funktionalität betrifft. Charakteristisch für Family Interfaces ist die Austauschbarkeit der implementierenden Klassen; eine Tatsache, die bei Plug-In-Interfaces (s.u.) nicht gegeben ist. Beispiel sind die das Interface `Set` implementierenden Klassen `HashSet` und `TreeSet`.

2.3.2 Client/Server-Interfaces

In der objektorientierten Programmierung findet sich vielfach eine Abhängigkeit von Objekten untereinander: Ein Objekt – der Client – benötigt die Funktionalität eines weiteren – des Servers. Ein vom Server implementiertes Interface wird als *Client/Server-Interface* bezeichnet – es definiert die Funktionalität, auf welche sich Clients verlassen können. Client/Server-Interfaces sind kontextspezifisch – d.h. sie bieten genau die Funktionen an, die ein Client aus seinem Kontext heraus benötigt. Ein kontextübergreifendes Interface wäre vom Client unabhängig und damit ein reines Server-Interface.

Client/Server-Interfaces sind i.d.R. partiell – d.h. sie definieren nur einen Teil der vom Server angebotenen Funktionen; pro Kontext können dies durchaus andere Funktionen sein; wobei Überlappungen zulässig sind. Client/Server-Interfaces zeigen eine aktive Rolle der implementierenden Klasse an – die Rolle, welche der Server in der Kollaboration spielt.

2.3.3 Tagging Interfaces

Tagging Interfaces (to tag – markieren) erlauben es, die Objekte bestimmter Klassen zu markieren/zu klassifizieren. Eine solche Markierung zeigt an, dass das Objekt in einer gewissen Weise genutzt werden kann, wobei keinerlei Methoden für diese Nutzung in der Klasse vorhanden sein müssen. Tagging Interfaces enthalten folgerichtig keinerlei Spezifikation für Methoden, sie stellen eine Art passives Filter dar.

Ein bekanntes Beispiel für ein Tagging Interface ist die Klasse `Serializable` der Java API, welche anzeigt, dass die entsprechende Klasse serialisiert werden kann. Das „tagging“ er-

laubt die Verwendung von Instanzen aller das Interface `Serializable` implementierenden Klassen unter diesem Typ, ohne dass die konkreten Klassen bekannt sein müssen.

2.3.4 Plug-In-Interfaces

Plug-In-Interfaces stellen eine erweiterte Form von Tagging Interfaces dar. Die Markierung gibt bei letzteren an, dass die implementierenden Klassen – rein passiv – in einer gewissen Weise genutzt werden können. Plug-In-Interfaces haben denselben passiven Charakter, jedoch werden hier auch Methoden vorgegeben, welche von den implementierenden Klassen zur Unterstützung der Nutzung implementiert werden müssen.

Der Name „plug-in“ stammt aus der Verwendung der Klassen, die diese Interfaces implementieren – die Funktionalität einer gewissen Klasse wird auf die das Plug-In-Interface implementierende Klasse angewendet. Ein Beispiel ist die Klasse `Collections` der Java-API – sie bietet die Möglichkeit, auf Listen mit Objekten eine Sortierung durchzuführen. Die Objekte der Liste müssen dazu selbst lediglich die Fähigkeit besitzen zu bestimmen, ob sie größer, kleiner, oder gleich einem anderen Objekt sind. Diese Fähigkeit wird durch die Implementation des `Comparable`-Interfaces angezeigt.

Klassen, welche Tagging- oder Plug-In-Interfaces implementieren, werden also von anderen Klassen zum eigenen Nutzen verwendet; sie werden z.B. sortiert oder gedruckt. Die Client/Server-Rollen sind hier also vertauscht: Obwohl eine andere Klasse als Client die implementierende Klasse nutzt, profitiert letztere – der Server – von dieser Nutzung. Ein Tagging- oder Plug-In-Interface kann also auch als Server/Client-Interface bezeichnet werden; es wird passiv genutzt.

Folgende Tabelle verdeutlicht die Attribute der jeweiligen Interfacetypen:

Tabelle 2-1: Interfacetypen

	Client/Server		**Server/Client**	
	Family	**Client/Server**	**Tagging**	**Plug-In**
Kontext	Übergreifend	Abhängig	Übergreifend	Abhängig/Übergr.
Genus	Aktiv/Passiv	Aktiv	Passiv	Passiv
Umfang	Partiell/Total	Partiell/Total	Leer	Partiell

2.4 Konzeptualisierung von Interfaces als Rollen

Die obigen Abschnitte haben die technische Seite der Nutzung von Interfaces beleuchtet – und damit die Phase der Implementation im objektorientierten Design. Eine der Stärken des objektorientierten Paradigmas ist jedoch der nahtlose Übergang zwischen Analyse, Design und Implementierung, ein Prozess, in dem Interfaces in den ersten Phasen bisher nur Randplätze eingenommen haben.

Um sicherzustellen, dass Interfaces bereits in den frühen Designphasen die ihnen gebührende Beachtung erlangen, ist eine sinnvolle konzeptuelle Entsprechung des Interfaces in der realen Welt nötig. Eine solche Entsprechung ist das Konzept der Rolle (Steimann 2000, Steimann 2001).

Das Konzept der Rolle steht, wie in Steimann 2000 gezeigt wird, gleichberechtigt neben dem der Klasse. Tatsächlich sind viele der in heutiger Anwendungssoftware modellierten Klassen eigentlich Rollen: Das beliebteste Beispiel sind hierbei Klassen wie `Kunde`, `Student` oder

`Angestellter`, welche eigentlich Rollen einer Klasse `Person` sind. Charakteristisch für Rollen sind dabei die folgenden beiden Eigenschaften:

- Eine Rolle kann dynamisch von einer Instanz einer Klasse angenommen und auch wieder abgelegt werden, während die Klasse der Instanz nicht geändert werden kann.
- Eine Rolle ist stets in einer Beziehung definiert, während eine Klasse alleine stehen kann.

Das Rollenkonzept ist, wie Steimann ausführt, sehr alt und stammt aus der Welt des Theaters; es wird heute jedoch auch in ganz anderen Wissenschaften wie der Soziologie oder der Linguistik verwendet (Steimann 2000). Über alle Verwendungen hinweg bleibt die Charakteristik der Rolle konstant, ein gewisses Verhalten oder Protokoll zu definieren, welches in einem bestimmten Kontext benötigt wird – und zwar unabhängig davon, von wem dieses Verhalten letztendlich gezeigt wird.

Steimann nutzt u.A. den Rollenbegriff von Guarino, welcher zwischen dem Konzept der natürlichen Typen (als Klassen modelliert) und dem der Rollen (als Interfaces modelliert) mit den Begriffen „fundiert“ und „semantisch rigide“ unterscheidet (Steimann 2000, S. 10):

- Ein Konzept gilt als *fundiert*, wenn die Existenz seiner Instanzen von anderen Instanzen abhängt, also stets eine Beziehung vorhanden sein muss (wobei die Teil-Ganzes-Beziehung explizit ausgeschlossen ist).
- Ein Konzept gilt als *semantisch rigide*, wenn es die Identität seiner Instanzen bestimmt, d.h. die Existenz ist von Beziehungen unabhängig, eine Instanz kann das Konzept aber niemals verlassen, ohne seine Identität zu verlieren.

Diese Definitionen passen erstaunlich gut auf Klassen und Interfaces der Sprache Java: Interfaces sind fundierte, aber nicht semantisch rigide Konzepte, während Klassen semantisch rigide, aber nicht fundiert sind.

Die so entwickelte Konzeptualisierung von Interfaces als Rollen hilft dem Entwickler, bereits während des objektorientierten Designs Interfaces zu identifizieren und in das System einzubringen. Steimann zeigt eine Umsetzung dieses Gedankens in UML (Steimann 2001).

2.5 Kontext und Kontextspezifität

2.5.1 Methodenteilmengen und Kontexte

In Abschnitt 2.3 ist bereits von Kontextspezifität eines Interfaces gesprochen worden. Dahinter steht das Konzept des Kontexts, welches nicht nur für Interfaces, sondern auch für die Nutzung von Klassen im Allgemeinen relevant ist. Um dieses Konzept zu erläutern, soll zunächst der Begriff der *Methodenteilmenge einer Klasse* definiert werden:

- Bei einer Methodenteilmenge handelt es sich um eine (nicht unbedingt echte) Teilmenge von Methoden einer bestimmten Klasse.
- Eine Klasse verfügt über 2^n verschiedene Methodenteilmengen, wobei n die Anzahl der Methoden darstellt.

Derartige Methodenteilmengen existieren unabhängig von einer Nutzung der Methoden in einem Klienten der Klasse. Das folgende Beispiel gibt einen Überblick über die Methodenteilmengen der Klasse `RMFile` (Real Media File).

```
public class RMFile implements AudioFile {
   public void play() {}
   public void stop() {}

   public Options getOptions() {}
   public void setOptions(Options option) {}
}
```

Abbildung 2-8: Die Klasse RMFile

Die zu dieser Klasse gehörigen Methodenteilmengen lassen sich graphisch in einem Verbund darstellen. Abbildung 2-9 zeigt diesen Verbund:

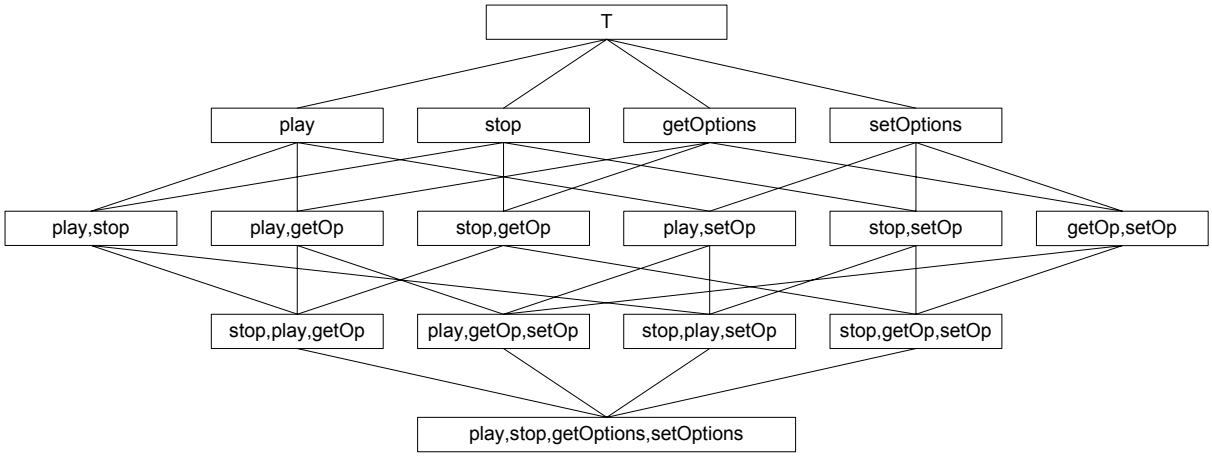


Abbildung 2-9: Mögliche Methodenteilmengen für RMFile

Am Fuß des Verbundes befinden sich alle öffentlichen Methoden der Klasse; am Kopf des Verbundes steht die leere Menge. Jeder Knoten im Verbund stellt eine Methodenteilmenge dar.

Verwendung findet eine Methodenteilmenge durch die Deklaration einer Variablen mit der Klasse (oder einem Interface der Klasse) und Nutzung genau dieser Methodenteilmenge auf dieser Variablen. Die Variable bildet dann einen *Nutzungskontext* (auch kurz *Kontext* genannt) dieser Methodenteilmenge der Klasse. Der Kontext stellt damit die Verbindung zwischen der Variable und der genutzte Methodenteilmenge dar. Von der Variablen wird dann gesagt, dass sie diesen Kontext *aufspannt*.

Eine Variable kann immer nur den Kontext für genau eine Methodenteilmenge vorgeben; auf der anderen Seite kann eine Methodenteilmenge jedoch im Kontext mehrerer Variablen genutzt werden.

2.5.2 Interfaces für Kontexte

Im Abschnitt 2.3 ist u.A. zwischen kontextspezifischen und kontextübergreifenden Interfaces unterschieden worden; es wurde verdeutlicht, dass diese Unterscheidung von der Nutzerseite abhängt, also nach der obigen Definition von den Variablen, welche mit den Interfaces typisiert sind, und den in den Kontexten der Variablen verwendeten Methodenteilmengen.

Die Kontextspezifität eines Interfaces hängt immer von der momentan betrachteten Variablen ab. Stimmt die im Kontext dieser Variablen verwendete Methodenteilmenge mit der im Interface definierten Methodenteilmenge der Klasse überein, so ist das Interface kontextspe-

zifisch für diesen Kontext. Genau genommen ist ein Interface spezifisch für eine Methodenteilmenge, diese wird jedoch erst durch den Kontext einer Variablen im Programm verfügbar gemacht und mit dem Interface in Verbindung gebracht. Ein Interface, welches für jede Variable, in welcher es deklariert ist, kontextspezifisch ist, ist insgesamt ein rein kontextspezifisch verwendetes Interface.

Wird das Interface dagegen von mehreren Variablen als Typ genutzt, die Kontexte mit unterschiedlichen Methodenteilmengen aufspannen – dies ist möglich, wenn weitere Teilmengen der im Interface enthaltenen Methodenteilmenge existieren – so handelt es sich um ein kontextübergreifendes Interface. In der Regel werden Interfaces für einige Kontexte spezifisch sein, für andere dagegen nicht.

Ein Beispiel für zwei unterschiedliche Nutzungskontexte zeigt folgender Code. Ein Teil eines Programms nutzt hier die Wiedergabefunktionen, ein anderer Teil die Optionsfunktionen von `RMFile`. Als Typ der Variablen wird dabei das Interface `AudioFile` eingesetzt, welches alle Methoden von RMFile bereits enthält (siehe auch unten).

```
public void usesPlayFunctions(AudioFile file)
{
      file.play();
      …
      file.stop();
}
…
public void usesOptionFunctions(AudioFile file)
{
      … = file.getOptions()
      …
      file.setOptions(…)
}
```

Abbildung 2-10: Beispiel für Kontexte

Die Deklaration der Variablen `file` als Parameter spannen je einen neuen Nutzungskontext (unter Verwendung des Typs `AudioFile`) von `RMFile` auf. In jedem Kontext wird jedoch nur auf eine Teilmenge der Funktionen von `AudioFile` zugegriffen – einmal auf die Funktionen zum Abspielen, einmal auf die für Optionen.

Sinnvoll wäre es hier, ein eigenes Interface für die Abspielfunktionen und eines für die Optionsfunktionen einzuführen. Im einen Fall wären dies also die Methoden `play` und `stop`, im anderen Fall `getOptions` und `setOptions`.

Im Verbund (Abbildung 2-11) ist diese Situation anschaulich sofort klar: Es existiert ein Interface `AudioFile`, welches die Methoden `play`, `stop`, `getOptions` und `setOptions` abdeckt, jedoch existiert kein derartiger Nutzungskontext. Stattdessen existieren zwei andere Nutzungskontexte, die sich im Verbund zwei Stufen höher befinden. Ideal wäre also die Auflösung des Interfaces `AudioFile` zugunsten von zwei neuen, welche direkt auf die jeweiligen Nutzungskontexte passen/abgestimmt sind.

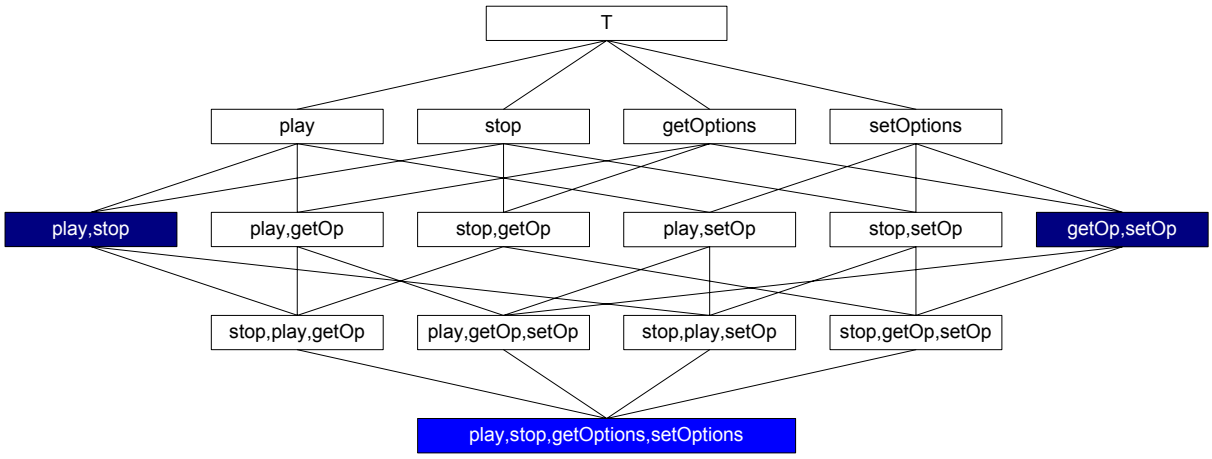


Abbildung 2-11: Kontextspezifische Interfaces für RMFile

Von der Unterscheidung zwischen kontextspezifischen und kontextübergreifenden Interfaces unabhängig ist die Trennung zwischen partiellen und totalen Interfaces; hier ist allein die Anbieterseite relevant. Ein totales Interface enthält immer genau die Methoden der Klasse (Fuß des Verbundes); ein partielles Interface enthält dagegen nur eine echte Teilmenge der Methoden der Klasse (alle anderen Knoten im Verbund).

Das Interface `AudioFile` ist ein totales Interface der Klasse `RMFile`:

```
public interface AudioFile {

      public void play();
      public void stop();

      public Options getOptions();
      public void setOptions();

}
```

Abbildung 2-12: Das Interface AudioFile

Zwischen den Eigenschaften kontextspezifisch/kontextübergreifend und partiell/total besteht dennoch ein gewisser Zusammenhang. In der Regel sind totale Interfaces kontextübergreifend, da sie sehr viele Methoden anbieten und daher von vielen Variablen genutzt werden können; kontextspezifische Interface sind dagegen i.d.R. partiell, da sie speziell auf eine Gruppe von Variablen und deren benötigte Methoden zugeschnitten sind. Die jeweiligen Umkehrschlüsse gelten jedoch meist nicht.

Interfaces, die sowohl partiell als auch kontextspezifisch sind, stellen einen extremen Fall dar: Alle mit einem solchen Interface typisierten Variablen müssen denselben Kontext aufspannen, und dieser muss mit dem Interface exakt übereinstimmen. Den Gegenpol bilden totale/kontextübergreifende Interfaces, in welchen keinerlei Spezifität enthalten ist. Wie Abbildung 2.11 verdeutlicht, liegen viele Interfaces jedoch auch zwischen den Extremen.

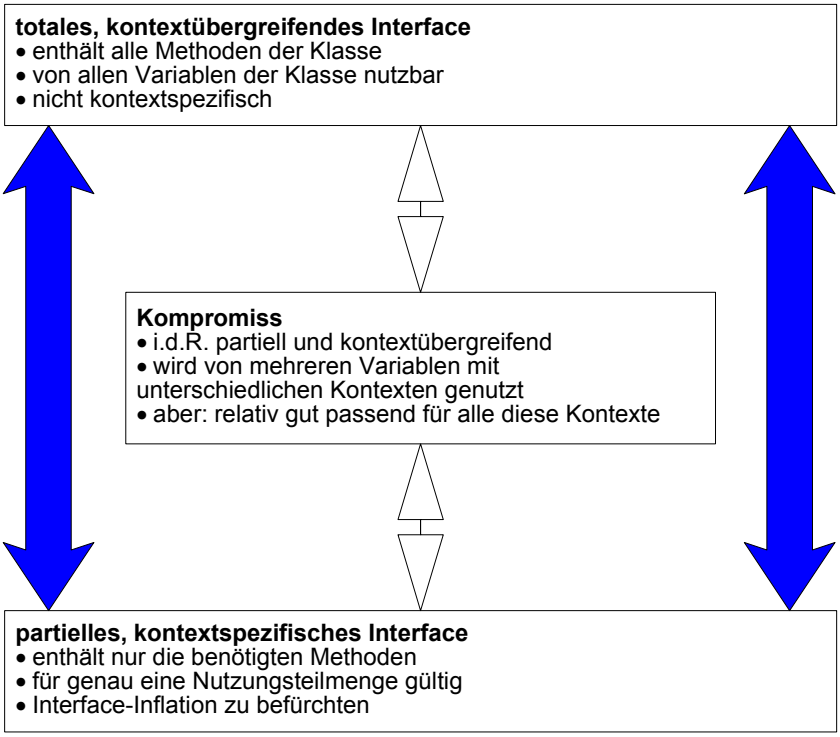


Abbildung 2-13: Totale/kontextübergreifende vs. partielle/kontextspezifische Interfaces

2.5.3 Grenzen der Kontextspezifität

Wie oben beschrieben werden in dieser Arbeit Kontexte betrachtet, die durch Variablendeklarationen aufgespannt werden. Zur Bestimmung des idealen Typs einer Variable werden die auf der Variable aufgerufenen Methoden des Typs betrachtet, welche in ihrer Gesamtheit den Kontext und das ideale Interface für diesen Kontext zugleich bestimmen.

Variablen stellen jedoch nicht die einzigen Punkte im Programm dar, an welchen Methoden aufgerufen werden können. So ist es möglich, mehrere Methodenaufrufe direkt hintereinanderzuschalten:

```
object1.getSomeObject2().doSomething()
```

Diese Form der Verkettung ist bereits kritisiert worden (vgl. Kapitel 3, Abschnitt 3.4.3, im so genannten *Law of Demeter*), ist jedoch häufig in Verwendung. Ein derartiger Aufruf bildet einen minimalen Kontext: Es wird lediglich eine einzige Methode aufgerufen; ein echter Kontext im Sinne einer Variablen existiert nicht. Aus diesem Grund werden derartige Methodenaufrufe in dieser Arbeit nicht weiter untersucht.

Die Einführung eines idealen Interfaces für jeden Kontext ist nicht immer machbar oder auch nur anstrebenswert: Bei Klassen mit vielen verschiedenen Kontexten kann die Zahl der Interfaces schnell explodieren. Zudem werden Variablen i.d.R. im Programm nicht nur an einer Stelle verwendet, sondern weitergeben, wodurch mehrere, zusammenhängende Kontexte entstehen, an deren Nahtstellen Kontextwechsel (*context switches*) auftreten. Die Betrachtung von Kontexten und kontextspezifischen Interfaces erfolgt in dieser Arbeit bewusst unter Auslassung der durch Kontextwechsel entstehenden Probleme; mit den durch diese Arbeit zur Verfügung gestellten Hilfsmitteln können diese jedoch in der weiteren Forschung genau untersucht werden (siehe Kapitel 6, Ausblick).

An dieser Stelle sei jedoch noch darauf hingewiesen, dass bei der Betrachtung von Kontextwechseln erneut das Konzept der Rolle ins Spiel kommt. Bislang wurde die technische Seite

der Nutzung von Klassen beleuchtet – die Nutzung von Methoden in einem gewissen Bereich des Codes; eine Nutzung, die durch automatisierte Tools erkannt werden kann. Daneben ist es jedoch auch möglich, dass durch einen oder mehrere Kontexte bestimmte Features einer Klasse in unmittelbarem konzeptuellem Zusammenhang stehen. Dieser Zusammenhang wird durch die Identifikation einer Gruppe von Features als Rolle ausgedrückt und kann nur durch einen Menschen erkannt werden. Eine konzeptuell adäquate Rolle als Interface implementiert sollte stets Vorrang vor rein technisch kontextspezifischen Interfaces haben.

2.6 Ziele der Verwendung von Interfaces

Im Folgenden sollen zwei Einsatzzwecke von Interfaces als erstrebenswerte Ziele objektorientierter Programmierung erörtert werden. Sie bilden die Grundlage des Interface- oder auch Rollenparadigmas – des *interface-based programming* – für diese Arbeit. Die Ziele dienen zudem als Ausgangspunkt für die weitere Betrachtung von Metriken und Metrik-Suiten, ausgehend von einer Bewertung der Literatur in Kapitel 3 über die Definition einer neuen Metrik-Suite in Kapitel 4 bis hin zur abschließenden Bewertung der Ergebnisse am Ende der Arbeit.

Als Grundlage für den Einsatz von Refactorings werden vielfach *bad smells*, „schlechte Gerüche“ im Code genannt (Fowler 2000). Um den Entwickler bei der Erkennung solcher *bad smells* zu unterstützen, können Metriken eingesetzt werden; dies wird von Simon et al. als „metrics based refactoring“ bezeichnet (Simon et al. 2001). Zu den hier definierten Zielen der Nutzung von Interfaces werden aus diesem Grunde auch *bad smells* identifiziert, welche durch Metriken aufgedeckt werden können. Neben der Auffindung von *bad smells* bieten Metriken jedoch auch eine Möglichkeit zur Evaluierung des Erfolges der Refactorings, indem sie einen Vorher-Nachher-Vergleich in Zahlen und Diagrammen erlauben.

Grundlegend für jede Anwendungsentwicklung ist der Einsatz von Interfaces zur vollständigen Entkopplung von Klassen; dieses Ziel wird im nächsten Abschnitt erläutert. Aufbauend auf diesem kann ein weiterführendes Ziel definiert werden, welches die intensive Nutzung von partiellen bzw. kontextspezifischen Interfaces sowie die Konzeptualisierung von Interfaces als Rollen wie oben beschrieben vorsieht (Abschnitt 2.6.2).

2.6.1 Einsatz von Interfaces zur vollständigen Entkopplung der Klassen

Zum Einsatz von Interfaces zur Entkopplung der Spezifikation und Implementierung von Klassen eines Designs äußern sich Gamma et al. (Gamma et al. 1995 S.24-25). Dieses Werk stammt noch aus der Prä-Java-Zeit und bezieht sich auf abstrakte Klassen in C++. Hier wird die Möglichkeit, Vererbung so zu verwenden, dass eine Familie von Objekten mit identischen Schnittstellen entsteht, als besonders wichtig herausgehoben. Eine Verwendung von abstrakten Klassen wie hier angeführt entspricht exakt der Definition von Interfaces in Java. Gamma et al. sehen folgende Vorteile in der Nutzung von Objekten über ihre Interfaces:

- *Klienten wissen nichts über die Klassen der Objekte, die sie benutzen, solange diese Objekte der von Klienten erwarteten Schnittstelle genügen.*
- *Klienten wissen nichts von den Klassen, die diese Objekte implementieren. Klienten kennen nur die abstrakte(n) Klasse(n), die ihre Schnittstellen definieren.* (Gamma et al. 1995 S. 25, Deutsche Ausgabe).

Interfaces tragen so in großem Maße zur Entkopplung von Subkomponenten innerhalb eines Softwaresystems bei. Gamma et al. erheben daher das Prinzip, interface-basiert zu programmieren, sogar zur Leitlinie für wiederverwendbares objektorientierten Design:

Program to an interface, not an implementation
(Gamma et al. 1995, S.24)

Viele der in Gamma et al. 1995 vorgestellten Patterns basieren auf diesem Prinzip; die dazugehörigen *bad smells* sind eindeutig: Schlecht riechen…

- …Klassen, die keine Interfaces implementieren[4]. Sie sollten auf mögliche Interfaces hin überprüft werden und um Interfaces erweitert werden.
- …mit Klassen typisierte Variablen[5]. Sie verstoßen gegen das Prinzip der Entkopplung und sollten durch interface-typisierte Variablen ersetzt werden.

Das erste Ziel zum Einsatz von Interfaces bezieht sich also rein auf die Verwendung von Interfaces zur vollständigen Entkopplung von Klassen, d.h. wo immer möglich soll die Spezifikation der Features[6] einer Klasse in ein Interface ausgelagert und jenes als Typ von Variablen verwendet werden. Dabei ist es unerheblich wie viele Interfaces verwendet werden – jeweils ein Interface pro Klasse ist bereits ausreichend, um eine Entkopplung zu erreichen.

Die beiden genannten *bad smells* lassen erkennen, dass sowohl die Anbieter- als auch die Nutzerperspektive relevant ist. Wird ein Interface zwar implementiert, jedoch nie verwendet, ist die Entkopplung nur aus Anbieter-, jedoch nicht aus Nutzersicht erreicht.

2.6.2 Einsatz von Interfaces als partielle Spezifikationen

Wie bereits in Abschnitt 2.3 erwähnt, können Interfaces nicht nur als vollständige (totale) Spezifikationen *aller* öffentlichen Features einer Klasse eingesetzt werden, sondern auch als partielle Spezifikationen *eines Teils* der öffentlichen Features einer Klasse. Ein partielles Interface ist dabei definiert als ein Interface, welches nur eine echte Teilmenge der vollständigen Menge öffentlicher Methoden einer Klasse abdeckt.

Das erstgenannte Einsatzziel von Interfaces zur vollständigen Entkopplung von Klassen kann durchaus als Pflicht für jede Anwendungsentwicklung verstanden werden. Eine Verwendung von partiellen Interfaces erfordert dagegen eine sorgfältige Abwägung und Fall-zu-Fall-Entscheidungen. Der Einsatz von Interfaces als partielle Spezifikation ist demnach als Aufsatz, als „Kür“ zur vollständigen Entkopplung zu sehen; er setzt die Verwendung von Interfaces bereits voraus.

Das Konzept des Kontexts und des kontextspezifischen Interfaces ist in Abschnitt 2.5 bereits näher beleuchtet worden. Das hier definierte zweite Ziel bezieht sich auf diese Konzepte; es sollen – wo immer möglich – kontextspezifische Interfaces verwendet werden. Um einer Interfaceexplosion vorzubeugen und auch Interfaces zu erlauben, welche mehrere Kontexte in bestmöglicher minimaler Form abdecken oder einer konzeptuellen Rolle der Klasse entsprechen, wird hier jedoch im Allgemeinen von partiellen Interfaces gesprochen.

[4] Selbst ein einzelnes Interface für eine Klasse, welches alle öffentlichen Features abdeckt, ist sinnvoll, da dieses Interface auch als Platzhalter für ggf. zukünftig implementierende Klassen fungiert. Dies ist insbesondere bei der Entwicklung von Frameworks von großer Bedeutung.
[5] Gewisse Attribute von Klassen, die Qualitäten beschreiben oder Teil-Ganzes-Beziehungen modellieren, sind bei Betrachtung von Interfaces als Rollen von dieser Regelung ausgeschlossen. Näheres findet sich in Steimann et al. 2003.
[6] Features eines Java-Interfaces sind neben abstrakten Methoden auch (finale) Variablen sowie (innere) Klassen und (innere) Interfaces. Im Sinne einer Protokollspezifikation sind jedoch lediglich die Methoden relevant, da sie das Verhalten vorgeben.

Zu derartigen Interfaces – bzw. der Nutzung einer Teilmenge der Features einer Klasse – äußert sich auch Fowler bei der Vorstellung des Refactorings „Extract Interface“ (Fowler 2000 S.341-342). Er empfiehlt die Einführung von eigenen Interfaces für Bereiche des Codes, in welchen nur Teilmengen der Methoden einer Klasse genutzt werden.

Technisch bieten partielle Interfaces folgende Vorteile:

- Es ist dadurch eindeutig, welche Teile des Codes welche Aspekte der Klassen nutzen – besonders dann, wenn den Interfaces sinnvolle Namen gegeben werden.
- Bei (zu) großen Interfaces kann das Problem auftreten, dass implementierende Klassen nur noch einem Teil des definierten Protokolls tatsächlich einen Sinn geben.
- Bei der Nutzung partieller Interfaces in Variablendeklarationen müssen neue (alternative) Klassen nur die jeweils benötigten (Teil-)Interfaces implementieren und können dann direkt genutzt werden.

Zusammenfassend gesagt sollten Interfaces also partiell, möglichst kontextspezifisch ausgelegt sein. Durch die Einführung solcher minimaler Interfaces werden die implementierenden Klassen nicht nur minimal eingeschränkt und maximal austauschbar, was zu einer erhöhten Anzahl potentieller implementierender Klassen führt; es gibt auch den genannten konzeptuellen Vorteil: so entstandene Interfaces können als Rolle der implementierenden Klasse angesehen werden (vgl. Steimann et al. 2003).

Leider findet man in heutigem Code nur wenige kontextspezifische Interfaces, sondern vielmehr

- überhaupt keine Interfaces, sondern die Nutzung von Klassen als Typen von Variablen.
- nicht-kontextspezifische Interfaces, welche ganze Hierarchien von Klassen abdecken, von welchen in vielen Kontexten aber nur wenige Features verwendet werden (Family Interfaces).

Bad smells für diese Art der Verwendung von Interfaces beziehen sich also direkt auf die Verwendung von Objekten einer Klasse: Sie entstehen, wo

- Variablen mit Klassen deklariert sind und im Kontext nur auf Teile der Klassenspezifikation zugegriffen wird.
- Variablen zwar mit Interfaces deklariert sind, jedoch im Kontext nur auf Teile dieser Interfaces zugegriffen wird (es sei denn, das Interface entspricht einer adäquaten Rolle oder die Klasse besitzt zu viele Kontexte).

Die hier definierten Ziele werden in Kapitel 3 und 4 aufgegriffen, um sinnvolle Metriken zur Messung der Erfüllung der Ziele aufzufinden bzw. zu definieren.

3 Metriken

Zwei Zitate sollen in die Domäne der Metriken einleiten:

You cannot control what you cannot measure.
(DeMarco 1982)

Beware! The domain of metrics is deep and muddy waters.
(Henderson-Sellers 1996)

Das erste Zitat von Tom DeMarco ist eine der Motivationen für die Nutzung von Metriken und Metrikprogrammen. Softwaremetriken übernehmen die Vermessung von Software – unter Vermessung versteht man, auch auf Software angewandt, die Zuordnung von Zahlen zu Attributen von Objekten der realen Welt. Erst die Möglichkeit, Software in Zahlen auszudrücken ermöglicht eine Kontrolle des Codes – sei es allein stehend oder in vergleichender Weise.

Mit dem zweiten Zitat von Brian Henderson-Sellers wird die Problematik der Vermessung von Software – des „Software Measurements" – ausgedrückt: ein Mangel an anerkannten, formalen Vorgehensweisen, der Kampf gegen das negative Image von Metriken und eine Flut von Einzelmetriken ohne zugrunde liegendes Konzept.

Die Wissenschaft des Software Measurements ist noch jung – die ersten Softwaremetriken wurden in den 60er Jahren (des letzten Jahrhunderts) eingesetzt, erste Bücher folgten in den 70ern. Heute lassen sich zwei Hauptziele des Software Measurements über die Literatur hinweg identifizieren:

1. Verbesserung (durch Messung) der *Qualität von Softwareprodukten* und
2. Verbesserung der *Steuerung/Kontrolle des Prozesses der Softwareentwicklung* durch Quantifizierung des Prozesses.

In den neueren Veröffentlichungen wie Henderson-Sellers 1996, Zuse 1997 oder De Champeaux 1997 wird von den Autoren deutlich gemacht, dass der Bereich des Software Measurements noch immer in der Grundlagenforschung steckt. Alle Autoren der erschienenen Metrik-Bücher sind jedoch vereint in dem Streben nach erweitertem Einsatz von Metrikprogrammen sowie in deren Formalisierung.

Dieses Kapitel gibt einen Überblick über Softwaremetriken. Dabei liegt der Fokus auf der Vermessung objektorientierter Software; traditionelle Metriken werden jedoch ebenfalls kurz beleuchtet. Das Kapitel ist wie folgt aufgebaut: Zunächst leitet eine Motivation für den Einsatz von Metriken in die Thematik ein; ein Kapitel zu den Anfängen des Software Measurements schließt sich an, gefolgt von Kategorisierungsmöglichkeiten für Metriken.

In dieser Arbeit sollen Metriken für die Analyse von Interfaces, also eines Konzepts der Objektorientierung, vorgestellt werden. Aus diesem Grund schließt sich an den oben genannten, eher allgemeinen Teil über Metriken eine genauere Betrachtung von objektorientierten Metriken und ausgewählten Metrik-Suiten an. Abschließend werden Metriken und Metrik-Suiten identifiziert, die zur Analyse von Interfaces geeignet sind.

Eine Betrachtung der formalen Evaluation von Metriken schließt das Kapitel ab.

3.1 Warum Metriken?

Henderson-Sellers listet folgende Vorteile der Implementation eines Metrikprogramms in der Entwicklung objektorientierter Systeme auf:

- *A quantitative understanding of both the architecture and the detailed design of the system so that successes can be built on and repeats of failures avoided. Without metrics, risk levels are significantly higher.*
- *A quantitative evaluation of the value of object technology (OT), for example, in terms of productivity, effort, time, and maintainability of systems.*
- *An objective evaluation as to whether a particular design/program has all the hallmarks of a good object-oriented design/program or whether it is "sheep dressed as lamb".*
- *The basis for good cost-estimation of object-oriented projects.*
- *An integral part of a well thought out management strategy for object-oriented projects.* (Henderson-Sellers 1996, S. xi)

Zur Nutzung von Metriken zur *Verbesserung der Qualität des Produktes* äußern sich Lorenz und Kidd. Sie schreiben, dass jeder Versuch, die Qualität eines Systems zu verbessern, die Messung des Status quo, die Identifikation von Maßnahmen aufgrund der Ergebnisse und weitere, vergleichende Messungen erfordert (Lorenz/Kidd 1994, S. 3). Ebenso ist die Qualität Fokus der IEEE-Definition einer Qualitätsmetrik. Der IEEE-Standard 1061 von 1992 definiert eine Softwarequalitätsmetrik wie folgt: „Eine Softwarequalitätsmetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit" (IEEE Standards Collection 1992, Übersetzung aus Hindel 1996 S. 2).

Hindel sieht Gründe für die Anwendung von Metriken so auch zuallererst in einer angestrebten Beurteilung von Qualitätseigenschaften (welche natürlich entsprechend näher zu definieren sind). Schlechte Werte können eine präventive Wartung und/oder Überarbeitung begründen (Hindel 1996).

Mills leitet die Notwendigkeit für Metriken aus der so genannten Softwarekrise ab und liefert damit eine Begründung für Metriken zur *Verbesserung des Entwicklungsprozesses* (Mills 1988). Der Begriff der Softwarekrise beschreibt eine Situation, in welcher sich viele Softwareprojekte unfreiwillig wieder finden und welche er durch folgende Punkte charakterisiert:

- grob ungenaue Zeit- und Kostenschätzungen,
- Produktion minderqualitative Software und
- eine Produktivitätsrate, die langsamer steigt als der Bedarf an Software.

Die Softwarekrise resultiert laut Mills aus Problemen beim Management von Softwareprojekten – Mills beschreibt das zu jener Zeit vorherrschende Softwaremanagement als ineffektiv, da es außerordentlich komplex war und nur wenig klar umrissene, verlässliche Metriken sowohl des Prozesses als auch des Produktes verfügbar waren. Zur Verbesserung des Softwaremanagement schlägt auch Mills Metriken vor, da „das Ziel der Anwendung von Metriken die Identifikation und Messung der essentiellen Parameter darstellt, welche die Softwareentwicklung beeinflussen" (Mills 1988, S. 4). Er beklagt zudem, dass Metriken zwar z.T. eingesetzt werden, jedoch selten in methodischer Art und Weise. Diese Ideen werden auch in De Champeaux 1997 (S. 8) aufgegriffen.

Das *Capability Maturity Model* (CMM) des Software Engineering Instituts der Carnegie Mellon University listet die Bestandteile eines Qualitätsmanagementprogramms für die Entwicklung von Software auf. Entwicklungsprozesse werden – je nach eingesetzten Verfahren und Maßnahmen – in 5 Reifegrade eingeteilt (siehe Tabelle 3-1). Ab dem 4. Reifegrad finden sich

auch Softwaremetriken in dieser Aufstellung, d.h., es wird ein quantitatives Verständnis für den Prozess der Entwicklung vorausgesetzt (Hindel 1996).

Tabelle 3-1: Capability Maturity Model

Reifegrad	**Merkmale**
1. Anfänglicher Prozess	▪ Ausgangslage (Chaos)
2. Wiederholbarer Prozess	▪ Vorgehensweisen sind klar definiert ▪ Auf Erfahrungen bei ähnlichen Projekten wird aufgebaut ▪ Projektmanagement
3. Definierter Prozess	▪ Der Prozess ist in seiner Gesamtheit definiert ▪ Trainingsprogramm ▪ Der Prozess wird auf jedes Projekt einzeln angepasst
4. Geführter Prozess	▪ Qualitätslevel für Produkt und Prozess sind definiert ▪ *Einsatz von Produkt-Metriken*[7] ▪ *Einsatz von Prozess-Metriken*
5. Optimierender Prozess	▪ Fokus auf kontinuierlicher Verbesserung des Prozesses ▪ Defekte sollen verhindert werden ▪ Kosten/Nutzen-Rechnungen für neue Technologien

Zum Abschluss der Motivation ist folgende Liste von Zuse passend, welcher obige Punkte zusammenfasst. Er führt genaue Punkte an, wozu Metriken sinnvoll sind:

- *[Als] […] Grundlage für Schätzungen,*
- *um den Projektfortschritt im Auge zu behalten,*
- *[um] (relative) Komplexität zu bestimmen,*
- *um uns zu helfen zu verstehen wann wir einen angestrebten Qualitätszustand erreicht haben,*
- *[um] Defekte zu analysieren,*
- *und um experimentell die beste Vorgehensweise zu bestätigen.*
- *Kurz: Sie helfen uns, bessere Entscheidungen zu treffen* (Zuse 1997, S. 42, eigene Übersetzung).

3.2 Anfänge des Software Measurements

Auf die Notwendigkeit für geeignete Messungen an Softwareprozessen und -produkten ist schon früh in der Entwicklung des noch jungen Software Engineerings hingewiesen worden. Eine der erste Produktmetriken war die Quelltext-Metrik LOC (Lines Of Code). LOC verfügt über zwei unschlagbare Eigenschaften: Die Zahl ist mittels Werkzeugen sehr einfach zu berechnen und intuitiv ist die Bedeutung allen Beteiligten sofort klar, was auch die Schätzung über noch zu schreibende Codemengen vereinfacht. Obwohl die genaue Definition, was tatsächlich als eine Zeile Code betrachtet wird, heftige Diskussionen ausgelöst hat, ist LOC die bekannteste aller Metriken überhaupt. Weitere frühe Metriken sind McCabes Cyclomatic Complexity (McCabe 1976) sowie die Halstead-Metriken (Halstead 1977). All diese Metriken beziehen sich auf die rein prozedurale Programmierung (im Gegensatz zur objektorientierten Programmierung, s.u.).

Zuse (Zuse 1997, S.57ff) gibt einen Überblick über die Geschichte der Softwaremetriken. In folgender Tabelle sind die Anfänge der Softwaremetriken für die strukturierte Programmierung dargestellt.

[7] Zur Unterscheidung zwischen Produkt- und Prozessmetriken siehe unten.

Tabelle 3-2: Anfänge der Softwaremetriken

Zeit	Werk
1950er	LOC (Lines of Code). Der genaue Zeitpunkt der Einführung ist unklar, jedoch ist es denkbar, dass Grace Hopper diese Metrik einsetzte, um die Größe des ersten Compilers A-0 zu messen (1952).
1968	Vermutlich erste Arbeit über Softwarekomplexität veröffentlicht (Rubey/Hartwick 1968).
1971	„An Analysis of Complexity" (Van Emden 1971).
1974	Wolverton unternimmt erste ernsthafte Versuche, die Produktivität von Programmierern mit Hilfe der LOC-Metrik zu bestimmen (Wolverton 1974).
1975	Der Begriff „software physics" wurde von Kolence erschaffen, um wissenschaftliche Methoden auf Computerprogramme anzuwenden (Kolence 1975).
1976	McCabe-Metriken eingeführt (McCabe 1976).
1977	Erstes Buch über Softwaremetriken: „Tom Gilb: Software Metrics" (Gilb 1977).
1977	Halstead führt den Begriff „software science" ein – aus ähnlichen Gründen wie zuvor Kolence den Begriff „software physics" (Halstead 1977).
1977	Halstead-Metriken eingeführt (Halstead 1977).
1977	Zipf's Law auf Programmiersprachen angewendet (Laemmel/Shooman 1977).
1978	Boehm schreibt Buch mit Richtlinien zum Einsatz von Metriken (Boehm et al. 1978).
1979	Function Points von Albrecht eingeführt (Albrecht 1979).
1981	COCOMO COnstructive COst MOdel (Boehm 1981)
	…

Ende der 80er-Jahre hat die Welt der Metriken auch das objektorientierte Paradigma erreicht. Folgende Tabelle gibt einen Überblick über die Anfänge der OO-Metriken:

Tabelle 3-3: Anfänge der OO-Metriken

Jahr	Werk
1988	Rocacher veröffentlicht erste Arbeiten zu OO-Metriken (Rocacher 1988).
1989	Lieberherr/Holland präsentieren „Law of Demeter" (Lieberherr/Holland 1989).
1989	Morris veröffentlich 9 OO-Metriken (Morris 1989).
1991	Metrik-Suite von Chidamber/Kemerer mit 6 Metriken (Chidamber/Kemerer 1991).
1993	Erste Metriken für Objektorientiertes Design (Sharble/Cowen 1993).
1994	Erstes Buch über OO-Metriken von Lorenz/Kidd (Lorenz/Kidd 1994).
1996	Buch über OO-Metriken von Henderson-Sellers (Henderson-Sellers 1996).
	…

Heute existiert eine unüberschaubare Anzahl sowohl traditioneller als auch objektorientierter Metriken. Xenos et al. geben einen Überblick über die bekanntesten Metriken und Tipps für die Verwendung (Xenos et al. 2000).

Fokus dieser Arbeit sind objektorientierte Metriken, daher soll auf diese im Abschnitt 3.4 näher eingegangen werden.

3.3 Kategorisierung von Metriken

Die bekannteste Klassifizierung von Metriken stammt von Fenton (Fenton 1991, S. 42). Hierbei werden Metriken in drei Klassen eingeteilt, namentlich Prozess-, Produkt- und Ressourcemetriken.

- *Prozessmetriken* messen Eigenschaften des zur Erstellung von Software eingesetzten Prozesses. Hierfür sind i.d.R. Messungen über Zeit nötig; ein Beispiel für eine Prozessmetrik ist z.B. die Anzahl gefundener Fehler pro Woche.
- *Produktmetriken* messen Eigenschaften des Produktes selbst. Sie stellen eine Momentaufnahme dar; z.B. die Anzahl implementierter Klassen oder die durchschnittliche Anzahl von Methoden einer Klasse.
- *Ressourcemetriken* messen zur Verfügung stehende Ressourcen wie Personal – meist in vergleichender Weise, d.h. die zur Verfügung stehende Menge wird der tatsächlich genutzten Menge gegenübergestellt.

Im Software Measurement wird zudem zwischen *direkten* und *indirekten* Metriken unterschieden. Hierzu schreibt Fenton:

> *Die direkte Messung eines Attributs ist eine Messung, die nicht von der Messung eines anderen Attributes abhängt. Die indirekte Messung eines Attributs ist eine Messung, die die Messung eines oder mehrerer weiterer Attribute mit sich bringt* (Fenton 1991 S.18, eigene Übersetzung).

Neben dieser Klassifizierung werden Metriken in interne (objektive) und externe (weniger objektive) Metriken eingeteilt (Fenton 1991, S. 43). Hierbei gilt:

- *Interne Metriken* messen direkt zugängliche Attribute des betrachteten Objekts. Sie sind objektiv in dem Sinne, dass sie nur das Objekt ohne seine Umgebung betrachten. Einfache Beispiele sind Zählmetriken wie LOC (Lines Of Code) oder die Anzahl der Klassen im System. Ihre Aussagekraft ist jedoch beschränkt, da meist nur einzelne, technische Details gemessen werden.
- *Externe Metriken* beziehen die Umgebung und den Faktor Mensch mit ein, sind also subjektiv behaftet. Hierbei dreht es sich um Attribute wie Wartbarkeit, Komplexität oder Änderbarkeit, die von hoher Aussagekraft, jedoch nur schwer messbar sind.

Externe Metriken sind ungleich schwieriger zu berechnen, obwohl sie sowohl für Manager als auch für Entwickler und Benutzer die interessanteren Werte liefern. Erschwert wird die Situation durch die Tatsache, dass für die meisten externen Attribute keine einheitliche Definition existiert, z.B. bei „Komplexität“. Softwarequalitätsmerkmale sind i.d.R. externe Maße und lassen sich wie in Abbildung 3-1 kategorisieren (Henderson-Sellers 1996, S.43).

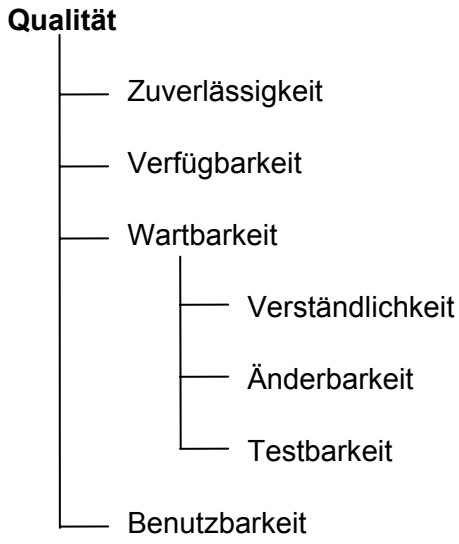


Abbildung 3-1: Kategorisierung von Softwarequalität

Zwischen der Unterscheidung in interne und externe Metriken sowie in direkte und indirekte Messungen besteht ein Zusammenhang. Interne Metriken können i.d.R. direkt gemessen werden, während externe Metriken i.d.R. nur indirekt zu berechnen sind – sie leiten sich aus anderen Metriken, meist internen, ab.

Problematisch ist also die Abbildung von internen auf externe Metriken bzw. die Beantwortung von Fragen nach Wartbarkeit oder Komplexität auf Grundlage von direkten Messungen wie der Anzahl der Klassen im System oder ähnlichem. Hier müssen geeignete Abbildungen entwickelt werden, welche die Ergebnisse der internen Metrik auf diejenigen der externen Metriken abbilden. Die Suche nach derartigen Abbildungen ist in diesem Feld noch lange nicht abgeschlossen.

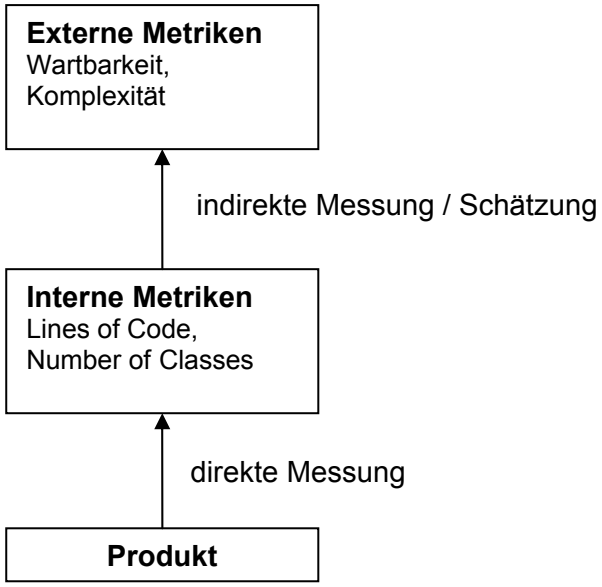


Abbildung 3-2: Externe vs. interne Metriken

Produktmetriken unterschieden sich ferner darin, in welchen Phasen des Entwicklungsprozesses die Berechnung erfolgen kann. Quelltext-Metriken erfordern bereits geschriebenen Code, während Design-Metriken bereits während der Designphase, beim objektorientierten Design z.B. auf UML-Diagramme, anwendbar sind. Vorteile dieser Metriken sind ihre frühe Einsatzmöglichkeit im Design des Systems und damit die Möglichkeit, früh auf Fehler zu reagieren.

Produktmetriken lassen sich ferner auch noch feiner unterteilen. Die größte Liste derartiger Unterkategorien findet sich bei Whitmire (Whitmire 1997) – siehe Tabelle 3-4.

Tabelle 3-4: Kategorien von Whitmire

Size	(Größe)
Complexity	(Komplexität)
Coupling	(Kopplung)
Sufficiency	(Zulänglichkeit)
Completeness	(Vollständigkeit)
Cohesion	(Zusammenhalt)
Primitiveness	(Primitivität)
Similarity	(Ähnlichkeit)
Volatility	(Unbeständigkeit)

An vielen Stellen ist vermerkt worden, dass Modelle von Softwarequalität multidimensional und hierarchisch aufgebaut sind (Littlefair 2001). Daraus folgt, dass einzelne Metriken bzw. die Zahlen, die sich aus ihnen ergeben, niemals einzelnes Maß für die Qualität einer Software sein können. Aus diesem Grunde ist es sinnvoll, solche komplexe externe Maße aus mehreren Metriken zu erschließen. Diese in ihrer Gesamtheit einem einzelnen Zweck dienenden Metriken werden in *Suiten* organisiert, welche meistens bis zu zehn Metriken enthalten.

Um sicherzustellen, dass Metriken stets einem Ziel in der Softwareentwicklung folgen und nicht belanglose Attribute von Softwaresystemen messen, ist eine formale Herangehensweise zu ihrer Definition unabdingbar. Eine solche Herangehensweise ist das Goal-Question-Metric-Paradigma (Basili et al. 1994) – auch kurz GQM genannt – welches in Kapitel 4 zur Definition der in dieser Arbeit vorgestellten Metrik-Suite verwendet und hier daher nicht näher beleuchtet wird.

3.4 Objektorientierte Metriken

Wie oben bereits angekündigt sind objektorientierte Metriken Fokus dieser Arbeit, daher soll auf diese hier näher eingegangen werden. Laut Zuse 1997, S. 276, werden objektorientierte Metriken meist in folgende drei Klassen eingeteilt:

- *Systemweite Metriken*: Derartige Metriken messen z.B. die Anzahl der Dateien, Klassen, Variablen oder systemweite Verhältnisse.
- *Baummetriken*: Derartige Metriken messen Attribute eines Vererbungsbaums, wie die Anzahl der Klassen oder die Höhe des Baums.
- *Klassenmetriken*: Derartige Metriken messen Attribute einer Klasse wie die Anzahl der Sub- oder Superklassen, Anzahl Felder und Methoden etc.

Henderson-Sellers gibt einen Überblick über verfügbare Metrik-Suiten und stellt eine Liste mit empfohlenen Suiten zusammen; ein weiterer, sehr umfangreicher Überblick – allerdings ohne Wertung – ist wie oben bereits erwähnt in Xenos et al. 2000 zu finden. Weitere Definitionen finden sich in Lorenz/Kidd 1994. Henderson-Sellers empfiehlt folgende Suiten (Henderson-Sellers 1996, S.150ff):

Tabelle 3-5: Objektorientierte Metrik-Suiten

Jahr	Veröffentlichung	Suite
1991	Chidamber/Kemerer 1991	Die bekannte Metrik-Suite von Chidamber/Kemerer („CK-Suite“) enthält 6 Metriken. Die Suite ist vielfach zitiert, einzelne Metriken sind jedoch auch kritisiert worden.
1992	Tegarden/Sheetz 1992	Tegarden/Sheets präsentieren eine größere Anzahl von Metriken auf verschiedenen Levels.
1992	Rajaraman/Lyu 1992	Die von Rajaraman/Lyu präsentierte Suite enthält einige Metriken mit einem Fokus auf Kopplung.
1993	Li/Henry 1993	Die von Li/Henry präsentierte Suite enthält 5 der 6 Metriken aus der CK-Suite sowie fünf weitere Metriken: zwei Kopplungs-, eine Zugriffs- und zwei Größe messende Metriken.
1993	Kolewe 1993	Enthält die Metriken der CK-Suite sowie weitere System-Level-Metriken
1993	Lorenz 1993	Lorenz präsentiert elf Design-Metriken, welcher u.A. Größe, Kopplung und Komplexität messen. Auch Prozessmetriken sind integriert.

1994	Brito e Abreu/Carapuca 1994	Diese, so genannte „MOOD-Suite“, ist neben der CK-Suite eine der bekannteren Suiten und enthält 6 Metriken.
1994	Henderson-Sellers/Edwards 1994	In dieser Suite werden 9 Metriken vorgestellt.
1994	Lorenz/Kidd 1994	Die von Lorenz/Kidd vorgestellte Suite enthält über 30 Metriken für unterschiedliche Einsatzzwecke.
1995	Graham 1995	Diese Suite greift die Metriken von Henderson-Sellers und der CK-Suite auf und definiert einige neue Metriken.
1995	Miller et al. 1995	Miller et al. stellen ein Tool für die Metrikberechnung vor, welches unter anderen zwei neue Metriken enthält.

Die bekanntesten Suiten aus obiger Liste sind die Suiten von Chidamber/Kemerer („CK-Suite“, Chidamber/Kemerer 1994) und Brito e Abreu/Carapuça („MOOD“, Brito e Abreu/Carapuca 1994), auf die im Anschluss eingegangen wird. Zusätzlich ist auch das von Lieberherr/Holland vorgestellte „Violations of the Law of Demeter“ (Lieberherr/Holland 1989) von Interesse, es soll hier ebenfalls besprochen werden.

3.4.1 Die CK-Suite

Die Metrik-Suite von Chidamber und Kemerer (Chidamber/Kemerer 1994) ist wohl die bekannteste aller objektorientierten Metrik-Suiten. Besonderheiten liegen im Zeitpunkt des Erscheinens (die hier zitierte Arbeit von 1994 stellt eine Erweiterung einer bereits 1991 vorgestellten Arbeit dar, vgl. Chidamber/Kemerer 1991), der direkten Ausrichtung auf die Objektorientierung sowie der formalen Evaluierung der präsentierten Metriken.

Die Suite wird ausführlich im Anhang in Abschnitt A) i) besprochen. Kurz zusammengefasst enthält die Suite die folgenden sechs Metriken (Tabelle 3-6):

Tabelle 3-6: Metriken der CK-Suite im Kurzüberblick

Metrik	Kategorie	Erläuterung
WMC	Komplexität	Weighted Methods per Class (Gewichtete Methoden pro Klasse). Ergebnis dieser Metrik ist die Summe über alle Methoden der Klasse, wobei pro Methode eine nicht näher definierte – d.h. beliebig wählbare – Funktion die Gewichtung der Methode bestimmt (z.B. anhand der inneren Methodenaufrufe o.Ä.).
DIT	Vererbung	Depth of Inheritance Tree (Tiefe des Vererbungsbaums). Diese Metrik berechnet die Länge des Vererbungsbaums von der aktuell betrachteten Klasse zur Wurzel des Baums; in Fällen von Mehrfachvererbung die maximale Länge.
NOC	Vererbung	Number of Children (Anzahl der Kinder). Diese Metrik bestimmt die Anzahl von direkten Subklassen der betrachteten Klasse.
CBO	Kopplung	Coupling between Object Classes (Kopplung zwischen Objektklassen). Ergebnis dieser Metrik ist die Anzahl der Klassen, zu welcher die momentan betrachtete Klasse Verbindungen aufweist.
RFC	Kopplung	Response for a Class (Reaktion auf eine Klasse). Das „Response Set” einer Klasse wird definiert als diejenigen Methoden, welche als Antwort auf eine an ein Objekt der Klasse gerichtete Nachricht aufgerufen werden. RFC ist die Summe dieser Methoden.

LCOM	Kohäsion	Lack of Cohesion in Methods (Fehlende Kohäsion in Methoden). Diese Metrik bestimmt den inneren Zusammenhalt einer Klasse, indem analysiert wird, welche Mengen von Methoden auf welche Mengen von Instanzvariablen zugreifen.

Einige der in dieser Suite enthaltenen Metriken lassen sich auch für die Analyse von Interfaces verwenden; in Abschnitt 3.5 werden diese Möglichkeiten genauer beleuchtet. In Kapitel 4 erfolgt ein Vergleich der in dieser Arbeit vorgestellten Metriken u.A. mit Metriken aus der CK-Suite.

3.4.2 Die MOOD-Suite

Die von Brito e Abreu und Carapuça vorgestellte Suite – MOOD steht für „Metrics for Object Oriented Design“ – enthält wie die CK-Suite sechs Metriken, allerdings mit einem anderen Fokus (Brito e Abreu/Carapuca 1994, Brito e Abreu et al. 1995). Brito e Abreu und Carapuça stellen vor der Definition der Metriken sieben Kriterien vor, welche ihrer Meinung nach von Metriken erfüllt werden müssen (siehe Anhang, Abschnitt B) ii)) und definieren ihre Metriken nach diesen Kriterien.

Die MOOD-Suite wird im Anhang in Abschnitt A) ii) ausführlich dargestellt. Kurz zusammengefasst enthält die Suite die folgenden sechs Metriken (Tabelle 3-7):

Tabelle 3-7: MOOD-Metriken im Kurzüberblick

Metrik	Kategorie	Erläuterung
MHF	Kapselung	Method Hiding Factor (Methoden-Verbergungs-Faktor). Diese Metrik gibt den Prozentsatz der verborgenen Methoden an den Gesamtmethoden der Klassen an.
AHF	Kapselung	Attribute Hiding Factor (Attribut-Verbergungs-Faktor). Diese Metrik gibt den Prozentsatz der verborgenen Attribute an den Gesamtattributen der Klassen an.
MIF	Vererbung	Method Inheritance Factor (Methoden-Vererbungs-Faktor). Diese Metrik gibt an, wie viel Prozent der Methoden der Klassen geerbt sind.
AIF	Vererbung	Attribute Inheritance Factor (Attribut-Vererbungs-Faktor). Diese Metrik gibt an, wie viel Prozent der Attribute der Klassen geerbt sind.
COF	Kopplung	Coupling Factor (Kopplungsfaktor). COF setzt die tatsächlich vorhandenen Verbindungen zwischen Klassen zu allen potentiell möglichen ins Verhältnis.
PF	Polymorphie	Polymorphism Factor (Polymorphiefaktor). Hierbei wird die Anzahl überschriebener Methoden der Klassen zur Anzahl insgesamt vorhandener Methoden der Klassen ins Verhältnis gesetzt.

Auch die MOOD-Metriken sind z.T. für die Anwendung auf Interfaces geeignet. Siehe hierzu Abschnitt 3.5 sowie Kapitel 4.

3.4.3 Violations of the Law of Demeter

Das Prinzip des Law of Demeter, welches 1989 zum ersten Mal vorgestellt wurde (Lieberherr/Holland 1989), lässt sich mit folgendem Leitsatz für Klassen zusammenfassen: „Sprich nur mit deinen direkten Freunden“. Anders als bei Metriken handelt es sich beim Law of Demeter zunächst um ein Gesetz: Es werden genaue Richtlinien aufgestellt, welche For-

men der Kommunikation zwischen zwei Klassen zulässig sind. Die Metrik besteht dann darin, Verletzungen dieser Regel aufzuzeigen – die *Violations of the Law of Demeter*.

Sharble/Cowen 1993 führen das Kürzel VOD für die Violations of the Law of Demeter ein und quantifizieren die Messung (vgl. Henderson-Sellers 1996, S. 117).

Im Anhang (Abschnitt A) iii)) findet sich eine ausführliche Beschreibung der *Violations of the Law of Demeter*.

3.5 Metriken und Metrik-Suiten mit Bezug zum Interfacekonzept

Fokus dieser Arbeit ist die Definition einer Metrik-Suite zur Analyse des Einsatzes von Interfaces in Java. Die Ziele des interfacebasierten Programmierens wurden bereits in Kapitel 2 vorgestellt; unter Berücksichtigung dieser sollen in diesem Kapitel Metrik-Suiten und Metriken erörtert werden, die für die Untersuchung des Einsatzes von Interfaces gemäß einem der beiden dort vorgestellten Ziele dienen können. In Kapitel 4 wird anschließend eine Suite mit sieben Metriken definiert, die z.T. hier vorgestellte Metriken enthält, z.T. jedoch auch neue, welche Lücken in den verfügbaren Metriken füllen.

Für die Aufstellung der Metriken und Metrik-Suiten in diesem Kapitel wurden verschiedene Quellen herangezogen; neben wissenschaftlichen Arbeiten wurden auch kommerziell und nichtkommerziell verfügbare Metriktools untersucht, welche mit der Unterstützung der Sprache Java meist auch Metriken für Interfaces anbieten (berücksichtigt wurden die Tools OptimalJ, JDepend, RefactorIT, EssentialMetrics und JRefactory).

Bislang ist mir nur eine einzige Metrik-Suite bekannt, welche sich explizit mit dem Phänomen des Interfaces beschäftigt – es handelt sich dabei um die in Steimann et al. 2003 vorgestellte Suite „Interface-related program metrics“. Diese Suite soll in ihrer Gesamtheit als Abschluss des Kapitels 3.5 in Abschnitt 3.5.4 vorgestellt werden.

Neben den in dieser Suite vorgestellten Metriken gibt es nur einige wenige Einzelmetriken, deren Fokus direkt auf der Nutzung des Interfaces an sich liegt; es handelt sich dabei um in den genannten Tools auf die Nutzung von Interfaces angepasste Klassenmetriken. Derartige Klassenmetriken, welche meist mit nur minimalen Veränderungen auch die Eigenschaften des Interfaces bzw. der entstehenden Interface-Hierarchien berücksichtigen können, sollen im Abschnitt 3.5.1 untersucht werden.

Metriken, welche die Nahtstelle der beiden Hierarchien – Implementations- und Interfacehierarchie – näher untersuchen, sind ebenfalls (noch) recht spärlich gesät. Mögliche (sowie wiederum angepasste) derartige Metriken werden im Abschnitt 3.5.2 besprochen.

Die Definition des Ziels 2 der interfacebasierten Programmierung in Kapitel 2 enthält das Konzept des Kontexts und des kontextspezifischen Interfaces. Während viele Metriken zwar die Nutzung eines Typs messen – z.B. RFC aus der CK-Suite – wird stets die Nutzung aus allen Kontexten heraus betrachtet, nicht jedoch der Kontext als eigenständiges Gebilde. In vielen Fällen werden auch die Verbindungen zweier Klassen – durch Felder – betrachtet, jedoch nicht die Parameter von Methoden oder lokale Variablen untersucht. Diese Tatsache wird in Abschnitt 3.5.3 erörtert.

3.5.1 Klassenmetriken mit Relevanz für Interfaces

Eine große Anzahl an Metriken, welche die Attribute von Klassen messen, ist auch für Interfaces nutzbar; z.B. Messungen der Stellung eines Typs in der Hierarchie, des Fan-In-Faktors (s.u.) oder der Anzahl der Methodenparameter in einem Typ. Nicht geeignet sind Metriken, welche Implementation voraussetzen; hierzu zählen v.a. Fan-Out-Faktoren (s.u.) sowie jede Metrik, welche auf Kontrollstrukturen operiert.

Die Verwendungsmöglichkeiten derartiger Metriken sind nicht auf eine bestimmte Metrikgattung beschränkt. Sowohl Metriken für Kopplung oder Größe als auch für Komplexität etc. sind mögliche Kandidaten.

Die Liste der für Interfaces nutzbaren Metriken ist lang; daher sollen hier nur einige Beispiele gegeben werden. Die hier angeführten Metriken sind Kapitel 6 aus Henderson-Sellers 1996 entnommen; sie sind in ihrer jeweiligen Originalversion lediglich auf Klassen definiert. Die Anpassung auf Interfaces ist jedoch wie folgt möglich:

- *Fan-In*. Die Fan-In-Metrik existiert seit der strukturierten Programmierung und bezeichnet dort die Anzahl der Stellen, an welchen die Kontrolle in eine betrachtete Einheit übergeben wird (im Gegenzug hierzu misst die Fan-Out-Metrik die von einer betrachteten Einheit angesprochenen externen Module). Seit der erstmaligen Verwendung durch Henry und Kafura (vgl. Henderson-Sellers 1996, S. 86) haben sich, insbesondere auch in der Objektorientierung, viele unterschiedliche Arten der Fan-Messung gebildet, um Vererbungskopplung, Kopplung durch Variablendeklaration oder Methodenaufrufe oder eben auch Protokollkopplung zu berechnen. Auf Interfaces ist lediglich die Fan-In-Messung anwendbar, welche in mindestens folgenden auf Klassen wie Interfaces anwendbaren Formen existiert:
    - *Inheritance Fan-In*: Anzahl Superklassen/Superinterfaces
    - *Coupling Fan-In*: Anzahl der Variablendeklarationen mit dieser Klasse/diesem Interface
    - *Method Call Fan-In*: Anzahl der Methodenaufrufe von Methoden in dieser Klasse/in diesem Interface.

- *Number of Parameters per Method*. Diese Metrik, z.B. von Lorenz/Kidd 1994 als „PPM“ präsentiert, kann Aufschlüsse über die Programmierweise geben.
- *Width/Height of Inheritance Hierarchy*. Hieraus lassen sich mehrere Metriken ableiten; sinnvoll sind derartige Messungen zur Analyse des Radius der Auswirkungen von Veränderungen auf gewisse Klassen bzw. Interfaces.
- *Number of Methods per Class* (bzw. *Interface*). Diese Metrik, z.B. in Lorenz/Kidd 1994 als „NIM“ (*Number of Instance Methods*) präsentiert, kann helfen, die korrekte Aufteilung von Verantwortung über mehrere Klassen zu sichern.

Die meisten der untersuchten Metriktools praktizieren diesen Ansatz der „Übersetzung” vorhandener Klassenmetriken auf Interfaces. In C++ ist dies ohnehin bereits gegeben, da Interfaces dort als abstrakte Klassen realisiert sind. OptimalJ sowie JRefactory verfügen über die Metriken Na und Ni für die Anzahl abstrakter Klassen bzw. Interfaces im System; ebenso RefactorIT mit der Metrik NOTa (Number of Abstract Types). EssentialMetrics wendet traditionelle Größenmessungen auf Interfaces an: ISA und ISO sind Messungen der Anzahl Attribute bzw. Methoden in einem Interface und NOCI die Anzahl der Subinterfaces.

Eine Gegenüberstellung der Interface- und Implementationshierarchien ist durch die Metrik DIP realisiert (Martin 2002). DIP steht für „Dependency Inversion Principle” und ist wie folgt definiert:

> *The DIP metric is defined as the percentage of dependencies in a package or class that has an interface or an abstract class as a target. A DIP of 100% indicates that all dependencies in a package diagram are based on interfaces or abstract classes.* (Knoernschild 2001)

Die Gegenüberstellung erfolgt hier jedoch für alle im Package definierten Abhängigkeiten, d.h. für alle definierten Klassen und Interfaces; die Analyse der Verwendung einer bestimmten Klasse ist nicht vorgesehen.

Ein weiterer derartiger Vergleich ist über die Metrik „Abstractness“ möglich (Martin 1994), welche das Verhältnis zwischen abstrakten Klassen bzw. Interfaces zu allen Typen in einem gewissen Package berechnet:

> *Abstractness: (# abstract classes in category ÷ total # of classes in category). This metric range is [0, 1]. 0 means concrete and 1 means completely abstract.* (Martin 1994)

Bei der direkten Übersetzung der Klassenmetriken, auch für Java-Interfaces, werden Interfaces allerdings nicht als eigenständiges Konzept betrachtet. Damit können die Besonderheiten des Interface-Konstruktes, wie sie in Kapitel 2 hervorgehoben worden sind, nur unzureichend bzw. gar nicht berücksichtigt werden. Insbesondere kann so zwar eine Gegenüberstellung von Interface- und Klassennutzung erzielt werden, jedoch können die Nahtstellen zwischen Interface- und Klassenhierarchie nicht analysiert werden. Dies ist Thema des nächsten Abschnitts.

3.5.2 Verbindung der Klassen- und Interfacehierarchien

Metriken, welche Vererbung messen (*inheritance metrics*) sind meist nur innerhalb einer Hierarchie definiert bzw. behandeln rein abstrakte Klassen nicht separat. Interessant sind jedoch auch die Nahtstellen der Klassen- und Interfacehierarchie, also die Implementation von Interfaces durch Klassen. Derartige Metriken erlauben eine Quantifizierung der Klassen-Interface-Beziehungen und ermöglichen dadurch ein tiefergehendes Verständnis dieser Beziehungen.

Nur wenige derartige Metriken sind mir bekannt; die meisten stammen zudem aus der bereits genannten Suite von Steimann (Steimann et al. 2003). Viele Metriken lassen sich jedoch, wie zuvor bereits in einer reinen Übersetzung, auch an den Hierarchiegrenzen ansetzen: So kann die Metrik DIT der CK-Suite nicht nur zur Messung der Klassenhierarchie (Depth of Inheritance Tree), sondern auch für die Messung der Tiefe der Interface-Hierarchie genutzt werden – und zwar nicht nur von einem Interface aus (dies wäre eine direkte Übersetzung der Metrik wie in Abschnitt 3.5.1 beschrieben) sondern auch über die Grenzen der Hierarchien hinweg, d.h. von einer Klasse aus auf den Interface-Baum: So kann die Tiefe der Klasse im Interface-Baum bestimmt werden.

Weitere Metriken, die in dieser Art und Weise verwendet werden können sind z.B. folgende:

- Neben DIT ist auch der „nesting level“ von Lorenz/Kidd 1994 oder die „class-to-root-depth“ von Tegarden/Sheetz 1992 geeignet (vgl. Henderson-Sellers 1996, S. 130).
- Die Metriken zur Messung der Vererbung von Methoden aus Lorenz/Kidd 1994, S. 66ff sind ebenso geeignet; z.B. kann “Number of methods inherited by a subclass” auch als „Number of methods inherited by a class from its interfaces” verwendet werden.
- NOC (Number of Children) kann sowohl als direkte Übersetzung („number of child interfaces”) (wie im Tool EssentialMetrics verwendet) als auch zur Verbindung der Hierarchien genutzt werden, z.B. als „number of implementing classes“.

- Die Gegenüberstellung von „public methods of class“ und „hidden methods of a class“ kann auf die Methoden eines Interfaces im Vergleich zu denen der Klasse erweitert werden – und misst dann die Abdeckung der Klasse durch die implementierten Interfaces.

Eine Metrik, in welcher bereits konkret der Übergang zwischen den Hierarchien berücksichtigt wurde, ist CIIfIn aus Patenaude et al. 1999: „class interface extension fan-in (number of parent interfaces)”.

3.5.3 Die Verwendung von Typen – Kontextanalyse

Nicht nur die Implementation von Interfaces durch Klassen ist für das Interfaceparadigma relevant (Anbieterseite), sondern insbesondere auch die Verwendung von Interfaces als Typen von Variablen (Nutzerseite), da erst dies die reale Entkopplung darstellt. Die Verwendung von Interfaces als Typen kann zunächst als eine Form der Kopplung verstanden werden, mögliche Messungen gehen jedoch – insbesondere wenn das zweite in Kapitel 2 definierte Ziel der interfacebasierten Programmierung berücksichtigt wird – wesentlich weiter, da hier die Einführung kontextspezifischer Interfaces gefordert wird. Metriken für die Messung der Anzahl interfacetypisierter Variablen – auch im Vergleich zu klassentypisierten Variablen – sind oben bereits besprochen worden (vgl. *DIP* oder *Abstractness* aus Martin 2002); jedoch existieren bisher keine Metriken, welche den direkten Vergleich zwischen der Anzahl der Variablen zulassen, die einerseits mit einer Klasse und andererseits mit einem implementierten Interface genau dieser Klasse deklariert sind.

Für die Messung der Verwendung einer Klasse existieren bereits Metriken, so z.B. die verschiedenen Ausprägungen des Fan-In. Diese Messung existiert in höchst unterschiedlichen Formen: Eine Möglichkeit ist die Messung der überhaupt vorhandenen Bindungen – also Variablendeklarationen – eine weitere die Messung jedes einzelnen Methodenaufrufs an eine Klasse. Diese Messungen sind natürlich auch für Interfaces möglich, allerdings wird dann nicht berücksichtigt, Instanzen welcher Klassen betroffen sind.

Bei Metriken zur Messung der Methodenaufrufe werden die gemessenen Daten jedoch entweder pro Empfängerklasse oder höchstens pro Senderklasse, nicht pro Variable geordnet. Beispiele sind (aus Li/Henry 1993):

- *DAC (Data Abstraction Coupling): number of nonsimple attributes of a distinct type in a class.*
- *MPC (Message-Passing Coupling): number of send statements defined in a class* (Henderson-Sellers 1996).

Diese Coupling-Metriken sind stets auf einer vollständigen Klasse definiert. Es existieren Ansätze, dies auf Methodenebene herunterzuziehen (*fan-in for methods* aus Henderson-Sellers 1996), ebenso auf Variablenebene (*fan-in/out* und *polymorphism* aus Tegarden/Sheetz 1992), jedoch ist das Konzept der Anwendung auf *spezielle, einzelne Kontexte einer Klasse*, d.h. die Verwendung jeweils einer konkreten Variable eines konkreten Typs noch nicht vorhanden; es werden stets alle Variablen eines Typs berücksichtigt.

Dies mag daran liegen, dass die Berechnung für jeden Kontext einen wesentlich erhöhten Zeitaufwand mit sich trägt. Betrachtet man obige Metriken, so sind diese sehr einfach zu berechnen:

- DAC: Der einmalige Durchlauf der Felder innerhalb einer einzelnen Klasse und eine anschließende Gruppierung sind ausreichend.
- MPC: Der einmalige Durchlauf der Methoden innerhalb einer einzelnen Klasse und eine anschließende Gruppierung sind ausreichend.

Eine Metrik, welche die Verwendung von Methoden in Variablen einer Klasse untersucht ist jedoch wesentlich komplexer zu berechnen. Hierfür ist es nötig, das gesamte Projekt nach Variablen abzusuchen, welche mit einer gewissen Klasse typisiert sind sowie alle im Gültigkeitsbereich der Variable auf dieser Variable aufgerufenen Methoden zu identifizieren.

Derartige Metriken werden in Kapitel 4 definiert. Eine mögliche Implementation wird in Kapitel 5 beschrieben.

3.5.4 Die Metrik-Suite von Steimann et al.

Die in Steimann et al. 2003 definierte Suite besteht aus acht Metriken, die direkt für die Analyse des Einsatzes von Interfaces konzipiert sind. Enthalten sind im Abschnitt 3.5.1 genannte, auf Interfaces angewandte Klassenmetriken sowie Metriken zur Untersuchung der Hierarchieschnittstellen. Kontextspezifische Metriken sind nicht enthalten.

Die Metriken der Suite sind in drei Kategorien eingeteilt: Klassenmetriken, Interfacemetriken und Systemmetriken.

Klassenmetriken:

- *Polymorphic Grade (PG):* Polymorphic Grade misst die Anzahl der (auch indirekt) implementierten Interfaces einer Klasse und entspricht somit der Metrik CIIfIn aus Patenaude et al. 1999. Die Metrik steht an der Schnittstelle zwischen Interface- und Klassenhierarchie.
- *Versatility (VERS):* Versatility (Vielseitigkeit) ist ein Anzeiger für die Überlappungsfreiheit der von der Klasse implementierten Interfaces. Der Wertebereich erstreckt sich von 0 (kein Interface implementiert) bis hin zu PG der Klasse (alle Interfaces paarweise verschieden). Ein Wert von 1 zeigt an, dass alle Interfaces dieselben Methoden enthalten. Je höher der Wert, desto unterschiedlicher die Klassennutzung, daher der Name.
- *Polymorphic Use (PU):* Diese Metrik berechnet den Quotienten aus der Anzahl mit einem Interface der Klasse typisierten Variablen geteilt durch die Anzahl mit einem Interface oder der Klasse selbst typisierten Variablen. Es handelt sich also um den Quotient aus den kumulierten Coupling Fan-In-Faktoren der implementierten Interfaces geteilt durch die kumulierten Coupling Fan-In-Faktoren der Interfaces sowie der Klasse.

Interfacemetriken:

- *Generality (IGEN):* Anzahl der Klassen im Programm, die das gewählte Interface implementieren. Dies entspricht dem Inheritance Fan-Out eines Interfaces, wenn hierarchieübergreifend gearbeitet wird.
- *Popularity (IPOP):* Anzahl der Variablen im Programm, die mit dem gewählten Interface typisiert sind. Dies entspricht dem Coupling Fan-In-Faktor des Interfaces.

Systemmetriken:

- *Interface to Class Ratio:* Anzahl Interfaces im Programm geteilt durch die Anzahl Klassen im Programm. Diese Metrik entspricht Abstractness aus Martin 1994, wenn das gesamte System betrachtet wird.
- *Interfaces typed to Class typed Variables Ratio:* Anzahl im Programm mit Interfaces deklarierte Variablen geteilt durch Anzahl im Programm mit Klassen deklarierte Variablen. Diese Metrik entspricht dem Dependency Inversion Principle (DIP) aus Martin 2002 für das gesamte System.

Die Metriken Polymorphic Grade sowie Polymorphic Use werden in die in dieser Arbeit definierte Metrik-Suite übernommen (vgl. Kapitel 4).

3.6 Evaluation von Metriken

Es ist von vielen Autoren und Forschern darauf hingewiesen worden, dass Softwaremetriken gewisse Qualitätsmerkmale erfüllen müssen, um als relevant und aussagekräftig gelten zu können (Fenton 1991, Henderson-Sellers 1996, Zuse 1997, De Champeaux 1997). Hierbei sind unterschiedliche Kriterien entstanden, die wünschenswerte Eigenschaften von Softwaremetriken beschreiben.

Derartige Kriteriensammlungen dienen dem Zweck, einen allgemeinen Qualitätsstandard für Softwaremetriken zu definieren. Die Kriterien werden dabei entweder aus der Praxis abgeleitet, z.B. durch Erfahrung oder Experimente, oder basieren auf theoretischen Annahmen.

Zuse untersucht verschiedene, in der Literatur vorgeschlagene Mengen von Kriterien zur Überprüfung von Metriken (Zuse 1997). Dabei werden die Kriteriensammlungen von 13 Forschern berücksichtigt, u.A. die Kriterien von Weyuker (siehe unten) sowie ein IEEE-Standard.

Zuse merkt an, dass es keine allgemeine Auffassung von geeigneten Kriterien zu geben scheint. Insbesondere schließen sich die Kriterien mancher Autoren gegenseitig aus. Dies zeigt, dass die Forschung in diesem Bereich noch keinesfalls abgeschlossen ist: „Die besprochenen Kriterien zeigen, dass nach mehr als zwanzig Jahren Forschung im Bereich quantitativer Methoden in der Softwaretechnik noch immer viele Fragen offen sind.“ (Zuse 1997, S. 398, eigene Übersetzung).

Zeitgleich mit dem Werk von Zuse hat Whitmire eine umfassende Sammlung von Kriterien vorgelegt. Anders als Zuse versucht Whitmire, die gefundenen Kriterien in Einklang zu bringen und durch eigene zu ergänzen (Whitmire 1997). Er teilt Designmetriken in neun Gruppen ein (unter anderem Größe, Komplexität, Kopplung: siehe Tabelle 3-4) und definiert für jede Gruppe eigene (jedoch oft überlappende) Merkmale für derartige Metriken. Ein Beispiel sind folgende fünf Kriterien für Kopplung, welche Whitmire aus Briand et al. 1996 übernommen hat:

1. Kopplung ist nicht negativ.
2. Kopplung kann null sein.
3. Hinzufügen einer Beziehung zwischen zwei Komponenten verringert die Kopplung nicht.
4. Vereinigen zweier Komponenten erhöht die Kopplung nicht.
5. Vereinigung von zwei vorher nicht verbundenen Komponenten ändert die Kopplung nicht.

Die bekanntesten formalen Kriterien für Metriken sind die neun Kriterien von Weyuker, welche 1988 für Komplexitätsmetriken definiert wurden. Die Definition wie auch Begründung der Metriken ist sehr mathematisch gehalten und orientiert sich auch an mathematisch überprüfbaren Eigenschaften von Metriken. Die Kriterien umfassend u.A. Monotonität, Granularität und weitere mathematische Prinzipien, z.B. die Auswirkungen auf Komplexitätsmaße bei der Komposition zweier Programmteile. Eine genaue Aufstellung dieser Kriterien findet sich im Anhang, Abschnitt B) i).

Neben mathematisch begründeten Kriterien – wie den Weyuker-Kriterien oder auch den von Whitmire betrachteten – gibt es auch weichere, verbal formulierte Kriterien. Eine Sammlung solcher Kriterien findet sich neben der Definition der oben bereits vorgestellten MOOD-Suite in Brito e Abreu/Carapuca 1994. Die Informalität der Kriterien – ein klarer Gegensatz zu den von Weyuker definierten Kriterien – ist für die Identifikation von geeigneten Metriken z.T. von großem Vorteil, da die Kriterien auf die praktische Verwendbarkeit der Metriken abzielen: Die Kriterien umfassen einfache Berechenbarkeit, Skalierbarkeit und weitere, informelle Kriterien

wie z.B. einen möglichst frühen Zeitpunkt der Berechenbarkeit während der Entwicklung eines Softwaresystems. Eine genaue Aufstellung findet sich wiederum im Anhang, Abschnitt B) ii).

Wie die Diskussion in Zuse 1997 zeigt, ist es momentan nicht möglich eine allgemein anerkannte Vorgehensweise für die Evaluierung von Metriken aufzuzeigen. Da die formale Evaluation kein zentrales Thema dieser Arbeit ist, wird hier aus diesem Grund nicht detailliert darauf eingegangen.

Die Kriterien der MOOD-Suite werden im Anhang erneut aufgegriffen, um die in Kapitel 4 präsentierte Metrik-Suite zu evaluieren; die Weyuker-Kriterien wurden als Gegenpol und erweiterte Information zu den Kriterien der MOOD-Suite mit aufgenommen.

4 Definition einer Metrik-Suite

Ein sinnvoller Einsatz von Java-Interfaces wurde in Kapitel 2 beschrieben und erläutert. Bereits vorhandene Metriken, mit welchen ein derartiger Einsatz untersucht und analysiert werden kann, sind in Kapitel 3 besprochen worden. Dabei wurde deutlich, dass zwar schon einige Metriken und eine Metrik-Suite zur Untersuchung von Interfaces existieren, jedoch in vielen Fällen der spezielle Fokus fehlt und die Untersuchung von Kontexten noch nicht integriert ist.

Diese Lücke schließt die in diesem Kapitel vorgestellte Metrik-Suite zur Untersuchung des Einsatzes von Interfaces in Java. Außerdem wird eine zusätzliche Analysemethode und deren Umsetzung als Utility mit Metrikcharakter (dem *Context Analyzer*) vorgestellt. In Kapitel 5 folgt die Beschreibung der Implementation der Suite sowie des *Context Analyzers*.

Bei den hier vorgelegten Metriken handelt es sich der Klassifikation nach um Produktmetriken objektorientierter Systeme und hierbei insbesondere um Quelltextmetriken, d.h. Metriken, die während der Implementationsphase direkt aus dem Quelltext gewonnen werden. Der spezielle Fokus liegt auf Interfaces der Programmiersprache Java. Die Metriken sind in ihrer Definition also sprachabhängig; über die Definition eines allgemeinen Interfaces lassen sie sich jedoch ohne Probleme auf andere Sprachen übertragen.

4.1 Geeignete Metriken

Refactorings sind für die Überarbeitung und Verbesserung von Code nahezu unersetzlich geworden. Bei großen Systemen stellt sich jedoch die Frage, an welchen Stellen Refactorings eingesetzt werden sollen; eine Frage, welche von Fowler mit den bereits erwähnten *bad smells* beantwortet wird (Fowler 2000).

Die zuverlässige Entdeckung von *bad smells* setzt jedoch einiges an Erfahrung bei den Entwicklern voraus (Fowler 2000, S.75). Während der Mensch stets die letzte Instanz vor der Durchführung von Refactorings darstellt, können Metriken – insbesondere bei großen Projekten – helfen, Stellen im Quelltext zu identifizieren, welche näherer Inspektion bedürfen.

In diesem Abschnitt wird eine Suite von Metriken vorgestellt, die dem Entwickler beim Auffinden der in Kapitel 2 genannten *bad smells* helfen soll. Gleichzeitig stellt sie ein Werkzeug zur Verfügung, um ganze Projekte auf ihre Interface-Nutzung hin zu untersuchen und diese Ergebnisse mit anderen Projekten zu vergleichen.

Dabei kommt ein besonderer Ansatz zur Entwicklung von Metriken zum Einsatz, welcher zum einen die Qualität der entwickelten Metriken sicherstellt und zum anderen eine Rückführbarkeit der (numerischen) Ergebnisse auf die ursprüngliche Fragestellung sichert. Es handelt sich dabei um das *Goal-Question-Metric*-Paradigma (Basili et al. 1994).

4.1.1 Goal-Question-Metric – Ein Überblick

Software-Systeme und -Produkte sind hochkomplexe Einheiten, welche viele Anhaltspunkte für eine Metrikberechnung bieten. Die Möglichkeiten, Software in Zahlen auszudrücken, sind unbegrenzt. Dies gilt auch für das Angebot an Software-Metriken.

Während viele Metriken, sinnvoll angewendet, zu verwertbaren Ergebnissen führen, gibt es jedoch auch eine große Anzahl von Metriken und Metrik-Programmen, die scheinbar um des

Selbstzwecks willen messen: Sie messen, was einfach zu messen ist – jedoch nicht das, was echte Rückschlüsse auf das betrachtete Produkt zulassen würde (Lichter 2003).

Aus dieser Erkenntnis wurde das GQM-(*Goal-Question-Metric*)-Paradigma entwickelt. Mit diesem ist es möglich, Metriken von zuvor definierten Zielen in der Software-Entwicklung abzuleiten, wodurch ihre Resultate eine festgelegte Bedeutung erhalten.

Der GQM-Ansatz führt zunächst als *Top-Down-Approach* von der Festlegung eines allgemeinen Ziels (*Goal*) zur Entwicklung von Fragen (*Questions*), aus welchen sich ergibt, inwieweit das Ziel in einem bestimmten Projekt erreicht worden ist, bis hin zur Festlegung von Metriken (*Metrics*), welche die Fragen quantitativ beantworten können.

Sind die Metriken einmal festgelegt, können deren Ergebnisse – basierend auf den zuvor festgelegten Fragen und Zielen – in einem *Bottom-Up-Approach* analysiert und interpretiert werden (vgl. Abbildung 4-1).

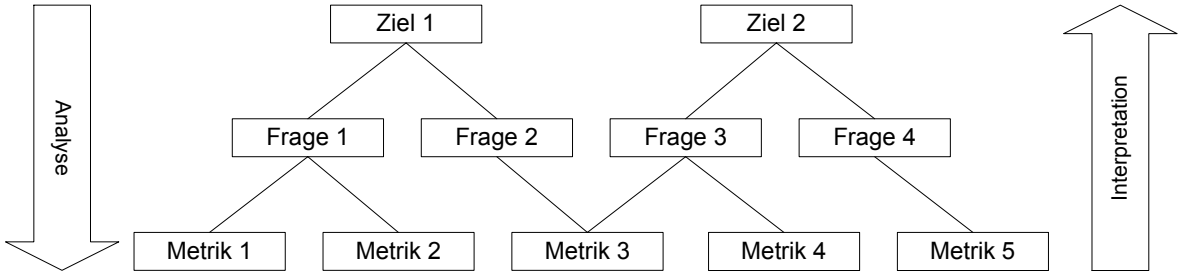


Abbildung 4-1: Goal-Question-Metric

GQM wurde zuerst an der University of Maryland als Teil des TAME-Projekts (ab 1984), später dann an der Universität Kaiserslautern (ab 1992) und dem Fraunhofer Institut für experimentelle Softwaretechnik (ab 1996) entwickelt (Differding et al. 1996). Seine Ursprünge hat das Projekt jedoch bei der NASA (Basili/Weiss 1984); hier wurde es verwendet, um Fehler in Projekten auszuwerten. Der Ansatz ist nicht nur von theoretischer Bedeutung, sondern wird in der Wirtschaft bereits eingesetzt (NASA, HP, Motorola, vgl. Basili et al. 1994 S.9).

Viele Beschreibungen des Goal-Question-Metric-Ansatzes schildern eine Herangehensweise aus Sicht des Managements, d.h. Ziele, Fragen und Metriken beziehen sich auf ein Unternehmen oder gewisse Arbeitsabläufe innerhalb eines Unternehmens. Der GQM-Ansatz lässt sich jedoch auch mit einer technischen Perspektive einsetzen, was hier geschehen soll.

4.1.1.1 Prinzipien

Das GQM-Paradigma basiert auf dem Grundsatz, dass die Erhebung von Metriken einem eindeutig definierten Ziel folgen sollte. Eine solche Herangehensweise hat u.A. die folgenden Vorteile (Differding et al. 1996):

1. Sie hilft bei der Entwicklung sinnvoller und für die Problemstellung relevanter Metriken – in Abbildung 4-1 durch den Pfeil nach unten markiert.
2. Das aufgestellte Ziel gibt den Kontext für die spätere Analyse und Interpretation der von den Metriken gesammelten Daten vor – in Abbildung 4-1 durch den Pfeil nach oben markiert.
3. Die aus der Auswertung der Daten gezogenen Schlussfolgerungen werden durch die klare Begründung der gewählten Metriken bestätigt.

Differding et al. nennen u.A. folgende Prinzipien für den Einsatz des Paradigmas:

1. Ausgehend von einem detailliert festgelegten Messziel muss die durchzuführende Analyse präzise und explizit spezifiziert werden.
2. Metriken müssen von Zielen und Fragen abgeleitet werden. Ziele und Fragen dürfen nicht auf bestehende Metriken angepasst entwickelt werden.
3. Für jede Metrik muss eine logische Grundlage – ausgedrückt durch die entwickelten Fragen – existieren, welche die Datenerhebung rechtfertigt und Richtlinien für die Verwendung der Daten vorgibt.
4. Die von den Metriken gesammelten Daten müssen unter Verwendung der Fragen und des Ziels interpretiert werden. Dies limitiert die Verwendung der Daten auf den tatsächlich vorgesehenen Bereich.

4.1.1.2 Funktionsweise

Die Anwendung des GQM-Paradigmas führt zur Spezifikation eines Maß-Modells, das genau auf die zu untersuchenden Attribute des zu betrachtenden Systems – z.B. einer entwickelten Anwendung – zugeschnitten ist. Das Maß-Modell ist in drei Stufen oder Ebenen aufgeteilt:

- *Konzeptuelle Stufe*: In dieser Stufe werden Ziele für sich innerhalb des Systems befindliche Objekte aufgestellt. Als Objekt können dienen:
    1. Produkte: Während des Lebenszyklus des Systems produzierte Artefakte, wie Programme, Dokumente, Spezifikationen.
    2. Prozesse: Über Zeit ablaufende Aktivitäten mit Bezug zur Entwicklung eines Systems, z.B. testen, entwerfen o.Ä.
    3. Ressourcen: Von Prozessen zur Produktion des Outputs genutzte Elemente, z.B. Personal, Software.
- *Funktionale Stufe*: Pro Ziel wird mindestens eine Frage entwickelt, die beschreibt, wie das jeweilige Ziel erreicht werden kann – basierend auf den das Ziel näher eingrenzenden Kriterien (siehe unten).
- *Quantitative Stufe*: Zu jeder Frage wird mindestens eine Metrik ausgearbeitet, welche aus dem Objekt der Betrachtung entnommene Daten so aufarbeitet, dass die Frage quantitativ beantwortet wird. Die Daten sind dabei entweder objektiv (basieren nur auf dem betrachteten Objekt) oder subjektiv (auch der jeweilige Blickwinkel des Betrachters wird miteinbezogen).

4.1.2 Definition von Zielen

Die Festlegung der Ziele ist kritisch für die Entwicklung des gesamten Maß-Modells, da ein Ziel den einzelnen Ausgangspunkt eines ganzen Baums von Fragen und Metriken darstellt (vgl. Abbildung 4-1).

Um das Ziel genau zu charakterisieren, listen Basili et al. sowie Differding et al. einige Entscheidungshilfen auf (Basili et al. 1994 S.4, Differding et al. 1996 S. 9). Diese beleuchten das System aus einer Managementperspektive heraus:

1. *Objekt*. Auf das Objekt bezieht sich das gesamte Ziel; das betrachtete Objekt kann z.B. das Endprodukt einer Softwareentwicklung sein.
2. *Zweck*. Der Zweck gibt an, aus welchem Grund die gesamte Messung durchgeführt wird. Es kann sich dabei z.B. um den Versuch handeln, eine Charakterisierung eines Objektes vorzunehmen.
3. *Fokus*. Die Untersuchung bezieht sich meist nicht auf ein gesamtes Produkt oder einen gesamten Prozess, sondern hat einen engeren Fokus auf das Objekt. So kann z.B. die *Zuverlässigkeit* eines Produkts betrachtet werden.

4. *Blickwinkel*. Ziele werden stets aus einem gewissen Blickwinkel, z.B. dem des Geschäftsführers, festgelegt. Das Ziel ist von diesem Blickwinkel aus gesehen zu erreichen.

Das GQM-Paradigma kann, in abgewandelter Form, jedoch auch aus der hier erforderlichen technischen Perspektive heraus angewendet werden. Ein Ziel wird dann mit größerem Bezug zur Technik definiert: Das Objekt bezeichnet eine Komponente der Anwendung; Fokus und Blickwinkel beziehen sich auf dieses enger eingegrenzte Objekt und sind damit ebenso in der Anwendung angesiedelt; der Zweck ist stärker dem Produkt verhaftet als dem Prozess oder den Ressourcen. Alle Vorteile der drei Ebenen des GQM-Ansatzes bleiben erhalten.

Das allgemeine Ziel der hier zu erarbeitenden Metrik-Suite wurde bereits in Kapitel 2 erörtert. Zwei Einsatzzwecke bzw. Einsatzziele von Interfaces sind dort definiert worden, die nun in Ziele des GQM-Paradigmas umgewandelt werden können.

Ziel 1 (Z1)

- Zweck: Vollständige Entkopplung *des*
- Fokus: Zugriffs *auf*
- Objekt: Klassen
- Blickwinkel: *sowohl aus* Anbieter- *als auch aus* Nutzerperspektive.

Z1 korrespondiert direkt zu dem ersten in Kapitel 2 dargelegten Einsatzziel von Interfaces: Der Zugriff auf Klassen soll dadurch entkoppelt werden, dass die öffentlichen Features der Klasse in Interfaces ausgelagert werden und die Klasse diese Interfaces implementiert. Der Klient der jeweiligen Klasse sollte dann nur noch über die in den Interfaces spezifizierten Methoden auf die Klasse zugreifen – er kennt idealerweise nur noch die Interfaces und nicht mehr die Klasse selbst.

(Positiv-) Beispiele für die Erfüllung dieses Ziels finden sich z.B. in der Java-API. Die Klasse `javax.swing.BoxLayout`, welche verschiedene Interfaces implementiert, wird z.B. nie direkt referenziert; alle Klienten arbeiten stets mit einem Interface; ebenso sind *alle* Features der Klasse `BoxLayout` in den Interfaces vorhanden, folglich müssen künftige Klienten ebenfalls nur ein Interface referenzieren (Abbildung 4-2).

```
/**
 * BoxLayout implementiert LayoutManager2, welches von LayoutManager
 * erbt, sowie Serializable
 **/
public class BoxLayout implements LayoutManager2, Serializable {

// geerbt von LayoutManager:
public void layoutContainer(Container target) { … }
public Dimension minimumLayoutSize(Container target) { … }
public Dimension preferredLayoutSize(Container target) { … }

//geerbt von LayoutManager2:
public synchronized float getLayoutAlignmentX(Container target) { … }
public synchronized float getLayoutAlignmentY(Container target) { … }
public synchronized void invalidateLayout(Container target) { … }
public Dimension maximumLayoutSize(Container target) { … }

//Konstruktoren, Felder und private Methoden …
}
```

Abbildung 4-2: Die Klasse BoxLayout

Neben der vollständigen Entkopplung wurde in Kapitel 2 auch schon der zweite Einsatzzweck bzw. das zweite Einsatzziel dargelegt – die Einführung partieller, möglichst kontextspezifischer Interfaces. Hieraus leitet sich das zweite GQM-Ziel (Z2) ab:

Ziel 2 (Z2)

- Zweck: (Sinnvolle) Einschränkung *der*
- Fokus: verfügbaren Features *von*
- Objekt: Interfaces *oder* Klassen[8]
- Blickwinkel: *aus dem* Kontext der Verwendung *heraus gesehen*.

Dies korrespondiert direkt zum zweiten in der Motivation dargelegten Einsatzzweck von Interfaces: In jedem Kontext, in welchem ein Objekt einer Klasse eingesetzt wird, sollten über Interfaces möglichst nur diejenigen Features sichtbar sein, die auch benötigt werden.

Ein Beispiel für dieses zweite Ziel ist die Klasse `WindowAdapter` aus der Java API. Die Klasse implementiert vier Interfaces: `EventListener`, `WindowListener`, `WindowStateListener` und `WindowFocusListener`. Die letzten drei deklarieren dabei `EventListener` als Superinterface; sie stellen drei verschiedene Zugriffsmöglichkeiten auf die Klasse dar.

```
public abstract class WindowAdapter
  implements WindowListener, WindowStateListener,
           WindowFocusListener
{…}
```

Abbildung 4-3: Die Klasse WindowAdapter

Die Klasse verfügt zudem über drei verschiedene Nutzungskontexte. Die Variablen, welche diese Nutzungskontexte aufspannen sind dabei jeweils mit genau dem Interface definiert, welches exakt die im Kontext genutzte Methodenteilmenge enthält. Damit sind die verfügbaren Features der Klasse `WindowAdapter` durch die Interfaces sinnvoll auf die Nutzungskontexte eingeschränkt.

Die beiden nun aufgestellten Ziele werden in den kommenden Abschnitten zunächst in eine Reihe von Fragen verfeinert, aus welchen schlussendlich Metriken entstehen werden.

4.1.3 Ableitung von Fragen

Die Fragen des GQM-Paradigmas werden direkt aus den Zielen entwickelt und dürfen sich nur auf das jeweilige Ziel und dessen beigeordnete Eigenschaften, insbesondere auf das betrachtete Objekt und den Blickwinkel beziehen. Wie oben bereits angemerkt, stehen die GQM-Ziele auf konzeptueller Stufe, während die GQM-Fragen auf funktionaler Stufe stehen. Durch das Beantworten der Fragen soll sich ergeben, ob ein Ziel erreicht ist oder nicht.

Zu Z1:
Um dieses Ziel zu erreichen, muss eine entsprechende Entkopplung des Zugriffs auf Klassen sichergestellt werden – sowohl von Anbieter- (der betrachteten Klasse) als auch von Nutzerseite (den Klassen, welche die Funktionen des Anbieters nutzen). Zunächst ist es wichtig zu erfahren, ob ein entkoppelter Zugriff auf eine Klasse überhaupt möglich ist:

[8] Im Falle einer Klasse sind entsprechende Interfaces einzuführen.

F1: Über wie viele Interfaces ist der Zugriff auf die Klasse möglich?
Die Antwort auf diese Frage lässt erkennen, ob ein entkoppelter Zugriff möglich ist (mindestens ein Interface vorhanden) und gibt eine Übersicht über die Größe der implementierten Hierarchie.

Hierdurch ist nun zwar geklärt, ob ein entkoppelter Zugriff generell möglich ist, jedoch nicht, wie viele Methoden der Klasse tatsächlich zugreifbar sind. Dies führt zu einer zweiten Frage:

F2: Wie viel % der Features einer Klasse sind über Interfaces zugreifbar?
Sind alle öffentlichen Features abgedeckt (100%), muss die Klasse von keinem Klienten direkt referenziert werden[9]. Zwar ist es durchaus möglich, dass ein Klient der Klasse auch mit weniger Features auskommt, hier sind dann jedoch kontextspezifische Interfaces nach Z2 die richtige Wahl, welche in ihrer Summe ebenfalls alle Features der Klasse enthalten können. Eine vollständige Entkopplung – also 100% – deckt auch jede zukünftige Verwendung dieser Klasse ab.

Dass ein Zugriff über Interfaces möglich ist muss leider nicht bedeuten, dass diese auch genutzt werden. Dies wird über die nächste Frage berücksichtigt:

F3: Wie viel % der Nutzung einer Klasse erfolgt über Interfaces?
Kann die Frage F3 positiv – d.h. mit 100% – beantwortet werden, ist der über Interfaces entkoppelte Zugriff von Klienten auf die Klasse Realität. Andere Werte zeigen den Grad an, zu welchem Klienten die Klasse über Interfaces nutzen.

Zu Z2:
Das zweite Ziel ist diffiziler und schwerer zu greifen. Um dieses Ziel zu erreichen, muss die Art der Nutzung von Klassen und Interfaces *in den verschiedenen Kontexten* analysiert werden.

Die Fragen zum zweiten Ziel verstehen sich – wie auch Z2 zu Z1 – als aufbauend zu denen im ersten Ziel; sie sind allerdings auch anwendbar, wenn noch keine vollständige Entkopplung erfolgt ist.

Wie bereits in der Motivation genannt sollen für Kontexte, in denen nur Teilmengen der Features vorhandener Interfaces oder Klassen genutzt werden, eigene Interfaces für die entsprechenden Klassen eingeführt werden. Bei der Entwicklung von Fragen für das Ziel 2 zielt folglich alles auf die Frage, welche Kontexte existieren, wie die Variablen typisiert sind oder sein könnten und ob die verwendeten Interface besser passend gemacht werden können.

F4: Wie viele unterschiedliche genutzte Methodenteilmengen existieren zu einer Klasse?
Diese Frage zielt auf die unterschiedlichen Verwendungen der Klasse ab. Die Beantwortung der Frage kann helfen, mögliche Interfaces zu dieser Klasse zu identifizieren. Da in diesem Zusammenhang auch interessant ist, wie viele Interfaces bereits implementiert sind, wird auch **F1** in die Liste der Fragen für Z2 aufgenommen.

F5: Welche Distanz weisen die verwendeten Typen zu ihren Nutzungskontexten[10] auf?
Betrachtet man den in Kapitel 2 (Abschnitt 2.5) aufgezeichneten Verband, so fällt auf, dass eine Variable prinzipiell mit allen Interfaces deklariert sein kann, welche sich auf dem Pfad von der im Nutzungskontext enthaltenen Methodenteilmenge zur Klasse befinden. Jeder der Knoten in diesem Pfad stellt ein mögliches Interfaces dar, jedoch müssen natürlich nicht alle

[9] Hinweis: Die Beantwortung von Frage F2 enthält implizit auch das Ergebnis von Frage F1. Die Anzahl der implementierten Interfaces ist jedoch im Vergleich zu den Antworten auf Frage F2 und F3 von großem Wert sowie auch für Ziel 2 relevant (siehe unten).

[10] Bei Distanzberechnungen oder Vergleichen von Methodenmengen ist stets die mit dem Kontext verknüpfte Methodenteilmenge gemeint; da diese jedoch nur im Rahmen des Kontexts praktische Bedeutung erlangt ist der Ausdruck Kontext bzw. Nutzungskontext hier passender.

Knoten auch als Interfaces realisiert sein. Im unglücklichsten Fall ist keiner der Knoten als Interface realisiert, sodass für die Variablendeklaration lediglich die Klasse – welche sich, genau wie das totale Interface der Klasse, am Fuß des Verbands befindet – verwendet werden kann.

Ein weiteres Problem ist die Tatsache, dass Knoten zwar mit Interfaces realisiert sein können, die Klasse diese Interfaces jedoch nicht unbedingt implementieren muss. Die Übereinstimmung kann dabei durchaus zufälliger Natur sein; ebenso kann das Interface in der Implementation jedoch übersehen worden sein. In letzterem Fall ist dem Umstand durch eine entsprechende Erweiterung der Klassendeklaration recht schnell abzuhelfen.

Die Entfernung eines durch eine einzelne Variable aufgespannten Nutzungskontexts (und damit natürlich auch der hierin enthaltenen Methodenteilmenge) von dem definierten Typ der Variablen im Verbund bezeichne ich im Folgenden als Kontextabstand. Der Abstand – oder auch die Distanz – ist äquivalent zur Anzahl überschüssiger Methoden, die zwar im Interface vorhanden sind, im Kontext jedoch nicht verwendet werden. Je höher der Abstand, desto unspezifischer ist der verwendete Typ. Ideal ist ein Abstand von Null – hier stimmen Nutzungskontext und verwendeter Typ direkt überein. Der maximal mögliche Abstand ist durch die Anzahl öffentlicher Methoden der Klasse vorgegeben, da diese der Höhe des Baums entspricht. Zu beachten ist, dass ein derartiger Abstand nicht nur an die genutzte Methodenteilmenge im Kontext, sondern auch an die den Kontext aufspannende Variable gebunden ist, da diese den Typ vorgibt.

F6: Welche Distanz kann mit bereits vorhandenen Interfaces erreicht werden?
Nicht immer sind die Variablen, welche Nutzungskontexte aufspannen, mit dem nahesten bereits existierenden Interface typisiert – viele Entwickler setzen oftmals die Klasse als Typ ein, obwohl ein besseres Interface bereits definiert und implementiert ist. Während es natürlich immer möglich ist, neue Interfaces einzufügen, die direkt auf den Nutzungskontext zugeschnitten sind, ist es einfacher, den Typ durch vielleicht bereits bestehende, besser als der aktuelle Typ auf den Kontext passende Interfaces zu ersetzen. Ein geeigneteres Interface als das momentan in einem Kontext verwendete befindet sich im Graph zwischen der im Nutzungskontext enthaltenen Methodenteilmenge und dem aktuell verwendeten Typ.

Die Beantwortung der Frage F6 gibt Aufschluss darüber, ob geeignetere Interfaces vorhanden sind und wenn ja, wie gut diese Interfaces auf den Nutzungskontext passen. Zu beachten ist, dass der bestmögliche Typ hier – anders als bei ACD – an die Methodenteilmenge im Kontext selbst und nicht an die den Kontext aufspannende Variable gebunden ist.

F7: Wie viel % der verfügbaren Features eines Interfaces werden in Kontexten genutzt?
Während der Entwicklung der Klassen und Interfaces kann es durchaus vorkommen, dass zu viele Features einer Klasse in ein Interface übernommen werden. Sind mehrere Interfaces vorhanden, so kann ein Feature auch in mehreren Interfaces vorkommen, es muss jedoch nicht auch überall genutzt werden. Zum Abschluss der Arbeiten an einem Projekt empfiehlt sich daher zu überprüfen, ob ein Interface minimal ist, d.h. ob es überflüssige Methoden enthält.

4.1.4 Definition von Metriken

Jeder Frage wird nun mindestens eine Metrik zugeordnet, wobei eine Metrik durchaus auch mehreren Fragen zugeordnet sein kann (siehe Abbildung 4-1). Mit den Metrikergebnissen lassen sich die den Zielen zugeordneten Fragen beantworten.

Zunächst werden einige Mengen, Funktionen und Relationen eingeführt, die in der späteren Definition der Metriken verwendet werden.

Definiert werden die folgenden Mengen:

- **I** als Menge aller Interfaces im System. Enthalten sind die Interfaces $I_1,...,I_{|I|}$.
- **C** als Menge aller Klassen im System. Enthalten sind die Klassen $C_1,\ldots,C_{|C|}$.
- **T** := **C** ∪ **I** als Menge aller Typen im System.
- **V** als Menge der Variablen im System. Dies schließt Felder, lokale Variablen und Parameter von Methoden mit ein: $V_1,\ldots,V_{|V|}$.

Ebenso werden die folgenden Funktionen definiert:

- *M(C)* und *M(I)* liefern die öffentlichen, nichtstatischen Methoden (keine Konstruktoren) einer Klasse $C \in$ **C** bzw. eines Interfaces $I \in$ **I**: $M_{C,1},\ldots,M_{C,|M(C)|}$ bzw. $M_{I,1},\ldots,M_{I,|M(I)|}$.
- *V(C)* und *V(I)* liefern die Variablen des Gesamtsystems, welche mit *C* bzw. *I* typisiert sind: $V_1,\ldots,V_{|V(C)|}$ bzw. $V_1,\ldots,V_{|V(I)|}$.
- *I*(*C*) liefert die Menge der Interfaces, die C implementiert, also
$$I(C) := \bigcup_{C \le I} \{I\}$$
- *K(V)* liefert die Menge der Methoden, die auf einer Variablen $V \in$ **V** aufgerufen werden (die genutzte Methodenteilmenge im Kontext von *V*).
- *V'*(C*)* liefert die Menge der Variablen, die mit C oder einem Interface von C deklariert sind, also
$$V'(C) := V(C) \cup \bigcup_{I \in I(C)} V(I)$$

Folgende Relationen werden definiert:

- Die Relation „:“: **V** × **T** steht für die Deklaration einer Variable mit einem Typ.
- Die Relation „≤“: **T** × **T** steht für die Subtypenbeziehung, also neben der Subklassen- und Subinterfacebeziehung auch für Implementierung eines Interfaces durch eine Klasse, jeweils auch über mehrerer Ebenen. „≤“ ist reflexiv, transitiv und antisymmetrisch (Halbordnung). Es gilt also insbesondere $T \le T$ für jedes $T \in$ **T**.

Im Folgenden steht *C* für eine Klasse aus **C**, *I* für ein Interface aus **I** und *V* für eine Variable aus **V**.

4.1.4.1 Metrik zu F1: Polymorphic Grade (PG)

Frage 1: Über wie viele Interfaces ist der Zugriff auf die Klasse möglich?

Der entkoppelte Zugriff auf Klassen ist genau dann möglich, wenn Interfaces existieren, die von den Klassen implementiert werden. Folgende Metrik liefert die Antwort auf die Frage F1:

Eingabe: Klasse *C*
Ausgabe: Anzahl Interfaces, die von *C* implementiert werden

Diese Metrik wird bereits in Steimann et al. 2003 definiert und dort als *Polymorphic Grade (PG – Grad der Polymorphie)* der Klasse bezeichnet. Ab einer Zahl von 1 ist ein entkoppelter Zugriff möglich.

Polymorphic Grade ergibt sich direkt als $PG(C) := |I(C)|$.

Hinweis: *I(C)* ist eine Menge und enthält daher keine doppelten Einträge. Doppelte Einträge könnten bei einer Hierarchie wie in Abbildung 4-4 unter Betrachtung von `SomeClass` entstehen. Hierbei wird `SomeInterface` von `SomeClass` direkt implementiert, ebenso werden die

in `SomeInterface` spezifizierten Methoden aber auch von `SomeSuperClass` geerbt, welche ebenfalls `SomeInterface` implementiert. `SomeInterface` darf jedoch nur ein einziges Mal gezählt werden.

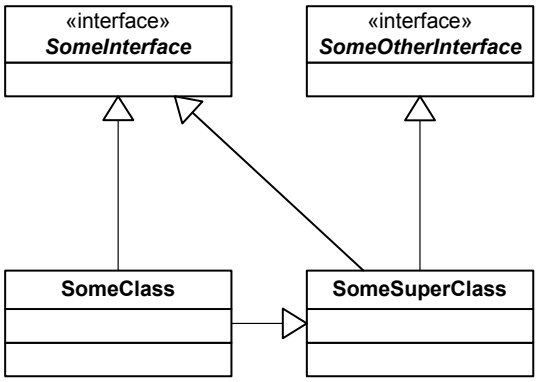


Abbildung 4-4: Mehrfache Vererbung von Interfaces

Die Situation bei `SomeOtherInterface` ist dagegen eindeutig: Dieses Interface wird von `SomeSuperClass` implementiert und direkt vererbt, wird also ebenfalls einmal gezählt. Der *Polymorphic Grade* von `SomeClass` (und nebenbei auch `SomeSuperClass`) ist also 2.

4.1.4.2 Metrik zu F2: Decoupling Accessibility (DA)

Frage 2: Wie viel % der Features einer Klasse sind über Interfaces zugreifbar?

Unter die hier betrachteten Features fallen die Methoden der Klassen. Innere Klassen und Felder sind nicht betroffen, da erstere in Interfaces keine konzeptuelle Entsprechung haben und somit mit Vorsicht zu genießen sind; letztere, da Felder in Interfaces lediglich `final` und `static` deklariert werden können. Eine Metrik, welche die Frage F2 beantwortet, ist folgende:

Eingabe: Klasse *C*
Ausgabe: Anzahl Methoden, die in einem Interface aus *I(C)* definiert sind, geteilt durch die Summe der Anzahl Methoden, die in *C* insgesamt definiert sind

Diese Metrik bezeichne ich als *Decoupling Accessibility (DA – Entkoppelte Zugriffsmöglichkeit)* einer Klasse. 100% geben an, dass alle Methoden der Klasse über Interfaces verfügbar sind; 0%, dass keine verfügbar ist.

Decoupling Accessibility ist wie folgt definiert:

$$DA(C) := \frac{\left| \bigcup_{I \in I(C)} M(I) \right|}{|M(C)|}$$

Ein Beispiel zeigt Abbildung 4-5. Zur Vereinigung aller Mengen *M(I)* gehören hier die Methoden `declaredInInterface()` sowie `declaredInOtherInterface()`; zu *M(C)* die beiden bereits genannten sowie `declaredInClass()`.

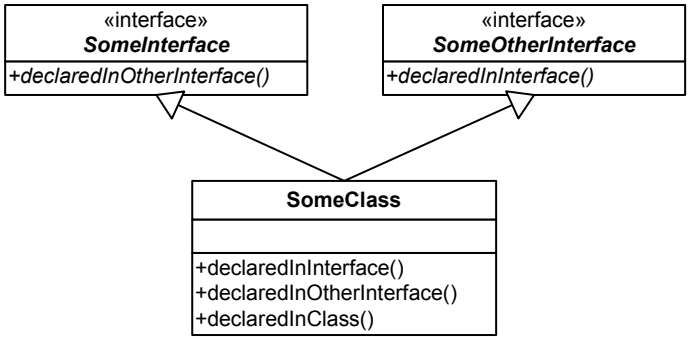


Abbildung 4-5: Vererbung von Methoden

Damit ist *DA* in diesem Fall $DA = \frac{2}{3} = 66{,}6\%$.

Diskussionswürdig ist in diesem wie auch in weiteren Fällen die Frage, ob von `Object` geerbte Methoden (u.A. `equals()`, `hashCode()`, `wait()`) in die Berechnung mit einbezogen werden sollen, oder ob dies nicht gewünscht ist. Eine Berücksichtigung würde den obigen Wert verändern, der Quotient würde (bei neun aus `Object` geerbten Methoden) 91% liefern. Da die Methoden in allen Typen vorhanden sind, hat der Umstand, dass sie in jedem Typ vorhanden sind, keinen Informationsgehalt; er schwächt lediglich das intuitive Verständnis des Metrikergebnisses. Die Empfehlung lautet folglich, die von `Object` geerbten Methoden nicht zu berücksichtigen.

4.1.4.3 Metrik zu F3: Polymorphic Use (PU)

Frage 3: Wie viel % der Nutzung einer Klasse erfolgt über Interfaces?

Die Nutzung einer Klasse über ein Interface korrespondiert zur Deklaration eines Feldes, Methodenparameters oder einer temporären Variable mit einem Interface der Klasse; eine Nutzung der Klasse direkt korrespondiert zur Deklaration derselben Elemente mit der Klasse selbst. Eine Metrik, welche die Frage F3 beantwortet, ist folgende:

Eingabe: Klasse *C*
Ausgabe: Anzahl Variablen mit einem Interface aus *I(C)* definiert geteilt durch Anzahl Variablen mit *C* oder einem Interface aus *I(C)* definiert

Diese Metrik ist bereits in Steimann et al. 2003 definiert und wird dort als *Polymorphic Use (PU – Nutzung der Polymorphie)* einer Klasse bezeichnet. 100% geben an, dass keine Variablen existieren, die direkt mit der Klasse deklariert sind; 0% geben an, dass keine mit Interfaces typisierte Variablen gefunden wurden.

Polymorphic Use ist definiert als

$$PU(C) := \frac{\left| \bigcup_{I \in I(C)} V(I) \right|}{|V'(C)|}$$

4.1.4.4 Metrik zu F4: Number Of Different Access Contexts (NODAS)

Frage 4: Wie viele unterschiedliche genutzte Methodenteilmengen existieren zu einer Klasse?

Um Frage F4 beantworten zu können, ist eine genaue Untersuchung der Variablen nötig, die mit der aktuellen Klasse oder einem Interface dieser Klasse deklariert sind – es ist zu identifizieren, welche Methodenteilmengen in den Kontexten der Variablen existieren. Eine Metrik, welche die Frage F4 beantwortet, ist die folgende:

Eingabe: Klasse *C*
Ausgabe: Anzahl paarweise verschiedener, genutzter Methodenteilmengen der Klasse

Diese Metrik bezeichne ich als N*umber Of Different Access Sets (NODAS – Anzahl unterschiedlicher Zugriffsmengen)* einer Klasse. Ein Ergebnis von 1 zeigt an, dass die Klasse überall in derselben Weise verwendet wird, d.h. die Methodenteilmengen der Nutzungskontexte aller Variablen gleich sind. Je höher das Ergebnis, desto mehr unterschiedliche Methodenteilmengen existieren – und desto mehr Interfaces können u.U. sinnvoll sein und eingeführt werden.

Es ist hierbei zu beachten, dass bei der Berechnung dieser Metrik alle Kontexte, in denen eine Variable mit der Klasse oder einem Interface der Klasse typisiert ist, erfasst werden. Es wird nicht überprüft, inwieweit die Interfaces bereits auf den Kontext zugeschnitten sind; dies ist Aufgabe folgender Metriken.

Number Of Different Access Sets ist definiert als

$$NODAS(C) := \left| \bigcup_{V'(C)} \{K(V)\} \right|$$

Im Folgenden soll neben der obigen Formel auch ein Algorithmus für die Berechnung der NODAS-Metrik skizziert werden. Folgender Algorithmus berechnet für jede Variable den Kontext und fügt die Menge der Methoden des Kontexts in eine Menge von Mengen ein. Die resultierenden Methodenmengen, welche sich in `allUsageSubSets` befinden, sind nicht mehr mit einer Variablendeklaration und deren Kontext verknüpft, es sind reine Methodenteilmengen.

```
Set allUsageSubSets = {};
for each Variable in V'(C) do {
     Set currentMethodSubset = K(Variable);
     allUsageSubSets.add(currentMethodSubset);
}
```

Abbildung 4-6: Algorithmus zur Berechnung der NODAS-Metrik

`allUsageSubSets` enthält nun alle paarweise verschiedenen genutzten Methodenteilmengen von C. Die Kardinalität von `allUsageSubSets` ist das Ergebnis der NODAS-Metrik. Die möglichen Methodenteilmengen bilden die Potenzmenge der Methoden der Klasse; der Wertebereich ist folglich $\left[0,2^{|M(C)|}\right]$.

Natürlich können die Informationen in `allUsageSubSets` – genauso wie die anderer Metriken – auch ausgegeben werden, um dem Entwickler bei der Entscheidung zu helfen, wo

Refactorings nötig sind. Eine derartige Ausgabe ist im *Context Analyzer* (siehe unten) realisiert.

Ein Beispiel für eine solche Berechnung kann anhand der Klasse `NumberFormat` aus der Java API gegeben werden. Innerhalb der API werden zwei Variablen mit `NumberFormat` deklariert, einmal in der inneren Klasse `DoubleRenderer` (`javax.swing.JTable`) und einmal in `DateFormat` (`java.text`) sowie dessen Subklasse `SimpleDateFormat`, welche ja ebenfalls Zugriff auf das (`protected` deklarierte) Feld hat. Die Verwendung in `DoubleRenderer` ist wie folgt:

```
static class DoubleRenderer extends NumberRenderer {
  NumberFormat formatter;
  public DoubleRenderer() { super(); }

  public void setValue(Object value) {
    if (formatter == null) {
      formatter = NumberFormat.getInstance();
    }
    setText((value == null) ? "" : formatter.format(value));
  }
```

Abbildung 4-7: Die Klasse DoubleRenderer

Die Verwendung in `DateFormat` umfasst wesentlich mehr Methoden, hier daher also nur Ausschnitte:

```
public abstract class DateFormat extends Format {

   protected NumberFormat numberFormat;

   public boolean equals(Object obj) {
      …
      return ((…) & numberFormat.equals(other.numberFormat));
      …
   }
   public Object clone() {
     …
     other.numberFormat = (NumberFormat) numberFormat.clone();
     …
   }
   …
}
public class SimpleDateFormat extends DateFormat {
   …
   private void initialize(Locale loc) {
       …
       numberFormat = NumberFormat.getIntegerInstance(loc);
       …
   }
   …
}
```

Abbildung 4-8: Die Klasse DateFormat

Die folgenden Felder spannen dabei Nutzungskontexte der Klasse `NumberFormat` auf:

- das Feld `formatter` in der Klasse `DoubleRenderer`
- das Feld `numberFormat` in der Klasse `DateFormat`

Die Methoden in *M(C)* sind (unter anderem)

- `String format(Object obj)`
- `Object clone()`
- `boolean equals(Object obj)`
- `String format(long number)`
- `void setMaximumIntegerDigits(int newValue)`

Bei der Suche nach `allUsageSubSets` entdeckt der Algorithmus bei der Durchsuchung der ersten Variablen ausschließlich die Verwendung der Methode `String format (Object obj)` – sie bildet das erste UsageSubSet. Bei der zweiten Variablen werden insgesamt acht Methoden, u.A. `clone()`, `equals()`, `format()` und `setMaximumIntegerDigits()` gefunden – sie bilden das zweite UsageSubSet. Der Algorithmus terminiert, NODAS ist 2.

4.1.4.5 Metrik zu F5: Actual Context Distance (ACD)

Frage 5: Welche Distanz weisen die verwendeten Typen zu ihren Nutzungskontexten auf?

Diese Frage zielt bereits relativ genau auf die benötigten Zahlen hin. Gefragt ist nach der tatsächlichen durchschnittlichen Distanz zwischen den Nutzungskontexten einer Klasse und den Typen der Variablen, welche die Kontexte aufspannen. Die Beantwortung dieser Frage erfordert zur Metrik NODAS ähnliche Berechnungen. Je nach Typ und Kontext einer Variable können ganz unterschiedliche Distanzen resultieren; daher wird eine durchschnittliche Distanz über alle Variablen berechnet. Die folgende Metrik übernimmt dies:

Eingabe: Klasse *C*
Ausgabe: Durchschnittliche Distanz der Nutzungskontexte von den Typen der aufspannenden Variablen

Diese Metrik wird von mir als *Actual Context Distance (ACD – Tatsächliche Kontextdistanz)* einer Klasse bezeichnet. Ein Wert von 0 zeigt an, dass alle mit der Klasse oder einem Interface der Klasse typisierten Variablen jeweils die vollständige Menge der Methoden verwenden, welche ihre Typen vorgeben; ein Wert nahe bei 1 gibt an, dass die Distanz zwischen den Kontexten der Variablen und ihren Typen sehr hoch ist, d.h. nur wenige Methoden von einer Vielzahl verfügbarer genutzt werden. Ein Wert von 1 steht für einen leeren Kontext, der hier nicht weiter relevant ist – derartige Kontexte entstehen hauptsächlich bei der Weiterreichung von Variablen durch Methoden; in jedem Fall wird keine Methode genutzt. Entscheidend ist hier jedoch nur der Zielkontext der Variable; nur dieser geht daher in die Berechnung ein.

Actual Context Distance ist wie folgt definiert:

$$ACD(C) := \frac{\sum_{T \in C \cup I(C)} \sum_{V \in V(T)} \frac{\left|M(T)\right| - \left|K(V)\right|}{\left|M(C)\right|}}{\left|V'(C)\right|}$$

Wiederum soll ein Algorithmus zur Berechnung skizziert werden. Die Berechnung von ACD kann in den bei NODAS bereits definierten Algorithmus eingefügt werden. Im bereits skizzierten Teil ist es erforderlich, zu jeder Methodenteilmenge auch die Variablen abzulegen, welche diese Methodenteilmenge verwenden. Im zweiten Teil erfolgt daraus dann die Berechnung der Metrik (siehe Abbildung 4-9).

Im Algorithmus wird die Variable `acdOfClass` definiert. Diese stellt das Resultat der ACD-Metrik dar. Der Wertebereich ist definiert durch das Intervall [0,1[.

Für die Berechnung des jeweiligen ACD-Wertes gibt es mehrere Ansätze. Die hier verwendete ist

acd = (|methodsAvailable|-|methodsUsed|) / |methodsOfClass|

Hierbei werden die tatsächlich im Kontext verwendeten Methoden von den im momentanen Typ vorhandenen abgezogen; dies wird dann durch die Gesamtanzahl der Methoden der Klasse geteilt. Das Verhältnis zur Gesamtgröße wird also angemessen berücksichtigt, wodurch auch größere Abstände zwischen Typ und Kontext einen guten Wert erhalten, wenn die Klasse wesentlich mehr Methoden enthält. Eine andere Möglichkeit ist

acd = (|methodsAvailable|-|methodsUsed|) / |methodsAvailable|

Hierbei wird die Gesamtgröße nicht berücksichtigt; dies führt zu einer Häufung der Werte bei Eins.

```
Set allUsageSubSets = {};
for each Variable in V'(C) do {
      Set currentMethodSubset = K(Variable);
      if (!allUsageSubSets.contains(currentMethodSubset)) {
            currentMethodSubset.
                  addVariable(Variable);
            allUsageSubSets.add(currentMethodSubset);
            }
      else {
            subSet = allUsageSubSets.
                  getSubSet(currentMethodSubset);
            subSet.addVariable(Variable);
      }
}

// Berechnung

cumulatedACD = 0;
numberOfVariables = 0;
for each subSet in allUsageSubSets do {
      methodsUsed = subSet.getMethods();
      for each variable in subSet.getVariables() do {
            numberOfVariables++;
            methodsAvailable
                  = variable.getDefinedType().getDefinedMethods();
            acd   = (|methodsAvailable|-|methodsUsed|)/
                              |methodsOfClass|;
            cumulatedACD += acd;
      }
}

acdOfClass = cumulatedACD / numberOfVariables;
```

Abbildung 4-9: Algorithmus zur Berechnung der ACD-Metrik

4.1.4.6 Metrik zu F6: Best Context Distance (BCD)

Frage 6: Welche Distanz kann mit bereits vorhandenen Interfaces erreicht werden?

Wie bei den Fragen bereits erläutert kann es durchaus bereits vorhandene und „bessere“ Interfaces als die momentan in einer Variablendeklaration verwendeten geben. Ein „besseres“ Interface enthält weniger zum Kontext überschüssige Methoden als das momentan verwendete; d.h. das „bessere“ Interface definiert alle Methoden, die im Kontext verwendet werden, und idealerweise nur diese; jede weitere Methode stellt eine Verschlechterung dar, da sich der Abstand erhöht. Die folgende Metrik zeigt mögliche bessere Werte auf:

Eingabe: Klasse *C*
Ausgabe: Durchschnittlich bestmögliche Distanz der Nutzungskontexte von ihren Typen

Diese Metrik wird von mir als *Best Context Distance (BCD – Beste Kontextdistanz)* einer Klasse bezeichnet. Ein Wert von 0 zeigt an, dass die Kontexte aller mit der Klasse oder einem Interface der Klasse typisierten Variablen jeweils genau die Methoden enthalten, welche die bestmöglichen Typen vorgeben; ein Wert nahe bei 1 gibt an, dass die Distanz zwischen Kontexten und Typen sehr hoch ist, d.h. nur wenige Methoden von einer Vielzahl verfügbarer genutzt wird. Ein Wert von 1 steht wieder für einen leeren Kontext, der hier nicht weiter relevant ist (s.o.). Der Wertebereich liegt zwischen 0 und ACD der Klasse.

Best Context Distance ist wie folgt definiert:

$$BCD(C) := \frac{\sum_{T \in C \cup I(C)} \sum_{V \in V(T)} \min_{U \in I(C) \mid M(U) \supseteq K(V)} \frac{|M(U)| - |K(V)|}{|M(C)|}}{|V'(C)|}$$

Die Berechnung von BCD kann zudem in den bei ACD bereits erweiterten Algorithmus (zweiter Teil) eingefügt werden:

```
cumulatedACD = 0;
cumulatedBCD = 0;
numberOfVariables = 0;

for each subSet in allUsageSubSets do
      methodsUsed = subSet.getUsedMethods();

      bestPossibleType = findBestPossibleType(C,I1..In, subSet);
      methodsAvailabeBest =
      bestPossibleType.getDefinedMethods();
      bcd = (|methodsAvailableBest|-
            |methodsUsed|)/|methodsOfClass|;

      for each variable in subSet.getVariables() do {
            […]
            cumulatedBCD += bcd;
      }
}

acdOfClass = cumulatedACD / numberOfVariables;
bcdOfClass = cumulatedBCD / numberOfVariables;
```

Abbildung 4-10: Algorithmus zur Berechnung der BCD-Metrik

Die Funktion `findBestPossibleType(List of Types, Used Methods)` gibt denjenigen Typ aus `List of Types` zurück, welcher die kleinstmögliche Zahl an zu den in der Menge `Used Methods` überlappenden Methoden aufweist. Wie bereits oben beschrieben hängt der bestmögliche Typ nicht von den Variablen, sondern allein von der im Kontext verwendeten Methodenteilmenge ab; dennoch wird der Typ für alle Variablen (und damit Kontexte) einzeln aufsummiert, denn nur so ist eine Vergleichbarkeit mit der ACD-Metrik gegeben, da die Methodenteilmengen mit den Variablen gewichtet werden. `bcdOfClass` stellt das Resultat der BCD-Metrik dar.

4.1.4.7 Metrik zu F7: Interface Minimization Indicator (IMI)

Frage 7: Wie viel % der verfügbaren Features eines Interfaces werden in Kontexten genutzt?

Die Beantwortung dieser Frage erfordert eine Analyse der aufgerufenen Features – betrachtet werden hier wieder Methoden – eines Interfaces. Sind die im gesamten Projekt verwendeten Features bekannt, können die Features des Interfaces optimiert/minimiert werden. Die folgende Metrik führt diese Analyse durch:

Eingabe: Interface *I*
Ausgabe: Anzahl Methoden, die in mindestens einem Kontext aufgerufen werden geteilt durch Anzahl im Interface verfügbarer Methoden.

Diese Metrik wird von mir als *Interface Minimization Indicator (IMI – Indikator für Schnittstellenminimierung)* eines Interfaces bezeichnet. Ein Ergebnis von 100% zeigt an, dass alle Methoden des Interfaces in mindestens einem Kontext verwendet werden, 0% zeigt an, dass keine einzige verwendet wird.

Interface Minimization Indicator ist wie folgt definiert:

$$IMI(I) := \frac{\left| \bigcup_{T \le I} \bigcup_{V(T)} K(V) \cap M(I) \right|}{\left| M(I) \right|}$$

Zudem kann folgender Algorithmus zur Berechnung genutzt werden:

```
Set usedMethods = {};
for each Variable in ⋃_{T≤I} V(T) do {
      for each Method in M(I) do {
            if (Method called on Variable)
                  usedMethods.addMethod (Method);
      }
}
IMI = usedMethods.size / M(I);
```

Abbildung 4-11: Algorithmus zur Berechnung der IMI-Metrik

Die Ergebnisse der IMI-Metrik können auch für eine Klasse genutzt werden, indem die Werte über die Interfaces der Klasse aufsummiert und anschließend durch PG geteilt werden.

Mit dieser letzten Metrik ist die Definition der Metriken der hier präsentierten Suite abgeschlossen. Eine Übersicht über erreichte Ziele, Fragen und Metriken gibt Abbildung 4-12.

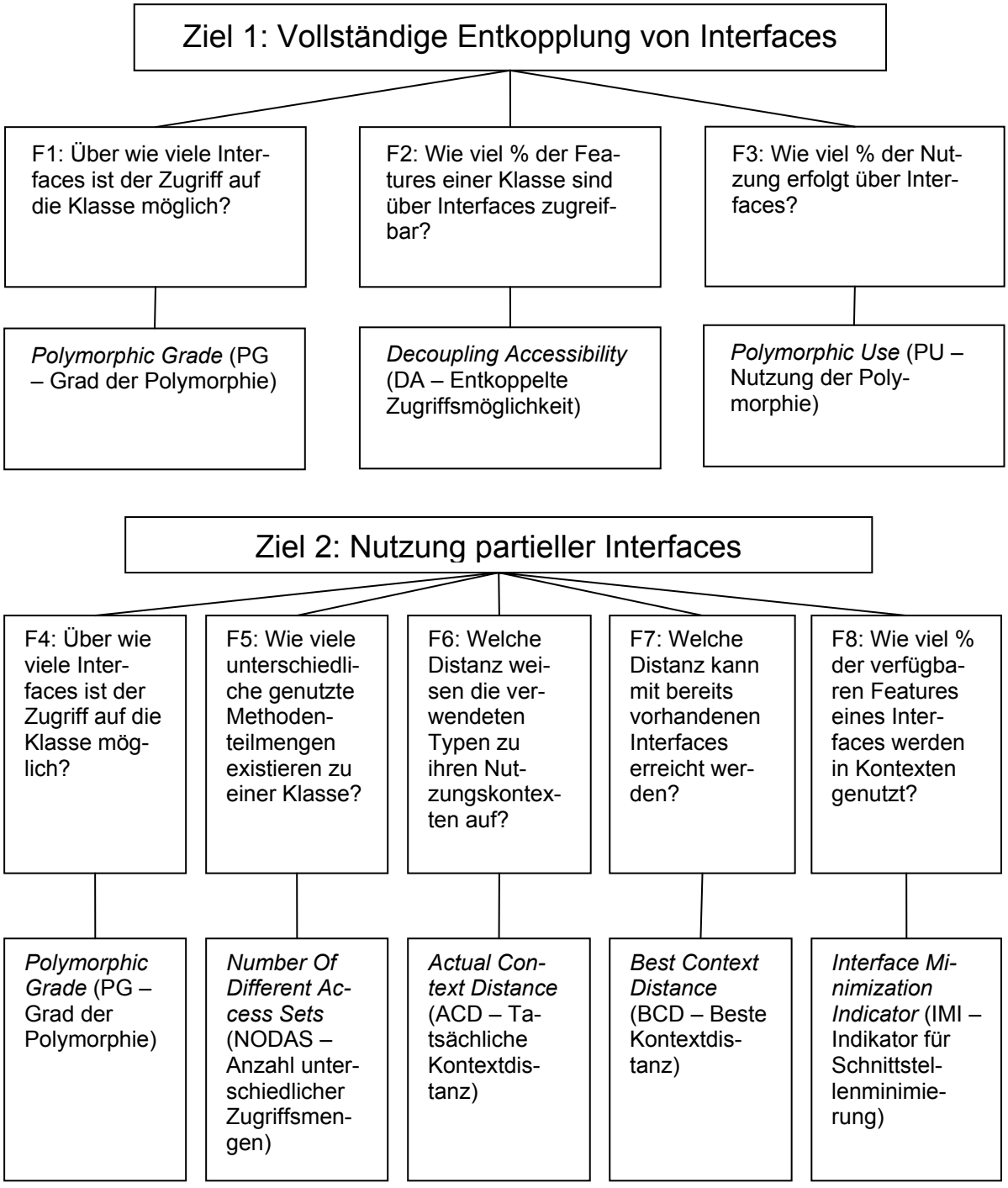


Abbildung 4-12: Übersicht über Ziele, Fragen und Metriken

4.2 Weitere Analyse: Der Context Analyzer

Wie bei der Vorstellung der Kontext-Metriken im vorigen Abschnitt bereits erwähnt ist es möglich, die für die Berechnung der Metriken benötigten Zwischenschritte in allen Detailstufen auch dem Entwickler zur Verfügung zu stellen.

Dabei entsteht eine neue Analysemethode, welche den Entwickler mit umfangreichen Informationen über die Kontexte einer Klasse, aufspannende Variablen, deren Typen und bestmögliche Typen versorgt – eine visuelle Navigationsmöglichkeit durch die Kontexte der Klasse entsteht, welche die Identifikation sinnvoller kontextspezifischer und/oder partieller Interfaces ermöglicht.

Die bei der Suche nach Kontexten und bestmöglichen Typen benötigten Algorithmen sind bereits bei den Metriken NODAS, ACD und BCD vorgestellt worden. Diese Algorithmen können neben der Implementation in Metrikwerkzeugen (vgl. Kapitel 5) auch für die hier genannte Analysemethode eingesetzt werden; implementiert wurden sie daher auch im *Context Analyzer* – ein in Verbindung mit dem Autor eines parallel laufenden Projekts implementiertes Tool (vgl. Meißner 2003), dessen Funktionen hier kurz vorgestellt werden sollen. Hinweise zur Implementation finden sich wie auch für die Metriken im Kapitel 5.

Der Context Analyzer operiert auf einer Klasse und zeigt die zur Klasse gehörigen Kontexte in übersichtlicher Weise an (siehe Abbildung 4-13).

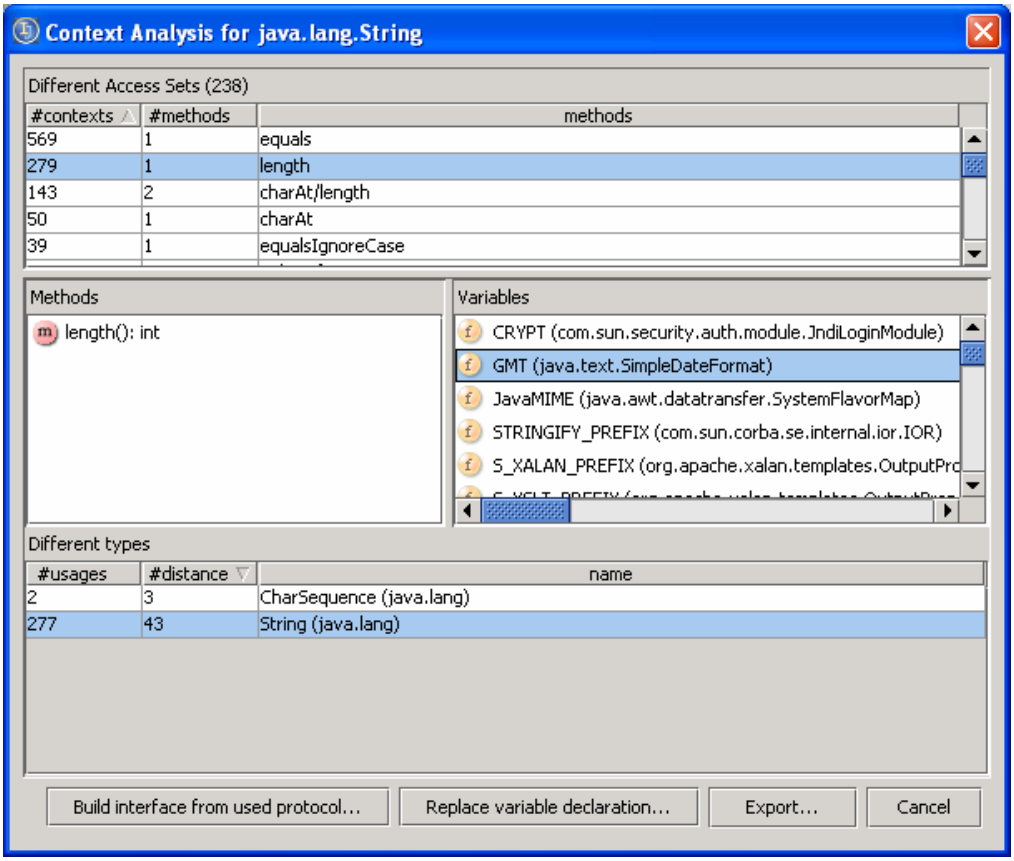


Abbildung 4-13: Der Context Analyzer

Der Context Analyzer hilft bei der Identifikation derjenigen Methodenteilmengen einer Klasse – und deren Kontexte – die am häufigsten aufgerufen werden. Im obigen Beispiel wird die Klasse `String` aus dem Package `java.lang` der Java API analysiert. Im oberen Bereich

des Dialogs ist die Anzahl der Nutzung der jeweiligen Methodenteilmengen in Kontexten angegeben, z.B. 569 Variablen mit Kontexten, in welchen nur die Methode `equals` enthalten ist; 279 Variablen mit Kontexten, in welchen nur die Methode `length` aufgerufen wird und 143 Variablen mit Kontexten, welche die Methoden `charAt` und `length` enthalten.

Die weiteren Listen und Tabellen im obigen Fenster liefern genauere Informationen über die ausgewählten Kontexte. Die Signaturen der Methoden finden sich ebenso wie die Variablen, welche die Kontexte aufspannen; außerdem die in den Variablen verwendeten Typen unter Angabe des Abstands zum Kontext, woraus unmittelbar der bestmögliche Typ im Sinne der BCD-Metrik abzulesen ist: dieser besitzt die geringste Distanz.

Mit diesen Informationen ist es möglich zu entscheiden, wann ein kontextspezifisches Interface Sinn macht (dies ist der Fall, wenn viele Variablen in ihrem Kontext dieselbe Methodenteilmenge verwenden), oder aber ein partielles Interface einzuführen ist (dies ist der Fall, wenn mehrere Methodenteilmengen mit geringer Anzahl an Variablen existieren und die Methodenteilmengen große Ähnlichkeit aufweisen). Aufgrund der angezeigten Methoden kann auch eine Untersuchung möglicher Rollen der Klasse erfolgen. Wie der Dialog zeigt, ist im Anschluss die Ausführung von Refactorings möglich (siehe Meißner 2003).

4.3 Zielerreichung mit den definierten Metriken

Wie Abbildung 4-12 nochmals verdeutlicht sind die Metriken PG, DA und PU zur Analyse der Erreichung des Ziels 1 gedacht, während NODAS, ACD, BCD und IMI – sowie PG als Vergleichsmetrik – über Ziel 2 Aufschluss geben.

Die Erreichung von Ziel 1 ist, wie auch die Definition von Ziel 1 selbst, relativ einfach zu bestimmen. Den Metriken PG, DA und PU können hierzu Schwellenwerte zugeordnet werden.

Das Ziel 1 ist demnach erreicht, wenn pro Klasse

- PG einen Wert von 1 oder größer als 1 annimmt,
- DA einen Wert von 1 erreicht und
- PU einen Wert von 1 erreicht.

Selbstverständlich wird es immer Klassen geben, welche diese Werte nicht erfüllen. Das Konzept der Interfaces – insbesondere unter Verwendung der Konzeptualisierung von Rollen – ist, wenn in den Termini des MVC-Ansatzes (Model-View-Controller, vgl. Krasner/Pope 1988) gesprochen wird, im Modell des Systems zuhause. Zu GUI-Klassen und/oder Utility-Klassen lassen sich nur mit viel Phantasie Rollen identifizieren.

Selbiges gilt in verstärktem Maße für Ziel 2. Die Erreichung dieses Ziels ist diffizil; mehr als in anderen Situationen können Metriken hier nur als Wegweiser dienen, der Entwickler ist die letzte Instanz. Es gibt für die Erweiterung von Klassen mit Interfaces mehr als eine Lösung; Abbildung 2-13 verdeutlichte diese Problematik bereits mit einer Gegenüberstellung kontextspezifisch/partieller und kontextübergreifend/totaler Interfaces. Eine absolute Aussage über nötige Schritte kann nicht getroffen werden; auch die Metrikergebnisse lassen Entscheidungsraum für den Entwickler frei.

Zwischen den vier für Ziel 2 identifizierten Metriken bestehen die in Abbildung 4-14 visualisierten Abhängigkeiten. Es hängt von der jeweiligen Architektur ab, wie viele und welche partiellen Interfaces erforderlich sind. Die Metriken geben den Rahmen für diese Entscheidungen vor.

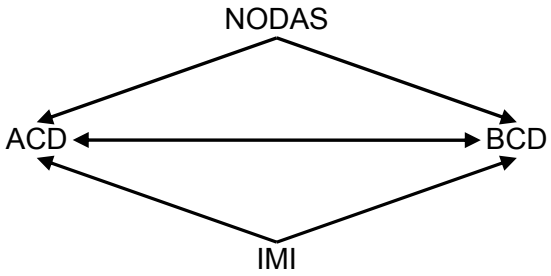


Abbildung 4-14: Gegenseitige Abhängigkeiten zwischen Metriken

Der erste Schritt in einem solchen Entscheidungsprozess ist der Vergleich zwischen den Ergebnissen der Metriken ACD und BCD. Eine starke Abweichung zwischen diesen Werten zeigt an, dass bereits partielle Interfaces dieser Klasse existieren, die allerdings noch darauf warten, in den entsprechenden Variablen eingesetzt zu werden.

Die Ergebnisse der NODAS- und IMI-Metriken helfen dagegen, die absoluten Werte der Metrik ACD (bzw. BCD) einzuschätzen. Eine große Anzahl an paarweise verschiedenen genutzten Methodenteilmengen, wie sie durch NODAS angezeigt wird, rechtfertigt höhere ACD/BCD-Werte, da es bei sehr vielen unterschiedlichen Methodenteilmengen schon aus Übersichtlichkeitsgründen nicht möglich sein wird, für alle ein kontextspezifisches Interface einzuführen. Als Vergleichswert für NODAS kann hier auch die Metrik PG eingesetzt werden, welche die Anzahl der von der Klasse implementierten Interfaces berechnet.

Ein geringer Wert bei IMI – kumuliert über die Interfaces der Klasse – zeigt dagegen viele nichtminimale Interfaces an, welche zu unnötig hohen ACD/BCD-Werten führen, da die Interfaces wesentlich mehr Funktionen anbieten, als in den Kontexten genutzt werden.

Insgesamt zeigen hohe Resultate der ACD/BCD-Metriken einer Klasse fehlende kontextspezifische Interfaces in den Kontexten an. Damit identifizieren die Metriken für Ziel 2 diejenigen Klassen des gesamten Projekts, die einer näheren Untersuchung bzgl. der Einführung von derartigen Interfaces bedürfen – um die genaue Natur der Kontexte zu analysieren, sind jedoch weitere Informationen nötig. Genau diese Informationen liefert der bereits vorgestellte Context Analyzer.

In der empirischen Evaluation (Abschnitt 4.4) wird die Analyse der Ziele Z1 und Z2 für bekannte Frameworks durchgeführt. Hierbei wird nochmals auf die Interpretation der Metrikwerte eingegangen.

4.4 Empirische Daten

Mit der empirischen Evaluation soll aufgezeigt werden, welche Ergebnisse von den oben entwickelten Metriken zu erwarten sind; ebenso sollen Vergleichswerte geschaffen werden. Die vorgelegten Werte wurden unter Verwendung des in Kapitel 5 vorgestellten MetricPlugin für die Entwicklungsumgebung IntelliJ IDEA berechnet. Daten aus vier Open-Source-Frameworks unterschiedlicher Größe wurden berücksichtigt. Folgende Projekte wurden betrachtet:

- *Java API* (1.4.1_02). Die Referenzimplementierung der Java API von SUN ist weit bekannt und erlaubt so eine direkte Wiedererkennung während der Metrikberechnung identifizierter Klassen und Interfaces. Mit 3231 Klassen und 829 Interfaces (ohne innere Typen) liegt die API im Mittelfeld der hier betrachteten Projekte.
- *Eclipse Framework* (2.1.1). Eclipse ist das größte hier betrachtete Projekt mit 6190 Klassen und 1241 Interfaces (ohne innere Typen). Die universelle Plattform zur IDE-

Entwicklung hat insbesondere durch die Referenzimplementierung einer Java-IDE Aufmerksamkeit erregt. Das Eclipse-Framework wurde mit der Win32-Version des SWT untersucht.

- *Joone* (Distributed Environment: 0.5.0; Editor: 0.7.5; Engine: 0.9.9). Als Beispiel eines sehr kleinen Projekts (215 Klassen, 67 Interfaces, ohne innere Typen) dient die „Java Object Oriented Neural Network Engine“; ein Framework zur Erzeugung, zum Training und zum Test neuronaler Netze.
- *Cayenne* (1.0 RC1). Cayenne ist ein objektrelationales Framework zur Unterstützung der Anbindung von Java-Programmen an relationale Datenbanken. Mit 362 Klassen und 105 Interfaces (ohne innere Typen) bewegt es sich im unteren Feld der hier betrachteten Projekte.

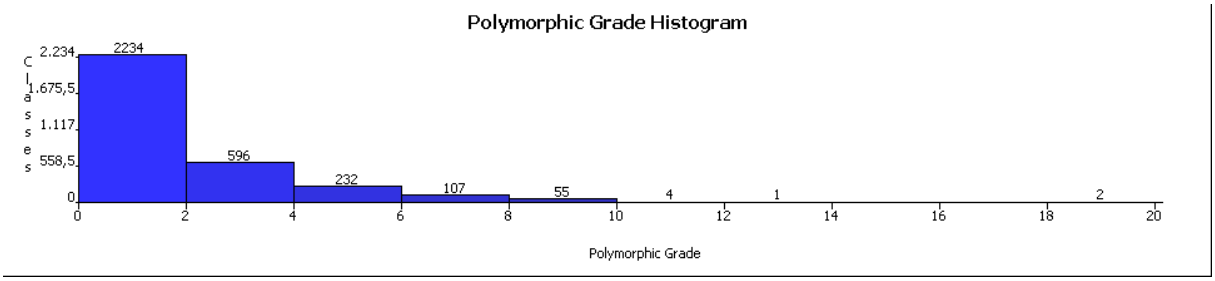
Während es sich bei der Java-API lediglich um eine Klassenbibliothek ohne spezielle Anwendung handelt (jede Anwendung wäre möglich gewesen; jedoch ist die Klassenbibliothek sehr allgemein gehalten und soll daher einzeln untersucht werden), ist in den anderen betrachteten Projekten jeweils eine Referenzimplementierung mit enthalten, was eine Nutzungsanalyse sinnvoll macht. In Eclipse ist die erwähnte Umsetzung einer Java-IDE enthalten; Cayenne enthält einen Editor für relationale Datenbanken; ebenso Joone für neuronale Netze.

4.4.1 Untersuchung von Ziel 1

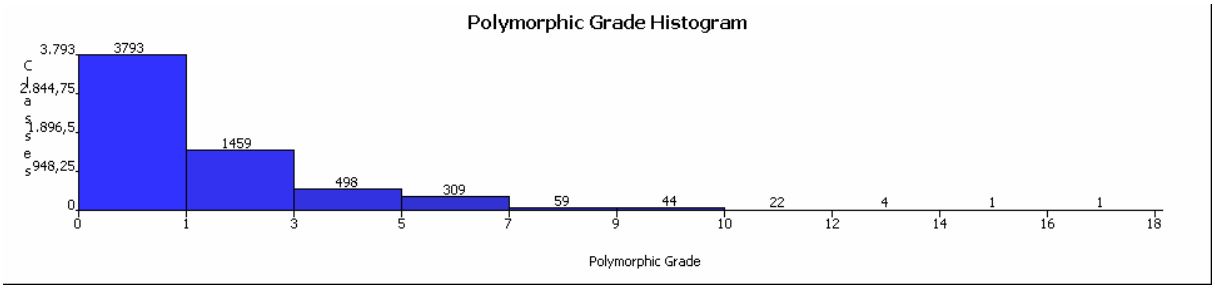
Zunächst soll eine Analyse der Metriken für das erste Ziel des Interfaceparadigmas erstellt werden: Der Nutzung von Interfaces zur vollständigen Entkopplung der Interaktion von Klassen.

Hierfür sind in Kapitel 4 die Metriken Polymorphic Grade (PG), Decoupling Accessibility (DA) sowie Polymorphic Use (PU) entwickelt worden.

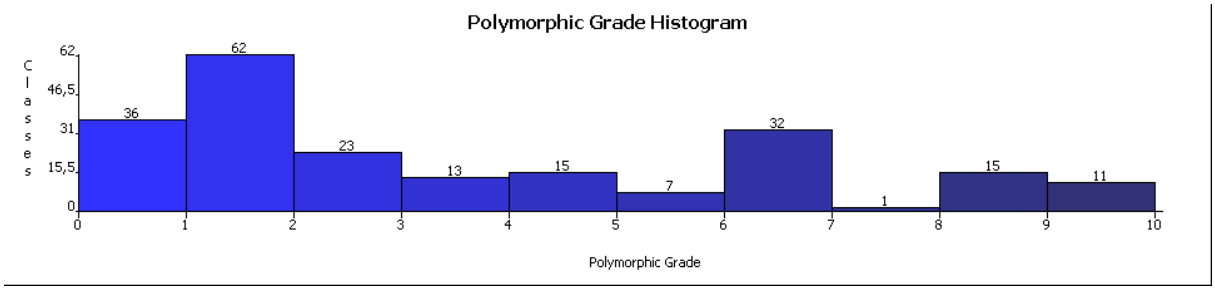
Um Aussagen über die Erreichung des ersten Ziels treffen zu können, ist zunächst von Interesse, wie viele Interfaces von den jeweiligen Klassen implementiert werden; begonnen wird mit der Analyse also bei Polymorphic Grade (Abbildung 4-15).



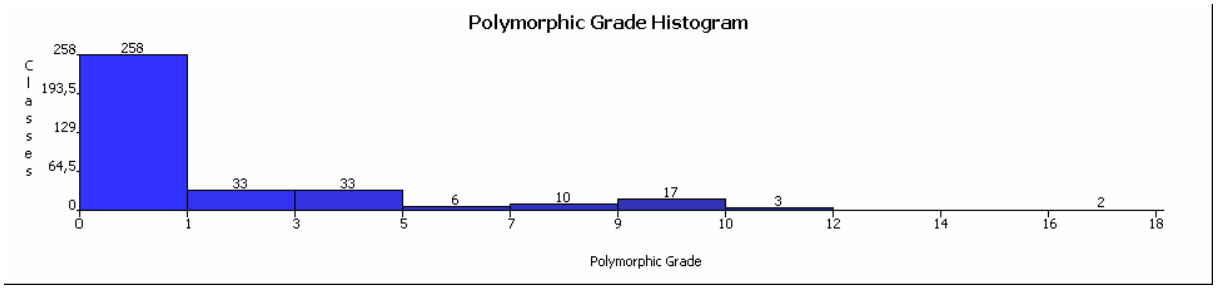
Java API



Eclipse



Joone



Cayenne

Abbildung 4-15: Polymorphic Grade

Bei der Betrachtung der Ergebnisse der Metrik Polymorphic Grade zwischen den verschiedenen Projekten fällt insbesondere die gleiche Verteilung ins Auge: Eine stark abfallende Kurve. Während in allen Projekten (z.T. wesentlich) mehr als 50% der Klassen mindestens ein Interface implementieren, ist die größte Häufung doch bei 0 – keinem einzigen implementierten Interface – zu finden. Ein Wert von PG=1 findet sich allerdings beinahe genauso oft (bei Joone sogar öfter). Erst ab PG=2 fällt die Kurve stark ab:

Tabelle 4-1: Polymorphic Grade der vier Projekte

	Java	Eclipse	Joone	Cayenne
PG=0	35,56%	37,11%	16,74%	38,12%
PG=1	33,58%	24,17%	28,84%	33,15%
PG=2	14,42%	12,73%	10,70%	7,18%
PG≥1	64,44%	62,89%	83,26%	61,88%
PG>2	16,43%	25,99%	43,72%	21,55%

Auffallend ist auch die Höchstgrenze, die bei den drei größeren Projekten bei 18, 18 und 20 liegt, trotz der äußerst unterschiedlichen Anzahl an Klassen und Interfaces. An dieser Stelle scheint eine natürliche Grenze gegeben zu sein; möglich wäre z.B. zu hohe Komplexität und damit ein erschwertes Verständnis bei weiteren implementierten Interfaces oder schlicht das Fehlen sinnvoller Tools zum Management der Interfaces. Der Höchstwert bei Joone ist lediglich 10, allerdings erreichen elf Klassen diesen hohen Wert.

Tabelle 4-2: Maximum und Durchschnitt der PG-Metrik

	Java	Eclipse	Joone	Cayenne
Maximum	20	18	10	18
Durchschnitt	1,44	1,7	3,07	1,93

Im Durchschnitt wird mindestens ein Interface von den Klassen implementiert; im Falle von Joone sogar drei.

Interessante Klassen im Sinne einer erhöhten Anzahl von implementierten Interfaces zeigen die Top 5 der Klassen nach PG absteigend sortiert:

Tabelle 4-3: Top 5 PG / Java API

Name	PG	DA	PU
java.awt.dnd.DnDEventMulticaster	20	1	1
java.awt.AWTEventMulticaster	18	1	0,996
java.beans.beancontext.BeanContextServicesSupport	12	0,763	1
com.sun.corba.se.internal.Activation.ServerManagerImpl	11	0,675	0,996
java.beans.beancontext.BeanContextSupport	11	0,75	1

Tabelle 4-4: Top 5 PG / Eclipse

Name	PG	DA	PU
org.eclipse.debug.internal.ui.views.console.ConsoleViewer	18	0,782	0,988
org.eclipse.debug.internal.ui.views.launch.LaunchView	16	0,75	0,998
org.eclipse.compare.internal.MergeSourceViewer	14	0,683	0,958
org.eclipse.debug.internal.ui.views.expression.ExpressionView	13	0,681	1
org.eclipse.debug.internal.ui.views.variables.VariablesView	13	0,681	0,995

Interessant an dieser Auflistung sind außerdem nicht nur die bereits erwähnten Höchstgrenzen der PG-Metrik trotz deutlich erhöhter Anzahl Klassen bei Eclipse, sondern auch die Werte bei Decoupling Accessibility: Trotz einer sehr hohen Anzahl von Interfaces sind die Klassen nicht vollständig entkoppelt, d.h. die Interfaces decken nicht alle öffentlichen Features der Klassen ab – bei Java sind immerhin zwei Klassen der Top 5 vollständig entkoppelt, bei Eclipse keine der Top 5.

Bei Eclipse fällt auf, dass die PU-Werte in vier Fällen fast, aber nicht ganz bei 1 liegen. Trotz vieler implementierter Interfaces sind in diesen Klassen noch immer Methoden vorhanden, welche in keinem Interface vorhanden sind. Um diese Methoden aufrufen zu können, sind klassentypisierte Variablen verwendet worden: Mit einer Decoupling Accessibility von 1 wäre dies nicht nötig gewesen.

Bei Joone und Cayenne zeigt sich ein ähnliches Bild:

Tabelle 4-5: Top 5 PG / Joone

Name	PG	DA	PU
org.joone.edit.JooneStandardDrawingView	10	0,165	0,909
org.joone.edit.DesiredLayerConnection	9	0,788	1
org.joone.edit.ErrorLayerConnection	9	0,788	1
org.joone.edit.InputLayerConnection	9	0,788	1
org.joone.edit.InputPluginConnection	9	0,788	1

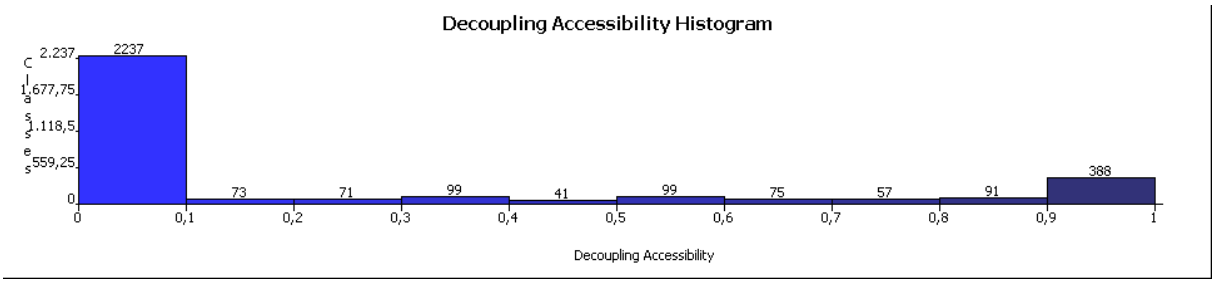
Tabelle 4-6: Top 5 PG / Cayenne

Name	PG	DA	PU
org.objectstyle.cayenne.modeler.BrowseView	18	0,083	0,981
org.objectstyle.cayenne.modeler.Editor	17	0,075	0,721
org.objectstyle.cayenne.modeler.datamap.DbRelationshipPane	12	0,052	0,975
org.objectstyle.cayenne.modeler.datamap.ObjRelationshipPane	12	0,052	0,975
org.objectstyle.cayenne.modeler.EditorView	12	0,039	0,932

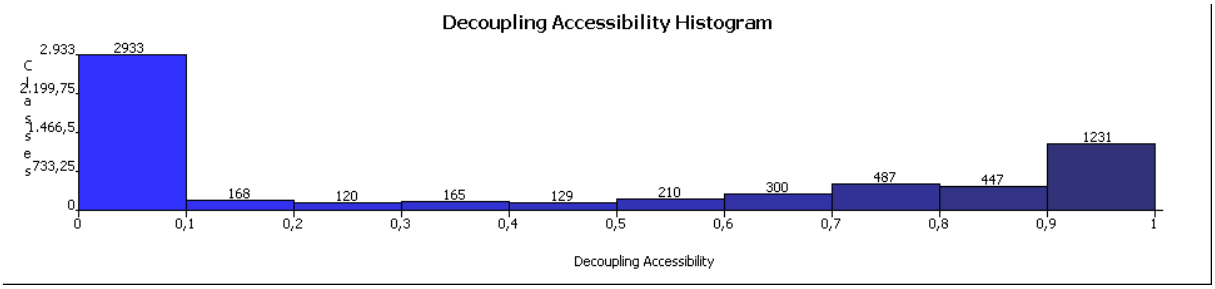
Die DA-Werte bei vier der fünf Klassen von Joone zeigen einen Wert von 0,788. Dies rührt von der Ableitung von einer gemeinsamen Superklasse (`LayerConnection`) her, in welcher einige nicht in Interfaces verfügbare Methoden deklariert sind, die jedoch entweder gar nicht, nur intern in der Klasse `LayerConnection` und ihrer Subklassen oder durch direkte Casts auf `LayerConnection` genutzt werden; daher wird trotzdem ein PU-Wert von 1 erreicht. Die Methoden könnten alle `protected`, z.T. sogar `private` deklariert werden.

Im Falle von Cayenne resultieren die äußerst geringen DA-Werte aus einer direkten Ableitung aller Klassen der Top 5 von Swing-Klassen, welche eine hohe Anzahl von Methoden mit sich bringen.

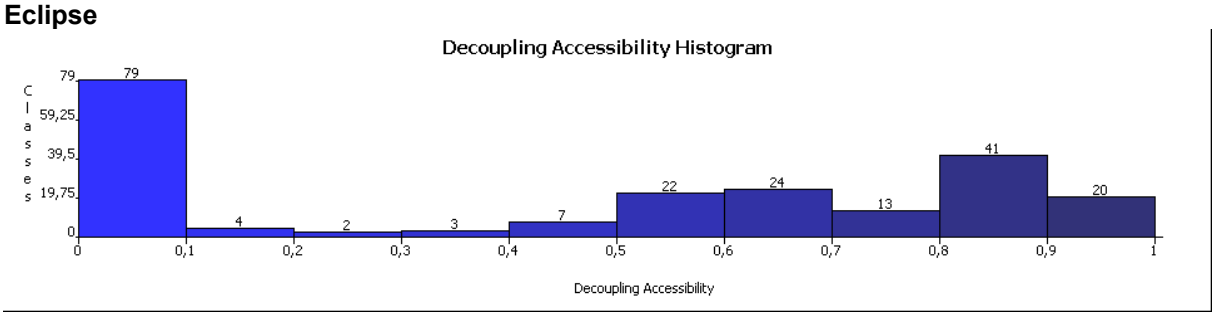
Eine interessante Frage ist hierbei nun, welche und wie viele Klassen tatsächlich eine Decoupling Accessibility von 1 erreichen. Bei dieser Metrik – welche nur auf dem Intervall [0,1] definiert ist – zeigt der Verlauf eine stark abfallende Kurve mit z.T. leichtem Akzent auf der rechten Seite:



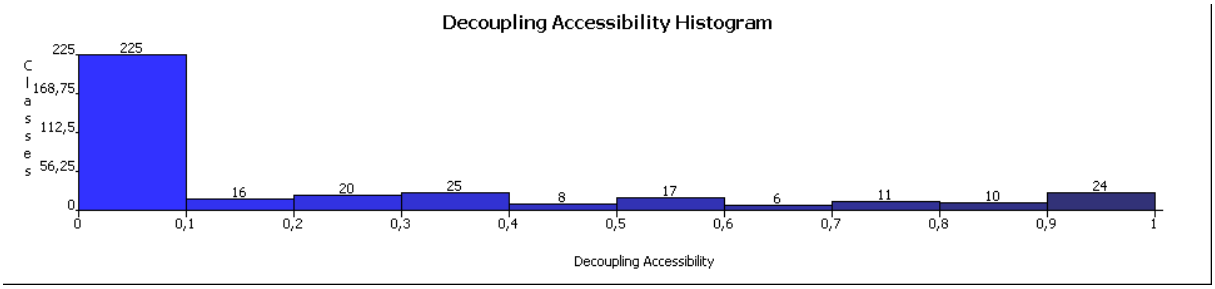
Java API



Eclipse



Joone



Cayenne

Abbildung 4-16: Decoupling Accessibility

Ein großer Akzent auf 0 und 1 könnte bedeuten, dass Klassen je entweder für die vollständige Nutzung via Interfaces entwickelt wurden – oder das genaue Gegenteil der Fall ist, also dem Interfacegedanken keinerlei Beachtung geschenkt wurde. Eine Häufung bei 1 zeigt in jedem Fall eine vermehrte Auseinandersetzung mit dem Interfacegedanken.

Bei Joone und Cayenne, also den kleineren Projekten, ist dagegen der Mittelwert sehr hoch, d.h. hier wurde der vollständigen Entkopplung nicht so hohe Aufmerksamkeit geschenkt wie in den großen Paketen. Die Häufung auf einem DA von 1 ist zwar bei den größeren Projek-

ten im Sinne einer Erfüllung von Z1 zu begrüßen, jedoch sind die absoluten Zahlen noch immer gering.

Tabelle 4-7: DA-Werte der vier Projekte

	Java	Eclipse	Joone	Cayenne
DA=0	60,45%	43,12%	23,26%	49,45%
DA=1	10,21%	15,53%	6,51%	4,42%
0<DA<1	29,34%	41,36%	70,23%	46,13%

Um auf die oben gestellte Frage – welche Klassen einen DA-Wert von 1 erreichen – zurückzukommen, zeigt folgende Tabelle den Durchschnittswert der implementierten Interfaces aller Klassen der betrachteten Projekte mit einem DA-Wert von 1:

Tabelle 4-8: Durchschnittlicher PG der Klassen mit DA=1

Projekt	Klassen mit DA=1	Durchschn. PG
Java API	330 (10,21%)	1,62
Eclipse	961 (15,53%)	1,96
Joone	14 (6,51%)	1,71
Cayenne	16 (4,42%)	1,18

Dies zeigt, dass die am besten entkoppelten Klassen eine im Gegensatz zu den Ausreißern relativ geringe Anzahl von Interfaces aufweisen. Zwischen den Ergebnissen der Metriken Decoupling Accessibility und Polymorphic Grade bei DA=1 lässt sich im Allgemeinen jedoch kein Zusammenhang herstellen: Klassen mit einer DA von 1 implementieren völlig unterschiedliche Anzahlen von Interfaces, von 1 bis 18 ist alles vorhanden.

Ein Scatterplot des Zusammenhangs zwischen DA und PG über die gesamten Daten in allen Projekten zeigt Abbildung 4-17.

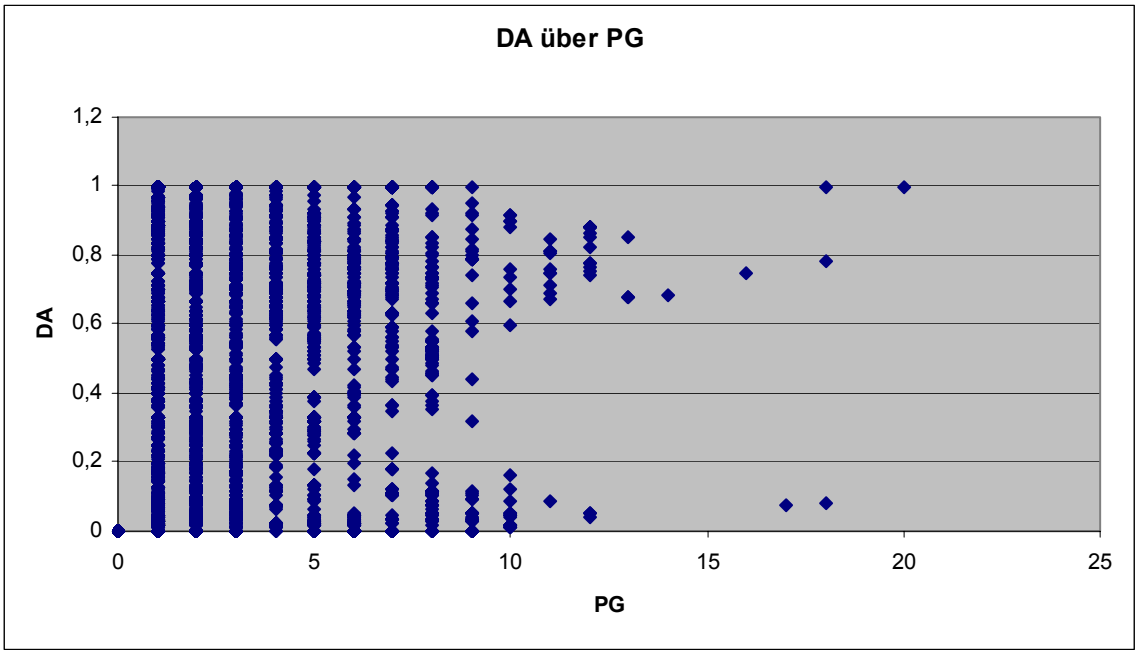
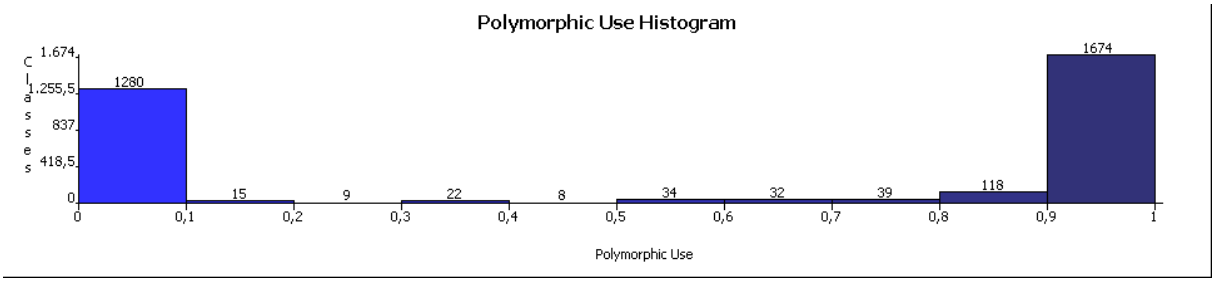


Abbildung 4-17: Scatterplot PG/DA

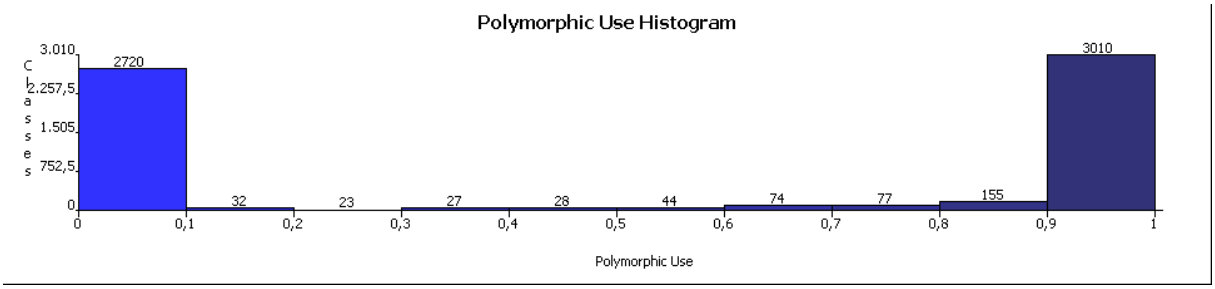
Nun ist geklärt, inwieweit die Klassen über Interfaces genutzt werden können, jedoch nicht, ob diese Nutzung auch erfolgt. Hierfür wird Polymorphic Use zu Rate gezogen.

Die bereits bei Polymorphic Grade gezeigten Tabellen der Top 5 Klassen nach PG absteigend sortiert zeigen an, dass Klassen mit sehr hohen PG-Zahlen auch über hohe PU-Werte verfügen; d.h. viele zur Klasse zuweisungskompatible Variablen sind mit Interfaces der Klasse deklariert. Besonders bei den großen Projekten finden sich Zahlen von 1 oder sehr nahe bei 1; bis auf zwei Ausreißer gilt dies auch für die Zahlen der kleineren Projekte.

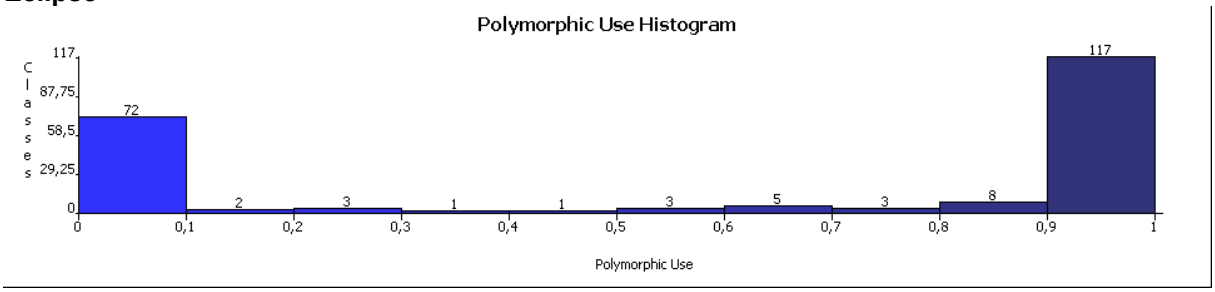
Um auch den Umkehrschluss zu untersuchen, soll nun die allgemeine Verteilung von Klassen im Bezug auf ihren PU-Wert dargelegt werden. Bei Polymorphic Use ist eine U-Verteilung zu beobachten; diese ist sogar sehr ausgeprägt:



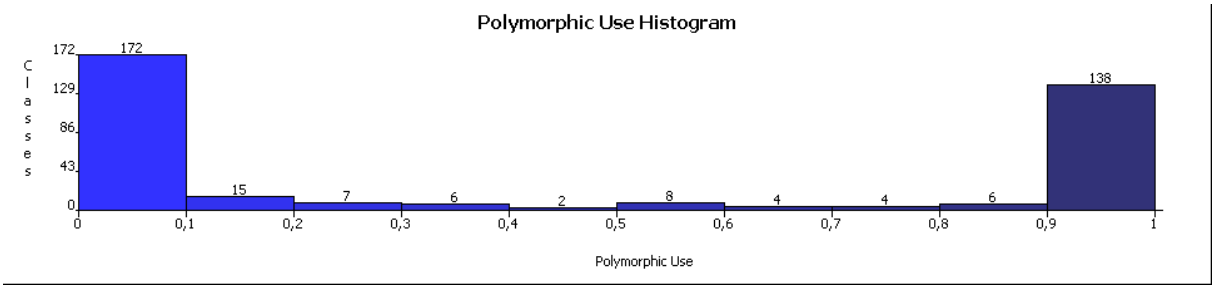
Java API



Eclipse



Joone



Cayenne

Abbildung 4-18: Polymorphic Use

Dies lässt wiederum darauf schließen, dass Klassen je entweder direkt zur Nutzung via Interfaces konzipiert und, dies ist nun durch PU geklärt, auch so verwendet worden sind – oder dies nicht der Fall ist; in diesem Fall wurde auch kein Interface verwendet (hohe Prozentsätze bei 0 und 1). Das Mittelfeld bleibt hierbei bis auf einen Fall hinter den Extrema zurück (Tabelle 4-9).

Tabelle 4-9: PU der vier Projekte

	Java	Eclipse	Joone	Cayenne
PU=0	39,28%	43,51%	33,02%	44,48%
PU=1	31,04%	28,35%	41,86%	23,76%
0<PU<1	29,68%	28,14%	25,12%	31,77%

Unter den Klassen mit PU=1 sind auch die Glanzlichter der PG-Metrik vertreten, so z.B. die Klasse `DnDEventMulticaster` aus der Java API (PG=20) oder `ExpressionView` aus Eclipse (PG=13); jedoch werden auch viele dieser PG-„hochdotierten“ Klassen nicht vollständig entkoppelt verwendet, so z.B. `AWTEventMulticaster` (PG=18) und `ServerManagerImpl` (PG=11) aus der Java API oder `ConsoleView` (PG=18) und `LaunchView` (PG=16) aus Eclipse (allerdings liegen diese Klassen sehr nahe bei einem PU-Wert von 1). Diese Zahlen legen nahe, dass der vollständigen Entkopplung noch nicht die Aufmerksamkeit geschenkt wird, die ihr zusteht.

Folgende Tabelle zeigt den Zusammenhang zwischen Klassen mit einer PU von 1 und den implementierten Interfaces dieser Klasse. Die Werte – sowohl die Anzahl der Klassen als auch der durchschnittliche PG-Wert – liegen allesamt höher als die korrespondierenden Durchschnittswerte aus Tabelle 4-8, in welcher Klassen mit einem DA-Wert von 1 untersucht wurden. Die tatsächliche Nutzung von Variablen mit Interfaces (PU) ist also für weit mehr Klassen Realität als eine vollständige Verfügbarkeit aller Methoden in Interfaces (DA); die Anzahl der implementierten Interfaces liegt bei der Nutzung allerdings ebenfalls höher.

Tabelle 4-10: Durchschnittlicher PG der Klassen mit PU=1

Projekt	Klassen mit PU=1	Durchschn. PG
Java API	1003 (31,04%)	1,96
Eclipse	1755 (28,35%)	2,60
Joone	90 (41,86%)	3,07
Cayenne	86 (23,76%)	1,93

Auch zwischen PG und PU lässt sich über die gesamten Daten aller Projekte ein Scatterplot aufzeichnen (Abbildung 4-19):

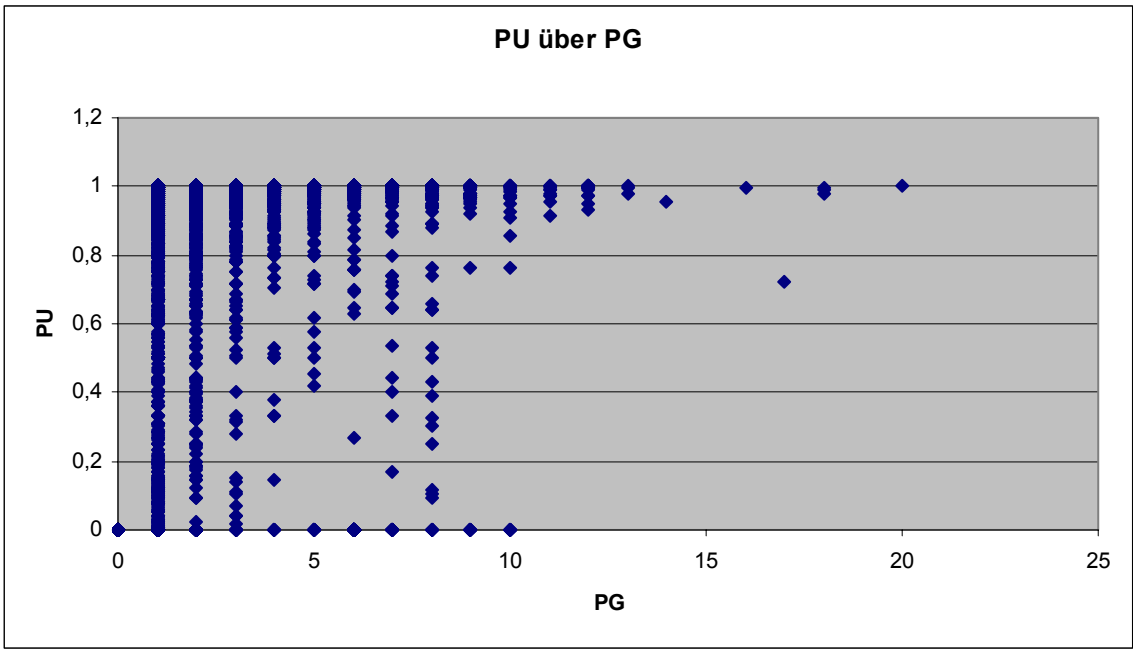


Abbildung 4-19: Scatterplot PG/PU

Damit sind die Untersuchungen zu Ziel 1 abgeschlossen – immerhin wird in über 50% der Klassen zumindest teilweiser Gebrauch der implementierten Interfaces in Variablendeklarationen gemacht (Prozentsatz PU>0), allerdings nur in bis zu 42% vollständig (PU=1). In allen Projekten lässt sich also noch einiges verbessern.

Die hier gezeigten Ergebnisse bestätigen die in vorigen Studien (vgl. Steimann et al. 2003) schon getroffenen Aussagen, dass die Interfacenutzung in Java-OO-Projekten noch sehr schwach ausgeprägt ist. Durch die Ergebnisse der Metrik Polymorphic Use ist dies nun auch für die Nutzung von Klassen bestätigt worden.

Zusammengefasst lässt sich sagen, dass jede der Metriken einen anderen Aspekt des Problems beleuchtet und so zur Erkenntnis über das Ziel 1 beiträgt. Die Metrik Decoupling Accessibility enthält zwar implizit bereits das Ziel der Entkopplung aus Anbieterseite, jedoch liefert die Metrik Polymorphic Grade nicht zu unterschätzende Aureißerwerte für die Analyse. Polymorphic Use letztendlich beleuchtet die Nutzerseite und ist so als echter Indikator für die in Klienten erreichte Entkopplung äußerst hilfreich.

4.4.2 Untersuchung von Ziel 2

Ziel 2, also die Nutzung von Interfaces als partielle, möglichst kontextspezifische Schnittstellenbeschreibungen, umfasst die Metriken Number of Different Access Sets (NODAS), Actual Context Distance (ACD), Best Context Distance (BCD) sowie den Interface Minimization Indicator (IMI) – zum Vergleich wird auch die bereits behandelte Metrik Polymorphic Grade (PG) herangezogen. Zusätzlich zu diesen Metriken wurde für Ziel 2 der Context Analyzer entwickelt, ein Tool zur genaueren Untersuchung der Kontexte einer Klasse, welcher hier ebenfalls eingesetzt werden soll.

Von besonderem Interesse ist dabei, wie in Abschnitt 4.2 dargelegt, ein Vergleich der Metriken ACD und BCD; das in Kapitel 5 vorgestellte MetricPlugin berechnet daher auch die Differenz.

Die NODAS-Metrik ist hervorragend geeignet, um Ausreißerklassen zu identifizieren und separat mit dem Context Analyzer zu untersuchen. NODAS und IMI sind wie in Abschnitt 4.2 dargelegt ebenso geeignet, die Höhe der ACD/BCD-Werte zu bewerten.

In allen betrachteten Projekten ist die Differenz zwischen Actual und Best Context Distance für die meisten Klassen sehr gering; beispielhaft sei hier der Verlauf des Deltas der Java API gezeigt (Abbildung 4-20). Dies ist zunächst ein gutes Zeichen: Die Variablen der Kontexte wurden meistens mit den besten verfügbaren Typen deklariert. Allerdings ist hierdurch noch nicht geklärt, wie gut die besten verfügbaren – und damit auch die tatsächlich verwendeten – Interfaces auf die Kontexte passen.

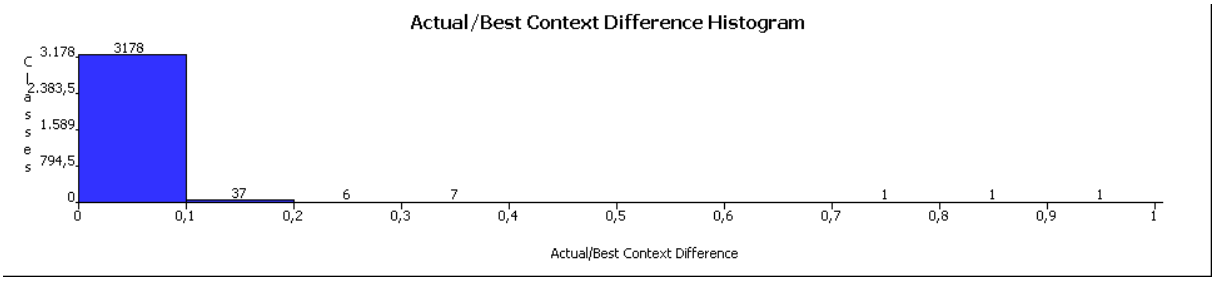
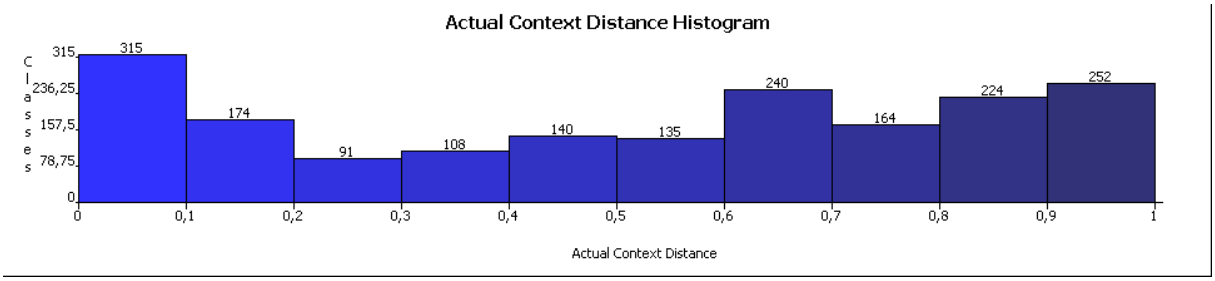


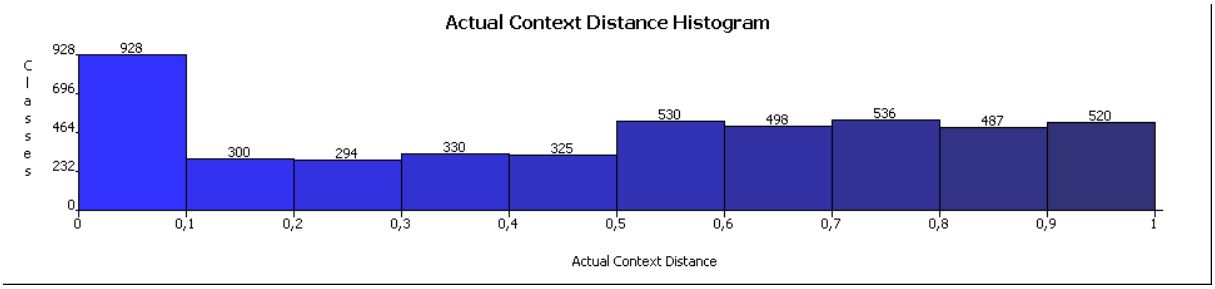
Abbildung 4-20: Delta zwischen ACD und BCD / Java API

Folgende Abbildung zeigt, dass sich die Verteilung der ACD/BCD-Werte über das gesamte Intervall hinzieht. Bei den größeren betrachteten Projekten liegt die absolut größte Häufung

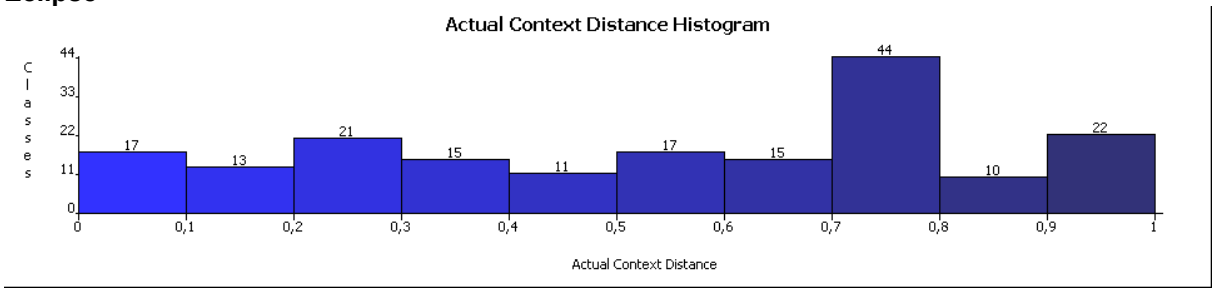
allerdings eher bei 0 als bei 1; dies zeigt an, dass die deklarierten Typen keine allzu großen Distanzen zu ihren Kontexten aufweisen:



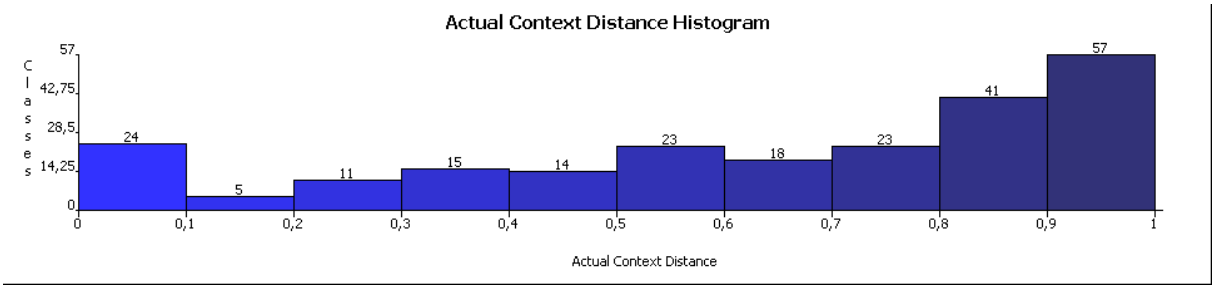
Java API



Eclipse



Joone



Cayenne

Abbildung 4-21: Actual Context Distance

Bei den kleineren Projekten, insbesondere bei Cayenne, zeigt sich allerdings ein anderes Bild: Hier ist eine Häufung gegen die 1 hin zu beobachten, was anzeigt, dass viele Kontexte sehr hohe Distanzen zu ihren Typen aufweisen.

Die Mehrzahl der betrachteten Kontexte liegt allerdings bei allen Projekten auf der rechten Seite des Diagramms, d.h. der Kontextabstand ist i.d.R. größer, z.T. sogar deutlich größer als 0, was auf eine schlechte Übereinstimmung zwischen Kontext und Typ in den meisten Fällen schließen lässt.

Folgende Tabelle zeigt die Top 5 Klassen im Bezug auf das Delta zwischen ACD und BCD (Eclipse):

Tabelle 4-11: Top 5 Delta ACD/BCD / Eclipse

Name	NODAS	ACD	BCD	Delta
org.eclipse.jdi.internal.PrimitiveTypeImpl	1	0,984	0,049	0,934
org.eclipse.jface.dialogs.ProgressMonitorDialog	4	0,661	0,045	0,617
org.eclipse.jdt.internal.core.index.impl.IndexedFile	5	0,745	0,248	0,497
org.eclipse.update.internal.core.FeatureExecutableFactory	1	0,5	0,1	0,4
org.eclipse.jface.action.SubMenuManager	65	0,801	0,449	0,352

Das hohe Delta zeigt an, dass für die Kontexte z.T. wesentlich bessere Typen existieren. Die Verwendung dieser Typen würde das System allgemeiner verwendbar machen. Eine Analyse mittels des Context Analyzers offenbart – beispielhaft – folgende Situation:

PrimitiveTypeImpl
`PrimitiveTypeImpl` implementiert das Interface `Type`. Es existieren zwei gleichartige Kontexte, in welchen nur Methoden aufgerufen werden, die bereits in `Type` angelegt sind; jedoch sind die Variablen mit `PrimitiveTypeImpl` typisiert.

IndexedFile
`IndexedFile` implementiert das Interface `IQueryResult`. Dieses Interface deckt jedoch nur eine der identifizierten Methodenteilmengen ab; allerdings wird diese in den Kontexten von 18 Variablen genutzt. Die Distanz würde in diesen Kontexten sogar auf 0 fallen.

Hinweis: Diese Klassen dienen lediglich als Beispiel. Wie bereits oben angedeutet muss die Verwendung eines Interfaces an den genannten Stellen nicht unbedingt sinnvoll sein; dies kann nur ein mit dem Projekt eng vertrauter Entwickler entscheiden.

Auch in der Java API finden sich einige wenige hohe bis sehr hohe Deltas, wie Tabelle 4-12 zeigt:

Tabelle 4-12: Top 5 Delta ACD/BCD / Java API

Name	NODAS	ACD	BCD	Delta
javax.security.auth.kerberos.KerberosTicket	1	0,957	0,043	0,913
java.awt.SentEvent	1	0,833	0	0,833
javax.security.auth.kerberos.KerberosKey	1	0,889	0,111	0,778
java.awt.image.ColorConvertOp	2	0,833	0,444	0,389
java.nio.channels.FileChannel	3	0,609	0,236	0,373

Wie zuvor ist auch hier die Anzahl der Nutzungsteilmengen begrenzt; die Situation gleicht der bereits in Eclipse erläuterten. Ein besonderer Fall findet sich bei `SentEvent`: Hier wird das Interface `ActiveEvent` implementiert; eine einzelne Variable im Projekt verwendet den Typ `SentEvent`, obwohl alle Methoden bereits in `ActiveEvent` verfügbar sind; mehr noch: Es handelt sich um exakt dieselbe Menge. Daher hat BCD den (idealen) Wert 0.

In den beiden kleineren Projekten Cayenne und Joone steigt das Delta nicht über 0,5, diese sollen hier nicht näher beleuchtet werden.

Interessante Klassen, an denen die Einführung von partiellen, möglichst kontextspezifischen Interfaces sinnvoll ist, lassen sich durch die Ausreißer in der NODAS-Metrik identifizieren. Wie bereits Polymorphic Grade aus dem ersten Ziel zeigt auch NODAS eine steil abfallende Kurve, d.h. viele Klassen mit wenigen verschiedenen verwendeten Methodenteilmengen und wenige Klassen mit sehr hoher Anzahl an verschiedenen verwendeten Methodenteilmengen. Folgende Abbildung zeigt das Histogramm der Java API, das eine sehr hohe Ähnlichkeit mit den drei übrigen Projekten besitzt.

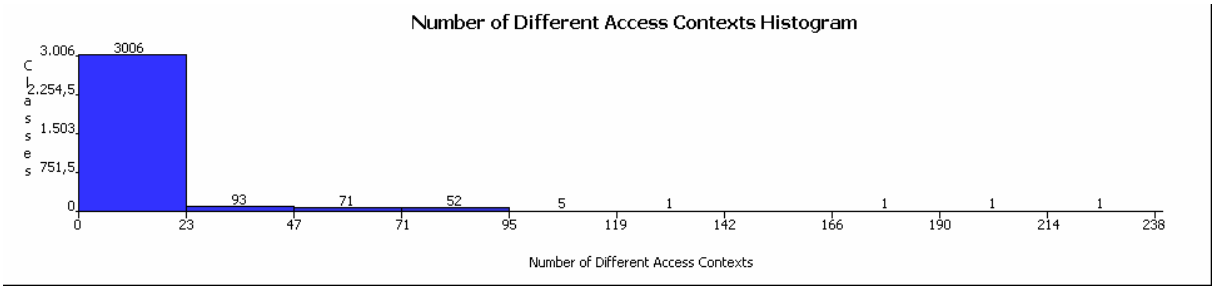


Abbildung 4-22: NODAS / Java API

Folgende Tabellen zeigen die Liste der Top 5 Klassen im Bezug auf NODAS der vier betrachteten Projekte.

Tabelle 4-13: Top 5 NODAS / Java API

Name	NODAS	ACD	BCD	IMI	PG
java.lang.String	238	0,941	0,734	1	3
java.util.Vector	197	0,782	0,729	1	5
java.awt.Component	173	0,986	0,971	1	3
java.awt.Graphics	136	0,946	0,946	-	0
javax.swing.JComponent	116	0,978	0,968	1	3

Diese Tabelle zeigt einige äußerst bekannte Klassen der Java API: Die Klassen `String`, `Vector`, `Graphics` und `(J)Component`. Die unterschiedlichste Nutzung zeigt die Klasse `String` mit 238 unterschiedlichen genutzten Methodenteilmengen. Die IMI-Metrik-Werte zeigen an, dass in vier Fällen die implementierten Interfaces keine Methoden besitzen, die nirgendwo genutzt werden – die Minimierung der Interfaces würde die ACD/BCD-Werte also nicht verringern. Die Klasse `Graphics` implementiert kein Interface, daher existiert auch kein IMI-Wert.

Die Werte von ACD und BCD liegen allerdings – bis auf einen Fall – sehr nahe bei 1, was ein Anzeiger für fehlende partielle Interfaces ist, obwohl dies durch die hohen NODAS-Werte abgeschwächt wird. Die vielen genutzten Methodenteilmengen lassen jedoch auch viel Raum für die Einführung neuer kontextspezifischer/partieller Interfaces. Der Context Analyzer zeigt folgende Ergebnisse für `String` und `Vector`:

Tabelle 4-14: Kontextanalyse für String / Java API

Methoden	Variablen	Tatsächliche Typen	Beste Typen
equals	568	String	Jedes Interface[11]
length	279	String (277), CharSequence (2)	CharSequence (3)
length, charAt	143	String	CharSequence (2)

[11] Anzumerken ist hierbei, dass jedes Interface per Ableitung von `Object` bereits die Methode „`equals()`“ enthält – ein Sonderfall der Programmiersprache Java. Der Context Analyzer grenzt die von `Object` geerbten Methoden bei der Berechnung jedoch explizit aus, da die meisten Metriken durch deren Berücksichtigung verfälscht würden. Dadurch ist es dem Context Analyzer in diesem – seltenen – Fall nicht möglich zu erkennen, dass `CharSequence` auch für den ersten Kontext der bessere Typ wäre; tatsächlich ist für diesen Kontext sogar jedes Interface passend.

Tabelle 4-15: Kontextanalyse für Vector / Java API

Methoden	Variablen	Tatsächliche Typen	Beste Typen
size, elementAt	100	Vector	Vector (108)
size, elementAt, addElement	61	Vector	Vector (107)
addElement	49	Vector	Vector (109)

Die Tabellen zeigen die drei jeweils meistgenutzten Methodenteilmengen der Klassen `String` und `Vector` – die meistgenutzte Methodenteilmenge wird in 568 Kontexten genutzt und enthält lediglich `equals`. Die Tabellen zeigen außerdem die tatsächlich in den Kontexten genutzten Typen („Tatsächliche Typen“) mit der Häufigkeit der Nutzung (z.B. im Falle der zweiten genutzten Methodenteilmenge bei `String` 277 Variablen, die mit `String` sowie zwei Variablen, die mit `CharSequence` typisiert sind) sowie die für die Kontexte am besten passenden Typen („Beste Typen“) und die Distanz zu diesen Typen (was der Anzahl überschüssiger Methoden entspricht – z.B. drei Methoden bei `CharSequence` in der zweiten genutzten Methodenteilmenge).

Im Falle von `String` ist ein partielles Interface namens `CharSequence`, welches u.A. die Methoden `length` und `charAt` enthält, bereits verfügbar, wird jedoch nur in 2 von 990 Kontexten eingesetzt (möglich sind alle drei Methodenteilmengen der Tabelle). Im Falle von `Vector` wäre ein Interface `DynamicArray` mit den in den drei meistgenutzten Methodenteilmengen genutzten Methoden (`size`, `elementAt`, `addElement`) machbar, ist jedoch (noch) nicht implementiert.

Interessant ist zudem, dass `String` und `Vector` scheinbar gleichartig genutzt werden, und zwar wie dynamische Arrays: In jedem Fall werden Methoden genutzt, welche die Größe zurückliefern (`size`/`length`) oder Elemente herausgeben (`elementAt`/`charAt`). Das könnte die Einführung eines gemeinsamen Interfaces `Arrayable` rechtfertigen. Für dessen Verwendung müssten die Methoden allerdings umbenannt werden, denn die Methode zur Bestimmung der Größe heißt im einen Fall `length`, im anderen `size`; gleiches gilt den Zugriff auf ein bestimmtes Element. Die Erkennung derartiger Muster in Klassen ist im Falle unterschiedlicher Namen bzw. Signaturen der Methoden jedoch nicht automatisch durchführbar.

Tabelle 4-16: Top 5 NODAS / Eclipse

Name	NODAS	ACD	BCD	IMI	PG
org.eclipse.core.internal.resources.Project	313	0,502	0,454	0,976	6
org.eclipse.core.internal.resources.File	262	0,55	0,422	0,981	6
org.eclipse.help.internal.util.FastStack	260	0,677	0,454	0,935	5
org.eclipse.jdt.internal.core.BinaryType	236	0,414	0,336	0,998	7
org.eclipse.core.internal.resources.Folder	236	0,514	0,496	0,975	6

Auffallend bei Eclipse ist die Herkunft der Klassen mit hohen NODAS-Werten: Drei von fünf stammen aus dem Package `org.eclipse.core.internal`. Auch hier zeigen sich wieder Abweichungen zwischen ACD und BCD; der PG-Wert liegt über dem Durchschnitt. Die folgenden Tabellen zeigen eine Analyse der Klasse `Project` und der Klasse `FastStack`.

Tabelle 4-17: Kontextanalyse für Project / Eclipse

Methoden	Variablen	Tatsächliche Typen	Beste Typen
getFullPath	145	Project (5), IContainer (16), IProject (26), IResource (98)	IResource (51)
getType	123	IContainer (9), IResource (114)	IResource (51)
getName	88	IContainer (2), IResource (39), IProject (47)	IResource (51)

Bei dieser Klasse wäre es möglich, alle Kontexte direkt mit `IResource` zu deklarieren – oder aber – wie bei `Vector` aus der Java API – ein neues partielles Interface einzuführen, welches z.B. die ersten drei Methodenteilmengen abdeckt.

Tabelle 4-18: Kontextanalyse für FastStack / Eclipse

Methoden	Variablen	Tatsächliche Typen	Beste Typen
add	495	Collection (26), List (469)	Collection (14)
add, size, toArray	332	Collection (12), List (320)	Collection (12)
iterator	250	Collection (27), List (223)	Collection (14)

Die Klasse `FastStack` wird anders als die zuvor betrachteten Klassen in keiner Variablendeklaration verwendet – in jedem Kontext werden nur die Interfaces `Collection` und `List` eingesetzt. Dennoch lässt sich noch etwas verbessern: In den meisten Fällen ist `List` im Einsatz, obwohl `Collection` ausreichend gewesen wäre.

Folgende Tabelle zeigt die Top 5 im Bezug auf NODAS des Projekts Cayenne:

Tabelle 4-19: Top 5 NODAS / Cayenne

Name	NODAS	ACD	BCD	IMI	PG
org.objectstyle.cayenne.access.ToManyList	65	0,723	0,512	0,939	3
org.obj[…]nne.modeler.control.EventController	65	0,952	0,952	0,143	1
org.objectstyle.cayenne.access.IncrementalFaultList	63	0,746	0,527	0,908	2
org.objectstyle.cayenne.map.DbRelationship	48	0,916	0,859	1	1
org.objectstyle.cayenne.map.ObjEntity	48	0,958	0,794	1	1

Bei Cayenne zeigen ACD und BCD Abweichungen, was darauf schließen lässt, dass sich die Situation bereits mit „Bordmitteln“ verbessern lässt. IMI liegt bis auf einen Fall nahe oder direkt bei 1, d.h. die absoluten Werte von ACD/BCD lassen sich durch die Änderung der Interfaces nur wenig verringern.

Folgende Tabellen zeigen die Kontexte der ersten beiden Klassen:

Tabelle 4-20: Kontextanalyse für ToManyList / Cayenne

Methoden	Variablen	Tatsächliche Typen	Beste Typen
add	87	List (83), Collection (4)	Collection (14)
iterator	74	List (60), Collection (14)	Collection (14)
get, size	46	List	List (23)

Tabelle 4-21: Kontextanalyse für EventController / Cayenne

Methoden	Variablen	Tatsächliche Typen	Beste Typen
fireDomainEvent, getCurrentDataEvent	2	EventController	EventController (72)
fireObjEntityEvent, getCurrentDataMap, getCurrentObjEntity	2	EventController	EventController (71)
addDataMap, fireDataDisplayEvent, fireDataMapEvent, getCurrentDataDomain, getCurrentDataNode	1	EventController	EventController (69)

Wie bereits `FastStack` bei Eclipse ist `ToManyList` kein klassischer Fall: i.d.R. kommen hohe Werte bei ACD/BCD zustande, da Variablen mit der Klasse statt einem Interface defi-

niert sind. Der Context Analyzer zeigt bei dieser das Interface `List` (und damit auch `Collection`) implementierenden Klasse jedoch eine hohe Anzahl Variablen, die mit `List` deklariert sind, obwohl `Collection` angemessener (und ausreichend) gewesen wäre. Auch hier wiederum hätte das Programm modularer gestaltet werden können.

Die zweite Tabelle zeigt die genutzten Methodenteilmengen der Klasse `EventController`. Sie implementiert lediglich ein Interface, welches in keinem Kontext genutzt wird, aber auch in keinem Kontext genutzt werden kann (ACD=BCD) – es handelt sich um das Interface `Controller` (ebenfalls aus der Cayenne-Hierarchie). Bei dieser Klasse könnte über die Verwendung von Server-Interfaces nachgedacht werden, jedoch sind die genutzten Methodenteilmengen sehr verschieden; zudem wird jede Methodenteilmenge in nur wenigen Kontexten genutzt (die Maximalanzahl ist wie oben ersichtlich zwei) – in diesem Fall wäre es besser, auf kontextspezifische Interfaces zu verzichten. Ggf. kann über partielle Interfaces – losgelöst vom Kontext – nachgedacht werden.

Tabelle 4-22: Top 5 NODAS / Joone

Name	NODAS	ACD	BCD	IMI	PG
org.joone.engine.Layer	35	0,863	0,549	0,717	6
org.joone.edit.jedit.JEditTextArea	29	0,983	0,983	0,333	3
org.joone.io.StreamInputSynapse	29	0,478	0,272	0,804	6
org.joone.engine.learning.TeachingSynapse	27	0,436	0,432	0,942	4
org.joone.engine.learning.TeacherSynapse	26	0,247	0,22	0,961	6

Im Falle von Joone zeigt die Kontextanalyse bei der Klasse `Layer` eine Implementation des Interfaces `NeuralLayer`, welches die ersten drei verwendeten Methodenteilmengen abdeckt, in den Typendeklarationen aber nicht verwendet wird. Im Falle der Klasse `StreamInputSynapse` sind gleich zwei Interfaces möglich, die zum Teil, aber nicht in jedem Fall genutzt werden.

Tabelle 4-23: Kontextanalyse für Layer / Joone

Methoden	Anzahl	Tatsächliche Typen	Beste Typen
addInputSynapse	5	Layer	NeuralLayer (22)
addOutputSynapse	5	Layer	NeuralLayer (22)
setRows	3	Layer	NeuralLayer (22)

Tabelle 4-24: Kontextanalyse für StreamInputSynapse / Joone

Methoden	Anzahl	Tatsächliche Typen	Beste Typen
getName	8	StreamInputSynapse (4), OutputPatternListener (3), InputPatternListener (1)	OutputPatternListener (10)
setMonitor	3	StreamInputSynapse (2), OutputPatternListener (1)	OutputPatternListener (10)
isInputFull	3	StreamInputSynapse (2), InputPatternListener (1)	InputPatternListener (13)

Die oben bereits gezeigten Diagramme zu ACD zeigen zwar eine Häufung der Klassen nahe bei 0, jedoch liegt die Mehrzahl der Klassen weit über 0. Dies führt zu der Frage, welche Klassen einen ACD/BCD-Wert von 0 aufweisen. Folgende Tabelle zeigt eine Übersicht über die NODAS/PG-Zahlen dieser Klassen:

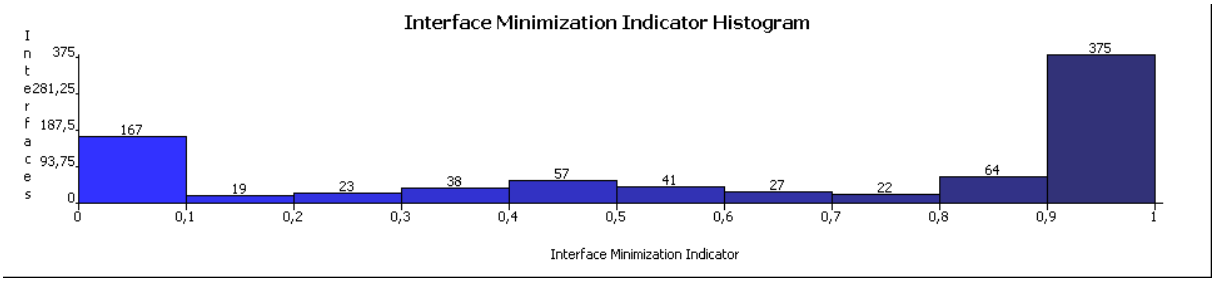
Tabelle 4-25: Klassen mit ACD=0

Projekt	Klassen mit ACD= 0	Durchschn. NODAS	Durchschn. PG
Java API	198 (6,13%)	1,16	1,37
Eclipse	548 (8,85%)	1,15	1,66
Joone	17 (7,91%)	1	2,88
Cayenne	23 (6,35%)	1,52	3,73

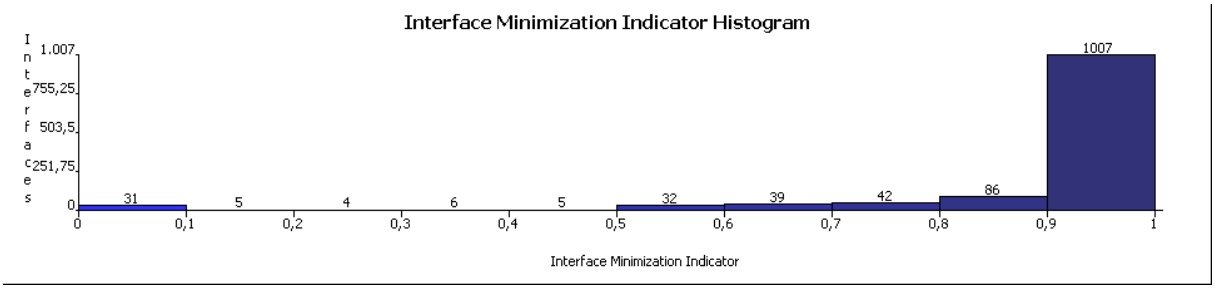
Diese Zahlen zeigen an, dass Klassen mit einem ACD-Wert von 0 nicht besonders unterschiedlich genutzt werden. Bei den großen Projekten ist die Anzahl implementierter Interfaces gering – vermutlich sind zwar wenige Interfaces implementiert worden, diese dann jedoch vollständig genutzt worden. Bei den kleineren Projekten sind durchschnittlich mehr Interfaces beteiligt.

Nach Betrachtung der Metriken ACD, BCD und NODAS soll zum Abschluss der Fokus auf die IMI-Metrik gelegt werden. Die IMI-Werte hängen stark von den betrachteten Projekten ab: Da hier eine globale Nutzungsanalyse durchgeführt wird, müssen alle Klassen, welche ein Interface jemals nutzen werden, betrachtet werden. Dies wird nur selten möglich sein, daher sollten die Resultate der IMI-Metrik stets nur als Indikator gewertet werden: Weicht der Wert sehr stark von 1 ab, sollte das entsprechende Interface untersucht werden[12].

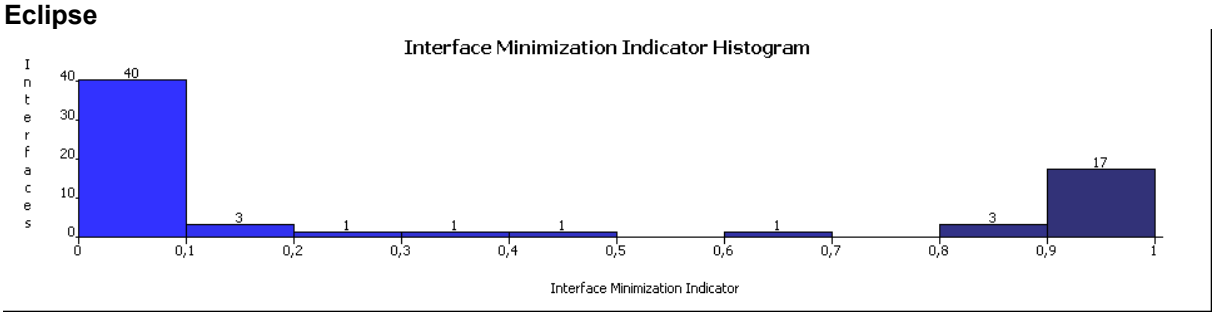
Die folgenden Histogramme zeigen die IMI-Übersicht der betrachteten Projekte (Abbildung 4-23).



Java API

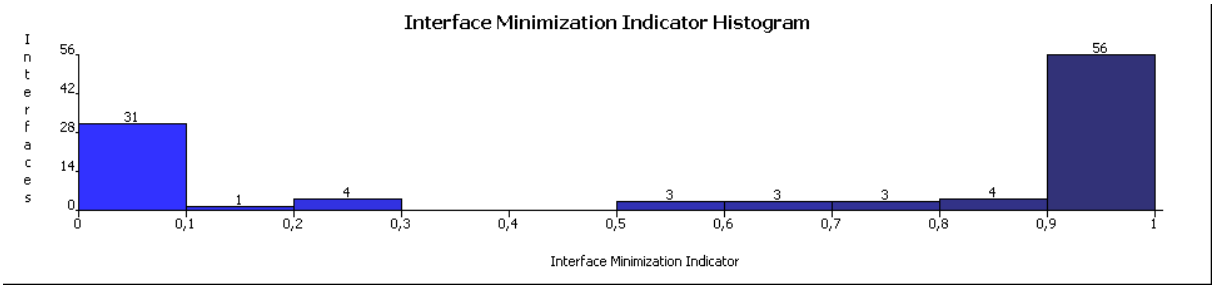


Eclipse



Joone

[12] Die IMI-Metrik teilt dieses Interpretationsproblem mit vielen weiteren, auf Fan-In-Faktoren basierenden Metriken.



Cayenne

Abbildung 4-23: Interface Minimization Indicator

Bei IMI ist ein großer Unterschied zwischen den kleineren und den größeren Projekten zu entdecken: Während viele Interfaces der Java API und Eclipse einen Wert von 1 oder nahe 1 aufweisen, gilt dies für die kleineren Projekte nicht. Allerdings ist es durchaus möglich, dass die Interfaces der kleineren Projekte schlicht mehr Funktionen für externe Anwendungen enthalten, die durchaus sinnvoll sind, jedoch in den enthaltenen Anwendungen nicht verwendet werden.

Damit ist die Untersuchung zu Ziel 2 abgeschlossen. Die hohen ACD- und BCD-Werte zeigen an, dass kontextspezifische Interfaces in den meisten Fällen noch nicht Realität sind. Weitere Forschung und praktische Tests sind hier jedoch nötig, welche Entscheidungshilfen für die Implementation derartiger Interfaces, insbesondere im Vergleich mit kontextübergreifenden, partiellen Interfaces geben.

Auch hier wiederum lässt sich sagen, dass jede Metrik einen Aspekt des Problems beleuchtet. Wie die bereits in Abschnitt 4.3 dargelegte Abhängigkeitsgrafik zeigt, stehen sich insbesondere ACD und BCD gegenüber, während NODAS und IMI helfen, deren absolute Werte einzuschätzen. Insbesondere NODAS dient, wie zuvor Polymorphic Grade, der Ausreißeranalyse. Bei der Untersuchung von IMI muss stets bedacht werden, dass alle das Projekt nutzende Klassen mit in die Untersuchung einbezogen werden müssen; ansonsten sind die Ergebnisse unvollständig. Dies ist insbesondere bei Frameworks ohne umfassende Beispielanwendung der Fall.

4.5 Vergleich mit anderen Suiten

In diesem Abschnitt werden die hier definierten Metriken mit ähnlichen oder äquivalenten Metriken aus der Literatur verglichen.

Meine Literatursuche hat ergeben, dass sich bislang nur die in Steimann et al. 2003 vorgestellte Metrik-Suite explizit des Interface-Phänomens angenommen hat. Ebenso existieren nur wenige Metriken, welche Interface- und Implementationshierarchie trennen und deren Nahtstellen untersuchen; diese sind in Kapitel 3 bereits vorgestellt worden.

In vielen Fällen können die hier vorgestellten Metriken allerdings durch Kombination mehrerer, bereits vorhandener Metriken und besonderem Fokus auf die Interfacehierarchie simuliert werden.

4.5.1 Polymorphic Grade

Bei der Metrik Polymorphic Grade, welche aus Steimann et al. 2003 übernommen wurde, handelt es sich um die Zählung der implementierten Interfaces einer Klasse. An dieser Stelle findet also ein Hierarchiewechsel statt: Ausgehend von einer Klasse wird die Verbindung zur Interfacehierarchie untersucht. Für die Analyse der Klassenhierarchie existieren bereits Met-

riken mit demselben Zweck: Die wohl am besten geeignete Metrik ist der bekannte Fan-In-Faktor der Klasse, angewandt auf die Vererbung: *Class Inheritance Fan-In* berechnet die Anzahl der Superklassen einer Klasse.

Nicht vergleichbar ist z.B. die Metrik DIT (*Depth of Inheritance Tree*) der CK-Suite Hier wird lediglich die Maximallänge des Baums betrachtet; da Interface-Hierarchien jedoch i.d.R. Mehrfachvererbung erlauben ist diese Zahl i.A. nicht zu der Anzahl der Superinterfaces äquivalent.

Patenaude et al. 1999 haben die Verbindung zwischen den Hierarchien durch ihre Metrik CIIfIn geschaffen. Die Metrik mit dem Titel *Class interface extension fan-in* entspricht genau der Metrik Polymorphic Grade.

4.5.2 Decoupling Accessibility

Die in dieser Arbeit neu definierte Metrik Decoupling Accessibility berechnet das Verhältnis von in Interfaces vorgeschriebenen zu allen öffentlichen Methoden der Klasse.

Während eine Metrik in dieser Art noch nicht existiert, ist eine Nachbildung über bereits vorhandene Metriken möglich. So existiert die Metrik NMI (Number of Methods Inherited), die in Lorenz/Kidd 1994 definiert wird. Lorenz/Kidd verwenden diese Metrik, um Implementationsvererbung, also die Vererbung von Verhalten anzuzeigen. Einer Anwendung auf die Interfacehierarchie – also zur Anzeige von Protokollvererbung – steht jedoch nichts im Weg. Wird an dieser Stelle auch ein Hierarchiewechsel durchgeführt – werden also die *Number of Methods Inherited* aller Interfaces einer Klasse berechnet – so entspricht dies dem Zähler der Gleichung zur Berechnung der Decoupling Accessibility.

Der Nenner wiederum ist schlicht die in vielen Quellen, u.A. auch als NIM von Lorenz/Kidd definierte Metrik *Number of Methods*, wobei hier nur öffentliche, nichtstatische Methoden berücksichtigt werden dürfen.

4.5.3 Polymorphic Use

Die aus Steimann et al. 2003 übernommene Metrik Polymorphic Use, welche das Verhältnis mit einem Interface einer Klasse typisierter Variablen zu allen mit der Klasse oder einem Interface der Klasse typisierten Variablen berechnet, kann einerseits recht einfach nachgebildet werden, andererseits existiert bereits eine ähnliche Definition.

Hilfreich für die Nachbildung ist erneut die bekannte Fan-In-Messung, welche (wie in Kapitel 3 bereits beschrieben) in vielen unterschiedlichen Formen vorkommt. Relevant ist hier zum einen der Kopplungs-Fan-In der Klasse – die Anzahl der Variablen, welche mit der Klasse deklariert sind – zum anderen der Kopplungs-Fan-In aller Interfaces der Klasse – die Anzahl der Variablen, welche mit einem der Interfaces deklariert sind.

Aus diesen beiden Fan-In-Faktoren lässt sich Polymorphic Use errechnen, indem die Summe der Fan-Ins der Interfaces durch die Summe der Fan-Ins der Klasse und aller Interfaces geteilt wird.

Die Metrik *Dependency Inversion Principle* (DIP) aus Martin 2002 ähnelt der Metrik Polymorphic Use, der Fokus liegt jedoch auf einem Package und mehreren Klassen:

> *The DIP metric is defined as the percentage of dependencies in a package or class that has an interface or an abstract class as a target. A DIP of 100% indicates that all*

> *dependencies in a package diagram are based on interfaces or abstract classes* (Knoernschild 2001).

Die hier betrachtete Metrik PU untersucht jedoch die mit einer *einzelnen* Klasse und deren Interfaces im *gesamten* System typisierte Variablen, während die Metrik DIP Variablen *beliebiger* Klassen und Interfaces in *einem* Package bzw. den Interna *einer* Klasse berechnet.

4.5.4 Number of Different Access Contexts

Die in dieser Arbeit definierte NODAS-Metrik ist eine von drei Metriken, welche speziell die von Variablen einer Klasse bzw. eines Interfaces aufgespannten Kontexte betrachten – ein Konzept, welches bisher in der Metrik-Literatur so noch nicht vorhanden ist. NODAS berechnet die Anzahl paarweise verschiedener Methodenteilmengen, d.h. Kontexte, in welchen jeweils unterschiedliche Mengen von Methoden enthalten sind.

In Kapitel 2 wurde bereits auf die Problematik der Messung dieser Kontexte hingewiesen, was den rechnerischen Zeitaufwand und die Komplexität angeht. Tatsächlich lässt sich auch in der betrachteten Literatur keine Metrik finden, die eine vergleichbare Messung vornimmt.

Der Kopplungs-Fan-In-Faktor einer Klasse würde einer Metrik des Namens *Number of Variables* entsprechen. Der Schritt zu *Number of Different Access Contexts* erfordert jedoch die Analyse der auf jeder einzelnen Variablen aufgerufenen Methoden. Dies wird erst durch die NODAS-Metrik erreicht.

4.5.5 Actual Context Distance / Best Context Distance

Die in dieser Arbeit definierten Metriken ACD und BCD beziehen sich rein auf die Analyse von Kontexten; für sie gelten daher die bereits bei NODAS gezogenen Schlussfolgerungen.

4.5.6 Interface Minimization Indicator

Auch die Metrik IMI ist im Rahmen dieser Arbeit entstanden. Kurz gesagt berechnet sie den Prozentsatz überflüssiger Methoden in einem Interface.

Eine bereits recht ähnliche Messung ist die NOT-Metrik (Number of Tramps) aus Sharble/Cowen 1993. Sie misst die überflüssigen Parameter einer Methode, jedoch bezogen auf die Verwendung des Parameters innerhalb der Methode.

IMI kann auch durch die Berechnung von Fan-In-Faktoren simuliert werden; in diesem Fall durch Kopplungs-Fan-Ins der Methoden des Interfaces. Ist der Fan-In-Faktor einer Methode – hierbei müssen allerdings alle relevanten Projekte berücksichtigt werden – gleich 0, so wird die Methode nicht benötigt. Aus diesen Informationen sowie der Number of Methods, welche wie in Kapitel 2 beschrieben auch auf Interfaces angewendet werden kann, lässt sich IMI errechnen.

5 Implementation der Werkzeuge

In Kapitel 4 ist der Haupteinsatzzweck der in dieser Arbeit definierten Metriken bereits genannt worden. Es handelt sich um Quelltext-Metriken für die Implementierungsphase, die dem Entwickler direktes Feedback bezüglich der Erreichung der in Kapitel 2 definierten Ziele der interfacebasierten Programmierung geben sollen.

Derartige Informationen werden vom Entwickler dort benötigt, wo er seine Arbeit verrichtet: In einer integrierten Entwicklungsumgebung (IDE). Viele Programme zur Berechnung von Metriken sind jedoch als externe Tools realisiert; sie erfordern somit einen separaten Programmstart und machen den Wechsel zwischen Metrikergebnissen und Programmcode zu einem umständlichen Unterfangen. Im Gegensatz dazu sollen die hier definierten Metriken direkt aus einer IDE aufrufbar sein und ihre Ergebnisse auch innerhalb der IDE verfügbar machen – obwohl natürlich eine Export-Möglichkeit bestehen soll.

Darüber hinaus sind die hier definierten Metriken z.T. recht komplex zu berechnen; sie stellen hohe Ansprüche an verfügbaren Informationen. Obwohl einige lediglich einen schlichten Baumdurchlauf erfordern (z.B. Polymorphic Grade) analysiert die Mehrzahl die Nutzung einer Klasse und ihrer Interfaces in allen Kontexten des gesamten Projekts (z.B. Actual Context Distance). Tabelle 5-1 gibt einen Überblick über die Komplexität der Berechnung der in Kapitel 4 definierten Metriken:

Tabelle 5-1: Berechnung der Metriken

Metrik	Berechnung
PG	Durchlauf des Implementations- und Interface-Vererbungsbaums in Richtung Wurzel.
DA	Analyse der Methodendeklarationen der Klasse und der aller Interfaces der Klasse (Vergleich des impliziten Interfaces mit allen expliziten).
PU	Analyse der Verwendung einer Klasse und aller ihrer Interfaces in Variablendeklarationen im gesamten Projekt.
NODAS	Analyse der mit der Klasse oder einem Interface der Klasse deklarierten Variablen und allen Methoden, die im Gültigkeitsbereich der Variablen aufgerufen werden – im gesamten Projekt.
ACD	Analyse der in den Kontexten aufgerufenen Methoden einer Klasse im Vergleich zu den Methoden der jeweils definierten Typen – im gesamten Projekt.
BCD	Wie ACD, jedoch zusätzlich Analyse des besten Typs.
IMI	Nutzungsanalyse jeder Methode eines Interfaces im gesamten Projekt.

Zur Berechnung der letzteren Metriken ist ein einfacher Quelltext-Parser nicht ausreichend; benötigt wird ein Abstract Syntax Tree (AST), welcher die Referenzen zwischen den Elementen auflösen kann. Wird z.B. eine Variable mit einer gewissen Klasse als Typ deklariert, so ist es erforderlich, von dieser Referenz auf die Klasse zu derjenigen Stelle im gesamten Baum zu springen, an welcher die Klasse definiert ist, d.h. zur Klassendefinition, die sich in einer ganz anderen Datei befinden kann.

Auch aus diesem Grund wurde eine moderne IDE als Implementationsumgebung gewählt; die benötigten *resolving parse trees* sind dort bereits vorzufinden.

Im Rahmen dieser Arbeit sind zwei Plug-Ins für die Entwicklungsumgebung JetBrains IntelliJ IDEA (IntelliJIDEA) entstanden. Sie enthalten die folgende Funktionalität:

- *MetricPlugin*. Dieses Plug-In erlaubt die Berechnung der in Kapitel 4 definierten Metriken sowie weiterer, im Abschnitt 5.4 gelisteten Metriken. Die Ergebnisse werden in-

nerhalb der IDE präsentiert; eine Navigation vom Ergebnis zu den dazugehörigen Typen ist möglich; ebenso eine graphische Darstellung der Ergebnisse als Histogramm. Das MetricPlugin ist erweiterbar; neue Metriken können über eine interne Plug-In-Architektur eingebunden werden.
- *ContextAnalyzerPlugin*. Dieses, in Verbindung mit dem Autor eines parallel laufenden Projekts entwickelte Plug-In (vgl. Meißner 2003) erlaubt die Visualisierung der Ergebnisse der kontextspezifischen Metriken NODAS, ACD und BCD des zweiten Ziels für ausgewählte Klassen. Die so verfügbare Visualisierung stellt das Bindeglied zwischen den Ergebnissen der Metriken einerseits und der konkreten Verbesserung durch Ausführung von Refactorings andererseits dar.

Die Integration der Plug-Ins in IDEA machen diese IDE zu einer sehr komfortablen Umgebung für die Analyse, Einführung und Wartung von Interfaces.

Dieses Kapitel ist wie folgt aufgebaut: Zunächst erfolgt eine Einführung in die Plug-In-Architektur von IDEA sowie den Aufbau des in IDEA verfügbaren Parse-Trees (Abschnitt 5.1) – dies dient zur Orientierung und schafft den Rahmen für die späteren Abschnitte, die sich mit der Implementierung der konkreten Plug-Ins beschäftigen. Die Erläuterung der Implementierung erfolgt in den Abschnitten 5.2 und 5.3. Hier findet sich eine Beschreibung der Architektur des MetricPlugins sowie des ContextAnalyzerPlugins. Abschließend listet Abschnitt 5.4 alle implementierten Metriken des MetricPlugins auf.

5.1 Die Plug-In-Architektur der Entwicklungsumgebung IDEA

Für die Implementation der Plug-Ins wurde die IDE IntelliJ IDEA der Firma JetBrains (IntelliJIDEA) verwendet. Diese reine Java-IDE wurde aus mehreren Gründen ausgewählt:

- IDEA verfügt über einen weit fortgeschrittenen Abstract Syntax Tree, in welchem die benötigten Querverweise bereits gezogen sind und welcher während der Programmierung stets aktuell bleibt. So ist es jederzeit möglich, nach Änderungen im Code z.B. erneut die Analyse einer Klasse mit dem Context Analyzer durchzuführen.
- IDEA verfügt über viele Refactorings, welche sich auch für die Arbeit mit Interfaces einsetzen lassen und bietet daher ein ideales Umfeld für die Arbeit nach dem Interfaceparadigma. Weitere Refactorings werden in Meißner 2003 vorgestellt.
- IDEA ist leicht über eine Open API um Plug-Ins erweiterbar.
- IDEA ist eine bekannte und beliebte IDE.

5.1.1 Die IDEA Open API

In IDEA besteht die Möglichkeit der Erweiterung durch Plug-Ins, genauer gesagt Java-Code, welcher speziellen Richtlinien genügt und innerhalb von IDEA ausgeführt wird. Die Nahtstelle zwischen IDEA und dem jeweiligen Plug-In stellt die Plug-In API, oder Open API, dar.

Die IDEA Plug-In-Dokumentation (JetBrains 2003) beschreibt die verschiedenen Formen möglicher Plug-Ins. IDEA-Plug-Ins fallen in eine der folgenden drei Kategorien: IDE Plug-In, Web-Server Plug-In, und Version Control Plug-In.

- *IDE Plug-Ins* erlauben die Erweiterung der IDE um eine neue Funktion.
    - Es besteht die Möglichkeit der direkten Integration in die IDEA-GUI, Eingliederung eigener Menüpunkte in Haupt- und Kontextmenüs sowie Zugriff auf und Manipulation von Dateien und Quelltext.
    - Möglich ist auch der Empfang von Events wie Öffnen und Schließen eines Projekts bis hin zu Cursorbewegungen.

- *Web-Server Plug-Ins* erlauben die Verwendung von eigenen Servlet-Containern.
- *Version Control Plug-Ins* erlauben die Verwendung eigener Version-Control-Systeme.

Ebenso werden die Pflichten der Plug-In-Implementation beschrieben. Jedes Plug-In muss, neben beliebigen weiteren Java-Klassen, über einige Komponenten verfügen, welche die Integration in das IDEA-Framework erlauben. Zum einen muss das Plug-In dem System bekannt gemacht werden – eine ein bereitgestelltes Interface implementierende Klasse dient dabei als Ansprechpartner – zum anderen muss ein Einsprungspunkt für die Ausführung von Code als Reaktion auf Benutzeraktionen vorhanden sein.

Es existieren zwei unterschiedliche Arten von grundlegenden Plug-In-Komponenten als Ansprechpartner innerhalb des Plug-Ins – eine Applikationskomponente und eine Projektkomponente. IDEA ist projektbasiert; d.h. für jedes eigenständige Softwareprodukt definiert der Entwickler ein eigenes Projekt; mehrere Projekte können dabei gleichzeitig geöffnet sein. Ein Plug-In kann wahlweise eine der beiden Komponenten oder auch beide implementieren:

- Für jedes in IDEA geöffnete Projekt wird eine neue *Projektkomponente* jedes Plug-Ins erzeugt. Diese Komponenten müssen das Interface `ProjectComponent` implementieren, welches u.A. die folgenden Methoden enthält:

  ```
  String getComponentName();
  void projectOpened();
  void projectClosed();
  ```

  Einem speziellen Konstruktor wird dabei eine Referenz auf das aktuelle Projekt übergeben.

- Eine *Applikationskomponente* wird nur ein einziges Mal im Lebenszyklus einer IDEA-Instanz erzeugt und ist für alle geöffneten Projekte gültig. Eine solche Komponente muss das Interface `ApplicationComponent` implementieren, welches u.A. folgende Methoden enthält:

  ```
  String getComponentName();
  void initComponent();
  void disposeComponent();
  ```

Um das Plug-In in der GUI und damit für den Benutzer verfügbar zu machen, müssen Einsprungspunkte für das Plug-In definiert werden. In einer speziellen XML-Datei, der `plugin.xml`, werden für diesen Zweck so genannte *Actions* definiert. Es kann z.B. ein zusätzlicher Menüpunkt in das Kontextmenü eingefügt werden; wird dieser ausgewählt, so wird die in der Datei `plugin.xml` definierte Action aufgerufen. Diese Action muss im entsprechenden Plug-In implementiert sein; es handelt sich dabei um eine eigene, von der abstrakten Klasse `AnAction` abgeleitete Klasse.

Subklassen von `AnAction` müssen insbesondere folgende Methode implementieren:

```
actionPerformed(AnActionEvent e) { … }
```

Mit der Ausführung dieser Methode übergibt IDEA die Kontrolle an das Plug-In. Alles weitere ist der Action-Klasse überlassen.

Neben vielen Möglichkeiten des Zugriffs auf bereits existierende Funktionen von IDEA und der Darstellung von Informationen über die GUI ist für die hier geplanten Zwecke die Verwendung des internen Abstract Syntax Tree von größter Bedeutung. Dieser ist Thema des folgenden Abschnitts.

5.1.2 Das Program Structure Interface (PSI)

Das Herzstück des IDEA-Editors stellt die von JetBrains „*Program Structure Interface (PSI)*“ genannte API zum Zugriff auf den internen Abstract Syntax Tree dar, der hier im folgenden „PSI-Tree“ bzw. „PSI-Baum“ genannt werden soll. Über diesen Baum sind alle Informationen über den Quelltext abrufbar; ebenso sind Änderungen am PSI-Baum möglich, welche dann von IDEA in den Code integriert werden.

Der PSI-Baum ist im Prinzip eine Abbildung des Java-Quelltexts. Die Knoten des Baums stellen einzelne Elemente des Quelltexts dar, z.B. Klassen, Methoden oder auch geschweifte Klammern oder Kommata. Die Hierarchie des Baums korrespondiert zu der Schachtelung des jeweiligen Codeblocks, angefangen von ganzen Dateien bis hin zu Semikola.

Um den Baum nützlich zu gestalten sind Referenzen wie z.B. Typreferenzen oder Methodenaufrufe bereits aufgelöst, d.h. mit den entsprechenden Ursprungselementen verbunden worden. Damit ist z.B. die Navigation von einer Variablendeklaration zum deklarierten Typ der Variable, also einer Klasse oder einem Interface, möglich. Hierdurch entsteht ein Referenznetzwerk zwischen den einzelnen Knoten.

Abbildung 5-1 zeigt einige im PSI-Baum verwendeten Elemente, die hier besonders relevant sind; die Abbildung ist nicht annähernd vollständig.

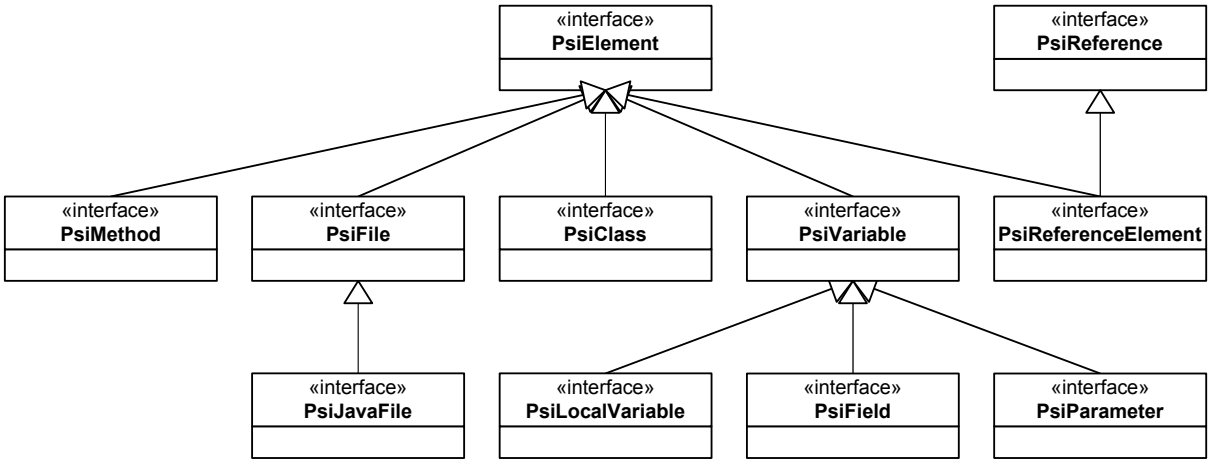


Abbildung 5-1: PSI-Interfaces

Neben der reinen Repräsentation des Source-Codes sind in diesen Elementen auch Methoden enthalten, um den Baum zu verändern. Alle Knoten im Baum implementieren ein Subinterface von `PsiElement`, welches die grundsätzlichen Attribute und Methoden bereitstellt, um im Baum zu navigieren (siehe Abbildung 5-2). Methoden implementieren `PsiMethod`, Klassen `PsiClass`, etc. Die Wurzel des Baums jeder Klasse ist ein `PsiJavaFile`-Element; die Wurzel der gesamten Package-Hierarchie das `PsiPackage`, welches das nicht benannte Root-Package repräsentiert.

```
public interface PsiElement extends UserDataHolder
{
      …
      public abstract PsiElement[] getChildren();
      public abstract PsiElement getParent();
      public abstract PsiElement getFirstChild();
      public abstract PsiElement getLastChild();
      public abstract PsiElement getNextSibling();
      public abstract PsiElement getPrevSibling();
      …
      public abstract PsiFile getContainingFile();
      …
      public abstract void accept(PsiElementVisitor psielementvisitor);
      …
      public abstract PsiElement add(PsiElement psielement)
        throws IncorrectOperationException;
      public abstract void delete()
        throws IncorrectOperationException;
      …
      public abstract boolean isWritable();
      …
      public abstract PsiReference getReference();
}
```

Abbildung 5-2: PsiElement

Querreferenzen wie z.B. ein Verweis auf den Typ einer Variablen sind im Baum als `PsiReferenceElements` enthalten, welche sowohl von `PsiElement` als auch von `PsiReference` ableiten. Ein Feld sieht im PSI-Baum wie folgt aus (Abbildung 5-3).

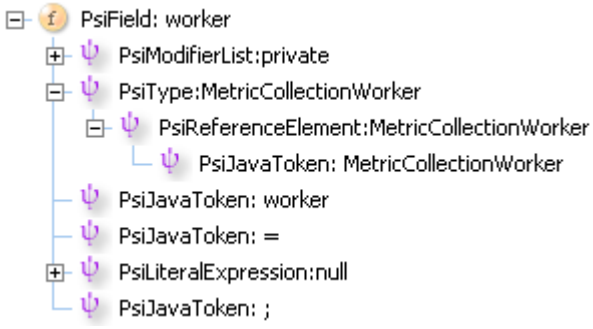


Abbildung 5-3: PsiField: „private MetricCollectionWorker worker = null;"

Das Interface `PsiReference` enthält neben anderen Funktionen die Methode

```
public abstract PsiElement resolve();
```

Diese Methode erlaubt die Auflösung des jeweiligen `PsiReferenceElements`, um das tatsächliche `PsiElement` zu erhalten.

Ein Beispiel eines kompletten PSI-Baums für eine Java-Klasse zeigt Abbildung 5-4.

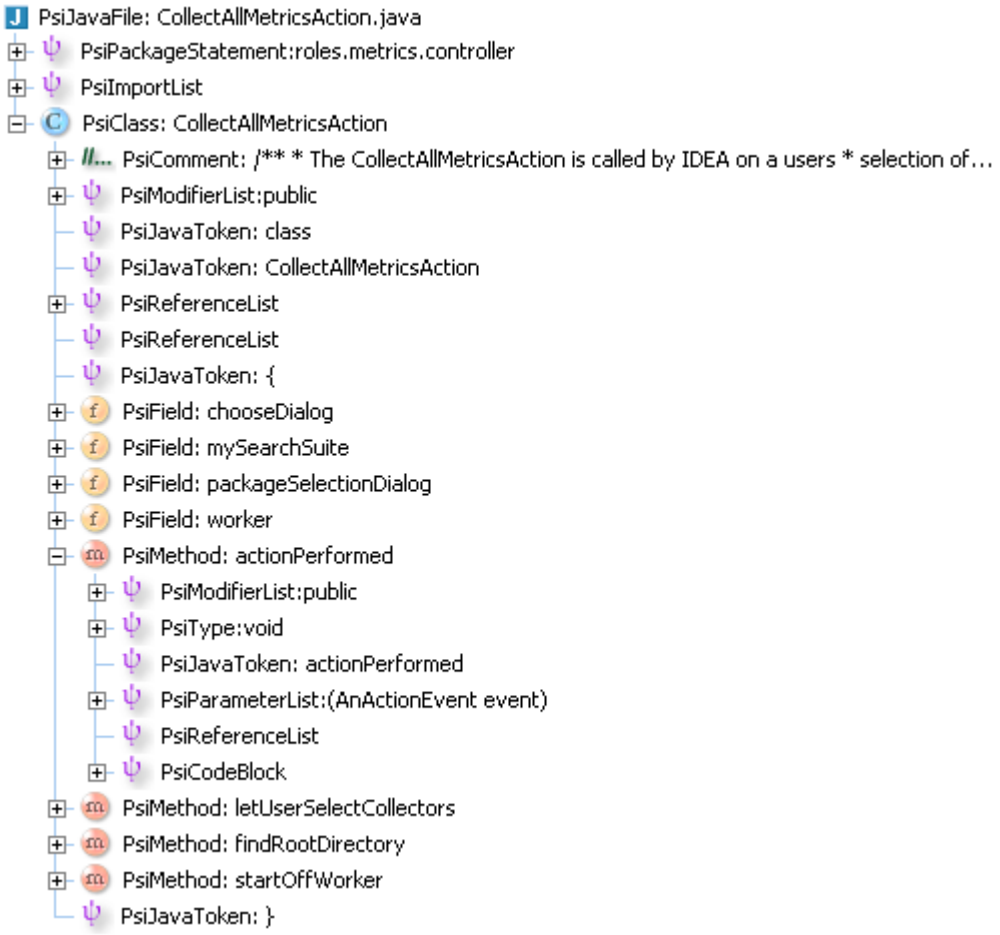


Abbildung 5-4: Beispiel eines PSI-Baums

In IDEA integrierte Plug-Ins haben vollständigen Zugriff auf den PSI-Baum für jede Datei des aktuellen Projekts. Dabei stehen Hilfsklassen zur Verfügung, welche die Navigation im Baum erleichtern. Zwei dieser Klassen sind hier besonders relevant: Über die Klassen `PsiElementVisitor` und `PsiRecursiveElementVisitor` ist es möglich, den Baum zu durchlaufen und bei gewissen Elementen Operationen auszuführen (siehe Abbildung 5-5). `PsiElementVisitor` und `PsiRecursiveElementVisitor` unterscheiden sich dadurch, dass in letzterem bereits die automatische Rekursion durch den Baum implementiert ist.

```
public abstract class PsiElementVisitor {
      …
      public void visitClass(PsiClass psiclass);
      public void visitPackage(PsiPackage psipackage);
      public void visitMethod(PsiMethod psimethod);
      …
      public void visitLocalVariable(PsiLocalVariable psilocalvariable);
      public void visitParameter(PsiParameter psiparameter);
      public void visitField(PsiField psifield);
      …
}
```

Abbildung 5-5: Die Klasse PsiElementVisitor

Visitoren (welche direkt zu dem in Gamma et al. 1995 genannten Visitor-Pattern korrespondieren) können z.B. verwendet werden, um alle Klassen des momentan in IDEA gewählten Projekts zu durchlaufen und für jede Klasse eine Metrik zu berechnen. Hierfür wird eine Sub-

klasse von `PsiRecursiveElementVisitor` gebildet, in welcher die entsprechenden Methoden – in diesem Fall mindestens `visitClass(…)` – überschrieben werden.

Der Visitor kann mittels der in allen `PsiElementen` vorhandenen Methode `accept()` von einem gewissen Element aus den Baum durchlaufen. Bei jedem gefundenen Element wird die korrespondierende Methode in der Visitor-Instanz aufgerufen.

Neben den genannten Visitoren existiert noch eine weitere Suchmöglichkeit im PSI-Baum, welche für die Metrikberechnung von größter Wichtigkeit ist. Über die Projektinstanz kann ein `PsiSearchHelper` angefordert werden: Dieser ist spezifisch für das geöffnete Projekt und greift auf den internen IDEA-Cache zu. Das Interface `PsiSearchHelper` bietet u.A. folgende Funktionen (Abbildung 5-6).

```
public interface PsiSearchHelper
{
      …
      public abstract PsiReference[] findReferences(PsiElement
psielement, PsiSearchScope psisearchscope, boolean flag);

      public abstract PsiClass[] findInheritors(PsiClass psiclass,
PsiSearchScope psisearchscope, boolean flag);
      …
}
```

Abbildung 5-6: Das Interface PsiSearchHelper

Die Funktionen `findReferences()` und `findInheritors()` haben eine sehr hohe Ausführungsgeschwindigkeit. Erstere eignet sich zur Analyse der Nutzung von Elementen – z.B., um Variablendeklarationen mit gewissen Klassen aufzuspüren – letztere liefert alle Erben einer gewissen `PsiClass` zurück.

Die folgenden Abschnitte zeigen die konkrete Verwendung der oben besprochenen Techniken für die Implementation des MetricPlugins bzw. des ContextAnalyzerPlugins.

5.2 Implementation des MetricPlugins

Wie bereits in der Einleitung dieses Kapitels beschrieben verstehen sich die in dieser Arbeit vorgestellten Metriken als direkte Hilfe für den Entwickler und sollten sich daher so gut wie möglich in seine gewohnte Arbeitsumgebung – in diesem Fall IDEA – eingliedern. Dies schließt ein:

- Aufruf der Metrikberechnung über schnell erreichbare Menüs bzw. Tastenkürzel
- Berechnung für vom Benutzer festzulegende Source-Pfade im Projekt
- Anzeige der Ergebnisse im System integriert (in Form eines sog. „ToolWindow“).

In diesem Abschnitt werden die grundlegenden Elemente der Implementation des MetricPlugins erläutert; eine komplette Durchsicht des Quelltextes würde den Rahmen sprengen. Für weitere Informationen kann die bis zum `private`-Level vollständige API-Dokumentation des MetricPlugins herangezogen werden.

Das MetricPlugin ist als `ProjectComponent` realisiert: Die von IDEA während der Instantiierung an den `ProjectComponent` übergebene Instanz des geöffneten IDEA-Projekts ist u.A. für die Erzeugung einer eigenen GUI für die Metrik-Anzeige innerhalb des Systems – über ein so genanntes ToolWindow (siehe unten) – nötig.

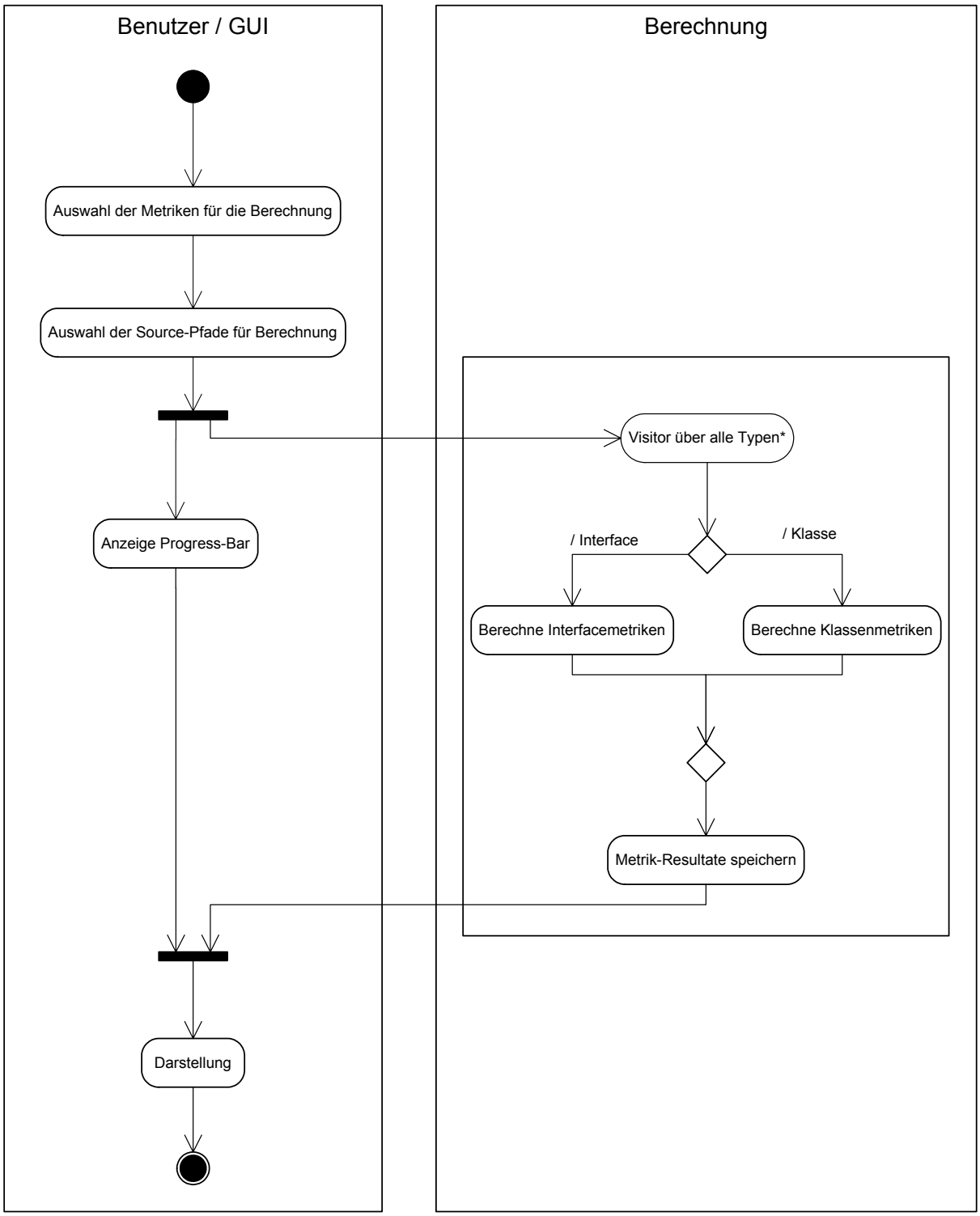


Abbildung 5-7: Programmablauf MetricPlugin

Das MetricPlugin, welches grundlegend nach dem MVC-Ansatz (Model-View-Controller) aufgebaut ist, besteht aus mehreren logischen Einheiten, die im Folgenden einzeln besprochen werden sollen:

- Eine *Plug-In-Architektur* für die Sammlung der Metriken über einen generischen Visitor im gesamten Projekt. Nach Auswahl der gewünschten Metriken durch den Benutzer werden diese über eine Factory instantiiert und für die Sammlung registriert.

- Den *einzelnen Metriken* selbst, welche als Plug-Ins in die oben genannte Architektur realisiert sind und somit beliebig erweitert werden können. Die einzelnen Metriken berechnen ihre Werte direkt aus dem von IDEA bereitgestellten PSI-Baum.
- Einer *Benutzeroberfläche* zur Anzeige der berechneten Metriken in textueller und graphischer Form.

Der Benutzer hat mit allen drei Komponenten zu tun, wenn er eine Metrikberechnung startet. Dabei durchläuft er – bzw. im späteren Verlauf das Programm – die in Abbildung 5-7 dargestellte Sequenz. Der Einstiegspunkt für die Nutzung des Plug-Ins wird durch die Auswahl eines bestimmten Menüpunktes (siehe unten) erreicht. Die Aktivierung des Menüpunkts führt zur Instantiierung der beigeordneten Action-Klasse durch IDEA; diese Verknüpfung wird in der `plugin.xml` gezogen (Abbildung 5-8).

```
<action
  id="metrics.MetricsPlugin.collectAllMetrics"
  class="roles.metrics.controller.CollectAllMetricsAction"
  text="_Metrics Collection..."
  description="Collects metrics in this project" />

<group id="MetricsPluginCAM" text="_Metrics">
  <reference id="metrics.MetricsPlugin.collectAllMetrics" />
  <add-to-group group-id="ToolsMenu" anchor="after" relative-to-
      action="ViewOfflineInspection" />
</group>
```

Abbildung 5-8: Definition der Action in der plugin.xml

5.2.1 Plug-In-Architektur

Obwohl die in dieser Arbeit präsentierte Metrik-Suite ein in sich abgeschlossenes Konzept darstellt, soll das MetricPlugin nicht an diese Suite gebunden sein. In der Tat sind bereits weitere Metriken implementiert, die nicht nur mit der interfacebasierten Programmierung zu tun haben. Ermöglicht wird dies auf einfache Art und Weise durch die offene Architektur des Systems. Neue Metriken können durch simple Implementation eines Interfaces definiert werden; die Registrierung bei einer Metriken-Fabrik macht sie zum Teil des Systems.

Abbildung 5-9 zeigt den konkreten technischen Zusammenhang zwischen den fünf relevanten Klassen/Interfaces der Architektur (`Metric`, `MetricFactory`, `ConcreteMetricRunSuite`, `GlobalMetricCollector` und `ResultSet`).

Die grundlegenden Klassen/Interfaces für die Implementation der genannten Plug-In-Architektur stellen dabei folgende Funktionen bereit:

- `Metric`. Dieses Interface muss von jeder konkreten Metrik implementiert werden. Instanzen der implementierenden Klassen werden nur von der MetricFactory erzeugt; sie sind im Singleton-Pattern realisiert (Gamma et al. 1995, S. 157).
- `MetricFactory`. Die nach dem Factory-Pattern (Gamma et al. 1995, S. 107) realisierte Objektfabrik erzeugt Objekte des Typs `Metric` und stellt eine Liste der verfügbaren Metriken bereit.
- `ConcreteMetricsRunSuite`. Pro Durchlauf der Metriken über das Projekt existiert eine derartige Suite, in welcher die Metriken für den konkreten Lauf registriert und die ResultSets mit den Ergebnissen pro Typ eingetragen werden.

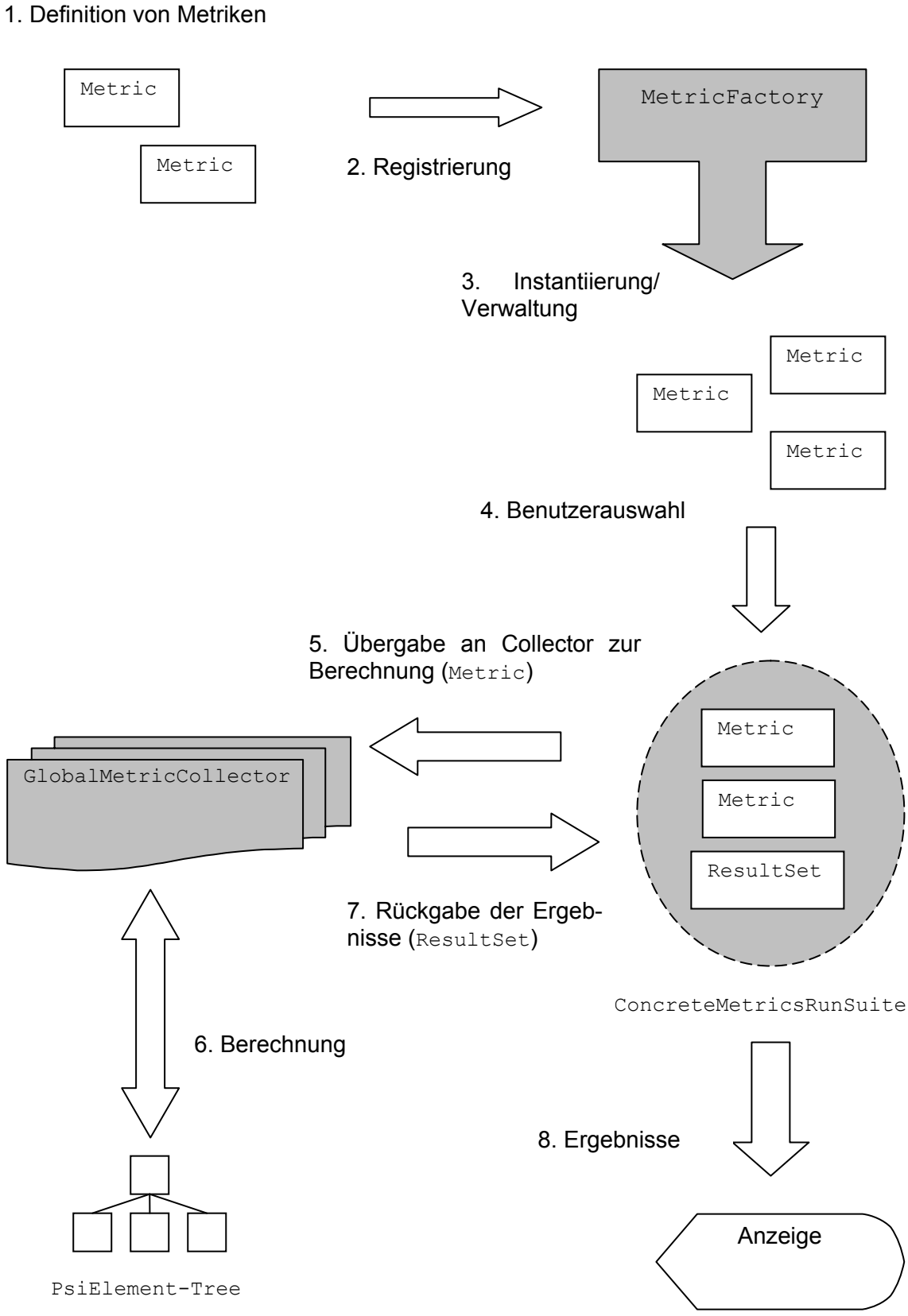


Abbildung 5-9: Funktionsweise des MetricPlugins

- `GlobalMetricsCollector`. Hierbei handelt es sich um einen von `PsiRecursiveElementVisitor` abgeleiteten Visitor nach dem Visitor-Pattern (Gamma et al. 1995, S. 301). Für einen konkreten Metrikdurchlauf wird dem Visitor eine `ConcreteMetricsRunSuite` übergeben; mit den dort registrierten Metriken wird dann das gesamte Projekt durchlaufen.
- `ResultSet`. Ein `ResultSet` stellt den Datencontainer für alle Metrikergebnisse eines Typs dar.

Ausgangspunkt im Programmablauf des MetricPlugins ist die Initialisierung der zuvor implementierten Metriken (1) in der `MetricFactory` (2). Letztere hat Kenntnis von allen im Programm zur Verfügung stehenden Implementierungen von Metriken – welche alle das Interface `Metric` implementieren – und bietet der Benutzeroberfläche eine Liste an, welche dem Benutzer in einem Auswahldialog präsentiert wird (Dialog siehe unten, Abbildung 5-16) (3).

Die in diesem Dialog angezeigten Informationen stammen zum größten Teil aus den implementierenden Klassen von `Metric` selbst. `Metric` enthält folgende Methoden (Abbildung 5-10).

```
public interface Metric {

    public String getName();
    public String getAbbreviation();
    public String getDescription();

    public int getTypeOfMetric();

    public void perform(ResultSet newResultSet, PsiClass
      theType, List interfaces, ConcreteMetricsRunSuite suite);

    public Metric[] getDepends();
}
```

Abbildung 5-10: Das Interface Metric

Die Methoden `getName()`, `getDescription()` und `getAbbreviation()` werden in erwähntem Dialog eingesetzt, um Informationen über die Metriken anzuzeigen. `getTypeOfMetric()` zeigt an, für welche Typen (Interfaces, Klassen) `perform()` aufgerufen werden muss. Da Metriken von den Werten anderer Metriken abhängen können, müssen diese ggf. zuerst berechnet werden; um dies anzuzeigen wird die Methode `getDepends()` zur Verfügung gestellt. Die Methode `perform()` wiederum wird zur eigentlichen Berechnung eingesetzt (siehe unten).

Nach Auswahl der Metriken durch den Benutzer werden diese in einer Instanz der Klasse `ConcreteMetricsRunSuite` registriert (4). Diese Klasse wird neben den so registrierten, die Metrikergebnisse berechnenden Klassen später auch die Ergebnisse in Form von `ResultSets` aufnehmen.

Die Klasse besitzt Methoden, um Metriken zu registrieren und zu deregistrieren sowie `ResultSets` aufzunehmen. Die Registrierung von Metriken erfolgt nach folgendem Schema (Abbildung 5-11).

```
…
Iterator it = theUsersSelectedCollectors.iterator();
while (it.hasNext()) {
      Metric current = (Metric) it.next();
      …
      // Resolve dependencies _before_ metric is registered
      searchSuite.registerMetric(current);
}
…
return searchSuite;
```

Abbildung 5-11: Registrierung der Metriken

Die so initialisierte `ConcreteMetricsRunSuite` wird nun an eine Instanz des `GlobalMetricsCollector` übergeben (5). Wie oben bereits angesprochen handelt es sich bei diesem um einen Visitor, der von `PsiRecursiveElementVisitor` ableitet – einer von IDEA bereitgestellten Klasse.

Ein derartiger Visitor arbeitet sich rekursiv von einem definierten Anfangspunkt durch den PSI-Baum und besitzt die Fähigkeit, an bestimmten Elementen des Baums Operationen durchzuführen. Der definierte Ausgangspunkt ist hier die Wurzel der Hierarchie – also das unbenannte Root-Package des Projekts. Mittels folgendem Aufruf wird die Berechnung angestoßen:

```
rootDirectory.accept(theCollector)
```

Der Aufruf – und damit die gesamte Berechung – erfolgt dabei innerhalb eines eigenen Threads, um die Anzeige eines Fortschrittbalkens zu ermöglichen.

Die wichtigste Methode bei der Berechnung im Visitor ist die Methode `visitClass(PsiClass theType)`. Da die Suite alle registrierten Metriken kennt, wird diese mit der Berechnung der Metriken beauftragt; sie delegiert die Berechnung an die konkreten implementierenden Klassen des Interfaces `Metric` weiter (6). Diese werden im folgenden Abschnitt besprochen; hier werden sie als Black Box betrachtet. Aus der Eingabe der Klasse/des Interfaces und eventuelle Superinterfaces werden konkrete Metrikergebnisse berechnet; diese werden in Form von `ResultSets` in der Suite abgelegt (7).

Ein `ResultSet` enthält alle berechneten Metrikwerte für einen gefundenen Typ, d.h. eine Klasse oder ein Interface (Abbildung 5-12). Die Werte werden dabei über das Metrikkürzel, also einen String identifiziert.

```
public class ResultSet implements Comparable {
    …
    public void addResult(String abbreviation, Object value) {
    public Object getResult(String abbreviation) {
    …
}
```

Abbildung 5-12: Die Klasse ResultSet

Nach Abschluss der Berechnungen werden die in der `ConcreteMetricsRunSuite` enthaltenen ResultSets ausgewertet. Ein Objekt der Klasse `MetricTableModel` nimmt die Werte für die Darstellung in der Resultattabelle auf; ein Objekt der Klasse `Histogram` die Werte jeder Einzelmetrik für die Darstellung als Histogramm (siehe unten). Die Objekte werden an die GUI zur Darstellung übergeben (8).

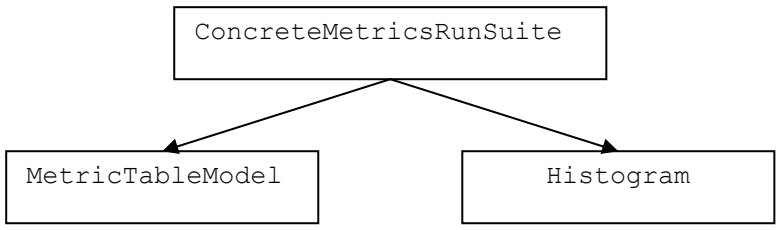


Abbildung 5-13: Ableitung der Resultatwerte

Der folgende Abschnitt beschreibt die Implementation einer Metrik genauer.

5.2.2 Implementation der Metriken

Das Interface `Metric` ist oben bereits kurz vorgestellt worden. Jede Metrik im System implementiert dieses Interface; so existieren z.B. eigene Klassen für die Berechnung von Polymorphic Grade, Decoupling Accessibility, Polymorphic Use etc.

Die Funktionen zur Berechnung der Metriken in den Klassen erhalten vollen Zugriff auf den PSI-Baum von IDEA, was alle Arten von Metrikimplementationen erlaubt.

Die relevanten Methoden für die Implementation sind in Abbildung 5-10 zu sehen. Sie sollen hier etwas ausführlicher erläutert werden.

```
public String getName();
```

Erwartet wird ein längerer, erläuternder Name, der im Dialog für die Metrikauswahl des Benutzers präsentiert wird.

```
public String getAbbreviation();
```

Dieses Kürzel dient zur späteren Identifikation der Metrik in der Resultattabelle sowie für die Speicherung von Metrikergebnissen in den ResultSets.

```
public String getDescription();
```

Liefert eine längere Beschreibung der Metrik für den Benutzerdialog zurück. HTML ist erlaubt.

```
public int getTypeOfMetric();
```

Zurückgegeben werden muss eine von drei in `Metric` definierten Konstanten:

- `METRIC_INFORMATIONAL`. Dieser Wert zeigt an, dass es sich bei dieser Metrik um eine reine Informationsmetrik handelt, welche einen textuellen String zurückgibt – z.B. das Package, zu welchem der Typ gehört.
- `METRIC_CLASS`. Dieser Wert zeigt an, dass die Metrik auf Klassen operiert.
- `METRIC_INTERFACE`. Dieser Wert zeigt an, dass die Metrik auf Interfaces operiert.

```
public void perform(ResultSet newResultSet, PsiClass
   theType, List interfaces, ConcreteMetricsRunSuite suite);
```

Die Methode `perform()` ist die bedeutsamste innerhalb des Interfaces `Metric`. Sie wird während des Durchlaufs aller im Projekt vorhandenen Typen mittels des `GlobalMetrics-Collector` durch die `ConcreteMetricsRunSuite` aufgerufen, um die Metrik für einen bestimmten Typ zu berechnen. Übergeben werden vier Parameter:

1. *ResultSet*. Hierbei handelt es sich um das für den aktuellen Typ unter Betrachtung vorgesehene `ResultSet`, dem eigene Werte hinzugefügt werden oder andere entnommen werden können. Es wird garantiert, dass alle Werte von Metriken, die im Depends-Block (siehe unten) angegeben sind, bereits enthalten sind.
2. *PsiClass*. Hierbei handelt es sich um den konkreten Typ, für welchen die Metrik berechnet werden soll. Instanzen von `PsiClass` können sowohl Klassen als auch Interfaces enthalten.
3. *List (of Interfaces).* Bei den hier vorgestellten Metriken handelt es sich um Berechnungen, die in fast jedem Fall mit den Interfaces einer Klasse zu tun haben, d.h. für jede Berechnung der Metriken ist eine Liste der implementierten Interfaces einer Klasse vonnöten. Die Suche nach diesen Interfaces wurde daher aus Effizienzgründen ausgelagert: Die Liste der implementierten Interfaces wird vor Berechnung der Metriken aufgestellt und im Anschluss an `perform()` übergeben.
4. *ConcreteMetricsRunSuite*. Um eine Rücknavigation zu ermöglichen, wird die Suite bei der Berechnung übergeben. So kann die Metrik auf bisherige Ergebnisse, die für andere Typen berechnet wurden, zurückgreifen.

Die Metrik hat in der Berechnung ihres Ergebnisses alle Freiheiten. Der berechnete Wert wird in das bereitgestellte `ResultSet` eingetragen. Ein Beispiel für eine sehr einfache Metrik – die weiter unten noch genauer vorgestellte, technische Metrik Generality, welche die Anzahl der implementierten Klassen eines Interfaces berechnet – ist in Abbildung 5-14 zu sehen.

```
public void perform(ResultSet newResultSet, PsiClass theType,
      List interfaces, ConcreteMetricsRunSuite suite) {

    final PsiSearchHelper searchHelper = theType.getManager().
            getSearchHelper();

    final PsiClass[] inh = searchHelper.findInheritors(theType,
            PsiSearchScope.PROJECT, true);

    int generality = 0;
    for (int i = 0; i < inh.length; i++)
        if (!inh[i].isInterface()) generality++;

    final String result = String.valueOf(generality);
    newResultSet.addResult(getAbbreviation(), result);

}
```

Abbildung 5-14: Berechnung der Metrik "Generality"

Die letzte hier zu besprechende Methode des Interfaces `Metric` ist `getDepends()`:

```
public Metric[] getDepends() { … }
```

Eine Metrik kann von den Werten anderer Metriken abhängen. Über die Methode `getDepends()` kann sie dies anzeigen; zurückgegeben wird ein Array des Typs `Metric`. Da die Metrik-Instanzen als Singletons realisiert sind, müssen sie innerhalb dieser Methode von der Factory angefordert werden.

Das Plug-In-Framework garantiert, dass vor dem Aufruf der `perform()`-Methode einer Metrik alle über `getDepends()` spezifizierten Metriken auf dem momentanen Typ aufgerufen worden sind.

5.2.3 Die Benutzeroberfläche

Das MetricPlugin erfordert an drei Stellen eine Interaktion des Benutzers. Zum einen vor der eigentlichen Berechnung der Metriken – hier muss der Benutzer eine Auswahl der gewünschten Metriken treffen sowie den Bereich des Codes festlegen, der untersucht werden soll – zum zweiten während der Berechnung in Form einer Fortschrittsanzeige mit der Möglichkeit des Abbruchs, und zum dritten im Anschluss an die Berechnung in Form einer geeigneten Präsentation der Ergebnisse.

Die Integration in das System ist ebenfalls Teil der Benutzerschnittstelle. Das Plug-In integriert sich in das Tools-Menü von IDEA. Diese Position wurde gewählt, da an dieser Stelle bereits weitere Tools zur Analyse des Codes (die IDEA-„Inspections“) bereitgestellt werden (Abbildung 5-15).

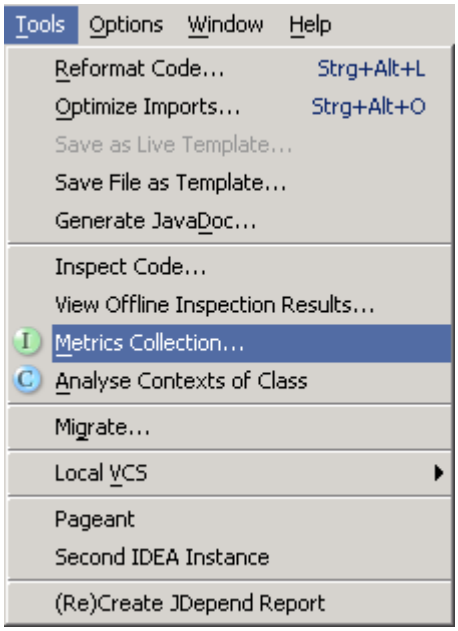


Abbildung 5-15: Einbindung in das Idea-Tools-Menü

Bei Aufruf des Menüpunkts wird dem Benutzer die Möglichkeit gegeben, die für ihn interessanten Metriken auszuwählen. Abbildung 5-16 zeigt den entsprechenden Dialog. Die im vorigen Abschnitt angesprochenen Abhängigkeiten zwischen den Metriken werden im Dialog reflektiert – wird eine Metrik ausgewählt, die Dependenzen besitzt, so werden diese ebenso ausgewählt.

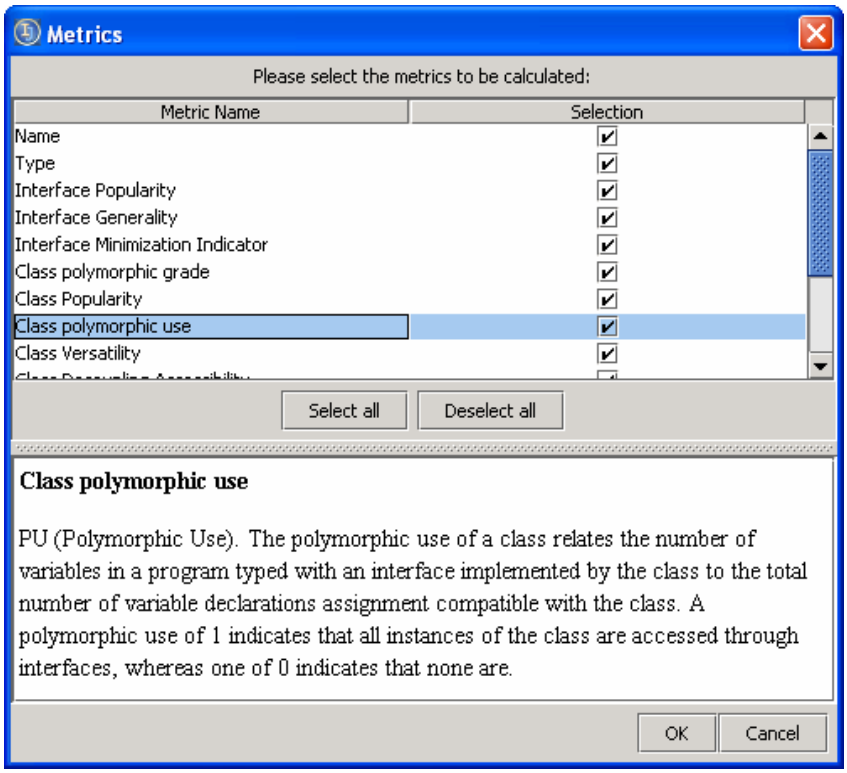


Abbildung 5-16: Metrikauswahldialog

Besitzt das ausgewählte Projekt mehrere Source-Roots, so kann der Benutzer im nächsten Schritt entscheiden, welche Verzeichnisse mit in die Berechnung einbezogen werden sollen. Dies ist über folgenden Dialog möglich (Abbildung 5-17).

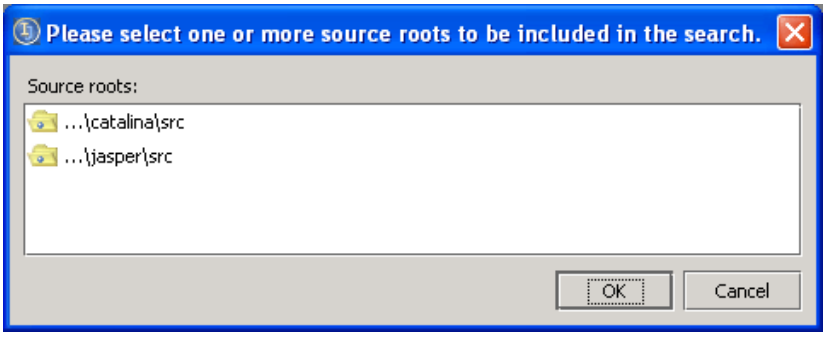


Abbildung 5-17: Auswahl der Source Roots

Während der Berechnungen weist ein Fortschrittsbalken den Benutzer auf den Stand der Berechnungen hin. Angegeben wird neben einem prozentualen Fortschritt (welcher aus den gesamt vorhandenen Typen zu den bereits besuchten Typen berechnet wird) auch die aktuell in der Berechnung befindliche Klasse (Abbildung 5-18).

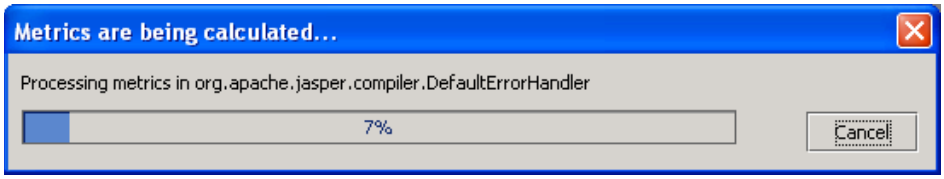


Abbildung 5-18: Fortschrittsanzeige bei Berechnung der Metriken

Ist die Berechnung abgeschlossen, werden die Ergebnisse präsentiert. IDEA stellt Ergebnisse eigener Untersuchungen, wie z.B. für die Suche oder für Refactorings, in so genannten ToolWindows dar (Abbildung 5-19).

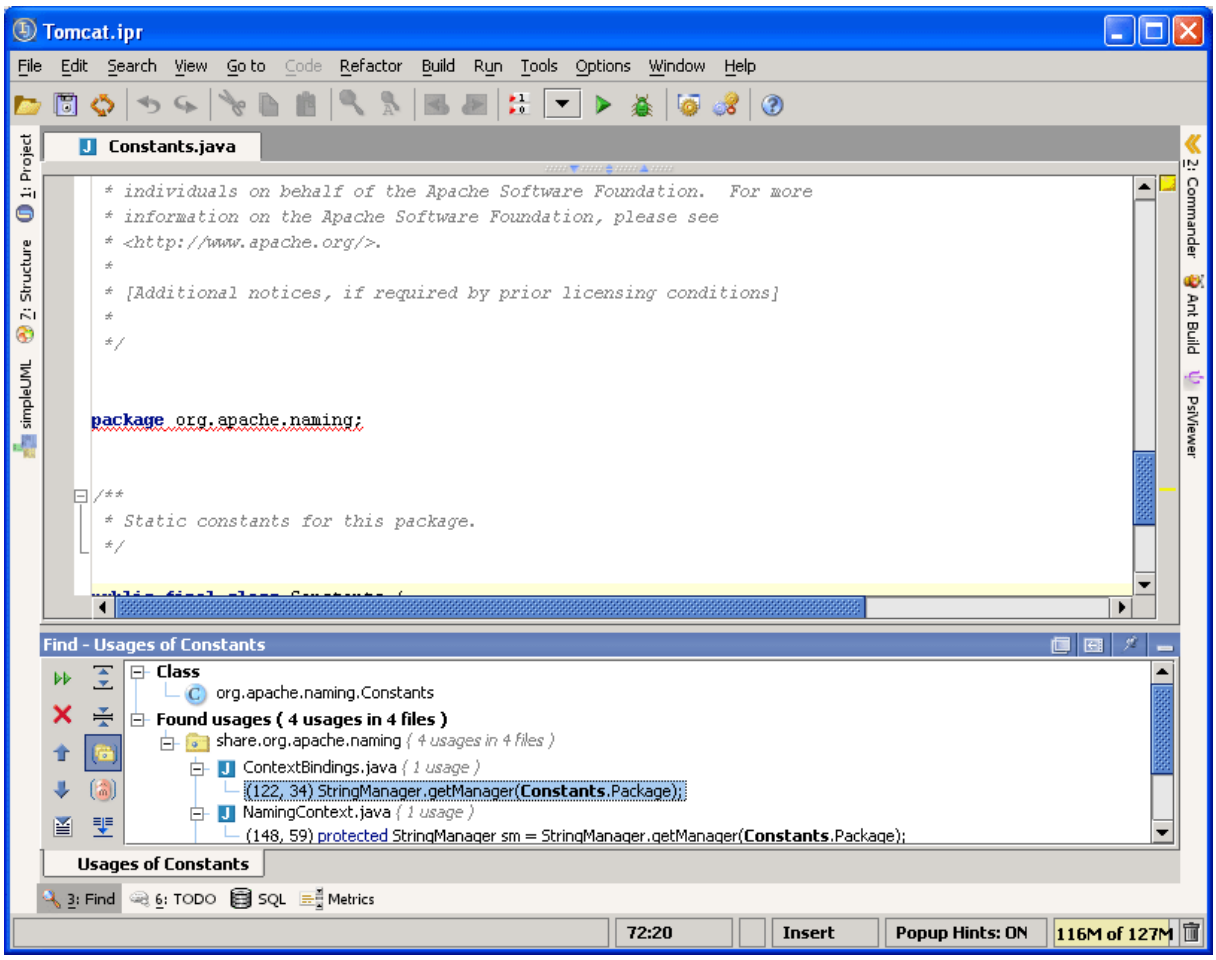
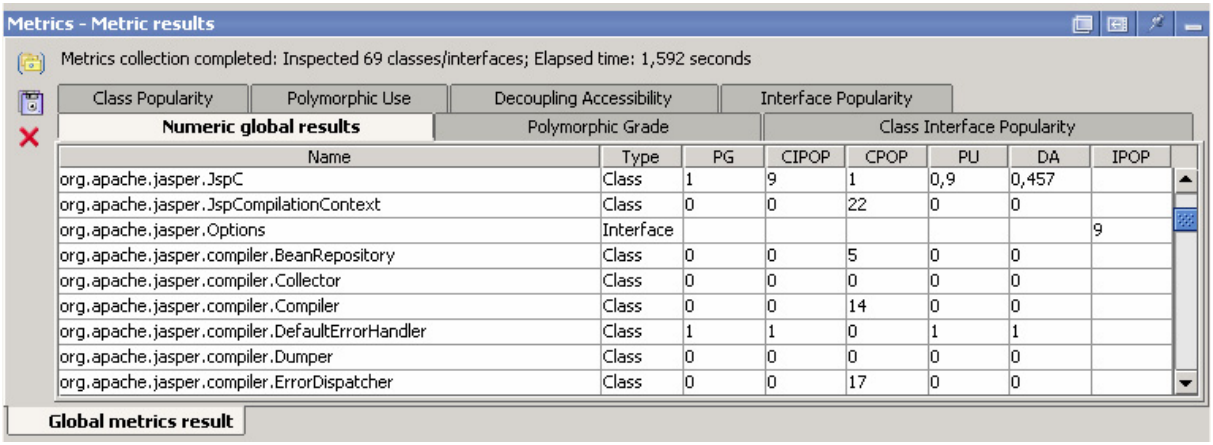


Abbildung 5-19: Das IDE-Fenster

Für die Metriken wird ein ähnliches ToolWindow verwendet. Es enthält die folgenden Informationen:

1. Eine textuelle Ausgabe der gewonnenen Ergebnisse des Metriklaufs in Form einer Tabelle (Abbildung 5-20).
2. Eine graphische Ausgabe in Histogrammform zu jeder Metrik (Abbildung 5-21).



Name	Type	PG	CIPOP	CPOP	PU	DA	IPOP
org.apache.jasper.JspC	Class	1	9	1	0,9	0,457	
org.apache.jasper.JspCompilationContext	Class	0	0	22	0	0	
org.apache.jasper.Options	Interface						9
org.apache.jasper.compiler.BeanRepository	Class	0	0	5	0	0	
org.apache.jasper.compiler.Collector	Class	0	0	0	0	0	
org.apache.jasper.compiler.Compiler	Class	0	0	14	0	0	
org.apache.jasper.compiler.DefaultErrorHandler	Class	1	1	0	1	1	
org.apache.jasper.compiler.Dumper	Class	0	0	0	0	0	
org.apache.jasper.compiler.ErrorDispatcher	Class	0	0	17	0	0	

Abbildung 5-20: Numerisches Resultat

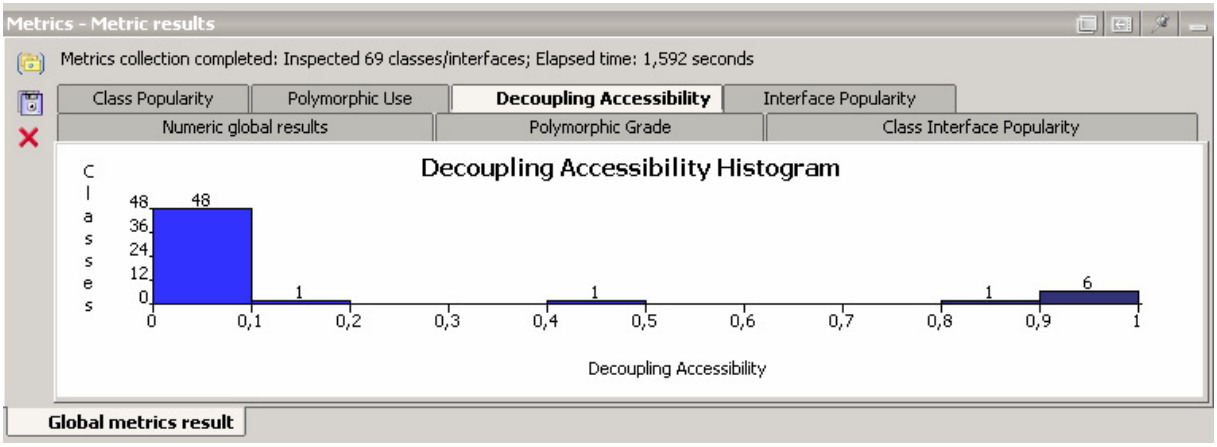


Abbildung 5-21: Graphisches Resultat

Der Export sowohl der textuellen Daten (im .csv-Format (Comma Separated Values)) als auch der graphischen Daten (im .png-Format (Portable Network Graphics)) ist möglich.

Das gesamte IDE-Fenster von IDEA, beispielhaft mit geöffnetem Find-ToolWindow, zeigt die folgende Abbildung:

5.3 Implementation des ContextAnalyzerPlugins

Die Aufgaben des ContextAnalyzers sind bereits in Kapitel 4 erläutert worden; hier soll es sich um die Implementation des Plug-Ins drehen. Im Vergleich zum MetricPlugin ist das ContextAnalyzerPlugin jedoch sehr einfach aufgebaut. Der prinzipielle Aufbau aller IDEA-Plug-Ins ist zudem identisch; daher wird hier nicht erneut darauf eingegangen.

Aufgabe des ContextAnalyzerPlugins ist es wie oben angegeben, die Kontexte einer Klasse in anschaulicher Weise zu visualisieren. Ausgangspunkt sind hier also, anders als im MetricPlugin, nicht die Source-Roots des Projekts, sondern einzelne Klassen. Es bietet sich hierbei an, die aktuelle im Editor oder in der Projektansicht gewählte Klasse zu verwenden. Aus diesem Grund ist der ContextAnalyzer nicht nur wie das MetricPlugin über das Tools-Menü aktivierbar (Abbildung 5-15), sondern auch über das Kontextmenü. Dies erlaubt es, den Analyzer auf beliebigen im Code ausgewählten Klassenreferenzen aufzurufen.

Das unten abgebildete Sequenzdiagramm zeigt den Ablauf. In Meißner 2003 findet sich eine erweiterte Variante, welche auch die möglichen, im Anschluss auszuführenden Refactorings berücksichtigt (Abbildung 5-22).

Der Visualisierung über einen eigenen Dialog – an dieser Stelle wurde aufgrund der Informationsmenge kein ToolWindow verwendet – geht die Berechnung der notwendigen Informationen voraus. Diese nutzt, wie auch die Metriken im MetricPlugin, den von IDEA bereitgestellten PSI-Baum direkt. Berechnet werden hierbei

- Die Kontexte von Variablen, welche mit der Klasse – oder einem Interface der Klasse – deklariert sind.
- Distanzen zwischen den Kontexten und den möglichen Typen und dadurch auch die im Sinne der BCD-Metrik bestmöglichen Typen.

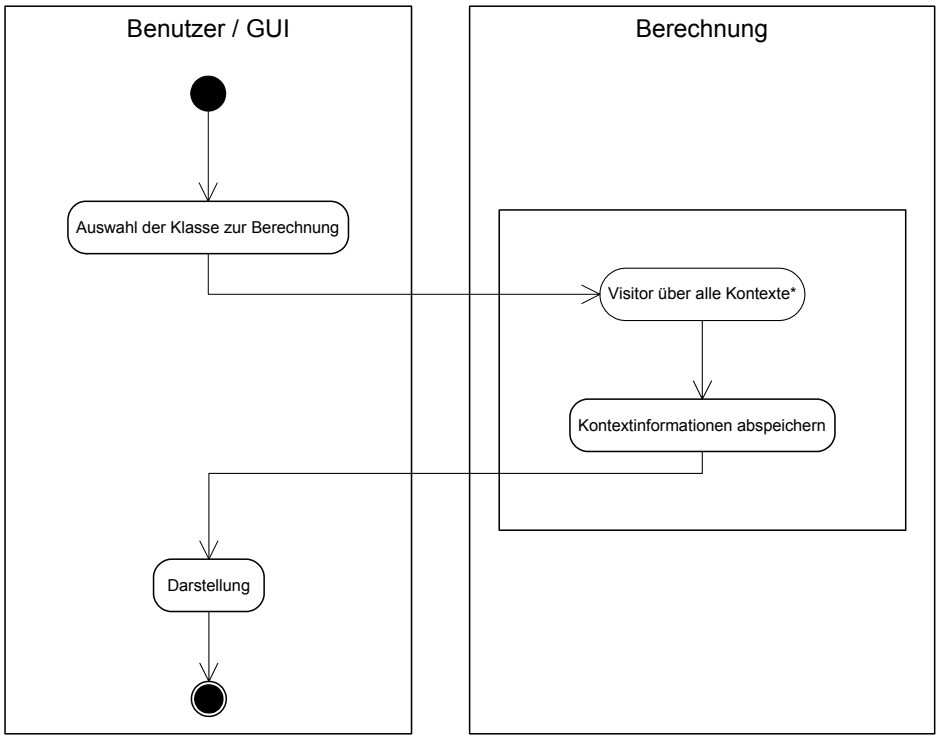


Abbildung 5-22: Programmablauf ContextAnalyzer

Die Berechnungen sind dabei identisch zu den für die Metriken ACD und BCD erforderlichen; für weitere Informationen wird daher auf Kapitel 4 verwiesen. Die Routinen zur Berechnung der Kontexte nutzen insbesondere den im vorigen Abschnitt vorgestellten `PsiSearchHelper`. Mit diesem ist es möglich, Referenzen auf die Methoden der betrachteten Klassen im gesamten Projekt zu finden; von diesen kann dann direkt auf die verwendeten Kontexte zurückgeschlossen werden.

Das ContextAnalyzerPlugin ist ganz auf die Präsentation der gesammelten Daten ausgerichtet. Ein Beispiel für eine mittels des Context Analyzers ausführbare Analyse zeigt die Untersuchung der Klasse `String` aus der Java API (Abbildung 5-23).

Die einzelnen Elemente des Dialogs sollen hier kurz erläutert werden.

- Im oberen Bereich findet sich eine Tabelle mit den paarweise verschiedenen genutzten Methodenteilmengen der Klasse. Aufgelistet sind jeweils die Anzahl der gefundenen Variablen, welche die Kontexte mit der jeweiligen Methodenteilmenge aufspannen, sowie die Anzahl der Methoden dieser Menge. Die Namen der Methoden sind zur Orientierung im Anschluss angegeben. Die Tabelle stellt den Ausgangspunkt für alle weiteren Felder dar; letztere reagieren auf die Auswahl in der Tabelle.
- Die Liste im mittleren Bereich links zeigt die vollständigen Signaturen der in der gewählten Methodenteilmenge enthaltenen Methoden an.
- Die Liste im mittleren Bereich rechts zeigt die Signaturen der Variablen, die einen Kontext mit der momentan gewählten Methodenteilmenge aufspannen. Zusätzlich sind die Klasse und ggf. Methode angegeben, in welcher die Variablen definiert sind.
- Die Tabelle im unteren Bereich des Dialogs zeigt die momentan für die Typisierung der Variablen verwendeten Klassen und/oder Interfaces an. Neben dem Namen des Typs sind Anzahl der Verwendungen (#usages) sowie die Distanz im Sinne der ACD- oder BCD-Metrik (#distance) gelistet. Der beste Typ ist derjenige mit dem geringsten Wert bei #distance.

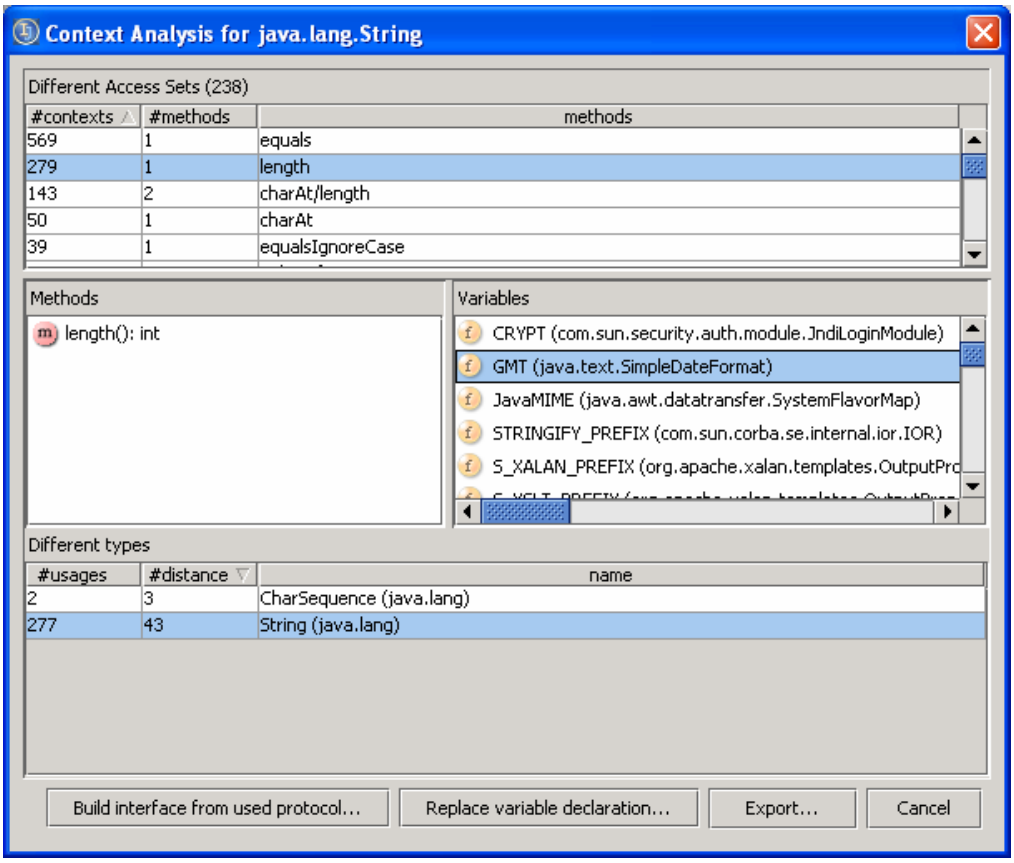


Abbildung 5-23: ContextAnalyzer-Oberfläche

Das ContextAnalyzerPlugin ist ein hochgradig interaktives Werkzeug. An dieser Stelle soll daher zum besseren Verständnis auf die konkrete Implementation verwiesen werden, welche an der in Kapitel 6 angegeben Adresse zum Download bereitsteht.

5.4 Implementierte Metriken

Die Plug-In-Architektur des MetricPlugins macht es auf einfache Art und Weise möglich, zusätzliche Metriken zu implementieren.

Daher sind neben den in dieser Arbeit entstandenen noch weitere Metriken implementiert worden, welche im Zusammenhang mit der interfacebasierten Programmierung und/oder auch der objektorientierten Programmierung im Allgemeinen stehen.

Insgesamt sind folgende zwei Suiten implementiert:

1. Die *Metrik-Suite für die Analyse des Einsatzes von Interfaces in Java* (bestehend aus den sieben in dieser Arbeit definierten Metriken)
2. Der Großteil der in Steimann et al. 2003 definierte Suite der „*Interface-related program metrics*“ (sechs von acht Metriken, siehe unten)

5.4.1 Metrik-Suite für die Analyse des Einsatzes von Interfaces in Java

Die Metrik-Suite besteht aus den Metriken Polymorphic Grade (PG), Decoupling Accessibility (DA), Polymorphic Use (PU), Number of Different Access Sets (NODAS), Actual Context

Distance (ACD), Best Context Distance (BCD) und Interface Minimization Indicator (IMI). Sie sind im Kapitel 4 ausführlich besprochen worden und wie dort definiert im MetricPlugin umgesetzt.

5.4.2 Interface-Related Program Metrics

Die in Steimann et al. 2003 vorgestellte Metrik-Suite besteht aus den Metriken Polymorphic Grade, Versatility, Polymorphic Use, Generality, Popularity, Interface to Class Ratio und Interface typed to Class typed Variables Ratio.

Polymorphic Grade (PG) und Polymorphic Use (PU) wurden in die in Kapitel 4 vorgestellte Suite übernommen und sind wie dort definiert implementiert. Versatility (VERS), Generality (als „Interface Generality“, IGEN) und Popularity (als „Interface Popularity“, IPOP) sind wie in Steimann et al. 2003 beschrieben implementiert (siehe hierzu auch Kapitel 3). Als Komplement zu Interface Popularity und um die Berechnung der zweiten Systemmetrik zu erlauben, ist zusätzlich die Metrik Class Popularity (CPOP) implementiert worden. Diese Metrik zählt die Variablen im Programm, welche mit der betrachteten Klasse deklariert sind.

Die beiden systemweiten Verhältnisse sind nicht als eigenständige Metriken implementiert, können aber aus den Ergebnissen der anderen Metriken direkt abgeleitet werden. Das Interface to Class Ratio ist durch die im MetricPlugin ausgegebenen Typen der untersuchten Klassen/Interfaces sehr einfach zu bestimmen; Interfaces typed to Class typed Variables Ratio lässt sich mittels Interface Popularity und Class Popularity direkt bestimmen.

6 Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Entwicklung einer Metrik-Suite zur Analyse des Einsatzes von Interfaces in Java. Dies ist durch

- eine Motivation zum Einsatz von Interfaces,
- eine Analyse vorhandener Metriken zur Untersuchung der Interfacenutzung,
- die Identifikation fehlender Metriken und
- die Definition neuer Metriken sowie weiterer Methoden zur Analyse der Nutzung von Klassen und Interfaces,
- die Durchführung einer empirischen Evaluation und
- die Implementation eines Tools zur Berechnung der Metriken

erreicht worden.

Zunächst wurde in Kapitel 2 das Konzept des Interfaces beleuchtet; speziell wurde hierzu das Interface-Konstrukt von Java erläutert und ein Vergleich mit anderen Programmiersprachen gezogen. Bei der anschließenden Kategorisierung von Interfaces wurden insbesondere zwei Formen von Interfaces hervorgehoben:

- *partielle* Interfaces. Diese enthalten nur eine Teilmenge der öffentlichen Methoden einer Klasse.
- *totale* Interfaces. Diese enthalten alle öffentlichen Methoden einer Klasse.

Im Anschluss wurde das Konzept des Kontexts entwickelt: Ein Kontext wird bei einer Variablendeklaration aufgespannt und umfasst alle auf dieser Variablen aufgerufenen Methoden. Sowohl partielle als auch totale Interfaces können kontextspezifisch sein; dies hängt von den Methodenteilmengen des (von der Variablen aufgespannten) Kontexts und derjenigen im (von der Variablen als Typ deklarierten) Interface ab.

In Kapitel 2 wurde außerdem eine Motivation zur Nutzung von Interfaces erarbeitet. Zwei Ziele der Nutzung von Interfaces wurden identifiziert:

1. Nutzung von Interfaces zur *vollständigen Entkopplung*; d.h. der ausschließliche Zugriff auf Klassen durch Interfaces. Die Beschaffenheit der Interfaces (partiell oder total) ist dabei irrelevant.
2. Nutzung von Interfaces als *partielle, möglichst kontextspezifische Spezifikationen*: Hierbei sollten die Interfaces einer Klasse möglichst exakt zu den Kontexten passen, in welchen sie verwendet werden.

In Kapitel 3 wurden bereits existierende Metriken und Metrik-Suiten auf ihre Verwendbarkeit für die oben genannten Ziele untersucht. Dabei wurde deutlich, dass zwar bereits einige Metriken existieren, mit welchen das Ziel 1 überprüft werden kann (wenn auch i.d.R. einige Anpassungen notwendig sind), Ziel 2 mit bekannten Metriken jedoch noch nicht überprüfbar ist.
Mit diesem Hintergrundwissen wurde in Kapitel 4 eine Metrik-Suite definiert, welche aus einigen bereits bestehenden sowie insbesondere für die Analyse von Ziel 2 neu entwickelten Metriken und einer Analyseprozedur besteht. Drei Metriken wurden für die Analyse von Ziel 1 definiert, wobei zwei aus einer bereits bestehenden Suite übernommen wurden; vier wurden für die Analyse von Ziel 2 neu entwickelt:

Metriken für Ziel 1:

- *Polymorphic Grade*: Anzahl der implementierten Interfaces einer Klasse.
- *Decoupling Accessibility*: Prozentsatz der Entkopplung der Klasse. 100% geben an, dass alle öffentlichen Features der Klasse über Interfaces zugreifbar sind.
- *Polymorphic Use*: Verhältnis mit einem Interface der Klasse typisierter Variablen zu allen Variablen, die mit der Klasse oder einem Interface der Klasse typisiert sind. 100% geben an, dass keine dieser Variablen mit der Klasse selbst typisiert ist.

Metriken für Ziel 2:

- *Number Of Different Access Sets*: Anzahl paarweise verschiedener genutzter Methodenteilmengen einer Klasse auf Variablen.
- *Actual Context Distance*: Distanz zwischen den deklarierten Typen der Variablen zu den genutzten Methoden auf diesen Variablen. Die Distanz ist definiert als die Anzahl überschüssiger Methoden, die zwar im Typ zur Verfügung stehen, im Kontext der Variablen jedoch nicht genutzt werden.
- *Best Context Distance*: Distanz zwischen den genutzten Methoden auf einer Variablen zum bestmöglichen Typ für diesen Kontext. Der bestmögliche Typ ist derjenige mit der geringsten Distanz zum Kontext – sei es ein implementiertes Interface oder, wenn nicht vorhanden, die Klasse selbst.
- *Interface Minimization Indicator*: Prozentsatz genutzter Methoden eines Interfaces. 100% geben an, dass alle Methoden des Interfaces in mindestens einem Kontext genutzt werden (weswegen das Interface nicht weiter verkleinert werden kann, also minimal ist).

Es wurde zudem ein Analysetool entwickelt: der Context Analyzer, welcher auf einzelne Klassen anwendbar ist und für diese eine genaue Analyse der verwendeten Kontexte sowie aller Typen der diese Kontexte aufspannenden Variablen durchführt.

Die Metriken wurden im Anschluss einzeln mit bereits bekannten Metriken verglichen. Der Vergleich ergab die Übereinstimmung der Metrik Polymorphic Grade mit einer bereits bekannten Metrik und deckte die Verwandtschaft dreier weiterer Metriken mit bereits bestehenden auf. Die kontextspezifischen Metriken NODAS, ACD und BCD scheinen ein neues Konzept darzustellen.

Mit der so identifizierten Metrik-Suite wurden anschließend anhand von vier Open-Source-Projekten, namentlich der Java API, des Eclipse Frameworks, der Neural Net Engine Joone und dem objektrelationalen Framework Cayenne praktische Beispielwerte errechnet. Resultat dieser Auswertung war eine Bestätigung der Ergebnisse früherer Studien zur Interfacenutzung: Diese ist nach wie vor nicht weit verbreitet. Mit Hilfe des Context Analyzers konnten Klassen und Kontexte identifiziert werden, bei welchen die Einführung kontextspezifischer und/oder partieller Interfaces möglich wären.

Die Untersuchung zeigt zudem, dass eine Analyse der Nutzung von Interfaces mit der entwickelten Metrik-Suite Ergebnisse liefert, die eine Identifikation von Klassen zum Refactoring erlauben und in vielen Fällen auch bereits auf ein konkretes Refactoring hinweisen. In diesem Sinne ist also der Zweck der entwickelten Suite erfüllt.

Das Kapitel 5 hat die Implementierung der vorgestellten Metrik-Suite sowie des Context Analyzers als Plug-In für die Entwicklungsumgebung IDEA beleuchtet. Die in diesem Kapitel angesprochenen Plug-Ins sowie damit in Zusammenhang stehende Literatur stehen auf http://www.kbs.uni-hannover.de/fuji/ zum Download bereit.

6.1 Beitrag dieser Arbeit

Diese Arbeit hat mögliche Einsatzzwecke für Interfaces näher beleuchtet und dabei zwei Ziele der interfacebasierten Programmierung definiert. Die erfolgte klare Separierung zwischen Ziel 1 und Ziel 2 kann dem Entwickler helfen zu entscheiden, welchen Weg er einschlagen möchte; die Definition der Ziele gibt ihm Richtlinien zur Hand, die ihn bei der Verfolgung des eingeschlagenen Weges leiten.

Die für Ziel 1 aufgestellten Metriken ermitteln die Nutzung von Interfaces zur Entkopplung und dienen somit der Messung des altbekannten Ziels der Modularisierung und Abstraktion. Die drei beschriebenen Metriken sind mit klaren Zielvorgaben belegt und liefern alle zur Entkopplung nötigen Informationen – von Anbieter-, aber auch von Nutzerseite.

Die Metriken für Ziel 2 sind ebenfalls der Förderung von Modularisierung und Abstraktion verpflichtet, gehen jedoch weiter: Hier wird Neuland beschritten. Diese Metriken und der Context Analyzer sind hervorragend geeignet, um dem Entwickler einen Überblick über die konkrete *Verwendung* seiner Klassen *in Kontexten* im Bezug auf die Nutzung von Interfaces, aber auch im Allgemeinen zu verschaffen. Die identifizierten Kontexte beschreiben exakt, welches Verhalten der Klassen im System an welchen Stellen verwendet wird; die Anzeige der bestmöglichen Typen gibt direkte Hinweise auf mögliche Interfaces.

Neben dem Thema Interfaces wurde also insbesondere das Konzept des Kontexts und, darin enthalten, des kontextspezifischen Interfaces bzw. bestmöglichen Typs eines Kontexts beleuchtet. Es sollte eine Sensibilisierung für das Thema Interfaces, aber auch für das Thema Kontext erreicht werden – die Analyse der Nutzung von Klassen ist ein interessantes Feld, für welches sich der Context Analyzer auch ganz ohne die Betrachtung von Interfaces einsetzen lässt.

Durch die klare Definition möglicher Interfacenutzung und des Konzepts des Kontexts sowie durch die zur Verfügung gestellten Hilfsmittel ist es wesentlich besser als zuvor möglich, geschriebenen Code einzuschätzen und Stellen für die Verbesserung zu finden. Durch die Analyse von Ausreißern, Differenzen in den Metrikwerten und mangelnder Korrelation zwischen den Werten von Klassen – eine Analyse, die in Abschnitt 4.4 beschrieben wurde – lassen sich sowohl für die Erreichung von Ziel 1 als auch Ziel 2 hinderliche Klassen identifizieren.

6.2 Kritische Betrachtung

Während der Untersuchung des Einsatzes von Interfaces ist es jedoch nicht nur als anstrebenswert beschrieben worden, kontextspezifische Interfaces einzuführen oder bereits vorhandene, besser passende Interfaces für die Typdeklaration zu verwenden; es ist auch auf Probleme und Grenzen derartiger Techniken hingewiesen worden, insbesondere auf die auftretenden Kontextwechsel (*context switches*). Probleme bei Kontextwechseln sind bereits in Kapitel 2 erläutert worden. Die von den Metriken und dem Context Analyzer identifizierten möglichen Typen für Kontexte, z.B. die „bestmöglichen Typen“ des Context Analyzers, sind zwar innerhalb der Grenzen des einen betrachteten Kontexts ideal; dies bedeutet jedoch nicht automatisch, dass der Typ im größeren Zusammenhang ebenfalls ideal sein muss, denn Kontexte stehen für gewöhnlich nicht alleine im Raum: Variablen, welche die Kontexte aufspannen, werden an andere Methoden weitergegeben oder ihrerseits von Methoden zurückgeliefert.

An jedem dieser Übergabepunkte kommt es i.d.R. zu Kontextwechseln, die dann, wenn der Typ stets kontextspezifisch sein soll, mit einem Typwechsel des Objekts verbunden sein können: An solchen Stellen müssen Casts eingesetzt werden. Die Untersuchung, wie oft derartige Kontextwechsel stattfinden und welche Methodik für einen möglichst nahtlosen und

allgemeingültigen Übergang geschaffen werden kann – denn explizite Kontextwechsel können durchaus als Chance verstanden werden, die im System vorhandenen Kontexte noch deutlicher herauszustellen – ist ein nächster Schritt, der mit den in dieser Arbeit aufgestellten Definitionen in Angriff genommen werden kann.

Bei Klassen mit hoher Anzahl unterschiedlicher genutzter Methodenteilmengen ist es zudem unmöglich, für jeden Kontext ein eigenes Interface einzuführen. Dies ist durch die in Kapitel 2 aufgestellte Beziehung (Abbildung 2-13) verdeutlicht: Kontextspezifische/partielle Interfaces bilden das eine Extrem, kontextübergreifende/totale Interfaces das andere. In der Mitte stehen kontextübergreifende/partielle Interfaces, welche idealerweise den *Rollen einer Klasse* entsprechen. Dies bedeutet, dass die korrekte Identifikation von Kontexten im Regelfall nicht zur Einführung absolut kontextspezifischer Interfaces führen wird, sondern zu besser passenden, kontextübergreifenden/partiellen Interfaces.

Die hier aufgestellten Metriken und der Context Analyzer können also nur die Tür zu einer besseren Nutzung von Interfaces öffnen: Hindurchgehen muss der Entwickler selbst.

6.3 Ausblick

Als Ausblick können folgende weitere Forschungsbereiche identifiziert werden:

- Zunächst ist eine weitere Evaluation der Metriken und des Context Analyzers nötig – insbesondere auch von Entwicklern größerer OO-Projekte selbst – um deren Nützlichkeit und Eignung für den angestrebten Zweck zu untersuchen, besonders vor dem Hintergrund der Kontextwechsel.

- Für die Nutzungsanalyse von Klassen – welche auch unabhängig von Interfaces möglich ist – ist eine graphische Visualisierung der genutzten Methodenteilmengen in Form der in Kapitel 2 gezeigten Verbände möglich. Eine solche Visualisierung kann dem Programmierer einen schnellen Überblick über die Verwendung seiner Klassen verschaffen.

- Der beste Kompromiss zwischen einem einzelnen, totalen/kontextübergreifenden und vielen partiellen, kontextspezifischen Interfaces ist bisher durch den Entwickler selbst zu entwickeln. Clustering-Algorithmen könnten anhand der aufgestellten Verbände automatisch Lösungsvorschläge erarbeiten, in welchen ein möglichst sinnvoller Kompromiss gefunden wird.

7 Schlussbemerkung

Mit der vorgestellten Metrik-Suite, welche durch eine ausführliche Betrachtung sowohl des Interface-Konstrukts als auch von Metriken allgemein in Sinn und Zweck begründet wurde und mit den Metriken zur Kontextanalyse Neuland betritt, meine ich einen Beitrag zum Verständnis der Nutzung von Interfaces in großen Softwareprojekten geleistet zu haben. Die in dieser Arbeit genannten Prinzipien und Ziele der interfacebasierten Programmierung sowie die Metriken zur Überprüfung dieser Anwendung können direkt zur Entwicklung von Java-Programmen mit einer besseren Nutzung von Interfaces eingesetzt werden.

Verzeichnisse

A) Abbildungsverzeichnis

B) Tabellenverzeichnis

C) Literaturverzeichnis

Albahari, B. (2000). A comparative Overview of C#, http://genamics.com/developer/csharp_comparative.htm#12, Zuletzt besucht: 16.07.2003

Albrecht, A. J. (1979). Measuring Applications Development Productivity. Proceedings of the IBM Applications Devision Joint SHARE/GUIDE Symposium, Monterey, CA.

Apple (2000). The Objective-C Programming language, http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf, Zuletzt besucht: 16.07.2003

Apple (2003). Cocoa Documentation, http://developer.apple.com/documentation/Cocoa/Cocoa.html, Zuletzt besucht: 31.07.2003

Basili, V. R., Caldiera, G., et al. (1994). The Goal Question Metric Approach. Encyclopedia of Software Engineering, John Wiley & Sons.

Basili, V. R. und Weiss, D. M. (1984). A Methodology for Collecting Valid Software Engineering Data. IEEE Transactions on Software Engineering SE-10(6): 728-738.

Baumgartner, G. und Russo, V. F. (1995). Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++. Software-Practice and Experience 25(8): 863-889.

Boehm, B. (1981). Software Engineering Ergonomics, Prentice Hall.

Boehm, B., Brown, J. R., et al. (1978). Characteristics of Software Quality. TRW Series of Software Technology, Volume 1, North-Holland Publishing Company, Amsterdam, New York, Oxford, 1978.

Briand, L., Morasca, S., et al. (1996). Property Based Software Engineering Measurement. IEEE Transactions on Software Engineering SE-22(1): 68-85.

Brito e Abreu, B. F. und Carapuca, R. (1994). Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. Journal of Systems and Software 26: 87-96.

Brito e Abreu, F., Goulao, M., et al. (1995). Towards the Design Quality Evaluation of Object-Oriented Software Systems. Fifth International Conference on Software Quality, Austin, Texas.

Chidamber, S. und Kemerer, C. (1991). Towards a Metrics Suite for Object Oriented Design. Proceedings of the OOPSLA 1991, Phoenix, Arizona, USA.

Chidamber, S. und Kemerer, C. F. (1994). A metrics suite for object oriented design. IEEE Transactions on Software engineering 20(6): 476-493.

Clements, P. C. und Northrop, L. M. (1996). Software Architecture: An Executive Overview. Pittsburgh, PA 15213, Sofware Engineering Institute, Carnegie Mellon University.

Cook, W. R. (1992). Interfaces and Specifications for the Smalltalk-80 collection classes. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Vancouver, British Columbia, Canada.

CORBAIDL Information technology - open distributed processing - interface definition language. ISO/IEC 14750, 1999.

De Champeaux, D. (1997). Object-Oritented Development Process and Metrics, Prentice Hall PTR.

Differding, C., Hoisl, B., et al. (1996). Technology Package for the Goal Question Metric Paradigm, Department of Computer Science, University of Kaiserslautern, Germany.

Dijkstra, E. W. (1968). The Structure of "THE"-Multiprogramming System. Communications of the ACM 18(8): 453-457.

EssentialMetrics (Power Software). http://www.powersoftware.com/em/, Zuletzt besucht: 26.07.2003

Fenton, N. E. (1991). Software metrics: A Rigorous Approach, Chapman and Hall, London.

Fowler, M. (2000). Refactoring: Improving the design of existing code, Addison-Wesley.

Gamma, E., Helm, R., et al. (1995). Design Patterns, Addison-Wesley.

Gilb, T. (1977). Software metrics. Cambridge, Massachusetts, Winthrop Publishers.

Gosling, J., Joy, B., et al. (2000). The Java Language Specification, Second Edition, Addison-Wesley.

Graham, I. M. (1995). Migrating to Object Technology. Workingham, UK, Addison-Wesley.

Halstead, M. H. (1977). Elements of Software Science, Elsevier North-Holland, Inc., New York.

Heijenoort, J. V. (1967). From Frege to Gödel, Harvard University Press.

Henderson-Sellers, B. (1996). Object-oriented metrics: Measures of complexity, Prentice Hall PTR.

Henderson-Sellers, B. und Edwards, J. M. (1994). Book Two of Object Oriented Knowledge: The Working Project, Prentice Hall.

Hindel, B. (1996). Qualität ist meßbar: Software-Metriken. Design&Elektronik 24: 50-55.

Holmevik, J. R. (1994). Compiling SIMULA: A Historical Study of Technological Genesis. IEEE Annals of the History of Computing 18(4): 25-37.

IEEE Standards Collection, S. E. (1992). Standard for a Software Quality Metrics Methodology. IEEE Std 1061.

IntelliJIDEA (JetBrains). http://www.intellij.com/idea, Zuletzt besucht: 24.07.2003

Isolec (1995). Open Distributed Processing - Reference Model - Part 2: Foundations International Standard 10746-2 Itu-T Recommendation X.902.

JDepend (Clarkware). http://www.clarkware.com/, Zuletzt besucht: 26.07.2003

JetBrains (2003). Introduction to IDEA 3.0 Plug-Ins, http://www.intellij.com/docs/PlugIns.pdf, Zuletzt besucht: 21.07.2003

JRefactory http://jrefactory.sourceforge.net/, Zuletzt besucht: 26.07.2003

Kay, A. (1993). The Early History of Smalltalk. ACM SIGPLAN 28(3): 69-75.

Kemerer, C. F. (1999). Metrics for Object Oriented Software: A retrospective. Sixth International Symposium on Software Metrics, Boca Raton, Florida, USA.

Knoernschild, K. (2001). Dependency Inversion Principle, http://www.kirkk.com/adobe/dip.pdf, Zuletzt besucht: 03.08.2003

Kolence, K. W. (1975). Software physics. Datamation: 48-51.

Kolewe, R. (1993). Metrics in object-oriented design and programming. Software Development 1: 53-62.

Krasner, G. und Pope, S. (1988). A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. Journal of Object Oriented Programming 1(3): 26-49.

Laemmel, A. und Shooman, M. (1977). Statistical (Natural) Language Theory and Computer Program Complexity. POLY/EE/E0-76-020, Dept. of Electrical Engineering and Electrophysics, Polytechnic Institute of New York, Brooklyn, August 15, 1977.

Li, W. und Henry, S. (1993). Object-oriented metrics that predict maintainability. Journal of Systems and Software 23(2): 111-122.

Lichter, H. (2003). Software-Qualitätssicherung (Vorlesung), http://www-lufgi3.informatik.rwth-

aachen.de/LUFGI3/EDUCATION/SS03/VL_SQS/INFOS/index.html, Zuletzt besucht: 12.07.2003

Lieberherr, K. M. (2003). Law of Demeter (Website), http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/general-formulation.html, Zuletzt besucht: 30.07.2003

Lieberherr, K. M. und Holland, I. (1989). Assuring Good Style for Object-Oriented Programs. IEEE Software: 38-48.

Littlefair, T. (2001). An investigation into the use of software code metrics in the industrial software development environment. Faculty of Communications, Health and Science, Edith Cowan University, Mount Lawley Campus.

Lorenz, M. (1993). Object-Oriented Software Development: A practical guide, Prentice Hall PTR.

Lorenz, M. und Kidd, J. (1994). Object-oriented software metrics, Prentice Hall PTR.

Marciniak, J. J. (1994). Encyclopedia of Software Engineering, John Wiley & Sons, Inc.

Martin, R. C. (1994). OO Design Quality Metrics, http://www.objectmentor.com/resources/articles/oodmetrc.pdf, Zuletzt besucht: 26.07.2003

Martin, R. C. (2002). Agile Software Development. Principles, Patterns, and Practices. New Jersey, Prentice Hall PTR.

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering SE-10(6): 308-320.

Meißner, A. (2003). Refactorings für den systematischen Einsatz von Interfaces in Java (Bachelorarbeit). Institut für Informationssysteme. Hannover, Universität Hannover.

Microsoft (2003). Microsoft Interface Definition Language, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_start_page.asp, Zuletzt besucht: 16.07.2003

Miller, J., Darroch, G., et al. (1995). Changing programming paradigm - an empirical investigation, Chapter 8 in Software Quality and Productivity. Theory, Practice, Education and Training. M. Lee, B.-Z. Barta and P. Juliff. London, Chapman&Hall: 62-65.

Mills, E. E. (1988). Software Metrics, SEI Curriculum Module SEI-CM-12.1.1. Pittsburgh, PA, Carnegie Mellon University.

Morris, K. L. (1989). Metrics for Object-Oriented Software Development (Master of Science in Management). Massachusetts Insitute of Technology.

Naughton, P. (2003). Java was strongly influenced by Objective-C, http://www.cs.umd.edu/users/seanl/stuff/java-objc.html, Zuletzt besucht: 16.07.2003

OCFAQ Objective-C Newsgroup Frequently Asked Questions, http://www.cs.uu.nl/wais/html/na-dir/Objective-C/answers.html, Zuletzt besucht: 16.07.2003

OptimalJ (CompuWare). http://www.compuware.com/products/optimalj/, Zuletzt besucht: 26.07.2003

Palm, J. D., Anderson, K. M., et al. (2003). Investigating the Relationship Between Violations of the Law of Demeter and Software Maintainability. Second International Conference on Aspect-Oriented Software Development, Boston, Massachusetts, USA.

Parnas, D. L. (1972). On the Criteria for Decomposing Systems into Modules. Communications of the ACM 15(12): 1053-1058.

Patenaude, J.-F., Merlo, E., et al. (1999). Extending Software Quality Assessment Techniques to Java Systems. Seventh International Workshop On Program Comprehension, Pittsburgh, Pennsylvania.

Rajaraman, C. und Lyu, M. R. (1992). Some coupling measures for C++ programs. Proceedings of TOOLS USA 92, Prentice Hall PTR.

RefactorIT (Aqris Software). http://www.refactorit.com/, Zuletzt besucht: 26.07.2003

Rocacher, D. (1988). Metrics Definitions for Smalltalk. Project ESPRIT 1257, MUSE MP9A, 1988.

Roy, G. und Roy, G. (2001). On the Applicability of Weyuker Property 9 to Object Oriented Structural Inheritance Complexity metrics. IEEE Transactions on Software Engineering 27(4).

Rubey, R. J. und Hartwick, R. D. (1968). Quantitative Measurement Program Quality. ACM National Computer Conference: 671-677.

Sadeh, B. und Ducasse, S. (2002). Adding Dynamic Interfaces to Smalltalk. Journal of Object-Oriented Programming 1(1): 63-79.

Shan, Y.-P. (1995). Smalltalk on the rise: Introduction to the Special Section. Communications of the ACM 38(10): 102-104.

Sharble, R. C. und Cowen, S. S. (1993). The object-oriented brewery: a comparison of two object-oriented development methods. ACM SIGSOFT Software Engineering Notes 18(2): 60-73.

Simon, F., Steinbrückner, F., et al. (2001). Metrics based refactoring. Conference on Software Maintenance and Reengineering (CSMR), Lissabon, Portugal.

Steimann, F. (2000). Formale Modellierung mit Rollen (Habilitationsschrift), Universität Hannover.

Steimann, F. (2001). Role=Interface: A merger of concepts. Journal of Object-Oriented Programming 14: 23-32.

Steimann, F. (2003). What Interfaces Can Be: Investigating the Use of Interfaces in Java Programming**:** unveröffentlichtes Manuskript.

Steimann, F., Siberski, W., et al. (2003). Towards the Systematic Use of Interfaces in JAVA Programming. Proceedings of PPPJ'2003, Kilkenny, Ireland.

Tegarden, D. P. und Sheetz, S. D. (1992). Object-oriented system complexity: an integrated model of structure and perceptions. OOPSLA '92 Workshop on Metrics for Object-Oriented Software Development, HICSS-92, IEEE, San Diego, unpublished.

Van Emden, M. H. (1971). An Analysis of Complexity. Mathematisches Zentrum, Amsterdam.

Weyuker, E. (1988). Evaluating software complexity measures. IEEE Transactions on Software Engineering 14: 1357-1365.

Whitmire, S. A. (1997). Object Oriented Design Measurement, John Wiley&Sons, Inc.

Willcock, J., Siek, J., et al. (2001). Caramel: A Concept Representation System for Generic Programming. OOPSLA 2001 C++ Template Workshop, Indiana University, Bloomington, USA.

Wolverton, R. W. (1974). The Cost of Developing Large-Scale Software. IEEE Transactions on Computer C-23(6): 615-636.

Xenos, M., Stavrinoudis, D., et al. (2000). Object-oriented metrics - a survey. Proceedings of the FESMA 2000, Madrid, Spain.

Zanden, V., Plank, J. S., et al. (2001). CS302 Lecture notes -- Introduction to Object Oriented Programming, http://www.cs.utk.edu/~plank/plank/classes/cs302/302/notes/Intro/, Zuletzt besucht: 12.08.2003

Zuse, H. (1997). A framework for software measurement, de Gruyter.

Anhang

A) Metrik-Suiten

A) i) Die Metrik-Suite von Chidamber/Kemerer

Die Besonderheit der CK-Suite liegt nicht allein in ihrem frühen Erscheinen (1991) und der direkten Ausrichtung auf die Objektorientierung, sondern auch in der Art und Weise, wie die Metriken vorgestellt werden. Eine Suite von sechs Metriken wird vorgeschlagen, welche u.A. Größe, Kohäsion und Kopplung messen. Zu jeder der Metriken werden zum einen empirische Daten aus zwei Industrieprojekten präsentiert, zum anderen werden die Metriken mit Hilfe der Kriterien von Weyuker (siehe unten) formal validiert. Insbesondere dieser letzte Aspekt hat große Aufmerksamkeit auf sich gezogen.

Neben dieser formalen Validierung – der eine Begründung der gewählten Metriken mittels Messtheorie vorangeht – legen Chidamber und Kemerer ihrer Metrik-Suite auch die ontologischen Prinzipien von Bunge zugrunde. Bunges 1977 vorgelegte „Treatise on Basic Philosophy“ begründet das Konzept der Objekte. Auf die Relevanz dieses Werkes für die Objektorientierung haben viele Autoren hingewiesen (Chidamber/Kemerer 1994, S. 378r).

Die Begründung auf Bunges Werk ist allerdings nicht ohne Kritik, wie Henderson-Sellers anführt (Henderson-Sellers 1996, S. 151). Graham (Graham 1995, S. 410) kritisiert, dass die von Bunge vorgelegte Definition von Objekten diesen ihre Identität über ihre Eigenschaften zuweist, während die Identität eines Objektes in der Objektorientierung unabhängig vom Wert seiner Eigenschaften ist.

Chidamber und Kemerer haben die 1991 vorgelegte Suite (Chidamber/Kemerer 1991) in einer weiteren Arbeit 1994 noch einmal überarbeitet (Chidamber/Kemerer 1994); hier soll diese jüngere Version vorgestellt werden. Definiert sind darin folgende Metriken (Tabelle A-1).

Tabelle A-1: CK-Suite

Metrik	Kategorie	Erläuterung
WMC	Komplexität	Weighted Methods per Class (Gewichtete Methoden pro Klasse). Ergebnis dieser Metrik ist die Summe über alle Methoden der Klasse, wobei pro Methode eine nicht näher definierte – d.h. beliebig wählbare – Funktion die Gewichtung der Methode bestimmt (z.B. anhand der inneren Methodenaufrufe o.Ä.).
DIT	Vererbung	Depth of Inheritance Tree (Tiefe des Vererbungsbaums). Diese Metrik berechnet die Länge des Vererbungsbaums von der aktuell betrachteten Klasse zur Wurzel des Baums; in Fällen von Mehrfachvererbung die maximale Länge.
NOC	Vererbung	Number of Children (Anzahl der Kinder). Diese Metrik bestimmt die Anzahl von direkten Subklassen der betrachteten Klasse.
CBO	Kopplung	Coupling between Object Classes (Kopplung zwischen Objektklassen). Ergebnis dieser Metrik ist die Anzahl der Klassen, zu welcher die momentan betrachtete Klasse Verbindungen aufweist.

RFC	Kopplung	Response for a Class (Reaktion auf eine Klasse). Das „Response Set" einer Klasse wird definiert als diejenigen Methoden, welche als Antwort auf eine an ein Objekt der Klasse gerichtete Nachricht aufgerufen werden. RFC ist die Summe dieser Methoden.
LCOM	Kohäsion	Lack of Cohesion in Methods (Fehlende Kohäsion in Methoden). Diese Metrik bestimmt den inneren Zusammenhalt einer Klasse, indem analysiert wird, welche Mengen von Methoden auf welche Mengen von Instanzvariablen zugreifen (s.u.).

Die in der CK-Suite definierten Metriken sollen nun im Einzelnen besprochen werden.

Weighted Methods per Class

Wie oben beschrieben ist WMC definiert als die Summe über alle Methoden der Klasse, wobei die Methoden entsprechend ihrer Komplexität gewichtet sein können.

Chidamber und Kemerer geben keine Gewichtungsfunktion vor, merken jedoch an, dass bei einer Methodenkomplexität von jeweils 1 der Wert der WMC-Metrik der Anzahl Methoden der Klasse entspricht. WMC korrespondiert direkt zur Komplexität eines *Dings*, wie es bei Bunge beschrieben ist. Chidamber und Kemerer führen an, dass Klassen mit vielen Methoden eine erhöhte Komplexität aufweisen und größeren Einfluss auf ihre Subklassen ausüben. Darüber hinaus ist die Wahrscheinlichkeit höher, dass derartige Klassen sehr applikationsspezifisch gehalten sind und aus diesem Grunde die Möglichkeit der Wiederverwendung einschränkt ist.

WMC ist ein Kompositum aus zwei weiteren Metriken aus Ken Morris' Dissertation (1988). Die Metriken „Methods/Object Class" und „Method Complexity" wurden für WMC zusammengefasst (Kemerer 1999).

Depth of Inheritance Tree

Wie oben beschrieben berechnet DIT die Länge des Vererbungsbaums von der aktuell betrachteten Klasse zur Wurzel des Baums; in Fällen von Mehrfachvererbung die maximale Länge des Baums.

DIT korrespondiert zu Bunges Vorstellung vom Gültigkeitsbereich von Attributen. Je tiefer eine Klasse in der Hierarchie steht, desto schwieriger ist es, ihr Verhalten vorherzusagen; ebenso bedeuten tiefe Vererbungsbäume eine erhöhte Komplexität, da mehr Methoden und Klassen beteiligt sind. Allerdings erhöht sich durch einen tiefen Vererbungsbaum auch die potentielle Wiederverwendung.

Depth of Inheritance Tree korrespondiert zu „Inheritance Tree Depth" aus Morris' Dissertation.

Number of Children

Die NOC-Metrik gibt für eine Klasse die Anzahl der direkten Subklassen an.

Wie bereits DIT korrespondiert auch NOC zur Vorstellung vom Gültigkeitsbereich von Attributen von Bunge. Chidamber/Kemerer merken an, dass Klassen mit einer hohen Anzahl von Subklassen großen Einfluss auf das System haben und damit stärker getestet werden sollten. Zudem steigt die Wahrscheinlichkeit für einen Missbrauch der Vererbung mit der Anzahl der Subklassen. Andererseits korrespondiert NOC auch zur Wiederverwendung, denn je

höher die Anzahl der Subklassen, desto mehr Attribute und Methoden werden wiederverwendet.

NOC ist eine Neuentwicklung; sie korrespondiert zu keiner in Morris Dissertation angegebenen Metrik.

Coupling between Object Classes

Wie oben bereits beschrieben berechnet diese Metrik die Anzahl der Klassen, zu welcher die momentan betrachtete Klasse Verbindungen aufweist.

Zwei Klassen sind dabei gekoppelt – abgeleitet von Bunge – wenn die Methoden einer Klasse Instanzvariablen oder Methoden einer anderen Klasse nutzen. Wie Henderson-Sellers 1996 (S. 112) anmerkt, schließt die Definition der Kopplung von Chidamber/Kemerer die durch Vererbung entstehende Kopplung mit ein.

Chidamber/Kemerer sehen eine exzessive Kopplung zwischen Klassen als schädlich für ein Design an; die Möglichkeit der Wiederverwendung sinkt, da die Klassen nur schwer in anderen Kontexten eingesetzt werden können. Aus diesem Grund sollten die Verbindungen zwischen Klassen auf ein Minimum beschränkt werden – dieses Prinzip wird in *Violations of the Law of Demeter* (siehe unten) zur Leitlinie erhoben. Die Messung der Anzahl an Verbindungen zu anderen Klassen kann zudem als Indikator für die nötige Testkomplexität genutzt werden.

Die CBO-Metrik kann in gewisser Weise als komplementär zu DIT/NOC gesehen werden. Letztere messen die durch Vererbung entstandenen Verbindungen zu anderen Klassen, während CBO (auch) durch „traditionelle“ Methodenaufrufe entstandene Verbindungen zählt.

CBO ist die Umsetzung der in Morris' Dissertation definierten Metrik „Coupling“ (Kemerer 1999).

Response For a Class

Die Metrik RFC berechnet die Anzahl der Methoden, welche in Reaktion auf von außen in die Klasse eingehenden Nachrichten ausgeführt werden müssen. Dies schließt zum einen alle Methoden der Klasse mit ein, zum anderen alle Methoden, welche von diesen aus aufgerufen werden.

Mathematisch ist RFC von Chidamber/Kemerer wie folgt definiert: $RFC = \left|\{M\} \cup_{all\ i} \{R_i\}\right|$, wobei

- {M} die Menge der Methoden in der Klasse darstellt und
- $\{R_i\}$ die Menge der Methoden angibt, die von Methode i der Klasse aufgerufen wird.

In einer Fußnote wird verdeutlicht, dass $\{R_i\}$ nur die erste Stufe von aufgerufenen Methoden enthält, folglich also nicht rekursiv definiert ist.

Die durch RFC berechnete Anzahl potentiell aufgerufener Methoden korrespondiert laut Chidamber/Kemerer zum Wartungs- und Testaufwand für eine Klasse, da sich mit der Anzahl aufgerufener Methoden die Komplexität der Klasse erhöht.

Die RFC-Metrik leitet sich nicht aus einer der Morris-Metriken ab.

Lack Of Cohesion in Methods

Die LCOM-Metrik bestimmt, wie oben bereits beschrieben, den inneren Zusammenhalt einer Klasse, indem analysiert wird, welche Mengen von Methoden auf welche Mengen von Instanzvariablen zugreifen. Greifen alle Methoden auf alle Instanzvariablen zu, so ist die Kohäsion hoch; greift jede Methode nur auf eine (und jeweils eine andere) Instanzvariable zu, ist keine Kohäsion vorhanden.

Die formale Definition ist $LCOM = \max(P - G,0)$, wobei P die Anzahl von Methodenpaaren ist, welche keine Instanzvariablen teilen (Schnittmenge ist leer) und G die Anzahl von Methodenpaaren ist, welche auf mindestens eine gemeinsame Instanzvariablen zurückgreifen (Schnittmenge enthält mindestens ein Element). Je größer LCOM, desto höher die interne Kohärenz der Klasse.

Chidamber/Kemerer merken an, dass eine geringe Kohäsion die Komplexität erhöht; die entsprechende Klasse sollte vermutlich in zwei oder mehr (Sub-)klassen untergliedert werden. Kohäsion bedeutet Kapselung und ist damit anstrebenswert. Es wird zudem angemerkt, dass jede Art der Messung von Verschiedenheit innerhalb einer Klasse Designfehler in der Klasse aufdecken kann.

LCOM korrespondiert zu den Kohärenz-Metriken aus Morris' Dissertation (LCOM baut auf dem „Fan-in"-Faktor auf).

A) ii) Die MOOD-Suite

Die von Brito e Abreu und Carapuça vorgestellte Suite – MOOD steht für „Metrics for Object Oriented Design" – enthält wie die CK-Suite sechs Metriken, allerdings mit einem anderen Fokus (Brito e Abreu/Carapuca 1994, Brito e Abreu et al. 1995). Brito e Abreu und Carapuça stellen vor der Definition der Metriken sieben Kriterien vor, welche ihrer Meinung nach von Metriken erfüllt werden müssen (siehe Abschnitt B) ii)) und definieren ihre Metriken nach diesen Kriterien.

Der Schwerpunkt der MOOD-Suite liegt, so die Autoren in der Einleitung, auf den Hauptabstraktionen des objektorientierten Paradigmas – der Vererbung, der Kapselung, des Information Hidings und der Polymorphie. Insbesondere wird auch Wert auf die Wiederverwendbarkeit des Codes gelegt, da dies zu höherer Softwarequalität und Entwicklungsproduktivität führt.

Eine Besonderheit der Metriken besteht darin, dass alle Metriken als Quotienten ausgedrückt sind. Sie sind dimensionslos und in Prozent angeben – 0% steht jeweils für „keine Nutzung", 100% für „maximale Nutzung" (Brito e Abreu et al. 1995). Weitere Vorteile sind die formale Definition und die Sprachunabhängigkeit – es wird lediglich ein OO-System vorausgesetzt.

Die Metriken sind als Systemmetriken definiert, d.h. jede Metrik liefert einen einzelnen Wert für das Gesamtsystem. Brito e Abreu et al. definieren folgende Metriken (Tabelle A-2).

Tabelle A-2: MOOD-Suite

Metrik	Kategorie	Erläuterung
MHF	Kapselung	Method Hiding Factor (Methoden-Verbergungs-Faktor). Diese Metrik gibt den Prozentsatz der verborgenen Methoden an den Gesamtmethoden der Klassen an.

AHF	Kapselung	Attribute Hiding Factor (Attribut-Verbergungs-Faktor). Diese Metrik gibt den Prozentsatz der verborgenen Attribute an den Gesamtattributen der Klassen an.
MIF	Vererbung	Method Inheritance Factor (Methoden-Vererbungs-Faktor). Diese Metrik gibt an, wie viel Prozent der Methoden der Klassen geerbt sind.
AIF	Vererbung	Attribute Inheritance Factor (Attribut-Vererbungs-Faktor). Diese Metrik gibt an, wie viel Prozent der Attribute der Klassen geerbt sind.
COF	Kopplung	Coupling Factor (Kopplungsfaktor). COF setzt die tatsächlich vorhandenen Verbindungen zwischen Klassen zu allen potentiell möglichen ins Verhältnis.
PF	Polymorphie	Polymorphism Factor (Polymorphiefaktor). Hierbei wird die Anzahl überschriebener Methoden der Klassen zur Anzahl insgesamt vorhandener Methoden der Klassen ins Verhältnis gesetzt.

Hiding Factors (AHF/MHF)

Wie oben bereits beschrieben gibt MHF den Prozentsatz an verborgenen Methoden der Gesamtmethoden der Klassen an; AHF dagegen den Prozentsatz an verborgenen Attributen zu allen Attributen der Klassen. Mathematisch definieren die Autoren MHF wie folgt:

$$MHF = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

wobei TC die Gesamtanzahl der Klassen im System darstellt, M_h die die Anzahl verborgener Methoden in Klasse C_i und M_d die Gesamtanzahl von Methoden in C_i angibt; AHF äquivalent dazu als

$$AHF = \frac{\sum_{i=1}^{TC} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

mit A_h als die Anzahl der verborgenen Attribute in Klasse C_i sowie A_d als Anzahl aller Attribute der Klasse C_i.

Laut Brito e Abreu und Carapuça messen AHF und MHF die Nutzung des Information-Hiding-Prinzips, welches durch den Mechanismus der Kapselung in den OO-Sprachen umgesetzt wird (Brito e Abreu et al. 1995). Die Nutzung des Information Hidings für die Entwicklung von OO-Systemen erlaubt u.A.

- *mit der Komplexität fertig zu werden, indem komplexe Komponenten als „black boxes" betrachtet werden,*
- *durch die Verfeinerung der Implementation provozierte „Seiteneffekte" zu reduzieren,*
- *die Nutzung eines Top-Down-Vorgehens,*
- *Systeme inkrementell zu testen und zu integrieren* (Brito e Abreu et al. 1995, S. 13, eigene Übersetzung).

Die AHF-Metrik betrachtet Attribute: hier macht die häufige oder sogar ausschließliche Nutzung der Kapselung Sinn – idealerweise sind alle Attribute verborgen und nur über Get-

ter/Setter zugreifbar. Folglich sind hohe Werte der AHF-Metrik ideal; bei sehr geringen sollte die betreffende Klasse dagegen näher untersucht werden.

Die MHF-Metrik betrachtet dagegen Methoden. Eigentlich ist die Anzahl öffentlicher Methoden einer Klasse ist auch ein Anzeiger für ihre Funktionalität – wird diese erhöht, erniedrigt sich jedoch der Wert der MHF-Metrik. Brito e Abreu et al. lösen diesen Widerspruch auf, indem auf den Top-Down-Approach verwiesen wird: Eine öffentliche Methode sollte immer nur die „Spitze des Eisbergs“ sein, welche zwangsläufig zu verborgenen Methoden führt, wenn eine schrittweise Zergliederung der Methode durchgeführt wird.

Sinnvolle Werte der MHF-Metrik liegen folglich nicht nahe bei 0 – denn dies zeigt eine nur schwach gekapselte Implementation an – und ebenso nicht bei 1, denn dies zeigt eine nur geringe Funktionalität an. Die idealen Werte liegen daher in der Mitte des Intervalls [0,1].

Inheritance Factors (AIF/MIF)

Wie bereits oben beschrieben geben AIF und MIF jeweils an, wie viel Prozent der Attribute bzw. Methoden der Klassen geerbt sind. Mathematisch definieren die Autoren AIF wie folgt (aus Brito e Abreu et al. 1995):

$$AIF = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

A_i berechnet die geerbten Attribute, A_a alle Attribute einer Klasse. MIF ist äquivalent dazu definiert als

$$MIF = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

mit M_i als Anzahl der geerbten Methoden und M_a als Anzahl aller Methoden einer Klasse. Die Anzahl der geerbten Attribute bzw. Methoden enthält nur nicht veränderte Methoden, d.h. überschriebene Methoden sind nicht enthalten.

Laut Brito e Abreu und Carapuça messen AIF bzw. MIF die in OO-Systemen eingesetzte Vererbung, also die Möglichkeit, eine Klasse von Superklassen erben zu lassen und so Teile aus deren Implementation zu übernehmen, was eine direkte Wiederverwendung erlaubt sowie weitere Vorteile (Polymorphie) mit sich bringt.

Die Autoren sehen – wie bei MHF – auch bei AIF und MIF mittlere Werte im Intervall [0,1] als erstrebenswert an, da ein Wert nahe bei 1 auf umfangreiche Vererbungshierarchien schließen lässt, die wiederum Verständlichkeit und Testbarkeit einschränken, während ein Wert nahe bei 0 eine nur sehr schwache Nutzung von Vererbung zeigt.

Ein Wert von 0 zeigt in jedem Fall an, dass keinerlei Vererbung genutzt wird (d.h. es existiert keine Vererbungshierarchie – oder alle geerbten Methoden werden überschrieben).

Coupling Factor (COF)

Wie oben bereits beschrieben setzt COF die tatsächlich vorhandenen Verbindungen zwischen Klassen zu allen potentiell möglichen Verbindungen ins Verhältnis. Dabei wird nur die Kopplung über Nachrichten berücksichtigt, nicht die Vererbung. Die Autoren definieren eine Kopplung – ausgedrückt als die Tatsache, dass eine Klasse Klient einer anderen Klasse ist –

wie folgt: C_c ist Klient von C_s, wenn C_c mindestens eine Referenz auf ein Feature von C_s enthält. Mögliche Referenzen sind

- Methodenaufrufe aus Klasse C_c zu Klasse C_s.
- Öffentliche oder private globale Variablen des Typs C_s in C_c.
- Öffentliche oder private Methodenparameter oder Hilfsvariablen des Typs C_s in C_c.

C_c und C_s dürfen hierbei nicht in einem direkten oder indirekten Vererbungsverhältnis stehen, da nur Kopplungen aufgrund von herkömmlichem Nachrichtenaustausch gemessen werden sollen.

COF ist dann wie folgt definiert (dies ist die erweiterte Definition aus Brito e Abreu et al. 1995):

$$COF = \frac{\sum_{i=1}^{TC}\left[\sum_{j=1}^{TC} is\_client(C_i, C_j)\right]}{TC^2 - TC - 2 \times \sum_{i=1}^{TC} DC(C_i)}$$

TC stellt wie bereits bekannt die Gesamtanzahl der Klassen dar. *is_client* führt den oben näher beschrieben Test aus: Getestet wird, ob C_i Klient von C_j ist (1=Ja, 0=Nein). Da jede Klasse im System potentiell Klient aller anderen Klassen (Eigenreferenzen sollen hier ausgeschlossen sein) ist, ist die Maximalanzahl der Klient/Server-Beziehungen TC^2-TC. $2 \times \sum_{i=1}^{TC} DC(C_i)$ berechnet die Anzahl der Kopplungen, die durch Vererbung entstehen; diese sollen hier nicht berücksichtigt werden.

Die COF-Metrik liefert folglich einen Wert zurück, welcher die Kopplung zwischen den Klassen beschreibt, wobei die Kopplung sowohl durch das Senden von Nachrichten als auch durch semantische Assoziationslinks zustande kommen kann (Brito e Abreu et al. 1995, S. 14 oben).

Vielfach ist angemerkt worden, dass die Kopplung zwischen Klassen möglichst minimal gehalten werden sollte – die Metrik *Violations of the Law of Demeter* hat genau dieses Prinzip zur Hauptaufgabe (s.u.). Starke Kopplung erhöht die Komplexität, verringert die Kapselung (und damit die Wiederverwertung) und ist der Verständlichkeit und Wartbarkeit abträglich. Aus diesem Grunde sollte die COF-Metrik keine hohen Werte erzielen. Auf der anderen Seite ist jedoch Kopplung nötig, um überhaupt Funktionalität erzielen zu können, was die 0 auf der anderen Seite als erstrebenswertes Ziel ebenso ausschließt. Der ideale Wert liegt folglich in der Mitte des Intervalls [0,1].

Polymorphism Factor (PF)

Wie bereits oben beschrieben setzt PF die Anzahl überschriebener Methoden der Klassen zu den insgesamt vorhandenen Methoden in den Klassen ins Verhältnis.

Mathematisch ist PF wie folgt definiert (dies ist die Definition aus Brito e Abreu et al. 1995, die gegenüber der ursprünglichen Version aus Brito e Abreu/Carapuca 1994 leicht verändert wurde):

$$PF = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}\left[M_n(C_i) \times DC(C_i)\right]}$$

TC steht wie zuvor für die Gesamtanzahl an Klassen im System. $M_o(C_i)$ bezeichnet die Anzahl überschriebener Methoden in Klasse C_i; $M_n(C_i)$ die Anzahl neuer Methoden in Klasse C_i und $DC(C_i)$ die Anzahl der Subklassen der Klasse C_i.

Insgesamt geht es bei der Berechnung von PF um Polymorphie – also um den Mechanismus, Methodenaufrufe je nach aktuellem Typ des jeweiligen Objekts korrekt zu delegieren.

Der Zähler stellt hier die „tatsächliche Anzahl möglicher verschiedener polymorpher Situationen“ dar (Brito e Abreu et al. 1995, S. 7, eigene Übersetzung). Wird also eine Nachricht an eine Klasse C_i gesendet – dies entspricht einem Methodenaufruf – so gibt es mehrere Methodenimplementationen, an welche die Nachricht (statisch oder dynamisch) gebunden werden kann – nämlich ebenso viele, wie es überschriebene Methoden gibt.

Der Nenner stellt dagegen die „maximale Anzahl möglicher verschiedener polymorpher Situationen“ der Klasse C_i dar (ebd.). Das Maximum ist erreicht, wenn sämtliche Methoden in allen Subklassen überschrieben worden sind.

Mit der PF-Metrik wird die Nutzung der Polymorphie im Gesamtsystem gemessen. Während die (meist dynamische) Bindung von Methodenaufrufen an die jeweiligen Implementationen die Komplexität verringern sollte, gilt dies nicht unbedingt für die Testbarkeit, da das Debugging bei starker Nutzung der Polymorphie schwieriger wird. Wie bereits bei anderen Metriken ist also auch hier die goldene Mitte die richtige Wahl und damit ein mittlerer Wert im Interval [0,1].

A) iii) Violations of the Law of Demeter

Das Prinzip des Law of Demeter, welches 1989 zum ersten Mal vorgestellt wurde (Lieberherr/Holland 1989), lässt sich mit folgendem Leitsatz für Klassen zusammenfassen: „Sprich nur mit deinen direkten Freunden“. Anders als bei Metriken handelt es sich beim Law of Demeter zunächst um ein Gesetz: Es werden genaue Richtlinien aufgestellt, welche Formen der Kommunikation zwischen zwei Klassen zulässig sind. Die Metrik besteht dann darin, Verletzungen dieser Regel aufzuzeigen – die *Violations of the Law of Demeter*.

Sharble/Cowen 1993 führen das Kürzel VOD für die Violations of the Law of Demeter ein und quantifizieren die Messung (vgl. Henderson-Sellers 1996, S. 117).

Das Law of Demeter dient dazu, die Qualität objektorientierten Codes durch die Minimierung von Kopplungen zu erhöhen, da diese der Wiederverwendung im Wege stehen, indem sie die Modularität einschränken. Es erlaubt den Zugriff auf direkt von einem Objekt angebotene Dienste (z.B. Aufruf von Methoden), verbietet jedoch einen Eingriff in das Objekt hinein zum Zugriff auf ein Sub-Objekt und dessen angebotenen Dienste.

Es existieren zwei Formen des Laws (vgl. Palm et al. 2003 sowie Lieberherr 2003):

1. *Klassenform*. Hierbei dürfen die Methoden M einer Klasse C nur Nachrichten senden an
    a. Klassen von Parametern von M (sowie die Klasse von this/self und die Klassen globaler Variablen),
    b. Klassen direkter Teilobjekte (berechnet oder gespeichert):
        i. Klassen von Objekten, die von aufgerufenen Methoden zurückgegeben werden sowie an Klassen der Attribute dieser Objekte
        ii. Klassen der Elemente einer Kollektion, die von aufgerufenen Methoden zurückgegeben werden
    c. Klassen von Objekten, die in M erzeugt werden

2. *Objektform*. Hierbei dürfen Methoden M einer Klasse C nur Nachrichten senden an
    a. Parameter von M (sowie this/self und globale Variablen),
    b. Direkte Teilobjekte (berechnet oder gespeichert):
        i. Objekte, die von aufgerufenen Methoden zurückgegeben werden sowie deren Attribute
        ii. Elemente einer Kollektion, die von aufgerufenen Methoden zurückgegeben werden.
    c. Objekte, die in M erzeugt werden

Der Hauptunterschied zwischen diesen beiden Formen ist die Überprüfbarkeit. Die Klassenform kann statisch geprüft werden, während dies für die Objektform nicht möglich ist.

Das Law of Demeter enthält folgende Prinzipien (Lieberherr/Holland 1989):

- *Kontrolle der Kopplung*. Die Einschränkung möglicher Kopplungen in Klassen trägt zu deren Modularisierung sowie einer erhöhten Wiederverwendbarkeit bei.
- *Information Hiding*. Das Information Hiding existiert in vielen Formen; eine davon ist das Verbergen der inneren Struktur („structure hiding“). Das Law of Demeter erzwingt diese Form: Im Allgemeinen verhindert es den direkten Zugriff einer Methode auf innere Teile eines Objekts. Eine „Programmierung der kleinen Schritte“ wird angestrebt: Der Entwickler muss die die Teil-Ganzes-Hierarchie unter Verwendung von Zwischenmethoden schrittweise durchlaufen.
- *Einschränkung des Zugriffs auf Informationen*. Neben dem Information Hiding, welches Methoden einer Klasse komplett verbergen kann, ist es z.T. auch sinnvoll, die Methoden zwar öffentlich zur Verfügung zu stellen, jedoch den Zugriff auf die Methoden zu beschränken.
- *Eingrenzung von Informationen*. Da das Law of Demeter die lokalen Informationen einer Klasse im Fokus hat, muss ein Entwickler nur die der Klasse nahe stehenden Typen bei der Programmierung berücksichtigen, was die Komplexität des Programmierens erheblich verringert.
- *Strukturelle Induktion*. Das auf der Arbeit von Frege aufbauende Prinzip der Komposition (vgl. Heijenoort 1967) wird berücksichtigt. Durch dieses sind einfachere Korrektheitsprüfungen (durch strukturelle Induktion) möglich.

Die Berechnung der *Violations of the Law of Demeter* ist nur für die Klassenform möglich und erfordert zwei Inspektionen je zu überprüfender Nachricht:

- Die Analyse des Empfängers einer Nachricht.
- Eine Flussanalyse auf vorangegangene Zuweisungen zum Empfänger der Nachricht (eine solche Zuweisung kann bei nachfolgenden Nachrichten zu Verletzungen führen).

Implementationen von Programmen, welche Verletzungen des Law of Demeter aufspüren und anzeigen können, liegen bereits vor (Palm et al. 2003). Das von Palm et al. vorgestellte System „DemeterCop“ wurde genutzt, um Verletzungen des Laws aufzuspüren, zu beheben und die Auswirkungen auf die Wartbarkeit des Programms zu untersuchen. Palm et al. stellen zwei Methoden vor, um Verletzungen des Law of Demeter zu beheben; eine von diesen scheint einen positiven Einfluss auf die Wartbarkeit zu haben, jedoch ist weitere Forschung nötig, da das System noch sehr jung ist (Palm et al. 2003, S. 4).

Die Anwendung des Law of Demeter resultiert aufgrund der Verringerung der Kopplung – so schreiben die Autoren in Palm et al. 2003, S. 4) – in einer erhöhten Wartbarkeit. Die Anwendung des Laws hat jedoch ebenfalls ihre Schattenseiten. Wie Lieberherr/Holland in ihrer Zusammenfassung anmerken, kann die strikte Anwendung in einer starken Erhöhung der Zahl von Methoden und Methodenparametern und verringerter Ausführungsgeschwindigkeit resultieren und zum Teil sogar zu verringerter Lesbarkeit des Codes führen (Lieberherr/Holland 1989, S. 19).

Aus diesem Grund sollte das Law of Demeter nicht als Dogma angesehen werden. Die Autoren geben bereits Hinweise auf mögliche Gründe, das Law absichtlich zu verletzen, u.A.:

- Sind Funktionen als „Black Box“ realisiert, z.B. in Frameworks, so kann der Entwickler der Funktion die Verantwortung für jede Änderung übernehmen. Er hat dafür zu sorgen, dass die Methoden abwärtskompatibel bleiben.
- Ist die Ausführungszeit des Programms kritisch, kann die Verletzung ebenfalls gerechtfertigt sein (Beispiel sind die Friend-Functions in C++).
- Sind wohlbekannte und stabile – d.h. sich nicht weiter verändernde – Klassen Teil des Projekts, kann auf diese unter Umgehung des Laws direkt zugegriffen werden.

B) Kriterien zur Evaluation von Metriken

B) i) Die Weyuker-Kriterien

Wie oben beschrieben wird die formale Evaluation von Metriken von vielen Forschern empfohlen (Fenton 1991, Chidamber/Kemerer 1994, Zuse 1997), um aus Metriken eine echtes, verlässliches Messinstrument bilden zu können. Die bekanntesten Kriterien für eine solche Evaluierung von Metriken sind die neun Kriterien von Weyuker – eine Liste von formalen Eigenschaften, welche Komplexität messende Softwaremetriken aufweisen sollten (Weyuker 1988).

Obwohl sich die von Weyuker aufgeführten Kriterien auf Komplexitätsmetriken beziehen, sind viele der Kriterien auch für die Messung anderer Eigenschaften von Software interessant; dies muss im Einzelfall abgewogen werden. Ebenso sind die Kriterien nicht ohne Tadel. Von mehreren Seiten wurde Kritik geübt (Chidamber/Kemerer 1994, Henderson-Sellers 1996, De Champeaux 1997); es ist jedoch noch keine abschließende Beurteilung möglich.

Die vorgelegten Kriterien beziehen sich in ihrer Ursprungsform auf die strukturierte Programmierung; in dieser Form sollen sie zunächst auch vorgestellt werden. Eine Anwendung auf die objektorientierte Programmierung findet sich im Anschluss.

In der folgenden Liste stehen P und Q für beliebige Programmteile[13] im betrachten Softwaresystem, |P| für das Ergebnis einer auf den Programmteil P angewendete Metrik.

1. Non-coarseness („Nicht-Grobkörnigkeit“)

Eine Metrik, die für alle Programmteile dasselbe Ergebnis liefert, ist nutzlos. Formal ausgedrückt, muss daher folgende Gleichung gelten: $\exists P \exists Q(|P| \neq |Q|)$.

2. Granularity („Granulation“)

Eine Metrik darf nur für eine endliche Zahl von Programmteilen dasselbe Ergebnis liefern. Formal gilt: Sei c eine nichtnegative Zahl. Dann gibt es nur endlich viele Programmteile, denen c als Ergebnis der gewählten Metrik zugeordnet wird. (Dieses Kriterium verstärkt Kriterium 1 darin, dass eine Metrik die vorhandenen Programmteile des Systems nicht nur in „ein paar wenige“ Metrikwerte einteilen soll).

[13] Weyuker definiert eine eigene Programmiersprache zur Erläuterung der Metriken. Sie merkt jedoch an, dass „die meisten Ideen der Arbeit nicht wirklich von speziellen Details einer Programmiersprache abhängen“ (Weyuker 1988, eigene Übersetzung). Daher soll hier allgemein von Programmteilen gesprochen werden; dies können z.B. Funktionen der Sprache C sein.

3. Non-uniqueness ("Nicht-Einmaligkeit")

Eine Metrik darf keine zu hohe Auflösung besitzen – zwei Programmteile müssen dasselbe Ergebnis einer Metrik erhalten können. Formal ausgedrückt gibt es unterschiedliche Programmteile P und Q, sodass $|P| = |Q|$ gilt.

4. Design details are important („Wichtigkeit der Details des Designs")

Bieten zwei Programmteile dieselbe Funktionalität, sind jedoch unterschiedlich implementiert, so kann die Metrik für beide Klassen unterschiedliche Werte liefern. Formal ausgedrückt gilt: $\exists P \exists Q(P \equiv Q \,\&\, |P| \neq |Q|)$. Der Operator „≡" stellt hierbei dasselbe Verhalten dar. Dieses Kriterium ist sehr komplexitätsspezifisch und ggf. für andere Metrikarten nicht anwendbar.

5. Monotonicity („Monotonität")

Werden zwei Programmteile verschmolzen, so darf die auf den resultierenden Programmteil angewendete Metrik keinen geringeren (Komplexitäts-)Wert erzielen als die Resultate der jeweils auf die einzelnen Programmteile angewendeten Metrik. Formal ausgedrückt muss folgende Gleichung gelten: $\forall P \forall Q(|P| \leq |P+Q| \, und \, |Q| \leq |P+Q|)$.

6. Non-equivalence of interaction ("Unterschiedlichkeit der Interaktion")

Wird ein Programmteil R mit zwei verschiedenen Programmteilen P und Q verschmolzen, so muss |P+R| nicht gleich |Q+R| sein, selbst wenn |P|=|Q| gilt, da sich der Grad der Interaktion zwischen P und R sowie Q und R unterscheiden kann. Formal gilt: $\exists P \exists Q \exists R(|P| = |Q| \wedge |P+R| \neq |Q+R|)$ (Der „+"-Operator wird als kommutativ betrachtet).

7. Non-equivalence of permutation („Nichtgleichheit der Permutation")

Die Resultate der Metrik sollten von der Anordnung der Programmteile des betrachteten Objekts abhängen. Es gibt Programmteile P und Q, sodass Q die Permutation der Anweisungsfolge aus P darstellt, jedoch $|P| \neq |Q|$ gilt.

8. Renaming („Umbenennung")

Die Metrik eines Programmteils darf ihren Wert nicht ändern, wenn sich der Name des Programmteils ändert.

9. Interaction increases complexity („Interaktion erhöht Komplexität")

Die Komplexität des Ganzen kann höher sein als die Summe seiner Teile; d.h. der Wert |P+Q| kann größer dem Wert |P|+|Q| sein. Formal: $\exists P \exists Q(|P| + |Q| < |P+Q|)$.

Da ein Großteil der heutigen Softwareentwicklung in objektorientierten Sprachen erfolgt und der Fokus dieser Arbeit auf Java liegt, ist insbesondere die Evaluation von Metriken für OO-Sprachen von Interesse. Chidamber/Kemerer haben die Weyuker-Kriterien bereits auf das objektorientierte Paradigma angepasst. Aufgrund des Paradigmenwechsels sind jedoch nicht alle Kriterien in der gleichen Weise wie ursprünglich definiert anwendbar; bei manchen ist eine Anwendbarkeit überhaupt nicht gegeben.

Grundlegend kann man die obige Definition von P und Q als Programmteile durch die Definition als Klassen ersetzen. Chidamber/Kemerer merken an, dass von Weyukers neun Kriterien die Kriterien 2 und 7 für objektorientierte Umgebungen keine direkte Relevanz haben:

- Kriterium 2 („Granularität“) wird, da endliche Anzahlen von Projekten mit jeweils endlichen Anzahlen von Klassen betrachtet werden, auf jede Metrik zutreffen, die auf Klassenebene misst.
- Kriterium 7 („Nichtgleichheit der Permutation“) fordert, dass eine Permutation des Inhalts eines zu messenden Objekts den Wert einer Metrik verändern kann. Da hier jedoch Klassen betrachtet werden, ist diese Forderung hinfällig, da die Anordnung von Komponenten innerhalb einer Klasse (Methoden, Felder, innere Klassen etc.) keinen Einfluss auf die Funktionalität hat.

De Champeaux (De Champeaux 1997) kritisiert Teile der Verwendung der Weyuker-Kriterien durch Chidamber/Kemerer. Insbesondere wird Kriterium 9 hinterfragt, welches aussagt, dass die Komplexität eines Kompositums von Klassen höher sein kann als die Komplexität der Summe der Einzelteile. De Champeaux führt Situationen an, in denen dies genau umgekehrt sein kann, da Synergieeffekte auftreten können, welche den Wert der Metriken verringern.

Tatsächlich ist dies auch genau das Kriterium, welches durch keine der von Chidamber/Kemerer vorgestellten Metriken erfüllt wird und auch von diesen in der Zusammenfassung angezweifelt wird. Roy et al. empfehlen, dieses Kriterium bei der Evaluierung von Vererbungsmetriken für objektorientierte Software zu ignorieren (Roy/Roy 2001).

Zusammengenommen ergeben sich somit folgende Kriterien, die grundlegend auf den von Weyuker angeführten Kriterien aufbauen und die Anpassungen auf die Objektorientierung von Chidamber/Kemerer enthalten. Das Kriterium 2 entfällt, da es immer erfüllt ist; die Kriterien 7 und 9 entfallen, da sie nicht anwendbar sind. Die Nummerierung ist hier angepasst worden, um fortlaufende Zahlen zu erhalten. In Klammern sind die von de Champeaux verwendeten Namen der Kriterien enthalten.

Kriterium 1: *Non-coarseness* (Artefact separation):
Hinreichend großer Wertebereich der Metrik.
Kriterium 2: *Non-uniqueness* (Incidental collusion):
Gleiche Werte sind jedoch erlaubt.
Kriterium 3: *Design details are important* (Form versus meaning):
Die Funktion ist für das Metrik-Ergebnis irrelevant.
Kriterium 4: *Monotonicity* (Monotonicity):
Die Vereinigung von Klassen erreicht einen höheren oder gleichen Wert als die jeweils einzelnen Klassen.
Kriterium 5: *Non-equivalence of interaction* (Composition instability):
Bei Vereinigung zweier Klassen identischen Metrikwertes mit einer dritten müssen die jeweiligen Kompositionen nicht denselben Metrikwert erhalten.
Kriterium 6: *Renaming* (Renaming):
Die Metrik einer Klasse darf ihren Wert nicht ändern, wenn sich der Name der Klasse ändert.

B) ii) Die MOOD-Kriterien

Brito e Abreu et al. definieren in Brito e Abreu/Carapuca 1994 neben der Suite aus sechs Metriken eine Liste von (informalen) Kriterien, welche neben der Evaluierung der eigenen Metriken auch zu vergleichenden Evaluationen anderer Metriken in der Metrik-Gemeinde dienen soll.

Die Liste von wünschenswerten Eigenschaften der Metriken enthält sieben Kriterien:

1. *Die Metrikberechnung muss formal definiert sein.*
2. *Metriken sollten Ergebnisse liefern, die zwischen unterschiedlichen Softwaresystemen vergleichbar sind. Abgesehen von Metriken, welche Größe messen, sollten Metriken daher unabhängig von der Systemgröße sein.*
3. *Metriken sollten entweder dimensionslos sein oder in einer bekannten und definierten Einheit definiert sein.*
4. *Metriken sollten früh im Lebenszyklus eines Systems berechenbar sein.*
5. *Metriken sollten auch im Kleinen anwendbar sein.*
6. *Metriken sollten einfach zu berechnen sein.*
7. *Metriken sollten sprachunabhängig sein* (Brito e Abreu et al. 1995, S.2f, eigene Übersetzung).

Diese Liste der Kriterien ist informell – ganz im Gegensatz zu den obigen, mathematisch definierten Weyuker-Kriterien. Die informelle Beurteilung ist insbesondere vor dem Hintergrund der Ziele von Metrikprogrammen wichtig: Eine mathematisch einwandfreie Metrik ist z.B. nicht nützlich, wenn sie erst nach der Implementierung eines Systems Werte liefert, die jedoch viel früher benötigt werden, oder die Berechnung derart komplex ist, dass der Aufwand sich nicht lohnt.

Die oben definierten Kriterien beleuchten also den unmittelbaren Einsatz von Metriken. Durch das Raster der Kriterien werden Metriken aussortiert, deren praktische Verwendbarkeit nicht unbedingt gesichert ist.

C) Evaluation der Metrik-Suite

Um einen Vergleich der Fähigkeiten der in dieser Arbeit vorgestellten Metriken zu anderen Metriken möglich zu machen, ist die Evaluation anhand bekannter Kriterien sinnvoll. Derartige Kriterien sind im vorigen Abschnitt besprochen worden.

Sinnvoll erscheint hier eine Evaluation nach den Kriterien von Brito e Abreu/Carapuca 1994, da die hier vorgestellten Metriken zur direkten Auswertung durch den Entwickler und weniger für die Eingliederung in ein komplexes Metrikprogramm gedacht sind. Folgende Tabelle ist das Resultat der Auswertung. Anmerkungen finden sich im Anschluss.

Tabelle A-3: Evaluationsergebnisse

Metrik/ Kriterium	PG	DA	PU	NODAS	ACD	BCD	IMI
Krit. 1	X (1)	X (1)	X (1)	X (1)	X (1)	X (1)	X (1)
Krit. 2	/ (2)	X (3)	X (3)	/ (2)	X (3)	X (3)	X (3)
Krit. 3	X (4)	X	X	X (4)	X	X	X
Krit. 4	- (5)	- (5)	- (5)	- (5)	- (5)	- (5)	- (5)
Krit. 5	- (6)	X	- (6)	- (6)	- (6)	- (6)	- (6)
Krit. 6	X	X	X	- (7	- (7)	- (7)	X
Krit. 7	X (8)	X (8)	X (8)	X (8)	X (8)	X (8)	X (8)

X = Erfüllt; - = Nicht erfüllt; / = Nicht relevant.

(1) Die hier vorgestellten Metriken sind in Kapitel 4 formal im Sinne mathematischer Formeln und Algorithmen definiert worden.
(2) Bei diesen Metriken handelt es sich um Größe messende Metriken.
(3) Diese Metriken liefern Ergebnisse im Intervall [0,1] und sind damit von der Systemgröße unabhängig.

(4) Die Metriken PG und NODAS geben jeweils klar definierte Einheiten an, einmal die Anzahl implementierter Interfaces und einmal die Anzahl paarweise verschiedener genutzter Methodenteilmengen. Die anderen Metriken sind in Prozent definiert.
(5) Die hier vorgestellten Metriken sind für die Implementationsphase gedacht.
(6) Da bei den hier vorgestellten Metriken die Verwendung von Klassen und Interfaces gemessen wird, sind sie nur auf komplette Programme anwendbar. Allerdings kann die Menge der betrachteten Klassen auf einen Teilbereich eingeschränkt werden.
(7) Die mit Kontexten arbeitenden Metriken erfordern einen hohen Rechenaufwand.
(8) Diese hier vorgestellten Metriken können auf alle Sprachen angewendet werden, welche zu Java-Interfaces äquivalente Konstrukte bieten oder diese simulieren können. Eine geeignete Überprüfungsroutine vorausgesetzt kann z.B. eine vollständig abstrakte Klasse in C++ als Interface gesehen werden.

Die obige Tabelle kann dabei helfen, die Nützlichkeit der Metriken für den eigenen Verwendungszweck zu beurteilen.

D) Weitere Informationen zu MetricPlugin und ContextAnalyzerPlugin

Die in Kapitel 5 beschriebenen Plug-Ins stehen wie in Kapitel 6 beschrieben zum Download bereit. Für eine einfache Nutzung der Plug-Ins werden lediglich die bereitgestellten JAR-Archive benötigt, welche neben den `.class`-Dateien auch die in Kapitel 5 beschriebene `plugin.xml` für IDEA beinhalten.

Für die korrekte Funktion der Plug-Ins wird eine funktionsfähige Version von IntelliJ IDEA Version 3.0 bis 3.1 vorausgesetzt. Innerhalb des IDEA-Verzeichnisses befindet sich das Verzeichnis `plugins`. Es ist ausreichend, die JAR-Dateien in dieses Verzeichnis zu kopieren und IDEA neu zu starten, um die Plug-Ins zu aktivieren.

Für eine Erweiterung des Codes sind die Sourcen erforderlich, diese stehen separat zum Download bereit. Enthalten ist die vollständige API-Dokumentation. Bis auf triviale Fälle sind alle Methoden und Felder bis zum `private`-Level vollständig dokumentiert.

Dem Paket liegt eine `build.xml` zur Verwendung mit Apache Ant bei, mit welcher eine entsprechende JAR-Datei aus den Sourcen erzeugt werden kann.