Ein Eclipse-Framework zur automatischen Bestimmung nützlicher Interfaces in Java-Programmen

Frank Fiedler

24. April 2007

Die Entkopplung von Klassen durch die Benutzung von Interfaces gilt allgemein als wichtiges Element eines guten objektorientierten Programmierstils. Untersuchungen zeigen jedoch, dass davon in der Praxis zu wenig Gebrauch gemacht wird. Deshalb werden in dieser Arbeit Werkzeuge neu und weiterentwickelt, die die Einführung neuer Interfaces unterstützen sollen. Besonderes Augenmerk liegt dabei auf der nachträglichen Einführung, die in verschiedenen Situationen wünschenswert ist. Die Schwierigkeit besteht darin, zu erkennen welche neuen Interfaces sinnvoll sind und was diese genau beinhalten sollen. Hierfür wird jeweils für einen konkreten Typ zunächst analysiert, wie er genau genutzt wird. Es wird dann ein Werkzeug angeboten, mit dem man gestützt auf diese Analyse neue Interfaces entwerfen kann. Dabei gilt es, eine Balance zwischen maximaler Entkopplung und der Anzahl der neu eingeführten Typen zu erreichen. Als Unterstützung hierfür wird eine Möglichkeit integriert, diese Faktoren automatisch abzuschätzen und ein oder mehrere Interfaces für jeden Typ vorzuschlagen. Dazu wird eine Metrik und ein kombinatorischer Algorithmus benutzt, die mit dem Eclipse Erweiterungsmechanismus integriert sind, so dass ihre Implementierungen ausgetauscht werden können. Einige Beispielimplementierungen sind bereits enthalten; deren Verhalten wird unter verschiedenen Aspekten untersucht.

Erklärung

Ich versichere, dass ich diese Masterarbeit selbständig verfasst, nur die angegebenen Quellen und Hilfsmittel verwendet und Zitate als solche kenntlich gemacht habe.

Hannover, den _____

Frank Fiedler _____

Danksagung

Ich danke meiner Familie für die aufgebrachte Geduld und Prof.Dr. Friedrich Steimann für die trotz engem Terminkalender stets gründliche und inspirierende Betreuung.

Titelbild

Das Bild *Ein unmögliches Dreieck mit einem seltsamen Dreh* stammt von dem schwedischen Künstler Oscar Reutersväld. Es wurde entnommen aus dem Buch *Unglaubliche optische Illusionen*, Tosa Verlag 2004.

Inhaltsverzeichnis

1 Einleitung

> „Program to an interface, not an implementation.“
> (Erich Gamma et al.)

Diese griffige Formulierung findet sich in dem berühmten Buch der „Gang of Four“[1] über Entwurfsmuster [14] und ist gut geeignet, das Thema dieser Arbeit einzuleiten. Interfaces ermöglichen die Trennung von Spezifikation und Implementierung und sind dadurch in der Lage, die Komponenten von Softwaresystemen voneinander zu entkoppeln.

Eine wesentliche Stärke von Komponenten besteht darin, Komplexität beherrschbar zu machen, und die Komplexität von Softwaresystemen nimmt ständig zu. Dadurch steigt auch die Bedeutung von Komponenten sowie deren wirksamer Entkopplung.

Ein weiterer Trend ist die Tatsache, dass sich ein immer größerer Teil der Arbeit eines Programmierers in Richtung Wartung von Code – im Gegensatz zur Neuentwicklung – verschiebt. Um die Wartbarkeit zu verbessern muss man die Struktur der Software verbessern, um z.B. „Sünden“ wie doppelten Code zu vermeiden. Die Suche nach eleganten Lösungen führt dann häufig zum Einsatz von Interfaces; dies kann man unter anderem an den Beispielen in dem erwähnten Buch über Entwurfsmuster sehen.

Wenn man konkrete Projekte untersucht, zeigt sich allerdings, dass die Benutzung von Interfaces hinter den Forderungen zurückbleibt. Anscheinend gibt es praktische Gründe und Hindernisse, die dem systematischen Einsatz von Interfaces entgegenstehen. Einer davon ist sicher, dass Interfaces in jedem Fall ein zusätzliches Konstrukt sind, das allein, also ohne Implementierung nicht benutzbar ist. Deshalb entsteht neue Software oftmals zunächst als ein Geflecht von Klassen. Bleibt also die nachträgliche Einführung. Diese birgt aber gewisse Probleme: Zwar wird sie in modernen Entwicklungsumgebungen durch Refactorings[2] oder Mechanismen wie Quickfixes[3] unterstützt, bei der eigentlichen Entscheidung, welche Interfaces einzuführen sind und was diese enthalten sollen, bleibt der Programmierer aber auf sich gestellt. Es gibt verschiedene Ansätze, neue Interfaces aus der tatsächlichen Typnutzung abzuleiten, dies ergibt aber manchmal zu viele neue Typen, so dass man zwar den Merksatz von Gamma et al. umgesetzt hat, aber das eigentliche Ziel, die bessere Wartbarkeit, durch die unnötig große Typhierarchie nach [32] und [34] eher erschwert wird.

[1] So werden die vier Autoren von [14] oft bezeichnet.

[2] In dieser Arbeit werden deutsche Fachbegriffe nur dann verwendet, wenn sie dem englischen Begriff direkt zugeordnet werden können. Ansonsten wäre es zu verwirrend, weil ja in der Software und deren Dokumentation die englischen Ausdrücke benutzt werden.

[3] Ein Feature der Entwicklungsumgebung Eclipse, welches es ermöglicht, z.B. fehlende Implementierungen für Methoden automatisch zu ergänzen, uvm.

1.1 Ein motivierendes Beispiel

1.1.1 Ein CMS-System mit FTP-Client

Angenommen man arbeitet bei einer Software-Firma, die ein eigenes Content Management System[4] vertreibt. Es besitzt einen eigenen FTP Client[5], der für den Dateiaustausch mit den jeweiligen Servern zuständig ist. Der Dateiaustausch ist natürlich eine sehr häufig benötigte Funktion in einem CMS, deswegen wird der FTP Client an sehr vielen Codestellen benutzt.

Hier sind drei Beispiele dafür, wie der Typ innerhalb des Programms genutzt wird:

```
public class Example1 {
boolean checkConnectionData(String host, String user,
    String passwd) throws IOException {
  FTPClient client = new FTPClient();
  client.connect(host);
  client.login(user, passwd);
  return client.isConnected();
}
}
```

```
public class Example2 {
private FTPClient client;
public InputStream getInputStream(String remotePath)
    throws IOException {
  if (client.isConnected()) {
    InputStream result = client
      .retrieveFileStream(remotePath);
    return result;
  } else {
    return null;
  }
}
}
```

```
public class Example3 {
public void makeLogDir(FTPClient connectedClient,
    String[] messages) throws IOException {
  connectedClient.makeDirectory("log");
  connectedClient.changeWorkingDirectory("log");
  OutputStream os = connectedClient
    .storeFileStream("messageLog");
```

[4] Ein System zur Verwaltung von Webinhalten.

[5] Als Inspiration für den Client diente der *FTPClient* des Apache Commons-net Projekts. Damit wurden auch die Screenshots erstellt.

```
  writeStrings(messages, os);
}
private void writeStrings(String[] messages, OutputStream os) {...}
}
```

Wie man sich leicht vorstellen kann, durchziehen ähnliche Passagen hundertfach den Code des gesamten Projekts. Man kann beobachten, dass an jeder dieser Stellen:

- Ein Objekt vom Typ *FTPClient* auftritt.
- Auf diesem eine Anzahl an Methoden aufgerufen wird.

Durch die vielen Referenzen entsteht eine starke Kopplung zwischen den Klassen, in denen sie auftauchen und der Klasse *FTPClient*. Dies bringt jedoch in letzter Zeit zunehmend Probleme mit sich:

- Der FTP Client besitzt eine ausgedehnte Typhierarchie, die vor Urzeiten in der Firma entwickelt wurde und mittlerweile als veraltet gilt.
- Die Anzahl der Methoden ist im Laufe der Jahre auf über 200 angestiegen. Dadurch starrt man beim Programmieren auf eine unübersichtliche Liste von möglichen Funktionsaufrufen; das Ausführen alltäglicher, einfacher Funktionen wie etwa das Umschalten des Transfermodes zwischen Ascii und Binary gestaltet sich schwierig. (Abb.1)
- Dadurch ist die Einarbeitung neuer Programmierer schwierig. Wenn die Programmierer nicht mit den Eigenheiten des FTP Clients vertraut sind, verursachen sie oft schwer auffindbare Fehler, indem sie ihn nicht auf die richtige Weise benutzen; dabei bräuchten sie für ihre spezielle Aufgabe oft nur einen winzigen Bruchteil seines Protokolls, z.B. nur fünf Methoden.
- Die Kunden haben in letzter Zeit wegen der Sicherheitsbedenken gegen FTP vermehrt den Wunsch geäußert, dass auch andere Protokolle wie SCP und SFTP mit dem CMS System genutzt werden können.

Aus diesen Gründen hat die Geschäftsführung beschlossen, dass die Abhängigkeit von dem FTP Client verringert werden muss, evt. soll er in Zukunft durch eine zugekaufte Komponente ersetzt werden. Wünschenswert wäre es, wenn man gar nicht wieder in eine solche Abhängigkeit von einer konkreten Implementierung geräte, damit in Zukunft flexibler auf neue Anforderungen reagiert werden kann.

Die Aufgabe an die Programmierer besteht nun darin, den Code schrittweise zu entkoppeln, dabei die Funktionsfähigkeit des Gesamtsystems aber natürlich zu erhalten. Also eine typische Aufgabe für Refactorings. Welche könnten dafür in Frage kommen?

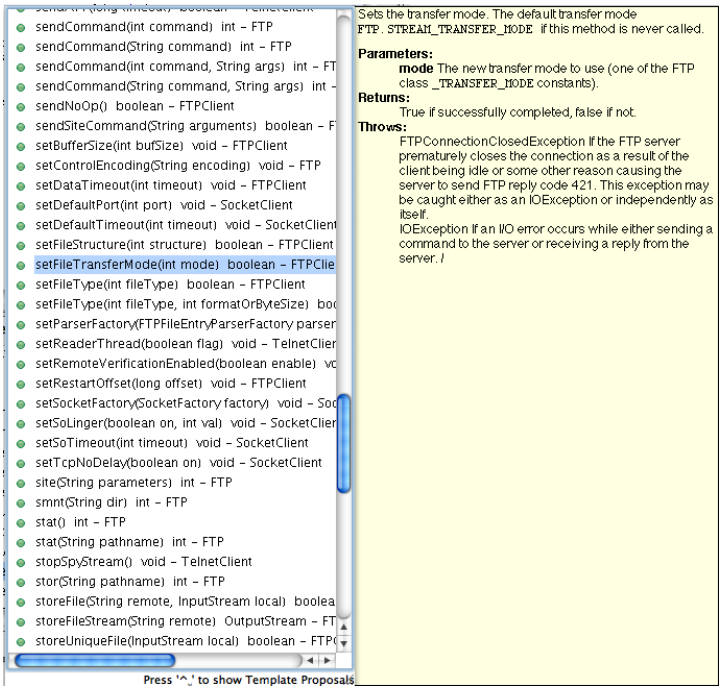


Abbildung 1: Ist dies die richtige Funktion zum Einstellen des Transfermodes?

1.1.2 Versuch 1: Generalize Declared Type

Es besteht ja bereits eine ausgedehnte Hierarchie von Supertypen der Klasse *FTPClient*. Das Refactoring *Generalize Declared Type* kann auf einer konkreten Typdeklaration aufgerufen werden, und überprüft dann, ob diese Deklaration vielleicht mit einem Supertyp des verwendeten Typs ersetzt werden könnte. Voraussetzung ist, dass der Supertyp das benötigte Protokoll zur Verfügung stellt. Die Abhängigkeit von dem Supertyp, also dem allgemeineren Typ stellt eine geringere Kopplung dar, und wäre geeignet, die Flexibilität des Codes zu erhöhen. In dem ersten Codebeispiel wird ja z.B. *nur der Login geprüft*, mit dem FTP-Protokoll hat das ja noch nicht so viel zu tun. Vielleicht ist dieser Teil des Protokolls in einem Supertyp wie *TelnetClient* bereits enthalten? Ein Aufruf auf dem ersten Beispiel ergibt den Dialog aus Abb.2. Wie man sieht, kann diese Deklaration nicht ersetzt werden. Man kann folgende Probleme festhalten:

- Man erfährt nicht, *warum* keiner der Supertypen in Frage kommt.
- Die Supertypen sind spontan erst mal nichtssagend bis verwirrend, es sieht nach einem langwierigen Unterfangen aus, sich in die Typhierarchie einzuarbeiten.
- Auch wenn ein passender Typ gefunden worden wäre, würde man immer noch von diesem abhängen. Das würde aber die Migration zu einer komplett neuen FTP-Komponente erschweren oder verhindern.

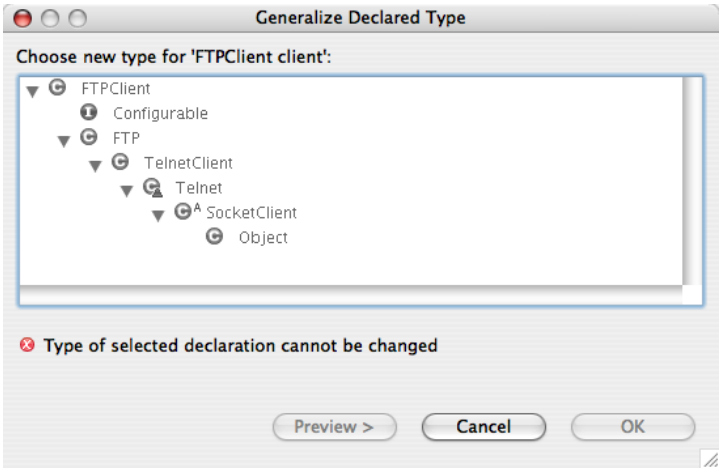


Abbildung 2: Der Dialog des *Generalize Declared Type* Refactorings in Eclipse

- Generell bricht das Refactoring ab, wenn eine Variable einer anderen vom gleichen Typ zugewiesen wird.
- In jedem Fall wird nur dieses eine Codestelle geändert, man müsste also das Refactoring für jede Stelle einzeln aufrufen.

Abhilfe zumindest für den letzten Punkt verspricht ein anderes, global arbeitendes Refactoring, das als nächstes versucht wird.

1.1.3 Versuch 2: Use Super Type Where Possible

Der Unterschied bei diesem Refactoring besteht darin, dass man es auf einem *Typ* aufruft, also in dem Bespiel etwa auf der Klasse *FTPClient*. Es erscheint dann ein Dialog wie in Abb.3. Je allgemeiner der Typ ist, den man hier auswählt, desto geringer ist das Protokoll, was er anbietet, und desto geringer auch die Wahrscheinlichkeit, dass man viele Deklarationen ersetzen kann. In diesem Fall wird, auch wenn man den unmittelbaren Supertyp *FTP* anwählt, keine einzige Deklaration gefunden, die mit diesem umdeklariert werden könnte. Das liegt natürlich u.a. daran, dass nur die drei aufgeführten Codebeispiele vorhanden sind, bei einem echten Projekt mit hunderten von Referenzen wäre die Wahrscheinlichkeit, dass etwas gefunden wird höher. Jedoch gibt es auch bei diesem Verfahren offensichtlich Probleme:

- Für die Stellen, die geändert werden sollen, wird oft kein passender Supertyp gefunden.
- Wir erhalten keinen Hinweis, welche Methoden in dem Supertyp noch *fehlen*, damit er an mehr Stellen passen würde, sonst könnten wir den Supertyp vielleicht modifizieren oder einen neuen, besser passenden einführen.

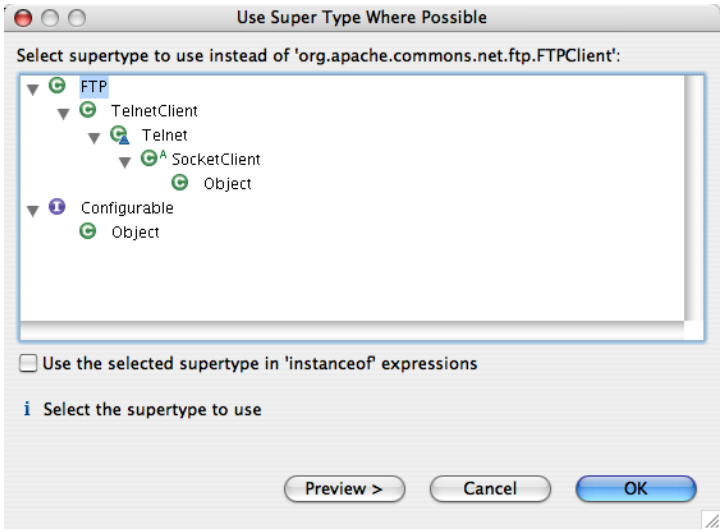


Abbildung 3: Der Dialog des *Use Super Type Where Possible* Refactorings.

Der zweite Punkt legt bereits nahe, dass die Lösung vielleicht eher darin besteht, selbst neue Typen einzuführen. Idealerweise sollten diese nur die wirklich benötigten Methoden enthalten. Mit dem in [32] beschriebenen Infer Type gibt es ein Werkzeug, was genau dies leistet.

1.1.4 Versuch 3: Infer Type

Infer Type wird wiederum, ähnlich wie *Generalize Declared Type* aus Versuch 1, auf einer Deklaration aufgerufen, nicht auf einem Typ. Es analysiert dann, welche Teile des Prokolls auf diesem Element benutzt werden, dabei werden auch Implikationen wie die Zuweisung an eine andere Variable u.a. berücksichtigt. Dann wird ein genau passender Typ berechnet, und, sofern er nicht bereits existiert, angeboten ihn einzuführen. Es ist also verwandt mit *Generalize Declared Type*, bietet aber zusätzlich die Möglichkeit, genau passende Typen zu erzeugen. Dadurch kommt man dem Ziel bereits ein gutes Stück näher, denn die so erzeugten Typen stellen wegen ihres minimalen Protokolls eine gute Entkopplung dar. Als Nebeneffekt wird die Programmierung wie gefordert vereinfacht und sicherer gemacht, weil die neuen Typen nur über einen kleinen, übersichtlichen Satz an Methoden verfügen. Dabei ist Infer Type im Gegensatz zu *Generalize Declared Type* auch in der Lage, Zuweisungen zu berücksichtigen. Da Infer Type nicht auf existierende Supertypen beschränkt ist, sondern neue Interfaces einführen kann, ist man auf diese Weise auch dem Ziel der kompletten Unabhängigkeit von einer konkreten FTP-Implementierung nähergekommen: Eine neue FTP-Komponente müsste nur die entsprechenden Interfaces implementieren, um einsetzbar zu sein, was sich z.B. durch die Programmierung entsprechender Adapterklas-

sen[6] lösen liesse. Aber auch die Verwendung von INFER TYPE löst nicht alle Probleme:

- INFER TYPE muss ebenfalls auf jeder zu ändernden Codestelle aufgerufen werden.
- Da das verwendete Protokoll sich sehr oft mehr oder weniger unterscheidet, schlägt INFER TYPE für verschiedene Codestellen immer wieder neue Interfaces vor. Es sollen aber nicht so viele neue Typen eingeführt werden.

Während die Einführung neuer Interfaces steht also außer Frage steht, soll ihre Anzahl nicht höher sein als sinnvoll ist. Außerdem sollen am besten alle Codestellen auf einmal mit ihnen umdeklariert werden. Deswegen wird ein weiterer Versuch unternommen.

1.1.5 Versuch 4: Extract Interface & Replace Occurences...

Das Refactoring *Extract Interface* tut eigentlich genau das gewünschte, vor allem, wenn man wie in der Abb.4 zu sehen gleich das Häkchen bei „Use the extracted interface type where possible“ setzt. Allerdings zeigt der Screenshot

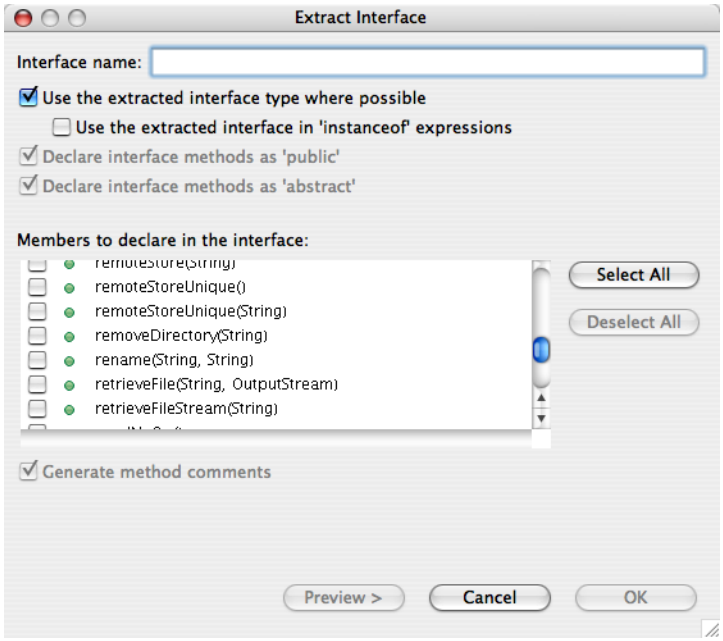


Abbildung 4: Der Dialog des *Extract Interface* Refactorings. Man beachte den kleinen Scrollbalken: Hier ist eine Wahl aus *sehr vielen* Methoden zu treffen.

auch, dass hier wieder das Problem des ausufernden Protokolls auftritt: Man hat keine Hilfe dabei, aus den vielen Methoden die relevanten herauszufinden.

[6] Siehe *Adapter Pattern* in [14].

Wenn man zu wenige markiert, werden viele Codestellen nicht umdeklariert; markiert man zu viele, wird die Programmierung der Adapterklassen unnötig erschwert, und die Ziele Entkopplung, Vereinfachung und verbesserte Sicherheit in Frage gestellt.

Man vermisst also bei diesem Refactoring schmerzlich die Information über die tatsächliche Typnutzung, wie sie INFER TYPEfür die Deklaration, auf der es aufgerufen wurde bietet. Grundsätzlich wäre man auch gar nicht abgeneigt, mehrere Interfaces einzuführen, z.B. könnte man sich eines vorstellen, das nur Lesezugriff auf den Server bietet. Damit könnte man die Sicherheit an Codestellen, die nur lesen sollen, erhöhen. Aber es müsste so beschaffen sein, dass es genug Codestellen gibt, an denen man es einsetzen kann. Gesucht ist also eine Art „gemeinsamer Nenner" bei der Typnutzung. Um diesem Ziel näherzukommen, wäre ein Überblick über die verschiedenen Arten, wie ein Typ im Projekt genutzt wird, sicher sehr hilfreich.

1.1.6 Die IntoJ Suite

Es gibt bereits ein Werkzeug, was diesen Überblick ermöglicht, und zwar den TYPE ACCESS ANALYZER innerhalb der INTOJ SUITE[7]. Er analysiert die Nutzung eines Typs, und stellt das Resultat auf verschiedene Weise dar. Allerdings bieten diese Ansichten keine Möglichkeit, mehrere Nutzungsweisen so wie gewünscht zusammenzufassen und daraus Interfaces abzuleiten und einzuführen.

Man kann die Idee sogar noch einen Schritt weitertreiben: Wenn die Daten über die verschiedenen Nutzungsweisen des Typs bereits vorliegen, kann man aus ihnen vielleicht auch gleich *automatisch ableiten*, welche davon am besten zu einem neuen Interface zusammengefasst werden sollen. Also angenommen, es sind viele Codestellen da, die *nur die Lese-Methoden* des Protokolls benutzen. Dann müsste es doch möglich sein, daraus abzuleiten, dass ein Nur-Lese-Interface offensichtlich sinnvoll wäre, und es sozusagen „per Knopfdruck" zur Einführung vorzuschlagen!

1.2 Beitrag dieser Arbeit

Diese Arbeit beschäftigt sich mit einer Implementierung des im vorigen Abschnitt geforderten Werkzeugs. Dazu wurden Teile des TYPE ACCESS ANALYZER benutzt, verändert und mit neu geschriebenen Komponenten kombiniert. Insgesamt ist so ein Framework zum Design neuer Interfaces entstanden, der *Interface-Designer*. Der Zweck besteht darin, den Entwurf neuer Interfaces auf der Basis einer Analyse der tatsächlichen Typnutzung zu unterstützen. Dabei wird dem Entwickler eine Möglichkeit geboten, neue Interfaces zu definieren, indem er sich u.a. auf eine grafische Darstellung der Typnutzung stützt und darin mehrere Nutzungsweisen des Typs zusammenfassen

[7] Für einen Überblick über die INTOJ SUITE siehe Anhang B und [18], [19].

kann um dadurch das Protokoll neuer Interfaces zu definieren. Darüberhinaus werden Erweiterungspunkte angeboten, um Algorithmen bereitzustellen die diesen Vorgang automatisieren. Einige Beispielimplementierungen werden bereitgestellt. Um eigene Erweiterungen leicht testen zu können, gibt es ferner die Möglichkeit, einen automatisierten Entwurf auf ganze Projekte anzuwenden und die Ergebnisse zu analysieren, zu speichern und zu exportieren.

1.3 Aufbau der Arbeit

In Kapitel 2 werden die grundlegenden Begriffe erklärt. Dann wird in Kapitel 3 die Implementierung im Detail beschrieben. Insbesondere wird die Benutzung der Erweiterungspunkte erklärt sowie die Beispielimplementierungen vorgestellt. In Kapitel 4 wird die Implementierung zunächst auf ein einfaches und ein etwas komplexeres Beispiel angewendet. Dann wird die projektweite Analyse an vier Beispielprojekten durchgeführt. Zum Schluss wird anhand einiger Stichproben die Plausibilität der Vorschlagsautomatik erprobt. In Kapitel 5 werden Kritikpunkte an dem benutzen Ansatz besprochen, und die Ergebnisse der Fallbeispiele diskutiert. Eine kurze Zusammenfassung und ein Ausblick auf mögliche Weiterentwicklungen schließen die Arbeit ab.

2 Grundlagen und Begriffe

Die gewählte Thematik hat die Eigenschaft, relativ klar und einfach zu erscheinen, solange man sich auf einer höheren Abstraktionsebene bewegt. Begibt man sich allerdings hinab in die Welt real existierender Programmiersprachen und Programme, hat man schnell mit eine Reihe von Sonderfällen und Ausnahmen zu tun. Der Teufel steckt wie so häufig im Detail. Deshalb muss man schon bei der Begriffsklärung immer sehr genau unterscheiden, in welchem Kontext man sich gerade bewegt:

- Theoretisch-wissenschaftlicher Kontext: In wissenschaftlichen Publikationen. Fachausdrücke wie „Interface“ werden hier gemäss ihrer wissenschaftlichen Definition benutzt. „Wissenschaftlich“ bedeutet: Sie ist für den Leser nachvollziehbar, weil sie entweder direkt im Text steht oder darauf verwiesen wird, wo man sie nachlesen kann.
- In der Java-Welt: Die Bedeutung der Ausdrücke richtet sich nach der *Java Language Specification*[40].
- In einem Programm: Innerhalb einer Software und deren Dokumentation werden oft Ausdrücke in einer Weise benutzt, die nur dort Gültigkeit hat. Manchmal wird das an irgendeiner Stelle erklärt, oft ist so eine Erklärung aber schwer zu finden oder fehlt völlig.

In dieser Arbeit wird bei Unklarheiten immer auf den Kontext hingewiesen. Allerdings ist bei der Lektüre der externen Quellen in dieser Hinsicht erhöhte Aufmerksamkeit angebracht, die genannten Unterschiede können sonst für viele Missverständnisse und Ungereimtheiten sorgen.

2.1 Entkopplung durch Interfaces

Unter Kopplung versteht man im Zusammenhang mit der objektorientierten Programmierung den Grad der Abhängigkeit zwischen Klassen. Eine starke Kopplung ist dabei nach [34] in vielen Fällen ganz natürlich, wenn Klassen eng zusammenarbeiten um ein gemeinsames Ziel zu erreichen. Sie ist jedoch unerwünscht, wenn sie Modulgrenzen überschreitet, d.h. wenn die beiden Klassen, zwischen denen die Kopplung besteht, verschiedenen Modulen zuzuordnen sind. Parnas beschreibt in [26] ein Modul als einen Teil einer Software, der Verantwortung für eine bestimmte Teilaufgabe übernimmt("responsibility assignment") und nennt Kriterien nach denen eine solche Aufteilung vorzunehmen ist. In dem einleitenden Beispiel kann man etwa den FTP-Client als Modul indentifizieren, dessen Verantwortungsbereich der Dateitransfer von und zum Server ist.

Allerdings gibt es in Java kein spezielles Sprachkonstrukt für Module. Auf die Funktionalität wird dagegen immer über Typen zugegriffen, also indem eine Klasse mit einem bestimmten Typ deklarierte Objekte besitzt und auf ihnen Methoden aufruft. Wenn dieser Typ also zu einem anderen Verantwortungsbereich (bzw. Modul) gehört, wäre es wünschenswert, die entstehende Kopplung möglichst gering zu halten. Dies kann in Java erreicht werden, indem man in den Deklarationen statt Klassen Interfaces benutzt. Zur Verdeutlichung kann man sich das an einem Beispiel klarmachen: Wir haben eine Klasse *Book*, die von der Klasse *Shop* benutzt wird:

```
public class Book {
  private String title;
  private int price;
  public String getTitle(){
    return title;
  }
  public int getPrice(){
    return price;
  }
}


public class Shop {
  public void printPrice(Book book){
    System.out.println(book.getPrice());
  }
}
```

Durch den Typ des Parameters der Methode *printPrice(..)* entsteht eine Kopplung der Klasse *Shop* an die Klasse *Book*. Die Klasse *Book* muss für die Klasse *Shop* auch auffindbar sein, sich also im Classpath befinden, und sofern sie nicht im selben Package liegt, über eine *import* Anweisung bekannt gemacht worden sein.

Man kann nun diese Kopplung verringern, indem man ein Interface *ShopItem* einführt:

```
public interface ShopItem {
  int getPrice();
}
```

Es enthält genau den Teil des Protokolls von *Book*, der in der Methode *printPrice(..)* benötigt wird, also nur die Methode *getPrice(..)*. Um das Interface zu benutzen sind zwei weitere Schritte nötig:

1. Die Klasse *Book* implementiert das Interface.
2. Der Typ des Methodenparameters in *printPrice(..)* wird mit *ShopItem* ersetzt.

Die Klasse *Book* sieht dann so aus:

```
public class Book implements ShopItem {
  private String title;
  private int price;
  public String getTitle(){
    return title;
  }
  public int getPrice(){
    return price;
  }
}
```

Und die Klasse *Shop* so:

```
public class Shop {
  public void printPrice(ShopItem item){
    System.out.println(item.getPrice());
  }
}
```

Die vergrößerte Flexibilität besteht darin, dass auch andere Klassen das Interface *ShopItem* implementieren können, z.B. eine Klasse *MovieDVD*:

```
public class MovieDVD implements ShopItem{
  private int price;
  private String director;
  public String getDirector(){
    return director;
  }
  public int getPrice(){
    return price;
  }
}
```

Objekte dieser Klasse können dann ebenso gut von der Methode *printPrice(..)* verarbeitet werden, was den Nutzen der Methode erhöht. Sonst hätte man womöglich erwogen, mehrere Methoden wie *printBookPrice(..)* und *printDVDPrice(..)* einzuführen. Die Lösung mit einer einzigen Methode ist wartungsfreundlicher, weil leichter neue Artikel hinzugefügt werden können. Außerdem können alle Eigenschaften, die für *alle* Shop-Items wichtig sind, wie etwa Verfügbarkeit, Artikelnr. usw nun gemeinsam im Interface *ShopItem* gepflegt werden, was ebenfalls die Wartung vereinfacht.

Zusammenfassend kann man festhalten, dass die Abhängigkeit von der Klasse *Book* durch die Abhängigkeit von dem Interface *ShopItem* ersetzt wurde (siehe Abb.5). Die Klassen *Book*, bzw. *MovieDVD* müssen der Klas-

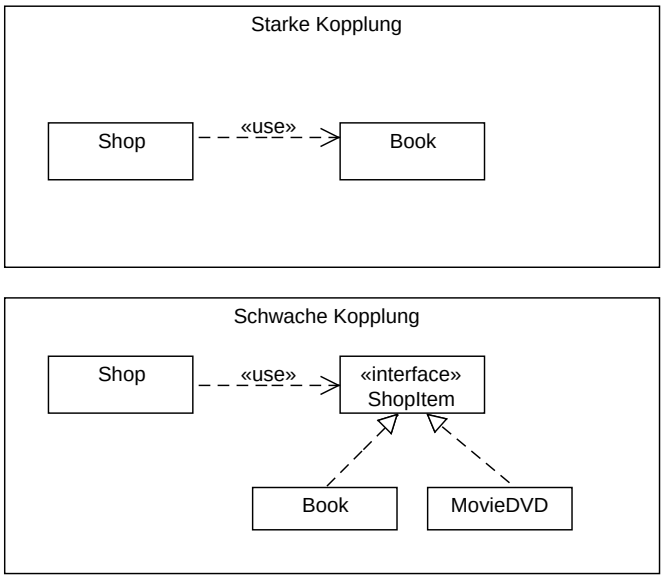


Abbildung 5: Die Entkopplung mit dem Interface *ShopItem*.

se *Shop* nicht mehr bekannt sein. In diesem Fall hätte man einen ähnlichen Effekt auch durch die Einführung einer gemeinsamen Superklasse von *Book* und *MovieDVD* und das Verschieben der Methode *getPrice()* in diese Superklasse erreichen können. Auch dies hätte eine Verringerung der Kopplung bewirkt, denn diese manifestiert sich nach [28] zusätzlich in der *Größe* des zur Verfügung stehenden Protokolls:

> „Das grundsätzliche Ziel [bei der Abschwächung der Kopplung] besteht darin, dass die Instanzen einer jeden Klasse so wenig wie möglich mit Instanzen anderer Klassen zusammenarbeiten, d.h. Verbindungen besitzen bzw. eingehen oder Nachrichten senden. Unterstützt wird diese Maxime durch schmale Schnittstellen, d.h. die Anzahl der öffentlichen (Attribute und) Operationen einer Klas-

> se sollte so klein sein, dass gerade noch die Dienstnutzer befriedigt werden und so viele Interna wie möglich verborgen bleiben."

Die Einführung einer Superklasse ist jedoch fragwürdig, weil gar nicht gesagt ist, ob hier irgend etwas vererbt werden soll, Ziel ist ja nur die Substituierbarkeit. Ein weiterer Nachteil ist: Alle Klassen, die dann mit der *printPrice(..)* Methode benutzbar sein sollen, müssten dann von der Superklasse erben. Da Java keine Mehrfachvererbung kennt, können sie dann nicht mehr von einer anderen Klasse erben. Die stärkste Flexibilisierung wird also mit Interfaces erreicht, insbesondere wenn diese nur die benötigten Methoden enthalten.

2.2 Begriffe und Konzepte im Zusammenhang mit Entkopplung

2.2.1 Server, Client, Protokoll

Diese Ausdrücke assoziiert man zunächst einmal mit Netzwerken und verteilten Systemen. Im Zusammenhang mit der objektorientierten Programmierung haben sie jedoch andere Bedeutungen.

Server & Client Ein *Client* ist eine Klasse, die die Dienste einer anderen Klasse benutzt, indem sie z.B. deren Methoden aufruft. Die Klasse, die die Dienste zur Verfügung stellt, ist in diesem Sinne der *Server*.

Protokoll Das Protokoll bezeichnet die Teile einer Klasse, die durch den Client benutzbar sind. Man beachte, dass das Protokoll in Java zwar nicht von der Identität eines Clients abhängt, wohl aber davon, wo sich der Client befindet(in welcher Package), und in welchem Verhältnis er zur anbietenden Klasse steht. Ist der Client vom selben Typ wie der Server, gehören auch *private* Methoden zum Protokoll. Befindet er sich im selben Package, gehören alle *default*[8] Elemente dazu, bei Vererbung *protected* Elemente, usw. Im Folgenden wird von einem Protokollbegriff in Sinne von mit *public* deklarierten, nicht-statischen Methoden ausgegangen. Dadurch wird implizit auf die Entkopplung auf der Package-lokalen Ebene verzichtet, denn die entsprechenden Methoden werden ja von dem Protokollbegriff nicht mehr erfasst. Da sich die Entkopplung aber wie in Kapitel 2.1 erwähnt auf die „Schnittstellen der größeren Programmeinheiten" konzentrieren soll, ist diese Beschränkung durchaus erwünscht.

Etwas anders ist die Situation bei den mit *protected* markierten Teilen, denn es ist durchaus üblich, dass etwa ein Framework benutzt wird, indem man eigene Klassen von dessen Klassen erben lässt, wie z.B. bei *Swing*. Die

[8] In Java haben alle Felder und Methoden, die ohne Modifier notiert werden, eine Package-weite Sichtbarkeit. Da kein Schlüsselwort dafür vorgesehen ist, nennt man diese Reichweite für gewöhnlich *default*.

protected Methoden werden dabei über Modulgrenzen hinweg benutzt. Hier gilt jedoch, dass, wie in [32] angemerkt, durch die Vererbung bereits eine stärkere Kopplung etabliert ist als durch den mit *protected* gekennzeichneten Teil des Protokolls, so dass auch hier eine Entkopplung nicht vielversprechend wäre.

Bleibt noch die Frage, ob man in Java auch *Felder*, die mit *public* markiert sind, zum Protokoll rechnet. Sie werden ohne Zweifel in vielen Programmen in dieser Form benutzt. Ein Problem ergibt sich jedoch, wenn man versucht, sie mittels eines Interfaces zu entkoppeln, denn Interfaces besitzen nur Methoden, keine Felder. Der Versuch, das Interface durch eine gemeinsame, möglicherweise abstrakte Superklasse zu ersetzen, schränkt wiederum die Flexibilität ein. Eine elegantere Lösung ist also, die Felder *private* zu machen, und den Zugriff nur über Accessoren, auch bekannt als Getter- und Setter-Methoden, zu gestatten. In modernen Entwicklungsumgebungen wird dazu meist ein automatisches Refactoring angeboten.

2.2.2 Deklarationselemente

In Kapitel 2.1 wurde gezeigt, wie sich die Kopplung in Form eines Methodenparameters manifestiert. Dies passiert an allen Stellen, an denen etwas mit einem Typ deklariert ist, also:

- Felddeklarationen
- temporäre Variablendeklarationen
- Methodenparameter
- Rückgabewerte von Methoden

Im Folgenden werden diese Konstrukte [41] folgend unter dem Begriff *Deklarationselemente* zusammengefasst. Nicht darunter fällt das Erscheinen einer Typbezeichnung in *instanceof* oder Cast-Operatoren. Die Gründe sind ähnlich wie bei den Erwägungen zum Protokollbegriff: Wie in [32] ausgeführt, wird davon ausgegangen, dass die Benutzung eines Typs in diesen Operatoren eine explizite Absicht des Programmierers ausdrückt, dass genau dieser Typ dort gewünscht ist, und deshalb nicht durch einen anderen ersetzt werden soll.

2.2.3 Subprotokolle, Subprotokoll Graph, Access Sets

Von dem Protokoll – also der Menge der öffentlichen Methoden – kann man Untermengen bilden, die im Folgenden *Subprotokolle* genannt werden. Das wird anhand einer kleinen Beispielklasse anschaulich gemacht:

```
public class DemoA {
    public void m1(){};
    public void m2(){};
```

```
    public void m3(){};
    public void m4(){};
}
```

Die Klasse hat offensichtlich ein Protokoll, das aus vier Methoden besteht. Die Menge aller Untermengen, die Potenzmenge, besitzt durch die Teilmengeneigenschaft eine Halbordnung und bildet nach [33] einen Verband. Dieser lässt sich als Hasse-Diagramm darstellen. Der Beispieltyp *DemoA* ergibt das Diagram in Abb.6. Diese Form der Darstellung wird im Folgenden als *Subpro-*

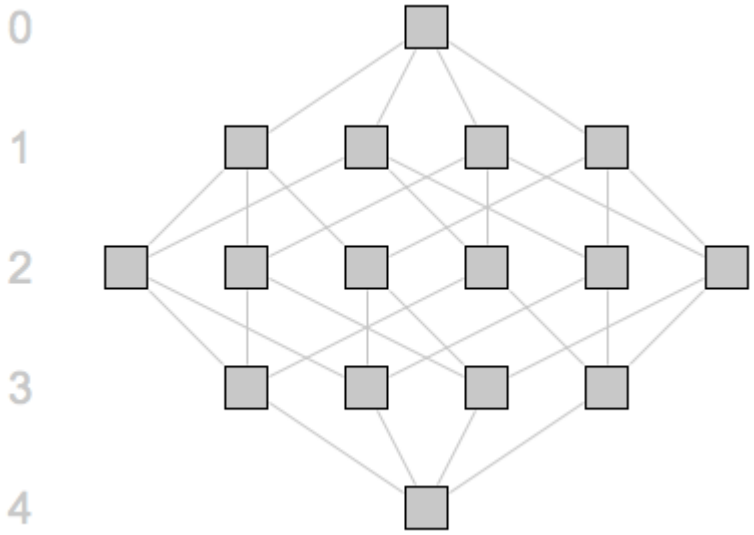


Abbildung 6: Ein einfacher Graph für eine Klasse mit 4 Methoden

tokollgraph bezeichnet, mitunter, der Bezeichnung in der Implementierung folgend, auch als *Subprotocol Lattice*[9].

Jeder Knoten in diesem Graph steht für ein Subprotokoll. Die untere Spitze bildet das Protokoll selbst mit seinen vier Methoden. In der Reihe darüber finden sich alle Subprotokolle mit drei Methoden, in der nächsthöheren Reihe alle mit zwei, usw. Die Zahlen am linken Rand geben also jeweils die Anzahl der Methoden für die Subprotokolle in dieser Reihe an. Der höchste Knoten steht dementsprechend für das leere Protokoll. Man beachte, dass das Protokoll des Beispiels eigentlich größer ist, denn in Java erben bekanntlich alle Objekte vom Typ *Object*, zu ihrem Protokoll gehören also auch die zwölf von *Object* geerbten Methoden. Diese Methoden werden im Graph aber weggelassen. Jede Kante zwischen zwei Knoten steht für eine Teilmengenbeziehung:

[9] Innerhalb der IntoJ Suite wird diese Darstellungsform auch als *Access Set Lattice Graphical View* bezeichnet. Das ist insofern abweichend, als Access Sets in der hier gegebenen Definition *benutzte* Subprotokolle sind, der *Access Set Lattice Graphical View* enthält aber eben auch unbenutzte Subprotokolle.

Das Subprotokoll der unteren Menge schließt das der oberen mit ein. Da diese Beziehung transitiv ist, werden die Beziehungen über mehrere Ebenen nicht gezeigt. Man beachte insbesondere, dass man für jede Menge an Subprotokollen einen genau definierten Knoten findet, der ihre Vereinigungsmenge repräsentiert, gleiches gilt für ihre Schnittmenge.

Man kann sich nun vorstellen, dass jeder dieser Knoten einen potentiellen Typ repräsentiert; man könnte ja einen Typ einführen, der das entsprechende Subprotokoll besitzt. Jeder Knoten, den man dann abwärts entlang der Kanten findet, wäre ein möglicher Subtyp dieses Typs, denn er besitzt mindestens die Methoden des Supertyps und könnte diesen somit ersetzen.

Betrachtet man die Menge aller Deklarationselemente, die mit dem Basistyp deklariert sind, ergibt sich mit großer Wahrscheinlichkeit, dass nicht alle von ihnen alle vier Methoden, also das ganze Protokoll benutzen. Tatsächlich benutzen in der Praxis die meisten Deklarationselemente nur einen kleinen Teil der Methoden, die der deklarierte Typ bereitstellt. Da in dem Graph alle Subprotolle vorhanden sind, kann man also jedes Deklarationselement genau einem Knoten zuordnen. Das Subprotokoll eines Deklarationselementwird im Folgenden als *Access Set* bezeichnet. Ein *Access Set* ist also ein Subprotokoll das *benutzt wird.*[33]

2.3 Kopplungsmetriken

Eine Softwaremetrik stellt nach [42] ein Maß für eine bestimmte Eigenschaft von Software dar. Für das Thema dieser Arbeit von Bedeutung sind einige Metriken, die sich mit den Themen „Kopplung“ und „Interfaces“ beschäftigen.

2.3.1 Actual Context Distance

Wie in Kapitel 2.1 bereits angemerkt wurde, ist ein Merkmal von Kopplung die Anzahl der bereitgestellten und der konkret benötigten Methoden. Damit beschäftigt sich die in [31] vorgestellte Metrik *Actual Context Distance*, die dort folgendermaßen eingeführt wird:

Jedes statisch typisierte, objektorientierte Programm besitzt eine Menge an Typen, T. Jeder Typ $A \in T$ besitzt eine Menge von Attributen und Methoden, $\mu(A)$, deren Untermenge

$$\pi(A) := \{m \in \mu(A) \mid m \; ist \; eine \; mit \; public \; deklarierte, \; nicht - statische \; Methode\}$$

als sein *Protokoll* bezeichnet wird. Die Metrik bezieht sich immer auf ein Deklarationselement, A a. Die Größe des von A bereitgestellten Protokolls ist folglich $|\pi(A)|$. Zusätzlich bezeichnet $\iota(a)$ das Protokoll, das von a *tatsächlich benötigt* wird, und $|\iota(a)|$ entsprechend dessen Größe. Die *Actual Context Distance* ergibt sich dann als Funktion des Deklarationselementes a und des

Typs A, mit dem es deklariert ist:

$$ACD(a, A) := \frac{|\pi(A)| - |\iota(a)|}{|\pi(A)|}$$

Wenn alle bereitgestellten Methoden auch benötigt werden, ist der ACD-Wert 0. Wird dagegen keine einzige der bereitgestellten Methoden benötigt, ist der Wert 1. Wird nur die Hälfte benötigt, ist der Wert 0,5, usw. In [32] wird dann noch der ACD-Wert eines *Typs* eingeführt, indem der durchschnittliche ACD-Wert aller Deklarationselemente des Typs gebildet wird. Im Sinne einer Entkopplung ist also ein *kleiner* ACD-Wert ein *guter* Wert, denn er bringt zum Ausdruck, dass man sich der in Kapitel 2.1 geforderten „schmalen Schnittstelle“ annähert. In dem oben angeführten Codebeispiel wurde der ACD-Wert durch die Einführung des Interfaces *ShopItem* verbessert. Auf dem Deklarationselement *book* wurde vorher eine der zwei Methoden, die die Klasse *Book* bereitstellt genutzt. Das ergibt einen ACD-Wert von 0,5. Nachdem das Deklarationselement mit dem Interface *ShopItem* umdeklariert wurde, wurde das komplette bereitgestellte Protokoll benutzt, weil das Interface nur diese eine Methode anbietet. Der ACD-Wert sank dadurch auf den optimalen Wert 0.

2.3.2 Popularität von Interfaces

In [29] wird die Popularität, bzw. „popularity“ eines Interfaces eingeführt. Sie ist als „die Anzahl der Variablen, die dieses Interface implementieren“ definiert.

2.3.3 Verhältnis der Metriken, Decoupling/Growth Ratio

Wie man gesehen hat, kann der ACD-Wert verbessert werden, wenn man neue Interfaces einführt. Wie in [9] erläutert, geht eine starke Verbesserung dabei normalerweise einher mit einer sinkenden Popularität der eingeführten Interfaces: Je spezieller ein Interface ist, an desto weniger Stellen kann man es für gewöhnlich verwenden. In [9] wird untersucht, was die Einführung maximal kontextspezifischer Interfaces für ein ganzes Projekt bewirken würde, und als Ergebnis konstatiert, dass dadurch, wie schon vermutet, sehr viele Interfaces mit geringer Popularität eingeführt werden, so dass die Anzahl der Typen in einem Programm extrem anwächst.

Interessant ist jedoch die Beobachtung, dass ein kleiner Teil der Interfaces sehr populär ist und dadurch zum Umdeklarieren eines großen Teils der Deklarationselemente benutzt werden könnte. Der Rest der neuen Typen ist unpopulär, findet also nur selten Verwendung. In Zahlen ausgedrückt, sind in [9] 20% der neuen Typen bereits in der Lage, 80% der Deklarationselemente abzudecken, in [32] wird erwähnt, dass für die beiden Beispielprojekte DrawSWF und JHotDraw bereits 12% bzw. 8% der neuen Typen jeweils

50% der Deklarationselemente abdecken. Eine genaue Ansicht über die Verteilung findet sich in Abb.7. Diese Beobachtung führte zur Entwicklung der

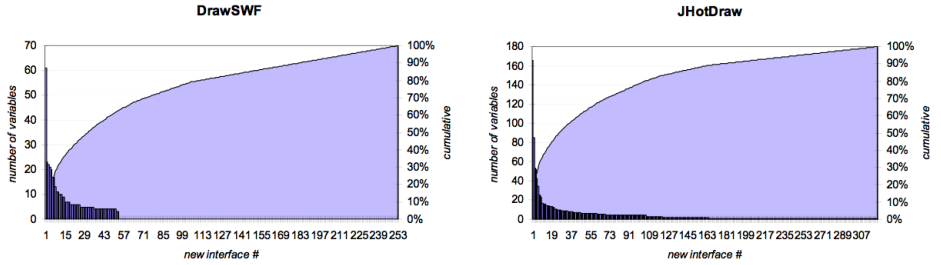


Abbildung 7: Anzahl der umdeklarierten Variablen pro neuem Interface (aus [32]).

Metrik in [8], die das Ziel verfolgt, eine automatische Einführung nur der populären Typen zu ermöglichen.

Um beurteilen zu können, ob diesem Ziel damit nähergekommen wurde, wäre es sinnvoll, die erreichte Verbesserung des ACD-Werts ins Verhältnis zu setzen zum Anwachsen der Typen eines Programms. Für diesen Wert wird nun die Bezeichnung *Decoupling/Growth Ratio* eingeführt. Er ergibt sich, indem man die zwei Zustände – vor und nach der Einführung der neuen Typen – miteinander vergleicht. Die Menge der Typen im gesamten Projekt *vorher* sei T_{before}, die *nachher* T_{after}. Weiterhin sei der durchschnittliche ACD-Wert aller Deklarationselemente vorher $\varnothing ACD_{before}$ und der nachher $\varnothing ACD_{after}$. Die Decoupling/Growth Ratio (DGR) ergibt sich dann als:

$$DGR = (\varnothing ACD_{before} - \varnothing ACD_{after}) \frac{|T_{before}|}{|T_{after}|}$$

Man beachte, dass diese Formel zwei gänzlich verschiedene Aspekte des Vorgangs in Beziehung setzt: Einerseits wird die Kopplung verringert, was sich in der Verbesserung des ACD-Wertes wiederspiegelt. Andererseits wird die Anzahl der Typen im Projekt erhöht. Zwischen den beiden Aspekten gibt es einen *Tradeoff*, denn je mehr man den durchschnittlichen ACD-Wert verbessert, desto mehr neue Typen braucht man dafür. Die DGR gibt zwar diesen Tradeoff wieder, nimmt aber keinerlei *Gewichtung* der beiden Aspekte vor. In der Praxis kann diese Gewichtung aber durchaus eine Rolle spielen, und je nach Situation unterschiedlich sein: Mitunter ist eine größtmögliche Entkopplung vielleicht wichtig, in einer anderen Situation ist es evtl. wichtiger, nicht zu viele neue Typen einzuführen.

2.4 Verfahren zur Typinferenz

Eine Frage wurde bisher nicht behandelt, und zwar, wie denn eigentlich ermittelt wird, welches das *benötigte Protokoll* eines Deklarationselementes

ist. Der Oberbegriff der dazu verwendeten Verfahren ist *Typinferenz*. Innerhalb des Eclipse Framework wird das in [41] beschriebene Verfahren verwendet, was auf dem Konzept der *Type Constraints* nach [22] beruht. Der der Implementierung dieser Arbeit zugrundeliegende Type Access Analyzer verwendet derzeit noch die *Static Class Hierarchie Analysis* nach [7]. Zur Zeit wird aber an einem neuen Analysewerkzeug [20] gearbeitet, was ebenfalls auf *Type Constraints* basiert und eine bessere Integration in die Eclipse Plattform verspricht.

2.5 Agile Softwareentwicklung

Die *Agile Softwareentwicklung* entstand aus einer gewissen Unzufriedenheit mit traditionellen Prozessmodellen. Insbesondere das klassische *Wasserfallmodell* [16] wurde als bürokratisch und zu schwerfällig empfunden [12]. Die agile Vorgehensweise sollte leichtgewichtiger und flexibler sein und so den Softwareentwicklungsprozess verbessern. In dem „agilen Manifest“ [5] werden einige Leitsätze der agilen Softwareentwicklung definiert:

> **Individuals and interactions** over processes and tools
> **Working software** over comprehensive documentation
> **Customer collaboration** over contract negotiation
> **Responding to change** over following a plan

Für das Thema dieser Arbeit ist die agile Softwareentwicklung insofern von Interesse, als sie die *nachträgliche* Einführung von Interfaces rechtfertigt. Nach dem Wasserfallmodell wäre das ein eher peinlicher Vorgang, weil es bedeuten würde, dass beim vorangegangenen Entwurf offensichtlich etwas vergessen wurde. Bei der agilen Softwareentwicklung hat es jedoch keinen negativen Beigeschmack. Dort finden sich, ebenfalls unter [5], Prinzipien wie:

> „Working software is the primary measure of progress.“

und:

> „Simplicity – the art of maximizing the amount of work not done – is essential.“

Es ist also zunächst völlig akzeptabel und sogar erwünscht, eine möglichst einfache, funktionierende Lösung zu erstellen. Die Verfeinerung der Architektur erfolgt dann später mit Refactorings – aber erst, wenn sie auch benötigt wird.

3 Das Interface-Designer Projekt

Nun soll das Kernstück dieser Arbeit, nämlich der Interface-Designer näher vorgestellt werden. Dabei wird, ausgehend von den Anforderungen, zunächst ein Überblick über die generelle Architektur gegeben. Dann werden spezielle Architekturentscheidungen auf einer etwas detaillierteren Ebene beschrieben, um dann schließlich die wichtigsten Klassen und ihr Zusammenspiel zu erklären. Anschließend wird beschrieben, wie die extension-points zu benutzen sind und wie die bereits enthaltenen Erweiterungen dieser Punkte arbeiten.

3.1 Anforderungen

Für das Projekt wurden vor der Entwicklung Anforderungen festgelegt. Dazu gehört etwa die Implementierung als Eclipse-Plugin und die Benutzung des Type Access Analyzer als Datenquelle. Eine Reihe weiterer Anforderungen ergab sich im Laufe des Projekts, entweder als neue Ideen oder auch als Reaktion auf konkrete Probleme. Insgesamt kamen die folgenden Anforderungen zusammen:

- Das Projekt sollte als Eclipse-Plugin realisiert werden.
- Die Daten für die Nutzungsanalyse sollten aus dem Type Access Analyzer der IntoJ Suite gewonnen werden.
- Die Typnutzung sollte mithilfe der Views aus der IntoJ Suite in verschiedener Form dargestellt werden (grafische und tabellarische Anzeige der gefundenen Access Sets)
- Die Views sollten so angepasst werden, dass eine Mehrfachselektion von Access Sets möglich ist. Einmal selektiert, sollte es möglich sein, aus den selektierten Access Sets ein Interface abzuleiten, dessen Protokoll durch die Vereinigung der Methoden der Access Sets definiert wird.
- Das abgeleitete Interface soll in einem neu zu erstellenden *Interface Set View* angezeigt werden, wo man es weiterbearbeiten kann und weitere Interfaces für denselben Typ hinzufügen kann. Dabei soll angezeigt werden, wie viele Deklarationselemente mit den einzelnen Interfaces umdeklariert werden würden, wenn man jedes einzelne Deklarationselement mit dem minimalen neuen Typ umdeklariert.
- Supertypen sollen bei der Analyse *nicht* berücksichtigt werden, und auch mit schon bestehenden Supertypen deklarierte Deklarationselemente *nicht* mit erfasst werden.[10]
- Die Analyse soll auf dem Protokollbegriff von [8] aufbauen, also nur mit *public* deklarierte, nicht-statische Methoden erfassen.

[10] Der Grund hierfür liegt darin, dass die Optimierung der Typhierarchie mit existierenden Typen ein Thema einer gesonderten Arbeit ist [2]. Es ist geplant, diese Komponenten in Zukunft in einer Suite zu integrieren.

- Es sollte eine Möglichkeit geschaffen werden, diejenigen Elemente des Subprotokollgraphen in allen Views auszugrauen, nach deren Aufnahme in das Interface Set die Umdeklarierung bestimmter Deklarationselemente nicht mehr eindeutig wäre. Dabei sollte ein entsprechender *Tooltip* angezeigt werden.
- Es sollte eine Möglichkeit geschaffen werden, die Interfaces auch wirklich zu erzeugen. Dazu sollte das Eclipse *Extract Interface* Refactoring aufgerufen werden und dabei in dem Wizard die Methoden für das geplante Interface gleich vorselektiert sein.
- Um den Vorgang noch weiter zu unterstützen, sollte es für den Benutzer möglich sein, eigene Metriken einzubinden, die gestützt auf die Nutzungsanalyse eine Bewertung der geplanten Interfaces berechnen können. Die Ergebnisse dieser Bewertung sollen während des Entwurfs von Interfaces in den Views angezeigt werden können.
- Es soll eine Möglichkeit geschaffen werden, Algorithmen einzubinden, die automatisch eine Menge von Interfaces vorschlagen können. Diese Algorithmen können dabei auf die Analysedaten und die Metrik zurückgreifen.
- Beide Elemente sollten als Eclipse extension-points angelegt werden, damit sie vom Benutzer durch eigene Implementierungen erweitert werden können. Dabei sollen alle gefundenen Implementierungen über die Eclipse Preferences zur Laufzeit ausgewählt werden können, um die Ergebnisse gut vergleichen zu können.
- Beispielimplementierungen sollten gleich mitgeliefert werden. Die Beispielimplementierung für die Metrik sollte auf der in [8] vorgestellten Metrik aufbauen.

Nachdem die Implementierung bis hierhin fertiggestellt war, ergab sich, dass es interessant sein könnte, die Vorschlagsfunktion auf der Basis der eingestellten Vorschlags- und ggfs. Metrikalgorithmen an einem ganzen Projekt zu testen. So könnte man schnell einen Eindruck von der Wirkungsweise der eigenen Algorithmen bekommen. Also wurde eine Möglichkeit geschaffen, für alle Typen eines Projekts Vorschläge zu erstellen. Diese werden dann in einem speziellen View angezeigt, der es auch ermöglicht, eine Zusammenfassung gängiger kopplungsbasierter Metriken anzuzeigen, und die Ergebnisse zu speichern, zu laden und als LaTeX-Tabelle zu exportieren.

Quasi als ein „Nebenprodukt“ entstand die Möglichkeit, einen zusätzlichen Eintrag im Refactoring-Menü hinzuzufügen, nämlich *Extract Interface by Usage*. Dies öffnet nach einer Analyse direkt den *Extract Interface*-Wizard, wobei bestimmte Methoden schon vorselektiert sind. Welche das sind, hängt von der eingestellten Metrik ab. Der Sinn besteht darin, dem Benutzer eine Art Abkürzung anzubieten, wenn er bereits entschieden hat, nur ein einziges Interface einzuführen.

3.2 Architektur

3.2.1 Integration in die IntoJ Suite

Die Eclipse Platform verfügt über ein ausgereiftes Plugin-Konzept[11], das die Erstellung lose gekoppelter Frameworks stark vereinfacht. Vor diesem Hintergrund wäre es naheliegend, die Integration mit den Komponenten der IntoJ Suite ausschließlich über deren Extension-points zu gestalten, so dass man die Dienste der Komponenten benutzt und ansonsten eine neue, unabhängige Komponente entwirft. Leider ist das so nicht möglich. Ein grundsätzliches Problem besteht darin, dass man ja die Ergebnisse einer Nutzungsanalyse benötigt. Diese muss Informationen über die Typnutzung bereitstellen, also über Access Sets, Deklarationselemente, usw. Diese Informationen sind von ihrer Natur her komplex, und so handelt es sich auch bei dem vom Type Access Analyzer übergebenen *TAAccessSetLattice*, der diese Informationen bereitstellt, um ein komplexes Gebilde aus einer Vielzahl von Klassen, von denen man dadurch abhängig wird. Die Situation ist ähnlich, wie wenn man mit dem Eclipse Java-Model arbeiten will: Auch dann muss man ja die Klassen aus *org.eclipse.jdt.core* importieren.

Hinzu kam, dass auf den originalen Code der IntoJ Suite während der Entwicklung kein Schreibzugriff bestand. Es mussten aber zahlreiche Änderungen an den bestehenden Plug-ins vorgenommen werden, u.a. einige wichtige Bugfixes. Eine Möglichkeit in dieser Situation wäre, den benötigten Code zu kopieren und in einer eigenen Version weiterzubearbeiten. Damit diese dann aber gleichzeitig mit der IntoJ Suite installiert sein kann, müssten alle globalen Bezeichner – die das Zusammenwirken sämtlicher geladener Plugins steuern – geändert werden, weil sie sich sonst gegenseitig überschreiben. Diese Vorgehensweise verbot sich aber unter anderem deshalb, weil geplant war, die Änderungen, Bugfixes usw. zu einem späteren Zeitpunkt wieder in die IntoJ Suite zu integrieren. Damit das „mergen“ der Quelltexte dabei noch einigermaßen zu bewerkstelligen ist, sollten nur die nötigen Änderungen an den Originalquelltexten vorgenommen werden.

Da dieser Merge aber bis jetzt nicht geschehen ist, ist zunächst durch diese Rahmenbedingungen zwangsläufig ein eigenständiges Projekt entstanden, was nicht gleichzeitig mit der IntoJ Suite installiert werden kann. Man kann also im gegenwärtigen Zustand eher von einer *Benutzung* der IntoJ Suite sprechen als von einer Integration in diese.

Dabei werden die folgenden Plugins verwendet:

- org.intoJ
- org.intoJ.analysisCore
- org.intoJ.typeAccessAnalyzer
- org.intoJ.typeAccessCalculators

[11] Eine sehr gute, übersichtliche Einführung findet sich in [2].

- org.intoJ.inferType

Die Startversionen für die Entwicklung war dabei bei allen Plugins die Version 1.1.0. Eine neue Versionsnummmer wäre beim Mergen in die IntoJ Suite festzulegen. Direkt gebraucht wird von den Plugins nur die Analysefunktion des Type Access Analyzer, durch die internen Abhängigkeiten müssen aber alle anderen ebenfalls vorhanden sein[12]. Die Contributions der Plugins zur GUI in Form von eigenen Views, Buttons, usw wurden entfernt, weil sie teilweise durch bestimmte Codeänderungen nicht mehr funktionierten und auch im Rahmen des Interface Designers nur ablenken. Ansonsten wurde aus den beschriebenen Gründen möglichst wenig Quelltext innerhalb der Plugins verändert. Die Funktionen des Interface Designers wurden dann in diesen beiden Plugins implementiert:

- org.intoJ.designer
- org.intoJ.feedrefactoring

Auf die Gründe für die Aufteilung auf zwei Plugins wird gleich noch eingegangen.

Die Benutzung des Type Access Analyzer und des damit verbundenen Datenmodells stellt für das Projekt eine Voraussetzung dar, denn schließlich sind die daraus gewonnenen Daten die Grundlage für alles andere. Gleichzeitig stellt es aber auch eine gewisse Begrenzung dar, denn da fast alle Klassen des Projekts mit den Nutzungsdaten zu tun haben, durchzieht die Abhängigkeit zu den Klassen dieser Datenstruktur praktisch das ganze Projekt. Dies ist besonders bedauerlich, da die Analyse des Type Access Analyzer bestimmte Einschränkungen[13] hat und auch nicht immer zuverlässige Ergebnisse liefert. Abhilfe ist in Sicht in Form eines neuen Werkzeugs zur Nutzungsanalyse[20]. Wenn dies fertiggestellt ist, besteht die nächste Aufgabe darin, den Interface-Designer an das veränderte Datenmodell des neuen Werkzeugs anzupassen. Man könnte sich fragen, ob man bereits jetzt den Zugriff auf die Daten durch geeignete Kapselung des *TAAccessSetLattice* entkoppeln soll. Dies erscheint jedoch aus verschiedenen Gründen nicht ratsam:

- Die Datenstruktur des Type Access Analyzer setzt sich aus einer Vielzahl von Klassen zusammen, die zu einem Großteil Wrapper für Typen des Eclipse Java Model sind. Man müsste also auch eine Vielzahl an Interfaces einführen.
- Bevor nicht feststeht, wie auf die *neuen* Analysedaten zugegriffen wird, ist eine solche Entkopplung wenig sinnvoll, da man praktisch eine weite-

[12] Das *inferType* Plugin muss vorhanden sein, weil die *Calculators* von ihm abhängen, wird aber ansonsten nicht benutzt. Man beachte, dass dieses *inferType* nicht identisch ist mit dem in [32] beschriebenen.

[13] Siehe dazu Kapitel 5.2.1.

re künstliche Struktur schafft, ohne ein Bild von den Gemeinsamkeiten und Unterschieden der alten und neuen Struktur zu haben.

- Die alten Strukturen habe einige architektonische Mängel[14], die die Entkopplung verkomplizieren.

Insgesamt kann man diese Probleme alle unter die Rubrik „mangelnde Integration in das Framework“ stellen. Denn die Begrenzungen entstehen dadurch, dass parallel zu dem Eclipse-Model noch ein eigenes Model definiert wird. Konsequenterweise benutzt das neue Tool[20] zur Codeanalyse den Eclipseeigenen Constraint-basierten Ansatz nach [41].

3.2.2 Strukturelle Überlegungen

Der Start des *Extract Interface* Refactorings mit vorselektierten Werten ist nur möglich, indem man Code aus *internal*-Packages von Eclipse benutzt. Da die Stabilität dieser APIs nicht garantiert ist, kann es passieren, dass diese Teile des Codes mit einer zukünftigen Version nicht mehr funktionieren. Die entsprechenden Funktionen wurden daher in ein eigenes Plugin in einem eigenen Feature untergebracht. Sie sind allesamt externe Contributions zum restlichen Interface-Designer, also hat dieser keine Abhängigkeiten von ihnen. Der Interface-Designer funktioniert deshalb weiter[15], falls diese Codeteile ungültig werden. Das Plugin *org.intoj.feedrefactoring* ist für den Start von *Extract Interface* zuständig. Ein weiterer Vorteil ist, dass man die Stellen, die angepasst werden müssen, durch die Abspaltung in ein eigenes Plugin leichter findet.

Auf die Definition von eigenen *internal* Packages oder nicht exportierten Packages wurde komplett verzichtet. Solche Einschränkungen sind sehr hilfreich für riesige Frameworks, bei denen die eigenen Komponenten von hunderten anderen Entwicklern für ihre Produkte benutzt werden. Bei einem kleinen Projekt mit wenig Entwicklern wie diesem, was ohnehin der ständigen Veränderung – und dem kompletten Verständnis – bedarf, sind sie nutzlos oder eher hinderlich.

Das zentrale Thema in dem Interface-Designer sind natürlich Interfaces, und so könnte man denken, es gäbe eine zentrale Model-Klasse *InterfaceModel* o.ä. Dies ist jedoch nicht der Fall. Der Grund ist, das eine solche Klasse nicht benötigt wird. Das vorhandene Datenmodell reicht nämlich für diesen Zweck vollkommen aus. Zur Beschreibung eines Interfaces genügt ein Typ plus eine Menge an Methoden, die eine Untermenge seiner public non-static Methoden ist. Beides hat man bereits in Form des *TAAccessSetLattice*. Dieser liefert den Typ, die Menge der Methoden wird durch eine Menge von

[14] Z.B. nimmt der Graphical View Schreibzugriffe auf den *TAAccessSetLattice*vor, so dass nach der Darstellung im Graphical View eine andere Anzahl an Access Sets geliefert wird als vorher.

[15] Es würde lediglich der Button zum Starten von Extract Interface verschwinden, man müsste also die Methoden dann von Hand in den Wizard übertragen.

Objekten vom Typ *AccessSet*[16] beschrieben. Es stellt sich dabei die Frage: Warum benötigt man eine *Menge* von *AccessSet*-Objekten? Ein Access Set steht doch bereits für die benötigte „Menge von Methoden".

Der Grund liegt darin, dass die Transparenz für den Benutzer gewahrt werden soll. Ein Interface wird für gewöhnlich aus mehreren *AccessSet*-Objekten zusammengesetzt[17], diese Art des Entwurfs ist im Prinzip das zentrale Anliegen des Interface-Designers. Wenn man aber die *AccessSet*-Objekte zu einem einzigen vereinigte – welches dann die Vereinigung ihrer Protokolle enthielte – ginge die Information verloren, welche *AccessSet*-Objekte ursprünglich zu diesem Protokoll geführt haben, und diese liessen sich dann z.B. nicht mehr in der grafischen Ansicht markieren. Außerdem ist der Zugriff auf die Deklarationselemente nur möglich über die *AccessSet*-Objekte, so dass diese wiederum gesondert gespeichert werden müssten, oder neue Strukturen für den Zugriff auf die Deklarationselemente geschaffen werden müssten.

Aus diesen Gründen wird ein Interface innerhalb des Designers für gewöhnlich als eine Menge von *AccessSet*-Objekten beschrieben, also ein Objekt vom Typ:

Set<TAAccessSet>

(Der Typ für Access Sets ist im TYPE ACCESS ANALYZER die Klasse *TAAccessSet*) Bei einem Interface Set haben wir es darüberhinaus mit einer Menge von *Interfaces* zu tun, dieses ist also eine Menge von Mengen von *TAAccessSet*-Objekten:

Collection[18]*<Set<TAAccessSet>>*

Wenn wir mit mehreren Interface Sets zu tun haben, ist dies entsprechend eine *Menge von Mengen von Mengen*[19].

3.3 Klassenstruktur und Packages

Von den ca. 70 Klassen des Projekts sollen hier nur die vorgestellt werden, deren Kenntniss nötig ist, um den Gesamtaufbau zu verstehen und eigene Erweiterungen schreiben zu können. Der große Umfang entstand vor allem durch die Tatsache, dass viel GUI zu implementieren war; die GUI-Programmierung ist ja dafür bekannt, große Codemengen zu benötigen. Zu diesem Projekt gehören fünf Views:

[16] Innerhalb des IntoJ Projekts werden Objekte vom Typ *AccessSet* abweichend von der Definition in dieser Arbeit auch für *unbenutzte* Subprotokolle verwendet.

[17] Genaugenommen müsste es natürlich heißen: „Das Protokoll eines Interfaces". Die verwendete Formulierung ist zwar nicht ganz korrekt, lässt sich aber wesentlich leichter lesen.

[18] Die Verwendung von Collection statt Set für den äußeren Typ erwies sich in der Implementierung als flexibler.

[19] Wenn man bedenkt, dass ein *TAAccessSet* auch wiederum für eine Menge (von Methoden) steht, könnte man sogar sagen: Eine Menge von Mengen von Mengen von Mengen.

- *Interface Set* ist der zentrale View des Interface Designers, der die bisher entworfenen Interfaces anzeigt und diverse Aktionen ermöglicht.
- *Subprotocol Lattice Graphical View* ist eine modifizierte Form des ähnlich genannten Views aus der IntoJ Suite.
- *Access Sets* basiert auf dem *Access Set Lattice Tabular View* aus der IntoJ Suite
- *Details* funktioniert ähnlich wie der *MultiInfoView* aus der IntoJ Suite.
- Der *ACD Analysis Results* View sammelt die Ergebnisse, wenn man einen Algorithmus zum Vorschlagen von Interfaces für ein ganzes Projekt testet.

(Wie man den Interface-Designer benutzt ist in Anhang D erklärt).

Jeder View benötigt für gewöhnlich eine Sammlung an Hilfsklassen in Form von Actions, Labelprovidern, Sortern, so dass schnell eine große Zahl an Klassen entsteht.

Als erster Überblick werden im Folgenden die Packages beschrieben. Dann wird das grundlegende Zusammenspiel der Klassen getrennt nach Funktionsbereichen mit Hilfe von UML-Diagrammen erläutert, und schließlich im einzelnen auf Implementierungsdetails eingegangen.

3.3.1 Packages von *org.intoJ.designer*

org.intoJ.designer – Enthält die Klassen, die für den Zugriff auf die zentralen Datenstrukturen des Interface-Designers gebraucht werden.

org.intoJ.designer.metrics – Enthält alles, was im Zusammenhang mit der Erweiterung des Metrik-Erweiterungspunktes von Bedeutung ist.

org.intoJ.designer.proposer – Enhält alles zu dem Erweiterungspunkt für die Vorschlagsalgorithmen.

org.intoJ.designer.ui – GUI Klassen, die nicht zu einem View gehören: Die Interface-Designer Perspektive, sowie die Preference Pages für Metriken und Vorschlagsalgorithmen.

org.intoJ.designer.ui.views – Die Views samt ihrer Hilfsklassen.

org.intoJ.designer.ui.actions – Die Actions.

org.intoJ.designer.util – Die Klasse *Util*, die global benötigt Funktionen zur Verfügung stellt, die Iteratoren für den Iterative Interface Set Proposer, u.a.

Das org.intoJ.feedrefactoring Plugin Dieses Plugin ist mit seinen neun Klassen und seiner einfachen Funktionalität sehr übersichtlich, deswegen braucht es hier nicht extra aufgeführt zu werden.

3.4 Implementierung

Die Implementierung ist relativ umfangreich[20], deswegen werden nun jeweils einzelne Teilbereiche betrachtet. Diese ergeben sich aus den internen Abläufen der Anwendungsfälle. Ein typischer Anwendungsfall wäre etwa:

1. Der Anwender ruft auf einem Typ seines Programms die Analyse auf.
2. Diese wird dann in den 3 Views dargestellt: *Subprotokoll Lattice Graphical View*, *Access Sets* und *Details*.
3. Er markiert in einem der ersten beiden Views einige Access Sets, und importiert diese als neues Interface in den *Interface Set* View.(wird evtl. mehrmals wiederholt)
4. Er erzeugt diese Interfaces, indem er mehrmals das Extract Interface Refactoring aufrufen lässt.

Für die Implementierung ergeben sich dadurch die folgenden Anforderungen:

- Analysedaten beschaffen,
- in die Analyse-Perspektive wechseln,
- Daten in den Views darstellen, Views bei Selektionen synchron halten,
- beim Import in den Interface Set View das gewählte Interface abspeichern und ermitteln, welche Deklarationselemente damit umdeklariert werden sollen, sowie
- das Extract Interface Refactoring mit einem vorkonfigurierten Wizard starten.

3.4.1 Die Analyse starten

Um die Analysedaten zu erhalten, wird der TYPE ACCESS ANALYZER benutzt. Der in dem Plugin *org.intoJ.typeAccessAnalyzer* vorgesehene extensionpoint *accessSetLatticeListener* erweist sich als nicht hinreichend, weil er nur die Möglichkeit bietet, in Art des *Observer Patterns* [14] das Ergebnis einer anderswo angestoßenen Analyse mitgeteilt zu bekommen. Diese Aktion löst aber auch etliche andere Aktionen aus. Es bleibt daher nur die Möglichkeit, direkt auf die *Calculators* zuzugreifen. Dies geschieht mit der Methode:

```
IAccessSetCalculator calc=
TypeAccessAnalyzerPlugin.getDefault().getSelectedCalculator();
```

[20] Ca. 11000 LOC.

Hat man so das gewünschte Objekt erhalten, erhält man den *TAAccessSetLattice* mit folgendem Aufruf:

```
TAAccessSetLattice lattice=
calc.calculateAccessSetsOfType(type, monitor);
```

Wobei die Variable *type* für den Typ steht, auf dem man die Analyse aufrufen möchte, *monitor* steht für den Fortschrittsbalken, der dem Benutzer ein grafisches Feedback über den Fortschritt liefern soll, und auch einen evt. Abbruch handeln muss[21]. Die Codeteile, die sich mit der Behandlung von Exceptions, dem Starten eines Threads, usw. beschäftigen, sind in diesem und auch den folgenden Codebeispielen weggelassen. Die Erzeugung eines *TAAccessSetLattice* ist gekapselt in der Klasse *LatticeMaker* in der Package *org.intoJ.designer*. Alle Klassen, die eine Analyse auf einem Typ durchführen wollen, tun dies also, indem sie diese Klasse benutzen.

Bevor die Daten weiterverarbeitet werden, soll kurz angedeutet werden, wie die gewünschten Informationen aus dem *TAAccessSetLattice* gewonnen werde. Für eine genaue Beschreibung sollte man die Dokumentation [19] und die Quelltextkommentare zu Rate ziehen. Wichtig sind die folgenden Methoden von *TAAccessSetLattice*:

- *TypeElement getRootType()* liefert den Typ, der analysiert wurde.
- *TypeSet getSupertypes()* liefert die Supertypen diese Typs. Das sind die Klassen, von denen er erbt, sowie die Interfaces, die von ihm selbst oder einer dieser Klassen implementiert werden.
- *Collection getTAAccessSets()* liefert die Access Sets.

Wie schon erwähnt, bekommt man es hier schnell mit einer Fülle an Model-Klassen zu tun. Dringend benötigt werden noch die Deklarationselemente; diese erhält man über die Access Sets, und zwar deren Methode *DESet_Key getDeclarationElements()*. Der Rückgabewert, *DESet_Key*[22] stellt eine eigene Collection-Klasse zur Aufnahme von Objekten des Typs *DeclarationElement* dar.

Bemerkenswert an dieser Struktur ist, dass die Deklarationselemente *Teil der Access Sets* sind. Das heißt, es gibt kein Deklarationselement ohne ein Access Set, und daraus resultiert auch, dass Deklarationselemente, deren Access Set nicht ermittelt werden kann oder nicht unter den verwendeten Protokollbegriff fällt, nicht erfasst werden. Wenn also z.B. auf einem Deklarationselement eine Methode mit *default* Sichtbarkeit aufgerufen wird, verschwindet es aus dem *TAAccessSetLattice*. Dies ergibt in gewisser Hinsicht durchaus Sinn, denn die Deklarationselemente, die nicht unter den Protokollbegriff fallen, sollen, bzw. können auch nicht umdeklariert werden.

[21] Leider ist weder das eine noch das andere in den *Calculators* korrekt implementiert, die Analyse lässt sich daher nicht unterbrechen.

[22] Man beachte unbedingt die Ausführungen in den Quelltextkommentaren zu den Unterschieden der Collection Klassen *DESet_Key* und *DESet_Signature*.

3.4.2 Die Daten filtern

Der Protokollbegriff, der im Interface-Designer verwendet wird, weicht etwas ab von dem des TYPE ACCESS ANALYZER. Der *TAAccessSetLattice* enthält deshalb mehr Elemente als er soll, nämlich auch die Deklarationselemente, in denen auf mit *public* deklarierte Felder und statische Elemente zugegriffen wird. Diese Elemente müssen also entfernt werden. Architektonisch gesehen wäre es am elegantesten, dies bei der Gewinnung der Daten zu tun, also indem man den *Calculator* z.B. entsprechend konfiguriert[23]. Da dies jedoch nicht ohne weiteres möglich ist, ist der einfachste Weg, die unerwünschten Elemente nachträglich herauszufiltern. Zu diesem Zweck dient die Klasse *FilteredLattice*, die zwei weitere Hilfsklassen benötigt, *FilteredTypeElement* und *FilteredAccessSet*, alle zu finden im Paket *model*.

Beim Konstruktoraufruf eines gefilterten Lattice werden der originale Lattice und einige Flags übergeben, die bezeichnen, was gefiltert werden soll, also in diesem Fall entsprechend der Vorgabe so:

```
private TAAccessSetLattice filtered(
        TAAccessSetLattice lattice) {
    if (lattice instanceof FilteredLattice)
        return lattice;
    return new FilteredLattice(lattice,
            INCLUDE_ONLY_PUBLIC_NONSTATIC_METHODS
                    | NO_SUPERTYPES);
}
```

Der Rückgabetyp ist wieder vom Typ *TAAccessSetLattice*. Dies ist erforderlich, damit der gefilterte Typ auch mit den „alten“ Views, insbesondere der grafischen Darstellung funktioniert. Es wird durch den einfachen Trick erreicht, dass der gefilterte Lattice von dem normalen erbt. Das dabei ererbte Verhalten wird zunächst deaktiviert, indem alle öffentlichen Methoden überschrieben werden. Zusätzlich hält er eine Referenz auf den ungefilterten Lattice als *Sklaven* und benutzt diesen teilweise zur Implementierung der überschriebenen Methoden. So kann genau kontrolliert werden, dass die herausgegebenen Daten entsprechend der Flags gefiltert werden.

3.4.3 Die Daten verteilen

Die nächste Aufgabe besteht darin, die erhaltenen Informationen an alle Views zu verteilen und sicherzustellen, dass alle Views auch stets dieselben, aktuellen Daten anzeigen. Dafür bietet sich das *Observer Pattern* [14] an. Es

[23] In der Tat wurde der Calculator *zusätzlich* verändert, und zwar so, dass bereits bei der Analyse die Supertypen herausgefiltert werden. Dies war notwendig, weil die Calculators sonst bei bestimmten Codeelementen mit einer Fehlermeldung abbrachen. Ein Nebeneffekt ist die bessere Performance der Analyse.

ist in Form der Klasse *LatticeProvider* und dem Interface *LatticeInterest*[24] implementiert. In der Klasse *LatticeProvider* wird, nachdem der Lattice erzeugt wurde, eine Referenz auf ihn gespeichert. Dazu wird die Methode

```
public static void setNewLattice(
        TAAccessSetLattice accessSetLattice)
```

aufgerufen. Dadurch werden die angemeldeten Listener benachrichtigt, das sind in der Regel die beteiligten Views. Wenn der letzte Listener sich abmeldet, wird die Referenz auf den *TAAccessSetLattice* auf *null* gesetzt, um den Speicherplatz wieder freizugeben. Durch dieses Verfahren wird erreicht, dass der *TAAccessSetLattice* nicht „verloren geht“, wenn z.B ein View geschlossen und wieder geöffnet wird. Dies ist insbesondere wichtig, weil die Berechnung u.U. lange dauert. Durch die Trennung der Vorgänge *Erzeugen* und *Verteilen* mit Hilfe der Klassen *LatticeMaker* und *LatticeProvider* haben benutzende Klassen zudem die Möglichkeit, einen Lattice zu berechnen, ohne ihn anzeigen zu lassen.

3.4.4 Die Synchronisation und Kommunikation der Views

Die verschiedenen Views sind zunächst einmal unabhängige Komponenten, die einander nicht zu kennen brauchen. Es stellen sich also zwei Fragen:

1. Wenn Elemente in einem View markiert werden, wird die Markierung in den anderen Views jeweils mitgeführt bzw. die Anzeige angepasst. Wie ist das möglich, wenn sich die Views nicht kennen?
2. Wenn man Subprotokolle bzw. Access Sets markiert hat, kann man sie als neue Interfaces in den Interface Set View importieren. Woher weiß der Interface Set View, welche Methoden die Interfaces enthalten müssen?

Antwort auf beide Fragen bietet die Eclipse Plattform in der Gestalt des *Selection Service*. Eine genaue Einführung findet sich in [27] und [17], hier genügt es, nur seine grundlegenden Eigenschaften zu nennen: Es handelt sich um eine Art Broadcast-Dienst, der das, was gerade vom Benutzer selektiert ist, an alle interessierten Komponenten verteilt. Eine solche Selektion könnte alles mögliche sein, also etwa Text in einem Editor, oder auch ein ganzes Projekt im Package Explorer, eben alles, was man in der GUI selektieren kann.

Interessierte Klassen können wählen, ob sie nur über die Selections informiert werden wollen, oder auch selbst Selections in den Service einspeisen. Um diese Rollen annehmen zu können, müssen sie jeweils ein bestimmtes Interface implementieren und sich bei der Plattform registrieren. Der *Selection*

[24] Das Interface wurde bewusst nicht *LatticeListener* genannt, um eine Verwechslung mit dem extension-point *TAAccessSetLatticelistener* aus dem Type Access Analyzer zu vermeiden.

Service stellt darüberhinaus gleich bestimmte Datenstrukturen wie Listen und Bäume zur Verfügung, um auch komplexe Daten auf einheitliche Art zur Verfügung zu stellen. Die Empfänger einer Selection stellen dann für gewöhnlich durch eine Reihe von *instanceof* und *cast* Operationen fest, um was für eine Selection es sich handelt, und ob sie für sie von Interesse ist.

Für die Implementierung des Interface-Designers bot es sich an, eine Liste von Access Sets[25] über den Selection Service zu verteilen. Ähnlich wurde dies bereits in der IntoJ Suite gehandhabt. Allerdings besteht der Unterschied darin, dass dort immer *nur ein* Access Set selektiert sein kann. Wenn also eine Auswahl von Access Sets in den Interface Set View importiert wird, erhält dieser eine Liste von Access Sets über den Selection Service. Die Information, welches Interface der Benutzer dadurch definieren möchte, ist in dieser Liste implizit als Vereinigungsmenge ihrer Methoden, im IntoJ Suite-Jargon *Invocations* genannt, gegeben. Der Interface Set View muss jedoch auch ermitteln, welche Deklarationselemente mit welchem der neuen Interfaces umzudeklarieren wäre. Unter anderem deshalb verwendet er als Model eine spezielle Datenstruktur, die nun genauer vorgestellt wird.

3.4.5 Das Model des Interface Set View

Zu der Datenstruktur gehört das Interface *InterfaceSet*, die entsprechende Implementierung *InterfaceSetImpl*, die Hilfsklasse für die Elemente eines Interface Sets, die Klasse *InterfaceSetElement*, sowie die Enums *Type* und *Mode*. Sie befinden sich alle im Package *model*. Zusammenfassend gesagt, ist ein Interface Set eine spezialisierte Collection, die intern eine Collection von Typen und eine Collection von Deklarationselementen verwaltet. Abbildung 8 zeigt ein UML-Diagramm der wichtigsten beteiligten Klassen. Das Diagramm stellt nur die in diesem Abschnitt erläuterten Zusammenhänge dar, außerdem wurden die „use“-Abhängigkeiten zu den Datenstrukturen des Type Access Analyzer weggelassen. Der Zugriff auf das Model geschieht über das Interface *InterfaceSet*, was auch gleich die Factory in Form einer statischen Klasse bereitstellt. Man erhält also ein Objekt vom Typ *InterfaceSet* durch folgenden Aufruf:

```
InterfaceSet interfaceSet = InterfaceSet.Factory
        .create();
```

Das Interface Set ist so angelegt, dass ein einziges Objekt für verschiedene Interface Sets verschiedener Typen weiterverwendet werden kann. Einmal erzeugt, wird es mit einem Lattice durch einen Aufruf von:

```
public void createNew(TAAccessSetLattice accessSetLattice);
```

[25] Da im Type Access AnalyzerObjekte vom Typ *TAAccessSet* sowohl für Access Sets als auch für Subprotokolle, also unbenutzte Access Sets benutzt werden, wird der Ausdruck Access Set im Zusammenhang mit der Implementierung auch in diesem Sinn verwendet, um die Lesbarkeit zu erhalten.

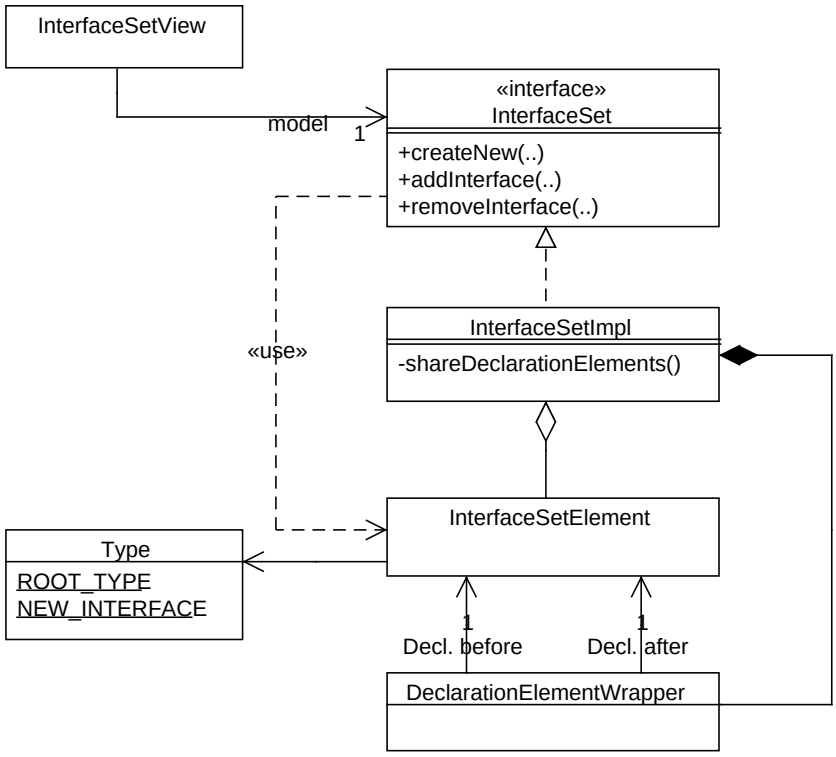


Abbildung 8: Das Model des Interface Set View.

initialisiert. Dabei wird der Basistyp, also der Typ, durch dessen Analyse der *TAAccessSetLattice* entstanden ist, als ein Interface Set Element gespeichert. Er wird dazu in die Klasse *InterfaceSetElement* verpackt und in der internen Collection gespeichert. In derselben Collection werden dann auch die neuen Interfaces gespeichert. In der Collection befinden sich also ein *existierender* Typ, nämlich der Basistyp[26], und evt. mehrere *neue, erst noch geplante* Typen – die neuen Interfaces. Die Unterscheidung um was es sich jeweils handelt geschieht durch das Enum *Type*, für das jedes Interface Set Element ein Feld besitzt[27]. Der Typ eines Interface Set Elements – also entweder *ROOT_TYPE* oder *NEW_INTERFACE* – lässt sich dann mit der Methode

```
public Type getType()
```

der Klasse *InterfaceSetElement* erfragen.

[26] Im Programm als *root type* bezeichnet.

[27] Ursprünglich war auch das Erfassen von Supertypen und bereits existierenden Interfaces im Interface Set vorgesehen. Deswegen sind auch diese Typen in dem Enum *Type* vorhanden. Durch die vorgegebene Filterung der Analyse von diesen Elementen wird davon jedoch gegenwärtig kein Gebrauch gemacht.

Nun besitzt man also ein Interface Set, was als einziges Element den Root Type enthält. Wie beschrieben, sollen neue Interfaces in der Form einer Liste von Access Sets hinzugefügt werden. Das geschieht mittels der Methode:

```
public void addInterface(Set<TAAccessSet> accessSets);
```

Hier ist die Implementierung dieser Methode in *InterfaceSetImpl*:

```
public void addInterface(Set<TAAccessSet> accessSets) {
  (..)
  String name = "New_" + count++;
  elements.add(new InterfaceSetElement(accessSets, Util
      .getUnionOfMethods(accessSets), name, NEW_INTERFACE));
  shareDeclarationElements();
}
```

Die Variable *elements* ist der interne Container für die Elemente. Man kann hier sehen, wie ein neues Element durch den Konstruktoraufruf von *InterfaceSetElement* erzeugt und in die Collection eingefügt wird. Die Parameter *name* und der *Type*, also in diesem Fall *NEW_INTERFACE*, sind dabei klar. Was sich nicht selbst erschließt, ist, warum sowohl die Menge der Access Sets als auch die Vereinigungsmenge ihrer Methoden mit übergeben wird. Der Grund hierfür wurde bereits in Abschnitt 3.2.2 angesprochen: Es soll nicht nur die Information gespeichert werden, die für die Definition des Interfaces erforderlich ist (also dessen Protokoll), sondern auch, mit welcher Selektion von Access Sets *dieses Interface erzeugt wurde*. Wenn ein Interface dann im Interface Set View selektiert wird, sollen die für die Konstruktion benutzten Access Sets in den anderen Views automatisch wieder markiert werden.

Übertragen auf den betrachteten Konstruktoraufruf bedeutet das: Der erste Parameter steht für die Selektion des Benutzers, der zweite für das resultierende Protokoll. Beides wird mit dem Element gespeichert.

In der Methodenimplementierung wird als nächstes die Methode *shareDeclarationElements()* aufgerufen. Sie ist wichtig, denn sie repräsentiert eine in den Anforderungen geforderte Funktionalität: anzuzeigen, welche Deklarationselemente mit welchen der neuen Interfaces umdeklariert würden, wenn man die neuen Interfaces einführt. Es geht also sozusagen um einen Blick in die Zukunft. Dieser ist zwangsläufig nur eine Annäherung an das, was tatsächlich entstünde, denn das Eclipse Refactoring verwendet einen eigenen Algorithmus, um zu entscheiden, welche Deklarationselemente umdeklariert werden. Daher kann das Resultat von der berechneten „Vorschau“ durchaus abweichen. Dies ist ein Problem, was durch den gewählten Ansatz auf der Analyse des Type Access Analyzer aufzubauen unvermeidlich entsteht; eine absolut zutreffende Vorschau liesse sich nur erreichen, indem man denselben Algorithmus wie das Eclipse-Refactoring verwenden würde.

Man muss sich also momentan damit zufrieden geben, dass die Zuordnung der Deklarationselemente zu den Interfaces eine Art Vorschau ohne absolute Garantie ist.

Wie wird diese nun berechnet? Die Methode *shareDeclarationElements()* verzweigt sich weiter in die folgende Methode:

```
private void assignDEwithNewInterfacesOnly() {
  // Sortieren nach der Anzahl der Methoden,
  // absteigend
  Collections.sort(elements,
      new Comparator<InterfaceSetElement>() {
        public int compare(InterfaceSetElement e1,
            InterfaceSetElement e2) {
          return e1.getCompareValue()
              - e2.getCompareValue();
        }
      });
  // Den Status quo wiederherstellen
  for (DeclarationElementWrapper de :
    declarationElementWrappers) {
    if (de.declaredWithBefore != null) {
      de.declaredWithAfter = de.declaredWithBefore;
    }
  }
  // Jedem DE das kleinstmögliche der neuen
  // Interfaces zuordnen
  for (InterfaceSetElement e : elements) {
    if (e.type != NEW_INTERFACE)
      continue;
    for (DeclarationElementWrapper d :
      declarationElementWrappers) {
      if (e.getSignatureInvocations().containsDeclarations(
          d.getInvocationsSignature())) {
        d.declaredWithAfter = e;
      }
    }
  }
}
```

Um die Bedeutung dieses Codes nachzuvollziehen, muss man sich zunächst klar machen, dass das Interface Set ja praktisch zwei Zustände nebeneinander verwaltet. Diese zwei Zustände beziehen sich auf den Jetzt-Zustand oder den Status quo, in dem alle Deklarationselemente mit dem Root Type deklariert sind. Der andere Zustand betrifft den Zeitpunkt, *nachdem alle Refactorings durchgeführt wurden*, und unterscheidet sich insbesondere dadurch, dass die Deklarationselemente nun mit den in diesem Zug neu eingeführten Interfaces umdeklariert sind. Diese beiden Zustände werden direkt mit den Deklarationselementen gespeichert; dafür gibt es in *InterfaceSetImpl* die innere Klasse

DeclarationElementWrapper[28]. Sie besitzt zur Modellierung der zwei Zustände die Felder:

```
InterfaceSetElement declaredWithBefore;
InterfaceSetElement declaredWithAfter;
```

Sie verweisen auf jeweils ein Element des Interface Sets. Dabei zeigt die erste Variable auf den Typ, mit dem das Deklarationselement tatsächlich deklariert ist, also immer auf den Root Type[29]. Die zweite zeigt auf den Typ, mit dem das Deklarationselement nach sämtlichen Refactorings (voraussichtlich) deklariert wäre, in der Belegung dieser Variablen manifestiert sich also die angestrebte Vorschau. In der oben abgedruckten Methode *shareDeclarationElements()* muss die richtige Belegung für die zweite Variable gefunden werden. Dazu wird der folgende Algorithmus angewendet:

1. Die Typen im Interface Set werden sortiert, so dass die mit dem kleinsten Protokoll *zuletzt* kommen.
2. Die bestehende Vorschau wird gelöscht, indem alle *declaredWithAfter* Felder auf den Wert von *declaredWithBefore* gesetzt werden.
3. Die neuen Interfaces werden der Reihenfolge nach durchgegangen, und jeweils alle Deklarationselemente, bei denen dies möglich[30] ist, mit ihnen umdeklariert. Dadurch, dass die neuen Typen absteigend nach der Größe ihres Protokolls sortiert sind, stellt dieses Verfahren sicher, dass jedes Deklarationselement mit dem minimalen Typ assoziiert wird.

Dieser Algorithmus wird immer ausgeführt, wenn ein Interface zum Interface Set hinzugefügt oder aus ihm entfernt wurde.

Zum Entfernen eines Interfaces und zum Zugriff auf sämtliche Elemente gibt es dann die für Collections typischen Methoden, wie z.B.:

```
public void removeInterface(InterfaceSetElement e);
public List<InterfaceSetElement> getElements();
public void clear();
```

Auf diese alle einzeln einzugehen würde hier zu weit führen, ihre Bedeutung ist in der Javadoc von *InterfaceSet* beschrieben.

[28] Bei der oben beschriebenen Initialisierung des Interface Sets mit einem Lattice werden aus ihm alle Deklarationselement extrahiert und in diese Wrapper verpackt gespeichert. Dazu dient die interne Collection *declarationElementWrappers*.

[29] Da sie immer auf den Root Type zeigt, könnte man auf diese Variable folglich momentan auch verzichten. Mit ihr ist es aber einfacher, das Konzept in der Zukunft auch auf Deklarationselemente auszudehnen, die **nicht** mit dem Root Type sondern etwa mit einem bereits existierenden Interface deklariert sind.

[30] Weil ihr Protokoll das Access Set dieser Deklarationselemente einschließt.

3.4.6 Das Model an den Selection Service anpassen

Im Zusammenhang mit dem oben erwähnten *Selection Service* ergibt sich beim Interface Set View eine kleine Komplikation: Wenn ein Element in ihm selektiert wird, würde der normale SelectionProvider des verwendeten Table Viewers[31] Objekte vom Typ *InterfaceSetElement* an den Selection Service senden. Mit diesen können aber die anderen Views nichts anfangen. Deshalb hat der Interface Set View einen speziellen Selection Provider, der von den Elementen wieder die Menge von Access Sets erfragt, die bei ihrer Erstellung benutzt wurde. Diese Menge wird an den Selection Service geschickt, so werden dieselben Access Sets in den anderen Views automatisch wieder selektiert.

3.4.7 Die Interfaces erzeugen: Aufruf von Extract Interface

Die GUI-Elemente, mit denen das Extract Interface Refactoring gestartet werden kann, finden sich wie bereits beschrieben in einem anderen Plugin, nämlich in *org.intoJ.feedrefactoring*. Normalerweise hat man auf den Wizard, der bei dem Refactoring erscheint, keinen Zugriff. Deshalb befindet sich in der Package *org.intoJ.feedrefactoring.ui* eine Kopie des Original-Wizards aus dem Eclipse Framework. Dieser nimmt dann in seinem Konstruktor (zusätzlich zu dem Objekt vom Typ *Refactoring*, was auch der normale Konstruktor benötigt) die Daten entgegen, mit denen er vorkonfiguriert werden soll:

```
public FeedExtractInterfaceRefactoringWizard(
    ExtractInterfaceRefactoring refactoring,
    IMember[] preselected, String interfaceName) {
this(refactoring);
  (..)
}
```

Wie man sieht, wird der Name, den der Benutzer auch bereits im Interface Set View bestimmen kann übergeben, sowie die zu selektierenden Methoden als Array von *IMember*. Ebenfalls benötigt wird natürlich der Typ, auf dem das Refactoring ausgeführt werden soll, als Objekt vom Typ *IType*. An diesem Punkt kehrt man also zurück zum Eclipse Java-Model. Um diese Daten bequem zu erhalten, stellt *InterfaceSet* spezielle Methoden zur Verfügung. Mit

```
public IType getRootType();
```

erhält man den Basistyp als *IType*. Mit

```
public IMember[] getMembersForType(
    InterfaceSetElement element);
```

[31] Die *Viewer* sind ein von der Platform angebotenes Konstrukt, das den Bau von Models für Tabellen und Bäume in der GUI erheblich erleichtert, genaueres findet man in [27]. Sie bieten auch bereits einen „eingebauten“ SelectionProvider, der einfach die dargestellten Objekte in den Selection Service einspeist.

erhält man die Methoden eines speziellen Elements, und mit

```
public List<RefactoringData> getRefactorings();
```

eine Liste mit Refactorings, gespeichert in Objekten vom Typ *RefactoringData*. Diese Klasse fasst die drei benötigten Informationen: Typ, Methoden und Name zusammen.

Der eigentliche Start des Refactorings geschieht dann in der Klasse *ExtractInterfaceStarterImpl*:

```
public class ExtractInterfaceStarterImpl implements
    ExtractInterfaceStarter {

public void start(IType type, IMember[] preselected,
    String interfaceName) {
  Shell shell = PlatformUI.getWorkbench().getDisplay()
      .getActiveShell();

  final ExtractInterfaceRefactoring refactoring =
    new ExtractInterfaceRefactoring(
      new ExtractInterfaceProcessor(type,
          JavaPreferencesSettings
              .getCodeGenerationSettings(type
                  .getJavaProject())));
  if (!ActionUtil.isProcessable(shell, refactoring
      .getExtractInterfaceProcessor().getType())) {
    return;
  }

  try {

    new RefactoringStarter()
        .activate(
            refactoring,
            new FeedExtractInterfaceRefactoringWizard(
                refactoring, preselected, interfaceName),
            shell,
            RefactoringMessages.
            OpenRefactoringWizardAction_refactoring,
            true);
  } catch (JavaModelException e) {
    e.printStackTrace();
  }
}
}
```

Da diese Klasse ja wegen der Abhängigkeit von *internal* Packages vielleicht einmal angepasst werden muss, wird auf sie über das Interface *ExtractInterfaceStarter* zugegriffen. Die Factory ist auch hier gleich in das Interface

integriert, der Aufruf sieht dann also, nachdem man die nötigen Informationen beisammen hat, so aus:

```
ExtractInterfaceStarter.Factory.create().start(type,
      methods, name);
```

Zu beachten ist dabei, dass der Aufruf im UI Thread erfolgen muss.

3.4.8 Unerwünschte Subprotokolle markieren, Tooltip anzeigen

Ein weiteres gefordertes Feature ist die Möglichkeit, dass in den Views „Subprotocol Lattice Graphical View“ und „Access Sets“ angezeigt wird, welche Access Sets sinnvollerweise *nicht* mehr zu einem neuen Interface hinzugefügt werden sollen. Dazu müssen diese Views Kenntnis von dem Inhalt des Interface Set haben. Um die lose Kopplung der Views zu erhalten, greifen sie jedoch nicht direkt auf den Interface Set View zu, sondern melden sich bei der Klasse *InterfaceSetProvider* als Listener an, ganz ähnlich wie es bereits beim *TAAccessSetLattice* gehandhabt wurde. Das Interface *InterfaceSet* bietet dann zwei Methoden, die die gestellte Aufgabe erfüllen:

```
public boolean isAllowed(DESet_Signature dess);
public String getToolTip(DESet_Signature dess);
```

Die erste Methode dient dazu, zu erfragen, ob ein bestimmtes Protokoll, hier repräsentiert durch das Objekt vom Typ *DESet_Signature*, zum Einfügen in das Interface Set empfohlen ist. Wenn nicht, kann mit der zweiten Methode ein String mit einer Begründung abgefragt werden, der dann an den Tooltip des Elements in den Views angehängt wird.

In der Implementierung werden entsprechend den Anforderungen diejenigen Protokolle abgelehnt, die eine neue Umdeklarierungsmöglichkeit für Deklarationselemente eröffnen würden, die bereits einem neuen Interface im Interface Set zugeordnet sind. Beim Testen stellte sich heraus, dass es für den Benutzer einigermaßen verwirrend ist, wenn mit dieser nicht ganz einfachen und auch nicht zwingenden Begründung plötzlich eine potentiell große Zahl von Elementen ausgegraut wird, deswegen ist das Feature per Voreinstellung deaktiviert.

3.4.9 Die Metriken integrieren

Der erste Anwendungsfall ist damit abgeschlossen. Ein weiteres Element des Interface-Designers ist die Fähigkeit, Metriken einzubinden. Sie sollen jeweils ein einzelnes Interface oder ein Interface Set als Input benutzen und dazu einen numerischen Wert liefern. Der Wert soll eine Art Hilfe für den Benutzer sein, zu erkennen, ob sich die Einführung des bzw. der Interfaces lohnt. Es soll möglich sein, auf einfache Art eigene Metriken einzubinden und zu testen, ob sie geeignet sind, diesen Anspruch zu erfüllen.

Dazu ist es sinnvoll, die berechneten Werte direkt in den Views anzuzeigen. Im *Subprotocol Lattice Graphical View* kann der Benutzte ein einzelnes Interface definieren, indem er einen oder mehrere Knoten markiert. Deshalb ist es naheliegend, in ihm den Metrik-Wert für ein *einzelnes* Interface anzuzeigen. Der Interface Set View zeigt dagegen eine *Menge* von Interfaces, und sollte deshalb den Metrik-Wert für die gesamte Menge anzeigen. Also stellt sich die Frage, wie die Views auf der Basis ihres Inhalts die Werte erhalten.

Einen Überblick über die Klassenhierarchie der Metriken findet man in Abbildung 9. Auch hier wird wieder auf die Darstellung der Abhängigkeiten von

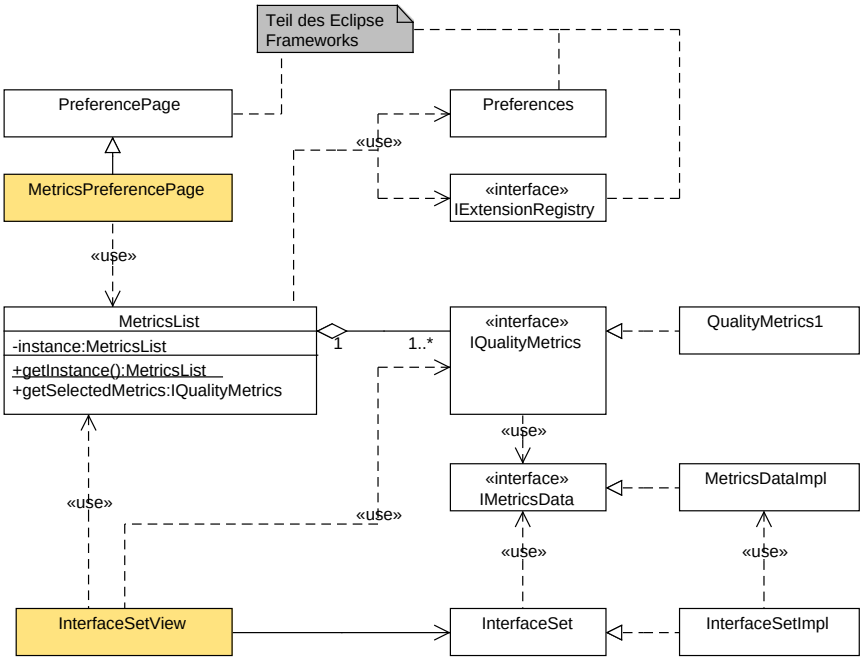


Abbildung 9: Der Zugriff auf die Metriken. (Die GUI-Klassen sind farbig gekennzeichnet.)

den Type Access Analyzer Datenstrukturen verzichtet. Es wurde beim Entwurf der Klassen allerdings großer Wert darauf gelegt, bei den Metriken diese Abhängigkeit zu vermeiden. Der Grund ist, dass Metriken unabhängig von der verwendeten Datenstruktur testbar sein sollen. Die Interfaces *IQualityMetrics* und *IMetricsData* weisen deshalb keine Abhängigkeiten von den Klassen des Type Access Analyzer auf. Das Interface *IMetricsData* stellt dabei eine Abstraktion der für die Metrik relevanten Daten eines konkreten Typs dar. Es ist bewusst sehr einfach gehalten:

```
public interface IMetricsData {
  int getNumberOfMethods();
```

```
  int getNumberOfDeclarationElements();
  int getUsedMethodsOfDeclarationElement(int index);
}
```

Die erste Methode liefert die Größe des Protokolls, die zweite die Anzahl der mit diesem Typ deklarierten Deklarationselemente. Ist die Anzahl bekannt, kann man durch die Übergabe eines Index an die dritte Methode die Anzahl der tatsächlich benutzten Methoden, also die Größe des Access Sets für das mit dem Index spezifizierte Deklarationselement erhalten.

Die Entkopplung von der Datenstruktur wird nun erreicht, indem die eigentliche Metrik nur mit Objekten vom Typ *IMetricsData* und nicht mit der Datenstruktur selbst arbeitet. Das Interface *IQualityMetrics* definiert die folgenden Methoden:

```
public interface IQualityMetrics {
  String getName();
  String getDescription();
  float getQuality(IMetricsData rootType,
    IMetricsData aInterface);
  float getQuality(IMetricsData rootBeforeRefac,
    IMetricsData rootAfterRefac,
    Collection<IMetricsData> interfaceSet);
}
```

Es werden also keine Klassen des *TAAccessSetLattice* in dem Interface benutzt. Die ersten beiden Methoden dienen zur Anzeige der Metriken in der GUI. Die erste *getQuality(..)* Methode dient dann der Ermittlung des Werts für ein *einzelnes* Interface, die zweite der Ermittlung für eine *Gruppe* von Interfaces. Die metrischen Daten des Basistyps werden dabei gesondert von denen der neuen Interfaces erfasst.

Irgendwo im Programm müssen dann natürlich die Daten des Sprachmodells in die passende Form gebracht werden, um von den Metrikimplementierungen verarbeitet werden zu können, sprich: in Objekte vom Typ *IMetricsData*. Das Interface *InterfaceSet* bietet dazu entsprechende Methoden.

Die Klasse *MetricsList* dient zur Erzeugung, Auswahl und Verwaltung der implementierten Metriken. Sie benutzt dazu die Klassen des Eclipse Frameworks, um alle bereitgestellten Metriken zu finden und zu instanziieren. Sie selbst dient dann der für die Metrikauswahl zuständigen GUI-Klasse *MetricsPropertyPage* als Datenlieferant und übernimmt auch das persistente Speichern der Auswahl. Für die restlichen Klassen des Interface-Designers ist ihre Methode *getselectedMetrics(..)* wichtig, weil sie die aktuell ausgewählte Metrik-Instanz liefert.

3.4.10 Die Vorschlagsalgorithmen integrieren

Analog zur Klasse *MetricsList* existiert auch eine Klasse *ProposerList*, die die analogen Dienste für den zweiten extension-point anbietet. Dieser reprä-

sentiert einen Algorithmus, um automatisch ein Interface Set vorzuschlagen. *Wie* der Algorithmus dies tut, ist nicht festgelgt, er kann dazu auf die Metriken zugreifen, muss es aber nicht. Das Framework für die *Proposer* (so wird der Vorschlagsalgorithmus im Programm bezeichnet) hat als zentrale Klasse die abstrake Klasse *AbstraktInterfaceSetProposer*. Wenn der entsprechende extension-point erweitert wird, muss eine Klasse per plugin.xml bekannt gemacht werden, die von dieser Klasse erbt. Sie implementiert dann den Algorithmus. Die Klasse *AbstraktInterfaceSetProposer* wurde so konstruiert, dass es möglichst einfach ist, eigene Algorithmen mit sehr wenig Code hinzuzufügen, wie man in Kapitel 3.5.2 sehen wird. In Abb.10 ist zu sehen, wie die Proposer mit den GUI-Elementen und den anderen bereits vorgestellten Elementen zusammenhängen.

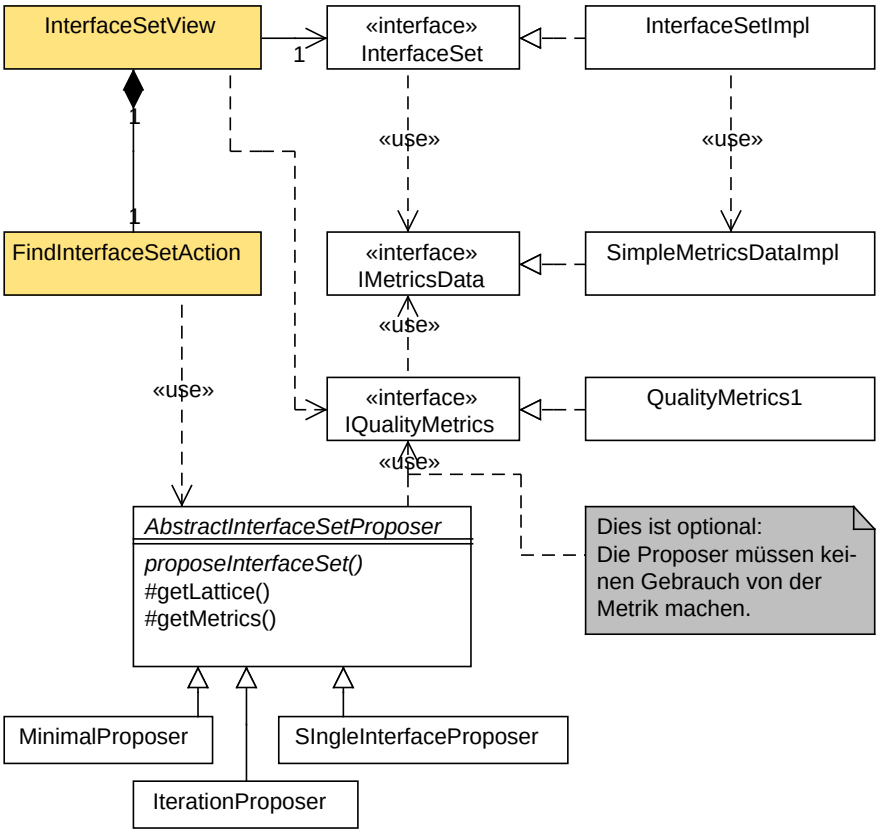


Abbildung 10: Die Einbindung der Proposer (GUI-Klassen farbig unterlegt).

3.5 Erweiterung der extension-points

Die Erweiterung der extension-points folgt dem in [27] vorgestellten Muster: Es gibt einen *deklarativen Teil* in der Konfigurationsdatei *plugin.xml*. In diesem wird eine Klasse spezifiziert, die ein bestimmtes Interface implementieren muss, oder von einer bestimmten abstrakten Klasse erben muss. Durch diesen *Implementations-Teil* wird das *Verhalten* der Erweiterung definiert. Damit es möglich ist, die entsprechenden Klassen oder Interfaces zu benutzen, müssen sie dem neuen Plugin bekannt sein, d.h. sich in seinem Classpath befinden. Dieser wird innerhalb des Eclipse Frameworks für jedes Plugin gesondert entsprechend den Angaben in seinen Konfigurationsdateien berechnet[32]. Die Datei *Manifest.mf* muss für eine Erweiterung des Interface-Designers mindestens folgenden Eintrag enthalten:

```
Require-Bundle: org.eclipse.core.runtime,
 org.intoJ.designer,
 org.intoJ.typeAccessAnalyzer
```

Wenn man nicht die Datei selbst editiert, sondern den speziellen Konfigurationseditor der Eclipse PDE benutzt, findet sich diese Information im Reiter *Dependencies*. Dieser muss mindestens die Einträge aus Abb.11 beinhalten.

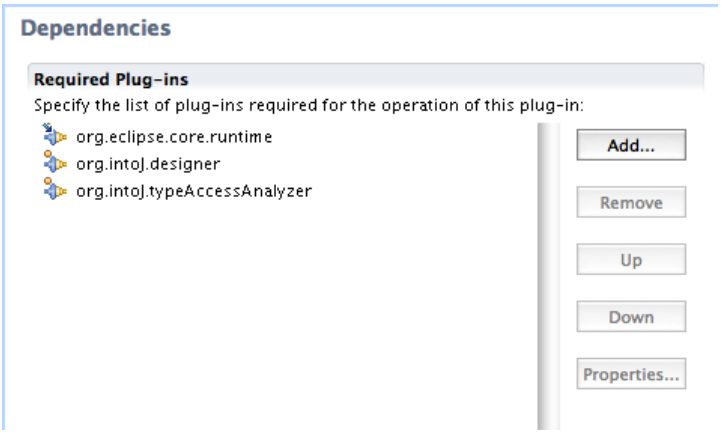


Abbildung 11: Der Reiter „Dependencies“ im Konfigurationseditor der Eclipse PDE.

3.5.1 Der *metrics* extension-point

Am einfachsten versteht man den Aufbau des Eintrags in der plugin.xml Datei an einem Beispiel. Hier ist der Eintrag für die bereitgestellte Metrik-Implementierung:

```
<extension
  point="org.intoJ.designer.qualityMetrics">
```

[32]Für eine genaue Beschreibung wird wieder auf [2] verwiesen.

```
  <metrics class="org.intoJ.designer.metrics.QualityMetrics1">
  </metrics>
</extension>
```

Die Deklaration nennt dabei den extension-point, der erweitert werden soll, in diesem Fall *org.intoJ.designer.qualityMetrics*. Das Attribut *class* ist der vollqualifizierte Name der Klasse, die die Metrik repräsentiert. Dazu muss sie das oben beschriebene Interface *IQualityMetrics* implementieren.

3.5.2 Der *proposer* extension-point

Analog dazu gestaltet sich die Erweiterung des extension-points für die Proposer:

```
<extension
  point="org.intoJ.designer.interfaceSetProposer">
  <proposer class="org.intoJ.designer.proposers.IterationProposer">
  </proposer>
</extension>
```

Die Proposer müssen dabei alle von der Klasse *AbstractInterfaceSetProposer* erben. Was dabei zu beachten ist, sieht man am besten an den bereits integrierten Beispielimplementierungen.

3.6 Die Beispielimplementierungen

Die bereitgestellten Implementierungen dienen zur Demonstration der Möglichkeiten, die die Erweiterungspunkte bieten, und sollen gleichzeitig demonstrieren, wie die Punkte zu nutzen sind.

3.6.1 Die bereitgestellte Metrik

Die bereitgestellte Metrik *QualityMetrics1* wird in [8] ausführlich hergeleitet. An Stelle der dort verwendeten Bezeichner wird hier der Einheitlichkeit halber die in [32] eingeführte Terminologie benutzt, wie auch schon in Kapitel 2.3.1 bei der Beschreibung des ACD-Wertes. Ausgangspunkt ist also wiederum T, die Menge aller Typen des Programms. $A \in T$ sei der als „Root Type“ bezeichnete Basistyp und $\{A\ a\}$ die Menge der mit ihm deklarierten Deklarationselemente. Wiederum bezeichnet $\iota(a)$ das *benötigte Protokoll* eines Deklarationselementes a. Außerdem sei S die Menge der für A neu einzuführenden Interfaces und N und M beliebige Elemente dieser Menge mit den Protokollen $\pi(N)$ und $\pi(M)$. All diese Typen sollen von A implementiert werden, es gilt also: $N \in S \Rightarrow A :< N$. Wir definieren dann $D(A, N, S)$ als die Menge der Deklarationselemente, die mit A deklariert sind, aber mit dem neuen Interface $N \in S$ umdeklariert werden sollen:

$$D(A, N, S) = \{A\ a | \pi(N) \supseteq \iota(a) \wedge \forall\ M \in S : \pi(M) \supseteq \iota(a) \Rightarrow |\pi(M)| \geq |\pi(N)|\}$$

Im Klartext: Um ein Deklarationselement dem Interface N zuzurechnen, muss N das benötigte Protokoll bieten, sonst würde das Programm syntaktisch unkorrekt. Außerdem soll es unter den neuen Interfaces in S kein anderes geben, was diese Forderung auch erfüllt, aber dabei mit weniger Methoden auskommt, also noch „besser passt“. Diese Beschreibung entspricht somit dem auf Seite 35 beschriebenen Algorithmus. Wir benutzen d für ein beliebiges Deklarationselement aus $D(A, N, S)$. Der Metrik-Wert für das Interface Set S für den Typ A ergibt sich dann nach [8] als:

$$v(A,S) = \sum_{N \in S} \left(\frac{|D(A,N,S)|}{|\{A\ a\}|} \cdot \sum_{d \in D(A,N,S)} \frac{|\iota(d)|}{|\pi(N)|} \right) \cdot \frac{1}{\sum_{a \in \{A\ a\}} \frac{|\iota(a)|}{|\pi(A)|}}$$

Das erste Summenzeichen iteriert dabei über alle Interfaces im Interface Set. In den großen Klammern repräsentiert der erste Faktor, also $\frac{|D(A,N,S)|}{|\{A\ a\}|}$ den Anteil der gesamten Deklarationselemente von A, die mit dem Interface N umdeklariert werden. Der zweite Faktor in den großen Klammern, also $\sum_{d \in D(A,N,S)} \frac{|\iota(d)|}{|\pi(N)|}$ berechnet für jedes dieser Deklarationselemente welchen Anteil der vom Interface N bereitgestellten Methoden es tatsächlich benötigt, und bildet die Summe dieser Werte. Der hinten angehängte Faktor $\frac{1}{\sum_{a \in \{A\ a\}} \frac{|\iota(a)|}{|\pi(A)|}}$ soll eine Art Normalisierung herbeiführen, indem er bewirkt, dass der Metrik-Wert für ein Interface Set, das nur aus dem impliziten Interface von A besteht immer eins ergibt.

Man kann folgende Eigenschaften der Metrik festhalten:

- Wenn ein Interface zum Umdeklarieren vieler Deklarationselemente geeignet ist, wird der erste Faktor größer.
- Wenn die umdeklarierten Deklarationselemente einen großen Anteil der Methoden des Interfaces tatsächlich benötigen, wird der zweite Faktor größer.

Die beiden Eigenschaften bilden einen Tradeoff, denn Interfaces, die an vielen Stellen benutzt werden können, werden in der Regel ein größeres Protokoll bereitstellen, dadurch wird der zweite Faktor kleiner.

Der Wert eines einzelnen Interfaces entspricht dem Sonderfall $S = \{N\}$. Diese Formeln wurden in der Klasse *QualityMetrics1* implementiert.

3.6.2 Die bereitgestellten Proposer

Es werden drei Proposer-Implementierungen mitgeliefert. Die erste macht dabei Gebrauch von der aktuell eingestellten Metrik, wenn keine eigenen Implementierungen hinzugefügt wurden, ist dies zunächst immer die im vorigen Abschnitt beschriebene. Die anderen beiden generieren das Interface Set allein auf Grund der Analysedaten des Type Access Analyzer, indem sie aus diesen triviale Sonderfälle von Interface Sets ableiten.

Iterative Interface Set Proposer Die Aufgabe dieses Proposers ist es, über alle möglichen Interface Sets zu iterieren, und das Interface Set vorzuschlagen, das entsprechend der eingestellten Metrik den höchsten Wert liefert. Dazu werden einige Annahmen gemacht: Es wird zunächst davon ausgegangen, dass man die Aufgabe darauf zurückführen kann, über alle möglichen Kombinationen von Access Sets[33] zu iterieren. Es wird weiter vorausgesetzt, dass in einem Interface Set ein Access Set immer nur zu einem Interface gehören soll. Wenn man die Menge der Access Sets betrachtet, ergibt sich dann die Menge aller möglichen Aufteilungen in Untermengen, also die Menge aller möglichen *Mengenpartitionen* als gleichbedeutend mit der Menge aller Interface Sets. Hinzu kommt, dass nicht alle Access Sets in einem Interface Set benutzt werden müssen. Also müssen auch die Mengenpartitionen aller Untermengen betrachtet werden. Zusammengenommen wird also über die Menge aller Mengenpartitionen aller Untermengen der Menge der Access Sets iteriert.

Es ist naheliegend, dass diese Zahl sehr schnell sehr groß wird. Die Zahl aller möglichen Mengenpartitionen entspricht nach [43] der *Bellschen Zahl*[34], die ist für eine Menge der Größe n:

$$B_n = \frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}$$

Dies ist zu multiplizieren mit der Zahl der Untermengen, die für eine n-elementige Menge bekanntlich 2^n ist. Es ergibt sich als Anzahl der Iteratorschritte folgender Wert:

$$StepsFor(n) = \frac{2^n}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}$$

Man beachte, dass n von der Anzahl der Access Sets abhängt, *nicht* von der Größe des Protokolls. Allerdings kommen durchaus auch viele Typen mit etlichen Access Sets in normalen Projekten vor, so dass es offensichtlich nicht immer möglich ist, diese Iteration komplett zu durchlaufen. Der Iterative Interface Set Proposer (IISP) geht daher folgendermaßen vor:

1. Zunächst wird berechnet wie viele Iterationen nötig wären.
2. Dann wird mit einer Proberechnung festgestellt, wie lange der aktuelle Computer dafür brauchen würde.
3. Ist die Zeit länger als 10 Sekunden, wird ein Dialog angezeigt, der dem Benutzer die Möglichkeit gibt, die Anzahl der einbezogenen Access Sets zu reduzieren (siehe Abb.12). Dabei wird eine Abschätzung angezeigt, wie lange die Berechnung dauern würde.

[33] Hier wieder im definierten Sinn als *benutzte* Subprotokolle.
[34] Diese und die anderen Formeln wurden in der Klasse *Util* implementiert.

4. Hat der Benutzer eine Wahl, sagen wir x, getroffen, werden die Access Sets nach folgendem System reduziert:
   a) Es wird jedes Access Set als einzelnes Interface betrachtet und der Metrik-Wert berechnet.
   b) Nur die x Access Sets mit den höchsten Werten werden benutzt.

Abbildung 12: Der Dialog zur Reduzierung der verwendeten Access Sets.

Dieses Verfahren erwies sich als unpraktikabel, wenn ein ganzes Projekt analysiert werden soll, weil dieser langwierige Vorgang ja ohne Benutzereingaben, in der Art eines Batch-Jobs ablaufen soll. Deshalb wird in diesem Fall *vor* dem Beginn der Analyse eine Zeitgrenze abgefragt, wie lange die Iteration höchstens dauern soll. Der IISP verwendet dann immer nur so viele Access Sets, dass diese Grenze nicht überschritten wird. Um die richtige Anzahl zu ermitteln, führt er eine Testberechnung durch, misst dabei die Zeit, und berechnet daraus, mit wie vielen Access er unter der Zeitschranke bleiben kann. Das Ergebnis erwies sich dabei als wesentlich zuverlässiger, wenn die Messung *zweimal* durchgeführt wird. Das ist vermutlich darauf zurückzuführen, dass die JVM Optimierungen vornimmt, sobald sie erkennt, dass bestimmte Codeteile sehr oft ausgeführt werden. Dadurch ist die Ausführungsgeschwindigkeit beim zweiten Mal in der Regel kürzer und näher an dem Wert, der für die tatsächliche Berechnung anfällt. Das Ergebnis ist durch dieses Verfahren indeterministisch, weil die Anzahl der einbezogenen Access Sets vom Laufzeitverhalten abhängt. Falls man ein deterministisches Ergebnis benötigen würde, könnte man natürlich eine fixe Höchstanzahl von Iteratorschritten anstatt der Zeitgrenze vorgeben.

Die Berechnung der Iteration wird aufgespalten in die Berechnung der Untermengen in der Klasse *SubSetIterator* und die Berechnung der Partitionen in der Klasse *SetPartitionsIterator*. Die Klasse *SetPartitionsOfSubsetsIterator* kombiniert dann je einen dieser Iteratoren, um das gewünschte Ergebnis zu liefern. Bei der Berechnung der Partitionen drängt sich zunächst eine rekursive Lösung auf, diese hat aber einen enormen Speicherverbrauch, deshalb wurde der in [25] vorgestellte Algorithmus benutzt. Er arbeitet mit konstantem Speicherverbrauch.

Minimal Interfaces Proposer Dieser Proposer schlägt ein Interface Set vor, das für jedes gefundene Access Set ein Interface einführt. In dem Spektrum aller *möglichen* Interface Sets ist dies also *das mit den meisten neuen Typen*, was aber auch die optimale Entkopplung bietet. Konzeptionell ist der Effekt vergleichbar mit einer Anwendung von Infer Type auf alle Deklarationselemente des Basistyps. In der Praxis ergeben sich jedoch Unterschiede durch die verschiedenen Verfahren zur Typinferenz, und die Tatsache, dass Infer Type auch bestehende Typen mitbenutzt, was der Interface Designer generell nicht tut. Dennoch ist es aber interessant, diesen Infer Type-ähnlichen Proposer zur Verfügung zu haben. Man kann ihn etwa als Vergleich und Maßstab für eigene Metriken benutzen, indem man ermittelt, wie stark eine eigene Metrik die Anzahl der neu einzuführenden Typen im Vergleich zu den maximal speziellen Interfaces, die der Minimal Interfaces Proposer (MIP) vorschlägt, verringert.

Am Beispiel dieses recht simplen Proposers kann man gut sehen, wie man einen Proposer entwirft. Der Eintrag in der plugin.xml sieht so aus:

```
<extension
  point="org.intoJ.designer.interfaceSetProposer">
  <proposer class="org.intoJ.designer.proposers.MinimalProposer">
  </proposer>
</extension>
```

Er ist also ähnlich simpel wie bei den Metriken. Hier der Quelltext der Klasse *MinimalProposer*:

```
public class MinimalProposer
    extends
    AbstractInterfaceSetProposer {

@Override
public String getName() {
  return "Minimal Interfaces Proposer";
}

@Override
public String getDescription() {
  return ".."; // längere Beschreibung,
               // die in den Preferences angezeigt wird
}

@Override
public void proposeInterfaceSet(
    IProgressMonitor monitor) {
  TAAccessSetLattice lattice = getLattice();
  if (lattice == null)
    return;
```

```
  Collection<Set<TAAccessSet>> result =
    new ArrayList<Set<TAAccessSet>>();

  Collection<TAAccessSet> accessSets = getAccessSets();
  monitor
      .beginTask(getName() + " processing "
          + lattice.getRootType().toString(), accessSets
          .size());

  for (TAAccessSet as : accessSets) {
    HashSet<TAAccessSet> s = new HashSet<TAAccessSet>();
    s.add(as);
    result.add(s);
    monitor.worked(1);
  }
  setResult(result);
  monitor.done();
}
}
```

Die Methoden *getName()* und *getDescription()* erfüllen die analoge Funktion wie die gleichnamigen Methoden bei den Metriken. Die eigentliche Arbeit wird in der Methode *proposeInterfaceSets* getan. Man beachte, dass das Ergebnis, entsprechend der Erklärung in Kapitel 3.2.2 vom Typ *Collection<Set<TAAccessSet>>* nicht etwa als Rückgabewert übergeben wird, sondern mit der Methode *setResult()* gespeichert wird. Das liegt daran, dass auch die Nebenläufigkeit bereits in der abstrakten Superklasse gekapselt ist und der Zugriff auf das Ergebnis von einem anderen Thread aus erfolgt.

An dem Beispiel kann man auch sehen, dass auf die benötigten Objekte, also den Lattice und die Access Sets direkt durch Convenience-Methoden des Supertyps zugegriffen werden kann. Solche Methoden gibt es auch für den Zugriff auf die eingestellte Metrik und zum Erzeugen eines *InterfaceSet*. Das dient dazu, dass man als Programmierer die relevanten Objekte leichter „entdecken“ kann, ohne sich erst mit der gesamten Klassenhierarchie beschäftigen zu müssen. Was die weiteren Feinheiten beim Erstellen von Proposern betrifft, sei auf die Quelltextkommentare der Klassen *AbstractInterfaceSetProposer* und *IterationProposer* verwiesen.

Single Interface Proposer Dieser Proposer ist gewissermaßen das Gegenstück zum Minimal Interfaces Proposer, denn er schlägt immer ein Interface Set vor, das aus einem einzigen Interface besteht. Dieses ergibt sich aus der Kombination aller Access Sets und enthält daher alle Methoden, die auf irgendeinem Deklarationselement benutzt wurden. Es stellt somit das speziellste Interface dar, was alle Access Sets enthält. Gemeinsam mit dem

MIP spannt der Proposer so gewissermaßen einen Bereich für mögliche Interface Sets auf: Von einem Interface, was die gesamte tatsächliche Typnutzung abdeckt, bis zu den vielen, speziellen Interfaces des Minimal Interface Set Proposers.

3.7 Dokumentation

Die Dokumentation des Interface-Designers wurde mit den entsprechenden Erweiterungspunkten in die Eclipse-Hilfe eingefügt. Von ihr kann auch direkt auf die API-Dokumentation (inklusiver verlinkter Quelltexte) zugegriffen werden, die per Javadoc erstellt wurde. Innerhalb dieser Arbeit findet sich zusätzlich eine Anleitung als Anhang D.

3.8 Deployment

Das Plugin *org.intoJ.designer* wurde zusammen mit den benötigten Plugins der IntoJ Suite in einem Feature zusammengefasst. Ein weiteres Feature beinhaltet das Plugin *org.intoJ.feedrefactoring*, was die Verknüpfung mit dem Extract Interface Refactoring bereitstellt. Die Features können direkt von einer Update-Site installiert werden:

`http://www.frankfiedler.de/id/updatesite`

Ein Kopie dieser Update-Site befindet sich auch auf der beiliegenden CD. Da die grafischen Elemente das GEF-Feature benötigen, muss dies vorher von der Eclipse Update Site installiert werden. Eine Schritt-für-Schritt Anleitung hierzu befindet sich auf der IntoJ Homepage [18]. Man beachte, dass der Interface-Designer entsprechend den Erläuterungen in Kapitel 3.2.1 im Moment nicht gleichzeitig mit der IntoJ Suite installiert sein darf. Durch das komplexe Configurations-Management von Eclipse kann es auch zu Problemen kommen, wenn die IntoJ Suite oder Teile von ihr oder ältere Versionen des Interface Designers irgendwann einmal installiert waren. Auch der „Configuration Manager“ von Eclipse ist da nicht von Nutzen. Um auf Nummer sicher zu gehen, empfiehlt es sich, eine frische Version von Eclipse zu entpacken und darin GEF und den Interface Designer zu installieren. Um dies zu vereinfachen, ist für Windows bereits eine Eclipse Version mit vorinstalliertem Interface Designer auf der CD, zu finden im Verzeichnis *win/eclipse_interface_designer*. Diese braucht man nur auf die Festplatte zu kopieren und zu starten, es muss also nichts mehr installiert werden. Basis für Entwicklung und Test war Eclipse 3.2.1.

4 Fallbeispiele

Grundsätzlich ergibt sich bei der Suche nach Fallbeispielen das Problem, das auch schon Martin Fowler in [11](S.1) bei der Suche nach Refactoring-Fallbeispielen beschreibt: Ein aussagekräftiges Beispiel ist zu lang und ermüdend für den Leser. Ist das Beispiel aber kurz, ist der Vorteil durch das vorgestellte Verfahren nicht erkennbar. Bei dem Thema dieser Arbeit tritt das besonders stark zu Tage, denn Analysen zur Typnutzung und insbesondere Metriken entfalten ihre Aussagekraft und ihren potentiellen Mehrwert eigentlich erst bei einer gewissen Größe und Unübersichtlichkeit des Projekts, ansonsten gibt ein kurzer Blick in den Code mehr Aufschluß.

Außerdem versteht sich der Interface-Designer als erweiterbares Framework, das insbesondere einen agilen Prozess der Softwareerstellung unterstützen soll. Ein „echtes“ Fallbeispiel müsste also so aussehen, dass man das Framework tatsächlich erweitert. Dann müsste getestet werden, ob sich Vorteile ergeben, wenn man das Framework verwendet, man könnte dazu etwa zwei Programmierteams an dem selben Programm arbeiten lassen und feststellen, ob durch die Verwendung des Interface-Designers Zeit gespart wurde, oder ob bessere Resultate erzielt wurden. Das ist natürlich jenseits der Möglichkeiten dieser Arbeit. Außerdem sind wegen der in Kap.5.2.1 behandelten Probleme davon im Moment keine aussagekräftigen Ergebnisse zu erwarten.

Aus diesen Gründen begnügen sich die Beispiele damit, das Verhalten der mitgelieferten Implementierungen unter verschiedenen Aspekten zu beleuchten. Dabei soll das Potential und die Grenzen der momentanen Implementierungen sichtbar werden. In einer Fallstudie zur Anwendung auf ganze Projekte wird demonstriert, wie man die integrierten Tools benutzen kann, um einen globalen Eindruck von den Auswirkungen der automatischen Refactoring-Vorschlagsfunktion zu bekommen. Auch unter dem Vorbehalt der nicht verlässlichen Ergebnisse ist dies nicht uninteressant, allerdings müssen die Tests wiederholt werden, sobald die in Arbeit befindlichen neuen Analysewerkzeuge bereitstehen.

4.1 Fallbeispiel 1: Die Klasse *Actor*, Entkopplung durch einen neuen Typ

Als erstes Beispiel wird die Beispielklasse aus [8] in etwas abgewandelter Form[35] aufgegriffen. An diesem Beispiel kann man sich einen ersten Eindruck von der Wirkungsweise der Metrik machen und insbesondere sehen, wie durch

[35] In [8] unterläuft dem Autor ein Fehler bei der Beschreibung der Deklarationselemente: Es heißt dort, dass die Elemente in D_1 entweder *deleteRole(..)* oder *addRole(..)* benutzen, später ist aber davon die Rede, dass per Typinferenz a la INFER TYPE ein *einziges* minimales Interface für diese vier Deklarationselemente gefunden wird. Es müssten aber zwei sein, weil zwei verschiedene benutzte Methoden ja zwei unterschiedliche Access Sets bedeuten. In unserem Beispiel benutzen daher die Deklarationselemente in D_1 alle nur *deleteRole(..)*.

sie die Anzahl der neu einzuführenden Typen verringert wird.

Es gibt also die Klasse *Actor*:

```
public class Actor {
  private Set roles;
  public void addRole(Role role) {
    roles.add(role);
  }
  public void deleteRole(Role role) {
    roles.remove(role);
  }
  public Set getRoles() {
    return Collections.unmodifiableSet(roles);
  }
  public void doSomethingUnrelated1() {};
  public void doSomethingUnrelated2() {};
  public void doSomethingUnrelated3() {};
  public void doSomethingUnrelated4() {};
}
```

Die Klasse steht, wie man sich denken kann für einen Schauspieler, der verschiedene Rollen annehmen und ablegen kann. Neben den ersten drei Methoden, die mit dieser Fähigkeit zu tun haben, hat er vier weitere Methoden die der Einfachheit halber nur als *doSomethingUnrelated_x()* aufgeführt werden.

Da sich die vorzunehmende Analyse auf die *Nutzung* stützt, wird eine weitere Klasse gebraucht, die die Klasse *Actor* benutzt. Dazu wird die Klasse *ActorUser* eingeführt, die hier nur teilweise gezeigt wird:

```
public class ActorUser {
  private Role r=new Role(){};
  void m1() {
    Actor de1 = new Actor();
    de1.deleteRole(r);
  }
  void m2() {
    Actor de2 = new Actor();
    de2.deleteRole(r);
  }
  ...
}
```

In ihr finden sich insgesamt acht Deklarationselement vom Typ *Actor*. Davon benutzen vier die Methode *deleteRole(Role)* und zwei die Methoden *deleteRole(Role)* und *getRoles()*. Die verbleibenden zwei benutzen *addRole(Role)* und *getRoles()*.

Man hat also:

- Eine Klasse *Actor* mit einem Protokoll, das aus *sieben* Methoden besteht.
- *Acht* mit *Actor* deklarierte Deklarationselemente.
- *Drei* verschiedene Access Sets: Eines mit einer Methode, zwei mit je zwei Methoden.
- Von den acht Deklarationselementen benutzen *vier* das Access Set mit einer Methode, und je *zwei* eines der Access Sets mit zwei Methoden.

Das Interesse gilt nun den Metrikdaten für verschiedene denkbare Interfacemengen. In [8] wird deren Gewinnung anhand der in Kap. 3.6.1 erwähnten Formel genau aufgeschlüsselt, hier wird statt dessen der Interface-Designer benutzt um das Ergebnis zu ermitteln.

Der Aufruf der Analyse[36] auf dem Typ *Actor* schaltet um in die *Interface-Designer* Perspektive, die die Nutzung der Klasse wie in Abb.13 gezeigt darstellt. Das genaue Zusammenspiel der Views ist in Anhang D beschrieben.

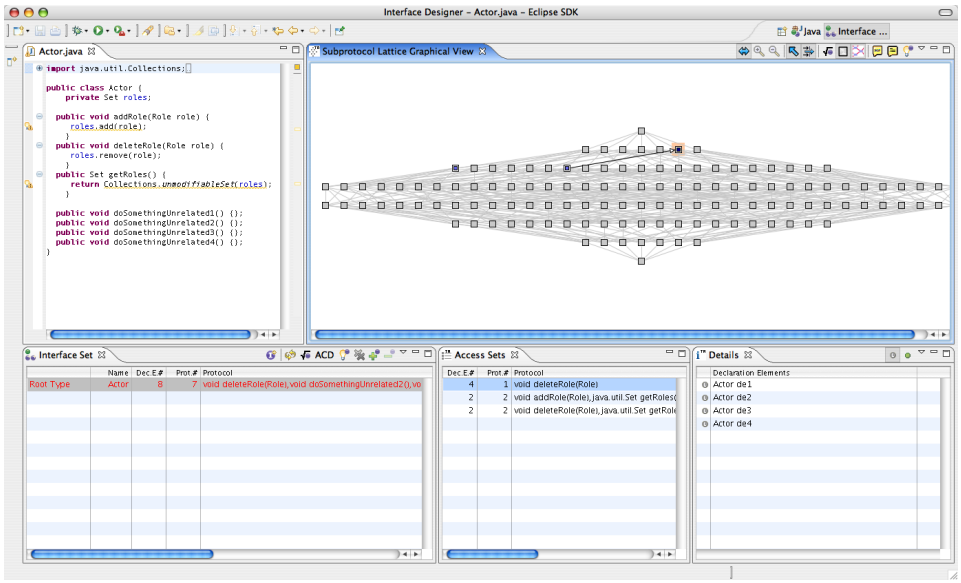


Abbildung 13: Die Klasse *Actor* in der *Interface-Designer* Perspektive

Von den verschiedenen Möglichkeiten, Interfaces zu entwerfen, soll nun die Vorschlagsautomatik getestet werden. Dabei sollen, um eine Basis für den Vergleich zu haben, zunächst *möglichst kontextspezifische* Interfaces für jedes der drei Access Sets erzeugt werden. Dazu wird in den Preferences der *Minimal Interfaces Proposer*[37] ausgewählt. Dann ergibt ein Klick auf das Glühbirnensymbol im *Interface Set View* das Interface Set aus Abb.14. Die Metrik kann durch einen Druck auf das Wurzelsymbol eingeblendet werden (in der Abbildung ist die Metrik noch nicht eingblendet). Wie erwartet, sieht

[36] Siehe dazu auch die Anleitung in Anhang D.
[37] Das Verhalten des MIP wird beschrieben in Kap.3.6.2 auf Seite 47.

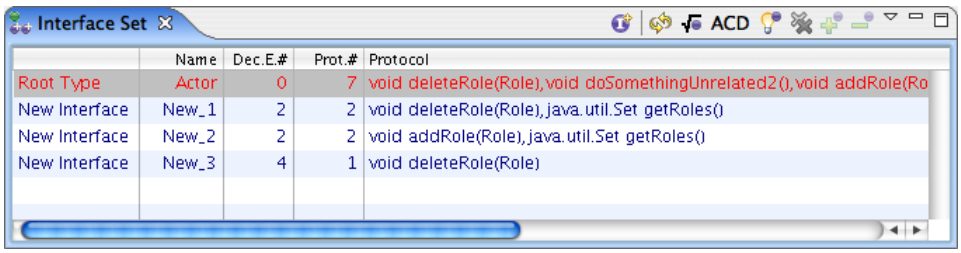


Abbildung 14: Ein Interface Set für *Actor*, erzeugt mit dem MIP.

man dort den auch in [8] errechneten Wert 1.75. Nun wird in den Preferences der *Iterative Interface Set Proposer* als Interface Set Proposer eingestellt. Durch einen erneuten Klick auf das Glühbirnensymbol erhält man das Interface Set aus Abb. 15. Die Metrikanzeige zeigt den korrekten Wert 2.33. Bevor

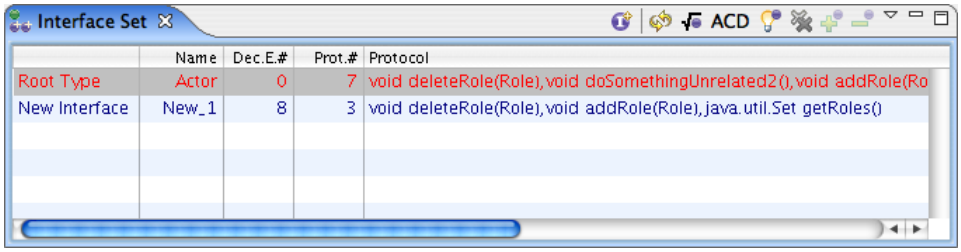


Abbildung 15: Ein Interface Set für *Actor*, erzeugt mit dem IISP.

das Interface nun eingeführt wird, sollte man ihm zunächst einen aussagekräftigen Namen geben. Dazu kann man auf das Interface im *Interface Set* View doppelklicken und in dem folgenden Dialog den Namen *Roleplayer* eingeben (kein Screenshot). Dann erscheint der Typ mit seinem neuen Namen im *Interface Set* View. Um ihn wirklich zu erzeugen, klickt man schließlich auf den „Create Interfaces“ Button. Dies bringt den Zwischendialog aus Abb.16. Hier

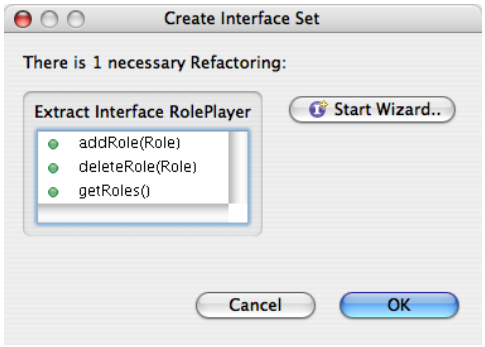


Abbildung 16: Der Zwischendialog zur Erzeugung des Interface Sets für *Actor*.

startet nun eine Druck auf *Start Wizard.*. das Eclipse Refactoring, in dem die ensprechenden Methoden bereits vorselektiert sind (Abb.17). Zum Abschluss

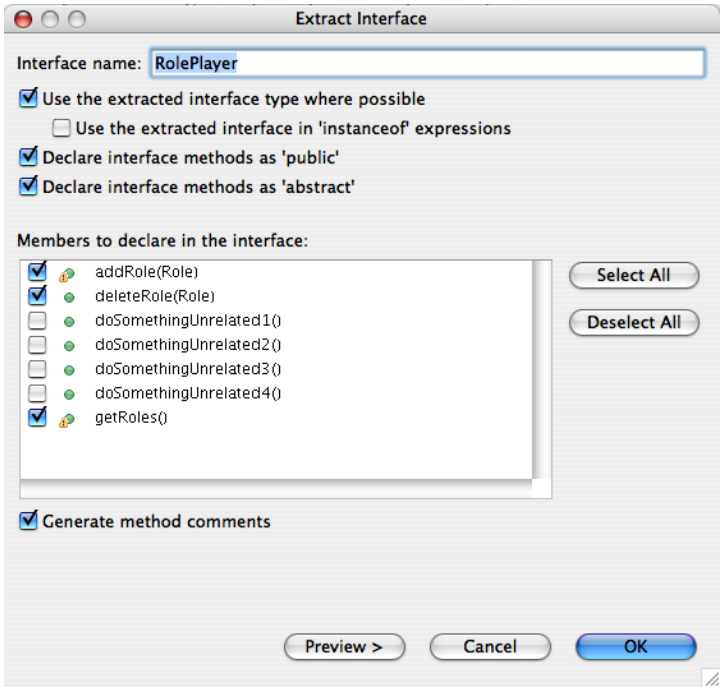


Abbildung 17: Das Eclipse Refactoring "Extract Interface.." für die Klasse *Actor*.

kann man einen kurzen Blick auf die Änderungen im Quelltext werfen, die durch das Refactoring entstanden sind. Eingeführt wurde das neue Interface *RolePlayer*:

```
public interface RolePlayer {
  public abstract void addRole(Role role);
  public abstract void deleteRole(Role role);
  public abstract Set getRoles();
}
```

Dises wird nun von *Actor* implementiert:

```
public class Actor implements RolePlayer {
  private Set roles;
  public void addRole(Role role) {
    roles.add(role);
  }
  ...
}
```

Des weiteren sind die Deklarationselemente in *ActorUser* nun mit dem neuen Interface statt mit der Klasse *Actor* deklariert:

```
public class ActorUser {
  private Role r=new Role(){};
```

```
  void m1() {
    RolePlayer de1 = new Actor();
    de1.deleteRole(r);
  }
  void m2() {
    RolePlayer de2 = new Actor();
    de2.deleteRole(r);
  }
  ...
}
```

Ergebnis Man wäre also nun an dem Punkt, wo man weitere Klassen das Interface *RolePlayer* implementieren lassen können. Diese können dann ebenfalls von der Klasse *ActorUser* benutzt werden, die dann sinnvollerweise auch einen neuen, allgemeineren Namen erhalten kann. Die Klasse *ActorUser* ist somit von der Klasse *Actor* entkoppelt worden. Die Flexibilität des Codes wurde erhöht. Weiterhin wurde gezeigt, dass der IIPS in Verbindung mit der Metrik in diesem Fall in der Lage war, auf der Basis der tasächlichen Typnutzung die verschiedenen Access Sets in plausibler Weise zusammenzufassen.

Abkürzung, wenn nur ein Interface erzeugt werden soll Wenn man bereits entschieden hat, den Typ *Actor* nur durch ein einziges Interface zu entkoppeln, kann man den Prozess verkürzen, indem man den Typ wie anfangs markiert, und dann direkt im *Refactor* Menü den Punkt *Extract Interface by Usage* auswählt. Dies bringt direkt den Refactoring Dialog aus Abb.17. Der Nachteil ist, dass man so nicht erkennt, ob man mit mehreren Interfaces evtl. ein Interface Set mit einer höheren Bewertung hätte erzielen können. Es ist aber möglich, dass der Programmierer bereits entschieden hat, dass mehrere Interfaces nicht in Frage kommen.

4.2 Fallbeispiel 2: Entkopplung mit mehreren Typen, Auswirkung der Metrik

Nun soll ein etwas komplexeres Beispiel betrachtet werden, um zu testen, wie sich die Metrik in Fällen verhält, in denen mehrere neue Typen eingeführt werden sollen. Zunächst wird die Klasse *Person* betrachtet:

```
import java.util.Collection;
public class Person {

    private Collection qualifications;
    private Collection bookList;
    private Collection portfolio;
    private int salary;
```

```
    public boolean hasQualification(int x) {
        return qualifications.contains(x);
    }
    public int getSalary(){return salary;}
    public void setSalary(int salary) {
        // ...
    };

    public void borrowBook (int bookNr){
        // ...
    };
    public Collection getBorrowedBooks(){return bookList;};

    public boolean ownsStock(int wkn){return portfolio.contains(wkn);}

}
```

Die Methoden erwecken den Eindruck, eine Person in verschiedenen Rollen zu repräsentieren. Z.B. könnte man die ersten drei Methoden einer Rolle als Angestellter einer Firma zuordnen.

Als menschlicher Betrachter schließt man dies aus den *Namen* der Methoden und evt. aus der *Semantik ihrer Implementierungen*, die hier nur angedeutet ist. Die automatische Analyse mit dem Interface-Designer ist natürlich darauf angewiesen, die *Nutzung* zu analysieren, deswegen muss man Klassen hinzufügen, die den Typ *Person* auf verschiedene Weise benutzen.

Zum einen die Klasse *Company*:

```
public class Company {
  public void hire(Person p_comp1){
    if (p_comp1.hasQualification(37)){
      ...
    }
    p_comp1.setSalary(2500);
  }
  private void computeSalary(Person p_comp2) {
    if (p_comp2.hasQualification(17)){
      p_comp2.setSalary(3000);
    }
  }
  private void raiseSalary(Person comp3){
    comp3.setSalary(comp3.getSalary()+100);
  }
  private int getSalaryOf(Person p_comp4){
    return p_comp4.getSalary();
  }
}
```

Die Klasse *BookLibrary*:

```
public class BookLibrary {
  public void handleBookRequest(Person p_lib, int nr){
    if (p_lib.getBorrowedBooks().size()<5){
      p_lib.borrowBook(nr);
    }
  }
}
```

Die Klasse *NewsSite*:

```
public class NewsSite {
  Person actualVisitor;
  public String getStartPage(){
    StringBuilder result=new StringBuilder();
    int wkn=123456;
    ...
    if (actualVisitor.ownsStock(wkn)){
      result.append("Price: "+getMarketPriceFor(wkn));
    }
    return result.toString();
  }
  private String getMarketPriceFor(int wkn) {
    String result="";
    ...
    return result;
  }
}
```

Und die Klasse *Bank*:

```
public class Bank {
  void sellStuff() {
    Person p_bank=getPersonFromSomewhere();
    if (true){
      if (p_bank.ownsStock(123123)){..}
    }
  }

  private Person getPersonFromSomewhere() {..}
}
```

Wie man sieht, benutzen diese Klassen den Typ *Person auf unterschiedliche Weise*. Die Deklarationselemente sind so benannt, dass man sie auf Grund ihres Namens bereits einer Klasse zuordnen kann, das ist etwas einfacher, als wenn alle nur p heißen.

Nun wird die Analyse für die Klasse *Person* gestartet, man erhält den Subprotokoll-Graph von Abb.18. Wie im vorigen Beispiel, kann man Aufschluss über die Methoden und Deklarationselemente der einzelnen Knoten über den

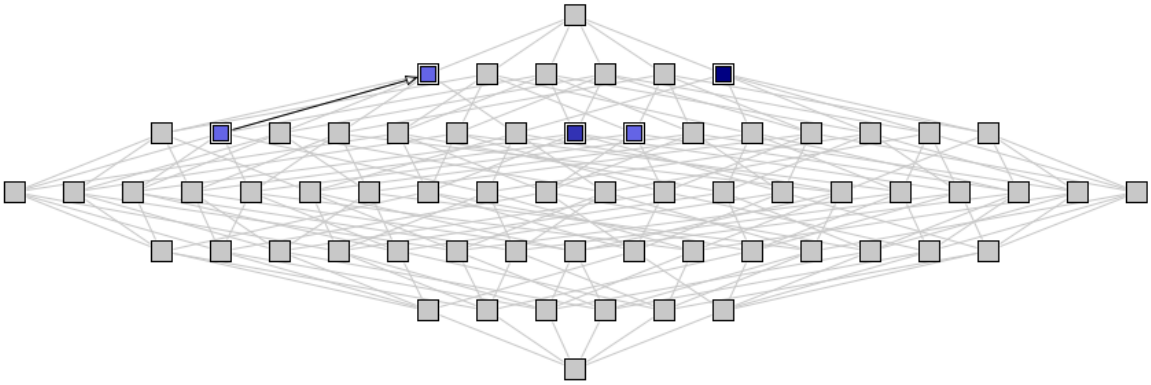


Abbildung 18: Die Klasse *Person* und ihre Benutzung durch das Beispielprogramm im *Subprotocol Lattice Graphical View*.

Details View erhalten: Markiert man zum Beispiel den oberen rechten Knoten, und klickt im *Details* View auf das Methoden-Symbol, erscheinen dort die Methoden dieses Access Sets.(Abb.19)

Klickt man im *Details* View auf das kleine „L“, sieht man dort die Deklarationselemente, die dieses Access Set teilen.(Abb.20) Es soll jetzt wieder die

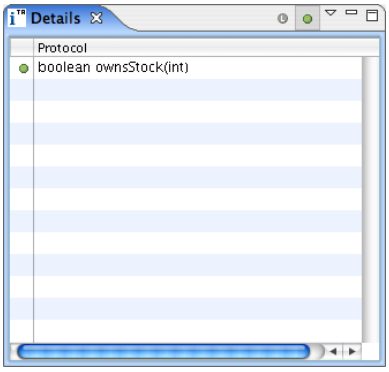


Abbildung 19: Der *Details* View in der Methodenansicht

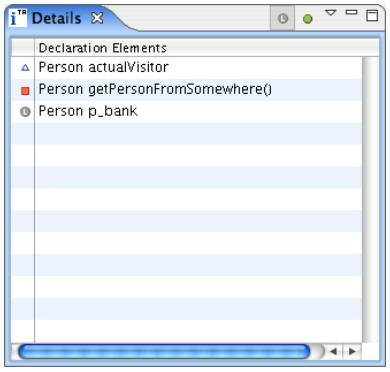


Abbildung 20: Der *Details* View in der Deklarationselemente-Ansicht

Vorschlagsfunktion getestet werden. Wieder wird, wie im vorigen Beispiel, zunächst der *Minimal Interfaces Proposer* eingestellt. Man erhält fünf neue Typen, denn es gibt ja fünf Access Sets.(Abb.21) Fünf neue Typen sind *viel*, wenn man bedenkt, dass das Beispiel nur aus fünf Klassen besteht! Nachdem man in den *Preferences* den *Iterative Interface Set Proposer* (IISP) eingestellt hat, liefert die Vorschlagsfunktion nur noch 3 neue Typen.(Abb.22)

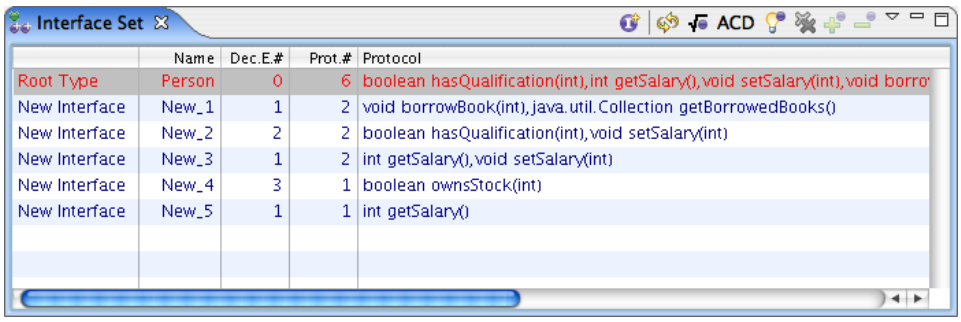


Abbildung 21: Das vom MIP ermittelte Interface Set für *Person*

Diese sollen etwas näher betrachtet werden, um zu sehen, wie dies zustande

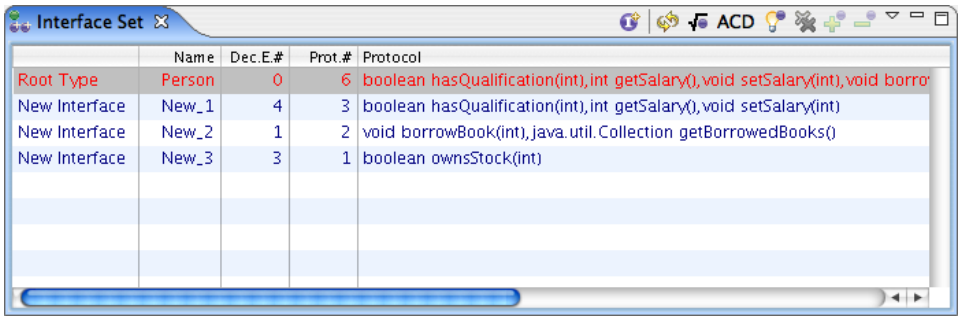


Abbildung 22: Das vom IISP ermittelte Interface Set für *Person*

gekommen ist. Dazu wird das erste Interface im Interface Set View markiert.

Die Access Sets, aus denen es zusammengesetzt ist, werden automatisch im *Access Sets* View und *Subprotocol Lattice Graphical* View markiert (siehe Abb.23). Man kann erkennen, dass die folgenden überlappenden Access Sets zu einem einzigen zusammengefasst wurden:

1. *hasQualification(), getSalary()*
2. *getSalary(), setSalary()*
3. *getSalary()*

Nun soll überprüft werden, wie sich der Prozess der Namensgebung bei den verschiedenen Interface Sets gestalten würde. Um die Interfaces zu benennen, kann man auf sie im Interface Set View doppelklicken oder den Namen erst im Extract Interface Refactoring angeben. Um eine Übersicht zu erhalten, kann man aber einfach auf den *Create Interfaces* Button im Interface Set View drücken. Dann kann man die Zwischendialoge für die vom MIP und vom IISP vorgeschlagenen Interface Sets vergleichen (Abb.24 und Abb.25). Wie man sieht, ist es in dem mit der Metrik gefundenen Interface Set wesentlich

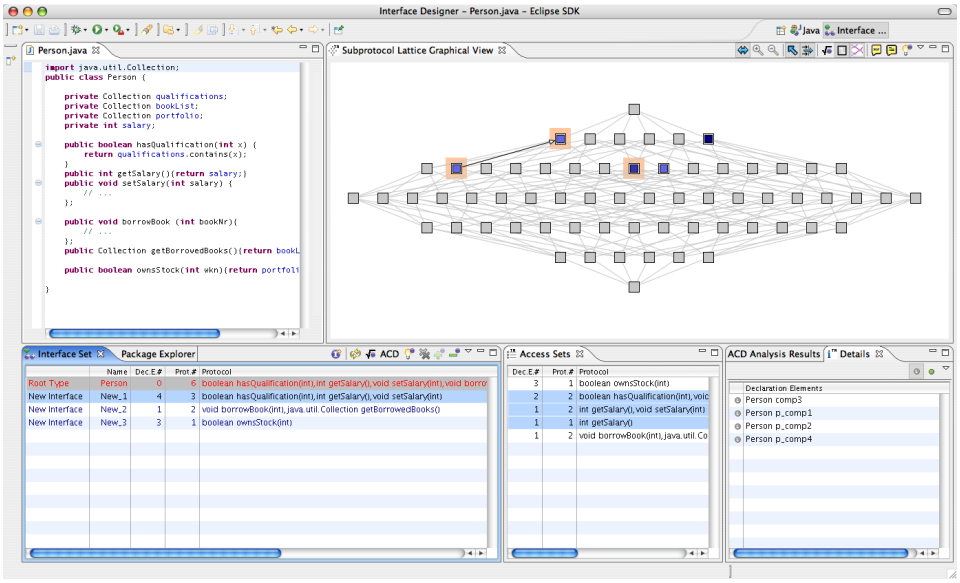


Abbildung 23: Die Zusammensetzung eines Interfaces im Tabular View sichtbar gemacht.

einfacher, sinnvolle Namen für die Interfaces zu finden; anbieten würde sich etwa: *Employee*, *BookLender* und *StockOwner*.

Die weiteren Schritte, also das Benennen und das Einleiten des Refactorings, verlaufen analog zum ersten Beispiel.

Es soll noch ein weiteres Experiment vorgenommen werden: Bisher ist das Beispiel ja quasi maßgeschneidert, so dass sich eine Aufteilung in die drei getrennten Funktionalitäten zwingend ergibt. Was passiert aber nun, wenn auf einem Deklarationselement plötzlich Teile des Protokolls benötigt werden, die nach der jetzigen Aufteilung zu einem anderen Funktionsbereich gehören? Dazu lässt man die Klasse *Bank*, die bisher brav nur die Methode *ownsStock(..)* aufgerufen hat, zusätzlich noch die Methode *getSalary()* aufrufen:

```
public class Bank {
  void sellStuff() {
    Person p_bank=getPersonFromSomewhere();
    // now we use a second method on this declaration element:
    if (p_bank.getSalary()>3000){
      if (p_bank.ownsStock(123123)){..}
    }
  }
  private Person getPersonFromSomewhere() {..}
}
```

Diese Methode gehörte nach dem zuvor berechneten Interface Set zum Interface *Employee*, auf das die Klasse *Bank* ja keinen Zugriff hätte, da ihr De-

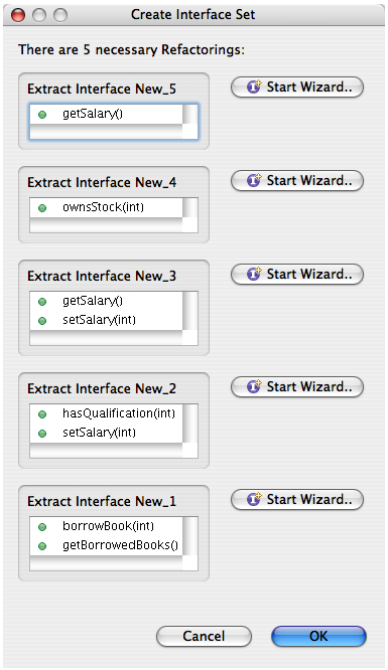


Abbildung 24: Der Create Dialog des vom MIP vorgeschlagenen Interface Sets

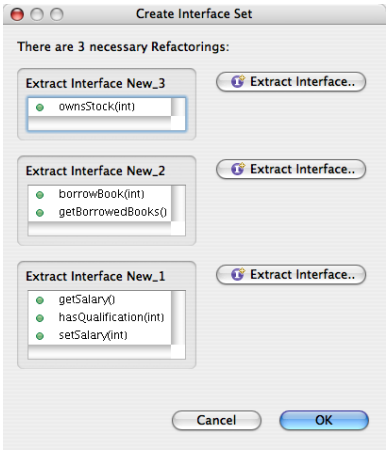


Abbildung 25: Der Create Dialog des vom IISP vorgeschlagenen Interface Sets

klarationselement mit dem Interface *Stockowner* umdeklariert werden sollte.

Nachdem nun die Klasse verändert wurde, lässt man zunächst die Analyse mit einem Klick auf den *Refresh* Button neu berechnen. Dann wird die Vorschlagsfunktion gestartet und man erhält das Interface Set aus Abb.26. Man

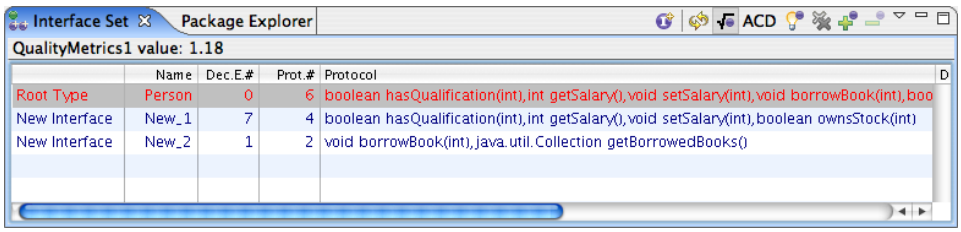


Abbildung 26: Das Interface Set der Klasse *Person* nach der Veränderung des Zugriffs

erhält ein Interface Set, das nur noch aus zwei neuen Interfaces besteht. Was ist geschehen?

Wenn man sich die Veränderung genauer anschaut, sieht man, dass die Methode *ownsStock(..)*, die vorher im Interface *StockOwner* gelandet ist, nun mit den Methoden von *Employee* zusammen ein Interface bildet.(In Abb.26 heißt das Interface „New_1“.) Man könnte also sagen, dass die Interfaces

StockOwner und *Employee* miteinander verschmolzen wurden.

Die Begründung dafür liegt – natürlich – in der Metrik. Dazu sollte man einen Blick auf die Metrikdaten der beiden Interface Sets werfen. Das Interface Set *vor* der Änderung an der Klasse *Bank* (mit den drei Interfaces) wird mit IS_{old} bezeichnet, das danach (mit den zwei Interfaces) als IS_{new}. Dann ergibt sich das Bild aus Abb.27. Die Codeänderung hat also bewirkt, dass

	IS_{old}	IS_{new}
Metrik-Wert vor der Codeänderung	1.21	1.16
Metrik-Wert nach der Codeänderung	0.68	1.18

Abbildung 27: Die Metrikdaten für die beiden Interface Sets von *Person*

IS_{new} besser abschneidet, und deshalb ist es vom IISP, der ja alle Möglichkeiten durchspielt als neues Interface Set vorgeschlagen worden. Man kann auch nachvollziehen, warum IS_{old} nach der Codeänderung so schlecht abschneidet, wenn man es einfach von Hand zusammenbaut[38]. (Abb.28). Wie

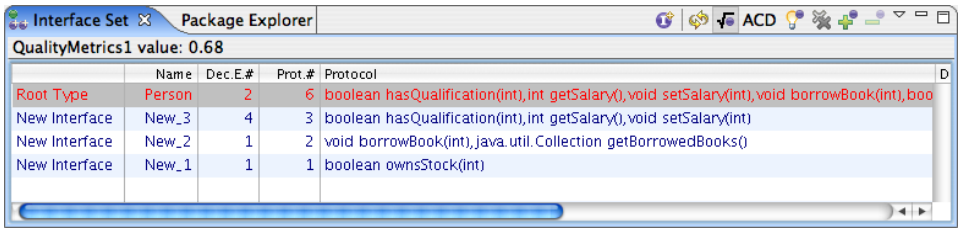


Abbildung 28: Das „alte“ Interface Set von *Person* nach der Codeänderung

in der Spalte *Dec.E.#* sichtbar sind zwei Deklarationselemente, die noch in Abb.22 mit dem Interface „New_3“ bzw. *StockOwner* umdeklariert werden sollen, nun weggefallen und werden statt dessen mit der Klasse *Person* belassen. Dadurch sinkt die Popularität des Interfaces „New_3“ auf 1, und dies ist, wie man an den Metrikwerten sieht, für die Metrik nicht ausreichend, um die Einführung diese Typs zu rechtfertigen.

Man könnte nun weiter experimentieren und weitere Deklarationselemente einführen, die *nur ownStock(..)* benutzen, und so herausfinden, ab welcher Schwelle sich die Einführung dieses Typs laut der Metrik wieder „lohnt“. Auf diese Weise lässt sich ein Eindruck gewinnen, ob eine Metrik im Einzelfall in der Lage ist, plausibel erscheinende Interfaces vorzuschlagen. Ein möglicher Maßstab wäre, dass es die gleichen Interfaces sind, die man selbst nach einem Studium des Quelltextes vorgeschlagen hätte.

Um ein allgemeineres Bild vom Verhalten einer Metrik zu bekommen, und auch um interessante Kandidaten für Refactorings in einem Projekt zu finden, kann es nützlich sein, sich Interface Sets automatisch für alle Typen eines Projekts vorschlagen zu lassen.

[38] Wie das geht, steht ebenfalls in der Anleitung in Anhang D.

4.3 Anwendung auf komplette Programme

In [9] wurde das INFER TYPE Refactoring benutzt, um komplette Projekte zu refaktorisieren. Wie schon erwähnt, kommt der Autor dort zu dem Schluss, dass dieses Vorgehen wegen der großen Zahl neu eingeführter Typen wenig vielversprechend ist. Auch in [32] wird angemerkt:

> „[..]we observe that quite often it is natural to reference a class directly, and introducing an interface for this reference just makes no sense.[..] in particular, it is not useful to introduce interfaces just because it can be done – rather, there must be a concrete expected benefit, such as access protection or a future variation point in the design."

Diese Ausführungen beziehen sich auf INFER TYPE. Da die Beispielimplementierungen versuchen, einige der beschriebenen Probleme zu vermeiden, und mittels der Metrik ein Versuch unternommen wird, möglichst nur „relevante" Typen einzuführen, wäre es interessant, was eine Anwendung auf ganze Projekte ergibt. Dennoch trifft natürlich das letzte Argument aus dem genannten Zitat – also dass man für gewöhnlich nicht „irgendwie" entkoppeln will, sondern mit einem konkreten Ziel – auch hier zu.

Die automatische Durchführung der Refactorings wie in [9] ist nicht möglich, weil für das Refactoring im Interface-Designer das Eclipse Refactoring *Extract Interface* benutzt wird, und dieses lässt sich nicht ohne weiteres automatisch durchführen[39]. Was sich jedoch automatisieren lässt, ist die Analyse mit Hilfe der INTOJ SUITE *Calculators* und die Suche nach dem optimalen Interface Set durch den eingestellten Proposer. Verglichen wird also das Ergebnis der *Vorschau*, was nicht zwingend mit dem tatsächlichen Resultat übereinstimmt. Weiterhin steht mit dem *Minimal Interfaces Proposer* ein Proposer zur Verfügung, der das Verhalten von INFER TYPE in gewisser Weise simuliert.

Es ist also möglich, eine Art Vorschau zu erhalten, wie sich das Projekt verändern würde, wenn man die eingestellten Algorithmen auf alle Typen eines Projektes anwendet. In einem speziellen View werden die Ergebnisse für alle Typen des Projekts angezeigt. Dort kann man sehen, wie viele neue Typen die benutzten Algorithmen für jeden Typ vorgeschlagen haben und wie dadurch der ACD-Wert verbessert würde. Es bleibt natürlich weiterhin dabei, dass die reale Umsetzung durch das Eclipse-Refactoring wegen der in Kapitel 5.2.1 beschriebenen Probleme abweichende Ergebnisse bringen kann. Bis diese Diskrepanz also behoben ist, ist das Resultat ohne Gewähr.

[39] Das Refactoring Framework von Eclipse gibt vor, dass die Ausführung eines Refactorings in Interaktion mit dem Benutzer durchgeführt wird. Beim Start eines Refactorings ist dazu jeweils ein passender Wizard bereitzustellen, der ein Fortsetzen des Vorgangs in mehreren Zwischenschritten nur gestattet, wenn bestimmte Voraussetzungen erfüllt sind.

Die Übersicht bringt den zusätzlichen Vorteil, dass man einen Eindruck von der Typnutzung im ganzen Projekt erhält. Wenn man sich für einen speziellen Typ näher interessiert, und z.B. nachvollziehen möchte, wie dessen Ergebnisse zu Stande gekommen sind, kann man ihn durch einfaches Doppelklicken in den anderen Views anzeigen. Dort kann man dann auf gewohnte Weise mit ihm arbeiten, indem man sich die Deklarationselemente anzeigen lässt usw. Außer der Anzeige der einzelnen Typen lässt sich auch eine Zusammenfassung mit zusätzlichen Metrikwerten ansehen, die unter anderem die *Decoupling/Growth Ratio*[40] anzeigt. Zu beachten ist, dass diese Form der projektweiten Analyse implizit unterstellt, dass die Typen eines Projekts alle zu einem Programm gehören. Man könnte aber durchaus mehrere Programme in einem Projekt unterbringen, auch wenn das wenig Sinn ergäbe. Die errechneten Daten wären dann jedenfalls nicht mehr aussagekräftig.

Die Analyse lässt sich starten, indem man im Package Explorer ein Projekt markiert und dann aus dem Kontextmenü *Analyze possible ACD decrease* auswählt. Ist dann der IISP eingestellt, wird noch ein Dialog angezeigt, der ein globales Zeitlimit für die Berechnung des „besten“ Interface Sets abfragt. Da ja alle Typen automatisch analysiert werden sollen, ist später keine Interaktion mit dem Benutzer mehr möglich.

Die Analyse dauert schon bei mittelgroßen Projekten recht lange[41]. Sie wurde in der beschriebenen Form auf einige der in [9] getesteten Projekte angewendet.

Ein direkter Vergleich der Werte zeigt teilweise große Unterschiede zu den dort gefundenen Zahlen. Die Gründe dafür lassen sich teilweise nur vermuten. In [9] wird nicht immer eindeutig spezifiziert, was die Metriken genau zählen, der Begriff „declaration element“ wird etwa ohne weitere Erklärung oder Referenz benutzt. Um *reproduzierbare* Ergebnisse zu erhalten, wäre eine genaue Beschreibung aller Spezialfälle und Ausnahmen erforderlich. Darüberhinaus wird nicht beschrieben, mit welchen Werkzeugen die Analysedaten gewonnen wurden und welche Versionen der Beispielprogramme analysiert wurden. Die generelle Vorgehensweise ist natürlich ebenfalls anders: Es werden die Analysedaten erhoben, dann das Projekt durch eine nur grob beschriebene globale Anwendung von Infer Type refaktorisiert, und dann die Analyse erneut durchgeführt. Im Interface-Designer werden dagegen wie beschrieben nur verschiedene *Vorschauen* verglichen, die auf den Daten des Type Access Analyzer basieren. Auch dessen Spezifikation ist vage: In der Dokumentation [19] finden sich nur sehr kurze, krude Erklärungen; die Quelltextkommentare bieten ebenfalls nur wenig Hilfe. Es ist also prinzipiell bereits schwierig, genau zu ermitteln, was die Programme tun *sollen*; hinzu kommt die Möglichkeit von Fehlern in den Implementierungen.

[40] Vorgestellt in Kapitel 2.3.3 auf Seite 18.

[41] Die Berechnung benötigt auch sehr viel Speicher, man sollte also unbedingt Eclipse mit entsprechenden Kommandozeilenparametern so viel wie möglich Speicher zur Verfügung stellen.

Einige Unterschiede kann man identifizieren: So wird die Analyse in [9] auch auf innere und anonyme Klassen ausgedehnt. Bei den *Calculators* des Type Access Analyzer führt dies zu einer Fehlermeldung. Ohnehin erscheint es sinnvoller, sich bei der Entkopplung auf die Top-Level Klassen[42] zu beschränken. Schließlich ist es erwünscht, Interfaces vor allem an Modulgrenzen einzuführen, und diese verlaufen jedenfalls *nicht* innerhalb einer einzigen Quelltextdatei. Ein weiterer, bereits erwähnter Unterschied ergibt sich daraus, dass Infer Type bereits existierende Typen in die Analyse miteinbezieht, der Interface-Designer jedoch nicht.

Insgesamt ist also ein direkter Vergleich der Daten mit denen aus [9] nicht sinnvoll, der Nutzen besteht dagegen darin, verschiedene Implementierungen innerhalb des Interface-Designers vergleichen zu können.

Einen ausführlichen Auszug der Analysedaten findet man als Anhang A. Hier eine Zusammenfassung:

		JChessBoard	GoGrinder	Mars	DrawSWF
Type Number Growth	MIP	184 %	169 %	126 %	108 %
	IISP	79 %	66 %	43 %	44 %
ACD Difference	MIP	0.645	0.608	0.500	0.471
	IISP	0.271	0.279	0.181	0.269
Decoupling/Growth Ratio	MIP	0.35	0.36	0.40	0.44
	IISP	0.34	0.43	0.42	0.62

Zur Erläuterung:

- Unter *Type Number Growth* wird der prozentuale Zuwachs von Typen im Projekt verstanden. Also 100 % in dieser Zeile bedeutet: Es gäbe nach den Refactorings doppelt so viele Typen wie vorher.
- *ACD Difference* steht für die Entkopplung, die durch die Refactorings erreicht würden, gemessen an einer Senkung des durchschnittlichen ACD-Werts.
- Die *Decoupling/Growth Ratio* setzt die beiden vorigen Werte ins Verhältnis, indem die *ACD Difference* durch den Zuwachs an Typen geteilt wird. So ergibt sich die aus Kapitel 2.3.3 bekannte Formel:

$$DGR = (\varnothing ACD_{before} - \varnothing ACD_{after}) \frac{|T_{before}|}{|T_{after}|}$$

- Die Zeilen *MIP* und *IISP* stellen immer die Ergebnisse gegenüber, die durch den *Minimal Interfaces Proposer* und die Kombination aus *Iterative Interface Set Proposer* und *QualityMetrics1* gefunden werden.

[42] In Java ist pro Quelltextdatei genau eine mit *public* deklarierte Klasse erlaubt. Das ist die Top-Level Klasse.

Hier noch einmal die Durchschittswerte:

		Ø
Type Number Growth	MIP	147 %
	IISP	58 %
ACD Diff.	MIP	0.556
	IISP	0.250
Decoupling/Growth Ratio	MIP	0.39
	IISP	0.45

4.4 Problem der Namensgebung an Stichproben

Eine wichtige Forderung an den Vorschlagsalgorithmus ist noch nicht überprüft worden, und zwar, ob die gefundenen Interfaces *plausibel* sind. Dieser Punkt ist als „weicher" Faktor natürlich schwer mit wissenschaftlichen Methoden zu fassen. Er ist aber dennoch von zentraler Bedeutung, denn letzlich wird ein Programmierer Interfaces ablehnen, wenn sie nicht plausibel sind.

Geht man von dem in [30] vorgestellten Konzept des „Interface als Rolle" aus, sollte der Name eines Interfaces die Rolle beschreiben für die es steht. Gelingt es nicht, einen aussagekräftigen Namen zu finden, könnte dies ein Hinweis darauf sein, dass das Interface nicht eindeutig mit einer Rolle identifiziert werden kann.

Ein aussagekräftiger Name im Sinne einer Rollenbeschreibung muss zwei Kriterien erfüllen:

1. Alle Methoden im Interface können mit der durch den Namen assoziierten Rolle in Verbindung gebracht werden. Beispiel: Ein Interface wie *IJavaElement* sollte keine Methode *getHostname()* haben, weil man sie nicht mit Elementen einer Programmiersprache, sondern eher mit einem Rechner im Netzwerk assoziiert.
2. Die Rolle, die durch den Namen vorgegeben wird, muss durch die Methoden auch erfüllt werden. Ein Interface mit einem Namen wie *Comparable* muss irgendwelche Vergleichsmethoden enthalten, wenn es etwa nur die Methode *playSound()* enthielte, wäre das unplausibel.

Aus praktischer Sicht käme zu den passenden Namen noch eine weitere Forderung hinzu, nämlich dass die Namen sich *hinreichend unterscheiden* sollen. Es ist sicherlich nicht der Verständlichkeit zuträglich, wenn zu viele sehr ähnlich klingende Typen die Hierarchie bevölkern[43].

Aber wie soll man diese Eigenschaften testen? In Ermangelung eines besseren Verfahrens werden hier einfach einige Klassen aus den Beispielprojekten herausgegriffen, und bei ihnen die Namensgebung getestet. Es wird begonnen

[43] Als Beispiel kann man die Klassen der Java GUI Frameworks anführen: Dort gibt es eine Klasse *java.awt.Window*, von der *javax.swing.JWindow* und *java.awt.Frame* erben. Von letzterer erbt dann noch *javax.swing.JFrame*.

mit einem relativ simplen Beispiel, der Klasse *SplashScreen* aus dem Projekt GoGrinder.

4.4.1 GoGrinder: SplashScreen

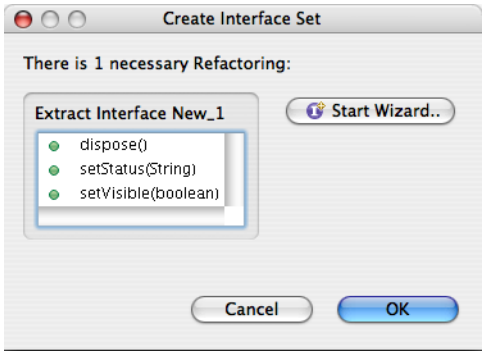


Abbildung 29: Die Namenssuche bei der Klasse *SplashScreen*

Die Klasse *SplashScreen* erbt von *JWindow* und stellt erwartungsgemäß ein Fenster dar, das bei länger andauernden Vorgängen erscheinen kann. Es ermöglicht keine Benutzerinteraktion, sondern informiert den Benutzer lediglich darüber, was gerade geschieht, ähnlich einem Fortschrittsbalken. Wenn der Vorgang abgeschlossen ist, verschwindet das Fenster selbsttätig wieder.

Allgemein gesprochen, muss das Protokoll dieser Rolle also Methoden anbieten für die Funktionen: „Erscheinen“, „Nachricht anzeigen“ und „Verschwinden“. Zwei der im Dialog vorgeschlagenen Methoden kann man direkt diesen Funktionen zuordnen: *setVisible(..)* zum Erscheinen und Verschwinden und *setStatus(..)* zur Anzeige einer Nachricht. Die dritte Methode, *dispose()* dient zur Freigabe der Resourcen, wenn der Splashscreen nicht mehr benötigt wird. Das vorgeschlagene Interface erscheint also recht plausibel, man könnte es z.B. *ISplashScreen* nennen. Man kann sich auch gut vorstellen, dass die dadurch erreichte Entkopplung Vorteile bringt. Die Klasse *Splashscreen* erbt wie gesagt von *JWindow* und ist daher auf den Einsatz innerhalb einer Swing-GUI beschränkt. Benutzt man dagegen das vorgeschlagene Interface, kann man einen entsprechenden Splashscreen auch mit anderen GUI-Frameworks implementieren, beispielsweise SWT. Man könnte sich auch leicht eine Implementierung ohne GUI vorstellen, die einfach die Nachricht auf der Kommandozeile ausgibt. Das könnte von Nutzen sein, wenn man etwa Komponenten auf einem Server testen will.

Man beachte, dass das vorgeschlagene Interface einen sehr hohen Metrikwert erhält, in diesem Fall 92. Das liegt daran, dass die zugrundeliegende Klasse *SplashScreen* von *JWindow* ein riesiges Protokoll erbt, typischerweise 200-300 Methoden. Von diesen werden nur einige wenige benutzt, so dass der

ACD-Wert nahe 1 ist, also extrem schlecht. Durch ein neues Interface wird hier also eine immense Verbesserung im Sinne einer besseren Ausnutzung des angebotenen Protokolls erreicht.

4.4.2 Mars: Host

Das nächste Beispiel ist etwas komplexer, denn es ist die Klasse *Host* aus dem *Mars* Projekt. Ihre Aufgabe laut Javadoc:

```
/** Represents a single host monitored by MARS. Each Host contains a
set of Services, which contain service specific information.
*/
```

Es handelt sich offenbar um eine Model-Klasse, die Informationen über einen Host speichert und eine Menge an Services für ihn verwaltet. Das von den Beispielimplementierungen vorgeschlagene Interface Set besteht aus den in Abb.30 und 31 gezeigten zwei Interfaces. Betrachtet man die Methoden der

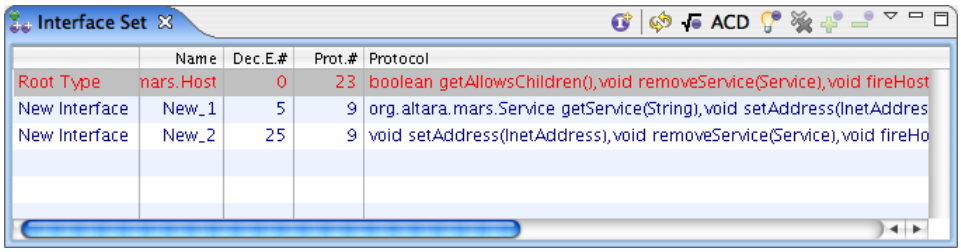


Abbildung 30: Das Interface Set der Klasse *Host*

beiden Interfaces genauer, findet man im Interface *NEW_2* eher Methoden, die das Model mit Daten füllen; so etwa die Methoden:

- *setName(..)*
- *setAdress(..)*
- *addService(..)*
- *removeService(..)*

Auch die Methode *fireHostChanged()* kann man diesem Funktionsbereich zuordnen, weil sie nach Änderungen am Model aufgerufen wird, um die Listener zu informieren. Dies könnte man unter eine Rolle *WriteableHost* fassen. Verwirrend ist aber, dass sich einige dieser schreibenden Methoden auch in dem anderen Interface wiederfinden. Wenn man dieses also einer Nur-lese-Rolle zuordnen wollte, etwa durch einen Namen wie *ReadOnlyHost* wäre eine Methode wie *setAdress(..)* darin wenig plausibel. Es ist somit in diesem Beispiel schwierig, die gefundenen Interfaces ausreichend voneinander abzugrenzen, so dass sich plausible Rollen – und entsprechend plausible Namen – finden liessen. Immerhin bringen die Vorschläge eine gewisse Inspiration, wie sich

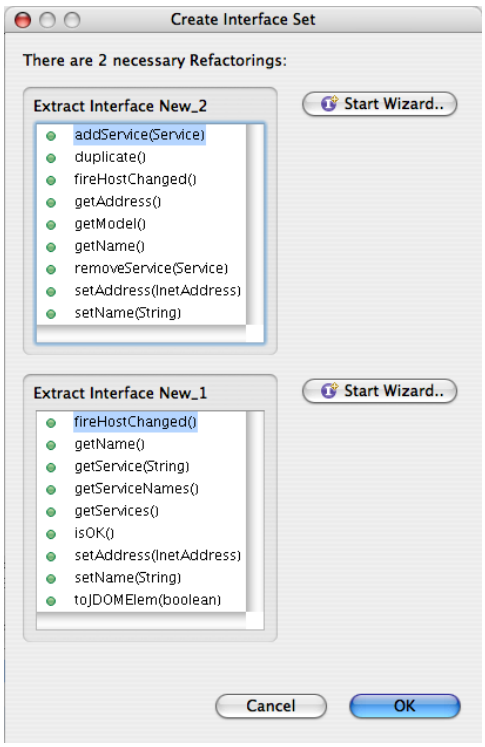


Abbildung 31: Die Namenssuche bei der Klasse *Host*

das Protokoll der Klasse möglicherweise aufteilen liesse. Die Tatsache, dass in allen Programmteilen anscheinend schreibend auf das Model zugegriffen wird, könnte auch einen Hinweis auf Mängel in der Architektur sein. Dies würde bedeuten, dass die Klasse tatsächlich in einem architektonischen Sinne *falsch benutzt* wird; eine nutzungsbasierte Analyse kann dann nur diesen Umstand wiedergeben, und ihn nicht selbst verändern.

Um solche Aspekte und andere wirklich beurteilen zu können, ist eine intime Kenntnis des gesamten Projekts erforderlich. Deshalb wird als letztes Beispiel eine Klasse aus einem eigenen Projekt herangezogen.

4.4.3 Kaskade: Position

Die Klasse *Position* stammt aus einem Strategiespiel namens *Kaskade*[44], und repräsentiert einen Spielstand. Es handelt sich um eine komplexe Model-Klasse, die im ganzen Projekt sehr häufig verwendet wird. Als Interface Set wird dennoch nur ein einziges Interface vorgeschlagen, wie in Abb.32 zu sehen. Die Klasse selbst verfügt über noch weit mehr Funktionen, das Interface reduziert also die Größe des zur Verfügung stehenden Protokolls deutlich. Außerdem

[44] Entstanden im Rahmen eines Programmierpraktikums an der Fernuniversität Hagen, zu finden unter: `http://letsgothere.de/kaskade/`. In der Programmdokumentation (innerhalb des Programms aufrufbar) findet sich eine Beschreibung der Klasse *Position* unter „Grundidee und Konzepte“.

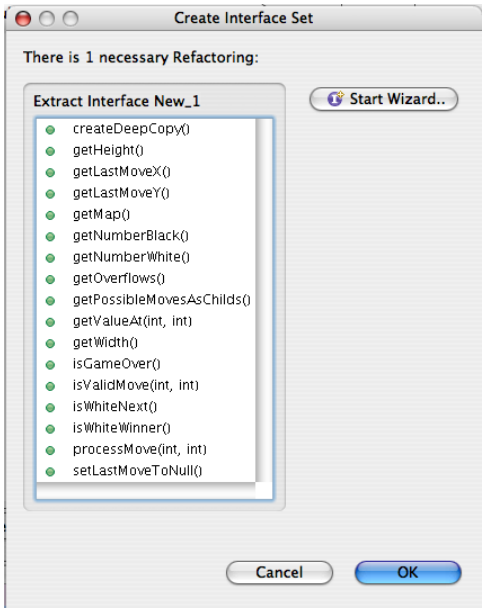


Abbildung 32: Die Namenssuche bei der Klasse *Position*

kann es an sehr vielen Stellen benutzt werden; durch die Kombination dieser Eigenschaften erhält es einen hohen Wert in der Beispielmetrik. Wirft man einen Blick auf die Methoden, ergibt sich das Problem, dass sie eigentlich alle wesentlichen Funktionsbereiche der Klasse berühren. Es ist also schwierig, eine spezielle Rolle zu identifizieren, die dieses Interface im Gegensatz zur Originalklasse spielen könnte. Es stellt im Prinzip eine Vereinigung der am häufigsten vorkommenden Access Sets dar, man hätte es also auch konstruieren könne, indem man beim totalen Interface der Klasse (also dem, das das gesamte Protokoll enthält), die Methoden entfernt, die nur an wenigen Stellen benutzt werden. Aber wie könnte man dieses Interface nennen? Ein Name wie *IPosition* würde ein totales Interface suggerieren, das überall anstelle von *Position* benutzt werden kann. Ein Name wie *ReducedPosition* erscheint ebenfalls wenig sinnvoll, weil er zu vage ist und keinen Hinweis gibt, was hier warum reduziert ist.

In der Tat wird die Klasse *Position* durchaus in verschiedenen Kontexten auf verschiedene Weise benutzt, z.B. benutzen die GUI-Komponenten nur einen Auschnitt ihres Protokolls, weil sie einerseits nur lesend zugreifen, und andererseits bestimmte zusätzliche Informationen für die Animation benötigen. Aber diese unterschiedliche Nutzung hat sich nicht in einem Interfacevorschlag manifestiert. Was sind die Gründe dafür? Zwei Gründe lassen sich auf Anhieb in Schwächen des IISP finden:

1. Der IISP erwägt nur Interfaces, die sich durch die Kombination von Access Sets[45] gewinnen lassen. Dies ist ein Problem, wenn ein wün-

[45]Hier wieder im Sinne der *benutzten* Subprotokolle.

schenswertes Interface an keiner Stelle im Programm in disjunkter Form benutzt wird. Dies kann man sich gut an dem Beispiel *Position* verdeutlichen: Angenommen, alle Deklarationselemente benutzen eine gewisse „Basisfunktionalität“ der Klasse. Dies könnten etwa die Methoden sein um zu ermitteln, welche Steine auf einem bestimmten Feld liegen, welcher Spieler am Zug ist, usw. Dann werden an einigen Stellen *zusätzlich* bestimmte „Spezialfunktionen“ benutzt. Naheliegend wäre nun, ein Interface mit den Basisfunktionen und eines mit den Spezialfunktionen zu erzeugen. Wo die Spezialfunktionen gebraucht werden, wird dann ein Typ benutzt, der beide Interfaces implementiert; denkbar wäre auch, dass das Spezialinterface von dem Basisinterface erbt. Der IISP kann aber das Spezialinterface nicht vorschlagen, weil es an keiner Stelle als Access Set vorkommt, es sind immer noch die Basisfunktionen mit dabei. Dies könnte behoben werden, indem man z.B. auch die Differenzmengen als Grundlage der Iteration hinzunimmt. Allerdings würde sich dadurch die Datenmenge weiter vergrößern, diese ist jedoch ohnehin ein kritischer Faktor.

2. Die Heuristik zur Datenreduktion, die der IISP verwendet, ist sehr simpel. Bei einer großen Anzahl von gefundenen Access Sets würde die Iteration bekanntlich zu lange daueren. In der Praxis ist die Schwelle etwa bei zehn Access Sets. Werden mehr Access Sets gefunden, was z.B. bei *Position* der Fall ist, wird die Datenmenge reduziert, indem nur die Access Sets verwendet werden, die für sich genommen die besten Metrikwerte erreichen. So fallen also einige Access Sets weg, die aber möglicherweise gerade die gesuchte Spezialfunktionalität enthalten haben. Dadurch wird also das 1. Problem u.U. noch verschlimmert. Die Problematik zeigt gewisse Parallelen zu den bei der KI verwendeten Heuristiken zur Datenreduktion: Bei Schach gibt es etwa Algorithmen, die bei einer sehr schlechten Bewertung des 1.Zuges die darauf aufbauenden weiteren Züge nicht berechnen. Programme, die so arbeiten, haben dann das Problem, dass sie nicht im Stande sind, die Dame zu opfern, auch wenn der Gegner im nächsten Zug schachmatt wäre.

   Im Fall des IISP müsste eine Heuristik gefunden werden, die besser entscheiden kann, was weggelassen wird. Dazu könnte man z.B. in Betracht ziehen, wie sehr sich Access Sets überschneiden. Access Sets, die sich in weiten Teilen überschneiden könnten zusammengefasst und als Einheit behandelt werden. Außerdem könnte die Reduktion mehrstufig erfolgen. Im Moment werden ja alle Access Sets zunächst einzeln bewertet, dann reduziert und dann alle Kombinationen aus den verbleibenden berechnet. Man könnte jedoch auch erst einmal Gruppen von Access Sets bilden, und innerhalb dieser Gruppen schon Kombinationen berechnen und bewerten. Von den Ergebnissen könnte dann die weitere Reduktion abhängig gemacht werden.

5 Diskussion

5.1 Allgemeine Probleme

5.1.1 Allgemeine Probleme von Refactorings zur Einführung allgemeinerer Typen

Auf die allgemeinen Probleme von Refactorings zur Einführung allgemeinerer Typen wird in [33] genauer eingegangen, die dort angeführten Punkte haben auch alle Gültigkeit für den Interface-Designer:

- Obwohl Refactorings nach [11] das externe Verhalten eines Programms nicht verändern sollen, tun sie es bei Refactorings wie *Extract Interface* dennoch: Denn an den Stellen, wo vorher eine konkrete Klasse gefordert war, wird nun jeder Typ akzeptiert, der das Interface implementiert. Gerade dadurch wird das Programm ja flexibler. Es kann aber zu Problemen führen, wenn dadurch Typen akzeptiert werden, mit denen der Programmierer nicht gerechnet hat und die besser abgelehnt würden.
- Es besteht die Gefahr, dass ein impliziter Vertrag ignoriert wird, weil er sich nicht in der Syntax des Typs ausdrückt. In [33] wird als Beispiel ein Ersatz des Typs *Set* durch einen Typ *Collection* genannt. Dies könnte anhand der Nutzungsanalyse vorgeschlagen werden, wenn beide Klassen z.B. eine Methode *iterator()* haben, und nur diese an der Codestelle benutzt wird. Es kann aber sein, dass die Mengen-Eigenschaft – also keine Duplikate zu enthalten – an dieser Stelle wichtig ist, ein Umstand der durch die Nutzungsanalyse nicht erkannt wird.
- Mehrere Probleme ergeben sich im Zusammenhang mit überladenen Methoden. Diese werden in Java nach einem mehrschrittigen Verfahren statisch gebunden, und zwar zwar nach dem Prinzip, die zu einem Aufruf am „besten passendste“ Signatur zu finden. Kann dies nicht eindeutig bestimmt werden, führt es zu einem Compilerfehler. Das Umdeklarieren mit anderen Typen kann hier zu verschiedenen, in [33] grob umrissenen Problemen führen.
- Alles was mit Reflection zu tun hat, ist aus der Sicht eines automatisierten Refactorings nicht nachvollziehbar, da hier Typ- und Methodennamen berechnet werden können, aus Dateien gelesen, usw.

Ein Punkt aus [33] muss für den Interface-Designer etwas modifiziert betrachtet werden, nämlich das Problem, dass ein Interface, das kontextspezifisch eingeführt wird, sich möglicherweise an eine andere Stelle fortpflanzt, etwa als Methodenparameter. In dem neuen Kontext ergibt es aber möglicherweise keinen Sinn, denn der mit dem Kontextwechsel einhergehende Rollenwechsel drückt sich in dem Interface nicht aus. Als eine mögliche Lösung wird genannt, den Rollenwechsel durch Casts explizit zu machen. Da im Interface-Designer prinzipiell die Möglichkeit besteht, die Interfaces nicht

ganz so spezifisch anzulegen wie bei dem in [32] behandelten INFER TYPE, ist zu erwarten, dass das Problem tendenziell seltener auftritt.

5.1.2 Grundlegende Probleme von Metriken

> „Beware! The domain of metrics is deep and muddy waters."
> (Brian Henderson-Sellers 1996 in [15])

Auch wenn die Messdaten zuverlässig sind, ist bei Schlussfolgerungen auf der Basis von Metrikdaten extreme Vorsicht angebracht. Man darf sich nicht von der vermeintlichen Objektivität umfangreicher Tabellen wie in Anhang A blenden lassen. Ohne hier zu tief in das Thema einzusteigen, kann man einige offensichtliche Schwächen der benutzten Metriken ausmachen.

Die Kopplung wird im Sinne des ACD-Werts berechnet, indem die Anzahl der benutzten Methoden gezählt wird. Da dieser Ansatz notgedrungen die Semantik der Methoden unberücksichtigt lässt, ist seine Aussagekraft begrenzt. Oftmals wird die Anzahl der Methoden durch *Convenience-Methoden* erhöht. So etwa in der Klasse *Position* aus dem vorigen Abschnitt. Die Methode *isGameOver()* gibt darüber Auskunft, ob ein Spieler gewonnen hat und das Spiel somit beendet ist. Die Information liesse sich auch aus der Anzahl der Spielsteine jedes Spielers ableiten, auf die Methode könnte also auch verzichtet werden. Die Funktionalität – das Spielende abzufragen – wird jedoch sehr häufig benutzt, und dies jedesmal aus der Anzahl der Steine zu folgern wäre mühsam und fehlerträchtig. Die Methode in eine Hilfsklasse auszulagern würde dem Prinzip der Objektorientierung zuwiderlaufen. Sie ist also in der Klasse *Position* ganz gut aufgehoben. Auf den ACD-Wert hat die Einführung einer solchen Convenience-Methode jedoch einen eigenartigen Effekt: Jede Klasse, die die Methode benutzt, wird dadurch als stärker an die Klasse *Position* gekoppelt bewertet. Macht die Klasse die Berechnung selbst gilt sie als schwächer gekoppelt, was insofern fragwürdig ist, als das Ergebnis ja immer gleich ist, die „schwächer gekoppelte" Version arbeitet lediglich mit mehr dupliziertem Code.

Ein weiterer Effekt ist, dass an Codestellen, die die Convenience-Methoden *nicht* benutzen, durch deren Einführung plötzlich laut der ACD-Metrik eine „unnötige Kopplung" entsteht, weil ein geringerer Teil des zur Verfügung gestellte Protokolls genutzt wird (weil sich das Protokoll durch die Convenience-Methode vergrößert hat). Dies tritt besonders deutlich zutage, wenn gleich mehrere alternative Methoden angeboten werden.

Beispiel: Man will die Größe einer rechteckigen Komponente festlegen. Das kann auf verschiedene Arten möglich sein:

```
setWidth(int width)
setHeight(int height)
setSize(Point size)
setSizeEqualTo(Rectangle rect)
```

Dabei werden in allen Methoden die gleichen Daten transportiert, in der dritten und vierten Methode werden die Breite und Höhe lediglich aus den übergebenen Objekten ausgelesen. Wahrscheinlich wird an einer Codestelle immer nur eine der Methoden benutzt, aber eine „unnötige Kopplung“ entsteht durch die Anwesenheit der anderen Methoden im Protokoll eigentlich nicht; sie stellen eher minimale Variationen des Protokolls „Größe setzen“ dar, die ein eleganteres Programmieren ermöglichen. Die durch den ACD-Wert ausgedrückte Kopplung ist also mit einer gewissen Vorsicht zu betrachten.

Hinzu kommt, dass die Entkopplung selber auch kein Selbstzweck ist. Man will mit ihr übergeordnete Ziele wie vereinfachte Wartung, schnellere Entwicklung, Kostensenkung, usw. erreichen. Es liegt auf der Hand, dass nicht *jede* Entkopplung an irgendeiner Stelle dem Erreichen dieser Ziele dient. So ist z.B. ein hoher Grad an Kopplung innerhalb eines Moduls (bzw. einer Komponente) durchaus erwünscht. Man hat also mit einer Kombination aus zwei unbekannten Faktoren zu tun:

- Beim ACD-Wert muss man die Frage stellen: Gibt er wirklich die Kopplung zweier Klassen exakt wieder?
- Angenommen, es wäre so, und man senkt die Kopplung zweier Klassen, bleibt die Frage: Dient dies wirklich den übergeordneten Zielen leichtere Wartung, Kostensenkung, usw.?

Vor diesem Hintergrund sollte man die Metriken eher nicht als Basis für Schlussfolgerungen betrachten, sondern als – potentiell unzuverlässige – Tippgeber, die den Programmierer auf besondere Stellen im Code aufmerksam machen können. Wenn auf diese Weise etwa ein „bad smell“, wie er in [11] beschrieben ist, schneller oder leichter entdeckt wird, ist das ein konkreter Nutzen.

Ähnliches kann man sagen über das Konzept, Interfaces zu präferieren die möglichst oft benutzt werden. Man kann sich leicht eine Situation vorstellen, wo ein Interface sehr speziell sein muss, vielleicht nur einmal benutzt wird, aber dafür *genau an der richtigen Stelle*. Also ist auch hier wieder das Problem das fehlende automatische Programmverstehen.

5.1.3 Nachteile von Interfaces

Bei allen Vorzügen, die die Verwendung von Interfaces mit sich bringt, darf man nicht vergessen, dass sie auch Nachteile haben. Bereits erwähnt wurde der größere Wartungsaufwand, da mehrere Codestellen, also Interface und Implementierungen, parallel gepflegt werden müssen. Darüberhinaus gibt es zwei weitere Nachteile:

- Das dynamische Binden zur Laufzeit ist möglicherweise langsamer als statisches Binden und verschlechtert dadurch die Performance.

- Das Debuggen wird durch Interfaces teilweise erheblich erschwert, weil ohne statischen Bindung mit dem Code allein nicht festgestellt werden kann, welche Implementierung zur Ausführung kommt. Es ist also viel schwerer, die relevanten Codestellen zu finden.

Zum ersten Punkt kann man sagen, dass er in sehr vielen Fällen ohne Bedeutung ist, so wird in [1] etwa eine Methode beschrieben wird, dynamisches Binden ohne Performanceverlust zu implementieren. Das Problem tritt also nur noch in Fällen auf, in denen die JVM von derartigen Optimierungen keinen Gebrauch macht, oder als „potentielles Problem", wenn man nicht weiß auf welcher (oder welchen) JVM(s) der Code später laufen wird. Generell bilden nach [11](S.69) Strukturverbesserungen und Performance eines Programms einen (möglichen) Tradeoff. Der Autor rät jedoch davon ab, auf Strukturverbesserungen zu verzichten, weil man bloss *vermutet*, dass die Performance darunter leidet. Er empfiehlt dagegen, zunächst eine möglichst gute Struktur anzustreben und dann mögliche Performanceprobleme – sofern sie auftreten – mithilfe von Profilern genau zu lokalisieren; dabei stelle sich oftmals heraus, dass die tatsächlichen Engpässe ganz woanders liegen als vermutet. Die bessere Struktur erleichtere dann das Performance-Tuning.

Vor diesen Hintergrund erscheint es meistens nicht ratsam, aus Performancebedenken auf Interfaces zu verzichten. Ausnahmen wären möglicherweise Systeme mit extremen Rahmenbedingungen, wie etwa Real-Time Systeme oder Software für mobile Geräte mit extrem begrenzten Resourcen.

Der zweite angeführte Nachteil von Interfaces ist schon eher stichhaltig. Jeder, der schon einmal ein Programm mit vielen Interfaces debuggt hat, weiß das: In den IDEs gibt es immer eine Möglichkeit, per Tastendruck zur Implementierung einer Methode zu springen, bei Interfaces ist man dann zunächst in einer Sackgasse[46]. Normalerweise ist der nächste Schritt, sich alle Implementierungen anzeigen zu lassen; falls es nur eine gibt, weiß man dann schon, wo man weitersuchen muss. Stehen mehrere zur Auswahl, bleibt oft als einzige Lösung, einen Breakpoint an der problematischen Stelle zu setzen, die Anwendung zu starten und dann *nachzusehen*, welcher konkrete Typ auftritt. Entsprechend komplizierter wird es, wenn man es mit einem Geflecht dynamischer Objekte zu tun hat, bei dem etwa ein Fehler nur in einer ganz bestimmten Kombination von konkreten Typen auftritt. Zudem kann die zeitliche Abfolge eine Rolle spielen: Ein Fehler entsteht beispielsweise nur, wenn dynamische Objekte in einer bestimmten Reihenfolge auftreten; in diesem Fall muss beim Debuggen aufgezeichnet werden, welche konkreten Typen aufgetreten sind und den Fehler verursacht haben.

Moderne IDEs bieten immer ausgefeiltere Debugging-Werkzeuge, um diese Probleme anzugehen. Dennoch bedeutet die Möglichkeit des dynamischen

[46] Bei *überschriebenen* Methoden ist die Situation mitunter noch verwirrender, da man immer in der Implementierung des deklarierten Typs landet. Diese wird aber vielleicht gar nicht ausgeführt. Bei einem Interface weiß man immerhin, dass man nun den dynamischen Typ feststellen muss.

Bindens, dass es schwieriger wird, festzustellen, welcher Code zur Laufzeit tatsächlich ausgeführt wird. Auf der anderen Seite erleichtert das dynamische Binden das Testen und die Fehlersuche, weil es durch die Anwendung von Prinzipien wie *Inversion of Control* [10] überhaupt erst die saubere Trennung von Komponenten ermöglicht. Die Leistungsfähigkeit moderner Applikationen wäre ohne die Verwendung von Komponenten wie z.B. externen Datenbanken nicht möglich. Wenn es dabei unmöglich (oder sehr aufwändig) wäre, einen Fehler in einer bestimmten Komponente zu isolieren, wären die Probleme viel größer als die, die durch das dynamische Binden entstehen: Man wüsste ja nicht einmal, wo man mit der Suche beginnen soll.

Darüberhinaus muss man bedenken, dass das dynamische Binden ja kein Selbstzweck ist. Es dient dazu, die Struktur zu verbessern und Software einfacher verständlich und leichter wartbar zu machen, indem Dinge wie duplizierter Code vermieden werden. Der Vorteil, eine Implementierung direkt anspringen zu können, löst sich schnell in Luft auf, wenn der fehlerhafte Code an etlichen Stellen parallel vorkommt und einzeln korrigiert werden muss.

Man kann also festhalten, dass durch Interfaces (oder dynamisches Binden im allgemeinen) zwar das Debugging im Detail komplizierter wird, sie bei einem sinnvollen Einsatz aber Vorteile bieten, die diesen Mehraufwand rechtfertigen; bei Systemen ab einer gewissen Komplexität erscheinen sie praktisch unverzichtbar.

5.2 Probleme beim Interface-Designer

5.2.1 Probleme im Zusammenhang mit dem Type Access Analyzer

Die Weiterentwicklung der Sprache Java schreitet rasch voran, besonders seitdem sie sich zu einem industriellen Schwergewicht entwickelt hat. Änderungen wie die in Java 5 erfordern dabei ständige Änderungen an Tools wie der IntoJ Suite, bzw dem Type Access Analyzer. Die IntoJ Suite ist nun schon einige Jahre alt und wurde auch nicht durchgehend gepflegt, daher gibt es im Moment eine Reihe von Einschränkungen, unter denen auch der *Interface-Designer* zu leiden hat.

- Die Neuerungen von Java 5 wie z.B. Generics und der *enhanced for loop* sind nicht verarbeitbar. Ihre Analyse führt sowohl zu Fehlern als auch zu nicht gefundenen Deklarationselementen.
- Auch bei älterem Code werden gelegentlich Deklarationselemente nicht gefunden, besonders im Zusammenhang mit Arrays. Meistens sind die Fehler nur gering, bei einigen speziellen Fällen jedoch auch sehr gravierend.
- Die Classpath Auflösung der *Calculator* ist ungenügend. Es werden nur Referenzen im selben Projekt gefunden, referenzierte Projekte oder Ele-

mente, die per Konfiguration in den Classpath aufgenommen werden, also Plugins werden nicht entdeckt. Für den ersten Fall kann man als Workaround das referenzierte Projekt in einen eigenen Source Folder kopieren, der in dem zu analysierenden Projekt angelegt werden muss.

Wenn man den Eindruck hat, dass Deklarationselemente eines Typs nicht gefunden werden, kann man sich einen Überblick verschaffen, indem man z.B. auf dem Kontextmenü des Typs im Java Editor *References/Project* auswählt. Dann werden alle Referenzen auf den Typ im gleichen Projekt in einem eigenen View angezeigt. Indem man diese einzeln durchgeht, kann man sich selbst die relevanten Stellen anschauen und sehen, ob die Analyse zutrifft. Beim *Extract Interface* Refactoring sollte man immer die Vorschaufunktion verwenden, um zu sehen, ob der Eclipse Algorithmus evt. zu anderen Ergebnissen gekommen ist, als vorher im *Interface Set View* angezeigt worden ist. In vielen Fällen findet das *Extract Interface* Refactoring mehr Deklarationselemente die geändert werden können, man sollte also sichergehen, dass das Resultat auch so gewünscht ist.

5.2.2 Probleme im Zusammenhang mit dem Eclipse Refactoring

Der Interface Designer benutzt nicht nur die IntoJ Suite als Datenquelle, sondern auch noch eine weitere Komponente am Ende des Workflows, nämlich das *Extract Interface* Refactoring von Eclipse. Dieses Vorgehen bringt eine Reihe von Vorteilen, allerdings hat das Eclipse Refactoring auch einige Eigenschaften, die eine Integration einschränken:

- Die Analyse geht davon aus, dass das Protokoll eines Typs in ein Interface ausgelagert werden kann, dazu gehören auch die ererbten Methoden. Das Eclipse Refactoring bietet jedoch keine ererbten Methoden an. Tritt dieser Fall auf, wird in einem Dialog darauf hingewiesen, welche Methoden nicht ins Eclipse Refactoring übernommen werden können. Der Benutzer hat dann die Möglichkeit, diese Methoden in der analysierten Klasse (z.B. durch einen einfachen Aufruf von *super.method()*) zu überschreiben. Dann können sie wie gewohnt ins Eclipse Refactoring übernommen werden. Wenn gewünscht, kann man danach diese Methoden wieder entfernen.
- Wenn sich die einzuführenden Interfaces überlappen, kann es sein, dass bestimmte Deklarationselemente mit verschiedenen Interfaces umdeklariert werden könnten. Das ist immer dann der Fall, wenn das benötigte Protokoll des Deklarationselementes in mehreren Interfaces vollständig enthalten ist. In diesem Fall ist die *Reihenfolge* wichtig, in der das *Extract Interface* Refactoring für die einzelnen Interfaces aufgerufen wird. Sind nämlich Deklarationselemente erst einmal umdeklariert, werden sie von einem weiteren Aufruf von *Extract Interface* auf dem Basistyp nicht mehr angetastet. Um das im Interface Set View prognostizierte Ergebnis

auch wirklich zu erhalten, werden deshalb die Refactorings im *Create Interface Set* Dialog nach ihrer Größe sortiert zum Aufruf angeboten (siehe Abb.33). Wird die Reihenfolge nicht eingehalten, führt das da-

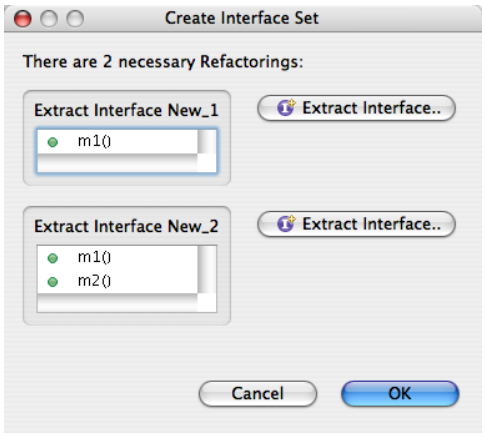


Abbildung 33: Bei der Einführung mehrerer überlappender Interfaces ist die Reihenfolge wichtig.

zu, dass Deklarationselemente möglicherweise nicht wie gewünscht mit dem minimalen Interface umdeklariert werden. Wenn man etwa bei dem Beispiel in Abb.33 zuerst das Interface *NEW_2* einführt, werden Deklarationselemente, deren Access Set nur aus *m1()* besteht, auch mit umdeklariert. Diese bleiben dann mit *NEW_2* deklariert, wenn man das Interface *NEW_1* einführt, was nicht dem gewünschten (und prognostizierten) Resultat entspricht.

Sind zwei der einzuführenden Interfaces gleich groß, ergibt sich keine verbindliche Reihenfolge; dennoch kann es aber vorkommen, dass Deklarationselemente für beide Interfaces in Frage kommen, wenn die Interfaces sich überschneiden und die Schnittmenge das Access Set der Deklarationselemente enthält. In diesem Fall liegt die Entscheidung beim Programmierer: Die Deklarationselemente werden mit dem Interface umdeklariert, dessen *Extract Interface* Refactoring zuerst aufgerufen wird. Ist geplant, die Deklarationselemente teilweise mit dem einen, teilweise mit dem anderen Interface umzudeklarieren, muss dies manuell erfolgen, indem die Preview-Funktion des *Extract Interface* Refactorings benutzt wird. Dort lassen sich die geplanten Änderungen am Quelltext einzeln deaktivieren

5.2.3 Prinzipbedingte Probleme

Nicht direkt ein Problem, aber doch eine Einschränkung ergibt sich aus dem Umstand, dass die Analyse immer auf einem konkreten Typ durchgeführt wird. Manche Interfaces wie z.B. *Comparable* oder *Serializable* entfalten jedoch ihren Nutzen erst dadurch, dass sie von verschiedenen Typen implemen-

tiert werden. Eine Nutzungsanalyse eines einzigen Typs kann nur begrenzt Hinweise auf die Nützlichkeit solcher Interfaces finden. Man kann sich dies am Beispiel *Comparable* verdeutlichen: Wenn eine Methode wie *compareTo(Object o)* beispielsweise ein Access Set bildet, könnte ein Interface mit dieser Methode für den untersuchten Typ durchaus vorgeschlagen werden (je nach verwendetem Algorithmus). Die automatische Verwendung dieses Interfaces für andere Typen übersteigt jedoch die Möglichkeiten der Nutzungsanalyse. Zum einen rein durch die Vorgehensweise: Es wird ja immer nur ein Typ für sich betrachtet. Dies könnte man bei einer projektweiten Analyse anders handhaben. Aber selbst wenn andere Typen auch über eine Methode *compareTo(Object o)* verfügen, würde sich die Annahme, dass diese Funktion auch dasselbe *tut*, nur auf die – möglicherweise zufällige – Namensgleichheit stützen. Die Verwendung desselben Interfaces könnte dann zu Problemen führen. Es wäre höchstens denkbar, bei der Einführung eines neuen Interfaces darauf hinzuweisen, dass ein Interface mit gleichen (oder fast gleichen) Methodensignaturen bereits existiert; der Programmierer müsste dann entscheiden, ob er das Interface auch in diesem Fall verwenden kann. Vielleicht stellt dies keinen gravierenden Nachteil dar, wenn bei anderen Interface-Typen Fortschritte erzielt werden können. Man sollte sich aber jedenfalls nicht der Illusion hingeben, dass sich jede Art von Interface automatisch ableiten lässt[47].

In dieser Hinsicht ist auch eine andere Begrenzung von Bedeutung: Die Nutzungsanalyse stützt sich natürlich nur auf den Code, der *erreichbar*[48] ist. Nicht einbezogen werden kann Code, der auf irgendeine Weise unerreichbar ist – sei es, dass er erst noch geschrieben werden muss, oder er wurde noch nicht ins Repository eingecheckt, oder er gehört zu einem anderen Projekt, bei der Enwicklung von Frameworks nicht ungewöhnlich. In der Praxis kann sich dies dadurch bemerkbar machen, dass z.B. die Vorschlagsautomatik ein Interface vorschlägt, wenn man aber dessen Protokoll inspiziert, sieht man, dass eine *wichtige Methode fehlt*. Dieser Fall tritt dann ein, wenn eine Methode zu der Rolle, die das Interface repräsentiert, sinngemäß unweigerlich dazugehört, aber durch Zufall in dem bereits geschriebenen Code noch nicht benutzt wurde. Oder, je nach verwendeter Metrik, nicht *oft genug*.

5.2.4 Probleme der Beispielmetrik

An der implementierten Beispielmetrik nach [8] fällt auf, dass bei ihr das implizite Interface des Basistyps, also das, was sich aus den mit *public* deklarierten Methoden ergibt, rechnerisch genauso wie ein neues Interface behandelt wird. Dadurch ist die Metrik nicht geeignet, wirklich eine komplette Entkopplung zu erreichen, wo dann auch die Implementierung in jedem Fall

[47] Zumindest wenn man von der momentanen Situation ausgeht, dass die Semantik von Methoden nicht formal spezifiziert wird, sondern allenfalls als Kommentar.

[48] In [41] als „closed-world assumption“ bezeichnet.

ausgetauscht werden kann. Das gehört allerdings nach [8] wohl auch nicht zu den Entwurfszielen der Metrik, sondern eher die Reduzierung der Anzahl der neu einzuführenden Typen. Es gibt jedoch noch eine Reihe anderer Probleme.

Wenn die Popularität eines neuen Interfaces auf Null sinkt, es also an keiner einzigen Stelle benutzt würde, sinkt sein Metrik-Wert ebenfalls auf Null. Dieser Fall müsste aber – da er ja unerwünscht ist – eigentlich „bestraft“ werden und nicht mit dem neutralen Element Null bewertet. So ist das Ergebnis, dass beim IISP mitunter Interface Sets vorgeschlagen wurden, in denen solche unbenutzten Interfaces vorkamen, denn die Metrik des Sets hat sich dadurch nicht verschlechtert. Das Problem wurde dann durch eine kleine „Schummelei“ behoben: Wenn der IISP ein Interface Set mit gleichem Metrikwert, aber weniger Interfaces findet, sieht er das als besser an. Eigentlich sollte dieser Qualitätsbegriff aber in der Metrik integriert sein.

Die Normalisierung der Metrik nach [8] bewirkt zwar, dass die Metrik des impliziten Interfaces immer 1 ergibt, bietet aber leider wenig Rückschlüsse darauf an, welche Typen in einem Programm von einem Refactoring besonders stark profitieren. Das liegt daran, dass die Metrik durch riesige, aus Libraries ererbte Protokolle verzerrt wird. Wenn etwa ein Typ von *JFrame* erbt, dann jedoch mit einem Interface umdeklariert wird, das nur wenige Methoden enthält, ergibt das riesige Metrikwerte. Solche Typen sind aber meist nicht die interessantesten Kandidaten für Refactorings.

Die Metrik basiert im Grunde auf einer Abwägung zwischen der Verbesserung des ACD-Werts und einer möglichst großen Popularität der Interfaces. Es wäre eigentlich naheliegend, die Gewichtung dieser beiden Elemente irgendwie zu parametrisieren.

5.3 Kritische Betrachtung der Fallbeispiele

5.3.1 Die gezielte Anwendung auf einzelne Typen

Wie im Kapitel Fallbeispiele bereits erwähnt, reichen die stichprobenartigen Versuche natürlich nicht aus, um eine fundierte Aussage über den Nutzen zu treffen. Das würde eine längere, praxisbezogene Studie erfordern. Eine Voraussetzung dazu wäre, dass für die Nutzungsanalyse ein Werkzeug bereitsteht, was zuverlässig und performant arbeitet und mit der aktuellen Java-Version umgehen kann.

Streng wissenschaftlich gesehen, kann man also aus den bisherigem Tests *nichts* folgern. Man muss sich damit begnügen, einige Eindrücke festzuhalten, diese könnten jedenfalls für die Weiterentwicklung des Werkzeugs von Nutzen sein:

- In bestimmten Fällen, wie etwa den konstruierten Fallbeispielen in Kap.4.1 und Kap.4.2, ist das Ergebnis der automatischen Konstruktion beeindruckend, bei der Anwendung auf reale Klassen sind die Ergebnisse

dagegen sehr gemischt.

- Hat man eine genaue Vorstellung, was wovon entkoppelt werden soll, ist es oft gar nicht so einfach, das mit dem Tool zu erreichen. Das liegt daran, dass der Ansatz, der dem Tool zugrunde liegt, darauf basiert, einen konkreten Typ herauszugreifen und die Frage zu stellen: „Was für Interfaces könnten sich für diesen Typ anbieten?“

  Verfolgt man jedoch ein konkretes Ziel bei der Entkopplung, ist der Ansatz ein etwas anderer: Man hat eine mehr oder weniger präzise Vorstellung, welche Teile entkoppelt werden sollen, diese umfassen evtl. auch mehrere Klassen. Dann sollen alle Änderungen gemacht werden, *die dazu nötig* sind. Dabei können auch ganz andere Änderungen als die Einführung neuer Interfaces nötig sein.

  Die Situation tritt häufig ein, wenn man bereits existierenden Code refaktorisieren will, dieser Code aber geschrieben wurde, ohne speziell auf eine spätere Einführung von Interfaces zugeschnitten zu sein. Dann können die folgenden Probleme auftreten:

  – Deklarationselemente, die nicht dem Protokollbegriff entsprechen, sind für die Analyse unsichtbar. Das kann dazu führen, dass ein Deklarationselement, das unbedingt umdeklariert werden muss, um das angestrebte Ziel zu erreichen, gar nicht gesehen wird, weil es etwa eine mit *public* deklarierte Variable direkt benutzt, also ohne die Verwendung einer Accessormethode. Dies liesse sich relativ leicht durch die Einführung einer solchen Methode beheben, aber der Programmierer erhält keine Unterstützung dabei, die entsprechenden Deklarationselemente zu finden und zu ermitteln warum die Analyse sie nicht erfasst. Gleiches gilt, wenn statische Teile des Protokolls benutzt werden, oder die Sichtbarkeit von Methoden erhöht werden müsste.
  – Noch komplizierter wird es, wenn Deklarationselemente aus weniger trivialen Gründen nicht in die Analyse aufgenommen werden. Beispiel: Eine Model-Klasse besitzt eine Referenz auf den View, der eine Subklasse von *JFrame* ist. Dies ist im Sinne des MVC-Architekturmusters nicht günstig, weil es die Nutzung des Models unnötig fest an den konkreten View koppelt. Die Entkopplung soll es ermöglichen, die Model-Klasse auch mit anderen Views nutzen zu können, z.B. im Rahmen einer Webapplikation. Das Problem ist aber, dass die Model-Klasse Dialoge benutzt, um Fehler anzuzeigen:

    ```
    ApplicationWindow frame;
    ...
    JOptionPane.showMessageDialog(
      frame, "Error occured.");
    ```

    Die Variable *frame* ist dabei die Referenz auf den View. Leider führt die Herausgabe der Referenz an die Library-Methode dazu, dass das

Access Set des Deklarationselementes *frame* nicht festgestellt werden kann, und es deshalb komplett aus der Analyse verschwindet. Dies kann zunächst recht leicht durch einen Cast behoben werden:

```
JOptionPane.showMessageDialog(
  (Component)frame, "Error occured.");
```

Dadurch erscheint das Deklarationselement zumindest wieder in der Analyse. Allerdings kommt man so einer Entkopplung von der View-Klasse nicht viel näher, denn nun muss *frame* immer noch von einem Typ sein, der von *java.awt.Component* erbt. Als schnelle, wenn auch unsaubere Lösung würde sich anbieten, den Cast durch eine Methode *getComponent()* in *ApplicationWindow* zu ersetzen. Diese Methode würde dann in das (neu einzuführende) View-Interface aufgenommen, mit dem *frame* umdeklariert werden soll. Angenommen, das Interface heißt *IView*, dann sieht der Code nachher so aus:

```
IView frame;
...
JOptionPane.showMessageDialog(
  frame.getComponent(), "Error occured.");
```

Die Views müssten dann nicht mehr von *Component* erben, sondern nur noch wissen, wie sie sich ein Objekt vom Typ *Component* beschaffen können. Wenn das nicht geht, hätten sie auch die Möglichkeit, *null* zurückzugeben, was auch einen erlaubten Wert für die Methode *showMessageDialog(..)* darstellt. Vielleicht wird der Programmierer aber auch entscheiden, dass die Tatsache, dass das Model überhaupt selbst mit dem Benutzer in Kontakt tritt bereits eine architektonische Schwäche darstellt. In diesem Fall könnte man die Fehlermeldung, wenn sie für den Benutzer verzichtbar ist, einfach an ein Logging-System übergeben. Muss der Benutzer informiert werden, könnte man die Aufgabe deligieren. Dazu könnte man z.B. zunächst sicherstellen, dass nur der Controller schreibend auf das Model zugreift. Wenn man nun annimmt, dass der Fehler nur durch Aktionen (also Methodenaufrufe) des Controllers auftreten kann, kann man ihn mit dem Java-Exception-Mechanismus an den Controller zurückpropagieren. Dieser kennt dann den aktuellen View, und kann diesen veranlassen, eine angemessene Fehlermeldung anzuzeigen.

Alles in allem können also einige komplexe Überlegungen erforderlich sein, um das gewünschte Ziel zu erreichen. Die Analyse ist hier keine große Hilfe, insbesondere weil wiederum das entscheidende Deklarationselement nicht angezeigt wird, und man auch keine Rückmeldung erhält, warum das so ist.

Man könnte natürlich einwenden, dass die genannten Tasks eben nicht zu den Einsatzbereichen, bzw. Entwurfszielen des Tools gehören. Aber

oftmals weiß man zu Anfang nicht, ob eine Lösung neue Interfaces oder eben andere Maßnahmen erfordert. Auch wenn ein Tool nicht jede beliebige Aktivität unterstützen kann (und sollte) erscheint es dennoch unbefriedigend, dass man in den genannten Fällen keinen Hinweis erhält, was man zusätzlich oder alternativ zur Einführung neuer Interfaces vielleicht tun muss.

5.3.2 Die Anwendung auf ganze Projekte

Auch hier gilt natürlich die Einschränkung durch die Unzuverlässikeit der Messergebnisse, allerdings fällt das vermutlich für einen Vergleich zwischen den Vorschlägen des MIP und IISP schwächer ins Gewicht, da beide auf der gleichen, potentiell unkorrekten Datenbasis operieren. Das vorweggeschickt, kann man zumindest erkennen, dass die Zahl der neuen Typen wirkungsvoll verringert wurde. Es werden fast nie mehr als drei Interfaces für einen Typen vorgeschlagen, was praktikabler wirkt als die extrem vielen Typen, die sich bei der Einführung maximal kontextspezifischer Interfaces ergeben. Demgegenüber steht aber, dass der ACD-Wert bei den Typen, die durch den IISP vorgeschlagen wurden, auch nur viel weniger verbessert wurde. Dadurch ist im Verhältnis von Entkopplung zu Wachstum, also der Decoupling/Growth Ratio ein nur kleiner, aber immerhin feststellbarer Zuwachs zu verzeichnen. Wie schon erwähnt, bilden die Verringerung des ACD-Wertes und das Zahl der neuen Typen einen Tradeoff, da eine starke Verbesserung des ACD-Wertes die Einführung von besonders speziellen, also vielen neuen Typen erfordert.

Bei der Decoupling/Growth Ratio werden die Werte einfach geteilt, es wird also keine spezielle Gewichtung[49] vorgenommen. Wie könnte eine solche Gewichtung nun das Ergebnis beeinflussen?

Grundsätzlich kann man mit einer Gewichtung festlegen, was größere Bedeutung haben soll: Die Verbesserung des ACD-Wertes oder die Zahl der neuen Typen zu begrenzen. Das Ergebnis muss irgendwo zwischen den beiden Extremen liegen: Entweder ein Interface pro Access Set, also maximal kontextspezifisch, oder (höchstens) ein Interface pro Typ. Was ist nun erstrebenswert? Dies ist offenbar von Fall zu Fall verschieden, je nachdem was mit der konkreten Entkopplung erreicht werden soll. Manchmal ist sicher die Erzeugung möglichst weniger neuer Typen im Vordergrund, manchmal sind dagegen möglichst schmale Schnittstellen wichtiger. Bei der Anwendung auf ganze Projekte wird kein konkretes Entkopplungsziel verfolgt; es geht ja vielmehr darum, das typische Verhalten eines Algorithmus kennen zu lernen. Deswegen wurde bei der Decoupling/Growth Ratio, die sich ja immer auf ein

[49] Diese Gewichtung ist im Prinzip vergleichbar mit der in Kap.6.2.2 beschriebenen Gewichtung der Metrik. Während aber die Gewichtung bei der Decoupling/Growth Ratio das Ergebnis einer Projektweiten Analyse betrifft, bezieht sich eine Gewichtung der Metrik auf die Vorschlagsfunktion für Interfaces und Interface Sets.

ganzes Projekt bezieht, auf eine Gewichtung verzichtet. Im konkreten Fall kann die Gewichtung aber Sinn ergeben, diese müsste aber bei der Metrik ansetzen, wie in Kap.6.2.2 beschrieben.

5.4 Usability

In Punkto Bedienbarbeit springen einige Probleme sofort ins Auge:

- Die Analyse durch den TYPE ACCESS ANALYZER dauert unangenehm lange.
- Extrem störend sind natürlich auch die unter 5.2.1 genannten Probleme.
- Die verwendeten Begriffe und auch die Form der grafischen Darstellung sind für die Zielgruppe *Programmierer* nicht vertraut, sondern müssen erst erlernt werden.
- Das Konzept, erst mal funktionierenden Code zu schreiben, und sich dann daraus Interfaces ableiten zu lassen, kann man ja einem agilen Ansatz zuordnen. Die Benutzung des Interface-Designers gestaltet sich aber nicht sehr „lightweight“. Immerhin ist es ein kompliziertes Tool, das nur eine sehr spezielle Nischenfunktion abdeckt. Dagegen verlangt es aber einen hohen Lernaufwand, braucht viel Zeit, und präsentiert sehr viel Information, die teilweise für das gewünschte Ziel gar nicht von Bedeutung ist. Dadurch ist bei dem momentanen Status des Projekts für einen Programmierer nicht sofort ersichtlich, wo der Nutzen für ihn liegt, der den Lernaufwand aufwiegen würde.
- Ein unschöner Effekt entsteht durch die vorgegebene Anforderung, dass mit Supertypen deklarierte Elemente nicht erfasst werden sollen. Dadurch verschwinden Deklarationselemente, die mit neuen Interfaces umdeklariert werden, danach aus dem Blickwinkel des Interface-Designers, er kann also das Ergebnis seiner eigenen Arbeit nicht kontrollieren.

5.5 verwandte Arbeiten

Die ersten Beiträge zu den zugrundeliegenden Refactorings finden sich bei Opdyke ([23]), in Opdyke et al. ([24]) wird sogar bereits die Extraktion abstrakter Superklassen behandelt, die man als das Pendant zu Java-Interfaces in C++ betrachten kann. Das *Extract Interface* Refactoring findet erste Erwähnung in dem mehrfach erwähnten Buch von Fowler ([11]), das neben einer umfangreichen Sammlung von Refactorings insbesondere wertvolle Diskussionen zum generellen Nutzen und praktischen Aspekten von Refactorings enthält. Die Optimierung der Typhierarchie unter Verwendung von bereits existierenden Typen ist Thema von ([3],[13],[41]); [2] beschreibt eine Implementierung für Eclipse, die derartige Optimierungsmöglichkeiten im Quelltext anzeigt und Quickfixes bereitstellt. Dabei können auch durch eine

Einbindung von Infer Type kontextspezifische Typen neu erzeugt werden. Eine generelle Einführung in Infer Type, das sicherlich die engste Verwandschaft mit dem in dieser Arbeit beschriebenen Ansatz aufweist, findet sich in [32]. Dort und in [33] gibt der Autor auch einen Überblick über andere Typinferenzalgorithmen, von denen besonders der in Eclipse verwendete Constraint-basierte Ansatz [41] hervorzuheben ist. Auf diesem basiert auch eine neuere Version von Infer Type [20]. Ein weiterer Typinferenzalgorithmus, der zumindest auf den ersten Blick große Ähnlichkeit aufweist, ist KABA ([38],[39]). Auch hier ist das Ziel, eine Klassenhierarchie anhand einer Analyse der tatsächlichen Nutzung zu optimieren. Zudem wird hier ebenfalls ein grafischer Lattice anhand der Typnutzung erzeugt, der allerdings anders aufgebaut ist als der Subprotokoll Lattice. Auch sind die Ziele bei KABA ganz anders gelagert: Es geht darum, möglichst kleine Klassen zu generieren, zu diesem Zweck werden von den analysierten Klassen Subklassen gebildet, und die benötigten Methoden in der so entstandenen Hierarchie so weit wie möglich nach unten verschoben. So wird für jede unterschiedliche Typnutzung eine genau passende Klasse erzeugt. Auf diese Weise wird der Speicherbedarf der Objekte verringert; Entkopplung gehört aber nicht zu den Zielen von KABA. Im Gegenteil werden nach [33] die Abhängigkeiten eher verstärkt, da jeder Client nun von einer hochspeziellen Subklasse abhängig ist. Im Vergleich dazu erscheint der von Infer Type und dem Interface-Designer verwendete Ansatz, von konkreten Typen zu abstrahieren, indem neue Interfaces eingeführt werden, eher entgegengesetzt. Da die Interfaces nicht instanziert werden können, wird an der Instanzerzeugung im Gegensatz zu KABA nichts verändert.

Zusammenfassend kann man sagen, dass KABA das Ziel verfolgt, die tatsächlich auftretenden Objekte zu minimieren, Infer Type und auch der hier vorgestellte Interface-Designer streben dagegen an, die deklarierten Typen von Deklarationselementen zu minimieren bzw. zu verkleinern und so die Flexibilität des Codes zu erhöhen.

Zum Thema Metriken ist die kritische Auseinandersetzung in [15] zu empfehlen, als Vorgehensweise wird wegen der Schwierigkeiten, Metriken nutzbringend zu interpretieren, häufig der *Goal Question Metric Approach* nach [4] verwendet. Für das Konzept, die Zahl der neuen Typen mit Hilfe einer bewertenden Metrik zu begrenzen, ist bisher noch kein Beispiel bekannt, es gibt allerdings mehrere Ansätze, die Metriken zur Erkennung eines „bad smell“ einsetzen, und dadurch Refactorings anstoßen. In [6] wird ein Eclipse-Framework vorgestellt, das es ermöglicht, Smell-Detektoren über einen Erweiterungspunkt einzubinden, und so verschiedene Arten von „Smells“ zu erkennen. Der Detektor hat dabei auch die Möglichkeit, gleich eine Lösung des Problems in Form von Quickfixes anzubieten.

6 Schlussbetrachtungen

6.1 Zusammenfassung

Das Ziel dieser Arbeit war, zusätzliche Werkzeuge bereitzustellen, die den Programmierer bei der Einführung von Interfaces unterstützen. Durch die Konzeption als erweiterbares Framework sollen diese Werkzeuge auch eine Basis für zukünftige Erweiterungen darstellen.

Dazu wurde anhand eines Fallbeispiels zunächst konstatiert, welche Unterstützung in diesem Bereich bereits existiert. Es zeigte sich, dass in bestimmten Situationen ein Werkzeug wünschenswert wäre, was einerseits auf einer Analyse der Nutzung aufbaut wie Infer Type, aber gleichzeitig einen Überblick über die gesamte Nutzung eines Typs, unabhängig von einem einzelnen Deklarationselement, gestattet. So sollte der Benutzer in die Lage versetzt werden, auf informierte Weise eine Abwägung zu treffen zwischen der Einführung sehr spezieller oder eher allgemeiner neuer Interfaces.

Ein solches Werkzeug wurde dann in Form des Interface-Designers als Eclipse Erweiterung implementiert. Um den Programmierer bei der beschriebenen *Abwägung* noch weiter zu unterstützen, wurde auch eine Funktion zur Automatisierung dieser Abwägung implementiert. Die Funktion basiert auf einer Metrik und einem Algorithmus, die beide über den Erweiterungsmechanismus von Eclipse angebunden wurden. Auf diese Weise kann der Benutzer sie gegen eigene Implementierungen austauschen.

Die mitgelieferten Beispielimplementierungen für Metrik und Algorithmus wurden dann an einigen Fallbeispielen getestet. Um einen Eindruck vom Verhalten der Automatik zu bekommen, wurde sie auch auf komplette Projekte angewendet und die Ergebnisse verschiedener Algorithmen verglichen.

6.2 Ausblick und mögliche Weiterentwicklung

6.2.1 Integration einer Constraint-basierten Analyse

Die wichtigste Weiterentwicklung dieses Projekts besteht natürlich darin, die Analyse auf ein neues Verfahren umzustellen, das korrekte Werte liefert. Dies verspricht, die in Kapitel 5.2.1 genannten Probleme zu lösen.

6.2.2 Verbesserungsmöglichkeiten bei der Metrik

Was die implementierte Metrik angeht, so könnte man diese erweitern, indem man sie etwa parametrisierbar macht. Die Einführung neuer Interfaces bewegt sich ja immer innerhalb des Spektrums *maximal kontextspezifisch*, wie etwa bei Infer Type, und dem Anstreben einer maximalen Popularität, wie sie mit der Einführung eines totalen Interfaces erreicht wird. Diese beiden Faktoren bilden auch die Grundlage der Metrik. Man könnte dies nun verfeinern, indem man die Metrik etwa mit einem Schieberegler ausstattet, an

dem man einstellt, wo zwischen diesen beiden Polen das Resultat angeordnet sein soll. So könnte man von Fall zu Fall festlegen, ob eher kontextspezifische oder eher populäre Typen vorgeschlagen werden sollen.

6.2.3 Accessibility

Es wäre schön, wenn es gelänge, die Hürden zur Benutzung der Werkzeuge zu senken. Wenn mehr Programmierer diese Tools benutzen, wäre es möglich, ein Feedback über den Nutzen in realen Projekten zu erhalten, was der Weiterentwicklung sicher zu Gute käme. Um dieses Ziel zu erreichen, wäre es möglicherweise sinnvoll, das Benutzerinterface und den angebotenen Workflow in Hinblick auf einen leichteren Einstieg zu optimieren, indem man sie stärker in die normalen Abläufe integriert. Die Daten einer Nutzungsanalyse könnten z.B. direkt in den Extract Interface Wizard engeblendet werden, indem etwa neben jeder Methode angezeigt wird, wieviel *mehr* Deklarationselemente umdeklariert werden, falls man diese Methode mit in das Interface hineinnimmt. Auch die Benutzung der Fachausdrücke ist in dieser Hinsicht zu überdenken. In den Refactoring Wizards von Eclipse wird beispielsweise nicht der Ausdruck Deklarationselement sondern *occurence of type* verwendet. Als eine Anregung in dieser Richtung kann man die Erweiterung des Refactoring Menüs um den Punkt *Extract Interface by Usage*[50] sehen. Das ist zumindest eine Möglichkeit, den Programmierer am Nutzen der Analyse zu beteiligen, ohne dass er gezwungen wird, sich mit den theoretischen Hintergründen auseinander zu setzen.

6.2.4 Stärkere Ausrichtung auf ein Ziel

Was weiterhin sehr vielversprechend erscheint, ist der in [33] erwähnte Ansatz, die Entkopplung stärker zielgerichtet durchzuführen, indem man eine bestimmte *Perspektive* definiert. So könnte man beispielsweise festlegen, dass nur Deklarationselemente in einer bestimmten Klasse oder Package in die Analyse mit einbezogen werden sollen, und so einen bestimmten *Blickwinkel* auf einen Typ erhalten. Statt des in [33] erwähnten *einzelnen Interfaces*, was dann einzuführen sei, könnte man den Vorschlagsalgorithmus des Interface-Designers benutzen, um zu entscheiden, ob *ein oder mehrere* Interfaces sinnvoll sind.

Im Grunde liefe das darauf hinaus, dass der Programmierer Modulgrenzen vorgibt und sich die Entkopplung darauf konzentriert. Da Java keinen echten Modulbegriff vorgibt, könnte alles mögliche als Modulgrenze dienen, ausser den erwähnten Klassen und Packages etwa auch ein Source-Folder[51] oder

[50] Erklärt in der Anleitung in Anhang D.

[51] Source-Folder sind nicht identisch mit Packages. Bei der Angabe des Classpath können u.a. mehrere Verzeichnisse angegeben werden. Die in den Klassen deklarierten Packages korrespondieren dann mit dem Pfad *relativ* zu den angegebenen Verzeichnissen. D.h. Klassen können sich im selben Package

ein Plugin. Denkbar wäre auch, Modulgrenzen über Namenskonventionen oder Annotationen festzulegen. Von diesen Einschränkungen würde auch der Vorschlagsalgorithmus durch die damit erreichte Datenreduktion profitieren.

6.2.5 Der *menschliche Faktor*

Abschließend soll noch ein etwas anderer Aspekt in die Diskussion geworfen werden, der in der Literatur selten oder gar nicht zu finden ist. Ein wesentliches Ziel der Anstrengungen in diesem Bereich ist ja eine leichtere Wartung der refaktorisierten Software. Dabei wird immer automatisch angenommen: Je weniger Code geändert werden muss, desto leichter ist die Wartung. Dies ignoriert aber die Tatsache, dass die Änderung von einem Menschen vorgenommen wird, der den Code erst einmal *verstehen* muss. Für Menschen ist aber der einfachste Weg oft nicht der kürzeste, sondern der, der ihrem Wesen und ihren bisherigen Erfahrungen am ehesten entspricht. In manchen Fällen ist also vielleicht eine weniger optimierte, aber dafür sofort erfassbare Lösung besser.

6.3 Fazit

Die Entwicklung des Tools nach den Vorgaben krankt leider an den Beschränkungen der zugrundeliegenden Analysewerkzeuge, worunter natürlich die Benutzbarkeit und die Aussagekraft der Ergebnisse leidet. Die implementierten Algorithmen glänzen zwar bei bestimmten Beispielen, enttäuschen aber bei anderen. Möglicherweise kann das Experimentieren mit den Erweiterungspunkten hier noch Fortschritte bringen. Das wirkliche Potential liegt aber nach der Meinung des Autors darin, den hier beschriebenen Ansatz mit den im Ausblick genannten anderen Konzepten zu vereinigen und in eine leicht benutzbare Form zu bringen. Die vorliegende Implementierung ist also eher als Zwischenschritt und Experimentierplattform anzusehen.

befinden, obwohl sie in anderen Ordnern auf der Festplatte sind. In der Eclipse IDE entsprechen die „Source-Folder“ diesen Classpath-Einträgen.

Abbildungsverzeichnis

Literatur

[1] B Alpern, A Cocchi, S Fink, D Grove, D Lieber: *Invokeinterface Considered Harmless*
in: Proc. of OOPSLA (2001) 108-124.

[2] Bach, Markus: *Design und Implementierung eines Eclipse-Plug-Ins zur Anzeige von möglichen Typgeneralisierungen im Quelltext*
Masterarbeit, Fernuniversität Hagen, 2007.
`http://www.fernuni-hagen.de/ps/docs/Masterarbeit-Bach.pdf`

[3] I Balaban, F Tip, RM Fuhrer: *Refactoring Support for class library migration*
in: Proc. of OOPSLA (2005) 265-279.

[4] VR Basili,G Caldiera,D Rombach: *The goal question metric approach*
Encyclopedia of Software Engeneering (John Wiley & Sons 1994).

[5] K Beck, M Fowler et al.: *Manifesto for Agile Software Development*
`http://agilemanifesto.org/` (20.04.2007)

[6] A Bhattacharrya, RM Fuhrer: *Smell detection for Eclipse*
in: Proc. of OOPSLA (2004) 22.

[7] J Dean, D Grove, C Chambers: *Optimization of objectoriented programs using static class hierarchy analysis*
in: Proc. of ECOOP (1995) 77-101.

[8] Forster, Florian: *Measuring The Quality Of Inferred Interfaces.*
IWSM/Metrikon 2006
`http://www.fernuni-hagen.de/ps/pubs/interfacequality.pdf`

[9] Forster, Florian: *Cost and Benefit of Rigorous Decoupling with Context-Specific Interfaces.*
in: Proc. of PPPJ (2006) 23-30.
`http://www.fernuni-hagen.de/ps/pubs/pppj2006.pdf`

[10] Fowler, Martin. *Inversion of Control Containers and the Dependency Injection pattern*
`http://www.martinfowler.com/articles/injection.html` (20.04.2007)

[11] Fowler, Martin: *Refactoring: Improving the Design of Existing Code.*
Addison-Wesley 2000.

[12] Fowler, Martin: *The New Methology*
`http://www.martinfowler.com/articles/newMethodologyOriginal.html`
(20.04.2007)

[13] RM Fuhrer, F Tip, A Kiezun, J Dolby, M Keller: *Efficiently refactoring java applications to use generic libraries*
in: Proc. of ECOOP (2005) 71-96.

[14] E Gamma, R Helm,R Johnson,J Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*
Addison-Wesley 1996.

[15] Henderson-Sellers, Brian: *Object-oriented metrics: Measures of complexity*
Prentice Hall 1996.

[16] Henrich, Andreas: *Kurs 01895: Management von Softwareprojekten*
2001 Fernuniversität Hagen, FB Inf., 01895-3-01-S1 (KE1 S.34).

[17] Hoffmann, Marc: *Eclipse Workbench: Using the Selection Service*
`http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html` (20.04.2007)

[18] Die Homepage der IntoJ Suite
`http://www.intoj.org/`

[19] IntoJ Dokumentation
`http://www.intoj.org/doc_itsuite.html`

[20] Kegel, Hannes: *Constraint-basierte Typinferenz für Java 5*
Diplomarbeit, Fernuniversität Hagen, 2007 (in Bearbeitung).

[21] Mayer, Philip: *Analyzing the Use of Interfaces in large OO projects.*
in: OOPSLA (2003) Companion 382-383.

[22] F Nielson, HR Nielson, C Hankin: *Principles of Program Analysis*
2. Auflage, Springer 2005.

[23] Opdyke, William: *Refactoring Object-Oriented Frameworks*
Dissertation, University of Illinois at Urbana-Champaign, 1992.

[24] WF Opdyke, RE Johnson: *Creating abstract superclasses by refactoring*
in: ACM Conf. on Computer Science (1993) 66-73.

[25] Orlov, Michael: *Efficient Generation of Set Partitions*
`http://www.cs.bgu.ac.il/~orlovm/papers/partitions.pdf` (20.04.2007)

[26] DL Parnas: *On the criteria to be used in decomposing systems into modules*
CACM 15:12 (1972) 1053-1058.

[27] J Des Rivieres, W Beaton: *Eclipse Platform Technical Overview*
`http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html` (20.04.2007)

[28] HW Six, M Winter: *Methodische Entwicklung objektorientierter Desktop-Applikationen*
KE 6: Grundlagen des Entwurfs und Grobentwurfs
Fernuniversität in Hagen, Fak. f. Mat. u. Inf., 01793-7-06-S 1

[29] F Steimann, W Siberski, T Kühne: *Towards the Systematic Use of Interfaces in Java Programming*
in: Proc. of PPPJ (2003) 13-17.

[30] Steimann, Friedrich: *Role=Interface: A Merger of Concepts*
Journal of Object-Oriented Programming 14:4 (2001) 23-32.

[31] F Steimann,P Mayer,A Meißner: *Decoupling Classes with Inferred Interfaces*
in: Proc. of SAC (2006) 1404-1408.

[32] Steimann, Friedrich: *The Infer Type Refactoring and its Use for Interface-Based Programming*
in JOT 6:2 Special Issue OOPS Track at SAC 2006, February 2007 99-120.
`http://www.jot.fm/issues/issue_2007_01/article5`

[33] F Steimann, P Mayer: *Type Access Analysis: Towards Informed Interface Design*
(unveröffentlicht)

[34] Steimann, Friedrich: *Kurs 01853: Moderne Programmiertechniken und -methoden*
Kurseinheit 1: Interfacebasierte Programmierung
Fernuniversität in Hagen, FB Inf., 01853-6-01-S 1

[35] F Steimann, P Mayer: *Patterns of interface-based programming*
JOT 4:5 (2005) 75-94.

[36] Friedrich Steimann: Friedrich Steimann: *Formale Modellierung mit Rollen*
Habilitationsschrift, Universität Hannover, 2000.

[37] Die Eclipse Update Site des TAA 2
`http://www.fernuni-hagen.de/ps/prjs/TAA/update/`

[38] Mirko Streckenbach: *KABA — A System for Refactoring Java Programs*
Phd.Thesis, Universität Passau, 2005.

[39] M Streckenbach, G Snelting: *Refactoring class hierarchies with KABA*
in: Proc. of OOPSLA (2004) 315-330.

[40] Sun Microsystems, Inc.: *The Java Language Specification, Third Edition*
`http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html`
(20.04.2007)

[41] F Tip,A Kiezun, D Bäumer: *Refactoring for generalization using type constraints*
in: Proc. of OOPSLA (2003) 13-26.

[42] Wikipedia: *Artikel über Software Metrics*
`http://en.wikipedia.org/wiki/Software_metrics` (22.2.07)

[43] Wikipedia: *Artikel über Mengenpartitionen*
`http://de.wikipedia.org/wiki/Partition_%28Mengenlehre%29`
(03.04.07)

A Die Ergebnisse der ACD Analysen

Hier sind die genauen Ergebnisse der ACD Analysen für die Beispielprojekte aus Kap.4.3 zu finden. Die Tabellen wurden aus Platzgründen so gekürzt, dass sie immer auf einer Seite Platz finden.

Sie wurden erzeugt, indem für jeden Top-Level Typ des Projekts ein optimales Interface Set errechnet wird. Dabei wurde beim ersten Mal die Kombination Minimal Interfaces Proposer & *QualityMetrics1* und beim zweiten Durchlauf die Kombination Iterative Interface Set Proposer & *QualityMetrics1* verwendet.

(Man beachte, dass beim ersten Durchlauf die eingestellte Metrik egal ist, da der Minimal Interfaces Proposer keinen Gebrauch davon macht; sein Interface Set ergibt sich einzig aus der Menge der Access Set.)

Als Datenquelle wurde immer der *Transitive Castful Calculator* aus dem TYPE ACCESS ANALYZER Projekt eingestellt. Dessen Analyseergebnisse sind in bestimmten Sonderfällen unrichtig, also kann auch die Korrektheit der folgenden Daten nicht garantiert werden.

Als Zeitlimit für den *IISP* wurde immer 600 Sekunden eingestellt. Ein höheres Zeitlimit würde die Anzahl der neuen Typen wahrscheinlich leicht erhöhen, weil die Datenreduktion, die der *IISP* anwendet, um unter dem Zeitlimit bleiben zu können, tendenziell dazu führt, dass weniger Typen vorgeschlagen werden. Bei einigen Durchläufen kam es zu Fehlermeldungen, die möglicherweise die Ergebnisse verfälscht haben, siehe dazu Kap.5.2.1.

Analysis of Project JChessBoard-1.5
Using: Minimal Interfaces Proposer and QualityMetrics1

Type Name	Number of Decl. Elem.	Number of New Types	ACD Ø (before)	ACD Ø (after)	ACD Decrease per new type
Move	33	10	0.578	0.091	0.049
VirtualBoard	15	10	0.963	0.133	0.083
GameNode	7	5	0.948	0.000	0.190
BoardConnector	4	2	0.781	0.250	0.266
JChessBoard	4	2	0.998	0.250	0.374
History	3	2	0.986	0.000	0.493
Protocol	1	1	0.111	0.000	0.111
ConnectionListener	1	1	0.286	0.000	0.286
AI	1	1	0.556	0.000	0.556
GameTable	1	1	0.991	0.000	0.991
Chat	0	0	0.000	0.000	0.000
ConnectionIndicator	0	0	0.000	0.000	0.000
Settings	0	0	0.000	0.000	0.000
BoardEditor	0	0	0.000	0.000	0.000
InfoPanel	0	0	0.000	0.000	0.000
PGN	0	0	0.000	0.000	0.000
Notation	0	0	0.000	0.000	0.000
ChessClock	0	0	0.000	0.000	0.000
VisualBoard	0	0	0.000	0.000	0.000

Summary

	before	after		
Number of Types:	19	54 (+184 %)		
Number of Decl. Elem.:	70	(unchanged)		
ACD Ø:	0.745	0.100	Difference:	0.645
Decoupling-Growth Ratio : **0.35**				

Analysis of Project JChessBoard-1.5
Using: Iterative Interface Set Proposer and QualityMetrics1

Type Name	Number of Decl. Elem.	Number of New Types	ACD Ø (before)	ACD Ø (after)	ACD Decrease per new type
Move	33	1	0.578	0.540	0.038
VirtualBoard	15	2	0.963	0.732	0.115
GameNode	7	3	0.948	0.250	0.233
BoardConnector	4	1	0.781	0.417	0.365
JChessBoard	4	2	0.998	0.250	0.374
History	3	2	0.986	0.000	0.493
Protocol	1	1	0.111	0.000	0.111
ConnectionListener	1	1	0.286	0.000	0.286
AI	1	1	0.556	0.000	0.556
GameTable	1	1	0.991	0.000	0.991
Chat	0	0	0.000	0.000	0.000
ConnectionIndicator	0	0	0.000	0.000	0.000
Settings	0	0	0.000	0.000	0.000
BoardEditor	0	0	0.000	0.000	0.000
InfoPanel	0	0	0.000	0.000	0.000
PGN	0	0	0.000	0.000	0.000
Notation	0	0	0.000	0.000	0.000
ChessClock	0	0	0.000	0.000	0.000
VisualBoard	0	0	0.000	0.000	0.000

Summary

	before	after		
Number of Types:	19	34 (+79 %)		
Number of Decl. Elem.:	70	(unchanged)		
ACD Ø:	0.745	0.474	Difference:	0.271
Decoupling-Growth Ratio : **0.34**				

Analysis of Project GoGrinder-1.14-full
Using: Minimal Interfaces Proposer and QualityMetrics1

Type Name	Number of Decl. Elem.	Number of New Types	ACD Ø (before)	ACD Ø (after)	ACD Decrease per new type
ProbData	33	19	0.897	0.061	0.044
NodeMark	29	9	0.866	0.241	0.069
ProbCollection	28	19	0.979	0.036	0.050
WGFNode	28	10	0.667	0.179	0.049
Command	25	6	0.913	0.480	0.072
WGFController	24	3	0.773	0.000	0.258
Board	20	8	0.776	0.150	0.078
SimpleMark	19	3	0.938	0.579	0.120
EditTool	14	1	0.000	0.000	0.000
ProgressDialog	12	5	0.993	0.000	0.199
SGFNode	9	6	0.979	0.333	0.108
SGFParseException	9	1	0.990	0.889	0.101
Selection	8	5	0.438	0.000	0.087
Node	7	1	0.995	0.857	0.137
HighScore	6	3	0.813	0.167	0.215
AddCommand	6	6	0.861	0.000	0.144
Reason	6	4	0.690	0.000	0.173
SelectionStats	5	3	0.400	0.000	0.133
Task	4	1	0.938	0.000	0.938
CompositeCommand	4	2	0.875	0.250	0.313
NodeLabel	4	3	0.896	0.000	0.299
Test	4	2	0.571	0.000	0.286
GobanPanel	4	3	0.996	0.000	0.332
SGFController	3	2	0.647	0.000	0.324
TagList	3	2	0.500	0.000	0.250
Controller	2	1	0.600	0.000	0.600
MoveCommand	2	1	0.889	0.500	0.389
LineMark	2	1	0.857	0.500	0.357
TestAnswer	2	1	0.000	0.000	0.000
WidgetPanel	2	1	0.978	0.000	0.978
ChoiceDialog	2	1	0.997	0.000	0.997
SplashScreen	2	2	0.993	0.000	0.496
TagListener	2	1	0.000	0.000	0.000
ExceptionHandler	1	1	0.000	0.000	0.000
FileUtils	1	1	0.000	0.000	0.000
RemoveCommand	1	1	0.857	0.000	0.857
UndoController	1	1	0.333	0.000	0.333
TagListener	1	1	0.500	0.000	0.500
AllTest	1	1	0.889	0.000	0.889
WGFFrame	1	1	0.983	0.000	0.983
SelectionDialog	1	1	0.980	0.000	0.980
NewVersionDialog	1	1	0.997	0.000	0.997
AboutDialog	1	1	0.997	0.000	0.997
StatusListener	1	1	0.000	0.000	0.000
GS	0	0	0.000	0.000	0.000
Messages	0	0	0.000	0.000	0.000
Main	0	0	0.000	0.000	0.000
RemoveMarkCommand	0	0	0.000	0.000	0.000
NumberTool	0	0	0.000	0.000	0.000
LetterTool	0	0	0.000	0.000	0.000
LocalMoveTool	0	0	0.000	0.000	0.000
RealTool	0	0	0.000	0.000	0.000
TerritoryTool	0	0	0.000	0.000	0.000
MarkTool	0	0	0.000	0.000	0.000
MoveTool	0	0	0.000	0.000	0.000
LabelTool	0	0	0.000	0.000	0.000
FakeTool	0	0	0.000	0.000	0.000
SGFParser	0	0	0.000	0.000	0.000
Validator	0	0	0.000	0.000	0.000
SGFUtils	0	0	0.000	0.000	0.000
WGFParser	0	0	0.000	0.000	0.000
OneTest	0	0	0.000	0.000	0.000
LineTest	0	0	0.000	0.000	0.000
StringChecker	0	0	0.000	0.000	0.000
SequenceTest	0	0	0.000	0.000	0.000
PointTest	0	0	0.000	0.000	0.000
AllLineTest	0	0	0.000	0.000	0.000
OneLineTest	0	0	0.000	0.000	0.000
LineTestAnswer	0	0	0.000	0.000	0.000
MultipleChoiceTest	0	0	0.000	0.000	0.000
OrderedTest	0	0	0.000	0.000	0.000
ProbFrame	0	0	0.000	0.000	0.000
TagCB	0	0	0.000	0.000	0.000
SettingsDialog	0	0	0.000	0.000	0.000
HighScoreDialog	0	0	0.000	0.000	0.000
⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮

Summary

	before	after		
Number of Types:	87	234 (+169 %)		
Number of Decl. Elem.:	341	(unchanged)		
ACD Ø:	0.790	0.182	Difference:	0.608
Decoupling-Growth Ratio : **0.36**				

Analysis of Project GoGrinder-1.14-full
Using: Iterative Interface Set Proposer and QualityMetrics1

Type Name	Number of Decl. Elem.	Number of New Types	ACD Ø (before)	ACD Ø (after)	ACD Decrease per new type
ProbData	33	1	0.897	0.745	0.152
NodeMark	29	2	0.866	0.734	0.066
ProbCollection	28	2	0.979	0.791	0.094
WGFNode	28	1	0.667	0.618	0.049
Command	25	3	0.913	0.700	0.071
WGFController	24	3	0.773	0.000	0.258
Board	20	2	0.776	0.636	0.070
SimpleMark	19	1	0.938	0.658	0.280
EditTool	14	1	0.000	0.000	0.000
ProgressDialog	12	2	0.993	0.306	0.344
SGFNode	9	2	0.979	0.741	0.119
SGFParseException	9	1	0.990	0.889	0.101
Selection	8	1	0.438	0.338	0.099
Node	7	1	0.995	0.857	0.137
HighScore	6	2	0.813	0.333	0.240
AddCommand	6	2	0.861	0.542	0.160
Reason	6	2	0.690	0.333	0.179
SelectionStats	5	1	0.400	0.400	0.000
Task	4	1	0.938	0.000	0.938
CompositeCommand	4	1	0.875	0.500	0.375
NodeLabel	4	1	0.896	0.375	0.521
Test	4	1	0.571	0.400	0.171
GobanPanel	4	1	0.996	0.375	0.621
SGFController	3	2	0.647	0.000	0.324
TagList	3	1	0.500	0.250	0.250
Controller	2	1	0.600	0.000	0.600
MoveCommand	2	1	0.889	0.500	0.389
LineMark	2	1	0.857	0.500	0.357
TestAnswer	2	1	0.000	0.000	0.000
WidgetPanel	2	1	0.978	0.000	0.978
ChoiceDialog	2	1	0.997	0.000	0.997
SplashScreen	2	1	0.993	0.333	0.659
TagListener	2	1	0.000	0.000	0.000
ExceptionHandler	1	1	0.000	0.000	0.000
FileUtils	1	1	0.000	0.000	0.000
RemoveCommand	1	1	0.857	0.000	0.857
UndoController	1	1	0.333	0.000	0.333
TagListener	1	1	0.500	0.000	0.500
AllTest	1	1	0.889	0.000	0.889
WGFFrame	1	1	0.983	0.000	0.983
SelectionDialog	1	1	0.980	0.000	0.980
NewVersionDialog	1	1	0.997	0.000	0.997
AboutDialog	1	1	0.997	0.000	0.997
StatusListener	1	1	0.000	0.000	0.000
GS	0	0	0.000	0.000	0.000
Messages	0	0	0.000	0.000	0.000
Main	0	0	0.000	0.000	0.000
RemoveMarkCommand	0	0	0.000	0.000	0.000
NumberTool	0	0	0.000	0.000	0.000
LetterTool	0	0	0.000	0.000	0.000
LocalMoveTool	0	0	0.000	0.000	0.000
RealTool	0	0	0.000	0.000	0.000
TerritoryTool	0	0	0.000	0.000	0.000
MarkTool	0	0	0.000	0.000	0.000
MoveTool	0	0	0.000	0.000	0.000
LabelTool	0	0	0.000	0.000	0.000
FakeTool	0	0	0.000	0.000	0.000
SGFParser	0	0	0.000	0.000	0.000
Validator	0	0	0.000	0.000	0.000
SGFUtils	0	0	0.000	0.000	0.000
WGFParser	0	0	0.000	0.000	0.000
OneTest	0	0	0.000	0.000	0.000
LineTest	0	0	0.000	0.000	0.000
StringChecker	0	0	0.000	0.000	0.000
SequenceTest	0	0	0.000	0.000	0.000
PointTest	0	0	0.000	0.000	0.000
AllLineTest	0	0	0.000	0.000	0.000
OneLineTest	0	0	0.000	0.000	0.000
LineTestAnswer	0	0	0.000	0.000	0.000
MultipleChoiceTest	0	0	0.000	0.000	0.000
OrderedTest	0	0	0.000	0.000	0.000
ProbFrame	0	0	0.000	0.000	0.000
TagCB	0	0	0.000	0.000	0.000
SettingsDialog	0	0	0.000	0.000	0.000
HighScoreDialog	0	0	0.000	0.000	0.000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Summary

	before	after		
Number of Types:	87	144 (+66 %)		
Number of Decl. Elem.:	341	(unchanged)		
ACD Ø:	0.790	0.511	Difference:	0.279
Decoupling-Growth Ratio : **0.43**				

Analysis of Project mars-src-2.2.7
Using: Minimal Interfaces Proposer and QualityMetrics1

Type Name	Number of Decl. Elem.	Number of New Types	ACD Ø (before)	ACD Ø (after)	ACD Decrease per new type
Service	44	22	0.890	0.182	0.032
Status	38	12	0.436	0.000	0.036
Host	30	11	0.780	0.033	0.068
MarsModel	13	3	0.815	0.000	0.272
StatusChangeEvent	13	2	0.631	0.154	0.238
Controller	10	8	0.821	0.100	0.090
SendExpectProbe	10	3	0.880	0.200	0.227
Plugin	10	4	0.840	0.500	0.085
Probe	7	1	0.829	0.571	0.257
ProbeFactory	6	5	0.611	0.000	0.122
ClientDebugger	6	2	0.167	0.000	0.083
Editor	5	2	0.100	0.000	0.050
Session	5	0	1.000	1.000	0.000
StatusChangeListener	4	1	0.500	0.500	0.000
ProbeListener	4	1	0.500	0.500	0.000
ProbeEvent	4	2	0.813	0.500	0.156
Main	3	2	0.292	0.000	0.146
PluginDisplay	3	2	0.417	0.333	0.042
MarsView	3	2	0.992	0.000	0.496
Queue	3	3	0.476	0.000	0.159
LongOpt	2	1	0.875	0.500	0.375
Getopt	2	2	0.500	0.000	0.250
InvalidServiceTypeException	2	0	1.000	1.000	0.000
MarsModelListener	2	0	1.000	1.000	0.000
NotificationListener	2	0	1.000	1.000	0.000
ClientDebuggerFactory	2	1	0.000	0.000	0.000
Debug	2	1	0.000	0.000	0.000
PluginRegistry	2	2	0.417	0.000	0.208
StatusView	2	1	0.000	0.000	0.000
InvalidDocumentException	1	1	0.818	0.000	0.818
ProbeWorker	1	1	0.941	0.000	0.941
SendExpectClient	1	1	0.143	0.000	0.143
ServiceTreeContextMenuSupport	1	1	0.667	0.000	0.667
ServiceTreeKeyActionSupport	1	0	1.000	1.000	0.000
FaultListContextMenuSupport	1	0	1.000	1.000	0.000
LoadExceptionHandler	1	1	0.000	0.000	0.000
JdbcProbe	1	0	1.000	1.000	0.000
SimpleSmtpClient	1	1	0.875	0.000	0.875
GetoptDemo	0	0	0.000	0.000	0.000
Version	0	0	0.000	0.000	0.000
XmlProbeFactory	0	0	0.000	0.000	0.000
Notifier	0	0	0.000	0.000	0.000
Displayable	0	0	0.000	0.000	0.000
Enableable	0	0	0.000	0.000	0.000
DetailListModel	0	0	0.000	0.000	0.000
ServiceTreeRenderer	0	0	0.000	0.000	0.000
FaultListRenderer	0	0	0.000	0.000	0.000
HostEditorPanel	0	0	0.000	0.000	0.000
ServiceTypeComboBox	0	0	0.000	0.000	0.000
ServiceTreeChangeAdapter	0	0	0.000	0.000	0.000
EditorDialog	0	0	0.000	0.000	0.000
ServiceParamEditor	0	0	0.000	0.000	0.000
StatusChangeThreadAdapter	0	0	0.000	0.000	0.000
FaultListModel	0	0	0.000	0.000	0.000
ChangeListRenderer	0	0	0.000	0.000	0.000
ServiceEditorPanel	0	0	0.000	0.000	0.000
MarsAbstractRenderer	0	0	0.000	0.000	0.000
ChangeListModel	0	0	0.000	0.000	0.000
ProbeThreadAdapter	0	0	0.000	0.000	0.000
ChangeListPanel	0	0	0.000	0.000	0.000
PluginMenu	0	0	0.000	0.000	0.000
ListContextMenuSupport	0	0	0.000	0.000	0.000
KeyActionSupport	0	0	0.000	0.000	0.000
ExtensionLoader	0	0	0.000	0.000	0.000
Worker	0	0	0.000	0.000	0.000
TreeKeyActionSupport	0	0	0.000	0.000	0.000
IconService	0	0	0.000	0.000	0.000
ContextMenuSupport	0	0	0.000	0.000	0.000
TreeContextMenuSupport	0	0	0.000	0.000	0.000
ClientDebuggerPlugin	0	0	0.000	0.000	0.000
SessionEntry	0	0	0.000	0.000	0.000
SessionList	0	0	0.000	0.000	0.000
CSVLogPlugin	0	0	0.000	0.000	0.000
HTTPSProbeFactory	0	0	0.000	0.000	0.000
MarsHTTPSProbe	0	0	0.000	0.000	0.000
⋮	⋮	⋮	⋮	⋮	⋮
⋮	⋮	⋮	⋮	⋮	⋮

Summary

	before	after		
Number of Types:	81	183 (+126 %)		
Number of Decl. Elem.:	248	(unchanged)		
ACD Ø:	0.682	0.181	Difference:	0.500
Decoupling-Growth Ratio : **0.40**				

Analysis of Project mars-src-2.2.7
Using: Iterative Interface Set Proposer and QualityMetrics1

Type Name	Number of Decl. Elem.	Number of New Types	ACD Ø (before)	ACD Ø (after)	ACD Decrease per new type
Service	44	1	0.890	0.768	0.122
Status	38	1	0.436	0.436	0.000
Host	30	2	0.780	0.437	0.171
MarsModel	13	2	0.815	0.256	0.279
StatusChangeEvent	13	1	0.631	0.385	0.246
Controller	10	1	0.821	0.691	0.130
SendExpectProbe	10	2	0.880	0.300	0.290
Plugin	10	2	0.840	0.650	0.095
Probe	7	1	0.829	0.571	0.257
ProbeFactory	6	1	0.611	0.611	0.000
ClientDebugger	6	1	0.167	0.167	0.000
Editor	5	1	0.100	0.100	0.000
Session	5	0	1.000	1.000	0.000
StatusChangeListener	4	1	0.500	0.500	0.000
ProbeListener	4	1	0.500	0.500	0.000
ProbeEvent	4	1	0.813	0.625	0.188
Main	3	1	0.292	0.292	0.000
PluginDisplay	3	1	0.417	0.417	0.000
MarsView	3	1	0.992	0.222	0.770
Queue	3	1	0.476	0.476	0.000
LongOpt	2	1	0.875	0.500	0.375
Getopt	2	1	0.500	0.250	0.250
InvalidServiceTypeException	2	0	1.000	1.000	0.000
MarsModelListener	2	0	1.000	1.000	0.000
NotificationListener	2	0	1.000	1.000	0.000
ClientDebuggerFactory	2	1	0.000	0.000	0.000
Debug	2	1	0.000	0.000	0.000
PluginRegistry	2	1	0.417	0.300	0.117
StatusView	2	1	0.000	0.000	0.000
InvalidDocumentException	1	1	0.818	0.000	0.818
ProbeWorker	1	1	0.941	0.000	0.941
SendExpectClient	1	1	0.143	0.000	0.143
ServiceTreeContextMenuSupport	1	1	0.667	0.000	0.667
ServiceTreeKeyActionSupport	1	0	1.000	1.000	0.000
FaultListContextMenuSupport	1	0	1.000	1.000	0.000
LoadExceptionHandler	1	1	0.000	0.000	0.000
JdbcProbe	1	0	1.000	1.000	0.000
SimpleSmtpClient	1	1	0.875	0.000	0.875
GetoptDemo	0	0	0.000	0.000	0.000
Version	0	0	0.000	0.000	0.000
XmlProbeFactory	0	0	0.000	0.000	0.000
Notifier	0	0	0.000	0.000	0.000
Displayable	0	0	0.000	0.000	0.000
Enableable	0	0	0.000	0.000	0.000
DetailListModel	0	0	0.000	0.000	0.000
ServiceTreeRenderer	0	0	0.000	0.000	0.000
FaultListRenderer	0	0	0.000	0.000	0.000
HostEditorPanel	0	0	0.000	0.000	0.000
ServiceTypeComboBox	0	0	0.000	0.000	0.000
ServiceTreeChangeAdapter	0	0	0.000	0.000	0.000
EditorDialog	0	0	0.000	0.000	0.000
ServiceParamEditor	0	0	0.000	0.000	0.000
StatusChangeThreadAdapter	0	0	0.000	0.000	0.000
FaultListModel	0	0	0.000	0.000	0.000
ChangeListRenderer	0	0	0.000	0.000	0.000
ServiceEditorPanel	0	0	0.000	0.000	0.000
MarsAbstractRenderer	0	0	0.000	0.000	0.000
ChangeListModel	0	0	0.000	0.000	0.000
ProbeThreadAdapter	0	0	0.000	0.000	0.000
ChangeListPanel	0	0	0.000	0.000	0.000
PluginMenu	0	0	0.000	0.000	0.000
ListContextMenuSupport	0	0	0.000	0.000	0.000
KeyActionSupport	0	0	0.000	0.000	0.000
ExtensionLoader	0	0	0.000	0.000	0.000
Worker	0	0	0.000	0.000	0.000
TreeKeyActionSupport	0	0	0.000	0.000	0.000
IconService	0	0	0.000	0.000	0.000
ContextMenuSupport	0	0	0.000	0.000	0.000
TreeContextMenuSupport	0	0	0.000	0.000	0.000
ClientDebuggerPlugin	0	0	0.000	0.000	0.000
SessionEntry	0	0	0.000	0.000	0.000
SessionList	0	0	0.000	0.000	0.000
CSVLogPlugin	0	0	0.000	0.000	0.000
HTTPSProbeFactory	0	0	0.000	0.000	0.000
MarsHTTPSProbe	0	0	0.000	0.000	0.000
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Summary

	before	after		
Number of Types:	81	116 (+43 %)		
Number of Decl. Elem.:	248	(unchanged)		
ACD Ø:	0.682	0.501	Difference:	0.181
Decoupling-Growth Ratio : **0.42**				

Analysis of Project drawswf-src-1.2.9
Using: Minimal Interfaces Proposer and QualityMetrics1

Type Name	Number of Decl. Elem.	Number of New Types	ACD Ø (before)	ACD Ø (after)	ACD Decrease per new type
InStream	59	29	0.779	0.000	0.027
CachableRed	59	21	0.810	0.102	0.034
Color	55	4	0.835	0.200	0.159
Rect	41	5	0.902	0.317	0.117
Instance	34	1	0.324	0.324	0.000
SWFTagTypes	33	13	0.729	0.242	0.037
MultipleGradientPaint	27	3	0.545	0.148	0.132
Frame	26	11	0.846	0.192	0.059
AlphaTransform	25	2	0.978	0.560	0.209
SWFActions	24	6	0.213	0.000	0.036
AlphaColor	24	2	0.991	0.875	0.058
OutStream	23	16	0.742	0.000	0.046
SWFShape	21	5	0.524	0.143	0.076
SoundInfo	15	1	0.917	0.333	0.583
DrawObject	13	6	0.788	0.231	0.093
SWFVectors	13	3	0.077	0.000	0.026
FontDefinition	13	4	0.709	0.308	0.100
CompositeRule	13	2	0.410	0.077	0.167
Font	11	5	0.830	0.273	0.111
Movie	11	4	0.885	0.091	0.198
SoundStreamHead	11	1	0.091	0.091	0.000
DrawObjectList	10	3	0.990	0.100	0.297
Symbol	10	1	0.600	0.600	0.000
DrawingPanel	9	4	0.996	0.000	0.249
AlphaColorChooser	9	5	0.987	0.000	0.197
IIconFactory	9	2	0.500	0.000	0.250
SWFTags	9	1	0.000	0.000	0.000
SVGComposite	9	1	0.972	0.889	0.083
SWFText	8	2	0.021	0.000	0.010
TileStore	8	1	0.375	0.375	0.000
DrawSWFConfig	7	2	0.857	0.429	0.214
SWFWriter	7	0	1.000	1.000	0.000
TileGenerator	7	1	0.000	0.000	0.000
DrawSWFFont	6	2	0.214	0.167	0.024
JGradientChooser	6	2	0.990	0.000	0.495
BestsolutionConfiguration	6	1	0.667	0.333	0.333
Actions	6	0	1.000	1.000	0.000
Placement	6	0	1.000	1.000	0.000
SWFReader	6	2	0.750	0.000	0.375
TileLRUMember	6	2	0.861	0.500	0.181
MainWindow	5	1	0.996	0.000	0.996
RemoteSaveConfig	5	4	0.822	0.000	0.206
Button	5	3	0.800	0.000	0.267
ActionParser	5	2	0.500	0.000	0.250
Matrix	5	1	0.943	0.200	0.743
PadMode	5	0	1.000	1.000	0.000
LRUCache	5	2	0.633	0.000	0.317
AlphaColorJButton	4	1	0.995	0.000	0.995
ImportedSymbol	4	0	1.000	1.000	0.000
Shape	4	1	0.983	0.750	0.233
Text	4	1	0.800	0.000	0.800
PluginLoader	3	2	0.444	0.000	0.222
AboutWindow	3	1	0.996	0.000	0.996
FontDialog	3	3	0.987	0.000	0.329
FontLoader	3	2	0.222	0.000	0.111
MovieClip	3	0	1.000	1.000	0.000
TimeLine	3	2	0.600	0.000	0.300
Transform	3	0	1.000	1.000	0.000
MP3Frame	3	1	0.429	0.000	0.429
ButtonRecord2	3	1	0.889	0.667	0.222
TagWriter	3	0	1.000	1.000	0.000
LinearGradientPaint	3	3	0.519	0.000	0.173
RadialGradientPaint	3	3	0.533	0.000	0.178
PictureDialog	2	1	0.994	0.000	0.994
SplashScreen	2	2	0.992	0.000	0.496
IconProvider	2	0	1.000	1.000	0.000
FlashGenerator	2	1	0.000	0.000	0.000
DrawMenuInterface	2	1	0.000	0.000	0.000
RemoteSaveDialog	2	1	0.996	0.000	0.996
DrawToolbarInterface	2	1	0.500	0.000	0.500
SolidColoredIcon	2	1	0.857	0.000	0.857
Sound	2	0	1.000	1.000	0.000
ExportedSymbol	2	1	0.500	0.500	0.000
TagParser	2	0	1.000	1.000	0.000
ADPCMHelper	2	2	0.700	0.000	0.350
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Summary

	before	after		
Number of Types:	227	472 (+108 %)		
Number of Decl. Elem.:	841	(unchanged)		
ACD Ø:	0.710	0.239	Difference:	0.471
Decoupling-Growth Ratio : **0.44**				

Analysis of Project drawswf-src-1.2.9
Using: Iterative Interface Set Proposer and QualityMetrics1

Type Name	Number of Decl. Elem.	Number of New Types	ACD Ø (before)	ACD Ø (after)	ACD Decrease per new type
InStream	59	1	0.779	0.646	0.133
CachableRed	59	2	0.810	0.687	0.061
Color	55	2	0.835	0.223	0.306
Rect	41	2	0.902	0.374	0.264
Instance	34	1	0.324	0.324	0.000
SWFTagTypes	33	1	0.729	0.729	0.000
MultipleGradientPaint	27	1	0.545	0.208	0.337
Frame	26	1	0.846	0.744	0.103
AlphaTransform	25	1	0.978	0.598	0.380
SWFActions	24	1	0.213	0.213	0.000
AlphaColor	24	2	0.991	0.875	0.058
OutStream	23	1	0.742	0.620	0.122
SWFShape	21	1	0.524	0.345	0.179
SoundInfo	15	1	0.917	0.333	0.583
DrawObject	13	1	0.788	0.718	0.071
SWFVectors	13	1	0.077	0.077	0.000
FontDefinition	13	1	0.709	0.654	0.055
CompositeRule	13	1	0.410	0.115	0.295
Font	11	2	0.830	0.482	0.174
Movie	11	2	0.885	0.314	0.285
SoundStreamHead	11	1	0.091	0.091	0.000
DrawObjectList	10	2	0.990	0.167	0.412
Symbol	10	1	0.600	0.600	0.000
DrawingPanel	9	3	0.996	0.111	0.295
AlphaColorChooser	9	1	0.987	0.289	0.698
IIconFactory	9	2	0.500	0.000	0.250
SWFTags	9	1	0.000	0.000	0.000
SVGComposite	9	1	0.972	0.889	0.083
SWFText	8	1	0.021	0.021	0.000
TileStore	8	1	0.375	0.375	0.000
DrawSWFConfig	7	1	0.857	0.619	0.238
SWFWriter	7	0	1.000	1.000	0.000
TileGenerator	7	1	0.000	0.000	0.000
DrawSWFFont	6	1	0.214	0.214	0.000
JGradientChooser	6	1	0.990	0.111	0.879
BestsolutionConfiguration	6	1	0.667	0.333	0.333
Actions	6	0	1.000	1.000	0.000
Placement	6	0	1.000	1.000	0.000
SWFReader	6	2	0.750	0.000	0.375
TileLRUMember	6	1	0.861	0.583	0.278
MainWindow	5	1	0.996	0.000	0.996
RemoteSaveConfig	5	1	0.822	0.467	0.356
Button	5	2	0.800	0.200	0.300
ActionParser	5	2	0.500	0.000	0.250
Matrix	5	1	0.943	0.200	0.743
PadMode	5	0	1.000	1.000	0.000
LRUCache	5	1	0.633	0.267	0.367
AlphaColorJButton	4	1	0.995	0.000	0.995
ImportedSymbol	4	0	1.000	1.000	0.000
Shape	4	1	0.983	0.750	0.233
Text	4	1	0.800	0.000	0.800
PluginLoader	3	1	0.444	0.444	0.000
AboutWindow	3	1	0.996	0.000	0.996
FontDialog	3	1	0.987	0.389	0.598
FontLoader	3	1	0.222	0.222	0.000
MovieClip	3	0	1.000	1.000	0.000
TimeLine	3	1	0.600	0.333	0.267
Transform	3	0	1.000	1.000	0.000
MP3Frame	3	1	0.429	0.000	0.429
ButtonRecord2	3	1	0.889	0.667	0.222
TagWriter	3	0	1.000	1.000	0.000
LinearGradientPaint	3	1	0.519	0.278	0.241
RadialGradientPaint	3	1	0.533	0.333	0.200
PictureDialog	2	1	0.994	0.000	0.994
SplashScreen	2	1	0.992	0.333	0.659
IconProvider	2	0	1.000	1.000	0.000
FlashGenerator	2	1	0.000	0.000	0.000
DrawMenuInterface	2	1	0.000	0.000	0.000
RemoteSaveDialog	2	1	0.996	0.000	0.996
DrawToolbarInterface	2	1	0.500	0.000	0.500
SolidColoredIcon	2	1	0.857	0.000	0.857
Sound	2	0	1.000	1.000	0.000
ExportedSymbol	2	1	0.500	0.500	0.000
TagParser	2	0	1.000	1.000	0.000
ADPCMHelper	2	1	0.700	0.500	0.200
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.

Summary

	before	after		
Number of Types:	227	326 (+44 %)		
Number of Decl. Elem.:	841	(unchanged)		
ACD Ø:	0.710	0.441	Difference:	0.269

Decoupling-Growth Ratio : **0.62**

B Die IntoJ Suite

Hervorgegeangen aus verschiedenen Abschlussarbeiten der Universität Hannover, stellt die IntoJ Suite[18] eine Sammlung von Werkzeugen zur Analyse und Verbesserung von Code bereit. Hauptziel ist dabei, die interfacebasierte Programmierung zu unterstützen. Ausser den eigentlichen Tools ist auch die mitgelieferte Dokumentation zu beachten, sowie die ebenfalls auf der Homepage vorhandenen für das Projekt relevanten Publikationen. Die – zumindest teilweise – Lektüre ist Vorraussetzung für die Benutzung der Werkzeuge, weil sich sonst die Bedeutung der Spezialausdrücke nicht erschließt.

B.1 Der Type Access Analyzer

Herzstück der IntoJ Suite ist ein Framework zur Code-Analyse, der Type Access Analyzer. Die Analyse selbst ist dabei delegiert an sog. *Calculators*, die über den Eclipse extension-point Mechanismus integriert werden. Es ist also möglich, die Implementierung auszuwechseln, und durch eigene Entwicklungen zu ergänzen. Welcher *Calculator* aktuell benutzt wird, kann in den zur IntoJ Suite gehörenden Preferences eingestellt werden. Näheres zu den Eigenschaften der verschiedenen mitgelieferten Calculators kann man in der IntoJ Dokumentation[19] nachlesen, an dieser Stelle genügt es, festzuhalten, dass der hier verwendete *Transitive Castful Calculator* auf der erwähnten statischen Analyse nach [7] beruht.

Diese *Calculators* haben die Fähigkeit, zu einem gegebenen Typ ein Objekt vom Typ `TAAccessSetLattice` zu liefern, das API zur Ermittlung der Analysedaten bereitstellt. Die gewonnen Daten können dann auf verschiedene Weise benutzt werden – das naheliegendste ist zunächst, sie darzustellen. Dazu bietet die IntoJ Suite zwei Views, die eine grafische und eine tabellarische Ansicht auf dieselben Daten bieten.

Die grafische Ansicht entspricht Abb.6 auf Seite 16, die tabellarische verwendet eine Tabelle, um pro Zeile ein Access Set anzuzeigen. Die Kästen aus Abb.6 geben durch ihre Einfärbung bereits einen Hinweis darauf, ob und wie oft das Subprotokoll das sie repräsentieren, irgendwo im Projekt benutzt wird. Je dunkler das Kästchen, desto häufiger wird das entsprechende Subprotokoll benutzt. Aufschluss über die Methoden im Subprotokoll erhält man, wenn man den Mauszeiger kurz über einem Knoten schweben lässt: Oft

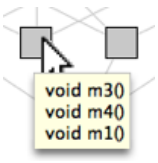


Abbildung 34: Der Tooltip eines Knotens im Subprotokoll Graph

möchte man mehr über einen Knoten, bzw. ein Subprotokoll erfahren, z.B. welche Deklarationselemente es nun konkret benutzen. Dafür gibt es einen weiteren View, den Info View. Dieser zeigt jeweils Details zu dem markierten Knoten und kann zwischen drei Arten von Informationen umgeschaltet werden:

- Methods: Der View zeigt die Methoden des markierten Knotens an, also das, was auch im Tooltip erscheint.
- Declaration Elements: Man sieht die Deklarationselemente, die das markierte Subprotokoll verwenden.
- Matching Types (if any): Sofern Supertypen existieren, die das markierte Subprotokoll verwenden, werden diese hier angezeigt.

Besonders hilfreich ist, dass die im Info View angezeigten Deklarationselemente mit den Stellen im Code, wo sie sich befinden verknüpft sind. Ein Doppelklick auf ein Deklarationselement markiert die entsprechende Stelle im Java-Editor, so dass man sich das Deklarationselement in seinem Programmkontext ansehen kann. Für eine genauere Beschreibung der Benutzung der Views sollte man wiederum die IntoJ Dokumentation[19] konsultieren, die Beschreibung der grafischen Ansicht findet sich auch innerhalb dieser Arbeit als Anhang C.

Es sei darauf hingewiesen, dass es mittlerweile eine neue Version des Type Access Analyzer gibt, die unter [37] heruntergeladen werden kann, und die nicht Teil der IntoJ Suite ist. Sie kommt ohne einen Info View aus, da die entsprechenden Informationen direkt in den entsprechenden Knoten eingeblendet werden können.(Siehe Abb.35) Dadurch kann auf einen extra Info View verzichtet werden. Wir beziehen uns im weiteren jedoch ausschließlich auf den Type Access Analyzer innerhalb der IntoJ Suite. Wer Interesse hat, sich mit dem Type Access Analyzer 2 weiter zu beschäftigen, findet in [33] viele Anregungen zu dessen Benutzung.

B.2 Die Metrics Suite

Ein weiterer Bestandteil der IntoJ Suite ist die Metrics Suite. Sie wird aufgerufen, indem man im Kontextmenu eines Java-projekts den Menüpunkt „Calculate Metrics“

B.3 Infer Type

Im Gegensatz zu den bisher aufgezählten Features bietet Infer Type die Möglichkeit, die gewonnen Analysedaten nicht nur anzuzeigen, sondern zur Verbesserung des Programmcodes zu benutzen. Es bietet ein Refactoring an, das ähnlich wie das bekannte *Extract Interface* aus Eclipse arbeitet. Der Unterschied ist jedoch, dass Infer Type immer auf einem Deklarationselement

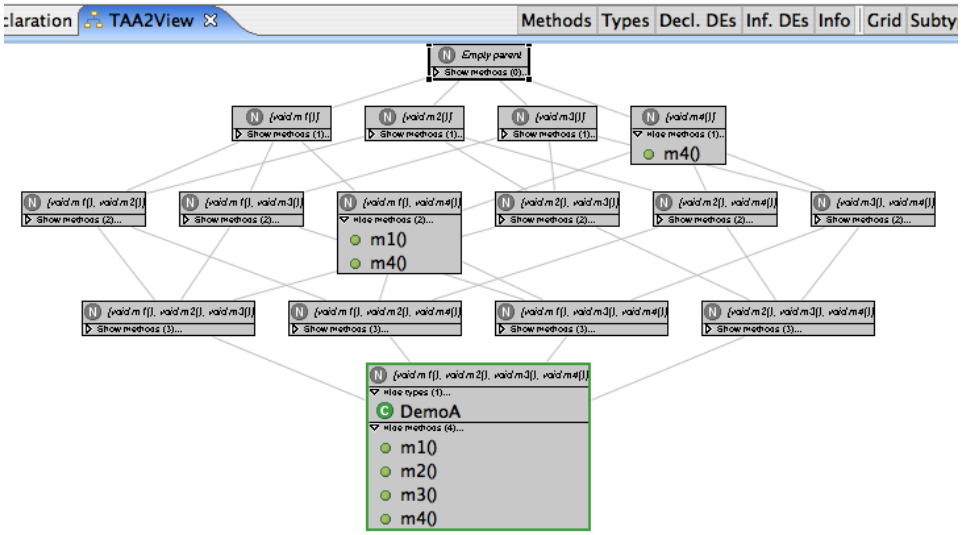


Abbildung 35: Die Klasse DemoA.java im TAA 2 View

aufgerufen wird. Das heißt, man markiert zunächst ein Deklarationselement im Editor und wählt dann *Infer Type* aus dem Kontextmenü oder dem Refactoring Menü.

Dann wird anhand einer statischen Analyse das Access Set dieses Deklarationselements berechnet und dem Benutzer präsentiert. Falls bereits ein oder mehrere Interfaces existieren, die dem gefundenen Access Set genau entsprechen, wird vorgeschlagen eines von diesen zu benutzen. Ansonsten schlägt Infer Type die Einführung eines neuen Interfaces vor, welches genau die Methoden des Access Sets enthält. Es ist also sozusagen auf dieses Deklarationselement „maßgeschneidert“. Ist der Benutzer einverstanden, kann er noch den Namen des Interfaces eingeben, und dann das Refactoring starten. Es besteht im Normalfall[52] aus diesen Schritten:

- Das neue Interface erzeugen.
- Den Typ, mit dem das Deklarationselement bisher deklariert ist, das neue Interface implementieren lassen.
- Das Deklarationselement mit dem neuen Interface umdeklarieren.

Der Mehrwert gegenüber dem normalen *Extract Interface* besteht darin, dass man ein *maximal kontextspezifisches Interface* erhält. Dieses bietet eine maximale Entkopplung für das gegebene Deklarationselement und senkt entsprechend den ACD-Wert des Deklarationselements auf 0. Eine detaillierte Diskussion von Infer Type und der verschiedenen Einsatzmöglichkeiten findet sich unter [32].

[52] Eine exakte Fallunterscheidung findet sich unter [32] und [31].

C Die Dokumentation des Graphical View in der IntoJ Suite

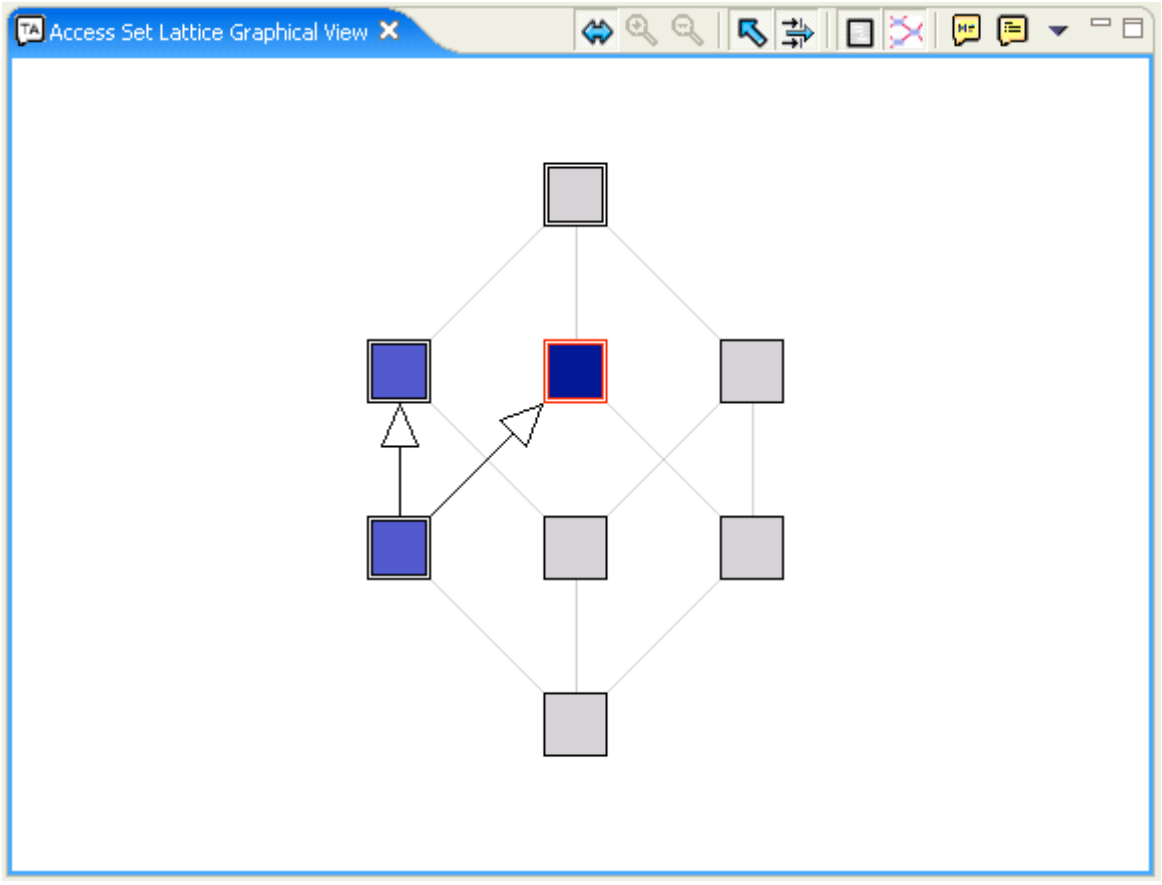
The graphical view

The graphical view contains the same information as the tabular view, only this time in form of a diagram. The graphical view is part of the intoJ perspective, which will be opened automatically when performing type analysis.

Depending on the method count in the type you're analyzing, the full or reduced view of the access sets of your type will be displayed:

- The full view consists of all possible access sets of the current type, that is the complete access set lattice. Unused access sets will be displayed in grey, whereas used access sets will be displayed in different shades of blue, depending on how many variables use the access set - the darker the node, the more variables use the set. The full view is available for types with up to seven methods.
- The reduced view consists of the used access sets only. Different shades of blue are used here, too, indicating how many variables use the access set.

The full view looks like this:



The symbols represent the following:

Unused access set - that is, an access set of this class which theoretically exists but is not used on any set of declaration elements.

Used access set - the darker the shade of blue, the more declaration element sets use this access set.

Access set for which a supertype of the analyzed type covers the declaration elements used in this

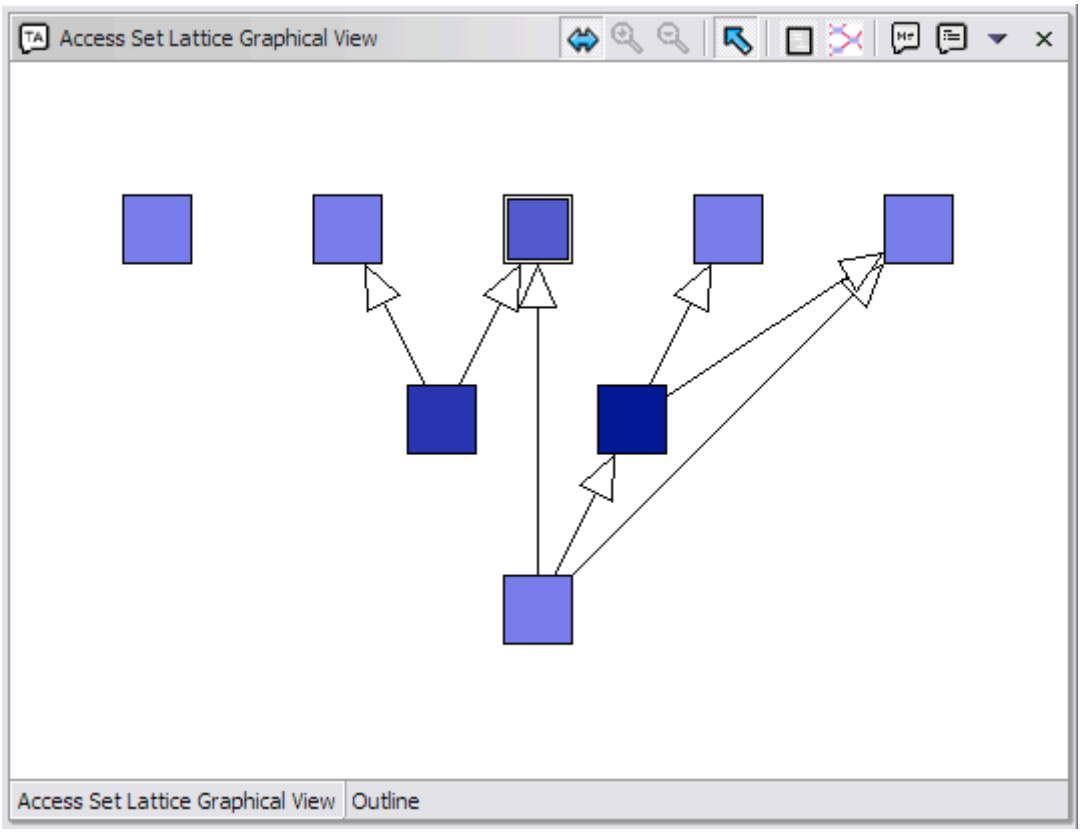
access set.


Access set for which a supertype of the analyzed type contains **exactly** the declaration elements used in this access set.


Relation - indicates that an access set includes all of the methods of another access set. This relation can be seen as the generalisation relation of UML.

While in full view, you may decide to hide either the unused nodes completely or just the grid network using the  buttons (see below).

The reduced view looks like this:



The symbols are the same as in full view, the grid network including the unused access sets, however, is not available due to space limitations.

In both views, you can use the following additional functions to tweak the graphical display (from left to right):

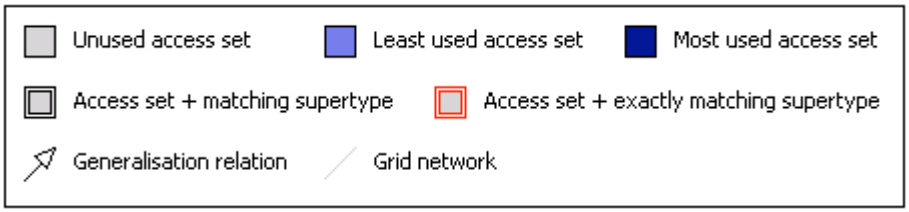
- **Fit in view**: Automatically fits the current access set lattice to the window size.
- **Zoom in**: Enlarges the current access set lattice by 20% (not available when in "fit in view mode")
- **Zoom out**: Reduces the current access set lattice by 20% (not available when in "fit in view mode")

- **Show generalisation relations**: Shows or hides the generalisation relationships between used access sets.
- **Filter generalization relations**: Hides or shows generalisation relationships between unused access sets (only available when in "reduced" mode).

- **Show unused access sets**: Shows or hides unused access sets (only available when in "full" mode)
- **Show grid network**: Shows or hides the grid network between access sets (only available when in "full" mode)

- **Show access set size**: Displays how many methods are used in each layer of the lattice.
- **Show legend**: Displays some information about the graph.

The legend looks like this:



The nodes in the graphical view also support selection. If you select one of the nodes (which represent access sets), its information will be displayed in the info view at the bottom left of the screen. For used access sets, all three elements (declaration elements, invocations, and supertypes) are available; for unused sets, only declaration elements and supertypes will be displayed. In addition to selecting the nodes with your mouse, you can also use the keyboard to navigate the nodes once one is selected.

Note that the graphical and tabular views are synchronized: Whenever you select an access set in one of the views, it will get selected in the other view, too.

D Anleitung des Interface Designers

D.1 Der Aufruf der Analyse

Um die Analyse eines Typs zu starten, wird er im Package Explorer ausgewählt und per Rechtsklick das Kontextmenü aufgerufen.(Abb.36) Aus diesem wählt man den Punkt *Design Interfaces for Type*. Die Analyse die nun ge-

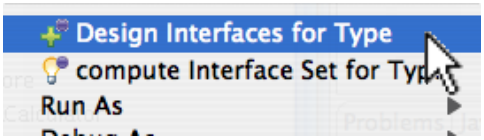


Abbildung 36: Der Aufruf der Analyse im Kontextmenü eines Typs.

startet wird, ist sehr speicher- und rechenintensiv. Es ist daher **unbedingt zu empfehlen**, einen schnellen Rechner mit viel Arbeitsspeicher zu verwenden. Damit der von Eclipse auch genutzt werden kann, muss Eclipse mit entsprechenden Kommandozeilenparametern gestartet werden, z.B.: *-vmargs -Xms400M -Xmx400M*. Während der Analyse wird ein Fortschrittsbalken angezeigt. Ist sie beendet, wird automatisch die *Interface Designer Perspective* aktiviert.

D.2 Die Perspektive und ihre Elemente

Die Elemente der *Interface Designer* Perspektive sind in Abb.37 zu sehen. Wie bei Eclipse üblich, ist eine Perspektive nichts anderes als eine Sammlung von Views in einer bestimmten Anordnung, die gemeinsam eine bestimmte Tätigkeit unterstützen sollen, hier eben den Entwurf von neuen Interfaces. Wenn der Benutzer die Anordnung verändert, Views entfernt oder hinzufügt wird das gespeichert und die Perspektive beim nächsten Mal wieder in der vom Benutzer geänderten Form präsentiert. Es gibt jedoch jederzeit die Möglichkeit, über *Window/Reset Perspective* die Anfangsanordnung wiederherzustellen.

Die *Interface Designer* Perspektive besteht aus den folgenden Elementen:

1. Der Editor-Bereich. Hier werden die vom Benutzer zur Zeit geöffneten Editoren angezeigt.
2. Der *Subprotokoll Lattice Graphical View* bietet eine grafische Darstellung des Subprotokoll Lattice aus Kapitel 2.2.3. Die Darstellung richtet sich nach der Größe des Protokolls des analysierten Typs. Bei Protokollen bis zu einer gewissen Größe werden alle Subprotokolle angzeigt, so dass die typischen, diamantartigen Bilder entstehen. Bei großen Protokollen würde dies jedoch zu unübersichtlich, deshalb werden nur noch die tatsächlich benutzten Subprotokolle, also die *Access Sets* angezeigt.

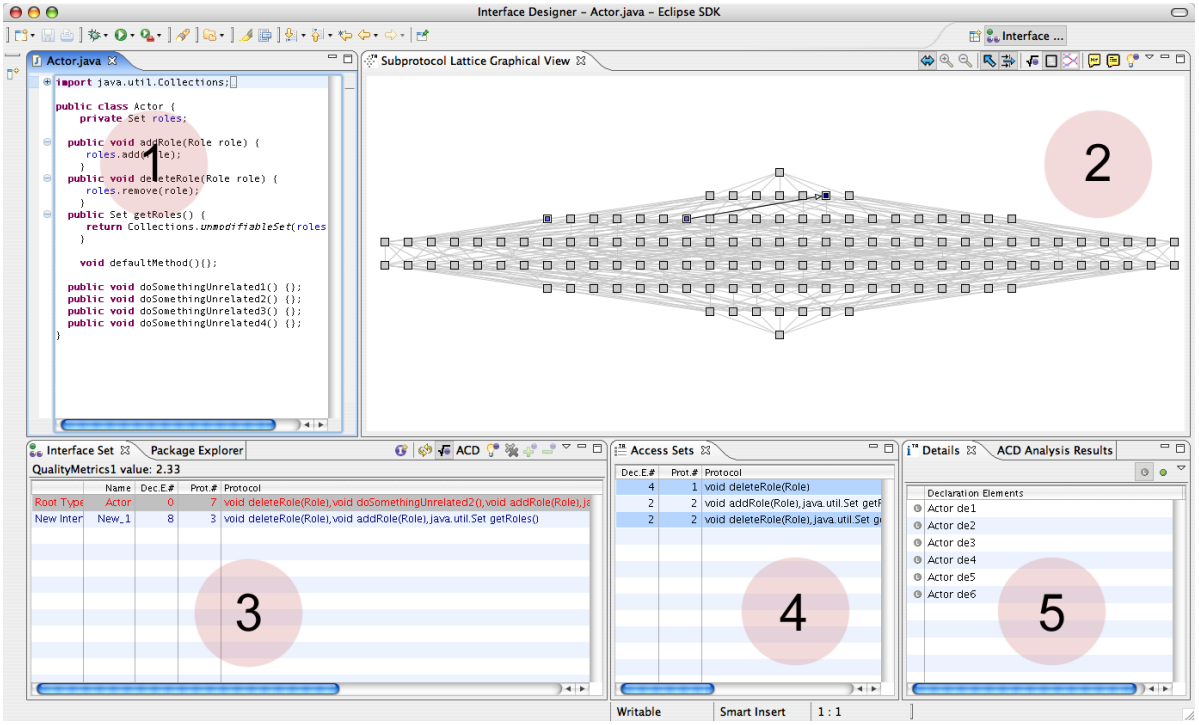


Abbildung 37: Die Elemente der *Interface Designer* Perspektive.

3. Der *Interface Set* View. Hier wird in der ersten Zeile der Basistyp angezeigt, also der Typ, auf dem die Analyse gestartet wurde. Die anderen Zeilen zeigen die bereits entworfenen Interfaces an.
4. Der *Access Sets* View zeigt ähnlich wie der *Subprotokoll Lattice Graphical View* die Analyseergebnisse an, aber nicht in grafischer Form, sondern als Tabelle. Dabei werden nur Access Sets angezeigt, die Anzeige aller Subprotokolle würde zu unübersichtlich.
5. Der *Details* View zeigt zusätzliche Informationen zu den Access Sets bzw. Subprotokollen an, die in einem der Views 2 oder 4 gerade markiert sind. Er kann entweder das Subprotkoll oder die Deklarationselemente anzeigen.

Nun werden die Funktionen der Views im einzelnen erklärt.

D.2.1 Der *Subprotokoll Lattice Graphical View*

Dieser View basiert auf dem *Access Set Lattice Graphical View* aus der IntoJ Suite. Die meisten Eigenschaften und Aktionen sind identisch, Unterschiede ergeben sich eigentlich nur dadurch, dass innerhalb des *Interface Designers* die bereits existierenden Supertypen und Interfaces ausgeblendet werden. Deshalb konzentriert sich diese Anleitung darauf, die *neuen* Elemente und Unterschiede zu beschreiben. Wenn man mit dem Graphical View der

IntoJ Suite nicht vertraut ist, sollte man also zunächst die Anleitung dafür lesen, die als Anhang C beigefügt wurde.

Die neuen Funktionen sind in Abb.38 gekennzeichnet.

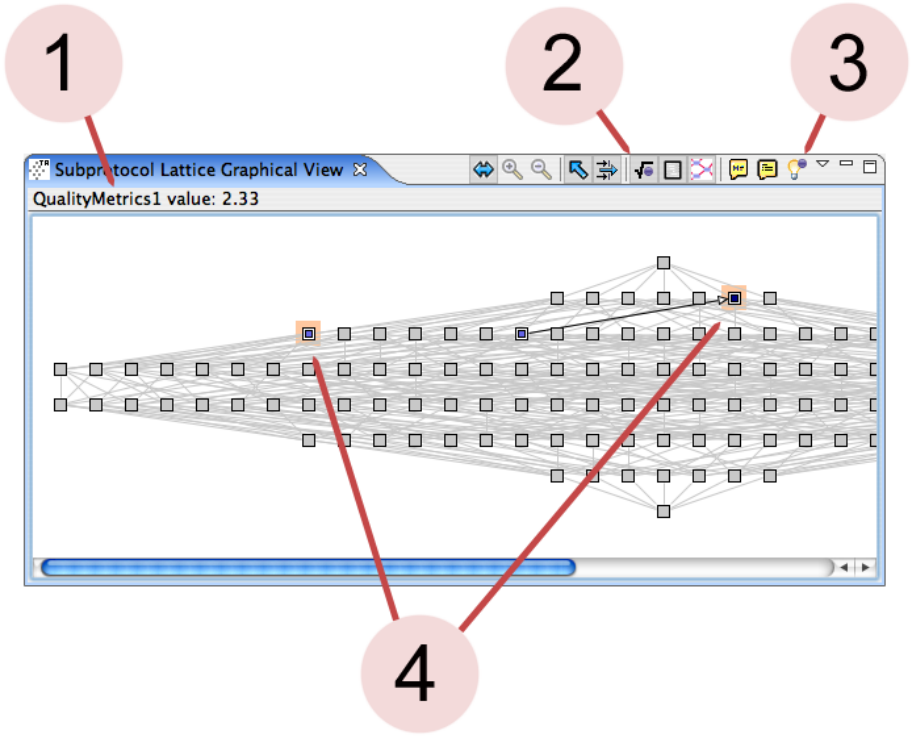


Abbildung 38: Der *Subprotokoll Lattice Graphical View* und seine neuen Funktionen.

4 Eine wichtige neue Eigenschaft ist die Möglichkeit, mehrere Knoten zu markieren. Dazu hält man die Ctrl-Taste gedrückt und klickt mit der linken Maustaste auf die zu markierenden Knoten. Die Markierungen werden automatisch mit den Markierungen im *Access Sets* View synchronisiert.

1 + 2 Es gibt die Möglichkeit, für das durch die momentane Knotenmarkierung beschriebene Interface den Metrikwert der aktuell eingestellten Metrik anzeigen zu lassen. Dabei wird unter (1) der Name der eingestellten Metrik angezeigt, mit dem Button (2) lässt sich die Anzeige ein- und ausblenden.

3 Der Button mit der Glühbirne (3) schlägt automatisch das Interface mit dem höchsten Metrik-Wert vor. Dazu werden alle Möglichkeiten, die Access Sets zu kombinieren durchprobiert, und die mit dem höchsten Metrikwert im Graph markiert. Die Funktion unterscheidet sich von der Vorschlagsfunktion im *Interface Set* View dadurch, dass in diesem View

immer nur ein *einzelnes* Interface gesucht wird. Falls das Interface Set mit dem höchsten Metrikwert nur aus einem Interface besteht, bringen die Funktionen das gleiche Resultat. Gibt es aber ein Interface Set aus mehreren Interfaces[53], das höhere Metrikwerte erreicht als alle einzelnen Interfaces, lässt sich dies nur durch die Funktion im Interface Set View ermitteln. Dauert die Berechnung länger, wird ein Fortschrittsbalken angezeigt. Ist die Anzahl der Access Sets sehr groß, so dass die Berechnung zu lange dauern würde, wird die Anzahl reduziert, indem die weniger aussichtsreichen Access Sets weggelassen werden.

D.2.2 Der *Access Sets* View

Dieser View zeigt ebenfalls das Ergebnis der Analyse an. Die Darstellung beschränkt sich von vorneherein auf die Access Sets, Subprotokolle werden nicht angezeigt. Zu jedem Access Set wird in der ersten Spalte angezeigt, wieviele Deklarationselemente dieses Access Set haben. In der zweiten Spalte steht, wie groß das Protokoll des Access Sets ist, also aus wieviel Methoden es besteht. In der dritten Spalte werden die Methoden selbst gezeigt.

Auch der Access Sets View ermöglicht die Mehrfachselektion mit Ctrl-Mausklick. In ihm markierte Access Sets werden mit den Markierungen im Graphical View synchronisiert.

D.2.3 Der *Details View*

Der *Details* View zeigt Informationen über das (oder die) markierten Access Set(s) an. Sein Inhalt ist also davon abhängig, was im Graphical View oder im Access Sets View gerade markiert ist. Dabei kann der Details View zwei verschiedene Arten von Zusatzinformationen anzeigen:

- Klickt man auf das Methodensymbol, wird das Protokoll des markierten Access Sets angezeigt. Sind mehrere markiert, werden ihre vereinigten Protokolle angezeigt. Diese Information ist im Prinzip auch dem Access Sets View zu entnehmen, die Darstellung im *Details* View ist aber wegen der Anordnung untereinander übersichtlicher.
- Klickt man auf das kleine „L“(Das Eclipse-Symbol für „Language Element“), werden die Deklarationselemente angezeigt, die das markierte Access Set haben. Sind mehrere Access Sets markiert, werden alle Deklarationselemente angezeigt, die eins der markierten Access Sets haben.

[53] Ob es ein solches gibt, hängt natürlich von der verwendeten Metrik ab. Bei der mitgelieferten Beispielimplementierung *QualityMetrics1* wäre das dann der Fall, wenn der Basistyp auf sehr unterschiedliche Art genutzt wird, so dass sich die Einführung partieller Interfaces anbietet, ein Beispiel findet man in Kapitel 4.2.

D.2.4 Der *Interface Set View*

Der Interface Set View ist gewissermaßen das Herzstück des Interface Designers, hier findet der eigentliche Entwurf von Interfaces statt. Wenn ein Typ analysiert wurde, wird er in der obersten Zeile des Views angezeigt, mit der Typbeschreibung „Root Type“ in der ersten Spalte. Im Zuge des Entwurfs werden dann ein oder mehrere neue Interfaces hinzugefügt, die in der ersten Spalte die Bezeichnung „New Interface“ haben. Zu jedem Typ wird in der zweiten Spalte sein *Name* angezeigt. Die neuen Interfaces bekommen bei ihrer Einführung automatisch Namen wie *New_1*, *New_2*, usw. Man kann sie jederzeit umbenennen, indem man auf sie doppelklickt. Das führt zum Dialog aus Abb.39, wo man einen neuen Namen für ein Interface eingeben kann. Dabei werden als Anhaltspunkt für die Namensgebung die Methoden des Interfaces mit angezeigt. In der dritten Spalte des Interface Set Views wird die

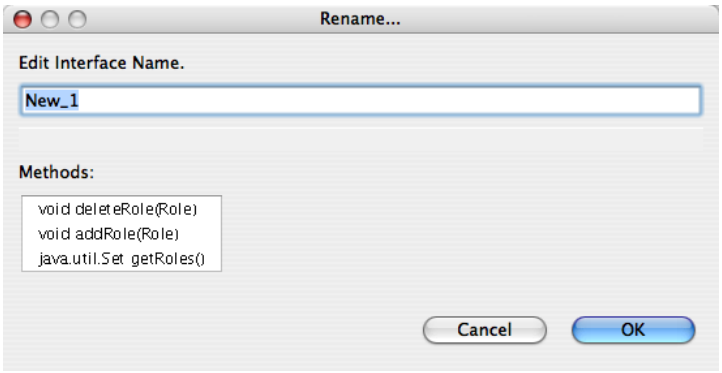


Abbildung 39: Der Dialog zum Benennen eines Interfaces

Anzahl der mit diesem Typ deklarierten, bzw. mit diesem Typ umdeklarierbaren Deklarationselemente angezeigt. Am Anfang sind alle Deklarationselemente mit dem Root Type deklariert. Wenn man neue Interfaces hinzufügt, kann man an dieser Spalte erkennen, wieviele Deklarationselemente mit dem jeweiligen Interface umdeklariert würden, wenn jedes Deklarationselemente mit dem minimalen Interface, das sich im Interface Set befindet, und was das benötigte Protokoll bietet, umdeklariert würde. Man beachte, dass bereits existierende Interfaces oder Supertypen nicht miteinbezogen werden, es wird also davon ausgegangen, dass nur die neu einzuführenden Interfaces aus dem Interface Set View zum Umdeklarieren der Deklarationselemente benutzt werden sollen.

Die nächsten beiden Spalten zeigen dann – ähnlich wie im *Access Sets* View – die Größe des Protokolls und die Methoden des Protokolls an. Wenn man ganz nach rechts scrollt, finden sich dort in einer sechsten Spalte übrigens auch noch die Deklarationselemente. Meistens ist jedoch unpraktikabel, sie in diesem View zu betrachten, besser dafür geeignet ist der Details View.

D.3 Tasks

In diesem Abschnitt wird erklärt, wie man typische Aufgaben mit dem Interface Designer ausführt.

D.3.1 Die integrierte Hilfe benutzen

Über den Menüpunkt *Help/Help Contents* erhält man einen Überblick über alle verfügbaren Hilfe-Dokumente. Unter der Überschrift „IntoJ Suite“ befindet sich die Hilfe zum Interface-Designer.

Tipp: Die Hilfe wird normalerweise in einem externen Browser angezeigt. Auf manchen Systemen ist es aber praktischer, den Help-View von Eclipse zu benutzen. Dieser lässt sich mit *Window/Show View/Other../Help* öffnen.

D.3.2 Interfaces durch Mehrfachselektion definieren

Interfaces hinzufügen Da der Interface-Designer zum Entwurf von Interfaces dient, stellt sich die Frage, wie die Interfaces denn nun „designt“ werden. Normalerweise tut man das ja, etwa im Extract Interface Dialog, indem man von den Methoden des Typs einige aus einer Liste auswählt. Beim Interface-Designer wird jedoch ein anderer Weg beschritten. Das Design der Interfaces stützt sich direkt auf die Analyse der Typnutzung. Das bedeutet: Um ein Interface zu entwerfen, markiert man ein oder mehrere Access Sets im Graphical View oder im Access Sets View. Dann drückt man im Interface Set View auf den „Import selected Subprotocol(s) as new Interface“ Button (Das Pluszeichen). Dadurch wird im Interface Set View ein neues Interface hinzugefügt, dessen Protokoll sich aus der Vereinigung aller Protokolle der markierten Access Sets ergibt. Man kann dann sofort in der Spalte „Dec.E.#“ des Interface Set Views sehen, wie viele der momentan noch mit dem Basistyp deklarierten Deklarationselemente mit dem neuen Interface umdeklariert werden könnten, wenn man es einführt. In dieser Weise können beliebig viele Interfaces hinzugefügt werden.

Interfaces löschen Um ein Interface wieder zu löschen, markiert man es und klickt dann auf den „Remove Interface from Interface Set“ Button (Das Minuszeichen).

alle Interfaces löschen Sollen alle Interfaces aus dem Interface Set gelöscht werden, drückt man dazu auf den Button „Delete All“.

Die Analyse wiederholen Mitunter kommt es vor, dass man während dem Entwurf von Interfaces noch etwas an den Klassen selber ändert. Das macht sogar oft Sinn:

- Der Interface-Designer berücksichtigt keinerlei Optimierungspotential, was durch die Benutzung bereits bestehender Typen in den Deklarationen entsteht. Deswegen könnte man erwägen, *vor* dem Interface-Designer das in [2] beschriebene Werkzeug einzusetzen, was genau dies leistet.
- Deklarationselemente, die nicht unter den verwendeten Protokollbegriff fallen, also die etwas anderes von einem Typ als nicht-statische, mit *public* gekennzeichnete Methoden verwenden, werden bei der Analyse nicht gefunden. Vielleicht möchte man sie aber auch undeklarieren lassen. Dann empfielt es sich, die Zugriffe, die ihren Ausschluss bewirkt haben, durch Zugriffe zu ersetzen die unter den Protokollbegriff fallen. Dazu könnte man etwa Getter- und Settermethoden einführen, die den direkten Zugriff auf Felder ersetzen.

Nach solchen Änderungen ist das Analyseergebnis natürlich nicht mehr aktuell. Durch einen Druck auf den „Analyze this type again“ Button kann man die Analyse für den aktuell angezeigten Typ wiederholen lassen.

D.3.3 Interfaces benennen

Man kann in dieser Phase auch die Interfaces schon benennen. Dazu macht man einen Doppelklick auf das Interface im Interface Set View und gibt in dem anschließend erscheinenden Dialog einen neuen Namen ein.

D.3.4 Interfaces wirklich erzeugen

Ist man mit dem Entwurf fertig und möchte die neuen Interfaces tatsächlich einführen, so drückt man auf den „Create Interfaces“ Button. Es erscheint ein Zwischendialog wie in Abb.40. Die mit *Extract Interface..* beschrifteten Buttons rufen jeweils den Extract Interface Dialog zur Erzeugung des konkreten Typs auf. Man beachte, dass nicht garantiert werden kann, dass die Deklarationselemente danach exakt in der gleichen Art umdeklariert sind, wie dies vorher im Interface Set View angezeigt wurde. Man sollte immer von der Preview-Funktion des Extract Interface Refactorings Gebrauch machen, um sich davon zu überzeugen, dass wirklich das gewünschte Resultat eintritt.

D.3.5 Nur ein Interface erzeugen

Möchte man nur ein einzelnes Interface aus dem Interface Set View erzeugen, so macht man einen Rechtsklick darauf, und wählt in dem erscheinenden Kontextmenü *Create this Type* aus.

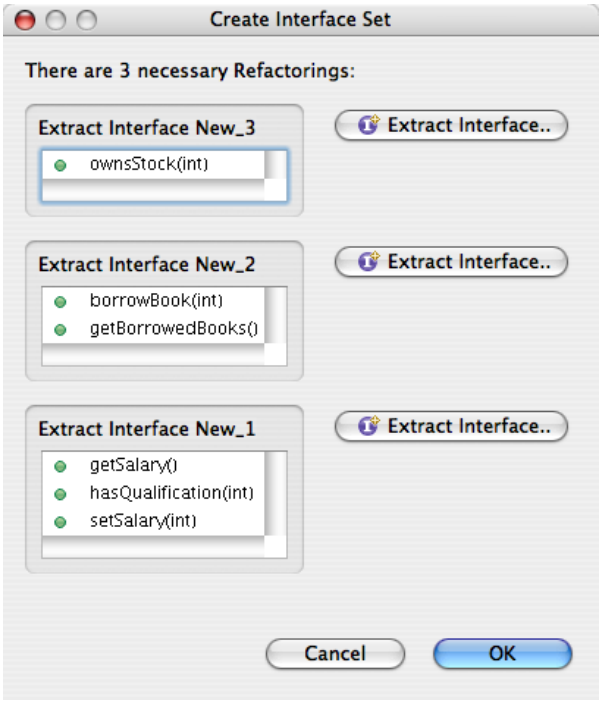


Abbildung 40: Ein Zwischendialog zum mehrfachen Aufruf von Extract Interface..

D.3.6 Metrikwerte anzeigen lassen

Der Interface Set View bietet die Möglichkeit, sich die Metrikwerte für das aktuelle Interface Set anzeigen zu lassen. Ein Druck auf den Button „Show Quality Metrics value“ blendet die Anzeige ein oder aus. Dabei wird links oben der *Name* der eingestellten Metrik angezeigt und der berechnete Wert. Es wird die aktuell in den Preferences eingestellte Metrik verwendet.

Verbesserung des ACD-Wertes anzeigen lassen Ein Druck auf den mit „ACD“ beschrifteten Button zeigt den durchschnittlichen ACD-Wert[54] aller Deklarationselemente an. Dabei bezeichnet „before“ den aktuellen Zustand, und „after“ den voraussichtlichen Zustand, wenn alle neuen Interfaces im Interface Set eingeführt würden. „Average Decrease per new type“ zeigt die durchschnittliche Verbesserung, geteilt durch die Anzahl der neuen Interfaces.

D.3.7 Ungünstige Elemente ausgrauen

Der Interface-Designer View hat ein Feature, was es ermöglicht, bestimmte Elemente der anderen Views auszugrauen. Dazu wählt man im Menü des

[54] Siehe Kapitel 2.3.1

Views *Mark not recommended subprotocols*. Es werden dann alle Subprotokolle ausgegraut, deren Aufnahme in das Interface Set in Form eines neuen Interfaces es ermöglichen würde, dass bereits mit einem neuen Interface deklarierte Deklarationselemente mit ihnen umdeklariert werden könnten. Im Graphical View wird ein entsprechender Hinweis als Tooltip angezeigt. Da diese Funktion potentiell verwirrend ist, ist sie in der Voreinstellung abgeschaltet.

D.3.8 Undo

Da das Eclipse Refactoring verwendet wurde, funktioniert die übliche Methode, die Änderung(en) wieder rückgängig zu machen: Man öffnet den Basistyp im Java-Editor und wählt aus dem *Edit* Menü „undo Extract Interface“ . Auf diese Weise können sämtliche mit dem Refactoring verbundenen Änderungen in einem Schritt rückgängig gemacht werden.

D.3.9 Die Vorschlagsfunktion benutzen

Durch einen Druck auf den „Propose Interface Set“ Button wird automatisch ein Interface Set vorgeschlagen, und im Interface Set View angezeigt. Evt. vorher dort vorhandene Interfaces werden dabei ohne weitere Warnung überschrieben. *Wie* der Vorschlag zustande kommt, hängt vom eingestellten *Proposer* ab. Falls der Proposer Gebrauch von der eingestellten Metrik macht, hängt das Ergebnis auch davon ab.

D.3.10 Direkt ein Interface Set für einen Typ erhalten

Es gibt auch eine Möglichkeit, die Analyse und den Vorschlag eines Interface Sets für eine Typ gleich gemeinsam ausführen zu lassen. Dazu wählt man aus dem Kontextmenü aus Abb. 41 nicht den Punkt „Design Interfaces for Type“ sondern den Punkt „Compute Interface Set for Type“. Das Ergeb-

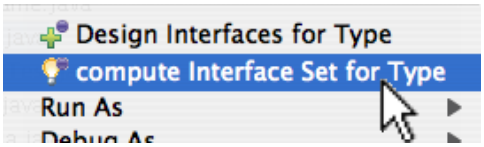


Abbildung 41: Der Aufruf der Analyse im Kontextmenü eines Typs.

nis entspricht dem Ergebnis, wenn man erst die Analyse durchführen lässt und dann die Vorschlagsfunktion benutzt. Die Kombination beider Vorgänge spart jedoch etwas Zeit.

D.3.11 Direkt zum Extract Interface Dialog

Wenn man sich sicher ist, dass man *nur ein* Interface für einen Typ einführen möchte, kann man eine andere Abkürzung verwenden. Man markiert den Typ wie gewohnt im Package Explorer, und wählt dann aus dem Refactor Menü den Punkt *Extract Interface by Usage*(Siehe Abb. 42). Dadurch werden

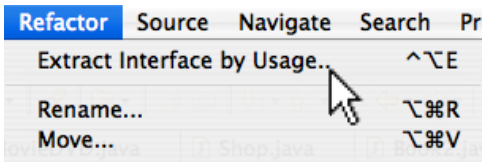


Abbildung 42: Der Aufruf von Extract Interface by Usage.

automatisch folgende Schritte ausgeführt:

- Die Analyse wird durchgeführt.
- Das beste einzelne Interface wird bestimmt. Dabei kommt derselbe Algorithmus zum Einsatz, der beim Graphical View benutzt wird, wenn man auf den Vorschlagsbutton drückt.
- Das Extract Interface Refactoring wird gestartet, und die Methoden des errechneten Interfaces vorselektiert.

Der Vorteil an dieser Methode ist, dass man *nichts* über die ganzen Views zu lernen braucht, um sie nutzbringend anwenden zu können. Der Nachteil ist, dass so immer nur ein Interface vorgeschlagen wird.

D.3.12 Die Voreinstellungen verändern

Da alle wichtigen Algorithmen im Interface-Designer über Extension-Points eingebunden wurden, hängt seine Funktion stark davon ab, welche Einstellungen in den Eclipse *Preferences* vorgenommen werden. Die Einstellungen sind persistent, bleiben also nach dem Beenden von Eclipse erhalten. Zu finden sind die Einstellungen unter *Window/Preferences/intoJ Preferences*. Dort können die folgenden Einstellungen gemacht werden:

- *Calculators* Diese Einstellung stammt noch aus der IntoJ Suite. Es wird empfohlen, hier immer den *Transitive Castful Calculator* eingestellt zu lassen.
- *Interface Quality Metrics* Hier wird die zu verwendende Metrik eingestellt. Im Auslieferungszustand ist nur das Implementierungsbeispiel „QualityMetrics1“ anwählbar. Wenn der entsprechende Erweiterungspunkt erweitert wird[55], erscheinen hier weitere Metriken. Die hier eingestellte Metrik wird an allen Stellen verwendet, wo auf Metrikwerte

[55]Siehe Kap.3.5.

zugegriffen wird, sei es um sie direkt anzuzeigen oder um z.B. ein Interface zu berechnen.

- *Interface Set Proposer* Hier wird der Algorithmus für die Vorschlagsautomatik eingestellt. Dieser Algorithmus wird gestartet, wenn die Vorschlagsfunktion durch einen Druck auf den Button „Propose Interface Set“ (die Glühbirne) im Interface Set View gestartet wird. Man beachte: Die Vorschlagsfunktion im Graphical View macht nicht von diesem Algorithmus Gebrauch, sondern benutzt einen anderen, einfacheren Algorithmus, der nicht verändert werden kann. Er macht allerdings auch von der eingestellten Metrik Gebrauch, ist also in seinem Verhalten von ihr abhängig. Auch bei den Proposern ist es so, dass eigene Erweiterungen in diesem Dialog erscheinen und anwählbar sind.

D.3.13 Die Vorschlagsfunktion auf ein ganzes Projekt anwenden

Wenn man, wie in Kap.3.5 beschrieben, einen oder beide Erweiterungspunkte benutzt hat, um eigene Algorithmen hinzuzufügen, möchte man sich sicher schnell ein Bild von ihrer Wirkung machen, um z.B. zu sehen, wie viele Interfaces der eigene Proposer pro Klasse vorschlägt, ob es bei bestimmten Klassen „Aussreißer“ gibt, o.ä.

um diese Evaluation effizienter zu gestalten, gibt es die Möglichkeit, die Vorschlagsfunktion gleich auf alle Klassen eines Projektes anzuwenden. Dabei werden nur die Top-Level Klassen in Betracht gezogen, weil normalerweise eine Entkopplung der anderen Typen nicht sinnvoll ist, und die Anzahl der neu eingeführten Typen nur grundlos erhöht.

Man startet die Vorschlagsfunktion für ein ganzes Projekt, indem man im Package Explorer nun ein *Projekt* selektiert, und im Kontextmenü „Analyze possible ACD decrease“ anwählt. Je nach eingestelltem Proposer kann es sein, dass man noch nach Rahmenwerten für die Berechnung gefragt wird, siehe Abb.43. Dann werden alle Top-Level Typen des Projekts analysiert,

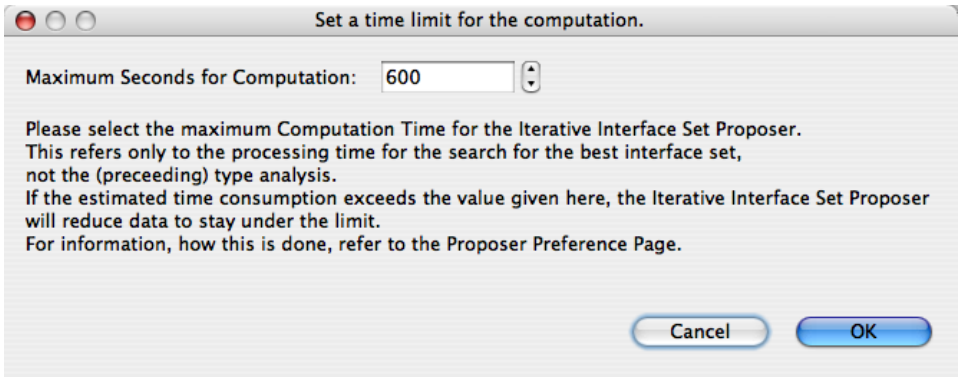
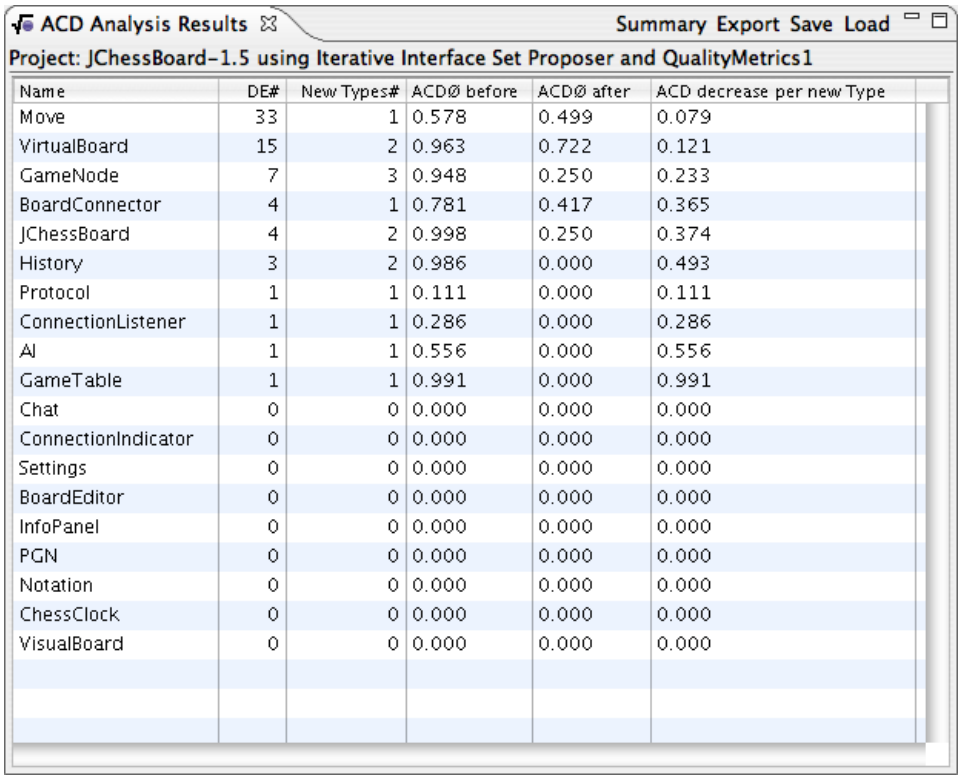


Abbildung 43: Der Dialog des Iterative Interface Set Proposers, wenn er im Batch-Mode gestartet wird.

und für jedes ein Interface Set mithilfe des aktuell eingestellten Proposers und, sofern der Proposer davon Gebrauch macht, mittels der eingestellten Metrik berechnet. Die Anzeige der Ergebnisse erfolgt in einem extra View, dem *ACD Analysis Results* View. Während der Analyse werden die gerade analysierten Typen in den verschiedenen Ansichten dargestellt.

D.3.14 Der *ACD Analysis Results* View

Dieser View gestattet eine Übersicht über alle Typen eines Projekts (siehe Abb.44). Die Typen werden nach der Anzahl der Deklarationselemente,



Name	DE#	New Types#	ACDØ before	ACDØ after	ACD decrease per new Type
Move	33	1	0.578	0.499	0.079
VirtualBoard	15	2	0.963	0.722	0.121
GameNode	7	3	0.948	0.250	0.233
BoardConnector	4	1	0.781	0.417	0.365
JChessBoard	4	2	0.998	0.250	0.374
History	3	2	0.986	0.000	0.493
Protocol	1	1	0.111	0.000	0.111
ConnectionListener	1	1	0.286	0.000	0.286
AI	1	1	0.556	0.000	0.556
GameTable	1	1	0.991	0.000	0.991
Chat	0	0	0.000	0.000	0.000
ConnectionIndicator	0	0	0.000	0.000	0.000
Settings	0	0	0.000	0.000	0.000
BoardEditor	0	0	0.000	0.000	0.000
InfoPanel	0	0	0.000	0.000	0.000
PGN	0	0	0.000	0.000	0.000
Notation	0	0	0.000	0.000	0.000
ChessClock	0	0	0.000	0.000	0.000
VisualBoard	0	0	0.000	0.000	0.000

Abbildung 44: Der *ACD Analysis Results* View

die mit ihnen deklariert sind sortiert angezeigt, so dass man schnell einen Eindruck bekommt, welches die populärsten Typen im Projekt sind. In den Spalten werden verschiedene Informationen zu jedem Typ angezeigt:

- Spalte 1 zeigt den Namen des Typs.
- Spalte 2 zeigt die Anzahl der mit diesem Typ deklarierten Deklarationselemente.
- Spalte 3 zeigt die Anzahl der neuen Typen, die die Vorschlagsautomatik für diesen Typ vorgeschlagen hat.

- Spalten 4,5 und 6 zeigen den durchschnittlichen ACD-Wert der Deklarationselemente vor und nach der Einführung der neuen Interfaces, sowie die Differenz beider Werte geteilt durch die Anzahl der neuen Typen.

Der View bietet darüberhinaus einige auf diesen Daten ausführbare Aktionen an:

- Durch einen Doppelklick auf einen Typ führt man die Analyse auf diesem Typ erneut aus. Danach wird er in allen Views und im Java-Editor angezeigt. Dies ist also gut geeignet, um einen View, der in der Übersicht aufgefallen ist, etwas genauer „unter die Lupe zu nehmen".
- *Save* und *Load* ermöglichen es, die kompletten Analyseergebnisse in einer Datei zu speichern und zu laden. Das ist angesichts der langen Berechnungsdauer sehr hilfreich, um etwa Ergebnisse verschiedener Proposer miteinander zu vergleichen. Die Verknüpfungen mit dem Quelltext bleiben auch beim Speichern und Laden erhalten, sofern wieder in den selben Workspace geladen wird, und die Quelltexte noch an den selben Orten auffindbar sind.
- *Summary* zeigt eine Zusammenfassung der Daten und einige Durchschnittswerte und Verhältnisse.
- *Export* ermöglicht es, die Daten als LaTeX-Tabelle zu exportieren. So können sie leicht in eine druckbare Form (z.B. PDF) gebracht werden. Die TeX-Datei enthält all Daten, aber ab der 75. Zeile sind sie auskommentiert, um gut auf einer Seite Platz zu finden. Die Daten der *Summary* werden jeweils unten in einer Zusatztabelle angehängt. Die Tabellen in Anhang A sind mit dieser Funktion erstellt worden.