Entwicklung eines Analysewerkzeugs zur Verwaltung von Modulabhängigkeiten in Softwareprojekten

Diplomarbeit im Studiengang Wirtschaftsinformatik an der Fachhochschule Köln, Abt. Gummersbach

Jan Ploski (11024993)

6. August 2003

Betreuer: Prof. Dr. Erich Ehses

Zweitprüfer: Prof. Dr. Frank Victor

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Unpraktischer Softwareentwurf

Es ist disputabel, ob die Softwareentwicklung heute einfacher oder schwieriger als in den Pionierzeiten der Programmierung ist. Einerseits verfügt der moderne Entwickler über eine Reihe von Methoden, Werkzeugen und anderen Hilfsmitteln, die sich durch ihre jahrelange Verwendung bei der Bewältigung von Programmieraufgaben bewährt haben. Andererseits ist die Komplexität der zu entwickelnden Systeme im Vergleich zu älteren Zeiten wesentlich gestiegen. Obwohl die Menge des zur Erledigung einer bestimmten Aufgabestellung erforderlichen Wissens ungefähr gleich groß geblieben ist, ist man heute auch bei kleineren Projekten auf die Verwendung einer beträchtlichen Anzahl von Softwarekomponenten angewiesen, die ihrerseits wieder von weiteren Komponenten Gebrauch machen usw. Dementsprechend wird die Qualität der an Endkunden gelieferten Produkte durch die Merkmale des oft außerhalb der direkten Einwirkung des Lieferanten liegenden Code beeinflusst.

Von dieser komponentenbasierten Welt der Softwareentwicklung gibt es heute keine Flucht mehr und auch keinen Grund, danach zu suchen. Die Zerteilung von gesamten Softwaresystemen in kleinere Einheiten, die sich unabhängig voneinander verstehen, verändern und prüfen lassen, bringt schließlich Vortei-

le mit sich, die gleichermaßen von den Vätern der Informatik und den frühen Projektleitern ersehnt wurden: eine verbesserte Qualität und gesunkene Entwicklungskosten.

Dies ist eine Wunschvorstellung – sie wird in dem grauen Programmieralltag nur selten erfüllt. Der typische Programmierer dürfte seine Arbeitszeit eher mit ermüdender Fehlersuche, hastigen Korrekturen und der Befriedigung der Kundenwüsche durch Einbau neuer Features verbringen. Wenn die Software einmal "läuft", heißt die Aufgabe "erledigt". Von der sogenannten "Softwarearchitektur" hat man natürlich vieles gehört und manches gelesen (hoffentlich nicht nur in Werbeprospekten), aber wie man die "gute Architektur" von dem "üblichen Notzustand" unterscheidet, ist weniger offenbar. Dafür braucht man ja wie alle großen Architekten das richtige Gefühl und die langjährige Erfahrung[1].

Mangels Aufmerksamkeit auf den Entwurf werden die natürlichen Zerfallsprozesse [2] in der Software fortgesetzt. Häufig gibt es weder Zeit noch Geld, um die innere Struktur der Software zu verbessern, ohne neue Funktionen einzubauen – ein Versagen der Planung. Allerdings weiß man auch bei ausreichenden Ressourcen nicht recht, wie man diese Aufgabe systematisch bewältigen soll. Sicherlich erkennt jeder Programmierer mit etwas Erfahrung den "unschönen" Code und kann ihn korrigieren, indem offenbare Redundanzen beseitigt oder vereinbarte Richtlinien genauer befolgt werden. Diese Maßnahmen, bekannt unter dem Begriff Refactoring [3], werden meistens auf der untersten Implementierungsebene stellenweise ergriffen. Möglicherweise werden dabei die Zusammenhänge zwischen Klassen mit Hilfe von Entwicklungsmustern umgeordnet – die (subjektiv beurteilte) Lesbarkeit des Programmtextes gilt als Erfolgsindikator. Was fehlt, ist ein Überblick des gesamten entstehenden Systems.

Diese Diplomarbeit versucht, einen Beitrag zur Verbesserung der alltäglichen Softwareentwicklung durch eine werkzeugunterstützte, interaktive Untersuchung der Softwaremerkmale auf einer höheren Ebene als Klassen und Objekte zu leisten. Durch die Bereitstellung eines geeigneten Werkzeugs, genannt

[1] es gibt Bemühungen, dieses Wissen zu kodifizieren; vgl. dazu [1]

Classdep, wird für einen relativ breiten Entwicklerkreis[2] die Möglichkeit geschaffen, sich mit der Gestaltung der Softwareentwürfe – die demnächst auch genauer definiert werden – praktisch zu befassen.

Nennt man die Begriffe "Softwareentwurf" und "Objektorientierung" in einem Zug, so kommt schnell eine dreibuchstabige Abkürzung in den Sinn: UML (*Unified Modeling Language*). Bei Classdep handelt es sich um kein Modellierungswerkzeug aus der Kategorie der UML-Diagrammeditoren. Diese gehen meist davon aus, dass man den Entwurf einer Software ihrer Implementierung voranstellt und die beiden nachher getrennt pflegt. Obwohl UML keine Vorgehensweise bei ihrer Nutzung vorschreibt, wird typischerweise zunächst ein Ausschnitt der für den Anwendungsbereich relevanten Realität in einem abstrakten Modell erfasst. Dieses besteht aus mehreren Diagrammen in verschiedenen Detaillierungsstufen. Anschließend wird das Modell in einer bestimmten Programmiersprache umgesetzt (an dieser Stelle können auch mehrere Sprachen gegeneinander abgewogen werden).

Während diese sorgfältige Vorgehensweise unbestrittene Vorteile hat, darf man nicht vergessen, dass der Entwurf in der Praxis auch von der Implementierung beeinflußt wird (Lernprozess) und beim Eintritt von Anforderungsänderungen zusammen mit der Implementierung angepasst werden muss. Nur wenige und relativ teure Entwicklungsumgebungen erlauben es, die einmal erstellten UML-Diagramme mit Änderungen im Code auf dem Laufenden zu halten (und umgekehrt). Zusätzlich ist die Modellierungssprache sehr umfangreich. Dies verursacht zum einen eine hohe Entscheidungslast bei ihren Anwendern: welche UML-Diagramme mit welchem Inhalt lassen die Qualität eines Entwurfs am besten beurteilen? Zum anderen besteht die Gefahr, dass man bei der Pflege von ausgefeilten Zeichnungen (deren Ausdrücke meistens viel zu viel Papier verbrauchen) vor lauter Bäume den Wald nicht mehr sieht. Dann wird eventuell auch der Zweck dieser Bemühungen verpasst: die Herstellung von guter Software. Während man UML-Diagramme zur Kommunikation und als Denkstütze

[2] Java-Programmierer, vor allem Anwender der offenen Entwicklungsumgebung Eclipse

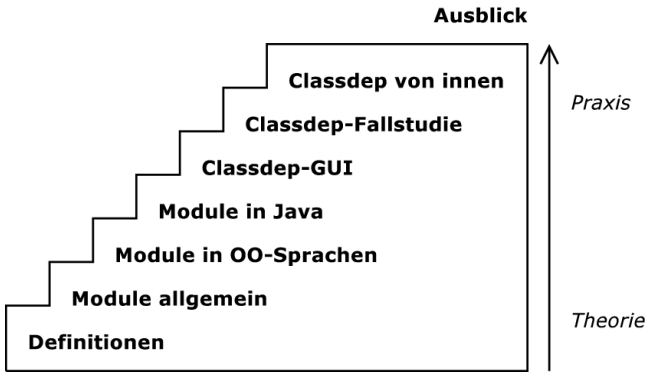

Abbildung 1.1: Aufbau der Diplomarbeit

bei der Ausarbeitung von Konzepten gut gebrauchen kann, soll ein Werkzeug zur Abhängigkeitsanalyse dort helfen, wo jeder Entwurf zur Realität wird – bei seiner implementierungsnahen sauberen Umsetzung im Quellcode.

1.2 Aufbau dieser Arbeit

Im zweiten Kapitel werden die im Umfang dieser Arbeit geltenden Begriffsdefinitionen und der allgemeine Ansatz der statischen Quellcodeanalyse eingeführt.

Das dritte Kapitel begründet den Bedarf nach Modularisierung und stellt eine Sammlung von Empfehlungen und Techniken dar, die eine bessere Strukturierung der Software fördern. Insbesondere werden in diesem Zusammenhang die Prinzipien der objektorientierten Programmierung beleuchtet, die auf der einschlägigen Literatur und Expertenmeinungen beruhen. Anschließend wird die Eignung der statischen Quellcodeanalyse als Mittel zum Überprüfen der Einhaltung beschriebener Empfehlungen diskutiert.

Das vierte Kapitel schildert eine spezifische Art und Weise der Modulbildung für Java-Projekte, die aus Erfahrungen des Autors mit der Verwendung des entwickelten Werkzeugs Classdep resultierte. Hier wird die Idee verfolgt, durch einen systematischen Programmierstil eine verbesserte Grundlage für die durchgehende Auswertung von Modulabhängigkeiten herzustellen. Die beschriebene Technik der Abbildung von Programmmodulen auf Java-Pakete

dürfte übrigens auch ohne Werkzeugeinsatz nützlich sein. Aus diesem Grund wird sie der Beschreibung von Classdep vorangestellt.

Im fünften Kapitel werden die Funktionen von Classdep erörtert. Die Maskenbeschreibungen sollen dem Leser einen ersten Einblick in die Einsatzmöglichkeiten der statischen Quellcodeanalyse geben. Bei dieser Gelegenheit wird auch der Bezug der von Classdep angebotenen Klassen- und Paketdiagramme zu den entsprechenden UML-Diagrammen dargelegt.

Das sechste Kapitel schildert die Verwendung von Classdep zur Analyse eines zufällig gewählten Java-Projekts mittlerer Größe. Die im vorherigen Kapitel beschriebene Funktionalität des Werkzeugs wird nun in einem praktischen Einsatzszenario auf die Probe gestellt. Die detaillierten Ausführungen erfassen die Art und Weise der Werkzeugverwendung zusammen mit einem für die Abhängigkeitsanalyse typischen Gedankenfluß.

Schließlich wird in den beiden letzten Kapiteln auf Implementierungsdetails von Classdep, insbesondere auf seine Integration in die Entwicklungsumgebung Eclipse, und auf die zukünftigen Verbesserungsmöglichkeiten eingegangen.

Kapitel 2

Konzepte und Definitionen

Als Gegenstand dieser Arbeit wurde ein Werkzeug zur **Analyse** der Software entwickelt und beschrieben. Das Wort "Analyse" bedeutet die Auflösung eines Ganzen in seine Bestandteile. Bei einem abstrakten Produkt wie Software kann diese Auflösung auf unterschiedliche Weisen erfolgen und dementsprechend zu unterschiedlichen Erkenntnissen führen. Man kann etwa auf einer niedrigen Abstraktionsebene jedes Programm als eine Menge von Quellcode- und Ressourcendateien betrachten. Das gleiche Programm kann aber auch in für seine Benutzer relevante Funktionen zergliedert werden. In diesem einführenden Kapitel wird der für den Rest der Arbeit geltende Betrachtungsstandpunkt festgelegt, indem die untersuchten Softwarebestandteile und ein Verfahren zur Erfassung ihrer Beziehungen definiert werden.

2.1 Module und Softwareentwurf

Der Begriff **Modul** wird in dieser Arbeit zunächst sehr allgemein, dann aber bei der Werkzeugbeschreibung recht spezifisch verwendet. Ein Modul ist eine logische Einheit innerhalb eines Softwareprodukts, die durch ihren Namen und ihren definierten Funktionsbereich von anderen Bestandteilen der gleichen Software unterschieden werden kann. Der Funktionsbereich eines Moduls wird

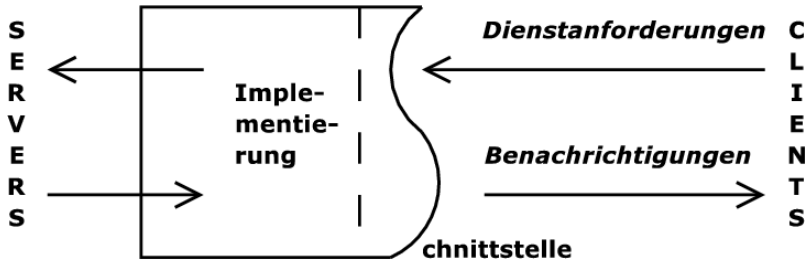

Abbildung 2.1: Das allgemeine Modulkonzept und Modulinteraktionen

durch Dienste festgelegt, die das Modul anderen Modulen anbietet und von ihnen beansprucht. Diese Dienste gehören zu folgenden grundsätzlichen Kategorien:

- ein Modul stellt anderen Modulen Hilfsmittel zur Erledigung ihrer Aufgaben bereit (die Dienstverbraucher werden in diesem Kontext oft als Clients bezeichnet)
- ein Modul benachrichtigt andere Module über das Eintreffen von interessanten Ereignissen

Diese beiden Rollen entsprechen den verschiedenen Kommunikationsflüssen zwischen Modulen. Im ersten Fall übernimmt ein Client-Modul die Initiative bei der Nutzung des Dienstleisters, im letzteren wird die Kommunikation durch das benachrichtigende Modul angestoßen und damit die Steuerung an das benachrichtigte Modul überreicht (*callback*). Es ist durchaus möglich, dass ein einzelnes Modul beide Rollen gleichzeitig erfüllt (Abbildung 2.1).

Eine bestimmte, zielgerichtete Konkretisierung des Modulbegriffs in Bezug auf die Java-Entwicklung mit Classdep wird später bei der Diskussion des Analysewerkzeugs vorgeschlagen. An dieser Stelle werden nur exemplarisch einige Möglichkeiten der Abbildung des Modulkonzepts auf Entitäten aus der Softwarewelt aufgeführt:

- eine Funktion oder Prozedur in der strukturierten Programmierung

- alle Funktionen und Datenstrukturen, die in einer gemeinsamen C-Quellcodedatei definiert sind
- eine Klasse in einer objektorientierten Programmiersprache
- eine COM-Komponente (Component Object Model von Microsoft®) in der Windows™-Umgebung [7]
- eine Enterprise JavaBean in einem Java™Application Server [8]
- ein Web Service
- ein Plugin in der Eclipse-IDE [11]
- ein Paketbündel in der Programmiersprache Java

Um die weiteren Überlegungen und insbesondere den Gegenstand der Abhängigkeitenanalyse im Sinne dieser Diplomarbeit zu konkretisieren, wird die folgende, einschränkende Moduldefinition verwendet:

Ein Modul besteht aus einer Menge von Typen, die in einem objektorientierten Programm definiert sind. Jeder Typ wird zu höchstens einem Modul zugeordnet.

Der in der obigen Definition verwendete Begriff "Typ" kann mathematisch durch eine Aufzählung von zusammengehörigen Funktionen, Axiomen und Vorbedingungen definiert werden, die gemeinsam eine (möglicherweise unendliche) Menge von Typinstanzen spezifizieren.[1] Aus einem anderen Blickpunkt kann man das Wort "Typ" als Oberbegriff für die von einer Programmiersprache zur Spezifizierung und Implementierung eines abstrakten Typs angebotenen syntaktischen Mittel verwenden. So werden die in Java vorkommenden Klassen und Interfaces zusammenfassend als "Typen" bezeichnet.[2] Im Rest dieser Arbeit wird der Typbegriff kontextabhängig in beiden Bedeutungen verwendet: an diesen Stellen, wo die statische Abhängigkeitsanalyse behandelt wird, mit

[1] vgl. dazu [10], S. 130
[2] vgl. *reference types* in der Java-Sprachspezifikation [4]

Bezug auf den Quellcode, ansonsten in seiner abstrakten Fassung. Die syntaxorientierte Begriffsverwendung dürfte einem Anwender von Classdep vertrauter als die formale wissenschaftliche Typdefinition vorkommen.

Es ist erwähnenswert, dass es speziell im Bezug auf objektorientierte Programmiersprachen auch andere Definitionen des Begriffs "Modul" gibt. Nach Anschauung von B. Meyer sollten in objektorientierten Programmen Klassen als einzige Form der Module gelten.[3] Die Aussonderung einer groberen Modulart erscheint dennoch nutzvoll. Sie folgt aus der Beobachtung, dass gewisse Klassengruppen zusammenhängender als andere sind, und dass es in der Regel auch solche Klassen gibt, deren Verwendung auf eine bestimmte Menge anderer Klassen beschränkt werden kann und sollte. [16]

Eine weitere Überlegung ist, ob jeder Typ zwingend zu einem Modul gehört. Ist das der Fall, so besteht jedes Programm aus einer geschlossenen Menge von Modulen, die ihrerseits aus einzelnen Typen bestehen. Lässt man auch "modullose" Typen zu, wie die Definition nahelegt, so kann man nicht jede Software als ein Bündel aus Modulen betrachten. Die Entscheidung, ob eine (oder sogar jede) Klasse im Kontext eines Moduls existiert, hängt eng mit den verwendeten Zuordnungskriterien zusammen. Grundsätzlich gilt, dass die Zuordnung jeder Klasse zu einer benannten Gruppe das Verständnis der groben Softwarestruktur fördert. Diese Gruppe darf im Sinne der obigen Definition als "Modul" bezeichnet werden – auch wenn sie kein "richtiges" Modul gemäß den Empfehlungen und Prinzipien aus den folgenden Kapiteln darstellt. Man kann also eine vollständige Unterteilung eines Softwareprodukts in Module anstreben oder auch manche Typen, wie z. B. solche, die zu externen Klassenbibliotheken gehören, außer Acht lassen. Entwickelt man eine Bibliothek oder ein Framework, so sollte man überlegen, welcher modulare Aufbau für den Benutzer sinnvoll wäre.

Es gibt immer zahlreiche Möglichkeiten, ein Programm in Module zu unterteilen, ohne seine für den Anwender relevante Funktionalität zu beeinflußen.

[3] vgl. dazu [10], S. 24

Jede solche Modularisierung resultiert in anderen Kommunikationsflüssen und Aufgabenzuständigkeiten der einzelnen Module. Die Gesamtheit dieser Aufgaben und Beziehungen wird im Folgenden als **Softwareentwurf** bezeichnet. Auch diese Definition ist einschränkend: man könnte den Begriff ebenfalls auf die Rollen und Beziehungen einzelner Typen erweitern, wie es etwa bei der Dokumentierung von Entwurfsmustern üblich ist. Die Qualität des Entwurfs, in beiden Begriffsfassungen, bestimmt solche Merkmale wie die Entwicklungskosten und Pflegeaufwand mit.

2.2 Modulabhängigkeiten

Bei der Definition des Modulkonzepts wurde bereits auf Kommunikationsarten und die Zusammenarbeit der Module bei der Erfüllung von Benutzeranforderungen hingewiesen. Die Beziehungen zwischen Modulen können in Form von gerichteten Abhängigkeiten genauer erfasst und analysiert werden. Die Abhängigkeiten werden mengentheoretisch als eine Relation auf der Modulmenge definiert. Ein Modulpaar gehört zur Relation, wenn das eine Modul von dem anderen abhängt. Legt man die Bedeutung des Wortes “abhängen” genauer fest, so kann man auch die allgemeinen Eigenschaften der entstandenen Relation untersuchen (transitiv?, reflexiv?, symmetrisch?).

Im Rahmen dieser Diplomarbeit wird, falls nicht anders angedeutet, die folgende Definition einer **Modulabhängigkeit** verwendet:

Das Modul A hängt vom Modul B ab, wenn in der Deklaration mindestens eines zum Modul A gehörenden Typs Referenzen zu Typen aus B vorkommen.

Die in dieser Definition vorkommenden Begriffe “Typdeklaration” und “Typreferenz” sind spezifisch für objektorientierte Programmiersprachen. Damit wird das Arbeitsthema auf eine (umfangreiche) Untermenge aller Software fokussiert. Es wird angenommen, dass jeder Typ aus einer Menge von Methoden und Datenvariablen besteht, die in seiner Deklaration (d.h. im Programmtext)

angegeben werden. Die “Typreferenzen” sind zur Compilierungszeit erkennbare Namen, die zur Festlegung des zulässigen Wertebereichs für Variablen, Parameter und Rückgabewerte dienen. Ihr Vorkommen ist charakteristisch für statisch typisierte Programmiersprachen wie C++ und Java. Ferner werden auch statt “Typ”, wenn es um die Syntax geht, die konkreteren Begriffe “Klasse” und “Interface” verwendet, in der für die Programmiersprache Java üblichen Bedeutung. [4]

Die Modulabhängigkeiten sind im definierten Sinne nicht transitiv: wenn das Modul A vom Modul B abhängt, und ein weiteres Modul C von A abhängt, redet man noch von keiner Abhängigkeit zwischen C und A. Die Relation ist reflexiv (jedes Modul hängt von sich selber ab), weil die im Modul enthaltenen Typen normalerweise Referenzen zu anderen Typen aus dem gleichen Modul (oder auch zu sich selber) beinhalten. Die Abhängigkeiten sollten, wie später noch genauer erklärt wird, niemals symmetrisch sein – wenn das Modul A vom Modul B abhängt, so darf das Modul B nicht gleichzeitig von A abhängen.

Es ist zu unterstreichen, dass Modulabhängigkeiten laut der obigen Definition nicht das gesamte Spektrum der Abhängigkeiten umfassen, die bei der Softwareentwicklung und insbesondere bei Entwurfsentscheidungen eine Rolle spielen. Es wurde vielmehr eine Definition gewählt, die sich mit einem relativ kleinen Programmieraufwand praxisnah in einem Werkzeug umsetzen lässt und dabei einfach verständlich ist. Eine alternative, mächtigere Definition, die eine Relation als Obermenge der vorherigen bildet, sei hier zum Vergleich aufgeführt:

Das Modul A hängt vom Modul B ab, wenn während der Programmlaufzeit vorkommen kann, dass ein Objekt eines zum Modul A gehörenden Typs eine Referenz zum Objekt eines zum Modul B gehörenden Typs enthält.

Auf Sachverhalte, die sich mit dieser erweiterten Definition untersuchen lassen aber unserer einfacheren Betrachtungsweise entweichen, wird im nächsten Kapitel noch hingewiesen.

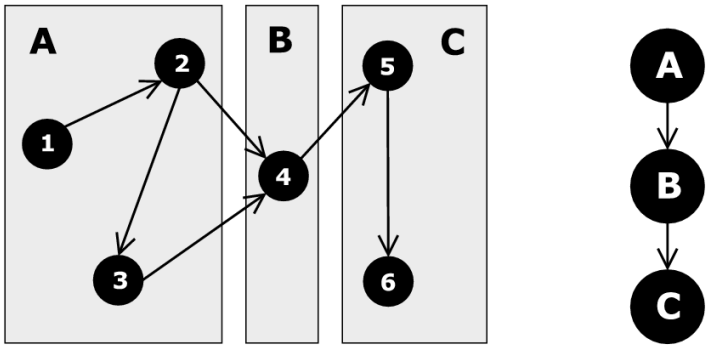

Abbildung 2.2: Ableitung von Modulabhängigkeiten (rechts) aus Typabhängigkeiten (links)

2.3 Analyseverfahren

Als **Analyse** werden in dieser Diplomarbeit Verfahren bezeichnet, die zu der Ermittlung der Menge von Modulabhängigkeiten und ihrer Auswertung führen. Die Arten der Abhängigkeitsanalyse hängen mit verwendeten Definitionen der Modulabhängigkeiten eng zusammen. Man kann die Analyseverfahren mit Hilfe von zwei Dimensionen klassifizieren:

- statische Analyse (Analyse des Quellcode)
- dynamische Analyse (Analyse der Laufzeitstrukturen)

und

- syntaktische Analyse (an Hand einer Grammatik)
- semantische Analyse (Analyse des Zwecks und Bedeutung)

Zur Ermittlung der im vorherigen Kapitel definierten Modulabhängigkeiten ist eine statische, syntaktische Analyse des vorhandenen Programmtextes ausreichend. Hierzu wird für jede im Programm befindliche Typdeklaration der von einem Parser erzeugte abstrakte Syntaxbaum traversiert und diejenigen Fragmente genauer betrachtet, die Typreferenzen als Unterknoten enthalten.

Als Ergebnis bekommt man einen gerichteten Graphen, in dem alle Programmtypen als Knoten vorkommen und die Kanten die Anwesenheit einer oder mehreren Zieltypreferenzen im Quelltyp bedeuten. Durch eine Zuordnungsfunktion werden einzelne Typen zu Modulen gruppiert und anschließend ein Graph mit Modulknoten gebildet. Die ein- und ausgehenden Kanten (also Abhängigkeiten) für ein Modul ergeben sich aus den entsprechenden Kantenmengen für jeden enthaltenen Typ, wie in Abbildung 2.2 veranschaulicht.

Kapitel 3

Richtlinien zur Modulbildung

Im vorherigen Kapitel wurden Module als Mittel zur Gruppierung von Typen in einem objektorientierten Programm eingeleitet und bereits einige Fragen zur Beziehung zwischen Typen und Modulen aufgeworfen. Im Folgenden werden die Vorteile der Modularisierung genauer erläutert und manche Ideen zur Modulbildung vorgestellt. Als Entscheidungshilfe bei der Festlegung der Modulinhalte und Auswertung von Modulabhängigkeiten werden allgemeine und objektorientierte Prinzipien der Softwareentwicklung herangezogen. Die Eignung der statischen Quellcodeanalyse als Methode zur Erfassung und Beurteilung der Softwareentwürfe wird nach der Darstellung dieser Prinzipien untersucht und zusammenfassend bewertet.

3.1 Motivation

Modular aufgebautes System, elegantes Programm, flexible Architektur – diese und ähnliche Begriffe werden von Entwicklern und Marketing-Abteilungen gerne verwendet, ohne mehr als ihr intuitives Verständnis anzusprechen. Jeder Softwarehersteller möchte, dass seine Produkte diese positiv klingenden Attribute auch tatsächlich besitzen. Allerdings spielen auch Termine und kurzfristige Entwicklungskosten eine Rolle. Es ist leicht einzusehen, dass die Ent-

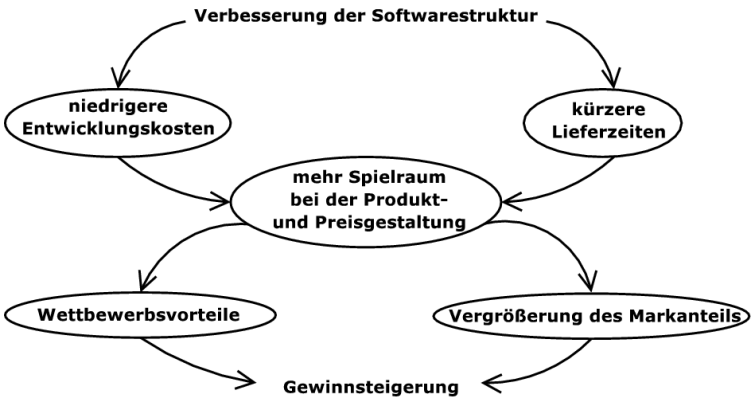

Abbildung 3.1: Wirtschaftliche Argumente für die Auseinandersetzung mit der Softwarestruktur

wicklung einer "sehr guten" Software einen komplexeren Prozess, mehr Programmierzeit und eine höhere Qualifizierung der Beteiligten erfordert, als es zur schnellstmöglichen Erfüllung von gestellten Anforderungen notwendig wäre. Setzt man ein sorgfältiges automatisiertes Testen und ein paralleles Refactoring des geschriebenen Code ein, so verbringt man damit erfahrungsgemäß die gleiche Zeitmenge wie mit dem Einbau neuer Funktionen. Dieser Aufwand kann nicht durch rein ästhetische Beweggründe berechtigt werden. Die wirklichen Argumente für die Verbesserung der internen Softwarestruktur sind:

- Reduzierung der **langfristigen** Entwicklungskosten
- Verkürzung der Lieferzeiten von nächsten Versionen [5]

Die beiden Ziele führen zusammen zu Wettbewerbsvorteilen und können auf lange Sicht bei der Vergrößerung des Marktanteils eines Softwareherstellers helfen (Abbildung 3.1). In nächsten Kapiteln wird erläutert, warum der ansonsten unsichtbare Softwareentwurf bei der Erfüllung dieser Ziele eine Rolle spielt. Es wird auf Prinzipien hingewiesen, deren Verwendung die Entstehung von schlechten Entwürfen vermeiden kann. Schlechter Entwurf bringt nach R. Martin [14] folgende Nachteile mit sich:

- Starrheit – jede Änderung erfordert viele Folgeänderungen in unterschiedlichen Modulen

- Zerbrechlichkeit – mit jeder Änderung ist die Gefahr verbunden, dass unerwartete Systemteile zu funktionieren aufhören
- Unbeweglichkeit – Module sind kaum wiederverwendbar, da ihre Absonderung von anderen Modulen sehr schwer fällt

3.1.1 Erhöhung der Testbarkeit

Unter den Gründen für erhöhte Entwicklungskosten und für verlängerte Entwicklungszeiten sticht einer besonders hervor: Fehler. [12] Die Fehleranzahl alleine ist daher ein hinreichend aussagekräftiger Maßstab für die Erreichung der Geschäftsziele eines Softwareherstellers. Im Unterschied zu Defekten in materiellen Produkten entstehen Softwarefehler niemals durch einen "natürlichen Verschleiß beweglicher Teile". Sie werden von Entwicklern beim Hinzufügen zusätzlicher Funktionen und bei Anpassungen miteingebaut.

Man kann allgemein zwei Entstehungsszenarien für Fehler unterscheiden:

- die Spezifikation für eine bestimmte Funktion fehlt zum Zeitpunkt der Entwicklung; die (intuitiv) implementierte Verhaltensweise stellt sich später als unerwünscht oder inkomplett heraus
- nach einer Code-Änderung wird die bereits erfüllte Spezifikation einer Funktion nicht mehr eingehalten (Regression)

Ein brauchbares und in den letzten Jahren popularisiertes Mittel zur Verhütung von Fehlern beider Arten sind automatisierte Modultests. Ein Modultest überprüft die Verhaltensweise eines bestimmten Code-Fragments durch Vergleich mit seiner Spezifikation.[1] Wie der Name sagt, konzentriert man sich üblicherweise auf ein einzelnes "Modul" – das Wort wird dabei aber oft im Sinne von "Klasse" verwendet. Das in Java mit JUnit [19] eingebürgerte Verfahren bei der Entwicklung von Modultests besteht darin, für jede Klasse eine

[1] in manchen Fällen **ersetzt** der Test die Spezifikation

entsprechende Modultestklasse und je einen Testfall für jede nichttriviale Methode zu erstellen. Auch wenn man ein umfassenderes Modulkonzept verwendet (wie etwa ein Paket mit mehreren Klassen), kann dieses Vorgehen ohne Änderungen übernommen werden. Die Klassen und ihre Methoden stellen nach wie vor die kleinsten testbaren Bausteine dar. Die Tests sollten auf einer möglichst niedrigen Ebene ablaufen, um die Testüberdeckung zu maximieren und den Suchbereich für entdeckte Fehler einzuschränken.

Für den Erfolg und Aufwand bei der Modultesterstellung sind Modulabhängigkeiten und explizit festgelegte Modulschnitstellen von entscheidender Bedeutung – eine Einsicht, die in den einführenden Publikationen zum Thema Unit-Testing durch eine bequeme Wahl einfacher Beispiele oft verschwiegen wird. Eine wesentliche Anforderung an ein Testsystem ist die unbeaufsichtigte Wiederholbarkeit aller Tests, d.h. ihre Automatisierung. Diese Anforderung ist nicht erfüllt, wenn vor der Ausführung einzelner Tests eine aufwändige Vorbereitung ihrer Umgebung oder nachher eine manuelle Auswertung der Testergebnisse erforderlich ist. Die Schwierigkeit bei der Erstellung eines Modultests besteht darin, die Vorbedingungen für den Testablauf außerhalb der eigentlichen Softwareanwendung zu schaffen, ohne den normalen Benutzer zu beteiligen oder seine Bedienungsschritte zu simulieren.[2] Jede getestete Klasse referenziert in der Regel andere Klassen. Durch diese Tatsache entstehen rekursiv Ketten von Abhängigkeiten, die das Erstellen des getesteten Objekts und den Aufruf seiner Methoden ohne eine vorherige Initialisierung vieler anderer Objekte unmöglich machen. Diese anderen Objekte werden möglicherweise während des Testlaufs gar nicht oder nur indirekt vom getesteten Objekt verwendet. Die resultierende Erhöhung des Aufwands für die Testvorbereitung führt praktisch zu folgenden Konsequenzen:

- nur die leicht testbaren, alleinstehenden Module werden überhaupt geprüft

[2] Tests, die das Benutzerverhalten abbilden (Akzeptanz- bzw. Funktionstests), sind auch sinnvoll, aber hier nicht gemeint.

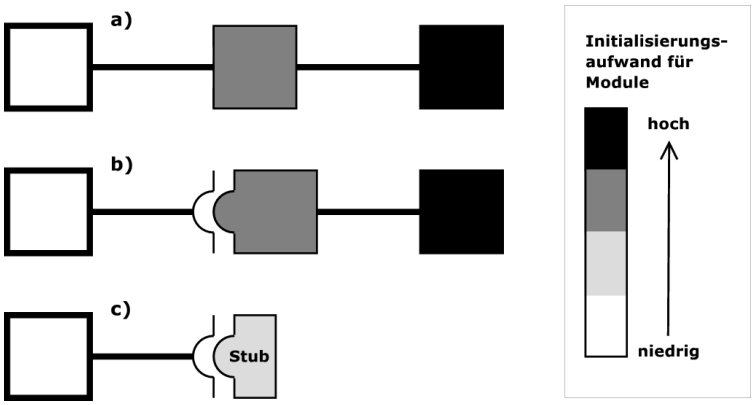

Abbildung 3.2: Erleichterung von Tests durch die Aussonderung einer Modulschnittstelle

- Module, die sich mitten in einer Abhängigkeitskette befinden, werden nur indirekt geprüft

Die Schwierigkeiten bei der Herstellung einer Testumgebung können auch mit der Laufzeitperformanz der Tests verbunden sein: eine Sammlung von Modultests, die zur Ausführung zu lange braucht, wird von Entwicklern seltener gestartet als eine ausreichend schnelle. Die Zeitspanne zwischen der Entstehung eines Defekts und seiner Entdeckung wird entsprechend breiter.

Die Entkopplung von Modulen mit Hilfe von abstrakten Schnittstellen ist ein Schlüssel zur Lösung der beschriebenen Probleme mit Testbarkeit. Die syntaktische Analyse bietet direkte Hinweise auf die Stellen im Code, wo die Einführung einer abstrakten Schnittstelle zwischen Modulen erforderlich ist. Kommunizieren Module miteinander über solche Schnittstellen, so können bei einem Modultest alle unmittelbar referenzierten Module durch vereinfachte Repräsentationen (auch *stubs* genannt) ersetzt und die Abhängigkeitsketten durchbrochen werden.

Abbildung 3.2 veranschaulicht die allgemeine Vorgehensweise: der oberste Teil (a) zeigt die Module vor der Aussonderung der Schnittstelle. Die dicken Verbindungslinien stehen für Abhängigkeiten. Das zu testende Modul ist als Rechteck auf der linken Seite abgebildet. Durch die Schnittstellendefinition (b) kann man dieses Modul von den beiden anderen entkoppeln. Die Nutzung eines

einfachen Stubs mit der passenden Schnittstelle beim Test (c) vermindert die insgesamt für die Testdurchführung erforderliche Vorbereitungsarbeit.

Kann man Module voneinander isolieren und leicht getrennt testen, so ist zwar noch nicht garantiert, dass die Tests alle oder sogar die meisten Fehler entdecken. Der Erfolg hängt zusätzlich vom Verhältnis der möglichen und geprüften Kontrollflüsse ab. Selbst ihre Ermittlung kann zum hartnäckigen Problem werden, besonders in nebenläufigen Programmen. Allerdings liegt es nahe, dass durch die Aufteilung des getesteten Programms auch die Genauigkeit der Tests erhöht werden kann.

Zur Veranschaulichung dieser Aussagen betrachten wir ein Szenario, in dem die Testbarkeit durch die Verbesserung der Modulabhängigkeiten erhöht werden kann. Gegeben sei eine Klasse `DataWindow`, die ein Fenster mit zwei Schaltflächen und einer Liste implementiert. Beim Anklicken der ersten Schaltfläche sollen Datensätze aus einer Datenbank geholt und in der Liste angezeigt werden. Beim Anklicken der anderen Schaltfläche werden die angezeigten Datensätze gelöscht und das Programm beendet. Wir wollen nun einen automatischen Modultest für die Klasse `DataWindow` schreiben.

Die folgenden Schwierigkeiten treten auf:

- man muß das Datenbankmodul vor dem Testlauf korrekt initialisieren
- man benötigt eine Datenbank mit dem korrekten Inhalt
- man braucht einen anzeigefähigen Rechner (man kann also z. B. nicht direkt auf dem CVS-Server im Rechenzentrum testen)
- nach dem Aufblenden des Fensters müssen Benutzeraktionen wie Mausbewegungen und -klicks simuliert werden
- die Inhalte der einzelnen grafischen Komponenten müssen abgefragt werden (wurden die Datensätze überhaupt angezeigt, sind es wirklich alle Datensätze, die in der Datenbank vorliegen?)

- durch das Anklicken der Schaltfläche “Ende” wird die virtuelle Maschine gemäß der Anforderungsbeschreibung gestoppt, was die Verwendung dieses Tests als Teil einer Testsammlung verhindert

- nach dem erfolgreichen Test verschwinden Datensätze aus der Datenbank; für den nächsten Testlauf müssen sie also neu eingefügt werden

Um der Aufgabe gewachsen zu sein, müsste der für die Implementierung des vorgestellten Tests zuständige Programmierer über tiefere Kenntnisse der Datenbankkonfiguration, der GUI-Klassenbibliothek und der Testumgebung verfügen. Es ist viel wahrscheinlicher, dass der Test gar nicht automatisiert wird, sondern erst vom Benutzer in der Zielumgebung des Programms durchgeführt wird. Als Preis wird die Absicherung gegen zukünftige Regressionsfehler bezahlt.

Der erhebliche Testaufwand kann vermindert werden, indem das Modul `DataWindow` in geeignete Untermodule zerteilt wird, und zwischen ihnen abstrakte Schnittstellen eingeführt werden. Zum Beispiel könnte man für den Datenbankzugriff eine Schnittstelle mit der Methode `delete` verwenden. Bei der Durchführung des Tests wird das eigentliche Datenbankmodul mit einem speziellen Testmodul ersetzt, dessen Rolle darin besteht, die von `DataWindow` an die Datenbank verschickten Nachrichten zu verifizieren. Das simulierte Ereignis “Beenden des Programms” muss etwa einen Aufruf von `delete` bewirken. Ob diese Methode dann tatsächlich Datensätze löscht oder nicht ist beim Testen von `DataWindow` irrelevant.

Nach der Modulzerlegung wird zugleich weniger und tiefer getestet: die Richtigkeit der Hilfsmodule wird nicht überprüft, andererseits können die internen Abläufe im getesteten Modul einzeln unter die Lupe genommen werden.[3] Grundsätzlich gilt, dass für die Erleichterung der automatischen Tests solche Modulabhängigkeiten, in denen nur konkrete Klassen vorkommen, beseitigt werden sollten.

[3] *Endo-Testing*, vgl. dazu [20]

Es ist nicht immer praktisch, die Modulverwendung mit Hilfe einer Schnittstelle wegzuabstrahieren. Ein typisches Gegenbeispiel sind die grafischen Oberflächen. Die meisten Implementierungen von Bibliotheksklassen wie `Button` oder `List` sind direkt an die Gegebenheiten des Betriebs- oder Fenstersystems gekoppelt und lassen sich nicht einfach durch schnittstellenkonforme Stubs ersetzen. Da es sich um fertig gelieferte, geschlossene Komponenten handelt, lässt sich dieser Entwurfsfehler kaum vom Bibliotheksanwender korrigieren. Die Einführung einer zusätzlichen Abstraktionsschicht wie im vorherigen Beispiel ist zwar möglich, aber meist mit einem größeren Aufwand verbunden. In dieser Situation ist eine strikte Trennung des GUI-Moduls ("Darstellung") vom Datenverarbeitungsmodul ("Programmlogik") empfehlenswert. [21] Dieses Beispiel macht deutlich, dass man beim Modulentwurf möglicherweise durch Randbedingungen eingeschränkt ist, die sich aus dem Design der Drittherstellermodule ergeben, und zeigt zugleich, dass durch eine geschickte Modularisierung die Unzulänglichkeiten solcher externen Komponenten umgangen werden können.

3.1.2 Organisation der Teamarbeit

Bei größeren Projekten mit mehreren beteiligten Entwicklern führt der bei Code-Veränderungen entstehende Kommunikationsaufwand zu einer wesentlichen Verlangsamung der Arbeit. Seine Rolle wurde von F. Brooks mit der bekannten Aussage erklärt, dass das Hinzufügen von Entwicklern zu einem verspäteten Softwareprojekt seinen Abschluß noch verzögere. [6] Der Grund dafür ist, dass die neuen Teilnehmer vom bestehenden Team zunächst in das Projekt eingeführt und später auf dem Laufenden gehalten werden müssen.

Es erscheint also naheliegend, nach Verfahren zu suchen, die den Kommunikationsaufwand reduzieren. Eine reibungslose Verteilung der Arbeit an unabhängige Projektmitglieder ist nur dann möglich, wenn die auszuführenden Aufgaben miteinander lose gekoppelt sind. Insbesondere sollten die Aktivitäten der Mitarbeiter weder Störungen (Unterbrechungen) noch Wartezeiten bei ih-

ren Kollegen verursachen. Da sich dieses Ziel im Allgemeinen nicht realisieren lässt – sonst wäre hier von keinem gemeinsamen "Softwareprojekt" die Rede – bemüht man sich, die notwendige Kommunikation zu verwalten, indem man sie auf bestimmte Zeiträume beschränkt und die Anzahl der beteiligten Personen vermindert. Durch eine Veränderung in einem gut definierten Softwaremodul sind potentiell (nur) seine Clients betroffen. Werden Module getrennt voneinander versioniert und veröffentlicht, so können die Entwickler der Client-Module die Integration auf einen passenden Zeitpunkt verschieben, ohne dass ihre Arbeit direkt beim Eintritt der Veränderung unterbrochen werden muss. Betrifft die Veränderung nur die Implementierung eines verwendeten Moduls, so läuft die Integration glatt ab. Die Entwickler verschiedener Module müssen erst dann Absprache halten, wenn Modulschnittstellen verändert werden.

Während die Aussonderung der Module ein "bewährtes" Mittel zur Arbeitsorganisation in Softwareprojekten ist, gibt es auch andere, neuere Ansätze. Die Befürworter von *Extreme Programming* [22] und manchen anderen "agilen Softwareentwicklungsmethoden" [23] empfehlen, dass der gesamte Quellcode von allen Teammitgliedern veränderbar sein sollte und unterstreichen die Bedeutung der "kontinuierlichen Integration". Es wird darauf hingewiesen, dass eine stark isolierte Arbeit und verzögertes Zusammenfügen der Softwaremodule zu größeren Schwierigkeiten in späteren Projektphasen (wegen Missverständnissen) und zur eventuellen Personalknappheit (wegen zu engen Zuständigkeiten) führen kann. Statt Entwickler voneinander mit Modulschnittstellen abzugrenzen und ihre Absprachen auf Release-Sitzungen zu verlegen, greifen deswegen XP'ler zu Mitteln, die die regelmäßige Kommunikation beschleunigen. Als Beispiele seien das flächendeckende, automatisierte Testen des Quellcode und die paarweise Programmierung genannt.

Allerdings ergeben sich aus einer engen Zusammenarbeit von allen Teammitgliedern am gesamten Quellcode auch Gefahren: John Guttag bemerkt in seinem Beitrag zur sd&m-Konferenz von 2001, dass eine zu breite Informations-

verteilung in Softwareprojekten schlecht sei, da es einen Weg zur unerwünschten Kopplung sonst unabhängiger Aspekte öffnet. [5] Es kann dann leicht passieren, dass sich Entwickler auf solche Merkmale stark verlassen, die ursprünglich nur als vorübergehende Lösungen oder Implementierungsdetails gemeint waren. Es geht hier um Verstöße gegen *information hiding*, die im folgenden Kapitel noch genauer untersucht werden. Die werkzeugunterstützte Quellcodeanalyse kann bei der Vermeidung dieser Verstöße zum Teil helfen – vielleicht lässt sich so die Freizügigkeit der XP-Entwickler mit der klassischen Weisheit von *divide et impera* vereinbaren. Wenn man die Schnittstellen nicht vorab entwerfen und einfrieren kann (oder möchte), dann sollte man im Laufe der Entwicklung regelmäßig sicherstellen, dass sie "korrekt gewachsen" sind.

3.2 Das *information hiding*-Prinzip von Parnas

In seinem in 1972 erschienenen Artikel [9] macht D. Parnas die Beobachtung, dass die Art der Modularisierung eines Systems gravierende Konsequenzen für seine zukünftige Anpassungsfähigkeit haben kann. Es werden zwei Versionen des gleichen Programms kurz dargestellt und der erforderliche Anpassungsaufwand diskutiert, der aus Veränderungen von ausgewählten Entwurfsentscheidungen resultiert.

Die Module in der ersten Version des von Parnas behandelten Beispiels werden auf der Grundlage des Verarbeitungsablaufs gebildet – jedes Modul entspricht damit einer einzelnen Laufphase des Programms und verwandelt die von seinem Vorgänger (oder, im Fall des Input-Moduls, vom Anwender) gelieferte Eingabe in eine fest definierte Ausgabe. Das letzte Modul in der Kette produziert die erwünschten Gesamtergebnisse.

Die Module in der zweiten (verbesserten) Version des gleichen Programms werden mit der Absicht gebildet, bestimmte Entscheidungen vor anderen Modulen hinter ihren öffentlichen Schnittstellen zu verstecken. Auf die Reihenfolge der Modulaufrufe wird weniger Rücksicht genommen; sie ist nach wie vor im

Code des Hauptmoduls verkörpert.

Es dürfte nicht allzu überraschend vorkommen, dass sich im letzteren Entwurf die beabsichtigten Änderungen leichter realisieren lassen als in dem ursprünglichen, in der die Module über gemeinsame, exakt spezifizierte Datenstrukturen eng miteinander gekoppelt wurden.

Die empfehlenswerte Vorgehensweise besteht also nach Parnas darin, diejenigen Implementierungsentscheidungen, die entweder voraussichtlich verändert werden, oder zumindest leicht veränderbar bleiben sollten, in die einzelnen Module so einzubinden, dass bei ihrer Veränderung die vorab definierten Modulschnittstellen unberührt bleiben.

Diese Philosophie hat seit der Veröffentlichung des Artikels eine breite Zustimmung gefunden. Mit der Entwicklung von objektorientierten Sprachen wurden an Programmierer bequeme Mittel überreicht, mit denen die Konzepte des *information hiding* sich praktisch umsetzen lassen.

3.3 Abhängigkeitsanalyse und *information hiding*

Im Zusammenhang mit dem *information hiding*-Prinzip stellt sich die Frage, inwiefern die Abhängigkeitsanalyse im Sinne der vorliegenden Diplomarbeit bei der Verbesserung der Modularisierung hilfreich sein kann. Insbesondere ist überlegenswert, ob durch eine statische Quellcodeanalyse diejenigen Schwachstellen, die durch die Verletzung des *information hiding* entstehen, automatisch erkannt werden können. Für die weitere Diskussion werden einige einfache Fallbeispiele herangezogen.

3.3.1 Fallbeispiel 1

Als erstes betrachten wir zwei mögliche Schnittstellen eines Moduls, dessen Rolle ähnlich wie im Artikel von Parnas als Zeilenspeicher bei der Indexerstellung festgelegt wird. Die Schnittstelle in Abbildung 3.3 verletzt das *information*

```
interface LineStorage
{
    public byte[] getLineBytes();
}
```

Abbildung 3.3: Verletzung des *information hiding* im Zeilenspeichermodul

```
interface LineStorage
{
    public void addLine(String line);

    public String getLine(int index);

    public int getLineCount();
}
```

Abbildung 3.4: Gewährleistung des *information hiding* im Zeilenspeichermodul

hiding-Prinzip, indem die anderen Module Zugriff auf die einzelnen Bytes des Zeilenspeichers bekommen. Eine abstraktere Schnittstelle in Abbildung 3.4 verheimlicht die interne Speicherverwaltung hinter geeigneten Zugriffsoperationen und entlastet damit andere Module von der Notwendigkeit, über Implementierungsdetails zu wissen.

Die automatische Analyse der Modulabhängigkeiten liefert in beiden Fällen gleiche Ergebnisse (die Client-Module werden als abhängig von der Schnittstelle des Zeilenspeichermoduls gekennzeichnet). Dieses triviale Beispiel verdeutlicht, dass das *information hiding*-Prinzip über die syntaktische Ebene des Quellcode hinausgeht, und dass zur Beurteilung von automatisch signalisierten Modulabhängigkeiten ein tieferes Verständnis der Programmstruktur notwendig ist. Man muss sich stets die Frage stellen, welche Informationsarten und -mengen über die Modulschnittstellen hinaus an ihre Nutzer (d.h. an andere Module) übermittelt werden.

Im ersten Fallbeispiel veröffentlicht das Zeilenspeichermodul die Information, dass die Zeilen in einem Byte-Array gespeichert werden. Damit die Module den Inhalt dieses Byte-Arrays interpretieren können, muss ebenfalls Informati-

on über die darin verwendeten Datenformate (Bedeutung der einzelnen Array-Elemente) verbreitet werden. Diese Informationsart kann übrigens nur außerhalb des Programmcode geliefert werden - etwa in Form einer schriftlichen Spezifikation oder Kommentare. Eine solche in natürlicher Sprache verfasste Beschreibung muss dann in allen Modulen in ausführbaren Code umgesetzt werden, wodurch ggf. offenbare Redundanzen und mit großer Wahrscheinlichkeit Implementierungsfehler verursacht werden.

Im zweiten Fallbeispiel übermittelt die Schnittstelle trotz der erhöhten Methodenanzahl eine geringere Informationsmenge. Es wird lediglich veröffentlicht, dass die Zeilen eine bestimmte Reihenfolge haben, durch die Angabe der Zeilennummer erhältlich sind, und dass man auch neue Zeilen hinzufügen kann. Genauer genommen wird durch diese Schnittstelle auch das Zeilenformat verbreitet, das zum Austausch von Zeilen mit anderen Modulen dient (die Klasse String repräsentiert eine geordnete Folge von Unicode-Zeichen). Eine ausführliche Spezifikation der internen Speicherung entfällt bzw. sie ist nur bei der Implementierung des Zeilenspeichermoduls wichtig.

Man kann den gleichen Sachverhalt auch aus einer etwas anderen Perspektive betrachten. Interpretiert man jede Schnittstelle als eine Menge an Zusicherungen, die von einem implementierenden Modul gegenüber Client-Modulen gemacht werden, so wird ebenfalls sichtbar, dass das erste Fallbeispiel den anderen Modulen mehr verspricht als zur Erledigung ihrer Aufgaben notwendig wäre. Es wird eine Zusicherung gemacht, dass die Zeilen in einem Byte-Array vorliegen und ein bestimmtes Format auf Byte-Ebene aufweisen. Im Gegensatz dazu werden im zweiten Fallbeispiel nur allgemeine Versprechungen bezüglich der festen Zeilenreihenfolge und der Möglichkeit des beschränkten Lese- und Schreibzugriffs abgegeben.

Eine Analogie zu Geschäftsverträgen dürfte die Unterschiede noch besser veranschaulichen. Im ersten Fall schildert der Lieferant seine Leistungen gegenüber dem Besteller auf genaueste Weise und bietet gleichzeitig keine Er-

folgsgarantie ohne enge Kooperation und Aufwand seitens des Kunden. Im zweiten Fall übernimmt der Lieferant die Haftung für einen bestimmten Erfolg und fordert vom Besteller nur die minimale erforderliche Zusammenarbeit. In der realen Welt würde das zweite Szenario den Lieferanten stark benachteiligen und alleine zum Vorteil des Kunden kommen – es ist kaum vorstellbar, dass ein derartiger kompromissloser Vertrag geschlossen würde. Im Gegensatz dazu kann man in der Welt der Softwaremodule die Schnittstellen meist zum gegenseitigen Vorteil verallgemeinern. Das gilt vor allem dann, wenn die betroffenen Module zum gleichen Programm gehören und von der gleichen Person entwickelt werden.

3.3.2 Fallbeispiel 2

Im vorherigen Abschnitt wurde ein Beweis dafür geliefert, dass die syntaktische Analyse der Modulabhängigkeiten nicht jede Verletzung des *information hiding*-Prinzips aufdeckt. Im folgenden wird ein Gegenbeispiel präsentiert, in dem man aus Ergebnissen der Analyse Rückschlüsse auf die fehlende Informationsverheimlichung ziehen kann. Es handelt sich um ein hypothetisches Datenbankmodul, welches die Aufgabe der dauerhaften Objektspeicherung unter eindeutig zugewiesenen Identifikatoren ("Primärschlüsseln") übernimmt. Die Abbildungen 3.5 und 3.6 zeigen wie im vorherigen Fall zwei Varianten der gleichen Modulschnittstelle.

In der ersten Version der Schnittstelle wird das *information hiding*-Prinzip durch die öffentliche Verwendung der Klasse `java.sql.SQLException` verletzt. Die Tatsache, dass die Implementierung des Moduls eine SQL-Datenbank verwendet, wird hinsichtlich der gelieferten Operationen unnötigerweise den Clients bekannt gemacht. Bei einer Umstellung auf einen anderen Speicherungsmechanismus müssen alle Client-Module angepasst werden, auch wenn die Anpassung geringfügig sein dürfte (Umbenennen des Ausnahmetyps in `try..catch`-Blöcken).

```
import java.sql.SQLException;

interface Database
{
    public Object load(String id)
        throws SQLException;

    public void store(String id, Object obj)
        throws SQLException;
}
```

Abbildung 3.5: Verletzung des *information hiding* im Datenbankmodul

```
class DatabaseException extends Exception
{
    // ...
}

interface Database
{
    public Object load(String id)
        throws DatabaseException;

    public void store(String id, Object obj)
        throws DatabaseException;
}
```

Abbildung 3.6: Gewährleistung des *information hiding* im Datenbankmodul

Die zweite Version vermeidet das Problem durch die Einführung eines moduleigenen Ausnahmetyps (`DatabaseException`).

Im Gegensatz zum vorherigen Fallbeispiel liefert nun die syntaktische Abhängigkeitsanalyse bei der Untersuchung beider Versionen verschiedene Ergebnisse. In der schlecht entworfenen Version wird eine Abhängigkeit der Client-Module vom Paket `java.sql` aufgedeckt, die in der verbesserten Version nicht mehr existiert. Wir sehen also, dass die Analyse unter Umständen die Problemstellen automatisch findet, auch wenn sie nicht als einzige Maßnahme zur Einhaltung des *information hiding* gebraucht werden kann.

Allgemein kann man alle Verstöße gegen das *information hiding* in zwei Gruppen einordnen: solche, die zur Verwendung sonst geheimer Module führen (erkennbar durch die syntaktische Code-Analyse) und solche, die nur den Charakter der Modulabhängigkeiten, nicht aber ihre Anwesenheit, Anzahl oder Richtung beeinflussen (i. A. unerkennbar ohne semantische Analyse).

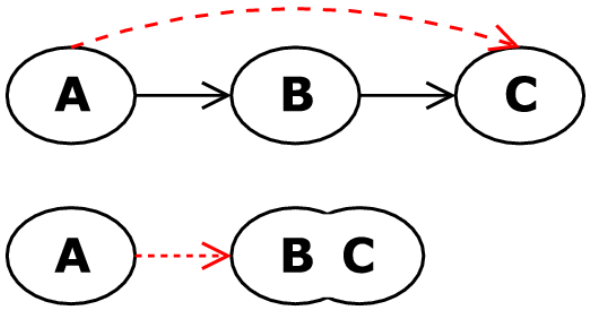

Abbildung 3.7: Zwei Arten der Verstöße gegen *information hiding*

Die Modulabhängigkeiten aus den zwei betrachteten Fallbeispielen sind in Abbildung 3.5 verallgemeinert dargestellt. Der rote Pfeil entspricht jeweils einer unerwünschten Abhängigkeit, die Ovale stehen für Module. Die untere Darstellung beinhaltet einen Hinweis auf die Ursache, warum die unerwünschte Abhängigkeit durch die syntaktische Analyse nicht aufgedeckt wird. Verschmilzt man zwei Module in eines, so entsteht leicht die Gefahr, dass die unerwünschte Abhängigkeit sich auf der syntaktischen Ebene nicht von ande-

ren, erwarteten Abhängigkeiten unterscheidet. Das umgekehrte Verfahren, die Zerteilung eines Moduls in seine öffentliche Schnittstelle und ihre Implementierung, ist eine empfehlenswerte Technik zur Steigerung der Aussagekraft der syntaktischen Analyse.

Zum Abschluß der Fallbeispieldiskussion betrachten wir die Realisierung der Modulzerteilungsidee im Hinblick auf das Zeilenspeichermodul. Die in der schlechten Schnittstelle aufgeführte Methode `getLineBytes()` liefert als Rückgabewert einen `byte`-Array. Im ersten Schritt ersetzen wir diesen Array durch eine Behälterklasse mit der gleichen Funktionalität (etwa `ByteArray` mit Operationen `get`, `set` und `length`). Darauf folgt die Zuordnung der Klasse zu einem der Untermodule – zum Schnittstellenmodul oder zum Implementierungsmodul.[4] Da es sich um eine konkrete Klasse handelt, die eventuell durch eine andere ersetzt werden könnte, ist ihre Platzierung in das Implementierungsmodul naheliegend. Führt man anschließend die syntaktische Analyse durch, so werden Abhängigkeiten der Client-Module von dem Implementierungsmodul über die Schnittstelle hinweg offensichtlich. Eine noch einfachere Heuristik, die zum gleichen Ergebnis führt, besteht darin, alle Methodensignaturen im Schnittstellenmodul auf die Anwesenheit der Typreferenzen zum Implementierungsmodul hin zu untersuchen. Werden solche Referenzen gefunden, so gibt es eine syntaktische Abhängigkeit des Schnittstellenmoduls vom Implementierungsmodul, was bereits einen Verstoß gegen das *information hiding*-Prinzip vermuten lässt.

[4] Das Zeilenspeichermodul wird in zwei Module gemäß der spezifischen Begriffsdefinition aus dem zweiten Kapitel zerlegt.

3.4 Objektorientierte Entwurfsprinzipien

In folgenden Abschnitten werden ausgewählte Prinzipien erläutert, die sich bei der objektorientierten Programmierung etabliert haben, und ihre Verwendungsmöglichkeiten bei der Festlegung von Modulinhalten und Modulgrenzen dargelegt.

3.4.1 Das *Open-Closed*-Prinzip

In seinem Artikel von 1996 [13] beschreibt R. Martin ein Prinzip, dessen Verwendung die Vorteile der objektorientierten Programmiersprachen, speziell die erleichterte Wiederverwendbarkeit und Pflege von Programmen, erreichen lässt. Das *Open-Closed*-Prinzip wurde erstmalig von B. Meyer [10] genannt und besagt:

Module sollten offen für Erweiterungen, aber geschlossen für Veränderungen sein.

Damit ist gemeint, dass eine Erweiterung der Benutzeranforderungen an die Software möglichst keine Modifikationen im Quellcode eines vorhandenen Moduls (oder sogar mehrerer Module) verursachen sollte. Stattdessen sollte der Entwickler in der Lage sein, das Programm zu erweitern, indem er ein neues Client-Modul zum System hinzufügt, das die beabsichtigte neue Verhaltensweise den bestehenden Modulen verleiht. Im Kontext von objektorientierten Programmiersprachen kann dies durch eine geschickte Vererbung und durch die Ausnutzung der Polymorphie erreicht werden.

Konkrete Beispiele findet man in der Beschreibung von Entwurfsmustern wie *Strategy* und *Abstract Factory*. [1] Akzeptiert ein Modul ein *Strategy*-Objekt von seinem Client-Modul, so kann der Nutzer selber entscheiden, welcher konkrete Algorithmus bei der Ausführung der gewünschten Operation verwendet werden soll. Das aufgerufene Modul ist damit für eine externe Erweiterung offen – die Notwendigkeit, seinen Quellcode anzupassen, um den neuen Algorithmus

zu implementieren, entfällt. Ähnlich könnte durch die Verwendung einer *Abstract Factory* die Angabe der innerhalb eines Moduls erzeugten Objekttypen seinen Client-Modulen überlassen werden.

Der grundsätzliche Vorteil aus der Einhaltung des *Open-Closed*-Prinzips ist, dass die Gefahr von Regressionsfehlern reduziert oder sogar vollkommen eliminiert wird. Idealerweise erfordert der Einbau einer zusätzlichen Softwarefunktion keinerlei Anpassungen im bereits existierenden Code, wodurch seine Richtigkeit erhalten bleibt. Dies bedeutet aber noch nicht unbedingt, dass sich nach der Erweiterung alle für den Benutzer interessanten bisherigen Funktionen weiterhin korrekt verhalten werden. Es ist vorstellbar, dass der "externe Beitrag" zu einem Modul die internen Abläufe vollkommen destabilisiert, z. B. wenn von allen Clients gemeinsam genutzte Zustandsdaten gefälscht werden. Allerdings weiß man dann, dass die Fehlerquelle im Erweiterungscode liegt, was die Fehlersuche erleichtert.

Die Verwendung des *Open-Closed*-Prinzips führt zu einer Abschwächung von Abhängigkeiten zwischen Modulen. Insbesondere werden dadurch statische Abhängigkeiten der Module von ihren Clients vermieden.

Die Einhaltung des beschriebenen Prinzips kann schwierig sein, da es eine Prognose der zukünftigen Veränderungen und Nutzungsarten eines Moduls impliziert. Der für diese Prognose und eine entsprechende Gestaltung der Modulschnittstelle erforderliche Aufwand kann unter Umständen unberechtigt sein, wenn die vermuteten Erweiterungen nachher nicht eintreten. Es liegt also nahe, dass das *Open-Closed*-Prinzip nicht für alle Module gleichermaßen relevant ist, sondern vor allem für solche, die breit verwendet werden – möglicherweise von zuvor unbekannten Clients. Die Geltung dürfte für auslieferbare Klassenbibliotheken und "Utility"-Module besonders hoch sein.[5]

[5] vgl. dazu die Diskussion in [18]

3.4.2 Schnittstellentrennung

Beim Entwurf einer Modulschnittstelle sollte man an die unterschiedlichen Clients denken, die das Modul verwenden. Es kann sich dabei herausstellen, dass bestimmte Clients nur eine Untermenge der in einer Modulschnittstelle angebotenen Typen brauchen. In diesem Fall sollte man überlegen, ob das Modul nicht lieber in entsprechende Untermodule aufgeteilt werden sollte. Wären die Untermodule voneinander unabhängig, so erscheint ihre Isolierung vorteilhaft.

Die Modulschnittstelle entsteht zunächst als Vereinigung von Schnittstellen der modulinternen Klassen, mit Ausnahme solcher Klassen, die für die Client-Module komplett irrelevant sind. Es kann allerdings sinnvoll sein, die eins-zu-eins-Zuordnung zwischen den Implementierungsklassen und öffentlichen Interfaces eines Moduls aufzuheben. Ein Anlass dazu ist die Beobachtung, dass bestimmte Clients die angebotenen Interfaces nur teilweise nutzen, es gibt also Methoden, die von manchen Clients niemals aufgerufen werden. Es kann sich auch herausstellen, dass manche Clients mit einer groberen Schnittstelle als andere auskommen könnten – möglicherweise verwenden sie immer die gleiche Kombination von Methoden in gleicher Reihenfolge.[6] In solchen Fällen ist das Prinzip der Schnittstellentrennung (*Interface Segregation Principle* [15]) anzuwenden:

Clients sollten nicht gezwungen werden, von Schnittstellen abhängig zu sein, die sie nicht komplett nutzen.

Praktisch bedeutet dies, dass die Modulschnittstelle von den öffentlichen Schnittstellen (`public`-Methoden) der modulinternen Klassen abgekoppelt wird – möglicherweise werden mehrere Interfaces aus der Modulschnittstelle durch eine und dieselbe Klasse implementiert, oder umgekehrt: es gibt mehrere Klassen, die das gleiche Interface auf unterschiedliche Weisen implementieren. Für die Client-Module ist die Zuordnung zwischen Klassen und Interfaces völlig

[6] vgl. Facade-Pattern in [1]

transparent; für jeden Client kann so die bequemste und einfachste Schnittstelle angeboten werden. Darüber hinaus können die Implementierungsklassen innerhalb des Moduls ihre sonstigen `public`-Methoden aufrufen, die den Client-Modulen unbekannt sind.

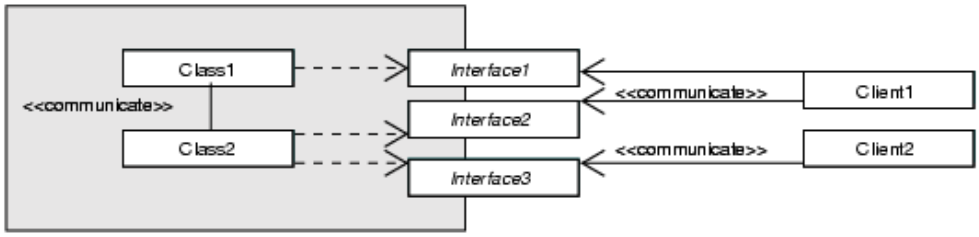

Abbildung 3.8: Beziehungen zwischen Clients eines Moduls, Interfaces und Implementierungsklassen

3.4.3 Umkehrung von Abhängigkeiten

Der traditionelle Ansatz der Top-Down-Programmierung schreibt einen hierarchischen Aufbau der Module vor. Die übergeordneten, allgemeinen Module verwenden und steuern Untermodule, die auf die Erfüllung bestimmter Aufgaben spezialisiert sind, aber ihrerseits auf noch feineren Hilfsmodulen basieren und so weiter. In der Regel enthalten die Untermodule keine Rückreferenzen zu ihren Aufrufern, und verdanken dieser Tatsache ihre Wiederverwendbarkeit.

Dieser Ansatz ist leicht verständlich und bringt die ersehnte Ordnung in ein Softwareprojekt. Zugleich birgt er auch eine Gefahr in sich. Verlassen sich die Module zur Erfüllung ihrer Funktion auf konkrete Klassen aus Untermodulen, so müssen sie zwingend zusammen mit diesen Untermodulen genutzt werden. Daraus folgt, dass man bei der Wiederverwendung immer einen ganzen Unterbaum aus der Modulhierarchie herausgreift, und nicht die einzelnen Module. Das ist recht inflexibel – will man den vorhandenen Code woanders nutzen, so ist man gezwungen, entweder (fast) alles oder kaum etwas aus dem Ursprungsprojekt zu entleihen. Das Problem wird zusätzlich dadurch hervorgehoben, dass

Änderungen in Untermodulen und das Hinzufügen neuer Untermodule Anpassungen in Modulen erfordert, die sich höher in der Hierarchie befinden. Diese Erscheinung weist übrigens auch auf die Verletzung des bereits beschriebenen *Open-Closed*-Prinzips hin.

Zur Vermeidung der negativen Folgen eines streng hierarchischen Modulaufbaus kann das Prinzip der Abhängigkeitsumkehrung (*Dependency Inversion Principle* [14], hier in einer etwas geänderter Fassung zitiert) umgesetzt werden:

Die Implementierung übergeordneter Module soll nicht von dieser der untergeordneten Module abhängen. Beide Implementierungen sollten von abstrakten Schnittstellen abhängen.

Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

Praktisch bedeutet das, dass jede Situation, wo eine Klasse eine konkrete Klasse aus einem anderen Modul aufruft (oder Instanzen davon erzeugt), untersucht werden sollte. Insbesondere muss man überlegen, welche Methoden aus der verwendeten Klasse für die Client-Klasse wirklich relevant sind. Sie sollten dann in einer abstrakten Schnittstelle erfasst werden, die von der Server-Klasse implementiert und von der Client-Klasse genutzt wird. Die Aussonderung der Schnittstelle ist kein rein automatischer Prozess, der auf der Quellcodeebene erfolgt oder etwa nur in statisch getypten Programmiersprachen notwendig ist. Vielmehr handelt es sich um eine Wegabstrahierung der irrelevanten Merkmale einer konkreten Implementierung. In der Regel werden dabei Methoden allgemeiner als zuvor benannt. Möglicherweise muss man die in ihren Signaturen vorkommenden Klassen ebenfalls verallgemeinern und durch abstrakte Schnittstellen ergänzen. In diesem Prozess wird Information versteckt – hier sehen wir also wieder das allgemeine *information hiding*-Prinzip im Einsatz.

Ist man mit den entstandenen Abstraktionen zufrieden, so bleibt noch die Entscheidung, zu welchem der beiden Module sie gehören sollten. Verschiebt

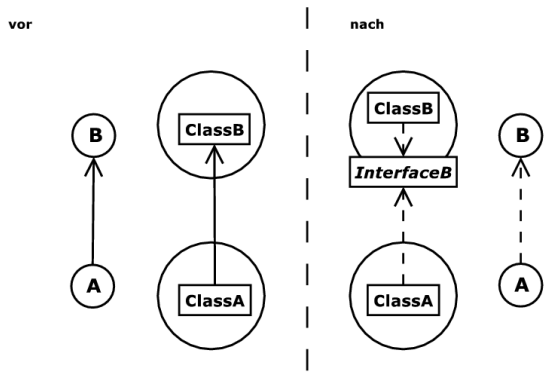

Abbildung 3.9: Umkehrung der Abhängigkeiten, das Interface bleibt im Implementierungsmodul

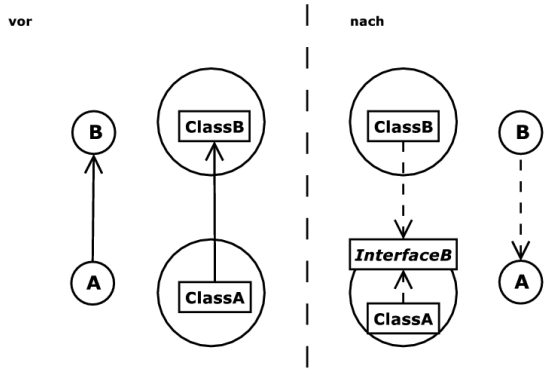

Abbildung 3.10: Umkehrung der Abhängigkeiten, das Interface wird zum Client-Modul verschoben

man Interfaces von dem implementierenden Modul zum Client-Modul, so hat man mit einer tatsächlichen Umkehrung der Abhängigkeit zwischen beiden Modulen zu tun.[7] Dies ist im Prinzip immer dann möglich, wenn das ehemalige Client-Modul keine Objekte direkt erzeugt, deren Typen im Server-Modul implementiert sind.

Hat man freie Wahl, so erscheint es sinnvoll, sich bei der Zuordnung von Interfaces zu Modulen nach ihrem breiteren Verwendungskontext zu richten. Gibt es noch andere Client-Module, die das Server-Modul über die gleichen Schnittstellen ansprechen könnten? In diesem Fall sollten die Interfaces in dem

[7] Die Umkehrung der Abhängigkeiten zwischen betroffenen Klassen passiert ohnehin.

implementierenden Modul bleiben (Abbildung 3.9). Beschreiben die Interfaces eher die Anforderungen des ursprünglichen Client-Moduls, die auch von anderen Server-Modulen erfüllt werden könnten? Dann sind sie besser in diesem Modul als Teil seiner öffentlichen Schnittstelle untergebracht (Abbildung 3.10). Üblicherweise sind beide Aspekte mehr oder weniger ausgeprägt – schließlich ist jeder Vertrag eine Sache zwischen allen beteiligten Parteien – die Auswahl des "Ablageorts" spielt eine geringere Rolle. Der meiste Nutzen ist alleine durch die Spezifizierung der abstrakten Schnittstellen erreicht – jedes Modul kann leichter wiederverwendet werden, ohne etwaige Untermodule zwanghaft mit sich in seinen neuen Verwendungskontext zu ziehen.

Es fällt auf, dass die Umkehrung der Abhängigkeiten auf der Modulebene die Entstehung von Frameworks fördert. Die Framework-Klassen dürfen nicht von konkreten Implementierungsklassen außerhalb des Frameworks abhängen, da man sie sonst nicht in anderen Projekten wiederverwenden könnte. Außerdem müssen alle Abstraktionen, die in einem Framework vorkommen, zu seinem Teil gemacht werden. Der Vertrag eines Frameworks ist einem Franchise-Vertrag ähnlich: er wird vom Franchise-Geber (hier: Framework) formuliert, vielen Annehmern (Framework-Benutzern) in derselben Form angeboten, beinhaltet Geschäftsideen (die "Ablaufsteuerung") und verspricht die Unterstützung bei der Umsetzung der Geschäftsziele (vorgefertigte Funktionen).

3.4.4 Bewertung von Abhängigkeiten

Die Modulabhängigkeiten eines Projekts spiegeln die tatsächlichen und potentiellen Kommunikationsflüsse zwischen Modulen wider. Das Vorliegen einer Abhängigkeit weist darauf hin, dass es im Client-Modul Typen gibt, die **möglicherweise** Nachrichten an Typen aus dem Server-Modul senden. Diese Information reicht sicher noch nicht aus, um Softwareentwürfe zu bewerten. Ein Entwurf mit wenigen Abhängigkeiten kann durchaus schlechter als einer mit mehreren sein. Es ist offenbar, dass die Anzahl von Abhängigkeiten durch

die Modulanzahl nach oben beschränkt ist – ein Entwurf mit wenigen Modulen wird automatisch auch weniger Abhängigkeiten haben. Eine einfache Abhilfe wäre die Betrachtung des Verhältnisses der Modulanzahl zur Anzahl von Abhängigkeiten. Auch dies erscheint schon nach einer kurzen Überlegung nicht genug aufschlußreich: zwei Entwürfe, die genau die gleiche Modulanzahl und den identischen Abhängigkeitenverlauf zwischen Modulen haben, können immerhin sehr unterschiedlich sein, weil die Modulinhalte anders sind.

Diese Beobachtungen legen nahe, dass man den einzelnen Modulabhängigkeiten (subjektive) Gewichte zuordnen kann. Die **Stärke** einer Abhängigkeit kann durch die Beantwortung folgender Fragen abgeschätzt werden:

- wie schwierig ist es, die Abhängigkeit zu beseitigen?
- welche Auswirkungen hat sie, wenn man die betroffenen Modulen ändert?

Starke Abhängigkeiten können zur unerwünschten Starrheit und Zerbrechlichkeit des Programmcode führen. Sie beeinflußen negativ die Beweglichkeit der Module. Zusätzlich sollte man noch überlegen, ob eine vorliegende Abhängigkeit das Verständnis der betroffenen Module eher behindert oder fördert. Wenn nur eine kleine Untermenge von Typen aus einem Modul von einer ebenfalls kleinen Typgruppe aus einem anderen abhängt, so ist die Modulabhängigkeit kaum aufschlussreich. In diesem Fall kann man durch die Modulzerteilung und die damit verbundene Einführung neuer, “sprechender” Abhängigkeiten die beabsichtigte Programmstruktur hervorheben.

R. Martin bedient sich zur Bewertung von Paketabhängigkeiten des Begriffs “Stabilität”. [17] Ein Paket[8] wir als stabil bezeichnet, wenn es sich schwer verändern lässt. Dies ist dann der Fall, wenn viele andere Pakete davon abhängen, da Änderungen sich auf sie eventuell ausbreiten. Martin formuliert nach der Begriffseinführung das folgende Prinzip (*Stable Dependencies Princi-*

[8]dieser Begriff wird von Martin als “Gruppe von Klassen” verwendet, ähnlich wie das Wort “Modul” in dieser Arbeit

ple):

Die Abhängigkeiten zwischen Paketen sollten in der Richtung der Paketstabilität verlaufen. Ein Paket sollte nur von solchen Paketen abhängig sein, die stabiler als es selbst sind.

Dieses Prinzip ist leider weniger brauchbar, als es zunächst erscheint. Das Problem besteht darin, dass einerseits die Paketstabilität von den Abhängigkeiten abgeleitet wird, und andererseits die Abhängigkeiten mit Hilfe der Stabilität bestimmt werden sollen. Die Verwendbarkeit ist also auf den Fall des Hinzufügens neuer Module beschränkt (allgemeiner: auf die Abhängigkeitsanalyse eines Moduls unter der Voraussetzung der Richtigkeit aller anderen Abhängigkeiten).

Möchte man die Stärke einer Abhängigkeit besser einschätzen, so ist es hilfreich, ihren syntaktischen Charakter zu untersuchen. Die Verwendung einer Referenz zur Schnittstelle eines Typs lässt mehr Spielraum bei Veränderungen als das Ansprechen einer konkreten Klasse. Dieses ist vermutlich leichter abzuschaffen als eine Vererbungsbeziehung, die eine Abhängigkeit nicht nur von aufgerufenen Methoden, sondern vom gesamten Vertrag einer konkreten Basisklasse darstellt. Eine direkte Objekterzeugung mit der Namensangabe einer Implementierungsklasse ist schwerwiegender als die Delegation derselben Aufgabe zu einer abstrakten Factory. Aggregation (Objekt ist Bestandteil eines anderen) ist stärker als Assoziation (Objekt wird zwar als Attribut verwendet, aber nicht untergeordnet), die wiederum stärker als das Vorkommen der Typreferenz in einer Methodensignatur erscheint. Die Syntaxanalyse ist ein effizientes Mittel, sich die erste Orientierung zu verschaffen, da sie sich durch ein Werkzeug gut unterstützen lässt und kein tiefes Verständnis der Programmbedeutung erfordert.

Die Untersuchung von Abhängigkeitsstärken liefert zwar wertvolle Hinweise auf das mit zukünftigen Änderungen verbundene Risiko. Sie ist aber, ähnlich wie die einfache Zählung, noch nicht zur Bewertung eines Entwurfs hinreichend.

Zusätzlich zum obigen Prinzip der stabilen Abhängigkeiten empfiehlt Martin noch das folgende (*Stable Abstractions Principle*):

Pakete, die am stabilsten sind, sollten am abstraktesten sein. Instabile Pakete sollten konkret sein. Die Abstraktheit eines Pakets sollte proportional zu seiner Stabilität sein.

Auch hier könnte man das Wort Paket mit dem Modulbegriff ersetzen. Es fällt dann auf, dass es sich hier um eine andere Fassung des Prinzips der Umkehrung von Abhängigkeiten handelt. Martin argumentiert, dass im Gegensatz zu Klassen die Abstraktheit von Paketen nicht ohne weiteres feststellbar ist: eine Klasse kann entweder eindeutig abstrakt sein oder nicht, aber ein Paket kann ja mehrere Klassen enthalten, von denen nur manche abstrakt sind.

Aus dieser Beobachtung folgt der Gedanke, zur Messung der Paketabstraktheit eine Metrik einzuführen. Martin schlägt vor, das Verhältnis der Anzahl abstrakter Klassen zur Gesamtanzahl von Klassen im Paket auszurechnen. Dieses Bewertungsverfahren erscheint zwar besser als nichts, dennoch ist bei seiner Anwendung Vorsicht geboten. Es ist nämlich vorstellbar, dass zwei Module, die auf Grund der Anzahl von Interfaces und Implementierungsklassen den gleichen Wert der Abstraktheitsmetrik aufweisen, sich in der Wirklichkeit auf sehr unterschiedlichen Abstraktionsebenen befinden. Zum Beispiel könnten die angesprochenen Module zu zwei voneinander entfernten Schichten in einer geschichteten Architektur gehören. Wenn das Modul aus der untersten Schicht von einem aus der obersten abhängig wäre, so wäre es ein Verstoß gegen das Stabilitätsprinzip, obwohl man es nicht durch den Vergleich von Metriken feststellen könnte.

Den beiden Prinzipien “Umkehrung von Abhängigkeiten” und “stabile Abstraktionen” liegt die Beobachtung zu Grunde, dass die allgemeinen Bestandteile eines Systems von ihrer Natur her unveränderlicher als die spezifischen sind. Dies klingt plausibel: je mehr an Einzelheiten man von der Beschreibung eines Sachverhalts weglässt, desto größer die Wahrscheinlichkeit, dass dieselbe

Beschreibung auch auf Varianten zutrifft.

Wie soll man aber die Abstraktheit eines Moduls messen – oder mindestens mit dieser eines anderen Moduls vergleichen – wenn die einfachen Metriken unzuverlässig erscheinen? Ein möglicher Weg wäre, sich "feinere" Metriken zu überlegen. Andererseits kann man jedoch auf die quantitative Erfassung verzichten. Stattdessen kann man untersuchen, welche **Begriffe** verwendet werden müssten, um den Funktionsumfang eines Moduls zu beschreiben. Man betritt somit den Bereich der Linguistik, und die Analyse des Programmcode wird zum Abenteuer mit der natürlichen Sprache – oder zumindest mit dem Wortschatz des Anwendungsgebiets und unserer eigenen Implementierung.

Die meisten Begriffe können mit Hilfe von anderen definiert (oder "erläutert") werden, auf die man auf der Suche nach ihrer Bedeutung wiederum den gleichen Zerlegungsprozess anwenden kann. Die Länge der Begriffskette, die zu den "Grundbegriffen" eines Anwendungsgebiets führt, oder sogar zu Grundbegriffen unserer Alltagssprache, dürfte als inverses Abstraktheitsmaß des Startbegriffs gelten. Die Abstraktheit eines Begriffs ist also durch die Anzahl von möglichen Verallgemeinerungsschritten, die auf ihn anwendbar sind, bestimmt. Es handelt sich, im Gegensatz zu der Metrik von Martin, um ein unscharfes Maß, da es keine Vorschriften gibt, die den elementaren Verallgemeinerungsschritt festlegen. Qualitativ kann man sagen, dass ein Modul, dessen verwendeter Wortschatz aus sehr abstrakten Begriffen besteht, selbst sehr abstrakt ist und daher als Ziel von Abhängigkeiten vorkommen sollte. Ein solches, dessen Inhalte durch Begriffe näher erläutert werden, die in anderen Modulen definiert sind, gilt als konkreter als sie. Natürlich sind auch die in Modulen vorkommenden Begriffe nicht gegeben – sie können zum einen im Laufe der Entwicklung verallgemeinert oder konkretisiert werden, zum anderen kann man sie zwischen Modulen verschieben. Die beträchtliche Anzahl der Freiheitsgrade sollte uns jedoch von der Suche nach Verbesserungen nicht abschrecken.

3.5 Wiederverwendung und Wartung

In den vorherigen Abschnitten wurde mehrmals betont, dass die Unabhängigkeit der Module und die Abstraktheit ihrer Schnittstellen eine bessere Wiederverwendung des Code unterstützt. Dies ist zwar korrekt, man sollte aber beachten, dass die Befolgung der beschriebenen Prinzipien nicht **hinreichend** für eine erfolgreiche Wiederverwendung ist. Martin weist diesbezüglich darauf hin, dass man unter diesem Begriff keine einfache Übernahme einzelner Module in andere Projekte verstehen sollte (auch wenn sie leicht realisierbar ist). [16] Vielmehr sollte die Wiederverwendung eine Arbeitsteilung bedeuten: die Module sollten auch nach der Vergrößerung ihres Benutzerkreises immer noch zentral weiterentwickelt werden. Als Begründung dafür nennt man einerseits den Wunsch, in den Genuß der zukünftigen Verbesserungen des fremden Code zu kommen. Andererseits (pessimistisch betrachtet) geht es aber auch darum, sich von der aufwändigen Pflege desselben Code zu befreien – die Probleme des Modulentwicklers sollten nach der Wiederverwendung seines Werks möglichst nicht zu unseren eigenen werden. Wie bei geschäftlichen *make or buy*-Entscheidungen sind Vor- und Nachteile des "Outsourcing auf Modulebene" abzuwägen und die Interaktion der beteiligten Personen und Softwarebestandteile zu regeln.

3.5.1 Versionierung

Wenn die Module als wiederverwendbare Einheiten gelten sollen, so ist ihre Versionierung erforderlich. Die Einführung der Modulversionierung in einem Release-Tracking-System schafft Pufferzeiten für die Reaktion der Client-Module auf Veränderungen in den von ihnen verwendeten Modulen. Die Notwendigkeit dieser Verzögerung ergibt sich aus der Tatsache, dass man trotz sorgfältiger Gestaltung von Modulschnittstellen ihre Unveränderbarkeit nicht garantieren kann. Schnittstellenänderungen, sei es auf syntaktischer oder auf semantischer Ebene, erfordern intelligente Anpassungen bei ihren Benutzern.

Dies ist weniger ein Problem, wenn ein Modul nur innerhalb seines Ur-

sprungsprojekts existiert, und wenn sich der Quellcode von allen betroffenen Client-Modulen ebenfalls in der Reichweite des Entwicklers befindet. Bei der Wiederverwendung in anderen Projekten, die möglicherweise von anderen Programmierern vorangetreibt werden, sind allerdings Kommunikation und bewußte Synchronisation erforderlich.

Durch die Einführung von Versionsnummern für Module, die zur Wiederverwendung freigegebenen wurden, wird eine Orientierungsgrundlage für ihre Benutzer hergestellt. Zusätzlich müssen die Schnittstellenänderungen, die zwischen Versionen erfolgt sind, nachvollziehbar sein – etwa durch die Dokumentation oder ein Konfigurationsmanagementsystem. So können Fehler und Verwirrung, die aus unkoordinierten Änderungen der zusammenarbeitenden Module resultieren, vermieden werden. Auch wenn sich die Modulschnittstelle zwischen Versionen nicht ändert, bedarf der Umstieg auf eine andere Version eines Integrationstests. Der Grund dafür ist die Existenz von "impliziten" Abhängigkeiten, die im Abschnitt über die statische Quellcodeanalyse noch genauer erläutert werden.

Mit der versionierten Freigabe ist immer ein gewisser administrativer Aufwand verbunden, der sich dann erhöht, wenn die älteren Versionen parallel zu der neuesten aktualisiert werden (es kommt oft vor, dass in den veröffentlichten Versionen noch Fehlerkorrekturen erledigt werden). Dieser Aufwand ist dann vertretbar, wenn er durch den entstehenden Nutzen zumindest kompensiert wird:

- durch (mehrfache) Wiederverwendung der Module in anderen Projekten
- durch geringere projektinterne Koordination bei Modulveränderungen

Die Existenz des Versionierungsaufwands verursacht, dass sich einzelne Klassen schlecht als wiederverwendbare Einheiten eignen. Auch Module eines Programms werden in der Regel nicht vom Anfang an für ihre spätere Wiederverwendung konzipiert (anders als z. B. gesamte Klassenbibliotheken). Es sollte

jedoch leicht möglich sein, sie beim Bedarf zu versionieren und zu veröffentlichen.

3.5.2 Eingrenzung der Änderungen

Bei der Festlegung von Modulinhalten sollte man sich darum bemühen, dass die Modulgrenzen mit den logischen Bereichen übereinstimmen, die von nachträglichen Änderungen betroffen werden. In objektorientierten Sprachen bedeutet das, dass Klassen, die aus gemeinsamen Gründen angepasst werden müssen, zum gleichen Modul gehören sollen. Theoretisch wäre es bei der Einhaltung des *Open-Closed*-Prinzips immer der Fall (die Änderung würde sich immer auf das Hinzufügen eines neuen Moduls beschränken). In der Praxis sind jedoch Anpassungen der bestehenden Module ganz üblich und müssen strategisch verwaltet werden. [16]

Es ist schwierig, die zukünftigen Änderungen vorherzusagen. Stattdessen kann man beim Eintritt eines Änderungswunschs, der mehrere Module betrifft, überlegen, ob die Modulinhalte nicht so umstrukturiert werden könnten, dass eine ähnliche Änderung in der Zukunft nur auf ein einzelnes Modul beschränkt ist. Diesem Vorgehen liegt die Hoffnung zu Grunde, dass manche im Code erfassten Ideen beständiger als andere sind. Ziele, die mit der Zuordnung von gemeinsam entwickelten Klassen zum selben Modul verfolgt werden, sind ökonomischer Natur:

- Minimierung der Anzahl der Module, die bei einer Änderung neu kompiliert und getestet werden müssen
- Minimierung der Anzahl der jeweils involvierten Personen(stunden)
- Verringerung des Kommunikationsbedarfs
- Verringerung der Missverständnisse, die bei der Kommunikation vorkommen

3.5.3 Zyklenfreiheit

Verwendet man Versionierung zur Freigabe von Modulen, um die Auswirkungen ihrer Veränderungen auf Client-Module zeitlich zu verzögern, so muss man beachten, dass in jede Versionsnummer implizit die Versionen aller Module hineinfließen, von denen das betrachtete Modul zum Zeitpunkt der Freigabe **transitiv** abhängt (Abbildung 3.11). Diese bedauerliche Tatsache resultiert aus der bereits erwähnten Eventualität der inkompatiblen Schnittstellenänderungen. Die Beschränkung der Anzahl der "mitfreigegebenen" Module ist ein weiteres Argument für die Beseitigung der Modulabhängigkeiten.

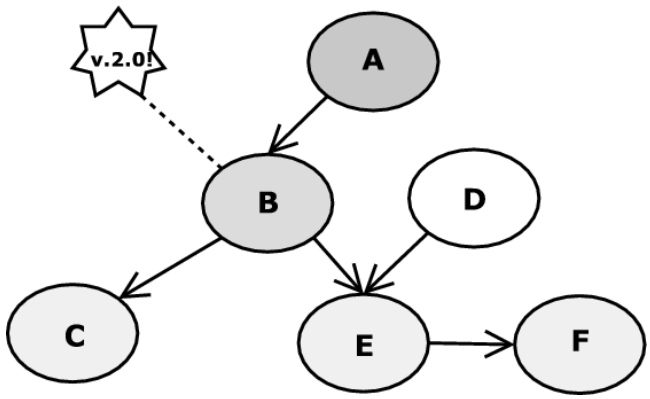

Abbildung 3.11: Rolle der Abhängigkeiten bei der Modulversionierung

In Abbildung 3.11 ist die Rolle der Abhängigkeiten bei der Versionsfreigabe veranschaulicht. Die hellgrau hintelegten Module fließen in die Freigabe der neuen Version des Moduls B mit ein und müssen zum Bauen des abhängigen Moduls A ebenfalls wie das Modul B ausgecheckt werden. Die Entwickler des Moduls D bleiben von diesem Release allerdings unbetroffen.

Eine besonders gravierende Wirkung haben bei der Versionsfreigabe Zyklen, die im Modulabhängigkeitsgraph auftreten können. Alle Module, die Knoten eines Zyklus sind, müssen zusammen freigegeben werden, da ansonsten Deadlock-Situationen entstehen: zur Freigabe des einen Moduls ist die letzte freigegebene Versionsnummer des anderen erforderlich, für deren Bildung wiederum die Versionsnummer des freizugebenden Moduls benötigt wird. Anstatt

die Versionsnummern miteinander zu synchronisieren und viele ansonsten direkt unabhängige Module immer zusammen freizugeben, sollte man sich eher um die Zyklenfreiheit in Abhängigkeitsgraphen kümmern. Zum Glück ist ihre Realisierung mit Hilfe der bereits dargestellten Umkehrung der Abhängigkeiten in meisten Fällen nicht schwierig. Eine andere Technik, mit der man Zyklen ebenfalls beseitigen kann, ist die Aussonderung von gemeinsamen Untermodulen.

Die Beseitigung von Zyklen in Modulabhängigkeiten bietet Vorteile nicht nur für die unabhängige Versionierung und Modulfreigabe. Ein weiterer wichtiger Grund für die Zyklenfreiheit ist das erleichterte Verständnis des Quellcode. Um die Funktion eines Moduls zu begreifen, benötigt man zuvor Wissen über Module, die vom betrachteten verwendet werden. Verfolgt man in der Suche nach Informationen die Abhängigkeitskette rekursiv, so stellt sich bei vorhandenen Zyklen heraus, dass man sich parallel mit allen beteiligten Modulen beschäftigen muss, um die Rolle eines beliebigen einzelnen Knoten zu verstehen. Vermutlich wird die Dauer des Lernprozesses nicht nur durch die Existenz, sondern auch durch die Stärke (Anzahl und Art) der vorliegenden Abhängigkeiten mitbestimmt. Die besonders einfachen Abhängigkeiten dürften leicht außer Acht gelassen werden, um die Rekursion “mental zu überwinden”. Es ist aber offensichtlich, dass die Beseitigung der Zyklen das Problem der unbekannten Lernreihenfolge vollständig löst und deswegen anzustreben ist.

3.6 Schwächen der statischen Quellcodeanalyse

Die statische Analyse des Quellcode, also die Suche nach expliziten Referenzen im Programmtext, ist ein unkompliziertes Verfahren, welches sogar für ein mittelgroßes Projekt eine umfangreiche Menge von Abhängigkeiten ermittelt. Es wäre jedoch ein Fehler, dieses Ergebnis als einzige Grundlage für die Befassung

mit dem Softwareentwurf zu nehmen. Man kann leicht zeigen, dass die statische Analyse unvollständig ist. Es gibt durchaus interessante Abhängigkeiten, die von unserer einführenden Definition nicht erfasst sind. In diesem Abschnitt werden einige Beispiele aus dem Kontext von Java vorgestellt. Nimmt man noch andere Programmiersprachen hinzu, so wird die Menge der durch die statische Quellcodeanalyse unerfassbaren Abhängigkeiten sicher noch umfangreicher.

Das erste Beispiel, dass jedem Java-Programmierer direkt einfallen sollte, ist die Verwendung von Reflection [24]. Eine abgeschlossene Menge der in einem Java-Programm verwendeten Klassen kann im Allgemeinen nicht während der Compilierung bestimmt werden. Ein Programm könnte während der Laufzeit einen Klassennamen dynamisch erzeugen (z. B. aus einer Konfigurationsdatei auslesen), ein Objekt instanziieren und Methoden auf diesem Objekt durch die Angabe ihrer Namen (die ebenfalls dynamisch konstruiert werden können) und Parameterwerte aufrufen. Die Verwendung von Reflection wird oft empfohlen, um die Abhängigkeiten von konkreten Implementierungsklassen loszuwerden. Dies trifft sehr wohl auf die Compilierungsphase zu. Man sollte jedoch nicht vergessen, dass die tatsächliche Abhängigkeit während der Programmlaufzeit immerhin existiert.

Wenn man Reflection verwendet, so tut man es meistens bewusst und hat gute Gründe dafür. Es ist nicht sehr schwierig, die damit verbundenen versteckten Abhängigkeiten im Hinterkopf zu behalten, vor allem, wenn man den Einsatz des Reflection-API nur auf wenige Module beschränkt. Es gibt leider Abhängigkeiten, die viel subtiler sind.

Es klingt zunächst überraschend, dass ein Programm durch solche Software (negativ) beeinflußt werden kann, die mit ihm keine definierten Schnittstellen besitzt. Diese Aussage ist aber bis auf einige wenige kontrollierte Umgebungen richtig, wenn man berücksichtigt, dass die heutigen führenden Betriebssysteme eine gemeinsame Nutzung von Ressourcen durch mehrere “unabhängige” Prozesse erlauben. Das Betriebssystem bietet zwar einen Speicherschutz und

bemüht sich, die beschränkten Ressourcen so zu verwalten, dass die Prozesse von der Existenz ihrer Nachbarn nicht erfahren. Das Problem der Ressourcenverteilung ist jedoch nicht allgemein lösbar, wenn die Nachfrage das Angebot übersteigt – eine Situation, die in der Welt der Computersysteme gar nicht unüblich ist. Außer Beschränkungen in der Hardwareleistung kommen noch Probleme mit der Verwaltung des gemeinsamen Zugriffs auf logische Ressourcen, wie z. B. Dateien hinzu.

Die Abhängigkeiten, die sich aus der Knappheit gemeinsam verwendeten Ressourcen und damit verbundenen "Seiteneffekten" ergeben, möchte man als Softwareentwickler gerne vernachlässigen. Schließlich sei es die Sache des Anwenders, wenn ein System überlastet oder mit "inkompatibler" Software betrieben wird. Tatsächlich kann man in den meisten Fällen wenig dagegen unternehmen, außer eine bestimmte Laufzeitumgebung vorzuschreiben. Doch es gibt eine indirekte Lösung – man sollte dafür sorgen, dass seine Software die anderen Programme nicht negativ beeinflußt, also mit gemeinsamen Ressourcen sparsam umgehen und Situationen vermeiden, die zu Notzuständen in unabhängigen Anwendungen führen könnten.

Erwähnenswert sind auch solche Abhängigkeiten, die sich aus der Verwendung gemeinsamer Datenformate und Kommunikationsprotokollen ergeben. Auch sie werden von der statischen Quellcodeanalyse in der Regel nicht beleuchtet. Die Schwierigkeiten ergeben sich nicht nur bei der Kommunikation zwischen verschiedenen Programmen, die aufeinander nicht gut genug "abgestimmt" sind (d.h. die formale Spezifikation des Datenformats oder Protokolls verletzen, wenn eine solche existiert). Das Verlassen auf bestimmte Datenformate kann auch innerhalb einer Anwendung zu argen Problemen führen. Es passiert immer dann, wenn die Formate sich unabhängig von dem Code verändern (oder vice versa). Darüber hinaus können die Daten selber Abhängigkeiten (Konsistenz- und Integritätsregeln) enthalten, die mit Hilfe der Quellcodeanalyse natürlich nicht aufzudecken sind. Die Klassenabhängigkeiten loszuwerden,

indem man sie in einer externen Datenbank vergräbt, scheint in diesem Zusammenhang eine besonders schlechte Idee zu sein.

Im Bereich der webbasierten Anwendungen gibt es besondere Skriptsprachen, mit denen die Erzeugung der für den Browser bestimmten Ausgaben erleichtert werden soll (Beispiele aus der Java-Welt sind: JSP, WebMacro, Velocity). Diese Skriptsprachen bieten alle die Möglichkeit, Java-Objekte zu manipulieren, oder sogar neue Klassen zur Bearbeitung von HTTP-Anfragen zu definieren. Die in den Skriptsprachen enthaltenen statischen Abhängigkeiten liegen heute außerhalb des Wirkungsbereichs der meisten Analysewerkzeuge. Bei ihrer Verwendung ist daher Vorsicht geboten. Die Situation verschlimmert sich bei solchen Skriptsprachen, die dynamisch typisiert sind. Sie alle verlassen sich zur Erledigung ihrer Aufgaben auf Reflection – mit den zuvor erwähnten Nachteilen. Die bequeme Schreibweise und die wahrgenommene Einfachheit solcher Sprachen sollte also gegen Probleme abgewogen werden, die sich aus der erschwerten Analyse ergeben können – allgemein sollte ihr Einsatz dort erfolgen, wo sie den meisten Nutzen bieten, d.h. bei der Outputerzeugung.

Schließlich ist zu bedenken, dass die mit der statischen Analyse festgestellten Abhängigkeiten keine Schlüsse auf die Beziehungen zwischen Objekten zur Laufzeit ziehen lassen. Zwei Module, die von der statischen Struktur her völlig unabhängig sind, können ihr Verhalten gegenseitig beeinflußen, besonders dann, wenn ein Datentransport über mehrere Objekte hinweg erfolgt. Dies bedeutet, dass die Ursache eines Fehlers, der in einem Modul zum unerwünschten Verhalten führt, nicht unbedingt im gleichen Modul zu suchen ist. Es gibt Techniken, die einen Schutz gegen die Ausbreitung von Folgefehlern im laufenden Programm führen. An dieser Stelle denke man an die “defensive” Programmierung, Zusicherungen und das *Law of Demeter* [27].

Kapitel 4

Modulbildung für Classdep

4.1 Motivation

In vorherigen Kapiteln wurde angemerkt, dass die Erkennung von Abweichungen zwischen dem Ist- und Soll-Entwurf mit Hilfe der statischen Quellcode-Analyse unterschiedlich erfolgreich sein kann. Insbesondere können die Verletzungen von dargestellten Entwurfsprinzipien durch die syntaxgesteuerte Quellcodeanalyse nur indirekt festgestellt werden, da der Zusammenhang zwischen der syntaktischen Struktur einer Compilierungseinheit und ihrer Bedeutung nicht einfach automatisch zu ermitteln ist. Dieses Problem ist beim Reverse-Engineering wohl bekannt. Bei der Compilierung gehen manche Informationen verloren, die im Quellcode noch enthalten sind. Ähnlich gibt es auch bei der Umsetzung eines abstrakten Anwendungsmodells in eine konkrete Implementierung Informationsverluste. Noch anders ausgedrückt, mehrere Abbildungen des gleichen Sachverhalts auf Mengen von Quellcodedateien sind möglich.

Man möchte aus der Analyse des Quellcode Rückschlüsse auf Entwurfsschwächen ziehen, die möglicherweise auf einer abstrakteren Ebene – der Modulebene – korrigierbar sind. Es soll im Folgenden untersucht werden, ob die Analyse durch Einhaltung von bestimmten Programmierkonventionen unterstützt werden kann. Da die gängigen Programmiersprachen unser äußerst hilfreiches

Modulkonzept nur ansatzweise unterstützen, muss man sich dabei mit vorhandenen Ausdrucksmitteln begnügen. In diesem Kapitel wird ein Verfahren vorgeschlagen, wie man mit Hilfe von Java-Paketen den modularen Entwurf eines Programms widerspiegeln kann, um spätere Auswertungen der Quellcodeanalyse zu erleichtern. Das Ziel ist, aus den Analyseergebnissen möglichst viele direkt verwendbare und keine überflüssigen Informationen herauszuholen. Man sucht nach konkreten Antworten auf folgende Fragen:

- welche Entwurfsprinzipien werden im analysierten Code verletzt?
- welche unerwünschten Konsequenzen ergeben sich aus ihrer Nichteinhaltung?
- welche Maßnahmen auf Quellcodeebene sind erforderlich, um die Situation zu verbessern?

4.2 Abbildung von Modulen auf Java-Pakete

Jedes Java-Programm besteht aus Modulen, die untereinander Nachrichten austauschen müssen, um die Benutzeranforderungen zu erfüllen. Jedes Modul setzt sich laut Definition aus Kapitel 1 aus mehreren Typen (Klassen und Schnittstellen) zusammen. Wenn man ein Modul einzeln betrachtet, sollte eine klare Zuordnung der enthaltenen Klassen und Interfaces entweder zum **Implementierungsteil** oder zur **öffentlichen Modulschnittstelle** möglich sein. Der Zweck dieser Gruppierung ist, solche Informationen zu verstecken, die für andere Module irrelevant sind. Man möchte die Modulschnittstelle vereinfachen und ihren Gebrauch für andere Module erleichtern. Die modulinterne Information darf durch alle Implementierungsklassen innerhalb eines Moduls gemeinsam genutzt werden, natürlich mit der üblichen objektorientierten Datenkapselung.

Je mehr Information man hinter einer Modulschnittstelle verstecken kann, um so besser. Den Beitrag eines Moduls zum gesamten Programm kann man

an der Einfachheit der Modulschnittstelle, an der Komplexität der Implementierung und an der Anzahl von Client-Modulen messen. Es gelten grundsätzlich die gleichen Regeln wie bei der Bildung einzelner Klassen, nur werden jetzt als Bausteine Typen statt Methoden und Datenvariablen betrachtet.

Wie kann man die vorgenommene Modulzerteilung in der statischen Quellcodestruktur wiederfinden? Es gibt bei Java-Programmen (mindestens) zwei verschiedene Wege, die eine nachherige Analyse erleichtern:

- jedes Modul wird auf genau ein Java-Paket abgebildet. Typen, die zum Implementierungsteil gehören, werden als paketintern deklariert (kein `public`-Schlüsselwort in der Typdeklaration). Typen, die zur Modulschnittstelle gehören, werden dagegen als öffentlich deklariert.

- jedes Modul wird auf genau zwei Java-Pakete abgebildet: ein Schnittstellenpaket (`com.example.module`) und ein Implementierungspaket (`com.example.module.internal`). Alle Typen, die zum Schnittstellenpaket gehören, werden als öffentlich deklariert. Fast alle Typen, die zum Implementierungspaket gehören werden als paketintern deklariert. Eine Ausnahme sind solche Typen, die für die Erzeugung nach außen sichtbarer Objekte verantwortlich sind oder wegen anderer Konventionen[1] nicht paketintern sein dürfen.

Der zweite Weg ist dann zu bevorzugen, wenn die Modulschnittstelle mehr als eine Implementierung haben kann. Außerdem macht die konsequente Verwendung von zwei Paketen je Modul die (neuen) Entwickler aufmerksamer auf den modularen Aufbau als die einfachere Einpaketlösung, deren Absicht eventuell übersehen werden könnte. Im Allgemeinen sind die Clients eines Moduls nur an den zum Schnittstellenpaket gehörigen Typen interessiert. Außerdem möchten manche Clients Instanzen von diesen Typen erzeugen. Hierzu können sie die öffentliche `Factory`-Klasse aus dem Implementierungspaket gebrauchen.

[1] z. B. wegen Anforderungen eines verwendeten Frameworks

Der Vorteil der beiden dargestellten Paketbildungsarten ist, dass die konkreten Implementierungsklassen aus einem Modul von keinen Client-Modulen direkt instanziiert oder referenziert werden können. Diese Tatsache wird dank ihrer paketinternen Zugriffsbeschränkung durch den Compiler garantiert.

Alle verbliebenen statischen Abhängigkeiten sind demzufolge solche, die von einem Modul zur Schnittstelle eines anderen Moduls bzw. zu der für die Objekterzeugung bestimmten Implementierungsklasse gerichtet sind. Die Aufgabe der Analyse besteht dann darin, die Anzahl der Abhängigkeiten durch die Verbesserung der Modulschnittstellen zu reduzieren und eventuell vorhandene Abhängigkeitszyklen zu beseitigen.

4.3 Inhalt der Modulpakete

Die Beziehung zwischen dem Implementierungspaket und dem Schnittstellenpaket eines Moduls bedarf einer genaueren Betrachtung:

- das Schnittstellenpaket enthält zum größten Teil nur Interfaces
- außer Interfaces sind dort auch einfache Werteklassen (*value objects*), Enum-Klassen und Exceptions vorstellbar
- enthält das Modul Klassen, die zur Vererbung in anderen Modulen entworfen sind, so werden sie im Implementierungspaket als `public` deklariert
- das Implementierungspaket enthält konkrete Klassen, die Interfaces aus dem Schnittstellenpaket implementieren
- das Implementierungspaket enthält eine Singleton-Klasse namens *ModuleName*`Factory` mit Methoden, die nach dem Muster `create`*TypeName* benannt werden; diese Methoden verwenden Interfaces aus dem Schnittstellenpaket als Parameter- und Rückgabewerte

- das Schnittstellenpaket darf keine Referenzen zum Implementierungspaket enthalten
- in Klassen des Implementierungspakets werden falls möglich Typreferenzen verwendet, die zum Schnittstellenpaket führen; eine direkte Verwendung von konkreten Klassennamen ist im Implementierungspaket auch zulässig
- die Objekterzeugung im Implementierungspaket passiert entweder direkt mit dem Operator `new` oder mit Hilfe einer Factory-Klasse; handelt es sich um Objekte aus einem anderen Modul, so muss seine Factory verwendet werden

Die wichtigsten von diesen Regeln sind diese, die die Verwendung von Interfaces im Schnittstellenpaket und die Verwendung von Factories bei der Objekterzeugung fordern. Wie am folgenden Beispiel deutlich wird, ermöglichen sie getrennte Modultests und eine verbesserte Wiederverwendbarkeit.

4.4 Beispiel

Das beschriebene Verfahren zur Einordnung des Modulinhalts in zwei Java-Pakete soll mit einem Beispiel besser veranschaulicht werden. In Abbildung 4.1 ist der Inhalt eines Programms vor der Aussonderung des Moduls `JCrypt` skizziert. Das geplante Modul soll als Funktion die Verschlüsselung von Zeichenketten übernehmen, die bisher in den drei Klassen `Encrypter1`, `Encrypter2` und `Key` im Hauptprogramm implementiert wurde.

Als Erstes wird die Modulschnittstelle (Paket `com.example.jcrypt`) von der Implementierung (Paket `com.example.jcrypt.internal`) getrennt. Der in den `Encrypter`-Klassen enthaltene Code gehört eindeutig zur Implementierung. Die beiden Klassen werden also nach `com.example.jcrypt.internal`

```
com.example
    ...
    Encrypter1.java     : public class Encrypter1
    Encrypter2.java     : public class Encrypter2
    Key.java            : public class Key
    ...
```

Abbildung 4.1: Ausgangssituation vor der Aussonderung des JCrypt-Moduls

```
com.example.jcrypt
    IEncrypter.java     : public interface IEncrypter
com.example.jcrypt.internal
    Encrypter1.java     : class Encrypter1 implements IEncrypter
    Encrypter2.java     : class Encrypter2 implements IEncrypter
```

Abbildung 4.2: Erster Schritt zum Modullayout (JCrypt)

verschoben, und der Zugriff wird in ihren Deklarationen auf paketintern gesetzt. Da sie in dieser Form für den Rest des Programms vollkommen unsichtbar wären, muss man die Signaturen ihrer öffentlichen Methoden mit Hilfe von Interfaces exportieren und nach `com.example.jcrypt` verschieben. Bei diesem Schritt bemerkt man oft, dass nicht alle `public`-Methoden jeder Klasse von den gleichen Modul-Clients gemeinsam genutzt werden. Es bietet sich an, die unnötigen Methoden aus Interfaces wegzulassen bzw. die Menge aller Methoden in Untermengen zu splitten, die in je einem Interface deklariert werden.[2] Umgekehrt können die Signaturen öffentlicher Methoden mehrerer Klassen auch in ein gemeinsames Interface verlegt werden. In unserem Beispiel wird ein Interface `IEncrypter` ausgesondert, das von den beiden `Encrypter`-Klassen implementiert wird (Abbildung 4.2).

Nehmen wir an, dass man bei der Klasse `Key` feststellt, dass ihre Implementierung nicht zum JCrypt-Modul gehören soll. Die in einem `Key` enthaltenen Informationen werden aber sehr wohl von `Encrypter1` verwendet. Eine Lösung ist, das Interface `IKey` in dem Schnittstellenpaket von JCrypt zu deklarieren

[2] vgl. Abschnitt 3.4.2, Schnittstellentrennung

und innerhalb von der Implementierungsklasse `Encrypter1` zu verwenden. Ein konkretes `IKey`-Objekt muss vom Client-Modul zur Erzeugung von `Encrypter1` geliefert werden. Auf diese Weise bleibt das Modul JCrypt für Veränderungen in der Schlüsselimplementierung offen – ein Beispiel für die Verwendung des *Open-Closed*-Prinzips.

Das JCrypt-Modul nutzt seinen Clients nichts, wenn es keine Möglichkeit gibt, die `Encrypter`-Objekte zu erzeugen. Zum Implementierungspaket wird deshalb eine Factory-Klasse hinzugefügt, die man mit dem Zugriffsmodifikator `public` deklariert (Abbildung 4.3). Die Client-Module können diese Klasse mit einem Aufruf wie `JCryptFactory.instance().createEncrypter(myKey)` gebrauchen. `JCryptFactory` ist ein Singleton – da das JCrypt-Modul im gesamten Programm nur einmal vertreten ist. Diese Eigenschaft gilt allgemein für alle Module.

Eine Besonderheit ist die Methode `JCryptFactory.setInstance`, deren Aufrufe von Client-Modulen auf Grund ihres paketinternen Zugriffs unerlaubt sind. Die Existenz dieser Methode ermöglicht einen Trick, der beim Testen von **Client-Modulen** von JCrypt besonders hilfreich ist. Es ist üblich, dass die Pakethierarchie für Testfälle die Pakethierarchie des getesteten Programms widerspiegelt. Diese Anordnung kann man ausnutzen, um eine Hilfsklasse bereitzustellen, die den Aufruf von `JCryptFactory.setInstance` **nur aus Testklassen** ermöglicht. Will man nun ein Modul testen, das von JCrypt Gebrauch macht, aber die eigentliche Implementierung der `Encrypter`-Klassen nicht unbedingt erfordert, so kann man am Anfang des Tests die JCrypt-Factory durch eine abgeleitete Klasse ersetzen, die vereinfachte oder sich speziell verhaltende Instanzen erzeugt.[3] Ein Grund für die Verwendung dieses Tricks in unserem Beispiel wäre, wenn die Erzeugung von `Encrypter`s lange dauern würde – was bei der Initialisierung von manchen kryptografischen Algorithmen durchaus plausibel klingt.

Es ist bemerkenswert, dass das Testen von JCrypt-Clients die Richtigkeit

[3] vgl. http://martinfowler.com/bliki/MakingStubs.html

```
package com.example.jcrypt.internal;

public class JCryptFactory
{
    private static JCryptFactory instance;

    public IEncrypter createEncrypter1(IKey key)
    {
        return new Encrypter1(key);
    }

    public IEncrypter createEncrypter2()
    {
        return new Encrypter2();
    }

    public synchronized static JCryptFactory instance()
    {
        if (instance == null) instance = new JCryptFactory();
        return instance;
    }

    synchronized static void setInstance(JCryptFactory instance)
    {
        this.instance = instance;
    }
}
```

Abbildung 4.3: Implementierung einer Modul-Factory-Klasse (JCrypt)

```
com.example.jcrypt
    IEncrypter.java     : public interface IEncrypter
    IKey.java           : public interface IKey
com.example.jcrypt.internal
    Encrypter1.java     : class Encrypter1 implements IEncrypter
    Encrypter2.java     : class Encrypter2 implements IEncrypter
    JCryptFactory.java  : public class JCryptFactory
```

Abbildung 4.4: Beispiel eines Modullayouts (JCrypt)

der JCrypt-Modulimplementierung weder erfordert noch überprüft. Alle Informationen, die zu einer kryptografisch sicheren Umwandlung eines Klartexts in seine verschlüsselte Form notwendig sind, bleiben den Client-Modulen verheimlicht. Bei der Verwendung von JCrypt kommt es nur darauf an, die Umwandlung überhaupt "irgendwie" durchzuführen.

Die resultierende Zuordnung von Typen zu Modulpaketen für das beschriebene Beispiel ist in Abbildung 4.4 zu sehen. Man sollte beachten, dass die Existenz eines Moduls wie JCrypt nicht vorab geplant werden muss, sondern sich erst dann herauskristallisiert, wenn erstens einige zusammenhängende Klassen in der entwickelten Programmstruktur bemerkt werden und zweitens ihre Isolierung vorteilhaft erscheint. Der Vorteil wäre die im Beispielfall die bereits erwähnte Beschleunigung der Testabläufe oder auch die Erhöhung der Verständlichkeit des gesamten Programms. Im Laufe der Entwicklung sollte eine sinnvolle Modulstruktur erkennbar werden und nach einiger Zeit ihre Stabilisierung eintreten. Inwieweit es erreicht ist, lässt sich empirisch feststellen: z. B. an der Anzahl von Typverschiebungen zwischen Modulen oder Schnittstellenanpassungen in einem bestimmten Zeitraum.

4.5 Gestaltung der Pakethierarchie

Dem Leser wird vielleicht auffallen, dass die beiden früher vorgeschlagenen Paketbildungsarten zu sehr flachen Pakethierarchien führen. Bei umfangreiche-

ren Projekten könnte diese Struktur unübersichtlich werden. Dieses Phänomen kann man reduzieren, indem man eine weitere Paketart zulässt: Pakete, die selber keine Modulteile darstellen, sondern lediglich zur logischen Gruppierung vorhandener Module dienen.

Es ist zum Beispiel vorstellbar (und oft empfehlenswert), alle Module eines Programms in zwei Kategorien zu unterteilen: die `ui`-Hierarchie und die `model`-Hierarchie. Die obersten Pakete `ui` und `model` gelten als reine Behälter: die eigentlichen Modulpakete befinden sich eine Stufe niedriger in der Pakethierarchie (z. B. `ui.editor`, `ui.graph` und `model.editor`, `model.graph`).

Ein anderer Fall, wo "Nicht-Modul-Pakete" hilfreich sein könnten, ist die Gruppierung von Klassen innerhalb des Implementierungsteils eines umfangreichen Moduls. Würde zum Beispiel der Implementierungsteil von `ui.editor` mehr als 20 Klassen enthalten, so wäre es naheliegend, dass man sie nach ihrer Art (irgendwie) sortiert und zwei zusammenarbeitende Implementierungspakete verwendet. Leider muss man in diesem Fall manche Klassen aus den beiden Implementierungspaketen als `public` bezeichnen, was den beabsichtigten Schutz vor direkten Client-Zugriffen verletzt.

Wird man mit dem Problem eines geschwollenen Implementierungspakets konfrontiert, so kann man sich überlegen, ob eine Modulzerteilung mit Hilfe des zuvor dargestellten Verfahrens nicht vorteilhafter wäre. Wenn es sich um ein Schnittstellenpaket handelt, so ist die Notwendigkeit der Modulzerteilung noch wahrscheinlicher. Die Begründung liegt darin, dass jedes Modul eine möglichst **einfache** Schnittstelle seinen Clients anbieten sollte.

Entscheidet man sich für den Weg der Modulzerteilung, um eine übermäßige Vergrößerung der Typanzahl je Modul zu verhindern, so hat man bald mit einer ebenfalls beträchtlichen Modulanzahl zu tun. Nimmt man zum Beispiel als Faustregel an, dass jedes Modul ungefähr 5-9 Implementierungsklassen enthalten soll, um beherrschbar zu bleiben, so steigt die Modulanzahl linear mit der Klassenanzahl im Gesamtsystem. In diesem Fall wäre empfehlenswert, die

Module mit Hilfe einer übergeordneten Struktur zusammenzufassen: vielleicht können alleinstehende Anwendungen ausgesondert oder andere "Untersysteme", die zusammen ein Ganzes bilden. Beispielsweise wurde im Eclipse-Projekt das Konzept eines Plugins als Einheit der Wiederverwendbarkeit und Versionierung eingeführt. [11] Auf diese Weise werden heute bereits Tausende von Klassen verwaltet, wobei für jeden Plugin-Entwickler nur die Schnittstellen der von ihm gebrauchten Drittplugins relevant sind. Obwohl die beschriebenen Prinzipien möglicherweise auch auf eine grobe Modularisierung großer IT-Systeme und Anwendungen verwendbar wären, wird auf ihre eingehende Untersuchung hier verzichtet.

4.6 Nachteile des vorgeschlagenen Verfahrens

Die hier vorgestellte praktische Herangehensweise zur modularen Verwendung von Java-Paketen ist relativ neu und noch kaum verbreitet. Sie wurde vom Autor erfolgreich bei der Programmierung des Werkzeugs Classdep eingesetzt und wird sicherlich noch in zukünftigen Projekten erprobt. An dieser Stelle sollten einige Überlegungen geäußert werden, die auf mögliche Schwachstellen des Verfahrens hinweisen.

Das Verfahrens führt zur Erhöhung der Gesamtanzahl von Paketen und Klassen – das ist eben der Preis für eine strikte Trennung zwischen Schnittstellen- und Implementierungsteilen im Quellcode in einer Programmiersprache, die die Bildung höherer Module nicht unterstützt. In Laufzeitumgebungen, wo die Anzahl von Klassen z. B. wegen Speicherknappheit eine Rolle spielt (eingebettete Systeme), muss man auf die Verwendung von vielen "überflüssigen" Interfaces möglicherweise verzichten. Dieses Argument ist mit dem vergleichbar, dass man in manchen Fällen aus Performanzgründen überhaupt höhere Programmiersprachen vermeiden sollte.

Die Klassenvererbung ist über Implementierungspakete hinweg unmöglich. Allgemein verwendbare Basisklassen müssen explizit in Schnittstellenpakete

verschoben werden, eine *ad hoc*-Vererbung, die man gerne für leichte Anpassungen von vorhandenen Klassen nutzt, ist ausgeschlossen. Die vorgeschlagene Art der Modularisierung eignet sich daher nicht besonders zur Erstellung von Klassenbibliotheken, sondern eher für komponentenweise Entwicklung, wo die einzelnen Module als *black boxes* mit klar vorgesehenen Erweiterungsmöglichkeiten agieren.

Die Verwendung von Singletons auf die vorgeschlagene Art und Weise ist zwar recht praktisch, aber sie könnte zu einem (zu) frühzeitigen Laden aller Implementierungsklassen eines Moduls führen. Es gibt Techniken, die ein verzögertes Klassenladen zwecks Performanzoptimierung verhüten. Wie mit den meisten Optimierungsmaßnahmen kann man aber ihren Einsatz auf den Zeitpunkt verschieben, wenn sie wirklich erforderlich werden.

Es gibt Situationen, wo man den Zugriff auf die konkreten Implementierungsklassen nicht willkürlich auf "paketintern" beschränken kann. Es ist immer dann der Fall, wenn die Klasse als Teil eines Frameworks verwendet wird, das die Instanziierung der Objekte selber übernimmt. Deklariert man die Implementierungsklassen als `public`, so können aber auch andere Module die Objekte direkt mit `new` erzeugen. Dadurch wird die Modulschnittstelle umgangen, es handelt sich also um unerwünschte Modulabhängigkeiten.

Ein typisches Beispiel sind alle Servlet-Klassen. Sie bilden lediglich eine Schnittstelle zwischen der Web-Anwendung und dem Servlet-Engine, aber sie müssen dennoch als `public` deklariert werden. Zum Glück kann man solche Klassen meistens vom Rest ihres Ursprungsmoduls isolierien. Man legt für sie ein neues Modul an, das zum Client des alten gemacht wird und selber von keinen Clients genutzt werden darf.

Schließlich ist nochmals zu unterstreichen, dass es sich um eine Konvention handelt – und Konventionen werden leicht verletzt. Wird das Verfahren bei der Programmierung in einem Team eingesetzt, so muss man sich darum kümmern, dass neue Teammitglieder damit vertraut werden. Wieviel Zeit braucht man für

eine komplette Initiation? Mit Sicherheit nicht weniger als die Lektüre dieses Kapitels dauert, wahrscheinlich nicht länger als seine Erarbeitung erforderte und hoffentlich einen Bruchteil davon, was man auf der Suche von vermiedenen Fehlern verbringen würde.

Kapitel 5

Funktionsweise und Einsatzmöglichkeiten von Classdep

In diesem Kapitel wird das entwickelte Werkzeug vorgestellt. Zunächst werden Funktionsmerkmale aufgeführt und ihr Bezug zu der syntaxbasierenden Abhängigkeitsanalyse erläutert. Im nächsten Kapitel wird dann die Verwendung von Classdep an einem Beispiel geschildert.

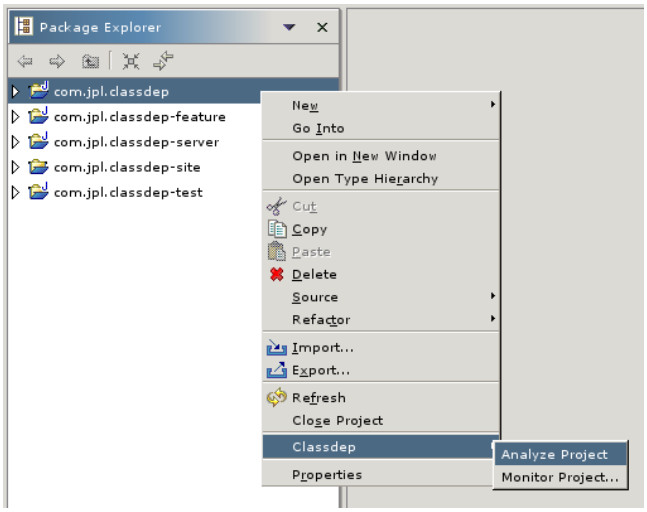

Abbildung 5.1: Auswahl eines Java-Projekts zum Anstoßen der Abhängigkeitsanalyse

Classdep wurde als Eclipse-Plugin entwickelt. Die gesamte Funktionalität wird also im Kontext der Eclipse-IDE angeboten (der interessierte Leser findet nähere Informationen im Kapitel über die Implementierung von Classdep). Zunächst wird ein Java-Projekt ausgewählt, für das die Abhängigkeitsanalyse durchgeführt werden soll (Abbildung 5.1). Nach der syntaktischen Analyse aller im Projekt befindlichen Java-Quellcodedateien wird von Classdep eine neue "Perspektive" dargestellt. Die Verarbeitung kann auf einem modernen Rechner einige Sekunden dauern – die Projektgröße ist dabei entscheidend. Bei der "Classdep-Perspektive" handelt es sich um eine Zusammenstellung von classdep-eigenen und von einigen zu Eclipse gehörenden Benutzeroberflächen, die gemeinsam zur Unterstützung der Ergebnisauswertung und Untersuchung des Quellcode dienen.

5.1 Referenzenvisualisierung

Der erste Schritt einer Abhängigkeitsanalyse besteht darin, die existierenden Abhängigkeiten zu ermitteln und auf eine für den Benutzer bequeme Art und Weise darzustellen. Betrachtet man den Quellcode eines Java-Projekts, so existieren die Abhängigkeiten, wie im einführenden Kapitel definiert, in erster Linie zwischen Klassen und Interfaces. Enthält der Quelltext einer Klasse oder Schnittstelle eine Referenz zu einer anderer Klasse oder Schnittstelle, so hat man mit einer Abhängigkeit zu tun. Als Referenzen wird jedes Auftreten des exaten Namens der referenzierten Klasse in der referenzierenden bezeichnet.

Es ergibt sich die Frage, wie man die riesige Menge von Klassenabhängigkeiten, die in einem Projekt normalerweise existieren, am sinnvollsten darstellen kann, ohne den Benutzer zu überfordern. In Classdep werden dazu die tabellarischen und grafischen Darstellungen des Abhängigkeitsgraphen angeboten, in denen die vorkommenden Klassen jeweils mit Hilfe ihrer umfassenden Java-Pakete gruppiert werden.

5.1.1 Tabellarische Darstellung

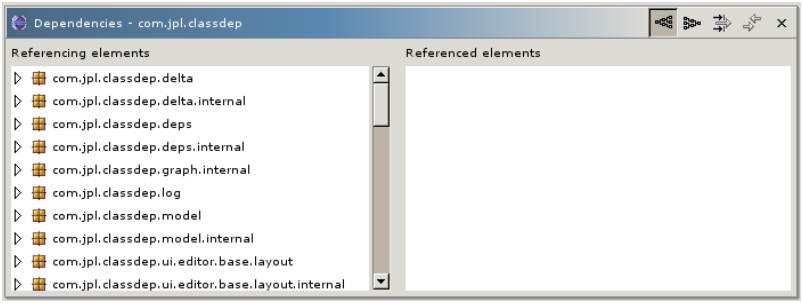

Abbildung 5.2: Tabellarische Darstellung der Abhängigkeiten (Einstieg)

Die Abbildung 5.2 zeigt die Benutzeroberfläche von Classdep nach der Auswahl eines zu untersuchenden Java-Projekts. Der Darstellungsbereich ist in zwei Hälften unterteilt: auf der linken Seite erscheinen immer die referenzierenden Elemente (wie z. B. Pakete und Klassen), auf der rechten Seite immer die referenzierten. Zunächst bleibt der rechte Teil leer, da in dem linken noch keine Auswahl vorgenommen wurde. Wählt man nun ein oder mehrere Elemente durch Anklicken aus, so werden die von ihnen referenzierten Elemente eingeblendet (Abbildung 5.3).

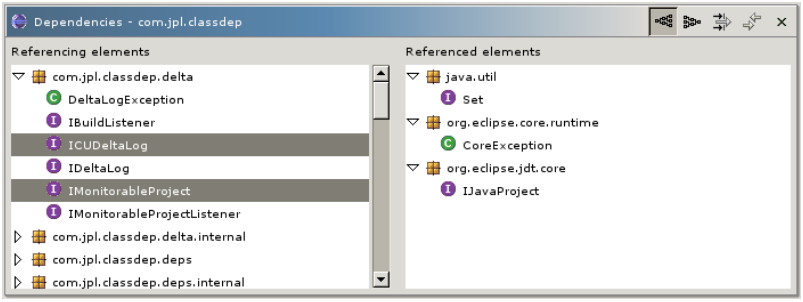

Abbildung 5.3: Tabellarische Darstellung der Abhängigkeiten (nach einer Auswahl)

Man kann die Abhängigkeiten ebenfalls von ihrem Ziel her angehen. Hierzu muss der Anzeigemodus auf referenzierende Elemente umgeschaltet werden. Die rechte Hälfte des Displays übernimmt dann die steuernde Rolle – wählt man dort ein Element aus, so werden in der linken Hälfte nur diejenigen Elemen-

te sichtbar, deren Deklarationen Referenzen zu dem ausgewählten beinhalten (Abbildung 5.4, Schaltfläche zum Moduswechsel im roten Kreis).

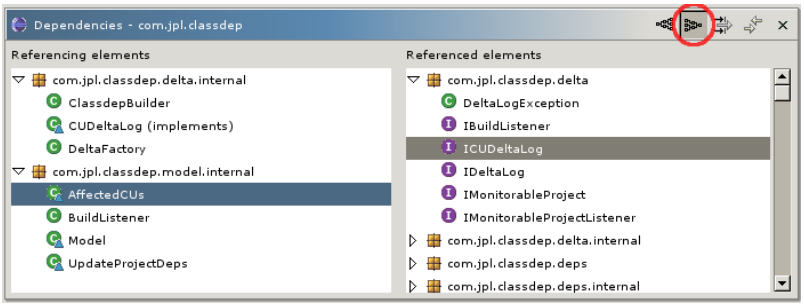

Abbildung 5.4: Umschalten des Anzeigemodus: Anzeige der referenzierenden Elemente

In Abbildung 5.4 ist noch ein weiteres Funktionsmerkmal von Classdep sichtbar: die Klassifizierung von Abhängigkeiten, die in der Anmerkung `implements` neben der Klasse `CUDeltaLog` auftaucht. Jede gefundene Referenz wird hinsichtlich der Stelle ihres Vorkommens und der Art beteiligter Elemente (Klassen oder Interfaces) untersucht. Dadurch wird die Abhängigkeit zu einer der folgenden Kategorien zugeordnet:

- Verwendung eines Interface
- Erweiterung eines Interface
- Implementierung eines Inteface
- Verwendung einer konkreten Klasse
- Objekterzeugung
- Vererbung einer konkreten Klasse

Mit den Kategorien sind Gewichte verbunden, die zur Kennzeichnung der Abhängigkeitsstärke verwendet werden, in der obigen Liste aufsteigend geordnet. In späteren Versionen des Werkzeugs könnte die Kategorienmenge noch ergänzt werden.

Selektiert man sowohl im linken als auch im rechten Anzeigebereich eine Klasse, so können mit Hilfe des Kontextmenüs diese Stellen im Quellcode gefunden werden, wo die der Abhängigkeit zu Grunde liegenden Referenzen vorkommen. Auf Wunsch wird von Classdep der Java-Editor geöffnet und die entsprechenden Zeilen hervorgehoben (Abbildung 5.5)

```
class AffectedCUs
{
    private Set removedCUs;
    private Set outdatedCUs;
    private Set updatedCUs;

    public AffectedCUs(ICUDeltaLog deltaLog, IUpdateableProjectDeps prevDeps)
        throws CoreException
    {
        Set addedCUs = new HashSet();
        removedCUs = new HashSet();
        outdatedCUs = new HashSet();
        updatedCUs = new HashSet();
```

Abbildung 5.5: Ausschnitt eines Java-Editors mit gefundenen Referenzen; vgl. Abbildung 5.4

5.1.2 Grafische Darstellung

Wenn es um die Visualisierung und das Verstehen von Graphen geht, arbeitet man gerne mit Darstellungen in Form von Diagrammen. Classdep erfüllt diesen Wunsch durch zwei Diagrammarten: das Paketdiagramm und das Klassendiagram. Die Diagramme unterscheiden sich voneinander (nur) durch die angezeigte Knotenart.

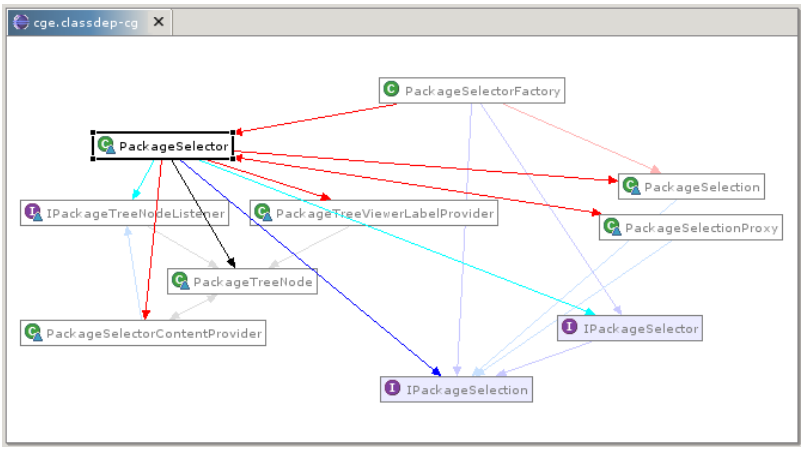

Abbildung 5.6: Ein mit Classdep erstelltes Klassendiagram

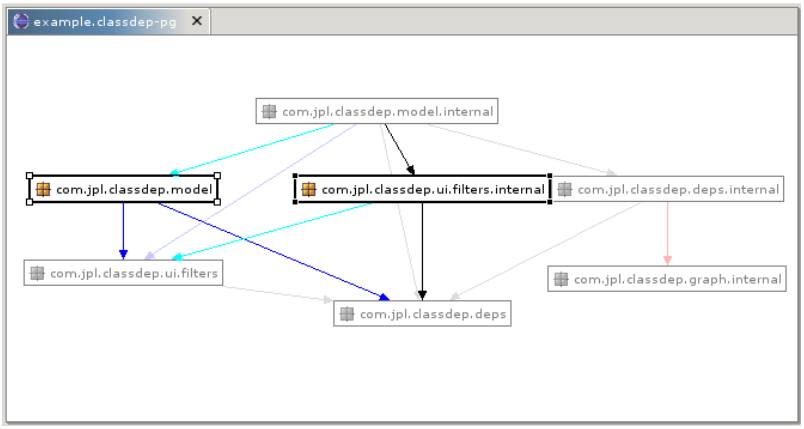

Abbildung 5.7: Ein mit Classdep erstelltes Paketdiagram

Beispiele beider Diagrammarten sind in Abbildungen 5.6 und 5.7 zu sehen. In beiden Fällen stehen die gerichteten Kanten für Abhängigkeiten und die Kantenfarbe kodiert die stärkste vorliegende Abhängigkeitsart:

- blau – Verwendung eines Interface
- violett – Erweiterung eines Interface
- cyan – Implementierung eines Interface
- grau – Referenz zu einer Klasse
- rot – Objekterzeugung
- grün – Vererbung

Die Knoten im Klassendiagramm werden mit den aus anderen Bereichen von Eclipse bekannten Symbolen versehen. Im Paketdiagramm wird für alle Knoten das gleiche Paketsymbol verwendet. Eine Knoten- und Kantenauswahl ist durch Anklicken möglich. Die ausgewählten Knoten können auf der unbeschränkten Diagrammfläche frei positioniert werden. Zusätzlich steht eine Menüaktion zur Verfügung, die ein hierarchisches Layout automatisch erzeugt. Diese Möglichkeit sollte zunächst zu einer groben Ausrichtung der Knoten genutzt werden. Eine nachträgliche manuelle Anpassung der Knotenpositionen ist empfehlenswert.

Die Kanten, die von oder zu den ausgewählten Knoten verlaufen, werden in einer intensiveren Farbe und im Vordergrund gezeichnet. So kann das Bild durch einfache Veränderungen der Selektion interaktiv untersucht werden.

Sind ein Diagrammeditor und die tabellarische Sicht beide aufgeblendet, so kann man die Elementauswahl in der Tabelle mit der Knoten- und Kantenauswahl im Editor synchronisieren lassen. Durch Nutzung dieser Funktion kann das Diagramm als Übersicht mit der Tabellendarstellung als Detailsicht verknüpft werden, um die Navigation durch Paket- und Klassenabhängigkeiten zu erleichtern.

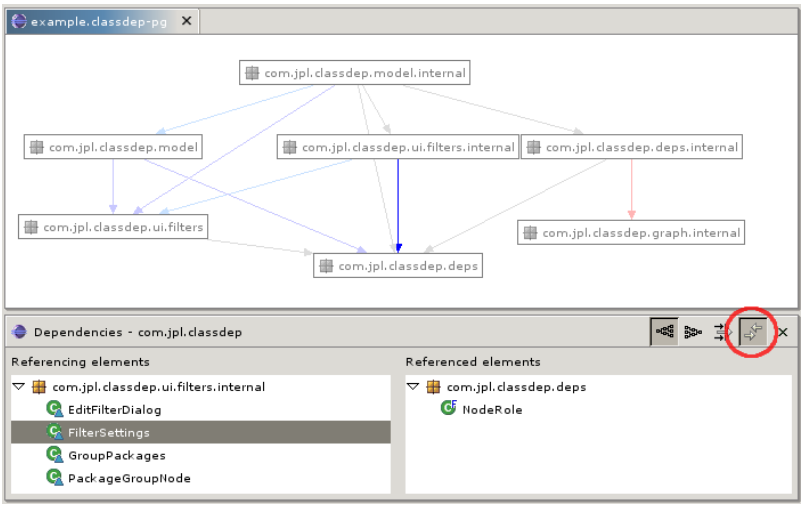

Abbildung 5.8: Synchronisierung der tabellarischen Darstellung mit dem Diagrammeditor (Verknüpfungsaktion im roten Kreis)

In Abbildung 5.8 kann man die Oberfläche nach dem Anklicken einer Kante (blau gezeichnet) sehen. In der Tabelle werden dementsprechend nur die referenzierten Klassen aus `com.jpl.classdep.deps` angezeigt. In der linken Hälfte erscheinen alle Klassen aus dem referenzierenden Paket, wobei diejenigen, die an der Abhängigkeit beteiligt sind, automatisch selektiert werden (im Beispiel gibt es nur eine solche Klasse). Befindet sich die Tabelle im anderen Modus (Anzeige von referenzierenden Elementen), so findet die automatische Filterung analogisch in der linken Hälfte und die Auswahl in der rechten Hälfte statt.

5.1.3 Vergleich mit UML-Diagrammen

In der Einleitung dieser Diplomarbeit wurde Classdep von UML-Werkzeugen abgegrenzt. Vergleicht man die vorher erläuterten Klassen- und Paketdiagramme mit der UML-Notation, so stellt man dennoch Ähnlichkeiten fest. Classdep nutzt in der Tat eine kleine Untermenge der von UML angebotenen Darstellungsmittel. Wenn man von visuellen Feinheiten absieht, unterscheidet sich Classdep von UML-Editoren hauptsächlich durch seinen spezifischen Verwendungszweck.

Die breite Palette von Modellelementen macht UML zu einer mächtigen und allgemeinen Sprache. Ein Klassendiagramm kann nach Fowler in Abhängigkeit von der gewählten Abstraktionsebene drei Rollen erfüllen: Konzepte im untersuchten Problembereich erfassen, Schnittstellen zwischen Typen spezifizieren oder auch Implementierungsdetails der Klassen zeigen. [25] Besonders wenn es um den Gedankenaustausch geht, sind die abstrakteren Sichtweisen zu bevorzugen. Es fällt dabei auf, dass diese Art der Klassendiagramme kaum automatisch aus dem Quellcode generiert werden kann.[1]

In Classdep wird die Unmöglichkeit der genauen Ermittlung der Quellcodebedeutung zum Anlaß genommen, Kanten im Klassendiagramm vereinfacht darzustellen und auf die Anzeige der Klasseninhalte komplett zu verzichten. Wenn man nicht leicht herausfinden kann, welche Aspekte der Klassen für den Benutzer interessant sind, so soll man sich lieber auf die Lieferung der minimalen, übersichtlichen Informationsmenge beschränken. Die zu Grunde liegende Idee ist, dass man einen Entwurf bereits dann beurteilen und verbessern kann, wenn man sich auf eine aus dem Quellcode leicht ermittelbare Beziehungsart konzentriert – die Abhängigkeiten. In UML-Klassendiagrammen kommen auf der Implementierungsebene üblicherweise noch viele sonstige Details vor. Kardinalitäten, lange Methoden- und Feldnamen, Beziehungsnamen, verschiedene Pfeilsymbole, benutzerdefinierte Stereotypen, Trennlinien und unterschiedliche

[1] Es sei denn, dass man den Code zuvor aus Diagrammen erzeugte und die semantischen Hinweise für die spätere Rekonstruktion irgendwo hinterlegte.

Schriftarten – sie alle dürften beim Überblicken von Klassenabhängigkeiten hinderlich wirken. Möchte man mehr über eine Abhängigkeit wissen, so liegt die endgültige Auskunft im Quellcode – dieser sollte auf jeden Fall leicht einsehbar sein.

Es gibt in UML keine entsprechende Diagrammart für Paketdiagramme von Classdep. Das Paketkonzept wird als allgemeines Mittel zur Gruppierung von Modellelementen eingeführt. Die UML-Spezifikation schreibt dabei keine konkreten Richtlinien für die Paketbildung vor, obwohl die Literatur die Nutzung von Paketen zur Reduzierung der wahrgenommenen Systemkomplexität und zur Untersuchung von Abhängigkeiten nahelegt. [25] Die Darstellung der Pakete erfolgt zusammen mit anderen Elementen innerhalb von UML-Klassendiagrammen. Die Beziehungen können zwischen Elementen über die Grenzen der umfassenden Pakete hinweg verlaufen oder aber auch ganze Pakete betreffen. Classdep bietet auch für Paketdiagramme eine stark vereinfachte Notation. Die Knoten werden im Sinne von Java-Paketen als Gruppen von Klassen interpretiert und bei der Darstellung nicht aufgegeliedert – die Kanten verbinden also immer ganze Pakete. Die Paketgruppen von Classdep entsprechen der in UML verfügbaren Darstellung von verschachtelten Paketen. Es wird allerdings auf die Visualisierung rekursiver Verschachtelungen verzichtet.

Man erkennt sofort, dass die Klassen- und Paketdiagramme im Classdep-Stil weniger Bildschirmplatz als die vergleichbaren UML-Diagramme in Anspruch nehmen. Der Grundgedanke bei der Entwicklung des Werkzeugs war, eine einfache Notation zur Verfügung zu stellen, die viel leichter interaktiv zu manipulieren als die herkömmlichen UML-Diagramme ist und weitere Informationen auf Wunsch des Benutzers vermittelt. Die Classdep-Diagramme sind vorwiegend kurzlebige, flexible Sichten auf den Ist-Zustand eines Entwurfs, die der Entwickler während einer Analysesitzung als Hilfsmittel zur Begutachtung der Softwarestruktur verwendet. Im Gegensatz dazu erfüllen die UML-Diagramme eher die Rolle von statischen Soll-Zeichnungen, die bestimmte Aspekte des Ent-

wurfs spezifizieren.

5.1.4 Filterung und Gruppierung

Die Anzahl von Paketen und Klassen, die in einem Java-Projekt mittlerer Größe mitwirken, ist sehr beträchtlich. Ohne einen Filterungsmechanismus müssten Diagramme eine entsprechend riesige Fläche beanspruchen und wären alleine auf Grund der Kantenanzahl vollkommen unübersichtlich (Abbildung 5.9).

Abbildung 5.9: Ausschnitt eines ungefilterten Paketdiagrams mit über 50 Knoten

Der Sinn der Visualisierung besteht nicht darin, abstrakte Kunst zu erzeugen, um nachher Bürowände mit ihr zu tapezieren. Das menschliche Gehirn arbeitet selektiv. Es ist bekannt, dass man im Kurzzeitgedächtnis gleichzeitig nur ungefähr sieben Elemente behalten kann, eine Voraussetzung dafür, ihre Zusammenhänge zu untersuchen. Gleichwohl haben Wissenschaftler festgestellt, dass es auf den Informationsgehalt der Elemente nicht ankommt: es ist sogar schwieriger, sich 7 zufällige Buchstaben zu merken als 7 sinnvolle Wörter, die jeweils aus mehreren Buchstaben bestehen. Daraus folgt für unsere Diagrammdarstellung, die ja von einem Menschen verstanden und bearbeitet werden soll, dass die Knotenanzahl unabhängig von der Projektgröße begrenzbar sein sollte.

Es gibt zwei Wege, die Knotenanzahl bei der Darstellung eines Graphen zu verkleinern. Man kann entweder den Graph in Teilgraphen zerteilen, die jeweils

getrennt voneinander analysiert werden, oder man kann die einzelnen Knoten zu übergeordneten Knoten zusammenfassen und damit einen neuen, vereinfachten Graph entstehen lassen. Beide Vorgehensweisen haben ihre Stärken und Schwächen. Sie können also in Classdep kombiniert werden: die im Paketdiagramm dargestellte Knotenmenge und die Elementmenge in der Tabelle ergibt sich aus einem Zusammenspiel von definierten Filtern und Paketgruppen.

Ein Paketfilter dient dazu, die Darstellung ausgewählter Pakete zu verhindern. Ein Paket (und damit auch die enthaltenen Klassen) kann auf Wunsch des Benutzers entweder immer versteckt bleiben, oder nur dann, wenn es sich in einer der beiden Rollen "referenzierend" oder "referenziert" befindet. Die benannten Filter werden auf der Projektebene durch Paketauswahl definiert und können von mehreren Diagrammen und der tabellarischen Darstellung gemeinsam benutzt werden (Abbildungen 5.10 und 5.11). Speziell für die tabellarische Darstellung können solche Abhängigkeiten von der Anzeige ausgeschlossen werden, die innerhalb eines Pakets verlaufen oder nur einen einzelnen Typ betreffen (Selbstreferenzen).

Ebenfalls als Teil des Filterobjekts können Paketgruppen definiert werden. Sie ermöglichen die Zusammenfassung mehrerer Pakete unter einem gemeinsamen Namen. Gehört ein Paket laut dem für ein Diagramm ausgewählten Filter zu einer Paketgruppe, so wird sein Knoten durch den Gruppenknoten ersetzt. Seine Abhängigkeiten werden der Paketgruppe zugeschrieben. In der tabellarischen Darstellung erscheinen die Paketgruppen als übergeordnete Knoten des Baums. Analog zur Auswahl eines Pakets, die eine automatische Auswahl aller enthaltenen Klassen bedeutet, gilt die Auswahl einer Paketgruppe als Auswahl aller gruppierten Pakete.

Mit Hilfe von Paketgruppen können Bibliotheken, die aus mehreren Paketen bestehen, zu einem einzelnen Knoten im Abhängigkeitsdiagramm zusammengefasst werden. Eine weitere Einsatzmöglichkeit besteht in der Gruppierung von Paketen, die zu bestimmten Kategorien gehören (z. B. Benutzeroberfläche,

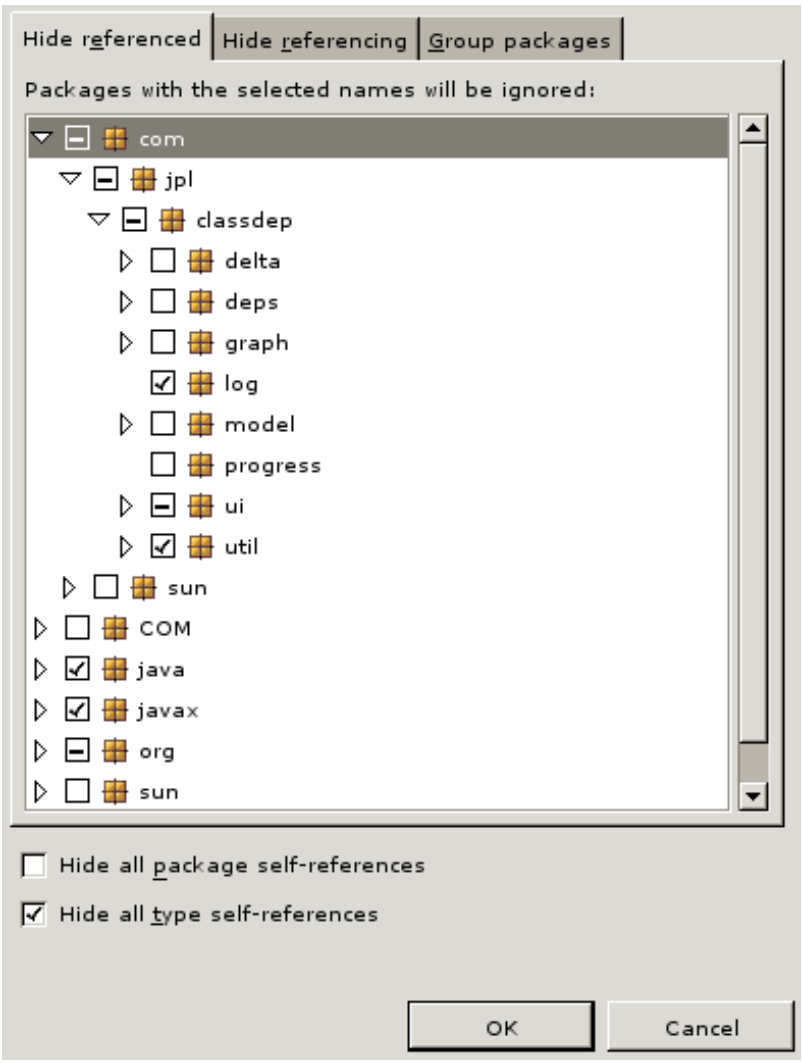

Abbildung 5.10: Maske zur Definition eines Filters

Programmlogik), um Übersichtsdiagramme zu erstellen. Schließlich können Paketgruppen zur Abgrenzung von Modulen verwendet werden. Dies gilt insbesondere dann, wenn man sich gemäß Empfehlungen aus dem vorherigen Kapitel für die Erstellung von zwei Paketen je Modul entscheidet. Um ein Diagramm zu bekommen, das alle Modulabhängigkeiten darstellt, muss man die jeweiligen Paketpaare in Classdep als Paketgruppen definieren. Wenn die Zugriffsmodifikatoren der einzelnen Implementierungsklassen auf “paketintern” gesetzt sind, dann ist das Moduldiagramm ebenso aussagekräftig wie ein Diagramm mit einzelnen Paketen, aber es wirkt dank der verringerten Knotenanzahl über-

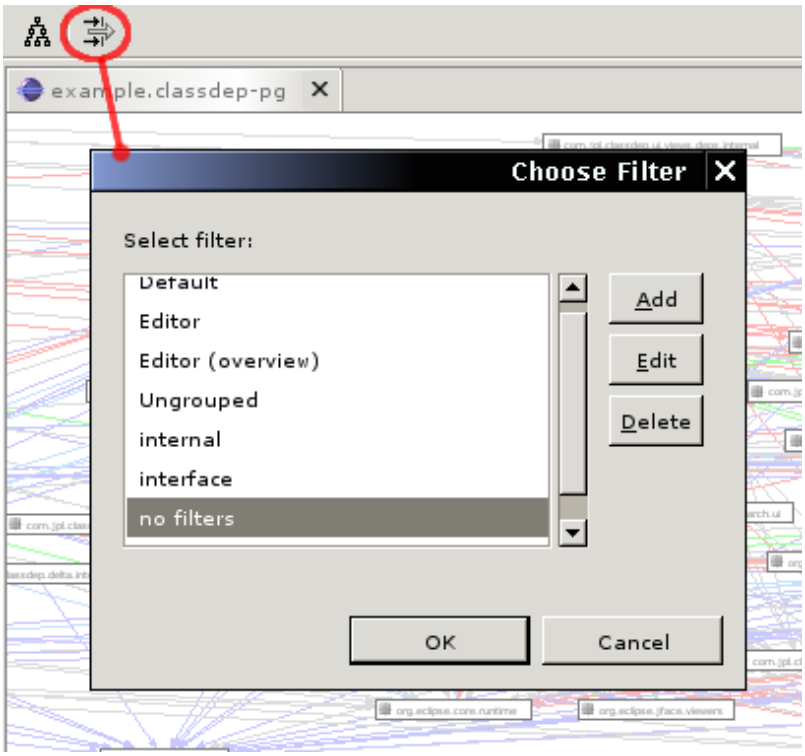

Abbildung 5.11: Auswahl eines Filters bei der Konfiguration des Diagrammeditors

sichtlicher.

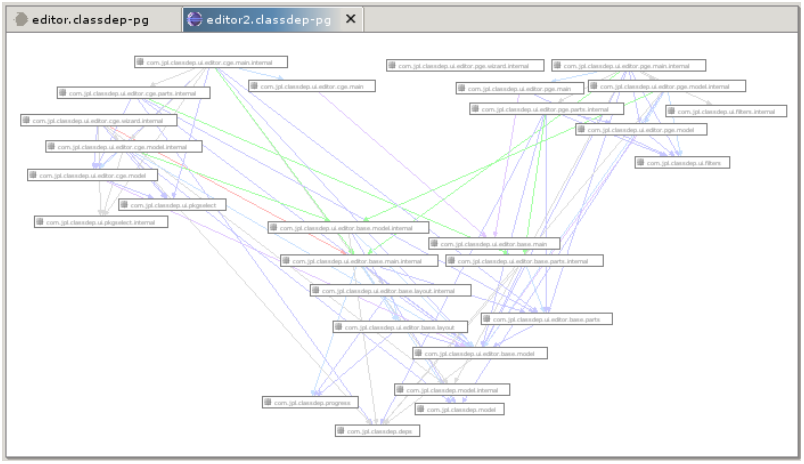

Abbildung 5.12: Diagramm ohne Einsatz von Paketgruppen

Abbildungen 5.12 und 5.13 zeigen, wie man den gleichen Sachverhalt (hier: den modularen Aufbau der Classdep-Diagrammeditore) durch die Anwendung von Paketgruppen besser visualisieren kann.

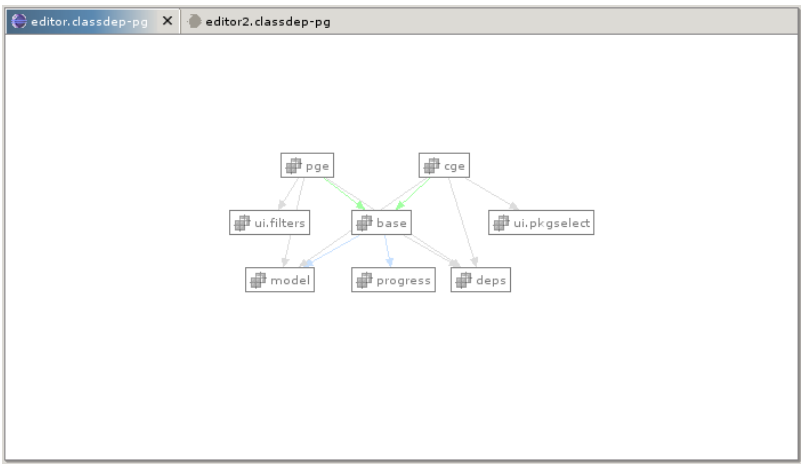

Abbildung 5.13: Diagramm mit Einsatz von Paketgruppen

Bei der Erstellung von Klassendiagrammen werden die beschriebenen Filter außer Acht gelassen – im Klassendiagramm erscheinen immer nur diese Klassen, die zum einen der vom Benutzer ausgewählten Pakete gehören (Abbildung 5.14). Die Auswahl gilt nur innerhalb des einzelnen Diagramms und wird im Unterschied zu Paketfiltern nicht zentral abgespeichert.

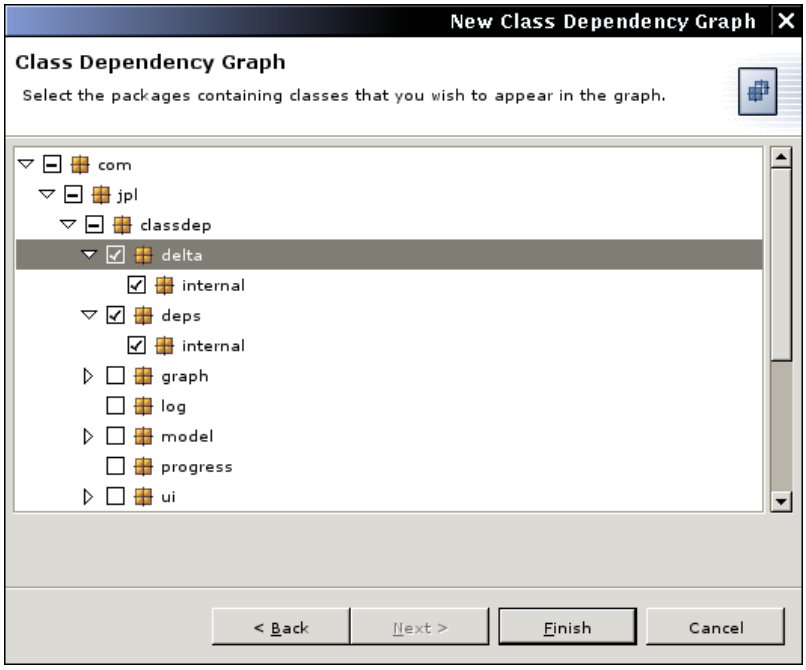

Abbildung 5.14: Paketauswahl bei der Erstellung eines Klassendiagramms

5.2 Refactoring

Hat man eine Abhängigkeit identifiziert und festgestellt, dass sie unerwünscht ist, so kann man sie durch geeignete Eingriffe in den Quellcode entfernen. Solche Veränderungen, die die Funktionsweise des Programms nicht beeinflußen, sondern lediglich die interne Struktur verändern, bezeichnet man als Refactoring (Umstrukturierung). [3]

Bei der Beseitigung von Abhängigkeiten werden die folgenden Refactoring-Aktionen besonders oft eingesetzt:

- Klasse (Oberklasse, Unterklasse) oder Interface extrahieren
- Paket extrahieren oder Pakete zusammenführen
- Assoziationsrichtung zwischen Klassen bzw. Klasse und Interface eändern

Classdep bietet keine direkte Unterstützung für Refactoring, sondern verlässt sich auf die in Eclipse bereits vorhandene Funktionalität. Eclipse aktualisiert bei der Verschiebung und Umbenennung von Klassen automatisch die enthaltenen Referenzen, was zur Veränderung der Abhängigkeitsgraphen führen kann. Die Änderungen werden von Classdep spätestens bei einer erneuten Analyse des Projekts aufgegriffen.

5.3 Zyklenauflösung

Im Kapitel 3.5 wurden die Nachteile von Zyklen in Abhängigkeitsgraphen erläutert. Classdep unterstützt den Benutzer bei der Identifizierung von Zyklen in Paketabhängigkeiten und bietet damit die erste Hilfe bei ihrer Beseitigung.

Abbildung 5.15 zeigt die von Classdep verwendete Darstellungsart für Zyklen. Für jeden gefundenen einfachen Zyklus wird in die Liste ein Baum eingetragen, der einen beliebigen Knoten des Zyklus als Wurzel hat. Es können, wie in der Abbildung zu sehen ist, auch mehrere Bäume mit dem gleichen

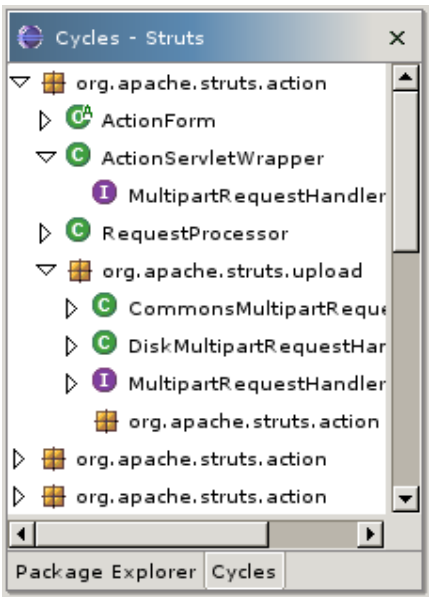

Abbildung 5.15: Darstellung von Zyklen im Paketabhängigkeitsgraphen

Knoten anfangen (hier: `org.apache.struts.action`). Klappt man einen Paketknoten auf, so werden als Unterknoten die referenzierenden Klassen aus dem Paket angezeigt (hier z. B. `ActionForm`). Als letzter Unterknoten erscheint das referenzierte Paket, also das nächste Glied des Zyklus. Wird ein Klassenknoten aufgeklappt, so werden die referenzierten Klassen sichtbar (die Klasse `ActionServletWrapper` referenziert `MultipartRequestHandler` aus dem Paket `org.apache.struts.upload`). Der letzte Unterknoten des Baums ist gleich dem Wurzelpaket – damit ist der Zyklus geschlossen.

Ähnlich wie in der tabellarischen Darstellung kann der Quellcode der angezeigten Klassen durch das Anklicken ihrer Knoten im Editor eingeblendet werden. Wählt man eine referenzierende Klasse, so werden zusätzlich die Referenzen zu allen referenzierten Klassen aus dem Folgeknoten des Zyklus im Quelltext hervorgehoben.

5.4 Einbindung in den Entwicklungsprozess

5.4.1 Inkrementelle Abhängigkeitsanalyse

Classdep wurde mit dem Gedanken entwickelt, die Abhängigkeitsanalyse zum festen Bestandteil eines Softwareentwicklungsprozesses zu machen. Obwohl das Werkzeug zur gelegentlichen Bearbeitung eines fertig geschriebenen Code genutzt werden kann, soll seine Stärke darin liegen, direkt bei der Entstehung der unerwünschten Abhängigkeiten zu helfen. Die Motiviation dafür ist, dass Abhängigkeiten, die auf der Verletzung von *information hiding* und anderen beschriebenen Prinzipien beruhen, sich schnell auf verschiedene Bereiche des Projekts ausbreiten und in späteren Phasen nur schwierig zu beseitigen sind. Es ist eine Binsenweisheit, dass ein Fehler im Entwurf viel mehr Korrekturaufwand als Implementierungsfehler erfordert.

Damit die Abhängigkeitsanalyse mit Classdep eine normale Softwareentwicklung begleiten kann, muss sie besonders effizient ablaufen. Der Entwickler darf nicht bei der Bearbeitung des Programmcode aufgehalten werden, jegliche Wartezeiten bei den einzelnen Arbeitsschritten sind negativ zu beurteilen, da sie zu Ablenkungen führen und damit sowohl die Produktivität erniedrigen als auch Frust bei Entwicklern verursachen. Daraus folgt, dass die Dauer der Abhängigkeitsanalyse nicht linear mit der Projektgröße steigen darf – eine Herausforderung an die Implementierung von Classdep.

Der konstante, niedrige Aufwand für die Abhängigkeitsanalyse wird in Classdep durch die Option der “Projektüberwachung” erreicht. Ist diese Option für ein Java-Projekt eingeschaltet, so wird der Abhängigkeitsgraph dauerhaft abgespeichert und alle Veränderungen von Quellcodedateien verbucht, um eine spätere Datenerfassung nur auf die notwendigen Teile des Projekts zu beschränken. Je öfter die Analyse vom Benutzer angefordert wird, desto kürzer dauert ihre Vorbereitung. Gleichzeitig ist anzumerken, dass die ständige Überwachung von Veränderungen sehr schnell abläuft und den Entwickler bei Quell-

codeveränderungen nicht verlangsamt. Die eigentliche Referenzensuche wird nach wie vor nur auf Wunsch durch das Aufblenden der Classdep-Masken und Diagrammeditoren angestoßen. Arbeitet man innerhalb der "Classdep-Perspektive", so wird der Vorgang bei jedem Abspeichern einer Java-Quelldatei gestartet – die Rechnerleistung entscheidet dann, ob die dadurch entstehenden Verzögerungen noch akzeptabel sind.

5.4.2 Überwachung der Paketabhängigkeiten

Wenn man die "Projektüberwachung" in Classdep eingeschaltet hat, werden bei jeder Quellcodeanalyse die gefundenen Abhängigkeiten mit diesen aus dem vorherigen Durchlauf verglichen. Die neu hinzugekommenen Paketabhängigkeiten so wie Paketabhängigkeiten, die nicht mehr existieren, werden dann als die Aufgabenanzeige von Eclipse eingetragen (Abbildung 5.16).

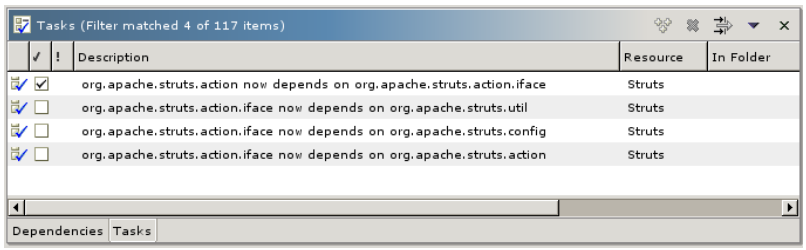

Abbildung 5.16: Benachrichtigung über Änderungen von Paketabhängigkeiten

Die einzelnen Positionen aus der Liste können vom Entwickler als "fertig" markiert und gelöscht werden. Der Sinn der Benachrichtigung besteht darin, die Aufmerksamkeit auf die Veränderungen im Abhängigkeitsgraphen zu lenken. Einerseits soll dadurch erreicht werden, dass neue Abhängigkeiten nur mit einer bewußten Zustimmung des Programmierers eintreten. Andererseits möchte man die für die Auswertung der Diagramme und Tabellen benötigte Zeit minimieren. Das heißt, dass man nicht bei jeder Sitzung mit Classdep "von Null' anfangen sollte. Beide Ziele werden durch die automatische Überwachung von Paketabhängigkeiten verfolgt. Der Entwickler kann von der Betrachtung der

bereits bekannten Sachverhalte absehen und sich auf die Überlegung und Verifizierung des neuen Zustands konzentrieren.

Kapitel 6

Fallstudie der Classdep-Verwendung

In diesem Kapitel wird der Einsatz von Classdep an einem ausgewählten Java-Projekt, Hibernate 2.0.1, veranschaulicht. Hibernate ist eine Klassenbibliothek zur Abbildung von Objekten auf Tabellen in relationalen Datenbanken – aus der Sicht eines Java-Programms also ein "Persistenzmodul", das zur dauerhaften Speicherung von Daten genutzt werden kann. [26]

Die beschriebenen Schritte wurden "live" aufgezeichnet. Sie spiegeln ein Experiment und einen Lernprozess wider, in dem Fehler vorkommen und erst nachträglich korrigiert werden. Man könnte nach der Beendigung der Fallstudie auf den Inhalt zurückblicken und die offenbaren "Sackgassen" und Irrtümer aus der Beschreibung ausschneiden. Dieser Versuchung wurde hier widerstanden. Damit wurde das Ziel verfolgt, nicht den kürzesten, sondern den tatsächlichen Weg zur Gewinnung der geschilderten Erkenntnisse wiederzugeben.

Der Quellcode von Hibernate umfasst mehr als 30 Pakete und 400 Java-Dateien. Es handelt sich dementsprechend um ein mittelgroßes Projekt, dessen detaillierte Beschreibung den Rahmen dieser Arbeit sprengen würde. Wir werden jedoch sehen, dass man Nutzen aus der Abhängigkeitsanalyse bereits dann ziehen kann, wenn man sich mit einem kleinen Ausschnitt eines unbekann-

ten Projekts befasst. Die Rolle dieser Beobachtung ist nicht zu unterschätzen, da in vielen Projekten detailliertes Wissen über alle Klassen und ihre Zusammenhänge nur bei wenigen Entwicklern vorhanden ist (wenn überhaupt). Wir verzichten vor der Durchführung unserer Analyse auf die Lektüre der umfangreichen Online-Dokumentation und lesen auch den Quellcode von Hibernate nicht als Vorbereitung durch.[1]

Wir werden unsere Studie nur auf wenige Pakete beschränken und versuchen, aus der Beobachtung von Abhängigkeiten auf der Klassenebene eine verbesserte Zuordnung der Klassen zu Paketen zu erschließen. Unser Ziel wird sein, die unerwünschten Paketabhängigkeiten entweder ganz zu eliminieren oder durch solche zu ersetzen, die die Programmstruktur verständlicher machen. Wir werden uns bei dieser Umstrukturierung vor allem auf das objektorientierte Prinzip der Umkehrung von Abhängigkeiten verlassen.

6.1 Projekteinrichtung und Übersicht

Nach dem Herunterladen von Hibernate werden alle Dateien aus dem gezippten Bündel als ein alleinstehendes Java-Projekt nach Eclipse importiert. Der Quellcode von Hibernate verwendet noch andere Bibliotheken, die in Binärform als JAR-Dateien mitgeliefert werden. Diese Bibliotheken muss man in den Projekt-Classpath einbinden, damit der Code compilierbar wird (und damit auch von Classdep analysierbar ist).

Nachdem das Projekt angelegt worden ist, können wir die Option "Projektüberwachung" (*Monitor project*) in Classdep einschalten. Wir wollen Veränderungen verfolgen, ohne den gesamten Quellcode jedes Mal neu verarbeiten zu lassen. Anschließend generieren wir ein Paketdiagramm, das zunächst mit den voreingestellten Filtern und keinen Paketgruppen erstellt wird. Wir stellen nach der automatischen Knotenausrichtung schnell fest, dass das Diagramm viel zu groß geworden ist (Abbildung 6.1).

[1] dies ist aber keine allgemeine Empfehlung!

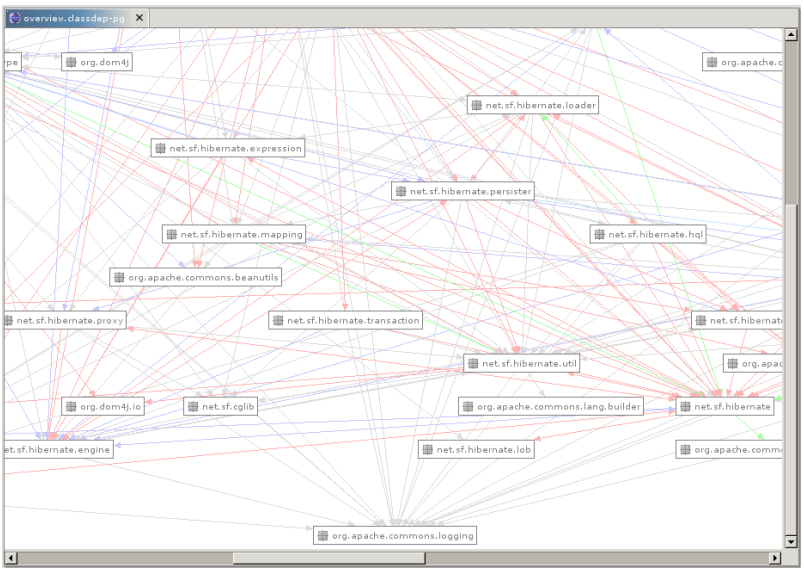

Abbildung 6.1: Ausschnitt des ersten Paketdiagramms für Hibernate

6.2 Einrichtung von Filtern

Ein üblicher Schritt, der zur Vereinfachung eines Paketdiagramms führt, ist die Beschränkung der dargestellten Knotenmenge auf die Pakete, die zum analysierten Projekt gehören. Classdep blendet als Voreinstellung bereits die Pakete aus `java.*` und `javax.*` aus. In unserem Fall wollen wir zusätzlich alle Pakete verstecken, deren Namen nicht mit dem Präfix `net.sf.hibernate` anfangen. Nach der Einrichtung des geeigneten Filters, einer erneuten Ausrichtung und einigen manuellen Knotenverschiebungen passt das Diagramm (verkleinert) auf einen Bildschirm (Abbildung 6.2). Es wäre noch viel erfreulicher, wenn wir es auf ungefähr 7 Knoten beschränken könnten. Wir wissen aber nicht genug, um jetzt eine sinnvolle Gruppierung oder eine weitere Filterung vorzunehmen.

6.3 Erster Eindruck und Bemerkungen zum Paketdiagramm

Das Paketdiagramm und Zyklenauflistung von Hibernate vermittelt uns, dass in diesem Projekt der Pflege von Paketabhängigkeiten keine besondere Bedeutung

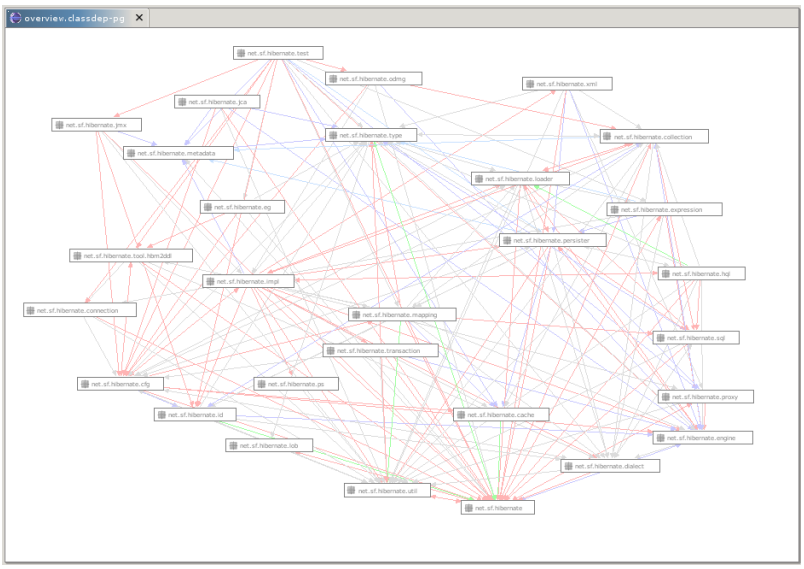

Abbildung 6.2: Gefiltertes Paketdiagramm für Hibernate

beigemessen wurde:

- im Diagramm ist keine klare hierarchische Struktur erkennbar (Pfeile sind sowohl nach unten als auch nach oben gerichtet),
- es gibt mehrere Pakete, die Ziel von vielen Abhängigkeiten sind (viele Konzentrationspunkte der Pfeilköpfe),
- Abhängigkeitszyklen kommen vor (im Diagramm nicht sichtbar, aber in der Zyklenanzeige).

Diese Aussagen treffen leider auf die meisten Java-Projekte zu. Es liegt daran, dass das Paketkonzept normalerweise nur zur losen Gruppierung von "verwandten" Klassen genutzt wird. In der einschlägigen Literatur wird die Rolle der Java-Pakete bei der Organisierung eines globalen Namensraums für Klassen hervorgehoben und die sonstigen Einsatzmöglichkeiten vernachlässigt. Deswegen legen die meisten Entwickler keinen besonderen Wert auf die Definition der Paketinhalte. Der Grad der "Klassenverwandschaft", der als Leitfaden bei der Zuordnung zu einem bestimmten Paket dient, ist nur schwer einzuschätzen. Es reicht daher in der Praxis, dass eine Klasse von ihrem Namen her den anderen

im gleichen Paket ähnlich ist, oder es werden Ordnungsvorschriften implementiert, die auf anderen leicht sichtbaren Merkmalen basieren. Beispiele:

- die Benutzeroberfläche gehört zum Paket `ui`
- alle Exceptions gehören zum Paket `error`
- alle Helper-Klassen (Klassen, die man gerne überall verwendet) gehören zum Paket `util`
- wenn man zwei Klassen oft zusammen importiert, dann sollten sie zum gleichen Paket gehören

Diese Heuristiken sind nicht schlecht (die letzte verfolgt implizit die Idee der Eingrenzung von Änderungen), aber sie reichen in der Regel zur Entstehung eines klar strukturierten Paketabhängigkeitsgraphen nicht aus. Die Situation wird durch eine natürliche Tendenz verschlimmert, Verschiebungen von Klassen zwischen Paketen zu vermeiden. Sie ergibt sich als Produkt mehrerer Faktoren. Die wichtigsten seien im Folgenden genannt:

- fehlende Automatisierung des Refactoring in der verwendeten Entwicklungsumgebung
- Sorgen, dass zu viele verschobenenen Dateien das Versionierungssystem (wie CVS) durcheinander bringen
- generelle Abneigung gegen Änderungen und Widerstand gegen Neuigkeiten
- fehlendes Wissen über den Inhalt und Sinn vorhandener Pakete
- Vorstellung, dass bisherige Benutzer die Klassen im neuen Paket nicht leicht finden werden
- veröffentlichte APIs, unveränderliche Dokumentation (wie z. B. Bücher)

6.4 Festlegung der Analyserichtung

Trotz aller im vorherigen Abschnitt aufgeführten Gründe sollte man die in vorherigen Kapiteln beschriebenen Prinzipien des objektorientierten Entwurfs und die Gefahren, die sich aus ihrer Verletzung ergeben, nicht aus den Augen verlieren. Wir möchten in weiteren Schritten die Einhaltung dieser Prinzipien in Hibernate an einigen Beispielen untersuchen und, falls möglich, die Paketinhalte so umordnen, dass die Abhängigkeiten entsprechend den Empfehlungen verlaufen.

Wir werden uns auf das Prinzip der Umkehrung von Abhängigkeiten auf Paketebene konzentrieren: sehr abstrakte Pakete, die Grundkonzepte von Hibernate umfassen, sollten sich auf der untersten Hierarchiestufe befinden. Da eine getrennte Einschätzung der Abstraktheit jedes Pakets (etwa durch Analyse des Vokabulars) einen großen Aufwand bedeuten würde, wollen wir das Problem von der anderen Seite mit Classdep angreifen: wir betrachten ein Paket, von dem viele andere abhängen, und überprüfen, ob es sich tatsächlich um ein abstraktes Gebilde handelt. Falls ja, dann bemühen wir uns darum, die Abhängigkeiten dieses Pakets von konkreteren Paketen zu beseitigen. Zusätzlich möchten wir versuchen, das Paket als Modul mit einer öffentlichen Schnittstelle und einer dahinter versteckten Implementierung zu betrachten, um dem *information hiding*-Prinzip gerecht zu werden.

6.5 Analyse des Inhalts von `net.sf.hibernate`

Es gibt gute Gründe zu vermuten, dass das Paket `net.sf.hibernate` sehr abstrakt ist – oder zumindest sein sollte. Erstens gibt es im Diagramm (Abbildung 6.2) viele Abhängigkeiten, die auf dieses Paket gerichtet sind. Zweitens ist in seinem Namen im Unterschied zu allen anderen Paketen kein Suffix enthalten. Handelt es sich tatsächlich um eine Sammlung von grundlegenden Abstraktionen des Hibernate-Projekts? Wir können mit der tabellarischen Darstellung

unsere Hypothese untersuchen – deuten die Namen der enthaltenen Typen auf ihre Abstraktheit hin? (Abbildung 6.3).

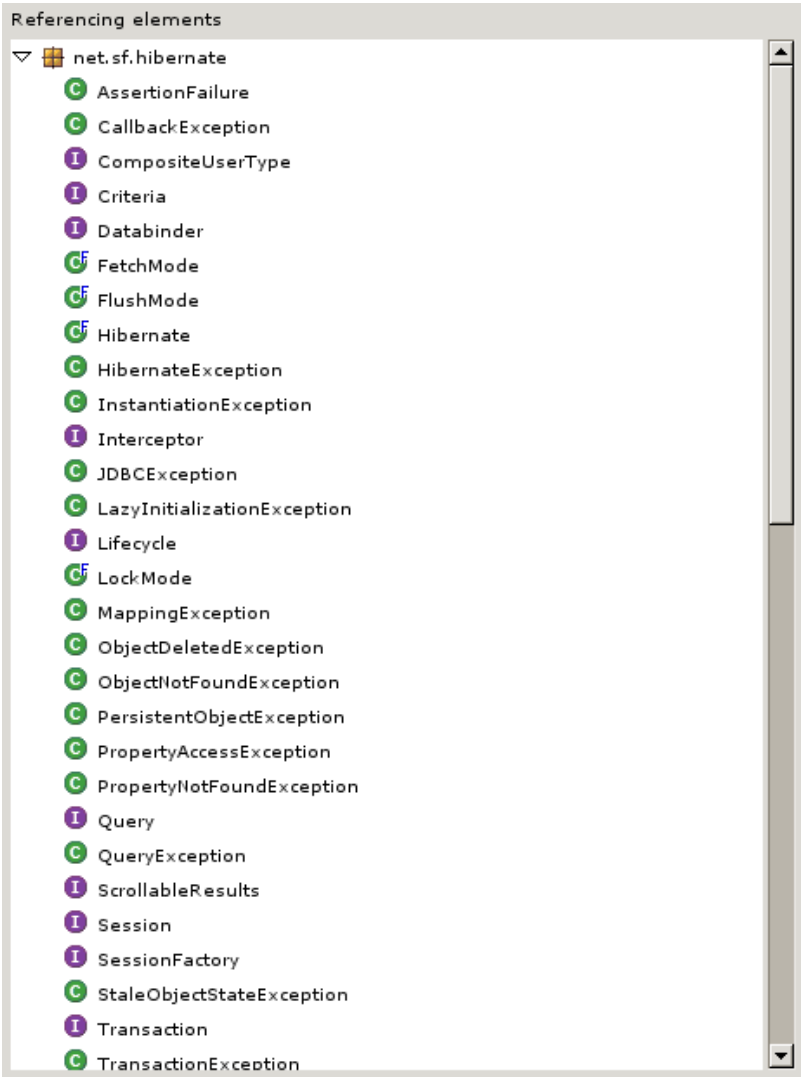

Abbildung 6.3: Namen einiger Typen aus dem Paket `net.sf.hibernate`

Es ist erkennbar, dass der Inhalt von `net.sf.hibernate` aus drei Kategorien von Elementen besteht: Exceptions, Interfaces (die tatsächlich einfache, allgemeine Namen tragen) und anderen Klassen (wie z. B. `FlushMode`). Die große Anhäufung von Exception-Klassen ist verdächtig: inwiefern hängen diese Klassen mit den in den Interfaces verkörperten Konzepten zusammen? Ein Klassendiagramm hilft, die Beziehungen innerhalb des betrachteten Pakets zu beleuchten (Abbildung 6.4).

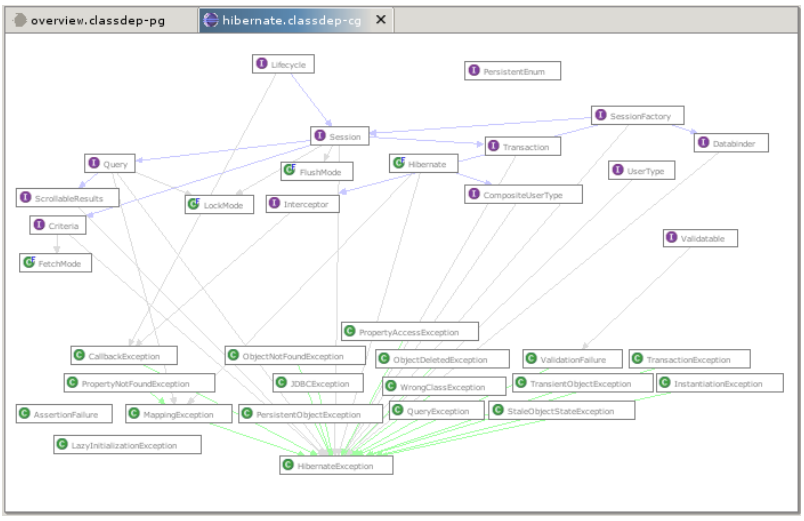

Abbildung 6.4: Klassenbeziehungen im Paket `net.sf.hibernate`

Im Klassendiagramm von `net.sf.hibernate` fällt auf, dass die meisten dort definierten Exceptions von keinen anderen Klassen in dem Paket referenziert werden. Da sie keine RuntimeExceptions sind, deren Aufführung in Methodensignatueren optional wäre, deutet die Abwesenheit von Referenzen darauf hin, dass sie kein Bestandteil der Schnittstelle sind. Zugleich merken wir, dass die meisten Klassen und Interfaces Verweise auf die Klasse `HibernateException` enthalten. Diese Klasse bildet offenbar eine Basis für alle Exceptions des Projekts (es lässt sich leicht durch eine Suche bestätigen). Sie ist also sicher ganz grundlegend und befindet sich an der richtigen Stelle, wenn `net.sf.hibernate` unserer Vorstellung der Abstraktheit entsprechen soll.

Für die anderen Exceptions gilt dies jedoch nicht. Es handelt sich bei ihnen um sehr konkrete Klassen, von denen jede nur in bestimmten Hibernate-Paketen genutzt wird – die referenzierenden Klassen lassen sich mit der tabellarischen Darstellung von Classdep jeweils leicht ermitteln. Wir kommen zum Schluß, dass die Deklaration dieser spezifischen Exceptions in `net.sf.hibernate` unerwünscht ist. Die beste Lösung wäre, sie denjenigen Paketen zuzuordnen, in welchen sie gemeldet werden. Eine Voraussetzung wäre, dass es bis auf `HibernateException` keine Ausnahmen gibt, die von mehreren Paketen referenziert werden. Wir wollen diese Hyptohese aber nicht gleich überprüfen,

sondern bei der Betrachtung von `net.sf.hibernate` bleiben. Der einfachste Weg, damit fertig zu werden, besteht darin, alle "unpassenden" Exception-Klassen in ein neues Paket zu verlegen. Dieses neue Paket kann nachher getrennt analysiert und dann möglicherweise aufgelöst werden – ein guter Punkt für unsere TODO-Liste.

6.6 Untersuchung von Abhängigkeiten

In folgenden beiden Abschnitten richten wir unseren Blick auf die Abhängigkeiten, die von und zu dem Paket `net.sf.hibernate` verlaufen. Wir werden uns bemühen, für jede Abhängigkeit die Ursache zu ermitteln und sie entweder zu rechtfertigen oder abzuschaffen.

6.6.1 Aufwachen aus dem Winterschlaf

Wir haben bereits den ersten Schritt gemacht, um das Paket `net.sf.hibernate` abstrakter zu machen – die konkreten abgeleiteten Klassen von `HibernateException` wurden aus dem Paket entfernt. Damit sind die Abhängigkeiten, die von ihnen verursacht wurden, ebenfalls aus dem Kontext von `net.sf.hibernate` verschwunden. Ist aber das Paket dadurch richtig unabhängig geworden? Ein Paketdiagramm veranschaulicht die noch vorhandenen Abhängigkeiten (Abbildung 6.5). Durch den Einsatz eines Filters für referenzierende Pakete wurden die irrelevanten Knoten ausgeblendet.

Wir sehen, dass es immer noch eine beträchtliche Reihe von ausgehenden Abhängigkeiten gibt. Es gibt hierfür zwei mögliche Begründungen: entweder ist der Paketinhalt nicht ganz so abstrakt, wie wir es ursprünglich vermutet haben – dies würde bedeuten, dass die in Abbildung 6.5 dargestellten Pakete noch abstrakter als `net.sf.hibernate` sind – oder wir haben nach wie vor mit Klassen zu tun, die besser woanders untergebracht wären.

Um zwischen den beiden Fällen zu unterscheiden, verwenden wir die ta-

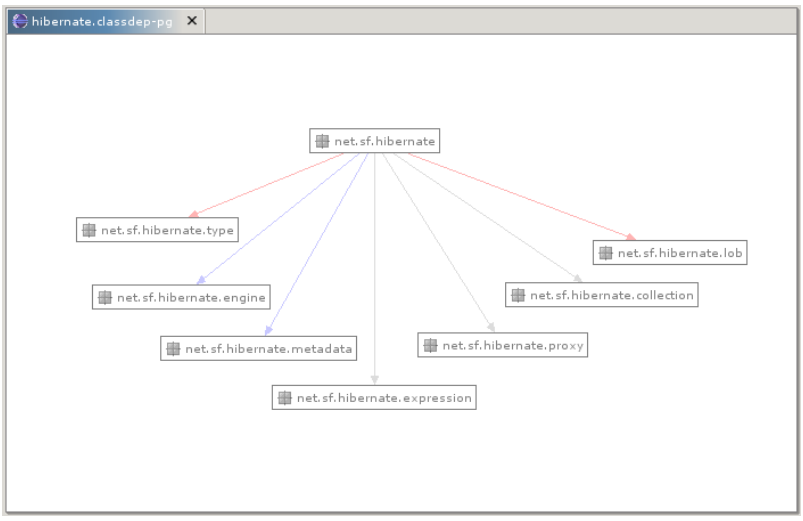

Abbildung 6.5: Abhängigkeiten des Pakets `net.sf.hibernate` nach der Auslagerung der Exception-Klassen

bellarische Darstellung zur Anzeige von Quellklassen für Abhängigkeiten von `net.sf.hibernate`. Gehen die meisten Abhängigkeiten nur von bestimmten Klassen aus, oder sind sie gleichmäßig über den gesamten Paketinhalt verteilt?

Wir stellen dadurch fest, dass es einige Klassen und Interfaces gibt, die von `net.sf.hibernate.type` abhängen. Es wird in diesen Fällen jeweils nur das Interface `net.sf.hibernate.type.Type` referenziert. Vielleicht verkörpert diese Schnittstelle ein Grundkonzept von Hibernate? Wir sollten besorgt sein, da das `type`-Paket offenbar auch **Rückabhängigkeiten** von `net.sf.hibernate` aufweist. Dieses Problem lassen wir jedoch zunächst bei Seite[2] und konzentrieren uns auf ein deutlicheres: die konkrete Klasse `Hibernate` scheint eine riesige Quelle von Abhängigkeiten zu sein, die fast alle zu sehr konkreten Klassen verlaufen (Abbildung 6.6).

Können wir die Klasse `Hibernate` aus dem Paket entfernen, so wie wir es früher mit den Exceptions gemacht haben? Es würde dafür sprechen, wenn wir für diese Klasse ein sinnvolles neues Zuhause finden könnten. Wird sie vielleicht nur von einem bestimmten anderen Paket verwendet, wo sie auch hingehört? Das Leben wäre zu einfach – und wir haben mit einem realen Stück Code zu

[2] es ist schon das zweite verzögerte Problem!

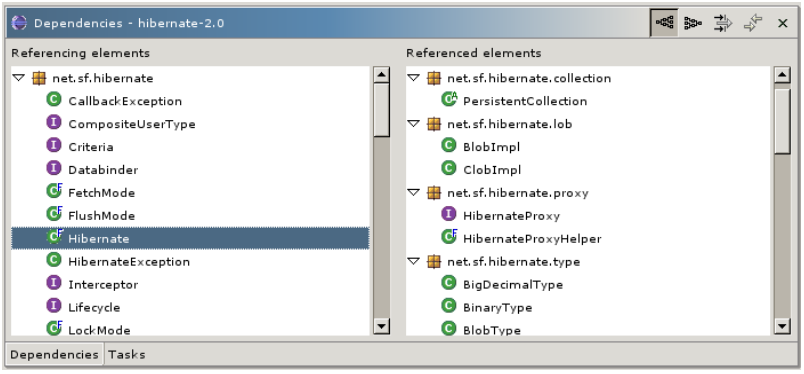

Abbildung 6.6: Abhängigkeiten von `net.sf.hibernate.Hibernate`

tun, der ohne Hilfe von Classdep geschrieben wurde! Die Klasse `Hibernate` scheint leider von überall her referenziert zu werden (Abbildung 6.7).

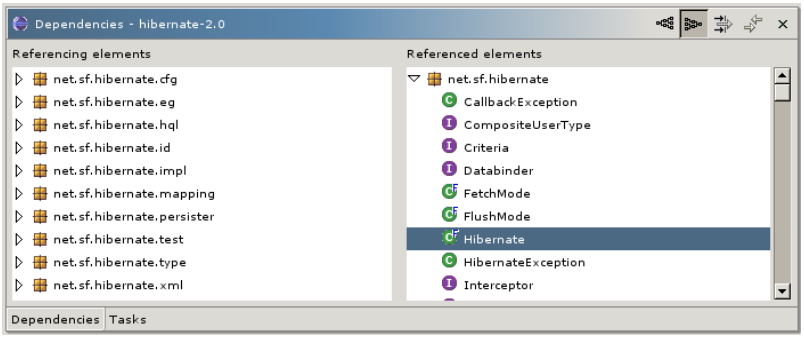

Abbildung 6.7: Abhängigkeiten zu `net.sf.hibernate.Hibernate`

Wir sind bis zu dieser Stelle alleine mit Diagrammen gut ausgekommen – jetzt muss aber zur Lösung des Rätsels der Quellcode von `Hibernate` herangezogen werden. Was ist an dieser Klasse, dass sie zugleich zu einer Referenzenquelle macht und andererseits unzählige Abhängigkeiten von außen heranzieht?

Die Javadoc-Beschreibung der Klasse lautet:

Provides access to the full range of Hibernate built-in types. `Type` *instances may be used to bind values to query parameters. A factory for new* `Blob`*s and* `Clob`*s. Defines static methods for manipulation of proxies.*

Die Klasse ist ein Behälter, aus dem globale Konstanten herausgezogen werden. Sie enthält offenbar ein `public static final`-Feld für jede konkrete Implementierung von `Type`. Im gleichen "Sack" finden wir noch drei statische

```
public static void initialize(Object proxy)
    throws HibernateException
{
    if (proxy == null) return;
    else if (proxy instanceof HibernateProxy)
        HibernateProxyHelper.getLazyInitializer(
            (HibernateProxy) proxy).initialize();
    else if (proxy instanceof PersistentCollection)
        ((PersistentCollection) proxy).forceLoad();
}
```

Abbildung 6.8: Verzicht auf Polymorphie – oder fehlende Kohäsion?

Methoden, die offenbar wenig mit den Typkonstanten zu tun haben und ihre Arbeit an die Klasse `net.sf.hibernate.HibernateProxyHelper` delegieren. Sowohl Signaturen als auch Definitionen dieser Methoden sehen für ein geübtes Auge ziemlich unangenehm aus (Abbildung 6.8).

Jetzt ist ein guter Zeitpunkt, über den Nutzen der statischen Quellcodeanalyse nachzudenken. Wir haben durch die reine Betrachtung der Abhängigkeitsdiagramme eine Stelle im Quellcode gefunden, wo offenbar noch einige grundsätzliche Stilverbesserungen erforderlich sind. Dies ist kein Zufall und nicht außergewöhnlich. Die Analyse zwingt zum Nachdenken und Hinterfragen des Code auch lange nach seiner Entstehung. Sie zeigt Symptome, die auf Ursachen zurückgeführt werden können.

Wir wollen uns aber nicht von unserem Ziel ablenken lassen, das Paket `net.sf.hibernate` in Ordnung zu bringen. Deswegen schlagen wir wieder den schnellsten Weg ein und verlagern die Klasse `Hibernate` dorthin, wo sie auf Grund ihres Inhalts besser hineinpasst. Da es sich im Wesentlichen um eine Sammlung von `Type`-Instanzen und Methoden zu ihrer Verarbeitung handelt, wird die Klasse nach `net.sf.hibernate.type` verbannt. Das Problem mit den statischen Methoden für “proxies” verschwindet dadurch nicht aus dem Projekt, aber sehr wohl aus unserem Blickfeld. Noch eine letzte Bemerkung: Entwurfsmacken kommen selten vereinzelt vor. Wenn wir über den Namen der

Klasse `Hibernate` nachdenken, so fällt auf, dass er zu Ihrem Inhalt nicht passt – die Namen `Types` oder `TypeConstants` wären schon angemessener.

6.6.2 Eingehende Abhängigkeiten

Die Abbildung 6.9 stellt die nach der Entfernung der Klasse `Hibernate` verbleibenden Abhängigkeiten des Pakets `net.sf.hibernate` dar. Es fällt auf, dass im Vergleich zur Abbildung 6.5 einige Abhängigkeiten verschwunden sind und andere ihren Charakter auf einen schwächeren geändert haben (Änderung der Kantenfarbe).

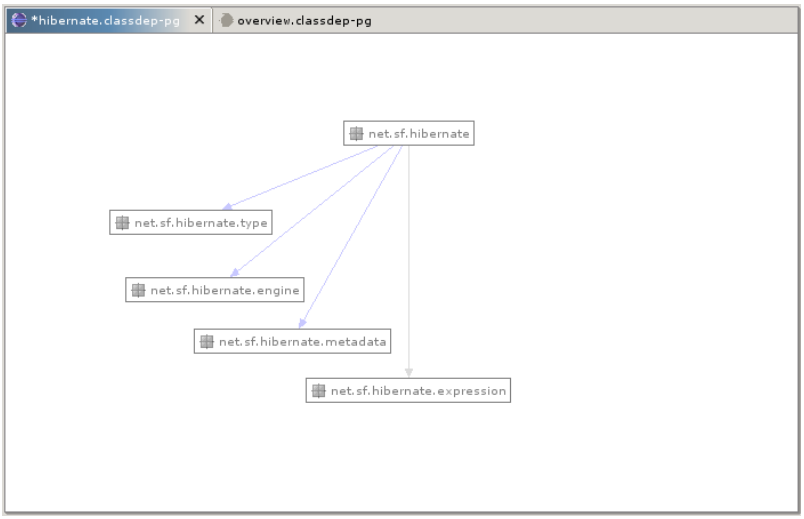

Abbildung 6.9: Abhängigkeiten vom Paket `net.sf.hibernate` nach der Auslagerung von `Hibernate`

Für die einzelnen Abhängigkeiten schauen wir uns nun die Quellen und Ziel an. Wir sehen, dass die Abhängigkeiten bis auf die zu `net.sf.hibernate.expression` zwischen Interfaces verlaufen. Es ist ein Hinweis darauf, dass das Paket `net.sf.hibernate` nicht so abstrakt ist, wie wir ursprünglich angenommen haben. Offenbar baut sein Inhalt auf Konzepten (Interfaces) aus anderen Paketen auf.

Bisher haben wir uns darum bemüht, die Abhängigkeiten von anderen Paketen abzuschaffen. Die obige Beobachtung lässt uns nun in Frage stellen, ob

das Paket von so vielen anderen referenziert werden sollte: wo liegt die Ursache dieser Abhängigkeiten?

Es ist leicht herauszufinden, dass sehr viele von ihnen durch die Basisklasse `HibernateException` und eine davon abgeleitete `MappingException` verursacht werden. Wir haben diese Klassen zuvor nicht mit anderen Exceptions zum neuen Paket `net.sf.hibernate.error` verlagert, da sie uns als "abstrakt genug" für ihren aktuellen Standort vorgekommen sind. Jetzt sieht es aber danach aus, dass sie für das Paket `net.sf.hibernate` **zu abstrakt** sind. Würde ihre Verschiebung nach `error` das Problem der bidirektionalen Abhängigkeiten zwischen `net.sf.hibernate` und den in Abbildung 6.9 aufgeführten Paketen lösen? Wir können diese Hypothese einfach ausprobieren – dank der in Eclipse eingebauten Unterstützung von Refactoring erledigt die IDE die ganze mühselige Arbeit für uns.

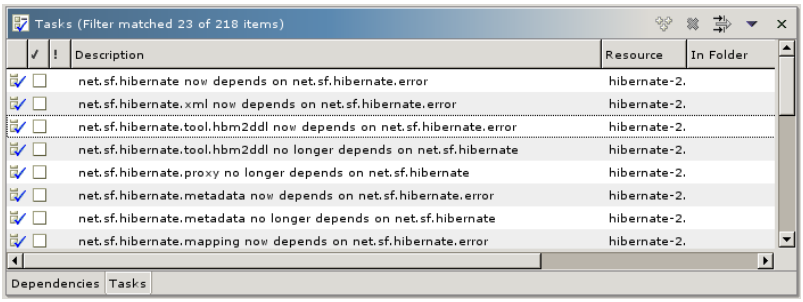

Abbildung 6.10: Veränderungen in Abhängigkeiten nach der Auslagerung von `HibernateException`

Abbildung 6.10 bestätigt unsere Vermutung, dass viele Abhängigkeiten, die zu `net.sf.hibernate` führten, nichts mit den Inhalten des Pakets zu tun hatten – sie wurden nämlich mit der Klasse `HibernateException` nach `net.sf.hibernate.error` mitgezogen.

Wir haben uns eine längere Zeit das Klassendiagram von `net.sf.hibernate` nicht mehr angesehen. Es eignet sich gut zur Verifizierung des Paketinhalts nach erfolgten Veränderungen (Abbildung 6.4).

Es fällt in diesem Diagramm auf, dass es einige Klassen und Interfaces gibt, die von den anderen offenbar nicht referenziert werden. Nach einer genaueren

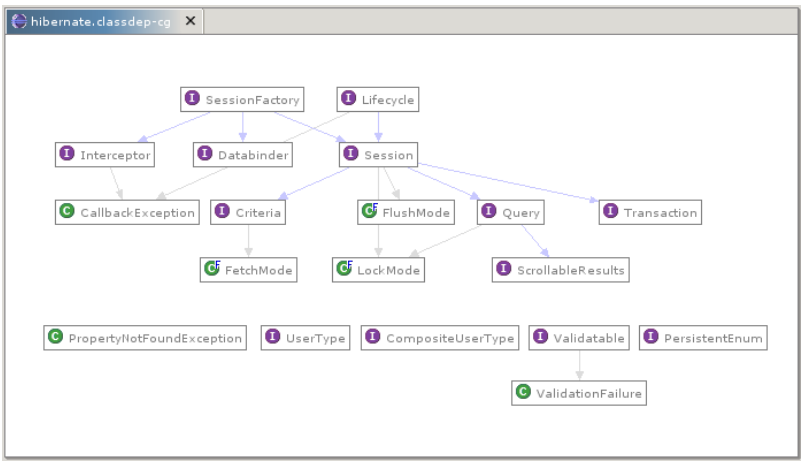

Abbildung 6.11: Neues Klassendiagram des Pakets `net.sf.hibernate`

Betrachtung der Abhängigkeiten und Namen dieser Typen kommen wir zu folgenden Ergebnissen:

- `PropertyNotFoundException` wird nach `net.sf.hibernate.error` verschoben. Diese Klasse ist von `MappingException` abgeleitet, die früher bereits zu dem neu angelegten `error`-Paket verlegt wurde
- `UserType` und `CompositeUserType` finden ihr neues Zuhause in `net.sf.hibernate.type`. Der erste Beweggrund für die Entscheidung war der Suffix "Type" in Namen der Klassen. Zweitens gab es Referenzen, die zu einer dieser Klassen aus dem Paket `type` verwiesen.
- `PersistentEnum` wird ebenfalls nach `net.sf.hibernate.type` verschoben. Das ist das einzige Paket, woher dieses Interface referenziert wurde.
- `Validatable` und `ValidationFailure` bleiben (vorläufig?), wo sie sind. Dafür spricht die Tatsache, dass die erste Schnittstelle durch eine Klasse aus `net.sf.hibernate.impl` implementiert wird.

Die resultierenden Abhängigkeiten von `net.sf.hibernate` entsprechen unseren Vorstellungen, bis auf die immer noch vorhandene Abhängigkeit, die vom Paket `net.sf.hibernate.type` verläuft, welches wir nun für abstrakter als

`net.sf.hibernate` halten. In Abbildung 6.12 wurden zur Erhöhung der Anschaulichkeit die irrelevanten Pakete zu einer Gruppe zusammengefasst.

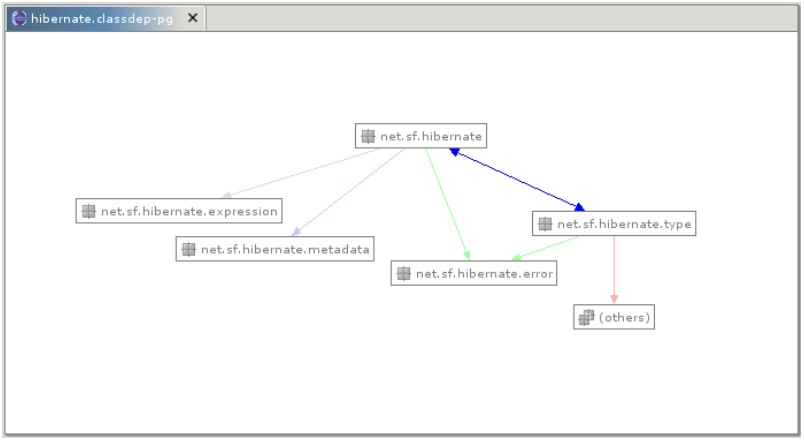

Abbildung 6.12: Paketabhängigkeiten von `net.sf.hibernate` nach der Entfernung unreferenzierter Klassen

Die Abhängigkeit des `type`-Pakets von `net.sf.hibernate` scheint auf den ersten Blick schwach zu sein: die konkrete Klasse `TypeFactory` verweist dort auf das Interface `Lifecycle`. Im Quellcode kommt die Referenz an nur einer einzigen Stelle vor (Abbildung 6.13). Dennoch handelt es sich um einen schwierigen Fall. Die Verschiebung von `Lifecycle` zum referenzierenden Paket kommt nicht in Frage, da diese Schnittstelle noch andere mitziehen müsste (vgl. Abbildung 6.11). Wie können wir die unerwünschte Abhängigkeit sonst loswerden?

Die Lösung besteht darin, das referenzierende Paket `net.sf.hibernate.type` in zwei Pakete zu zerteilen: ein Schnittstellenpaket und ein Implementierungspaket. Im ersten werden Abstraktionen zusammengefasst (alle Interfaces), im letzteren alle konkreten Implementierungen von `Type` zusammen mit der problematischen `TypeFactory`-Klasse, die Typinstanzen erzeugt und die Referenz zum Interface `Lifecycle` enthält. Nach dieser Operation sehen die Paketabhängigkeiten wie in Abbildung 6.14 aus.

Wir merken, dass die "unerwünschte" Abhängigkeit des abstrakten `type`-Pakets von `net.sf.hibernate` nicht mehr existiert – sie wurde durch die

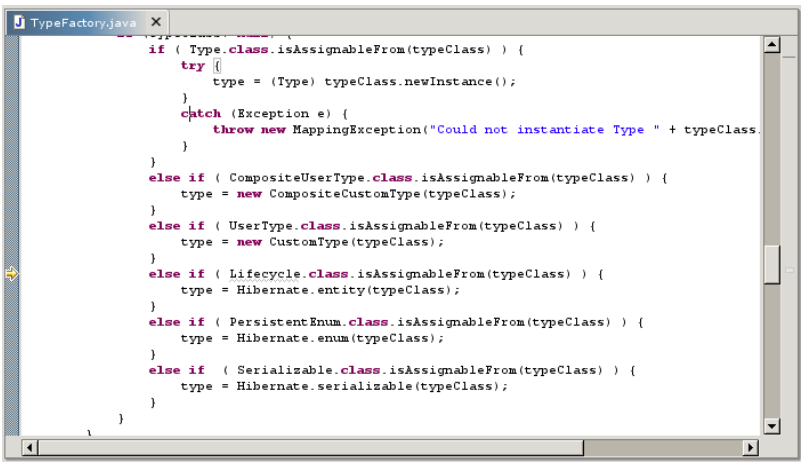

Abbildung 6.13: Referenz zu `net.sf.hibernate.Lifecycle` innerhalb von `net.sf.hibernate.type.TypeFactory`

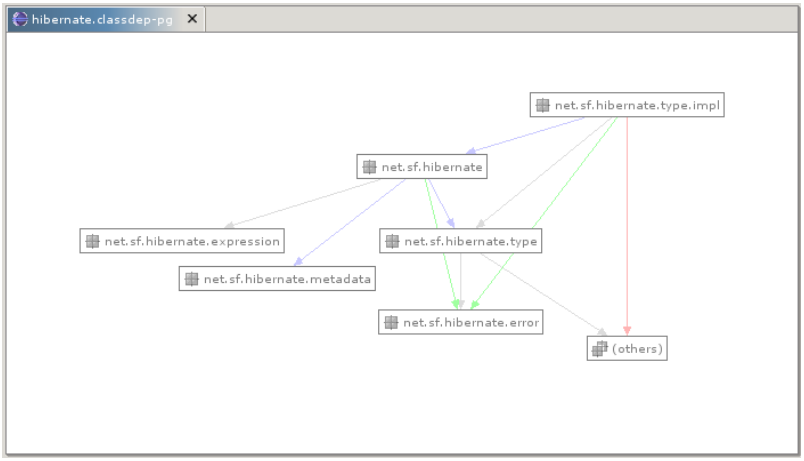

Abbildung 6.14: Paketabhängigkeiten nach der Zerteilung des Pakets `net.sf.hibernate.type`

Abhängigkeit des neuen `net.sf.hibernate.type.impl`-Pakets ersetzt. Diese Abhängigkeit, obwohl sie natürlich die gleiche Quell- und Zielklasse wie die frühere hat, ist nun akzeptabel, wenn wir annehmen, dass das neu angelegte `type.impl`-Paket viel konkreter als das abstrakte `net.sf.hibernate` ist, welches wiederum konkreter als das Schnittstellenpaket `net.sf.hibernate.type` einzuschätzen ist. Damit können wir unsere Bearbeitung der Abhängigkeiten von `net.sf.hibernate` beenden.

6.7 Modularisierung

Alle bisherigen Überlegungen in der Fallstudie konzentrierten sich auf die Beziehungen und Beurteilung der Abstraktheit der **Pakete** – das Wort "Modul" wurde dabei kein einziges Mal verwendet. Inwiefern könnte man das Paket `net.sf.hibernate` in seiner Endform nach den durchgeführten Änderungen als Modul betrachten?

Es ist leicht am Inhalt von `net.sf.hibernate` zu erkennen (Abbildung 6.15), dass es sich um ein abstraktes Schnittstellenpaket handelt – es könnte wohl der öffentliche Teil eines Moduls werden, es fehlt aber noch die Implementierung.

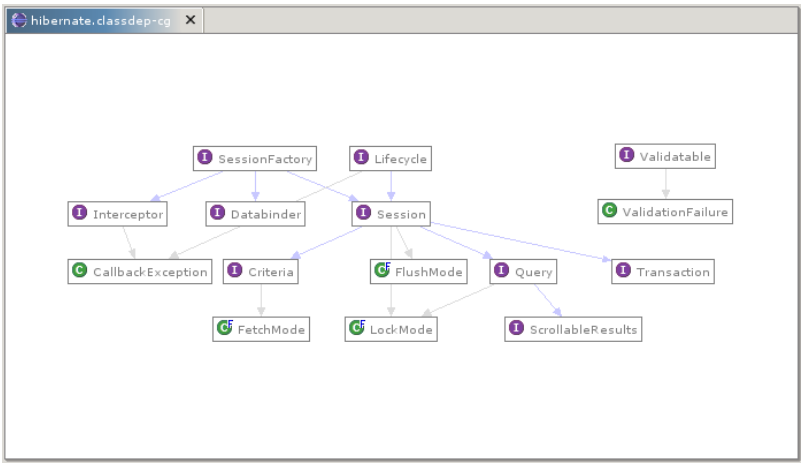

Abbildung 6.15: Endgültiger Inhalt von `net.sf.hibernate` nach Veränderungen

Wir haben diesmal Glück, da die entsprechenden Implementierungsklassen bereits von Hibernate-Entwicklern bereits in einem weiter Paket zusammengefasst wurden: `net.sf.hibernate.impl`. Die Abhängigkeiten zwischen den beiden Paketen lassen sich in einem Klassendiagramm veranschaulichen (Abbildung 6.16).

Die Trennung eines Moduls in zwei Pakete entspricht bereits den Richtlinien aus dem vierten Kapitel (Modulbildung für Classdep). Es wäre noch zu prüfen, ob die Sichtbarkeit der Klassen im Implementierungspaket nach außen

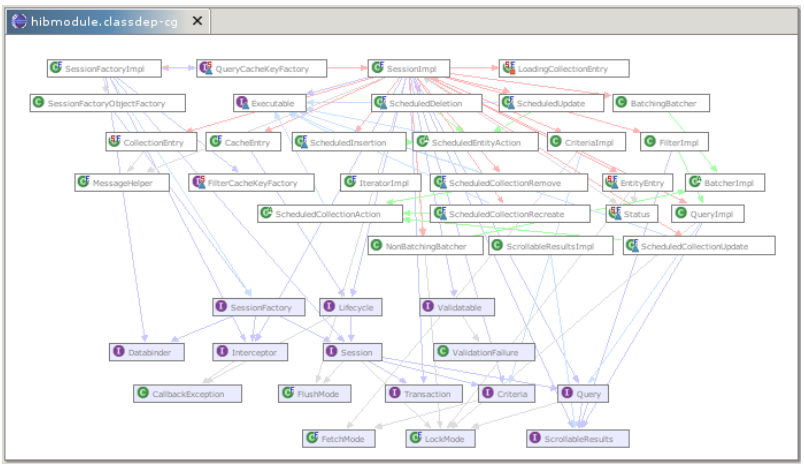

Abbildung 6.16: Inhalte des Implementierungspakets (oben) und Schnittstellenpakets (unten) für `net.sf.hibernate`

beschränkt ist. Ein Blick in die Deklarationen der Klassen reicht: die meisten Klassen aus `net.sf.hibernate.impl` sind zur Zeit `public`. Bevor wir diesen Entwurfsfehler korrigieren können, müssen wir herausfinden, welche Klassen von anderen Paketen direkt referenziert werden. Mit einer geeigneten Auswahl können wir diese Information leicht aus der tabellarischen Darstellung herausholen (Abbildung 6.17).

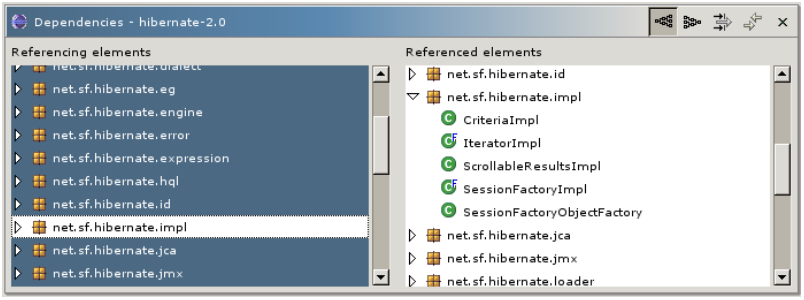

Abbildung 6.17: Die von außen referenzierten Klassen aus `net.sf.hibernate.impl`

Es gibt offenbar fünf Klassen, die wir nicht sofort durch die Entfernung des Zugriffsmodifikators `public` verstecken können. Dabei handelt es sich zum einen um “normale” Klassen, deren Instanzen außerhalb des Moduls erzeugt werden, zum anderen auch um Factory-Klassen, deren statische Methoden von Clients verwendet werden. Durch die Einführung einer neuen Factory-Klasse,

`HibernateFactory`, die die Objekterzeugung von den Clients gemäß Empfelungen aus Kapitel 4 übernimmt, lassen sich die Abhängigkeiten auf nur zwei konkrete Klassen beschränken. Zwanzig andere Klassen aus dem Paket `net.sf.hibernate.impl` können nun eine beschränkte Sichtbarkeit bekommen. Die Kommunikation mit dem Modul erfolgt über die in `net.sf.hibernate` definierte Modulschnittstelle, die aus zehn Interfaces zusammengesetzt ist. Wir müssen uns also als Modulnutzer schlimmstenfalls mit 2 konkreten Klassen beschäftigen statt mit 20.

6.8 Ergebnisse der Fallstudie

Als nächstes könnten wir jetzt eine Auseinandersetzung mit der Schnittstelle des ersten ausgesonderten Moduls durchführen – sind alle Clients an ihrer gesamten Form interessiert? Wenn nicht, dann sollten wir eine bessere Gliederung der Schnittstelle mit Hilfe des *Interface Segregation*-Prinzips überlegen.

Darüber hinaus bleiben in Hibernate noch viele Pakete, die ebenfalls von einer Modularisierung profitieren könnten. Im Prinzip müsste man für jedes von 30 Paketen eine ähnliche Abhängigkeitsanalyse wie die beschriebene durchführen, die Abstraktionsstufe jedes Pakets einschätzen, und die direkte Verwendung von Implementierungsklassen vermeiden.

Es fällt auf, dass wir uns im Laufe der Fallstudie vorwiegend mit leicht sichtbaren Merkmalen der Klassen und recht wenig mit ihrer Bedeutung beschäftigt haben. Dies erscheint als erster Schritt in Richtung Modularisierung sinnvoll – die offenbarsten Fehler können, wie wir gesehen haben, durch die ausschließliche Betrachtung der statischen Abhängigkeiten aufgedeckt und mit etwas Mühe beseitigt werden. Bei diesem Vorgehen bietet Classdep eine wertvolle Unterstützung. Es ist jedoch zu erwarten, dass erst ein tieferes Verständnis der Klassenbedeutung zu einem wirklich flexiblen modularen Aufbau führt. Wir haben mehrmals über die Abstraktheit von Klassen und Paketen spekuliert und dabei sind Irrtümer nicht auszuschließen. Sicherlich sind noch einige

Schwächen im betrachteten Modul und seiner Interaktion mit dem Rest des Systems unentdeckt geblieben. Wir erwarten, dass sie bei der Untersuchung weiterer Module ans Licht kommen.

Die Analyse von Klassen- und Paketabhängigkeiten kann als gutes Mittel zur Gewinnung von Einsichten über die innere Struktur und Funktionsweise eines Systems dienen. Im Unterschied zu den üblicherweise als Lernunterlagen verwendeten Dokumentationsseiten und Anwendungsbeispielen erfordert die Analyse eine aktive Auseinandersetzung mit dem Unbekannten. Sie wirft Fragen auf, die von uns ansonsten gar nicht gestellt würden, und lenkt gleichzeitig zu einer systematischen Erforschung von Antworten. Der Wunsch nach einer **Verbesserung** des vorliegenden Quellcode sollte in diesem Lernprozess niemals vernachlässigt werden. Er bindet eine emotionale Komponente ein, wodurch die sonst uninteressante und komplizierte Aktivität ein konkretes, motivierendes Ziel erhält. Damit wird auch eine bessere Einprägung der analysierten Sachverhalte möglich.

Die Abhängigkeitsanalyse eines fremden Projekts ist mit der Lösung eines großen Puzzles vergleichbar. Beide sind langwierige Prozesse, in denen die einzelnen Teile beleuchtet, mit anderen verglichen und umgeordnet werden müssen. Das Analysewerkzeug und unser Wissen über bestimmte Merkmale vorteilhafter Entwürfe, verkörpert in den Prinzipien und Richtlinien, sind wie ein gemaltes Muster. Sie reichen noch nicht alleine zur sofortigen Lösung des Rätsels, aber sie vermitteln uns, dass wir Fortschritte machen und wann wir mit der Arbeit aufhören dürfen.

Es fällt nicht schwer, ein fast fertiges Puzzle zu vervollständigen, wenn sich die meisten Spielstücke bereits in ihren richtigen Positionen befinden. Je näher man sich an der Lösung befindet, desto geringer wird die Anzahl von auszuprobierenden Kombinationen und frustrierenden Sackgassen, desto größer wird auch die Hoffnung, dass die Mühe sich wirklich lohnt.

Ebenso ist es mit dem Softwareentwurf und insbesondere mit der Beherr-

schung von Abhängigkeiten. In einem Projekt, dessen Abhängigkeiten vom Anfang an gepflegt werden, ist die schnelle Berichtigung der eintretenden Probleme mit keinem riesigen Aufwand verbunden – diese Erkenntnis folgt nicht aus der Fallstudie, sondern aus Erfahrungen einer anderen Nutzungsweise von Classdep. Wenn die Abhängigkeiten sich dagegen außer Kontrolle entwickeln – ein Vorgeschmack spürten wir bei der Befassung mit Hibernate – so erreicht man einen Stand, der sich nur noch mit großen (Zeit)Opfern wieder in Ordnung bringen lässt.

Man sollte keine Angst vor der steigenden Anzahl der Puzzle-Stücke kriegen – dies ist bei jedem komplexeren Softwareprojekt eine unvermeidbare Entwicklung – aber vor der Verwechslung der richtigen Stellen, wo diese Stücke hineinpassen. Mit Hilfe der Modularisierung strebt man an, das große Puzzle “Projekt” in eine Reihe leichterer aufzusplitten, die miteinander verknüpft sind. Ganz sicher fällt dann die Ermittlung der passenden Positionen für neue, unbekannte Stücke leichter.

Kapitel 7

Implementierung von Classdep

Das Werkzeug Classdep wurde als keine alleinstehende Java-Anwendung programmiert, sondern als Plugin für die IDE (*Integrated Development Environment*) Eclipse, welches ursprünglich durch die Firmen IBM und OTI entwickelt und später als Open-Source-Projekt veröffentlicht wurde. In diesem Kapitel wird auf Gründe eingegangen, die zur Entscheidung über die Integration von Classdep und Eclipse führten.

Im Anschluß werden solche Implementierungsmerkmale kurz erläutert, die für ein Werkzeug zur statischen Quellcodeanalyse charakteristisch sind. Dazu gehören die Verarbeitung der abstrakten Syntaxbäume und die Verwendung von bestimmten Graphalgorithmen.

7.1 Geschichte von Classdep

Die Geschichte von Classdep fing in August 2001 an, also zwei Jahre vor der Verfassung dieser Diplomarbeit. Damals wurde vom Autor die Idee eines Abhängigkeitsbrowsers für Java-Programme erstmalig als eine kleine Anwendung umgesetzt. Diese Urversion des Tools bot die ungefähre Funktionalität der tabellarischen Darstellung an. Auch die Zyklenermittlung und die Benachrichtigung über veränderte Paketabhängigkeiten gehörten zu den verfügbaren

Funktionen.

Im Unterschied zum neuen Classdep analysierte die erste Version keinen Quellcode von Java-Programmen, sondern lediglich den compilierten Bytecode. Daraus folgte eine wichtige Einschränkung: die Auskunft über Abhängigkeiten endete immer auf der Klassenebene – dem Entwickler wurde dann überlassen, die verantwortlichen Referenzen im Programmtext zu finden und auszuwerten. Nach einer externen Veränderung der Quelldateien mussten sie neu compiliert und wieder vom Anfang an analysiert werden.

Wegen der umständlichen Bedienung wurde die alte Version von Classdep bald nach dem Abschluß ihrer Entwicklung nur noch sporadisch eingesetzt. Als "umständlich" haben sich dabei nicht irgendwelche Fehler in der Benutzeroberfläche herausgestellt – diese wären leicht zu beseitigen gewesen – sondern bereits die Notwendigkeit, ein zusätzliches Programm außerhalb der gewöhnten Entwicklungsumgebung jedes Mal auszuführen und seine Ergebnisse mit dem Quellcode zu kombinieren. Bei üblich bearbeiteten Projektgrößen überschritt die für die Bytecode-Analyse erforderliche Wartezeit selten eine Minute. Nichtsdestotrotz stellte die Verwendung von Classdep eine Ablenkung dar, wodurch die Nutzungsvorteile deutlich gemindert wurden.

Der persönliche Wechsel der Entwicklungsumgebung von einer skriptbasierten (`gvim + jikes + ant`) zu einer grafisch orientierten (Eclipse) ließ die Ideen von Classdep frisch überlegen. Der starke Bedarf nach einer verbesserten Orientierung und nach erhöhter Qualität des produzierten Quellcode bestand nach wie vor, getrieben durch den Wunsch nach Zuverlässigkeit bei der Aufwandschätzung und Durchführung von Kundenaufträgen. Die IDE überzeugte besonders durch eine leichtere Veränderbarkeit des Quellcode dank dem eingebauten Refactoring. Dies war eine Aufgabe, deren Erfüllung vorher ebenfalls das Verlassen des Editors und den Einsatz von externen Werkzeugen erforderte. Es gab nun wieder Hoffnung, die Abhängigkeitsanalyse in einen unauffälligen, ja vielleicht sogar angenehmen Bestandteil der täglichen Programmierarbeit

zu verwandeln. Gleichzeitig schien es wichtig, parallel zur Implementierung des Werkzeugs theoretische Grundlagen zu erforschen, die zu seiner optimalen Verwendung führen. Das Ergebnis war eine komplette Neuentwicklung von Classdep mit Hilfe von objektorientierten Prinzipien, die zum Gegenstand dieser Diplomarbeit wurde. Kann man von einem Erfolg reden? Die alte Version wurde vergessen. Die neue wird heute tatsächlich benutzt und wartet auf die Erschließung der Vertriebswege.

7.2 Integration mit Eclipse

Es gibt mehrere nennenswerte Vorteile der IDE-Integration für Classdep:

- nahtlose Einbindung des Werkzeugs in den persönlichen Entwicklungsprozess
- Erweiterung um zusätzliche Funktionen, die nur im Kontext einer IDE Sinn machen
- Ausnutzung des vorhandenen, modernen Eclipse-Compilers zur Verarbeitung des Quellcode
- Ausnutzung der eingebauten Framework-Klassen zur Implementierung komplexer Elemente der Benutzerschnittstelle (wie z. B. Assistenten, Diagrammeditoren)
- Verbesserung und Vereinheitlichung der Benutzerschnittstelle durch die Befolgung von ergonomischen Richtlinien des Eclipse-Projekts [28]
- Erleichterung von Entwurfsentscheidungen durch die Nutzung bewährter Grundkonzepte der Eclipse-Workbench [29]
- Zugang zum vollständigen, lauffähigen Quellcode von Eclipse als Hilfe bei der Lösung von Implementierungsproblemen und auf der Suche nach Beispielen

- kein Bedarf nach einer zusätzlichen Installationssoftware dank dem standarisierten Distributionsmechanismus für Plugins
- Erreichung eines breiten Benutzerkreises

Die folgenden Abschnitte geben einen Überblick über die Integration von Classdep mit Eclipse.

7.2.1 Die Plugin-Architektur von Eclipse

Es gibt viele IDEs von verschiedenen Herstellern. Ein Nachteil der IDE-Verwendung hielt den Autor sehr lange bei der bewährten Kombination von einfachen, mächtigen Kommandozeilewerkzeugen fest. Bei den IDEs gab es schon immer Bedienungsaufgaben, deren Erfüllung sehr langsam oder sogar "unmöglich" war. Die Anpassung an die Bedürfnisse des Programmierers erfolgte immer in einem engen Rahmen, der vom jeweiligen IDE-Hersteller festgesetzt wurde. Die Wahl einer IDE bedeutete also einen teilweisen Verzicht auf die Freiheit der produktiven Innovation. Sicher konnten bestimmte Aufgaben, wie z. B. die GUI-Entwicklung, schneller mit Mausklicks als mit Tastendrücken erledigt werden. Die Schwierigkeiten fingen dann an, wenn man über die Vorstellungen eines IDE-Anbieters hinausging.

Bei Eclipse sollte es anders werden. IBM konzipierte das Produkt als eine "offene Platform", die von Entwicklern nicht nur frei konfiguriert, sondern auch beliebig erweitert werden sollte. Damit diese Vorstellung auch zur Realität wird, entschied sich die Firma für eine liberale Open-Source-Lizenz und stellte das Produkt allen interessierten Personen kostenlos zur Verfügung. Zum Nachweis, dass die "Grundlage" tatsächlich die erwünschten Erweiterbarkeitspotentiale besitzt, wurde darauf eine vollständige Java-IDE gebaut – und mit gleichen günstigen Nutzungsbedingungen veröffentlicht. Heute gehört Eclipse zu den führenden Java-Entwicklungsumgebungen[1] – ihr schneller Aufstieg ist wohl auf

[1] vorläufig zweiter Platz in der Online-Umfrage "Best Java IDE Environment" der Zeitschrift Java Developer's Journal in 2003

ihrem spezifischen Anwendungsgebiet mit dem von Linux als Betriebssystem vergleichbar.

Eclipse erfüllt den Wunsch nach Flexibilität durch eine gut durchdachte Plugin-Architektur. Als Plugin wird ein Bündel aus Java-Code, Ressourcen und einer deklarativen Beschreibung (dem sog. Plugin-Manifest) bezeichnet. Beim Start der Eclipse-IDE werden alle installierten Plugins entdeckt und ihre Manifeste in ein zentrales Plugin-Verzeichnis eingetragen. Die Rolle jedes Plugins besteht darin, im Kontext der Eclipse-Plattform bestimmte Dienste für den Benutzer oder auch für andere Plugins anzubieten.

Die Aktivierung eines Plugins, d.h. das Laden und Ausführen seines eigentlichen Code, erfolgt üblicherweise erst beim Eintritt eines Ereignisses, an welchem der Plugin-Entwickler sein Interesse in der Beschreibungsdatei deklariert hat. Es könnte sich dabei z. B. um die Auswahl einer Menüoption oder das Öffnen einer bestimmten Datei handeln. Die Ereignismenge ist durch den Entwurf von Eclipse nicht nach oben beschränkt: neue Plugins dürfen neue Erweiterungspunkte definieren, die später von anderen implementiert werden.

Noch allgemeiner betrachtet besitzt jedes Plugin Zugriff auf das Plugin-Verzeichnis über das Plattform-API (*Application Programming Interface*) und kann daraus Beschreibungen aller lokal installierten Plugins auslesen. Damit können solche Plugins gefunden werden, die an bestimmten Erweiterungspunkten des aktiven Plugins interessiert sind oder ihrerseits bestimmte Dienste anbieten. Dem Plugin-Entwickler obliegt es, die Dienste anderer Plugins im Manifest seines eigenen anzufordern und die eigenen Dienste zu veröffentlichen. In meisten Fällen erwartet ein Plugin von seinen Nutzern außer der Deklaration im Manifest auch die Implementierung von bestimmten APIs und stellt für sie im Gegenzug direkt ansprechbare Klassen zur Verfügung. Eine gute technische Behandlung der Konzepte, die der Plugin-Architektur von Eclipse zu Grunde liegen, findet man in [11].

Bei Eclipse-Plugins handelt es sich um eine spezielle Form des allgemeinen

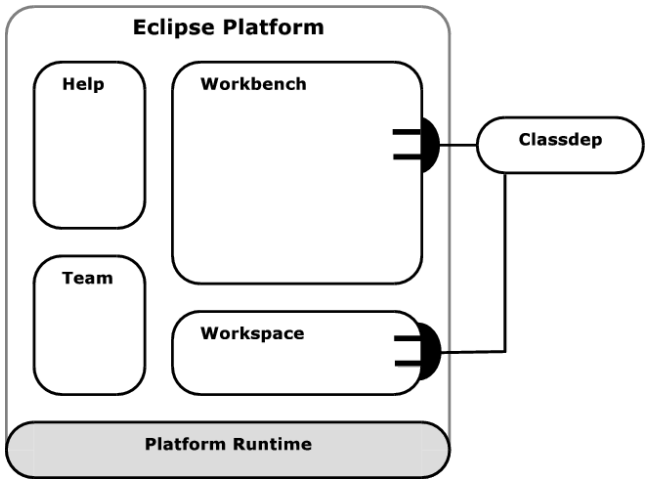

Abbildung 7.1: Einbindung von Classdep in Eclipse

Modulkonzepts. Hinter einer Schnittstelle, die aus den im Plugin-Manifest angegebenen Erweiterungspunkten und frei dokumentierten APIs besteht, wird die Implementierung bestimmter Funktionen versteckt. Die Kommunikation zwischen Plugins soll nur über ihre expliziten Schnittstellen erfolgen, auch wenn sich übrige Implementierungsklassen in der Reichweite des Programmierers befinden (d.h. als `public` deklariert sind). Alle beschriebenen Prinzipien der Modularisierung sind bei der Entwicklung von Eclipse-Plugins direkt anwendbar.

7.2.2 Einbindung von Classdep in Eclipse

Nach der allgemeinen Einführung der Plugin-Architektur wird nun am Beispiel von Classdep ihre konkrete Nutzung demonstriert. Classdep ist in die Eclipse-Plattform durch die Implementierung von acht verschiedenen Erweiterungspunkten eingebunden, die durch zwei Plugins aus der Basisinstallation angeboten werden:

Erweiterungspunkt	**Rolle der Erweiterung**
`org.eclipse.ui.editors`	Verknüpfung von Diagrammeditorklassen mit den Suffixen der Diagrammdateien

`org.eclipse.ui.views`	Eintragung der tabellarischen Darstellung und Zyklendarstellung in die Hierarchie von Views, die der IDE-Benutzer öffnen kann
`org.eclipse.ui.popupMenus`	Hinzufügen des Classdep-Untermenü zum Kontextmenü für Java-Projekte
`org.eclipse.ui.newWizards`	Hinzufügen der Assistenten zur Erstellung von Klassen- und Paketdiagrammen
`org.eclipse.core.resources.natures`	Deklaration einer "Classdep-Natur" für überwachte Java-Projekte[2]
`org.eclipse.core.resources.builders`	Einbindung einer Builder-Klasse zur Protokollierung von Quelldateiänderungen für Projekte mit der "Classdep-Natur"
`org.eclipse.core.resources.markers`	Deklaration eines neuen Task-Typs zur Benachrichtigung des Benutzers über veränderte Paketabhängigkeiten

Darüber hinaus verlässt sich Classdep auf Klassen, die durch insgesamt acht andere Plugins geliefert werden. Classdep bietet zur Zeit keine eigenen Erweiterungspunkte oder APIs für andere Plugins, obwohl eine zukünftige Einrichtung solcher Schnittstellen durchaus denkbar ist.

Bei der Integration mit Eclipse stellte sich heraus, dass die vorhandenen Erweiterungspunkte für eine zufriedenstellende Implementierung der gewünschten Funktionalität nicht ausreichten. Die Kernaufgabe von Classdep besteht darin, Referenzen zu Klassen und Interfaces in Java-Quellcodedateien zu ermitteln und auf dieser Basis die intern gepflegten Abhängigkeitsgraphen zu aktualisieren. Ein großer Teil der notwendigen Verarbeitungslogik ist bereits im eingebauten Java-Compiler von Eclipse implementiert. Leider bietet das Compiler-Plugin zur Zeit keine Erweiterungspunkte, die eine spezielle Auswertung des vom Parser während der Compilierung erstellten Syntaxbaums

ermöglichen würden. Die zur Verfügung stehenden dokumentierten APIs, die zu einer nachträglichen Bearbeitung des Syntaxbaums eingesetzt werden können, haben sich aus Performanzgründen als ungeeignet erwiesen. Stattdessen werden also von Classdep konkrete Implementierungsklassen aus dem Compiler-Plugin (`org.eclipse.jdt.core`) verwendet. Ihre Verfügbarkeit und Unveränderlichkeit steht jedoch in den zukünftigen Versionen von Eclipse nicht fest.

Diese Verletzung des *information hiding*-Prinzips im Kontext von Classdep und Eclipse weist auf ein praktisches Problem hin, das bei der Modularisierung von Programmen vorkommt: eine Schnittstelle kann auch **zu viel** Information verstecken, wodurch das betroffene Modul für seine Nutzer weniger interessant wird. Bei diesem Entwurfsfehler werden Clients dazu gezwungen, sich auf undokumentierte Implementierungsmerkmale zu verlassen oder die gleichen Aufgaben wie das Server-Modul redundant zu lösen (Kopieren der Implementierung). Beide Wege führen auf lange Sicht zu Schwierigkeiten. Die offenbare Gegenmaßnahme besteht darin, bereits beim Entwurf von Modulschnittstellen an Bedürfnisse von zukünftigen Clients zu denken. Dies sollte gemäß dem *Open Closed*-Prinzip passieren, wobei man jedoch noch nicht vorhandene Anforderungen leicht übersehen kann. Eine andere, praxisnähere Lösung besteht in einer nachträglichen Schnittstellenerweiterung je nach Bedarf. Werden die Module wie im Fall von Classdep und JDT-Compiler getrennt und von verschiedenen Personen entwickelt, so erfordert die Schnittstellenanpassung verständlicherweise mehr Zeit. Als zusätzliches Hindernis kommt die Anforderung hinzu, dass die gewünschte Erweiterung einer Modulschnittstelle für alle ihre bisherigen Clients reibungslos verlaufen muss und nach der Veröffentlichung kaum rückgängig gemacht werden darf. [30]

7.3 Modularer Aufbau

In Abbildung 7.2 sind die wichtigsten Module von Classdep aufgeführt. Um eine bessere Übersicht im Diagramm zu erreichen, wurden die einzelnen Mo-

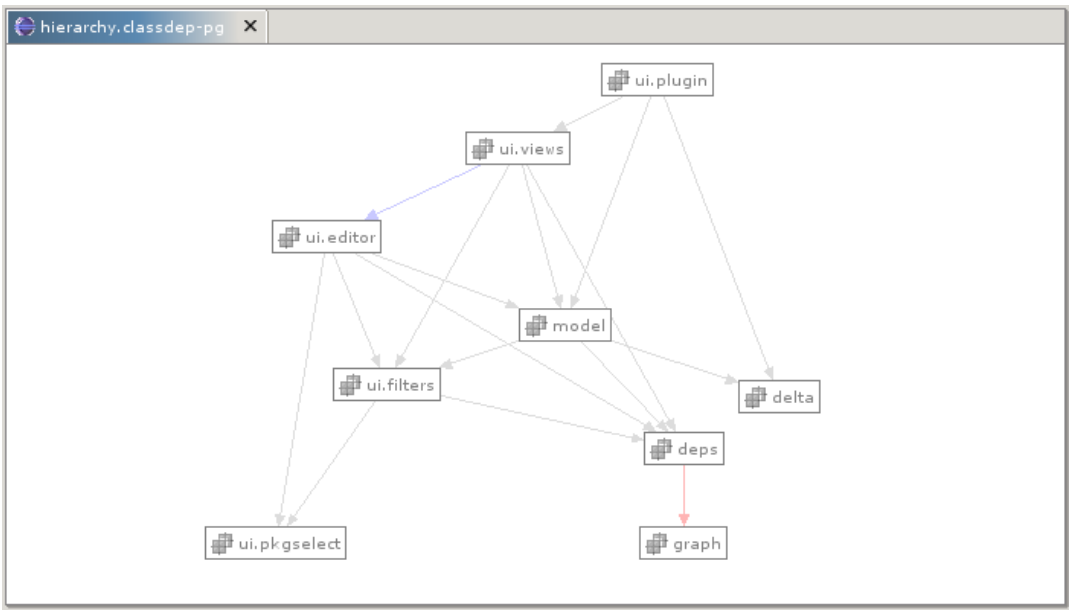

Abbildung 7.2: Modulabhängigkeiten in Classdep

dule aus den Bereichen `ui.editor` und `ui.views` mit Hilfe von Paketgruppen zusammengefasst.

Auf der untersten Ebene der Hierarchie befindet sich das Modul `graph`, welches eine Implementierung einer allgemeinen Graphklasse und im übernächsten Kapitel beschriebenen Graphalgorithmen enthält. Sein einziger Nutzer ist das wichtige Modul `deps` (*dependencies*), in dem Schlüsselkonzepte zur Beschreibung der Abhängigkeiten in Java-Projekten definiert sind. Im Herzen des Moduls befindet sich die Klasse `ProjectDependencies`, die eine Schnittstelle zur Abfrage aller innerhalb eines Projekts vorliegenden Typ- und Paketabhängigkeiten implementiert. Zur Modulimplementierung gehören auch Klassen, die den JDT-Compiler von Eclipse steuern und eine `ProjectDependencies`-Instanz mit Daten versorgen. Die meisten Clients dieses Moduls brauchen ausschließlich den Lesezugriff auf die Abhängigkeitsdatenbank. Diese Tatsache wird im Entwurf ausgenutzt, um eine entsprechende Schnittstellentrennung vorzunehmen – die Klasse `ProjectDependencies` bietet zwei Schnittstellen nach außen: `IProjectDependencies` (Lesezugriff) und `IUpdateableProjectDeps` (Lese- und Schreibzugriff). Die Visualisierung der Abhängigkeiten und Behandlung von

Benutzeraktionen liegt auf jeden Fall außerhalb des Kompetenzbereichs des Moduls `deps`.

Das Modul `ui.filters` bietet eine Schnittstelle zur Filterung und Gruppierung von Knoten vor ihrer visuellen Darstellung. Hier werden die Konzepte einer Paketgruppe und eines Filters definiert. Zur Implementierung gehören darüber hinaus Masken, mit denen die Inhalte der Filterobjekte manipuliert werden können. Für den Rest der Anwendung sind Filter unveränderbar. Sie werden typischerweise in Schleifen bei der Verarbeitung von Knotenmengen benutzt. Zusätzlich wird vom Modul `ui.filters` eine Schnittstelle zum Auflisten und Laden aller verfügbaren Filter je Projekt angeboten. Die Implementierung dieser Schnittstelle basiert zur Zeit auf den sog. *persistent resource properties* von Eclipse. In einer späteren Version wird sie möglicherweise mit einer direkten Speicherung von Filtern im Dateisystem ersetzt.

Im Modul `ui.pkgselect` wurde die Maske zur Paketauswahl und implementiert, die zur Zeit nur der Klassendiagrammerstellung eingesetzt wird. Da es sich aber um ein allgemeines Element der Benutzeroberfläche handelt, wird dieses Modul vermutlich auch in anderen Masken Verwendung finden. Eine Ausgliederung dieses Moduls aus Classdep und sein direkter Einsatz in ähnlichen Plugins ist denkbar. Dafür sprechen auch die fehlenden Abhängigkeiten von anderen Modulen.

Das Modul `delta` ist für die "Überwachung" der Java-Projekte zuständig. Es protokolliert Veränderungen der Quelldateien, die von Eclipse gemeldet werden, und bietet immer Auskunft über Unterschiede im Projektzustand seit der letzten inkrementellen Aktualisierung der Abhängigkeitsgraphen. Dieses Modul, wie man im Diagramm sehen kann, weiß nichts von den restlichen – es arbeitet auf der Projektressourcenebene (Dateien und Verzeichnisse) ausschließlich mit der IDE zusammen.

Es gibt einen guten Grund, warum das Modul `model` sich in der Mitte des Diagramms befindet. Dieses Modul koordiniert die anderen Module und ver-

waltet die dauerhafte Speicherung von Projektabhängigkeiten. Eine Singleton-Instanz der Klasse `Model` wird bei der Plugin-Initialisierung angelegt und ist für alle Module ansprechbar, die sich mit der Visualisierung der Abhängigkeiten befassen. Ein Beispiel für die steuernde Rolle von `model` ist die Auswertung der Inhalte des `delta`-Protokolls bei der inkrementellen Aktualisierung des Abhängigkeitsgraphen im `deps`-Modul. Im Modul `model` wird auch das "aktive Java-Projekt" vermerkt, welches zu jedem Zeitpunkt von allen Classdep-Views dargestellt wird. Wählt der Benutzer ein anderes Projekt zur Analyse, so sorgt `model` für eine entsprechende Benachrichtigung der GUI-Module. Es kann also nicht vorkommen, dass die Zyklendarstellung die Inhalte eines anderen Projekts als die tabellarische Darstellung zeigt.

Interessanterweise ist `model` ein "Problemmodul", dessen Programmierung die meisten Schwierigkeiten bereitete – ein Schwachpunkt im Entwurf von Classdep. Diese Beobachtung könnte man vielleicht auf Module verallgemeinern, die eine ungefähr gleiche Anzahl von ein- und ausgehenden Abhängigkeiten besitzen.

Die Modulgruppe `ui.editor` umfasst die allgemeinen Diagrammeditormodule und die speziellen davon abgeleiteten Module der Paket- und Klassendiagrammeditore. Die Abhängigkeiten zeigen, dass die Datenbeschaffung über das Modul `model` erfolgt. Die bereits erwähnte Nutzung von `ui.pkgselect` in Klassendiagrammen ist in Abbildung 7.2 ebenfalls als Abhängigkeit zu sehen.

Die Zyklendarstellung und die tabellarische Darstellung von Abhängigkeiten wurden zur Modulgruppe `ui.views` zusammengefasst. Die Views kommunizieren ähnlich wie `ui.editor` mit `model`, um die darzustellenden Abhängigkeiten zu erfragen. Die Abhängigkeit von `ui.editor` ist in der Verknüpfung der Auswahl in der tabellarischen Darstellung mit Diagrammeditoren begründet.

Auf der obersten Ebene der Modulhierarchie befindet sich `ui.plugin`. In diesem Modul ist die Classdep-Perspektive definiert. Es beinhaltet auch die Plugin-Klasse, deren einzige Instanz von Eclipse initialisiert wird. Die Abhän-

gigkeit von `delta` ergibt sich aus der Tatsache, dass die Benutzeraktion "Projektüberwachung ein- und ausschalten" im Modul `ui.plugin` implementiert ist, während die eigentliche Protokollierung innerhalb von `delta` erfolgt.

Zwei Module, die fast von überall referenziert werden, wurden im Diagramm weggelassen: `log` und `ui.resource`. Das Modul `ui.resource` bietet Zugriff auf die Datenbank mit von Classdep verwendeten Zeichenketten und Grafiken. Die Zentralisierung solcher Informationen ist eine Standardtechnik, die eine Übersetzung der Benutzeroberfläche in andere Sprachen erleichtert.

Das Modul `log` wird von anderen Modulen mit der Protokollierung "ungewöhnlicher" Ereignisse und Programmzustände beauftragt, die ggf. auf interne Fehler in Classdep hinweisen. Die jetzige Implementierung schreibt Nachrichten in das `Error Log` von Eclipse, wo sie für den informierten Benutzer beim Auftreten von Problemen einsehbar sind.

7.4 Syntaxanalyse

Die ausschließliche Informationsquelle für alle von Classdep angebotenen Anzeigen ist der Quellcode eines Java-Projekts. Jedes Mal, wenn der Benutzer sich eine aktuelle Darstellung von Abhängigkeiten in der einen oder anderen Form wünscht, wird möglicherweise die Syntaxanalyse einer oder mehrerer Java-Quelldateien ausgelöst.

Wie bereits erwähnt, beauftragt Classdep dann den eingebauten Compiler von Eclipse mit der Erstellung eines abstrakten Syntaxbaums für jede zu analysierende Datei. Da der normale Compilerlauf mit der Syntaxanalyse und mit der Auflösung von Typnamen nicht endet, sondern von einer Bytecode-Erzeugungsphase gefolgt wird, musste die Compiler-Klasse mit Hilfe einer abgeleiteten Klasse angepasst werden. Die eigentliche Implementierung des Parsers und die damit erledigte Überprüfung der Eingaberichtigkeit bleiben davon unberührt.

Als Ergebnis der Syntaxanalyse wird von der angepassten Compiler-Klasse

ein fertiger Syntaxbaum geliefert – d.h. eine gemäß Java-Grammatik aufgegliederte Darstellung des Quelldateiinhalts. Falls es Syntaxfehler gab, werden gewisse Teile des Syntaxbaums gar nicht generiert. Eine zuverlässige Abhängigkeitsanalyse ist deswegen nur auf einem compilierbaren Quellcode durchführbar.

Der Baum wird mit Hilfe eines von Classdep implementierten Visitor-Typs ausgewertet. Nur diejenigen Knoten, in welchen Typreferenzen vorkommen können (wie z. B. Parameterlisten, Variablendeklarationen), werden näher untersucht, alle anderen einfach übersprungen. Bis auf wenige Ausnahmen müssen zwischen einzelnen Knotenbesuchen keine Zustandsdaten verwaltet werden. Für eine gefundene Typreferenz werden je zwei Knoten im Typabhängigkeitsgraphen erzeugt, soweit sie noch nicht existieren, und eine gerichtete Kante, die sie verbindet.

7.5 Verwendete Graphalgorithmen

Im Herzen von Classdep liegt die Verwaltung der Abhängigkeitsgraphen. Für jedes Java-Projekt werden spätestens vor dem ersten Öffnen einer Abhängigkeitsanzeige zwei gerichtete Graphen erzeugt: der Typabhängigkeitsgraph und der Paketabhängigkeitsgraph. Im Typabhängigkeitsgraphen wird jede Kante mit der vorliegenden Abhängigkeitsart attributiert. Die im Paketabhängigkeitsgraphen enthaltene Information lässt sich aus dem Typabhängigkeitsgraphen durch das Zusammenfassen von Knoten ableiten und dient zur Beschleunigung der Abfragen. Dabei werden alle Typknoten, die zum gleichen Paket gehören, zu einem Paketknoten zusammengefasst, dessen Mengen an ein- und ausgehenden Kanten eine Vereinigung der entsprechenden Kantenmengen der enthaltenen Typknoten darstellen.

Bis auf die Zyklenermittlung sind die Abfragen der Graphinhalte auf das Finden von Kanten beschränkt, die zum oder vom jeweiligen Knoten oder einer vorgegebenen Menge von Knoten führen. Da es sich um dünne Graphen han-

delt, werden zur Speicherung Adjazenzlisten verwendet. Um das Problem der parallelen Kanten (Duplikate bei mehreren Referenzen zum gleichen Typ) zu eliminieren, sind die Adjazenzlisten als Hashtabellen implementiert. Die Reihenfolge der Elemente in den Listen spielt keine Rolle. Für jeden Knoten werden zwei Mengen von Nachbarknoten abgelegt. Ein Nachbar wird zu der jeweiligen Menge zugeordnet, wenn er über eine Kante vom betrachteten Knoten erreichbar ist oder eine Kante besitzt, die zum betrachteten Knoten führt. Diese redundante Speicherung des Kantenverlaufs erlaubt es, Abfragen nach ein- und ausgehenden Kanten mit dem gleichen Zeitaufwand zu beantworten. Die Nachteile liegen im verdoppelten Aktualisierungsaufwand und dem erhöhten Speicherplatzbedarf für die Kanten.

Die dynamische Aktualisierung der beiden Graphen erfolgt bei "überwachten" Projekten, wenn Änderungen in Java-Quelldateien festgestellt werden. Liegen Änderungen vor (gelöschte oder aktualisierte Dateien), so werden zunächst alle Knoten aus dem Typabhängigkeitsgraphen entfernt, die für in den betroffenen Dateien deklarierte Typen stehen. (In jedem Typknoten wird der Name der deklarierenden Java-Datei abgelegt). Zusätzlich werden auch alle Knoten gelöscht, die für Typen stehen, die zu der zuvor ermittelten Typmenge Referenzen enthielten. Zusammen mit den beiden Knotenmengen werden ebenfalls ihre ausgehenden Kanten entfernt. Danach werden (nur) die veränderten und neu hinzugekommenen Java-Dateien geparst und die in ihnen gefundenen Referenzen zum Einfügen von Knoten und Kanten in den Typabhängigkeitsgraphen verwendet. Anschließend wird der Paketabhängigkeitsgraph auf den neuesten Stand gebracht. Dies geschieht durch das Löschen von betroffenen Paketknoten und Kanten und ihr erneutes Einfügen an Hand des aktualisierten Typabhängigkeitsgraphen.

Die Zyklenermittlung erfolgt mit Hilfe einer Tiefensuche im Paketabhängigkeitsgraphen, bei der eine Kantenklassifizierung stattfindet. Eine rekursive Suche wird von jedem Knoten mit ausgehenden Kanten erneut gestartet, es sei

denn, dass dieser Knoten bereits verarbeitet wurde. Bei jedem Durchlauf wird eine geordnete Liste von besuchten Knoten geführt, die beim Einschlagen eines neuen Pfads um seinen Startknoten erweitert wird. Beim Verlassen des Pfads (Rückkehr aus der rekursiven DFS-Routine) wird das letzte Element aus der Liste entfernt. Wenn mit Hilfe dieser Liste eine Rückkante festgestellt wird, d.h. eine Kante, die zu einem bereits besuchten Knoten verläuft, dann wird der Listeninhalt als Zyklus weggespeichert. Die Nachbarknoten werden in diesem Fall nicht mehr besucht, um eine endlose Rekursion zu vermeiden.

Die Zyklensuche im Paketabhängigkeitsgraphen beansprucht trotz ihrer einfachen Implementierung nur einen Bruchteil der Verarbeitungszeit, die für die Graphaktualisierung notwendig ist – der meiste Aufwand liegt in der davor stattfindenden syntaktischen Analyse der Quellcodedateien. Möchte man die Effizienz der Zyklenermittlung dennoch steigern, so wäre es angemessen, als Vorstufe die stark zusammenhängenden Graphkomponenten zu finden. Eine stark zusammenhängende Komponente ist ein Teilgraph, der aus Knoten besteht, die paarweise voneinander erreichbar sind – nur in solchen Teilgraphen können Zyklen überhaupt vorhanden sein. Es gibt effiziente Algorithmen zur Ermittlung der stark zusammenhängenden Komponenten. In Classdep wurde der SCC-Algorithmus von Kosaraju [31] implementiert. Er wird aber bisher nicht eingesetzt.

Zur Ausrichtung von Knoten in Diagrammeditoren wird zur Zeit ein externes Programm verwendet – “dot” aus dem Paket Graphviz [32]. Das Programm ist zentral auf einem Server installiert und wird von Classdep über ein Web-Frontend mit der Layout-Aufgabe beauftragt. Eine Beschreibung der von dot verwendeten Graphalgorithmen findet man in [33].

Kapitel 8

Fazit und Weiterentwicklung

Zum Anfang dieser Diplomarbeit gab es für Classdep einen klar umrissenen Funktionsumfang, der sich aus Merkmalen der alten Version zusammensetzte. Die Vorteile und Realisierbarkeit der IDE-Integration standen aber damals noch unter einem Fragezeichen. Es war das erste Plugin-Projekt eines relativ frischen Eclipse-Anwenders, und es wurde dementsprechend mit einer gesunden Dosis von Mißtrauen gestartet. “Classdep wird so implementiert, dass seine Nutzung innerhalb von Eclipse, im alleinstehenden Modus mit Benutzeroberfläche oder auch im Batch-Modus möglich sein wird. Als Benutzeroberfläche werden wahlweise Swing oder SWT angeboten” – so hießen die ersten Planungen. Es stellte sich ziemlich schnell heraus, dass diese Annahmen unrealistisch waren, weil die beabsichtigte Portierung eines bestehenden Java-Programms in eine neue Umgebung durch eine komplette Neuimplementierung ersetzt wurde. Insbesondere hat die enge IDE-Integration Vorrang vor dem ursprünglichen Wunsch nach unterschiedlichen Nutzungsszenarien genommen. Heute ist klar, dass die Trennung von Eclipse nicht nur mit einem hohen Aufwand verbunden wäre, sondern auch den Nutzwert des Werkzeugs mindern würde. Die erfolgreiche Neuentwicklung von Classdep ist ein perfektes Beispiel dafür, dass man in manchen Fällen weder die Benutzeranforderungen noch den richtigen Softwareentwurf vorab fixieren kann – oder soll.

Ein Funktionsmerkmal, mit dem in der frühen Entwicklungsphase große Hoffnungen verbunden wurden, wurde gar nicht implementiert: die automatische Zuordnung der neuen Klassen zu Paketen oder die Optimierung von Paketabhängigkeiten durch automatische Klassenverschiebungen zwischen Paketen. Die Ausführungen über die Schwächen der statischen Analyse, die Unzulänglichkeiten von Abstraktheitsmetriken und die zusätzlichen Überlegungen zur Ermittlung der Code-Bedeutung, die unter anderem in der Fallstudie erkennbar wurden, lassen vermuten, dass die Festlegung der Klassen-Paket-Beziehungen ohne Entwicklerbeteiligung nicht realisierbar ist. Dennoch bleibt die Frage offen, ob man keine direktere Unterstützung des üblichen Entscheidungsprozesses anbieten könnte. Als Modell wäre ein Szenario zu überlegen, in dem alle Typen eines bereits gut strukturierten Projekts in ein einzelnes Paket zusammengeführt werden. Wie schwierig wäre es für einen Außenseiter die ursprüngliche ordentliche Pakethierarchie wiederherzustellen? Welche konkreten Lösungshilfen könnte das Werkzeug für dieses Problem anbieten? Eine weitere Verwendung von Classdep zur Analyse von bestehenden Softwareprojekten könnte zur Beantwortung dieser Fragen beitragen.

Eine grundlegende Frage betrifft den allgemeinen Nutzen der statischen Quellcodeanalyse, insbesondere im Vergleich mit anderen Qualitätssicherungsmaßnahmen (das Zeitbudget eines Projekts ist schließlich immer nach oben beschränkt). Bis auf subjektive Benutzerurteile liegen zur Zeit keine Daten vor. Bei einer wissenschaftlichen Untersuchung müsste man in einer kontrollierten Umgebung über einen längeren Zeitraum Metriken einsetzen, die eine Messung der Softwarequalität und Entwicklerproduktivität ermöglichen. Es erscheint dabei sinnvoll, die Anwendung der Abhängigkeitsanalyse auf bestehende Projekte von ihrem durchgehenden Einsatz bei neuen Projekten zu unterscheiden. Es gibt Gründe zu vermuten, dass der Analyseaufwand mit der Anzahl der im Projekt befindlichen unerwünschten Abhängigkeiten überproportional steigt.

Im Rahmen der Fallstudie wurde die Rolle der Abhängigkeitsanalyse bei der Erforschung unbekannter Softwareprojekte erwähnt. Es stellt sich die Frage, inwiefern der Einsatz eines Analysewerkzeugs einerseits mit der Lektüre der Dokumentation und andererseits mit der aktiven Schulung des Einsteigers durch ein Projektmitglied konkurrieren kann. In diesem Zusammenhang wäre auch interessant, die Einsatzmöglichkeiten der Quellcodeanalyse bei der gleichzeitigen Einarbeitung mehrerer neuen Projektteilnehmer zu untersuchen – kann die Werkzeugverwendung die Kommunikation zwischen solchen Personen fördern?

Die Wunschliste mit zukünftigen Features beinhaltet aber auch viel konkretere Punkte, die eher Implementierungs- als Forschungsarbeit erfordern. Hierzu gehören:

- verbesserte Unterstützung von Refactoring (z. B. durch Direktmanipulation von Diagrammen)
- einfachere Erstellung und Auflösung von Paketgruppen (durch Markierung von Paketknoten oder Angabe von regulären Ausdrücken)
- Hervorheben von Abhängigkeitszyklen in Paketdiagrammen
- eine Konfigurationsmaske mit globalen Plugin-Einstellungen
- Exportieren von Abhängigkeitsdaten in einem portablen Format
- Kopieren von Filtereinstellungen bei der Erstellung neuer Filter
- direkte Einbindung der Graphlayoutalgorithmen (Verzicht auf Netzwerkkommunikation)
- bequeme Skalierung von Knotenabständen in Diagrammen
- Zerteilung in mehrere möglichst unabhängige Plugins
- Schaffung und Dokumentierung von Schnittstellen für externe Plugins
- Erstellung der integrierten Online-Hilfe in englischer Sprache

- Erstellung von Webseiten, Werbematerial und Tutorials

Ein wichtiger verbleibender Punkt ist die Veröffentlichung von Classdep und die Gewinnung an Publizität für seinen Einsatz. Der Autor ist davon überzeugt, dass eine breitere Benutzung des entwickelten Werkzeugs sowohl für andere Entwickler als auch für die Erweiterung des Funktionsumfangs vorteilhaft wäre. Vor der Veröffentlichung sollten jedoch noch die heute bekannten Bedienungsprobleme in Classdep beseitigt und geeignete Vetriebswege gefunden werden. In erster Linie ist zwischen einer Kommerzialisierung des Tools und einer freien Veröffentlichung des gesamten Quellcode zu entscheiden. In beiden Fällen müssten die Beteiligung von externen Beiträgern und die Lizenzbedingungen überlegt und geregelt werden.

Literaturverzeichnis

[1] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: Design Patterns, 1st edition, Addison-Wesley Pub Co, 1995.

[2] Foote Brian; Yoder, Joseph: Big Ball of Mud, Fourth Conference on Patterns Languages of Programs, 1997.

[3] Fowler, Martin: Refactoring: Improving the Design of Existing Code, 1st edition, Addison-Wesley Pub Co, 1999.

[4] Gosling, James; Joy, Bill; Steele Jr., Guy L.: The Java™Language Specification, 2nd edition, Addison-Wesley Pub Co, 2000.

[5] Guttag, John: Abstract Data Types, sd&m Conference 2001, <http://www.sdm.de/en/termine/sdmkonf-2001>

[6] Brooks, Frederick P.: The Mythical Man-Month, 1st edition, Addison-Wesley Pub Co, 1995.

[7] Williams, Sara; Kindel, Charlie: The Component Object Model: A Technical Overview, <http://msdn.microsoft.com/library/en-us/dncomg/html/msdn_comppr.asp>

[8] Green, Dale: The J2EE™Tutorial: Enterprise Beans, <http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts.html>

[9] Parnas, D.: On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, Vol. 15, No. 12, Dezember 1972, pp. 1053-1058.

[10] Meyer, B.: Object-Oriented Software Construction, 2nd edition, Prentice Hall PTR, 2000.

[11] Bolour, A.: Notes on the Eclipse Plugin Architecture, <http://eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html>

[12] Boehm, B.; Basili, Victor R.: Software Defect Reduction Top 10 List, Computer, Januar 2001 (Vol. 34, No. 1), <http://www.cebase.org/www/researchActivities/defectReduction/top10>

[13] Martin, Robert C.: The Open Closed Principle, C++ Report, Januar 1996, <http://www.objectmentor.com/resources/articles/ocp.pdf>

[14] Martin, Robert C.: The Dependency Inversion Principle, Juni 1996 C++ Report, 1996, <http://www.objectmentor.com/resources/articles/dip.pdf>

[15] Martin, Robert C.: Interface Segregation Principle, C++ Report, August 1996, <http://www.objectmentor.com/resources/articles/isp.pdf>

[16] Martin, Robert C.: Granularity, C++ Report, November/Dezember 1996, <http://www.objectmentor.com/resources/articles/granularity.pdf>

[17] Martin, Robert C.: Stability, C++ Report, 1997, <http://www.objectmentor.com/resources/articles/stability.pdf>

[18] Portland Pattern Repository's Wiki: Open Closed Principle,
<http://c2.com/cgi/wiki?OpenClosedPrinciple>

[19] JUnit, Testing Resources for Extreme Programming,
<http://www.junit.org>

[20] Mackinnon T.; Freeman S.; Craig P.: Endo-Testing: Unit Testing with Mock Objects, eXtreme Programming and Flexible Processes in Software Engineering - XP2000 Conference,
<http://www.mockobjects.com>

[21] Jeffries, Ron; Anderson, Ann; Hendrickson, Chet: Extreme Programming installed, 1st edition, Addison-Wesley Pub Co, 2001.

[22] Wells, D.: Extreme Programming: A gentle introduction,
<http://www.extremeprogramming.org>

[23] Manifesto for Agile Software Development,
<http://www.agilemanifesto.org>

[24] Java Core Reflection,
<http://java.sun.com/j2se/1.3/docs/guide/reflection>

[25] Fowler, M.: UML konzentriert, 1. Auflage, Addison Wesley Verlag GmbH, 1998.

[26] Hibernate: Relational Persistence For Idiomatic Java,
<http://hibernate.sf.net>

[27] Lieberherr, Karl. J.; Holland, I.: Assuring good style for object-oriented programs, IEEE Software, September 1989, pp 38-48.

[28] Springgay D.; Li, J.; Jones, J.; Adams, G.:
Eclipse User Interface Guidelines,
<http://www.eclipse.org/articles/Article-UI-Guidelines/Index.html>

[29] Object Technology International, Inc.:
Eclipse Platform Technical Overview, Februar 2003,
<http://www.eclipse.org/whitepapers/eclipse-overview.pdf>

[30] Des Rivieres, J.: Evolving Java-based APIs, Juni 2001,
<http://eclipse.org/eclipse/development/java-api-evolution.html>

[31] Sedgewick, R.: Algorithms in C, Part 5: Graph Algorithms, 3rd edition, Addison-Wesley Pub Co, 2001.

[32] AT&T Labs Research: Graphviz – Open Source Graph Drawing Software,
<http://www.research.att.com/sw/tools/graphviz/>

[33] Gansner, E.; Koutsofios, E.; North, S.; Vo, K.-P.: A Technique for Drawing Directed Graphs,
<http://www.research.att.com/sw/tools/graphviz/TSE93.pdf>

Anhang A

Entwurfsdiagramme von Classdep

Auf den folgenden Seiten befinden sich ausgewählte Paketdiagramme, die den Ist-Zustand der Implementierung von Classdep zum Abschluß dieser Diplomarbeit darstellen. Ihr einfaches Aussehen sollte zum Nachweis dienen, dass man mit Hilfe von Paketen die interne Struktur eines Java-Programms tatsächlich verständlicher machen kann. Die Paketdiagramme in dieser Form ersetzen nicht die anderen Kommunikationswege zum Vermitteln der Projektorganisation. Sie könnten dennoch als Leitfaden für die Einführung eines neuen Entwicklers oder als Grundlage einer Arbeitsteilung verwendet werden.

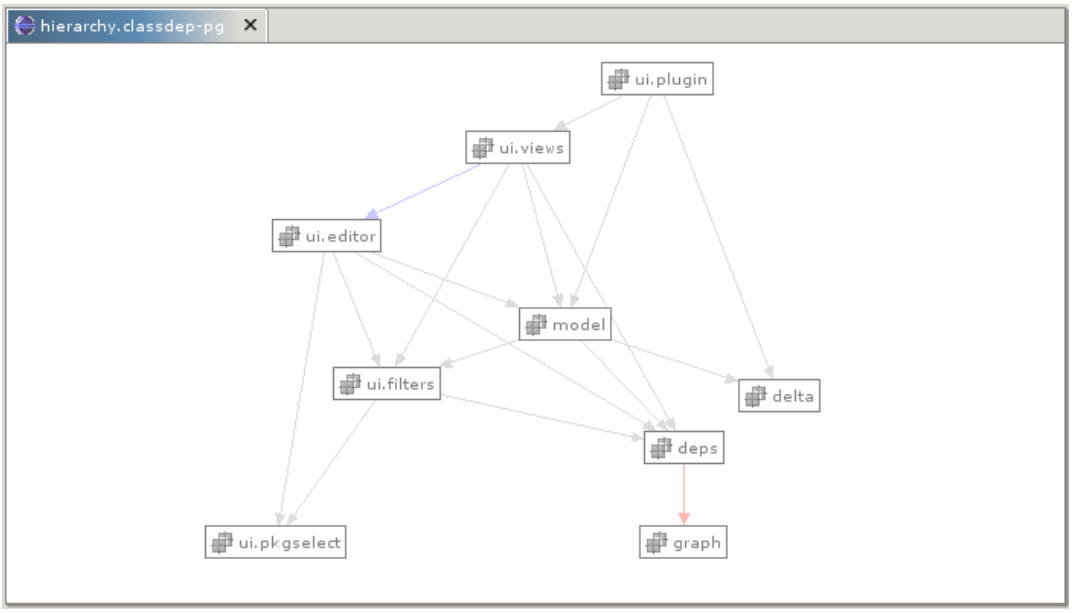

Abbildung A.1: Modulabhängigkeiten in Classdep

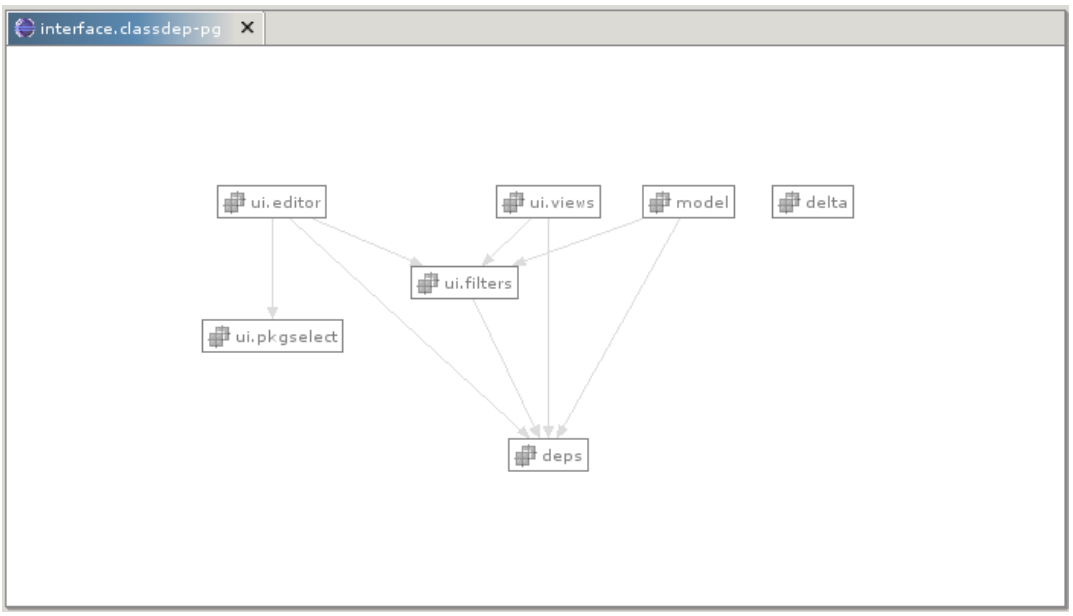

Abbildung A.2: Abhängigkeiten zwischen Schnittstellenpaketen in Classdep

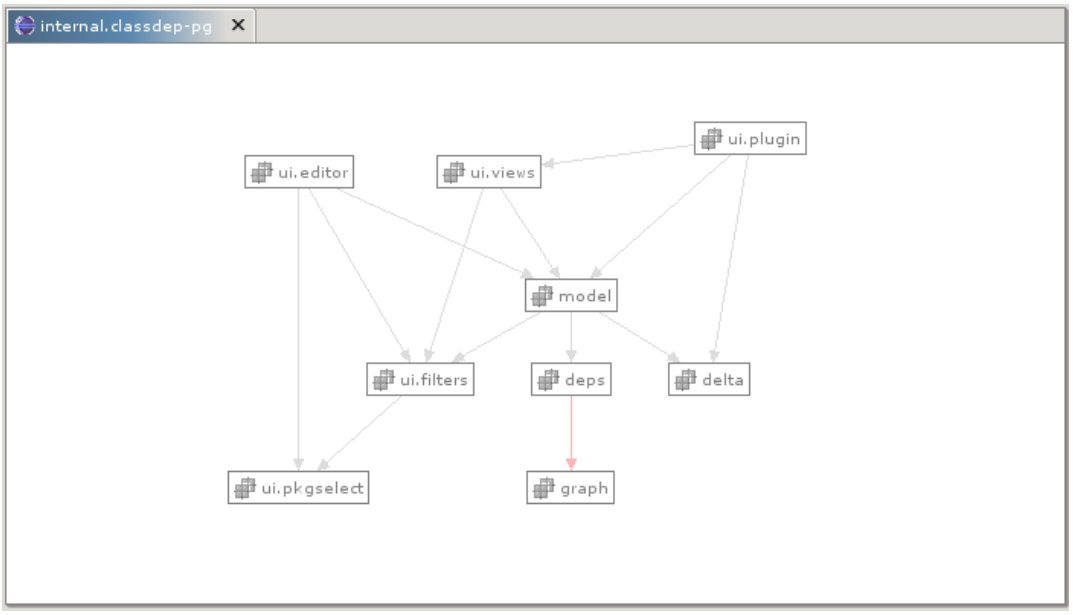

Abbildung A.3: Abhängigkeiten zwischen Implementierungspaketen in Classdep; Objekterzeugung und Zugriffe auf das `model`-Singleton

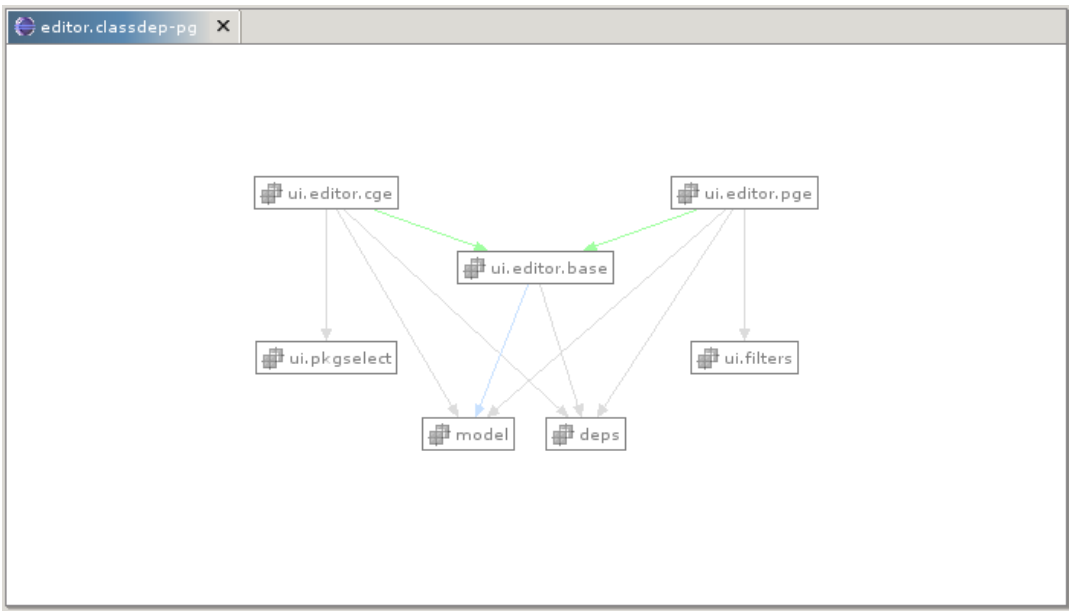

Abbildung A.4: Diagrammeditormodule im Überblick; Inhalt der Paketgruppe `ui.editor` aus Abbildung A.1

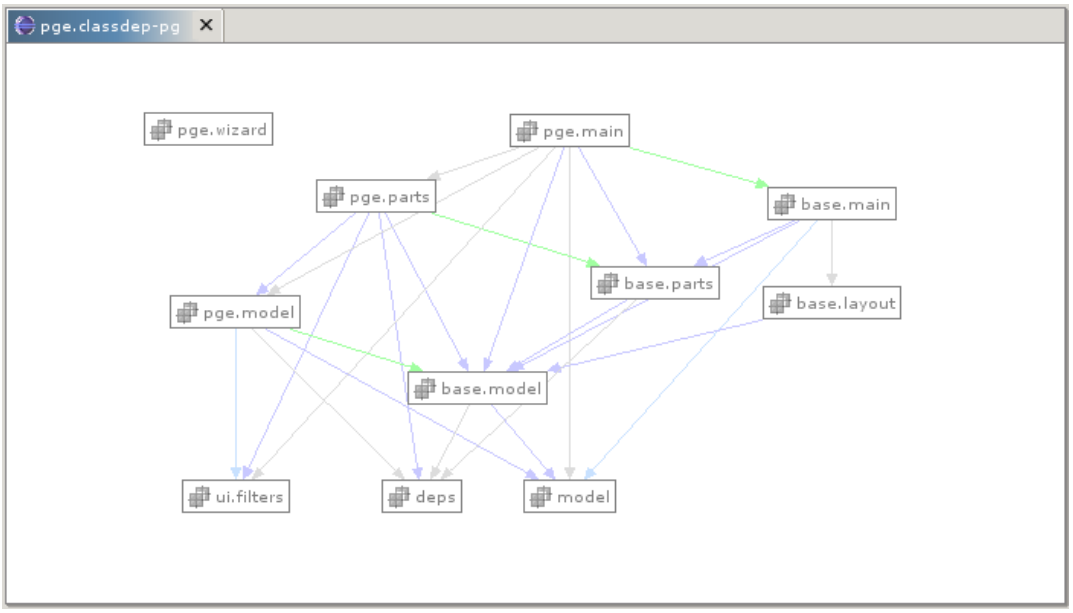

Abbildung A.5: Modulabhängigkeiten des Paketdiagrammeditors; Inhalt der Paketgruppe `ui.editors.pge` aus Abbildung A.4

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Gummersbach, 6. August 2003