Refactoring in Langzeitprojekten — Fallstudie

Elena Kolodizki, Markus Pizka
Technische Universität München
Institut für Informatik
Boltzmannstr. 3
85748 Garching

Martin Ober
msg systems ag
Robert-Bürkle-Straße 1
85737 Ismaning

Report: ViSEK/039/D
Version: 1.0
Klassifikation: extern

Garching, 24.10.03

Abstract

In vielen lange laufenden Software-Projekten verschlechtert sich das anfänglich gute Design zunehmend aufgrund unter Zeit- und Kostendruck durchgeführter Erweiterungen und Änderungen. Die Qualität des Quellcodes sinkt aufgrund von Redundanzen und Inkonsistenzen. Refactoring bietet durch semantik-erhaltende Trannsformationen des Codes die Möglichkeit vorhandene Programm-Strukturen zu verbessern ohne, dass die Software von Grund auf neu geschrieben werden muss. Die Fallstudie untersucht die Vor- und Nachteile dieser Technik im Hinblick auf Langzeitprojekte und gibt Empfehlungen, wie und wo sie am effzientesten angewandt werden kann.

Im ersten Teil der Arbeit wird neben einer Einführung in das Themengebiet Refactoringëin Einblick in diverse Techniken, wie beispielsweise Metrikenerhebung und -beurteilung, gegeben, mit deren Hilfe man auf eventuelle Schwächen im Software-Design hingewiesen wird. Ferner wird dort das Ergebnis der Evaluation sowohl einiger Werkzeuge, die Entwickler bei der Code-Analyse unterstützen können, als auch spezieller Tools, die bei der Durchführung des Refactorings behilflich sind, prasentiert.

Der zweite Teil bildet den Schwerpunkt dieser Arbeit und widmet sich der Betrachtung von Fallbeispielen aus dem Code des untersuchten Projektes REX. Anhand dieser Beispiele werden die durch den Einsatz von Refactoring hervorgerufenen Änderungen kritisch bewertet. Dabei werden nicht nur subjektive Aspekte, wie zum Beispiel Übersichtlichkeit oder Wartbarkeit des Quellcodes, sondern auch Metriken- und Performanzänderungen unter die Lupe genommen. Ausgehend von den so gewonnenen Erkenntnissen werden Empfehlungen bezüglich des Einsatzes von Refactoring in Langzeitprojekten im Allgemeinen und in REX im Besonderen ausgearbeitet.

Weitere Vorgehensweisen zur Verbesserung der Code-Struktur von Langzeitprojekten werden im dritten Teil vorgestellt, wobei nicht nur die Vorzüge und Gefahren der jeweiligen Technik präsentiert, sondern auch Wege zu ihrer gelungenen Einführung in bereits laufenden Projekten aufgezeigt werden. Abschließend wird eine Übersicht über derzeit noch offene Fragen auf dem Gebiet der Code-Umstrukturierungen gegeben. Es werden weiter Überlegungen angestellt, welche Grenzen Refactoring-Werkzeugen generell gesetzt sind, und welche Funktionalitäten ihre Hersteller noch ausbauen müssen, um die Akzeptanz bei Entwicklern zu erhöhen.

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Ausarbeitung entstand im Rahmen einer an der Technischen Universität München in Zusammenarbeit mit der msg systems AG durchgeführten Diplomarbeit mit dem Titel ,,Einsatz von Refactoring in Langzeitprojekten - Fallstudie".

In diesem Kapitel wird eine thematische Einordnung der Arbeit gegeben. Die Ziele der Arbeit werden in Abschnitt 1.2 erläutert. Es folgt eine kurze Einführung in das zu untersuchende Projekt REX. In Abschnitt 1.4 wird die Struktur dieser Ausarbeitung vorgestellt. Hinweise zu typographischen Konventionen findet man schließlich in Abschnitt 1.5.

1.1 Motivation

Eines der Merkmale, die der Begriff ,,Software-Qualität" nach DIN ISO 9126 (vergleiche [DIN_ISO_9126]) umfasst, ist die **Änderbarkeit**. Dabei wird der für die Durchführung der Änderungen benötigte Aufwand beurteilt, der umso höher ist, je schwieriger es ist, den Quellcode zu ändern. Insbesondere bei IT-Projekten, die über Jahre andauern, besteht die Gefahr, dass sich auch ein anfänglich gutes Design durch ständige Erweiterungen der Funktionalität, die zum Teil von verschiedenen Entwicklern unter Zeitdruck durchgeführt werden, ständig verschlechtert, so dass der Code zum Schluss teilweise unübersichtlich und schwer änderbar wird. Dies führt zunehmend dazu, dass eine Code-Änderung an einer Stelle folgenschwere Fehler an einer anderen Stelle verursacht. In diesem Fall steht die Projektleitung vor einem Dilemma: soll man das System komplett neu schreiben (Reengineering, vergleiche Abschnitt 3.2.1.1) oder kann man die entstandenen Probleme eventuell durch sukzessive Code-Verbesserungen lösen? Die erste Alternative würde nicht nur enorme Kosten zur Folge haben, sondern in manchen Situationen auch überhaupt nicht in Frage kommen, da man zum Beispiel die im Rahmen eines Wartungsvertrags zu pflegende Software nicht ohne weiteres durch ein komplett neues Produkt austauschen kann. Mit Hilfe der zweiten Vorgehensweise, zu der auch Refactoring gehört, kann man bestehenden Code Stück für Stück verbessern, was aus folgenden Gründen nicht nur kostengünstiger, sondern auch effektiver ist:

- Das System bleibt während der ganzen Zeit, in der die notwendigen Umstellungen und Umstrukturierungen durchgeführt werden, voll funktionsfähig. So können bei Bedarf Erweiterungen und Änderungen der Code-Bereinigung vorgeschoben werden.
- Jede komplette Neuentwicklung beinhaltet besonders am Anfang viele Bugs, die erst im Laufe des Produktionseinsatzes entdeckt werden. Von den Anwendern eines Programms kann aber kein Verständnis dafür erwartet werden, dass solche Funktionen, die seit einiger Zeit schon fehlerfrei gelaufen sind, auf einmal fehlerbehaftet sind. Insofern würde es sehr schwierig sein, nicht nur das Management, sondern auch die Benutzer dafür zu

begeistern, dass die Anwendung wegen der nachlassenden Code-Qualität von Grund auf neu geschrieben werden muss.

1.2 Ziele

Diese Arbeit beschäftigt sich mit der Untersuchung des Einsatzes von Refactoring-Techniken in Projekten, die länger als ein Jahr andauern. Dabei sollen im Einzelnen folgende Schritte durchgeführt werden:

- Methoden und Werkzeuge zur Analyse von Programmen sollen vorgestellt und evaluiert werden.
- Eine kurze Einführung in das Themengebiet ,,Refactoring" soll gegeben werden.
- Refactoring-Werkzeuge sollen sowohl im Hinblick auf die Palette der von ihnen unterstützten Refactoring-Techniken als auch bezüglich ihrer Handhabung untersucht werden.
- Anhand des Codes von Projekt REX soll eine beispielhafte Beurteilung der kurz- und langfristigen Vorteile, aber auch der Nachteile von Refactoring, insbesondere hinsichtlich der entstehenden Kosten und der gesamten Organisation des Refactorings im betroffenen Projekt, stattfinden.
- Empfehlungen zur weiteren Vorgehensweise in REX in Bezug auf einen möglichen Einsatz von Refactoring sowie generelle Verbesserungsvorschläge sollen gegeben werden.
- Ein Maßnahmen-Katalog zur Verbesserung der Code-Struktur in Langzeitprojekten soll erarbeitet werden; dabei soll nicht nur der Einsatz von Refactoring kritisch beurteilt, sondern es sollen auch andere Techniken vorgestellt werden, die dem Projektablauf zugute kommen können.

1.3 Vorgeschichte des Projektes REX

Bei REX[1] handelt es sich um ein mittelgrosses, kommerzielles Software-Projekt, dass über einen Zeitraum von mehr als 3 Jahren durchgeführt wurde. An dem abgeschlossenen Software-Projekt werden im Rahmen der Wartung langfristig weiterhin Anpassungen an veränderte Anforderungen vorgenommen werden.

Abbildung 1.1 gibt einen groben Überblick über die REX-Architektur. Die Implementierung basiert auf der von Sun entwickelten J2EE-Plattform. Wie in zahlreichen anderen moderen web-basierten Systemen kommunizieren Java Swing Client-Front-Ends mit einem Applikations-Server (BEA WebLogic Server, siehe [www_bea]), der über TopLink ([www_toplink]) auf die zentral Datenbank zugreift.

Erste Schätzungen haben ergeben, dass die vom Auftraggeber gewünschte Funktionalität innerhalb eines Jahres zur Verfügung gestellt werden kann. Im Laufe der Software-Entwicklung zeigte sich, dass der Zeitplan nicht einhalten konnte, und dass einige Komponenten nicht wie gewünscht funktionierten. Die Ursachen sind vielfältig:

[1] anonymisierter Name; REX steht für Refactoring Example

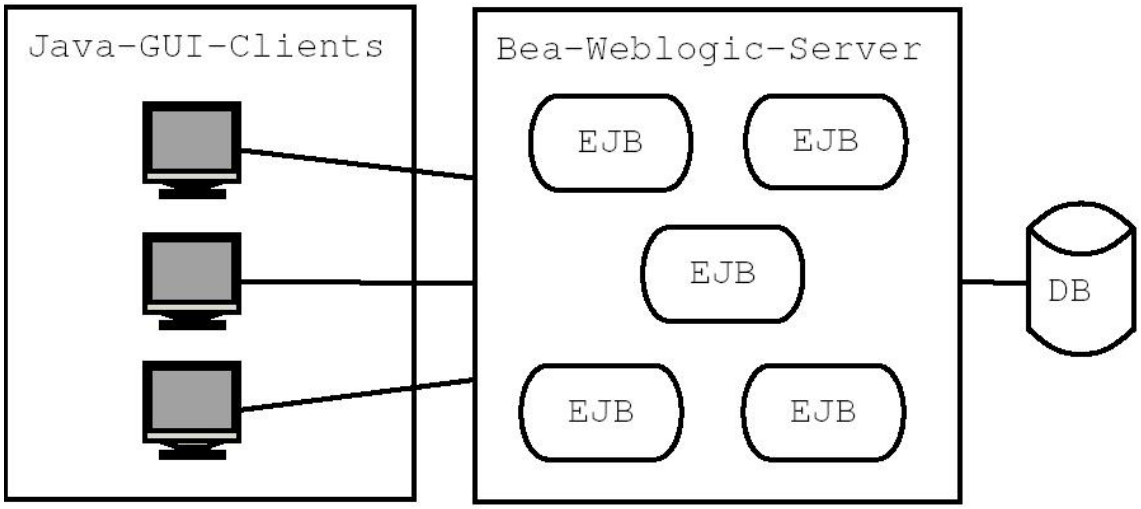

Abbildung 1.1: Architektur von REX

- Die Spezifikation der zu erstellenden Software war nicht detailliert genug und ließ zu viel Interpretationsraum; einige wichtige Funktionalitäten haben in der Spezifikation gefehlt.
- Die fachlichen Anforderungen waren lange Zeit unklar; die nachträgliche Detaillierung der Kundenwünsche zog zum Teil umfangreiche Änderungen nach sich.
- Die Komplexität der zu implementierenden Funktionalitäten war nicht sofort zu erkennen und wurde deshalb deutlich unterschätzt.
- Die Beschreibung der zur Verfügung zu stellenden Funktionalitäten war über mehrere Dokumente verteilt, so dass es nicht möglich war, alle Anforderungen bezüglich einer zu implementierenden Komponente auf einmal zu sehen, was letztendlich dazu führte, dass bei der Realisierung vieles vergessen wurde.
- Die erste Aufwandsschätzung war zu optimistisch.
- Es gab kein einheitliches Design, was zu Redundanzen führte.
- Die internen Tests fanden zu spät statt.
- Der Ausfall zweier Projektbeteiligter hatte Auswirkungen auf die Implementierung.

Während die meisten Probleme gelöst werden konnten, erscheint das Design der Anwendung nach wie vor uneinheitlich und unübersichtlich, was dazu führt, dass:

- das Hinzufügen neuer Funktionalität zu viel Zeit benötigt, da man Änderungen an sehr vielen Stellen statt nur an einer Stelle durchführen muss.
- Teilweise werden durch Beseitigung eines Fehlers neue Fehler eingebaut (so genannte „Bad Fixes").
- Da zum Teil mehrfacher Code vorhanden ist, wird leicht vergessen, die gewünschte Änderung an allen Stellen durchzuführen.

1.4 Gliederung der Studie

In Kapitel 2 werden einige Techniken und Werkzeuge vorgestellt, mit deren Hilfe man ein Projekt in Bezug auf seine Schwachstellen untersuchen kann, um Hinweise darauf zu bekommen, welche Code-Stellen besondere Aufmerksamkeit und eine eventuelle Überarbeitung erfordern. Eine Einführung in das Themengebiet „Refactoring" folgt in Kapitel 3, Werkzeuge zu seiner Unterstützung werden in Kapitel 4 vorgestellt. Eine Untersuchung der Auswirkungen der im Projekt REX durchgeführten Refactoring-Maßnahmen in Hinsicht auf ihre Vor- und Nachteile findet in Kapitel 5 statt. In Kapitel 6 werden die Ergebnisse der Arbeit zusammengefasst und abschließend beurteilt, auch findet man dort einen Ausblick auf noch anstehende Untersuchungen sowie Entwicklungen von Vorgehensweisen im Zusammenhang mit Verbesserung der Code-Struktur in Langzeitprojekten.

1.5 Typographische Konventionen

Um den Lesefluss nicht unnötig zu behindern, wurde darauf verzichtet, alle Vorkommnisse der urheberrechtlich geschützten Namen - wie beispielsweise Java, J2EE, Sun, Borland, TogetherSoft, Together ControlCenter, Small Worlds, CloneDR - entsprechend zu kennzeichnen.

Des Weiteren wurden zur Erleichterung der Lesbarkeit folgende Vereinbarungen getroffen:

- Alle Klassen- und Methodennamen sowie nicht sofort als solche erkennbare Bezeichnungen von Refactoring-Techniken werden *kursiv* dargestellt.
- Quellcode und sonstige Befehle werden in `Schrift mit fester Breite` gedruckt.
- **Fette Schrift** weist auf besonders wichtige Details hin.

2 Analyse von Java-Programmen

Dieses Kapitel beschreibt, wie man in Java geschriebenen Quellcode analysieren kann, um seine Schwachstellen aufzudecken. Möglichkeiten zur Lösung der so gefundenen Probleme werden in Kapitel 3 vorgestellt.

In Abschnitt 2.1 werden etablierte Messmethoden behandelt, mit deren Hilfe Ungereimtheiten und grobe Verstöße gegen geltende Konzepte beim objektorientierten Programmieren aufgespürt werden können. Die Mindestanforderungen an ein Mess- und Analysewerkzeug werden in Abschnitt 2.2 vorgestellt. Im darauf folgenden Abschnitt werden einige Werkzeuge vorgestellt, die die Analyse von Quellcode beziehungsweise die Erstellung der Metriken unterstützen. Die während der Analyse des Projektes REX erhaltenen Ergebnisse findet man in Abschnitt 2.4. Eine zusammenfassende Bewertung der Ergebnisse, die man durch die Analyse von Java-Programmen erreichen kann, findet in Abschnitt 2.5 statt.

2.1 Metriken

Während quantitative Aussagen über die Produkte traditioneller Industriezweige gang und gäbe sind, gehören Messungen im Bereich der Software-Entwicklung immer noch nicht zum Alltag vieler Unternehmen. Die Forschung im Bereich der Software-Metriken wird deshalb durch die Tatsache erschwert, dass nur wenige Vergleichsdaten vorliegen.

Wenn man sich dazu entscheidet, Messungen an Software durchzuführen, ist es sehr wichtig, das Ziel im Auge zu behalten, das man mit Hilfe von Metriken erreichen will, um nicht unnötige Daten zu ermitteln, aber auch um den Überblick zu behalten. Die Hauptaufgabe dieser Arbeit besteht darin, festzustellen, ob und wenn ja, wo und wie man ein Programm durch Anwendung von Refactoring-Techniken (siehe Abschnitt 3.1) umstrukturieren soll. Metriken können Unterstützung bei der Beantwortung dieser Fragen bieten, indem beispielsweise die Überschreitung bestimmter vorher festgelegter Grenzwerte festgestellt wird. Auch kann man anhand der Metriken die Ergebnisse von Refactoring greifbar machen beziehungsweise beurteilen.

Im Folgenden werden die zur Bewerkstelligung dieser Aufgaben und für die objektorientierte Programmierung geeigneten beziehungsweise wichtigen Metriken vorgestellt.

2.1.1 Lines of Code (LOC)

Diese die Anzahl der Zeilen an Code angebende Metrik kann trotz ihrer vermeintlichen Trivialität auf mindestens drei verschiedene Arten und Weisen erfasst werden:

1. Die Gesamtzahl der Zeilen an Programmcode inklusive Kommentare und Leerzeilen;

2. Anzahl der Zeilen mit ausführbarem Code;
3. Anzahl der Zeilen, die voneinander durch den Begrenzer (*engl. delimiter*, in Java ';') getrennt sind.

Meistens liefern die Werkzeuge zur Metriken-Erstellung sowohl die Gesamtzahl der Zeilen als auch die Anzahl der Zeilen mit ausführbarem Code; oft wird auch die Anzahl der Kommentarzeilen ermittelt. Im Folgenden wird LOC wie in Punkt 2 beschrieben ermittelt.

LOC hat sowohl Vor- als auch Nachteile:
Vorteile:

- Man kann sich mit Hilfe von LOC schnell einen Überblick über die Größe des Programms verschaffen.
- Die im Code durchgeführten Änderungen können quantitativ verfolgt werden.

Nachteile:

- Anhand des Wertes kann man keine Aussagen bezüglich der Qualität des Codes machen.
- Es gibt kein vorgegebenes Intervall, in dem sich der Wert bewegen sollte.

2.1.2 Number of Import Statements (NOIS)

Die Auskunft über die Anzahl der importierten Klassen und/oder Packages (das heißt der durch Delimiter getrennten Code-Stellen, in denen das Wort „import" vorkommt), kann man mit Hilfe dieser Metrik erhalten. Beispielsweise hat folgender Code-Abschnitt:

```
import java.util.*;
import java.io.File;
```

den NOIS-Wert 2. Wenn eine Klasse oder Package mehrmals durch identische Import-Anweisung importiert wird, wird sie genauso oft mitgezählt. Falls eine Klasse erst durch den vollständigen Namen (Package- und Klassenname) im Code auftritt, wird sie bei der Ermittlung von NOIS nicht mit berücksichtigt.

Ein zu hoher Wert dieser Metrik kann ein Hinweis auf eine hohe Abhängigkeit von anderen Klassen sein (gesetzt den Fall, dass die importieren Klassen und Packages in dieser Klasse wirklich benötigt werden und nicht nach Änderungen vergessene Überbleibsel sind). Das führt seinerseits dazu, dass die in diesen Klassen durchgeführten Änderungen diese Klasse dahingehend beeinflussen könnten, dass sie auch mit angepasst werden müsste.

2.1.3 Number of Attributes (NOA)

Diese Metrik gibt die Anzahl der Attribute einer Klasse an, wobei diese aus der Summe

- aller Instanzvariablen (Number of Instance Variables (NIV)) und
- der Summe aller Klassenvariablen (Number of Class Variables (NCV))

gebildet wird:

$$\mathbf{NOA} = \mathbf{NIV} + \mathbf{NCV}$$

2.1.4 Number of Methods (NOM)

Die Anzahl der Methoden einer Klasse kann mit Hilfe dieser Metrik ermittelt werden.

$$\mathbf{NOM} = \mathbf{NEM} + \mathbf{NHM}, \text{ wobei}$$

- NEM (Number of External Methods) die Anzahl der öffentlichen Methoden (also *public*) und
- NHM (Number of Hidden Methods) die Anzahl der geschützten und privaten Methoden (das heißt *protected* oder *private*)

ist.

Man sollte versuchen, die Schnittstelle einer Klasse übersichtlich zu gestalten (das heißt, dass die Anzahl der als *public* deklarierten Methoden gering sein sollte). Ein zu hoher NOM-Wert ist ein Indiz dafür, dass eine Klasse zu viele Verantwortlichkeiten hat und durch Aufteilung in mehrere in sich geschlossene Klassen vereinfacht werden sollte.

2.1.5 Cyclomatic Complexity (CC)

Mit Hilfe der zyklomatischen Komplexität einer Klasse zählt man die Anzahl der möglichen Pfade des Flussgraphes, indem man seine Verzweigungen, also die Anzahl der *if*-, *for*-, und *while*-Klauseln, ermittelt. Diese Metrik wurde von McCabe (vergleiche [McCabe76]) eingeführt und folgendermaßen definiert:

$$\mathbf{CC} = \mathbf{L} - \mathbf{N} + 2 * \mathbf{P}, \text{ wobei}$$

- L die Anzahl der Verzweigungen im Kontrollflussgraph,
- N die Anzahl der Knoten dieses Graphes und
- P die Anzahl der miteinander unverbundenen Teile des Graphes

ist. Manchmal werden bei der Ermittlung dieser Metrik noch switch-Statements mit berücksichtigt. Die in dieser Arbeit vorkommenden Werte der CC-Metrik wurden ohne Rücksicht auf sie erhoben, so wie es der Erfinder dieser Metrik auch empfohlen hat (vergleiche [WatMcC96]).

Mit steigendem Wert dieser Metrik verschlechtert sich die Wartbarkeit des Codes (siehe [EbeDum96]). Watson empfiehlt in [WatMcC96] den CC-Wert 15 beim Einsatz von erfahrenen Entwicklern und unter Hinnahme eines erhöhten Testaufwands, ansonsten sollte die Grenze von 10 nicht überschritten werden. SEI[1] schlägt dagegen eine differenziertere Beurteilung wie in Tabelle 2.1 vor (vergleiche [www_sei_cc]):

Die im Folgenden aufgeführten CC-Werte beinhalten auch die Anzahl der Methoden der Klasse (NOM, vergleiche Abschnitt 2.1.4), weil die in den Abschnitten 2.3.2 und 2.3.3 vorgestellten Werkzeuge diese Zahl bei der Ermittlung der CC-Metrik berücksichtigen.

[1] Software Engineering Institute

CC	Risikobeurteilung
$1-10$	einfaches Programm, ohne viel Risiko
$11-20$	komplexer, mäßiges Risiko
$20-50$	komplex, hohes Risiko
>50	unmöglich zu testendes Programm (sehr hohes Risiko)

Tabelle 2.1: CC-Metrik-Grenzwerte

2.1.6 Response for a Class (RFC)

Diese Metrik gibt die Anzahl der Methoden einer Klasse (NOM, siehe Abschnitt 2.1.4) zuzüglich der Anzahl der unterschiedlichen Methoden an, die von dieser Klasse aufgerufen werden; dabei ist es irrelevant, ob all diese Methoden wirklich genutzt werden oder nicht. Kahlbrandt gibt in [Kahlbr01] folgende formale Definition:

$$\mathbf{RFC} = |M| \cup \bigcup_{i=1}^{n} |R_i|, \text{ wobei}$$

- n die Anzahl der Methoden der Klasse,
- $M = \{m_1,..,m_n\}$ die Menge der Methoden der Klasse,
- R die Menge der Methoden, die m_i aufruft,
- $|X|$ die Kardinalität der Menge X ist.

Die in den Abschnitten 2.3.2 und 2.3.3 vorgestellten Werkzeuge betrachten M als die Menge aller Methoden einer Klasse inklusive der vererbten Methoden, entsprechend ist n die Anzahl aller in einer Klasse verfügbaren Methoden. Daher beziehen sich die im Folgenden aufgeführten Werte der RFC-Metrik auf diese erweiterte Definition.

Die Komplexität einer Klasse ist umso geringer, je kleiner der Wert von RFC ist. Bei großen Programmen sollte RFC den Empfehlungen des SATC[2] zufolge den Wert 100 nicht überschreiten.

2.1.7 Coupling between Objects (CBO)

CBO gibt - wie der Name schon sagt - Aufschluss darüber, wie stark die untersuchte Klasse von anderen nicht in ihrer Vererbungshierarchie enthaltenen Klassen abhängt. Der CBO-Wert setzt sich aus der Anzahl der unterschiedlichen Typen zusammen, die in

- Deklarationen von Attributen,
- Rückgabewerten von Methoden,
- Methodenparametern,
- lokalen Variablen,
- *throws*-Klauseln der Methoden

[2]Software Assurance Technology Center (siehe [www_satc])

vorkommen sowie auf die ,,gecastet" wird. Dabei werden einfache Typen (wie beispielsweise *int*, *long* und *boolean*) sowie Typen aus dem *java.lang*-Package nicht berücksichtigt.

Je größer der Wert, desto empfindlicher ist die Klasse für die in anderen Klassen durchgeführten Änderungen. Ein hoher CBO-Wert bedeutet auch hohe Komplexität der Klasse, was seinerseits einen schwer nachvollziehbaren Code bedeutet. Laut [www_oom_empflng] sollte man einen CBO kleiner gleich fünf anstreben.

2.1.8 Lack of Cohesion in Methods (LCOM)

Kohäsionsmangel ist eine Metrik, mit deren Hilfe man die Kapselung beurteilen kann. Laut Rosenberg (siehe [RosHya97]) existieren mindestens zwei verschiedene Verfahren zur Ermittlung von LCOM:

LCOM1: Man berechnet die Anzahl der Methoden, die unterschiedliche Mengen von Attributen benutzen beziehungsweise verändern. Dabei geht man folgenderweise vor:

1. Für je zwei Methoden betrachtet man die von ihnen benötigten Attribute;
2. Falls die jeweiligen Attributsätze disjunkte Mengen bilden, wird P um 1 erhöht;
3. Im Falle, dass wenigstens eines der Attribute von den beiden Methoden verwendet wird, erhöht man Q um 1.

$$\textbf{LCOM1} = (P > Q)\ ?\ (P - Q) : 0$$

LCOM2: Für jedes Attribut der Klasse wird der Prozentsatz der Methoden ermittelt, die es nutzen oder verändern. Danach bildet man den Durchschnitt der errechneten Werte und substrahiert ihn von 100%. Ein hoher Wert deutet auf eine schlechte Kapselung hin.

Kohäsionsmangel dient als ein Indiz dafür, dass die Klasse in mehrere Klassen aufgespaltet werden sollte.

2.1.9 Schlussbemerkung

Die hier aufgelisteten Metriken sind bei weitem nicht die Einzigen, die die Anwendung von Refactoring-Techniken quantitativ beleuchten beziehungsweise Aufschluss über die Notwendigkeit von Refactoring geben können. Allerdings werden sie in der Literatur am häufigsten erwähnt, wenn es um Qualitätsmessungen geht.

Eine ausführliche Beschreibung von diesen und anderen Metriken findet man in den im Literaturverzeichnis unter der Rubrik ,,Metriken" aufgelisteten Quellen, wobei die Bücher von Ebert und Dumke (vergleiche [EbeDum96]) sowie von Thaller (siehe [Thalle00]) jeweils als mehr oder minder ausführliche Einführungen in das Themengebiet ,,Metriken" dienen, die Arbeiten von Rosenberg et al. (vergleiche [RosHya97], [RosHya98], [RoStGa99]) dagegen explizit OO-Metriken als ein Instrument zur Verbesserung von Software-Qualität behandeln. Watson zusammen mit dem Erfinder der CC-Metrik McCabe (siehe [WatMcC96]) behandeln diese als Instrument zum strukturierten Testen von Programmen.

2.2 Anforderungen an Mess- und Analysewerkzeuge

Spezielle Programme können die Erfassung von Metriken und die Code-Analyse automatisieren. Nachfolgend werden die Anforderungen an solche Tools nach deren Wichtigkeit absteigend sortiert zusammengestellt:

1. Ermittlung der in den Abschnitten 2.1.1 - 2.1.8 erläuterten Metriken, um eventuelle Kandidaten für Refactoring zu finden;

2. benutzerdefinierte Auswahl der zu analysierenden Klassen, aber auch der zu erfassenden Metriken, damit man schnell die Veränderungen einer bestimmten Klasse beispielsweise nach ihrer Umstrukturierung untersuchen kann;

3. Export der erfassten Metriken, der eine weitere Bearbeitung beziehungsweise Analyse der Daten erlaubt; dabei sind zum Beispiel folgende Szenarien denkbar:

   - Falls Metriken den Quellcode des gesamten Projektes umfassen, sollen die erhobenen Daten an die jeweils zuständigen Entwickler weitergeleitet werden, damit diese entsprechende Maßnahmen zur Beseitigung der festgestellten Schwächen im Code ergreifen können;
   - Falls eine regelmäßige Metrikenerhebung ein integrierter Teil des Entwicklungsprozesses ist und von jedem Entwickler eingesetzt wird, sollen die erfassten Daten zu Dokumentations- und Planungszwecken verwendet werden;

4. Angabe von Obergrenzen für die Metriken, so dass man auf ihre Überschreitung hingewiesen wird (zum Beispiel durch rote Farbe).

5. Hinweise auf Probleme (wie zum Beispiel überflüssige Abhängigkeiten zwischen einzelnen Klassen, oder leere *catch*-Blöcke) im Code;

6. graphische Darstellung der Abhängigkeiten zwischen den einzelnen Klassen (beispielsweise in Form von UML-Diagrammen) sowie Diagramm-Export in einem üblichen Format (zum Beispiel als EPS-Datei), um das Design von Komponenten veranschaulichen und dokumentieren zu können;

7. Stichwortsuche im betreffenden Quellcode einer Klasse, damit man die seitens des Tools gefundenen Probleme beurteilen kann;

8. schnelle Analyse des Quellcodes oder keine spürbaren Behinderungen bei der Arbeit mit anderen Anwendungen, was eine effiziente Arbeit mit dem Werkzeug garantiert; dies ist besonders dann unabdingbar, wenn Code-Analyse während des Entwicklungsprozesses von jedem Entwickler zur Verbesserung der Qualität des von ihm produzierten Codes eingesetzt wird;

9. leicht erlernbare Bedienung des Werkzeugs, so dass die Entwickler die Analyse ohne Umwege erledigen können;

10. ausführliche Benutzerdokumentation, die erklärt, wie man das Tool sinnvoll einsetzen kann und die bei der Lösung eventuell entstandener Probleme hilft;

2.3 Werkzeuge zur Erstellung von Metriken und Analyse von Java-Programmen

Nachfolgend werden einige ausgewählte Werkzeuge zur Analyse von Java-Programmen vorgestellt und im Bezug auf die im vorangegangenen Abschnitt aufgestellten Anforderungen beurteilt, wobei sich die Reihenfolge der jeweils aufgelisteten Vor- und Nachteile an der des Anforderungskatalogs orientiert. Einen kritischen Vergleich der evaluierten Werkzeuge findet man in Abschnitt 2.3.4.

2.3.1 Small Worlds

Small Worlds wurde von Information Laboratory entwickelt und dient der Analyse von objektorientiertem Code (siehe auch [www_SmallW]).

Hier folgt eine Zusammenfassung der wichtigsten Funktionalitäten des Tools, auch wird auf einige Probleme hingewiesen, mit denen man während des Arbeitens mit dem Werkzeug konfrontiert wird.

Vorteile beziehungsweise vorhandene Funktionalitäten:

- In der vom Werkzeug bei der Analyse generierten Tabelle ,,Local Dependencies" wird für jede Klasse die Anzahl der Klassen, die diese unmittelbar
  - benutzen,
  - beinhalten,
  - erweitern und
  - implementieren

  präsentiert.
- Die Tabelle ,,Global Dependencies" gibt Aufschluss darüber, wie viele Klassen
  - von dieser Klasse lokal abhängen (,,dependencies"),
  - diese Klasse benutzen (,,dependents"),
  - betroffen sind, wenn sich diese Klasse ändert (,,affects") sowie

  wie viele Male diese Klasse maximal betroffen sein würde, wenn andere Klassen geändert würden (,,affected"). Abbildung 2.1 zeigt einen Screenshot des Dialogs.
- Es gibt die Möglichkeit, anzugeben, welche Packages und/oder Klassen von der Analyse ausgeschlossen werden sollen.
- Die Tabellen mit zahlenmäßigen Abhängigkeiten können sortiert und/oder im HTML-Format exportiert werden.
- Ein zusammenfassender Bericht der Analyse erleichtert das Auffinden von Schwachstellen, dabei wird insbesondere auf folgende Unzulänglichkeiten beziehungsweise Gefahrenquellen hingewiesen:

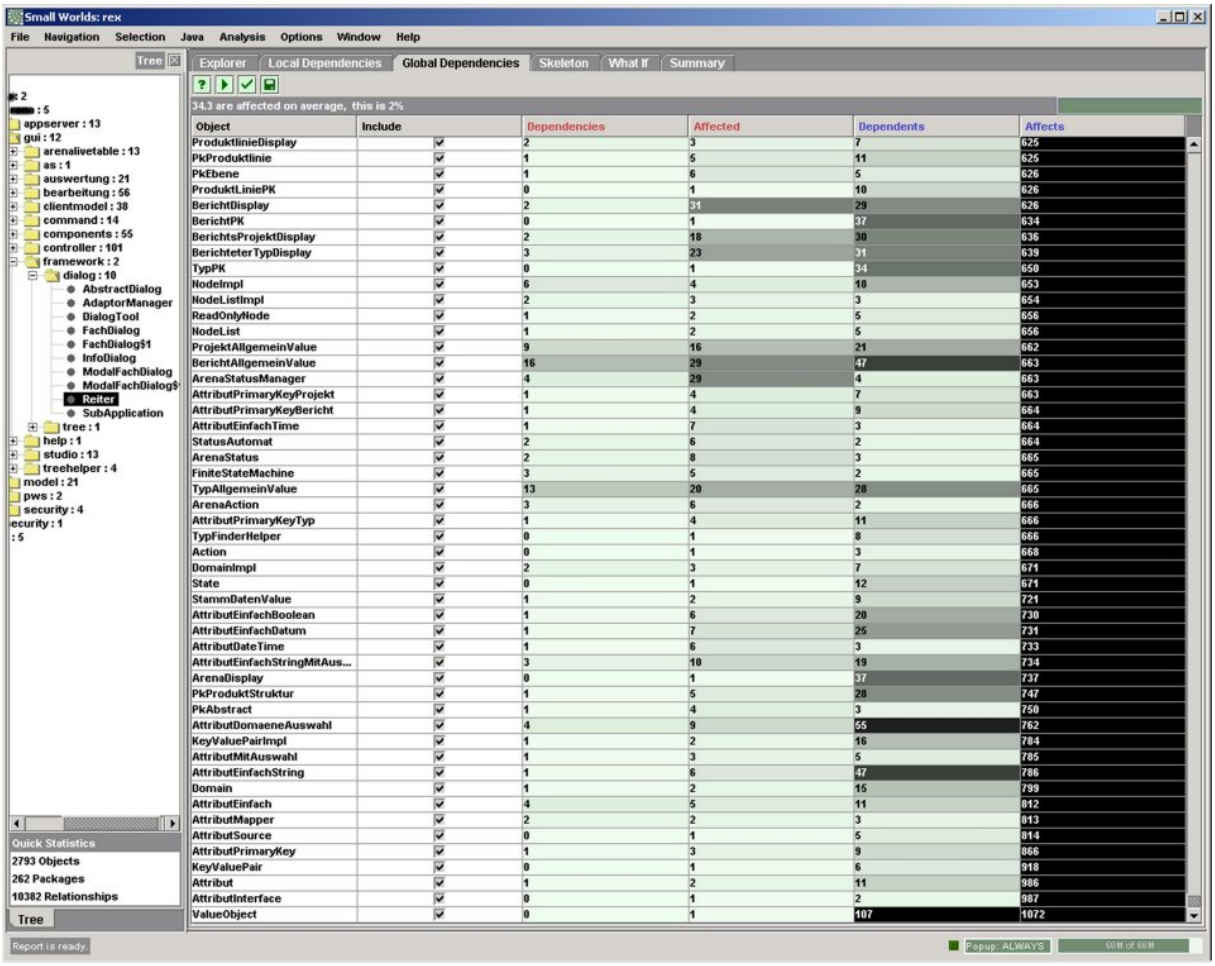

Abbildung 2.1: Small Worlds: Global-Dependencies-Ansicht

Tangle: Eine Aufzählung der Klassen, die voneinander gegenseitig abhängig sind, so dass die Änderung einer der Klassen Einfluss auf alle anderen Klassen des Tangles hat.

Local Butterfly: Klassen beziehungsweise Interfaces, von denen unmittelbar sehr viele andere Klassen oder Interfaces abhängen.

Global Butterfly: Klassen, deren Änderung Anpassungen in vielen anderen Klassen nötig machen kann.

Local Hub: Klassen, die von sehr vielen Klassen abhängig sind und von denen viele Klassen abhängen, das heißt dass diese Klassen zu viele Aufgaben übernehmen.

Local Breakable: Klassen, die unmittelbar von sehr vielen Klassen abhängen.

- Abhängigkeiten zwischen den Klassen beziehungsweise Packages können unter anderem als UML-Diagramme angezeigt werden, die im GIF-Format exportiert werden können.
- Es gibt eine kurze und prägnante Zusammenfassung der Klasseninformationen (wie zum Beispiel Package-Name, API der Klassen, Anzahl der Abhängigkeiten, et cetera).

- Den Quellcode der Klasse kann man sich im nicht editierbaren Modus ansehen, dabei ist der zu benutzende Viewer frei wählbar.
- Die Projektdatei ist im Unterschied zu Understand For Java (siehe Abschnitt 2.3.3) recht klein (bei REX ist sie lediglich 8 KB groß).
- Auffinden von Schwachstellen sowohl in C++- als auch in Java-Code ist möglich.
- Das Programm ist ressourcenschonend, so dass das Arbeiten mit anderen Anwendungen während der laufenden Analyse (die bei REX beispielsweise 20 Minuten andauert) nicht behindert wird.
- Kontextbezogene Hilfe sowie FAQ sind vorhanden.

Nachteile beziehungsweise Probleme:

- Es ist sehr mühsam, Analysen auf einzelne Klassen zu begrenzen, weil man alle anderen Packages und für die Analyse nicht relevanten Klassen aus demselben Package wie die zu untersuchenden Klassen explizit von der Analyse auszuschließen hat.
- Zwar wird bei den Tabellen mit den ermittelten zahlenmäßigen Abhängigkeiten zwischen den Klassen beziehungsweise Interfaces (siehe Abbildung 2.1 auf Seite 17) mit dem jeweils steigenden Wert die Färbung der Tabellenzelle immer dunkler (was als ein Hinweis auf eventuelle Probleme mit der Klasse gedeutet werden kann), es gibt jedoch keine Möglichkeit, die jeweiligen Grenzwerte anzugeben.
- Der eingebaute Editor für den Quellcode hat keine Suchfunktion.
- Die Analyse-Ergebnisse werden nur zum Teil in der Projektdatei abgespeichert, so dass beim Öffnen eines bereits analysierten Projektes der Rest der Analyse wiederholt wird, wodurch man mit Wartezeiten (bis zu fünf Minuten bei REX) rechnen muss.
- Die erst nach dem Programmstart angelegten Verzeichnisse (sei es mit Quellcode oder Archivdateien) können seitens des Tools nicht erkannt werden.
- Es gibt keine Möglichkeit, anzugeben, in welches Verzeichnis die Projektdatei abgelegt werden soll.
- Die Fehlermeldungen sind nicht ausführlich genug, so dass sich die Fehlersuche sehr mühsam gestaltet.
- Eine gezielte Suche nach Stichwörtern in der Online-Hilfe steht nicht zur Verfügung.
- Die Programmbedienung ist teilweise nicht intuitiv, besonders verwirrend ist die Tatsache, dass man keine Speichern-Funktion hat, um das neu angelegte zu analysierende Projekt beziehungsweise Änderungen in den Analyse-Einstellungen zu speichern (in diesen Fällen speichert das Tool alle Änderungen automatisch in die Projektdatei, die im Installationsverzeichnis angelegt wird).

2.3.2 Together ControlCenter

Das ursprünglich von TogetherSoft entwickelte und vor kurzem von Borland übernommene Werkzeug „Together ControlCenter" (siehe [www_together]) kann zur Ermittlung von Metriken und Durchführung von Audits eingesetzt werden.

Die wesentlichen Merkmale und Funktionalitäten kann man den folgenden Aufzählungen entnehmen.

Vorteile beziehungsweise vorhandene Funktionalitäten:

- Alle in den Abschnitten 2.1.1 - 2.1.8 beschriebenen Metriken können erhoben werden.
- Ferner ist es möglich, nicht nur die Metrikenermittlung auf bestimmte Klassen einzugrenzen, sondern auch die zu erfassenden Metriken aus einer Liste auszuwählen.
- Metrikenexport ist sowohl im HTML- als auch im durch Tabulatoren getrennten Text-Format möglich.
- Überschreitungen der vom Benutzer festgelegten Obergrenzen für Metriken werden entsprechend gekennzeichnet.
- Mit Hilfe von Audits wird man auf Schwachstellen im Code aufmerksam gemacht.
- Aus dem Quelltext können UML-Diagramme erstellt und im GIF-, SVG- und WMF-Format exportiert werden.
- Das Werkzeug ist eine vollwertige IDE[3] mit einem eingebauten Quellcode-Editor.
- Die Ermittlung einer begrenzten Anzahl von Metriken einiger weniger Klassen kann schnell durchgeführt werden.
- Die Bedienung des Werkzeugs ist schnell erlernbar.

Nachteile beziehungsweise Probleme:

- Die vom Tool erstellten UML-Diagramme beziehen sich nur auf das jeweils aktuelle Package, Beziehungen zwischen den einzelnen Packages werden nicht dargestellt.
- Eine vollständige Metrikenerhebung kostet sehr viel Zeit und beansprucht einen großen Teil der Systemressourcen (bei REX: mehrere Stunden sowie über 90% der Systemressourcen wie CPU und RAM).

2.3.3 Understand for Java

Mit Hilfe des von Scientific Toolworks, Inc. (STI) entwickelten Werkzeugs mit dem Namen „Understand for Java" (siehe [www_UndFJa]) kann man nicht nur Metriken erstellen, sondern auch durch den Code navigieren.

Im Folgenden werden die Vor- und Nachteile des Tools zusammengefasst.

Vorteile beziehungsweise vorhandene Funktionalitäten:

[3] engl. Integrated Development Environment: integrierte Entwicklungsumgebung

- Die ersten Analyseergebnisse umfassen die Metrik LOC (vergleiche Abschnitt 2.1.1), Aufzählung der Methoden und Attribute, Superklasse sowie Klassen und Methoden, die entweder die jeweils untersuchte Klasse benutzen oder von ihr verwendet werden.
- Metriken wie NIV (siehe Abschnitt 2.1.3), NEM und NHM (vergleiche Abschnitt 2.1.4), CC (siehe Abschnitt 2.1.5), RFC (vergleiche Abschnitt 2.1.6), CBO (siehe Abschnitt 2.1.7), LCOM2 (vergleiche Abschnitt 2.1.8) sowie viele weitere (zum Teil vom Hersteller des Werkzeugs definierte) können ermittelt und im Comma-Separated-Format exportiert werden.
- Es besteht die Möglichkeit, weiterführende Analyse-Berichte generieren zu lassen, in denen zusätzliche Details (wie zum Beispiel nicht benutzte Objekte sowie Methoden und vieles andere mehr) ermittelt werden können. Diese Berichte kann man sowohl im HTML- als auch im Text-Format exportieren.
- Die Abhängigkeiten zwischen den Klassen können graphisch in Form von Bäumen dargestellt werden (siehe Abbildung 2.2 auf Seite 21). Folgende Diagramme sind möglich:

  Call Tree: Abhängigkeiten von den aufgerufenen Klassen,

  CallBy Tree: Abhängigkeiten von den Aufrufer-Klassen,

  Extended Tree: der Vererbungsbaum der Superklasse,

  ExtendedBy Tree: der Vererbungsbaum der Unterklassen.

  Diese können dann im JPEG-, BMP- oder PNG-Format exportiert werden.
- Die Navigation im Quellcode der zu untersuchenden Klasse ist in einem bequemen Editor (nach der Art der meisten gängigen IDEs) möglich.
- Eine sehr ausführliche Dokumentation ist vorhanden.

Nachteile beziehungsweise Probleme:

- Die Metriken LCOM1 (siehe Abschnitt 2.1.8) und NOIS (vergleiche Abschnitt 2.1.2) können nicht ermittelt werden.
- Es ist nicht möglich, die Analyse auf nur einige Klassen zu begrenzen.
- Das Einlesen des Quellcodes sowie die Ermittlung der ersten Analyseergebnisse, wie beispielsweise die LOC-Metrik, dauerte bei REX im ersten Versuch zweieinhalb Stunden, wobei die Fehlermeldungen nach einem kurzfristigen Umschalten zu einer anderen Anwendung erst am Schluss sichtbar wurden, da sich das Programmfenster von Understand for Java nicht aktualisieren konnte. Nach Beseitigung aller Fehler und Angabe aller benötigten Quellen nahm diese Prozedur bei REX immerhin noch 30 Minuten in Anspruch.
- Die Speicherauslastung liegt bereits beim Parsen des Quellcodes bei 80-90% vom gesamten zur Verfügung stehenden virtuellen Speicher (RAM), so dass das Arbeiten mit anderen Programmen stark behindert wird.
- Die Generierung von Berichten dauert mehrere Stunden, wobei die Prozentangaben der Progress-Bar irreführend sind (diese war bei REX bereits nach 20 Minuten bei 99%, obwohl die restliche Analyse anschließend noch über 5 Stunden beanspruchte).

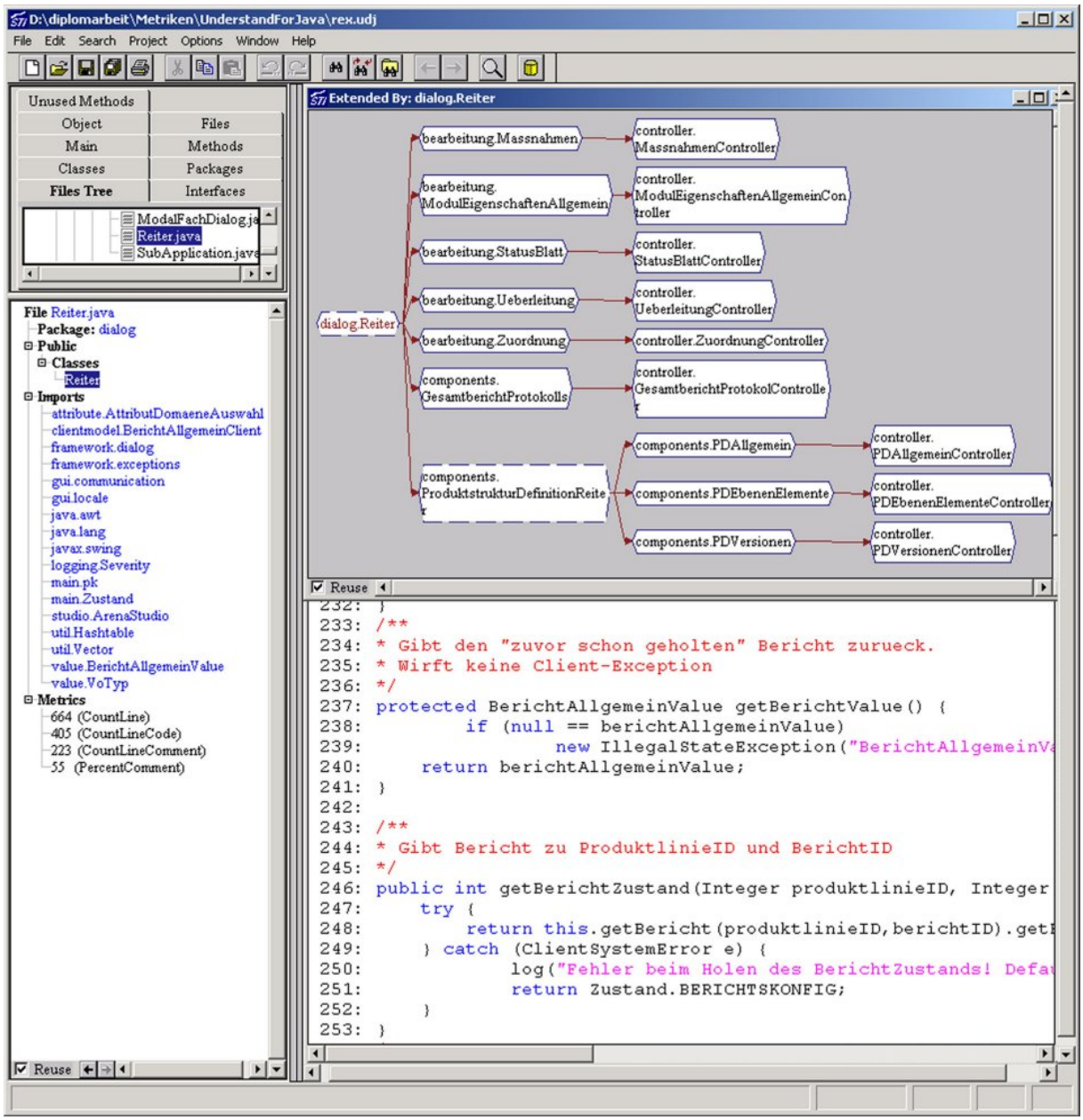

Abbildung 2.2: Understand For Java: Diagramm-Ansicht

- Zwar wird in der Dokumentation behauptet, dass bei fehlenden Benutzerangaben die Anwendung automatisch das JDK-eigene Archiv *src.jar* lokalisiert, aber das Tool scheint dies doch nicht zu können.
- Bei der Berichterstellung wurden für 28,5 MB Quellcode 350 MB Daten erzeugt, wobei die Größe der Projektdatei auf 60 MB stieg.

2.3.4 Vergleich der Tools

Anhand von Tabelle 2.3 auf Seite 23 sieht man die Unterschiede, aber auch die Gemeinsamkeiten der in den vorangegangenen Abschnitten beschriebenen Werkzeuge zur Analyse und Metrikenerstellung von Java-Programmen.

2.4 Analyseergebnisse des Projektes REX

Mit Hilfe der in den Abschnitten 2.3.1 - 2.3.3 vorgestellten Werkzeuge wurde das Projekt REX analysiert. Die Ergebnisse dieser Analyse werden im Folgenden zusammengefasst:

Metrik	**Mittelwert**	**höchster Wert**
LOC	131	3.287
NOIS	6	76
NOA	4	213
NOM	13	310
CC	22	521
RFC	98	907
CBO	8	145
LCOM1	399	49.892
LCOM2	77	100

Tabelle 2.2: REX-Metriken vor dem Refactoring

- Tabelle 2.2 gibt eine Übersicht über die in REX, bestehend aus 1.553 Klassen und Interfaces, erhobenen Metriken. Insgesamt sind 202.734 Zeilen mit ausführbarem Code vorhanden, die Gesamtzahl der Attribute beläuft sich auf 6.841, die der Methoden auf 20.538. Zwar geht aus den durchschnittlichen Werten mit Ausnahme der zu hohen LCOM-Metriken und einer Überschreitung der NASA-Empfehlung für CBO um 3 Einheiten beziehungsweise 60% (vergleiche Abschnitt 2.1.7) hervor, dass die REX-Klassen durchschnittlich gesehen keine Auffälligkeiten zeigen, jedoch stellt man anhand der höchsten ermittelten Werte für die jeweilige Metrik fest, dass einige Klassen eventuell dem Refactoring unterzogen werden müssen, um an Übersichtlichkeit und Struktur zu gewinnen.
- Small Worlds fand 59 Klassen, die in mehrere Klassen zerlegt werden sollten, weil sie zu viele Verantwortlichkeiten haben. Diese Klassen gehören folgenden Kategorien an:
  - 28 Client-Klassen (Controller und Views),
  - 17 Server-Klassen (Beans),
  - 7 Value-Objekte,
  - 5 sonstige serverseitige Klassen sowie
  - 2 Test-Klassen (Unit-Test).
- 43 gegenseitig abhängige Klassen konnten von Small Worlds ermittelt werden.

Eigenschaft	**Small Worlds**	**Together Control-Center**	**Understand for Java**
Identifikation von Schwachstellen im Code	Tangles, Butterflies, et cetera	Metriken (alle in den Abschnitten 2.1.1 - 2.1.8 genannten sowie viele andere mehr), Audits	Metriken (LOC, RFC, CBO, LCOM2 und so weiter)
Auswahl der zu erstellenden Metriken für ausgewählte Klassen	bedingt	ja	bedingt
Generierte Metriken können exportiert werden	HTML-Format	CSV- und HTML-Format	Text- und HTML-Format
Diagrammexport	GIF-Format	GIF-, SVG- und WMF-Format	JPEG-, BMP- und PNG-Format
J2EE-Unterstützung	nicht vorhanden	vorhanden	nicht vorhanden
Quellcode kann eingesehen werden	ja, aber keine Suchmöglichkeit, es sei denn externer Editor	ja	ja
Systemressourcenverbrauch	behindert andere Tools nicht	hoch, wenn viele (> 20) Klassen zu analysieren sind	inakzeptabel
Programmbedienung	zum Teil sehr gewöhnungsbedürftig	leicht erlernbar	kurze Einarbeitung erforderlich
Benutzerdokumentation	kontextbezogen im Tool, FAQ.html	sehr ausführlich	sehr ausführlich
Preis	$1.750	€8.970 - €14.352	$459

Tabelle 2.3: Gegenüberstellung der Analyse-Werkzeuge

- Viele der von Small Worlds festgestellten Überschreitungen der intern festgelegten Grenzwerte betreffen entweder die Framework-Klassen, Value-Objekte oder Klassen, die eine eindeutige Zuordnung von Daten erlauben (und als eine Art von Primary Keys fungieren), so dass auch hier weitere Untersuchungen seitens der Entwickler erforderlich sind.

- Mit Hilfe von Understand for Java konnte zwar festgestellt werden, dass in REX

  – 12.134 Methoden unbenutzt sind, allerdings muss jede einzelne Methode seitens der Entwickler erneut inspiziert werden, weil das Tool weder das Reflection API[4] noch J2EE unterstützt und auf die Anwendung dieser Techniken zurückzuführende Methoden als ,,unused" kennzeichnet. Ferner wird dabei auch auf alle Methoden

[4]Dieses Merkmal von Java wird verwendet, wenn man beispielsweise die Methoden, die aufgerufen werden müssen, erst zur Laufzeit bestimmt (vergleiche *java.lang.reflect.Method* in [www_java_api]).

hingewiesen, die aufgrund der Interface-Implementierung in der Klasse vorkommen, aber die nicht direkt aufgerufen werden.

– 7.738 Objekte nicht benötigt werden, was sich durch nachträgliche ,,manuelle" Analyse in einigen Fällen revidieren lässt:
  * J2EE-spezifisches Design,
  * lokale Verschattung (durch Übergabeparameter) der Klassen-Attribute, wobei die Übergabeparameter seitens des Tools als ,,unused" fehl identifiziert werden,
  * Übergabeparameter einer wegen einer Interface-Implementierung aufgeführten Methode (beispielsweise bei der Implementierung von *java.awt.event.MouseListener* muss die Methoden-Signatur der beim Klicken einer Maustaste aufgerufenen Methode folgendermaßen aussehen

    mouseClicked(MouseEvent e)...,

    wobei in der Methode die im Übergabeparameter beinhaltenen Informationen eventuell nicht benötigt werden).

2.5 Fazit

Obwohl Metriken den Anschein von Objektivität verleihen, dürfen Design-Entscheidungen nicht einzig und allein von ihnen abhängig gemacht werden, um so lediglich das Ziel zu verfolgen, bestimmte Metriken in Grenzen zu halten. Vielmehr sollte man Metriken als eine zusätzliche Informationsquelle ansehen, mit deren Hilfe beispielsweise auf Klassen hingewiesen wird, die genauer untersucht werden sollen. Vor allem darf nicht vergessen werden, dass ein zu hoher Wert einer der Metriken nicht immer ein Indiz für Probleme im Code ist und umgekehrt.

Keines der untersuchten Tools erfüllt alle in Abschnitt 2.2 gestellten Anforderungen. Together ControlCenter überzeugt mit den Auswahlmöglichkeiten bezüglich der zu analysierenden Klassen und zu erstellenden Metriken. Der größte Nachteil liegt in der Rechengeschwindigkeit, vor allem dann, wenn man umfangreiche Daten erheben will.

Im Vergleich zum Together ControlCenter, bietet Understand For Java die Möglichkeit, diejenigen Klassen, Methoden und Variablen, die nicht benutzt werden, zu identifizieren. Allerdings ist auch dort eine zusätzliche manuelle Überprüfung notwendig, weil das Tool weder J2EE noch das Reflection-API unterstützt und deshalb zum Beispiel einige Methoden, die tatsächlich benötigt werden, in die Liste der unbenutzten Methoden aufnimmt.

Small Worlds kann die Stellen im Programmcode aufspüren, deren Änderungen viele andere Anpassungen nach sich ziehen werden (beispielsweise anhand der ,,Tangles"); dies ist auch die einzige Metrik, bei der nicht nur die ,,gefährdete" Klasse, sondern auch die Klassen, die sie zur solchen machen, erwähnt werden. Das Tool ermittelt zwar die Klassen beziehungsweise Interfaces, die nach Meinung der Erfinder als besonders riskant eingestuft werden können, die eigentliche Analyse des Problems wird den Entwicklern aber nicht abgenommen, so dass diese letztendlich mit den bekannten Suchtechniken (beispielsweise bei der Suche nach einer Klasse, die in der Klasse benutzt wird) das Problem selbst lokalisieren müssen. Auch zeigte Small Worlds zum Beispiel für REX eine Stabilität bezüglich der Änderungen von über 90% an. Dieser Wert wird anhand der Anzahl der durchschnittlich in einer Beziehung stehenden

Klassen ermittelt. Er ist bei REX zwar laut Small Worlds 3.72, täuscht aber darüber hinweg, dass es zum Teil Klassen gibt, die weit über dem Durchschnittswert liegen. Insofern ist es fraglich, ob der Einsatz solcher Tools spürbare Erleichterungen bei der Suche nach Problemen im Quellcode mit sich bringt.

Zusammenfassend kann man sagen, dass die ausgewählten Tools zur Zeit nur begrenzte Analyse-Möglichkeiten zur Verfügung stellen. Hinweise auf eventuelle Probleme im Quellcode, sei es anhand überschrittener Metriken oder aufgrund von zu vielen Abhängigkeiten, müssen seitens der Entwickler weiter untersucht werden, um festzustellen, ob die betroffene Code-Stelle wirklich einer Überarbeitung bedarf.

3 Refactoring

Insbesondere seit der Veröffentlichung des Buches von Martin Fowler (siehe [Fowler00]) ist der Begriff ,,Refactoring" zu einem ,,Buzzword" geworden, wenn es um die Verbesserung von Software-Design geht. Was seit knapp zehn Jahren vor allem mit Hilfe des auf den von William Opdyke definierten Refactorings (vergleiche [Opdyke92]) basierenden *Refactoring Browser*, der von Don Roberts und John Brant entwickelt wurde (siehe [www_refBrowser]), besonders bei den Smalltalk-Entwicklern zum Alltag gehört, hält immer stärkeren Einzug in die Anwendungsentwicklung auch bei C++- und Java-Programmierern.

Ziel dieses Kapitels ist es, nicht nur eine Einführung in das Themengebiet ,,Refactoring" und eine Übersicht über die relevanten Refactoring-Techniken zu geben, sondern auch die Vor- und Nachteile von Refactoring unter die Lupe zu nehmen.

3.1 Refactoring-Techniken

Refactoring wird als ein Prozess definiert, der zu einer Umstrukturierung des Codes führt, ohne dass dessen Sematik geändert wird. Das Ziel von Refactoring ist es, den Quelltext verständlicher, lesbarer und wartbarer zu machen, wodurch auch das Design der Software nachhaltig verbessert werden soll.

Es gibt eine Vielzahl (über 40) an möglichen Refactorings, die angewandt werden können. Eine vollständige Liste aller bekannten Refactorings findet man auf der Refactoring-Homepage (siehe [www_refact]). Die folgende Aufzählung bietet lediglich eine kurze Übersicht über einige wichtige und besonders oft verwendete Refactoring-Techniken:

Variable/Attribut/Methode/Klasse/Package umbenennen: Beim Umbenennen versucht man, mittels eines aussagekräftigen Namens die von einer Variable/Methode/Klasse beziehungsweise von einem Attribut/Package zu erfüllenden Aufgaben zu verdeutlichen.

Methode extrahieren/integrieren: Im Falle von Extrahieren wird eine Methode in mehrere (mindestens zwei) kleinere Methoden zerlegt, wobei der in andere Methode ausgelagerte Code-Abschnitt an der alten Stelle durch einen entsprechenden Aufruf der neuen Methode ersetzt werden. Beim Integrieren einer Methode wird deren Aufruf gegen den Methodenrumpf ausgetauscht.

Variable einführen: Sehr lange und komplizierte Ausdrücke können dadurch vereinfacht werden, dass man für jeden Teilausdruck eine Variable einführt und den jeweiligen Ausdruck durch diese ersetzt.

Methode/Attribut verschieben: Dabei verschiebt man eine Methode/ein Attribut in eine andere Klasse, in der sie/es semantisch gesehen angebrachter ist.

Klasse extrahieren/integrieren: Hier geht es darum, eine Klasse in mehrere Klassen zu zerlegen beziehungsweise mehrere Klassen zu einer Klasse zusammenzufassen.

Superklasse extrahieren/integrieren: Bei der Extraktion einer Superklasse wird ein Teil der Methoden und Attribute der bestehenden Klasse in die neu zu erstellende Superklasse verschoben, bei der Integration der Superklasse wird diese in die Unterklasse komplett mit aufgenommen.

Bedingte Anweisung durch Polymorphismus ersetzen: Falls in jedem Zweig der Anweisung ähnliche Operationen durchgeführt werden, wird eine Superklasse erstellt. Von ihr sind die Klassen abzuleiten, deren Methoden die jeweilige bedingte Anweisung repräsentieren, so dass man anstelle dieser Anweisung lediglich die Methoden der Superklasse aufrufen muss.

Parameterobjekt einführen: Dabei ersetzt man alle einer Methode übergebenen Parameter durch ein Objekt, das die Werte beinhaltet.

Attribut einkapseln: Bei diesem Refactoring werden Methoden zum Ändern („Setter") und Lesen („Getter") eines Attributs hinzugefügt, so dass der Zugriff (zumindest von anderen Klassen) auf dieses Attribut nur über diese Methoden möglich wird.

Fast alle diese Techniken lassen sich in mehrere kleine Schritte zerlegen, so dass die Wahrscheinlichkeit minimiert werden kann, dass bedingt durch Unachtsamkeit Fehler entstehen können. Nach jedem dieser kleinen Schritte wird empfohlen, die betroffene Code-Stelle ausführlich zu testen (zum Beispiel mit Hilfe der Unit-Tests, siehe dazu auch Abschnitt 3.4.4). Natürlich kann man verschiedene Refactorings miteinander kombinieren, um beispielsweise das Design eines Code-Abschnitts sukzessive zu verändern.

Bevor man aber den Quellcode einem Refactoring unterzieht, ist es wichtig, sich klar zu machen, was man erreichen will. Es sind unter anderem folgende Szenarien denkbar:

- Man muss beispielsweise in einem Dialog eine Funktionalität hinzufügen, stellt nach einiger Zeit fest, dass diese schon in einem anderen Dialog vorhanden ist. Hier ist es sinnvoll, diese Funktionalität in eine Klasse oder Methode auszulagern, die dann von allen Klassen oder Methoden, in denen sie benötigt wird, benutzt werden kann. Damit würde die Entstehung des mehrfachen Codes verhindert.

- Zur Bewältigung einer Aufgabe müssen einer Methode im Laufe der Zeit immer mehr Parameter mit übergeben werden, so dass die Parameterliste sehr lang und unübersichtlich wird. Hier sollte man über die Einführung einer Klasse nachdenken, in der die entsprechenden Werte zu setzen sind; eine Instanz dieser neu geschaffenen Klasse kann dann der Methode übergeben werden.

- Wenn man im Rahmen eines Code-Reviews zu dem Schluss kommt, dass viele der in den Abschnitten 2.1.1 - 2.1.8 erwähnten Metriken die empfohlenen Grenzwerte überschreiten beziehungsweise sehr hoch sind, und dass es gegenseitige Abhängigkeiten zwischen den Klassen gibt, sollte man gezielt „aufräumen", indem man zum Beispiel einzelne Methoden zwischen den Klassen verschiebt oder gar neue Klassen extrahiert.

3.2 Software-Entwicklungs-Techniken und Refactoring

In den Abschnitten 3.2.1 und 3.2.2 werden die Unterschiede von Refactoring gegenüber anderen wichtigen Software-Techniken zur Verbesserung der Qualität, wie beispielsweise Reengineering, herausgearbeitet sowie seine Bedeutung für einige häufig eingesetzte Vorgehensmodelle, wie zum Beispiel evolutionäre Entwicklung, untersucht.

3.2.1 Abgrenzung gegenüber anderen Techniken

Refactoring wird als eine Möglichkeit gesehen, die Code-Struktur zu verbessern. Wie unterscheiden sich andere Software-Techniken, die auch eine Verbesserung der Code-Struktur und Software-Qualität zum Ziel haben, von Refactoring?

3.2.1.1 Reengineering

Während des Reengineerings wird ein vorhandenes System mit dem Ziel inspiziert, ein komplett neues System zu schreiben, dessen zugrunde liegende Anforderungen zwar dem alten System entsprechen, dessen Implementierung sich aber völlig davon unterscheidet (oft werden Legacy-Systeme durch Reengineering auf den neuesten Stand der Technik gebracht, um die Wartungskosten zu reduzieren).

Der entscheidende Unterschied zwischen Refactoring und Reengineering besteht darin, dass man beim Refactoring den bestehenden Code und seine Semantik weitgehend beibehält und nur durch Restrukturierungen das Design verbessert. Beim Reengineering spielt dagegen der lauffähige alte Code nur selten eine Rolle, da dieser durch eine komplett neue Lösung ersetzt wird, wodurch natürlich die Gefahr besteht, Fehler einzubauen. Außerdem sind die Kosten von Reengineering weit höher als die von Refactoring.

3.2.1.2 Redesign

Beim Redesign wird die Software-Architektur unter die Lupe genommen, wobei eine bis alle Komponenten neu gestaltet werden. Die Änderungen werden allerdings im Gegensatz zum Refactoring nicht in vielen kleinen Schritten, sondern in einem großen Schritt durchgeführt, wobei zumindest ein Teil des Systems von Grund auf neu implementiert wird. Der wesentliche Unterschied zwischen Redesign und Reengineering liegt darin, dass beim Reengineering sich nicht nur die Programmiersprache ändern kann, sondern auch die Analyse der Anforderungen erneut durchgeführt werden muss.

3.2.2 Bedeutung von Refactoring beim Einsatz von ausgewählten Vorgehensmodellen

Refactoring ist ein fester Bestandteil mancher Software-Entwicklungs-Prozesse (siehe Abschnitt 3.2.2.1), bei den anderen ist sein Einsatz empfehlenswert (vergleiche Abschnitt 3.2.2.2).

3.2.2.1 Extreme Programming

Das von Kent Beck (siehe [Beck00]) eingeführte Extreme Programming gehört zu der Gruppe der agilen Prozesse (vergleiche [www_agil_dev]) und basiert auf dem inkrementellen Ansatz

der Software-Entwicklung. Nach den Extreme-Programming-Prinzipien muss während der Design-Phase nur das Allernötigste grob entworfen werden, wobei immer der einfachste Weg zu wählen ist; im Laufe der Implementierung wird das Design dann mit Hilfe von Refactoring sukzessive verfeinert und verbessert.

3.2.2.2 Evolutionäre Entwicklung

Beim evolutionären Ansatz werden die Entwicklungszyklen der Software einerseits dadurch gekürzt, dass in jedem Zyklus nur einige ausgewählte Features analysiert und implementiert werden; andererseits werden die Zyklen solange wiederholt, bis das zu entwickelnde System den vollen von ihm erwarteten Funktionsumfang zur Verfügung stellt.

Dabei besteht die Gefahr, dass die Software-Struktur durch stetige Veränderungen stark beeinträchtigt wird. Vor allem aufgrund der häufig mangelnden Dokumentation (vergleiche [Sommer00]) der einzelnen Zyklen kann Refactoring bei der Implementierung notwendig werden. Dabei geht es in erster Linie um Refactoring im Kleinen wie zum Beispiel *Methode extrahieren*, *Variable verschieben*, et cetera.

3.3 Vorteile des Refactorings

Die Vorteile von Refactoring beruhen auf der verbesserten Wartbarkeit (vergleiche Abschnitt 3.3.1) und Kapselung der Klassen (siehe Abschnitt 3.3.2).

3.3.1 Wartbarkeit

Die Wartung des Quellcodes kann durch Refactoring beispielsweise auf folgende Arten und Weisen vereinfacht werden:

- *Methode extrahieren* und *Methode in die Superklasse verschieben* helfen, Redundanz zu vermeiden, indem *eine* Funktionalität nur *einmal* implementiert wird. Dies bedeutet wiederum, dass eine später erforderliche Änderung nur an *einer* Code-Stelle durchgeführt werden muss, so dass eine zeitaufwendige Suche nach allen Code-Abschnitten, in denen diese Anpassung nötig wäre, entfällt.
- *Variable/Attribut/Methode/Klasse umbenennen* sowie *Variable einführen* führen zu einer besseren Lesbarkeit und damit auch Übersichtlichkeit des Quellcodes, was den Entwicklern hilft, sich schneller in einen Code-Abschnitt einarbeiten zu können, wenn zum Beispiel eine Änderung oder Erweiterung ansteht.

All dies verbessert entscheidend die Software-Qualität und hat entsprechend eine höhere Kunden-/Benutzerzufriedenheit zur Folge, da die Wartungszyklen verkürzt werden können.

3.3.2 Kapselung

Die Kapselung der Klassen wird durch Refactoring-Techniken wie *Attribut/Methode verschieben* verbessert, indem beispielsweise Methoden, die zur Erfüllung ihrer Aufgaben hauptsächlich Methoden und Attribute einer anderen Klasse benötigen, in diese verschoben werden. *Klasse extrahieren* hilft dadurch, dass eine Klasse, die mehrere Funktionalitäten zur Verfügung stellt, in mehrere in sich geschlossene Klassen aufgeteilt wird.

3.4 Probleme, die durch Refactoring entstehen können

Neben den in Abschnitt 3.3 aufgelisteten Vorteilen, die der Einsatz von Refactoring mit sich bringt, dürfen die damit verbundenen Nachteile und Probleme nicht aus den Augen verloren werden. Wie bei der Anwendung jeder anderen Technik sollte auch bei Refactoring das Kosten-Nutzen-Verhältnis ausgewogen sein. Besonders in Anbetracht der Tatsache, dass sich die Vorteile des Refactorings nur schwer messen lassen, ist eine präzise Beurteilung der mit Refactoring verbundenen Probleme und Gefahren von großer Bedeutung. In den Abschnitten 3.4.1 - 3.4.5 werden die kritischen Aspekte von Refactoring vorgestellt und diskutiert.

3.4.1 Performanz

Jeder Methodenaufruf und jede Instanziierung einer Klasse kosten Rechenzeit; wenn sich also die Anzahl der Methoden und/oder Klassen vergrößert (wie es zum Beispiel bei *Methode/Klasse extrahieren* der Fall ist), kann man davon ausgehen, dass das Programm langsamer wird.

Zu beachten ist außerdem die Tatsache, dass der seitens der einmal instanziierten und nicht mehr benötigten Objekte beanspruchte Speicherplatz von der Garbage Collection wieder freigegeben werden muss, was umso länger dauert, je mehr Objekte zu „löschen" sind. Trotz Garbage Collection darf die Speicherplatzproblematik nicht vernachlässigt werden, denn jedes neu zu erzeugende Objekt verbraucht Speicherplatz. Beispielsweise besteht in der Programmiersprache Java die Gefahr, eine *OutOfMemoryException* zu bekommen, weil womöglich noch vorhandene Referenzen auf ein Objekt verhindern, dass der von ihm allokierte Speicherplatz mittels Garbage-Collection bereinigt wird.

3.4.2 Kosten von Refactoring

Wie jede andere Code-Änderung kostet auch Refactoring Zeit. Kleine Umstellungen wie beispielsweise *Methode umbenennen* sind mit Unterstützung entsprechender Werkzeuge (siehe auch Kapitel 4) innerhalb nur weniger Sekunden durchführbar. Die größeren Umstrukturierungen, die vor allem Vererbungshierarchie oder Polymorphismus betreffen, können dagegen Tage dauern. Daher sollten größere Refactorings nicht kurz vor einem anstehenden *Release* durchgeführt werden, sonst ist die Gefahr, eventuell aus Unachtsamkeit entstandene Fehler nicht rechtzeitig erkennen zu können, viel zu groß.

3.4.3 Pflege der Dokumentation

Die Pflege der Dokumentation, in erster Linie der Spezifikation, darf nach dem Refactoring nicht vergessen werden. Nicht nur UML-Diagramme, sondern auch die zum Teil langen Textpassagen, die Schnittstellen- oder Komponentenentwurf beschreiben, müssen immer auf dem aktuellen Stand gehalten werden. Oft wird die Dokumentation von einer bestimmten Person verwaltet; dann ist es notwendig, dass die im Code durchgeführten Änderungen dieser Person mitgeteilt werden. Das Beispiel der Dokumentationspflege zeigt erneut, dass Refactoring auch eine zeitaufwendige Aufgabe sein kann.

3.4.4 Unit-Tests

Unit-Tests können bei der Durchführung von Refactoring auf zweierlei Arten betroffen sein. Einerseits müssen sie mit angepasst werden (wenn zum Beispiel Methodennamen geändert oder Methoden in andere Klassen verschoben werden). Andererseits müssen eventuell neue Unit-Tests geschrieben werden, falls durch Refactoring eine neue Klasse entstanden ist (beispielsweise bei Ersetzung der Methodenparameter durch ein Parameterobjekt). In beiden Fällen wäre also wiederum mit zusätzlichem Zeitaufwand zu rechnen.

3.4.5 Einarbeitung der Entwickler in den neuen Code

Schließlich muss auch beachtet werden, dass, falls der von einem Entwickler geschriebene Code durch einen anderen Entwickler einem Refactoring unterzogen wird, es vorkommen kann, dass sich der ursprüngliche Entwickler zunächst im neuen Code zurechtfinden muss, bevor er weiterarbeiten kann. Größere Refactorings sollten daher mit anderen betroffenen Entwicklern vorher abgesprochen werden, damit die Entwickler nicht gegeneinander arbeiten und sich schneller im geänderten Code orientieren können.

4 Refactoring unterstützende Werkzeuge für Java-Code

Nachdem im vorangegangenen Kapitel eine Einführung in das Thema ,,Refactoring" gegeben worden ist, wird in diesem Kapitel der Frage nachgegangen, ob man mit Hilfe spezieller Werkzeuge die Durchführung von Refactoring vereinfachen kann. Hier werden außerdem die Anforderungen an Refactoring-Werkzeuge ermittelt. In Abschnitt 4.2 werden einige ausgewählte Tools zur Unterstützung von Refactoring evaluiert. Abschließend folgt eine Empfehlung hinsichtlich des zu benutzenden Werkzeugs.

4.1 Anforderungen an Refactoring-Werkzeuge

Die in Abschnitt 3.4.2 erwähnten Kosten von Refactoring können mit Hilfe spezieller Werkzeuge gesenkt werden. Deshalb ist es bei der Wahl eines Refactoring-Werkzeugs vor allem wichtig, dass es

- die für die Durchführung von Refactoring erforderliche Zeit minimiert und
- die aus Unachtsamkeit entstandenen Fehler verhindert, indem es dem Programmierer nicht nur einen großen Teil der Umstellungen abnimmt, sondern diesen auch auf die im Rahmen von Refactoring entstandenen Fehler - beispielsweise Unerreichbarkeit bestimmter Attribute/Methoden nach dem Verschieben einer Methode in eine andere Klasse - hinweist.

Um die Durchführung von Refactoring mittels Tools zu erleichtern und die Entwickler zu seiner Anwendung zu ermutigen, sind folgende Funktionalitäten unentbehrlich (die Aufzählung erfolgt nach absteigender Wichtigkeit):

1. **Das Werkzeug soll interaktiv bedienbar sein.** Da man als Entwickler nicht die Zügel aus der Hand geben und immer den Überblick über die durchgeführten Änderungen behalten will, ist es wichtig, dass der Programmierer die Stellen bestimmt, die einem Refactoring unterzogen werden sollen, und selbst die gewünschte Refactoring-Technik auswählt.
2. **Das große Spektrum der Refactorings soll unterstützt werden.** Dadurch soll dem Entwickler die meiste mit dem Refactoring verbundene Arbeit - wie zum Beispiel die Suche nach Referenzen auf die lokal definierten Variablen, die dann als Parameter der neu zu erstellenden Methode (vergleiche beispielsweise *Methode extrahieren*) übergeben werden müssen, Finden aller Referenzen auf eine Methode, Variable oder Klasse und anschließendes Testen - abgenommen werden. Dadurch soll gewährleistet werden, dass Refactoring schneller von statten geht sowie weniger fehleranfällig ist, und die Entwickler eher geneigt sind, es durchzuführen.

3. **Eine Undo-Funktion soll verfügbar sein.** Falls ein Entwickler feststellen sollte, dass das von ihm durchgeführte Refactoring wenig Sinn macht beziehungsweise aus irgendwelchen anderen Gründen unpassend ist, sollte es möglich sein, den Zustand des Quellcodes vor dem Refactoring wiederherzustellen. Das Vorhandensein der Undo-Funktion ermutigt die Entwickler, eine Änderung am Programm eher durchzuführen, denn im Fall der Fälle kann immer ohne allzu großen Aufwand zum Ausgangszustand zurückgekehrt werden.

4. **Die Refactorings sollen schnell durchführbar sein.** Es ist wichtig, dass die Durchführung von Refactoring den Entwicklungsfluss nicht sonderlich verlangsamt, weil es zum einen sehr nervenaufreibend ist, mehrere Minuten zu warten, bis das Tool mit der ihm auferlegten Aufgabe fertig ist, zum anderen soll man beim Refactoring nicht unnötig viel Zeit verschwenden, so dass man für die eigentlichen Aufgaben (zum Beispiel Erweiterungen des Codes) keine Zeit mehr haben wird, was die Anwendung von Refactoring-Techniken nicht gerade begünstigen würde.

5. **Die Integration in eine IDE soll möglich sein beziehungsweise die IDE soll bereits Refactoring-Funktionalitäten beinhalten.** Die ständige Verfügbarkeit der werkzeugseitigen Refactoring-Unterstützung motiviert die Entwickler, diese auch in Anspruch zu nehmen. Es wäre aber ziemlich umständlich, wenn man jedes Mal vor der Durchführung eines Refactorings ein externes Tool starten, den Quellcode darin importieren und eventuell noch zusätzliche andere Bibliotheken einbinden müsste.

6. **Benutzerfreundlichkeit und intuitive Bedienung beim Einsetzen des Tools spielen eine nicht zu vernachlässigende Rolle.** Denn je einfacher ein Werkzeug zu bedienen ist, desto eher ist man geneigt, es auch einzusetzen, wohingegen komplizierte Schritte bei der Benutzung erst mühsam erlernt werden müssen und daher abschreckend wirken.

7. **Der Entwickler soll sofort auf die im Laufe des Refactorings entstandenen Fehler hingewiesen werden.** Beispielsweise kann es bei *Methode verschieben* passieren, dass die verschobene Methode andere Methoden oder Attribute ihrer ursprünglichen Klasse benötigt. Es muss also gewährleistet werden, dass die Fehler sofort lokalisiert und beseitigt werden können, da deren Ursprung nachvollziehbar ist. Daher wäre es denkbar, dass der Entwickler im Falle der Entstehung von Fehlern einen entsprechenden Hinweis bekäme.

4.2 Refactoring-Tools

Mit steigender Popularität von Refactoring erschienen auch mehrere Werkzeuge am Markt, die diese Technik für die Programmiersprache Java unterstützten. Dabei unterscheidet man zwischen solchen Werkzeugen, die als Plug-In in eine - zum Teil von einer anderen Firma entwickelte - IDE integriert werden können, und denen, die bereits fester Bestandteil einer IDE sind.

Tabelle 4.1, die sowohl kostenlose als auch kommerzielle Tools aufführt, ist bei weitem nicht vollständig, macht aber deutlich, dass eine sorgfältige Wahl eines Refactoring-Werkzeugs

Name	Homepage	Preis	IDE
CodeGuide	[www_codeguide]	$269 - $299	Standalone
DPT	[www_dpt]	kostenlos	Standalone
Eclipse	[www_eclipse]	kostenlos	Standalone
IntelliJ IDEA	[www_idea]	$367 - $599	Standalone
JBuilder	[www_jbuilder]	$2.999	Standalone
jFactor	[www_jfactor]	$695	Plug-In für VisualAge for Java
jRefactory	[www_jrefactory]	kostenlos	Plug-In für JBuilder, Elixir IDE
RefactorIT	[www_refactorit]	€200	Standalone, Plug-In für JBuilder, Forte4Java, Sun ONE Studio, Oracle9i JDeveloper
teikade	[www_teikade]	kostenlos	Standalone
Together ControlCenter	[www_together]	€8.970 - €14.352	Standalone
Transmogrify	[www_transmogrify]	kostenlos	Plug-In für JBuilder, Forte4Java
VisualAge for Java	[www_visualage_java]	$142	Standalone

Tabelle 4.1: Refactoring-Werkzeuge für Java

notwendig ist. Da es im Rahmen dieser Ausarbeitung nicht möglich war, alle Werkzeuge unter die Lupe zu nehmen, wurden nur drei ausgewählt, wobei folgende Kriterien berücksichtigt wurden:

- Das Werkzeug muss stabil und nach Möglichkeit fehlerfrei sein.
- Technischer Support muss vorhanden sein, damit man für die eventuell entstandenen Probleme innerhalb einer annehmbaren Zeit Lösungen finden kann.

In den Abschnitten 4.2.1 - 4.2.3 werden drei Produkte vorgestellt, die allesamt im interaktiven Modus arbeiten. Jedes der Werkzeuge ist eine vollwertige IDE, so dass man Refactoring neben der eigentlichen Entwicklung durchführen kann. Eines der Tools (Eclipse) ist eine Weiterentwicklung des bei der msg systems AG bis vor kurzem eingesetzten VisualAge for Java, das andere (IntelliJ IDEA) ist eine Empfehlung von der Refactoring-Homepage (vergleiche [www_refact]), und das dritte Tool (Together ControlCenter) wird bereits bei der msg systems AG als Entwicklungsumgebung und Modellierungswerkzeug verwendet. Die Evaluation erfolgte anhand von REX.

4.2.1 Eclipse

Die von mehreren namhaften Firmen wie Borland, IBM, RedHat, SuSE, TogetherSoft und einigen anderen ins Leben gerufene Plattform *eclipse.org* für Integration von Entwicklungswerkzeugen stellt im Rahmen des *Eclipse Project* eine mächtige IDE zur Verfügung. Die anhand der Evaluation ihrer Version 2.0.1 samt des Java-Plug-Ins mit dem Namen *Java Development Tools (JDT)* gewonnenen Eindrücke werden im Folgenden knapp zusammengefasst.

Vorteile beziehungsweise vorhandene Funktionalitäten:

- Die von Eclipse unterstützten Refactoring-Techniken sind im Wesentlichen
  - Klasse/Methode/Attribut/Variable umbenennen,
  - Klasse in ein anderes Package verschieben,
  - Methode extrahieren,
  - Methode/Attribut in die Superklasse verschieben,
  - statische Methode in eine andere Klasse verschieben sowie
  - Attribut einkapseln.
- Eine Undo-Funktion steht zur Verfügung.
- Die Durchführung von Umstrukturierungen mit Hilfe des Tools ist sehr performant.
- Beim Anstoßen eines Refactorings erhält man gegebenenfalls Hinweise (zum Beispiel was man auswählen muss oder aus welchem Grund das gewünschte Refactoring nicht möglich ist), falls das Tool es nicht durchführen kann. Die Dialoge, die den Benutzer bei der Durchführung von Refactoring unterstützen, sind übersichtlich (siehe Abbildung 4.1). Während des Refactorings hat der Entwickler nicht nur die Möglichkeit, dessen Durchführung vorzeitig abzubrechen oder um mehrere Schritte zurückzugehen, sondern auch mit zu entscheiden, welche der Refactoring-Schritte gar nicht ausgeführt werden sollen (siehe zum Beispiel die Checkboxes im oberen Dialog-Abschnitt der Abbildung 4.1).
- Dank der Tatsache, dass ein *incremental Java compiler*[1] in Eclipse zum Einsatz kommt, wird man sofort auf die während des Refactorings eingefügten Fehler hingewiesen (zum Beispiel, dass die verschobene Methode auf Attribute oder Methoden zugreift, die von der neuen Klasse aus nicht sichtbar sind).

Nachteile beziehungsweise Probleme:

- Eclipse bietet keine Unterstützung für wichtige Refactoring-Techniken wie beispielsweise *Superklasse/Interface extrahieren*.
- Die Import-Anweisungen, die nach der Verschiebung eines Attributs oder einer Methode in der aktuellen Klasse nicht mehr erforderlich sind, werden nicht automatisch gelöscht.
- Bei Projekten, die mehrere externe Bibliotheken benutzen (bei REX über 40 JAR-Dateien), muss man, um einen *java.lang.OutOfMemoryError* bei der Kompilation zu vermeiden, den für JVM[2] verfügbaren Speicherplatz mit den Optionen `-ms` und `-mx` erhöhen.

4.2.2 IntelliJ IDEA

Die Evaluationsergebnisse der Version 3.0 der von JetBrains, Inc. entwickelten IDE mit dem Namen IntelliJ IDEA werden im Folgenden vorgestellt.

Vorteile beziehungsweise vorhandene Funktionalitäten:

[1] Dieser Compiler muss nicht explizit aufgerufen werden, sondern wird von der IDE selbst jedes Mal automatisch ausgeführt, wenn irgendwelche Änderungen am Code vorgenommen werden.
[2] engl. Java Virtual Machine

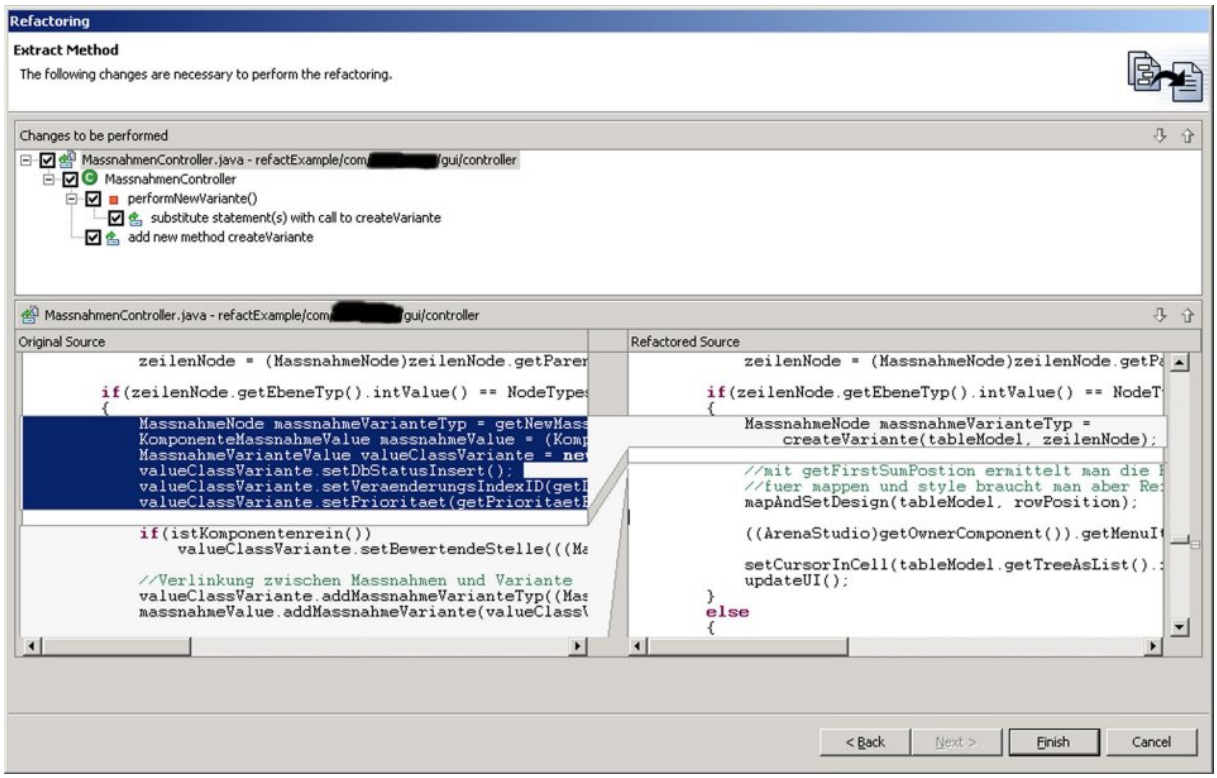

Abbildung 4.1: Eclipse: Methode extrahieren

- Das Tool bietet eine umfangreiche Auswahl an Refactorings, unter denen sich nicht nur
  - Package/Klasse/Methode/Variable umbenennen,
  - Klasse in ein anderes Package verschieben,
  - Methode/Superklasse/Interface extrahieren,
  - Attribut/Methode in die Super-/Unterklasse verschieben,
  - Attribut einkapseln,

  sondern auch
  - Konstante/Variable/Parameter einführen,
  - statische Methode/statisches Feld in die Superklasse verschieben,
  - Methodensignatur ändern,
  - Vererbung durch Delegation ersetzen

  sowie einige andere befinden. Bei den Umbenennen-Refactorings wird die Möglichkeit angeboten, im gesamten Quellcode nach Vorkommnissen alter Namen umbenannter Klassen/Methoden/Attribute/Variablen zu suchen, so dass sowohl Kommentare als auch Änderungen in Bezug auf die Benutzung des Reflection APIs mit angepasst werden können. Bei *Methode extrahieren* kann man die Reihenfolge der Übergabeparameter ändern. Sowohl bei *Attribut einkapseln* (siehe Abbildung 4.2 auf Seite 37) als auch bei

Attribut/Methode in die Super-/Unterklasse verschieben besteht die Möglichkeit, gleich mehrere Attribute beziehungsweise Methoden dem Refactoring zu unterziehen, was eine erhebliche Zeitersparnis mit sich bringt.

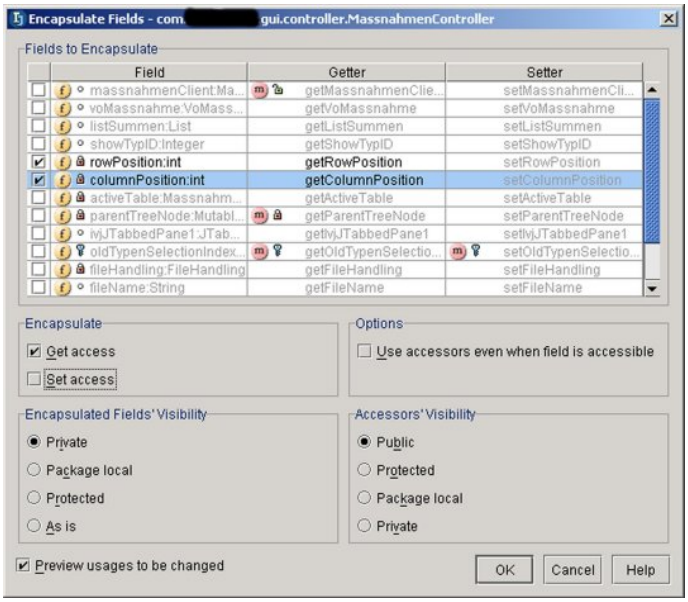

Abbildung 4.2: IntelliJ IDEA: Attribute einkapseln

- Eine Undo-Funktion ist vorhanden und funktioniert innerhalb weniger Sekunden.
- Die Durchführung von Refactorings nimmt sehr wenig Zeit in Anspruch, so dass die Arbeit nicht durch lästiges Warten beeinflusst wird.
- Alle Refactorings können über (Kontext-)Menüeinträge angestoßen werden; falls dazu die Voraussetzungen fehlen, sieht man eine ausführliche Fehlermeldung mit dem Hinweis, wie man doch noch das ausgewählte Refactoring durchführen kann (zum Beispiel bei *Methode extrahieren*: einen Code-Abschnitt auswählen, der extrahiert werden soll).
- Man bekommt umgehend Hinweise auf die im Rahmen von Refactoring entstandenen Fehler. So weist das Programm den Benutzer beispielsweise darauf hin, falls die in die Superklasse zu verschiebenden Methoden auf Attribute zugreifen, die von der Superklasse aus nicht erreicht werden können. In diesem Fall kann man entweder das Refactoring abbrechen, oder die Fehler ignorieren, um diese dann später per Hand zu korrigieren, wobei im letzteren Fall die betroffenen Code-Stellen farblich hervorgehoben werden.

Nachteile beziehungsweise Probleme:

- Genauso wie bei Eclipse (vergleiche Abschnitt 4.2.1) wird bei IntelliJ IDEA keine automatische Bereinigung der Import-Statements nach einem Refactoring vorgenommen. Allerdings werden die nicht mehr benötigten Importe farblich hervorgehoben.
- Bei der Undo-Funktion wurde das im Rahmen des Klasse-Verschieben-Refactorings neu angelegte Verzeichnis (für das bis dahin noch nicht existierende Package) seitens des Tools nicht automatisch gelöscht.

4.2.3 Together ControlCenter

Die Vorzüge und Probleme bei der Refactoring-Durchführung mit der Version 6.0 des Together ControlCenter von Borland werden im Folgenden erläutert.

Vorteile beziehungsweise vorhandene Funktionalitäten:

- Das Tool stellt über Menüeinträge Unterstützung für folgende Refactoring-Techniken zur Verfügung:
  - Parameter/lokale Variable umbenennen,
  - Klasse in ein anderes Package verschieben,
  - Methode/Superklasse/Interface extrahieren,
  - Attribut/Methode in die Super-/Unterklasse verschieben,
  - Attribut einkapseln.

  Außer der über Menüeinträge unterstützten Refactorings besteht die Möglichkeit, Refactorings direkt in UML-Diagrammen durchzuführen (vergleiche auch [Astels02]). Beispielsweise kann man in einem Klassendiagramm Methoden mit Drag&Drop zwischen den einzelnen Klassen verschieben, was dann im Code innerhalb weniger Sekunden nachgezogen wird.
- Eine Undo-Funktion ist vorhanden und erledigt ihre Arbeit meistens (siehe hierzu „Probleme") einwandfrei.
- Man bekommt umgehend eine Warnung, falls das gewünschte Refactoring Konflikte verursachen kann. Dabei hat man die Wahl zwischen Abbrechen der Operation und Ignorieren der Warnung, um das Problem später manuell zu lösen. Die betreffenden Code-Stellen werden im letzteren Fall farblich hervorgehoben. Bei *Attribut/Methode verschieben* werden seitens des Tools andere Attribute und/oder Methoden vorgeschlagen, die, um Konflikte zu vermeiden, mit verschoben weden sollten. Allerdings bietet Together ControlCenter keine Möglichkeit, sich den jeweiligen Methodenkörper anzeigen zu lassen, so dass man nur anhand des Namens entscheiden muss, ob der Vorschlag sinnvoll ist und akzeptiert werden kann.

Nachteile beziehungsweise Probleme:

- Auf der Seite der wichtigen Refactorings fehlt das Verschieben einer Methode in eine andere Klasse.
- Bei der Durchführung von Refactorings direkt in den UML-Diagrammen entfallen die sonst üblichen Warnhinweise, falls beispielsweise eine verschobene Methode auf Attribute beziehungsweise Methoden zugreift, die der neuen „Mutter"-Klasse unbekannt sind.
- Das Programm erfordert eine hohe Rechnerleistung; selbst dann, wenn der Rechner die minimalen Anforderungen und Empfehlungen des Herstellers erfüllt (Pentium III, 512 MB RAM und 500 MHz), muss man zum Teil mehrere Minuten warten, bevor ein ausgewähltes Refactoring durchgeführt oder zurückgenommen wird.
- Die Bedienung des Tools muss mühsam erlernt werden:
  - Die verfügbaren Refactoring-Techniken erscheinen erst dann im Menü, wenn eine Code-Stelle markiert ist, so dass man zunächst teilweise entweder mittels Trial&Error-Verfahrens oder Auseinandersetzung mit der Dokumentation herausfinden muss, wie man das eine oder andere Refactoring anstoßen kann beziehungsweise welche Refactoring-Techniken überhaupt verfügbar sind.

– Viele Dialoge, die den Benutzer bei der Refactoring-Durchführung unterstützen, sind nicht intuitiv bedienbar. So ist beispielsweise bei *Methode extrahieren* (siehe Abblildung 4.3) zwar möglich, den Kommentar zu ändern beziehungsweise zu ersetzen, der Versuch, mit Hilfe der Enter-Taste den Quelltext der zu extrahierenden Methode anzupassen, führt aber zu einer sofortigen Ausführung des Refactorings.

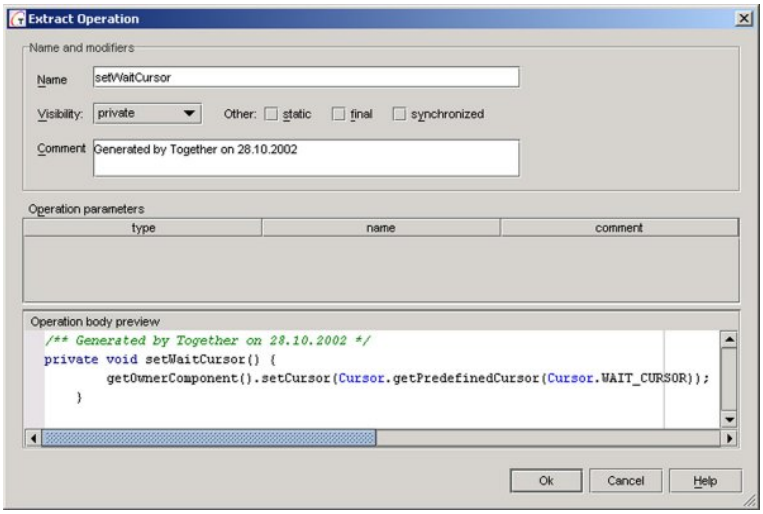

Abbildung 4.3: Together ControlCenter: Methode extrahieren

4.2.4 Vergleich der Tools

In Tabelle 4.2 sieht man nochmal die wichtigsten Aspekte der drei untersuchten Refactoring-Werkzeuge kurz zusammengefasst gegenübergestellt.

4.3 Fazit

Zusammenfassend kann man sagen, dass IntelliJ IDEA der klare Sieger der Evaluation ist. Die Anzahl der seitens des Tools unterstützten Refactoring-Techniken übertrifft die von Eclipse und Together ControlCenter. Weitere Vorteile liegen in der Schnelligkeit, Interaktion mit dem Benutzer (im Vergleich zu den beiden anderen untersuchten Werkzeugen bietet IntelliJ IDEA dem Entwickler mehr Auswahlmöglichkeiten bei der Durchführung von Refactoring, sei es die Bestimmung der Parameterreihenfolge bei *Methode extrahieren* oder die vielen zusätzlichen Features bei *Attribut einkapseln*) sowie in den Hinweisen auf vorhandene Probleme. Deshalb wird IntelliJ IDEA voraussichtlich in den nächsten Jahren der weit verbreiteten IDE von Borland Konkurrenz machen, nicht zuletzt, weil auch die J2EE-Technologie unterstützt wird.

Together ControlCenter unterstützt zwar viele Refactorings, erfordert aber eine gewisse Einarbeitungszeit. Ein zu hoher Preis und die Trägkeit bei der Durchführung von Refactorings und nicht zuletzt das Fehlen solcher wichtigen Refactoring-Technik wie *Methode verschieben* sind die entscheidenden Nachteile.

	Eclipse	**IntelliJ IDEA**	**Together ControlCenter**
Anzahl der unterstützten Refactorings	zufrieden stellend	sehr groß	zufrieden stellend
Undo-Funktion	vorhanden	vorhanden, aber nicht ganz sauber zum Beispiel beim Anlegen einer Superklasse in einem neu zu erstellenden Package	vorhanden
Schnelligkeit	sehr schnell	sehr schnell	sehr langsam
Benutzerfreundlichkeit	gut	sehr gut	zufrieden stellend
Sofortiger Hinweis auf Fehler nach Refactoring	vorhanden	vorhanden, wenn man die betreffende Klasse inspiziert	vorhanden, wenn man die betreffende Klasse Zeile für Zeile inspiziert
Unterstützung der Reflection API bei Refactoring	fehlt	vorhanden	fehlt
Probleme beziehungsweise Fehler bei der Durchführung von Refactoring	Import-Statements werden zum Teil nicht angepasst	Import-Statements werden zum Teil nicht angepasst	Import-Statements werden zum Teil nicht angepasst; Exceptions werden bei einer extrahierten Methode nicht zusammengefasst, sondern erscheinen mehrfach in der *throws*-Liste; kommt mit verketteten Initialisierungen bei der Methoden-Extraktion nicht zurecht
Preis	kostenlos	$367 - $599	€8.970 - €14.352

Tabelle 4.2: Gegenüberstellung der Werkzeuge

Eclipse hat im Wesentlichen den Vorteil des inkrementellen Compilers, so dass man jederzeit die Übersicht über alle (neu entstandenen und alten bereits seit längerer Zeit vorhandenen) Fehler hat; nicht zu vernachlässigen ist natürlich auch die Tatsache, dass die IDE kostenlos zur Verfügung gestellt wird.

Trotzdem sind die seitens der untersuchten Werkzeuge zur Verfügung gestellten Möglichkeiten begrenzt, so dass man bei jedem zweiten Refactoring-Vorhaben entweder etwas vorher oder nachher manuell nachbessern muss. Viel zu oft versagen die Tools gänzlich ihre

Unterstützung, vor allem beim Methode-Extrahieren- und Methode-In-Eine-Andere-Klasse-Verschieben-Refactoring, wobei diese beiden Refactoring-Techniken gerade die sind, die besonders häufig eingesetzt werden.

5 Einsatz von Refactoring in REX

Dieses Kapitel widmet sich der Untersuchung des Einsatzes von Refactoring im Projekt REX, wobei auch die Frage beantwortet werden muss, ob dieser letztendlich die erhofften Vorteile bringen kann. Der erste Teil des Kapitels beschreibt hauptsächlich die durch einzelne Refactoring-Techniken zustande gekommen Veränderungen, im zweiten Teil werden der generelle Einsatz von Refactoring in REX kritisch beurteilt und Vorschläge für das weitere Vorgehen diskutiert.

5.1 Dokumentation der in REX durchgeführten Refactorings

In diesem Abschnitt werden die wichtigsten und wohl gängigsten Refactoring-Techniken exemplarisch anhand einiger ausgewählter Klassen des Projekts REX untersucht. Da Aussagen wie ,,der Code ist verständlicher geworden" beziehungsweise ,,der Code ist jetzt besser strukturiert" einer sehr subjektiven Natur entstammen, erscheint es sinnvoll, auch objektive Kriterien wie Metrikenänderungen heranzuziehen, die durch den Einsatz von Refactoring hervorgerufene Änderungen im Code veranschaulichen und eventuelle Verbesserungen oder auch Verschlechterungen belegen.

Resultierend aus diesen Erwägungen wird der Einfluss von Refactoring folgendermaßen untersucht:

1. Es werden jeweils einige Beispiel-Klassen beziehungsweise -Interfaces ausgesucht, die für die Anwendung der ausgewählten Refactoring-Technik besonderes geeignet zu sein scheinen.
2. Ausgewählte Metriken dieser Klassen werden ermittelt.
3. Die Refactoring-Technik wird angewandt.
4. Es erfolgt entweder nur eine textuelle oder eine anhand von UML-Diagrammen verdeutlichte Beschreibung der durchgeführten Änderungen.
5. Die erreichten Verbesserungen beziehungsweise Verschlechterungen werden subjektiv beurteilt.
6. Dieselben Metriken wie in Punkt 2 werden erneut erhoben und die eventuell festgestellten Veränderungen analysiert.
7. Es wird abgeschätzt, wie häufig die jeweils untersuchte Refactoring-Technik im gesamten REX-Quelltext angewandt werden kann.
8. Auswirkungen auf die Performanz werden untersucht.

9. Abschließend werden die im Laufe der Anwendung der jeweiligen Refactoring-Technik gewonnenen Erkenntnisse zusammengefasst, und das Refactoring wird hinsichtlich seiner Vor- und Nachteile beurteilt.

Aus Übersichtlichkeitsgründen wird bei der Dokumentation der oben beschriebenen Vorgehensweise in den Abschnitten 5.1.1 - 5.1.11 folgendermaßen verfahren:

1. Begründung der Auswahl der Beispielklassen,
2. Beschreibung der durchgeführten Änderungen und der dadurch hervorgerufenen subjektiven Verbesserungen und/oder Verschlechterugen,
3. Analyse der Vorher-/Nachher-Metriken,
4. Häufigkeit der Anwendung in REX,
5. Überlegungen und Messungen bezüglich der möglichen Performanzänderungen,
6. Abschließende Beurteilung der Refactoring-Technik.

5.1.1 Anpassung von Import-Statements

Bei dieser Refactoring-Technik geht es darum, nicht (mehr) benötigte Import-Statements aus dem Quelltext zu entfernen beziehungsweise implizite Importe (wie zum Beispiel den Import eines ganzen Packages) durch explizite Import-Statements (also Importieren nur der jeweils verwendeten Klassen) zu ersetzen. Beispielsweise würde man
`import java.util.*;`
durch
`import java.util.Date;`
`import java.util.Vector;`
ersetzen, falls nur die Klassen *java.util.Vector* und *java.util.Date* tatsächlich benötigt werden.

5.1.1.1 Begründung der Auswahl der Beispielklassen

Um die Vor- und Nachteile dieses Refactorings zu demonstrieren, wurden folgende Klassen ausgewählt:

1. *com.rex.gui.controller.MassnahmenController*, weil diese Klasse über eine Liste der Import-Statements verfügt, die eine ganze Bildschirmlänge verbraucht;
2. *com.rex.gui.bearbeitung.WertweitergabeDialog*, weil in dieser Klasse von Small Worlds (vergleiche Abschnitt 2.3.1) ein ,,Tangle" gefunden wurde; eine anschließende manuelle Analyse des Quellcodes ergab, dass diese Klasse ihre Unterklasse importiert und in einer Methode die Unterklasse als Parameter hat.

5.1.1.2 Beschreibung der durchgeführten Änderungen

In diesem Abschnitt wird das Refactoring der Import-Statements anhand der im vorangegangenen Abschnitt erwähnten Beispielklassen veranschaulicht sowie seine Folgen diskutiert.

Die von der Klasse *com.rex.gui.bearbeitung.WertweitergabeDialog* ursprünglich importierten Klassen und Packages sind in den folgenden Code-Zeilen aufgeführt:

```
import java.awt.*;
import java.util.*;
import javax.swing.*;
import javax.swing.border.*;
import com.msg.framework.gui.locale.*;
import com.msg.framework.model.value.attribute.*;
import com.rex.gui.components.*;
import com.rex.gui.controller.WertweitergabeDialogController;
import com.rex.gui.framework.dialog.*;
import com.rex.model.value.*;
```

Eine Inspektion dieser Liste hat ergeben, dass manche der Importe nicht mehr benötigt werden – sie sind vermutlich so genannte „Leichen", die durch diverse Code-Änderungen zustande kamen. In der Methode, in der eine Instanz der Unterklasse als Parameter übergeben wurde, sieht man, dass diese nicht verwendet wird und daher unter Anpassung des entsprechenden Methodenaufrufs entfernt werden kann. Nach der Eliminierung der nicht mehr verwendeten Importe und Extraktion der wirklich benötigten Klassen aus den importierten Packages sieht der betroffene Code-Abschnitt folgendermaßen aus:

```
import java.awt.Dimension;
import java.awt.GridLayout;
import java.util.Vector;
import javax.swing.BoxLayout;
import javax.swing.JPanel;
import javax.swing.JTextField;
import com.msg.framework.gui.locale.LocalizedCheckBox;
import com.msg.framework.gui.locale.LocalizedLabel;
import com.msg.framework.model.value.attribute.AttributEinfachBoolean;
import com.rex.gui.framework.dialog.FachDialog;
```

Zwar hat sich in diesem Beispiel die Anzahl der Zeilen mit Import-Statements nicht geändert, aber es fällt auf, dass einige der Importe nicht mehr vorkommen, die neu hinzugekommenen hingegen sofort Aufschluss darüber geben, welche externen Klassen in der betroffenen Klasse benötigt werden. Die Klasse ist dadurch **übersichtlicher** geworden, da man auf den ersten Blick erkennt, welche anderen Klassen beziehungsweise Interfaces verwendet werden, und in welchen Packages diese zu finden sind; dies könnte wiederum beispielsweise die API-Recherche erleichtern.

Anderes sieht es bei der Klasse *com.rex.gui.controller.MassnahmenController* aus: Die Anpassung der Import-Statements hat dazu geführt, dass sich die Import-Liste fast verdoppelt hat (vergleiche auch Tabelle 5.2 auf Seite 48) und nun 75 Zeilen in Anspruch nimmt. Eine solche Import-Liste ist **schwer überschaubar**. Deshalb hält sich der Nutzen des Refactorings in dieser Klasse in Grenzen.

Für beide hier vorgestellten Beispiele gilt, dass die betroffenen Klassen durch die expliziten Import-Statements bei eventuell noch anstehenden Änderungen wesentlich einfacher gepflegt werden können. Das heißt also, dass es leichter geworden ist, die nicht mehr benötigten

Importe zu entfernen, da man nicht die Verwendung aller Klassen eines importierten Packages überprüfen muss, um festzustellen, dass keine davon mehr gebraucht wird, sondern es genügt, nur die in der geänderten, gelöschten beziehungsweise verschobenen Methode benötigten Klassen oder Interfaces genau zu untersuchen. Mit anderen Worten wurde die jeweilige Klasse **änderbarer**, vorausgesetzt, dass die Entwickler sich die Mühe machen, die nicht mehr benötigten Importe zu entfernen.

Der **Zeitaufwand** für dieses Refactoring ist eher als **niedrig** einzustufen, vor allem, wenn man auf die Hilfe der verwendeten IDE zurückgreift; viele IDEs bieten die Möglichkeit, die im Code verwendeten, aber noch nicht importierten Klassen durch wenige ,,Mausklicks" hinzuzufügen. Allerdings entfällt diese Hilfe in einigen Fällen, zum Beispiel bei Together Control-Center, wenn es sich bei den zu importierenden Klassen um solche Klassen beziehungsweise Interfaces handelt, auf die ,,gecastet" wird; an dieser Stelle ist es dann notwendig, die erforderlichen Importe manuell hinzuzufügen, wobei die genauen Packagenamen vorher zu bestimmen sind. Auch müsste man die durch Refactoring geänderten Importe per Hand sortieren, um die Import-Liste übersichtlich zu gestalten, denn Hilfen bei der Sortierung bieten IDEs derzeit (noch) nicht.

5.1.1.3 Erhebung von Metriken vor und nach dem Refactoring

Im Folgenden werden bei den in Abschnitt 5.1.1.1 vorgestellten Klassen Metriken verglichen, die vor und nach der Anpassung der Import-Statements erhoben worden sind. Zu beachten ist, dass diejenigen Metriken, die durch Refactoring nicht geändert worden sind, auch nicht weiter verfolgt beziehungsweise diskutiert werden.

In der Klasse *com.rex.gui.controller.MassnahmenController* wurden die Import-Statements angepasst; dabei stellte sich heraus, dass durch 40 Import-Statements (siehe Tabelle 5.2 auf Seite 48) implizit 697 Klassen und Interfaces importiert wurden:

Import-Statement	**Anzahl der impliziten Importe**
`java.awt.*`	95
`java.awt.event.*`	38
`java.util.*`	54
`java.util.List`	1
`javax.swing.*`	122
`javax.swing.tree.MutableTreeNode`	1
`java.io.*`	76
`com.basis.ehl.logging.Severity`	1
`com.klg.jclass.table.*`	95
`com.klg.jclass.table.beans.SpannedCellsWrapper`	1
`com.klg.jclass.table.data.*`	19
`com.msg.framework.exceptions.*`	25
`com.msg.framework.file.FileHandling`	1

Import-Statement	**Anzahl der impliziten Importe**
com.msg.framework.gui.locale.LocalizedText	1
com.msg.framework.model.value.attribute.AttributDomaeneAuswahl	1
com.rex.model.value.BerichtAllgemeinValue	1
com.msg.framework.model.value.domain.Domain	1
com.msg.framework.model.value.domain.KeyValuePairImpl	1
com.msg.framework.model.value.node.NodeImpl	1
com.msg.jclass.table.data.*	11
com.rex.model.auswertungen.value.*	43
com.rex.gui.bearbeitung.Massnahmen	1
com.rex.gui.clientmodel.AuswertungClient	1
com.rex.gui.clientmodel.BerichtAllgemeinClient	1
com.rex.gui.clientmodel.MassnahmenClient	1
com.rex.gui.framework.dialog.InfoDialog	1
com.rex.model.main.display.*	13
com.rex.model.main.nm.nmReport	1
com.rex.model.main.nm.nmAK	1
com.rex.model.main.nm.nmID	1
com.rex.model.main.nm.nmGroup	1
com.rex.model.main.State	1
com.rex.model.massnahme.value.*	18
com.rex.model.massnahme.value.INodeMassnahmeValue	1
com.rex.model.modul.ModulHelper	1
com.rex.model.statusblatt.value.*	42
com.rex.model.statusblatt.vo.test.*	21
com.rex.gui.studio.REX_Studio	1
com.rex.security.FunktionPermission	1
Summe	697

Tabelle 5.1: Auswertung der Import-Statements: MassnahmenController

Aus der Tabelle 5.2 geht hervor, dass nach der Anpassung der Import-Statements die Anzahl der Import-Anweisungen um 35 gestiegen ist, womit auch ein ebensolcher Anstieg des LOC-Wertes zu begründen ist.

Tabelle 5.3 veranschaulicht die Veränderung der Metriken bei der Eliminierung eines unbenutzten Übergabeparameters einer Methode und anschließender Anpassung der importierten Klassen am Beispiel der Klasse *com.rex.gui.bearbeitung.WertweitergabeDialog*.

Metrik	Vorher	Nachher
LOC	1.317	1.352
NOIS	40	75

Tabelle 5.2: Import-Statement-Anpassung: Metrikenänderungen beim MassnahmenController

Metrik	Vorher	Nachher
LOC	282	282
NOIS	10	10
CBO	10	9

Tabelle 5.3: Import-Statement-Anpassung sowie Eliminierung einer unbenötigten Import-Klasse: Metrikenänderungen beim WertweitergabeDialog

Wie bereits in Abschnitt 5.1.1.2 erwähnt, hat sich die Anzahl der Import-Anweisungen nicht geändert, weil zwar einige Import-Anweisungen gänzlich weggefallen sind, die aus dem gleichen Package stammenden Klassen dagegen nicht mehr durch ein Import-Statement importiert werden. Allerdings sank die Anzahl der Klassen und Interfaces, die der Klasse bekannt gemacht worden sind, von 448 auf 10 (vergleiche Tabelle 5.4). Die LOC-Metrik hat sich durch die Umstellungen nicht verändert, denn Importe nehmen denselben Platz wie vorher ein, die Eliminierung eines Übergabeparameters hat lediglich die Auswirkung auf den CBO-Wert, der erwartungsgemäß um 1 gefallen ist.

Import-Statement	Anzahl der impliziten Importe
`java.awt.*`	95
`java.util.*`	54
`javax.swing.*`	122
`javax.swing.border.*`	10
`com.msg.framework.gui.locale.*`	26
`com.msg.framework.model.value.attribute.*`	27
`com.rex.gui.components.*`	28
`com.rex.gui.controller.WertweitergabeDialogController`	1
`com.rex.gui.framework.dialog.*`	8
`com.rex.model.value.*`	77
Summe	448

Tabelle 5.4: Auswertung der Import-Statements: WertweitergabeDialog

5.1.1.4 Häufigkeit der Anwendung

Einige REX-Klassen benötigen eine Bereinigung der Import-Statements aus folgenden Gründen:

- Wenn einige Methoden geändert oder gar gelöscht wurden und somit die eine oder andere Import-Anweisung überflüssig wurde, wurden diese nicht immer sofort eliminiert.
- Sehr oft findet man bei REX Importe ganzer Packages vor, obwohl lediglich wenige Klassen des importierten Packages wirklich verwendet werden.

5.1.1.5 Auswirkungen auf die Performanz

Die Anzahl und Art der Importe spielen keine Rolle bei der Größe der Class-Dateien oder den Ladezeiten. Die unnötigen beziehungsweise impliziten Importe haben lediglich negative Auswirkungen auf die Kompilationszeit (vergleiche [www_java_imports]). Diese steigt an, wenn eine längere Liste nach der Übereinstimmung mit einem Klassen- beziehungsweise Interface-Namen durchsucht werden muss.

Beispielsweise werden in der Klasse *com.rex.gui.WertweitergabeDialog* von den implizit importierten 448 Import-Statements (siehe Tabelle 5.4) lediglich 10 benötigt, was seinerseits bedeutet, dass der Compiler bis zu 44, 8-mal mehr Klassen zu durchsuchen hat, bis eine passende gefunden werden kann.

5.1.1.6 Fazit

Zusammenfassend kann man sagen, dass die Anpassung der Import-Statements insbesondere bei langfristigen Projekten, bei denen sehr viele Änderungen der Klassen zu erwarten sind, sich meistens positiv auf Übersichtlichkeit und Änderbarkeit auswirkt, es sei denn, es werden zum Beispiel aus den *javax.swing*- und java.awt-Packages mehrere Klassen gebraucht. Dann wäre vielmehr zu überlegen, ob es sich wirklich lohnen würde, alle benötigten Importe explizit hinzuschreiben, da eine solche Liste schon eine ganze Bildschirmlänge benötigen und daher unübersichtlich wirken könnte.

Dieses Refactoring benötigt relativ wenig Zeit (sofern die verwendete IDE entsprechende Hilfsmittel zur Verfügung stellt) und kann von jedem Programmierer im Team problemlos durchgeführt werden.

5.1.2 Klassen in ein anderes Package verschieben

Ziel dieses Refactorings ist es, Klassen, die ähnliche Aufgaben haben oder einander ergänzen, in *einem* Package unterzubringen.

5.1.2.1 Begründung der Auswahl der Beispielklassen

Bei näherer Betrachtung der REX-Klassen fällt auf, dass die Packages, in denen sich manche Klassen befinden, aus der Historie heraus ,,gewachsen" sind, so dass es für Außenstehende nur schwer nachvollziehbar ist, wo eine bestimmte Klasse zu finden ist. Ein Beispiel ist der Maßnahmen-Dialog. Dort liegen die Klassen, die beispielsweise für die im Dialog darzustellende Tabelle verantwortlich sind (angefangen von der Platzierung im Dialog über das Aussehen der einzelnen Zellen, die Reihenfolge der Spalten bis zur Anzeige der Werte) in vier verschiedenen Packages.

5.1.2.2 Beschreibung der durchgeführten Änderungen

In diesem Abschnitt wird beschrieben, welche subjektive Änderungen sich durch das Verschieben von Klassen in ein gemeinsames Package ergeben. Tabelle 5.5 zeigt die alten und die neuen Packages der betroffenen Klassen.

Klasse	**Package-Name vorher**	**Package-Name nachher**
Massnahmen	com.rex.gui.bearbeitung	com.rex.gui.massnahmen
MassnahmenController	com.rex.gui.controller	com.rex.gui.massnahmen
MetaTypeMassnahme	com.rex.model.massnahme.value	com.rex.gui.massnahmen
MetaTypeMassnahmeImpl	com.rex.model.massnahme.value	com.rex.gui.massnahmen
MassnahmenLiveTable	com.rex.model.statusblatt.vo.test	com.rex.gui.massnahmen
MassnahmenExport	com.rex.model.statusblatt.vo.test	com.rex.gui.massnahmen
MassnahmeAdapter	com.rex.model.statusblatt.vo.test	com.rex.gui.massnahmen

Tabelle 5.5: Verschiebung mehrerer zusammenhängender Klassen in ein gemeinsames Package

Die neue Package-Struktur bringt einige Vorteile mit sich. So sieht man auf einen Blick, welche Klassen für die im Dialog dargestellten Daten verantwortlich sind, dass heißt, dass die Struktur **übersichtlicher** geworden ist. Auch eine **Wiederverwendung** der Klassen als ganze Komponente ist **einfacher** geworden, da man nicht mühselig alle betroffenen Klassen aus unzähligen Packages heraussuchen muss, sondern das ganze Package sofort importieren kann.

Dieses Refactoring konnte dank der Unterstützung seitens IntelliJ IDEA (vergleiche Abschnitt 4.2.2), das einzige der drei evaluierten Tools (siehe Abschnitt 4.2), das erlaubt, Klassen in ein neu zu kreierendes Package zu verschieben (bei den beiden anderen Tools muss man das neue Package bei Bedarf vor dem Refactoring anlegen), **sehr schnell durchgeführt** werden.

5.1.2.3 Erhebung von Metriken vor und nach dem Refactoring

Bei der Betrachtung der Metriken vor und nach dem Refactoring kann man feststellen, dass sich erwartungsgemäß nur zwei davon, nämlich LOC und NOIS, ändern. Anhand der Tabelle 5.6 sieht man die Metrikenänderungen nach den im vorangegangenen Abschnitt beschriebenen Umstrukturierungen in der Package-Struktur. Die Klassen *Massnahmen* und *MassnahmenLiveTable* zeigen keine Veränderungen. In den Klassen *MassnahmenExport* sowie *MassnahmenAdapter* ist jeweils ein fallender LOC- sowie NOIS-Wert zu beobachten, was dadurch zu erklären ist, dass die von diesen Klassen benutzen Klassen und Interfaces nun im selben Package sind und nicht mehr importiert werden müssen. Bei den Klassen *MetaTypeMassnahme*, *MetaTypeMassnahmeImpl*, aber insbesondere *MassnahmenController* sind dagegen steigende Werte dieser Metriken feststellbar, was darauf zurückzuführen ist, dass eine schwache bis mittelstarke Abhängigkeit zu den Klassen der jeweils alten Packages vorhanden ist, weshalb diese nun explizit importiert werden müssen.

Klasse/ Interface	Vorher LOC	Nachher LOC	Vorher NOIS	Nachher NOIS
Massnahmen	78	78	5	5
MassnahmenController	1.327	1.333	40	46
MetaTypeMassnahme	56	57	1	2
MetaTypeMassnahmeImpl	193	194	2	3
MassnahmenLiveTable	474	474	24	24
MassnahmenExport	113	112	14	13
MassnahmeAdapter	508	506	23	21

Tabelle 5.6: Metrikenänderungen bei Verschiebung mehrerer zusammenhängender Klassen in ein gemeinsames Package

5.1.2.4 Häufigkeit der Anwendung

Vor allem auf der Client-Seite weist REX ein großes Potential für dieses Refactoring auf. Die Gründe dafür sind folgende:

- Einige wichtige Klassen sind im Package, dessen Name das Wort ,,test" enthält, untergebracht. Dies entstand aufgrund des Zeitdrucks in einer wichtigen Projektphase, als Klassen, die zuvor lediglich zu Testzwecken entwickelt wurden, nach ihrer positiven Validierung nicht in ein passendes Package verschoben wurden. Da in kürzester Zeit viele Referenzen auf diese Klassen entstanden sind, verzichtete man darauf, die Verschiebung durchzuführen, um nicht viele Klassen bezüglich ihrer Import-Statements anpassen zu müssen.
- Die Package-Struktur der Client-Klassen ist in großen Teilen nicht nach Modulen, sondern nach generellen Funktionalitäten (wie beispielsweise Views, Controller, et cetera) aufgebaut. Dies führt dazu, dass Klassen, die gemeinsam eine Aufgabe erfüllen, über mehrere Packages verstreut sind.

5.1.2.5 Auswirkungen auf die Performanz

Es spielt keine Rolle, ob beispielsweise zwei interagierende Klassen in demselben oder in verschiedenen Packages untergebracht sind. Daher gibt es beim Verschieben von Klassen in ein anderes Package keine Auswirkungen auf die Performanz.

5.1.2.6 Fazit

Die Schlussfolgerung aus den bei der Verschiebung zusammenhängender Klassen in ein gemeinsames Package angestellten Beobachtungen lautet: Dieses Refactoring ist sinnvoll und notwendig; es kann in kürzester Zeit bewerkstelligt werden. Allerdings ist es vor seiner Durchführung notwendig, dass sich die betroffenen Entwickler absprechen, damit unnötige Irritationen verhindert werden, wenn einer der Entwickler seine Klassen nicht mehr an der gewohnten Stelle finden wird.

5.1.3 Bedingung zerlegen

Bei diesem Refactoring geht es darum, lange bedingte Ausdrücke dadurch zu vereinfachen, dass man Teile der Abfragen in eigene, einen Booleschen Wert zurückgebende Methoden mit aussagekräftigen Namen ausgelagert. Insbesondere diejenigen bedingten Anweisungen, die viele *if-else*-Klauseln besitzen und deren eigentliche Abfragen komplexer Natur sind, bieten viel Potential für diese Art von Refactoring.

5.1.3.1 Begründung der Auswahl der Beispielklassen

Zur Veranschaulichung dieses Refactorings wird in den Abschnitten 5.1.3.2 - 5.1.3.3 ein kurzes Beispiel behandelt. Es handelt sich hierbei um einen Ausschnitt der Methode *activateMenuItemsAfterCursorPositionChange()* der Klasse *com.rex.gui.massnahmen.MassnahmenController*. Ausgewählt wurde dieses Beispiel aus folgenden Gründen:

- Die zyklomatische Komplexität (CC, vergleiche Abschnitt 2.1.5) dieser Klasse liegt mit 190 deutlich über dem empfohlenen Wert von 15 und lässt vermuten, dass es in dieser Klasse viele *if*-Klauseln gibt.
- Eine Analyse der *if*-Klauseln ergab, dass insbesondere in der Methode *activateMenuItemsAfterCursorPositionChange()* die bedingten Ausdrücke sehr unübersichtlich sind, weil sie sich über mehrere Zeilen erstrecken.
- Es war auch auffällig, dass nicht nur in der besagten Methode, sondern in einigen anderen Methoden der Klasse wiederholt dieselben Abfragen gemacht wurden.

5.1.3.2 Beschreibung der durchgeführten Änderungen

Im Laufe des Refactorings konnten unter anderem folgende Code-Zeilen

```
if((zeilenNode.getEbeneTyp().intValue() ==
    NodeTypesMassnahme.MASSNAHME_VARIANTE_TYP) &&
   (this.status != ModulType.READ_ONLY))
```

durch

```
if(isVarianteEbene(zeilenNode) && isWritable())
```

ersetzt werden. Weil die neuen Bedingungen **übersichtlicher** als die alten sind, konnte die ganze Methode überarbeitet werden, indem einige der *if-else*-Klauseln zusammengefasst werden konnten. Auch ist es nun **einfacher**, **Änderungen einzuarbeiten**, weswegen sich die **Wartbarkeit** der betroffenen Methode erheblich **verbessert** hat. Wenn man sich nun beispielsweise die neu geschaffene Methode *isVarianteEbene(...)* genauer anschaut, kann man feststellen, dass diese nicht nur an vielen anderen Stellen der Klasse aufgerufen werden kann, um die bedingten Ausdrücke zu vereinfachen, sondern im Zuge des Methode-Verschieben-Refactorings in eine andere Klasse verschoben werden kann.

Leider bietet keines der untersuchten Refactoring-Tools Unterstützung für die Zerlegung von Bedingungen. Insofern ist an dieser Stelle nicht nur mit relativ **hohem Zeitaufwand**

sondern auch mit dem **Entstehen neuer Fehler** zu rechnen. Es empfiehlt sich daher, diese Refactoring-Technik nur dann anzuwenden, wenn man sich ohnehin wegen einer anstehenden Änderung beziehungsweise Fehlerbehebung ausführlich mit einer entsprechenden Code-Stelle beschäftigt.

5.1.3.3 Erhebung von Metriken vor und nach dem Refactoring

Die vor und nach der Durchführung des Bedingung-Zerlegen-Refactorings, bei dem drei bedingte Ausdrücke durch Aufrufe der neu geschaffenen Methoden ersetzt worden sind, erhobenen Metriken zeigen zum Teil deutliche Veränderungen, insbesondere in Bezug auf den LCOM1-Wert, was dadurch zu erklären ist, dass (in diesem Fall) drei neue Methoden hinzukamen, von denen eine auf ein Attribut der Klasse, die restlichen auf keines davon zugreift (siehe Abschnitt 2.1.8). Die nur um eine Zeile erhöhte Länge des Quellcodes ist darauf zurückzuführen, dass zwar drei neue Methoden hinzukamen, die allein schon durch die Methodenrümpfe jeweils mindestens drei zusätzliche Zeilen in Anspruch nehmen, dass es aber durch die kürzeren Namen möglich wurde, nicht nur die jeweiligen bedingten Ausdrücke auf je einer Zeile unterzubringen, sondern auch Code-Abschnitte zusammenzufassen, um so den doppelten Code zu eliminieren. Tabelle 5.7 zeigt die vor und nach dem in Abschnitt 5.1.3.2 beschriebenen Refactoring gemessenen Werte.

Metrik	**Vorher**	**Nachher**
LOC	1.356	1.357
LCOM1	1.614	1.811
NOM	63	66
RFC	766	769

Tabelle 5.7: Bedingung zerlegen: Metrikenänderungen beim MassnahmenController

5.1.3.4 Häufigkeit der Anwendung

Einige Client-Klassen beinhalten durchaus bedingte Ausdrücke, deren Verständnis durch dieses Refactoring vereinfacht werden kann. Allerdings ist die Suche nach den möglichen Kandidaten für eine solche Umstellung sehr zeitaufwendig, weil man die Klassen Zeile für Zeile manuell untersuchen muss.

5.1.3.5 Auswirkungen auf die Performanz

Diese Art von Refactoring wirkt sich negativ auf die Performanz aus. Die dies belegenden Messungen sind im Wesentlichen mit denen bei der Methoden-Extraktion identisch (siehe Abschnitt 5.1.7.5), weil letztendlich ein Code-Abschnitt in eine neue Methode ausgelagert wird, was bei der Ausführung des Programms einem Unterprogramm-Aufruf gleich kommt.

5.1.3.6 Fazit

Zusammenfassend ist hervorzuheben, dass die Zerlegung von Bedingungen zwar der Übersichtlichkeit und der Wartbarkeit einer Anwendung zugute kommt, aber vor allem wegen mangelnder Tool-Unterstützung mit erheblichen Risiken wie zum Beispiel der Fehlerentstehung

und einem nicht zu vernachlässigenden Zeitaufwand (unter anderem für das anschließende Testen) verbunden ist. Deshalb ist die Anwendung dieses Refactorings nur dann ratsam, wenn man sich sowieso aus Wartungs- beziehungsweise Erweiterungsgründen mit der betroffenen Code-Stelle auseinander setzen muss.

5.1.4 Parameterobjekt einführen

Diese Refactoring-Technik dient dazu, lange und unübersichtliche Parameterlisten von Methoden zu vereinfachen. Dabei wird aus der Parameterliste eine Klasse für Datenhaltung mit *get*- und *set*-Methoden kreiert, oder eine bereits vorhandene passende Klasse dazu verwendet, deren Instanz anstelle der alten Parameterliste der Methode übergeben wird.

5.1.4.1 Begründung der Auswahl der Beispielklassen

Gespräche mit Entwicklern sowie das Studium der Änderungshistorie der Maßnahmen-Komponente haben ergeben, dass eine Methode zum Setzen von Zellen-Editoren in einer Tabelle (*mapRow(...)*) im Laufe der Zeit oft geändert werden musste, weil neue Anforderungen hinzukamen. Um diesen Anforderungen gerecht zu werden, wurden der Methode zusätzliche Parameter übergegeben, wodurch die Parameterliste lang geworden ist.

5.1.4.2 Beschreibung der durchgeführten Änderungen

Abbildung 5.1 auf Seite 55 stellt die durchgeführten Umstellungen dar; dabei ist es zu beachten, dass bei den Klassendiagrammen nur die Methoden und Attribute zu sehen sind, deren Signatur sich im Rahmen des Refactorings geändert hat oder die neu hinzugekommen sind. Aus der Parameterliste der *mapRow(...)* Methode der Klasse *com.rex.gui.massnahmen.MassnahmenLiveTable* wurde eine neue Klasse *com.rex.gui.massnahmen.MappingData* geschaffen, die Methoden zum Setzen (*set(...)*) und Lesen (*get()*) der entsprechenden Klassenattribute zur Verfügung stellt. Bei der Methode *mapRow(...)* wurde nicht nur die Parameterliste angepasst, indem eine Instanz der neu

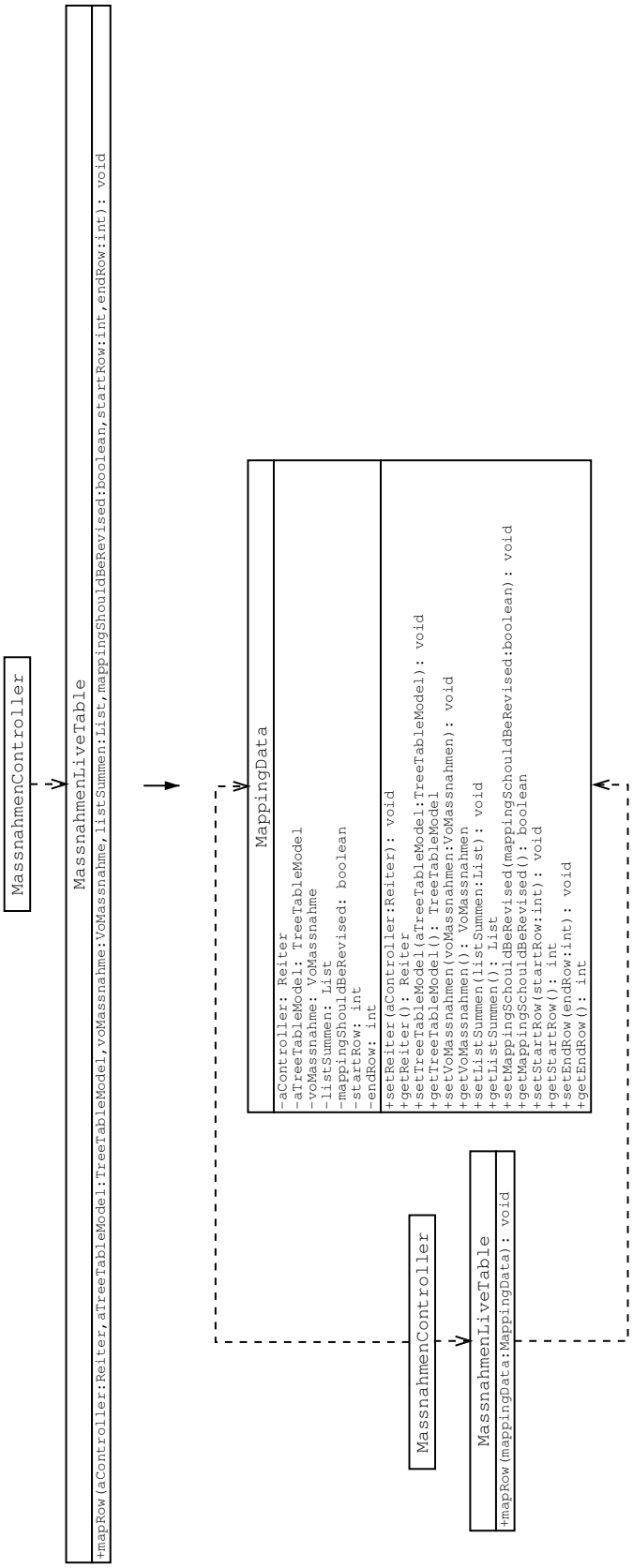

Abbildung 5.1: UML-Diagramm: Parameterobjekt einführen

kreierten Klasse als einziger Parameter fungiert, sondern auch die jeweiligen Zugriffe auf die ehemaligen Parameter durch entsprechende Aufrufe der Getter-Methoden der eingeführten Klasse ersetzt. Vier Methoden der Klasse *com.rex.gui.massnahmen.MassnahmenController* mussten ebenfalls angepasst werden, weil sie alle Aufrufe der Methode *mapRow(..)* enthielten.

Einer der offensichtlichsten Vorteile ist zweifellos die **Minimierung der Verwechslungsgefahr** bei den beiden Übergabeparametern vom Typ *int*. Im Falle einer möglichen Erweiterung der benötigten Übergabeparameter muss man meistens nicht die Parameterliste ändern, was bei gleichen Typen zusätzliche Fehlerquelle darstellen würde, sondern lediglich das übergebene Objekt um erforderliche Attribute und entsprechende *get*- und *set*-Methoden erweitern. Allerdings muss dann auch die zu übergebende Instanz jeweils so geändert werden, dass die benötigten Attribute auch vor der Übergabe gesetzt werden.

Für diese Art von Refactoring war keine Tool-Unterstützung verfügbar, weshalb mit einem **beträchtlichen Zeitaufwand** für seine Durchführung und das anschließende Testen gerechnet werden muss. Vor allem muss vor dem Refactoring überprüft werden, ob eine Klasse, die die benötigten Attribute beinhaltet, bereits vorhanden ist oder erst erstellt werden muss. Außerdem ist die Gefahr, dass eines oder mehrere der benötigten Attribute des Parameterobjekts aus Versehen nicht gesetzt werden, als groß einzustufen.

5.1.4.3 Erhebung von Metriken vor und nach dem Refactoring

In Tabelle 5.8 sind die Metrikenänderungen nach der Einführung des Parameterobjektes festgehalten:

Klasse	**Vorher LOC**	**Nachher LOC**	**Vorher CBO**	**Nachher CBO**	**Vorher RFC**	**Nachher RFC**
MappingData	-	56	-	4	-	14
MassnahmenLiveTable	592	592	38	39	868	876
MassnahmenController	972	1.006	60	61	703	710

Tabelle 5.8: Parameterobjekt einführen: Metrikenänderungen

LOC: In der Klasse

- *com.rex.gui.massnahmen.MappingData* befinden sich jeweils 7 Klassenattribute, 7 Setter- und Getter-Methoden, je 3 Zeilen lang sowie vier Import-Statements; die restlichen 3 Zeilen resultieren aus der Klassen-Deklaration mit den zugehörigen Klammern.
- *com.rex.gui.massnahmen.MassnahmenLiveTable* gibt es keine Veränderung der Zeilenanzahl, weil die ,,alte" Parameterliste durch ein neues Parameterobjekt sowie die Referenzen auf die Parameter durch entsprechende Methodenaufrufe der neuen Klasse ersetzt wurden.
- *com.rex.gui.massnahmen.MassnahmenController* ist das Ansteigen des LOC-Wertes um 34 damit zu erklären, dass in 4 Methoden vor dem jeweiligen Aufruf

der Methode *mapRow(...)* nicht nur ein Objekt der nun für den Übergabeparameter benötigten Klasse instanziiert, sondern auch alle (sieben) erforderlichen Werte gesetzt werden mussten; hinzu kommen noch 2 geschweifte Klammern bei einer *if*-Anweisung, die vorher fehlten, weil eine Zeile mit dem Aufruf von *mapRow(...)* ohne Klammern sein durfte, neun Zeilen (Instanzierung des Übergabeparameters, Setzen der Werte des Übergabeparameters, Zeile mit dem Aufruf von *mapRow(...)*) dagegen nicht.

CBO: In der Klasse

- *com.rex.gui.massnahmen.MappingData* erklärt sich der CBO-Wert von 4 dadurch, dass 4 der 7 Klassenattribute weder einfache Typen (wie beispielsweise *int* oder *boolean*) noch *java.lang.**-Typen sind (und nur diese werden mitgezählt, vergleiche Abschnitt 2.1.7).
- *com.rex.gui.massnahmen.MassnahmenLiveTable* erhöhte sich die CBO-Metrik, weil eine neue Klasse (*com.rex.gui.massnahmen.MappingData*) zusätzlich referenziert wird.
- *com.rex.gui.massnahmen.MassnahmenController* stieg der CBO-Wert um 1 aufgrund der nach dem Refactoring notwendig gewordenen Interaktion mit der neu geschaffenen Klasse.

RFC: In der Klasse

- *com.rex.gui.massnahmen.MappingData* setzt sich der RFC-Wert von 14 aus den je 7 Getter- und Setter-Methoden zusammen.
- *com.rex.gui.massnahmen.MassnahmenLiveTable* mussten nicht nur 7 Getter-Methoden des Parameterobjektes, sondern auch eine *set(...)*-Methode aufgerufen werden.
- *com.rex.gui.massnahmen.MassnahmenController* müssen alle 7 Setter-Methoden des Parameterobjekts gesetzt werden, bevor die *mapRow(...)*-Methode aufgerufen werden darf.

5.1.4.4 Häufigkeit der Anwendung

In REX gibt es sehr wenige Stellen, an denen man dieses Refactoring anwenden kann. Das Aufspüren der Methoden mit langen Übergabeparameterlisten, aber vor allem die Verifizierung, ob man alle diese Parameter in einem bereits vorhandenen Objekt zusammenfassen kann oder eine entsprechende Sammel-Klasse erst erstellen muss, kostet relativ viel Zeit.

5.1.4.5 Auswirkungen auf die Performanz

Die Untersuchung der Auswirkungen auf die Performanz wurde anhand des in Anhang A.1 aufgeführten Beispiels bestehend aus Ersetzung von 4 Übergabeparametern durch Übergabe eines entsprechenden Parameterobjektes angestellt. Dabei konnte eine $2,79$fache Verzögerung bei der Verwendung eines Parameterobjektes statt einer langen Parameterliste beobachtet werden (bei 10.000.000 Testläufen benötigte das Testprogramm 2.913 gegenüber 1.046 Millisekunden), wobei in dieser Zahl der Overhead nicht mit berücksichtigt ist, der erforderlich ist, um eine Instanz des Parameterobjekts anzulegen und mit sinnvollen Daten

zu belegen. Auch muss später der Garbage Collector länger arbeiten, um die nicht mehr benötigten Objekte zu löschen. Mit anderen Worten soll insbesondere bei in Bezug auf Performanz kritischen Stellen auf diese Art von Refactoring besser verzichtet werden.

Was den Speicherplatzbedarf angeht, so dürfte dieser allein schon durch die eventuell notwendige Instanziierung des Parameterobjekts zwar nur geringfügig aber doch ansteigen.

5.1.4.6 Fazit

Die Ersetzung einer langen Parameterliste durch ein entsprechendes Parameterobjekt ist vor allem dann angebracht, wenn als Parameter mehrere Instanzen eines und desselben Typs (zum Beispiel *java.lang.String* oder *int*) übergeben werden, wodurch die Verwechslungsgefahr und somit die Fehleranfälligkeit der Anwendung minimiert wird. Auch ist dieses Refactoring angemessen, wenn davon auszugehen ist, dass die Parameterliste in Zukunft weiter wachsen wird; selbst in diesem Fall ist es allerdings illusorisch zu glauben, dass ein Parameterobjekt verhindert, Änderungen an all jenen Stellen durchführen zu müssen, an denen die betroffene Methode aufgerufen wird, falls die Parameterliste beziehungsweise nach dem Refactoring das Parameterobjekt erweitert wird, denn auch bei einem zu übergebenden Parameterobjekt müssen die entsprechenden Werte vorher gesetzt werden. Angesichts der für das Refactoring zu veranschlagenden Zeit macht es wenig Sinn, dieses immer anzuwenden, nur um die Parameterlisten kurz zu halten; eventuell muss dann eher über eine andere Umstellung wie zum Beispiel Verschieben oder Extraktion einer Methode nachgedacht werden.

5.1.5 Typenschlüssel durch Strategie ersetzen

Unter einem Typenschlüssel versteht man eine oder mehrere Variablen, deren Wert das Verhalten einer Klasse beeinflusst. In manchen Fällen, beispielsweise wenn sich diese Werte während des Lebenszyklus des betroffenen Objektes ändern können, ist es allerdings nicht möglich, die benannte Klasse direkt durch Polymorphismus zu ersetzen. Eine mögliche Lösung dieses Problems bietet das von Gamma et al. entworfene Strategie-Pattern (siehe [GaHeJoVl95]). Bei diesem Refactoring wird eine Klassenhierarchie kreiert, deren Superklasse eine Methode hat, in der im Abhängigkeit von dem Typenschlüssel eine Instanz der entsprechenden Unterklasse zurückgegeben wird. Die Klasse, deren Verhalten vom jeweiligen Typenschlüssel abhängig ist, kann dann beispielsweise im Rahmen der in Abschnitt 5.1.6 vorgestellten Ersetzung einer bedingten Anweisung durch Polymorphismus so umgestellt werden, dass die unterschiedlichen Klassenverhalten in die jeweilige Unterklasse ausgelagert werden können.

5.1.5.1 Begründung der Auswahl der Beispielklassen

Gespräche mit einem Verantwortlichen für eine serverseitige Komponente von REX haben ergeben, dass eine Bean-Klasse zum Teil unterschiedlichen Code ausführt, und zwar abhängig vom Wert eines Attributs, weshalb in dieser Klasse viel mehrfacher Code vorhanden ist. Die Sun-Spezifikation von EJB (siehe [www_j2ee]) gibt Vorgaben bezüglich der Klassenorganisation, die eine Verwendung von Polymorphismus in EJBs selbst ausschließen. Deshalb scheint das im Folgenden vorgestellte Beispiel besonders geeignet, dieses Refactoring zu demonstrieren. Zwar beinhaltet das hier behandelte Beispiel eine kleine Abweichung von der klassischen Variante (vergleiche [Fowler00]), weil ein Interface mit den jeweiligen Typenschlüsseln bereits vorhanden war, so dass statt der Kreation einer Superklasse diese durch entsprechende

Anpassungen des Interfaces erstellt werden kann. Trotzdem lassen sich die im Laufe der Durchführung dieses Refactorings gewonnenen Erkenntnisse auch auf diejenigen Varianten übertragen, bei denen die jeweiligen Typkürzel noch in der Ursprungsklasse sind.

5.1.5.2 Beschreibung der durchgeführten Änderungen

Anhand der Abbildung 5.2 kann man die im Laufe der Umstellung zustande gekommenen Änderungen verfolgen:

- Das bereits vorhandene Interface *com.rex.model.auswertungen.value.AuswertungTypes* wurde in eine abstrakte Klasse umgewandelt, die zwei zusätzliche Methoden enthielt, von denen nur *newType(...)* implementiert werden musste.

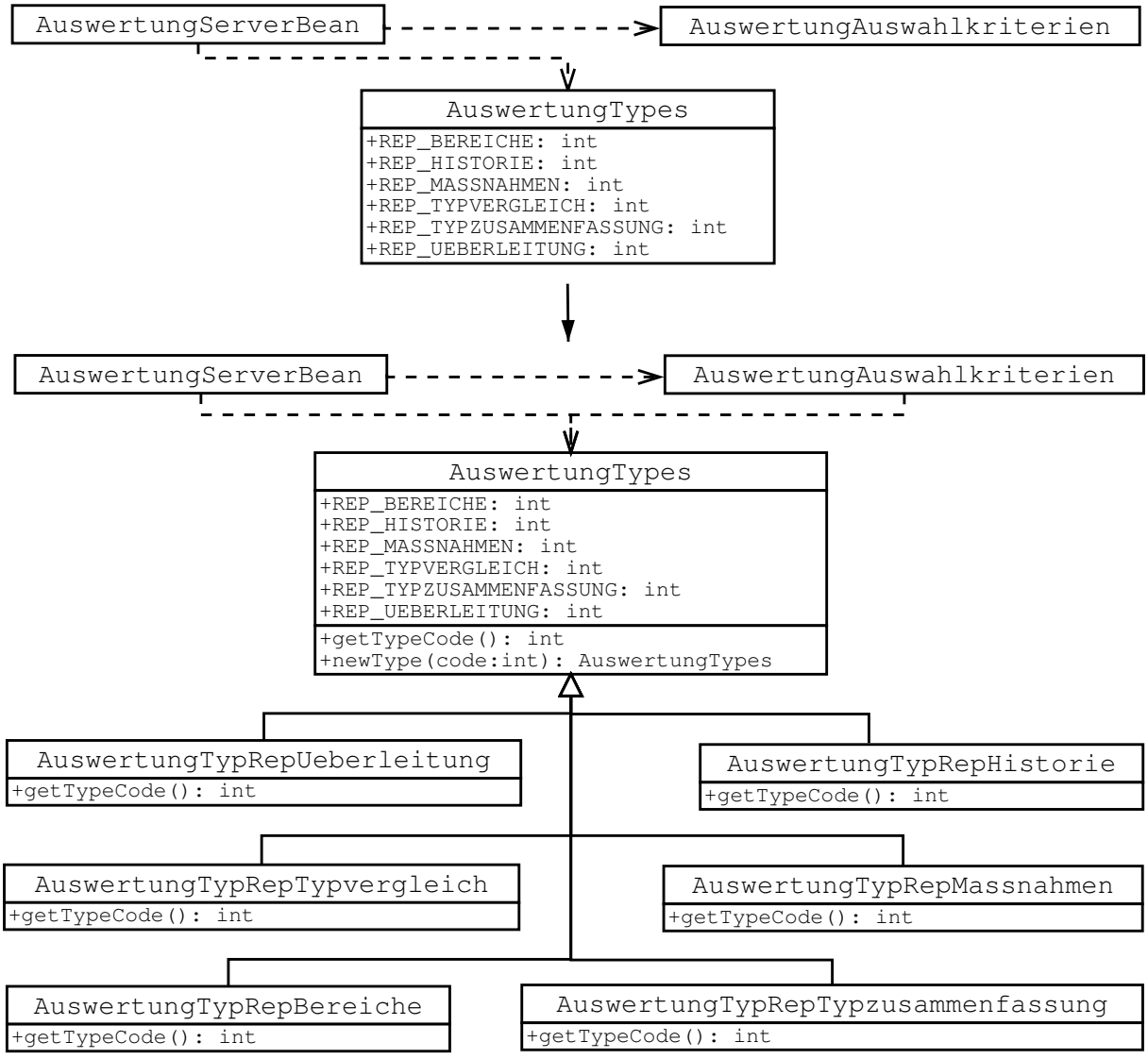

Abbildung 5.2: UML-Diagramm: Typschlüssel durch Strategie ersetzen

- Im Package *com.rex.model.auswertungen.value* wurden die Klassen
  - *AuswertungTypRepBereiche*,
  - *AuswertungTypRepHistorie*,
  - *AuswertungTypRepMassnahmen*,
  - *AuswertungTypRepTypvergleich*,
  - *AuswertungTypRepTypzusammenfassung*,
  - *AuswertungTypRepUeberleitung*

  erstellt, in denen jeweils die Methode *getTypeCode()* implementiert wurde.
- In der Klasse *com.rex.model.auswertungen.value.AuswertungAuswahlkriterien* wurden die Methoden *get-* und *setAuswertungTypID(...)* so angepasst, dass jeweils eine entsprechende Unterklasse der Klasse *com.rex.model.auswertungen.value.AuswertungTypes* kreiert beziehungsweise zurückgegeben wird.

Die Durchführung dieses Refactorings nahm **viel Zeit** in Anspruch, da man angesichts fehlender Tool-Unterstützung sehr viel Code manuell anpassen musste. Der Vorteil der hier vorgestellten Umstellung liegt darin, dass man dank einer nun vorhandenen Vererbungsstruktur auf die durch Polymorphismus zur Verfügung gestellten Möglichkeiten zurückgreifen kann (beispielsweise lassen sich dadurch *switch-case*-Statements vermeiden, was mehr Übersichtlichkeit mit sich bringt).

5.1.5.3 Erhebung von Metriken vor und nach dem Refactoring

In diesem Abschnitt werden die gemessenen Änderungen der Metriken präsentiert.

Klasse/ Interface	**Vorher LOC**	**Nachher LOC**	**Vorher CBO**	**Nachher CBO**
AuswertungAuswahlkriterien	170	170	4	5
AuswertungTypes	9	32	0	1
AuswertungTypRepBereiche	-	8	-	0
AuswertungTypRepHistorie	-	8	-	0
AuswertungTypRepMassnahmen	-	8	-	0
AuswertungTypRepTypvergleich	-	8	-	0
AuswertungTypRepTypzusammenfassung	-	8	-	0
AuswertungTypRepUeberleitung	-	8	-	0
Summe	179	250	4	6

Tabelle 5.9: Typenschlüssel durch Strategie ersetzen: Metrikenänderungen[1]

Aus den Tabellen 5.9 - 5.11 geht hervor, dass

Klasse/ Interface	**Vorher RFC**	**Nachher RFC**	**Vorher CC**	**Nachher CC**
AuswertungAuswahlkriterien	47	51	47	47
AuswertungTypes	0	3	0	2
AuswertungTypRepBereiche	-	2	-	1
AuswertungTypRepHistorie	-	2	-	1
AuswertungTypRepMassnahmen	-	2	-	1
AuswertungTypRepTypvergleich	-	2	-	1
AuswertungTypRepTypzusammenfassung	-	2	-	1
AuswertungTypRepUeberleitung	-	2	-	1

Tabelle 5.10: Typenschlüssel durch Strategie ersetzen: Metrikenänderungen[2]

Klasse/ Interface	**Vorher LCOM2**	**Nachher LCOM2**	**Vorher NOM**	**Nachher NOM**
AuswertungTypes	-	50	-	2
AuswertungTypRepBereiche	-	-	-	1
AuswertungTypRepHistorie	-	-	-	1
AuswertungTypRepMassnahmen	-	-	-	1
AuswertungTypRepTypvergleich	-	-	-	1
AuswertungTypRepTypzusammenfassung	-	-	-	1
AuswertungTypRepUeberleitung	-	-	-	1

Tabelle 5.11: Typenschlüssel durch Strategie ersetzen: Metrikenänderungen[3]

- die Code-Länge (LOC) um knapp 40 % gestiegen ist, was vor allem mit der Einführung von sechs neuen Klassen zusammenhängt.
- sich der CBO-Wert geringfügig vergrößert hat, weil beispielsweise eine neue Abhängigkeit zwischen den Klassen *com.rex.model.auswertungen.value.AuswertungTypes* und *com.rex.model.auswertungen.value.AuswertungAuswahlkriterien* hergestellt wurde.
- die zyklomatische Komplexität unbedeutend erhöht worden ist; dies ist mit dem Hinzukommen neuer (einfacher) Methoden zu erklären.
- die RFC-Metrik erwartungsgemäß gestiegen ist, was
  - zum einen darauf zurückgeführt werden kann, dass in der Klasse *com.rex.model.auswertungen.value.AuswertungAuswahlkriterien* zusätzliche Methoden-Aufrufe notwendig wurden;
  - zum anderen damit verbunden ist, dass in den neuen Klassen sowie im nun zu einer Klasse gewordenen Interface *com.rex.model.auswertungen.value.AuswertungTypes* neue Methoden implementiert worden sind.
- die LCOM2-Metrik in der Klasse *com.rex.model.auswertungen.value.AuswertungTypes* auf 50 gestiegen ist; dies hat nicht nur mit der Umwandlung dieser Klasse aus

einem Interface zu tun, sondern auch mit der Tatsache, dass auf 6 Attribute nur 2 Methoden kommen, wovon eine wiederum als *abstract* deklariert ist.

5.1.5.4 Häufigkeit der Anwendung

Theoretisch könnte man dieses Refactoring an einigen sehr wenigen Stellen in REX durchführen. Allerdings ist in jedem konkreten Fall zu entscheiden, ob eine solche Umstellung mehr Nutzen als Kosten bringt.

5.1.5.5 Auswirkungen auf die Performanz

Die Auswirkungen auf die Performanz und Speicherplatzbedarf sind im Wesentlichen dieselben wie bei der Extraktion einer Klasse (vergleiche Abschnitt 5.1.11.5), weil zur Ausführung betroffener Operationen der alten Klasse neue Klassen instanziiert werden müssen, was Zeit und Speicherplatz kostet.

5.1.5.6 Fazit

Zusammenfassend kann man sagen, dass dieses Refactoring nur dann sinnvoll ist, wenn dadurch der Einsatz von Polymorphismus ermöglicht wird, sonst sind die mit seiner Durchführung verbundenen Kosten (Zeitinvestition für die notwendigen Umstellungen und vor allem für das anschließende Testen) sowie Gefahren (Fehler beim Copy&Paste, da alle Klassen manuell erstellt werden müssen, wofür die Programmierer aus Effizienzgründen einen Teil des neu zu erstellenden Codes kopieren und entsprechend anpassen werden, was dann an der einen oder anderen Stelle doch vergessen wird) nicht zu verantworten.

5.1.6 Bedingte Anweisung durch Polymorphismus ersetzen

Dieses Refactoring wird meistens in Kombination mit der in Abschnitt 5.1.5 beschriebenen Ersetzung eines Typenschlüssels durch Strategie verwendet. Dabei werden die in einer Klasse auf Basis eines Typenschlüssels vorhandenen bedingten Anweisungen durch entsprechende Aufrufe polymorpher Methoden ersetzt.

Im Folgenden werden nicht nur die in REX in diesem Zusammenhang durchgeführten Änderungen und ihre Auswirkungen dokumentiert, sondern auch ein kritischer Blick auf diese Refactoring-Technik im Ganzen geworfen und Empfehlungen bezüglich ihrer Anwendung gegeben.

5.1.6.1 Begründung der Auswahl der Beispielklassen

Für die Durchführung dieses Refactorings wurde das bereits aus Abschnitt 5.1.5.2 bekannte Beispiel benutzt, die Begründung seiner Wahl ist in Abschnitt 5.1.5.1 dargelegt.

5.1.6.2 Beschreibung der durchgeführten Änderungen

Der folgende Code-Abschnitt beschreibt die Ausgangssituation dieses Refactorings und soll durch die Verwendung von Polymorphismus vereinfacht werden:

```
String pdfReport = null;
String auswertung = null;
//krit ist Instanz von AuswertungAuswahlkriterien
Integer auswTypId = krit.getAuswertungTypID();

switch(auswTypId.intValue())
{
    case AuswertungTypes.REP_BEREICHE:
        pdfReport = this.getPDFBereicheBerichtAsString(krit);
        auswertung = "Einkauf/Technologien/Sparten";
        break;
    case AuswertungTypes.REP_MASSNAHMEN:
        pdfReport = this.getPDFMassnahmenBerichtAsString(krit);
        auswertung = "Massnahmen";
        break;
    case AuswertungTypes.REP_TYPVERGLEICH:
        pdfReport = this.getPDFTypVergleichBerichtAsString(krit);
        auswertung = "Typvergleich";
        break;
    case AuswertungTypes.REP_UEBERLEITUNG:
        pdfReport = this.getPDFUeberleitungBerichtAsString(krit);
        auswertung = "Ueberleitung";
        break;
    case AuswertungTypes.REP_HISTORIE:
        pdfReport = this.getPDFHistorieBerichtAsString(krit);
        auswertung = "Historie";
        break;
    case AuswertungTypes.REP_TYPZUSAMMENFASSUNG:
        pdfReport = this.getPDFZusammenfassungTypBerichtAsString(krit);
        auswertung = "Typ-/PV-/Modulbericht";
        break;
}
```

Nach der genauen Code-Analyse stellte sich heraus, dass folgende zusätzliche Schritte notwendig waren, bevor die gewünschte Umstellung durchgeführt werden konnte:

- Die bis dahin als *private* deklarierten Methoden
  - *getPDFBereicheBerichtAsString(...)*,
  - *getPDFHistorieBerichtAsString(...)*,
  - *getPDFMassnahmenBerichtAsString(...)*,
  - *getPDFTypVergleichBerichtAsString(...)*,
  - *getPDFUeberleitungBerichtAsString(...)*,
  - *getPDFZusammenfassungTypBerichtAsString(...)*

  der Klasse *com.rex.appserver.auswertung.bpo.server.AuswertungServerBean* mussten nach außen hin sichtbar gemacht werden, weshalb entsprechende

Veränderungen nicht nur an der Bean, sondern auch am Remote-Interface (*com.rex.appserver.auswertung.bpo.server.AuswertungServer*) durchzuführen waren.

- Der Klasse *com.rex.model.auswertungen.value.AuswertungAuswahlkriterien* wurde eine neue Methode hinzugefügt, die eine Instanz der Klasse *com.rex.model.auswertungen.value.AuswertungTypes* liefert.

Erst nachdem diese Änderungen durchgeführt worden waren, konnte man sich dem eigentlichen Ziel dieses Refactorings widmen, indem in allen sechs Unterklassen der Klasse *com.rex.model.auswertungen.value.AuswertungTypes* die Methoden *getAuswertung()* und *getPDFReport(...)* implementiert wurden. Danach konnte man den ursprünglichen Code-Abschnitt so umformen, dass er folgendes Aussehen bekam:

```
//krit ist Instanz von AuswertungAuswahlkriterien
Integer auswTypId = krit.getAuswertungTypID();
String pdfReport = krit.getAuswertungTyp().getPDFReport(this, krit);
String auswertung = krit.getAuswertungTyp().getAuswertung();
```

Wie man schon der vorhergehenden Beschreibung entnehmen kann, erfordert die Ersetzung einer bedingten Anweisung durch Polymorphismus einen **beträchtlichen Zeitaufwand**, was in diesem konkreten Fall zusammen mit der in Abschnitt 5.1.5 vorgestellten Umstrukturierung mehrere Stunden kostete. Unbestritten ist, dass der neue Code **kompakter** und somit **übersichtlicher** als der alte ist. Auch **entspricht** er eher dem **OO-Ansatz**. Die Frage, ob die Wartung des Codes jetzt einfacher geworden ist, kann nicht eindeutig beantwortet werden, denn trotz besserer Übersichtlichkeit des Codes

- ist es sicherlich schneller, wenn man eine Änderung in einer und derselben Methode (obschon in allen *case*-Stetements) anstatt in vielen verschiedenen Klassen durchzuführen hat.

- ist es nicht ausgeschlossen, dass man eine der durchzuführenden Anpassungen in einer der Methoden der Unterklassen eher vergisst, als wenn man sukzessive innerhalb des jeweiligen *case*-Statements eine entsprechende Änderung durchführt.

5.1.6.3 Erhebung von Metriken vor und nach dem Refactoring

Sowohl vor der Anwendung dieser Refactoring-Technik als auch danach wurden Metriken erhoben, um die Änderungen quantitativ beurteilen zu können. Aus den Tabellen 5.12 - 5.15 geht hervor, dass

- die Code-Länge der Klasse, in der die zu ersetzende bedingte Anweisung untergebracht war (*com.rex.appserver.auswertung.bpo.server.AuswertungServerBean*) zwar nach dem Refactoring zurückgegangen, in allen anderen Klassen aber gestiegen ist, so dass die LOC-Metrik insgesamt gesehen um über 3 % gestiegen ist.

- der CBO-Wert nur in den Klassen, in denen die neu geschaffenen Methoden aus der bedingten Anweisung deklariert oder implementiert worden sind, gestiegen ist, in allen anderen Klassen blieb er naturgemäß unverändert.

- die RFC-Metrik sowie die zyklomatische Komplexität in allen Klassen mit Ausnahme der Klasse *com.rex.appserver.auswertung.bpo.server.AuswertungServerBean* gestiegen ist, was mit der erhöhten Anzahl der Methoden zusammenhängt.

Klasse/ Interface	**Vorher LOC**	**Nachher LOC**	**Vorher CBO**	**Nachher CBO**
AuswertungServer	14	20	6	6
AuswertungServerBean	1309	1282	98	98
AuswertungAuswahlkriterien	170	173	5	5
AuswertungTypes	32	35	1	4
AuswertungTypRepBereiche	8	19	0	3
AuswertungTypRepHistorie	8	19	0	3
AuswertungTypRepMassnahmen	8	19	0	3
AuswertungTypRepTypvergleich	8	19	0	3
AuswertungTypRepTypzusammenfassung	8	19	0	3
AuswertungTypRepUeberleitung	8	19	0	3
Summe	1573	1624	110	131

Tabelle 5.12: Bedingte Anweisung durch Polymorphismus ersetzen: Metrikenänderungen[1]

Klasse/ Interface	**Vorher RFC**	**Nachher RFC**	**Vorher CC**	**Nachher CC**
AuswertungServer	12	18	7	13
AuswertungAuswahlkriterien	51	52	47	48
AuswertungTypes	3	5	2	4
AuswertungTypRepBereiche	2	5	1	3
AuswertungTypRepHistorie	2	5	1	3
AuswertungTypRepMassnahmen	2	5	1	3
AuswertungTypRepTypvergleich	2	5	1	3
AuswertungTypRepTypzusammenfassung	2	5	1	3
AuswertungTypRepUeberleitung	2	5	1	3

Tabelle 5.13: Bedingte Anweisung durch Polymorphismus ersetzen: Metrikenänderungen[2]

- das Hinzufügen einer einzelnen Methode in die Klasse *com.rex.model.auswertungen.value.AuswertungAuswahlkriterien* zu einer Verschlechterung der Kohäsion geführt hat, was dem 5 %-Anstieg des LCOM1-Wertes zu entnehmen ist.
- in der Klasse *com.rex.model.auswertungen.value.AuswertungTypes* eine Versechsfachung des LCOM1-Wertes und ein 50 %-Anstieg des LCOM2-Wertes festzustellen ist, was mit dem Vorhandensein von 6 Attributen, einer implementierten Methode und 3 Methodendeklarationen zu erklären ist.

Klasse/ Interface	Vorher LCOM1	Nachher LCOM1	Vorher LCOM2	Nachher LCOM2
AuswertungAuswahlkriterien	896	936	97	97
AuswertungTypes	1	6	50	75

Tabelle 5.14: Bedingte Anweisung durch Polymorphismus ersetzen: Metrikenänderungen[3]

Klasse/ Interface	Vorher NOM	Nachher NOM	Vorher NOIS	Nachher NOIS
AuswertungServer	7	13	4	4
AuswertungAuswahlkriterien	44	45	1	1
AuswertungTypes	2	4	1	2
AuswertungTypRepBereiche	1	3	0	2
AuswertungTypRepHistorie	1	3	0	2
AuswertungTypRepMassnahmen	1	3	0	2
AuswertungTypRepTypvergleich	1	3	0	2
AuswertungTypRepTypzusammenfassung	1	3	0	2
AuswertungTypRepUeberleitung	1	3	0	2

Tabelle 5.15: Bedingte Anweisung durch Polymorphismus ersetzen: Metrikenänderungen[4]

5.1.6.4 Häufigkeit der Anwendung

Auch dieses Refactoring kann man genauso wie das Typenschlüssel-Durch-Strategie-Ersetzen-Refactoring einige wenige Male in REX einsetzen. Es eignet sich besonders an den Stellen, an denen mehrere gleiche *switch*-Statements vorkommen. Die Einführung einer auf Polymorphismus zurückgreifenden Vererbungsstruktur würde dazu führen, dass man beim Hinzufügen eines neuen *Cases* letztendlich ,,nur" eine neue Unterklasse implementieren müsste, wodurch wiederum gewährleistet wäre, dass man nichts (das heißt keines der *switch*-Statements) vergisst.

5.1.6.5 Auswirkungen auf die Performanz

Bei diesem Refactoring dürfte es zu Performanzverlusten und Nachteilen bezüglich des Speicherplatzverbrauchs kommen, weil

- man zusätzliche Klassen instanziieren muss, um eine Aufgabe zu erledigen.
- die Aufgabe nicht vor Ort (das heißt in dem jeweiligen *case*-Statement), sondern in einer anderen Methode, zudem noch einer anderen Klasse, erledigt wird (vergleiche dazu auch Abschnitte 5.1.7.5 und 5.1.11.5).

5.1.6.6 Fazit

Angesichts des zu veranschlagenden Zeitaufwands ist diese Art von Refactoring nur dann sinnvoll, wenn man mehrere im Code verstreute bedingte Anweisungen hat, die eine und dieselbe Bedingung verwenden. Dadurch könnte die Erweiterung des Codes (also das Hinzufügen weiterer Bedingungen) erleichtert werden, da man nicht mehr den gesamten Code nach bedingten

Anweisungen durchsuchen müsste, um festzustellen, an welchen Stellen die Anpassungen vorgenommen werden sollen, sondern ,,nur noch" eine neue Klasse implementieren müsste, bei der klar ersichtlich ist, welche Methoden sie zu implementieren hat (diese sind als *abstract* in der Superklasse gekennzeichnet). In allen anderen Fällen würde dieses Refactoring mehr Kosten als Nutzen bringen.

5.1.7 Methode extrahieren

Mit Hilfe dieses Refactorings kann man eine Methode in mehrere kleinere Methoden aufspalten. Vor allem dann, wenn eine Methode viele Aufgaben hat, ist es oft ratsam, für jede der Aufgaben eine eigene Methode zu benutzen, wenn diese kleinen Methoden an einer anderen Stelle im Code aufgerufen werden können, um so den mehrfachen Code zu vermeiden.

5.1.7.1 Begründung der Auswahl der Beispielklassen

Für die Demonstration der durch dieses Refactoring hervorgerufenen Änderungen wurden zwei Beispiel-Methoden der Klasse *com.rex.gui.massnahmen.MassnahmenController* ausgewählt:

- *actionPerformed(...)*, weil in dieser Methode mehrmals die gleichen Methoden aufgerufen wurden, um die Cursor-Anzeige zu verändern.
- *addSumToNode(...)*, weil dort dreimal ein gleicher Code-Abschnitt mit lediglich unterschiedlichen Variablen vorkam;

5.1.7.2 Beschreibung der durchgeführten Änderungen

Beim ersten Beispiel konnten zwei Methoden extrahiert werden: eine zum Setzen des Sanduhr-Cursors, eine andere zum Rückkehr in den Anfangszustand des Cursors. Dabei sieht beispielsweise die erste Methode folgendermaßen aus:

```
public void setWaitCursor()
{
    getOwnerComponent().setCursor(
        Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
}
```

Die Methode *actionPerformed(...)*, in der insgesamt 6 verschachtelte Methodenaufrufe durch je einen Methodenaufruf ersetzt werden konnten, ist **übersichtlicher** und **leichter wartbar** geworden, denn jede Anpassung beim Setzen eines passenden Cursors muss nun nicht an je drei, sondern an einer Stelle durchgeführt werden. Da die beiden neuen Methoden zur Cursor-Verwaltung auch von einigen anderen Klassen der Vererbungshierarchie von *com.rex.gui.massnahmen.MassnahmenController* verwendet werden können, würde ihr Verschieben in die Superklasse weitere Vorteile mit sich bringen (siehe Abschnitt 5.1.9).

Beim zweiten Beispiel konnte der Code ebenfalls von Wiederholungen befreit werden, wodurch er kompakter, also auch **übersichtlicher** wurde. Gleichzeitig ist der Quelltext wartbarer geworden, da einige evtl. notwendige Änderung nun nur noch an einer zentralen Stelle durchgeführt werden muss. Ferner wurde durch die Extraktion deutlich, dass die extrahierte Methode in einer anderen Klasse besser untergebracht wäre. Dies kann mit Hilfe

des Methode-In-Andere-Klasse-Verschieben-Refactorings bewerkstelling werden (vergleiche Abschnitt 5.1.11).

Die Durchführung des Refactorings nahm bei den beiden hier vorgestellten Beispielen dank IDE-Unterstützung **relativ wenig Zeit** in Anspruch. Allerdings trifft diese Aussage nicht generell für dieses Refactoring zu, da die besagte Unterstützung bei den in den Abschnitten 4.2.1 - 4.2.3 vorgestellten Tools komplett entfällt, falls beispielsweise ein *return*-Statement innerhalb des zu extrahierenden Code-Fragments vorkommt: dort muss dann manuell ein entsprechender Boolescher Ausdruck eingebaut werden.

5.1.7.3 Erhebung von Metriken vor und nach dem Refactoring

Nachdem im vorangegangenen Abschnitt die Code-Änderungen anhand zweier Beispiele erläutert worden sind, wird hier auf die Metrikenänderungen eingegangen.

Metrik	**Vorher**	**Nachher**
LOC	1.335	1.341
CC	190	192
NOM	62	64
RFC	787	789
LCOM1	1.549	1.684

Tabelle 5.16: Methode extrahieren: Metrikenänderungen nach setWait-/DefaultCursor()-Extraktion in MassnahmenController

Tabelle 5.16 zeigt die messbaren Veränderungen der Metriken nach der Extraktion von *setWait-/DefaultCursor()*-Methoden aus *actionPerformed(...)*. Die Erhöhung der Anzahl der Zeilen der Klasse ist dadurch erklärbar, dass die extrahierten Methoden jeweils aus einer Zeile bestehen (in dieser Ausarbeitung wurde der extrahierte Abschnitt aus Platzgründen auf zwei Zeilen verteilt), das heißt, dass die Code-Stellen, an denen die Ersetzung der alten durch die neuen Methoden-Aufrufe stattfand, von der Länge her nicht kürzer wurden, die Methodenrümpfe der extrahierten Methoden aber hinzukamen. Die CC-, NOM-, RFC- und LCOM1-Metriken sind gestiegen, weil zwei neue Methoden dazugekommen sind. Diese Metriken-Verschlechterungen können aber durch das Methode-In-Superklasse-Verschieben-Refactoring (siehe Abschnitt 5.1.9) in diesem Fall wieder gut gemacht werden.

Metrik	**Vorher**	**Nachher**
LOC	1.357	1.353
CC	190	191
NOM	62	63
RFC	788	789
LCOM1	1.549	1.616

Tabelle 5.17: Methode extrahieren: Metrikenänderungen nach setSummeValue()-Extraktion in MassnahmenController

Die in der Tabelle 5.17 aufgeführten Metrikenwerte vor und nach der Extraktion von *setSummeValue(...)* zeigen wiederholt, dass sich die CC-, NOM-, RFC- sowie LCOM1-Metriken aufgrund der Erhöhung der Methodenanzahl verschlechtern. Das Sinken der LOC-Metrik ist mit der Beseitigung des mehrfachen Quellcodes zu erklären.

Zwar sinkt in den beiden hier vorgestellten Beispielen die CC-Metrik nicht, unter bestimmten Voraussetzungen kann es aber bei *Methode extrahieren* dazu kommen: Wenn im zu extrahierenden Code-Abschnitt beispielsweise eine *if*-Klausel vorkommt, und die extrahierte Methode an mehreren Stellen eingesetzt werden kann, so dass diese *if*-Klausel lediglich in der extrahierten Methode, nicht aber in den betroffenen Code-Abschnitten vorkommen wird.

5.1.7.4 Häufigkeit der Anwendung

Vor allem einige Client- und sehr wenige Server-Klassen können durch die Zerlegung in mehrere Methoden folgendermaßen profitieren:

- Mehrfach vorhandener Code kann dadurch leichter identifiziert und entfernt werden.
- Die Methoden werden übersichtlicher.

Beim Client sind Controller-Klassen, beim Server vor allem einige Enterprise-Java-Beans (EJBs, siehe[www_j2ee]) betroffen, wobei zu beachten ist, dass fast alle EJBs sehr lange Methodenrümpfe besitzen, die jedoch in vielen Fällen wegen der Verwendung von TopLink (siehe [www_toplink]) nicht sinnvoll zerlegt werden können.

5.1.7.5 Auswirkungen auf die Performanz

Die Vermutung, dass durch die Extraktion einer Methode die Gesamtlaufzeit des Programms erhöht wird, wurde anhand der Messungen mit Hilfe der Testklassen (vergleiche Anhang A.2) bestätigt. Ein Methodenaufruf kostet zwar lediglich circa 38 Nanosekunden (bei 1.000.000 Testläufen wurden ohne Methodenextraktion 19, mit - 57 Milisekunden gemessen), prozentual gesehen hat die Extraktion einer Methode eine 300%ige Verlangsamung mit sich gebracht. Je mehr Rechenzeit in der ausgelagerten Methode an sich erforderlich ist, desto weniger fällt ein dadurch zustande gekommener Mehraufwand ins Gewicht; gänzlich verschwinden wird er aber nie.

5.1.7.6 Fazit

Abschließend kann man sagen, dass es durch Methoden-Extraktion in Abhängigkeit von der ausgelagerten Code-Struktur zur Verkleinerungen der LOC- und CC-Werte kommen kann. Die RFC- und LCOM1-Metrik erhöhen sich. Besonders empfehlenswert ist dieses Refactoring damit nur im Zusammenhang mit einem anderen Refactoring, mit dessen Hilfe die extrahierte Methode in eine andere Klasse verschoben wird. Es ist aber weniger sinnvoll, alle Methoden so zu zerlegen, dass sie nur wenige Code-Zeilen lang sind; vielmehr empfiehlt es sich nur, um mehrfachen Code zu vermeiden.

5.1.8 Superklasse extrahieren

Das Ziel dieses Refactorings ist es, einen Teil der Methoden und/oder Attribute in die Superklasse zu extrahieren, um

- durch Vererbung doppelten Code zu vermeiden und
- die Klasse im Allgemeinen übersichtlicher zu gestalten.

5.1.8.1 Begründung der Auswahl der Beispielklassen

Im Folgenden wird die in REX durchgeführte Extraktion einer Superklasse am Beispiel der zentralen Klasse REX_Studio erläutert. Fast alle anderen REX-Client-Klassen befolgen das *Model-View-Controller*-Konzept (siehe [www_mvc]) in einer leicht abgeänderten Form, in der die Controller-Klasse die View erbt. Aus Zeitgründen wurde zunächst nicht das gleiche Konzept in der Klasse REX_Studio angewendet. Diese Klasse diente in ihrer ursprünglichen Version sowohl der Darstellung der GUI-Elemente auf dem Bildschirm als auch der Verarbeitung der Benutzerbefehle und dem Anstoßen weiterer Aktionen als Reaktion auf diese. Zwar besaß die Klasse eine innere Klasse, die als Handler der aus der Menüleiste angestoßenen Befehle fungierte, aber die eigentliche Verarbeitung geschah in der Hauptklasse. Zudem sind die inneren Klassen dafür bekannt, dass man sie nicht *debuggen* kann, was das Auffinden von Fehlern erheblich erschwert.

5.1.8.2 Beschreibung der durchgeführten Änderungen

Wie bereits in Abschnitt 5.1.8.1 beschrieben wurde, bietet es sich an, die innere Klasse in die Hauptklasse (*com.rex.gui.studio.REX_Studio*) zu integrieren, und alles, was die eigentliche Darstellung der Oberflächenelemente anbelangt, in eine neu zu erstellende Superklasse *com.rex.gui.studio.REX* auszulagern. Abbildung 5.3 zeigt dies schematisch.

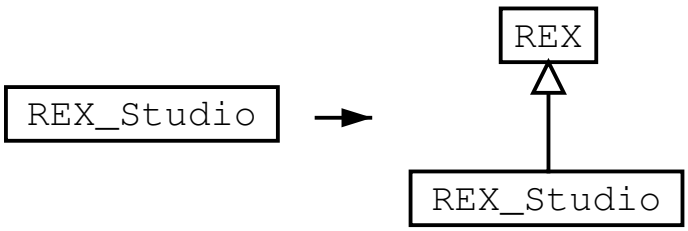

Abbildung 5.3: Klassen-Diagramm: Superklasse extrahieren

Theoretisch wäre eine solche Aufspaltung mit Hilfe der entsprechenden Tools innerhalb weniger Minuten zu erreichen. Leider sah es in der Praxis anderes aus. Für die Extraktion von 66 Methoden und 64 Attributen sowie das anschließende Testen war ein ganzer Tag notwendig, weil man nicht nur die einzelnen Methoden ,,nach oben" verschieben konnte, sondern diese teilweise zuvor zerlegen musste, da sie manchmal beispielsweise sowohl für die Darstellung einer Statuszeile als auch für die Aktionen beim Setzen eines neuen Textes verantwortlich waren.

Die seitens des Tools (in diesem Fall Together ControlCenter, vergleiche Abschnitt 4.2.3) erbrachte Unterstützung erwies sich als bei weitem nicht ausreichend. Vor allem wurden folgende Mängel festgestellt:

- Die alten Import-Statements in REX_Studio wurden nach der Verschiebung von Methoden in die Superklasse nicht automatisch gelöscht, obwohl sie in der Unterklasse nicht mehr benötigt wurden. Auch stellte sich heraus, dass in der neuen Klasse nicht alle notwendigen Importe vorhanden waren, aber auch, dass manche Import-Statements aufgetaucht sind, die weder in der alten Klasse benötigt wurden, noch in der neuen extrahierten Klasse von Nöten waren. Das Aufräumen der Importe per Hand nahm sehr viel Zeit in Anspruch.
- Alle bei der Extraktion der Superklasse zu extrahierenden, aber aufgrund der doch ziemlich langen Methodenliste vergessenen Methoden mussten dann einzeln (zwar auch mit Hilfe des Tools, aber mit einem erheblich höherem Zeitaufwand) in die Superklasse verschoben werden: das heißt, dass es nicht möglich war, im jeweils nächsten Schritt mehr als eine zu verschiebende Methode anzugeben.

Zum Schluss bot sich zwar ein Anblick der beiden Klassen, die nun gut strukturiert waren und klare, also nicht verwobene Funktionen hatten (das heißt, dass die beiden Klassen **übersichtlicher** wurden), aber angesichts des **sehr hohen Zeitaufwands** ist es insbesondere bei reinen Verschönerungen (also wenn dadurch kein mehrfach vorhandener Code beseitigt werden kann) fraglich, ob sich dieses Refactoring wirklich lohnt.

5.1.8.3 Erhebung von Metriken vor und nach dem Refactoring

Nachdem die durchgeführten Änderungen bereits beschrieben worden sind, ist die Analyse der Metrikenänderungen von Interesse. Anhand der Tabellen 5.18 - 5.21 auf den Seiten 71 - 72 kann man die Metrikenänderungen beim *com.rex.gui.studio.REX_Studio* bei der Integration der inneren Klasse (*IvjEventHandler*) und der Extraktion der Superklasse *com.rex.gui.studio.REX* sehen.

Klasse/ Interface	**Vorher LOC**	**Nachher LOC**	**Vorher NOIS**	**Nachher NOIS**
REX	-	1.153	-	23
REX_Studio	3.287	2.063	42	49
IvjEventHandler	70	-	42	-
Summe	3.357	3.216	84	72

Tabelle 5.18: Superklasse extrahieren + innere Klasse integrieren: Metrikenänderungen[1]

Die Metrikenänderungen haben folgende Ursachen:

- Die LOC-Metrik ist gesunken, was mit der Reformatierung des Quellcodes seitens des Tools (in diesem Fall Together ControlCenter, siehe Abschnitt 4.2.3) während des Refactorings zu tun hat.

Klasse/ Interface	**Vorher NOM**	**Nachher NOM**	**Vorher LCOM1**	**Nachher LCOM1**
REX	-	66	-	2.143
REX_Studio	180	124	16.395	7.809
IvjEventHandler	9	-	-	-
Summe	189	190	16.395	9.952

Tabelle 5.19: Superklasse extrahieren + innere Klasse integrieren: Metrikenänderungen[2]

- Die Anzahl der Importe einer Inner-Klasse entspricht der Anzahl der Importe der Klasse, die diese beheimatet. Eine Steigerung des NOIS-Wertes vor allem in der Klasse *com.rex.gui.REX_Studio* ist damit zu erklären, dass Together ControlCenter beim Refactoring, in dessen Verlauf einige der Methode immer wieder zwischen der Super- und Unterklasse verschoben wurden, alle notwendigen Importe explizit hinzufügt, die alten (nach dem Refactoring in der Klasse nicht mehr erforderlichen) aber nicht automatisch entfernt. Außerdem ergänzt das Tool beim Verschieben einer Methode die Liste der Importe in der neuen Klasse mit denjenigen Import-Statements, die nicht benötigt werden, weil der Klassenname in der Methode voll ausgeschrieben ist (das heißt, dass sie bereits die Package-Angaben beinhaltet).

Klasse/ Interface	**Vorher RFC**	**Nachher RFC**	**Vorher CC**	**Nachher CC**
REX	-	435	-	130
REX_Studio	769	733	366	271
IvjEventHandler	44	-	35	-
Summe	813	1.168	401	401

Tabelle 5.20: Superklasse extrahieren + innere Klasse integrieren: Metrikenänderungen[3]

Klasse/ Interface	**Vorher CBO**	**Nachher CBO**	**Vorher NOA**	**Nachher NOA**
REX	-	20	-	64
REX_Studio	132	118	101	36
IvjEventHandler	10	-	0	-
Summe	142	138	101	100

Tabelle 5.21: Superklasse extrahieren + innere Klasse integrieren: Metrikenänderungen[4]

- Die Summe der Attribute ist dadurch um 1 kleiner geworden, dass ein Attribut mit der Instanz der früheren Inner-Klasse nach deren Integration nicht mehr benötigt wird.
- Die Gesamtanzahl der Methoden stieg um 1, weil eine Methode zerlegt werden musste, da sie sowohl reine Darstellungsaufgaben hatte als auch auf die Benutzereingaben reagierte und damit zum Teil in der alten Klasse bleiben musste.

- Zwar blieb die Summe der CC-Metriken auf dem alten Niveau, aber in der Klasse *com.rex.gui.studio.REX_Studio* fiel sie, was geringere Komplexität und bessere Testbarkeit mit sich bringt.
- Der gefallene CBO-Wert deutet auf eine bessere Trennung der Klassenaufgaben hin.
- Die gestiegene Gesamtsumme des RFC-Wertes kann dadurch erklärt werden, dass bei seiner Ermittlung in *com.rex.gui.studio.REX_Studio* auch alle Methoden (NOM) der Superklasse mitgezählt werden. Das Sinken der RFC-Metrik in *com.rex.gui.studio.REX_Studio* hängt damit zusammen, dass die Methoden, die von den in die Superklasse verschoben Methoden aufgerufen werden, nicht mehr mitgezählt werden.
- Die beim LCOM1-Wert zu beobachtende Beinahehalbierung ist mit der besseren Kapselung zu begründen, da zum Beispiel diejenigen Attribute und Methoden, die für die graphische Darstellung zuständig sind, nun von denjenigen Attributen und Methoden, die Events verarbeiten, getrennt sind.

5.1.8.4 Häufigkeit der Anwendung

Dieses Refactoring kann in einigen wenigen Fällen in Client-Klassen durchgeführt werden, vor allem um weitere Stufen in die bereits existierende Vererbungshierarchie einzuführen und somit mehrfachen Code, insbesondere bei *Views*, zu vermeiden. Da es bei REX mehr als 50 verschiedene Dialoge gibt, ist die Suche nach Ähnlichkeiten allerdings als sehr zeitaufwendig einzustufen.

5.1.8.5 Auswirkungen auf die Performanz

Beim Instanziieren einer Klasse werden auch die Konstruktoren aller ihrer Superklassen aufgerufen, die im Hintergrund mit instanziiert werden, was natürlich Zeit und zusätzlichen Speicherplatz kostet. Der Aufruf einer sich in der Superklasse befindlichen Methode ist langsamer als der Aufruf einer in derselben Klasse untergebrachten Methode (siehe dazu Abschnitt 5.1.9.5).

5.1.8.6 Fazit

Dieses Refactoring sorgt für bessere Übersichtlichkeit und eventuell für Vermeidung von mehrfachem Code, falls von der Superklasse mehrere Klassen erben können. Ansonsten ist von diesem Refactoring eher abzuraten, da der zu investierende Zeitaufwand enorm ist, und es sich nicht lohnt, eine Superklasse zu extrahieren, nur weil man meint, dass vielleicht irgendwann eine weitere Klasse hinzukommen könnte, die auch die gleiche Superklasse benutzt. Insbesondere bei einer Entwicklungsgeschichte, die sich vorher wenig um Polymorphismus und Vererbung gekümmert hat, ist davon auszugehen, dass der vorhandene Quellcode schon so spezialisiert ist, dass kaum eine Verallgemeinerung ohne größere Umstellungen möglich ist.

5.1.9 Methode in die Superklasse verschieben

Auch dieses Refactoring dient vor allem der Vermeidung von mehrfach vorhandenem Code. Funktionen, die mehreren Unterklassen gemein sind, können in die Superklasse verschoben werden.

5.1.9.1 Begründung der Auswahl der Beispielklassen

In Abschnitt 5.1.7 wurde die Extraktion zwei kurzer Methoden (*setDefaultCursor()* und *setWaitCursor()*) aus der *actionPerformed(...)*-Methode der Klasse *com.rex.gui.massnahmen.MassnahmenController* beschrieben. Ihre Superklasse *com.rex.gui.framework.dialog.Reiter* besitzt weitere Unterklassen (zum Beispiel *com.rex.gui.controller.StatusblattController* sowie *com.rex.gui.controller.UeberleitungController*), die diese beiden Methoden verwenden könnten. Aus diesem Grund erscheint das Verschieben der ausgewählten Methoden zur Cursor-Steuerung in die Superklasse sinnvoll.

5.1.9.2 Beschreibung der durchgeführten Änderungen

Abbildung 5.4 zeigt einen Ausschnitt aus dem Klassen-Diagramm der betroffenen Super- und Unterklasse vor und nach dem Refactoring.

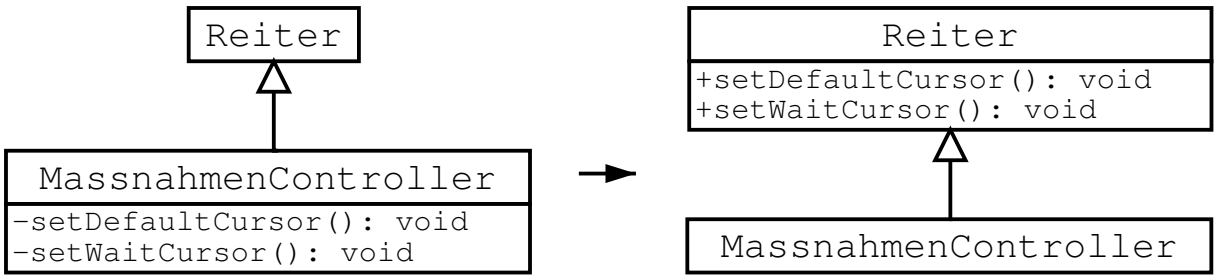

Abbildung 5.4: Klassen-Diagramm: Methoden in Superklasse verschieben

Der Hauptvorteil dieses Refactoring liegt darin, dass die sich nun in der Superklasse befindlichen Methoden auch von anderen Unterklassen benutzt werden können, und somit der Quelltext aufgrund der Vermeidung von mehrfachem Code **wartbarer** wird.

Zwar ist der eigentliche für die Durchführung dieses Refactorings notwendige **Zeitaufwand** dank IDE-Unterstützung (in diesem Fall Together ControlCenter, vergleiche Abschnitt 4.2.3) eher als **gering** einzuschätzen, die notwendigen Anpassungen in anderen Unterklassen, die von der so erweiterten Superklasse abgeleitet sind, können aber **zeitaufwendig** sein, vor allem dann, wenn diese von anderen Entwicklern programmiert worden sind. Daher ist in diesem Fall eine Absprache zwischen allen betroffenen Entwicklern besonders sinnvoll.

5.1.9.3 Erhebung von Metriken vor und nach dem Refactoring

Wenn man sich die Metriken vor und nach dem Refactoring anschaut (siehe Tabelle 5.22), stellt man fest, dass durch das Verschieben zweier Methoden in die Superklasse

- die LOC-, NOM- sowie CC-Metriken in der Unterklasse erwartungsgemäß gesunken und in der Superklasse um je denselben Wert gestiegen sind;
- der CBO-Wert in der Unterklasse um 1 gesunken ist, weil die Klasse *java.awt.Cursor* nicht mehr benötigt wird; der unveränderte Wert dieser Metrik in der Superklasse ist

damit zu erklären, dass alle Klassen, die in den beiden verschobenen Methoden referenziert werden, dort bereits in anderen Methoden verwendet worden sind;

- die RFC-Metrik der Superklasse wegen des Hinzukommens der zwei Methoden um 2 gestiegen ist; die RFC-Metrik der Unterklasse dagegen lediglich um 1 kleiner geworden ist, weil zwar die Anzahl der Methoden der Superklasse, nicht aber die Anzahl der in diesen Methoden aufgerufenen „fremden" Methoden in den RFC-Wert der Unterklasse mit einfließt;
- ungleich starke Vergrößerungen beziehungsweise Verkleinerungen der LCOM1-Metriken der Super- beziehungsweise Unterklasse hervorgerufen werden, was mit der unterschiedlichen Anzahl an Methoden und von diesen verwendeten Attributen zusammenhängt.

Metrik	**Reiter**	**MassnahmenController**	**Summe**
Vorher LOC	450	1.341	1.791
Nachher LOC	456	1.335	1.791
Vorher NOM	38	64	102
Nachher NOM	40	62	102
Vorher CBO	21	85	106
Nachher CBO	21	84	105
Vorher CC	64	192	256
Nachher CC	66	190	256
Vorher RFC	466	789	1.255
Nachher RFC	468	788	1.256
Vorher LCOM1	861	1.684	2.545
Nachher LCOM1	948	1.549	2.597

Tabelle 5.22: Metrikenänderungen bei Verschiebung zweier Methoden in die Superklasse

5.1.9.4 Häufigkeit der Anwendung

In REX gibt es sehr wenige Fälle, in denen man eine Methode in eine bereits existierende Superklasse sinnvoll, das heißt, dass dadurch beispielsweise mehrfach vorhandener Code eliminiert werden kann, verschieben kann. Dies ist aus folgenden Gründen der Fall:

- bereits vorhandene Ausnutzung des Polymorphismus bei Value-Objekten;
- Implementierung der Server-Klassen mit der *J2EE*-Technologie von Sun (vergleiche [www_j2ee]);
- die Tatsache, dass die Implementierung der Client-Klassen auf einem leicht abgeänderten MVC-Konzept (siehe Abschnitt 5.1.8.2) basiert, führt dazu, dass die Controller-Funktionalitäten nicht ohne eine vorhergehende Verflachung der Vererbungshierarchie verallgemeinert werden können, die seinerseits sehr zeitintensiv und fehlerbehaftet sein würde.

5.1.9.5 Auswirkungen auf die Performanz

Die Performanzmessungen wurden mit Hilfe der in Anhang A.2 aufgeführten Klassen vorgenommen. Dabei stellte sich heraus, dass der Aufruf einer Methode aus der Superklasse eine Verlangsamung von knapp 16 % mit sich brachte (für 1.000.000 Testläufe wurden 19 gegenüber 22 Millisekunden gebraucht).

5.1.9.6 Fazit

Meistens sind für die Verschiebung einer Methode in die Superklasse gute Kenntnisse aller ihrer Unterklassen notwendig, damit man nur diejenigen Methoden in die Superklasse verschiebt, die von mehr als einer Klasse verwendet werden können. Vor allem dann, wenn die Klassen von unterschiedlichen Entwicklern stammen, ist daher eine Absprache oder ein längeres Studium der ,,fremden" Klassen unumgänglich. Der Vorteil dieses Refactorings liegt in der Eliminierung des mehrfachen Codes und der damit verbundenen Erleichterung bei der Wartung der Klassen. Der für dieses Refactoring notwendige Zeitaufwand kann gering bis mittelgroß sein, in Abhängigkeit davon, ob die betroffene Methode auch in einer anderen Klasse bereits vorhanden ist, oder ob sie dort erst mühsam extrahiert werden muss. Falls die anderen Unterklassen der dieses Refactoring vornehmenden Person unbekannt sind, ist mit zusätzlichem Zeitaufwand zu rechnen, bis die betroffenen Code-Stellen gefunden worden sind.

5.1.10 Klasse extrahieren

Ziel dieses Refactoring ist es, eine Klasse in zwei oder mehr Klassen zu zerlegen, um die von ihr zur Verfügung gestellten Funktionalitäten beziehungsweise die von ihr zu erledigenden Aufgaben besser kapseln und leichter warten zu können.

Im Folgenden wird anhand eines Beispiels demonstriert, welche Veränderungen die Extraktion einer Klasse mit sich bringen kann. Ferner wird hier der Frage nachgegangen, was dabei beachtet beziehungsweise in Kauf genommen werden muss.

5.1.10.1 Begründung der Auswahl der Beispielklassen

Die schon von anderen Beispielen her bekannte Klasse *com.rex.gui.massnahmen.MassnahmenController* dient primär der Arbeit mit einer Tabelle, in der bestimmte Datensätze angezeigt und bearbeitet werden können. So ,,erfährt" diese Klasse über *Listener*, dass der Benutzer des Dialogs einen Befehl zur Ausführung gegeben hat oder sich in einer Tabellenzelle im Editier- beziehungsweise Kontextmenü-Modus befindet. Eine manuelle Analyse des Quelltextes dieser Klasse ergab, dass sie beispielsweise über zwei Methoden verfügt, die das minimale beziehungsweise maximale Element eines Integer-Arrays bestimmen. Diese Methoden müssen nicht unbedingt in dieser Klasse untergebracht sein. Es wäre ggf. vorteilhaft, sie in eine dafür prädestinierte Klasse zu verschieben, die eventuell auch an anderen Stellen benutzt werden kann.

5.1.10.2 Beschreibung der durchgeführten Änderungen

Abbildung 5.5 zeigt ein Klassen-Diagramm, das die Extraktion zweier Methoden in die neue Klasse *Util* veranschaulicht.

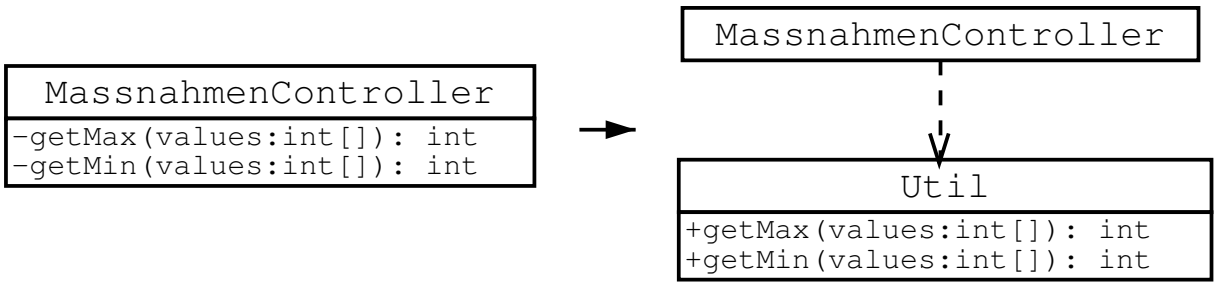

Abbildung 5.5: Klassen-Diagramm: Klasse extrahieren

Die Klasse ist *com.rex.gui.massnahmen.MassnahmenController* damit etwas **übersichtlicher** geworden ist, da sie über weniger Methoden verfügt, und sich ihre Methoden nur mit den eigentlichen Aufgaben der Klasse beschäftigen. Andererseits kann man die von der Klasse *com.msg.framework.util.Util* zur Verfügung gestellten Methoden auch leichter an einer anderen Stelle benutzen (vor allem, weil sie von anderen Entwicklern eher gefunden werden).

Die für die Refactoring-Durchführung benötigte **Zeit** ist dank IDE-Unterstützung (in diesem Fall IntelliJ IDEA, vergleiche Abschnitt 4.2.2) als **nicht allzu hoch** einzuschätzen. Natürlich darf man nicht vergessen, dass man vor der Extraktion einer Klasse

- die zu extrahierenden Methoden festzulegen hat, wozu die ganze Klasse entsprechend begutachtet werden muss;
- zu überprüfen hat, ob eine Klasse, in die die ausgewählten Methoden semantisch passen würden, bereits existiert;
- im Fall, dass keine bereits existierende passende Klasse gefunden werden konnte, ein Package aussuchen oder eventuell auch erst anlegen muss, in dem die neue Klasse unterzubringen ist.

5.1.10.3 Erhebung von Metriken vor und nach dem Refactoring

In diesem Abschnitt werden die Metriken vor und nach der Extraktion einer Klasse mit zwei Methoden aus der Klasse *com.rex.gui.massnahmen.MassnahmenController* diskutiert (siehe Tabelle 5.23).

Die Erhöhung des CBO- und NOIS-Wertes in der Klasse *com.rex.gui.massnahmen.MassnahmenController* ist durch die zusätzlich dazu gekommene Klasse *com.msg.framework.utils.Util* erklärbar. Die zyklomatische Komplexität der ursprünglichen Klasse ist um 6 Punkte gesunken, weil ein Teil der Logik nun in der Hilfsklasse ist. Auffallend ist auch die Verkleinerung des LCOM1-Wertes, was dadurch zustande gekommen ist, dass die Klasse *com.rex.gui.massnahmen.MassnahmenController* weniger Methoden hat, die eine jeweils unterschiedliche Anzahl der Klassen-Attribute benutzen.

Metrik	MassnahmenController	Util	Summe
Vorher LOC	1.278	-	1.278
Nachher LOC	1.253	29	1.282
Vorher NOM	68	-	68
Nachher NOM	66	2	68
Vorher CBO	77	-	77
Nachher CBO	78	0	78
Vorher CC	179	-	179
Nachher CC	173	6	179
Vorher RFC	778	-	778
Nachher RFC	778	2	780
Vorher LCOM1	1.976	-	1.976
Nachher LCOM1	1.833	-	1.833
Vorher NOIS	81	-	81
Nachher NOIS	82	0	82

Tabelle 5.23: Metrikenänderungen bei Extraktion einer Klasse

5.1.10.4 Häufigkeit der Anwendung

Die Extraktion einer Klasse kann in REX an einigen Stellen angewandt werden. Auffällig ist vor allem die Tatsache, dass sowohl auf dem Client als auch auf dem Server viele Berechnungen (wie beispielsweise Ermitteln eines Kehrwertes, Maximums oder Minimums) direkt in der jeweils betroffenen Klasse gemacht werden, obwohl sie nur am Rande etwas mit der wirklichen Aufgabe der Klasse zu tun haben. In diesen Fällen ist es empfehlenswert, die jeweils betroffenen Methoden in eine neu zu erstellende Klasse auszulagern, damit sie auch in anderen Klassen benutzt werden können.

5.1.10.5 Auswirkungen auf die Performanz

Die Extraktion einer Klasse wirkt sich negativ auf die Performanz aus, denn es kommen zusätzliche Kosten (sowohl Zeit als auch Speicherplatz) für die Instanziierung einer neuen Klasse und das eventuell notwendige Aufräumen mit der Garbage Collection hinzu. Der Aufruf einer Methode, die sich in einer anderen Klasse befindet, hat dagegen keine Auswirkungen auf die Performanz (vergleiche dazu Abschnitt 5.1.11.5).

5.1.10.6 Fazit

Zusammenfassend kann man sagen, dass dieses Refactoring an sich mit einem geringen bis mittelgroßen Zeitaufwand verbunden ist, kaum Risiken birgt, jedoch einen eventuell hohen Kommunikations- beziehungsweise Dokumentationsaufwand mit sich bringt. Die Übersichtlichkeit und die Kapselung der Klassen werden verbessert.

5.1.11 Methode in eine andere Klasse verschieben

Auch dieses Refactoring dient der besseren Kapselung von Klassen. Mit seiner Hilfe werden Methoden in eine andere, bereits vorhandene Klasse verschoben, in der sie semantisch gesehen

günstiger platziert sind.

5.1.11.1 Begründung der Auswahl der Beispielklassen

Im Laufe der manuellen Analyse der Methode

checkPrioritaeten(KomponenteMassnahmeValue kmv),

die sich in der Klasse *com.rex.gui.massnahmen.MassnahmenController* befand, konnte festgestellt werden, dass diese Methode, deren Aufgabe es ist, die Korrektheit einiger Benutzereingaben zu überprüfen, keine Attribute der sie beheimatenden Klasse benötigte und nur auf einer Instanz der Klasse *com.rex.model.massnahme.value.KomponenteMassnahmeValue* operierte. Es lag also nahe, diese Methode in diejenige Klasse zu verschieben, deren Attribute und Methoden sie für die Bewältigung ihrer Aufgabe verwendete.

5.1.11.2 Beschreibung der durchgeführten Änderungen

Das hier vorgestellte Beispiel für dieses Refactoring lässt sich am einfachsten anhand von Abbildung 5.6 veranschaulichen.

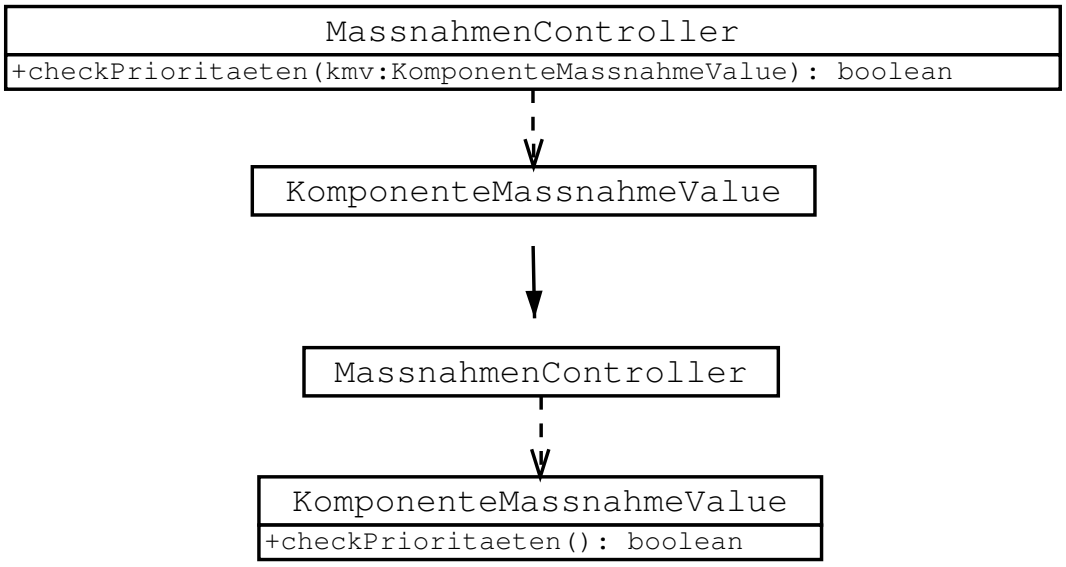

Abbildung 5.6: Klassen-Diagramm: Methode verschieben

Der Hauptvorteil dieser Methodenverschiebung besteht sicherlich darin, dass aus der Klasse *com.rex.gui.massnahmen.MassnahmenController* eine Methode, die nicht direkt mit der Aufgabe der Klasse etwas zu tun hatte, entfernt wurde. Dagegen findet man diese Methode nun in der Klasse *com.rex.model.massnahme.value.KomponenteMassnahmeValue*, auf deren Attribut sie zugreift. Somit sind die beiden Klassen nicht nur **übersichtlicher**, sondern theoretisch auch **wiederverwendbarer** geworden, da nun diese Methode zur Plausibilitäts-Prüfung leichter gefunden werden kann. Allerdings ist es fraglich, ob eine so

spezielle Methode, welche die nur in diesem konkreten Fall getätigten Benutzereingaben überprüft, an einer anderen Stelle ihre Wiederverwendung finden kann.

Bezüglich des für die Durchführung dieses Refactorings erforderlichen **Zeitaufwands** kann man sagen, dass er sich **in Grenzen hält**. Die untersuchten Refactoring-Tools unterstützen eine Verschiebung von Methoden in eine andere Klasse nur unzureichend, so dass neben dem Auffinden der zu verschiebenden Methode noch sehr viel Handarbeit notwendig ist.

5.1.11.3 Erhebung von Metriken vor und nach dem Refactoring

Anhand von Tabelle 5.24 sieht man die Metrikenänderungen am Beispiel, das im vorhergehenden Abschnitt beschriebenen worden ist.

Metrik	**KomponenteMassnahmeValue**	**MassnahmenController**
Vorher LOC	82	1.323
Nachher LOC	102	1.303
Vorher NOM	11	68
Nachher NOM	12	67
Vorher CBO	3	80
Nachher CBO	4	79
Vorher CC	21	188
Nachher CC	25	184
Vorher RFC	133	783
Nachher RFC	137	780
Vorher LCOM1	48	1.994
Nachher LCOM1	60	1.922
Vorher LCOM2	79	93
Nachher LCOM2	81	93

Tabelle 5.24: Metrikenänderungen bei Verschiebung einer Methode in eine andere Klasse

Man sieht sofort, dass sich die Metriken LOC, NOM, CBO, CC und - mit einer leichten Abweichung - auch RFC der beiden betroffenen Klassen um jeweils denselben Wert verkleinert (bei der Klasse, in der die Methode vor der Verschiebung untergebracht war) beziehungsweise vergrößert (bei der Klasse, in der sich die Methode nach der Verschiebung befindet) haben. Beim LCOM1-Wert beobachtet man dagegen ein ungleich höheres Absinken in der „alten" im Vergleich zum Wachstum in der „neuen" Klasse. Diese Tatsache lässt sich mit Hilfe der Definition von LCOM1 erklären (siehe Abschnitt 2.1.8), weil in der Klasse

- *com.rex.model.massnahme.value.KomponenteMassnahmeValue* vor dem Refactoring 4 Attribute und 11 Methoden, nach dem Refactoring 4 Attribute und 12 Methoden
- *com.rex.gui.massnahmen.MassnahmenController* vor dem Refactoring 14 Attribute und 68 Methoden, nach dem Refactoring 14 Attribute und 67 Methoden

vorhanden sind, wobei die zu verschiebende Methode nur auf eines der Attribute der Klasse *com.rex.model.massnahme.value.KomponenteMassnahmeValue* zugreift.

Methode in eine andere Klasse verschieben kann manchmal zu Verkleinerung der Gesamtsumme der RFC-Metriken führen, und zwar dann, wenn wenigstens eine Remote-Methode, die aus der verschobenen Methode heraus aufgerufen worden ist, in der ,,alten" Klasse von keiner weiteren Methode aufgerufen werden, in der ,,neuen" Klasse dagegen bereits von anderen Methoden aufgerufen wird oder ein Bestandteil der neuen Klasse ist.

5.1.11.4 Häufigkeit der Anwendung

Die Verschiebung einer Methode in eine andere Klasse ist wohl eine der Refactoring-Techniken, die in REX am häufigsten angewandt werden kann, weil fast alle Klassen mit Ausnahme der Value-Objekte Methoden beinhalten, die Operationen auf Objekten anderer Klassen ausführen. Dies kam vor allem dadurch zustande, dass Entwickler schnell eine neue Funktionalität hinzufügen mussten, ohne genau prüfen zu können, ob Teile der zu implementierenden Erweiterungen besser in eine andere Klasse passen würden.

5.1.11.5 Auswirkungen auf die Performanz

Mit Hilfe der in Anhang A.2 aufgeführten Klassen wurden Performanz-Messungen vor und nach der Verschiebung einer Methode in eine andere Klasse durchgeführt. Dabei konnten keine Verzögerungen beziehungsweise Beschleunigungen festgestellt werden.

5.1.11.6 Fazit

Wenn man nun alle Erkenntnisse, die im Zusammenhang mit diesem Refactoring stehen, zusammenfasst, kann man sagen, dass die Vorteile im Wesentlichen bessere Übersichtlichkeit und erhöhte Wiederverwendbarkeit sind. Zwar unterstützen entsprechende Tools die Refactoring-Durchführung nur mangelhaft, man kann aber trotzdem relativ schnell die eigentliche Methodenverschiebung bewerkstelligen. Die meiste benötigte Zeit wird dafür verwendet, die Methoden zu bestimmen, die in einer aktuellen Klasse fehl am Platz sind, und eine neue Klasse für diese Methoden zu finden.

5.2 Beurteilung des Ergebnisses

In den Abschnitten 5.1.1 bis 5.1.11 wurden einzelne Refactoring-Techniken unter die Lupe genommen, um die mit ihrer Hilfe erreichbaren Code-Verbesserungen, den für ihre Durchführung benötigten Zeitaufwand und die nach ihrer Anwendung eventuell zu beobachtenden Metriken- sowie Performanz-Änderungen besser beurteilen zu können. Im Folgenden wird der Frage nachgegangen, wie hoch die Kosten sind, und welche Vorteile es mit sich bringt, wenn man zum Beispiel ein ganzes REX-Modul bestehend aus 12 Client-Klassen dem Refactoring unterzieht.

Mit Hilfe einer zeilenweisen Untersuchung des Quellcodes wurden die jeweils passenden und angemessenen Refactoring-Techniken angewandt. Als erstes wurden viele betroffene Klassen in ein gemeinsames Package verschoben. In jeder einzelnen Klasse wurden Methoden

daraufhin überprüft, ob man sie zerlegen sollte. Einige der bereits vorhandenen Methoden, aber auch die durch die Zerlegung neu entstandenen, konnten in andere, passendere Klassen verschoben werden. Anschließend wurden die Import-Statements der jeweiligen Klassen angepasst.

Für die Analyse und Umstrukturierung dieses Moduls waren mehrere Manntage notwendig. Leider war es aus organisatorischen Gründen nicht möglich, am ,,lebenden" Code zu arbeiten und zu beobachten, ob nun bei nachfolgenden Code-Erweiterungen erhebliche Zeiteinsparungen festgestellt werden könnten. Auch ist es fraglich, ob man nach dem Refactoring die im Code noch vorhandenen Fehler (Bugs) schneller beseitigen kann, oder ob der Zeitaufwand letztendlich in etwa der gleiche sein wird. Nur an den Stellen, wo mehrfacher Code beseitigt werden konnte (was im größeren Stil eher selten der Fall war, dafür aber bei den Bedingungen ziemlich oft vorkam), wird man mit Sicherheit die Vorteile von Refactoring zu schätzen wissen. Allerdings ist selbst dort zu hinterfragen, ob es sich aus Zeitgründen lohnt, den gesamten Quelltext mühsam gezielt nach mehrfachem Code zu durchsuchen, um diesen dann zu beseitigen, oder ob es nicht effizienter wäre, die Programmierer dazu anzuregen, mehrfach vorhandenen Code, den sie im Laufe der Weiterentwicklung beziehungsweise Fehlerbeseitigung entdecken, gleich zu eliminieren.

Die Tabellen 5.25 - 5.28 auf den Seiten 83 - 84 zeigen die Metriken der betroffenen Klassen vor und nach dem Refactoring. Man sieht, dass in der Summe lediglich die CBO- und RFC-Metriken kleiner wurden. Die restlichen Metriken sind dagegen alles in allem gestiegen, wobei man bei den einzelnen betroffenen Klassen mehr oder minder starke Schwankungen, was den Vorher- und Nachher-Wert anbetrifft, beobachten kann. Mit anderen Worten konnte die Kapselung der Klassen durch das Refactoring verbessert werden, worauf niedrigere CBO- und RFC-Werte hindeuten, was seinerseits einen niedrigeren Testaufwand bei anstehenden Änderungen und Erweiterungen vermuten lässt. Die Einarbeitungszeit dürfte zwar bei einzelnen Klassen kürzer werden, sich bei den anderen aber verlängern, was man aus den Schwankungen der LOC- und CC-Metriken schließen kann.

5.2.1 Wie viel Zeit braucht man für Refactoring und was bringt es wirklich?

Generell kann man sagen, dass Tools derzeit Refactoring nur unzureichend unterstützen: Zwar können sie den Entwicklern bei der Anwendung der einfachsten Refactoring-Techniken wie beispielsweise *Variable/Methode/Klasse umbennenen* oder *statische Methode verschieben* behilflich sein, aber bereits bei der Extraktion einer Methode muss man entweder das ganze Refactoring oder einen unerheblichen Teil davon manuell durchführen. Bei zu extrahierenden Code-Abschnitten setzen die Tools zu enge Grenzen bezüglich der Beschaffenheit und Lage des betreffenden Codes. Daher ist für die Durchführung von Refactoring relativ viel Zeit aufzuwenden und mit Flüchtigkeitsfehlern zu rechnen; zumindest in REX musste man für über die Hälfte der Umstrukturierungen im Code auf Tool-Unterstützung verzichten - das manuelle Refactoring kostete im optimalen Fall fünf bis zehn Minuten (Testen und Auffinden der Stellen, an denen Refactoring-Techniken angewandt werden mussten, nicht inbegriffen).

Gemessen an dem für das Refactoring benötigten Zeitaufwand erscheint dieser nur dann gerechtfertigt, wenn mehrfach vorkommender Code eliminiert wird. Alles andere kostet ledig-

Klasse/ Interface	**Vorher LOC**	**Nachher LOC**	**Vorher NOIS**	**Nachher NOIS**
AttributDomaeneAuswahl	124	166	5	6
AuswertungAuswahlkriterien	173	202	1	4
AuswertungTypDaten	32	35	2	2
KomponenteMassnahmeValue	61	118	3	3
MetaTypeMassnahme	56	57	1	2
MetaTypeMassnahmeImpl	193	194	2	3
Massnahmen	78	94	4	8
MassnahmeAdapter	508	490	23	21
MassnahmenController	1327	964	40	57
MassnahmenLiveTable	474	586	24	35
MassnahmeNode	84	193	2	6
Reiter	450	465	16	25
Util	-	29	-	0
VoMassnahme	91	138	4	10
Summe	3.651	3.731	127	182

Tabelle 5.25: Anwendung diverser Refactoring-Techniken: Metrikenänderungen[1]

Klasse/ Interface	**Vorher NOM**	**Nachher NOM**	**Vorher LCOM1**	**Nachher LCOM1**
AttributDomaeneAuswahl	25	28	366	435
AuswertungAuswahlkriterien	45	46	936	981
AuswertungTypDaten	5	5	13	20
KomponenteMassnahmeValue	10	13	37	73
Massnahmen	4	7	34	24
MassnahmeAdapter	23	12	27	15
MassnahmenController	62	62	1.549	1.209
MassnahmenLiveTable	22	25	191	263
MassnahmeNode	8	17	32	121
Reiter	38	40	861	948
Util	-	2	-	-
VoMassnahme	20	24	162	222
Summe	262	281	4.208	4.311

Tabelle 5.26: Anwendung diverser Refactoring-Techniken: Metrikenänderungen[2]

lich Zeit; mehr noch, es ist zweifelhaft, ob eine aus Gründen der objektorientierten Programmierung in eine andere Klasse verschobene Methode an irgendeiner anderen Stelle im Code eingesetzt werden wird, da viele der Methoden äußerst spezialisiert und auf die Bedürfnisse der einzelnen Dialoge oder Rechenvorschriften abgestimmt sind.

Wenn man das Buch von Martin Fowler (siehe [Fowler00]) zum ersten Mal liest, ist man anfangs von folgenden Gedanken beherrscht, die man allerdings im Laufe der Anwendung der

Klasse/ Interface	**Vorher RFC**	**Nachher RFC**	**Vorher CC**	**Nachher CC**
AttributDomaeneAuswahl	56	67	33	39
AuswertungAuswahlkriterien	52	64	48	49
AuswertungTypDaten	39	40	7	8
KomponenteMassnahmeValue	131	140	16	28
Massnahmen	427	417	11	11
MassnahmeAdapter	189	185	59	56
MassnahmenController	787	698	190	149
MassnahmenLiveTable	850	841	54	62
MassnahmeNode	116	143	12	25
Reiter	466	450	64	66
Util	-	2	-	6
VoMassnahme	47	64	21	29
Summe	3.160	3.111	515	528

Tabelle 5.27: Anwendung diverser Refactoring-Techniken: Metrikenänderungen[3]

Klasse/ Interface	**Vorher CBO**	**Nachher CBO**	**Vorher LCOM2**	**Nachher LCOM2**
AttributDomaeneAuswahl	10	11	94	94
AuswertungAuswahlkriterien	5	10	97	97
AuswertungTypDaten	2	2	76	79
KomponenteMassnahmeValue	3	4	37	37
MassnahmeAdapter	31	29	84	83
MassnahmenController	85	59	92	91
MassnahmenLiveTable	31	37	89	90
MassnahmeNode	8	11	85	85
Summe	263	251	654	656

Tabelle 5.28: Anwendung diverser Refactoring-Techniken: Metrikenänderungen[4]

dort aufgezählten Refactoring-Techniken teilweise revidieren muss:

- ,,Das kenne ich, das wende ich sowieso schon die ganze Zeit an". Mit anderen Worten wurden unter dem Begriff ,,Refactoring" zum Teil altbekannte Vorschläge zum effizienteren Programmieren zusammengefasst, die erfahrene Entwickler anwenden.
- ,,Interessante Vorschläge, man muss sie auf jeden Fall umsetzen". In diesem Fall tritt spätestens dann die Ernüchterung ein, wenn man versucht, den bestehenden Code mit Hilfe von Refactoring umzustellen, denn meist entspricht der Code doch nicht dem Idealfall, und es sind viele kleine Schritte notwendig, bis man eine gewünschte Refactoring-Technik anwenden kann. Das Ergebnis sieht zwar ,,schöner" aus, aber es ist fraglich, ob eine zum Teil über fünfzehn Minuten andauernde Umstellung dafür sorgen wird, dass der Code später schneller erweitert beziehungsweise gewartet werden kann als vorher.

5.2.2 Empfehlungen zum weiteren Vorgehen in REX

Unter Berücksichtigung der Tatsachen, dass:

- REX auch weiterhin geändert und erweitert werden wird;
- es in REX abwechselnd Phasen gibt, in denen
  - die Entwickler kaum Zeit haben, um die ihnen zugeteilten Aufgaben zu erfüllen,
  - es aufgrund offener Fragen im Fachkonzept und durch Warten auf Fertigstellung abhängiger Module zu Verzögerungen kommt, d.h. es ,,ruhiger" zugeht;
- man das Projekt wirtschaftlich organisieren muss, das heißt, dass man keine Ressourcen, wie zum Beispiel Zeit, verschwenden darf;
- nicht von allen Team-Mitglieder langjährige Erfahrung vorausgesetzt werden kann und deshalb Fehler auch in Zukunft möglich sind;
- Refactoring nicht zum Selbstzweck werden darf;
- man einem Entwickler nicht zumuten kann, Tage lang Refactoring durchzuführen, weil es dann als eine Routine-Arbeit auffasst und mit weniger Sorgfalt bewerkstelligen würde;

erscheinen folgende, nach ihrer Wichtigkeit sortierte Vorgehensweisen zur besseren Wartbarkeit des Projektes REX sinnvoll:

1. Dem Thema ,,Wartbarkeit des Quellcodes" sollte mehr Bedeutung beigemessen werden, indem es in die Projektplanung mit einfließt.
2. Alle Team-Mitglieder sollten in einem kurzen Workshop auf Refactoring aufmerksam gemacht werden, wobei die essentiellen Refactoring-Techniken vorgestellt werden sollen. Dabei sollen aus dem Team zwei erfahrene Entwickler ausgesucht werden, die als Ansprechpartner in Fragen Refactoring zur Verfügung stehen. Bei Weiterentwicklung und Bug-Fixing soll Refactoring zur Vermeidung von mehrfach vorkommendem Code benutzt werden. Falls es aus Zeitgründen nicht möglich sein sollte, an der einen oder anderen Stelle Refactoring sofort durchzuführen, sollte die betroffene Code-Stelle samt einer kurzen Beschreibung in das bei REX bereits eingesetzte *bugzilla*[1] aufgenommen und dort entsprechend verwaltet werden.
3. In regelmäßigen Zeitintervallen, zum Beispiel nach dem Integrationstest vor der Auslieferung einer Leistungsstufe, empfiehlt es sich, Code-Reviews seitens der Vertreter der Qualitäts- oder Technologie-Management-Abteilung durchzuführen, so dass ihre Empfehlungen zur Verbesserung der Code-Struktur in der Bug-Fixing-Phase eingearbeitet werden können. Dabei ist es wichtig, dass die gefundenen Probleme mit den betroffenen Entwicklern persönlich besprochen werden, um bessere Akzeptanz der Code-Reviews zu erreichen und diese nicht als Straf-Maßnahme zu diskreditieren. Generelle Empfehlungen zur Verbesserung der Code-Struktur sollen an die Projektleitung weitergegeben werden.

[1] Bugzilla ist ein *Bug-Tracking-System*, welches eine übersichtliche Darstellung der in einem Programm entdeckten Bugs in einem Web-Browser ermöglicht, wobei alle Kommentare zu dem Bug einsehbar sind; siehe auch [www_bugzilla]

4. Es sollte über die Einführung eines oder mehrerer Informationssysteme nachgedacht werden, in denen nicht nur auf neue Klassen zur Bewältigung spezieller Aufgaben hingewiesen werden kann, sondern auch Hinweise darauf, welche Lösungsansätze für immer wieder vorkommende Probleme wo zu finden sind, eingetragen werden können.

5.2.3 Übertragbarkeit der Ergebnisse auf andere Projekte

Die während des Refactorings von REX gewonnenen Erkenntnisse dürften mit großer Wahrscheinlichkeit auch auf andere, in einer objektorientierten Programmiersprache entwickelte Langzeitprojekte zutreffen, da unter dem oft herrschenden Zeitdruck insbesondere vor einer anstehenden Auslieferung folgende Programmier- beziehungsweise Design-Fehler gehäuft vorkommen dürften:

- Es wird prozedural statt objektorientiert programmiert, wodurch die einzelnen Methoden
  - lang und unübersichtlich werden,
  - in der „falschen" Klasse untergebracht werden, das heißt, dass sie zwar in der Klasse benötigt werden, aber in einer anderen Klasse besser platziert wären.
- Code-Abschnitte werden an eine andere Stelle kopiert.
- Programmierung ohne detaillierte Ausarbeitung des Objekt-Modells für alle zu erstellenden Klassen.

Auch darf der wirtschaftliche Faktor nicht aus den Augen gelassen werden, denn in allen Projekten stehen nur begrenzte Mittel wie Personal, Zeit und Budget zur Verfügung, so dass man es sich nicht leisten kann, beliebig lang an Code-Verschönerungen zu arbeiten.

5.3 Einsatz von Refactoring in einem Langzeitprojekt

In den Kapiteln 3 und 4 wurden Refactoring und entsprechende Werkzeuge vorgestellt, außerdem fand in Kapitel 5 eine Analyse einiger wichtiger und der gebräuchlichsten Refactoring-Techniken am Beispiel ausgewählter Klassen des Projektes REX statt.

Nun stellt sich die Frage, wie Refactoring in einem Projekt mit langer Laufzeit eingeführt und erfolgreich angewandt werden kann.

Bevor die Beantwortung dieser Frage möglich wird, muss man das Ziel klar definieren, das man mit Hilfe von Refactoring erreichen will. Aus diversen Untersuchungen geht hervor (vergleiche unter anderem [Sommer00]), dass die meiste Zeit bei Projekten in die Wartung von Software investiert wird, also in das Beseitigen von Bugs sowie die Anpassung an Umgebungsveränderungen und neue beziehungsweise geänderte Wünsche der Benutzer, die im Laufe des Software-Lebenszyklus mit hoher Wahrscheinlichkeit aufkommen. Das größte Hindernis ist dabei mehrfach vorhandener Code, durch den bedingt wird, dass man eine Änderung gleich an mehreren Stellen nachzuziehen hat, was oft daran scheitert, dass manche dieser Code-Stellen bei der Durchführung einer Modifikation vergessen beziehungsweise übersehen werden. Aus

diesem Grund sollte das oberste Ziel von Refactoring darin liegen, mehrfach vorhandenen Code zu eliminieren. Zum Erreichen dieses Ziels sind folgende Refactoring-Techniken besonders wichtig:

- Methode extrahieren,
- Superklasse extrahieren,
- Methode in die Superklasse verschieben,
- Klasse extrahieren,
- Methode in eine andere Klasse verschieben,
- Bedingung zerlegen.

Für die Durchführung von Refactoring existieren generell zwei Wege:

1. Refactoring nebenbei zu betreiben, das heißt, jedes Mal, wenn einem Entwickler zum Beispiel mehrfach vorhandener Code auffällt, könnte er diesen durch entsprechende Maßnahmen sofort beseitigen. Der Vorteil dieser Vorgehensweise liegt auf der Hand: der gefundene Programmierfehler (in diesem Fall duplizierter Code) kann auf der Stelle aus der Welt geschafft werden. Allerdings könnte es unter Umständen unmöglich sein, erforderliche Code-Änderungen kurzfristig durchzuführen, weil beispielsweise ein Auslieferungstermin bevorsteht, und man es sich nicht mehr leisten kann, kurz davor Umstellungen durchzuführen, die ausführlich getestet werden müssten. Es ist dennoch denkbar, dass man für die Refactorings, deren Durchführung notwendig ist, aber zum aktuellen Zeitpunkt wegen Terminschwierigkeiten unmöglich erscheint, eine Liste einführt, die nach dem Prinzip eines Bug-Tracking-Systems funktionieren sollte. Entwickler könnten dann Refactoring-Vorschläge (kurze Beschreibung des Problems sowie Aufzählung betroffener Klassen und/oder Methoden) in das System eintragen, um sie, sobald sie dafür mehr Zeit haben, zu verwirklichen. Weiterhin wäre es wünschenswert, dass entweder die Entwickler selbst oder die Projektleitung diese Vorschläge priorisieren und Deadlines setzen würden, bis wann jeder der Vorschläge spätestens umgesetzt werden müsste.

2. Es ist vorstellbar, besonderes wenn sich Refactoring nicht von Anfang an als Technik zur Verbesserung der Code-Struktur in einem Projekt etabliert hat, dass man Refactoring einem oder mehreren Team-Mitgliedern beziehungsweise eventuell einem speziell dafür eingesetzten Personal überlässt, das dann den gesamten Code in regelmäßigen Zeitabschnitten Zeile für Zeile analysieren muss, um die Stellen zu finden, wo die Anwendung von Refactoring angebracht ist. Vorteilhaft ist hierbei, dass man auf diese Weise nicht nur zufällig entdeckte Stellen umstrukturieren, sondern gezielt nach Problemen suchen kann. Der Nachteil liegt vermutlich im höheren Zeitaufwand wegen der Einarbeitungszeit, da sich die Entwickler größtenteils mit fremdem Code auseinander setzen müssen, was jedoch dadurch vereinfacht werden könnte, dass es Rücksprachen zwischen den eigentlichen Entwicklern und den Team-Mitgliedern gibt, die den Code einem Refactoring unterziehen. Dies hätte zum einen den Vorteil, dass die Entwickler auf die Schwachstellen und Unzulänglichkeiten in ihrem Code hingewiesen werden würden und diese in

Zukunft eher vermeiden könnten. Zum anderen würden Entwickler, die sich mit Refactoring beschäftigen, umfangreichere Informationen zur Code-Struktur und Hinweise auf eventuell bestehende Probleme bekommen, was ihre Arbeit erheblich erleichtern würde. Trotzdem ist beispielsweise das Aufspüren von mehrfachem Code ohne entsprechende Hilfe wie zum Beispiel CloneDR (siehe [www_semDes]) kaum möglich.

Obwohl die im Punkt 1 vorgestellte Vorgehensweise nicht gewährleistet, dass alle Code-Stellen, die einer Umstrukturierung bedürfen, wirklich gefunden werden, scheint sie dennoch der effektivste Weg zu sein, Refactoring durchzuführen, vor allem, weil sie bedarfsorientierter ist (das heißt, dass nur die Stellen, die nochmals gewartet beziehungsweise erweitert werden, und nicht der gesamte Code (was sehr zeitraubend wäre), bei Bedarf einem Refactoring unterzogen werden). Damit diese Vorgehensweise den maximalen positiven Effekt mit sich bringt, müssen alle Entwickler im Team angehalten werden, Refactoring durchzuführen und/oder bekannte Schwachstellen im Code entsprechend zu notieren, um diese später umzustellen. Neben den bereits aufgezählten Vorteilen bietet die zur Software-Entwicklung parallele Anwendung von Refactoring-Techniken folgende positive Effekte:

- Die Entwickler haben größere Verantwortung und sind eher geneigt, alles von Anfang an richtig zu machen, bevor sie der Denkweise verfallen: ,,Es wird schon jemand richten". In der Praxis kommt es vor, dass die Bereinigung der Code-Struktur während einer ,,heißen" Entwicklungsphase auf einen späteren Zeitpunkt verschoben wird, so dass bewusst ,,unsauber", dafür aber schnell programmiert wird, in der Hoffnung, dass man später, wenn man Zeit hat, alles verbessern wird.

- Die Durchführung von Refactoring auf einem Code-Abschnitt wird nicht als persönlicher Angriff betrachtet und deshalb angefeindet, sondern findet eher Verständnis auf Seiten der Entwickler.

Unabhängig davon, wie Refactoring in einem Projekt letztendlich eingesetzt wird, ist es empfehlenswert, Entwickler auf seine Vorteile, aber auch die in ihm verborgenen Gefahren hinzuweisen und ihnen zu erklären, dass

- sie Refactoring-Techniken wenn immer sinnvoll nutzen sollten. Bei Bedarf sollte eventuell ein kurzer Workshop von erfahrenen Programmierern veranstaltet werden, in dem Refactoring-Verfahren und -Werkzeuge vorgestellt werden.

- Refactoring seitens speziell dafür geschulten Personals außerhalb des Teams in regelmäßigen Zeitabschnitten durchgeführt wird.

6 Schlussfolgerung und Ausblick

Im Rahmen dieser Studie wurde untersucht, wie man mit Hilfe von Refactoring die Code-Struktur von Langzeitprojekten verbessern kann. In den Abschnitten 6.1 - 6.3 folgt nun eine abschließende Betrachtung der dabei gewonnenen Erkenntnisse.

6.1 Refactoring-Tools

In Kapitel 4.2 wurden Werkzeuge evaluiert, die Entwickler beim Refactoring unterstützen können. Manche der im Kapitel 5 beschriebenen Refactorings wurden mit Hilfe dieser Tools bewerkstelligt. Allerdings stellte sich nach und nach heraus, dass die seitens der Tools zur Verfügung gestellte Unterstützung mangelhaft ist. In Anbetracht der Tatsache, dass zumindest bei einem der untersuchten Refactoring-Werkzeuge (IntelliJ IDEA, siehe Abschnitt 4.2.2) die Anzahl der unterstützen Refactoring-Techniken innerhalb der letzten drei Monate stark vergrößert wurde, ist anzunehmen, dass in naher Zukunft zumindest ein Teil der heute noch bestehenden Probleme gelöst werden wird.

Folgende Aufzählung bietet einen Überblick über die derzeit noch fehlenden Funktionalitäten:

1. Die Verschiebung einer nicht statischen Methode in eine andere Klasse, wobei die zu verschiebende Methode auf keine Elemente der aktuellen Klasse, das heißt weder auf Attribute noch auf Methoden, zugreift, soll möglich sein.

2. Eine Möglichkeit, anzugeben, dass bei der Verschiebung einer Methode in eine andere Klasse ein Übergabeparameter desselben Typs nicht mehr benötigt wird, und stattdessen alle Operationen auf diesem Parameter durch `this` ersetzt werden sollen.

3. Geeignete Mechanismen bei einer Methoden-Extraktion (zum Beispiel durch Rückgabe eines Booleschen Wertes und seine entsprechende Behandlung), so dass die in der ursprünglichen Methode vorkommenden Abbrüche (wie *return*-Statements) entsprechend berücksichtigt werden können (momentan gibt es in solchen Fällen lediglich einen Hinweis darauf, dass man in diesem Code-Abschnitt keine Methoden-Extraktion durchführen kann).

4. Unterstützung für das Bedingung-Zerlegen-Refactoring (vergleiche Abschnittt 5.1.3), so dass man beispielsweise mit Hilfe der Methoden-Extraktion Teile der Abfragen durch entsprechende Methodenaufrufe ersetzen kann.

5. Ersetzen gleicher Abfragen beziehungsweise mehrfach vorhandenen Codes innerhalb einer Klasse oder Methode durch entsprechende vom Benutzer angegebene Methodenaufrufe, beispielsweise nach einer Methoden-Extraktion.

6. Automatische Entfernung der nach einer Methoden-Verschiebung nicht mehr benötigten Import-Statements aus der alten Klasse.

Bei all den Vorteilen, die Refactoring-Werkzeuge mit sich bringen, darf man nicht aus den Augen verlieren, dass jeder einzelne Entwickler immer aufs neue entscheiden muss, ob und welche Refactoring-Technik an dieser oder jener Stelle angebracht ist. Es ist illusorisch zu erwarten, dass es je ein Tool geben wird, das automatisch alle notwendigen Refactorings durchführt, indem es selbst entscheidet, an welcher Stelle welche Umstrukturierung angemessen ist, und diese dann ohne Eingreifen des Benutzers anwendet.

6.2 Einsatz von Refactoring in Langzeitprojekten

Eine Untersuchung der Vor- und Nachteile des Refactorings am Beispiel des Projektes REX wurde in Kapitel 5 beschrieben. Die wichtigsten Erkenntnisse werden hier nochmal knapp vorgestellt:

subjektive Vorteile:

- Durch kürzere Methoden beziehungsweise weniger Methoden pro Klasse kann die Übersichtlichkeit des Quellcodes erhöht werden (hierfür sind insbesondere *Methode/Klasse extrahieren* sowie *Methode/Klasse verschieben* hilfreich).
- Eine Minimierung der Verwechslungsgefahr kann beispielsweise durch die Ersetzung einer langen Parameterliste (mit Parametern gleichen Typs) durch ein Parameterobjekt erreicht werden.

objektive Vorteile:

- Vermeidung von mehrfach vorhandenem Code, zum Beispiel mit Hilfe der Methoden-Extraktion, erleichtert spätere Wartung und führt zur besseren Änderbarkeit des Quelltextes, weil eine Änderung beziehungsweise Erweiterung an einer und nicht an mehreren Stellen durchgeführt werden muss;
- Die durch Anwendung einer Kombination verschiedener Refactoring-Techniken hervorgerufene Senkung der CBO-Metrik deutet auf bessere Kapselung und damit Wiederverwendbarkeit hin, ein geringerer RFC-Wert signalisiert geringere Komplexität und somit leichtere Testbarkeit und Erweiterbarkeit.

Nachteile:

- Die für Refactoring zu veranschlagende Zeit variiert von wenigen Minuten bis zu mehreren Stunden in Abhängigkeit davon, welche Refactoring-Technik angewandt werden soll und ob ihre Durchführung am ausgewählten Code-Abschnitt durch spezielle Tools unterstützt wird, was in etwa der Hälfte der Fälle bei REX nicht möglich war.
- Bedingt durch die zum Teil nicht vorhande Tool-Unterstützung sowie durch Unachtsamkeit und Nachlässigkeit bei der Durchführung von Refactoring kann die Entstehung von Fehlern begünstigt werden. Daher sind ausführliche Tests nach jedem Refactoring unabdingbar, diese kosten aber wiederum Zeit.

- Einige Refactoring-Techniken haben negative Auswirkungen auf die Performanz, so führt beispielsweise Extraktion einer Methode zu einer Verzögerung bei der Code-Ausführung.

Zusammenfassend kann man sagen, dass sich Refactoring nur dann lohnt, wenn man dadurch Redundanz vermeiden kann; alles andere hat derzeit mehr Kosten als Nutzen, da

- allein die Tatsache, dass sich eine Methode in einer anderen ,,richtigeren" Klasse befindet, nicht zwangsläufig dazu führt, dass sich Fehler schneller finden lassen oder der Code leichter wartbar ist;
- manchmal zu viele kleine Methoden/Klassen eher störend sind, weil man zum Beispiel bei der Fehlersuche permanent ,,hüpfen" muss, um den entsprechenden Code-Abschnitt zu begutachten;
- die meisten Methoden bereits so spezialisiert sind, dass eine erleichterte Wiederverwendung durch Verschiebung in eine andere Klasse kaum denkbar ist.

Aufgrund des nicht zu vernachlässigenden Testaufwands empfiehlt es sich, Refactoring nur parallel zur laufenden Entwicklung durchzuführen, und zwar erst dann, wenn man sich sowieso wegen anstehender Änderungen beziehungsweise Erweiterungen mit einer bestimmten Code-Stelle beschäftigt und dabei entdeckt, dass mehrfach vorhandener Code eliminiert werden kann.

Abschließend kann man festhalten, dass bloße Anwendung von Refactoring-Techniken weder eine Garantie für eine durchgehende gute Code-Qualität ist, noch kann sie andere Qualitätssicherungsmaßnahmen wie beispielsweise Code-Reviews ersetzen.

6.3 Ausblick

Auf dem Gebiet des Refactorings stehen noch weitere Untersuchungen an, mit deren Hilfe man die Vor- gegen die Nachteile dieser Technik abwägen kann, um so eventuell neue Anwendungsszenarien zu finden beziehungsweise die hier empfohlenen Vorgehensweisen zu verbessern. Dabei sollen folgende Faktoren berücksichtigt werden:

Wirtschaftliche Faktoren:

- Mit Hilfe von exakten Messungen sollte ermittelt werden, ob sich durch den Einsatz von Refactoring-Techniken Entwicklungszeit einsparen lässt. Man könnte folgenderweise vorgehen: zwei gleich erfahrene Entwickler sollen eine Erweiterung durchführen und/oder einen vorhandenen Bug beseitigen, wobei einer der Entwickler mit dem ursprünglichen Code arbeitet, der andere Entwickler dagegen mit demselben, aber zuvor einem Refactoring unterzogenen Code. Die jeweils benötigte Zeit soll gemessen und verglichen werden.
- Im Rahmen von Kundenbefragungen sollte ermittelt werden, was ihnen wichtiger ist:
  - schnell eine lauffähige Software zu bekommen, deren interne Code-Struktur Mängel aufzeigt, was dazu führen könnte, dass sich die Fehlerbehebungs- und Erweiterungszyklen als langwierig erweisen, oder

  - längere Zeit auf eine ,,perfekte" Software zu warten und dafür entsprechend mehr zahlen zu müssen.

  Diese Befragungen sollten unter Aufteilung nach den jeweiligen Gruppen:

  - private Anwender,
  - mittelständische Unternehmen,
  - große Auftraggeber

  durchgeführt werden.

Soziale Faktoren:
Es wäre interessant Refactoring an einem laufenden Projekt zu betreiben und dabei die Reaktionen der betroffenen Entwickler zu analysieren:

- Hat sich dadurch das Arbeitsklima verschlechtert oder verbessert?
- Sind Spannungen zwischen den einzelnen Entwicklern zu beobachten?
- Hat die Verantwortung einiger Entwickler bezüglich des Designs nachgelassen, weil sie der Ansicht sind, dass sowieso ein anderer Programmierer die von Ihnen gemachten Fehler in der Code-Struktur verbessern wird oder hat sich die Qualität durch den Einsatz der Technik verbessert?

A Performanzmessungen

A.1 Parameterobjekt einführen

TestParameterObject.java

```
package performance;
/**
 * This class shows the delay caused by use of a parameter object.
 * With its help changes in performance are demonstrated by
 * comparing the time necessary to execute both a method with
 * many parameters and a method which has got this class as
 * only parameter
 */
public class TestParameterObject
{
    public static void main(String[] args)
    {
        TestParameterObject thisClass = new TestParameterObject();
        thisClass.testTimeForParameterObject();
    }
    private void testTimeForParameterObject()
    {
        int numberOfRuns = 10000000;
        long withParameterObject = 0;
        long withLongParameterList = 0;
        //parameters for the test
        String name = "Smith";
        int age = 24;
        int semester = 9;
        boolean married = false;
        withParameterObject =
            timeWithParameterObject(
                numberOfRuns,
                name,
                age,
                semester,
                married);
        withLongParameterList =
            timeWithLongParameterList(
```

```
                numberOfRuns,
                name,
                age,
                semester,
                married);
        System.out.println(" totally " + numberOfRuns + " cycles:");
        System.out.println(
            "time using a parameter object ="
                + withParameterObject
                + " ms");
        System.out.println(
            "time using a long parameter list ="
                + withLongParameterList
                + " ms");
    }
    private long timeWithLongParameterList(
        int numberOfRuns,
        String name,
        int age,
        int semester,
        boolean married)
    {
        int entry;
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < numberOfRuns; i++)
        {
            entry = generateEntry(name, age, semester, married);
        }
        long endTime = System.currentTimeMillis();
        long duration = endTime - startTime;
        return duration;
    }
    private long timeWithParameterObject(
        int numberOfRuns,
        String name,
        int age,
        int semester,
        boolean married)
    {
        ParameterObject paramObj = new ParameterObject();
        paramObj.setName(name);
        paramObj.setAge(age);
        paramObj.setSemester(semester);
        paramObj.setMarried(married);
        int entry;
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < numberOfRuns; i++)
```

```
        {
            entry = generateEntry(paramObj);
        }
        long endTime = System.currentTimeMillis();
        long duration = endTime - startTime;
        return duration;
    }
    private int generateEntry(
        String name,
        int age,
        int semester,
        boolean married)
    {
        int result = name.hashCode();
        result += age;
        result += semester;
        if (married)
            result += 1;
        return result;
    }
    private int generateEntry(ParameterObject paramObj)
    {
        int result = paramObj.getName().hashCode();
        result += paramObj.getAge();
        result += paramObj.getSemester();
        if (paramObj.isMarried())
            result += 1;
        return result;
    }
}
```

ParameterObject.java

```
package performance;
/**
 * Class for holding parameter data
 */
public class ParameterObject
{
    private String name;
    private int age;
    private int semester;
    private boolean married;
    public String getName()
    {
        return name;
    }
```

```
    public void setName(String name)
    {
        this.name = name;
    }
    public int getAge()
    {
        return age;
    }
    public void setSemester(int semester)
    {
        this.semester = semester;
    }
    public int getSemester()
    {
        return semester;
    }
    public void setAge(int age)
    {
        this.age = age;
    }
    public boolean isMarried()
    {
        return married;
    }
    public void setMarried(boolean married)
    {
        this.married = married;
    }
}
```

A.2 Methode extrahieren/verschieben

MoveMethod.java

```
package performance;
/**
 * This class shows whether a delay is caused
 * by extracting/moving a method
 */
public class MoveMethod extends SuperclassOfMoveMethod
{
    public static void main(String[] args)
    {
        MoveMethod thisClass = new MoveMethod();
        thisClass.testTimeMoveMethod();
    }
    private void testTimeMoveMethod()
```

```
{
    int numberOfRuns = 1000000;
    long[] totalTimeWithoutMovingMethod =
        timeWithoutMovingMethod(numberOfRuns);
    long[] totalTimeForExtractedMethod =
        timeForExtractedMethod(numberOfRuns);
    long[] totalTimeForMethodInSuperclass =
        timeForMethodInSuperclass(numberOfRuns);
    long[] totalTimeForMethodInExtractedClass =
        timeForMethodInExtractedClass(numberOfRuns);
    System.out.println("totally " + numberOfRuns + " cycles:");
    System.out.println("total time for doing nothing");
    System.out.println(
        "without moving method = "
            + totalTimeWithoutMovingMethod[0]
            + " ms");
    System.out.println(
        "for extracted method in the same class = "
            + totalTimeForExtractedMethod[0]
            + " ms");
    System.out.println(
        "with moving method in superclass = "
            + totalTimeForMethodInSuperclass[0]
            + " ms");
    System.out.println(
        "with moving method in extracted class = "
            + totalTimeForMethodInExtractedClass[0]
            + " ms");
    System.out.println(
        "total time for doing arithmetical operations");
    System.out.println(
        "without moving method = "
            + totalTimeWithoutMovingMethod[1]
            + " ms");
    System.out.println(
        "for extracted method in the same class = "
            + totalTimeForExtractedMethod[1]
            + " ms");
    System.out.println(
        "with moving method in superclass = "
            + totalTimeForMethodInSuperclass[1]
            + " ms");
    System.out.println(
        "with moving method in extracted class = "
            + totalTimeForMethodInExtractedClass[1]
            + " ms");
}
```

```
private long[] timeWithoutMovingMethod(int times)
{
    long[] duration = new long[2];
    long startTime;
    long endTime;
    int number;
    String string;
    //computing time for doing nothing
    startTime = System.currentTimeMillis();
    for (int i = 0; i < times; i++)
    {
        //do nothing
    }
    endTime = System.currentTimeMillis();
    duration[0] = endTime - startTime;
    //computing time for arithmetical operations
    number = initNumber();
    startTime = System.currentTimeMillis();
    for (int i = 0; i < times; i++)
    {
        number = number * i;
        number = number % 2;
        number = number * 10;
    }
    endTime = System.currentTimeMillis();
    duration[1] = endTime - startTime;
    return duration;
}
private long[] timeForExtractedMethod(int times)
{
    long[] duration = new long[2];
    long startTime;
    long endTime;
    int number;
    String string;
    //computing time for doing nothing
    startTime = System.currentTimeMillis();
    for (int i = 0; i < times; i++)
    {
        this.doNothing();
    }
    endTime = System.currentTimeMillis();
    duration[0] = endTime - startTime;
    //computing time for arithmetical operations
    number = initNumber();
    startTime = System.currentTimeMillis();
    for (int i = 0; i < times; i++)
```

```
        {
            this.doArithmeticalOperation(i, number);
        }
        endTime = System.currentTimeMillis();
        duration[1] = endTime - startTime;
        return duration;
    }
    private long[] timeForMethodInSuperclass(int times)
    {
        long[] duration = new long[2];
        long startTime;
        long endTime;
        int number;
        String string;
        //computing time for doing nothing
        startTime = System.currentTimeMillis();
        for (int i = 0; i < times; i++)
        {
            super.doNothing();
        }
        endTime = System.currentTimeMillis();
        duration[0] = endTime - startTime;
        //computing time for arithmetical operations
        number = initNumber();
        startTime = System.currentTimeMillis();
        for (int i = 0; i < times; i++)
        {
            super.doArithmeticalOperation(i, number);
        }
        endTime = System.currentTimeMillis();
        duration[1] = endTime - startTime;
        return duration;
    }
    private long[] timeForMethodInExtractedClass(int times)
    {
        long[] duration = new long[2];
        long startTime;
        long endTime;
        int number;
        String string;
        ExtractedClass obj = new ExtractedClass();
        //computing time for doing nothing
        startTime = System.currentTimeMillis();
        for (int i = 0; i < times; i++)
        {
            obj.doNothing();
        }
```

```
        endTime = System.currentTimeMillis();
        duration[0] = endTime - startTime;
        //computing time for arithmetical operations
        number = initNumber();
        startTime = System.currentTimeMillis();
        for (int i = 0; i < times; i++)
        {
            obj.doArithmeticalOperation(i, number);
        }
        endTime = System.currentTimeMillis();
        duration[1] = endTime - startTime;
        return duration;
    }
    protected void doArithmeticalOperation(int count, int number)
    {
        number = Math.abs(count - number);
        number = number % 2;
        number = number * 10;
    }
    protected void doNothing()
    {
    }
    private int initNumber()
    {
        return 25;
    }
}
```

SuperclassOfMoveMethod.java

```
package performance;
/**
 * Class which consists of two methods.
 * With its help changes in performance can be demonstrated
 * by comparing the necessary time for calling this methods
 * from this class to calling the same methods from the
 * initial class.
 */
public class SuperclassOfMoveMethod
{
        protected void doNothing()
    {
    }
    protected void doArithmeticalOperation(int count, int number)
    {
        number = Math.abs(count - number);
        number = number%2;
```

```
        number = number*10;
    }
}
```

ExtractedClass.java

```
package performance;
/**
 * Class which consists of two methods.
 * With its help changes in performance by extracting a
 * class can be demonstrated by comparing necessary time
 * for calling this method from this class to calling
 * the same method from the initial class.
 */
public class ExtractedClass
{
    protected void doNothing()
    {
    }
    protected void doArithmeticalOperation(int count, int number)
    {
        number = Math.abs(count - number);
        number = number % 2;
        number = number * 10;
    }
}
```

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Refactoring

[Astels02] Dave Astels: *Refactoring With UML*, In Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, Alghero, Sardinia, Italy, 2002

[CinNix00] Ó. M. Cinnéide & P. Nixon: *Composite Refactorings for Java Programs*, Proceedings of the Workshop on Formal Techniques for Java Programs, European Conference on Object-Oriented Programming, 2000

[Fowler00] Martin Fowler: *Refactoring. Wie Sie das Design vorhandener Software verbessern*, Addison-Wesley 2000

[Hunger00] Michael Hunger: *Refactoring. Benefits and Disadvantages of an Amazing Technique*, Belegarbeit, PDAI, TU Dresden, 2000 (http://emw.inf.tu-dresden.de/de/pdai/Forschung/refactoring/refactoring_html/index.html)

[Korman98] W. F. Korman: *Elbereth: Tool Support for Refactoring Java Programs*, M.S. Thesis, Technical Report CS98-590, Department of Computer Science and Engineering, University of California, San Diego, June 1998

[MaCrDe01] Dan Malks, John Crupi, Deepak Alur: *Core J2EE Patterns: Best Practices and Design Strategies*, Prentice Hall PTR, 2001

[Opdyke92] Willialm F. Opdyke: *Refactoring Object-Oriented Frameworks*, PhD thesis, University of Illinois, Urbana-Champaign, 1992

[Pipka02] Jens Uwe Pipka: *Refactoring in a „Test First"-World*, In Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering, Alghero, Sardinia, Italy, 2002

[Robert99] D. B. Roberts: *Practical analysis for refactoring*, Dissertation, University of Illinois at Urbana-Champaign, 1999

[www_codeguide] http://www.omnicore.com/

[www_dpt] http://www.kupin.net/uni/dpt/

[www_eclipse] http://www.eclipse.org/

[www_idea] http://www.intellij.com/idea/

[www_jbuilder] http://www.borland.com/jbuilder/index.html

[www_jfactor] http://www.instantiations.com/jfactor/

[www_jrefactory] http://jrefactory.sourceforge.net/csrefactory.html

[www_refact] http://www.refactoring.com/

[www_refactorit] http://www.refactorit.com/

[www_refBrowser] http://chip.cs.uiuc.edu/users/brant/Refactory/

[www_semDes] http://www.semanticdesigns.com/

[www_teikade] http://www.pfu.co.jp/teikade

[www_together] http://www.togethersoft.com/products/controlcenter/index.jsp

[www_transmogrify] http://transmogrify.sourceforge.net/

[www_visualage_java] http://www-3.ibm.com/software/ad/vajava/

[www_wiki] http://c2.com/cgi/wiki

Metriken

[EbeDum96] Christof Ebert, Reiner Dumke: *Software-Metriken in der Praxis. Einführung und Anwendung von Software-Metriken in der industriellen Praxis*, Springer Verlag, 1996

[Kahlbr01] Bernd Kahlbrandt: *Skript zur Vorlesung Software Engineering II*, FH Hamburg, 2001

[McCabe76] Tomas J. McCabe: *A Complexity Measure*, IEEE Trans. on Software Engineering, 1976

[RosHya97] L. H. Rosenberg & L. E. Hyatt: *Software Quality Metrics for Object-Oriented Environments*, Crosstalk Journal, 1997 (http://satc.gsfc.nasa.gov/support/CROSS_APR97/oocross.html)

[RosHya98] L. H. Rosenberg & L. E. Hyatt: *Applying and Interpreting Object-Oriented Metrics*, Software Technology Conference, 1998 (http://satc.gsfc.nasa.gov/support/STC_APR98/apply_oo/apply_oo.html)

[RoStGa99] L. H. Rosenberg, R. Stapko, A. Gallo: *Risk-based Object Oriented Testing*, Twenty-Fourth Annual Software Engineering Workshop, 1999 (http://sel.gsfc.nasa.gov/website/sew/1999/topics/rosenberg_SEW99paper.pdf)

[Thalle00] Georg Erwin Thaller: *Software- Metriken. Einsetzen - bewerten - messen*, Verlag Technik, Berlin, 2000

[WatMcC96] Arthur H. Watson & Thomas J. McCabe: *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.*, Gaithersburg, 1996 (http://hissa.ncsl.nist.gov/HHRFdata/Artifacts/ITLdoc/235/mccabe.html)

[www_oom] http://yunus.hun.edu.tr/ sencer/oom.html

[www_oom_empflng] http://satc.gsfc.nasa.gov/metrics/codemetrics/oo/thresholds/

[www_oom_Links] http://www.cetus-links.org/oo_metrics.html

[www_sei_cc] http://www.sei.cmu.edu/str/descriptions/cyclomatic.html

[www_SmallW] http://www.thesmallworlds.com/

[www_UndFJa] http://www.scitools.com/uj.html

Software-Engineering

[Beck00] Kent Beck: *Extreme Programming. Das Manifest*, Addison-Wesley, 2000

[Bulka00] Dov Bulka: *Java Performance and scalability*, Volume 1, Addison-Wesley, 2000

[DeDuNi99] Serge Demeyer, Stéphane Ducasse & Oscar Nierstrasz: *Object-Oriented Software Reengineering*, Vorlesungsunterlagen, Universität Bern, 1999

[GaHeJoVl95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addison-Wesley, 1995

[LiRoWo02] Martin Lippert, Stefan Roock, Henning Wolf: *Software entwickeln mit eXtreme Programming*, dpunkt-Verlag, 2002

[Sommer00] Ian Sommerville: *Software Engineering*, Addison-Wesley 2001

[www_agil_dev] Jim Highsmith: *What Is Agile Software Development?* http://stsc.hill.af.mil/crosstalk/2002/oct/highsmith.asp

[www_bea] http://www.bea.com/

[www_bugzero] http://www.websina.com/bugzero/

[www_bugzilla] http://www.bugzilla.org/

[www_cf] www.knowledgebasesoftware.com/

[www_coWiki] http://www.develnet.org/

[www_f2w] http://f2w.sourceforge.net/

[www_gnats] http://www.cs.utah.edu/dept/old/texinfo/gnats/gnats.html

[www_java_api] http://java.sun.com/j2se/1.3/docs/api/

[www_java_codeConv] http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[www_javadoc] http://java.sun.com/products/jdk/1.2/docs/tooldocs/win32/javadoc.html

[www_java_imports] http://developer.java.sun.com/developer/technicalArticles/Interviews/Packaging/

[www_j2ee] http://java.sun.com/j2ee/

[www_jitterBug] http://samba.anu.edu.au/cgi-bin/jitterbug

[www_junior] http://users.pandora.be/exentryc/

[www_mvc] http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html

[www_ozibug] http://www.ozibug.com/

[www_polar] http://www.polarsoftware.com/

[www_prtracker] http://www.prtracker.com/

[www_satc] http://satc.gsfc.nasa.gov/

[www_topdesk] http://www.topdesklite.com/index.html

[www_toplink] http://www.webgain.com/products/toplink/

Software-Qualität

[DIN_ISO_9126] *Software-Engineering - Qualität von Software-Produkten - Teil 1: Qualitätsmodell*, ISO/IEC 9126-1, Ausgabe: 2001-06

[www_din_iso] http://www.bas.uni-essen.de/Software_Qualitaet.pdf

Stichwortverzeichnis