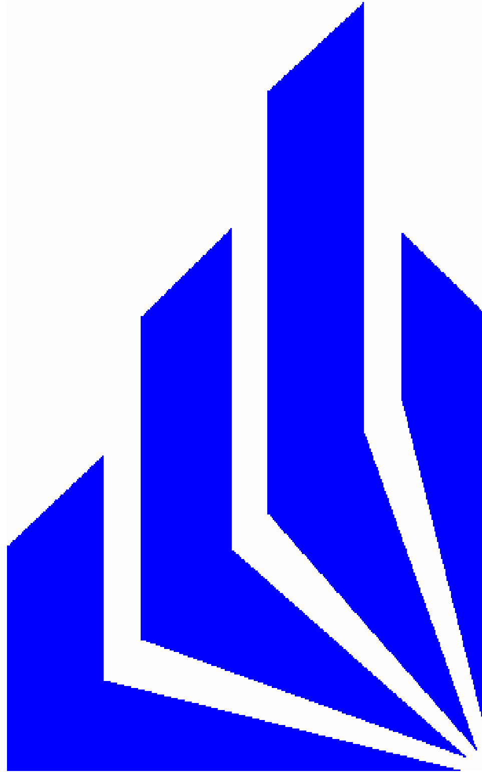

Qualitätssicherung durch Refactoring

Von: Adrian Heidenreich

Seminar Qualitätssicherung

SS2004

Qualitätssicherung durch Refactoring

Von: Adrian Heidenreich

1. **Übersicht**
2. **Einführung**
   - ***2.1.*** *Der Qualitätsbegriff bei Software*
   - ***2.2.*** *Refactoring und Qualitätssicherung*
   - ***2.3.*** *Definition von Refactoring*
   - ***2.4.*** *Anwendungsmotivation zum Refactoring*
   - ***2.5.*** *Einsatzmöglichkeiten von Refactoring*
   - ***2.6.*** *Refactoring und Performance*
3. **Ansatzpunkte für Refactoring**
   - ***3.1.*** *Wann und wo kann man refaktorieren? „Bad Smell Heuristics“*
   - ***3.2.*** *Ausgesuchte Bad Smells mit Beispielen*
     - ***3.2.1.*** *Duplicated Code*
     - ***3.2.2.*** *Long Method*
     - ***3.2.3.*** *Large Class*
     - ***3.2.4.*** *Feature Envy*
     - ***3.2.5.*** *Data Clumps*
4. **Arten des Refactoring**
   - ***4.1.*** *Ein einfaches Beispiel für schlechten Code*
   - ***4.2.*** *Eine Verbesserung durch Refactoring*
   - ***4.3.*** *Was ist passiert?*
   - ***4.4.*** *Refactoring bei UML-Diagrammen*
     - ***4.4.1.*** *Ein schlecht modelliertes Zustandsdiagramm*
     - ***4.4.2.*** *Eine Verbesserung durch Refactoring*
5. **Vorgehensweise beim Refactoring - Tests**
6. **Refactoring Tools**
7. **Refactoring in der Praxis**
   - ***7.1.*** *Grenzen des Refactoring*
   - ***7.2.*** *Zusammenfassung*
8. **Quellen**

2. Einführung

2.1 Der Qualitätsbegriff bei Software

Qualität von Software misst sich Grundsätzlich an unterschiedlichen Faktoren. Das wichtigste Kriterium ist die Frage, ob die Software die Aufgabe erfüllt für die sie entworfen wurde, und wie effizient sie dies tut. Auch die Benutzer- und Bedienfreundlichkeit spielt bei bestimmter Software (z.B. Office-Anwendungen) eine große Rolle in der Bewertung der SW-Qualität. Ein ganz wesentlicher Aspekt, der häufig nicht ausreichend beachtet wird ist allerdings die Erweiterbarkeit (expandability), Anpassbarkeit (flexibility) und Wiederverwendbarkeit (reusability) von Software [2]. So wird Software (insbesondere wenn sie speziell für einen bestimmten Kunden geschrieben wurde), im Laufe ihres „Software Lebenszyklus" immer wieder an neue Umstände (z.B. veränderte Geschäftsprozesse) angepasst oder gar in ihrer Funktionalität erweitert. Diese Softwarequalität spielt auch unter wirtschaftlichen Aspekten eine enorme Rolle, da es bei schlecht strukturierter Software wesentlich schwieriger und damit zeitintensiver und kostspieliger ist, Anpassungen oder Erweiterungen vorzunehmen [1].

2.2. Refactoring und Qualitätssicherung

Damit eine Software die Qualitätskriterien Erweiterbarkeit, Anpassbarkeit und Wiederverwendbarkeit erfüllen kann, muss die Software vor allem eine Qualität besitzen: die Verständlichkeit des Codes. So dürfte selbst ein guter Programmierer beim Durchsehen von 12000 lines of code arge Verständnisprobleme desselbigen haben, wenn er auf ein unübersichtliches Durcheinander von Hierarchien, Parameterübergaben, Klassen und Methoden stößt. Dies ist unter gerade unter Zeit- und damit Kostengesichtspunkten kritisch, da jede Stunde die z.B. ein Consulter damit verbringt, Code zu verstehen, eine teure und unproduktive Stunde für den Auftraggeber ist . Um also die Verständlichkeit des Codes zu gewährleisten, sollte er so sauber strukturiert sein, dass die einzelnen Funktionalitäten und die Zusammenhänge innerhalb des Codes auch dem nicht mit der Software vertrautem Programmierer schnell klar werden [1]. Hier kommt Refactoring als ein Mittel zur Qualitätssicherung bei Software ins Spiel.

2.3. Definition von Refactoring:

„A Change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour." [1, S.53]

(Zu Deutsch: „Eine Veränderung an der internen Struktur einer Software, sodass die Arbeitsweise derselben einfacher nachvollziehbar ist und Modifizierungen an ihr einfacher durchzuführen sind, ohne jedoch dabei ihre äußere Funktionalität zu ändern.")

2.4. Anwendungsmotivation zum Refactoring

„Refactoring is the process of taking a running program and adding to it's value, not by changing it's behaviour or necessarily adding functionality but by giving it more of these qualities that enable us to continue developing at speed." [1, S.60]

Refactoring ist ein Werkzeug, das für mehrere Zwecke eingesetzt werden kann und soll [1]. Zum einen verbessert Refactoring das Design von Software [1, S.55]; durch wiederholte Änderungen an der Struktur und Funktionalität von Programmen, vor allem wenn diese Änderungen nur temporären Zielen dienen und zudem ohne ausreichende Kenntnis der Software und ihres Codes vorgenommen werden, kann es über einen längeren Zeitraum zu „software decay" (Software-Fäulnis) kommen. Ohne Refactoring verliert der Code seine Struktur, wird schwammig und unübersichtlich [1, S.55].

Zudem hilft Refactoring dabei, eventuelle Bugs im Code zu finden [1, S.57]. Sobald man Refactoring anwendet, muss man sich intensiv mit dem Code der zu bearbeitenden Software auseinandersetzen, was zusammen mit dem Zuwachs an Übersichtlichkeit zu einer verbesserten Fehlerfindung führen kann (siehe auch Abs.5).

Ein letzter, aber nicht unwichtiger Aspekt ist die Tatsache, dass Refactoring ein sehr gutes Instrument sein kann, um den Softwareentwicklungsprozess zu beschleunigen bzw. auf Geschwindigkeit zu halten. Besonders bei großen und komplexen Anwendungen hilft das Refactoring dabei, den Code strukturiert zu halten und gewährleistet damit auf mittel- bis langfristige Sicht einen kontinuierlich schnellen Codierungsvorgang[1, S.57]. Refactoring kann also ein sehr effektives Instrument für die Qualitätssicherung bei Softwareprodukten sein.

2.5. Einsatzmöglichkeiten von Refactoring

Refactoring ist in verschiedenen Situationen einsetzbar. Grundidee hierbei ist, Refactoring nicht als gesonderten Prozess zum Qualitätsmanagement anzusehen, sondern konsequent während des Codiervorganges Refactoring zu betreiben [1, S.58]. Insbesondere die code review [1, S.59] am Ende eines Programmiervorgangs dient als guter Meilenstein um den bereits geschriebenen Code nocheinmal zu überprüfen und aus dem Gesamtkontext heraus zu refaktorieren.

Eine auf diesem Konzept aufbauende Methode der ständigen code review und des kontinuierlichen Refactoring ist das sogenannte Extreme Programming [3], bei dem je zwei Programmierer gleichzeitig und zusammen an einem Rechner und vor einem Monitor arbeiten, um eine doppelte code review direkt in den Codierungsprozess einzubauen [1, S.59] und somit positiv auf die Softwarequalität einzuwirken.

Eine Möglichkeit, Refactoring auch bei UML-Diagrammen einzusetzen, gibt Dave Astels [6] vor. Hier wird insbesondere schon bei der Planung einer neuen Software Refactoring betrieben, da die zum Entwurf genutzten UML-Diagramme auf mögliche Verbesserungen in ihrer Struktur geprüft werden können. Zudem kann bei bereits geschriebenem Code die Umstrukturierung desselben besser geplant werden, was auch in Verbindung mit Traceability (siehe auch Referat „Qualitätssicherung durch Traceability“ [7]) von UML-Charts ein interessanter Ansatz ist.

2.6. Refactoring und Performance

Grundsätzlich kann Refactoring zu Performance-Verlusten führen [1, S.69]. Eine gründliche (Um)Strukturierung eines Programms kann also durchaus längere Laufzeiten bzw. höhere Hardwareanforderungen zufolge haben. Allerdings kann die Performance bei diesbezüglich sensitiven Systemen (z.B. Echtzeitsysteme), nach abgeschlossenem Refactoring häufig erheblich einfacher verbessert werden, als dies bei nicht bearbeiteten Systemen der Fall wäre. Durch die offensichtlichere Struktur des Codes lassen sich Laufzeit- oder Resourcenintensive Prozesse besser erkennen, zumal es sich hierbei häufig um relativ kurze Codeabschnitte handelt, deren „Bremswirkung“ ohne Refactoring möglicherweise nicht sofort ins Auge fallen würde [1, S.70]. Grundsätzlich gilt hier die Regel, daß 10% des Codes etwa 90% der Laufzeit ausmachen, also nur ein relativ kleiner Teil des Codes viele Resourcen verbraucht [5]. Demnach sollte Code zuerst refaktoriert werden, um anschließend die gewünschte

Performancesteigerung durch die von McConnell [4] beschriebene „hot spot“-Bearbeitung zu erreichen [1, S.70].
Daß hierbei hinsichtlich der Performancesteigerung allerdings auch Grenzen gegeben sind, wird in Abs. 6 noch verdeutlicht.

3. Ansatzpunkte für Refactoring

3.1. Wann und wo kann man refaktorieren? „Bad Smells Heuristics“

Es stellt sich nun die Frage, an welcher Stelle im Code eines vorliegenden Programmes refaktoriert werden kann, um den gewünschten Effekt aus Abs. 2.4 zu erzielen.
Fowler propagiert hierfür eine Methodik, die er „Bad Smell Heuristics“, also frei übersetzt „Auffinden schlechter Gerüche“, nennt [1, S.75]. Hierbei geht es tatsächlich um einen eher intuitiven als um einen wissenschaftlichen Ansatz [1, S.75], um die Codeteile zu erkennen, die durch Refactoring vereinfacht bzw. verbessert werden können. „Bad Smells“ („schlechte Gerüche“) sind Code-Teile wie z.B. Methoden, Klassen oder Hierarchien, die sich durch ein bestimmtes Merkmal („bad smell“) auszeichnen. Diese „Bad Smells“ sind ein Anhaltspunkt für eine Entscheidung durch den Programmierer, ob und an welchen Stellen ein Refactoring sinnvoll ist. Im weiteren Teil sollen hier einige dieser Anhaltspunkte („Bad Smells“) beschrieben werden.

3.2. Ausgesuchte „Bad Smells“ mit Beispielen:

3.2.1. Duplicated Code („Redundanter Code“)

Redundanter Code ist das am häufigsten auftretende Merkmal für schlecht strukturierten Code [1, S.76]. Wenn in einem Programm eine Code-Struktur mehrmals in gleicher oder sehr ähnlicher Form auftritt, so ist es wahrscheinlich, dass hier ein Fall von redundantem Code vorliegt. Ein einfaches Beispiel für redundanten Code wäre zum Beispiel zwei Methoden einer Klasse, die für unterschiedliche Eingabeparameter die gleiche Aufgabe erfüllen. Hier liegt die Idee nahe, die beiden Methoden zu einer einzigen zusammenzufassen.

3.2.2. Long Method („Lange Methode“)

Lange Methoden, in denen große Teile des Codes aus Verständnisgründen kommentiert sind, sind eine weiterer Anhaltspunkt für Refactoring. Längere Methoden sind grundsätzlich schwerer zu verstehen, da die Komplexität häufig mit dem Umfang steigt [1, S.77]. Fowler [1] befürwortet in solchen Fällen, Methoden zu stückeln, also aus einer langen und komplexen Methode, der Übersicht und Verständlichkeit halber mehrere Kleine zu machen. Deren Aufgabe und Arbeitsweise wiederum muss dann nicht mehr in Kommentaren erklärt werden, sondern soll aus dem Methodennamen hervorgehen[1, S.77].

3.2.3. Large Class („große Klasse”)

Klassen, die sehr viele Aufgaben erfüllen, können wiederum einen Ansatz zum Refactoring bieten. So kann eine zu große Klasse daran erkannt werden, dass sie zu viele Instanzvariablen besitzt, um noch übersichtlich zu sein. Zudem liegt bei einer übergroßen Klasse der Verdacht nahe, dass diese redundanten Code enthält. Ähnlich wie eine Klasse mit zuvielen Instanzvariablen ist eine Klasse mit zu viel Code gefährdet, unübersichtlich und chaotisch zu sein [1, S.78].

3.2.4. Shotgun Surgery („Schrotschuss-Operation“)

Dieser „bad smell“ liegt vor, wenn bei einer Änderung an einer Klasse viele kleine Änderungen an anderen Klassen vorgenommen werden müsse um die Funktionalität des Programms aufrechtzuerhalten. Ein grundlegendes Problem hierbei ist vor allem, alle anderen Programmteile ausfindig zu machen, in denen zusätzliche Änderungen notwendig sind. Dies kann zu ernsthaften Problemen führen [1, S.80].

3.2.5. Feature Envy („Neid um Leistungsmerkmale“)

Eine Grundlage der objektorientierten Programmierung ist es, Daten mit allen dazugehörigen Prozessen in einem Objekt zusammenzufassen. Wenn nun eine Methode einer Klasse ihre Daten überallher bezieht, nur nicht aus der Klasse der sie zugeordnet ist, besteht der begründete Verdacht, daß diese Methode der Klasse zugeordnet werden sollte, aus der sie die meisten Daten bezieht. [1, S.80]

4. Arten des Refactoring

Der Verständlichkeit halber sind bis jetzt nur typische Beispiele für schlechten Code zusammen mit rudimentären Lösungsansätzen gegeben worden. Fowler [1] beschreibt allerdings Lösungsansätze anhand von 70 Refactoring-Standardprozeduren, die zum Teil explizit auf bestimmte „Bad Smells“ zugeschnitten sind. Die erwähnten Prozeduren lassen sich grundsätzlich in fünf Kategorien einteilen [5]:

- Methoden umbauen
    - *z.B. durch Prozedur „Extract Method“ (Methode extrahieren)*
- Eigenschaften zwischen Objekten verschieben
    - *z.B. durch Prozedur „Extract Class“ (Klasse extrahieren)*
- Daten organisieren
    - *z.B. durch Prozedur „Replace Array with Object“ (Array durch Objekt ersetzen)*
- Bedingte Anweisungen vereinfachen
    - *z.B. durch Prozedur “Introduce Null Object“ (Null-Objekte einführen)*
- Methodenaufruf vereinfachen und Generalisierungen
    - *z.B. durch Prozedur “Extract Subclass” (Unterklasse extrahieren)*

Der Übersicht dieser Ausarbeitung zuliebe soll an dieser Stelle auf die Aufzählung und Erklärung jeder einzelnen der 70 Refactoring-Prozeduren verzichtet werden. Stattdessen soll die Methodik des Refactorings hier anhand eines einfachen Beispieles in javaähnlichem Pseudocode dargelegt werden.

4.1. Ein einfaches Beispiel für schlechten Code[1]

```
void Mario(String beta)
{
/* This method checks whether a String equals the Name “Mario” ”John” or “Jim” and
prints out which one of the names was found in the String*/

String m =”mario”;
String j1 = “john”;
String j2= “jim”;

        if beta.equals(m){
                System.out.println (“Mario gefunden”);
        }
        else if beta.equals(j1){
                System.out.println (“John gefunden”);
        }
        else if beta.equals(j2){
                System.out.println (“Jim gefunden”);
        }
        else {
                System.out.println (“Keiner der Namen gefunden”);
        }
}
```

4.2. Eine Verbesserung durch Refactoring[1]

```
void compareNames (String vorname) {

String name1  =”mario”;
String name2  = “john”;
String name3  = “jim”;

   if vorname.equals(name1) | vorname.equals(name2) | vorname.equals(name3){
         printName(vorname);
   }
   else {
         noName();
    }
}

void printName (String vorname){
         System.out.println („Der Name” + vorname + „wurde gefunden“);
}

void noName(){
         System.out.println („Es wurde keiner der Namen gefunden“);
}
```

[1] Codeausschnitte wurden unter Zuhilfenahme des Buches Java Gently [11] erstellt

4.3. Was ist passiert?

Die refaktorierte Version (4.2) des voranstehenden Codes (4.1) weist einige Veränderungen auf. So ist der tatsächliche Code (ohne Kommentare) scheinbar länger geworden, Methoden- und Variablendeklarationen haben sich verändert, es fehlt die Kommentare und aus einer Methode sind drei geworden.
Dies ist ein Beispiel für die konsequente Beseitigung von Bad Smells, unter anderem anhand der Refactorings „Extract Method“ [1, S.110] und „Rename Method“ [1, S.273]. Die Veränderungen führen in diesem kleinen Rahmen (der Code ist aus Verständlichkeitsgründen bewußt simpel gehalten) nicht notwendigerweise zu wesentlich besserer Überschaubarkeit, können aber bei größeren und komplexeren (also realistischen) Codeabschnitten einen erheblichen Beitrag zur Transparenz leisten.

4.4. Refactoring bei UML-Diagrammen

4.4.1. Ein schlecht modelliertes Zustandsdiagramm[1]

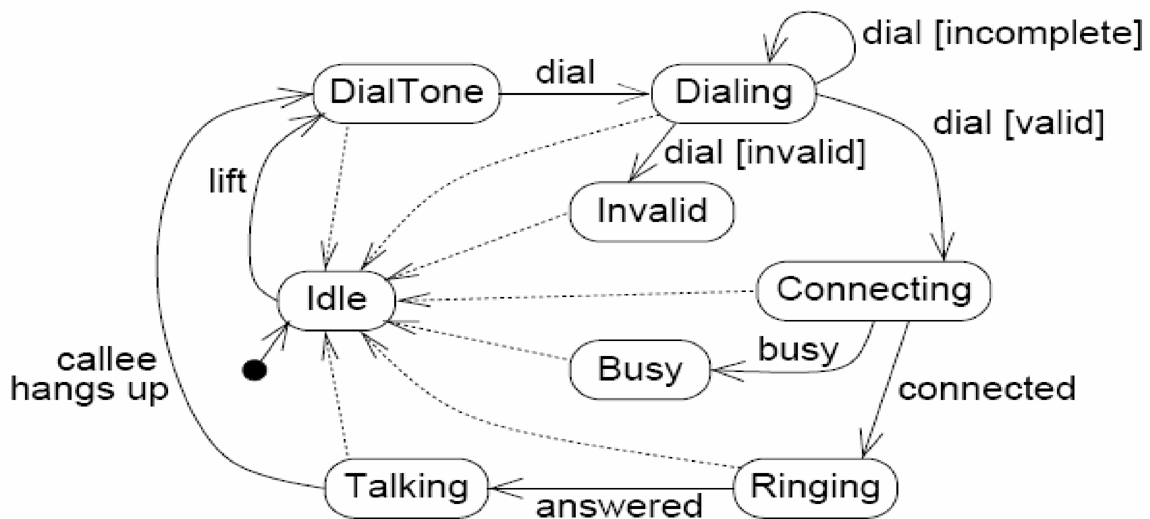

Figure 3. Initial phone state diagram (dotted transitions are triggered when the caller hangs up).

4.4.2. Eine Verbesserung durch Refactoring[1]

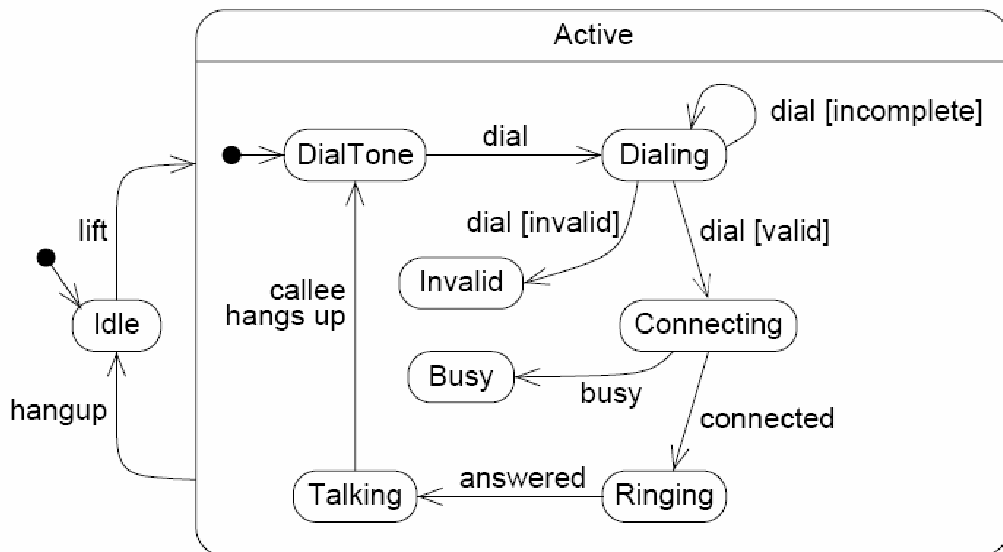

Figure 4. Final phone state diagram.

Hier wurde Refactoring für UML-Diagramme angewandt, da das Ausgangsmodell unübersichtlich und hinsichtlich der Erweiterung um neue Funktionalitäten der Telefonanlage unzureichend strukturiert war.
Dazu wurde ein neuer Superstate „Active“ implementiert, aus dem dann der Zustand „Idle“ sowie der Startzustand herausgenommen wurden. Durch die Entzerrung des Aufhängevorgangs wurde das gesamte Modell wesentlich vereinfacht.

[1] Die Diagramme entstammen aus dem Manuskript “Refactoring UML Models“ [8]

5. Vorgehensweise beim Refactoring - Tests

„It is essential for refactoring that you have good tests. [...] Before you start refactoring, check that you have a solid suite of tests. These tests should be self checking." [1, S.8]

Einer der wichtigsten Schritte beim Prozess des Refactorings ist das Testen. Hierbei sollen Blackbox-Tests durchgeführt werden, um einen beim Refactoring veränderten Programmausschnitt auf seine unveränderte Funktionalität zu überprüfen (siehe: 2.3 Definition von Refactoring). Durch Refactoring können unter Umständen Fehler in das Programm eingebracht werden, welche wiederum zu Fehlfunktionen oder anderen Problemen führen könnten („bugs"). Durch eine iterative Vorgehensweise, also einem Wechsel zwischen (gegebenenfalls) Programmieren, Refaktorieren und dem Testen des refaktorierten Codes nach Abschluss jeder Refactoringprozedur soll dies verhindert werden [1, S.7-8].
Fowler [1] präferiert hierbei sich selbst überprüfende, automatisierte Tests [1, S.90]. Diese haben gegenüber manuellen Tests die Vorteile, durch flexibilität und problemlose Mehrfachverwendung („repeated use") sowie ihre Hilfe bei der Fehlerfindung („bug detector") die Produktivität bei der Softwareerzeugung erhöhen [1, S.90-91]. Dieser Vorteil wiegt den Zeitaufwand für das Schreiben der Tests insbesondere bei größeren Projekten auf [1, S.89-90]. Prinzipiell sollen diese automatisierten Tests den eigentlichen Programmablauf simulieren, dabei jedoch Ein- und Ausgaben der getesteten Codeteile auf Korrektheit überprüfen und gegebenenfalls dokumentieren.

6. Refactoring Tools

Eine andere Möglichkeit zu refaktorieren ist es, Refaktoring Werkzeuge („Tools") einzusetzen, die den Programmierer bei der Arbeit unterstützen. Hier gibt es, abhängig von der jeweiligen Programmiersprache, einige Programme auf dem Markt. Als wesentliche Werkzeuge seien hier der Refactoring-Browser für Smalltalk [9] sowie IntelliJ Idea [10] genannt. Beide Tools bieten Funktionen zum Refaktorieren an, die z.B. bei dem Verschieben von Methoden automatisch die programminternen Zuweisungen verändern, was unter anderem wegen der Vermeidung von Aufwändigen Tests (siehe Abs. 5) für den Programmierer eine erhebliche Erleichterung darstellt.

Einen interessanten Ansatz bietet die Idee, den Computer anhand von UML-Diagrammen Code generieren zu lassen (wie z.B. bei Fujaba), da sich hier die Möglichkeit bietet anhand der Diagramme zu refaktorisieren. Dies kann zum Beispiel Vorgänge wie das Verändern von Vererbungsstrukturen o.ä. beim Refaktorisieren erheblich erleichtern, da die Veränderungen besser visualisiert werden und so für viele einfacher nachvollziehbar sind [6].

7. Refactoring in der Praxis

7.1. Grenzen des Refactoring

Grundsätzlich ist Refactoring immer und überall dort anzuwenden, wo objektorientiert Programmiert wird. Probleme können allerdings auftreten, wenn Echtzeitsysteme im Spiel sind, da diese durch das Refactoring an Performance einbußen können. Dieser Performanceverlust ist nicht in jedem Fall durch Optimierung zu beheben.

Probleme treten auch im Zusammenhang mit Datenbanken auf, da hier unter Umständen am Datenmodell bzw. dem Datenbankschema Änderungen vorgenommen werden müssen. Da Datenbanken insbesondere in Unternehmen häufig eine zentrale Rolle in den Informationssystemen spielen, können Veränderungen an ihnen nur schwer vornehmbar, oder gar unmöglich sein [1, S.62-64]. Hier ist Refactoring nicht immer anwendbar.

Ein ganz wichtiger Aspekt ist die Behandlung von Designfehlern bei Software. Eine schlecht entworfene Software kann unter Umständen nicht unmittelbar durch Refactoring verbessert werden. Häufig sollte in diesem Fall die Softwarearchitektur nocheinmal überdacht werden [1, S.65-66]. Fowler [1] schreibt dazu allerdings, bislang nicht über genug profunde Daten zu verfügen, um Richtlinien für einen solchen Fall aussprechen zu können [1, S.65].

Zu den zwei Fällen, in denen auf Refactoring verzichtet werden sollte, zählt Fowler [1] zum einen Software, die als ganzes oder in Teilen nicht funktioniert bzw. nicht die gewünschten Aufgaben erfüllt. In diesem Fall ist eine komplette Neuerstellung des Codes meist unumgehbar [1, S.66]. Eine Möglichkeit um in diesem Fall eine Abwägung zwischen Refactoring und Neuerstellung zu treffen ist es, den bestehenden Code in viele verkapselte Komponenten zu zerlegen und dann jede Komponente für sich auf Funktionalität zu überprüfen [1, S.66]. Wenn im Anschluss daran viele der

Komponenten nicht funktional sind, sollte der Code neu erstellt werde. Allerdings gibt Fowler[1] auch hier an, sich nicht auf fundierte Daten, sondern auf Erfahrungswerte zu berufen [1, S.66].

Der zweite Fall, in dem grundsätzlich auf Refactoring verzichtet werden sollte, ist der Zwang zur kurzfristigen Einhaltung einer Deadline, da hier das Ziel, die Software zum vorgegebenen Zeitpunkt voll funktionsfähig abzuliefern, über dem Ziel der guten Strukturierung durch Refactoring steht. Ein unvollständig refaktoriertes Programm ist im Regelfall noch nicht voll funktional und kann deshalb zu einem Nichteinhalten der Deadline führen, was besonders aus betriebswirtschaftlicher Sicht extrem problematisch sein kann [1, S.66].

7.2. Zusammenfassung

Refactoring ist ein wirksames Mittel zur Qualitätssicherung bei Softwareprodukten, insbesondere bei solchen, die in einer objektorientierten Programmiersprache codiert worden sind. Refactoring verbessert das Design von bereits geschriebenem Code und wirkt dadurch dem Verfall von Software über die Zeit entgegen.
Der durch nachträgliche Veränderungen am Code hervorgerufene Effekt der immer schlechter werdenden Strukturierung und Übersichtlichkeit wird durch Refactoring minimiert oder sogar aufgehoben. Dadurch ermöglicht Refactoring die Sicherung der Softwarequalitätsmerkmale Erweiterbarkeit, Anpassbarkeit und Wiederverwendbarkeit und rentiert sich somit besonders auf mittel- und langfristige Sicht.
Desweiteren unterstützt ein geordnetes Design von Softwarecode den Entwickler bei der Fehlersuche und kann es ihm ermöglichen, die Laufzeiten der Software zu verbessern.
In der Praxis ist Refactoring insofern auch aus betriebswirtschaftlicher Sicht interessant, da es im Regelfall wesentlich weniger an Zeit, Aufwand und somit letztendlich Geld kostet eine Software durch Refactoring in Ordnung zu halten, als eine durch Softwareverfall unerweiterbar und damit unbrauchbar gewordenen Software weiterzupflegen.

9. Quellen

[1] Martin Fowler, “Refactoring: Improving the Design of existing code”,
Addison-Wesley 1999

[2] Demeyer und Bosch, “Object-Oriented Technology, ECOOP’98 Workshop Reader”,
Springer, pp. 483-485

[3] Kent Beck, “Extreme Programming Explained: Embrace Change”,
Addison-Wesley 1999

[4] Steve McConnell, “Code Complete: A practical Handbook of Software Construction”,
Microsoft Press 1993

[5] Prof. Bruno Schäffer, “Objektorientierte Softwareentwicklung – Refactoring”,
http://people.canoo.com/schaeffer/vl03/handouts/Refactoring.pdf

[6] Dave Astels, “Refactoring with UML”,
http://www.agilealliance.com/articles/articles/RefactoringWithUML.pdf

[7] Thomas Minzenmay, Seminararbeit “Qualitätssicherung durch Traceability”,
Universität Paderborn, 2004

[8] Gerson Sunyé et al., “Refactoring UML Models”
http://www.irisa.fr/triskell/publis/2001/Sunye01b.pdf

[9] Refactoring Browser for Smalltalk
http://st-www.cs.uiuc.edu/users/brant/Refactory/

[10] IntelliJ Idea, Java Development Tool
http://www.jetbrains.com/idea/

[11] Judith Bishop, “Java Gently”, Third Edition,
Addison-Wesley 2001

[12] Refactoring Homepage
www.refactoring.com