Refactoring Rails

Studienarbeit

Studienbereich Wirtschaft

im Studiengang Wirtschaftsinformatik

an der Berufsakademie

Ravensburg

Verfasser: Cornelius Knall

Kurs:WI2005-2

Datum: 13.02.2008

Inhaltsverzeichnis

1 Einleitung

Der Ausdruck „Refaktorisieren“ (Refactoring) kommt aus den Kreisen der *Small-Talk* Entwickler, wurde aber sehr schnell in anderen Programmiersprachen übernommen. Insbesonders in Frameworks ist Refaktorisieren ein grundlegender Bestandteil. Da beim Refaktorisieren Änderungen am laufenden Code gemacht werden, ist dies sehr riskant, da sich subtile Fehler einschleichen können. Refaktorisieren hilft dem Entwickler dabei Fehler zu vermeiden, bzw. auf Fehler aufmerksam zu machen und hilft den Code besser zu verstehen und ihn in eine strukturierte Form zu bringen.

Diese Studienarbeit soll dem Leser einen Überblick geben, was Refaktorisieren bedeutet und wie es am effektivsten angewandt wird. Zu Beginn wird erläutert was Refaktorisieren bedeutet, und weshalb Refaktorisieren eingesetzt werden sollte. Da Testen beim Refaktorisieren unabdingbar ist, werden verschiedene Methoden vorgestellt, die dem Entwickler helfen Tests zu entwerfen, bzw. Code zu entdecken der nicht durch Tests abgedeckt ist. Diese Methoden werden an spezifischen Tools von *Ruby on Rails* dargestellt.

Im Hauptteil der Arbeit werden verschiedene Methoden des Refaktorisieren vorgestellt. Zum einen werden diese Methoden allgemein erklärt und an geeigneter Stelle werden die Methoden speziell für *Ruby on Rails* beschrieben. Am Ende dieser Arbeit werden die Vor- und Nachteile des Refaktorisieren diskutiert.

Wichtigstes Ziel der vorliegenden Studienarbeit ist es, dem Leser einen grundsätzlichen Einblick in die Technologie des Refaktorisierens an Hand von *Ruby on Rails* zu geben und deutlich zu machen, wie Refaktorisieren effektiv eingesetzt werden kann. Sie soll Interesse wecken und einen Einblick schaffen, der den Einstieg erleichtert.

Um diese Studienarbeit inhaltlich zuverstehen, sollte der Leser grundlegende Kenntnisse in der objektorientierten Softwareentwicklung mitbringen. Da im Verlauf dieser Arbeit auf Klassen, Methoden und Attribute, und wie diese durch Refaktorisieren bearbeitet werden, eingegangen wird.

2 Was ist Refaktorisieren

Refaktorisieren ist der Prozess, ein Softwaresystem so zu verändern, dass das Programm dasselbe Verhalten hat wie vor dem Refaktorisieren. Der Code aber eine bessere interne Struktur erhält[1]. Refaktorisieren ist ein strukturiertes Vorgehen , um den Code zu bereinigen, und die damit auftretenden Fehler zu minimieren. Im Kern verbessert Refaktorisierung das Design des Codes, nachdem er geschrieben wurde[2].

Wenn heute Software entwickelt wird, wird zuerst das Design entworfen und modelliert. Erst nach dem alles modelliert wurde, wird mit der Programmierung begonnen. Im Laufe der Zeit wird dieser Code verändert, und die Integrität des Systems und seine Entwurfs gemäße Struktur schwindet. Dies geht soweit, dass schließlich die Qualität des Codes von dem entworfenen „Ingenieursniveau“ nun schlussendlich auf ein „Hackerniveau“ gesunken ist.

Refaktorisieren ist das Gegenteil dieser Gepflogenheit[3]. Wird Refaktorisieren eingesetzt, so hilft Refaktorisieren den Code zu verbessern. Refaktorisieren ist einfach, wenn nicht sogar primitiv. Beim Refaktorisieren, wird ein Feld aus einer Klasse in eine andere geschoben. Code wird aus einer langen Methode gelöscht und eine neue kurze Methode wird erstellt. Code wird entlang der Vererbungshierarchie verschoben, so wird der gesamte Code bereinigt und übersichtlicher gestaltet. Jedoch wird die Funktionsweise der Software nicht beeinflusst. Refaktorisieren hat den Arbeitsschwerpunkt des Verschiebens.

Das Design der Software, entsteht während der Entwicklung. Das Design wird während der Entwicklung kontinuierlich verbessert und bleibt auch bei fortschreitender Entwicklung gut[4].

[1] vgl. Fowler, Refactoring, S. 39 (2005)
[2] vgl. Fowler, Refactoring, S. 40 (2005)
[3] vgl. Fowler, Refactoring, S. 48 (2005)
[4] vgl. Fowler, Refactoring, S. 50 (2005)

3 Prinzipien des Refaktorisierens

Refaktorisierung ist nicht das „Allheilmittel“ um jede „Softwarekrankheit“ zu kurieren, aber Refaktorisieren ist ein sehr nützliches Werkzeug um einen geschrieben oder zu überarbeitenden Code besser in den Griff zu bekommen. Refaktorisieren ist ein Werkzeug, das für verschiedene Aufgaben eingesetzt werden kann und sollte[5].

Der alleinige Einsatz von Refaktorisieren funktioniert, ist aber nicht die effektivste Art der Entwicklung. „Extreme Programmer“[6] wird häufig unterstellt, dass sie diesen Ansatz befürworten. Jedoch entwerfen auch extreme Programmierer bevor die eigentliche Entwicklung beginnt. Sie vertreten die Sicht, dass jede spätere Änderung am Design teuer ist. Die Verschiebung des Schwerpunktes von der *perfekten Lösung* hin zu einer *sinnvolle Lösung* wird durch das Refaktorisieren ausgelöst.Und dadurch sind Änderungen am Design nicht mehr teuer und aufwendig, sondern werden kontinuierlich während der Entwicklung vollzogen[7].

3.1 Verbesserung des Designs

Jede Änderung am Code, sei es eine Implementierung neuer Methoden oder eine Änderung um kurzfristige Ergebnisse zu erreichen, verschlechtern das Design. Es wird schwieriger das Design zu erkennen und zu verstehen. Ist dies nicht mehr gewährleistet, so verschlechtert sich die interne Struktur des Codes. Refaktorisieren ist wie Code aufräumen, indem Dinge an die richtigen Stellen verschoben werden. Regelmäßiges Refaktorisieren hilft den Programmierern den Code in seiner angedachten Form zuhalten.

Ein wichtiger Aspekt bei der Verbesserung eines Designs ist das filtern und löschen von redundantem Code. Ein Indiz für schlecht geschriebenen Code ist, dass mehrere Befehle benötigt werden, die an verschiedenen Stellen das gleiche bewirken. Dabei ist zu erwähnen, dass die Verringerung von Code sich nicht auf die Performance auswirkt. Jedoch hat die Eliminierung von Duplikaten eine große Auswirkung auf

[5] vgl. Fowler, Refactoring, S. 54 (2005)
[6] vgl. Kent, Extreme Programming, S. 60 (2000)
[7] vgl. Fowler, Refactoring, S. 57 (2005)

die Lesbarkeit und Änderbarkeit des Codes. Durch die Eliminierung von Duplikaten, wird ausgeschlossen, dass eine Änderung nicht das gewollte Ergebnis liefert. Bei Duplikaten kann es vorkommen, dass eine Änderung nicht das erwartete Ergebnis zurück liefert, weil an einer anderen Stelle der gleiche Befehl das gleiche tut nur in einem anderen Kontext, und diese Stelle nicht geändert wurde. Durch die Eliminierung von Duplikaten wird sicher gestellt, dass der Code alles nur einmal und wirklich nur einmal sagt. Das ist der Kern eines guten Designs[8].

Ein Kernpunkt von Ruby on Rails ist das *DRY-Prinzip*. Die Abkürzung *DRY* sthet für *Dont't repeat yourself* und wurde von Dave Thomas und Andy Hunt[8] geprägt. Das DRY-Prinzip besagt, dass weder Daten noch Funktionalitäten redundant vorkommen sollten. Rails setzt dieses DRY-Prinzip in allen Bereichen um[9].

3.2 Verbesserung der Wartbarkeit

Refaktorisieren hilft nicht nur beim Verbessern des Design, sondern erhöht auch die Lesbarkeit des Codes und somit wird der Code für andere Entwickler leichter verständlich. Durch den Einsatz des Refaktorisierens schon bei der Entwicklung, wird vermieden, dass sich Änderungen, die zu einem späteren Zeitpunkt gemacht werden müssen, nur mit hohem Zeitaufwand realisieren lässt. Primär Ziel vieler Entwicklungen, ist es schnell eine lauffähige Applikation zu erstellen. Die Wartbarkeit tritt sehr häufig in den Hintergrund. Setzt man bei der Entwicklung Refaktorisieren ein, so wird der Code besser verständlich, und der Zeitaufwand ist dabei gering. Programmieren in diesem Sinne heißt alles zu tun, um klar zu sagen, was der Entwickler meint.

Ein weiterer Punkt ist die Fehlererkennung. Die Fehlererkennung durch reines Lesen des Quellcodes ist recht mühsam und nicht jeder Entwickler findet dabei alle Fehler. Durch die strukturierte Vorgehensweise des Refaktorisierens, stößt der Entwickler automatisch auf Fehler und kann diese beheben.

Die obigen Punkte stellen den praktischen Nutzen deutlich heraus: Refaktorisieren

[8] vgl. Fowler, Refactoring, S. 43 (2005)

[9] vgl.Wirdemann, Rapid Web Development mit Ruby on Rails, S.8 (2007)

hilft dem Entwickler schneller zu programmieren. Dies hört sich auf den ersten Blick paradox an, da beim Entwickeln selbst schon eine gewisse Zeit aufgebracht werden muss um zu refaktorisieren, aber die Zeitersparnis, wenn es um Weiterentwicklungen oder Änderungen durch andere Entwickler, geht ist ausschlaggebend.

Ein gutes Design ist entscheidend, um das Tempo der Softwareentwicklung aufrechtzuhalten. Refaktorisieren hilft Software schneller zu entwickeln, weil es den Verfall des Design des Systems stoppt. Es kann das Design sogar verbessern[10].

3.3 Wann Refaktorisieren

Wenn über Refaktorisieren gesprochen wird, kommt meist die Diskusion auf, wann refaktorisiert werden soll. Soll nun ein bestimmter Zeitplan aufgestellt werden, bzw. soll in bestimmten Intervallen refaktorisiert werden. Refaktorisierung sollte nicht in einen bestimmt Ablaufzyklus geschehen. Viel mehr ist Refaktorisieren ein fortlaufender Prozess bei der Entwicklung von Software[11].

Ein guter Zeitpunkt zum Refaktorisieren ist, wenn in eine bereits fertige Software Funktionen hinzugefügt werden müssen. Zum einen versteht der Entwickler den Code besser, zum anderen kann der Code für spätere Implementierungen vorbereitet werden. Ein naheliegender Grund um zum Refaktorisieren zu greifen, wäre wenn Fehler aus einer Software eliminiert werden müssen. Um diese Fehler zu finden und zu beheben, ist diese Technik von Vorteil. Ein letzter Punkt wann es von Vorteil sein kann zu Refaktorisieren, sind die so genannten *Code-Reviews* (siehe 5.6). Dies ist ein guter Zeitpunkt, da mehrere Entwickler den Quellcode analysieren.

3.4 Wann nicht Refaktorisieren

Nun stellt sich die Frage, ob man immer Refaktorisieren sollte. Es gibt ein paar wenige Ausnahmen, Ausnahmefälle in denen nicht refaktorisiert werden soll. Zu welchem Zeitpunkt und wie Refaktorisiert werden soll, muss jeder Entwickler für sich entscheiden. Ein gutes Beispiel, wann nicht refaktorisiert werden soll, ist, wenn

[10] vgl. Fowler, Refactoring, S. 58 (2005)
[11] vgl. Fowler, Refactoring, S. 61 (2005)

der zu bearbeitende Code so schlecht strukturiert ist, dass die Entwicklungszeit durch Neuimplementierung geringer sein wird. Auch sollte Refaktorisieren vermieden werden, wenn das Projekt kurz vor Fertigstellung steht. Muss zu diesem Zeitpunkt refaktorisiert werden, ist dies ein Indiz dafür, dass im Design oder in der Entwicklung etwas misslungen ist. Hier bietet es sich an Refaktorisierung, nach der Auslieferung des Produktes, durch Updates etc. zu gewährleisten.

4 Tests aufbauen

Wenn refaktorisiert werden soll, sind gute Test unabdingbar. Selbst wenn dem Entwickler ein gutes Testframework, dass die Refaktorisierung weitgehend automatisiert, zur Verfügung steht, sind immer noch Test notwendig. Es wir noch einige Zeit in Anspruch nehmen, bis diese Tools alle möglichen Refaktorisierungen automatisieren. Entwickler sollten dies nicht als Nachteil betrachten, sondern als Gelegenheit das Programmiertempo zu steigern.

4.1 Warum testen

Der Hauptgrund, warum getestet werden soll, ist sicherlich die mühselige Fehlersuche. Die meiste Zeit bei der Entwicklung von Applikationen nimmt die Fehlersuche in Anspruch, das eigentliche Programmieren selbst, nimmt neben der Modellierung nur wenig Zeit in Anspruch. Es gibt bestimmt keine Entwickler die Fehlerfrei programmieren. Jeder Programmierer kann von zeitaufwändigen Fehlersuchen berichten. Jedoch hat Testen eine abschreckende Wirkung. Um selbsttestende Test zuschreiben, wird Zeit in Anspruch genommen, da zusätzlicher Code in die Klassen implementiert werden muss. Ein anderes „Übel“ ist, dass manuelle Testen.

Am nützlichsten ist es Tests schon zu Beginn der Entwicklung zu schreiben. Ein Dogma des Refaktorisierens ist: „Kein Refaktorisieren ohne Tests“. Die Idee des häufigen Testen ist ein wichtiger Teil des extremen Programming[12].

[12] vgl. Beck, Extreme Programming, S. 60 (2000)

Und hier kommt ein Vorteil von Ruby on Rails ins Spiel, dieses Framework liefert Tools zum Test mit, z.B. rCov (siehe 4.2), Autotest (siehe 4.3) oder Unit-Tests (siehe 4.4). Im folgenden wird ein Überblick über diese Verfahren gegeben, da eine genauere Betrachtung den Rahmen dieser Studienarbeit sprengen würde.

4.2 rCov: code coverage for Ruby

rCov[13] ist im klassischen Sinne kein Werkzeug, welches Tests erzeugt. Jedoch ist es ein sehr nützliches Tool um heraus zu finden, welche Segmente im Code nicht durch Tests abgedeckt sind. rCov wird normalerweise dazu benutzt, um Code Segment zu filtern, die nicht durch Tests abgedeckt werden. rCov unterstützt den Entwickler, von Ruby Programmen, seinen Code besser zu verstehen und zu optimieren. Kent Beck behauptet, dass nur Code existiert, der durch Tests abgedeckt und validiert ist. Code welcher zwar vorhanden ist, jedoch nicht durch Testfälle abgedeckt und gesichert ist, existiert nicht[14].

4.3 Autotest

Autotest[15] ist ein Testtool, dass im Hintergrund läuft und arbeitet. Autotest ist Bestandteil der Testsuite „ZenTest“. Einmal gestartet, arbeitet Autotest im Hintergrund und testet in Echtzeit die vom Entwickler programmierten Methoden. Der Entwickler ist in der Lage zeitnah seine Änderungen überprüfen zu lassen, und muss nicht manuell seine Änderungen im Browser testen.

Wie bei jeder Testumgebung, müssen zuerst die notwendigen Test geschrieben werden. Diese Test dienen Autotest als Vorlage für die Überprüfung des Codes. Jedesmal, wenn der Entwickler Änderungen an seinem Code speichert, testet Autotest diese automatisch im Hintergrund und gibt die jeweiligen Statusberichte auf der Konsole aus.
Autotest ist ein „must-have“ für jeden Ruby on Rails Entwickler.

[13] weitere Informationen: http://eigenclass.org/hiki.rb?rcov
[14] vgl. Beck, Extreme Programming, S. 64 (2000)
[15] weitere Informationen: http://nubyonrails.com/articles/autotest-rails

4.4 Unit-Tests

Mit dem Aufkommen der agilen Software-Entwicklung, insbesondere „Extreme Programming“, ist eine neue Art des Testen entstanden: *Unit Tests*

Diese besondere Art des Testen begründet sich darin, dass keine manuellen Tests gemacht werden, sondern das die Tests programmiert werden. So wird gewährleistet, dass diese Tests immer wieder aufs neue automatisch ausgeführt werden können. Diese Art von Tests gilt als Voraussetzung für das Refaktorisieren, insbesondere bei dynamischen Sprachen wie Ruby. Von besonderer Bedeutung sind diese Tests, da hier ein automatisches und sicheres Refaktorisieren nicht ohne weiteres möglich wäre [16].

Inspiriert durch das im Java Umfeld weit verbreitete Testframework JUnit, wurden in den letzten Jahren immer mehr Testframeworks für andere Sprachen entwickelt, PHPUnit für PHP. Das Framework für Unit Tests in Ruby heißt *Test::Unit* und ist seit Ruby 1.8.1 fester Bestandteil der Distribution. Im Gegensatz zu Autotest, welches im Hintergrund läuft, muss Test::Unit jedesmal, wenn getestet werden soll, manuell gestartet werden.

5 Refaktorisierung

5.1 Übel riechender Code

In den Abschnitt zuvor wurde erläutert, was Refaktorisieren ist und wann es eingesetzt werden soll. Des weiteren wurde gezeigt, wie Test das Refaktorisieren erleichtert und unterstützt. In den nächsten Abschnitten wird nun beschrieben, was für Indizien es gibt, um heraus zu finden ob refaktorisiert werden soll und wie die Vorgehensweise ist. In diesem Abschnitt wird nun der Sachverhalt dargelegt, woran der Entwickler erkennt, das refaktorisiert werden soll.

Das Anwenden der verschiedenen Methoden zum Refaktorisieren ist für einen erfahrenen Entwickler kein großes Problem. Das sind einfache Dinge. Jedoch zu erkennen

[16] vgl. Wirdemann, Rapid Web Development mit Ruby on Rails, S.327 (2007)

wann diese Dinge getan werden sollten, ist keine solche Routineaufgabe. Dem Entwickler soll das Gespür gegeben werden, Dinge im Code zu sehen, die durch Refaktorisieren erheblich verbessert werden könnten[17].

Im folgenden werden 4 Indizien beschrieben, die einem Entwickler die Suche nach „übelriechendem Code“ erleichtert, bzw. unterstützt.

5.1.1 Duplizierter Code

Nach Fowler (Refactoring, 2005, S.68) „Nummer eins in der Gestanksparade ist duplizierter Code [...]“. Wenn die gleiche Codestruktur an mehreren Stellen im Code vorkommt, deutet dies darauf hin, dass durch Entfernen der Duplikate der Code besser wird.

Das einfachste Problem mit dupliziertem Code ist, wenn der gleiche Code in verschiedenen Methoden einer Klasse vorkommt. Dann muss nur *Methode extrahieren* (siehe 5.2.3) angewandt werden. Nun muss die neue Methode in den alten Methoden richtig referenziert werden.

Ein weiteres häufiges Problem ist, wenn der gleiche Ausdruck in zwei voneinander erbenden *Unterklassen* vorkommt. Hier sollte zuerst wieder in beiden Unterklassen *Methode extrahieren* (siehe 5.2.3) angewandt werden, und anschließend, falls es notwendig ist *Attribut verschieben* (siehe 5.3) erfolgen.

5.1.2 Lange Methode

Ein Programm das stabil und performant läuft, ist ein Programm mit den kürzesten Methoden[18]. Eine Methode ist umso schwerer zu lesen und zu verstehen, umso länger sie ist. Deshalb sollten Methoden kurz und knapp sein, aber das wesentliche erfüllen. Um dieser Vorgabe gerecht zu werden, wird auch hier *Methode extrahieren* (siehe 5.2.3) angewandt. Hier wird nach Teilen der Methode gesucht, die gut zusammen passen. Wurden solche Ausdrücke gefunden, werden sie zusammengefasst und in eine neue Methode extrahiert. So wird aus einer langen Methode viele kleinere Methoden

[17] vgl. Fowler, Refactoring, S. 68 (2005)
[18] vgl. Fowler, Refactoring, S.69, (2005)

generiert.

5.1.3 Große Klasse

Wenn eine Klasse versucht zu viel zu tun, zeigt sich dies oft daran, dass die Klasse viele Instanzvariablen hat. Hat eine Klasse viele Instanzvariablen, deutet dies auf duplizierenden Code hin. Wie eine Klasse mit zu viel Instanzvariablen ist auch eine Klasse mit zu viel Code eine Gefahr für duplizierten Code. Eine einfache Lösung ist die Redundanz zu beseitigen. Die geeigneten Lösungen sind hier *Klasse extrahieren*.

Klasse extrahieren bedeutet, wenn eine Klasse die Arbeit, die von zwei anderen Klassen zu erledigen wäre, erledigt. Es wird eine neue Klasse erstellt, und die zu verschiebenden Attribute und Methoden, werden in die neue Klasse verlegt und entsprechend referenziert.

5.1.4 Lange Parameterliste

Früher wurde alles, was eine Funktion benötigt, als Parameter übergeben. Jedoch bei Objekten, braucht man dies nicht mehr. Wird in einer Methode ein Parameter gebraucht, der einem nicht zur Verfügung steht, so kann ein neues Objekt erstellt werden, welches diese fehlenden Parameter nach lädt.

Diese langen Parameterlisten können durch explizite Methoden ersetzt werden. D.h. es stehen keine Parameter zur Verfügung, sondern es werden ganze Methoden übergeben. Sollten sich jedoch die Parameter häufig ändern, so ist von dieser Methode abzusehen.

5.2 Methode verschieben

Eine der wichtigsten, wenn nicht sogar die wichtigste Entscheidung im Objektdesign betrifft die Verteilung der Verantwortlichkeiten[19]. Dies bedeutet, es soll vermieden werden, dass eine Methode sehr oft auf Attribute anderer Klassen zugreifen. Es ist aber nicht immer möglich dies schon im Entwurf zu realisieren, sondern fällt oft erst bei der Implementierung oder sogar noch später auf. In der Entwurfsphase sind

[19] vgl. Fowler, Refactoring, S. 139 (2005)

zukünftige Weiterenwicklungen nicht bekannt, und so kann zukünftig eine Methode verstärkt Attribute einer anderen Klasse benutzen. Wenn einem Entwickler diese Problematik bewusst wird, so sollte die Refaktorisierungmethode *Methode verschieben* eingesetzt werden.

Methoden zu verschieben ist die grundlegende Art des Refaktorisierens. Methoden zu verschieben wird dann angewandt, wenn Klassen zu viel Verhalten haben, oder wenn Klassen zu eng miteinander arbeiten. In dem Methoden verschoben werden, so werden Klassen einfacher und die Verantwortlichkeiten ersichtlicher.

Methode verschieben ist nur eine Möglichkeit um Methoden zu refaktorisieren. Martin Fowler hat hierzu in seinem Buch „Refactoring“ noch einige mehr, jedoch würde die Beschreibung dieser Refaktorisierungen den Rahmen dieser Arbeit sprengen.

Im folgenden wird die Vorgehensweise von *Methode verschieben* für das *Model* und den *Controller* beschrieben.

5.2.1 Methoden verschieben (Model)

Warum sollten Methoden im Model verschoben werden:
Eine Methode nutzt mehr Attribute einer anderen Klasse, als die eigenen[20].
Vorgehensweise:

1. Untersuchung aller Attribute der Ausgangsklasse, die die Ausgangmethode benutzen. Wird festgestellt, dass diese Attribute mit verschoben werden können, d.h. die Attribute haben keine weiteren Referenzen, dann werden die Attribute gleich mit verschieben. Wird ein Attribut von einer anderen Methode referenziert, so ist abzuwägen, ob diese Methode auch verschoben werden kann.

2. Deklarieren der neuen Methode in der anderen Klasse

3. Kopieren des Codes der Ausgangsmethode in die neue Zielmethode. Anpassung der neuen Methode, damit diese funktionsfähig wird, d. h. Anpassungen aller Referenzen und Assoziationen in der neuen Methode. Gegebenenfalls muss eine Migration erstellt werden.

[20] vgl. Trotter, Cashion, Refactoring Rails, RailsConf (2007)

4. Die Ausgangsmethode zu einer delegierenden Methode machen
5. Testen
6. Wenn die Möglichkeit besteht, die alte Methode zu entfernen, müssen alle vorhandenen Referenzen auf die neue Methode angepasst werden
7. Testen

5.2.2 Methoden verschieben (Controller)

Warum sollten Methoden im Controller verschoben werden:
Ein Indikator zum Verschieben von Methoden im Controller ist, wenn eine Methode ein Objekt benutzt, dass von einem anderen Controller beherrscht wird. Dies ist eine gewöhnliche Refaktorisierung um ein sogenannte RESTful Design zu erlangen[21].

RESTful ist der Ausdruck für ein Design, dass nur die 7 Basis Operationen: index, show, new, create, edit, update, and destroy im Controller verwendet[22].
Vorgehensweise:

1. Feststellen ob die zu verschiebende Methode verschoben werden kann. Wenn Referenzen auf diese Methode bestehen, die nicht verschoben werden können, dann sollte die Methode nicht verschoben werden.
2. Methode nach neuen Controller kopieren
3. Alle notwendigen Referenzen, auf die alte Methode im alten Controller, temporär auf den neuen Controller um leiten.
4. Testfälle anpassen und testen
5. Über die routes.rb alle entsprechenden Aufrufe auf den neuen Controller umleiten. Und die Testmethoden anpassen
6. Testen

[21] vgl. Trotter, Cashion, Refactoring Rails, RailsConf (2007)
[22] weitere Informationen: http://www.railsjitsu.com/restful-thinking-in-rails-1

5.2.3 Methode extrahieren

Methode extrahieren wird sehr häufig dann angewandt, wenn eine Klasse sehr viel Zeilen Code hat. Ziel ist es diese langen Codefragmente in kleinere kompaktere Methoden zu überführen. Ein weiterer Vorteil von feinkörnigen Methoden ist, dass sie selbsterklärend sind, und keine Kommentare benötigen. Jedoch ist bei vielen kleine Methoden darauf zu achten, dass die Namensgebung signifikant ist. Kleine Methoden funktionieren nur wenn sie guten Namen haben. Methoden sollten danach benannt werden, was sie tun und nicht wie sie es tun.[23].

Vorgehensweise:

1. Erstellen einer neuen Methode mit Benamung der Aufgabe
2. Kopieren des extrahierenden Codes aus der Ausgangsmethode
3. Extrahierten Code auf Referenzen und lokale Variablen durchsuchen
4. Existieren temporäre Variablen im extrahierten Code, so werden diese auch in der neuen Methode temporär
5. Neuer Methode die lokal gültigen Variablen als Parameter des extrahierten Codes übergeben
6. Ersetzen des alten Codes durch Aufruf der neuen Methode

Wie bereits erwähnt muss natürlich auch hier regelmäßig getestet werden. Geeignete Test sind nach jedem Schritt unabdingbar. Somit ist gewährleistet, dass diese Refaktorisierung erfolgreich ist.

5.3 Attribute verschieben

Ähnliche Gründe wie bei *Methode verschieben* gibt es für die Refaktorisierung *Attribute verschieben*. Auch hier ist der ausschlaggebende Gedanke der, wenn ein Attribut mehr von Methoden einer anderen Klasse verwendet wird, als die eigene. Nun gilt zu entscheiden, ob eventuell sogar die ganze Methode verschoben werden kann oder

[23] vgl. Fowler, Refactoring, S. 106 (2005)

nicht. Ist die Position der Ausgangs Methode gerechtfertigt, so wird nur das Attribut verschoben. Ansonsten ist es besser wenn die komplette Methode verschoben wird. Ein schöner Nebeneffekt von *Attribut verschieben* ist eine Performance Verbesserung. Es muss jedesmal diejenige Klasse geladen werden, aus welcher das Attribut benötigt wird, dies sind unnötige Aufrufe[24].

Um ein sicheres Refaktorisieren zu Gewähr leisten, gibt es 8 Punkte die zu beachten sind:

1. Es muss festgelegt werden, wie das neue Attribut auf die alte Klasse zu referenzieren ist. Dies wird meistens durch eine Assoziation realisiert. Um diese neue Assoziation zu erstellen, kann es notwendig werden, eine Migration zu erstellen.

2. Um das neue Attribut zu erstellen, muss eine Migration geschrieben werden, dannach werden die Daten verschoben und zum Schluss wird noch das Attribut aus der alten Klasse entfernt.

3. Um das neue Attribut zu referenzieren, wird in der alten Klasse eine *getter*- und eine *setter*-Methode erstellt.

4. Ist die geschehen, werden die Testfälle aktualisiert, sodass diese auf die neue Situation angepasst sind.

5. In diesem Schritt wird nun ausgiebig getestet.

6. Ersetzen aller Referenzen des Attributs durch geeignete Methoden der Zielklasse

7. Wenn eine Verschiebung der *getter*- und *setter-Methoden* in die neue Klasse möglich ist, werden diese verschoben, damit unnötige Datenbankaufrufe vermieden werden.

8. Nun müssen die letzten Änderungen wieder ausgiebig getestet werden

[24] vgl. Trotter, Cashion, Refactoring Rails, RailsConf (2007)

5.4 Assoziationen verschieben

Warum sollten Assoziationen verschoben werden:
Assoziationen werden verschoben, hier in diesem Zusammenhang werden Assoziationen nicht verschoben, sondern gelöscht oder hinzugefügt, wenn eine Assoziation überflüssig geworden ist, bzw. wenn eine zusätzliche Assoziation gebracht wird..[25].
Vorgehensweise:

1. Es sollte festgestellt werden, wie und wenn überhaupt man die Assoziationen der alten Klasse referenzieren kann.
2. Erstellen eines Migrationsscriptes, um die neue ID-Spalte der Assoziativen Klasse hinzu zufügen. Hier ist der Abschnitt mit *belongs_to* wichtig. Migration der alten Daten in die neue Spalte, und entferne die alte Spalte.
3. Alle *belongs_to*, *has_many* und *has_one* a Beziehungen auf die neue Assoziation ändern.
4. Erstellung von *getter*- und *setter*-Methoden für die alte Assoziation
5. Testen
6. Änderung der *getter*- und *setter*-Methoden für die neue Assoziation
7. Testen
8. Wenn möglich löschen der delegierenden Methode
9. Testen

5.5 Refaktorisierungsaufwand verringern

Eine nicht ganz umstrittene Aussage ist, dass Refaktorisieren einen zusätzlichen Aufwand darstellt. Diese Aussage kann nicht völlig wiederlegt, jedoch etwas reduziert werden. Auf folgende Aussage „Refaktorisierung ist eine überflüssige Aktivität. Ein Entwickler wird dafür bezahlt, neue Element zu entwickeln mit denen Umsatz erzielt werden kann“, gibt es folgende Antworten:

[25] vgl. Trotter, Cashion, Refactoring Rails, RailsConf (2007)

- heutzutage gibt es Werkzeuge und Techniken, mit denen schnell und effektiv refaktorisiert werden kann,
- durch Einsatz von Refaktorisierung, wird der Code schon zu einem frühen Zeitpunkt validiert, dies erspart viel Zeit um Fehler etc. mühselig zu suchen und zu eliminieren.
- die Wiederverwertbarkeit wird erhöht.

Obwohl Refaktorisierung auf den ersten Blick mühselig und als Overhead angesehen wird, so erscheint es als nützlich, wenn die Refaktorisierung in den kompletten Softwareentwicklungsprozess einbezogen wird.

5.6 Sicheres Refaktorisieren

Sicherheit ist ein schwierig zu definierendes Konzept[26]. Hierbei geht es nicht um die Sicherheit einer Software, damit diese sicher vor unbefugtem Zugriffen ist, sondern viel mehr darum, dass die vorhandene Software durch Refaktorisierung sein Verhalten nicht ändert. Eine grundlegende Definition ist, dass eine sichere Refaktorisierung ein Programm nicht kaputt macht. Ziel der Refaktorisierung ist, wie bisher beschrieben, das Programm zu restrukturieren, ohne das Verhalten zu ändern und nach der Refaktorisierung muss die Software genau das tun, was diese vor dem Refaktorisieren gemacht hat.

Es gibt verschiedene Möglichkeiten um sicher zu refaktorisieren:

- Auf die Fähigkeiten des Entwicklers vertrauen
- Vertrauen, dass der Compiler übersehene Fehler findet
- Vertrauen, dass das eingesetzte Refaktorisierungstool übersehene Fehler aufdeckt
- Vertrauen, dass Code-Reviews die letzten verbliebenen Fehler entdeckt

[26] vgl. Fowler, Refactoring, S. 407 (2005)

So einfach wie es sich in der Theorie anhört, ist sicheres Refaktorisieren in der Praxis leider nicht. Zu einem sind Entwickler auch nur Menschen, und damit fehlbar, zum anderen werden Tests von Entwicklern geschrieben und damit auch fehlbar. Es gibt Fehler die offensichtlich sind, aber auch Fehler die im Detail stecken. Ein guter Ansatz sind die Code-Reviews. Hierbei gehen mehrere Entwickler geschrieben Code durch und analysieren diesen, dabei werden viele Fehler, die dem Ersteller verborgen geblieben sind, auf gedeckt.

Um komplexere Refaktorisierungen durchzuführen, besteht die Möglichkeit Werkzeuge einzusetzten. Diese Werkzeuge helfen z.B. Referenzierungen auf Variable, die entfernt werden sollen, aufzudecken. Bei kleineren Programmen, vermag der Entwickler den Überblick zu haben, jedoch findet ein Tool Referenzen schneller und zuverlässiger. Der Entwickler muss nun entscheiden, ob die Refaktorisierung sicher ist oder nicht.

Sicherheitsansätze haben das Ziel zu garantieren, dass durch Refaktorisieren keine *neuen* Fehler in ein Programm hineinkommt. Diese Ansätze beheben keine Fehler, die schon vor der Refaktorisierung im Programm enthalten waren, sondern unterstützt und helfen solche Fehler zu entdecken und zu beheben[27].
Werden diese Punkte berücksichtigt, steht einer erfolgreichen Refaktorisierung nichts mehr im Wege.

5.7 Beteiligte Personen / Abteilungen

Wie oben schon beschrieben, ist es von großem Vorteil, die Technologie des Refaktorisierens ein zusetzten, wenn diese Technik in den gesamten Softwareentwicklungsprozess integriert ist. Von daher sind nicht nur die Entwickler in diesen Prozess zu integrieren, sondern auch die Projektleiter bis hin zur Geschäftsleitung. In den meisten Fällen lassen sich diese Personen leicht von der Notwendigkeit überzeugen, da die Qualität des Softwareproduktes mit Refaktorisieren steigern lässt. Sollte es jedoch unerwartet Widerspruch bzw. Abneigung für den Einsatz von Refaktorisierung geben, so sollten die Entwickler von sich aus entscheiden, ob die Technik eingesetzt

[27] vgl. Fowler, Refactoring, S. 411 (2005)

werden soll oder nicht. Am Endergebnis sind sich alle einig, dass das Endprodukt allen Richtlinien entsprechen soll.

6 Vor- und Nachteile

6.1 Vorteile

Refaktorisierung dient der Verbesserung der Wartbarkeit des Designs, sodass es den Entwicklern leichter fällt den vorhandenen Code funktional zu erweitern. Refaktorisierung dient auch der Verbesserung der Wiederverwendbarkeit. Durch Modularität der einzelnen Softwarefragmente, ist es leichter diese für ähnliche oder wiederkehrende Probleme zu verwenden.

Ein weiterer Aspekt der Refaktorisierung ist der, wie oben erwähnte Effekt, dass die Codestruktur wesentlich verbessert wird. Bei heutigen Softwareentwicklungen wird die Software nicht von einem einzigen Entwickler programmiert, sondern es arbeitet ein ganzes Team von Entwickler daran. Durch Einsatz dieser Technologie, werden Änderungen durch andere Entwickler vereinfacht, da diese den Sachverhalt schnell verstehen und sich besser im Code zurecht finden.

Da Refaktorisierung Tests voraussetzt, ist gewährleistet, dass der geänderte, bzw. erweiterte Code auch fehlerfrei funktioniert. Dies ist ein großer Vorteil von dynamischen Programmiersprachen wie Ruby, die auf testgetriebene Entwicklung setzten. Da die testgetriebene Entwicklung anfangs etwas mehr Zeit in Anspruch nimmt, sollte dieser Punkt eigentlich zu den Nachteilen gehören. Jedoch wenn man sich das Endergebnis anschaut, kann unter dem Strich gesagt werden, dass testgetriebene Softwareentwicklung schnell und stabiler abläuft. Und die anfänglich investierte Zeit in die Unit Tests, macht die nicht vorhandene Fehleranalyse am Ende der Fertigstellung mehr als wett.

6.2 Nachteile

Ein großer Nachteil des Refaktorisieren ist die enorme Komplexität dieser Vorgehensweise. Martin Fowler beschreibt in seinem Buch „Refactoring“ weit mehr als 60 Methoden des Refaktorisierens. Jedoch werden nur sehr selten alle Methoden angewandt.

Refaktorisieren ist riskant, da Änderungen am laufenden Code gemacht werden, die zu Fehlern führen können. Refaktorisieren ist nicht immer die beste Lösung, manchmal ist der zu überarbeitende Code, bzw das Programm in einem sehr schlechten Zustand, dass es einfacher und effektiver ist, dass ganze Programm von Grund auf neu zu entwickeln. Bei trivialen Entwicklungen, spricht der zusätzliche Aufwand gegen ein Refaktorisieren, jedoch wird diese Methode hauptsächlich bei großen Entwicklungen eingesetzt.

Die Einführung von Refaktorisieren ist nicht ganz unproblematisch, wenn Refaktorisieren benutzt wird, sollte das ganze Entwicklerteam daran teilnehmen. Es zwar von Vorteil für andere Entwickler die einen bereinigten Code bearbeiten müssen, jedoch kann es schnell demotivierend wirken, wenn ein bereinigter Code chaotisch zurück kommt.

7 Fazit und Ausblick

Refaktorisieren kann dem Entwickler helfen, den zu bearbeitenden Code besser zu verstehen. Jedoch ist Refaktorisieren nicht das *Wundermittel* für alle Probleme. Gerade zu Beginn ist es eher abschreckend, da Testen verpflichtend für Refaktorisieren ist. Dies bedeutet es wird zusätzlicher Code benötigt. Ein Stärke von *Ruby on Rails* ist die testgetriebene Entwicklung. Rails bietet dem Entwickler schon zu Beginn an Tests zu entwickeln, so dass Tests zum Refaktorisieren schon vorhanden sind.

Durch die steigende Komplexität von Software, wird in Zukunft der Gesichtspunkt des Refaktorisieren immer stärker werden. Ziel ist es, schnellst möglich eine lauffähige und funktionierende Software zu entwickeln. Dabei bleibt oft die Qualität des Sourcecodes auf der Strecke. Dadurch werden Änderungen oder Weiterentwicklungen

nur unnötig verzögert. Um dieser Entwicklung entgegen zu treten, ist Refaktorisieren genau die richtige Methode. Durch Refaktorisieren wird das Programmiertempo gesteigert und gleichzeitig verbessert sich die Qualität der Software.

Was das Refaktorisieren an geht, werden Entwickler heutzutage von IDE's tatkräftig unterstützt. Eclipse, zum Beispiel hält solch ein Tool bereit. Dieses Refactoring Plug-in ist ins RDT integriert.

Literatur

[1] Fowler, Martin (200), Refactoring, Studentenausgabe, München (Addison-Wesley)

[2] Beck, Kent (2000), Extreme Programming, München (Addison-Wesley)

[3] Wirdemann, Ralf; Baustert, Thomas (2007), Rapid Web Development mit Ruby on Rails, 2. aktualisierte und erweiterte Auflage, München (Hanser)

[4] Thomas, Dave; Hansson,Heinemeier, David (2006), Agile Web Development with Rails, 2. Auflage, Raleigh (The Progmatic Bookshelf)

[5] Gamme, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1996), Entwurfsmuster, 5. korrigierter Nachndruck, München (Addison-Wesley)

[6] Rupp, Chris (2007), Requirement-Engineering und Management, München, Wien (Hanser)

[7] Cashion, Trotter (2007), Refactoring Rails, RailsConf Europe

[8] Thomas, Dave; Hunt, Andy (2004), Der Pragmatische Programmierer, München, Wien (Hanser)

[9] rCov(15.03.2008): http://eigenclass.org/hiki.rb?rcov

[10] autotest(15.03.2008): http://nubyonrails.com/articles/autotest-rails

[11] RESTful Design (15.03.2008): http://www.railsjitsu.com/restful-thinking-in-rails-1