Aggressive Refactoring

von Thomas Kestler, elevato GmbH

Die meisten Entwickler scheuen sich davor Ihren Code oder den anderer Entwickler zu überarbeiten, doch diese Angst ist unbegründet. Refactoring ist ein notwendiger Teil der Softwareentwicklung. Man kann sogar postulieren, dass Software, die nie refactored wird, ein tote Software ist. Der folgende Artikel soll Mut zum Refactoring machen.

Niemand hat die Weisheit mit Löffeln eingenommen, daher stellt sich bei der Softwareentwicklung immer wieder die Erkenntnis, dass man in die falsche Richtung gelaufen ist. Ein unerfahrener Entwickler will sich das nicht eingestehen und programmiert um das Problem herum. Er mag sich auch einsagen, dass er jetzt keine Zeit hat, das richtig zu machen und löst das Problem per Copy/Paste, Verletzung von Sichtbarkeitsregeln (public statt private) oder andere „Hilfsmittel“. Solche Abkürzungen wären dann erlaubt, wenn man gleich ein **FIXME** in den Code einträgt und kurz danach behebt (weil dringender Releasetermin ansteht und dieser Fix schnell gemacht werden muss). Werden solche „Lösungen“ aber zur Regel, entsteht eine nicht wartbare Software (jeder kennt solche Software aus der Praxis). Halten wir also fest: Refactoring ist in der Softwareentwicklung etwas ganz normales und notwendiges.

Es gibt zwei Arten von Refactoring: das ständige Refactoring, welches ein erfahrener Entwickler ständig während der Entwicklung durchführt und das geplante Refactoring, welches der Optimierung nach einer Analyse dient (Konsolidierung). Das Refactoring kann je nach Projekt unterschiedlich gehandhabt werden. Der Idealzustand ist ein neues Projekt, bei dem das Refactoring von Anfang an gelebt und eingeplant wird. D.h. man hat im Projektplan den Feature Freeze entsprechend lange vor dem Code Freeze, sowie eine Phase für Code Review und Refactoring eingeplant. Wichtig ist es auch, dass während der Entwicklung ein Softwarearchitekt den Code laufend reviewed (am besten, indem er aktiv mitarbeitet) und so frühzeitig Fehlentwicklungen erkennen kann. Es bestehen ausreichend Testmittel (Modultests, Integrationstests, Webtests), um die Funktion der Software jederzeit (möglichst automatisiert) zu verifizieren. Wie gesagt, ein Idealzustand.

Meist muss das Refactoring jedoch erfolgen, wenn der Code schon größtenteils fertiggestellt wurde und Probleme offensichtlich wurden. Ein Beispiel dafür könnte eine Outsourcing-Entwicklung sein, bei dem der Hersteller die Software weitestgehend erstellt hat und diese nun in Betrieb und Wartung durch den Auftraggeber überführt werden soll. Oftmals wird dann bei der Analyse des Code festgestellt, dass dieser nicht den internen Anforderungen an Softwarequalität genügt und zur Gewährleistung einer langfristigen Wartbarkeit überarbeitet werden muss. Meist ist der Zeitrahmen bis zur Produktivsetzung dann sehr knapp, dass viele Projektleiter ein aggressives Refactoring meiden. Natürlich muss im Einzelfall abgewägt werden, ob man ein Refactoring noch vor Produktivsetzung oder erst danach durchführen will. Oftmals stellen sich im Betrieb auch fachliche Mängel in der Spezifikation heraus, z.B. weil die Abläufe in der Filiale vor Ort nicht genügend bekannt waren, was ja im engeren Sinne kein Refactoring ist, sondern eine Änderung der Funktionalität. Meist lässt sich das aber gleich sehr gut mit einem Refactoring verbinden.

Mit modernen Entwicklungswerkzeugen wie Eclipse ist das Refactoring handwerklich einfach und sicher zu handhaben. Man muss keine Angst mehr haben eine Stelle im Code zu vergessen, man sieht sofort die Kompilierbarkeit des gesamten Projektes und die Auswirkungen. Die Werkzeuge des Refactoring wie Extract Methode usw. (siehe Martin Fowler, Refactoring) sind leicht und konsistent durchführbar. Ein Versionskontrollsystem wie z.B. SVN oder CVS stellt ein sicheres Netz dar über dem man unbeschwert arbeiten kann.

Aggressive Refactoring bedeutet, dass man große Teile des Codes (wenn nicht alles) überarbeiten muss/will und dafür auch bereit ist für eine gewisse Zeit keinen lauffähigen Stand (im Entwicklungszweig wohlgemerkt) zu haben. Dem stehen sanftere Methoden des Refactoring gegenüber, bei denen die Lauffähigkeit und Funktion der Software im Vordergrund stehen. Das aggressive Refactoring kann im Endergebnis die Methode zur Erreichung der geforderten Qualität mit dem geringsten Aufwand sein, weil sich Entwickler andernfalls oft nicht trauen, die wirklich faulen Stellen auszuräumen (Problemumgehung anstatt Problembehebung).

Für die Durchführung eines aggressive Refactoring müssen einige Voraussetzung erfüllt werden:

- Die Probleme wurden analysiert und praxistauglich Lösungsansätze entwickelt (Proof-of-Concept nötig)
- Es besteht bereits eine praxistaugliche Systemarchitektur, die schnell umgesetzt werden kann
- Überprüfbare Ziele werden definiert
- Die Entwicklung wird angehalten (keine neuen Features für einige Zeit)
- Das Entwicklerteam wird verkleinert
- Falls Testmittel bereits bestanden, müssen diese ausreichend sein (Test Coverage), andernfalls müssen Sie jetzt, während des Refactoring erstellt werden
- Das Refactoring wird radikal (inklusive Re-Writing) durchgeführt, keine Tabus
- Qualitätssicherung, Tests und Messung der Zielerreichung werden durchgeführt

Bevor man loslegt, müssen die Probleme der bestehenden Software identifiziert und beschrieben werden. Dann müssen Lösungsvorschläge ausgearbeitet werden, welche auch wirklich umsetzbar sind. Dazu sollte man diese Lösungsvorschläge in jedem Fall zuerst exemplarisch umsetzen (Implementierungspflicht, Proof-of-Concept). Andernfalls läuft man Gefahr, dass man während des Refactoring erkennt, dass die geplante Lösung keine ist. Diese Empfehlung sollte nicht auf die leichte Schulter genommen werden.

Wenn im Rahmen des Refactoring die Architektur verändert werden soll (Schichtenmodell), dann muss diese Architektur bereits praxiserprobt sein. Das Refactoring-Projekt ist keine Spielweise zum Ausprobieren neuer Architekturen oder Komponenten. Die Entwickler müssen mit der Architektur und den verwendeten Komponenten gut vertraut sein, vor allem wenn neue Komponenten eingeführt werden (z.B: Lucene für Suchfunktion). Die neue Architektur sollte vor allem die Wartbarkeit und Erweiterbarkeit berücksichtigen (Problemanalyse, Logging, PlugIn-Konzept, etc.), ggfs. aber auch Lösungen für erkannte Probleme bieten (Performance, Skalierbarkeit, Sicherheit).

Vor Beginn des Refactoring sollten überprüfbare Ziele definiert werden. Hier einige Beispiele: Verringerung der Lines of Code um 20%, Verringerung der durchschnittlichen zyklischen Komplexität von 30 auf 10, Klare Schichtentrennung, Verringerung Compiler-Warnings (bz.w Checkstyle, etc) auf < 100, strikte Einhaltung Coding Conventions, einheitliche Fehlerbehandlung, Einsatz von Pattern (siehe unten), Reduzierung der in der Session gespeicherten Daten auf max 50KB/Thread, einheitliche Komponenten-/Technologieverwendung, Entfernung JSP-Scriptlet-Code, etc. Darüber hinaus gibt es weiche Ziele, die schwerer zu formulieren sind: der Code soll sich nach dem Refactoring besser anfühlen, griffiger und verständlicher sein. Um die nötige, spätere Akzeptanz der Ergebnisse des Refactorings beim gesamten Entwicklerteam zu erreichen, sollten diese Kriterien vor der Definition gemeinsam erarbeitet werden.

Wenn das Refactoring beginnt, können keine neuen Features umgesetzt werden. Das klingt logisch, muss aber auch kommuniziert werden. Der Zeitrahmen für das Refactoring sollte nicht mehr als wenige Wochen betragen und hart begrenzt werden: schafft man es in dieser Zeit nicht, dann ist etwas falsch gelaufen und man sollte die Aktion begraben, zunächst mit dem letzten Stand vor dem Refactoring weiter arbeiten und nochmals analysieren, wie das Refactoring besser anzugehen ist. Allenfalls hat man wenige Wochen Zeit verloren, man sollte sich nicht im Refactoring verlieren!

Das Aggressive Refactoring betrifft meist alle Teile des Code und ist sehr dynamisch, zu dynamisch für ein großes Team. Deshalb sollte das Aggressive Refactoring durch ein kleines Team aus den besten Entwicklern (inklusive Systemarchitekt) durchgeführt werden (am besten max. 3-5 Entwickler). Diese kommen erfahrungsgemäß sehr schnell voran, sofern Sie das Projekt und die fachlichen und technischen Anforderungen bereits gut kennen.

Die große Angst ist meist, dass die Software nach dem Refactoring nicht mehr funktionsfähig ist. Verifiziert werden kann dies nur durch ausreichende Tests, also müssen die Testmittel (Testspezifikation, Werkzeuge, Scripts, etc.) möglichst vor dem Refactoring vorhanden sein. Das Refactoring betrifft vor allem die Modultests, die ja mit angepasst werden und oftmals fast völlig neu geschrieben werden müssen. Wenn aber schon entsprechend gute Testfälle in den bisherigen Modultests bestanden haben, ist das meist kein Problem. Auf Ebene der Integrationstests und Webtests hat das Refactoring weniger Auswirkungen, da die Schnittstellen ja relativ stabil bleiben (müssen). Existieren nicht genügend Modultests, so müssen diese jetzt während des Refactoring erstellt werden! Integrationstests und Webtests sollten allerdings bereits vor Start des Refactoring bereitgestellt werden (sonst reicht die knappe Zeit dafür nicht aus).

Das verkleinerte Team soll die Möglichkeit erhalten, den Code radikal umzubauen. Eine gute Methode ist das Erstellen von Skeletons, also Code-Gerüsten, welche den prinzipiellen Ablauf und die Architektur abbilden und dann mit den Codefragmenten aus dem bestehenden Code gefüllt werden. Dazu erhalten die neuen Klassen eigene Namen oder kommen in eigene Packages (letzteres ist eleganter, das Ändern von Klassennamen wird von der IDE ja wieder überall nachgezogen). Sofern dies möglich ist, kann das Refactoring Team exemplarisch einen Funktionsbereich der Software umarbeiten und dies kann dann als Template zur Überarbeitung anderer Funktionsbereiche durch weitere Entwickler dienen (das kann nötig werden bei großen Projekten, sollte aber tunlichst erst dann erfolgen, wenn sichergestellt ist, dass das Refactoring eine tragfähige Vorgabe geliefert hat). Eine zu frühe Vergrößerung des Teams kann zu erheblichen Problemen führen: statt eingesparter Zeit endet man im Chaos.

Eine weitere Variante des Refactoring ist das inkrementelle Refactoring, bei dem das Refactoring in mehreren Schritten nach Themenbereichen oder Querschnittsaufgaben erfolgt: z.B. zuerst Exception-Handling und Logging überarbeiten, dann Architektur, dann Transaktionsmanagement, dann Speicherverwaltung, usw. Die Motivation hierbei liegt in der

Risikominimierung, weil man immer nur begrenzte Veränderungen durchführt. Als Nachteil ist aber zu nennen, dass die Stellen im Code mehrfach angefasst werden müssen, was in einem höherem Zeitaufwand resultiert. Wenn durch das inkrementelle Refactoring im Endergebnis ein vergleichbares Ergebnis in Bezug auf Code-Qualität erreicht wurde, kann man das inkrementelle Verfahren mit dem Aggressive Refactoring vergleichen. Gerade wenn projektfremde Entwickler das Refactoring durchführen sollen, kann das inkrementelle Refactoring das Risiko verringern.

Das Refactoring bietet eine gute Möglichkeit, die Software durch Verwendung von Design Patterns zu standardisieren. Allerdings sei davor gewarnt der Patternitis zu verfallen. Von den ca. 30 GoF Patterns sind in der Praxis oft nur 5-10 Patterns wirklich relevant. Die wichtigsten sind: Adapter, Facade, Factory, Strategy, Inversion of Control. Es gibt zu dem Thema ein gutes Buch von Kerievsky: Refactoring to Patterns (ISBN 0-321-21335-1). Als sehr tragfähig für Java Projekte hat sich das Spring Framework erwiesen, sofern die Software noch nicht darauf basiert, sollte man dessen Einsatz ernsthaft prüfen. Neben den weithin bekannten GoF Patterns sollten auch bestehende Best Practises berücksichtigt werden. Das Dumme daran ist nur, dass es so viele davon gibt. In einem Unternehmen, wie z.B. einer Bank, werden viele Projekte durchgeführt und oftmals gibt es viele und erhebliche Unterschiede zwischen diesen Projekten, obwohl man vermuten könnte, dass es gerade hier wesentlich mehr Gemeinsamkeiten geben sollte. Die Konsolidierung der Entwicklungs-methoden, die Erarbeitung der Best Practises und der Codebasis/Komponenten (durch Refactoring) könnten erhebliche Potenziale erschließen, findet aber in der Praxis meist nicht statt („dafür haben wir keine Zeit“). Oft kann man die historische Entwicklung der Projekte am Code der einzelnen Projekte ablesen, ohne dass die Erkenntnisse der neueren Projekte in ältere Projekte zurückfließen würden (Feedback). Diese älteren Projekte sind dann meist die ungeliebten Projekte, an die niemand mehr Hand anlegen will. Ein aggressives Refactoring kann aus solchen „Leichen“ wieder aktuelle, wartbare Projekte machen – und wenn man sich selbst nicht die Finger schmutzig machen will, kann man das extern vergeben.

Thomas Kestler ist Geschäftsführer der elevato GmbH und als Dozent und Sprecher auf Konferenzen aktiv. Im Januar 2009 hielt er einen Vortrag zum Thema Aggressive Refactoring auf der OOP 2009 in München. http://www.elevato.de