Refactoring

Dipl.-Inform. Ralf Reißing

Universität Stuttgart, Institut für Informatik, Abteilung Software Engineering
Breitwiesenstraße 20-22, D-70565 Stuttgart
e-mail: reissing@informatik.uni-stuttgart.de
http://www.informatik.uni-stuttgart.de/ifi/se/people/reissing.html

(erschienen in Informatik Spektrum 3/99, ©Springer-Verlag 1999)

Software, die benutzt wird, muß laufend den sich ändernden Anforderungen angepaßt werden, d.h. sie muß gewartet werden. Leider führt Software-Wartung in der Regel zur Degeneration der Struktur der Software. Das macht weitere Änderungen immer schwieriger, bis schließlich nur noch eine Neuimplementierung sinnvoll ist. Lehman [5] hat diese Entwicklung beobachtet und festgestellt, daß es sich bei diesem Phänomen um eine Art Naturgesetz handelt.

Eine Möglichkeit, dieser „natürlichen“ Degeneration der Software-Struktur entgegenzuwirken, ist die kontinuierliche Software-Restrukturierung, die dafür sorgt, daß die Software jederzeit verständlich und änderbar bleibt. Bei objektorientierter Software spricht man anstelle von Restrukturierung auch von Refactoring. Fowler [4] definiert den Begriff Refactoring folgendermaßen: „Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behavior of that software component.“

Das Refactoring umfaßt also konstruktive Maßnahmen, die dazu dienen, eine vorhandene Software oder Teile davon verständlicher und leichter änderbar zu machen, ohne dabei das beobachtbare Verhalten zu verändern. Die Verständlichkeit einer Software ist hoch, wenn ihre Struktur ihrem Verhalten angemessen ist, es also eine enge Verbindung zwischen Struktur und Funktion gibt. Dann können zu ändernde Teile leichter identifiziert und geändert werden. Eine geeignete Aufteilung der Zuständigkeiten zwischen den Komponenten der Struktur bewirkt, daß notwendige Änderungen sich nur lokal auswirken. Durch Änderungen der Funktion können allerdings die vorhandenen Strukturen ungeeignet werden, weshalb es dann angebracht ist, sie mittels Refactoring in neue, bessere Strukturen zu überführen.

Einige typische Beispiele für Refactorings sind:

- Definition einer neuen abstrakten Oberklasse zu mehreren bestehenden Klassen, um deren Gemeinsamkeiten aufzunehmen und eine einheitliche Schnittstelle zu schaffen. Die Gemeinsamkeiten können so in der Oberklasse definiert und auf die Unterklassen vererbt werden.
- Verschieben einer Klasse in und zwischen Vererbungshierarchien, um semantisch falsche oder ungeschickte Spezialisierungen zu korrigieren und Vererbung effektiver zu nutzen.
- Verschieben von Attributen und Methoden zwischen Klassen, um unnötige Parameter beim Methodenaufruf und unnötige Assoziationen zwischen Klassen zu vermeiden.
- Verbergen direkt sichtbarer Attribute und Definition geeigneter Zugriffsmethoden, um eine bessere Datenkapselung zu erreichen.
- Ersetzen eines Code-Abschnitts durch einen Methodenaufruf, um lange Methoden zu zerschlagen und duplizierte Code-Abschnitte zusammenzuführen.

- Ändern des Namens von Klassen, Attributen, Methoden und Parametern, um deren Bedeutung klarer zu dokumentieren.

Die genannten Änderungen ziehen natürlich Änderungen bei den Verwendern der betroffenen Teile nach sich, falls die Schnittstellen verändert werden. Man sieht anhand der genannten Beispiele, daß sich Refactoring auf den weiten Bereich von detaillierten Code-Strukturen, z.B. einzelnen Code-Abschnitten einer Methode, bis hin zu Entwurfsstrukturen, z.B. den Vererbungshierarchien, auswirkt. Ein umfassender Katalog von Refactorings mit Anwendungsbeispielen in Java findet sich in [4]. Beck [2] gibt ein kleineres Beispiel für Refactoring in Smalltalk.

Gerade im Bereich der evolutionären (d.h. iterativen und inkrementellen) Software-Entwicklung wird Refactoring eine hohe Bedeutung zugeschrieben. Bedingt durch die Vorgehensweise haben frühe Versionen der Software in der Regel eine schlechte Struktur, da laufend bestehende Teile geändert und neue Teile hinzugefügt werden. Wird nicht durch Refactoring gegengesteuert, entsteht schnell ein unwartbares Monstrum. Aber auch das Vorgehensmodell, bei dem erst implementiert wird, wenn das System vollständig entworfen wurde, kommt nicht ohne Refactoring aus. Die in der Entwurfsphase entwickelte Struktur wird sich in der Implementierungsphase als unvollkommen herausstellen, weil es unmöglich ist, alle Aspekte im voraus zu überschauen. Während der Implementierung erkennt man auch, welche Strukturen besser geeignet sind, die gewünschte Funktionalität zu tragen.

Neben den genannten Wirkungen auf Verständlichkeit und Änderbarkeit hat Refactoring noch andere positive Effekte. Zum Beispiel kann duplizierter Code, der durch Kopieren und Einfügen entstanden ist, durch geeignetes Refactoring wieder zusammengeführt (vgl. das Copy-and-Paste Programming AntiPattern in [3]) und damit Redundanz entfernt werden. Auch als Technik, an unbekannten Code heranzugehen, kann man Refactoring einsetzen. Man bringt die bereits verstandenen Programmteile in eine besser verständliche Struktur und kann dadurch das eigene Verständnis sofort im Code reflektieren. Interessanterweise ist es oft so, daß man während des Refactorings auf Fehler im Code aufmerksam wird.

Eine große Gefahr beim Refactoring ist es, durch die Änderungen an der Struktur neue Fehler in die Software hineinzubringen. Daher wird empfohlen, für alle betroffenen Software-Komponenten vorher umfangreiche Tests (sowohl auf Klassen- als auch auf Systemebene) bereitzustellen, die das korrekte Funktionieren der Software prüfen. Während des Refactorings werden nach jedem größeren Schritt die Tests erneut ausgeführt, um zu prüfen, ob durch Unachtsamkeit die Funktion der Software verändert wurde. Tritt beim Test ein Fehler zutage, kann seine Ursache so leichter eingegrenzt werden: Es muß an den zuletzt vorgenommenen Änderungen liegen. Dieses Vorgehen ist aber nur eine Kontrolle, die keinerlei Sicherheit bietet, daß wirklich nichts verändert wurde.

Daher hat Opdyke in seiner Dissertation [6] semantikerhaltende Refactorings (für C++) untersucht, die beweisbar das Programmverhalten nicht verändern, sofern ihre Vorbedingungen erfüllt sind. Werden nur solche semantikerhaltenden Transformationen durchgeführt, können keine neuen Fehler in die Software hineingetragen werden. Die von Opdyke angegebenen Refactorings können, da sie formal spezifiziert sind, durch ein Werkzeug auf ihre Zulässigkeit geprüft und gegebenfalls auch durchgeführt werden.

Werkzeugunterstützung beim Refactoring ist sehr wichtig, da alle Änderungen an der Struktur, die sich auf die öffentliche Schnittstelle einer Komponente auswirken, weitreichende Folgen für alle Verwender dieser Komponente haben. Diese müssen an die neue Schnittstelle angepaßt werden, was bei der Durchführung von Hand sehr mühsam werden kann. Werden die zu ändernden Stellen dagegen von einem Werkzeug identifiziert und die Anpassungen automa-

tisch vorgenommen, ist die Hemmschwelle zu einer weitreichenden Änderung, die einen positiven Effekt auf die Software-Struktur hat, viel niedriger.

Für verschiedene Smalltalk-Dialekte gibt es bereits den Refactoring Browser von Roberts und Brant ([7]; siehe auch http://st-www.cs.uiuc.edu/users/brant/Refactory/). Der Refactoring Browser ist in die normale Entwicklungsumgebung integriert und kann Opdykes Refactorings (angepaßt an Smalltalk) und andere Refactorings (unter anderem inspiriert von [1]) automatisch durchführen. Für Programmiersprachen wie C++ oder Java scheint es noch keine Refactoring-Werkzeuge zu geben, was wohl auch an der komplizierten Grammatik und Sprachsemantik liegt.

Ein großes Hindernis beim Refactoring ist allerdings kein technisches: Ein Software-Projekt-Manager muß davon überzeugt sein (oder werden), daß es sinnvoll und notwendig ist, wenn seine Entwickler keinen neuen Code schaffen, sondern an bereits vorhandenem Code Refactorings durchführen. Das wirkt zunächst sehr unproduktiv, schließlich arbeitet der vorhandene Code ja zufriedenstellend. Wozu dann Aufwand investieren, ihn zu ändern? Wie bei den meisten präventiven Maßnahmen im Software Engineering ist es auch beim Refactoring so, daß es sich um eine Investition in die Zukunft handelt, die die zu erwartenden Wartungskosten verringert und die Produktivität in der Wartung erhöht. Ob sich die Investition tatsächlich lohnt, ist schwer vorhersagbar. Die bisher dokumentierten Erfahrungen mit Refactoring in der Praxis sind allerdings recht positiv.

Literatur

1. Beck, K.: Smalltalk Best Practice Patterns. Prentice Hall 1996. Auch auf deutsch erschienen: Beck, K.: Smalltalk: Praxisnahe Gebrauchsmuster. Prentice Hall 1997
2. Beck, K.: Make it Run, Make it Right: Design Through Refactoring. Smalltalk Report, 6(4), 19-24, 1997
3. Brown, W.; Malveau, R.; McCormick, H.; Mowbray, T.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley 1998
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley 1999 (noch nicht erschienen; siehe auch http://www.awl.com/cseng/titles/0-201-89542-0/refactor/index.html)
5. Lehman, M.: Programs, Life Cycles and Laws of Software Evolution. Proceedings of the IEEE, 68(9), 1060-1076, 1980
6. Opdyke, W.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, 1992
7. Roberts, D.; Brant, J.; Johnson, R.: A Refactoring Tool for Smalltalk. Theory and Practice of Object Systems, 3, 253-263, 1997