Refactorings

Designverbesserungen in existierendem Code

Sven Sommerlade

Inhaltsverzeichnis

Literaturverzeichnis

[1] Refactoring (Improving the Design of existing Code), Martin Fowler

[2] Design Pattern (Elements of Reusable Object-Oriented Software), Gamma, Helm, Johnson, Vlissides

[3] Lifecycle and Refactoring Patterns that Support Evolution and Reuse, Brian Foote, William F. Opdyke

0.1 Was sind Refactorings?

> *Refactoring, ist das Ändern von Software, um die interne Struktur zu verbessern, ohne dass sich das Externe Verhalten ändert.*

Jede Änderung von funktionierendem Code ist sehr riskant, da diese Auswirkungen auf andere Teile der Software haben kann. Das kann zur Folge haben, dass das gesamte Programm danach nicht mehr lauffähig ist.
Deshalb muss man einige Dinge beachten, um den Code konsistent zu halten. Wenn mehrere Teile der Software neu geschrieben werden, verliert man leicht den Überblick, und übersieht einige Auswirkungen. Deshalb ist es sinnvoll immer nur kleine, einfache Änderungen vorzunehmen, um die Fehlerquellen gering zu halten. Ebenfalls ein gutes Mittel, um Fehler zu vermeiden, ist der Test. Da nach jeder kleinen Verbesserung ein gesamter Test des Programmteils nötig ist, sollte man sich vorher Testfälle konstruieren. Diese können dann automatisch ausgeführt werden. In einigen Programmiersprachen gibt es bereits entsprechende Tools, wie beispielsweise Junit für Java.
Ein weiterer wichtiger Punkt, vor allem für größere Änderungen, ist das systematische Vorgehen. Da jedes Refactoring auf empirische Erfahrungen beruht, gibt es einen "best practise"-Weg zur Durchführung. Deshalb existiert für jedes Refactoring im Katalog ein eindeutiger Ablaufmechanismus um typische Fehler zu verhindern.

0.2 Wann sollte man Refactorings anwenden?

Es stellt sich die Frage, in welchen Situationen es sinnvoll ist Refactorings anzuwenden. Beim Hinzufügen von neuer Funktionalität, kann es geschehen, dass die bestehende Objektstruktur den neuen Anforderungen nicht mehr genügt. Anstatt aber die neuen Funktionen unzureichend auf das alte Design anzupassen, sollte man eher den bestehenden Code so verändern, dass diese leicht zu integrieren sind.
Durch Refactorins ist es ebenfalls möglich, Fehler leichter zu lokalisieren. Man ändert die betroffenen Klassen und Methoden so um, dass sie kleiner und übersichtlicher werden. Darin sind Fehler einfacher zu entdecken.
Wenn man bei einem "Code Review" seine eigenen Objekte nicht mehr auf Anhieb versteht, ist dies ein Zeichen für falsche Namensgebung und/oder schlechte Strukturierung. Dies sind ebenfalls gute Gründe den Code einem Refactoring zu unterziehen.
Diese und weitere Auffälligkeiten im Code, die das Verständnis und die Wiederverwendbarkeit verschlechtern und Veränderungen erschweren, werden "Bad Smells" genannt. Einige der bekanntesten sind zum Beispiel : "Duplicate Code", "Long Method", "Large Class", usw. Für jedes dieser Bad Smells existiert eine Liste von Refactorings, die diese zu gutem Code umwandeln können.

0.3 Warum sollte man Refactorings anwenden?

Das verbesserte Software Design ist wahrscheinlich der wichtigste Punkt für Refactorings. Man behält eine konsistente Struktur bei, obwohl man immer wieder Code hinzufügt, im Gegensatz zu dem üblichen Software-Verfall, wenn neue Funktionalität hinzugefügt wird. Die Nebeneffekte davon sind, zum Einen, ein besseres Codeverständnis, wobei durch gute Namensgebung und übersichtlichen Klassen und Methoden deren Aufgaben einfacher zu begreifen sind. Zum Anderen, kann man, wie schon erwähnt, Fehler einfacher finden, wenn das Design dementsprechend verständlich ist.
Ein weiterer Punkt für Refactorings ist, dass man durch sie schneller programmieren kann. Dies ist anfangs nicht so ganz einzusehen, da man in erster Linie mehr Aufwand durch die kontinuierlichen Änderungen hat. Andererseits ist, durch das verbesserte Design, das Hinzufügen von Funktionalität wesentlich vereinfacht worden. Außerdem besitzt man ein besseres Codeverständnis, was ebenfalls die eigene Produktivität erhöht.

0.4 Wann sollte man Refactorings mit Vorsicht verwenden?

Es gibt natürlich auch Situationen, in denen Refactoring keine gute Lösung ist. In vielen Softwareprojekten werden Datenbanken eingesetzt. Sie benötigen eine festgelegte Zugriffsmöglichkeit, die nicht einfach zu verändern ist. Dies könnte einige Refactorigs schwierig bis unmöglich machen. (Dieses Problem könnte aber, beim Entwurf, durch ein zusätzliches Layer zwischen der Datenbank und der Software behoben werden)
Manchmal ist es auch nötig gewisse Interfaces für andere User zu veröffentlichen, welche aber nach einigen Refactorings hinfällig sein können. Da aber die Interfaces öffentlich sind, müssen in allen Klassen, die das Interface benutzen, die alten Methoden bestehen bleiben und umgeleitet werden. Damit wären die Änderungen des Refactorings sinnlos.
Refactoring ist ein Mittelweg zwischen Neuentwurf, und unstrukturiertem Ändern im Code. Leider entfernt sich der, durch Refactorings geänderte Entwurf mit der Zeit zunehmend vom optimalen Entwurf. Deshalb kann es bei größeren Designänderungen besser sein die Software neu zu entwerfen anstatt sie zu refactorieren.

0.5 Ziele des Refactorings

Mit Refactorings kann man also Software Design verbessern. Aber wie soll dieser verbesserte Code aussehen, oder zu welchem Ziel hin soll man refactorieren?
Das Ziel von Softwareentwicklung allgemein ist :

> *Software zu entwickeln, die heute benutzbar und morgen wiederverwendbar ist.*

Dies ist auch Ziel des Refactorings, aber es ist noch sehr vage. Wiederverwendbare Software entspricht jedoch häufig speziellen Mustern die bereits als "Design Pattern" beschrieben sind.

0.5.1 Design Pattern

Definition :

> *Design Pattern sind Beschreibungen von kommunizierenden Klassen und Objekten, die helfen ein generelles Problem in einem bestimmten Umfeld zu lösen.*

Zu Design Pattern gibt es, ähnlich den Refactorings, einen Katalog in dem viele davon aufgeführt sind. Die Beschreibung eines Patterns ist folgendermaßen aufgebaut.

Name : Name und Klassifikation des Pattern

Absicht des Patterns : Kurze Zusammenfassung, was dieses Pattern tut.

Alternative Namen : Andere bekannte Bezeichnungen für dieses Pattern, falls welche existieren.

Motivation : Szenario, da ein Designproblem zeigt, und wie dieses vom Pattern gelöst wird.

Anwendbarkeit : Situationen, in denen das Pattern angewandt werden kann.

Struktur : Grafische Darstellung der Klassen im Pattern

Teilnehmende Klassen und Objekte : Die Klassen und Objekte im Pattern, und deren Aufgaben.

Beziehungen der Klassen und Objekten : Wie die Klassen und Objekte zusammen wirken.

Konsequenzen : Welche Vor- und Nachteile hat man durch die Benutzung des Patterns.

Implementierung : Gibt es sprachspezifische Unterschiede bei der Implementierung

Beispielcode : Codefragmente, die darstellen, wie das Pattern implementiert wird.

Bekannter Einsatz des Patterns : Beispiele, wo das Pattern eingesetzt wird

Ähnliche Pattern : Pattern, die eine ähnliche Funktion haben, aber unter anderen Umständen eingesetzt werden.

0.6 Wie sind Refactorings aufgebaut?

Die Refactorings im Katalog sind nach folgender Art beschrieben :

Name : Name des Refactorings

Zusammenfassung : In welcher Situation benötigt man dieses Refactoring, und was tut es.

Motivation : Beschreibt in welcher Situation man das Refactoring anwenden soll, und in welcher nicht.

Beschreibung : Schritt für Schritt-Beschreibung wie man es ausführt

Beispiel : Einige einfache Beispiele wie es benutzt wird.

0.7 Einfaches Beispiel : Extract Method

Name : Extract Method

Zusammenfassung : Dieses Refactoring faßt Code-Fragmente zusammen

Motivation : Wird eingesetzt um zu lange Methoden aufzuteilen, oder unpassende Methodennamen zu verhindern.

Beschreibung : Zuerst erzeugt man eine neue Methode mit sinnvollem Namen, und kopiert den Code hinein. Danach müssen die Methoden angepasst werden, indem die lokalen Variablen in Parameter umgewandelt werden.Außerdem muß überprüft werden, ob temporäre Variablen nur noch im extrahierten Code stehen, welche dann dort als temporäre Variablen deklariert werden müssen. Wenn eine lokale Variable der Quellmethode verändert wird, wird diese Rčkgabewert der Methode (bei mehreren anderes Pattern verwenden).Nun durchsucht man die Zielmethode nach lokalen Variablen, die jetzt Parameter sind.
Wenn alle Variablen geändert wurden, kann man kompilieren. Hat dies funktioniert, ersetzt man den extrahierten Code durch einen Funktionsaufruf der neuen Methode, kompiliert erneut und testet.

Beispiel : Ein einfaches Codebeispiel

Vorher :

```
void printOwing(double previousAmount) {
  Enumeration e = _orders.elements();
  double outstanding = previousAmount * 1.2 );
  print Banner();
//calculate outstanding
  while(e.hasMoreElements()) {
    Order each = (Order) e.nextElement();
    outstanding += each.getAmount();
  }
  printDetails(outstanding);
}
```

Nachher :

```
void printOwing(double previousAmount) {
  double outstanding = getOutstanding(previousAmount * 1.2);
  print Banner();
  printDetails(outstanding);
}
double getOutstanding(double initialValue) {
  double result = initialValue;
  Enumeration e = _orders.elements;
  while ... {
    result += each.getAmount();
    \dots
  }
  return result;
}
```

0.8 Was existiert schon?

Am Anfang der Entwicklung wurden die Refactorings, wenn möglich direkt, in den Refactoring Browser von Smalltalk integriert. Dieser Browser ist ein Tool, welches einige Refactorings automatisch ausführen kann, so daß die Tests nach jedem Schritt entfallen können.
Mittlerweile gibt es auch andere Entwicklungsumgebungen, die automatisches Refactoring anbieten (Beispielsweise IntelliJ für Java).
Dies ist ein Zeichen dafür, daß Refactorings in Zukunft für Frameworks genauso wichtig werden könnten, wie zum Beispiel Versionskontrolle (CVS), was jede größere Umgebung bereits bereitstellt.

0.9 Kritik und Ausblick

Zusammenfassend sind mir einige Dinge aufgefallen, die beim Refactoring problematisch werden könnten.
Die Fehlersuche mit Refactorings ist sehr gefährlich, da man Änderungen in fehlerhaften Code macht, und somit diese Fehler weiter verbreiten und verteilen kann.
Als nächsten Punkt möchte ich die Ziele des Refactorings ansprechen. Trotz der Möglichkeit "Design Pattern" als Vorlage zu nehmen, ist das Ziel immer noch recht unklar. Man weiß vorher nicht, nach welcher Art von Änderung der Code verständlicher ist. Dieses Problem kann aber durch Tools für automatisches Refactoring behoben werden. Es ist damit möglich, einige Änderungen zu testen und diese wieder Rückgängig zu machen, bis das Design ansprechend genug ist.
Ein weiterer Punkt ist, daß die erstellten Testfälle nicht zwangsmäßig zu dem neuen Code passen. Dies ist ein großer Nachteil beim automatischen Testen, sowie für Refactoring insgesamt.
Abschließend ist noch zu bemerken, daß alle Refactorings auf objektorientierte Sprachen bezogen sind. Einige davon sind aber ebenso nützlich für funktionale Sprachen, oder ältere Konzepte. Andererseits können nur wenige davon auf neuere Ansätze, wie beispielsweise aspektorientierte Programmierung, angewandt werden.
Meiner Meinung nach, wäre es deshalb nötig, die Darstellung im Katalog um einen Punkt für die Sprachklasse zu erweitern. Außerdem müßten dann für neue Sprachklassen eigene Refactorings gefunden werden. Dies ist aber noch nicht möglich, da keine genügend große CVS-Archive, zur Untersuchung, für diese Sprachen existieren.