Programmierkonzepte in der Physikalischen Chemie

20. Refactoring

Albert-Ludwigs-Universität Freiburg

Dr. Till Biskup

Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2018/19

Zentrale Aspekte

- Software wird und muss sich immer wieder verändern, da sich die Anforderungen ändern.
- Refactoring ist die Verbesserung der internen Struktur existierenden Codes ohne Änderung seiner Funktionalität.
- Zeitdruck ist kein Argument gegen Refactoring. Unterlassen verursacht später (erheblich) mehr Aufwand.
- Refactoring sollte langsam, wohlüberlegt und sorgfältig durchgeführt werden.
- Gute Tests und eine ausreichende Testabdeckung sind zwingende Voraussetzung für erfolgreiches Refactoring.

Übersicht

Motivation und Voraussetzungen

Warum und wann sollte man refaktorieren?

Strategien und Vorgehen beim Refactoring

Zusammenhang mit Entwurfsmustern

Ausgangspunkt: Code verändert sich

Die Wahl der Metapher kann entscheidenden Einfluss haben.

UNI FREIBURG

> *As a program evolves, it will become necessary to rethink earlier decisions and rework portions of the code. This process is perfectly natural. Code needs to evolve; it's not a static thing. [...]*
>
> *Rather than construction, software is more like* gardening—*it is more organic than concrete.*
>
> – Hunt und Thomas

- Die Wahl der Metapher ist mitunter entscheidend.
  - Eine valide Detailplanung vorab ist praktisch unmöglich.
  - Programmierung ist zu komplex, um überschaubar zu sein.
  - Programmentwicklung ist ein evolutionärer Prozess – ähnlich wie das Verfassen eines längeren Textes.

Hunt und Thomas: The Pragmatic Programmer, Addison-Wesley, Boston 1999, S. 184

Gründe für Veränderungen am Code

Veränderte Anforderungen und erweiterte Kenntnis

UNI FREIBURG

- Änderungen in den Anforderungen an den Code
  - Anforderungen ändern sich über die Zeit.
  - Komplexität: Anforderungen nicht komplett vorhersehbar
- Erweiterungen
  - Ziel: Wiederverwendbarkeit, auch in anderem Kontext
  - Die meisten Programme fangen klein an.
- vertieftes Verständnis der Problemstellung
  - wirkliches Verständnis oft erst durch die Umsetzung
  - Der Teufel steckt häufig im Detail.
- Zunahme der eigenen Kenntnisse und Fähigkeiten
  - Kenntnis von Konzepten
  - praktische Erfahrung in der Umsetzung

Definition

Veränderung der Struktur ohne Einfluss auf die Funktion

UNI FREIBURG

> *Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*
> *It is a disciplined way to clean up code*
> *that minimizes the chances of introducing bugs.*
> *In essence when you refactor you are improving the design of the code after it has been written.*

– Martin Fowler

- Refactoring verfolgt ein zweifaches Ziel:
  - Verständnis: Verbesserung der Lesbarkeit von Code
  - Flexibilität: Erleichterung der Veränderung von Code
- keine Veränderung des beobachtbaren Verhaltens

Fowler: Refactoring, Addison-Wesley, Boston 1999, S. xvi

Wortherkunft

Zusammenhang mit Modularisierung („Faktorisierung“)

UNI FREIBURG

> *The word „refactoring“ in modern programming grew out of Larry Constantine’s original use of the word „factoring“ in structured programming, which referred to decomposing a program into its constituent parts as much as possible [...].*
>
> – Steve McConnell

- Modularisierung
  - Strategie zur intellektuellen Beherrschbarkeit von Code
  - erhöht gleichzeitig Lesbarkeit und Wiederverwendbarkeit
- Refactoring: Wiedererlangung der Modularität
  - Lesbarkeit und Modularität sind Ergebnis eines aktiven, bewussten Prozesses.

McConnell: Code Complete, Microsoft Press, Redmond 2004, S. 565

Motivation für Refactoring

Funktionierender Code ist nicht gut genug.

UNI FREIBURG

> *[Refactoring] is about the difference between getting something to work and getting something right. It is about the value we place in the structure of our code.*
>
> – Robert C. Martin

- Funktionierender Code ist nicht gut genug.
  - Lesbarkeit ist entscheidend für Veränderungen.
  - Gutes Design ist Ergebnis eines evolutionären Prozesses.
  - Ohne ständiges Refactoring nimmt die Codequalität ab.
- Refactoring zahlt sich aus.
  - Codequalität beeinflusst die Entwicklungsgeschwindigkeit.
  - Kleine, kontinuierliche Schritte sind realisierbar.

Martin: Agile Software Development, Prentice Hall, Upper Saddle River 2003, S. 31

Code hat drei Aufgaben

Funktionalität ist nur eine davon.

UNI FREIBURG

> *What does it take to make a module easy to read and easy to change? [...] It takes attention. It takes discipline. It takes a passion for creating beauty.*
>
> – Robert C. Martin

- Funktionalität
  - Daseinsberechtigung des jeweiligen Codeabschnittes
- Veränderbarkeit
  - Verantwortung des Entwicklers: Veränderung so einfach wie möglich zu machen.
- Kommunikation
  - Code wird viel häufiger gelesen als geschrieben.

Martin: Agile Software Development, Prentice Hall, Upper Saddle River 2003, S. 32

Voraussetzungen für erfolgreiches Refactoring

Funktionierender Code, Tests, Versionsverwaltung

UNI FREIBURG

> *Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is.*
> *With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse.*
>
> – Michael C. Feathers

- funktionierender Code
- automatische Tests (mit hoher Abdeckung)
- Versionsverwaltungssystem (als Sicherheitsnetz)

Feathers: Working Effectively with Legacy Code, Prentice Hall, Upper Saddle River 2005, S. xvi

Warum sollte man refaktorieren?

Um Code lesbarer und einfacher veränderbar zu machen

UNI FREIBURG

Ein paar Gründe für Refactoring

- Doppelungen im Code
  - häufige Fehlerquelle, insbesondere wenn weit verteilt
  - erschweren die Weiterentwicklung und Pflege
- nichtorthogonales Design
  - (ungewollte) Abhängigkeiten zwischen Funktionen
  - erschwert Wiederverwendbarkeit und Weiterentwicklung
- veraltetes Wissen
  - Dinge verändern sich, Anforderungen entwickeln sich.
  - Das eigene Wissen ob der Problemstellung wächst.

☛ wann immer Code mühsam zu lesen oder zu verändern ist („*Code Smells*“) oder man Ideen für Verbesserungen hat

Wann sollte man refaktorieren?

Die kurze Antwort: immer

> *...refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts. You don't decide to refactor, you refactor because you want to do something else, and refactoring helps you to do that other thing.*
>
> – Martin Fowler

- Hinzufügen neuer Funktionalität
- Beheben eines Fehlers (Bugs)
- Code-Review
- Dritter Schritt der testgetriebenen Entwicklung (TDD)

☛ Refactoring ist Mittel zum Zweck, um Code zu verändern.

Fowler: Refactoring, Addison-Wesley, Boston 1999, S. 58

Schreibe einen fehlschlagenden Unittest

1

TDD

2

Bringe den Test dazu durchzulaufen

3

Refaktoriere

verändert nach: Freeman und Pryce: Growing Object-Oriented Software, Guided by Tests, Addison Wesley, Upper Saddle River 2010, S. 6

Wann sollte man *nicht* refaktorieren?

Refactoring ist kein Allheilmittel – und kann missbraucht werden

UNI FREIBURG

- Refactoring statt kompletter Neuentwurf
  - immer vom konkreten Einzelfall abhängig
  - Kosten-Nutzen-Abwägung: Refactoring kann Zeit kosten.
  - Neuentwurf nur bei Strategien für besseren Code
- „Refactoring“ als Tarnung für Versuch und Irrtum
  - Refactoring ist eine Veränderung *funktionierenden* Codes.
  - Wildes Herumhacken qualifiziert nicht als Refactoring.
- Refactoring ohne ausreichende Tests
  - Refactoring ändert die Struktur unter Erhalt der Funktion.
  - Tests sind eine Voraussetzung für Refactoring.
- Änderung veröffentlichter Schnittstellen
  - Problem: Auswirkungen nicht mehr lokal kontrollierbar.
  - mögliche Lösung: temporär parallel zwei Schnittstellen

Grundsätzliches Vorgehen

Systematisch: langsam, überlegt, vorsichtig

UNI FREIBURG

> *To avoid digging your own grave,*
> *refactoring must be done systematically.*
>
> – Erich Gamma

- nicht refaktorieren *und* neue Funktionalität hinzufügen
  - Neue Funktionalität ist häufig ein Auslöser für Refactoring.
  - Code sollte *zuerst* und *nur* refaktoriert werden.
- Gute Tests sind eine Voraussetzung für Refactoring.
  - Ziel ist der Erhalt der Funktionalität.
  - Nur (automatisierte) Tests stellen Funktionalität sicher.
- kleine, wohlüberlegte Schritte
  - nach jedem kleinen Schritt alle Tests laufen lassen
  - Funktionalität sollte bei jedem Schritt erhalten bleiben.

Vorwort in: Fowler: Refactoring, Addison-Wesley, Boston 1999

Unterschiedliche Ebenen von Code

Refactoring kann unterschiedliche Ebenen von Code betreffen

- Daten
  - Bsp.: Variablen und deren Namen
- Statements
  - Bsp.: (komplexe) Boolesche Ausdrücke
- Routinen
  - Bsp.: Aufteilen langer Funktionen/Methoden
- Implementierung von Klassen
  - Bsp.: Verschieben von Methoden innerhalb der Vererbung
- Schnittstellen von Klassen
  - Bsp.: Verschieben von Methoden zwischen Klassen
- komplette Systeme
  - Bsp.: Austausch von Fehlercodes und Ausnahmen

Konkrete Refactoring-Schritte

Eine Reihe einfacher, anschaulicher Beispiele

UNI FREIBURG

- Umbenennen von Variablen, Funktionen, Klassen
  - Ziel: bessere Lesbarkeit, höhere Ausdrucksstärke
  - wichtig: alle Vorkommen entsprechend umbenennen
  - Durch Suchen&Ersetzen-Funktion (oft) einfach realisierbar.
- Aufteilen komplexer Statements auf mehrere Zeilen
  - Ziel: bessere Lesbarkeit, kürzere Zeilen
  - Beispiel: komplexe Bedingungen in `if`-Abfragen
- Aufteilen einer Funktion/Methode
  - Ziel: bessere Lesbarkeit, Modularität, DRY Code
  - Eine Funktion sollte immer nur eine Aufgabe erfüllen.

☛ Es gibt ganze Kataloge hilfreicher Refactoring-Strategien.

Details in: Fowler: Refactoring, Addison-Wesley, Boston 1999

Konkrete Refactoring-Schritte

Vorteile einer guten IDE

- Umbenennen von Variablen, Funktionen, Klassen
  - manuelles Suchen&Ersetzen mühsam und fehleranfällig
  - Gute IDEs kennen den Kontext.
  - Umbenennen überall (inkl. Tests) einfach möglich
- Aufteilen einer Funktion/Methode
  - Codeblock braucht oft temporäre Variablen
  - können automatisch als Parameter übergeben werden
  - neue Funktion/Methode *sprechend* benennen!
- Verschieben von Methoden zwischen Klassen
  - sowohl horizontal als auch vertikal
  - innerhalb der Vererbungshierarchie automatisierbar

☛ Gute moderne IDEs zeichnen sich durch durchdachte Unterstützung diverser Refactoringschritte aus.

Refactoring als kontinuierlicher Prozess

Kleine Schritte sind machbar und zahlen sich aus.

UNI FREIBURG

> *I strongly recommend that you* always *practice such refactoring for* every *module you write and for* every *module you maintain. The time investment is very small compared to the effort you'll be saving yourself and others in the near future.*
>
> – Robert C. Martin

- Refactoring ist *machbar* und skaliert.
  - Pfadfinderregel: nur ein wenig besser machen, nicht perfekt
  - Die Zeit für das „große Aufräumen" fehlt meist/immer.
  - Refactoring ist oft einfach und behebt die Ursachen.
- Refactoring erspart Zeit – sehr bald und allen.
  - erhält/ermöglicht Erweiter- und Veränderbarkeit des Codes

Martin: Agile Software Development, Prentice Hall, Upper Saddle River 2003, S. 42

Zusammenhang mit Entwurfsmustern

Entwurfsmuster sind das Ziel, Refactoring der Weg dorthin.

UNI FREIBURG

> *There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else.*
>
> – Martin Fowler

Die Motivation hinter beiden ist identisch:

- Code-Doppelungen reduzieren oder entfernen
- komplizierten Code vereinfachen
- Absicht besser im Code selbst ausdrücken

Fowler: Refactoring, Addison-Wesley, Boston 1999, S. 107

Anforderungen ändern sich

Refactoring bewahrt vor zu wenig und zu viel Komplexität

- Entwurfsmuster sind ein zweischneidiges Schwert.
  - Zu viel und zu früh verkompliziert unnötig den Code.
  - Zu wenig macht Code unübersichtlich und schwerfällig.
- Refactoring sorgt für Flexibilität
  - Voraussetzung für den gewinnbringenden Einsatz: Kenntnis der Muster und Refactoring-Strategien
  - Muster müssen nicht vorsorglich verwendet werden.
  - Prognosen sind schwierig... besonders in die Zukunft.
- drei Richtungen für Refactoring
  - auf Entwurfsmuster zu
  - zu Entwurfsmustern
  - weg von Entwurfsmustern

☛ Meist eine Serie „atomarer“ Refactoringschritte

Details in: Kerievsky: Refactoring to Patterns, Addison-Wesley, Boston 2005

Zentrale Aspekte

- Software wird und muss sich immer wieder verändern, da sich die Anforderungen ändern.
- Refactoring ist die Verbesserung der internen Struktur existierenden Codes ohne Änderung seiner Funktionalität.
- Zeitdruck ist kein Argument gegen Refactoring. Unterlassen verursacht später (erheblich) mehr Aufwand.
- Refactoring sollte langsam, wohlüberlegt und sorgfältig durchgeführt werden.
- Gute Tests und eine ausreichende Testabdeckung sind zwingende Voraussetzung für erfolgreiches Refactoring.