

Programmierkonzepte in der Physikalischen Chemie

20. Refactoring

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Dr. Till Biskup

Institut für Physikalische Chemie
Albert-Ludwigs-Universität Freiburg
Wintersemester 2016/17



Zentrale Aspekte



- ❏ Software wird und muss sich immer wieder verändern, da sich die Anforderungen ändern.
- ❏ Die Codequalität sollte sich über die Änderungen hinweg verbessern und nicht verschlechtern.
- ❏ 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.

Motivation und Voraussetzungen

Warum und wann sollte man refaktorisieren?

Strategien und Vorgehen beim Refactoring

Zusammenhang mit Entwurfsmustern

“ *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 entscheidend.
 - Eine valide Detailplanung vorab ist praktisch unmöglich.
 - Programmierung ist zu komplex, um überschaubar zu sein.
 - Programmentwicklung ist ein evolutionärer Prozess.

- ▶ 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

“ *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

- ▶ Ziel: Vereinfachung von Verständnis und Veränderung
- ▶ keine Veränderung des beobachtbaren Verhaltens

“ *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.

“ *[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.

“ *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.

“ *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)

Warum sollte man refaktorisieren?

Um Code lesbarer und einfacher veränderbar zu machen

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 die Wiederverwendbarkeit
- ▶ veraltetes Wissen
 - Dinge verändern sich, Anforderungen entwickeln sich.
 - Das eigene Wissen ob der Problemstellung wächst.
- ☞ wann immer Code mühsam zu lesen ist oder man Ideen hat, wie er verbessert werden könnte

Wann sollte man refaktorisieren?

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
- 👉 Refactoring ist Mittel zum Zweck, um Code zu verändern.

Wann sollte man *nicht* refaktorisieren?

Refactoring ist kein Allheilmittel – und kann missbraucht werden

- ▶ 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 *funktionierender* 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

“ *To avoid digging your own grave, refactoring must be done systematically.*

– Erich Gamma

- ▶ nicht refaktorisieren 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.

- ▶ Daten
 - Bsp.: Variablen und deren Namen
- ▶ Statements
 - Bsp.: (komplexe) Boolesche Ausdrücke
- ▶ Routinen
 - Bsp.: Aufteilen langer Funktionen
- ▶ 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

- ▶ Umbenennen von Variablen, Funktionen, Klassen
 - Ziel: bessere Lesbarkeit, höhere Ausdrucksstärke
 - wichtig: alle Vorkommen entsprechend umbenennen
 - Durch Suchen& Ersetzen-Funktion einfach realisierbar.

- ▶ Aufteilen komplexer Statements auf mehrere Zeilen
 - Ziel: bessere Lesbarkeit, kürzere Zeilen
 - Beispiel: komplexe Bedingungen in `if`-Abfragen

- ▶ Aufteilen einer Routine/Methode
 - Ziel: bessere Lesbarkeit, Modularität, DRY Code
 - Eine Routine sollte immer nur eine Aufgabe erfüllen.

- ☞ Es gibt ganze Kataloge hilfreicher Refactoring-Strategien.

“ *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

- ▶ Entwurfsmuster sind ein zweiseitiges 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



- ❏ Software wird und muss sich immer wieder verändern, da sich die Anforderungen ändern.
- ❏ Die Codequalität sollte sich über die Änderungen hinweg verbessern und nicht verschlechtern.
- ❏ 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.