

Wissenschaftliche Softwareentwicklung

Bausteine einer rechnergestützten Datenauswertung,
die den Kriterien der Wissenschaftlichkeit genügt

18. Refactoring

Till Biskup

Physikalische Chemie

Universität Rostock

12.12.2024





- ❏ 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 ist eine Frage der Professionalität. Es sollte langsam, wohlüberlegt und sorgfältig umgesetzt werden.
- ❏ Gute Tests und eine ausreichende Testabdeckung sind zwingende Voraussetzung für erfolgreiches Refactoring.
- ❏ Gute IDEs unterstützen auch komplexere Refactoring-Schritte und steigern so die Produktivität und Codequalität.

Motivation und Voraussetzungen

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 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.

- ▶ 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

- ▶ Refactoring verfolgt ein zweifaches Ziel:
 - Verständnis: Verbesserung der Lesbarkeit von Code
 - Flexibilität: Erleichterung der Veränderung von Code
- ▶ 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.

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

Referenz: Yourdon und Constantine: Structured Design, Prentice Hall, Englewood Cliffs 1979

“ *[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.
⇒ Softwareentropie
- ▶ 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)

Motivation und Voraussetzungen

Wann sollte man refaktorisieren?

Strategien und Vorgehen beim Refactoring

Zusammenhang mit Entwurfsmustern

Wann sollte man refaktorisieren?

Code Smells als Kriterien für Refactoring



“ *If it stinks, change it.*

– Grandma Beck

Ein paar Beispiele für *Code Smells*

- ▶ Doppelungen im Code
 - ▶ nichtorthogonales Design
 - ▶ zu lange Funktionen/Methoden
 - ▶ nicht aussagekräftige Namen
 - ▶ Fallunterscheidungen statt Polymorphie
 - ▶ Kommentare
- ☞ wann immer Code mühsam zu lesen oder zu verändern ist oder man Ideen für Verbesserungen hat

zitiert in: Fowler: Refactoring, Addison-Wesley, Boston 1999, S. 75

Wann sollte man refaktorisieren?

Die kurze Antwort: immer



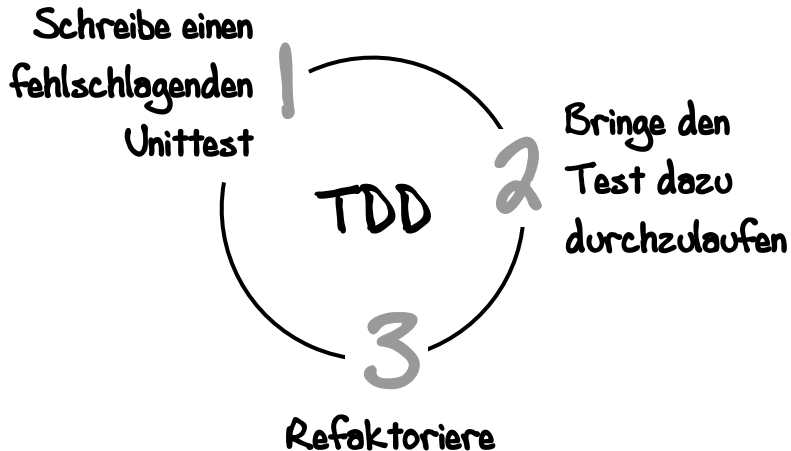
“ *...refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts. You don't decide to refactor, you refactor because you want to do something else, and refactoring helps you to do that other thing.*

– Martin Fowler

- ▶ Hinzufügen neuer Funktionalität
- ▶ Beheben eines Fehlers (Bugs)
- ▶ Code-Review
- ▶ Dritter Schritt der testgetriebenen Entwicklung (TDD)
- 👉 Refactoring ist Mittel zum Zweck, um Code zu verändern.

Erinnerung: Testgetriebene Entwicklung (TDD)

Der Zyklus testgetriebener Entwicklung



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

Wann sollte man *nicht* refaktorisieren?

Refactoring ist kein Allheilmittel – und kann missbraucht werden



- ▶ „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
- ▶ Refactoring statt kompletter Neuentwurf
 - immer vom konkreten Einzelfall abhängig
 - Kosten-Nutzen-Abwägung: Refactoring kann Zeit kosten.
 - Neuentwurf nur bei Strategien für besseren Code

Motivation und Voraussetzungen

Wann sollte man refaktorisieren?

Strategien und Vorgehen beim Refactoring

Zusammenhang mit Entwurfsmustern

“ *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.
 - Refactoring ist ein *eigener* Commit in der Versionsverwaltung.
- ▶ 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.

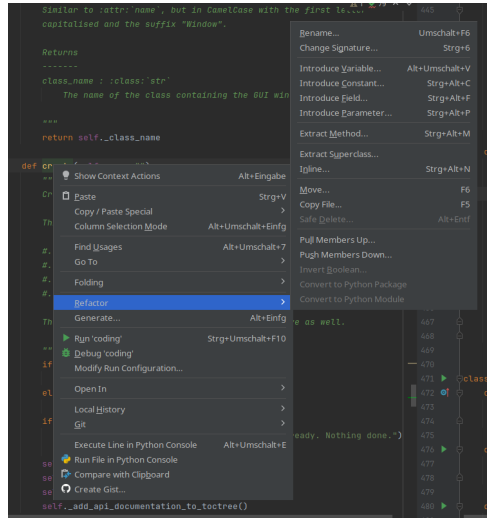
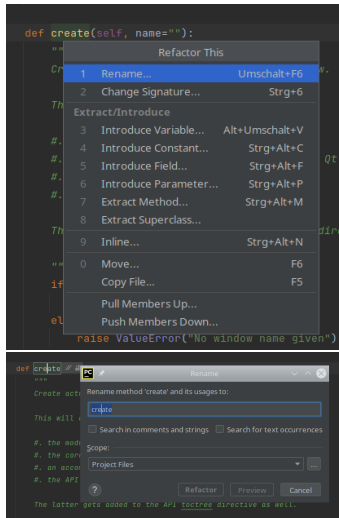
- ▶ Daten
 - Bsp.: Variablen und deren Namen
- ▶ Statements
 - Bsp.: (komplexe) Boolesche Ausdrücke
- ▶ Routinen
 - Bsp.: Aufteilen langer Funktionen/Methoden
- ▶ Implementierung von Klassen
 - Bsp.: Verschieben von Methoden innerhalb der Vererbung
- ▶ Schnittstellen von Klassen
 - Bsp.: Verschieben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

- ▶ Umbenennen von Variablen, Funktionen, Klassen
 - Ziel: bessere Lesbarkeit, höhere Ausdrucksstärke
 - wichtig: alle Vorkommen entsprechend umbenennen
 - Durch Suchen&Ersetzen-Funktion ggf. realisierbar.
- ▶ Aufteilen komplexer Statements auf mehrere Zeilen
 - Ziel: bessere Lesbarkeit, kürzere Zeilen
 - Beispiel: komplexe Bedingungen in `if`-Abfragen
- ▶ Auslagern komplexer Bedingungen
 - eigene Funktion/Methode mit *sprechendem* Namen
- ▶ 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.

- ▶ 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
 - zumindest innerhalb der Vererbungshierarchie automatisierbar
- ☞ Gute moderne IDEs zeichnen sich durch durchdachte Unterstützung diverser Refactoringschritte aus.

Refactoring-Unterstützung in einer IDE

Beispiel: PyCharm



“ *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

Motivation und Voraussetzungen

Wann sollte man refaktorisieren?

Strategien und Vorgehen beim Refactoring

Zusammenhang mit Entwurfsmustern

“ *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 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



- ❏ 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 ist eine Frage der Professionalität. Es sollte langsam, wohlüberlegt und sorgfältig umgesetzt werden.
- ❏ Gute Tests und eine ausreichende Testabdeckung sind zwingende Voraussetzung für erfolgreiches Refactoring.
- ❏ Gute IDEs unterstützen auch komplexere Refactoring-Schritte und steigern so die Produktivität und Codequalität.