

Refactorings in großen Softwareprojekten



(C) 2004

Martin Lippert

lippert@acm.org

Stefan Roock

stefan@stefanroock.de

Was ist Refactoring?



⌘ „A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior“

[Fowler 99]

Kleine Refactorings



- ⌘ Die meisten Fowler-Refactorings bewegen sich auf einer elementaren Ebene:
 - ☑ Extract Method, Introduce Parameter, etc.
- ⌘ Gehören heute zum Handwerkszeug eines jeden Entwicklers
- ⌘ Viele sind automatisiert durch IDEs
 - ☑ Siehe z. B. Eclipse, IDEA, ...

Kleine Refactorings sind nicht alles - leider



⌘ In größeren Projekten treten in der Regel immer wieder größere und deutlich schwierigere Refactorings auf:

- ☒ Refactorings, die zentrale oder große Teile des Systems verändern
- ☒ Refactorings, die eine veröffentlichte Schnittstelle zu anderen Systemteilen betreffen
- ☒ Refactorings, die sich bis auf die Datenbank auswirken

Unser Fokus für heute



⌘ In größeren Projekten treten in der Regel immer wieder größere und schwierigere Refactorings auf:

Refactorings, die zentrale oder große Teile des Systems verändern

- ☒ Refactorings, die eine veröffentlichte Schnittstelle zu anderen Systemteilen betreffen
- ☒ Refactorings, die sich bis auf die Datenbank auswirken

Große Refactorings: Charakterisierung



- ⌘ dauern länger als ein Tag
- ⌘ führen zu Änderungen an vielen Systemteilen
- ⌘ betreffen mehr als einen Entwickler / ein Pair
- ⌘ großes Refactoring muss zerlegt werden
- ⌘ die Folgen der Einzelschritte lassen sich nur schwer absehen
- ⌘ großes Refactoring muss explizit geplant werden
- ⌘ Zwischenschritte müssen integriert werden
- ⌘ es muss schlechter werden, bevor es besser werden kann (Umleitungen)

Große Refactorings: Ein Schritt zurück, zwei vor :-)



Große Refactorings: Ursachen und Beispiele

⌘ Typhierarchien ändern sich

- ☒ Wenn sich eine zentrale Typhierarchie verändert, hat das teilweise schwerwiegende Auswirkungen auf große Teile des Systems

⌘ Architektur-Smells:

- ☒ die zu ändernde Entwurfsentscheidung wurde an mehr als einer Stelle ausgedrückt (Verletzung des DRY-Prinzips)
- ☒ Besonders interessant auf Ebene von Benutzungs- und Vererbungsbeziehungen zwischen Klassen, Packages, Subsystemen und Schichten

Große Refactorings: Probleme



- ⌘ man läuft schnell in Sackgassen
- ⌘ wegen Interferenzen mit restlicher Entwicklung kann man nicht einfach so zurück
- ⌘ man verliert schnell den Überblick
- ⌘ die Planung ist sehr schwierig
- ⌘ unter Projektdruck neigen Refactorings zum Versanden
 - ⏏ auf halbem Wege abgebrochene Refactorings verschlechtern die Systemstruktur statt sie zu verbessern

Wie können wir damit umgehen?



⌘ Regeln und Best Practices:

- ☑ Wir geben Hinweise im Umgang mit großen Refactorings, die sich in Projekten bewährt haben

⌘ Fragmente:

- ☑ Typische wiederkehrende Mechanismen zur Beseitigung von Architektur-Smells

Wie können wir damit umgehen?



⌘ Regeln und Best Practices:

- ☑ Wir geben Hinweise im Umgang mit großen Refactorings, die sich in Projekten bewährt haben

⌘ Fragmente:

- ☑ Typische wiederkehrende Mechanismen zur Beseitigung von Architektur-Smells

Regeln und Best Practices



⌘ Refactorings explizit in den Planungsprozess einbeziehen

- ☑ Refactoring-Budget pro Iteration
- ☑ Refactoring-Iterationen bei Bedarf
- ☑ Regelmäßige Refactoring-Iterationen

Regeln und Best Practices



⌘ Refactoring-Planungs-Session

- ☑ Größere Refactorings mit dem gesamten Team diskutieren und planen

Regeln und Best Practices



⌘ Refactoring-Pläne erstellen

- ☑ Refactoring-Route aufschreiben
- ☑ Refactoring-Plan als Tracking-Instrument nutzen

Regeln und Best Practices



⌘ Umleitungen

- ☒ Um ein Refactoring in kleine Schritte zu zerlegen, müssen häufig Umleitungen in den Code eingebaut werden
- ☒ So kann ein Refactoring schrittweise durchgeführt werden und das System ist trotzdem immer lauffähig
- ☒ Umleitungen müssen aber als solche gekennzeichnet werden

Regeln und Best Practices



⌘ Safe-Points

- ☑ Refactoring in kleine Schritte zerlegen
- ☑ Nicht nach jedem kleinen Schritt wird die Struktur des Systems besser (Umleitungen)
- ☑ Safe-Point definieren: nach welchen Schritten hat System einen verbesserten Stand erreicht, aber noch nicht das endgültige Design

Regeln und Best Practices



⌘ Branches und Safe-Points

- ☒ Branches nicht für das komplette Refactoring (Merge-Aufwände würden zu groß werden)
- ☒ Stattdessen Branches jeweils bis zu einem definieren Safe-Point durchführen und dann mergen

Regeln und Best Practices

⌘ To-Do-Listen

- ☑ Wir nutzen die Fehler und Warnungen des Compilers als To-Do-Listen, um ein Refactoring abzuarbeiten
- ☑ Aber:
 - ☑ Es dürfen pro Schritt nur konstant viele Compile-Fehler auftreten
 - ☑ Die Warnungen müssen in einer beliebigen Reihenfolge abgearbeitet werden können

Regeln und Best Practices



⌘ Automatisierte Akzeptanztests

- ☑ Für kleine Refactorings dienen uns häufig Unit-Tests als Sicherheitsnetz
- ☑ Bei großen Refactorings müssen häufig auch viele Unit-Tests angepasst werden
- ☑ Wir verwenden dann automatisierte Akzeptanztests als Sicherheitsnetz

Regeln und Best Practices



⌘ Inline Method

- ☑ Wir können uns das von vielen IDEs automatisierte Inline-Method-Refactoring zu Nutze machen, um Umleitungen aufzulösen
- ☑ Es wird die alte Implementation auf Basis der neuen Struktur implementiert
- ☑ Anschließend wird diese Implementation „inlined“

Wie können wir damit umgehen?



⌘ Regeln und Best Practices:

- ☑ Wir geben Hinweise im Umgang mit großen Refactorings, die sich in Projekten bewährt haben

⌘ Fragmente:

- ☑ Typische wiederkehrende Mechanismen zur Beseitigung von Architektur-Smells

Fragmente



⌘ Verschieben von Klassen

- ☑ Oft eine klassische und einfache Lösung, um Zyklen zwischen Packages, Subsystemen oder Schichten zu beseitigen
- ☑ Gut unterstützt durch IDEs („Move“)

Fragmente



⌘ Objektgeflecht-Fassade einführen

- ☑ Kann genutzt werden, um die Komplexität eines Subsystems zu verbergen
- ☑ Funktioniert nur dann sinnvoll, wenn das API der Fassade nicht zu groß wird

Fragmente



⌘ Klasse in Vererbungshierarchie verschieben

☒ Potenziell sehr schwieriges Refactoring, wenn im System stark von Polymorphie Gebrauch gemacht wird

☒ Mechanics schwierig, aber schon teilweise vorhanden

(work in progress auf <http://www.martinlippert.com>)

Fragmente



⌘ Zyklenbeseitigung

- ☑ Entweder klassisch durch Auftrennung von Klassen
- ☑ Oder mit DIP (*Dependency Inversion Principle*)

Fragmente



⌘ Plug-In einführen

- ☑ Systemteile ausgliedern
- ☑ Meist verbunden mit Inversion of Control Containers und Dependency Injection Pattern
- ☑ (siehe beispielsweise Eclipse)

„Refactorings in großen Softwareprojekten Komplexe Restrukturierungen erfolgreich durchführen“

- ⌘ Begriff: Refactoring
- ⌘ Architektur-Smells
- ⌘ Charakteristika großer Refactorings
- ⌘ Bausteine großer Refactorings
- ⌘ Prozess-Aspekte
- ⌘ Datenbanken und Refactoring
- ⌘ APIs und Refactoring

