Software - Architekten
Software - Techniker
it-wps

Refactorings in großen Softwareprojekten

Martin Lippert
ml@it-wps.de
lippert@acm.org

Stefan Roock
sr@it-wps.de
stefan@stefanroock.de

it-wps

- Kontext & Definitionen
- Architektur-Smells
- Best Practices für das Vorgehen bei großen Refactorings
- Wiederkehrende Fragmente für große Refactorings

- Nicht im Fokus
  - Refactorings von veröffentlichten Schnittstellen (APIs)
  - Refactorings, die sich bis auf die Datenbank auswirken

- Angrenzende Themenbereiche
  - Migration von Anwendungen
  - Refactoring to Patterns
  - Working Effectivly with Legacy Code

- Consultanting und Entwicklung
- Agile Methoden seit 1999.
- Fokus auf eXtreme-Programming.
- Projekte
  - 2 Entwickler für 6 Monate
  - 10 Entwickler für 11 Wochen
  - 30 Entwickler für 3 Jahre

Was ist Refactoring?

it-wps

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

[Fowler 99]

Häufig werden Refactorings durchgeführt, um *Code-Smells* zu beseitigen.

Code-Smells sind Defizite, die sich häufig auf nur eine Klasse beziehen.

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, …

Haben sich weit über agile Methoden hinaus etabliert.

- **Refactoring wird in Mikro-Schritten ausgeführt.**
- **Diese Schritte können als Mechanics formuliert werden.**
- **System ist nach jedem Mikro-Schritt lauffähig.**
- **Kontinuierliche Integration.**

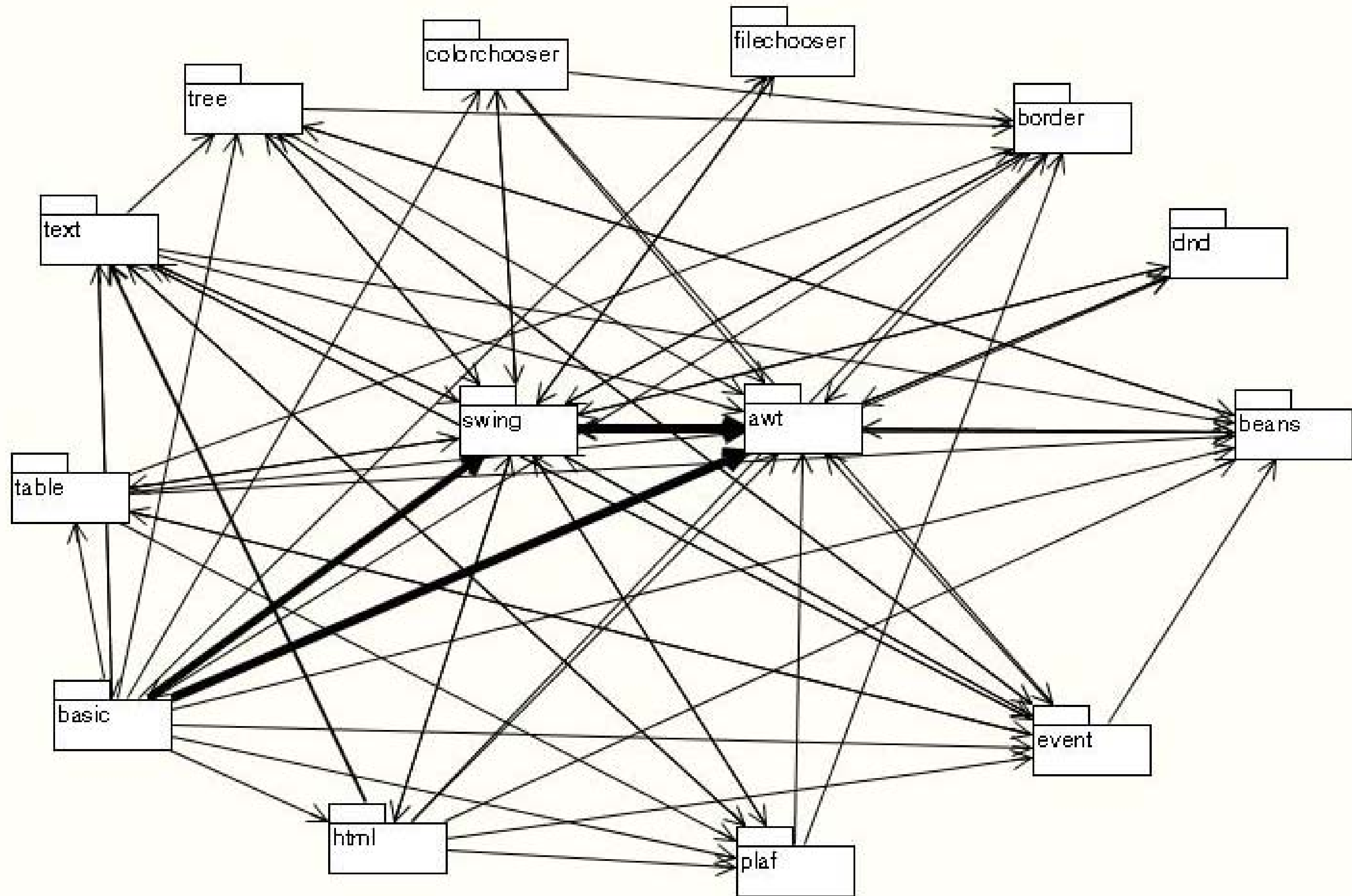
- **Aber es gibt Ausnahmen (z.B. *Rename Class* ohne Automatisierung)**
- **Daher**
  - ➔ ***Sichere Refactorings*** - während der Mechanics können keine Compilefehler auftreten.
  - ➔ ***Unsichere Refactorings*** - während der Mechanics kann das System zerbrechen.

- **Kleine Refactorings beseitigen Code-Smells.**
- **Darüber hinaus entstehen in größeren Projekten häufig strukturelle Probleme, sogenannte *Architektur-Smells*.**
  - ➔ Architektur-Smells sind potenzielle Defizite in den Beziehungen zwischen Paketen, Modulen, Klassen.
  - ➔ Unsere Erfahrung: Jedes größere Projekt hat Architektur-Smells. (Größeres Projekt: mehr als 6 Entwickler, länger als 6 Monate)

- ***Große Refactorings* beseitigen Architektur-Smells.**

Beispiel für Architektur-Smell: Zyklen

it-wps



Weitere Architektur-Smells

- **Parallele Vererbungshierarchien**
- **Falsche Verwendung von Vererbung**
- **Zyklen zwischen Klassen, Packages, Subsystemen, Schichten**
- **Technologie auf Vorrat, Übergeneralisierung**
- **Unbenutzter Code**
- **zu viele Abhängigkeiten zu Basisklassen**
- **keine Subsysteme, Schichten**
- **zu große Packages, Subsysteme, Schichten**
- **Subsystem-API umgangen**
- **Subsystem-API zu groß**
- **Schichtung durchbrochen**
- **...**

Warum entstehen Architektur-Smells?

it-wps

- Mißverständnis: Um Architektur muss man sich bei agilen Methoden nicht kümmern.
- Zeitdruck.
- Aufgeschobene kleine Refactorings.
- Prototyp wird produktiv.
- Unvollständiges/inkonsistentes Bild der Anforderungen.
- System sehr groß und unübersichtlich.
- Zu viele Entwickler im Team.
- Jeder macht mal Fehler.

Architektur-Smells finden

it-wps

- **Selbst Entwickeln**
  - ➔ Was ist im Weg?
- **Entwicklern zuhören:**
  - ➔ „Das hier nervt, aber wir haben keine Zeit, das umzustellen."
  - ➔ „Das hier passt überhaupt nicht, aber wenn wir das umstellen, laufen wir Gefahr, alles kaputt zu machen."
- **Tools zur Architekturanalyse, z.B.**
  - ➔ **Sotograph** (http://www.sotograph.de).
  - ➔ **XRadar** (http://xradar.sourceforge.net)
  - ➔ Dr. Freud (http://www.freiheit.com)
  - ➔ ...
- **Weitere Tools, z.B.**
  - ➔ JDepend
  - ➔ PMD
  - ➔ Checkstyle
  - ➔ ...

Große Refactorings: Charakterisierung

it-wps

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

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

it-wps

Große Refactorings: Probleme

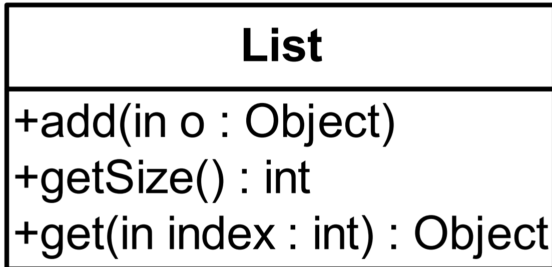
it-wps

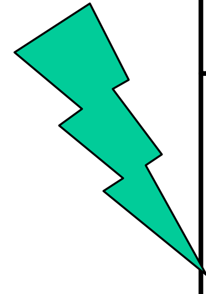
- **man läuft schnell in Sackgassen**
- **wegen Interferenzen mit restlicher Entwicklung kann man nicht einfach so zurück**
- **man verliert schnell den Überblick**
- **die Planung ist sehr schwierig**
- **Sicherheit wg. Zerbrochenen Unit-Tests reduziert**
- **unter Projektdruck neigen Refactorings zum Versanden**
  - ➔ auf halbem Wege abgebrochene Refactorings verschlechtern die Systemstruktur statt sie zu verbessern

Beispiel

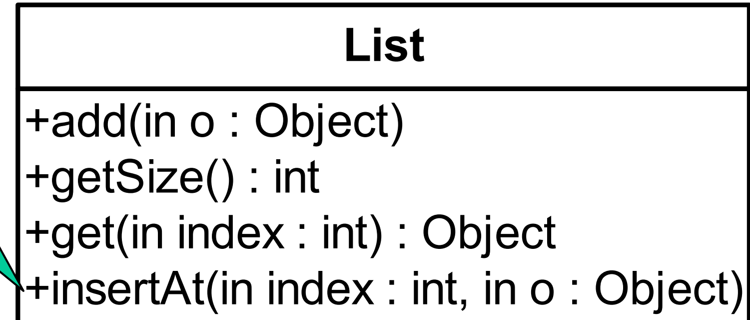
it-wps

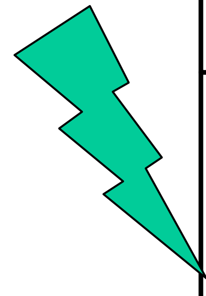
1

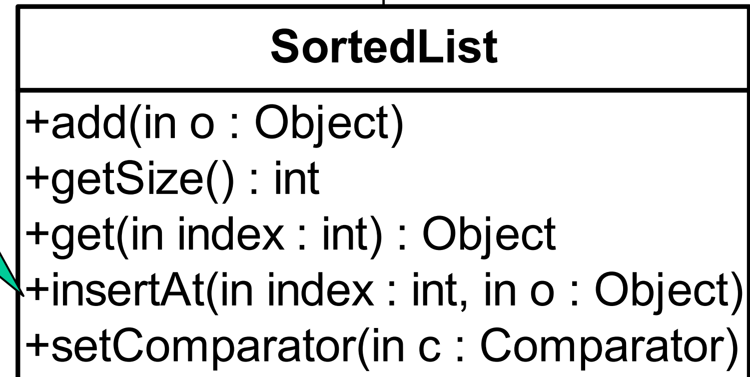




2

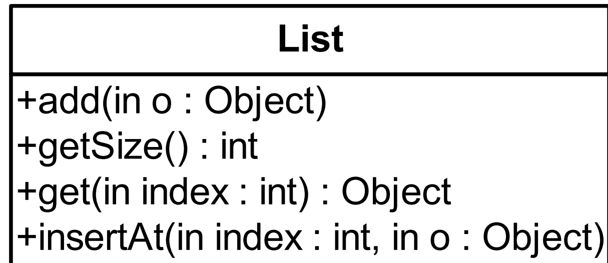


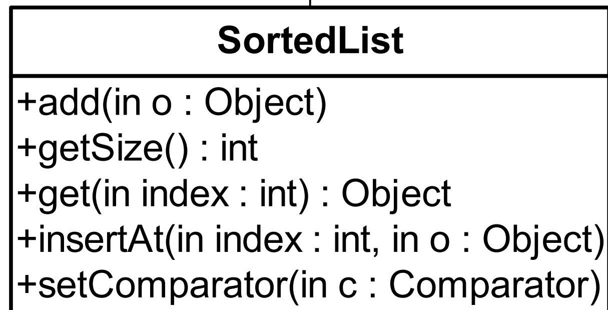


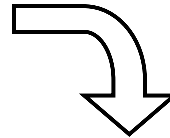


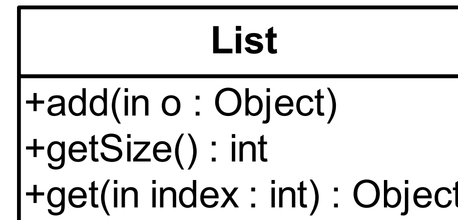
- **Problemfall: *insertAt* ohne Sinn für *SortedList***

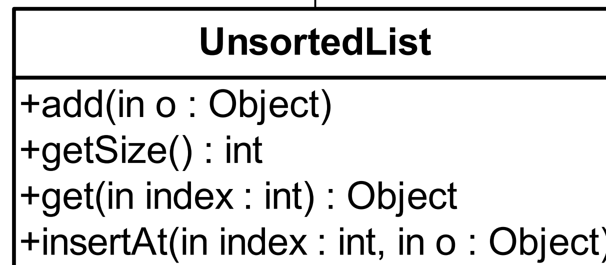
Große Refactorings: Beispiel
Das Ziel

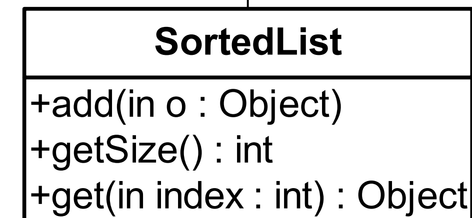








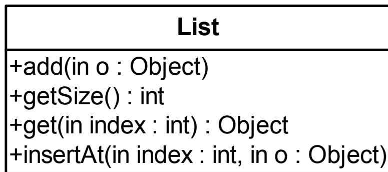


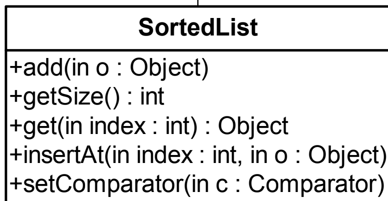


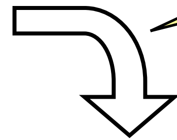
Große Refactorings: Beispiel
Der erste Versuch

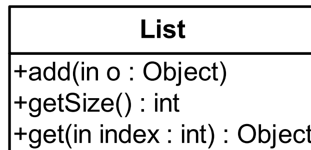
- List ist eigentlich eine unsortierte Liste.
- Daher benennen wir *List* nach *UnsortedList* um.
- Anschließend ziehen wir aus *UnsortedList* die Oberklasse *List* heraus.

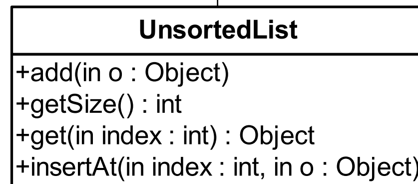
- Zuletzt hängen wir *SortedList* in der Vererbungshierarchie unter *List*.

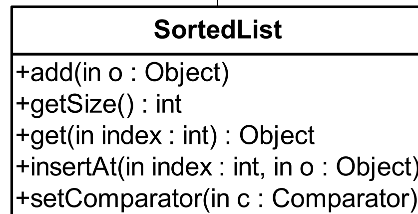


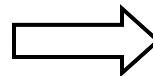


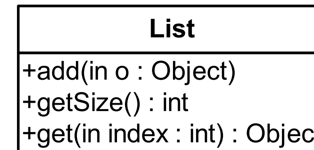


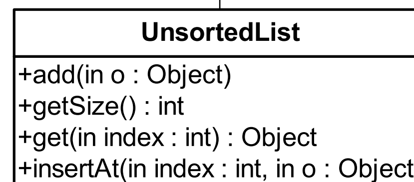


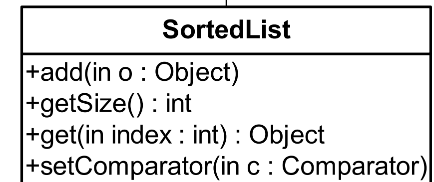










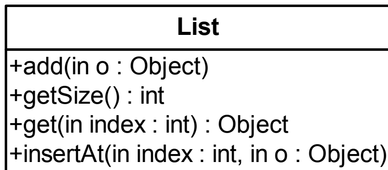


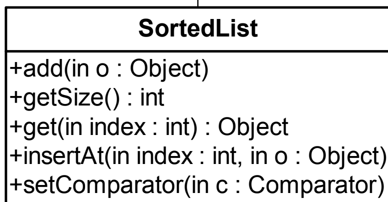
Große Refactorings: Beispiel
Die Sackgasse

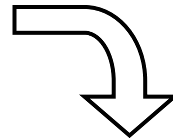
it-wps

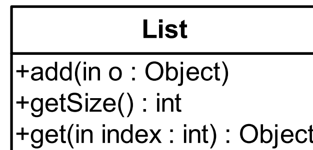
Die Umstellung des Klientencodes ist problematisch.

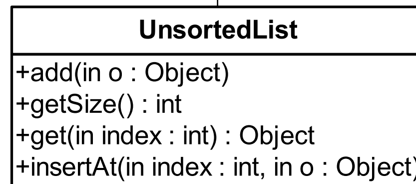
Merke: Umhängen in der Typhierarchie ist generell problematisch.

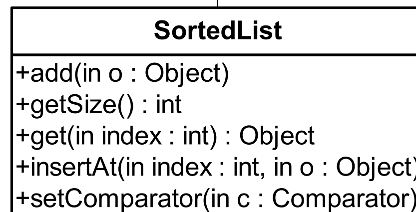


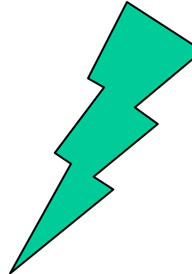


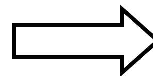


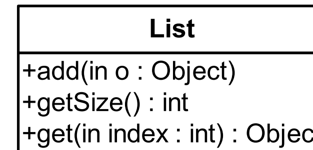


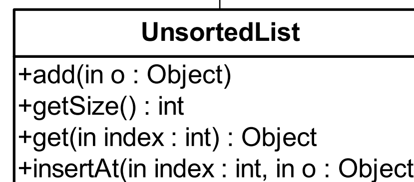


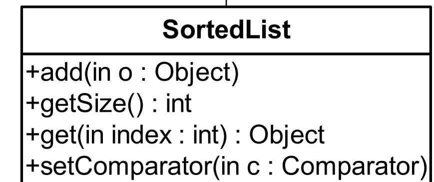






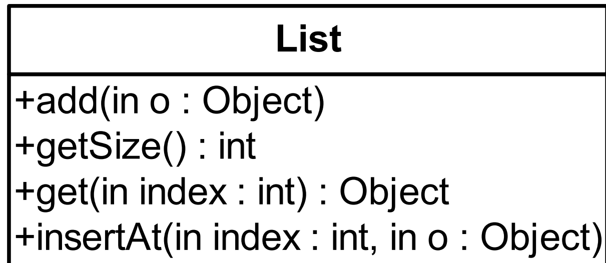


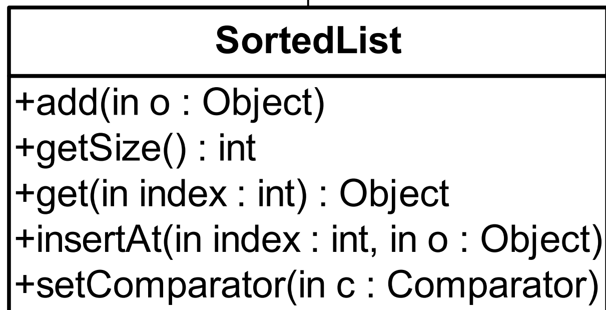


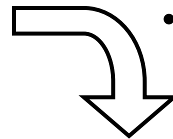


Große Refactorings: Beispiel
So geht's

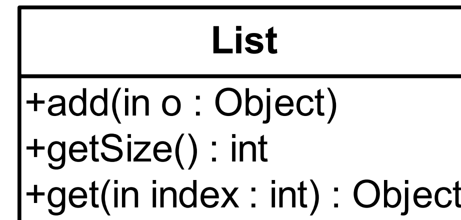
it-wps

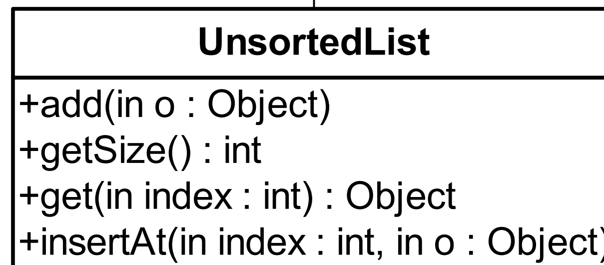


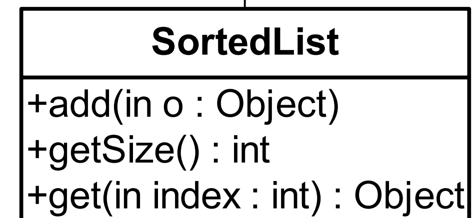




- ***List.insertAt*** **auf** ***deprecated*** **setzen**
- **Leere Klasse** ***UnsortedList*** **von** ***List*** **ableiten**
- ***List.insertAt*** **nach** ***UnsortedList*** **kopieren**
- **Referenzen auf** ***List.insertAt*** **entfernen**
- ***List.insertAt*** **löschen**







Best Practices für das Vorgehen bei großen Refactorings

it-wps

Refactorings explizit in den Planungsprozess einbeziehen

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

it-wps

Refactoring-Planungs-Session

- Größere Refactorings mit dem gesamten Team diskutieren und planen
- Spannungsfeld: Upfront-Design vs. Refactoring-Planung

Refactoring-Pläne erstellen

- Refactoring-Route aufschreiben
- Refactoring-Plan prominent *veröffentlichen*
- Refactoring-Plan als Tracking-Instrument nutzen
- Unsichere Refactoring-Schritte kennzeichnen
- Unsichere Refactoring-Schritte nach Möglichkeit an den Anfang stellen

it-wps

Umleitungen

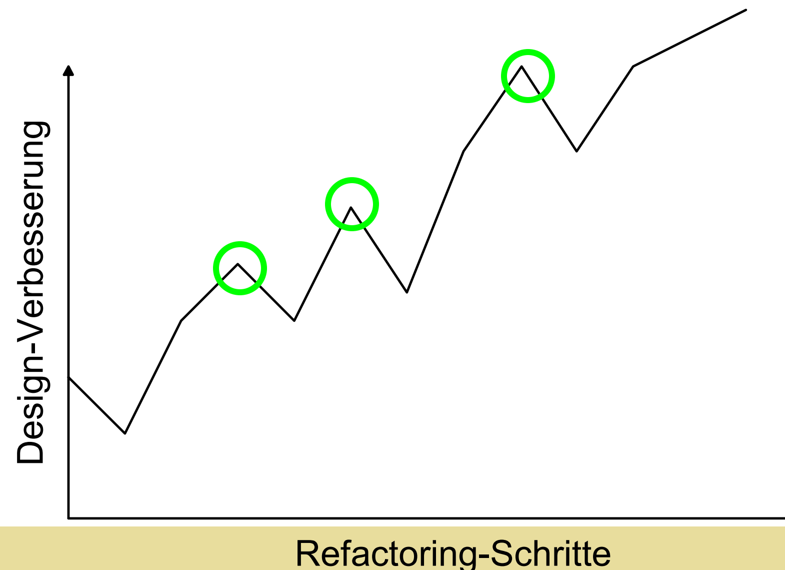
- ➔ Um ein Refactoring in kleine Schritte zu zerlegen, müssen häufig Umleitungen in den Code eingebaut werden.
- ➔ So kann ein Refactoring schrittweise durchgeführt werden und das System ist trotzdem immer lauffähig.
- ➔ Umleitungen müssen aber als solche gekennzeichnet werden.
  - Z.B. mit *deprecated* Tag.

it-wps

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 das System einen verbesserten Stand erreicht, aber noch nicht das endgültige Design



it-wps

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

it-wps

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 wenige viele Compile-Fehler auftreten
  - Die Warnungen müssen in einer beliebigen Reihenfolge abgearbeitet werden können

it-wps

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

it-wps

Inline Method

- ➔ Wir können uns das von vielen IDEs automatisierte Inline-Method-Refactoring zu Nutze machen, um Umleitungen (teilweise) aufzulösen

- ➔ Neue Struktur steht neben der alten Struktur.
- ➔ Die alte Struktur wird auf Basis der neuen Struktur implementiert.
- ➔ Anschließend wird diese Implementation „inlined“.

- ➔ Siehe: Tammo Freese: „*Inline Method Considered Helpful*“.

Best Practices: Inline Method 2/3

it-wps

```
/**
 * @deprecated use druckeDokument instead
 */
public void drucke (String dok) {
  druckeDokument(new Dokument(dok));
}

public void druckeDokument (Dokument obj) {
  ... implementation ...
}
```

```
...
String meinDokument = ...;
...
meinDrucker.drucke(meinDokument);
...
```

Best Practices: Inline Method 3/3

it-wps

```java
/**
 * @deprecated use druckeDokument instead
 */
public void drucke (String dok) {
  druckeDokument(new Dokument(dok));
}

public void druckeDokument (Dokument obj) {
  ... implementation ...
}
```

```java
...
String meinDokument = ...;
...
meinDrucker.druckeDokument(
              new Dokument(meinDokument));
...
```

Fragmente für große Refactorings

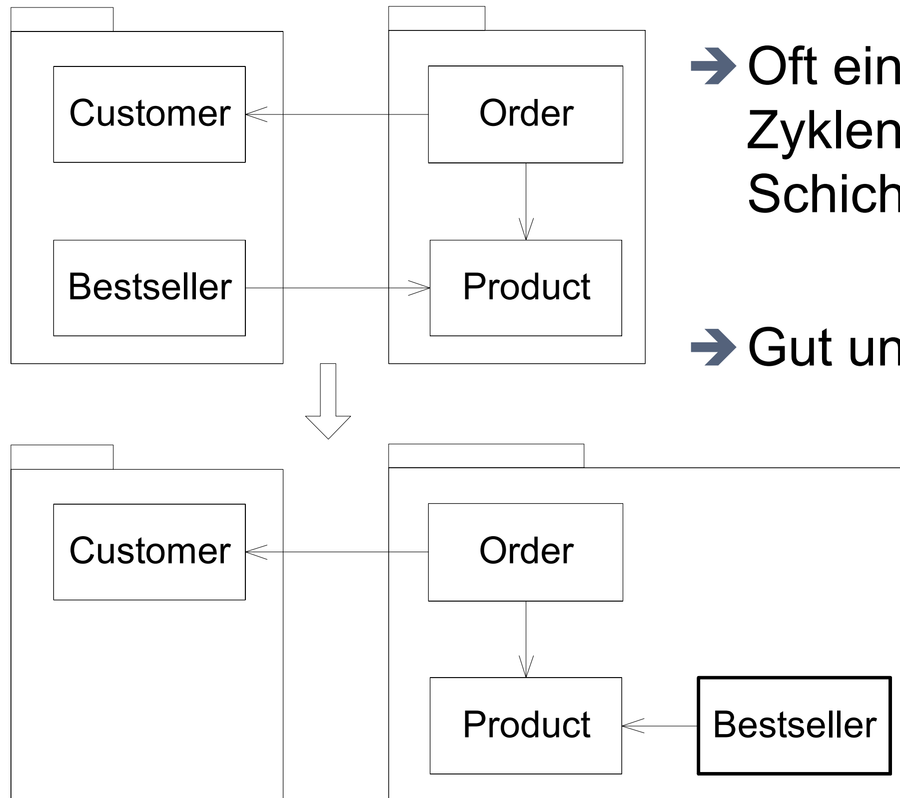
- Große Refactorings sind nicht vollständig automatisierbar.
- Ein Katalog großer Refactorings ist noch nicht verfügbar.
- Es gibt aber Refactoring-Fragmente, die man benutzen kann, um häufige vorkommende Architektur-Smells zu beseitigen.
  - haben Mechanics

Beispiele für Fragmente

- Klassen verschieben (einfache Zyklenbeseitigung)
- Vererbungshierarchien umstellen (Mechanics noch unvollständig)
- Fassaden einführen (Abhängigkeiten reduzieren)
- Interface einführen (Entkopplung, Zyklen beseitigen)
- Klassenvererbung durch Interface ersetzen (Entkopplung)
- Plugin einführen (Abhängigkeiten reduzieren)

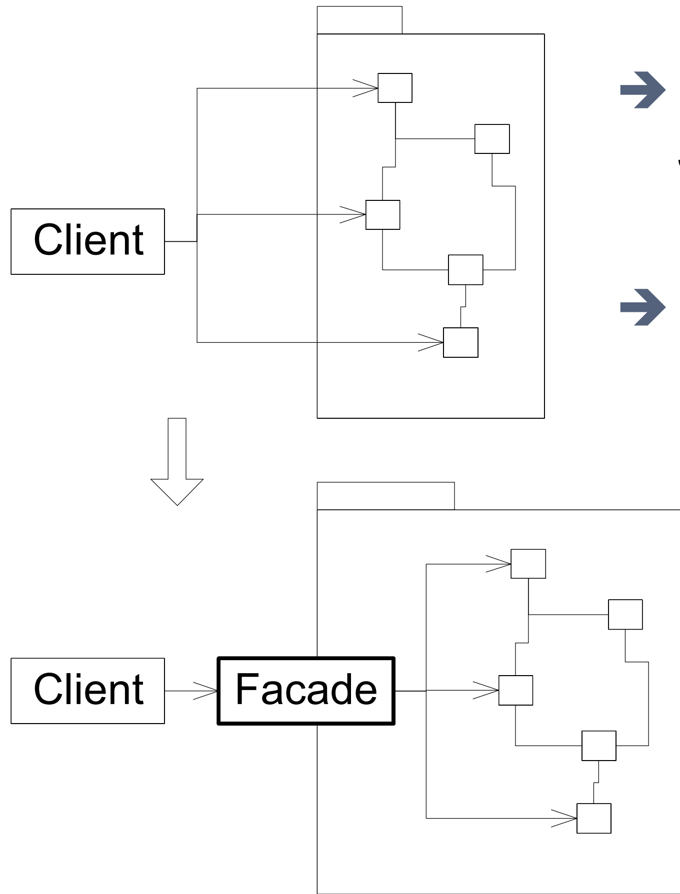
Fragment: Verschieben von Klassen

it-wps



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

Fragment: 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

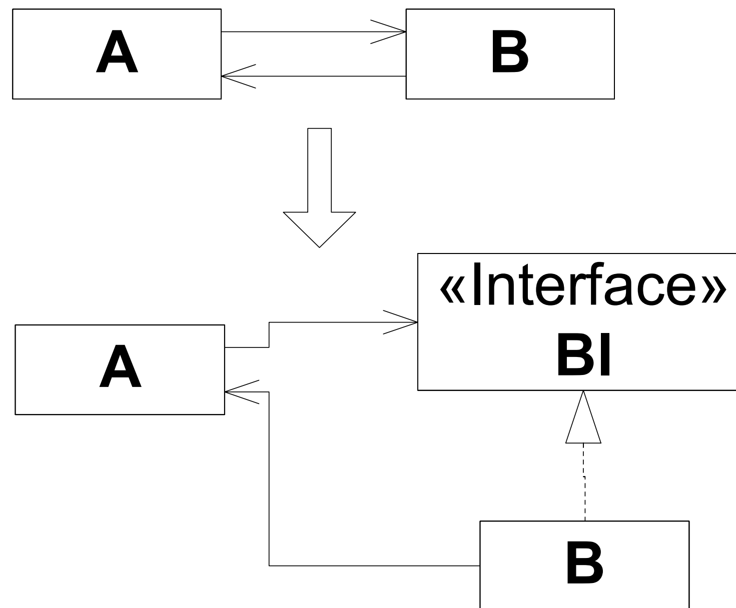
Fragment: Vererbungshierarchie umstellen

- Potenziell sehr schwieriges Refactoring, wenn im System stark von Polymorphie Gebrauch gemacht wird
- Mechanics schwierig; bisher nur teilweise vorhanden
- Vererbungsbeziehung müsste als *deprecated* gekennzeichnet werden können.
- *JMigrator*
  - erkennt zusätzliche Meta-Tags für große Refactorings: `deprecated-inheritance, future`
  - Open-Source
  - Eclipse-Plugin
  - Download bei Source-Forge: http://www.sf.net/projects/jmigrator

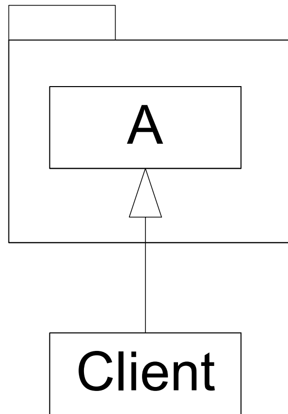
Fragment: Interface einführen

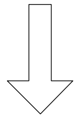
it-wps

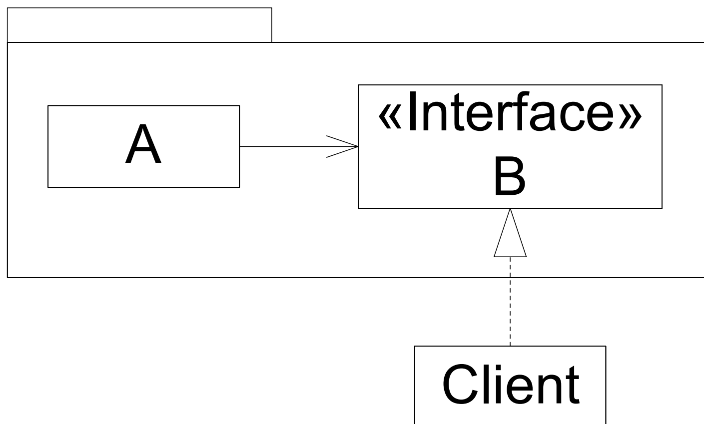
- ➔ Z.B., um Zyklen zwischen Klassen zu beseitigen
- ➔ DIP (*Dependency Inversion Principle*)
- ➔ zieht häufig „*Plugin einführen*“ nach sich



Fragment: Klassenvererbung auf Interface umstellen 1/2

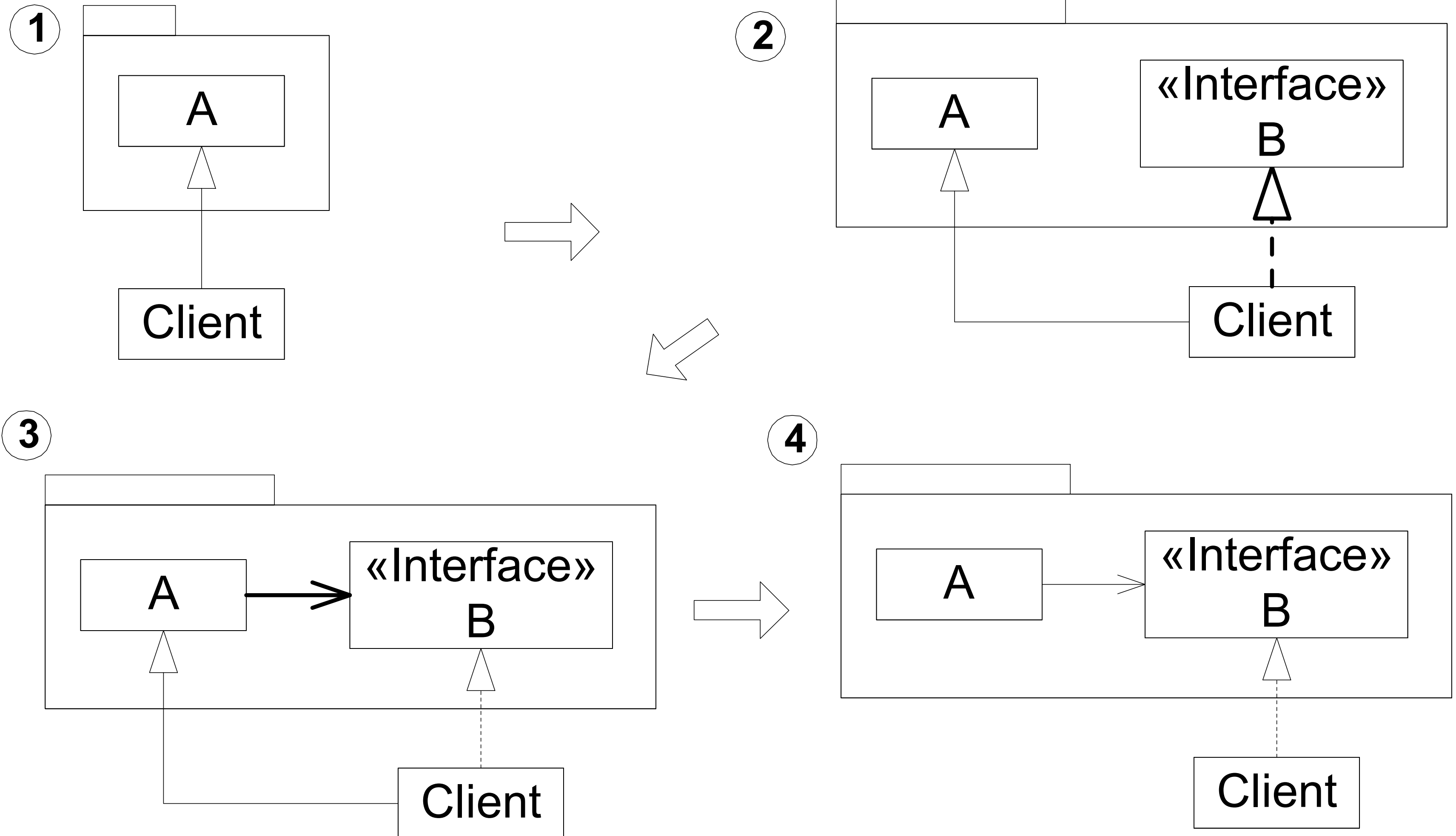




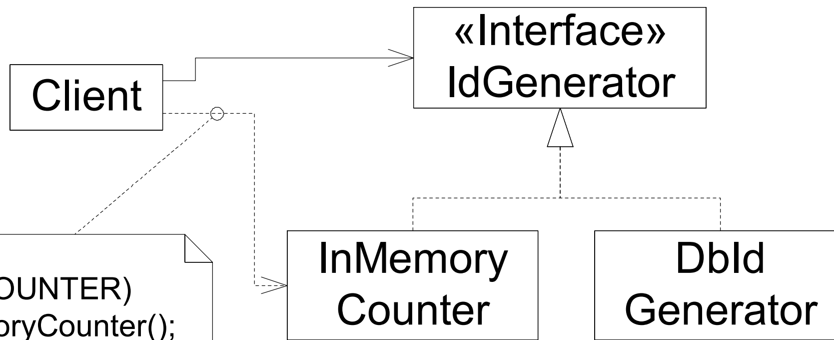


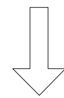
- Entkopplung
- White-Box-Framework --> Black-Box-Framework
- zieht häufig „*Plugin einführen*“ nach sich

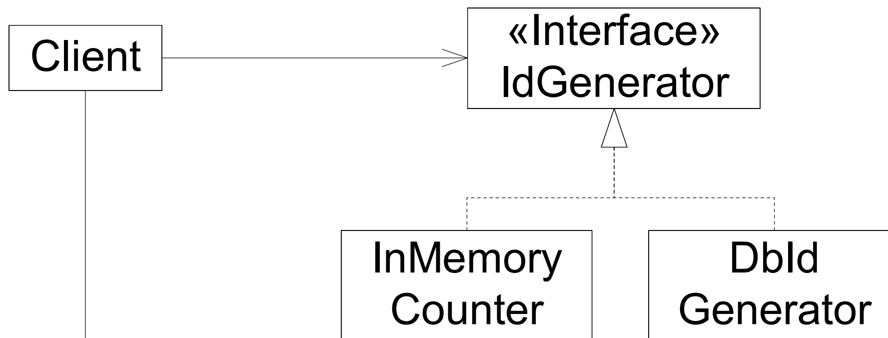
Fragment: Klassenvererbung auf Interface umstellen 2/2

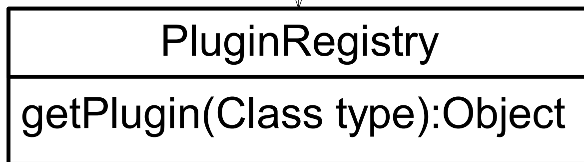


Fragmente: Plugin einführen









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

- ➔ Jedes größere Projekt hat Architektur-Smells.
- ➔ Subjektive Einschätzung der Architektur und tatsächlicher Zustand häufig weit auseinander.
- ➔ Zyklen zwischen Schichten, Subsystemen und Packages sind sehr häufige Architektur-Smells.
- ➔ Häufig sind große Refactorings nicht so schlimm, wie sie zuerst scheinen.
  - Zyklen zwischen Schichten & Subsystemen lassen sich zu einem relevanten Anteil häufig durch Verschieben von Klassen beseitigen.
- ➔ Große Refactorings werden zu lange aufgeschoben.
- ➔ Wenn sie doch angefangen werden, versanden sie häufig.
- ➔ Mut zum Risiko notwendig („*rumroocken*“).

„Refactorings in großen Softwareprojekten Komplexe Restrukturierungen erfolgreich durchführen“

it-wps

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

it-wps

- Open-Source
- Eclipse-Plugin
- erkennt zusätzliche Meta-Tags für API-Refactorings und große Refactorings
  - `deprecated-inheritance`
  - `future`
- Download bei Source-Forge
  - http://www.sf.net/projects/jmigrator