Refactoring in kleinen und großen Projekten

Dipl.-Informatiker Martin Lippert
Senior IT-Berater
martin.lippert@it-agile.de

Dipl.-Informatiker Andreas Havenstein
Softwareentwickler
andreas.havenstein@it-agile.de

http://www.it-agile.de

Über uns...

Martin Lippert
martin.lippert@it-agile.de

- Senior IT-Berater bei it-agile GmbH in Hamburg
- Schwerpunkt-Gebiete
  - Software-Architektur
  - Agile Softwareentwicklung
  - Eclipse-Technologie
  - Refactoring
  - Aspektorientierte Programmierung

Andreas Havenstein
andreas.havenstein@it-agile.de

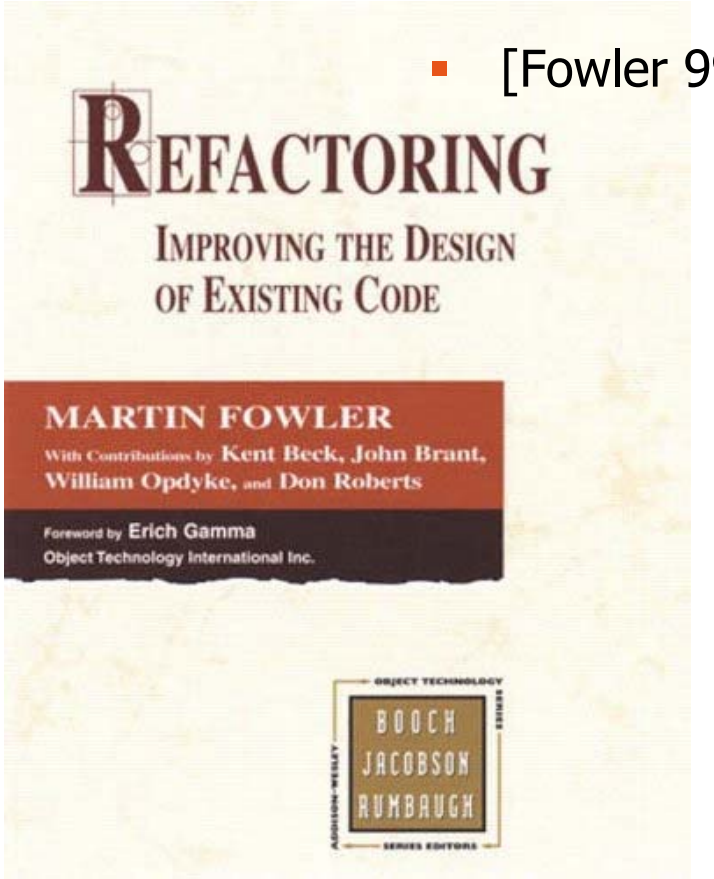
- Softwareentwickler bei it-agile GmbH in Hamburg
- Schwerpunkt-Gebiete
  - Agile Softwareentwicklung
  - Refactoring
  - J2EE

Überblick

- Die Idee
  - Eine kurze Einführung zum Thema Refactoring
  - Sicherheitsnetze beim Refactoring
- Die Realität
  - Refactoring in der Praxis
- Die Herausforderungen im Kleinen
  - Das tägliche Refactoring
  - Tipps & Tricks
- Die Herausforderungen im Großen
  - Warum sind größere Refactorings anders als kleine?
  - Best Practices
- Die Werkzeuge
  - Probleme und Ergebnisse analysieren
- Das Geheimnis
  - Think!!! - Refactorings zusammensetzen
- Fazit

Was ist Refactoring?

- **„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 deuten *Code-Smells* auf nötige Refactorings hin.
- Code-Smells sind Defizite, die auf Schwachstellen hinweisen, z.B.:
  - zu lange Methoden
  - zu große Klassen
  - zyklische Beziehungen

Refactoring: Eine kurze Einführung

- Refactoring ist die Umstrukturierung eines Softwaresystems unter Beibehaltung seiner Funktionalität.
  - Nicht jede Änderung ist ein Refactoring.
- Definition ist kontextabhängig.
  - Beispielweise verändert fast jede Änderung am Code sein Laufzeitverhalten.
  - Aber: Meistens liegt der Schwerpunkt auf der fachlichen Funktionalität (was der Benutzer an der Oberfläche sieht)
- Eine wichtige Frage:
  - Wie stellen wir sicher, dass ein Refactoring nicht aus versehen die Funktionalität verändert?

Sicherheitsnetz Unit-Tests

- Die Funktionalität wird durch Unit-Tests geprüft:
  - Vor und nach einem Refactoring müssen alle Unit-Tests durchlaufen
  - Unit-Tests fungieren als Sicherheitsnetz für das Refactoring
  - Ohne Unit-Tests ist es sehr unsicher, Refactorings durchzuführen

➔ **Unit-Tests sind eine Voraussetzung für Refactorings!!!**

- Aber was tun, wenn keine Unit-Tests vorhanden sind?
  - Zuerst schrittweise Unit-Tests implementieren
  - Dann Refactoring durchführen

Sicherheitsnetz IDE

- Automatisierte Refactorings von der IDE sichern zu, dass das Verhalten der Software nicht verändert wird.
  - Eclipse oder IDEA sind diesbezüglich recht umfangreich
- Aber Vorsicht:
  - Auch diese sehr fortgeschrittenen IDEs können Fehler enthalten, komplizierte Zusammenhänge ggf. nicht erkennen und nicht wirklich sicherstellen, dass sich das Verhalten der Software nicht verändert
- Rename und Move sind sicher
- Extract Method beispielsweise kann schon problematisch werden
  - Beispiel…

Extract Method und Inner Classes

- Ein kleines Beispiel…

Sicherheitsnetz Akzeptanztests

- Unter Umständen müssen Unit-Tests an Veränderungen im Code angepasst werden.
- Wie können sie dann noch als Sicherheitsnetz fungieren?
  - Wenn die Unit-Tests mit Laufe des Refactorings verändert werden müssen, können sich Fehler einschleichen

- Wir nutzen Akzeptanztests, um Refactorings auf höherer Ebene abzusichern.
  - Z.B. Akzeptanztests mit FIT oder Fitnesse

Generelle Prinzipien

- Refactoring wird in Mikro-Schritten ausgeführt.
- Diese Schritte können als Mechanics formuliert werden
  - Siehe Mechanics in [Fowler 99]
- 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.

Refactoring in der Praxis

- Das Ideal:
  - Bevor eine neue Anforderung implementiert wird, zuerst prüfen, ob die Struktur für die neue Anforderung geeignet ist. Wenn nicht: Refactoring
  - Anforderung implementieren. Dabei ggf. weitere Refactorings.
  - Nach Implementierung des Refactorings prüfen, ob die Struktur noch sauber ist. Wenn nicht: Refactoring.

- Beobachtung: Refactorings werden viel zu selten durchgeführt.

- Warum?
  - „Ist doch egal…" ?
  - Mangelnde Disziplin?
  - Aufgeschobene Refactorings werden immer größer?
  - Keine Testcases, um Refactorings zu überprüfen? Kein Sicherheitsnetz?

Folgen

- Die Struktur des Systems degeneriert und es wird immer schwieriger, Refactorings durchzuführen
- Die nötigen Refactorings werden immer größer und damit auch risikoreicher

➔ **Refactoring ist heute elementarer Bestandteil der Softwareentwicklung!!!**

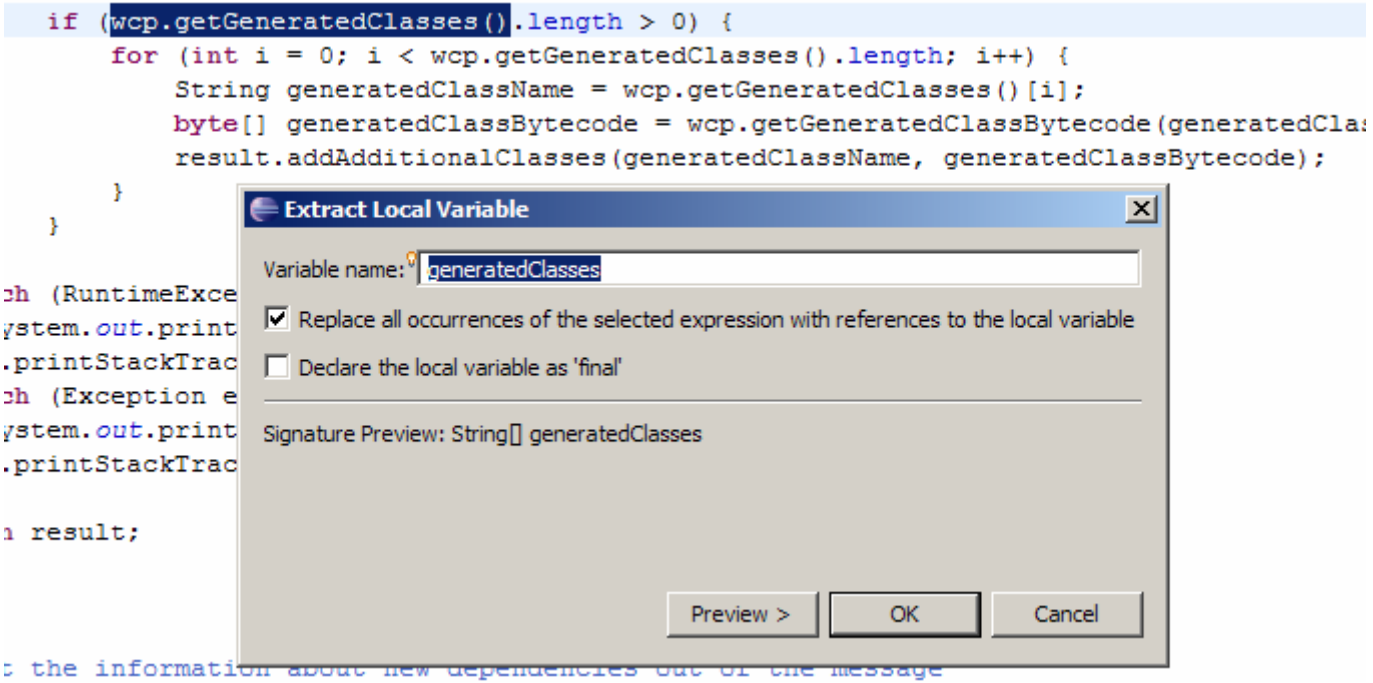
Besser viele kleine Refactorings

- Häufig kleine Refactorings durchzuführen ist nicht schwer:
  - Das braucht wenig Zeit
  - Ist häufig gut unterstützt durch moderne IDEs
  - Ist besser als selten größere Refactorings durchzuführen

➔ **Refactorings möglichst durch die IDE erledigen lassen!!!**

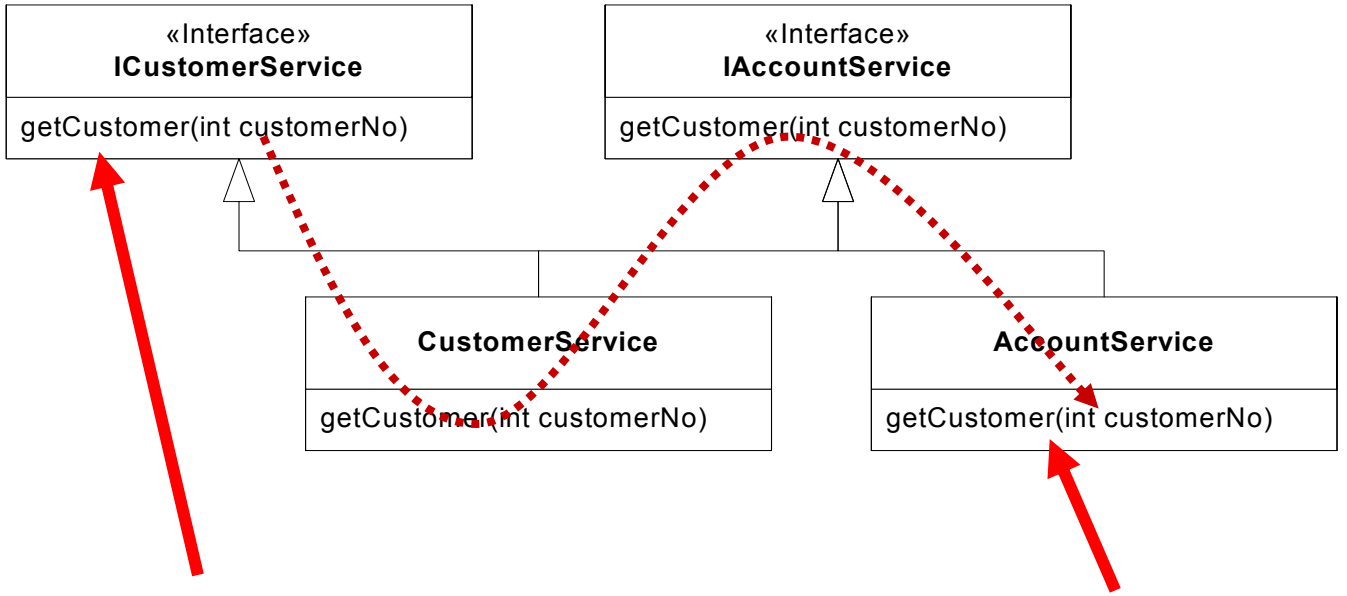
No Refactoring by Copy&Paste

- Refactoring mit Copy&Paste gehört der Vergangenheit an!!!



Interessante Effekte

- Z. B. Methode aus einem Interface umbenennen



Rename Method an dieser Stelle verändert auch den Methodennamen

Trotzdem große Refactorings?

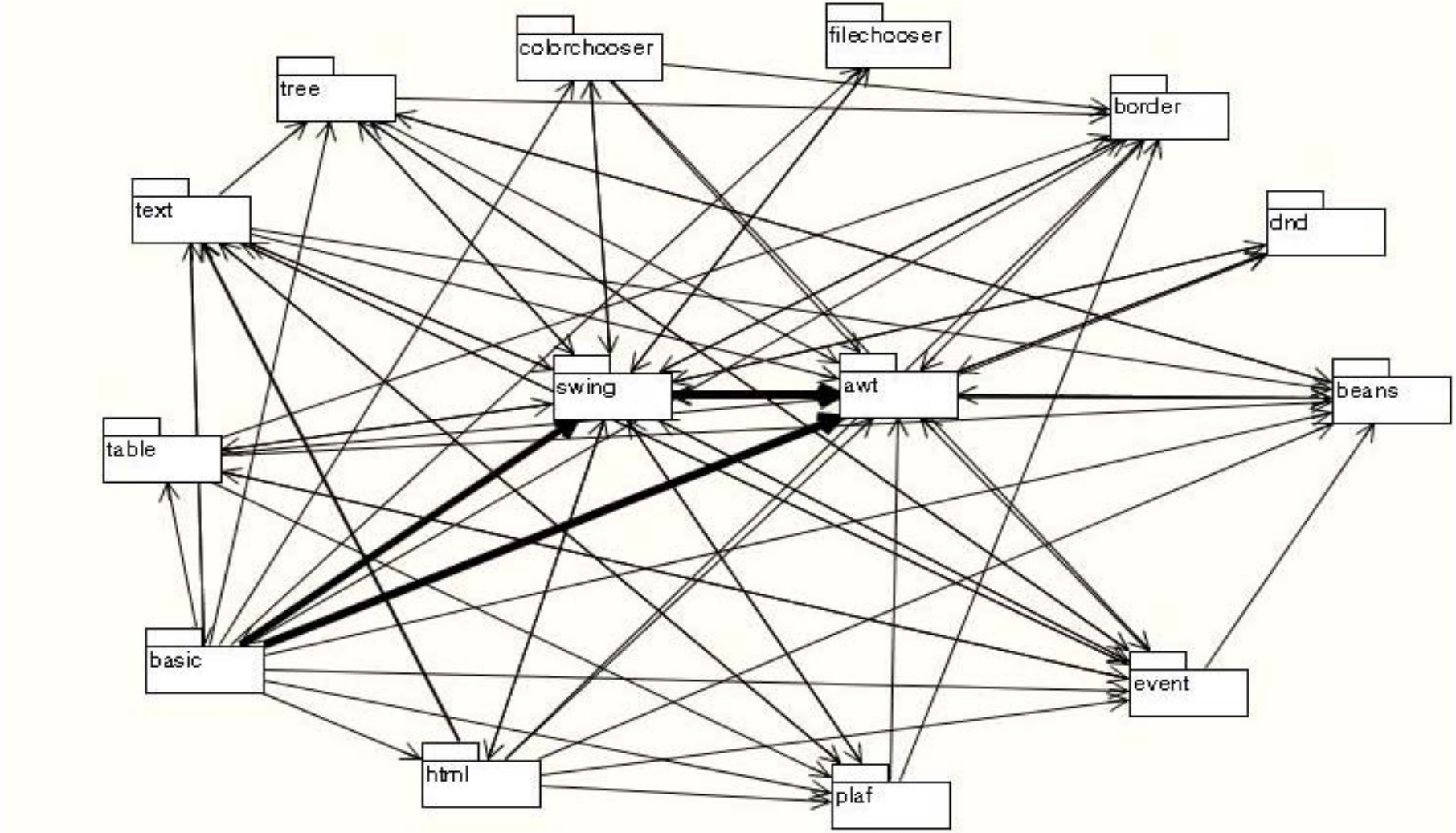
- Viele kleine Refactorings sind gut und unersetzbar.
- Sie dienen uns auch dazu, die Architektur des Systems weiter zu entwickeln.

- Können trotzdem größere Refactorings nötig werden?
- Ja!
  - 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 und große Refactorings

- In größeren Projekten entstehen häufig strukturelle Probleme, so genannte *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* helfen, Architektur-Smells zu beseitigen.

Beispiel für Architektur-Smell: Zyklen



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
- ...

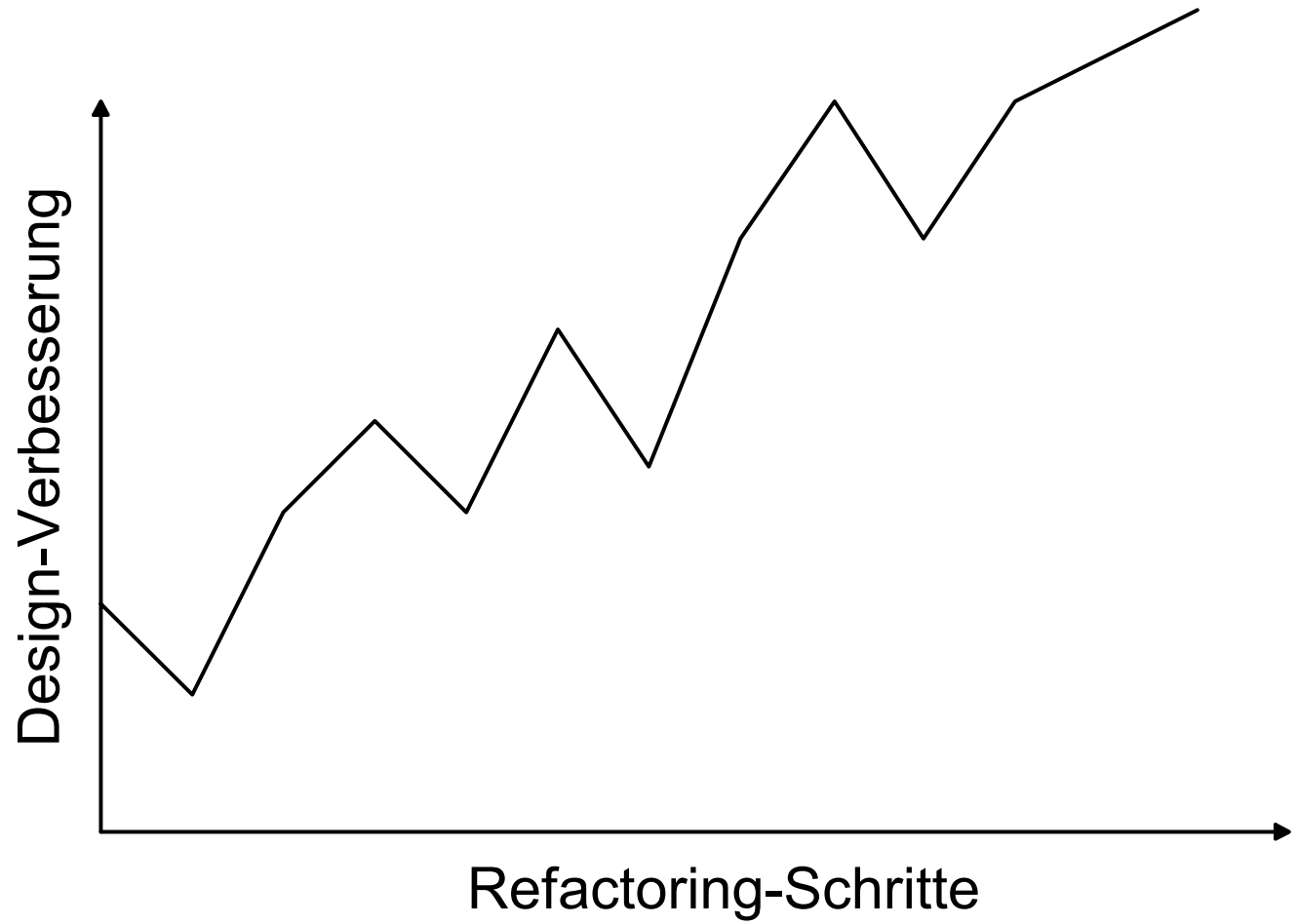
Architektur-Smells finden

- 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

- 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 :-)



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
- Sicherheit wg. Zerbrochenen Unit-Tests reduziert
- unter Projektdruck neigen Refactorings zum Versanden
  - auf halbem Wege abgebrochene Refactorings verschlechtern die Systemstruktur statt sie zu verbessern

Probleme lösen

- Best Practices:
  - Immer schön refaktorisieren, wenn einem etwas auffällt
  - Refactoring-Tools ausgiebig nutzen (da schlummern viele Features, die noch gar nicht richtig eingesetzt werden)
  - Tools zum Identifizieren von Schwächen nutzen (möglichst auch ständig bei der Entwicklung)
  - Nicht vor Refactorings zurückschrecken, aber vorher Tests bauen
  - Refactorings im Team diskutieren

  - Patterns und Practices für große Refactorings

Best Practices: Einplanung von großen Refactorings

- Refactorings explizit in den Planungsprozess einbeziehen
  - Refactoring-Budget pro Iteration
  - Refactoring-Iterationen bei Bedarf
  - Regelmäßige Refactoring-Iterationen

Best Practices: Refactoring-Planungs-Session

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

Best Practices: Refactoring-Pläne

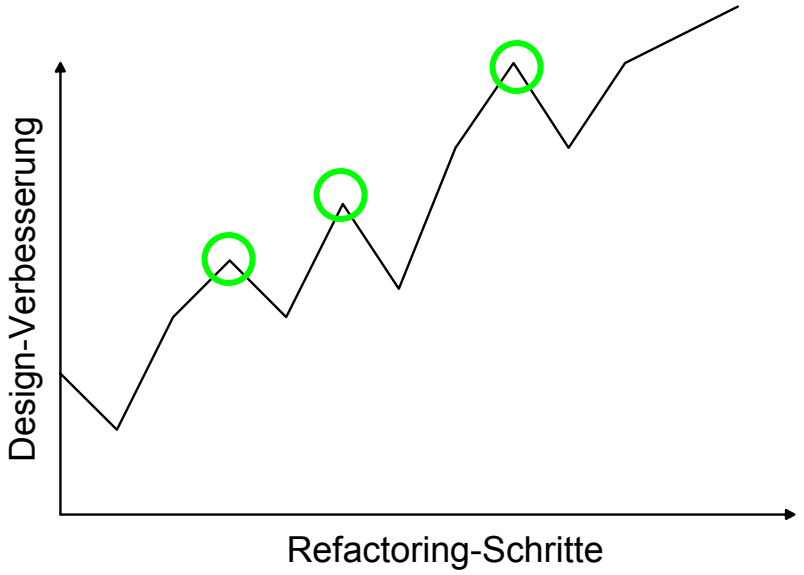
- Refactoring-Pläne erstellen
  - Refactoring-Route aufschreiben
  - Refactoring-Plan prominent *veröffentlichen*
  - Refactoring-Plan als Tracking-Instrument nutzen
  - Unsichere Refactoring-Schritte kennzeichnen
  - Unsichere Refactoring-Schritte nach Möglichkeit an den Anfang stellen

Best Practices: Umleitungen

- 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.

Best Practices: Safe-Points

- 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



Best Practices: Branches

- 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

Best Practices: Inline Method 1/3

- 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

```
/**
 * @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

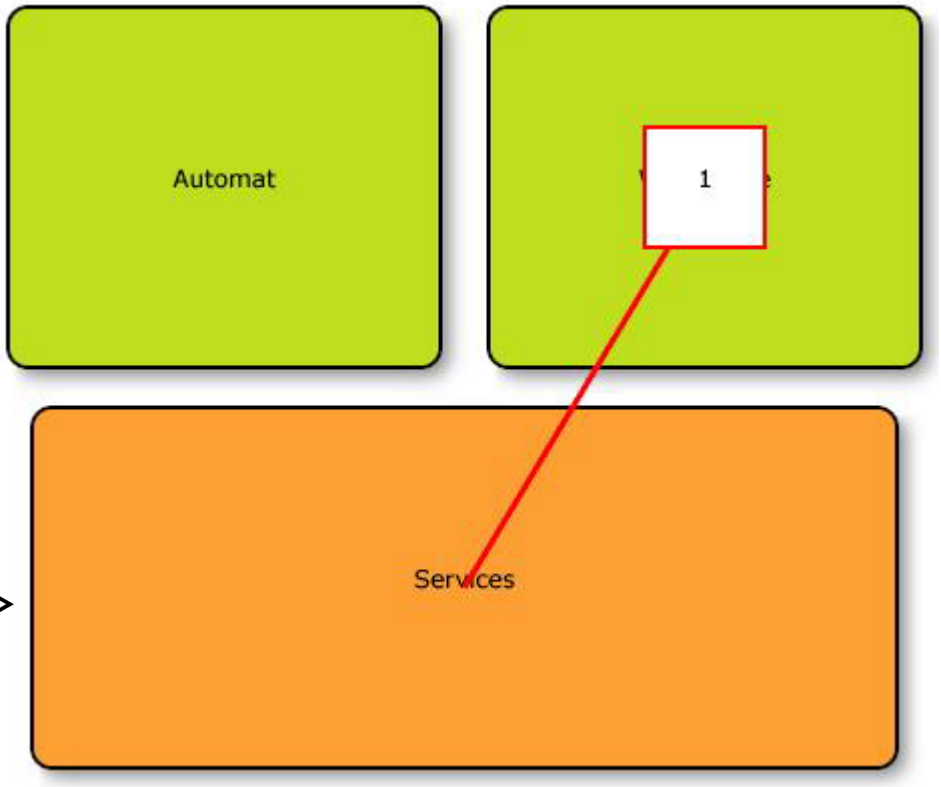
```
/**
 * @deprecated use druckeDokument instead
 */
public void drucke (String dok) {
  druckeDokument(new Dokument(dok));
}

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

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

Toolunterstützung für große Refactorings

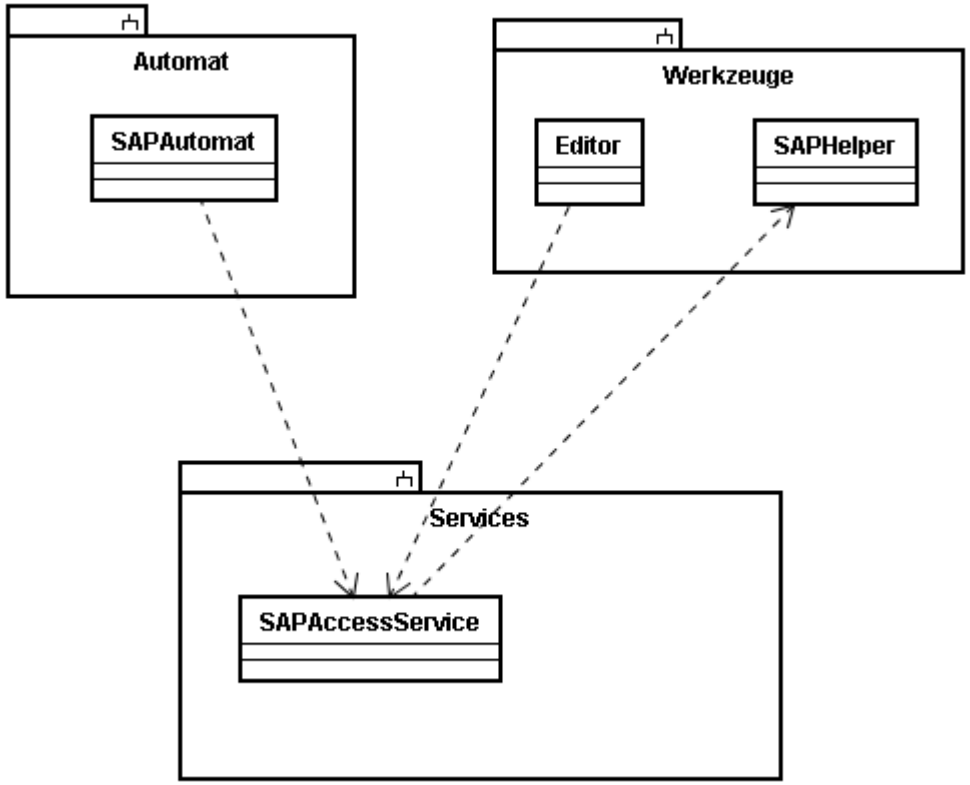
- Mit XRadar die Architektur definieren und kontrollieren („keep the graph green to keep the code clean…“)

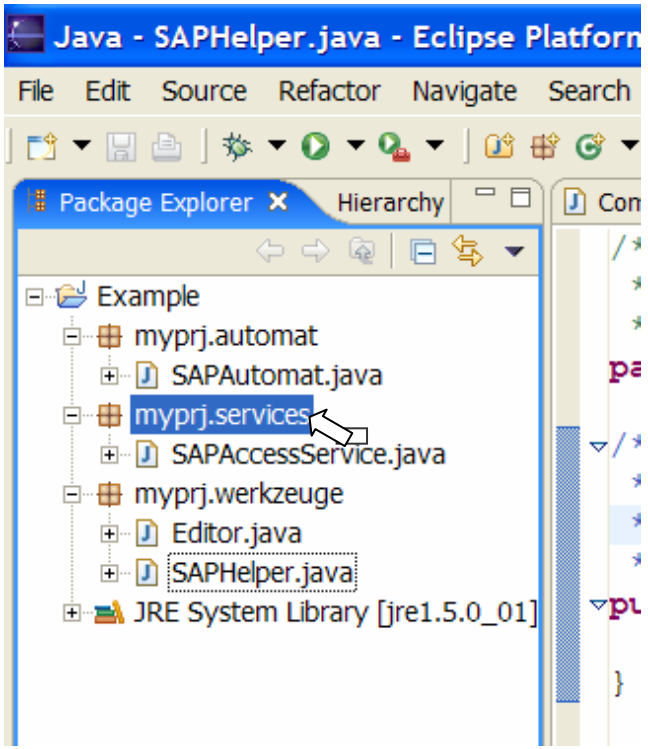
```
<radar-config>
  <subsystems>
    <subsystem id=„Automat“ level=„1“>
      <included-packages>
        <package-root value=„myprj.automat“/>
      </included-packages>
      <legal-subordinates>
        <subsystem id=„Services“/>
      </legal-subordinates>
    </subsystem>

    <subsystem id=„Services“ level=„2“>
      <included-packages>
        <package-root value=„myprj.services“/>
      </included-packages>
      <legal-subordinates>
      </legal-subordinates>
    </subsystem>
  ...
```



Toolunterstütztes Refactoring: Einfaches Beispiel

- Mit *Move-Refactorings* eine saubere Architektur herstellen

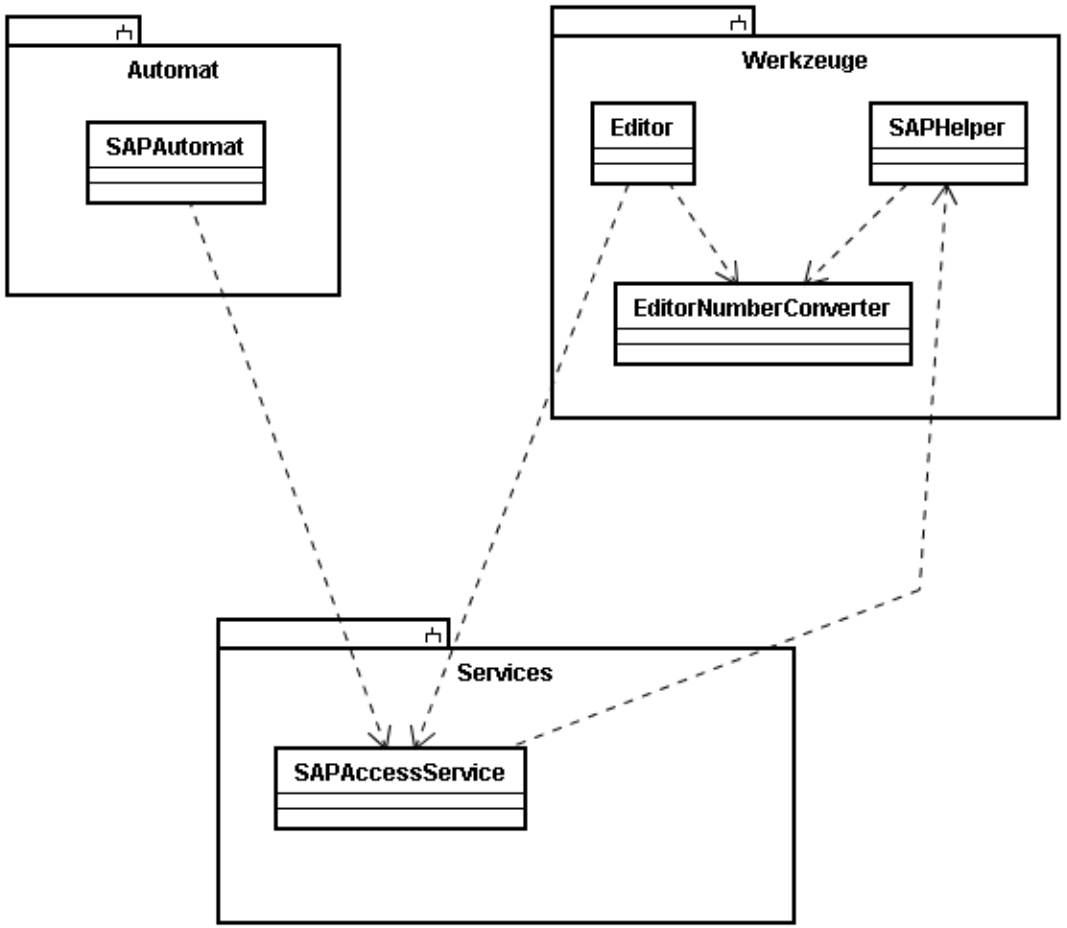




- Drag & Drop-Refactoring in Eclipse

Toolunterstütztes Refactoring: Komplexeres Beispiel

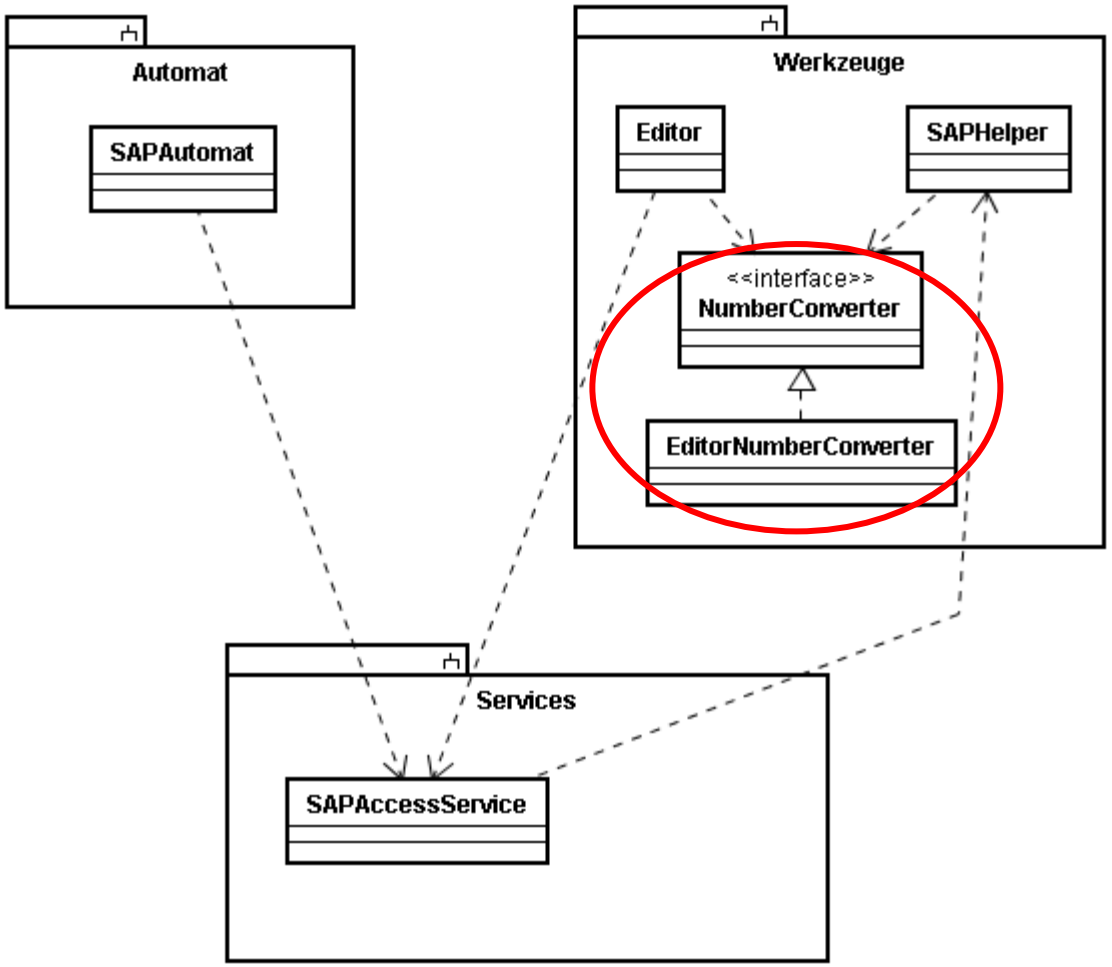
- Zyklische Package-Beziehungen lassen sich wegen starker Kopplung meist nicht mit einfachen Move-Refactorings auflösen.



- Auflösen verbotener Beziehungen durch lose Kopplung
- Mechanics:
  1. *Extract Interface* auf **`EditorNumberConverter`**
  2. *Move Interface*: Das **`NumberConverterInterface`** verschieben ins Service-Package
  3. *Introduce Factory* auf dem Konstruktor von **`EditorNumberConverter`**
  4. Factory-Methode auf Lookup ändern
  5. *Inline Method*: Lookup
  6. *Move Class*: **`SAPHelper`** verschieben

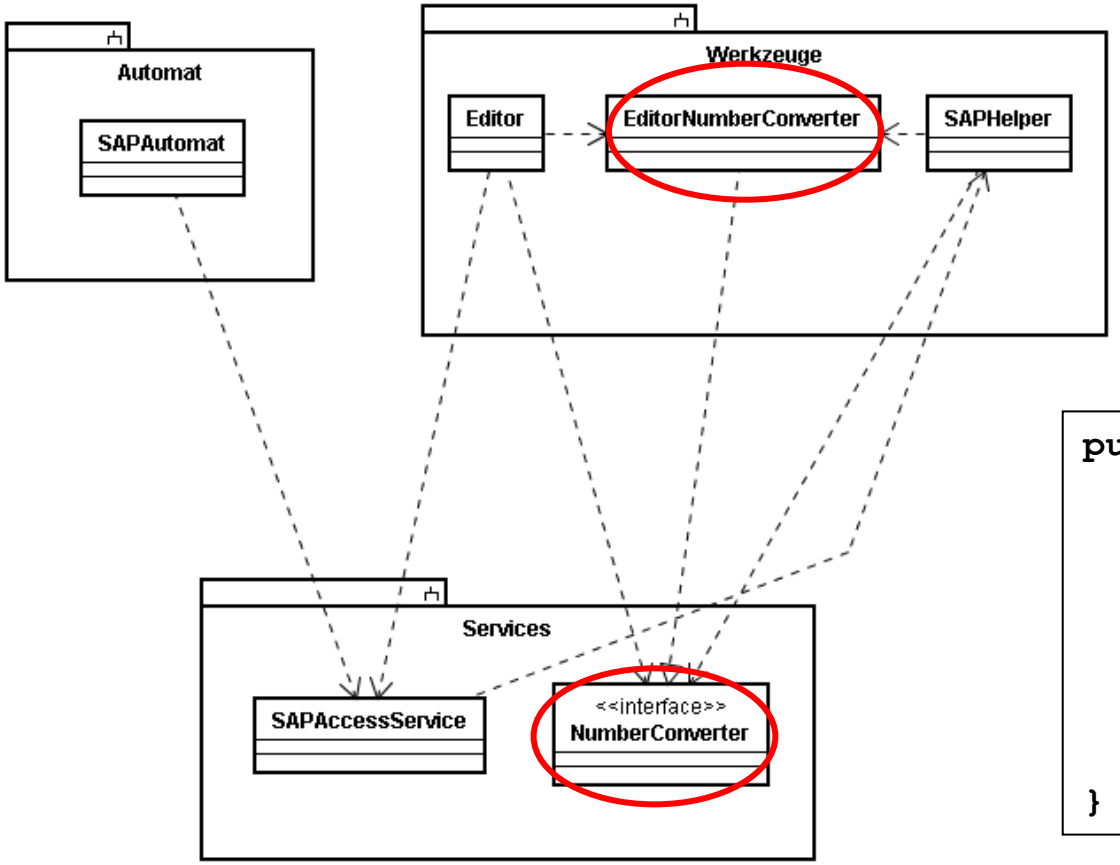
1. Extract Interface

- *Extract Interface* auf **`EditorNumberConverter`** ausführen.



2. Move Interface

- Beziehung zu der konkreten Implementationsklasse ist auch nach *Extract Interface* und *Move Interface* vorhanden.
- Struktur temporär schlechter als vor dem Refactoring!




```
public class SAPHelper {

  public SAPHelper()
  {
     _nc = new EditorNumberConverter();
  }

  NumberConverter _nc;
}
```

3. Introduce Factory

- Eine Factory-Methode hilft, das Erzeugen von der konkreten Erzeugungsstrategie zu entkoppeln.

```
public class EditorNumberConverter implements NumberConverter {

  public static NumberConverter createNumberConverter(){
    return new EditorNumberConverter();
  }

  private EditorNumberConverter(){
  }
}
```

```
public class SAPHelper {

  public SAPHelper()
  {
    _nc = EditorNumberConverter.createNumberConverter();
  }

  NumberConverter _nc;
}
```

4. Lookup-Erzeugungsstrategie

- Vollständige Entkopplung von der konkreten Implementationsklasse wird erreicht durch Verwendung eines Registry/Container-Lookups.
- Registries stellen beispielsweise das PicoContainer- oder Spring-Framework bereit. Konkrete Implementationen zu Interfaces können dort registriert werden.

```
public class EditorNumberConverter implements NumberConverter {

  public static NumberConverter createNumberConverter(){
    return (NumberConverter)Container.lookup(NumberConverter.class);
  }

  private EditorNumberConverter(){
  }
}
```

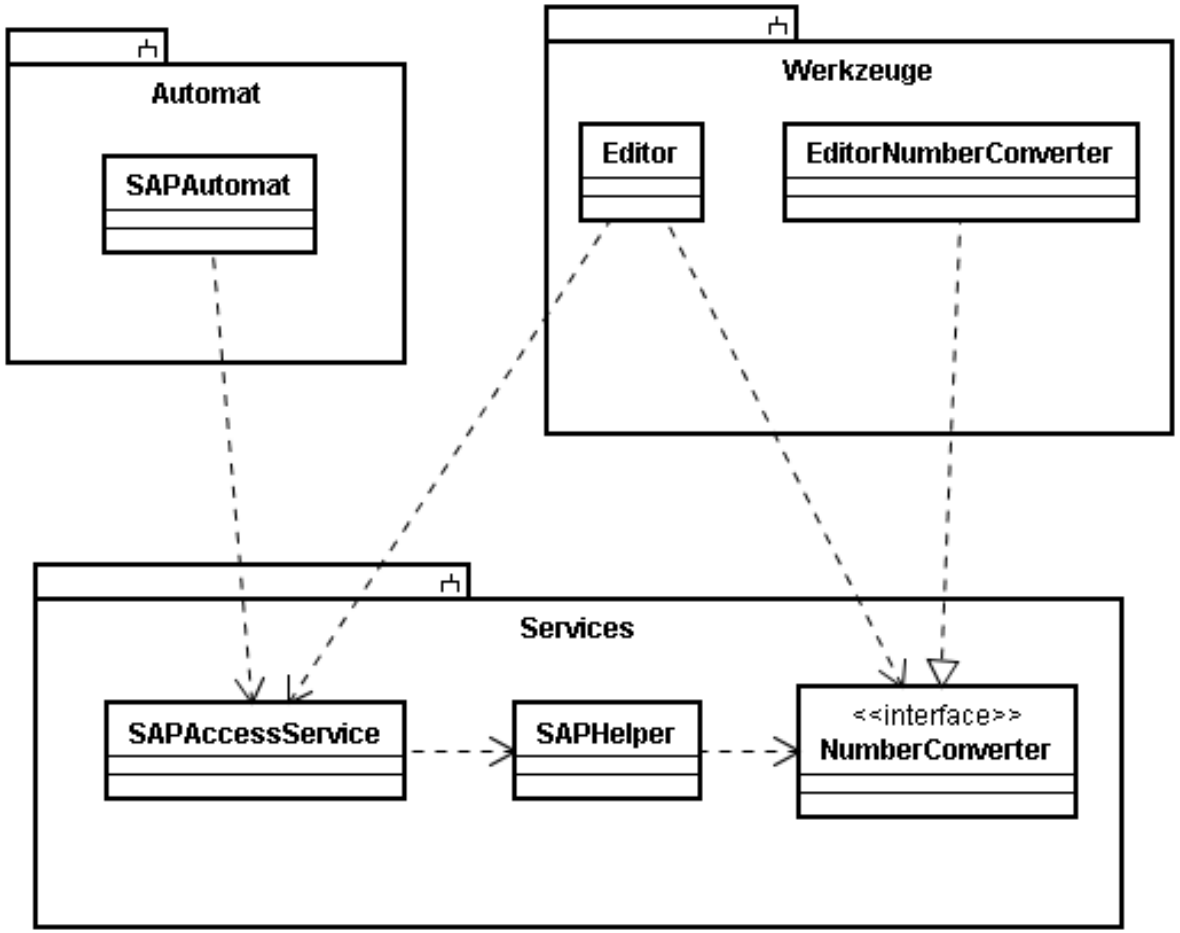
5. Inline Method

- Die Aufrufe an der konkreten Factory können jetzt ersetzt werden durch die entkoppelten Lookups.
- Durch Inline Method wird der Factory-Methoden-Aufruf durch den auf den Interfaces basierenden Lookup ersetzt.

```
public class SAPHelper {

  public SAPHelper()
  {
    _nc = (NumberConverter)Container.lookup(NumberConverter.class);
  }

  NumberConverter _nc;
}
```

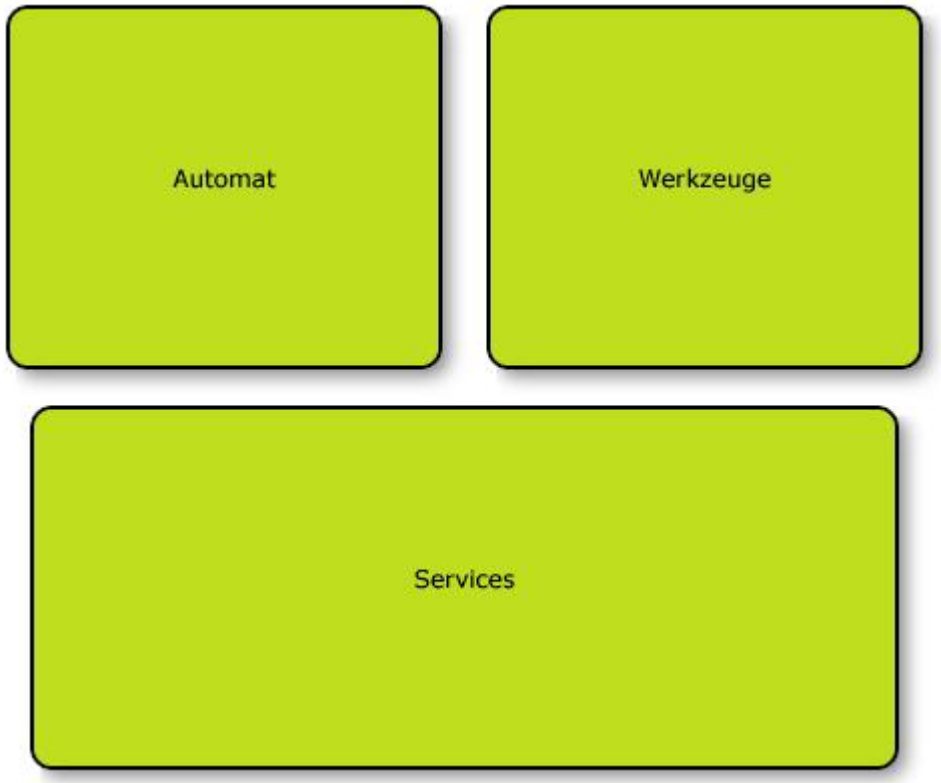
6. Move Class

- Abschließend kann die entkoppelte Klasse einfach verschoben werden.



Ergebniskontrolle

- XRadar zeigt, ob das Refactoring erfolgreich war.
- Bemerkenswert: Bis auf Schritt 4 (Lookup einführen) sind alle Schritte sicher automatisiert mit Eclipse durchführbar!



Fazit

- **Code zu refaktorisieren ist wichtiger als Code zu schreiben.**
  - **Oder: Refactoring is more important than coding.**
  - Wir verbringen viel mehr Zeit damit, bereits vorhandenem Code zu bearbeiten, als neuen Code zu implementieren.
- Nutzen Sie Refactoring-Tools intensiv!!!
- Refactorings sind nur mit Unit-Tests wirklich sicher durchführbar!
- Besonders wichtig:
  - **Refactorings dürfen nicht aufgeschoben werden!**
  - **Refactorings müssen im Team kommuniziert werden!**
  - **Fragen Sie die Experten! ;-)**

Ein wenig Werbung... ;-)

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

Vielen Dank

- Fragen und Feedback jederzeit gerne!

- Martin Lippert: martin.lippert@it-agile.de
- Andreas Havenstein: andreas.havenstein@it-agile.de