Software Engineering: Zusammenfassung

Christof Zellweger

23. August 2007

Inhaltsverzeichnis

1 Projekt Automation **2**
1.1 Warum,wann und was automatisieren? . . . 2
1.2 One Step Builds . . . 2
1.3 Build-Tools . . . 2
1.3.1 Ziele von Build-Tools . . . 2
1.3.2 make . . . 2
1.3.3 ant . . . 5

2 Refactoring **7**
2.1 Einführung . . . 7
2.2 Refactoring Zyklus . . . 8
2.3 Code Smells . . . 8
2.3.1 einige Smells innerhalb von Klassen . . . 9
2.4 Ergänzungen zu Refactoring to Patterns . . . 9
2.4.1 Replace Constructors with Creation Methods . . . 9
2.4.2 Extract Adapter . . . 10
2.4.3 Replace Hard-Coded Notifications with Observer . . . 10

3 Pragmatic Programming **11**
3.1 Was heisst pragmatisch? . . . 11
3.2 verschiedene Praktiken . . . 11

4 Test-Driven Development (TDD) **12**
4.1 TDD Zyklus . . . 12
4.2 Mock-Objekte . . . 13
4.2.1 Beispiel zu Mock: Self-Shunt . . . 13

5 Advanced Version Managment und Subversion **14**
5.1 CVS (concurrent versions system) versus SVN . . . 14

6 Ruby **15**
6.1 Beispiele . . . 15

7 Design By Contract **17**
7.1 Beispiel: Küchenmixer . . . 18

1 Projekt Automation

1.1 Warum,wann und was automatisieren?

- *warum?*
  - jeder Entwickler hat eigene IDE Installation
  - IDE-Versionenkonflikt
  - Wiederholbarkeit von Ergebnissen ist nicht garantiert
- *wann?*
  - vor Code schreiben, nach Aufsetzen des Repositories
- *was?*
  - Alles, was mehr als einmal gemacht werden muss!
  - also z.B. Compilieren, Codegenerierung, automatisierte Unit-Tests, JAR Generierung etc.

1.2 One Step Builds

- Builds sollten **CRISP** sein:
  - ***C****omplete*: Vollständige Builds werden von Grund auf neu erstellt und benutzen nur die im Build-Rezept angegebenen Bestandteile
  - ***R****epeatable*: sämtliche zur Erzeugung des Programm benötigten Dateien befinden sich im Repository, so, dass ständig jede Version durch auschecken der entsprechenden Dateien erneut erstellt werden kann; wiederholbare Builds sind also *konsistent*
  - ***I****nformative*: informative Builds sorgen für detaillierte Informationen, die die Ursache jedes Fehlers aufzeigen: eine fehlende Datei, die aber benötigt wird, eine Quelldatei, die sich nicht kompilieren lässt, oder ein fehlgeschlagener Test;
  - ***S****chedulable*: falls vollständig und wiederholbar; ein zeitgesteuerter Build kann zu einer bestimmten Uhrzeit (z.B. Mitternacht), in regelmäßigen Abständen (z.B. jede Stunde), zu einem Ereignis (wenn wir z.B. Quellcode einchecken) oder einer nach dem anderen stattfinden.
  - ***P****ortable*: sollte auf allen [Ziel-] Plattformen/Rechnern den Build-Prozess laufen lassen können, z.B. auf verschiednene UNIX-Systemen (heisst nicht zwangsläufig, dass eine UNIX-Anwendung auch auf Windows gebaut werden kann)

1.3 Build-Tools

1.3.1 Ziele von Build-Tools

- konsistente Generierung der Ergebnisse
- minimaler Generierungsaufwand
- unabhängig von manueller Intervention

1.3.2 make

- universelles und bewährtes Tool, „programmierung“ mittels C-Syntax
- bessere Lesbarkeit als ant

- Prinzipien von make sind:
  - Abhängigkeiten zwischen Dateien
  - Generierungsschritte, um Abhängige Datei(en) aus Quelle zu erstellen
  - vernünftiges Standardverhalten (kann beeinflusst werden)
  - Dateien werden anhand ihrer Namen-Sufixes klassifiziert

Elementare Makefiles

- enthalten:
  - Variablen, Targets, Dependents (Quellcodedateien, Abhängigkeiten), Rules/Tasks
- zeilenorierentiert: Leerzeilen trennen Targets
- Rules sind mit $< TAB >$ eingerückt
- Sytax:

```
Target: Dependents
    Task
```

Beispiel

```
#########################
//hello.h
//-----------------------
#ifndef Hello_H
#define Hello_H
extern void doHello();
#endif
#########################
```

```
#########################
//hello.cpp
//-----------------------
#include "hello.h"
#include "world.h"
#include <iostream>
void doHello(){
  std::cout << "Hello" << getWorld() << std::endl;
}
#########################
```

```
#########################
//world.h
//-----------------------
#ifndef World_H
#define World_H
#include <string>
extern std::string getWorld();
#define ANOTHERWORLD "Mars"
#endif
```

```
#########################
//world.cpp
//-----------------------
#include "world.h"
std::string getWorld(){
 return ANOTHERWORLD;
}
#########################
```

```
#########################
//main.cpp
//-----------------------
#include "hello.h"
#include "world.h"
#include <iostream>
int main(){
```

```
doHello();
std::cout << "getWorld() delivers: " << getWorld() << std::endl;
std::cout << "ANOTHERWORLD is: " << ANOTHERWORLD << std::endl;
}
```

dazugehöriges makefile

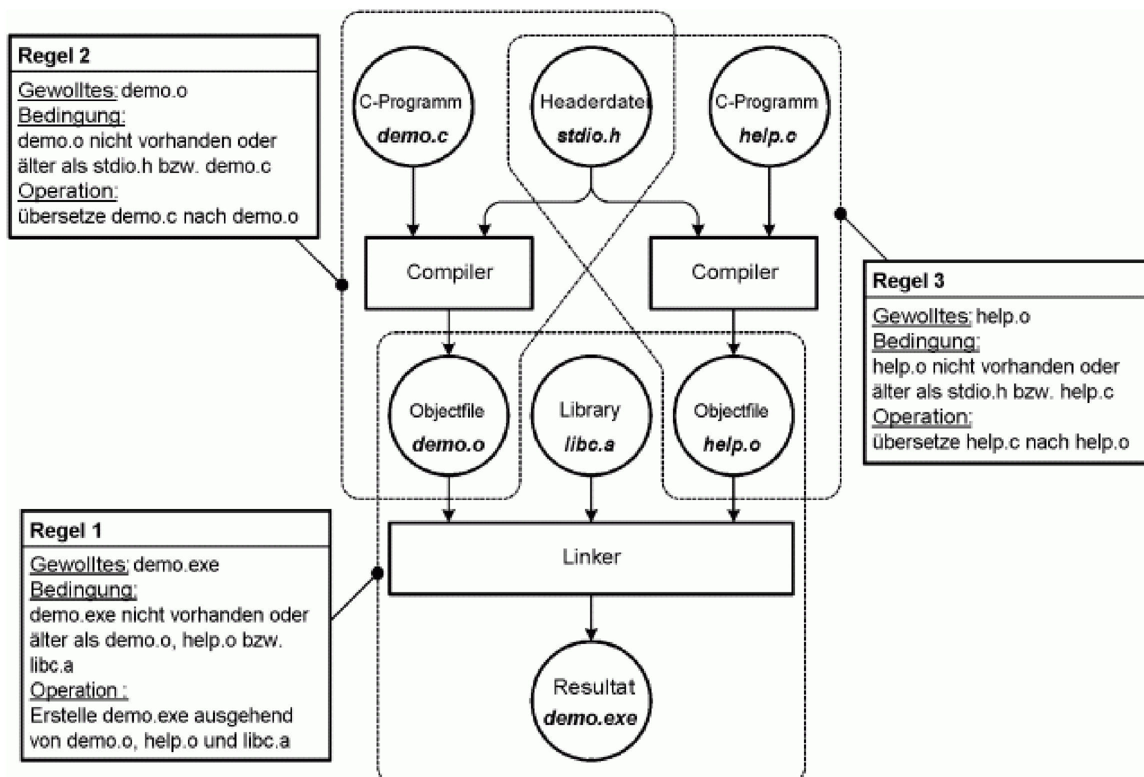
```
INC = hello.h world.h
SRC = main.cpp hello.cpp world.cpp
OBJ = main.o hello.o world.o
helloworld: $(OBJ)
	$(LINK.cc) -o $@ $^

clean:
	$(RM) hello.o main.o world.o helloworld.exe

depend: $(SRC) $(INC)
	makedepend $^
	// $^ bezeichnet alle Dateien, die rechts von
	// target-name (depend) stehen
#DO NOT DELETE

main.o: hello.h world.h
hello.o: hello.h world.h
world.o: world.h
```

Abhängigkeitsbaum bei make



Dazugehöriges makefile könnte wie folgt aussehen:

```
#Regel 1
demo.exe : demo.o help.o libc.a
    cc -o demo.exe demo.o help.o

#Regel 2
demo.o : demo.c stdio.h
    cc -c demo.c

#Regel 3
help.o : help.c stdio.h
    cc -c help.c
```

1.3.3 ant

Fragen

- **Q**: Wie organisieren Sie Abhängigkeiten von „targets“?
- **A**: mittels `depends`-Keyword: <target name=“`name`“ **depends**=“`anotherTarget`“>

- **Q**: Wie wählen Sie ein bestimmtes Target beim Aufruf von ant aus?
- **A**: mittels `ant -f <xml-File> <target>`

Attribute für `java`-Task in ant:

Attribute	Description	Required
classname	the Java class to execute.	Either `jar` or `classname`
jar	the location of the jar file to execute (must have a Main-Class entry in the manifest). Fork must be set to true if this option is selected. See notes below for more details.	Either `jar` or `classname`
args	the arguments for the class that is executed. deprecated, use nested ¡arg¿ elements instead.	No
classpath	the classpath to use	No
classpathref	the classpath to use, given as reference to a PATH defined elsewhere	No
fork	if enabled triggers the class execution in another VM (disabled by default)	No

Beispiel:

Listing 1: Ant-Script

```
<project default="run">
        //Zugriff mit ${src}//irgendwas
  <property name="src" value="src" />
  //Zugriff mit ${build}/irgendwas
  <property name="build" value="build" />

  <target name="compile">
```

```
8      <javac srcdir="." />
9    </target>
10
11   <target name="jar" depends="compile">
12     <jar destfile="hello.jar"
13          basedir="."
14          //alle class-Files, in einem Unterordner von basedi
15          includes="**/*.class"
16          />
17   </target>
18
19   <target name="run" depends="jar">
20     <java classname="hello"
21           classpath="hello.jar"
22           fork="true"
23           />
24   </target>
25
26   <target name="mail" depends="run">
27         <mail from="test1@hsr.ch"
28               replyto="test1@hsr.ch"
29               tolist="test3@hsr.ch"
30               message="mail aus anttask"
31               messagemimetype="multipart/mixed"
32               mailhost="smtp.hsr.ch"
33               files="hello.java"/>
34   </target>
35 </project>
```

- wird mit XML geschrieben
- Elements:
  - *project*: Root Element, hat 3 Attribute (name, default, basedir)
    - *name*: Name des Projekts, nicht zwingend anzugeben
    - *default*: default Target, welches ausgeführt wird, sofern kein anderen angegeben wird
    - *basedir*: Directory, von welchem aus alle relativen Pfade im Ant-Buildfile referenziert werden; wird es nicht angegeben, wird parent-directory von build-file angenommen
  - *property*: ermöglicht Deklaration von benutzerdefinierten Variablen, welche im Build File verwendet werden können; „name“ gibt den Namen an, „value“ den gewünschten Wert; Referenzierung erfolgt mittels `${propertyName}`
  - *target*: wird als Wrapper für verschiedene Aktionen benutzt; muss Namen haben, damit es refernziert werden kann; innerhalb von einem *target*-Block wird beispielsweise `javac` ausgeführt, welcher die java-Files kompiliert

Zusammenfassung

- Automatisierung im Entwicklungsprozess schont die Nerven

  - Flüchtigkeitsfehler werden entdeckt, bevor sie lästig werden
  - keine Produktivitätseinbrüche durch unnötige Fehlersuche

- Alle müssen am gleichen Strang ziehen
  - zentraler Build Server
  - keine Broken Builds über längere Zeit (sonst Broken Window Effekt!)

- Immer wieder Überprüfen und Verbessern

2 Refactoring

2.1 Einführung

- *Refactoring*: Prozess, die innere Struktur eines Softwaresystems zu verbessern, ohne dass sich das von aussen beobachtbare Verhalten ändert

- Refactoring: (Substantiv) ist ein definierter Veränderungsschritt im Refactoring-Prozess

- Refactorings besitzen *Namen*

- damit es überhaupt zum Refactoring kommt, müssen zuerst die *Probleme erkannt werden*, welche Design des Codes schlecht machen, d.h. Software Engineering Prinzipien sind verletzt (wie z.B. Low Coupling, High Cohesion, allg. GRASP-Patterns): hierzu werden sogenannte *Code Smells* als Hilfsmittel eingesetzt (siehe „Code Smells“)

- Liste von Refactorings auf: `www.refactoring.com/catalog/`

Refactoring	Inverses Refactoring
Add Parameter	Remove Parameter
Change Bidirectional Association to Unidirectional	Change Unidirectional Association to Bidirectional
Change Reference to Value	Change Value to Reference
Change Unidirectional Association to Bidirectional	Change Bidirectional Association to Unidirectional
Change Value to Reference	Change Reference to Value
Collapse Hierarchy	Extract Subclass
Extract Class	Inline Class
Extract Method	Inline Method
Extract Subclass	Collapse Hierarchy
Hide Delegate	Remove Middle Man
Inline Class	Extract Class
Inline Method	Extract Method
Inline Temp	Introduce Explaining Variable
Introduce Explaining Variable	Inline Temp
Move Field	Move Field
Move Method	Move Method
Parameterize Method	Replace Parameter with Explicit Methods
Pull Up Field	Push Down Field
Pull Up Method	Push Down Method
Push Down Field	Pull Up Field
Push Down Method	Pull Up Method
Remove Middle Man	Hide Delegate
Remove Parameter	Add Parameter
Rename Method	Rename Method
Replace Delegation with Inheritance	Replace Inheritance with Delegation
Replace Parameter with Explicit Methods	Parameterize Method
Substitute Algorithm	Substitute Algorithm

- **Zielhierachie beim Refactoring**
  1. alle Tests laufen fehlerfrei

2. kein duplizierter Code, keine duplizierte Logik
3. Code enhält alle wichtigen Absichten
4. System hat nur die absolut nötigsten Klassen und Methoden implementiert

- wird auch **OOAO** genannt: *Once and only once*!

2.2 Refactoring Zyklus

- Voraussetzung: lauffähiges Programm, Automatische Unit Tests sind vorhanden
  1. Problem erkennen: GRASP-Patterns verletzt, Smells, Prinzipien von SE nicht eingehalten
  2. Refactoring wählen
  3. Refactoring in kleinsten Schritten durchführen: nur kleine Schritte wählen, damit Risikon nicht so gross; geht etwas schief, muss nicht viel Rückgängig gemacht werden, man vergisst die einzelnen Schritte nicht so leicht
  4. nach jedem Schritt testen, d.h. GreenBar muss immer wieder vorhanden sein

- gutes Beispiel zum Prozess in den Folien „Refactoring“, ab Folie 10

2.3 Code Smells

- *Code Smell*: Konstrukt, das auf ein Problem hinweist und eine Umarbeitung des Programm-Quelltextes nahelegt; muss nicht zwingend ein Fehler sein; oft als „unschönes Design / Programmieren“ wahrgenommen

- häufige Smells sind:
  - Large method: Methode ist zu gross, zu schnell gewachsen
  - Large Class: Klasse ist zu gross und somit unübersichtlich; geht oft Hand in Hand mit der Verletzung des Prinzips der Kohäsion
  - Feature envy: Klasse, die starken Gebrauch von Methoden von anderer Klasse macht
  - Inappropriate intimacy: Klasse, die von Implementationsdetails von anderer Klasse abhängig ist
  - Refused bequest: Klasse überschreibt eine Basismethode einer Basisklasse, wodurch allerdings der Vertrag (Contract) der Basisklasse verletzt wird (*Liskov Subtitution Principle*: sagt aus, wann ein Datentyp als Unterklasse eines anderen Typs modelliert werden soll, beispielsweise wäre das Prinzip verletzt, wenn man ein Quadrat als Unterklasse eines Rechtecks machen würde, da die Aussage „Breite und Länge können unabhängig voneinander skaliert werden“ zwar für das Rechteck aber nicht für das Quadrat gilt; des weiteren wäre die is-a-Beziehung, die bei Vererbung gilt, nicht erfüllt)
  - Lazy class: Klasse, die nicht genug „macht“ um ihre Existenz zu rechtfertigen
  - Duplicated method: Methode, welche sehr ähnlich wie eine andere Methode ist oder sogar das gleiche macht (womöglich auf eine etwas andere Art)
  - Contrived Complexity: Benutzung von komplizierten Design Pattern, wo man ein Problem auch einfacher lösen könnte (*strive for simplicity*, strebe nach Einfachheit)

2.3.1 einige Smells innerhalb von Klassen

Smell	Refactorings
Comments	Extract Method, Rename Method
Long Method	Extract Method, Replace Temp with Query, Introduce Parameter Object, Replace Method with Method Object
Long parameter List	Replace Parameter with Method, Parameter Object
Duplicated Code	Extract Method, Pull up field, Form Template Method, Substitute Algorithm
Large Class	Extract Class, Extract Subclass
Type Embedded in Name	Rename Method
Uncommunicative Name	Rename Method
Inconsistent names	Rename Method
Dead Code	Delete the code
Speculative Generality	

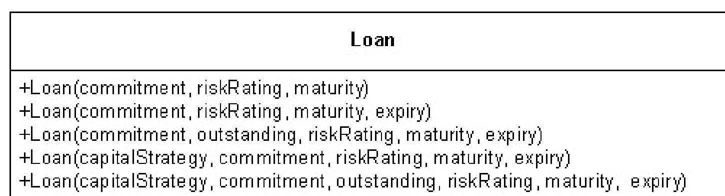
- weitere Smells zwischen Klassen etc. sind vorhanden: siehe Folien Code Smells
- siehe Folien für genauere Beschreibung der einzelnen Smells
- zu empfehlen: Zusammenfassung „Smells“ von

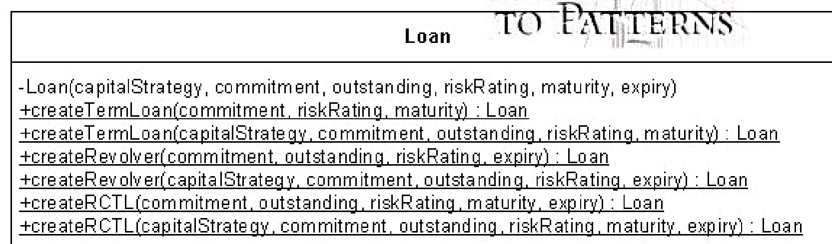
2.4 Ergänzungen zu Refactoring to Patterns

Für eine ausführlichere Zusammenstellung von Refactorings to Patterns, siehe „Smells to Refactorings - Quick Reference Guide“ auf studentenportal.

2.4.1 Replace Constructors with Creation Methods

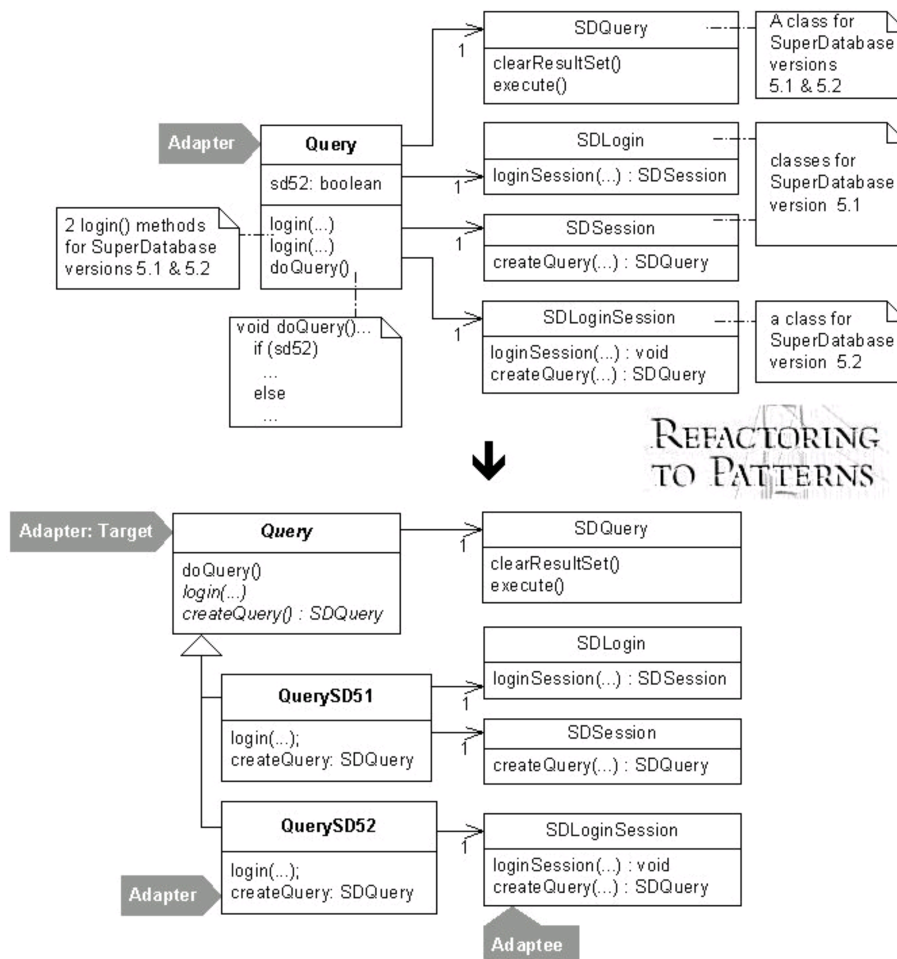
- Constructors on a class make it hard to decide which constructor to call during development.
- *„Replace the constructors with intention-revealing Creation Methods that return object instances.“*





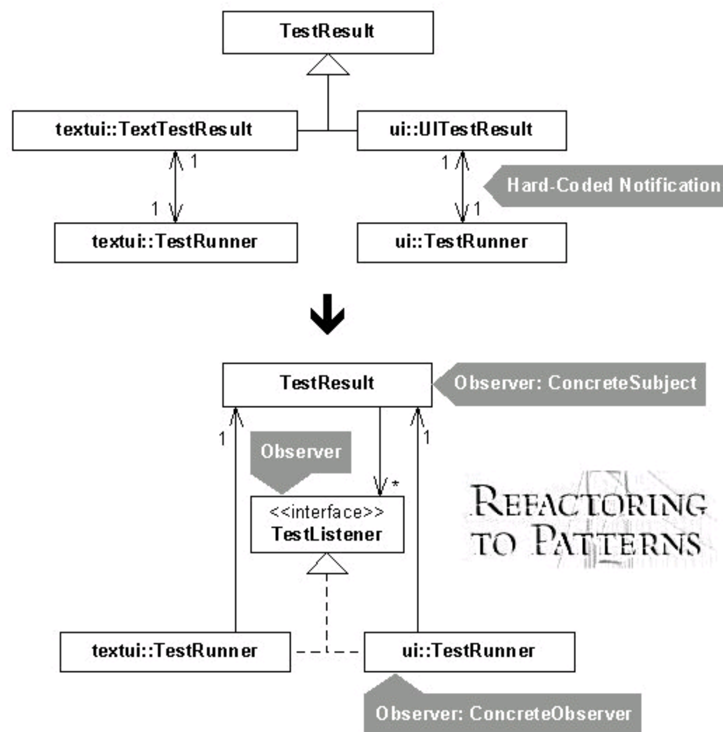
2.4.2 Extract Adapter

- One class adapts multiple versions of a component, library, API or other entity.
- Ein Adapter hat das Problem, dass er spezifische Funktionalität einer Library-Version vielleicht nicht zur Verfügung stellt.
- *„Extract an Adapter for a single version of the component, library, API or other entity.„*



2.4.3 Replace Hard-Coded Notifications with Observer

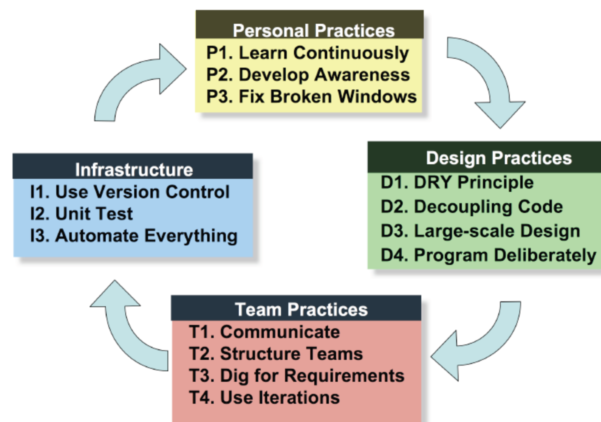
- Ein Objekt benachrichtigt ein anderes Objekt durch einen Methodenaufruf. Nun sollen weitere Objekte benachrichtigt werden. Welches Refactoring könnte hier interessant sein?
- *„Remove the subclasses by making their superclass capable of notifying one or more instances of any class that implements an Observer interface.“*



3 Pragmatic Programming

3.1 Was heisst pragmatisch?

- Das tun, was funktioniert: doing what works
- An dem arbeiten, woran man soll: working at what to do



3.2 verschiedene Praktiken

- *kontinuierliches Lernen*: Investieren in Wissen und Fähigkeitenerwerb; ständige Weiterbildung; Lernen, was funktioniert; Anwendungen üben
- *Fix Broken Windows*: kleine Probleme sofort beheben, bevor sie zu grossen Problemen werden; SPÄTER KOMMT NIE!

- *Don't repeat yourself (DRY)* Prinzip: jeder Wissensbestandteil muss eine einzige, eindeutig anerkannte, massgebliche Repräsentation im System haben; statt Copy-Paste → gemeinsame Funktionen einsetzen; externe Konfigurationsdaten (Property Files)
- *Entkopplung des Codes*: Orthogonalität (unabhängige Variation konzeptuell unabhängiger Aspekte, setzt voraus, dass Teile separiert werden, welche nicht zusammengehören); Low Coupling: einfacher zu ändern, einfacher zu verstehen, einfacher zu testen; Code sollte immer *nur aus genau einem Grund* geändert werden; Kohäsion erhöhen
- *Kopplung im Grossen*: darauf achten, dass Architektur die Kopplung möglichst gering hält; Kopplung ist *transitiv*, d.h. es kann leicht zu Abhängikeitszyklen kommen; zirkuläre Abhängikeiten möglichst vermeiden, da Komponenten dann nicht mehr *einzeln* getestet, ausgetauscht oder repaiert werden können; womgölicht eine Fassade (Facade) einsetzen
- *bewusstes Programmieren*:
  - Plan haben, was man erreichen will
  - sich auf wesentliche Dinge konzentrieren
  - nur auf verlässliche Dinge verlassen, welche man versteht
  - Annahmen stets dokumentieren
- *Infrastruktur*: Einsatz von bewährten Methoden wie Versionsmanagment (CVS, SVN), automatisiertes Unit-Testing, möglichs alles automatisieren, was möglich ist
- *Team Praktiken*
  - Kommunizieren: viel, offen, effizient und direkt mit Teamkollegen kommunizieren; sich klar und deutlich ausdrücken; Wiki Wwebs einsetzen
  - Teams strukturieren: eine unnötigen Schichten einbauen; auf Kommunikation zwischen Mitgliedern achten; sollte Teams so organiseren, wie man Software organisieren will
  - Dig for requirements: Requirements ändern sich ständig → Design for Change; Missverständnisse zwischen Kunden und Lieferant durch klare Kommunikation ausmerzen

4 Test-Driven Development (TDD)

- **TDD**: zuerst Testcase schreiben und dann den Code, welcher nur genau diesen Testcase erfüllt
- gibt klares und unmittelbares Feedback
- wird nicht als Test-Methode angeschaut sondern als *Methode, um Software zu designen*

4.1 TDD Zyklus

1. *Test schreiben / hinzufügen*: sobald ein neues Feature hinzugefügt wird, muss als Erstes ein Test hierfür geschrieben werden; Programmierer muss Requirements/Spezifikationen des Features genau kennen (Use Cases), sonst kann er keinen guten Test dazu schreiben; dieser Test muss fehlschlagen, weil ja das Feature noch gar nicht implementiert ist
2. *Alle Tests laufen lassen, neu hinzugefügter muss fehlschlagen*: dies garantiert, dass jeder Test auch eine zu ihm passende Implementation in Form des neuen Features benötigt

3. *schreibe Code/Feature*: nun kann das Feature implementiert werden; **WICHTIG**: Code sollte nur genau den Test bestehen, nicht schon sehr elegant und möglichst raffiniert sein, dies kommt in späteren Schritten

4. lasse die automatisierten Tests laufen und kontrollieren, ob alles korrekt abläuft

5. *Refactoring des Codes*: verbessere nun den Feature-Code und den Test-Code, bis er den Anforderungen an Design und bewährten Praktiken genügt

4.2 Mock-Objekte

- Soll die Interaktion eines Objektes mit seiner Umgebung überprüft werden, muss vor dem eigentlichen Test die Umgebung nachgebildet werden
- Mock-Objekte implementieren die Schnittstelle, über die das zu testende Objekt auf seine Umgebung zugreift
- Sie stellen sicher, dass die erwarteten Methodenaufrufe vollständig, mit den korrekten Parametern und in der erwarteten Reihenfolge durchgeführt werden und geben vordefinierte Ergebnisse an das zu testende Objekt zurück
- sollten eingesetzt werden, wenn „echte“ Objekte:
  - nicht deterministische Ergebnisse liefert
  - Schwierigkeiten bei der Vorbereitung oder während der Ausführung bereitet (Benutzeroberflächentest)
  - Verhalten zeigen soll, das nur schwer auszulösen ist (z. B. einen Netzwerkfehler)
  - langsam ist (z. B. eine vollständige Datenbank, die vor dem Test erst initialisiert werden müsste)
  - noch nicht existieren
  - Informationen und Methoden ausschließlich zu Testzwecken (und nicht für seine eigentliche Aufgabe) zur Verfügung stellen müsste

4.2.1 Beispiel zu Mock: Self-Shunt

ursprüngliche Klasse:

```
1  public class ScannerTest extends TestCase{
2    public ScannerTest (String name) {
3      super (name);
4    }
5
6    public void testScanAndDisplay (){
7      Scanner scanner = new Scanner();
8      Display display = new Display();
9      Item item = scanner.scan ();
10     display.displayItem (item);
11     //and now, what assert???? wie testen?
12   }
13 }
```

als Self-Shunt implementiert:

```java
import junit.framework.TestCase;

// Verhalten wie ein Display
public class ScannerTest extends TestCase implements Display{
  public ScannerTest (String name) {
    super (name);
  }
  public void testScan () {
  // gebe dich selbst als als Instanz von Display mit
    Scanner scanner = new Scanner(this);
  // scan ruft displayItem auf seinem display-Object auf,
  // was vorhin übergeben wurde
    scanner.scan();
    assertEquals (new String("Cornflakes"), lastItem);
  }
  // Implementation von Display.displayItem ()
  public void displayItem (String item) {
    lastItem = item;
  }
  private String lastItem;
}
```

5 Advanced Version Managment und Subversion

5.1 CVS (concurrent versions system) versus SVN

- Das Versionsschema von Subversion bezieht sich *nicht mehr auf einzelne Dateien wie bei CVS*, sondern auf das ganze Repository

- Subversion speichert beim `Checkout`, `Update` und `Commit` in einem gesonderten Verzeichnis (.svn) Client-seitig eine zweite Kopie jeder Datei. Dadurch verdoppelt sich der Speicherbedarf einer Arbeitskopie, allerdings bietet dies bei entfernten Repositorys auch einige Vorteile. So können einige Aktionen, wie Anzeige der lokalen Änderungen, komplett ohne Netzwerkzugriff erfolgen und Subversion muss beim Commit nur die geänderten Teile einer Datei übertragen, während CVS die Änderungen Server-seitig berechnet und somit jeweils die gesamte Datei übertragen muss

- Während `Tags` und `Branches` in CVS eine klare semantische Bedeutung haben, kennt Subversion nur das *Konzept der Kopie*, die je nach Nutzungsart Tag- oder Branch-Charakter haben kann

- Jede Kopie in Subversion ist demnach automatisch ein Branch dieser Datei oder des Verzeichnisses. Tags entstehen in Subversion, wenn man eine Kopie anlegt und später auf ihr keine Änderungen mehr vornimmt. Aufgrund des Fehlens einer Tag- und Branch-Semantik obliegt die Strukturierung und Verwaltung von Tags und Branches dem Benutzer und Administrator

- Dabei hat sich bewährt, für Projekte die Basisverzeichnisse *trunk* (engl. Stamm), *branches* (engl. Verzweigungen) und *tags* (engl. Markierungen) anzulegen. Ein *trunk* enthält dabei

den Letztstand des Projekts, in *branches* werden weitere Unterverzeichnisse mit alternativen Entwicklungspfaden verwaltet und in *tags* wird eine Kopie vom *trunk* oder einem der *branches* als Unterverzeichnis angelegt

- Nachteile von CVS, die durch SVN behoben wurden:
  - Dateien können nicht verschoben werden. Sie müssen gelöscht und neu angelegt werden. Die History geht verloren.
  - Keine atomaren Commits. Führen zwei Entwickler gleichzeitig ein Commit aus, kann es zu Synchronisations-Problemen kommen.
  - Es wird zwischen binären und Textdateien unterschieden.
  - Verzeichnisse können nicht „für immer“ gelöscht werden.

6 Ruby

→ siehe Foo's „Ruby condensed“ auf studentenportal. Fettes Dankeschön an ihn!

6.1 Beispiele

Loops

```
10.times{puts "hello"}
(1..10).each{|ind| puts ind.to_s + " hello"}
```

Datum

```
s = "Mon May 22 29:22:01 2006"
if s =~ /^(Mon|Tue|Wed|Thu|Fri|Sat|Sun) \
(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec) \
[0-3][0-9] [0-2][0-9](:[0-5][0-9]){1,2}\s+[0-9]{4,4}/
  puts s + " is a date "
else
  puts s + " is not a valid date"
end
```

File

```
File.open("inhalt.txt"){|file|
   file.each { |l| print l.gsub(/\r\n/, ' ') }
}
```

```
File.read("liste.cpp").each_line{ |l|
    if(l =~ /^\s*#include\s+[<"]([^>"]+)[">]/)
      list << $1 + ".h"
    end
}
list.sort.each{|ind| puts ind}
```

```
File.open(ARGV[0]) do |file|
  print file.read.gsub(/\n/," ")
end
end{lstlisting}

\begin{lstlisting}
if ( s =~ /^(1[012]|[1-9]):([0-5]\d)\s*(am|pm)$/i )
 hour,min,pm=$1.to_i,$2.to_i, $3 == "pm"
 hour += 12 if pm && hour < 12
 hour -= 12 if !pm && hour >= 12
```

```
IO.readlines("inhalt.txt").reverse!.each{|ent| puts ent}
```

```
filecontent = IO.read(ARGV[0])
@filecontent.scan(/\w+/).each{|word|
      #word.downcase!
      @wordHash[word] = @wordHash[word]+1
}
@stat = @wordHash.sort {|a,b| a[1]<=>b[1]}.reverse!.slice!(0..@limit)
```

römische Zahlen

Listing 2: beachte die Reihenfolge der romanlist, essentiell wichtig!

```
class Integer
    @@romanlist = [["M", 1000],["CM", 900],
                   ["D",  500],["CD", 400],
                   ["C",  100],["XC",  90],
                   ["L",   50],["XL",  40],
                   ["X",   10],["IX",   9],
                   ["V",    5],["IV",   4],
                   ["I",    1]]

    def to_roman(zahl)
        remains = zahl
        roman = ""
        for sym, num in @@romanlist
            while remains >= num
                remains -= num
                roman << sym
            end
        end
        roman
    end
```

Regex

```
s=gets()
if s =~ /[-+]?((\d+\.\d*|\.\d+)(e[-+]?\d{1,3})?)|(\d+e[-+]?\d{1,3})/
  puts s + " is a floating number "
else
  puts s + " is not a valid floating number"
end
```

```
#filtern von HTML-Tags
/<((\/[\w-]+)|([\w-]+(\s+[\w_]+="[^"]+")*\/?))>/
```

```
pattern.match(string) #true/false
string.sub(pattern, replacement) #erstes vorkommen
string.gsub(pattern, replacement) #alle vorkommen
#In the block form, the current match string is passed in
#as a parameter, and variables such as $1, $2, $`, $&, and
#$' will be set appropriately.
str.gsub(pattern) {|match| block }
variables such as $1, $2, $`, $&, and $' will be set appropriately.
```

```
#matching words
/\b([A-Za-z]+)\b/
```

```
def regmatch (r,str)
 if r =~ str then
   puts "#{$`} > >#{$&} < <#{$'}"
 end
end
regmatch (/ab/," abba ")# -> >>ab <<ba
```

```
# Make a word frequency count
seen = Hash.new(0)
while (gets)
    gsub(/(\w[\w'-]*)/) { |word|
        seen[word.downcase] += 1
    }
end
# output hash in a descending numeric sort of its values
seen.sort { |a,b| b[1] <=> a[1] }.each do |k,v|
    printf("%5d %s\n", v, k )
end
```

7 Design By Contract

- Contracts für Systemoperationen: betrachtete Komponente ist System; Teil der Domain-analyse
  - **Non-redundancy principle**: *Under no circumstances shall the body of a routine ever test for the routinge's precondition.*

  - **Preconditions**: was gilt *vor Ausführung* der Systemoperation?
  - **Postconditions**: was gilt *nach Ausführung* der Systemoperation?
- Contract für Klasse:
  - **Precondition**: Bedingung, die *vor Aufruf* erfüllt sein muss; Verantwortlichkeit liegt beim Aufrufer; beispielsweise muss bei Methode `calcSquare(double x)` garantiert sein, dass $x >= 0$ gilt
  - **Postcondition**: Bedingung, die *nach Aufruf* erfüllt sein muss; Verantwortlichkeit ist Implementation der Methode
- **Klasseninvarianten**: Garantien für Aufrufer, die immer, d.h. vor und nach Methodenaufrufen gelten

7.1 Beispiel: Küchenmixer

Listing 3: Verwendung von icontract

```
/**
* @invariant getSpeed() > 0 implies isFull() // Don't run empty
* @invariant getSpeed() >= 0 && getSpeed() < 10 // Range check
*/
public interface KitchenBlender {
 /**
 * Return the speed, 0 (off) through 10
 */
 public int getSpeed();

 /**
 * @pre Math.abs(getSpeed() - x) <= 1 // Only change by one
 * @pre x >= 0 && x < 10 // Range check
 * @post getSpeed() == x // Honor requested speed
 */
 public void setSpeed(final int x);

 public boolean isFull();

 /**
 * @pre !isFull() // Don't fill it twice
 * @pre getSpeed()== 0 // fill only when stopped
 * @post isFull() // Ensure it was done
 */
 public void fill();

 /**
 * @pre isFull() // Don't empty it twice
 * @pre getSpeed()== 0 //empty only when stopped
 * @post !isFull() // Ensure it was done
 */
 public void empty();

}
```