

LEGACY CODE

Arbeit und Umgang mit Legacy Code

Maurice Müller

2025-03-17

Grundlagen

Was ist Legacy Code?

- *alter* Code, der übernommen wurde
- schwer wartbarer Code
- schwer änderbarer Code
- ...

Code ohne Tests

(und damit schwer wartbar und schwer änderbar)

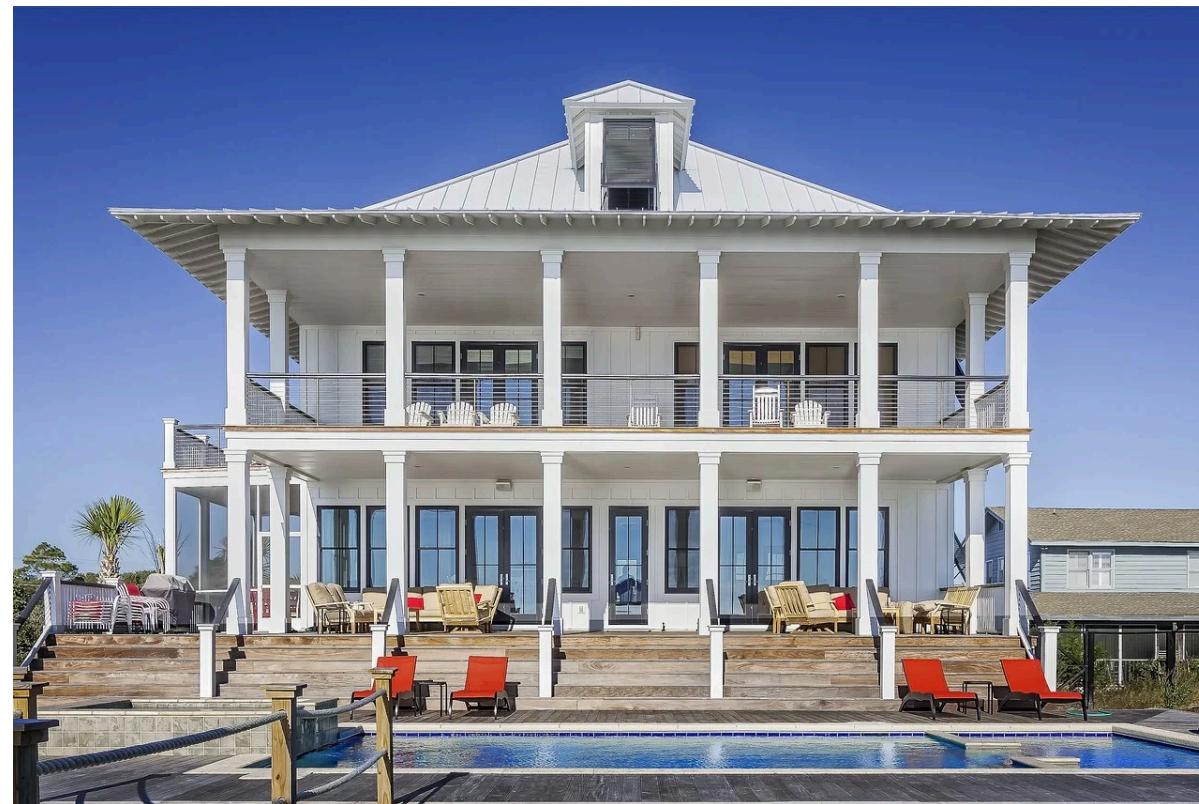
Als Antwort auf M. Feathers Frage, wie sich das neue Team macht:

| *They are writing legacy code, man.*

**Hört auf Legacy Code zu schreiben.
Und schreibt Tests.**

Warum ist das überhaupt wichtig?

- normal 8h / Tag Arbeit
- Arbeit = hauptsächlich Code lesen und schreiben
- Arbeit macht in einem sauberen Umfeld mehr Spaß



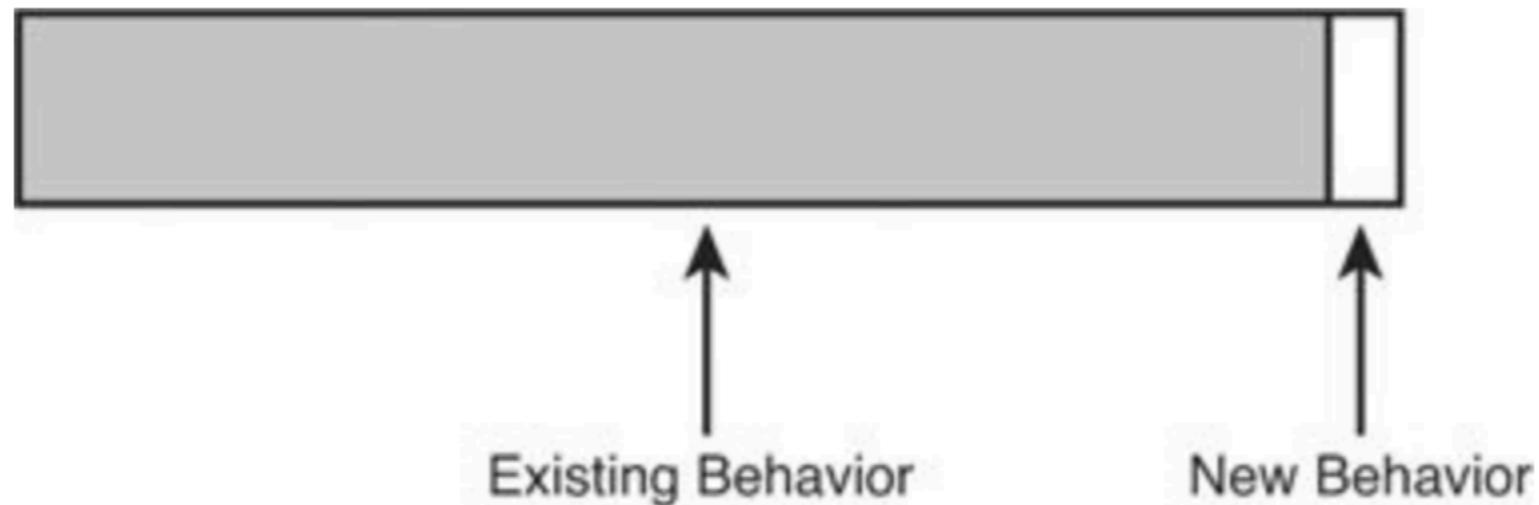




Warum ändert sich Software?

- Funktionalität hinzufügen (New Feature)
- Fehler beheben (Bug Fix)
- Struktur verbessern (Refactoring)
- Ressourcen optimieren (Optimizing)

Alte Funktionalität muss erhalten bleiben



Quelle: Working Effectively With Legacy Code von M.C. Feathers

- häufig wird sich nur auf die neue Funktionalität konzentriert
- dabei ist der alte Teil viel größer

Wichtige Fragen bei Änderungen

- Was für Änderungen müssen gemacht werden?
- Wie kann man feststellen, dass die Änderungen korrekt durchgeführt wurden?
- Wie kann man feststellen, dass Funktionierendes immer noch funktioniert?

Tests!

Realität

"Wenn es nicht kaputt ist, ändere es nicht."

"What? Create another method for that? No, I'll just put the lines of code right here in the method, where I can see them and the rest of the code. It involves less editing, and it's safer."

- mehr Leute einstellen
- Edit & Pray

Auswirkungen beim Vermeiden von Änderungen

- keine neuen Klassen, keine neuen Methoden
 - die alten Dinge werden riesig und damit unverständlich
- man rostet und verlernt, wie man Dinge leicht und schnell ändert
- Angst vor Änderungen wächst stetig

⇒ Code degeneriert immer mehr

Test Harness

Testabsicherung



Quelle: <https://www.space-figuren.de/>

Grundsätzlich 2 Arten zu arbeiten:

- **Edit and Pray / Bearbeiten und Beten**
 - "Mit besonderer Vorsicht arbeiten"
 - Vorsicht != sicheres Arbeiten
- **Cover and Modify / Absichern und Ändern**
 - mit Sicherheitsnetz arbeiten

Regressionstest

Ein Test, der durch ständig wiederholte Ausführung sicherstellt, dass Modifikationen keine neuen Fehler verursachen.

- sollte möglichst klein sein
 - besser verständlich
 - schnellere Fehlersuche möglich
- sollte möglichst schnell sein
 - sonst führt ihn niemand aus
 - Beispiel: 100ms sind deutlich zu lang → bei 10.000 Tests (1000 Klassen á 10 Tests) = >16min
- sollte möglichst isoliert ablaufen
 - keine Abhängigkeiten zur DB, Netzwerk, ...

Problem: Abhängigkeiten

1. vor Änderungen sollte man Tests schreiben
2. um Tests zu schreiben, müssen Abhängigkeiten aufgelöst werden
3. zur Auflösung muss man Änderungen machen
4. vor Änderungen sollte man Tests schreiben

Beispiele von Abhängigkeiten

- Objekte, die Übergeben werden
 - besonders problematisch: große Objekte / Gott-Klassen
- Schnittstellen zu externen Systemen
 - z.B. DB, Netzwerk, Dateisystem, ...
- Zustand, der vorhanden sein muss
 - z.B. initialisierte globale Variablen

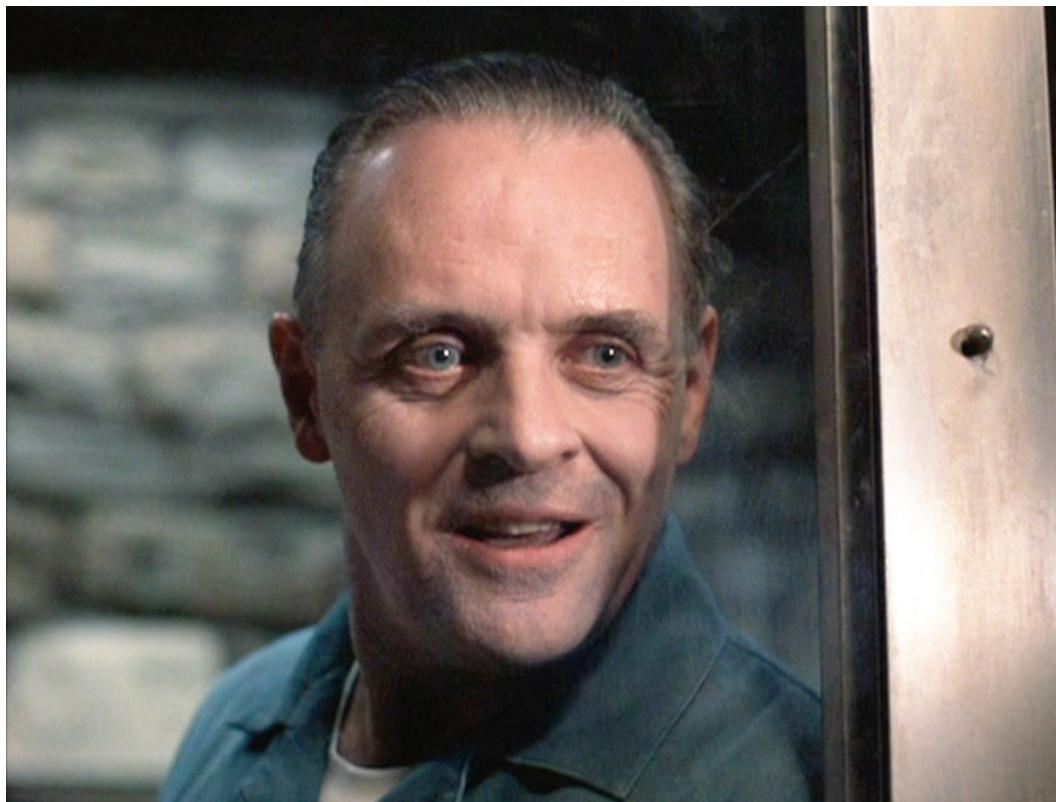
Pragmatische Lösung

- falls es geht: vorher einen Test schreiben
 - ggf. einen großen und langsamen Test (der ggf. eine Klasse testet, die die eigentliche Klasse verwendet)
- falls das nicht geht:
 - Abhängigkeiten möglichst nicht-invasiv brechen (konservativ)
 - möglichst wenig ändern
 - geänderte Code-Stelle nachträglich testen

Allgemeines Vorgehen

1. Änderungspunkte identifizieren
2. Testpunkte finden
3. Abhängigkeiten brechen
4. Tests schreiben
5. Änderungen durchführen

Auch wenn der Code auf den ersten Blick nicht schlimm aussieht:
Steckt ihn in eine Zwangsjacke (Testabsicherung) und lasst ihm keine
Chance.



Quelle: <https://rockmyvegansocks.com>

Er wartet in Wirklichkeit nur darauf, euch das eigene Gehirn zum
Feierabend zu servieren.

Fake- und Mock-Objekte

Primäre Strategie, um Abhängigkeiten zu brechen

Beispiel: Display-Anzeige

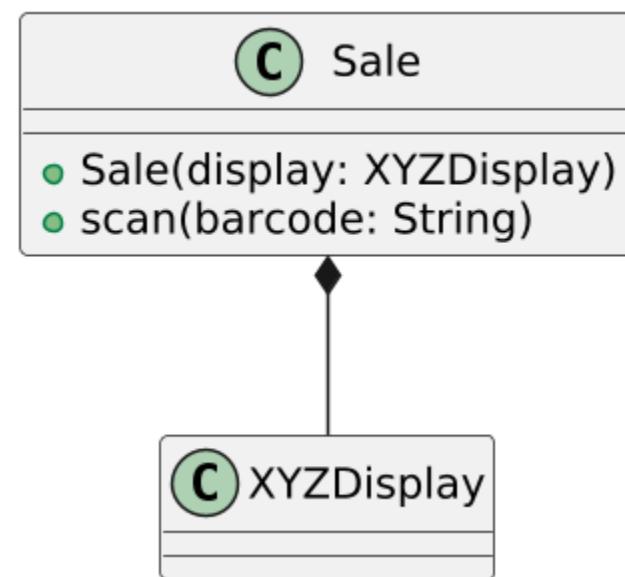
```
class Sale {  
  
    void scan(String barcode) {  
        String product;  
        // do a lot of work to get the right product  
        // ...  
        Display.getXYZDisplay().showLine(product);  
        // ...  
    }  
}
```

Aufgabe: es soll getestet werden, ob das richtige Produkt auf dem Display angezeigt wird.

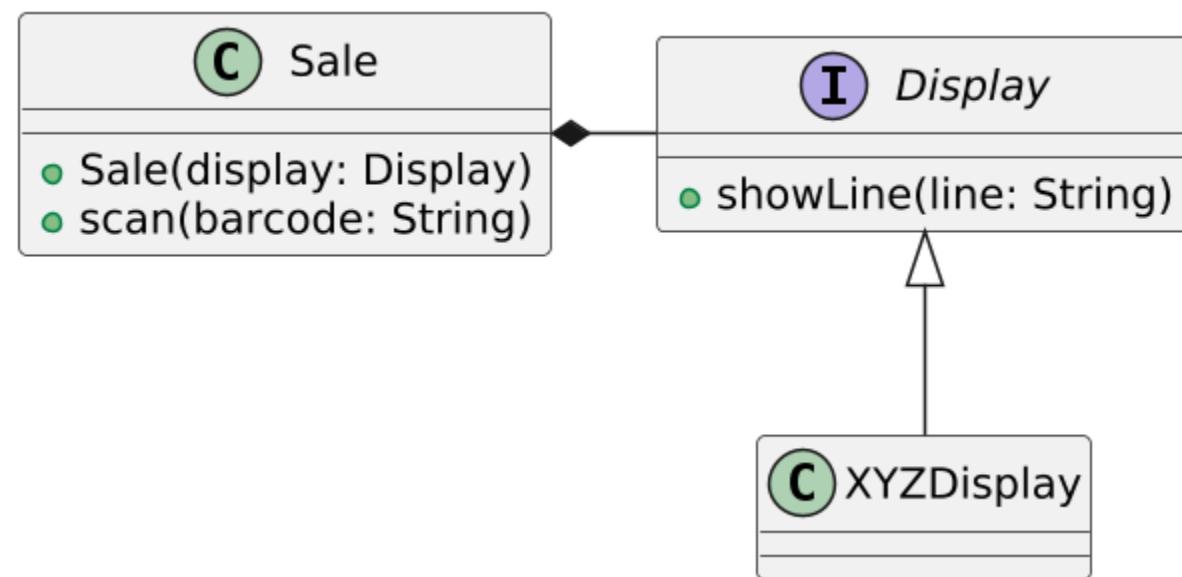
Schwierig zu testen, weil das Display direkt in der Methode verwendet wird.

Lösung: das Display von außen reingeben → neue Klasse

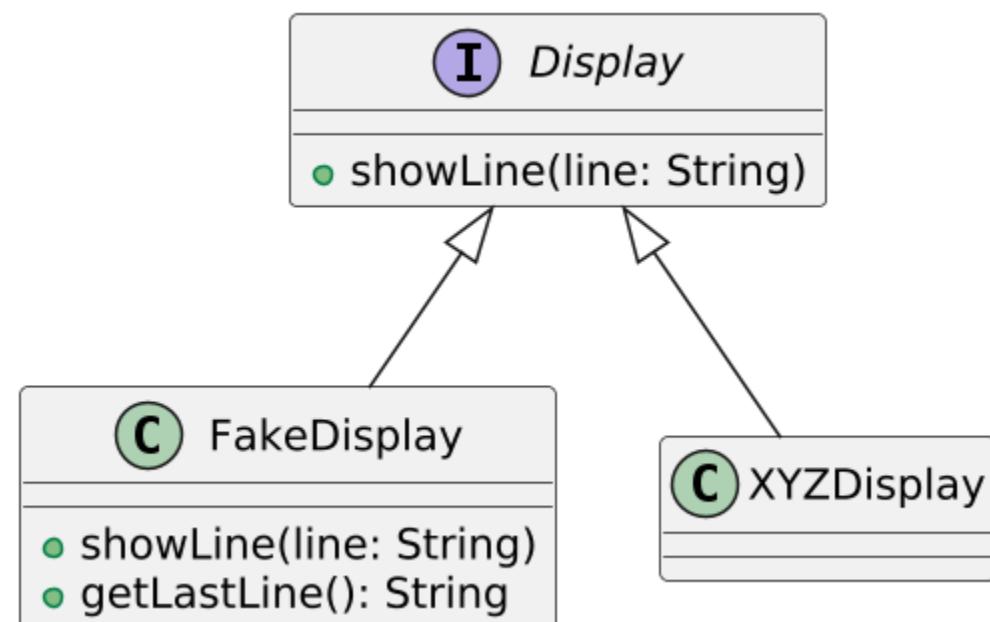
Display übergeben



Interface extrahieren



Brechen mit Fake-Objekt



Unit-Test mit Fake-Objekt

```
public class SaleTest {  
  
    @Test  
    public void testScan() {  
        FakeDisplay display = new FakeDisplay();  
        Sale sale = new Sale(display);  
  
        sale.scan("123456");  
  
        assertEquals("Brot 2,99€", display.getLastLine());  
    }  
}
```

Implementierung FakeDisplay

```
public class FakeDisplay implements Display {  
  
    private String lastLine;  
  
    @Override  
    public void showLine(String s) {  
        lastLine = s;  
    }  
  
    public String getLastLine() {  
        return lastLine;  
    }  
}
```

Mock-Objekte

- sind wie FakeObjekte
- bringen die Evaluierung mit

```
public class SaleTest {  
  
    @Test  
    public void testScan() {  
        MockDisplay display = new MockDisplay();  
        Sale sale = new Sale(display);  
  
        sale.scan("123456");  
  
        display.assertLastLineEquals("Brot 2,99€");  
    }  
}
```

Seam Model

Nahtmodell

Definitionen

"A seam is a place where you can alter behavior in your program without editing in that place."

– M.C. Feathers

"Eine Naht ist eine Stelle, an der die Logik des Programs geändert werden kann, ohne die Stelle selbst zu ändern."

"Every seam has an enabling point, a place where you can make the decision to use one behavior or the other."

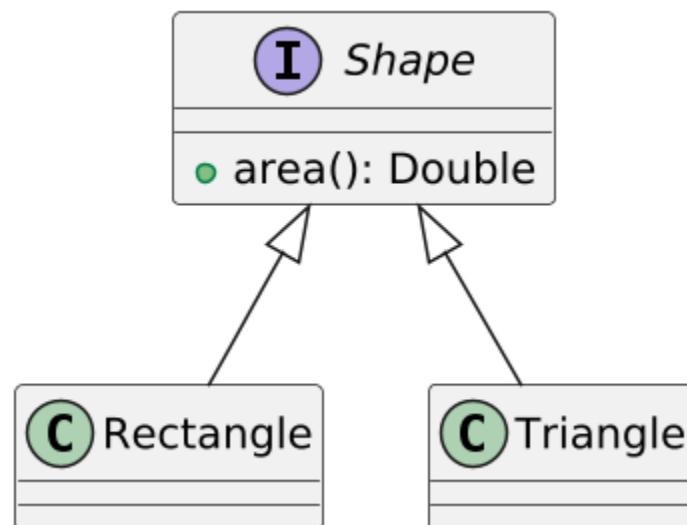
– M.C. Feathers

"Jede Naht hat einen Aktivierungspunkt, eine Stelle, an der entschieden wird, welche Logik benutzt wird."

Object Seams / 00-Nähte

Polymorphie

```
shape.area();
```



Die Methode *area()* ist die Naht, aber wo ist der Aktivierungspunkt?

BEISPIELAUFRUF DER METHODE

```
class Calculator {  
  
    public double calc() {  
        List<Rectangle> shapes = List.of(new Rectangle(2, 2, 4), ...);  
        return shapes.stream().mapToDouble(Rectangle::area).sum();  
    }  
}
```

Das ist keine Naht → hier ist alles hart codiert. Wie kann man das ändern?

```
public double calc(List<Shape> shapes) {  
    return shapes.stream().mapToDouble(Shape::area).sum();  
}
```

- Naht: der Parameter / das Interface

Aktivierungspunkt

```
@Test
void testCalc() {
    Calculator calculator = new Calculator();
    double result = calculator.calc(List.of(new FakeShape(1.1),
        new FakeShape(2.2), new FakeShape(3.3)));
    assertEquals(6.6, result);
}

class FakeShape implements Shape {
    private final double area;
    FakeShape(double area) {
        this.area = area;
    }
    double area() {
        return this.area;
    }
}
```

Vererbung

Anstatt:

```
class Calculator {  
  
    public double complicatedCalculation(List<Shape> shapes) {  
        double x = 0.0;  
        double area = Calculate(shapes);  
        // ...  
        return x + area;  
    }  
  
    private static double Calculate(List<Shape> shapes) {  
        // ...  
    }  
}
```

Besser:

```
class Calculator {  
  
    public double complicatedCalculation(List<Shape> shapes) {  
        double x = 0.0;  
        double area = calculate(shapes);  
        // ...  
        return x + area;  
    }  
  
    protected double calculate(List<Shape> shapes) {  
        // ...  
    }  
}
```

- Naht: Methode, die im Test überschrieben werden kann

Aktivierungspunkt

```
class FakeCalculator extends Calculator {  
    protected double calculate(List<Shape> shapes) {  
        return 1.0;  
    }  
}  
  
@Test  
void testDependentClass() {  
    OtherClass myObject = new OtherClass(new FakeCalculator());  
    assertThat(123, myObject.doSomething());  
}
```

Autowiring

```
class MyClass {  
  
    @Autowired  
    private AutowiredClass autowiredClass;  
  
    //...  
}
```

- das Framework entscheidet, welche Klasse instanziert / zugewiesen wird
- im Test wird das entsprechende Mock-Objekt konfiguriert

Link Seams / Linker-Naht

- viele Sprachen haben einen s.g. *Linker*
 - ein *Linker* verbindet Programmmodule zu einem ausführbaren Programm
 - er löst z.B. Aufrufe auf
- C/C++ hat einen dedizierten Linker
- in Java macht das der Compiler

Classpath

- in Java wird im Classpath nach kompilierten Klassen gesucht
- ändert man den Classpath, können andere Klassen geladen werden

```
java -classpath /home/java/MockClasses;/home/java/ProductionClasses ...
```

- die Reihenfolge ist wichtig: erster Treffer wird benutzt

Preprocessing Seams / Präprozessor-Naht

z.B. in C/C++ existiert der Makro-Präprozessor

```
// FILE: localdefs.h
// ...
#ifndef TESTING
...
int standard_result = 1;

#define calculation()\ \
    {return standard_result;}
...

#endif

// FILE: myprogramm.cpp

extern int calculation();

#include "localdefs.h"

void do_something() {
    // ...
    int result = calculation();
    // ...
}
```

Die Naht sind Methodenaufrufe, der Aktivierungspunkt ist das Makro.

Sprouts and Wrapper

Keimlinge und Hüllen

oder: Wie man mit wenig Zeit Grundlagen zur Verbesserung schaffen kann



Sprout Method

Beispiel

```
class PaymentService {

    public void sendPaymentReminder(List<Sale> sales) {
        List<Email> emails = new ArrayList<>();
        for(Sale sale : sales) {
            if(sale.getPaymentDate() == null) {
                Email email = new Email();
                email.setTo(sale.getRecipientEmail());
                email.setSubject("Payment Reminder");
                email.setBody("Please pay invoice no " +
                             sale.getInvoiceNumber());
                email.setFrom("invoice@example.com");
                emails.add(email);
            }
        }
        EmailBatchSender.send(emails);
    }

}
```

```
//...
public void sendPaymentReminder(List<Sale> sales) {
    List<Email> emails = new ArrayList<>();
    Set<Long> saleIds = new HashSet<>();
    for(Sale sale : sales) {
        if(sale.getPaymentDate() == null &&
           !saleIds.contains(sale.getId())) {
            Email email = new Email();
            email.setTo(sale.getRecipientEmail());
            email.setSubject("Payment Reminder");
            email.setBody("Please pay invoice no " +
                         sale.getInvoiceNumber());
            email.setFrom("invoice@example.com");
            emails.add(email);
            saleIds.add(sale.getId());
        }
    }
    EmailBatchSender.send(emails);
}
```

Warum ist das nicht so gut?

- schwierig zu testen
 - es ist kein Test vorhanden für den alten Code
 - die Abhängigkeit zu EmailBatchSender
 - die alte Logik hat funktioniert, wir wollen aber nur die neue Logik testen
- vorhandener Code wird noch unübersichtlicher
- nicht wiederverwendbar → andere benötigen evtl die gleiche Funktionalität

```
public void sendPaymentReminder(List<Sale> sales) {  
    List<Email> emails = new ArrayList<>();  
    for(Sale sale : uniqueSales(sales)) {  
    }  
    EmailBatchSender.send(emails);  
}  
  
private List<Sale> uniqueSales(List<Sale> sales) {  
    Set<Long> ids = new HashSet<>();  
    List<Sale> uniqueSales = new ArrayList<>();  
    for (Sale sale : sales) {  
        if (ids.contains(sale.getId()))  
            continue;  
        uniqueSales.add(sale);  
    }  
    return uniqueSales;  
}
```

Warum ist das besser?

- leichter zu testen
 - die Methode könnte *protected* oder *package private* sein, um im Test darauf zuzugreifen
 - die Methode könnte während der Entwicklung *public* sein
- saubere Trennung zwischen altem und neuem Code
- wiederverwendbar

- sobald jemand die Methode wiederverwendet, ist die Saat aufgegangen → es hat sich gelohnt
- es könnten sich mehrere ähnliche Sprout-Methods ansammeln
 - z.B. uniqueSales, salesWithSameRecipient, salesWithHighVolume, ...
 - → auslagern in eine eigene Klasse (z.B. SalesFilter)
- Sprout-Methods können Anhaltspunkte für spätere Refactorings sein

Nachteile

- keine unmittelbare Verbesserung des alten Codes
- man könnte beim Lesen über diesen Methoden-Aufruf 'stolpern'
 - z.B. wenn alles innerhalb der alten Methode ausgeführt wird, aber nur ein kleiner trivialer Teil delegiert wird → man fragt sich unwillkürlich 'Warum?'

Sprout Classes

- wenn Sprout-Method nicht ausreicht, weil man keine Möglichkeit hat, die aktuelle Klasse aufgrund ihrer Abhängigkeiten zu testen (d.h. evtl. nicht mal zu instanziieren)

Beispiel

```
class PaymentService {  
  
    PaymentService(NonFakeable obj) {  
        obj.doSomething();  
    }  
  
    public void sendReminder(List<Sale> sales) {  
        // ...  
        // this method needs some extension  
        // ...  
    }  
}
```

Lösung

```
class PaymentService {  
    // ...  
  
    public void sendReminder(List<Sale> sales) {  
        // ...  
        SaleFilter filter = new SaleFilter();  
        filter.uniqueSales(sales);  
        // ...  
    }  
}
```

```
class SaleFilter {  
    void uniqueSales(List<Sale> sales) {  
        // ...  
    }  
}
```

→ *SaleFilter* kann vollständig und einfach getestet werden

Vor-/Nachteile

- (+) wiederverwendbar
- (+) testbar
- (+) führt zu SRP
- (+) Anhaltspunkt für Refactoring oder Erweiterungen
- (-) erhöhte Komplexität im Design
- (-) entgegen dem aktuellen Design

Wrap Method

- Erweiterung zu *Sprout Method*
- klarere Trennung zwischen neuem und altem Code
- v.a. dann hilfreich, wenn es sich nur um temporäre Kopplung handelt
 - es verhindert, dass der temporär gekoppelte Code zu stark ineinander verwächst und später nicht mehr trennbar ist

Beispiel: Nachbestellung

```
class ShoppingCart {  
    // ...  
    void buy() {  
        double sum = 0.0;  
        for(Article article : articles) {  
            sum += article.getPrice();  
        }  
        creditCard.charge(sum);  
    }  
}
```

- Neues Feature: jedes Mal, wenn ein Artikel verkauft wird, soll dieser nachbestellt werden.

Möglichkeit 1: Direkte Implementierung

```
class ShoppingCart {  
    // ...  
    void buy() {  
        double sum = 0.0;  
        for(Article article : articles) {  
            sum += article.getPrice();  
        }  
        creditCard.charge(sum);  
        // [put reorder logic here]  
    }  
}
```

- nicht wiederverwendbar
- nicht einzeln testbar
- schwierig zu verstehen, was die Methode alles macht (Stichwort: SRP)

Möglichkeit 2: Sprout Method

```
class ShoppingCart {  
    // ...  
    void buy() {  
        double sum = 0.0;  
        for(Article article : articles) {  
            sum += article.getPrice();  
        }  
        creditCard.charge(sum);  
        reorderItems();  
    }  
    void reorderItems() {  
        for(Article article : articles) {  
            // [reorder logic]  
        }  
    }  
}
```

- wiederverwendbar
- einzeln testbar
- aber seltsam zu testen, da *articles* ein Feld der Klasse ist
- *buy()* ist besser zu verstehen, aber immer noch etwas unübersichtlich

Möglichkeit 2.1: Sprout Method mit Parameter

```
class ShoppingCart {  
    // ...  
    void buy() {  
        double sum = 0.0;  
        for(Article article : articles) {  
            sum += article.getPrice();  
        }  
        creditCard.charge(sum);  
        reorderItems(articles);  
    }  
    void reorderItems(Set<Article> articles) {  
        for(Article article : articles) {  
            // [reorder logic]  
        }  
    }  
}
```

- zusätzlich zum vorherigen ist diese Methode durch den Parameter einfacher zu testen

Möglichkeit 3: Wrap Method

```
void buy() {
    chargeCreditCard(articles);
    reorderArticles(articles);
}
private void chargeCreditCard(
    List<Article> articles) {
    double sum = 0.0;
    for(ShoppingItem item : items) {
        sum += item.getPrice();
    }
    creditCard.charge(sum);
}
private void reorderArticles(List<Article> articles) {
    // ...
}
```

Vor-/Nachteile

- (+) wie bei Sprout-Methods
- (+) neue Funktionalität kann nicht mit aktueller Funktionalität vermischt werden
- (-) Methodennamen könnten schwer zu finden sein
 - (o) Renaming mit modernen IDEs aber kein Problem

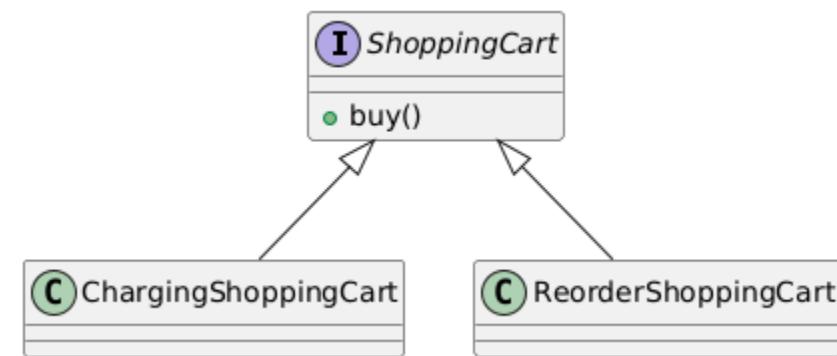
Wrap Class

- wenn die neue Funktionalität unabhängig von der aktuellen Klasse ist oder nichts mit dem vorhandenen Code dort zu tun hat
- wenn die aktuelle Klasse einfach nicht größer werden sollte

Beispiel

```
class ShoppingCart {  
    // ...  
    void buy() {  
        double sum = 0.0;  
        for(Article article : articles) {  
            sum += article.getPrice();  
        }  
        creditCard.charge(sum);  
    }  
}
```

→ Interface extrahieren und in neue und alte Klasse implementieren



```
class ReorderShoppingCart {

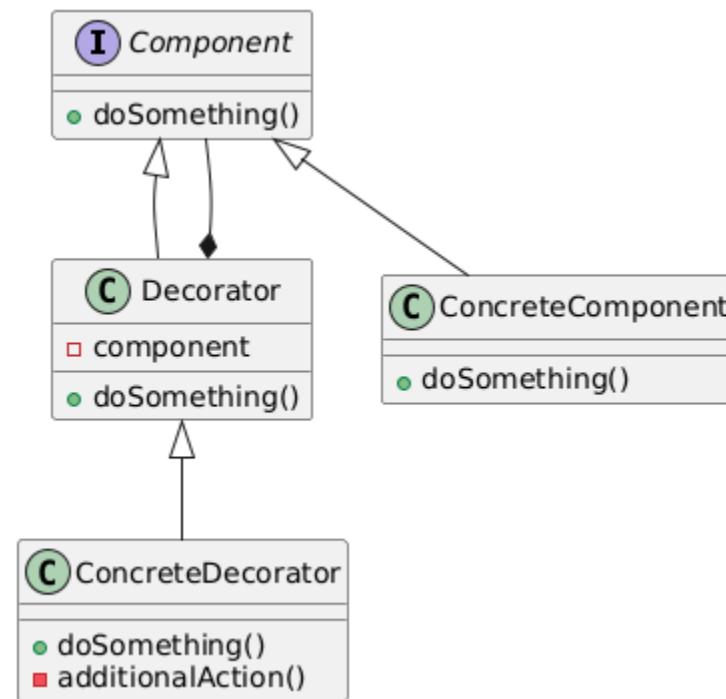
    private final ShoppingCart cart;

    ReorderShoppingCart(ShoppingCart cart) {
        this.cart = cart;
    }

    void buy() {
        reorderArticles(cart.getArticles());
        cart.buy();
    }

    private reorderArticles(List<Article> a) {
        //...
    }
}
```

Decorator-Pattern



Vor-/Nachteile

- (+) alte Klasse muss nicht angefasst werden
- (+) neue Funktionalität leicht zu testen
- (-) Debuggen und Lesen kann anstrengender werden, da es viele Schichten geben kann
- (-) wenn die neue Funktionalität in vorhandenem Code genutzt werden soll, müssen alle Aufrufer angepasst werden

Compile Time

- Compile-Zeit sollte möglichst kurz sein, um möglichst schnell Feedback zu bekommen

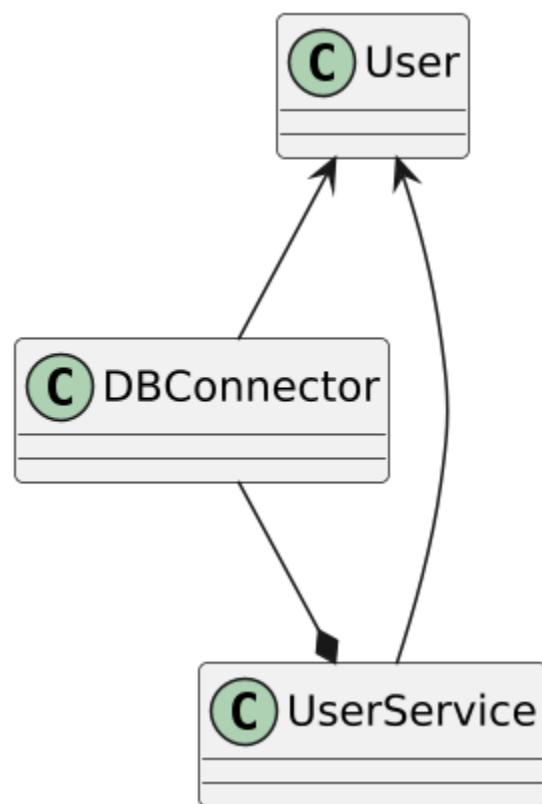
Lag Time



Quelle: Courtesy NASA/JPL-Caltech.

Build Dependencies

Was passiert, wenn wir die Klasse *User* ändern? Z.B. eine Methode extrahieren?

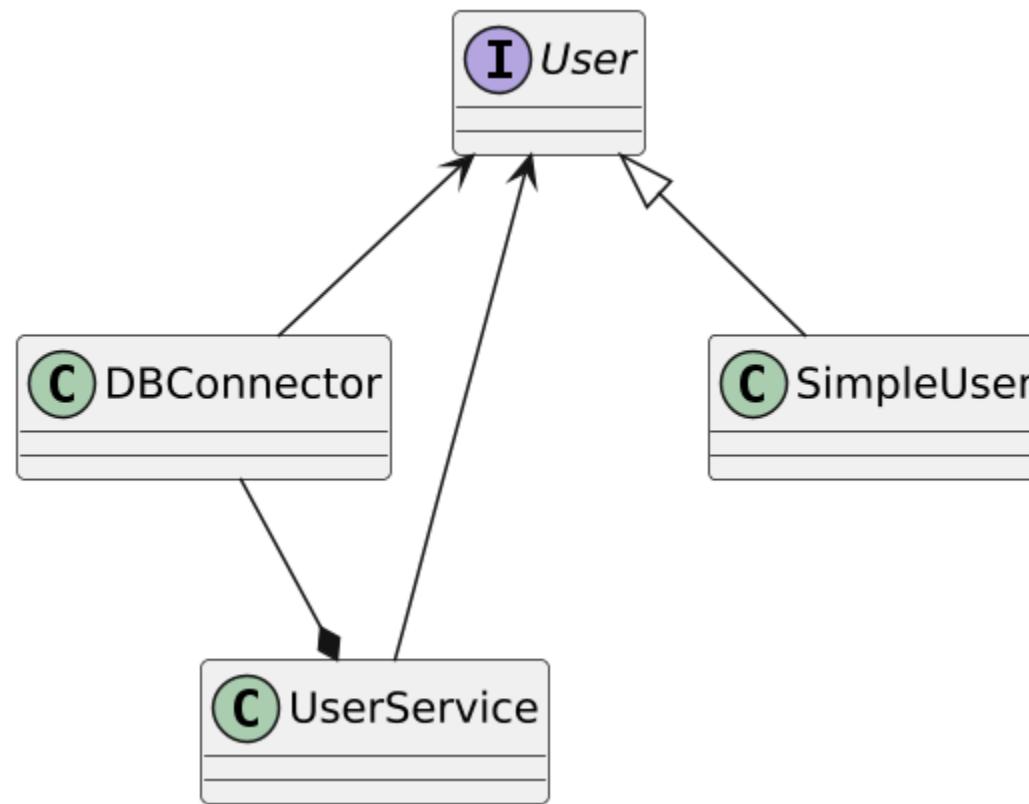


Alle Klassen müssen neu kompiliert werden, selbst wenn sich die Methoden-Signaturen nicht ändern.

→ Lösung?

Interface

Was passiert, wenn wir jetzt die Klasse *SimpleUser* ändern?



Nur *SimpleUser* muss neu compiliert werden.

→ führt zusätzlich zu DIP

Schwierige Klassen

Hauptprobleme:

- abhängige Objekte können nur schwer erzeugt werden
- der Konstruktor hat Seiteneffekte
- der Konstruktor enthält Logik, die ebenfalls getestet werden muss

Allgemeines Vorgehen

- Testmethode anlegen
- die Klasse instanziieren
 - für alle Objekte *null* übergeben
- kompilieren und ausführen
- Fake-Objekte anlegen, wenn *null* zu Fehlern führt
 - ggf. *Extract Interface* oder *Sublcass And Override* anwenden

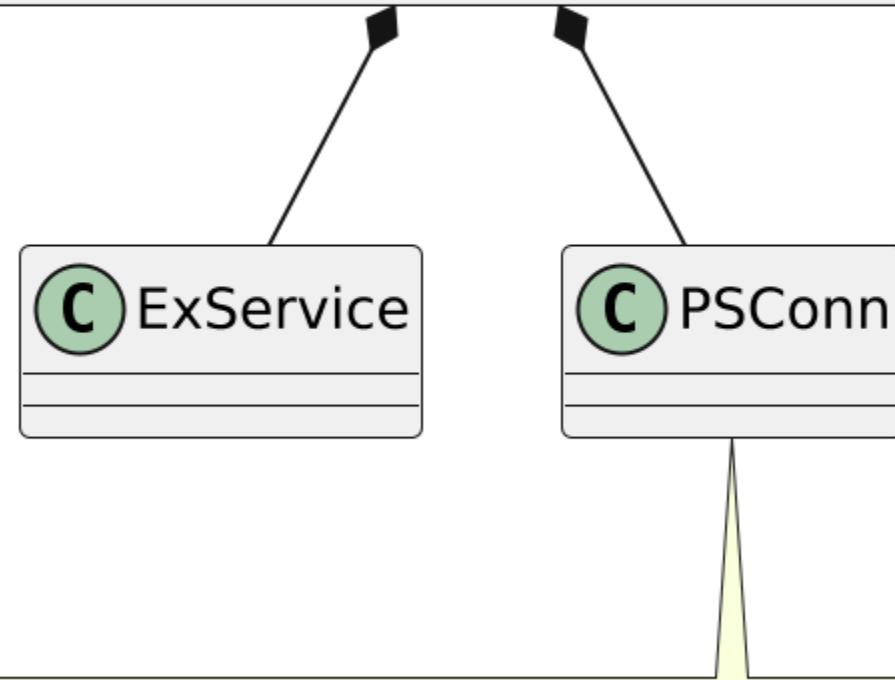
BEISPIEL: ALLG. VORGEHEN BEI KLASSEN

```
class PaymentService {  
  
    PaymentService(PSConn connection,  
                  ExService service,  
                  int id) {  
        // alle Argumente werden jeweils  
        // einem Property/Feld zugewiesen  
    }  
    //...  
  
    void pay(double usd, CreditCard card) {  
        // benutzt connection: PSConn  
    }  
}
```

Die **pay**-Methode soll erweitert werden und muss daher in einen Test.

C PaymentService

- PaymentService(conn: PSConn, ser: ExService, id: int)
- pay(usd: double, card: CreditCard)



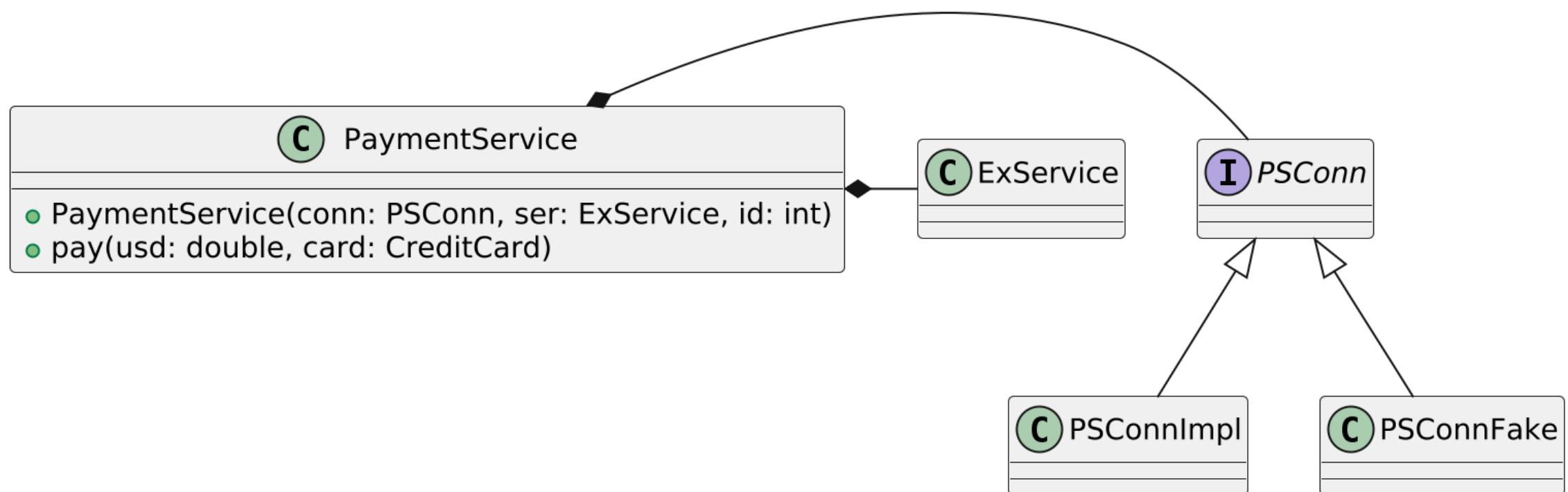
baut im Konstruktor eine Verbindung zum PaymentServer auf

TEST: 1. VERSUCH

```
void testPay() {  
    PaymentService service = new PaymentService(null, null, 0);  
    service.pay(333.33, new CreditCard());  
}
```

- endet in einer *NullPointerException*, weil *PaymentServerConnection* nicht vorhanden war

EXTRACT INTERFACE



TEST: 2. VERSUCH

```
void testPay() {  
    PaymentService service = new PaymentService(new PSConnFake(), null, 0);  
    service.pay(333.33, new CreditCard());  
}
```

→ der Test kompiliert und läuft durch → jetzt können die echten Tests geschrieben werden

Konstruktor mit Nebeneffekten

```
class PaymentService {  
  
    PaymentService() {  
        PSConn conn = new PSConn(PS_URL);  
        conn.connect();  
        // ...  
    }  
  
    // ...  
}
```

Ein Verbindung zum *PaymentServer* wollen wir unter keinen Umständen im Test haben. Sie ist langsam und zudem nur für den Produktivbetrieb.

→ Was machen wir?

PARAMETRIZE CONSTRUCTOR

```
class PaymentService {  
  
    PaymentService() {  
        this(new PSConn(PS_URL));  
    }  
  
    PaymentService(PSConn conn) {  
        conn.connect();  
        // ...  
    }  
  
}
```

Construction Blob

```
class PaymentService {  
    // ...  
  
    PaymentService(String pw, Creator creator) {  
        // ...  
        ValueExtractor extractor = new ValueExtractor(pw);  
        this.values = extractor.getSomeValues(pw);  
        this.created = this.values.create(creator);  
  
        this.verifier = new PaymentVerifier(pw, this.values,  
                                           this.created);  
        // ...  
    }  
}
```

Wir brauchen ein Mock-Objekt für *PaymentVerifier* in unseren Tests.

Wie machen wir das?

VERSUCH 1: PARAMETRIZE CONSTRUCTOR

```
class PaymentService {  
    // ...  
  
    PaymentService(String pw, Creator creator) {  
        this(pw, creator, verifier);  
        // ...  
        ValueExtractor extractor = new ValueExtractor(pw);  
        this.values = extractor.getSomeValues(pw);  
        this.created = this.values.create(creator);  
  
        this.verifier = new PaymentVerifier(pw, this.values,  
                                           this.created);  
        // ...  
    }  
  
    PaymentService(String pw, Creator c, PaymentVerifier v) {  
        // ...  
    }  
}
```

In Java nicht möglich, da ein anderer Konstruktor immer als erstes aufgerufen werden muss und an dieser Stelle noch kein *PaymentVerifier* existiert.

VERSUCH 2: SUPERSEDE INSTANCE VARIABLE

Instanzvariable überschreiben

```
class PaymentService {  
    // ...  
  
    PaymentService(String pw, Creator creator) {  
        // ...  
    }  
  
    void supersedePaymentVerifier(PaymentVerifier verifier) {  
        this.verifier = verifier;  
    }  
}
```

VOR-/NACHTEILE

- (+) Test mit Mock-Objekt ist möglich
- (+) Legacy Code wurde nicht angefasst
- (-) PaymentVerifier kann von Benutzern überschrieben werden

Singletons

```
class PaymentService {  
  
    PaymentService(String url) {  
        PSConn conn = ConnProvider.getInstance().connect(url);  
        // ...  
    }  
  
}
```

STATIC SETTER

```
class ConnProvider {  
  
    private static ConnProvider inst;  
  
    public static void setTestInst(ConnProvider cp) {  
        inst = cp;  
    }  
  
    public ConnProvider() {}  
  
}
```

VOR-/NACHTEILE STATIC SETTER

- (-) Überschreiben der Instanz nun möglich → widerspricht Singleton-Pattern
- (-) es können mehrere Instanzen gleichzeitig zur Laufzeit existieren
 - immerhin: Methodename verrät, dass es nur für Tests gedacht ist
- (-) Konstruktor muss *public* sein, sonst können keine neuen Instanzen erzeugt werden
- (+) einfache und schnelle Möglichkeit
- (+) mit *Extract Interface* können Fake- und Mock-Objekte übergeben werden
 - zusätzlich kann der Konstruktor wieder *private* sein

STATIC RESET

```
class ConnProvider {  
  
    private static ConnProvider inst;  
  
    public static void resetInstance() {  
        inst = null;  
    }  
  
    public static ConnProvider getInst() {  
        // lazy init  
    }  
  
    private ConnProvider() {}  
  
}
```

VOR-/NACHTEILE STATIC RESET

- (+) überschreiben der Instanz nicht möglich
- (+) Konstruktor bleibt *private*
- (-) es können mehrere Instanzen gleichzeitig zur Laufzeit existieren
- (-) keine Fake- oder Mock-Objekte möglich

Schwierige Methoden

Hauptprobleme:

- die Methode ist vom Test nicht aufrufbar (z.B. *private*)
- die Argumente sind nicht leicht zu erzeugen / ersetzen (s. auch *Schwierige Klassen*)
- die Methode hat sehr negative Nebeneffekte und kann daher nicht im Test aufgerufen werden (z.B. Änderungen in der DB)
- ein Objekt wird von der Methode verwendet, das man zur Validierung braucht, aber nicht 'greifbar' ist

Private Methoden

Wie kann man diese testen?

- Sichtbarkeit erhöhen (*protected* oder *public*)
- auslagern in eigene Klasse

Die Notwendigkeit eine private Methode testen zu müssen, ist ein starker Indikator für zu viele Verantwortlichkeiten innerhalb einer Klasse. Auslagern wäre hier der Idealfall.

==== !

===== Sichtbarkeit erhöhen: Public

Alt

Neu

