

LEGACY CODE

Arbeit und Umgang mit Legacy Code

Maurice Müller

2025-03-24

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)

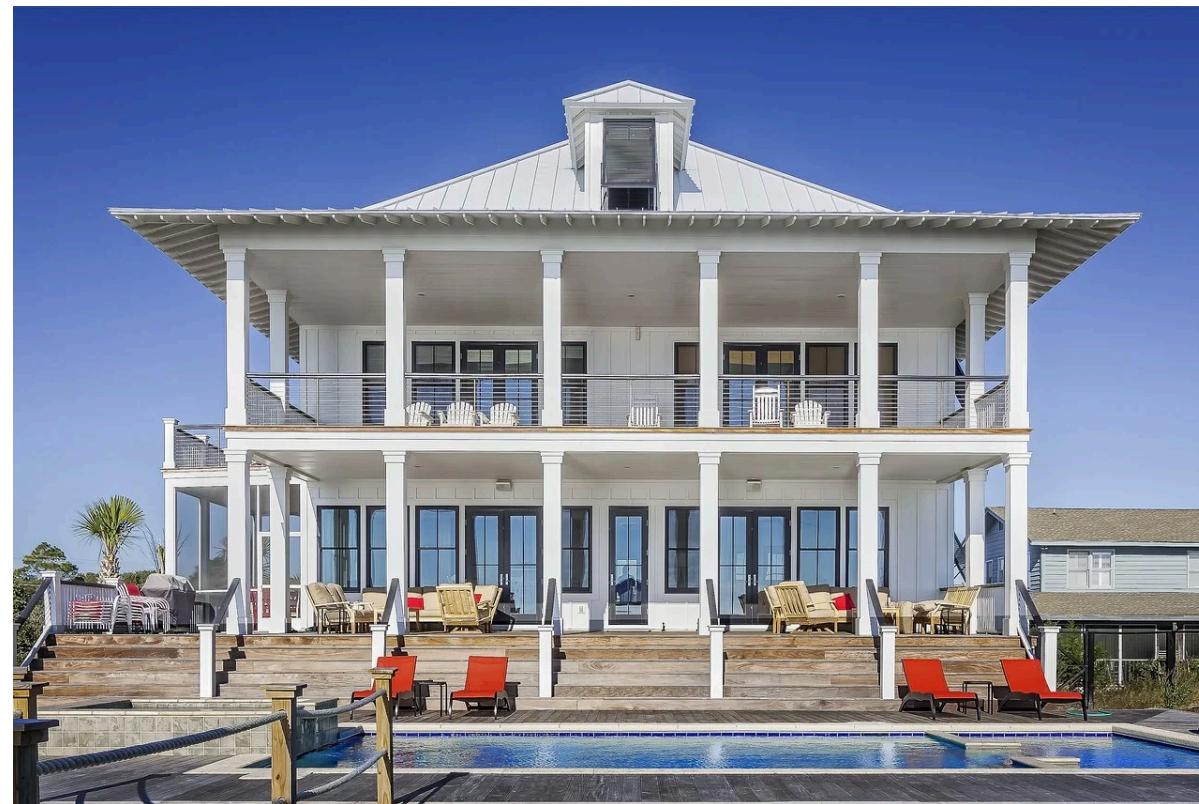
Als Antwort auf M. Feathers Frage, wie sich das neue Team macht:

| *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



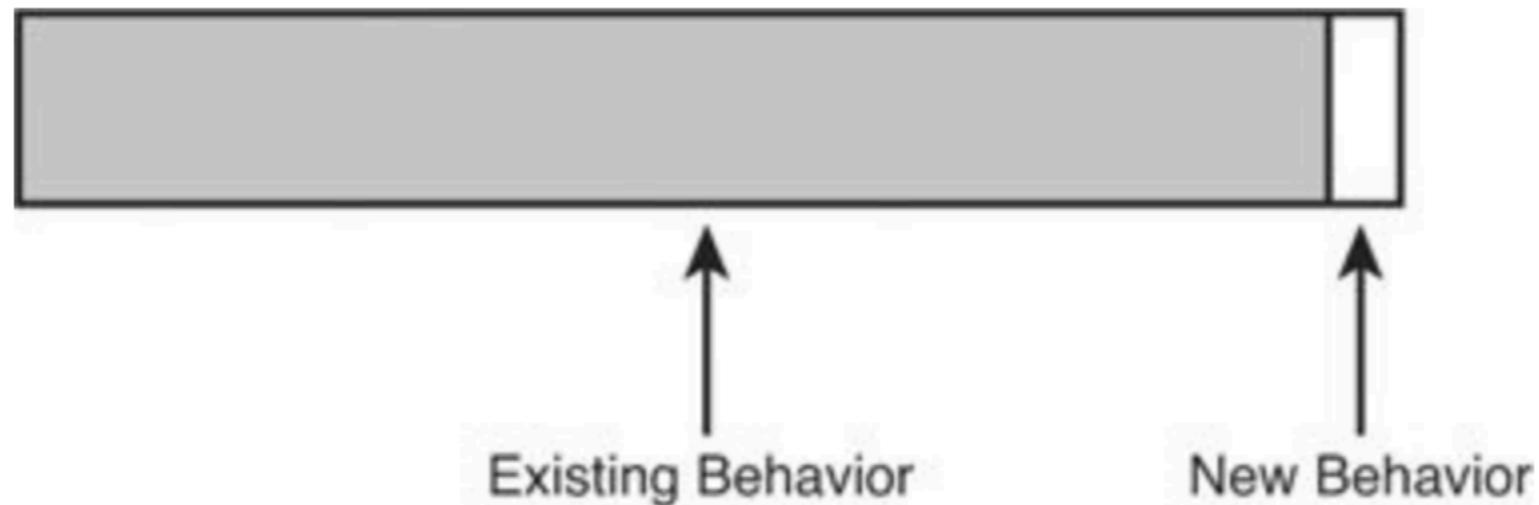




Warum ändert sich Software?

- 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

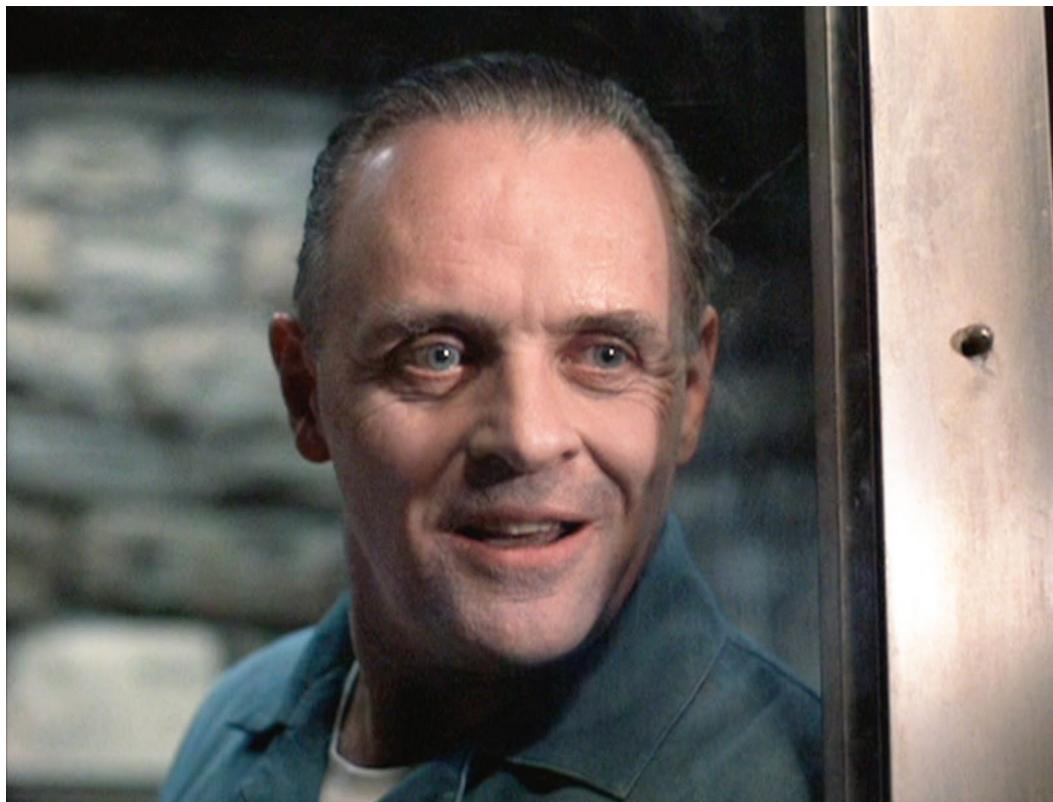
Pragmatische Lösung

- 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

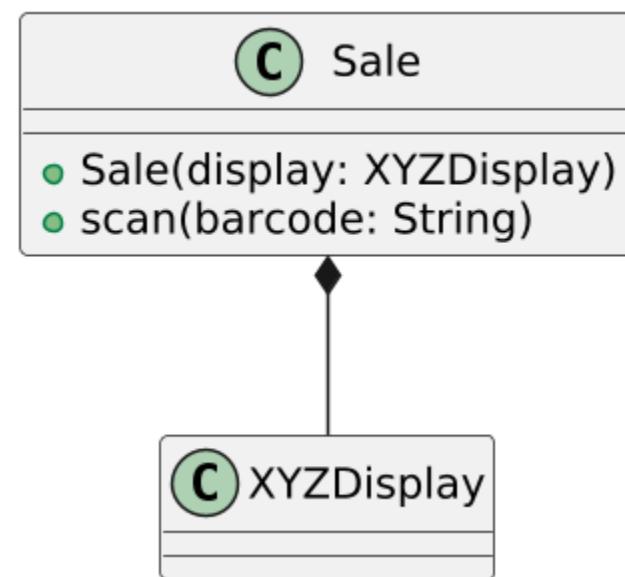
```
class Sale {  
  
    void scan(String barcode) {  
        String product;  
        // do a lot of work to get the right product  
        // ...  
        Display.getXYZDisplay().showLine(product);  
        // ...  
    }  
}
```

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

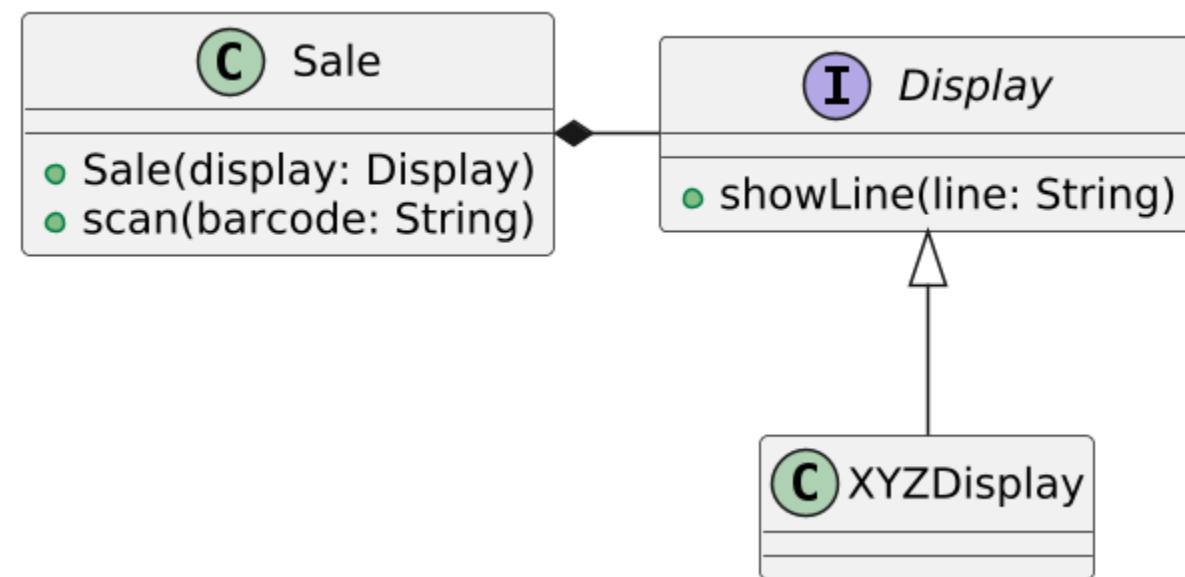
Schwierig zu testen, weil das Display direkt in der Methode verwendet wird.

Lösung: das Display von außen reingeben → neue Klasse

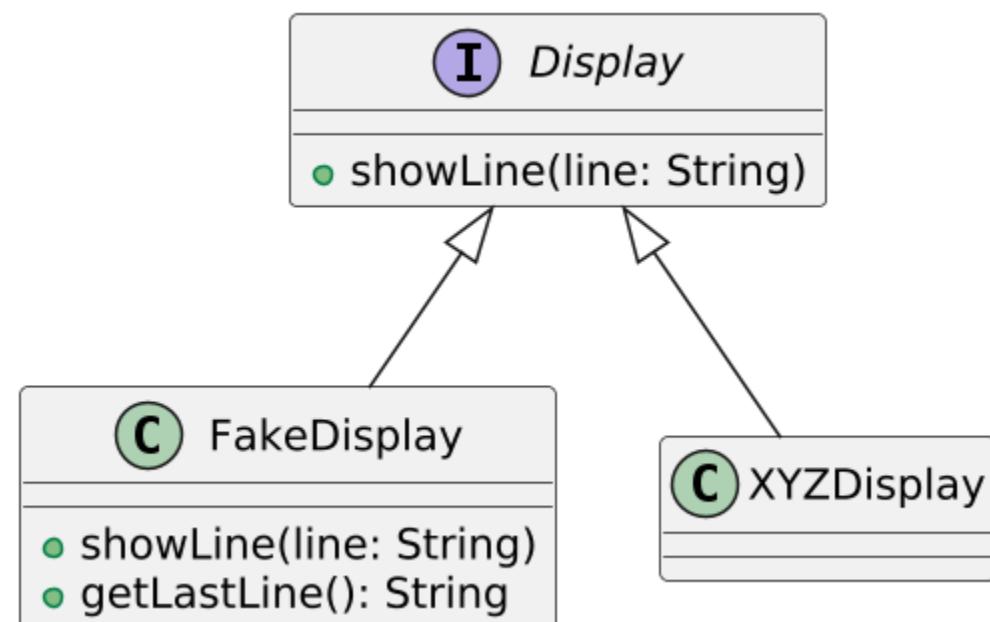
Display übergeben



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."

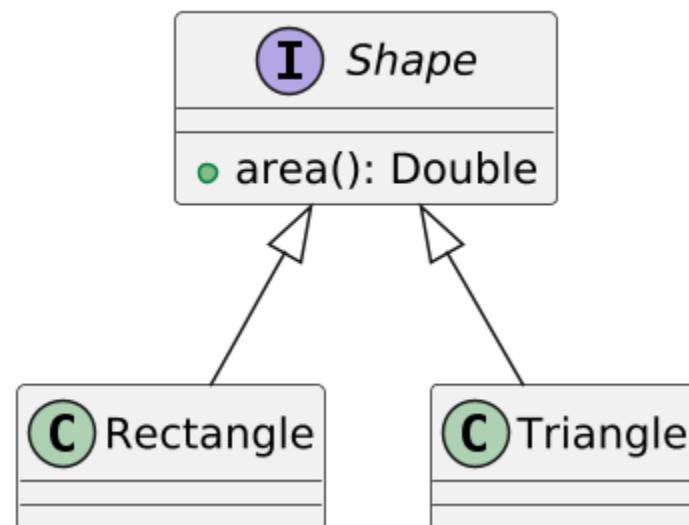
– M.C. Feathers

"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()\ \
    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

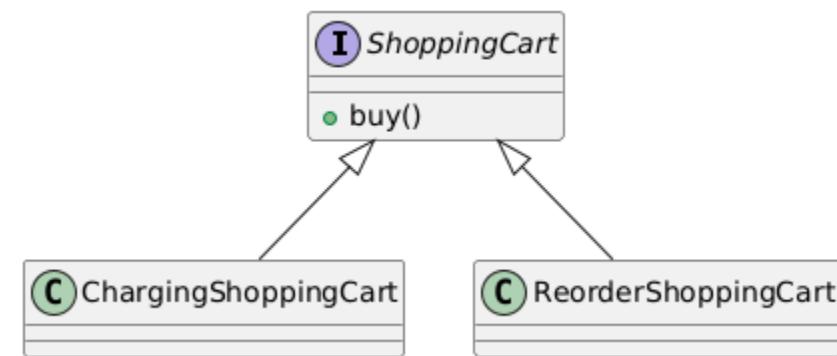
Wrap Class

- 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 {

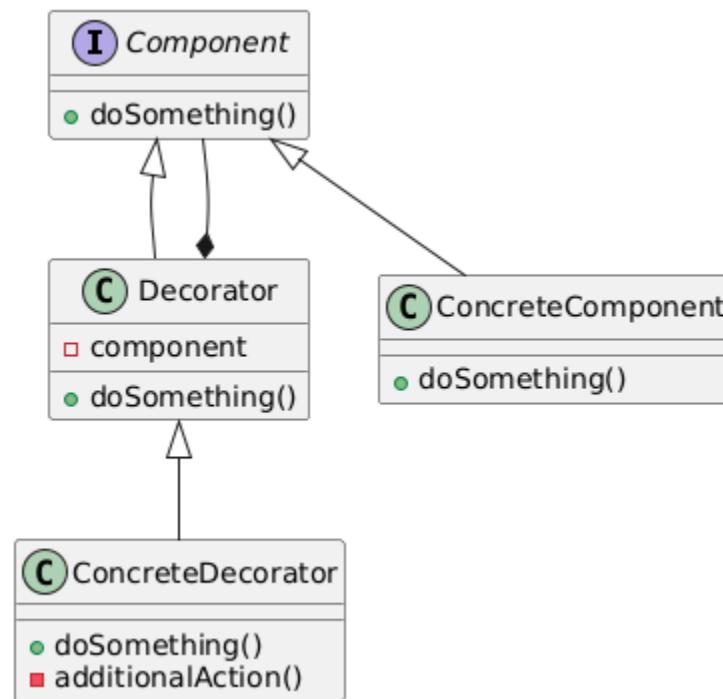
    private final ShoppingCart cart;

    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

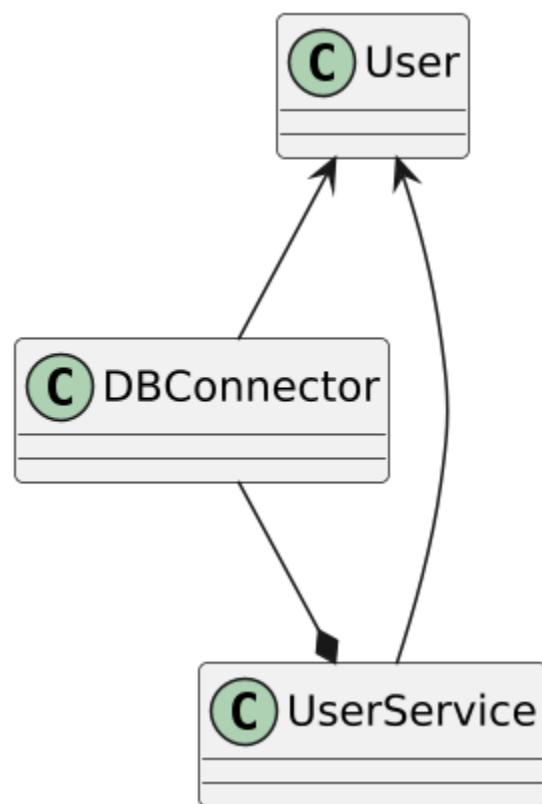
Lag Time



Quelle: Courtesy NASA/JPL-Caltech.

Build Dependencies

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

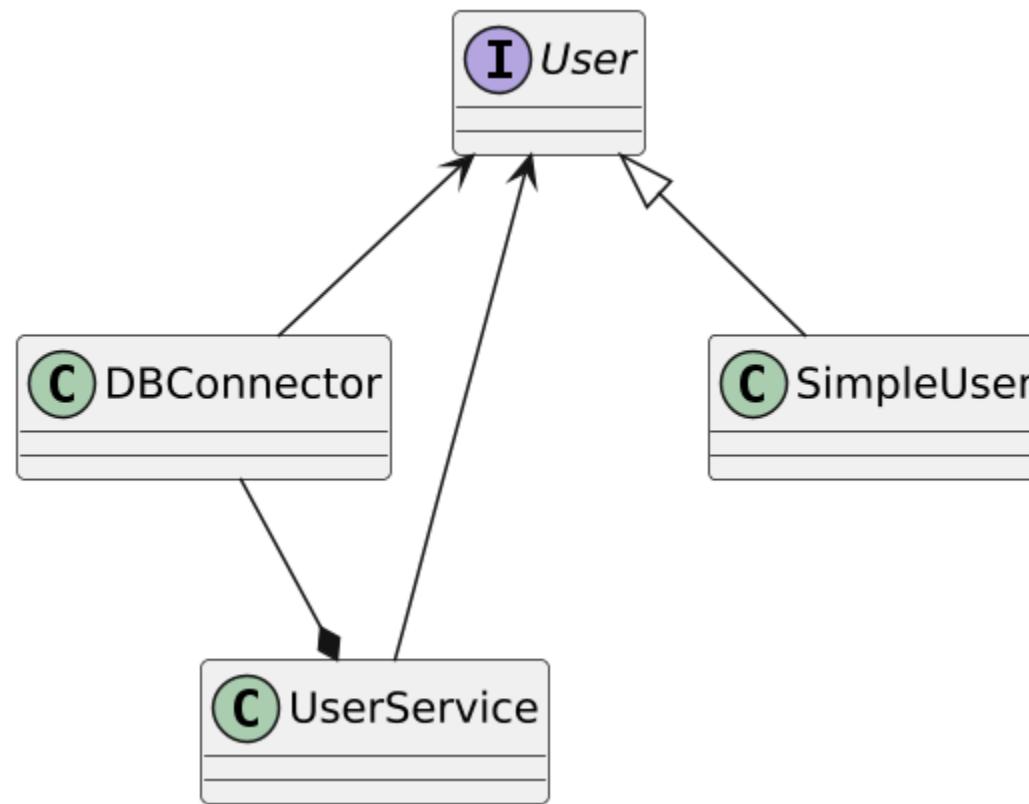


Alle Klassen müssen neu kompiliert werden, selbst wenn sich die Methoden-Signaturen nicht ändern.

→ 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 *Subclass And Override* anwenden

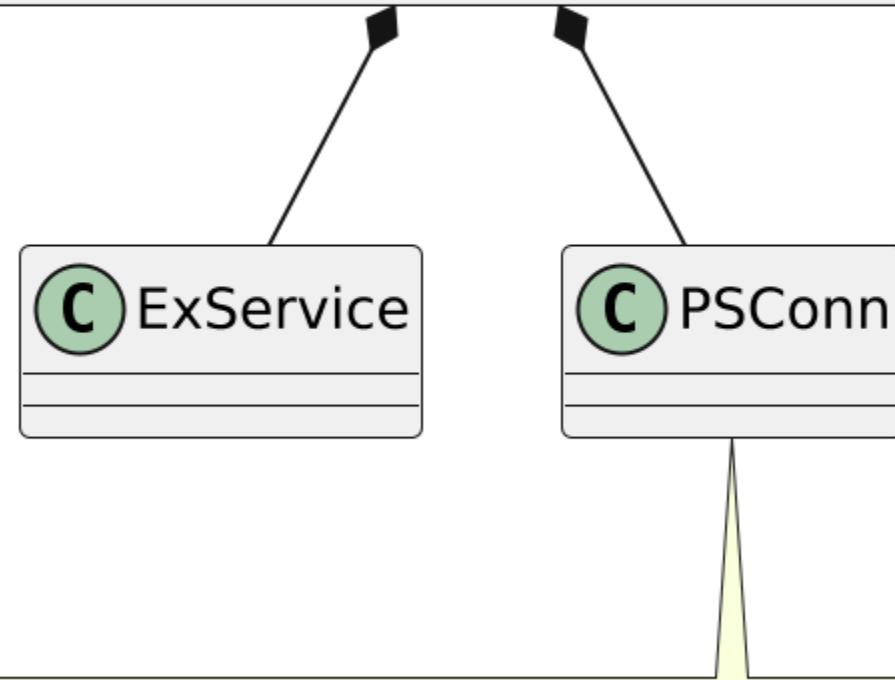
BEISPIEL: ALLG. VORGEHEN BEI KLASSEN

```
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)



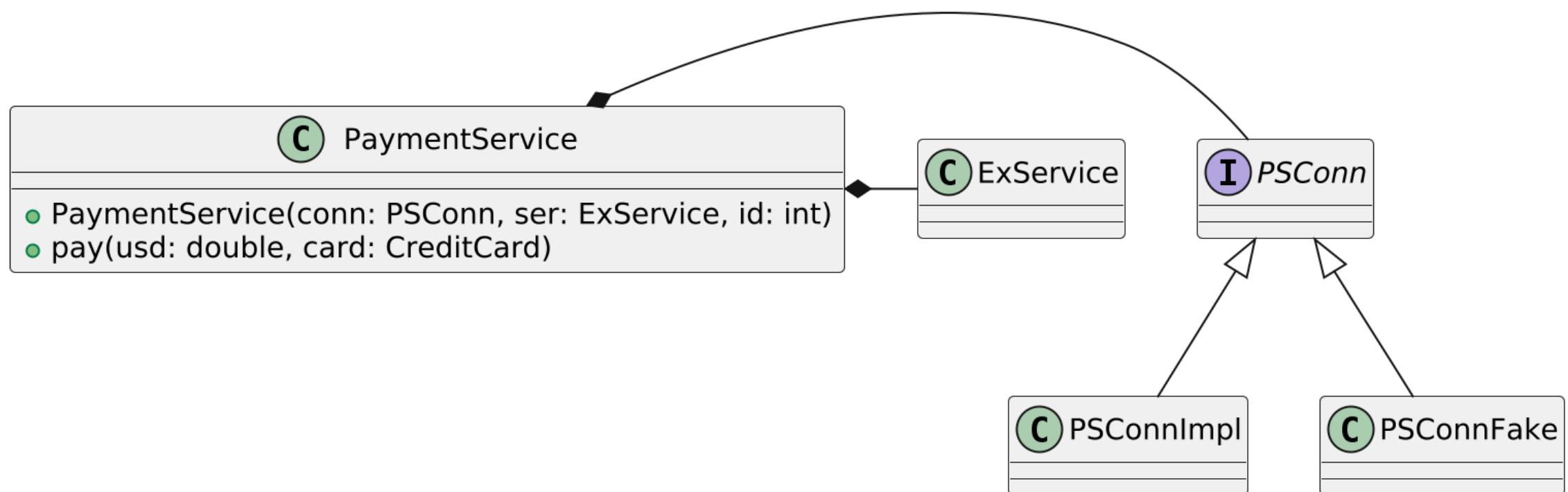
baut im Konstruktor eine Verbindung zum PaymentServer auf

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

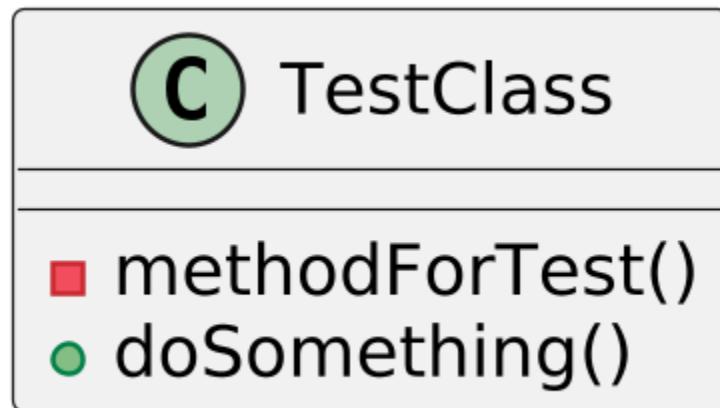
Wie kann man diese testen?

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

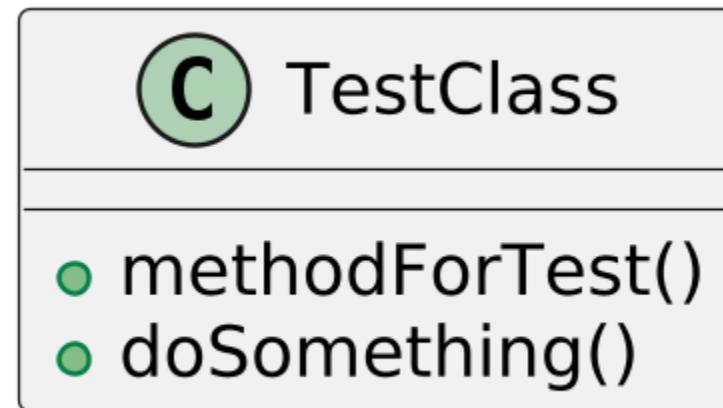
Die Notwendigkeit eine private Methode testen zu müssen, ist ein starker Indikator für zu viele Verantwortlichkeiten innerhalb einer Klasse. Auslagern wäre hier der Idealfall.

SICHTBARKEIT ERHÖHEN: PUBLIC

Alt



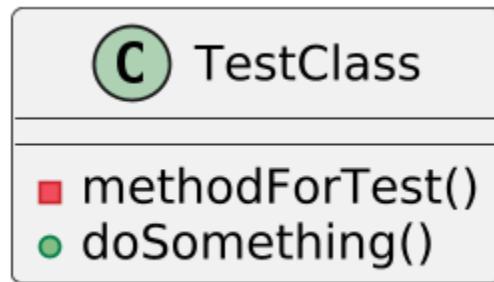
Neu



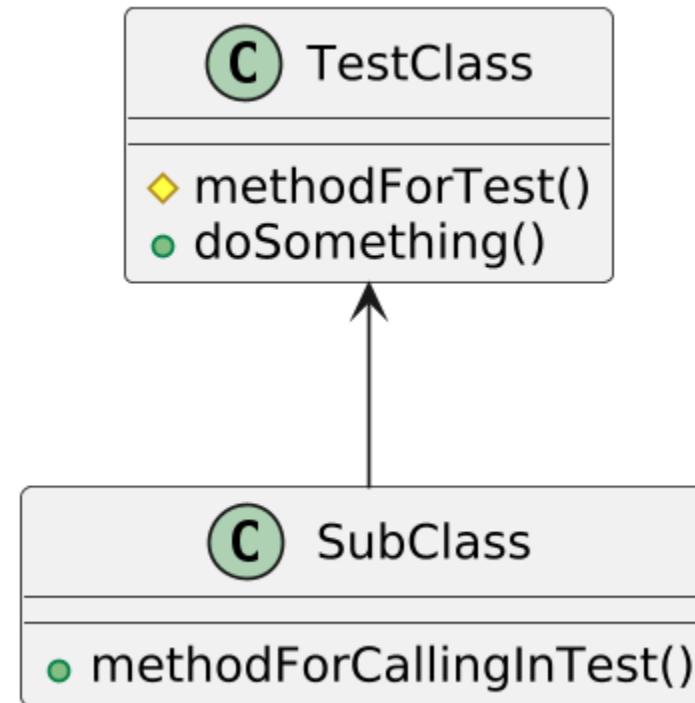
Nur dann sinnvoll, wenn die Methode von anderen Benutzern (sinnvoll) verwendet werden kann und keine Nebeneffekte in der aufgerufenen Klasse hat.

SICHTBARKEIT ERHÖHEN: PROTECTED

Alt



Neu



guter Kompromiss zwischen Testbarkeit $\leftarrow \rightarrow$ Sichtbarkeit

Methoden mit Nebeneffekten

Folgende Beispiele zu *Methoden mit Nebeneffekten* wurde aus dem Buch 'Working Effectively With Legacy Code' übernommen

```
public class AccountDetailFrame extends Frame
    implements ActionListener, WindowListener {

    private TextField display = new TextField(10);
    //...
    public AccountDetailFrame(...) { ... }

    public void actionPerformed(ActionEvent event) {
        String source = (String)event.getActionCommand();
        if (source.equals("project activity")) {
            DetailFrame detailDisplay = new DetailFrame();
            detailDisplay.setDescription(getDetailText() +
                " " + getProjectionText());
            detailDisplay.show();
            String accountDescription =
                detailDisplay.getAccountSymbol();
            accountDescription += ": ";
            //...
            display.setText(accountDescription);
            //...
        }
    }
}
```

EXTRACT METHOD

```
public void actionPerformed(ActionEvent event) {  
    String source = (String)event.getActionCommand();  
    performCommand(source);  
}  
  
public void performCommand(String source) {  
    if (source.equals("project activity")) {  
        DetailFrame detailDisplay = new DetailFrame();  
        detailDisplay.setDescription(getDetailText() +  
            " " + getProjectionText());  
        detailDisplay.show();  
        String accountDescription =  
            detailDisplay.getAccountSymbol();  
        accountDescription += ":";  
        //...  
        display.setText(accountDescription);  
        //...  
    }  
}
```

EXTRACT METHOD: EXTERNE KOMPONENTE ISOLIEREN

```
private TextField display;
private DetailFrame detailDisplay;

public void performCommand(String source) {
    if (source.equals("project activity")) {
        showDescription(getDetailText() + " " +
                        getProjectionText());
        //...
        setDisplayText(getAccountDescription());
        //...
    }
}

void showDescription(String description) {
    detailDisplay = new DetailFrame();
    detailDisplay.setDescription(description);
    detailDisplay.show();
}

String getAccountSymbol() {
    return detailDisplay.getAccountSymbol();
}

void setDisplayText(String text) {
    display.setText(text);
}

String getAccountDescription() {
    return getAccountSymbol() + ": ";
}
```

TESTEN

z.B. mit *Subclass and Override*

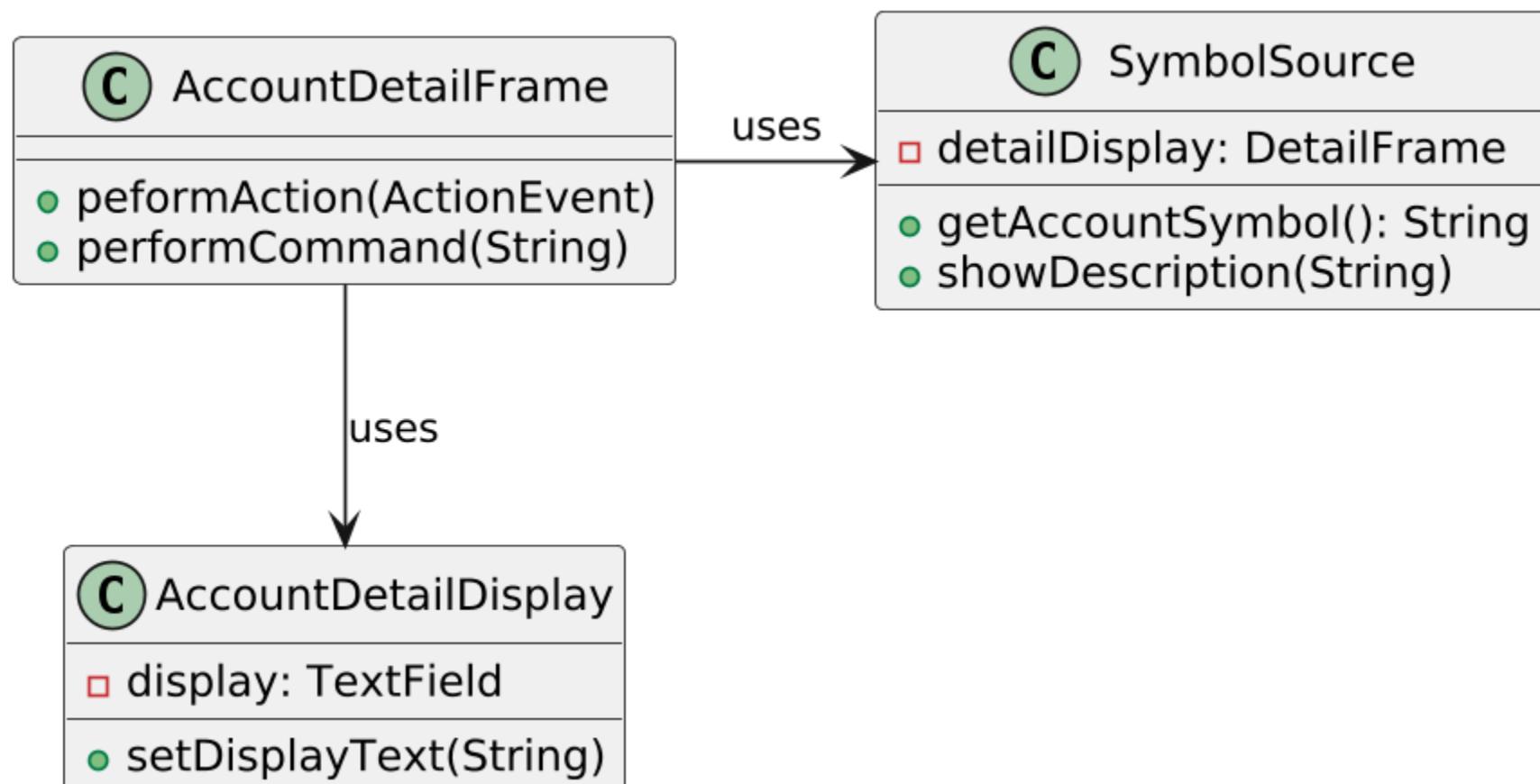
Subclass and Override

```
public class TestingAccountDetailFrame extends AccountDetailFrame {  
    String displayText = "";  
    String accountSymbol = "";  
  
    void showDescription(String d) {}  
  
    String getAccountSymbol() {  
        return accountSymbol;  
    }  
  
    void setDisplayText(String t) {  
        displayText = t;  
    }  
}
```

Test

```
public void testPerformCommand() {  
    TestingAccountDetailFrame frame = new TestingAccountDetailFrame();  
    frame.accountSymbol = "TEST";  
    frame.performCommand("project activity");  
    assertEquals("TEST: ", frame.displayText);  
}
```

VERANTWORTLICHKEITEN TRENNEN



Schwierige Argumente

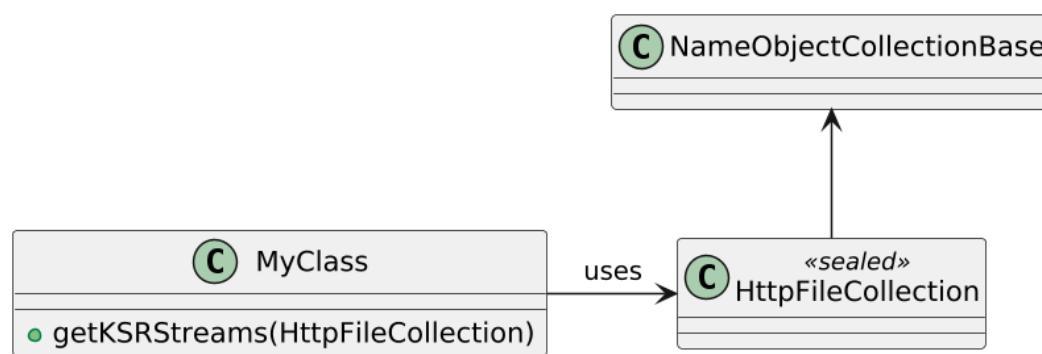
- zum Großteil in *Schwierige Klassen* abgehandelt
- Spezialfall: Harte SDK bzw. Library Kopplung
 - **MOMENT!** Das SDK ist doch nicht schlecht?!
 - **Teaser:** Das SDK nicht, eine starke Kopplung ggf. schon.

Folgende Beispiele zur SDK Kopplung wurde aus dem Buch 'Working Effectively With Legacy Code' übernommen

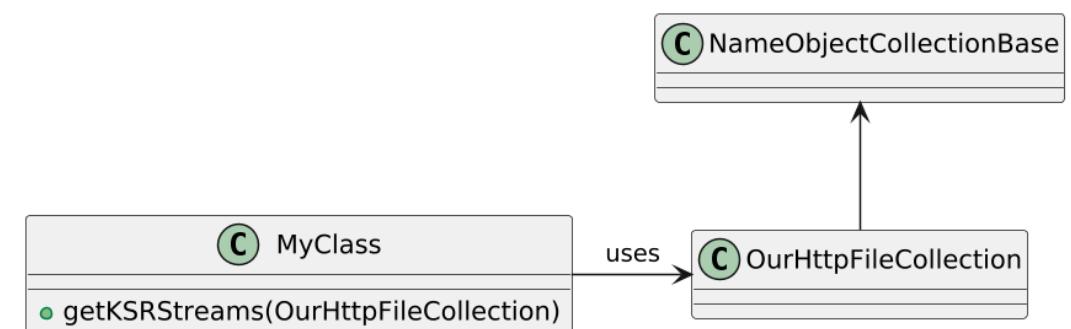
```
public IList getKSRStreams(HttpFileCollection files) {  
    ArrayList list = new ArrayList();  
    foreach(string name in files) {  
        HttpPostedFile file = files[name];  
        if (file.FileName.EndsWith(".ksr") ||  
            (file.FileName.EndsWith(".txt")  
                && file.ContentLength > MIN_LEN)) {  
            // ...  
            list.Add(file.InputStream);  
        }  
    }  
    return list;  
}
```

Problem: *HttpFileCollection* und *HttpPostedFile* sind Teil des SDKs, *sealed (final)* und können nicht instanziert werden. → *Extract Interface* und *Subclass and Override* ist nicht möglich.

Alt

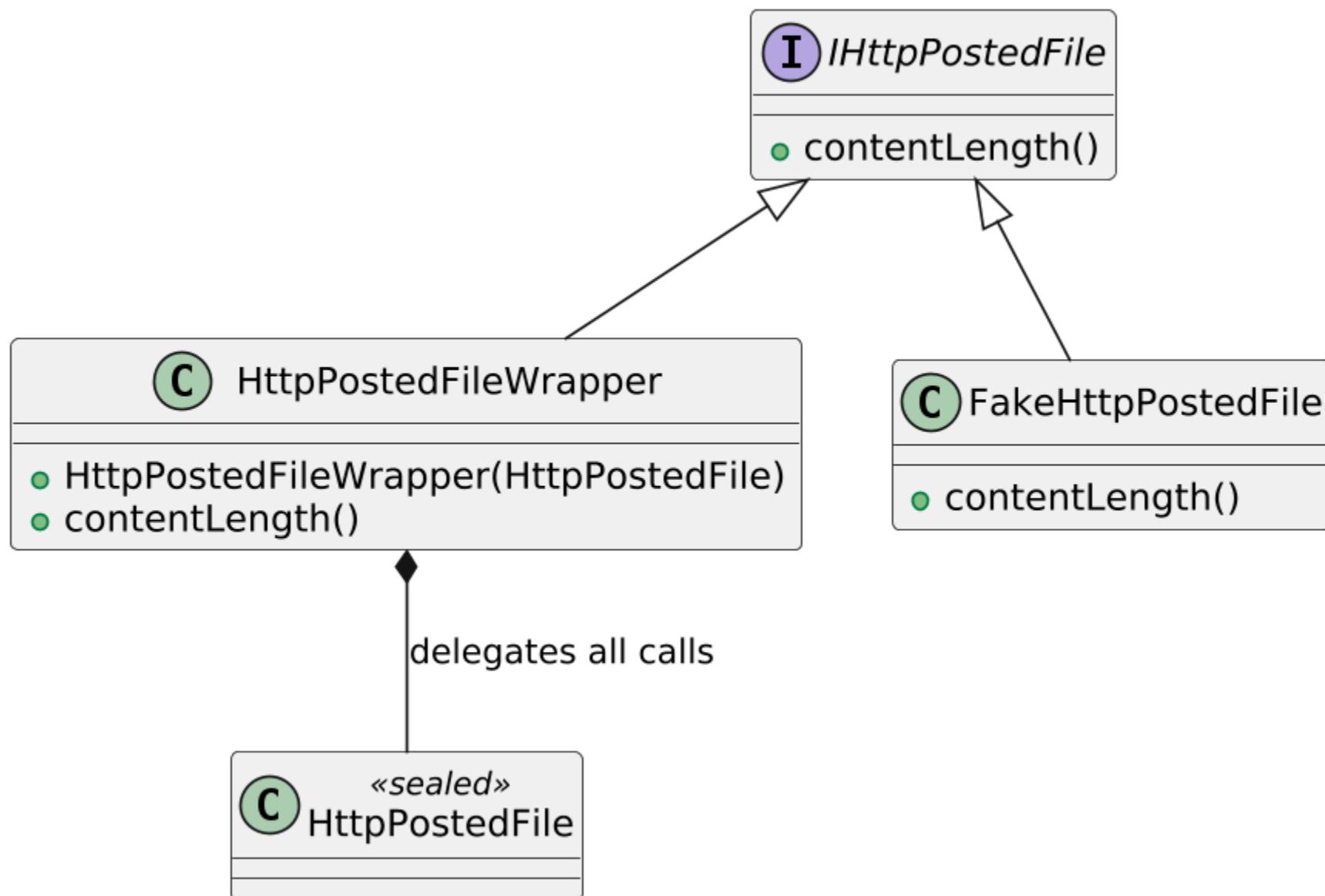


Neu



```
public IList getKSRStreams(OurHttpFileCollection files) {  
    ArrayList list = new ArrayList();  
    foreach(string name in files) {  
        HttpPostedFile file = files[name];  
        if (file.FileName.EndsWith(".ksr") ||  
            (file.FileName.EndsWith(".txt")  
             && file.ContentLength > MAX_LEN)) {  
            //...  
            list.Add(file.InputStream);  
        }  
    }  
    return list;  
}
```

SKIN AND WRAP THE API



```

1 public IList getKSRStreams(OurHttpFileCollection files) {
2     ArrayList list = new ArrayList();
3     foreach(string name in files) {
4         IHttpPostedFile file = files[name]; (1)
5         if (file.FileName.EndsWith(".ksr") ||
6             (file.FileName.EndsWith(".txt")))
7             && file.ContentLength > MAX_LEN)) {
8             ...
9             list.Add(file.InputStream);
10        }
11    }
12    return list;
13 }
```

- (1): Hier wird das neue Interface benutzt

ALTE METHODE DELEGIERT AN NEUE METHODE

```
public IList getKSRStreams(HttpFileCollection files) {  
    return getKSRStreams(new OurHttpFileCollection(files));  
}  
  
public IList getKSRStreams(OurHttpFileCollection files) {  
    // new method  
}
```

- so müssen die aktuellen Aufrufer nicht angepasst werden

VOR-/NACHTEIL

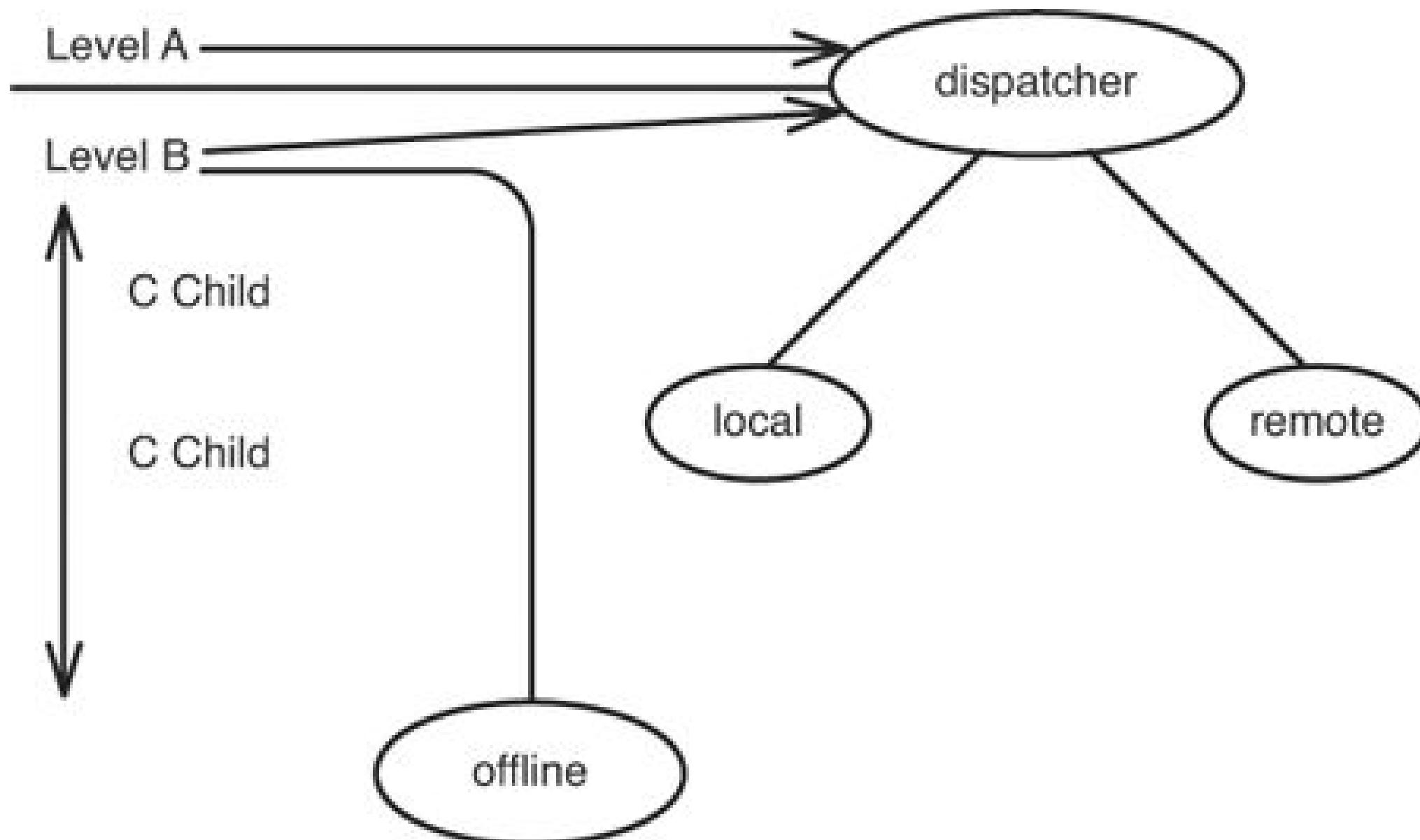
- (-) Komplexität steigt
- (-) alle *HttpPostedFile*-Instanzen müssen in *HttpPostedFileWrapper* gewrappt werden
 - (o) temporär kann die alte Methode die neue Methode aufrufen
- (+) saubere Auftrennung der Abhängigkeiten
- (+) Testabsicherung möglich

Besseres Code-Verständnis

Sketching / Skizzieren

- während dem Lesen eine Skizze erstellen
- es kann - **muss aber nicht** - UML sein
- hilft beim Diskutieren mit anderen
- Papier, Whiteboard, ...

BEISPIEL SKETCHING



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

Listing Markup

- Code ausdrucken
- **Verantwortlichkeiten** farblich oder mit Symbolen markieren
- Code einkreisen, der in eine Methode ausgelagert werden soll
 - die *Kopplungsanzahl* (coupling count) vermerken
- Nebeneffekte einer Änderung markieren
 - Codezeilen markieren, die geändert werden sollen
 - alle Variablen und Methoden markieren, die durch die Änderung betroffen sind
 - mit einer anderen Farbe alle Variablen und Methoden markieren, die durch die betroffenen Variablen und Methoden betroffen sind
 - wiederholen des letzten Schritts

Scratch Refactoring

- Code refactoren ohne zu committen und einzuchecken
 - → *trockenes Refactoring*
- Code-Verständnis erhöht sich
- man bekommt ein Gefühl, wohin es gehen kann
- evtl. ein Zeitfenster setzen (~30min)
- Nachteil: man könnte sich zu sehr auf den Weg versteifen und andere Lösungen nicht mehr sehen

Delete Unused Code

NOW!

Lösche unbenutzten Code. JETZT!

Keine Struktur vorhanden

Legacy Systeme tendieren dazu, keine Struktur oder die Struktur verloren zu haben.

- weil sich niemand um die Struktur kümmert
- weil das System sehr komplex ist und es lange dauert, bis man das große Bild sieht
- weil man nur beschäftigt ist, einen Notfall nach dem anderen abzuarbeiten

Telling the Story of the System

Die Geschichte des Systems erzählen

- "Wie sieht die Architektur des Systems aus?"
 - mit 2-3 Sätzen vereinfacht erklären, als ob man mit einer Person spricht, die nichts von dem System weiß
- man wird die - für einen selbst - wichtigsten Punkte nennen
- vermutlich wird die Vereinfachung nicht vollständig der Wahrheit entsprechen
- die Vereinfachung kann die Anleitung sein, wie das System besser wird
- in jedem Fall wird das Verständnis gefördert

Naked CRC / Nacktes CRC

- CRC = Class, Responsibilities and Collaborations
 - mit Karteikarten wird das System erklärt
 - jede Karteikarten entspricht einer Klasse
 - die Verantwortlichkeiten und zusammenhängende Klassen werden notiert
 - wenn eine Verantwortlichkeit nicht passt → durchstreichen und einer anderen Klasse hinzufügen oder eine neue Karte / Klasse anlegen

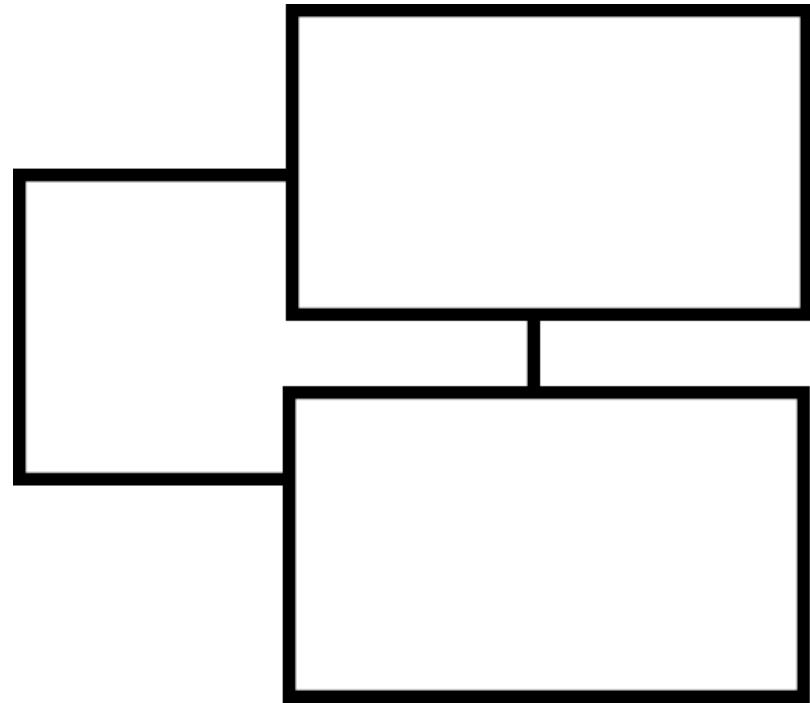
- Naked CRC / Nacktes CRC
 - die Karteikarten haben keine Bezeichungen
 - Karteikarten entsprechen Instanzen und nicht Klassen
 - Karteikarten werden gruppiert, um Zugehörigkeit zu zeigen
 - Schritt für Schritt werden die Karteikarten abgelegt
- ein physisches Bild manifestiert sich und ist schneller zu verstehen

BEISPIEL: NAKED CRC

Client Session



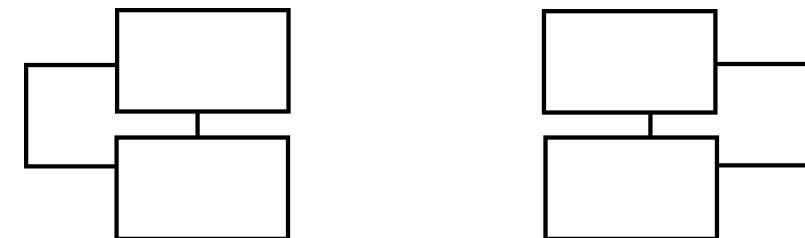
Client Session mit eingehender
und ausgehender Verbindung



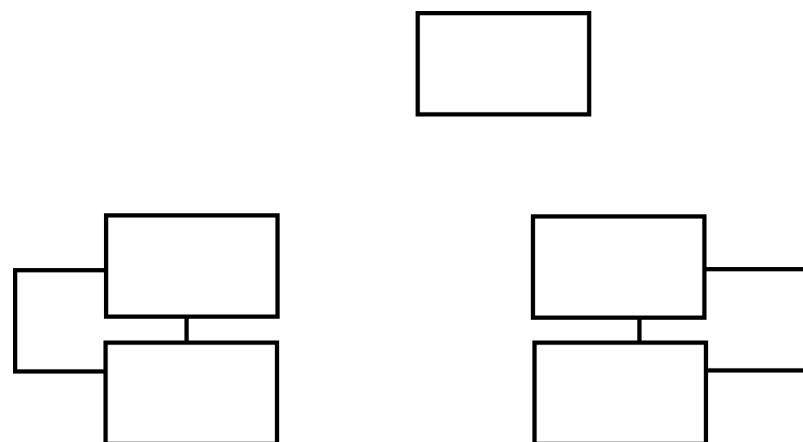
Client Session und Server Session



Server Session mit eingehender und ausgehender Verbindung



Logging-Service



Ändern ohne Fehler

Risikominimierung bei der Abhängigkeitsauflösung

oder

Arbeiten ohne Testabdeckung

Hyperaware Editing

- sich bewusst machen, ob eine Änderung folgen hat
 - z.B. Kommentaränderungen nicht, Variablenänderungen hingegen schon
- Test-Driven arbeiten falls möglich
 - direktes Feedback ist wichtig

Single-Goal Editing

- eine Änderung nach der anderen
- im Prozess der Änderung nicht andere Sachen 'mitfixen'
 - andere Sachen notieren, um sie *nach* der eigentlichen Änderung zu beheben
 - große Änderungen und viele Änderungen sind schwerer zu integrieren und korrekt zu implementieren, als kleine Änderungen und wenige Änderungen

Preserve Signatures

- wo es geht, die ursprüngliche Signatur beibehalten
- es minimiert den Änderungsaufwand und trägt somit unmittelbar zu *Single-Goal Editing* bei
- Signaturänderungen sind normalerweise die 'richtigen' Refactorings und sollten nicht ohne Testabdeckung durchgeführt werden

Lean on the Compiler

- intuitiv passiert das sehr häufig (in Java)
- nach einer Änderung kompilieren und die Fehler nach und nach beheben
 - IntelliJ geht noch einen Schritt weiter und zeigt 'Auswirkungen' an
- **Grenzen:** der Compiler findet natürlich nicht alles → Grenzen müssen bewusst sein
 - z.B. löschen einer überschriebenen Methode → Code kompiliert, kann sich jetzt aber ganz anders verhalten

Pair Programming

- 4 Augen sehen wesentlich mehr als 2
- 2 Köpfe haben mehr Ideen als einer
- beim Erklären der Problemstellung können Dinge klarer werden
- Wissen wird direkt geteilt
- ein Chirurg arbeitet normalerweise auch nie allein

Es wird einfach nicht besser

Was war/ist Deine Hauptmotivation Software-Entwickler zu werden?

Geld?

Dann ist es sowieso egal, ob es besser wird oder nicht. Das Geld bekommst Du auch so ;)

Um genau zu sein, ist dann ein nicht durchschaubares Legacy-System das Beste, was Dir passieren kann: es garantiert Dir Arbeit bis an Dein Lebensende.

Spaß am Entwickeln?

Dann ist es Einstellungssache. Du kannst Spaß am Refactoring entwickeln und Du wirst längerfristig 'Inseln' im Code generieren, bei denen es richtig Spaß macht, zu entwickeln.

Techniken zum Auflösen von Abhängigkeiten

- im Grunde sind es Refactorings ohne Tests
- die Idee ist, möglichst wenig zu ändern, um die Klassen und Methoden in eine Testabsicherung zu bekommen
- die Techniken dienen *nicht* zur Verbesserung des Designs
 - in vielen Fällen widersprechen sie etablierten Design-Regeln
- das 'richtige' Refactoring sollte gemacht werden, sobald die Testabsicherung steht

Adapt Parameter / Parameter anpassen

- | | |
|-------------|---|
| Wann | <ul style="list-style-type: none">• eine Methode lässt sich aufgrund ihrer Parameter nur schwer in eine Testabsicherung bringen• <i>Extract Interface</i> ist nicht möglich / zu aufwendig |
| Was | <ul style="list-style-type: none">• neue Abstraktion einführen<ul style="list-style-type: none">■ Beschränkung auf das Wesentliche■ Wrappen des alten Parameters |

BEISPIEL: ADAPT PARAMETER PROBLEM

```
class Server {

    void checkParams(HttpServletRequest req) {
        String[] vals =
            req.getParameterValues(PARAM_1);
        if (vals != null && vals.length > 0) {
            String currentParam = vals[0];
            // checks currentParam
        }
        vals = req.getParametersValues(PARAM_2);
        if (vals != null && vals.length > 0) {
            String currentParam = vals[0];
            // checks currentParam
        }
        // repeats the above code for other params
    }
}
```

HttpServletRequest ist ein Interface von Java EE.

MÖGLICHE LÖSUNGEN

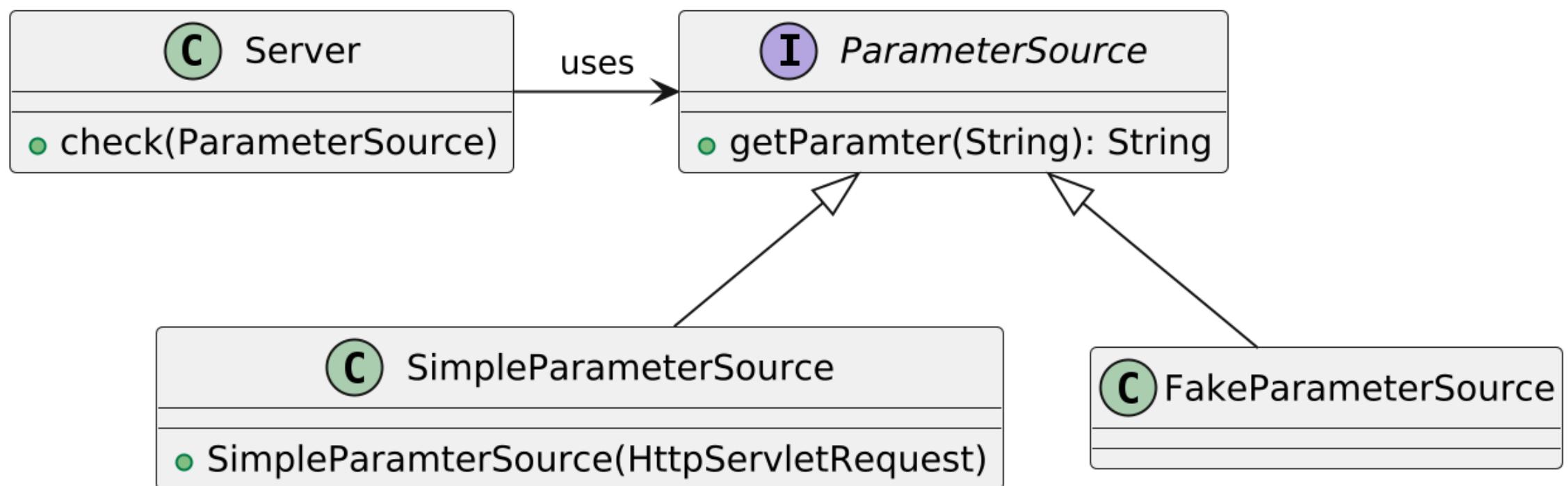
- Fake/Mock-Objekt
 - entweder selbst implementieren oder
 - ein Mocking-Framework benutzen (z.B. Mockito)
 - Problem
 - Mocking-Frameworks können teuer (langsam) sein
 - selbst implementieren kann unverständlich sein (das Interface hat >23 Methoden)
- oder Adapt Parameter verwenden*

ANWENDUNG: ADAPT PARAMETER

- im Endeffekt geht es um die RequestParameter
 - neue Klasse erstellen, die die alte wrappt
 - nur die notwendigen Methoden implementieren

```
void checkParams(ParameterSource paramSource) {  
    String currentValue = paramSource.getParameter(PARAM_1);  
    if (currentValue != null) {  
        // check currentValue  
    }  
    // redo for the other parameteres  
}
```

UML: ADAPT PARAMETER



VOR-/NACHTEILE: ADAPT PARAMETER

- (-) etwas mehr Änderungen am Legacy Code
- (-) Signatur ändert sich (*kein Preserve Signature!*)
- (+) Entkopplung des Library-Codes vom eigenen Code
- (+) einfacherer und verständlicherer Code
- (+) einfachere und verständlichere Tests
- (o) 'notwendiges' Refactoring schon zu Beginn

Expose Static Method

Wann

- eine Klasse lässt sich nicht (einfach) instanziieren
- eine Methode muss geändert werden
- die Methode nutzt keine Instanzvariablen

Was

- die Methode kopieren und statisch machen
- die ursprüngliche Methode delegiert den Aufruf an die statische

BEISPIEL: EXPOSE STATIC METHOD

```
class UserService {  
  
    // ...  
  
    void calculateOffset(int i, long l) {  
        Calculator calculator = new Calculator(l, true);  
        long offset = calculator.getOffset();  
        return i + offset;  
    }  
}
```

- die Methode hängt von keiner Instanzvariablen ab → statische Methode extrahieren

```
class UserService {  
  
    // ...  
  
    static int CalculateOffset(int i, long l) {  
        Calculator calculator = new Calculator(l, true);  
        long offset = calculator.getOffset();  
        return i + offset;  
    }  
  
    int calculateOffset(int i, long l) {  
        return CalculateOffset(i, l);  
    }  
}
```

VOR-/NACHTEILE: EXPOSE STATIC METHOD

- (+) Code verhält sich exakt so wie vorher
- (+) erfolgreiches *Preserve Signature*
- (+) fragwürdige Verantwortlichkeit herausgestellt

Break Out Method Object

Methoden-Objekt herausbrechen

Wann

- wenn die Klasse nicht einfach instanziierbar ist
 - wenn es sich um eine lange Methode handelt
 - und/oder wenn die Methode Instanzvariablen benutzt
- wenn *Expose Static Method* nicht funktioniert

Was

- die Methode in eine eigene Klasse auslagern
- evtl. die Parameter als Instanzvariablen anlegen
- die alte Methode an die neue delegieren lassen

BEISPIEL: BREAK OUT METHOD OBJECT

```
class UserService {  
  
    boolean check(User u) {  
        // ...  
        if (!checkAge(u)) return false;  
        // ...  
        return true;  
    }  
  
    private boolean checkAge(User u) {  
        // ...  
    }  
}
```

Schritt 1: extrahieren der Methode → Parameter als Instanzvariablen und ursprüngliche Klasse mit übergeben

```
class UserCheck {

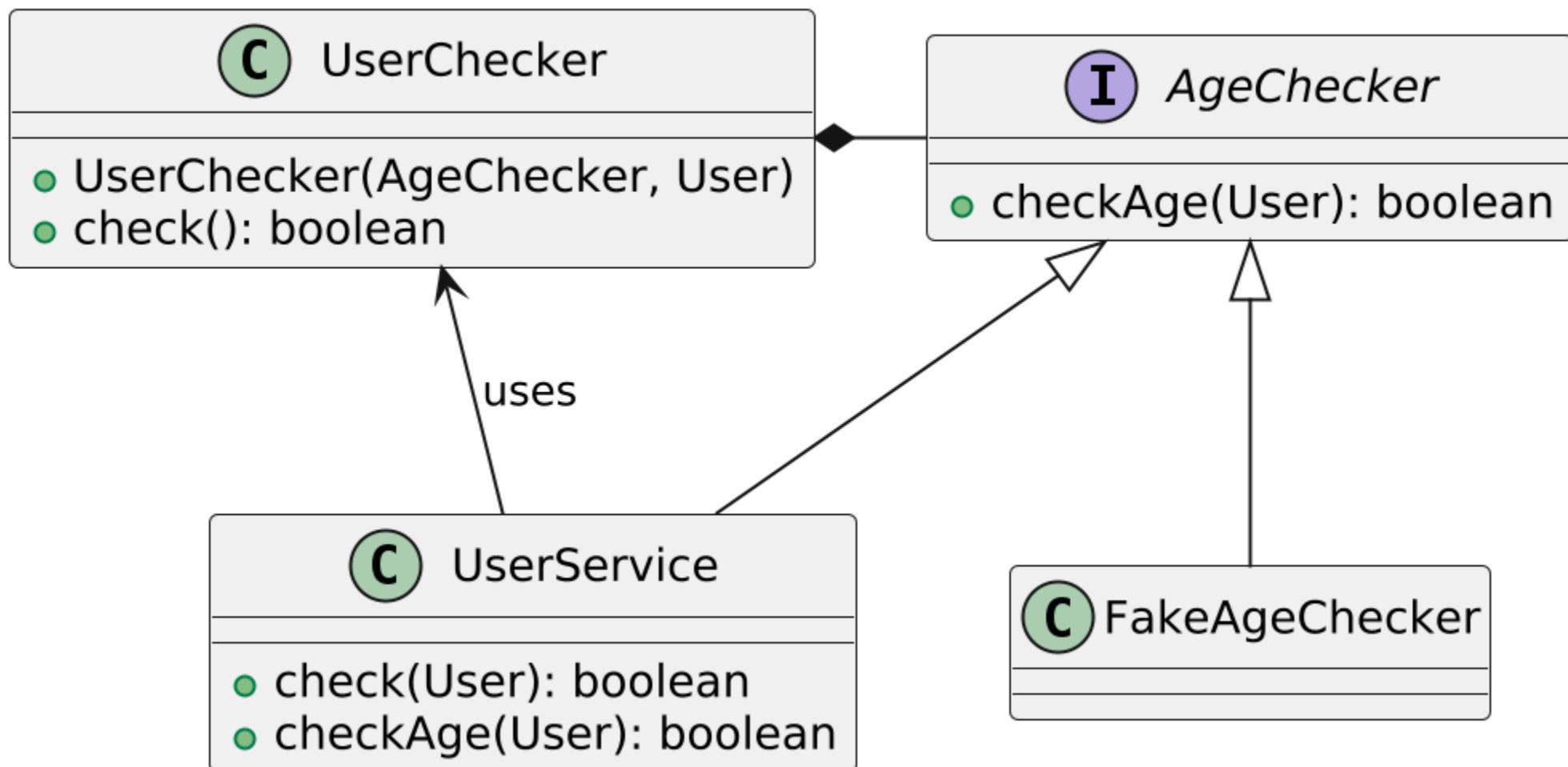
    private UserService userService;
    private User user;

    UserChecker(UserService userService,
                User user) {
        this.user = user;
        this.userService = userService;
    }

    boolean check() {
        // ...
        if (!userService.checkAge(u))
            return false;
        // ...
        return true;
    }

}
```

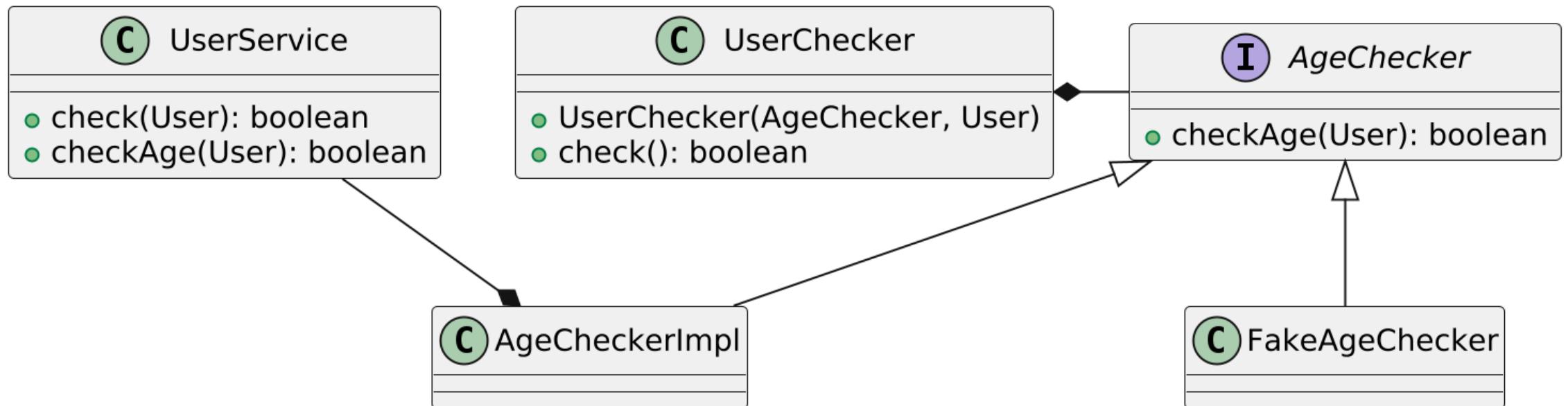
Schritt 2: Extract Interface



Schritt 3: Testen

```
@Test  
public void testCheck() {  
    FakeAgeChecker ageChecker = new FakeAgeChecker();  
    User user = new User();  
    UserChecker checker =  
        new UserChecker(ageChecker, user);  
    boolean result = checker.check();  
    // do evaluation  
}
```

Ausblick: 'richtiges' Refactoring



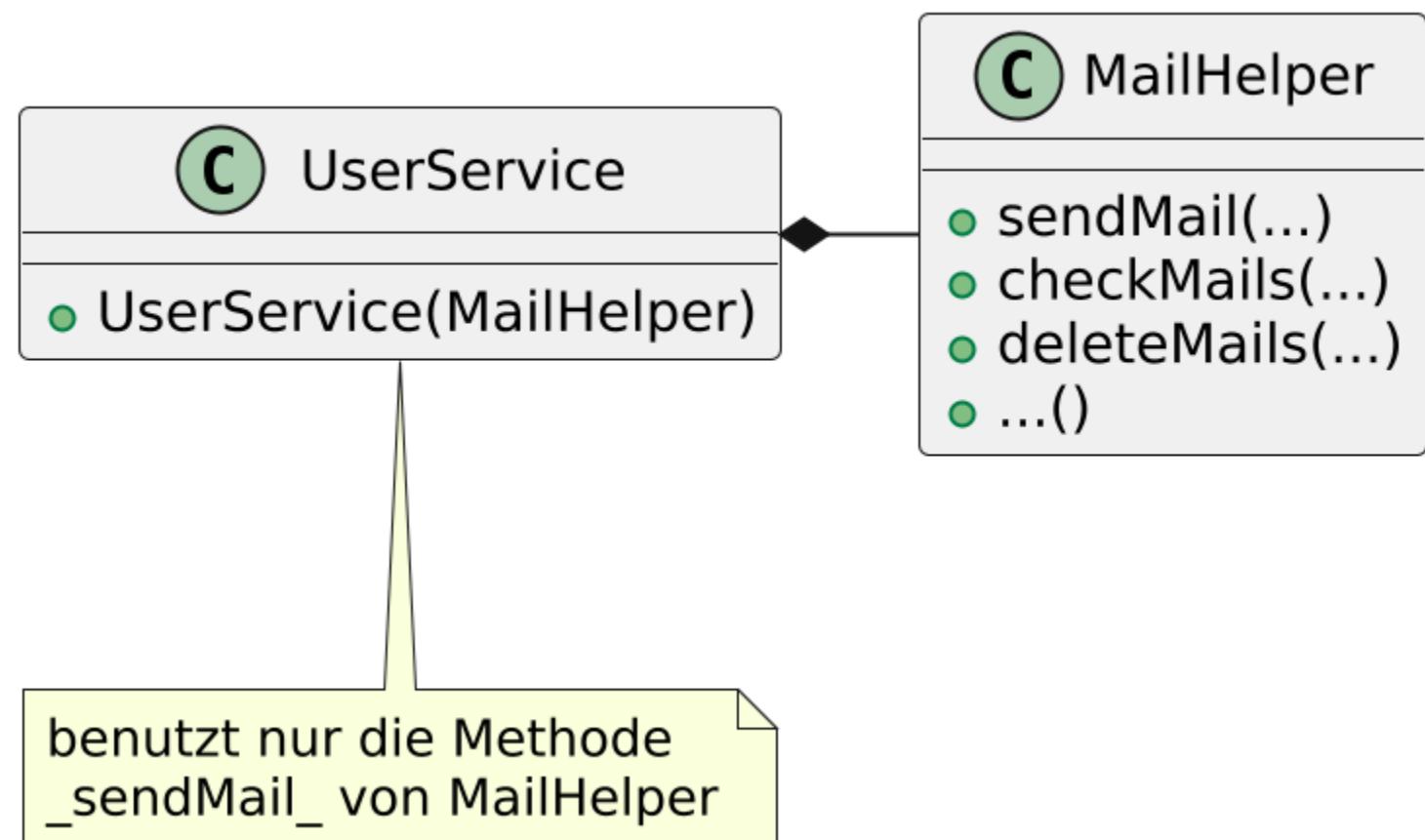
VOR-/NACHTEILE: BREAK OUT METHOD OBJECT

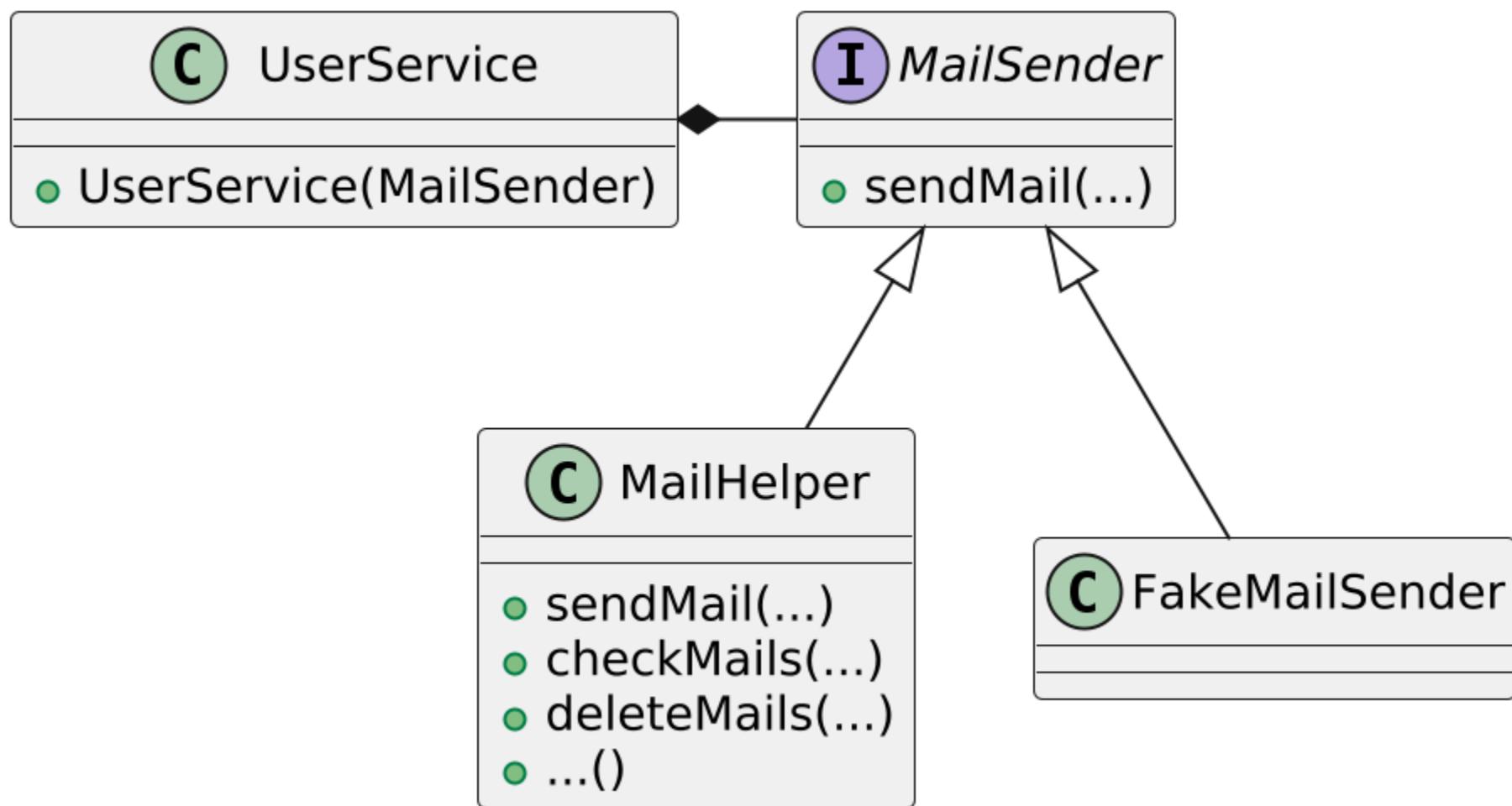
- (-) ohne weiteres Refactoring ein seltsamer Stand
- (o) evtl. Grundlage für neue Konzepte
- (+) Isolierung von Verantwortlichkeiten

Extract Interface

- | | |
|-------------|--|
| Wann | <ul style="list-style-type: none">• ein Objekt als Argument (im Konstruktor oder einer Methode) lässt sich nicht leicht erzeugen und/oder als Fake-Objekt implementieren• wir haben die Kontrolle über die Klasse des Objekts<ul style="list-style-type: none">■ d.h. es ist keine SDK- oder 3rd-Party-Library-Klasse |
| Was | <ul style="list-style-type: none">• die notwendigen Methoden in ein Interface extrahieren• die ursprüngliche Klasse implementiert das Interface• in der ursprünglichen Methode / Konstruktor wird die ursprüngliche Klasse mit dem Interface ersetzt |

BEISPIEL: EXTRACT INTERFACE





VOR-/NACHTEILE: EXTRACT INTERFACE

- (+) 'sicheres' Refactoring, da der Compiler unterstützt
- (+) bessere Abstraktion
- (+) IDE-Unterstützung (z.B. IntelliJ)
- (-) ggf. Änderung der Signatur (kein *Preserve Signature*)

Subclass and Override Method

Wann

- eine Methode muss geändert / getestet werden, die im Test nicht sichtbar ist
- **oder** eine Methode soll aufgrund ihrer Nebeneffekte im Test nicht ausgeführt werden, wird aber von der zu testenden / ändernden Methode aufgerufen
- von der Klasse der Methode kann abgeleitet werden

Was

- Subklassen für den Test anlegen,
 - die nun entweder die Methode vollständig nach außen sichtbar machen
 - oder die entsprechende Methode mit einem leeren Body überschreiben

BEISPIEL: SUBCLASS AND OVERRIDE METHOD

```
class UserService {  
    public void doSomething(User u) {  
        // ...  
        expensiveDbCall(u); // not necessary for test  
        // ...  
    }  
  
    private void expensiveDbCall(User u) {  
        // ...  
    }  
}
```

```
class UserService {  
    public void doSomething(User u) {  
        // ...  
        expensiveDbCall(u); // not necessary for test  
        // ...  
    }  
  
    protected void expensiveDbCall(User u) {  
        expensiveDbCallInternal(u);  
    }  
  
    private void expensiveDbCallInternal(User u) {  
        // ...  
    }  
}
```

BEISPIEL: FUNKTIONALITÄT 'NULLEN'

```
class UserServiceForTest extends UserService {  
    public void doSomething(User u) {  
        // ...  
        expensiveDbCall(u); // not necessary for test  
        // ...  
    }  
  
    @Override  
    protected void expensiveDbCall(User u) {  
        // empty body  
    }  
}
```

```
class UserServiceTest {  
  
    @Test  
    public void doSomethingTest() {  
        UserService userService = new UserServiceForTest();  
        userService.doSomething();  
        // ...  
    }  
  
}
```

BEISPIEL: FUNKTIONALITÄT TESTEN

```
class UserServiceForTest extends UserService {  
    public void doSomething(User u) {  
        // ...  
        expensiveDbCall(u);  
        // ...  
    }  
  
    public void actualTestMethod(User u) {  
        expensiveDbCall(u);  
    }  
}
```

```
class UserServiceTest {  
  
    @Test  
    public void expensiveDbCallTest() {  
        UserServiceForTest userService = new UserServiceForTest();  
        FakeUser user = new FakeUser();  
        userService.actualTestMethod(user);  
        // ...  
    }  
  
}
```

VOR-/NACHTEILE: SUBCLASS AND OVERRIDE METHOD

- (+) einfaches aufrufen oder *nullen* von privaten Methoden
- (-) *Preserve Signature* nicht eingehalten
 - da die Methode auf die Klasse beschränkt ist und die neue Methode alles delegiert, sollte es keine Probleme geben
- (-) Komplexität in den Tests steigt (v.a. durch die zusätzliche Subklassen)

Extract and Override Factory Method

Wann

- ein Konstruktor initialisiert Instanzvariablen, die wir zum Testen brauchen
- oder ein Konstruktor initialisiert Instanzvariablen, die wir nicht brauchen und die bei der Initialisierung Nebeneffekte haben
- *Parametrize Constructor* ist nicht möglich

Was

- die Logik zur Initialisierung der Instanzvariablen in eine Factory-Methode auslagern und in einer Subklasse überschreiben
 - **Achtung:** Funktioniert nicht in C++, da im Konstruktor nicht auf überschriebene virtuelle Methoden der Subklassen zugegriffen werden kann

BEISPIEL: EXTRACT AND OVERRIDE FACTORY METHOD

→ *RepStore* soll mit einem FakeObjekt ersetzt werden

```
public class UserService {

    public UserService() {
        // ...
        CareTaker cTaker = new CareTaker(DEFAULT_CONFIG);
        Rep r = cTaker.getRep();
        Store s = new Store(r.getDefault());
        this.rs = new SimpleRepStore(r, s);
        // ...
    }
}
```

```
public class UserService {

    public UserService() {
        // ...
        CareTaker cTaker = new CareTaker(DEFAULT_CONFIG);
        Rep r = cTaker.getRep();
        Store s = new Store(r.getDefault());
        this.rs = makeRepStore(r, s);
        // ...
    }

    protected RepStore makeRepStore(Rep r, Store s) {
        return new SimpleRepStore(r, s);
    }
}
```

BEISPIEL: EIN FAKE-OBJEKT ZURÜCKGEBEN

```
public class UserServiceForTest {

    @Override
    protected RepStore makeRepStore(Rep r, Store s) {
        return new FakeRepStore();
    }

    public FakeRepStore getRepStore() {
        if (this.rs == null) return null;
        return (FakeRepStore)this.rs;
    }
}
```

```
public class UserServiceTest {  
  
    @Test  
    public void testDoSomething() {  
        UserServiceForTest service = new UserServiceForTest();  
        service.doSomething();  
        FakeRepStore store = service.getRepStore();  
        // evaluate properties on store  
    }  
}
```

BEISPIEL: DIE ERZEUGUNG (UND DAMIT DIE NEBENEFFEKTE) UNTERBINDEN

```
public class UserServiceForTest {  
  
    @Override  
    protected RepStore makeRepStore(Rep r, Store s) {  
        return null;  
    }  
}
```

```
public class UserServiceTest {  
  
    @Test  
    public void testDoSomething() {  
        UserServiceForTest service = new UserServiceForTest();  
        service.doSomething();  
        // ...  
    }  
}
```

VOR-/NACHTEILE: EXTRACT AND OVERRIDE FACTORY METHOD

- (+) einfaches eigenes Erzeugen oder *nullen* von Instanzvariablen
- (-) Komplexität in den Tests steigt (v.a. durch die zusätzliche Subklassen)

Extract and Override Getter

Wann

- ein Konstruktor initialisiert Instanzvariablen, die wir zum Testen brauchen
- *Extract and Override Factory Method* ist nicht möglich
 - z.B. in C++
- *Parametrize Constructor* ist nicht möglich

Was

- *getter*-Methode für die Instanzvariable anlegen (protected)
- Zugriff auf die Instanzvariable nur über den *getter*
- den *getter* in eine Subklasse überschreiben

BEISPIEL: EXTRACT AND OVERRIDE GETTER

```
// .h file
class UserService
{
    private:
        RepStore *repStore;
    public:
        UserService();
}

// .cpp file
UserService::UserService()
{
    // ...
    repStore = new SimpleRepStore(r, s);
    // ...
}
```

Schritt 1: Lazy-Getter einführen

```
// .h file
class UserService
{
    private:
        RepStore *repStore;
    protected:
        RepStore* getRepStore();
    public:
        UserService();
}

// .cpp file
UserService::UserService()
: repStore(0)
{
    // ...
}

RepStore* UserService::getRepStore() {
    if (repStore != 0) return repStore;
    // ...
    repStore = new SimpleRepStore(r, s);
    return repStore;
}
```

Schritt 2: Neue Subklasse für den Test

```
class UserServiceForTest : public UserService
{
public:
    RepStore* getRepStore() {
        return &fakeRepStore;
    }

    FakeRepStore* fakeRepStore;
}
```

Schritt 3: Testen

```
TEST(someVariable, UserService)
{
    UserServiceForTest* service = new UserServiceForTest();
    // ...
    FakeRepStore* repStore =
        dynamic_cast<FakeRepStore*> (service->getRepStore());
    // do evaluation on repStore
}
```

VOR-/NACHTEILE: EXTRACT AND OVERRIDE GETTER

- (+) einfaches eigenes Erzeugen von Instanzvariablen
- (-) Komplexität in den Tests steigt (v.a. durch die zusätzliche Subklassen)
- (-) alles muss über den neuen *getter* passieren → fehleranfällig

Supersede Instance Variable

Wann

- eine Instanzvariable wird für den Test gebraucht
- es gibt keinen *setter* für die Variable
- *Parametrize Constructor / Parametrize Method* ist nicht möglich
- **Alternativen:** *Extract and Override Factory Method, Extract and Override Getter*

Was

- eine *supersede*-Methode anlegen (z.B. *supersedeXyz*)
 - ggf. *protected* und nur per Test-Subklasse darauf zugreifen
- im Test die *supersede*-Methode verwenden, um den ursprünglichen Wert zu überschreiben

BEISPIEL: SUPERSEDE INSTANCE VARIABLE

→ *RepStore* soll mit einem FakeObjekt ersetzt werden

```
public class UserService {  
  
    private RepStore rs;  
  
    public UserService() {  
        // ...  
        CareTaker cTaker = new CareTaker(DEFAULT_CONFIG);  
        Rep r = cTaker.getRep();  
        Store s = new Store(r.getDefault());  
        this.rs = new SimpleRepStore(r, s);  
        // ...  
    }  
}
```

Schritt 1: supersede-Methode anlegen

```
public class UserService {  
  
    private RepStore rs;  
  
    public UserService() {  
        // ...  
    }  
  
    public void supersedeRepStore(RepStore rs) {  
        this.rs = rs;  
    }  
}
```

Schritt 2: Testen

```
@Test  
public void testUserService() {  
    UserService service = new UserService();  
    FakeRepStore store = new FakeRepStore();  
    service.supersedeRepStore(store);  
    // ...  
}
```

VOR-/NACHTEILE: SUPERSEDE INSTANCE VARIABLE

- (-) Überschreiben einer vorher nicht-überschreibbaren Variable möglich
 - das kann mit *protected* eingeschränkt werden
 - *supersede* deutet immerhin darauf hin, dass die Methode nicht missbraucht werden sollte → noch deutlicher:
supersedeXyzForTestOnly
- (+) einfaches setzen einer Instanzvariable
- (+) alte Funktionalität bleibt vollständig erhalten

Parametrize Constructor

- | | |
|-------------|--|
| Wann | <ul style="list-style-type: none">• ein Konstruktor initialisiert eine (Instanz-)Variable• die (Instanz-)Variable wird für den Test gebraucht bzw. muss überschrieben werden |
| Was | <ul style="list-style-type: none">• einen neuen Konstruktor anlegen, der zusätzlich ein Objekt für die (Instanz-)Variable entgegennimmt• der alte Konstruktor ruft den neuen Konstruktor mit dem ursprünglichen Wert auf• der neue Konstruktor kann im Test verwendet werden<ul style="list-style-type: none">■ z.B. um ein Fake-Objekt zu übergeben |

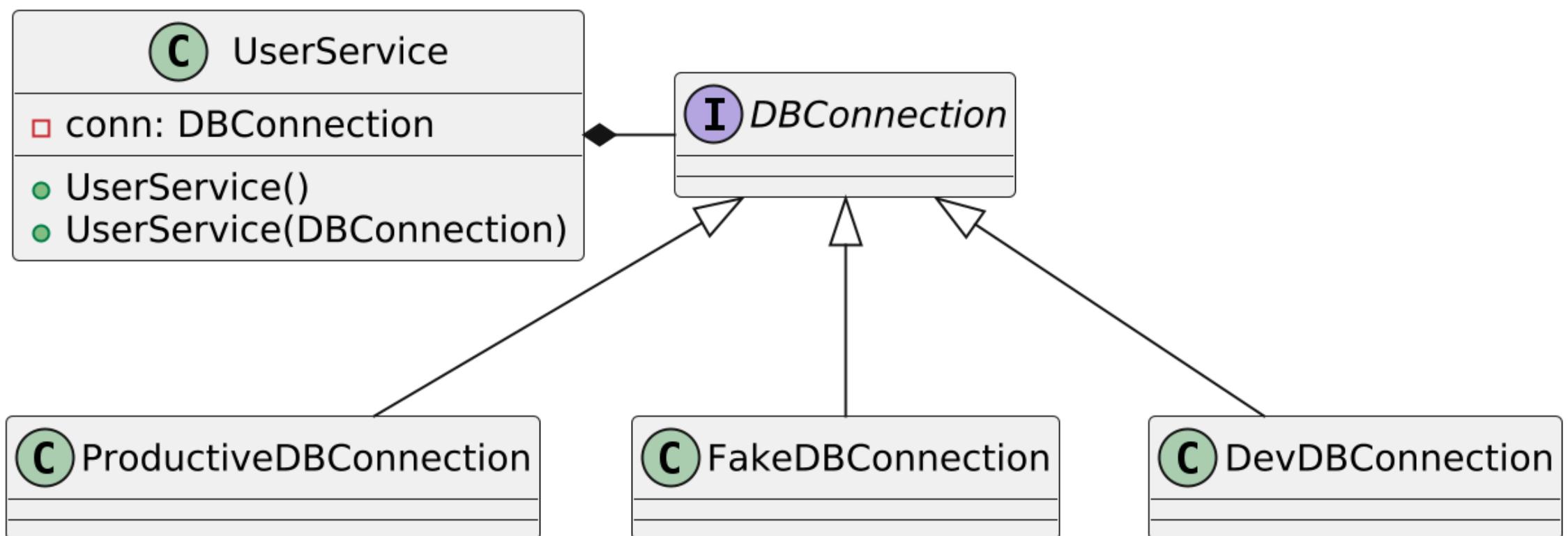
BEISPIEL: PARAMETRIZE CONSTRUCTOR

```
class UserService {  
  
    DBConnection conn;  
  
    UserService() {  
        conn = new DBConnection();  
        // ...  
    }  
}
```

Schritt 1: Neuen Konstruktor anlegen und alten Konstruktor delegieren lassen

```
class UserService {  
  
    DBConnection conn;  
  
    UserService(DBConnection conn) {  
        this.conn = conn;  
        // ...  
    }  
  
    UserService() {  
        this(new DBConnection());  
    }  
}
```

Schritt 2: ggf. Extract Interface



Schritt 3: Testen

```
@Test  
public void testUserService() {  
    FakeDBConnection conn = new FakeDBConnection();  
    UserService service = new UserService(conn);  
    //...  
}
```

VOR-/NACHTEILE: PARAMETRIZE CONSTRUCTOR

- (+) sehr saubere Möglichkeit, Abhängigkeiten zu brechen
- (+) alte Funktionalität bleibt vollständig erhalten

Parametrize Methode

Wann

- ein Methode initialisiert eine Variable
- die Variable wird für den Test gebraucht bzw. muss überschrieben werden

Was

- einen neuen Methode anlegen, die zusätzlich ein Objekt für die Variable entgegennimmt
- die alte Methode ruft die neue Methode mit dem ursprünglichen Wert auf
- die neue Methode kann im Test verwendet werden
 - z.B. um ein Fake-Objekt zu übergeben

BEISPIEL: PARAMETRIZE METHOD

```
class UserService {  
  
    void writeUserToDB(User u) {  
        DBConnection conn = new DBConnection(PROD_USER);  
        conn.write(u);  
    }  
}
```

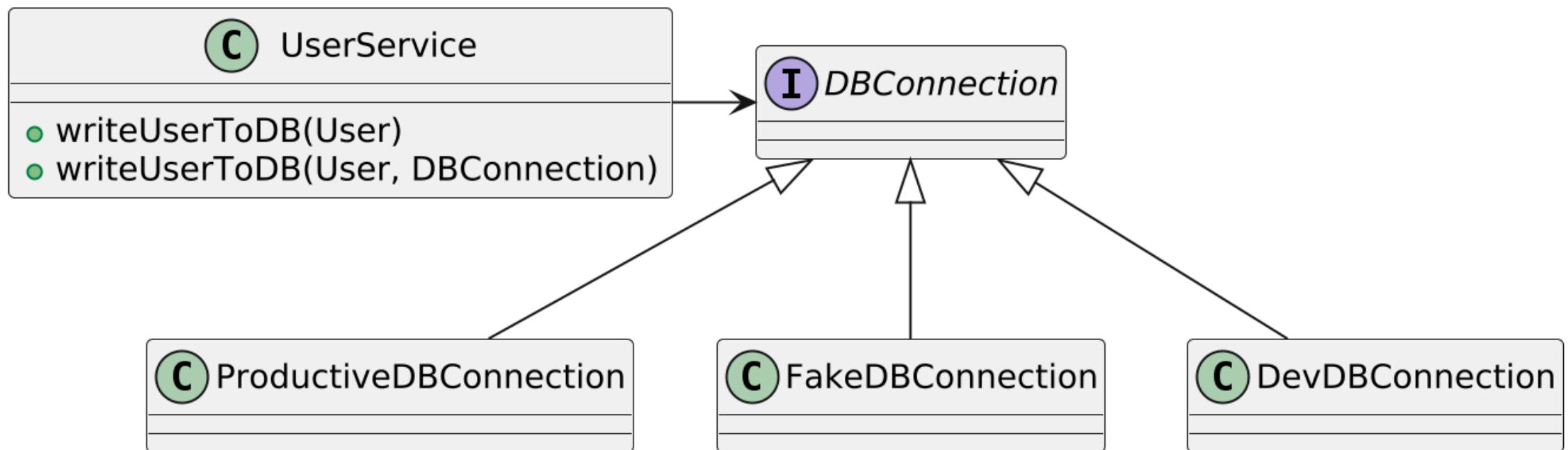
Schritt 1: Neue Methode anlegen und alte Methode delegieren lassen

```
class UserService {

    void writeUserToDB(User u, DBConnection conn) {
        conn.write(u);
    }

    void writeUserToDB(User u) {
        writeUserToDB(u, new DBConnection(PROD_USER));
    }
}
```

Schritt 2: ggf. Extract Interface



Schritt 3: Testen

```
@Test  
public void testWriteToDB() {  
    FakeDBConnection conn = new FakeDBConnection();  
    UserService service = new UserService();  
    service.writeToDB(new User(), conn);  
    //...  
}
```

VOR-/NACHTEILE: PARAMETRIZE METHOD

- (+) sehr saubere Möglichkeit, Abhängigkeiten zu brechen
- (+) alte Funktionalität bleibt vollständig erhalten

Literatur

Working Effectively With Legacy Code von M.C. Feathers

