Technische Hochschule Ingolstadt
Fakultät für Elektrotechnik und Informatik

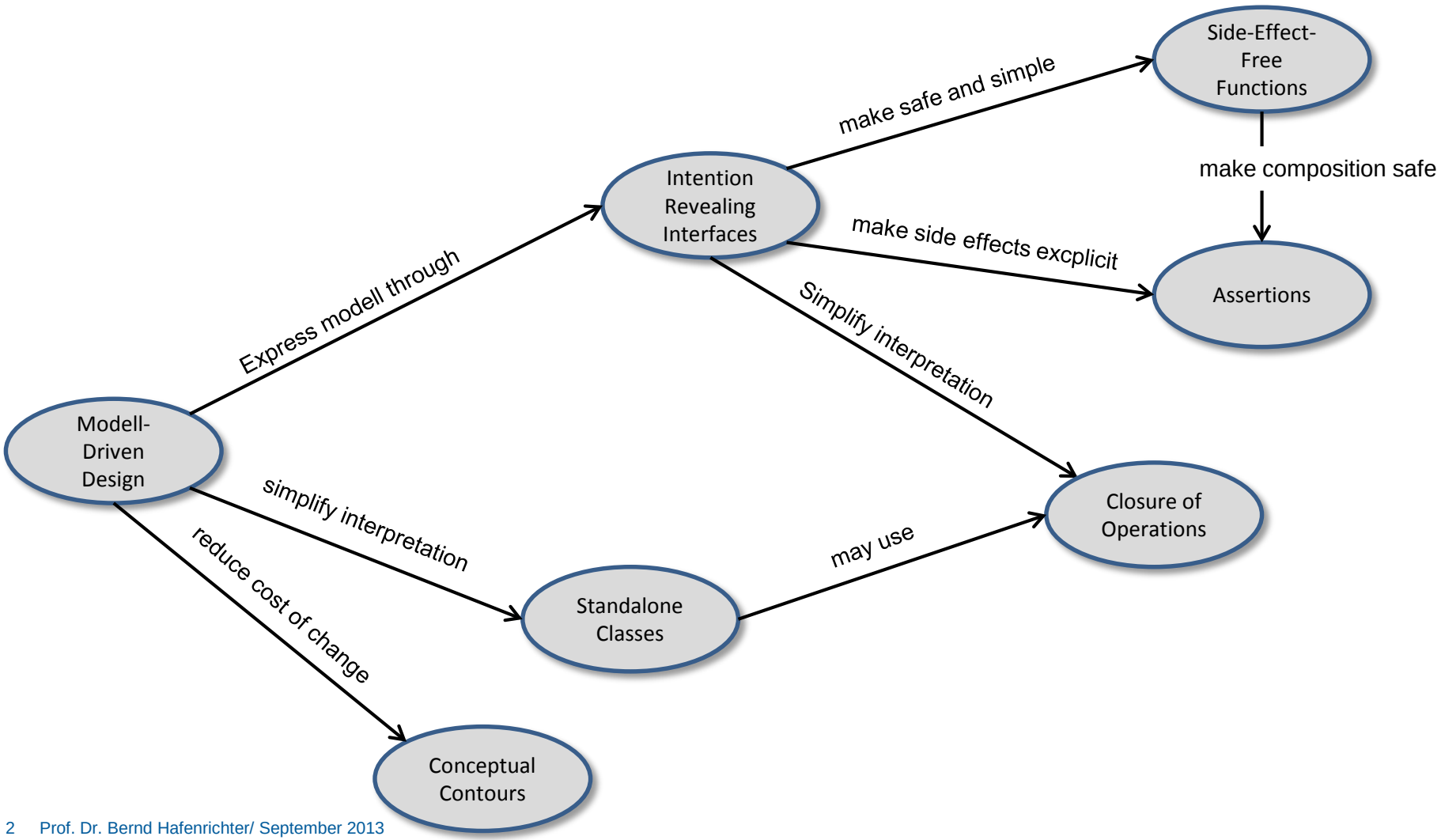
Zukunft in Bewegung

Domain-Driven Design
Supple Design

Architektur- und Entwurfsmuster der Softwaretechnik

Prof. Dr. Bernd Hafenrichter 06.03.2015

Design-Techniken: Supple Design



Intention Revealing Interfaces

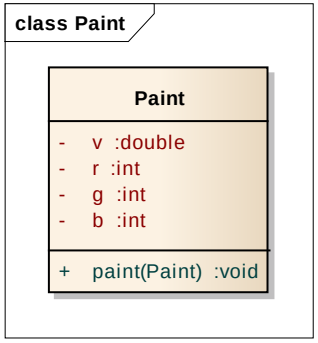
Problem

- Wenn ein Entwickler die Implementierung einer Funktion kennen muss um Sie anzuwenden geht die Kapselung verloren
- Muss der Zweck eines Objekts aus dessen Namen abgeleitet werden kann eine falsche Interpretation entstehen
- Dies führt zu konzeptuellen Fehler, da verschiedene Entwickler auf Basis verschiedener Sachverhalte arbeiten.

Lösung

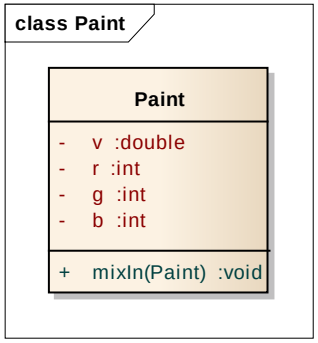
- Benennen Sie Klassen und Operationen um deren Zweck bzw. Wirkung zu beschreiben, ohne Bezug auf die konkrete Realisierung zu nehmen.
- Dadurch werden Interna‘s von den Nutzern verborgen
- Die Namen sollen stimmig in Bezug auf die Ubiquitous Language sein
- Schreiben Sie einen Verhaltenstest bevor Sie die Methode implementieren.

Intention Revealing Interfaces




```
public void paint( Paint paint ) {

    this.v = paint.v + v;

    this.r = (int)( 0.5 * this.r  + 0.5 * paint.r );
    this.g = (int)( 0.5 * this.g  + 0.5 * paint.g );
    this.b = (int)( 0.5 * this.b  + 0.5 * paint.b );
}
```

Umbenennen der Methode




```
public void mixIn( Paint paint ) {

    this.v = paint.v + v;

    this.r = (int)( 0.5 * this.r  + 0.5 * paint.r );
    this.g = (int)( 0.5 * this.g  + 0.5 * paint.g );
    this.b = (int)( 0.5 * this.b  + 0.5 * paint.b );
}
```

Side-Effect-Free Functions

Side-Effect

- Eine Funktion welche den Zustand eines Systems verändert so dass nachfolgende Operationen betroffen sind

Problem

- Ein Cascade von Funktionsaufrufen mit Seiten-Effekten kann in Ihrer Wirkung schwer vorhergesagt werden.
- Ein Entwickler muss die Implementierung sowie die Implementierung aller verwendeten Funktionen um das Ergebnis vorherzusagen
- Der Nutzen der Abstraktion/Interfaces geht verloren wenn Entwickler diese Barriere durchbrechen müssen

Side-Effect-Free Functions

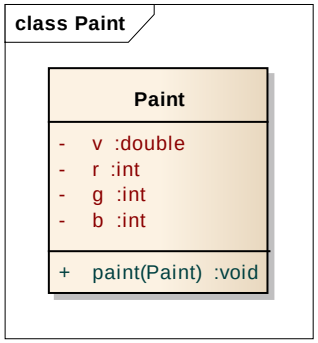
Side-Effect

- Eine Funktion welche den Zustand eines Systems verändert so dass nachfolgende Operationen betroffen sind

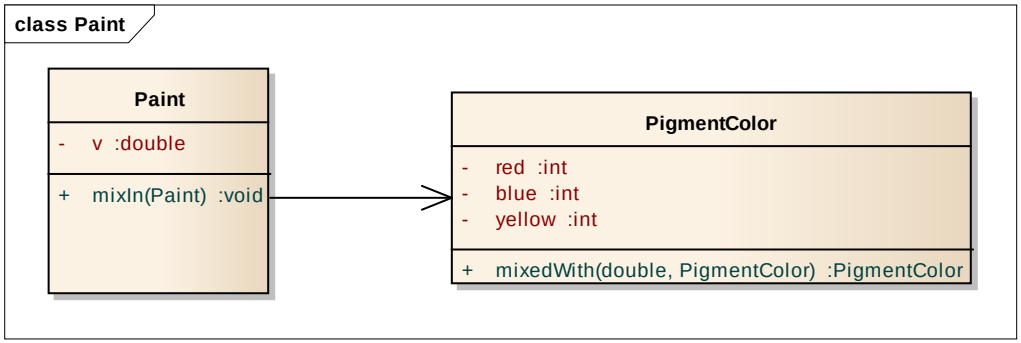
Lösung

- Platzieren Sie soviel Logik wie möglich in Funktionen welche Ergebnisse ohne beobachtbare Seiten-Effekte liefern
- Isolieren Sie „Commandos“ (=verändern den Systemzustand) in einfachen Operationen welche kein Domäneninformation zurückliefern
- Kontrollieren Sie Seiteneffekte indem Sie komplexe Logik in Value-Objects verschieben falls das Konzept sich selbst repräsentiert

Side-Effect-Free-Functions



Auslagern der Farbattribute in ein Value-Object „PigmentColor“




```
public void mixIn( Paint paint ) {

  this.v     = this.v + paint.v;
  this.color = color.mixWith(paint.getColor());
}
```

Fragement: Klasse Paint

```
public PigmentColor mixWith( PigmentColor color ) {

  PigmentColor result = new PigmentColor();

  result.r = (int)( 0.5*this.r  + 0.5*color.r );
  result.g = (int)( 0.5*this.g  + 0.5*color.g );
  result.b = (int)( 0.5*this.b  + 0.5*color.b );

  return result;
}
```

Fragement: Klasse PigmentColor

Assertions

Problem

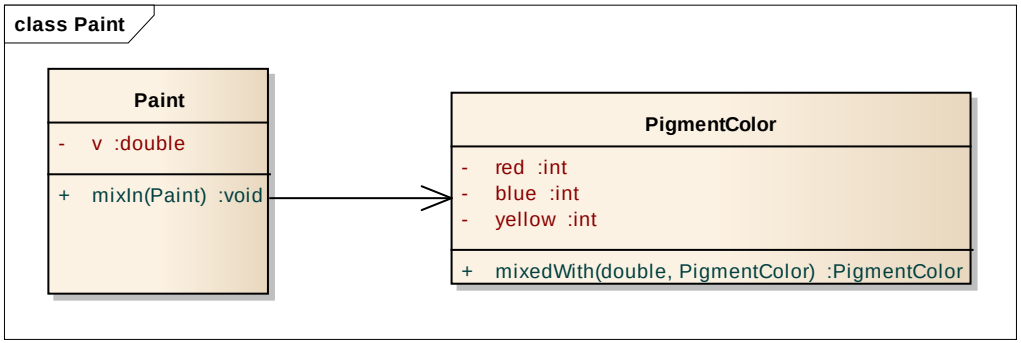
- Operationen mit Seiten-Effekten können nicht ganz ausgeschlossen werden
- Sind die Seiten-Effekte nur implizit durch die Implementierung definiert muss jeder Entwickler die Implementierung kennen.

Lösung

- Machen Sie die Wirkung einer Funktion/Command explizit
- Definieren Sie die erwarteten Seiten-Effekte mit Hilfe von Assertions
  - Post-Conditions für Operations
  - Invarianten für Klassen und Aggregate
- Kann eine Assertion nicht direkt implementiert werden müssen automatisierte Testfälle entwickelt werden.
- Geben Sie die Assertions innerhalb von Modellen und Diagrammen an

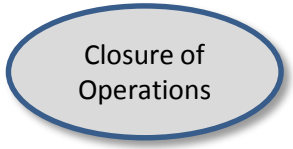
Assertions

Assertions




```
context Paint:mixIn(p : Paint) : void
  post: self.v = self.v@pre + p.v
```

Closure of Operations



Motivation

- Die Addition von natürlichen Zahlen liefert wiederum eine natürliche Zahl
- Das dies immer gilt kann man sagen die Addition auf natürlichen Zahlen ist abgeschlossen unter der Addition

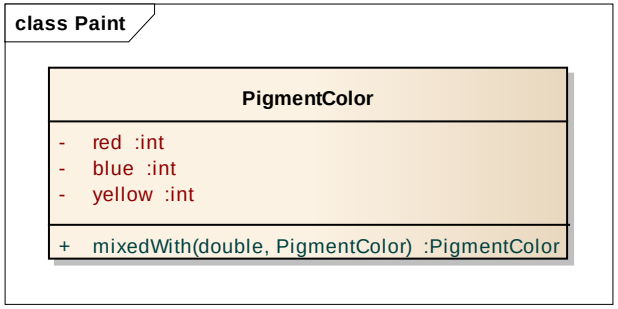
Lösungsansatz

- Definieren Sie Operationen mit dem gleichen Ergebnistyp wie der Parametertyp
- Hat die Implementierende Klasse Zustandswissen wird diese als Argument verwendet und der Rückgabetyp entspricht der implementierenden Klasse

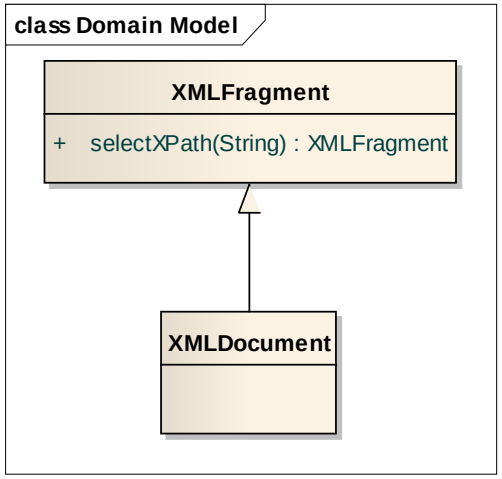
Vorteil

- Beliebige Verkettung/Kombination von Operationen
- Interpretation wird vereinfacht

Closure of Operations




```
PigmentColor red   = new PigmentColor(255,   0,  0);
PigmentColor green = new PigmentColor(  0, 255,  0);
PigmentColor gray  = new PigmentColor( 50,  50, 50);
PigmentColor mixed = null;

Mixed = red.mixedWith( green ).mixedWith( gray )
```




```
XMLDocument     document = XMLHelper.loadXML(…);


Document.selectXPath("./child/age").selectXPath( sum(
"./@value" )
```

Standalone Classes

Motivation

- Je höher die Abhängigkeit sind die von Operationen und Assoziationen impliziert werden desto schwieriger ist dass Modell zu verstehen.
- Abhängigkeiten entstehen durch Assoziationen und Argumente/Rückgabetypen von Operationen

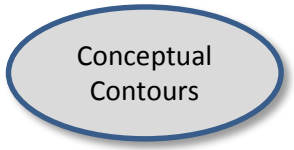
Lösungsidee

- Reduzieren Sie die Anzahl der Abhängigkeiten auf das notwendige (essentielle) Minimum
- Dadurch wird die Komplexität des Domänenmodells verringert
- Hat eine Klasse keine Abhängigkeiten muss nur diese Klasse selbst verstanden werden.

Conceptual contours

```
interface ManagerService {

   public void sendMail( ... );

   public void createCustomer( ... );

   public void deleteCustomer( ... );

   public void sendBill( ... );

   public void checkPaiment( ... );

   public HTLM createHTMLViewForKonto( ...);

   public HTLM createPDFViewForKonto( ...);
}
```

Conceptual contours



Motivation

- Werden Elemente eines Modells/design in einer monolithischen Struktur zusammengefasst besteht die Gefahr dass Funktionalität verdoppelt wird
- Werden Klassen/Methoden zu feingranular aufgeteilt erhöht dies die Komplexität da der Client verstehen muss wie die kleinen Einheiten zusammenspielen

Lösungsidee

- Anpassen des Designs an die Fachlichkeit
- Zerlegen Sie Designelemente (Operationen, Interfaces, Klassen und Aggregate) in zusammengehörige (cohesive) Einheiten unter Berücksichtigung der verschiedenen Sparten/Aspekte der Domäne
- Beobachten Sie die Aspekte der Veränderungen und suchen sie nach konzeptuellen Abgenzungen (Konturen)

Conceptual contours

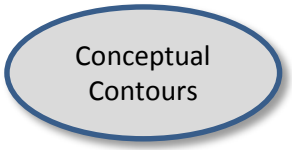
```
interface EMailService {

   public void sendMail( ... );
}
```

```
interface CustomerService {

   public void createCustomer( ... );

   public void deleteCustomer( ... );

}
```

```
interface InvoiceService {

   public void sendBill( ... );

   public void checkPaiment( ... );
}
```

```
interface ExportService {

   public HTLM createHTMLViewForKonto( ...);

   public PDF  createPDFViewForKonto( ...);
}
```

Conceptual contours



Motivation

- Werden Elemente eines Modells/design in einer monolithischen Struktur zusammengefasst besteht die Gefahr dass Funktionalität verdoppelt wird
- Werden Klassen/Methoden zu feingranular aufgeteilt erhöht diese die Komplexität da der Client verstehen muss wie die kleinen Einheiten zusammenspielen

Begründung

- Durch Refactorings wird das Modell an die stabilen Aspekte der Fachlichkeit angepasst
- Es steigt die Wahrscheinlichkeit das zukünftige Änderungen mit dem bestehenden Design leichter umgesetzt werden können