

## Quelle

Goldberg/Findler/Flatt: "Super and Inner - Together at Last",  
OOPSLA 2004

Implementiert in PLT Scheme

Beispiele in (Pseudo)-Java-Code

## Methoden Überschreiben

Seit Smalltalk können Subklassen Methoden überschreiben  
Überschreiben *ersetzt* aber Funktionalität

⇒ Nicht mehr klar, dass Subklasse anstelle ihrer Superklasse  
verwendet werden kann

## Methoden Überschreiben und super

**super** bietet Zugriff auf überschriebene Methode

Wenn die Subklassenmethode **super** aufruft, erweitert sie i.d.R. nur  
die überschriebene Methode

Aber: Die Subklasse entscheidet, ob sie **super** aufruft oder nicht

Superklasse kann dies nicht erzwingen

## Anreichern statt Überschreiben

Programmiersprache Beta(1987) bietet kein Überschreiben aber  
"Anreichern"

Subklasse kann neue Implementierung für Methoden angeben

- Aufgerufen wird stets Implementierung in Superklasse
- Superklasse kann mit **inner** die Implementierung aus der  
Subklasse aufrufen

⇒ Dynamic-Dispatch beginnt in der Wurzel der Klassenhierarchie,  
Suche nach unten bis zum dynamischen Typ

## Diskussion Anreichern

Vorteil:

- Superklassenmethode wird immer aufgerufen
- Keine willkürliche Neudefinition möglich
- Fördert (aber garantiert nicht) Subklassen, die das Verhalten ihrer Superklassen nur erweitern

Nachteil:

- Subklasse hat keinerlei Kontrolle über Code der Superklasse

## Überschreiben/Anreichern und Zuverlässigkeit

Überschreiben

- Wenn für korrekte Funktionalität die Methode der Superklasse aufgerufen werden muss, ist Zuverlässigkeit gefährdet
- Methode der Superklasse hat keine Möglichkeit sicherzustellen, dass sie aufgerufen wird

Anreichern

- Zuverlässigkeit wird erhöht, weil Methoden der Superklasse in jedem Fall aufgerufen werden

⇒ Anreichern erstrebenswert für Zuverlässigkeit

## Beispiele aus Java-Swing

- `JComponent.paint`: This method actually delegates the work of painting to three protected methods: `paintComponent`, `paintBorder`, and `paintChildren`.. A subclass that just wants to specialize the UI delegate's `paint` method should just override `paintComponent`.
- `JComponent.paintComponent`: Further, if you do not invoke super's implementation you must honor the `opaque` property, that is if this component is opaque, you must completely fill in the background in a non-opaque color. If you do not honor the `opaque` property you will likely see visual artifacts.

## Überschreiben/Anreichern und Re-Use

Überschreiben:

- Re-Use wird gefördert, da Anpassung der Subklasse einfach
- Subklasse kann Methoden komplett ersetzen, aber restlichen Code wiederverwenden

Anreichern:

- Weniger Re-Use, da Implementierung von Subklassen eingeschränkt ist

⇒ Überschreiben erstrebenswert für Re-Use

## Plan

Jetzt: Beispielanwendung, die sowohl Überschreiben als auch Anreichern benötigt

Erst in Java, dann in Beta (mit Java-Syntax)

Danach: Beta-ähnliche Methoden zu Java hinzufügen

## Das Beispiel

Klassenhierarchie für GUI-Toolkit:

- Window: `paint()` malt Hintergrund weiß
  - BorderLayout: `paint` malt Rahmen
  - Button: fügt `onClick` hinzu, `paint` malt Label
    - ImageButton: `paint` malt Bild in den Button
    - HighlightButton: `paint` malt blauen Hintergrund in den Button

## Realisierung in Java: Window und BorderLayout

`paint` in `BorderWindow` überschreibt und ruft `super` auf

```
class Window { ...
    void paint () {
        ... ; // Hintergrund weiss
    }
}

class BorderLayout extends Window { ...
    void paint () {
        super.paint(); //malt Hintergrund
        ... ; // malt Rahmen
    }
}
```

## Button in Java

Button überschreibt `paint`, *muss* `super.paint` aufrufen

```
class Button extends BorderLayout {
    void paint () {
        super.paint(); //malt Rahmen
        ... ; // malt Label
    }
}
```

`super.paint()` zu vergessen wäre ein Bug!

## ImageButton in Java

Problem:

- `super.paint()` würde Label malen. Bild müßte Label übermalen  
`paint` aus Button muss aber aufgerufen werden

Java-Lösung: Superklasse teilt Funktionalität auf in zwei Methoden:

- In Button wird `paint final`, ruft `super.paint` und `paintInside()` auf
- `paintInside` malt innerhalb des Rahmens, kann überschrieben werden
- Abgeleitete Klassen überschreiben oder erweitern `paintInside`

Entspricht dem Design-Pattern "Template-Method"

## Button mit Template-Method

Aufspalten von `paint`, `paint final` machen

```
class Button extends BorderLayout { ...
    final void paint () {
        super.paint();
        paintInside();
    }
    void paintInside () {
        ... // malt Label
    }
}
```

## ImageButton und HighlightButton

`paintInside` überschreiben, `super.paintInside` nicht aufrufen

```
class ImageButton extends Button { ...
    void paintInside () {
    }
}
```

`paintInside` überschreiben, `super.paintInside` aufrufen

```
class HighlightButton extends Button { ...
    void paintInside () {
        ... // malt dunklen Hintergrund
        super.paintInside(); // malt Label
    }
}
```

## Diskussion Template-Method

- Drückt Intention nur indirekt aus
- Bei nachträglicher Einführung umfangreiche Änderungen in den Subklassen nötig
- Erschwert das Arbeiten mit Mixins, weil sich der Methodenname ändert

## Beispiel mit Anreichern

Jetzt: Implementierung mit Anreichern wie in Beta

Syntax: Wie Java, aber kein `super`, dafür `inner`

Ergänzung

- `inner.f` *else stmt*
- Ruft *f* aus Subklasse auf, wenn diese *f* definiert, ansonsten werte *stmt* aus

## Beispiel mit inner

Window ruft `inner` auf, um Subklassen malen zu lassen

```
class Window {
    void paint() {
        ... ; // malt Hintergrund weiss
        inner.paint(); // lass Subklassen malen
    }
}

class BorderWindow extends Window { ...
    void paint () {
        ...; // malt Rahmen
        inner.paint(); // lass Subklassen malen
    }
}
```

Problem gelöst: Rahmen wird sicher gemalt

## Neues Problem

Unerwünschte Änderung: Hintergrund wird immer gemalt

Abhilfe (analog zu Template-Method):

- Window spaltet `paint` auf in `paint` und `paintBackground`
- `paint` ruft `paintBackground` auf, wenn Subklasse `paint` nicht anreichert
- Subklasse ruft `paintBackground` explizit auf

## Neues Window/BorderWindow

```
class Window { ...
    void paintBackground () {
        ... // malt Hintergrund
    }
    void paint () {
        inner.paint() else paintBackground();
    }
}

class BorderWindow extends Window { ...
    void paint () {
        paintBackground();
        ... // malt Rahmen
        inner.paint();
    }
}
```

## Button

Erneut Problem: Malen des Labels soll von der Subklasse implementiert werden können

Analoge Lösung: `paintLabel` einführen und nur aufrufen, wenn `paint` nicht angereichert:

```
class Button extends BorderWindow { ...
    void paintLabel () {
        ... ; // malt Label
    }
    void paint() {
        inner.paint() else paintLabel();
    }
}
```

## Restliche Klassen

```
class ImageButton extends Button { ...
    void paint() {
        ... ; // malt Bild, ruft paintLabel nicht auf
    }
}

class HighlightButton extends Button { ...
    void paint () {
        ...; //malt Hintergrund
        paintLabel(); // malt Label (ererbte)
    }
}
```

## Kombination super und inner

Sowohl Überschreiben als auch Anreichern allein sind unbefriedigend

⇒ Brauchen beide

Lösung: Erweiterung von Java um Beta-Methoden und `inner`

- Schlüsselwort `beta` kennzeichnet Methoden, die nur angereichert werden können
- Schlüsselwort `java` kennzeichnet Methoden, die überschrieben werden dürfen

## Methodenaufruf: Die beiden Extentfälle

Keine `beta`-Methoden vorhanden:

- Wie bisher
- Von Klasse des Empfängerobjekts nach oben
- Bei `super` Suche von unten nach oben

Keine `java`-Methoden vorhanden:

- Wie in Beta
- Suche von Wurzel der Klassenhierarchie nach unten
- Bei `inner` Suche weiter nach unten

## Methodenaufruf: Kombination

Sowohl `java` als auch `beta` vorhanden:

- Suche von oben nach unten nach erster `beta`-Methode

`beta`-Methoden haben also höhere Präzedenz

- Sinnvoll, da sie nicht übergangen werden sollen

## inner

- In `beta`-Methoden steht `inner` zur Verfügung
- `inner` ruft die oberste `beta`-Methode unterhalb der aufrufenden Methode auf
- `inner` in dieser Methode ruft nächste `beta`-Methode auf u.s.w.
- Gibt es keine `beta`-Methode, so ruft `inner` die unterste `java`-Methode auf

## super

Verhalten von `super` nicht ganz wie in Java:

- Superklasse wird weiterhin statisch bestimmt
- Aber Superklassenmethode kann nur aufgerufen werden, wenn sie keine `beta`-Methode ist
  - Intuition: `beta`-Methode wurde auf dem Weg nach unten bereits ausgeführt
  - Andernfalls besteht die Gefahr von Zyklen
  - Außerdem: Keine Anwendungen für anderes Verhalten

## Beispiel mit super/inner

`paint` soll komplett austauschbar sein  $\Rightarrow$  `java`

```
class Window { ...
    java void paint(){
        ... ; // malt Hintergrund
    }
}
```

Rahmen soll auf jeden Fall gemalt werden  $\Rightarrow$  `beta`

```
class BorderWindow extends Window { ...
    beta void paint() {
        super.paint(); // malt Hintergrund
        ...; // male Rahmen
        inner.paint(); // lasse Subklassen malen
    }
}
```

## Button

Malen des Labels ist optional ⇒ java

```
class Button extends BorderWindow { ...
    java void paint(){
        ...; // malt Label
    }
}
```

## ImageButton und HighlightButton

Bild soll auf jeden Fall gemalt werden ⇒ beta

```
class ImageButton extends Button { ...
    beta void paint(){
        ... ; // male Bild (rufe super nicht auf)
        inner.paint();
    }
}
```

Malen des Hintergrunds optional ⇒ java

```
class HighlightButton extends Button { ...
    java void paint(){
        ... ; // male dunklen Hintergrund
        super.paint(); // male Label
    }
}
```

## Die CLOS-Sicht

- **:before** und **:after** sind auch reine Anreicherungen
- **:around** kann Anreichen oder völlig ändern

Klassen die als Mixin gedacht sind, verwenden oft Methodenkombination anstelle von Überschreiben

Aber:

- Superklasse hat keine Möglichkeit, Überschreiben zu verhindern
- **call-next-method** sucht nächst allgemeinere Methode, d.h. kein **inner**

## Zusammenfassung

Seit Smalltalk sind Überschreiben und **super** populär

Vorteil: Re-Use leicht möglich

Problem: Zuverlässigkeit leidet

Beta bietet mit reiner Anreicherung und **inner** mehr Zuverlässigkeit aber weniger Möglichkeiten zum Re-Use

⇒ Kombination von Überschreiben und Anreicherung ist sinnvoll und möglich