

Mixins

Ein Mixin fügt Klassen Funktionalität hinzu

Implementiert einen bestimmten Aspekt, orthogonal zu anderen Klassen

Erweitert häufig bestehende Methoden um neue Funktionalität

Konzeptuell ist ein Mixin eine Funktion Klasse -> Klasse

Entstanden als Pattern in CLOS, mittlerweile eigenständiges Sprachkonzept

MixedJava

Mixins hier nach Flatt et. al.

Implementierung existiert für PLT Scheme

Wird in PLT Scheme sehr viel eingesetzt

Hier kein Scheme sondern Sprachentwurf MixedJava = Java + Mixins

Syntax für Mixins in MixedJava

Definition von Mixin: Wie Klassendefinition aber Schlüsselwort **mixin** statt **class**

Unterschied: **extends** nimmt als Argument ein Interface (keine Klasse)

Dieses Interface heißt *Vererbungs-Interface*

Vererbungs-Interface beschreibt die Klassen, auf die das Mixin angewendet werden kann

Bedeutung von Mixins in MixedJava

Klasse um Mixin erweitern: **class c2 = m(c1)**

Mixin **m** erweitert Klasse **c1** um Felder und Methoden

- **c1** muss das Vererbungs-Interface von **m** implementieren
- **m** überschreibt nur die Methoden, die in seinem Vererbungs-Interface angegeben sind
- Alle anderen Methoden aus **c1** sind unverändert in **c2**
- **c2** implementiert außerdem
 - Das Vererbungs-Interface von **m**
 - Alle Interfaces aus der **implements**-Klausel von **m**

Beispiel: Türen im Adventure

```
interface IDoor {
    boolean canOpen(Person p);
    boolean canPass(Person p);
}

class Door extends Object implements IDoor{
    boolean canOpen(Person p){
        return true;
    }
    boolean canPass(Person p){
        return true;
    }
}
```

Weitere Türen als Subklassen

```
class LockedDoor extends Door {
    boolean canOpen(Person p) {
        if (!p.hasItem(theKey))
            return false;
        else return super.canOpen(p);
    }
}

class ShortDoor extends Door {
    boolean canPass(Person p) {
        if (p.height() > 140) {
            return false; // too tall
        }
        else return super.canPass(p);
    }
}
```

Problem

Klasse für "Kleine Türen mit Schloss" nur möglich, indem Code aus LockedDoor kopiert wird:

```
class LockedShortDoor extends ShortDoor {
    boolean canOpen(Person p) {
        if (!p.hasItem(theKey))
            return false;
        else return super.canOpen(p);
    }
}
```

⇒ Sehr unschön, weil kein Code-Reuse

Lösung mit Mixins

Mixins für Erweiterung von Türen

```
mixin Locked extends IDoor {
    boolean canOpen(Person p) {
        if (!p.hasItem(theKey))
            return false;
        else return super.canOpen(p);
    }
}

mixin Short extends IDoor {
    boolean canPass(Person p) {
        if (p.height() > 140)
            return false; // too tall
        else return super.canPass(p);
    }
}
```

Kombination von Mixins

Jetzt können alle gewünschten Klassen definiert werden:

```
class LockedDoor = Locked(Door);
class ShortDoor = Short(Door);
class LockedShortDoor = Locked(Short(Door));
```

Weitere Variation

Angenommen, eine weitere Tür `MagicDoor` soll nur Personen durchlassen, die ein Zauberbuch dabei haben

Könnten Mixin `Magic` implementieren

Beob: Gemeinsamkeit mit `LockedMixin`: Person muss spezielles Item dabei haben

⇒ Abstraktion im Mixin `Secure`

Dann Kombination von Mixins

Secure-Mixin

```
interface ISecure extends IDoor {
    Object neededItem();
}

mixin Secure extends IDoor implements ISecure {
    Object neededItem () { return null;}
    boolean canOpen (Person p) {
        Object item = neededItem ();
        if (!p.hasItem(item))
            return false;
        else return super.canOpen(p);
    }
}
```

Spezialisierung

Jetzt Implementierung der Spezialfälle für Schlüssel und Zauberbuch:

```
mixin NeedsKey extends ISecure {
    Object neededItem() {
        return theKey;
    }
}

mixin NeedsSpell extends ISecure {
    Object neededItem () {
        return theSpellBook;
    }
}
```

Mixins mittels Mehrfachvererbung

Mehrfachvererbung kann Mixins simulieren:

- Definition des Mixins ist (abstrakte) Klasse die Aspekt des Mixins implementiert
- Klassen, denen Mixin hinzugefügt werden soll, erben von Mixin-Klasse

Betrachten jetzt Realisierung in C++ und CLOS

Mixins in C++ (nach Stroustrup)

Interface als pure virtual base class:

```
struct IDoor {  
    virtual bool canOpen(const Person_c &p) = 0;  
    virtual bool canPass(const Person_c &p) = 0;  
};
```

Klasse Door implementiert dieses Interface:

```
struct Door : virtual IDoor {  
    bool canOpen(const Person_c &p) {  
        return true;  
    }  
    bool canPass(const Person_c &p) {  
        return true;  
    }  
};
```

Mixins in C++

Mixin definieren: `Locked` erbt von "Interface"

```
struct Locked : IDoor {  
    bool canOpen(const Person_c &p) {  
        if (!p.hasItem(the key)) return false;  
        return true; // kein super.canOpen()  
    }  
};
```

Aufruf von `super.canOpen` wurde weggelassen: siehe später

Mixin anwenden

Anwendung durch Mehrfachvererbung

Doppeldeutigkeiten müssen von Hand aufgelöst werden:

```
struct LockedDoor : Door, Locked {  
    bool canOpen(const Person_c &p){  
        return Locked::canOpen(p);  
    }  
    bool canPass(const Person_c &p){  
        return Door::canPass(p);  
    }  
};
```

Beob: Auch `canPass` muss aufgelöst werden

Mixins in C++: super

Fehlt noch: Mixin sollte überschriebene Methode aufrufen können

- Kein `super` in C++,
- `::` funktioniert auch nicht, weil Superklasse dem Mixin nicht bekannt

Einziger Ausweg: Mixin mittels Templates über Superklasse parametrisieren

Mixins mit Templates

Idee dazu: Mixin erbt von Templateparameter

```
template<typename super>
struct Locked : super {
    bool canOpen(const Person_c &p) {
        if (!p.hasItem(theKey)) return false;
        return super::canOpen(p);
    }
};
```

```
struct LockedDoor : Locked<Door> { };
```

Beobachtungen:

- Keine Mehrfachvererbung mehr
- Mixin ist wieder Funktion von Klasse nach Klasse

Mixins in CLOS

Erinnerung: In CLOS löst Linearisierung Namenskonflikte bei MFV auf

Linearisierung erlaubt es, Mehrfachvererbung zu verwenden, um Mixins zu realisieren:

- Definiere Mixin als Klasse, die Aspekt implementiert
- Leite neue Klasse von Mixin und Basisklasse ab
- Mixin steht vor (also links) der Basisklasse
- Methode des Mixins wird somit zuerst gefunden
- Mixin-Methoden rufen `call-next-method` auf, um ursprüngliche Funktionalität zu erhalten

Door-Beispiel in CLOS

```
(defclass door-class () ())
(defmethod canOpen ((d door) (p person)) t)

(defclass door-mixin () ())
(defmethod canOpen ((d door-mixin) (p person))
  (if (not (hasItem p the-key))
      nil
      (call-next-method)))

(defclass locked-door (door-mixin door-class) ())
```

Linearisierung für `locked-door`:

```
locked-door, door-mixin, door-class
```

Probleme mit Linearisierung

Wünschenswerte Eigenschaft für Vererbung:

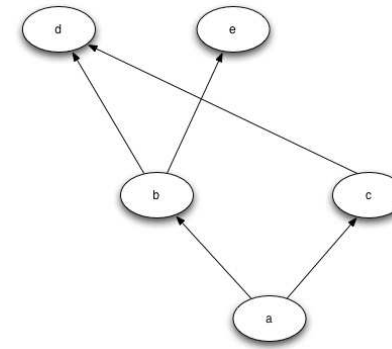
Kapselung von Klassendefinitionen: Art der Definition einer Superklasse beeinflusst Subklasse nicht

⇒ Anforderung an Linearisierung:

- **Monotonie**: Für jede Eigenschaft, die die Linearisierung einer Klasse zuweist, gibt es eine direkte Superklasse, die ebenfalls diese Eigenschaft hat
- **Stabilität**: Faktorisiert man einen Teil einer Klasse in eine neue Superklasse auf, so ändern sich die Eigenschaften der Subklassen nicht

Zeigen nun: CLOS-Linearisierung ist nicht in allen Fällen monoton und stabil

CLOS ist nicht stabil



Beob: $d < e$

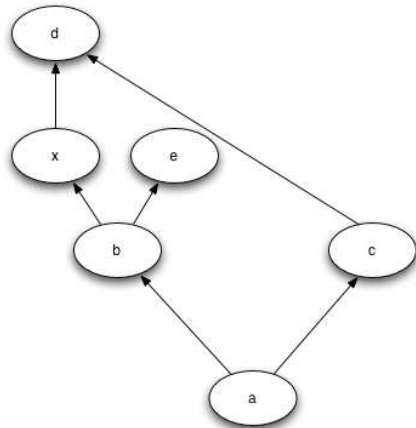
Linearisierung für **a**: a, b, c, d, e

Beispiel:

- **d** definiert Slot **x** mit Slotoption **:type = int**
- **e** definiert Slot **x** mit Slotoption **:type = string**

⇒ **a** besitzt Slot **x** mit **:type = int**

Aufspalten von b in b und x



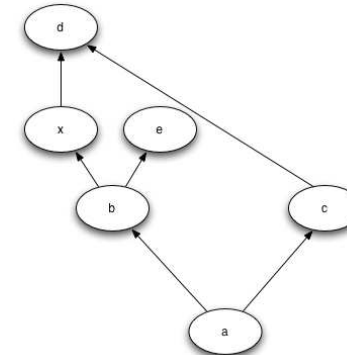
Beob: $d < e$ verschwunden!

Linearisierung für **a**: a, b, x, e, c, d

Jetzt hat **a** Slot **x** mit **:type = string**

Also Änderung in **a** wegen Aufspaltung in Superklasse **b**!

CLOS ist nicht monoton



Linearisierung für **a**: a, b, x, e, c, d

Linearisierung für **b**: b, x, d, e

Linearisierung für **c**: c, d

:type = string für **x** in **a**, aber in Superklassen **b** und **c** ist **:type = int**

Also in **a** anders als in allen Superklassen!

Linearisierung und Kapselung

Konsequenz aus Linearisierungsalgorithmus ohne Stabilität und Monotonie: Kapselung von Klassendefinitionen wird gebrochen:

- Superklassendeklaration einer Klasse beeinflusst Konfliktauflösung ihrer Subklassen auf nicht-triviale Weise
- Programmierer muss gesamte Vererbungshierarchie kennen

Zusammenfassung zu MFV und Mixins

In C++ kann MFV Mixins nicht wirklich realisieren

In CLOS ermöglicht Linearisierung MFV zu verwenden

- Aber Linearisierung wird schnell unübersichtlich
- Außerdem: CLOS bietet keine Möglichkeit, Klassen auf die ein Mixin angewendet wird, einzuschränken

⇒ Mehrfachvererbung ist nicht das geeignete Mittel, um Mixins zu spezifizieren

Mixins anstatt Mehrfachvererbung

Mixins wird nachgesagt, ausreichend mächtig zu sein, um Mehrfachvererbung zu ersetzen

- Linearisierung entfällt, weil Mixinkomposition Reihenfolge festlegt
- Es gibt wenige Anwendungsfälle für "echte" Mehrfachvererbung
 - Stroustrup: "Mehrfachvererbung nur mit pure virtual base classes"

Mixins und Klassen als Werte

Frage: Kann man Mixins in Sprachen implementieren, in denen Klassen Werte sind?

Antwort: Nur wenn die Klassendefinitionen Ausdrücke sind, ist es möglich, Mixins als Funktionen zu implementieren

Problem: Bindung von `self` und `super` sind subtiler bei Mixins

PLT Scheme verwendet diese Technik:

```
(define locked-mixin
  (lambda (IDoor)
    (class IDoor
      (public
        (canOpen (lambda ()
                  ...))))))
(define locked-door (locked-mixin door))
```

Beispiele für Mixins in PLT Scheme

- **text:searching-mixin:** Erweitert Texteditoren um Suchfunktionalität
- **canvas:color-mixin:** Fügt einem Canvas Hintergrundfarbe hinzu
- **frame:status-line-mixin:** Fügt einem Frame eine Statuszeile hinzu

Zusammenfassung Mixins

Mixins fügen Klassen neue Funktionalität hinzu

- Mixins sind konzeptuell Operationen auf Klassen
- Fördern die Wiederverwendbarkeit
- Vermeidet die Probleme der Mehrfachvererbung

Mehrfachvererbung kann Mixins nicht zufriedenstellend simulieren

Mixins fördern das Programmieren gegen Schnittstellen anstatt Implementierungen