Modellierung und Programmierung 2

Aspektorientierte Programmierung

Überblick

- **Literatur:** G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin: **Aspect-Oriented Programming**.
  In: Proceedings of the European Conference on Object-oriented Programming, Springer, 2007

- Generalized Procedures
- AOP
  - Definition
  - Beispiele
  - Umsetzung
- AOP mit Python

Generalized Procedures
Rückblick Modellierung und Programmierung

- **Modellierung**
  - Ziel: **Entwurf** eines Programmes zur Lösung einer Aufgabe
  - **concern**: Anliegen bzw. Einheit des Programms (units)
  - → Zerlegung der Problems in „concerns“
  - Programm entsteht durch Zusammensetzen der Units (Komposition)
- **Programmierung**
  - Ziel: **Umsetzung** und **Komposition** der Units

Generalized Procedures
Rückblick Modellierung und Programmierung

- **Modellierung**
  - ...
- **Programmierung**
  - Ziel: **Umsetzung** und **Komposition** der Units
  - Paradigmen:
    - imperativ
    - prozedural / modular
    - objektorientiert
    - funktional

Generalized Procedures
Rückblick Modellierung und Programmierung

- **Modellierung**
  - ...
- **Programmierung**
  - Ziel: **Umsetzung** und **Komposition** der Units
  - Paradigmen:
    - imperativ
    - prozedural / modular
    - objektorientiert
    - funktional
  - Gemeinsamkeit:
    - Systemeinheiten sind **Prozeduren**
    - Komposition erfolgt durch **Prozeduraufruf**
  - → Implementierung als **Generalized Procedures (GP)**

Generalized Procedures
Rückblick Modellierung und Programmierung

- **Modellierung**
  - Ziel: **Entwurf** eines Programmes zur Lösung einer Aufgabe
  - **concern**: Anliegen bzw. Einheit des Programms (units)
  - → Zerlegung der Problems in „concerns“
  - Programm entsteht durch Zusammensetzen der Units (Komposition)
- **Programmierung**
  - Ziel: **Umsetzung** und **Komposition** der Units
  - → Implementierung als **Generalized Procedures (GP)**

Generalized Procedures
Rückblick Modellierung und Programmierung

- **Modellierung**
  - Ziel: **Entwurf** eines Programmes zur Lösung einer Aufgabe
  - **concern**: Anliegen bzw. Einheit des Programms
  - → Zerlegung der Problems in **funktionale Einheiten** = **„functional decomposition“**
  - Programm entsteht durch Komposition der Units
- **Programmierung**
  - Ziel: **Umsetzung** und **Komposition** der Units
  - → Implementierung der **funktionalen Einheiten** als **Generalized Procedures (GP)**

Generalized Procedures
„cross-cutting concerns“ und „entanglement“

- **Modellierung**
  - **concern**: Anliegen bzw. Einheit des Programms
  - → Zerlegung der Problems in **funktionale Einheiten**
- Manche Anliegen betreffen **alle** funktionalen Einheiten
- Bsp.: **Logging**
  - Bei Funktionseintritt soll Funktionsname und Parameterliste in eine Datei geschrieben werden
  - Bei Funktionsaustritt soll Rückgabewert in eine Datei geschrieben werden
  - → Logging-Bausteine tauchen überall auf:
    ```
    printf( „Start of function %s, with params %s“, name, params );
    ```
- → „cross-cutting concerns“

Generalized Procedures
„cross-cutting concerns“ und „entanglement“

- „cross-cutting concern“: Eigenschaft, die mehrere funktionale Einheiten betrifft
- Probleme:
  - **Scattering**: Implementierung über das ganze System verteilt
  - **Entanglement**: Implementierung ist mit anderen Einheiten des Systems verwoben
- Ursache:
  - mehrere **Kompositionsebenen**
  - Verschiedene concerns werden auf verschiedene Weise zusammengesetzt, müssen aber koordiniert werden

Generalized Procedures
„cross-cutting concerns“ und „entanglement“

- Ursache:
  - mehrere **Kompositionsebenen**
  - Verschiedene concerns werden auf verschiedene Weise zusammengesetzt, müssen aber koordiniert werden
- Beispiel **Logging**:
  - Ebene 1: Funktionen des Programms

```
(defun getBestFive (input)
  (returnFirstFive (sort input)))
```

  - Ebene 2: Ablauf eines Funktionsaufrufs

```
(progn (log 'start name params)
       (call name params)
       (log 'end name))
```

AOP – Definition

- Eine Eigenschaft des Systems ist
  - eine **Komponente**, wenn sie sauber in eine **GP** gekapselt werden kann
    → *sauber = an **einer** Stelle implementiert, leicht verfügbar und kombinierbar*
    → *functional units*
  - ein **Aspekt**, wenn sie nicht sauber gekapselt werden kann
    → *cross-cutting concerns*
    → *betrifft oft Eigenschaften, die sich auf Effizienz oder Semantik des ganzen Programms auswirken*
- **GP** erlaubt Abgrenzung der Komponenten voneinander
- **Ziel AOP**: Mechanismus zum Abgrenzung von Komponenten und Aspekten untereinander

AOP – Beispiele

- Standard-Beispiele: Logging, Caching, Zugriffsrechte, Authentifizierung

Anwendung	Sprache	Komponenten	Aspekte
Digitale Bibliothek	Objektorientiert	Bücher, Sammlungen, Drucker	Netzwerktraffic minimieren Synchronisation über mehrere Rechner Fehlerbehandlung
Matrix-Rechnung	Prozedural	Lineare Algebra-Operationen	Matrix-Repräsentation Numerische Fehler
Bildverarbeitung	funktional	Bildfilter	Statische Speicherverwaltung Laufzeiteffizienz Zwischenergebnisse wiederverwenden

AOP – Beispiel Texterkennung

- Basisfilter:
  - Invertierung

```
(defun not (a) (let ((result (new-image)))
  (loop for i from 1 to width do
    (loop for j from 1 to height do
      (set-pixel result i j (not (get-pixel a i j)))))))
```

  - and, or, down, up
- Zusammengesetzte Filter:
  - `(defun remove (a b) (and a (not b)))`
  - `(defun top-edge (a) (remove a (down a)))`
  - `(defun bottom-edge (a) (remove a (up a)))`
  -

```
(defun horizontal-edge (a)
  (or (top-edge a)
      (bottom-edge a)))
```

- Problem: elegant, aber ineffizient (Laufzeit und Speicher)

AOP – Beispiel Texterkennung

- Filter (Komponenten) als Makro und High-Level formuliert:

```
(define-filter 'or (a b)
  (pixelwise (a b) (aa bb) (or aa bb)))
```

- Laufzeiteffizienz (Aspekt) durch Filtervereinigungsregeln umgesetzt:

```
(cond ((and (eq (loop-shape node) 'pointwise)
            (eq (loop-shape input) 'pointwise))
       (fuse loop input 'pointwise
         :inputs (splice …)
         :loop-vars (splice …)
         :body (subst …))))
```

  Legt fest, wann zwei Loops vereinigt werden und wie das geschieht.

- Zusammensetzung durch „Aspekt Weaver“
  - Erstellt Datenflussgraph aus Filterdefinitionen
  - Formt Graph mit Regeln des Aspektprogramms um

Umsetzung AOP

- Zwei Sprachen:
  - **„join points“**: Schnittmenge zwischen Komponenten und Aspekten
  - **Komponentensprache**: Definition und Komposition von Komponenten
  - **Aspektsprache**: Definition und Komposition der Aspekte
- **Aspect-Weaver:** führt „Ko-Komposition“ von Aspekten und Komponenten durch
  - Join-point-Repräsentation aus dem Komponentenprogramm
  - Ausführung des Aspektprogramms
  - Code-Generation

AOP mit Python

- Python ist funktionale Sprache
  - → Python behandelt Funktionen wie Werte
- Python ist dynamische Sprache
  - → jedes Objekt kann zur Laufzeit durch etwas anders ersetzt werden
- Python erlaubt reflection
  - Jeder Scope ist in einer Map gespeichert und kann durchsucht werden

AOP mit Python

- → wir bauen higher-order functions
- Beispiel Logging:

```
def Logging(f):
    def loggedFunc(*args, **kwargs):
        # log start of function
        result = f(*args, **kwargs)
        # log result
        return result
    return loggedFunc
```

- Anwendung:

```
def MyFunc(): …
MyFunc = Logging(MyFunc)
```

- Shorter: Use decorators

```
@Logging
def MyFunc(): …
```

- Löst entanglement

AOP mit Python

- Lösung von Scattering:
  - Funktion, die Namespaces durchsucht und Decorator auf jede passende Funktion anwendet
  - Problem: unlesbarer Code
- Further reading:
  - http://www.slideshare.net/meij200/aop-in-python-api-design-12937607
  - https://wiki.python.org/moin/PythonDecoratorLibrary#Memoize
  - http://dietbuddha.blogspot.de/2012/11/python-class-decorator-logging.html

AOP mit Python

- Anwendung:
```
def MyFunc(): …
MyFunc = Logging(MyFunc)
```
- Shorter: Use decorators
```
@Logging
def MyFunc(): …
```
- → löst entanglement