

# Betriebssystemtechnik

Operating System Engineering (OSE)

## “Policy-Based Design”<sup>[1]</sup>

(A. Alexandrescu)



1

### Motivation

- Problemfeld Entwurfsentscheidungen
  - „wie man es macht, macht man es falsch“
  - Lösungen können je nach Anwendungsfall mehr oder weniger (oder gar nicht) geeignet sein
  - Entscheidung sollte nicht beim Entwickler einer Klasse liegen, sondern beim Anwender
- Ziel: Klassen konfigurierbar gestalten
  - Schaffung einer Auswahlmöglichkeit für verschiedene Verhaltensweisen durch den Anwender

© 2006 Olaf Spinczyk

2

### Idee von Policy-based Design

- Klassen mit komplexem Verhalten werden in eine **Host-Klasse** und mehrere kleinere **Policy-Klassen** zerlegt
  - jede **Policy** kapselt einen Belang der **Host**-Klasse
  - Verbindung der komplexen Klasse mit den **Policy**-Klassen über C++ *Templates*
- **Policies** sind vom Benutzer der Klasse auswählbar
  - der Benutzer kann also das Verhalten der **Host**-Klasse durch geeignete Auswahl von **Policy**-Klassen anpassen



© 2006 Olaf Spinczyk

3

### Beispiel: NewCreator

- ... ist ein Klassen-*Template*, das neue Objekte erzeugt:

```
template <class T>
class NewCreator {
public:
    static T *create() {
        return new T;
    }
};

Widget *w = NewCreator<Widget>::create();
```

- **NewCreator** erzeugt alle Objekte mit **new**
  - in manchen Fällen könnte anderes Verhalten erwünscht sein

© 2006 Olaf Spinczyk

4

## Beispiel: MallocCreator

- Alternative zum NewCreator

- MallocCreator fordert Speicher mit `malloc()` an
- Objekt wird mit *Placement New* in den Speicher gelegt

```
template <class T>
class MallocCreator {
public:
    static T *create() {
        void *buf = malloc(sizeof(T));
        if (!buf) return 0; // oder Exception werfen
        return new(buf) T; // placement new
    }
    ...
}

Widget *w = MallocCreator<Widget>::create();
```



© 2006 Olaf Spinczyk

5

## Beobachtung

- MallocCreator und NewCreator besitzen dieselbe Schnittstelle

- beide verfügen über die Methode `create()`, die ein neues Objekt erstellen soll

- eine gemeinsame abstrakte Basisklasse, die die gemeinsame Schnittstelle manifestieren würde, gibt es aber nicht

- NewCreator lässt sich trotzdem im Quellcode durch MallocCreator ersetzen

- **statische Polymorphie**

© 2006 Olaf Spinczyk

6

## Der Begriff „Policy“

- Policies sind Schnittstellen für konfigurierbare Belange einer Klasse
- Policy-Klassen implementieren die von der Policy vorgegebene Schnittstelle
- Beispiel: *Creation Policy*
  - kapselt das Erstellen von neuen Objekten
  - Schnittstelle besteht aus der Methode `create()`, welche ein neues Objekt erzeugen soll
  - *Templates* NewCreator und MallocCreator sind mögliche Policy-Klassen; es sind aber noch weitere denkbar



© 2006 Olaf Spinczyk

7

## Verwenden der Policy

- Policy-Klassen werden bei der Übersetzung mittels *Template*-Parameter in die Host-Klasse eingebunden
- Beispiel WidgetManager:

```
template <class CreationPolicy>
class WidgetManager {
public:
    void newWidget() {
        Widget *w = CreationPolicy::create();
        ...
    }
    ...
};
```



© 2006 Olaf Spinczyk

8

## Auswahl der *Policy*-Klasse

- Anwender von WidgetManager können die Creation Policy nun auswählen:
  - Mit NewCreator:  
`typedef WidgetManager<NewCreator<Widget> > WM;`
  - oder mit dem MallocCreator:  
`typedef WidgetManager<MallocCreator<Widget> > WM;`
- Problem:
  - Man muss immer „Widget“ mit angeben, obwohl klar ist, dass der WidgetManager Widgets anlegen will



© 2006 Olaf Spinczyk

9

## Template Template Parameter

- Lösung: Host-Klasse bekommt die *Policy*-Klasse als *Template Template* Parameter übergeben:
  - einfacher zu lesen
  - weniger fehleranfällig: Benutzer gibt den Objekttyp nicht selbst an
  - flexibler: WidgetManager kann auch andere Objekte erzeugen

```
template <template <class> class CreationPolicy>
class WidgetManager {
public:
    void newWidget() {
        Widget *w = CreationPolicy<Widget>::create();
        ...
    }
    ...
};
```

// jetzt kompakter und einfacher:  
`typedef WidgetManager<NewCreator> WM;`

© 2006 Olaf Spinczyk

10

## Weitere Möglichkeiten

- die Struktur der Host-Klasse ist durch Erben von der *Policy*-Klasse erweiterbar
  - Host erbt Member-Variablen von den Policies
  - Policies können zusätzliche Methoden in die Host-Klasse einbringen („Enriched Policies“)
- Policies sind so nicht nur auf Verhalten beschränkt



© 2006 Olaf Spinczyk

11

## Entwurf einer Klasse mit Policies

- Auswahl der Policies
  - Entscheiden, welche Belange durch Policies realisiert werden
  - Schwierige Entwurfsentscheidungen in Policies auslagern
  - bei mehreren Policies: möglichst orthogonale Semantik definieren
- Konsequenzen
  - Policies müssen beim Entwurf der Host-Klasse festgelegt werden!
  - nachträgliche Einführung zusätzlicher Policies schwierig (im Gegensatz zum Erweitern durch Policy-Klassen)
  - Policy ist im Code der Host-Klasse verankert
    - „join points“ sind damit explizit vorgegeben
    - keine gute Möglichkeit zur Modularisierung quer schneidender Belange



© 2006 Olaf Spinczyk

12

## Zusammenfassung und Ausblick

- *Policy-based Design* ...
  - erlaubt die getrennte Realisierung konfigurierbarer Belange in Form von *Policies*
  - ermöglicht flexible Klassen, deren Verhalten der Anwender an seine Bedürfnisse anpassen kann
  - basiert auf Sprachelementen von C++
    - *Templates*
    - *Template Template Parameter*
- ein größeres Anwendungsbeispiel folgt im Abschnitt zu „*Synthesizing Objects*“



## Literatur

- [1] A. Alexandrescu, *Modern C++ Design - Generic Programming and Design Patterns Applied*. Addison-Wesley, 2002, ISBN 0-201-70431-5.

