Policy-based Design

Vortrag im Seminar Aspektorientierte Systemprogrammierung

Von Alexander Beisig

Übersicht

Motivation

Idee des Policy-based Design

Beispiel: Alternative Templates zum Erstellen neuer Objekte

Der Begriff „Policy“

Verwendung von Policies

Aufteilung von Klassen in Policies

Beispiel: Policy-based Smart Pointer

Zusammenfassung

Motivation

- Problemfeld Entwurfsentscheidungen
  - viele Entwurfsentscheidungen haben mehrere mögliche Lösungen
  - Lösungen können je nach Gegebenheiten mehr oder weniger (oder auch gar nicht) geeignet sein
  - Entscheidung sollte nicht beim Entwickler einer Klasse liegen, sondern beim Benutzer

⇒ Ziel: Schaffen einer Auswahlmöglichkeit für verschiedene Verhaltensweisen

Idee von Policy-based Design

- Klassen mit komplexem Verhalten werden zerlegt in kleinere Policy-Klassen
  - Jede Policy kapselt einen Belang der sog. *Host-Klasse*
  - Verbindung der komplexen Klasse mit den Policy-Klassen über C++-Templates
- Policies sind vom Benutzer der Klasse auswählbar
  - Benutzer kann Host-Klasse durch Auswahl geeigneter Policy-Klassen anpassen

NewCreator

- Beispiel eines Klassen-Template
  - NewCreator dient zum Erzeugen neuer Objekte

```
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

MallocCreator

- Alternative zum NewCreator
  - MallocCreator fordert Speicher an mit malloc
  - Objekt wird mit placement new in diesen 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
  }
};
```

Beobachtung

- MallocCreator und NewCreator besitzen dieselbe Schnittstelle
  - beide verfügen über die Methode Create, die ein neues Objekt erstellen soll
  - Schnittstelle ist aber im Code nicht explizit angegeben
- NewCreator lässt sich im Quellcode durch MallocCreator ersetzen
- statische Polymorphie

Der Begriff „Policy“

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

Verwenden der Policy

- Policy-Klassen werden bei der Übersetzung mittels Templates in die Host-Klasse eingebunden

- Beispiel WidgetManager:

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

Auswahl der Policy-Klasse

- Benutzer von WidgetManager können die Creation Policy auswählen:
  - Mit NewCreator:

```
typedef WidgetManager<NewCreator<Widget> > MyWidgetManager;
```

  - oder mit MallocCreator:

```
typedef WidgetManager<MallocCreator<Widget> > MyWidgetManager;
```

- Problem:
  - Man muss immer „Widget“ mit angeben, obwohl klar ist, dass der WidgetManager Widgets anlegen will

Template Template Parameter

- Lösung: Templates können als Parameter statt Typen auch Templates verwenden

- Beispiel mit Template Template Parameter (TTP):

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

Vereinfachung durch TTP

- Einsatz des WidgetManagers ohne TTP:

```
typedef WidgetManager<NewCreator<Widget> > MyWidgetManager;
```

- Vereinfachung durch TTP:

```
typedef WidgetManager<NewCreator> MyWidgetManager;
```

- Vorteile aus der Verwendung von TTP:
  - einfacher zu lesen
  - weniger fehleranfällig, da andere Angabe als Widget Fehler verursachen kann
  - flexibler: WidgetManager könnte Policy auch zum Erzeugen anderer Objekte verwenden

Weitere Möglichkeiten

- Struktur der Host-Klasse durch Beerbung von Policy-Klassen erweiterbar
  - Host-Klasse erbt Variablen von den Policies
  - Policies können zusätzliche Methoden in die Host-Klasse einbringen („Enriched Policies“)

$\Rightarrow$ Policies sind nicht nur auf Verhalten beschränkt

Entwurf einer Klasse mit Policies

- Auswahl der Policies
  - Entscheiden, welche Belange von Policies gehandhabt werden sollen
  - Abwägungen/Entwurfsentscheidungen Policies überlassen
  - bei mehreren Policies: Möglichst unabhängig voneinander
- Implementierung
  - Host-Klasse verwendet die Policies zur Bereitstellung seiner Funktionalität
- Policies müssen beim Entwurf ausgewählt werden!

Policies für SmartPtr

- SmartPtr ist eine generische Smart-Pointer-Klasse
  - Verwaltet einen einfachen Zeiger
  - Verhält sich (zumeist) wie der einfache Zeiger
  - Bietet aber mehr Funktionalität als der einfache Zeiger
- Zerlegung in Policies (Beispiel)
  - Ownership: verwaltet den Besitz des Zeigers
  - Checking: führt automatische Überprüfungen aus
  - Conversion: legt fest, ob der SmartPtr implizit in den Typ des verwalteten Zeigers konvertiert werden kann

Ownership Policy

- Zielsetzung
  - Verschiedene Strategien um sicher zu stellen, dass verwaltetes Objekt genau ein mal freigegeben wird
  - Schwierig beim Zuweisen und Kopieren von SmartPtr-Objekten
- mögliche Policy-Klassen (Auswahl)
  - DeepCopy: verwaltetes Objekt wird mit kopiert
  - RefCounted
  - DestructiveCopy: nur die Kopie besitzt das Objekt

Checking und Conversion Policies

- Checking Policy
  - SmartPtr kann beim Dereferenzieren den Wert des Zeigers überprüfen
  - Policy-Klassen: NoCheck, EnforceNotNull
- Conversion Policy
  - SmartPtr kann in Operatoren überladen, um implizit in den Typ des Zeigers umwandelbar zu sein
  - Policy-Klassen: AllowConversion, DisallowConversion

Das SmartPtr Klassen-Template

- Die fertige Deklaration des Klassen-Template

```
template <class T, // Der Typ des Zeigers
          template <class> class OwnershipPolicy,
          template <class> class CheckingPolicy,
          class ConversionPolicy>
class SmartPtr;
```

- Einsatz von SmartPtr

```
typedef SmartPtr<Widget, RefCounted, NoCheck, AllowConversion>
   WidgetPtr;

WidgetPtr w(new Widget); w->doSomething();
```

Umsetzung der Ownership Policy

- Schnittstelle der Policy:
  - SmartPtr kopiert verwalteten Zeiger mit Methode Clone
  - SmartPtr gibt mit Methode Release gibt Besitz am Objekt auf (Rückgabewert gibt an, ob Objekt freigegeben werden soll)
- Beispiel DeepCopy:

```
template <class T> struct DeepCopy {
  T* Clone(const T* ptr) { return new T(*ptr); }
  bool Release(const T* ptr) { return true; }  // immer freigeben
};
```

Reference Counting

- Beispiel RefCounting:

```
template <class T> struct RefCounting {
  RefCounting() : pcount(new int(1)) {}
  RefCounting(const RefCounting& other) : pcount(other.pcount) {}
  T* Clone(const T* ptr) {
    ++(*pcount); return ptr;
  }
  bool Release(const T* ptr) {
    if (0 == --(*pcount)) {
      delete pcount; return true; // Objekt wird freigegeben
    }
    return false; // Objekt wird nicht freigegeben
  }
};
```

Einbinden in die Host-Klasse

- die Host-Klasse verwendet die Policy-Schnittstelle

```
template <class T, ...>
class SmartPtr : private OwnershipPolicy<T>, ... {
  T* zeiger;
  ...
public:
  SmartPtr(SmartPtr& other) :
    OwnershipPolicy<T>(other), ...,
    zeiger(Clone(other.zeiger)) { }
  ~SmartPtr() {
    if (Release(zeiger)) delete zeiger;
  }
};
```

Beobachtungen

- Host-Klassen abhängig von Policy-Schnittstellen
  - Policy-Schnittstellen bestimmen Entwurf der Host-Klasse
  - nachträgliche Einführung von zusätzlichen Policies ist schwierig
- Verankerung der Policies im Code der Host-Klasse
  - „Join Points“ zwischen Host-Klasse und Policy-Klassen sind vorgegeben
  - keine gute Möglichkeit zur Modularisierung von querschneidenden Belangen

Zusammenfassung

- Policy-based Design...
  - erlaubt die Aufteilung einer Klasse in Policies, die einzelne Belange der Klasse kapseln
  - ermöglicht flexible Klassen, deren Verhalten an verschiedene Gegebenheiten angepasst werden können
  - basiert auf reinem C++ und benötigt keine Hilfsmittel
- Aber:
  - Modularisierung querschneidender Belange schwierig, da Policies in der Host-Klasse verankert