Vortrag im Seminar Aspektorientierte Systemprogrammierung

Policy-based Design

Referent: Alexander Beisig
Studiengang: Informatik
Datum: 10.5.2004

Übersicht

Motivation

Beim Entwurf von generischen Klassen müssen häufig Entscheidungen getroffen werden, für die es keine eindeutigen Lösungen gibt. Je nach Einsatzumfeld können die verschiedenen Lösungen mehr oder weniger geeignet erscheinen. Die Auswahl wird weiter dadurch erschwert, dass sich Anforderungen im Entwicklungsprozess verändern können. Ein Beispiel für eine solche Abwägung stellen automatische Überprüfungen einer Smart-Pointer-Klasse dar. Man könnte eine Smart-Pointer-Klasse so entwerfen, dass das Dereferenzieren eines Nullpointers durch automatische Tests ausgeschlossen wird. Hiermit ist aber auch ein gewisser Performanceverlust verbunden, so dass der Nutzen der automatischen Überprüfung mit den Nachteilen abgewogen werden muss.

Der Entwickler einer Klasse mit komplexem Verhalten wird häufig nicht in der Lage sein, die optimalen Entwurfsentscheidungen vorherzusehen. Zudem können verschiedene Einsatzfelder auch unterschiedliche Anforderungen haben und damit unterschiedliche Strategien erfordern. Daher ist es erstrebenswert, dass die Entscheidung über solche Abwägungen nicht beim Entwickler liegen sollte, sondern beim Benutzer der Klasse, der ein besseres Verständnis für das Einsatzumfeld hat als der Entwickler.

Entsprechend dem Prinzip der Trennung der Belange sollte man also bei der Entwicklung generischer Klassen das Ziel verfolgen, die mit Abwägungen verbunden Belange einer Klasse von dieser loszulösen. Somit wird es dem Benutzer bei solchen Belangen ermöglicht, das geeignete Verhalten für die Klasse auszuwählen. Policy-based Design ist ein Ansatz, mit Hilfe von C++-Templates dieses Ziel zu erreichen.

Idee von Policy-based Design

Die Idee von Policy-based Design besteht darin, eine Klasse mit komplexem Verhalten in kleinere sog. Policy-Klassen aufzuteilen. Jede der Policy-Klassen kapseln dabei einen Belang der Host-Klasse, mit dem eine Abwägung verbunden ist. Die Verbindung zwischen Host-Klasse und Policy-Klassen erfolgt dabei bei der Übersetzung mit Hilfe von C++-Templates.

Durch diese Aufteilung wird erreicht, dass der Benutzer durch die Auswahl geeigneter Policy-Klassen das Verhalten anpassen kann. Da Policy-Klassen normalerweise unabhängig voneinander sind, kann der Benutzer dabei die Policy-Klassen für verschiedene Belange der Klasse frei kombinieren.

Beispiel: Erstellen neuer Objekte

Um die genaue Bedeutung der Begriffe „Policy“ und „Policy-Klasse“ zu erklären, wollen wir ein einfaches Beispiel betrachten. Ein Entwickler könnte auf die Idee kommen, das Erzeugen neuer Objekte in einem Klassen-Template namens `NewCreator` zu kapseln.

```
template <class T>
class NewCreator {
public:
  static T* Create() { return new T; }
};
...
Widget* w = NewCreator<Widget>::Create();
```

Das Klassen-Template benutzt den Typ-Parameter T wie einen Platzhalter: Wenn das Template auf die Klasse `Widget` angewendet wird, instanziiert der Compiler automatisch eine passende `NewCreator`-Klasse, indem er überall das T durch `Widget` ersetzt.

Für sich allein genommen macht der `NewCreator` natürlich wenig Sinn, und zudem hat er eine Eigenschaft, die für manche Anwendungsfälle ein Nachteil sein könnte: Er erzeugt alle seine Objekte mit Hilfe des Operators `new` und verwendet immer den Default-Konstruktor.

Eine andere Möglichkeit, den Speicher für das neue Objekt anzufordern, besteht in der Verwendung der Funktion `malloc`. Damit alle nötigen Konstruktoren aufgerufen werden, wird danach *placement new* verwendet. Bei *placement new* wird wird der Speicher nicht von `new` angefordert, sondern es wird ein Zeiger mit der Adresse angegeben, an der das Objekt erstellt werden soll. Weitere Alternativen sind denkbar, die z. B. bei der Erstellung Argumente an den Konstruktor liefern oder neue Objekte als Kopien eines vorgegebenen Prototypen erstellen.

```
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
  }
};
```

`MallocCreator` und `NewCreator` erfüllen beide den selben Zweck, neue Objekte zu erzeugen. Sie unterscheiden sich nur in der Art und Weise, wie dies geschieht. Auch wenn das nicht im Code explizit angegeben ist, so haben die beiden Templates doch eine gemeinsame Schnittstelle: Beide stellen nur eine statische Methode namens `Create` zur Verfügung, die ein neues Objekt der gewünschten Klasse zurückliefert.

Aufgrund dieser gemeinsamen Schnittstelle wäre es ohne weiteres möglich, `NewCreator` an allen Stellen im Programmcode textuell durch `MallocCreator` zu ersetzen, um das Verhalten des Programms bezüglich der Erstellung neuer Objekte auszutauschen. Da die Auswahl der Implementierung dieser Schnittstelle bei der Übersetzung stattfindet, spricht in diesem Zusammenhang auch von *statischer Polymorphie*.

Der Begriff „Policy“

Eine „Policy“ ist nichts anderes als eine Schnittstelle für Klassen oder Klassen-Templates, die einen bestimmten Belang einer Klasse kapseln. Die Schnittstelle besteht dabei aus Syntax und Semantik der Methoden, aber auch von Membervariablen und Typdefinitionen.

Policy-Klassen sind Implementierungen einer Policy-Schnittstelle. Da Policies sehr häufig Schnittstellen für Klassen-Templates sind, ist die Bezeichnung Policy-Klasse etwas irreführend: Sehr häufig handelt es sich bei Policy-Klassen nicht um Klassen, sondern um Klassen-Templates. Policy-Klassen sind üblicherweise nicht dazu gedacht, für sich selbst zu stehen, sondern werden in die Host-Klassen eingebunden.

Um auf das obige Beispiel zurückzukommen, kann man also sagen, dass `NewCreator` und `MallocCreator` beide Policy-Klassen einer Policy zum Erstellen neuer Objekte sind. Die Implementierungen der Policy, nennen wir sie *Creation Policy*, müssen Klassen-Templates sein. Abgesehen davon erwartet diese einfache Schnittstelle nur eine Methode `Create`, die eine ohne Argumente aufgerufen wird und die ein neues Objekt zurückliefert.

Für jede Policy kann es eine unbegrenzte Anzahl an Policy-Klassen geben. `NewCreator` und `MallocCreator` sind daher nur beispielhafte Implementierungen der *Creation Policy*. So lange sich alternative Policy-Klassen an die Schnittstelle der Policy halten, können sie später ohne Probleme an Stelle von `NewCreator` und `MallocCreator` verwendet werden.

Verwendung von Policies

Die Policy-Klassen werden bei der Übersetzung mit der Host-Klasse verbunden. Dazu muss die Host-Klasse als Klassen-Template geschrieben werden, das für jede Policy einen Parameter erhält. Vom Benutzer wird dann die Host-Klasse mit den von ihm gewünschten Policy-Klassen instanziiert. Als Beispiel hier das Klassen-Template `WidgetManager`, welche eine Menge von `Widgets` verwalten soll:

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

Da `WidgetManager` auch das anlegen neuer `Widgets` ermöglichen soll, liegt es nahe die bereits erwähnte *Creation Policy* zu verwenden. Daher wird die *Creation Policy* zum Parameter des Templates. `WidgetManager` benutzt die Methode `Create` des Template-Parameters um neue `Widgets` anzulegen. Dabei ist es egal, ob als *Creation Policy* nun `NewCreator` oder `MallocCreator` oder eine andere Policy-Klasse übergeben wird.

Üblicherweise wird der Benutzer zur Vereinfachung eine typedef-Deklaration verwenden:

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

bzw.

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

Bis jetzt hat `WidgetManager` aber noch einen Schönheitsfehler: Bei der Verwendung der Klasse muss der Benutzer zur Policy-Klasse stets den Parameter `Widget` mit angeben, obwohl aus Sicht des Entwicklers von `WidgetManager` eigentlich klar ist, dass `Widgets` angelegt werden sollen.

Dieses Problem lässt sich durch die Verwendung von Template-Template-Parametern (kurz TTPs) lösen. Diese ermöglichen es, statt eines konkreten Typs ein Template zum Parameter des Templates zu machen. Das bedeutet, dass nicht die

konkrete Instanz `NewCreator<Widget>` sondern das Template `NewCreator` als Parameter an `WidgetManager` übergeben werden kann. Dadurch kann `WidgetManager` selbst `NewCreator` mit dem notwendigen Typ instanziieren. Das sieht im Beispiel so aus:

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

`CreationPolicy` ist hier nicht mehr Platzhalter für eine Klasse, sondern für ein Klassen-Template. WidgetManager selbst ist für die Erzeugung der Instanz mit Parameter `Widget` verantwortlich.

Mit dem TTP kann der Benutzer den `WidgetManager` nun einfacher verwenden:

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

statt

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

Abgesehen von der Ersparnis an Schreibarbeit und der verbesserten Lesbarkeit hat die Verwendung des TTPs hier noch weitere Vorteile. So war die alte Schreibweise auch eine Möglichkeit für Fehler: Sollte ein unbedachter Programmierer einen anderen Template-Parameter als `Widget` angeben, so könnte es in `WidgetManager` zu Fehlern kommen. Zudem ist `WidgetManager` durch den TTP nicht darauf beschränkt, die Policy nur zum Verwenden von `Widgets` zu verwenden, sondern könnte auch Objekte anderer Klassen mit der selben Policy erzeugen.

Es sollte noch angedeutet werden, dass sich durch Ableitung der Host-Klassen von ihren Policy-Klassen weitere Möglichkeiten für Policies ergeben. So übernimmt die Host-Klasse etwa alle in den Policy-Klassen eingeführten Variablen und Methoden. Wenn eine Policy-Klasse die Host-Klasse um öffentliche Methoden erweitert, so spricht man von *Enriched Policies*. Policies sind also nicht nur auf Verhalten beschränkt, sie können auch zur Erweiterung der Struktur der Host-Klasse eingesetzt werden.

Aufteilung einer Klasse in Policies

Eine wichtige Entscheidung beim Entwurf einer Policy-basierten Klasse ist die Auswahl der Belange, die in einer Policy ausgelagert werden sollen. Um eine größtmögliche Flexibilität und damit Wiederverwendbarkeit der Klasse zu erreichen, sollten grundsätzlich alle mit Abwägungen verbundenen Belange der Klasse zu Policies werden. Da die Policies als einzelne Parameter des resultierenden Template aber unabhängig übergeben werden, muss darauf geachtet werden, dass keine Abhängigkeiten zwischen den verschiedenen Policies der Klasse auftreten.

Wenn die Auswahl der Policies abgeschlossen ist, so kann die Host-Klasse als Template implementiert werden. Dabei werden an den entsprechenden Stellen die Methoden (bzw. Variablen oder Typen) der Policy verwendet. Die Entscheidung, wo die Policy angewandt wird, liegt also nicht bei der Policy selbst, sondern in der Host-Klasse. Da die Host-Klasse mit Hilfe der Policies implementiert wird, ist sie zudem von diesen abhängig: Die spätere Einführung einer zusätzlichen Policy ist nicht ohne weiteres möglich. Aus diesem Grund ist es so wichtig, schon beim Entwurf die richtige Aufteilung in Policies vorzunehmen.

Beispiel: Policy-based Smart-Pointer

Als Beispiel für die Aufteilung einer Klasse in Policies werden wir die Smart-Pointer-Klasse `SmartPtr` betrachten (vgl. Alexandrescu S. 157ff). Ein Smart-Pointer ist eine Klasse, die einen einfachen („rohen“) Zeiger verwaltet, ihren Benutzern aber zusätzliche Funktionalität bietet. Im allgemeinen sollte sich eine Smart-Pointer-Klasse in ihrer Verwendung wie der zugrundeliegende Zeiger verhalten, ein Smart-Pointer sollte also z.B. dereferenzierbar sein.

Smart-Pointer sind eine schöne Anwendungsmöglichkeit für Policy-based Design, da hier viele nicht eindeutige Entwurfsentscheidungen zu treffen sind. Die hier vorgestellte Auswahl an Policies ist auch nur als Beispiel zu verstehen, es sind durchaus noch komplexere Aufteilungen denkbar.

In diesem Beispiel soll die Smart-Pointer-Klasse drei Policies verwenden. Die *Ownership Policy* soll die Verwaltung und den Besitz des Objekts übernehmen, auf das der rohe Zeiger zeigt. Eine *Checking Policy* gibt an, welchen automatischen Prüfungen der Smart-Pointer unterzogen werden soll. Schließlich wird die *Conversion Policy* festlegen, ob der Smart-Pointer sich direkt und implizit in den Typ des Zeigers umwandeln lässt.

Die *Ownership Policy* ist die komplexeste dieser drei Policies. Sie soll den Besitz des Objekts, auf das der Zeiger zeigt, verwalten. Es ist also ihre Aufgabe, sicher zu stellen, dass das Objekt genau dann freigegeben wird, wenn es nicht mehr benötigt wird. Gerade bei Kopien und Zuweisungen zwischen Smart-Pointern kann es recht schwierig werden festzustellen, wann ein Objekt nicht mehr benötigt wird. Zu diesem Zweck wurden verschiedene Strategien entwickelt, die alle ihre Vor- und Nachteile haben. Eine einfache Methode wäre hier `DeepCopy`: Dabei wird bei der Kopie des Zeigers auch eine Kopie des Objekts erstellt, auf das gezeigt wird. Somit können verschiedene Smart-Pointer nie auf das gleiche Objekt zeigen, und jeder Smart-Pointer kann sein Objekt in seinem Destruktor freigeben. Die Brauchbarkeit dieses Ansatzes ist natürlich beschränkt, da das ständige Kopieren der Objekte dem Sinn von Zeigern in Frage stellt. Eine fortschrittliche Methode ist das Zählen von Referenzen auf die Objekte (Policy-Klasse `RefCounting`), hier gibt es für jedes Objekt einen Zähler, der angibt, wie viele Smart-Pointer auf ein Objekt zeigen. Erreicht dieser Wert 0, so kann das Objekt freigegeben werden. Nachteile hierbei sind der Aufwand und das Scheitern bei zyklischen Bezügen zwischen Objekten. Als dritte Möglichkeit einer Policy-Klasse sei noch `DestructiveCopy` angeführt. Hier geht der Besitz am Objekt bei einer Zuweisung an den Ziel-Pointer über, der ursprüngliche Smart-Pointer wird invalidiert. Ein Beispiel einen Smart-Pointer der nach dem Prinzip von `DestructiveCopy` arbeitet, ist der Standard-C++-Datentyp `std::auto_ptr`.

Neben der Verwaltung des Objektbesitzes kann man Smart-Pointer auch zur automatischen Überprüfung des Zeigers verwenden. So ist beispielsweise ein Test auf einen Nullpointer vor einer Dereferenzierung denkbar. Dieses Verhalten soll von der *Checking Policy* übernommen werden. Dazu ist es natürlich notwendig, dass die Host-Klasse an geeigneten Stellen die Überprüfungsmethoden dieser Policy aufruft. Mögliche Policy-Klassen wären eine `NoCheck`-Klasse, bei der alle Überprüfungsmethoden einfach leer sind um die Performance nicht zu beeinflussen, und eine `EnforceNotNull`-Klasse, die sicher stellt, dass bei einer Dereferenzierung kein Nullpointer verwendet wird. Es wäre vorstellbar,

verschiedene Arten von `EnforceNotNull` zu schreiben, die sich in der Art der Reaktion auf den Fehler unterscheiden.

Als letztes soll noch eine *Conversion Policy* verwendet werden, welche angibt, ob der Smart-Pointer Operatoren so überladen soll, dass er sich implizit vom Compiler in den Typ des von ihm verwalteten Zeigers umwandeln lässt. Während dieses Feature oft sehr komfortabel sein mag, hat es auch Nachteile. So kann man dadurch einfach an den rohen Zeiger gelangen, der eigentlich in der Implementierung versteckt sein sollte. Zusätzlich besteht die Gefahr, dass durch das Verwenden dieses rohen Zeigers die Mechanismen zur Verwaltung des Objektbesitzes ausgehebelt werden und Fehler verursachen. Die Entscheidung, ob die automatische Typkonvertierung erlaubt werden soll, wird also dem Benutzer des Smart-Pointers überlassen. Für diese Policy kommen eigentlich nur zwei Policy-Klassen in Frage: `AllowConversion` und `DisallowConversion`.

Die Deklaration des Klassen-Template für `SmartPtr` wird so aussehen:

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

Es fällt auf, dass die Policies wieder Klassen-Templates erwarten, da sie den Typ des Zeigers selbst als Parameter benötigen. Die Ausnahme hier ist die `ConversionPolicy`, da sie keine Information über den Typ des Zeigers haben muss, ist sie als einfacher Klassenparameter aufgeführt.

Zum Einsatz des Templates werden später am besten typedefs verwendet:

```
typedef SmartPtr<Widget, RefCounted, NoCheck, AllowConversion>
   WidgetPtr;
...
WidgetPtr w(new Widget);
w->doSomething();
```

In diesem Fall handelt es sich also um einen `SmartPtr`, der auf `Widget`s zeigt, Reference Counting benutzt, keine automatischen Überprüfungen ausführt und implizite Umwandlung in den Typ `Widget*` erlaubt. Sollten sich die Anforderungen ändern, so kann `WidgetPtr` in den von Policies abgedeckten Bereichen leicht angepasst werden.

Da die gesamte Implementierung der Klasse `SmartPtr` und der dazugehörigen Policy-Klassen recht umfangreich ist, wollen wir uns mit der Betrachtung der Implementierung der *Ownership Policy* begnügen. Damit die ausgewählte *Ownership Policy* feststellen kann, wann ein von Smart-Pointern verwaltetes Objekt freigegeben werden kann, sind nur zwei Methoden in der Policy nötig: Die Methode `Clone` wird von einem `SmartPtr` verwendet, der eine Kopie des verwalteten Zeigers erstellt werden soll. Diese Methode ist nötig, da das Kopieren eines Zeigers die Anzahl der Besitzer des Objekts erhöht, auf das der Zeiger zeigt. Die andere Methode ist die Methode `Release`, die aufgerufen wird, wenn ein `SmartPtr` seinen Besitz am Objekt aufgibt. Der Rückgabewert gibt an, ob das Objekt dadurch alle Besitzer verloren hat und damit freigegeben werden soll.

So implementiert die Policy-Klasse `DeepCopy` diese Methoden:

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

Es wird also in `Clone` nicht eine Kopie des Zeigers, sondern immer eine Kopie des Objekts erstellt. Zurückgegeben wird ein Zeiger auf das neue Objekt. Dadurch haben altes und neues Objekt nach der Operation genau einen Besitzer, beim Freigeben des Zeigers mit `Release` kann also immer auch das Objekt freigegeben werden.

Die Policy-Klasse `RefCounting` sieht hingegen so aus:

```
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;
    }
```

```
    return false;
  }
};
```

Bei jedem `Clone` wird der Referenzzähler erhöht, und `Release` veranlasst das freigeben des Objekts nur dann, wenn der Zähler 0 erreicht hat. Da es einen Zähler für jedes verwaltete Objekt (und nicht für jeden Zeiger) geben soll, verdient die Verwaltung des Zählers besonderes Interesse. Da jeder `SmartPtr` seine eigene Instanz von `RefCounted` haben wird, wird bei der Kopie eines `SmartPtrs` auch seine `RefCounted`-Instanz und damit der Zeiger auf den Zähler mitkopiert. Wird ein `SmartPtr` jedoch nicht als Kopie angelegt, wird der Default-Konstruktor von `RefCounted` angewendet, wo ein neuer Zähler angelegt wird. Dieses Verhalten führt also dazu, dass jeweils alle `SmartPtr`-Kopien, die auf dasselbe Objekt zeigen, einen Zähler teilen.

Damit die *Ownership Policy* funktionieren kann, ist es natürlich noch notwendig, dass die Host-Klasse an den entsprechenden Stellen die Methoden der Policy-Klassen aufruft. Ein Auszug aus der `SmartPtr`-Definition:

```
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;
  }
};
```

Damit jeder `SmartPtr` seine eigene Instanz der `OwnershipPolicy`-Klasse erhält, wird hier Implementierungsvererbung verwendet. Der Copy-Kontruktor verwendet `Clone`, um für sich eine Kopie vom Zeiger des Original-Smart-Pointers zu erstellen. Zudem stellt er sicher, dass auch die `OwnershipPolicy` vom Original kopiert wird. Im Destruktor des `SmartPtrs` wird `Release` aufgerufen, um den Besitz des Objekts aufzugeben. Wenn `Release` zurückgibt, dass das Objekt auch keine anderen Besitzer mehr hat, so wird es freigegeben.

Interessanterweise ist es nicht nötig, `Clone` oder `Release` an weiteren Stellen zu verwenden, da sich z.B. der Zuweisungsoperator auch mit Hilfe des Copy-Konstruktors implementieren lässt.

Die beispielhafte Implementierung der Ownership Policy verdeutlicht die Abhängigkeit der Host-Klassen von den Schnittstellen der verwendeten Policies. Beim Entwurf einer generischen Klasse wie unserem `SmartPtr` mit Hilfe von Policy-based Design, werden nach der Auswahl der Policies zunächst die Schnittstellen der Policies definiert, und die Host-Klasse wird entsprechend dieser Schnittstellen entworfen. Es ist entsprechend schwer, nachträglich zusätzliche Policies in eine derartige Klasse einzubringen.

Die Host-Klasse verwendet in ihrer Implementierung die von der Policy bereitgestellten Methoden, Variablen und Typdefinitionen. Damit gibt sie aber auch die Verbindungspunkte zwischen Policy-Klasse und Host-Klasse vor; eine Policy-Klasse kann nicht selbst den Ort ihrer Anwendung bestimmen. Zudem wird dadurch erreicht, dass in der Host-Klasse Programmcode zur Bereitstellung der Hauptfunktionalität der Klasse und Code zum Einbinden der Policy-Klassen stark vermischt ist. Dies widerspricht eigentlich dem Grundsatz der Trennung der Belange und macht es schwierig, mit Policy-based Design querschneidende Belange zu modularisieren.

Zusammenfassung

Policy-based Design erlaubt die Aufteilung einer Klasse in Policies, die einzelne Belange der Klasse kapseln. Durch die Verwendung von Policies gewinnt eine Klasse an Flexibilität und Wiederverwendbarkeit, da sie sich in ihrem Verhalten an die Anforderungen anpassen lässt.

Ein Vorteil liegt in der Tatsache, dass Policy-based Design auf reinem C++ basiert und daher keine zusätzlichen Hilfsmittel nötig werden. Zudem müssen die Entwickler keine neue Sprache lernen, wobei allerdings einige der verwendeten Konstrukte dennoch nicht ganz einfach zu verstehen sind.

Nachteilig auf die Modularisierung wirkt sich aus, dass die Verwendung der Policies im Code der Host-Klasse fest verankert ist. Das macht das Hinzufügen neuer Policies schwierig und erlaubt es kaum, die Stellen zu verändern, an denen die Policies angewendet werden. Querschneidende Belange lassen sich daher auch nur schlecht mit Hilfe von Policies ausdrücken.

Quellen

Alexandrescu, Andrej (2001): „Modern C++ Design“, Addison-Wesley

Koenig, Andrew (2000): „Accelerated C++“, Addison-Wesley

Lippman, Stanley B. (2000): „Essential C++“, Addison-Wesley