Aspektorientierte Systemprogrammierung

Aspect-Oriented Design Patterns

Rainer Sand
rainer.sand@gmx.de

17.05.2004

1. Einleitung

Heutige Softwareprodukte haben längst einen Stand an Komplexität erreicht, der es für Firmen immer schwieriger bzw. kostspieliger werden lässt, entwickelte Software zu pflegen, zu warten und zu erweitern. Um so wichtiger wird es bereits zu Beginn, also bei der Entwurfsplanung, Methoden einzusetzen, die es erlauben, flexible, wiederverwendbare und einfach zu pflegende Softwareentwürfe herzustellen.
Dieser Text befasst sich zunächst mit den klassischen Entwurfsmustern (der sog. „Gang-of-Four"), versucht, deren Probleme bei der objektorientierten Implementierung anhand von Beispielen aufzuzeigen, und stellt schließlich Lösungen mit Konzepten der Aspektorientierten Programmierung vor.

2. Was ist ein Entwurfsmuster?

Auf diese Frage lässt sich am Besten mit einem Zitat von Christopher Alexander antworten: „Jedes Muster beschreibt ein in unserer Umwelt beständiges wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen." [AIS++77, Seite x]
Entwurfsmuster sollen also helfen, oft wiederkehrende Problemstellungen so zu lösen, dass der entstehende Entwurf maximale Flexibilität für zukünftige Änderungen aufweist und für ähnliche Anforderungen wiederverwendet werden kann. Erich Gamma beschreibt in seinem Buch [1] 23 Entwurfsmuster, katalogisiert sie, bewertet sie und zeigt Implementationen sowie Anwendungsfelder dafür auf. Er unterscheidet zwischen Erzeugungsmustern, wie zum Beispiel der *„abstract factory"*, Strukturmustern wie dem *„adapter"* und Verhaltensmustern wie dem *„observer"* oder der *„strategy"*. Für ihn besitzen Entwurfsmusterbeschreibungen vier grundlegende Elemente:

- **Mustername:** vermittelt knapp und präzise den wesentlichen Gehalt des Musters
- **Problemabschnitt:** beschreibt in welchem Kontext das Mustern anzuwenden ist bzw. welches Problem adressiert wird
- **Lösungsabschnitt:** beschreibt die Elemente, aus denen der Entwurf besteht sowie ihre Beziehungen, Zuständigkeiten und Interaktionen
- **Konsequenzabschnitt:** diskutiert, wie das Muster seine Ziele zu erreichen versucht und welche Vor- und Nachteile sich durch die Anwendung des Musters ergeben

Anhand weiterer Punkte wie Zweck, Anwendbarkeit, Struktur, Teilnehmer, Interaktionen und Beispielcode soll im folgenden Kapitel der Beobachter (engl. Observer) im Detail erläutert werden.

3. Beschreibung eines Entwurfsmuster am Beispiel des „Beobachters“

Als **Motivation** soll die Konstruktion von Benutzungsschnittstellen in Smalltalk-80 genannt werden, welche das sog. MVC-Paradigma verwendet, das aus drei Klassen, nämlich Model, View und Controller besteht. Das Model-Objekt stellt das Anwendungsobjekt dar, das View-Objekt seine Bildschirmrepräsentation und das Controller-Objekt bestimmt die Möglichkeiten, mit denen die Benutzungsschnittstelle auf Benutzungseingaben reagieren kann – legt also das Kommunikations-protokoll zwischen Model und View fest.
Ziel ist es, die View- und Model-Objekte zu entkoppeln. Je weniger die Klassen voneinander wissen, desto besser lassen sie sich auch später erweitern bzw. gegen andere austauschen. Ihre Kommunikation sollte nach Möglichkeit nur noch über wohldefinierte Schnittstellen erfolgen. Weiterhin soll die Option bestehen, dem Model-Objekt beliebig viele Views zuzuordnen, ohne dass es davon Kenntnis hat, wie viele oder welche dies sind. Diese Art der Interaktion ist auch als Publish-Subscribe bekannt. Das Subjekt ist der Publizierer von Nachrichten, die es aussendet, ohne zu wissen, wer die Beobachter sind. Ein Beispiel sind Daten aus einer Tabellenkalkulation (Model), die sowohl als Tabelle, Kuchen- und Säulendiagramm (Views) repräsentiert werden können (Abbildung 1). Zur Vereinfachung wurde hier der Controller weg-gelassen, um die zwei wesentlichen Teilnehmer, Subjekt und Beobachter, herauszustellen, die bereits hier schematisch das Entwurfsmuster „Observer“ darstellen.

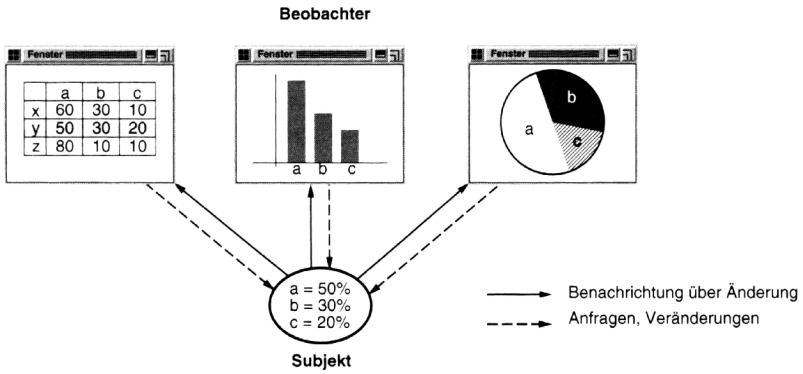

Abbildung 1

Zweck des „Observer“-Musters ist es in einer 1-zu-n-Abhängigkeit zwischen Objekten dafür zu sorgen, dass bei einer Änderung des Zustandes alle abhängigen Objekte automatisch aktualisiert werden. Hierfür steht dem Beobachter-Objekt eine *Aktualisiere()*-Methode zur Verfügung, die das Subjekt über *Benachrichtige()* zur Ausführung bringt, wenn es seinen Zustand ändert. Für jedes Subjekt können mit den Methoden *MeldeAn()* und *MeldeAb()* mehrere Beobachter registriert oder entfernt werden. Dies wird in der **Musterstruktur** (Abbildung 2) nochmal grafisch aufbereitet.

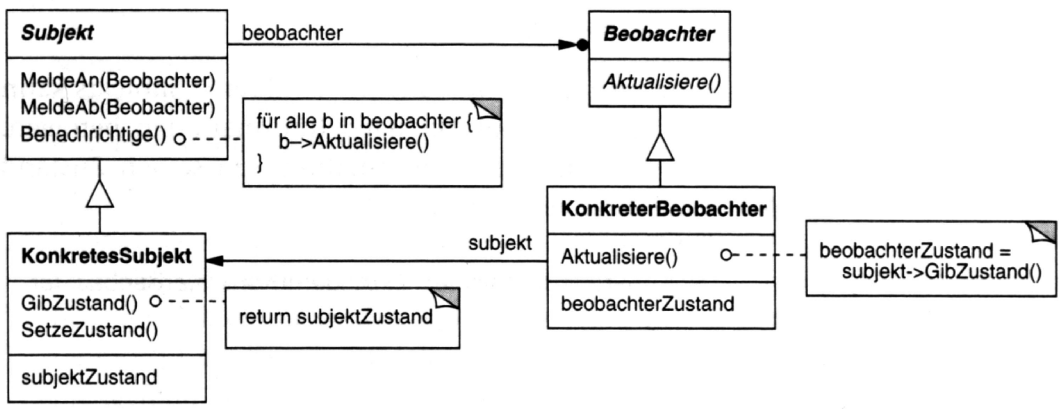

Abbildung 2

Aus dieser Struktur lassen sich leicht die nötigen **Teilnehmer** identifizieren. Beide Bereiche bilden den Lösungsabschnitt. Zum einen findet sich ein Subjekt, welches seine Beobachter kennt und eine Schnittstelle zum An- und Abmelden dieser anbietet. Der Beobachter definiert eine Aktualisierungsschnittstelle für Objekte, die über Änderungen eines Subjekts benachrichtigt werden sollen. Das KonkreteSubjekt speichert den für den KonkretenBeobachter relevanten Zustand und benachrichtigt seine Beobachter, wenn sich sein Zustand ändert. Zuletzt verbleibt der KonkreteBeobachter, der eine Referenz auf ein KonkretesSubjekt verwaltet. Zudem speichert er den Zustand, der mit dem des Subjekts in Einklang stehen soll und implementiert die Aktualisierungsschnittstelle der Beobachterklasse.
Über die **Anwendbarkeit** (Problemabschnitt) des Musters lassen sich drei wesentliche Aussagen treffen. Der „Beobachter" kann verwendet werden:
1. wenn eine Abstraktion zwei Gesichtspunkte besitzt, von denen der eine von dem anderen abhängt und eine Kapselung dieser in unterschiedliche Objekte möglich ist
2. wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt und man nicht weiß, wie viele geändert werden müssen
3. wenn ein Objekt in der Lage sein sollte, andere Objekte zu benachrichtigen, ohne Annahmen darüber treffen zu dürfen, wer diese anderen Objekte sind.

Interaktion zwischen den Teilnehmern ist eine Grundvoraussetzung der meisten Entwurfsmuster. Im Falle des Observers findet man folgende zwei Interaktionen. Zum einen benachrichtigt das konkrete Subjekt seine Beobachter, wenn eine Zustandsänderung stattfindet, die deren Zustand in Bezug auf den eigenen inkonsistent machen könnte. Zum anderen kann der konkrete Beobachter nach einer Benachrichtigung Informationen vom Subjekt abfragen (z. B. *gibZustand()*, Abb. 3). Er benutzt diese Informationen, um seinen Zustand mit dem des Subjekts zu synchronisieren..

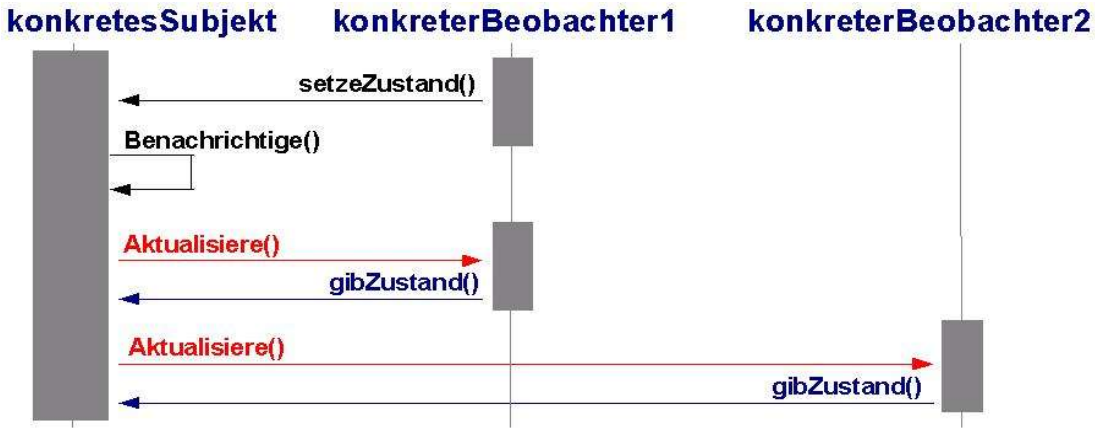

Abbildung 3

Der **Konsequenzabschnitt** beschreibt nun weitere Vorteile aber auch Verpflichtungen, die sich durch den Einsatz des Observer-Musters ergeben. Ein Punkt ist die Unterstützung einer Broadcast-Kommunikation. Anders als eine normale Anfrage, muss die von einem Subjekt abgeschickte Nachricht ihre Empfänger nicht spezifizieren, da die Benachrichtigung automatisch an die interessierten Beobachter verteilt wird. Das Subjekt beachtet nicht weiter, wie viele interessierte Objekte vorhanden sind; es sorgt lediglich dafür, die Benachrichtigung zu verschicken. Der Beobachter ist dafür zuständig, diese Nachricht zu ignorieren oder zu bearbeiten.
Ein Problem stellen beim Observer unerwartete Aktualisierungen dar. Da Beobachter nichts voneinander wissen, haben sie auch keine Vorstellung davon, was eine Änderung eines Subjekts insgesamt kostet. Eine scheinbar harmlose Operation auf dem Subjekt kann zu einer Kaskade von Aktualisierungen bei den Beobachtern und von ihnen abhängigen Objekten führen. Durch schlecht definierte Abhängigkeiten können dadurch zumeist unsinnige, aber auch folgenschwerere Aktualisierungen entstehen.

Da es für ein Entwurfsmuster keine universelle **Implementierung** gibt, gliedert man diesen Bereich in zwei Teile. Im ersten werden allgemeine Möglichkeiten der Implementierungen diskutiert, der zweite beschreibt anhand von Beispielcode eine konkrete Anwendung.
Ein möglicher Punkt, der bei der Implementierung eine Rolle spielt, ist die Abbildung von Subjekten auf ihre Beobachter. Eine für das Subjekt einfache Lösung ist, sich eine Referenz auf die zu benachrichtigenden Beobachter zu merken. Diese Speicherung kann allerdings zu teuer werden, wenn viele Subjekte und wenige Beobachter existieren. Eine Alternative hierzu ist den großen Speicherplatzaufwand gegen einen höheren Zeitverbrauch auszutauschen, indem man ein assoziatives Lookup verwendet (zum Beispiel eine Hash-Tabelle), um die Abbildung vom Subjekt auf seine Beobachter zu verwalten. Zwar nimmt ein Subjekt dann ohne Beobachter keinen Speicher mehr in Anspruch, aber die Kosten für den Zugriff auf Beobachter nehmen zu.
Ein weiterer Gesichtspunkt, der zur Debatte steht, ist das Problem mit fehlerhaften Referenzen auf gelöschte Subjekte in den Beobachtern. In der Regel ist das Löschen der Beobachter keine Lösung, da andere Objekte sie möglicherweise referenzieren, oder weil sie zusätzlich weitere Subjekte beobachten. Daher sollte ein Subjekt die Beobachter informieren können, wenn es gelöscht wird, sodass die Referenz darauf zurückgesetzt werden kann.
Folgender **Beispielcode** definiert als Subjekt einen Zeitgeber. Als Beobachter werden eine Digital- und eine Analoguhr ergänzt, die bei einer Zeitveränderung durch die Methode *Tick()* informiert werden sollen.
Eine abstrakte Klasse definiert dafür die Schnittstelle der Beobachterklasse:

```
class Observer {
   public:
     Observer() {};
     ~Observer() {};
     virtual void Update(Subject* theChangeSubject) = 0;
};
```

Diese Implementierung unterstützt mehrere Subjekte für jeden Beobachter. Das an die Update-Operation weitergereichte Subjekt ermöglicht es dem Beobachter festzustellen, welches Subjekt sich geändert hat.
Folgende abstrakte Klasse definiert die Schnittstelle der Subjektklasse:

```
class Subject {
   public:
     Subject() {};
     ~Subject() {};

     void addObserver(Observer*);  // meldet Observer beim Subjekt an
     void removeObserver(Observer*);  // meldet Observer beim Subjekt ab
     void Notify();        // informiert alle registrierten Observer

   private:
     vector<Observer*> _observers;     // Liste der registrierten Observer
};

   void Subject::Notify () {
      int count = _observers.size();

     for (int i = 0; i < count; i++)
       (_observers[i])->Update(this);  // jeder Observer wird aktualisiert
   }
```

Der Zeitgeber (*ClockTimer*) ist eine konkrete Subjektklasse zum Speichern und zum Verwalten der Tageszeit. Er bietet eine Schnittstelle zum Abfragen einzelner Zeiteinheiten wie zum Beispiel der Stunde, Minute und Sekunde.

```
class ClockTimer : public Subject {
   public:
    ClockTimer() { /* initialisire den Zeitstempel */ };

    int GetHour();   // liefert Stunde (verwendet von Observer)
    int GetMinute(); // liefert Minute (verwendet von Observer)
    int GetSecond(); // liefert Sekunde (verwendet von Observer)

    void Tick();  // erfasst aktuelle Zeit

   private:
    char timebuf[128]; // speichert den Zeitstempel
};
```

Die *Tick*-Operation wird in regelmäßigen Abständen von einem internen Taktgeber aufgerufen, um eine akurate Zeitbasis bereitzustellen. *Tick()* aktualisiert den internen Zustand des *ClockTimers* und ruft *Notify()* auf, um Beobachter über die Zustandsänderung zu informieren:

```
   void ClockTimer::Tick() {
       // Aktualisiere den internen Zustand
       // ...
       Notify();
   }
```

Wir können jetzt die Klasse *DigitalClock* definieren, welche die Tageszeit anzeigt. Die Schnittstelle der Beobachterklasse wird der *DigitalClock* durch Mixin-Erben von *Observer* hinzugefügt.

```
class DigitalClock: public Observer {
   public:
    DigitalClock(ClockTimer *);  // Aufgabe:  _s->addObserver (this);
    ~DigitalClock();             // Aufgabe:  _s->removeObserver (this);

       void Update(Subject *);   // aktualisiere Zeit

       void Draw();   // Zeit ausgeben mit Hilfe von _s->GetHour() etc.

   private:
    ClockTimer *_subject;   // Referenz auf das Subjekt
};
```

Bevor nun die *Notify*-Operation die Uhr darstellen lässt, überprüft sie zuerst, ob das benachrichtigende Subjekt auch das beobachtete Subjekt der Uhr ist:

```
   void DigitalClock::Update (Subject *theChangedSubject) {
      if(theChangedSubject == _subject)
      Draw();
   }
```

Eine *AnalogClock*-Klasse kann auf dieselbe Art definiert werden:

```
class AnalogClock: public Observer {
   public:
     AnalogClock(ClockTimer *);
     ~AnalogClock();

      void Update(Subject *);
      void Draw();
      // ...
};
```

Der folgende Code erzeugt eine *AnalogClock* und eine *DigitalClock*, die immer dieselbe Tageszeit anzeigen. Der Aufruf von *Tick()* des *timers* veranlasst jetzt die grafische Ausgabe:

```
ClockTimer timer;

DigitalClock digitalClock(&timer); // Uhr erzeugen und timer registrieren
AnalogClock analogClock(&timer);  // Uhr erzeugen und timer registrieren

timer.Tick();  // timer bestimmt seine Zeit => Ausgabe durch die Uhren
```

5. Probleme des objektorientierten Musterentwurfs

Betrachtet man die beteiligten Klassen, so wird schnell klar, dass sowohl Pattern-Code als auch Verhaltenseigenschaften des ursprünglichen Objekts vermischt enthalten sind. Allen Teilnehmer-Klassen mussten Methoden hinzugefügt werden, so dass sie in der gewünschten Weise kommunizieren können. Dies bringt einige Nachteile in Bezug auf die Lesbarkeit des Quellcodes mit sich. Es wird für spätere Änderungen schwierig sein, Pattern-Code von Applikationscode zu unterscheiden. In der Literatur spricht man daher vom so genannten „**Confusion Problem**“. Die zunehmende Verzahnung der beiden Codetypen macht es in großen Projekten auch fast unmöglich, Objekte einmal mit bzw. ohne Pattern-Code im System zu testen. Des weiteren ergeben sich zumeist Schwierigkeiten bei der Anfertigung einer Dokumentation für Programme mit vielen Entwurfsmustern.
Mit „**Indirection Problem**“ bezeichnet man Nebeneffekte, die durch Delegation entstehen. Zum einen kann die Performance eines Programms durch zu viel bzw. ungewollte Kommunikation leiden, zum anderen wird auch der Quellcode unverständlich und schwerer lesbar, gerade wenn dem Leser die verwendeten Muster bzw. deren Bedeutung in dem Kontext nicht bekannt sind.
Das „**Encapsulation Breaking Problem**“ findet sich überall dort, wo delegierte Methodenaufrufe dazu führen, dass das aufrufende Objekt Zugriff auf private bzw. geschützte Attribute des aufgerufenen Objekts benötigt. Hierzu müssen diese öffentlich zugänglich gemacht werden, was in den meisten Fällen jedoch nicht möglich ist bzw. nicht möglich sein sollte.
Unter den „**Inheritance Related Problems**“ fasst man jene Probleme zusammen, die durch oder mit Vererbung entstehen. Die meisten Softwareprodukte besitzen bereits eine ausgeprägte Klassen-hierarchie. Unterstützt nun die verwendete Programmiersprache zum Beispiel keine Mehrfachvererbung, kann man nicht ohne weiteres ein Entwurfsmuster hinzufügen. Ebenso können Konflikte beim Versuch auftreten einer Klasse mehrere Muster bzw. Musterrollen durch Vererbung zuzuweisen, da es hierbei zu kritischen Überschneidungen und Abhängigkeiten kommen kann.
Die vier genannten Arten von Problemen lassen sich zum größten Teil auf zwei wesentliche Ursachen reduzieren: zum einen die mangelte Trennung von Pattern-Code und Applikationscode und zum anderen der Verlust der Datenkapselung aufgrund der Delegation.
Genau hier ist der Ansatzpunkt der Aspektorientierten Programmierung, deren Konzepte elegante Lösungen für diese letzten beiden Punkte versprechen.

6. Lösung mit Konzepten der Aspektorientierten Programmierung

Der Pattern-Code ist ein sog. „**crosscutting concern**“. Will man die oben genannten Probleme lösen, so empfiehlt es sich, den eigentlichen Code für das Entwufsmuster zu identifizieren und anschließend in ein separates Modul, in unserem Fall ein Aspekt, zu kapseln.
Würden wir jetzt die Notify(), addObserver() und removeObserver()-Methoden aus den Subjekten entfernen und in einen Aspekt packen, so wäre zwar das Confusion Problem gelöst, aber der eigentliche Sinn des Entwurfsmusters, die Wiederverwendbarkeit, wäre nicht mehr gewährleistet, da der entstehende Aspekt nur unser Beispielszenario zufrieden stellen würde. Wir müssen also zuerst

die allgemeinen Observer-Strukturen, die generell für das Muster gelten, herausarbeiten und daraufhin eine geeignete Implementierung zurechtlegen.
Typisch für jeden Observer ist auf jeden Fall die grundlegende Existenz eines Subjekts und eines Beobachters. Des weiteren muss bei jeder Benachrichtigung eine bestimmte Anzahl von Beobachtern informiert werden, d. h. wir brauchen eine Abbildung in Form von z. B. einer Liste, die die nötigen Informationen enthält, welche Beobachter benachrichtigt werden müssen. Als letztes Merkmal, welches allen Observer-Mustern anhaftet, ist die Update-Logik zu nennen. Jede zukünftige Implementierung des Musters wird auf bestimmte Ereignisse einen Trigger anstoßen müssen, der veranlasst, dass alle Beobachter aktualisiert werden.
Damit kommen wir zu den spezifischen Belangen, die für jeden Anwendungsfall separat entschieden werden müssen. Zuerst ist hier die Rollenverteilung zu nennen. Es muss festgelegt werden, welche Klassen bzw. Objekte als Beobachter und als Subjekt fungieren dürfen. Dabei sind konkrete Namen zu nennen, z.B. *DigitalClock* darf Beobachter, *ClockTimer* soll Subjekt sein. Weiterhin müssen die einzelnen Ereignisse spezifiziert werden, die den Update-Mechanismus auslösen können. Im Beispiel wäre es der Aufruf der Tick()-Methode, die die Clock-Objekte veranlassen sollte, ihre Zeit zu aktualisieren. Als dritten Punkt benötigt man noch das Wissen über den genauen Updatevorgang, d. h. es ist die Art und Weise zu implementieren, wie der Beobachter zu aktualisieren ist. Für unser Beispiel genügt es, den internen Zustand für die Zeit (*hour*, *minute* und *second*) neu zu setzen.
Mit diesen Vorüberlegungen ist es nun möglich, ein allgemeines Observer-Muster mit Konzepten der Aspektorientierten Programmierung zu entwerfen.
Im Folgenden wird das abstrahierte Observer-Muster, implementiert in AspectC++ [5], betrachtet. Als Kapselung wird ein abstrakter Aspekt (*ObserverPattern*) benutzt, der erst in seiner späteren konkreten Ableitung (*ClockObserver*) die anwendungsspezifischen Verhaltensweisen bestimmt.

```
1 aspect ObserverPattern {
2
3  public:
4     class Subject {};  // Schnittstellen des Subjekts
5     class Observer {   // Schnittstellen des Beobachters
6       public:
7         virtual void update (Subject *) = 0; // Methode für Aktualisierung
8     };
9
10 private:
11  SubjectMap _perSubjectObservers;  // Subjekt hat eine Menge von Observern
12   // vom konkreten (abgeleiteten) Aspekt bereitzustellen:
13  pointcut virtual subjectChange (Subject &subject) = 0;
14
15  virtual void updateObserver (Subject *subject, Observer *observer) = 0;
16
17 public:
18  void addObserver (Subject *subject, Observer *observer) { /* ... */ }
19  void removeObserver (Subject *subject, Observer *observer) { /* ... */ }
20
21  advice subjectChange (subject) : after (Subject &subject) {
22    // finde die Observermenge oset zu dem Subjekt subject
23    // ...
24    for (ObserverSet::iterator iter=oset.begin (); iter!=oset.end (); ++iter)
25      updateObserver (&subject, *iter);
26  }
27};
```

In den Zeilen 4 bis 8 definiert man die beiden nötigen Schnittstellenklassen *Subject* und *Observer* als Teilnehmerrollen, welche später den jeweiligen Klassen zugeordnet werden müssen. Die Abbildung bzw. das Mapping zwischen den Subjekten und den Beobachtern wird durch eine Datenstruktur *SubjectMap* (Zeile 11) gelöst, die zu jedem Subjekt eine Menge von Beobachtern speichert. In Zeile 13 wird durch einen abstrakten Pointcut die Schnittstelle zur Benachrichtigung von

Zustandsänderungen definiert. Der zugehörige after-Advice in Zeile 21 bis 26 implementiert die zentrale Update-Logik des Musters. Er formuliert die Regel, dass nach jedem Aufruf von Methoden, die im Pointcut *subjectChange()* spezifiziert sind, alle registrierten Beobachtern aktualisiert werden. Hierzu wird *updateObserver()* mit den jeweiligen Parametern für alle interessierten Beobachter aufgerufen. Diese Methode wird als (abstrakte) Schnittstelle in Zeile 15 deklariert und muss in der entsprechenden Ableitung anwendungsfallbezogen implementiert werden. Um sich bei einem Subjekt zu registrieren, muss die Methode *addObserver()* (Zeile 18) bzw. zum Abmelden *removeObserver()* (Zeile 19) verwendet werden.

Das nächste Codefragment zeigt den konkreten Aspekt. Der *ClockObserver* wird von dem abstrakten *OberserverPattern* abgeleitet (Zeile 1). Dort legt man zuerst die Methoden fest, die eine Benachrichtigung der Beobachter auslösen sollen. In Zeile 3-4 wird daher der Pointcut auf die *Tick*-Operation fixiert. Damit haben wir implizit dem *ClockTimer* die Rolle des Subjekts gegeben, was im Introduction-Advice (Zeile 9) auch explizit geschieht. Anschließend wird die Teilnehmerrolle des Beobachters an *DigitalClock* und *AnalogClock* zugewiesen (Zeile 6, 10). Die, durch den Aspekt festgelegte, Schnittstellenmethode *updateObserver()* wird in Zeile 16 bis 18 unter Verwendung eines weiteren Interfaces, nämlich der *update()*-Operation des Observers, implementiert. Dabei kommt es schließlich zum Aufruf von *Draw()* (Zeile 13), sodass automatisch die Uhrzeit bei jeder Aktualisierung ausgegeben werden kann.

```
1  aspect ClockObserver : public ObserverPattern {
2
3    pointcut subjectChange (Subject &subject) =
4     execution ("void ClockTimer::Tick()") && that (subject);
5
6    pointcut observers() = "DigitalClock"||"AnalogClock";
7
8   public:
9    advice "ClockTimer" : baseclass (Subject);  // ClockTimer ist Subjekt
10   advice observers () : baseclass (Observer); // die Uhren sind Beobachter
11
12   advice observers () : void update (ObserverPattern::Subject *subject) {
13      Draw ((const ClockTimer &)*subject);
14   }
15
16   virtual void updateObserver(Subject *subject, Observer *observer) {
17      observer->update (subject);
18   }
19 };
```

Es ist uns somit gelungen, ein allgemeines Beobachter-Muster mit Hilfe eines Aspekts so zu kapseln, dass Pattern-Code und Applikationscode in getrennten Modulen zur Verfügung stehen. Dies bedeutet auch, dass das Confusion Problem gelöst wurde. Durch die Möglichkeit der Einfügungen (Introductions), die uns erlauben, neue Basisklassen nachträglich einzubinden, können auch die meisten Inheritance Related Problems entkräftet werden. Das gleiche gilt für das Encapsulation Breaking Problem, da mit einem Advice neue Methoden in bereits existierenden Klassen eingebracht werden können. Diese haben dann die Möglichkeit auf privaten Elementen innerhalb des Klassenkontextes zu operieren, ohne dass diese explizit der Öffentlichkeit zugänglich gemacht werden müssen. Das Indirection Problem wird ebenfalls durch das statische Einbinden der Methoden gelöst, da nun nicht mehr auf Delegation als Mittel der Kommunikation zwischen Objekten zurückgegriffen werden muss.

7. Zusammenfassung

Durch den Einsatz von neuen, aspektorientierten Programmiertechniken wurde aufgezeigt, wie sich die Probleme klassisch implementierter Entwurfsmuster auf elegante und wiederverwendbare Art und Weise lösen lassen. Bei ihren Forschungen konnten Jan Hannemann und Gregor Kiczales [3] für 17 von 23 Entwurfsmustern modularisierte Aspekte in AspectJ entwickeln und meist den gesamten Pattern-Code aus den Teilnehmerklassen extrahieren. Allerdings bauen diese Aspekte auf individuellen Konzepten von AspectJ auf und können bei anderen aspektorientierten Programmiersprachen zu Entwurfsproblemen führen. Obwohl also noch einige Arbeit auf diesem Gebiet zu tun ist, so fehlt bis heute diese unabhängige Beschreibung der Gang-of-Four nach standardisierten AOP-Konzepten, ist es ein eindrucksvoller Anfang. Das Potential, dass hinter dieser neuen Entwurfsmethodik steht und wie dadurch zukünftige Designentwicklungen und -entscheidungen beeinflusst werden, wird deutlich.

8. Literaturhinweise

[1] Gamma, E. et al. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994

[2] Jan Hannemann, Gregor Kiczales
Design Pattern Implementation in Java and AspectJ
http://www.cs.ubc.ca/labs/spl/projects/aodps.html

[3] Ouafa Hachani, Daniel Bardou
Using Aspect-Oriented Programming for Design Patterns Implementation
http://www-lsr.imag.fr/OOIS_Reuse_Workshop/Papers/Hachani.pdf

[4] Ouafa Hachani, Daniel Bardou
On Aspect-Oriented Technology and Object-Oriented Design Patterns
www.comp.lancs.ac.uk/computing/users/chitchya/AAOS2003/Assets/hachani_bardou.pdf

[5] O. Spinczyk, A. Gal, und W. Schröder-Preikschat. AspectC++:
An Aspect-Oriented Extension to the C++ Programming Language. In *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of *Conferences in Research and Practice in Information Technology* ACS, 2002.
http://www.cse.unsw.edu.au/tools/