

Programmieren in C++ (6): Entwurfsmuster

Silke Seehusen, Fachhochschule Lübeck

Moderne Konzepte wie Entwurfsmuster (design pattern) bieten Hilfestellungen beim Entwurf und bei der Implementierung von objektorientierten Systemen. Es werden exemplarisch zwei Entwurfsmuster, der Observer und die Komposition, vorgestellt und eine mögliche Implementierung in C++ diskutiert.

Entwurfsmuster, Toolkits und Frameworks

In den letzten Jahren sind diverse Artikel und einige Bücher erschienen, in denen Entwurfsmuster beschrieben werden. Solche Entwurfsmuster sind keine neue Erfindung der letzten Jahre. Sie gehören zum Repertoire einer erfahrenen Softwareentwicklerin. Das Neue ist, daß dieser Erfahrungsschatz jetzt systematisch aufgearbeitet und beschrieben wird, so daß er einer breiteren Öffentlichkeit zugänglich wird. Insbesondere Neulinge in der Softwareentwicklung können davon profitieren.

Ein **Entwurfsmuster** beschreibt ein bestimmtes, in einem bestimmten Kontext immer wiederkehrendes Entwurfsproblem sowie ein generisches Schema zur Lösung dieses Problems. Das Lösungsschema wird durch seine Komponenten, den Beziehungen zwischen ihnen, ihre individuellen Verantwortlichkeiten sowie durch ihr Zusammenspiel spezifiziert (nach [1]).

Jedes Entwurfsmuster basiert auf langjähriger praktischer Erfahrung im Entwurf von Softwaresystemen.

Ein **Toolkit** ist eine Menge wiederverwendbarer Klassen, die allgemeine Funktionalität zur Verfügung stellt. Die Klassen eines Toolkits stehen in der Regel in enger Beziehung zueinander. Ein Toolkit gilt auch als das objektorientierte Äquivalent zur Unterprogramm-bibliothek. Ein Toolkit bedingt in der Regel keine festgelegte Architektur des Anwendungsprogramms.

Ein **Framework** (Gerüst) ist eine Menge von kooperierenden Klassen, die einen wiederverwendbaren Entwurf einer bestimmten Art von Software darstellen. Ein Framework wird durch anwendungsspezifische Unterklassen von Klassen des Frameworks zu einer bestimmten Anwendung getrimmt. Das Framework diktiert die Architektur der Anwendung. Es definiert die allgemeine Struktur, die Aufteilung in Klassen und Objekte, deren Hauptverantwortlichkeiten, Kommunikation und den Kontrollfluß.

Im Gegensatz zu Toolkits und Frameworks wird mit einem Entwurfsmuster keine Klassenbibliothek zur Verfügung gestellt. Entwurfsmuster können eher zum Entwurf und zur Implementierung von Toolkits und Frameworks verwendet werden.

Im folgenden werden zwei Entwurfsmuster vorgestellt und deren Implementierung in C++ am Beispiel diskutiert. Die Beschreibung der Entwurfsmuster ist an [2] angelehnt.

Observer

Oft taucht das Problem auf, daß, wenn sich der Zustand eines bestimmten Objekts ändert, eine Menge von davon abhängigen Objekten benachrichtigt werden soll.

Ein typisches Beispiel ist die visuelle Darstellung eines Objekts. Die Darstellung muß immer aktualisiert werden, wenn sich der Zustand des Objekts ändert. Insbesondere kann es mehrere Darstellungen eines Objekts geben.

Ein Beispiel sei eine Uhr für die aktuelle Tageszeit. Die Uhrzeit kann als digitale oder analoge Anzeige dargestellt werden.



In der Terminologie von Entwurfsmustern wird das Objekt, von dessen Zustand die anderen Objekte abhängen, **Subjekt** genannt. Die abhängigen Objekte werden **Observer** genannt.

Die Zusammenarbeit des Subjekts mit seinen Observern wird folgendermaßen festgelegt:

- Wenn sich der Zustand des Subjekts ändert, benachrichtigt es alle seine Observer.
- Wenn ein Observer von einer Änderung benachrichtigt wird, kann er Informationen vom Subjekt abrufen und agiert entsprechend seiner Aufgabe.

Wenn die oben genannte Uhr das Subjekt und die Analogdarstellung der Observer ist, benachrichtigt die Uhr bei jeder Zeitänderung die Analogdarstellung, die ihrerseits die aktuelle Uhrzeit von der Uhr abfragt und gegebenenfalls die neue Uhrzeit darstellt. Gegebenenfalls bedeutet hier, daß je nach Genauigkeit der Darstellung auf dem Bildschirm die neue Uhrzeit ein neues Bild ergibt oder nicht.

Das Entwurfsmuster, das insgesamt **Observer** genannt wird, beschreibt diese Abhängigkeiten. Ein Subjekt kann eine variable Anzahl Observer haben, die sich jeweils dem Subjekt bekannt machen müssen. Das Bekanntmachen wird auch **subscription** und das Benachrichtigen der Observer wird **publish** genannt, womit ein passendes Bild aus dem Verlagswesen benutzt wird.

Implementierung

Es gibt mehrere Möglichkeiten, die Lösung eines Problems nach diesem Entwurfsmuster zu implementieren. Die Implementierungen unterscheiden sich im wesentlichen durch die Variabilität der Beziehungen zwischen Subjekt und Observer. Das Schema einer typischen Implementierung ist in Abbildung 1 dargestellt.

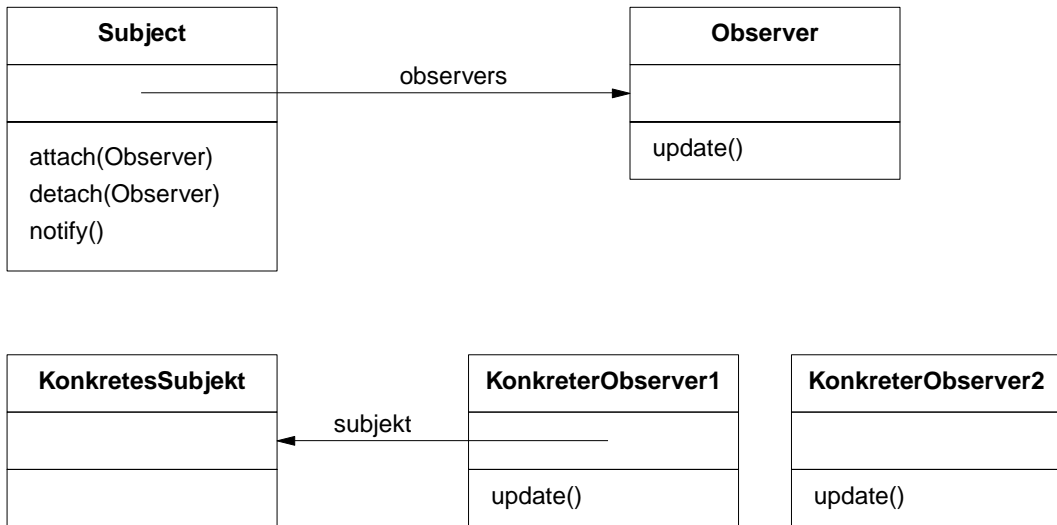


Abbildung 1: Struktur des Entwurfsmusters Observer

Die allgemeine Definition eines Subjekts wird durch die Klasse *Subject* und die eines Observers durch die Klasse *Observer* festgelegt. Ein Subjekt stellt die Operationen *attach* und *detach* zur Verfügung, mit denen sich ein Observer beim Subjekt an- und abmelden kann. Mit der Operation *notify()* benachrichtigt ein Subjekt dann alle seine Observer.

```

class Subject
{
public:
    virtual ~Subject();
    virtual
        void attach (Observer*);
    virtual
        void detach (Observer*);
    virtual void notify ();
protected:
    Subject();
private:
    typedef
        list<Observer *> _list_t;
    _list_t      _observers;
};

```

Um die Erzeugung von abstrakten Subjekten zu verhindern, wird der Konstruktor als *protected* festgelegt.

Ein Observer stellt zur Benachrichtigung bei einer Änderung des Subjekts die Funktion *update()* bereit:

```

class Observer
{
public:
    virtual ~Observer();
    virtual void update
        (Subject * subject) = 0;
protected:
    Observer();
};

```

Auch abstrakte Observer kann es nicht geben, da *update()* "pure virtual" definiert ist. Die Funktion *Subject::notify()* kann dann folgendermaßen implementiert werden:

```

void Subject::notify ()
{
    for ( _list_t::iterator

```

```

        i=_observers.begin();
        i!= _observers.end();
        i++ )
    (*i)->update(this);
}

```

Konkrete Subjekte und Beobachter

Ein konkretes Subjekt ist z.B. eine Uhr:

```

class Uhr : public Subject
{
public:
    Uhr( int sec = 0 );
    void tick();//Sekundentick
    void reset (int sec=0);
    int stunde();
    int minute();
    int sekunde();
private:
    time_t      _time; //Zeit
    // ...
};

```

Ändert sich die Uhrzeit, werden die Observer durch Aufruf der geerbten Funktion *Subject::notify()* benachrichtigt:

```

void Uhr::tick()
{
    // ...
    _time = time ( NULL );
    notify();
}

```

Der konkrete Observer *StreamUhr* gibt z.B. die Uhrzeit auf die Standardausgabe aus:

```

class
StreamUhr : public Observer
{
public:
    StreamUhr( Uhr * );
    virtual ~StreamUhr();
    virtual
        void update (Subject *);
private:
    Uhr *      _subject;
};

```

Da eine Darstellung einer Uhr ohne eine Uhr nicht leben kann, wird die Bindung an einer Uhr schon durch den Konstruktor vorgenommen:

```

StreamUhr::
StreamUhr ( Uhr * uhr )
    : _subject ( uhr )
{
    _subject->attach(this);
}

```

Entsprechend ruft der Destruktor die Funktion *detach(this)* auf.

Die Aktualisierung, die von der Uhr aufgerufen wird, ist bei der Standardausgabe relativ einfach:

```

void StreamUhr::
update ( Subject * changed)
{

```

```

if ( changed == _subject )
{ cout
  <<_subject->stunde()<<':'
  <<_subject->minute()<<':'
  <<_subject->sekunde()
  << endl;
}
}

```

Sie erzeugt Ausgaben wie:

```

19:33:59
19:34:60
19:34:1

```

Analoguhr

Eleganter ist eine analoge Darstellung.



Abbildung 2: Analoguhr

Auch eine Analoguhr ist ein Observer:

```

class
  AnalogUhr : public Observer
{
public:
  AnalogUhr
    (Uhr *, Punkt mitte =
      Punkt(100,100),
      Koordinate radius=75 );
  virtual ~AnalogUhr();
  virtual
    void update(Subject *);
private:
  Uhr * _subject;
  int _std;
      // angezeigte Stunde
  int _min;
      // angezeigte Minute
  Bild * _bild;

```

```

        // Bild fuer die
        // Darstellung
    // ...
};

```

Die Funktion *update()* erzeugt die Darstellung. Auch die Analoguhr registriert sich zunächst mit der Funktion *Subject::attach()* bei der Uhr und wird dann bei Zeitänderungen von der Uhr benachrichtigt. Eine Uhr kann viele Darstellungen gleichzeitig besitzen, so auch eine akustische Anzeige wie die Kuckucksuhr. Welche Darstellungen wirklich jeweils generiert werden, kann von der Anwendung und per Dialog vom Benutzer bestimmt werden.

Ein System, das nach dem Entwurfsmuster Observer implementiert worden ist, läßt sich relativ einfach um weitere konkrete Observer erweitern. Dabei wird das Subjekt in der Regel nicht verändert. Spätere Änderungen einer Anwendung beinhalten oft eine Änderung der Darstellung, gerade in Zeiten wechselnder Fenstersysteme, Benutzeroberflächen und Stilvorlieben.

Spannender als eine Uhr sind unterschiedliche Darstellungen z.B. bei einem Flugsimulator. Eine Sicht ist z.B. das Bild, das ein Pilot beim Blick nach vorn aus dem Cockpit hat. Eine andere Sicht ist die Instrumentenanzeige.

Ein Observer, der eine visuelle Darstellung eines Subjekts bewirkt, wird auch Sicht oder (englisch) View genannt.

Kompositionen

Fast jede interaktive Applikation steckt voller verschiedener Sichten eines oder mehrerer Objekte. Im folgenden wird die Darstellung einer Anwendung diskutiert, die eine einfache Bildrekonstruktion mit einem Hopfieldnetz beinhaltet. Ein Hopfieldnetz ist neuronales Netz, das sich insbesondere für diese Aufgabenstellung eignet. An verschiedenen Sichten bieten sich an:

- das Bild, das rekonstruiert werden soll,
- der Zustand des Netzes,
- Netzparameter wie Lernfaktor und
- gemessene Werte wie Anzahl Schaltvorgänge und bisher verbrauchte CPU-Zeit.

Jede dieser Sichten kann wieder aus Teilsichten aufgebaut sein. Der Zustand des Netzes besteht aus dem Zustand der Neuronen und den Werten der Gewichtematrix, die für sich jeweils eine Sicht darstellen. Es ist somit eine rekursive Struktur. Da die Gesamtsicht der Anwendung mit dem Entwurfsmuster Observer konzipiert werden sollte und das Hopfieldnetz, das Subjekt, in der Regel nur die Gesamtsicht bei Zustandsänderungen benachrichtigen sollte, ohne die rekursive Struktur zu kennen, bietet sich hier als weiteres Entwurfsmuster die Komposition an.

Entwurfsmuster Composite

Das Entwurfsmuster **Composite** (Komposition) dient der Zusammensetzung von Objekten in hierarchischen Strukturen. Ein Klient, Benutzer einer Komposition, benutzt ein Objekt ohne zu wissen, ob es ein elementares oder ein zusammengesetztes Objekt ist.

Das Entwurfsmuster beschreibt die Beziehungen und das Zusammenspiel zwischen den Objekten. Beteiligt sind

- die **Komponente (Component)**, die die Schnittstelle der Objekte definiert, die den elementaren und zusammengesetzten Objekten gemein ist,
- die **Blätter (leaf)**, die die Schnittstelle eines elementaren, also nicht zusammengesetzten Objekts festlegt,

- die **Komposition (Composite)**, die die Schnittstelle einer zusammengesetzten Komponente festlegt, und
- der **Klient (Client)**, der ein beliebiger Benutzer der Komponenten ist.

Der Klient benutzt die Komponenten nur über die Schnittstelle, die allgemein für alle Komponenten festgelegt ist. Wenn Komponente, Blatt und Komposition jeweils Klassen sind, dann sind Blatt und Komposition von der Komponente abgeleitet.

Die eigentliche Idee bei diesem Entwurfsmuster besteht in der rekursiven Struktur, die dadurch entsteht, daß eine Komposition wieder eine Menge von Komponenten enthalten kann. Die Komposition sorgt dafür, daß eine Operation, die von der Komponente zur Verfügung gestellt wird, sinngemäß für alle in ihr enthaltenen Komponenten ausgeführt wird.

Implementierung

Beim obigen Beispiel soll das Entwurfsmuster auf die Sicht des Hopfieldnetzes angewendet werden. Eine Sicht ist natürlich ein Observer. Damit eine Sicht auch aus mehreren Sichten zusammengesetzt werden kann, wird entsprechend dem Entwurfsmuster die Komponente als die Klasse *ComponentObserver* definiert:

```
class ComponentObserver
    : public Observer
{
public:
    // ...
    virtual void add
        (ComponentObserver*);
    virtual void remove
        (ComponentObserver*);
    virtual CompositeObserver*
        getComposite ()
            {return 0;}
};
```

Die Funktion *update()* wird den Klienten zur Verfügung gestellt, die von *Observer* geerbt wird. Die Funktionen *add()* und *remove()* dienen zum Zusammenstellen von Kompositionen. Die Funktion *getComposite()* dient zur Unterstützung des Traversierens von zusammengesetzten Komponenten. In der Klasse *ComponentObserver* wird die Funktion für Blätter voreingestellt.

Die Komposition wird durch die Klasse *CompositeObserver* festgelegt:

```
class CompositeObserver
    : public ComponentObserver
{
public:
    virtual void add
        (ComponentObserver*);
    virtual void remove
        (ComponentObserver*);
    // ...
    virtual ComponentObserver*
        GetComposite()
            { return this; }
    void update (Subject *);
private:
    typedef
        list<ComponentObserver *>
            _list_t;
    _list_t      _components;
public:
    typedef _list_t::iterator
        Iterator;
```

```
};
```

Die Funktionen *add()* und *remove()* werden auf entsprechende Operationen auf einer einfachen Liste abgebildet.

Die Funktion *update()*, die eine Aktualisierung der Sicht bewirken soll, muß in einer Komposition überdefiniert werden, damit die Aktualisierung automatisch eine Aktualisierung aller Teilsichten beinhaltet. Bei einer konkreten Komposition wird in der Funktion zusätzlich die Koordination der Sichten durchgeführt, z.B. eine relative Positionierung der Teilsichten in einem Fenster. Der Iterator dient dem Traversieren.

Ein Blatt ist jetzt eine konkrete, nicht zusammengesetzte Sicht, z.B. ein Beobachter des Zustands der Neuronen in der Klasse *STATEobserver*.

```
class STATEobserver
  : public ComponentObserver
{
public:
  STATEobserver(HOPNET * );
  virtual ~STATEobserver();
  virtual
    void update(Subject *);
protected:
  HOPNET * _subject;
  // darzustellendes Netz
  Bild * _bild;
  // Bild der Neuronen
  // ...
};
```

Ein Objekt der Klasse *STATEobserver* verhält sich im Prinzip wie ein Observer. Es muß die Funktion *update()* zur Verfügung stellen, die in diesem Beispiel den Zustand der Neuronen als eine Art Bitmap darstellt.

Der Klient ist in unserem Beispiel das Hopfieldnetz, das beobachtet wird.

In dem Beispiel werden die zwei Entwurfsmuster Observer und Composite zusammen angewendet. Jedes Entwurfsmuster löst ein anderes Problem. Das Entwurfsmuster Observer wird zur Festlegung der Beziehung zwischen Subjekt und Beobachter, hier einer Sicht, benutzt. Das Entwurfsmuster Composite wird benutzt, um das Problem der zusammengesetzten Sichten zu lösen.

Controller

Beide Entwurfsmuster erlauben eine benutzer- und programmgesteuerte Veränderung der Sichten. Damit der Benutzer die Sichten verändern kann, muß eine Sicht zusätzlich mit einem Controller versehen werden, der auf externe Ereignisse, z.B. einem Mausklick oder einem eingetippten Kommando, reagiert und die Ereignisse gegebenenfalls in Aktionen auf die Sicht oder auf das Subjekt abbildet.

Mit der Ergänzung um Controller ist die Anwendung nach der bekannten Standard-Architektur Model-View-Controller (MVC) entworfen, die in der Smalltalk-Gemeinde entwickelt wurde. Model ist in unserer Terminologie das Subjekt, und View steht für den Observer. Die Standard-Architektur kann mit den Entwurfsmustern Observer und Composite realisiert werden.

Zusammenfassung

Für Entwurfsmuster lassen sich in C++ relativ direkt Implementierungen angeben.

- Damit unterstützt C++ die Implementierung von Entwürfen, in denen Entwurfsmuster identifiziert worden sind.

- Die Vorteile, die Entwurfsmuster bieten, können durch eine Implementierung in C++ direkt genutzt werden.
- Unter anderem werden Programme dadurch verständlicher, da auf allgemein bekannte Lösungen zurückgegriffen werden kann.
- Insbesondere können dadurch Frameworks und Toolkits einfacher entwickelt und einfacher benutzt werden.

In der Literatur der letzten Jahre finden sich mehr und mehr Entwurfsmuster. Eine Vereinheitlichung der Terminologie bedarf wohl noch einiger Zeit, aber die Anfänge sind gemacht. So wie Containerklassen (z.B. Listen) zum Standard in der Programmierung geworden sind, werden in Zukunft hoffentlich auch Entwurfsmuster zum Standardrepertoire in der Softwareentwicklung gehören.

Quellen

Die vollständigen Programme befinden sich auf dem FTP-Server *ftp.informatik.uni-osnabrueck.de* unter *pub/hanser/um*.

Literatur

- [1] F. Buschmann "Was ist ein Entwurfsmuster?" *OBJEKTspektrum* Nr. 3, 1995, S. 82-84.
- [2] E. Gamma, R. Helm, R. Johnson, J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley, 1995.