Design und Implementation

Tilman Vierhuff

5. August 2004

Was euch heute erwartet

- Richtlinien zu gutem Programmierstil und Effizienz
- Entwurfsmuster

Implementationsrichtlinien

Packages

- Sammlung miteinander verwandter Klassen
- Gemeinsame Verwandtschaft (z. B. Unterklassen von `Component` in `java.awt`)
- Gemeinsame Kategorie (z. B. `Tokenizer`, `Iterator`, `Calendar` usw. in `java.util`)
- Modulartige Kapselung zusammenarbeitender Klassen
- Private Hilfsklassen mit eingeschränkter Sichtbarkeit

Klassen

- Beschreibung von Objekt-Bestandteilen und -Verhalten zur (evtl. indirekten) Erzeugung von Instanzen
- in Ausnahmefällen zur Sammlung statischer Methoden (z. B. `java.lang.Math`)

Interfaces

- Implementationsunabhängige Schnittstellen
- ermöglichen moderate Formen der Mehrfachvererbung
- auch als leere Marker-Interfaces (z. B. `java.lang.Cloneable`)

Vererbung

- Vererbung in Java bedeutet ist ein
- Negativbeispiele: `Vector` und `Stack`
- Anwendung für
  - Familien verwandter Objekte (`AVM`, `Atom`, `List`, `Pair`, `Coref`)
  - Konkrete Datentypen (`List`, `ArrayList` und `LinkedList`)
  - Spezialisierung mit zusätzlichen Eigenschaften (`Reader`, `FilterReader`, `PushBackReader` usw.)
  - Verbergen von Implementationsdetails (Plattformabhängigkeiten)
- Dasselbe gilt für `public`-Vererbung in `C++`

Eigenschaften guter Schnittstellen

- vollständig: Alle notwendigen Operationen vorhanden
- minimal: nicht durch andere Methoden ersetzbar (gelegentliche Ausnahme: unterschiedliche Parameter)
- allgemein: Keine überflüssigen Einschränkungen
- parametrisierbar: Wiederverwendbarkeit durch Anpassung
- unabhängig: möglichst wenig Vernetzung mit anderen Typen (außer vordefinierten)
- abgesichert: Nicht mehr Zugriff gestatten als unbedingt notwendig

Methoden

- Objektverhalten auf Anfrage von außen
- öffentliche Methoden erhalten und hinterlassen das Objekt immer in konsistentem Zustand
- keine Kettenaufrufe (`obj.init(); obj.calculate(); obj.finish();`); wenn mehrere Schritte notwendig sind, zusammen kapseln
- Sonderfälle Konstruktoren / Destruktor
- im allgemeinen Beschränkung auf eine Aufgabe (mehrfache Aufgaben in C++ zum Teil unsicher)

Methoden (II)

- Selectors: Ermittlung von Eigenschaften;
  Beispiel: `int size()`
- Modifiers: Veränderung; Beispiel: `void setElementAt()`
- Converters: Umwandlung; Beispiel: `String toString()`
- Iterators: Sonderfall, schrittweises Weiterrücken mit Rückgabe; Beispiel: `Object next()`
- Sonstige, z. B. `void printStackTrace(OutputStream)`
- Kombination von `getX()` und `setX()` im allgemeinen nicht sinnvoll (= `public`!)

Bezeichner

- Bezeichner drücken aus, was der Sinn eines Objekts / einer Methode ist (`setElementAt()`)
- Bezeichner drücken aus, ob Kosten mit einer Methode verbunden sind (`loadLexicon()`)
- Richtige Interpretationsebene nutzen (`getArrayList()` vs. `getColors()`)
- Geschmackssache: Typkennzeichnung (z. B. Zeiger mit `p`)
- in C++: Wer ist für dynamische Objekte zuständig? Namensgebung: `T * allocate()`, `adoptPointer(T *)`

Standardmethoden

- Von der Sprache vorgesehene Minimalschnittstelle
- In Java:

`String toString`	nützlich zum Debuggen
`boolean equals(Object)`	wenn flacher Vergleich zuwenig
`int hashCode()`	gleichzeitig mit `equals`
`Object clone()`	bei Bedarf
`void finalize()`	ausnahmsweise für Ressourcen

Generierte Methoden in C++

- vom Compiler automatisch definiert
- bei Member-Zeigern: selbst definieren oder durch `private` Deklaration verbieten

`Klasse()`	Defaultkonstruktor
`Klasse(const Klasse&)`	„Copy“-Konstruktor
`Klasse& operator=(const Klasse&)`	Zuweisung
`(virtual) ~ Klasse()`	(virtueller) Destruktor

Generierte Methoden in C++

- vom Compiler automatisch definiert
- bei Member-Zeigern: selbst definieren oder durch `private` Deklaration verbieten

`Klasse()`	Defaultkonstruktor
`Klasse(const Klasse&)`	„Copy“-Konstruktor
`Klasse& operator=(const Klasse&)`	Zuweisung
`(virtual) ~ Klasse()`	(virtueller) Destruktor

(für `operator&()` und `operator&() const` ist die Standarddefinitionen meist adäquat)

Effizienz

- Richtige Wahl der Algorithmen
- Verschachtelte Rekursion und Iteration vermeiden
- Schleifen und Rekursionen „sauber“ halten
- Mehrfachberechnungen vermeiden
- Kleine Verbesserungen:
  - `final` deklarieren
  - Einfache Datentypen verwenden
  - Iteratoren verwenden
  - Initialisierung statt Zuweisung im Konstruktor
  - In C++: Temporäre Variablen vermeiden

Dynamische Speicherverwaltung

- `new` sucht und verwaltet passenden Speicher
- Garbage Collector (bzw. `delete`) gliedert wieder ein
- überflüssige Reservierung und Freigabe vermeiden
- Objekte gleichen Wertes gemeinsam benutzen
- Objekte wiederverwenden und verändern
  statt neu anlegen und umkopieren
- Gilt auch für Stringkonkatenation und dynamische Arrays
- In C++ besonders kritisch: Explizite Speicherverwaltung

Typbestimmung zu Laufzeit

- `instanceof` und `java.lang.reflect` (in C++ `typeid` und `dynamic_cast<>()`) selten sinnvoll
  - nicht einfach erweiterbar
  - manuelle Wartung notwendig
  - kostenträchtig
- Eleganter: Wenn möglich, Vererbung und Methodenredefinition

In C++: `inline`

- Code wird eingesetzt statt Funktionen aufgerufen
- Nur für kleinste Methoden sinnvoll
- Codeaufblähung kann erheblichen Overhead verursachen (Caching, Page faults)
- insbesondere unsinnig bei Konstruktoren
- Methoden mit `static`-Variablen und `virtual`-Bindung können nie `inline` expandiert werden
- Klassenabhängigkeit und offene Implementationsdetails
- nur ein Vorschlag an den Compiler

Exceptions

- Abbruch und Sprung zum nächsten passenden `catch`
- Verlockend einfach zum Verlassen komplexer Berechnungen
- Exception zu verstehen als Ausnahmezustand, wenn es keinen anderen Ausweg gibt (z. B. Konstruktoren)
- Nach Möglichkeit Standard-Exception-Klassen benutzen
- Fangen von Exceptions in sicheren Systemen unbedingt erforderlich
- Kein `catch(Exception)` oder gar `catch(Throwable)`

Unsichere Methode in C++

```
class Stack{
private: T buf[100];
         int pos;
public:  Stack();
         void push(const T& t);
         const T pop();
}
```

Unsichere Methode in C++

```
class Stack{
private: T buf[100];
         int pos;
public:  Stack();
         void push(const T& t);
         const T pop();
}
Stack::Stack(): pos(0){}
```

Unsichere Methode in C++

```
class Stack{
private: T buf[100];
         int pos;
public:  Stack();
         void push(const T& t);
         const T pop();
}
void Stack::push(const T& t){
    buf[pos++] = t;
}
```

Unsichere Methode in C++

```
class Stack{
private: T buf[100];
         int pos;
public:  Stack();
         void push(const T& t);
         const T pop();
}
T Stack::pop(){    // ??
}
```

Unsichere Methode in C++

```
class Stack{
private: T buf[100];
         int pos;
public:  Stack();
         void push(const T& t);
         const T pop();
}
T Stack::pop(){
    return buf[--pos];
}
```

Unsichere Methode in C++

```
class Stack{
}
T Stack::pop(){
    return buf[--pos];
}

main(){
    Stack s;
    T t;
    s.push(t);
    t = s.pop();
}
```

Unsichere Methode in C++

```
class Stack{
}
T Stack::pop(){
    return buf[--pos];
}

main(){
    Stack s;
    T t;
    s.push(t);
    t = s.pop();    // Problem bei Exception in Zuweisung!!!
}
```

Single und Multiple Dispatching

- Aufruf von (virtuellen) Methoden in Java und C++ zur Laufzeit nur vom Typ des Empfängerobjekts abhängig
- Manchmal Abhängigkeit von mehr Typen erwünscht
- Lösung: Mehrfacher Methodenaufruf mit Auflösung eines Objekts pro Aufruf
  - \+ Übersichtlichkeit
  - \+ Vermeidung expliziter Typanfragen und -konvertierungen
  - – Bei zusätzlichen Typen Veränderung der Oberklasse
  - – Höherer Anfangsaufwand
  - ◦ Aufwand bei Aufruf nicht höher als für `switch`/Konversion

Single Dispatching in Java

```
abstract class AVM{
    abstract AVM unify(AVM);
}
```

Single Dispatching in Java

```
abstract class AVM{
    abstract AVM unify(AVM);
}
class Atom extends AVM{
    String value;
    AVM unify(AVM that){
        if(that instanceof Atom
            && ((Atom)that).value.equals(this.value))
            return this;
        return null;
    }
}
```

Single Dispatching in Java

```
abstract class AVM{
    abstract AVM unify(AVM);
}
class AttrPair extends AVM{
    String attribute;
    AVM value;
    AVM unify(AVM that){
        if(that instanceof AttrPair
            && ((AttrPair)that).attr.equals(attr))
            return value.unify(((AttrPair)that).value);
        return null;
    }
```

Double Dispatching in Java

```
abstract class AVM{
    abstract AVM unify(AVM);
    AVM unify(Atom){     return null; }
    AVM unify(AttrPair){ return null; }
    AVM unify(List){     return null; }
    AVM unify(Coref){    return null; }
}
```

Double Dispatching in Java

```
abstract class AVM{
    abstract AVM unify(AVM);
    AVM unify(Atom){     return null; }
    // ...
}
class Atom extends AVM{
    String value;
    AVM unify(AVM that){  return that.unify(this); }
    AVM unify(Atom that){
        return value.equals(that.value) ? this : null;
    }
}
```

Double Dispatching in Java

```
abstract class AVM{
    abstract AVM unify(AVM);
    AVM unify(AttrPair){ return null; }
    // ...
}
class AttrPair extends AVM{
    String attribute;
    AVM value;
    AVM unify(AVM that){      return that.unify(this);  }
    AVM unify(AttrPair that){
        return attr.equals(that.attr) ?
            value.unify(that.value) : null;
    }
```

Entwurfsmuster

Entwurfsmuster

- Grundlegende OOP-Designregeln:
  - Dinge $\longrightarrow$ Objekte
  - Aktionen $\longrightarrow$ Methoden
  - Gemeinsamkeiten $\longrightarrow$ Oberklassen
- In größeren Projekten außerdem wichtig: Interaktion zwischen verschiedenen Klassen und Objekten
- Entwurfsmuster: Typische Ansätze der Modellierung
- Erzeugungsmuster, Strukturmuster, Verhaltensmuster

Erzeugungsmuster

Prototypen (Exemplars)

- Objekterzeugung durch Objekt gleichen Typs
- In Java: Kopieren mit `Object clone()`
- In C++: „Copy-Konstruktor“ (durch statische Typisierung eingeschränkt)
- Beispiel: Kopieren von Lexikoneinträgen zur Weiterverarbeitung

Prototypen in Java

```
class Prototype implements Cloneable{
    public Object clone(){
        Object klon;
        try{
            klon = (Prototype) super.clone();
        }catch(CloneNotSupportedException e){
             // kann eigentlich nicht passieren
        }
        // hier klassentypische Details
        return klon;
    }
}
```

Singletons

- Klassen, von denen nur eine einzige Instanz existieren darf
- Beschränkter Zugang zur Konstruktion
- Beispiel: Eindeutige, zentrale Ressourcen, z. B. Datenbank

```
class Singleton{
     private static instance = new Singleton();
     private Singleton(){}
     public Singleton getInstance(){
         return instance;
     }
}
```

Singletons falsch in C++

```
class Singleton{
private:
    static Singleton instance;
    Singleton();
public:
    static Singleton& getInstance();
};
Singleton Singleton::instance;
Singleton::Singleton(){}
Singleton& Singleton::getInstance(){
    return instance;
}
```

Zeitpunkt der Initialisierung von `instance` undefiniert!

Singletons in C++

```
class Singleton{
private:
    Singleton();
public:
    static Singleton& getInstance();
};
Singleton::Singleton(){}
Singleton& Singleton::getInstance(){
    static Singleton instance;
    return instance;
}
```

Fabrikmethode (Virtueller Konstruktor)

- Erzeugung von Typen, die erst zur Laufzeit bekannt werden
- Erzeugte Objekte als Rückgabewerte von Methoden
- Erzeugungsmethode entscheidet über Wahl des Konstruktors
- Beschränkter Zugang zur Konstruktion
- Ergebnistyp ist Oberklasse oder -interface des tatsächlichen Typs
- Beispiel: Einlesen aus Dateien

Fabrikmethode in Java


```
public class Token{
    protected Token();
    Token next(PushbackReader r){
        char c = s.read();
        unread(c);
        if(c>='0' && c<='9') return new Number(r);
        else                 return new Name(r);
    }
}
public class Number extends Token{ ... }
public class Name   extends Token{ ... }
```

Fabrikklasse (Abstract Factory, Kit)

- Erzeugung von Objekten, deren Typ geheimgehalten wird
- Erzeugte Objekte als Rückgabewerte von Methoden
- Sammlung von Erzeugungsmethoden in einem Objekt
- Beschränkter Zugang zur Konstruktion
- Ergebnistyp ist Oberklasse oder -interface des tatsächlichen Typs
- Benutzercode ist unabhängig vom tatsächlichen Typ der Objekte
- Beispiel: Hardwareabhängige Spezialisierungen

Fabrikklasse in Java


```
class Factory{
    public Window newWindow(){
       return new LinuxWindow();
    }
    public Button newButton(){
       return new LinuxButton();
    }
}
```

Erbauer (Builder)

- Trennung von Repräsentation und Konstruktion
- Zur Kombination komplizierter Konstruktionsalgorithmen mit unterschiedlichen Repräsentationen
- Konstruktionsalgorithmus wird mit erwünschtem Repräsentations-Erbauer parametrisiert
- Beispiel: Compiler-Frontend mit unterschiedlichen Backends; Gleiche Erzeugung des Syntaxbaums mit variabler Codegenerierung

Strukturmuster

Brücke (Bridge, Handle/Body)

- Entkopplung von Abstraktion und Implementation
- Paar von Klassen für Interaktion mit Benutzer und systemabhängiges Verhalten
- Hilft, Neukompilierung zu vermeiden, wenn Schnittstelle gleich bleibt
- Implizite Fabrikmethode
- Beispiel: Plattformabhängige Grafik (z. B. `java.awt.peer`)

Brücke in Java

```
public class Handle{                    // plattformunabhaengig
    private Body body = new Body();
    public void tuWas(){
        body.tuWas();
    }
}

private class Body{                     // plattformabhaengig
    public void tuWas(){
      ...
    }
}
```

Proxy (Surrogat)

- Verändert den Zugriff auf ein Objekt
- Als Repräsentation ferner oder teurer Ressourcen
- Erlaubt Lazy Evaluation, Zugriffsschutz (Parallelverarbeitung, Firewall), Smart Pointer
- Keine veränderte Funktionalität

Proxy in Java

```
public class Proxy{
    private HugeStructure huge;
    public Object getDetail(){
        if(huge==null)  huge = new HugeStructure();
        return huge.getDetail();
    }
}
private class HugeStructure{
    Object[] contents = new Object[1000000]
    Object getDetail(){
        return contents[0];
    }
}
```

Kompositum

- Verwandtschaftsbeziehung von Einzelobjekten und Containerobjekten
- Gemeinsame abstrakte Oberklasse
- Operationen der Oberklasse
- Beispiel: Dateien und Verzeichnisse

Kompositum in Java


```
public abstract class AVM{
    abstract AVM unify(AVM that);
}
public class Atom extends AVM{
    private final String value;
    AVM unify(AVM that){ ... }
}
public class List extends AVM{
    private final ArrayList avms;
    AVM unify(AVM that){ ... }
}
```

Fliegengewicht (Flyweight)

- In großer Zahl gemeinsam benutzte Objekte mit wenigen Eigenschaften
- Beschränkter Zugang zur Konstruktion
- Verhindert Speicherverschwendung
- Verwaltungsaufwand beim „Erzeugen“ bzw. Aufsuchen der Objekte
- Interne Speicherung in Hashtabelle oder Suchbaum
- Beispiel: Atomare AVMs

Fliegengewicht

```
public class Atom extends AVM{
    private final String value;
    private static final Hashtable atoms = new Hashtable();
    private Atom(String value){
        this.value = value;
    }
    public Atom create(String value){
         Atom neu = (Atom) atoms.get(value);
         if(neu==null)
             atoms.put(value, neu = new Atom(value));
         return neu;
    }
}
```

Fassade

- Zentrale Schnittstellenklasse für Packages
- Erleichtert die Handhabung umfangreicher Utility-Sammlungen
- Verbirgt Implementationsdetails
- Vermindert Abhängigkeiten
- Ähnelt einer Brücke für ganze Package

Adapter (Wrapper)

- Anpassung der Schnittstellen vorhandener Klassen
- Keine eigene Funktionalität, nur Weiterleitung
- Nützlich bei Integration verschiedener Bibliotheken
- In `C++` typischer Kandidat für `inline`-Methoden

Decorator

- Dynamische Erweiterung von Objekten zur Laufzeit
- Vorgeschaltete Objekte schalten Zusätze ein, um und aus
- Decorator-Klassen von dekorierten Klassen abgeleitet
- Beispiele: Graphische Anzeigen, Streamklassen (`java.io`)

Verhaltensmuster

Beobachter (Observer)

- Objekte, die vom Zustand anderer Objekte abhängen
- Anmeldung im beobachteten Objekt
- Automatische Nachrichten bei relevanten Veränderungen
- Abhängigkeiten werden automatisiert
- Sinnvoll in großen, verteilten Anwendungen
- Beispiel: Mehrere Darstellungen derselben Daten

Observer in Java

```
public class Data extends java.util.Observable{
    public void commitChanges(){
        setChanged();
        notifyObservers();
    }
}
public class Viewer implements java.util.Observer{
    public Viewer(Data data){
        data.addObserver(this);
    }
    public void update(Observable o, Object arg){
        System.out.println(o.toString());
    }
```

Iterator

- Durchlaufen aller Elemente eines Containers
- Trennt Zugriff von der Datenstruktur
- Erlaubt beliebig viele gleichzeitige Durchläufe
- Einheitliche, containerunabhängige Schnittstelle

Iterator in Java

```
public class Container{
    Object [] inhalt;
    public Iterator iterator(){
        return new java.util.Iterator(){
            int wo;
            public boolean hasNext(){
                return wo<inhalt.length;
            }
            public Object next(){
                return inhalt[wo++];
            } };
    }
}
```

Benutzung von STL-Iteratoren in C++

```
#include<vector>
#include<iostream>
main(){
    std::vector<int> v;
    for(int i=0; i<10; ++i)
        v.push_back(i);
    std::vector<int>::iterator i=v.begin(), j=v.end();
    for(; i!=j; ++i)
        std::cout<<*i<<" ";
    std::cout<<std::endl;
}
```

STL-ähnlicher Iterator (unvollständig!)

```
class container{
private:  int inhalt[10];
public:   class iterator{
              private:  int *wo;
              public:   iterator(int *w){  wo = w;      }
                        int &operator*(){  return *wo; }
                        void operator++(){ ++wo;       }
                        iterator operator++(int){
                            return iterator(wo++);
                        }
          };
          iterator begin(){  return iterator(inhalt);     }
          iterator end(){    return iterator(inhalt+10);  }
```

Vermittler (Mediator)

- Zentralisierte Kommunikation zwischen großer Anzahl verschiedener Objekte
- Einheitliches Protokoll
- Mehrfache Abhängigkeiten vereinfacht auf mehrere Einzel-Beziehungen
- Beispiel: Zentrale Steuerung der Kaffemaschine

Schablone (Template Method)

- Algorithmus mit Spezialisierung durch Untermethoden
- Oberklasse definiert Rahmenhandlung mit abstrakten Lücken
- Unterklasse füllen abstrakte Lücken durch Überschreiben
- Typischerweise verwandte Probleme mit gleichem Verwaltungsaufwand

Schablone in Java

```
abstract class Find{
    int[] array;
    int find(){
        for(int i=0; i<array.length; ++i)
            if(pred(array[i])) return array[i];
        return 0;
    }
    abstract boolean pred(int);
}
```

Schablone in Java

```
abstract class Find{
    int[] array;
    int find(){
        for(int i=0; i<array.length; ++i)
            if(pred(array[i])) return array[i];
        return 0;
    }
    abstract boolean pred(int);
}
class FindFive extends Find{
    boolean pred(int i){ return i==5; }
}
```

Schablone in Java

```
abstract class Find{
    int[] array;
    int find(){
        for(int i=0; i<array.length; ++i)
            if(pred(array[i])) return array[i];
        return 0;
    }
    abstract boolean pred(int);
}
class FindEven extends Find{
    boolean pred(int i){ return i%2==0; }
}
```

Strategie (Policy)

- Familie von Algorithmen mit gleicher Schnittstelle
- Algorithmusobjekte als Parameter anderer Algorithmen verwendbar
- Algorithmen je nach Bedarf einfach austauschbar
- Nur für variierendes Verhalten bei gleicher Verwendung
- Beispiel: Zeilenumbruchstrategien, Registerbelegung in Compilern

Zustand (State)

- Eigenschaften eines Objekts als eigenes Subobjekt
- Zustand als Ganzes austauschbar und wiederherstellbar
- Beispiel: Dialogzustände, Adaption an bestimmte Benutzer

Zustand in Java

```
class UserModel{
    String name;
    Drink favourite;
}
class DialogueState{
    UserModel um;
    changeUser(UserModel um){
        this.um = um;
    }
}
```

Befehl (Command, Action)

- Repräsentation von Benutzerinteraktionen als Objekte
- Weitergabe von Befehlsobjekten zwischen Programmteilen
- Abspeichern von Befehlsobjekten in Datenstrukturen
- Implementation von Undo/Redo
- Protokollieren von Änderungen

Literatur

- **Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software** von Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

  Auch auf englisch erhältlich. Obwohl ich sonst eher zur englischen Originalliteratur rate, ist in diesem Fall die deutsche Übersetzung empfehlenswert, weil sie nicht nur gut gemacht wurde, sondern sowohl die englischen Originalbegriffe als auch ihre deutschen Pendants enthält

Literatur

- Für die, die in C++ programmieren:
  Effektiv C++ programmieren und **Mehr effektiv C++ programmieren** von Scott Meyers.

  Dringend anzuratende Richtlinien zu gutem Programmierstil in C++, darunter wirkliche Notwendigkeiten, potentielle Fehlerquellen, Tips zu Effizienz und Eleganz, alles sehr gut und ausführlich erklärt

Literatur

- Falls jemand von euch glaubt, sich mit C++ auszukennen: **Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions** von Herb Sutter.

  Inzwischen auch auf deutsch übersetzt, die Übersetzung kenne ich aber nicht. Knobelaufgaben mit ausführlich beschriebenen Lösungen, die praktisch jedem beweisen, dass C++ weit weniger trivial ist als die meisten denken. Wer es durchgearbeitet hat, kann sich auch am Folgeband versuchen.

Viel Erfolg!