KONZEPTE DER OO PROGRAMMIERUNG

1. *Mechanismen für die Software Wiederverwendung*

Objekt orientierte Technologien wurden von Anfang damit verkauft, dass wir damit endlich die Möglichkeit haben, Software in Form von Komponenten zu erstellen und wieder zu verwenden, analog zu den ICs. Diese Idee wurde speziell von Brad Cox in seinem Buch über Objective-C , der Basis eines bekannten Betriebssystems eines berühmten Computers (Next).

Zweifellos wurde die Software-Wiederverwendung durch OO Technologien stark verbreitet. Wir wollen in diesem (allgemein gehaltenen) Kapitel untersuchen, wie dies durch die Konzepte *Vererbung* und *Komposition* ermöglicht wird.

Vererbung wird in Java durch zwei Konzepte ermöglicht: Interfaces und Unterklassenbildung.

In diesem Kapitel möchten wir schlicht besser verstehen, wie diese Konzepte sich konkret zeigen, auswirken und wie sie sinnvoll genutzt werden können.

1.1. Substituierbarkeit

Substituierbarkeit ist eines der mächtigsten und wichtigsten Konzepte des Software Engineerings , der Software Entwicklung und der Objekt Orientierten Programmierung.

Substituierbarkeit : eine *Variable* wird von einem bestimmten Typus deklariert und speichert einen *Wert* von einem andern Typus.

Wie ist dies machbar? Wie sieht dies im Detail aus? Wie kann man das an einem einfachen Beispiel erklären und verstehen?

Substitution durch Vererbung mit Hilfe von Unterklassen: Vererbung von Verhalten
Betrachten wir das Spiel SOLITAIRE:

```
public class Solitaire {
        static public CardPile allPiles[  ];            // Deklaration
        public void init() {
                ...
                allPiles = new CardPile[13];            // Allokation
                ...
                allPiles[0] = new DeckPile(335,30); // Initialisierung
                allPiles[1] = new DiscardPile(268, 30);
        }
        ...
}
```

- **allPiles** wird vom Typus **CardPile** deklariert, kann also Werte vom Typ **CardPile** aufnehmen.
- der Variable wird aber ein Wert vom Typus **DeckPile** oder vom Typus **DiscardPile**, einer *Unterklasse* der Klasse **CardPile** zugewiesen.

Substitution durch Vererbung mit Hilfe von Interfaces: Vererbung von Spezifikation
Betrachten wir dazu das Kaonenspiel:

```
class CannonWorld extends Frame {
        ...
        private class FireButtonListener implements ActionListener {
                public void actionPerformed(ActionEvent e) {
                        ...
                }

                public CannonWorld() {
                        ...
                        fire.addActionListener(new FireButtonListener() );
                }
        }
}
```

- **FireButtonListener** implementiert **ActionListener**
- in der Methode **addActionListener** wird eigentlich eine Variable vom Typus **ActionListener** erwartet. Es wird aber statt dessen eine Variable des Untertyps **FireButtonListener** verwendet

1.1.1. Die Is-A und die Has-A Regel

Umgangsprachlich ist der Unterschied klar:

- IS-A : ein Mensch ist ein Lebewesen
  Damit hat der Mensch vieles mit andern Lebewesen gemeinsam.
  Der Mensch ist ein spezielles Lebewesen, ein Spezialfall.
- HAS-A : ein Mensch hat einen Intellekt (aber vermutlich nicht jedes Lebewesen)
  Darin unterscheidet sich der Mensch von andern Lebewesen.
  Intellekt und Lebewesen stehen aber nicht in einer IS-A Beziehung: der Intellekt ist kein Lebewesen.

In der Regel ist der Unterschied klar.

Machen Sie sich den Unterschied am Beispiel "Auto" noch einmal klar.

Wichtig für uns ist die Implikation für die Programmierung und speziell der Vererbung.
Versuchen Sie am Autobeispiel in Java die Unterschiede der IS-A und der HAS-A Regel (in Java) klar zu machen!

IS-A impliziert Vererbung (extends);
HAS-A impliziert Komposition.

1.1.2. Vererbung von Programmcode und Vererbung von Verhalten

Wir haben bereits gelernt, dass es zwei Typen von Vererbung in Java gibt:

- Vererbung von Verhalten und Programmcode mit Hilfe der Unterklassenbildung : **extends**
- Vererbung von Spezifikation mit Hilfe von Schnittstellen : **implements**

Wir haben beide Verfahren bereits mehrfach eingesetzt:

- Unterklassen

```
//      Cannon Game
public class CannonGame extends Frame {
        ...
}
```

- Interfaces

```
//      Cannon Game
class CannonWorld extends Frame {
        ...
        // Fire Button implementiert das Listener Interface
        private class FireButtonListener implements ActionListener {
                public void actionPerformed(ActionEvent e) {
                        ...
                }
        }
}
```

Unterklassen sollten immer dann eingesetzt werden, wenn dadurch Programmcode wieder verwendet werden kann.

Interfaces sollten immer dann eingesetzt werden, wenn die Unterklasse ein bestimmtes Verhalten zeigen soll, aber Programmcode noch nicht vorliegt.

1.2. Beschreibung von Komposition und Vererbung

Software kann auf zwei Arten **wieder verwendet** werden:

- durch **Komposition**
- durch **Vererbung**

Hier verwenden wir diese beiden Techniken, um die Unterschiede zwischen IS-A und HAS-A zu verdeutlichen.

Betrachten wir ein Beispiel, welches wir aus den Übungen bereits kennen: **Stack**s

Wir wollen Stacks mit Hilfe des Datentyps **Vector** implementieren.

Im wesentlichen sieht der abstrakte Datentyp wie folgt aus:

```
class Vector {
        public boolean isEmpty() {...}                  // ist ein Element vorhanden?
        public int size() {...}                         // wie viele Elemente sind vorhanden?
        public void addElement(Object value) {...}  // füge ein neues Element ein
        public Object lastElement() {...}               // letztes Element lesen
        public Object removeElement(int Index) {...}// lösche Element an der Stelle Index
        ...
}
```

Diese Version unterscheidet sich von der Standard Implementation in der Java Bibliothek, weil es hier nur um das Wesentliche geht! isEmpty heisst dort empty zum Beispiel.

Jetzt möchten wir Stacks auf zwei Arten implementieren, wie oben bereits erwähnt!

14.1.1. Die Verwendung von Komposition

Um unseren abstrakten Stack zu implementieren, verwenden wir als erstes die Komposition.
In diesem Falle besteht der Zustand der Objekte aus mindestens zwei Teilen:
da das Objekt eine Instanz einer zusammen gesetzten Klasse ist, verwenden wir (Teile) ein(es) bereits bestehenden Objektes, bezw. einer bestehenden Klasse.

Hier der Programmcode:

```
class Stack {
        private Vector dieDaten;

        public Stack() {
                dieDaten = new Vector();
        }

        public boolean isEmpty() {
                return dieDaten.isEmpty();
        }

        public Object push(Object neuerWert)
                return dieDaten.addElement(neuerWert);
        }

        public Object peek() {
                return dieDaten.lastElement();
        }

        public Object pop() {
                Object result = dieDaten.lastElement();
                dieDaten.removeElementAt(dieDaten.size() - 1);
                return result;
        }
}
```

Inwiefern verwenden wir hier Komposition?
Wir verwenden bereits bestehende Methoden, die für die Datenstruktur Vector definiert sind.
Wir können also auf bereits bestehenden Code zurück greifen.
Zusätzlich werden die Daten mit Hilfe einer private Datenstruktur die Daten gespeichert.

Allerdings:
die zwei Datenstrukturen Vector und Stack gemäss dieser Implementation unterscheiden sich grundsätzlich und können nicht mehr substituiert werden.

1.2.1. Die Verwendung von Vererbung

Jetzt implementieren wir die gleiche Datenstruktur, den Stack, mit Hilfe der Vererbung. Wie wohl? In diesem Beispiel ist dies offensichtlich der Vector, den wir bereits im vorigen Beispiel mit verwendet haben.

Hier die Implementation:

```
class Stack extends Vector {
        public void push(Object value) {
                addElement(value);
        }
        public Object peek() {
                return lastElement();
        }
        public Object pop() {
                Object result = lastElement();
                removeElementAt(dieDaten.size() - 1);
                return result;
        }
}
```

In dieser Implementation ist der Stack, wie in der Java Bibliothek, eine Unterklasse der Klasse Vector.
Vorteil:
verschiedene Methoden müssen nicht implementiert werden, da sie bereits in der Oberklasse existieren!

1.3. Gegenüberstellung von Komposition und Vererbung

Was sind nun Vorteile und Nachteile dieser beiden unterschiedlichen Implementationen?

- Vererbung beinhaltet implizit die Substituierbarkeit, da die Unterklasse natürlich eine Oberklasse besitzt, von der sie Methoden und Datenstrukturen erbt.
  Im Falle der Komposition besteht dieser Vorteil definitiv nicht!
- Komposition ist oft die einfachere der beiden Methoden. Bei ihr werden die vererbten Teile explizit sichtbar, es ist klar, welche Methoden für die neue Datenstruktur definiert sind und welche nicht.
- Bei der Vererbung muss der Programmierer erst mal prüfen, welche Methoden für die ursprüngliche Datenstruktur definiert wurden. Dies wird in unserem Beispiel deutlich, da wir verschiedene Methoden der Superklasse verwenden.
  Dieses Phänomen, dass man also immer zwischen der Oberklasse und der Unterklasse hin und her wandern muss, um das Programm zu verstehen, bezeichnet man auch als **"yo-yo" Problem**.
- Da bei Vererbung Code wieder verwendet wird, sind die Programme, die durch Vererbung entstehen, in der Regel kürzer als Programme, die durch Komposition entstehen. Unser obiges Beispiel zeigt dies bereits deutlich.
- Vererbung kann negative Seiteneffekte produzieren:
  wenn bestimmte Methoden in der Superklasse definiert wurden, die eigentlich in der Unterklasse besser nicht eingesetzt werden, besteht (ausser in der Oberklasse : durch Definition der Methoden als **final**) später keine Möglichkeit mehr, unsinnige Methoden weiter zu verwenden.

- die Implementation der Komposition ist zwangsweise flexibler als die der Vererbung: in unserem Stack Beispiel können wir auch eigentlich an Stelle des Vector eine andere Datenstruktur definieren.
- Implementation durch Vererbung erlaubt auch den Zugriff auf geschützte Methoden und Datenfelder (**protected :** Zugriff innerhalb des gleichen Packages ist möglich, falls eine direkte Verbindung besteht, bezw. die Klassen im selben Ast sind, wie wir bereits besprochen haben [Gosling]).
  Dies ist natürlich bei der Komposition nicht möglich (der Zugriff ist nur möglich sofern die Methoden oder Datenfelder **public** sind)!
- Die Wartung eines Programmes hängt von vielen Faktoren ab:
  kurze Programme lassen sich besser warten - was für die Vererbung spricht
  Programme, die kohärent sind, lassen sich besser warten - was für Komposition spricht
- Laufzeitverhalten:
  Komposition verursacht in der Regel langsamere Programme, da die Programmaufrufe komplex sein können.

Zusammenfassend :
ob Vererbung oder Komposition - beide Techniken haben Vorteile und Nachteile. Es muss also im konkreten Fall entschieden werden, welche Technik eingesetzt werden soll.

1.4. Kombination von Vererbung und Komposition

Java selber bietet eine Vielfalt von Beispielen, an denen man beide Techniken UND Kombinationen davon untersuchen kann:

- Input Output, ein Thema das wir noch besprechen werden, verwendet beide Techniken bis zur Verzweiflung des Programmierers!

Inp**utStream**	SubKlassen
⟶	ByteArrayInputStream
⟶	FileInputStream
⟶	PipedInputStream
⟶	SequenceInputStream
⟶	StringBufferInputStream

Das abstrakte Konzept sieht wie folgt aus:

- lesen besteht primär darin, Bytes zu lesen, eines nach dem andern : Stream; InputStream

Das konkrete Konzept sieht so aus:

- die Daten werden aus dem Speicher, einem File, ... gelesen:
  ByteArrayInputStream
  FileInputStream
- die Daten werden eventuell zur Performance Steigerung in Buffern zusammen gefasst
  StringBufferInputStream
- wir müssen eventuell Informationen über die gelesenen Daten sammeln, wie zum beispiel die Zeilennummer bei einer zeilenorientierten Datei:
  FilterInputStream
- ...

Wir haben also beliebig viel Freiraum, uns kreativ zu bewegen oder verloren zu gehen.

1.5. Neue Formen der Software Wiederverwendung

In diesem Kapitel besprechen wir die dynamische Komposition, eine Form der Komposition, die erst zur Laufzeit entschieden wird, im Gegensatz zur statischen Komposition.

Innere Klassen haben wir bereits kennen gelernt: es handelt sich um Klassen, die innerhalb einer andern Klasse definiert werden.

Als Beispiel verwenden wir das Verhalten eines Frosches, der sich bekanntlich aus der Kaulquappe entwickelt
(http://www.greenpeace.de/GP_DOK_3P/KIDS/UNTERH/KU9703C.HTM)

```
class Frosch {
        private FroschVerhalten Verhalten;

        public Frosch() {
                Verhalten = new KaulkuappeVerhalten(); // anderswo definiert
        }

        public wachse() {       // je nach Alter bezw. Alterskategorie sieht das anders aus
                if (Verhalten.erwachsen() )
                        Verhalten = new VerhaltenEinesErwachsenenFrosches();
                Verhalten.wachse();
                Verhalten.schwimme();
        }
}
```

Das Verhalten einer Kaulkuappe und eines Frosches sind recht verschieden. Hier simulieren wir dynamisches Verhalten durch die Abfrage nach dem Alter, bezw. der Alterskategorie.

14.1.1. Dynamische Komposition

Bei der Komposition, im Gegensatz zur Vererbung, kann man das Verhalten der objekte zu einem späteren Zeitpunkt festlegen, also *dynamisch.* Die Konstruktion solcher Klassen und Objekte wird daher auch als *dynamische Komposition* bezeichnet!

Beim Frosch beispiel haben wir mehrere Möglichkeiten:

- wir können das Verhalten des Frosches fix festlegen und statisch lassen
- wir können das verhalten des Frosches analog zum Beispiel oben, dynamisch ändern.

Schauen wir uns dazu ein erweitertes Beispiel des Frosches an:

```
abstract class FroschVerhalten {
        public boolean erwachsen() { return false;}
        public void wachse();
        public void schwimme();
}

class KaulkuappenVerhalten extends FroschVerhalten {
        private int Alter = 0;
        public boolean erwachsen() {if(++Alter>24) return true;}
        public void wachse() {...}
        public void schwimme() {...}
}

class VerhaltenEinesErwachsenenFrosches extends Verhalten {
        public void wachse() {...}
        public void swim() {...}
}
```

Das abstrakte Verhalten muss auf Grund des Attributes **abstract** konkret implementiert werden.

Das Verhalten des Frosches ändert sich mit seinem Alter, wie wir unschwer aus der Beschreibung erkennen können, es wird also *dynamisch.* Die Art und Weise, wie wir das erreichen, kann durchaus als Muster für ähnliche Fälle gelten!

Diese Beispiel kombiniert Vererbung mit Komposition, da einzelne Methoden explizit spezifiziert werden, andere aus der Oberklasse übernommen werden.

1.5.1. Vererbung bei Inneren Klassen

Als weiteres Beispiel für die Kombination von Vererbung und Komposition haben wir bereits beim Flipperspiel angetroffen:

```
public class PinBallGame extends Frame {
        ...
        private class MouseKeeper extends MouseAdapter {...}
        private class PinBallThread extends Thread {...}
}
```

Die Klassen **MouseKeeper** und **PinBallThread** werden beide als Unterklassen definiert, von unterschiedlichen Oberklassen.

Zusammen "konstruieren" sie dann die Klasse **PinBallGame**, die ihrerseits die Klasse **Frame** verfeinert.

Da die Klassen **MouseKeeper** und **PinBAllThread** innerhalb der Klasse **PinBallGame** definiert werden, handelt es sich dabei um **innere Klassen**, *inner classes*.

1.5.2. Namenlose Klassen

Wir treffen oft auf folgende Situation:

- wir benötigen eine Spezialisierung einer Klasse, eigentlich aber nicht im grösseren Rahmen, sondern nur ein einziges Mal:
- wir definieren die Subklasse
- wir instanzieren die Klasse

  und dann benötigen wir die Klasse eigentlich nicht mehr.

Frage: gibt es nicht eine bequemere Form für solche Einmalaktionen, eine Art Shortcuts?

Java kennt dafür den Klassendefinitions-Ausdruck, ein scheussliches Wort!
Wie sieht das konkret aus?

Betrachten wir ein Beispiel!
zuerst die normale Definition:

```
class CannonWorld extends Frame {
        ...
        private class FireButtonListener implements ActionListener {
                public void actionPerformed(ActionEvent e) {
                        ...
                }
        }
        public CannonWorld() {
                fire.addActionListener(new FireButtonListener() );
                ...
        }
}
```

Und jetzt das Ganze etwas verknorzter:

```
class CannonWorld extends Frame {
            ...
        public CannonWorld() {
            fire.addActionListener(new ActionListener() {       // hier folgt die Definition
                public void actionPerformed(ActionEvent e) {
                    ...
                }
              } );  // hier endet fire.addActionListener mit der runden Klammer
        ...
        }
}
```

Die Klasse, die gleich anschliessend an die geschweifte Klammer { hinter new ActionListener folgt, definiert eine Klasse, eine sogenannte *unnamed class*, weil wir ihr keinen expliziten Namen geben.

Der Vorteil ist zwar gering: wir brauchen der Klasse keinen neuen Namen zu geben. Im obigen Beispiel sparen wir den Klassennamen **FireButtonListener**, einer inneren Klasse, wie man aus dem ersten Codefragment erkennen kann.

Der Nachteil ist offensichtlich:
wer ist in der Lage solchen Code noch fehlerfrei zu lesen? Wo muss die runde Klammer hin?

1.6. Zusammenfassung

Die üblichsten Methoden der Software Wiederverwendung sind

- Vererbung und
- Komposition

Beide Techniken haben Vorteile und Nachteile. In der Regel braucht man bei komplexeren Systemen beide Techniken.

Die IS-A Technik zeigt einem mit Hilfe eines Sprachtestes, ob ein Konzept als Unterklasse eines andern definiert werden kann.

Die HAS-A Frage liefert einem Hinweise auf Komposition.

1.7. Selbststudium - Fragen

1 Erklären Sie in eigenen Worten den Begriff Substituierbarkeit!
2 Was versteht man unter der IS-A Regel? Was versteht man unter der HAS-A Regel? Wie hängen diese zwei Regeln mit Vererbung und Komposition zusammen?
3 Wie hängen Interface und Vererbung zusammen?
4 Welche Vorteile hat Komposition im Vergleich zur Vererbung?
5 Welche Vorteile hat Vererbung, die es in der komposition nicht gibt?
6 Inwiefern kombiniert ein **FilterStream** Vererbung und Komposition?

1.8. Übungen

1 Ein Set, eine Menge, ist eine ungeordnete Sammlung von Werten.
Beschreiben Sie, wie man mit Hilfe der Datenstruktur **Vector** Mengen implementieren könnte.
Würden Sie dafür eher Komposition oder eher Vererbung einsetzen?

2 Modifizieren Sie die abstrakte Definition des **Stacks,** die wir weiter vorne gegeben haben. Fügen Sie eine Ausnahme /Exception hinzu, die geworfen wird, wenn versucht wird, einen Wert aus einem leeren Stack zu lesen, oder ein Element aus einem leeren Stack zu löschen.

1.9. Studienhinweise

1.9.1. Lernziele

Nach dem Durcharbeiten dieses Kapitels kennen Sie

- den Unterschied zwischen IS-A und HAS-A und können diese zwei Regel gegeneinander abgrenzen
- die Mechanismen und die Unterschiede zwischen Vererbung und Komposition
- Vorteile und Nachteile der Vererbung und Komposition

1.9.2. Fragen

1 Was versteht man unter dem Prinzip der Substituierbarkeit?
2 Wie kann man feststellen, ob man eine IS-A oder eine HAS-A Beziehung vor sich hat?
3 Geben Sie drei Beispiel aus dem Alltag für IS-A Beziehungen;
Geben Sie drei Beispiele aus dem Alltag für HAS-A Beziehungen.
4 Erläutern Sie in eigenen Worten, was man unter der Technik der Software Komposition versteht.
5 Erläutern Sie in eigenen Worten, was man unter der Technik der Vererbung für die Software Wiederverwendung versteht.
6 Was ist der Unterschied zwischen **private** und **public in Bezug auf die Vererbung?**
7 Nennen Sie einige Nachteile der Komposition gegenüber der Vererbung und umgekehrt.
8 Was versteht man unter dem yo-yo Problem?
9 Was verhindert die Verbreitung von Software in Form von Komponenten , ähnlich wie die Chips in der Hardware?

KONZEPTE DER OO PROGRAMMIERUNG