KONZEPTE DER OO PROGRAMMIERUNG

Polymorphismus verstehen

Polymorphismus

Der Begriff *polymorphisch* stammt aus dem Griechischen und besagt im Wesentlichen: "mehrere Formen" (*poly* = mehrere, *morphos* = Form). Morphos hängt auch mit dem griechischen Gott Morphus zusammen. Dieser konnte Schlafenden, im Traum also, in unterschiedlichen Formen erscheinen; er war also echt polymorph. Auch der Homo Sapiens ist eigentlich polymorph: es gibt Menschen unterschiedlicher Farben und Formen (Zwergstämme, Bantus, ... ,Luzerner). In der Chemie kann eine polymorphe Substanz in mindestens zwei unterschiedlichen Arten kristallisieren: Carbon ist ein Beispiel dafür - in den Formen Graphite oder Diamant.

1.1. Variationen des Polymorphismus

Bei den Objekt Orientierten Programmiersprachen ist der Polymorphismus eine natürliche Konsequenz der *IS-A* Relation und der Mechanismen des Message Passings, der Vererbung und der Substitution. Der Vorteil der Objekt Orientierten Programmierung besteht darin, dass diese Konzepte flexibel kombiniert werden können und dadurch Programmcode mehrfach verwendet und unterschiedlich kombiniert werden kann.

Purer Polymorphismus beschreibt Funktionen, deren Argumente unterschiedliche Datentypen sein können. Im Polymorphismus reinster Art existiert eine Funktion, eine Methode, die den Programmcode-Rumpf darstellt, und die unterschiedlich interpretiert werden kann.

Das andere Extrem tritt dann ein, wenn mehrere Funktionen vorhanden sind (unterschiedliche Code Bodies), alle mit dem selben Namen - auch als *Overloading* oder *ad hoc Polymorphismus* bezeichnet.

Zwischen diesen zwei Extremen gibt es die *overriding* und *deferred Methoden.*

Wir werden die einzelnen Begriffe genauer erläutern und mit Beispielen hoffentlich klären.

1.2. Polymorphe Variablen

Polymorphismus wird, mit Ausnahme des Overloading / Überladens, mit Hilfe von *polymorphen Variablen* ermöglicht, zusammen mit der Idee der Substituierbarkeit.

Eine polymorphe Variable hat viele Gesichter; sie kann Werte von unterschiedlichem Typ aufnehmen. Polymorphe Variablen verwenden das Konzept der Substituierbarkeit. Das heisst konkret: obschon eine Variable einen Wert eines bestimmten Typs erwartet, kann dieser jederzeit durch einen Subtyp ersetzt werden.

In dynamischen Sprachen wie Smalltalk sind alle Variablen potentiell polymorph - jede Variable kann Werte eines beliebigen Typs aufnehmen, ein Konzept, welche die Datenprüfung sicher nicht erleichtert und die Zuverlässigkeit der Programme sicher nicht erhöht.

In Smalltalk beschreibt man den Typ mit Hilfe der zugeordneten Methoden. Der Benutzer kann also seine eigenen Datenstrukturen definieren, Datenstrukturen im Sinne Abstrakter Datentypen.

In Java und ähnlichen Programmiersprachen ist die Situation etwas komplexer. Polymorphismus geschieht in Java in Form eines Unterschiedes zwischen dem deklarierten (statischen) Datentyp und dem aktuell (dynamisch) angetroffenen Datentyp.

Ein typisches Beispiel für eine polymorphe Variable ist das Datenfeld **allPiles** des Solitaire Spieles:

- ursprünglich wurde das Array so deklariert, dass es Elemente vom Datentyp **CardPile** aufnehmen kann.
- in Wirklichkeit werden aber in **allPiles** beliebige Kartenstapel gespeichert, also auch Elemente der Subklassen
- die Methode **display()** zeigt den Kartenstapel an, unabhängig davon, ob es sich um die Grundklasse (statisch oder die zur Laufzeit (dynamisch) aktuelle (Sub-)Klasse ist.

```
public class Solitaire extends Applet {        // wäre doch mal was!
        ...
        static CardPile allPiles[ ];     // deklariert ein Array
        ...
        public void paint(Graphics g) {
                for (int i=0; i<13; i++) allPiles[i].display(g); // was auch immer g ist
        }
        ...
}
```

1.3. Overloading / Überladen

Eine Methode wird *überladen* wenn es zwei Funktions- / Methoden-Rümpfe gibt, welche mit dieser Methode zusammenhängt.

Überladen und *Überschreiben* sind zwei verwandte Konzepte, Überladen ist das allgemeinere der zwei Konzepte. Überschreiben ist ohne Überladen nicht möglich.

Beim Überladen ist der *Name* der Methode der polymorphe Teil - die Methode zeigt unterschiedliches Verhalten.

Anders ausgedrückt: es gibt eine abstrakte Methode, welche verschiedene Argumente aufnehmen kann. Der aktuell ausgeführte Programmcode hängt von den Argumenten ab. Der Compiler ist in der Regel in der Lage, zur Übersetzungszeit bereits zu bestimmen, welche Variante benutzt werden muss. Dadurch ist auch noch eine Codeoptimierung möglich.

1.3.1. Ein Beispiel aus dem Alltag

Sie kennen das Standardbeispiel aus dem Alltag:

- Bibliothekar(in) oder
- Floristin

Wir kennen keine Details bezüglich des Vorgehens der Personen. Wir wissen aber, dass wir oder die/der Bibliothekar(in) unterschiedlich reagieren, wenn wir einen Artikel beschaffen möchten. Wir reagieren also unterschiedlich, obschon die "Methode" dieselbe ist : "Artikel beschaffen".

1.3.2. Überladen und Coercion

Betrachten wir als Beispiel die Entwicklung von Programm-Bibliotheken, welche gemeinsame Klassen umfassen, um komplexe Datenstrukturen zu behandeln.

Zum Beispiel könnte die Klasse Datenstrukturen, wie Sets / Mengen, Arrays, Warteschlangen, ... beschreiben. Diverse dieser Klassen könnten ein und die selbe Methode **add** verwenden, oder benötigen, zum Beispiel, um ein neues Element einer Menge hinzu zu fügen.

Wir haben also folgende Situation:

- wir haben sematisch unterschiedliche Funktionen
- die eine ähnliche Funktion bewerkstelligen

Als Beispiel betrachten wir die Funktion "+". Die Addition zweier Integer Zahlen läuft wesentlich anders ab, als die Addition zweier Gleitkomma-Zahlen. Die Operation "+" wird also "überladen", da sie zwei unterschiedliche Operationen ausführen muss, obschon für uns die Addition einer Integer- oder einer Gleitkomma-Zahl kein wesentlicher Unterschied ist!

Ein ähnliches Konzept ergibt sich bei der sogenannten Koerxion / Coercion:

- ein Wert eines Typs wird in einen Wert eines andern Typs umgewandelt

Bei der Arithmetik gemischter Datentypen kann folgende Situation eintreten:

- es kann unterschiedliche Funktionen geben : "+" für Integer; "+" für Gleitkomma,..., "+" für Integer + Gleitkomma, "+" für Gleitkomma + Integer
- es kann zwei unterschiedliche Funktionen geben:
  "+" für Integer + Integer; "+" für Gleitkomma + Gleitkomma
  Im Falle von Gleitkomma + Integer wird Coercion angewandt, Datenumwandlung der Integer Zahl in eine Gleitkomma Zahl und anschliessende Addition.
- es gibt nur eine "+" Operation:
  alle Argumente werden umgewandelt in Gleitkomma-Zahlen (Coercion).

Welche Form konkret angewandt wird, hängt nicht zuletzt vom Compiler ab.

1.3.3. Überladen von mehreren separierten Klassen

Man kann zwei Arten des "Überladens" unterscheiden:

- der selbe Funktionsname wird in zwei oder mehreren Klassen verwendet, die mit einander nicht mittels Vererbung verbunden sind.
- die zweite Form tritt dann auf, wenn mehrere Funktionen mit dem selben Namen in der selben Klasse vorhanden sind

Wir haben für beide Situationen bereits Beispiele angetroffen:

- **isEmpty** ist ein Beispiel für die erste Art des Überladens (für **Vector, Hashtable,**...)
- **print** ist ein Beispiel für das Überladen, weil wir mit print unterschiedliche Datentypen drucken können.

Überladen / Overloading impliziert NICHT Änlichkeit

Uns ist verschiedentlich die Funktion **draw** begegnet, die zwar immer den selben Namen hat, aber völlig unterschiedliche Reaktionen zur Folge hat.

draw hat als Parameter ein graphisches Objekt und von daher kann die Aktion völlig unterschiedlich ausfallen.

Dieses "Überladen" ist auch nicht Ausdruck eines schlechten Stiles. In der Tat treten verschiedene Funktionen wie **add(), delete(), draw(), ...** immer wieder auf, aber in sehr unterschiedlichen Ausprägungen.

1.3.4. Parametrisches Überladen

Eine andere Form des Überladens tritt bei Methoden auf, die alle den selben Namen haben, aber unterschiedliche und eine unterschiedliche Anzahl Parameter haben. Diese Form des Overloadings bezeichnet man als *parametrisches Überladen.*

Parametrisches Überladen trifft man vor allem bei den Konstruktoren.

Als Beispiel können wir wieder eine geometrische Figur, wie das Rechteck nehmen:

- wir können es mit Hilfe eines Ursprunges und den Seitenlängen spezifizieren:
  {(0,0), a, b}
- wir können aber auch den nordwestlichen und den südöstlichen Punkt des Rechteckes angeben:
  {(0,0), (15,5)}
- weitere Varianten sind sicher denkbar

Somit können wir auch für den Konstruktor eine entsprechende Vielzahl von Varianten angeben.

1.4. Overriding / Überschreiben von Methoden : Wiederholung

Wir haben die klassische Methode des Überschreibens von Methoden bereits besprochen. Hier möchten wir nur auf einige spezielle Aspekte eingehen.

In einer (typischerweise abstrakten) Klasse wird eine generelle Methode deklariert. Diese Definition beschreibt in der Regel nur das allgemein gültige Verhalten der Klasse, die Struktur der Methode.

In einer Unterklasse wird diese Methode verfeinert. Wir sagen auch, diese Methode *überschreibe* die erste Methode.

Natürlich haben wir dann das yoyo-Problem, da wir die übergeordnete Definituion auch verstehen müssen.

16.1.1. Ersetzen und Verfeinern

Wir haben bereits unterschiedliche Formen des Überschreibens kennen gelernt:

- Überschreiben durch *Ersetzen*
  In diesem Falle wird der ursprüngliche Programmcode nicht ausgeführt.
- Überschreiben durch *Verfeinerung*
  In diesem Falle wird der ursprüngliche Programmcode plus zusätzlicher Code ausgeführt.

In der Regel wendet man die "Ersetzen" Sematik an dh. man überschreibt das Verhalten.

Beim Verfeinern wird man mit der Pseudovariablen **super** arbeiten. Im Solitaire Programm haben wir verschiedene Anwendungsfälle dieser Semantik:

```
class DiscardPile extends CardPile {
        public void addCard(Card aCard) {
                if (! aCard.faceUp() )
                        aCard.flip();
                super.addCard(aCard);
        }
}
```

Im Falle der *Konstruktoren* wird *immer* die *Verfeinerungs-Semantik* angewandt:

- ein Konstruktor einer Subklasse führt immer *erst* den Konstruktor der Oberklasse aus
- erst *anschliessend* wir der Zusatzcode des spezifischen Klassen-Konstruktors ausgeführt.
- falls der Konstruktor der Oberklasse Argumente benötigt, dann werden diese der Pseudovariablen **super** als Argumente mitgegeben.

Beispiel:

```
class DeckPile extends CardPile {
        DeckPile(int x, int y) {
                // zuerst den Konstruktor der Oberklasse ausführen
                super(x,y);
                // dann den neuen Kartenstapel kreieren
                for (int i=0; i<52; i++)
                        for (int j=0; j<=12;j++)
                                addCard(new Card(i,j) );
                // jetzt müssen wir die Karten mischeln
                Random generator = new Random();
                for (int i=0; i<52; i++) {
                        int j=Math.abs(generator.nextInt() ) % 52;
                        // vertauschen der zwei Karten
                        Object temp = the Pile.elementAt(i);
                        thePile.setElementAt(thePile.elementAt(j), i);
                        thePile.setElementAt(temp,j);
                }
        }
}
```

Wenn immer wir einen Konstruktor in dieser Art und Weise definieren, muss das **super** Statement die erste ausführbare Anweisung sein.

Falls die **super** Anweisung fehlt, dann wird der *Default-Konstruktor* der Oberklasse aufgerufen.

1.5. Abstrakte Methoden

Methoden, die als *abstract* deklariert werden, kann man auch als *deferred* Methoden, als Methoden auffassen, die *später* definiert werden. Solche Klassen werden in der Oberklasse spezifiziert, in der Unterklasse realisiert.

Interfaces sind ein Spezialfall dieses Mechanismus.

Bei abstrakten Methoden und Interfaces werden die Methoden in der Implementationsklasse überschrieben. Im Falle der abstrakten Klassen gibt es nichts ausser dem WAS zu vererben. Das WIE muss voll in der Unterklasse angegeben werden.

Abstrakte Methoden (und Klassen) haben erlaubt eine abstrakte Spezifikation.

Betrachten wir zwei typische Fälle:

1. wir haben geometrische Figuren und definieren **Shape** als Abstraktion, **Kreis, Rechteck** als konkrete Formen.
   Wenn wir jetzt die Methode **draw** definieren, so kann dies auf unterschiedlichen Ebenen geschehen: auf der Stufe von **Shape** und auf der Stufe der konkreten Formen.
   draw auf Stufe **Shape** ist kaum sinnvoll definierbar.
   Abstrakt macht es aber Sinn, in der Klasse **Shape** anzugeben, dass es eine entsprechende Methode gibt.
2. in vielen Sprachen, Java zum Beispiel, kann der Compiler den Code signifikant optimieren, wenn zur Compilezeit bereits feststeht, dass in der Unterklasse eine Methode definiert wird. Es bietet sich in diesem Falles also an, die Methode abstrakt möglichst früh zu definieren.

1.6. Purer Polymorphismus

In der Literatur wird der Begriff *Polymorphismus* unterschiedlich definiert. Eine übliche Definition (auch oft als reiner / purer Polymorphismus bezeichnet) versteht darunter Situationen, bei denen die Methoden mit einer unterschiedlichen Anzahl Parameter (und unterschiedlichem Datentyp) aufgerufen werden.

Überladen wird eine Methode dann genannt, wenn es unterschiedliche Methoden gibt, die alle gleich heissen und an unterschiedlichen Stellen im Programm definiert werden.

Die Grenze zu ziehen zwischen überladen und polymorph ist nicht immer eindeutig.

Beispiel:

```
public abstract class Number {
        public abstract int intValue();
        public abstract long longValue();
        public abstract float floatValue();
        public abstract double doubleValue();
        public byte byteValue() {
                return (byte)intValue();
        }
        public short shortValue() {
                return (short)intValue();
        }
}
```

Hier haben wir auf der einen Seite die Methode **intValue** , welche abstrakt definiert wird, also später (deferred) implementiert werden muss. Jeder Datentyp muss seine eigene Implementation dieser Methode liefern.

Die Methode **byteValue** wird nicht überschrieben, ist aber rein *polymorph: eine Methode, aber unterschiedliche Argumente (unterschiedliche Argumenttypen).*

1.7. Effizienz und Polymorphismus

In der Regel muss man beim Programmieren Kompromisse machen. Im Falle des Polymorphismus müssen wir uns bewusst sein, dass wir folgende Faktoren haben:

- Einfachheit der Entwicklung
- Einfachheit der Verwendung
- Lesbarkeit
- Effizienz

und diese nicht alle gleichzeitig optimieren können. Wir sind also gezwungen Kompromisse zu machen.

Im Falle der Methode **byteValue** kennt die Methode den Datentyp des Argumentes nicht, nicht endgültig. Deswegen ist eine Implementierung auch kaum auf effiziente Art und Weise möglich.

Wir müssen also *in der Regel* mit Effizienzproblemen rechnen. Die Vorteile der Objektorientierung überwiegen jedoch in der Regel bei weitem, ausser bei Echtzeitanwendungen.

1.8. Zusammenfassung

Polymorphismus wird für unterschiedliche Konzepte verwendet. Die wichtigste Anwendung sind die polymorphen Variablen, also einer Variable, die unterschiedliche Datentypen aufnehmen kann.

Überladen, overloading passiert dann, wenn zwei oder mehr Funktionen den selben Namen haben.

Falls diese Methoden zu Oberklasse und Unterklasse gehören, dann spricht man von *überschreiben, overriding.*

Die Verwendung von Polymorphismus optimiert tendenziell die Programm-Emtwicklungszeit und die Programm Zuverlässigkeit, auf Kosten der Laufzeit-Effizienz.

In der Regel überwiegen die Vorteile die Nachteile bei weitem.

1.9. Fragen

Die folgenden Fragen sind Selbsttestaufgaben, ohne Musterlösung!

1 Erläutern Sie den Begriff "Polymorphismus" im üblichen Sprachgebrauch.
2 Wann ist eine Variable polymorph?
3 Erläutern Sie das Konzept : "überschreiben" einer Methode.
4 Was heisst es, wenn man sagt, dass ein Wert einem andern Datentyp coerciert wurde?
5 Was versteht man unter parametrischem Überladen?
6 Was versteht man unter "überschreiben" und inwiefern unterscheidet sich "überschreiben" von "überladen"?
7 Welches ist die Standard Semantik für das Überschreiben von Methoden? und Konstruktoren?
8 Was versteht man unter einer "abstrakten Methode"?
9 Kennzeichnen Sie "puren Polymorphismus".
10 Warum sollte sich der Programmierer nicht zu viele Sorgen wegen schlechterer Effizienz polymorpher Programmtechniken machen?

1.10. Übungen

1 Beschreiben Sie die unterschiedlichen Formen des Polymorphismus im Programmbeispiel **PinBallGame**

2 Beschreiben Sie die unterschiedlichen Formen des Polymorphismus im Programmbeispiel **Solitaire**

1.11. Studienhinweise

1.11.1. Lernziele

Nach dem Lesen dieses Kapitels sollten Sie in der Lage sein:

- unterschiedliche Formen des Polymorphismus, die man in den objekt orientierten Programmiersprachen kennt, zu beschreiben
- unterscheiden zu können, zwischen
  überladen
  überschreiben und dem
  späteren (deferred) Implementieren von Methoden
- diese Konzepte an Hand von Programmbeispielen zu erläutern

1.11.2. Selbstest-Fragen

1. Was versteht man unter einem polymorphen Konzept?
2. Warum lassen sich Funktionen mit polymorphen Variablen einfacher in dynamischen als in Sprachen wie Java zu implementieren.
3. Warum ist es schwierig allgemein gültige Methoden in Sprachen wie PASCAL zu definieren (zum Beispiel eine Methode zur Bestimmung der Anzahl Elemente in einer Liste)
4. Was versteht man unter polymorphen Variablen?
5. Erläutern Sie den Unterschied zwischen *Überladen* und *Coersion.*
6. Definieren Sie drei unterschiedliche Methoden, mit deren Hilfe man zwei Zahlen zusammen zählen kann.
7. Warum ist es für einen Programmierer einfacher, Funktionsnamen zu überladen, im Gegensatz zur Verwendung eindeutiger Methodennamen.
8. Was ist der Unterschied zwischen parametrischem Überladen und dem Überladen von Methoden in unterschiedlichen Klassen
9. Erläutern Sie den Unterschied zwischen Überladen und Überschreiben.
10. Was versteht man unter einer "deferred" Methode?
11. Was versteht man unter "purem" Polymorphismus?
12. Warum sind polymorphe Prozeduren in der Regel weniger effizient als nicht-polymorphe Varianten?
    Warum sollte dieser Effizienzverlust den Programmierer nicht stören?

KONZEPTE DER OO PROGRAMMIERUNG