1 OBJEKTORIENTIERTE PROGRAMMIERUNG

1.1 Einleitung

Die Basis der objektorientierte Programmierung liegt bei drei Grundelemente: Unterstützung vomn Vererbungsmechanismen, Unterstützung von Datenkapselung und Unterstützung von Polymorphie.

Die Unified Modelling Language (UML) ist ein Modellierungsmittel. Die objektorientierte Analyse betrachtet eine Domäne als System von kooperierenden Objekten. Entwurfsmuster sind Elemente wiederverwendbarer objektorientierter Software und dienen als objektorientierte Methoden.

Jede Sprache hat ihre eigene Stärken, jede macht bestimmte Sachen einfach, und jede hat ihre eigenen Idiome und Muster. Je mehr Programmiersprachen man kennt, desto mehr Vorgehensweisen lernt man oft.

Korrektheit: Software soll genau das tun, was von ihr erwartet wird. Benutzerfreundlichkeit: Software soll einfach und intuitiv zu benutzen sein. Effizient: Software soll mit wenigen Ressourcen auskommen und gute Antwortzeiten für Anwender haben. Wartbarkeit: Software soll mit wenig Aufwand erweiterbar und änderbar sein.

1.2 Basis oder Objektorientierung

Die **Datenkapselung**, die **Polymorphie** und die **Vererbung** sind drei Werkzeugen, die einem erlaubt, die Zielsetzungen der Entwicklung von Software anzugehen. Der Speicher enthält Daten, die bearbeitet werden, andererseits enthält er Instruktionen, die bestimmen, was das Programm macht.

Routinen sind das Basiskonstrukt der strukturierten Programmierung. Indem ein Programm in Unterprogramme zerlegt wird, erhält es seine grundsätzliche Struktur. Eine Routine kann entweder Parameter haben oder ein Ergebnis zurückgeben. Eine Routine wird auch als Unterprogramm bezeichnet.

Eine **Funktion** ist eine Routine, die einen speziellen Wert zurückliefert, ihren Rückgabewert.

Eine **Prozedur** ist eine Routine, die keinen Rückgabewert hat. Eine Übergabe von Ergebnissen an einen Aufrufer kann trotzdem über die Werte der Parameter erfolgen.

Die **Strukturierung** der Instruktionen und der Daten erfolgen durch Verzweigungen, Zyklen und Routinen. Man benutzt globale und lokale, statisch und dynamisch allozierte Variablen, deren Inhalte definierte Strukturen wie Datentypen, Zeiger, Records, Arrays, Listen, Bäume oder Mengen haben. Das Strukturieren ist die Beherrschung der Komplexität.

Daten und Routinen sind voneinander getrennt. Die Objektorientierung verändert diese zwei durch die Einführung von Objekten. Daten gehören nun explizit einem Objekt. Objekte können ihre Daten verändern oder auch lesend auf sie zuzugreifen. Ein Objekt kann selbst dafür sorgen, dass die Konsistenz der Daten gewahrt bleibt.

Das Prinzip der **Datenkapselung** stellt die Konsistenz von Daten sicher und somit ist die Korrektheit gewährleistet. Vorgehensweisen und interne Daten können reduziert werden, ohne das Resultat zu beeinflussen.

Die **Polymorphie** ist die Vielgestaltigkeit. Ein Bezeichner nimmt abhängig von seiner Verwendung Objekte unterschiedlichen Datentyps an. Die Nutzung der Polymorphie führt zu wesentlich flexibleren Programmen. Sie steigert damit die Wartbarkeit und Änderbarkeit.

Objekte erhalten durch **Vererbung** die Attribute und Methoden anderer Objekte.

1.3 Prinzipien des objektorientierten Entwurfs

Unter einem **Modul** (Objjekte, Klassen, Datentypen) versteht man einen überschaubaren und eigenständigen Teil einer Anwendung. Solch ein Modul hat nun innerhalb einer Anwendung eine ganz bestimmte Aufgabe, für die es die Verantwortung trägt. Ein Modul hat eine oder mehrere Verantwortungen. Module bestehen aus weitere abhängigen Untermodulen.

Das Prinzip der einzigen Verantwortung besagt, dass jedes Modul genau eine Verantwortung übernimmt und ede Verantwortung genau einem Modul zugeordnet werden soll. Die Verantwortung setzt bestimmte zeitliche ändernbare Anforderungen des Moduls um.

Ein Modul soll zusammenhängend (kohäsiv) sein. Alle Teile eines Moduls sollten mit anderen Teilen des Moduls zusammenhängen und voneinander abhängig sein. Haben Teile eines Moduls keinen Bezug zu anderen Teilen, kann man davon ausgehen, dass man diese Teile als eigenständige Module implementieren kann. Eine Zerlegung in Teilmodule bietet sich damit an.

Wenn für die Umsetzung viele Module zusammenarbeiten müssen, bestehen Abhängigkeiten zwischen diesen Modulen. Die Module sind gekoppelt. Die Kopplung zwischen Module sollte möglichst gering sein. Das kann man erreichen, indem man die Verwntwortung für die Koordination der Abhängigkeiten einem neuen Modul zuweist. Hierbei sollte man aber darauf achten, dass man bestehende Abhängigkeiten durch die Einführung eines vermittelten Moduls nicht verschleiert.

Eine Aufgabe, die ein Programm umsetzen muss, betrifft häufig mehrere voneinander zusammenhängende und abgschlossene Anliegen, die getrennt betrachtet und als getrennte Anforderungen formuliert werden können. Eine identifizierbare Funktionalität eines Systems sollte innerhalb dieses Systems nur einmal umgesetzt sein, Wiederholungen vermeiden.

Module sollten angepasst werden, offen für Erweiterung und geschlossen für Änderung. Das Modul kann so strukturiert werden, dass die Funktionalität, die fpr eine Variante spezifisch ist, sich durch eine andere Funktionalität leicht ersetzen lässt. Die Funktionalität der Standardvariante muss dabei nicht unbedingt in ein separates Modul ausgelagert werden. Das Modul soll definierte Erweiterungspunkte bieten, an die sich die Erweiterungsmodule anknüpfen lassen.

Erweiterungspunkte kann man in der Regel durch das Hinzufügen einer **Indirektion** erstellen. Das Modul darf die variantenspezifische Funktionalität nicht direkt aufrufen. Das Modul konsultiert eine Stelle, die bestimmt, ob die Standardimplementierung oder ein Erweiterungsmodul aufgerufen werden soll.

Jede Abhängigkeit zwischen zwei Module sollte explizit formuliert und dokumentiert sein. Ein Modul sollte immer von einer klar definierten Schnittstelle des anderen Moduls abhängig sein und nicht von der Art der Implementierung der Schnittstelle. Die Schnittstelle eines Modulssollte getrennt von der Implementierung betrachtet werden können.

Eine Abstraktion beschreibt das in einem gewählten Kontext Wesentliche eines Gegenstands oder eines Begriffs. Durch eine Abstraktion werden die Details ausgeblendet, die für eine bestimmte Betrachtungsweise nicht relevant sind. Abstraktionen ermöglichen es, unterschiedliche Elemente zusammenzufassenm, die unter einem bestimmten Gesichtspiunkt gleich sind.

1.4 Die Struktur der Obektorientierung

2 JAVA

2.1 Elemente der Programmiersprache Java

2.1.1 Bytecode

Der Compiler erzeugt aus dem Quellcode den so genannten Bytecode. Dieser Code ist binär und Ausgangspunkt für die virtuelle Machine zur Ausführung.

2.1.2 Java Virtual Machine

Die Java Virtual Machine kümmert sich um den Bytecode, den Quellcode auszuführen. Die Laufzeitumgebung lädt den Bytecode, prüft ihn und führt ihn in einer kontrollierten Umgebung aus. Java ist plattform- und Betriebssystemunabhängig.

2.1.3 Objektorientierung

Eine Laufzeitumgebung eliminiert viele Fehler. Objektorientierte Programmierung versucht, die Komplexität des Softwareproblems besser zu modellieren. Menschen denken objektorientiert, darum Java bildet diese ab. Objekte bestehen aus Eigenschaften, also Dinge, die ein Objekt "hat" und "kann". Objekte entstehen aus Klassen, das sind Beschreibungen für den Aufbau von Objekten.

Primitive Datentypen für numerische Zahlen oder Unicode-Zeichen werden nicht als Objekte betrachtet. Das Java-Security-Modell sicherstellt den Ablauf. Der Verifier liest Code und überprüft die Korrektheit. Treten Sicherheitsprobleme auf, werden sie durch Exceptions zur Laufzeit gemeldet. Das Security-Manager überwächt Zugriffe auf das Dateisystem, die Netzwerk-Ports, externe Prozesse und weitere Systemressourcen.

In Java gibt es keine Zeiger auf Speicherbereiche, dagegen führt Java Referenzen ein. Eine Referenz repräsentiert ein Objekt, und eine Variable speichert diese Referenz, sie wird Referenzvariable genannt. JVM verbindet die Referenz mit einem Speicherbereich und einem Referenztyp; der Zugriff, Dereferenzierung genannt, ist indirekt. Referenz und Speicherblock sind getrennt.

In Java gibt es keine benutzerdefinierten überladenen Operatoren. Da das Operatorzeichen auf unterschiedlichen Datentypen gültig ist, nennt sich so ein Operator "überladen". Bei Zeichenketten werden Pluszeichen als Konkatenation angewendet. Java braucht keine Präprozessoren.

2.1.4 Java Platform

Mit dem ava Development Kit (JDK) lassen sich Java SE-Applikationen entwickeln. Dem JDK sind Hilfsprogramme beigelegt, die für die Java-Entwicklung nötig sind. Dazu zählen der essenzielle Compiler, abe rauch andere Hilfsprogramme, etwa zur Signierung von Java-Archiven oder zum Start einer Management-Konsole.

Das Java SE Runtime (JRE) enthält genau das, was zur Ausführung von Java-Programmen nötig ist. Die Distribution umfasst nur die JVM und Java-Bibliotheken, aber weder den Quellcode der Java-Bibliotheken noch Tools wie Management-Tools.

2.1.5 Das erste Programm

Ein Compiler übersetzt bzw. transformiert das geschriebene Programm in eine andere Repräsentation nämlich den Bytecode und erzeugt aus dem Program mit Endung `.java` die Daten `.class`.

Wenn der Compiler aufgrund eines syntaktischen Fehlers eine Übersetzung in Java-Bytecode nicht durchführen kann, spricht man von einem Compilerfehler.

Eine Laufzeitumgebung liest die Bytecode-Datei Anweisung für Anweisung aus und führt sie auf den konkreten Mikroprozessor aus. Der Interpreter bringt das Programm zur Ausführung.

Ein Java-Projekt braucht eine ordentliche Ordnerstruktur, und hier gibt es zur Organisation der Dateien unterschiedliche Ansätze. Die einfachste Form ist, Quellen, Klassendateien und Ressourcen in ein Verzeichnis zu setzen. Es gibt zwei Verzeichnisse `src` für die Quellen und `bin` für die erzeugten Klassendateien. Ein eigener Ordner `lib` ist sinnvoll für Java-Bibliotheken.

2.2 Imperative Sprachkonzepte

2.2.1 Elemente von Java

Unter dem Begriff **Semantik** versteht man die Lexikalik, Syntax und Semantik eines Programms. Der Compiler verläuft diese Schritte bevor er den Bytecode erzeugt.

Ein **Token** ist eine lexikalische Einheit, die dem Compiler die Bausteine des Programms liefert. Der Compiler erkenntan der Grammatik einer Sprache, welche Folgen von Zeichen ein Token bilden.

Whitespaces sind Leerzeichen, Tabulatoren, Zeilenvorschub und Seitenvorschubzeichen.

Neben den Trennern gibt es noch zwölf ASCII-Zeichen geformte Tokens, die als **Separator** definiert werden: `( ) { } [ ] ; , . ... @ ::`

Für Variablen, Methoden, Klassen und Schnittstellen werden **Bezeichner**, auch **Identifizierer** genannt, vergeben. Unter Variablen sind dann Daten verfügbar. Methoden sind die Unterprogramme in objektorientierten Programmiersprachen, und Klassen sind die Bausteine objektorientierter Programme. Ein Bezeichner ist eine Folge von Zeichen, die fast beliebig sein kann. Die Zeichen sind Elemente aus dem Unicode-Zeichensatz. Der Bezeichner muss mit einem Java-Buchstaben beginnen.

Ein Java-Buchstabe umfasst unsere lateinische Buchstaben “A” bis “Z”, “a” bis “z”, sondern auch viele Zeichen aus dem Unicode-Alphabet, den Unterstrich, Währungszeichen, griechische oder arabische Buchstaben. Java unterscheidet zweischen Gross- und Kleinschreibung. Nicht erlaubt sind Zahlen am Anfang, Leerzeichen, Ausrufezeichen, reservierte Wörter oder reservierte Schlüsselwörter.

Ein **Literal** ist ein konstanter Ausdruck wie die Wahrheitswerte `true` und `false`, Integrale Literale für Zahlen, Fliesskommaliterale, Zeichenliterale wie \n, String-Literale für Zeichenketten wie “Hello World”, Referenztypen wie `null`.

Bestimmte Wörter sind reservierte Schlüsselwörter com Compiler besonders behandelt. Schlüsselwörter bestimmen die Sprache eines Compilers. Es können keine eigenen Schlüsselwörter hinzugefügt werden. Schlüsselwörter sind:

`abstract`, `continue`, `for`, `new`, `switch`, `assert`, `default`, `goto`, `package`, `synchronized`, `boolean`, `do`, `if`, `private`, `this`, `break`, `double`, `implements`, `protected`, `throw`, `byte`, `else`, `import`, `public`, `throws`, `case`, `enum`, `instanceof`, `return`, `transient`, `catch`, `extends`, `int`, `short`, `try`, `char`, `final`, `interface`, `static`, `void`, `class`, `finally`, `long`, `strictfp`, `volatile`, `const`, `float`, `native`, `super`, `while`.

Der Compiler überliest alle Kommentare und die Trennzeichen bringen den Compiler von Token zu Token. Zeilenkommentare kann man mit Schrägsstrichen `//` und kommentieren den Rest einer Zeile bis Zeilenumbruchzeichen aus. Blockkommentare (“Wie”) kommentiert in Blöcke mit `/* */` aus. Javadoc-Kommentare (“Was”) sind besondere Blockkommentare mit `/** */` und beschreibt die Methode oder die Parameter, aus denen sich später die API-Dokumentation generieren lässt. Kein Kommentar kommt in den Bytecode.

2.2.2 Anweisungen

Programme sind Ablauffolgen, die im Kern aus Anweisungen bestehen. Sie werden zu grösseren Bausteinen zusammengesetzt, den Methoden, die wiederum Klassen bilden. Klassen selbst werden in Paketen gesammelt, und eine Sammlung von Paketen wird als Java-Archiv ausgeliefert.

Durch Anweisungen werden Algorithmen geschrieben. Anweisungen können Ausdrucksanweisungen, Zuweisungen oder Methodenaufrufe, Fallunterscheidungen, Schleifen für Wiederholungen sein.

Anweisungen müssen in einen Rahmen gepackt werden. Dieser Rahmen heisst Kompilationseinheit und deklariert eine Klasse mit ihren Methoden und Variablen. Anweisungen ausserhalb von Klassen sind nicht erlaubt. Der Klassenname ist ein Bezeichner und beinhaltet die gleiche Dateiname. Klassennamen beginnen mit Grossbuchstabe und Methoden sind kleingeschrieben. Zwischen den geschweiften Klammern folgen Deklarationen von Methoden und zwischen den Methoden die Anweisungen.

Eine besondere Methode ist `public static void(String[] args){}`. Die `main(String[])`-Methode ist für die Laufzeitumgebung etwas Besonders, denn beim Aufruf des Java-Interpreters mit einem Klassennamen wird diese Methode als Erstes ausgeführt. Demnach werden die Anweisungen ausgeführt, die innerhalb der geschweiften Klammern stehen. Der Parameter `args` wird immer verwendet.

Haltet man sich nicht an die Syntax für den Startpunkt, so kann der Interpreter die Ausführung nicht beginnen und man hätte einen semantischen Fehler produziert, obwohl die Methode korrekt gebildet ist.

Die Methode `println(...)` gibt Meldungen auf der Konsole aus. Innerhalb der Klammern können Argumente angegeben werden. Zeichenketten oder **Strings** sind Argumente oder eine Folge von Buchstaben, Ziffern oder Sonderzeichen in doppelten Anführungszeichen. Die Methode `println(...)` gehört zum Typ `out` und diese zu `System`.

```
1 public class PrimeraClase
2 {
3     public static void main(String args[])
4     {
5         System.out.println("Hola Mauri");
6     }
7 }
```

Java erlaubt Methoden, die gleich heissen, denen aber unterschiedliche Dinge übergeben werden können; diese Methoden nennt man **überladen**. Viele `println()`-Methoden akzeptieren zahlartige Argumente und sind überladen.

```
1 public class OverloadedPrintln {
2     public static void main( String[] args ) {
3         System.out.println( "Verhaften Sie die ueblichen Verdaechtigen!" );
4         System.out.println( true );
5         System.out.println( -273 );
6         System.out.println(); // Gibt eine Leerzeile aus
7         System.out.println( 1.6180339887498948 );
8     }
9 }
```

Die Methode `printf()` ermöglicht variable Argumentenlisten gemäss einer Formatierungsanweisung. Die Formatierungsanweisung \n setzt einen Zeilenumbruch, \d ist ein Platzhalter für eine ganze Zahl, \f ist ein Platzhalter für eine Fliesskommazahl, \s ist eine Zeichenkette oder etwas, was in einen String konvertiert werden soll.

```
1 public class VarArgs {
2     public static void main( String[] args ) {
3         System.out.printf( "Was sagst du?%n" );
4         System.out.printf( "%d Kanaele und ueberall nur %s.%n", 220, "Katzen" );
5     }
6 }
```

Methodenaufrufe lassen sich als Anweisungen einsetzen, wenn sie mit einem Semikolon abegschlossen sind, man spricht von einer **Ausdrucksanweisung** (expression statement). Jeder Methodenaufruf mit Semikolon bildet eine Ausdrucksanweisung. Dabei ist es egal, ob die Methode selbst eine

Rückgabe liefert oder nicht.

Die Methode `Math.random()` liefert als Ergebnis einen Zufallswert zwischen 0 (inklusiv) und 1 (exklusiv).

In einer objektorientierte Programmiersprache sind alle Methoden an bestimmte Objekte mit einem Zustand gebunden. Alle Operationen und Zustände sind an Objekte bzw. Klassen gebunden. Der Aufruf einer Methode auf einem Objekt richtet die Anfrage genau an dieses bestimmte Objekt. Die Methode `println(...)` gehört zu einem Objekt `out`, das die Bildschirmausgabe übernimmt. Dieses Objekt `out` bzw. `err` wiederum gehört zu der Klasse `System`.

Die Deklaration einer Klasse oder Methode kann einen oder mehrere **Modifizierer** enthalten, die zum Beispiel die Nutzung einschränken oder parallelen Zugriff synchronisieren. Der Modifizierer `public` ist ein Sichtbarkeitsmodifizierer. Er bestimmt, onb die Klasse bzw. die Methode für Programmcode anderer Klassen sichtbar ist oder nicht. Der Modifizierer `static` zwingt den Programmierer nicht dazu, vor dem Methodenaufruf ein Objekt der Klasse zu bilden. Dieser Modifizierer bestimmt die Eigenschaft, ob sich eine Methode nur über ein konkretes Objekt aufrufen lässt oder eine Eigenschaft der Klasse ist, sodass für den Aufruf kein Objekt der Klasse nötig wird.

Ein Block fasst eine Gruppe von Anweisungen,die hintereinander ausgeführt werden. Ein Block `{}` ist eine Anweisung, die in geschweiften Klammern eine Folge von Anweisungen zu einer neuen Anweisung zusammenfasst. Ein Block kann überall dort verwendet werden, wo auch eine einzelne Anweisung stehen kann. Der neue Block hat jedoch eine Besonderheit in Bezug auf Variablen, da er einen lokalen Bereich für die darin befindlichen Anweisungen inklusive der Variablen bildet.

Ein Block ohne Anweisung nennt sich ein leerer Block. Er verhält sich wie eine leere Anweisung, also wie ein Semikolon. Es gibt innere und äussere Blöcke. Blöcke fassen Anweisungen zusammen.

2.2.3 Datentypen, Variablen und Zuweisungen

Java speichert Variablen. Eine Variable ist ein reservierter Speicherbereich und belegt eine feste Anzahl von Bytes. Variablen und Ausdrücke haben einen **Datentyp** und einen **Datenwert**. Der Datentyp bestimmt die zulässigen Operationen. Java ist eine streng typisierte Programmiersprache. Datentypen werden unterteilt in **primitive Datentypen** (Zahlen, Unicode-Zeichen und Wahrheitswerte) und **Referenztypen** (Zeichenketten, Datenstrukturen, Zwergpinscher) und Bytecode durch den Compiler einfacher erzeugt.

Die primitive Datentypen unterteilen sich in **arithmetische Typen** und **Wahr-heitswerte**. Folgende Tabelle vermittelt einen Überblick

Typ	Belegung (Wertebereich)
boolean	`true` oder `false`
char	16-Bit-Unicode-Zeichen (0x0000 ... 0xFFFF)
byte	–27 bis 27 – 1 (–128 ... 127)
short	–215 bis 215 – 1 (–32.768 ... 32.767)
int	–231 bis 231 – 1 (–2.147.483.648 ... 2.147.483.647)
long	–263 bis 263 – 1 (–9.223.372.036.854.775.808 ... 9.223.372.036.854.775.807)
float	1,40239846E–45f ... 3,40282347E+38f
double	4,94065645841246544E–324 ... 1,79769131486231570E+308

Es gibt mehr negative Werte als positive Werte, das liegt an der Kodierung im Zweierkomplement. Bei `float` und `double` ist das Vorzeichen nicht angegeben, die Wertebereiche unterscheiden sich nicht, die kleinsten und grössten darstellbaren Zahlen können sowohl positiv als auch negativ sein.