Grundlagen der Programmkonstruktion

Kontrollfragen - 2. Test

Kapitel 3

Wodurch unterscheidet sich der objektorientierte vom prozeduralen Programmierstil?
Bei der prozeduralen Programmierung werden Programme durch kleinere Teilprobleme (=Prozeduren) aufgebaut. Diese Art der Programmierung ist besonders gut geeignet, um kleine Programme zu schreiben, die einzelne Algorithmen implementieren.
In der objektorientierten Programmierung hingegen wird ein Programm in mehrere abstrakte Maschinen (welche von außen als Objekte betrachtet werden) unterteilt, welche miteinander arbeiten, ohne dass eine Maschine Details der anderen kennt.
Der größte Unterschied jener Stile ist demnach, dass bei der objektorientierten Programmierung die zentrale Kontrolle aufgegeben wird, während wir uns bei der prozeduralen Programmierung die Kontrolle über alle Details des gesamten Ablaufs verschaffen.

Was ist ein Objekt?
Ein Objekt ist eine Instanz einer Klasse. Sein Zustand wird durch seine Variablen definiert, sein Verhalten durch die Methoden. Seine Identität ist eindeutig.

Für welche Arten von Programmen eignet sich die objektorientierte Programmierung gut?
Große Programme mit einem langen Softwarelebenszyklus
Umfangreiche Programme die leicht zu erweitern sein sollen.

Mit welchen Problemen muss man bei der Entwicklung großer Programme rechnen?
Sie beruhen auf einer Vielzahl an einzelnen Algorithmen (welche jeweils kleine Teilaufgaben lösen), die auf komplexe Weise miteinander verbunden sind damit daraus ein in sich konsistentes, ganzes Programm entsteht
Aufgrund der großen Komplexität ist es in der Regel nicht möglich, ein großes Programm in einem einzigen Schritt aus vielen einzelnen Teilen zusammenzusetzen
Langlebige Software muss über einen langen Zeitraum gewartet werden (aufwendig, kostenintensiv)
Die Struktur eines Programmes ist ein entscheidendes Qualitätsmerkmal (Einfachheit, Verständlichkeit, Wartbarkeit)

Was versteht man unter inkrementeller Softwareentwicklung?

Die schrittweise Erstellung eines Programms. Anfangs wird nur ein kleiner Teil der Aufgaben gelöst. Schritt für Schritt werden Teile hinzugefügt und Erfahrungen aus den vorherigen Schritten fließen in die Entwicklung mit ein.

Was bedeutet der Begriff Faktorisierung? Wann ist eine Faktorisierung gut, wann nicht?
Faktorisierung bedeutet die Aufteilung großer Programme in kleine Einheiten , in welchen zusammengehörige Eigenschaften und Aspekte des Programms zusammengefasst sind. Wenn mehrere Stellen in einem Programm die gleichen Befehle ausführen, sollen diese Stellen durch Aufrufe einer Methode ersetzt werden, die diese Befehle ausführen.
Gute Faktorisierung zeichnet sich aus, wenn man eine Änderung all dieser Stellen durch eine einzige Änderung der Methode bewirkt.

Hat ein Programm Eigenschaften und Aspekte, die nichts miteinander zu tun haben, so müssen diese klar voneinander getrennt werden. Eine Änderung soll unabhängig voneinander vorgenommen werden. In diesem Fall ist eine Faktorisierung keine gute Idee.
Die Entscheidung ob eine Faktorisierung durchgeführt werden soll, oder nicht, ist dabei oft sehr schwierig. An dieser Stelle nur zwei Stichworte: Klassenzusammenhalt und Objektkoppelung (Siehe Seite. 222)

Wodurch unterscheiden sich Objektvariablen von lokalen Variablen?
Lokale Variablen existieren nur solange bis aus der aufgerufenen Prozedur (in der sie definiert werden) zurückgekehrt wird
Objektvariablen dienen der Ablage von Daten eines Objekts und sind überall im Programm gültig, wo es eine Referenz auf das Objekt gibt

Was ist und wozu dient Kapselung, Data Hiding und Datenabstraktion?
Die Eigenschaften eines Objekts, Variablen und Methoden zu einer Einheit zusammenzuführen nennt man Datenkapselung. Zusammen mit data hiding, dem Verstecken von Details durch einen entsprechenden Zugriffsmodifzierer (z.B. private), spricht man von Datenabstraktion.

Was ist eine Nachricht, und warum spricht man vom Senden von Nachrichten und nicht einfach nur vom Aufruf von Methoden?
Aufforderung an das Objekt, eine seiner Methoden auszuführen (welche Methode mit welchen aktuellen Parametern)
Um klar zu machen, dass wir es mit voneinander weitgehend unabhängigen Objekten zu tun haben, die miteinander kommunizieren
Man denkt beim Nachrichtensenden in erster Linie nicht daran, dass dadurch ein bestimmter Teil eines wohldefinierten Algorithmus ausgeführt wird, sondern hat nur eine grobe Vorstellung davon, was die Methode bewirkt.
Sofern es zu Polymorphismus (“Vielgestaltigkeit”) kommt, wird je nach Objekttyp eine

andere Methode ausgeführt - welche das ist, entscheidet aber nicht der Aufrufer.

Was versteht man unter einer Schnittstelle eines Objekts, was unter seiner Implementierung?
Schnittstelle: wird aus den Methodendeklarationen und deren abstrakter Beschreibung gebildet; entspricht der Aussenansicht eines Objekts
Implementierung: legt das in der Schnittstelle unvollständig definierte Verhalten im Detail fest; entspricht der Innenansicht eines Objekts

Ich hätt eher gesagt: Seite 170f

Was sind und wozu verwendet man Klassen?
Beschreiben die Struktur (Implementierung) eines Objektes im Detail
Kann man als Bauplan für die Erzeugung neuer Objekte zur Laufzeit verstehen

Wie kann man in Java die Sichtbarkeit beeinflussen?
Mittels Modifiern
Paket-Sichtbarkeit (standard) , public, private, protected (Sichtbar innerhalb der Klasse + dessen vererbten Klassen)

Wo sollen die meisten Objektvariablen sichtbar sein?
Nur dort wo es auch notwendig ist, dass auf die Variablen zugegriffen werden kann

Was haben Getter- und Setter-Methoden mit Data Hiding zu tun?
Getter und Setter ermöglichen üblicherweise einen Einblick in die innere Datenstruktur einer Klasse und wirken daher der Absicht des Data Hiding direkt entgegen. Ein vielfacher Rückgriff auf Getter/Setter weist daher auf schlechte Faktorisierung der Klassen hin - die Kapselung ist nicht gegeben. Zusätzlich wird in der Praxis in Gettern/Settern häufig direkt die Referenz auf Objektvariablen übergeben, was bedeutet, dass hier auch Seiteneffekte auftreten können (die referenzierte Instanz kann geändert werden, wenn sie nicht final deklariert ist) - das ist ein Sicherheitsproblem und wirkt ebenfalls gegen die Idee des Objekts als Black Box.

Durch getter / setter kann ich den einfachen zugriff auf daten ermoeglichen, die ich intern womoeglich ganz anders represaentiere. BeispielS. 179 steht:: Bitfield klasse die das feld intern in einem int / long speichert, aber den zugriff per index ermoeglicht und boolean erwartet / zurueckliefert.

public void setX(double newX) { x = newX; }
public double getX() { return x; }
Es handelt sich um sogenannte Setter- bzw. Getter-Methoden, die nur eine Objektvariable setzen oder abfragen. Durch solche Methoden bekommt man Zugriff auf Objektvariablen, obwohl die Variablen selbst als private deklariert sind. Lehrbücher verwenden diese Beispiele schlicht wegen ihrer Einfachheit. In der Programmierpraxis sollte man Setter- und Getter-Methoden vermeiden, so gut es möglich ist, da durch sie die Vorteile des

Data Hiding weitgehend verloren gehen.

Erklären sie die Begriffe Identität, Zustand und Verhalten.
Identität: eindeutig, Adresse des Objekts im Speicher, über die Identität ist das Objekt ansprechbar, Objekte sind “identisch”, wenn Referenzen auf denselben Speicherplatz verweisen, kann durch Zustandsänderung nicht verloren gehen
Zustand: definiert durch Objektvariablen, Objekte sind “gleich” wenn sie denselben Zustand (+ Verhalten, durch die gemeinsame Klasse gegeben) haben, Gleichheit kann durch Zustandsänderung (z.B. Variablenwert Änderung) verloren gehen
Verhalten: definiert durch Methoden - was wird bei Erhalt einer Nachricht getan. Objekte haben dasselbe Verhalten, wenn sie bei gleicher Nachricht und im gleichen Zustand das Gleiche machen

Wie vergleicht man in Java Objekte auf Identität bzw. Gleichheit?
Identität: == (bzw. != für Ungleichheit)
Gleichheit: equals (g eigene Klassen selbst zu definieren)

Wozu dient ein Konstruktor und wie definiert man ihn?
Dient zur Herstellung des Initialzustandes eines Objekts (Initialisierung der Variablen)
Aufbauzur wie normale Methoden, aber: Name entspricht dem der Klasse, kein Ergebnistyp

Wofür verwendet man die Pseudovariable this und Ausdrücke der Form this(...)?
“this” ist eine Referenz auf das Objekt, in dem man sich gerade befindet
this(...): ruft einen anderen Konstruktor der selben Klasse auf

Wie setzt man statische Methoden und Klassenvariablen ein?
Schlüsselwort: “static” kennzeichnet statische Methoden/Klassenvariablen
Methoden: Nachricht wird an Klasse, nicht an Objekt gesendet (es muss keine Instanz einer Klasse erzeugt werden, um eine statische Methode aufrufen zu können)
Variablen: gehören zur Klasse selbst. Auch hier muss keine Instanz genutzt werden, um auf Variablen zuzugreifen.

Was unterscheidet Konstanten von Klassenvariablen?
Konstanten besitzen neben dem Schlüsselwort “static” auch noch “final”
Konstanten haben stets denselben Wert, Klassenvariablen können verändert werden, d.h. Konstanten können nur einmalig Werte zugewiesen werden.
Konstantenname wird meist GROSS geschrieben (vgl. Programmierstil)
Definition als “public” häufig sinnvoll
Der Wert der Konstante liegt häufig zu Beginn der Laufzeit bereits fest, kann jedoch auch während der Laufzeit einmalig zugewiesen werden.

Wozu dienen Interfaces?
Beschreiben beliebig komplexe Schnittstellenstrukturen
Führen klare Trennung zwischen Teilen eines objektorientierten Programms ein
Führen zu besserer Wartbarkeit und v.a. auch automatisierter Testbarkeit (Unittests)
Behandlung verschiedener Objekte mit einheitlichem Interface (bsp.: Plugin)

Was meint man, wenn man von der Implementierung eines Interfaces spricht?
Konkrete Ausformulierung der Methoden in der implementierenden Klasse durch Überschreiben aller im Interface festgelegten Methoden. Weiters die Implementierung der zu implementierenden Variablen oder/und Properties nach den Richtlinien des geerbten Interfaces.

Wann spricht man von Polymorphismus? Welche Rolle spielt der Polymorphismus in der objektorientierten Programmierung?
Eigenschaft, dass ein Objekt mehrere Typen haben kann
Rolle: Reduziert den Programmieraufwand (z.B. bei Punkt und Scheibe kann man mit einer Methode Berechnung unterschiedlicher Objekte durchführen) und Wartung wird reduziert. - verschiedene Datentypen können mit dem selben Interface verarbeitet werden. Das alles bezieht sich auf subtype-polymorphism, daneben gibt es aber auch noch parametric und ad-hoc polymorphism.
siehe auch http://de.wikipedia.org/wiki/Polymorphie_%28Programmierung%29

wir haben das auf Seite 193 in der Mitte als Antwort genommen:

Polymorphismus heißt, dass ein Objekt mehrere Typen haben kann. Es gilt zusätzlich, dass jeder Typ gleichzeitig Unter- bzw. Obertyp von sich selbst ist. Da die Schnittstellen frei sind von den Einschränkungen der Einfachvererbung, wird durch Schnittstellen ein sehr flexibler Programmentwurf möglich. So kann ein Interface für zwei unterschiedliche Sachen verwendet werden.

Unter welchen Bedingungen ist ein Typ U Untertyp eines Typs T?
wenn es im Diagramm von U aus einen Pfad zu T gibt (T ist Obertyp von U), also wenn U von T oder einem Untertypen von T abgeleitet ist;
kann in Java mit obj*U instanceof objT* geprüft werden
jeder Typ ist gleichzeitig Ober- bzw. Untertyp von sich selbst

Wozu benötigt man dynamisches Binden?
Wenn die auszuführende Methode erst zur Laufzeit bestimmt werden soll. (diese Methode kann dann polymorph sein)

Inwiefern hängt dynamisches Binden mit Mehrfachverzweigungen zusammen?
Dynamisches Binden ist aufgrund der Mehrfachverzweigung bei der Ausführung etwas weniger effizient als statisches Binden.

Warum ist dynamisches Binden gegenüber switch-Anweisungen zu bevorzugen?
Unter anderem ist dynamisches Binden gegenüber einer switch-Anweisung effizienter. Außerdem ist es auch so, dass sich ein Programm, welches dynamisches Binden benutzt, mit weniger Code/mit weniger Aufwand ausbauen lässt, als ein Programm, das eine switch-Anweisung verwendet. Hinzu kommt auch noch, dass die switch-Anweisung anfälliger für Fehler ist.

Weil dynamisches Binden effizienter ist.

Wenn Schnittstellen (Interfaces) verwendet werden, kann ein beliebiges (auch noch nicht bekanntes) Objekt dahinterliegen, welches diese Schnittstelle implementiert hat. Somit ist dynamisches Binden flexibler als switch-Anweisungen, welche einen Cast auf den entsprechenden Typ machen müssten und die entsprechende Methode aufrufen müssen.

Welchen Zweck haben Spezialisierungen und Analogien zur realen Welt?
Ein auf die reale Welt und Spezialisierung konzentrierter Entwurf lässt sich oft relativ einfach erweitern, da alles von Anfang an aus einem weiteren Blickwinkel betrachtet wird als eigentlich notwendig ist.
Spezialisierungen und Analogien zur realen Welt sind Hilfsmittel, die uns helfen zu erkennen, wie gut ein bestimmtes Typdiagramm für ein Programm geeignet ist.

Was besagt das Ersetzbarkeitsprinzip?
Ein Typ U soll genau dann Untertyp eines Typs T sein, wenn Objekte vom Typ U überall verwendbar sind, wo Objekte vom Typ T erwartet werden.

Was versteht man unter Vererbung?
Die Ableitung einer Klasse aus einer anderen, also die Strukturübernahme samt Variablen und Methoden. Jede Klasse kann (in Java) nur von EINER Klasse erben.

Anmerkung: es können jedoch beliebig viele Interfaces implementiert werden.

Erklären Sie die Begriffe Basisklasse, abgeleitete Klasse, Unterklasse und Oberklasse.
Basisklasse == Oberklasse: Klasse, von der abgeleitet wird
abgeleitete Klasse == Unterklasse: erbt Methoden der Ober-/Basisklasse

Was ist eine überschriebene Methode?

Eine geerbte Methode die, in der abgeleiteten Klasse/Unterklasse überschrieben wird, also deren Inhalt durch neuen Programmcode ersetzt wird. (selbe Signatur, polymorphe Methode, siehe wikipedia)

Warum deklariert man Variablen nicht generell als protected?
Als “protected” deklarierte Variablen sind in abgeleiteten Klassen sichtbar. Der direkte Zugriff auf Variablen in der Oberklasse stellt aber einen schlechten Programmierstil dar.

Wie werden Objekte abgeleiteter Klassen initialisiert?
Zuerst wird immer ein Konstruktor der Oberklasse aufgerufen
mittels super(...) können der Konstruktor bestimmt und Variablen übergeben werden
lt. Skriptum: “ Beim Erzeugen eines neuen Objekts wird zuerst immer ein Konstruktor der Oberklasse ausgeführt, dann der Konstruktor der Unterklasse. Mittels super(...) kann man den Konstruktor der Oberklasse bestimmen und Parameter weiterleiten.”

Wozu dient super(...) und wo kann diese Anweisung verwendet werden?
zur genauen Spezifikation des aufzurufenden Konstruktors in der Oberklasse

Unterscheidet sich ein Methodenaufruf von einem Variablenzugriff hinsichtlich dynamischem Binden?
Beim Zugriff auf Variablen erfolgt kein dynamisches, sondern statisches Binden (Skriptum S. 208 bzw. Folien vom 27.10.11 S.15)

Zu welchem Zweck kann man Klassen und Methoden mit einem Modifier abstract bzw. final versehen?
abstrakte Klassen: es können keine Instanzen dieser Klasse erstellt werden,
abstrakte Methoden: nicht implementiert, müssen von abgeleiteten Klassen implementiert werden (ähnlich wie bei Interfaces!)
final Klassen: von diesen Klassen können keine weiteren Klassen abgeleitet werden
final Mehoden: die Methode kann in keiner abgeleiteten Klasse überschrieben werden

Wie kann man durch Interfaces zusätzliche Struktur in ein Programm bringen?
ein Interface entspricht einer Rolle, die die Klasse übernehmen kann, zb Interface Drawable für Klassen, die sich selbst zeichnen können. (rechteck, bild, etc)
Interfaces können von mehreren Klassen implementiert werden und übernehmen damit die Rolle eines gemeinsamen Obertyps aller dieser Klassen.

Welche Methoden sind in Object vorhanden und welchen Zweck haben sie?+

-) String toString(): umwandeln in Zeichenkette
-) boolean equals(Object) überprüft auf Gleichheit
-) int hashCode() gleiche/verschiedene Objekte -> gleiche hashWerte
-) Class getClass() Klasse des Objekts (= dynamischer Typ)
-) Object clone() erzeugt eine Kopie des Objekts
-) void finalize() aufräumen vor Speicherfreigabe
-) wait, notify, notifyAll nebenläufige Programmierung
(Folien vom 31.10.11 S.1)

Wie kann man zur Laufzeit den dynamischen Typ, also die Klasse eines Objekts feststellen (drei Möglichkeiten)?

- Class getClass() gibt die Klasse des Objekts zurück, siehe oben
- instanceof Operator (von Object) auf einen bestimmten Klassennamen ist das selbe wie this.getClass() == o.getClass() und liefert sicher False wenn die 2 Objekte verschieden sind. (siehe Skriptum S 215. erster Absatz) “o” steht für ein beliebiges initialisiertes Objekt.
- objekt.class, Klasse.class liefert die aktuelle Klasse zurück. Die Klasse eines Objekts ist schreibgeschützt und kann ohne Neuinitialisierung während der Laufzeit nicht verändert werden.

Was unterscheidet Casts auf Referenztypen von solchen auf elementaren Typen? Warum soll man sie vermeiden?
Casts auf Referenztypen , im Gegensatz zu Casts auf elementaren Typen, *ändern* (nicht nur temporär) den deklarierten Typ eines Ausdrucks. Genau deshalb sind Casts auf Referenztypen sehr fehleranfällig und sollen wenn möglich vermieden werden.

Wodurch unterscheiden sich die Pseudovariablen this und super voneinander?
this bezieht sich auf diese Klasse, super auf die übergeordnete. ist zb praktisch wenn man auf überschriebene Methoden zugreifen will, um code wiederzuverwenden. (Subklasse erweitert die Funktion der Superklasse einfach)

Warum eignet sich hashCode nicht für Vergleiche von Objekten?
hashCode gibt für gleiche Objekte immer die selbe Zahlenfolge aus, aber die Umkehrung gilt nicht, also können zwei Objekte ungleich sein, obwohl hashCode gleiche Zahlen liefert.

Welche Informationen sollen in Kommentaren von Klassen, Interfaces, Methoden, Konstruktoren und Objektvariablen enthalten sein?
Gute Kommentare beschreiben das, was ein geübter Programmierer nicht auf einem Blick aus dem Programmcode herauslesen kann, aber zum Verständnis wissen muss.
Klassen und Interfaces: allgemeine Informationen zum Typ; Zweck und Grobstruktur von Objekten des Typs
Methoden und Konstruktoren: Informationen, die man beim Schicken von Nachrichten

und Erzeugen von Objekten benötigt
Objektvariablen: Zweckbeschreibung, wenn nicht aus Name und Kontext ersichtlich

Welche Arten von Zusicherungen in Form von Kommentaren kann man unterscheiden?
Vorbedingung
Nachbedingung
Invariante
History Constraints

Inwiefern können Namen und Kommentare altern? Was kann man dagegen tun?
Durch Programmänderungen oder Erweiterungen kann sich der Einsatz und somit die Bedeutung der Variablen ändern, wodurch der ursprüngliche Name - falls er auf den Einsatz der Variable abgestimmt war- auch nicht mehr passend ist.
Genauso können Kommentare einen veralteten Einsatz einer Variable oder Methode beschreiben, was ebenso eine Änderung nötig macht.
Beim Ändern von Programmen müssen daher die Kommentare angepasst bzw. Variablen und Methoden umbenannt werden.

Wie können schlecht gewählte Namen und Kommentare zu unnötigem Programmcode führen?
Dadurch wird das Verständnis beim Lesen des Programmteiles beeinträchtigt und das Vertrauen in den Programmteil sinkt, weshalb womöglich von anderen Programmierern zusätzliche Absicherungen eingebaut werden (S. 221)

Was zeichnet gut faktorisierte Programme aus?
Bei guter Faktorisierung sind lokale Programmänderungen möglich: Es reicht, genau die Stelle, die man ändern will, zu betrachten. Es gibt durch lokale Änderungen keine oder nur sehr geringe Auswirkungen auf das restliche Programm.

Erklären Sie die Begriffe Klassenzusammenhalt und Objektkopplung. Wie hängen sie mit der Faktorisierung zusammen?
Klassenzusammenhalt: Grad des Zusammenhanges der Inhalte einer Klasse (hoch, wenn alle Variablen und Methoden gut zusammenpassen und die Namen und Kommentare treffende Beschreibungen darstellen)
Objektkopplung: Stärke der Abhängigkeit von Objekten voneinander (stark, wenn viele nach außen sichtbare Variablen oder Methoden, die Nachrichten schicken)

Bei der Faktorisierung will man einen hohen Klassenzusammenhalt und eine möglichst schwache Objektkopplung

Wie kann man den Klassenzusammenhalt und die Objektkopplung abschätzen?

Durch gedankliche Änderungen bzw. Durchlaufen eines Programmteiles kann man diese Aspekte gut abschätzen.

Wann sind notwendige Änderungen von Kommentaren gefährlich, wann eher harmlos?
Änderungen von Kommentaren bei Schnittstellen sind gefährlich, da sich damit nicht nur die Methode ändert, sondern auch alle Stellen im Programm, die eine Nachricht an die betroffene Schnittstelle übermitteln.
Änderungen von Kommentaren in einer Methode sind relativ harmlos, wenn die Schnittstelle gleich bleibt.
die kommentare müssen zu den Methoden u Schnittstellen passen - änderst du ein Kommentar, dann musst du die Methode/Schnittstelle anpassen!

Wie spezifiziert man das Verhalten?
Namen und Kommentare spezifizieren das Verhalten. Sie sind für die Einhaltung des Ersetzbarkeitsprinzips entscheidend. (Skript S.225)

Wann ist das Verhalten eines Untertyps mit dem eines Obertyps kompatibel?
Wenn sich alle Methoden in allen Objekten vom Untertyp so verhalten wie man es von entsprechenden Methoden des Obertyps erwartet hätte.

Wodurch entkoppelt Ersetzbarkeit Programmteile voneinander?
Ein Programmcode, der eine Nachricht auf ein Objekt sendet, muss nur Informationen über dieses Objekt wissen, braucht aber keine Informationen über die Untertypen dieses Objekts man dafür sorgen, dass jeder Anwender einen deklarierten Typ haben kann, der den tatsächlichen Bedürfnissen entspricht. Dabei muss man sich ganz auf das Verhalten der Methoden konzentrieren, also dafür sorgen, dass jede Methode in einem Untertyp ein zur entsprechenden Methode im Obertyp kompatibles Verhalten hat. Wenn man so vorgeht, entstehen Typhierarchien, die für eine gute Entkopplung sorgen, und in denen Typen weit objekts. Trotzdem kann dieser Code über das Objekt Nachrichten auf die Untertypen senden, ohne das zu wissen, und der Code wird trotzdem ausgeführt. Damit braucht man keine direkte Verbindung zwischen dem Code und jedem Untertyp zu haben, womit die Programmteile entkoppelt werden.

Welche Typen sind eher stabil?
Stabil sind vor allem Typen, die häufig verwendet werden und deswegen bereits gut getestet sind. Generell sind Typen weiter oben in der Typhierarchie stabiler als solche weiter unten.

Wo soll man besonders auf stabile Typen achten?
Die Wartbarkeit ist besser, wenn Objektvariablen und formale Parameter mit stabilen Typen deklariert sind, die sich im Laufe der Zeit kaum ändern. (Skript S.226)

Warum ist es nicht sinnvoll, möglichst viel Programmcode von Oberklassen erben zu wollen?
Beim Aufbau einer Typhierarchie soll man nicht darauf achten, möglichst viele

Methoden von einer Oberklasse zu erben. Stattdessen sollen in der Typhierarchie eher stabil sind. Anfangs hat man vielleicht das Gefühl, dass man dabei durch eine höhere Anzahl von Klassen und Interfaces unnötigen Programmcode schreibt. Letztendlich erspart man sich jedoch das Schreiben von viel Programmcode, und das Programm wird einfacher wartbar.

Kapitel 4

Was versteht man unter Algorithmen und Datenstrukturen, und wie hängen diese beiden Begriffe zusammen?
Algorithmus: Handlungsvorschrift von endlich vielen Schritten zur Lösung eines Problems
Datenstruktur: beschreibt, wie Daten relativ zueinander angeordnet sind und wie auf die einzelnen Elemente zugegriffen werden kann
hängen stark voneinander ab; Algorithmen setzen bestimmte Datenstrukturen voraus, können ohne diese also nicht entwickelt werden. Bei der Konstruktion von Programmen entscheiden wir anhand der benötigten Eigenschaften der Zugriffsoperationen, welche Datenstruktur zur Lösung unserer Aufgabe am ehesten passt.

Unter welchen Bedingungen sind zwei Algorithmen bzw. Datenstrukturen gleich? Wann sind sie es nicht?
Wenn sie dieselben funktionalen Eigenschaften liefern, also dieselben Ergebnisse.
In den nichtfunktionalen Eigenschaften (Aufwand, Speicherbedarf, Effizienz) unterscheiden sie sich jedoch.
B.S. 234 (unten)

Sind gleich wenn sie die gleiche Handlungsvorschrift befolgen. Können allerdings unterschiedlich implementiert sein. Unterschiedliche Algorithmen können gleiche Ergebnisse liefern.
Gleiche Ergebnisse sind kein hinreichendes Kriterium für gleiche Algorithmen.

Nennen Sie fünf unterschiedliche Datenstrukturen.
Array
verkettete Liste
Binärer Baum
Hashtabelle
Stack
Red-Black Tree
Tree

Wozu dienen Lösungsstrategien?

Unter einer Strategie versteht man das langfristig orientierte Vorgehen in grundlegenden Fragen. Dabei sollen vor allem die strategischen Ziele erreicht werden. Vereinfachung eines Systems bzw. von Algorithmen und Datenstrukturen

Warum sind so viele Datenstrukturen rekursiv?
Komplexe Algorithmen sind in rekursiven Datenstrukturen einfacher zu programmieren, kompakter, intuitiver und sie lassen sich später, dank der Übersichtlichkeit, besser bearbeiten.
Im Gegensatz zu rekursiven Methoden, rekursive Datenstrukturen sind kaum durch nicht-rekursive Datenstrukturen ersetzbar.

Rekursive Datenstrukturen beschreiben beliebig große Datenmengen, sie können dynamisch wachsen oder schrumpfen und sind daher flexibel.

Welche Zugriffsoperationen haben Stacks, verkettete Listen und binäre Bäume üblicherweise?
Stack: push, pop
Verkettete Liste: add, remove, contains
Binärer Baum: add, remove, contains

Welche charakteristischen Merkmale zeichnen eine verkettete Liste und einen binären Baum aus?
Verkettete Liste: jeder Listeneintrag verweist auf den nächsten Eintrag, Anzahl der Elemente unbeschränkt, Hinzufügen einfach, Elemente nicht direkt über Index zugreifbar, Suche aufwendig
Binärer Baum: jeder Eintrag verweist auf bis zu 2 Einträge (Sortierung!), gegenüber verketteten Liste ist die Suche effizienter und das Hinzufügen aufwendiger

Wie hängen Datenstrukturen mit gerichteten Graphen zusammen?
Praktisch alle rekursiven Datenstrukturen lassen sich durch gerichtete Graphen veranschaulichen. Eigenschaften dieser Graphen entsprechen auch denen der Datenstrukturen.

Wodurch unterscheiden sich rekursive Methoden von entsprechenden iterativen (und nicht rekursiven)?
Rekursive Methoden sind im Normalfall deutlich kürzer, da iterative Methoden lokale Variablen zur Zwischenspeicherung benötigen, wohin gegen in rekursiven Methoden von “this” Gebrauch gemacht wird. Weiters erspart man sich unter Umständen aufwendige bedingte Anweisungen. Dagegen erhält man bei Verwendung rekursiver Methoden eine hohe Anzahl von Methodenaufrufen.

Was haben rekursive Datenstrukturen und rekursive Methoden mit vollständiger Induktion gemeinsam?
Für Induktion und rekursive Datenstrukturen braucht man eine Basis. Formal können wird über vollständige Induktion beweisen, das nach Beendigung eines Aufrufs für alle betrachteten Datenelemente bestimmte Eigenschaften erfüllt sind.
…?!noch was
(S 247):
Rekursive Methoden und rekursive Datenstrukturen haben viele Gemeinsamkeiten mit vollständiger Induktion. Diese mathematische Beweismethode beruht auf den natürlichen Zahlen. Wir beweisen, dass eine Aussage für die Zahl 1 (oder 0) gilt. Wenn diese Aussage unter der Annahme, dass sie für eine beliebige natürliche Zahl n gilt, auch für n + 1 gilt, dann gilt sie tatsächlich für jede natürliche Zahl n.
Induktionsanfang und Induktionsschritt sind auch bei der Programmierung zu überprüfen.

Wie kann man den Aufwand eines Algorithmus abschätzen?
Wie groß der konstante Aufwand ist, können wir durch Messen der Laufzeit bzw. des Speicherverbrauchs feststellen.
Man vernachlässigt Details wie der Hardware, der Programmiersprache, der Implementierung oder Datenmenge und berechnet ungefähr, wie viele Operationen pro Datenelement durchgeführt werden, also wie die Anzahl der Operationen mit der Anzahl an Datenelementen in Verbindung steht.

Wofür stehen *O*(1), *O*(log(*n*)), *O*(*n*), *O*(*n*(log(*n*)), *O*(*n*^2) und *O*(2^*n*)? Wie wirkt sich eine Verdopplung oder Verhundertfachung von *n* aus?
O = Ordnung: gibt den Aufwand bei n Elementen an
O(1): konstant
O(log(n)): logarithmisch
O(n): linear
O(n(log n)): super-linear
O(n^2): quadratisch
O(2^n): exponentiell
Auswirkung: kommt darauf an, mit wievielen Anweisungen eine Operation implementiert wurde, aber die grobe Aufwandabschätzung bleibt gleich (100 * n ist immer noch linear, (100*n)² noch immer quadratisch, etc.) Je nach Art des Aufwands kann für höhere Werte von n allerdings ein bedeutend höherer Aufwand beim Abarbeiten des Programms entstehen.

Wieso kann man konstante Faktoren bei der Aufwandsabschätzung einfach ignorieren?
Weil diese bei wachsenden n kaum mehr eine Auswirkung auf die Größenordnung der Gesamtkosten haben. Sie werden von jeder anderen Art von Kosten “dominiert”.

Wie hoch ist der Aufwand für das Einfügen bzw. das Suchen in

der verketteten Liste, im binären Baum sowie in der Hashtabelle im Durchschnitt und im schlechtesten Fall? Was ist der jeweils schlechteste Fall und wann tritt er ein?
Verkettete Liste: Suche: durchschnitt: O(n)/maximal: O(n); Einfügen: O(1), schlechtester Fall: gesamte Liste muss durchgegangen werden
Binärer Baum: Suche: durchschnitt: O(log(n))/maximal :O(n), schlechtester Fall, wenn Baum zu Liste entartet
Hashtabelle: Suche :durchschnitt: fast konstant/maximal: O(n); Einfügen im besten fall O(1), im schlechtesten Fall O(n), schlechtester Fall, wenn alle Hashwerte gleich sind.

Wie funktionieren Bubblesort, Mergesort und Quicksort? Wie hoch ist der Aufwand dafür im Durchschnitt und im schlechtesten Fall?
Bubblesort: durchlaufe Liste und vertausche zwei benachbarte Elemente, wenn Reihenfolge nicht stimmt, bis der Durchlauf keine Änderung mehr bewirkt; durchschnitt: $O(n^2)$/maximal: $O(n^2)$
Mergesort: teile Array solange bis nurmehr einzelne Elemente, füge Elemente in richtiger Reihenfolge wieder zusammen (Ebene für Ebene);durchschnitt: O(n(log n))/ maximal: O(n(log n))
Quicksort: wähle Pivotelement (teilt die Menge), links davon nur Elemente kleiner als Pivotelement, rechts davon größere Elemente, rekursiver Aufruf bis alle Elemente in der richtigen Reihenfolge; durchschnitt: O(n(log n))/maximal:$O(n^2)$

Was ist eine binäre Suche?
Starte bei Wurzel, ist das gesuchte Element kleiner, setze Suche bei linkem Knoten fort, sonst beim rechten, usw...

Was unterscheidet generische von nicht-generischen Klassen?
Generisch: Der Programmcode wird unabhängig von einem bestimmten Typ zu halten und wird somit verallgemeinert.

Generische Typen in den generischen Klassen werden durch den entsprechend angegebenen Typ ersetzt. Für jeden Typ wird eine Kopie der generischen Klasse erstellt. Generische Klassen müssen erst vom Compiler (?) erstellt werden, wohingegen nicht-generischWeil Iteratoren häufig eingesetzt werden, gibt es in Java eine spezielle Syntax für for-Schleifen über Containern ähnlich den for-Schleifen über Arrays: In „for (T v: c) . . . “ läuft die Schleifenvariable v vom Typ T über alle Elemente des Containers c, wobei c eine Instanz von Iterable<T> sein muss. Die Elemente werden über den Iterator ausge- lesen und in der vom Iterator vorgegebenen Reihenfolge abgearbeitet

Klassen as-is vom Compiler übernommen werden.

Was unterscheidet einen Typ von einem Typparameter? Kann man Typen und Typparameter gleich verwenden?
Typ: konkreter Typ, ersetzt zur Laufzeit den Typparameter
Typparameter: selbst kein Typ, wird durch konkreten Typ ersetzt

Wie kann man primitive Typen wie int als Elementtypen in generischen Containern verwenden?
Man verwendet den dazugehörigen Referenztyp “Integer” (von boolean: “Boolean”; double: “Double”...) (S.276)
Die Umwandlung in den dazugehörigen Referenztyp wird dabei bei Bedarf automatisch durchgeführt.

Wozu dienen Schranken bei gebundener Generizität?
Bei der Deklaration eines Typparameters gibt man einen Typ als Schranke an. Der durch die Schranke gebundene Typparameter kann nur durch einen Untertyp der Schranke ersetzt werden.

Welchen speziellen Zweck hat rekursive gebundene Generizität?
Erlaubt uns Typen von formalen Parametern so einzuschränken, dass sie gleich den Klassen sind, in denen die Methoden stehen.

Inwiefern ähneln sich Untertypbeziehungen und Generizität? Wodurch unterscheiden sie sich in ihrer Anwendbarkeit?

Was sind und warum verwendet man Iteratoren?
Ermöglicht das einfache Nacheinander-auslesen von Elementen eines Containers.

Welche Schwierigkeiten treten bei der Implementierung von Iteratoren im Zusammenhang mit Rekursion häufig auf? Wie löst man sie?
Rekursion ist nicht möglich, weil das Durchwandern nach jedem gefundenen Element abgebrochen werden muss und erst wieder beim nächsten Aufruf von next fortgesetzt werden kann. (S 286)
Programmiersprachen verwenden einen für den Programmierer nicht sichtbaren Stack, um darin die Variablen geschachtelter Methodenaufrufe zu speichern, auch die von rekursiven Aufrufen.

Durch welches spezielle Sprachkonstrukt unterstützt Java die Verwendung von Iteratoren?

innere Klassen

Wodurch wird die Verwendung fertiger Programmteile erschwert? Wie kann man den Ursachen dafür begegnen?
Mangelndes Vertrauen
Mangelndes Wissen
Unterschiedliche Modelle
“Ich kann es besser”-Denkweise

Mangelndem Vertrauen und Wissen kann man vorbeugen, indem man sich genauere Informationen über die Programmteile verschafft und man gegebenenfalls die Teile auch selbst testet.

Welche Vor- und Nachteile hat die Top Down Strategie gegenüber der Bottom Up Strategie? Wie lassen sich diese beiden Strategien miteinander kombinieren?
Bei Top Down fällt es leichter, den großen Überblick zu bewahren (so beginnt man z.B. bei der main-Methode und arbeitet davon nach unten). Der Nachteil dieser Strategie ist jedoch der, dass dieser Ansatz die Objektorientierung zumeist nicht so gut umsetzt wie Bottom Up. Die Kombination würde z.B. so aussehen, dass man Top Down für die grobe Struktur wie z.B. Klassenhierarchie verwendet und Bottom Up dann für die Ausprogrammierung einzelner Klassen.

Für welche Aufgaben bietet sich die schrittweise Verfeinerung an?
große Aufgaben, große Programme

Mit welchen Teilaufgaben sollte man bei schrittweiser Verfeinerung beginnen? Warum ist das so?
Beginnen sollte man mit jenen, die am schwierigsten zu lösen sind und auf die es am ehesten ankommt.#

Welche Vorteile und Schwierigkeiten können sich aus der schrittweisen Verfeinerung ergeben?
Vorteile: gute Rückmeldung über die bisherige Qualität des Programms, was wiederum nützlich für die Weiterentwicklung ist (Erfahrungen gehen in weitere Entwicklung ein); Flexibilität bei Anforderungsänderungen
Nachteile: Bei einer Erweiterung des Programms kann es zu Problemen kommen, da man feststellt, dass Datenstrukturen oder die gewählte Faktorisierung für den neuen Programmteil nicht oder nur unzureichend geeignet sind.