# Kapitel 1: (60 Fragen)

### 1. Was versteht man unter einem Programmierparadigma?

**Paradigma**: bestimmte Denkweise, Art der Weltanschauung

**Programmierparadigma**: Ein Weg Programmiersprachen nach ihren Eigenschaften und Stil, der beim Programmieren praktiziert wird einzuteilen.

Diese Eigenschaften sind:

* Dahinterliegendes Berechnungsmodell
* Struktur von Kontrollfluss (Programmablauf) und Datenfluss
* Problemlösung durch Aufteilung in überschaubareren Teilen

Beliebte Programmierparadigmen:

**Imperativ** auf Maschinenbefehlen aufbauend

* + - *Prozedurales Paradigma* (Programme in kleinere Teilaufgaben aufspalten. Die entstehenden Teilprogramme werden Prozeduren genannt. Praktisch in allen imperativen Sprachen.)
    - *Objektorientiertes Paradigma* (Programmierung mit abstrakten Datentypen, behandelt Daten und Funktionen als eine Einheit)

**deklarativ** auf formalen Modellen beruhend (man deklariert das erwünschte Resultat)

* + - *Funktionales Paradigma* (Funktionen nicht nur angewendet sondern auch wie Daten miteinander verknüpft, als Parameter verwendet und als Funktionsergebnisse)
    - *Logikorientiertes Paradigma* (logische Aussagen die verbunden werden)

### 2. Wozu dient ein Berechnungsmodell?

Ein Berechnungsmodell ist ein Kalkül.

Sie ist ein formales System mit eigenen Axiomen (Regeln).

Dient als mathematische Berechnungsgrundlage (formale Grundlage) für jede Programmiersprache.

Kalkül muss:

* + - in sich konsistent und vollständig sein sein (widerspruchsfrei)

Zusätzlich erforderlich:

* + - Turing-vollständig sein (alles berechnen können was mit einer Turingmaschine berechenbar ist)
    - Praktische Umsetzbarkeit zu Programmiersprache

### 3. Welche Berechnungsmodelle werden in Programmierparadigmen verwendet, und welche charakteristischen Eigenschaften haben sie?

Die Berechnungsmodelle von Programmiersprachen werden aus diesen Formalismen (Konzepten) entwickelt wenn es sinnvoll ist.

Funktionen

* primitiv-rekursive Funktionen:

Gehen von einer vorgegebenen Menge an einfachen Funktionen aus

Komposition von anderen Funktionen oder sich selbst (Rekursion) bilden neue Funktionen

Nicht alleine turing-vollständig

* Turing-Vollständigkeit erreicht durch μ-rekursive Funktionen und λ-Kalkül

Prädikatenlogik

* Mathematisches Werkzeug
* Heutzutage häufig bei Abfragesprachen von relationalen Datenbanken.
* Horn Klauseln und halbautomatisierte Beweise

Constraint-Programmierung

* Das Problem wird deklarativ mittels Einschränkungen beschrieben
* Überall vereinbar außer mit logikorientierter Programmierung

Temporale Logik und Petri-Netze

* Temporale Logik: logische Ausdrücke mit zeitlichen Abhängigkeiten (Aussage gilt vor oder nach Ereignis)
* Petri-Netze: visualisieren zeitliche Abhängigkeiten
* Lassen sich ineinander umwandeln
* Sinnvoll für Nebenläufigkeit

Freie Algebren

* stark vereinfacht formulierte Algebren
* ermöglichen Erzeugung von allen beliebigen Strukturen, auch Datenstrukturen
* „universelle Algebra“, Axiome selbst aufstellen

Prozesskalküle

* Prozesse vergleichbar mit Threads in nebenläufigen Systemen
* CSP (Communicating sequential processes), Operationen nur zum senden und Empfangen von Daten
* π-Kalkül für reaktive Systeme und λ-Kalkül

Automaten

* Automatentheorie, beschreiben nur Syntax, sind aber turing-vollständig
* Visualisierung

WHILE, GOTO usw

* Typische Sprachkonstrukte die sich über die Zeit entwickelt haben mit einfachen Operationen aus Assembler
* PRAM (Parallel Random Access Memory) – Sprachen erweitern dadurch, dass mehrere Operationen gleichzeitig auf unterschiedlichen Speicherzellen zugreifen können
* Stark verbunden mit imperativer Programmierung

### 4. Welche Eigenschaften von Berechnungsmodellen sind für deren Erfolg häufig (mit)bestimmend?

Berechnungsmodelle müssen sich als Grundlage für eine Programmiersprache innerhalb eines Programmierparadigmas eigenen.

Sie müssen sich dazu eignen praktische Aufgaben zu lösen.

Erfolg eines Programmierparadigmas beeinflusst durch Programmierwerkzeuge und Syntax und Semantik-Details und nicht das Berechnungsmodell.

Einflussfaktoren:

Kombinierbarkeit und Skalierbarkeit

* Bestehende Programmteile sollen sich möglichst einfach zu größeren Einheiten kombinieren lassen.

*Gut:* Komposition von Funktionen zur Beschreibung komplexerer Funktionen

*Gut:* Objektorientierte Programmierung

*Schlecht:* Kombination von Automaten (wird zu komplexer Struktur)

Konsistenz

* Ein Formalismus reicht nicht
* Alle Formalismen sollen gut zusammenpassen die für eine Sprache gewählt werden. Formalismen sind von Haus aus oft miteinander inkompatibel, deshalb wählt man eine möglichst kleine Anzahl.

Abstraktion

* Höhere Programmiersprachen sollen meistens nicht von Hardware oder (Betriebs-)System-Details abhängen, sondern möglichst *portabel sein* und auf möglichst vielen Systemen laufen können.
* Es gibt verschiedene Abstraktionsarten und Abstraktionsgrade

Systemnähe

* Paradigma ist Systemnahe wenn es Hardware Aspekte direkt anspricht, Befehle auf Systemebene möglich sind.
* ↑Systemnähe bedeutet: ↑Effizienz, ↓Portabilität, ↓Sicherheit da direkt auf das System zugegriffen werden kann

Unterstützung

* Von Community: Bibliotheken, Programmiersprachen, IDEs, Tools usw.
* Geld von Investoren, einflussreichen Unternehmen

Beharrungsvermögen

* Änderungen passieren langsam. Für einen Wechsel braucht es sehr überzeugende Gründe, wie etwa eine *Killerapplikation*, also einer erfolgreichen Software, die den Erfolg des Paradigmas deutlich aufzeigt.

Wichtige Werkzeuge in der Softwareentwicklung sind Compiler und Interpreter.

JIT (Just-In-Time) Übersetzung

* Portabilität und einfache Bedienbarkeit von Interpretern mit Effizienz von Compilern.
* Hat erst Durchsetzung mancher Sprachen ermöglicht

### 5. Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen? Wie äußert sich dieses Spannungsfeld?

Spannungsfeld: Neue praktische Erfahrungen vs. Widersprüchliche Forderungen an Programmiersprachen innerhalb verschiedener Paradigmen.

Es ist nicht möglich alle diese Forderungen gleichzeitig und im vollem Umfang zu erfüllen:

1. Flexibilität / Ausdruckskraft: Knappe Darstellungsform / Text die alle möglichen Programmabläufe darstellt (mit Flexibilität ist eigentlich Freiheit gemeint. Möglichst wenige Einschränkungen von Sprache)

* Eher alte dynamische Programmierung auf Kosten von 2

1. Lesbarkeit / Sicherheit: Inkonsistenzen, Absichten und Sicherheitslücken lassen sich leicht erkennen

* Eher alte statische Programmierung auf Kosten von 1

1. Einfachheit / Verständlichkeit

* Wird in jüngeren Sprachen für mehr 1,2 eher vernachlässigt obwohl immer damit beworben wird

### 6. Was ist die strukturierte Programmierung? Wozu dient sie?

**Prozedurale Programmierung (Paradigma):**

Die prozedurale Programmierung ergänzt das imperative Konzept aufeinander folgender Befehle: Algorithmus in überschaubare

Teile ( Je nach Sprache: Unterprogramm, Routine, Prozedur oder Funktion genannt) zerlegen.

Manche sehen es als Gegenstück zur OOP, weil keine Verkapselung von Prozeduren und Variablen.

**Strukturierte Programmierung (Paradigma):**

Erweitert das Paradigma der prozeduralen Programmierung.

Verlangt die Beschränkung auf genau drei Kontrollstrukturen:

* Sequenz (hintereinander auszuführende Programmanweisungen)
* Auswahl/Selektion (Verzweigung – if, switch)
* Wiederholung/Iteration (Schleifen – loop, rekursion)

**Sie dient dazu:**

* GOTO Anweisung vermeiden
* jede Kontrollstruktur einen eindeutigen Einstiegs- und Austiegspunkt
* mehr Lesbarkeit, aber geringere Flexibilität und Ausdruckskraft (aber Lesbarkeit so stark dass alle verringerte Flexibilität in Kauf genommen haben).

**Beschränkt sich nicht nur auf imperative Paradigmen:**

Auch funktionale und logikorientierte Programmierung nutzen eine Variante davon:

* Schleifen 🡪 Rekursion
* Hintereinanderausführung von Prozeduren 🡪 Komposition von Funktionen

### 7. Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?

**Seiteneffekt:**

Auch Wirkung genannt, bewirkt Veränderung eines Zustandes (Variablenwert).

Jeder Programmfortschritt wird über Seiteneffekte erzielt (z.B. Variablenzuweisungen, Ein- und Ausgaben, Zustandsänderungen).

**Versteckter Seiteneffekt:**

Gefährlich, versucht man zu vermeiden.  
 Beispiel:

f() sortiert ein Array und sammelt statistische Daten. Die sequentielle Ausführung f(x);f(y) wäre kein Problem,

aber f(x);f(x);f(y) würde Array doppelt sortieren und statistischen Daten fälschen.

**Umgang mit versteckten Seiteneffekten in verschiedenen Paradigmen:**

Deklarative Paradigmen:

*Referentielle Transparenz* aller Ausdrücke, durch die Seiteneffektfreiheit, denn es werden keine Variablen (Zustandslos) und keine Wertezuweisungen (Seiteneffektfrei) verwendet.

Einzige Ausnahme: Seiteneffekte erlaubt für Ein- und Ausgaben ganz oben in der Aufrufhierarchie (die erste Funktion die aufgerufen wurde) und Ausgabe auf Konsole.

Imperative Paradigmen:

Konkret in OOP: versteckte Seiteneffekte möglichst reduzieren

* korrekte und vollständige Dokumentation
* Man nimm prinzipiell an dass es Querverbindungen gibt und beschränkt Zustandsänderungen möglichst auf einzelne Objekte, damit sie möglichst überschaubar bleiben. 🡪 Man muss somit nicht den gesamten Programmzustand überblicken können (prozedual).
* Man versucht Zustandsänderungen (Variablenwert-Zuweisungen) möglichst nur auf einzelne Objekte zu beschränken.
* Primitive Operationen (addition, division,…) sind immer referentiell transparent.

**Querverbindung**

Aufrufhierarchie (auch mit Funktionen möglich) =/= Querverbindung („Quer“ zwischen Objekten)

Eine Querverbindung ist eine Zustandsänderung ausgelöst in einem anderen Objekt.

Bei vielen Objekten schwer zu überblicken, deshalb große Gefahr für versteckte Seiteneffekte.

### 8. Was bedeutet referentielle Transparenz, und wo findet man referentielle Transparenz?

**Referentielle Transparenz:**

Wenn der Ausdruck durch seinen Wert ersetzt werden kann, ohne die Semantik des Programms dadurch zu ändern.

* 3 + 4 darf mit 7 vertauscht werden

Wenn der Wert des Ausdrucks überall im Programm gleich (hängt von Umgebung ab) und nicht vom Zeitpunkt oder einer bestimmten Reihenfolge der Auswertung. Das ist nur möglich wenn f seiteneffektfrei ist und nicht von Variablen abhängt deren Werte sich im Laufe der Zeit ändern könnten.

* f(x) + f(x) darf mit 2 \* f(x) vertauscht werden (obwohl Bedeutung der Funktion unbekannt ist)

Funktionen in imperativen Paradigmen können referentiell Transparent sein, müssen es aber nicht.

### 9. Wieso passt referentielle Transparenz nicht gut mit Ein- und Ausgabe zusammen, und wie kann man das Dilemma lösen?

Problem: Wenn eine Ausdruck keine Zustände ändern kann, dann kann man nicht herausfinden was es berechnet hat.

Ein- und Ausgaben sind Seiteneffekte, die vollständige referentielle Transparenz unmöglich machen.

Sie werden benötigt, aber sind nicht erlaubt.

**Deklarative Sprachen**

In funktionalen Sprachen erlaubt man Ein- und Ausgaben nur gut sichtbar ganz oben in der Aufrufhierarchie (für die erste aufgerufene Funktion die das End-Ergebnis zurückliefert) und verbannt sie aus allen anderen Funktionen.

Ausgabe auf Konsole wird nicht als Seiteneffekt gezählt.

**Objektorientierte Programmierung**

Ein und Ausgabe von Funktionen sind nicht referentiell transparent.

### 10. Welchen Zusammenhang gibt es zwischen Seiteneffekten und der objektorientierten Programmierung?

Siehe Frage 8)

### 11. Was sind First-Class-Entities? Welche Gründe sprechen für deren Verwendung, welche dagegen?

**First-Class-Entities**

* In OOP sind es die Objekte
* In funktionalen Sprachen sind es Funktionen

Uneingeschränkte Verwendung (Können wie normale Daten verwendet werden):

* Zur Laufzeit erzeugt werden
* als als Eingabeparameter übergeben und Ausgabe übernommen werden
* in Variablen abgelegt werden

Bringen viele Sonderfälle mit sich die berücksichtigt werden müssen, großer Aufwand bei Umsetzung, deshalb geringe Anzahl an FCEs pro Sprache, Aufwand muss sich rentieren:

Beispiel in funktionalen Sprachen: Funktionen nur sinnvoll als FCEs wenn Funktionen höherer Ordnung möglich.

### 12. Was haben Funktionen höherer Ordnung mit einem applikativen Programmierstil zu tun?

In funktionalen Sprachen sind Funktionen die First-Class-Entities:

Das ermöglich die Existenz von Funktionen höherer Ordnung mit Funktionen als Eingangsparameter möglich

**Applikative Programmierung (Paradigma)**

Nur in Sprachen möglich, in denen Funktionen höherer Ordnung möglich sind.

Häufige Anwendung von Funktionen höherer Ordnung im Programm.

Skriptum: Ein häufiger Gebrauch von Funktionen höherer Ordnung führt zu dem Paradigma, der **applikativen Programmierung**, bei der man quasi Schablonen von Programmteilen schreibt, die dann durch Übergabe von Funktionen (zum Füllen der Lücken in den Schablonen) ausführbar werden.

Man versucht Methoden / Funktionen höherer Ordnung auch in OOP einzuführen ohne, dass sie referentiell transparent sind. Siehe Streams in Java.

### 13. Welche Modularisierungseinheiten gibt es, was sind ihre charakteristischen Eigenschaften, und wodurch unterscheiden sie sich?

**Modularisierungseinheiten**

In objektorientierten Sprachen kann es unterschiedliche Modularisierungseinheiten geben.

Wir zerlegen das Programm in einzelne Modularisierungseinheiten. Geringe Abhängigkeit voneinander, leicht gegen einander austauschbar.

Das bewirkt: Programmorganisation, Skalierbarkeit, höchste Flexibilität und Wartbarkeit über einen langen Zeitraum, unabhängiges Arbeiten von Entwicklern

**Module**

* *Übersetzungseinheit* (Einheit, die Compiler in einem Stück abarbeitet)
* Getrennte Übersetzung
* Schnittstellen bestimmen Übersetzungs-Reihenfolge: Modul B *importiert* Modul A (kann Modul A aufrufen) -> Modul A muss vor B übersetzt werden. -> Können sich nicht gegenseitig importieren, deshalb müssen Abhängigkeiten zyklenfrei sein.
* Erst mit *Linker* zur Laufzeit zu ausführbarem Programm verbunden.
* Insgesamt schnellere Übersetzungs-Zeit, da bei Änderungen nicht alle Module neu übersetzt werden müssen sondern nur welche die voneinander abhängig sind.
* In Java und den meisten Sprachen: Klassen und Interfaces mit allen enthaltenen statischen Variablen und statischen Methoden eigentlich Module.
* Datenabstraktion wie bei Objekten möglich: Encapsulation und Data-Hiding:  
  Enthält alle Deklarationen / Definitionen von zusammengehörenden Variablen und Methoden.

**Modulschnittstellen**

* Beinhalten Implementierung von Klassen / Interfaces aus Modulsicht. Daraus werden Abhängigkeiten (Inhalte, die von außen zugreifbar sind) zwischen den eigentlichen dahinterliegenden Modulen hergeleitet.
* Wichtig: module Dateien in Java sind keine Module sondern Modulschnittstellen

*Exportiert*:

Von anderen Modulen zugreifbar, Änderungen wirken sich auf alle die das Geänderte importieren.

Es entstehen neue getrennte Namensräume (aus dem eigenen und aus dem importierten Modul)

Namen im eigenen Modul-Namensraum müssen eindeutig sein.

Da nicht wie bei Objekten Instanzen gebildet werden können weist jeder eindeutige Name eines Moduls auf ein eindeutiges Verhalten hin.

Manchmal entstehen *Namensraum-Konflikte*: während Import kann man durch Umbenennung oder Qualifikation lösen (Überschreibung von bestehenden Namen)

*Privat*:  
 Nur vom eigenen Modul aus zugreifbar, kann vom Compiler optimiert werden.

**Objekt**

* Keine Übersetzungseinheiten
* erst zur Laufzeit erzeugt
* kapseln Variablen und Methoden zu Einheiten, schützen private Inhalte (Encapsulation und Data-Hiding (private Inhalte verstecken), gemeinsam Datenabstraktion)
* können einander zyklisch referenzieren
* Anders als Module sind sie immer First-Class-Entities
* haben Identität, Zustand und Verhalten (genauso wie Module)
* mehrere Instanzen mit selbem Namen/Verhalten möglich, deshalb differenziert man zwischen Identität und Gleichheit:

*Identisch*: Gleiches Objekt, gleiche Identität

*Gleich*: Gleicher Zustand, gleiches Verhalten, auch wenn nicht identisch (Kopie)

**Klasse**

* ein Modul und damit eine Übersetzungseinheit
* Dienen als Vorlage / Bauplan zur Instanzierung von Objekten

Der Begriff „Klasse“ kommt von der Klassifizierung anhand des Verhaltens, da alle Objekte der selben Klasse sich gleich verhalten. Deshalb sind Java Interfaces auch in diesem Sinn Klassen.

Gleiche Klasse -> Gleiches Verhalten

* Klasse selbst hat eigentlich keine statischen Variablen und Methoden weil sie ausschließlich als Vorlage zum instanzieren eines Objektes dient. -> Die statischen Variablen und Zustände sind Inhalte des darüberstehenden Moduls.
* Zyklische Abhängigkeiten zwischen Klassen verboten

*Klassenableitung*: Vererbung oder Ersetzbarkeit, dadurch kann ein Objekt mehrere Typen haben.

**Komponente (Metadaten über dependencies, nicht in Java)**

* Aus einem Modul wird durch *Parametrisierung* einer Komponente: Zuerst Festlegung der zu importierenden Quellen zur Übersetzungszeit, danach Bestimmung von deren Herkunft zur Laufzeit. -> Deshalb zyklische Abhängigkeiten erlaubt.
* Übersetzungseinheit, eigenständiges Stück Software, dass in ein Programm eingebunden wird
* Für sich alleine nicht lauffähig, nicht unabhängig, da sie die Existenz anderer Komponenten voraussetzt und deren Dienste in Anspruch nimmt. Komponenten interagieren miteinander und sind nicht.
* Verschachtelung möglich: Module in Komponenten oder Komponente in Modulen

**Namespace / Namensraum (in Java: Package)**

* Jede Modularisierungseinheit bildet ein eigenes Namensraum. Die Namen müssen in diesem Namensraum eindeutig sein. In anderen Einheiten können die Namen aber gleich sein.

Der Namensraum als Modularisierungseinheit dient aber dazu andere Modularisierungseinheiten global zu verwalten. Innerhalb eines Namensraums sind Namen von Modulen, Klassen, Komponenten eindeutig.

* Hilfreich gegen Namenskonflikte.
* Ohne globale Namensraumverwaltung müssen alle konkreten Dateipfade benötigter Modularisierungseinheiten angeführt werden – flexibel, aber unsicher.
* Namensräume fassen Modularisierungseinheiten zu einer Einheit zusammen, hierarchisch organisiert (wie Package in Java oder URL) ohne getrennte Übersetzbarkeit zu zerstören

Beispiel: a.b.C bezeichnet die Klasse C im Namensraum b welcher im Namensraum a steht.

### 14. Welche Bedeutung haben Schnittstellen für Modularisierungseinheiten? Warum unterscheidet man zwischen von außen zugreifbaren und privaten Inhalten?

**Allgemein:** Schnittstellen trennen Außensicht von versteckten Implementierungsdetails (Abstraktion). Clients (andere Einheiten die auf Schnittstelle zugriefen) können sich auf Schnittstellenstruktur verlassen.

Keine Änderungen notwendig wenn Server (genutzte Einheit) Implementierung ändert.

**Schnittstelle von Modulen**

Definition siehe Frage 13)

* Compiler braucht Schnittstelleninformation für Übersetzungsreihenfolge
* um zu wissen welche Modulinhalte von außen zugreifbar und aufrufbar sind (exportiert) und welche innerhalb des Moduls gebraucht werden (privat)

**Schnittstelle von Klassen und Objekten**

Durch *Klassenableitung* bei Klassen: Vererbung oder Ersetzbarkeit, dadurch kann ein Objekt mehrere Typen (Schnittstellen) haben.

### 15. Was ist und wozu dient ein Namensraum?

Siehe Frage 13)

### 16. Warum können Module nicht zyklisch voneinander abhängen, Komponenten aber schon?

Siehe Frage 13)

**Module**

* Module können wegen der getrennten Übersetzung nicht zyklisch voneinander abhängen (sich gegenseitig importieren).
* Module importieren Inhalte ganz bestimmter, namentlich genannter Module.

**Komponenten**

* Bei Komponenten sind aber zyklische Abhängigkeiten möglich, da zuerst die zu importierende Komponente festgelegt wird und erst später bestimmt wird woher dieser herkommen soll.
* Komponenten importieren Inhalte von zur Übersetzungszeit nicht spezifizierten anderen Komponenten, die erst beim Einbinden in ein Programm zur Laufzeit bekannt werden.

### 17. Was versteht man unter Datenabstraktion, Kapselung und Data Hiding?

**Datenabstraktion:** Datenabstraktion = Encapsulation + Data Hiding

Reduziert einen bestimmten Datenkörper auf eine vereinfachte Repräsentation.

**Encapsulation:** Zustand (Variablen) und Verhalten (Methoden) zu einer Einheit kapseln

**Data Hiding:** Private Inhalte verstecken

### Entfernte Frage aus Pool: Wodurch unterscheiden sich Komponenten von Modulen?

Siehe Frage 16)

### 18. Warum und inwiefern ist die Einbindung von Komponenten komplizierter als die von Modulen?

Siehe Frage 16)

Während Module als voneinander unabhängige Einheiten gesehen werden können, interagieren die Komponenten untereinander. Außerdem wird bei Komponenten erst zur Laufzeit klar woher importiert wird.

### 19. Wie kann man globale Namen verwalten?

Für globale Namen (außerhalb von Packages) ist Entwickler Verantwortlich -> Eindeutige Namen wählen.

Ansonsten mit Namensräumen als Modularisierungseinheit die einzelnen Namensräume der Modularisierungseinheiten verwalten.

### 20. Was versteht man unter Parametrisierung? Wann kann das Befüllen von „Lücken“ durch welche Techniken erfolgen?

**Parametrisierung**

In Modularisierungseinheiten Lücken (Parameter) lassen, die man später füllt (einen konkreten Wert zuordnet).

* Kann Flexibilität erhöhen aber auch Laufzeit: bei Änderung einer Einheit müssen alle verlinkten Einheiten ebenso geändert werden.
* Kann auch Wartung erschweren.

**Dynamisches Befüllen zur Laufzeit**

Lücken zur Laufzeit befüllt.

Daten sind First-Class-Entities. Lücken sind Variablen.

Ermöglicht Dependency-Injection: Die Verantwortung für das Erzeugen und Initialisieren von Objekten wird an eine zentrale Stelle (z.B. eine Klasse) delegiert. Von der zentralen Stelle kann man die Abhängigkeiten zwischen den Objekten leichter überblicken und steuern.

**Konstruktor**

Beim Erzeugen eines Objekts wird ein Konstruktor mit formalen Parametern ausgeführt, der die Objektvariablen initialisiert. Nicht nur in OOP Sprachen.

* Objekt-Konstruktor

Wird bei jeder Erzeugung eines Objekts aus einer Klasse aufgerufen. Initialisiert Objektvariablen.

* Statischer Konstruktor

Wird nur ein einziges Mal bei Erstverwendung einer Klasse aufgerufen.

* Anonymer Konstuktoren

Wird nur ein einziges Mal aufgerufen.

Reihenfolge der Konstuktor-Aufrufe: statischer-, anonymer-, Objekt-Konstruktor.

**Initialisierungsmethode (Setter-Methode)**

Wenn Konstruktoren nicht verwendbar.

Beispiele: Objekte durch Kopieren erzeugt (Aufruf von Konstruktor danach nicht möglich) oder bestehendes Objekt muss zur Laufzeit befüllt werden muss bevor man es verwenden darf.

Initialisierung nach Erzeugung, nur in imperativen Paradigmen möglich.

**Zentrale Ablage**

Werte an zentraler Stellen (z.B globalen Variablen oder Konstanten) ablegen, von wo sie bei der Objekterzeugung oder Verwendung benutzt werden. Zugriff auch durch Vererbung möglich. Auch statisch einsetzbar.

**Statisches Befüllten zur Übersetzungszeit**

Lücken zur Übersetzungszeit gefüllt. Weil nicht alles eine First Class Entity ist und während der Laufzeit geschrieben werden kann. Problem: Nachträgliches Ändern erschwert.

**Generizität**

Alle Arten von Modularisierungseinheiten (außer Objekte) können generische Parameter haben die mit Generizität befüllt werden.Man unterscheidet generische Klassen und Methoden.

Meistens sind generische Parameter für Konzepte die keine FCEs sind, wie zB Typen – dann Typparameter genannt.

**Annotationen**

Annotationen beginnen mit einem @-Zeichen und sind optionale Parameter. Die Lücken die gefüllt werden beim Setzen selbst definert. Können Einfluss auf System-Werkzeuge wie Compiler oder Betriebssystem haben (zB kann Compiler Warnungen ausgeben) oder ignoriert werden. Auch dynamisch abfragbar mit Reflexion.

**Aspekte**

Aus AOP. Metaprogrammierung: Spezifiziert Punkte im Programm und was dort passieren soll (bestimmten Code ausführen, oder Aufruf durch anderen Aufruf ersetzen) vor der Übersetzung. Man fügt zu einem bestehenden Programm von außen neue Aspekte hinzu. (Die Lücken die gefüllt werden beim Setzen selbst definert.) Aspect- Weaver führt es aus.

Zeitliche Abfolge der Füllung:

(Übersetzungszeit ist die Zeit, die ein Compiler braucht den Sourcecode zu übersetzen.)

- Generizität: zur Übersetzungszeit

- Annotationen: Übersetzungszeit und / oder Laufzeit

- Aspekte: vor Übersetzung

### 21. Warum braucht man zur Parametrisierung in der Objekterzeugung neben Konstruktoren gelegentlich auch Initialisierungsmethoden?

Siehe Frage 20) Initialisierungsmethode

### 22. Welche Vor- und Nachteile hat die zentrale Ablage von Werten zum Zweck der Parametrisierung?

Vorteil gegenüber anderen dynamischen Parametrisierungs-Arten:  
Auch für statische Modularisierungseinheiten bereits beim Übersetzen verwendbar.

### 23. Was unterscheidet Generizität von den verschiedenen Formen der Parametrisierung zur Laufzeit?

Generizität ist eine Form der Parametrisierung zur Übersetzungszeit.

### 24. Was sind Annotationen und wozu kann man sie verwenden? Wodurch unterscheiden sie sich von Generizität?

*Gemeinsamkeiten:*

* Beide statische Parametrisierung (zur Übersetzungszeit)

*Unterschiede:*

* Die Löcher, die durch Annotationen befüllt werden, sind im Gegensatz zur Generizität nirgends im Programm festgelegt
* Annotation auch eine Art der Metaprogrammierung, Generizität ausschließlich eine Art der Parametrisierung

### 25. Was versteht man unter aspektorientierter Programmierung?

Siehe Frage 20) Aspekte

### 26. Wodurch unterscheidet sich Parametrisierung von der Ersetzbarkeit, und warum ist die Ersetzbarkeit von so zentraler Bedeutung?

Ersetzbarkeit nur in objektorientierten Sprachen.

**Anwendungsgebiete / Warum Ersetzbarkeit von so zentraler Bedeutung ist**

* Indirekte Code-Wiederverwendung, Typhierarchie mit verschiedenen Schnittstellen aufstellen
* Dynamisches Binden. (Wenn ein Objekt an einer Variable zugewiesen wird und der Objekttyp (spezifischster Typ) ein Untertyp des deklarierten Typs der Variable ist).
* Aufruf einer Methode mit einem Argument, dessen Typ ein Untertyp des Typs des entsprechenden formalen Parameters ist

­­­­­­**Unterschied zwischen Parametrisierung und Ersatzbarkeit:**

Ersatzbarkeit / War

* Die Untertypbeziehungen können bei dynamischer Typisierung zur Laufzeit abgefragt werden, aber sie müssen vor der Übersetzung festgelegt werden.

Parametrisierung

* Bei der statischen und dynamischen Parametrisierung kann man jederzeit auf die die Parameter zugreifen (wenn public).
* Code-Wiederverwendung durch Aufruf von bestehenden Methoden in Parameter (falls abstrakter Datentyp). Erhöht aber Objekt-Kopplung wenn versucht so zu faktorisieren, sollte man vermeiden.

Problem bei Parametrisierung:

* Änderung einer Modularisierungseinheit bedeutet meistens auch Änderungen an allen Stellen, wo sie als Wert zugewiesen wird. Erschwert dadurch das erweitern.
* Erst mit Ersetzbarkeit (ohne oder zusätzlich zu Parametrisierung) Modularisierungseinheiten ohne zu viel Aufwand nachträglich die Modularisierungseinheit änderbar.

### 27. Wann ist A durch B ersetzbar?

**Ersetzbarkeit**

Nur in objektorientierten Sprachen. Man unterscheidet zwischen nominalen und strukturellen Untertypbeziehungen.

Es muss die Erwartung erfüllt werden, dass eine Einheit die andere ersetzen kann.

Ersetzbarkeit ist gegeben wenn die Schnittstellen von beiden Einheiten das Gleiche beschreiben, jedoch darf der Untertyp in Schnittstelle mehr Details festlegen die in Obertyp noch offen sind.

Untertyp kann mehr Details festlegen als Obertyp (also Obertyp erweitern), muss aber jedenfalls alles bereitstellen, was vom Obertypen erwartet wird.

**Ersetzbarkeitsprinzip**

*Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.*

**Allgemeine Eigenschaften von Untertypbeziehungen**

Reflexivität Jede Klasse ist Untertyp von sich selbst

Transivität [ a -> b -> c ] bedeutet [ a -> c ]

### 28. Wodurch kann festgelegt sein, ob A durch B ersetzbar ist?

Es kann zB bei Klassenableitung zu ungeplanten Ersatzbarkeitsbeziehungen kommen. Deshalb ist es wichtig zu überprüfen ob die bestehenden Untertypbeziehungen richtig sind.

**Signatur**

Statisch prüfbar. Einfachste Form der Schnittstelle, bestimmt welche Inhalte der Modularisierungseinheit von außen zugreifbar sind: Namen von Inhalten, Typen von Parametern.

Problem: Bedeutung der Inhalte bleibt offen

Namen in der Signatur in einer Schnittstelle sagen nichts über die Struktur (Implementierungsdetails) aus, deshalb sind sie alleine nicht ausreichend.

Gefahr: gleich heißen aber führen Unterschiedliches aus.

**Abstraktion realer Welt**

Eine Abstraktion ist ein informeller Text, die zusätzlich zur Signatur die Modularisierungseinheit beschreibt.

Bei der *konzeptuellen Ersetzbarkeit* (reale Welt als Grundlage der Abstraktion), sind Zusicherungen besonders wichtig. Dieser Ansatz ist gut aber beruht nur auf Intuition, deshalb gefährlich

**Zusicherungen / Erwartungen**

Genaue Beschreibung der erlaubten Erwartungen an eine Modularisierungseinheit (*Design by Contract*).

**Überprüfbare Protokolle**

Formale Beschreibungen der Erwartungen, die bereits Compiler auf Konsistenz überprüfen kann wenn sie nicht zu kompliziert sind. Können Beziehung zwischen Client und Server (Client Server Protokolle) oder mehreren Einheiten gleichzeitig regeln.

### 29. Was ist die Signatur einer Modularisierungseinheit?

Methodensignatur ist die Schnittstelle einer einzelnen Funktion.

Die Signatur einer Modularisierungseinheit ist aber die gesamte Schnittstelle, die nach außen sichtbare *Struktur*. Beinhaltet Namen und Typen (deklarierter Typ bei Variablen und alle Parametertypen bei Methoden).

### 30. Wie verhält sich die Signatur einer Modularisierungseinheit zur Abstraktion, die durch diese Modularisierungseinheit gebildet wird?

Eine Abstraktion ist ein informeller Text, die zusätzlich zur Signatur die Modularisierungseinheit beschreibt.

Signatur (=struktureller Typ) allein nur beschränkt zur Abstraktion geeignet, da komplett unterschiedliche Modularisierungseinheiten (und unterschiedliche damit assoziierte abstrakte Konzepte) zufällig die gleiche Signatur haben können.

### 31. Was sind Zusicherungen, und welche Rolle spielen sie für Modularisierungseinheiten?

Beschreiben die erlaubten Erwartungen an (die Schnittstelle einer) Modularisierungseinheit.

Schnittstelle als Vertrag zwischen Modularisierungseinheit (Server) und Verwendern (Clients). Vertrag beschreibt Verantwortlichkeiten.

Zusicherungen werden als Teil des nominalen Typs betrachtet und in Java sind Zusicherungen nur durch Kommentare auszudrücken. Weil sie als Teil des Objekttyps gelten müssen sie die Bedingungen der Ersetzbarkeit erfüllen.

**Vorbedingungen – Pre Conditions**

Was darf sich Server von Client erwarten?

@param, „Welche Eigenschaften sollen die Eingangsparameter haben?“

Im Untertyp sind sie schwächer oder gleich

Bei Vererbung logische OR-Verknüpfung

**Nachbedingungen – Post Conditions**

Was darf sich Client von Server erwarten?

@result, „Was sagt das Ergebnis aus?“ Beschreibt Methodenergebnis / Änderungen / Eigenschaften des Objektzustands

Im Untertyp sind sie stärker oder gleich

Bei Vererbung logische AND-Verknüpfung

**Invarianten – Invariants**

Welche Eigenschaften sind in allen Programmzuständen erfüllt?

Gelten für alle Objektvariablen, Zuständigkeit des Servers.

Im Untertyp sind sie stärker oder gleich -> Wenn Variable von außen sichtbar dann gleich

Bei Vererbung logische AND-Verknüpfung (wenn nicht gleich)

**History-Constraints**

In welchen Aufrufreihenfolgen dürfen Clients mit Server interagieren?

Schränken die Entwicklung von Objekten oder Objektvariablen im Laufe der Zeit ein.

**Server-kontrolliert:** ähneln Invarianten (z.B. Wert einer Variable darf nur größer werden)

Im Untertyp sind sie stärker oder gleich -> Wenn Variable von außen sichtbar dann gleich

**Client-kontrolliert:** Reihenfolge der Methodenaufrufe einschränken. Z.B. Methode "Initialize" kann in jedem Objekt nur einmal aufgerufen werden.

Im Untertyp sind sie stärker oder gleich

### 32. Wann sind Typen miteinander konsistent, und was sind Typfehler?

Typen sind miteinander konsistent, wenn die Typen der Operanden mit der Operation zusammenpassen (Die Operation auf diesen Typen definiert ist), andernfalls tritt ein Typfehler auf.

Bei elementarten Typen:

* Typkonsistenz wenn Typ von Operand und Operation übereinstimmt
* Nur wenige Sprachen überprüfen Typkonsistenz nicht. zB Assembler, dann treten

### 33. Wie schränken Typen die Flexibilität ein, und warum verwendet man Typen trotzdem?

**Typsysteme**

Das Typsystem ist der Teil der Programmiersprache welcher die Typisierung (die Überprüfung) durchführt.

**Typisierung**

Ziel der Typisierung ist die Vermeidung von Typverletzungen / Erhaltung der Typkonsistenz in der Sprache.

Sprachen die ein Typsystem beinhalten nennt man typisiert.

Fast allen aktuellen Programmiersprachen sind typisiert um, Zuweisungsfehler zu vermeiden, Typkonsistenz zu sichern.

Nachteil:

* Einschränkung der Flexibilität weil nicht beliebige Werte und Ausdrücke verwendbar sondern nur solche, die vorgegebenen Typen entsprechen

Vorteil:

* Bessere Planbarkeit, Lesbarkeit, Zuverlässigkeit und Möglichkeit zu abstrahieren weil man weiß welche Typen zu erwarten sind
* Erstellung von Operationen die nur für Werte bestimmter Typen zugelassen sind

### 34. Welche Gründe sprechen für den Einsatz statischer Typprüfungen, welche dagegen?

**Einteilung der Typisierungsarten**

Strongly vs. Weakly / loosly stark vs. schwach

Static vs. Dynamic Kompilierzeit vs. Laufzeit

Manifest vs. Inferred explizit vs. implizit mit Typinferenz

Nominal vs. Structural nominal vs. strukturell

**Statische /** [**Dynamische Typisierung**](https://de.wikipedia.org/wiki/Statische_Typisierung)  
Typprüfungen können zur Übersetzungszeit oder zur Laufzeit vorgenommen werden.

Beeinfluss Zeitpunkt an dem Fehler entdeckt werden

Dynamische Typisierung: JavaScript, Python und Ruby

Statische Typisierung: Java (hat aber auch dynamische Typprüfungen), C#, Eiffel, Swift, Haskell, TypeScript

Statische Typisierung

Vorteile:

* Weniger Typfehler zur Laufzeit, verbesserte Zuverlässigkeit, Fehler zeigen sich früher
* Leichtere Lesbarkeit da Information von statischen Typen zuverlässig (wenn explizit deklariert)
* Typinferenz auch möglich
* weniger Fallunterscheidungen im Compiler da Typ früh bestimmt, verbessert Planbarkeit
* Mehr Informationen für [Optimierungen](https://de.wikipedia.org/wiki/Programmoptimierung) zur Übersetzungszeit

Nachteile:

* Man kann nicht alles statisch herleiten zb Array-Grenzen (aber manche Sprachen kommen ausschließlich mit statischer Typprüfung aus, zb Haskell). Schränkt Sprache aber ein und kann effektiv zu mehr Aufwand führen.
* Dynamische Sprachen möglicherweise sogar sicherer, weil diese besser getestet werden (bei Suche nach Typfehlern werden nebenbei auch andere Fehler erkannt)

Dynamische Typisierung

Vorteile:

* optionale Typdeklarationen möglich

### 35. Was versteht man unter Typinferenz? Welche Gründe sprechen für bzw. gegen den Einsatz?

**Explizite / Implizite Typisierung**  
Datentyp kann explizit genannt werden oder per [Typableitung](https://de.wikipedia.org/wiki/Typableitung) (*type inference*) ermittelt werden.

Von Typinferenz spricht man, wenn Typen im Quellcode nicht explizit hingeschrieben werden, aber der Compiler diese im Rahmen der statischen Typprüfung aus der Programmstruktur ableiten kann. Dann kann die Deklarierung der Typen optional sein

Implizite Typisierung / Typisierung

Vorteile:

* Typinferenz spart das Anschreiben von Typen.

Nachteile:

* Zur Verbesserung der Lesbarkeit kann und soll man Typen explizit anschreiben -> Explizite Typisierung verbessert Lesbarkeit.
* Typinferenz und Ersetzbarkeit durch Untertypen verträgt sich nicht (im selben Teil des Programms).

### 36. Zu welchen Zeitpunkten können Entscheidungen getroffen werden (Typen und Entscheidungsprozesse)?

Bedeutende Entscheidungen bezüglich der Programmstruktur und des Programmablaufs die Typen und Typendarstellung beeinflussen und die jeweiligen Zeitpunkte.

Typen verknüpfen zu unterschiedlichen Zeitpunkten vorliegende Informationen miteinander.

Entscheidungs-Zeitpunkte:

* Sprachdefinition
* In der Sprachimplementierung z.B. int als 32-Bit Zweierbitkomplement definiert
* Programme können daran nichts ändern.
* Erstellung der Übersetzungseinheiten
* Hier werden die meisten wichtigen Entscheidungen getroffen, auf die man beim Programmieren den meisten Einfluss hat – Flexibilität entscheidend
* Parametrisierung
* manche Entscheidungen erst durch Parametrisierung bei Einbindung vorhandener Module, Klassen, Komponenten getroffen.
* Unterschied, ob generischer Typparameter oder referentieller Typ (wie Object)

Bei primitiven Datentypen wird in dieser Stufe automatisch das Auto-Boxing / Auto-Unboxing durchgeführt.

Bei abstrakten Datentypen reserviert Compiler Platz für Referenz.

* Übersetzungszeit
* Entscheidungen und Optimierungen. Eher weniger Bedeutung, alles wichtige bereits im Programmcode festgelegt oder liegt erst zur Laufzeit vor.
* Laufzeit
* Zur Laufzeit kann man Initialisierungsphase (bei dynamischer Parametrisierung) von der eigentlichen Programmausführung unterscheiden.

Es zeigt folgendes:

* Je früher Entscheidungen getroffen werden, desto weniger ist zur Laufzeit zu tun und desto weniger Programmcode (Fallunterscheidungen) braucht man.

Vor allem statisch geprüfte Typen helfen dabei. Sie verlagern die Entscheidung nach vorne, später daher weniger Fallunterscheidungen (dynamische Typprüfungen) nötig.

* Ohne statisch geprüfte Typen wäre mehrfacher Aufwand nötig: Sogar wenn man weiß, dass die Variable eine ganze Zahl enthält, muss der Compiler eine beliebige Referenz annehmen und zur Laufzeit eine dynamische Typprüfung durchführen.
* Helfen getroffene Entscheidungen über gesamten folgenden Zeitraum konsistent zu halten
* Compiler reserviert je nach Typ einer Variable mehr oder weniger Speicherplatz (je nachdem, wie viel Speicher für diesen Typ nach Sprachdefinition vorgesehen) – in Java für jeden primitiven Datentyp bestimmte Anzahl an Bits, für Typparameter so viel wie für Referenz nötig
* Unsichere Entscheidungen werden eher nach hinten verschoben und zu einem Zeitpunkt getroffen, zu dem bereits viele andere damit zusammenhängende Entscheidungen getroffen wurden

### 37. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?

Siehe Frage 36)

### 38. Wie beeinflussen Typen die Planbarkeit weiterer Schritte?

Siehe Frage 36)

Je früher Entscheidungen getroffen werden, desto weniger ist zur Laufzeit zu tun und desto weniger Programmcode (Fallunterscheidungen) braucht man.

### 39. Was ist ein abstrakter Datentyp?

**Abstraktion**

Abstraktion bedeutet, dass der Typ einer Modularisierungseinheit gedanklich für ein Konzept aus der realen Welt steht.

Abstraktion der realen Welt als Grundlage für Ersetzbarkeit, nennt man die *konzeptuelle* *Ersetzbarkeit*.

Eine Abstraktion kann aber der informelle Text als Kommentar sein, der Konzept aus der realen Welt beschreibt.

**Datenabstraktion**

Ist Kapselung mit Data Hiding.

Vereinfacht die Repräsentation eines Datenkörpers. Dadurch abstrakter Datentyp als Schnittstelle einer Modularisierungseinheit.

**Encapsulation** Zustand (Variablen) und Verhalten (Methoden) zu einer Einheit kapseln

**Data Hiding** Trennung der Innenansicht von der Außenansicht einer Modularisierungseinheit. Private Inhalte verstecken.

### 40. Was unterscheidet strukturelle von nominalen Typen?

**Nominale / Strukturelle Typisierung**

Man denkt beim Programmieren hauptsächlich abstrakt in Konzepten, nur selten an Signaturen.

Daher verwenden Programmiersprachen zum Großteil nominale Typen statt struktureller Typen. (in Java, C#, C++, . . . )

Struktureller Typ

Die Signatur einer Modularisierungseinheit beinhaltet Namen und Typen der von der nach außen sichtbaren Struktur, der Schnittstelle.

Signatur der Modularisierungseinheit bestimmt den Typen und die Untertypen-Beziehung, unabhängig davon ob Untertypbeziehung explizit deklariert wurde oder nicht.

gleiche Signatur == gleicher struktureller Typ

Problem: zufällig die gleiche Signatur möglich, dadurch zufällig die gleichen strukturellen Typen

Lösung: jeder Modularisierungseinheit einen eigentlich nicht verwendeten Inhalt mitgeben, dessen Name das Konzept dahinter beschreibt.

Duck-Typing

Bei struktureller und dynamischer Typisierung.

Strukturelle Typen werden dynamisch bei Bedarf geprüft. Dynamische Sprachen wie Python, Objective-C, Smalltalk und Ruby betrachten Klassen als getrennt von Typen, sodass sich strukturelle Typen ergeben.

Nominaler Typ

Eindeutiger Name der Klasse, von der das Objekt erzeugt wurde bestimmt Typen.

Dadurch dass alle Objekte der selben Klasse, die gleiche Signatur haben entscheiden Name und Zusicherung ob

Untertypbeziehung vorhanden ist.

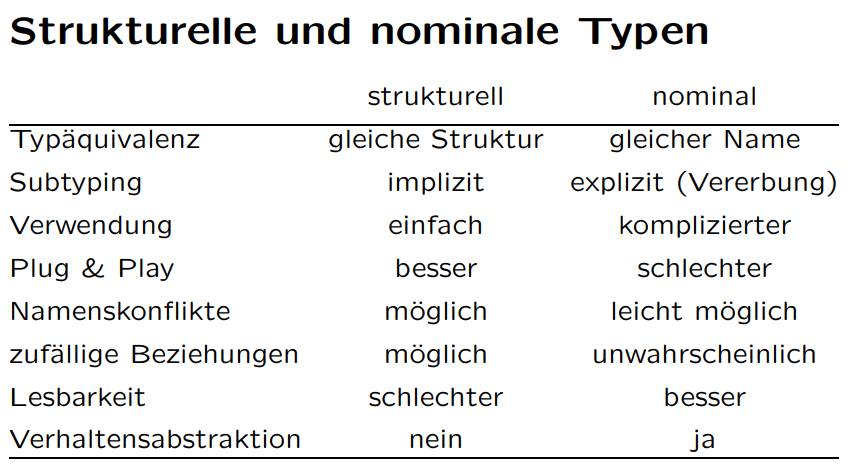
**Bestandteile eines nominalen Typs:**

1. Name der Klasse / Interface
2. Signatur (gesamte Schnittstelle)
3. Zusicherungen

Explizite Deklarationen in der Klasse bestimmen Untertypbeziehungen.

Nur wenn gleicher Name == gleicher nominaler Typ

(Zwei Objekte können die gleiche Struktur haben aber der Name entscheidet).



### 41. Warum verwenden wir in Programmiersprachen meist nominale Typen, in theoretischen Modellen aber hauptsächlich

### strukturelle?

Man denkt beim Programmieren hauptsächlich abstrakt in Konzepten, nur selten an Signaturen.

Daher verwenden Programmiersprachen zum Großteil nominale Typen statt struktureller Typen. (in Java, C#, C++, . . . )

Dadurch dass alle Objekte der selben Klasse, die gleiche Signatur haben und bei nominalen Typen es bereits im Voraus gesichert ist, dass die Signaturen übereinstimmen betrachtet man in theoretischen Modellen eher die strukturellen Typen um die Voraussetzungen zu definieren die die Signaturen erfüllen müssen.

### 42. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?

Untertypen werden durch das Ersetzbarkeitsprinzip definiert. Ohne Ersetzbarkeit gibt es keine Untertypen.

### 43. Warum kann ein Compiler ohne Unterstützung durch Programmierer(innen) nicht entscheiden, ob ein nominaler Typ

### Untertyp eines anderen nominalen Typs ist?

Basiert mehr auf Intuition und Abstraktion der echten Welt, man muss nicht immer die Signaturen im Kopf behalten.

Sie können nicht automatisch verglichen werden um Untertypbeziehungen aufzuzeigen, weil nominale Typen auf abstrakten Konzepten aufbauen und mit Zusicherungen und Kommentaren beschrieben werden.

Deklarieren Obertyp explizit, Untertyp muss vom Obertyp abgeleitetsein.

### 44. Erklären Sie Einschränkungen bei Untertypbeziehungen zusammen mit statischer Typprüfung.

**Einschränkungen durch statischer Typisierung**

Typen von Eingangsparametern dürfen in Untertypen nicht spezifischer (weiter Unten in der Typhierarchie) werden.

Dadurch wäre die Kontravarianz bei Schriebzugriff nicht mehr gegeben sondern man hätte Kovarianz.

* Siehe kovariantes Problem. Wird in der Praxis häufig benötigt, gibt aber keine Möglichkeit das statisch zu prüfen.

Dynamische Prüfungen aber möglich. (Man kann mit Typabfragen kovariantes Problem lösen)

Beispiel:

boolean compare(T x)

darf nicht überschrieben werden zu:

@Override

boolean compare(U x)

Aus demselben Grund sind auch Wertebereichseinschränkungen im Allgemeinen nicht statisch prüfbar; das bedeutet z.B., dass int nicht als Untertyp von long betrachtet werden kann, obwohl jede Zahl in int auch in long vorkommt.

### 45. In welchem Zusammenhang verwendet man Higher-Order-Subtyping und F-gebundene Generizität?

**Generizität**: Ausdrücke enthalten Parameter wie List<A>, Typparameter ist nun durch Integer zb ersetzbar generierte int liste

Nicht ausschließlich in objektorientierten Sprachen, sondern alle mit statischer Typprüfung damit man Einschränkungen auf Typparameter berücksichtigen kann.

* *F-gebundene Generizität*: nutzt Untertypbeziehungen zur Beschreibung von Einschränkungen (Java, C#)
* *Higher Order Subtyping / Matching*: Einschränkungen über Untertyp-ähnliche Beziehungen, die aber wegen Details keine Untertypbeziehungen sind. Direkterer Weg. Wäre eine andere Art der Typbeziehung die in Java ausgelassen wurde, weil es sonst zu Verwirrung führen würde (C++, Haskell)

### 46. Wie konstruiert man rekursive Datenstrukturen?

Beispielsweise für die Beschreibung unbeschränkter Datenstrukturen. Geht nur mittels *induktiven Konstruktionen* und *Mengenvereinigung*.

Benötigt eine *Fundiertheit* bei Konstruktion die nicht leer sein darf (zB Leafs bei Bäumen, ab denen dann keine weiteren Verzweigungen folgen). Diese bestimmt quasi den Abbruchpunkt der Rekursion.

Beispiel (Haskell):

data Lst = end | elem(Int,Lst)

In Java Fundiertheit in Sprachdefinition durch Wert „null“ gegeben (Nachteil: man muss immer mit null als Wert rechnen, auch bei nicht rekursiven Datenstrukturen)

### 47. Was versteht man unter Fundiertheit rekursiver Datenstrukturen? Welche Ansätze dazu kann man unterscheiden?

Siehe Frage 46)

### 48. Warum wird Typinferenz in objektorientierten Sprachen meist nur lokal beschränkt eingesetzt?

**Einschränkungen durch impliziter Typisierung**  
Typinferenz und Ersetzbarkeit durch Untertypen verträgt sich nicht (im selben Teil des Programms).

Aber verträgt sich gut mit Generizität.

### 49. Wie können statisch geprüfte Typen beliebige Eigenschaften von Werten propagieren?

**Propagieren von Eigenschaften mittels statischer Typisierung**

Eine Funktion kann nur aufgerufen werden, wenn der Typ des Arguments mit dem des formalen Parameters übereinstimmt. Dafür wird Information über das Argument (den Wert) an die aufgerufene Funktion propagiert.

Es propagiert auch Information von der aufgerufenen Funktion zum Aufrufer wenn man Ergebnistypen verwendet und oder einer Variable einen Wert zuweist.

Diese Arten des Propagierens von Information funktioniert nicht nur für Typen, sondern für alle statisch bekannten Eigenschaften.

### 50. Erklären Sie folgende Begriffe:

**Objekt**

Ein Objekt wird als Kapsel beschrieben, in der sich Variablen und Routinen befinden.

Zur Laufzeit erzeugt.

**Klasse**

Eine Klasse gibt die Struktur eines oder mehreren Objekten vor. Diese können mittels eines Konstruktors aus einer Klasse erzeugt werden. Die Klasse ist eine Art Schablone.

Spezifischste Schnittstelle eines Objekts ist dessen Klasse (Objekt „gehört“ gleichzeitig auch zu weiteren möglichen Oberklassen).

**Vererbung**

Ist eine Art des Ableitens einer Unterklasse von einer Oberklasse. Reine Vererbung erspart meist nur Schreibaufwand, da Änderungen nur an einer Stelle vorgenommen werden müssen.

In meisten verbreiteten objektorientierten Sprachen im Wesentlichen zwei Änderungsmöglichkeiten:

* Erweiterung (zusätzliche Variablen, Methoden und Konstruktoren)
* Überschreiben (Methoden der Oberklasse durch neue überschrieben, meist Methode aus der Oberklasse trotzdem noch von Unterklasse aus zugreifbar (Keyword super))

**Identität**

Jedes Objekt ist über eindeutige und unveränderliche Identität identifizier‐ und ansprechbar

**Zustand**

Setzt sich aus den momentanen Variablenbelegungen zusammen. Ist änderbar.

**Verhalten**

Reaktion eines Objekts beim Erhalten einer Nachricht. Ist abhängig von

1. Der aufgerufenen Methode(Routine)
2. Den übergebenen Parametern
3. Den momentan aktuellen Zustand des Objekts (Variablenwerte)

**Schnittstelle**

Einfach gesagt nur die Köpfe der Methoden, welche Aufschluss über Eingangs und Ausgangsparameter geben

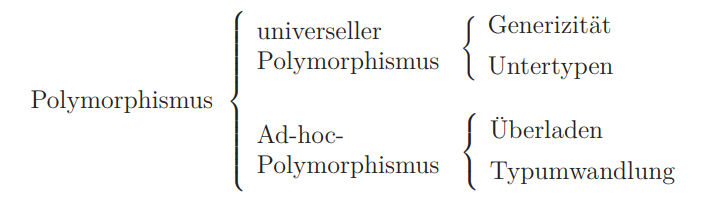
Spiegelt Außensicht auf Objekt wider, verdeckt Implementierungsdetails; Abstraktion – nur das, was für Zugriffe benötigt wird; entspricht in Sprachen mit statischer Typprüfung Typ eines Objekts

**deklarierter, statischer und dynamischer Typ**

Polymorphismus bedeutet, dass eine Variable gleichzeitig mehrere Typen haben darf.

1. OOP Sprachen sind polymorph
2. Konventionelle statisch typisierte Sprachen (C, Pascal) sind monomorph: Jede Variable oder Funktion hat einen eindeutigen Typ.

Es gibt veschiedene Arten von Polymorphismus:



In einer OOP Sprache hat eine Variable oder ein formaler Parameter gleichzeitig folgende Typen:

**Deklarierter Typ** (explizite) Anfangsdeklaration des Typs

**Dynamischer Typ** Der spezifischste Typ, des in der Variablen gespeicherten Werts. 🡪 Dynamisches Binden

(Compiler kennt dynamische Typen nicht)

**Statischer Typ** Wird vom Compiler (statisch) ermittelt und liegt irgendwo zwischen deklariertem und dynamischem Typ.

Abhängig von Compiler-Qualität. (Meistens für Optimierungen verwendet).

**Faktorisierung, Refaktorisierung**

Die Zerlegung eines Programms in Einheiten mit zusammengehörigen Eigenschaften - wie man ein Polynom in seine Bruchteile faktorisiert.

Ziel: Lesbarkeit, Wartbarkeit, vereinfachte Kontrollstruktur.

* Einsatz des Abstraktionsprinzips
* Gute Faktorisierung -> starker Klassenzusammenhalt -> gut abgeschlossene und somit leicht wiederverwendbare Klassen.

Refaktorisierung: Änderung einer bestehenden Zerlegung eines Programms in Klassen/Objekte; Funktionalität unverändert, nur Struktur angepasst

**Verantwortlichkeiten**

beschrieben werden durch:

* was ich weiß — Zustand der Objekte (Objektvariablen)
* was ich mache — Verhalten der Objekte (Methoden)
* wen ich kenne — sichtbare Objekte, Klassen, etc. (alle erreichbaren Klassen im *Scope*)

**Klassenzusammenhalt**

Grad der Beziehungen zwischen den Verantwortlichkeiten innerhalb der Klasse sollte hoch sein. Nicht messbar sondern nur intuitiv erfassbar.

Hoch wenn alle Variablen und Methoden

1. eng zusammenarbeiten
2. durch den Namen der Klasse gut beschrieben sind
3. durch das Entfernen einer einzigen die ganze Klasse beinflussen
4. Dadurch gut abgeschlossene und somit leicht wiederverwendbare Klassen

Test: Entfernen von Variablen und Methoden oder Umbenennung: Wenn Klasse nicht mehr funktioniert → guter Zusammenhalt

Ein hoher Klassen-Zusammenhalt bedeutet eine gute Zerlegung (Faktorisierung) des Programms in einzelne Klassen beziehungsweise Objekte.

**Objektkopplung**

Abhängigkeit der Objekte voneinander sollte niedrig sein.

Hoch wenn:

* Viele Methoden und Variablen nach außen sichtbar
* einzelne Objekte häufig miteinander kommunizieren, und dabei viele Parameter übergeben

Schwache Objekt-Kopplung deutet auf gute Kapselung hin, bei der Objekte voneinander so unabhängig wie möglich sind.

* Wenn Klassenzusammenhalt hoch, dann meistens Objektkopplung niedrig und vice versa.

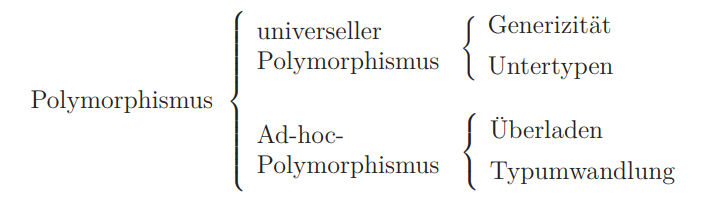
**Deshalb:**

Immer nur private als Modifier benutzen und möglichst getter und setter Methoden vermeiden und alles klassenintern übergeben.

**51. Welche Arten von Polymorphismus unterscheidet man? Welche davon sind in der objektorientierten Programmierung wichtig? Warum?**

**Polymorphismus**

Es gibt veschiedene Arten von Polymorphismus:



**Universelle Polymorphismen (spezifisch für OOP)**

Typen die zueinander in Beziehung stehen haben die gleiche Struktur. (bei Ad-hoc nicht)

**Generizität**: **„parametrischer Polymorphismus“**

gemeinsamer Code, der über Typparameter mehrere Typen haben kann

Ausdrücke enthalten Parameter wie List<A>, Typparameter ist nun durch Integer zb ersetzbar generierte int liste.

Nicht ausschließlich in objektorientierten Sprachen, sondern alle mit statischer Typprüfung damit man Einschränkungen auf Typparameter berücksichtigen kann.

* *F-gebundene Generizität*

nutzt Untertypbeziehungen zur Beschreibung von Einschränkungen (Java, C#)

* *Higher Order Subtyping / Matching*

Einschränkungen über Untertyp-ähnliche Beziehungen, die aber wegen Details keine Untertypbeziehungen sind. Direkterer Weg. Wäre eine andere Art der Typbeziehung die in Java ausgelassen wurde, weil es sonst zu Verwirrung führen würde (C++, Haskell)

**Untertypbeziehungen: „Enthaltender Polymorphismus**“

Gemeinsame Schnittstellen für unterschiedliche Objekte

In OOP am wichtigsten. Deshalb alles was mit Untertypen zu tun hat: meist *Objektorientierter Polymorphismus*, kurz: Polymorphismus. (Stichworte: deklarierter, dynamischer, statischer Typ, Vererbung.)

Ableitung von Klassen mit Subtyping bzw. Vererbung. Basiert auf Ersetzbarkeitsprinzip. Auszuführende Methoden werden erst bei der Programmausführung festgestellt (dynamisch).

**Ad-hoc Polymorphismen**

keine strukturellen Ähnlichkeiten

**Überladen**

Methoden haben nur Name gemeinsam, sonst nichts zwingend gleich

Routinen mit gleicher Funktionalität und Namen, aber verschiedenen (Anzahl von) Parametern.

Überladene Methoden müssen keine strukturellen Ähnlichkeiten besitzen, deshalb „ad hoc“.

**Typumwandlung**

Casts auf elementaren Typen (kaum Ähnlichkeiten in interner Darstellung)

Umwandlung eines Wertes in ein Argument welches die Routine erwartet.

Auch bei Casting / Typenumwandlung keine ähnlichen Strukturen vorausgesetzt.

### 52. Wann sind zwei gleiche Objekte identisch und wann sind zwei identische Objekte gleich?

*Identisch*: Wenn es sich um das identische Objekt, also das selbe Abbild im Speicher, handelt (wird mit == verglichen).Zwei durch verschiedene Variablen referenzierte Objekte sind identisch wenn es sich um dasselbe Objekt handelt.

*Gleich*: Wenn die Parameter gleich sind (vergleich mit equals() Methode).

Zwei Objekte sind gleich wenn sie denselben Zustand und dasselbe Verhalten haben, auch wenn sie nicht identisch sind. In diesem Fall ist ein Objekt eine Kopie des anderen.

### 53. Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?

Siehe Frage 39)

Datenabstraktion und Datenkapselung sind komplimentäre Konzepte. Erst Datenkapselung und Data-Hiding gemeinsam ergeben Datenabstraktion

### 54. Was besagt das Ersetzbarkeitsprinzip? (Häufige Prüfungsfrage!)

**Ersetzbarkeitsprinzip**

*Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.*

### 55. Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig (mehrere Gründe)?

Untertypen werden durch das Ersetzbarkeitsprinzip definiert. Ohne Ersetzbarkeit gibt es keine Untertypen.

**Anwendungsgebiete von Untertypbeziehungen**

1. Indirekte Code-Wiederverwendung, Typhierarchie mit verschiedenen Schnittstellen aufstellen
2. Möglichkeit zur praxistauglichen nachträglichen Änderung von Modularisierungseinheiten verbessert Wartbarkeit, Planbarkeit
3. Dynamisches Binden. (Wenn ein Objekt an einer Variable zugewiesen wird und der Objekttyp (spezifischster Typ) ein Untertyp des deklarierten Typs der Variable ist).
4. Aufruf einer Methode mit einem Argument, dessen Typ ein Untertyp des Typs des entsprechenden formalen Parameters ist

### 56. Wann und warum ist gute Wartbarkeit wichtig?

Da Wartungskosten ca. 70 % der Gesamtkosten ausmachen. Gute Wartbarkeit erspart viel Geld insbesondere, wenn Programm langen Lebenszyklus hat, lange verwendet wird. Langfristig Vorteile von robustem Programm am besten bemerkbar.

### 57. Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?

Der Klassenzusammenhang soll hoch sein & Die Objektkopplung soll schwach sein.

Durch gute Faktorisierung, Wahrscheinlichkeit geringer, dass bei Programmänderung auch die Zerlegung in Klassen und Objekte geändert werden muss.

### 58. Welche Arten von Software kann man wiederverwenden, und welche Rolle spielt jede davon in der Softwareentwicklung?

**Wiederverwendung**

Ziel: Wiederverwendung von …

* bewährtem Code wichtig um Aufwand zu ersparen
* Konzepte von Untertypen, Vererbung und Generizität wurden für Code Wiederverwendung eingesetzt
* Bibliotheken, Projektinterne- oder Programminterne Wiederverwendung.
* Erfahrung (Entwurfsmuster)
* Programme: werden oft genau für häufige (wieder)verwendung entwickelt
* Daten: oft überdauern Daten die Lebensdauer der Programme.
* Erfahrungen: können zwischen sehr unterschiedlichen Projekten ausgetauscht werden.

Code‐Wiederverwendung erfordert braucht große Investitionen in die Wiederverwendbarkeit aber es kann sich manchmal langfristig rentieren.

### 59. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?

verbessern Wartbarkeit/Modifizierbarkeit, ändern Struktur ohne Funktionalität zu beeinflussen

### 60. Wofür ist die objektorientierte Programmierung gut geeignet, und wofür ist sie nicht gut geeignet?

**Richtige Paradigmenwahl**

Paradigmenwahl beeinflusst Entwicklung:

**Objektorientierte Programmierung**

1. Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt
2. Wiederverwendung von Code mit Untertypen und Ersetzbarkeit
3. Datenabstraktion
4. Wartbarkeit in langem Einsatz-Zeitraum

**Prozedurale, funktionale und logikorientierte Programmierung**

1. Algorithmen (in OOP ungeeignet da aufwändig über mehrere Objekte verteilt -> Erhöht Entwicklungsaufwand und senkt Verständlichkeit)
2. Einmaliger Einsatz von Software
3. Schnelles Programmieren

# Kapitel 2: (22 Fragen)

### 1. In welchen Formen (mindestens zwei) kann man durch das Ersetzbarkeitsprinzip Wiederverwendung erzielen?

Es gibt zwei Arten von Code-Wiederverwendung, direkt und indirekt. Wobei was bei direkter Wiederverwendung mittels Vererbung es sich um eine reine Vererbungsbeziehung oder um eine Vererbung mit Untertypbeziehung handeln kann.

Untertypbeziehungen

* Indirekte Code-Wiederverwendung
* Untertypen können oft einen Großteil des Codes ihres Obertypen wieder verwenden (Obertyp erweitern)

Vererbung mit Untertypbeziehung (meistens von is-a Beziehungen)

* Direkte Code‐Wiederverwendung

### 2. Wann ist ein struktureller Typ Untertyp eines anderen strukturellen Typs? Welche Regeln müssen dabei erfüllt sein? Welche zusätzliche Bedingungen gelten für nominale Typen bzw. in Java? (Hinweis: Häufige Prüfungsfrage!)

**Nominale Typen und Ersetzbarkeit (Nominal Subtyping)**

Basiert mehr auf Intuition und Abstraktion der echten Welt, man muss nicht immer die Signaturen im Kopf behalten.

Sie können nicht automatisch verglichen werden um Untertypbeziehungen aufzuzeigen, weil nominale Typen auf abstrakten Konzepten aufbauen und mit Zusicherungen und Kommentaren beschrieben werden.

Deklarieren Obertyp explizit, Untertyp muss vom Obertyp abgeleitetsein.

Dadurch dass alle Objekte der selben Klasse, die gleiche Signatur haben entscheiden Name und Zusicherung ob

Untertypbeziehung vorhanden ist.

**Bestandteile eines nominalen Typs:**

1. Name der Klasse / Interface
2. Signatur (gesamte Schnittstelle)
3. Zusicherungen

**Strukturelle Typen und Ersetzbarkeit (Structural Subtyping)**

Signatur der Modularisierungseinheit bestimmt den Typen und die Untertypen-Beziehung.

Unabhängig davon ob Untertypbeziehung explizit deklariert wurde oder nicht.

**Ersetzbarkeitsregeln**

Dadurch dass alle Objekte der selben Klasse, die gleiche Signatur haben und bei nominalen Typen es bereits im Voraus gesichert ist, dass die Signaturen übereinstimmen betrachtet man in theoretischen Modellen eher die strukturellen Typen um die Voraussetzungen zu definieren die die Signaturen erfüllen müssen.

Nehmen wir an:

1. [T <- U] U Untertyp von T
2. (T (A)) Etwas in Obertyp T vom deklarierten Typ A
3. (U (B)) Etwas in Untertyp U vom deklarierten Typ B

Für jede Variable in T muss es eine in U geben wobei [ A <-> B ] (Invarianz)

Für jede Konstante in T muss es eine U geben wobei [ A <- B ] (Kovarianz)

Für jede Methode in T muss es eine in U geben mit gleichen Namen, gleicher Parameteranzahl, Exceptions und Methoden Ein- und Ausgaben. bei Parametern muss man zusätzlich unterscheiden ob Lese oder Schreibzugriff.

Verallgemeinert:

**[ A <- B ] Kovarianz – bei Lesezugriff**

Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp.

> Konstanten, Ergebnistypen, Ausgangsparameter, Ausnahmen

**[ A -> B ] Kontravarianz – bei Schreibzugriff**

Alles was im Typ A geschrieben werden kann muss auch im Typ B geschrieben werden können.Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp.

> Eingangsparameter

**[ A <-> B ] Invarianz – bei Lese und Schreibzugriff**

Der deklarierte Typ eines Elements im Untertyp ist äquivalent zum deklarierten Typ des entsprechenden Elements im Obertyp. (Wichtig: Alles was Invariant ist, ist auch Kovariant)

> Variable, Durchgangsparameter

**Einschränkungen in Java und ähnlichen Sprachen**

Java (benutzt nominale Typisierung) ist strenger als es eigentlich sein müsste: Ergebnistyp kovariant, alle anderen Typen invariant, damit unterscheidbar vom Überladen.

Auch wenn wir theoretisch kontravariante Eingangs-Parametertypen haben könnten gibt es sie nicht.

**Beispiel:**

Annahme: variante Parametertypen erlaubt und strukturelle Typen (nicht in Java)

class **T** {

public **T** method(**U** p) { ... }

}

class **U** {

public **U** method(**T** p) { ... }

public void foo() { ... }

}

Hier wurde korrekt abgeleitet:

1. Der Ergebnistyp ist kovariant verändert
2. Der Eingangsparametertyp ist kontravariant verändert

Wäre die Methode foo in U nicht vorhanden, dann wären U und T sogar äquivalent, da sie die gleiche Struktur hätten und die Unterklasse die Oberklasse nicht erweitern würde.

In Java würde (nach expliziter Deklaration der Untertypbeziehung) die Methode method nur überladen werden (da Eingangsparameter nicht kovariant).

### 3. Sind die in Punkt 2 angeschnittenen Bedingungen (sowie das, was Compiler prüfen können) hinreichend, damit das Ersetzbarkeitsprinzip erfüllt ist? Wenn nicht, was muss noch beachtet werden?

**Allgemeine Eigenschaften von Untertypbeziehungen**

Reflexivität Jede Klasse ist Untertyp von sich selbst

Transivität [ a -> b -> c ] bedeutet [ a -> c ]

Antisymmetrie wenn [ a -> b ] ist [ b -/-> a ]

**Überprüfung der Untertypenbeziehung ohne Compiler**

Es kann zB bei Klassenableitung zu ungeplanten Untertypenbeziehungen kommen. Deshalb ist es wichtig zu überprüfen ob die bestehenden Untertypbeziehungen richtig sind.

Compiler ist nicht in der Lage alle Bedingungen für Ersetzbarkeit zu überprüfen, deshalb:

1. **Signatur**

Statisch prüfbar. Einfachste Form der Schnittstelle, bestimmt welche Inhalte der Modularisierungseinheit von außen zugreifbar sind: Namen von Inhalten, Typen von Parametern.

Problem: Bedeutung der Inhalte bleibt offen.

Namen in der Signatur in einer Schnittstelle sagen nichts über die Struktur (Implementierungsdetails) aus, deshalb sind sie alleine nicht ausreichend.

Gefahr: gleich heißen aber führen Unterschiedliches aus.

Problem: Zufällig gleiche Signatur (= struktureller Typ) möglich

1. **Abstraktion realer Welt**

Eine Abstraktion ist ein informeller Text, die zusätzlich zur Signatur die Modularisierungseinheit beschreibt.

Bei der *konzeptuellen Ersetzbarkeit* (reale Welt als Grundlage der Abstraktion), sind Zusicherungen besonders wichtig. Dieser Ansatz ist gut aber beruht nur auf Intuition, deshalb gefährlich

1. **Zusicherungen / Erwartungen**

Genaue Beschreibung der erlaubten Erwartungen an eine Modularisierungseinheit (*Design by Contract*).

**Auch möglich: Überprüfbare Protokolle**

Formale Beschreibungen der Erwartungen, die bereits Compiler auf Konsistenz überprüfen kann wenn sie nicht zu kompliziert sind. Können Beziehung zwischen Client und Server (Client Server Protokolle) oder mehreren Einheiten gleichzeitig regeln.

### 4. Was bedeutet Ko-, Kontra- und Invarianz, und für welche Typen in einer Klasse trifft welcher dieser Begriffe zu? (Hinweis: Häufige Prüfungsfrage!)

Siehe Frage 3)

### 5. Was sind binäre Methoden, und welche Schwierigkeiten verursachen sie hinsichtlich der Ersetzbarkeit?

**Binäres Problem**

Ist ein Ersatzbarkeitsproblem: Eingangsparameter dürfen nur kontravariant sein, sonst verletzen sie das Ersetzbarkeitsprinzip.

In der Praxis wünscht man sich manchmal gerade kovariante Eingangsparametertypen. Entsprechende Aufgabenstellungen nennt man kovariante Probleme / binäre Probleme.

Beispiel:

Wir wollen, dass Point 3D Untertyp von Point 2D ist. [Point2D <- Point3D]

Idee: Weil Point3D eine Variable mehr enthält als Point 2D, ist sie als Point2D einsetzbar.

public class **Point2D** {

protected int x, y; // von außen sichtbar

public boolean **equal**(**Point2D** p) {

return x == p.x && y == p.y;

}

}

public class **Point3D** {

protected int x, y, z;

public boolean **equal**(**Point3D** p) {

return x == p.x && y == p.y && z == p.z;

}

}

Problem:

1. Die implementierte equal-Methode (nicht zu verwechseln mit equals() ) erfüllt nicht die Kriterien für Untertypbeziehungen:

Bei Eingabe-Parameter hat man einen Schreibzugriff, deshalb ist Kontravarianz gefordert.

Man soll Obertypen von Point 3D (also Point2D) eingeben können, aber das wäre nicht möglich.

Die Eingangsparameter sind kovariant. Deshalb kovariantes Problem.

1. Problem bei *konzeptueller Ersatzbarkeit*. Fällt ohne Zusicherungen nicht auf. (siehe Beispiel am Ende)

**Binäre Methoden**

Eine Methode wie equal oben, bei der ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt binäre Methode.

Binäre Methoden werden häufig benötigt, sind über Untertypbeziehungen (ohne dynamische Typabfragen und Casts) aber prinzipiell nicht realisierbar.

Die Eigenschaft binär bezieht sich darauf, dass der Name der Klasse in der Methode mindestens zweimal vorkommt – *einmal als Typ von this und mindestens einmal als Typ eines formalen Parameters*.

* Binäre Methoden werden häufig benötigt, sind über Untertypbeziehungen (ohne dynamische Typabfragen und Casts) aber prinzipiell nicht realisierbar
* Widersprechen dem Ersetzbarkeitsprinzip. Es ist sinnlos, in solchen Fällen Ersetzbarkeit anzustreben.
* Binäre Methoden empfangen einen Parameter der Klasse, der gleich der Klasse ist, was bei Vererbung problematisch ist.

### 6. Wie soll man Typen formaler Parameter wählen um gute Wartbarkeit zu erzielen?

Für gute Wartbarkeit: Man soll Parametertypen vorrauschauend und möglichst allgemein wählen.

### 7. Warum ist dynamisches Binden gegenüber switch- oder geschachtelten if-Anweisungen zu bevorzugen?

**Switch- oder geschachtelte if-Anweisungen ersetzen**

Dynamisches Binden ist gegenüber switch- oder geschachtelten if-Anweisungen (und dynamischen Typabfragen) zu bevorzugen.

Es ist nicht möglich bei großem Programmen die Übersicht zu behalten und die Programmteile konsistent zu halten.

public void gibAnredeAus(int art, String name) {

switch(art) {

case 1: System.out.print("S.g. Frau " + name);

break;

case 2: System.out.print("S.g. Herr " + name);

break;

default: System.out.print(name);

}

}

1. Weil beim Einfügen von neuen Bedingungen alle switch Anweisungen in allen Methoden des gesamten Programmes die eine Fallunterscheidung durchführen aktualisiert werden müssen. -> Bei Multimethoden braucht man sie aber nur mittels einer neuen Klasse einführen (Untertyp von Adressat) ohne Struktur von anderen Klassen zu ändern.
2. Es ist unintuitiv int oder Strings als Parameter für Fallunterscheidung zu verwenden.

### 8. Dient dynamisches Binden der Ersetzbarkeit und Wartbarkeit?

Durch dynamisches Binden können Optionen (Unterklassen) einfach hinzugefügt/entfernt werden, ohne die Struktur der anderen Klassen zu ändern.

### 9. Welche Arten von Zusicherungen werden unterschieden, und wer ist für die Einhaltung verantwortlich? (Hinweis: Häufige Prüfungsfrage!)

**Zusicherungen**

Beschreiben die erlaubten Erwartungen an (die Schnittstelle einer) Modularisierungseinheit.

Schnittstelle als Vertrag zwischen Modularisierungseinheit (Server) und Verwendern (Clients). Vertrag beschreibt Verantwortlichkeiten.

Zusicherungen werden als Teil des nominalen Typs betrachtet und in Java sind Zusicherungen nur durch Kommentare auszudrücken. Weil sie als Teil des Objekttyps gelten müssen sie die Bedingungen der Ersetzbarkeit erfüllen.

**Vorbedingungen – Pre Conditions**

Was darf sich Server von Client erwarten?

@param, „Welche Eigenschaften sollen die Eingangsparameter haben?“

Im Untertyp sind sie schwächer oder gleich

Bei Vererbung logische OR-Verknüpfung

**Nachbedingungen – Post Conditions**

Was darf sich Client von Server erwarten?

@result, „Was sagt das Ergebnis aus?“ Beschreibt Methodenergebnis / Änderungen / Eigenschaften des Objektzustands

Im Untertyp sind sie stärker oder gleich

Bei Vererbung logische AND-Verknüpfung

**Invarianten – Invariants**

Welche Eigenschaften sind in Zuständen erfüllt?

Gelten für alle Objektvariablen, Zuständigkeit des Servers (aber auch Client wenn nicht private).

Im Untertyp sind sie stärker oder gleich -> Wenn Variable von außen sichtbar dann gleich

Bei Vererbung logische AND-Verknüpfung (wenn nicht gleich)

**History-Constraints**

In welchen Aufrufreihenfolgen dürfen Clients mit Server interagieren?

Schränken die Entwicklung von Objekten oder Objektvariablen im Laufe der Zeit ein.

**Server-kontrolliert:** ähneln Invarianten (z.B. Wert einer Variable darf nur größer werden)

Im Untertyp sind sie stärker oder gleich -> Wenn Variable von außen sichtbar dann gleich

**Client-kontrolliert:** Reihenfolge der Methodenaufrufe einschränken (z.B. Methode "Initialize" kann in jedem Objekt nur einmal aufgerufen werden)

Im Untertyp sind sie schwächer oder gleich -> Einschränkung bezogen auf Reihenfolge von Methodenaufrufen (Traces bzw. Trace-Set). Trace-Set in T muss Teilmenge von Trace-Set in U sein (U kann mehr Aufrufreihenfolgen erlauben.

Wichtig: schwächer bedeutet weiter oben in Typhierarchie, stärker bedeutet weiter unten in Typhierarchie

1. Schwächer oder gleich stark eingeschränkt: Kovariant
2. Stärker oder gleich stark eingeschränkt : Kontravariant
3. Gleich stark eingeschränkt: Invariant

Das liegt daran dass Aufrufer beim Aufruf eines Untertypen weniger oder gleich viel einzuhalten hat (lockerer), aber alle anderen Erwartungen erfüllt werden müssen wie beim Obertyp oder noch mehr umfassen (strenger sein) sollten.

### 10. Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten, damit das Ersetzbarkeitsprinzip erfüllt ist? Warum? (Hinweis: Häufige Prüfungsfrage!)

Siehe Frage 9)

### 11. Warum sollen Signaturen und Typen stabil bleiben? Wo ist Stabilität besonders wichtig?

**Stabilität an der Wurzel: Bewahrung der Abwärts-Kompatibilität bei Untertypenbeziehungen**

Schnittstellen und Typen sollen deshalb stabil bleiben, weil die Änderung eines Typs weiter oben in der Typenhierarchie, der viele Untertypen besitzt, auch dazu führen kann, dass alle Untertypen geändert werden müssen bzw das Ersetzbarkeitsprinzip plötzlich nicht mehr gilt. Auch bei Zusicherungen.

### 12. Was ist im Zusammenhang mit allgemein zugänglichen (= möglicherweise nicht nur innerhalb des Objekts geschriebenen) Variablen und Invarianten zu beachten?

Server kann Zugriff auf diese nicht kontrollieren, somit Invarianten nicht gewährleisten (Client müsste sie selbst kontrollieren). Deshalb sollen Objektvariablen möglichst nicht durch andere Objekte verändert werden.

Verantwortung des Programmierers. Muss in Zusicherungen bedacht werden.

### 13. Wie genau sollen Zusicherungen spezifiziert sein?

nicht zu schwammig, aber auch nicht zu viele Implementierungsdetails; unmissverständlich!

Man soll Zusicherungen so einsetzen das der Klassenzusammenhalt hoch ist und die Objektkopplung minimiert wird.

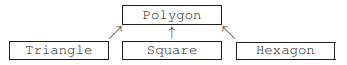
### 14. Wozu dienen abstrakte Klassen und abstrakte Methoden? Wo und wie soll man abstrakte Klassen einsetzen?

**Abstrakte Klassen**

Können keine Objekte erzeugen aber können Code enthalten dass an **konkrete Klassen** vererbt wird.

Hilfreich bei Erstellung eines einheitlichen Obertyps, stabiler als konkrete Klassen, enthalten vollständig implementierte und *abstrakte Methoden* die nicht implementiert werden, dienen nur zur Schnittstellen definition.

Hauptvorteil zum Interface ist die Code-Wiederverwendung. Sinnvoll um duplizierten Code zu vermeiden. Ermöglichen auch Patterns wie „Template-Method“.



public abstract class Polygon {

public abstract void draw();

// draw a polygon on the screen

}

public class Triangle extends Polygon {

public void draw() {

// draw a triangle on the screen

...;

}

}

### 15. Ist Vererbung das gleiche wie das Ersetzbarkeitsprinzip? Wenn Nein, wo liegen die Unterschiede?

**Unterschied zwischen Vererbungs- und Untertypenbezieung**

Sowohl bei der Vererbungs- als auch bei der Untertypbeziehung wird die Schnittstelle des Obertypen implementiert und kann erweitert werden.

Aber der Unterschied liegt darin dass bei einer Untertypbeziehung (abgesehen vom Implementieren der Schnittstelle des Obertyps) auch der Untertyp die Erwartungen / Zusicherungen erfüllt und Obertyp vollständig ersetzen kann.

Der Compiler kennt keinen Unterschied zwischen Vererbung und dem Ersetzbarkeitsprinzip.

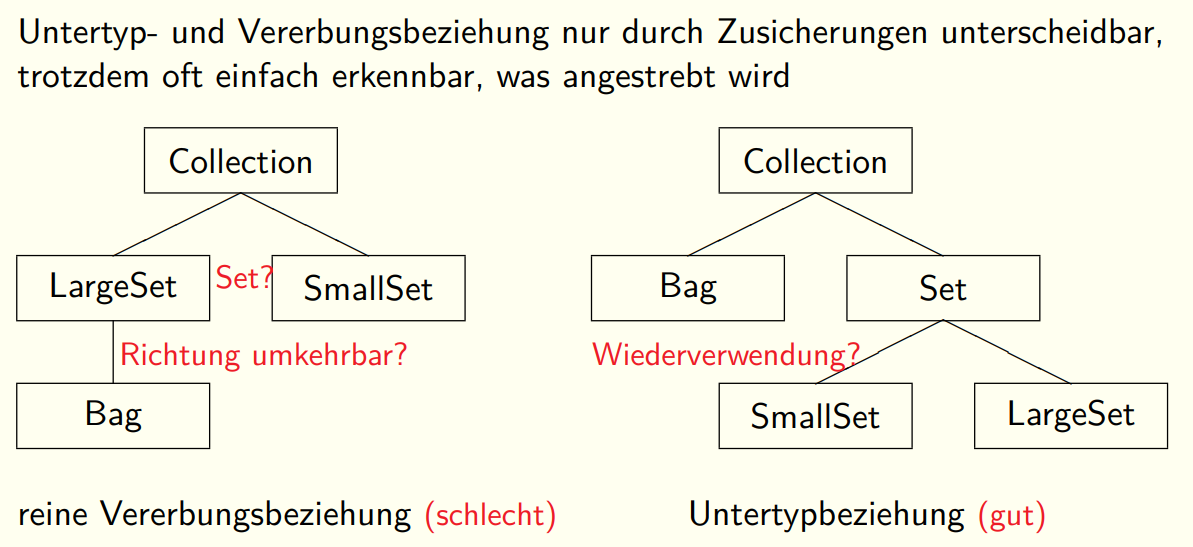
**Vererbung** direkte Codewiederverwendung (leicht sichtbar)

**Untertypbeziehung** indirekte Codewiederverwendung (nur schwer sichtbar) durch Ersetzbarkeit

weniger direkte Codewiederverwendung als Vererbung (da Zusicherungen zu berücksichtigen) – man kann nicht einfach alles übernehmen

**Vererbungsbeziehungen**

1. Reine Vererbungsbeziehung Ziel: reine Wiederverwendung der Implementierung
2. Vererbungs- und Untertypbeziehung Ziel: Typenhierarchie (meistens von is-a Beziehungen) und Vererbung



Auch wenn es auf den ersten Blick so wirkt als würde reine Vererbung eine höhere Codewiederverwendung erzielen, so ist durch das Nichtvorhandensein des Ersetzbarkeitsprinzips:

* die Wartung wesentlich aufwändiger
* keine Möglichkeit dynamisch zu binden und andere Vorteile der Ersetzbarkeit auszunutzen

Nachteile überwiegen den Vorteil einen Code-Anteil nicht erneut schreiben zu müssen.

### 16. Worauf kommt es zur Erzielung von Codewiederverwendung eher an, auf Vererbung oder Ersetzbarkeit? Warum?

Siehe Frage 15)

### 17. Was bedeuten folgende Begriffe in Java?

**Objektvariable**

gehört zu Objekt, Instanz einer Klasse/eines Typs, kann sich zwischen jedem Objekt unterscheiden

**Klassenvariable**

Variablen die zu einer Klasse gehören, in allen Instanzen der Klasse gleich sind. Klassenvariablen werden mit dem Modifier static erstellt.

**statische Methode**

gehört zur Klasse, nicht zu einem Objekt; daher auch Keyword this nicht nutzbar

**Static-Initializer = Klassenkonstruktor**

Ein „static initializer“ ist ein Konstruktor zum Initialisieren von Klassenvariablen. Syntax: static { …. }. Er wird vor der ersten Verwendung der Klasse ausgeführt

**Nested Classes – Innere Klasse**

class EnclosingClass {

class InnerClass { ... }

}

* Gehört zu einem Objekt der umschließenden Klasse
* Objektvariablen und nicht-statische Methoden umschließender Klasse direkt von innen zugreifbar auch wenn private
* darf keine static nested Klassen enthalten, da die Klasse selbst von Objekt abhängig, daher können innere Klassen von ihr nicht statisch sein
* Neue Instanz: a.new InnerClass() wobei a eine Instanz von EnclosingClass ist. Ganz wie gewohnt aufrufbar.

**Probleme in Java** (weil in späteren Java Versionen eingeführt)

* private modifier in nested Class hat keinen Einfluss, Zugriff von umschließender Klasse überall möglich

**Static Nested Classes – Geschachtelte statische Klasse**

class EnclosingClass {

static class StaticNestedClass { ... }

}

* Gehört der umschließenden Klasse
* Objektvariablen und statische Methoden umschließender Klasse direkt von innen zugreifbar auch wenn private
* Neue Instanz: new EnclosingClass.StaticNestedClass()

Wurde erst später in Java eingeführt, deshalb interpretiert als EnclosingClass.StaticNestedClass()

**Probleme in Java** (weil in späteren Java Versionen eingeführt)

* private modifier in static nested Class hat keinen Einfluss, Zugriff von umschließender Klasse überall möglich
* static nested class muss public sein, damit man von außen auf Klasse zugreifen kann, da als EnclosingClass.StaticNestedClass interpretiert. Verzeichnis ähnliche Struktur.

**Best Practice**

Obwohl private modifier in nested Class nicht funktioniert, nicht darauf zugreifen.

Am nützlichsten ist die Innere Klasse zB für Iteratoren, wenn man von der Nested Class auf die Enclosing Class zugreift und nicht umgekehrt.

**final Klasse und final Methode**

Mit final kann man verhindern, dass Methoden in überschrieben werden (final Methoden) oder Klassen erben können (final Klassen).

In manchen Programmierstilen in denen man bewusst den Einsatz von Vererbung minimieren will sehr beliebt.

**Paket, Class-Path, import-Anweisung**

* Paket-Name = Name des Verzeichnisses
* Jedes Paket ist ein eigener *Namespace*
* Verzeichnis mit Quellcode (.java) und compiliertem code (.class). Beide müssen gleich heißen.
* Es darf aber nur eine public class in jeder datei sein.
* Bei geschachtelten Ordnern braucht man explizite packages. (komplizierter)
* Default-Paket ist die Basis der *Verzeichnissturktur* / *Class Path*

Beispiel:

Angenommen wir haben die Methode foo(); in myClasses/examples/AClass.java

Import-Anweisung: importiert Paket/Klasse aus einem Paket. Nun muss Classpath zum Paket/zur Klasse nicht immer explizit angeben werden.

Aufruf ohne Import:

myClasses.examples.test.AClass.foo();

Aufruf mit Import:

import.myClasses.examples.test;

test.AClass.foo();

beziehungsweise

import.myClasses.examples.test.AClass;

import myclasses.test.\*;

AClass.foo();

Vor dem import darf nur stehen:

package PFAD\_ZU\_PACKAGE

### 18. Wo gibt es in Java Mehrfachvererbung, wo Einfachvererbung?

* Mehrfachvererbung ist nur bei Interfaces möglich
* Einfachvererbung bei Interfaces, abstrakten und normalen Klassen.

Bei Klassen darf es nur Einfachvererbung geben.

Unterklassen können nur eine Oberklasse haben, aber mehrere Interfaces implementieren.

### 19. Welche Arten von import-Deklarationen kann man in Java unterscheiden? Wozu dienen sie?

import Classname; // ganze Klasse

import packageName.\*; // alle in diesem Ordner

import Datei; // diesen Pfad bis zu diesem Punkt importieren

### 20. Wozu benötigt man eine package-Anweisung?

Um klar zu spezifizieren, zu welchem package ein Java-File gehört und wo in der Hierarchie es liegt.

Programm nur übersetzbar, wenn package-Name relativ zu Class-Path stimmt.

### 21. Welche Möglichkeiten zur Spezifikation der Sichtbarkeit gibt es in Java, und wann soll man welche Möglichkeit wählen?

**Sichtbarkeit in Java (modifier)**

Scope nur bei Bedarf ausweiten aber idealerweise alles private halten.

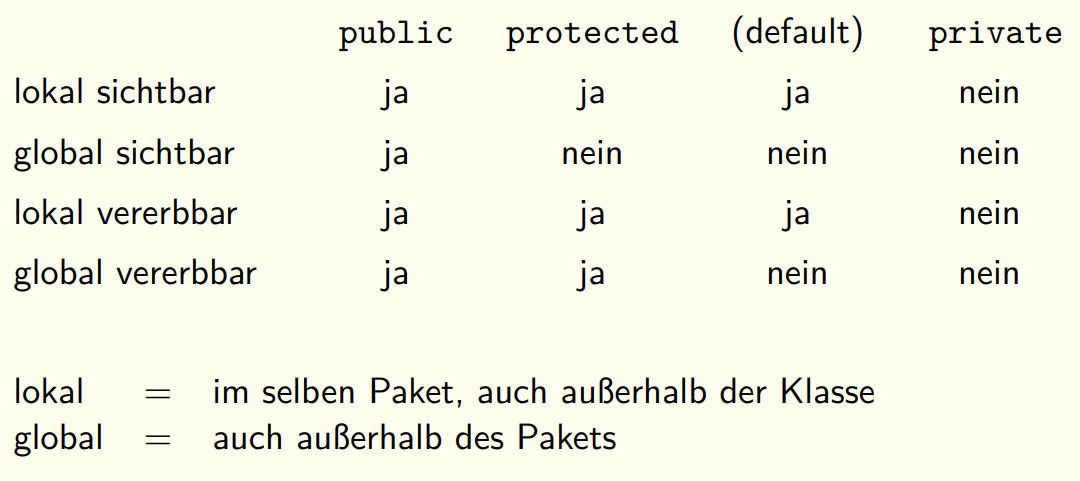
**public** überall sichtbar, vererbbar

**protected** in anderem Paket nicht sichtbar, aber zu einem Untertyp in einem anderen Paket vererbbar

Wichtig: Protected -> nur sichtbar in der Package und in den Unterklassen -> schlechter Stil!

**default** Wie protected, aber nur im eigenen Paket vererbbar.

**private** Nur innerhalb der Klasse sichtbar. Nirgends vererbbar.



### 22. Wodurch unterscheiden sich Interfaces in Java von abstrakten Klassen? Wann soll man Interfaces verwenden? Wann sind abstrakte Klassen besser geeignet?

Interfaces entsprechen abstrakten Klassen in denen alle Methoden als abstract definiert wurden (keine Implementierung enthalten). Sie stellen sicher, dass alle in ihnen angegebenen Methoden auch in derselben Form von ihren Unterklassen verwirklicht werden. Klassen können von mehreren Interfaces erben, aber nur von einer abstrakten Klasse.

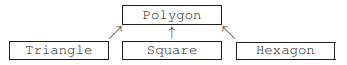
Abstrakte Klassen sollte man dann vorziehen, wenn die Unterklasse nur von einer Oberklasse (ebendiese abstrakte Klasse) erben soll und man eventuell Methoden schon im Vorhinein implementieren will, welche dann in die Unterklasse mit übernommen werden.

**Abstrakte Klassen**

Können keine Objekte erzeugen aber können Code enthalten dass an **konkrete Klassen** vererbt wird.

Hilfreich bei Erstellung eines einheitlichen Obertyps, stabiler als konkrete Klassen, enthalten vollständig implementierte und *abstrakte Methoden* die nicht implementiert werden, dienen nur zur Schnittstellen definition.

Hauptvorteil zum Interface ist die Code-Wiederverwendung. Sinnvoll um duplizierten Code zu vermeiden. Ermöglichen auch Patterns wie „Template-Method“.



public abstract class Polygon {

public abstract void draw();

// draw a polygon on the screen

}

public class Triangle extends Polygon {

public void draw() {

// draw a triangle on the screen

...;

}

}

**Interfaces**

Interfaces erlauben nur static final variablen

Mit default und static können auch Methoden in Interfaces implementiert werden.

Zusicherungen am wichtigsten unter allen Java Klassen.

public interface Z{

static final double x = 3.0;

default double fooZ() {

return fooX() + fooY();

}

}

Die Methoden und per default immer public. deshalb muss man es nicht immer extra dazuschreiben.

Es ist Mehrfachvererbung möglich.

Interfaces sind stabiler als Klassen und immer diesen vorzuziehen.

# Kapitel 3: (19 Fragen)

### 1. Was ist Generizität? Wozu verwendet man Generizität?

**Eigenschaften von Generizität**

* Kaum Einschränkungen, keine kovarianten Probleme
* generische Parameter (in Java weil es keine anderen mehr gibt einfach Typparameter) für Klasse oder Methoden
* statischer Mechanismus, Compiler macht aber keine Kopien für alle Typen
* vereinfacht Wartung und Lesbarkeit
* vernachlässigbare Auswirkungen auf die Laufzeit
* Fehlermeldungen schwer verständlich ausdrückbar

Zweck: Wartbarkeit, Sicherheit, Wiederverwendung, Lesbarkeit

**Eigenschaften von Untertypbeziehungen**

* Ersetzbarkeit, geeignet für unvorhergesehene Wiederverwendung
* Es gibt kovariante Probleme -> keine binären Methoden ausdrückbar ohne dynamischen Typabfragen
* Dynamisches Binden

Sie ergänzen sich gegenseitig.

Mit Generizität, ohne subtyping wären *heterogene Listen* nicht möglich.

Mit subtyping, ohne Generizität wären *homogene Listen* nicht möglich in denen statisch zu compile-Zeit Fehler erkannt werden.

### 2. Was ist gebundene Generizität? Was kann man mit Schranken auf Typparametern machen, das ohne Schranken nicht geht?

In Java durch **Schranken** (Angabe von Untertypen oder Obertypen) einen Typparameter binden:  
 **Gebundener Typparameter:** public class Scene <T extends Scalable & Projectable>

Man darf beliebig viele definieren und sie mit „&“ trennen.

Weiters ist es egal ob T Scalable implementiert oder extended.

**Ungebundener Typparameter:** public class Scene <T>

Ist äquivalent zu public class Scene <T extends Object>

Man kann Typparameter auch **rekursiv** einsetzen:  
 public static <A extends Comparable<A>> A max (Collection<A> xs) {…}

### 3. In welchen Fällen soll man Generizität einsetzen, in welchen nicht?

Guter Einsatz

* gleich strukturierte Klassen/Routinen (Containerklassen, Libraries)
* Fälle, wo Änderungen der Parametertypen absehbar

Schlechter Einsatz

* als Ersatz für Untertypbeziehungen (wenn scheinbar beide einsetzbar – Generizität kann von Änderungen betroffene Bereiche nicht einschränken, da in Client-Code Typparameterersetzungen spezifiziert werden müssen)
* wenn keine Notwendigkeit

### 4. Was bedeutet statische Typsicherheit in Zusammenhang mit Generizität, dynamischen Typabfragen und Typumwandlungen?

Das besondere an Generizität ist die statische Typsicherheit da sich Compile-Time-Fehler zeigen wenn die Typparameter nicht kompatibel sind und man zB versucht eine String-Liste mit Integern zu füllen.

Deshalb sollte man Generizität vor allem bei Containern verwenden (Liste, Stack, Hashtabelle, Menge usw.).

### 5. Was sind (gebundene) Wildcards als Typen in Java? Wozu kann man sie verwenden?

**Wildcards**

Wenn implizite Untertypen nicht angenommen werden entsteht ein Problem:

void drawAll(List<Polygon> p) { // draws all polygons in list p }

Man kann hier zB nicht alle Untertypen von Polygon als Liste lesen -> eigentlich sollten nur das Schreiben (neue Werte zuweisen) ein Problem sein.

**Lösung:** void drawAll(List<? extends Polygon> p) { ... }

Das Fragezeichen (Wildcard) steht für einen beliebigen Typ, der ein Untertyp von Polygon ist. Nun kann man drawAll auch mit Argumenten vom Typ List<Triangle> und List<Square> aufrufen.

Der Compiler erlaubt nur Stellen, für deren Typen in Untertypbeziehungen Kovarianz gefordert ist; das sind Lesezugriffe.

Gelegentlich gibt es auch Parameter, deren Inhalte in einer Methode nur geschrieben und nicht gelesen werden:

void addSquares (List<? extends Square> from, List<? super Square> to ) {

//add squares from ’from’ to ’to’

}

In to wird nur geschrieben, von to wird nicht gelesen.

Als Argument für to können wir daher List<Square>, aber auch List<Polygon> und List<Object> angeben. Als Schranke spezifiziert das Schlüsselwort super, dass jeder Obertyp von Square erlaubt ist.

Der Compiler erlaubt die Verwendung von to nur an Stellen, für deren Typen in Untertypbeziehungen Kontravarianz gefordert ist; das sind Schreibzugriffe.

Auch wichtig: <?> entspricht <? extends Object>.

**Kompliziertere Anwendung von Wildcards**

public class MaxList**<**A extends Comparable**<**? **super** A**>>** extends List<A> {

// Find the Maximum of a comparable List<A>

}

Eine Schranke <A extends Comparable<A> scheint auf den ersten Blick klarer auszudrücken, dass auf den Listenelementen compareTo benötigt wird, weil wir die Liste nur lesen möchten, aber durch die doppelten eckigen Klammern wird super zu extends (bzw umgekehrt).

Es ist schwierig sich als Mensch vorzustellen wie Kovarianz und Kontravarianz zusammenspielen.

### 6. Welche Arten von Generizität kann man hinsichtlich ihrer Übersetzung und ihrem Umgang mit Schranken unterscheiden?

### Welche Art wird in Java verwendet, und wie flexibel ist diese Lösung?

### Homogene Übersetzung (in Java)

Für die Übersetzung generischer Klassen und Methoden in ausführbaren Code:

* Heterogene Übersetzung: Automatisierung des Erstellens von Kopien für einzelne Typen durch Compiler
* Homogene Übersetzung: Automatisierung des Erstellens der nötigen Type-Casts durch Compiler

Bei der heterogenen Übersetzung wird jede Klasse (egal ob generisch oder nicht-generische), in genau eine Klasse mit JVM-Code übersetzt. -> Kaum Unterschiede zu nicht generischen Klassen.

**Schritte in der homogenen Übersetzung**

Übersetzt eine generische in genau eine nicht-generische JVM (JavaVirtualMachine)‐Code Klasse.

**1.** spitze Klammern samt Inhalten weglassen

**2.** Typparameter durch (erste) Schranke oder Object ersetzen

**3.** Typumwandlungen bei Aufrufen einfügen wenn Ergebnis- oder Parametertyp Typparameter ist

Manchmal auch Casting notwendig (zB bei Rückgabetypen).

Der entstehende Nachteil in der neuen (nicht von compiler ausgeführten) Form ist, dass Fehlermeldungen sich nicht mehr bei Compile-Zeit zeigen.

|  |  |
| --- | --- |
| public interface Collection<A> {  void add(A elem);  Iterator<A> iterator();  }  public interface Iterator<A> {  A next();  boolean hasNext();  } | public interface Collection {  void add(Object elem);  Iterator iterator();  }  public interface Iterator {  Object next();  boolean hasNext();  } |

**Heterogene Übersetzung**

Bei der heterogenen Übersetzung wird für jede Verwendung einer generischen Klasse oder Methode mit anderen Typparametern eigener übersetzter Code erzeugt:  
 Templates in C ++

Einfach Copy-and-Paste vom Compiler für jeden Datentyp (kann Vorteilhaft sein für primitive Datentypen weil Compiler optimiert)

Im Vergleich zur homogenen Übersetzung:

Vorteile:

* effizienter da keine Typumwandlung und Code optimierbar -> int, char, . . . direkt verwendbar
* uneingeschränkt verwendbar
* keine Typumwandlungen zur Laufzeit und entsprechende Überprüfungen

Nachteile:

* oft viele Klassen und große Programme, Speicherintensiv
* Klassenvariablen kopiert (= unklare Semantik)
* keine Raw-Types verwendbar
* generischer und nichtgenerischer Code inkompatibel
* schlechtere Fehlermeldungen

### 7. Wie kann man Generizität simulieren? Worauf verzichtet man, wenn man Generizität nur simuliert?

Siehe Frage 6)

Homogene Übersetzung durch Programmierer -> Verlust der statischen Typsicherheit.

### 8. Was wird bei der heterogenen bzw. homogenen Übersetzung von Generizität genau gemacht?

Siehe Frage 6)

### 9. Was muss der Java-Compiler überprüfen um sicher zu sein, dass durch Generizität keine Laufzeitfehler entstehen?

Siehe Frage 6)

### 10. Welche Möglichkeiten für dynamische Typabfragen gibt es in Java, und wie funktionieren sie genau?

**Dynamische Typabfragen (getClass, instanceof)**

Prozeduale und Funktionale Programmiersprachen haben keine dynamischen Programminformationen die erst zum Zeitpunkt der Ausführung bestimmt werden, in OOP werden sie aber sogar benötigt!

**A.getClass()** liefert dynamischen Typen

**A instanceof B** boolsche Aussage über dynamischen Typen: Liefert true, wenn A ein Untertyp von B ist, sonst false.

Allgemein gilt:

* Dynamisches Binden ist besser als Dynamische Typabfragen
* Wenn es sich vermeiden lässt sollte man dynamische Typabfragen vermeiden da oft zur Umgehung statischer Typsicherheit eingesetzt

Beispiel für Vermeidung:

if (x instanceof T1)

doSomethingOfTypeT1((T1)x);

else if (x instanceof T2)

doSomethingOfTypeT2((T2)x);

...

else

doSomethingOfAnyType(x);

Ersetzen durch:

x.doSomething();

Ausgenommen man muss vielfach dieselbe Methode in allen Untertypen implementieren was die Wartbarkeit negativ beeinflussen würde.

### 11. Was wird bei einer Typumwandlung in Java umgewandelt – der deklarierte, dynamische oder statische Typ? Warum?

**Sichere Typumwandlung (Casting)**

Casting / dynamische Typumwandlung ändert den deklarierten Typen zur Laufzeit (nützlich bei Methodeneingabe-Parametern um Typ spezifischer zu machen, aber nicht allgemeiner).

* Wenn möglich dynamischen Typumwandlung vermeiden und stattdessen Generizität verwenden oder dynamisches Binden

Schwierigkeiten beim Ersatz der Typumwandlung durch dynamisches Binden:

* bei sehr allgemeinen Typen (sollte vermieden werden)
* Erweiterungen sind nicht erlaubt
* wichtige private Methoden
* sehr viele Untertypen werden benötigt

**Gefahren bei Casting**

* Class Cast Exceptions
* die statische Typ-Überprüfung durch den Compiler wird umgangen, Fehler werden versteckt
* schlechte Wartbarkeit beim Einsatz vieler Typumwandlungen

**Up-Cast** Casting Richtung Obertyp

Up-Cast ist nicht so gebräuchlich und nicht nicht fehleranfällig.

<A> Collection<A> **up**(List<A> xs) {

return (Collection<A>)xs;

}

**Down-Cast** Casting Richtung Untertyp

1. Down-Cast nach dynamischer Typabfrage:

dynamische Typabfrage stellt sicher, dass das Objekt einen entsprechenden dynamischen Typ hat

alternativer Programmzweig nötig und fehleranfällig wenn Abfragen zu allgemein, sollte man vermeiden

<A> List<A> **down**(Collection<A> xs) {

if (xs instanceof List<A>)

return (List<A>)xs;

else { ... }

}

1. Down-Cast wie bei Generizität, aber nur händisch überprüft:

gleichförmige Ersetzung der Typparameter und

keine impliziten Untertypbeziehungen

wobei Intuition oft irreführend, denn:

List<String> -/-> List<Integer> impliziert auch List -/-> List

Daraus folgt: List darf nie Untertyp von List sein

List<String> **bad**(Object o) {

if (o instanceof List<String>) // error

return (List<String>)o; // error

else { ... }

}

Problem bei bad: -> Keine Information über gemeinsamen Typ / Schranke

### 12. Welche Gefahren bestehen bei Typumwandlungen?

Siehe Frage 11)

### 13. Wie kann man dynamische Typabfragen und Typumwandlungen vermeiden? In welchen Fällen kann das schwierig sein?

Siehe Frage 11)

### 14. Welche Arten von Typumwandlungen sind sicher? Warum?

Siehe Frage 11)

### 15. Was sind kovariante Probleme und binäre Methoden? Wie kann man mit ihnen umgehen oder sie vermeiden?

**Kovariantes / Binäres Problem**

Ist ein Ersatzbarkeitsproblem: Eingangsparameter dürfen nur kontravariant sein, sonst verletzen sie das Ersetzbarkeitsprinzip.

In der Praxis wünscht man sich manchmal gerade kovariante Eingangsparametertypen. Entsprechende Aufgabenstellungen nennt man kovariante Probleme / binäre Probleme.

Beispiel:

Wir wollen, dass Point 3D Untertyp von Point 2D ist. [Point2D <- Point3D]

Idee: Weil Point3D eine Variable mehr enthält als Point 2D, ist sie als Point2D einsetzbar.

public class **Point2D** {

protected int x, y; // von außen sichtbar

public boolean **equal**(**Point2D** p) {

return x == p.x && y == p.y;

}

}

public class **Point3D** {

protected int x, y, z;

public boolean **equal**(**Point3D** p) {

return x == p.x && y == p.y && z == p.z;

}

}

Problem:

1. Die implementierte equal-Methode (nicht zu verwechseln mit equals() ) erfüllt nicht die Kriterien für Untertypbeziehungen:

Bei Eingabe-Parameter hat man einen Schreibzugriff, deshalb ist Kontravarianz gefordert.

Man soll Obertypen von Point 3D (also Point2D) eingeben können, aber das wäre nicht möglich.

Die Eingangsparameter sind kovariant. Deshalb kovariantes Problem.

1. Problem bei *konzeptueller Ersatzbarkeit*. Fällt ohne Zusicherungen nicht auf. (siehe Beispiel am Ende)

**Binäre Methoden**

Eine Methode wie equal oben, bei der ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt binäre Methode.

Binäre Methoden werden häufig benötigt, sind über Untertypbeziehungen (ohne dynamische Typabfragen und Casts) aber prinzipiell nicht realisierbar.

Die Eigenschaft binär bezieht sich darauf, dass der Name der Klasse in der Methode mindestens zweimal vorkommt – *einmal als Typ von this und mindestens einmal als Typ eines formalen Parameters*.

* Binäre Methoden werden häufig benötigt, sind über Untertypbeziehungen (ohne dynamische Typabfragen und Casts) aber prinzipiell nicht realisierbar
* Widersprechen dem Ersetzbarkeitsprinzip. Es ist sinnlos, in solchen Fällen Ersetzbarkeit anzustreben.
* Binäre Methoden empfangen einen Parameter der Klasse, der gleich der Klasse ist, was bei Vererbung problematisch ist.

**Weiteres Beispiel für kovariante Probleme:**

Wir möchten Tiger und Rind korrekt füttern. Dafür erfolgt eine dynamische Typabfrage des Futters.

friss() wird hier überladen mit Futter und Fleisch, es wäre aber sinnvoller generell friss von Tier zu entfernen und das kovariante Problem an erster Stelle zu vermeiden.

Dadurch muss man sich entweder merken welches Futter ein Tier genau frisst oder man überlädt friss() mit Fleisch und Gras.

**Dynamische Typabfrage und Überladung als Lösung**

abstract class Futter { ... }

class Fleisch extends Futter { ... }

class Gras extends Futter { ... }

abstract class Tier {

public abstract void **friss**(Futter x);

}

class Rind extends Tier {

public void friss(Gras x) { ... }

public void friss(Futter x) {

if **(x instanceof Gras)**

friss(**(Gras)**x);

else

werdeKrank();

}

}

class Tiger extends Tier {

public void friss(Fleisch x) { ... }

public void friss(Futter x) {

if **(x instanceof Fleisch)**

friss(**(Fleisch)**x);

else

fletscheZaehne();

}

}

### 16. Wie unterscheidet sich Überschreiben von Überladen, und was sind Multimethoden?

**Überschreiben**

* Methode in Unterklasse überschreibt Methode in Oberklasse, sie wird also verwendet, egal welcher Typ deklariert ist

**Überladen**

* Methode in Unterklasse koexisitiert zu Methode in Oberklasse (tritt bei Java immer auf wenn: Eingangsparameter nicht invariant, Ausgangsparameter nicht kovariant)
* Bei überladenen Methoden entscheidet stets der deklarierte Typ über das Verhalten, was unerwünschtes Verhalten hervorruft.
* Mit super. aufrufbar

**Multimethoden**

* Wie Überladen, nur dynamischer Typ des Objektes welche die Methode aufruft entscheidet welche Methode aufgerufen wird.

### 17. Wie kann man Multimethoden simulieren? Welche Probleme können dabei auftreten?

Mehrfaches dynamisches binden wie es von Multimethoden gebraucht wird, kann man durch wiederholtes einfaches Binden simulieren. Dieses Prinzip nennt man das Visitor Pattern. Das Problem: Die Anzahl der zusätzlich nötigen Methoden, wird bei steigender Klassenanzahl schnell sehr groß!

### 18. Was ist das Visitor-Entwurfsmuster?

Siehe Kapitel 5, Frage 1)

### 19. Wodurch ist Überladen problematisch, und in welchen Fällen ergeben sich kaum Probleme?

1)   Unbewusstes Überladen statt überschreiben.

2)   Wenn nicht klar entschieden werden kann welche der Methoden aufgerufen werden soll

# Kapitel 4: (15 Fragen)

### 1. Wie werden Ausnahmebehandlungen in Java unterstützt?

Ausnahmen sind Objekte die das **Throwable**-Interface implementieren

+-----------+

| Throwable |

+-----------+

/ \

/ \

+-------+ +-----------+

| Error | | Exception |

+-------+ +-----------+

/ | \ / | \

\\_\_\_\_\_/ \\_\_\_\_\_\_/ \

unchecked checked +-------------------------+

| RuntimeException |

+-------------------------+

/ | | \

\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_/

unchecked

Java Documents:

“If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception”

Checked (at compile time) = checked Exceptions

Unchecked (at compile time) = Error, Runtime Exceptions

Zwei wichtige Unterklassen von diesem Interface sind:

* **Error (Unchecked)**

Unterklassen von Error werden hauptsächlich für vordefinierte, schwerwiegende Ausnahmen des Java-Laufzeitsystems verwendet und deuten auf echte Fehler hin, die während der Programmausführung entdeckt wurden.

Es ist praktisch kaum möglich, solche Ausnahmen abzufangen. ihr Auftreten führt fast immer zur Programmbeendigung. Beispiele: OutOfMemoryError und StackOverflowError. (Jeder Versuch Programm fortszusetzen führt zum selben Error)

* **Exception**

Unterklassen von Exception sind wieder in zwei Bereiche gegliedert.

Oft ist es sinnvoll Ausnahmen, die Objekte von Exception sind, abzufangen und den Programmablauf an geeigneter Stelle fortzusetzen.

1. **Checked**

Checked steht für checked at compile time

vorhersehbare Ausnahmefehler, häufig von uns selbst definiert

Man kann darüber diskutieren, ob es vernünftiger ist, statt überprüfte Ausnahmen (checked-Exceptions) Untertypen von RuntimeException zu verwenden.

2. **RuntimeException (Unchecked)**

nicht überprüfbare, unvorhersehbare, meist vom System vorgegebene Exceptions

Beispiele: IndexOutOfBoundsException (Arrayzugriff außerhalb des Indexbereichs)

NullPointerException (Senden einer Nachricht an null)

ClassCastException (Typumwandlung, wobei der dynamische Typ nicht dem gewünschten Typ entspricht)

Vom Java-Laufzeitsystem werden nur Objekte der vordefinierten Unterklassen von Error und RuntimeException als Ausnahmen ausgelöst. Programme können aber explizit auch andere Ausnahmen mit dem throw Befehl werfen.

*Try-Catch-Blöcke* definieren den Punkt ab dem ein Programm fortgesetzt wird.

try { ... }

catch(Exception e) {}

finally { ... }

* Wird der try-Block ausgeführt, so wird in jedem Fall auch der finally- Block ausgeführt, unabhängig davon, ob Ausnahmen aufgetreten sind oder nicht.
* Tritt eine nicht abgefangene Ausnahme im try-Block oder in einer catch-Klausel auf, wird der finally-Block vor Weitergabe der Ausnahme ausgeführt.
* Tritt während der Ausführung des finally- Blocks eine nicht abgefangene Ausnahme auf, wird der finally-Block nicht weiter ausgeführt und die Ausnahme weitergegeben.

### 2. Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?

**Ersetzbarkeit / Untertypenbeziehungen und Ausnahmebehandlung**

1. Methode in der Unterklasse darf nur weniger Exceptions werfen als in der Oberklasse (und nur die aus der Oberklassenmethode)
2. Methoden in der Unterklasse sollen nur dann Exceptions werfen wenn Aufrufe der Oberklasse dies auch unter vergleichbaren Bedingungen erwarten (abhängig von Zusicherungen):

class A {

void foo() throws SelfMadeException, SyntaxError { ... }

}

class B extends A {

void foo() throws SelfMadeException { ... }

}

Beispiel: Nachbedingungen beeinflussen die Exceptions

Reminder - Nachbedingungen dürfen im Untertypen nur stärker werden, nicht schwächer

**Oberklasse** Wenn Eingabewert < 5 dann wirf eine Exception

**Unterklasse** Ausschließlich in diesem Fall / unter dieser Bedingung darf eine Exception geworfen werden ( nur < 5 )

Wenn der Server sich dazu verantwortet eine Exception zu werfen wenn Client einen Wert <5 übergibt, weil das die Vorbedingung ist, dann ist das eine zusätzliche Verantwortung für den Server.

### 3. Wozu kann man Ausnahmen verwenden? Wozu soll man sie verwenden, wozu nicht?

**Guter Stil: Sinnvoller Einsatz von Ausnahmen**

* Generell sind Ausnahmen sparsam und wirklich nur in Ausnahmesituationen einzusetzen. Bevorzugter weise sollten diese noch in derselben Klasse gefangen werden, wo sie auch geworfen wurden.
* Typabfragen (auch bei Exceptions – durch vielen catch Blöcken bei try-catch-finally Block) sollen - soweit es möglich ist - vermieden werden um die Wartbarkeit zu verbessern.
* Unvorhergesehene Programmabbrüche

Wenn Ausnahme nicht abgefangen wird, Programm abbrechen und Stack-Trace in der Konsole zeigen mit ausführlichem Fehlerhinweis

* Kontrolliertes Wiederaufsetzen

im praktischen Einsatz soll das Programm auch dann noch funktionieren, wenn ein Fehler aufgetreten ist

zb: Kommerzielle Anwendungen wie „Program crashed, please report error to the following site…“

* Ausstieg aus Sprachkonstrukte

allgemein das vorzeitige Abbrechen der Ausführung von Blöcken, Kontrollstrukturen, Methoden, etc. in außergewöhnlichen Situationen

* Rückgabe alternativer Ergebniswerte

Es ist möglich, dass die Methode an den Aufrufer in Ausnahmesituationen Objekte zurückgibt, die nicht den deklarierten Ergebnistyp der Methode haben

**Schlechter Stil: Falscher Einsatz von Ausnahmen**

* Vorzeitiger Ausstieg aus Sprachkonstrukten: zu früh aus der Hierarchie im Stack heraussteigen -> es soll nachvollziehbar sein wo man aus dem Programm ausgestiegen ist.
* Rückgabe alternativer Werte statt Ausnahme zu werfen: Returnen von anderen Ergebniswerten wegen schlechter Programmstruktur -> Beispielsweise das returnen von einem String mit „Ein Fehler ist aufgetreten“ bei einer Exception in einer Methode die Strings returned.
* Kontrollierte / überprüfte exceptions eher gar nicht verwenden, sie sind fehleranfällig weil sie nur in den unteren 2 Fällen als Beispiel für schlechten Stil benutzt werden -> Java ist die einzige Sprache die nicht von überprüften exceptions zu applicated exceptions umgestiegen ist
* Wenn man von außen eine Methode ansprechen will die eine throws Klausel beinhaltet wird der Umgang damit sehr schwer!

**Beispiele für schlechten Stil**

// Ohne Ausnahmebehandlung: pro Iteration zwei Vergleiche mit null

while (x != null)

x = x.getNext();

// Mit Ausnahmebehandlung: pro Iteration nur ein Vergleich mit null

try { while (true)

x = x.getNext();

} catch (NullPointerException e) {}

Kann zwar effizienter sein, aber es ist schlechter Stil und spart minimal zeit weil wir sehr gute compiler haben, abgesehen davon können auch in getNext() Fehler sein die wir hier fälschlicherweise abfangen.

// trickreiche Verwendung von Ausnahmen:

try { throw x }

if (x instanceof T1) {...} catch (T1 x) {...}

else if (x instanceof T2 {...} catch (T2 x) {...}

... ...

else if (x instanceof Tn {...} catch (Tn x) {...}

else {...} catch (Exception x) {...}

Rechts besser als links, aber schlechte Wartung weil wir immer neue Verzweigungen hinzufügen müssen wenn wir eine neue Exception einbauen wollen. Generell sind dynamische Typabfragen bei Exceptions zu vermeiden.

Ohne Ausnahmebehandlung:

public static String addA (String x, String y) {

if (onlyDigits(x) && onlyDigits(y)) { ... }

else **return "Error"**;

}

Mit Ausnahmebehandlung:

public static String addB (String x, String y) throws NoNumberString {

if (onlyDigits(x) && onlyDigits(y)) { ... }

else **throw new NoNumberString();**

}

### 4. Durch welche Sprachkonzepte unterstützt Java die nebenläufige Programmierung? Wozu dienen diese Sprachkonzepte?

**Nebenläufige Programmierung (Multithreading)**

Sequentielle Programmierung 🡪 Nebenläufige Programmierung

Früher benutzte man nur einen CPU-Kern und jetzt haben wir mehrere CPU-Kerne die echt parallel arbeiten können.

Prozesse werden in Java einem oder mehreren Threads (in der Java Virtual Machine) zugeteilt.

Wenn man mehrere Prozessor-Kerne (Cores) hat dann wird echt parallelisiert, ansonsten wird nur die Illusion der Parallelisierung bereitgestellt.

**Threads** Ein Thread ist ein (von anderen Threads) unabhängiger Kontrollfluss

**Gefahren**

* Bei Multithreading hat man nicht dieselbe Kontrolle über Programmausführungsreihenfolge da nur mehr sequentially consistent innerhalb eines einzelnen Threads, aber Threads arbeiten parallel, warten nicht aufeinander und können einander unterbrechen
* Gleichzeitige und überlappte Zugriffe auf Variablen → Fehler wenn inkonsistente Zustände existieren (häufig!)

Der Programmierer muss sich um die Synchronisation kümmern

Synchronisation ist die häufigste Fehlerquelle

* Testen ist sehr schwierig

**Beispiel**: Nicht atomare Befehle innerhalb eines Threads in einer nebenläufigen Umgebung

Wenn nicht Befehle synchronisiert werden können Fehler entstehen!

Hier: i und j werden nach wenigen Tausend Aufrufen von schnipp() nicht mehr gleich sein.

public class Zaehler {

private int i = 0, j = 0;

public void schnipp() { i++; j++; }

}

public class Zaehler {

private int i = 0, j = 0;

public **synchronized** void schnipp() { i++; j++; }

}

**Synchronized** Soll gleichzeitige und überlappte Zugriffe vermeiden „*Mutual-Exclusion*“ während Zustände inkonsistent sein können. Synchronized Blöcke und Methoden sollten nur ganz kurz laufen.

Feinst-mögliche Granularität (Körnung) zu wählen, da Synchronisierung teuer bezüglich Laufzeit ist, da die anderen Transaktionen auf die momentan laufende warten müssen.

Synchronized bedeutet dass wir am Objekt nicht parallel mehrere Dinge ausführen können.

Nur Synchronized Blöcke dürfen gleichzeitig miteinander interagieren.

LOCK wird auf ein Objekt gesetzt -> bei rekursiven aufrufen dürfen wir zb wieder zurück zu uns selbst weil wir im lock-objekt sind aber andere aufrufer müssen warten. -> Wenn LOCK auf einem Thread gesetzt wurde, dann müssen wir warten bis dieser fertig ist.

public class Zaehler {

private int i = 0, j = 0;

public void schnipp() {

**synchronized(this)** { i++; }

**synchronized(this)** { j++; }

}

}

In einem synchronized Block: „Lock“ wird auf das Argument von synchronized gesetzt: also hier „this“ (bei synchronized Methoden passiert das automatisch).

i und j sind nur unterschiedlich wenn man es zwischen den beiden synchronized blöcken abfragt oder nur kurzfristig, aber es wird kein Aufruf vergessen.

**Gefahren beim Einsatz von Synchronisation**

Auf Synchronisation achten auch wenn man vorgefertigte Datenstrukturen aus der Bibliothek verwendet.

Man hat zwar versucht in Java über die Namensgebung Struktur in die Thread-Sicherheit von Klassen zu bringen, aber ganz ist das nicht gelungen: Beispielsweise sind Vector und Hashtable sicher, die ähnlichen Klassen LinkedList und HashMap aber nicht.

Es die Gefahr, dass wir zu viel synchronisieren und zu viele Threads verzögern die warten müssen und die Vorteile der Parallelisierung verlieren.

**Deadlocks**

unendliche Verzögerung, durch zyklischen Abhängigkeiten zwischen zwei oder mehr Threads.

* Programmteil wartet darauf dass irgendwo eine Ressource frei wird nachdem vorher angefordert wurde, und kann dadurch unendliches Warten erzeugen -> blockieren kann man aber feststellen

Wenn man aber die Reihenfolge stark beeinflusst um Deadlocks und zyklische Abhängigkeiten (die zum warten zwingen) zu vermeiden, kann man auch die Nebenläufigkeit zu sehr einschränken.

Deshalb riskiert man oft einfach Deadlocks.

Es gibt Beweis-Werkzeuge für Java die zeigen können dass keine Deadlocks auftreten.

**Liveness Properties: Livelocks & Starvation**

* Starvation: Jemand braucht Ressource so stark dass andere vehungern -> sehr schwierig in den Griff zu bekommen
* Livelock: Ähnlicher Effekt wie Deadlock, aber dadurch das sich flags ununterbrochen ändern und kein Fortschritt erreicht wird, ähnlich wie in einer Dauerschleife

Da Livelocks und Starvation viele unterschiedliche Ursachen und Auswirkungen haben können, gibt es dafür auch keine klaren formalen Definitionen. Keine formalen Beweise möglich, daher bleibt nur das Testen.

**Lösungen**

Auch die Entwurfsmuster an denen man sich immer versucht zu orientieren lösen Fehler nicht: Man muss synchronisation wirklich nur so einfach halten wie es nur geht und gut testen.

### 5. Wozu brauchen wir Synchronisation? Welche Granularität sollen wir dafür wählen?

Siehe Frage 4)

### 6. Zu welchen Problemen kann Synchronisation führen, und was kann man dagegen tun?

Siehe Frage 4)

### 7. Wozu dienen Annotationen? Wann setzt man sie sinnvoll ein?

**Annotation in Java**

Optionaler Parameter der in Java Code selbst eingefügt wird und keine externe Beschreibung wie in AspectJ.

* an (fast) beliebige Sprachkonzepte anheftbar
* im Java-Code statisch gesetzt
* von gesamter Werkzeugkette bis zur Laufzeit auslesbar
* haben ohne explizite Überprüfungen (zu runtime oder compile time) keine Auswirkung auf Programm

(wenn zu runtime, dann ist die Annotation eine Reflexion)

* können selbst definiert werden
* können Argumente enthalten

Bisher benutzt: @Override

Der Compiler prüft, ob die Methodendefinition mit dieser Annotation versehen ist und verlangt nur in diesem Fall, dass die Methode eine andere Methode überschreibt.

**Selbst erstellte Annotationen**

**Anwendung**

@BugFix(who="Kaspar", date="2017-12-20", level=3,

bug="class unnecessary and maybe harmful",

fix="contents of class body removed")

public class Buggy { }

**Erstellung**

Annotationen sind eine Adaption der Syntax von Interfaces

@Retention(RetentionPolicy.RUNTIME) // predefined annotation in Java

@Target({ElementType.TYPE}) // predefined annotation in Java

public **@interface** BugFix {

String who() default "me"; // author of bug fix

String date(); // when was bug fixed

int level() default 1; // importance level 1-5

String bug(); // description of bug

String fix(); // description of fix

}

Wie man sieht kann man auch default Werte angeben.

Einschränkungen in @interface:

* nur Eingabeparameter-lose Methoden
* wenn Methodenname value ist, dann muss man nicht zusätzlich value = … schreiben
* Als Ergebnistyp nur:

alle elementaren Typen (int, double, . . . )

Aufzählungstypen (enum)

String

Class

andere Annotationen

eindimensionale Arrays dieser Typen (als „Mengen“)

**Vordefinierte Annotationen**

Bugfix selbst ist mit Annotationen versehen.

Man kann auch die runden Klammer bei Annotationen wenn nicht nötig weg lassen: @Override()

@Retention(RUNTIME) // gleich mit @Retention(value=RUNTIME)

@Target({ANNOTATION\_TYPE, TYPE}) // initialisiert eindimensionales Array

// @Target(ANNOTATION\_TYPE) auch möglich

public @interface Retention {

RetentionPolicy value();

}

public enum RetentionPolicy {

CLASS, RUNTIME, SOURCE

}

public @interface Target {

ElementType[] value();

}

public enum ElementType {

ANNOTATION\_TYPE, CONSTRUCTOR, FIELD, LOCAL\_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE

}

@Target legt fest wo Annotation angeheftet wird (Zum Beispiel an Methode oder ganzen Klasse -> siehe API von ElementType)

@Retention legt fest, wie lange die gerade definierte Annotation sichtbar bleiben soll / leben soll.

Element aus Enum RetentionPolicy.

* RUNTIME : Falls @Retention(RUNTIME) wird echtes Interface erzeugt und man nennt es dann eine Reflexion. Die Annotation ist dann auch zur Laufzeit zugreifbar.
* SOURCE : Es wird die Annotation vom Compiler genau so verworfen wie Kommentare.

Solche Annotationen sind nur für Werkzeuge, die auf dem Source-Code operieren, von Interesse.

* CLASS : sorgt dafür, dass die Annotation in der übersetzten Klasse vorhanden bleibt, aber während der Programmausführung nicht mehr sichtbar ist. Das ist nützlich für Werkzeuge, die auf dem Byte-Code operieren.

@Documented sorgt dafür, dass die Annotation in der generierten Dokumentation vorkommt

@Inherited sorgt dafür, dass das annotierte Element auch in einem Untertyp als annotiert gilt.

**Reflexion in Java**

Die Technik, mit der man zur Laufzeit auf Annotationen zugreifen kann, nennt man Reflexion bzw. Reflection oder Introspektion. Reflexion erlaubt uns zur Laufzeit auf viele Details eines Programms zuzugreifen.

Wurde über @Retention(RUNTIME) festgelegt, dass eine Art von Annotationen auch zur Laufzeit zugreifbar ist, dann generiert der Compiler ein entsprechendes Interface. Dieses sieht für obiges Beispiel so aus:

public interface BugFix extends java.lang.annotation.Annotation {

String who();

String date();

int level();

String bug();

String fix();

}

Dann kann man auch Werte von Annotationen zur Laufzeit aufrufen.

So wie getClass die Klasse eines Objekts als Objekt der Klasse Class ermittelt, so enthält die statische Variable class jeder Klasse das entsprechende Objekt der Klasse Class.

Ausgangspunkt ist meist ein Objekt vom Typ Class. Class implementiert eine Reihe von Methoden, mit denen man sich die Details der Klasse ansehen kann. So wie man über getAnnotation und getAnnotations Informationen über Annotationen bekommt, kann man sich über getMethod und getMethods Informationen über Methoden und über getField und getFields Informationen über Objekt- und Klassenvariablen holen.

Ähnliches gilt für Konstruktoren, Oberklassen, das Paket und so weiter. (Class hat viele nützliche Methoden, genau wie die Klassen Method und Field).

String s = "";

BugFix a = Buggy.class.getAnnotation(BugFix.class);

if (a != null) { // null if no such Annotation

s += a.who() + " fixed a level " + a.level() + " bug";

}

Method[] ms = Buggy.class.getMethods();

// verschiedene analoge Methoden auch auf Method, Field

Annotation[] as = Buggy.class.getAnnotations();

for (Annotation a : as) {

if (a instanceof BugFix)

String s = ((BugFix)a).who; ...

}

**Anwendungen von Annotationen und Reflexionen**

**Annotation**

Häufig verwendet man @Override. Statt dieser Annotation wäre auch ein Modifier (wie zb public, private, default, usw…) sinnvoll gewesen, aber aufgrund der geschichtlichen Entwicklung hat sich eine Annotation angeboten.

Man kann jede Annotation als Modifier sehen, den ein (Pre-) Compiler oder das Laufzeitsystem versteht. Manchmal stolpert man über eine @Deprecated-Annotation, mit der Programmelemente, die man nicht mehr verwenden sollte, gekennzeichnet werden. Eigentlich ein Kommentar aber Annotation ermöglicht dass Compiler bei Benutzung der Programmelemente eine Warnung ausgibt.

Eine gefährliche Annotation ist @SuppressWarnings. Diese Annotation weist den Compiler an, alle Warnungen zu unterdrücken.

Mit der Annotation @FunctionalInterface können seit Java 8 Interfaces gekennzeichnet werden, die genau eine abstrakte Methode enthalten.

**Reflexion**

Reflexion ist eine Variante der Metaprogrammierung, einer schon sehr alten Programmiertechnik, mit der es viel Erfahrung gibt.

Während durch Metaprogrammierung das gesamte Programm zur Laufzeit sicht- und änderbar ist, kann Reflexion die Programmstruktur nicht ändern. Man weiß, dass man mit diesen Techniken in speziellen Fällen sehr viel erreichen kann. Man kennt aber auch die Gefahren. Daher versucht man solche Techniken in der tagtäglichen Programmierung zu vermeiden.

Beispiel:

static void execAll(String n, Object... objs) {

for (Object o : objs) {

try { o.getClass().getMethod(n).invoke(o); }

catch(Exception ex) {...}

}

}

Das ist gefährlich weil wir nicht wissen was die mit invoke aufgerufenen Methoden machen.

Möglicherweise ist die Methode nicht public, verlangt (andere) Argumente, oder existiert gar nicht. In diesen Fällen werden Ausnahmen geworfen, die wir irgendwie abfangen müssen.

Die Gefahr kommt hauptsächlich daher, dass wir keinerlei Verhaltensbeschreibungen der mit invoke aufgerufenen Methoden haben.

### 8. Wie lange können Annotationen leben? Wofür ist welche Lebensdauer sinnvoll?

je nach Wert von @Retention: SOURCE – verworfen wie Kommentare; CLASS – in übersetzter Klasse vorhanden, während Programmausführung nicht sichtbar (für Werkzeuge, die auf Byte-Code operieren); RUNTIME – während Laufzeit zugreifbar

### 9. Wie kann man eigene Annotationen deklarieren? Welche Gemeinsamkeiten und Unterschiede zu Interfaces bestehen?

Siehe Frage 7)

### 10. Wie kann man zur Laufzeit auf Annotationen zugreifen?

Reflexion; Ausgangspunkt: Objekt vom Typ „Class“ – Methoden: getAnnotation(s), getMethod(s), getField(s), invoke(Method m)

### 11. Was ist aspektorientierte Programmierung? Wann setze ich sie sinnvoll ein?

**AOP - Aspektorientierte Programmierung**

Die AOP ist ein Programmierparadigma für die objektorientierte Programmierung, um die gleichen Funktionalitäten über mehrere Klassen hinweg zu verwenden (*Cross-Cutting Concern*) -> Ziel: Code Wiederverwendbarkeit

***Separation-of-Concerns***: Wir versuchen die Software immer nach der Funktionalität eines Programms auf einzelne Modularisierungseinheiten aufzuteilen.

Software hat grundsätzlich bestimmte Aufgaben zu erfüllen:

1. Die sogenannten ***Core-Level-Concerns – Kernfunktionalitäten***. Dies sind Anforderungen an die Software, die man meist gut in einzelnen Funktionen kapseln kann. Ein Beispiel wäre die Berechnung eines Wertes. Für diese Art der Faktorisierung eignet sich die OOP gut mit Kapselung in Klassen.
2. Die ***System-Level-Concerns*** (betreffen das gesamte System) oder technische Randbedingungen. Diese Anforderungen können nicht einfach gekapselt werden, da sie an vielen Stellen implementiert werden müssen. Ein Paradebeispiel dafür ist das *Logging*, die Protokollierung des Programmablaufs in Logdateien. Der Aufruf des Loggers ist für die eigentliche Funktionalität nicht notwendig, muss aber trotzdem in den Quelltext integriert werden. Ein weiteres Beispiel wäre die Transaktionierung von Zugriffen auf eine Ressource wie z. B. eine Datenbank.

Das Problem der miteinander verwobenen Anforderungen wird auch als Cross-Cutting Concerns bezeichnet, denn sie „schneiden“ quer durch alle logischen Schichten des Systems. AOP ist das Werkzeug, um die logisch unabhängigen Belange auch physisch voneinander zu trennen. Dabei wird angestrebt, Code zu erzeugen, der besser wartbar und wiederverwendbar ist.

* **Cross-Cutting Concerns – Querschnittfunktionalitäten :** querschnittliche Belange einer Software meist sind es nichtfunktionale Anforderungen an Software wie etwa Sicherheitsaspekte, die bei konventioneller Programmierung quer verstreut über den gesamten Code realisiert werden – beispielsweise immer wiederkehrende Prüfungen der Form „darf dieser Code gerade ausgeführt werden?“. Die Aspektorientierte Programmierung (AOP) bietet die Möglichkeit, solchen Code zentral zu formulieren.

Sinnvoller Einsatz: Bestimmte Aktionen im Programm zuverlässig in einer Log-Datei protokollieren, ohne den Source-Code des Programms dafür ändern zu müssen, die Generierung bestimmter Debug-Information veranlassen und diese auch leicht wieder wegnehmen.

**Allgemeine Definition eines Aspekts**

Die aspektorientierte Programmierung kapselt Verhalten, das mehrere Klassen betrifft, in Aspekten.

Ein Aspekt beschreibt dabei:

* eine Funktionalität
* alle Stellen im Programm, an denen diese Funktionalität angewendet werden soll

**AspectJ**

Erweitert Java Sprache mit AOP und erleichtert zB auch Debugging.

Es gibt aber für fast alle anderen Programmiersprachen ebenso aspektorientierte Erweiterungen (z.B. AspectL für Lisp und LOOM.NET für C#).

Wie Meta-Programmierung, nur mit großen Einschränkungen. Es werden alle Stellen im Code mit einer Aspect Datei angesprochen zu denen Code-Snippets hinzugefügt werden sollen.

Ist Open-Source aber von Eclipse entwickelt und ist verfügbar als Zusatzmodul in der Eclipse IDE. Das Programmierframework Spring wird gerne in Kombination mit AspectJ verwendet.

**join point** Ausführungspunkt in einem Programm

z.B. der Aufruf einer Methode oder der Zugriff auf ein Objekt

**pointcut** Wählt joint point sund sammelt kontextabhängige Information dazu

z.B. die Argumente eines Methodenaufrufs oder das Zielobjekt

**advice** Programmcode der vor (before()), um (around()) oder nach (after()) dem Join-Point ausgeführt wird

**aspect** Ein Aspekt ist wie eine Klasse das zentrale Element in AspectJ

Enthält alle Deklarationen, Methoden, Pointcuts und Advices

01 public class Test{

02 Methodenausf. public static void main(String[]a) {

03 Konstruktoraufruf Point pt1 = new Point(0,0);

04 Methodenaufruf pt1.incrXY(3,6);

05 }

06 }

07

08 Klasseninit public class Point {

09 Objektinit private int x;

10 private int y;

11 public Point(int x, int y) {

12 Feldzugr.(write) this.x = x;

13 this.y = y;

14 }

15 public void incrXY(int dx, int dy){

16 Feldzugr.(read) x = this.x + dx;

17 y =+ dy;

18 }

19 }

In Zeile 02 ist ein möglicher Join-Point die Ausführung des Beginns der Methode main

In Zeile 03 der Aufruf des Konstruktors Point

In Zeile 12 der schreibende Zugriff auf des Feld this.x

Ein Pointcut definiert Join-Points. Es können anonyme Pointcuts definiert werden, meistens werden aber benannte Join-Points verwendet. Die Syntax sieht folgendermaßen aus, wobei Signatur einfach nur die syntaktische Darstellung eines Join-Points im Java-Code ist:

[Sichtbarkeit] pointcut Name ([Argumente]) : Pointcuttyp(Signatur)

**Joint-Point-Punkte im Programm**

* Konstruktor (Ausführung, Aufruf)
* Feldzugriff (schreibend/lesend)
* Initialisierung einer Klasse Objektinitialisierung (Konstruktor, (pre-)initialization) Exceptions(handler)
* …

### 12. Was bedeutet Separation-of-Concerns?

Siehe Frage 11)

### 13. Was sind Core-Concerns, was Cross-Cutting-Concerns?

Siehe Frage 11)

### 14. Was sind Join-Points, Pointcuts, Advices und Aspekte, und wozu braucht man sie?

Siehe Frage 11)

15. An welchen Programmpunkten können sich Join-Points befinden?Siehe Frage 11)

# Kapitel 5: (14 Fragen)

### 1. Erklären Sie folgende Entwurfsmuster und beschreiben Sie jeweils Anwendungsgebiet, Struktur, Eigenschaften und wichtige Details der Implementierung unter Verwendung vorgegebener Namen:

Siehe Zusammenfassung Kapitel 5

### 2. Welche Arten von Iteratoren gibt es, und wofür sind sie geeignet?

**Interne und externe Iteratoren**

Interne Iteratoren (Streams oder *map* aus Haskell)

Man übergibt dem Iterator eine Operation (z.B. in Form einer Methode oder Funktion), die auf alle Elemente des Aggregats gemapped wird. zB Stream-Operationen oder ein Aufruf der Methode forEach in einer Map.

* Kontrolliert selbst wann nächste Iteration erfolgt / enthält die Schleife selbst, von außen sind Methoden wie next und hasNext nicht zugänglich. **->** Einfacher weil man keine Anwendung / Logik für für Schleifenposition braucht.
* Gut in funktionalen Sprachen umsetzbar, auch mit Lambda Ausdrücken und Streams in Java. -> Gute parallele Verarbeitung von großen Datenmengen.
* Sinnvoller wenn die Beziehungen zwischen den Elementen (zB Position im Baum) erhalten werden soll, da sie bei externen Iteratoren verloren gehen. Das kann aber Parallelisierung erschweren.

Externe Iteratoren (bekannt aus EP2)

In objektorientierten Sprachen gut umsetzbar.

* Anwender bestimmt wann nächste Abarbeitung erfolgt.
* Einsatzmöglichkeiten flexibler weil man mit ihnen zwei Aggregate miteinander vergleichen kann.
* Schlechter für komplexe Beziehungen, schwieriger zu steuern.

### 3. Wie wirkt sich die Verwendung eines Iterators auf die Schnittstelle des entsprechenden Aggregats aus?

Die Schnittstelle bleibt immer gleich.

Iteratoren stellen Zugriffsmöglichkeiten bereit, diese muss Schnittstelle nicht mehr selbst unterstützen.

### 

### 4. Inwiefern können geschachtelte Klassen bei der Implementierung von Iteratoren hilfreich sein?

Mit Hilfe von geschachtelten Klassen kann man einen Iterator in eine Containerklasse integrieren -> direkte Zugriffe.

Sie haben ermöglichen Zugriff auf private Inhalte von Klassen (mehr Implementierungsdetails).

Dies erhöht aber die Klassenabhängigkeit zwischen Aggregat und Iterator noch weiter.

### 5. Was ist ein robuster Iterator? Wozu braucht man Robustheit?

* Es kann gefährlich sein, ein Aggregat zu verändern, während es von einem Iterator durchwandert wird. Es passiert leicht, dass Elemente doppelt oder gar nicht abgearbeitet werden.

Lösung 1: Aggregat kopieren und über Kopie iterieren

Lösung 2: ***robusten Iterator*** implementieren (schwierig). Ermöglicht das Verändern eines Aggregats (Zugriff mittels Iterator), während es von einem (anderen) Iterator durchlaufen wird ohne es vorher zu kopieren.

* oft praktisch, Iteratoren auch auf leeren Aggregaten bereitzustellen (auch bei internen Iteratoren, damit keine Exceptions auftreten. In Java: flatMap, wie Map nur keine Sonderbehandlung für leere Streams notwendig)

### 6. Wird die Anzahl der benötigten Klassen im System bei Verwendung von Factory-Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?

**Anzahl der benötigten Klassen und Objekte im Vergleich zu einem System, dass keine Patterns verwendet**

Factory-Method / Virtual-Constructor

Anzahl der Klassen nimmt zu, da parallel zur eigentlichen Klassenhierarchie eine parallele Creator‐Klassenhierarchie geführt werden muss.

Anzahl der benötigten Klassen: steigt

Anzahl benötigter Objekte gleich (lediglich „Ort“ der Objekterzeugung verschoben)  
 Anzahl der benötigten Objekte: bleibt gleich

Prototype  
 unverändert aber im Vergleich zu Factory deutlich vermindert (keine Parallelstrukturen)

Anzahl der benötigten Klassen: bleibt gleich

unverändert (in bestehende Objekte integriert) abe reher erhöht, da Prototypes, die nicht zwingend verwendet werden, auch erstellt werden  
 Anzahl der benötigten Objekte: steigt evtl. leicht

Singleton

Visitor Pattern

Iterator / Cursor

Template-Method

Decorator / Wrapper  
 Verringert die Anzahl der Klassen, da man durch das einfache Erweitern von Objekten mittels Dekoratoren eventuelle Unterklassen einsparen kann.

Anzahl der benötigten Klassen: steigt

erhöht, da Decorator-Objekte erstellt werden, die mit dem Basis-Objekt „verknüpft“ werden (Es existiert jeweils ein Objekt mit Erweiterung und eines ohne, viele kleine Objekte)  
 Anzahl der benötigten Objekte: steigt

Proxy / Surrogate

Für alle Objekte auf die mit einem Proxy zugegriffen werden soll, sind zusätzliche Proxy‐Klassen notwendig

Anzahl der benötigten Klassen: steigt leicht

erhöht, da für jedes zusätzliche Zugriffsverwaltungsfeature weiteres Proxy-Objekt nötig (Verkettung)  
 Anzahl der benötigten Objekte: Bleibt gleich (Objekterzeugung nicht nötig) oder steigt

### 7. Wird die Anzahl der benötigten Objekte im System bei Verwendung von Factory-Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?

Siehe Frage 6)

### 8. Vergleichen Sie Factory-Method mit Prototype. Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?

**Im Vergleich: Factory Method und Prototype**

Factory Method: Delegiert Erzeugung an Unterklassen

* mühsam die Struktur zu erweitern
* Jede Klasse braucht eine Erstellerklasse -> Anzahl der Klassen nimmt schnell zu, deshalb muss sie gering gehalten werden

Prototyp: Erzeugt Kopien von deklarierten Typen

* mühsam die neu erzeugten Typen zu verwalten
* Benötigt keine neuen Klassen, da in jeder Klasse eine entsprechende Kopiermethode besteht.
* Allerdings ist die Implementierung einer Methode, welche tiefe Kopien erzeugt meist sehr aufwändig.

Generell ist die Factory Method wohl eher geeignet, wenn man genau weiß was man will und Initialisierungslogik vor Client verstecken will, Prototype hingegen bietet sich in sehr „dynamischen“ Systemen an und wenn Objekterzeugung teuer/aufwändig.

### 9. Wo liegen die Probleme in der Implementierung eines so einfachen Entwurfsmusters wie Singleton?

**Herausforderungen bei Implementierung**

Will man mehrere verschiedene Untertypen von Singletons anbieten, muss man trotzdem die Bedingung aufrechterhalten, dass immer nur eine Instanz der Oberklasse Singleton existent ist. Dies lässt sich meist nur mittel switch‐ Anweisungen oder ähnlichen Sprachkonstrukten erreichen -> Schlechte Wartbarkeit und Flexibilität.

Jede Variante hat eigen Nachteile (feste Verdrahtung der Alternativen durch switch, Implementierung von instance() in Untertypen – alle Unterklassen müssen Methode korrekt implementieren, Verwendung von kleiner Datenbank – Eintrag neuer Alternativen nicht in Verantwortung von Singleton, usw) -> Entweder der Obertyp kennt alle Untertypen, oder die Untertypen müssen alle instance() implementieren.

### 10. Welche Unterschiede und Ähnlichkeiten gibt es zwischen Decorator und Proxy?

**Unterschiede zwischen Decorator und Proxy**

Ein Proxy kann die gleiche Struktur wie ein Decorator haben. Aber Proxies dienen einem ganz anderen Zweck:

Ein Decorator / Wrapper:

* erweitert ein Objekt um zusätzliche Verantwortlichkeiten
* Er kann einem Objekt Funktionalitäten hinzufügen, aber auch wieder entziehen.
* Decorator erzeugt Objekt nicht.

Proxy / Surrogate:

* Kontrolliert Zugriff auf das Objekt

Er stellt praktisch ein Bindeglied zwischen dem „realen Objekt“ und seinem Surrogat dar.

* Proxy kann Objekt erzeugen falls nötig

### 11. Welche Probleme kann es beim Erzeugen von Kopien im Prototype geben? Was unterscheidet flache Kopien von tiefen?

**Clone() in Java**

Um die Verwendung dieses Entwurfsmusters zu fördern, haben die Entwickler von Java die Methode clone bereits in Object definiert -> muss überschrieben werden und das zu klonende Objekt muss das Interface cloneable implementieren:

* *flache Kopien*: (default Implementierung) Variablenwerte *identisch* mit dem Wert in der Kopie (gleiche Referenz)
* *tiefe Kopien*: alle Variablenwerte einzeln kopieren, sodass geklontes Objekt *gleiche* Zustände hat

Probleme bei flacher Kopie:

Eine flache Kopie legt nur Referenzen auf die Variablen eines Objekts an (der Wert jeder Variable in der Kopie ist identisch mit dem Wert der entsprechenden Variable im originalen Objekt). Dadurch ist neues Objekt abhängig vom alten.

Probleme bei tiefer Kopie:

Wenn man zum Beispiel das überschriebene clone() aber einfach nur rekursiv auf zyklische Strukturen anwendet, entsteht ein Stackoverflow.

### 12. Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)

Dekoratoren eignen sich gut dazu, die Oberfläche beziehungsweise das Erscheinungsbild eines Objekts zu erweitern. Sie sind nicht gut für inhaltliche Erweiterungen geeignet. Auch für Objekte, die von Grund auf umfangreich sind, eignen sich Dekoratoren kaum.

### 13. Kann man mehrere Decorators bzw. Proxies hintereinander verketten? Wozu kann so etwas gut sein?

**Verkettung von mehreren Decorators**

Bei Decorators könnte man etwa zu einem Objekt einen Rahmen mittels eines entsprechenden Decorators hinzufügen, und diesen Rahmen vorher mit Hilfe eines anderen Decorators grün färben.

Bei Decorators können auf diese Art und Weise z.B. umfangreiche GUIs gebildet werden.

**Verkettung von mehreren Proxies**

Bei Proxies würde es zum Beispiel Sinn machen, ein Virtual Proxy zu verwenden, um die Erzeugung eines Objekts zu verzögern, und dieses mit einem Protection Proxy zu verknüpfen, um etwaige Zugriffsbeschränkungen zu implementieren.

Beliebige Möglichkeiten, wie z.B. das Kapseln mehrerer Funktionalitäten.

### 14. Was unterscheidet Hooks von abstrakten Methoden?

Hooks sind Operationen mit Default-Verhalten (in Interface oder abstrakter Klasse definiert), das bei Bedarf überschrieben oder erweitert werden kann; abstrakte Methoden müssen überschrieben werden