

# OOP

## Kapitel 1

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

Denkweise / Weltanschauung / Programmierstile mit bestimmten Eigenschaften.

Übersicht:

- Imperatives Paradigma (baut auf Maschinenbefehlen auf)
  - Prozedurales Paradigma
  - Objektorientiertes Paradigma
- Deklaratives Paradigma (baut auf formalen Modellen auf)
  - Funktionales Paradigma
  - Logikorientiertes Paradigma

### 2. Wozu dient ein Berechnungsmodell?

- Ist die Grundlage jedes Paradigmas. Muss konsistent und turing-vollständig sein (also es muss alles ausdrücken können, was als berechenbar gilt).

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

- ???
- Funktionen:
  - primitiv-rekursive Funktionen (Funktionen setzen sich aus anderen Funktionen oder sich selbst (Rekursion) zusammen)
  - $\mu$ -rekursive Funktionen → für Turing-Vollständigkeit
- Prädikatenlogik
- Constraint-Programmierung
- Temporale Logik & Petri Netze (lassen sich jeweils ineinander umwandeln)
- Freie Algebren
- Prozesskalkül
  - CSP (Communicating Sequential Processes)
  - $\pi$ -Kalkül (eignet sich für reaktive Systeme, die auf Ereignisse reagieren)
- Automaten
- While, Goto usw. (haben eine starke Verbindung zur imperativen Programmierung)
  
- Eigenschaften
  - konsistent
  - turing-vollständig

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

- Kombinierbarkeit (bestehende Programmteile zu größeren hinzufügen)
- Konsistenz (1 Formalismus reicht nicht -> mehrere -> alle Konzepte einer Sprache ineinander konsistent)
- Abstraktion (portabel, wichtig, nimmt stetig zu)
- Systemnähe (Vorteil: einfacher Hardwarezugriff, Nachteil: weniger portabel, Security)
- Unterstützung (durch Programmiersprachen, IDEs, Tools usw.)
- Beharrungsvermögen (Killerapplikation = so überzeugend, dass andere das Paradigma übernehmen)

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

- Flexibilität und Ausdruckskraft
- Lesbarkeit und Sicherheit
- Konzepte verständlich, klar was machbar was nicht

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

Weiterentwicklung der prozeduralen Programmierung  
3 Kontrollstrukturen

- Sequenz (ein Schritt nach dem anderen)
- Auswahl (if)
- Wiederholung (Schleife / Rekursion)

Jede Struktur hat nur einen Einstiegs- und Ausstiegspunkt. Kein Goto!

7. Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?

- versteckte Seiteneffekte
- referentielle Transparenz anstreben (d.h. man kann Funktionen/Variablen durch deren Wert ersetzen ohne Seiteneffekte)
- Dokumentation
- keine Seiteneffekte

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

- keine Seiteneffekte
- Funktionen ersetzbar durch Aufruf, der dasselbe retourliefert
- $f(x)+f(x) = 2*f(x)$
- Variablen ersetzbar durch deren Wert

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

- weil Ein- / Ausgaben Seiteneffekte sind und Programmfortschritt mit Seiteneffekten erreicht wird
- in FP: I/O nur ganz oben in Aufrufhierarchie → referentielle Transparenz überall sonst gewährleistet

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

- man muss annehmen, dass welche existieren
- Querverbindungen bestehen oft nur lokal (in einem Objekt / einer Klasse)
- Änderungen auf Zustandsebene einzelner Objekte erkennbar → man muss nicht den ganzen Programmzustand überblicken können (prozedural)

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

- uneingeschränkt verwendbare Konzepte
- bei FP: Funktionen
- viel komplizierter, da viele Sonderfälle
- für manche Konzepte nötig um darauf Paradigmen aufzubauen

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

- "Schablonen für Programmteile schreiben" -> mit Funktionen füllen, nur mit Funktionen höherer Ordnung möglich

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

Modul:

- klein
- Klasse / Interface
- in einem Stück vom Compiler verarbeitet
- schnell übersetzt
- keine Zyklen
- Zyklen auflösen: Trennung Schnittstelleninformationen / Implementierung

Objekt:

- idR zur Laufzeit erzeugt
- Kapselung: Variablen & Methoden zu logischen Einheiten
- Data-Hiding: private Inhalte verstecken
- praktisch immer First-Class-Entities
- Identität, Zustand, Verhalten --> gleich <-> identisch

Klasse:

- Bauplan / Schablone für Objekt
- Variablen & Methoden, nicht Werte
- in Java (oft) Modul = Übersetzungseinheit
- abstrakte Klassen / Interfaces -> zyklische Abhängigkeiten auflösen

Komponente:

- ähneln Modulen, beides Namensräume und Übersetzungseinheiten
- ein eigenständiges Stück Software welches eingebunden werden kann
- nicht alleine lauffähig -> setzt Existenz anderer Komponenten voraus
- offen wo importierte Inhalte herkommen -> zyklische Abhängigkeiten kein Problem

Namensraum:

- jede Modularisierungseinheit ist ein Namensraum
- → Namenskonflikte abfedern
- globale Namen!
- zusammengefasste Modularisierungseinheiten

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

- Änderungen an privaten Inhalten wirkt sich nicht auf andere Objekte aus
- von außen zugreifbare über Schnittstelle definiert

15. Was ist und wozu dient ein Namensraum?

- siehe 13

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

- Modul:  $A \leftarrow B \rightarrow A$  muss vor B übersetzt werden und B muss vor A übersetzt werden
- Komponente: offen wo importierte Inhalte herkommen

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

- Abstraktion: Kapselung + Data Hiding
- Kapselung: zusammenfassen von Variablen und Methoden zu logischen Einheiten
- Data Hiding: verstecken von privaten Inhalten

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

- Komponenten von denen importiert wird muss zusätzlich angegeben werden

19. Wie kann man globale Namen verwalten?

- z.B. Pakete in Java

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

- Lücken lassen, die man später befüllt
- Steigerung von Flexibilität
- Wann:
  - zur Laufzeit
  - 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?

- zirkuläre Abhängigkeiten von Objekten
- Erzeugung von Objekten durch Kopien

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

- für statische Modularisierung einsetzbar -> bereits beim Übersetzen einfügbar

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

- Art der Parametrisierung wo Lücken bereits zur Übersetzungszeit gefüllt werden
- keine First Class Entities Lücken
- grundsätzlich einfach, und erprobt, mit Einschränkungen aber kompliziert da keine First Class Entities (?)

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

- Optionale Parameter die zu Sprachkonstruktion hinzugefügt werden können
- z.B. @Override
- statisch (Übersetzungszeit) / dynamisch (Laufzeit) abfragbar
- nur für statische Informationen geeignet

25. Was versteht man unter aspektorientierter Programmierung?

- hinzufügen von Aspekten, kein füllen von Lücken
- Aspekt = Menge von Punkten im Programm + was an diesen Stellen passieren soll  
--> Aspect-Weaver -> einfügen der Aspekte vor Übersetzung
- sehr leicht Sachen hinzufügen / entfernen
- Problem: manchmal Implementierungsdetails vorausgesetzt

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

- praxistaugliche nachträgliche Ersetzbarkeit von Modularisierungseinheiten

27. Wann ist A durch B ersetzbar?

- wenn ein Austausch von A durch B keine Änderungen nach sich zieht

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

- ziemlich kompliziert im Detail
- Übereinstimmung von Signaturen
- Abstraktion realer Welt (zB Fahrrad ist kein Auto)
- Zusicherungen

29. Was ist die Signatur einer Modularisierungseinheit?

- Inhalte auf die von außen zugegriffen werden kann
- beschrieben durch Namen (+ Typen) von Parametern und Ergebnissen

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

- Signatur und Abstraktion müssen passen um Ersetzbarkeit zu Gewährleisten
- Auto → Fahrzeug ← Fahrrad  
→ Fahrzeug durch Auto ODER Fahrrad ersetzbar  
aber Auto nicht durch Fahrrad oder umgekehrt ersetzbar

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

- genaue Beschreibung der Erwartungen an eine Modularisierungseinheit
- Design-by-Contract is the buzzword of choice
- Server <-> Client
- Vor / Nachbedingungen, History-Constraints (Server / Client controlled)
- Invarianten
- Beispiele !!

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

- wenn Typen der Operanden mit der Operation zusammenpassen
- wenn nicht dann Typfehler

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

- statische Prüfung -> Fehler zur Übersetzungszeit wenn Konsistenz nicht gegeben
- dynamische Prüfung → Fehler zur Laufzeit wenn Konsistenz nicht gegeben

- warum? Zuverlässigkeit. einfacheres lesen und verstehen von Programmen

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

- Zuverlässigkeit (pseudo - nur "leichte" Fehler werden gefunden)
- einfacheres lesen & verstehen
- nicht zu komplex: Compiler nicht überfordern

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

- automatisches erkennen des Typs aus dem Kontext
- Lesbarkeit kann leiden, Übersetzungszeit auch

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

- Sprachdefinition & Sprachimplementierung
- Erstellung von Übersetzungseinheiten
- Einbindung von Modulen / Komponenten
- Compiler
- Initialisierung / Programmausführung

37. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?

- wie viel Speicher reserviert werden muss
- welcher Wertebereich möglich ist
- zB: int → 32 bit  
Generizität → Typparameter → Referenz
- je früher desto weniger zur Laufzeit → weniger Fallunterscheidungen
- Typfehler und daraus folgende Fehler früh erkannt → nicht so schwerwiegend

38. Wie beeinflussen Typen die Planbarkeit weiterer Schritte?

- frühe Entscheidungen erleichtern Planbarkeit
- man muss sich nicht überlegen welche Werte (... Typen) eine Variable annehmen kann, sondern sieht es durch den Typ
- Typen nicht immer durch das ganze Programm planbar
  - → müssen manchmal geändert werden
  - → statische Typprüfung hilft hierbei sehr
- in dynamischen Sprachen oft: Variablenname lässt auf Typ schließen

39. Was ist ein abstrakter Datentyp?

- Trennung der Innensicht & Außensicht & Kapselung
- hinter der Modularisierungseinheit steht somit ein Konzept
- abstrakter Datentyp = Schnittstelle einer Modularisierungseinheit
- Signatur = Schnittstelle / Typ

- -> struktureller Typ (Namen / Parametertypen / Ergebnistypen)
- Abstraktion = Typ steht für Konzept in realer Welt
- strukturelle Typen können zufällig selbe Signatur haben
  - -> Lösung: nicht verwendeten Inhalt mitgeben, dessen Name das Konzept beschreibt
- Theorie: strukturelle Typen
- Praxis: nominale Typen, Abstraktion wichtig, denken in abstrakten Konzepten
- Oft komplexe Konzepte, aber man kann auf int zB als abstrakten Datentyp stehen
  - -> verschleiert zB die Repräsentation in der Maschine

40. Was unterscheidet strukturelle von nominalen Typen?

\* Nominale Typen

\* Signatur

\* eindeutiger Name (bei Objekt = Name der Klasse)

\* 2 nominale Typen äquivalent, wenn Namen übereinstimmen

\* Strukturelle Typen

\* Äquivalent wenn Signatur übereinstimmt

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

\* Praxis: nominale Typen, da Abstraktion von großer Bedeutung, man denkt an abstrakte Konzepte nicht Signaturen

\* Theorie: weil man die in 39 beschriebene Lösung benutzen kann zur eindeutigen Unterscheidung

42. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?

\* ohne Ersetzbarkeit keine Untertypbeziehungen

\* U ist Untertyp von T, wenn U überall verwendet werden kann, wo T erwartet wird

43. Warum kann ein Compiler ohne Unterstützung durch Programmierer(innen) nicht entscheiden, ob ein nominaler Typ Untertyp eines anderen nominalen Typs ist?

\* Compiler hat kein Wissen über die abstrakten Konzepte

\* Kompatibilität von Konzepten

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

\* Typen von Funktions- bzw. Methoden-Parametern dürfen in Untertypen nicht stärker werden

\* zB: U Untertyp von T, `compare(T t)` darf nicht durch `compare(U u)` überschrieben werden, obwohl oft benötigt

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

- \* Subtyping -> OOP
- \* Generizität -> bei statischer Typprüfung
- \* F-gebundene Generizität
  - > Untertypbeziehungen zur Einschränkung von generischen Typparametern  
zB in Java / C#
- \* High-Order-Subtyping (Matching)
  - > Higher- Order-Subtyping, auch Matching genannt geht einen eher direkten Weg und beschreibt Einschränkungen über Untertyp-ähnliche Beziehungen, die wegen Unterschieden in Details aber keine Untertypbeziehungen sind. Dieser Ansatz wird auf unterschiedliche Weise beispielsweise in C++, aber auch in der funktionalen Sprache Haskell verwendet.
  - > = ???

46. Wie konstruiert man rekursive Datenstrukturen?

- \* induktiv
- \* endliche Menge  $M_0$ , die einfache Werte enthält
- \* endliche viele Möglichkeiten aus  $M_i \rightarrow M_{i+1}$  zu generieren ( $i \geq 0$ )
- \* Haskell: `data Lst = end | elem(Int, Lst)`

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

- \* die Grundlage der Fundierung ist  $M_0 \rightarrow$  darf nicht leer sein
- \* "Ende der Rekursion"
- \* Java: `null`

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

- \* wenn gleichzeitig Ersetzbarkeit durch Untertypen verwendet wird, dann funktioniert Typinferenz nicht (an derselben Stelle)
- \* (Typinferenz und Generizität schon)

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

- \* Funktion: Kompatibilität von Parameter und Operand
- \* Information wird propagiert, nicht nur von Typen, sondern für alle statisch bekannten Eigenschaften

50. Erklären Sie folgende Begriffe:

- Objekt, Klasse, Vererbung
- \* Objekt
  - \* Methoden und Variablen gekapselt
  - \* zur Laufzeit
  - \* Modularisierungseinheit

- \* Identity / State / Behaviour

- \* Klasse

- \* nicht in allen OOP Sprachen!
- \* gibt Struktur von Objekten vor
- \* neue Objekte über Konstruktoren erzeugbar

- \* Vererbung

- \* neue Klassen aus existierenden Klassen ableiten
- \* spart Schreibaufwand
- \* vereinfacht Änderungen
- \* nur erweitern / überschreiben in Unterklasse

- Identität, Zustand, Verhalten, Schnittstelle

- \* Identität

- \* für eindeutige Bestimmung von Objekt (ansprechen um Nachrichten zu senden)
- \* vereinfacht: Speicheradresse

- \* Zustand

- \* zusammengesetzt aus Werte der Variablen eines Objekts
- \* Gleichheit / Equality

- \* Verhalten

- \* beschreibt wie sich Objekte beim Erhalten einer Nachricht verhalten
- \* Stack: push / pop -> hinzufügen / entfernen

- deklarierter, statischer und dynamischer Typ

- \* deklarierter Typ: Typ einer Variable bei expliziter Typdeklaration

- \* dynamischer Typ: der spezifischste Typ (= tatsächliche) Typ einer Variable

- \* statischer Typ: zwischen deklariertem und dynamischen Typ, von Compiler bestimmt, kann an Stellen unterschiedlich sein für dieselbe Variable, zur Optimierung verwendet

- Faktorisierung, Refaktorisierung

- \* Faktorisierung: Zerlegung eines Programmes in zusammengehörige Einheiten

- \* Refaktorisierung: Entwurf auf Anheb nicht optimal -> verbessern = Refaktorisierung

- Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung

- \* Verantwortlichkeiten (einer Klasse)

- \* was weiß ich (Beschreibung des Zustands der Objekte)
- \* was mach ich (Verhalten der Objekte)
- \* wen kenne ich (sichtbare Objekte / Klassen)

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

- \* Polymorphie: wenn eine Variable oder Methode gleichzeitig mehrere Typen haben kann
- \* Ad-Hoc
  - \* überladen
  - \* Typumwandlungen
- \* universeller
  - \* Generizität
  - \* Untertypen

\* in OOP hauptsächlich Untertypen

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

- \* 2 gleiche Objekte identisch: wenn sie dieselbe Identität haben? Lul
- \* 2 identische Objekte gleich: immer, da die Identität das Objekt eindeutig bestimmt und es sich somit um dasselbe Objekt handelt.
- \* Gleichheit wird mittels equals überprüft
- \* Identität wird mittels == überprüft, bzw in anderen Sprachen oft ===

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

\* siehe Frage 17

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

\* Ein Typ U ist Untertyp eines Typs T, wenn jede Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird.

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

\* Code Wiederverwendung

56. Wann und warum ist gute Wartbarkeit wichtig?

\* Money Money Money (Wartungskosten hoch)

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

\* KZ hoch, OK niedrig

\* gute Faktorisierung, höhere Abstraktion & besseres Hiding, Programmänderungen lokaler

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

- \* Programme
- \* Erfahrungen (Entwurfsmuster)
- \* Code
- \* Daten
- \* Globale Bibliotheken
- \* Fachspezifische Bibliotheken
- \* Projektinterne Wiederverwendung
- \* Programminterne Wiederverwendung

59. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?

- \* hinführen auf eine gute Faktorisierung & stabiles Design
- \* ändert Struktur aber belässt Funktionalität

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

- \* Entwicklung von Systemen deren Gesamtkomplexität die Komplexität einzelner Algorithmen deutlich übersteigt
- \* sonst ned

## Kapitel 2

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

- Untertypbeziehungen (bestehender Code bekommt einen Untertyp und muss nicht geändert werden)
- direkte Codewiederverwendung (Vererbung)

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!)

- Untertypbeziehungen
  - Reflexivität (T Untertyp von T)
  - Transitivität (T von U und U von V  $\rightarrow$  T von V)
  - Antisymmetrie (T von U und U von T  $\rightarrow$  T == U)
- U Untertyp von T wenn
  - Für jede Konstante in T gibt es diese auch in U, und sie muss ein Untertyp sein
  - Für jede Variable in T gibt es eine entsprechende Variable in U, wobei die deklarierten Typen der Variablen äquivalent sind (weil lesen und schreiben  $A \rightarrow B$  &  $B \rightarrow A$  voraussetzt)

- gleichnamige Methode existiert
  - Ergebnistyp in Unterklasse darf Untertyp sein
  - Anzahl Parameter gleich
  - deklarierte Typ von Parameter in U darf Obertyp in Parameter in T sein
- private Inhalte haben keinen Einfluss auf Untertypenbeziehungen
- Java
  - extends / implements zusätzlich
  - grundsätzlich alle Typen invariant, Rückgabewerte von Methoden können kovariant sein.  
(sonst überladen, statt überschreiben)

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?

- Zusicherungen

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

- Kovarianz: deklarierte Typ eines Elementes im Untertyp ist Untertyp desselben Elementes im Obertyp
  - Konstanten
  - Ergebnisse von Methoden
  - Ausgangsparameter von Methoden
- Kontravarianz: deklarierte Typ eines Elementes im Untertyp ist Obertyp desselben Elementes im Obertyp
  - Eingangsparameter
- Invarianz: deklarierte Typ eines Elementes im Untertyp ist äquivalent zum selben Element im Obertyp
  - Variablen

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

- formaler Parametertyp ist gleich der Klasse = binäre Methode:
  - `X { ... equals(X x) {...} }`
- Kovariante Eingangstypen und binäre Methoden widersprechen Ersetzbarkeitsprinzip

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

- Parametertypen vorausschauend & möglichst allgemein wählen

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

- bessere Lesbarkeit
- einfacher erweiterbar, da nur neue Klasse erstellt werden muss nicht alle if & switch Anweisungen erweitert werden müssen

8. Dient dynamisches Binden der Ersetzbarkeit und Wartbarkeit?

- Ja, die Wartbarkeit wird verbessert → einfacher erweiterbar
- es kann jederzeit auch ein Untertyp verwendet werden ohne zusätzlichen Aufwand

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

- Vorbedingung (Client)
- Nachbedingung (Server)
- Invariante (Server, Clients für direkte Schreibzugriffe verantwortlich)
- History-Constraint
  - Server-kontrolliert: wie Invarianten, schränken aber zeitlichen Ablauf ein (Server, wenn Clients betroffene Variablen direkt beschreiben können, dann auch Clients)
    - Bsp: Counter
  - Client-kontrolliert: Reihenfolge von Methodenaufrufen, zB initialize (Methodenaufrufe erfolgen nur durch Clients, also Clients für Einhaltung verantwortlich)

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

- Vorbedingungen: **schwächer oder gleich** in Untertypen (oder-Verknüpfung)
- Nachbedingungen: **stärker oder gleich** in Untertypen (und-Verknüpfung)
- Invarianten: **stärker oder gleich** in Untertypen
- History-Constraint Server-kontrolliert: **stärker/gleich**, jedoch von konkreter Formulierung der Bedingungen abhängig
- History-Constraint Client-kontrolliert: **schwächer/gleich**, jedoch bezogen auf Einschränkungen in der Reihenfolge von Methodenaufrufen (Trace)

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

- für bessere Wartbarkeit

- wenn Programmänderungen nur lokal sind und die Schnittstellen der Klasse stabil bleiben, ist nur die Klasse selbst betroffen, aber nicht die Unterklassen
- an der Wurzel ist Stabilität besonders wichtig

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 und somit Invarianten nicht gewährleisten → deshalb sollen Objektvariablen möglichst nicht durch andere Objekte verändert werden

13. Wie genau sollen Zusicherungen spezifiziert sein?

- unmissverständlich formuliert
- nicht zu viele Details für bessere Wartbarkeit (Objektkopplung minimieren, Klassenzusammenhalt maximieren)

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

- Abstrakte Klassen:
  - Beschreibung der Struktur
  - keine Instanz erstellbar
  - um einen einheitlichen Obertyp zu erstellen
  - können abstrakte Methoden und vollständige implementierte Methoden enthalten
  - sind eher stabil als konkrete Klassen
  - Java:
    - abstract class
    - besondere Form sind Interfaces (dürfen keine Objektvariablen beschreiben)
- Abstrakte Methoden:
  - Methoden die nur Methodenrumpf aber keine Implementierung beinhalten (konkrete Unterklassen müssen Implementierungen bereitstellen)
  - zur Schnittstellendefinition im Obertyp

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

- Compiler kennt keine, ohne Kommentare auch nicht unterscheidbar
- Vererbung ist reine Code Wiederverwendung
- Bei dem Ersetzbarkeitsprinzip müssen jedoch die Zusicherungen hinsichtlich Ober und Unterklasse eingehalten werden.

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

- Vererbung hat hohe direkte Codewiederverwendung, aber ohne Ersetzbarkeit führt dies zu einer schweren Wartbarkeit
- Ersetzbarkeit erzielt Codewiederverwendung nicht im Untertyp, sondern in den Implementierung wo diese Typen verwendet werden, was zu einer hohen Codewiederverwendung und leichteren Wartbarkeit führt

17. Was bedeuten folgende Begriffe in Java?

• Objektvariable, Klassenvariable, statische Methode • Static-Initializer:

- Objektvariable
  - gehören zum Objekt, der Instanz
  - können verschieden sein zwischen Objekten
- Klassenvariable
  - gehören zur Klasse
  - sind gleich für alle Instanzen der Klasse, mit static gekennzeichnet
- statische Methode
  - gehören zur Klasse und über Klassennamen aufrufbar (Klasse A: A.x)
- static-Initializer
  - static {...}, dessen Inhalt (Codesequenz) vor der ersten Verwendung der Klasse ein einziges Mal ausgeführt wird. Kann zur Initialisierung der Klassenvariablen (statische Variablen) verwendet werden.

• geschachtelte und innere Klasse:

- geschachtelte Klasse
  - Klasse innerhalb einer Klasse
  - können überall definiert sein, wo Variablen deklariert werden dürfen
  - innerhalb geschachtelter Klassen sind private Variablen / Methoden verwendbar
  - statische geschachtelte Klasse
    - Zugriff auf (private) Klassenvariablen / statische Methoden (der umliegenden Klasse), aber nicht auf Objektvariablen / nichtstatische Methoden der darüberliegenden Klasse
    - in Objekten statisch geschachtelter Klassen sind Objektvariablen / nichtstatische Methoden normal zugreifbar
  - geschachtelte innere Klasse
    - Zugriff auf (private) Variablen & Methoden
    - dürfen keine statischen Variablen, Methoden oder Klassen beinhalten

• final Klasse und final Methode:

- Final Klasse
  - Klasse kann nicht überschrieben werden (i.e. extends nicht möglich)

- Final Methode
  - Methode kann nicht überschrieben werden (i.e. override nicht möglich)
- Paket, Class-Path, import-Anweisung:
  - Paket
    - Name des Verzeichnis in dem sich Klasse befindet, Namespace
  - Class-Path
    - Basis der Verzeichnisstruktur für Pakete
    - Klassen im `_Default Paket_` (beim Class-Path) haben keine package Anweisung
  - import-Anweisung
    - optional package Anweisung, dann import Anweisungen, sonst nirgendwo!
    - `import myclasses.examples.test`  
`test.MyClass.foo();`
    - `import myclasses.examples.test.MyClass`  
`MyClass.foo();`
    - `import myclasses.examples.test.*`  
`MyClass.foo();`

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

- (abstrakte) Klassen: Einfachvererbung
- Interfaces: Mehrfachvererbung (ist dann aber eigentlich nicht Vererbung sondern reines Subtyping)

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

- Datei importieren
- Klasse importieren
- alle Klassen in einem Paket importieren (mit `*`)
- → siehe Frage 2.17
- dienen um Schreibaufwand zu verringern

20. Wozu benötigt man eine package-Anweisung?

- Übersetzung geht nur, wenn package Name relativ zum Class-Path richtig ist
  - reines Rumkopieren geht nicht
  - verhindert "aus dem Kontext reißen" und dass die Datei nicht in einem anderen Paket verwendet wird

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

- default / private / protected / public
- default
  - nur im selben Paket sichtbar und vererbbar
- public
  - überall sichtbar
  - kann überschrieben werden bei Vererbung
  - Variablen nie, Methoden erlaubt
- protected
  - im selben Paket verfügbar
  - in anderen Paketen NUR mittels Vererbung verfügbar
- private
  - nur innerhalb derselben Klasse sichtbar
  - für Variablen und Methoden für interne Abläufe

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

- Interface = abstrakte Klasse, wo alle Methoden abstrakt sind und keine Variablen möglich sind
  - Ausnahme: default Implementierungen
- beginnen mit **interface** statt **abstract class**
- Methoden / Konstanten sind public, auch ohne Modifier
- Klassen können von mehreren Interface erben aber nur von einer Klasse (Interfaces unterstützen Mehrfachvererbung)

## Kapitel 3

1. Was ist Generizität? Wozu verwendet man Generizität?

- Generische Interfaces / Klassen / Typen / Methoden enthalten Parameter, für die ein Typ eingesetzt wird
- es wird ein Typparameter verwendet, der dann durch den übergebenen Typen ersetzt wird
- Generizität wird ab Java 1.5 unterstützt
- Programmstück muss nur einmal (allgemein) geschrieben werden und kann für mehrere Typen verwendet werden
- für Containertypen (Liste, Maps, Tables), Iteratoren

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

- gebundene Generizität ermöglicht Angeben von Schranken auf Typparameter (mittels extends: eine Klasse und beliebig viele Interfaces möglich; durch & getrennt)
- nur Untertypen der Schranken möglich

- bestimmten Typ (bzw. Untertyp) erwarten, der z.B. ein bestimmtes Interface implementiert und deshalb bestimmte Methoden enthalten muss

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

- wenn es die Wartbarkeit verbessert
- für gleich strukturierte Klassen / Methoden verwenden
- für Container (Listen, Stacks, ...) & Iteratoren
- für Methoden wie Such- und Sortierfunktionen
- abwägen gegen Subtyping

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

- Compiler überprüft ob Werte eingefügt werden können (String in List<Integer> geht nicht)
- dynamische Typabfragen: ??? Java ermöglicht getClass / instanceof
- Typumwandlungen: ???

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

- werden durch ein ? dargestellt, bieten mehr Flexibilität
- stehen für einen beliebigen Untertypen des angegebenen (Ober)Typs (wird kein (Ober)Typ angegeben, dann ist Object der (Ober)Typ)
- keine impliziten Untertypen: Y Untertyp von X, in List<X> kein Y einfügbar, da List<Y> kein Untertyp von List<X> ist
- <? extends Foo> kovariant → lesen erlaubt, Fragezeichen steht für beliebigen Typ, der Untertyp von Foo ist
- <? super Foo> kontravariant → schreiben erlaubt, Fragezeichen steht für beliebigen Typ, der Obertyp von Foo ist
- kann sehr kompliziert werden: <A extends Comparable<? super A>>
  - but why? Interfaces dürfen nur 1x implementiert werden, Subtyp kann also Foo<Y> implements Comparable<X> sein, was nicht funktionieren würde mit <A extends Comparable<A>>

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?

- Homogen <- Java
  - in genau eine Klasse umgewandelt
  - Typparameter durch unterste Schranke ersetzt, sonst Object
  - Compiler fügt Typecasts ein um die Rückgabewerte zu konvertieren
  - langsamer dafür speichersparender (nur 1 Klasse)
  - statisch sichergestellt, dass alle Elemente denselben Typ haben
- Heterogen
  - für jeden Typparameter einer generischen Klasse wird zur Compile-Zeit eine neue Klasse erstellt

- schneller aber speicherintensiver (viele Klassen mgl.)

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

- selbst eine homogene Übersetzung durchführen und die Typcasts hinzufügen
- statische Typsicherheit entfällt, Compiler erkennt nicht wenn man einen String in `List<Integer>` einfügen will

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

- homogene Übersetzung:
  - jede Klasse (generisch und nicht-generisch) wird in genau eine Klasse mit JVM-Code übersetzt
  - gebundene Typparameter werden durch erste Schranke des Typparameters ersetzt (wenn nicht vorhanden: durch `Object` ersetzt)
  - wird Parameter genommen oder zurückgegeben, dann wird er davor oder danach dynamisch in den Typ, der den Typparameter ersetzt, umgewandelt
- heterogene Übersetzung:
  - für jede Verwendung einer generischen Klasse / Methode (mit anderen Typparametern) wird eigener übersetzter Code erzeugt
  - entspricht "Copy&Paste"

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

- Wildcards → Ko- / Kontravarianz
- sonst: Invarianz

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

- **getClass**, gibt Klasse des Objekts zurück
- **instanceof**, gibt Bool-Wert zurück; `x instanceof Foo` auch true, wenn x eine Instanz von Bar ist und Bar extends / implements Foo

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

- der deklarierte Typ (Bsp: `(Point3D) p`)
- wie kommt man überhaupt auf was anderes? → Dynamischer Typ eines Objekts ist nicht veränderbar, statischer Typ vom Compiler erzeugt / geschätzt

12. Welche Gefahren bestehen bei Typumwandlungen?

- Laufzeitfehler

- es können Ausnahmebehandlungen auftreten
- falsche Ergebnisse
- falsche Daten in Datenbank abgespeichert werden
- statische Typüberprüfungen durch den Compiler werden ausgeschaltet, obwohl man sich eventuell noch auf statische Typsicherheit verlässt

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

- Dynamisches Binden kann dynamische Typabfragen vermeiden
- Generizität

14. Welche Arten von Typumwandlungen sind sicher? Warum?

- wenn in den Obertyp des deklarierten Objekttyps umgewandelt wird (Up-Cast)
  - sicher → da Untertyp überall verwendet werden kann, wo der Obertyp erwartet wird
- nach erfolgreicher instanceof / getClass Abfrage umwandeln (Alternativzweig bei Scheitern)
  - sicher → da der benötigte dynamische Typ sicher vorhanden ist
- manuelle homogene Übersetzung (aufwändig, unterliegt menschlichen Fehlern)
  - sicher → da unter richtiger Verwendung wie Generizität, jedoch sollte immer Generizität bevorzugt werden

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

- **kovariante Probleme:**
  - wenn man sich kovariante Eingangstypen wünscht
- **binäre Methoden:**
  - ein Eingangsparameter in Methode hat selben Typ wie die Klasse, welche die Methode enthält
  - spezialfall kovarianter Probleme
- vermeiden, da kovariante Eingangsparameter das Ersetzungsprinzip verletzen
- umgehen: Multimethoden (dynamische Argumenttypen)
  - bei  $y.x(z)$  wird  $z$  auch auf den spezifischsten Typ konvertiert und die entsprechende Methode aufgerufen, #Java does not support this
  - Multimethoden umgehen → Visitor Pattern (Multimethoden simulieren)

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

- Überschreiben = bei Vererbung selben Methodenkopf, aber den einen Methodenrumpf überschreiben
- Überladen = selber Methodename mit anderem Parametertyp (in Untertyp zB)
- Multimethoden = auch beim Parametertyp dynamische Binden einsetzen (Rind / Gras / Futter Bsp)

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

- Visitor Pattern: mehrfaches dynamisches binden, Problem: Anzahl der Methoden

18. Was ist das Visitor-Entwurfsmuster?

- Elementklasse & Besucherklasse
- Operationen werden von Elementklassen in Besucherklassen ausgelagert
- Bsp aus UE: Vivarien -> addSwarmToExpensiveAquarium
  - M Vivarien
  - N Swarms
  - → M\*N Methoden!

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

- problematisch
  - unbewusstes überladen statt überschreiben
  - wenn nicht klar welche Methode aufgerufen wird
    - m(Foo f, Bar b)
    - m(Bar b, Foo f)
    - → Foo extends Bar
    - → m(Foo(), Foo())
- unproblematisch
  - Parametertypen unterscheiden sich an mindestens einer Stelle (an einer Stelle keine Untertypbeziehungen)
  - alle Parametertypen einer Methode beinhaltet NUR Oberklassen der anderen Methode

## Kapitel 4

1. Wie werden Ausnahmebehandlungen in Java unterstützt?

- Ausnahmen sind gewöhnliche Objekte (Throwable Instanzen)
- hauptsächlich Error / Exception verwendet
- Error
  - vordefinierte, schwerwiegende Ausnahmen der Java Runtime
  - (fast) immer Programmbeendigung
  - z.B. OutOfMemoryError, StackOverflowError
- Exception
  - überprüfte: selbst definierte
  - nicht überprüfte: RuntimeException, z.B. NullPointerException, IndexOutOfBoundsException, ClassCastException

2. Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?

- Exception müssen im Methodenkopf nach throws stehen
- Untertypen dürfen keine Exceptions werfen, die im Obertyp nicht deklariert sind

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

- + unvorhergesehene Programmabbrüche (Fehlerhinweis ausgeben)
- + kontrolliertes Wiederaufsetzen (Programm woanders fortsetzen)
- Ausstieg aus Sprachkonstruktion wie Schleifen usw.
- alternative Rückgabewerte (da nur ein Rückgabewert pro Methode möglich ist)
- sparsam einsetzen

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

- Threads
- zum parallelen Abarbeiten → Speed Up

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

- um Inkonsistenzen zu vermeiden (Bsp.: Race Condition mit z.B. Bankkonto)
- möglichst feine Granularität (nur nötiges synchronisieren)

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

- Dead-Locks (keiner arbeitet → nichts passiert)
  - vermeiden durch lineare Anordnungen (i.e. keine Zyklen)
- Starvation
  - vermeiden durch faire Queues

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

- Markierungen setzen, auf die Laufzeitsystem / Entwicklungswerkzeuge reagieren können
- ohne Überprüfung einfach ignoriert
- erlauben Sprachkonstrukte hinzuzufügen ohne Sprache zu ändern = epic stuff
- übliche Annotations
  - @Override
  - @Deprecated
  - @SuppressWarnings
  - @FunctionalInterface

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

- SOURCE (vom Compiler wie Kommentare verworfen)
- CLASS (in Class File enthalten)
- RUNTIME (zur Laufzeit abfragbar)

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

```
@RetentionPolicy(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE) // not mandatory
@interface MyAnnotation {
    String value;
    MyEnum foo;
    int bar default 0;
}
```

```
@MyAnnotation(value = "asdf", foo = MyEnum.thatCase)
public class Baz {...}
```

10. Wie kann man zur Laufzeit auf Annotationen zugreifen?

- Reflexion
- Ausgangspunkt: Objekt vom Typ `Class`
  - `getAnnotation(s)`
  - `getMethod(s)`
  - `getField(s)`
  - `invoke(Method m)`

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

- kapselt Verhalten, das mehrere Klassen betrifft, in Aspekten.
- Ein Aspekt beschreibt dabei eine Funktionalität, als auch alle Stellen im Programm, an denen diese Funktionalität angewendet werden soll.

12. Was bedeutet Separation-of-Concerns?

- Modulare Aufteilung des Programms

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

- Core-Concerns lassen sich in eine Klasse packen
- Cross-Cutting-Concerns betreffen das ganze Programm (z.B. Logging, Authentifizierung)

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

- Join-Points
  - identifizierbare Stelle während der Programmausführung
  - z.B. Methodenaufruf, Zugriff auf Objekt
- Pointcuts
  - Programmkonstrukt, das einen Join-Point auswählt und kontextabhängige Informationen sammelt
  - zB Argumente des Methodenaufrufs, Zielobjekt

- Advices
  - der Code, der um den Join-Point ausgeführt wird (before, around, after)
- Aspect
  - wie eine Klasse
  - zentrales Element von AspectJ
  - enthält alle Deklarationen von Join-Points / PointCuts / Advices / Methoden

15. An welchen Programmpunkten können sich Join-Points befinden?

- Methode
  - Ausführung
  - Aufruf
- Konstruktor
  - Ausführung
  - Aufruf
- Feldzugriff
  - get
  - set
- Initialisierung einer Klasse
  - staticinitialization
- Objektinitialisierung (durch Konstruktor)
  - preinitialization
  - initialization
- Exceptions
  - handler

## Kapitel 5

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

- erzeugend:
  - Factory-Method (Virtual-Constructor)
  - Prototype
  - Singleton
- strukturell / behavior
  - Decorator (Wrapper)
  - Proxy (Surrogate)
  - Template-Method

### **Decorator (Wrapper):**

- Anwendungsgebiet
  - um dynamisch Verantwortlichkeiten zu Objekten hinzuzufügen / entfernen zu können

- Bsp: Scroll-Bar
- wenn Vererbung unpraktisch / nicht unterstützt (final) ist
- Struktur
  - Component Interface (Window)
  - ConcreteComponent Klasse (WindowImpl)
  - (abstrakte) Klasse Decorator (WinDecorator implements Window), enthält Referenz auf Component
- Eigenschaften
  - flexibler als statische Vererbung (zur Laufzeit hinzugefügt)
  - ConcreteComponent muss nicht die gewünschte Funktionalität enthalten
  - Vermeidung von Klassen die weit oben in Hierarchie bereits viele Methoden und Variablen haben
  - viele kleine Objekte
    - einfach konfigurierbar
    - schwer verständlich & wartbar
- wichtige Details

### **Factory-Method (Virtual Constructor):**

- Anwendungsgebiet
  - Klasse soll Objekte erzeugen, deren Klasse sie nicht kennt
  - Klasse möchte, dass Unterklassen bestimmen welche Art von Objekten erzeugt wird
  - Verantwortlichkeiten an Unterklasse(n) delegieren
  - wenn zentrale Verwaltung von Allocation / Release von Objekten gewünscht
- Struktur
  - eigene Creator-Klassenhierarchie parallel zur eigentlichen Klassenhierarchie
  - abstract Document / Text extends Document / Picture extends Document
  - abstract DocCreator: Document create() / TextCreator extends DocumentCreator / ...
- Eigenschaften
  - flexibel, Entwicklung von Unterklassen vereinfacht
  - Verknüpfung paralleler Klassenhierarchie
  - viele Unterklassen nötig
- wichtige Details
  - Konstruktoren leer / gleiche Parameter -> kein Problem
  - Konstruktoren unterschiedlich -> zentral Ablage / statisch festlegen

### **Iterator:**

- Anwendungsgebiet
  - sequentieller Zugriff auf eine Ansammlung von Objekten (= Aggregate) ohne innere Darstellung offenzulegen
    - List / Array / ...
- Struktur
  - Aggregate → iterator() / ConcreteAggregate
  - Iterator → next(), hasNext() / ConcreteIterator
- Eigenschaften
  - verschiedene Abarbeitungen (zB Reihenfolge bei Bäumen)

- einheitliche Schnittstelle für Aggregate
- Iterator verwaltet selbst Zustand → mehrere Iteratoren gleichzeitig möglich
- wichtige Details
  - interne vs
    - kontrollieren selbst wann nächstes Element
    - next / hasNext nicht zugänglich,
    - zB forEach auf eine Map, stream Operationen
    - einfacher zu verwenden
  - externe Iteratoren
    - in Schleife nach nächstem Element fragen
    - flexibler
    - eignen sich um 2 Aggregate miteinander zu vergleichen
    - historisch gesehen in OOP bedeutender, jedoch Lambda-Ausdrücke immer wichtiger

### Prototype:

- Anwendungsgebiet
  - neu zu erzeugende Objekte durch Kopieren eines Prototyps erzeugen
  - zB Polygon aus kopieren eines Polygons erstellen
  - Klassen, von denen Objekte erzeugt werden sollen, erst zur Laufzeit bekannt
  - Vermeidung einer Hierarchie von Creator Klassen (Factory-Method)
  - nur wenig unterschiedliche Zustände für Objekt einer Klasse
- Struktur
  - Client → Referenz auf Prototype
  - Prototype → clone
  - ConcretePrototype1 → clone, ConProto2 → clone, ...
- Eigenschaften
  - verstecken konkrete Produktklassen vor dem Client -> reduzieren der Klassen die der Client kennen muss
  - keine Änderungen wenn neue Produktklassen kommen
  - Prototypen können während der Laufzeit geändert werden im Gegensatz zur Klassenstruktur
  - vermeidet große Zahl an Unterklassen
  - erlaubt dynamische Konfiguration von Programmen
- wichtige Details
  - jede Unterklasse muss clone implementieren
    - → oft schwer, da nicht nur "eigene" Klassen (Bibliotheken)
    - → !zyklische Referenzen!
  - clone in Java vordefiniert (flach), Cloneable implementieren
  - tiefe Kopien schwer
  - Prototype-Manager zur Verwaltung von Prototypen möglich
  - clone hat (oft) keine Parameter, Initialisierungsmethoden notwendig
  - in statischen Sprachen sinnvoll
  - in dynamischen Sprachen direkt unterstützt

### Proxy (Surrogate):

- Anwendungsgebiet
  - Platzhalter für ein anderes Objekt und kontrolliert Zugriffe darauf

- zB Objekterzeugung ist aufwändig → erst erzeugen (durch Proxy), wenn wirklich benötigt
- Struktur
  - Interface / abstract class Subject als gemeinsame Schnittstelle → request(...)  
(gemeinsame Schnittstelle für Proxy und RealSubject)
  - class RealSubject implements Subject
  - class Proxy implements Subject, Referenz auf ein RealSubject
- Eigenschaften
  - Chaining möglich
  - Darstellung nicht existenter Objekte
  - Proxy muss nur Subject kennen nicht RealSubject
- wichtige Details → Arten
  - Remote Proxies (Platzhalter für Objekte in anderen Namensräumen -> Festplatte / andere Rechner, komplexe Kommunikationskanäle)
  - Virtual Proxies (Erzeugung bei Bedarf)
  - Protection Proxies (Zugriffsrechtkontrolle)
  - Smart References (einfache Zeiger ersetzen -> Reference Count, synchronisieren von Zugriffen)

### Singleton:

- Anwendungsgebiet
  - nur 1 Instanz einer Klasse + globaler Zugriff darauf
  - Klasse erweiterbar durch Vererbung, Anwender sollen die erweiterte Methode ohne Änderungen verwenden können
- Struktur
  - Klasse mit statischer Methode instance()
  - Konstruktoren nicht public (in der Regel)
- Eigenschaften
  - vermeiden Global-namespace Cluttering
  - Vererbung wird unterstützt
  - verhindert Instanzen außerhalb der Kontrolle der Klasse
  - mehrere Instanzen möglich + Objekt änderbar
  - flexibler als statische Methoden (unterstützen dynamic binding nicht)
- wichtige Details
  - normales Singleton sehr einfach
  - bei Unterklassen von Singleton -> kompliziert

### Template-Method:

- Anwendungsgebiet
  - unveränderlichen Teil eines Algorithmus nur einmal implementieren
  - Unterklassen das Verhalten des veränderbaren Teils festzulegen
  - Code-Duplication vermeiden (zusammenfassen von gemeinsamen Verhalten in Unterklassen)
  - Erweiterungen in Unterklassen kontrollieren:
    - Template-Methods rufen Hooks auf
    - nur diese Hooks können überschrieben werden
- Struktur

- AbstractClass → templateMethod() / primitiveOperation1() / primOp2() / ...
- ConcreteClass extends AbstractClass → primitiveOperation1() / primOp2() / ...
- Eigenschaften
  - fundamentale Technik zur direkten Codewiederverwendung
  - in Bibliotheken / Frameworks sinnvoll
  - umgekehrte Kontrollstruktur: "Hollywood Prinzip" -> "Don't call us, we'll call you"
    - → Oberklasse ruft Methoden der Unterklasse auf
- wichtige Details
  - Anzahl der primitiven Operationen sollte möglichst klein sein
  - genau Spezifizierung der Hooks (welche müssen / dürfen überschrieben werden)

### **Visitor (siehe Abschnitt 3.4.2):**

- Anwendungsgebiet
  - Multimethoden durch mehrfaches dynamisches Binden simulieren
- Struktur
  - Visitor → visit(Element) / visit(otherElement)
  - Element → accept(Visitor) / accept(OtherVisitor)
- Eigenschaften
  - Klassenanzahl steigt sehr stark an ( $N \cdot M$ )
- wichtige Details
  - Visitor- & Elementklasse oft gegeneinander austauschbar

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

- siehe oben

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

- siehe oben

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

- ja, um Start zu supplien usw.

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

- haben kein Problem mit Aggregatsänderungen während Durchwanderung (hinzufügen, löschen)

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?

- default Implementierungen

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?

- eher erhöht (siehe oben)

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

- Prototype > Factory wenn Objekterzeugung teuer/aufwändig
- Factory > Prototype um Initialisierungslogik vor Client zu verstecken

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

- siehe 5.1, wenn zB immer nur ein Druckertreiber existieren soll, aber verschiedene zur Auswahl stehen

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

- Decorator erzeugt das Objekt nicht, Proxy könnte dies schon
- Proxy weiß üblicherweise den spezifisch(er)en Typ, Decorator nur Base Type
- Decorator fügt Verantwortlichkeiten hinzu, Proxy kontrolliert Zugriff auf das Objekt

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

- zyklische Referenzen
- flache Kopien: nur das eine Objekt wird kopiert
- tiefe Kopien: es werden auch alle Referenzen und deren Referenzen usw. kopiert

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

- Oberfläche erweitern gut möglich
- nicht gut für inhaltliche Erweiterungen
- nicht gut für bereits umfangreiche Decorator

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

- ja ist möglich
- Security? idk

14. Was unterscheidet Hooks von abstrakten Methoden?

- Hooks können Implementierungen haben, müssen somit nicht abstrakt sein
- Hooks werden von der Oberklasse aufgerufen