Fragenkatalog Lösungen (WS 2021|2022)

3.0 VU Objektoriente Programmierung 185.A01

11. Juli 2022

Zusammenstellung aus allen Fragen des semesterbegleitenden Skripts.

Kapitel 1 (Paradigmen der Programmierung)

1. Was versteht man unter einem Programmierparadigma?

   - Programmierstil mit **Eigenschaften** wie zugrunde liegendem Berechnungsmodell, Strukturierung, Programmablauf oder Datenfluss, Aufteilung eines Programms in überschaubare Einzelteile.
   - Definiert sich durch seine Sprache und Werkzeuge.
   - **Imperative Paradigma** auf Maschinenbefehlen aufbauende (nach Programmaufteilung: prozedurale und objektorientierte P.) versus **deklarative Paradigma** nach formalen Modellen beruhende (funktionale und logikorientierte P.).
   - Paradigmen haben immer ein Berechnungsmodell, auf dem sie aufbauen.

2. Wozu dient ein Berechnungsmodell?

   - Berechnungsmodelle oder Formalismus haben formalen Hintergrund und **dienen als Grundlage für ein Paradigma**. Berechnungsmodelle beschreiben ausschließlich die Programmausführung (36).

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

   - Formalismen beschreiben Konzepte.
   - Beispiele für Konzepte/Modelle/Formalismen: Funktionen, Prädikatenlogik, ConstraintProgrammierung, Temporale Logik und Petri-Netze, freie Algebren, Prozesskalküle, Automaten, WHILE/GOTO
   - Bspw. Funktionen: Es gibt für die Beschreibung von Funktionen unterschiedliche Formalismen (z.B. primitiv-rekursive Funktionen (durch Komposition und Rekursion anderer Fkt., nicht volllständig), μ-Funktionen (turingvollständig, auf partielle Fkt. angewendet), λ-Funktionen(turingvollständig, Fkt. ohne Notwendigkeit von Rekursion))

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

- Welche Berechnungsmodelle:
  - Beispiele für Konzepte/Modelle/Formalismen: Funktionen, Prädikatenlogik, ConstraintProgrammierung, Temporale Logik und Petri-Netze, freie Algebren, Prozesskalküle, Automaten, WHILE/GOTO
- **Eigenschaften von Berechnungsmodellen** - praktische Kriterien für den Erfolg von Paradigmen:
  - **Kombinierbarkeit**: bestehende Programmteile sollen sich möglichst einfach zu größeren Einheiten kombinieren lassen, ohne dass die größere Einheiten ihre einfache Kombinierbarkeit einbüßen. Funktionen gutes Bsp.: sie setzen sich aus weiteren Aufrufen von anderen Fkt. zusammen. Skalierbarkeit wichtig, d.h. auch große Programmteile/Einheiten müssen gut kombinierbar sein.
  - **Konsistenz**: Ein Formlismus reicht nicht als Grundlage aus, daher braucht man mehrere. Diese mehrere Formalismen/Konzepte sollen in sich und miteinander konsistent sein/sollen gut zusammenpassen. Gute Sprachen kommen mit wenigen aber konsistenten Konzepten aus. Konsistenz ist in der Praxis wichtiger als Optimalität.
  - **Abstraktion**: Eines der wichtigsten Ziele höherer Programmiersprachen - unabhängig von Hardwaredetails, möglichst (System-)portabel.
  - **Systemnähe**: Programme müssen effizient auf realer Hardware ausführbar sein (das Paradigma spiegelt dabei wesentliche Teile der Hardware/Betriebssystem wieder). Konzept geht auf Kosten der Portabilität/siehe Abstraktion.
  - **Unterstützung**: Der Erfolg von Paradgimen hängt nicht nur von dem zugrundeliegenden Konzept, sondern auch von der Unterstützung (Personen, die dahinterstehen, einflussreiche Firmen (fördern JAVA oder C#))
  - **Beharrungsvermögen**: Durch Innovation in Softwareentwicklung kann es zu Paradigmenwechsel kommen, braucht aber überzeugenden Grund (Killerapplikation = erfolgreiche Software mit neuem Paradigma)

5. Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen?

   Wie äußert sich dieses Spannungsfeld?

   - Widersprüchliche Ziele:
     - **1) Flexibilität und Ausdruckskraft**: In kurzen Texten die Dstl. aller vorstellbaren Programmabläufe
     - **2) Lesbarkeit und Sicherheit**: sollen Absichten und mögliche Inkonsistenzen von Programmteilen leicht erkennen lassen
     - **3) verständliche Konzepte**: was ist einfach machbar
   - dynamische Programmierung: bevorzugt 1) ggüber 2)
   - statische Programmierung: bevorzugt 2) ggüber 1)
   - jüngere, hauptsächlich oo-Programmierspr. erzielen hohen Grad an 1) und 2) aber nicht 3)
   - Tendenz: immer weniger 3)

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

   - **Strukturierte Programmierung** (als Bsp. für Pendelbewegung bzgl. Entwicklung dynam./stat. Progr.) Mehr Struktur in prozedurale Pr. bringen. Jedes Programm bzw. jeder Rumpf einer **Prozedur** ist aus 3 einfachen Kontrollstrukturen aufgebaut: **Sequenz, Auswahl, Wiederholung**.

- Gegenbsp. zu dieser Prozedur ist **GOTO** (mehr Flexibilität und Ausdruckskraft, schlechtere Lesbarkeit), dafür **Prozedur** (bessere Lesbarkeit und damit auch bessere Kombinierbarkeit, zusätzlich noch Einstiegs- und Ausstiegspunkt wohldefiniert)
- **Wozu dient sie**: bringt mehr Struktur in Prozeduren und erhöht Lesbarkeit/Sicherheit und bringt durch Einfachheit/Modularisierung wieder ein Stück verlorener Flexibilität

7. Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?

- **Seiteneffekte (SE)** und **Querverbindungen (QV)**: Problem der imperativen Programmierung. Programmfortschritt erzielt man durch SE (durch Zuweisung neuer Werte an Variablen = offensichtliche/erwartete SE). Problematisch sind nicht erwartete, sondern versteckte SE und QV.
  - was sind SE: Wertzuweisungen an Variablen, Ein- und Ausgaben
- **Lösungsansätze** (generell durch gute Dokumentation):
  - **deklarative Paradigma**: radikaler Ansatz - durch **referentielle Transparenz** bzgl. Ausdrücke (= ref. Ausdruck kann durch seinen Wert ersetzt werden, also keine neue Wertzuweisung an Variablenwerte). Z.B. ersetzen einer Fkt. f(x) (Ausdruckvereinfachung), wenn gilt: f(x) + f(x) = 2 * f(x)
  - **oo Paradigma**: gemäßigter Ansatz - nur teilweise referentielle Transparenz, dort wo möglich (bspw. bei Operationen wie +,-,*,/), ansonsten offensiver Umgang, bspw. lokale Beschränkung auf Objekte (Objekt in diesem Zsmhang Strukturierungsmittel, O. fassen Prozeduren und Variablen zu Einheiten zusammen)

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

- Referentielle Transparenz:
  - **Was**: Ref. Ausdruck kann durch seinen Wert ersetzt werden, sodass es nicht zu SE (besonders versteckt im Hintergrund, versteckte neue Wertzuweisungen) kommen kann.
  - **Wo**: Prinzipiell dort, wo es (besonders neue, updates) (Variablen)zuweisungen geben könnte, aber auch darüber hinaus.

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

- Weil Ein- und Ausgaben SE sind und sie nicht gut sichtbar in der Aufrufhierarchie sind (bspw. versteckt in Fkt.aufrufen). Dies versucht man zu lösen, indem man sie nur gut sichtbar ganz oben in der Aufrufhierarchie erlaubt.

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

- Oo Sprachen gehen offensiv mit SE um (hier gibt es keine/kaum referentielle Tr.), hier werden diese durch Strukturierungsmittel wie Objekte "gebändigt" bzw. sind für oo. Programme überhaupt sinnstiftend (Objekte können sich mit der Zeit durch SE verändern).

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

- **Was**: Werden behandelt wie normale Daten bzw. sind Datentypen einer Sprache, die man frei Variablen zuweisen kann. Zeichnen sich durch uneingeschränkte Verwendbarkeit (in dem jeweiligen Programmierkontext) aus (und ziehen so aber auch viele Sonderfälle nach sich). Es sind Daten als Übergabeparameter oder Rückgabewerte einer Funktion oder einer Prozedur oder diese Daten können Variablen zugewiesen werden.
- **Bsp.**
  - funktionale Sprachen: Funktionen sind hier FCE. Funktionen als Parameter sind dabei als eine Art Kontrollstruktur zu sehen.
  - oo Sprache: Objekte sind FCE. Besondere Objekte wie Iteratoren bilden in OOP Kontrollstrukturen.

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

- **Applikativer Programmierstil** = (Ist ein Paradigma.) Häufiger Gebrauch von Funktionen höherer Ordnung. Hier erstellt man quasi Schablonen von Programmteilen und übergibt Funktionen als Parameter (und als kleine Kontrollstrukturen).

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

- **Programmieren im Kleinen** (Variablen, Typen, Funktionen) versus **Programmieren im Großen** (Modularisierungseinheit)
- Übersicht über **Modularisierungseinheiten** (Modul, Objekt, Klasse, Komponente, Namensraum) (bilden alle eigene Namensräume) und ihre Eigenschaften:
  - **Modul** = eine Übersetzungseinheit (Einheit, die Compiler in einem Stück bearbeitet, z.B. Interface oder Klasse). Module lassen sich unabhängig voneinander entwickeln.
    - Haben Schnittstelle (= der Ort, wo zusammengefasste Information über Inhalte des Moduls, die auch in anderen Modulen verwendet werden). Schnittstellen unterscheiden klar zwischen Moduleninhalten (äußere Inhalte (exportiert) und innere (privat)).
    - Importierte Inhalte müssen innerhalb eines Moduls eindeutig sein und bilden Namensräume (explizite Import ermöglicht getrennte Namensräume) - Namenskonflikte kann man durch Umbennenung oder durch Qualifikation des Namens auflösen.
    - Module können (in der Regel) nicht zyklisch voneinander abhängen, genauso wie Schnittstelleninformationen.
  - **Objekt** = keine Übersetzungseinheit, kapseln Variablen und Methoden zu logische Einheit (Kapselung) und schützen private Inhalte vor Zugriffen von außen (Data-Hiding) (Kapselung+Data-Hiding=Datenabstraktion).
    - Bilden wie Module eigene Namensräume, weil Änderungen privater Teile sich nicht auf andere Objekte auswirken.
    - Objekte sind im Ggsatz zu Modulen first-class-entities.
    - Wichtigste Eigenschaften sind: Identität (identisch: wenn gleiche Referenz, gleich: wenn gleiches Verhalten), Zustand, Verhalten.
  - **Klasse** = Schablone für Erzeugung neuer Objekte, spezifizieren wichtigste Eigenschaften, belegen sie aber nicht mit Werten. Zyklische Abhängigkeiten zwischen Klassen sind verboten. (Auflösung solcher Abhängigkeiten durch Ableitung von anderen Klassen oder Interfaces)

  - **Komponente** = ist ein eigenständiges Stück Software, das in Programme eingebunden und nicht alleine lauffähig ist. Wie Module sind K. Übersetzungseinheiten und Namensräume, sind aber flexibler: Komponente importieren Inhalte von zur Übersetzungszeit nicht genau bekannten anderen Komponenten (Module importieren Inhalt von ganz bestimmten, namentlich genannten anderen Modulen).
  - Namensraum (siehe unten)

14. Welche Bedeutung haben Schnittstellen für Modularisierungseinheiten?

    Warum unterscheidet man zwischen von außen zugreifbaren und privaten Inhalten?

- **Schnittstellen** sind Ort mit zusammengefasster Informationen über die Inhalte, die mit der Umwelt ausgetauscht werden sollen.
- **Zugriff von außen** (Export/Import) können andere Objekte verändern, **private Inhalte** sind im Sinne der Datenabstraktion geschützt (ermöglicht dadurch erst Modularisierung von Programmteilen)

15. Was ist und wozu dient ein Namensraum?

- Modularisierungseinheiten wie Module, Klassen, Objekte, Komponente bilden Namensräume und verhindern Namenskonflikte (gilt nicht für globale Namen, stehen außerhalb der Modularisierungseinheit).
- (eigene Definition) Ein Raum oder ein Abschnitt, indem Namen eindeutig sein müssen (bspw. sind Namensräume Module, Objekte, Komponente).
- Bsp.: a.b.C = Klasse C im Namensraum b, welcher im Namensraum a steht

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

- **Module**: Der Vorteil der getrennten und damit schnelleren Übersetzung. Hier bekannt, von welchen anderen Modulen importiert wird, unbekannt, wo Importiertes eingesetzt wird.
- **Komponente**: Bilden ebenfalls eigene Übersetzungseinheit, wie Module. Hier unbekannt, von welchen anderen Komponenten importiert wird und ebenso unbekannt, wo Importiertes eingesetzt wird. Dadurch Verringerung der Abhängigkeit der Komponenten voneinander und dadurch kein Problem mit zyklischen Abhängigkeiten.

17. Was versteht man unter Datenabstraktion, Kapselung und DataHiding?

- **Datenabstraktion** = DataHiding + Kapselung
  - **DataHiding**: private Inhalte werden vor Zugriff von außen geschützt.
  - **Kapselung**: Objekte kapseln Variablen und Methoden zusammen zu einer logischen Einheit

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

- Komponente sind für sich nicht lauffähig (im Gegensatz zu Modulen) und brauchen daher die Existenz anderer Komponente. Allerdings: Wenn Komponente Inhalte anderer Komponente importieren, ist zur Übersetzungszeit nicht genau bekannt, von welchen Komponenten importiert wird. Im Gegensatz dazu bietet die Klarheit bzgl. des Imports bei Modulen (Module importieren Inhalte von ganz bestimmten, namentlich genannter anderer Module) Erleichterung.

19. Wie kann man globale Namen verwalten?

- Man kann sie mit eindeutigen Namen verwalten, um so Modularisierungseinheiten zu adressieren (URI-Adressen).

20. Was versteht man unter Parametrisierung?

Wann kann das Befüllen von „Lücken“ durch welche Techniken erfolgen?

- **Parametrisierung** = Bewusste Lücken in Modularisierungseinheiten, Befüllen zur Laufzeit.
- Ist ein Schlüsselkonzept zur Steigerung der Flexibilität von Modularisierungseinheiten.
- Wertzuweisung an Variablen auf unterschiedl. Weise/Techniken:
  - **Konstruktor** mit formalen Parametern
  - **Initialisierungsmethode** (Erzeugung und Initialisierung in unterschiedlichen Schritten)
  - **Zentrale Ablage** = globale Variablen oder Konstante
- Wann Befüllung: Bei der Erzeugung (Konstruktor) oder auch danach (Initialisierungsmethoden).

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

- Bspw., wenn Objekte durch Kopieren erzeugt werden oder zwei zu erzeugende Objekte voneinander abhängen. Dann kann das danach entstehende Objekt ja nicht mehr auf den Konstruktor zugreifen und benötigt daher Initialisierungsmethoden, um Lücken in der Variablenbelegung zu füllen.

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

- **Dependency-Injection** = Übertragung der Verantwortung von Erzeugung und Initialisierung von Objekten an eine zentrale Stelle (z.B. eine Klasse), von da aus Überblicken und Steuerung der Abhängigkeiten zwischen Objekten.
- **Vorteil(e)** = bessere Übersicht/Steuerung, alles ist gebündelt
- **Nachteil(e)** = bei Programmänderungen müssen alle Lücken, die womöglich aufeinander aufbauen, gefunden und ebenfalls verbessernd abgearbeitet werden. Daneben werden bei nachträglicher Änderungen alte Namen aus Bequemlichkeit beibehalten, obwohl sie nicht mehr (möglicherweise) vollständig sinnvoll sind. Zentrale Stellen können diese Bequemlichkeit fördern und so Programminkonsistenzen vergrößern.

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

- **Generizität** = Form der Parametrisierung, bei der Lücken bereits zur Übersetzungszeit befüllt werden (29), anstatt zur Laufzeit mit Befüllung von Daten, welche dann immer first-class-entities (als Variablen darstellbar) sind. Kurz: Einsetzen von Typen als Parameter.
- Alle Modularisierungseinheiten außer Objekte können generisch sein.
- Befüllen mit generischem Parameter (gen. Par. stehen für Konzepte und sind keine first-class-entities (fce), manchmal sind gen. Par. auch Typen, dann nennt man sie **Typparameter**)

- Schwierigkeit: Da Typparameter und auch gen. Parameter im Allgmeinen keine fce sind, ist die Einschränkung auf generische Parameter schwer auszudrücken.

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

- **Was**: A. sind optionale Parameter, kann man zu untersch. Sprachkonstrukten hinzuzählen (bspw. „@Override“). Sie werden als Werkzeug verwendet (wenn Compiler @Override versteht, kann er bspw. Warnung ausgeben) oder werden ignoriert. Man kann über spezielle Funktionen abfragen, mit welchen A. ein Sprachkonstrukt versehen ist. Insgesamt funktionert aber alles, als ob es keine A. gibt (solange A. nicht explizit abgefragt werden).
- Gemeinsamkeit z. Generizität: Annotation wirkt sich ebenfalls statisch zur Übersetzungszeit aus, (Eignen sich wie G. für statische Informationen.)
- **Unterschied z. Generizität**: Die Lücken, die durch Annotation befüllt werden, sind im Programm nirgends festgelegt.

25. Was versteht man unter aspektorientierter Programmierung (AOP)?

- Aspektor. Pr. = Hinzufügen von Aspekten zu einem Programm von außen. Man kommt in der Regel ohne Spezifikation von Lücken aus.
- Aspekt = Ein A. spezifiziert eine Menge an Punkten im Programm (z.B. alle Stellen, an denen eine Methode aufgerufen wird) und was dort passieren soll (z.B. welcher Code vor und nach dem Aufruf passieren soll). Oder man kann bspw. die Generierung bestimmter Debug-Informationen veranlassen.
- Aspect-Weaver = Ein Werkzeug, das angewendet auf das Programm und die Aspekte das Programm entsprechend modifiziert (manche A.-W. erledigen Aufgabe erst zur Laufzeit).
- Problem: Bestimmung der betroffenen Punkte im Programm (für die Aspekt benötigt wird) erfordert Wissen über Implementierungsdetail.
- Zusatz: Z.B. „AspectJ“ als AOP-Extension für Java. Hier kann man Aspekte ähnlich wie Klassen (public class Foo{...}) so definieren:
  aspect Name {
  pointcut method() : execution(set*(...));*
  before() : method() {...};
  after() : method() {...}
  }

26. Wodurch unterscheidet sich Parametrisierung von der Ersetzbarkeit,

und warum ist die Ersetzbarkeit von so zentraler Bedeutung?

- **Unterschied**: Während Modularisierungseinheit (ME) A und ME B bei der Parametrisierung voneinander abhängen (können), ist es bei der Ersetzbarkeit genau andersherum: ME A und B dürfen nicht voneinander abhängen, ansonsten wären sie nicht gegeneinander austauschbar.
- **Zentrale Bedeutung** (in der OOP): Ermöglicht viele Vorteile, so kann man Programme beliebig erweitern, ohne bestehenden Code anpassen zu müssen (weniger Wartungsaufwand bei Erweiterung) oder durch Ersetzbarkeit ist die Grundlage für Erzeugung neuer Programmversionen gegeben.
- **Bedingung für Ersetzbarkeit:** E. ist nur dann als gegeben anzusehen, wenn es eine Klassenableitung (also eine Hierarchie für Vererbung und für das Ersetzbarkeitsprinzip) gibt.

27. Wann ist A durch B ersetzbar?

- Eine ME A ist durch eine ME B genau dann ersetzbar, wenn ein Austausch von A durch B keinerlei Änderungen an Stellen nach sich zieht, an denen A mit B ersetzt wird.

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

- Schwierig festzulegen, wann etwas ersetzbar ist. Hängt von der Erwartung von außen ab. Daher nur ME (die alle erlaubten Betrachtungsweisen von außen klar festlegen) ersetzbar und es sind Schnittstellen (um Erwartbarkeit zu sehen und zu definieren) erforderlich.
- Schnittstellen kann man verschiedenartig spezifizieren: Signatur, Abstraktion realer Welt, Zusicherungen, überprüfbare Protokolle

29. Was ist die Signatur einer Modularisierungseinheit?

- **Signatur** = Ist eine Schnittstelle, die es ermöglicht, von außen auf Inhalte einer ME zuzugreifen. (Schnittstelle ist Signatur der ME.)
- **Problem:** Man kann sich nicht nur auf Signaturen verlassen, da diese gelegentlich auch gleich sein können.

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

- Schnittstellen kann man statt einer Signatur auch durch Namen und informelle Texte beschreiben. Diese Schnittstellen entsprechen abstrakten Sichtweisen von Objekten aus der realen Welt. (Bsp. Abstraktion Fahrzeug, darin kann man ein Fahrrad oder auch ein Auto "einsetzen").
- **Relation Signatur-Abstraktion**: Signatur und Abstraktion müssen aufeinander abgestimmt sein und müssen daher passen (Konzept beruht allerdings auf Intuition).

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

- Zusicherung ist eine **genaue Beschreibung** der **erlaubten Erwartung an eine ME**, um Fehler auszuschließen. Beschreibung bezieht sich auf **Verwendungsmöglichkeiten** aller nach **außen sichtbaren Inhalte der ME**. (Design-By-Contract)
- **Design-By-Contract:** Eine Schnittstelle entspricht einem Vertrag zwischen Client und Server und wird durch Zusicherungen festgelegt. (Vorbedingung, Nachbedingung, Invariante, History-Constraints)

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

- Typen sind konsistent, wenn die Typen der Operanden mit der Operation zusammenpassen, andernfalls tritt ein Typfehler auf (also wenn Typen der Operanden nicht mit der Operation zusammenpassen).

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

- **Wie:** Typen schränken das Verwenden von beliebigen Werten oder Ausdrücken ein, dadurch verliert man Freiheit, aber man gewinnt bessere Planbarkeit (durch verbesserte Erwartbarkeit). Sie stellen eine bestimmte Form der Abstraktion dar, das hat positiven Einfluss auf Zuverlässigkeit und Lesbarkeit.

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

- **Statische Typprüfung** sichern zu, dass es zur Laufzeit keine Typfehler geben wird. (Z.B. verwendet Haskell nur statische Typprüfung.) Statische Prüfung ist allerdings begrenzt, so muss man Arraygrößen dynamisch prüfen. Ein weiterer wichtiger Vorteil ist die Lesbarkeit und das Verstehen des Codes durch explizit hingeschriebene Typen.
- **Dynamische Typprüfung** zur Laufzeit ermöglicht zwar eine Offenheit, stellt aber die Gefahr dar, Typfehler zur Laufzeit zu erzeugen (und verringert auch die Lesbarkeit des Codes - was kann ich an Stelle X erwarten).

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

- Viele Typen kann ein Compiler aus der Programmstruktur **herleiten**, das nennt man Typinferenz. So muss man bei statischter Typprüfung nicht alle Typen hinschreiben.
- **Vorteil(e)**: Bis zu einem gewissen Grad erhöht die statische Typprüfung das Verständnis von Programmen, da Programmstellen nur gewissen Typen zulassen, die Flexibilität wird eingeschränkt und verhindert somit allzu komplexe Programmstrukturen.
  - Vorteile kurz: bessere Lesbarkeit, bessere Nachvollziehbarkeit (auch bei Veränderungen, Compiler zeigt zur Übersetzungszeit inkonsistente Stellen auf), bessere Planbarkeit (da es dazu zwingt, frühe, stabile Entscheidungen zu treffen und implizit den Entscheidungsspielraum für unsichere Entscheidungen einengt)
- **Nachteil(e)**: Lässt man das explizite Hinschreiben weg und verlässt sich zu sehr auf die Typinferernz, wird wieder die Lesbarkeit und das Nachvollziehen erschwert.

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

- Alle bedeutenden Entscheidungen hstl. Programmstruktur - und ablauf werden schon bei der Programmerstellung getroffen - Entscheidungspunkte:
  - In der **Sprachdefinition/Sprachimplementierung**: z.B. int als 32-Bit-Zweierkompl.dstl.
  - Zeitpunkt der Erstellung von **Übersetzungseinheiten**: Welche Klassen, welche Interfaces
  - durch **Parametrisierung** bei der **Einbindung von Modulen/Klassen/Komponente**
  - vom Compiler getroffene Entscheidungen (nicht so wichtig, nur Optimierung)
  - Unterscheidung der Initialisierungsphase (Einbindung von ME oder Parametrisierung) von der Programmausführung
- Prinzip: Je früher Entscheidungen getroffen werden, desto weniger ist während der Laufzeit zu tun und desto weniger Programmcode entsteht.

37. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?

- Typen verknüpfen die zu unterschiedlichen Zeitpunkte vorliegenden Informationen miteinander. Statisch geprüfte Typen helfen dabei, einmal getroffene Entscheidungen über den folgenden Zeitraum konsistent zu halten. Bspw. eine Variable bekommt Typ int, diese Variable wird in der Übersetzungseinheit Klasse verwendet und man kann über die gesamte Laufzeit von diesem Typ ausgehen. Anders wäre es bei einem Typparameter statt einem int (hier muss man in der Klasse von einer Referenz unbekannten Typs ausgehen).

38. Wie beeinflussen Typen die Planbarkeit weiterer Schritte?

- Typen dokumentieren anfänglich getroffene Entscheidungen (frühe Entscheidung meistens sehr stabil). Frühe Entscheidungen erleichtern die Planbarkeit weiterer Schritte, da unsichere Entscheidungen sich herauskristalisieren und ans Ende der Entscheidungsphase geschoben werden, wenn der Entscheidungsspielraum bereits eingeengt ist (Einengung gibt Entscheidungssicherheit).
- **Probleme:** Wenn in Typen dokumentierte Entscheidungen revidiert werden müssen, sind alle davon abhängigen Programmteile entsprechend anzupassen. (Bei statischer Typprüfung hilft der Compiler diese Stellen zu finden, denn dort sind dann die Typen nicht mehr konsistent.)

39. Was ist ein abstrakter Datentyp?

- Ist das Konzept von der Trennung der Innensicht von der Außensicht einer ME (Data-Hiding und Kapselung). Die nach außen sichtbaren Inhalte einer ME bestimmen dabei die Verwendbarkeit der ME. Die Innenansicht bleibt unberücksichtig - abstrakt.
- Ein A.D. ist im Wesentlichen eine Schnittstelle einer ME (betrachte Signatur der ME als Schnittstelle/als Typ), denn das ist ja von außen alleine sichtbar und verwendbar.

40. Was unterscheidet strukturelle von nominalen Typen?

- **Struktureller Typ**, wenn die die Signatur der ME als Schnittstelle bzw. Typ betrachtet wird (wie bei a.D., denn mehr ist nach außen hin nicht sichtbar). (Der Typ ist nur von dem **Namen, Parametertypen und Ergebnistypen** der nach außen sichtbaren Inhalte abhängig.) - von der nach außen sichtbaren Struktur/sie abstrahieren über Implementierungsdetails.
  - Wenn zwei ME die gleiche Signatur haben, dann haben sie den gleichen strukturellen Typen. (Achtung: hier wieder das Problem der zufällig gleichen Signatur. Daher eindeutiger Name, der das Konzept dahinter beschreibt wichtig.)
- **Nominaler Typ**, wenn neben der Signatur (=alles was nach außen sichtbar ist, alles was im Objekt/Klasse public ist) auch einen eindeutigen Namen. (Bspw. entspricht der Typ eines Objekts dem Namen der Klasse, von der das Objekt erzeugt wurde.) Objekte sind gleich, weil sie den gleichen nominalen Typ haben, also wenn ihr Typ vom gleichen Klassennamen (entscheidend) (und auch von der gleichen Signatur) ist. **Zusätzlich zum Namen (= auch schon eine Zusicherung) und Signatur sind die Zusicherungen das dritte Merkmal von nominalen Typen** (83).
- **Zusammenfassung**: Zwei nominale Typen sind äquivalent, wenn sie den gleichen Namen haben, zwei strukturelle Typen sind äquivalent, wenn sie die gleiche Struktur (hier Signatur, Außensicht) haben.

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

- **Theorie/strukureller Typ:** S.T. eignen sich beschränkt zur Abstraktion, da ja die Signatur nur zufällig gleich sein kann, daher kann der gleiche Typ auch für untersch. Konzepte stehen und alles hängt von der Programmierdisziplin ab (geeignete Namen für dahinterliegende Konzepte, aber nicht obligatorisch).

- **Praxis/nominaler Typ:** Durch den n.T. lässt sich durch eindeutige Namen zusammen mit Signaturen Abstraktionen zu Konzepten eindeutiger zuweisen. Man denkt beim Programmieren hauptsächlich abstrakt in Konzepten - sprich: die dahinterliegende Abstraktion, die Implementierungsdetails, wird nun wichtig, um einen geeigneten Namen zu finden, das hat viel mehr mit der Praxis zu tun.

42. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?

- Untertypbeziehungen werden durch das Ersetzbarkeitsprinzip definiert: *„Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.“* Es braucht also das Hierarchiedenken von der Ersetzbarkeit.

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

- Weil der Compiler die hinterliegende Abstraktion und Konzepte eines nominalen Typs nicht mit Regeln entscheiden kann, daher muss der Programmierer diese explizit hinschreiben. (Bei strukturellen Typen kann man verregelt anhand der gegebene Signatur prüfen, wer Untertyp von wem ist bzw. was der allgemeinere und was der spezifischere Typ ist.)

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

- Typen von Funktions- bzw. Methodenparameter dürfen im Untertyp nicht stärker werden.
- z.B.: Obertyp hat Methode *boolean compare (T x)*, VERBOT eines spezielleren Parametertyps der Methode/Funktion im Untertyp, also nicht: *boolean compare(U x)* ⇒ siehe **Kovariantenproblem**

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

- **Generizität:** Konzept, Typen als Parameter einsetzen. Schwierigkeit, wenn es Einschränkungen auf Typparameter gibt, dafür **2 gleichwertige formale Ansätze**:
- **F-gebundene Generizität**: nutzt Untertypbeziehungen zur Beschreibung von Einschränkungen (in Java)
- **Higher-Order-Subtyping/Matching**: direkterer Weg, beschreibt Einschränkungen durch Untertyp-ähnliche Beziehungen, indem Unterschiede in den Details berücksichtigt werden (C++, Haskell)
- In welchem Zusammenhang: um Einschränkungen auf Typparameter bei Generizität einschränken zu können (wie kann man Einschränkung beschreiben → siehe „extends“ bei Generizität (Schranken)).

46. Wie konstruiert man rekursive Datenstrukturen?

- **Induktive Konstruktion:** Man beginnt mit einer endlichem Menge M_0, definiert das Ende (Fundierung) und beschreibt, wie man mit endlich vielen Möglichkeiten aus einer Menge M_i die Menge (durch Mengenvereinigung) M_{i+1} erhalten kann (Fortschritt).

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

- Es gibt einen Basisfall M_0, wobei M_0 nicht leer sein darf (=Fundiertheit)
- Allgemein: M_0 ist nicht-rekursiv, M_i mit $i > 0$ ist rekursiv
- Da ein Wert in Java immer zumindest **null** ist, ist Fundiertheit hier immer schon gegeben (anders bspw. in Haskell)

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

- Typinferenz funktioniert nicht, wenn gleichzeitig (an derselben Stelle im Programm) Ersetzbarkeit durch Untertypen verwendet wird. Untertyp verträgt sich nicht gut mit Typinferenz (siehe Frage 43: allg. Nutzung von nominalen Typen in der Programmierpraxis, jedoch gibt es kein automatisches Verfahren (wichtig für Typinferenz), um die vorliegende Typhierarchie zu entscheiden, daher vertragen sich Untertypbeziehungen und Typinferenz nicht so gut).

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

- Typen stellen Kompatibilität zwischen Operator und Operanden sicher.
- **Informationen** werden im Allgemeinen durch **Typen** (bspw. Typen von formalen Parameter in Methoden - Information: "Lasse hier nur int als Argument zu"), im Speziellen durch **statische Eigenschaften** (bspw. Parameter braucht einen gewissen Typ, Wert darf nicht null sein, Wert darf nur größer als 2 sein etc.) in Form von *Übergabeparameter*, aber auch durch *Ergebnistypen* (bspw. bei Methoden) propagiert.

50. Erklären Sie folgende Begriffe:

- (1) Objekt, Klasse, Vererbung
- (2) Identität, Zustand, Verhalten, Schnittstelle
- (3) deklarierter, statischer und dynamischer Typ
- (4) Faktorisierung, Refaktorisierung
- (5) Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung

- (1) Objekt, Klasse, Vererbung:
  - **Objekt**: grundlegende Modularisierungseinheit, die zustandsbehaftet ist und mittels Klassen als Schablone Methoden und Variablen zu einer logischen Einheit vereint (Kapselung). Objekte können sich untereinander Nachrichten schicken (in Form von Methodenaufrufen). Objekte werden entweder durch Konstruktoren oder durch Kopieren erzeugt.
  - **Klasse**: Jedes Objekt gehört zu der Klasse, in der das Objekt implementiert ist/Klassen sind Schablonen für Objekte. Objekte sind Instanzen der (spezifischsten) Klasse (und eigentlich auch deren Oberklassen).
  - **Vererbung**: Neue Klassen aus bestehenden Klassen ableiten und es werden nur die Unterschiede der Klassen angegeben. Zwei Änderungsmöglichkeiten:
    - **Erweiterung**: Die Unterklasse erweitert die Oberklasse um neue Variablen/Methoden/Konstruktoren.

    - **Überschreiben**: Methoden der Oberklasse werden überschrieben.
    - *Eine Unterklasse kann überall dort verwendet werden, wo die Oberklasse erwartet wird.*
    - Nur **Einfachvererbung** in Java erlaubt, d.h., dass jede Unterklasse immer nur genau von einer anderen Klasse abgeleitet werden kann. (Bei Mehrfachvererbung kann eine Klasse mehrere Oberklassen haben.)

- (2) Identität, Zustand, Verhalten, Schnittstelle:
  - **Identität**: Objekte sind eindeutig durch eine unveränderliche Identität (vereinfacht Vorstellung der Identität durch eine Speicheradresse).
  - **Zustand**: Der Zustand setzt sich aus den Werten der Variablen im Objekt zusammen und ist meist änderlich.
  - **Verhalten**: Das Verhalten des Objekts beschreibt das Verhalten des Objekts beim Empfang einer Nachricht.
  - **Schnittstelle**: Manchmal als Signatur aufgefasst (Objekt als Struktur), öfters als Abstraktion über Implementierungsdetails (struktureller Typ/Abstrakter Datentyp) aufgefasst zeigt die Sicht auf ein Objekt. Wie kann man mit einem Objekt von außen interagieren (welche Methoden kann ich aufrufen/sind public), wobei ein "Objekt seine Schnittstellen implementiert", d.h. die Implementierungsdetails festlegt. Schnittstellen trennen also die Außen- von der Innensicht.
- (3) deklarierter, statischer und dynamischer Typ :
  - Bsp.: ***Vehicle fiat = new Auto();***
  - **deklarierter Typ**: Typ, mit dem die Variable deklariert wurde, hier: "Vehicle"
  - **dynamischer Typ**: Der spezifischste Typ, wobei der Compiler im Allgmeinen diese nicht kennt.
  - **statischer Typ**: Zwischen deklariertem und dynamischen Typ und wird vom Compiler ermittelt und kann daher an unterschiedlichen Stellen auch unterschiedlich sein. (Es hängt vom Compiler ab, wie spezifisch der statische Typ ist.)
- (4) Faktorisierung, Refaktorisierung:
  - **Faktorisierung**: Zerlegung eines Programms in Einheiten mit zusammengehörenden Eigenschaften. Zur guten Faktorisierung gehören gut beschreibende Funktionsnamen, aber auch das Objekt als Konzept ist im Sinne guter Faktorisierung, die die Wartbarkeit und auch die Lesbarkeit eines Programms verbessert.
    - **Faustregel guter Faktorisierung:**
      - Verantwortlichkeiten bekannt
      - Klassenzusammenhalt hoch
      - Objektkopplung niedrig (gute Kapselung)
  - **Refaktorisierung**: Wenn die Faktorisierung (mehrmals) geändert werden muss im Zuge von Verbesserungen oder anderen Gründen oder kurz: ***Änderung der Zerlegung in Klassen und Objekte bei Programmänderung.***
    (Programme werden (üblicherweise) nach dem **Wasserfallmodell** (Entwicklungsschritte: Analyse, Design, Implementierung, Verifikation und Validierung) entwickelt und in zyklischen Prozessen verbessert (hier kann man auch auf neue Anforderungen reagieren).)
- (5) Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung:
  - Im Sinne guter Faktorisierung gilt:

  - **Verantwortlichkeiten**: Verantwortlichkeit einer Klasse durch 3 w-Fragen:
    * (1) was **weiß** ich (Zustand Objekt)
    * (2) was **mache** ich (Verhalten Objekt)
    * (3) wen **kenne** ich (sichtbare Objekte/Klassen)
  - **Klassenzusammenhalt**: *Grad der Beziehung zwischen Verantwortlichkeiten der Klassen.*
    * hoch: wenn alle Variablen und Methoden arbeiten eng zusammen und sind durch den Namen der Klasse gut beschrieben
  - **Objektkopplung**: *Abhängigkeit der Objekte untereinander.*
    * stark: wenn (1) viele Methoden/Variablen nach außen sichtbar, (2) im laufenden System häufig Nachrichten und Variablenzugriffe zwischen Objekten auftreten, (3) Anzahl der Parameter in Methoden groß

51. Welche Arten von Polymorphismus unterscheidet man?

    Welche davon sind in der objektorientierten Programmierung wichtig? Warum?

- Wenn eine Variable oder eine Methode gleichzeitig mehrere Typen haben kann. Z.B. können Formale Parameter einer polymorphen Funktion an verschiedene Typen gebunden sein.
- (1) **universeller Polymorphismus**: (Generizität, Untertypen) Typen, die zueinander stehen, haben eine gleichförmige Struktur - **Generizität** durch gemeinsamen Code (kann über Typparameter mehrere Typen haben), **Untertypen** durch gemeinsame Schnittstelle für untersch. Objekte.
- (2) **Ad-hoc-Polymorphismus**: (Überladen, Typumwandlung) Hier gibt es keine gemeinsame Struktur
- Zusammengefasst: Jeder Polymorphismus, der mit Untertypen zu tun hat, nennt man auch „**objekt-orientier Polymorphismus**“.

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

- **gleich**: Zwei Objekte sind gleich, wenn sie den **gleichen Zustand** und das **gleiche Verhalten** haben. Zwei Objekte können auch gleich sein, auch wenn sie nicht identisch sind (dann sind sie Kopien voneinander), wobei sich Zustände zweier gleichen Objekten unabhängig voneinander ändern können.
- **identisch**: Zwei Objekte sind identisch, wenn sie zwar unterschiedliche Namen haben, aber einen **gemeinsamen Speicherplatz** haben (Vorsicht, dieser kann sich auch durch bspw. Speicherbereinigung ändern, ohne dass die Identität verloren geht). Es handelt sich also um ein Objekt mit zwei unterschiedlichen Namen.

Allgemeine Wiederholungsfragen

53. Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe?

Wenn Nein, worin unterscheiden sie sich?

- (siehe Frage 17) Datenabstraktion = Datenkapselung + Data-Hiding

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

- **Definition** (65): *„Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.“*
- **Definition**: *EP besagt, dass Programme so gestaltet werden sollten, dass man problemlos jeden Programmteil durch einen anderen ersetzen kann, wobei nur Modularisierungseinheiten gegeneinander ausgetauscht werden können* (bspw. bei nachträglichen Änderungen). Abstraktionen der realen Welt oder Klassenableitung stellen dabei wichtige Grundlagen dar. Umgekehrt ist das EP die Grundlage für Untertypbeziehungen. Das EP ist für strukturelle Typen eindeutig defniert, für nominale Typen aufgrund der „uneindeutigen“/formal schwer nachvollziehbaren, auf Intuition beruhende Abstraktion nicht.
- Diskussion bzw. weitere/noch einmal aufgesplittete Aspekte:
  - Eines der wichtigsten Konzepte, die das EP ermöglicht, sind **Untertypbeziehung**.
  - Ersetzbarkeit zwischen ME A und B ist nur dann gegeben, wenn die **Schnittstelle** von B das Gleiche beschreibt, wie von A (31), jedoch sich bei Ersetzbarkeit nicht nur auf Signaturen verlässt (siehe struktureller und nominaler Typ), sondern auch die **Abstraktion** (dahinter) passen sollten.
  - Ersetzbarkeit wird nur dann als gegeben erachtet, wenn es auch **Klassenableitungen** gibt. (34)
  - **Parametrisierung** kann helfen, kann aber auch einen gegenteiligen Effekt haben. (31) (Helfen kann es, wenn es sich um **universellen Polymorphismus** (Generizität, Untertypen) handelt und ME demnach flexibler aufeinander zugreifen können (**objektorientierte Polymorphismus**, z.B. durch dynamisches Binden). Nicht hilfreich kann es sein, wenn Signaturen und Abstraktionen nicht zusammenpassen, wie es beim Überladen (**Ad-hoc-Polymorphismus**) sein kann, aber nicht sein muss. Besonders wenn Namen von MEs nicht (mehr) auch die dahinterliegende Abstraktion passen.)
  - **Vererbung** hat aufgrund seiner Definition nichts mit Ersetzbarkeit zu tun, auch wenn beide als zusammengehörig betrachtet werden, (man sollte sie besser voneinander trennen). (34)
  - **Abstraktionen der realen Welt** sind eine in der Praxis wichtige Grundlage für die Ersetzbarkeit. (35)
  - Das EP ist für **strukturelle Typen eindeutig defniert**, für nominalen aufgrund der zusätzlichen Abstraktion nicht. (40)
  - **Generizität** ist kein vollwertiger Ersatz für Ersetzbarkeit. (42)
  - **Wiederverwendung von Code** (und in diesem Zusammenhang Ersetzbarkeit und Untertypen) ist wahrscheinlich der wichtigste Grund für den Erfolg von OOP. (59)

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

- (siehe Frage 54)
- Stichwortartig:
  - ermöglicht die **Wiederverwendung von Code**

  - ermöglicht die **Erzeugung neueren Programmversionen**
  - ermöglicht zusätzlich neben der Abstraktion die **Wartbarkeit von Software**
  - HINWEIS: Wenn die Wiederverwendung und Wartbarkeit nur eine untergeordnete Bedeutung hat (bspw. weil Software nur einmaligeingesetzt wird), kann OOP nicht ihre Vorteile ausspielen.

56. Wann und warum ist gute Wartbarkeit wichtig?

- (siehe Frage 54,55)
- **warum:** Wartbarkeit ermöglicht...
  - ...Software lange zu gebrauchen und daher ist es erforderlich, Programme mittels Programmversionen weiterzuentwickeln.
- **wann...**
  - ...wenn es darum geht, Software lange zu gebrauchen und es daher erforderlich ist, Programme mittels Programmversionen weiterzuentwickeln. (allgemein betrachtet)
  - ...in der Planungsphase, wenn es darum geht, gute Modularisierung (Modularisierungseinheiten) zu bilden. (entwicklungstechnisch betrachtet)
  - ...in der (meist zyklischen) Entwicklungsphase, wenn es darum geht, (durch Programmdisziplin) stets (!) Signaturen/Schnittstellen mit dahinterliegenden Abstraktionen im Einklang zu halten (geeignete Namen bspw.). (entwicklungstechnisch betrachtet)
- **wie...**
  - ...durch Modularisierung von Programmen. (22)
  - ...durch inkrementelle Verfeinerung bei der Softwareentwicklung (z.B. bei Untertypen: von Interface über Klassen bis hin zu Komponenten) (45)
  - ...durch dynamisches Binden und Untertypbeziehungen. (52)
  - ...durch gute Faktorisierung. (53)

57. Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung?

Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?

- **Vorteile:** Verbesserung von: Wartbarkeit, Lesbarkeit/Verständlichkeit von Programmcode, ermöglicht Ersetzbarkeit, ermöglicht Wiederverwendung, ermöglicht letztlich dadurch erst Langlebigkeit von Software
- Ein hoher Klassenzusammenhalt und eine niedrige Objektkopplung ermöglichen eine gute Faktorisierung. (55)
- **Faustregeln:**
  - *Der Klassen-Zusammenhalt soll hoch sein.* (56) (direkt)
  - *Die Objekt-Kopplung soll schwach sein.* (56) (direkt)
  - *Gute Faktorisierung kann die Wartbarkeit eines Programms wesentlich verbessern.* (53)
    (Bezug: Durch (Re-)Faktorisierung werden Einheiten in einem Programm erstellt/verbessert/verstärkt. Geeignete Klassen- und Objektbildung sind Teil der (Re-)Faktorisierung.)(indirekt, bezogen zuerst auf Fakorisierung)

- *Man soll die reale Welt simulieren, aber nur so weit, dass die Komplexität dadurch nicht erhöht wird.* (54)
  (Bezug: Hier besonders auf die Objektbildung bezogen bzw. das Zusammenfassen bzw. das Aufteilen und Aufgreifen von Merkmalen der realen Welt. Nicht jede Eigenschaft ist bspw. für ein Programm relevant und nicht alles muss ins Kleinste auf bspw. Klassen aufgeteilt/ausgelagert werden.)(indirekt, bezogen zuerst auf Fakorisierung)

58. Von welchen Kriterien hängen Klassenzusammenhalt und Objektkopplung genau ab?

- (Klassenzusammenhalt gegeben, wenn: Methoden und Variablen gut zusammenpassen.)
- (Objektkopplung: Sichtbarkeit nach außen, Kommunikation von Objekten untereinander, Anzahl der Parameter in Methoden.)
- Kriterien:
  - **Gute Modularisierung**, denn so besteht die Chance, dass innerhalb einer Klasse bereits Zusammengehöriges versammelt ist (aus der Planungsphase).
  - **Gute Untertypbeziehungen**, die sich aus der Planungsphase ergeben und die auch im weiteren Entwicklungsprozess eingehalten werden sollen. Ist gute Untertypenbeziehung vorhanden, ist damit eine konzeptionelle Voraussetzung ausgedrückt, die logische Sinneinheiten in Form von Modularisierungseinheiten dokumentiert. (einfach gesagt: Alles ist so geplant, dass das, was logisch/sinnhaft zusammengehört, bereits in Einheiten (z.B. in Klassen) zusammengetragen worde ist. Dadurch ist es dann leicht, Klassenzusammenhalt herzustellen, da bereits Zusammengehöriges zusammen steht. Leitfaden: *„Wenn der Klassenzusammenhalt (bereits) hoch ist, dann ist oft die Objekt-Kopplung schwach und umgekehrt.“* (56))
  - Wenn insgesamt eine **gute Abstraktion mit einer geeigneten Signatur** vorhanden ist. Dadurch muss nicht durch viel Kommunikation zwischen Objekten Informationen an die Stellen erst gebracht werden, wo sie dann wirklich gebraucht werden.
  - Gute Trennung von Außen- und Innensicht. **Was soll nach außen sichtbar sein** (nur das Nötigste) bzw. **gute Kapselung**. Das richtige Planen und Schaffung von solchen Voraussetzungen (bzgl. der (strukturellen) Signatur) verhindert im Nachhinein unnötige Kommunikation zwischen Objekten.

59. Wie kann man überprüfen, wie hoch der Klassenzusammenhalt und wie stark die Objektkopplung ist?

- Klassenzusammenhalt messen (obwohl das nur intuitiv machbar ist):
  - Wenn man Variablen oder Methoden aus einer Klasse entfernt und dann dem Programm etwas Wesentliches fehlt, so ist der KZH hoch.
- Objektkopplung:
  - Wenn nur "das Nötigste"nach außen sichtbar ist.
  - Wenn die Kommunikation zwischen Objekten (Nachrichten oder Variablenzugriffe) niedrig ist, (ansonsten stehen womöglich die falschen Berechnungen und die falschen Werte an den falschen Stellen oder Stellen, die zusammen gehören, stehen noch nicht zusammen).
  - Wenn die Anzahl an Parametern von nach außen sichtbaren Methoden groß ist, so kann dies ein Indikator dafür sein, dass eine ME/Objekt (noch) nicht alles an Variablen/Berechnungen durch Methoden zur Verfügung steht, was es eigentlich bräuchte.

60. Wie wirkt sich ein hoher Klassenzusammenhalt auf die Faktorisierung und Refaktorisierbarkeit aus?

- Ein hoher Klassenzusammenhalt ermöglicht eine gute Faktorisierung, weil es logische Einheiten zusammenfasst. Ein hoher Klassenzusammenhalt deutet also auf eine gute Faktorisierung hin. Damit ist aber dann auch gesagt, dass die Wahrscheinlichkeit bei Programmänderung klein ist, zu Refaktorisieren (Änderung der Zerlegung in Klassen und Objekte). (Ein Programm mit hohem Klassenzusammenhalt besitzt also eine "niedrige Refaktorisierbarkeit".)

61. Wie wirkt sich eine schwache Objektkopplung auf die Faktorisierung und Refaktorisierbarkeit aus?

- (siehe Frage 60) Da ein hoher KZH und eine schwache OK zusammenhängen, gilt das Gleiche wie Frage 60: Ein Programm mit schwacher OK besitzt eine "niedrige Refaktorisierbarkeit", weil es bereits eine gute Faktorisierung erreicht hat.

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

- Arten:
  - **bewährte Software** → diese spart Entwicklungsaufwand
  - **(Klassen-)Bibliotheken**: dort sind bereits (bewährte) Software/Konzepte/einfache Klassen für die Wiederverwendung gesammelt (komplexere Klassen meistens bereits zu projektbezogen) → erspart Entwicklungsaufwand und schützt (eher) vor Fehlentscheidungen (gemeint: Neues entwickeln, viel in Wiederverwendbarkeit stecken und letztlich Software nur einmalig/geringfügig einzusetzen - daher erst mit Vorhandenem arbeiten und wenn es sich "lohnt", dieses weiterzuentwickeln/neu entwickeln/anzupassen)
- **Projektinterne Wiederverwendung** → Programmteile sind in unterschiedlichen Projektversionen wiederverwendbar ⇒ erspart Entwicklungsaufwand
- **Programminterne Wiederverwendung** → bessere Lesbarkeit durch Wiederverwendung von Code

63. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?

- Refaktorisierung ist die Änderung der Zerlegung in Klassen und Objekten nach Programmänderung und erhält somit eine gute Faktorisierung des Programms. Durch eine (normalerweise) zyklische Entwicklungsphase kommt es (meistens) zu Refaktorisierungen. Refaktorisierung und (damit wiedererlangte gute) Faktorisierung steigern die Wiederverwendung.
- Faustregel: *Ein vernünftiges Maß rechtzeitiger (früh genug) Refaktorisierung führt häufig zu gut faktorisierten Programmen.* (57)

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

- Wenn die Wiederverwendung und Wartbarkeit von untergeordneter Bedeutung ist (bspw. wenn Software nur einmalig eingesetzt wird), so kann OOP ihre Vorteile nicht ausspielen. Erst durch langen Einsatz und langen Wartungszeitraum machen sich die Konzepte der OOP erst vorteilhaft bemerkbar. (59)

Kapitel 1 | Themenübersicht

Kapitel 2 (Untertypen und Vererbung)

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

   - Man kann Methoden aufrufen, die Argumente haben, wobei beim Aufruf Argumente verwendet werden können, die Untertypen der formalen Parameter (Typen der Argumente) sind. So kann durch eine (im Hintergrund) eingerichtete Hierarchie (z.B. durch eine Schnittstelle wie einem Interface) schon definierte Typen verwendet werden, indem man sie in bspw. neue Methoden als Argument wiederverwenden kann.
   - Man hat bereits Variablen f mit einem Typ Fahrzeug erstellt, so kann man in ähnlicher Weise dieser Variablen f ein Objekt x zuweisen, wobei der Typ Auto des Objekts x Untertyp des Typs Fahrzeug ist ($Fahrzeug\ f = Auto\ x \mid Auto < Fahrzeug$).
     So kann man „alte“ Objekte durch geeignete Hierarchie im Hintergrund (Interfaces als Schnittstelle definieren Hierarchie der nominalen Typen) wiederverwenden.

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

   - (Typen = Schnittstellen von Objekten, die in Klassen bzw. Interfaces spezifiziert wurden.)
   - (1) Wann Untertypbeziehung zwischen zwei strukturellen Typen, wenn alle drei Bedingungen stets eingehalten sind:
     - **reflexiv** - die Typen sind Untertyp von sich selbst
       $(Fahrzeug < Fahrzeug)$
     - **transitiv** - ist Typ Reifen Untertyp von Typ Auto, und Typ Auto Untertyp von Typ Fahrzeug, dann ist Typ Reifen auch Untertyp von Typ Fahrzeug
       $(Reifen < Auto,\ Auto < Fahrzeug \Rightarrow Reifen < Fahrzeug)$
     - **antisymmetrisch** - ist ein Typ Auto Untertyp von Typ Quad und Typ Quad ist Untertyp von Typ Auto, dann sind Quad und Auto äquivalent.
       $(Auto < Quad,\ Quad < Auto \Rightarrow Auto \iff Quad)$

- (2) Beliebige strukturelle Typen heißen Fahrzeug und Auto, weitere struk. (deklarierte) Typen Vierrad und Fiat, alle Regeln müssen im Sinne des EP eingehalten werden:
  - Fall 1: **lesender Zugriff auf Konstante in Typ Fahrzeug**, dann gilt:
    - *class Auto { final static Fiat ... }* und *class Fahrzeug { final static Vierrad ... }*
    - **wenn** Auto < Fahrzeug, **dann** $Fiat < Vierrad$
    - Für eine Konstante (= nur lesender Zugriff) mit deklariertem Typ Vierrad im Typ Fahrzeug gibt es eine Konstante mit deklariertem Typ Fiat im Typ Auto, sodass gilt, ist Auto Untertyp von Fahrzeug, so muss auch Fiat Untertyp von Vierrad sein (um die Erwartung bei lesendem Aufruf zu wahren)
  - Fall 2: **lesender und schreibender Zugriff auf (sichtbare) Variable in Typ Fahrzeug**, dann gilt:
    - *class Auto { public Fiat ... }* und *class Fahrzeug { public Ferrari ... }*
    - **wenn** Auto < Fahrzeug, **dann** $Fiat \iff Ferrari$
    - Für eine Variable (= lesender und schreibender Zugriff) mit deklariertem Typ Ferrari im Typ Fahrzeug gibt es eine Variable mit deklariertem Typ Fiat im Typ Auto, sodass gilt, ist Auto Untertyp von Fahrzeug, so muss Fiat äquivalent zu Ferrari sein.
    - **Invarianz** → die deklarierten Typen ($Fiat \iff Ferrari$) sind äquivalent, also die betrachteten Elementypen (hier Elementtyp = Variablentyp, könnte auch Konstantentyp oder Ergebnistyp sein) ändern sich nicht verhältnismäßig zu den Typen, in welchen sie enthalten sind.
  - Fall 3: **gleiche(!) (sichtbare) Methode in Typ Auto und Typ Fahrzeug**, dann gilt:
    - *class Auto { public Ergebnistyp1* ***methodeGleich****(Fiat arg1) {...} }* und *class Fahrzeug { public Ergebnistyp2* ***methodeGleich****(Reifen arg2) {...} }*
    - **wenn** Auto < Fahrzeug, **dann**
      - $Ergebnistyp1 < Ergebnistyp2$
      - $Fiat > Reifen$

  - · **Anzahl der formalen Parameter** in beiden Methoden gleich (arg1 in Auto, arg2 in Fahrzeug, jeweils ein Parameter)
  - ∗ Für gleiche Methoden im Typ Auto und Typ Fahrzeug (methodeGleich()) gilt:
    - · der deklarierte Ergebnistyp der Methode in Auto ist Untertyp der deklarierten Methode in Fahrzeug
    - · der deklarierte Typ (von jedem) formalen Parameter in Auto ist Obertyp des deklarierten Typs des entsprechenden formalen Parameter in Fahrzeug
    - · **Anzahl der formalen Parameter** in beiden Methoden gleich
  - ∗ Begründung:
    - · für Ergebnistyp: es gilt das Gleiche wie für Konstanten/lesender Zugriffe auf Variablen |**Kovarianz** → gleiches Verhalten von Typen (Auto < Fahrzeug) und Ergebnistypen von Methoden und Konstanten, sie variieren in dieselbe Richtung
    - · für formale Parameter: es gilt das Gleiche wie für schreibende Zugriffe auf Variablen |**Kontravarianz** → konträres Verhalten von formalen Parametern, während bei Typen (Auto < Fahrzeug) gilt, gilt Gegenteiliges bei Eingangsparametern ($Fiat > Reifen$), sie varrieren in entgegengesetzter Richtung
    - · (gleiche Anzahl an Parameter: ähnliches Verhalten wie bei strukturellen Typen, daneben braucht jeder formale Parameter eine Entsprechung)

- **Für nominale Typen** (= hier müssen Untertypbeziehungen explizit angegeben/ausgeschrieben werden (durch „extends“ oder „implements“)) **kann man nicht einfach/automatisch Gleichheit wie bei strukturellen Typen annehmen** (gleiche Signatur (alles was public ist und nach außen sichtbar ist gleich), daher Typen gleich))**, daher sind folgende zusätzliche Bedingungen zu beachten:**
  - ***Kovariante Eingangsparameter und binäre Methoden widersprechen dem EP, daher sinnlos, in solchen Fällen Ersetzbarkeit anzustreben.*** (Fausregel)(70)
    (**binäre Methode** = wenn Typ des formalen Parameters einer Methode in einer Klasse gleich dem Typen dieser Klasse (Typ von this) ist (z.B. *class Point2D{ public meth(Point2D arg){...} }* dann ist meth() eine binäre Methode)
  - Da nominale Typen stärker eingeschränkt sind als strukturelle, soll zusätzliche Bedingung gelten: **alle Typen sind invariant** (**außer kovariante Ergebnistypen**) (Begründung: Bei Zugriffen auf Konstanten und Variablen wird statisch, nicht dynamisch gebunden, also braucht man hier keine Ersetzbarkeit.)
    (statisch gebunden = der deklarierte Typ einer Variablen bzw. einer Konstanten ist entscheidend.)

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?

- Was der Compiler prüfen kann betrifft lediglich strukturelle Typen (also automatische Prüfung von Signaturen, statisches binden vom Compiler) und das Überprüfen von deklarierten Typen (im Fall von nominalen Typen) auf Invarianz (im Fall von Variablen und Konstanten).
- Um bei nominalen Typen Ersetzbarkeit (vollständig) zu erreichen, sind neben der Untertypbeziehung weitere Konzepte (Modularisierung, Refaktorisierung, Parametrisierung) in Betracht zu ziehen (vgl.

Faustregel bzgl. binärer Methoden und kovariante Eingangsparameter (70)) bzw. bei übermäßig vielen binären Methoden ist evtl. eine Refaktorisierung mit neuer Hierarchie (neue Interfaces können neue Hierarchien definieren), die Typen mit binären Methoden implementieren, damit ein neues Verhältnis zu (inkompatiblen) anderen Typen bilden und so im Gesamten Wiederverwendbarkeit erzielen. (kurz: Durch Refaktorisierung die Verhältnisse (Hierarchien) neu ordnen, um oben beschriebene Regeln einzuhalten.)

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

   - (siehe Frage 2)
   - **Invarianz**: Deklarierte Typ eines Elements im Untertyp ist äquivalent zum dekl. Typ des entsprechenden Elements im Obertyp. Die betrachtenden Elementypen (hier Typen der Variablen) variieren nicht (zueinander). (z.B. dekl. Typen von Variablen)
   - **Kovarianz**: Deklarierter Typ eines Elements im Untertyp ist Untertyp zum dekl. Typ des entsprechenden Elements im Obertyp. Typen und die betrachteten darin enthaltenen Elementtypen variieren in die gleiche Richtung (z.B. Konstante oder Ergebnistypen von Methoden)
   - **Kontravarianz**: Deklarierte Typ eines Elements im Untertyp ist Obertyp zum dekl. Typ des entsprechenden Elements im Obertyp. Typen und die betrachteten darin enthaltenen Elementtypen variieren in die entgegengesetzte Richtung (z.B. Typen von formalen Parametern)

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

   - **Binäre Methode** sind Methoden, deren formale Parameter (mind. einer) den gleichen Typen hat, wie der Typ der Klasse, in der sie stehen. Binär deswegen, weil der Typ des „this“ und der Typ von mind. einem Parameter gleich sind.
   - (70) Nach der Faustregel widersprechen binäre Methoden dem EP (weil **nicht kontravariante formale Parameter**), daher soll in diesem Fall keine Ersetzbarkeit unter Untertypbeziehung angestrebt werden.
   - z.B.: *class Auto { public Ergebnistyp1* ***methodeGleich****(Auto arg1) {...} }*

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

   - Formale Parameter sind kontravariant zu wählen, d.h. siehe Nr.2
   - **Warum Kontravarianz (intuitiv gesprochen)**: Wenn man einen Typen durch einen Untertypen ersetzt und eine gleichnamige Methode benutzt, so muss der (die) Parameter mindestens vom gleichen Typ sein, auf keinen Fall kann dieser spezieller sein, ansonsten könnte man in der aufgerufenen Methode durch den spezielleren Parameter eine speziellere Operation fordern, die allerdings in der Typanforderung/Typvereinbarung nach außen gar nicht erwartet wird/durch die Zusicherung des servers an den Aufrufer-client nicht zugesichert wurde (Erinnerung bzgl. Typ: Ein Typ ist die Schnittstelle eines Objekts, also alles, was nach außen sichtbar ist und wurde durch eine Klasse spezifiziert. Dabei ist „der geforderte Typ“ erreicht, wenn die in dem Typ beschriebenen Operanden auf die geforderten Operationen passen. Der Aufrufer-client kann also vom server nicht speziellere Operationen (gefordert durch spezielleren formalen Parameter) fordern, die der server dem client vorher nicht zugesichert hatte.).

- **Weitere Bedingung**: **Typen** und **Schnittstellen** sollen auch über Programmänderungen hinweg **stabil bleiben** (Stabilität meist erst nach einigen Refaktorisierungsschritte). Durch statische Bindung werden dynmische Typen nicht beachtet, sodass durch geeignete Hierarchiebildung (durch geeignete Interface-Struktur, Typstruktur (73)) Langlebigkeit und Stabilität gute Wartbarkeit erzielen. Dabei hat eine Änderung ganz oben in der Typstrukur wesentlichere Auswirkungen auf die Wartbarkeit, als Änderungen weiter unten. **Daher besser von Klassen ableiten, deren Schnittstellen bereits stabil sind.** (Fausregel 74) (Bezug: Formale Parameter können ja Teil der (nach außen sichtbaren/public-Methoden) Schnittstelle sein.)
- (Aufbauend aus letztem Punkt:) **Formale Typen sollen eher allgemein und vorausschauend gehalten sein!** (74)

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

- **Dynamisches Binden** ermöglicht im Sinne der besseren Wartbarkeit und Ersetzbarkeit Neuerungen schlicht hinzuzufügen, ohne etwas im ursprünglichen Code ändern zu müssen. Dagegen erhöhen **Switch-Strukturen** bzw. geschachtelte **if-Anweisungen** den Aufwand, wenn ein Programmtext erst einmal groß und unübersichtlich geworden ist und Dinge hinzugefügt werden sollen. Hier müssen alle (!) Stellen gefunden werden, die mit der Änderungen/Neuerung zu tun haben, um die Programmkonsistenz zu wahren. (Das wird sehr schnell fehleranfällig.)

8. Dient dynamisches Binden der Ersetzbarkeit und Wartbarkeit?

- Dynamisches Binden dient beiden. Durch die leichte Ersetzbarkeit (Austausch aber auch Hinzufügen von Neuem) wird die Wartbarkeit erleichtert bzw. nicht durch Auffinden von betroffenen Stellen und dann Nachjustieren im Code unnötigt erschwert.

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

- **Vorbedingung**: Verantwortlich ist der Object-Client. Bspw. wenn es darum geht, Parameter mit bestimmten Vorgaben (dem Server) zu liefern.
- **Nachbedingung**: Verantwortlich ist der Object-Server. Eine Nachbedingung beschreibt meistens das Verhalten einer Methode inklusive Rückgabewert bzw. das Ergbnis der Methode.
- **Invariante**: Der Object-Server ist für Invarianten auf Variablen bzw. Vor- und Nachbedingungen nach Methoden verantwortlich, es sei denn, es gibt direkte Schreibzugriffe des Clients auf Variablen, dann liegt die Verantwortlichkeit beim Client. Eine Invariante impliziert dabei eine Nachbedingung auf jeder Methode des Servers.
- **History-Constraint**: schränken die Entwicklung von Objekten im Laufe der Zeit ein:
  - **Server-kontrolliert**: Ähneln Invarianten, schränken aber zeitliche Veränderungen von Variableninhalten in Objekten ein.
  - **Client-kontrolliert**: Wenn es um die Reihenfolge von Methodenaufrufe (von außen) geht, so kann das nur der Client einschränken und sind daher dafür verantwortlich. (Bspw. initialize() vor allem anderen)

10. Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten,

damit das Ersetzbarkeitsprinzip erfüllt ist? Warum? **(Hinweis: Häufige Prüfungsfrage!)**

- Auch für Zusicherungen gilt das Ersetzbarkeitsprinzip EP. Dabei ist Typ Auto Untertyp vom Typ Fahrzeug (Auto < Fahrzeug). (Dabei ist mit Methode aus Auto die entsprechende Methode aus Fahrzeug gemeint.)
- **Vorbedingung**:
  - Vorbedingung { Methode aus Auto } ≤ Vorbedingung { Methode aus Fahrzeug }
  - Jede Vorbedingungen auf Methoden im Untertyp dürfen nur schwächer oder gleich stark sein, weil Aufrufer eigentlich nur Zusicherungen zu Methoden aus dem Obertyp kennt und daher auf Spezialisierungen nicht eingehen kann.
  - **Oder-Verknüpfung** von Vorbedingung aus Obertyp und die aus Untertyp machen die Vorbedingung aus Untertyp gleich/schwächer, mehr Auswahlmöglichkeiten.

- **Nachbedingung**:
  - { Methode aus Auto } Nachbedingung ≥ { Methode aus Fahrzeug } Nachbedingung
  - Jede Nachbedingung auf eine Methode im Untertyp dürfen nur stärker oder gleich stark sein, weil Aufrufer eigentlich nur Obertyp kennt und als Ergebnis mindestens das erwartet, was der Obertyp garantiert, nicht jedoch weniger.
  - **And-Verknüpfung** von Nachbedingunen aus Obertyp und die aus Untertyp machen Nachbedingung aus Untertyp gleich/stärker, weil man mehr erwarten kann.

- **Invariante**:
  - {Invariante Auto } ≥ {Invariante Fahrzeug }
  - Jede Invariante im Untertyp darf nur stärker oder gleich stark sein, weil Aufrufer eigentlich nur den Obertyp kennt und als Ergebnis mindestens das erwartet, was der Obertyp garantiert, nicht jedoch weniger. Hier hängen Invarianten und Nachbedingungen nach Methodenausführungen zusammen, denn Invarianten werden in der Regel nur durch Methodenaufrufe manipuliert (Invarianten implizieren Nachbedingungen auf allen Methoden).
  - **And-Verknüpfung** von Invarianten aus Obertyp und die aus Untertyp machen Invarianten aus Untertyp gleich/stärker, weil man mehr erwarten kann.

- **Server-kontrollierte History-Constraint**:
  - {SK History-Constraint Auto } ≥ {SK History-Constraint Fahrzeug }

  - Gleiches wie für Invarianten, wobei schwer von stärkeren oder schwächeren Bedingungen zu sprechen ist.
  - **And-Verknüpfung** von Invarianten aus Obertyp und die aus Untertyp machen Invarianten aus Untertyp gleich/stärker, weil man mehr erwarten kann.

- **Client-kontrollierte History-Constraint (HC)**:
  - {CK HC Traces Methode aus Auto } ≤ {CK HC Traces Methoden aus Fahrzeug }
  - Das Gleiche wie Vorbedingungen (bezogen auf die Reihenfolge der Methodenaufrufe = **Trace**). Aufrufreihenfolgen von Methoden im Untertyp müssen gleich oder schwächer (mehr Aufrufreihenfolgen erlaubt) geregelt sein als im Obertyp. Wenn Aufrufer nur Obertyp kennt und dessen Reihenfolge, so darf diese im Untertyp nicht spezieller sein. Allerdings ist bei der Überprüfung von CK HC die Menge aller Clients zu kontrollieren
  - Teilmengenbeziehung zwischen Trace-Set des Fahrzeugs und des Autos:
    $\{TraceSet_{Fahrzeug} \subseteqq TraceSet_{Auto} \mid TraceSet = alle\ möglichen\ Traces\}$

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

- Für nominale Typen werden neben Signatur und Name (bereits schon Teil der Zusicherung) auch alle Zusicherungen (die nach außen durch Kommentare kommuniziert werden) mitdazugezählt. Zusicherungen enthalten alle über die Abstraktion durch Namen hinausgehenden Vertragsbestandteile, die nicht vom Compiler geprüft werden können.
- **Warum**: Zusicherungen sollen aufgrund der besseren Wartbarkeit stabil bleiben. Ähnlich wie bei Signaturen in der Typhierarchie soll es bei Refaktorisierungen oder allg. Programmänderungen möglichst zu keinen oder sehr eingeschränkten Änderungen in der Signatur, aber auch in den Zusicherungen kommen. Andernfalls kann es durch (aufgrund von Änderungen entstandenen) falsche Zusicherungen zu schweren Missverständnissen und Fehlern kommen, die kaum nachzuvollziehen sind.
- **Faustregel**: *Zusicherungen sollen stabil bleiben. Das ist für Zusicherungen in Typen nahe an der Wurzel der Typhierarchie ganz besonders wichtig.* (83)
- **Wo**: Besonders an der Wurzel der Typhierarchie (also jenem Interface, welches der Ursprung für viele weitere Interfaces (extend) oder Klassen (implement) als Ableitung bedeutet, bzw. jene Klasse als Ursprung, wenn es sich dann um Vererbungsbeziehungen zwischen Klassen handelt) sollten Zusicherungen stabil sein (und daher eher einfach gehalten (siehe Frage 13)).

12. Was ist im Zusammenhang mit allgemein zugänglichen

(= möglicherweise nicht nur innerhalb des Objekts geschriebenen) Variablen und Invarianten zu beachten?

- Für Invariant allgemein sind zwar hauptsächlich Server, aber auch Clients - wenn sie direkt darauf zugreifen - zuständig, besonders hinsichtlich allgemein zugänglichen Variablen, da diese Variablen ja von allen möglichen Stellen aus verändert werden können. Um die Verantwortlichkeit zu beachten ist es händisch notwendig zu ermitteln, wie, wo, wann und wer auf diese zugreift, um daraus dann notwendige Zusicherungen zu formulieren.

13. Wie genau sollen Zusicherungen spezifiziert sein?

- Steht in Zusicherungen nur das Nötigste, sind Clients und Server relativ unabhängig voneinander. Der Typ ist bei Programmänderung stabil. Aus Gründen der Wartbarkeit sollten Zusicherungen nur dort eingesetzt werden, wo tatsächlich Informationen gebraucht werden, die über die der Signatur hinausgehen (Unabhängigkeit Server-Client wahren).
- **Ziel Zusicherungen**: ausreichend viel Info für **Verhaltensbeschreibung**, so wenig wie möglich aus Gründen der **Wartbarkeit**, maßvoll eingesetzt um den **Klassenzusammenhalt** zu maximieren und die **Objektkopplung** zu minimierern und damit die **Ersetzbarkeit** weiterhin zu gewährleisten.
- **Faustregel**: *Zur Verbesserung der Wartbarkeit sollen Zusicherungen keine unnötigen Details festlegen.* (84)
- **Faustregel**: *Alle von geschulten Personen nicht in jedem Fall erwarteten, aber vorausgesetzten Eigenschaften sollen explizit als Zusicherungen im Programm stehen.* (84)
- (siehe Frage 11)

14. Wozu dienen abstrakte Klassen und abstrakte Methoden?

Wo und wie soll man abstrakte Klassen einsetzen?

- **Allgemein**: Von abstrakten Klassen lassen sich keine Objekte direkt bilden (es kann aber Objekte des Typs einer abstrakten Klasse geben - deklarierte Typ kann eine abstrakte Klasse sein, nicht aber der dynamische Typ). Interfaces sind eine besondere Form von abstrakten Klassen, weil sie keine Objektvariablen (Trait) zulassen.
- **Konkrete Bildung**: *public abstract class Polygon { public abstract void draw()**{hier keine Implementierung möglich}**; }*
- **Wozu**: Sie bilden jene stabile Obertypen, von denen Obertypen/deklarierte Typen von Objekten, aber auch formale Parameter sein sollten. Selber können sie keine Implementierung (Body/Methodenrumpf) haben, damit dienen sie lediglich im Sinne der Typhierarchie als Vorlage/Blueprint für Klassen, die sich von diesen mittels Vererbung ableiten. Demnach haben abstrakte Klassen eine strukturierende Aufgabe innerhalb des Typhierarchiegebildes eines Programms.
- **Wo** und **wie**: Abstrakte Klassen soll man aus Stabilisierungsgründen als deklarierter Typ für Objekte oder für formale Parameter in Methoden nutzen (da sie wie gesagt aufgrund fehlender Implementierung eher stabil sind).

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

- Es ist nicht (wirklich) das gleiche, da Vererbung zur direkten Codewiederverwendung ohne Einschränkungen benutzen kann. Generell implizieren beide (Vererbung, EP) Beziehungen zwischen Klassen, jedoch unterscheidet man 3 Beziehungsarten:
  - **Untertyp** mit EP, Untertypen setzen Vererbung voraus
  - **(reine) Vererbung** mit Code aus der Oberklasse (die man abändern kann), wobei EP irrelevant ist, weil es nicht wie bei der Untertypbeziehung (hstl. nominaler Typen) keine Einhaltungspflicht hstl. Zusicherungen gibt
    - **Ziel der reinen Vererbung**: So viele Teile der Oberklasse wiederverwenden, wie möglich, auch wenn dadurch Zusicherungen zerstört werden (pragmatischer Gesichtspunkt, kein konzeptioneller). (94)

  - **Is-a-Beziehung** sind abstrakte Einheiten, kommen in der frühen Entwicklungsphase vor und entwickeln sich im Laufe der Entwicklung/Refaktorisierungsschritte zu Klassen weiter. (z.B. Beziehung „Student ist eine Person“ mit abstrakten Einheiten „Student“ und „Person“)
- **Unterschied:** Aus Sicht des Compilers gibt es keinen Unterschied, da Untertypbeziehungen Vererbungsbeziehungen voraussetzen und vom Compiler überprüfbare Signaturen und Namen mit Ableitungen (extend) entsprechend gewählt werden müssen. Nur hstl. von Zusicherungen gibt es einen Unterschied, die allerdings nur händisch überprüfbar ist - es stellt sich dann nur die Frage: „Sind Zusicherungen zwischen Oberklasse und Unterklasse kompatibel?“.

16. Worauf kommt es zur Erzielung von Codewiederverwendung eher an,

auf Vererbung oder Ersetzbarkeit? Warum?

- Es kommt klar auf die Ersetzbarkeit an, diese ermöglicht Wiederverwendung im großen Stil und nicht bloße Erhöhung von Codewiederverwendung im Kleinen.
  Warum: Ersetzbarkeit hat eine **konzeptuelle Komponente**, Typhierarchien sind konzeptuell und ist nicht wie bei (reiner) Vererbung bloß **pragmatisch** angelegt. D.h., dass neben reiner Überprüfung des Namens und der Signatur auch Zusicherungen, die durch reine Vererbung verloren gehen (können), zu berücksichtigen ist, um den Sinn des Programms zu erhalten (ansonsten kann es zu unerwünschten Programmverhalten kommen). Daneben erhält das EP wie Wartbarkeit und wartet Klassenzusammenhalt und Objektkopplung. All diese Aspekte opfert man bei reiner Vererbung für ein wenig Programmcodeersparnis. **Daher ist die Untertypbeziehung (EP) der reinen Vererbung klar vorzuziehen.**
- Jedoch **kann** sich auch **reine Vererbung positiv auf die Wartbarkeit auswirken**, wenn es bspw. darum geht, alle Stellen einer Programmänderung durch Vererbungsbeziehung (automatisch) ausfindig zu machen, in dem man etwas weiter oben in der Vererbungshierarchie ändert und es sich automatisch weiter „unten“ auswirkt.

17. Was bedeuten folgende Begriffe in Java?

- (1) Objektvariable, Klassenvariable, statische Methode
- (2) Static-Initializer
- (3) geschachtelte und innere Klasse
- (4) final Klasse und final Methode
- (5) Paket, Class-Path, import-Anweisung

- (1) Objektvariable, Klassenvariable, statische Methode:
  - **Objektvariable** = auch **Instanzvariable**, ist eine Variable, gehört zu einem Objekt. Falls im Objekt für eine deklarierte OV keine Initialisierung gegeben ist, so wird eine Defaultinitilisierung vorgenommen (nicht so für lokale Variablen).
  - **Klassenvariable** = gehören zu keinem bestimmten Objekt, sondern mittels „static“ direkt zur Klasse - sie gehören also allen Objekten der Klasse gemeinsam, d.h., dass bei einer *class Fahrrad {private static int value = 0; ...public void increaseValue(){this.value++;}* zwei Objekte obj1 und obj2 vom Typ Fahrrad durch abwechselndes

obj1.increaseValue() und obj2.increaseValue() stets Fahrrad.value erhöhen.

- **Statische Methode** = ebenfalls wie Klassenvariablen gehört diese Methode nicht einem Objekt, sondern ist direkter Teil der Klasse, d.h. dass eine Methode *public static void infoAboutThisClase(){...}* nicht mit einem Objekt aufgerufen wird, sondern direkt mit dem Klassennamen: Fahrrad.infoAboutThisClass().

- (2) **Static-Initializer** = „Konstruktor“ für Klassenvariablen innerhalb der Klasse mit *class Fahrrad{public static int var; class{var = 42;} }* , also einfach nur mit *static {...}* initialisiert, wobei jede Klasse beliebig Static-initializer haben kann.

- (3) Geschachtelte und innere Klasse:
  - **Innere Klasse** = Innerhalb einer Klasse mit *(private) class Node{...}* innerhalb einer Klasse definiert, wobei „public“ wenig Sinn macht, da die innere Klasse nur für die umgebende Klasse gedacht sein sollte. Bspw. bei Listenklassen sehr sinnvoll die Node in der List-Klasse direkt zu platzieren. Methoden und Objektvariablen (auch private) und auch Klassenvariablen sind in inneren Klassen ebenfalls nutzbar. Allerdings dürfen inneren Klassen selber keine statischen Methoden oder keine statisch geschachtelten Klassen beinhalten, sehr wohl aber noch weitere innere Klassen.
  - **Geschachtelte Klasse** = Innerhalb einer Klasse mit *static class Node{...}* definiert bzw. mit dem Schlüsselwort „static“ und gehören zu umschließenden Klasse. Außen können sie mit *Fahrrad.InnererKlassenName() = new Fahrrad.InnererKlassenName("valueString", null);* aufgerufen und benutzt werden. Bspw. bei Listenklassen ebenfalls sehr sinnvoll die Node in der List-Klasse direkt zu platzieren.

- (4) final Klasse und final Methode:
  - **final Methode**: final verhindert das die Methode aus der Oberklasse in einer Unterklasse überschrieben wird.
  - **final Klasse**: Diese Klassen haben keine Unterklassen. (Abstrakte Klassen dürfen natürlich nicht final sein.) Sinnvoller Einsatz: Verwendung von Interfaces und abstrakten Klassen, sobal eine Klasse diese implementiert bzw. extendet, soll diese final sein (Programmierstil wie bei Sather), dieser Stil erhöht die Verlässlichkeit, nicht ein Objekt einer Klasse zu haben, welches mit überschriebenen Methoden arbeitet.

- (5) Paket, Class-Path, import-Anweisung
  - **Paket**: Das Verzeichnis (was dem Paket entspricht) enthält alle compilierten Java-Klassen (Endung *.class* mit Maschinencode „cafe babe 0000 0037 00d5 0a00...“) (Endung *.java* ist der Quellcode mit den „normalen“ Java-Code) und das Verzeichnis bildet einen Namensraum. Der Name des Verzeichnisses ist der Paketname, zu dem die Klasse gehört.
  - **Class-Path**: Ist an der Basis der Verzeichnisstruktur für Java-Klassen mit den Quellcode-Dateien (Endung *.java*)
  - **import-Anweisung** (und **Pakete)**: Falls man eine statische Methode aus einer Klasse importieren will, ist Folgendes zu beachten:
    - **ganz oben in einer jeden Klasse** steht **das Paket**, in der die Klasse steht (bspw. „package com.jetbrains;“, weil die Klasse im Ordner „com“ mit Unterordner “jetbrains“ steht)
    - **import eines anderen Pakets**: Zusätzlich zum Paket „com.jetbrains“ kann man aus einem anderen Paket bspw. „testPaket.test"(mit Ordner „testPaket“ und Unterordner „test“, wo alle gewünschten Klassen stehen, die man importieren will) Klassen importieren (Paketname = Name und Pfad des Pakets)
    - **Verwendung**:

- nur (!) oben in der Klasse, in der das importierte Paket verwendet werden soll, schreibt man unter „package [Paketname];“ *import testPaket.test.** (bzw. statt Stern eine genaue Klasse, die importiert werden soll)
- im Code kann man nun nur(!) alle **statischen** Methoden der importierten Klasse verwenden, indem man den Klassennamen voranstellt: *TestKlasse.testMethode()*

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

- **Klassen unterstützen nur Einfachvererbung** (=nur einen direkte Vorgänger in Vererbungshierarchie). Eigenschaften Einfachvererbung:
  - **Variablen** in der Unterklasse überdecken jene aus der Oberklasse (überschreiben diese aber wie bei Methoden nicht, sondern überdecken diese bloß und exisiteren somit gleichzeitig, somit kann man diese aus der Oberklasse in der Unterklasse direkt ansprechen mit *super* oder direkter mit *((Oberklasse)this.varName*)
  - **Methoden** (überschrieben oder nicht) kann man nur mittels *super* ansprechen (also nicht mit Typumwandlung = ändern den deklarierten Typ)
  - **statische Methoden** aus Oberklassen kann man normal mit vorangestelltem Klassennamen nutzen
  - **Überschreiben** versus **Überladen**: nur wenn Methodenname, Parameteranzahl und Parametertypen jeweils gleich, dann kann man überschreiben, ansonsten überladet man
- **Interfaces erlauben Mehrfachvererbung**:
  - **Interfaces**:
    - beginnen mit *interface*
    - enthaltene Variablen sind immer *static-final*-Konstanten
    - mit *default* als Modifier bezeichnet man nicht-statische Methoden und ohne default statische Methoden
    - Methoden und Konstanten sind auch ohne Modifier *public*
    - nach *extends* können mit Kommata getrennt mehrere andere Interfaces stehen, von denen das Interface erbt
    - Interface immer vor abstrakten Klassen (nur wenn eine Objektvariable benötigt wird) verwenden
  - **Klassen - keine Mehrfachvererbung**: Für Klassen ist es immer nur maximal möglich, mit *extend* von genau einer Klassen zu erben. Stattdessen bzw. zusätzlich kann eine Klasse aber mit *implement* von mehreren Interfaces erben (besser: abgeleitet sein).
    (Es ist möglich: *public class Rectangle extends Polygon implements Paper, Math {...}*)
    Zusätzlich gilt, wenn eine Klasse ein Interface implementiert, welches eine default-Methode besitzt (*default double draw(){hier kann selbst im Interface wegen default eine Implementierung stehen}*), so muss die Klasse diese Methode nicht extra bei sich implementieren.

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

- Es gibt im Wesentlichen zwei Arten von import-Deklarationen:
  - import: „Normal“ über import im Kopf der Klasse, wobei man mit „ordnerName.*“ (Stern) alle Klassen aus einem Paketordner auswählen kann.
    bpsw. *import testPaket.test.*;* → danach dann Verwendung der Klasse und der Methode im Programmtext mit ***TestKlasse.testMethode(argument);***

  - direkte Nutzung im Programmtext durch vollständige Angabe des Verzeichnis-Paths: bspw. ***testPaket.test.TestKlasse.testMethode(argument);***

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

- Es gibt am Kopf (als oberste Zeile vor allem anderen) eine einzige Zeile mit der *„package“*-Anweisung. Diese gibt den Namen und das Paket an, in welchem die Klasse in der Quelldatei gehört, bspw.: ***package testPaket.test;*** (Klasse im Ordner testPaket mit dem Unterordner test, $testPaket \rightarrow test \rightarrow TestKlasse$) (relativ zum Class-Path)
- Zeile mit Schlüsselwort *package* stellt also sicher (zu einem gewissen Grad), dass Datei nicht einfach aus dem Kontext gerissen in einem anderen Paket steht

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

- Sichtbarkeitstabelle (112):

Sichtbarkeit/Ererbarkeit	public	protected	default	private
sichtbar im selben Paket	ja	ja	ja	nein
sichtbar im anderen Paket	ja	nein	nein	nein
ererbar im selben Paket	ja	ja	ja	nein
ererbar im anderen Paket	ja	ja	nein	nein

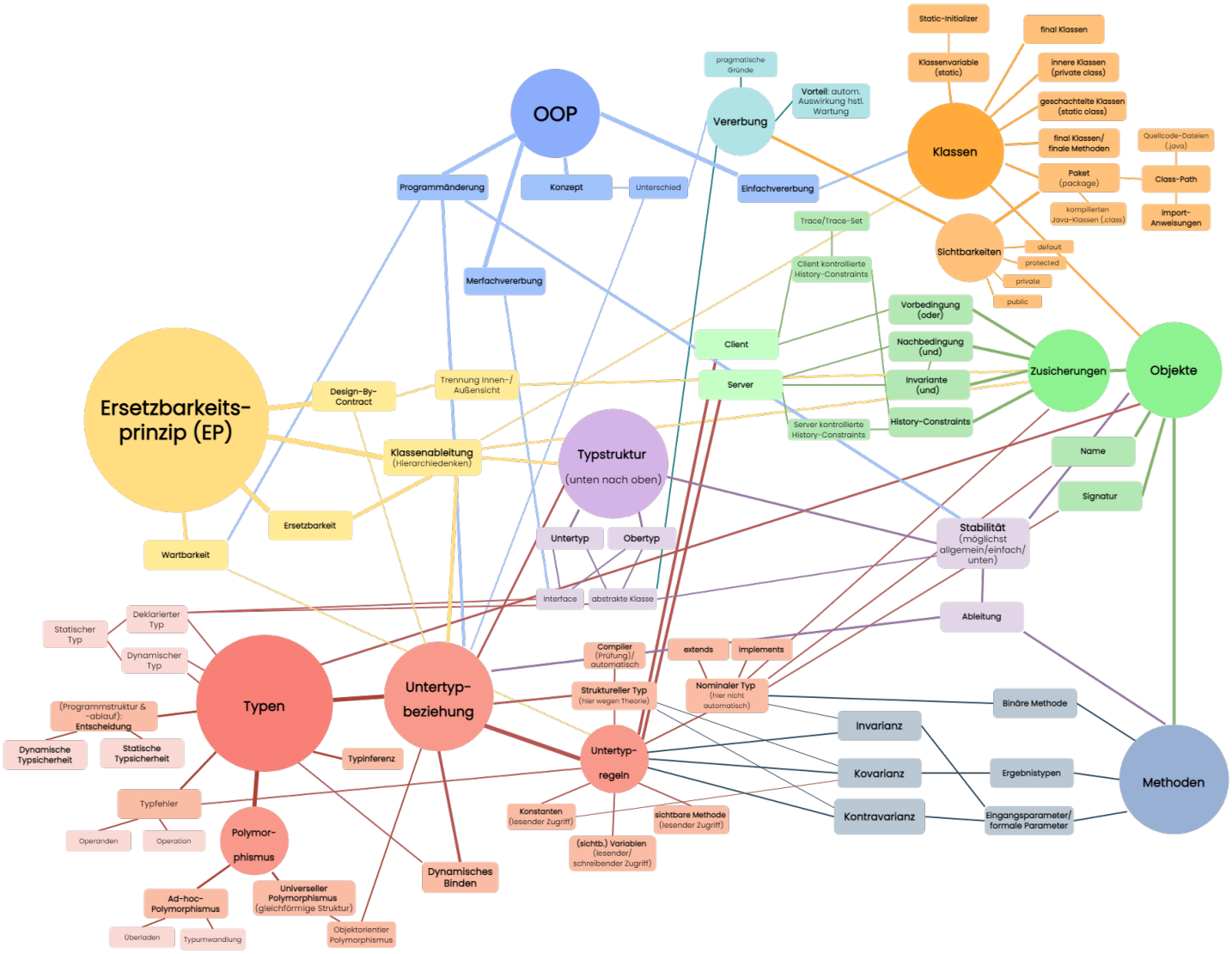
- Wann:
  - **public**: Methoden, Konstruktoren und Konstanten (von Klassen/Objekten), die man außen benötigt (→ Signatur). Variablen eher nicht.
  - **private**: Alles, was nur innerhalb der Klasse verwendet werden soll.
  - **protected**: → Bedeutung: Wenn Methoden und Konstruktoren (gelegentlich auch Variablen) nicht für die Klasse/Objekte, aber für die Unterklassen zugreifbar sein sollen (Stichwort wann). Protected Variablen meiden und nur indirekter Zugriff in Unterklassen auf protected Variablen. Einheiten sind im gleichen Paket sichtbar, in anderen nicht. Generell gilt protected als schlechter Stil.
  - **default**: Wenn weder public, private noch protected. Default-Sichtbarkeit fast gleich wie protected, allerdings falls man Vererbung an andere Pakete zusätzlich ausschließen will, sollte man default verwenden (Stichwort wann).

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

- **Unterschied**: Interfaces können im Gegensatz nur begrenzt Variablen bzw. Konstanten (public static ...) und nur begrenzt Methodenimplementierung (default-Methoden) haben. Abstrakte Klassen hingegen können all das haben, was normale Klassen haben können, außer, dass kein Objekt sie instanzieren kann.

- **Einsatz Interfaces**: Interface dienen hauptsächlich als Schnittstelle und „Organisator“ in der Typhierarchie bzw. um im Programm Untertypbeziehungen an konkreten Programmpunkten umzusetzen/festzulegen.
- **Einsatz abstrakte Klasse**: Falls Objektvariablen und (reine) Vererbung benötigt wird, können abstrakte Klassen nützlicher sein.
- **Zusammenfassend**: Analog zur Faustregel *„Interfaces sind Klassen vorzuziehen.“* (110) soll im Sinne der EP auf Interfaces zurückgegriffen, weil diese hauptsächlich Untertypbeziehungen umsetzen und zusätzlich hstl. der Wartbarkeit eher stabiler als abtrakte Klassen sind (beachte zurückhaltende Implementierung und ausgeprägte und stabile Zusicherungen in Interfaces). Hier greift dann die Faustregel, dass man eher von stabilen Klassen erben/ableiten soll.

Kapitel 2 | Themenübersicht



Kapitel 3 (Generizität und Ad-hoc-Polymorphismus)

1. Was ist Generizität?

   Wozu verwendet man Generizität?

   - **Was**: Ein statischer Mechanismus/Sprachkonzept, der in generischen Typen, Klassen und Methoden enthaltenen Parameter (einfache (Typ-)Namen statt explizite Typen) Typen einsetzt (→ Typparameter).
   - **Wozu**: Um Programmstücke ohne Festlegung auf einen bestimmten Parameter nur einmal zu schreiben. Bspw. ist dies nützlich bei einer Erzeugung für einen Listentyp, wobei noch nicht bekannt ist, von welchem Typ die Elemente der einzlenen Nodes sein werden.

     ```
     public class List<A> implements Collection<A> {...}
     ```

   - **Werte bei Typparamter**: Statt für gewisse **Elementartypen** wie int, char und boolean sind entsprechende **Referenztypen** (Integer, Character oder Boolean) zu nutzen. ⇒ z.B.:

     ```
     List<Integer> list = new List<>();
     ```

     Außerdem ist einschränkend die Erzeugung von neuen Objekten mit *new A()* verboten.
   - **Umwandlung zwischen Elementar- und Referentypen** (z.B. int $\rightleftarrows$ Integer) mit **Autoboxing** (und **Autounboxing**) passiert automatische Typumandlung, daher statt *xs.add(new Integer(0))* einfach nur *xs.add(0)*. Das heißt, dass Typumwandlungen (Cast) zwischen elementaren Typen (8 in Java: int, double, char, boolean, byte, short, long, float; elementare Typen sind statisch-typisiert, d.h., dass sie vor Benutzung zuerst deklariert werden müssen) und Referenztypen (Entsprechungen zu elementaren Typen: Integer, Double, Character, Boolean, Byte, Short, Long, Float; aber auch alle anderen Referenztypen wie bspw. Auto, Square, Point, ..., wobei die Konstruktoren der jeweiligen Klassen die Referenz zu einem Objekt returnieren) erlaubt sind (144).

     ```
     int zahl = 1;
     Integer zahl2 = 2;
     (Integer)zahl; /*erlaubt*/
     (int)zahl2; /*erlaubt*/
     ```

   - **Tipp**: Will man sich bzgl. des Typs beim Aufruf noch nicht festlegen, so nimmt man einfach den Typ von allen Typen, „Object“:

     ```
     List<Object> liste = new List<>();
     ```

   - **Generizität bietet statische Typsicherheit**: inkompatible Typen in Programmstruktur werden statisch erkannt und als Fehler gemeldet, durch **Typinferenz erkennt der Compiler** anhand der Typdeklarationen beim Aufruf (also vor Laufzeit), welche Typen eingesetzt wurden.
   - **Vorteil**: Generizität verringert Schreibaufwand, erhöht statische Typsicherheit, erhöht Lesbarkeit (und Übersichtlichkeit im gesamten Programm)
   - **Generizität bei Methoden**: bspw.

     ```
     public A getVar(A param) {A var = param; return var;}
     ```

bzw.

```
public <A,B> A getVar(A param1, B param2) {A var = param; return param1;}
```

⇒ A ist der Ergebnistyp, Ausdruck <A,B> gibt Auskunft über die in der Methode (Parameter und Ergebnistyp) verwendeten Typen. **Zusatz**: Es ist dabei Folgendes zu beachten:

```
public class List<A> implements Collection<A> {
...
public <A, B, M, N> M getVar(A param1, M param2, B param3, int param4) {A var = param1;
return param2;}
... }
```

<A, B, M, N> ist der Ausdruck, der die Typvariablen speziell für diese Methode einführt. Falls der Typparameter für die Klasse <A> heißt, so wird dieser bei dieser Methode durch <A, B...> versteckt, d.h., da der Typparameter eh schon allgemein in der Klasse gilt, ist es unnötig, diesen noch einmal extra in der Methode im voranstehenden Ausdruck zu schreiben, um damit den deklarierten Typ A für das Argument param1 zu bezeichnen.

- **keine implizite Untertypbeziehung**, d.h., ist

```
class MyList<A> extends List<List<A> > {...}
```

, dann ist *MyList<String>* Untertyp von *List<List<String> >*, **aber *MyList<X>* ist kein Untertyp von *List<Y>*** (beachte hier untersch. Typen X und Y, meint den generellen Fall). **Implizit** heißt hier, dass die Untertypbeziehung zwischen den Klassen MyList und List nicht auf die Typparameter übertragen wird/dass die Untertypbeziehung von MyList und List nicht auch eine Untertypbeziehung zwischen X und Y impliziert.

2. Was ist gebundene Generizität?

Was kann man mit Schranken auf Typparametern machen, das ohne Schranken nicht geht?

- **Problem**: Über den einzusetzenden Typ, der Typparameter ersetzt, ist nichts bekannt → nicht bekannt, ob und welche bestimmte Methoden oder Variablen Objekte von diesem Typ haben ⇒ **Schranken**
- **Gebunden**:
  - **Schranken**: **Typparameter sind gebunden**, d.h. gebe pro Typparameter bspw. (max.) eine Klasse oder beliebig viele Interfaces als Schranke an → nur Untertypen dürfen Typparameter ersetzen (somit statisch bekannt, welche Signatur zu erwarten ist → Obertyp) (**verwende Objekte des Typparameters wie Objekte der Schranke**):
    - z.B.

      ```
      public class Scene <T extends Scalable> implements Iterable<T> {...}
      ```

      , wobei Scalable die Schranke (Obertyp) darstellt, es steht immer *extends*, niemals *implements*, mehrere Schranken mit „&“ verbunden
    - gebunden heißt „mit Schranke“, ungebunden hat keine Schranke bzw. Object ist dann Schranke/Obertyp

- **Was**: Schranken erlauben es einem, Methoden und Variablen (Verhalten) von einem Typen, der in den Typparameter eingesetzt wird, erwarten zu können. Dadurch kann man besser planen, was bei einfacher Generizität nicht geht. Bspw. ist die Klasse für Integer ausgelegt mit Methoden, die etwas errechnen, so wird man mit dem Typ String Probleme bekommen. Die Einschränkung durch Schranken ermöglicht auf diese Weise paradoxerweise mehr Freiheit in der konkreten Implementierung, da man sich auf mehr verlassen kann.

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

- Einsatz immer sinnvoll, wenn die Wartbarkeit verbessert wird (bspw. durch bessere Lesbarkeit/ Übersichtlichkeit). Ob das zutrifft kann durch folgende **Faustregeln** ermittelt werden:
  - **Gleich strukturierte Klassen und Methoden (bessere Lesbarkeit)**, also wenn das Grundkonzept einer Klasse gleich ist (bspw. Klassen: Container, Listen, Stacks, Hashtabellen, Mengen etc.) oder bei Methoden mit typischen „Containerfunktionen“ (Container=Sammelbecken für alles Mögliche) wie bspw. Such- oder Sortierfunktion.
    Faustregel 1: *„Containerklassen sollen generisch sein.“* (130)
    Faustregel 2: *„Klassen und Methoden in Bibliotheken sollten generisch sein.“* (131) → bezogen auf Programmentwicklung
  - **Abfangen erwarteter Änderungen (EP wegen Untertypbeziehungen**, Ersetzbarkeitsprinzip in formalen Parametern von Methoden umsetzen, um die Wartbarkeit bei Programmänderungen einzugrenzen. Formale Parameter in Methoden sollten daher Typparameter sein (durch gebundene Wildcards noch genauere Steuerung möglich)). Z.B. für Kontomethode *addMoney(A Betrag)*, wobei A vom Interface-Typ Currency ist und dadurch Euro- oder Dollartypen (mit $Euro < Currency$ und $Dollar < Currency$) einsetzbar sind. (hier absehbar = Einsatz von Software in untersch. Ländern)
    Faustregel 3: *Man soll Typparameter als Typen formaler Parameter verwenden, wenn Änderungen der Parametertypen absehbar sind.*
  - **Verwendbarkeit**, Generizität und Untertypbeziehungen sind oft gegeneinander austauschbar, aber nicht generellles Ersetzen gegeneinander möglich, vergleiche:

  Fall A Eine **homogene** Liste (Elemente nur vom selben Typ) kann **mit Generizität** ohne Untertypbeziehung herstellen und dieses statisch sicherstellen.

  Fall B Eine **heterogene** Liste (Elemente haben untersch. Typen) ist **mit Generizität *ohne* Untertypbeziehung** nicht möglich (Untertypbeziehung in Generizität durch gebundene Wildcards mit (Ober-/Unter-)Schranken).
  Faustregel 4: *Untertypbeziehungen und Generizität ergänzen aber ersetzen sich nicht gegenseitig! Untertypbeziehungen ist Generizität prinzipiell vorzuziehen.* (eigene Faustregel nach (133))
  - **Laufzeiteffizienz**, zwar hat die Verwendung von Generizität keine bzw. kaum Auswirkung auf die Laufzeit (dynam. Binden eher schon), dennoch zahlen sich solche Optimierung in der Regel nicht aus.
    Faustregel 5: *Man soll Überlegungen zur Laufzeiteffizienz beiseite lassen, wenn es um die Entscheidung zwischen Generizität und Untertypbeziehungen geht.* (134)
  - **Natürlichkeit**, auch, wenn aufgrund von Erfahrung man zu einem Mechanismus tendiert (Generizität versus Subtyping), sollte man auch eine Kombination immer in die Überlegung miteinbeziehen.

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

- **statische Typsicherheit in Zusammenhang mit Generizität**: Statische Typsicherheit lässt sich durch die Typparameter herstellen, das heißt, dass der Compiler bereits statisch anzeigen kann, ob Untertypbeziehungen eingehalten worden sind, wenn Typnamen bei Aufruf von generischen Klassen oder generischen Methoden konkret angegeben werden müssen (Stichwort Typinferenz). Zudem lässt sich durch gebundene Generitzität oder auch durch gebundene Wildcards stärke bzw. präzisere Einschränkungen bzgl. der gegebenen Typhierarchie herstellen.
- **dynamische Typabfragen → dynamischer Typ**:
  - **instanceof-Operator**: Fragt, ob der dynamische Typ eines Referenzobjekts Untertyp eines gegebenen Typs ist. Mit ($Auto < Fahrzeug$) und ($Auto\ a = new\ Auto();$) ergibt die dynamische Typabfrage ($a\ instanceof\ Fahrzeug$) **true**.
    Wäre ($Reifen < Auto$) und ($Auto\ a = new\ Auto();$), so ergibt die dynamische Typabfrage ($a\ instanceof\ Reifen$) **false**.
  - **getClass()**: Mit dieser Abfrage lässt sich der dynamische Typ ermitteln, bspw. bei ($Fahrzeug\ a = new\ Auto();$) wäre das Auto.
- **Typumwandlung**: → **deklarierter Typ**
  Cast ist eine explizite Typumwandlung, die den **deklarierten Typ eines Objekts** innerhalb der Typhierarchie aufwärts (also einen spezielleren Typ zu einem allgemeineren Typen) oder abwärts (also einen allgemeineren zu einen spezielleren Typen machen) umwandelt. Dabei aufpassen, dass es durch die Casts nicht zu einer *ClassCastException* (CCE) zur Laufzeit kommt, da auf einmal der deklarierte Typ spezieller als der dynamische Typ ist. Auch kann es zu einer CCE kommen, wenn zwischen Untertypen gecastet werden, die sich zwar von einem gemeinsamen Obertyp ableiten, aber ansonsten auf gleicher Ebene innerhalb der Typhierarchie stehen (bspw.: $Fahrzeug\ f = new\ Auto()$; → (Fahrrad)f (mit $Fahrrad < Fahrzeug$) ⇒ CCE, da es nun theoretisch heißen würde: $Fahrrad\ f = new\ Auto()$.
  Zu einem Typen außerhalb der Typhierarchie casten würde bereits statisch durch den Compiler als einen *inconvertible type* angezeigt werden.
  - **Hier speziell Bezug zur Generizität:** Bei der **homogenen Übersetzung** einer generisch definierten Klasse in eine nicht-generisch definierten Klassen bedarf es (speziell für die Rückgabetypen) Casts. Bspw. können generische Iteratoren mit der next()-Methode alles Mögliche an Typen zurückgeben (je nach eingesetztem Typparameter). (146) → Bsp.:

```
List xs = new List();
xs.add((Integer)new Integer(0));
Integer x = (Integer)xs.iterator().next();
```

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

- **Problem**: Generizität unterstützt nicht impliziete Untertypbeziehung, wie bei Arrays, d.h z.B.

```
void drawAll(List<Polygon> p) {...}
```

  kann man beim Einsetzen in die Parameter nicht $List<Polygon>$ mit $List<Triangle>$ ersetzen, weil man implizite Untertypbeziehung ($Triangle < Polygon$) annimmt.
  Vom Parameter p darf nur gelesen werden (Compiler prüft und gibt eine Fehlermeldung, wenn für die geforderten Typen in Untertypbeziehung **Kovarianz** bspw. durch schreiben nicht eingehalten wird).
  Wo das Verbot von implizite Untertypbeziehung wegen der statischen Typsicherheit gut ist, gibt es in diesem Fall ein Problem, das aber lösbar ist mit ⇒ **gebundene Wildcards**

- **Lösung Wildcards**: Bei einem Beispiel (siehe oracle.com-Bsp. für Wildcards) wie folgendes:

```
Collection<String> coll = new Collection<String>() {
... Implementierung von add() und Iterator<String> iterator() ... }

Collection<Object> collString = coll; /*NICHT MÖGLICH - incompatible types*/
Collection<?> collString = coll; /*MÖGLICH wegen Wildcard*/
```

  Mittels Wildcard *Collection<?>* können wir beliebige Collection erzeugen, während die Lösungen mit Object dies nicht zulässt. Das hat hier allerdings nichts damit zu tun, dass Object Obertyp (supertype) von allen anderen Klassen ist, nur hier ist zu beachten: ($Collection < Object > \neq_{Untertypbeziehung} Collection < String >$). Also können wir nicht argumentieren, dass wenn der Inhalt von Collection der allgemeinste Typ ist, so sich die Untertypbeziehung von Object zu allen anderen Klassen auch auf die Collection übertragen (siehe nicht vorhandene implizite Untertypbeziehung → siehe Frage 1).

- **gebundene Wildcards** = Fragezeichen mit einer Schranke nach oben (also nur Obertypen erlaubt) oder nach unten (nur Untertypen erlaubt), bspw.
  - eine Methode mit Wildcards mit **unteren** (**extends, nur gelesen**(from)/**Kovarianz**) und **oberen** (**super, nur geschrieben**(to)/**Kontravarianz**) Schranke:

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

  - Bspw.: Mit ($Square < Polygon$) wäre ein Aufruf **möglich** mit

```
List<Square> listSquares = new List<>();
List<Polygon> listPolygon = new List<>();
addSquares(listSquares, listPolygon);
```

    , weil von listSquares (Square Untertyp von Square, reflexiv) gelesen (Kovarianz) und auf listPolygon (Polygon Obertyp von Square, hier wäre auch ein Typ List<Square> ebenfalls möglich da Square auch Obertyp von Square ist) geschrieben wird (Kontravarianz).
    Jedoch wäre es **nicht möglich** mit ($Point < Square$) und:

```
List<Point> listPoints = new List<>();
addSquares(listPolygon, listPoints)
```

    , da für den ersten und zweiten Parameter Kovarianz und Kontravarianz verletzt würde.

- **Wozu** (nochmal explizit): Wildcards werden in formalen Parametern von Methoden, aber nicht in der Angabe der Typparameter einer Klassen verwendet. Also wäre Folgendes **nicht erlaubt**:

```
public class Auto<? extends Fahrzeug> {...} /*FALSCH*/
public class Auto<T extends Comparable<? extends A> > {...} /*RICHTIG*/
```

  Dabei kann man bei formalen Parametern ein bloßes Fragezeichen *public void show(Typ<?> param){...}* oder auch spezieller eine gebundene Wildcard mit Schranke (hier nach unten) angegeben werden *public void show(Typ<? extends Obertyp> param){...}*.
  Wozu: Um einerseits statische Typsicherheit an Programmstellen zu erzeugen und andererseits, um Untertypbeziehung gepaart mit Codewiederverwendung herzustellen. Bspw.:

```
public class Point<T extends Square, P extends Polygon, A> {
...
public void function(Point<?, ?, ?> param) {...}
...}
```

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?

- Arten von Generizität (nicht in Hinsicht auf die Übersetzung):
  - [1] **einfache Generitzität**
    - normale/einfache Generizität:

      ```
      public class List<A> implements Collection<A> {...}
      ```

  - [2] **gebundene Generizität**
    - Schranke (nur) mit extends (bzw. super) in der Spitzklammer:

      ```
      public class Scene<T extends Scalable> implements Iterable<T> {...}
      ```

  - [3] **F-gebundene Generizität**
    - Form der Generizität mit rekursiven Typparameter (mit rekursiven Klassennamenverwendung) nach dem formalen Modell bei Klassen

      ```
      public class Integer implements Comparable<Integer> {...}
      ```

      oder bei Methoden

      ```
      public static <A extends Comparable<A> > A max(Collection<A> xs){A w = ...; return w;}
      ```

      mit der Rekursion in der Spitzklammer
    - Besonders bei Klassen lassen sich durch die rekursive Verwendung von Klassennamen direkt Bedingungen (durch die Klasse erzeugte Signatur) der später einzusetzenden Parameter definieren.
- Arten von Generizität hstl. Übersetzung (Übersetzung generischer Klassen und Methoden in ausführbaren Code):
  - **homogene Übersetzung**: (wird in Java verwendet), dabei folgende Schritte:

    1.) Jede generische bzw. nicht-generische Klasse wird in genau eine Klasse mit JVM-Code (java virtual machine = ermöglicht es einem Computer ein Java-Programm laufen zu lassen) übersetzt

    2a.) Jeder gebundene Typparameter wird in die erste Schranke übersetzt bzw. in Object, wenn ungebundener Typparameter (es wird also, weil keine konkrete Typinformation vorhanden ist, der allgemeinste Typ angenommen → Object):

```
public void function(A param) {...} /*nicht übersetzt*/
public void function(Object param) {...} /*übersetzt*/
```

2b.) Wenn Methode ein Objekt des Typparameters als Param. nimmt/zurückgibt, wird Typ des Objekts vor/nach Methodenaufruf dynamisch in den Typ (der den Typparameter ersetzt) umgewandelt.

– **heterogene Übersetzung**:

∗ Für jede Verwendung einer generischen Klasse oder Methode mit anderen Typparameter wird eigener übersetzter Code erzeugt.

∗ Nachteil: Große Anzahl an übersetzter Klassen und Methoden

∗ Vorteil: Für alle Typen eigener Code → daher keine Laufzeiteffizienzeinbußen für Ersatz von Typparameter von elementaren Typen (Wie int, char, boolean), keine Typumwandlung zur Laufzeit und damit zsmhängende Überprüfungen, für jede übersetze Klasse eigene Optimierungen anwendbar (hängen von Typen ab)

- In **Java wird homogene Übersetzung** verwendet. **Homogene Übersetzung ist dabei flexibler**, da alles speziell übersetzt wird und es nicht wie bei der heterogenen Übersetzung übersetze Templates gibt, dafür ist aber die Laufzeiteffizienz schlechter als bei der heterogenen Übersetzung.

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

- Wie in altem Java-Bibliotheken kann man Generizität durch Untertypbeziehung zusammen mit „Object“ simulieren. (147) Der Ergebnistyp müsste dann allerdings immer mittels Typumwandlungen in den gewünschten Typ gecastet werden.
  Somit stellt Object den Typparameter dar, da ja jeder Elementar- und Referenztyp prinzipiell Untertyp von Object ist. Bspw.:

```
public <T,Q> A fct(T param1, Q param2) {return null;} /*generisch*/
public Object fct(Object param1, Object param2 ) {return null;} /*simulierte Generizität*/
```

- **Verlust**: Dabei verzichtet man aber auf **bessere Lesbarkeit** (nur eine generische Blueprint-Klasse) und **höhere Typsicherheit**, da durch Typparameter und besonders durch gebundene Typparameter deklarierte Typen allgemein in Klassen oder speziell bei formalen Parametern in Methoden eingegrenzt werden und falls fehlerhaft, diese Fehler der Compiler bereits statisch anzeigen kann.

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

- (siehe Frage 6)
- **heterogene Übersetzung in Kürze**: Halte für verschiedene Übersetzungen der generischen Klasse in eine nicht-generische Klassen verschiedene „Schablonen“ vor.
- **homogene Übersetzung in Kürze**: Zwei Schritte: Erzeuge zunächst JVM-Code und übersetze dann (grob, siehe für Details Frage 6) alles Generische in die unterste Grenze (gebunden) oder (falls nicht gebunden/keine Schranke) in Object. Abfolge:

  (1) Spitze Klammern weglassen

  (2) jedes andere Vorkommen eines Typparameters durch Object bzw. durch eine möglicherweise vorhandene erste Schranke ersetzen

(3) Argumente und Ergebnisse werden in die deklarierten Typen umgewandelt (falls Ergebnistypen bzw. formalen Parameter Typparameter sind und durch Typen ersetzt wurden)

Zusatz: Nachdem eine Klasse übersetzt wurde, bspw. *Iterator<A>* zu *Iterator* oder *List<A>* zu *List*, so nennt man diese übersetzten Klassen **Raw-Types** oder man spricht von **Type-Erasures** (=Hinweis, dass Typinformation weggelassen wurde). (In gewisser Weise sind alle Klassen und Interfaces, die Container darstellen, als Raw-Types zu betrachten.) Dagegen meint man mit **parametrisierten Typ einer generischen Klasse**, dass man für den Typparameter einen konkreten Referenztyp angibt, bspw. für *Iterator<A>* → *Iterator<Integer>*.
Der **Sinn für Raw-Types** (also ein Referenztyp durch Nutzung des Namens eines generischen Typs unter Weglassen des Typparameters) ist einerseits „backwards-compatibility“ (mit Java 1.4, hier homogenes Übersetzen). **Andererseits viel wichtiger**, man erhält durch Raw-Types in Fällen wie List<Object> und List<String> einen **Obertypen für formale Parameter in Methoden**, vgl. folgendes Bsp.:

```
public void function(List<Object> param) {...} /*Parameter ist kein Raw-Type*/
public void functionRawType(List param) {...} /*Parameter ist Raw-Type „List“*/
List<String> liste = new List<>();
function(liste); /*FEHLER*/
functionRawType(liste) /*KLAPPT*/
```

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

- Laufzeitfehler, die entstehen könnten, falls keine statische Typsicherheit (durch den Compiler) garantiert wird/also alles, was statisch als Fehler durch den Compiler in Zusammenhang mit Generizität angezeigt wird:
  - richtiges Auflösen von Typparameter
  - Schranken (bei gebundenen Typparameter/Wildcards) auflösen und Untertypbeziehungen kontrollieren (für Typparametern in Klassen, in formalen Parametern und als Ergebnistyp in Methoden, als deklarierter Typ in Klassenvariablen und Konstanten)
  - generell, dass Untertypbeziehungen in den Typparametern eingehalten werden (anschließend an die gebundenen Typparametern)
  - dass keine Wildcards (gebunden oder ungebunden) als Typparametern von Klassen vorkommen
- Vom Compiler kann allerdings nicht kontrolliert werden: ClassCastException (deklarierter Typ ist (echter) Untertyp vom dynamischen Typ), die durch Typumwandlung durch explizite Casts bei Übersetzung von Generizität auf Argumenten entstehen können. Diese führen zu Laufzeitfehlern, ohne, dass sie der Compiler vorher anzeigen kann.

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

- (siehe Frage 4)

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

- Man wandelt den deklarierten Typen bzgl. Referenztypen bzw. bei elementaren Typen den statischen Typen um. Hier ist Vorsicht geboten, nicht dass der neu erzeugte deklarierte/statische Typ zu einem Untertyp des momentan vorhandenen dynamischen Typen wird, das würde eine ClassCastException werfen. (siehe Frage 4) Beispiel (mit Untertypbeziehung: $Point < Square < Polygon$):

```
int number = 2; /*mit statischem Typ int*/
System.out.println(((int)number).getClass()); /*geht nicht, weil elementarer Typ int nur statisch-typisiert ist und man daher mit getClass() den dyn. Typ nicht abfragen kann*/
System.out.println(((Integer)number).getClass()); /*ergibt class java.lang.Integer*/
```

```
Polygon square = new Square(); /*dynamische Typ ist spezieller als deklarierte Typ*/

System.out.println(((Square)number).getClass()); /*klappt, dyn. und dekl. Typ gleich speziell*/
System.out.println(((Point)number).getClass()); /*FALSCH, dyn. Typ allgemeiner als dekl. Typ: ClassCastException*/
```

- **Warum:** Neben dem Grund, dass **Java eine statisch-typisierte Sprache** ist (Statikbezug wichtig), gibt es einen rein praktischen Grund: Während man durch dynamisches Binden gewollt den dynamischen Typen ständig ändert, indem man Untertypen des deklarierten Typen einsetzt, muss es auch eine **Chance geben**, den **statischen Typen**, der viel Entscheidungsmacht in der Untertypbeziehung besitzt, **ändern zu können**.

12. Welche Gefahren bestehen bei Typumwandlungen?

- Wie bereits in Frage 4 und Frage 11 mit der ClassCastException beschrieben:
  Es kann bei einer Umwandlung in einen deklarierten Typen, der (wirklicher) Untertyp (d.h. hier nicht reflexive Eigenschaft wie Square ist Untertyp von sich selbst, sondern eine Klasse, die wirklich „unterhalb“/sich aus Square ableitet wie Point) eines dynamischen Typs **erst zur Laufzeit** zu einer **ClassCastException** kommen. Daraus folgt also eine gewissen **Typunsicherheit**.
- **Faustregel**: Wenn die Programmiersprache Generizität anbietet, dann soll diese und nicht dynamische Typumwandlung verwendet werden. (149)

13. Wie kann man dynamische Typabfragen und Typumwandlungen vermeiden?

    In welchen Fällen kann das schwierig sein?

- **Vermeidung dyn. Typabfr. bzw. Typumw.:** Im Allgmeinen lassen sich Typabfragen durch Untertypbeziehung in Form von Dynamischen Binden und durch Generizität vermeiden. Hier erfolgt die richtige Zuweisung mittels Typhierarchie und Typinferenz durch den Compiler.
- **Schwierigkeiten:** Manchmal ist es nicht einfach, durch dynam. Binden dynam. Typabfragen zu ersetzen, vgl. folgende Situationen:
  - Der **deklarierte Typ** von Variable x ist **zu allgemein** (z.B. Object). Konkret: Bei einem Cast eines Objekts mit deklariertem Typ Object kann man **alles** annehmen, der Sinn geht völlig verloren. Z.B.:

```
Object x = new Obect();
System.out.println((Person)x); /*Kontext Menschen*/
System.out.println((Fahrzeug)x); /*völlig anderer Kontext*/
```

- **Die Klassen des dekl. Typen und dessen Untertypen können nicht erweitert werden, weil sie bspw. final sind (auch kein Zugriff auf Source-Code).** Hier wären Wrapper-Klassen besser geeignet, um mit diesen wieder dynamisch Binden zu können.
- **Der dekl. Typ von x hat sehr viele Untertypen.** Durch dynam. Binden und nutzen einer speziellen Methode muss dann sichergestellt werden, dass **alle** anderen abgeleiteten Klassen ebenfalls diese Methode haben (auch wenn nicht unbedingt erforderlich für alle) → **Erhöhung der Wartbarkeit**.

14. Welche Arten von Typumwandlungen sind sicher? Warum?

- **Anwendung:** Wenn Generizität nicht erlaubt ist bzw. man in einer Sprache ohne Generizität programmiert, so gilt, dass man sichere Formen der Typumwandlung einsetzen soll.
- **Typumwandlung ist sicher, wenn...**

  → **„Up-Cast“**(gute Lsg): in einen **Obertyp des deklarierten Objekttyps gecastet** wird (siehe ansonsten ClassCastException)

  → **„Down-Cast“**(bessere Lsg): **vor dem Cast eine dynamische Typabfrage erfolgt** und sicherstellen, dass das Objekt einen entsprechenden dynamischen Typ hat (siehe Frage 4: ClassCastException, d.h., man bringt mit der dyn. Typabfrage in Erfahrung, welcher Kontext vorhanden ist und in welche Richtung es gehen soll. Bspw. falls dekl. Typ Fahrzeug, dann caste nach Auto etc.) **Problem**: diese Lösung impliziert else-Zweig (falls darin aber nur Exception, dann keine sichere Typumwandlung mehr)

  → **„Down-Cast“**(Lsg am besten): **Programmstück generisch schreiben, dann händisch auf mögliche Typfehler untersuchen, dann homogene Übersetzung durchführen** (gute, gründliche Lösung)

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

- **Exkurs**: **Frage: Warum verletzten kovariante Eingangsparameter das Ersetzbarkeitsprinzip** → Kernantwort: Das **Zusicherungskonzept wird verletzt**, d.h., wenn der Client speziellere Vorbedingungen hat (diese sind dann für den Server unvorhersehbar), dann kann der Server seine Nachbedingungen nicht mehr garantieren. Folgendes Beispiel mit Verhältnissen *(Tiger < Tier)* und *(Fleisch < Futter)*, die letztendlich ein „kovariante Problem“-Verhältnis beschreiben:

  Klasse Tier mit Bedingungen:

```
public abstract class Tier { ...
//Zusicherung-Vorbedingung: das hereinkommende Futter kann nur ernaehren
//Zusicherung-Nachbedingung: das Tier ist satt und lebt
public void fressen(Futter param) {
param.ernaehrtTier(); }
...}
```

Klasse Tiger mit Bedingungen:

```
public class Tiger extends Tier { ...
//Zusicherung-Vorbedingung: das hereinkommende Futter kann nur ernaehren
//Zusicherung-Nachbedingung: das Tier ist satt und lebt
public void fressen(Fleisch param) {
param.ernaehrtTier();
 param.killedTier();
/*da Fleisch spezieller als Futter ist, kann es auch weitere Methoden haben*/}
...}
```

Aufruf von außen (Klasse *HarmlosesFutter* verhält sich korrekt mit kontravariantem Eingangsparameter in fressen()-Methode und ist hier absolut gleich zur abstrakten Klasse Futter, damit diese aufrufbar ist):

```
 Tiger tiger = new Tiger ();
Fleisch fleischFutter = new Fleisch();
HarmlosesFutter harmlosesFutter = new HarmlosesFutter();
tiger.fressen(harmlosesFutter); /*Tiger lebt - Erwartung erfüllt*/
 tiger.fressen(fleischFutter); /*Tiger stirbt - Unberechenbarkeit*/
```

Das **kovariante Problem** (bzw. die **nicht mehr kontravarianten Eingangsparameter**) ist zusammengefasst jenes Problem, welches durch die „falschen Zusicherungen“ in den Vorbedingung die **„Unberechenbarkeit der Methode“** erzeugt.
Da die Typen der Eingangsparameter nun spezieller sind - sie können mehr (*killedTier()*-Methode in der Fleisch-Klasse) - wird das Zusicherungsverhältnis (Kapitel 2, Frage 10), (*Zusicherung auf Vorbedingungen im Untertyp sind schwächer oder gleich stark als in den entsprechenden Methoden im Obertyp*) vom Client nicht mehr eingehalten (Futter dient nur durch Ernährung zum Lebenserhalt mit Methode *ernaehrtTier()*), sodass der Server ebenfalls nicht mehr seine Nachbedingung (Tier ist satt und lebt (dadurch)) einhalten kann. So verliert die Methode fressen() gänzlich ihren Sinn.
Zusatz: In der Klasse Tiger gibt es nun die überladene fressen()-Methode mit zusätzlichem *killedTier()*-Methodenaufruf. Da von außen der deklarierte Typ des Aufrufes, mit dem er den Eingangsparameter aufruft, in der Klasse Tiger den dazu aktuell deklarierten Typen sucht (einmal Typ Futter, einmal Typ Fleisch), lebt der Tiger, wenn er harmloses Futter frisst (speziellster Typ ist Futter), stirbt aber bei Fleisch, weil Fleisch der speziellste Typ ist.

- **Kovariante Problem (KP)** = Man wünscht sich manchmal gerade kovariante Eingangsparameter, obwohl diese das EP verletzten. Verletzten deswegen, weil durch speziellere deklarierte Eingangsparameter alle Zusicherungen nicht mehr eingehalten werden können (weder vom Client, noch vom Server) und damit die gesamte Methode unberechenbar wird. Durch die Unberechenbarkeit kann man diese Methode nicht ersetzen (im Sinne der Verhaltensabschätzung). (151)
- **Lösung Kovariante Problem (KP)**:

Lösung 1: **schlechte Lösung** - Hierzu bieten sich **dynamische Typabfragen und Typumwandlungen** an. So könnte man im vorigen Beispiel dynamisch abfragen, welchen Typ das Futter hat und so entsprechend regieren. Dadurch wird das Verhalten der Methode berechenbarer. **Jedoch** nicht gänzlich berechenbar, da die Suche nach dem speziellen Typ zwar **theoretisch** eindeutig ist (und damit ist dann auch eigentlich die Methode berechenbar), aber **praktisch** wäre bei einem Programm mit einer sehr komplexen Untertypbeziehung es nicht mehr realisitisch abzuschätzen, ob nun der richtige speziellste Typ im Eingangsparameter genommen wird (implizite Unberechenbarkeit).
Aus menschlicher Sicht wird damit die Methode **praktisch** unberechenbar, man kann das Verhalten nur noch durch sehr viel testen (der Compiler prüft richtiges Verhalten) oder durch sehr aufwändiges, händisches prüfen des Beziehungsmusters, was die Wartung bei großen Programmen zu sehr steigert, abschätzen/einordnen.

Lösung 2: **bessere Lösung** - Methoden wie fressen() aus dem Obertyp gänzlich entfernen. Somit muss zwar an jeder Stelle diese Methode in den entsprechenden Stellen der Untertypen implementiert werden, allerdings lässt sich so viel besser das Verhalten der Methode **praktisch** nachvollziehen/im Blick behalten. Die Lesbarkeit ist damit so weit hergestellt, dass man auch wieder von Ersetzbarkeit (im praktischen/menschlichen Sinne) ausgehen kann.

Lösung 3: **alternative, gute Lösung** - Dazu kann man sich **binäre Methoden** als Beispiel vorstellen, die „naturgemäß“ kovariante Probleme per se darstellen. Dabei sollte man so vorgehen:

(1) Setzte die **problematische Methode** im **Obertyp** auf **final**, sodass diese in den Untertypen nicht mehr überschrieben werden kann.

(2) Leite Untertypen zwar von einem **gemeinsamen Obertypen** ab, trenne aber diese voneinander (sollen nicht direkt miteinander in Beziehung stehen, sodass sie sich nicht doch beeinflussen können).

(3) Nun kann man neben final in der Methode Berechenbarkeit (und damit Ersetzbarkeit) herstellen, indem man durch **Typumwandlung/Cast** auf Typen umstellt, die wirklich an dieser Programmstelle gewollt sind.
So ergibt sich zwar immer noch ein mögliches Problem der **unpraktischen Handhabe** (wenn Programm sehr groß ist, aber diese umpraktische Handhabe ist genauso groß, als wenn man Methoden direkt in Untertypen schreibt → vgl. (Lösung 2)), aber durch diesen **„bewussten" Cast** greift man explizit in die Programmstelle ein **(explizite Veränderung des deklarierten Typen, nicht bloß dynamische Abfrage des dynamischen Typens)** und überlässt somit die Programmsteuerung **praktisch gesehen** nicht mehr der Komplexität (und damit dem Zufall = (implizite) Unberechenbarkeit).

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

- **Überschreiben** = wenn eine Klasse von einer anderen Klasse erbt, so kann sie gleichlautende Methoden aus dieser überschreiben. Dabei muss (wie gesagt) der Name der Methode, die Anzahl der Parameter, der Ergebnistyp und die entsprechenden formalen Parameter (also deren deklarierte Typen) alle gleich sein. Dann kann man das **Verhalten der Methode überschreiben**.
- **Überladen** = Wenn eine Klasse von einer anderen Klasse erbt, so kann eine gleichlautende Methode angelegt werden, wobei diese nicht überschrieben, sondern überladen wird.
  Dabei stimmt immer noch der Name der Methode und der Rückgabetyp mit der geerbten Methode überein.
  Allerdings muss man jetzt die Anzahl der Parameter und/oder die deklarierten Typen der formalen Parameter ändern. Falls die Parameteranzahl gleich ist, muss es wie gesagt einen Typunterschied an (mind.) einer Parameterposition geben, wobei die sich unterscheidenden Typen nicht in einer Untertypbeziehung stehen dürfen.
  Bei einem Aufruf von außen entscheidet der Compiler einzig durch das matchen der deklarierten Typen der formalen Parameter (neben dem Namen der Methode natürlich), welche gleichlautende Methode auszuwählen ist. Bei der Auswahl ist also **deklarierte Typ** eines formalen Parameters entscheidend.

  Faustregel: „Man soll Überladen nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden." (157) (= sicherheitshalber den speziellsten Typ annehmen)

  Hinweis: Überladen hat nichts Direktes mit Vererbung zu tun, sondern kann auch in einer Klasse, die von nichts erbt auftreten, indem man einfach einen Methodennamen öfters mit untersch. Parametern/untersch. Parametertypen nutzt.
- **Multimethode** = Genau das gleiche Konzept wie beim Überladen mit dem bedeutenden Unterschied: Bei der Auswahl ist also **der dynamische Typ** des Aufrufers (x) und des formalen Parameters (y) entscheidend (bspw. *x.equals(y)*). Es wird bei einem Methodenaufruf mehrfach dynamisch gebunden. **Allerdings kann man dieses Konzept in Java nur simulieren.**

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

- **Multimethoden** nutzen **mehrfaches dynamisches Binden**. „Mehrfach" bezieht sich hierbei darauf, dass einerseits der dynamische Typ des Aufrufers (x) untersucht und genutzt wird, andererseits alle dynamischen Typen (Achtung: in Java wird immer der deklarierte Parametertyp verwendet) der Parameter (y, z) relevant sind und für die Methodenauswahl entscheidend sind. (siehe Beispiel Aufruf *x.friss(y,z);*)

(Bei Java wird normalerweise nur **einfach dynamisch gebunden**, nämlich nachdem durch den dynamischen Aufrufertyp die Klasse und danach durch den deklarierten Typ des Parameters die richtige Methode aus den überladenen Methoden ausgewählt wurde. Dann wird zur weiteren Verwendung der speziellste (dynamische) Typ für die Verwendung des Parameters (im Methodenrumpf) benutzt. Es gibt also in Java **nicht reine Multimethoden**, sondern diese werden **immer durch mehrmales einfaches dynamisches Binden hintereinander *simuliert***.)

- **Simulation**: Indem man entsprechende Schnittstellen für die Aufrufertypen und die Parametertypen zur Verfügung stellt. Dadurch lässt sich Multimethoden simulieren, vlg.:

(Aufrufer) Obertyp: Tier, Untertypen: Rind, Tiger (Methoden überschrieben)

(Typparameter) Obertyp: Futter, Untertypen: Gras, Fleisch (Methoden überschrieben)

(Aufruf) tier.friss(futter)

→ (1. dynam. Binden) Unterscheidung zwischen Objekten Rind und Tiger und dementsprechend Methodenauswahl in einer der beiden Klassen

→ (2. dynam. Binden) Auswahl aus den Untertypen von Futter in der Methode des bereits ausgewählten Aufrufers=Tier

- **Probleme:** Einerseits stellt das händisch erzeugte mehrfache Binden (durch Hintereinanderreihung von einfachem Binden) viel Schreibaufwand und damit viel Wartungsaufwand da. Auch wird die Objektkopplung stärker, da die Objekte untereinander stärker abhängen und aufeinander angwiesen sind. Das schadet letztendlich dem Ersetzbarkeitsprinzip.
  Andererseits kann durch ein (stark erhöhtes) Mehrfachbinden (händisch hergestellt) die Lesbarkeit/Übersichtlichkeit beeinträchtigt werden, wodurch es zusätzlich zu Fehlern (falsche Erwartungen an Programmfunktionalität) kommen kann.

18. Was ist das Visitor-Entwurfsmuster?

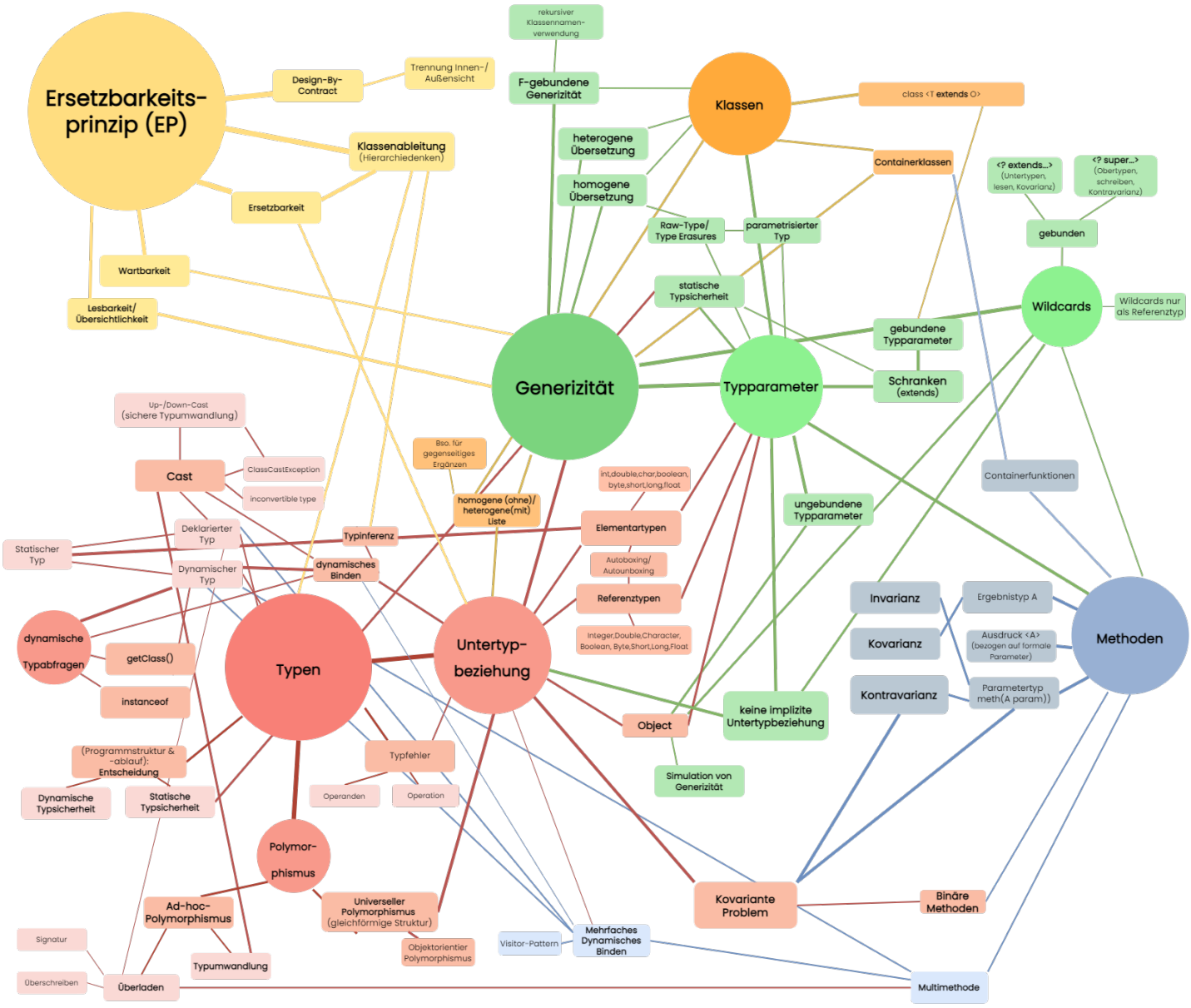
- Das **Visitor-Entwurfsmuster (VEM)** beinhaltet **mehrfaches dynamisches Binden** und ist ein klassischess **Entwurfsmuster (Design-Pattern)**. Entsprechend dem Beispiel aus Frage (15/)17 heißen Futter ***Visitors*** und die darin enthaltenen Methoden ***Visitormethoden***, Klassen wie Tiere sind dabei die ***Elementklassen***.
  Visitor- und Elementklassen sind gegeneinander austauschbar.
  Nachteil: Die Anzahl der zu implementierenden Methoden wird schnell sehr groß. (z.B. m Tierarten, n Futterarten = $m * n$ Visitormethoden notwendig)

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

- **Problem**: Es kann problematisch werden, wenn der deklarierte Typ des formalen Parameters durch **Cast** bei „zu vielen“ überladenen Methoden ermittelt werden muss. Dies ist für die Programmwartung hinderlich, da aus der **Wartungssicht** viel von konkreten Implementierung bekannt sein muss (man entscheidet sich ja bewusst für eine bestimmte überladene Methode).
  Das heißt im Sinne der **Signatur**, dass prinzipiell nach außen ein Methodennamen überhaupt nicht eindeutig ist, und damit der Aufrufer viel über Details der Klasse wissen muss (widerspricht dem **Data-Hiding**).
- **Kaum Problem**: Kaum problematisch (anknüpftend an oberer Problemdarstellung) ist es, wenn einerseits nur **sehr wenig überladene Methoden** vorhanden sind und diese sich **durch ihre deklarierten**

Typen stark unterscheiden (bspw. nimmt die eine Methode String als deklarierten formalen Parametertyp, während die andere int verwendet).

Kapitel 3 | Themenübersicht



Kapitel 4 (Kreuz und quer)

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

   - Ausnahmen sind (in Java) einfache Objekte vom Typ **Throwable** bzw. von dessen beiden Untertypen:
     - **Error**: für schwerwiegende Ausnahmen des Java-Laufzeitsystems, schwerwiegende Fehler, Auftreten führt meistens zum Programmabbruch, Bsp. für Untertypen von Error sind **StackOverflowError, OutOfMemoryError**
     - **Exception**: untergliedert in zwei Bereiche (Unterklassen)
       - **überprüfte Ausnahmen**: meist selbst definiert, hier dürfen Methoden nur Objekte von Error und RuntimeException oder im Kopf der Methode ausdrücklich angegebene Ausnahmen werfen, z.B.: *void foo() throws Help{...}*
       - **nichtüberprüfte Ausnahmen**: vorgegeben, sind Objekte von **RuntimeException**, weitere Untertypen: **NullPointerException, ClassCastException etc.**
   - **explizite Ausnahmen**: mit Hilfe von *throw*-Anweisungen, Bsp.:

```
public class Help extends Exception {
   public Help (String errorMessage)
     super(errorMessage);
}
  ... (in irgendeiner Methode einer Klasse)
  public void fct() throws Help {
    if(helpNeede()) throw new Help("Fehlernachricht");
  }
```

   - **Abfangen von Ausnahmen** die geworfen wurden durch **try-catch-Blöcke**

```
try {...}
catch(Help e) {...}
catch (Exception e) {...}
finally {...} /*wenn definiert, tritt dieser immer auf, auch wenn try klappt*/
```

     Jeder catch-Block enthält genau einen formalen Parameter und wird ausgelöst, falls Objekt, das geworfen wird, vom Typ des Parametertyps ist (catch fängt Ausnahme ab).

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

   - **Ausnahmen und das EP:** Die im Obertyp definierte throw-Klausel (Ausnahmetyp) muss auch die entsprechende Methode im Untertyp liefern oder weniger (Ersetzbarkeitsprinzip), sie darf also Ausnahmen der Obertyp-Methode auslassen und damit weniger haben. (168)
   - **Nachbedingung:** In diesen zugesichertes Werfen einer Ausnahme bei einer bestimmten Stelle muss der Untertyp ebenfalls erfüllen.

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

   - **Wozu:** In folgenden Fällen:

  - **Unvorhergesehene Programmabbrüche:** Programmabbruch bei einem Fehler mit Stack-Trace (mit genauerer Information)
  - **Kontrolliertes Wiederaufsetzen:** Im praktischen Einsatz (also nicht in der Programmentwicklung) soll das Programm möglichst weiterlaufen. Hierzu definierter Punkt, an dem das Programm weiterlaufen soll.
  - **Ausstieg aus Spachkonstrukten (gezielter Einsatz):** Einsatz nicht nur bei Fehlern, sondern auch brauchbar für Aussteigen aus Kontrollstrukturen, Methoden etc. (hier erwartetes Auftreten)
  - **Rückgabe alternativer Ergebnistypen (gezielter Einsatz):** Durch Ausnahmen ist es möglich, andere Typen als den Ergebnistypen an den Aufrufer zurückzugeben.
- **Wozu nicht:** Dazu als Kontrast folgende Faustregeln sinnhaft wiedergegeben:
  (1) Ausnahmen nur sparsam und wenn in echten Ausnahmesituation verwenden → Wartbarkeit und Lesbarkeit
  (2) Ausnahmen nur verwenden, wenn Programmlogik vereinfacht wird (bspw. wenn viele bedingte Anweisungen durch eine catch-Klausel zsmgefasst werden).
  - **explizit nicht:** Wenn catch-Blöcke (z.B. *catch(T1 x) {...}* wie switch-Anweisungen Typen abfangen, denn hierbei handelt es sich implizit um dynamische Typafragen, die man zugunsten der besseren Wartbarkeit durch dynamisches Binden ersetzen sollte.

4. Durch welche Sprachkonzepte unterstützt Java die nebenläufige Programmierung?

   Wozu dienen diese Sprachkonzepte?

- **new Thread(p).start():** Jeder Aufruf von **new Thread(p)** erzeugt einen neuen Thread, der durch den Aufruf von **start()** zu laufen beginnt, wobei der **Parameter p** ein **Objekt von Runnable** ist; start() bewirkt die Ausführung von **p.run()** im neuen Thread (run() muss durch *implements Runnable* in der Klasse überschrieben sein).

```
public class Produzent implements Runnable {
...
  public void run() {...}
} ...
Druckertreiber t = new Druckertreiber(...);
for (int i = 0; i < 10; i++) {
   Produzent p = new Produzent(t);
    new Thread(p).start();
}
```

- **Locking:** Um Programmteile zu synchronisieren. Ein **Lock** wird immer auf das Objekt gesetzt, auf das sich das Argument im *synchronized(Argument)* bezieht, bei Methoden ist das das Aufrufer-Objekt, bei this in einem Block das Objekt selbst.
  - **synchronized Methoden:** Angewendet auf alle Methoden, die auf Objekt- oder Klassenvariablen (lesend/schreibend) zugreifen, um eben diese Methoden atomar auszuführen. Zugriffe werden also gereiht und Inkonsistenzen so verhindert. (175)

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

Dabei gilt, dass *synchronised Methoden* nur kurz laufen sollen (ansonsten erhöhen sie die Dauer der Blockade zu arg).

- **synchronized Blöcke:**

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

- **Vordefinierte Methoden (wait(), notify(), notifyAll()):**
  - **wait():** Ist ein in Object vordefinierte Methode und blockiert den aktuellen Thread solange, bis er wieder aufgeweckt wird.
  - **notifyAll():** Weckt alle Threads in der Warteliste des aktuellen Objekts wieder auf.
  - **notify():** Ähnlich wie notifyAll(), jedoch wird hier nur ein Thread aus der Warteliste des Objekts aufgeweckt. (**Achtung**: hier könnte ein Thread aufgeweckt werden, der aufgrund seiner Situation sich sofort wieder schlafen legen muss und damit alles blockiert → notify()-Methode besser meiden).
  - wait(), notify() und notifyAll() müssen alle in einem synchronized Methode/Block stehen. Bei wait() wird aktiver Thread in Warteliste des Objekts (in Argument von synchronized bestimmt) gehängt, welcher den Lock hält.
- **Monitor-Konzept:** Synchronisation wird dabei weder zur Objektschnittstelle (also nicht bspw. in Interfaces definierte Zusicherungen), noch zur Untertypbeziehung (außer Zusicherungen) berücksichtigt. **Synchronisation in Untertypbeziehungen durch Einbeziehen von Client-kontrollierten History-Constraints**, d.h., dass Methoden in Untertypen durch *wait(), notify(), notifyAll()* nicht stärker synchronisiert sein dürfen als entsprechende Methoden im Obertyp (bspw. wenn Methode aus Obertyp in einer best. Situation *wait()* aufruft, so darf das auch die Methode aus dem Untertyp). (184/185)
- Konzept **Streams:** (zsm mit Lambda-Ausdrücken) unterstützt es einen *funktionalen, applikativen Programmierstil*. Im Wesentlichen entspricht es einem **internen Iterator über der gegebenen Funktion auf alle Elemente einer Datenstruktur**. Z.B.:
  Sequentielle Ausführung

```
HashSet<String> nums = ...; // "1", "2", ...
int sum = nums.stream()
          .mapToInt(Integer::parseInt)
          .reduce(0, (i, j) -> i + j);
```

Der Stream iteriert über Zeichenkette.
Parallele Ausführung

```
HashSet<String> nums = ...; // "1", "2", ...
int sum = nums.parallelStream()
        .mapToInt(Integer::parseInt)
        .reduce(0, (i, j) -> i + j);
```

Operationen werden auf dem Stream als Tasks über einem Thread-Pool parallel abgearbeitet (Vorsicht: dabei dürfen Variable nicht auf gemeinsame Variable zugreifen).

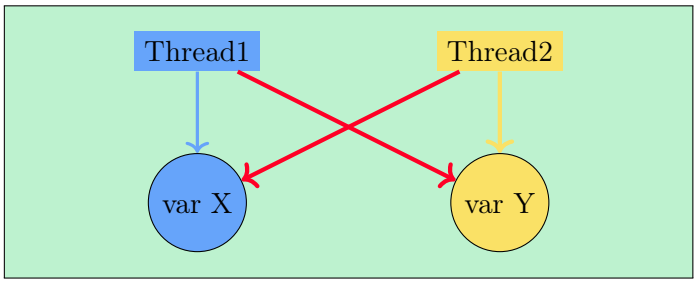
- **Thread-sichere Datenstrukturen:** Java-Paket *java.util.concurrent* stellt eine Reihe synchronisierter Varianten von Datenstrukturen zur Verfügung (z.B. *ConcurrentHashMap*).

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

- **Wozu:** Um ein Programm möglichst günstig auszulasten.
- Die Granularität der Synchronisation soll so gewählt sein, dass möglichst **kleine, logische konsistente Blöcke → Methoden** (am besten) **entstehen** (so einfach und lokal wie möglich). Dabei gilt es **Teilaufgaben** zu finden, die unabhängig voneinander sind, also bspw. nicht auf gemeinsame Variablen zugreifen (besonders möglichst wenig Schreibzugriffe).
  Vorsicht hier vor **Historie-Constraints** im Programm, also Vorgaben bzgl. der Abarbeitungsreihenfolge, falls es nicht vermeidbar ist, dass Daten voneinander (indirekt → beeinflussen ein „neutrales" Objekt ohne direkten Zugriff bspw.) abhängen.
- **Synchronisierte Blöcke** sollen nur **kurz laufen**, und nur **wichtige Methoden** sollen durch ***synchronized*** **gelockt werden**.

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

- Es gibt verschieden geartete Problemtypen, hier eine Auflistung:
  - **Ersetzbarkeitsprinzip-Verletzungen:** Das Monitor-Konzept deutet bereits darauf hin, dass Ersetzbarkeit durch (zu viel) Nebenläufigkeit verletzt wird. Aufteilung von Synchronisationseinheiten wird dabei nicht zu einer Objektschnittstelle gezählt, jedoch sind eben solche Schnittstellen grundlegend für die Modularisierung von Programmen unter dem EP.
  - **Faktorisierung von Software:** Der objekt-orientierte Ansatz folgt anderen Gesichtspunkten, als die Zerlegung von Aufgaben in nebenläufige Teilaufgaben. Bspw. erniedrigen nebenläufige Gesichtspunkte den Klassenzusammenhalt und führen somit zu höherem Wartungsaufwand.
  - **Deadlocks:** Zyklische Abhängigkeiten zwischen zwei oder mehreren Threads. Z.B.:



Thread1 hält Lock auf *varX*, *Thread2* hält Lock auf *varY* → *Thread1* will Lock auf *varY* und **wartet**, *Thread1* will Lock auf *varY* *Thread2* und **wartet** ebenfalls ⇒ **Deadlock**.

Liveness-Properties: Ähnlich wie Deadlocks sind das Phänomene wie **Livelocks** und **Starvation**.(183)

- **Komplexitätserhöhung:** Beim Zusammenstellen von unabhängigen Programmteilen und beim Aufteilen in Abarbeitungsphasen kann es schnell passieren, dass die angstrebte nebenläufige Lösung sehr komplex wird (→ Beeinträchtigung der Wartung) und es womöglich zu einem **schlechten Overhead** (sehr viele Verwaltungsdaten etc.) kommt. (182)

- **Fazit:** Wird in der Praxis nicht häufig verwendet, besser auf Java-Pakete wie *java.util.concurrent*, *java.util.concurrent.atomic* und *java.util.concurrent.locks* zurückgreifen (sind gut durchdachte und effiziente Implementierungen).
  - ***Future* im Paket *java.util.concurrent*:** Ähnlich wie *Future* in funktionalen Programmen handelt es sich hierbei um eine Variable, in der ein zu berechnendes Ergebnis abgelegt wird, wobei der lesende Thread, der darauf zugreifen will, solange blockiert wird, bis das Ergebnis vorhanden ist.
  - **Interface *Executor* im Paket *java.util.concurrent*:** verteilt Aufgaben auf verfügbare Threads aufgeteilt.

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

- **Notation:** Annotationen beginnen mit einem „@“ und werden vom Compiler geprüft (bspw. bei *@Override* für überschriebene Methoden). Zusätzlich können Annotationen **Argumente** enthalten.
- **Wozu:** Annotation sind zusätzliche syntaktische Elemente/Erweiterungen in Programmen und geben wie eine Art Modifier **Metainformation** über das Programm wieder, bestes Beispiel ist die *@Override* Annotation, die zwingend von Obertypen abgeleiteten und zu überschreibenden Methoden in Untertypen kennzeichnet.

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

- Annotationen werden statisch definiert (wenn selbst, dann über Interface-artige Klassen). Diese leben über die gesamte Zeit hinweg und werden vom Compiler geprüft (bspw. können Annotationen, die nur bei Methoden stehen dürfen, auch nur dort erwartet).
  Will man dagegen Annotationen auslesen, so wird erst zur Laufzeit ein Interface generiert, dass allerdings ausgelesen werden kann (es werden also die Werte, die in den entsprechenden Klassen angegeben wurden, ausgelesen).

9. Wie kann man eigene Annotationen deklarieren?

   Welche Gemeinsamkeiten und Unterschiede zu Interfaces bestehen?

- **Eigene Annotationen definieren**: *New → Java Class → Name:... mit Annotaion*
- **Anwendung:**

```
package com.jetbrains;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE}) // kann überall stehen
public @interface BugFix {
        String who(); // author of bug fix
        String date(); // when was bug fixed
        int level(); // importance level 1-5
        String bug(); // description of bug
        String fix(); // description of fix
}
```

Statt *@Target({ElementType.TYPE})* kann auch @Target({ElementType.METHOD}) stehen, dann dürfte die Annotation nicht überall, sondern nur bei Methoden stehen. Durch die Definition *@Target({ElementType.METHOD})* ist die selbst definierte Annotaion überall bei Methoden anwendbar, bspw.:

```
@BugFix(who="P", date="11.02.2022", level=2, bug="Problembeschr....", fix="Lösung...")
public class Tiger extends Tier {

        public void fressen(Fleisch param) {...} ...}
```

- **Gemeinsamkeiten:** Es lassen sich wie bei Interfaces ebenfalls kein Rumpf implementieren und Modifier wie public können weggelassen werden.
- **Unterschiede:** Im Gegensatz zu Interfaces kann man direkt auf die zur Laufzeit erstellten Interface-Methoden zugreifen und diese praktisch auslesen und nutzen.

10. Wie kann man zur Laufzeit auf Annotationen zugreifen?

- **Reflexion/Reflection/Introspektion:** Ist die Technik (ist eine Variante der *Metaprogrammierung*, kann aber nicht die Programmstruktur wie andere Varianten verändern (192)), die zur Laufzeit durch *@Retention(RUNTIME)* die Zugriffsmöglichkeit festelegt und ein entsprechendes Interface zur Laufzeit generiert:

```
public interface BugFix extends Annotation {
        String who();
        String date();
        int level();
        String bug();
        String fix();
}
```

Und so kann man auf die Annotation **zugreifen**:

```
String s = "";
  BugFix a = Tiger.class.getAnnotation(BugFix.class);
if (a != null) { // null if no such Annotation
        s += a.who()+"fixed a level "+a.level()+"bug";
}
```

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

- **Was:** Aspektorientierte Programmierung **kapselt Verhalten**, das **mehrere Klassen betrifft**, in Aspekte. Ein Aspekt **beschreibt Funktionalität** (an einer Stelle), als auch alle Stellen im Programm, an denen dieser Aspekt angewendet wird. (Hintergrund: Aufteilung des Programms in Modularisierungseinheiten, wobei manche Aspekt mehrere Bereiche im Programm entsprechen (*Cross-Cutting-Concerns/Quer-schnittsfunktionalität*)).
  Dabei fasst ein Aspekt Deklaration, Pointcuts und Advices zusammen und sieht wie eine Klasse aus (*aspect* statt *class*).
- **Einsatz:** Sinvoll ist der Einsatz, wenn ein Programm viele *Cross-Cutting-Concerns* aufweist (und diese durch Refaktorisierung nicht weiter eingedämmt werden können), um somit die **Verständlichkeit des Programms zu erhöhen**.

12. Was bedeutet Separation-of-Concerns?

- ***Separation-of-Concerns*:** Entspricht der Faktorisierung eines Programms in verscheidene Modularisierungseinheiten (*Concerns*). Hierbei gibt es die Schwerpunkte (*Core-Concerns*), aber auch Funktionalitäten, die sich quer über das Programm ziehen (*Cross-Cutting-Concerns*) (194)

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

- **Core-Concerns:** Sind die Kernfunktionalitäten eines Programms und sollten leicht zu kapseln (klar in Modularisierungseinheiten abtrennbar) sein. Z.B. bei einer *Banksoftware* wäre das Funktionalität wie *Konten, Geldtransfer, Wertpapiere etc.*.
- **Cross-Cutting-Concerns:** Sind Funktionalitäten, die sich quer über ein Programm ziehen können und als Aspekt (eines Programms) zusammengefasst gesehen werden kann. Z.B. *Banksoftware*, hierbei wär die Zugriffskontrolle auf die in Grundfunktionen ein sich über das Programm hinwegziehender Aspekt.

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

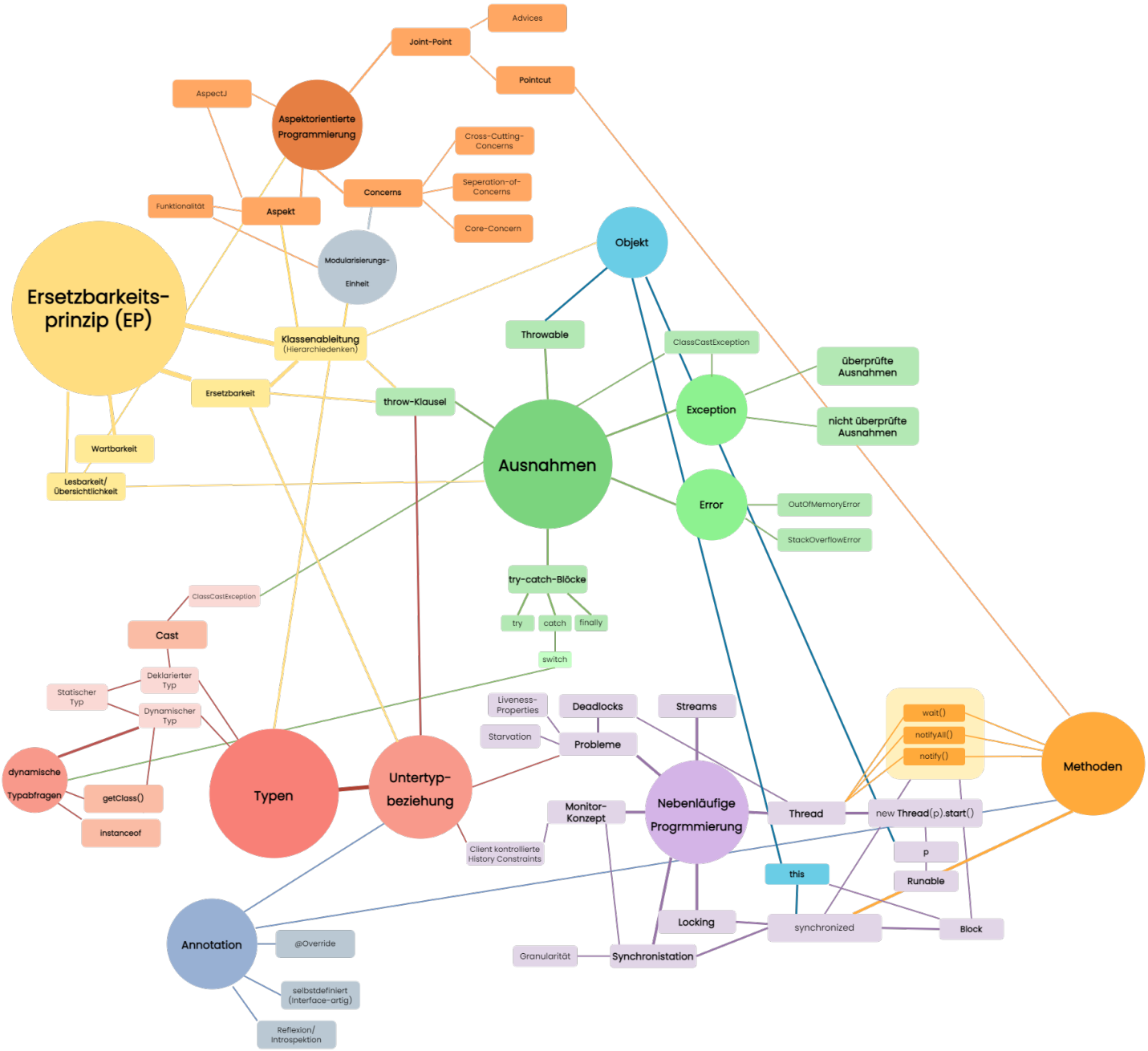
- **Joint-Point:** Ist eine identifizierbare Stelle während einer Programmausführung (z.B. Aufruf einer Methode).
- **Pointcut:** Ist ein Programmkonstrukt, das einen Joint-Point auswählt und **kontextabhängig Informationen sammelt** (z.B. Argumente eines Methodenaufrufs). (Es gibt mehrere Pointcut-Typen, bspw. *execution(MethodSignature), call(MethodSignature), execution(ConstructorSignature)*(=Ausführen eines Konstruktors), *get(FieldSignature), set(FieldSignature)...*)
- **Advices:** Ein Advice ist jener Programmcode, der vor (before()), um (around()) oder nach (after()) dem Join-Point ausgeführt wird.

- **Aspekte:** Ein Aspekt ist wie eine Klasse das **zentrale Element in AspectJ**. Er enthält alle Deklarationen, Methoden, Pointcuts und Advices.

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

- **Programmpunkte:** (Ist eine identifizierbare Stelle während der Programmausführung →) Aufruf einer Methode, Zugriff auf ein Objekt...

Kapitel 4 | Themenübersicht

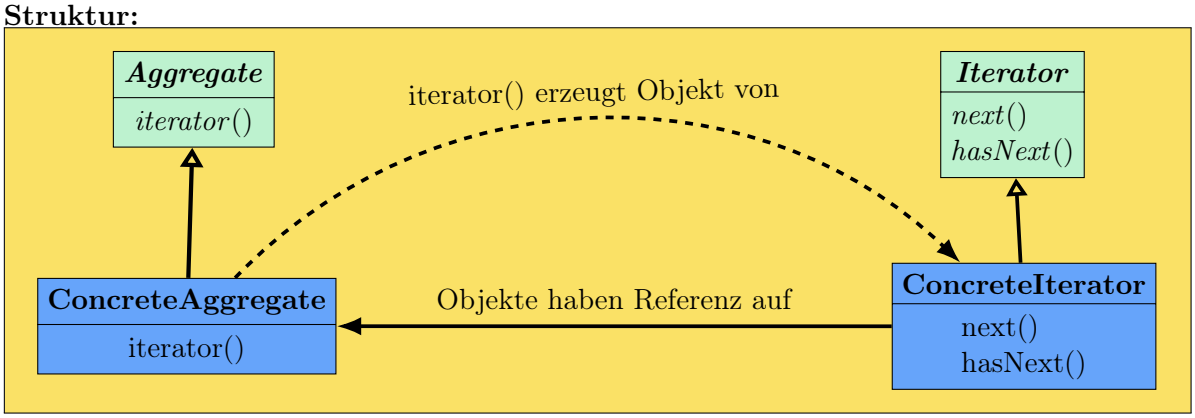


Kapitel 5 (Software-Entwurfsmuster)

1. Erklären Sie folgende Entwurfsmuster und beschreiben Sie jeweils Anwendungsgebiet, Struktur, Eigenschaften und wichtige Details der Implementierung unter Verwendung vorgegebener Namen:
   (1) Decorator
   (2) Factory-Method
   (3) Iterator
   (4) Prototype
   (5) Proxy
   (6) Singleton
   (7) Template-Method
   (8) Visitor (siehe Abschnitt 3.4.2)

(1) **Iterator-Entwurfsmuster:**

- **Name**:
  Iterator oder Cursor (207-211)
- **Problemstellung:**
  Sequentieller Zugriff auf Elemente eines *Aggregats* (=Sammlung von Elementen/*Collection*), ohne innere Darstellung zu kennen.
- **Anwendungsgebiet:**
  Wir brauchen eine **einheitliche Schnittstelle** mit **polymorpher Iteration** (=Abarbeitung versch. Aggregatstrukturen, Liste/Bäume ...).
  Wir wollen **mehrere (gleichzeitig bzw. überlappende) Abarbeitungen** der Elemente im Aggregat ermöglichen.
  Wir wollen auf den Inhalt zugreifen, ohne innere Darstellung zu kennen.
- **Struktur:**



Abstrakte Klasse/Interface: Iterator, Aggregate (in Java auch *Iterable)*

Klassen: *ConcreteAggregate* (iterator()-Methode erzeugt neues Objekt von ConcreteIterator), *ConcreteIterator* (jedes Objekt hier von braucht Referenz auf ConcreteAggregate)

- **Eigenschaft:**
  1.) Eigenschaft(/Problem): Untersch. Arten der Abarbeiten eines Aggregats (z.B. Bäume: lmr, mlr, lrm → „interpretierte“ sequentielle Reihenfolge möglich, besser „netraulere Darstellung“ mit *((linkesChild)root(rechtesChild))*)
  2.) Eigenschaft: Iteratoren vereinfachen die Schnittstelle von Aggregaten, immer derselbe Zugriff auf Aggregationen (→ polymorphe Iteration, alle haben einfach: next(), hasNext(), remove(),...)
  3.) Eigenschaft: Mehrere Abarbeitungen auf einem Aggregat möglich.
  Andere Eigenschaften:
  - *interne Iteratoren:* (kontrollieren selbst wann nächste Iteration)
  - *externe Iteratoren:* (Anwendung bestimmt next()) Iteratoren.
- **Details der Implementierung:**

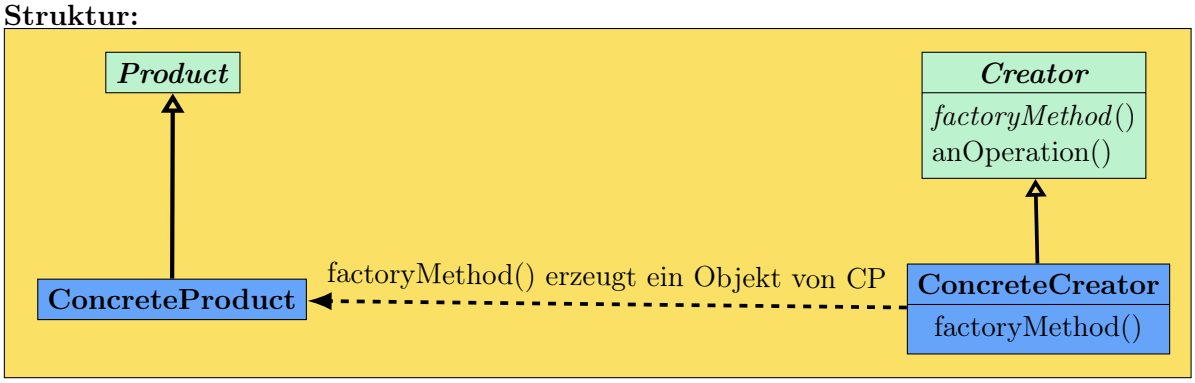
  interne Iteratoren: bspw. Streams mit *HashSet<String> nums = ...; int sum = nums.stream();*, weiteres Bsp. *forEach()* in einer *Map*, oder bspw. interne Iteratoren, da ein Argument (aus formalen Parameter) auf die Elemente des Aggregats angewendet wurde (via Parameter „mitgelieferter“ *Comparator<A>* mit Vgl.methode)

externe Iteratoren: nutzen Methoden wie *next()* und *hasNext()*.

Implementierung: (a) Algorithmus zum Durchwandern im Aggregat und Iterator nutzt dieses, (b) Algorithmus zum Durchwandern im Iterator (dann ist es leichter, mehrere untersch. Algorithmen zu haben), (c) Iteratoren in *inneren Klassen* im Aggregat

(2) **Factory-Method-Entwurfsmuster (FM):**

- **Name**:
  Factory-Method oder Virtual-Constructor (212-215) (**„Erzeugendes Entwurfsmuster“**)
- **Problemstellung:**
  Man möchte im Programm eine einzige Stelle haben, an der man etwas Neues/ein neues Objekt erstellt.
  Dabei ist ersten noch nicht bekannt, **welches Produkt** überhaupt entstehen soll, zweitens **mit welcher Produktion** dieses hergestellt werden soll. (Daher abstrakte *Creator* und abstrakte *Product* Klassen).
- **Anwendungsgebiet:**
  1.) welcher Produzent ⇒ *NewFahrzeugManager.**set**(Fahrzeugbauer c)* (→ private **Fahrzeugbauer** c)
  2.) welches Produkt ⇒ *NewFahrzeugManager.**newFahrzeug**()* (→ public **Fahrzeug** newFahrzeug(), hängt vom **Fahrzeugbauer c** im Argument von set(...) ab, welcher Fahrzeug-Typ zurückkommt)
- **Struktur:**



Abstrakte Klasse/Interface: Product, Creator (dabei ist *Product* ein allg. Obertyp aller Objekte, die von factoryMethod() erzeugt werden können)

Klassen: *ConcreteProduct* (CP), *ConcreteCreator* (CC), wobei CC durch die Methode factoryMethod() ein Objekt von CP erstellt

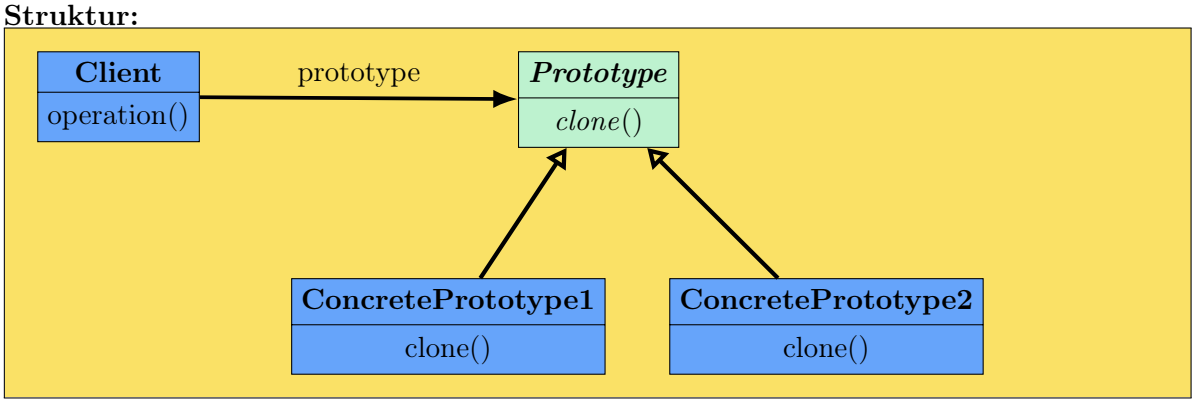
- **Eigenschaft:**
  1.) Ist Definition einer Schnittstelle für Objekterzeugung: Unterklassen entscheiden, von welcher Klasse die erzeugten Objekte sind (hier: *public void set(**Fahrzeugbauer c**) { this.c = c; }* in *NewFahrzeugManager*)
  2.) FM bieten **Anknüpfungspunkte (Hooks)** für Unterklassen. (Hooks werden in Unterklassen überschrieben (hier bspw. Fahrzeug durch Auto oder Fahrrad). Oberklassen hängen jetzt auch von Unterklassen ab (wegen typbestimmenden Argument)).
  3.) **Verknüpfen parallele Klassenhierarchien** → Creator-Hierarchie mit Product-Hierarchie (hilfreich bei Kovariantenproblem: Aufgrund der nicht mehr kontravarianten Eingangsparameter kann es dazu kommen, dass Unterklassen hereinkommen mit spezielleren Methoden → eine Methode *generiereFutter()* würde in Tier immer das richtige Futter erzeugen, sodass es nicht zu unerwünschten (speziellen) Methoden kommt (Kapitel 3, Frage 15)).

– **Details der Implementierung:**

```
/*Product*/
public abstract class Fahrzeug { ... }

/*ConcreteProduct*/
public class Auto extends Fahrzeug { ... }


/*Creator*/
public abstract class Fahrzeugbauer {
        protected abstract Fahrzeug create();          /*create()==factoryMethod()*/
}

/*ConcreteCreator*/
public class Autobauer extends Fahrzeugbauer {
        protected Fahrzeug create() { return new Auto(); } }


public class NewFahrzeugManager {
        private Fahrzeugbauer c;
        public void set(Fahrzeugbauer c) { this.c = c; }
        public Fahrzeug newFahrzeug() { return c.create(); }
}
```

(3) **Prototype-Entwurfsmuster:**

- **Name**:
  Prototype (215-218) (**Erzeugendes Entwurfsmuster**)
- **Problemstellung:**
  1.) Klassen (von deren Objekte geclonet werden soll) sind erst zur Laufzeit bekannt. (→ erst zur Laufzeit ist der Typ bekannt: **originalCar**.clone())
  2.) Eine (parallele) Hierarchie wie Factory-Method (Creator-Klassen und Product-Klassen) vermieden werden soll (hat insg. viele Klassen).
  3.) Jedes Objekt soll nur wenige untersch. Zustände haben.
- **Anwendungsgebiet:**
  1.) **Anwendung zusammengefasst:** Jede **ConcretePrototype**-Klasse (leitet sich vom Interface/abstr. Klasse ***Prototype*** ab) hat ihre eigene Implementierung der Methode *clone()*.
  Diese *clone()*-Methode macht im Wesentlichen das Entwurfsmuster aus, denn jedes Objekt kann eine shallow-Copy (flache Kopie: Wert jeder Variable in originalem/kopiertem Objekt *identisch*) bzw. deep-Copy (tiefe Kopie: Wert jeder Variable in originalem/kopiertem Objekt *gleich*) von sich selber zur Laufzeit(!) anlegen. Damit flexibler als Factory-Method, wo man schon zur Compile-Zeit mit Übergabe eines konkreten Arguments entscheiden muss, von welchem Typ das zu behandelnde Objekt sein muss.
  2.) In **hochdynamischen Systemen** kann neues Verhalten durch Objektkomposition (Objekte werden aus anderen neu zusammengesetzt) statt durch Definition neuer Klassen erzeugt werden (z.B. durch Spezifikation von Werten in Objektvariablen).
- **Struktur:**



Abstrakte Klasse/Interface: *Prototype* (möglicherweise abstrakt) mit möglicher clone()-Methode (für Eigenkopie)

Klassen: Klassen *ConcretePrototype1*, *ConcretePrototype2* überschreiben clone()-Methode aus der Oberklasse nach ihren Bedürfnissen (Zustände anders)

- **Eigenschaft:**
  1.) Verstecken die konkreten Produktklassen (erst zur Laufzeit bekannt) vor Anwender/Clients. (siehe abstrakte Klasse Auto) ⇒ Reduzieren Anzahl der Klassen, die Anwender kennen müssen.
  2.) Prototypen können zur Laufzeit dazugegeben oder weggenommen werden.

3.) Erlaube die Spezifikation neuer Objekte durch veränderbare Werte (Objektkomposition und „Weiterentwicklung“ von Objektvariablen durch kopieren und weiterleben).

4.) Prototypen vermeiden eine übertrieben große Anzahl an Unterklassen (keine parallele Klassenhierarchien).

5.) Erlauben die **dynamische Konfiguration von Programmen**.

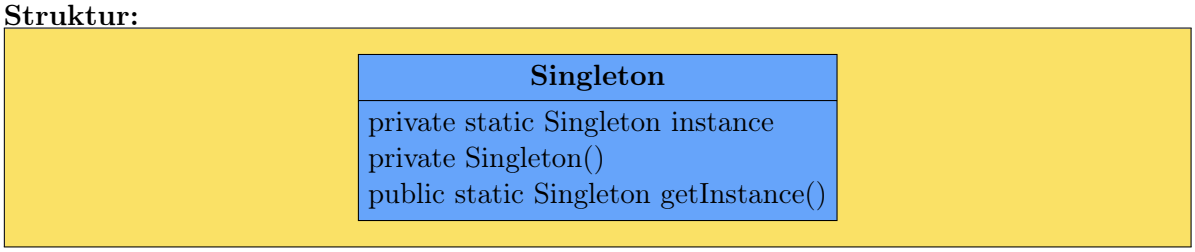
6.) In *Object* (in Java) ist die *clone()*-Methode bereits vordefiniert und kann somit von jedem Objekt abgeleitet und/order überschrieben werden.

– **Details der Implementierung:**

```
/*Prototype*/
public abstract class Auto{
        public abstract Auto clone(); }

/*ConcretePrototype1*/
public class Ferrari extends Auto (implements Cloneable) {
        private String eigenschaft;
        public Ferrari(String eigenschaft) {this.eigenschaft=eigenschaft};
        @Override
        public Auto clone() { return new Ferrari(eigenschaft); };
...}

/*ConcretePrototype2*/
public class Fiat extends Auto (implements Cloneable) {
        private String eigenschaft;
        public Fiat(String eigenschaft) {this.eigenschaft=eigenschaft;}
        @Override
        public Auto clone() { return new Fiat(eigenschaft); }; /*eigene Impl. von copy()*/
...}


/*eine Art Main-Methode*/
public class Client {
        private Auto carClone = originalCar.clone(); /*erst zur Laufzeit Automarke bekannt*/
        carClone.move();
}
```

(4) **Singleton-Entwurfsmuster:**

- **Name**:
  Singleton (218-221) (**Erzeugendes Entwurfsmuster**)
- **Problemstellung:**
  Man möchte einen Bereich im Programm haben, der **alleine** über eine gewisse Datenmenge verwaltet und verhindern, dass dieser Zustand in dublizierter Form parallel weiterentwickelt wird (= „normales“ objektorientiertes Verhalten).
- **Anwendungsgebiet:**
  1.) **Zusammengefasst Verwendung/Anwendung:** (1) einen globalen Zugriff (statische OV und getInstance() ermöglichen nur statischen Klassenaufruf von außen), (2) eine einzige Instanz (privater Konstruktor, statische Methoden), (3) daneben originale Funktionalität (Drucker, Chat,...)
  2.) Man möchte in einem System nur einen Drucker-Spooler (=nimmt Druckaufträge an und setzt sie in eine Warteschleife) haben.
  3.) Einen einzigen Chat in einem System (mit Interaktionsfunktion).
- **Struktur:**

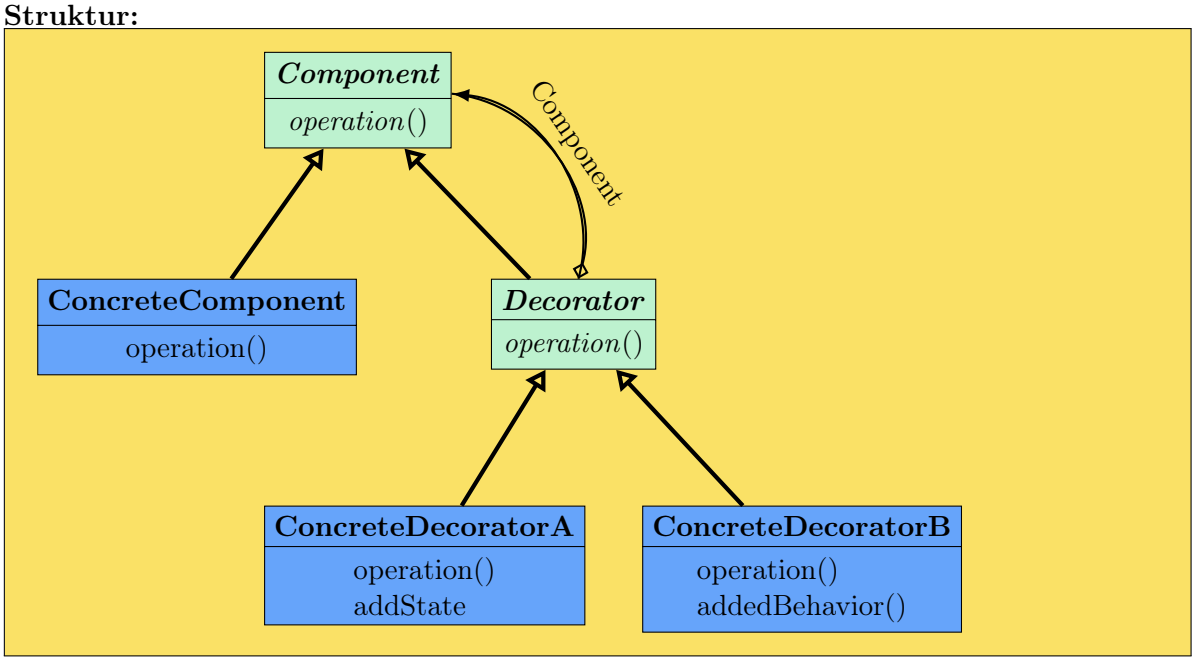


  Klasse: *Singleton* (bspw. SingletonDruckerSpooler ...) hat die Entwurfsstruktur „inkorporiert“, d.h. in der Klasse selbst ist alles angelegt: statische Objektvariable, statische getInstance()-Methode, privater Konstruktor.
- **Eigenschaft:**
  1.) **Grundeigenschaften:** Eine Klasse hat nur eine Instanz und erlaubt globalen (kontrollierten) Zugriff auf das einzige Objekt.
  2.) Konzept unterstützt Vererbung.
  3.) Erlauben auch mehrere Referenzen → können also Erzeugung mehrerer Objekte mit Referenz auf einzige Instanz ermöglichen
  4.) Unterstützt dynamisches Binden und ist damit flexibler als statische Klassen.
- **Details der Implementierung:**

```
public class Singleton {
        private static Singleton instance; /*alternativ zu private → protected*/

        private Singleton() {instance = null} /*Besonderheit: privater Contructor*/
        (protected Singleton() {instance = null}) /*alternativer Contructor*/

        public static Singleton getInstance(){
                if(instance==null) {instance = new Singleton();}
                return instance;
        }
}

public class SingletonA extends Singleton {
        protected SingletonA() { ... }

        /*getInstance() muss implementiert werden und
        retourniert Singleton, nicht SingletonA*/
        public static Singleton getInstance() {
                if(singleton==null) singleton = new SingletonA();
                return singleton;
        }
}

...

public static void main(String[] args){
        Singleton x = Singleton.getInstance(); /*statischer Aufruf*/
        Singleton y = Singleton.getInstance();
        System.out.println("Das ist Singleton x: "+ x + ", und Singleton y: "+ y)
        /*Ausgabe: „Das ist Singleton x: com.jetbrains.Singleton@64bf3bbf, und Singleton y:
com.jetbrains.Singleton@64bf3bbf“*/
}
```

(5) **Decorator-Entwurfsmuster:**

- **Name**:
  Decorator oder Wrapper (222-225) (**Strukturelles Entwurfsmuster**)
- **Problemstellung:**
  Man versucht ein Problem zu lösen, das man statisch mit Vererbung und Ableitung lösen will, bei dem es sich dann aber herausstellt, dass zur Laufzeit oder allgemein über die Zeit immer wieder neue Anforderungen bzw. Einschränkungen hinzukommen. Dann eignet sich das Decorator-Entwurfsmuster. Decorator-EM führt allerdings zu einem System mit vielen (ähnlichen) Objekten.
- **Anwendungsgebiet:**
  1.) Flexible Alternative zur Vererbung (→ dynamische Verantwortlichkeiten zu einzelnen Objekten hinzufügen). Wenn Erweiterung einer Klasse durch Vererbung unpraktisch (→ große Anzahl an Unterklassen).
  2.) **Zusammengefasst Funktion/Anwendung:** D. erlaubt ein Objekt dynamisch zu modifizieren. (Zur Laufzeit will man Funktionalität hinzufügen)
- **Struktur:**



Abstrakte Klasse/Interface: *Component*, *Decorator*, wobei jede Component immer einen Decorator hat (→ jede Pizza hat ein Topping). Decorator definiert Schnittstelle für Verantwortl., die dynamisch zu Komponenten (Pizza) hinzugefügt werden, wobei jedes Decoratorobjekt eine Referenz („Component“) zum Objekt, zu dem die Verantwortl. hinzugefügt wird, erhält.

Klassen: Klassen *ConcreteComponent* (z.B. PlainPizza), *ConcreteDecoratorA* (z.B. Mozzarella), *ConcreteDecoratorB* (z.B. Tomaten)

- **Eigenschaft:**
  1.) D. erlaubt ein Objekt dynamisch zu modifizieren. (Zur Laufzeit will man Funktionalität hinzufügen)
  2.) Anders als bei Vererbung erfolgt das **Hinzufügen von Verantwortung zur Laufzeit** (also nicht statisch zur Compilezeit) und **zu einzelnen Objekten** und nicht zu ganzen Klassen!

3.) ConcreteComponent (hier PlainPizza) muss noch nicht volle Funktionalität haben.

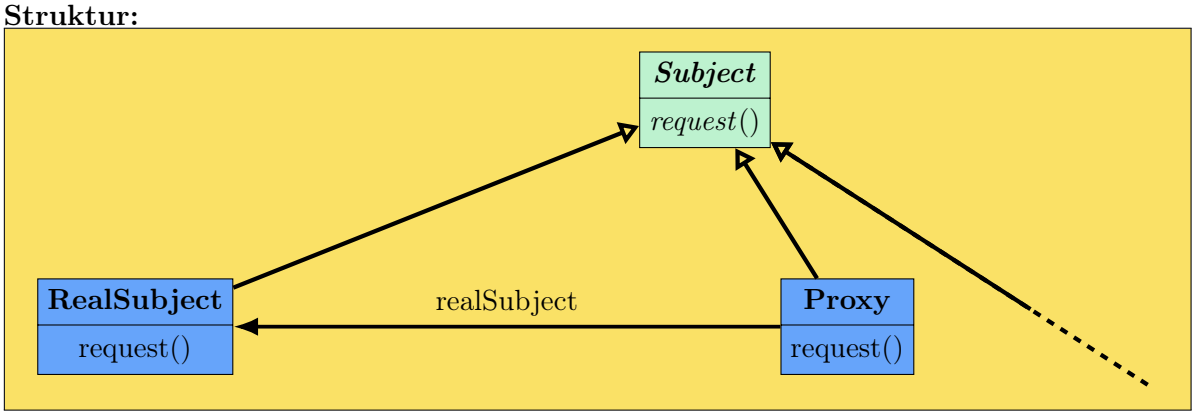
4.) Objekte von ConcreteComponent und ConcreteDecorator sind **nicht identisch**! (Nicht auf **Objektidentität verlassen**!)

– **Details der Implementierung:**

```
/*Component*/
public interface Pizza {
        void showDesciption(String text);
}


/*ConcreteComponent*/
public class PlainPizza implements Pizza {
        public void showDesciption(String text) { ... }
}


/*Decorator*/
public abstract class ToppingDecorator implements Pizza {
        protected Pizza pizza;
        public ToppingDecorator(Pizza newPizza){pizza=newPizza};
        public void showDesciption(String text) { pizza.showDesciption(text); }
}


/*ConcreteDecoratorA*/
public class Mozzarella extends ToppingDecorator {
        public Mozzarella(Pizza p) { pizza = p; }
        public void beOnPizza(String text) { ... }
        public Pizza noMozzarella() {
                Pizza p = pizza;
                pizza = null; // no longer usable
                return p;
        }
}

public static void main(String[] args){
        Pizza p = new PlainPizza(); // no Mozzarella
        Pizza p2 = new Mozzarella(new PlainPizza()); // direct with Mozzarella
        Mozzarella m = new Mozzarella(p); // add Mozzarella
        p = m; // m aware of Mozzarella, p not
        p.showDesciption("Zutaten"); // no matter if Mozzarella or not
        m.beOnPizza("zu fettig"); // works only with scroll bar
        p = m.noMozzarella(); // remove Mozzarella
}
```

(6) **Proxy-Entwurfsmuster:**

- **Name**:
  Proxy oder Surrogate (225-228) (**Strukturelles Entwurfsmuster**)
- **Problemstellung:**
  Man möchte ein teures Objekt (teuer=viele Daten) laden, allerdings möchte man einigermaßen Ressourcen sparen.
- **Anwendungsgebiet:**
  1.) **Zusammenfassung Anwendung:** Lade anstelle eines teuren Objekts zuerst einen Platzhalter und ersetze dieses dann erst bei Bedarf. (Im Wesentlichen, wenn eine „intelligente Referenz“ (also kein bloßer Zeiger) auf ein Objekt notwendig ist.)
  2.) Folgende **Situationen**:
  · **Remote-Proxies** (Platzhalterobjekte von Objekten in anderen Namensräumen)
  · **Virtual-Proxies** (erzeugen Objekte bei Bedarf)
  · **Protection-Proxies** (kontrollieren Zugriffe auf Objekte/untersch. Zugriffsrechte verwalten)
  · **Smart-References** (ersetzen einfache Zeiger, weil sie zusätzliche Aktionen beherrschen (z.B. zählen Referenzen auf das Objekt mit))
- **Struktur:**



Abstrakte Klasse/Interface: *Subject* ist eine abstrakte Klassen, gemeinsame Schnittstelle für Objekte und Proxies

Klassen: *RealSubject* definiert das eigentliche Objekt, *Proxy* ist der Objektplatzhalter, indem es eine Referenz auf RealSubject verwaltet (Proxy=Ersetzobjekt unter dem Interface Subject). Der leere Pfeil deutet weitere Proxies an, die auf RealSubject gleichberechtigt zugreifen können. „realSubjet“ samt Pfeil von Proxy nach RealSubject meint, dass ein Proxy ein RealSubject kennen muss, um ein Objekt von RealSubject ersetzen zu können.

- **Eigenschaft:**
  1.) Platzhalterobjekt
  2.) Mehrere untersch. Proxies (auch von untersch. Typen) können auf ein Objekte von RealSubjekt zugreifen, diese werden dann meisten in Aggregaten (z.B. Listen) organisiert.
  3.) Kontrollierte Zugriffe auf Objekte (RealSubject)

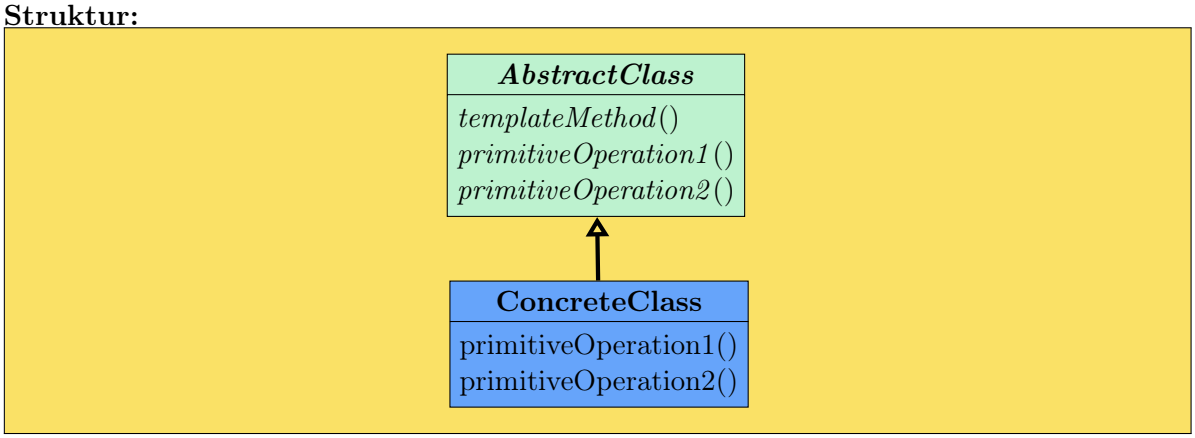
4.) Nicht wie Decorator ((zeitliche) Erweiterung eines Objekts (nicht Klasse) um Funktionalität), sondern bloße Kontrolle über Zugriff auf das (zeitlich) vertretende (reale) Objekt.

– **Details der Implementierung:**

```
/*Subject*/
public interface Something {
        void doSomething();
}

/*RealSubject*/
public class ExpensiveSomething implements Something {
        public void doSomething() { ... }
}

/*Proxy*/
public class VirtualSomething implements Something {
        private ExpensiveSomething real = null; /*Referenz auf teures Objekt*/
        public void doSomething() {
                if (real == null) real = new ExpensiveSomething();
                real.doSomething();
        }
}
```

(7) **Template-Method-Entwurfsmuster:**

- **Name**:
  Template-Method (228-230) (**Verhaltenbeschreibendes Entwurfsmuster**)
- **Problemstellung:**
  Für ein Problem gibt es ein Verhalten/**Routine**/Algorithmus (immer wieder gleiches Verhalten)
- **Anwendungsgebiet:**
  1.) Implementiere unveränderlichen Teil eines Algorithmus nur einmal. (Unterklassen sorgen sich um genauere Implementierung.)
  2.) Gemeinsames Verhalten mehrerer Unterklassen vereint in einer einzigen Klasse. (Vermeidet Dublikate).
  3.) Steuerung von Verhalten (in Unterklassen) durch Hooks (→ Defaultmethoden, in Unterklassen wird gesteuert (durch bspw. Booleans) welche Hooks überschrieben werden sollen.)
- **Struktur:**



Abstrakte Klasse/Interface: *AbstractClass* bündelt das Verhalten (primitiveOperation1(), primitiveOperation2()) aus den Unterklassen. Primitive Operation, die überschrieben werden müssen, sind als *abstract methods* in der Oberklasse implementiert. *templateMethod()* soll nicht überschrieben werden und „feuert“ Paket an Methoden.

Klassen: *ConcreteClass* leitet sich aus Oberklasse ab/erbt und überschreibt alles Konkrete

- **Eigenschaft:**
  1.) Konzept stellt fundamentale Technik zur Wiederverwendung von Programmcode dar. (Sind in Klassenbilbiotheken und Frameworks sinnvoll → faktorisieren gemeinsames Verhalten)
  2.) Führen zu umgekehrter Kontrollstruktur → *Hollywood-Prinzip* („Don’t call us, we’ll call you.“: Oberklasse ruft Methoden der Unterklassen auf)
  3.) Hooks (=meist leere Default-Klassen in der abstrakten Oberklasse, auch als abstrakte Klasse denkbar) sorgen für Rahmenverhalten.
- **Details der Implementierung:**

```
/*Abstract-Class*/
public abstract class Auto {
        public String marke;
        /*templateMethod() um Routine zu starten*/
        public void templateMethod(){
                System.out.println("Das Auto "+ marke + "mit: ");
                hasFourDoors();
                hasTwoDoors();
                hasColor();
        }
        /*Hook (methods)*/
        abstract boolean hasFourDoors();
        abstract boolean hasTwoDoors();
        abstract String hasColor();
}
/*Erste ConcreteClass*/
public class Fiat extends Auto {
        public Fiat(String marke){super.marke = marke;}

        @Override
        boolean hasFourDoors() {System.out.println("Doors: Hat vier Türen.");return false;}
        /*übernimmt Hook*/
        @Override
        boolean hasTwoDoors() {return false;}
        @Override
        String hasColor() {System.out.println("Farbe: Ist weiß.");return null;}
}
/*Zweite ConcreteClass*/
public class Ferrari extends Auto {
        public Fiat(String marke){super.marke = marke;}

        /*übernimmt Hook*/
        @Override
        boolean hasFourDoors() {return false;}
        @Override
        boolean hasTwoDoors() {System.out.println("Doors: Hat zwei Türen.");return false;}
        @Override
        String hasColor() {System.out.println("Farbe: Ist rot.");return null;}
}

public static void main(String[] args) {
        Auto a1 = new Ferrari("Ferrari");
        Auto a2 = new Fiat("Fiat");
        a1.templateMethod();
        a2.templateMethod();
}
```

(8) **Visitor-Entwurfsmuster:**

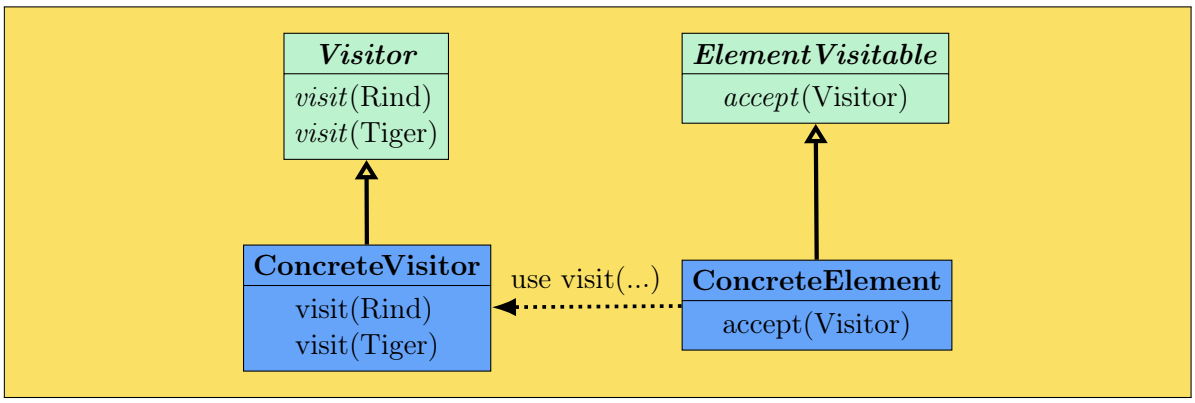
- **Name**:
  Visitor-Pattern (159-161, 204) (**Verhaltenbeschreibendes Entwurfsmuster**)
- **Problemstellung:**

  Beispielsweise könnte die Situation vorliegen, dass es in einem Programm viele Stellen mit dynamischen Typabfragen (viel typbezogene if-Anweisungen) bzw. viele Typumwandlungen gibt. Diese kann man im Zuge einer Refaktorisierung in mehrfaches dynamisches Binden umwandeln (Achtung: die Umwandlung kann viele Methoden erzeugen).
- **Anwendungsgebiet:**

  1.) **Zusammenfassende Struktur**
  Client $\xrightarrow{\text{1. Dynamisches Binden}}$ *Elementklassen – Ebene* $\xrightarrow{\text{2./X. Dynamisches Binden}}$ *Visitorklassen – Ebene*

  2.) Eine Visitorklassen auf der x. Ebene ist damit zugleich auch eine Elementklasse (= Ausgangspunkt für weiteres Dynamisches Binden).

  3.) Jede weitere Ebene stellt im Prinzip einen **Verteiler für weiteres dynmisches Binden** dar, das ist auch der Grund, warum mit jeder weiteren Ebene sehr viele Klassen zum Programm hinzukommen: Mit k dynamischen Binden, m Elementklassen (Tierklassen) und n_i Möglichkeiten für i-te Bindung (i = 1...k) (n = Futterarten):
  Bedeutung für Anzahl Visitormethoden: $Anzahl - Visitormethoden = m * n$
  Bedeutung für Anzahl Dynamisches Binden: $n_1 + n_1 * n_2 + ... + n_1 * n_2 * ... * n_k$
- **Struktur:**



  Abstrakte Klasse/Interface: Das Interface *ElementVisible* ist das Interface für die Element-Klassen, die der Client beim 1. Dynamischen Binden nutzt. Das Interface *Visitor* definiert alle Visitor-Typen und stellt die 2./X. Dynamische Bindestelle dar (ausgehend von der vorgeschaltenden Element). Die Visitor-Methode in ElementVisitable ist bspw. vom Typ Futter und die *accept()*-Methode entspricht der *friss()*-Methode, *visit*() entspricht bspw. *vonTigerGefressen*()

  Klassen: Klassen *ConcreteElement* und *ConcreteVisitor* leiten sich entsprechend ab

- **Eigenschaft:**

  1.) Setzt mehrfaches dynamisches Binden um und vermeidet so dynamische Typabfragen und Typumwandlungen (Casts) → siehe **Multimethoden** (=zur Laufzeit Festlegung der Methodenauswahl durch dynam. Typen von x und y in x.method(y))

2.) **Hauptmerkmal: Element-Klassen**(Aufrufer) und **Visitor-Klassen**(Parameter) d.h. **kurz:**

Elementklasse.*accept*(**Visitorklasse**).

Jedoch sind Element- und Visitorklassen aber auch gegeneinander austauschbar (Visitormethoden können auch in den vormals Elementklassen stehen)

3.) **Problematisch allerdings:** (mögliche) hohe Anzahl an Methoden

– **Details der Implementierung:**

```
/*Element-Klassen*/
public abstract class Tier {
        public abstract void friss(Futter futter);
}
/*Element-Klassen*/
public class Rind extends Tier {
        public void friss(Futter futter) { futter.vonRindGefressen(this);}
}
/*Element-Klassen*/
public class Tiger extends Tier {
        public void friss(Futter futter) { futter.vonTigerGefressen(this);}
}


/*Visitor-Klassen*/
public abstract class Futter {
        abstract void vonRindGefressen(Rind rind);
        abstract void vonTigerGefressen(Tiger tiger);
}
/*Visitor-Klassen*/
public class Gras extends Futter {
        void vonRindGefressen(Rind rind) { ... }
        void vonTigerGefressen(Tiger tiger) { tiger.fletscheZaehne();}
}
/*Visitor-Klassen*/
public class Fleisch extends Futter {
        void vonRindGefressen(Rind rind) { rind.erhoeheWahrscheinlichkeitFuerBSE();}
        void vonTigerGefressen(Tiger tiger) { ... }
}


public static void main(String[] args) {
        ...
        Farm farm = new Farm(); /*unbekannte Klasse Farm*/
        farm(tier.friss(futter)); /*tier wurde irgendwann vorher deklariert/initialisiert*/
        /* 1. Dynam. Binden.: tier → Rind/Tiger */
        /* 2. Dynam. Binden.: futter → Gras/Fleisch */
}
```

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

   - **interne Iteratoren:** Z.B. Streams. Diese Art der Iteratoren kontrollieren selbst, wann der nächste Iterationsschritt (next()) aufgerufen wird wie bspw. Stream, der ohne explizites *next()*-Aufruf von außen auskommt. Bei internen Iteratoren ist die Schleife selbst enthalten, sodass Methoden wie *hasNext()* oder *next()* von außen nicht aufrufbar sind. Anderes Bsp. wäre eine *forEach* in einer *Map*. Interne Iteratoren sind meist einfacher zu verwenden.
   - **externe Iteratoren:** Diese werden außerhalb durch Bildung eines Iteratorobjekts über einem beliebigen Aggregat (z.B. Liste, Baum, Heap, Map etc.) erzeugt und kontrolliert (while-Schleife mit *hasNext() und innerhalb next()-Aufruf*). Externe Iteratoren sind flexibler (zwei Aggregate sind so leichter miteinander vergleichbar) und häufiger im Einsatz.

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

   - **Vorteil:** Durch die (einheitliche) Verwendung eines Itertors wird dem Aufrufer ermöglicht, ohne Kenntnis von außen (bequem = immer die gleichen Methoden (hasNext, next, remove)) auf jegliches Aggregat zugreifen zu können.
   - **Nachteil:** Nachteilig ist, dass durch die Unwissenheit über das Aggregat und/oder das Unwissen über konkrete Implementierungsdetails des Iterators bei gewissen Aggregaten durch **sequentiellen Zugriff** die spezielle „Aggregatstruktur“ abhanden/zerstört wird.
     Bspw. wäre es bei einem **Binary-tree** entscheidend zu wissen, ob die Ausgabe unter Inorder/Preorder/Postorder oder bei einem **normalen Stack** die Ausgabe unter LIFO oder FILO geschieht (etc.).

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

   - Man kann Iteratoren direkt im Aggregat selbst durch eine geschachtelte Klasse/innere Klasse definieren. So kann diese Klasse bzw. der Iterator auf private Inhalte des Aggregats zugreifen. Insgesamt muss der **Algorithmus zur (sequentiellen) Durchwanderung des Aggregats** entweder direkt im Aggregat selbst (häufiger) oder in der Iterator-Klasse (einfacher) definiert sein. Wenn das Aggregat dann noch auf **private Implementierungsdetails des Aggregats zugreifen** kann, so kann der Durchwanderungsalgorithmus weitaus einfacher in der inneren Iterator-Klasse implementiert werden.
     Problematisch: starke Abhängigkeit zwischen Aggregat und Iterator noch erhöht.

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

   - **Problem:** Problematisch ist der Moment, bei welchem ein Iterator-Objekt über ein Aggregat erzeugt und aktiviert wird, zugleich das Aggregat aber verändert wird (hinzufügen/löschen von Elementen). Hier können doppelte/falsche Ausgaben erfolgen.
   - **Lösung:** Im Moment der Erzeugung eines Iterators wird das Aggregat kopiert (einfache Lösung). Das ist aber eine nicht so gute Lösung, **robuste Iteratoren** bekommen den gleichen Effekt ohne diesen Lösungsansatz hin.
   - **Wozu Robustheit:** Ein Iterator soll ein Aggregat wahrheitsgemäß wiedergeben. Klarerweise führt eine falsche Wiedergabe zu willkürlichen und unberechenbaren Programmeigenschaften, **Robustheit** erhält Berechenbarkeit und Korrektheit eines Programms.

6. Wird die Anzahl der benötigten **Klassen** im System bei Verwendung von Factory-Method(1), Prototype(2),

Decorator(3) und Proxy(4) (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?

(1) Mit der **Factory-Method** (erzeugendes Entwurfsmuster) wird die Anzahl an Klassen **erhöht** (die parallele Creator-Klassen- und Product-Klassenhierarchie bildet viele Klassen aus).
(**Zentral** in der Factory-Method ist die Kopplung des **ConcretePoduct** an einen **ConcreteCreator** durch ein Argument.)

(2) Mit dem **Prototype** (erzeugendes Entwurfsmuster) wird die Anzahl an Unterklassen **vermindert** (durch *clone()*-Methode lassen sich neue Objekte durch Objektkomposition statt durch Vererbung erzeugen/weiterleben).
(**Zentral** für eine **ConcretePrototype**-Klasse ist die **Bereitschaft**, sich **clonen** zu können (jede Klasse enthält eine *clone()*-Methode).)

(3) Mit dem **Decorator** (strukturelles Entwurfsmuster) wird die Anzahl an Unterklassen eher **vermindert** (da das „normale“ Konzept Vererbung mit Produktion einer großen Klassenanzahl wäre, d.h. immer wenn eine Klasse eine spezielle Funktion braucht, braucht man eine neue Klasse, die speziell diese Anforderung erfüllt).
(**Zentral** ist hierbei, dass Objekte dynamisch zur Laufzeit mit neuen Funktionen modifiziert werden können (können Funktionen erhalten, aber auch wieder abgegeben).)

(4) Mit dem **Proxy** (strukturelles Entwurfsmuster) wird die Anzahl an Unterklassen eher **erhöht** (, weil es zu einer „teuren Klasse“ immer mindestens ein Proxy geben muss, im „Normalfall“ nimmt man den teuren Zugriff ohne extra Proxy in Kauf).
(**Zentral** ist hierbei, dass man statt Laden eines teuren Objekts zuerst einen „leichteren“ Proxy nutzt, der möglichst intelligent ein teures Objekt referenziert. Ähnlichkeit hat es mit dem Decorator, dieser erweitert und verwaltet aber die Funktionalität eines Objekts zur Laufzeit, wobei der Proxy nur den Zugriff verwaltet.)

⇒ Übersicht von (vermindert) < (unverändert) < (erhöht) bzgl. **Klassen**-Anzahl:

(Prototype, Decorator) < **(-)** < **(Factory-Method, Proxy)**:

7. Wird die Anzahl der benötigten **Objekte** im System bei Verwendung von Factory-Method(1), Prototype(2), Decorator(3) und Proxy(4) (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverändert?

(1) Mit der **Factory-Method** (erzeugendes Entwurfsmuster) bleibt die Anzahl der Objekte **unverändert** (es geht hier nur darum, die Unterklassen in der Bestimmung der Typen mit einzubeziehen (durch die Argumente), das steigert aber nicht ursächlich die Anzahl an Objekten, die ohnehin benötigt werden).

(2) Mit dem **Prototype** (erzeugendes Entwurfsmuster) wird die Anzahl an Objekten eher **vermindert** (falls neue Objekte von einem Typ gebraucht werden, so muss dafür nicht erst extra eine neue Klasse mit jeweiligen neuen Objekten erzeugt werden, sondern es kann durch clone() und Objektkomposition diese neuen Anforderungen erreicht werden).

Falls von einem Objekt mehr Funktionalität gefordert wird, so muss nicht extra eine neue Klasse erstellt und ein Objekt daraus gebildet werden, sondern ein bestehendes Objekt kann wiederverwendet werden, indem es dynamisch neue Funktionalität zur Laufzeit erhält (oder auch wieder abgenommen bekommt).

(3) Mit dem **Decorator** (strukturelles Entwurfsmuster) wird die Anzahl an Objekten **erhöht** (um die Funktionalität eines bestehenden Objekts dynamisch erweitern zu können braucht es oftmaliges Umkopieren und Hilfsobjekte, es entstehen viele ähnliche Objekte, mehr, als wenn man von vorhinein statisch die Funktionalität festlegt; man spart sich damit dann eine hohe Anzahl an Objekten, verliert aber auch die (dynamisch) Felxibilität).

(4) Mit dem **Proxy** (strukturelles Entwurfsmuster) wird die Anzahl an Objekten eher **erhöht** (, da ähnlich wie bei den Unterklassen Referenzen auf „teure" Objekte/Klassen selbst mit einem Proxy-Objekt vorab erstellt werden muss).

⇒ Übersicht von (vermindert) < (unverändert) < (erhöht) bzgl. **Objekt**-Anzahl:

(Prototype) < **(Factory-Method)** < **(Decorator, Proxy)**:

8. Vergleichen Sie Factory-Method mit Prototype.

   Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?

- Beides sind erzeugende Entwurfsmuster - daher geht es bei Problemen immer grundsätzlich um Möglichkeitserweiterungen (im funktionalen Sinne):
- Das Muster **Factory-Method** ist ebenfalls gut geeignet, falls noch nicht bekannt ist, wann welcher Typ wo zum Einsatz kommen wird, allerdings muss hier die Funktionalität und damit auch die Kenntnis über die Klassen im Vorhinein bekannt sein.
- **Prototype**: Wenn man noch nicht weiß, welche Anforderungen/Funktionalitäten (mittels Typen) ein Programm während des Programmverlaufs erfüllen muss und es daher zu Veränderung zur Laufzeit kommen wird, so ist dieses dynamische Entwurfsmuster besser geeignet → clone()-Methode setzt den dynamischen Aspekt um. Zudem muss der Anwender noch kein konkretes Wissen über die Produktklassen haben, diese sind ähnlich wie bei Iteratoren über Aggregate versteckt.

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

- **Problem 1:** Es ist zunächst schwierig sicherzustellen, dass ein einziges Objekt (Singleton) von dem vorgegebenen Typ erstellt wird (leichter wäre unendlich viele oder keins). Das kann man durch die statische Objektvariable, die statische getInstance()-Methode und ganz besonders durch den privaten Konstruktor(!) erreichen.
- **Problem 2:** Der Zugriff auf ein Singleton muss von überall (global) erfolgen können.
- **Problem 3:** Neben der Abschottung nach außen muss das Singleton-Objekt auch noch seine „normalen" Aufgaben erledigen (bspw. Chat-Singleton oder Spooler-Singleton).

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

- **Ähnlichkeiten:** Ähnlich sind sich beider Muster in ihrem strukturellen Element, nämlich die Struktur eines Objekts ändern zu können: **Decorator** durch Transformieren der Objekte dynamisch zur Laufzeit, um so Objekte mit neuen Funktionen auszustatten oder diese wieder zu nehmen; es geht hier also um eine zeitliche Erweiterung/strukturelle Umgestaltung.
  Auch **Proxies** können Objekte strutkurell verändern, indem sie ihnen (wenn sie „teuer" sind) vorgeschalten werden.

Beide greifen dabei auf Objektebene/Instanzen von Klassen zu, es geht dabei nicht um strukturelle Veränderung von Klassen selbst (, sondern eben nur um die Veränderungen der Klassen-Instanzen (=Objekte)).

- **Unterschiede: Decoratoren** haben eine zeitlichen Zugriff auf Objekte als Ziel zwecks temporärer struktureller Veränderung (hstl. Funktionalität).
  Proxies haben dagegen bloße Zugriffskontrolle als primäres Ziel vor Augen (smarte Proxies, evtl. noch Erweiterung um funktionale Eigenschaften).

11. Welche Probleme kann es beim Erzeugen von Kopien im Prototype geben?

    Was unterscheidet flache Kopien von tiefen?

- Problematisch kann die *clone()*-Methode sein, da nicht ganz klar ist, wie eine Kopie einer Klasse aussehen soll. Einerseits gibt es das Problem zwischen *shallow copy* und *deep copy*. Andererseits ist zudem nicht klar, wieviel von einer Klasse kopiert werden muss, um alles Wesentliche zu erhalten (Signatur alleine wäre zu wenig).
- **Unterschied:**
  - **flache Kopie (shallow copy):** Bezieht sich lediglich darauf, nur den Pointer bzw. die Referenz auf ein Objekt/Klasse zu kopieren. Es handelt sich also nicht um eine wirkiche Kopie im Sinne, dass man das zu kopierende Objekt/Klasse cloned.
  - **tiefe Kopie (deep copy):** Hierbei wird wirklich ein Clone eines Objekts/einer Klasse erzeugt, d.h., dass die in einem Objekt/einer Klasse enthaltende Daten kopiert werden und dadurch auch eine neue Speicherstelle benötigt wird und der Clone sich dann selbstständig weiterentwickeln kann.

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

- **Gute Eignung:** Wenn es darum geht, Objekte (nicht Klassen) möglichst dynamisch strukturell zu modifizieren, so ist dieses Entwurfsmuster sehr gut geeignet.
  Zudem geht das Decorator-Muster sparsamer mit Klassen- und Objektanzahl um. (Achtung: problematisch sind viele ähnliche (!) Objekte, die durch dieses Muster entstehen)
- **Schlechte Eignung:** Einerseits ist dieses Pattern weniger gut geeignet, wenn bereits bzgl. der Struktur und den Aufgaben (hstl. Funktionen) im Vorfeld alles geklärt ist und dynamisch nichts weiter hinzugefügt werden muss. Andererseits ist dieses Entwurfsmuster nicht geeignet, wenn man sich auf Objektidentität verlassen muss, dafür wird zuviel auf Zwischenobjekte umkopiert.

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

- Decorators und Proxies haben gemein, dass sie beide strukturverändernd sind. Ein Hintereinanderschalten würde beide Vorteile (aber auch Nachteile) kombinieren.
  Durch den Decorator fließt dynamisch zur Laufzeit Objekt-Flexibilität ein, durch den Proxy kann immer wieder datenschonend auf besonders große Objekte zugegriffen werden, wobei die Perfomanz durch die Ressourcenschonung erhöht werden kann.
  Was hier geschieht ist, dass man Objekte funktional verschlankt oder erweitert und im Falle der Erweiterung (Decorator) trotzdem ressourcensparend auf wesentliche Eigenschaft schnell und schlank zugreifen kann (Proxy).

14. Was unterscheidet Hooks von abstrakten Methoden?

- Der **wesentliche Unterschied** zwischen beiden ist, dass **abstrakte Methoden** in der Unterklassen überschrieben werden **müssen**, um dort die Implementierung bzw. Funktionalität festzulegen.
  Dagegen **können Hooks** (welche Default-Verhalten beschreiben, bspw. run(), exit(),...) leer bleiben (keine Implementierung) bzw. **müssen** in Unterklassen **nicht überschrieben** werden, nur, wenn man die Funktionalität verändern will. Es wird allerdings erwartet, dass Hooks in Unterklassen überschrieben werden. (230)

Kapitel 5 | Themenübersicht