Objektorientierte Programmiertechniken

## Kapitel 1

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

Unter Programmierparadigma versteht man eine bestimmte Denkweise oder Art der Weltanschauung.
Entsprechend entwickelt man Programme in einem gewissen Stil.
Bei den Paradigmen unterscheidet man zwischen:

### 2. Wozu dient ein Berechnungsmodell?

Hinter jedem Paradigma steckt ein Berechnungsmodell, die immer einen formalen Hintergrund haben.
Sie müssen konsistent und in der Regel Turing−vollständig sein, also alles ausdrücken können, was als berechenbar gilt. Eignet sich nur dann als Grundlage eines Paradigmas, wenn die praktische Umsetzung ohne übermäßig großen Aufwand zu Programmiersprachen, Entwicklungsmethoden und Werkzeugen hinreichender Qualität führt.

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

**Funktionen:** spielen eine zentrale Rolle. Primitiv-rekursive Funktionen: gehen von einer vorgegebenen Menge einfacher Funktionen aus und bilden daraus durch Komposition und Rekursion neue Funktionen. Können vieles berechnen aber nicht alles was berechenbar ist. Turing-Vollständigkeit erreicht man durch µ-rekursive Funktionen, wo der µ−Operator angewandt auf partielle Funktionen das kleinste aller möglichen Ergebnisse liefert.

**Prädikatenlogik:** etabliertes, mächtiges mathematisches Werkzeug.

**Constraint−Programmierung:** Einschränkungen auf Variablen wie etwa „x< 5“ zusätzlich zu solchen wie „A oder B ist wahr“. Fortschrittliche Beweistechniken können solche Contraints vergleichsweise effizient lösen. Heute zum auflösen von Contraints vorwiegend fertige Bibliotheken, die fast überall eingebunden werden können.

**Temporale Logik und Petri−Netze:** In temporaler Logik kann man in logischen Ausdrücken recht einfach zeitliche Abhängigkeiten abbilden. Beispielsweise kann man festlegen, dass eine Aussage nur vor oder nach einem bestimmten Ereignis gilt. Verwendung: Synchronisation in nebenläufigen Programmen oder die Steuerung von Maschinen.
Petri-Netze gute Möglichkeiten zur grafischen Veranschaulichung komplexer zeitlicher Abhängigkeiten bieten, ist die Beweisführung in temporalen Logiken einfacher.

**Freie Algebren:** erlauben uns die Spezifikation beinahe beliebiger Strukturen (beispielsweise von Datenstrukturen) über einfache Axiome. wichtig im Zusammenhang mit Modulen und Typen.

**Prozesskalküle:** Modellierung von Prozessen(vergleichbar mit Threads) in nebenläufigen Systemen.

**Automaten:** Man unterscheidet Automaten unterschiedlicher Komplexität, die gleichzeitig verschiedene Arten von Grammatiken und entsprechende Sprachklassen darstellen. Wegen ihrer anschaulichen Darstellung werden Automaten nicht selten zur Spezifikation des Systemverhaltens verwendet.

### While, GoTo & Co

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

**Kombinierbarkeit:** Bestehende Programmteile sollen sich möglichst einfach zu größeren Einheiten kombinieren lassen, ohne dass diese größeren Einheiten dabei ihre einfache Kombinierbarkeit einbüßen.

**Konsistenz:** Oft müssen mehrere Formalismen kombiniert werden, etwa solche zur Beschreibung von Algorithmen und andere zur Beschreibung von Datenstrukturen, zu einer Einheit verschmelzen. Alle Konzepte sollen in sich und miteinander konsistent sein, also gut zusammenpassen. Konsistent ist in der Praxis wichtiger als Optimalität. → wenige konsistente Konzepte, dadurch nötige Einfachheit.

**Abstraktion:** ursprünglichstes und noch immer wichtigstes Ziel der Verwendung höherer Programmiersprachen ist die Abstraktion über Details der Hardware und des Systems. Programme sollten nicht von solchen Details abhängen, sondern möglichst portabel sein.

**Systemnähe:** Programme müssen effizient auf realer Hardware ausführbar sein. Effizienz lässt sich scheinbar am leichtesten erreichen, wenn das Paradigma wesentliche Teile der Hardware und des Betriebssystems direkt widerspiegelt. Unverzichtbar wenn Hardwarekomponente direkt angesprochen werden müssen.

**Unterstützung:** Beste Paradigma hat keine Wert, wenn entsprechende Programmiersprachen, Entwicklungswerkzeugen sowie vorgefertigte Programmteile fehlen. So manches ursprünglich nur mittelmäßige Paradigma hat sich wegen guter Unterstützung trotzdem durchgesetzt.

**Beharrungsvermögen:** Jeder Paradigmenwechsel bedeutet, dass so manches Wissen verloren geht und neue Erfahrungen erst gemacht werden müssen. Für eine Wechsel braucht es sehr überzeugende Gründe: Wechsel in Bereich schon vollzogen wesentlich erfolgreicher als im alten Paradigma. Killerapplikationen: Programme die Erfolg einer Technik oder eines Paradigmas deutlich zeigen → Potenzial althergebrachte Techniken oder Paradigmen in diesem Bereich zu ersetzten.

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

## Wie äußert sich dieses Spannungsfeld?

Neue praktische Erfahrungen im Spannungsfeld zwischen zum Teil widersprüchlichen Forderungen.
Es ist es unmöglich, folgende Forderungen gleichzeitig und in vollem Umfang zu erfüllen:
• Flexibilität und Ausdruckskraft sollen in kurzen Texten die Darstellung aller vorstellbaren Programmabläufe ermöglichen.
• Lesbarkeit und Sicherheit sollen Absichten hinter Programmteilen sowie mögliche Inkonsistenzen leicht erkennen lassen.
• Die Konzepte müssen verständlich bleiben und es muss klar sein, was einfach machbar ist und was nicht.

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

Die strukturierte Programmierung bringt mehr Struktur in die prozedurale Programmierung.
Jedes Programm bzw. jeder Rumpf einer Prozedur ist nur aus drei einfachen Kontrollstrukturen aufgebaut:
• Sequenz (ein Schritt nach dem anderen)
• Auswahl (Verzweigung im Programm)
• Wiederholung (Schleife, Rekursion oder ähnliches)
wesentliches Ziel: jede Kontrollstruktur nur je eine wohldefinierten Einsteigs- und Ausstiegspunkt hat. Leichteres Verfolgen des Programmpfades, erhöhte Lesbarkeit → auf Kosten einer etwas verringerten Flexibilität und Ausdruckskraft. Positiven Erfahrungen mit erhöhter Lesbarkeit überwiegen über Flexibilität.

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

Deklarative Paradigmen: radikaler Ansatz
strebt referentielle Transparent als Eigenschaft an. Ausdruck referentiell transparent, wenn er durch seinen Wert ersetzt werden kann, ohne die Semantik des Programms dadurch zu ändern.
Bsp: f(x) + f(x) = 2 f(x)
Objektorientierte Paradigmen: gemäßigter Ansatz
gibt keine referentielle Transparenz, man nimmt an, dass es Querverbindungen gibt und beschränkt sie lokal auf einzelne Objekte

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

Ein Ausdruck ist *referentiell transparent,* wenn er durch seinen Wert ersetzt werden kann, ohne die Semantik des Programms dadurch zu ändern. (Bsp: 3+4 lässt sich durch 7 oder 14/2 ersetzten)

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

Ein− und Ausgaben sind Seiteneffekte, die referenzielle Transparenz unmöglich macht.

Gelöst hat man es, indem man Ein− und Ausgaben nur gut sichtbar und ganz oben in der Aufrufhierarchie erlaubt und verbannt sie aus allen anderen Funktionen.

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

In der OOP geht man, wo es keine referentielle Transparenz gibt, mit Seiteneffekten ganz offensiv um. Man nimmt immer an, dass es entsprechende Querverbindungen gibt. Wenn man sie lokal auf einzelne Objekte beschränkt, kann man sie jedoch überschaubar halten.

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

First Class Entities sind Funktionen, die wie normale Daten verwendbar sind, in Variablen abgelegt werden können, als Argumente an andere Funktionen übergeben und als Ergebnisse von Funktionen zurückbekommen werden können.
− Häufig sehr kompliziert als vergleichbare Konzepte, weil die uneingeschränkte Verwendbarkeit eine große Zahl an zu berücksichtigenden Sonderfällen nach sich zieht.
+ Die uneingeschränkte Verwendbarkeit bringt Vorteile beispielsweise beim Einsatz mit Funktionen höherer Ordnung (Funktionen mit Funktionen als Parameter).Sie bringen eine neue Dimension in die Programmierung. Häufiger Gebrauch von Funktionen höherer Ordnung führt zu einem eigenen Paradigma, der applikativen Programmierung, bei der man quasi Schablonen von Programmteilen schreibt, die dann durch Übergabe von Funktionen ausführbar werden.

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

Ein häufiger Gebrauch von Funktionen höherer Ordnung führt zu einem eigenen Paradigma, der *applikativen Programmierung.* Man schreibt quasi Schablonen von Programmteilen, die dann durch Übergabe von Funktionen (zum Füllen der Löcher in den Schablonen) ausführbar werden.

## 13. Welche Modularisierungseinheiten kann man unterscheiden, und was sind ihre charakteristischen Eigenschaften?

• **Modul** (Übersetzungseinheit, zyklenfrei, enthält Deklarationen bzw. Definitionen von zusammengehörigen Variablen, Typen, Prozeduren, Funktionen, usw.); getrennt übersetzte Module werden von einem Linker oder erst zur Laufzeit zum ausführbaren Programm verbunden. Schnittstelle des Moduls: zusammengefasste Informationen über Inhalte des Moduls, die auch in anderen Modulen verwendbar sind.

• **Objekt** (zur Laufzeit erzeugt, keine Einschränkungen hinsichtlich zyklischer Abhängigkeiten, kapseln Variablen und Methoden zu logischen Einheiten (Kapselung) und schützen private Inhalte vor Zugriffen von außen (Data-Hiding) → Datenabstraktion
wichtigste Eigenschaften: Identität, Zustand, Verhalten; Identität und Gleichheit: zwei durch verschiedene Variablen referenzierte Objekte sind identisch wenn es sich um ein und dasselbe Objekt handelt. Zwei Objekte sind gleich wenn sie denselben Zustand und dasselbe Verhalten haben, auch wenn sie nicht identisch sind.(Kopie)

• **Klasse** Schablone für die Erzeugung neuer Objekte beschrieben. gibt die Variablen und Methoden des neuen Objektes vor und spezifiziert seine wichtigsten Eigenschaften.
möglich Klassen von Klassen abzuleiten;
oft auch ein Modul; zyklische Abhängigkeiten zwischen Klassen sind verboten;

• **Komponente** eigenständiges Stück Software, das in ein Programm eingebunden wird. alleine ist Komponente nicht lauffähig, da sie die Existenz anderer Komponenten voraussetzt und deren Dienste in Anspruch nimmt.
Komponenten ähneln Modulen: Beides sind Übersetzungseinheiten und Namensräume, die Datenkapselung und Data-Hiding unterstützen.Komponente flexibler: Während ein Modul Inhalte ganz bestimmter namentlich genannter anderer Module importiert, importiert eine Komponente Inhalte von zur Übersetzungszeit nicht genau bekannten anderen Komponenten. Erst beim Einbinden in das Programm werden diese anderen Komponenten bekannt. Modulen und Komponenten offen, wo exportierte Inhalte verwendet werden, aber bei Komponenten ist zusätzlich noch offen, von wo importierte Inhalte kommen. Letzteres verringert die Abhängigkeit der Komponenten voneinander. Deswegen gibt es bei der getrennten Übersetzung kein Problem mit zyklischen Abhängigkeiten.

• **Namensraum** jede Modularisierungseinheit bildet eigenen Namensraum, Namenskonflikte gut abfedern; globale Namen (Namen von Klassen , Modulen und Komponenten) gilt das nicht → das zu regeln wird oft Softwareentwickler überlassen. Hilfsmittel: standardmäßig vorgegebene Verzeichnisse, in denen automatisch nach verwendeten Modul- und Klassennamen gesucht wird; fortgeschrittener Modularisierungseinheiten die man Namensräume nennt: fassen mehrere Modularisierungseinheiten zu einer Einheit zusammen, ohne die getrennte Übersetzbarkeit zu zerstören.

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

Klar definierte Schnittstellen zwischen Modulen sind sehr hilfreich: Einerseits braucht der Compiler Schnittstelleninformation um Inhalte anderer Module verwenden zu können, andererseits ist diese Information auch beim Programmieren wichtig um Abhängigkeiten zwischen Modulen besser zu verstehen. Meist wird nur ein kleiner Teil des Modulinhalts von anderen Modulen verwendet. Schnittstellen unterscheiden klar zwischen Modulinhalten, die von anderen Modulen zugreifbar sind, und solchen, die nur innerhalb des Moduls gebraucht werden.
Erstere werden exportiert, letztere sind privat. Private Modulinhalte sind von Vorteil: Sie können vom Compiler im Rahmen der Programmiersprachsemantik beliebig optimiert, umgeformt oder sogar weggelassen werden, während für exportierte Inhalte eine gewissen Regeln entsprechende Zugriffsmöglichkeit von außen bestehen muss. Änderungen privater Modulinhalte wirken sich nicht auf andere Module aus. Änderungen exportierter Inhalte machen hingegen oft entsprechende Änderungen in anderen Modulen nötig, die diese Inhalte verwenden.

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

**Namensraum** jede Modularisierungseinheit bildet eigenen Namensraum, Namenskonflikte gut abfedern; globale Namen (Namen von Klassen , Modulen und Komponenten) gilt das nicht → das zu regeln wird oft Softwareentwickler überlassen. Hilfsmittel: standardmäßig vorgegebene Verzeichnisse, in denen automatisch nach verwendeten Modul- und Klassennamen gesucht wird; fortgeschrittener Modularisierungseinheiten die man Namensräume nennt: fassen mehrere Modularisierungseinheiten zu einer Einheit zusammen, ohne die getrennte Übersetzbarkeit zu zerstören.

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

Weil das aufgrund der getrennten Übersetzung nicht möglich ist. Wenn ein Modul B Inhalte eines Moduls A importiert, kann A keine Inhalte von B importieren. Das hat mit der getrennten Übersetzung zu tun: Modul A muss vor B übersetzt werden, damit der Compiler während der Übersetzung von B bereits auf die übersetzten Inhalte von A zugreifen kann. Würde A auch Inhalte von B importieren, könnten A und B nur gemeinsam übersetzt werden, was der Definition von Modulen widerspricht.
Sowohl bei Modulen als auch Komponenten ist offen, wo exportierte Inhalte verwendet werden, aber bei Komponenten ist zusätzlich offen, von wo importierte Inhalte kommen. Letzteres verringert die Abhängigkeit der Komponenten voneinander. Deswegen gibt es bei der getrennten Übersetzung kein Problem mit zyklischen Abhängigkeiten.

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

Datenabstraktion: Kapselung in Kombination mit Data−Hiding
Kapselung: Variablen und Methoden werden (in Objekten ) zu logischen Einheiten zusammengefasst
Data−Hiding: private Inhalte werden vor Zugriffen von außen geschützt.

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

Die Einbindung von Komponenten in Programme ist aufwendiger als die von Modulen, da dabei auch die Komponenten festgelegt werden müssen, von denen etwas importiert wird. Oft werden zuerst die einzubindenden Komponenten zum Programm hinzugefügt und erst in einem zweiten Schritt festgelegt, von wo importiert wird.

## 19. Wie kann man globale Namen verwalten und damit Namenskonflikte

**verhindern?**
globale Namen (Namen von Klassen , Modulen und Komponenten) gilt das nicht → das zu regeln wird oft Softwareentwickler überlassen. Hilfsmittel: standardmäßig vorgegebene Verzeichnisse, in denen automatisch nach verwendeten Modul- und Klassennamen gesucht wird; fortgeschrittener Modularisierungseinheiten die man Namensräume nennt: fassen mehrere Modularisierungseinheiten zu einer Einheit zusammen, ohne die getrennte Übersetzbarkeit zu zerstören.

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

Es kann zur Laufzeit erfolgen:
Konstruktor: Beim Erzeugen eines Objekts werden Objektvariablen initialisiert
Initialisierungsmethode: Objekte die durch Kopieren erzeugt werden.
Zentrale Ablagen: Daten werden bei zentralen Ablagen abgelegt und bei Initialisierung abgeholt.

Generizität: Löcher werden bereits zur Übersetzungszeit befüllt. Was, womit die Löcher gefüllt werden sollen, wird zunächst durch generische Parameter bezeichnet. In allen Lächern, die mit Demselben befüllt werden sollen, stehen auch dieselben generischen Parameter.
Annotationen: optionale Parameter die man zu unterschiedlichsten Sprachkonstrukten hinzufügen kann. Annotationen können sich statisch zur Übersetzungszeit auswirken (@Override) oder dynamisch zur Laufzeit abgefragt werden.
Aspekte: Aspekt spezifiziert eine Menge von Punkten im Programm (Stelle an denen bestimmte Methoden aufgerufen werden) sowie das, was an diesen Stellen passieren soll.(etwa vor oder nach dem Aufruf zusätzlichen Code ausführen oder den Aufruf durch einen anderen ersetzen). Modifizierungen durch Aspekte passieren meist vor der Übersetzung, es ist aber auch während der Laufzeit möglich.

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

Wenn Objekte durch Kopieren erzeugt werden oder zwei zu erzeugende Objekte voneinander abhängen; man kann das später erzeugte Objekt nicht an den Konstruktor des zuerst erzeugten Objekts übergeben.

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

Die Abholung bei Verwendung ist auch für statische Modularisierungseinheiten verwendbar, die bereits zur Übersetzungszeit feststehen.

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

Löcher werden bereits zur Übersetzungszeit gefüllt.

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

Annotationen sind optionale Parameter, die man zu Sprachkonstrukten hinzufügen kann. Annotationen werden von verschiedenen Werkzeugen verwendet, oder aber einfach ignoriert. Sie ähneln also Kommentaren, die aber bei Bedarf auch zur Steuerung des Programmablaufs herangezogen werden können.
Die Löcher, die durch Annotationen befüllt werden, sind im Gegensatz zur Generizität nirgends im Programm festgelegt. Daher ist die Art und Weise, wie die mitgegebenen Informationen zu verwenden sind, ebenso unterschiedlich wie die Anwendungsgebiete.

## 25. Was versteht man unter aspektorientierter Programmierung?

In der aspektorientierten Programmierung kommt man in der Regel ohne Spezifikation von Löchern im Programm aus. Stattdessen fügt man zu einem bestehenden Programm von außen neue Aspekte hinzu.

## 26. Wodurch unterscheiden sich die verschiedenen Formen der Parametrisierung von der Ersetzbarkeit, und warum ist die Ersetzbarkeit in der objektorientierten Programmierung von so zentraler Bedeutung?

Die Änderung einer Modularisierungseinheit macht mit hoher Wahrscheinlichkeit auch Änderungen

an allen Stellen nötig, an denen diese Modularisierungseinheit verwendet wird. Konkret: Wenn die Löcher sich ändern, dann muss sich auch das ändern, was zum Befüllen der Löcher verwendet wird. → behindern die Wartung gewaltig. Nachträgliche Änderungen an den Löchern in Modularisierungseinheit kaum durchführbar.
Praxistaugliche nachträgliche Änderung von Modularisierungseinheiten verspricht der Einsatz von Ersetzbarkeit statt oder zusätzlich zur Parametrisierung. Modularisierungseinheit A ist durch andere Modularisierungseinheit B ersetzbar, wenn ein Austausch von A durch B keinerlei Änderungen an Stellen nach sich zieht, an denen A (bzw. nach der Ersetzung B) verwendet wird.

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

…, wenn ein Austausch von A durch B keinerlei Änderungen an Stellen nach sich zieht, an denen A (bzw. Nach der Ersetzung B) verwendet wird.

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

Ersetzbarkeit zwischen A und B ist dann gegeben, wenn die Schnittstelle von B dasselbe beschreibt wie die von A. Jedoch kann die Schnittstelle von B mehr Details festlegen als die von A, also etwas festlegen, was in A noch offen ist. entsprechende Schnittstellen auf verschiedene Weise spezifizieren: Signatur; Abstraktion realer Welt, Zusicherungen, überprüfbare Protokolle.

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

Schnittstelle einer Modularisierungseinheit die spezifiziert welche Inhalte von außen zugreifbar sind. Diese Inhalte werden nur über ihre Namen und gegebenenfalls die Typen von Parametern und Ergebnissen beschrieben. Bedeutung der Inhalte bleibt offen.

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

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

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

Um Fehler auszuschließen ist eine genaue Beschreibung der erlaubten Erwartungen an eine Modularisierungseinheit nötig. Diese Beschreibung bezieht sich auf die Verwendungsmöglichkeiten aller nach außen sichtbaren Inhalte.
Entspricht Vertrag zwischen den Modularisierungseinheiten (als Server) und ihren Verwendern (als Clients).
klar geregelt wie sich Zusicherungen aus unterschiedlichen Schnittstellen zueinander verhalten müssen, damit Ersetzbarkeit gegeben ist.

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

Wenn die Typen der Operanden mit der Operation zusammenpassen, sind die Typen konsistent. Andernfalls tritt ein Typfehler auf.

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

Es gibt statische und dynamische Typprüfungen. Statische Typprüfungen haben eine Vorteil gegenüber dynamischen: Man kann sicher sein, dass zur Laufzeit keine entsprechenden Typfehler auftreten. Bei dynamischer Prüfung muss man immer mit Typfehlern rechnen. Allerdings ist die statisch verfügbare Information begrenzt. Das heißt man muss manche Aspekte (etwa Array-Grenzen) dynamisch prüfen oder man schränkt die Flexibilität der Sprache so weit ein, dass statische Prüfungen immer ausreichen.

### 34. Was ist der Hauptgrund für den Einsatz statischer Typprüfungen?

Der Hauptgrund für statische Typprüfungen scheint die verbesserte Zuverlässigkeit der Programme zu sein.
Das stimmt zum Teil, aber nicht auf direkte Weise. Typkonsistenz bedeutet ja nicht Fehlerfreiheit, sondern nur die Abwesenheit ganz bestimmter, eher leicht auffindbarer Fehler.

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

Auch bei statischer Typprüfung müssen nicht alle Typen explizit hingeschrieben werden. Viele Typen kann ein Compiler aus der Programmstruktur herleiten; man spricht von Typinferenz.
Zur Verbesserung der Lesbarkeit kann und soll man Typen explizit anschreiben.

Bis zu gewissem Grad erhöhen aber auch statische Typprüfungen alleine die Verständlichkeit, obwohl keine Typen dastehen. Dass hat mit der geringen Flexibilität zu tun , darf keine so komplexen Programmstrukturen verwenden, dass der Compiler bei der Prüfung der Typkonsistenz überfordert wäre. → einfachere Strukturen für Compiler und Mensch besser verständlich.

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

- Einiges bereits in der Sprachdefinition festgelegt; Programme können daran nichts ändern;
- Zeitpunkt der Erstellung von Übersetzungseinheiten werden die meisten wichtigen Entscheidungen getroffen. → braucht viel Flexibilität.
- manche Entscheidungen erst durch Parametrisierung bei Einbindung vorhandener Module, Klassen, Komponenten getroffen. Geht nur wenn eingebundenen Modularisierungseinheiten dafür vorgesehen.
- Compiler getroffene Entscheidungen eher weniger Bedeutung. Alles wichtige bereits im Programmcode festgelegt oder liegt erst zur Laufzeit vor.
- Zur Laufzeit kann man Initialisierungsphase(Einbindung von Modularisierungseinheiten und der Parametrisierung, wo dies erst zur Laufzeit möglich) von der eigentlichen Programmausführung unterscheiden.

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

Je früher Entscheidungen getroffen werden, desto weniger ist zur Laufzeit zu tun. Ohne statisch geprüfte Typen wäre mehrfacher Aufwand nötig: Sogar wenn man weiß, dass die Variable eine ganze Zahl enthält, muss der Compiler eine beliebige Referenz annehmen und zur Laufzeit eine dynamische Typprüfung durchführen. Frühe Entscheidungen sind besser, da auch Typfehler eher entdeckt werden und früher ausgebessert werden können.

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

Wenn man weiß, dass eine Variable vom Typ int ist, braucht man kaum mehr Überlegungen darüber anstellen, welche Werte in der Variablen enthalten sein könnten. Statt auf Spekulationen baut man auf Wissen auf. Um sich auf einen Typ festlegen zu können, muss man voraussehen (also planen), wie bestimmte Programmteile im fertigen Programm verwendet werden. Man wird zuerst jene Typen festlegen, bei denen man kaum Zweifel an der künftigen Verwendung hat. → frühe Entscheidungen daher oft sehr stabil.

## 39. Was ist ein abstrakter Datentyp?

Die Trennung zwischen Innenansicht und Außenansicht einer Modularisierungseinheit (Data−Hiding). Die nach außen hin sichtbaren Inhalte bestimmen die Verwendbarkeit der Modularisierungseinheit. Private Inhalte bleiben bei der Verwendung unbekannt und die gesamte Modularisierungseinheit daher auf gewisse Weise abstrakt. Hinter jeder Modularisierungseinheit steht ein abstraktes Konzept, das im Idealfall eine Analogie in der realen Welt hat.

## 40. Was unterscheidet strukturelle von nominalen Typen?

Struktureller Typ: Typ der Modularisierungseinheit hängt nur von Namen, Parametertypen und Ergebnistypen der nach außen sichtbaren Modulinhalte abhängt
Nominaler Typ: Neben Signatur auch einen eindeutigen Namen. Der Typ eines Objekts entspricht dem Namen der Klasse von der das Objekt erzeugt wurde.

## 41. Warum verwenden Programmiersprachen meist nominale Typen?

Man gibt jeder Modularisierungseinheit einen eigentlich nicht verwendeten Inhalt mit, dessen Name das Konzept dahinter beschreibt. Damit werden gleiche Signaturen für unterschiedliche Konzepte ausgeschlossen. Wegen dieser stets vorhandenen Möglichkeit zur Abstraktion werden in der Theorie überwiegend strukturelle Typen betrachtet.
Beim Programmieren denkt man hauptsächlich in abstrakten Konzepten und selten an Signaturen daher nominale Typen.

## 42. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?

Untertypen werden durch das Ersetzbarkeitsprinzip definiert. Ohne Ersetzbarkeit gibt es keine Untertypen.
Faustregel: Ein Typ U ist Untertyp von Typ T wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.

## 43. Warum kann ein Compiler ohne Unterstützung durch Programmierer nicht

**entscheiden, ob ein nominaler Typ Untertyp eines anderen nominalen Typs ist?**
Abstrakte und daher den Regeln für Ersetzbarkeit nicht zugängliche Konzepte lassen sich ja nicht automatisch vergleichen. Untertypbeziehungen werden über Zusicherungen (über Kommentare) händisch vom Programmierer erstellt, der Compiler kann diese nicht prüfen.

## 44. Welche wichtige Einschränkung gibt es bei Untertypbeziehungen zusammen mit statischer Typprüfung?

Typen von Funktions− bez. Methodenparametern dürfen in Untertypen nicht stärker werden.
Bsp: boolean compare(T x) kann im Untertyp nicht zu boolean compare(U x) überschrieben sein.
Wird in der Praxis häufig benötigt, gibt aber keine Möglichkeit das statisch zu prüfen.

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

Wenn man Einschränkungen auf Typparameter berücksichtigt.
F−gebundene Generizität nutzt Untertypbeziehungen zur Beschreibung von Einschränkungen und wird z.B. in Java und C# eingesetzt.
Higher−Order−Subtyping, auch Matching genannt, geht einen eher direkten Weg und beschreibt Einschränkungen über Untertyp−ähnliche Beziehungen, die wegen Unterschieden in Details aber keine Untertypbeziehungen sind. Dieser Ansatz wird auf unterschiedliche Weise beispielsweise in C++, aber auch in der funktionalen Sprache Haskell verwendet.

## 46. Wie konstruiert man rekursive Datenstrukturen?

Mittels induktiven Konstruktionen; Mengenvereinigung

## 47. Was versteht man unter Fundiertheit rekursiver Datenstrukturen?

Man muss klar zwischen $M0$ (nicht−rekursiv) und der Konstruktion aller
Mi mit i > 0 (rekursiv) unterscheiden, wobei $M0$ nicht leer sein darf. Diese
Eigenschaft nennt man *Fundiertheit*.

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

Typinferenz funktioniert nicht, wenn gleichzeitig(also an der derselben Stelle im Programm) Ersetzbarkeit durch Untertypen verwendet wird.

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

Eine Funktion kann nur aufgerufen werden, wenn der Typ des Arguments mit dem des formalen Parameters übereinstimmt. Dabei wird Information über das Argument an die aufgerufene Funktion propagiert. Entsprechendes gilt auch für das Propagieren von Information von der aufgerufenen Funktion zur Stelle des Aufrufs unter Verwendung des Ergebnistyps und bei der Zuweisung eines Wertes an eine Variable. Genau diese Art des Propagierens von Information funktioniert nicht nur für Typen im herkömmlichen Sinn, sondern für alle statisch bekannten Eigenschaften.

## 50. Erklären Sie folgende Begriffe:

- **Objekt, Klasse, Vererbung**
- **Identität, Zustand, Verhalten, Schnittstelle**
- **deklarierter, statischer und dynamischer Typ**
- **Faktorisierung, Refaktorisierung**
- **Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung**

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

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

**Vererbung:** ermöglicht es,neue Klassen aus bereits existierenden Klassen abzuleiten. Dabei werden nur die Unterschiede zwischen der abgeleiteten Klasse und der Basisklasse, von der abgeleitet wird, angegeben.

**Identität:** Jedes Objekt ist über eindeutige und unveränderliche Identität identifizier− und ansprechbar

**Zustand:** Setzt sich aus den momentanen Variablenbelegungen zusammen. Ist änderbar

**Verhalten:** beschreibt wie sich das Objekt beim Empfang einer Nachricht verhält.

**Schnittstellen:** eines Objektes beschreibt das Verhalten des Objekts in einem Abstraktionsgrad der für Zugriffe von außen notwendig ist.

**deklarierter Typ:** das ist der Typ, mit dem die Variable deklariert wurde.

**dynamischer Typ:** spezifischster Typ, den der in der Variablen gespeicherte Wert hat.

**statischer Typ:** wird vom Compiler statisch ermittelt und liegt irgendwo zwischen deklariertem und dynamischen Typ.

**Faktorisierung:** Zerlegung eines Programms in Einheiten mit zusammengehörigen Eigenschaften.

**Refaktorisierung**: die Faktorisierung des Programms ist auf Grund von Änderungen oder schlechter Planung nicht mehr gut es muss die Zerlegung in Klassen und Objekte geändert werden.

**Verantwortlichkeiten:** gehören zu einer Klasse; Können durch drei Ws beschrieben werden:

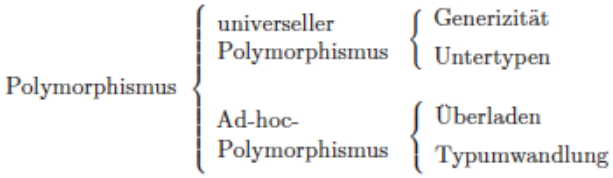
- „Was ich weiß“ - Beschreibung des Zustands des Objekte
- „was ich mache“ - Verhalten der Objekte
- „wen ich kenne“ - sichtbare Objekte, Klassen etc.

**Klassenzusammenhalt:** versteht man den Grad der Beziehung zwischen den Verantwortlichkeiten der Klasse. Zusammenhalt ist hoch, wenn alle Variablen und Methoden eng zusammenarbeiten und durch den Namen der Klasse gut beschrieben sind. Klassenzusammenhalt soll hoch sein und Objektkopplung schwach.

**Objektkopplung:** Abhängigkeit der Objekte voneinander. Objektkopplung ist stark wenn:

- viele Methoden und Variablen nach außen sichtbar
- im laufenden System Nachrichten und Variablenzugriffe zwischen unterschiedlichen Objekten häufig auftreten.
- und die Anzahl der Parameter dieser Methoden groß ist.

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




In der objektorientierten Programmierung sind Untertypen von überragender Bedeutung, die anderen Arten des Polymorphismus existieren eher nebenbei. Daher nennt man alles, was mit Untertypen zu tun hat, oft auch objektorientierten Polymorphismus oder nur kurz Polymorphismus.

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

Zwei durch verschiedene Variablen referenzierte Objekte sind *identisch* wenn es sich um ein und dasselbe Objekt handelt. Zwei Objekte sind *gleich* wenn sie denselben Zustand und dasselbe Verhalten haben, auch wenn sie nicht identisch sind. In diesem Fall ist ein Objekt eine *Kopie* des anderen.
Zustandsgleichheit: Wenn die Parameter gleich sind (wird mit equals verglichen)
Ident: Wenn es sich um das identische Objekt, also das selbe Abbild im Speicher, handelt (wird mit == verglichen)

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

Kapselung: Zusammenfügen von Daten und Routinen zu einer Einheit, sodass Routinen ohne die Daten nicht ausführbar sind. Bedeutung der Daten oft nur den Routinen bekannt.
Data hiding: Objekt wird als Grey− oder Blackbox gesehen. Man kennt nur die Schnittstellen, hat aber wenig Vorstellung darüber, was im inneren des Objektes abläuft.
Datenabstraktion: Kapselung zusammen mit Data hiding. Bsp: Datentypen (In welcher

Datenstruktur die Daten intern gespeichert sind, ist ohne Bedeutung)

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

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

### 55. Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig?

Aufgrund der hohen Code−Wiederverwendung. Ohne Ersetzbarkeit keine Untertypen.

### 56. Warum ist gute Wartbarkeit so wichtig?

Da Wartungskosten ca. 70 % der Gesamtkosten ausmachen. Gute Wartbarkeit erspart Unmengen an Geld!

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

Der Klassenzusammenhang soll hoch sein & Die Objektkopplung soll schwach sein
gute Faktorisierung, Wahrscheinlichkeit geringer, dass bei Programmänderung auch die Zerlegung in Klassen und Objekte geändert werden muss.

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

Á Programme: meistens darauf hin entwickelt häufig (wieder) verwendet zu werden. Zahlt sich dadurch erst aus großen Aufwand in die Entwicklung zu stecken.
Á Daten: Daten in Datenbanken und Dateien wiederverwendet. Oft längere Lebensdauer als Programme die sie benötigen oder manipulieren.
Á Erfahrungen: häufig unterschätzt die Wiederverwendung von Konzepten und Ideen in Form von Erfahrung. können zwischen sehr unterschiedlichen Projekten ausgetauscht werden.
Á Code: Viele Konzepte wie zum Beispiel Untertypen, Vererbung und Generizität wurden im Hinblick auf Wiederverwendung von Code entwickelt. Kann mehrere Arten unterscheiden:

- Bibliotheken: einige Klassen in Klassenbibliotheken werden sehr häufig wiederverwendet. wenige, relativ einfache Klassen kommen für die Aufnahme in Bibliotheken in Frage.
- Projektinterne Codewiederverwendung: hochspezialisierte Programmteile sind nur innerhalb eines Projekts in unterschiedlichen Programmversionen wiederverwendbar. wegen Komplexität erspart bereits eine einzige Wiederverwendung viel Arbeit.
- Programminterne Wiederverwendung: Code in einem Programm kann zu unterschiedlichen Zwecken oft wiederholt ausgeführt werden. Durch den Einsatz eines Programmteils in mehreren Aufgaben wird das Programm einfacher und leichter wartbar.

Faustregel: Code−Wiederverwendung erfordert beträchtliche Investitionen in die Wiederverwendbarkeit. Man soll diese tätigen, wenn ein tatsächlicher Bedarf absehbar ist.

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

Refaktorisierungen ermöglichen das Hinführen des Projektes auf ein stabiles gut faktorisiertes Design.
Gute Faktorisierung => starken Klassenzusammenhalt => gut abgeschlossene und somit leicht wiederverwendbare Klassen.
Sie ändert die Struktur eines Programms, lässt aber dessen Funktionalität unverändert.
Es wird dabei also nichts hinzugefügt oder weggelassen, und es werden auch keine inhaltlichen Änderungen vorgenommen.
Faustregel: Ein vernünftiges Maß rechtzeitiger Refaktorisierungen führt häufig zu gut faktorisierten Programmen und dadurch zu stabilen Klassen die für Vererbung und Untertypbeziehungen wichtig sind.

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

Wenn Algorithmen zentral im Mittelpunkt stehen ist sie nicht gut geeignet. OOP steht die Datenabstraktion im Mittelpunkt, aber Algorithmen müssen unter Umständen aufwendig auf mehrere Objekte aufgeteilt werden. Das kann den Entwicklungsaufwand von Algorithmen erhöhen

und deren Verständlichkeit verringern.
Faustregel: Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

## Kapitel 2

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

1. Durch das Verwenden von Untertypbeziehungen: Untertypen können oft einen Großteil des Codes ihres Obertypen wieder verwenden.
2. Direkte Code−Wiederverwendung: (einfache Vererbung)

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

Ein Typ U ist Untertyp eines Typs T, wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird. Allgemeine Bedingungen für eine Untertypbeziehung sind zum Beispiel Reflexivität, Transitivität und Antisymmetrie. In Java ist die Grundvoraussetzung für eine Untertypbeziehung, dass der Untertyp eine abgeleitete Klasse des Obertyps darstellt, also mittels extends oder implements entweder direkt oder indirekt vom Obertyp abgeleitet ist. Zudem müssen Konstanten im Untertyp Untertypen von Konstanten im Obertyp sein und Variablen müssen im Untertyp den gleichen Typen wie im Obertyp haben. Auch bei Methoden muss der Ergebnistyp im Untertyp ein Untertyp des Ergebnistyps im Obertyp sein, außerdem muss die Anzahl der formalen Parameter gleich sein und der Typ jedes formalen Parameters im Untertyp ein Obertyp des entsprechenden Parametertypen im Obertyp sein.
Siehe Skriptum: Seite 66- 69

### 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?

Die Bedingungen sind nicht vollständig ausreichend, um Ersetzbarkeit zu garantieren. Vor allem Zusicherungen spielen ebenfalls eine große Rolle, wenn es darum geht, ob ein Typ Unter- bzw Obertyp eines anderen Typs ist – das kann allerdings nur vom Programmierer überprüft werden, der Compiler hat hierauf keinen Einfluss. Gleiches gilt für Vorbedingungen, Nachbedingungen und Invarianten.

- Reflexivität Jeder Typ ist Untertyp von sich selbst
- Transitivität Wenn B gleich Untertyp von A ist und C gleich UT von B ist => C ist UT von A
- Antisymmetrie Wenn A UT von B ist und B UT von A ist => A und B sind gleich

### 4. Was bedeutet Ko−, Kontra− und Invarianz, und für welche Typen in einer Klasse trifft welcher dieser Begriffe zu?

**Kovarianz:** Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp. Zum Beispiel sind deklarierte Typen von Konstanten und von Ergebnissen der Methoden sowie von Ausgangsparametern kovariant. Typen und die betrachteten darin enthaltenen Elementtypen variieren in dieselbe Richtung.

**Kontravarianz:** Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp. Zum Beispiel sind deklarierte Typen von formalen Eingangsparametern kontravariant. Typen und die betrachteten darin enthaltenen Elementtypen variieren in entgegengesetzte Richtungen.

**Invarianz:** Der deklarierte Typ eines Elements im Untertyp ist äquivalent zum deklarierten Typ des entsprechenden Elements im Obertyp. Zum Beispiel sind deklarierte Typen von Variablen und Durchgangsparametern invariant. Die betrachteten in den Typen enthaltenen Elementtypen variieren nicht.

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

Eine Methode, bei der ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt binäre Methode. Die Eigenschaft binär bezieht sich darauf, dass der Name der Klasse in der Methode mindestens zweimal vorkommt – einmal als Typ von this und mindestens einmal als Typ eines
expliziten Parameters. Werden häufig benötigt, sind über Untertypbeziehungen (ohne dynamische Typabfragen und Casts) nicht realisierbar.

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

Man soll Parametertypen vorrauschauend und möglichst allgemein wählen.

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

Weil beim Einfügen von neuen Bedingungen (neuen Anredearten z.B.) alle switch Anweisungen aktualisiert werden müssen. Werden weitere Anreden gebraucht kann man diese leicht durch hinzufügen neuer Klassen einführen. Es sind keine zusätzlichen Änderungen nötig. Insbesondere bleiben die Methodenaufrufe unverändert. Es ist kaum möglich bei großem Programmen die Übersicht zu behalten und die Programmteile konsistent zu halten.

### 8. Welche Rolle spielt dynamisches Binden für die Ersetzbarkeit und Wartbarkeit?

Ohne die Verwendung von dynamischen Binden, müssten an vielen Stellen schwer zu wartende switch und if- Anweisungen eingesetzt werden. Soll am Programm eine Änderung vorgenommen werden, genügt es bei dynamischen Binden oft nur eine Klasse hinzuzufügen. Ohne dynamischen Binden, müssen jedoch oft an vielen Stellen im Programm Änderungen vorgenommen werden. Ohne Dynamisches Binden hingegen müsste man die switch-Anweisung entsprechend erweitern! Auf den ersten Blick mag es scheinen, als ob der konventionelle Ansatz mit switch Anweisung kürzer und auch einfach durch Hinzufügen einer Zeile Šnderbar wäre. Am Beginn der Programmentwicklung trifft das oft auch zu. Leider haben solche switch Anweisungen die Eigenschaft, dass sie sich sehr rasch über das ganze Programm ausbreiten. Dann ist es schwierig, zum Einfügen der neuen Anredeart alle solchen switch-Anweisungen zu finden und noch schwieriger, diese Programmteile über einen längeren Zeitraum konsistent zu halten. Der objektorientierte Ansatz hat dieses Problem nicht, da alles auf die Klasse Adressat und ihre Unterklassen konzentriert ist. Es bleibt auch dann alles konzentriert, wenn zu gibAnredeAus() weitere Methoden hinzukommen.

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

Vorbedingungen (Pre-Conditions): Sind vom Cilent (Methodenaufrufer) zu überprüfen und müssen vor Ausführung der Methode gegeben sein.
Nachbedingungen (Post-Conditions): Geben den Zustand an, welcher nach Aufruf der Methode gegeben sein muss. Werden vom Server sichergestellt
Invarianten (Invariants): Allgemeine Bedingungen, an die sich sowohl Server als auch Client halten müssen
History Contraints: schränken Entwicklung von Objekten im Laufe der Zeit ein. Zwei Arten:
Server-kontrolliert: schränken zeitliche Veränderungen der Variableninhalte eines Objekts ein. zb. Zähler wird immer größer, nie kleiner. Server für Einhaltung verantwortlich.
Client-kontrolliert: Reihenfolge der Methodenaufrufe kann eingeschränkt werden. Clients für Einhaltung verantwortlich.

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

Vorbedingungen (PreConditions): Jede Vorbedingung auf einer Methode in T muss eine Vorbedingung auf der entsprechenden Methode in U implizieren. Vorbedingungen in Untertypen können nur schwächer werdenals entsprechende Vorbedingungen in Obertypen.
Aufrufer der Methode, nur T kennt und nur die Erfüllung der Vorbedingungen in T sicherstellen kann. Wenn Methode aber in U ausgeführt wird, muss Vorbedingung automatisch erfüllt sein, wenn sie in T erfüllt ist. Oder-Verknüpfungen
Nachbedinungen (PostConditions): Nachbedingungen in Untertypen können stärker werden als

entsprechende Nachbedingungen in Obertypen. Aufrufer der Methode kennt nur T verlässt sich auf Erfüllung der Nachbedingungen in T, auch wenn Methode in U ausgeführt wird. Nachbedingung in T muss automatisch erfüllt sein wenn sie in U erfüllt ist.  Und- Verknüpfung
Invariante: Invarianten in Untertypen können stärker, dürfen aber nicht schwächer sein als Invarianten im Obertyp. Client der nur T kennt, sich auf die Erfüllung der Invariante in T verlassen kann, auch wenn tatsächlich ein Objekt von U verwendet wird. Invariante in T muss automatisch erfüllt sein wenn die in U erfüllt ist.
Server-kontrollierter History-Contraint: Prinzip dasselbe wie für Invarianten. Für alle Objektzustände x und y in U und T soll gelten: wenn Server-kontrollierte History-Contraints in T ausschließen, dass ein Objekt von T im Zustand x durch Veränderung im Laufe der Zeit in Zustand y kommt, dann müssen dass auch die server-kontrollierten History-Contraints in U ausschließen. Einschränkungen in T müssen auch auf U gelten.
U kann Entwicklung der Zustände stärker einschränken.
Client-kontrollierte History-Contraints: gilt im Prinzip dasselbe wie für Vorbedingungen, jedoch bezogen auf Einschränkungen in der Reihenfolge von Methodenaufrufen → Trace. Jede entsprechend T erlaubte Aufrufreihenfolge muss auch entsprechend U erlaubt sein. Möglich dass U mehr Aufrufreihenfolgen erlaubt als T. Das durch Client-kontrollierte History-Contraints in T beschriebene Trace-Set muss also eine Teilmenge des durch Client-kontrollierte History-Contriants in U beschriebenen Trace-Sets sein.s

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

Schnittstellen und Typen sollen deshalb stabil bleiben, weil die Änderung eines Typs, der viele Untertypen besitzt, oft weitreichende Konsequenzen haben kann, die auch dazu führen kann, dass alle diese Typen geändert werden müssen bzw das Ersetzbarkeitsprinzip plötzlich nicht mehr gilt. Deswegen sollte man darauf achten, nur von stabilen Typen Untertypen zu bilden, um solche Vorkommnisse zu vermeiden. Insbesondere ist deswegen die Stabilität an der Wurzel besonders wichtig.

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

Der Zustand von Variablen, auf die der Client direkten Schreibzugriff hat, kann nur sehr schwer bis gar nicht vom Server kontrolliert werden, insofern ist hier besondere Vorsicht walten zu lassen, was die Einhaltung von Invarianten betrifft (Verantwortung des Programmierers).

## 13. Wie genau sollen Zusicherungen spezifiziert sein?

Zusicherungen sollen stabil bleiben (besonders wichtig für Zusicherungen in Typen an der Wurzel der Typhierarchie). Zur Verbesserung der Wartbarkeit sollen Zusicherungen keine unnötigen Details festlegen → soll sie nur dort einsetzen wo tatsächlich Informationen benötigt werden. Soll Zusicherungen so einsetzen das der Klassenzusammenhalt hoch ist und die Objektkopplung minimiert wird.

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

Abstrakte Methoden: Methoden, die nur den Methodenkopf und keine Implementierung enthalten. Sie eignen sich gut dazu um in einem Obertyp zu bestimmen, dass in einem Untertyp eine gewisse Methode implementiert werden muss. Dienen zur Schnittstellendefinition im Obertypen!
Abstrakte Klassen: Sind Klassen, von denen keine Instanzen erstellt werden können. Sie eignen sich um einen einheitlichen Obertyp (z.B.: Tier) von mehreren Untertypen (z.B.: Esel, Schwein,…) zu erstellen. Abstrakte Klassen können sowohl abstrakte Methoden, als auch vollständig implementierte Routinen enthalten.

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

Vererbung und Ersetzbarkeitsprinzip ist nicht dasselbe. Während bei der Vererbung primär die direkte Codewiederverwendung im Vordergrund steht, beruht das Ersetzbarkeitsprinzip auf einer klaren Obertyp-Untertyp-Beziehung und verwendet somit eher indirekte

Codewiederverwendung. Der wichtigste Unterschied dabei ist, dass bei einer reinen Vererbungshierarchie im Normalfall keine Untertypbeziehungen bestehen.

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

Man sollte prinzipiell immer darauf achten, dass das Ersetzbarkeitsprinzip erfüllt ist – die Wartung eines Programmes wird dabei um ein Vielfaches erleichtert, außerdem entstehen dadurch auch andere Vorteile wie etwa dynamisches Binden. Das bisschen Codewiederverwendung mehr, das die Vererbung mit sich bringt, steht im Normalfall in keiner Relation zur damit einhergehenden erschwerten Wartung.

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

**Instanzvariable:** auch Objektvariable; sind Variablen, die zu Objekten (= Instanzen einer Klasse) gehören.

**Klassenvariable:** Eine Klassenvariable ist nicht einer bestimmten Instanz einer Klasse sondern der Klasse selbst zugehörig. Sie wird mithilfe des Schlüsselwortes „static“ definiert.

**Statische Methode:** Diese bezieht sich ebenfalls auf die Klasse und nicht auf eine Instanz. Eine statische Methode kennt kein „this“, da sie ja nicht in einer bestimmten Instanz aufgerufen wird.

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

**Geschachtelte Klasse:** Eine statische Klasse innerhalb einer anderen Klasse wird als „geschachtelte Klasse“ bezeichnet.

**Innere Klasse:** Eine innere Klasse gehört immer zu einer Instanz der sie umschließenden Klasse. Innere Klassen dürfen keine statischen Methoden und keine statischen geschachtelten Klassen enthalten, da diese von einer Instanz der äußeren Klasse abhängen würden und dadurch nicht mehr statisch wären.

**Final Klasse:** Kann nicht vererbt werden. kann keine Unterklasse abgeleitet werden.

**Final Methode:** Kann nicht überschrieben werden.

**Paket:** Ein Paket ist eine Zusammenfassung von Klassen.

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

Mehrfachvererbung ist nur bei Interfaces möglich, Einfachvererbung hingegen auch bei abstrakten und normalen Klassen.
Unterklassen können nur eine Oberklasse haben, aber mehrere Interfaces implementieren.

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

Importieren direkt im Code über: import Namespace.Package.Class
Um vorgefertigte Pakete zu importieren um beispielsweise fertige ArrayLists und HashMaps verwenden zu können.
Man kann entweder ein einzelnes Paket, eine einzelne Klasse oder mit Hilfe eines Wildcard-Operators (*) sämtliche Klassen eines Pakets importieren.

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

package paketName; → paketName bezeichnet Name und Pfad des Paketes, zu dem die Klasse in der Queldatei gehört. Ist eine solche Zeile in der Quelldatei vorhanden, muss der Aufruf von javac zur Compilation der Datei oder java zur Ausführung der übersetzten Datei im Dateinamen den Pfad enthalten, der in paketName vorgegeben ist. Diese Zeile stellt also sicher das die Datei ncht einfach aus dem Kontext gerissen und in einem anderen Paket verwendet wird.

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

In Java unterscheidet man zwischen public, protected und private. Alle Methoden, Konstruktoren und Konstanten (in ganz seltenen Fällen Variablen- ist aber verpönt), die man bei der Verwendung der Klasse oder von Objekten der Klasse benötigt, sollen public sein.
Man verwendet private für alles, was nur innerhalb der Klasse verwendet werden soll und außerhalb der Klasse nicht verständlich zu sein braucht. Die Bedeutung von Variablen ist außerhalb der Klasse in der Regel ohnehin nicht verständlich, private daher ideal.

Wenn Methoden und Konstruktoren(gelegentlich auch Variablen) für die Verwendung einer Klasse und ihrer und ihrer Objekte nicht nötig,sind, aber in Unterklassen darauf zugegriffen werden muss, verwendet man am besten protected.

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

Interfaces sind im Prinzip eingeschränkte abstrakte Klassen, in denen alle Methoden abstrakt sind. Interfaces unterstützen im Gegensatz zu abstrakten Klassen auch Mehrfachvererbung. Außerdem dürfen in Interfaces keine normalen Variablen deklariert werden, außer sie werden als „static final“ deklariert. Alle Methoden in einem Interface sind public Instanzmethoden. Wenn möglich, sollte man immer Interfaces abstrakten Klassen vorziehen.
Abstrakte Klassen sind besser geeignet wenn sie als Obertyp fungiert und bereits fertig implementierte Methoden enthält die die Untertypen nutzen können (bessere Wartbarkeit, weniger Code).

## Kapitel 3

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

Generizität ist ein statischer Mechanismus, wo in Klassen und/oder Methoden Typparameter statt konkreten Typen verwendet werden. Man unterscheiden generische Klassen und Methoden. Generizität ermöglicht das Verwenden von Typparametern (z.B.: A), für welche bestimmte Typen(String, Person,..) eingesetzt werden können. Alle Vorkommen dieser Typparameter werden durch den angegebenen Typ ersetzt. Generizität eignet sich besonders gut bei Containerklassen, welche für mehrere Datentypen verwendbar sein sollen.

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

Im Rumpf einer einfachen generischen Klasse oder Methode ist über den Typ, der den Typparameter ersetzt, nichts bekannt. Insbesondere ist nicht bekannt, ob Objekte dieser Typen bestimmte Methoden oder Variablen haben.
Über manche Typparameter benötigt man mehr Information um auf Objekte der entsprechenden typen zugreifen zu können. Gebundene Typparameter liefern diese Information: In Java kann man für jeden Typparameter eine Klasse und beliebig viele Interfaces als Schranken angeben. Nur Untertypen der Schranken dürfen den Typparameter ersetzen. Damit ist statisch bekannt, dass in jedem Objekt des Typs, für den der Typparameter steht, die in den Schranken festgelegten öffentlich sichtbaren Methoden und Variablen verwendbar sind.

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

Man soll Generizität immer verwenden, wenn es mehrere gleich strukturierte Klassen (oder Typen) beziehungsweise Methoden gibt. Typische Beispiele dafür: Containerklassen, wie Listen, Stacks, Hashtabellen, Mengen etc und Methoden, die auf Containerklassen zugreifen z.B. Suchfunktionen. Generizität ist am besten dann einzusetzen, wenn die Verwendung die Wartbarkeit erleichtert. (Containerklassen, Iteratoren,…)
Man schreibt ein Programmstück nur einmal und kennzeichnet Typparameter als solche. Statt einer Kopie verwendet man nur den Namen des Programmstücks zusammen mit den Typen, dien an Stelle der Typparameter zu verwenden sind. Erspart sich also mühsames kopieren des Codes und ersetzen der Typparameter per Hand. → erzeugt Probleme bei der Wartung: nötige Änderungen des kopierten Programmstücks müssen in allen Kopien gemacht werden = erheblicher Mehraufwand.

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

Statische Typsicher bedeutet die Garantie vom Compiler, dass im Programm keine Fehler bezüglich falscher Typumwandlung etc zur Laufzeit auftreten.
Der Compiler überprüft, dass eingefügte Typen dem durch den Typparameter festgelegten Typ entsprechen.
Beispiel: List <Integer> x = new List < Integer > ();
x.add(„hallo“) // Fehler, nicht vom Typ Integer!
x.add(25) //OK

In Java erzeugt der Compiler gar keine Kopien der Programmstücke,
sondern kann durch Typumwandlungen (Casts) ein und denselben Code
für mehrere Zwecke – etwa Listen mit Elementen unterschiedlicher Typen
– verwenden.
Generizität ist mit einigen Einschränkungen auch in dynamischen Typabfragen und Typumwandlungen einsetzbar. Skript Seite 150

## 5. 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?

Prinzipiell unterscheidet man zwischen einer homogenen und einer heterogenen Übersetzung von Generizität. Bei der homogenen Übersetzung, wie sie in Java verwendet wird, wird jede Klasse in genau eine Klasse übersetzt, wobei jeder gebundene Typparameter einfach durch die erste Schranke des Typparameters, und jeder ungebundene Typparameter durch Object ersetzt wird. Wenn eine Methode eine Instanz eines Typparameters zurück gibt, wird der Typ der Instanz nach dem Methodenaufruf dynamisch in den Typ, der den Typparameter ersetzt, umgewandelt. Der Compiler garantiert hierbei die Typsicherheit. Bei der heterogenen Übersetzung wird für jede Verwendung einer generischen Klasse oder Routine mit anderen Typparametern eigener übersetzter Code erzeugt („copy and paste“). Dabei wird jedes Vorkommen eines Typparameters durch die entsprechenden Typen ersetzt.
Die heterogene Übersetzung ist insofern etwas flexibler, da man Eigenschaften von Typen verwenden kann, ohne eine Schranke vorzugeben – bei der Übersetzung wird dann für jeden Typ einzeln überprüft, ob er auch die verwendeten Eigenschaften zur Verfügung stellt. Dafür gibt es allerdings Einbußen bei der Qualität von Fehlermeldungen, da sie sich auf generierten Programmcode beziehen können, den man normalerweise nicht zu sehen bekommt.

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

Eine Wildcard kann für jeden beliebigen Typ stehen, der eine vorgegebene Schranke erfüllt. Beispielsweise sind für den Ausdruck <? extends Polygon> alle Typen verwendbar, die Untertypen von Polygon sind. Das Ganze funktioniert auch in die andere Richtung, mit <? super Polygon> sind sämtliche Obertypen von Polygon erlaubt. Dabei muss man beachten, dass erste Variante nur lesenden (kovarianten), und die zweite nur schreibenden (kontravarianten) Zugriff ermöglicht.

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

Generizität kann manuell simuliert werden in dem die gewünschte Klasse nur Typen vom Typ „Object“ (oder eine andere erste obere Schranke) verwendet. Um mit einer derartigen Simulation zu arbeiten benštigt man aber etliche Typumwandlungen. Allerdings entfällt hier der Vorteil der statischen Typsicherheit. Bei der Verwendung von Generizität werden eventuelle derartige Fehler (In eine Liste mit Typparameter String will man eine Person einfügen) bereits vom Compiler erkannt. Simuliert man Generizität aber nur nach oben beschriebener Art, werden solche Fehler erst zu Laufzeit bemerkt!!

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

siehe Frage 5

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

Der Compiler überprüft, dass eingefügte Typen dem durch den Typparameter festgelegten Typ entsprechen.
Beispiel: List <Integer> x = new List < Integer > ();
x.add(„hallo“) // Fehler, nicht vom Typ Integer!
x.add(25) //OK → also statische Typsicherheit

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

getClass : Liefert den dynamischen Typen der Klasse.
U instanceOf T: Liefert true, wenn U ein Untertyp von T ist, sonst false.

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

Typumwandlungen sind auf Referenzobjekten nur durchführbar, wenn der Ausdruck, dessen deklarierter Typ in einen anderen Typ umgewandelt werden soll, tatsŠchlich den gewünschten Typ – oder einen Untertyp davon – als dynamischen Typ hat oder gleich null ist.

## 12. Welche Gefahren bestehen bei Typumwandlungen?

Dynamische Typumwandlungen können leicht Fehler in einem Programm verdecken und die Wartbarkeit erschweren. Fehler werden oft dadurch verdeckt, dass der deklarierte Typ einer Variablen oder eines formalen Parameters nur mehr wenig mit dem Typ zu tun hat, dessen Instanzen man als Werte erwarten würde. Im günstigsten Fall erhält man dann noch eine Fehlermeldung, es könnte aber auch sein, dass zum Beispiel falsche Daten in eine Datenbank geschrieben werden, da sich der tatsächlich verwendete Typ etwas anders verhält als man es erwarten würde. Solche Fehler sind dann meist nur sehr schwer zu finden. Der Grund hierfür ist meistens die Umgehung der statischen Typsicherheit durch den Compiler.

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

Dynamisches Binden und(oder) die Verwendung von Generizität kann helfen Typabfragen und Typumwandlungen zu vermeiden.
Solche Fehler kann man zum Beispiel durch den Einsatz dynamischen Bindens vermeiden. Dies kann allerdings schwierig sein, wenn der deklarierte Typ eines Objekts zu allgemein ist oder die Klassen, die dem deklarierten Typen entsprechen, nicht erweitert werden können. Auch wenn man auf private Methoden und Variablen zugreifen möchte, kann sich dynamisches Binden als schwierig herausstellen. Eine weitere Hürde ergibt sich, wenn der deklarierte Typ sehr viele Untertypen hat und eine Methode nicht in diesem Obertyp implementiert werden kann – in diesem Fall müssen alle Untertypen diese Methode auf die gleiche Art und Weise implementieren, was wiederum die Wartbarkeit erschwert.

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

Typumwandlungen sind sicher, wenn in einen Obertyp des deklarierten Objekttyps umgewandelt wird, oder davor eine dynamische Typabfrage erfolgt, die sicher stellt, dass das Objekt einen entsprechenden dynamischen Typ hat, oder man das Programmstück so schreibt, als ob man Generizität verwenden würde, dieses Programmstück händisch auf mögliche Typfehler, die bei Verwendung von Generizität zu Tage treten, untersucht und dann die homogene Übersetzung durchführt. Der erste Fall ist sicher, da er nur einen harmlosen up-cast darstellt. Beim zweiten Punkt muss sichergestellt sein, dass es eine sinnvolle Alternative gibt, falls der Typvergleich fehl schlägt (Zusicherungen!). Beim dritten Punkt muss man vor allem darauf achten, dass wirklich jeder (gedachte) Typparameter durch den gleichen Typ ersetzt wird und keine impliziten Untertypbeziehungen vorkommen (beispielsweise wird sowohl List<Integer> als auch List<String> nach einer homogenen Übersetzung einfach zu List, hier besteht dann natürlich große Verwechslungsgefahr).

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

Als kovariante Probleme bezeichnet man solche Probleme, in denen es wünschenswert wäre, kovariante Eingangsparametertypen zu haben. Dies würde jedoch das Ersetzbarkeitsprinzip verletzen (Eingangsparametertypen dürfen nur kontravariant sein). Zur Lösung kovarianter Probleme bieten sich dynamische Typabfragen und Typumwandlungen an. Als binäre Methoden werden solche Methoden bezeichnet, die mindestens einen Parameter haben, der der eigenen Klasse entspricht (deswegen auch binär -> der Parametertyp kommt mindestens zweimal vor). Probleme, die aus binären Methoden heraus entstehen, kann man lösen, indem man die Methode in einer neuen abstrakten Klasse implementiert, die Obertyp der Klasse mit der binären Methode ist.

## 16. Wie unterscheidet sich überschreiben von überladen, und was sind Multimethoden?

Beim Überschreiben wird eine Methode des Obertyps im Untertyp ersetzt, dies setzt jedoch

voraus, dass die Methode im Untertyp exakt die gleiche Signatur wie die Methode im Obertyp hat – ansonsten wird die Methode nämlich nur überladen, das heißt, sie existieren nebeneinander. Für je zwei überladene Methoden gleicher Parameteranzahl soll es zumindest eine Parameterposition geben, an der sich die Typen der Parameter unterscheiden, nicht in Untertyprelation zueinander stehen und auch keinen gemeinsamen Untertyp haben, oder alle Parametertypen der einen Methode sollen Obertypen der Parametertypen der anderen Methode sein, und bei Aufruf der einen Methode soll nichts anderes gemacht werden, als auf die andere Methode zu verzweigen, falls die entsprechenden dynamischen Typen der Argumente dies erlauben. Bei Multimethoden ist die Auswahl der entsprechenden Methode nicht nur vom dynamischen Typ, auf dem die Methode ausgeführt wird, abhängig, sondern auch vom dynamischen Typ der formalen Parameter.

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

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

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

Das Visitor Pattern beruht auf mehrfachem dynamischem Binden (in Java durch wiederholtes einfaches Binden simuliert). Dabei unterscheidet man zwischen sogenannten Visitorklassen und Elementklassen, wobei Visitor- und elementklassen oft gegeneinander austauschbar sind. Elementklassen rufen dabei die Visitorklasse mit sich selbst als Argument auf, in der Visitorklasse wird dann die entsprechende Methode ausgeführt. Der große Nachteil hier besteht darin, dass die Anzahl der benötigten Methoden sehr schnell sehr groß wird. Das Visitor Pattern stellt eine mögliche Lösung für kovariante Probleme dar.

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

1) Unbewusstes Überladen statt überschreiben.
2) Wenn nicht klar entschieden werden kann welche der Methoden aufgerufen werden soll
Ein Beispiel:
Methode(Futter f, Gras g){ };
Methode(Gras g, Futter f){ };
Aufrufer(Gras1, Gras2);
Diese Problematik kann vermieden werden wenn, …
… sich die Parametertypen min. in einer Stelle unterscheiden (an einer Stelle keine Untertypbeziehungen bestehen)
… aller Parametertypen der einen Methode nur Oberklassen der anderen Methode beinhaltet und bei Aufruf der Methode soll nichts anderes gemacht werden als auf die andere Methode zu verzweigen.

## Kapitel 4

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

Ausnahmen sind in Java gewöhnliche Objekte, die über spezielle Mechanismen als Ausnahmen verwendet werden. Alle Instanzen von Throwable sind dafür verwendbar. Praktisch verwendet man nur Instanzen der Unterklassen von Error und Exception, zwei Unterklassen von Throwable.
Unterklassen von Error werden hauptsächlich für vordefinierte, schwerwiegende Ausnahmen des Java−Laufzeitsystems verwendet und deuten auf echte Fehler hin, die während der Programmausführung entdeckt wurden. Es ist praktisch kaum möglich, solche Ausnahmenen abzufangen; ihr Auftreten führt fast immer zur Programmbeendigung. z.B. StackOverflowError
Unterklassen von Exception sind in zwei Bereiche gegliedert: überprüfte Ausnahmen (häufig von uns selbst definiert) stehen im Kopf der Methode nach throws.
und nicht überprüfbare, meist vom System vorgegeben, Objekte von RuntimeException z.B. IndexOutOfBoundsException (Arrayzugriff außerhalb des Indexbereiches).

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

Die im Kopf von Methoden deklarierten Ausnahmetypen sind für das Bestehen von Untertyprelationen relevant. Das Ersetzbarkeitsprinzip verlangt, dass die Ausführung einer

Methode eines Untertyps nur solche Ausnahmen zurückliefern kann, die bei Ausführung der entsprechenden Methode des Obertyps erwartet werden. Daher dürfen Methoden in einer Unterklasse in der throws-Klausel nur Typen anführen, die auch in der entsprechenden throws-Klausel in der Oberklasse stehen. In Unterklassen dürfen Typen von Ausnahmen aber durchaus weggelassen werden.

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

Ausnahmen sollte man aus Gründen der Wartbarkeit nur in echten Ausnahmesituationen und vor allem sparsam einsetzen. Einsatzgebiete sind unter anderem unvorhergesehene Programmabbrüche, kontrolliertes Wiederaufsetzen, Ausstieg aus Sprachkonstrukten, aber auch die Rückgabe alternativer Ergebniswerte. Auf keinen Fall sollte man Ausnahmebehandlung als Ersatz für geschachtelte Typabfragen oder Ähnliches missbrauchen.

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

Mehrere Threads laufen nebeneinander; Programmierung wird aufwendiger; Müssen verhindern dass durch gleichzeitige Zugriffe die aus Variablen gelesenen und in Variablen geschriebenen Werte inkonsistent werden.
Java verwendet dazu sogenannte synchronized-Methoden, die garantieren, dass immer nur ein Thread zur gleichen Zeit auf ein Objekt zugreifen kann. Neben synchronized-Methoden gibt es auch die Möglichkeit, nur einzelne Blöcke innerhalb einer Methode als synchronized zu kennzeichnen.

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

Synchronisation wird benötigt, um sicherzustellen, dass immer nur ein Thread zu einem bestimmten Zeitpunkt ein Objekt bearbeitet. Die Granularität sollte man so wählen, dass eher kleine, logisch konsistente Blöcke entstehen, um einerseits sicherzustellen, dass keine anderen Threads auf ein Objekt verändernd zugreifen, während es schon von einem Thread bearbeitet wird, andererseits aber auch die anderen Threads nicht zu lange warten müssen, bis das Objekt wieder freigegeben wird (Deadlock).

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

Deadlocks: zyklische Abhängigkeiten zwischen zwei oder mehreren Threads; (Thread A wartet auf Thread B und B wiederum wartet auf A)
Vermeidung: Lineare Anordnung aller Objekte im System, Locks dürfen nur in dieser Reihenfolge angefordert werden
Lifelocks: alle arbeiten, behindern sich aber gegenseitig sodass kein Fortschritt entsteht
Starvation: Thread kann nicht auf die benötigte Ressource zugreifen, kann nicht weiterarbeiten
nennt man zusammengefasst: Liveness-Properties

## Kapitel 5

### 1. Erklären Sie folgende Entwurfsmuster und beschreiben Sie jeweils das Anwendungsgebiet, die Struktur, die Eigenschaften und wichtige Details der Implementierung: Decorator, Factory−Method, Iterator, Prototype, Proxy, Singleton, Template−Method, Visitor

Zuerst einmal eine Einteilung:
Erzeugungsmuster: Factory Method, Prototype, Singleton
Strukturelle Entwurfsmuster: Decorator, Proxy
Entwurfsmuster für Verhalten: Iterator, Template−Method, Visitor

#### Factory Method

Der Zweck einer Factory Method, auch Virtual Constructor genannt, ist die Definition einer Schnittstelle für die Objekterzeugung, wobei Unterklassen entscheiden, von welcher Klasse die erzeugten Objekte sein sollen. Die tatsächliche Erzeugung der Objekte wird in Unterklassen verschoben. Generell ist dieses Entwurfsmuster anwendbar, wenn eine Klasse neue Objekte erzeugen soll, deren Klasse aber nicht kennt, oder eine Klasse möchte, dass ihre Unterklassen die Objekte bestimmen, die die Klasse erzeugt, oder Klassen Verantwortlichkeiten an eine von

mehreren Unterklassen delegieren, und man das Wissen, an welche Unterklasse delegiert wird, lokal halten möchte. Factory Methods dienen zudem als Anknüpfungspunkte (hooks) für Unterklassen. Die Erzeugung eines neuen Objekts mittels Factory Method ist fast immer flexibler als die direkte Objekterzeugung. Vor allem wird die Entwicklung von Unterklassen vereinfacht. Zudem können sie parallele Klassenhierarchen, wie etwa die Creator-Hierarchie mit der Product-Hierarchie, verknüpfen. Dies kann unter anderem auch bei kovarianten Problemen hilfreich sein. Die Factory Method wird entweder in der abstrakten Creatorklasse ebenfalls als abstrakte Methode definiert, oder es wird gleich eine Default-Implementierung vorgenommen.

### Prototype

Erzeugt neue Instanzen, indem gegebene Vorlagen (Prototypen) kopiert und bei Bedarf vorher geändert werden.
Wird angewendet, wenn:
Á jedes Objekt einer Klasse nur wenige unterschiedliche Zstände haben kann; einfacher Prototype für jeden Zustand zu erzeugen und zu kopieren als Objekte mit new zu erzeugen und Zustände anzugeben
Á die Klassen von denen Objekte erzeugt werden sollen, erst zur Laufzeit bekannt sind.
Á eine parallele Hierarchie von Creator und Product Klassen (Factory Method) vermieden werden soll
Aufbau: Alle Klassen, welche Prototypen enthalten die kopiert werden sollen sind Untertypen einer möglicherweise abstrakte Klasse, deren Methode clone() von ihnen überschrieben werden muss. Diese Methode gibt eine Kopie des Objektes der Klasse zurück, in der sie aufgerufen wurde.

### Singleton (=einelementige Menge)

Das Entwurfsmuster Singleton sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf diese Instanz. Ein Singleton ist zudem ohne Vererbung erweiterbar. Singletons vermeiden außerdem globale Variablen, unterstützen Vererbung, erlauben unter Umständen auch mehrere Instanzen (wobei sie aber immer die vollständige Kontrolle darüber haben, wieviele Instanzen erzeugt werden können) und sind flexibler als statische Methoden, da diese kaum Änderungen erlauben und dynamisches Binden nicht unterstützen.

### Decorator

Auch Wrapper genannt ermöglicht die dynamische Vergabe von zusätzlichen Verantwortlichkeiten an einzelne Objekte, ohne diese Verantwortlichkeit auf die gesamte Klasse zu übertragen.
Wird angewendet, wenn:
Á Á Man dynamisch Verantwortlichkeiten vergeben, aber auch wieder entziehen will.
Á Á Erweiterungen einer Klasse durch Vererbung unpraktisch sind (z.B. große Zahl an Unterklassen zu vermeiden)s

### Proxy

Ist ein Platzhalter (surrogate) für ein anderes Objekt und kontrolliert den Zugriff auf diesen. Steht zwischen dem Aufrufer und dem eigentlichen Objekt.

**Remote Proxies:** Platzhalter für nicht lokale Objekte, welche in anderen Namensräumen existieren (in einem anderen Netzwerk,…) Sie koordinieren die Kommunikation mit diesen (leiten Nachrichten weiter),…..

**Virtual Proxies:** Wird verwendet, wenn man ein Objekt erst erstellen möchte, wenn man es wirklich braucht (da Erstellung eventuell aufwändig). Bis dahin wird ein Platzhalter verwendet.

**ProtectionProxies:** Kontrollieren den Zugriff auf Objekte. Einsetzbar, wenn je nach Zugreifer und Situation unterschiedliche Zugriffsrechte gelten sollen.

**Smart References:** ersetzen einfache Zeiger. Sie können bei Zugriffen zusätzliche Aktionen ausführen.

### Iterator

Ein Iterator ermöglicht den sequenziellen Zugriff auf die Elemente eines Aggregats.
Bietet Möglichkeiten wie:
Next: liefert nächstes Element
hasNext: true, wenn noch ein Elment vorhanden, sonst false
(eventuell noch previous und hasPrevious)
Wird eingesetzt wenn folgende Eigenschaften gewünscht sind:
Á Á auf den Inhalt eins Aggregats zugreifen zu können, ohne die innere Darstellung offen legen

zu müssen.
Á Á Mehrere überlappende Zugriffe
Á Á Vorhandensein einer einheitlichen Schnittstelle für die Abarbeitung verschiedener Aggregatstrukturen
Interne Iteratoren: kontrollieren selbst, wann die nächste Iteration erfolgt. Externen Iteratoren bestimmt des der Anwender.
Wenn Elemente dazugefügt oder entfernt werden passiert es leicht das Elemente doppelt oder gar nicht abgearbeitet werden. → robuster Iterator verhindert das; braucht viel Erfahrung diese zu schreiben.

### Template Method

Definiert das Grundgerüst eines Algorithmus in einer Operation, überlässt die Implementierung einiger Schritte aber einer Unterklasse. Template-Methods erlauben einer Unterklasse, bestimmte Schritte zu überschreiben, ohne die Struktur des Algorithmus zu ändern.
Wird eingesetzt wenn:
Á Á Den unveränderlichen Teil eines Algorithmus nur einmal implementieren zu müssen. Die veränderbaren Teile aber von Unterklassen bestimmen zu lassen.
Á Á Gemeinsames Verhalten von Unterklassen in einer Oberklasse zusammenfassen zu können
Á Á Erweiterungen in den Unterklassen kontrollieren zu können (beispielsweise durch Template_Methods, die Hooks aufrufen und nur das Überschreiben dieser Hooks in Unterklassen ermöglichen.)

Das **Visitor Pattern** beruht auf mehrfachem dynamischem Binden (in Java durch wiederholtes einfaches Binden simuliert). Dabei unterscheidet man zwischen sogenannten Visitorklassen und Elementklassen, wobei Visitor- und elementklassen oft gegeneinander austauschbar sind. Elementklassen rufen dabei die Visitorklasse mit sich selbst als Argument auf, in der Visitorklasse wird dann die entsprechende Methode ausgeführt. Der große Nachteil hier besteht darin, dass die Anzahl der benötigten Methoden sehr schnell sehr groß wird. Das Visitor Pattern stellt eine mögliche Lösung für kovariante Probleme dar.

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

Man unterscheidet zwischen
interner Iterator: Kontrolliert selbst, wann die nächste Iteration erfolgt. Es ist kein Konstrukt zur Durchmusterung (wie eine Schleife) notwendig.
Dem Iterator wird hier meist nur eine Routine übergeben, welche von diesem auf alle Elemente ausgeführt wird.
externer Iterator: Anwender bestimmt, wann das nächste Element abgearbeitet werden soll.
Der Einsatz ist meist komplizierter als bei internen Iteratoren, dafür die Einsatzmöglichkeiten flexibler!

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

Die Schnittstelle bleibt immer gleich. Auch eventuelle Änderungen innerhalb des Aggregats beeinflussen die Kommunikation nach außen, welche über den Iterator erfolgt nicht.

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

Wenn Iteratoren private Implementierungsdetails des zugehörigen Aggregats verwenden wollen, macht es Sinn, die Iteratoren mittels geschachtelter Klassen innerhalb des Aggregats zu definieren. Dadurch wird aber leider die ohnehin schon starke Abhängigkeit zwischen Aggregat und Iterator noch weiter verstärkt.

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

Beim Verändern eines Aggregats, während es von einem (anderen) Iterator durchlaufen wird kann zu Problemen (Ein Element wird zweimal gelesen, ist doppelt vorhanden,..) führen. Eine einfache Lösung ist es das Aggregat vorher zu kopieren und von einer Kopie zu lesen.
Ein *robuster Iterator* ermöglicht das Verändern eines Aggregats (Zugriff mittels Iterator), während es von einem (anderen) Iterator durchlaufen wird ohne es vorher zu kopieren.

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

**unverändert?**
Factory Method: Anzahl der Klassen nimmt zu, da parallel zur eigentlichen Klassenhierarchie eine parallele Creator− Klassenhierarchie geführt werden muss.
Prototype: hat keine (zumindest keine negativen) Auswirkungen auf die Anzahl der Klassen
Decorator: Verringert die Anzahl der Klassen, da man durch das einfache Erweitern von Objekten mittels Dekoratoren eventuelle Unterklassen einsparen kann. (z.B.: Fenster mit Scrollbar und ohne Scrollbar)
Proxy: Für alle Objekte auf die mit einem Proxy zugegriffen werden soll, sind zusŠtzliche Proxy− Klassen notwendig!

**7. Wird die Anzahl der benštigten Objekte im System bei Verwendung von Factory−Method, Prototype, Decorator und Proxy (gegenüber einem System, das keine Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt sie unverŠndert?**
Factory Method: unverändert (Creator Á abstrakte Klasse)
Prototype: unverändert (in bestehende Objekte integriert)
Decorator: erhöht (Es existiert jeweils ein Objekt mit Erweiterung und eines ohne, viele kleine Objekte)
Proxy: vermindert (Wenn ein Objekt nie benötigt wird, wird es auch nie erzeugt)

**8. Vergleichen Sie Factory−Method mit Prototype. Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?**
Die Factory Method kann bei kovarianten Problemen hilfreich sein, dies geht allerdings mit einer parallelen Klassenhierarchie einher, welche man mit Prototype vermeiden kann. Generell ist die Factory Method wohl eher geeignet, wenn man genau weiß was man will, Prototype hingegen bietet sich in sehr „dynamischen“ Systemen an.

**9. Wo liegen die Probleme in der Implementierung eines so einfachen Entwurfsmusters wie Singleton?**
Sobald man mehrere Alternativen bei der Erzeugung eines Singletons anbieten will, können Probleme auftreten, da sich dies meist nur mit switch-Anweisungen oder Ähnlichem realisieren lässt, was sowohl zu Lasten der Flexibilität als auch der Wartbarkeit geht.

**10. Welche Unterschiede und Ähnlichkeiten gibt es zwischen Decorator und Proxy?**
Ein Proxy kann dieselbe Struktur wie ein Decorator haben. Ein Proxy jedoch ist für die Zugriffssteuerung auf ein Objekt verantwortlich. Er stellt praktisch ein Bindeglied zwischen dem „realen Objekt“ und seinem Surrogat dar.
Ein Decorator erweitert hingegen die Verantwortlichkeiten eines Objekts. Er kann folglich einem Objekt Funktionalitäten hinzufügen, aber auch wieder entziehen.

**11. Welche Probleme kann es beim Erzeugen von Kopien im Prototype geben? Was unterscheidet flache Kopien von tiefen?**
Wenn man zum Beispiel clone() einfach nur rekursiv auf zyklische Strukturen anwendet (um etwa eine tiefe Kopie eines Objekts zu erzeugen), erhält man eine Endlosschleife, die zum Programmabbruch aufgrund von Speichermangel führt. Eine flache Kopie legt nur Referenzen auf die Variablen eines Objekts an (der Wert jeder Variable in der Kopie ist identisch mit dem Wert der entsprechenden Variable im originalen Objekt), während eine tiefe Kopie tatsächlich alles „kopiert“ und man ein komplett neues, unabhängiges Objekt erhält.

**12. Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)**
Dekoratoren eignen sich gut dazu, die Oberfläche beziehungsweise das Erscheinungsbild eines Objekts zu erweitern. Sie sind nicht gut für inhaltliche Erweiterungen geeignet. Auch für Objekte, die von Grund auf umfangreich sind, eignen sich Dekoratoren kaum.

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

Man kann – bei Decorators könnte man etwa zu einem Objekt einen Rahmen mittels eines entsprechenden Decorators hinzufügen, und diesen Rahmen vorher mit Hilfe eines anderen Decorators grün färben. Bei Proxies würde es zum Beispiel Sinn machen, ein Virtual Proxy zu verwenden, um die Erzeugung eines Objekts zu verzögern, und dieses mit einem Protection Proxy zu verknüpfen, um etwaige Zugriffsbeschränkungen zu implementieren.

## 14. Was unterscheidet Hooks von abstrakten Methoden?

Hooks sind Punkte in einer sonst vorgegebenen Implementierung, die von Unterklassen ersetzt werden können. Abstrakte Methoden müssen in jedem Fall überschrieben werden.