

Kapitel 1

1. Erklären Sie folgende Begriffe:

- **Objekt (10):** wichtigstes Konzept; grundlegende Einheit in der Ausführung des Programms; „Kapsel“ die zusammengehörige Variablen und Routinen enthält
- **Klasse (13):** jedes Objekt gehört zu genau einer Klasse, die die Struktur des Objekts (dessen Implementierung) im Detail beschreibt; eine Klasse hat Konstruktoren durch die alle Objekte die zur Klasse gehören erzeugt werden (Instanzen einer Klasse)
- **Vererbung (20):** *Inheritance*; ermöglicht Ableitung neuer Klassen aus bereits existierenden Klassen; anzugeben sind lediglich die Unterschiede; spart Schreibaufwand; vereinfacht Programmänderungen; Erweiterung oder Überschreiben möglich
- **Identität (11):** Eindeutige unveränderliche Kennung eines Objekts mit dem es ansprechbar ist; vereinfacht dargestellt ist sie die Adresse des Objekts im Speicher
- **Zustand (11):** setzt sich zusammen aus den Werten der Variablen im Objekt; änderbar
- **Verhalten (12):** beschreibt wie sich ein Objekt beim Empfangen von Nachrichten verhält, also beim Aufruf einer Routine → Methode; abhängig von Nachricht (Methodennamen + aktuelle Parameter → Argumente) und Zustand des Objekts
- **Schnittstelle (12):** beschreibt Verhalten des Objekts in einem für Zugriffe von außen notwendigen Detaillierungsgrad; mehrere Schnittstellen pro Objekt möglich (für unterschiedliche Sichtweisen); oft nur Köpfe enthalten, manchmal auch Konstanten; Objekt implementiert seine Schnittstelle; jede Schnittstelle kann das Verhalten beliebig vieler Objekte beschreiben; Schnittstellen entsprechen den Typen des Objekts
- **Instanz einer Klasse (13):** „Objekt“; alle Instanzen einer Klasse haben die selben Implementierungen und Schnittstellen aber unterschiedliche Instanzen haben unterschiedliche Identitäten und Variablen (genauer Instanzvariablen) obwohl die Variablen gleichen Namens und Typs sind. Auch die Zustände können sich unterscheiden
- **Instanz einer Schnittstelle ():** Objekt
- **Instanz eines Typs ():** Objekt
- **Deklariertes Typ (17):** Typ, mit dem die Variable deklariert wurde ; existiert nur bei expliziter Typdeklaration
- **Statischer Typ (17):** Wird vom Compiler (statisch) ermittelt; kann spezifischer als deklarierter Typ sein.; kommt in Sprachdefinitionen nicht vor
- **Dynamischer Typ (17):** spezifischste Typ, den der in der Variable gespeicherte Wert tatsächlich hat; oft spezifischer als deklarierte Typen und können sich mit jeder Zuweisung ändern; Compiler kennt sie nur im Spezialfall, wenn dynamische und statische Typen einander stets entsprechen; werden für Typüberprüfung zur Laufzeit verwendet
- **Nachricht (10):** werden von Objekten zur Kommunikation untereinander verwendet.
- **Methode (13):** Routine, die beim Empfang von Nachrichten ausgeführt wird
- **Konstruktor (13):** Routine zur Erzeugung und Initialisierung neuer Objekte; alle Objekte die zur Klasse gehören werden durch Konstruktoren dieser Klasse erzeugt.
- **Faktorisierung (24):** Zusammenfassung zusammengehöriger Eigenschaften und Aspekte des Programms zu Einheiten; Gute Faktorisierung kann die Wartbarkeit (und auch die Lesbarkeit) eines Programms deutlich erhöhen)

- **Refaktorisierung (31):** ändert die Struktur eines Programms, lässt dessen Funktionalität aber unverändert; in früher Phase ohne größere Probleme und Kosten möglich; ein vernünftiges Maß rechtzeitiger Refaktorisierung führt häufig zu gut faktorisierten Programmen.
 - **Verantwortlichkeiten (29):** einer Klasse; „was ich weiß“ – Beschreibung des Zustands der Instanzen; „was ich mache“ – Verhalten der Instanzen; „wen ich kenne“ – sichtbare Objekte, Klassen, etc.
 - **Klassenzusammenhalt (29):** *class coherence*; Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse; nicht einfach messbar aber meist intuitiv fassbar; hoch, wenn alle Variablen und Methoden der Klasse eng zusammenarbeiten und der Klassenname aussagekräftig ist; Erniedrigung des Klassenzusammenhalts durch Entfernen von Methoden oder Variablen bzw. sinnändernde Umbenennung der Klasse; Klassenzusammenhalt soll hoch sein.
 - **Objektkopplung (30):** Abhängigkeiten der Objekte voneinander; stark wenn Anzahl der nach außen sichtbaren Methoden und Variablen groß, Nachrichten und Variablenzugriffe zwischen unterschiedlichen Objekten häufig auftreten, die Anzahl der Parameter dieser Methoden groß ist; Objektkopplung soll schwach sein.
 - **Softwareentwurfsmuster (34):** *Design Pattern*; geben immer wieder auftauchenden Problemstellungen und deren Lösungen Namen, damit die Entwickler einfacher darüber sprechen können; beschreiben welche Eigenschaften man sich von den Lösungen erwarten kann; Wahl der Lösung deren Nachteile am ehesten akzeptabel sind; besteht aus Name, Problemstellung, Lösung, Konsequenzen; nicht zu exzessiv einsetzen, da das zu einem komplexen und undurchsichtigen Programm führen kann.
2. **Welche Arten von Polymorphismus unterscheidet man? Welche davon sind in der objektorientierten Programmierung wichtig? Warum?**
 - a. S.17: Universeller Polymorphismus (Generizität, enthaltender Polymorphismus), Ad-hoc-Polymorphismus (Überladen, Typumwandlung)
 - b. Generizität (parametrischer Polymorphismus), enthaltender Polymorphismus (Subtyping), Überladen, Typumwandlung
 - c. Ohne Polymorphismus ist keine Hierarchie möglich und viele Patterns funktionieren auch nur aufgrund des Polymorphismus
 3. **Wann sind zwei gleiche Objekte identisch, und wann sind zwei identische Objekte gleich?**
 - a. S. 11: Zwei Objekte sind gleich (equals) wenn sie den gleichen Zustand und das gleiche Verhalten haben. Identisch (==) sind zwei gleiche Objekte dann, wenn sie die gleiche Identität haben. Zwei idente Objekte sind damit auch immer gleich.
 4. **Sind Datenabstraktion, Datenkapselung und data hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?**
 - a. **Datenabstraktion (13):** Kapselung + data hiding
 - b. **Datenkapselung (11):** Zusammenfügen von Daten und Routinen zu einer Einheit
 - c. **Data hiding (13):** Verstecken von Daten und Implentierungen; man braucht wenn man das Objekt verwenden will nur eine Schnittstelle des Objekts zu kennen, nicht aber dessen Inhalt.
 5. **Was besagt das Ersetzbarkeitsprinzip?**
 - a. S. 18: Ein Typ U ist ein Untertyp eines Typs T (bzw. T ist ein Obertyp von U) wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird.

- 6. Nennen Sie die Schritte im Softwareentwicklungsprozess entsprechend dem Wasserfallmodell und in zyklischen Modellen.**
 - a. S. 26: Analyse, Entwurf (design), Implementierung, Verifikation, Validierung
- 7. Warum ist gute Wartbarkeit so wichtig?**
 - a. S.24: Wenn eine Software erfolgreich ist und einen langen Lebenszyklus hat machen die Wartungskosten ca. 70% der Gesamtkosten aus. Daher ist es wichtig, dass das Programm gut wartbar ist, was sich in der Einfachheit, Lesbarkeit, Lokalität der Programmänderungen und der Faktorisierung äußert.
- 8. Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?**
 - a. S.30: Der Klassenzusammenhalt sollte hoch und die Objektkopplung schwach sein. Wenn diese Faustregeln erfüllt sind, ist das Programm gut faktorisiert. Dadurch beeinflussen Programmänderungen wahrscheinlich weniger Objekte unnötig.
- 9. Welche Arten von Software kann man wiederverwenden, und welche Rolle spielt jede davon in der Softwareentwicklung?**
 - a. S. 32: Programme, Daten, Erfahrungen, Code, Globale Bibliotheken, Fachspezifische Bibliotheken (projektinterne Wiederverwendung, programminterne Wiederverwendung)
- 10. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?**
 - a. Eine sehr große.
- 11. Nennen Sie die wichtigsten Paradigmen der Programmierung und ihre essentiellen Eigenschaften.**
 - a. **Imperative Programmierung (36):** Prozedural, Objektorientiert; Anweisungen werden in festgelegter Reihenfolge ausgeführt
 - b. **Deklarative Programmierung (37):** Funktional, logikorientiert: Beschreiben Beziehungen zwischen Ausdrücken in einem System. Es gibt keine zustandsändernden Anweisungen
 - c. **Modularisierungseinheiten (39):** abstrakte Datentypen, Module, Komponentenprogrammierung, Generische Programmierung
- 12. Wofür ist die objektorientierte Programmierung gut geeignet, und wofür ist sie nicht gut geeignet?**
 - a. Gut geeignet für Programme, die komplexer als die enthaltenen Algorithmen sind. Schlecht geeignet für Programme wo die Algorithmen sehr komplex sind.

Kapitel 2

1. **In welcher Form kann man durch das Ersetzbarkeitsprinzip Wiederverwendung erzielen?**
 - a. indirekt, z.B. Treiber der ausgebaut wird, stabile Schnittstellen
2. **Unter welchen Bedingungen, die von einem Compiler überprüfbar sind, ist ein Typ im Allgemeinen Untertyp eines anderen Typs? Welche zusätzlichen Bedingungen müssen in Java gelten? (Hinweis: Sehr häufige Prüfungsfrage!)**
 - a. S.45: Konstanten sind kovariant; deklarierte Typen der Variablen sind gleich; Methodenergebnisse sind kovariant, die formalen Parameter der Methoden jedoch kontravariant.
3. **Sind die in Punkt 2 angeschnittenen Bedingungen hinreichend, damit das Ersetzbarkeitsprinzip erfüllt ist? Wenn nicht, was muss noch beachtet werden?**
 - a. S.56: Client-Server-Beziehungen wie Vorbedingungen, Nachbedingungen und Invarianten müssen ebenfalls erfüllt sein.
4. **Welche Rolle spielt dynamisches Binden für die Ersetzbarkeit und Wartbarkeit?**
 - a. S.56: Dynamisches Binden ist switch-Anweisungen und geschachtelten if-Anweisungen stets vorzuziehen. Mit dynamischen Binden erhöht man die Wartbarkeit, da neue Unterklassen leicht hinzugefügt werden können bzw. Änderungen relativ leicht an zentraler Stelle möglich sind.
5. **Welche Arten von Zusicherungen werden unterschieden, und wer ist für deren Einhaltung verantwortlich?**
 - a. S.56: Vorbedingungen, Nachbedingungen, Invarianten. Vorbedingungen müssen vom Client, Nachbedingungen vom Server und Invarianten vom Server aber auch vom Programmierer eingehalten werden.
6. **Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten, damit das Ersetzbarkeitsprinzip erfüllt ist? Warum? (Hinweis: Häufige Prüfungsfrage!)**
 - a. S.62: Vorbedingungen in Untertypen können schwächer, dürfen aber nicht stärker sein weil ein Aufrufer der eventuell nur den Obertyp kennt nur dessen Vorbedingung sicherstellen kann. Nachbedingungen in Untertypen können stärker, dürfen aber nicht schwächer sein weil der Aufrufer der eventuell nur den Obertyp kennt sich darauf verlassen können muss dass die Nachbedingung erfüllt ist Invarianten in Untertypen können stärker, dürfen aber nicht schwächer sein weil der Aufrufer der eventuell nur den Obertyp kennt sich darauf verlassen können muss dass die Invariante erfüllt ist
7. **Warum sollen Schnittstellen und Typen stabil bleiben? Wo ist Stabilität besonders wichtig?**
 - a. S.52: Um die betroffenen Stellen bei einer Programmänderung zu minimieren. Die Stabilität von Schnittstellen an der Wurzel der Typhierarchie ist wichtiger als an den Blättern.
8. **Was ist im Zusammenhang mit allgemein zugänglichen (public) Variablen und Invarianten zu beachten?**
 - a. Die Sicherstellung der Invarianten kann in diesem Fall nicht von der Klasse selber überprüft werden und man muss besondere Vorsicht walten lassen.
9. **Wie genau sollen Zusicherungen spezifiziert sein?**
 - a. S.61: Zusicherungen sollen möglichst stabil sein (an der Wurzel der Typhierarchie besonders wichtig) und keine unnötigen Details festlegen.

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

- a. S.67: Abstrakte Klassen dienen zur Festlegung des Typs. Außerdem fördern sie die direkte Wiederverwendung von Code.

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

- a. S.70: Direkte Vererbung zielt meist nur darauf ab Code zu sparen und fasst dementsprechend zusammen. Beim Ersetzbarkeitsprinzip steht die Untertypbeziehung im Vordergrund.

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

- a. S. 72: Auf die Ersetzbarkeit, weil dadurch die Wartung vereinfacht wird und weitere Möglichkeiten (wie z.B. dynamisches Binden) ermöglicht werden

13. Was bedeuten folgende Begriffe in Java?

- a. **Instanzvariable (78):** Variablen, die zu den Instanzen einer Klasse gehören. Jede Instanz enthält eigene Kopien dieser Variablen.
- b. **Klassenvariablen (78):** Variablen, die nicht zu einer bestimmten Instanz, sondern direkt zur Klasse gehören (static)
- c. **Statische Methode (79):** Durch static gekennzeichnete Methode; gehört ebenfalls zur Klasse und nicht zu einer Instanz; z.B. main-Methode; this ist nicht verwendbar
- d. **Static initializer (79):** „Konstruktor“ für statische Variablen; static { ... }; wird bei erstmaliger Verwendung der Klasse ausgeführt
- e. **Geschachtelte Klasse (80):** statische Klassen innerhalb anderer Klassen
- f. **Innere Klasse (80):** Jede Klasse gehört zu einer Instanz der umschließenden Klasse. Innere Klassen dürfen keine statischen Methoden bzw. geschachtelte Klassen enthalten.
- g. **Final Klasse (83):** Kann nicht vererbt werden.
- h. **Final Methode (83):** kann nicht in Unterklasse überschrieben werden
- i. **Paket (84):** Zusammenfassung zu Einheiten, was Übersicht, Wartung und Sichtbarkeit verbessert.

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

- a. Einfachvererbung: (abstrakte) Klassen; Mehrfachvererbung: Interfaces

15. Welche Arten von import-Deklarationen kann man in Java unterscheiden?

- a. S. 84: einzelnes Paket, einzelne Klasse, alle Klassen eines Pakets (.*)

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

- a. S.86: Public: alles; protected: alles außer Sichtbarkeit in anderem Paket; default: protected ohne Vererbung in anderem Paket; private: nichts
- b. Alle öffentlich benötigten Methoden, Konstanten (evtl. Variablen) public; nur von der Klasse selbst benötigte Methoden, Variablen private; momentan „nicht nötig“, vlt. bei späteren Erweiterungen hilfreich: protected

Kapitel 3

1. **Was ist Generizität? Wozu verwendet man Generizität?**
 - a. S.94: Typparameter; parametrischer Polymorphismus; Man verwendet Generizität anstelle expliziter Typen um den Code übersichtlicher zu halten, sich Schreibarbeit zu sparen etc.; Verwendung vor allem bei Collections-Klassen wie Listen etc.
2. **Was ist gebundene Generizität? Was kann man mit Schranken auf Typparametern machen, was ohne Schranken nicht geht?**
 - a. S.100: Gebundene Typparameter liefern zusätzliche Information und man kann daher davon ausgehen, dass ein eingesetzter Typ bestimmten Vorgaben (eben jenen der Schranke) entspricht.
3. **In welchen Fällen soll man Generizität einsetzen, in welchen nicht?**
 - a. S.105: Generell ist der Einsatz von Generizität immer sinnvoll, wenn er die Wartbarkeit verbessert, wie z.B. in gleich strukturierten Klassen und Routinen oder zum Abfangen erwarteter Änderungen. Z.b. in Collections-Klassen
4. **Was bedeutet statische Typsicherheit?**
 - a. Der Compiler garantiert dass die Typsicherheit gegeben ist.
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?**
 - a. Homogene und heterogene Übersetzung; gebundene und einfache Generizität; Homogene Übersetzung wird in Java verwendet und ist nicht sehr flexibel da man Schranken angeben muss.
6. **Was sind (gebundene) Wildcards als Typen in Java? Wozu kann man sie verwenden?**
 - a. S.103: Das sind Typen die Typparameter ersetzen. Man kann sie verwenden, um Sicherheitsprobleme zu beseitigen und die Generizität näher zu spezifizieren. Man kann damit Untertypbeziehungen Typparametern spezifizieren.
7. **Wie kann man Generizität simulieren? Worauf verzichtet man, wenn man Generizität nur simuliert?**
 - a. S.119: Man kann Generizität durch Verwendung eines Obertyps (z.B. Object) und daraus folgenden Typüberprüfungen und –umwandlungen simulieren. Nur ist das nicht so elegant wie mit Generizität und man verliert die statische Typsicherheit (Typkompatibilität).
8. **Was wird bei der heterogenen bzw. homogenen Übersetzung von Generizität genau gemacht?**
 - a. S.110: Bei der heterogenen Übersetzung (C++) wird für jede Verwendung einer generischen Klasse oder Routine mit anderen Typparametern eigener übersetzter Code erzeugt („Copy and Paste“). Nachteil davon ist eine größere Anzahl übersetzter Klassen und Routinen, Vorteil hingegen ist dass zur Laufzeit keine Typüberprüfungen und –umwandlungen gebraucht werden und auf jede übersetzte Klasse eigene Optimierungen anwendbar sind. Homogene Übersetzung (Java) erzeugt pro generische Klasse wie üblicherweise eine JVM-Klasse. Jeder gebundene Typparameter wird durch die Schranke des Typparameters ersetzt, jeder ungebundene Typparameter durch Object. Gibt eine Methode eine Instanz eines Typparameters zurück, wird der Typ der Instanz nach dem Methodenaufruf

dynamisch in den Typ, der den Typparameter ersetzt, umgewandelt. Im Gegensatz zur simulierten Generizität wird hierbei aber die Typkompatibilität vom Compiler garantiert.

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

- a. S.114: getClass gibt die Klasse des Objekts als Ergebnis zurück. Mit instanceof kann man überprüfen, ob der dynamische Typ eines Referenzobjekts Untertyp eines gegebenen Typs ist.

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

- a. Der deklarierte Typ. Weil es das einzige ist, was der Compiler überprüfen kann.

11. Welche Gefahren bestehen bei Typumwandlungen?

- a. Dass sich der Typ nicht in den entsprechenden umwandeln lässt.

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

- a. Durch dynamisches Binden. Bei kovarianten Problemen.

13. Welche Arten von Typumwandlungen sind sicher? Warum?

- a. S.120: Umwandlungen in einen Obertyp des deklarierten Objekttyps; dynamische Typabfrage davor um entsprechenden dynamischen Typ sicherzustellen; Programmstück so schreiben, als würde man Generizität verwenden, händisch auf mögliche Typfehler untersucht und die homogene Übersetzung durchführen.

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

- a. S.123: Man wünscht sich in der Praxis oft kovariante Eingangsparametertypen (verletzt aber das Ersetzbarkeitsprinzip). Lösung: Typabfragen und Typumwandlungen bzw. besser meiden. Binäre Methoden sind ein Spezialfall kovarianter Probleme. Sie haben mindestens einen formalen Parameter dessen Typ stets gleich der Klasse ist die die Methode enthält. Lösung: Neuen abstrakten Obertyp schreiben und Vergleich etc in abstrakte Methode auslagern.

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

- a. S.128: Beim Überschreiben sind die Eingangsparameter einer Methode genau gleich in Typ und Anzahl. Nur der Rückgabotyp darf ein Untertyp der überschriebenen Methode sein. Beim Überladen unterscheiden sie sich mindestens in einem Kriterium, d.h. sie existieren nebeneinander. Bei Multimethoden werden zur Methodenauswahl generell die dynamischen Typen aller Argumente verwendet, wodurch man dann statt überladener Methoden Multimethoden hätte.

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

- a. S.132: Man kann Multimethoden durch mehrfaches dynamisches Binden simulieren. In Java wird mehrfaches dynamisches Binden durch wiederholtes einfaches Binden simuliert. Ein Problem dass dabei auftreten kann ist dass die Anzahl der benötigten Methoden sehr schnell sehr groß wird.

17. Was ist das Visitor-Entwurfsmuster?

- a. S.133: klassisches Entwurfsmuster; es besteht aus Visitor- (z.B. Futter) und Elementklassen (z.B. Tier); diese sind oft gegeneinander austauschbar; großer Nachteil: die Anzahl der benötigten Methoden wird sehr schnell sehr groß.

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

- a. Wenn man überlädt und die Argumente Untertypen voneinander sind, dann nimmt der Compiler bei der Methodenauswahl immer den spezifischsten Untertypen. Dabei kann man sich leicht irren.

19. Wie werden Ausnahmebehandlungen in Java unterstützt?

- a. Exceptions; try – catch Blöcke; Errors (Virtual Machine); RuntimeExceptions

20. Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?

- a. 137: Ausnahmen dürfen in Methoden von Untertypen in der throws-Klausel nur Typen anführen, die auch in den entsprechenden throws-Klauseln der Oberklasse stehen, wobei aber Typen weggelassen werden können.

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

- a. S.138: Unvorhergesehene Programmabbrüche, Kontrolliertes Wiederaufsetzen, Ausstieg aus Sprachkonstrukten, Rückgabe alternativer Ergebniswerte; nicht als if-Abfrage missbrauchen (while-Schleife).

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

- a. Threads, synchronized

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

- a. Synchronisation wird benötigt um zu garantieren, dass auf einen bestimmten Bereich nie mehr als ein Thread gleichzeitig zugreifen kann. Dies dient zum Verhindern von diversen Problemen. Die Granularität der Synchronisation ist so zu wählen, dass eher kleine, logisch konsistente Blöcke entstehen, in deren Ausführung man vor Veränderungen durch andere Threads geschützt ist

Kapitel 4

1. Erklären Sie folgende Entwurfsmuster und beschreiben Sie jeweils das Anwendungsgebiet, die Struktur, die Eigenschaften und wichtige Details der Implementierung:
 - a. **Factory Method (152)**: Unterklassen erzeugen neue Klassen; doppelte Klassenhierarchie macht das ganze etwas mühsam; Erstellung von „unbekannten“ Untertypen; es gibt eine Factory und für jede Klasse einen Creator und eine konkrete Klasse
 - b. **Prototype (155)**: neue Objekte werden während der Laufzeit durch Clonen erzeugt; einzelne Objekte unterscheiden sich nur sehr gering durch Parameter; wenig Klassen notwendig
 - c. **Singleton (159)**: nur eine Instanz einer Klasse; volle Kontrolle darüber; auch mit mehreren Instanzen möglich
 - d. **Decorator (162)**: Hinzufügen und Wegnehmen von Verantwortlichkeiten zur Laufzeit; man soll sich nicht auf die Objektidentität verlassen; Verwendung wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind (z.B. zu viele Unterklassen); mehr Flexibilität als statische Vererbung; Vermeidung von Überladen von Klassen die in der Hierarchie weit oben stehen; führt zu vielen kleinen Objekten
 - e. **Proxy (165)**: „Zwischenklasse“, um Zugriffe zu regeln bzw. Objekte erst wenn nötig wirklich zu instanzieren
 - f. **Iterator (168)**: ermöglicht sequentiellen Zugriff auf die Elemente eines Aggregats; Unterscheidung interner / externer Iterator
 - g. **Template Method (172)**: Codewiederverwendung z.B. bei Algorithmen; gleicher Teil (z.B. Algorithmus) wird im abstract-Teil implementiert
 - h. **Visitor (133)**: Das Visitor Pattern wird zur Kapselung von Operationen auf Objekten verschiedener Klassen benutzt. Veränderungen an den Operationen können zentral durchgeführt werden wobei die Objekte meistens unangetastet bleiben. Es gibt das Visitor- und das Element-Interface. Letzteres stellt nur eine Methode zur Verfügung, meist `accept(Visitor v)`. Ersteres stellt für jede zu besuchende Klasse eine eigene Methode zur Verfügung. Diese wird in der `accept`-Methode der jeweiligen, Element-implementierenden Klasse aufgerufen. Das Ganze funktioniert nach dem sogenannten Double Dispatching. Beim double dispatching gibt sich das aufrufende Objekt selbst an den Visitor weiter. Dieser wiederum führt dann die entsprechenden 'Funktionen' in der vom Objekt aufgerufenen `visit`-Methode aus. (siehe dazu auch <http://www.developia.de/developia/viewarticle.php?cid=22203> und [http://de.wikipedia.org/wiki/Besucher_\(Entwurfsmuster\)](http://de.wikipedia.org/wiki/Besucher_(Entwurfsmuster)))
2. 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?
 - a. **Factory Method**: erhöht
 - b. **Prototype**: unverändert (evtl. geringfügig erhöht)
 - c. **Decorator**: erhöht
 - d. **Proxy**: erhöht

3. **Wird die Anzahl der benötigten Objekte 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?**
 - a. **Factory Method:** unverändert
 - b. **Prototype:** erhöht
 - c. **Decorator:** erhöht
 - d. **Proxy:** vermindert
4. **Vergleichen Sie Factory Method mit Prototype. Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?**
 - a. Für „dynamische“ Klassen ist Prototype besser; wenn man genau weiß was man braucht dann ist die Factory Method zu bevorzugen.
5. **Welche Unterschiede gibt es zwischen Decorator und Proxy?**
 - a. Proxy spielt nur Ersatz für ein Objekt. Decorator kann auch neue Methoden hinzufügen.
6. **Welche Probleme kann es beim Erzeugen von Kopien im Prototype geben? Was unterscheidet flache Kopien von tiefen?**
 - a. Es kann zu Schleifen kommen. Flache Kopie bedeutet dass nur eine Referenz auf die Variablen des kopierten Objektes angelegt wird anstatt sie komplett zu kopieren.
7. **Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)**
 - a. Gut geeignet für Oberfläche und Erscheinungsbild eines Objekts. Nicht gut geeignet für inhaltliche Erweiterungen und umfangreiche Objekte.
8. **Kann man mehrere Decorators bzw. Proxies hintereinander verketteten? Wozu kann so etwas gut sein?**
 - a. Ja. Decorator: hinzufügen mehrerer Features; Proxy: einmal Zugriffskontrolle, einmal Objektreferenz
9. **Was unterscheidet hooks von abstrakten Methoden?**
 - a. S.173: Hooks sind Operationen mit in „AbstractClass“ definiertem Default-Verhalten, das bei Bedarf in Unterklassen überschrieben oder erweitert werden kann; oft besteht das Default-Verhalten darin, nichts zu tun.
10. **Welche Arten von Iteratoren gibt es, und wofür sind sie geeignet?**
 - a. Interne und externe Iteratoren; externe sind besser geeignet für Vergleiche und man hat mehr Kontrolle über sie; für Implementierungsdetails (z.B. Beziehungen zwischen den Objekten) sind interne besser geeignet
11. **Inwiefern können geschachtelte Klassen bei der Implementierung von Iteratoren hilfreich sein?**
 - a. S.171: Wenn Iteratoren private Implementierungsdetails des Aggregats verwenden, ist es sinnvoller die Iteratorklasse in die Aggregatklasse zu schachteln. Leider verstärkt das die starke Abhängigkeit zwischen Aggregat und Iterator noch stärker.
12. **Was ist ein robuster Iterator? Wozu braucht man Robustheit?**
 - a. S.171: Ein robuster Iterator wird bei Veränderungen des Aggregats während seiner Arbeit nicht beeinflusst. Robustheit braucht man, damit man nicht jedesmal zeitaufwändig das komplette Aggregat kopieren muss, um Probleme zu vermeiden.
13. **Wo liegen die Probleme in der Implementierung eines so einfachen Entwurfsmusters wie Singleton?**

- a. Das Singleton verletzt mehrere Regeln eines guten objektorientierten Designs. Code welcher die Singleton-Klasse benutzt hängt von dieser ab, er ist nicht ohne sie nutzbar, was ihn dazu auch noch schwer zu testen macht. Der Code ist dann nicht gegen ein Interface programmiert, sondern gegen eine konkrete Klasse, und daher fest an das Singleton gekoppelt, anstatt einer gewünschten losen Kopplung zwischen beiden Klassen. (siehe <http://www.phpfatesme.com/blog/softwaretechnik/das-singleton-richtig-implementiert/>)