Beliebte Prüfungsfragen ausgearbeitet

1. Frage: Entwurfsmuster (Design-Patterns)

- dienen der Wiederverwendung kollektiver Erfahrung
- zur effizienten Erstellung und Wartung von Software ist Erfahrung wichtig

Elemente des Entwurfsmusters:

- **Name:** geeignete Entwurfsmusternamen
- **Problemstellung:** Beschreibung des Problems, Einsatzmöglichkeiten
- **Lösung:** allgemein gehaltene Lösungsbeschreibung
- **Konsequenzen:** Eigenschaften der Lösung, Vor- und Nachteile

Erzeugungsmuster: Factory Method, Prototype, Singleton
Strukturelle Entwurfsmuster: Decorator, Proxy
Entwurfsmuster für Verhalten: Iterator, Template-Method, Visitor

Factory Method (Virtual Constructor)

- Definition einer Schnittstelle für Objekterzeugung

Ermöglicht ein Objekt durch Aufruf einer Methode anstatt durch direkten Aufruf eines Konstruktors zu erzeugen. Die Objekterzeugung wird in Unterklassen verlagert. Für jede Klasse, welche durch eine Factory erzeugt werden soll, muss eine eigene Creator-Klasse erstellt werden. All jene Klassen sind Untertypen einer allgemeineren Creator-Klasse. Somit existiert parallel zur eigentlichen Klassen-Hierarchie eine eigene Creator-Klassen-Hierarchie.

Anwendung:

- Eine Klasse Objekte erzeugen soll, deren Klasse aber nicht kennt
- Wenn eine Klasse möchte, dass ihre Unterklassen die Art der Objekte bestimmen, welche die Klasse erzeugt
- Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren (Wissen um Unterklasse soll lokal bleiben)
- Wenn Allokation und Freigabe von Objekten zentral in einer Klasse verwaltet werden soll

Eigenschaften:

- Anknüpfungspunkte(Hooks) für Unterklassen
- Flexibel, Entwicklung von Unterklassen vereinfacht
- Verknüpfung paralleler Klassenhierarchien (Creator- mit Product-Hierarchie)
- oft große Anzahl an Unterklassen nötig (in C++ könnte man die Zahl der Klassen durch Verwendung von Templates klein halten)

Prototype

- Prototyp-Objekt spezifiziert Art eines neuen Objekts

Erzeugt neue Objekte, indem gegebene Vorlagen (Prototypen) kopiert und bei Bedarf vorher geändert werden.

Anwendung:

- wenn Klasse des neuen Objekts erst zur Laufzeit bekannt
- Vermeidung von Creator- parallel zu Product-Hierarchie (Factory-Method)
- Die zu erzeugenden Klassen sich nur in wenigen Parametern unterscheiden;
Objekte nur wenige unterschiedliche Zustände annehmen
(einfacher für jeden möglichen Zustand einen Prototypen zu erzeugen und diesen zu kopieren als durch new zu erzeugen (Konstruktoraufruf) und dabei die passenden Zustände anzugeben)

Eigenschaften:

- versteckt Product-Klassen (aus Factory-Method) vor Anwendern (Client)
- reduziert damit Anzahl der Klassen die Anwender kennen müssen
(geänderte Product-Klassen beeinflussen Anwender daher nicht)
- können zur Laufzeit jederzeit hinzugefügt oder entnommen werden
(Klassenstruktur darf sich nicht ändern)
- Zustand eines Prototyps kann sich jederzeit ändern, Klassen nicht
- in hochdynamischen Systemen: Verhalten durch Objektkomposition (Zusammensetzten neuer Objekte aus mehreren bestehenden) statt Klassendefinition festlegbar
- vermeidet große Anzahl an Unterklassen
- erlaubt dynamische Konfiguration von Programmen auch in Sprachen wie C++

Notwendig, dass jede konkrete Unterklasse Methode „clone“ implementiert
(oft schwer, Klassen aus Klassenbibliotheken, zyklische Referenzen)
- Deshalb „clone“ in Java für flache Kopien in Object vordefiniert
(verwendbar wenn Cloneable implementiert)
- Erzeugen tiefer Kopien schwierig — zyklische Strukturen
- Prototyp-Manager zur Verwaltung der Prototypen
(kleine Datenbanken in denen nach geeigneten Prototypen gesucht wird)
- „clone“ hat oft keine geeigneten Parameter, Initialisierungsmethoden nötig
- sinnvoll in statischen Sprachen wie Java und C++
- von dynamischen Sprachen direkt unterstützt

Singleton

- Sichert zu dass Klasse nur 1 Instanz hat und erlaubt globalen Zugriff auf dieses Objekt

Anwendbar:

- Wenn es genau ein Objekt einer Klasse geben soll das global zugreifbar ist
- Wenn die Klasse durch Vererbung erweiterbar sein soll, Anwender die erweiterte Klasse ohne Änderungen verwenden können

Singleton besteht nur aus einer gleichnahmigen Klasse mit einer statischen Methode „instance“, welche das einzige Objekt der Klasse zurückgibt.

Eigenschaften:

- Erlauben kontrollierten Zugriff auf das einzige Objekt
- Vermeiden durch Verzicht auf globale Variablen unnötige Namen und die uanngenehemen Eigenschaften globaler Variablen
- Unterstützen Vererbung
- Verhindern, dass irgendwo Instanzen außerhalb der Kontrolle der Klasse erzeugt werden
- Erlauben auch mehrere Instanzen

- Flexibler als statische Methoden, da statische Methoden kaum Änderungen erlauben und dynamisches Binden nicht unterstützen

Decorator (Wrapper)
- ermöglicht die dynamische Vergabe von zusätzlichen Verantwortlichkeiten
an einzelne Objekte, ohne diese Verantwortlichkeit auf die gesamte Klasse zu übertragen. Alternative zur Vererbung
Anwendung:
- dynamisches Hinzufügen von Verantwortlichkeiten ohne Beeinflussung anderer Objekte
- Man dynamisch Verantwortlichkeiten vergeben, aber auch wieder entziehen will.
- Erweiterungen einer Klasse durch Vererbung unpraktisch sind (Vermeidung einer großen Zahl an Unterklassen, oder keine Vererbung unterstützt wird (final Klassen))

Eigenschaften:
- mehr Flexibilität als statische Vererbung (Verantwortlichkeiten dynamisch dazu oder weg)
- vermeidet Klassen, die weit oben in der Klassenhierarchie mit vielen Methoden überladen sind
- Instanzen von „Decorator“ haben andere Identitäten als Instanzen von „ConcreteComponent“ -> nicht auf Objektidentität verlassen
- oft viele kleine Objekte (einfach konfigurierbar, aber schwer wartbar)

abstrakte Klasse „Decorator“ nicht nötig, aber sinnvoll
„Component“ so klein wie möglich halten
gut geeignet zur Erweiterung der Oberfläche,
schlecht geeignet für inhaltliche Erweiterungen und für umfangreiche Objekte

Proxy (Surrogate)
Als Platzhalter für Objekt verwendet (und kontrolliert den Zugriff darauf),
welches erst dann erzeugt wird wenn man es wirklich braucht (bsp: umfangreiche Daten laden), falls nie darauf zugegriffen wird erspart man sich den Aufwand der Objekterzeugung,
Jedes Platzhalterelement enthält einen Zeiger auf das eigentliche Objekt (falls es existiert) und leitet Nachrichten weiter

Anwendung:
- Remote Proxies: Platzhalter für Objekte, die in anderen Namensräumen existieren (auf Festplatten, anderen Rechnern), Nachrichten an Objekte werden von den Proxies über komplexe Kommunikationskanäle weitergeleitet
- Virtual Proxies: erzeugen Objekte bei Bedarf (da Erzeugung eines Objekts aufwendig sein kann), erst erzeugt wenn wirklich nötig
- Protection Proxies: kontrollieren Zugriffe auf Objekte, sinnvoll wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen
- Smart References: ersetzen einfache Zeiger, können bei Zugriffen zusätzliche Aktionen ausführen
  Verwendung: Mitzählen von Referenzen (damit das Objekt entfernt werden kann wenn es keine Referenz mehr darauf gibt),
  Laden persistenter Objekte in den Speicher, wenn das erste Mal darauf zugegriffen wird (ca wie bei virtuellem Proxy),
  zusichern, dass während des Zugriffs auf das Objekt kein gleichzeitiger Zugriff durch einen anderen Thread erfolgt (Lock setzen)

Klasse Proxy: verwaltet Referenz auf „RealSubject“, ersetzt eigentliches Objekt (Ersetzbarkeit), kontrolliert Zugriffe auf eigentliches Objekt, weitere Verantwortlichkeiten von Art abhängig

- mehrere Proxies können verkettet sein
- Darstellung nicht existierender Objekte, Objekte anderer Dateien
- manchmal kennt „Proxy“ nur „Subject“
- selbe Struktur wie Decorator möglich, aber anderer Zweck
  (Decorator erweitert ein Objekt um zusätzliche Verantwortlichkeiten, Proxy kontrolliert den Zugriff auf das Objekt)

Iterator (Cursor) ermöglicht sequentiellen Zugriff auf die Elemente eines Aggregats (Sammlung von Elementen (z.B. Collection)) ohne die innere Darstellung des Aggregats offenzulegen.

Verwendbar:
- Auf Inhalt eines Aggregats zuzugreifen ohne innere Darstellung darzulegen
- Mehrere (gleichzeitige bzw. überlappende) Abarbeitungen der Elemente in einem Aggregat zu ermöglichen
- Um eine einheitliche Schnittstelle für Abarbeitung versch. Aggregatstrukturen zu haben (Polymorphe Iterationen zu unterstützen)

3 wichtige Eigenschaften:
- Unterstützen verschiedene Varianten in der Abarbeitung von Aggregaten
  (mehrere Iteratorklassen pro Aggregatklasse)
- Sie vereinfachen die Schnittstelle von „Aggregate“ da Zugriffsmöglichkeiten die über Iteratoren bereitgestellt werden durch die Schnittstelle von „Aggregate“ nicht unterstützt werden müssen
  es kann zusätzliche Methoden geben die das Aggregat weiter vereinfachen
- Auf einem Aggregat können gleichzeitig mehrere Abarbeitungen stattfinden, da jeder Iterator selbst den aktuellen Abarbeitungszustand verwaltet.

Interne: kontrollieren selbst wann die nächste Iteration erfolgt
Es ist kein Konstrukt zur Durchmusterung (Schleife) notwendig
(z.B- forEach in Map), (oft einfacher zu verwenden), (funktionale Programmierung: gute Unterstützung für die Erzeugung und Übergabe von Routinen an Iteratoren)
(interne besser wenn komplexe Beziehungen zw Elementen zu Berücksichtigen sind), (wenn Beziehungen zwischen Elementen bei der Abarbeitung benötigt werden)

Externe: Anwender bestimmen wann sie das nächste Element abarbeiten
(flexibler, leicht zwei Aggregate miteinander zu vergleichen)
(in OOP hauptsächlich externe (kann sich aber ändern))

Aggregatänderungen während es von einem Iterator durchwandert wird können gefährlich sein, denn wenn Elemente dazugefügt oder entfernt werden passiert es schnell das Elemente nicht oder doppelt abgearbeitet werden (dafür gibt es: *Robusten Iterator*)

Eine einfache Lösung ist es das Aggregat vorher zu kopieren und von einer Kopie zu lesen.

auch auf leeren Aggregaten brauchbar

Template-Method

- Definiert das Grundgerüst eines Algorithmus in einer Operation, überlässt die Implementierung einiger Schritte aber einer Unterklasse
- Erlaubt einer Unterklasse bestimmte Schritte zu überschreiben ohne die Struktur des Algorithmus zu ändern

Anwendung:

- Um den unveränderlichen Teil eines Algorithmus nur einmal implementieren zu müssen, die veränderbaren Teile aber von Unterklassen bestimmen zu lassen
- Wenn gemeinsames Verhalten von Unterklassen in einer Oberklasse zusammengefasst werden soll (um Duplikate im Code zu vermeiden)
- Um Erweiterungen in den Unterklassen kontrollieren zu können (Hooks(Anknüpfungspunkte))

Eigenschaften:

- Wiederverwendung von Programmcode (in Klassenbibliotheken und Frameworks sinnvoll)
- Führen zur umgekehrten Kontrollstruktur- „Hollywood Prinzip“, Oberklasse ruft die Methoden der Unterklassen auf (nicht wie in den meisten Fällen umgekehrt)
- Sie ruft meist nur eine von mehreren Arten von Operationen auf:
    - Factory Methods
    - Hooks
    - abstrakte primitive Operationen die einzelne Schritte im Algorithmus ausführen
    - konkrete Operationen in ConcreteClass
    - konkrete Operationen in Abstract Class, Operationen die allgemein für Unterklassen sinnvoll

  Es ist wichtig dass genau spezifiziert ist, welche Operationen Hooks (dürfen überschrieben werden) und welche abstrakt sind (müssen überschrieben werden).
- Ziel bei der Entwicklung von Templates: Anzahl der primitiven Operationen möglichst klein halten

Visitor Pattern

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!
Visitorklassen und Elementklassen (sind oft gegeneinander austauschbar)

2. Frage) Kovarianz, Kontravarianz, Invarianz

Kovarianz: Typ von Konstante, Ergebnis, Ausgangsparameter
Typ von Element im Untertyp ist Untertyp des Elementtyps im Obertyp

Wann und warum Kovarianz?

Ersetzbarkeit bei Lesezugriff auf Konstante, Ergebnis, Ausgangsparameter

Kontravarianz: Typ von Eingangsparameter
Typ von Element im Untertyp ist Obertyp des Elementtyps im Obertyp
Wann und Warum Kontravarianz?
Ersetzbarkeit bei Schreibzugriff auf Eingangsparameter

Invarianz: Typ von Variable, Durchgangsparameter
Typ von Element im Untertyp ist äquivalent zu Elementtyp im Obertyp
Wann und warum Invarianz?
Ersetzbarkeit bei schreibendem und lesendem Zugriff
Sowohl Kovarianz als auch Kontravarianz gefordert
In Java alle Typen Invariant, außer Rückgabewerte, die sind kovariant

3.Frage) Ersetzbarkeitsprinzip

Ersetzbarkeitsprinzip:
Untertypbeziehungen sind durch das Ersetzbarkeitsprinzip definiert:
Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.
Man benötigt das Ersetzbarkeitsprinzip für:

- Den Aufruf einer Methode mit einem Argument, dessen Typ ein Untertyp des Typs des entsprechenden formalen Parameters ist
  (Argument dessen Typ Untertyp des formalen Parameters)
- Für die Zuweisung eines Objekts an eine Variable, wobei der Typ des Objekts ein Untertyp des deklarierten Typs der Variable ist
  (Zuweisung x = y; wobei Typ von x Untertyp des deklarierten Typs von y)

<u>Typ U ist Untertyp von T wenn:</u>

- Für jede Konstante in T (Typ A) gibt es eine entsprechende Konstante in U (Typ B), wobei B Untertyp von A
- Für jede Variable in T (Typ A) gibt es eine entsprechende Variable in U (Typ B), wobei A und B äquivalent sind
- Für jede Methode in T gibt es eine entsprechende gleichnamige Methode in U wobei:
  - Ergebnistyp der Methode in U ein Untertyp des Ergebnistyps der Methode in T ist
  - Anzahl der formalen Parameter und Parameterarten gleich
  - Parametertypen passen zueinander
  - Methode U wirft nicht mehr Exceptions als die in T

Eingangsparameter: Argument wandert von Aufrufer zu aufgereufenen Methode,
A Untertyp von B
Ausgangsparameter: Ergebnis wandert von aufgerufener Methode zu Aufrufer
B Untertyp von A
Durchgangsparameter: gleichzeitiger Ein und Ausgangsparameter
A und B äquivalent

In welchen Formen kann man durch das Ersetzbarkeitsprinzip Wiederverwendung erzielen?

1) Verwendung von Untertypbeziehungen: Untertypen können oft einen Großteil des Codes ihres Obertypen verwenden
2) Direkte Code-Wiederverwendung: einfache Vererbung

Wann ist ein struktureller Typ Untertyp eines anderen strukturellen Typs? Welche Regeln müssen dabei erfüllt sein? Welche zusätzlichen Bedingungen gelten für nominale Typen bzw. in Java?

- In Java muss zusätzlich über „extends“ oder “implements“ abgeleitet werden
- In Java sind alle Typen Invariant außer Rückgabewerte von Methoden können kovariant sein. Sonst wird die Methode überladen, statt überschrieben

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

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.
Sie sind über Untertypbeziehungen nicht realisierbar
Kovariante Eingangsparameter und binäre Methoden widersprechen dem Ersetzbarkeitsprinzip.

4. Frage) Zusicherungen (Assertions)

Beschreibung des Verhaltens eines Objekts, gesehen als Vertrag zwischen Server und Client

Vorbedingung(Pre-Conditions): Sind vom Cilent (Methodenaufrufer) zu überprüfen und müssen vor Ausführung der Methode gegeben sein.
Sie beschreiben hauptsächlich welche Eigenschaften die Argumente mit denen die Methode aufgerufen wird, erfüllt sein müssen
(z.B. Array von aufsteigend sortierten Zahlen von 0-90)

Nachbedingung (Post-Conditions): Müssen vom Server sichergestellt werden. Geben den Zustand an, welcher nach Aufruf der Methode gegeben sein muss. Sie beschreiben die Eigenschaften des Methodenergebnisses und Änderungen/Eigenschaften des Objektzustandes.
(z.B Methode zum Einfügen von Elementen: am Ende muss das Element in der Menge vorhanden sein)

Invarianten (Invariants): Server erfüllt Invarianten des Objekts,
direkte Schreibzugriffe von Clients auf Variablen des Servers kann er aber nicht kontrollieren, dafür sind Clients.

History-Constraint: Bedingungen die die Entwicklung von Objekten im Laufe der Zeit einschränken

Server-kontrolliert: ähneln Invarianten, schränken aber zeitliche Veränderungen der Variableninhalte eines Objekts ein.
(z.B. Zählervariable, kann nur größer werden)

Client-kontrolliert: Reihenfolge von Methodenaufrufen einschränken,
nur Clients können Methoden aufrufen und ihre Reihenfolge bestimmen

Zusicherungen in Untertypen

Vorbedingung: Vorbedingungen in Untertypen können schwächer sein, aber niemals stärker als im Obertypen

Nachbedingungen: Nachbedingungen im Untertyp können stärker aber nicht schwächer sein als im Obertypen

Invarianten: Invarianten in Untertypen können stärker, aber nicht schwächer sein

Server kontrollierte History Constraints: können in Untertypen stärker aber nicht schwächer sein

Client kontrollierte History Contraints: können schwächer sein, aber niemals stärker als im Obertypen

5. Frage) Klassenzusammenhalt (class cohesion), Objektkopplung, Verantwortlichkeiten

= Grad der Beziehung zwischen Verantwortlichkeiten der Klasse

Hoch wenn: Variablen und Methoden eng miteinander zusammenarbeiten und durch Klassenname gut beschrieben

Klassenzusammenhalt soll hoch sein: Hinweis auf gute Faktorisierung, verringert Wahrscheinlichkeit für nötige Änderungen

(Faktorisierung: Zerlegung eines Programms in Einheiten mit zusammengehörigen Eigenschaften)

Wie testet man auf Klassenzusammenhalt?

Schlechte Methode: Methoden entfernen und schauen ob noch alles funktioniert

Besser: Methoden umbenennen

Verantwortlichkeiten einer Klasse:

Was ich weiß (Zustand der Objekte)

Was ich mache (Verhalten der Objekte)

Wen ich kenne (sichtbare Objekte, Klassen)

- Wer die Klasse entwickelt ist zuständig für Änderungen in den Verantwortlichkeiten

Objekt Kopplung

= Abhängigkeit der Objekte voneinander
Stark wenn: viele sichtbare Methoden und Variablen, viele Nachrichten im laufenden System, viele Parameter in Methoden

Objektkopplung soll schwach sein: Hinweis auf gute Kapselung, weniger unnötige Beeinflussung bei Änderungen

Klassenzusammenhalt soll maximiert und Objektkopplung minimiert werden

6. Frage) Aspektorientierte Programmierung

= Aufteilung der Funktionalitäten eines Programms auf einzelne Modularisierungseinheiten
(Seperation of Concerns)

- In der OOP funktioniert die Aufteilung gut für Kernfunktionalitäten (Core Concerns), die sich leicht in eine Klasse packen lassen
- Manchmal gibt es aber Anforderungen die alle Bereiche eines Programms betreffen (Cross-Cutting-Concerns/ Querschnittsfunktionalitäten)- diese werden in Aspekten gekapselt
- Aspekt beschreibt wie Programmcode zu ändern ist
- Wir verwenden AspectJ
- Aspect Weaver wendet Aspekte auf Klassen an
- In Aspect J kapselt ein Aspekt vollständig alle Teile von Cross-Cutting-Concerns, die Implementierung der Funktionalität und Spezifikationen der Stellen wo diese angewendet wird

Begriffe in AspectJ:
Join-Point: identifizierbare Stelle während einer Programmausführung (z.B Aufruf einer Methode, Zugriff auf Objekt)
Pointcut: wählt Join Point aus und sammelt kontextabhängige Infos dazu (z.B Argumente eines Methodenaufrufs)
Advice: Code der vor, um oder nach dem Join Point ausgeführt wird
Aspect: zentrale Element in AspectJ, erhält alle Deklarationen, Methoden, Pointcuts und Advices

execution(MethodSignature) Ausführung einer Methode
call(MethodSignature) Aufruf einer Methode
execution(ConstructorSignature) Ausführung eines Konstruktors
call(ConstructorSignature) Aufruf eines Konstruktors
get(FieldSignature) lesender Feldzugriff
set(FieldSignature) schreibender Feldzugriff

Da AspectJ die Syntax von Java erweitert, können AspectJ Programme nicht mit javac übersetzt werden sondern werden mit dem Compiler ajc übersetzt. (alle Klassen zusammen und nicht einzeln übersetzt)
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.

Annotationen
= man versieht unterschiedliche Programmteile mit Markierungen (Annotationen), Laufzeitsystem und Entwicklungswerkzeuge prüfen das Vorhandensein bestimmter Markierungen und reagieren darauf entsprechend
= optionaler Parameter der an fast beliebige Sprachkonzepte anheftbar ist, im JavaCode statisch gesetzt wird, von gesamter Werkzeugkette bis zur Laufzeit auslesbar

- Zur Deklaration von Annotationen hat man die Syntax von Interfaces übernommen und adaptiert
- Falls @Retention(RUNTIME) wird echtes Interface erzeugt
- Zugriff durch Reflection (über Class)

7. Frage) Generizität

An Stelle expliziter Typen werden im Programm Typparameter verwendet. Das sind einfache Namen, die später durch Typen ersetzt werden.
Man muss Programmcode nur 1 mal schreiben und kann es für mehrere Typen verwenden.
Nur Referenztypen: Integer, Character, Boolean…
Generizität bietet statische Typsicherheit.
Generizität vermeidet Typfehler und kann die Lesbarkeit eines Programms verbessern.

Einfache Generizität ist für einige Verwendungszwecke nicht ausreichend: denn es ist nichts über den Typen der den Typparameter ersetzt bekannt(ob Objekte dieser Typen bestimmte Methoden oder Variablen haben).
Man kann Schranken setzen- eine Klasse oder beliebig viele Interfaces. Nur Untertypen dieser Schranken dürfen den Typparameter ersetzen. Default Schranke (also wenn keine angegeben) ist Object.
F-gebundene Generizität= Generizität mit rekursiven Typparametern

Genetizität unterstützt keine impliziten Untertypenbeziehungen.

Wildcards: Nichtunterstützung impliziter Untertypbeziehungen kann auch Nachteile haben. Dadurch erlaubt der Compiler die Verwendung von Parametern nur an Stellen, für deren Typen in Untertypbeziehungen Kovarianz gefordert ist (Lesezugriffe).
Manchmal gibt es auch Parameter deren Inhalte in einer Methode nur geschrieben und nicht gelesen werden: dann Erlaubt der Compiler nur die Verwendung eines Parameters , für deren Typen in Unterbeziehungen Kontravarianz gefordert ist. (Schreibzugriffe)
In der Praxis etwas kompliziert: Richtung (ko/kontravariant) dreht sich mit jeder Klammerebene um. (Compiler kann das aber gut lesen und warnt zuverlässig wenn falsche Wildcards verwendet werden)

Einschränkungen in Java: Typparameter können nicht zur Erzeugung neuer Objekte verwendet werden.

Verwendung von Generizität:

- Wenn es mehrere gleich strukturierte Klassen bzw. Methoden gibt (Containerklassen)
- Klassen und Methoden in Bibliotheken sollen generisch sein
- wenn Änderungen der Parametertypen absehbar sind, soll man Typparameter als Type formaler Parameter verwenden

Untertypbeziehungen und Generizität sind eng miteinander verknüpft- eine sinnvolle Verwendung con Generiität setzt das Bestehen geeigneter Untertypbeziehungen voraus.
Beide helfen Änderungen im Code klein zu halten.
Untertypbeziehungen und Generizität ergänzen sich, Generizität ist hilfreich, wenn das Ersetzbarkeitsprinzip nicht erfüllt ist, während Untertypbeziehungen den Ersatz eines Objekts eines Obertyps durch ein Objekt eines Untertyps auch abhängig von Parametern ermöglichen.
Nur Untertypen können Ersetzbarkeit gewährleisten.

Bsp: homogene Listen (sichergestellt dass sie nur Elemente eines Typs enthalten)- das ist ohne Generizität nicht möglich.
Heterogene Listen (Elemente unterschiedlicher Typen)- dafür werden Untertypbeziehungen gebraucht.

In java wird **homogene Übersetzung**: jede generische Klasse, so wie jede nicht-generische Klasse werden in genau 1 Klasse mit JVM Code übersetzt.

Heterogene Übersetzung: für jede Verwendung einer generischen Klasse oder Methode mit anderen Typparametern wird ein eigener übersetzter Code erzeugt. Da für alle Typen ein eigener Code erzeugt wird, sind elementare Typen als Ersatz der Typparameter geeignet. Es müssen zur Laufzeit keine Typumwandlungen stattfinden. In c++ zum Beispiel.
Effizienter da keine Typumwandlungen, aber oft viele Klassen und große Programme, keine Raw Types verwendbar

In C++ muss man keine Schranken angeben durch die heterogene Übersetzung von Templates. Es wird für jede übersetzte Klasse getrennt geprüft ob die Typen alle vorausgesetzten Eigenschaften erfüllen.

Raw-Types: übersetzte Klassen
Java übersetzt alles homogen bleibt aber Typsicher

Kovariante Probleme: Typen von Eingangsparametern können nur kontravariant sein. Kovariante Eingangsparameter verletzen das Ersetzbarkeitsprinzip.
In der Praxis wünscht man sich aber oft kovariante Eingangsparameter – diese Probleme nennt man Kovariante Probleme.
Zur Lösung dieser Probleme dienen dynamische Typabfragen und Typumwandlungen. Kovariante Probleme soll man meiden.
Binäre Methoden: Spezialfall von kovarianten Problemen. Haben mindestens einen formalen Parameter dessen Typ stets gleich der Klasse ist welche die Methode enthält.
Überladen vs. Multimethoden
Überladen= statisches Binden
Überladen: Es wird stets der deklarierte Typ verwendet auch wenn der dynamische bekannt ist.
Man soll Überladen nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden.

Multimethoden= dynamisches Binden- wenn man dynamische Typen verwendet.

Java unterstützt keine Multimethoden.Man kann aber welche simulieren indem man mehrfach dynamisches Binden durch wiederholtes einfaches Binden anwendet. Damit eliminiert an dynamische Typabfragen und Typumwandlungen.

Universeller Polymorphismus:
enthaltender Polymorphismus durch Untertypbeziehungen:
- Ersetzbarkeit: unvorhersehbare Wiederverwendung,
- kann Clients von lokalen Codeänderungen abschotten,
- nicht immer verwendbar (kovariante Probleme)

parametrischer Polymorphismus = Generizität:
- kaum Einschränkungen (auch für kovariante Probleme),
- keine Ersetzbarkeit,
- Parameteränderung wirkt sich auf Clients aus

8.Frage) Modul, Komponente
Modul: Übersetzungseinheit, z.B. Interface oder Klasse, enthält Deklaration und Definitionen von zusammengehörenden Variablen, Typen, Prozeduren, Funktionen, Methoden,usw.
Module lassen sich relativ unabhängig voneinander entwickeln, Module können nicht zyklisch voneinander abhängen wegen getrennter Übersetzung

Komponente: eigenständiges Stück Software, das in Programm eingebunden wird, Übersetzungseinheit, ähneln Modulen sind aber flexibler,Komponente importiert Inhalte von zur Übersetzungszeit nicht genau bekannten anderen Komponenten, erst beim Einbinden in ein Programm werden diese anderen Komponenten bekannt, deshalb zyklische Abhängigkeit ist erlaubt