Was sind Objekt, Klasse und Vererbung
Ein Objekt ist das wichtigste Konzept in der objektorientierten Programmierung. Es ist die Grundlegende Einheit in der Ausführung eines Programms. Zur Laufzeit besteht ein Programm aus mehreren Objekten, die einander teilweise kennen und Nachrichten untereinander austauschen. Ein Objekt ist am ehesten mit einer Kapsel zu vergleichen, die zusammengehörige Variablen und Routinen (Funktionen, Prozeduren und Methoden) enthält.
Jedes Objekt gehört zu genau einer Klasse, die die Struktur des Objekts - dessen Implementierung – im Detail beschreibt. Außerdem beschreibt die Klasse Konstruktoren. Das sind Routinen zur Erzeugung und Initialisierung von neuen Objekten. Diese Objekte werden als Instanzen bezeichnet.
Vererbung ermöglicht neue Klassen aus bereits existierenden Klassen abzuleiten. Dabei werden nur die Unterschiede zwischen der abgeleiteten Klasse (Unterklasse) und der Basisklasse (Oberklasse) angegeben. Vererbung erspart dem Programmierer Schreibaufwand und Programmänderungen werden vereinfacht.

Identität, Zustand, Verhalten, Schnittstelle
Identität kennzeichnet ein Objekt eindeutig. Sie ist unveränderlich. Über die Identität kann man das Objekt ansprechen (ist also sozusagen die Adresse des Objekts)
Zustand setzt sich aus den Werten der Variablen im Objekt zusammen. Ist in der Regeln änderbar
Verhalten eines Objekts beschreibt, wie sich das Objekt beim erhalten einer Nachricht verhält, das heißt, was ausgeführt wird (Methoden).
Schnittstelle eines Objekts beschreibt das Verhalten des Objekts im Detailiertheitsgrad, der für Zugriffen von außen notwendig ist. Oft enthalten Schnittstellen nur die Köpfe der überall aufrufbaren Routinen.

deklarierter, statischer und dynamischer Typ
Deklarierter Typ ist der Typ, mit dem die Variable deklariert wurde. Dieser existiert natürlich nur bei expliziter Typdeklaration
Statischer Typ wird vom Compiler ermittelt und kann spezifischer sein als deklarierter Typ.
Dynamischer Typ ist der spezifischste Typ. Sie können sich mit jeder zuweisung ändern. Dynamische Typen werden unter anderem für die Typüberprüfung zur Laufzeit verwendet.

Nachricht, Methode, Konstruktor
Nachrichten werden zwischen Objekten verschickt und führen zur Ausführung von Methoden.
Methoden sind die Dinge die Objekte tun, wenn sie Nachrichten erhalten.
Der Konstruktor erzeugt eine Instanz einer Klasse.

Faktorisierung, Refaktorisierung
Bei der Faktorisierung sollen zusammengehörige Eigenschaften und Aspekte zu einer Einheit zusammengefasst werden. Gute

Faktorisierung führt dazu, dass zur Änderung aller Stellen auf die gleiche Art und Weise eine einzige Änderung in der Routine ausreicht. Gute Faktorisierung erhöht also die Wartbarkeit.
Bei der Refaktorisierung wird nach einiger Zeit Faktorisierung wieder ausgeführt, da neue Zusammenhänge klar werden.

Verantwortlichkeiten, Klassen Zusammenhalt, Objekt Kopplung Softwareentwurfsmuster

Welche Arten von Polymorphismus unterscheidet man? Welche davon sind in der objektorientierten Programmierung wichtig? Warum?
Es gibt 4 Arten von Polymorphismus die sich in zwei Gruppen aufteilen. Universelle Polymorphismen wie Generizität und enthaltener Polymorphismus und Ad-hoc Polymorphismen wie Überladen und Typumwandlungen. Die universellen Polymorphismen sind spezifisch für die objektorientierte Programmierung.

Sind Datenabstraktion, Datenkapselung, und data hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?

Nennen Sie die Schritte im Softwareentwicklungsprozess entsprechend dem Wasserfallmodell und in zyklischen Modellen.
Beim Wasserfallmodell werden die Schritte Analyse, Entwurf, Implementierung und Verifikation und Validierung der Reihe nach ausgeführt. Bei zyklischen Modellen werden die oben genannten Schritte in einem Zyklus wiederholt ausgeführt. Zyklische Prozesse verkraften Anforderungsänderungen besser, aber Zeit und Kosten sind schwerer zu planen.

Warum ist gute Wartbarkeit so wichtig?
Gute Wartbarkeit kann die Gesamtkosten erheblich reduzieren. Wartbarkeit setzt sich zusammen aus Einfachheit, Lesbarkeit, Lokalität und Faktorisierung.

Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?
Die Faustregeln sind „Der Klassen-Zusammenhalt soll hoch sein“ und „Die Objekt-Kopplung soll schwach sein“. Objektkopplung ist stark wenn die Anzahl der nach außen sichtbaren Methoden und Variablen groß ist, im laufenden Systm Nachrichten und Variablenzugriffe zwischen unterschiedlichen Objekten häufig auftreten und die Anzahl der Parameter der Methoden groß ist. Der Klassenzusammenhalt ist der Grad der Beziehung zwischen den Verantwortlichkeiten der Klasse. Dieser ist hoch wenn Variablen und Methoden der Klasse eng zusammenarbeiten.

Welche Arten von Software kann man wiederverwenden und welche Rolle spielt jede davon in der Softwareentwicklung?
Programme, Daten, Erfahrungen und Code lassen sich

wiederverwenden. Bei Programmen ist das noch sehr offensichtlich, da sie darauf ausgelegt sind wiederverwendet zu werden. Auch Daten in Datenbanken werden häufig wiederverwendet. Wichtig, aber häufig unterschätzt ist die Wiederverwendung von Erfahrungen in Form von Konzepten und Ideen. Die Wiederverwendung von Code sind zum Beispiel die Verwendung von Bibliotheken.

Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?
Gut faktorisierte Programme lassen sich leichter Wiederverwenden.

Nennen Sie die wichtigsten Paradigmen der Programmierung und ihre essentiellen Eigenschaften.
Imperative Programmierung wird dadurch charakterisiert das Programme aus Anweisungen (Befehlen) aufgebaut sind. Diese werden in einer festgelegten Reihenfolge ausgeführt. Untergruppen der imperativen Programmierung sind

- Prozedurale Programmierung, die den konventionellen Programmierstil beschreibt. Programme sind in sich gegenseitig aufrufende Prozeduren zerlegt. Programmzustände werden als global gesehen, das heißt Daten können überall im Programm verändert werden.
- Objektorientierte Programmierung ist eine Weiterentwicklung der strukturierten prozeduralen Programmierung und stellt den Begriff des Objekts in den Mittelpunkt. Der wesentliche Unterschied zur prozeduralen Programmierung ist, dass Routinen und Daten zu Objekten zusammengefasst werden. Dies macht die Wartung einfacher, hat aber zur Konsequenz das ein Algorithmus manchmal nicht nur an einer Stelle im Programm steht, sondern aufgeteilt sein kann.

Deklarative Programme beschreiben Beziehungen zwischen Ausdrücken in einem System. Es gibt hier keine zustandsändernden Anweisungen. Die wichtigsten Paradigmen in der deklarativen Programmierung sind die funktionale und logikorientierte Programmierung.

Wofür ist die objektorientierte Programmierung gut geeignet und wofür ist sie nicht gut geeignet?
Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

In welcher Form kann man durch das Ersetzbarkeitsprinzip Wiederverwendung erzielen?
Das Ersetzbarkeitsprinzip lautet: „Ein Typ U ist ein Untertyp eines Typs T, wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird“
Untertyp darf überall verwendet werden wo dessen Obertyp verwendet wird Man erreicht höhere Flexibilität beim erweitern von Klassenkomplexen (Beispiel Grafiktreiber), Schnittstellen müssen nicht geändert werden -> genauere Implementierung in Untertypen

Unter welchen Bedingungen, die von einem Compiler überprüfbar sind, ist ein Typ im Allgemeinen Untertyp eines anderen Typs? Welche zusätzliche Bedingungen müssen in Java gelten?

Für jede Konstante im Obertyp muss es eine entsprechende Konstante im Untertyp geben wobei der deklarierte Typ der Konstante im Untertyp ein Untertyp der Konstante im Obertyp ist.
Für jede Variable im Obertyp gibt es eine entsprechende Variable im Untertyp, wobei die deklarierten Typen der Variablen gleich sind.
Für jede Methode im Obertyp gibt es eine entsprechende Methode im Untertyp, wobei der deklarierte Ergebnistyp der Methode im Untertyp ein Untertyp des Ergebnistyps in Methode des Obertyps ist.
Eine Untertypenbeziehung in Java besteht dann, wenn die dem Untertyp entsprechende Klasse mittels extends oder implements von der dem Obertyp entsprechenden Klasse direkt oder indirekt abgeleitet wird.

Sind die im vorigen Punkt angeschnittenen Bedingungen hinreichend, damit das Ersetzbarkeitsprinzip erfüllt ist? Wenn nicht, was muss noch beachtet werden?
Zusicherungen : Vor-(Kontravariant), Nachbedingungen(Kovariant) sowie invarianten müssen für eine vollständige Funktionalität gewährleistet sein.

Welche Rolle spielt dynamisches Binden für die Ersetzbarkeit und Wartbarkeit?
Man spricht von dynamischem Binden wenn ein Methodenaufruf zur Laufzeit anhand des tatsächlichen Typs eines Objekts aufgelöst wird. Dynamisches Binden wird anstellen von switch Anweisungen und geschachtelten if Anweisungen verwendet und erhöht die Lesbarkeit und damit auch die Wartbarkeit.

Welche Arten von Zusicherungen werden unterschieden, und wer ist für deren Einhaltung verantwortlich?
Vorbedingungen sind Bedingungen, für deren Erfüllung vor der Ausführung der Methode der Client verantwortlich ist. Sie beschreiben hauptsächlich welche Eigenschaften die Argumente, mit denen die Methoden aufgerufen werden, erfüllen müssen.
Nachbedingungen sind Bedingungen, die nach Ausführung der Methode erfüllt werden müssen. Für sie ist der Server verantwortlich. Nachbedingungen beschreiben Eigenschaften des Methodenergebnisses und Änderungen bzw Eigenschaften des Objektzustandes.
Invarianten sind Bedingungen, für deren Erfüllung vor sowie nach dem Ausführen jeder Methode der Server verantwortlich ist. Direkte Schreibzugriffe auf Variablen des Servers kann der Server aber nicht kontrollieren. Dafür ist der Client verantwortlich.

Wie müssen sich Zusicherungen in Unter- und Obertypen zueinander verhalten, damit das Ersetzbarkeitsprinzip erfüllt ist? Warum?
Neben den Bedingungen die immer für das Ersetzbarkeitsprinzip gelten müssen auch noch weitere erfüllt sein. Jede Vorbedingungen auf einer Methode im Obertyp muss eine Vorbedingungen auf der entsprechenden Methode im Untertyp implizieren. Vorbedingungen können im Untertyp schwächer werden. Der Grund liegt darin das ein

Aufrufer der Methode, der nur den Obertyp kennt, nur die Erfüllung der Vorbedingungen im Obertyp sicherstellen kann. Daher muss die Vorbedingung auch im Untertyp automatisch erfüllt sein.
Jede Nachbedingungen auf einer Methode im Untertyp muss eine Nachbedingung auf der entsprechenden Methode im Obertyp implizieren. Nachbedingungen im Untertyp können stärker werden. Der Grund liegt darin das sich Aufrufer, die nur den Obertyp kennen sich auf die Erfüllung der Nachbedingung verlassen, auch wenn die Methode im Untertyp ausgeführt wird.
Jede Invariante im Untertyp muss eine Invariante im Obertyp implizieren. Auch hier kann die Invariante im Untertyp stärker werden. Der Grund liegt darin das sich ein Client der nur den Obertyp kennt sich auf die Erfüllung der Invariante verlassen kann, auch wenn diese im Untertyp ausgeführt wird.

Warum sollen Schnittstellen und Typen stabil bleiben? Wo ist Stabilität besonders wichtig?
Die Wiederverwendung funktioniert nur dann gut wen die Schnittstellen bzw Typen stabil bleiben, da eine Änderung auch zu Änderungen in den darauf aufbauenden Typen zu nötigen Änderungen führt. Besonders wichtig ist die Stabilität an der Wurzel der Typhierarchie. Man sollte Untertypen nur von stabilen Obertypen bilden.

Was sind Kovarianz, Kontravarianz und Invarianz
Man spricht von Kovarianz wenn der deklarierte Typ eines Elements im Untertyp ein Untertyp des deklarierten Typs des entsprechenden Obertyps ist. Zum Beispiel sind deklarierte Typen von Konstanten und Ergebnissen der Methoden kovariant. Typen und die betrachtetend darin enthaltenen Elementtypen variieren in dieselbe Richtung.
Bei der Kontravarianz ist der deklarierte Typ eines Elements im Untertyp ein Obertyp des deklarierten Typs des Elements im Obertyp. Zum Beispiel sind deklarierte Typen von formalen Eingangsparametern kontravariant. Typen und die btrachteten darin entahltenen Elementtypen variieren in entgegengesetzte Richtungen.
Invarianz liegt dann vor wenn der deklarierte Typ eines Elements im Untertyps gleich dem deklarierten Typ des entsprechenden Elements im Obertyp ist. Zum Beispiel sind deklarierte Typen von Variablen und Durchgangsparameter invariant. Die betrachteten in den Typen enthaltenen Elementtypen variieren nicht.

Was ist im Zusammenhang mit allgemein zugänglichen (public) Variablen und Invarianten zu beachten?
Public Variablen können von einem Client verändert werden, und somit Vor und Nachbedingungen von entsprechenden Methoden umgangen werden. Get/Setter Methoden sind hier eine Lösung, oder Refaktorisierung im Sinne einer besseren Struktur (Objektkopplung)

Wie genau sollen Zusicherungen spezifiziert sein?
Alle benötigten Zusicherungen sollen (explizit also Kommentare oder zumindest durch sprechende Namen implizit) im Programm stehen. Zur Verbesserung der Wartbarkeit sollen Zusicherungen

keine unnötigen Details festlegen. Sie sollen unmissverständlich formuliert sein und während der Programmentwicklung und Wartung ständig bedacht werden.

Wozu dienen abstrakte Klassen und abstrakte Methoden? Wo und wie soll man abstrakte Klassen einsetzen?
Abstrakte Klassen werden dann eingesetzt, wenn keine Instanzen der Klasse selbst, sondern nur Instanzen der Unterklassen benötigt werden. Methoden die in der abstrakten Klasse deklariert werden, müssen in allen Unterklassen enthalten sein. Abstrakte Klassen die keine Implementierungen enthalten, sind eher stabil als andere Klassen. Zur Verbesserung der Wartbarkeit soll man vor allem von stabilen Klassen erben.

Ist Vererbung dasselbe wie das Ersetzbarkeitsprinzip? Wenn Nein, wo liegen die Unterschiede?
Bei Vererbung wird scheinbar eine Kopie der Oberklasse angelegt und die Klasse durch Überschreiben und Erweitern abgeändert. Vererbung ist zur direkten Wiederverwendung von Code einsetzbar und damit unabhängig vom Ersetzbarkeitsprinzip. Eine Klasse die von einer anderen Erbt muss nicht dort Einsetzbar sein, wo die Oberklasse einsetzbar ist.

Worauf kommt es zur Erzielung von Codewiederverwendung eher an: auf Vererbung oder Ersetzbarkeit? Warum?
Wiederverwendung durch das Ersetzbarkeitsprinzip ist wesentlich wichtiger als direkte Wiederverwendung durch Vererbung. Verzichtet man auf Ersetzbarkeit wird die Wartung erschwert da sich kleine Änderungen auf das gesamte Programm auswirken können. Viele Vorteile der objektorientierten Programmierung gehen damit verloren.

Was bedeuten folgende Begriffe in Java?

Instanzvariable, Klassenvariable, statische Methode
Instanzvariablen sind Variablen die zu den Instanzen einer Klasse gehören. Werden durch Hinschreiben eines Typs gefolgt vom Namen und Strichpunkt deklariert.
Klassenvariablen gehören nicht zu einer Instanz, sondern zur Klasse selbst und werden mit Voranstellen von static deklariert.
Statische Methoden werden über den Namen einer Klasse aufgerufen, nicht über den Namen einer Instanz. Sie gehören ebenfalls zur Klasse, nicht zur Instanz.

static initializer
Ein static initializer besteht nur aus dem Schlüsselwort static gefolgt von geschwungenen Klammern. Diese Sequenz wird vor der ersten Verwendung der Klasse ausgeführt.

geschachtelte und innere Klasse
Geschachtelte Klassen sind Klassen innerhalb einer andere Klasse. Sie können überall dort definiert sein wo auch Variablen definiert werden dürfen.

final Klasse und final Methode
Final Klassen und Methoden können in Unterklassen nicht

überschrieben werden.

Paket

Jede compilierte Klasse in Java wird in einer eigenen Datei gespeichert. Das Verzeichnis, das diese Datei enthält, entspricht dem Paket, zu dem die Klasse gehört.

Wo gibt es in Java Mehrfachvererbung, wo Einfachvererbung?

Nur auf Interfaces wird Mehrfachvererbung unterstützt. Im restlichen Java wird nur Einfachvererbung unterstützt.

Welche Arten von import Deklarationen kann man in Java unterscheiden?

Es dürfen beliebig viele imports am Anfang einer Datei mit Quellcode stehen, sonst aber nirgends. Man kann entweder einzelne Pakete lokal bekanntmachen durch zb „import myclass examples.test". Dann sind alle public Methoden von Klassen im Paket test bekannt und mittels test.Klasse.methode() aufrufbar. Eine Klasse lässt sich mittels „import myclass.examples.test.Klasse" importieren. Ein Aufruf wäre dann mit Klasse.methode() möglich. Man kann aber auch alle Klassen in einem Paket importieren mittels „import myclass.examples.test.*" Auch danach lässt sich mit Klasse.methode() jede public Methode der Klasse aufrufen.

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

Es gibt die Möglichkeit eine Methode public zu definieren, dann ist sie überall sichtbar und werden vererbt. private Methoden sind nur in dem Bereich sichtbar und verwendbar in dem sie erstellt wurden. protected Methoden sind im selben Paket sichtbar, aber nicht in anderen Paketen.

Was sind Interfaces in Java, und wodurch unterscheiden sie sich von abstrakten Klassen? Wann soll man Interfaces verwenden? Wann sind abstrakte Klassen besser geeignet?

Interfaces sind eingeschränkte abstrakte Klassen in denen alle Methoden abstrakt sind. Sie beginnen mit dem Schlüsselwort interface. Normale Variablen dürfen nicht deklariert werden, wohl aber als static final. Alle Methoden sind public, das Schlüsselwort kann man weglassen da alles andere Verboten ist. Nach dem Schlüsselwort extends können mehrere, durch Komma getrennte Namen von Interfaces stehen. Dadurch gibt es hier Mehrfachvererbung.
Interfaces sollten immer dann eingesetzt werden wenn die oben beschriebenen Einschränkungen zu keinen Nachteilen führen.

Was ist Generizität? Wozu verwendet man Generizität?

Generische Klassen, Typen und Routinen enthalten Typparameter, für die später Typen eingesetzt werden. Generizität erspart also Schreibarbeit.

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

Bei gebundener Generizität wird einem Typparameter noch eine Schranke hinzugefügt. Nur Untertypen dieser Schranke dürfen dann den Typparameter ersetzen. Das führt zu gebundenen Typparametern die zusätzliche Informationen liefern und so Zugriff auf mehr und spezifischere Methoden ermöglichen.

In welchen Fällen soll man Generizität einsetzen, in welchen nicht?
Generell ist der Einsatz immer sinnvoll, wenn er die Wartbarkeit verbessert. Man soll Generizität immer verwenden wenn es mehrere gleich strukturierte Klassen bzw Routinen gibt, oder vorraussehbar ist, das es solche geben wird. Typische Beispiele dafür sind Containerklassen.
Generizität ermöglicht es, Programmteile unverändert zu lassen, obwohl sich Typen ändern. Insbesondere betrifft das Typen von formalen Parametern. Generizität ist dafür geeignet, erwartete Änderungen der Typen von formalen Parametern bereits im Voraus zu berücksichtigen.
Generizität kann dort nicht eingesetzt werden wo zum Beispiel eine Listenklasse benötigt wird die verschiedene Typen enthält. Solche heterogenen Listen sollten man mit Untertypenbeziehungen lösen.

Was ist f-gebundene Generizität?
F-gebundene Generizität unterstützt keine impliziten Untertypbeziehugen. Zum Beispiel besteht zwischen List<Y> und List<Y> keine Untertypbeziehungen wenn X und Y verschieden sind, auch dann nicht, wenn Y von X abgeleitet ist. Generizität mi rekursiven Typparametern wird nach dem formalen Modell, in dem solche Konzepte untersucht wurden, F-gebundene Generizität genannt.

Was bedeutet statische Typsicherheit?
Es wird zur Kompilierzeit Überprüft ob irgendwelche Type Conversion Errors auftreten (zb Zuweisung eines dynamischen Obertypen zu dem zugehörigen deklarierten Untertypen).

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?
Homogene Übersetzung: Dabei wird jede generische Klasse mit JVM Code übersetzt. Jeder gebundene Typparameter wir im übersetzten Code einfach durch die (erste) Schranke des Typparameters ersetzt, jeder ungebundene Typparameter durch Object. Wenn eine Methode eine Instanz eines Typparameter zurück gibt, wird der Typ der Instanz nach dem Methodenaufruf dynamisch in den Typ, der den Typparameter ersetzt, umgewandelt. Dies entspricht der Simulation einiger Aspekte von Generizität. Im Unterschied zur simulierten Generizität wird die Typkompatibilität aber vom Compiler garantiert.
Heterogene Übersetzung: Dabei wird für jede Verwendung einer generischen Klasse oder Routine mit anderen Typparametern ein eigener übersetzter Code erzeugt. Das entspricht also eher „copy & paste“, wobei in jeder Kopie alle Vorkommen von Typparametern

durch die entsprechenden Typen ersetzt sind. Dem Nachteil einer großen Anzahl übersetzter Klassen und Routinen steht der Vorteil gegenüber, da für alle Typen eigener Code erzeugt wird, sind einfache Typen (wie int, char, …) problemlos, ohne Einbußen an Laufzeiteffizienz, als Ersatz für Typparameter geeignet. Zur Laufzeit brauchen keine Typumwandlungen und damit zusammenhängende Überprüfungen durchgeführt werden. Außerdem sind auf jede übersetzte Klasse eigene Optimierungen anwendbar.

Was sind (gebundene) Wildcards als Typen in Java? Wozu kann man sie verwenden?
Wildcards werden als Platzhalter für Typen in gebundener Generizität verwendet, in Zusammenhang mit einer oberen bzw unteren Schranke

Wie kann man Generizität simulieren? Worauf verzichtet man, wenn man Generizität nur simuliert?
Generizität wird durch die händische homogene Übersetzung erreicht. Hierbei wird bei den ganzen Typecasts auf statische Typsicherheit verzichtet.

Was wird bei der heterogenen bzw homogenen Übersetzung von Generizität genau gemacht?
Homogene Übersetzung: Dabei wird jede generische Klasse mit JVM Code übersetzt. Jeder gebundene Typparameter wir im übersetzten Code einfach durch die (erste) Schranke des Typparameters ersetzt, jeder ungebundene Typparameter durch Object. Wenn eine Methode eine Instanz eines Typparameter zurück gibt, wird der Typ der Instanz nach dem Methodenaufruf dynamisch in den Typ, der den Typparameter ersetzt, umgewandelt. Dies entspricht der Simulation einiger Aspekte von Generizität. Im Unterschied zur simulierten Generizität wird die Typkompatibilität aber vom Compiler garantiert.
Heterogene Übersetzung: Dabei wird für jede Verwendung einer generischen Klasse oder Routine mit anderen Typparametern ein eigener übersetzter Code erzeugt. Das entspricht also eher „copy & paste", wobei in jeder Kopie alle Vorkommen von Typparametern durch die entsprechenden Typen ersetzt sind. Dem Nachteil einer großen Anzahl übersetzter Klassen und Routinen steht der Vorteil gegenüber, da für alle Typen eigener Code erzeugt wird, sind einfache Typen (wie int, char, …) problemlos, ohne Einbußen an Laufzeiteffizienz, als Ersatz für Typparameter geeignet. Zur Laufzeit brauchen keine Typumwandlungen und damit zusammenhängende Überprüfungen durchgeführt werden. Außerdem sind auf jede übersetzte Klasse eigene Optimierungen anwendbar.

Welche Möglichkeiten für dynamische Typabfragen gibt es in Java und wie funktionieren sie genau?
getClass: liefert die zugehörige Klasse des Objekts
instanceOf: überprüft ob ein Objekt Instanz einer Klasse ist

Was wird bei einer Typumwandlung in Java umgewandelt? Der deklarierte, dynamische oder statische Typ? Warum?

Die Typumwandlung wandelt den deklarierten Typ in einen anderen Typ (meist den gewünschten dynamischen Typ) um, da dem deklarierten Typ keine Details über die spezifischere Implementierung im dynamischen Typ bekannt ist, wodurch er nicht auf diese Funktionalität zugreifen kann.

Welche Gefahren bestehen bei Typumwandlungen?
Bei einem Cast „nach unten" (Typumwandlung in den Untertypen eines Obertypen) kann der Fehler auftreten, dass der dynamische Typ gar nicht dem gecasteten Typen entspricht (unsafe type conversion)

Wie kann man dynamische Typabfragen und Typumwandlungen vermeiden? In welchen Fällen kann das schwierig werden?
Durch dynamisches Binden sind solche Typecasts vermeidbar, da der Compiler hierbei den dynamischen Typ automatisch aufruft.
Probleme: der deklarierte Typ ist zu weit oben in der Typhierarchie -> sehr viele Möglichkeiten des dynamischen Bindens die nicht alle abgedeckt werden können, darüber hinaus muss die Methode für das dynamische Binden in sämtlichen Untermethoden mitimplementiert werde.
Die Typhierarchie kann nicht erweitert werden -> keine weitere Möglichkeiten alternative Lösungszweige durch dynamische Methodenaufrufe zu lösen Zugriff der Methoden auf private content

Welche Arten von Typumwandlungen sind sicher? Warum?
Cast von Untertypen auf Obertypen – Weniger erwartete Funktionalität -> kein Fehler
Dynamische Typabfragen im Programm – Prüfen auf Errors ab, bei Ungleicheit kein Cast
Logische Verfolgung des Programmflusses um Typecast Errors zu vermeiden

Was sind kovariante Probleme und binäre Methoden? Wie kann man mit ihnen umgehen oder sie vermeiden?
Binäre Methoden verlangen ein Objekt ihrer eigenen Klasse als Parameter ab Bildung einer gemeinsamen Oberklasse vermeidet das Problem einer Abwärtskompatibilität der Methode bezüglich Untertypen.
Kovariante Probleme beschreiben die Problemstellung, dass eine Modellierung einer gewissen Situation die Kovarianz der Eingangsparameter abverlangt (sprich man will in den Untertypen speziellere Parameter haben, um die Vorteile des dynamischen Bindens genießen zu können). Die Löschung der Schnittstelle löst hier das Problem, sodass man keine Typecast Checks durchführen muss. Man kann in so einem Fall nurmehr die Methoden mit den konkreten Untertypen aufrufen.

Wie unterscheidet sich Überschreiben von Überladen, und was sind Multimethoden?
Während man beim Überschreiben die entsprechende Methode im Obertypen neu definiert (diese ist trotzdem mit super aufrufbar), wird beim Überladen ein und die selbe Methode für mehrere Parameter implementiert. Es wird anhand dem deklarierten (!)

Übergabetypen ermittelt welche Methode aufgerufen werden soll. Dies ist auch der Unterschied zu Multimethoden, bei welchem neben der zu wählenden Klasse auch der Eingangsparameter dynamisch ermittelt wird.

Wie kann man Multimethoden simulieren? Welche Probleme können dabei auftreten?
Multimethoden verwenden mehrfaches dynamisches Binden: Die auszuführende Methode wird dynamisch durch die Typen mehrerer Argumente bestimmt. In Java gibt es nur einfaches dynamisches Binden. Trotzdem ist es nicht schwer, mehrfaches dynamisches Binden durch wiederholtes einfaches Binden zu simulieren.

Was ist das Visitor Entwurfsmuster?
Das Visitor Pattern ist ein klassisches Entwurfsmuster. Es beschreibt die simulierung von Multimethoden. Hierbei wird die Klasse welche durch direktes dynamisches Binden festgestellt wir als Visitorklasse, sowie Klassen die durch indirektes dynamisches Binden festgestellt werden als Elementklasse. Visitor- und Elementklassen sind oft gegeneinander austauschbar. Der große Nachteil am Visitor Entwurfsmuster ist, dass die Anzhal der benötigten Methoden schnell sehr groß wird.

Wodurch ist Überladen problematisch, und in welchen Fällen ergeben sich kaum Probleme?
Existieren ein oder mehrere Methoden welche als Paramter einen Untertypen den einer überladenen Methode haben, so kommt es oft zu einem Irrtum, Java würde dynamisch aus dem deklarierten Typen die richtige Methode aussuchen, was aber falsch ist. Es wird immer die Methode genommen deren Parameter die allgemeinste Schnittstelle darstellt.
Dies kann man umgehen indem mindestens eine der Parameterpositionen zweier überschriebener Methoden in keine Untertyprelation steht, bzw keine gemeinsamen Untertypen haben im Falle von Mehrfachvererbung. Oder alle Parameter der allgemeineren überschriebenen Methode sind Obertypen der Parameter der anderen Methoden, und die allgemeinere Methode verweist per dynamischer Typabfrage auf eine spezifischere Methode.

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 Error und Exception.
Unterklassen von Error werden hauptsächlich für vordefinierte schwerwiegende Ausnahmen des Java-Laufzeitsystems verwendet und deuten auf echte Fehler hin. Es ist praktisch kaum möglich, solche Ausnahmen abzufangen.
Unterklassen von Exception sind in zwei Bereiche gegliedert. Ausnahmen die von Programmierern selbst definiert wurden, und Ausnahmen die Instanzen von RuntimeException sind.
Oft ist es sinnvoll Instanzen von Exception abzufangen und den Programmablauf an geeigneter Stelle fortzusetzen.

Wie sind Ausnahmen in Untertypbeziehungen zu berücksichtigen?
Ausnahmen sind kontravariant, sprich im Untertyp dürfen nur weniger Exceptions geworfen werden als im Obertyp.

Wozu kann man Ausnahmen verwenden? Wozu soll man sie verwenden, wozu nicht?
Ausnahmen werden in folgenden Fällen eingesetzt:
Unvorhergesehene Programmabbrüche: Wird eine Ausnahme nicht abgefangen, kommt es zu einem Programmabbruch. Die entsprechende Bildschirmausgabe enthält genaue Informationen über Art und Ort des Auftretens der Ausnahme. Damit lassen sich die Ursachen von Programmfehlern leichter finden.
Kontrolliertes Wiederaufsetzen: Nach aufgetretenen Fehlern oder in außergewöhnlichen Situationen wird das Programm an genau definierten Punkten weiter ausgeführt. Im praktischen Einsatz soll das Programm auch dann noch funktionieren, wenn ein Fehler aufgetreten ist. Ausnahmebehandlungen wurden vor allem zu diesem Zweck eingeführt.
Ausstieg aus Sprachkonstrukten: Ausnahmen sind nicht auf den Umgang mit Programmfehlern beschränkt. Sie erlauben ganz allgemein das vorzeitige Abbrechen der Ausführung von Blöcken, Kontrollstrukturen, Methoden, usw. in außergewöhnlichen Situationen.
Rückgabe alternativer Ergebniswerte: In Java und vielen anderen Sprachen kann eine Methode nur Ergebnisse eines bestimmten Typs liefern. Wenn in der Methode eine unbehandelte Ausnahme auftritt, wird an den Aufrufer statt eines Ergebnisses die Ausnahme zurückgegeben, die er abfangen kann. Damit ist es möglich, dass die Methode an den Aufrufer in Ausnahmesituationen Objekte zurückgibt, die nicht den deklarierten Ergebnistyp der Methode haben.

Durch welche Sprachkonzepte unterstützt Java die nebenläufige Programmierung? Wozu dienen diese Sprachkonzepte?
Java unterstützt nebenläufige Programmierung in der mehrere Threads gleichzeitig nebeneinander laufen und Befehle aus verschiedenen Threads scheinbar gleichzeitig ausgeführt werden. Die Programmierung wird durch nebenläufige Threads aufwändiger, da man gelegentlich neue Threads erzeugen und kontrollieren muss, vor allem aber, da man diese Threads synchronisieren muss.

Wozu brauchen wir Synchronisation? Welche Granularität sollen wir dafür wählen?
Synchronisation wird benötigt um dafür zu sorgen, dass Threads die in Variablen schreiben, immer vorher fertig sind bevor die nächste daraus lesen kann, damit also zum Beispiel Zählungen nicht verschwinden. Das wird in Java mithilfe von „Locks" gemacht, die verhindern das ein Thread auf ein Objekt zugreift bevor der eigentliche Thread mit dem Objekt noch nicht fertig ist.

Was sind Deadlocks?
Von einem Deadlock spricht man dann, wenn zu viel Synchronisation

verwendet wurde. Deadlocks sind zyklische Abhängigkeiten von zwei oder mehr Threads. Hier gibt es einen Thread der ein Objekt blockiert und auf ein anderes wartet. Und einen zweiten Thread, der genau das blockiert und auf jenes wartet das der andere hält.

Erklären sie folgende Entwurfsmuster und beschreiben sie jeweils das Anwendungsgebiet, die Struktur, die Eigenschaften und wichtige Details der Implementierung:

Factory Method

Der Zweck einer Factory Method 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. Als Beispiel kann man sich ein System zur Verwaltung von Dokumenten unterschiedlicher Art vorstellen. Dabei gibt es eine abstrakte Klasse mit der Aufgabe, neue Dokumente anzulegen. Nur in einer Unterklasse, der die Art des neuen Dokuments bekannt ist, kann die Erzeugung, tatsächlich durchgeführt werden.
Generell ist das Entwurfsmuster anwendbar wenn eine Klasse neue Objekte erzeugen soll, deren Klasse aber nicht kennt, 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.
Die Struktur sieht wie folgt aus. Eine (oft abstrakte) Klasse Product ist ein gemeinsamer Obertyp aller Objekte, die von Factory Method erzeugt werden können. Die Klasse ConcreteProduct ist eine bestimmte Unterklasse davon (zb im Dokumentensystem Text). Die abstrakte Klasse Creator enthält neben anderen Operationen die Factory Methode. Diese kann von außen, aber auch beispielsweise in einer anderen Methode von der Klasse selbst verwendet werden. Eine Unterklasse ConcreteCreator implementiert Factory Method. Ausführungen dieser Methode erzeugen neue Instanzen von ConcreteProduct.
Factory Methods haben folgende Eigenschaften: Sie bieten 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. Sie verknüpfen parallele Klassenhierarchien, die Creator Hierarchie mit der Product Hierarchie.

Prototype

Das Entwurfsmuster Prototype dient dazu, die Art eines neu zu erzeugenden Objekts durch ein Prototyp-Objekt zu spezifizieren. Neue Objekte werden durch Kopieren des Prototyps erzeugt.
Zum Beispiel kann man in einem System, in dem verschieden Arten von Polygonen vorkommen, ein neues Polygon durch kopieren eines Bestehenden erzeugen. Das neue Polygon hat die selbe Klasse wie das, von dem die Kopie erstellt wurde. An der Stelle, an der der Kopiervorgang stattfindet, braucht diese Klasse nicht bekannt zu sein.
Generell ist das Entwurfsmuster anwendbar, wenn ein System

unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und dargestellt werden, und wenn die Klassen, von denen Instanzen erzeugt werden sollen, erst zur Laufzeit bekannt sind, oder vermieden werden soll, eine Hierarchie von Creator Klassen zu erzeugen, die einer parallelen Hierarchie von Product Klassen entspricht (Factory Pattern), oder jede Instanz einer Klasse nur wenige unterschiedliche Zustände haben kann; es ist oft einfacher, für jeden möglichen Zustand einen Prototyp zu erzeugen und diesen zu kopieren, als Instanzen durch new zu erzeugen und dabei passende Zustände anzugeben.
Das Entwurfsmuster hat folgende Struktur. Eine (möglicherweise abstrakte) Klasse Prototype spezifiziert eine (möglicherweise abstrakte) Methode clone um sich selbst zu kopieren. Die konkreten Unterklassen (zb Dreieck, Rechteck, usw im Polygon Beispiel) überschreiben diese Methode. Eine Klasse Client ist das Programm das dann clone in den Prototypen durch dynamisches Binden aufruft.
Prototypen haben folgende Eigenschaften. Sie verstecken die konkreten Produktklassen vor den Anwendern und reduzieren damit die Anzahl der Klassen, die Anwender kennen müssen. Die Anwender brauchen nicht geändert zu werden, wenn neue Produktklassen dazukommen oder geändert werden.
Prototypen können auch zur Laufzeit jederzeit dazugegeben und weggenommen werden. Im Gegensatz dazu darf die Klassenstruktur zur Laufzeit nicht verändert werden.
Prototypen erlauben die Spezifikation neuer Objekte durch änderbare Werte. In hochdynamischen Systemen kann neues Verhalten durch Objektkomposition statt durch die Definition neuer Klassen erzeugt werden.
Sie vermeiden übertrieben große Anzahlen an Unterklassen. Im Gegensatz zur Factory Method ist es nicht nötig, parallele Klassenhierarchien zu erzeugen.

Singleton
Das Entwurfsmuster Singleton sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf diese Instanz. Es gibt zahlreiche Anwendungsmöglichkeiten für dieses Entwurfsmuster. Beispielsweise soll in einem System nur ein Drucker-Spooler existieren. Eine einfache Lösung besteht in der Verwendung einer globalen Variable, aber diese verhindert nicht, dass mehrere Instanzen der Klasse erzeugt werden. Es ist besser, die Klasse selbst für die Verwaltung ihrer einzigen Instanz verantwortlich zu mache. Dies ist die Aufgabe eines Singleton.
Singleton ist anwendbar wenn es genau eine Instanz einer Klasse geben soll, und diese global zugreifbar sein soll und die Klasse durch Vererbung erweiterbar sein soll, und Anwender die erweiterte Klasse ohne Änderungen verwenden können soll.
Singletons haben folgende Eigenschaften. Sie erlauben den kontrollierten Zugriff auf die einzige Instanz. Sie vermeiden durch Verzicht auf globale Variablen unnötige Namen im System und alle weiteren unangenehmen Eigenschaften globaler Variablen. Sie unterstützen Vererbung. Sie erlauben auch mehrere Instanzen. Man kann die Entscheidung zugunsten nur einer Instanz im System jederzeit ändern und auch die Erzeugung mehrerer Instanzen

ermöglichen. Die Klasse hat weiterhin vollständige Kontrolle darüber, wie viele Instanzen erzeugt werden. Und sie sind flexibler als statische Methoden, da statische Methoden kaum Änderungen erlauben und dynamisches Binden nicht unterstützen.

Decorator

Das Entwurfsmuster Decorator gibt Objekten dynamisch zustätzliche Verantwortlichkeiten. Decorators stellen eine flexible Alternative zur Vererbung bereit.
Manchmal möchte man einzelnen Objekten zusätzliche Verantwortlichkeiten geben, nicht aber der ganzen Klasse. Zum Beispiel möchte man einem Fenster am Bildschirm Bestandteile wie eine scroll bar geben, anderen Fenstern aber nicht.
Im allgemeinen ist dieses Entwurfsmuster anwendbar um dynamisch Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte dadurch zu beeinflussen, für Verantwortlichkeiten, die wieder entzogen werden können und wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind, beispielsweise um eine sehr große Zahl an Unterklassen zu vermeiden, oder weil die Programmiersprache in einem speziellen Fall keine Vererbung unterstützt.
Das Entwurfsmuster hat folgende Struktur. Die abstrakte Klasse (oder das Interface) Component definiert eine Schnittstelle für Objekte, an die Veranwortlichkeiten dynamisch hinzugefügt werden können. Die Klasse ConcreteComponent ist eine konkrete Unterklasse davon. Die abstrakte Klasse Decorator definiert eine Schnittstelle für Verantwortlichkeiten, die dynamisch zu Komponenten hinzugefügt werden können. Jede Instanz dieses Typs enthält eine Referenz namens component auf eine Instanz des Typs Component, das ist das Objekt, zu dem die Verantwortlichkeit hinzugefügt ist.
Unterklassen von Decorator sind konkrete Klassen, die bestimmte Funktionalität wie beispielsweise scroll bars bereitstellen. Sie definieren neben den Methoden, die bereits in Component definiert sind, weitere Methoden und Variablen, welche die zusätzliche Funktionalität verfügbar machen. Wird eine Methode, die in Component definiert ist, aufgerufen so wird dieser Aufruf einfach an das Objekt, das über component referenziert ist, weitergegeben.
Decorators haben einige positive und negative Eigenschaften. Sie bieten mehr Flexibilität als statische Vererbung. Wie ei statischer Erweiterung einer Klasse durch Vererbung werden Verantwortlichkeiten hinzugefügt. Anders als bei Vererbung erfolgt das Hinzufügen der Verantwortlichkeiten zur Laufzeit und zu einzelnen Objekten, nicht ganzen Klassen. Die Verantwortlichkeiten können auch jederzeit wieder weggenommen werden.
Sie vermeiden Klassen, die bereits weit oben in der Klassenhierarchie mit Eigenschaften überladen sind. Es ist nicht notwendig, dass ConcreteComponent die volle gewünschte Funktionalität enthält, da durch das Hinzufügen von Dekoratoren gezielt neue Funktionalität verfügbar gemacht werden kann.
Instanzen von Decorator und die dazugehörenden Instanzen von ConcreteComponent sind nicht identisch. Beispielsweise hat ein Fenster Objekt, auf das über einen Dekorator zugegriffen wird, eine andere Identität als das Fenster Objekt selbst (ohne

Dekorator) oder dasselbe Fenster Objekt, auf das über einen anderen Dekorator zugegriffen wird. Bei Verwendung dieses Entwurfsmusters soll man sich nicht auf Objektidentität verlassen. Sie führen zu vielen kleinen Objekten. Ein Design, das Dekoratoren häufig verwendet, führt nicht selten zu einem System, in dem es viele kleine Objekte gibt, die einander ähneln. Solche Systeme sind zwar einfach konfigurierbar, aber schwer zu verstehen und zu warten.

Proxy

Ein Proxy stellt einen Platzhalter für ein anderes Objekt dar und kontrolliert Zugriffe darauf.
Es gibt zahlreiche sehr unterschiedliche Anwendungsmöglichkeiten für Platzhalterobjekte. Ein Beispiel ist ein Objekt, dessen Erzeugung teuer ist, beispielsweise weil umfangreiche Daten aus dem Internet geladen werden müssen. Man erzeugt das eigentliche Objekt erst, wenn es wirklich gebraucht wird, Statt des eigentlichen Objekts verwendet man in der Zwischenzeit einen Platzhalter, der erst bei Bedarf durch das eigentliche Objekt ersetzt wird. Falls nie auf die Daten zugegriffen wird, erspart man sich den Aufwan der Objekterzeugung.
Jedes Platzhalterobjekt enthält im Wesentlichen einen Zeiger auf das eigentlich Objekt und leitet in der Regel Nachrichten an das eigentliche Objekt weiter. Einige Nachrichten werden auch direkt vom Proxy behandelt.
Das Entwurfsmuster ist anwendbar, wenn eine intelligentere Referenz auf ein Objekt als ein simpler Zeiger nötig ist.
Hier sind einige übliche Situationen in denen Proxy eingesetzt werden kann:
Remote Proxies sind Platzhalter für Objekte, die in anderen Namensräumen (zb auf Festplatten oder auf anderen Rechnern) existieren. Nachrichten an die Objekte werden von den Proxies über komplexere Kommunikationskanäle weitergeleitet.
Virtual Proxies erzeuge Objekte bei Bedarf. Da die Erzeugung eines Objekts aufwändig sein kann, wird sie so lange verzögert, bis es wirklich einen Bedarf gibt.
Protection Proxies kontrollieren Zugriffe auf Objekte. Solche Proxies sind sinnvoll, wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen.
Smart References ersetzen einfache Zeiger. Sie können bei Zugriffen zusätzliche Aktionen ausführen. Typische Verwendngen sind das Mitzählen der Referenzen auf das eigentliche Objekt damit das Objekt entfernt werden kann, wenn es keine Referenz mehr darauf gibt. Das Laden von persistenten Objekten in den Speicher, wenn das erste mal darauf zugegriffen wird und das Zusichern, dass während des Zugriffs auf das Objekt kein gleichzeitiger Zugriff durch einen anderen Thread erfolgt.
Die Struktur ist recht einfach. Eine abstrakte Klasse (oder Interface) Subject definiert die gemeinsame Schnittstelle für Instanzen von den Klassen RealSubject und Proxy. Instanzen von RealSubject und Proxy können gleichermaßen verwendet werden, wo eine Instanz von Subject erwartet wird. Die Klasse RealSubject definiert die eigentlichen Objekte, die durch die Proxies

(Platzhalter) repräsentiert werden. Die Klasse Proxy definiert schließlich die Proxies. Diese Klasse verwaltet eine Referenz realSubject, über die ein Proxy auf Instanzen von RealSubject zugreifen kann. Sie stellt eine Schnittstelle bereit, die der von Subject entspricht, damit ein Proxy als Ersatz des eigentlichen Objekts verwendet werden kann. Sie kontrolliert Zugriffe auf das eigentliche Objekt und kann für dessen Erzeugung oder Entfernung verantwortlich sein. Und sie hat weitere Verantwortlichkeiten, die von der Art abhängen.

Iterator
Ein Iterator, auch Cursor genannt, ermöglicht den sequentiellen Zugriff auf die Elemente eines Aggregats (das ist eine Samlung von Elementen, zum Beispiel eine Collection), ohne die innere Darstellung des Aggregats offen zu legen.
Dieses Entwurfsmuster ist verwendbar um auf den Inhalt eines Aggregats zugreifen zu können, ohne die innere Darstellung offen legen zu müssen, mehrere Abarbeitungen der Elemente in einem Aggregat zu ermöglichen und eine einheitliche Schnittstelle für die Abarbeitungen verschiedener Aggregatsstrukturen zu haben, das heißt, um polymorphe Iterationen zu unterstützen.
Das Entwurfsmuster hat folgende Struktur. Die abstrakte Klasse (oder Interface) Iterator definiert eine Schnittstelle für den Zugriff auf Elemente sowie deren Abarbeitung. Die Klasse ConcreteIterator implementiert diese Schnittstelle und verwaltet die aktuelle Position in der Abarbeitung. Die abstrakte Klasse (oder Interface) Aggreate definiert eine Schnittstelle für die Erzeugung eines neuen Iterators. Die Klasse ConcreteAggregate implementiert diese Schnittstelle. Ein Aufruf von iterator in ConcreteAggregate erzeugt eine neue Instanz von ConcreteIterator. Um die aktuelle Position im Aggregat verwalten zu können, braucht jede Instanz von ConcreteIterator eine Referenz auf die entsprechende Instanz von ConcreteAggregate.
Iteratoren haben drei wichtige Eigenschaften. Sie unterstützen unterschiedliche Varianten in der Abarbeitung von Aggregate. Für komplexe Aggregate wie beispielsweise Bäume gibt es zahlreiche Möglichkeiten, in welcher Reihenfolge die Elemente abgearbeitet werden. Es ist leicht, mehrere Iteratoren für unterschiedliche Abarbeitungsreihenfolgen zu implementieren.
Iteratoren vereinfachen die Schnittstelle von Aggregate, da Zugriffsmöglichkeiten, die über Iteratoren bereitgestellt werden, durch die Schnittstellen von Aggregat nicht unterstützt werden müssen.
Auf ein und demselben Aggregat können gleichzeitig mehrere Abarbeitungen stattfinden, da jeder Iterator selbst den aktuellen Abarbeitungszustand verwaltet.
Es gibt zahlreiche Möglichkeiten zur Implementierung von Iteratoren. Anmerkungen dazu sind:
Man kann zwischen interenen und externen Iteratoren unterscheide. Interne kontrollieren selbst, wann die nächste Iteration erfolgt, bei externen bestimmt der Anwender, wann sie das nächste Element abarbeiten möchten. Externe Iteratoren sind flexibler als Interne. Oft ist es schwierig, externe Iteratoren auf Sammlungen von

Elementen zu verwenden, wenn diese Elemente zueinander in komplexen Beziehungen stehen. Durch die sequentielle Abarbeitung geht die Struktur dieser Beziehungen verloren. Beispielsweise erkennt man an einem vom Iterator zurückgegebenen Element nicht mehr, an welcher Stelle in einem Baum das Element steht. Wenn die Beziehungen zwischen Elementen bei der Abarbeitung benötigt werden, ist es meist einfacher, interne statt externe Iteratoren zu verwenden.
Der Algorithmus zum Durchwandern eines Aggregats muss nicht immer im Iterator selbst definiert sein. Auch das Aggregat kann den Algorithmus bereitstellen und den Iterator nur dazu benützen, eine Referenz auf das nächste Element zu speichern. Wenn der Iterator den Algorithmus definiert, ist es leichter, mehrere Iteratoren mit unterschiedlichen Algorithmen zu verwende. In diesem Fall ist es auch leichter, Teile eines Algorithmus in einem anderen Algorithmus wieder zu verwenden. Andererseits müssen die Algorithmen oft private Implemeniterungsdetails des Aggregats verwenden. Das geht natürlich leichter, wenn die Algorithmen im Aggregat definiert sind. In Java kann man Iteratoren durch geschachtelte Klassen in Aggregaten definieren, wie zum Beispiel den Iterator in der Klasse List. Dadurch wird die ohnehin schon starke Abhängigkeit zwischen Aggregat und Iterator aber leider noch stärker.
Es kann gefährlich sein, ein Aggregat zu verändern, während es von einem Iterator durchwandert wird. Wenn Elemente dazugefügt oder entfernt werden, passiert es leicht, dass Elemente doppelt oder nicht abgearbeitet werden. Eine einfache Lösung dieses Problems besteht darin, das Aggregat bei der Erzeugung eines Iterators zu kopieren. Meist ist die Lösung aber zu aufwändig. Ein robuster Iterator erreicht dasselbe Ziel, ohne das ganze Aggregat zu kopieren. Es ist recht aufwändig, robuste Iteratoren zu schreiben. Die Detailprobleme hängen stark von der Art des Aggregats ab.
Aus Gründen der Allgemeinheit ist es oft praktisch, Iteratoren auch auf leeren Aggregaten bereitzustellen. In einer Anwendung braucht man die Schleife nur so lange auszuführen, so lange es Elemente gibt. Bei leeren Aggregaten daher nie.

Template Method
Eine 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.
Dieses Entwurfsmuster ist anwendbar um den unveränderlichen Teil eines Algorithmus zu implementieren und es Unterklassen zu überlassen, den veränderbaren Teil des Verhaltens festzulegen.
Template Method ist anwendbar wenn gemeinsames Verhalten mehrerer Unterklassen (zum Beispiel im Zuge einer Refaktorisierung) in einer einzigen Klasse lokal zusammengefasst werden soll, um Duplikate im Code zu vermeiden.
Und um mögliche Erweiterungen in Unterklassen zu kontrollieren, beispielsweise durch Template Methods, die hooks aufrufen und nur das Überschreiben dieser hooks in Unterklassen ermöglichen.

Die Struktur dieses Entwurfsmusters ist recht einfach. Die (meist abstrakte) Klasse AbstractClass definiert (abstrakte) primitive Operationen, welche konkrete Unterklassen als Schritte in einem Algorithmus implementieren, und implementiert das Grundgerüst des Algorithmus, das die primitiven Operationen aufruft. Die Klasse ConcreteClass implementiert die primitiven Operationen.
Template Methods haben folgende Eigenschaften: Sie stellen eine fundamentale Technik zur direkten Wiederverwendung von Programmcode dar. Sie sind vor allem in Klassenbibliotheken sinnvoll, weil sie ein Mittel sind, um gemeinsames Verhalten zu faktorisieren. Sie führen zu einer umgekehrten Kontrollstruktur, die manchmal als Hollywood Prinzip bezeichnet wird. Die Oberklasse ruft Methoden der Unterklasse auf, nicht umgekehrt.
Sie rufen oft nur eine von mehreren Arten von Operation auf:

- konkrete Operationen (entweder in ConcreteClass oder in der Klasse, in der die Template Methoden angewandt werden)
- konkrete Operationen in AbstractClass, also Operationen, die ganz allgemein auch für Unterklassen sinnvoll sind
- abstrakte primitive Operationen, die einzelne Schritt im Algorithmus ausführen
- Factory Methods
- hooks, das 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.

Visitor
Dient zur Simulation von Multimethoden. Statt dem Überladen von Methoden um eine Unterscheidung anhand des Typen durchzuführen wird im ersten Schritt eine Visitor Klasse vom deklarierten Typen aufgerufen. Durch dynamisches Binden wird die Methode in der Richtigen Klasse aufgerufen. Als Parameter wird der deklarierte Typ der Elementklasse übergeben. In der Methode selbst wird dann wieder durch dynamisches Binden des Eingangsparameters die richtige Elementklasse ausgewählt und dort durch den Namen der Methode (nicht durch Parameter) die entsprechende Methode von allen durch die Fallunterscheidung möglichen ausgesucht.

Wird die Anzahl der benötigten Klassen im System bei Verwendung von Factory Method, Prototype, Decorator und Proxy (gegenüber einem System, das kein Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt es unverändert?
Factory Method : Anzahl der Klassen steigt
Prototype : Anzahl der Klassen wird niedriger
Decorator : Anzahl der Klassen sinkt
Proxy : Anzahl der Klassen steigt leicht

Wird die Anzahl der benötigten Objekte im System bei Verwendung von Factory Method, Prototype, Decorator und Proxy (gegenüber einem System, das kein Entwurfsmuster verwendet) eher erhöht, vermindert oder bleibt es unverändert?
Factory Method : Anzahl der Objekte bleibt gleich
Prototype : Anzahl der Objekte steigt

Decorator : Anzahl der Objekte steigt
Proxy : Anzahl der Objekte steigt

Vergleiche sie Factory Method mit Prototype. Wann stellt welches Entwurfsmuster die bessere Lösung dar? Warum?
Factory Method wird verwendet um Objekte zu erstellen deren Aufbau aber die zur Erstellung aufgerufene Klasse nicht kennt. Clients müssen sich nicht um die konkrete Implementierung der Unterklasse kümmern, die Verantwortlichkeit liegt in den Unterklassen. Das hinzufügen neuer Strukturen ist aber mühsam, da hierfür die Hierarchie 2seitig erweitert werden muss.
Prototype ist sehr dynamisch, da ständig neue Objekte gebildet werden können, ohne zusätzlich die Klassenstruktur zu erweitern. Die Verwaltung all dieser Objekte erweist sich aber als schwer, und eine klare Strukturierung und Aufteilung der Verantwortlichkeiten wie bei der Factory Method existiert nicht.

Welche Unterschiede gibt es zwischen Decorator und Proxy?
Beide Patterns können zusätzliche Verantwortlichkeit an die konkreten Klassen delegieren. Beim Decorator wird dies durch eine Wrapperklasse bewerkstelligt, wodurch dynamisch ständig im System Verantwortlichkeiten hinzugefügt und entfernt werden können. Der Proxy ist hingegen eine Verweisklasse, deren Anwendungszweck nur nebensächlich die zusätzliche Funktionalität ist, sondern die Zugriffskontrolle. Proxies sind darüber hinaus nicht mit einer so hohen Anzahl an kleinen Objekten wie Decorators konfrontiert.

Welche Probleme kann es beim Erzeugen von Kopien im Prototype geben? Was unterscheidet flache Kopien von tiefen?
Prototypen können Referenzen auf weitere Objekte (sprich Prototypen) haben. In Java wird standardmäßig nur eine flache Kopie in Objekt definiert. Bei tiefen Kopien werden die in den Verweisen stehenden Objekte kopiert, sodass nicht zwei Prototypen auf das selbe Objekt zeigen. Das Hauptproblem stellen hierbei Zyklen dar, die zu einer Endlosschleife führen können.

Für welche Arten von Problemen ist Decorator gut geeignet, für welche weniger? (Oberfläche versus Inhalt)
Decorator ist für oberflächenbasierende Anwendungen sehr praktisch (elementarer Aufbau), im Gegensatz dafür für umfangreiche und inhaltliche Anwendungen nicht geeignet.

Kann man mehrere Decorators bzw Proxies hintereinander verketten? Wozu kann so etwas gut sein?
Durch Verkettung mehrer Decorators kann eine einfache Struktur elementar zu einer sehr komplexen und funktional umfangreichen Komponente zusammengebaut werden, und auch jederzeit einzelne Funktionalitäten entfernt werden. Je nach Bedarf kann man zu jeder Zeit sein Objekt anpassen. Genauso ist dies bei Proxies, mehrere Funtkinalitäten werden in einer Art Fließbandarbeit abgearbeitet, wobei jeder Proxy sich um seine eigenen Verantwortlichkeiten kümmert

Was unterscheidet hooks von abstrakten Methoden?

Welche Arten von Iteratoren gibt es, und wofür sind sie geeignet? Inwiefern können geschachtelte Klassen bei der Implementierung von Iteratoren hilfreich sein?

Was ist ein robuster Iterator? Wozu braucht man Robustheit?