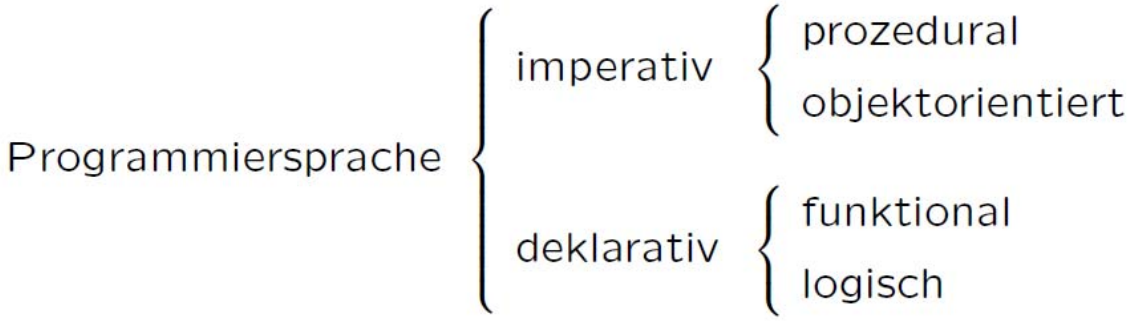
# Kapitel 1

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




**2. Wozu dient ein Berechnungsmodell?**
Hinter jedem Paradigma steckt ein Berechnungsmodell, die immer einen formalen Hintergrund haben. Sie müssen konsistent und in der Regel Turing-vollständig sein.

**3. Welche Berechnungsmodelle werden in Programmierparadigmen verwendet, und welche charakteristischen Eigenschaften haben sie?**
Funktionen:
Prädikatenlogik:
Constraint-Programmierung:
Temporale Logik und Petri-Netze:
Freie Algebren:
Prozesskalküle:
Automatentheorie:
While, GoTo & Co:

**4. Welche Eigenschaften von Berechnungsmodellen sind für deren Erfolg häufig (mit)bestimmend?**
Kombinierbarkeit, Konsistenz, Abstraktion, Systemnähe, Unterstützung, Beharrungsvermögen

**5. Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen? Wie äußert sich dieses Spannungsfeld?**

Es ist es unmöglich, folgende Forderungen gleichzeitig und in vollem Umfang zu erfüllen:

- Flexibilität und Ausdruckskraft sollen in kurzen Texten die Darstellung aller vorstellbaren Programmabläufe ermöglichen.
- Lesbarkeit und Sicherheit sollen Absichten hinter Programmteilen sowie mögliche Inkonsistenzen leicht erkennen lassen.
- Die Konzepte müssen verständlich bleiben und es muss klar sein, was einfach machbar ist und was nicht

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

Die strukturierte Programmierung bringt mehr Struktur in die prozedurale Programmierung.
Jedes Programm bzw. jeder Rumpf einer Prozedur ist nur aus drei einfachen Kontrollstrukturen aufgebaut:

- Sequenz (ein Schritt nach dem anderen)
- Auswahl (Verzweigung im Programm)
- Wiederholung (Schleife, Rekursion oder Ähnliches)

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

Deklarative Paradigmen: radikaler Ansatz
strebt referentielle Transparent als Eigenschaft an Bsp: f(x) + f(x) = 2 f(x)

Objektorientierte Paradigmen: gemäßigter Ansatz
man nimmt an, dass es Querverbindungen gibt und beschränkt sie lokal auf einzelne Objekte

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

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

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

Ein- und Ausgaben sind Seiteneffekte, die referenzielle Transparenz unmöglich macht.
Gelöst hat man es, indem man Ein- und Ausgaben nur gut sichtbar und ganz oben in der Aufrufhirachie geschoben und sie dadurch aus allen Funktionen verbannt hat.

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

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

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

First Class Entities sind Funktionen, die wie normale Daten verwendbar sind, in Variablen abgelegt werden können, als Argumente an andere Funktionen übergeben und als Ergebnisse von Funktionen zurückbekommen werden können.

- \- Häufig sehr kompliziert als vergleichbare Konzepte, weil die uneingeschränkte Verwendbarkeit eine große Zahl an zu berücksichtigenden Sonderfällen nach sich zieht.
- \+ Die uneingeschränkte Verwenbarkeit bringt Vorteile beispielsweise beim Einsatz mit Funktionen höherer Ordnung (Funktionen mit Funktionen als Parameter).Sie bringen eine neue Dimension in der Programmierung.

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

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

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

- Modul (Übersetzungseinheit, zyklenfrei, enthält Deklarationen bez. Definitionen von Variablen, Typen, Prozeduren, Funktionen, usw.)

- Objekt (zur Laufzeit erzeugt, keine Einschränkungen hinsichtlich zyklische Abhängigkeiten, kapseln Variablen und Methoden zu logischen Einheiten und schütze private Inhalte von Zugriffen // *Kapselung* und *Data-Hiding*)

- Klasse (Modul und Schablone für Objekterzeugung, gibt Variablen in Objekten vor und spezifiziert wichtigste Eigenschaften)
- Komponente (komplexere Initialisierung, eigenständiges Stück Software, dass in ein Programm eingebunden wird, ähneln Modulen)
- Namensraum (jede oben genannte Modularisierungseinheit bildet einen eigenen Namensraum, verhindert Namenskonflikte)

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

Klar definierte Schnittstellen zwischen Modulen sind sehr hilfreich: Einerseits braucht der Compiler Schnittstelleninformation um Inhalte anderer Module verwenden zu können, andererseits ist diese Information auch beim Programmieren wichtig um Abhängigkeiten zwischen Modulen besser zu verstehen. Meist wird nur ein kleiner Teil des Modulinhalts von anderen Modulen verwendet. Schnittstellen unterscheiden klar zwischen Modulinhalten, die von anderen Modulen zugreifbar sind, und solchen, die nur innerhalb des Moduls gebraucht werden.

Erstere werden exportiert, letztere sind privat. Private Modulinhalte sind von Vorteil: Sie können vom Compiler im Rahmen der Programmiersprachsemantik beliebig optimiert, umgeformt oder sogar weggelassen werden, während für exportierte Inhalte eine gewissen Regeln entsprechende Zugriffsmöglichkeit von außen bestehen muss. Änderungen privater Modulinhalte wirken sich nicht auf andere Module aus. Änderungen exportierter Inhalte machen hingegen oft entsprechende Änderungen in anderen Modulen nötig, die diese Inhalte verwenden.

**15. Was ist ein Namensraum?**

Die Namen für Objekte in einer Art Baumstruktur angeordnet und über entsprechende Pfadnamen eindeutig angesprochen. (Jede oben genannte Modularisierungseinheit bildet einen eigenen Namensraum, verhindert Namenskonflikte)

**16. Warum können Module nicht zyklisch voneinander abhängen?**

Weil das aufgrund der getrennten Übersetzung nicht möglich ist. Wenn ein Modul B Inhalte eines Moduls A importiert, kann A keine Inhalte von B importieren. Eine gemeinsame Übersetzung würde der Definition von Modulen wiedersprechen.

**17. Was versteht man unter Datenabstraktion, Kapselung und Data-Hiding?**

| | |
|---|---|
| Datenabstraktion: | Kapselung in Kombination mit Data-Hiding |
| Kapselung: | Daten werden zB. in einem Objekt gekapselt (Vektor x und y Koordinate) |
| Data-Hiding: | private Variablen werden in einem Objekt versteckt (Var ohne getter Methode) |

**18. Wodurch unterscheiden sich Komponenten von Modulen?**

Während ein Modul Inhalte ganz bestimmter, namentlich genannter anderer Module importiert, importiert eine Komponente Inhalte von zur Übersetzungszeit nicht genau bekannten anderen Komponenten. Erst beim Einbinden in ein Programm werden diese anderen Komponenten bekannt. Sowohl bei Modulen als auch Komponenten ist offen, wo exportierte Inhalte verwendet werden, aber bei Komponenten ist zusätzlich offen, von wo importierte Inhalte kommen. Letzteres verringert die Abhängigkeit der Komponenten voneinander. Deswegen gibt es bei der getrennten Übersetzung kein Problem mit zyklischen Abhängigkeiten.

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

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

**20. Wie kann man globale Namen verwalten und damit Namenskonflikte verhindern?**

Indem man vermehrt global eindeutige Namen zur Adressierung von Modularisierungseinheiten macht.

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

Es kann zur Laufzeit erfolgen:

| | |
|---|---|
| Konstruktor: | Beim Erzeugen eines Objekts werden Objektvariablen initialisiert |
| Initialisierungsmethode: | Objekte die durch Kopieren erzeugt werden. |
| Zentrale Ablagen: | Daten werden bei zentralen Ablagen abgelegt und bei Initialisierung abgeholt. |

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

Weil auch Objekte durch Kopieren erzeugt werden können und ein Aufrufen des Konstruktors dadurch nicht möglich ist.

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

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

**24. Was unterscheidet Generizität von den verschiedenen Formen der Parametrisierung zur Laufzeit?**
Löcher werden bereits zur Übersetzungszeit gefüllt.

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

Annotationen sind optionale Parameter, die man zu Sprachkonstrukten hinzufügen kann. Annotationen werden von verschiedenen Werkzeugen verwendet, oder aber einfach ignoriert.
Die Löcher, die durch Annotationen befüllt werden, sind im Gegensatz zur Generizität nirgends im Programm festgelegt. Daher ist die Art und Weise, wie die mitgegebenen Informationen zu verwenden sind, ebenso unterschiedlich wie die Anwendungsgebiete.

**26. Was versteht man unter aspektorientierter Programmierung?**

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

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

Notwendige Änderungen ist bei Modularisierungseinheiten nur schwer möglich, da dadurch auf Änderungen an anderen Stellen notwendig werden. Wenn die Löcher sich ändern, muss man auch das ändern, das zum Befüllen der Löcher benötigt wird.

**28. Wann ist *A* durch *B* ersetzbar?**

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

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

Die Ersetzbarkeit geht Hand in Hand mit Schnittstellen. Schnittstellen können also festgelegt sein.

**30. Was ist die Signatur einer Modularisierungseinheit?**

Die Signatur ist eine spezifizierte Schnittstelle, welche Inhalte der Modularisierungseinheit von außen zugreifbar macht. Diese Inhalte werden nur über ihre Namen und gegebenenfalls den Typen von Parameter und Ergebnisse beschrieben. (B muss alles enthalten und von außen zugreifbar machen, was A auch hat)

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

Eine Abstraktion ist ein informeller Text, die zusätzlich zur Signatur beschreibt.

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

Sie beschreiben die erlaubten Erwartungen an eine Modularisierungseinheit. Diese Beschreibung bezieht sich auf alle nach außen sichtbaren Inhalten.

**33. Wann sind Typen miteinander konsistent, und was sind Typfehler?**

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

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

Typen helfen bei der Klassifizierung dieser Werte. Viele Operationen sind nur für Instanzen bestimmter Typen definiert. Beispielsweise kann man nur ganze Zahlen oder Fließkommazahlen miteinander multiplizieren, aber keine Zeichenketten. Wenn die Typen der Operanden mit der Operation zusammenpassen, sind die Typen konsistent.

**35. Was ist der Hauptgrund für den Einsatz statischer Typprüfungen?**

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

**36. Was versteht man unter Typinferenz?**

Viele Typen kann ein Compiler aus der Programmstruktur herleiten; man spricht von Typinferenz. Beispielsweise braucht man in Haskell keinen einzigen Typ hinzuschreiben, obwohl der Compiler alle Typen statisch prüft (auf Basis von Typinferenz). Zur Verbesserung der Lesbarkeit kann und soll man Typen dennoch explizit anschreiben

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

Zum Zeitpunkt der Erstellung von Module werden die meisten wichtigen Entscheidungen getroffen. Hierzu braucht man viel Flexibilität.

**38. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?**

Unsichere Entscheidungen werden eher nach hinten verschoben und zu einem Zeitpunkt getroffen, zu dem bereits viele andere damit zusammenhängende Entscheidungen getroffen wurden und der Entscheidungsspielraum entsprechend kleiner ist.

**39. Wie beeinflussen Typen die Planbarkeit weiterer Schritte?**

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

**Was ist ein abstrakter Datentyp?**

Die Trennung zwischen Innenansicht und Außenansicht einer Modularisierungseinheit (Data-Hiding)
Kapselung in Kombination mit Data-Hiding

**41. Was unterscheidet strukturelle von nominalen Typen?**

Struktureller Typ: Typ der Modularisierungseinheit hängt nur von Namen, Parametertypen und Ereignistypen die nach außen sichtbar sind ab.
Nomialer Typ: Neben Signatur auch einen eindeutigen Namen. Der Typ eines Objekts entspricht dem Namen der Klasse.

**42. Warum verwenden Programmiersprachen meist nominale Typen?**

…, weil man beim Programmieren hauptsächlich abstrakt in Konzepten und nur selten in Signaturen denkt.

**43. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?**

Untertypen werden durch das Ersetzbarkeitsprinzip definiert. Ohne Ersetzbarkeit gibt es keine Untertypen.

Faustregel: Ein Typ U ist Untertyp von Typ T wenn jede Instanz von U überall dort verwendbar ist, wo T verwendet werden kann.

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

Abstrakte Regeln und nicht zugängliche Konzepte, lassen sich nicht automatisch vergleichen.
*(Bsp: Programmierbeispiel – ChangeBox ist kein Untertyp von Box, weil bei Box per definition keine Pixel mehr verändert werden dürfen. Obwohl kaum Unterschied, kann die ChanceBox NIE Untertyp sein)*

**45. Welche wichtige Einschränkung gibt es bei Untertypbeziehungen zusammen mit statischer Typprüfung?**

Typen von Funktions- bez. Methodenparametern dürfen in Untertypen nicht stärker werden.
Bsp: boolean compare(T x) kann im Untertyp nicht zu boolean compare(U x) überschrieben sein.

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

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

**47. Wie konstruiert man rekursive Datenstrukturen?**

Mittels induktiven Konstruktionen (Verketteten Listen udgl.)
data Lst = end | elem(Int,Lst)

**48. Was versteht man unter Fundiertheit rekursiver Datenstrukturen?**

Man muss klar zwischen $M_0$ (nicht-rekursiv) und der Konstruktion aller Mi mit i > 0 (rekursiv) unterscheiden, wobei $M_0$ nicht leer sein darf. Diese Eigenschaft nennt man *Fundiertheit*.

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

Typinferenz funktioniert nicht, wenn gleichzeitig Ersetzbarkeit durch Untertypen verwendet wird. (Typinferenz und Generezität schon)

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

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

**51. Erklären Sie folgende Begriffe:**
- **Objekt, Klasse, Vererbung**
- **Identität, Zustand, Verhalten, Schnittstelle**
- **deklarierter, statischer und dynamischer Typ**
- **Faktorisierung, Refaktorisierung**
- **Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung**

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

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

Vererbung: Ist das Ableiten einer Unterklasse von einer Oberklasse. Reine Vererbung erspart meist nur Schreibaufwand, da Änderungen nur an einer Stelle vorgenommen werden müssen.

Identität: Jedes Objekt ist über eindeutige und unveränderliche Identität identifizier- und ansprechbar

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

Verhalten: Reaktion eines Objekts beim Erhalten einer Nachricht. Ist abhängig von
1. Der aufgerufenen Methode(Routine)
2. Den übergebenen Parametern
3. Den momentan aktuellen Zustand des Objekts (Variablenwerte)

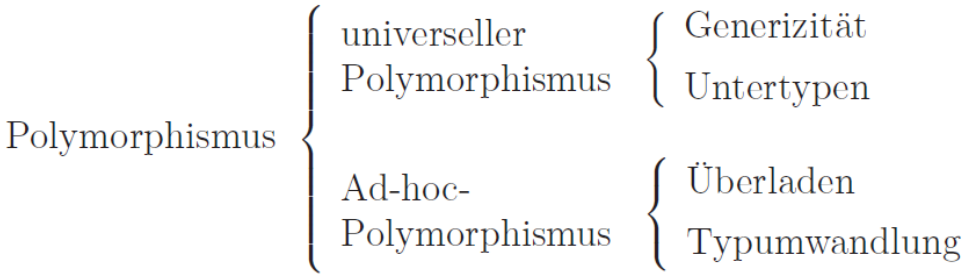
Schnittstellen: Einfach gesagt nur die Köpfe der Methoden, welche Aufschluss über Eingangs und Ausgangsparameter geben.

Instanz einer Klasse: Objekt

Instanz einer Schnittstelle: Es gibt keine direkten Instanzen von Schnittstellen. Alle Objekte welche die Schnittstelle implementieren können als Instanzen dieser Schnittstelle gesehen werden.

Instanz eines Typs: Typ = Ist eine bestimmte Sicht auf ein Objekt. Jedes Objekt kann mehrere Sichten beinhalten. Eine Sicht kann aus Schnittstellen und abstrakten Klassen bestehen. Jedes Objekt das diesen Typ implementiert ist eine Instanz des Typs

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




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

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

Zustandsgleichheit: Wenn die Parameter gleich sind (wird mit equals verglichen)
Ident: Wenn es sich um das identische Objekt, also das selbe Abbild im Speicher, handelt (wird mit == verglichen)

**54. Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?**

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

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

Ein Typ U ist Untertyp eines Typs T, wenn jede Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird.

**56. Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig?**

Aufgrund der hohen Code-Wiederverwendung

**57. Warum ist gute Wartbarkeit so wichtig?**

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

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

Der Klassenzusammenhang soll hoch sein & Die Objektkopplung soll schwach sein

gute Faktorisierung, Wahrscheinlichkeit geringer, dass bei Programmänderung auch die Zerlegung in Klassen und Objekte geändert werden muss, höhere Datenabstraktion und Data-Hiding

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

- Programme
- Erfahrung (Entwurfsmuster)
- Code
- Daten
- Globale Bibliotheken
- Fachspezifische Bibliotheken
- Projektinterne Wiederverwendung:
- Programminterne Wiederverwendung

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

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

Refaktorisierungen ermöglichen das Hinführen des Projektes auf ein stabiles gut faktorisiertes Design. Gute Faktorisierung => starken Klassenzusammenhalt => gut abgeschlossene und somit leicht wiederverwendbare Klassen.

Sie ändert die Struktur eines Programms, lässt aber dessen Funktionalität unverändert.
Es wird dabei also nichts hinzugefügt oder weggelassen, und es werden auch keine inhaltlichen Änderungen vorgenommen. „Das Rad wird nicht mehr neu erfunden."

Faustregel: Ein vernünftiges Maß rechtzeitiger Refaktorisierungen führt häufig zu gut faktorisierten Programmen.

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

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