kann den Entwicklungsaufwand von Algorithmen erhöhen und deren Verständlichkeit verringern. Diese Unterschiede haben einen Einfluss darauf, wofür die Paradigmen gut geeignet sind.

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

Wenn die Wiederverwendung und Wartbarkeit von untergeordneter Bedeutung ist, beispielsweise weil man die Software nur einmal einsetzt, kann die objektorientierte Programmierung ihre Vorteile nicht ausspielen. Diese kommen erst durch einen langen Einsatz- und Wartungszeitraum zum Tragen. In anderen Paradigmen kann man wesentlich rascher programmieren, da man sich nicht um die Wiederverwendbarkeit kümmern muss. Soll Software aber über einen längeren Zeitraum gewartet werden, zahlt es sich aus, durch objektorientierte Programmierung etwas mehr in die Entwicklung zu investieren. Sonst zahlt es sich nicht aus.

1.5 Wiederholungsfragen

Folgende Fragen sollen beim Erarbeiten des Stoffes helfen. Sie stellen keine vollständige Aufzählung möglicher Prüfungsfragen dar.

1. Was versteht man unter einem Programmierparadigma?
2. Wozu dient ein Berechnungsmodell?
3. Welche Berechnungsmodelle werden in Programmierparadigmen verwendet, und welche charakteristischen Eigenschaften haben sie?
4. Welche Eigenschaften von Berechnungsmodellen sind für deren Erfolg häufig (mit)bestimmend?
5. Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen? Wie äußert sich dieses Spannungsfeld?
6. Was ist die strukturierte Programmierung? Wozu dient sie?
7. Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?

8. Was bedeutet referentielle Transparenz, und wo findet man referentielle Transparenz?
9. Wieso passt referentielle Transparenz nicht gut mit Ein- und Ausgabe zusammen, und wie kann man das Dilemma lösen?
10. Welchen Zusammenhang gibt es zwischen Seiteneffekten und der objektorientierten Programmierung?
11. Was sind First-Class-Entities? Welche Gründe sprechen für deren Verwendung, welche dagegen?
12. Was haben Funktionen höherer Ordnung mit einem applikativen Programmierstil zu tun?
13. Welche Modularisierungseinheiten gibt es, was sind ihre charakteristischen Eigenschaften, und wodurch unterscheiden sie sich?
14. Welche Bedeutung haben Schnittstellen für Modularisierungseinheiten? Warum unterscheidet man zwischen von außen zugreifbaren und privaten Inhalten?
15. Was ist und wozu dient ein Namensraum?
16. Warum können Module nicht zyklisch voneinander abhängen, Komponenten aber schon?
17. Was versteht man unter Datenabstraktion, Kapselung und Data-Hiding?
18. Warum und inwiefern ist die Einbindung von Komponenten komplizierter als die von Modulen?
19. Wie kann man globale Namen verwalten und damit Namenskonflikte verhindern?
20. Was versteht man unter Parametrisierung? Wann kann das Befüllen von „Löchern“ durch welche Techniken erfolgen?
21. Warum braucht man zur Parametrisierung in der Objekterzeugung neben Konstruktoren gelegentlich auch Initialisierungsmethoden?
22. Welche Vor- und Nachteile hat die zentrale Ablage von Werten zum Zweck der Parametrisierung?

23. Was unterscheidet Generizität von den verschiedenen Formen der Parametrisierung zur Laufzeit?
24. Was sind Annotationen und wozu kann man sie verwenden? Wodurch unterscheiden sie sich von Generizität?
25. Was versteht man unter aspektorientierter Programmierung?
26. Wodurch unterscheiden sich die verschiedenen Formen der Parametrisierung von der Ersetzbarkeit, und warum ist die Ersetzbarkeit in der objektorientierten Programmierung von so zentraler Bedeutung?
27. Wann ist A durch B ersetzbar?
28. Wodurch kann festgelegt sein, ob A durch B ersetzbar ist?
29. Was ist die Signatur einer Modularisierungseinheit?
30. Wie verhält sich die Signatur einer Modularisierungseinheit zur Abstraktion, die durch diese Modularisierungseinheit gebildet wird?
31. Was sind Zusicherungen, und welche Rolle spielen sie für Modularisierungseinheiten?
32. Wann sind Typen miteinander konsistent, und was sind Typfehler?
33. Wie schränken Typen die Flexibilität ein, und warum verwendet man Typen trotzdem?
34. Welche Gründe sprechen für den Einsatz statischer Typprüfungen, welche dagegen?
35. Was versteht man unter Typinferenz? Welche Gründe sprechen für bzw. gegen den Einsatz?
36. Zu welchen Zeitpunkten können Entscheidungen getroffen werden (Typen und Entscheidungsprozesse)?
37. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?
38. Wie beeinflussen Typen die Planbarkeit weiterer Schritte?
39. Was ist ein abstrakter Datentyp?
40. Was unterscheidet strukturelle von nominalen Typen?

41. Warum verwenden wir in Programmiersprachen meist nominale Typen, in theoretischen Modellen aber hauptsächlich strukturelle?
42. Wie hängen Untertypbeziehungen mit Ersetzbarkeit zusammen?
43. Warum kann ein Compiler ohne Unterstützung durch Programmierer(innen) nicht entscheiden, ob ein nominaler Typ Untertyp eines anderen nominalen Typs ist?
44. Erklären Sie Einschränkungen bei Untertypbeziehungen zusammen mit statischer Typprüfung.
45. In welchem Zusammenhang verwendet man Higher-Order-Subtyping und F-gebundene Generizität?
46. Wie konstruiert man rekursive Datenstrukturen?
47. Was versteht man unter Fundiertheit rekursiver Datenstrukturen? Welche Ansätze dazu kann man unterscheiden?
48. Warum wird Typinferenz in objektorientierten Sprachen meist nur lokal beschränkt eingesetzt?
49. Wie können statisch geprüfte Typen beliebige Eigenschaften von Werten propagieren?
50. Erklären Sie folgende Begriffe:
    - Objekt, Klasse, Vererbung
    - Identität, Zustand, Verhalten, Schnittstelle
    - deklarierter, statischer und dynamischer Typ
    - Faktorisierung, Refaktorisierung
    - Verantwortlichkeiten, Klassenzusammenhalt, Objektkopplung
51. Welche Arten von Polymorphismus unterscheidet man? Welche davon sind in der objektorientierten Programmierung wichtig? Warum?
52. Wann sind zwei gleiche Objekte identisch und wann sind zwei identische Objekte gleich?
53. Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?
54. Was besagt das Ersetzbarkeitsprinzip? (Häufige Prüfungsfrage!)

55. Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig (mehrere Gründe)?

56. Wann und warum ist gute Wartbarkeit wichtig?

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

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

59. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?

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

Faktorisierung: Teile die zusammen gehören sollen zusammengefasst werden.

Klassen-Zusammenhalt: wenn eine Variable oder eine Methode entfernt wird fehlt etwas grundlegendes

Objekt-Kopplung: hoch wenn die Objekte oft Nachrichten untereinander austauschen und viele nach außen sichtbare Methoden haben.

1. Was versteht man unter einem Programmierparadigma?
   Eine Bestimmte Denkweise oder Art der Programmierung.

2. Wozu dient ein Berechnungsmodell?
   Hinter jedem Paradigma ist ein Berechnungsmodell. Es muss turing-vollständig sein (alle Berechnungen können die auch ein Computer kann) und konsistent

3. Welche Berechnungsmodelle werden in Programmierparadigmen verwendet und welche Eigenschaften haben sie?
   Funktionen: primitiv-rekursive
   Prädikatenlogik: vor allem in Datenbanken
   Constraint-Programmierung: x<5 oder A oder B ist wahr (heute werden dafür fertige Bibliotheken verwendet)
   Temporale Logik und Petri-Netze:
   Freie Algebren: stark vereinfachte Algebren (einfache Axiome)
   Prozesskalküle:
   Automaten: werden heute vorwiegend noch wegen der guten verständlichkeit verwendet
   WHILE, GOTO und CO: z.B.: bei MIPS nur LOOP und END LOOP

4. Welche Eigenschaften von Berechnungsmodellen sind für deren Erfolg häufig (mit)bestimmend?
   Universell einsetzbar, Abstraktion, Beharrungsvermögen, konsistenz

5. Im Spannungsfeld welcher widersprüchlichen Ziele befinden sich Programmierparadigmen? Wie äußert sich dieses Spannungsfeld?
   Flexibilität, Lesbarkeit, verständlich (Man will alles beschreiben aber es soll noch übersichtlich bleiben)

6. Was ist die strukturierte Programmierung? Wozu dient sie?
   Besteht aus drei Kontrollstrukturen:
   Sequenz (eins nach dem anderen), Auswahl (if, else), Wiederholung (Schleife, Rekursion)
   Bringt Struktur in Programmierung und bessere Lesbarkeit

7. Wie gehen unterschiedliche Paradigmen mit Seiteneffekten um?
   Deklarative Paradigmen: radikaler Ansatz = referentiell transparent (f(x)+f(x) = 2 f(x))
   Objektorientierte Paradigmen: Querverbindungen werden lokal auf einzelne Objekte beschränkt.

8. Was bedeutet referentielle Transparenz und wo findet man sie?
   Ein Ausdruck ist referentiell transparent wenn er durch seinen Wert ersetzt werden kann ohne die Semantik des Programms zu ändern (3+4 => 7)

9. Wieso passt referentielle Transparenz nicht gut mit Ein- und Ausgabe zusammen, und wie kann man das Dilemma lösen?
   Weil Ein- und Ausgabe auch Seiteneffekte sind => wird nur ganz oben in der Aufrufhierarchie erlaubt

10. Welchen Zusammenhang gib es zwischen Seiteneffekten und der objektorientierten Programmierung? Siehe 7

11. Was sind First-Class-Entities? Was spricht dafür, was dagegen?

    Funktionen können wie normale Daten verwendet werden => in Variablen ablegen usw..
    Sind komplizierter aber können als Kontrollstrukturen verwendet werden (Funktionen in Funktionen)

12. Was haben Funktionen höherer Ordnung mit einem applikativen Programmierstil zu tun?
    Es können Schablonen mit Löchern geschrieben werden die dann durch die Übergabe von Funktionen (zum Füllen der Löcher ausführbar werden)

13. Welche Modularisierungseinheiten gibt es, was sind ihre charakteristischen Eigenschaften und wodurch unterscheiden sie sich?
    Objekte: (Zustand, Identität, Verhalten) Daten Kapselung und Data Hiding = Datenabstraktion
    Klassen: Schablone für neue Objekte
    Komponente: eigenständiges Stück Software dass nur mit anderen Komponenten richtig funktioniert. Bei Übersetzungszeit ist offen von wo importierte Inhalte kommen
    Namensraum: fassen Modularisierungseinheiten zusammen ohne die getrennte Übersetzbarkeit zu zerstören. => z.B.: a.b.C (Klasse C im Namensraum b, welcher im Namensraum a steht)

14. Welche Bedeutung haben Schnittstellen für Modularisierungseinheiten? Warum unterscheidet man zwischen von außen zugreifbaren und privaten Inhalten?
    Zusammengefasst Informationen über Inhalte des Moduls, die auch in anderen Modulen verwendbar sind. Private Inhalte können vom compiler beliebig optimiert werden. Von außen zugreifbare können nicht beliebig verändert werden da sie mit anderen Modulen zusammenarbeiten und somit auch die anderen Module geändert werden müssten.

15. Was ist und wozu dient ein Namensraum?
    Ein Namensraum gilt pro Modul. Das heißt die Namen müssen nur in diesem Namensraum eindeutig sein. In anderen Modulen/Namensräumen können dieselben Namen verwendet werden. Beim Import kann man Namenskonflikte durch Umbenennungen vermeiden.

16. Warum können Module nicht zyklisch voneinander abhängen, Komponenten aber schon?
    Weil Module getrennt übersetzt werden. Wenn A von B abhängt muss B vor A übersetzt werden. Wenn jetzt B auch von A abhängen würde hätten wir einen Deadlock. Diese Abhängigkeiten kann man mit der Aufteilung eines Moduls in Schnittstelleninformation und Implementierung aufteilt. (z.B.: Interfaces und Klassen (die die Interfaces implementieren))
    Weil bei Komponenten erst zur Laufzeit bestimmt wird von wo importierte Inhalte kommen.
17. Was versteht man unter Datenabstraktion, Kapselung und DataHiding?
    Datenkapselung ist die Verschmelzung von zusammengehörigen Daten in eine Einheit.

DataHiding versteckt private Inhalte vor dem Zugriff von außen.
Datenabstraktion = Datenkapselung + DataHiding

18. Warum und inwiefern ist die Einbindung von Komponenten komplizierter als die von Modulen?
    Während Module getrennt voneinander als eigene selbstständige Einheiten angesehen werden können, interagieren Komponenten untereinander. Außerdem wird bei Komponenten erst zur Laufzeit klar von wo importiert wird.
19. Wie kann man globale Namen verwalten und damit Namenskonflikte verhindern?
    Es wird ganz den Softwareentwicklern überlassen. Man muss alle Dateien anführen die die Modularisierungseinheiten enthalten. Oft helfen standardmäßig vorgegebene Verzeichnisse in denen automatisch nach verwendeten Modul- oder Klassennamen gesucht wird.
20. Was versteht man unter Parametrisierung? Wann kann das Befüllen von „Löchern" durch welche Techniken erfolgen?
    In Modulen werden Löcher gelassen, die erst später gefüllt werden.
    Zur Laufzeit:
    - Konstruktor
    - Initialsierungsmethoden (z.B.: beim Kopieren von Objekten)
    - Zentrale Ablage (globale Variablen die erst zur Laufzeit abgeholt werden (z.B.: durch einfachen Zugriff oder Methoden)

    Generizität:

    Füllung der Löcher zur Übersetzungszeit. Man programmiert so dass mehrere Typen verwendet werden können. Einschränkungen lassen sich schwer umsetzen da es sich nicht um First-Class-Entities handet.

    Annotationen:

    @Override etc. Es lässt sich erfragen mit welcher Annotation etwas versehen ist (dynamisch zur Laufzeit) und können so den Programmablauf steuern. Oft wird dieses Konzept eingesetzt wenn Annotationen nicht nur von lokaler Bedeutung sind. (sondern auch Systemwerkzeuge)

    Aspekte:

    Ein Aspekt spezifiziert gewisse Punkte im Programm und was an diesen Stellen passieren soll. Ein Aspect Weaver modifiziert das Programm entsprechend den Aspekten. (z.B.: Debugging, Logging)

21. Warum braucht man zur Parametrisierung in der Objekterzeugung neben Konstruktoren gelegentlich auch Initialisierungsmethoden?
    Weil objekte auch kopiert werden können

22. Welche Vor- und Nachteile hat die zentrale Ablage von Werten zum Zweck der Parametrisierung?
    Sie ist auch für statische Modularisierungseinheiten verwendbar die bereits zur Übersetzungszeit feststehen.

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

24. Was sind Annotationen und wozu kann man sie verwenden? Wodurch unterscheiden sie sich von Generizität?
    Die Löcher die durch Annotationen gefüllt werden sind im Gegensatz zur Generizität nirgends im Programm festgelegt => Art und Weise der Verwendung sehr unterschiedlich.

25. Was versteht man unter aspektorientierter Programmierung?
    (siehe Frage 20)

26. Wodurch unterscheiden sich die verschiedenen Formen der Parametrisierung von der Ersetzbarkeit, und warum ist die Ersetzbarkeit in der objektorientierten Programmierung von so zentraler Bedeutung?
    Während bei der Parametrisierung Änderungen an den Löchern auch gleich Änderungen an den einzusetzenden Dingen der Löcher nach sich ziehen kann durch Ersetzbarkeit einfacher nachträgliche Änderungen einzelner Moduleinheiten erreicht werden.

27. Wann ist A durch B ersetzbar?
    Wenn der Austausch keinerlei Änderungen an Stellen nach sich zieht, an denen A verwendet wird.

28. Wodurch kann festgelegt sein, ob A durch B ersetzbar ist?
    Wenn die Schnittstelle von B dasselbe Beschreibt wie die Schnittstelle von A (aber die Schnittstelle von B kann mehr beschreiben als die von A)

29. Was ist die Signatur einer Modularisierungseinheit?
    Boolean Funktion(int x); => Signatur = Funktion(int)
    Boolean Funktion(double x); => Signatur = Funktion(double)
    Namen und Typen von Parametern

30. Wie verhält sich die Signatur einer Modularisierungseinheit zur Abstraktion, die durch diese Modularisierungseinheit gebildet wird?
    Zusätzlich zur Signatur werden Schnittstellen durch Namen und informelle Texte beschrieben.

31. Was sind Zusicherungen, und welche Rolle spielen sie für Modularisierungseinheiten?
    Genaue beschreibung der Ewartung an die Modularisierungseinheiten. Vertrag zwischen Modularisierungseinheit (Server) und ihren Verwendern (Clients).
    Vorbedingung: was können sich Server von den Clients erwarten?
    Nachbedingung: was kann sich der Client vom Server erwarten?
    Invarianten: welche Eigenschaften in konsit. Programmzuständen immer erfüllt sind.
    History-Constraints: wie und v.a. in welcher Reihenfolge Clients mir Servern agieren können.

32. Wann sind Typen miteinander konsistent, und was sind Typfehler?
    Wenn die Typen der Operanden mit den Operationen zusammenpassen. Sonst tritt ein Typfehler auf.

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

34. Welche Gründe sprechen für den Einsatz statischer Typprüfungen, welche dagegen?

Pro: Einmal getroffene Entscheidungen bleiben konsistent. Je früher Entscheidungen getroffen werden desto weniger ist zur Laufzeit zu tun. Typfehler werden früher erkannt, wo die Auswirkungen noch nicht so groß sind. Man ist dadurch aber auch eingeschränkter.

35. Was versteht man unter Typinferenz? Welche Gründe sprechen für bzw. gegen den Einsatz?
    Typinferenz spart das Anschreiben von Typen. Typinferenz und Ersetzbarkeit durch Untertypen verträgt sich nicht. (im selben Teil des Programms).
    Lesbarkeit kann dadurch sogar verbessert werden weil es nur dort eingesetzt wird wo der Typ eh glasklar ist.

36. Zu welchen Zeitpunkten können Entscheidungen getroffen werden (Typen und Entscheidungsprozesse)?
    KP ÜBERARBEITEN!

37. Welchen Einfluss können Typen auf Entscheidungszeitpunkte haben?
    Frühere Entscheidungen machen dann zur Laufzeit weniger Arbeit. Sogar wenn man weiß, dass die Variable eine ganze Zahl enthält, muss der Compiler eine beliebige Referenz annehmen und zur Laufzeit eine dynamische Typprüfung durchführen.

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

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

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

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

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

43. Warum kann ein Compiler ohne Unterstützung durch Programmierer(innen) nicht entscheiden, ob ein nominaler Typ Untertyp eines anderen nominalen Typs ist?
Für nominale Typen reichen einfache Regeln nicht aus. Abstrakte und daher den Regeln nicht zugängliche Konzepte lassen sich ja nicht automatisch vergleichen. In der Praxis muss man beim Programmieren explizit hinschreiben, welcher Typ Untertyp von welchem anderen ist.

44. Erklären Sie Einschränkungen bei Untertypbeziehungen zusammen mit statischer Typprüfung.
es gibt keine Möglichkeit, entsprechende Typen statisch zu prüfen. Dynamische Prüfungen sind natürlich möglich. Aus demselben Grund sind auch Wertebereichseinschränkungen im Allgemeinen nicht statisch prüfbar; das bedeutet z.B., dass int nicht als Untertyp von long betrachtet werden kann, obwohl jede Zahl in int auch in long vorkommt.

45. In welchem Zusammenhang verwendet man Higher-Order-Subtyping und F-gebundene Generizität?
Einfache Generizität ist leicht zu verstehen und auch vom Compiler leicht handzuhaben. Die Komplexität steigt jedoch rasch an, wenn man Einschränkungen auf Typparametern berücksichtigt, die vor allem (aber nicht nur) in der objektorientierten Programmierung benötigt werden. Im Wesentlichen gibt es zwei etwa gleichwertige formale Ansätze dafür: Fgebundene Generizität [7] nutzt Untertypbeziehungen zur Beschreibung von Einschränkungen und wird z.B. in Java und C# eingesetzt. HigherOrder-Subtyping, auch Matching genannt [1], geht einen eher direkten Weg und beschreibt Einschränkungen über Untertyp-ähnliche Beziehungen, die wegen Unterschieden in Details aber keine Untertypbeziehungen sind.

46. Wie konstruiert man rekursive Datenstrukturen?
Mittels induktiven Konstruktionen; Mengenvereinigung

47. Was versteht man unter Fundiertheit rekursiver Datenstrukturen? Welche Ansätze dazu kann man unterscheiden?
Man muss klar zwischen M0 (nicht-rekursiv) und der Konstruktion aller Mi mit i > 0 (rekursiv) unterscheiden, wobei M0 nicht leer sein darf. Diese Eigenschaft nennt man Fundiertheit
In Haskell muss es in jeder Typdefinition zumindest eine nicht-rekursive Alternative (z.B. end in Lst) geben.
Die Menge M0 ist etwa in Java schon in der Sprachdefinition vorgegeben und enthält nur den speziellen Wert null

48. Warum wird Typinferenz in objektorientierten Sprachen meist nur lokal beschränkt eingesetzt?
Typinferenz funktioniert nicht, wenn gleichzeitig (also an derselben Stelle im Programm) Ersetzbarkeit durch Untertypen verwendet wird.

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

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

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

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

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

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

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

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

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

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

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

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

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

Faktorisierung: Zerlegung eines Programms in Einheiten mit zusammengehörigen Eigenschaften.

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

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

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

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

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

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

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

- Polymorphismus
  - universeller Polymorphismus
    - Generizität
    - Untertypen
  - Ad-hoc-Polymorphismus
    - Überladen
    - Typumwandlung

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

52. Wann sind zwei gleiche Objekte identisch und wann sind zwei identische Objekte gleich?
    Zwei durch verschiedene Variablen referenzierte Objekte sind identisch wenn es sich um ein und dasselbe Objekt handelt. Zwei Objekte sind gleich wenn sie denselben Zustand und dasselbe Verhalten haben, auch wenn sie nicht identisch sind. In diesem Fall ist ein Objekt eine Kopie des anderen. Zustandsgleichheit: Wenn die Parameter gleich sind (wird mit equals verglichen) Ident: Wenn es sich um das identische Objekt, also das selbe Abbild im Speicher, handelt (wird mit == verglichen)
53. Sind Datenabstraktion, Datenkapselung und Data-Hiding einander entsprechende Begriffe? Wenn Nein, worin unterscheiden sie sich?
54. Was besagt das Ersetzbarkeitsprinzip? (Häufige Prüfungsfrage!)
    Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist wo ein Objekt von T erwartet wird.
55. Warum ist Ersetzbarkeit in der objektorientierten Programmierung so wichtig (mehrere Gründe)?
    Eine Möglichkeit zur praxistauglichen nachträglichen Änderung von Modularisierungseinheiten verspricht der Einsatz von Ersetzbarkeit statt oder zusätzlich zur Parametrisierung:
    Aufgrund der hohen Code-Wiederverwendung. Ohne Ersetzbarkeit keine Untertypen.

56. Wann und warum ist gute Wartbarkeit wichtig?
    Da Wartungskosten ca. 70 % der Gesamtkosten ausmachen. Gute Wartbarkeit erspart Unmengen an Geld! Wenn das Programm öfter verwendet wird

57. Wie lauten die wichtigsten Faustregeln im Zusammenhang mit Klassenzusammenhalt und Objektkopplung? Welche Vorteile kann man sich davon erwarten, dass diese Faustregeln erfüllt sind?
    Der Klassenzusammenhang soll hoch sein & Die Objektkopplung soll schwach sein gute Faktorisierung, Wahrscheinlichkeit geringer, dass bei Programmänderung auch die Zerlegung in Klassen und Objekte geändert werden muss.

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

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

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

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

59. Welche Rolle spielen Refaktorisierungen in der Wiederverwendung?
Refaktorisierungen ermöglichen das Hinführen des Projektes auf ein stabiles gut faktorisiertes Design.
Gute Faktorisierung => starken Klassenzusammenhalt => gut abgeschlossene und somit leicht wiederverwendbare Klassen. Sie ändert die Struktur eines Programms, lässt aber dessen Funktionalität unverändert.
Faustregel: Ein vernünftiges Maß rechtzeitiger Refaktorisierungen führt häufig zu gut faktorisierten Programmen und dadurch zu stabilen Klassen die für Vererbung und Untertypbeziehungen wichtig sind.

60. Wofür ist die objektorientierte Programmierung gut geeignet, und wofür ist sie nicht gut geeignet?
Wenn Algorithmen zentral im Mittelpunkt stehen ist sie nicht gut geeignet. OOP steht die Datenabstraktion im Mittelpunkt, aber Algorithmen müssen unter Umständen aufwendig auf mehrere Objekte aufgeteilt werden. Das kann den Entwicklungsaufwand von Algorithmen erhöhen und deren Verständlichkeit verringern. Faustregel: Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.