WS 2011: Grundlagen der Programmkonstruktion

Kontrollfragen - 3. Test

Weitere Ausarbeitung: https://docs.google.com/document/d/1HJbR-lPFZJkGZOe9b2EArbOaDsaKCh0fEmaoml4l0Sc/edit

• Was versteht man unter einer Spezifikation?
Jede Form einer mehr oder weniger rigorosen Beschreibung eines Systems ist eine Spezifikation. Dazu zählt auch die Beschreibung des Designs und die Implementierung selbst. Auch Zusicherungen zählen dazu.
(Skriptum S. 301)

• Wie genau soll eine Spezifikation sein? Von welchen Faktoren hängt diese Genauigkeit ab?
Ursprünglich muss eine Spezifikation nicht ganz genau sein, aber im Laufe der Zeit, mit der Entwicklung des Programms wird die Spezifikation immer genauer durch Annahmen der Programmierer und durch Konsultation mit dem Auftraggeber. Durch Letzeres wird auch sicher gestellt, dass durch diese Verbesserungen der Genauigkeit der Vertrag nicht gebrochen wird.

• Auf welche Arten kann man ein Programm spezifizieren?
Man kann das Programm informell durch Kommentare spezifizieren, oder formal mit assert-Anweisungen.

• Wie geht man vor, wenn man Widersprüche oder Ungenauigkeiten in einer Spezifikation entdeckt?
Man fragt nach bzw. berät sich mit den Auftraggebern/Urhebern?!
Ist ein Nachfragen nicht möglich oder nicht sinnvoll, nimmt man eine der möglichen Auslegungen des Widerspruchs an oder spezifiziert eine Ungenauigkeit genauer. In jedem Fall sollte man solche Annahmen/Auslegungen dann dokumentieren!

• Was versteht man unter Design by Contract?
Die Vorgehensweise, bei der man Klassen und Systeme durch Verträge zwischen Clients und Server beschreibt, nennt man Design by Contract.
Ziel ist das reibungslose Zusammenspiel einzelner Programmmodule durch die Definition formaler „Verträge“ zur Verwendung von Schnittstellen, die über deren statische Definition hinausgehen. Der Vertrag selbst besteht aus Invarianten sowie Vor- und Nachbedingungen.

• Welche Bestandteile eines Softwarevertrags sind vom Client zu erfüllen, welche vom Server?

Der Client kann durch Senden einer Nachricht an den Server einen angebotenen Dienst in Anspruch nehmen, wenn
• die Schnittstelle des Servers eine entsprechende Methode beschreibt,
• jeder Argumenttyp in der Nachricht Untertyp des entsprechenden formalen Parametertyps der Methode ist
• und alle Vorbedingungen der Methode erfüllt sind.
Der Server wird unter diesen Bedingungen die Methode ausführen und sicherstellen, dass unmittelbar nach Ausführung
• eine Instanz des Ergebnistyps als Antwort zurückkommt
• und alle Nachbedingungen der Methode und Invarianten des Servers erfüllt sind.
(Skriptum S. 304/305)
Client: Vorbedingungen fürs Ausführen einer Methode erfüllen.
Server: Invarianten und Nachbedingungen. Methode muss das gewünschte Ergebnis zurückgeben.

• Woran erkennt man, ob eine Bedingung eine Vorbedingung, Nachbedingung oder Invariante darstellt?

• Einschränkungen auf formalen Parametern, sowie alles, um das sich Aufrufer von Methoden kümmern müssen, sind Vorbedingungen.
• Beschreibungen unveränderlicher Eigenschaften von Objektzuständen stellen Invarianten dar.
• Alles andere sind Nachbedingungen. Sie beschreiben, was die Methoden tun, und machen meist den Großteil der Zusicherungen aus. Nachbedingungen müssen vom Aufgerufenem (der Funktion / Methode) geprüft werden.
(Skriptum S. 306 - 307)

• Wie müssen sich Vor- und Nachbedingungen bzw. Invarianten in Unter- und Obertypen zueinander verhalten?
• Vorbedingungen in Untertypen dürfen schwächer, aber nicht stärker als entsprechende Bedingungen in Obertypen sein. Wenn eine Vorbedingung im Obertyp beispielsweise x > 0 lautet, darf die entsprechende Bedingung im Untertyp x >= 0 sein. Im Untertyp steht die Verknüpfung beider Bedingungen mit ODER.
• Nachbedingungen und Invarianten in Untertypen dürfen stärker, jedoch nicht schwächer als die Bedingungen in Obertypen sein. Wenn eine Nachbedingung oder Invariante im Obertyp z.B. x >= 0 lautet, darf die entsprechende Bedingung im Untertyp x > 0 sein. Im Untertyp steht die Verknüpfung beider Bedingungen mit UND.
(Skriptum S. 306)

• Warum stellen Invarianten einen Grund dafür dar, dass man keine public Variablen verwenden sollte?

Wenn Methoden von anderen Objekten, die die Invarianten des Servers nicht kennen, zu diesen Variablen schreibenden Zugriff haben (die Variablen sind public), hat die Server-Methode keine vollständige Kontrolle über den Zustand des Objekts und kann deshalb die Invarianten nicht zusichern.

• Inwiefern lassen sich Bedingungen statt als Zusicherungen auch in Form von Typen ausdrücken?

Einige Zusicherungen lassen sich als Typen ausdrücken. Es ist vorteilhaft, wenn möglich, Zusicherungen mit Typen zu ersetzen, weil der Compiler die Kompatibilität der Typen zueinander statisch überprüft und einen Fehler meldet, wenn Typen nicht zusammenpassen. Typen können auch im Gegensatz zu Kommentaren nicht altern.

• Welche Rolle spielen Namen in Programmen (im Zusammenhang mit Zusicherungen)?

Gut gewählte Namen von Klassen, deren Methoden und Variablen sind eine Form von Zusicherungen, weil sie das statische Verständnis des Programms verbessern. Sie machen das Programm lesbarer und können Auskünfte über geforderte Datentypen geben.

• Was versteht man unter der Namensgleichheit bzw. Strukturgleichheit von Typen?

• Typäquivalenz aufgrund von Namensgleichheit bzw. explizite Untertypbeziehungen: Zwei Typen sind genau dann gleich, wenn die Typen dieselben Namen haben. Zwei Typen stehen genau dann in einer Untertypbeziehung, wenn eine explizite Beziehung zwischen den Namen dieser Typen (durch extends- und implements-Klauseln) hergestellt wurde.

• Typäquivalenz aufgrund von Strukturgleichheit bzw. implizite Untertypbeziehungen: Zwei Typen gelten als gleich, wenn ihre Instanzen dieselbe Struktur haben (unabhängig von Typnamen). Zwei Typen sind in Untertypbeziehung, wenn ein Typ zumindest alle Methoden unterstützt, die auch der andere unterstützt (auch ohne extends- oder implements-Klausel). Dadurch können Zusicherungen nicht geprüft werden.

• Was ist Duck Typing, was ist das Gegenteil davon?

• Wenn ein Objekt zumindest alle Methoden hat, die man von einer Instanz von Duck erwartet, dann kann das Objekt als Instanz von Duck verwendet werden.

• Das Gegenteil von Duck Typing verwendet man in stark typisierten Programmiersprachen:

Ein Objekt ist nur dann eine Instanz von Duck, wenn es durch new X(...) erzeugt wurde, wobei X gleich Duck oder ein explizit deklarierter Untertyp von Duck ist (beispielsweise durch class X extends Duck {...}).

Laut Wikipedia:
Duck-Typing ist ein Konzept der objektorientierten Programmierung, bei dem der Typ eines Objektes nicht durch seine Klasse beschrieben wird, sondern durch das Vorhandensein bestimmter Methoden.

Eventuell Verständlicher:

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck. Kurz: Egal, was die Klasse ansich darstellt - sofern sie alle Methoden darstellt, die ich für meine Absichten des Objekts benötige kann der Vogel als Ente angesehen werden - da die Klasse (der Vogel) das macht, was meine Ente können soll: So laufen, Quaken und Schwimmen wie eine Ente. Quak.

• Warum soll man Programme eher statisch als dynamisch verstehen?
Statisches Verstehen erleichtert das Programmverständnis, weil wir die unveränderbaren Aspekte des Programms einem nach dem anderen betrachten, und nicht alle gleichzeitig wie beim dynamischen Verstehen.

• Wozu verwenden wir assert-Anweisungen?
assert-Anweisungen verwenden wir, um die Zusicherungen auf eine eindeutige, formale Basis zu stellen und auch während der Laufzeit zu überprüfen.

• Sind assert-Anweisungen auch sinnvoll, wenn deren Überprüfung ausgeschaltet ist? Warum?
assert-Anweisungen sind auch dann sinnvoll, weil sie noch immer das statische Programmverständnis verbessern und damit stellen sie gültige Zusicherungen dar.

• Was ist eine Schleifeninvariante? Wozu verwenden wir sie?
Schleifeninvarianten sind Eigenschaften, die zu Beginn und am Ende eines jeden Schleifendurchlaufs erfüllt sein müssen. Wir verwenden sie als Zusicherungen und mithilfe der Schleifeninvarianten können wir Schleifen besser statisch verstehen.

• Wann muss eine Schleifeninvariante gelten?
Eine Schleifeninvariante muss zu Beginn und Ende jedes Schleifendurchlaufs gelten. Trotzdem kann sie im Schleifenrumpf kurzfristig verletzt sein, so lange sie am Ende des Schleifenrumpfs gilt.

• Wie gehen wir vor, um geeignete Schleifeninvarianten zu finden?
Bei der Suche nach Schleifeninvarianten geht man oft von den Nachbedingungen aus, die am Ende der Methodenausführungen gelten müssen. Sie liefern gute Hinweise darauf, welche Schleifeninvarianten benötigt werden, um das Programm verifizieren zu

können. Dann sucht man nach einer Möglichkeit, diese Nachbedingungen zu erfüllen.

• Wann ist der richtige Zeitpunkt um Zusicherungen (insbesondere auch Schleifeninvarianten) in den Programmcode zu schreiben?
Die zugehörigen Zusicherungen zum Programmcode sollen wir schon, bevor wir den Programmcode schreiben, aufstellen. Damit können wir von Anfang an eine statische Denkweise annehmen. So können wir das Programm besser statisch verständlich schreiben.

• Welche Formen von Zusicherungen ersetzen Schleifeninvarianten bei Verwendung von Rekursion statt Iteration?
Bei Rekursionen, statt Schleifeninvarianten verwenden wir jedoch Vor- und Nachbedingungen bzw. Invarianten auf Objektschnittstellen.

• Wodurch unterscheidet sich die partielle von der vollständigen Korrektheit eines Programms?
Die Einhaltung aller Zusicherungen garantiert die partielle Korrektheit des Programms, bei der alle Ergebnisse den Spezifikationen entsprechen, falls das Programm Ergebnisse liefert. Termination ist dafür nicht erforderlich. Vollständige Korrektheit erweitert die partielle Korrektheit um Termination und garantiert damit, dass das Programm Ergebnisse liefert, die den Spezifikationen entsprechen.

• Wann terminiert eine Schleife oder Rekursion?
Eine Schleife terminiert, wenn die Schleifenbedingung nicht länger erfüllt ist. Eine Rekursion terminiert, wenn es zu einem Zustand kommt, in dem keine weitere rekursiven Aufrufe gemacht werden können.

• Was muss man zeigen, um die Termination formal zu beweisen?
Um die Termination einer Schleife zu beweisen, müssen wir den Fortschritt pro Schleifendurchlauf abhängig von Eingabewerten in Zahlen fassen. Damit können wir eine von der Eingabe abhängige obere Schranke für die Anzahl der Schleifendurchläufe berechnen. Diese Berechnung braucht nicht genau zu sein, eine grobe Abschätzung reicht. ABER(!) es darf bei der “groben Abschätzung” nur eine zu hohe Schranke, nie aber eine zu niedrige Schranke gewählt werden. Es reicht also “eine” obere Schranke zu finden (die aber wirklich eine ist, also keine Zahl die drüber geht). Im Prinzip irgendeine. Dann Terminiert die Schleife/Rekursion.

• Gibt es praktische Unterschiede zwischen der Berechnung des zeitlichen Aufwands und dem Beweis der Termination? Wenn ja, welche?
Die Berechnung einer Schranke für die Anzahl der Schleifendurchläufe hat viel mit der

Aufwandsabschätzung eines Algorithmus im schlechtesten Fall gemeinsam. Trotzdem müssen wir beim Beweis der Termination nur den minimalen Fortschritt finden, und so können wir für eine Termination eine viel größeren Anzahl an Schritten erhalten als bei einer Aufwandabschätzung im schlechtesten Fall.
Beispiel: Binäre Suche: Maximallaufwand = log(n);
Termination = n;

• Spielen Terminationsbeweise auch in Programmen, die niemals terminieren sollen, eine Rolle?
Ja, weil auch Programme die niemals terminieren, müssen immer ein Fortschritt machen und mit jedem Schleifendurchlauf eine Teilaufgabe lösen.

• Inwiefern hängen Korrektheitsbeweise mit dem statischen Verstehen eines Programms zusammen?
Formale Beweise der Programmkorrektheit und ein statisches Programmverständnis gehen Hand in Hand. Wir verstehen ein Programm, indem wir uns überlegen, wie wir dessen Korrektheit beweisen können. Umgekehrt setzt ein Korrektheitsbeweis auch ein gutes Programmverständnis voraus.

• Was kann man mittels Model Checking machen?
Mittels Model Checking kann man mathematische und logische Eigenschaften eines Programms ganz automatisch überprüfen.

wiki: **Model Checking** (deutsch auch *Modellprüfung*) ist ein Verfahren zur vollautomatischen Verifikation einer Systembeschreibung (Modell) gegen eine Spezifikation (Formel).

• Was versteht man unter Denial-of-Service-Attacken?
Absichtliches, böswilliges Senden einer Unzahl an Anfragen mit dem Ziel, das System zu überlasten. Das Verhindern von DoS Attacken steht im Konflikt von dem Wunsch nach einer schnellen Reaktionszeit und einfacher Bedienung des Programms.

• Kann man die Fehlerfreiheit eines Programms sicherstellen? Wenn ja, wie?
Fehlerfreiheit kann nur durch formalen Beweisen sichergestellt sein.

• Ist es besser, beim Testen viele Fehler zu finden als wenige? Warum?
(Streitfrage!) Es ist besser wenige Fehler zu finden, weil wenn man viele Fehler findet, bedeutet das meistens, dass es noch mehr unentdeckte Fehler im Programm gibt. Außerdem führt das Beseitigen eines gefundenen Fehlers leicht zu anderen Fehlern, die wir unabsichtlich im Programm einbauen.

(wie es im Skriptum steht: wenn man zwei Programme mit der selben intensität laufen

lässt, beinhaltet das Programm bei dem mehr Fehler gefunden wurden, vermutlich auch noch mehr versteckte Fehler. Die Frage hier ist also zu allgemein gestellt, aber es lässt sich vermuten, dass beim (unachtsamen) programmieren einfach eine höhere Fehlerdichte zustande gekommen ist, wenn am Schluss viele Fehler gefunden werden)

• Wie können beim Beseitigen eines Fehlers neue Fehler entstehen?
Wenn wir einen Fehler rasch beseitigen ohne nachzudenken, ob die Fehlerursache wirklich an dieser Stelle ist, gibt es die Möglichkeit, dass wir nur ein Symptom des Fehlers beseitigen. Aber da an dieser Stelle kein tatsächlicher Fehler war, haben wir nur ein korrektes Programmteil geändert. Wenn die eigentliche Fehlerursache später entdeckt und beseitigt wird, stellt sich diese Ausbesserung plötzlich als neuer Fehler dar. Außerdem kann es vorkommen, dass man durch die Änderung eine Zusicherung verletzt.

• Soll man auch Fehler korrigieren, bei denen die Wahrscheinlichkeit für ein zufälliges Auftreten des Fehlers äußerst gering ist? Warum?
Ja man soll diese Fehler, falls entdeckt, immer beseitigen, weil sie ansonsten für Systemeinbrüche genutzt werden können. Auch wenn dies nicht der Fall ist kann dieser unwahrscheinliche Fall bei langer Laufzeit / vielen Aufrufen auftreten und evtl. folgeschwere Probleme hervorrufen. *(siehe MP3-Player / year Beispiel aus UE4)*

• Aus welchen Gründen könnte jemand das Eindringen in ein System absichtlich ermöglichen?
Es kommt vor, dass das Eindringen in ein System absichtlich ermöglicht wird, beispielsweise um das Testen und die Suche nach Fehlerursachen zu vereinfachen. Das ist aber sehr gefährlich und soll vermieden werden.
Natürlich ist es auch möglich, dass dies aufgrund krimineller Energie geschieht (in dem Fall hat man dann wohl seine guten Gründe und es ist aus Sicht des Programmierers wünschenswert ;]).

• Wodurch führt Testen zu einer Qualitätsverbesserung?
Testen führt zu einer indirekten Qualitätsverbesserung dadurch, dass uns beim Testen aufgedeckte Fehler gute Hinweise darauf geben, in welchen Bereichen wir die Qualität über andere Mittel verbessern müssen.

• Was versteht man unter Code Reviews?
Bei einem Code Review begutachtet ein erfahrener Reviewer den Quellcode eines Programms bzw. Programmteils, stellt Fragen und macht Verbesserungsvorschläge dazu. Über Code Reviews werden Fehler aus ganz unterschiedlichen Bereichen gefunden. Code Reviews stellen, im Gegensatz zum Testen, ein statisches Verfahren zur Qualitätsverbesserung dar.

• Welche Ziele, Gemeinsamkeiten und Unterschiede gibt es zwischen Unit-, Integrations-, System- und Abnahmetests.
Bei einem Unittest testet man die Funktionalität eines klar abgegrenzten Programmteils (z.B. einer einzelnen Klasse).

Beim Integrationstest wird die korrekte Zusammenarbeit zwischen den Units, die zuvor schon über Unittests einzeln getestet wurden, überprüft.

Beim Systemtest wird das gesamte System hinsichtlich der Erfüllung aller geforderten funktionalen und nichtfunktionalen Eigenschaften überprüft.

Der Abnahmetest überprüft, ob das gesamte System auch unter realen Bedingungen mit realen Daten seine Aufgaben erfüllt. Nach Bestehen dieses Tests geht das System tatsächlich in Betrieb.

Das Ziel all dieser Tests ist es, Fehler im Programm zu finden. Alle Tests machen das selbe, nur auf verschiedenen Ebenen.

• Was versteht man in der Informatik unter einem Stresstest?
Ein Stresstest überprüft das Verhalten eines Systems unter Ausnahmebedingungen. Es gibt zahlreiche Varianten wie den Crashtest, bei dem man versucht, das System zum Absturz zu bringen, und einen Lasttest, bei dem man das Verhalten eines (etwa durch viele gleichzeitige Benutzer) über die Grenze belasteten Systems testet.

• Was charakterisiert White-, Grey- und Black-Box-Tests?
Beim Black Box Test verwendet man keinerlei Wissen über die interne Realisierung von Programmdetails.

Beim White Box Test basieren Testfälle auf der internen Struktur des Programms. Wir entwickeln die Testfälle zusammen mit dem Programm.

Beim Grey Box Test entwickeln wir Testfälle zur genauen Spezifikation des Programms noch vor dessen Implementierung, also ist er eine Kombination aus dem Black und White Box Test.

• Wofür stehen bei Laufzeitmessungen real-, user- und sys-Werte?
real - Zeit, die das Programm zur Ausführung benötigte (quasi mitstoppen mit Uhr)

user - Zeit, die der Prozessorkern mit der Verarbeitung des Programms verbraucht hat (kann bei Multicore-Systemen größer als die Real-Time + Sys-Time sein sein, bei n-Core mit Faktor n)

sys - wieviel Zeit eines Prozessor-Kerns das Betriebssystem an Serviceleistungen für die Programmausführung erbracht hat

• Wovon hängen Laufzeiten ab, und wodurch unterscheiden sich gemessene Laufzeiten oft so stark voneinander?

Die gemessene Zeit hängt von zahlreichen Faktoren ab, beispielsweise Art und Menge der verarbeiteten Daten, anderer am Rechner gleichzeitig laufender Software und einer Unzahl an winzigen, undurchschaubaren Details in der Hard- und Software.

Genaue Gründe für die Unterschiede der Laufzeiten sind kaum zu finden. Mangels besserer Erklärung redet man sich gerne auf Cache Effekte aus, also Unterschiede in der Laufzeit, die dadurch verursacht werden, dass der Prozessor (meist wegen geänderter Speicheradressen) andere Datenmengen im Cache hält.

• Wodurch unterscheiden sich Latenz und Bandbreite voneinander, und wann dominiert einer dieser Begriffe den anderen?

Latenz (Verzögerung):
Zeit, die die Komponenten (CPU, Speicher,..) im PC brauchen um Daten auszutauschen. Zwischen Senden einer Anfrage und dem Erhalten der gewünschten Daten vergeht Zeit – die Latenzzeit.

Bandbreite:
Wie viel Daten pro Zeiteinheit übertragen werden können. Beim Übertragen von großen Blöcken auf einmal spricht man von einer großen Bandbreite.

Eine große Bandbreite ist jedoch nicht von großer Bedeutung, wenn viele kleine Datenblöcke benötigt werden. In diesem Fall spielt die Latenzzeit eine größere Rolle. (Wenn etwa viele kleine Blöcke von verschiedensten Adressen benötigt werden)

• Kann man gemessene Laufzeiten auf größere Datenmengen hochrechnen?
Nein, da ein Programm in der Regel aus einer Unzahl an Algorithmen besteht, die nicht zwangsläufig eine vergleichbare Laufzeit besitzen.

• Was versteht man unter weichen und harten Echtzeitsystemen?
In einem harten Echtzeitsystem hat eine Laufzeitüberschreitung fatale Konsequenzen, z.B. Airbag und Bremssystemen. In einem weichen Echtzeitsystem sind einzelne

Laufzeitüberschreitungen tolerierbar, z. B. Computerspiele.

• Was ist ein Stack Trace? Welche Informationen enthält er?

Am Beispiel einer Endlosrekursion (die zwangsläufig zu einem Stack-overflow führt, da endlos Rücksprungadressen usw. auf den Stack gespeichert werden) wird eine Exception ausgegeben mit dem sogenannten „Stack Trace“.
Dieser enthält wesentliche Informationen über den Stack (zum Zeitpunkt des Ausnahmezustandes). Hierbei handelt es sich nicht um unleserlichen Dump sondern um informative „Fehlermeldungen“.
Danach wird die Programmstelle, an der die Exception aufgetreten ist, angezeigt und die Abfolge der Methodenaufrufe die zum Punkt, an dem die Exception geworfen wird, führen.
http://de.wikipedia.org/wiki/Stacktrace

• Wie kann man mittels assert Anweisungen beim Auftreten von Fehlern etwas über den Programmzustand erfahren?

Indem man einen Wert / Debug Mitteilung nach einem Doppelpunkt übergibt, z.B.
assert x >= 0: “x = “ + x;

• Was versteht man unter Debug Output?

Für Troubleshooting ist es oft hilfreich beispielsweise Laufvariablen einer Schleife oder Zwischenergebnisse auszugeben. Dies dient jedoch nur der Fehlerfindung. Wenn diese abgeschlossen ist werden diese Outputs nicht mehr benötigt und sollten auskommentiert oder gelöscht werden.
Alternativ kann man sie auch mit einem if(debug) umklammern um alle Debug-Outputs jeder Zeit über eine boolsche variable ein-und ausschalten zu können.

• Wozu dienen Logdateien?

Debug Outputs, Zwischenergebnisse, Benutzereingaben, (unwesentliche) Fehlermeldungen oder Ähnliches können während des Programmablaufes (oder während des Debuggens) in eine sogenannte Logdatei (Protokoll-Datei) geschrieben werden. Sie ermöglicht es, beispielsweise nach dem Auftreten eines Fehlers, den Programmablauf nachvollziehen zu können und so zur Fehlerfindung beizutragen. Dies stört den Programmablauf kaum, der Benutzer bekommt nichts mit.

• Was kann man mit einem Debugger machen?

In Echtzeit in den Programmablauf eingreifen.
Man kann Haltepunkte setzen. Wenn man dies tut, wird das Programm, sobald es bei einem Haltepunkt (Breakpoint) angekommen ist, unterbrochen.
In diesem Zustand kann man:

- Variablenwerte auslesen und verändern
- Neue Haltepunkte hinzufügen/alte entfernen
- Programmausführung Schritt für Schritt fortsetzten (nach jedem Schritt

wird gehalten) und im Falle eines Methodenaufrufes wählen ob man am Anfang (step into) oder erst nach Rückkehr (step over) wieder anhalten möchte.

· Programmausführung generell fortsetzen (bis ggf nächsten Breakpoint)

• Warum ist das Finden von Fehlerursachen oft so schwierig?

Die Fehlerursache finden wir fast nie an der Stelle, an der sich der Fehler zeigt. Deshalb müssen wir die Fehlerursache immer wieder an einer vorherigen Stelle im Programmcode suchen, bis wir die tatsächliche Fehlerursache finden.

• Wie gehen wir beim Sammeln von Fakten zum Auffinden einer Fehlerursache vor?

Beim Sammeln von Fakten durchlaufen wir zyklisch folgende Schritte:

- Orientierung: Im ersten Schritt müssen wir uns zur Orientierung einen Überblick darüber verschaffen, in welchem Bereich der Fehler auftritt.
- Hypothese: Wir stellen eine Hypothese darüber auf, welche möglichst kleine und klar abgegrenzte Menge an Variablen, Methoden und Anweisungen an der Entstehung des Fehlers beteiligt sein könnte.
- Planung: Dann legen wir uns einen Plan zurecht, wie wir die interessanten Ausschnitte aus den Programmzuständen leicht verfolgen können.
- Durchführung: Nun sammeln wir die Daten wie geplant. Falls Probleme auftreten, müssen wir zurückgehen und die Planung oder die Hypothese überarbeiten.
- Auswertung: Schließlich werten wir die im vorigen Schritt gesammelten Daten aus.

(Diese Schritte sind detaillierter im Skriptum Seite 339-340 zu finden)

• Warum lassen sich Ursachen für Fehler, die vom Compiler gemeldet werden, häufig einfacher finden als die für andere Fehler?

Fehler die vom Compiler gemeldet werden beinhalten eine Referenz zur Code-Zeile. Der Fehler befindet sich meistens in dessen Umgebung. Bei anderen Fehlern weiß man bestenfalls nur, dass sie existieren.

Compiler verstehen das Programm allerdings nicht und erkennen daher nur relativ einfache Inkonsistenzen (zB Syntax) im Programm.

• Was versteht man unter einer Ausnahmebehandlung?

Unter Ausnahmebehandlung versteht man das Abfangen von Ausnahmen und das Behandeln dieser Ausnahme durch das Schreiben eines alternativen Programmzweigs als Ersatz für den unterbrochenen Zweig, damit das Programm weiterlaufen kann.

• Wie und warum fängt man Ausnahmen ab?
Ausnahmen werden mittels try-catch Blöcken abgefangen. Die Ausnahmen fangen wir ab, damit wir einen alternativen Programmzweig schreiben können, der diese Ausnahme behandelt. So wird der Programmdurchlauf von der Ausnahme nicht abgebrochen.

• Welche Ausnahmen können in Java immer an den Aufrufer propagiert werden, welche nicht?
Ausnahmen eines Typs, der ein Subtyp von Exception ist, müssen in der Methodensignatur mittels throws-Klausel deklariert werden(nicht ganz... Runtime Exception ist auch ein Subtyp von Exception) um an den Aufrufer weitergegeben zu werden, Ausnahmen eines Subtyps von Error und RuntimeException können immer an den Aufrufer propagiert werden.

• Wie wird nach dem passenden Exception Handler gesucht?
Die Ausnahme wird immer nach außen an den Aufrufer propagiert. Der erste passende Exception Handler, der so gefunden wird, wird ausgeführt.

• Wofür sind Ausnahmebehandlungen gedacht, wozu sollten sie eher nicht verwendet werden?
Ausnahmebehandlungen sind nur für echte Ausnahmesituationen gedacht, nicht für Situationen, die auch leicht durch andere Sprachkonstrukte beherrschbar sind.

• Wozu dienen die Schlüsselwörter throw und throws in Java, und wie verwendet man die entsprechenden Sprachkonzepte?
Die throw Anweisung benutzen wir, um eine Ausnahme für Ausnahmesituationen zu werfen, die vom System nicht erkennbar sind. Eine throw Anweisung sieht so aus:
throw new Exception ("Ursache für Ausnahme");
wobei eine neue Instanz von Exception erzeugt und geworfen wird.
Eine throws-Klausel besagt, dass im Rumpf einer Methode eine Ausnahme geworfen und an den Aufrufer weitergeleitet werden kann. Bsp:
public int test (String s) throws TestException

• Welche Schwierigkeiten können durch throws-Klauseln entstehen?
Nachteile von throws-Klauseln bestehen im höheren Programmieraufwand und in vermindeter Flexibilität. Wenn Ausnahmen über mehrere Aufrufebenen hinweg weitergeleitet werden sollen, müssen viele Methoden mit throws-Klauseln ausgestattet werden. Wenn wir eine Ausnahme später in der Programmentwicklung hinzufügen wollen, müssen wir vielleicht viele Methoden ändern.

• Was macht man mit finally-Blöcken? In welchen Situationen werden

sie ausgeführt?
In einem finally-Block steht der Programmcode zum Aufräumen von allem, was nach Ausführung des dazugehörenden try-Blocks aufgeräumt werden soll.
Finally Blöcke werden immer ausgeführt.

• Wofür sind bedingte Anweisungen besser geeignet als Zusicherungen?
Für Debug-Anweisungen sind bedingte Anweisungen besser geeignet. bsp:
`if(debug && !(x > 0)) System.out.println("Error - x = " + x);`
Auch alle Anweisungen die für die Funktion des Programms ausgeführt werden müssen, sollten nicht in asserts platziert werden. Asserts müssen nämlich zur Laufzeit nicht unbedingt aktiviert sein - im Gegenteil, in Normalfall sind sie inaktiv um Zeit zu sparen.
Ermöglichen von Fehlerbehandlungen (throw Exception()).

• Was unterscheidet Byte-Streams von Character-Streams in Java?
Byte Streams enthalten nur rohe Daten (Bytes), während Character Streams Zeichen eines Textes enthalten.

• Was unterscheidet gepufferte von ungepufferten Dateizugriffen?
Ungepufferte Zugriffe verwenden zum Lesen und Schreiben direkt die entsprechenden Befehle des Betriebssystems, während gepufferte Zugriffe aus einem bzw. in einen zwischengeschalteten Puffer schreiben und lesen und erst bei Bedarf den Puffer mit der Datei abgleichen.

• Wozu dient die Methode format in String?
Die Methode String.format() dient dazu, formatierte Strings zu erzeugen.
z.b: `String.format("Hello %s!", name);`

• Was ist eine Plausibilitätsprüfung?
Eine Plausibilitätsprüfung ist die Beurteilung, ob uns Daten als zuverlässig genug erscheinen, um damit weiterzurechnen, oder nicht.

• Wohin soll man den Code für Plausibilitätsprüfungen schreiben, wohin nicht?
An die Schnittstellen, an denen die Daten ins System übernommen werden bzw. in ein eigenes Modul (zB Klasse oder Paket), nicht über die Codebasis verteilt.

• In welchen Situationen soll man Plausibilitätsprüfung vornehmen, in welchen nicht?
Wenn die Eingabe direkt oder indirekt von einem Benutzer erfolgte, ist eine Plausibilitätsprüfung notwendig.
Wenn die gelieferten Daten dagegen von einem Programm stammen, dessen Funktion uns bekannt ist und dieses möglichwerweise die Daten bereits auf ihre Plausibilität geprüft hat, ist keine erneute Prüfung nötig.

• Wodurch ähneln Plausibilitätsprüfungen und Zusicherungen einander, wodurch unterscheiden sie sich?
Plausibilitätsprüfungen ähneln Zusicherungen, weil beide Bedingungen darstellen, die unverletzt bleiben müssen, damit das Programm wie erwartet funktionieren kann.
Die Unterschiede liegen darin, dass Zusicherungen in korrekt funktionierenden Programmen eigentlich niemals verletzt sein dürften, während man bei Plausibilitätsprüfungen immer mit einer Verletzung der Bedingungen rechnen muss. Zusicherungen sind außerdem im ganzen Programm verstreut, während Plausibilitätsprüfungen nur an Schnittstellen vorkommen. Zusicherungen sind im Normalfall ausgeschaltet, während Plausibilitätsprüfungen niemals ausgeschaltet werden dürfen.

• Was kann man mit der Validierung von Programmen bewirken?
Man überprüft, ob das richtige Programm entwickelt wurde, d.h. ob das Programm in der echten Anwendung für den vorgesehenen Zweck sinnvoll ist.