Testmethoden für sequentielle und nebenläufige Software-Systeme Zusammenfassung

Michael Gregorius

1. März 2003

http://www.michaelgregorius.de/

Vorwort

Diese Zusammenfassung wurde im Rahmen der Vorbereitung auf die dritte Hauptdiplomsprüfung (Inf3) im Studienfach Kerninformatik an der Universität Dortmund erstellt. Die Prüfung erfolgte am 17. Februar 2003 bei Eike Hagen Riedemann über die Themen ÜM, TM2 und TM3 aus seinem Buch *„Testmethoden für sequentielle und nebenläufige Software-Systeme“* ([RIEDEMANN 1997]) sowie dem Artikel *„Reviews in der objekt-orientierten Softwareentwicklung“* von Mario Winter ([WINTER 1997]).

Die Vorbereitungszeit lag bei ca. zwei Monaten, wobei ich auch nicht bestätigen kann, dass der Stoff in zwei Wochen lernbar wäre, wie ich es vor der Prüfung öfters gehört habe. Der Stoff ist im Prinzip nicht schwer, jedoch sehr umfangreich, da relativ viele Sachen einfach auswendig gelernt werden müssen.

Das Prüfungsklima ist allerdings wirklich so angenehm, wie es in vielen Protokollen steht. Ein Punkt, der dazu beträgt, dürfte der Vortrag zu Beginn der Prüfung sein. Denn dieser ist zum einen eine gute Gelegenheit, um sich warmzureden, zum anderen hat er bei mir dafür gesorgt, dass ich mich relativ schnell beruhigt hab. Die ersten zehn Minuten gehören also wirklich einem selbst, da Herr Riedemann nur zuhört, ohne einen zu unterbrechen. Dies sei hier auch nochmal erwähnt, da der Prüfling, der vor mir dran war, überrascht war, „dass man wirklich einen Vortrag halten muss“. Man sollte sich allerdings nicht davon beunruhigen lassen, dass der Protokollant zum Zeitpunkt des Vortrags kaum mitschreibt, das scheint normal zu sein. Allgemein sollte man sich von diesem Vortrag nicht verrückt machen lassen, denn im Prinzip besteht er lediglich aus einem Überblick über das dritte Kapitel im Buch. Man sitzt mit Herrn Riedemann an einem Tisch, erzählt das einfach alles runter (was nach ein paarmal üben locker klappt) und zeichnet dabei den „Testmethodenbaum“, den es auf seiner Homepage zum Download gibt. Die Stichpunkte meines Vortrags finden sich in Kapitel 14.

In Kapitel 13 findet sich eine kleine Zusammenfassung des Artikels, der in meiner Prüfung auch tatsächlich relativ ausführlich abgefragt wurde. Die anderen Kapitel haben alle ihre Entsprechungen im Buch und unterscheiden sich bezüglich dieser nur in den Kapitelnummern.

Ich habe auch noch zu einigen anderen Vorlesungen bzw. Prüfungen Zusammenfassungen geschrieben, welche auf meiner Homepage heruntergeladen werden können. Die Adresse lautet: `http://www.michaelgregorius.de/`. Auf dieser Seite findet sich auch eine Email-Adresse, unter der man mich erreichen kann. Über Kommentare, Meinungen und sonstige Rückmeldungen würde ich mich freuen. Viel Glück für die Prüfung!

Dortmund, den 1. März 2003 *Michael Gregorius*

Inhaltsverzeichnis

1 Beispiele für den Programmtest

Frage 1.1: Gebe ein Beispiel für ein sequentielles Programm!
Ein Beispiel für ein sequentielles Programm könnte ein **Textformatierer** sein, welcher eine Eingabe in Form von Wörtern bekommt, die durch Leerzeichen und Newlines getrennt sind und diese Eingabe so formatiert, dass:

1. Keine Zeile mehr als MAXPOS Zeichen enthält.
2. Zeilenumbrüche nur an Leerzeichen oder Newlines auftreten.
3. Die Anzahl der Wörter je Zeile maximal ist.

Diese Spezifikation ist informal und relativ ungenau. Sie kann wie folgt verfeinert werden, wobei teilweise die Fragen, die bei der obigen Spezifikation auftreten können, mit angegeben sind:

1. Was genau ist ein Wort? Ein Wort ist eine maximale Zeichenfolge, die keine Leerzeichen oder Newlines enthält.
2. Eine maximale Folge von Leerzeichen oder Newlines ist ist durch genau ein Leerzeichen zu ersetzen bzw. durch ein Newline, falls die maximale Folge am Ende einer auszugebenden Zeile auftritt. Ein Newline in der Eingabe spielt also dieselbe Rolle, wie ein Leerzeichen (also wie bei HTML).
3. Eine Folge von Newlines und Leerzeichen am *Anfang* der Eingabe ist ersatzlos zu streichen.
4. Bei der Berechnung der Zeilenlänge einer Ausgabezeile (zum Vergleich mit MAXPOS) wird das Newline am Ende *nicht* mitgezählt.
5. Was passiert, wenn ein Wort länger als MAXPOS ist? Dann wird die Bearbeitung abgebrochen.
6. An das Ende der Ausgabe wird das Zeichen EOT (End-of-Text) angehängt.

Frage 1.2: Gebe ein Beispiel für ein nebenläufiges Programms!
Ein Beispiel für ein nebenläufiges Programm ist das **Dining-Philosopher-Problem**. Nebenläufige Programme haben besondere Eigenschaften, welche wiederum zu bestimmten Fehlern führen können, die es bei sequentiellen Programmen nicht gibt. Dies sind:

- Blockierung von Teilprogrammen (Threads).
- Verhungern von Prozessen.

2 Grundlegende Problemstellungen und Lösungsansätze

2.1 Ursachen unangemessener und fehlerhafter Software

Frage 2.1: Welche Ursachen gibt es grob für Irrtümer und Fehler?
Grob kann zwischen zwei Ursachen bzw. Fehlerquellen unterschieden werden, wobei im Folgenden auf beide eingegangen wird:

- Austausch und Verarbeitung von Informationen (Kommunikation)
- Speichern von Informationen (Gedächtnisproblem)

Frage 2.2: Welche Hauptformen der Kommunikation können unterschieden werden?
Es kann zwischen drei Hauptformen der Kommunikation unterschieden werden:

intrapersonell: Unter interpersoneller Kommunikation wird die Informationsverarbeitung *innerhalb* einer Person verstanden, also das Denken und Wahrnehmen.

interpersonell: Als interpersonelle Kommunikation wird der Informationsaustausch *zwischen* zwei Personen bezeichnet.

mediengebunden: Unter dieser Art der Kommunikation wird der Austausch von Informationen zwischen einer (meist kleinen) Gruppe von **Kommunikatoren** (z.B. Betreiber einer Webseite) und einer (meist größeren) Gruppe von **Rezipienten** (z.B. Leser einer Webseite) verstanden.

Frage 2.3: Welche drei Stufen durchläuft die Informationen bei jeder Kommunikation?
Die Information durchläuft die folgenden drei Stufen bei der menschlichen und technischen Kommunikation:

Verschlüsselung: Die zu übertragende Information wird durch den Sender verschlüsselt, damit sie übermittelt werden kann (z.B. „Umwandlung“ von Gedanken in Sprache bzw. Laute).

Übermittlung: Die verschlüsselten Informationen werden durch das Medium übermittelt (Sprache wird in Form von Schwingungen durch Luft übertragen).

Entschlüsselung: Die Informationen werden durch den Empfänger entschlüsselt und interpretiert.

Auf jeder dieser Stufen kann dabei ein Fehler gemacht werden.

Frage 2.4: Welche Arten von Irrtümern kennst du?

Identitätsirrtum: Ein Identitätsirrtum liegt z.B. dann vor, wenn ein Programmierer der Variablen `netto` einen Wert zuweist anstatt der Variablen `brutto`. Diese Art von Irrtum ist ein Beispiel für einen Irrtum auf der intrapersonellen Ebene. Fehler bei der intrapersonellen Kommunikation können z.B. durch Übermüdung, Stress, mangelnde Motivaton oder auch nicht ausreichenden Informationen entstehen.

Erklärungsirrtum: Diese Art von Irrtum liegt vor, wenn nicht das gesagt oder geschrieben wird, was eigentlich gemeint ist.

Übermittlungsirrtum: Die Information wird inkorrekt übermittelt. Das Spiel „Stille Post“ gewinnt durch diese Art von Irrtum seinen Reiz.

Entschlüsselungsirrtum: Bei dieser Art von Irrtum, der, wie der Name schon sagt, während der Phase der Entschlüsselung geschieht, kann weiter differenziert werden:

1. Die Information wird vordergründig falsch gehört bzw. gelesen.
2. Die Information wird richtig aufgenommen, jedoch falsch interpretiert.

Inhaltsirrtum: Darunter wird ein Irrtum über qualitative oder quantitative Eigenschaften der Software verstanden. Er kann durch fehlerhafte oder auch durch fehlende Kommunikation entstehen. Ein Beispiel dazu findet sich in [Riedemann 1997] auf Seite 23: Für die Computerfachleute ist der *erste* Satellit der obere, für die Verkablungsfachleute der untere.

Frage 2.5: Was versteht man unter Gedächtnisproblemen?

Unter Gedächtnisproblemen versteht man die Probleme beim Speichern von Informationen, die zu Fehlern in Software-Systemen führen. So muss beim Programmieren oft versucht werden, wie ein Computer zu denken. Da der Kontroll- und Datenfluss in einem Programm aber oft von verschiedenen Variablen abhängt, muss ein Entwickler sich diese alle merken, wenn er das Programm im Kopf durchgeht. Für einen Computer stellt dies meist kein Problem dar, für einen Menschen ab einer bestimmten Anzahl von Variablen jedoch schon und so kommt es zu Fehlern. Besonders deutliche wird dies in **nebenläufigen Programmen**, die sich oft durch eine außerordentliche **Komplexität** auszeichnen.

Frage 2.6: Gebe ein Beispiel in dem mangelndes Problemverständnis zu Fehlern führen kann!

Unter mangelndem Problemverständnis vesteht man „Unklarheit(en) in den Anwenderwissenschaften, die beim Versuch der Software-Erstellung erst zutage treten, in Wirklichkeit aber schon vorher existierten“. Ein Beispiel ist die Erstellung einer Schach-Software, bei der das strategische Vorgehen in der mittlere Phase des Spiels fehlt. In diesem Fall kann eine Lösung auch nicht eindeutig angegeben werden, es gibt jedoch auch Fälle in denen eine angegeben werden könnte, die jedoch dem Programmierer nicht bekannt ist.

2.2 (Manifestation von Softwarefehlern)

Frage 2.7: Wie viele Fehler werden in einem Softwaresystem ungefähr erzeugt?

Bezüglich dieser Frage wurden verschiedene Studien angestellt. Sie ermittelten eine Fehlerrate von 30 bis 100 Fehlern pro 1000 Anweisungen.

2.3 (Fehlverhalten von Software und seine Kosten)

Das Fehlverhalten von Software hat natürlich aus Auswirkungen. Diese können technischer, ökonomischer, gesellschaftlicher und politischer Art sein und in der Bewertung ihrer Schwere unterschiedlich klassifiziert werden.

Frage 2.8: Gebe eine mögliche Klassifizierung der Schwere von Softwarefehlern!

Die Schwere von Softwarefehler könnte z.B. wie folgt eingeteilt werden:

kritisch: Kompletter Ausfall der Produktion bzw. Aufgabe nicht mehr erfüllbar (z.B. Explosion einer Ariane 5-Rakete aufgrund eines Overflow).

hoch: Beeinträchtigung der Produktion, Leistung herabgesetzt.

mittel: Programmmöglichkeiten können nicht voll ausgenutzt werden, jedoch kann dies kompensiert werden.

niedrig: Geringe oder „kosmetische“ Probleme, Leistung bleibt voll erhalten.

unproblematisch: Nicht geforderte Softwareverbesserung.

2.4 Vermeidung und Behebung von Fehlern

Frage 2.9: Gesetzt die Ursachen aus Kapitel 2.1 treffen zu. Was müsste getan werden, um Fehler zu vermeiden bzw. zu reduzieren?

Wenn es sich bei den in Kapitel 2.1 aufgezeigten Ursachen tatsächlich um die Hauptursachen für Software-Fehler handelt, so müssten im Prinzip nur entsprechende Gegenmaßnahmen ergriffen werden, wie z.B.:

Kommunikation verbessern: Die im Projekt verwendeten Kommunikationsmethoden müssten verbessert werden und eventuell vorhandene Kommunikationsblockaden abgebaut werden. Eine funktionierende Kommunikation setzt unter anderem eine **gemeinsame Erlebniswelt** voraus. Diese kann z.B. durch die Erstellung von **Prototypen** geschaffen werden, die Leute reden also nicht über irgendetwas abstraktes, sondern über etwas konkret vorliegendes.

Intrapersonelle Kommunikation: Die Motivation des Personals müsste verbessert werden und Ermüdung und Stress abgebaut werden.

Gedächtnisproblem: Das Gedächtnisproblem könnte mit Hilfe von **besseren Entwicklungswerkzeugen**, wie z.B. syntaxgesteuerte Editoren und ausführlichen Compilermeldungen, bekämpft werden.

Anwendungs- und Problemverständnis: Das Anwendungs- und Problemverständnis, das Problemlösungsverhalten sowie das allgemeine Verständnis der Entwicklungsumgebung (verwendete Programmiersprache, Hard- und Software) müssten verbessert werden.

Beherrschung der Komplexität: Nebenläufige Systeme oder Systeme mit mehreren Prozessoren müssten gemeistert werden.

Die hier beschriebenen Ansätze können Fehler jedoch nur reduzieren und nicht vollständig ausmerzen.

Frage 2.10: Welche Faustregel gilt für die Kosten von Fehler in Bezug auf die Entwicklungsphase, in der sie entdeckt werden?

Allgemein gilt, dass Fehler natürlich **so früh wie möglich** entdeckt werden sollten. Müssen zu einem relativ späten Zeitpunkt z.B. Fehler in der Anforderung behoben werden, so wird dies im Allgemeinen teurer als ein Fehler, der erst bei der Implementierung entstanden ist. Dabei kann unterschieden werden zwischen Fehlern in der Anforderung und in der Systemspezifikation sowie Fehlern, die in der Code-Phase entstehen.

Frage 2.11: Da nicht alle Fehler verhindert werden können, müssen sie während jeder Phase gesucht werden. Wie kann dies geschehen?

Eine Anforderungsdefinition kann durch ein gedankliches Durchgehen und Durchspielen der Konsequenzen überprüft werden. Dieses Vorgehen wird auch **Review** genannt. Bei einer Systemspezifikation kann ähnlich vorgegangen werden. Wenn es möglich ist, kann dieses auch durch eine **Simulation** des Systemverhaltens geprüft werden. Da diese Reviews jedoch den allgemeinen Kommunikationsproblemen unterliegen, können auch hier nicht alle Fehler gefunden und beseitigt werden.

Frage 2.12: Was wird unter der Verifikation eines Programms gegenüber einer Systemspezifikation verstanden?

Da Reviews nicht alle Fehler in der Anforderung und Systemspezifikation finden können, bleibt die Hoffnung, wenigstens Programme zu erstellen, die gegenüber der Spezifikation selbst korrekt sind. Der Beweis, dass ein Programm die Systemspezifikation erfüllt, wird **Verfikation** genannt. Dabei treten jedoch Probleme auf:

- Fehler im Compiler, im Betriebssystem oder in der Hardware werden nicht berücksichtigt.
- Ein fehlerhaftes Programm kann nicht als korrekt bewiesen werden und ein Beweis liefert auch nur begrenzte Informationen über die Fehlerursache.

Frage 2.13: Welche zwei Formen der Verifikation eines Programms können grob unterschieden werden?

- Laufen lassen des Programms mit Testdaten und Vergleich der Ergebnisse mit der Spezifikation.
- Inspektion des Quellcodes.

Frage 2.14: Was ist ein idealer Test?

Es sei ein Programm gegeben, welches n Fehler enthält. Dann würde es ausreichen, diese Programm mit n Testdaten zu testen, die jeweils einen dieser Fehler aufdecken. Ein solcher Test wird als ein **idealer Test** bezeichnet. Leider gibt es keinen Algorithmus, der zu einem Programm und einer funktionalen Spezifikation einen idealen Test berechnet.

Frage 2.15: Was ist ein erschöpfender Test?

Praktische Programme rechnen mir endlichen Zahlenwerten. Daher kann ein Programm theoretisch mit **allen möglichen Eingaben** getestet werden. Ein solcher Test wird **erschöpfender Test** genannt. In der Praxis ist ein solcher Test jedoch meist ungünstig. Ist zum Beispiel ein Programm gegeben, welches drei 16-Bit große Integer als Eingabe bekommt, so würde ein erschöpfender Test ca. 892 Jahre benötigen, wenn ein Test in 0.0001 Sekunden durchgeführt werden könnte:

$$\frac{2^{3\cdot 16}}{86400 \cdot 365} \cdot 0.0001 = \frac{2^{48}}{86400 \cdot 365} \cdot 0.0001 \approx 892.551$$

Frage 2.16: Was versteht man unter der „Psychologie des Testens“ und welche Konsequenzen ergeben sich aus ihr?

Das Auffinden von Fehlern stellt für das Programmierteam ein **psychologisches Problem** dar, da sie ihre Arbeit als einen konstruktiven Prozess wahrnehmen. Das Testen dagegen wird als ein destruktiver Prozess wahrgenommen, denn es gilt ja die Schwachstellen im Code zu finden. Nach Gruenberger macht ein Programmierer dabei die folgenden Phasen durch:

Optimismus: Testen ist nicht nötig; ich weiß, dass meine Programme perfekt sind.

Bockigkeit: Hoppla, mein „perfektes“ Programm ist falsch, allerdings will ich meine Programme nicht testen.

Dienst nach Vorschrift: Ich gebe kein Programm mehr für die Produktion frei, sondern teste und teste nur noch.

Kompromissbereitschaft: Es gibt einen goldenen Mittelweg. Die Kunst des Testens ist es, zu wissen, wann mit dem Testen aufgehört werden muss.

Es ist also wichtig, dass die Programmierer psychologisch geschult werden, **selbstlos zu programmieren** und zu testen. Desweiteren müssen Fehler anders bewertet werden. Ein Fehler soll nicht als ein charakterliches Merkmal der

produzierenden Menschen angesehen werden, sondern als ein Makel im Produkt, den es zu beseitigen gilt. Fehler sollen also **vorteilhaft attribuiert** werden.

Man könnte ebenfalls auf die Idee kommen, lieber zu zeigen, dass ein Programm funktionert, anstatt es destruktiv zu testen, jedoch ist dies nicht sehr erfolgversprechend.

Aus der Psychologie des Testens kann auch gefolgert werden, dass **eine Person nicht ihr eigenes Programm testen sollte**. Ein weiteres Argument hierfür sind eventuelle Missverständnisse bezüglich der Spezifikation. Wird die Spezifikation falsch interpretiert, so würde das Testen durch den Programmierer nichts bringen. Dies liegt daran, dass die Ergebnisse des Testens als richtig interpretiert werden würden, auch wenn sie nicht den von der Spezifikation geforderten Ergebnissen entsprechen.

Frage 2.17: Welche Organisationsschemata zum Testen kennst du?
Es können drei Schemata unterschieden werden:

- Test durch eine **Untergruppe des Projekts** bzw. durch eine **unabhängige Gruppe** in der Firma.
- Testen als **Funktion des Kunden (Akzeptanztest)**.
- Testen durch einen **unabhängigen Vertragspartner**. Dies wird auch **unabhängige Verfikation und Validierung** genannt.

Dabei ist die personelle Trennung der Programmier- und Testaufgabe jedoch umstritten, da hierzu eine **gute Programmdokumentation** und **objektive Kriterien für Mängel- und Fehlerfreiheit** nötig sind. Wird ein Programm durch den Programmierer selbst getestet, so sollten die Testergebnisse jedoch zusätzlich von einer unabhängigen Person begutachtet werden.

Frage 2.18: Was versteht man unter einer Inspektion des Quellcodes? Welche Täuschungen können dabei vorkommen?
Wie etwas weiter oben schon erwähnt wurde, besteht eine weitere Möglichkeit, ein Programm zu verifizieren, darin, den Quellcode zu inspizieren. Es wird also anhand des Quellcodes selbst überprüft, ob das Programm das in der Spezifikation geforderte erfüllt. Jedoch treten auch bei diesem Vorgehen wieder **Kommunikations-, Konzentrations- und Wissensprobleme** auf, so dass auch hier höchstwahrscheinlich nicht alle Fehler gefunden werden. Desweiteren können verschiedene Arten der Täuschungen auftreten:

Trick- oder Ablauftäuschung: Hierbei weicht der Programmtext von dem ab, was „normale“ Programmierer denken (z.B. Ausnutzung von Seiteneffekten).

Erwartungstäuschung: Diese Art der Täuschung entsteht durch eine bestimmte Erwartung mit der eine Peron an eine Aufgabe (in diesem Fall das Testen) herangeht. So kann es z.B. passieren, dass durch das Vorhandensein eines Kommentars auch erwartet wird, dass der Code das im Kommentar beschriebene realisiert.

Hemmungstäuschung: Bei dieser Täuschung entsteht durch zuvor gelerntes Wissen eine Hemmung. So kann ein Pascal-Programmierer den folgenden Fehler bei einem Vergleich in einem C/C++-Programm leicht übersehen, da `=` in Pascal tatsächlich ein Vergleichsoperator ist und nicht ein Zuweisungsoperator, wie in C/C++: `if ( a = b ) then ...` statt `if( a == b ) then ...`

Überdeckungstäuschung: Bei der Überdeckungstäuschung überdeckt ein grösserer, stärkerer Eindruck einen kleineren, schwächeren Eindruck: „Testmethodon“ wird in einem Buch über Testmethoden auch als „Testmethoden“ gelesen.

Wiederholungstäuschung: Diese Art der Täuschung entsteht durch den Analogieschluss, dass ähnliche Vorgänge (oder Programmcode) auch ähnliche Ergebnisse haben. Dies führt dazu, dass ähnliche Programmteile nur flüchtig geprüft werden, was bei Copy-and-Paste-Programmierung zu Fehlern führen kann.

2.5 Kosten und Nutzen des Testens

Frage 2.19: In welchem Verhältnis stehen die Kosten zur Beseitung eines Fehlers zu Entwicklungsphase, in der er gefunden wird?

Die Kosten, die durch die Beseitigung eines Fehlers entstehen, hängen stark von der Entwicklungsphase ab, in der er gefunden wird. Stahl/Nomicos gaben beispielhaft folgende Dollarbeträge an:

1. Anforderungs- und Systemspezifikation: 349 $
2. Entwurf: 876 $
3. Codierung und Modultest: 1750 $
4. Test und Integration: 12782 $

Je später ein Fehler gefunden wird, desto teurer wird also seine Behebung. Wie schon erwähnt wurde, kommen auf tausend Zeilen Code ungefähr 30 bis 100 Fehler, von denen jedoch bis zur Auslieferung ca. 95% behoben werden.

3 Qualitätsmanagement-, Prüf- und Testmethoden im Überblick

3.1 Ziel, Intention und Vorgehensweise beim Testen

Frage 3.1: Wie ist das Ziel oder der Zweck des Testens definiert? Welche Arten des Vorgehens können dabei unterschieden werden?
Das Ziel bzw. der Zweck des Testens ist wie folgt definiert:

Definition: Allgemeines Ziel des Programmtestens ist es, die Qualität von Softwaresystemen durch systematisches Ausführen der Software unter sorgfältig kontrollierten Umständen zu erhöhen.

Der Begriff der Qualität wird später definiert. Es kann beim Testen zwischen zwei Arten des Vorgehens unterschieden werden:

demonstrativ: Bei dieser Art des Vorgehens werden Testdaten mit dem Ziel ausgewählt, die Fehlerfreiheit eines Programms nachzuweisen.

destruktiv: Ziel dieser Vorgehensweise ist es, die Testdaten möglichst so zu wählen, dass Fehler gefunden werden. Auf diese Art und Weise kann die Qualität des Softwareprodukts durch Korrektur der Fehler verbessert werden. Desweiteren ist es hierbei nicht möglich, sich durch eine vorteilhafte Auswahl der Testdaten in einer trügerischen Sicherheit zu wiegen.

Frage 3.2: Wie ist das Testen selbst definiert?

Definition: Testen ist der Prozess, ein Programm mit der Intention auszuführen, Fehler zu finden.

Bei dieser Definition wird also von einem destruktiven Test ausgegangen. Wie in der obigen Definition schon gesagt wurde, muss das Testen dabei unter „sorgfältig kontrollierten Umständen“ geschehen. Daher wird die Definition um die folgende erweitert:

Definition: Testen ist eine (im allgemeinen stichprobenartige) Ausführung von Experimenten mit dem Prüfgegenstand (ein Modul, Teilsystem oder Programmsystem) unter spezifizierten Bedingungen.

Die drei obigen Definitionen des Testens sollen dabei die in diesem Text weiterhin benutzten Definitionen sein.

3.2 Einordnung des Gebiets Testen in das Qualitätsmanagement

3.2.1 Aufgabe des Qualitätsmanagement

In Kapitel 3.1 wurde schon der Begriff der Qualität erwähnt. Dieser wird in diesem Kapitel näher geklärt und es wird aufgezeigt, dass sich die Produktqualität eines Softwareprodukts nicht nur durch Testen und Fehlerreduktion erhöhen lässt. Dies liegt daran, dass sich die Produktqualität aus verschiedenen

Aspekten zusammensetzt, die nicht alle durch Testen erreicht oder verbessert werden können.

Frage 3.3: Worin liegt die Aufgabe des Qualitätsmanagement?
Die Aufgabe des Qualitätsmanagement ist es, definierte Qualitätsmerkmale für ein Softwareprodukt zu erreichen.

Frage 3.4: Wie ist der Begriff der „Qualität“ definiert?
Der Begriff der Qualität ist wie folgt definiert:

Definition: Qualität ist die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung festgelegter oder vorausgesetzter Erfordernisse beziehen.

Frage 3.5: Wie ist der in der Definition von „Qualität“ benutzte Begriff des „Merkmal“ definiert?
Der Begriff „Merkmal“ ist wie folgt definiert:

Definition: Ein Merkmal ist eine Eigenschaft, die das Unterscheiden von Einheiten einer Gesamtheit entweder in qualitativer oder quantitativer Hinsicht ermöglicht.

Frage 3.6: Wie ist die Produktqualität für Software definiert?

Definition: Die Produktqualität für Software setzt sich aus mehreren Qualitätsmerkmalen zusammen:

Funktionalität: Eine Menge von Merkmalen, die sich auf das Vorhandensein einer Menge von Funktionen und auf deren festgelegte Merkmale beziehen. Die Funktionen sind jene, die die festgelegten und vorausgesetzten Erfordernisse erfüllen.

Zuverlässigkeit: Eine Menge von Merkmalen, die sich auf die Fähigkeit der Software beziehen, ihr Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum zu bewahren.

Benutzbarkeit: Eine Menge von Merkmalen, die sich beziehen

- auf den Aufwand, der zur Benutzung erforderlich ist.
- auf die individuelle Bewertung einer solchen Benutzung durch eine festgelegte oder vorausgesetzte Gruppe von Benutzern.

Effizienz: Eine Menge von Merkmalen, die sich auf das Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen beziehen.

Änderbarkeit: Eine Menge von Merkmalen, die sich auf den Aufwand beziehen, der zur Durchführung vorgegebener Änderungen (Korrekturen, Verbesserungen oder Anpassungen) notwendig ist.

Übertragbarkeit: Eine Menge von Merkmalen, die sich auf die Eignung der Software beziehen, von einer Umgebung in eine andere übertragen zu werden (z.B. Portierung).

Die Testbarkeit dieser Merkmale nimmt dabei in dieser Auflistung nach unten hin ab. Wirklich gut testbar sind im Prinzip nur die Merkmale Funktionalität und Zuverlässigkeit, weshalb auf diese nun näher eingegangen wird.

Frage 3.7: Welche Merkmale vereinigt der Begriff der „Funktionalität“?

Definition: Die Funktionalität ist abhängig von der Ausprägung folgender Qualitätsmerkmale eines Softwareprodukts:

Angemessenheit: Ein Merkmal von Software, das sich auf das Vorhandensein und die Eignung einer Menge von Funktionen für spezifizierte Aufgaben bezieht. (*„Wurden benötigte Funktionen implementiert und eventuelle unnötige weggelassen?“* Beispiel: Ein Webserver servt Webseiten und kocht keinen Kaffee.)

Richtigkeit: Ein Merkmal von Software, das sich auf das Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen bezieht. (*„Machen die Funktionen das, was sie sollen?“*)

Interoperabilität: Ein Merkmal von Software, das sich auf ihre Eignung bezieht, mit vorgegebenen Systemen zusammenzuwirken. (*„Wie gut kann eventuelle andere, schon vorhandene Software benutzt werden?“* Beispiel aus der PG: 3D-Modelle können mit Blender erstellt und importiert werden.)

Ordnungsmäßigkeit: Ein Merkmal von Software, das bewirkt, dass die Software anwendungsspezifische Normen oder Vereinbarungen oder gesetzliche Bestimmungen und ähnliche Vorschriften erfüllt. (Wahrscheinlich für jegliche Art von Software zutreffend, bei der ein Fehler katastrophale Konsequenzen haben könnte.)

Sicherheit: Ein Merkmal von Software, das sich auf die Eignung bezieht, unberechtigten Zugriff (sowohl versehentlich als auch vorsätzlich) auf Programme und Daten zu verhindern. (*„Kann die Software missbraucht werden, um beliebige Daten vom Benutzer zu lesen?“*)

Frage 3.8: Welche Merkmale vereinigt der Begriff der „Zuverlässigkeit“?

Definition: Die Zuverlässigkeit ist abhängig von der Ausprägung folgender Qualitätsmerkmale eines Softwareprodukts:

Reife: Ein Merkmal von Software, das sich auf die Häufigkeit von Versagen durch Fehlzustände in der Software bezieht. (Beispiel: Version 2.0 ist „stabiler“ als 1.0.)

Fehlertoleranz: Ein Merkmal von Software, das sich auf ihre Eignung bezieht, ein spezifiziertes Leistungsniveau bei Nicht-Einhaltung ihrer spezifizierten Schnittstelle oder bei Software-Fehlern zu bewahren. (Beispiel: Ein Programm lädt eine Datei, die nicht im richtigen bzw. in einem total falschen Format ist.)

Wiederherstellbarkeit: Ein Merkmal von Software, das sich auf die Möglichkeit bezieht, bei einem Versagen mit angemessenem (Zeit-)Aufwand ihr Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen. (Beispiel: Crawler einer Suchmaschine, der abstürzt, soll nicht alle bis dahin schon eingecrawlten Seiten wieder eincrawlen.)

Jedes dieser Merkmale kann getestet werden, allerdings müssen die Merkmale, wie z.B. das Leistungsniveau in der Definition zu „Fehlertoleranz“ genau spezifiziert sein.

3.2.2 Maßnahmen des Qualitätsmanagements

Mit Maßnahmen des Qualitätsmanagement werden die gewünschten (oben aufgeführten) Qualitätsmerkmale herbeigeführt und überprüft. Dabei können zwei Maßnahmen unterschieden werden.

Frage 3.9: Welche (zwei) Maßnahmen des Qualitätsmanagement können unterschieden werden?
Es kann zwischen konstruktiven und analytischen Qualitätsmanagementmaßnahmen unterschieden werden.

Definition: Konstruktive Qualitätsmanagementmaßnahmen sind Methoden, Sprachen und Werkzeuge, die dafür sorgen, dass das entstehende Produkt von vornherein bestimmte Eigenschaften besitzt.

Ein gutes Beispiel hierfür ist die **Verwendung von Bibliotheken** in Projekten. Diese können z.B. abstrakte Datentypen zur Verfügung stellen oder Implementierungen von Algorithmen anbieten (z.B. Leda). Wenn die Bibliothek fehlerfrei implementiert ist, braucht man sich um diese Komponenten zumindest nicht mehr allzu viele Sorgen machen.

Die konstruktiven Qualitätsmanagementmaßnahmen können Fehler jedoch nicht vollständig verhindern. Deshalb müssen einige Qualitätsmerkmale im nachhinein überprüft werden:

Definition: Analytische Qualitätsmanagementmaßnahmen sind diagnostische Maßnahmen, die der nachträglichen Überprüfung gewünschter Qualitätsmerkmale dienen.

Diese analytischen Maßnahmen lassen sich nach den folgenden Gesichtpunkten unterscheiden:

Ablauf: Der Ablauf der Prüfung kann entweder **statisch** oder **dynamisch** geschehen. Geschieht sie statisch, so erfolgt die Prüfung in der Reihenfolge, wie der Prüfgegenstand (z.B. der Programmcode) aufgeschrieben wurde. Bei einer dynamischen Prüfung geschieht die Prüfung in der Reihenfolge der Ausführung des Prüfgegenstands.

Vorgehensweise: Eine **informelle** Prüfung wäre z.B. die Begutachtung durch einen Menschen. Können mathematische oder logische Verfahren angewandt werden, so spricht man von einer **formellen** Prüfung. Bei einem **experimentellen** Vorgehen wird das Verhalten des Prüfobjekts (also des Programms) beobachtet.

vollständig vs. stichprobenartig: Die Prüfung kann entweder **vollständig** sein (z.B. erschöpfender Test) oder **stichprobenartig** (einige ausgewählte Testdaten).

Beschreibung des Prüfgegenstand: Der Prüfgegenstand kann kann **formal**, **teilweise formal** oder **informell** *(„Lass mal 'ne Suchmaschine schreiben.“)* beschrieben sein.

Automatisierung der Prüfung: Von der Beschreibung des Prüfgegenstands und der Vorgehensweise hängt es ab, ob die Prüfung **vollständig, teilweise oder nicht automatisiert** stattfinden kann.

3.3 Statische und dynamische Prüfverfahren

Das folgende Unterkapitel beschäftigt sich nun mit den beiden, schon etwas weiter oben angesprochenen, Aspekten „Ablauf“ und „Vorgehensweise“. Statische und dynamische Verfahren können beide jeweils informell und formal durchgeführt werden.

3.3.1 Statische Prüfverfahren

Frage 3.10: Welche informellen und formalen Verfahren kennst du bei den statische Prüfverfahren?

Bei den statischen Verfahren gibt es die folgenden **informellen** Verfahren:

- Schreibtischtest durch eine Person
- Audits, Reviews und Inspektion durch Personengruppen

Diese Begutachtungen durch Menschen werden **meist vollständig** durchgeführt, was zwar einen **hohen Aufwand** darstellt, wobei jedoch **viele Fehler mit der entsprechende Stelle**, an der der Fehler auftritt, aufgedeckt werden. Zu den **formalen** statischen Verfahren zählen:

Syntaxanalyse: Hierbei wird die **syntaktische Konsistenz** des Programmcodes überprüft. Dazu zählt zum Beispiel, ob **Schnittstellen** richtig benutzt

werden. Diese Überprüfung **kann vom Compiler übernommen werden** (z.B. „Gibt es einen Konstruktor mit den Argumenten, die bei der Erzeugung einer Instanz übergeben werden?“).

Datenflussanalyse: Dies beinhaltet die Überprüfung des Datenflusses eines Programms oder einer Spezifikation. Es kann z.B. geschaut werden, ob eine **Variable mit undefiniertem Wert referenziert** wird. Dies kann teilweise auch vom Compiler übernommen werden.

Kontrollflussanalyse: Diese Analyse kontrolliert den Kontrollfluss eines Programms oder einer Spezifikation. Es wird z.B. geschaut, ob es **unendliche Schleifen** oder **nicht erreichbare Stellen im Programmcode** gibt.

Formale Verifikation: Bei diesem Vorgehen wird mit einem **mathematisch-logischen Verfahren** die Korrektheit des Programms bezüglich der Spezifikation gezeigt. Dazu werden die folgenden zwei Schritte gemacht:

1. Es wird gezeigt, dass das Programm korrekte Ergebnisse liefert, wenn es anhält.
2. Es wird gezeigt, dass das Programm nicht in eine endlose Schleife gerät.

Dieses Verfahren kann informell **per Hand** oder formal mit einem **automatischen Theorembeweiser** durchgeführt werden.

Die formalen statischen Verfahren dienen auch dazu, **Informationen aus der Programmstruktur abzuleiten**. Dazu zählen zum Beispiel der **Kontrollflussgraph** und der **Datenflussgraph**.

3.3.2 Dynamische Prüfverfahren

Auch hier lassen sich wieder informelle und formale Verfahren unterscheiden.

Frage 3.11: Welche informellen und formalen Verfahren kennst du bei den dynamischen Prüfverfahren?

Ein informelles Verfahren ist der **Walkthrough**, also ein manueller Durchlauf durch ein Dokument (z.B. Spezifikation oder Code) mit Testfällen. Im Gegensatz zu einem vollständigen informellen statischen Verfahren werden hierbei allerdings nicht alle Fälle abgedeckt.

Zu den formellen Verfahren zählen die folgenden beiden Verfahren:

Symbolische Ausführung: Ein Programmteil wird mit **symbolischen Werten** durchgerechnet. Auf diese Methode wird später in Kapitel 9.3 (Seite 88) noch näher eingegangen.

Testen (im engeren Sinn): Darunter ist die Ausführung des Programms mit **konkreten Daten** zu verstehen. Das Testen ist **aufgrund der Unzulänglichkeiten und Probleme der anderen Verfahren unverzichtbar**. Da ein idealer Test nur durch Zufall gefunden werden kann und ein erschöpfender Test meist nicht möglich ist, müssen bei diesem Verfahren also möglichst **ideale Stichproben** gefunden werden.

3.4 Grundsätzliche Teststrategien und Testansätze

Frage 3.12: Was sind Testbasen?

Bei einem systematische Test auf Fehler müssen stets jeweils **zwei Dokumente** gegeneinander getestet werden. Dies kann zum Beispiel ein Programm, also eine Implementation, sein, welches gegen die Spezifikation getestet wird. Die dafür relevanten Dokumente werden **Testbasen** genannt.

Frage 3.13: Welche grundsätzlichen Strategien sind beim Testen möglich? Welches gemeinsame Problem haben beide Strategien?

Im Prinzip kann zwischen zwei grundsätztlichen Strategien unterschieden werden: dem **spezifikationsorientierten** und dem **implementationsorientierten** Testen. Beide Verfahren haben ihre Vor- und Nachteile, auf die jedoch später eingegangen wird.

Es gibt jedoch auch ein gemeinsames Problem: Fehler in der Spezifikation können durch Testen nicht gefunden werden. Ist nämlich die Spezifikation selbst schon fehlerhaft, d.h. entspricht diese nicht den Anforderungen, die eigentlich an das Programm gestellt werden, so kann dies nicht durch Testen herausgefunden werden, wenn das Programm der Spezifikation nach korrekt implementiert wurde. Es müsste also erst gezeigt werden, dass die Spezifikation die Anforderungen korrekt widergibt. Dies kann jedoch in den meisten Fällen nicht mit formalen Verfahren erfolgen, da die Anforderungen selbst oft nur informell gegeben sind.

Frage 3.14: Wie wird allgemein beim spezifikationsorientierten Testen vorgegangen?

Beim spezifikationsorientierten Testen werden die Testdaten anhand der **Spezifikation** erzeugt. Es wird also nicht die interne Struktur eines Programms (wie sie z.B. bei einer Implementation vorliegt) berücksichtigt. Je aussagekräftiger und detaillierter die Spezifikation ist, desto höher ist im allgemeinen die Qualität der Testdaten. Sind zum Beispiel nur die Schnittstellen der Funktionen spezifiziert, so können nur Testdaten erstellt werden, die den Ein- und Ausgabebereich der Funktionen testen (z.B. was macht eine Funktion, die eine positive Zahl erwartet bei Eingabe einer negativen Zahl). Ist die Semantik einer Funktion (oder eines Moduls) jedoch ebenfalls relativ formal spezifiziert, so können weitere Testdaten erzeugt werden, die die entsprechende Semantik jeweils berücksichtigen und testen.

Frage 3.15: Welche Vorteile gibt es beim spezifikationsorientierten Testen? Welche Nachteile?

Beim spezifikationsorientierten Testen gibt es die folgenden **Vorteile**:

- Nicht implementierte, aber spezifizierte Teile eines Programms können entdeckt werden.
- Portierungen werden unterstützt, da die Testdaten implementationsunabhängig, nur aufgrund der Spezifikation erstellt wurden.

Die **Nachteile** sind:

- Zusätzlich implementierte Fallunterscheidungen, die in der Spezifikation nicht vorgesehen sind, können nur durch Zufall entdeckt werden. So würde z.B. die Zeile `if( persnum == 243) then ueberweiseGeld()` beim implementationsorientierten Testen durch eine Zweigüberdeckung eventuell entdeckt werden, weil diese Entscheidung auch einmal als wahr evaluliert werden muss.

- Ist die Spezifikation wenig aussagekräftig, so sind auch die Testdaten entsprechend unrepräsentativ.

Frage 3.16: Inwiefern kann das spezifikationsorientierte Testen auf nebenläufige Softwaresysteme angewandt werden?
Die bestehenden spezifikationsorientierten Testmethoden für sequentielle Softwaresysteme können für die sequentiellen Teile eines nebenläufigen Systems genutzt werden. Zum Auffinden von speziellen Fehlern nebenläufiger Systeme, wie z.B. Synchronisationsfehlern, müssen jedoch andere Verfahren benutzt werden.

Frage 3.17: Wie wird allgemein beim implementationsorientierten Testen vorgegangen?
Beim implementationsorientierten Testen ergeben sich die Testdaten aus der **internen Struktur des Programms**. Dabei werden im allgemeinen der **Kontroll- und Datenfluss** in einem Programm betrachtet. Es wäre z.B. wünschenswert, wenn für jeden möglichen Weg in einem Programm ein Testdatum vorhanden wäre. Da dies jedoch zu aufwändig ist, muss eine relevante Teilmenge von Programmpfaden gefunden werden. Dies geschieht durch Angabe von bestimmten Kriterien, die von den Testdaten erfüllt werden müssen. Wieder andere Techniken betrachten die **Struktur der einzelnen Anweisungen** eines Programms.

Frage 3.18: Welche Vorteile gibt es beim implementationsorientierten Testen? Welche Nachteile?
Die **Vorteile** des implementationsorientieren Testens sind:

- Eine automatische Unterstützung des Tests ist möglich, da das Programm eine Testbasis mit einer wohldefinierten Syntax und Semantik ist.

- Es können auch Programme „getestet“ werden, für die die Spezifikation unvollständig ist oder auch nur in den Köpfen der Entwickler existiert. Dies ist jedoch kein sehr objektiver Test, da die subjektive Vorstellung eines Entwicklers bestimmt, ob ein Programm das macht, was es machen soll.

- Zusätzliche Fallunterscheidungen und Verfeinerungen, die in der Spezifikation nicht vorgesehen sind, können getestet werden.

Und die **Nachteile**:

- Nicht implementierte Teile einer Spezifikation können durch den implementationsorientierten Test nur zufällig entdeckt werden.

Frage 3.19: Inwiefern kann das implementationsorientierte Testen auf nebenläufige Softwaresysteme angewandt werden?
Im Prinzip können die sequentiellen Verfahren auch hier wieder nur benutzt werden, um die sequentiellen Teile eines nebenläufigen Systems zu testen.

Frage 3.20: Was ist der Unterschied zwischen dem funktionalen und dem strukturellen Testansatz?
Der funktionale Testansatz wird auch **Black-Box-Test** genannt und der strukturelle Ansatz **White-Box-Test** bzw. **Glass-Box-Test**. Und im Prinzip überdeckt sich der funktionale Testansatz mit den Teststrategien des spezifikationsorientierten Testens und der strukturellen Testansatz mit den Strategien des implementationsorientierten Testens. Dem entsprechend passen auch die Namen. Beim Black-Box-Testen wird das zu testende System als eine Black-Box betrachtet, von der man nicht weiß, wie sie von Innen aussieht. In diesem Fall wird ja auch nur die Spezifikation benutzt, d.h. die innere Struktur des Programms interessiert nicht (bzw. ist nicht bekannt). Bei den strukturellen Ansätzen hingegen wird gerade die innere Struktur des Programms benutzt, um die Testdaten zu erstellen. Das Programm ist quasi gläsern, halt eine Glass-Box.

3.5 Testablauf

Frage 3.21: Was ist ein Testzyklus?
Beim Testen eines Software-Produkts kann grob in drei Phasen aufgeteilt werden. Dies sind:

- Testvorbereitung
- Testausführung
- Testauswertung

Diese drei Aktivitäten werden zeitlich hintereinander ausgeführt und bilden einen **Testzyklus**. In jeder Testphase (Modultest, Integrationstest, Systemtest – siehe Kapitel 3.6) wird der Testzyklus durchgeführt. Er wird jeweils so lange wiederholt, bis keine Fehler mehr auftreten und die gewünschte Testgüte erreicht ist.

Frage 3.22: Welche drei Aktivitäten können bei der Testvorbereitung unterschieden werden?
Bei der Testvorbereitung können folgende Aktivitäten unterschieden werden:

- Testplanung
- Testdatenerzeugung
- Testrahmenerzeugung

Frage 3.23: Was ist der Zweck der Testplanung?

Frage 3.24: Was passiert in der Phase der Testdatenerzeugung?
In der Phase der Testdatenerzeugung werden mit dem in der Testplanung vorgegebenen Verfahren die Testdaten erzeugt, mit denen das Programm getestet wird. Ein solches Verfahren kann z.B. das implementationsorientierte Verfahren des kontroll- oder datenflussbezogenen Testens sein.

Frage 3.25: Was ist ein Testfall?
Ein **Testfall** beschreibt ein Testdatum durch seine relevanten Eigenschaften. Dies bedeutet, dass sich ein Testfall meistens aus mehreren Testdaten zusammensetzt. Sollen z.B. Testdaten erzeugt werden, die den *true*-Zweig der Entscheidung `if( x > 0 )` ausführen, so sind die erlaubten Eingabedaten der zu diesem Testfall gehörigen Testdaten die Menge $\{1, 2, 3, \dots\}$ (wenn x nicht vorher durch Operationen transformiert wird).

Frage 3.26: Was ist ein Testdatum?
Ein **Testdatum** ist aus einem Eingabe- und aus einem Solldatum zusammengesetzt.

Frage 3.27: Was ist ein Eingabedatum? Was ist in diesem Zusammenhang ein Testwert? Und wann wird von einer Eingabe gesprochen?
Ein **Eingabedatum** ist ein Tupel von Werten für alle Eingabevariablen eines Programmes, wobei zusätzlich der Inhalt der Eingabedateien, der permanenten Dateien und Datenbanken spezifiziert ist. Die Komponenten des Eingabedatums heißen **Testwerte**. In manchen Fällen (z.B. bei Dialogprogrammen) wird bei einem Programmablauf eine Folge von Eingabedaten verarbeitet, die einfach **Eingabe** heißt.

Frage 3.28: Was ist ein Solldatum? Was ist ein Istdatum?
Ein **Solldatum** ist ein gemäß Spezifikation gewünschtes Ergebnis, welches erzeugt werden sollte, wenn das Programm mit einem Eingabedatum ausgeführt wird.

Ein **Istdatum** ist das Ergebnis, welches erzeugt wird, wenn das (implementierte) Programm mit einem Eingabedatum ausgeführt wird.

Frage 3.29: Wie ist das Ergebnis einer Programmausführung definiert und wann ist ein Programm korrekt?
Das **Ergebnis einer Programmausführung** ist ein Tupel von Werten für alle Ausgabevariablen, wobei außerdem der Inhalt der Ausgabedateien, der permanenten Dateien und Datenbanken mit spezifiziert ist.

Ein Programm ist korrekt, wenn Soll- und Ist-Datum bei allen Ausführungen übereinstimmen.

Frage 3.30: Was ist eine Testdatenmenge?
Die Testdatenmenge setzt sich aus den Repräsentanten einer jeden Klasse von Testdaten zusammen.

Frage 3.31: Gebe ein Beispiel für ein Testdatum mit Sollwert!
Als Beispiel für ein Testdatum wird im Buch ein Kontenverwaltungsprogramm einer Bank angegeben. Die Eingabe für dieses ist ein Betrag, der Vorgang (Abhebung/Einzahlung), die Kontonummer sowie eine Datenbank, die für jedes Konto den aktuellen Kontostand speichert. Ein Beispieltestdatum könnte also (90.10, „*E*“, 532496, 350.00) sein. Das Solldatum ist in diesem Fall ein Kontostand von 440.10 für die Kontonummer 532496 in der Datenbank sein.

Frage 3.32: Was wird im Rahmen der Testrahmenerstellung gemacht?
Bei der Testrahmenerstellung werden die für die Modultests benötigten Treiber und Platzhalter erzeugt. Treiber ersetzen dabei die Module, die das zu testende Modul benutzen bzw. ansteuern und Platzhalter die Module, die vom zu testenden Modul verwendet werden.

Frage 3.33: Wie können die Treiber und Platzhalter realisiert werden?
Die Treiber und Platzhalter können entweder als **interaktive Schnittstelle** zwischen Testperson und zu testendem Objekt realisiert werden oder die benötigten Daten für diese Komponenten werden **aus einer Datei** gelesen.

Frage 3.34: Welche Maßnahmen umfasst die Testauswertung?
Die Testauswertung umfasst im Prinzip zwei Schritte:

Messung der Testgüte: Der Grad der Erfüllung des Testkriteriums (wie z.B. Zweigüberdeckung) wird ermittelt.

Soll-Ist-Vergleich: In diesem Schritt wird das Testergebnis mit dem spezifizierten Sollwert verglichen. Im Prinzip kann dieser Schritt immer automatisch durchgeführt werden, sofern die Solldaten vorliegen. Ein sogenanntes **Testorakel** muss dabei die Sollwerte aus der Spezifikation ableiten. Dabei gibt es mehrere Möglichkeiten:

1. Ein **Mensch fungiert als Testorakel**, indem das Solldatum aus dem Eingabedatum anhand der Spezifikation abgeleitet wird.
2. Beim **diversitären Testen** wird das Programm von unabhängigen Entwicklergruppen mehrfach entwickelt und diese verschiedenen Versionen werden gegeneinander getestet. Dabei werden alle Fehler entdeckt, die nur in einer Teilmenge der verschiedenen Versionen exisitieren. Nicht entdeckt werden Fehler, die in allen Versionen vorkommen. Die Wahrscheinlichkeit, dass ein Fehler in allen Versionen auftritt sinkt mit der Anzahl der Versionen.
3. Wenn formale Spezifikationen vorliegen, kann daraus fast automatisch ein Prototyp erstellt werden. (Warum wird dann nicht gleich der Code genommen?)

3.6 Testphasen im Software-Entwicklungsprozess

Frage 3.35: Welche Testphasen können für ein implementiertes Programm unterschieden werden?

Die Testphasen für ein implementiertes Programm kann in drei Phasen unterteilt werden, die zeitlich hintereinander durchgeführt werden. Dem **Modultest**, dem **Integrationstest** und dem **Systemtest**. Dabei wird jedes Testobjekt gegen seine Spezifikation getestet.

Frage 3.36: Was wird im Modultest gemacht?

Ein Programmteil ist ein Modul, wenn er eine **logisch separierbare Einheit mit klar definierten Schnittstellen** bildet. Beim Modultest wird jedes Modul aus der Menge der Module **unabhängig von den anderen Modulen getestet**. Dabei ist die Erstellung von Treibern und Platzhaltern nötig.

Frage 3.37: Was wird beim Integrationstest gemacht?

Ziel des Integrationstests ist es, die **Schnittstellen zwischen den Modulen** zu testen sowie das **Zusammenspiel der Module** untereinander. Dabei orientiert man sich an der Benutzungshierarchie der Module bzw. bei objektorientierten Systemen auch an der Vererbungshierarchie. Das Vorgehen kann nichtinkrementell oder inkrementell sein. Zwei Strategien des inkrementellen Testens sind die **aufsteigende** und die **absteigende Strategie**.

Frage 3.38: Was geschieht beim Systemtest? Was ist ein Abnahmetest? Was ein Regressionstest?

Beim **Systemtest** wird das ganze System, nachdem alle Module integriert wurden, noch einmal gegen die Spezifikation getestet. Dabei werden die funktionalen als auch nichtfunktionalen Aspekte des Systems überprüft.

Erfolgt der Systemtest durch die Auftraggeber, so spricht man von einem **Abnahmetest**. Der Auftraggeber testet in diesem Fall also, ob das System seinen Vorstellungen und Anforderungen entspricht.

Ein **Regressionstest** tritt während bzw. nach einer Wartungsphase auf, wenn Änderungen am System vorgenommen werden. Es ist dann zu testen, ob sich durch die Änderung nicht ausversehen neue Fehler eingeschlichen haben.

Frage 3.39: Was ist ein Testprozess? Was ist ein Testprozessmodell? Welche Testprozessmodelle kennst du?

Ein **Testprozess** ist die Durchführung aller Testphasen und Testaktivitäten zur Realisierung eines Gesamttests. Ein **Testprozessmodell** beschreibt dabei die wichtigsten Eigenschaften des Testprozesses, also zum Beispiel die Art und Weise des Vorgehens. Es können zwei Testprozessmodelle unterschieden werden:

Phasenmodell: Beim Phasenmodell wird der Prozess des Testens als eine **eigenständige Phase** des Software-Entwicklungsprozess angesehen. Das Testen, inklusive Testvorbereitung, wird erst nach der Entwicklung des Systems durchgeführt. Dieses Modell wird in der Praxis noch häufig eingesetzt.

Lebenszyklusmodell: In diesem Modell werden die **Testaktivitäten parallel zur Entwicklung** durchgeführt. Dadurch können Informationen, die während der Entwicklung gewonnen werden auch zum Testen verwendet werden. Das Lebenszyklusmodell ist dem Phasenmodell vorzuziehen, jedoch ist es leider auch aufwändiger.

3.7 Überprüfung des Entwicklungs- und Qualitätsmanagementsystems

3.8 Testmanagement

Frage 3.40: Welche Aufgaben gehören unter anderem zum Testmanagement?

Zu den Aufgaben des Testmanagement gehört unter anderem die **Auswahl und Kombination der Testverfahren**, sowie die **Entscheidung über die Beendigung des Testens**.

Die Auswahl der Testverfahren kann dabei meist über Komplexitätsmaße geschehen, die bestimmen welche Konstrukte besonders häufig bzw. nicht so häufig vorkommen, so dass bestimmte Testverfahren zu bevorzugen sind. Als Kombination bieten sich allgemein die folgende Reihenfolge von Testverfahren an:

1. Statische Prüfung und eventuell symbolische Ausführung
2. Spezifikationsorientierter Test
3. Implementatiosorientierter Test
4. Formale Verifikation (bei kritischen Systemen)

Die Frage nach dem richtigen Zeitpunkt für das Ende des Testens wird leider oft von externen Faktoren, wie ungedultigen Auftraggebern oder festgelegten Terminen, beeinflusst. Sind diese Faktoren nicht gegeben, kann die **erzielte Testgüte als Abbruchskriterium** herangezogen werden. Sie kann aus benutztungsorientierter, fehlerorientierter oder struktureller Sicht beurteilt werden:

Benutzungsorientiert: Hier wird die **Fehleraufdeckungsrate** herangezogen. Geht diese gegen Null, kann mit dem Testen aufgehört werden. Leider ist dieses Maß sehr subjektiv, da es z.B. die **Testanstrengung** nicht berücksichtigt.

Fehlerorientiert: Dies geschieht über die **Mutationsanalyse** oder der **Fehlereinpflanzung**.

Strukturorientiert: Hier wird der Überdeckungsgrad der strukturellen Testkriterien herangezogen.

4 Kontrollflussbezogenes Testen

Dieses Kapitel entspricht dem siebten Kapitel des Buches ([Riedemann 1997]).

4.1 Kontrollflussgraphen und ihre Eigenschaften

Frage 4.1: Wie ist ein Kontrollflussgraph definiert? Was ist ein Kontrollflussschema?

Definition: Der **Kontrollflussgraph** eines Programms ist ein gerichteter Graph. Seinen Knoten sind (als Knotenmarkierung) Anweisungen oder Entscheidungsprädikate des Programms zugeordnet und seine Kanten stellen die Verbindungen zwischen Anweisungen und Anweisungsnachfolgern dar.

Dabei können ganze Anweisungsfolgen, die keine bedingten Anweisungen enthalten, einem Knoten zugeordnet sein. Das komplette Entscheidungsprädikat einer bedingten Anweisung ist dagegen getrennt vom Rest der bedingten Anweisung einem sogenannten **Entscheidungsknoten** zugeordnet, wobei für jeden Ausgang der Entscheidung eine Kante zu einem Nachfolgerknoten exisitiert, die mit ja (true) bzw. nein (false) markiert ist. Diese Kanten heißen **Entscheidungskanten**. Wenn eine Kante von Knoten k zu Knoten l führt, heißt l **Nachfolger(knoten)** von k und k **Vorgänger(knoten)** von l. Um Initialisierungsvorgänge (z.B. Parameterübergabe bei Prozeduren) modellieren zu können, enthält ein Kontrollflussgraph nur einen Knoten (genannt: **Anfangsknoten**), der keinen Vorgängerknoten hat und auch nur eine ausgehende Kante, genannt **Anfangskante**, besitzt. Außerdem soll ein Kontrollflussgraph nur einen Knoten, genannt **Endknoten**, enthalten, der keinen Nachfolger hat.

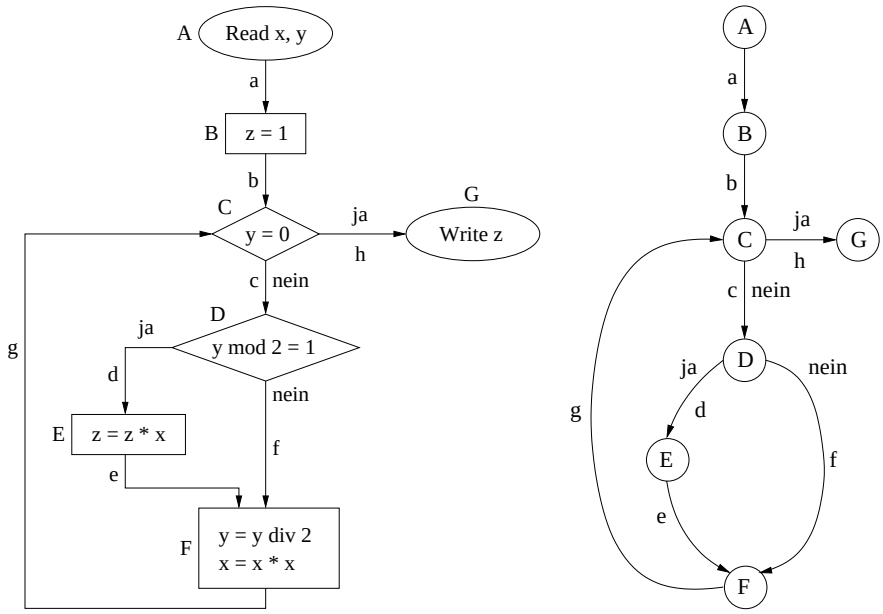
Das **Kontrollflussschema** zu einem Kontrollflussgraphen entsteht durch Ersetzung der Anweisungen und Entscheidungsprädikate durch formale Bezeichner. Falls es auf die Unterscheidung nicht ankommt, wird in beiden Fällen von **Kontrollflussgraphen** gesprochen.

Frage 4.2: Was ist ein Entscheidungs-Entscheidungs-Weg?

Ein Entscheidungs-Entscheidungs-Weg ist wie folgt definiert:

Definition: Ein **Entscheidungs-Entscheidungs-Weg** ist ein Wegstück, welches bei einem Entscheidungsknoten oder dem Anfangsknoten beginnt und alle folgenden Knoten und Kanten bis zum nächsten Entscheidungsknoten bzw. bis zum Endknoten des Kontrollflussgraphen (einschließlich) enthält.

Dabei besitzt diese Folgen von Knoten und Kanten, die folgende Eigenschaft: Wird die erste Kante eines Entscheidungs-Entscheidungs-Weges w ausgeführt, so werden alle Knoten und Kanten dieses Weges w ausgeführt. Dies kann man sich relativ schnell daran klar machen, dass auf dem gesamten Weg ja keine weitere Entscheidung mehr vorkommen kann, da es sich sonst nicht um einen Entscheidungs-Entscheidungs-Weg im Sinne der obigen Definition handeln würde. Aus diesem Grund kann man sich einfach nur stur entlang dieses Weges bewegen, bis das Ende erreicht ist.



(a) Kontrollflussgraph (b) Kontrollflussschema

Abbildung 1: Kontrollflussgraph und Kontrollflussschema zum Programm zur Berechnung von $z = x^{|y|}$ für $y \in \mathbb{N}$

Frage 4.3: Was ist ein Vereinigungsknoten?
Ein **Vereinigungsknoten** ist ein Knoten, der **mehr als einen Vorgänger** besitzt. Zwei Entscheidungs-Entscheidungs-Wege können sich zwischen einem Vereinigungsknoten und dem nächsten Entscheidungsknoten (bzw. Endknoten) überlappen.

Frage 4.4: Was ist ein Segment?
Da sich Entscheidungs-Entscheidungs-Wege überlappen können, wird nun versucht, mit Hilfe der Definition eines Segmentes den Kontrollflussgraphen noch weiter in überlappungsfreie Stücke (oder Segmente) zu zerlegen. Dies leistet die folgende Definition:

Definition: Ein **Segment** ist ein Wegstück mit folgenden Eigenschaften:

1. Der erste Knoten des Wegstücks ist der Anfangsknoten des Kontrollflussgraphen oder ein Entscheidungsknoten oder ein Vereinigungsknoten.
2. Der letzte Knoten ist der Endknoten des Kontrollflussgraphen oder ein Entscheidungsknoten oder ein Vereinigungsknoten.
3. Alle anderen Knoten haben nur eine Eingangs- und eine Ausgangskante.

Dabei sorgt Punkt 3 der Definition dafür, dass auf dem Weg keine weiteren Entscheidungs- oder Vereinigungsknoten liegen.

Frage 4.5: Was ist ein Flussdiagramm?
Ein **Flussdiagramm** ist eine graphische Darstellung eines Kontrollflussgraphen. Dabei werden der Anfangs- und der Endknoten jeweils als Ellipsen gezeichnet, Anweisungsknoten als Rechtecke und Entscheidungsknoten als Rauten.

Frage 4.6: Was ist ein Zyklus? Was ein einfacher Zyklus?
Es sei ein Kontrollflussgraph gegeben.

- Ein **Zyklus** ist ein Weg im Kontrollflussgraphen mit mindestens zwei Knoten, der an demselben Knoten beginnt und endet.
- Ein **einfacher Zyklus** ist ein Zyklus, bei dem alle Knoten voneinander verschieden sind (mit Ausnahme des Anfangs- und des Endknoten des Weges).

Frage 4.7: Was ist ein vollständiger Weg?
Ein **vollständiger Weg** (durch das Programm) ist ein Weg vom Anfangsknoten zum Endknoten des Kontrollflussgraphen des Programms oder ein unendlich langer Weg, der beim Anfangsknoten beginnt.

Frage 4.8: Was ist Wege(T)?
Für eine Menge T von Testdaten ist **Wege(T)** die Menge der vollständigen, endlichen Wege w des Kontrollflussgraphen, für die es ein Testdatum t aus T gibt, das den Weg w ausführt. Wenn zusätzlich die Abhängigkeit von Programm P ausgedrückt werden soll, wird Wege(T) mit **Wege(T,P)** bezeichnet.

Wege$_\infty$(T) bzw. **Wege$_\infty$(T,P)** bezeichnen die entsprechenden Wegemengen, bei denen auch unendlich lange Wege erlaubt sind.

Frage 4.9: Wann sind strukturierte Kontrollflussgraphen? Was sind stark bzw. schwach strukturierte Programme?
Bei **strukturierten** Kontrollflussgraphen haben alle Schleifen genau einen Eingangsknoten (der zur Schleife gehört) und genau einen Ausgangsknoten (der nicht zur Schleife gehört); die Schleifen überlappen sich nicht teilweise, d.h. bei sich überlappenden Schleifen ist eine Schleife vollständig in der anderen enthalten.

Ein **stark strukturiertes** Programm ist ein Programm, welches nur die Sequenz (also Anweisungen), die Alternative (Entscheidungsknoten, z.B. für if-Konstrukt) und die kontrollierte Iteration (`while`- und `repeat`-Schleifen) als Kontrollstrukturen zulässt. **Schwach strukturierte** Programm lassen zusätzlich den direkten Sprung aus einer Schleife heraus zu. In C++ würde dies, z.B. innerhalb einer `for`-Schleife, mit dem Schlüsselwort `break` gehen. Eine `for`-Schleife ist übrigens der Sonderfall einer `while`-Schleife.

Abbildung 2 verdeutlicht diese Eigenschaften. Alle drei Schleifen sind strukturiert, da sie jeweils nur einen Eingangsknoten E und einen Ausgangsknoten A besitzen. Jedoch sind nur die `while`-Schleife links und die `repeat`-Schleife in der Mitte stark strukturiert, da die schwach strukturierte Schleife rechts nur

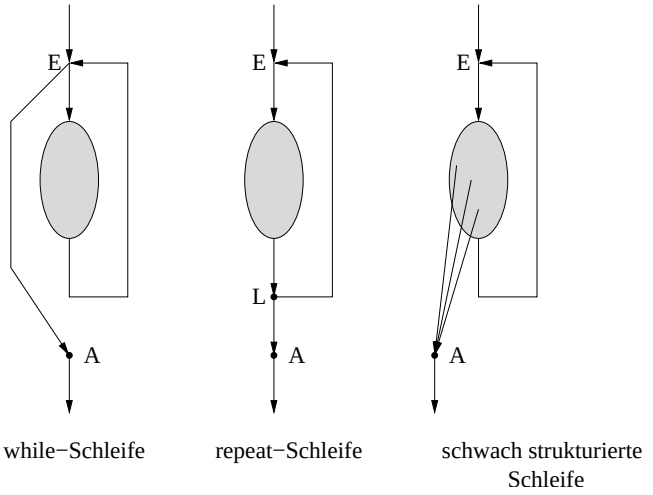


Abbildung 2: Stark und schwach strukturierte Programme

durch einen Sprung direkt aus der Schleife den Ausgangsknoten A erreichen kann.

Die Entscheidung, ob die Schleife nochmal durchlaufen wird, fällt bei der `while`- und der `repeat`-Schleife jeweils an einer anderen Stelle. Bei der `while`-Schleife wird die Entscheidung im Knoten A getroffen, weshalb es bei ihr auch **möglich ist, dass sie keinmal durchlaufen wird**. Bei der `repeat`-Schleife fällt die Entscheidung im Knoten L, welcher der letzte noch zur Schleife gehörende Knoten ist. Eine `repeat`-Schleife muss daher **mindestens einmal durchlaufen werden**.

Frage 4.10: Was ist ein wohlgeformter Kontrollflussgraph?
Ein **wohlgeformter** Kontrollflussgraph ist ein Kontrollgraph, bei dem es für jeden Knoten k einen Weg vom Anfangsknoten zum Endknoten gibt, auf dem k liegt. Somit gibt es in einem solchen Graphen keine unerreichbare Knoten und auch keine Schleifen, die keinen Ausgang besitzen (also quasi Sackgassen wären). Strukturierte Kontrollflussgraphen haben die Eigenschaft, wohlgeformt zu sein.

4.2 Methoden des kontrollflussbezogenen Testens

Frage 4.11: Welche Gemeinsamkeit haben die Methoden des kontrollflussbezogenen Testens?
Die verschiedenen Methoden des kontrollflussbezogenen Testens haben die Gemeinsamkeit, dass festgelegt wird, welche Wege bzw. Wegstücke wie oft ausgeführt werden sollen. Meistens soll ein Weg bzw. Wegstück nur mindestens einmal ausgeführt werden. Ein **Testwirksamkeitsmaß** gibt dann an, wieviele Wege bzw. Wegstücke die Forderung beim Ausführen einer Testdatenmenge erfüllen.

Frage 4.12: Welche Fehler können in Bezug auf die Programmstruktur unterschieden werden?
Es können drei Arten von Fehlern unterschieden werden:

Berechnungsfehler: Ein Berechnungsfehler kann z.B. durch einen Zuweisungsfehler erfolgen. Es wird der richtige Kontrollflussweg im Programm ausgeführt, jedoch ist mindestens ein berechneter Variablenwert falsch.

Bereichsfehler: Bei dieser Art von Fehler wird nicht der richtige Kontrollflussweg ausgeführt. Der Grund dafür ist, dass der **Eingabebereich** des vorliegenden Kontrollflussweges (d.h. die Menge der Datem, die diesen Weg ausführen) nicht mit dem Eingabebereich des Kontrollflussweges des korrekten Programms übereinstimmt. Dies kann z.B. durch eine falsche Bedingung in einem Entscheidungsknoten passieren.

Unterbereichsfehler: Die ist ein spezieller Fall eines Bereichsfehlers, bei dem ein Kontrollflussweg zu wenig oder zu viel im Programm vorkommt. Ein Beispiel hierfür ist eine Extraabfrage, wie z.B.:

```
if(personalnummer==234) then ueberweiseGeldInDieSchweiz()
```

Frage 4.13: Welche kontrollflussbezogenen Überdeckungen kennst du?
Es können grob vier Arten der kontrollflussüberzogenen Überdeckungen unterschieden werden:

1. C_0- oder Anweisungsüberdeckung
2. C_1- oder Zweigüberdeckung (bzw. Segment- oder Entscheidungsüberdeckung)
3. Pfadüberdeckungen (C_{SP}- und C_S(n)-Überdeckung)
4. Schleifenüberdeckung

4.2.1 Anweisungsüberdeckung

Frage 4.14: Worum geht es bei der Anweisungsüberdeckung? Wann erfüllt eine Menge von Testdaten die Anweisungsüberdeckung?
In jeder Anweisung eines Programms kann es zu einem Fehler kommen. Eine relativ schwache Forderung an die Testdatenmenge kann daher sein, dass die Testdaten so gewählt werden sollen, dass jede Anweisung mindestens einmal ausgeführt wird. Diese Forderung wird Anweisungsüberdeckung genannt und ist wie folgt definiert:

Eine Testdatenmenge T erfüllt die **C_0-Überdeckung** (für ein Programm P) genau dann, wenn es für jede Anweisung A des Programms P ein Testdatum t aus T gibt, das die Anweisung A ausführt.

Alternativ kann die Definition über die Benutzung von Wege(T) erfolgen: Eine Testdatenmenge T erfüllt die C_0-Überdeckung (für ein Programm P) genau dann, wenn es für jeden Knoten k des Kontrollflussgraphen von P einen Weg aus Wege(T,P) gibt, zu dem k gehört.

Frage 4.15: Wie ist das Testwirksamkeitsmaß (TWM) für die Anweisungsüberdeckung definiert?

Das Testwirksamkeitsmaß für die Anweisungsüberdeckung ist wie folgt definiert:

$$\mathrm{TWM}_0 = \frac{\text{Zahl der unter } T \text{ überdeckten Anweisungen}}{\text{Zahl aller Anweisungen}}$$

Dabei wird eine Anweisung A unter T überdeckt bzw. ausgeführt, wenn der zur Anweisung A gehörende Knoten k in Wege(T) vorkommt. Es ist also anzumerken, dass die einzelnen Anweisungen gezählt werden, d.h. ein Anweisungsknoten, in dem viele Anweisungen zusammengefasst sind, wird bei einer Übedeckung stärker ins Gewicht fallen als ein Knoten, der wenige Anweisungen enthält.

4.2.2 Zweig-, Segment- und Entscheidungsüberdeckung

Frage 4.16: Worum geht es bei der Zweigüberdeckung? Wann erfüllt eine Menge von Testdaten die Zweigüberdeckung?

Die Überdeckung von Knoten bzw. Anweisungen ist ein notwendiges Kriterium, um Fehler zu finden, jedoch kein hinreichendes. Es gibt also immer noch Fehler, die sich mit dieser Art der Überdeckung nicht finden lassen. Deshalb wird nun ein stärkeres Kriterium, die sogenannte C_1- oder Zweigüberdeckung eingeführt. Bei der Zweigüberdeckung wird nicht mehr nur gefordert, dass jeder Knoten mindestens einmal ausgeführt wird, sondern dass jede Kante mindestens einmal abgelaufen wird. Daher hat die Zweigüberdeckung auch ihren Namen, denn „Zweig“ ist ein Synonym für „Kante“. Die Definition ist der der Anweisungsüberdeckung sehr ähnlich:

Eine Testdatenmenge T erfüllt die **C_1-Überdeckung** (für ein Programm P) genau dann, wenn es für jede Kante k im Kontrollflussgraphen von P einen Weg in Wege(T,P) gibt, zu dem k gehört.

Aufgrund der Definition und dem Aussehen von Kontrollflussgraphen reicht es zur Feststellung der Zweigüberdeckung aus, sich auf die Entscheidungskanten bzw. die Segmente bzw. die Entscheidungs-Entscheidungswege zu beschränken. Dies wird jedoch erst etwas später im Text gezeigt.

Frage 4.17: Gebe ein Beispiel für ein Programm, bei dem die Anweisungsüberdeckung nicht ausreicht!

Abbildung 3 zeigt ein Programm, bei dem eine C_0-Überdeckung nicht ausreichend ist, um das Fehlverhalten aufzuzeigen. Links ist der Kontrollflussgraph des Programms, wie es eigentlich aussehen soll, rechts wie es programmiert wurde. Der Unterschied zwischen den Programmen wird klar, wenn man sie beide mit den Startwerten $x = x_0$ und $y = y_0$ ablaufen lässt. Für $x > 0$ verhalten sich beide Programme noch gleich, d.h. es ergibt sich: $x = (x_0)^2$ und $y = (x_0)^2 + 5$. Der Fehler macht sich erst bei $x \leq 0$ bemerkbar, wenn also die „nein“-Kante abgelaufen wird. In diesem Fall wird der Wert für x immer noch korrekt berechnet zu $x = x_0$, der Wert von y wird nun jedoch unterschiedlich berechnet. Das falsche Programm berechnet hier $y = y_0$ und das richtige $y = x_0 + 5$.

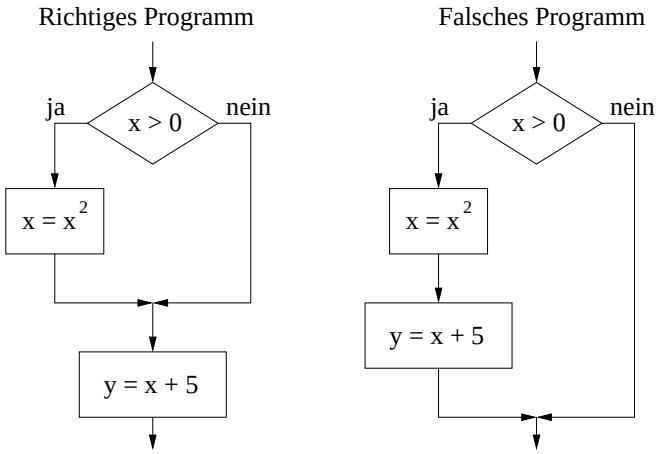


Abbildung 3: Kontrollflussfehler

Jedes Testdatum t, bei dem $x > 0$ und y beliebig ist, erfüllt also die C_0-Überdeckung, zeigt das Fehlverhalten jedoch nicht auf. So wird z.B. für $x = 5$ und $y = 0$ das Ist-Datum $y = 25$ und $y = 30$ berechnet, welches auch dem Solldatum entspricht. Der Fehler wird erst mit einem Testdatum aufgedeckt, bei dem $x \leq 0$ ist, also z.B. $x = -1$ und $y = 0$. Das Solldatum ist in diesem Fall $x = -1$ und $y = 4$. Das Ist-Datum des falschen Programms ist jedoch $x = -1$ und $y = 0$.

Frage 4.18: Zeige, dass es bei der Zweigüberdeckung reicht, sich auf die Entscheidungskanten bzw. Segmente bzw. Entscheidungs-Entscheidungs-Wege zu beschränken!
Zu zeigen ist also: Für eine Testdatenmenge T gilt:

A) T erfüllt die C_1-Überdeckung $\Leftrightarrow$

B) jede Entscheidungskante und die Anfangskante wird unter T ausgeführt $\Leftrightarrow$

C) jedes Segment wird unter T ausgeführt $\Leftrightarrow$

D) jeder Entscheidungs-Entscheidungs-Weg wird unter T ausgeführt.

Der Beweis dieser Behauptung wird durch **zyklische Implikation** erbracht. Dabei wird folgendes gezeigt: A) $\Rightarrow$ D) $\Rightarrow$ C) $\Rightarrow$ B) $\Rightarrow$ A).

Frage 4.19: Wie ist das TWM für die Zweigüberdeckung definiert?
Das TWM für die Zweig- bzw. Entscheidungsüberdeckung ist wie folgt definiert:

$$\mathrm{TWM}_1 = \frac{\text{Zahl der unter } T \text{ überdeckten Entscheidungskanten}}{\text{Zahl aller Entscheidungskanten}}$$

Hier ist anzumerken, dass nicht alle Kanten gezählt werden, sondern nur die Entscheidungskanten, da bei einer Ausführung aller Entscheidungskanten auch alle anderen Kanten ausgeführt werden.

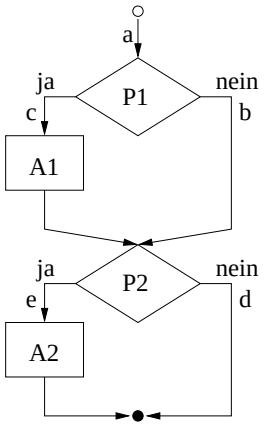


Abbildung 4: Kontrollflussgraph mit leeren Segmenten b und d

4.2.3 Kombination von Segmenten

Frage 4.20: Motiviere die Kombination von Segmenten!

Bei den bisher betrachteten Überdeckungen wurden jeweils nur Konstrukte im Kontrollflussgraph betrachtet, die sich an einer **bestimmten Stelle** im Graphen befinden. So ist es bei der Überdeckung einer Entscheidungskante (C_1-Übedeckung) z.B. egal, welche Entscheidungskante in dem Graphen vor der überdeckten Entscheidungskante ausgeführt wird. Das genaue Aussehen des Weges auf dem die Kante liegen soll ist also relativ egal, es wird nur gefordert, dass die Kante auf *irgendeinem* Weg liegt.

So reichen für das Programm in Abbildung 4 z.B. zwei Testdaten, um eine C_1-Überdeckung zu erreichen. Im ersten Fall sei $P1 = true$ und $P2 = false$; im zweiten Fall $P1 = false$ und $P2 = true$. Die beiden Testdaten ergeben dann die vollständigen Wege *abe* und *acd*, wobei alle Entscheidungskanten überdeckt werden. Was ist jedoch, wenn ein Fehler bei Ausführung der Segmente *ce* oder *bd* auftritt? Am besten wäre es also, wenn **alle Wege** ausgeführt werden würden. Diese Forderung ist jedoch in den meisten Fällen zu stark, weshalb sie in den folgenden Abschnitten etwas abgeschwächt wird.

Frage 4.21: Was versteht man unter Pfadüberdeckung?

Wie oben im Beispiel schon klar wurde, wäre es vorteilhaft, wenn man die Ausführung **aller Wege** durch das Programm als Überdeckungskriterium fordern würde. Diese Forderung ist jedoch in den meisten Fällen zu stark, da viel zu viele Fälle zu betrachten wären. Bei Programmen mit Schleifen, deren Durchlaufanzahl nicht nach oben begrenzt ist, gibt es z.B. unendlich viele Wege. Und auch bei Programmen ohne Schleifen kann die Anzahl der Wege sehr groß werden, wie Abbildung 5 zeigt. Das Programm besitzt k aufeinanderfolgende Verzweigungen und es ist leicht zu zeigen, dass es dementsprechend 2^k unterschiedliche Wege durch das Programm gibt. Daher werden nun weniger starke Kriterien eingeführt.

Frage 4.22: Wie ist die Segmentpaareüberdeckung definiert?

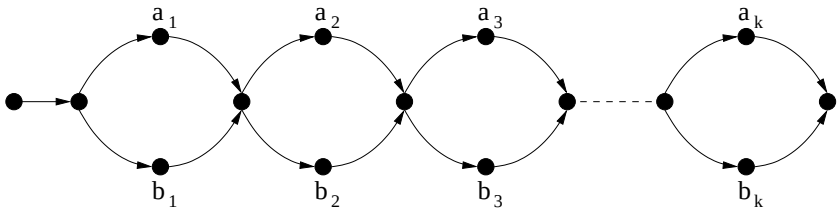


Abbildung 5: k aufeinanderfolgende Verzweigungen

Der korrektere Name für die Segmentpaareüberdeckung wäre eigentlich Entscheidungs-Entscheidungs-Weg-Paare-Überdeckung, da sie über Entscheidungs-Entscheidungs-Wege definiert wird. Sie trägt den Namen, da Segmente oft mit Entscheidungs-Entscheidungs-Wegen übereinstimmen.

Eine Testdatenmenge T erfüllt die **C_{SP}-Überdeckung** genau dann, wenn es für jedes Paar von Entscheidungs-Entscheidungs-Wegen q und r, die im Kontrollflussgraph direkt aufeinanderfolgen, ein Testdatum t in T gibt, das die Folge qr ausführt. qr ist also Teil eines Weges in Wege(T). Die C_{SP}-Überdeckung wird auch **Segmentpaareüberdeckung** genannt.

Frage 4.23: Wie ist die $C_S(n)$-Überdeckung definiert?

Die $C_S(n)$-Überdeckung ist eine **Verallgemeinerung der C_{SP}-Überdeckung** und ist wie folgt definiert:

Eine Testdatenmenge T erfüllt die **$C_S(n)$-Überdeckung** genau dann, wenn es für jede Folge von n Entscheidungs-Entscheidungs-Wegen $S_1, \ldots, S_n$, die im Kontrollflussgraph direkt aufeinanderfolgen, ein Testdatum t aus T gibt, welches die Folge $S_1, \ldots, S_n$ ausführt. Oder anders gesagt: $S_1, \ldots, S_n$ ist Teil eines Weges aus Wege(T).

Frage 4.24: Welche anderen Kriterien zur Kombination von Segmenten kennst du?

Eine weitere Überdeckungsvariante ist die LCMS[1]-Überdeckung, die sich an der Reihenfolge der Segmente im Quelltext orientiert. Aus irgendeinem Grund werden daher Sequenzen von `then`-Zweigen anders behandelt als Sequenzen von `else`-Zweigen.

Außerdem gibt es noch das Verfahren des Testens aller **fundamentaler Wege**. Dabei wird der Kontrollflussgraph als ungerichtet interpretiert und zusätzlich der Endknoten mit dem Anfangsknoten über eine Kante verbunden. Anschließend wird auf diesem Graphen ein Spannbaum berechnet, also ein kreisfreier, zusammenhängender Graph. Jede Kante des ursprünglichen Graph, die nicht im Spannbaum ist, bildet bei ihrer Hinzunahme mit einigen anderen Kanten genau einen Zyklus. Ein solcher Zyklus wird **fundamentaler Zyklus** genannt. Die Kanten, die durch ihre Hinzunahme den Zyklus bilden, werden **charakteristische Kanten** genannt.

[1] Lineare Codesequenz mit Sprung

4.2.4 Schleifenüberdeckung

Frage 4.25: Worum geht es bei der Schleifenüberdeckung? Welche Probleme gibt es dabei und wie werden sie gelöst?

Bei der Schleifenüberdeckung werden die Testdaten unter dem **Aspekt der Ausführung der Schleifen im Programm** betrachtet. Ist u die **untere Anzahl** der möglichen Schleifendurchläufe und o die **obere Anzahl**, dann müssten Testdaten gefunden werden, die die Schleife $u, u+1, \ldots, o$-mal durchlaufen, wobei auch noch alle möglichen Wege durch die Schleife gefordert werden können. Bei *for*-Schleifen mit konstanten Grenzen ist diese Forderung kein Problem, da hier $u = o$ gilt. Bei allen anderen Schleifen ist die **Anzahl der maximalen Durchläufen jedoch oft nicht bekannt bzw. sogar unbegrenzt**.

In diesem Fall kann ein Beweis über die formale Verfikation (siehe Kapitel 9.4) erfolgen oder die Anzahl der Durchläufe bzw. betrachteten Wege wird begrenzt. Zu diesem Zweck werden Wege in endlich vielen Klassen klassifiziert und aus jeder Klasse nur ein Repräsentant ausgeführt. Dabei gibt es verschiedene Möglichkeiten wie diese Wegeklassen bzw. wie fein sie gebildet werden.

Frage 4.26: Wann sind zwei Wege k-äquivalent?

Zwei Wege im Kontrollflussgraphen heißen **k-äquivalent**, wenn sie schleifenfrei und identisch sind oder wenn für jede Schleife gilt: sie durchlaufen die Schleife

1. weniger als k-mal und sind identisch (bzw. in inneren Schleifen k-äquivalent) oder
2. mindestens k-mal, wobei
   a) die ersten $k-1$ Durchläufe identisch (bzw. in inneren Schleifen k-äquivalent) sind,
   b) der k-te Durchlauf identisch (bzw. in inneren Schleifen k-äquivalent) ist, aber sich darin unterscheiden kann, dass die Kante zum Eingangsknoten der Schleife fehlt, wenn die Schleife genau k-mal durchlaufen wird,
   c) weitere Durchläufe $(k+1, k+2, \ldots)$ beliebig aussehen oder auch nicht vorhanden sein dürfen.

Im Prinzip wird diese Definition nur so kompliziert, da es vorkommen kann, dass Schleifen geschachelt sind. Für eine Schleife ohne innere Schleifen gilt bei diesem Kriterium nämlich ganz einfach, dass zwei Wege bezüglich dieses Schleifenstücks gleich sind, wenn sie bis zum k-ten Durchlauf gleich aussehen. Ab dem k-ten Durchlauf ist das Aussehen egal. Wege müssen also bei bis zu k Durchläufen gleich aussehen, danach wird alles **abgeschnitten**.

Etwas komplizierter wird es, wenn eine Schleife eine innere Schleife enthält. Dann müssen die Wege außerhalb dieser Schleife k-äquivalent sein, also bis zu k Durchläufe lang gleich aussehen und bezüglich der inneren Schleife ebenfalls k-äquivalent sein.

Um also die verschiedenen Klassen von Wegen heraus zu bekommen, können die Schleifen am besten von innen nach außen betrachtet werden.

Frage 4.27: Was drückt das Kriterium schwach 2-äquivalent aus?
Zwei vollständige Wege im Kontrollflussgraphen heißen **schwach 2-äquivalent**, wenn sie schleifenfrei und identisch sind oder wenn für jede Schleife gilt: sie durchlaufen die Schleife

1. höchstens einmal und sind identisch (bzw. in inneren Schleifen schwach 2-äquivalent) oder
2. mindestens zweimal, wobei der erste Durchlauf identisch ist (bzw. in inneren Schleifen schwach 2-äquivalent).

Das Kriterium ist also bezüglich der 2-Äquivalenz insofern abgeschwächt, als dass hier der zweite Durchlauf schon nicht mehr gleich aussehen muss, wenn die Schleife zwei- oder mehrfach durchlaufen wird.

Frage 4.28: Wie ist die strukturierte Pfadüberdeckung definiert?
Für $k > 0$ erfüllt eine Testdatenmenge T die **C_i(k)-Überdeckung** genau dann, wenn Wege(T) mindestens einen Weg aus jeder Klasse von k-äquivalenten Wegen enthält. (Beachte: Wege(T) ist die beim Ausführen der Tests aus T erzeugte Menge von ausgeführten vollständigen Wegen.) Die C_i(k)-Überdeckung heißt auch **strukturierte Pfadüberdeckung**.

Frage 4.29: Wie ist die (starke/schwache) C_{GI}-Überdeckung definiert?

1. Eine Testdatenmenge T erfüllt die **starke C_{GI}-Überdeckung** genau dann, wenn die Testdatenmenge T die **C_i(2)-Überdeckung** erfüllt.
2. Eine Testdatenmenge T erfüllt die **C_{GI}-Überdeckung** genau dann, wenn Wege(T) mindestens einen Weg aus jeder Klasse von **schwach 2-äquivalenten Wegen** enthält.
3. Eine Testdatenmenge T erfüllt die **schwache C_{GI}-Überdeckung** genau dann, wenn die Testdatenmenge T die **C_i(1)-Überdeckung** erfüllt.

Frage 4.30: Erkläre die C_{GI}-Überdeckungen anhand eines Beispiels!
Gegeben sei der Kontrollflussgraph aus Abbildung 6, der aus zwei geschachtelten *repeat*-Schleifen besteht. Die Klassen werden nun gebildet, indem dieser Graph **von Innen nach Außen** betrachtet wird. Dann gibt es jeweils die folgenden Klassen von Wegen bei den C_{GI}-Überdeckungen:

Schwache C_{GI}-Überdeckung: Um die schwache C_{GI}-Überdeckung zu erfüllen, muss eine Testdatenmenge die C_i(1)-Überdeckung erfüllen. Es wird nun erst einmal geschaut, welche Klassen es bezüglich der C_i(1)-Überdeckung in der inneren *repeat*-Schleife gibt. Die innere Schleife muss also im Sinne der schwachen C_{GI}-Überdeckung entweder 0-mal (weniger als 1-mal) durchlaufen werden oder einmal und danach ist egal, wie sie aussieht. Bei einer *repeat*-Schleife ist es unmöglich, diese 0-mal zu durchlaufen, und bei einem Mal hat man die Wahl zwischen b und c und danach kann irgendwas

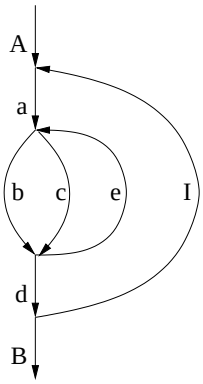


Abbildung 6: Geschachtelte *repeat*-Schleifen

passieren, d.h. die Schleife wird entweder verlassen oder noch ein paarmal durchlaufen. Dies wird durch die Menge $M = \{bx, cx\}$ beschrieben. Für die innere Schleife gibt es also zwei Äquivalenzklassen bezüglich der schwachen C_{GI}-Überdeckung.

Für die äußere Schleife gilt im Prinzip fast das gleiche. Es ist nicht möglich sie 0-mal zu durchlaufen, da es eine *repeat*-Schleife ist und bei einem oder mehr Durchläufe ergibt sich: $AaydzB$ mit $y \in M$ und z entweder die leere Sequenz oder beliebig viele andere Durchläufe. Also ergeben sich insgesamt die beiden Klassen $AabxdzB$ und $AacxdzB$.

C_{GI}-Überdeckung: Das Vorgehen dürfte nun aus obigem Beispiel klar sein.

- Die Wege sollen weniger als 2-mal durch die Schleife führen und identisch sein: Dies sind die Wege b und c.
- Die Wege sollen mindestens zweimal durch die Schleife führen und nur beim ersten Mal identisch sein (dies ist der Unterschied zur starken C_{GI}-Überdeckung!): Also bex und cex.

Also ergibt sich die Menge $K = \{b, c, bex, cex\}$ dieser vier Wegeklassen für die innere Schleife. Diese können nun wieder bei der Betrachtung der äußeren Schleife beliebig eingesetzt werden.

- Einmal durch: $AaxdB$ mit $x \in K$.
- Mindestens zweimal, nur das erste Mal identisch: $AaxdIay$ mit $x \in K$ und y mindestens ein weiterer Durchlauf durch die äußere Schleife.

Dementsprechend ergeben sich insgesamt $4 \cdot 2 = 8$ Wegeklassen.

Starke C_{GI}-Überdeckung: Im Gegensatz zur C_{GI}-Überdeckung müssen die Wege beim zweiten Durchlauf auch noch identisch sein (und danach kann kommen, was will). Also für die innere Schleife:

- Einmal durch: b oder c.

- Mindestens zweimal durch und beim zweiten mal gleich: $bebx$, $becx$, $cebx$ oder $cecx$. Wobei x ein beliebiges Weiterlaufen ist.

Für die innere Schleife gibt es also schon sechs Wegeklassen, die wieder in der Menge $L = \{b, c, bebx, becx, cebx, cecx\}$ zusammengefasst werden. Für die äußere gilt:

- Einmal durch: $AaxdB$ mit $x \in L$
- Mind. zweimal durch und die beiden ersten Mal gleich: $AaxdIaydzB$ mit $x, y \in L$ und z beliebig.

Insgesamt gibt es also $6 + 6 \cdot 6 = 6 + 36 = 42$ Klassen.

Frage 4.31: Welche weiteren Kriterien kennst du, die den Schleifenkriterien recht ähnlich sind?

Desweiteren können Kriterien angegeben werden, die die Häufigkeit betrachten, mit der Knoten Kanten oder Wegstücke des Kontrollflussgraphen durchlaufen werden. Ein vollständiger Weg w in einem Kontrollflussgraphen heißt:

k-iterativ: wenn jedes Wegstück v höchstens k-mal iteriert in w vorkommt, d.h. es gibt keine Wegstücke u, v, x mit $w = uv^{k+1}x$, wobei v nicht die Länge 0 hat.

einfach: wenn jede Kante höchstens einmal in w vorkommt.

2-Knoten-iterativ: wenn jeder Knoten höchstens zweimal im Weg w vorkommt.

Es sei angemerkt, dass ein k-iterativer Weg eine Schleife öfter als k-mal durchlaufen kann und einfache Wege öfter als einmal.

Das Kriterium „teste alle k-iterativen Wege“ ist jedoch leider nicht anwendbar, da hier auch schon für kleines k unendlich lange Wege entstehen können. Für $k = 1$, kann schon für eine Schleife, bei der es drei verschiedene Möglichkeiten des Durchlaufs gibt, ein unendlich langer Weg konstruieren.

5 Datenflussbezogenes Testen

5.1 Problemstellung und Modellbildung

Frage 5.1: Welche Ursachen kann es haben, wenn eine Anweisung ein falsches Ergebnis liefert?
Es können zwei Ursachen unterschieden werden, die dazu führen können, dass eine Anweisung ein falsches Ergebnis liefert:

- Die Anweisung selbst ist falsch. Dies ist z.B. der Fall, wenn im Programm $A = B + C * D$ steht, es jedoch richtig $A = B * C * D$ heißen muss.
- Die Anweisung selbst ist korrekt, jedoch wurde einer der referenzierten Werte vorher falsch berechnet. In der Anweisung $A = B + C * D$ könnte z.B. eine der Variablen B, C oder D falsch berechnet worden sein.

Diese Kapitel beschäftigt sich mit dem zweiten Grund; auf den ersten wird in Kapitel 6 eingegangen. Um die Fehler, die durch eine falsch berechnete Referenz zustande kommen zu finden, muss der Datenfluss ab der fehlerhaften Anweisung rückwärts betrachtet werden und die Interaktion zwischen Definitionen und Referenzen näher untersucht werden. Dazu wird die Definition des Kontrollflussgraph ein wenig erweitert.

Frage 5.2: Wie ist ein Datenflussgraph definiert? Wie ein Datenflussschema?
Ein **Datenflussgraph** ist ein Kontrollflussgraph, bei dem zusätzlich zu jedem Knoten k die Mengen DEF(k), UNDEF(k) und REF(k) gehören.

DEF(k) ist die Menge der Variablen x, für welche die Anweisungsfolge f, die zum Knoten k gehört, der Variablen einen Wert zuweist, der nicht anschließend in f undefiniert wird.

UNDEF(k) ist die Menge der Variablen x, für welche die Anweisungsfolge f, die zum Knoten k gehört, die Variable x in einen undefinierten Zustand überführt, ohne x anschließend in f neu zu definieren.

REF(k) ist die Menge der Variablen x, für welche die Anweisungsfolge f, die zum Knoten k gehört, die Variable x referenziert, ohne dass x vorher in f undefiniert wird. (Dabei wird generell vorausgesetzt, dass ein lokaler Datenfluss innerhalb eines Knotens k nicht vorkommt. Andernfalls ist die Anweisungsfolge entsprechend auf zwei oder mehrere Knoten aufzuteilen.)

Wenn der Datenflussgraph eine Funktion repräsentiert, die von einem aufrufenden Modul Informationen erhält (über Parameter oder globale Variablen), so wird ein Knoten k_{ein} zum Kontrollflussgraphen hinzugefügt. DEF(k_{ein}) ist die Menge der Variablen, die Informationen importieren. Von k_{ein} führt eine Kante zu dem bisherigen Startknoten des Kontrollflussgraphen.

Entsprechendes gilt, wenn die beschriebene Funktion Informationen an das aufrufende Modul zurückgibt (über Parameter oder globale Variablen): Ein

Knoten k_{aus} wird zum Kontrollflussgraphen hinzugefügt und von dem bisherigen Endknoten des Kontrollflussgraphen führt eine Kante zum Knoten k_{aus}. REF(k_{aus}) ist die Menge der Variablen, die Informationen exportieren.

Das **Datenflussschema** zu einem Datenflussgraphen entsteht durch Ersetzen der Anweisungen und Entscheidungsprädikate durch formale Bezeichner.

Frage 5.3: Was ist ein lokaler Datenfluss?

Ein **lokaler Datenfluss** in einem Knoten k liegt vor, wenn eine Variable in der zu k gehörenden Anweisungsfolge definiert und danach referenziert wird ohne zwischenzeitlich undefiniert zu werden. Da jedoch gerade die Interaktion zwischen der Definition und der Referenz einer Variablen getestet werden soll, ist dies ungünstig, so dass ein Kontroll- bzw. Datenflussgraph, bei dem so etwas auftritt, modifiziert werden muss, indem aus dem einen Knoten zwei Knoten gemacht werden.

Frage 5.4: Was muss in einem Datenflussgraphen für die Entscheidungsknoten gelten?

Für die Entscheidungsknoten k eines Datenflussgraphen muss gelten, dass in ihm nur Variablen referenziert werden. Es muss also $DEF(k) = UNDEF(k) = \emptyset$ gelten. Es darf also keine Anweisung der Form `if(b=c+d)` geben, wie dies unter anderem in C/C++ möglich ist.

Frage 5.5: Was versteht man unter der Sprechweise „eine Definition von x im Knoten k erreicht eine Referenz von x im Knoten l über den Weg w“?

Eine Definition von x im Knoten k erreicht eine Referenz von x im Knoten l über den Weg w genau dann, wenn der Weg w im Kontrollflussgraphen von k nach l führt und die Variable x auf diesem Weg nicht neu definiert oder undefiniert wird. Oder etwas formaler:

Sei $x \in$ DEF(k), $x \in$ REF(l), $w = (k_1, \ldots, k_m)$ ein Weg im Datenflussgraphen mit $k_1 = k, k_m = l$. Dann gilt für jedes i mit $1 < i < m : x \notin DEF(k_i), x \notin UNDEF(k_i)$.

5.2 Einfache Datenflusskriterien

Frage 5.6: Wann erfüllt eines Testdatenmenge das Kriterium „alle Definitionen“?

Eine Testdatenmenge T erfüllt das Kriterium **alle Definitionen** genau dann, wenn es für jede Variable x und jede Definition von x mindestens einen Weg in Wege(T) gibt, auf dem die Definition eine Referenz von x erreicht.

Frage 5.7: Wann erfüllt eines Testdatenmenge das Kriterium „alle DR-Interaktionen“?

In der Definition zu „alle Definitionen“ wurde nur gefordert, dass zu jeder Definition einer Variablen eine Referenz erreicht wird. Jedoch kann es im Programm mehrere Referenzen einer Variable geben und diese können auch alle zu Fehlern im Programm führen. Es liegt also nahe, die Forderung noch zu verstärken:

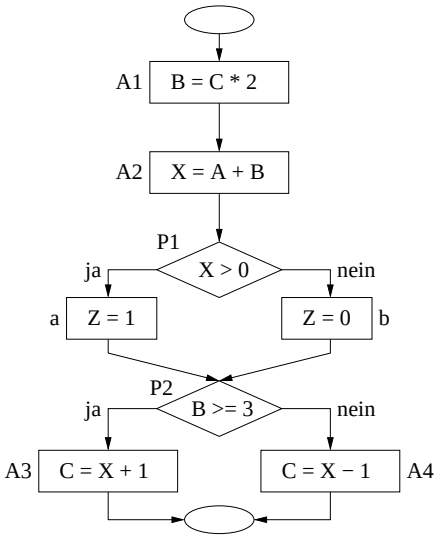


Abbildung 7: Flussdiagramm mit Definitionen und Referenzen von B und X

Eine Testdatenmenge T erfüllt das Kriterium **alle DR-Interaktionen** genau dann, wenn es für jede Variable x, jede Definition von x und jede Referenz von x, die davon erreicht wird, mindestens einen Weg in Wege(T) gibt, auf dem die Definition die Referenz von x erreicht.

Es werden bei diesem Kriterium also **alle Paare** von Definitionen und Referenzen getestet. Der Name ist anscheinend eine Abkürzung für alle Definitions-Referenz-Interaktionen. Leider werden bei diesem Kriterium jedoch nicht alle Entscheidungskanten ausgeführt (es findet also keine Entscheidungsüberdeckung im Sinne von Kapitel 4 statt).

Frage 5.8: Wann erfüllt eines Testdatenmenge das Kriterium „alle Referenzen“?

Leider werden bei dem Kriterium „alle DR-Interaktionen“ nicht alle Entscheidungskanten ausgeführt. Wenn verhindert werden soll, dass nur ein Testdatum ausgewählt wird, welches einen Weg abläuft, der an einem Entscheidungsknoten genau den Weg nimmt, bei dem ein Fehler *nicht* auftaucht, muss also dafür gesorgt werden, dass alle Entscheidungskanten abgelaufen werden. Dafür sorgt die folgende Definition, wenn auch, in Bezug auf normale Anweisungsknoten, etwas plump:

Eine Testdatenmenge T erfüllt das Kriterium **alle Referenzen** genau dann, wenn für jede Variable x, jede Definition von x in einem Knoten k, jede Referenz von x in einem Knoten l, die von der Definition in k erreicht wird, und für jeden Nachfolgerknoten m von l die Wegemenge Wege(T) mindestens ein Wegstück um enthält, wobei die Definition von x in k die Referenz von x in l über den Weg u erreicht.

Frage 5.9: Wann erfüllt eine Testdatenmenge das Kriterium „alle E-/einige B-Referenzen“? Wofür steht diese Abkürzung?

Die Abkürzung steht für das Kriterium **„alle Entscheidungsreferenzen/einige Berechnungsreferenzen“**, welches wie folgt definiert ist:

Eine Testdatenmenge T erfüllt das das Kriterium **alle E-/einige B-Referenzen** genau dann, wenn für jede Variable x und jede Definition von x in einem Knoten k gilt:

1. für jede Referenz von x in einem Entscheidungsknoten (im folgenden l genannt), die von der Definition in k erreicht wird, und jeden Nachfolger m von l enthält Wege(T) mindestens ein Wegstück um, wobei die Definition von x in k die Referenz von x in l über u erreicht;

2. falls es keine Referenzen von x in Entscheidungsknoten gibt, die von der Definition in k erreicht werden, enthält Wege(T) mindestens ein Wegstück u, auf dem die Definition von x in k irgendeine Referenz von x erreicht.

Mit der ersten Forderung wird die Zweig- bzw. Entscheidungsüberdeckung erreicht. Der zweite Teil entspricht dem Kriterium „alle Definitionen“.

Frage 5.10: Wann erfüllt eine Testdatenmenge das Kriterium „alle B-/einige E-Referenzen“?

Eine Testdatenmenge T erfüllt das das Kriterium **alle B-/einige E-Referenzen** genau dann, wenn für jede Variable x und jede Definition von x in einem Knoten k gilt:

1. für jede Referenz von x in einem Knoten (im folgenden l genannt), der nicht Entscheidungsknoten ist, aber von der Definition in k erreicht wird, enthält Wege(T) mindestens ein Wegstück u, wobei die Definition von x in k die Referenz von x in l über u erreicht;

2. falls die Definition von x in k nur in Entscheidungsknoten referenziert wird, enthält Wege(T) mindestens ein Wegstück u, auf dem die Definition von x in k irgendeine Referenz von x erreicht.

Mit dem ersten Punkt wird erreicht, dass in Berechnungsknoten das Kriterium „alle Referenzen“ erfüllt ist (auch wenn das bei Berechnungsknoten nicht sehr spannend ist und im Prinzip dem Kriterium „alle DR-Interaktionen“ entspricht). Mit Punkt zwei wird wieder mindestens das Kriterium „alle Definitionen“ garantiert.

Frage 5.11: Wann erfüllt eine Testdatenmenge das Kriterium „alle DR-Wege“?

Eine Testdatenmenge T erfüllt das das Kriterium **alle DR-Wege** genau dann, wenn für jede Variable x, jede Definition von x in einem Knoten k, jede Referenz von x in einem Knoten l, die von der Definition in k erreicht wird, und für jeden Nachfolgerknoten m von l die Wegemenge Wege(T) jedes Wegstück um mit folgenden Eigenschaften enthält:

1. die Definition von x in k erreicht die Referenz von x in l über den Weg u

2. u ist frei von Zyklen oder u ist ein einfacher Zyklus.

Bei diesem Kriterium werden also fast alle Interaktionen zwischen einer Definition und einer Referenz getestet, wobei jedoch keine Schleifen iteriert werden.

5.3 Verkettung von Datenflüssen

Frage 5.12: Worum geht es bei der Verkettung von Datenflüssen?
Wie schon in der Einleitung dieses Kapitels erwähnt, ist ein Ziel der datenflussorientierten Testverfahren, den Datenfluss bzw. die Interaktionen zwischen Definitionen und Referenzen rückwärts durch den Datenflussgraph zu verfolgen. Dies wird nun in diesem Unterkapitel konkretisiert, wobei nun für jede Referenz einer Variablen mehr oder weniger lange Ketten zurück durch den Graphen gebildet werden.

Frage 5.13: Was ist eine k-DR-Interaktion?
Für $k \geq 2$ heißt eine Folge $I = (l_1, x_1, l_2, x_2, \ldots, x_{k-1}, l_k)$ eine **k-DR-Interaktion** genau dann, wenn

1. $l_1, l_2, \ldots, l_k$ Knoten des Datenflussgraphen sind, wobei nur die Knoten l_1 und l_k identisch sein dürfen,
2. $x_1, \ldots, x_{k-1}$ Variablen sind (die nicht notwendigerweise verschieden sind),
3. es für jedes i mit $1 \leq i < k$ einen Weg w_i im Datenflussgraphen gibt, so dass gilt: Variable x_i wird im Knoten l_i definiert und diese Definition erreicht eine Referenz von x_i in Knoten l_{i+1} über den Weg w_i.

Es sei z.B. das Programm aus Abbildung 8 gegeben, welches eine binäre Suche auf einem Array A der Länge N ausführt, um das Datum F zu finden. Dann ist die Folge $I_1 = (1,\ \mathit{NPOS},\ 11)$ eine 2-DR-Interaktion. Oder ein Beispiel für eine 4-DR-Interaktion: $I_2 = (1,\ \mathit{LOW},\ 2,\ \mathit{MID},\ 6,\ \mathit{NPOS},\ 11)$.

Frage 5.14: Was ist ein Interaktionsweg?
Sei $I = (l_1, x_1, \ldots, x_{k-1}, l_k)$ eine k-DR-Interaktion. Ein Weg im Datenflussgraph der Form $w = w_1 \ldots w_{k-1}$ ist ein **Interaktionsweg für I** genau dann, wenn für jedes i mit $1 \leq i < k$ w_i ein Weg ist, über den eine Definition von x_i im Knoten l_i eine Referenz von x_i im Knoten l_{i+1} erreicht.

Ein Interaktionsweg für die oben als Beispiel aufgeführte Folge I_1 ist die Knotenfolge $(1, 2, 3, 4, 11)$. Oder für I_2 die Folge $(1, 2, 3, 5, 6, 7, 11)$. Es kann jedoch mehrere Interaktionswege zu einer k-DR-Interaktion geben.

Frage 5.15: Wann erfüllt eine Testdatenmenge das Kriterium „alle k-DR-Interaktionen“?
Sei k eine natürliche Zahl, $k \geq 2$. Eine Testdatenmenge T erfüllt das Kriterium **alle k-DR-Interaktionen** genau dann, wenn für jedes m mit $2 \leq m \leq k$, für jede m-DR-Interaktion $I = (l_1, x_1, \ldots, x_{m-1}, l_m)$ und für jeden Nachfolgerknoten n von l_m die Menge Wege(T) mindestens ein Wegstück un, enthält, wobei u ein Interaktionsweg für I ist.

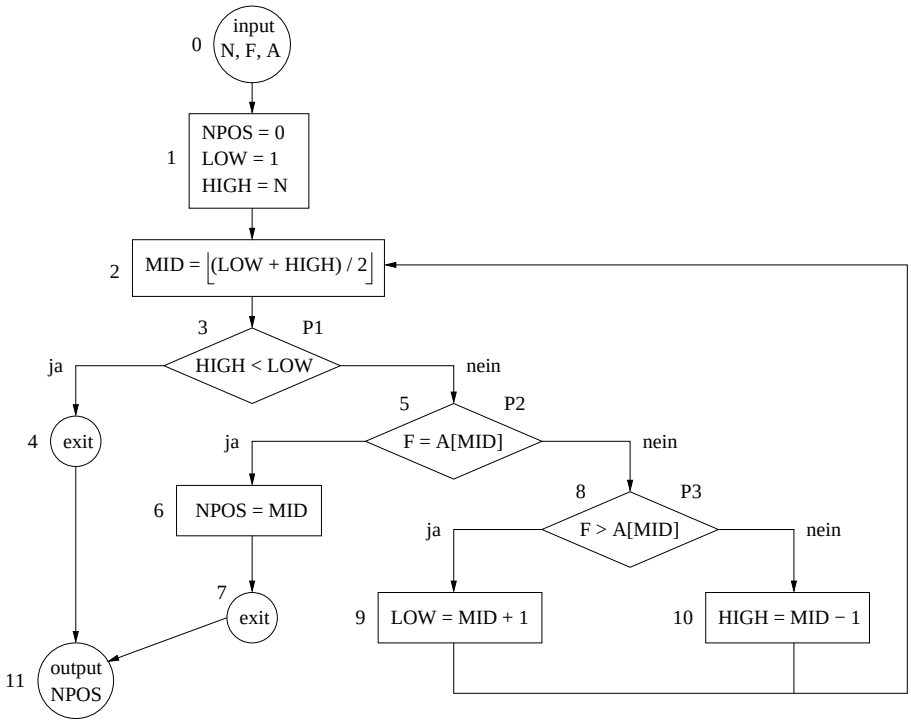


Abbildung 8: Suchprogramm *NPOS = SEARCH(N, A, F)*

5.4 Parallele Betrachtung von Datenflüssen

Frage 5.16: Worum geht es bei der parallelen Betrachtung von Datenflüssen?

Bei der Verkettung von Datenflüssen in Kapitel 5.3 wurde für jede einzelne Variable eine Kette von Interaktionen gebildet, die zur Referenz dieser Variablen führt. Sei z.B. die Anweisung $A = B + C * D$ gegeben. Dann würden bei der Verkettung von Datenflüssen jeweils für die Variable B, C und D Ketten gebildet werden, die unabhängig voneinander betrachtet werden. Bei der parallelen Betrachtung wird nun quasi für jede an der betrachteten Anweisung beteiligten Variable bzw. Referenz einer Variablen „einen Schritt zurück“ zur Definition dieser Variablen gegangen.

Frage 5.17: Wie ist ein Definitionskontext definiert? Was ist ein Kontextweg?

Eine Menge $DK = \{(k_1, x_1), (k_2, x_2), \ldots, (k_n, x_n)\}$ ist ein Definitionskontext für einen Knoten k des Datenflussgraphen mit $REF(k) = \{x_1, \ldots, x_n\} \neq \emptyset$, wenn folgendes gilt:

1. für jedes i mit $1 \leq i \leq n$ ist k_n ein Knoten, in dem die Variable x_i definiert wird,

2. es existiert ein **Kontextweg** für DK, mit folgender Eigenschaft für jedes i mit $1 \leq i \leq n$: es gibt eine Aufteilung von w in zwei Teilstücke w_i und

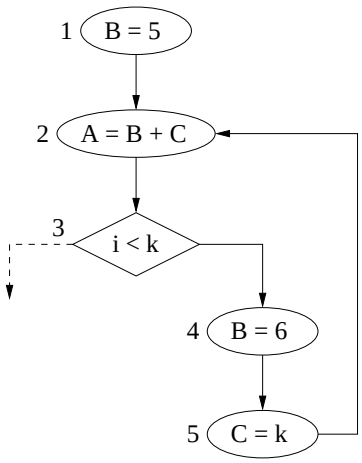


Abbildung 9: Beispiel für Definitionskontexte und -wege

v_i ($w = w_i v_i$), so dass die Definition von x_i im Knoten k_i die Referenz von x_i in k über den Weg v_i erreicht.

Der Weg wird im Punkt 2 dieser Aufzählung in zwei Teilwege aufgesplittet, da es wichtig ist, dass die Referenz von x_i über den **hinteren** Teil des Weges w, also über v_i, erreicht wird.

Gegeben sei der Beispielgraph aus Abbildung 9. Für diesen ist die Menge $\{(1, B),\ (5, C)\}$ z.B. *kein* Definitionskontext für den Knoten 2, da der einzig mögliche Weg der Weg $(1, 2, 3, 4, 5, 2)$ ist (ansonsten würde man nicht mehr zu Knoten 1 kommen) und in diesem die Definition von B im Knoten 1 nicht die Referenz von B im Knoten 2 im Sinne der Definition erreicht. Dies liegt daran, dass die Variable B im Knoten 4 neu definiert wird. Es ist also bei den Definitionswegen wichtig, dass wenn eine Definition den betrachteten Knoten erreicht, diese nicht wieder undefiniert oder redefiniert wird.

Desweiteren kann es passieren, dass es **verschiedene** Kontextwege für eine Kontextüberdeckung gibt.

Frage 5.18: Wann erfüllt eine Testdatenmenge das Kriterium „Kontextüberdeckung“?

Eine Testdatenmenge T erfüllt das Kriterium **Kontextüberdeckung** genau dann, wenn für jeden Knoten k des Datenflussgraphen und jeden Definitionskontext DK von k die Wegemenge Wege(T) mindestens ein Wegstück enthält, welches Kontextweg für DK ist.

Frage 5.19: Wie kann das Kriterium „Kontextüberdeckung“ verschärft werden?

Bisher wurden die Definitionskontexte ungeordnet betrachtet. Um das Kriterium der Kontextüberdeckung nun zu verschärfen, werden diese nun als geordnet betrachtet, wodurch auch geordnete Kontextwege entstehen. Bei einem geordneten Definitionskontext wird nun gefordert, dass die Definitionen der Variablen in der Reihenfolge im Kontextweg vorkommen, wie diese im Definitionskontext stehen. Angenommen es ist der ungeordnete Definitionskontext $DK =$

$\{(9, LOW),\ (10, HIGH)\}$ gegeben. Wenn es in diesem nun Kontextwege gibt, die einmal in 9 beginnen und dann über 10 wandern und umgekehrt, so können daraus zwei geordnete Definitionskontexte $GDK_1 = \{(9, LOW),\ (10, HIGH)\}$ und $GDK_2 = \{(10, HIGH),\ (9, LOW)\}$ gebildet werden.

Bei dem verschärften Kriterium wird dann gefordert, dass für jeden geordneten Definitionskontext mindestens ein Kontextweg als Teilstück in Wege(T) vorkommt. Während man beim „normalen“ Kriterium also für den Definitionskontext DK nur einen Kontextweg als Teilstück fordert, werden bei den Definitionskontexten GDK_1 und GDK_2 also zwei Wege gefordert.

Frage 5.20: Wie funktioniert das Definitionsbaumtesten?

Das Definitionsbaumtesten ist quasi eine Kombination aus den Kriterien „alle k-DR-Interaktionen“ und der Kontextüberdeckung. Es wird für einen Knoten ein Definitionskontext betrachtet und von diesem aus weitere Ketten von Definitionskontexten entsprechend der k-DR-Interaktion gebildet.

6 Ausdrucks-, anweisungs- und datenbezogenes Testen

In diesem Kapitel werden nun die einzelnen Ausdrücke und Anweisungen näher betrachtet. Dabei werden Kriterien aufgestellt, die Testdaten erfüllen müssen, um bestimmte Fehler in diesen zu finden.

6.1 Ausdrucks- und anweisungsbezogenes Testen

Frage 6.1: Zur Erinnerung: Welche Ursachen können Fehler in Anweisungen haben?

Für Fehler in Anweisungen kann es zwei Ursachen geben:

1. Die Anweisung ist korrekt, aber die referenzierten Werte wurde zuvor falsch berechnet (siehe Kapitel 5).
2. Die Anweisung ist falsch.

Mit der zweiten Ursache beschäftigt sich dieses Kapitel. Es wird sich dabei zeigen, dass die einzelnen Anweisungen zur Erkennung solcher Fehler präziser modelliert werden müssen und dass es auch nicht ausreicht ein Programm mit irgendwelchen Testdaten zu testen, sondern dass die Testdaten zum Aufspüren bestimmter Fehler auch bestimmte Kriterien bzw. Bedingungen erfüllen müssen.

Frage 6.2: Welche Fehlerarten können beim ausdrucks- und anweisungsbezogenen Testen unterschieden werden?

Die einzelnen Anweisungen in den Anweisungsknoten wurden **bisher recht grob modelliert**. Es wurde im Prinzip nur unterschieden, ob eine Variable definiert oder referenziert wurde, d.h. auf welche Variable zugegriffen und in welche etwas gespeichert wurde. Um Fehler innerhalb einer Anweisung finden zu können, was das Ziel dieses Kapitels ist, wird es jedoch notwendig, die einzelnen Anweisungen genauer zu betrachten und zu modellieren. Dabei werden die **Ausdrücke** in den einzelnen Anweisungen näher betrachtet sowie die **Relationen** dieser untereinander. Insgesamt ergeben sich also folgende Faktoren, die unterschieden werden:

1. Datenzugriff
2. Datenspeicherung
3. arithmetischer Ausdruck
4. arithmetische Relation
5. Boolescher Ausdruck

Die grobe Modellierung wird also einfach um eine feinere Modellierung erweitert.

Frage 6.3: Was versteht man unter dem Ansatz der „schwachen Mutationsanalyse“?

Bei der **schwachen Mutationsanalyse** wird nur gefordert, dass ein **fehlerhafter Programmzustand direkt nach einer fehlerhaften Anweisung** vorliegt. Bei der „normalen“ Mutationsanalyse wird eine stärkere Forderung gestellt, nämlich, dass sich ein Fehler bis zum Programmende bzw. bis zu einer Programmausgabe fortpflanzt. Dies muss nicht immer der Fall sein, da eine fehlerhafte Anweisung, bis das Programmende erreicht ist, durch andere Anweisungen überdeckt werden kann.

Frage 6.4: Beschreibe die Kriterien, die nötig sind, um einen falschen Datenzugriff zu erkennen!

Bei dieser Fehlerart wird auf eine falsche Variable zugegriffen. Damit dieser Fehler bemerkt werden kann, müssen die Testdaten so gewählt sein, dass alle Variablen vor dem Zugriff verschiedene Werte haben. Dieses Kriterium wird **Datenzugriffskriterium** genannt. Die Idee dabei scheint zu sein, dass wenn alle Variablen vorher verschieden sind und eine Variable nach der Anweisung einen falschen Wert hat, es möglich ist, die dafür verantwortliche Variable zu erkennen.

Wird z.B. die Anweisung $\lfloor MID = \frac{HIGH+LOW}{2} \rfloor$ des Programms aus Abbildung 8 auf Seite 44 ausgeführt, so muss die Differenz zwischen den Variablen LOW und $HIGH$ mindestens zwei betragen, da ansonsten $MID = LOW = HIGH$ gelten würde, in dem Fall, dass $LOW = HIGH$ gelten würde (was eh nicht sein darf, da alle Variablen verschieden sein sollen) oder $MID = LOW$, wenn $HIGH = LOW + 1$ gilt.

Frage 6.5: Beschreibe die Kriterien, die nötig sind, um eine fehlerhafte Datenspeicherung zu erkennen!

Bei dieser Art des Fehlers wird ein Wert in einer falschen Variablen gespeichert. Um diesen Fehler zu erkennen, muss der Wert, der einer Variablen zugeordnet wird, anders sein als der bisherige Wert. Dieses Kriterium heißt **Datenspeicherungskriterium**.

Wird zum Beispiel das Programm in Abbildung 8 mit einem Array der Größe 1 (also $N = 1$) und mit $F > A[1]$ (das gesuchte Datum befindet sich also nicht im Array) ausgeführt, so wird der Variablen MID in Anweisung 2 zweimal der Wert 1 zugeordnet. Angenommen beim zweiten Mal soll eigentlich eine andere Variable diesen Wert speichern, so würde dies nicht auffallen, da MID ja immer noch seinen richtigen Wert besitzen würde. Es würde also eventuell auffallen, dass die Variable, die beim zweiten Mal eigentlich den Wert speichern soll, sich nicht verändert, man würde jedoch nicht so schnell die Variable finden, die den Wert fälschlicherweise zugewiesen bekommen hat.

Frage 6.6: Was ist ein arithmetischer Ausdruck? Was ist ein Polynom? Was ist ein Multinom?

Die im folgenden betrachteten arithmetischen Ausdrücke bestehen aus Variablen oder Konstanten als Operanden und den Operatoren $+, -, *, /$ und $**$ (Exponentation). Wenn ein Ausdruck keine Division / enthält und nur eine Variable vorkommt, so ist er ein **Polynom**. Ansonsten, wenn mehrere Variablen vorkommen, ist es ein **Multinom**.

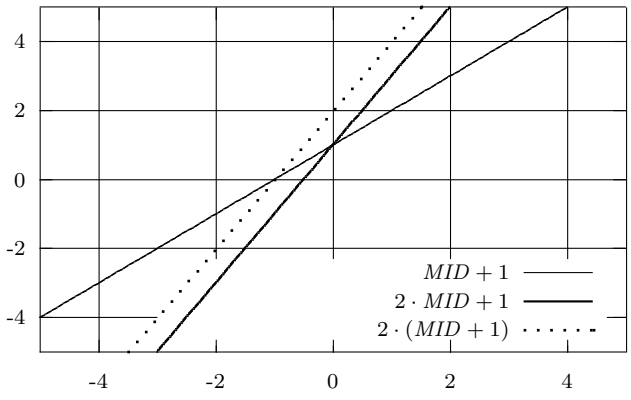


Abbildung 10: Auswirkung multiplikativer Fehler

Frage 6.7: Zu welchen Arten von Fehlern kann es bei arithmetischen Ausdrücken kommen?

Bei arithmetischen Ausdrücken können unter anderem folgende Arten von Fehlern vorkommen:

1. einfache additive oder multiplikative Fehler,
2. ein Fehler in einem Polynom oder Multinom, welcher die Variablenmenge nicht ändert und den höchsten Exponenten im Ausdruck nicht erniedrigt.

Wie müssen nun die Testdaten für die beiden Fälle aussehen?

1. Um einen multiplikativen Fehler zu finden, muss dafür gesorgt werden, dass der betroffene (Teil-)Ausdruck einen von 0 verschiedenen Wert erhält, damit der Fehler sich bemerkbar macht. Additive Fehler wirken sich dagegen immer aus. Diese Art von Kriterium heißt das **additive/multiplikative Fehler-Kriterium**.

   Wieder wird das Programm aus Abbildung 8 als Beispiel herangezogen. In Knoten 9 wird die Zuweisung $LOW = MID + 1$ gemacht. Angenommen, $k * MID + 1$ wäre der eigentlich korrekte Ausdruck. Der betroffene Teilausdruck wäre in diesem Fall MID und für $MID = 0$ würde der Fehler tatsächlich nicht auffallen, da LOW in beiden Fällen den Wert 1 zugewiesen bekommen würde. Ebenso wenn der eigentlich korrekte Ausdruck $k*(MID+1)$ wäre. In diesem Fall wäre der betroffene Ausdruck $MID+1$ und für $MID = -1$ würde der Fehler wiederum nicht auffallen. Abbildung 10 verdeutlicht dies für $k = 2$, wobei

   Additive Fehler wirken sich dagegen immer aus, was man sich schnell klar machen kann. Angenommen, der richtige Ausdruck sei $A(x)$ und der fehlerhafte Ausdruck sei $A(x)+k$ mit $k \neq 0$. Dann gibt es keine Lösung für die Gleichung $A(x) = A(x)+k$, da sich das ganze zu $0 = k$ wegkürzen ließe. Es gäbe also kein Testdatum x, bei dem sich für die beiden Ausdrücke ein gleicher Wert ergäbe. Dies kann man sich, wie in Abbildung 11 gezeigt, auch graphisch recht schnell klarmachen.

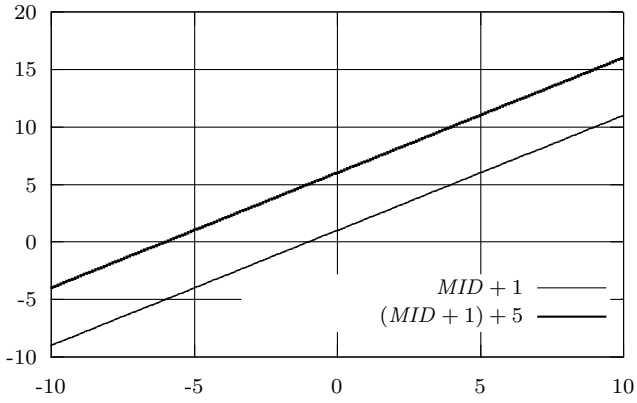


Abbildung 11: Auswirkung additiver Fehler

Frage 6.8: Wie sind arithmetische Relationen definiert? Zu welchen Arten von Fehlern kann es dabei kommen? Welche Testdaten werden benötigt?
Eine Relation ist der Form $A\ r\ B$ gegeben, wobei A und B zwei arithmetische Ausdrücke sind und r eines der Relationssymbole $<, =, >, \leq, \geq$ oder $\neq$. Dabei kann es zu zwei Arten von Fehlern kommen:

1. Die Relation $A\ r\ B$ enthält ein falsches Relationssysmbol r.
2. Die Relation $A\ r\ B$ ist falsch. Eigentlich müsste $(A + k)\ r\ B$ realisiert werden, wobei $k \neq 0$ gilt.

In beiden Fällen sind drei Testdaten nötig. Für diese muss jeweils gelten, dass $A - B$ knapp unter Null liegt (also der größte, negative Wert), $A - B = 0$ und $A - B$ knapp über Null (also der kleinste, positive Wert). Dieses Kriterium wird das **arithmetische Relations-Kriterium** genannt.

Doch warum wird genau so getestet? Im Prinzip bestehen die Relationssymbole nur aus den Elementen $<, =, >$. Alle anderen Symbole können durch diese Symbole dargestellt werden, indem diese mit einem logischen Oder ($\vee$) verknüpft werden. So kann z.B. $A \leq B$ als $(A < B) \vee (A = B)$ dargestellt werden oder $A \neq B$ als $(A < B) \vee (A > B)$. Wenn also ein falsches Relationssymbol verwendet wird, muss es sich in einem dieser drei „Grundsymbole“ vom eigentlichen unterscheiden. Wie aus Tabelle 1 ersichtlich ist, gibt es für jedes der Relationssymbole $<, \leq, =, \geq, >$ und $\neq$ mindestens ein anderes Symbol, welches sich nur in einem seiner drei Wahrheitswerte unterscheidet. Würde also nur mit zwei Werten getestet werden, so könnten bestimmte Vertauschungen übersehen werden. Desweiteren zeigt die Tabelle, dass wirklich mit allen drei Bedingungen $A < B$, $A = B$ und $A > B$ getestet werden muss, da es für jede dieser Bedingungen auch ein Paar von Relationssymbolen gibt, bei denen sich der Wahrheitswert nur genau für diese Bedingung unterscheidet.

Frage 6.9: Auf welche Arten können Boolesche Ausdrücke verwendet werden?
Boolesche Ausdrücke können auf zwei Arten verwendet werden:

	$A < B$	$A \leq B$	$A = B$	$A \geq B$	$A > B$	$A \neq B$
$A < B$	w	w	f	f	f	w
$A = B$	f	w	w	w	f	f
$A > B$	f	f	f	w	w	w

Tabelle 1: Wahrheitstabelle für $<, \leq, =, \geq, >, \neq$ unter den Voraussetzungen $A < B$, $A = B$ und $A > B$

- Berechnung eines Entscheidungsausgangs im Kontrollfluss.
- Berechnung des Wertes einer Booleschen Variablen.

Frage 6.10: Zu welchen Arten von Fehlern kann es bei (allgemeinen) Ausdrücken kommen?

Bei (allgemeinen) Ausdrücken können zwei Arten von Fehlern unterschieden werden.

1. Überflüssige Teilausdrücke
2. Vertauschte (Teil-)Ausdrücke

Um den ersten Fehler zu erkennen, muss dafür gesorgt werden, dass es für den Ausdruck und jeden seiner Teilausdrücke ein Testdatum gibt, bei dessen Ausführung der Ausdruck und der Teilausdruck verschiedenen Werte annehmen. Dieses Kriterium wird **kürzere Ausdrücke** genannt. Warum muss dies gemacht werden? Angenommen der Ausdruck $x*(x-2)+(x-1)$, der in Abbildung 12 graphisch dargestellt ist, sei gegeben. Er besteht also aus den beiden Teilausdrücken $x*(x-2)$ und $(x-1)$, die ebenfalls in Abbildung 12 dargestellt sind. Wenn nun z.B. der Teilausdruck $x*(x-2)$ überflüssig wäre, so würde dies für die Testdaten $x = 0$ und $x = 2$ nicht bemerkt werden, da der Teilausdruck $x*(x-2)$ für diese Werte Null wird und sich das Ergebnis des Ausdrucks $(x-1)$ nicht mehr von dem des Ausdrucks $x * (x - 2) + (x - 1)$ unterscheidet. Dies ist also ein Fall, bei dem sich das Ergebnis eines Teilausdrucks nicht von dem des ganzen Ausdruck unterscheidet. Und wann gilt dies? Im obigen Beispiel gilt dies genau dann, wenn:

$$x * (x - 2) + (x - 1) = (x - 1) \;\Leftrightarrow\; x * (x - 2) = 0 \;\Leftrightarrow\; x = 0 \vee x = 2$$

Aus Abbildung 12 ist dieses Ergebnis ebenfalls ersichtlich. An den Stellen, an denen die Funktion $x*(x-2)$ ihre Nullstellen hat, schneiden sich die Funktionen $x * (x - 2) + (x - 1)$ und $(x - 1)$. Aus diesem Grund wird gefordert, dass ein Ausdruck und alle seine Teilausdrücke verschieden voneinander sein müssen.

Um die zweite Forderung zu erfüllen, muss es für jeden Ausdruck im Programm ein Testdatum geben, das diesen Ausdruck ausführt und bei dessen Ausführung alle (Teil-)Ausdrücke im Programm verschiedene Werte haben. Dieses Kriterium wird **alle kürzeren Ausdrücke** genannt.

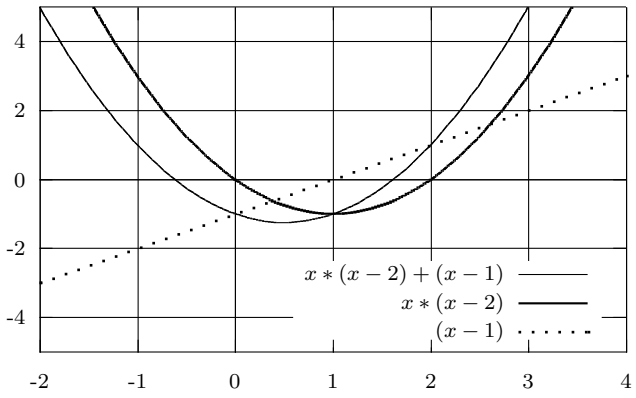


Abbildung 12: Erkennung überflüssiger Ausdrücke

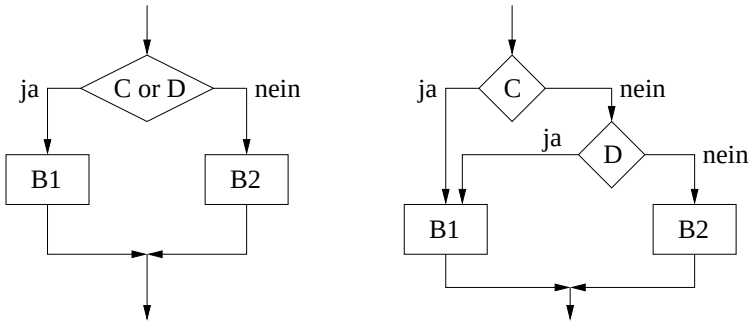


Abbildung 13: Übersetzung des Booleschen Ausdruck $C \vee D$

6.2 Test Boolescher Ausdrücke

Frage 6.11: Wie würde der Boolesche Ausdruck $(\mathbf{C} \vee \mathbf{D})$ übersetzt werden? Warum stellt sich die Frage hier an dieser Stelle?

Der Boolesche Ausdruck $C \vee D$ würde wahrscheinlich wie in Abbildung 13 übersetzt werden. Links ist der „normale“ Kontrollflussgraph gezeigt und rechts der übersetzte, verfeinerte Kontrollflussgraph. Die Variablen würden also einzeln getestet werden und sobald eine den Wahrheitswert *wahr* annimmt, wird der „wahr“-Zweig $B1$ ausgeführt. Bei einem logischen Und ($\wedge$) würde ähnlich vorgegangen werden, nur dass bei dem ersten *false* der „falsch“-Zweig ausgeführt werden würde.

Die Frage stellt sich deshalb hier an dieser Stelle, da es ein Ziel dieses Unterkapitels sein wird, ein Kriterium zu finden, welches alle Zweige des feiner modellierten Kontrollflussgraphen überdeckt.

Frage 6.12: Wie werden Boolesche Ausdrücke in einem ersten Ansatz im Buch modelliert?

Boolesche Ausdrücke werden zuerst relativ grob modelliert, d.h. sie werden nicht sehr genau betrachtet. Sie werden als eine Verknüpfung von **atomaren Prädikattermen** bzw. **atomaren Bedingungen** mit **logischen Operatoren** modelliert. Wie schon angedeutet, interessiert die genauere Struktur der

atomaren Bedingungen dabei erstmal nicht. Es wird nur angenommen, dass sie die Werte *true* und *false* annehmen können.

Ist zum Beispiel der Boolesche Ausdruck $(C \geq 1) \wedge D$ gegeben, so ist $\wedge$ der logische Operator und $(C \geq 1)$ und D sind die atomaren Prädikatterme.

Frage 6.13: Wodurch zeichnen sich atomare Prädikatterme aus?

Atomare Prädikatterme bzw. atomare Bedingungen zeichnen sich dadurch aus, dass sie keine logischen Operatoren wie z.B. $\vee, \wedge, \neg$ enthalten, sondern höchstens Relationssymbole wie z.B. $<, \leq, =>, \geq, >, \neq$. Ist nur eine Variable wie z.B. D gegeben, so muss es sich dabei um eine boolesche Variable handeln.

Frage 6.14: Wann erfüllt eine Testdatenmenge die C_2-Überdeckung? Wie wird diese Überdeckung noch genannt?

Eine Testdatenmenge T erfüllt die **C_2-Überdeckung** genau dann, wenn es für jede Verzweigung im Programm und für jeden atomaren Prädikatterm p des zur Verzweigung gehörenden Booleschen Ausdrucks zu jedem Wahrheitswert von p ein Testdatum t in T gibt, bei dessen Ausführung p diesen Wahrheitswert annimmt. Dieses Kriterium wird auch **atomare Bedingungsüberdeckung** genannt.

Oder anders gesagt, soll dafür gesorgt werden, dass jeder atomaren Prädikatterm mindestens einmal den Wert *true* und mindestens einmal den Wert *false* annimmt. Leider erfüllt dieses Kriterium nicht das zu Beginn dieses Kapitels angesprochene Ziel der Zweigüberdeckung im verfeinerten Kontrollflussgraphen. Es erfüllt noch nicht einmal das Kriterium der Zweigüberdeckung im „normalen“ Kontrollflussgraphen, wie man sich schnell an Abbildung 13 klarmachen kann. Die Testdaten $C = \mathit{true}$ und $D = \mathit{false}$ sowie $C = \mathit{false}$ und $D = \mathit{true}$ erfüllen das Kriterium der C_2-Überdeckung, jedoch wird beides mal der *true*-Zweig über $B1$ ausgeführt.

Frage 6.15: Wann erfüllt eine Testdatenmenge das Kriterium der „Zweig-/Bedingungsüberdeckung“?

Eine Testdatenmenge T erfüllt die **Zweig-/Bedingungsüberdeckung** genau dann, wenn sie die C_2-Überdeckung (atomare Bedingungsüberdeckung) und die C_1-Überdeckung (die Zweigüberdeckung) erfüllt.

Oder anders gesagt, soll dafür gesorgt werden, dass der gesamte Boolesche Ausdruck der Bedingung einmal den Wert *true* und einmal den Wert *false* annimmt. Dies ist für das Beispiel in Abbildung 13 für die Testdaten $C = \mathit{true}$ und $D = \mathit{true}$ sowie $C = \mathit{false}$ und $D = \mathit{false}$ gegeben, da in diesen beiden Fällen einmal der *true*- und einmal der *false*-Zweig ausgeführt wird und die C_2-Überdeckung ebenfalls gegeben ist. Jedoch wird nicht jeder Zweig des verfeinerten Kontrollflussgraphen überdeckt, da z.B. der *true*-Zweig der Bedingung D nicht abgelaufen wird.

Frage 6.16: Wann erfüllt eine Testdatenmenge die „minimale Mehrfachbedingungsüberdeckung“?

Eine Testdatenmenge erfüllt die **minimale Mehrfachbedingungsüberdeckung** genau dann, wenn für jede Verzweigung im Programm mit zwei Ausgängen und

A	B	$A \wedge B$	$A = B$	$A \vee B$	$\neg(A \wedge B)$
f	f	f	w	f	w
f	w	f	f	w	w
w	f	f	f	w	w
w	w	w	w	w	f

Tabelle 2: Maskierung von Fehlern

für den zur Verzweigung gehörenden Booleschen Ausdruck folgende Kombinationen der Wahrheitswerte der atomaren Prädikatterme, die mit ein- oder mehrstelligen logischen Operatoren verknüpft sind, ausgeführt werden: jede mögliche Kombination von Wahrheitswerten, bei denen die Änderung des Wahrheitswerts eines Terms den Wahrheitswert der logischen Verknüpfung ändern kann. Diese Art der Überdeckung wird auch **C_2(mM)-Überdeckung** genannt.

Dahinter steckt die Idee der **Maskierung** von Fehlern. Es wird davon ausgegangen, dass sich nur der Wahrheitswert eines atomarer Prädikatterm ändert. Es gibt nun aber Belegungen bei denen die Änderung einer atomaren Bedingung garnichts ändert, d.h. ein solcher Fehler würde nicht bemerkt werden. Tabelle 2 zeigt einige Beispiele dazu. Beim logischen „und“ würde z.B. die Belegung *ff* garnicht erst getestet werden, da sowohl *fw* als auch *wf* den selben logischen Wert ergeben.

Frage 6.17: Wann erfüllt eine Testdatenmenge die „Mehrfachbedingungsüberdeckung“?

Eine Testdatenmenge erfüllt die **Mehrfachbedingungsüberdeckung** genau dann, wenn für jede Verzweigung im Programm mit zwei Ausgängen und den zur Verzweigung gehörenden Booleschen Ausdruck jede mögliche Kombination der Wahrheitswerte der atomaren Prädikatterme ausgeführt wird.

Für die Bedingung $(C \vee D)$ müssten also die Kombinationen ff, ft, tf und tt getestet werden. Wie man sich leicht überlegen kann, beträgt die Anzahl der Testdaten für eine Bedingung mit n verschiedenen atomaren Prädikattermen $O(2^n)$.

6.3 Mutationsanalyse

Frage 6.18: Was ist das Ziel der Mutationsanalyse?

Das Ziel der Mutationsanalyse ist es, festzustellen, ob eine Testdatenmenge T bestimmte Fehler in einem Programm **garantiert** entdeckt. Ist dies nicht der Fall, können weitere Testdaten erzeugt werden, die die Fehlerfindung verbessern.

Frage 6.19: Wie wird bei der Mutationsanalyse vorgegangen?

Zuerst wird zu einem Programm P eine Menge von Programmen $P_1, \ldots, P_n$ erstellt, die sich von P nur „in einer Kleinigkeit“ unterscheiden. Diese Programme P_i werden, in Analogie zur Biologie, **Mutanten** von P genannt.

Auf jede Mutante $P_i,\ i = 1, \ldots, n$ werden nun so lange Tests t aus der Testdatenmenge T angewandt, bis sich das Ergebnis $P_i(t)$ der Mutante vom

Ergebnis $P(t)$ des ursprünglichen Programm unterscheidet. Wird ein solches Testdatum gefunden, wird die Mutante als „tot“ bezeichnet. Ansonsten ist sie bezüglich der Testdatenmenge nicht „tot“ zu kriegen. Oder etwas formaler:

1. Eine Mutante P_i ist **lebendig** bezüglich T und P genau dann, wenn für alle Tests t aus T gilt: $P_i(t) = P(t)$.
2. Eine Mutante P_i ist **tot** bezüglich T und P genau dann, wenn für mindestens einen Test t aus T gilt: $P_i(t) \neq P(t)$.

Frage 6.20: Wie ist ein Mutationsanalyse-Ergebnis zu bewerten, bei dem alle Mutanten tot sind?
Eine solches Ergebnis würde die Zuversichtlichkeit bezüglich der Testdatenmenge T erhöhen die „entsprechenden“ Fehler im Programm zu finden. Allerdings hängt dies auch von der Anzahl der Mutanten bzw. ihrer Eigenschaften ab:

Äquivalenz: Eine Mutante P_i kann zu einem Programm P **äquivalent** sein, d.h. es gilt für alle Eingabedaten t stets $P_i(t) = P(t)$. Wird in einem Programm die Anweisung `if(x>0)` z.B. zu `if(3*x>0)` geändert, wird sich das Programm immer noch genau gleich verhalten, es ist also äquivalent.

Anzahl: Es müssten genügend Mutanten erzeugt werden sein, die tatsächlich alle „typischen“ Fehler enthalten.

Kopplungseffekt: Es müsste der Kopplungseffekt gelten, das heißt, dass Testdaten, die einfache Fehler finden auch kompliziertere Fehler (die z.B. durch mehrere Mutationen entstehen) entdecken.

All diese Annahmen sind jedoch zweifelhaft, weshalb die Mutationsanalyse auch ihre Grenzen hat.

6.4 Datenbezogenes Testen

Frage 6.21: Was wird beim datenbezogenen Testen betrachtet?
Bisher wurden die Anweisungen bzw. der Kontroll- und Datenfluss in Programmen betrachtet. Beim datenbezogenen Testen werden nun die Daten selbst und die verwendeten Datenstrukturen näher betrachtet und bezüglich dieser Anforderungen gestellt, welche Testdaten für bestimmte Überdeckungen erfüllen müssen.

Frage 6.22: Welche Abstraktionsebenen von Daten kennst du?
Daten können auf verschiedenen Abstraktionsebenen betrachtet werden. Dies sind:

Datenkapsel: Dies ist die gröbste Einheit. Unter einer Datenkapsel versteht man eine **Menge von zusammenhängenden Daten, die in einer Zugriffseinheit abgespeichert sind**. Dies kann z.B. die Instanz einer Klasse oder eines `structs` sein.

Datenfelder: Eine Datenkapsel besteht aus verschiedenen Datenfeldern (z.B. Arrays, Listen, Variablen, etc.).

Repräsentativer Werte: Jedes Datenfeld kann eine bestimmte Menge repräsentativer Werte annehmen. Für eine Variable, die einen Enum speichert, können dies z.B. alle Belegungen sein. Oder eine Liste kann leer sein oder gefüllt.

Frage 6.23: Welche Arten der Datenüberdeckungen kennst du?
Die Datenüberdeckungen sind alle mit den verschiedenen, oben angesprochenen Abstraktionsebenen gekoppelt:

Datenkapselüberdeckung: Jede Datenkapsel wird mindestens einmal „angefasst“.

Datenfeldüberdeckung: Jedes Eingabe- und Ausgabefeld einer Datenkapsel muss mindestens einmal „angefasst“ werden.

Überdeckung repräsentativer Werte: Jedes Eingabe- und Ausgabefeld einer Datenkapsel muss mindestens einmal alle repräsentativen Werte enthalten.

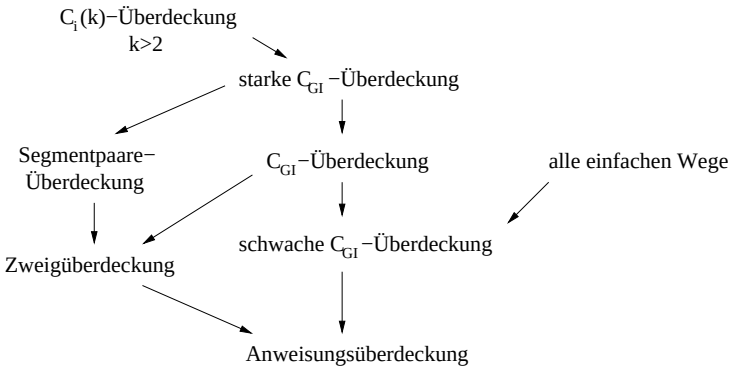


Abbildung 14: Partielle Ordnung einiger kontrollflussbezogener Testkriterien

7 Bewertung der implementationsorientierten Testkriterien

Frage 7.1: Warum ist ein Vergleich der Testkriterien notwendig?
In den Kapiteln 4 bis 6 wurde eine Fülle von Testkriterien angegeben. Diese sind jedoch **unterschiedlich aufwändig** und können, ebenfalls aus Gründen des Aufwands, auch **nicht alle angewandt werden**. Daher ist es nötig, die Kriterien untereinander zu vergleichen. Dabei sind folgende Arten von Vergleichen möglich:

- Welches Kriterium enthält ein anderes Kriterium?
- Wieviele Testdaten sind nötig, um ein Testkriterium zu erfüllen?
- Deckt ein Kriterium, welches ein anderes Kriterium enthält, mehr Fehler auf?

7.1 Enthaltensein und Unvergleichbarkeit von Testkriterien

Frage 7.2: Gebe die partielle Ordnung der kontrollflussbezogenen Testkriterien an!
Abbildung 14 zeigt die partielle Ordnung einiger kontrollflussbezogener Testkriterien. Dass die C_i(k)-Überdeckung (für $k > 2$) die starke C_{GI}-Überdeckung enthält, dürfte schon aufgrund der Definition klar sein, da die starke C_{GI}-Überdeckung der $C_i(2)$-Überdeckung entspricht.

Frage 7.3: Gebe die partielle Ordnung der datenflussbezogenen Testkriterien an!
Die partielle Ordnung einiger datenflussbezogener Testkriterien ist in Abbildung 15 dargestellt.

Frage 7.4: Gebe die partielle Ordnung der Bedingungskriterien an!
Die partielle Ordnung der Bedingungskriterien ist in Abbildung 16 darge-

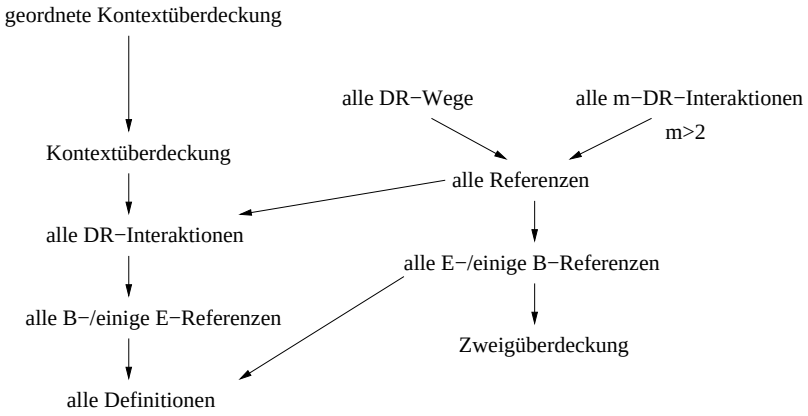


Abbildung 15: Partielle Ordnung einiger datenflussbezogener Testkriterien

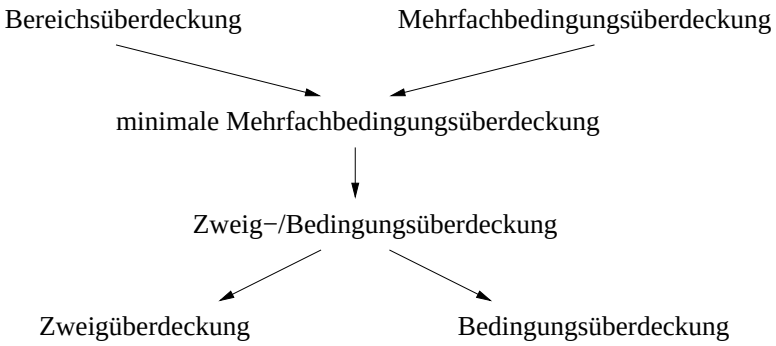


Abbildung 16: Partielle Ordnung der Bedingungskriterien

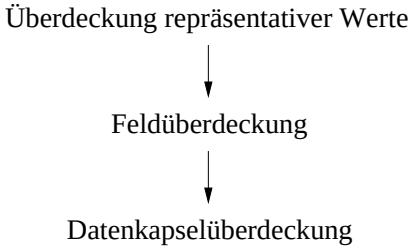


Abbildung 17: Ordnung der datenbezogenen Testkriterien

stellt. Diese Ordnung entspricht, von unten nach oben gesehen, der Reihenfolge, in der diese Kriterien vorgestellt und erweitert wurden. Da die atomare Bedingungsüberdeckung keine Zweigüberdeckung garantiert, wurde dies bei der Zweig-/Bedingungsüberdeckung explizit gefordert. Dass die Mehrfachbedingungsüberdeckung, bei der einfach alle Kombinationen der Wahrheitsbelegungen der atomaren Prädikatterme getestet werden, die minimale Mehrfachbedingungsüberdeckung enthält, dürfte ebenfalls recht schnell klar sein. Denn die minimale Mehrfachbedingungsüberdeckung testet ja eine bestimmte Teilmenge aller Kombinationen. Ebenso enthält die Bereichsüberdeckung die minimale Mehrfachbedingungsüberdeckung, da auch bei diesem Verfahren keine Kombinationen getestet werden, bei denen ein Fehler maskiert wird.

Frage 7.5: Gebe die Ordnung der datenbezogenen Testkriterien an!
Die Ordnung der datenbezogenen Testkriterien ist in Abbildung 17 dargestellt. Bei der Datenkapselüberdeckung sollen die Tastdaten so aussehen, dass jede Datenkapsel nur einmal „angefasst“ wird, d.h. dass eines der Datenfelder gelesen oder geschrieben wird. Besteht die Datenkapsel aus mehreren Datenfeldern, so gewährleistet das „Anfassen“ eines Feldes also die Datenkapselüberdeckung, nicht jedoch die Feldüberdeckung, da bei dieser Art der Überdeckung, jedes Feld einmal angefasst werden muss. Wird jedes Feld angefasst, heißt dies jedoch noch lange nicht, dass jeder repräsentative Wert einmal benutzt wird, so dass eine Feldüberdeckung also keine Überdeckung repräsentativer Werte garantiert. Das umgekehrte Enthaltensein dürfte nach diesen Betrachtungen recht schnell klar werden.

7.2 Anzahl der Testdaten pro Testkriterium

Frage 7.6: Gebe die Anzahl der Testdaten für einige kontroll- und datenflussbezogene Testkriterien an!
Zuerst einmal müssen einige Annahmen über das Aussehen des Kontroll- bzw. Datenflussgraphen gemacht werden. Dieser bestehe aus n Segmenten, wobei die Anzahl der ausgehenden Kanten und die Anzahl der Variablen pro Segment durch eine Konstante begrenzt seien. Dann gilt:

1. Die **Pfadüberdeckung** erfordert **unendlich viele Testdaten**.

   Um dies zu zeigen, muss nur ein Programm angegeben werden, welches unendlich viele Testdaten benötigt bzw. in welchem es unendlich viele

Wege gibt. Es sei ein Programm gegeben, welches für ein n die Summe $\sum_{i=1}^{n} i$ berechne (Algorithmus 1). Für verschiedene n werden verschiedene vollständige Wege durchlaufen. Also sind unendlich viele Testdaten nötig.

Algorithmus 1 Programm zur Berechnung von $\sum_{i=1}^{n} i$

```
x = 0;
while n > 0 do
  x = x + n;
  n = n - 1;
end while
```

2. Die **strukturierte Pfadüberdeckung**, die [**starke/schwache**] $\mathbf{C_{GI}}$-**Überdeckung** sowie das Kriterium **alle DR-Wege** erfordern $\mathbf{O(2^n)}$ Testdaten.

   Auch hier kann relativ schnell ein Fall angegeben werden, bei dem exakt 2^n Testdaten benötigt werden. Es sei ein Programm gegeben, welches aus einer Sequenz von n `if - then - else`-Anweisungen bestehe. Desweiteren werde eine Variable x zu Beginn des Programms definiert, welche in den beiden Zweigen der letzten `if - then - else`-Anweisung referenziert wird. Das Programm sieht also ähnlich aus, wie das Programm in Abbildung 5 auf Seite 34. Dann benötigen alle oben genannten Kriterien 2^n Testdaten, da jeweils jeder Weg durch die `if - then - else`-Anweisungen durchlaufen werden muss.

3. Die Datenflusskriterien **alle E-/einige B-Referenzen**, **alle B-/einige E-Referenzen**, **alle Referenzen** und die **Kontextüberdeckung** erfordern $\mathbf{O(n^2)}$ Testdaten.

   Die Grundidee bei diesem Beweis ist, dass zwischen allen Variablen Paare von Definitionen und Referenzen gebildet werden müssen. Es gibt n Segmente und die Anzahl der Variablen pro Segment ist durch eine Konstante k begrenzt. Es gibt also $O(k \cdot n)$ Variablen, wobei k die größte dieser Konstanten sei. Da k jedoch eine Konstante ist, gilt $O(k \cdot n) = O(n)$. Es gibt also $O(n)$ Variablen im Programm. Es stellt sich nun also die Frage, wieviele Paare zwischen diesen gebildet werden können bzw. wieviele Möglichkeiten es gibt, zwei Variablen aus n auszuwählen. Dies sind:

$$\binom{n}{2} = \frac{n!}{2! \cdot (n-2)!} = \frac{n \cdot (n-1)}{2} = \frac{1}{2} \cdot (n^2 - n) = O(n^2)$$

4. Das Kriterium **alle Definitionen** sowie die **Zweig-** und die **Anweisungs-Überdeckung** erfordern $\mathbf{O(n)}$ Testdaten.

   Für die genannten Kriterien reicht die Ausführung aller Segmente aus. Das schlimmste, was in diesem Fall passieren kann, ist ein Programm P mit verschachtelten `if - then - else`-Anweisungen, so wie es in Abbildung 18 skizziert ist. Da das Programm sich in diesem Fall immer

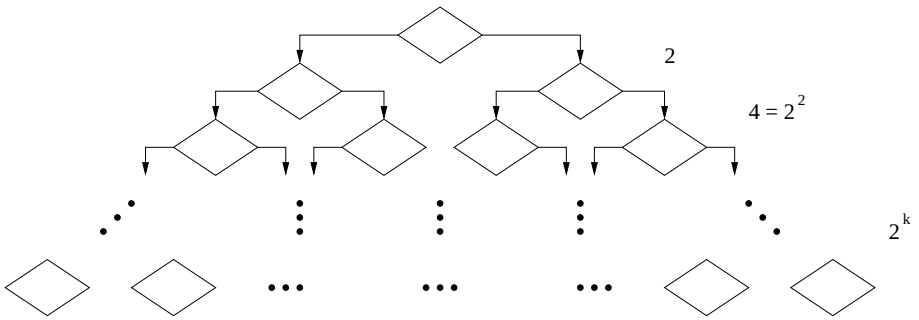


Abbildung 18: Worst-case für den Fall 4

mehr verzweigt, könnte man auf die Idee kommen, dass exponentiell viele Wege nötig sind. Dies ist jedoch nicht der Fall. Abbildung 18 zeigt ein Programm, bei dem die Schachtelungstiefe der `if - then - else`-Anweisungen k beträgt. Das heißt, auf der letzen Ebene befinden sich 2^k Knoten, welche über 2^k Wege erreichbar sind. Doch in welcher Relation stehen diese 2^k Wege zu den n Segmenten? Es gibt n Segmente, welche sich wie folgt zusammensetzen:

$$n = 2+4+\cdots+2^k = 2^1+2^2+\cdots+2^k = \sum_{i=0}^{k} 2^i - 1 = 2^{k+1}-1-1 = 2^{k+1}-2$$

Die Formel $n = 2^{k+1} - 2$ wird nun nach 2^k umgeformt:

$$\begin{aligned} 2^{k+1} - 2 &= n \\ 2^{k+1} &= n + 2 \\ 2 \cdot 2^k &= n + 2 \\ 2^k &= \frac{n+2}{2} \in O(n) \end{aligned}$$

Also müssen auch bei einer solchen „schlimmen“ Schachtelung nur linear viele Wege abgelaufen werden.

Frage 7.7: Gebe die Anzahl der Testdaten für einige Kriterien des Bedingungstesten an!

Im folgenden wird nun der Aufwand der verschiedenen Kriterien für das Testen einer einzelnen Entscheidung miteinander verglichen. Dabei wird davon ausgegangen, dass die Entscheidung aus n voneinander unabhängigen Prädikattermen besteht. Tabelle 3 vergleicht die verschiedenen Kriterien diesbezüglich.

Bedingungsüberdeckung: Es ist ziemlich schnell einzusehen, warum bei der Bedingungsüberdeckung nur zwei Testdaten benötigt werden. Zuerst wird irgendein Testdatum gewählt. Die einzelnen Prädikatterme werden jeweils zu *true* oder *false* ausgewertet. Das zweite Testdatum wird nun so gewählt, dass jeder Prädikatterm genau anders als beim ersten Mal ausgewertet wird.

Kriterium	Zahl der erforderlichen Tests pro Entscheidung mit n Termen
Bedingungsüberdeckung	2
C_2(mM)-Überdeckung	$\leq n + 1$
Bereichsüberdeckung mit linearen Grenzen in v Variablen	$\leq n \cdot (v + 3)$
Mehrfachbedingungsüberdeckung	$\leq 2^n$

Tabelle 3: Anzahl der Testdaten beim Bedingungstesten

C_2(mM)-Überdeckung: Bei der Abschätzung der minimalen Mehrfachbedingungsüberdeckung wird davon ausgegangen, dass das Entscheidungsprädikat nur den Operator „$\vee$“ *(or)* oder „$\wedge$“ *(and)* enthält. In letzterem Fall würde so getestet werden, dass zuerst alle atomaren Prädikatterme den Wert *true* erhalten. Anschliessend würden die Testdaten so gewählt werden, dass alle Prädikatterme bis auf einen den Wert *true* erhalten. Dies wird für jeden Prädikatterm gemacht. Da es n solcher Terme gibt, ergeben sich also mit dem ersten Test höchstens $n + 1$ Tests.

Bereichsüberdeckung: Kommt noch!

Mehrfachbedingungsüberdeckung: Bei der Mehrfachbedingungsüberdeckung müssen im schlimmsten Fall alle Kombinationen von Wahrheitsbelegungen der Prädikatterme getestet werden, also bei n Prädikattermen 2^n Belegungen. Es gibt jedoch auch Bedingungen, bei denen es garnicht möglich ist, alle Wahrheitskombinationen zu erzeugen.

Es sei zum Beispiel ein Programm gegeben, in dem getestet werden soll, ob ein Punkt (x, y, z) in einem achsenparallelen Quader liegt. Der Quader sei durch seine Grenzen (x_1, x_2), (y_1, y_2) und (z_1, z_2) gegeben. Damit ein Punkt innerhalb dieses Quaders liegt, muss gelten:

$$(x_1 \leq x) \ \wedge \ (x \leq x_2) \ \wedge \ (y_1 \leq y) \ \wedge \ (y \leq y_2) \ \wedge (z_1 \leq z) \ \wedge \ (z \leq z_2)$$

Wenn der Quader nicht gerade zu einem Punkt im $\mathbb{R}^3$ verkümmert, also $x_1 = x_2 = y_1 = y_2 = z_1 = z_2$ gilt, ist es nicht möglich einige Kombinationen von Wahrheitswerten zu erzeugen. Wenn für den Quader zum Beispiel $x_1 < x_2$ gilt, dann kann nicht gleichzeitig $(x_1 > x)$ (also $(x_1 \leq x)$ ist *false*) und $(x > x_2)$ (also $(x \leq x_2)$ ist *false*) gelten.

7.3 Aufgedeckte Fehler pro Testkriterium

Frage 7.8: Welche Fehler können beim ausdrucks- und anweisungsbezogenen Testen aufgedeckt werden?

Für das Aufdecken von Fehlern in Entscheidungsprädikaten gibt es einige theoretische Erkenntnisse. Das Aussehen der Entscheidungsprädikate muss dabei jedoch etwas eingeschränkt werden. Die atomaren Bedingungen dürfen nur mit

den Booleschen Operatoren $\vee, \wedge$ und $\neg$ kombiniert werden. Eine **atomare Bedingung** ist dabei entweder eine Boolesche Variable oder ein relationaler Ausdruck der Form $A\ r\ B$, wobei r einer der Relationsoperatoren $<, \leq, =, \neq, \geq$ oder $>$ ist. Ein Entscheidungsprädikat ohne relationale Ausdrücke heißt **Boolescher Ausdruck**. In diesem Fall kommen also nur Boolesche Variablen in dem Prädikat vor. Desweiteren seien die folgenden Voraussetzungen auch noch gegeben:

1. In den relationalen Ausdrücken der Entscheidungsprädikate kommt jede Variable insgesamt nur einmal vor.
2. Boolesche Ausdrücke enthalten jede Variable nur einmal.

Dann gelten die folgenden Aussagen:

1. Mit der minimalen Mehrfachbedingungsüberdeckung werden alle fehlerhaften Booleschen Operatoren von Booleschen Ausdrücken aufgedeckt.
2. Mit der BRO-Überdeckung (Boolesche und Relationale Operatoren) werden alle fehlerhaften Booleschen und relationalen Operatoren in Entscheidungsprädikaten gefunden.

Frage 7.9: Was ist ein effektiver Test für ein Polynom? Wann ist eine Testdatenmenge ein effektiver Test für ein Polynom?

Ein **effektiver Test** für ein Polynom ist ein Test, der Fehler erkennt, bei denen der Grad des (nach Spezifikation) korrekten Polyoms nicht erniedrigt wird. Dies bedeutet, dass der Grad des implementierten und eventuell fehlerhaften Polynoms entweder größer oder gleich dem korrekten Polynoms ist. Eine Testdatenmenge T ist ein effektiver Test für ein Polynom P vom Grade k genau dann, wenn T mindestens $k+1$ (verschiedene) Testdaten enthält.

Polynome lassen sich aufgrund ihrer **algebraischen Eigenschaften** effektiv testen. Der Grund dafür liegt darin, dass ein Polynom vom Grad k durch $k+1$ unabhängige Werte eindeutig festgelegt ist. Daraus folgt, dass zwei verschiedene Polynome vom Grad k also höchstens an k Stellen übereinstimmen können. Wenn also mit $k+1$ unabhängigen Testdaten getestet wird, so muss bei voneinander verschiedenen Polynomen also mindestens ein Ergebnis unterschiedlich sein.

Diese Eigenschaft von Polynomen ist auch der Grund, warum sich der Grad des fehlerhaften Polynoms nicht erniedrigen darf. In diesem Fall kann es nämlich vorkommen, dass das fehlerhafte und das korrekte Polynom für die Testdaten zufällig übereinstimmen. Dies ist der Fall, wenn Punkte genommen werden, an denen das falsche Polynom niedrigeren Grades das richtige Polynom höheren Grades schneidet. Abbildung 19 zeigt ein Beispiel für diesen Fall. Das korrekte Polynom wäre eigentlich $1.5 \cdot x^3 - 7.5 \cdot x$. Implementiert wurde jedoch irrtümlich $x^2 - 3$, also ein Polynom niedrigeren Grades, was bei einem effektiven Test **nicht** passieren darf. Denn wie in der Abbildung zu erkennen ist, würden drei Testdaten (da $x^2 - 3$ vom Grad 2 ist) nicht ausreichen, um das fehlerhafte Polynom von dem korrekten Polynom zu unterscheiden, da es drei Schnittpunkte gibt. Würde mit mindestens vier Testdaten getestet werden, so fiele der Fehler auf.

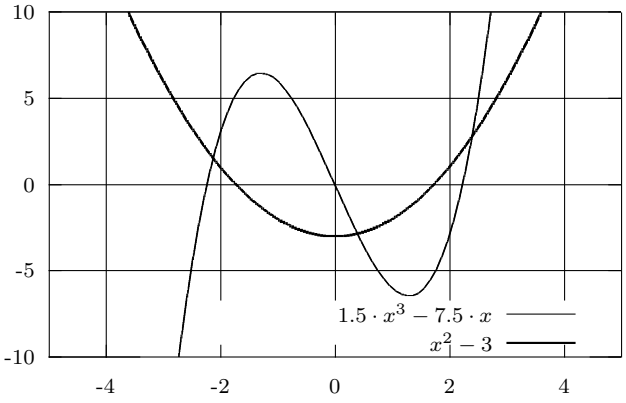


Abbildung 19: Fehlerhaftes Polynom mit erniedrigtem Grad

8 Testdatenerzeugung und Testwirksamkeitsmessung

Frage 8.1: Wie lassen sich Testwirksamkeitsmaße für die bisher vorgestellten Kriterien bestimmen?

Im Prinzip wird so vorgegangen, wie bei den Kriterien für die Anweisungs- und die Zweigüberdeckung. Es wird für das betrachtete Konstrukt der Quotient zwischen der Anzahl der durch die Testdaten ausgeführten bzw. überdeckten Konstrukte und der Anzahl aller Konstrukte gebildet, also:

$$TWM = \frac{\text{Anzahl der durch die Testdaten überdeckten Konstrukte}}{\text{Anzahl aller Konstrukte}}$$

Wie sich die Anzahl der ausgeführten Konstrukte bestimmen lässt, wird in Kapitel 8.1 erläutert. Wenn das Testwirksamkeitsmaß nicht 100% bzw. die angepeilte Wirksamkeit beträgt, ist es nötig die weitere Testdaten zu bestimmen, welche die Wirksamkeit erhöhen. Dies ist Thema von Kapitel 8.2.

8.1 Messung der Testwirksamkeit durch Instrumentierung

Frage 8.2: Was ist „Instrumentierung“?

Unter **Instrumentierung** versteht man die Veränderung eines Programms zur Feststellung, welche Konstrukte ausgeführt wurden. Dabei werden in die entsprechenden Konstrukte Messpunkte eingebaut, welche die Ausführung des Konstrukts protokollieren. Dies kann zum Beispiel ein Zähler sein, der erhöht wird, oder ein entsprechender Prozeduraufruf.

Frage 8.3: Wie kann die Zweigüberdeckung durch Instrumentierung festgestellt werden?

Um eine Zweigüberdeckung durch Instrumentierung festzustellen, reicht es aus, in jeden Entscheidungs-Entscheidungs-Weg einen Messpunkt einzubauen. Soll eine hundertprozentige C_1-Überdeckung festgestellt werden, können dabei Messpunkte eingespart werden, da es bei verschachtelten Schleifen ausreicht, nur in

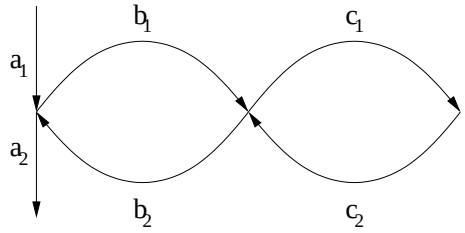


Abbildung 20: Mehrfach geschachtelte Schleifen

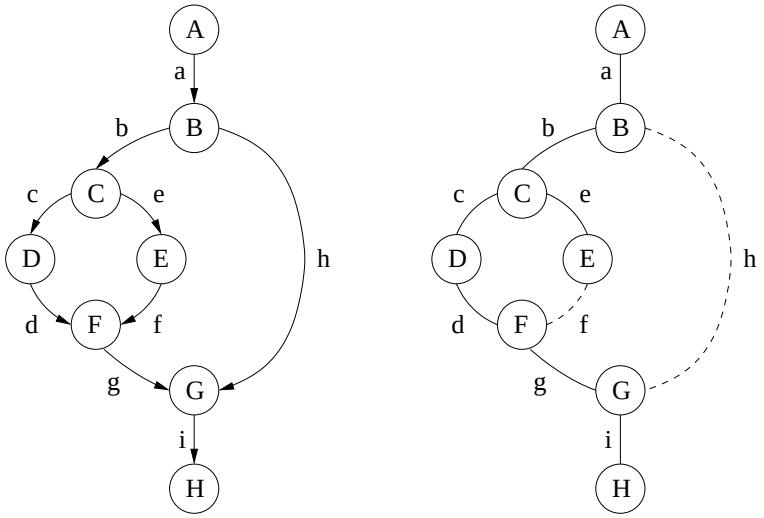


Abbildung 21: Einsparen von Messpunkten

die innerste Schleife einen Messpunkt einzubauen. Abbildung 20 zeigt ein Beispiel für einen solchen Fall. Es soll die hundertprozentige Überdeckung nachgewiesen werden, also muss auch die Schleife c_1c_2 mindestens einmal ausgeführt werden. Damit dies passiert, müssen jedoch auch die Wege a_1, b_1, b_2 und a_2 ausgeführt werden.

Frage 8.4: Inwiefern lassen sich Messpunkte bei der Instrumentierung einsparen?

Etwas weiter oben wurde schon gezeigt, dass sich Messpunkte einsparen lassen, wenn eine hundertprozentige Überdeckung festgestellt werden soll. Allerdings lassen sich auch in anderen Fällen Messpunkte einsparen.

Es sei ein **wohlgeformter Kontrollflussgraph** gegeben, der aus n Knoten und e Kanten bestehe. Desweiteren sei die **Zahl t der Testdaten bzw. Testläufe bekannt** und bei **jedem Testlauf terminiere das Programm**. Dann ist $e - n + 1$ die notwendige und hinreichende Zahl von Messpunkten, um die Ausführungszahlen für alle Anweisungen und Zweige des Programms zu bestimmen.

Das Prinzip, welches dabei verwendet wird, kann anhand von Abbildung 21 schnell gezeigt werden. Zuerst soll das Aussehen einer Entscheidung näher betrachtet werden. In diesem Fall wird einfach davon ausgegangen, dass sich

eine Entscheidung an einem Knoten (z.B. Knoten C in Abbildung 21) aufteilt und sich die gesplitteten Wege in einem anderen Knoten (z.B. F) wieder zusammenfinden. Dabei entsteht ein Kreis, wenn der Graph als ungerichtet betrachtet wird. Wenn nun die Anzahl der Testdaten, die am Eingangsknoten der Entscheidung ankommen, bekannt ist, reicht es aus, nur eine Seite der Entscheidung zu instrumentalisieren, da sich die Ausführungszahl der anderen Seite aus der Ausführungszahl der instrumentalisierten Seite berechnen lässt.

In Abbildung 21 wurde z.B. nur die Kante f instrumentalisiert (dargestellt durch die gestrichelte Linie auf der rechten Seite), da sich die Durchlaufzahlen von c und d ergeben als $z(c) = z(d) = z(b) - z(f)$. Doch wie ergibt sich $z(b)$? $z(b)$ ergibt sich wiederum als $z(b) = z(a) - z(h) = t - z(h)$.

Es muss also für jeden Kreis im als ungerichtet betrachteten Kontrollflussgraphen eine Kante ausgewählt werden, die instrumentalisiert wird. Doch wie kommt man an diese Kanten? Zuerst wird ein Spannbaum des Graphen errechnet. Lauf Definition ist ein Spannbaum ein kreisfreier, zusammenhängender Graph, in dem alle Knoten des Graphen vorkommen. Ist ein solcher Spannbaum für einen Graphen berechnet, so wird also die Hinzunahmen einer jeden Kante des ursprünglichen Graphen, die im Spannbaum *nicht* vorkommt, einen Kreis schließen. Also sind genau diese Kanten, welche auch **charakteristische Kanten** genannt werden, die gesuchten Kanten. Wieviele charakteristische Kanten gibt es? Der Spannbaum eines Graphen mit n Knoten besitzt genau $n - 1$ Kanten. Also gibt es $e - (n - 1) = e - n + 1$ charakteristische Kanten.

Wie oben schon angedeutet lassen sich mit diesen Kanten die Durchlaufzahlen aller Kanten errechnen. Dies ergibt sich aus der **Kirchhoff'schen Regel**, die besagt, dass die Summe der Ausführungszahlen der eingehenden Kanten gleich der Summe der Ausführungszahlen der ausgehenden Kanten ist.

Frage 8.5: Wie lässt sich die Bedingungsüberdeckung messen? Welche Probleme kann es dabei geben?

Bei der Bedingungsüberdeckung reicht es nicht mehr aus, nur die Zweige zu instrumentalisieren (zumal die Bedingungsüberdeckung eh keine Zweigüberdeckung garantiert). Vielmehr muss bei der Auswertung einer Bedingung protokolliert werden, welche Wahrheitswerte die einzelnen atomaren Bedingungen annehmen. Dies kann z.B. wie folgt geschehen. Gegeben sei die Anweisung

```
if( (A == 2) or (X > 1) ) then ...
```

Diese wird (wahrscheinlich durch den Compiler) ersetzt durch:

```
if( A == 2 ) then Merke( B1, true ) else Merke( B1, false );
if( X > 1 ) then Merke( B2, true ) else Merke( B2, false );
if( (A == 2) or (X > 1) ) then ...
```

Zu Problemen kann es jedoch kommen, wenn die Auswertung der Bedingungen **Seiteneffekte** hat. So ist es zum Beispiel in C/C++ die folgenden `if`-Abfrage möglich: `if( a++ == 2 ) then ...`. Würde wie oben beschrieben vorgegangen werden, würde die Variable `a` zweimal inkrementiert werden. Man könnte jedoch auch wie folgt vorgehen, wobei wieder die Abfrage von oben als Beispiel dienen soll:

```
if( INS( A == 2, 1 ) or INS( Y > 1, 2 ) ) then ...
```

Dabei ist `INS()` eine Funktion, die den logischen Wert des ersten Parameters zurück gibt, ihn jedoch zuvor in ein Array an die Position, die durch den zweiten Parameter gegeben ist, schreibt. Doch auch diese Technik versagt, wenn durch den Übersetzer eine **optimierte Entscheidungsauswertung** stattfindet. Dabei bricht die Auswertung bei einer *or*-Verknüpfungen nach der ersten wahren Bedingung ab und bei einer *and*-Verknüpfung nach der ersten falschen Bedingung.

Die Bedingungsüberdeckung kann also in den beiden folgenden Fällen gemessen werden:

- Es liegen keine Seiteneffekte vor. Es ist egal, ob die Entscheidungsauswertung optimiert durchgeführt wird, da alle atomaren Bedingungen in einem Extrablock vor der eigentlichen Bedingung protokolliert werden. In diesem Fall wird die ersten Methode benutzt.
- Es liegen Seiteneffekte vor. Dann kann die Bedingungsüberdeckung nur gemessen werden, wenn eine vollständige Entscheidungsauswertung stattfindet. Dies kann oft am Compiler eingestellt werden.

Frage 8.6: Wie lassen sich die normale und die minimale Mehrfachbedingungsüberdeckung messen?

Im Prinzip gilt für diese beiden Kriterien das oben genannte. Zur Protokollierung muss festgehalten werden, welche Kombination von Wahrheitswerten ausgeführt wurde. Jedoch ergibt sich auch hier ein Sonderfall, wenn Seiteneffekte vorliegen und eine optimierte Entscheidungsauswertung erfolgt. In diesem Fall kann nur die folgende **Variante der (minimalen) Mehrfachbedingungsüberdeckung** gemessen werden. Es sei eine n-stellige *and*-Verknüpfung mit den (atomaren) Bedingungen $B_1, \ldots, B_n$ gegeben. Dann können folgende Fälle protokolliert werden:

1. Alle (atomaren) Bedingungen B_1 bis B_n sind *true*.
2. Für jedes i mit $1 \leq i \leq n$:
   a) alle Bedingungen B_1 bis B_{i-1} sind *true*
   b) Bedingung B_i ist *false*
   c) Bedingungen B_{i+1} bis B_n werden nicht ausgewertet.

Es kann also immer nur so lange getestet werden, bis optimiert abgebrochen wird. Entsprechendes gilt für eine n-stellige *or*-Verknüpfung, nur sind hier *true* und *false* zu vertauschen.

8.2 Testdatenerzeugung

Frage 8.7: Wann müssen Testdaten erzeugt werden?
Testdaten müssen erzeugt werden, wenn der Grad der angestrebten Überdeckung noch nicht erreicht ist.

Frage 8.8: Welchen Lösungsansatz zur Testdatenerzeugung kennst du?
Der Lösungsansatz besteht darin, das Problem **iterativ** zu lösen. Dabei sieht die Problemstellung für einen Schritt wie folgt aus:

1. Gegeben ist das Programm und eine Testdatenmenge T, die keine hundertprozentige Überdeckung erreicht. Damit ist implizit die Menge der überdeckten und nicht überdeckten Konstrukte gegeben.

2. Ein nicht überdecktes Konstrukt wird gewählt und ein Testdatum bestimmt, welches dieses Konstrukt ausführt. Im Prinzip gibt es bei der Auswahl des Konstrukts wieder zwei Möglichkeiten:

   a) Das Konstrukt wird **zufällig** gewählt.

   b) Das Konstrukt wird **deterministisch** gewählt. Im Falle der Zweigüberdeckung ist z.B. folgendes Vorgehen möglich. Unter allen Wegen, die von den bisherigen Testdaten ausgeführt werden, ist das kürzeste Anfangsstück bis zu einer Verzweigung zu bestimmen, für die ein ausgehender Entscheidungsweg noch nicht überdeckt ist. Bei diesem Vorgehen ist es relativ wahrscheinlich, dass auf dem so ausgewählten neuen Weg noch viele andere Zweige liegen, die ebenfalls noch nicht überdeckt wurden. Diese Strategie wird auch **Pfadpräfix-Strategie** genannt.

Das eigentliche Problem besteht jedoch darin, ein Testdatum zu finden, welches den gewählten Weg ausführt.

Frage 8.9: Wie kann ein Testdatum ermittelt werden, das ein gewähltes Konstrukt ausführt?
Zur Lösung dieses Problems kann der folgende Lösungsansatz verwendet werden:

1. Bestimme einen Weg im Kontrollflussgraphen vom Anfang bis zu dem betrachteten Konstrukt.

2. Bestimmte, welche Bedingungen die Eingabevariablen des Programms erfüllen müssen, damit der Weg bis zu dem Konstrukt ausgeführt wird.

Der erste Schritt bei diesem Vorgehen ist ein **relativ einfaches graphentheoretisches Problem**. Allerdings gibt es oft mehrere Wege zu einem Konstrukt, welche, insbesondere wenn Schleifen auf dem Weg liegen, auch unterschiedlich lang sein können. Es sollten daher erst die kürzesten Wege betrachtet werden. Um zum Beispiel das Segment c_1c_2 in Abbildung 20 zu erreichen gibt es unendlich viele Wege. Allgemein sind diese der Form $a_1(b_1b_2)^k b_1c_1$ mit $k = 0, 1, 2, \ldots$, d.h. das Stück b_1b_2 kann beliebig oft wiederholt werden.

Ist ein solcher Weg endlich bestimmt, so müssen die Bedingungen für die Eingabevariablen ermittelt werden, damit der Weg ausgeführt wird. Diese Bedingungen ergeben sich aus den zu erfüllenden Prädikaten in den Entscheidungsknoten, die auf dem Weg liegen. Diese Bedingung wird **Wegebedingung** genannt. Die Prädikatbedingungen sind durch **„symbolische Ausführung rückwärts“** zu bestimmen.

Frage 8.10: Gebe ein Beispiel für die “symbolische Ausführung rückwärts“!
Als Beispiel dient das Programm in Abbildung 1(a) auf Seite 27. Nach dem in Kapitel 8.1 besprochenen reicht es zur Ermittlung der Zweigüberdeckung aus, die Entscheidungs-Entscheidungs-Wege (d, e, g) und (f, g) auszuführen. Dabei gibt es zwei kürzeste Wege zum Erreichen dieser beiden Entscheidungs-Entscheidungs-Wege:

- Der Weg $abcdeg$ für (d, e, g).
- Der Weg $abcfg$ für (f, g).

Auf die beiden Wege wird nun im einzelnen eingegangen:

1. Auf dem Weg bis zur Kante d liegen zwei Entscheidungsknoten. Welche Bedingungen müssen nun gelten damit an diesen beiden Entscheidungsknoten richtig verzweigt wird? Zuerst einmal muss der „nein“-Zweig der Bedingung $(y = 0)$ (Knoten C) abgelaufen werden. Es muss also $y \neq 0$ gelten. Danach muss der „ja“-Zweig der Bedingung $(y \mod 2 = 1)$ in Knoten D abgelaufen werden, die Bedingung muss also so gelten. Insgesamt muss bis hier also gelten:

   $$(y \bmod 2 = 1) \ \wedge \ y \neq 0$$

   Diese Bedingung wird von jeder ungeraden Zahl erfüllt. Allerdings sollte nicht einfach irgendeine ungerade Zahl gewählt werden, da Schleifen ja nicht zu oft (und schon garnicht unendlich oft) durchlaufen werden sollen. Es sollte also ein Testdatum gefunden werden, welches einen möglichst kurzen Weg durch das Programm ausführt. Damit die Schleife verlassen wird, sollte also nach dem ersten Durchlauf möglichst die Bedingung $(y = 0)$ gelten. Bevor der Knoten C jedoch wieder erreicht wird, wird y in Knoten F zuerst transformiert. Der neue Wert von y wird errechnet als $y = y \text{ div } 2$. Also muss y insgesamt drei Bedingungen erfüllen:

   $$(y \text{ div } 2 = 0) \ \wedge \ (y \bmod 2 = 1) \ \wedge \ y \neq 0$$

   Die Lösung ist der Schnitt der drei Mengen, die diese Bedingungen erfüllen, also:

   $$L = \{-1, 0, 1\} \cap \{y \in \mathbb{Z} \mid y \text{ ungerade}\} \cap \{y \in \mathbb{Z} \mid y \neq 0\} = \{-1, 1\}$$

   Die Testdaten bestehen also aus dem Tupel (x, y) wobei x beliebig ist und $y = -1$ oder $y = 1$ gelten muss.

2. Für den Weg $abcfg$ müssen ähnliche Bedingungen gelten:

$$(y \bmod 2 \neq 1) \ \wedge \ y \neq 0$$

Dabei kann der ersten Term zu $y \bmod 2 = 0$ umgeschrieben werden. Da auch in diesem Fall ein möglichst kurzer Weg ausgeführt werden soll, gilt auch hier:

$$(y \operatorname{div} 2 = 0) \ \wedge \ (y \bmod 2 = 0) \ \wedge \ y \neq 0$$

Die erste und die zweite Bedingung werden jedoch nur von $y = 0$ erfüllt, was dann im Gegenspruch zur dritten Bedingung ($y \neq 0$) steht. Also ist der kürzeste Weg $abcfgh$ nicht ausführbar. Wird jedoch für y z.B. $y = 2$ gesetzt, so wird der Weg $abcfgcdegh$ ausgeführt.

Frage 8.11: Welche Probleme kann es bei der Testdatenerzeugung geben?
Wie im obigen Beispiel zu sehen ist, kann es passieren, dass ein gewählter Weg nicht ausführbar ist. In diesem Fall kann also die Wegebedingung nicht erfüllt werden und es muss ein längerer Weg gewählt werden. Es stellt sich nun also die Frage, ob man der Wegebedingung denn ansehen kann, ob sie erfüllbar ist. Dies ist ist nur bei Wegen mit **einfachen Entscheidungen** der Fall:

Das Ausführbarkeitsproblem für einen Kontrollflussweg ist entscheidbar, wenn die **Wegebedingung linear in den Eingabevariablen bzw. -werten** ist. In diesem Fall kann mit den **Techniken der linearen Programmierung** eine Lösung gefunden werden bzw. gezeigt werden, dass keine Lösung exisitiert. Allgemein gilt jedoch leider:

Das Ausführbarkeitsproblem für Kontrollwege ist nicht entscheidbar. Dies liegt daran, dass es **nicht entscheidbar ist, ob ein System von Ungleichungen eine Lösung besitzt**. Die Wegebedingung für einen Kontrollflussweg stellt aber gerade ein solches System von Ungleichungen dar.

Desweiteren sind die folgenden Probleme bei der Anweisungs- und Zweigüberdeckung ebenfalls nicht entscheidbar:

1. Wird eine gegebene Anweisung durch irgendeine Eingabe ausgeführt?

2. Wird ein gegebener Entscheidungsausgang durch irgendeine Eingabe ausgeführt?

Desweiteren tritt noch das Problem der **Minimierung der Testdatenanzahl** auf, bei dem ich jedoch die Logik nicht so ganz verstehe (ein später erzeugtes Testdatum T_j, welches die von einem Testdatum T_i überdeckten Konstrukte ebenfalls überdeckt, soll weggelassen werden).

9 Statische Analyse und symbolische Ausführung

Frage 9.1: Worum geht es grob bei der statischen Analyse und der symbolischen Ausführung?

Bei der statischen Analyse und der symbolischen Ausführung wird ein Programm **nicht mit konkreten Eingabewerten**, sondern **„konzeptuell" ausgeführt**. Umgangssprachlich könnte man dies auch als „besonders genaues Anschauen der Programme" ausdrücken, wobei dieses Anschauen entweder manuell (also durch Menschen, informell) oder durch Rechner (formal) durchgeführt werden kann.

Frage 9.2: Welche Dokumente kommen für die statische Analyse in Betracht?

- Anforderungs- und Systemspezifikation
- Entwurfsspezifikation (Grob- und Feinentwurf)
- Programm in Quellsprache (Quellcode)
- Programm in Zielsprache (Objektcode)

Insbesondere die Dokumente aus 1 und 2 kommen für eine statische Analyse in Betracht, da sie im allgemeinen nicht ausführbar sind.

Frage 9.3: Welche Aspekte sind bei der Spezifikations- und Entwurfsanalyse zu überprüfen?

Bei der **Spezifikationsanalyse** sind die **folgenden Aspekte** zu überprüfen:

- **Notwendigkeit** (für Systemziele)
- **Vollständigkeit** (bezüglich Eingaben, Ausgaben, zu behandelnden Fällen, Umgebung, Leistung, Zuverlässigkeit, Benutzung)
- **Konsistenz** (Anforderungen untereinander, einheitliche numerischer Angaben [z.B. nur cm, m oder km])
- **Durchführbarkeit** (bei gegebener Hardware/Technologie)
- **Eindeutigkeit/Testbarkeit** (Begriffe und Sätze eindeutig; Transformationen eindeutig; beim Testen eindeutige Entscheidbarkeit, ob eine Anforderung erfüllt ist; Wartbarkeit)

Bei der **Entwurfsanalyse** gibt es teilweise ähnliche **Aspekte**:

- **Notwendigkeit** (bezüglich Anforderungsspezifikation)
- **Vollständigkeit** (bezüglich Anforderungsspezifikation)
- **Konsistenz** (Modulschnittstellen; Formate von Eingaben, Datenbanken und Dateien)

- **Korrektheit** (Algorithmen, mathematische Gleichungen, Kontroll-Logik des Feinentwurfs)

Dabei kann die Überprüfung sowohl bei der Spezifikationsanalyse als auch bei der Entwurfsanalyse durch **menschliche Begutachtung** erfolgen (siehe auch Kapitel 9.1). Um bei der Entwurfsanalyse die Konsistenz festzustellen, kann z.B. eine **Schnittstellenüberprüfung** durch den Compiler erfolgen und zur Nachweisung der Korrektheit kann die **mathematische Analyse** verwendet werden.

9.1 Informelle Analyse

Frage 9.4: Was versteht man unter einer informellen Analyse? In welchen Phasen der Software-Entwicklung sollte sie eingesetzt werden?
Die **informelle Analyse** entspricht der **„manuellen" Analyse** durch den Menschen und ist im Prinzip **bei allen nicht formalen Dokumenten notwendig**. Sie kann jedoch auch bei formalen Dokumenten (wie z.B. dem Quellcode) sinnvoll sein. Somit kann diese Art der Analyse **in jeder Phase des Software-Lebenszyklus** angewendet werden.

Frage 9.5: Welche Inspektionen sind bei der informellen Analyse vorzusehen, damit Fehler möglichst früh gefunden werden?
Im Prinzip kann zwischen drei Phasen unterschieden werden:

Spezifikation und Code: Hier werden zuerst die **interne Spezifikation** inspiziert (I0), dann die **Logik-Spezifikation als Entwurfsabnahme-Inspektion** (I1) und anschließend der **Code** (I2).

Ziel der drei Phasen I0, I1 und I2 ist es, die Qualität der Dokumente der frühen Entwicklungsphasen zu messen und zu beeinflussen.

Testplan und Testfälle: Hier kann zwischen zwei Phasen unterschieden werden. Zum einen wird der **Testplan** selbst inspiziert (IT1) sowie die **Testfälle und Testdaten** (IT2).

Ziel dieser Phasen ist es, die Qualität des Testens selbst zu erhöhen. In der Phase IT1 sollen z.B. Lücken bei der Funktionsabdeckung gefunden und beseitigt werden und in Phase IT2 sollen Fehler in den Testfällen und Testdaten gefunden werden. Dazu zählen zum Beispiel zu wenig Testfälle oder falsche erwartete Ausgaben (Solldaten).

Dokumentation: Da ein Software-Produkt meist nur dann sinnvoll verwendet werden kann, wenn eine gute Dokumentation vorliegt, sollte auch die Qualität der Dokumentation überprüft werden. Dies geschieht in den **Publikations-Inspektionen** PI0, PI1 und PI2 für das Benutzer,- Wartungs- und Installationshandbuch.

Frage 9.6: Welche verschiedenen Rollen gibt es bei einer Inspektion?

Im Prinzip gibt es vier Rollen bei einer Inspektion, die hier am Beispiel der I1- und I2-Inspektion erläutert werden. Bei den anderen Inspektionen (siehe oben) ist der Ablauf im Prinzip der gleiche, nur das Inspektionsmaterial und die Anzahl der Teilnehmer sind verschieden.

Moderation: Die Rolle des Moderators ist eine **„Schlüssel"-Rolle** und sollte daher nur von einer besonders **kompetenten Person** übernommen werden. Die Kompetenz bezieht sich dabei nicht nur auf die Programmierfähigkeiten, sondern auch auf Fingerspitzengefühl im Umgang mit Personen. Diese Rolle sollte **möglichst nicht von einem Projektmitglied** übernommen werden, um Unabhängigkeit zu wahren. Aufgaben des Moderators sind:

- Zusammenarbeit anregen
- Organisation (Dokumente prüfen und verteilen, Sitzungsräume reservieren, etc.)
- Inspektionsbericht erstellen
- Änderungen verfolgen

Entwurf: Die Person, die für den Entwurf des entsprechenden Moduls bzw. Dokument zuständig ist.

Codierung: Die Person, die für die Codierung zuständig ist.

Test: Die Person, die für das Testen zuständig ist.

Im allgemeinen sollten **höchstens vier Personen** an den Inspektionen teilnehmen. Wenn ein Modul jedoch viele Schnittstellen hat, sollten auch Entwickler des benutzten bzw. benutzenden Moduls dabei sein. Hat der Programmier auch den Entwurf gemacht, sollte er die Rolle der Entwerfers übernehmen und ein Programmierer aus einem ähnlichen Projekt die Rolle des Programmierers. Hat er das Programmstück auch getestet, muss eine weitere Person die Rolle des Testers übernehmen.

Frage 9.7: Wie lange sollte eine Inspektion höchstens dauern? Aus welchen Schritten besteht eine Inspektion?

Die Sitzungen sollten **nicht länger als zwei Stunden** dauern, wobei zwei Sitzungen à zwei Stunden pro Tag akzeptabel sind. Eine Inspektion besteht aus sieben Schritten:

1. Überblick (Gruppensitzung)
2. Vorbereitung (individuell)
3. Eigentliche Inspektion (Gruppensitzung)
4. Überarbeitung/Korrektur (Autor)
5. Verfolgung/Überprüfung (Moderator)
6. Auswertung/Nachbereitung (Moderator)

7. Inspektion als Einschub beim dynamischen Testen

Frage 9.8: Beschreibe die Schritte einer Inspektion im einzelnen!
Es wird nun einzeln auf die sieben Schritte eingegangen:

Überblick: Der Überblick geschieht in einer Gruppensitzung. Der **Entwerfer** gibt dabei einen **mündlichen Überblick** über den speziellen Programmteil und die Einordnung in die allgemeine Problemstellung. Bei einer I2-Inspektion (Code-Inspektion), die mit den gleichen Personen, wie bei der I1-Inspektion stattfindet, **kann dieser Schritt entfallen**, wenn die I1-Inspektion nicht zu lange her ist.

Desweiteren wird **schriftliches Material verteilt**. Bei einer I1-Inspektion wird die Entwurfsdokumentation verteilt, bei einer I2-Inspektion die Entwurfsdokumentation sowie der Programmtext.

Vorbereitung: Die Phase der Vorbereitung geschieht in Einzelarbeit. Zuerst muss die **Entwurfsdokumentation durchgelesen** werden, um ihre Logik und Intention zu verstehen. Desweiteren wird versucht, die fehlerträchtigen Konstrukte durch **Lesen einer Fehlerliste mit Häufigkeitsangaben** kennenzulernen. Außerdem werden **Checklisten** benutzt, in denen Anhaltpunkte stehen, wie Fehler gefunden werden können.

Eigentliche Inspektion: Die eigentliche Inspektion findet wieder in einer Gruppensitzung statt. Ein vom Moderator ausgewählter „Leser“ (im allgemeinen der **Codierer**) beschreibt, wie er den Entwurf implementieren wird bzw. implementiert hat. Dabei soll der **Entwurf in eigenen Worten umschrieben werden** und jedes Stück Logikentwurf bzw. jeder Zweig soll mindestens einmal angesprochen werden. Auch hier muss wieder **Dokumentation** in Form der Entwurfsspezifikation vorliegen sowie bei der I2-Inspektion die Programmtexte.

Das Ziel dieser Sitzung ist es, Fehler zu finden. Im allgemeinen geschieht dies während des Vortrags des Codierers. Dabei werden Fragen und Probleme **nur soweit verfolgt, bis ein Fehler erkannt wird**. Die Ursache des Fehlers bzw. seine Behebung soll nicht diskutiert werden. In dieser Phase besteht die Aufgabe des Moderators darin, die **Fehler zu notieren und zu klassifizieren**. Gibt es jedoch eine offensichtliche Lösung für ein Problem, wird auch diese notiert.

Das **Ergebnis dieser Sitzung** ist ein **schriftlicher Bericht**, der innerhalb eines Tages vom Moderator zu erstellen ist. In diesem werden die **Fehler vermerkt und bewertet**. Teil 1 dieses Berichts ist eine **Fehlerliste**, die die Beschreibungen der Fehler enthält, und Teil 2 ist eine **Übersicht**.

Überarbeitung/Korrektur: Alle im Inspektionsbericht notierten Fehler bzw. Probleme werden vom Erwerfer (bei I1) bzw. vom Codierer (bei I2) korrigiert bzw. gelöst.

Verfolgung/Überprüfung: Der Moderator überprüft, ob alle Fehler bzw. Probleme vom Entwerfer bzw. Codierer korrigiert bzw. gelöst wurden. Wurde dabei mehr als 5% des Produkts überarbeitet, wird eine neue Inspektion durchgeführt, ansonsten kann der Moderator selbst die Qualität der Überarbeitung überprüfen (oder eine neue Inspektion ansetzen).

Auswertung/Nachbereitung: Diesen Schritt gibt es aus zwei Gründen:

1. Module und Programmteile sollten **gemäß der Fehlerhäufigkeiten** bei der Inspektion **angeordnet werden**. Dabei sind die Module mit den meisten Fehlern noch einmal zu inspizieren bzw. härter zu testen.
2. Die Fehlersorten eines Moduls sollten mit der **mittleren Fehlersorten-Verteilung verglichen** werden, damit Fehlerursachen frühzeitig aufgespürt werden können bzw. der Programmierer entsprechend nachgeschult werden kann.

Inspektion als Einschub beim dynamischen Testen: Dieses Vorgehen ist bei sehr fehlerhaftem Code sinnvoll. Auswahlkriterien sind:

1. Module mit höchster Zahl von Testfehlern
2. Module mit geringer Testüberdeckung, die viele Inspektionsfehler bei der ersten Inspektion aufwiesen bzw. die durch die Programmierer als kritisch eingeschätzt werden.

Frage 9.9: Gebe Beispiele für Checklisten bei der I1- bzw. I2-Inspektion!

Frage 9.10: Wie sieht die Fehlerliste und die Übersicht des Berichts der Inspektionssitzung aus?

Frage 9.11: Was ist ein Walkthrough?

Ein Walkthrough unterscheidet sich von der oben beschriebenen Inspektion in Schritt 3. Beim Walkthrough gibt ein Gruppenmitglied einfache Testdaten vor und leitet die Gruppe an, das besprochene **Programmstück mit diesen Testdaten per Hand auszuführen**. Zwischenergebnisse werden dabei schriftlich festgehalten. Das Ziel ist dabei allerdings nicht die komplette Simulation eines Programmstücks, sondern das Anregen von Nachfragen bzw. einer Diskussion mit dem Entwickler des Programmstücks.

Frage 9.12: Welche Voraussetzungen müssen für eine informelle Analyse gegeben sein?

Voraussetzung für eine informelle Analyse sind **nachdrückliche Disziplin** und eine klar definierte Folge von Überprüfungsoperationen. Desweiteren sind die Teilnehmer in **effektiver Fehlersuche zu schulen**. Günstig ist auch eine vorbereitende Inspektion mit einem repräsentativen Programmstück. Die dabei

gefundenen Fehler sollten analysiert und klassifiziert werden und bilden dann so die Grundlage der **Testspezifikation**, welche in den folgenden Inspektionen verwendet und verbessert wird.

Frage 9.13: Was ist ein Review?
Unter Reviews versteht man Inspektionen und Walkthroughs.

Frage 9.14: Was sind die Vorteile von Reviews?
Einer der größten Vorteile der Reviews ist die **relativ schnelle Rückkopplung** der Ergebnisse an die Entwerfer und Programmierer. Diese lernen dabei:

- Welche Fehler(-sorten) sie am häufigsten machen.
- Wieviele Fehler sie machen.
- Wie diese Fehler gefunden werden können.

In den meisten Fällen verbessert ein Entwickler seine Fähigkeiten noch innerhalb des selben Projekts. Jedoch sollte diese Rückkopplung ausschließlich zum Vorteil des Entwicklers verwendet werden und nicht zur Erstellung von Leistungsmessdaten benutzt werden.

Desweiteren werden durch konsequente Reviews **an mehreren Stellen Fehler gefunden**:

- Von den Programmierern werden **vor der Übergabe des Materials** für die Review-Sitzung sowie bei der **eigenen Vorbereitung** auf die Review-Sitzung Fehler gefunden.
- Während der Reviewsitzung werden oft durch **synergetische Effekte** Fehler gefunden.
- Beim Code-Review werden oft Entwurfsfehler gefunden, durch neue, genauere Informationen (Beispiel: „Ach nein, dem Konstruktor muss auch die Instanz xy übergeben werden...“)

Die **qualitativen und wirtschaftlichen Vorteile** zeigen sich bei einem Experiment, bei dem es pro 1000 Zeilen Code (ohne Kommentare) zu folgenden Netto-Einsparungen kam:

- Inspektion der Logik (I1): +94 Programmierstunden
- Inspektion des Codes (I2): +51 Programmierstunden
- Inspektion des Modultests (I3): -20 Programmierstunden

Da bei der Inspektion des Modultest nichts eingespart werden kann, sollte dieser Schritt entfallen. Desweiteren zeigen einige andere Studien, wie die von Shooman, dass Code-Lesen sehr viel köstengünstiger ist als ein computergestütztes Fehlerfinden.

Frage 9.15: Welche Nachteile haben Reviews?
Review haben leider auch einige Nachteile bzw. es ergeben sich teilweise neue Probleme bei ihnen:

- Da sich Programmierer meist in einer **Programmiersprache sicherer fühlen** als in einer Spezifikations- oder Entwurfssprache, finden sie ein meist frühzeitiges Codieren sinnvoller als eine nochmalige Überprüfung des Entwurfs bzw. Testen sinnvoller als Inspizieren des Codes.
- **Manager sind von den Techniken nicht immer überzeugt**, da der Erfolg nicht direkt sichtbar ist und die Inspektionen Zeit kosten (die quasi Geld ist).

Desweiteren können bei einer Inspektion in Gruppen, neben den in Kapitel 2.4 vorgestellten Täuschungen, auch die folgenden Probleme auftreten:

- **Konzentration auf Funktionsfehler** und **Vernachlässigung anderer Qualitätsmerkmale**, wie z.B. Wartbarkeit, Portabilität oder Wiederverwendbarkeit.
- **Kein systematisches Suchen nach allen Fehlerarten**, evtl. **Verzetteln** beim Diskutieren von Konventionen für Kommentare oder von anderen Trivialitäten.
- **Dominanz in Gruppen**. Ein starker, gut vorbereiteter Teilnehmer verhindert die nützlichen Beiträge anderer Teilnehmer, und schlecht vorbereitete Teilnehmer können sich unbemerkt zurückhalten.

Die drei obigen Fehler können mit Hilfe von sogenannten **Einzelinspektor-Phasen** gelöst werden. Dabei bekommt eine einzelne Person eine **Checkliste**, anhand derer sie das vorliegende Dokumente auf **einen (oder wenige) Aspekt(e)** hin überprüft. Dabei bieten sich die folgenden Aspekte als Untersuchungsziel für entsprechende Phasen an:

1. **Syntaktische Aspekte der Dokumentation** (die Grammatik, Rechtschreibung, Formatierung)
2. **Layout des Quellcodes**
3. **Code-Lesbarkeit** (Variablennamen, Abkürzungen, Namensstandards)
4. **Programmierstil** (keine unnötigen *goto*s, keine Zuweisungen in Booleschen Ausdrücken)
5. **Korrektheit von Programmkonstrukten** (z.B. Inkrementierung von Schleifenvariablen)

9.2 Formale Analyse

9.2.1 Fehleranalyse

Frage 9.16: Welche Analysearten kommen bei der formalen statischen Analyse in Betracht?

Es lassen sich grob vier Analysearten unterscheiden:

Fehleranalyse im engeren Sinne: Zur Fehleranalyse im engeren Sinne zählt eine **Typ-, Variablen-, Prozedur- und Datei-Analyse**. Bei der **Typanalyse** wird geschaut, ob der Wertebereich eines Typ eingehalten wird und ob die Operationen auf ihm erlaubt sind. Im allgemeinen erledigt dies ein guter Compiler. Desweiteren werden spezielle Typangaben für Arrays, Schleifenzähler und Einheiten empfohlen.

Bei der **Variablenanalyse** werden die Lebens- und Gültigkeitsbereiche von Variablen analysiert. So kann zum Beispiel ein Zeiger eine längere Lebenszeit haben als das Objekt, auf das er zeigt, wenn dieses vorher mit `delete` oder `free` gelöscht wurde.

Ein Ziel der **Prozeduranalyse** ist unter anderem die Überprüfung. ob Schnittstellen eingehalten werden.

Zugriffe auf Dateien dürfen in den meisten Systemen nur unter Einbehalteung einer speziellen Reihenfolge von Befehlen erfolgen. Diese sind: Datei reservieren – Datei öffnen – Datei lesen/schreiben – Datei schließen. Ziel der **Analyse des Zugriffs auf Dateien** ist es, nachzuschauen, ob diese Reihenfolge eingehalten wird.

Analyse der Programmstruktur: Hier wird die Struktur des Programms näher untersucht. Unter anderem:

1. Ist das Programm strukturiert (d.h. ohne *goto*s)?
2. Hat das Programm unerreichbare Anweisungen (z.B. hinter *goto*s)?
3. Hat das Programm nicht ausführbare Wege?
4. Welche Schleifen terminieren in jedem Fall?

Programmierrichtlinien: Eine weitere Art der Analyse ist die Analyse, ob Programmierrichtlinien eingehalten wurden. Dabei wird überprüft, ob formale Richtlinien für Kommentare und das (Code-)Layout sowie ob die Namenskonventionen eingehalten wurden. Oder ob verbotene Konstrukte benutzt wurden.

Ausdrucksanalyse: Hier werden die folgenden Fragen gestellt:

1. Werden Grenzen von Arrays überschritten?
2. Werden Real-Zahlen auf (Un-)Gleichheit abgefragt? Also zum Beispiel `if(r==s) then` anstatt `if(abs(r-s) < epsilon) then`, wie es eigentlich korrekt wäre.
3. Untersuchung des Datenfluss im Programm auf Datenflussanomalien.

Im Laufe der ersten drei Analysearten können viele Informationen über das Programm gesammelt werden, aus denen die folgenden Dokumente erzeugt werden können:

1. Kontrollflussgraph

2. Komplexitätsmaße für einige Konstrukte (siehe Kapitel 12.2)
3. Operationsaufrufgraph (siehe Kapitel 10.2)

9.2.2 Datenflussanalyse

Frage 9.17: Was ist das Ziel der Datenflussanalyse?
Ziel der Datenflussanalyse ist es, **Datenflussanomalien** bei der Benutzung einer Variablen festzustellen. Im allgemeinen bedeutet dies, dass eine Variable entweder garnicht benutzt wird oder aber benutzt wird, obwohl sie (noch) nicht benutzt werden darf. Wie schon aus Kapitel 5 bekannt ist, kann eine Variable definiert, referenziert oder undefiniert werden. Anomalien in diesem Zusammenhang sind dann:

- Eine Referenz vor der ersten Definition. (Wert also unbestimmt)
- Zwei aufeinander folgende Definitionen ohne zwischenzeitliche Referenz.
- Eine Definition gefolgt von einer Freigabe *(undef)*.

Frage 9.18: Wie wird bei der Datenflussanalyse vorgegangen?
Die Datenflussanalyse geschieht in drei Schritten:

1. Die Wege („Pfade“) durch den Kontrollflussgraphen werden durchlaufen.
2. Für jede Variable wird dabei notiert, ob sie definiert, referenziert oder undefiniert wird, wobei die Abkürzungen d, r und u benutzt werden. Dabei wird bei einer Zuweisung die rechte Seite *vor* der linken Seite betrachtet. Dies ist bei Zuweisungen der Form $A = A + B$ wichtig. Auf diese Art und Weise entsteht für jede Variable und jeden Pfad ein sogenannter **Pfadausdruck** über dem Alphabet $\{d, r, u\}$.
3. Eine Anomalie liegt in den folgenden Pfadausdrücke vor, wobei α und β beliebige Folgen der Symbole d, r, u sind, also $\alpha, \beta \in \{d, r, u\}^*$:

   ur-Anomalie: $\alpha ur \beta$ – Eine Variable wird also referenziert, obwohl sie nicht mehr definiert ist.

   dd-Anomalie: $\alpha dd \beta$ – Eine Variable wird zweimal definiert, ohne zwischendurch referenziert zu werden.

   du-Anomalie: $\alpha du \beta$ – Eine Variable wird definiert und später gleich wieder undefiniert ohne zwischendurch benutzt zu werden.

Frage 9.19: Welche Programmkonstrukte bereiten bei der Datenflussanalyse Probleme?
Im Prinzip bereiten die Konstrukte Probleme, bei denen nicht schon vor der Ausführung bekannt ist, wie sie sich verhalten. Also die Konstrukte, die eher **dynamischer Natur** sind. Dies sind:

Aufrufe von Operationen: Hier können z.B. rekursive Aufrufe Probleme machen, da der dazugehörige Kontrollflussgraph im allgemeinen unendlich gross werden kann (z.B. rekursive Berechnung von $f(n) = n!$ durch $f(n) = n \cdot f(n-1)$. Für verschiedene n wird die Funktion verschieden oft aufgerufen, bei einer Datenflussanalyse hat n jedoch keinen konkreten Wert.

Dynamisch wachsende Strukturen: Dies sind Datenstrukturen, die zur Laufzeit dynamisch wachsen können, wie z.B. Listen, Bäume und Graphen. Zugriffe auf einzelne Elemente lassen sich nicht statisch analysieren.

Arrays: Bei Arrays ergibt sich z.B. das Problem wenn z.B. ein Aufruf wie $A[k] = i$ im Programm vorkommt, wobei k vorher aus einer Datei ausgelesen wurde. Welches $A[k]$ soll dann als definiert gelten?

Schleifen: Schleifen können ebenfalls beliebig oft durchlaufen werden. Es entsteht also ein ähnliches Problem, wie bei rekursiven Operationsaufrufen. Das Schleifenproblem wird jedoch beim LIVE/AVAIL-Algorithmus und bei der algebraischen Methode nach Forman korrekt gelöst.

Frage 9.20: Welche Algorithmen kennst du, um Datenflussanomalien festzustellen?

- LIVE-/AVAIL-Algorithmen
- Algebraische Methode nach Forman
- Algorithmus von Howden (nicht wirklich)

Frage 9.21: Wie funktionieren die LIVE-/AVAIL-Algorithmen? Welche Voraussetzungen müssen gegeben sein?

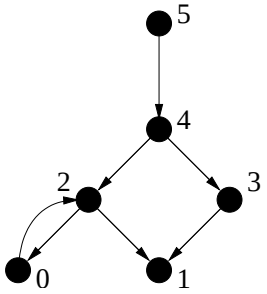
Die LIVE-/AVAIL-Algorithmen können benutzt werden um Datenflussanomalien festzustellen. Die eigentlichen LIVE-/AVAIL-Algorithmen operieren auf den Mengen *generate*, *kill* und *null*. Diese Begriffe müssen den Begriffen *def*, *ref* und *undef* zugeordnet werden, um in der Datenflussanalyse Anwendung zu finden.

Als Voraussetzung für diese Algorithmen muss ein Kontrollflussgraph gegeben sein, bei dem für jeden Knoten l die Mengen *generate(l)*, *kill(l)* und *null(l)* definiert sind und desweiteren gilt:

$$generate(l) \cup kill(l) \cup null(l) = tok$$

Dabei ist *tok* eine endliche Menge von „Token“, welche im Falle der Datenflussanalyse die Variablen sind.

Wird nun für eine Variable („Token“) A durch den Graph gelaufen, werden für jeden Knoten die Anfangsbuchstaben der Menge notiert, in der sich die Variable für diesen Knoten A befindet. Auf diese Art und Weise entsteht ein **Pfadausdruck** $P(A, w)$ über dem Alphabet $\{g, k, n\}$. Dieser kann zu einem **reduzierten Pfadausdruck** $P_r(A, w)$ umgewandelt werden, indem die



tok = {A, B}
null(5) = {A, B}
kill(4) = {A, B}
generate(3) = {B}, null(3) = {A}
generate(2) = {A}, null(2) = {B}
kill(1) = {A, B}
null(0) = {A, B}

Abbildung 22: Kontrollflussgraph mit Mengen *generate, kill, null*

n weggelassen werden. Ebenso lassen sich **alle** Wege, die von einem Knoten k wegführen bzw. die zu einem Knoten k hinführen, als ein sequentieller Pfadausdruck schreiben, in dem auch Alternativen und Iterationen vorkommen können. Auch dieser Pfadausdruck lässt sich reduzieren.

Abbildung 22 zeigt ein Beispiel anhand dessen die Begriffe erläutert werden. So ergibt der Weg für A über $w = 5, 4, 2, 0, 2, 1$ zum Beispiel den Pfadausdruck $P(A, w) = nkgngk$ und den reduzierten Pfadausdruck $P_r(A, w) = kggk$. Der Pfadausdruck $P(A, 5 \rightarrow)$ beschreibt für A alle möglichen Pfade von 5 weg und lautet $P(A, 5 \rightarrow) = k(g(ng)^* + n)k$. $P_r(A, 5 \rightarrow) = k(gg^* + \varepsilon)k$ ist der entsprechende reduzierte Pfadausdruck. $P(A, \rightarrow 3)$ beschreibt alle Wege für A zum Knoten 3 hin und lautet $P(A, \rightarrow 3) = nk$ oder reduziert $P_r(A, \rightarrow 3) = k$

Frage 9.22: Wie ist die Lebendigkeit einer Variable definiert? Wie die Verfügbarkeit?

Es werden nun für einen Knoten l die Mengen *live(l)* und *avail(l)* definiert. Dies ist die Menge der Variablen, die im Knoten l lebendig bzw. verfügbar sind.

Lebendigkeit: Es gilt $A \in live(l) \Leftrightarrow$ einer der Wege, die von l ausgehen, enthält als erste „Aktion“ (abgesehen von *null*) das Symbol g für *generate*. Dies bedeutet also, dass A auf irgendeinem Weg in Zukunft noch generiert werden kann.

Verfügbarkeit: Es gilt $A \in avail(l) \Leftrightarrow$ alle Wege, die zu l hinführen enthalten als letztes Symbol (abgesehen von *null*) das Symbol g. Dies bedeutet also, dass die generierte Variable A in l stets verfügbar ist, unabhängig davon, welcher Weg gewählt wird.

Frage 9.23: Wie werden die Lebendigkeit und die Verfügbarkeit einer Variablen mit Hilfe von Pfadausdrücken definiert?

Die Begriffe der Lebendigkeit und Verfügbarkeit können auch mit Hilfe von Pfadausdrücken definiert werden:

Lebendigkeit: $A \in live(l) :\Leftrightarrow P_r(A, l \rightarrow) = g \cdot p + p'$. Dabei sind p und p' Pfadausdrücke über g und k.

Verfügbarkeit: $A \in avail(l) :\Leftrightarrow P_r(A, \rightarrow l) = p \cdot g$. Dabei ist p ein Pfadausdruck über g und k.

Frage 9.24: Wie kann die Lebendigkeit und die Verfügbarkeit einer Variablen ermittelt werden?

Zur Ermittlung der Lebendigkeit bzw. Verfügbarkeit einer Variablen gibt es zwei iterative Algorithmen. Algorithmus 2 berechnet die Menge *live* für jeden Knoten des Graphen. Zuerst werden alle *live*-Mengen auf die leere Menge gesetzt und der Algorithmus so lange ausgeführt, bis sich keine der Mengen mehr ändert. Es wird nun in jedem Schritt der *while*-Schleife für jeden Knoten die neue Menge *live* berechnet. Da nur einer der reduzierten Pfadausdrücke das Symbol g enthalten muss, wird die Vereinigung der Mengen gebildet. Wie kann man sich den Algorithmus nun vorstellen? Wie schon gesagt, enthält die Menge *live* in einem Knoten eine Variable, wenn auf einem der Wege, von diesem Knoten weg, die Variable generiert wird und davor höchstens lauter *null* kommen. Betrachten wir nun einen Knoten x, dessen direkter Nachfolger n eine Variable generiert. Dieser Nachfolger n wird nun jedes Mal im Algorithmus an der $\bigcup$-Stelle betrachtet und sorgt dafür, dass die Variable in die Menge *live* des Knotens x gelangt, da die Variable ja in diesem Knoten erzeugt werden kann, nämlich einfach indem man den Weg entlang des Nachfolgers wählt. Wenn der Knoten x nun selber auch noch einen Vorgänger p hat, so wird er dafür sorgen, dass die Variable auch in dessen Menge *live* gelangt, mit einer Ausnahme: Wenn die sich die Variable in der Menge *kill* des Knotens x selbst befindet. Und genau dies drückt die große Vereinigung ($\bigcup$) aus. Eine Variable ist in der Menge *live* eines Knotens, wenn sie in einem seiner direkten Nachfolgerknoten generiert wird oder wenn sie sich in dessen Menge *live* befindet ohne dass sie auch in dessen *kill* ist. Wenn man sich diesen Algorithmus nun bildlich vorstellt, „kämpft“ sich ein *generate* also den Graphen hoch und wird dabei von jedem *null* weiter transportiert. Dabei kann sie nur von einem *kill* „gekillt“ werden.

Algorithmus 2 Der Algorithmus zur Berechnung von *live*

```
for j = 0 to |N| − 1 do
  live(j) = ∅;
end for
change = true;
while change do
  change = false;
  for j = 0 to |N| − 1 do
    previous = live(j);
    live(j) = ⋃_{l∈S(j)} [(live(l) \ kill(l)) ∪ generate(l)]
    if live(j) ≠ previous then
      change = true;
    end if
  end for
end while
```

Der Algorithmus zur Berechnung von *avail* läuft im Prinzip fast ähnlich. Nur dass hier alle Mengen, mit Ausnahme des Startknotens, auf *tok* gesetzt werden. Da hier ein Kriterium für *alle* Wege gefordert wird, wird jeweils der Schnitt der Mengen gebildet. Außerdem werden nun jeweils die Vorgängerknoten be-

trachtet. Die Definition einer Variable muss also von oben zum entsprechenden Knoten „durchsacken“, ohne dass zwischendurch ein *kill* erfolgt. Desweiteren muss dies für alle Wege gelten, die bei einem betrachteten Knoten ankommen.

Algorithmus 3 Der Algorithmus zur Berechnung von *avail*

```
avail(0) = ∅
for j = 1 to |N| − 1 do
  avail(j) = tok;
end for
change = true;
while change do
  change = false;
  for j = 1 to |N| − 1 do
    previous = avail(j);
    avail(j) = ⋂_{l∈P(j)} [(avail(l)\kill(l)) ∪ generate(l)]
    if avail(j) ≠ previous then
      change = true;
    end if
  end for
end while
```

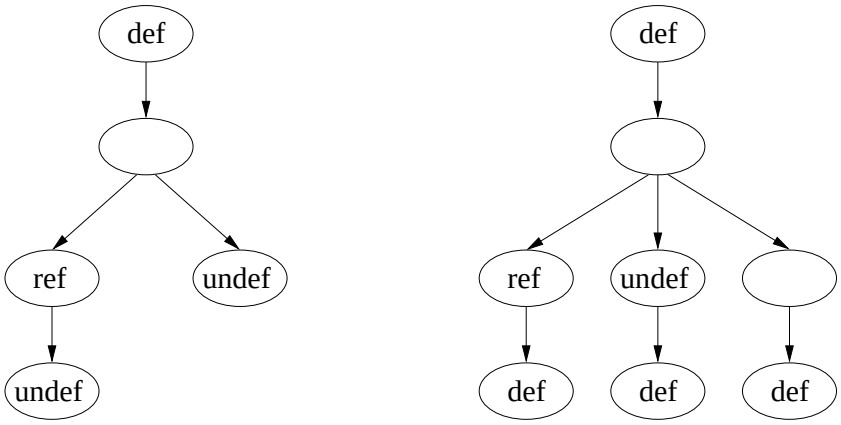
Frage 9.25: Wie kann eine ur-Anomalie mit Hilfe der LIVE-/AVAIL-Algorithmen festgestellt werden?

Eine ur-Anomalie kann mit Hilfe des *avail*-Algorithmus festgestellt werden. Wenn eine Variable in einem Knoten x referenziert wird, darf es keinen Weg zu diesem Knoten geben, auf dem die letzte Aktion ein *undef* war. Also wird nun für jeden Knoten l folgendes gesetzt: *generate(l)=def(l)* und *kill(l)=undef(l)*. Dann gilt für jeden Knoten l und jede Variable A:

$A \notin avail(l)$ und $A \in ref(l)$ genau dann, wenn ein Weg im Kontrollflussgraph mit einem Prädikatausdruck $\alpha ur \beta$ für A existiert, wobei gilt $\alpha, \beta \in \{r, d, u\}^*$ und die Referenz von A erfolgt im Knoten l. Es liegt also eine ur-Anomalie vor.

Wenn also $A \notin avail(l)$ gilt, dann gibt es einen Weg zu l, der als letzte „Aktion“ ein *kill* enthält, eventuell gefolgt von ein paar *null*. Der Pfadausdruck für diesen Weg lautet also αkn^* mit $\alpha \in \{r, d, u\}^*$. Es gilt *kill=undef* und *generate=def*, also entspricht *null* entweder einem *ref* oder garkeiner Benutzung der Variable. Dementsprechend lässt sich der Pfadausdruck übersetzen zu αur^*, also können nach dem *undef u* noch ein paar *ref r* folgen. Wenn dies jedoch für $A \notin avail(l)$ gilt und in l die Variable A referenziert wird, dann sieht der Ausdruck bis zum Knoten l wie folgt aus: $\alpha ur^* r$. Also liegt ein Pfadausdruck $\alpha ur\beta$ mit $\alpha, \beta \in \{r, d, u\}^*$ vor, wobei die Referenz von A im Knoten l erfolgt. Der letztgenannte Pfadausdruck charakterisiert jedoch gerade eine ur-Anomalie. Im Buch steht jetzt q.e.d., obwohl anscheinend die Rückrichtung fehlt.

Frage 9.26: Wie können die dd- und du-Anomalien mit Hilfe der LIVE-/AVAIL-Algorithmen festgestellt werden?



(a) du-Anomalie (b) dd-Anomalie

Abbildung 23: Feststellung der du- und dd-Anomalien

Die du- und dd-Anomalien können mit Hilfe des *live*-Algorithmus festgestellt werden. Abbildung 23 zeigt dabei die Fälle, die Auftreten können. Wie muss nun die Zuordnung zu den Mengen *generate, kill* und *null* geschehen, damit die beiden Anomalien erkannt werden? Betrachtet wird in beiden Fällen jeweils ein Knoten l in dem die Variable A definiert wird. In Abbildung 23(a) und 23(b) ist dies jeweils der oberste Knoten.

Betrachten wir zuerst die du-Anomalie. Eine du-Anomalie liegt vor, wenn eine Variable in einem Knoten definiert wird und in einem späteren Knoten undefiniert wird, ohne dass sie dazwischen benutzt, also referenziert, wird. Stellt man sich nochmals bildlich vor, dass sich eine Variable/ein Token den Graphen hochkämpfen muss, so muss sich also jedes *undef* „hochkämpfen“, damit man in einem Knoten in dem die Variable definiert wird also eine du-Anomalie feststellen kann. Das *undef* darf jedoch nur durchkommen, wenn keine Referenz auf dem Weg liegt. Damit dürfte die „Rollenverteilung“ im *live*-Algorithmus klar sein. Die Menge *generate* ist die Menge *undef* und die Menge *kill* entspricht der Menge *ref*. Eine Anomalie liegt im Fall $A \in def(l)$ und $A \in live(l)$ vor.

Bei einer dd-Anomalie sieht es ähnlich aus, nur dass sich hier ein *def* zu einem anderen *def* „hochkämpfen“ muss. Desweiteren kann dieses (im Sinne einer dd-Anomalie) jedoch durch ein *ref* und ein *undef* aufgehalten werden, da in beiden Fällen die Definitionen nicht mehr direkt aufeinander folgen. Also gilt $generate = def$ und $kill = ref \cup undef$. Auch hier gilt, dass eine Anomalie im Fall $A \in def(l)$ und $A \in live(l)$ vorliegt.

Frage 9.27: Wie funktioniert die algebraische Methode nach Forman?

Die **algebraische Methode von Forman** zur Bestimmung von Datenflussanomalien kann allgemein auf Kontrollflussdiagramme angewandt werden, deren Kontrollfluss sich in Form eines **regulären Ausdrucks** darstellen lassen. Dabei ist „$\cdot$“ die Konkatenation oder **Sequenz** im Kontrollfluss, „+“ die Alternative oder **Verzweigung** und „*“ der Kleene-Stern oder die **Iteration**.

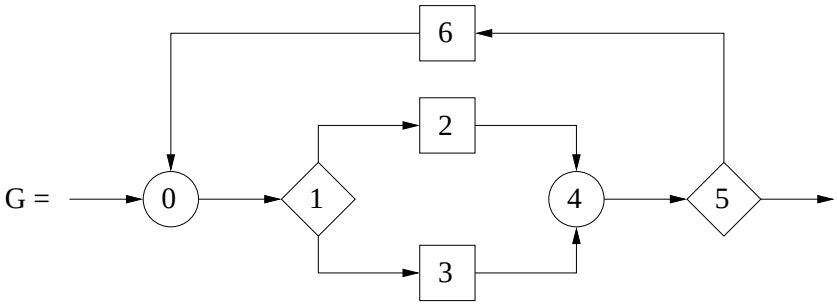


Abbildung 24: Kontrollflussschema G

So kann das Kontrollflussschema G aus Abbildung 24 zum Beispiel als der folgende reguläre Ausdruck dargestellt werden:

$$R(G) = 0 \cdot 1 \cdot (2 + 3) \cdot 4 \cdot 5 \cdot [6 \cdot 0 \cdot 1 \cdot (2 + 3) \cdot 4 \cdot 5]^*$$

Es werden nun für (alle) Variablen v auf verschiedenen Ebenen Konstrukte s betrachtet, wobei die folgenden Datenflussaussagen für die entsprechenden Konstrukte s algebraisch notiert werden:

an: Es liegt eine Anomalie im Konstrukt s vor.

ee: Es gibt einen Weg durch s ohne Aktion auf Variable v.

din, rin, uin: Es gibt einen Weg durch s, so dass die erste Aktion („in“) für v ein *def, ref* oder *undef* ist.

dout, rout, uout: Es gibt einen Weg durch s, so dass die letzte Aktion („out“) für v ein *def, ref* oder *undef* ist.

Dies sind acht Aussagen, die für jede Variable und jedes Konstrukt in einer acht-stelligen Binärzahl gespeichert werden können. Sei z.B. eine Variable v gegeben und die Anweisung $v = v + 1$. Dann ist die Binärzahl für $v = 00\ 010\ 100$, da die erste Aktion für v eine Referenz ist und die letzte Aktion eine Definition. Für jedes Konstrukt s wird ein Vektor angelegt, der aus den Binärzahlen für alle Variablen besteht. Werden nun zwei Konstrukte B und C zu einem neuen Konstrukt A zusammengesetzt, dann können die Bits der neuen Binärzahlen im neuen Bitvektor nach einigen Regeln errechnet werden. Dabei bezeichnet $s.x$ den neuen Wert der Aussage x für die jeweils betrachtete Variable im Bitvektor und das Konstrukt s. Es werden nun die Regeln für die Sequenz, die Verzweigung und die Iteration angegeben:

Konkatenation/Sequenz: Gegeben ist also $A = B \cdot C$. B wird vor C ausgeführt, d.h. der Ausgang („out“) von B ist an den Eingang („in“) von C angeschlossen. Es wird nun zuerst betrachtet, wie die Kontrollflussaussage $A.an$ berechnet wird. Es liegt eine Kontrollflussanomalie in A vor, wenn entweder eine in A oder B vorliegt, oder durch das aneinander fügen eine neue Anomalie entsteht. Dies können die bekannten Anomalien dd, ur

und *du* sein. Als logische Formel ausgedrückt also:

$$\begin{aligned} A.an \;=\; & B.an \;\vee\; C.an \;\vee\; \overbrace{(B.dout \wedge C.din)}^{\text{dd-Anomalie}} \\ & \vee\; \underbrace{(B.uout \wedge C.rin)}_{\text{ur-Anomalie}} \;\vee\; \underbrace{(B.dout \wedge C.uin)}_{\text{du-Anomalie}} \end{aligned}$$

$B.an$ und $C.an$ sorgen also dafür, dass sich eine Anomalie wie ein Krebsgeschwür fortpflanzt, sobald sie irgendwo auftaucht.

Damit es einen Weg durch A gibt, ohne dass etwas für eine Variable v passiert, muss es durch B und C einen Weg geben, ohne dass etwas passiert. Oder als Formel:

$$A.ee = B.ee \wedge C.ee$$

Damit die erste Aktion einer Variablen in A eine Definition, Referenz oder eine Undefinition ist, muss diese entweder als erstes in B gegeben sein oder in B passiert nichts und die Aktion ist die erste in C:

- $A.din = B.din \;\vee\; (B.ee \wedge C.din)$
- $A.rin = B.rin \;\vee\; (B.ee \wedge C.rin)$
- $A.uin = B.uin \;\vee\; (B.ee \wedge C.uin)$

Ähnliches gilt für die „out“-Aussagen:

- $A.dout = (B.dout \wedge C.ee) \;\vee\; C.dout$
- $A.rout = (B.rout \wedge C.ee) \;\vee\; C.rout$
- $A.uout = (B.uout \wedge C.ee) \;\vee\; C.uout$

Alternative/Verzweigung: Gegeben ist also $A = B + C$. In diesem Fall gilt:

$$A.x = B.x \vee C.x \text{ für alle } x \in \{an, ee, din, rin, uin, dout, rout, uout\}$$

Iteration: Es gelte $A = B^*$. Dies kann in Form einer Alternative geschrieben werden:

$$B^* = \varepsilon + B + B \cdot B + B^3 + B^4 + \cdots = \varepsilon + \sum_{i=1}^{\infty} B^i$$

Dabei steht ε für keine Ausführung von B. Es scheint nun aber so, als ob unendlich viele Alternativen betrachtet werden müssen. Dies ist jedoch zum Glück nicht der Fall. Denn es gilt, dass Datenflussanomalien, die in einer Schleife B^* bzw. vor oder nach dieser Schleife entstehen, entweder durch **keine oder durch zwei Iterationen erzeugt werden**. Es reicht daher die Betrachtung von $E + B \cdot B$ aus, wobei $E = 01\ 000\ 000$ ist. E sagt also aus, dass rein garnichts passiert, da nur $ee = true$ ist.

Frage 9.28: Welche Vor- und Nachteile hat die formale, statische Analyse?
Die formale, statische Analyse bietet die folgenden **Vorteile**:

- Sie erfordert nur **minimalen zusätzlichen Aufwand**. Der Quellcode muss nicht geändert werden, es müssen keine Treiber und Platzhalter erstellt werden und auch kein zusätzlicher Ausgabecode eingefügt werden.
- Da die **Analyse automatisch erstellt** wird, erfordert sie keine Änderungen der Arbeitsgewohnheiten der Programmierer.
- In **einem Lauf** können **mehrere Fehler entdeckt** werden.
- Es können **unausführbare Systeme** behandelt werden, also z.B. Systeme, bei denen einige Funktionen oder Unterprozeduren fehlen. Daher sollte die statische Analyse vor dem dynamischen Testen eingesetzt werden.

Es gibt jedoch auch **Nachteile**:

- Die formale, statische Analyse kann **dynamische Ereignisse nur begrenzt interpretieren**. Ist z.B. das folgende Programm gegeben, dann wird eine undefinierte Referenz von A als Fehler gemeldet, obwohl das Programm immer korrekt arbeiten wird:

  R=0;
  for $I = 1$ **to** 100 **do**
  if $I = 1$ **then**
  $A = 5$;
  end if
  $R = A * (I - 1) + R$;
  end for

  Dies liegt daran, dass angenommen wird, dass die *if*-Abfrage beim ersten Mal auch *false* werden kann, was aufgrund der *for*-Schleife davor jedoch nicht der Fall ist.

- **Ungewissheit der Diagnose**. Die entdeckten Anomalien müssen stets von einem Menschen gegengecheckt werden, um zu schauen, ob es sich wirklich um einen Fehler handelt. Bei einer dd-Anomalie ist die z.B. nicht immer der Fall, da bei einigen Programmen aus Effizienzgründen eine dd-Anomalie vorliegen kann. Dies ist sogar in dem LIVE- und dem AVAIL-Algorithmus bei der Variablen „change“ der Fall.
- **Abhängigkeit vom Programmierstil**. Bei einem schlechten Programmierstil „hagelt“ es nur so Warnungen.

9.3 Symbolische Ausführung

Frage 9.29: Wie funktioniert die symbolische Ausführung?
Die **symbolische Ausführung** eines Programms ist eine Erweiterung des Begriffs der Ausführung eines Programms mit konkreten Werten. Anstatt konkreter Werte werden **symbolische Werte** für die Eingabevariablen verwendet, mit denen entsprechend „gerechnet“ wird. Dabei müssen verschiedene Konstrukte unterschieden werden, auf die man im Verlauf der „Ausführung“ treffen kann. Auf diese wird nun jeweils einzeln eingegangen.

Ausdrücke/Zuweisungen: Der symbolische Wert einer Variablen A sei durch $w(A)$ gegeben. Also z.B. $w(A) = a$, was bedeutet, dass der symbolische Wert der Variablen A momentan a beträgt. Der symbolische Wert kann auch komplizierter aussehen, wie wir gleich sehen werden. In einer Zuweisung werden die Variablen dann durch ihre symbolischen Werte ersetzt, mit denen unter Umständen noch weitergerechnet werden kann, um den Ergebnisausdruck zu vereinfachen. An einem Beispiel wird dies schnell klarer.

Gegeben sei die Zuweisung $C = A + 2 \cdot B$. Die symbolischen Werte der Variablen A und B seien vor der Ausführung der Zuweisung $w(A) = a$ und $w(B) = b$. Dann ergibt sich der symbolische Wert der Variablen C als:

$$w(C) = w(A) + 2 \cdot w(B) = a + 2 \cdot b$$

Hier ist schon ein erster Fall, bei dem der symbolische Wert einer Variablen schon etwas komplizierter aussieht. Wie auch schon erwähnt kann es aber auch vorkommen, dass sich der symbolische Wert einer Variablen nach einer Zuweisung vereinfacht. Die ist zum Beispiel der Fall für die Anweisung $D = C - A$, wenn diese nach der obigen Zuweisung kommt. Dann vereinfacht sich $w(D)$ zu:

$$w(D) = w(C) - w(A) = a + 2 \cdot b - a = 2 \cdot b$$

Allerdings kann es dabei aufgrund der Computer-Arithmetik zu Rundungsfehlern kommen. So könnte es z.B. passieren, dass der letzte Ausdruck zu $2 \cdot b + 0.00000001 \cdot a$ „vereinfacht“ wird.

Verzweigungen: Verzweigungen können bei verschiedenen Konstrukten, wie *if-then-else*-Anweisungen oder Schleifen auftreten. Das Vorgehen der symbolischen Ausführung in Falle einer Verzweigung wird anhand des Beispiels `if(B) then A1 else A2` erläutert. In der Bedingung B, die abgefragt wird, werden die Variablen ebenfalls durch ihre symbolischen Werte ersetzt. Auf diese Art und Weise entsteht der symbolische Ausdruck $w(B)$. Für diesen können nun drei Fälle unterschieden werden:

1. Es lässt sich zeigen, dass **stets** $\mathbf{w(B) = true}$ gilt. In diesem Fall muss nur mit $A1$ symbolisch weiter gerechnet werden, da man sich auf einem Pfad befindet, auf dann die andere Bedingung nie gelten kann.

2. Es lässt sich zeigen, dass **stets** $\mathbf{w(B) = false}$ gilt. In diesem Fall muss, aus demselben Grund wie in 1, nur mit $A2$ weitergerechnet werden.
3. Es lässt sich **nicht zeigen**, dass stets $w(B) = \mathit{true}$ bzw. $w(B) = \mathit{false}$ gilt. In diesem Fall sind beide Zweige weiter zu verfolgen. Der *true*-Ausgang wird weiter verfolgt, indem $A1$ symbolisch ausgeführt wird, wobei dabei und im folgenden die sogenannte **Pfadbedingung** $w(B) = \mathit{true}$ als gültig anzunehmen ist. Der *false*-Ausgang wird genauso behandelt, nur dass $A2$ ausgeführt wird und die Pfadbedingung $w(B) = \mathit{false}$ anzunehmen ist.

Wenn das Programm keine Schleifen besitzt, bricht das Verfahren nach endlich vielen Schritten und Verzweigungen ab, da es in diesem Fall nur endlich viele Wege gibt. Die Menge der Wege kann in Form eines endlichen Baumes, einem **Ausführungsbaum**, dargestellt werden.

Enthält das Programm jedoch Schleifen, die beliebig oft durchlaufen werden können, wie z.B. im Falle `for(i=1; i<n; i++)`, dann wird der Ausführungsbaum unendlich groß. Man muss sich also auf endlich viele Durchläufe beschränken.

Algorithmus 4 Die Funktion ABSOLUTE

```
procedure ABSOLUTE (x: integer): integer{1}
var x, y: integer;{2}
if x < 0{3} then
  y = −x;{4}
else
  y = x;{5}
end if
return y;{6}
```

Ein Beispiel gibt Algorithmus 4, dessen Ausführungsbaum in Abbildung 25 dargestellt ist. Die Nummern in den Knoten entsprechend dabei den Nummern in den Kommentaren im Quellcode. In Knoten 1 wird der Variable x der symbolische Wert a zugewiesen; y ist noch undefiniert. Der nächste wichtige Punkt ist Knoten 3, in welchem die Bedingung $x < 0$ abgefragt wird. Die Variable wird durch den symbolischen Wert ersetzt, womit $a < 0$ gefordert ist. Da nicht gezeigt werden kann, dass $a < 0$ immer *true* bzw. immer *false* ist, werden beide Ausgänge weiter verfolgt. In einem Fall wird angenommen, dass $a < 0$ *true* ist, im anderen Fall, dass $a < 0$ *false* ist. In letzterem Fall gilt dann logischerweise $a \geq 0$. Es werden nun entsprechend die Zuweisungen gemacht, wodurch y im Fall $a < 0$ den symbolischen Wert $-a$ erhält und im Fall $a \geq 0$ den Wert a.

Sequenz zweier Anweisungen: Punkt 1 und 2 haben das Vorgehen für einzelne Konstrukte erläutert. Ein normales Programm besteht jedoch aus einer Abfolge solcher Konstrukte, also z.B. aus einer Abfolge mehrerer Zuweisungen und Verzweigungen. Gegeben sei also eine Sequenz $S = S_1; S_2$.

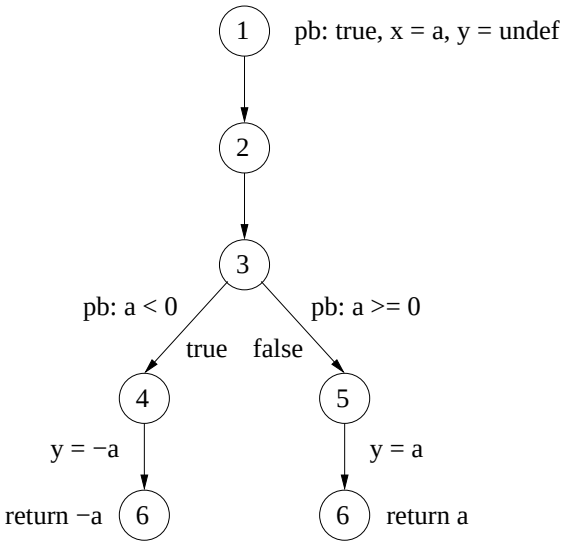


Abbildung 25: Symbolischer Ausführungsbaum für ABSOLUTE

Im Falle zweier Zuweisungen sind die symbolischen Werte der Variablen gemäß S_1 zu errechnen und dann entsprechend in S_2 zu benutzen. Im Prinzip war die Abfolge $C = A + 2 \cdot B;\ D = C - A$; in Punkt 1 schon ein Beispiel hierfür. Bleibt also nur noch die Frage, wie bei aufeinander folgenden Entscheidungen/Verzweigungen vorgegangen wird. Im Prinzip müssen in diesem Fall alle Pfadbedingungen gelten. Dies wird über ein logisches „und“ erreicht. Es sei also $p(S_1)$ die zu S_1 gehörende Pfadbedingung und $p(S_2)$ gehöre zu S_2. Dann ist $p(S_1) \wedge p(S_2)$ die zu S gehörende Pfadbedingung.

Operationsaufrufe: Es gibt zwei Möglichkeiten den Aufruf einer Operation, also einer Funktion oder Prozedur bei der symbolischen Berechnung zu behandeln:

1. Der **Aufruf wird durch eine symbolische Rechnung** gemäß der Parameterbelegung und der Anweisungen im Rumpf **ersetzt**. Dadurch können wieder neue Pfadbedingungen entstehen.
2. Beim Aufruf werden nur **die aktuellen symbolischen Werte der Parameter eingesetzt** und er wird **nicht weiter aufgelöst**.

Frage 9.30: Nenne die Vor- und Nachteile der symbolischen Ausführung!
Die symbolische Ausführung hat die folgenden **Vorteile**:

- Es ist **keine formale Programmspezifikation nötig**. Die berechneten Ausdrücke müssen lediglich betrachtet und überprüft werden.
- Ein symbolischer Test **deckt eine Vielzahl von normalen Testdaten ab**.

- Es werden insbesondere Fehler entdeckt, bei denen eine **Teilmenge der Eingaben** falsche Ergebnisse liefert.

Leider besitzt die symbolische Ausführung jedoch auch eine Menge **Nachteile**:

- Die **Überprüfung der Ergebnisse ist schwieriger als beim konventionellen Testen**, da symbolische Ausdrücke mit der Spezifikation verglichen werden müssen. Dies ist besonders der Fall, wenn unaufgelöste Operationsaufruf im Ergebnis – nicht jedoch in der Spezifikation – vorkommen.
- Die **verwendete Programmiersprache muss formal definiert sein**, damit der symbolische Interpreter arbeiten kann.
- Die Methode **benötigt mehr Voraussetzungen als das konventionelle Testen**, wie z.B. einen symbolischen Interpreter und eine Interpretation dessen Ergebnisse.
- Als **alleinige Methode ist die symbolische Ausführung nicht ausreichend**, da ein funktionsorientierter Test fehlt.
- Durch die Umformungen während der symbolischen Ausdrücke werden **Maschineneigenschaften nicht geeignet berücksichtigt**. Dazu zählen z.B. die Effekte der „Real-Arithmetik“.
- Es muss ein **Theorembeweiser** zur Verfügung stehen, der alle benötigten Umformungen durchführt. Für hinreichend mächtige Programmiersprachen kann es jedoch keinen vollständigen, automatischen Theorembeweiser geben, da ansonsten z.B, die Äquivalenz von Programmen nachweisbar wäre. Daher kann es bei einem verwendeten Theorembeweiser vorkommen, dass in einer Verzweigung beide Wege entlang gelaufen wird, obwohl nur einer ausfühbar ist.

Frage 9.31: Welche Konsequenzen haben die Fehlentscheidungen eines automatischen Theorembeweisers?
Es können zwei Konsequenzen unterschieden werden:

Erzeugung von Testdaten: Die symbolische Ausführung kann auch zur Erzeugung von Testdaten verwendet werden. Im Falle der Fehlentscheidung, dass ein eigentlich unausführbarer Weg ausführbar sei, ist es logischerweise unmöglich Testdaten zu finden, die diesen Weg ausführen.

Formale Verfikation: Bei der formalen Verifikation (siehe Kapitel 9.4) wird die symbolische Ausführung ebenfalls verwendet. Wird dabei ein unausführbarer Pfad weiterverfolgt, werden falsche Berechnungen vorgenommen, wodurch es passieren kann, dass das Ergebnis den Ausgabebedingungen (Zusicherungen) widerspricht. Obwohl das Programm also auf allen ausführbaren Pfaden korrekt arbeitet, kann dadurch entschieden werden, dass es nicht korrekt ist.

Frage 9.32: Warum wird beim Testen nicht immer nur mit symbolischen Werten gerechnet?

Auf den ersten Blick scheint es sinnvoller zu sein, immer mit symbolischen Werten zu rechnen und nicht mit konkreten Testdaten. Es gibt jedoch einige Punkte, die dagegen sprechen:

1. Bei der symbolischen Ausführung mit Werkzeugen verlässt man sich:
   a) auf die **korrekte Wiedergabe der Semantik der Programmiersprache** (z.B. Überläufe),
   b) auf die **korrekte Vereinfachung der symbolischen Ausdrücke** (Real-Arithmetik),
   c) auf die **korrekte Ermittlung der Werte von Bedingungen an Entscheidungen durch den Theorembeweiser**.
2. Die symbolische Ausführung **behandelt nur die funktionale Korrektheit** eines Programms. So sind z.B. **Leistungsmessungen** nicht möglich mit der symbolischen Ausführung.
3. Es ist oft **einfacher konkrete Ausgaben eines Programms mit der Spezifikation zu vergleichen**, da die symbolischen Ausgaben der symbolischen Ausführung oft **unzureichend vereinfacht** werden.

9.4 Formale Verifikation

Frage 9.33: Was ist der Zweck der formalen Verifikation? Welche Voraussetzungen müssen gegeben sein?

Zweck der formalen Verifikation ist es, automatisch zu überprüfen, ob ein Programm bzw. ein Programmteil das in der Spezifikation geforderte, leistet. Dazu wird, wie auch bei der symbolischen Ausführung, eine **klare, axiomatische Definition der Semantik der Konstrukte der Programmiersprache** benötigt. Damit automatisch verglichen werden kann, muss natürlich auch die Spezifikation des Programms in formaler Form vorliegen. Dies geschieht in der Form von **Zusicherungen**. Zusicherungen sind Formeln der Prädikatenlogik erster Stufe über den Variablen. Es können zwei Arten von Zusicherungen unterschieden werden:

Ausgabezusicherungen: Diese Zusicherungen geben das geforderte Resultat der Programmausführung an und haben die folgende Form:

PROVE (<Boolean>)

Eingabezusicherungen: Diese Zusicherungen formulieren einige Bedingungen, die eventuell an die Eingabevariablen gestellt werden und haben die Form:

ASSERT (<Boolean>)

Frage 9.34: Wie läuft die formale Verifikation ab?
Die formale Verifikation läuft in zwei Schritten ab:

Partielle Korrektheit: Im ersten Schritt wird die partielle Korrektheit des Programms gezeigt. Das heißt, dass nach jeder Beendigung des Programms die Ausgabezusicherung gilt, wenn die Eingabezusicherung gültig ist. Oder anders gesagt: Es ist **zu zeigen, dass die Ausgabezusicherung aus der Eingabezusicherung folgt**.

Totale Korrektheit: Es ist zu zeigen, dass das Programm **unter allen Umständen beendet wird**.

Frage 9.35: Wie wird bei Programmen ohne Schleifen vorgegangen?
Bei Programmen ohne Schleifen gibt es nur **endlich viele Wege** durch den Programmgraphen, die auch endlich lang sind. Daher kann wie folgt vorgegangen werden:

1. Für jeden Weg ist das Programm symbolisch auszuführen, wobei die Eingabezusicherung als anfängliche Pfadbedingung benutzt wird.
2. Aus dem berechneten symbolischen Ergebnis muss die Ausgabezusicherung folgen. Dies ist der eigentliche Schritt, der zu zeigen ist.

Frage 9.36: Wie wird bei Programmen mit Schleifen vorgegangen?
Bei Programmen mit Schleifen werden diese Schleifen „aufgeschnitten“ und fügt an diesen Schnitten sogenannte **induktive Zusicherungen** ein. Dadurch entsteht wieder ein Programm, in dem es nur endlich viele Wege im Kontrollflussgraphen gibt, die symbolisch ausgeführt wird. Diese Art der Ausführung heißt dann **Schnitt-symbolische Ausführung**. Auch bei dieser Art der Ausführung wird dann am Ende jedes Weges geschaut, ob die Ausgabezusicherung durch die Eingabezusicherung folgt.

Frage 9.37: Welche Vor- und Nachteile hat die formale Verifikation?
Bei der formalen Verifikation gibt es leider mehr Nachteile bzw. Probleme als Vorteile, wobei es sowohl theoretische als auch praktische Probleme gibt. Der **Vorteil** ist:

- Das **Programm wird für alle Eingaben** bezüglich der Ein- und Ausgabezusicherungen **als korrekt bewiesen**. Allerdings nur der Quellcode. Wenn der Semantik des Compilers anders aussieht als die des Verifizierers, kann es natürlich zu Unterschieden kommen.

Die **theoretischen Nachteile** sind:

- Die **Verifikationsbedingungen sind im allgemeinen nicht entscheidbar**. Falls die Bedingungen falsch ist (d.h. wenn das Programm falsch ist), hält ein korrekter Theorembeweiser eventuell nicht an.
- Die **induktiven Zusicherungen** können **nicht automatisch**, sondern nur durch Intuition ermittelt werden.

- Der totale Korrektheitsbeweis ist nicht immer möglich (Halteproblem).

Und dies sind die **praktischen Nachteile**:

- Der **Berechnungsaufwand** für den Theorembeweiser ist enorm.
- Die **Korrektheit eines automatischen Beweises setzt die Korrektheit des Theorembeweisers voraus**. Dieser sollte zumindest nichts falsches beweisen, wenn er schon manchmal „passen“ muss.
- Wenn das Programm falsch ist, wird der Beweis bezüglich der korrekten Zusicherungen stets fehlschlagen, was bei einem großen Programm fast immer der Fall sein dürfte.
- Desweiteren gibt es das **Problem der Fehlerbeseitigung**. Selbst wenn der verifizierer die Meldung „Programm nicht verifiziert“ ausgibt, weiß man noch lange nicht, wo der Fehler steckt. Und gute **Fehlerlokalisierer** sind auch nicht in Sicht.
- Ein als korrekt bewiesenes Programm kann dennoch falsche Resultate liefern. Dies kann mehrere Gründe haben:
  - Die **Semantik der Programmiersprache** wurde falsch modelliert.
  - Der Compiler, das Betriebssystem oder die Hardware können anders als angenommen arbeiten.
  - Die Eingabe- und die Ausgabezusicherungen in ihrer prädikatenlogischen Form können die eigentliche Anforderungsspezifikation des Auftraggebers falsch widergeben.

10 Testen „im Großen“

10.1 Probleme beim Testen großer Systeme

Frage 10.1: Welche Aspekte können zur Klassifikation von Testverfahren benutzt werden?

Mit den **bisher beschriebenen Methoden** wurde **nur die Funktion**, d.h. das **Eingabe- und Ausgabeverhalten** eines Programms getestet. Zur **Überprüfung der Qualität eines Systems** gehören aber noch **viele weitere Aspekte**. Diese können alle zur Klassifikation der Testverfahren benutzt werden. Im einzelnen sind dies:

Art des Prüfgegenstandes: **Große Systeme** bestehen nicht mehr lediglich aus Modulen sondern werden noch **stärker unterteilt**. Verschiedene **Module** bilden eine **Komponente**, diese wiederum **Teilsysteme** und diese schließlich **Systeme**. Alle diese Teile kommen als Prüfgegenstände in Frage, wobei der Test jeweils nach dem Prüfgegenstand benannt ist, also z.B. Komponententest.

Eine weitere Rolle spielt die **Stellung im Softwarelebenszyklus**. Bei Systemen, die in der Wartungsphase geändert werden, heißt der Test **Regressionstest** (im Sinne von „Rückbildung“ oder „Verschlechterung“), da durch Änderungen oft neue Fehler in das System eingebaut werden. Es ist also zu testen, ob so etwas passiert ist bzw. das der bisherige Fehler nicht mehr auftritt.

Merkmale der Software und das Ziel des Testens: Wie in Kapitel 3 schon erläutert wurde, besitzt Software verschiedene Merkmale, die alle zur Qualität des Produkts beitragen. Dies sind unter anderem:

1. **Funktionalität**:
   Ist die Funktionalität korrekt und vollständig? Dies wird im allgemeinen **dynamisch** getestet,
2. **Schnittstellen**:
   Werden an den Schnittstellen des betrachteten Prüfgegenstandes (Modul, Komponente) die richtigen Informationen übergeben? Dies wird vor allem **statisch** überprüft.
3. **Leistung**:
   Wie sieht das Zeitverhalten und der Speicherplatzbedarf aus? Auch Extremfälle sind zu testen (**Stresstest**).
4. **Mengengerüst**:
   Ist das System in der Lage die spezifizierte bzw. vermutlich anfallende Datenmenge zu verarbeiten? Was passiert, wenn die Menge leicht darüber liegt?
5. **Verfügbarkeit**:
   Das System sollte nicht zu oft abstürzen.

6. **Sicherheit**:
   Der lesende und insbesondere der schreibende Zugriff sollte nur authorisierten Personen erlaubt sein.
7. **Konfiguration**:
   Software, die für mehrere Kunden entwickelt wird, sollte in mehreren Varianten erstellt werden.
8. **Kompatibilität**:
   Neuere Versionen sollten auch mit älteren Komponenten Versionen bzw. Komponenten zusammenarbeiten.
9. **Benutzerfreundlichkeit**:
   Dazu zählt zum Beispiel die Aufgabenangemessenheit, die Selbsterklärungsfähigkeit, die Fehlerrobustheit, die Steuerbarkeit und die Erwartungskonformität.

Testpersonen: Nicht nur die Entwickler können die Software testen:

1. **Entwickler**: Führt Modul-, Integrations-, System- und Regressionstests durch.
2. **Potentielle Benutzer**: Diese führen Beta-Tests durch.
3. **Benutzer**: Führt Abnahme- und Betriebstests durch.

Umgebung des Prüfgegenstandes: Eine weitere Rolle spielt die Umgebung in der die Software getestet wird. Hier kann unterschieden werden zwischen:

1. **Rechner**:
   Ist der Rechner auf dem getestet wird der Entwicklungsrechner oder der Zielrechner. Nach einer Übertragung auf den Zielrechner ist ein **Installationstest** erforderlich.
2. **Anwendungsumgebung**:
   Kritische Software, von deren Funktion Menschenleben oder hohe Kosten abhängen, sollte erst in einer simulierten Umgebung getestet werden.

Testreferenz: Spezifikation und Benutzerhandbuch werden beim Systemtest als Testreferenz verwendet, Benutzerhandbuch und Benutzerwünsche werden beim Beta-Test und Abnahmetest referenziert.

Frage 10.2: Was ist ein monolithischer Test? Warum ist er bei großen Systemen ungünstig?

Bei kleineren Programmen ist es sinnvoll und möglich, das Programm als eine Einheit zu testen. Diese Art des Tests wird **monolithischer Test** genannt. Bei großen Systemen ist dieses Vorgehen jedoch nicht mehr so leicht möglich bzw. ungünstig. Dies sind die Gründe dafür:

- Es müssen **sehr viele Testdaten erzeugt werden**, um z.B. alle Zweige oder alle Wege (ohne Schleifen) in dem Programm auszuführen.

- Es ist **schwierig, fehlende Testdaten zu erzeugen**, da die Eingabedaten bis zum Erreichen eines Konstrukts eventuell einen langen Weg zurück legen müssen, auf dem sie eine lange Kette von Transformationen durchlaufen.
- Wenn ein Fehler auftritt, ist die **Lokalisierung des Fehlers** sehr schwer, da im Prinzip jede ausgeführte Anweisung für den Fehler verantwortlich sein kann und die Ausgabe-Wirkung einer Anweisung bis zur Ausgabe noch mehrfach transformiert wird.
- Fehler aus veschiedenen Modulen können sich **gegenseitig maskieren** und damit unerkannt bleiben.
- Parallelarbeit von mehreren Teams ist schwierig abzustimmen.

10.2 Modelle für große Systeme

Frage 10.3: Was ist die mikroskopische Sichtweise? Was ist die makroskopische Sichtweise?

Wie schon etwas weiter oben erwähnt, kann man sich bei kleinen Programmen eine **mikroskopische Sichtweise** leisten. Dazu werden Anweisungen und Daten betrachtet, die über den Kontroll- bzw. Datenfluss in Relation stehen. Bei großen Systemen geht dies allerdings nicht mehr, da die Komplexität enorm ansteigen würde.

Daher nimmt man bei großen Systemen eine **makroskopische Sichtweise** an. Dabei werden **Teile des Systems als eine Einheit mit ihren Relationen und Schnittstellen betrachtet**. Als Einheiten bieten sich zum Beispiel Operationen, Module oder noch größere Teile des Systems an.

Frage 10.4: Was ist ein Operationsaufrufgraph?

Ein Operationsgraph entsteht, wenn man **Operationen als Einheiten** betrachtet. Operationen stehen durch den **Aufruf anderer Operationen in Relation** und besitzen eine **Schnittstelle, die aus Parametern und globalen Variablen besteht**. Die Aufrufrelationen lassen sich dabei als ein Graph, dem sogenannten **Operationsaufrufgraph** darstellen, wobei die Operationen die Knoten sind und eine Kante von Operation P nach Q verläuft, wenn die Operation P die Operation Q aufruft. Der Graph kann aufgrund direkter oder indirekter Rekursionen **Zyklen** enthalten.

Frage 10.5: Was ist ein Modulgraph?

Ein Modulgraph entsteht, wenn **Module als Einheiten** betrachtet werden. Die Module stehen durch den **Aufruf von Operationen** aus anderen Modulen sowie der **Verwendung von Typdefinitionen** aus anderen Modulen in Relation. Beide Beziehungen werden als **Benutzt-Beziehung** aufgefasst. Die aufgerufenen Operationen sind jeweils Teil der **Export-** bzw. **Import-Schnittstelle**. Die Benutzt-Beziehung kann wieder als Graph dargestellt werden, dem sogenannten **Modulgraph**. Bei **ojektorientierten Systemen** zählt auch die **Vererbungsbeziehung** zur Benutzt-Beziehung.

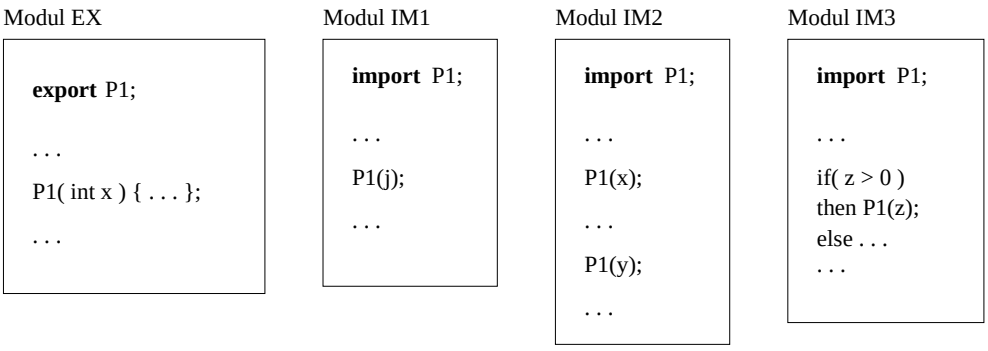


Abbildung 26: Programmcode von vier Modulen

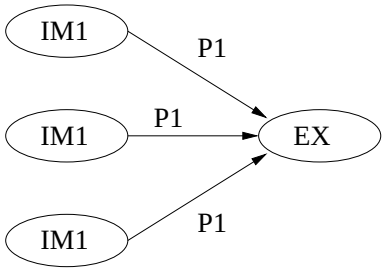


Abbildung 27: Modulgraph zur Struktur aus Abbildung 26

Frage 10.6: Welche zwei Arten von Modulhierarchien kennst du?
Allgemein sei erst einmal gesagt, dass davon ausgegangen wird, dass die **Benutzt-Relation der Module eine Hierarchie bildet**. Dies bedeutet, dass **keine Zyklen** bestehen und dass es **ein Modul an der Spitze der Hierarchie** gibt, welches von keinem anderen Modul benutzt wird.

Es kann dann zwischen zwei Arten von Modulhierarchien unterschieden werden. Entweder bildet die Hierarchie einen Baum, d.h. jedes Modul wird nur von einem anderen Modul benutzt, oder die Hierarchie bildet keinen Baum. In diesem Fall kann es auch passieren, dass es nicht möglich ist, die Hierarchie nicht in strengen Schichten anzuordnen.

10.3 Strategien für den Modul- und Integrationstest

Im folgenden wird zuerst beschrieben, wie einzelne Module getestet werden können und danach wie ein Integrationstest durchgeführt werden kann, wie also das Zusammenspiel der Module untereinander getestet werden kann.

10.3.1 Modultest

Frage 10.7: Was ist ein Modultest und worin besteht seine Aufgabe?
Ein **Modultest** ist das Testen eines einzelnen Moduls. Die Aufgabe des Modultests ist es, die realisierten Modulfunktionen und Datenbenutzungen mit der Modulspezifikation (welche als Ergebnis des Entwurfs vorliegt) zu vergleichen.

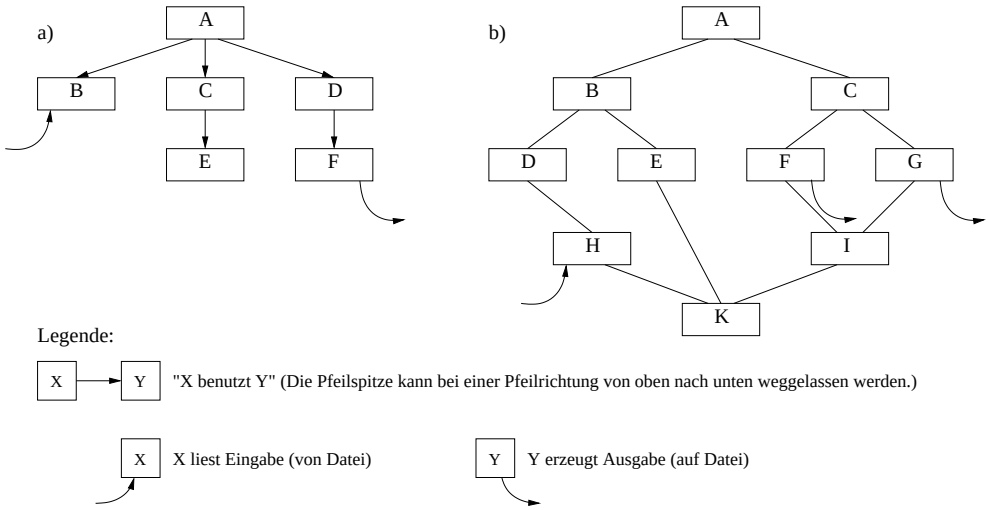


Abbildung 28: Verschiedene Modulhierarchien

Frage 10.8: Wie wird beim Modultest vorgegangen? Welche neuen Probleme bzw. Fragestellungen ergeben sich dabei?

Beim Modultest wird im Prinzip wie bei dem bisher beschriebenen Programmtest vorgegangen. Das heißt, es werden Testdaten nach den bisher beschriebenen Kriterien und Vorgehensweisen (implementationsorientiert, spezifikationsorientiert oder datenorientiert) erstellt und das Modul mit diesen getestet.

Dabei ergeben sich allerdings zwei neue Probleme. Wie schon etwas weiter oben gesehen, bestehen zwischen Modulen Benutzt-Beziehungen. Daher treten die folgenden Fragestellungen auf:

- Wie wird das Modul mit Eingabedaten versorgt? Wie wird es aktiviert?
- Was geschieht beim Aufruf bzw. bei der Benutzung von in der Benutzt-Hierarchie „tiefer liegenden“ Modulen?

Der ersten Punkt betrifft alle Module außer dem an der Spitze und der zweite Punkt betrifft alle Module außer die Module „am Ende“ der Hierarchie.

Gelöst werden diese beiden Problem durch das **Konzept der Treiber und Platzhalter**. Das erste Problem wird durch ein sogenanntes **Treibermodul** gelöst, welches das zu testende Modul **mit den entsprechenden Argumenten aufruft**. Diese Argumente können **Parameter, globale Variablen** oder **Werte für E-/A-Operationen** sein. Ein solcher Treiber kann unter Umständen auch automatisch erstellt werden.

Das zweite Problem wird durch sogenannte **Platzhalter** gelöst, welche die aufgerufenen **Module simulieren**. Diese müssen dem getesteten Modul natürlich hinzugefügt werden. Dabei kann die **Simulation verschieden komplex** sein:

- Der Platzhalter meldet nur, dass er aufgerufen wurde. Dies kann z.B. beim Testen einer GUI sinnvoll sein.

- Der Platzhalter liefert ein **konstantes Ergebnis**, welches im erlaubten Wertebereich liegt. Dies kann natürlich die Berechnungen im aufrufenden Modul verfälschen.
- Der Platzhalter liefert **korrekte Ausgaben für eine kleine Teilmenge** der erlaubten Eingaben, die beim Testen benötigt wird.

Bei letzter Methode müssen also zwei Dinge spezifiziert werden:

- Mit welchen Eingaben wird der Platzhalter aufgerufen?
- Welche Ausgabe soll der Platzhalter dafür abliefern?

Liegen diese Paare pro Testdatum vor, kann der Platzhalter automatisch erzeugt werden (wahrscheinlich fette *switch*-Anweisung in der Platzhalter-Funktion). Bei der Entscheidung, wie genau die Simulation eines Moduls sein soll, entsteht folgendes **Dilemma**:

- Wahl eines einfachen Platzhalters (1 und 2) ⇒ Das Modul kann falsche Ergebnisse liefern obwohl es eigentlich richtig arbeitet.
- Platzhalter wird genauer simuliert (Fall 3) ⇒ Bei entsprechender Überdeckung müssen viele Eingabe-/Ausgabepaare erzeugt werden, wobei in diesem Schritt sich wieder Fehler einschleichen können, insbesondere, wenn die Spezifikation ungenau ist.

Aufgrund des letzten Punktes wird ein Modul meist erst grob getestet und ein genauerer Test erst beim Integrationstest durchgeführt.

10.3.2 Zweck des Integrationstests

Frage 10.9: Worin besteht der Zweck des Integrationstests?
Beim Integrationstest wird quasi das Zusammenspiel der verschiedenen Module getestet. Er hat zwei Ziele:

1. Test der Schnittstellen zwischen den Modulen
2. Vollständiger Test des Moduls, welches in das bisherige System integriert wird, gemäß eines angestrebten Überdeckungsmaßes

Der zweite Zweck ergibt sich dabei aufgrund der etwas weiter oben angesprochenen Probleme, die sich bei der Platzhaltererstellung ergeben. Wie gut diese beiden Zwecke erfüllt werden können, hängt dabei von der Reihenfolge ab, in der die Module in das Gesamtsystem integriert werden können.

10.3.3 Nichtinkrementeller Integrationstest

Frage 10.10: Wie funktioniert der nichtmonolithische (nichtinkrementelle) Test?
Der nichtmonolithische Test besteht aus zwei Schritte:

1. Teste alle Module einzeln.
2. Teste danach das gesamte Programm.

Frage 10.11: Wieviele Treiber und Platzhalter müssen beim nichtmonolithischen Test erzeugt werden?
Wenn die Modulhierarchie aus insgesamt m Modulen besteht, müssen $\mathbf{m-1}$ **Treiber** erzeugt werden, nämlich für jedes Modul einer, außer für das Modul an der Spitze. Bei der Anzahl der Platzhalter muss zusätzlich die Art der Modulhierarchie unterschieden werden. Ist die Hierarchie ein Baum (wie in Abbildung 28a)), so müssen $\mathbf{m-1}$ **Platzhalter** erstellt werden. Denn jedes Modul, außer das an der Spitze, muss einmal (da es nur über eine Kante mit einem anderen Modul verbunden ist) durch einen Platzhalter ersetzt werden. Ist die Hierarchie kein Baum (siehe 28b)), so hängt die Anzahl der Platzhalter von der Summe der eingehenden Kanten aller Module ab (bzw. die Anzahl aller Kanten im Graphen). Ist also a_i die Anzahl der eingehenden Kanten des i-ten Moduls, beträgt die Anzahl der benötigten Platzhalter $\sum_{\mathbf{i=1}}^{\mathbf{m}} \mathbf{a_i}$.

Wenn also verschiedene Module ein Modul A benutzen, so müssen für A entsprechend viele Platzhalter erstellt werden (zumindest wenn Platzhalter-Strategie 3 verwendet wird), da A wahrscheinlich jeweils mit anderen Eingabedaten aufgerufen wird. Es könnte natürlich auch ein **„Super-Platzhalter“** erstellt werden, jedoch benötigt dies wiederum viel Kommunikation und Abstimmung, was einen der großen Vorteile (Parallelisierbarkeit) des nichtmonolitischen Tests wieder relativieren würde.

Frage 10.12: Was sind die Vorteile des nichtmonolithischen Tests?
Der nichtmonolithische Test besitzt die folgenden Vorteile:

1. Zu Beginn, während der einzelnen Modultests, gibt es Gelegenheit zu vielen **parallelen Aktivitäten**.
2. Da keine Teilweise Zusammenfassung von Modulen getestet wird (also jedes Modul nur einzeln), wird **weniger Testaufwand** benötigt als beim inkrementellen Testen.

Frage 10.13: Was sind die Nachteile des nichtmonolithischen Tests?
Die Nachteile sehen wie folgt aus:

1. **Fehler an den Schnittstellen** werden erst **spät erkannt**.
2. Die **Lokalisierung von Fehlern**, die erst beim **Test des gesamten Programms** auftreten, ist **schwierig**.

3. Der **Aufwand für die Treiber und Platzhalter** ist höher als beim inkrementellen Testen.

4. Auch hier ist ein **vollständiger Modultest** wegen des **Platzhalterproblems** schwer.

Der erstgenannte Nachteil kann durch einen **separaten Schnittstellentest** vermieden werden. Dabei werde für alle Paare von Modulen, zwischen denen eine Benutzt-Beziehung besteht, die Schnittstellen separat getestet. In Abbildung 28a) wären dies z.B. die Paare AB, AC, AD, CE und DF. Auch hier müssen Treiber und Platzhalter erzeugt werden. Für das Paar AB wären z.B. zwei Platzhalter für C und D zu erzeugen oder für das Paar CE ein Treiber, der die Rolle von A übernimmt.

10.3.4 Inkrementeller Integrationstest

Frage 10.14: Wie funktioniert das inkrementelle Testen?
Im Gegensatz zum nichtmonolithischen, nichtinkrementellen Testen werden beim inkrementellen Testen nicht alle Module parallel getestet und dann zum Programm „zusammengesteckt“. Stattdessen werden die Module nacheinander getestet und nach und nach ins System integriert. Die Vorgehensweise kann wie folgt beschrieben werden:

1. Teste zu Beginn irgendein Modul X. Dann besteht das „Teilsystem“ genau aus diesem Modul.

2. Füge ein Modul M hinzu, für welches mindestens eine der drei folgenden Bedingungen gilt:

   a) M benutzt keine anderen Module, *oder*

   b) wenn M andere Module benutzt, gehört mindestens eines von diesen zum bisherigen „Teilsystem“, *oder*

   c) wenn M von anderen Modulen benutzt wird, gehört mindestens eines von diesen zum bisherigen „Teilsystem“.

   Durch Hinzunahme des neuen Moduls M entsteht ein neues „Teilsystem“, welches nun als eine Einheit zu testen ist.

3. Wiederhole Schritt 2 mit einem weiteren, ungetesteten Modul bis das „Teilsystem“ das ganze Programm ist.

Frage 10.15: Was versteht man unter der Jo-Jo-Strategie?
Bei der Jo-Jo-Strategie werden die hinzuzufügenden Module so gewählt, dass abwechselnd in der Hierarchie der Module hinauf- und hinabgestiegen wird. Diese Strategie ist vorteilhaft, wenn ein Modul besonders gründlich getestet werden soll.

Frage 10.16: Welche Spezialfälle des inkrementellen Testens kennst du?
Es kann zwischen zwei Spezialfällen des inkrementellen Testens unterschieden werden:

- **Absteigender** (oder **top-down**) **Test**
- **Aufsteigender** (oder **bottom-up**) **Test**

Frage 10.17: Wie lautet die Strategie beim absteigenden Test?
Die Strategie des absteigenden Tests lautet wie folgt:

1. Beginne mit dem Modul an der Spitze der Hierarchie als Teilsystem.
2. Füge jeweils ein Modul, das nur von Modulen des bisherigen Teilsystem direkt benutzt wird, zum Teilsystem hinzu (bis das „Teilsystem“ das ganze System ist).

Frage 10.18: Wie lautet die Strategie beim aufsteigenden Test?
Die Strategie des absteigenden Tests lautet wie folgt:

1. Beginne mit einem Modul am „Boden“ der Hierarchie als Teilsystem. Also eines, welches kein anderes benutzt.
2. Füge jeweils ein Modul M zum Teilsystem hinzu, für das alle Module, die M direkt benutzt, zum bisherigen Teilsystem gehören (bis das „Teilsystem“ das ganze System ist).

Frage 10.19: Worauf sollte beim absteigenden Test geachtet werden?
Beim absteigenden Test sollte darauf geachtet werden, dass die **Module, die Eingaben verarbeiten bzw. Ausgaben produzieren, möglichst früh in das zu testende Teilsystem eingebunden werden**. Ansonsten müssen die Platzhalter diese Daten bzw. die „internen“ Repräsentation dieser Daten generieren. „Internen“ Repräsentationen deshalb, da die Daten auf dem Weg zwischen den Modulen teilweise transformiert werden und daher in jedem Modul unterschiedlich aussehen können. Ein Beispiel:

In Abbildung 28b) erzeugt das Modul H Eingaben. Soll nun Modul A getestet werden, so müssen für B und C Platzhalter erstellt werden. Wenn es in diesem Fall nun so ist, dass die Daten über A fließen, bevor sie in F und G ausgegeben werden, so muss der Platzhalter für B die Daten in der Form generieren, wie sie in B aussehen würden, und nicht so, wie sie tatsächlich in H in das System gelangen.

Daher sollte so vorgegangen werden, dass die neuen Module so gewählt werden, dass die Eingabe und Ausgabe möglichst früh zum Teilsystem gehört. Entsprechend kann auch bei **kritischen Modulen** vorgangen werden, von denen das Gesamtssystem entscheidend abhängt.

Frage 10.20: Was sind die Vorteile des inkrementellen Testens?
Das inkrementelle Testen besitzt viele Vorteile:

1. Der **Aufwand der Erstellung von Treibern und Platzhaltern ist geringer** als beim nichtinkrementellen Testen. Bei einer absteigenden Strategie werden keine Treiber benötigt und nur soviele Platzhalter wie beim nichtmonolithischen Testen. Bei der aufsteigenden Strategie werden keine Platzhalter benötigt und nur so viele Treiber wie beim nichtmonolithischen Testen.

2. **Schnittstellenfehler** werden bei jedem neuen „Dazubinden“ und somit **sehr früh entdeckt**.

3. Die **Fehlerlokalisierung ist einfach**, da sich ein Fehler sehr wahrscheinlich im neuen Modul bzw. an der Schnittstelle zwischen dem neuen Modul und dem Teilsystem befindet.

4. Die **zuerst ausgewählten Module** werden mit zusätzlichen Testdaten und somit sehr **gründlich getestet**.

Frage 10.21: Was sind die Nachteile des inkrementellen Testens?
Das inkrementelle Testen besitzt zwei Nachteile:

1. Der **Testaufwand ist höher** als beim nichtinkrementellen Testen.

2. Da das System nach und nach „zusammengesteckt“ wird, **kann nur sehr wenig parallel gearbeitet werden**.

Der ersten Nachteil ist dabei aufgrund der Vorteile 3 und 4 noch relativ leicht zu verschmerzen. Der zweiten Nachteil kann durch leicht modifzierte Ansätze abgemildert werden.

Frage 10.22: Was sind die Vorteile des absteigenden Testens?

1. **Fehler bzw. Probleme in den oberen Modulen** der Hierarchie, insbesondere Entwurfsprobleme, **werden frühzeitig und häufig entdeckt**.

2. Befinden sich die **Eingabe-/Ausgabefunktionen „oben“** in der Modulhierarchie, so ist eine **Darstellung der Testdaten** nach Integration dieser Module **einfacher**. In diesem Fall haben die Testdaten die Darstellung des späteren Systems.

3. Beim verzahnten absteigenden Codieren und Testen **steht zu jedem Zeitpunkt ein ausführbares Programmsekelett zur Verfügung** (weil die `main`-Funktion im ersten Modul ist und dort wahrscheinlich auch die anderen Subsysteme/Komponenten/Module instantiiert und initiailisert werden).

Frage 10.23: Was sind die Nachteile des absteigenden Testens?

1. Es müssen **(komplizierte) Platzhalter** erstellt werden (siehe oben).

2. Befinden sich die **Eingabe-/Ausgabefunktionen „unten“** in der Modulhierarchie, was bei einem guten Entwurf oft der Fall ist, so ist eine **Darstellung der Testdaten** vor der Integration dieser Module **schwierig**.

3. Die **Erzeugung von geeigneten Testdaten für den Modultest unterer Module kann unmöglich oder schwierig sein**. Unmöglich wird die Erzeugung z.B. wenn von einem Modul nur eine Teilmenge seiner Fähigkeiten benutzt wird (Beispiel: Löser für $ax^2 + bx + c = 0$) der nur zur Berechnung reellwertiger Lösungen benutzt wird). Schwierig wird die Erzeugung der Testdaten aufgrund der internen Repräsentationen von Ein- und Ausgabedaten.

4. Die **Beobachtung der Testausgabe** ist (vor dem Einbinden der Ausgabefunktionen) **schwieriger**.

5. Aufgrund der Nachteile 1 und 2 wird ein **vollständiger Test der „oberen“ Module bezüglich einer Überdeckung** (z.B. Zweigüberdekkung) **oft verschoben**.

Frage 10.24: Was sind die ambivalenten Eigenschaften des absteigenden Testens?

Mit ambivalenten Eigenschaften des absteigenden Testens sind Eigenschaften gemeint, die sowohl von Vorteil als auch von Nachteil sein können:

1. Beim absteigenden Testen können auch der Entwurf und die Programmierung (samit Test) verzahnt erfolgen. Dies kann jedoch auch Nachteile mit sich bringen. Wird in einem der unteren Module festgestellt, dass es vorteilhaft wäre, den Entwurf in einem der oberen Module zu ändern, so wird oft aufgrund des schon erbrachten Codier- und Testaufwand auf dem alten Entwurf beharrt. Wird dieser Entwurfsfehler in den unteren Modulen „ausgebügelt“, kann dies das spätere Verständnis und spätere Änderungen erschweren.

   Andererseits stehen bei diesem Vorgehen schon früh **Prototypen** zur Verfügung, mit denen Abweichungen bezüglich der Anforderungen bzw. zwischen Spezifikation und Entwurf frühzeitig festgestellt werden können. So kann ein Kunde zu jedem Zeitpunkt das System testen und so feststellen, ob es seinen Vorstellungen entspricht.

2. Module werden nur unter Bedingungen gestestet, unter denen sie auch im Gesamtsystem benutzt werden (Beispiel mit Löser, der Fehler bei komplexen Lösungen macht).

Frage 10.25: Was sind die Vorteile des aufsteigenden Testens?

Das aufsteigende Testen bietet die folgenden Vorteile:

1. Größere **Entwurfs- und Codier-Mängel bei den „unteren“ Modulen werden frühzeitig erkannt**.

2. **Testeingaben** für das neu hinzugekommene Modul sind **leicht zu erzeugen**, das dieses direkt vom Treiber aufgerufen wird (die Testeingaben werden also nicht mehr transformiert).

3. Das Überprüfen der Testergebnisse ist einfacher. (Weil in einem guten Entwurf die Eingabe-/Ausgabe-Funktionen unten sind?)

Frage 10.26: Was sind die Nachteile des aufsteigenden Testens?

1. **Treibermodule müssen erstellt werden**.

2. Bei einer Verzahnung von Codierung und Test **existiert ein Programmskelett erst, wenn das letzte Modul** (das an der Spitze der Hierarchie) **dazugebunden wird**.

3. Fehler und Probleme in den oberen Modulen werden erst spät aufgedeckt.

Frage 10.27: Welches Fazit kann für das inkrementelle Testen gezogen werden?

Am schlechtesten ist der monolithische Test, gefolgt vom nichtmonololithischen Test. Die inkrementellen Testmethoden besitzen jedoch auch gravierende Nachteile, weshalb eine Kombination der beiden Ansätze (auf- und absteigend) empfohlen wird (z.B. durch die Sandwich-Strategie).

11 Fehlerlokalisierung und -korrektur

Diese Kapitel beschäftigt sich mit den folgenden Fragen:

1. Welche Strategien gibt es allgemein zur Fehlerlokalisierung? Welches Wissen kann dabei von Hilfe sein?
2. Wie können Fehler im Programm lokalisiert werden?
3. Welche weiteren fehlerhaften Situationen gibt es?
4. Welche Prinzipien sollten bei der Fehlerlokalisierung gelten?
5. Welche Prinzipien sollten bei der Fehlerkorrektur gelten?
6. Was und wie sollte man aus gemachten Fehlern lernen, damit man sie nicht erneut macht?

11.1 Problemstellung, Lösungsstrategien und Wissensarten

Frage 11.1: Welche zwei Schritte müssen bei der Fehlerbehebung durchgeführt werden?
Die Fehlerbehebung ist ein zweiteiliger Prozess:

1. Bestimmung der exakten Natur des Fehlers und Lokalisierung im Programmtext (95%).
2. Korrektur des Fehlers (5%).

Frage 11.2: Welche drei Schritte sind bei genauerer Betrachtung zur Fehlerbehebung nötig?
Bei der Fehlerbehebung sind die folgenden drei Schritte nötig:

1. **Erkennen der Fehlersymptome** durch einen Soll/Ist-Vergleich zwischen implementiertem und spezifiziertem Programm.
2. **Auffinden der Fehlerursachen**, indem Funktion und Struktur des Programms betrachtet wird.
3. **Effizientes und korrektes Abändern des Programms**, wobei die Abänderungen ebenfalls zu überprüfen sind.

Frage 11.3: Warum wird die Fehlerbehebung ungern durchgeführt?
Gegenüber der Fehlerbehebung besteht aus den folgenden Gründen eine **psychologisch bedingte Abneigung**:

1. Das **Selbstwertgefühl** wird durch gefundene Fehler getroffen.
2. Fehler müssen unter **Zeitdruck** behoben werden. Dies kann sowohl an eng kalkulierten Lieferterminen liegen als auch an einem vom Programmierer selbst erzeugten Druck, den „leidigen“ Fehler schnell zu beheben.

3. Im Gegensatz zu anderen Ingenieursdisziplinen ist die Fehlerbehebung (und -lokalisierung) beim Programmieren **intellektuell sehr anstrengend**. Ein Beispiel:

Auto welches häufig stehen bleibt	$\Rightarrow$	Tacho bestimmt nicht Ursache.
Fehler in einem Computerprogramm	$\Rightarrow$	Der Fehler kann fast überall sein.

   Oder wie es Jaron Lanier ausdrücken würde: Die Relation zwischen Genotyp (dem Quellcode) und dem Phenotyp (das Programm, sein Verhalten) ist chaotisch.

Frage 11.4: Welche Wissensarten können bei der Fehlerbehebung hilfreich sein?
Die folgenden Wissensarten können bei der Fehlerbehebung hilfreich sein und sollten daher systematisch gesammelt und strukturiert werden:

W1) Wissen über das gewünschte (spezifizierte) Programm

W2) Wissen über das vorliegende Programm

W3) Verständnis der Programmiersprache

W4) Allgemeine Programmiererfahrung

W5) Wissen über den Anwendungsbereich

W6) Kenntnis der (üblichen) Fehler

Frage 11.5: Auf welches Aspekte eines Programms können sich die oben genannten Wissensaspekte beziehen?
Es gibt vier Aspekte:

1. Struktur des Programms
2. Verhalten des Programms
3. Funktion des Programms (Relation der Ausgaben zu den Eingaben)
4. Klassifikation des Verhaltens nach bestimmten Mustern (?)

Frage 11.6: Nach welchen Strategien (auch Kombinationen dieser) kann die Fehlerbehebung durchgeführt werden?
Es kann zwischen vier Strategien unterschieden werden, die auch teilweise kombiniert angewandt werden können:

Verifikationsstrategie: Bei der Verifikationsstrategie werden die beiden ersten **Wissensarten W1 und W2** benutzt. Dabei wird das vorliegenden Programm mit der Spezifikation des Programms bezüglich **Berechnungsäquivalenz verglichen**. Allerdings ist diese Strategie sehr **aufwändig**, da sie sehr viel Wissen über das Programm verlangt.

Strukturkontrollstrategie: Bei diesem Verfahren wird die **konsistente Verwendung von Programmkonstrukten**, wie z.B. Bezeichnern oder Typen überprüft.

Filterungsstrategie: Bei dieser Strategie wird versucht, die **Programmteile zu identifizieren**, die das fehlerhafte Verhalten verursacht haben können bzw. die Teile, die dafür nicht in Frage kommen.

Wird diese Strategie **zuerst eingesetzt**, kann bei **anderen Verfahren Aufwand eingespart** werden.

Strategie zur Erkennung stereotyper Fehler: Ziel dieser Strategie ist es, **bekannte Fehler** zu finden, bei denen ein **klarer Zusammenhang zwischen dem Fehlersymptom und der Fehlerursache** besteht.

11.2 Methoden der Fehlerlokalisierung

Frage 11.7: Welche Methoden zur Fehlerlokalisierung kennst du?
Es gibt unter anderem die folgenden Methoden:

1. Induktionsmethode
2. Deduktionsmethode
3. Fehlerlokalisierung durch Bestimmung der potentiellen Wege mit Fehlern
4. Fehlerlokalisierung durch Soll-/Ist-Vergleich der Berechnung auf einem Weg
5. Druckanweisungen einstreuen
6. Werkzeuge benutzen

Die ersten beiden Ansätze beruhen dabei auf bekannten Prizipien des **logischen Denkens**. Es wird im folgenden nun auf jede Methode einzeln eingegangen.

Frage 11.8: Wie funktioniert die Induktionsmethode?
Unter Induktion versteht man das **Schliessen vom Besonderen auf das Allgemeine**. Das „Besondere“ sind bei der Fehlerlokalisierung die Hinweise auf Fehlersymptome nach verschiedenen Testläufen und die Verbindungen zwischen den Symptomen sind das „Allgemeine“. Es werden also die besonderen Fälle rausgefischt und versucht auf die Gemeinsamkeiten zu schließen. Dies kann zum Beispiel hilfreich sein, wenn die Testdaten aufgrund dieser Gemeinsamkeiten alle einen bestimmten Weg im Kontrollflussgraph nehmen, auf dem dann dementsprechend mit großer Wahrscheinlichkeit der Fehler liegt.

Frage 11.9: Wie funktioniert die Deduktionsmethode?
Beim deduktiven Denken wird **vom Allgemeinen auf das Besondere geschlossen**. Das „Allgemeine“ sind im Falle der Fehlerlokalisierung allgemeine Hypothesen bzw. Theorien, was den Fehler verursacht haben könnte und das

„Besondere“ ist die Eliminierung der Hypothesen, die nicht zutreffen, sowie die Verfeinerung einer Hypothese, die wahrscheinlich zutrifft.

Frage 11.10: Welche Schritte umfasst die Induktionsmethode?
Die Induktionsmethode umfast fünf Schritte, die nun im einzelnen erläutert werden: Erläutere diese!

Schritt 1: Alle Symptome sind mit Hilfe der Testdaten zusammenzustellen. Dabei ist folgendes zu beachten:

- Günstig sind ähnliche, aber unterschiedliche Testdaten.
- Es sollte vorab geklärt werden, ob die Testdaten zulässig sind (innerhalb des Definitionsbereich, etc.).

Schritt 2: Die **Abweichung** zwischen Programmverhalten und der Spezifikation ist zu beschreiben und zu **klassifizieren**. Dazu kann eine **Symptomtabelle** mit acht Feldern dienen. Diese besteht aus vier Zeilen mit den Fragen „was“, „wann“, „wo“ und „welcher Umfang“ sowie den zwei Spalten „ist vorhanden“ und „ist nicht vorhanden“. Dabei gibt die „ist vorhanden“-Spalte an, was für ein Fehler wann (in welchen Situationen) wo (in welchem Modul, in welcher Ausgabe) auftritt. In welchem Umfang bedeutet z.B. ob der Fehler immer auftritt, manchmal oder gar zufällig. Die Spalte „ist nicht vorhanden“ entspricht dabei einer Filterstrategie und schließt z.B. bestimmte Module oder Testfälle von der näheren Betrachtung aus.

Schritt 3: Die **Beziehungen zwischen den Daten** sind zu **studieren**. Dies ist der wichtigste Schritt bei dieser Strategie. Ziel ist es, die Unterschiede zwischen der „ist vorhanden“- und „ist nicht vorhanden“-Spalte festzustellen. Können die Fälle, in denen ein Fehler auftritt, klar abgegrenzt werden von den Fällen, in denen er nicht auftritt, so kann meist auch die Fehlerursache eingegrenzt werden.

Schritt 4: Eine **Hypothese über die Fehlerursache ist aufzustellen**. In diesem Schritt ist die Erfahrung des Programmierers bzw. des Testers gefordert.

Schritt 5: Die **Hypothese ist zu beweisen**. Dies umfasst die folgenden beiden Schritte:

1. Es ist zu überprüfen, ob an der vermuteten Stelle im Programm wirklich ein Fehler exisitiert.
2. Es ist, z.B. durch einen manuellen Walkthrough mit einfachen Testdaten, zu verifizieren, ob der entdeckte Fehler alle Fehlersymptome erklären kann. Kann er dies nicht, wurde entweder nur ein Teil des Fehlers oder gar kein Fehler entdeckt.

Ein Beispiel für diese Strategie des Vorgehens ist im Buch [Riedemann 1997] auf Seite 423 gegeben (Programm welches für ungerade Anzahlen von Studenten den Median falsch ausrechnet).

Frage 11.11: Welche Schritte umfasst die Deduktionsmethode?
Die Deduktionsmethode umfasst ebenfalls fünf Schritte:

Schritt 1: Relevante **Testdaten zusammenstellen**.

Schritt 2: **Mögliche Ursachen** („Hypothesen“) für die fehlerhaften Testdaten **aufzählen**.

Schritt 3: **Ursachen eliminieren**. Bleibt in diesem Schritt keine Hypothese übrig, müssen neue Testdaten gesammelt werden. Bleiben mehrere Hypothese übrig so ist zuerst mit der wahrscheinlichsten fortzufahren.

Schritt 4: **Hypothese verfeinern**.

Schritt 5: **Verbleibende Hypothese beweisen**. Hierbei ist zu zeigen, dass sich die falschen und die korrekten Testdaten alle mit der Hypothese erklären lassen.

Frage 11.12: Welchem Vorgehen entsprechen die Induktions- und Deduktionsmethode beim Aufklären eines Mordfalls?
Das induktive Vorgehen entspricht der Fernsehserie „Columbo“:

1. Sammeln und Analysieren von Indizien.
2. Zusammensetzen anscheinend unbedeutender Einzelheiten, um Widersprüche und Ungereimtheiten aufzudecken.

Die deduktive Vorgehensweise entspricht dem Vorgehen Sherlock Holmes:

1. Eine Menge von Verdächtigen bestimmen.
2. Verdächtige ausscheiden (Alibi) und Verdachtsmomente verfeinern.
3. Schluss ziehen.

Frage 11.13: Was kann gemacht werden, wenn ein Fehler sich nicht mit der induktiven oder deduktiven Methode eingrenzen lässt?
In diesem Fall können die potentiellen Wege, die bei den fehlerhaften Ergebnissen ausgeführt werden, bestimmt werden. Dieses Vorgehen besteht aus zwei Schritten: Zuerst werden zu den fehlerhaften Testdaten $t_1, \ldots, t_n$ die ausgeführten Wege $w_1, \ldots, w_n$ sowie die Mengen der dazugehörigen Anweisungen $A_1, \ldots, A_n$ bestimmt. Es sind nun zwei Fälle möglich:

1. Es ist nur **eine Anweisung** für das Fehlverhalten verantwortlich. In diesem Fall liegt die fehlerhafte Anweisung a_f im Schnitt der Mengen aller Anweisungen $A_1, \ldots, A_n$: $a_f \in \bigcap_{i=1}^{n} A_i$
2. Es sind **mehrere Anweisungen** für das Fehlverhalten verantwortlich. Dann liegen diese Anweisungen in der Vereinigung aller A_i.

Allerdings ist die **Fehleranzahl** ja im allgemeinen **nicht von vornherein bekannt**. Aus diesem Grund muss meistens nach Fall zwei vorgegangen werden, wenn mit Fall eins keine Fehler gefunden werden können.

Im Buch befindet sich dazu ein Beispiel, bei dem das Programm zur Berechnung von $x^{|y|}$ (Abbildung 1(a)) modifiziert wird, so dass sich ein fehlerhaftes Verhalten ergibt. Betrachtet man nun die Wege, die bei korrekter und bei fehlerhafter Ausführung durchlaufen werden, so könnte man bei einer **optimistischen Strategie** davon ausgehen, dass nur die Knoten fehlerhaft sein können, die nicht in den korrekten jedoch in den fehlerhaften Wegen vorkommen. Diese Annahme ist jedoch falsch, da ein korrektes Ergebnis nicht bedeuten muss, dass keine fehlerhafte Anweisung ausgeführt wird. Im Beispiel wird für ein Testdatum gezeigt, dass es ein korrektes Ergebnis liefert, jedoch eine fehlerhafte Anweisung ausführt. Dabei wird der Fehler quasi durch eine Rundung „ausgebügelt“ bzw. maskiert.

Frage 11.14: Was ist ein „Slice“?

Bei der Betrachtung der fehlerhaften (endlichen) Wege können **bestimmte Anweisungen von vornherein als Fehlerquelle ausgeschlossen werden**, da sie mit der fehlerhaften Ausgabevariablen in keiner Relation stehen. Gegeben sei z.B. folgendes Programm:

```
read(i);
c:=i;
read(d);
if d=37 then
   a=c;
   b=2
else
   a=2;
end if
```

Angenommen statt $c = i$ müsste die Anweisung $c = i + 1$ lauten und der Fehler macht sich bemerkbar, dass statt $a = i + 1$ das Ergebnis $a = i$ herauskommt. Um diesen Fehler zu finden, sind mehrere Anweisungen unwichtig. Zum Beispiel die Anweisung im *else*-Zweig, die a den konstanten Wert 2 zuweist oder die Anweisung $b = 2$ im *then*-Zweig, die mit i, c oder a nichts zu tun hat. Also ist nur das folgende Slice relevant:

```
read(i);
c:=i;
read(d);
if d=37 then
   a=c;
else
   ...
end if
```

Frage 11.15: Angenommen ein fehlerhafter Weg wurde gefunden. Wie kann die fehlerhafte Anweisung aufgespürt werden?

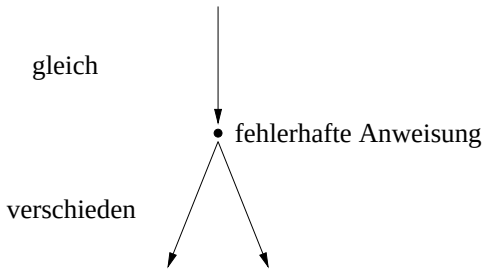


Abbildung 29: Soll-/Ist-Vergleich eines Programms

In diesem Fall kann der Weg mit dem fehlerhaften Testdatum durchgerechnet werden und in jedem Schritt wird der Ist-Programmzustand mit dem Soll-Programmzustand verglichen. Hier gibt es im groben zwei Vorgehensweisen:

1. Vorwärtsverfolgung des Weges
2. Rückwärtsverfolgung des Weges

Wenn ein Testdatum in eine unendliche Schleife läuft, ist natürlich nur eine Vorwärtsverfolgung möglich. Die Rückwärtsverfolgung bietet sich an, wenn der Fehler am Ende des Programms vermutet wird.

Bei der Vorwärtsverfolgung ist die fehlerhafte Anweisung gefunden, wenn der Ist-Zustand zum ersten Mal nicht mit dem Soll-Zustand übereinstimmt. Bei der Rückwärtsverfolgung ist er gefunden, wenn der Ist-Zustand zum ersten Mal mit dem Soll-Zustand übereinstimmt. Dies kann man sich relativ schnell an Abbildung 29 klar machen.

Frage 11.16: Welches Problem bzw. welche Fragestellung ergibt sich beim Vergleich des Soll- mit dem Ist-Zustand?

Es ergibt sich das Problem, **wie der Soll-Zustand mit dem Ist-Zustand verglichen werden soll** bzw. wie der Soll-Zustand ermittelt werden soll. Wenn die Spezifikation dies in jedem Detail beantworten könnte, so wäre sie ja fast schon die Implementierung.

Zumindest der Vergleich eines Soll-Zustandes mit einem Ist-Zustand lässt sich automatisieren. Die geschieht durch den Einbau von sogenannten **ausführbaren Zusicherungen** (executable assertions) in das Programm. Diese Zusicherungen haben die Form

ASSERT < Boolescher Audruck > Anweisung

So könnte z.B. angegeben werden, dass das Programm abbrechen soll, wenn in einem Array zum zweiten Mal nicht alle Einträge verschieden voneinander sind. Zusicherungen sollten z.B. an den folgenden Stellen bzw. in den folgenden Situationen verwendet werden:

1. Nach jeder **Initialisierung** von Variablen, Array oder sonstigen Datenstrukturen sollte eine Zusicherung eingebaut werden, die den (spezifizierten) Wertebereich abprüft.

2. Am Anfang und am Ende von **internen Prozeduren**.
3. Vor und nach **wichtigen Schleifen und Prozeduraufrufen**.
4. Vor möglichen **Singularitäten** (z.B. Division durch Null).
5. In wichtigen Schleifen sollten **induktive Zusicherungen** eingebaut werden.

Zusicherungen können **lebend** sein oder als Kommentare behandelt werden.

Frage 11.17: Wie sollte beim Einstreuen von Druckanweisungen vorgegangen werden? Was ist zu beachten?

Bei dieser Methode wird die Dynamik des Programm durch die Folge von ausgeführten Druckanweisungen protokolliert. Es werden also zusätzliche Ausgaben gemacht, die den aktuellen Werte von relevanten Variablen oder Datenstrukturen ausgeben. Dies ist insbesondere sinnvoll wenn keine normalen Programmausgaben vorliegen oder das Programm in eine Endlosschleife gerät. Dabei sollte jedoch aus den folgenden Gründen nicht zu unbedacht vorgegangen werden:

1. Es können dabei beträchtliche Datenmenge erzeugt werden.
2. Das Einfügen von Druckanweisungen erfordert eine Änderung des Programms, was dazu führen kann, dass Fehler maskiert, die Ausführungszeiten in Realzeitprogrammen verändern oder neue Fehler eingebaut werden.

Damit nicht allzu große Datenmenge anfallen, sollten die eingestreuten Druckanweisungen nach Möglichkeit nur die Stelle im Programm angeben bzw. nur die nötigsten Variablen ausdrucken. Eine solche Ausgabe wird im folgenden als ein **Schnappschuss** bezeichnet. Desweiteren sollten die Druckanweisungen an- und ausschaltbar sein bzw. die Menge der Informationen flexibel einstellbar sein (Debug-Level einstellbar).

Desweiteren kann die Ausgabemenge dadurch eingeschrönkt werden, dass Testdaten gewählt werden, die Schleifen nicht zu oft durchlaufen bzw. indem die Eingabe nach dem k-ten Durchlauf ausgeschaltet wird. Die Ausgabe sollte auch für Untermodule bzw. Unteroperationen ein- und ausschaltbar sein. Außerdem kann auch so vorgegangen werden, dass nicht alle Schnappschüsse eines Programms gespeichert werden, sondern nur die letzten k Schnappschüsse.

Zuletzt stellt sich noch die Frage, was alle protokolliert werden sollte:

1. Ein- und Ausgabesätze.
2. Die bei einer Verzweigung benutzten Abfragevariablen, damit die Programmlogik kontrolliert werden kann (z.B. in Schleifen oder vor einem *exit*).
3. In Berechnungen die Eingaben und Ausgaben (Ergebnisse), sowie bei langen Rechnungen die Zwischenergebnisse.

Frage 11.18: Welche Werkzeuge können die Fehlerbehebung unterstützen?
Ein Werkzeug zum Auffinden von Fehlern ist ein sogenannter „Debugger“. Bei den meisten Compilern kann angegeben werden, dass Debug-Informationen mit in die ausführbare Datei einkompiliert werden soll. Das Programm kann dann im Debugger ausgeführt werden, wobei im Source-Code (der mit einkompiliert ist) sogenannte **Breakpoints** gesetzt werden können oder der Debugger angewiesen werden kann, bestimmte Variablen zu überwachen (z.B. wenn sich diese ändern).

Frage 11.19: Welche weiteren fehlerhaften Situationen können Anlass zu einer Fehlersuche sein?
Hier können unter anderem vier Situationen betrachtet werden:

1. **Das Programm kompiliert nicht durch** bzw. der Compiler bricht ohne Fehlermeldung ab. In diesem Fall sollte das Programm so lange zerlegt werden, bis es durchkompiliert und dann nach und nach die verschiedenen Abschnitte wieder integrieren, bis es wieder nicht mehr kompiliert. Der letzte Abschnitt enthält dann wahrscheinlich den Fehler.

2. **Ausführung des Programms, keine Programmausgabe, evtl. Systemmeldungen**. „Mutanten“ schreiben, bei denen das Problem umgangen wird, bis es verschwunden ist oder behoben wurde.

3. **Vorzeitige Beendigung des Programmablaufs**. Vorgehen wie bis hier beschrieben und eventuelle Verwendung eines Core-Dump (wobei der Fehler allerdings noch vorher maskiert worden sein kann).

4. **Unendliche Schleife**. Vor und nach jeder Schleife eine Druckanweisung. Die letzte Anweisung „Anfang von Schleife x“ ist die fehlerhafte Schleife.

Frage 11.20: Welche Voraussetzungen und Prinzipien gibt es bei der Fehlerlokalisierung?
Folgende **Voraussetzungen** sollten nach Möglichkeit gegeben sein:

1. Das Programm ist zu **modularisieren**.

2. **Variablennamen** sollten eindeutig und beschreibend sein, d.h. die Codierung sollte den Test unterstützen.

Desweiteren können **Testhilfen** verwendet werden, die das Testen erleichtern oder unterstützen. Dies sind:

1. **Liste von Variablen und Konstanten**, z.B. zur Erkennung von Tippfehlern.

2. **Diagnosehilfen**, also Zusicherungen oder Druckanweisungen, **sind immer in das Programm einzubauen**, also nicht erst im Ernstfall.

3. **Diagnose-Anweisungen** sollte **leicht unterscheidbar** sein und bei der Kompilierung in Kommentare umgewandelt werden können.

Die **Vorgehensweise** bei der Lokalisierung von Fehlern sollte den folgenden Prinzipien folgen:

1. Probleme isolieren und **eines nach dem anderen lösen**.
2. Ein guter Diagnostiker sollte Fehler ohne Rechnerhilfe lokalisieren können.
3. Wenn man in eine **gedankliche Sackgasse** gerät, das Problem überschlafen oder einem Kollegen schildern.

Verwaltungtechnisch sollte so vorgegangen werden, dass die Ausgaben verschiedener Testläufe nach Datum, Uhrzeit und Version zu organisieren sind und dass unnötige Testausgaben wegzuwerfen sind.

Frage 11.21: Welche Prinzipien sind bei der Fehlerkorrektur zu beachten?
Die folgenden Prizipien sollten bei der Fehlerkorrektur beachtet werden. Zuerst einmal ist es wichtig, dass **nicht nur das Symptom eines Fehlers beseitigt** wird, sondern die Ursache, die das Symptom verursacht. Ist ein Fehler gefunden, sollte auch seine **Umgebung untersucht** werden, da Fehler oft gehäuft auftreten. Nach der Korrektur des Fehlers solle das Programm noch gründlicher als vorher getestet werden, da sich durch eine Korrektur oft neue Fehler einschleichen, und die Wahrscheinlichkeit, dass die Korrektur richtig ist, mit der Größe des Programms immer mehr sinkt. Aus diesen Gründen sollten bei großen Programmen **Regressionstests** durchgeführt werden. Da die Fehlerkorrektur **wie ein Programmentwurf** zu behandeln ist, sollten auch die beim Entwurf angewandten Prinzipien Anwendung finden. Zuguterletzt sollte ein Fehler natürlich immer **nur im Quellcode** und nicht im Objektcode beseitigt werden, da er nach einer erneuten Kompilierung sonst wieder auftritt.

Frage 11.22: Warum sollte eine Fehleranalyse durchgeführt werden?
Eine Fehleranalyse, deren Ziel es ist, die eigentliche Ursache des Fehlers herauszufinden, sollte durchgeführt werden, da es unökonomisch ist, immer wieder dieselben Fehler zu machen, die immer wieder lokalisiert und korrigiert werden müssen. Dadurch kann der ganze Entwicklungsprozess verbessert werden (**totales Qualitätsmanagement** - es wird die Qualität des Produkts als auch des Produktionsprozess verbessert), wodurch er einen höheren Reifegrad erreicht (**Reifegradmodell**). Es sind sich zu jedem Fehler also die folgenden Fragen zu stellen:

1. In welcher Entwicklungsphase wurde der eigentliche Fehler gemacht?
2. Durch welche mangelnden Kenntnisse, Fähigkeiten oder organisatorischen Regelungen wurde der Fehler verursacht.
3. Warum wurde der Fehler nicht eher entdeckt? Wie hätte er eher entdeckt werden können?
4. Wie wurde der Fehler entdeckt? (Positive Rückmeldung nach dem Motto „Weiter so!“)

12 Management des Testens und Prüfens

12.1 Komplexitätsmaße für die Aufwandsermittlung

Frage 12.1: Welche Komplexitätsmaße kennst du, um den Aufwand des Testens abzuschätzen?
Hier können zwei Methoden unterschieden werden:

- Quelltextlänge bzw. Anzahl der Zeilen (LOC - Lines Of Code)
- Zyklomatische Zahl $v(G) = e - n + 2$

Frage 12.2: Wie wird die Quelltextlänge gemessen? Was sind ihre Vorteile? Was ihre Nachteile?
Als **Quelltextlänge** wird die **Anzahl der Zeilen im Quelltext** bezeichnet, wobei Kommentare entweder mitgezählt werden können oder auch nicht. Ein Argument für das Mitzählen von Kommentaren wäre wahrscheinlich, dass Programme, die viel Kommentar benötigen, wahrscheinlich relativ komplex und daher schwerer zu verstehen sind. Das Verfahren bietet die folgenden Vorteile:

- Es ist **automatisch messbar**.
- Das Maß korreliert mit dem Aufwand zur Erstellung bzw. zum Verstehen des Programms.
- Es ist auf **alle Programmiersprachen** anwendbar.

Es hat jedoch auch zwei Nachteile:

- Es ist erst **nach der Implementierung** anwendbar.
- Es berücksichtigt keine **anderen Faktoren** (z.B. ob ein Programm **sequentiell oder nebenläufig** ist).

Frage 12.3: Wie ist die zyklomatische Zahl definiert? Was drückt sie aus? Welche Vorteile besitzt sie? Welche Nachteile?
Die **zyklomatische Zahl** ist für einen zusammenhängenden Kontrollflussgraph G definiert als:

$$v(G) = e - n + 2$$

Dabei ist e die Anzahl der Kanten des Graphen, also $|E|$, und n ist die Anzahl der Knoten ($|V|$). Es gelten dann die folgenden Beziehungen:

1. Wenn G der Graph eines strukturierten Programms ist, dann kann über die zyklomatische Zahl die **Anzahl der Verzweigungsknoten** ausgerechnet werden. Ist p die Anzahl der Verzweigungen, dann gilt folgender Zusammenhang:

   $$v(G) = p + 1$$

   Somit berechnet sich die Anzahl der Verzweigungen als $p = v(G) - 1$.

2. Ist der Kontrollflussgraph planar, dann beträgt die **Anzahl der Gebiete** genau $v(G)$.

3. Die Anzahl der **fundamentalen Wege** im Kontrollflussgraphen beträgt ebenfalls $v(G)$.

Die zyklomatische Zahl bietet folgende Vorteile:

- Sie ist **automatisch messbar**.
- Sie korreliert mit dem Erstellungs- und Verstehensaufwand. Insbesondere korreliert sie mit dem Testaufwand des Kriteriums **alle fundamentalen Wege**.
- Wenn beim Feinentwurf Pseudocode exisitert, kann sie schon in dieser Phase benutzt werden.

Jedoch besitzt auch die zyklomatische Zahl einige Nachteile:

- Sie ist **erst nach dem Algorithmenentwurf** einsetzbar.
- Sie misst nur die Anzahl der Verzweigungsknoten und **berücksichtigt nicht unterschiedliche Schachtelungen** von Verzweigungen **bzw. unterschiedlich komplexe Verzweigungsprädikate**.

12.2 Auswahl von Softwareprüfmethoden

Frage 12.4: Warum müssen bestimmte Prüfmethoden ausgewählt werden? Was ist das Problem dabei?

Allgemein möchte man natürlich eine Prüfmethode wählen, die den höchsten Prozentsatz an Fehlern mit möglichst wenig Aufwand findet. Da es aber kein Verfahren gibt, welches alle Fehler findet, müssen mehrere der bisher vorgestellte Verfahren kombiniert werden. Aus Kostengründen und in Hinblick auf die angestrebte Qualität können jedoch nicht alle Verfahren benutzt werden. Außerdem kommen in den meisten Fällen auch nicht alle Fehlerarten in einem Software-Produkt vor, sondern nur eine Teilmenge von diesen. Welche Fehlerarten dies sind, **ist aber logischerweise vor dem Testen nicht bekannt**.

Es wäre also schön wenn es Kriterien geben würde, mit Hilfe derer man aufgrund der vorliegenden Spezifikation bzw. des vorliegenden Programms (Quellcode) entscheiden könnte, welche Prüfmethoden verwendet werden sollten. Unter anderem hängt die Auswahl natürlich auch von dem zur Spezifikation bzw. zur Programmierung verwendeten Paradigma ab.

Es werden im folgenden nun **Maßzahlen** vorgestellt, die eine Auswahl aus 42 Prüfmethoden erlauben. Die **Grundidee** dieser Maßzahlen ist es, dass **die überdurchschnittlich komplexen Programmkomponenten die potentiell fehlerhaften Komponenten** sind. Die Komplexitätsmaße beziehen sich dabei auf spezielle Programmstrukturen, wie z.B. Schleifen, Entscheidungen, etc. Dabei wird ähnliche wie bei der Testwirksamkeitsmessung vorgegangen,

d.h. es wird die Anzahl bestimmter Konstrukte der Anzahl wieder anderer oder übergeordneter Konstrukte gegenüber gestellt.

Frage 12.5: Welche Arten von Maßzahlen kennst du?
Es können grob vier Arten von Maßzahlen unterschieden werden, wobei die Konstrukte, an denen sich die Maße orientieren, alle schon in den Kapiteln zum implementationsorientierten Testen aufgetaucht sind. Dies sind:

Kontrollflussmaße: Die Kontrollflussmaße versuchen zu charakterisieren, **welche Rolle der Kontrollfluss in einem Programm spielt**. In einem Programm, in dem es keine einzige Entscheidung gibt, welches also nur linear vom Anfang bis zum Ende abläuft, spielt der Kontrollfluss z.B. logischerweise keine wichtige Rolle und Kriterien wie z.B. die Mehrfachbedingungsüberdeckung sollten dann in diesem Fall keine Verwendung finden. Einige Beispiele für Kontrollflussmaße (aus dem Buch):

$$\begin{aligned} Z_2 &= \text{Anzahl der Entscheidungen } (= v(G)) \\ Z_3 &= \frac{\text{Anzahl der Entscheidungen}}{\text{Anzahl der Anweisungen}} \end{aligned}$$

Grob existieren die folgenden Komplexitätsmaße:

- Komplexitätstyp Entscheidungen (siehe Beispiel)
- Komplexitätstyp Entscheidungen/Anweisungen
- Struktur von Entscheidungen
- Struktur von Schleifen

Datenflussmaße: Die Datenflussmaße unterstützen die Entscheidung, ob und inwiefern **datenflussbezogene Testkriterien** herangezogen werden sollten. Wenn man sich z.B. zwischen einem kontroll- und einem datenflussbezogenen Test entscheiden möchte, kann folgende Maßzahl hilfreich sein:

$$F_1 = \frac{\text{Anzahl der Definitionen, B- und E-Referenzen}}{\text{Anzahl der Entscheidungen}}$$

Weiterhin gibt es Maßzahlen, welche die unterschiedlichen Formen des Datenzugriffs (z.B. E- oder B-Referenz) charakterisieren oder die die Länge von Datenflussketten einbeziehen. Kurz aufgezählt:

- Komplexitätstyp Datenzugriffe/Anweisungen
- Unterschiedliche Formen des Datenzugriffs
- Länge von Datenflussketten
- Datenzugriffe auf bestimmte Datenstrukturen

Datenstrukturmaße: Datenstrukturmaße können dazu dienen, festzustellen, inwiefern symbolische Ausführungen, formale Verifikation oder datenflussbezogene Testkriterien Anwendung finden können, da die genannten Verfahren teilweise Probleme mit Arrays, Zeiger und Dateien haben. Hier kann unterschieden werden zwischen:

- Komplexitätstyp Datenstrukturen
- Komplexitätstyp reelle Datentypen

Arithmetikmaße: Diese beiden Maße geben an, ob es ausreicht, kontroll- bzw. entscheidungsbezogene Testverfahren zu benutzen, oder ob berechnungsbezogene Verfahren (siehe Kapitel 6.1) benutzt werden sollten.

Frage 12.6: Inwiefern kann die Testmethodenauswahl automatisiert werden?

Die Testmethodenauswahl kann in Form eines **Expertensystem** automatisiert werden. Dieses geht von einem **Komplexitätsprofil** der zu prüfenden Software aus, welches aus den eben teilweise vorgestellten Maßen besteht. Das System kann dann eine Fehlerprognose für die Software ausgeben, sowie eine Kombination von Prüfverfahren generieren, die alle prognostizierten Fehler abdecken, möglichst redundanzfrei sind und allgemein akzeptierte Anforderungen an Prüfungen (z.B. C_1-Überdeckung) erfüllen.

Frage 12.7: Wie können Softwareprüfmethoden kombiniert werden?

Softwareprüfmethoden können entweder **lose** oder **streng gekoppelt** werden. Bei der losen Kopplung werden verschiedene Prüfverfahren unabhängig voneinander angewandt, d.h. entweder sequentiell oder parallel. Es empfiehlt sich **in jeder Entwicklungsphase** z.B. die folgende Reihenfolge:

1. Informelle Prüfung
2. Statische Analyse und symbolische Ausführung
3. Spezifikationsorientiertes Testen
4. Implementationsorientiertes Testen
5. Überprüfung von Zusicherungen

Es sollte **in jedem Fall ein gutes Werkzeug zur statischen Analyse verwendet werden**, da mit solchen Werkzeugen oft Fehler entdeckt werden können, ohne dass dazu Testdaten erzeugt werden müssen.

Eine **strenge** Kopplung ist bei Verfahren möglich, die sich vordergründig zwar auf verschiedene Aspekte beziehen, jedoch bei geeigneter Abstraktion vergleichbar sind.

Frage 12.8: Welche Möglichkeiten gibt es, die Fehleranzahl abzuschätzen?

Zur Abschätzung der Fehleranzahl können zwei Methoden benutzt werden:

1. Fehlereinpflanzung (error seeding)
2. Testen durch zwei unabhängige Gruppen

Frage 12.9: Wie kann die Fehleranzahl mit Hilfe der Fehlereinpflanzung abgeschätzt werden? Welche Vorteile hat das Verfahren? Welche Nachteile?
Bei der Methode der Fehlereinpflanzung **wird davon ausgegangen, dass echte und extra eingepflanzte Fehler in einem Programm mit der gleichen Rate gefunden werden**. Das Vorgehen ist dann wie folgt:

1. Von Personen, die die Tests nicht kennen, oder von Werkzeugen, die unabhängig von der Testdatengenerierung arbeiten, werden Fehler in das Programm P eingebaut, wodurch das Programm P' entsteht.
2. Das Programm P' wird mit der gleichen Menge von Tests T getestet wie das Programm P.
3. Die Menge der beim Test mit T gefundenen eingepflanzten Fehler wird bestimmt. Ebenso ist die Menge der bisher in P gefundenen Fehler bekannt.
4. Es wird davon ausgegangen dass, dass die Rate der gefundenen Fehlern gleich ist, also gilt:
$$\frac{GF}{F} = \frac{GE}{E}$$
Dabei steht GF für die Anzahl der gefundenen echten Fehler, F für die Anzahl echter Fehler, GE für die Anzahl gefundener, eingepflanzter Fehler und E für die Anzahl der eingepflanzten Fehler. Damit ergibt sich die Anzahl der Fehler also als:
$$F = E \cdot \frac{GF}{GE}$$

Das Verfahren besitzt die folgenden **Vorteile**:

- Der Testaufwand ist gering, da nur *ein* Programm mit mehreren Fehlern zu testen ist und nicht, wie z.B. bei der Mutationsanalyse mehrere Programme mit einem Fehler.
- Man erhält eine Abschätzung der echten Fehler.

Die **Nachteile** sehen wie folgt aus:

- Die Ermittlung, welcher eingepflanzter Fehler das Fehlverhalten ausgelöst hat, ist nicht immer möglich, da evtl. mehrere Fehler auf einmal ein Fehlverhalten bewirken können.
- Die Gültigkeit der Gleichung $\frac{GF}{F} = \frac{GE}{E}$ ist problematisch, da dabei einige Annahmen gemacht werden, die bei Softwareprodukten meist nicht gegeben sind. Zum Beispiel dass es eine **sehr große Anzahl von Fehlern** im Programm gibt, oder dass die **Verteilung** der eingeplanzten Fehlern der Verteilung der echten Fehlern entspricht. Außerdem wird davon ausgegangen, dass die **Wahrscheinlichkeit** einen eingepflanzten Fehler im Programm zu finden, genau so hoch ist, wie die Wahrscheinlichkeit, dass ein echter Fehler gefunden wird. Dabei sind echte Fehler wahrscheinlich viel subtiler. Desweiteren darf es bei diesem Ansatz nicht zu einer **Wechselwirkung** zwischen echten und eingepflanzten Fehlern kommen.

Frage 12.10: Wie kann die Fehleranzahl mit Hilfe von zwei unabhängigen Testgruppen ermittelt werden?

Es sind die folgenden Schritte anzuwenden:

1. Zwei unabhängige Gruppen G_1 und G_2 ermitteln jeweils Testdatenmengen T_1 und T_2 für das Programm P.

2. F_1 ist die Anzahl der Fehler, die durch Gruppe G_1 entdeckt wurden, F_2 die Menge die von G_2 entdeckt wurden und $F_{1\cap 2}$ die Menge von Fehlern, die von beiden Gruppen entdeckt wurde.

3. Wird von einer konstante Testeffizienz der Gruppen angenommen gelten die folgenden Verhältnisse:

$$\begin{aligned} \frac{F_1}{F} &= \frac{F_{1\cap 2}}{F_2} = \text{Testeffizienz von Gruppe } G_1 \\ \frac{F_2}{F} &= \frac{F_{1\cap 2}}{F_1} = \text{Testeffizienz von Gruppe } G_2 \end{aligned}$$

4. Die Gleichung aus 3 kann nun nach F umgestellt werden:

$$F = \frac{F_1 \cdot F_2}{F_{1\cap 2}}$$

5. Nun wird die Zahl der nicht gefundenen Fehler ermittelt. Dies ist die Anzahl aller Fehler ohne die Fehler die insgesamt von G_1 und G_2 ermittelt wurden:

$$NF = F - F_{1\cup 2} = F - (F_1 + F_2 - F_{1\cap 2})$$

   In dieser Formel wird nun F durch obiges Ergebnis ersetzt, so dass nur noch bekannte Größen in der Formel für NF sind:

$$NF = \frac{F_1 \cdot F_2}{F_{1\cap 2}} - (F_1 + F_2 - F_{1\cap 2}) = \frac{(F_1 - F_{1\cap 2}) \cdot (F_2 - F_{1\cap 2})}{F_{1\cap 2}}$$

13 Reviews in der objekt-orientierten Softwareentwicklung

Dieses Kapitel widmet sich dem Artikel *„Reviews in der objekt-orientierten Softwareentwicklung“* [Winter 1997]. Es wird zuerst eine Einleitung gegeben, in der auf die Probleme eingegangen wird, auf die herkömmliche Reviewmethoden im Zusammenhang mit objekt-orientierten Software-Systemen stoßen. Anschließend wird erläutert, **was ein Review ist** und **einige Arten des Reviews kurz vorgestellt**. Nachdem dies geschehen ist, wird auf die Besonderheiten objekt-orientierter Reviews eingegangen. Dabei werden jeweils Vorgehensweisen für das Review der verwendeten **Architektur**, der **statischen Strukturen** und der **Dynamik** des Software-Produkts erläutert. Danach wird auf die **Codeinspektion** näher eingegangen, wobei Aspekte wie **Wiederverwendbarkeit**, **Wartbarkeit**, **Portabilität** und **Effizienz** näher untersucht werden.

13.1 Einleitung

- Programme werden in immer mehr Bereichen des Lebens zur einzig kontrollierenden Instanz ⇒ Hohe Qualität der Software wichtiger denn je.
- Zur Qualitätssicherung haben sich die Reviews, insbesondere die **Inspektion nach Fagan**, als wirkungsvolle Mittel erwiesen.
- Zwei Gründe für Schwierigkeiten herkömmlicher Inspektionstechniken bei objekt-orientierter Software:
  - Komplexität wird von einzelnen Klassen auf deren Beziehungsgeflecht „verschmiert“
  - Struktur zerfasert durch weitere Freiheitsgrade, wie z.B. Polymorphismus

13.2 Reviews

- Reviews gehören zu den **manuellen, statischen Verfahren**.
- Aus der freiwilligen, unstrukturierten Betrachtung von Code (durch den Programmierer) wird ein präzise definierter Vorgang, bei dem der zeitliche und inhaltliche Ablauf festgelegt wird.
- **Ziel des Reviews**: Defekte im Entwicklungsprodukt aufzeigen. Dabei ist ein **Defekt** die **zu erwartende Abwesenheit eines Qualitätsmerkmals** wie z.B. Korrektheit, Benutzerfreundlichkeit, Robustheit, Wartbarkeit oder Wiederverwendbarkeit.
- Am weitesten verbreitet sind: **persönliches Review**, **Walkthrough** und **Inspektion**.

- **Persönliche Reviews** werden im Laufe der Entwicklung durchgeführt. Dabei kann entweder ein Programmstück per Hand durchgespielt werden oder der Code anhand persönlicher Check- und Fehlerlisten geprüft werden. Die Mitarbeiter können ihren Code auch gegenseitig prüfen.
- **Walkthrough** und **Inspektion** wurden schon in Kapitel 9.1 behandelt.
- Verbessung von Parnas: Unterschiedlich qualifizierte Reviewer (Benutzer, Entwickler, etc.) führen Reviews durch.
- **Phased inspections**:
  - Review in **mehreren aufeinander folgenden Phasen**
  - Gefundene Defekte werden nach jeder Phase sofort beseitigt
  - Es werden einige, wenige Aspekte des gesamten Produktes gegen einige Kriterien getestet, anstatt wie beim der „normalen“ Inspektion ein Teil des Produktes gegen viele Kriterien.
  - Zuerst **Einzelinspektorphasen** zur Suche einfacher, aber relativ wichtiger Defekte (Programmierrichtlinien, etc.) und anschließend **Multiinspektorphasen** zur Suche aller anderen Defekte.
- Gewonnene Daten werden nicht nur zum Hauptzweck des Prüfens des zu inspizierenden Dokuments genutzt, sondern **auch zur Verbesserung der Prüfmethoden und des Entwicklungsprozess**.

13.3 Objekt-orientierte Reviews

- Reviews sollten auf jeder Granularitätsstufe (Gesamtmodell, Teilsysteme/Frameworks, Cluster, Klassen) in jeder Entwicklungsphase erfolgen.
- Prinzipiell können getrennte Reviews bezüglich der Vererbungsbeziehung und der Objektbeziehungen (Konstruktions-/Behälterbeziehungen und Benutzungsbeziehungen) durchgeführt werden.
- Empfohlen wird ein Review der gesamten Architektur, da erst das Zusammenspiel von vererbten und redefinierten Eigenschaften die „Essenz des Entwurfs“ ausmacht.

13.3.1 Architektur

- Entwürfe lassen sich oft grob wohlbekannten **Architektur**-Mustern zuordnen. Beispiele hierfür wären die 3-Schichtenstruktur oder auch die Client-/Serverarchitektur.
- Die Architektur eines Teilsystems ist oft durch ein **Framework** vorgegeben (z.B. verwendete GUI-Bibliothek).
- Innerhalb von Teilsystemen finden sich oft bestimmte **Entwurfsmuster**.
- Es sollten im Rahmen eines Reviews dieser Muster also folgende Fragen in Bezug auf die Architektur/das Framework/das Muster gestellt werden:

- Welche Architektur/Frameworks/Muster sind (nicht) benutzt und warum (nicht)?
- Finden sich alle relevanten Teile wieder?
- Ist der Einsatz notwendig? Welches Kosten/Nutzenverhältnis ergibt sich?

13.3.2 Statische Struktur

- In diesem Schritt wird nun die Feinstruktur der Komponenten bis hin zur einzelnen Klasse betrachtet. Dabei werden die **Vererbungshierarchie** und die **Objektbeziehugen** näher betrachtet.
- Bezüglich der **Vererbungshierarchie** sollten die folgenden Fragen gestellt werden:
  - Ist dokumentiert, ob es sich um Vererbung im Sinne einer Generalisierung/Spezialisierung oder um Vererbung für Wiederverwendung handelt?
  - **Ist die Vererbungsbeziehung notwendig** oder lässt sich der gewünschte Effekt auch durch eine Benutztbeziehung bzw. mittels Aggregation erzielen?
  - Sind **Blätter** der Vererbungshierarchie **abstrakte Klassen**? Warum? Eliminierbar?
  - **Erben abstrakte Klassen von konkreten Klassen**? Warum? Eliminierbar?
  - Gibt es Klassen, die keine Attribute oder Methoden neu definieren bzw. redefinieren? Warum? Eliminierbar?
- Bei **Mehrfachvererbung** zusätzlich?
  - Ist der gewünschte Effekt klar dokumentiert?
  - Realisiert die Klasse mehrere Konzepte bzw. Aufgaben? Warum?
- Anhand der **Objektbeziehungen** lässt sich das System in Cluster, d.h. eng gekoppelte Klassen einteilen. Dabei sollte für **Klassen aus unterschiedlichen Clustern nur Benutztbeziehungen bestehen**. Daher sollte unter anderem gefragt werden:
  - Sind die **erkennbaren Cluster explizit als solche dokumentiert**?
  - Sind die **Cluster entkoppelt** (keine Konstruktions- oder Behälterbeziehungen)?
  - Exisitieren zwischen den Clustern **wenige, klar dokumentierte Benutztbeziehungen**?

13.3.3 Dynamik

- Zusätzlich zur Analyse der statischen Struktur sollten **alle wichtigen Aufgaben durchgespielt werden**.
- Das Ausführungsmodell ist **synchron und sequentiell**. Das heißt, es wird ein Wurzelobjekt erzeugt, welches andere Objekte erzeugt und bei diesen Dienste anstößt (Methoden aufruft), wodurch meist wiederum andere Dienste angestoßen werden.
- Es wird folgenden **objekt-orientierter Walkthrough** vorgeschlagen:
  - Die Objekte einer jeden beteiligten Klassen werden von einer Person vertreten, welche auch die aktuellen Zustände nachhält.
  - Es wird mit dem Ereignis begonnen, welches die durchzuspielende Aufgabe auslöst.
  - Es kann entweder sofort ab der entsprechenden Objektkonstellation begonnen werden oder aber erst beim Wurzelobjekt gestartet werden, bis eine entsprechende Konstellation vorliegt.
- Bei diesem Vorgehen sollten die folgenden Punkte besonders beachtet werden:
  - Sind alle verwendeten **atomaren Instanzvariablen initialisiert**?
  - Sind die **aktuellen Parameter** der vom Dienst ausgesendeten Botschaft (Methodenaufruf) **bekannt**? Stimmen z.B. die verwendeten Typen?
  - Werden die **Rückgabewerte** der Dienste adäquat weiterverarbeitet?

13.4 Code-Inspektion

- Quellcode kann nach verschiedenen inneren Qualitätsaspekten untersucht werden. Da dies jedoch sprach- bzw. firmenabhängig sein kann, werden nur allgemeine Kriterien aufgeführt:
  - **Wiederverwendbarkeit**. Sind die von der Anwendung zu überschreibenden bzw. bereitzustellenden Methoden dokumentiert und gekennzeichnet?
  - **Wartbarkeit**. Wurden alle Programmierrichtlinien eingehalten?
  - **Portabilität**. Wurden maschinennahe Befehle (Cast, Bitshift, etc.) eingesetzt? Wenn ja, sind diese gekapselt?
  - **Effizient**. Implementiert der Code den Algorithmus innerhalb der theoretischen Zeit/Raumkomplexität? Welche Kompromisse wurden warum getroffen?
- Auch bei der Codeinspektion wird ein objekt-orientierter Walkthrough empfohlen.

14 Vortrag Testmethoden

- **Ziel** ist die **Erstellung von Softwaresystemen**. Diese werden von Menschen erstellt, welche leider **Fehler** machen. Dafür kann es verschiedene **Ursachen** geben: Kommunikationsprobleme, Gedächtnisprobleme, zu hohe Komplexität oder mangelndes Wissen in den Anwenderwissenschaften.

- Leider können nicht alle Fehler von vornherein verhindert werden, daher müssen sie im nachhinein gesucht werden. Dies geschieht durch das Testen des Softwareprodukts.

- Das **Ziel des Testens** ist eine Erhöhung der Qualität des Softwareprodukts durch systematisches Ausführen der Software unter sorgfältig kontrollierten Umständen. Dabei wird der Testprozess mit der **Intention** ausgeführt, Fehler im Programm zu finden, also ein destruktiver Test. Die **Vorgehensweise** ist also eine im allgemeinen stichprobenartige Ausführung von Experimenten mit dem Prüfgegenstand unter spezifizierten Bedingungen. Warum stichprobenartig wird später noch erläutert.

- Dabei ist wichtig zu beachten, dass sich die Qualität eines Softwareprodukts nicht nur aus der Funktionalität sondern auch aus anderen Merkmalen zusammensetzt. Daher ist das Testen nur ein Aspekt des **Qualitätsmanagement**, dessen Ziel es ist, definierte Qualitätsmerkmale eines Softwareprodukts zu erreichen.

- Es können **zwei Arten von Qualitätsmanagementmaßnahmen** unterschieden werden:

  konstruktiv: Dies sind Methoden, Sprachen und Werkzeuge, die dafür sorgen, dass das entstehende Produkt **von vornherein** bestimmte Eigenschaften besitzt (Verwendung von Bibliotheken, moderne Programmiersprachen, etc.).

  analytisch: Dies sind Verfahren, welche **nachträglich** das Vorhandensein von gewissen Qualitätsmerkmalen überprüfen. Die im Buch vorgestellten Testmethoden fallen alle in diesen Bereich. Die Methoden der analytischen QMM können unter den folgenden Aspekten betrachtet werden (alles erklären):

  - Ablauf der Prüfung **statisch/dynamisch**
  - Vorgehensweise **informell/formal/experimentell**
  - Prüfung **vollständig** oder **stichprobenartig**.
  - Beschreibung des Prüfgegenstands ist **formal** (Quellcode), teilweise formal (Spezifikation) oder **informell** (Anforderungsspezifikation).
  - Prüfungen können vollständig, teilweise oder garnicht **automatisiert** werden.

- Ab hier Baum aufzeichnen und weiter erklären.

- Die **statischen Verfahren** können in formale und informelle Verfahren aufgeteilt werden.
- Zuerst die **informellen, statischen Verfahren**:
  - Schreibtischtest durch eine Person
  - Reviews durch eine Gruppe von Personen (z.B. Codeinspektion)

  Die informellen Verfahren entsprechem also einem **„genauen Hinschauen“** oder auch dem Erklären des Programms gegenüber anderen Personen. Diese Verfahren werden meist **vollständig** ausgeführt und sind daher sehr **aufwändig**. Der Vorteil dabei ist jedoch, dass **Fehler vor Ort** gefunden werden und diese Art der Prüfung in sehr **frühen Phasen** stattfinden kann.
- Bei den **formalen, statischen Verfahren** lassen sich vier Arten der Analyse unterscheiden:

  Syntaxanalyse: **Syntaktische Konsistenzbedingungen** werden überprüft, also z.B. ob der Quellcode syntaktisch korrekt ist.

  Datenflussanalyse: Der Datenfluss eines Programms wird überprüft. Dabei können z.B. Datenflussanomalien festgestellt werden.

  Kontrollflussanalyse: Der Kontrollfluss des Programms wird überprüft: z.B. auf offensichtlich unendliche Schleifen oder nicht erreichbaren Code.

  Formale Verifikation: Mit Hilfe von **mathematisch/logischen Verfahren** wird versucht, die Korrektheit eines Programms bezüglich einer Spezifikation zu zeigen. Dabei sind **zwei Schritte** zu zeigen:

  1. Programm liefert richtige Ergebnisse, wenn es anhält.
  2. Programm hält immer an.

  Kann von Hand oder per Theorembeweiser durchgeführt werden. Gibt jedoch erhebliche theoretische und praktische Probleme.

  Formale statische Verfahren dienen auch dazu, Informationen über die Programmstruktur abzuleiten $\Rightarrow$ **Kontroll- und Datenflussgraph**.
- Die **dynamischen Verfahren** lassen sich ebenfalls in formale und informelle Verfahren unterteilen.
- Ein **informelles, dynamisches Verfahren** ist der **Walkthrough**, der sich von der Inspektion nur im dritten Schritt unterscheidet. Dabei wird das Programm von Hand mit einem Eingabedatum durchgearbeitet, wobei also vom Eingabedatum abhängt, welche Teile des Programms betrachtet werden.
- Bei den **formalen, statischen Verfahren** finden sich die **symbolische Ausführung** und das **Testen im eigentlichen Sinne**.
- **Symbolische Ausführung:**

  - Programm wird nicht mit konkreten Werten sondern mit **symbolischen Werten** durchgerechnet. Beispiel für symbolische Werte geben ($w(C) = a + 2b$).
  - Bei **Entscheidungen** wird geschaut, welche Zweige ausführbar sind und die ausführbaren Zweige weitergerechnet. Dabei wird sich für jeden Zweig die sogenannte **Pfadbedingung** gemerkt.
  - **Schleifen** können nur für eine **begrenzte Anzahl** von Durchläufen betrachtet werden.
  - **Ergebnis** ⇒ Für jeden Pfad die symbolischen Werte der Variablen und der Pfadausdruck der angibt, wann diese Werte erreicht werden.

- **Testen im eigentlichen Sinne** ist eine **dynamische Prüfung** des Programms durch **experimentellen Ablauf**. Es ist aufgrund der Probleme der statischen Verfahren und der symbolischen Ausführung **unumgänglich**. Es wird unterschieden zwischen:

  - **Idealer Test:** Minimale Anzahl an Testdaten; nur zufälliges finden; nicht berechenbar (sonst Äquivalenz von Programmen entscheidbar).
  - **Erschöpfender Test:** Teste alle Eingaben; zu aufwändig; Beispiel.
  - **Stichproben-Tests:** Idealer Stichproben-Tests muss gefunden werden, der mit minimalen Stichproben maximale Anzahl an Fehlern ausdeckt ⇒ Kriterien, die von Testdaten erfüllt werden sollen, werden eingeführt

  Es lassen sich **zwei Teststrategien bzw. Testansätze** unterscheiden: Das **spezifikationsorientierte** und das **implementationsorientierte** Testen.

- **Spezifikationsorientiertes Testen:**

  - Testdaten und Testfälle werden anhand einer **Analyse der Spezifikation** gewonnen. Es wird also nicht die innere Struktur des Programms betrachtet.
  - **Vorteile:** In der Implementation **fehlende Teile** können gefunden werden. **Portierungen** werden unterstützt.
  - **Nachteile: Überflüssige Teile** in der Implementation werden nur zufällig gefunden. Die **Güte der Testdaten** hängt direkt von der Güte der Spezifikation ab.

- **Implementationsorientiertes Testen:**

  - Testdaten und Testfälle werden anhand einer **strukturellen Analyse der Implementation** gewonnen. Es wird also die innere Struktur des Programms betrachtet, wobei meist der **Kontrollfluss- oder Datenflussgraph** betrachtet wird.
  - Mit Hilfe der betrachteten Struktur werden **Überdeckungskriterien** definiert, welche die relevante Teilmenge aller möglichen Programmpfade festlegen.

- Struktur und Kontrollfluss soll untersucht werden ⇒ Kontrollflussbezogenes Testen.
- Fehlerhafte Anweisungen sollen untersucht werden ⇒ Datenflussbezogenes Testen oder ausdrucks-, anweisungs- oder datenbezogenes Testen.
- **Vorteile:** Es kann auch **ohne Spezifikation** getestet werden. **Zusätzlich implementierte Entscheidungen** werden gefunden. **Automatische Unterstützung** des Tests.
- **Nachteile: Nicht implementierte** Programmteile können nur zufällig gefunden werden.

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

Literatur

[Riedemann 1997] Riedemann, Eike Hagen (1997). *Testmethoden für sequentielle und nebenläufige Software-Systeme*. B.G. Teubner Stuttgart, Erste Aufl.

[Winter 1997] Winter, Mario (1997). *Reviews in der objekt-orientierten Softwareentwicklung*.