Softwarekonstruktion

Zusammenfassung der Vorlesung

Wintersemester 2014/2015

Fassung vom 13. März 2015

Inhaltsverzeichnis

In nahezu allen Bereichen von Wirtschaft und Gesellschaft spielen IT-Systeme eine wichtige Rolle. Der Einfluss der IT ist mittlerweile in fast allen Aspekten menschlichen Lebens deutlich spürbar, wobei in den letzten 10 Jahren die Erwartungen an die Vertrauenswürdigkeit der Systeme stark angestiegen ist, so etwa im Bereich *Cloud-Computing*. Allerdings wird gerade diese Anforderung oft nicht erfüllt. Dies liegt zum Teil daran, dass die bisher verwendeten Entwicklungsmethoden nicht mit den gestiegenen Erwartungen bei gleichzeitig steigender Systemkomplexität mithalten können.
Die Realisierung moderner Systeme erfolgt aus Kosten- und Flexibilitätsgründen meist über eine **offene Infrastruktur**[1]. Daraus folgt zwangsläufig, dass auch Personen auf diese Infrastruktur zugreifen können, die nicht unbedingt vertrauenswürdig sind. Daher muss der Zugriff **systemseitig** reguliert werden, was oft aus Kosten- und Flexibilitätsgründen auf Softwareebene gelöst wird. Eine vertrauenswürdige IT braucht also **sichere Software**.

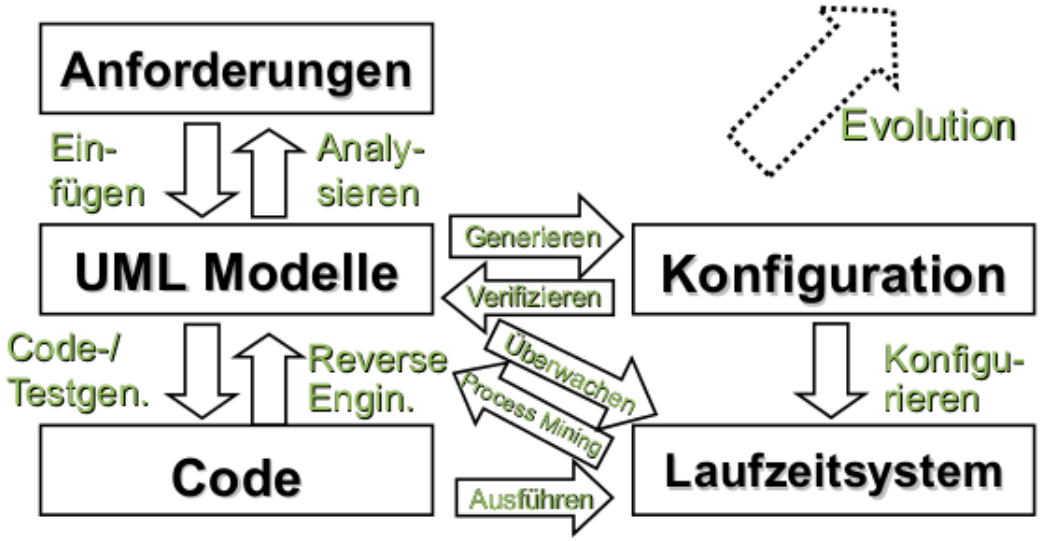


Abbildung 1: Die **modellbasierte Entwicklung** folgt diesem Schema.

Themen dieser Veranstaltung

Die Vorlesung gliedert sich in vier Teile, die sich jeweils über mehrere Termine erstrecken. Zunächst wird es um die **modellbasierte Softwareentwicklung** gehen, in der zweiten Hälfte um das **Qualitätsmanagement** bei der Softwareentwicklung. Eine feinere Gliederung der Themen geht aus dem Inhaltsverzeichnis hervor.

[1] Internet, mobile Netze

Software Engineering ist ...

»... die Übertragung und Nutzung von Praktiken aus der **Tontechnik**, um **wirtschaftliche** Software zu erhalten, die **sicher** ist und auch auf realen Maschinen **effizient** läuft.«[NR68]
»... das systematische Vorgehen bei der **Entwicklung**, **Nutzung**, **Wartung** und **Stilllegung** von Software.«[IEEE83]

Defizite in der Praxis

Im Gegensatz zu „herkömmlichen" Industrien (z.B. Automobil- oder Bauindustrie) sind die Herstellungsprozesse in der Software-Industrie nicht so planbar, zuverlässig, effektiv, effizient und flexibel. So kommt es häufiger zu Kosten- und Terminüberschreitungen sowie zur Auslieferung von ungenügender Software (Konzept der „Bananensoftware[2]"). Ebenso ist eine Produktivitätskontrolle nicht in dem Maße möglich, wie in anderen Fertigungsprozessen. Eine Qualitätskontrolle hingegen ist zwar möglich, jedoch wird diese – insbesondere das Testen – nach wie vor unterbewertet.

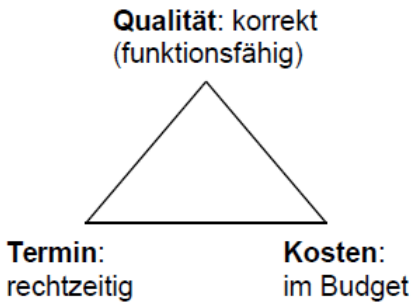


Abbildung 2: Schematische Darstellung der Interessenskonflikte bei der Softwareherstellung.

Verwendete Maßeinheiten

Ein *Function Point* (FP) entspricht (beispielsweise)

- etwa 55 *Lines of Code* (LOC) in C++ oder Java
- 20 LOC in Perl
- 13 LOC in SQL

Bei einem Desktop-Projekt mit einem FP dauert die Entwicklung ca. eine Woche mit einem Entwickler. Ein allgemeineres Projekt mit 100.000 FP und 100 Entwicklern dauert ca. 17 Monate.

[2] Die Software reift beim Kunden.

Teil I

Modellgetriebene Softwareentwicklung

In diesem Teil der Vorlesung geht es um die Vertiefung der modellbasierten Softwareeinwicklung. Insbesondere werden hier die aus SWT bekannten elementaren Inhalte fortgesetzt, aber auch weitere Prinzipien vorgestellt:

Modellbasierte Softwareentwicklung: Metamodellierung, Modelltransformation, Object Constraint Language

Geschäftsprozessmodellierung: Ereignisbasierte Prozessketten (EPK)

Petrinetze: Ausführungssemantik

UML-Werkzeugbau: Eclipse Modeling Foundation (EMF)

Geschichte der Softwareentwicklung

In den Anfängen der Softwareentwicklung (1950er-Jahre), wurde die Computer-Welt vor allem durch teure Hardware bestimmt, die Software der damaligen Zeit hatte eine beherrschbare Größe und war binär in Maschinensprache codiert. Später wurden die Assembler-Sprachen entwickelt, vor allem aus der Notwendigkeit heraus, dass komplexere Softwaresysteme nicht dazu geeignet waren, in Binärcode entwickelt zu werden. Durch mnemonische Symbole, Marken und Makros wurde die Arbeit an Softwaresystemen erheblich vereinfacht, was schließlich zur Ablösung der Maschinensprache durch Assembler führte.
Ebenfalls in den 1950er-Jahren wurde mit FORTRAN die erste höhere Programmiersprache für mathematisch-technische Probleme entworfen. Sie diente vielen weiteren höheren Sprachen als Vorbild. Etwa zehn Jahre später – Ende der 1960er-Jahre – wurde der Begriff des Software-Engineering geprägt. Dies geschah im Zuge der Software-Krise (1968), da aufgrund des wachsenden Software-Bedarfs sowie der immer größer werdenden Komplexität der Softwaresysteme Forderungen nach einer neuen Ingenieursdisziplin laut wurden. Ferner entstanden systematische Methoden zur Softwareentwicklung: Die strukturierte Analyse, sowie das strukturierte Design.

Strukturierte Analyse In dieser Methode haben Diagramme und Modelle einen hohen Stellenwert. Die Prozesse werden mithilfe von Datenflussdiagrammen abstrakt modelliert sowie durch Mini-Spezifikationen beschrieben.

Strukturiertes Design Funktionen werden in hierarchisch aufgebaute Modelle zerlegt und in Strukturdiagrammen festgehalten. Insbesondere dienen die Diagramme der Visualisierung der Abläufe sowie der Beschreibung von Schnittstellen.

Im Zuge der sog. Wiederverwendungskrise[3] Ende der 1980er-Jahre wurde das Programmierparadigma der Objektorientierung entwickelt. Dieser Ansatz ist bereits in wissenschaftlichen Veröffentlichung in den 1970er-Jahren vorgestellt worden. Objektorientierung bedeutet grundsätzlich:

- Ein System besteht aus mehreren Objekten
- Jedes Objekt besitzt ein definiertes Verhalten, einen inneren Zustand und eine eindeutige Identität

Terminologie IEEE 1471

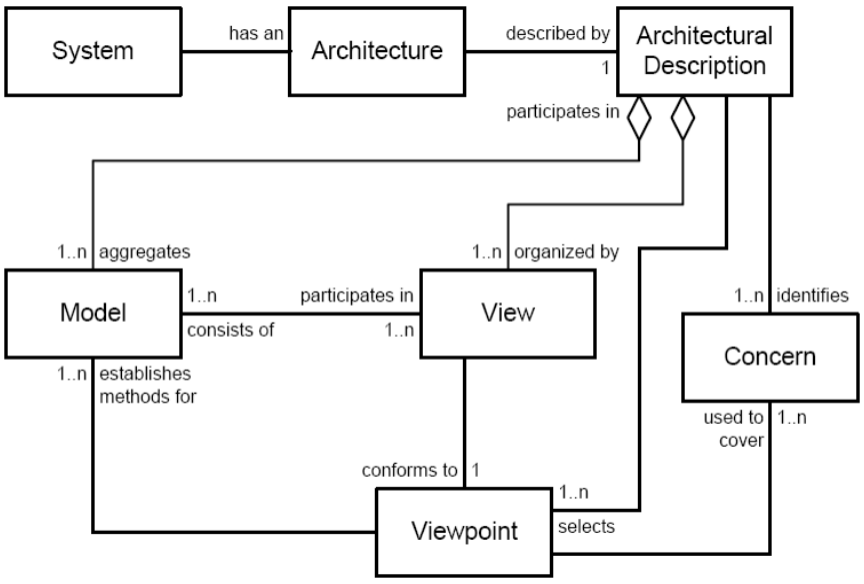


Abbildung 3: Terminologie IEEE 1471 in grafischer Darstellung

System: Ein aus Teilen zusammengesetztes und strukturiertes Ganzes, das eine Funktion hat und einen Zweck erfüllt.

Architektur: Die Organisation der Teile des Systems und deren Verhalten. Es werden insbesondere die Beziehungen unter einzelnen Komponenten ersichtlich, die zur Erreichung des Systemzwecks notwendig sind. Die Menge aller Artefakte wird in der architektonischen Beschreibung zusammengefasst. Diese Menge ist zur Beschreibung der Architektur zwingend notwendig.

Concern (Aspekt): Die spezifischen Anforderungen oder Bedingungen. Diese Anforderung sind zu erfüllen, wenn der Systemzweck erreicht werden soll.

Model: Beschreibung oder Spezifikation eines bestimmten Zwecks, die alle relevanten Aspekte umfasst.

[3] Es wurden gewaltige Softwarebestände angehäuft, die nur teilweise wiederverwendbar waren, sodass häufig ähnliche Problemstellungen immer wieder von null auf implementiert werden mussten.

Viewpoint: Beschreibung aller Elemente und Konstrukte, die die Erstellung des Modells zur Beschreibung von Systemaspekten ermöglichen. Der Viewpoint umfasst alle Regeln, Techniken und Notationsmittel zur Erstellung eines Modells.

View: Kohärente Menge von Modellen, es werden bestimmte Aspekte eines Systems beschrieben, aber auch Aspekte verborgen, die irrelevant sind. Man spricht in Bezug auf die View häufig auch von der **Sicht auf das System**. Die Darstellung erfolgt nach den Regeln und der Notation des jeweiligen Viewpoints.

1 Einführung und Motivation

Im Laufe der Geschichte der Softwareentwicklung war die Beherrschung der Komplexität stets das Hauptproblem. Bei der Entwicklung wird dabei bis heute das eigentliche Vorhaben abstrahiert. Der Abstraktionsgrad wurde während der Weiterentwicklung der Programmiersprachen[1]. Dabei gewinnen Modelle immer mehr an Bedeutung.
Zunehmende Abstraktion bei der Softwareentwicklung liegt vor allem in der zunehmenden Komplexität der Software begründet, die immer weiter zunimmt. Diese Zunahme kann man sich anhand der eingebetteten Software im Automobilbereich vor Augen führen: Um 1970 arbeiteten in den eingebetteten Systemen eines Autos etwa 100 LOC (Lines of Code), heute sind es bis zu 10 Millionen.

1.1 Das Problem der Komplexität

Die Komplexität ist jedoch nicht gleich Komplexität. Vielmehr unterscheiden wir drei Teilbereiche:

Technische Komplexität: Umfang der Datenmodelle, Verteilte Implementierung, Heterogenität der Infrastruktur

Funktionale Komplexität: Umfang der Funktionalität, Diversifizierung der Funktionalität und Mensch-Maschine-Schnittstelle

Entwicklungskomplexität: Einflussnahme des Kunden, Entwicklung von Zuliefererketten Qualitätsanforderungen, Innovationsdruck sowie der Aufbau auf vorhandenen Implementierungen (Überarbeitung)

Diese drei immer weiter zunehmenden Teilbereiche stehen in stetigem Konflikt mit der Erfüllung folgender Kriterien:

Qualität: Aus der Sicht der Nutzer stehen die Funktionsvielfalt, Nutzbarkeit, Sicherheit und Performanz im Vordergrund, während die Entwickler mehr die Wartbarkeit und die Wiederverwendbarkeit im Blick haben.

Kosten: Neben den Entwicklungskosten und der Vermarktbarkeit des Endprodukts stehen die entstehenden Kosten im Gesamtlebenszyklus[2] im Fokus.

Entwicklungszeit: Hierbei werden sowohl die Zeit bis zum Erreichen der Marktreife sowie die Reaktionszeit auf geforderte Änderungen betrachtet.

[1] Am Anfang war die Maschinensprache, gefolgt von Assembler. Danach wurden prozedurale Sprachen verwendet und schließlich die objektorientierten Sprachen. Über den objektorientierten Sprachen stehen nur noch Modellierungssprachen wie UML.
[2] Entwicklung, Inbetriebnahme, Wartung

1.2 Komplexität beherrschen

Zur Beherrschung der Komplexität gibt es neben der bereits erwähnten Abstraktion, also dem Ausblenden unwichtiger Details unter Verwendung geeigneter Modelle, noch weitere Ansätze:

Strukturierung und Modularisierung: Auch bei diesem Ansatz werden geeignete Modelle eingesetzt. Darüber hinaus wird die Gesamtaufgabenstellung in klar abgegrenzte Unterstrukturen aufgeteilt. Man spricht bei dieser Partitionierung von Aufgaben auch von einer **Dekomposition**.

Methodik und Systematik: Wenn in der Vergangenheit bereits bewährte Verfahren und Lösungsansätze entwickelt wurden, so werden diese wieder verwendet. Dabei geht es vor allem darum, nicht jedes mal „das Rad neu zu erfinden". Ferner wird der Entwicklungsprozess systematisiert, was auf Code-Ebene durch sog. *Design Patterns* erfolgt, bzw. bei der Softwarearchitektur durch Referenzarchitekturen.

1.3 Weitere akute Probleme

Steigende Anforderungen: Die Anforderungen an die Leistungsfähigkeit, Zuverlässigkeit und die Qualität der Software sind in der Vergangenheit ebenfalls spürbar gestiegen. Dies ist vor allem auf kurze Technologiezyklen sowie häufige Anforderungsänderungen zurück zu führen. Darüber hinaus entsteht ein hoher Druck durch die Reduzierung von Kosten – insbesondere in Zeiten konjunktureller Schwäche.

Fachlichkeit vs. Technik: Zur Zeit wird die Fachlichkeit von der Technik dominiert, insbesondere die Umsetzung fachlicher Basiskonzepte. So benötigen etwa Anwendungsentwickler ein sehr umfangreiches technisches Wissen, statt sich auf die Anwendungsdomäne zu konzentrieren. Die Verständigung zwischen Entwickler und Fachabteilung hingegen funktioniert auf völlig unterschiedlichen Abstraktionsebenen, wobei das Abstraktionsniveau heutiger Entwicklungsansätze häufig zu niedrig ist. Eine durch Fachlichkeit bestimmte Entwicklung liegt nach wie vor noch in weiter Ferne.

Fehlende Durchgängigkeit: In der gängigen Praxis der Softwareentwicklung existiert ein methodischer Bruch zwischen Analyse, Design und Implementierung. Nach Beginn der Implementierungsphase werden die Modelle meist nicht aktualisiert, was sie völlig nutzlos macht. Ein daraus resultierendes Problem ist die erschwerte Einarbeitung neuer Mitarbeiter in späteren Phasen sowie um das Vielfache erhöhte Rüstzeiten während der Betriebs- oder Wartungsphase.

Divergenz der Änderungszyklen: Technische und fachliche Belange müssen bei der Entwicklung getrennt werden, etwa durch einen besseren Investitionsschutz der Entwicklungsarbeit, eine verlängerte Lebensdauer der Fachkonzepte und eine größere Flexibilität bei Änderungen der Fachanforderungen.

1.4 Folgen dieser Probleme

Es bedarf bei Entwicklungsvorgängen einem erhöhten Abstraktionsniveau, was aus heutiger Sicht keinesfalls selbstverständlich ist. Dazu müssen Potential und Gleichförmigkeiten in verdichteter Form zusammen gefasst werden, wobei Ausdrucksmittel genutzt werden, die durchgägnig etabliert sind und genutzt werden. Mögliche Modelle wären:

- Abstrahieren und fokussieren auf das Wesentliche
- Schlagen von Brücken von der Welt der fachlichen Probleme in die Welt der technischen Lösungen

1.5 Kernideen der modellbasierten Softwareentwicklung

Im Entwicklungsprozess von Software sind Modelle ein zentrales Artefakt. Sie sollten von der ersten bis zu letzten Phase[3] im gesamten Entwicklungsprozess genutzt werden, wobei es zu vermeiden ist, dass es zu Modellbrüchen kommt. Ihr Einsatz erfolgt unter Verwendung von fachlicher Semantik, d.h. die fachlichen Anforderungen werden von der konkreten Technologie entkoppelt. Häufig findet zu diesem Zweck die **Unified Modeling Language (UML)** Verwendung, was aber nicht zwingend notwendig ist.

Diskussion: Modellbasierte Szenarien

Für welche Zwecke könnte man Ihrer Meinung nach Modelle in der modellbasierten Softwareentwicklung verwenden?

- Dokumentation und Kommunikation, dabei wird zwischen **Analysemodellen** (Fachexperte und SW-Architekt), **Enwurfsmodellen** (SW-Architekt und Entwickler) und **Implementierungsmodellen**, welche eine Verfeinerung des Modells der Entwurfsphase darstellen.
- Erkennung von Anti-Pattern[4]
- Code-Generierung
- Spezifikation, also das verbindliche Festhalten von Funktionen der Software zwischen Auftraggeber und IT-Firma
- Tests und Simulationen

1.5.1 Der Begriff „Modell“

Ein Modell ist die Abbildung der Welt auf eine diskrete Struktur. Bei der Modellierung wird das Ziel verfolgt, die reale Welt vereinfacht darzustellen, bzw. zu beschreiben. Insbesondere werden Aspekte wie Struktur, Beziehungen und Verhalten modelliert.

[3] Von der Anforderungsanalyse bis zur Wartung
[4] Wichtig klingender Begriff, Synonym für „scheiß Lösungsansatz“

1.6 Syntax vs. Semantik von Modellen

1.6.1 Menschliche Sicht

Die Syntax wird in zwei Untergruppen geteilt: Die **konkrete Syntax**, die die sichtbaren Modellelemente beinhaltet sowie die **abstrakte Syntax**, welche aus abstrakten Repräsentationen besteht. Abhängig von der Art des Modells stellt die Semantik die Bedeutung der Aussage des Modells dar. Beispielsweise stellt ein UML-Klassendiagramm dar, wie die Beziehungen zwischen Klassen aussehen, während ein UML-Statechart das Verhalten eines Systemteils darstellt.

1.6.2 Werkzeugsicht

Hier enthält die Syntax werkzeugrelevante Standard-Formate, etwa für die Speicherung der Modelle. Für die Semantik existiert kein einheitliches Format, für UML gibt es beispielsweise die Object Constraint Language (OCL), währen auch Alternativen wie Petrinetze, Abstract Sate Machines, uvm. existieren

1.6.3 Modellgetriebene Ansätze

Modelle zu Artefakten erster Klasse sind hinreichend zur Generierung eines Softwaresystems. Auf der nächsthöheren Abstraktionsstufe bietet eine Modellierungssprache dichtere Notationen für Teilmengen von Konzepten im Vergleich zu klassischen Programmiersprachen.

1.7 Model-Driven Architecture (MDA)

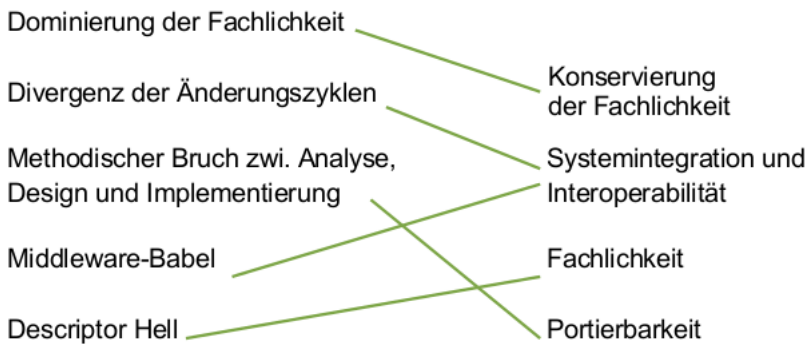
Die MDA ist ein konkrekter Ansatz für die modellbasierte Entwicklung, bei dem die Spezifikation der Software von ihrer technischen Umsetzung völlig unabhängig ist. Der Übergang von abstrakten zu technologiespezifischen Modellen erfolgt **vollautomatisiert** unter Verwendung von Transformationswerkzeugen. MDA bringt einige Vorteile mit sich:

- **Effiziente Softwareentwicklung** durch eine Automatisierung der Anwendungserstellung. Durch eine Kapselung der Technologien kann das Expertenwissen durch Transformationsvorschriften wiederverwendet und vorhandene Ressourcen optimal genutzt werden.
- **Portierbarkeit** erleichtert die systematische Abstraktion technischer Aspekte:
  - Migration von Applikationen auf neue Versionen benutzter Technologien
  - Portierung auf andere Zielumgebungen
  - Anwendungsentwicklung für mehr als eine Zielplattform
- **Systemintegration und Interoperabilität** erleichtern Wiederverwendung und steigern Produktivität durch Verwendung offener Standards sowie Trennung fachlicher Konzepte von konkreter technologischer Repräsentation.
- Konservierung der Fachlichkeit

- **Domäneorientierung**, also die Orientierung an der Fachlichkeit innerhalb des Enwicklungszyklus. So werden unnötige Beschränkungen durch technologische Vorgaben umgangen und die Geschäftsprozesse unterliegen einer höheren Flexibilität. Die Prozesse werden unter ständiger Pflege und Weiterentwicklung der Anwendungslogik plattformunabhängig modelliert.

  Die Domäneorientierung hat insbesondere das Ziel, die **Time-to-Market**-Zeit ohne Budgeterhöhung zu verkürzen.

Diskussion: Welche akuten Probleme der Softwareentwicklung lassen sich mit den Zielen der MDA verbinden?



1.8 Zentrale Begrifflichkeiten

Metamodell: Enthält die die Regeln, die bei der Erstellung des Modells beachtet werden müssen (abstrakte Syntax) sowie die Bedeutung der Elemente und Elementkonstellationen (Semantik).

Profile: Leichtgewichtige Erweiterung des Metamodells. Es werden neue Modellelemente zu Verfügung gestellt oder eine Semantik und Syntax bestehender Elemente erweitert. Außerdem werden die Bedingungen an die Elemente des Modells verschärft (Design-by-Contract).

Domain: Abgrenzbares, kohärentes Wissensgebiet. Im Metamodell werden die Konzepte einer Domäne in Bezug gesetzt und beschrieben.

Application (Anwendung): Ein zu erstellendes Stück Software, welches zu entwickelnde Funktionalitäten umfasst. Es können Systeme aus einer oder mehrerer Anwendungen zusammengesetzt werden, die auf einer oder mehrerer Plattformen ausgeführt werden können.

Plattform: Ausführungsumgebung einer Anwendung. Der Zugriff auf die Funktionalitäten erfolgt über Schnittstellen (z.B. GUI oder Peripherie), ohne dass die Implementierung bekannt sein muss. Die Hardware eines Computers bildet eine Plattform für das Betriebssystem, welches wiederum eine Plattform für einzelne Anwendungen darstellt.

Computation Independent Model (CIM): Liefert eine Sicht auf das Gesamtsystem. Das Modell wird im Vokabular seiner Domäne beschrieben, woei die Anforderungen an das System und seine Umwelt betont werden.

Platform Independent Model (PIM): Beschreibt die formale Struktur und Funktionalität eines Systems. Es wird dabei von der zugrunde liegenden Plattform abstrahiert, wobei die Details der Implementierung außer Acht gelassen werden.

Platform Specific Model (PSM): Analog zu PIM, aber mit plattformabhängigen Informationen.

2 Modellbasierte Softwareentwicklung

In diesem Kapitel werden wir uns den fortgeschrittenen Techniken der modellbasierten Softwareentwicklung zu wenden, z.B.:

- Metamodellierung
- UML-Erweiterungen
- Modelltransformation
- Design Patterns

UML kann eingesetzt werden, um Code und Testfälle zu generieren. Darüber hinaus ist es möglich mit den richtigen Werkzeugen Modelle zu validieren und zu transformieren. Damit diese vier genannten (in UML steckt noch mehr Potential) Anwendungen funktionieren können, müssen die dafür verwendeten Werkzeuge UML verstehen können, d.h. ihnen muss die **formelle Definition** der Notation bekannt sein.
Eine „formelle“ Definition der UML-Notation kann auf verschiedenen Wegen erfolgen (BNF-Grammatik, XML-Schema, etc.). UML wurde von den Entwicklern zu nächst nur als zentrales Konzept definiert. Diese Basis wurde schließlich unter Verwendung dieses Konzepts schließlich immer mehr vervollständigt, bis UML seinen heutigen Umfang erreicht hatte.

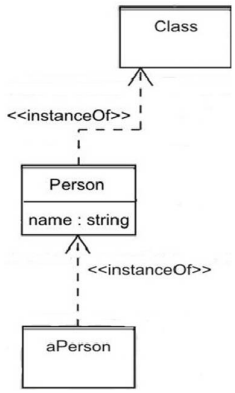


Abbildung 2.1: Das oben genannte Konzept (was auch das zentrale Konzept der objektorientierten Programmierung darstellt) wird an dieser Abbildung verdeutlicht: Zunächst wurde das Metamodell (Modell, das die Notation einer Modellierungsnotation definiert.) *Class* definiert, was im UML-Modell die Menge aller *Personen* enthält, also eine Menge von Instanzen (hier *aPerson*).

2.1 Metamodellierungs-Hierarchie

Schicht M0: Die Ebene des auszuführenden Softwaresystems.

Schicht M1: Ein UML-Modell, das aus dem UML-Metamodell instantiiert wird und die Menge aller validen Laufzeit-Instanzen definiert.

Schicht M2: UML-Metamodell, das die Menge aller validen UML-Modelle definiert.

Schicht M3: Meta-Metamodelle, definieren die Menge aller validen Metamodelle. Ein konkreter Ansatz hierfür ist die **Meta Object Facility (MOF)**[1], die damit definierten Modelle sind austauschbar mit XML Metadata Interchange (XMI), einem besonderen XML-Dialekt.

Schicht XYZ: Theoretisch könnte es auch noch weitere Schichten geben, was aber ab einer gewissen Tiefe einfach keinen Sinn mehr macht.

Anmerkung: Diese Hierarchie ist UML-spezifisch!

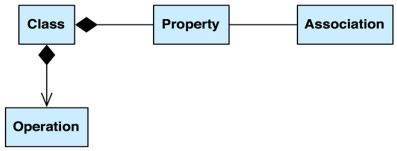
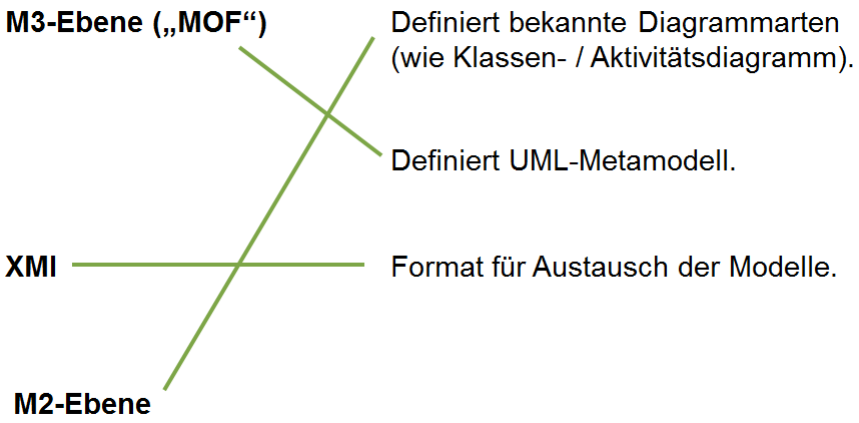


Abbildung 2.2: Eine Klasse (*Class*) besteht aus Attributen (*Property*) sowie Methoden (*Operation*) und steht mit anderen Klassen in einer Beziehung (*Association*).

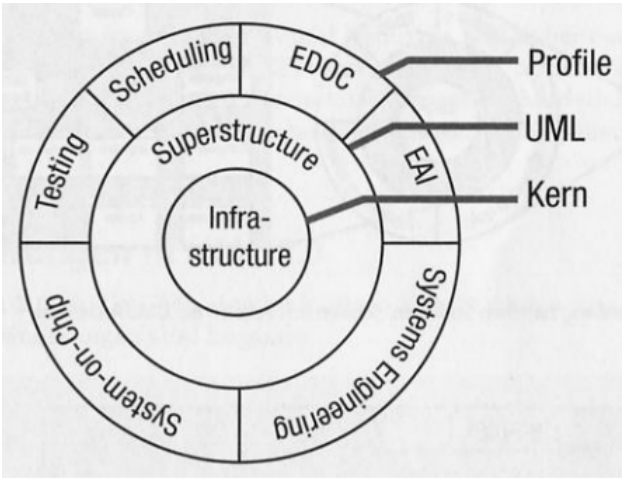
2.2 Modell und Metamodell der UML

Diskussion: Welche Aussagen passen zu den Kernelementen der UML?



[1] Ansatz der **Object Management Group (OMG)**

Schichten-Architektur



Quelle: UML für Studenten, Harald Störrle

Abbildung 2.3: Den Kern dieser Architektur bildet die **Infrastructure** (ein Metamodell), die Die eigentliche Notationsdefinition der UML bezeichnet man als **Superstructure**. **Profile** bilden mögliche (optionale) Erweiterungen.

2.3 Vor- und Nachteile der Metamodellierung

Vorteile

Die allgemeinen Vorteile liegen in der präzisen Definition der Modellierungsnotation und im einheitlichen Ansatz, der von mehreren Notationen verfolgt wird.
MOF besitzt eine wohlverstandene Notation für Metamodelle sowie eine weitgehende Werkzeugunterstützung.

Nachteile

Die Erstellung von Metamodellen ist mit relativ viel Aufwand verbunden und es ist eine gewisse Expertise von Nöten.
Bei MOF wird auf verschiedenen Ebenen dieselbe Notation verwendet, etwa in M2 und M3, was mitunter verwirrend sein kann.

2.4 Erweiterung der UML

Fehlendes Fachwissen ist ein großes und teures Problem bei der Entwicklung und Wartung von IT-Systemen. Diesem Problem kann entgegen gewirkt werden, indem die fachlichen Konzepte von Anfang an in die Modellierung mitaufgenommen werden und die fachlichen Zusammenhänge dokumentiert werden. Außerdem wird so die Codegenerierung bestimmter fachlicher Aspekte ermöglicht.
Wie bereits erwähnt, abstrahiert UML von jeder fachlichen Domäne, sodass es mitunter notwendig sein kann UML zu erweitern, da die Notation möglicherweise bestimmte Aspekte nicht erfassen kann. Zwar besteht auch die Möglichkeit, die Diagramme durch erläuternde Texte zu ergänzen, jedoch verliert dieser Ansatz schnell an Übersichtlichkeit und ist daher ungeeignet. Daher werden wir uns in diesem Abschnitt der Erweiterung der Notationselemente in UML widmen.

Die Modellierungsnotation kann durch eigene Metamodelle erweitert werden, was jedoch sehr aufwändig ist. Außerdem ist eine Verletzung der existierenden Semantik denkbar, wenn man eine direkte und beliebige Erweiterung des Metamodells zulässt. Eine geeignetere Methode zur Erweiterung sind so genannte **UML-Profile**, die über eine vorgesehene Schnittstelle das Metamodell lediglich verfeinern.

UML-Profile

Diese Profile sind eine Spezialisierung von Standard UML-Elementen zu konkreten Metatypen. Für verschiedene Anwendungsdomänen existieren verschiedene Profile. Es sind bereits vordefinierte Profile bei OMG verfügbar, es ist jedoch auch möglich, eigene zu definieren.

Definition eines Profils
Ein Profil ist ein Paket von *Stereotypen* und *Tagged Values*. Ein Klassendiagramm definiert Beziehungen zwischen neuem Stereotyp und einem zu beschreibenden Element.

Stereotyp: Spezialisiert Nutzung von Modellelementen <<label>>.

Tagged Value: Fügt Paare der Form ⟨tag=value⟩ zu stereotypisierten Elementen hinzu. Man spricht bei Tagged Values auch von Name-Wert-Paaren.

Constraint: Verfeinert die Semantik eines stereotypisierten Elements (b.B. mittels OLC)

Profil: Sammelt alle obigen Informationen

2.5 Modelltransformation

Ziel einer Transformation ist es, ein Quellmodell *A* unter Verwendung zusätzlicher Informationen in ein Zielmodell *B* zu überführen. Prinzipiell können alle definierbaren Modelle Ouell- und/oder Zielmodelle sein. Es wird unterschieden zwischen **horizontaler Transformation**, also der inhaltlichen Weiterentwicklung eines Modells, und der **vertikalen Transformation**. Letztere dient der Umwandlung eines Modells einer Abstraktionsebene in ein äquivalentes Modell einer anderen Abstraktionsebene. Sie ist außerdem ein Kernstück des MDA-Ansatzes. Ein weiteres konkretes Beispiel für eine Modelltransformation ist die **Codegenerierung**, also die Überführung eines Modells in Quellcode.

2.5.1 Codegenerierung

Ein Generator erzeugt Programmcode für spezifische Anwendungs- oder Programmklassen. Dabei wird ein generisches Programmmodell gekapselt, der letztlich erzeugte Code ist abhängig vom Modell, der Transformationslogik und den Parametern. Grob teilt sich ein **Generatorlauf** in folgende Phasen:

- Einlesen der Eingabespezifikation

- Einlesen der Parameter
- Anwenden der Transformationsregeln auf die Eingabespezifikation unter Berücksichtigung der Parameter
- Ausgabe von Programmcode

Dabei haben wir drei Phasen der Codegenerierung:

1. Programmierung eines Generators
2. Parametrierung und Ergänzung eines Modells
3. Parametrierter Aufruf des Generators und Erstellung des Programms

Die Codegenerierung arbeitet mit variabilisiertem Programmcode, dessen eindeutige Ausprägung durch Parametrierung des Programmcodes beeinflusst wird. Es ist jedoch zu beachten, dass die Entwicklung eines Generators sehr aufwändig sein kann, sodass die Eignung für Lösungen mit entsprechend großer Zahl an Variationen beschränkt ist. Allerdings ist die Qualität aller Lösungen immer gleich und der Wartungsaufwand wird zentralisiert. Außerdem können mehrere Lösungen in kurzer Zeit erstellt werden.

2.5.2 Modelltransforamtionssprachen

Modelltransforamtionssprachen sind ein Mittel zur Beschreibung von Transformationen von Instanzen eines Metamodells in ein anderes Metamodell.

Deklarative und imperative Transformationssprachen

Die Beschreibung der Transformation bei deklarativen erfolgt anhand von Regeln, welche mit Vor- und Nachbedingungen spezifizierbar sind. Viele deklarative Transformationsansätze sind durch Graphentransformation realisierbar. Imperative Transformationssprachen hingegen beschreiben die Transformation durch eine Sequenz von Aktionen.

Query View Transformation (QVT)

Die transformierten Metamodelle werden mittels MOF beschrieben. Dieser Standard der OMG besteht aus zwei Transformationssprachen:

QVT Relations: Deklarative Sprache, kann bidirektional und inkrementell transformieren.

QVT Operational Mapping: Imperative Sprache, die Relations verwenden kann.

Atlas Transformation Language (ATL)

Dies ist die Sprache des **Eclipse-M2M-Projekts**, sodass auch eine Werkzeugunterstützung in Eclipse gegeben ist. ATL ist eine hybride Sprache, d.h. sie ist sowohl deklarativ, als auch imperativ. Abfragen auf Modellen werden mit OCL realisiert und es können verschiedene Arten von Metamodellen verarbeitet werden.

2.6 Design Pattern

Da Designprobleme in der Regel nicht nur einmal auftreten sollte ein Designer seine Lösungen stets aufbewahren. Ein gute Designer benötigt viel Erfahrung, was im Klartext bedeutet, dass er sein wissen festhalten, vermitteln, verwenden und so erhalten sollte. Bei der Erstellung eines Designmodells sollte der Designer sich daher zunächst den nötigen Lösungsansatz erarbeiten und sich im Folgenden darüber Gedanken machen, wie dieser Ansatz verallgemeinert werden kann. Verallgemeinerte Lösungen werden auch Muster genannt und haben im Wesentlichen immer vier Bestandteile:

1. **Name**
2. **Problem**, inklusive
    - Annahmen
    - Enflussfaktoren
3. **Lösung:**
    - bildliche und
    - sprachliche Beschreibung
4. **Anwendung** des Musters:
    - Konsequenzen
    - Trade-offs

2.6.1 Beispielmuster: Observer

Wir werden nun beispielhaft die vier Bestandteile an dem bereits aus SWT bekannten Entwurfsmuster der **Observer** erläutern. Erstaunlicherweise lautet der Name dieses Musters „Observer“. Das Problem ist es, eine 1-zu-n-Beziehung zwischen einem zentralen Objekt und n abhängigen Objekten zu realisieren. Wenn sich der Zustand des zentralen Objekts ändert, so muss der Zustand aller n abhängigen Objekte aktualisiert werden.

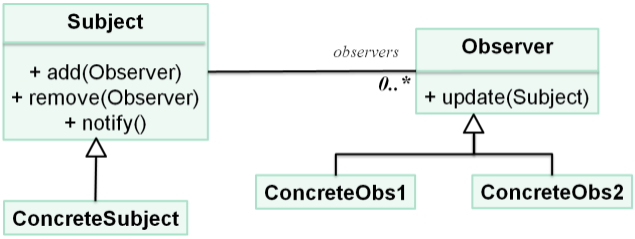


Abbildung 2.4: Diese Grafik zeigt die Struktur der Lösung dieses Problems. Es sei jedoch angemerkt, dass diese Lösung nicht anwendungsspezifisch sondern nur allgemein gehalten ist.

Die aus der Anwendung dieses Musters resultierenden Konsequenzen sind zum einen die Anpassbarkeit, die Modularität sowie die Erweiterbarkeit. Zum anderen ist ein Update nur bei Kenntnis aller Observer wirklich effizient. Verwendungen dieses Musters liegen u.a. in

- Smartphone Event Frameworks
- Mailing-Listen

2.6.2 Vor- und Nachteile von Mustern

Vorteile

Muster ermöglichen es, erworbenes Designwissen strukturiert festzuhalten. Strukturen werden explizit benannt und Entscheidungen dokumentiert. Wurde ein Entwurf einmal realisiert, so kann er leicht programmiersprachen- und implementierungsunabhängig wiederverwendet werden. Wiederverwendbarkeit ist eine wichtige Basis für Automatisierungen.

Nachteile

Muster können die Entwurfsoptionen jedoch auch einschränken. Es wird bewusst von der Implementierungsebene abstrahiert, sodass (wichtige) Implementierungsdetails zunächst ungelöst bleiben, wobei eine manuelle Implementierung mühsam um fehleranfällig ist.

3 Grundlagen der Object Constraint Language (OCL)

In diesem Kapitel werden wir uns von den Modellen zu den Objekten hinwenden. Die Object Constraint Language ist eine logikbasierte Notation für Einschränkungen in UML-Modellen, allerdings kann sie auch unabhängig von UML verwendet werden. Mit ihr kann zum Beispiel eingegrenzt werden, welche Klassen erreichbar, welche Attribute, Operationen und Assoziationen für Objekte dieser Klassen vorhanden sind. Außerdem beinhaltet sie Bedingungen an Wertebelegungen während der Ausführung der modellierten Systemteile. Durch eine präzise Spezifikation der Bedingungen wird Mehrdeutigkeit beseitigt. Ferner ist es möglich, durch die standardisierte Semantik die Bedingungen automatisch zu verifizieren. Es sei jedoch angemerkt, dass OCL keine Programmiersprache ist, es ist also keine Modellierung von Programmlogik möglich. Allerdings unterstützt OCL die Prüfung von strenger Typisierung. OCL-Ausdrücke haben darüber hinaus keinerlei Möglichkeiten, Zustände in einem Modell zu verändern, wohl aber diese zu spezifizieren.

3.1 Verwendung von OCL

OCL-Ausdrücke sind stets an UML-Modelle gebunden und beschreiben die Einschränkungen für Elemente des Modells, zu dem sie gehören. Wir werden in diesem Kapitel nicht sehr detailliert durch die Folien gehen, da sich viele der Erläuterungen aus den Bearbeitungen der Übungsaufgaben in Teil 3 ergeben. Es können insgesamt zwei Arten von Einschränkungen spezifiziert und verifiziert werden:

- **Fortlaufende** Zustandsbeschränkung mit Invarianten
- Zustandsbeschränkungen **vor** bzw. **nach** einem Methodenaufruf (mit Vor- und Nachbedingungen)

Ein OCL-Ausdruck kann folgende Basisformen haben:

```
1 context <identifier>
2 <constraintType> [<contraintName>]: <boolean expression>
```

Erläuterung: Ein Context kann auch auf Variablen anderer Contexts zugreifen und mehrere Cons-

`context`	Schlüsselwort aus OCL
`<identifier>`[1]	Klassen- oder Operationsname
`<constraintType>`	Schlüsselwort `inv`, `pre` oder `post`
`<constraintName>`	optionaler Name für diesen Constraint
`<booelanExpression>`	boolescher Ausdruck

traint Types haben. Sollten zwei Packages eines Modells jeweils eine Klasse mit demselben Namen haben, so müssen die Identifier mit dem Namen ihres Packages als Präfix spezifiziert werden.

[1] Kann ein zugehöriges Modellelement markieren oder referenzieren sowie innerhalb von `<booleanExpression>` genutzt werden.

Constraints sind Einschränkungen, die auf einem oder mehreren Teilen des UML-Modells gelten. Es gibt folgende Arten von Einschränkungen:

Class Invariant: Muss **immer** von allen Instanzen einer Klasse erfüllt sein. Wird häufig verwendet, um die Werte einzuschränken, die ein Attribut annehmen kann. Beispielsweise kann dies ein Wertebereich sein, oder die Bedingung, dass zwei Instanzen einer Klasse nicht ein Attribut mit demselben Wert belegen. Außerdem können über Invarianten Regeln für die Existenz von Objekten festgelegt werden.

Pre-Condition: Muss erfüllt sein, **bevor** eine Operation ausgeführt wird.

Post-Condition: Muss **nach** Ausführung einer Operation erfüllt sein.

In OCL-Ausdrücken sind vordefinierte Typen nutzbar, zum Beispiel primitive Datentypen, Collectiontypes oder Tupel, aber auch Klassifikatoren vom UML-Modell und dessen Eigenschaften, wie Attribute oder Operationen. Darüber hinaus sind folgende Schlüsselwörter definiert:

Konditionalausdrücke: `if-then-else-endif`

Boolesche Operatoren: `not, or, and, xor, implies`

Globale Definitionen: `def`

Lokale Definitionen: `let-in`

3.2 Vor- und Nachbedingungen

In Klassendiagrammen sind nur die Syntax und Signatur einer Operation definierbar, nicht aber die Semantik. Diese ist erst mit OCL definierbar, nämlich mittels Vor- und Nachbedingungen.

Vorbedingung: Bedingungen von Argumenten und dem initialen Objektzustand müssen erfüllt werden.

Nachbedingung: Muss unter der Annahme, dass die Vorbedingung erfüllt ist, am Ende der Operationsausführung erfüllt sein. Der initiale Zustand eines Objektfeldes kann mit der Postfix-Notation `@pre` referenziert werden. Der Rückgabewert wird mit dem Schlüsselwort `result` referenziert.

In Teil 3 werden einige Beispiele vorgestellt.

4 Ereignisgesteuerte Prozessketten (EPK)

EKP sind eine weitere Methode, um Abläufe in einer Software zu modellieren. Dabei geht es jedoch weniger ins technische Detail, es stehen mehr die Abläufe im Vordergrund, anhand derer ein Endnutzer die Software bedient. Wir wenden uns hierbei sowohl der Motivation hinter der Geschäftsprozessmodellierung zu, als auch den grundlegenden Konzepten.
Prozesse werden oft nicht dokumentiert, was es mitunter schwierig macht diese nachzuvollziehen. In der Realität sind automatisierte und manuelle Schritte oft miteinander vermischt, was diese Schwierigkeiten noch weiter verstärkt. Um komplexe Prozesse dennoch durchschauen zu können, muss die Prozessanalyse **kritische** Teilprozesse und Prozessübergänge herausstellen und **Optimierungspotentiale** sichtbar machen. Da eine solche Analyse ebenfalls sehr komplex werden kann, ist eine Werkzeug- und Methodenunterstützung unabdingbar.

4.1 Grundlagen des Prozess-Engineering

In heutigen Unternehmen sind die Geschäftsprozesse von zunehmender Komplexität. Um diese Komplexität zu bewältigen, betraf es einer gesteigerten Effizienz durch Standardisierung und Automatisierung.

Geschäftsprozess: Sammlung von **Aktivitäten**, die in Unternehmen oder Verwaltung gemäß bestimmter **Regeln** und mit Hinblick auf bestimmte **Ziele** ausgeführt werden.

Beispiele für Geschäftsprozesse findet man in vielen Bereichen des Alltags:

- Geschäftsreisen
- Gebäudemanagement
- Produktion
- ...

4.2 Grundlegende Elemente der EPK-Notation

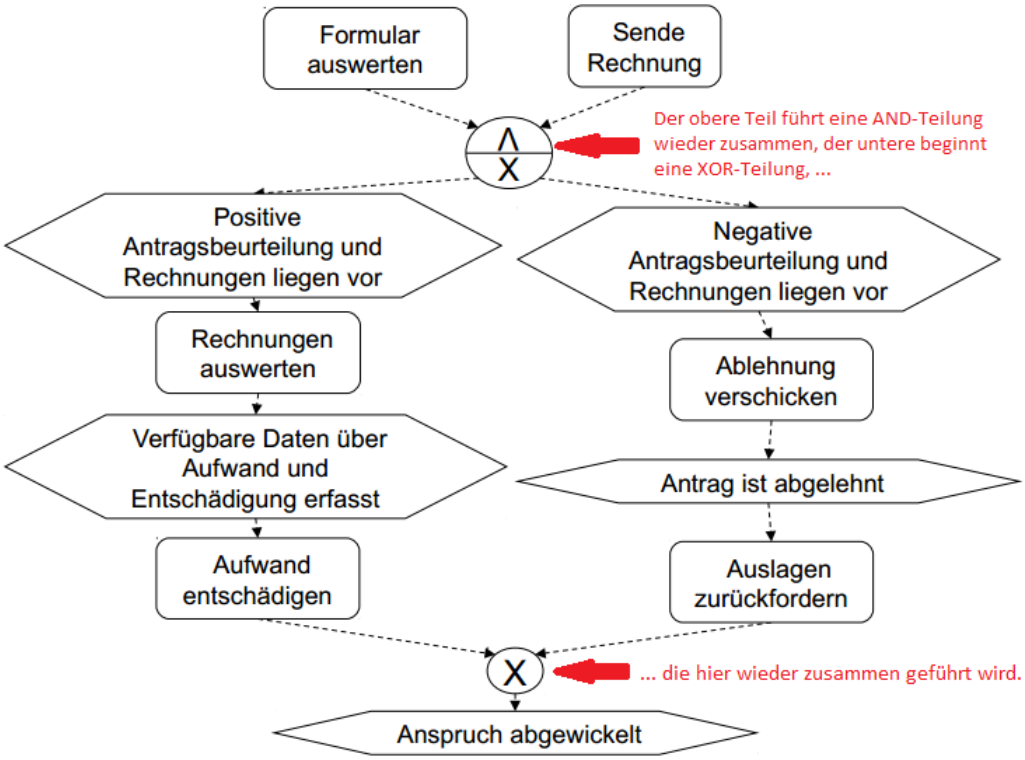
Funktionen: Funktionen sind abstrahierte atomare Aktionen in einem Geschäftsprozess, die nicht weiter teilbar sind. Dargestellt werden sie durch Rechtecke mit abgerundeten Ecken.

Ereignisse: Ereignisse sind Elemente eines Geschäftsprozesses, die entweder wegen einer Zustandsänderung eintreten oder Zustandsänderungen auslösen. Zu jeder EPK gehören Start- und Endereignis – ggf. auch mehrere. Ereignisse werden in der Notation als Sechsecke dargestellt

Kontrollfluss: Der „Durchlauf“ durch eine EPK. Die Regeln der Notation schreiben einen ständigen Wechsel zwischen Ereignissen und Funktionen vor.

Konnektoren: Konnektoren bieten die einzige Möglichkeit, Kontrollflüsse zu teilen oder zusammenzuführen. Es wird unterschieden zwischen dem AND-, OR- und XOR-Konnektor (nähere Beschreibung s. Übungen). Wird der Kontrollfluss von einem Konnektor eines Typs geteils, so kann er nur von einem Konnektor desselben Typs wieder zusammengeführt werden. Darüber hinaus kann es eine Oder-Entscheidung nach Ereignissen geben. Die Notation erlaubt es auch, eigene Konnektoren zu definieren. In der Vorlesung wurde der ET-Konnektor vorgestellt, der an eine binäre Entscheidungstabelle geknüpft ist.

Konnektoren sind auch in der Lage gleichermaßen eine Zusammenführung und eine erneute Teilung zu bewirken. Hierzu muss der jeweilige Konnektor jedoch zwei Operationen kombinieren. Dazu betrachten wir das Beispiel aus der Vorlesung:



Ressource: Notwendiges Mittel zur Ausführung einer Aufgabe. Wenn es sich dabei um eine Person handelt, so spricht man auch von *Akteuren*. Ressourcen fließen durch Ellipsen mit einem Senkrechten Strich in die Notaion ein.

Informationsobjekte: Diese werden in Geschäftsprozessen erstellt oder verändern (z.B. Dokumente). Diese Objekte werden im Modellierungskonzept unabhängig davon behandelt, ob sie physisch oder elektronisch vorliegen. Notiert werden Informationsobjekte durch einfache Rechtecke.

Unterprozesse: Unterprozesse werden durch eine Funktion unterlegt von einem Ereignis notiert. Innerhalb dieser Prozesse kann jedes Element der Notation verwendet werden. Dieses Nota-

tionselement dient der Übersichtlichkeit einer EPK. Die dahinter liegenden Prozesse werden dann in einer eigenen EPK modelliert.

5 Petrinetze

Pertrinetze sind 1962 von Adam Carl Petri entwickelt worden und dienen der Modellierung, Analyse und Simulation dynamischer Systeme mit Nebenläufigkeiten und nichtdeterministischen Merkmalen. Dabei wird insbesondere der Kontroll- und Datenfluss beschrieben.
Informell ausgedrückt besteht ein Petrinetz aus zwei Komponenten:

- Einem bipartiten gerichteten Graph, welcher wiederum zum einen aus Kanten besteht, zum anderen aus zwei Arten von Knoten:
  - *Stelle*: Zwischenablage von Informationen (dargestellt als Kreis)
  - *Transition*: Verarbeitung von Information (dargestellt als Quadrat)

  Per Definition kann es keine Kante geben, die zwei gleichartige Knoten miteinander verbindet. **(Statische Komponente)**
- Marken, die die Stellen mit Objekten belegen. Das dynamische Verhalten des Systems wird anhand des Durchlaufs der Marken durch das Netz beschrieben´. Eine von einer Marke belegte *Stelle* wird durch einen schwarzen Punkt auf der *Stelle* beschrieben. **(Dynamische Komponente)**

Wir betrachten zunächst das Beispielnetz aus der Vorlesung:

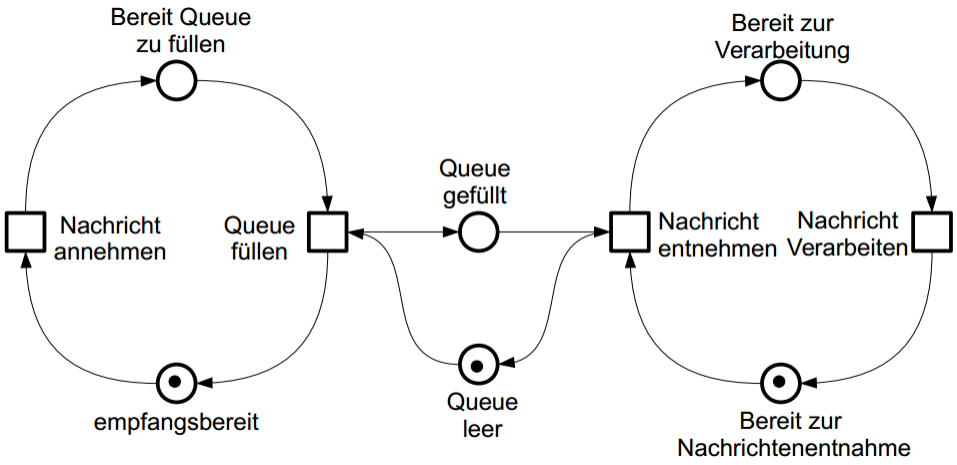


Abbildung 5.1: Die *Stellen* (**Beispiele:** Bedingungen, Medien, Nachrichtenkanäle, ...) sind mögliche lokale Zustände (passiv), auf den unteren *Stellen* wurden Marken platziert, die aussagen, dass die durch sie dargestellten Bedingungen im Moment gelten. Die *Transitionen* (**Beispiele:** Aktionen, Anweisungen, Transporte, ...) sind lokale Übergänge (aktiv). Der von den Kanten modellierte *Fluss* (**Beispiele:** Vor- und Nachbedingungen, Eingabe und Ausgabe von Programmen, ...) ist automatisch.

Bei Petrinetzen ist die Vielfachheit der Kanten zu beachten, die durch eine Zahl an der jeweiligen Kante vermerkt wird – ist die Kantenvielfachheit 1, so muss die Kante nicht beschriftet werden. Durch diesen Wert wird angegeben, wie viele Marken von der Kante konsumiert werden können.

5.1 Syntaxdefinition

Gegeben:

S: Endliche Menge von Stellen

T: Endliche Menge von Transitionen

Zu beachten: $S \neq \emptyset$, $T \neq \emptyset$ und $S \cap T \neq \emptyset$

F: Menge von Kanten mit $F \subseteq (S \times T) \cup (T \times S)$ **(binäre Relation)**. Es wird unterschieden zwischen *konsumierenden Kanten*, also jenen die von einer Stelle zu einer Transition führen, sodass sie die Marken aus der Stelle „entnehmen", sowie den *erzeugenden Kanten*, die eine Transition mit einer Stelle verbinden, also Marken zur Stelle hinzufügen.

K: Kapazität („Fassungsvermögen" der Stellen) mit $K\colon S \to \mathbb{N} \cup \{\infty\}$. Wird kein expliziter Wert angegeben, so wird davon ausgegangen, dass die Kapazität unendlich ist.

W: Kantenvielfachheit mit $W\colon F \to \mathbb{N} \setminus 0$

M_0: Globaler Startzustand („Anfangsmarkierung") mit $M_0\colon S \to \mathbb{N}$

Sind diese Voraussetzungen erfüllt, so ist (S, T, F, W, K, M, M_0) ein Petrinetz.

5.2 Ausführung

Die Ausführung eines Petrinetzes ist synonym für den Durchlauf der Marken. Der aktuelle Systemzustand wird durch die Verteilung der Marken auf die Stellen des Netzes ausgedrückt. Man spricht dabei auch von der **Markierung** der Stellen. Eine Markierung M mit $M\colon S \to \mathbb{N}$ ist zu jedem Zeitpunkt der Restriktion unterworfen, dass durch sie die Kapazität jeder einzelnen Stelle des Netzes nicht überschritten werden darf. Die initiale Markierung beschreibt den Anfangszustand eines Netzes.

5.2.1 Aktivierte Transition (Definition)

Eine Transition gilt als **aktiviert**, wenn sie die geforderte Anzahl an Marken erhalten kann und die Folgemarkierung die freigesetzten Marken aufnehmen kann, sprich:

- alle Stellen im Vorbereich der Transition besitzen ausreichend Marken
- alle Stellen im Nachbereich einer Transition haben ausreichende Kapazitäten.

Formal ist eine Transition t genau dann aktiviert, wenn

$$\forall s \in {\bullet t}\colon M(s) \geq W(s,t) \wedge \forall s' \in t{\bullet}\colon M(s') + W(t,s') \leq K(s')$$

Dabei ist ${\bullet t}$ der Vorbereich der Transition t und $t{\bullet}$ der Nachbereich der Transition t.

5.2.2 Schalten einer Transition

Bei der Ausführung eines Petrinetzes wird jeweils eine der aktivierten Transitionen von Zustand M_x nach Zustand M_{x+1} geschaltet. Dabei werden benötigte Marken auf Vorgänger-Stellen konsumiert und produzierte Marken auf Nachfolger-Stellen abgelegt. Die Anzahl der Marken orientiert sich jeweils an der Kantenvielfachheit, die Gesamtzahl der Marken im Netz kann sich verändern. Eine Folgemarkierung ist durch das Schalten von jeweils genau einer Transition erreichbar (nichtdeterministische Auswahl).

5.2.3 Erreichbarkeit

Für die Erreichbarkeit wurde in der Vorlesung folgende Notation eingeführt:

$$M[t>$$

Diese Notation sagt aus, dass die Transition t bei der Markierung M aktiviert ist. Der Ausdruck [> symbolisiert einen Pfeil. Existiert darüber hinaus eine direkte Folgemarkierung M', so wird dies folgendermaßen vermerkt:

$$M[t>M'$$

Außerdem beinhaltet die Notation folgende Elemente:

$M[w>$: Unter der Markierung M ist eine Liste w von Transitionen iterativ aktiviert. Dabei ist $w = [t_1, t_2, ..., t_n]$ und es gilt

$$M[t_1>M_1[t_2>M_2...[t_n>M_n$$

$M[\{t_1, t_2, ..., t_n\}>$: Unter der Markierung M ist eine Liste von Transitionen $[t_1, t_2, ..., t_n]$ aktiviert. Es sind also alle Permutationen als Schaltfolgen aktiviert, man spricht dabei auch von „nebenläufiger Aktivität".

$[M_0>: = \{M|\exists w \in T^* \text{ mit } M_0[w>\}$ Erreichbarkeitsmenge des Systems. Die Markierungen $M \in [M_0>$ heißen erreichbar.

Erreichbarkeitsalgorithmus

Wir benötigen ein Petrinetz als **Eingabe** und erhalten als **Ausgabe** eine Erreichbarkeitstabelle:

1. Trage ein Schema mit Spalten „Markierungsnummer", „Markierung" und „Schaltungen" die Anfangsmarkierung M_0 ein.
2. In aktueller Markierung M_i für jede Transition t aktiviert?
   - Falls t aktiviert, berechne Folgemarkierung.
     - Folgemarkierung bereits eine Markierung M_j?

- Wenn nicht: Benenne Folgemarkierung M_j (für ein neues $j > i$) und lege eine neue Zeile in der Tabelle für M_j an.

- In beiden Fällen: Trage $M_i[t > M_j$ in Zeile M_i, Spalte „Schaltungen“ ein.

3. M_i erledigt, falls alle Transitionen überprüft.

4. Alle eingetragenen Markierungen erledigt?

   Ja: Erreichbarkeitsanalyse abgeschlossen.

   Nein: Überprüfe nächste Markierung und fahre bei 2 fort.

5.3 Analyse von Systemen

Simulationen sind ein wichtiger Bestandteil der Analyse von Systemen. Durch sie kann gezeigt werden, dass bestimmte Situationen eintreten können. Es kann jedoch **nicht** gezeigt werden, dass bestimmte Situationen nicht eintreten können. Es handelt sich bei ihnen um einen Ausschnitt aus der Menge aller möglichen Verhalten.
Die Verifikation von Systemen, also der Beweis der Eigenschaften gehört ebenso zur Analyse. Die Eigenschaften von Systemen werden in zwei Arten gegliedert:

Statische Eigenschaften: Diese sind unabhängig von Markierungen, es geht dabei ausschließlich um die Topologie des Netzes.

Dynamische Eigenschaften: Sind abhängig von der Menge der erreichbaren Markierungen.

5.3.1 Sicherheit

Sei $P = (S, T, F, K, W, M_0)$ ein Petrinetz und die Abbildung $B\colon S \to \mathbb{N} \cup \{\infty\}$ ordne jeder Stelle eine kritische Markenzahl zu. Dann heißt P

- B-sicher (oder B-beschränkt), wenn für alle erreichbaren Markierungen die Anzahl der Markierungen pro Stelle durch B begrenzt wird, d.h. für alle $M \in [M_0 >$ und $s \in S$ gilt $M(s) \leq B(s)$.

- 1-sicher, 2-sicher, usw., wenn $B = 1$, $B = 2$, usw.

- beschränkt, wenn es eine natürliche Zahl b gibt, für die P b-sicher ist.

Eine Stelle s heißt b-sicher, wenn P b-sicher ist mit $B(s) = b$ und $B(s') = \infty$ für $s' \# s$.
Unterschied zwischen Kapazität und Sicherheit:

- Kapazität **begrenzt** Stellenmarkierung (a priori)

- Sicherheit **beobachtet** Stellenmarkierung (a posteriori)

5.3.2 Lebendigkeit von Transitionen

Transition t eines Petrinetzes $P = (S, T, F, K, W, M_0)$ heißt:

aktivierbar: In mindestens einer erreichbaren Markierung aktiviert, d.h. es existiert $M_1 \in [M_0 >$ mit $M_1[t >$

lebendig: In allen erreichbaren Markierungen aktivierbar, d.h. für alle $M_1 \in [M_0 >$ gilt: Existiert $M_2 \in [M_1 >$ mit $M_2[t >$

tot: In keiner erreichbaren Markierung aktiviert, d.h. für alle $M \in [M_0 >$ gilt $\neg M[t >$. Tot ist die logische Negation von aktivierbar.

5.3.3 Lebendigkeit von Petrinetzen

Ein Petrinetz $P = (S, T, F, K, W, M_0)$ heißt:

lebendig: In allen erreichbaren Markierungen ist jede Transition aktivierbar, d.h. $\forall M_1 \in [M_0 >$ und $t \in T$ gilt $\exists M_2 \in [M_1 >$ mit $M_2[t >$

deadlockfrei: In jeder erreichbaren Markierung ist mindestens eine Transition aktiviert, d.h. $\forall M_1 \in [M_0 >$ gilt $\exists t \in T$ mit $M_1[t >$

tot: Keine Transition ist aktiviert, d.h. $\forall t \in T : \neg M_0[t >$

5.4 Bewertung von Petrinetzen

Vorteile:

- Einfache und wenige Sprachelemente
- Grafisch gut darstellbar
- Marken bieten eine übersichtliche Visualisierung des Systemzustands
- Syntax und Semantik sind formal definiert
- Gut geeignet für kooperierende Prozesse

Nachteile:

- Petrinetze bieten zunächst keine Datenmodellierung, können aber dahingehend erweitert werden

6 Eclipse Modeling Foundation (EMF)

EMF ist ein Modellierungsframework und Tool zur Code-Generierung basierend auf einem strukturierten Datenmodell, welches die Grundlage für Interoperabilität zwischen EMF-basierten Anwendungen darstellt. Dabei wird von der Spezifikation in XMI ausgegangen und es stehen folgende Elemente zur Verfügung:

- Tools und Laufzeitunterstützung
- Adapterklassen, eine einfache Sicht und kommandobasiertes Editieren des Modells.
- Grundlegender Editor

EMF unterstützt darüber hinaus auch den Modellimport, also die Erzeugung von Metamodellen aus vorhandenem (Java-)Code.
Das Framework gliedert sich in vier Teile:

EMF . EMOF: *Essential MOF*, ein Teil der MOF 2.0-Spezifikation[1]. Wird genutzt, um mit Objektorientierung einfache Metamodelle zu definieren.

EMF . Ecore: Der Kern des Frameworks beinhaltet

- ein Metamodell, um Modelle zu beschrieben
- Laufzeitunterstützung für Modelle
- Persistenzunterstützung durch Standard XML-Serialisierung
- API, um EMF-Modelle generisch zu verändern

EMF . Edit: Generische wiederverwendbare Klassen, um Editoren für EMF-Modelle zu erstellen.

EMF . Codegen: EMF Codegenerieungsframework, kann den für einen Editor für EMF-Modelle benötigten Code generieren.

6.1 Was ist GEF?

Das Graphical Editing Framework wurde entwickelt, um Modelle grafisch zu erstellen und mit ihnen zu interagieren. So erfolgt die Verarbeitung von Nutzereingaben über Maus und Tastatur, es gibt Möglichkeiten das Modell zu verändern und die Änderungen zu wiederholen oder rückgängig zu machen. Es wird über ein Plugin in Eclipse eingebunden und verfolgt das Ziel wiederverwendete Funktionalitäten nicht immer wieder neu zu entwickeln.
Es basiert auf dem MVC-Pattern, was sich durch den Change-Update-Mechanismus vorteilhaft auf die Übersicht auswirkt, da das Modell stets in allen Views aktualisiert wird. Nachteilig ist allerdings, dass für dasselbe Modell mehrere View-Controller-Paare nötig sind. Bei Daten die sich oft ändern kann es außerdem passieren, dass die View die Veränderungen nicht schnell genug anzeigen kann.

[1] Es werden UML 2.0-Klassendiagramme verwendet, sodass die Modelle auch mit UML-Tools erstellt werden können.

6.2 Vor- und Nachteile von EMF und GEF

Vorteile:

- Kostengünstige Möglichkeit für modellbasierte Softwareentwicklung
- Effektivität durch automatische Konsistenzerhlatung der Modellrepräsentanten
- Mächtige Codegenerierung erspart viel stupiden Programmieraufwand

Nachteile:

- Modellierungssprachschatz nicht so mächtig wie UML, aber dennoch in den meisten Fällen ausreichend.

6.3 Bisher behandelte Standards im groben Überblick

OMG Standards

- MDA zur modellgetriebenen Softwareentwicklung
- UML und andere OMG-Modellierungsnotationen (z.B. Business Process Model and Notation (BPMN))

Eclipse Modeling Framework (EMF)

- Spezifische Realisierung der MOF-Konzepte mit Eclipse und Java
- Integriert im Eclipse Tools Project

Graphical Editing Framework (GEF)

- Framework zur Darstellung von Modellen
- Geschieht auf Basis eines EMF-Metamodells oder eigenständig

Graphical Modeling Framework (GMF)

Versuch, EMF und GEF zu integrieren

Teil II

Qualitätsmanagement

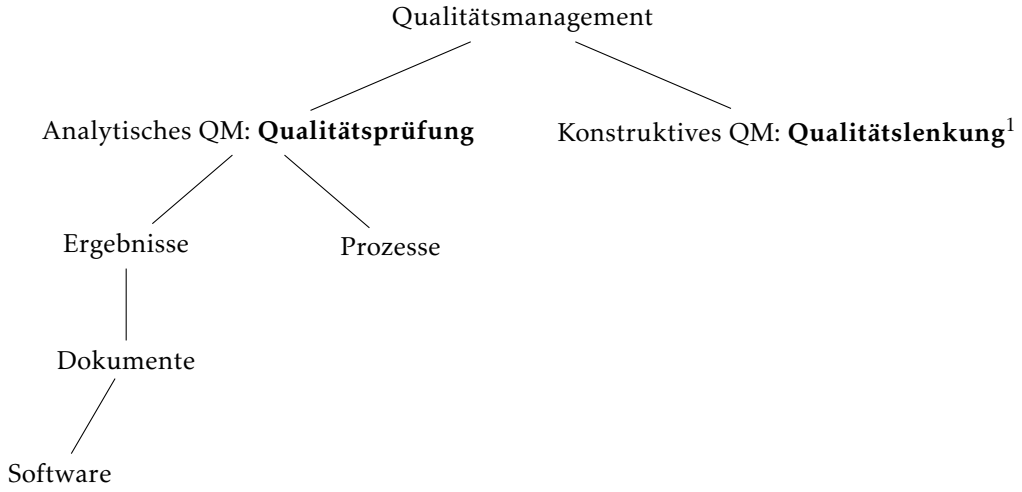
Dieser Anschnitt der Vorlesung entfernt sich von der Modellierung von Softwaresystemen und geht auf das Testen von Implementierungen ein. Wie wichtig laufende Qualitätskontrollen – auch und grade während der Entwicklungsphase – sind, zeigen diverse Beispiele die in der Vorlesung angesprochen wurden, aber hier nicht noch einmal vorgebetet werden sollten. Festzuhalten ist: Teste deine Software, sonst kommst du in die Entwickler-Hölle ☺
Softwaretests sind jedoch recht komplex. Schon bei einfachen Systemen können sehr umfangreiche Tests anstehen. Um dies zu veranschaulichen, betrachten wir zunächst das Beispiel aus der Vorlesung:

Zu testen: Ein Programm, welches 3 ganzzahlige Werte einliest und als Kantenlängen eines Dreiecks interpretiert. Dabei soll berechnet werden, um welche Art von Dreieck (ungleichseitig, gleichschenklig, gleichseitig) es sich handelt.

Idee, den Testaufwand zu begrenzen: Fallunterscheidungen.

Resümee: Allein für dieses einfache Problem bekommt man mindestens 31 Testfälle.

Es ist jedoch auch anzumerken, dass einige Qualitätsmerkmale von Software in Konflikt zu einander stehen. So konkurriert die **Effektivität** mit der **Zuverlässigkeit**, die **Sicherheit** mit der **Benutzbarkeit** und der **Effizienz**.
Neben der Qualitätsprüfung ist noch die sogenannte *Qualitätslenkung* zu erwähnen. Diese lässt sich durch die Qualität des Produktionsprozesses beeinflussen: Je effektiver dieser ist, desto Höher ist die Qualität des entwickelten Produkts. Sowohl die Qualitätsprüfung, als auch die -lenkung sind Teile des Qualitätsmanagements, wie diese in Verbindung stehen illustriert die folgende Grafik:



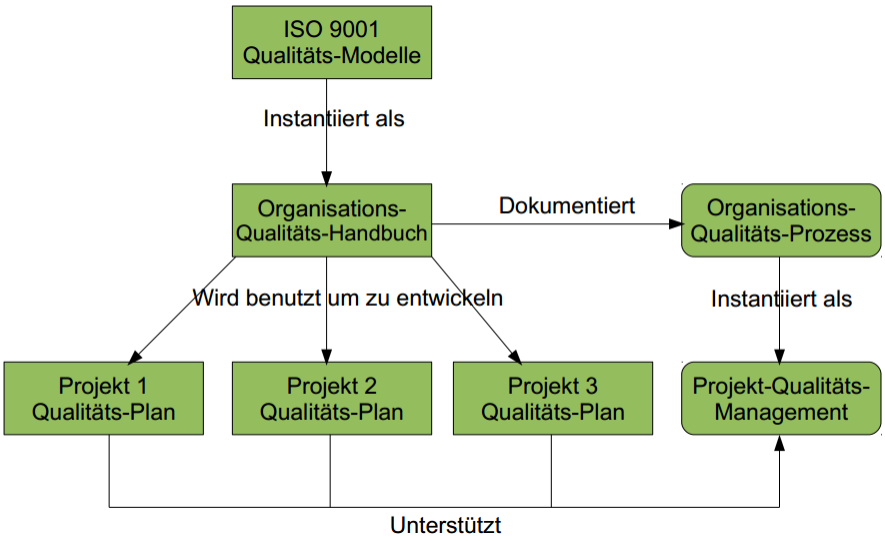
Die Qualität von Prozessen wird in der Normenreihe ISO 900x festgelegt.

ISO 9000: „Qualitätsmanagementsysteme – Grundlagen und Begriffe“

[1] Wird realisiert durch Normen, Standards, Ausbildungsmaßnahmen, Projektleitung.

- Erläutert Grundlagen für Qualitätsmanagementsysteme und in der Normenreihe verwendete Begriffe
- Erklärt den **prozessorientierten Ansatz** des Qualitätsmanagements

ISO 9001: „Qualitätsmanagement“



1 Grundlagen der Softwareverifikation

Wir wenden uns im folgenden Kapitel den Softwaretests zu. Dabei werden wir insbesondere auf die Definition von Softwarefehlern eingehen, aber auch auf verschiedene Arten von Tests. Letztere werden in späteren Kapiteln weiter vertieft, also nicht weinen, wenn das Thema an dieser Stelle ein bisschen kurz kommt.

1.1 Fehler

Zur Wiederholung widmen wir uns zu Beginn noch einmal der Frage, worin eigentlich der Unterschied zwischen Verifizierung und Validierung besteht:

Validierung: Prüfung, ob das Entwicklungsergebnis die individuellen Anforderungen bezüglich einer speziellen beabsichtigen Nutzung erfüllt. „Haben wir das **richtige System** realisiert?“

Verifizierung: Prüfung, ob die Ergebnisse einer Entwicklungsphase den Vorgaben der Phaseneingangsdokumente entsprechen. „Haben wir das System **richtig realisiert**?“

Fehler können hohe Kosten verursachen, so betragen etwa die (geschätzten) Verluste durch Softwarefehler in deutschen Mittelstands- und Großunternehmen ca. 84,4 Milliarden Euro pro Jahr. Darüber hinaus gehen die durch Computerausfälle aufgrund fehlerhafter Software hervorgerufenen Produktivitätsverluste etwa 70 Milliarden Euro im Jahr. Allgemein gilt: Je früher ein Fehler erkannt wird, desto geringer sind die durch ihn verursachten Kosten.

1.1.1 Fehler vs. Mangel

Ein **Fehler** ist

- die Nichterfüllung einer festgelegten Anforderung, bzw.
- eine Abweichung zwischen dem Ist- und dem Soll-Verhalten eines Systems.

Als **Mangel** hingegen bezeichnet man die nicht angemessene Erfüllung einer gestellten Anforderung oder einer berechtigten Erwartung in Bezug auf den beabsichtigten Gebrauch einer Anwendung.
Egal, ob Fehler oder Mangel: Eine Ursache existiert immer und er ist ab einem bestimmten Zeitpunkt der Fertigstellung in der Software vorhanden. Die meisten Fehler kommen jedoch erst bei der Ausführung der Software zum Tragen. Die Beschreibung des Sachverhalts bezeichnet man als **Fehlerwirkung** (*failure*), die Ursache einer Fehlerwirkung ist der so genannte **Fehlerzustand** (*fault*, manchmal auch als Defekt oder innerer Fehler bezeichnet). Ursache eines Fehlerzustands ist meist die **Fehlerhandlung** (*error*) einer Person. Darüber hinaus wird in IEEE 610 noch die **Fehlermaskierung** definiert, also die Verdeckung eines Fehlers durch einen anderen.

1.1.2 Definition einer Fehlerhandlung

1. Menschliche Handlung eines Entwicklers, die zu einem Fehlerzustand in der Software führt.
2. Menschliche Handlung eines Anwenders, die ein unerwünschtes Ergebnis im Sinne einer Fehlerwirkung zur Folge hat (Fehlbedienung).
3. Unwissentlich, versehentlich oder absichtlich ausgeführte Handlung oder Unterlassung, die unter gegebenen Umständen dazu führt, dass geforderte Funktionen eines Produkts beeinträchtigt sind.

1.1.3 Definition von Fehlerzustand und -wirkung

Ein **Fehlerzustand** ist ein inkorrektes Teilprogramm, welches eine Ursache für Fehlerwirkungen sein kann. Der Zustand eines Produkts oder einer seiner Komponenten, der unter spezifischen Bedingungen geforderte Funktion des Produkts beeinträchtigen kann, bzw. zu einer Fehlerwirkung führt, ist eine weitere Definition.
Eine **Fehlerwirkung** ist die Wirkung eines Fehlerzustands, die bei der Ausführung eines Programms nach „außen“ in Erscheinung tritt sowie die Abweichung zwischen einem spezifizierten Soll-Verhalten und einem aufgetretenen Ist-Verhalten.

1.2 Testen und Testfälle

Das Testen von Software kann auch als eine Messung der Qualität verstanden werden, zum Beispiel anhand der Anzahl gefundener Fehlerwirkungen. Durch Tests werden daher indirekt die Software- und Prozessqualität sowie das Vertrauen in die Qualität des Systems erhöht.
In der Praxis machen Softwaretests etwa 25 - 50 % des Entwicklungsaufwands aus. Die Intensität der Test sowie ihr Umfang werden im Abhängigkeit vom Risiko und der Kritikalität festgelegt

1.2.1 Aufbau von Testfällen

Ein Testfall besteht aus vier grundlegenden Elementen:

- Eingabewert
- Soll-Ergebnis
- Vorbedingungen (Systemzustand **vor** dem Testfall)
- Nachbedingungen (Systemzustand **nach** dem Ablauf)

Bei Ausführung eines Testfalls zeigt das Testobjekt ein Ist-Verhalten, welches mit dem Soll-Verhalten übereinstimmen sollte, sonst liegt möglicherweise eine Fehlerwirkung vor. Das eigentliche Soll-Verhalten wird von einem so genannten Testorakel bestimmt.

1.2.2 Testorakel

Ein Testorakel hat hauptsächlich drei Möglichkeiten, die Soll-Daten zu erzeugen:

Übliches Vorgehen in der Praxis: Das Soll-Datum wird aus dem Eingabedatum auf Grundlage der Spezifikation des Testobjekts abgeleitet.

Prototyp: Beispielsweise werkzeuggestützte Erzeugung durch formale Spezifikation. Es wird ein Prototyp des Testorakels für den Test des eigentlichen Programms erzeugt.

***Back-to-Back*-Test:** Ein Programm wird mehrfach von unabhängigen Entwicklungsgruppen parallel erstellt. Die einzelnen Programmversionen werden mit (identischen) Testdaten gegeneinander getestet. Kommt es dabei zu unterschiedlichen Ergebnissen, so liegt die Vermutung nahe, dass mindestens eine der Versionen fehlerhaft ist. Allerdings besteht auch das Risiko, dass auch Fehlerzustände unentdeckt bleiben, sofern sie in allen Versionen zu gleichen Fehlerwirkungen führen.

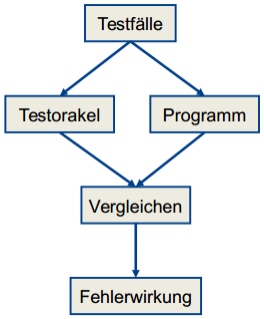


Abbildung 1.1: Dieses Schema zeigt, wo das Testorakel im Ablauf eines Tests „angesiedelt“ ist.

1.2.3 Die sieben Grundsätze des Testens

Grundsatz 1: Testen zeigt die Anwesenheit von Fehlern, jedoch nicht deren Abwesenheit!

Grundsatz 2: Vollständiges Testen ist nicht möglich.

Grundsatz 3: Mit dem Testen frühzeitig beginnen, um Fehlerzustände so früh wie möglich zu erkennen, um so die Kosten zu senken.

Grundsatz 4: Häufung von Fehlern („Fehlercluster“)

- Der Testaufwand muss proportional zur erwarteten und später beobachteten Fehlerdichte auf die Module fokussiert werden
- Die meisten Fehlerzustände stecken in einem kleinen Teil der Module
  - während der Testphase entdeckt
  - für die meisten Fehlerwirkungen im Betrieb verantwortlich

Grundsatz 5: Wiederholungen haben keine Wirksamkeit, werden Tests nur wiederholt, erhält man keine neuen Erkenntnisse. Sie müssen laufend geprüft, aktualisiert und modifiziert werden.

Grundsatz 6: Testen ist abhängig vom Umfeld, daher sollten sicherheitskritische Systeme intensiver getestet werden.

Grundsatz 7: Die Schlussfolgerung, dass jedes fehlerfreie System auch ein brauchbares System ist, ist ein Trugschluss, da das System auch dann nicht unbedingt den Vorstellungen des späteren Nutzers entspricht.

1.3 Testprozess

Die Planung und Steuerung eines Tests folgt grundlegend einem bestimmten Schema (in der Praxis jedoch nicht immer hundertprozentig):

- Analyse des Systems (anhand der Spezifikation) und Entwurf des Tests
- Testrealisierung und -druchführung
- Bewertung der Ausgangskriterien (Test-Ende-Kriterien) und Bericht
- Abschluss der Testaktivitäten

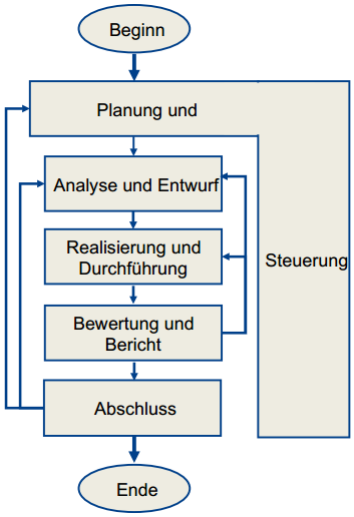


Abbildung 1.2: Die einzelnen Punkte werden zum Teil zeitlich überlappend durchgeführt. Der fundamentale Testprozess ist dabei für jede Teststufe individuell zu gestalten.

Wenden wir uns nun der Bestimmung der Komplexität von Programmen zu. Diese Bestimmung erfolgt (beispielsweise) anhand der *zyklomatischen Zahl*, die ein sehr oberflächliches Bild der Komplexität einer Software anhand des Codes wiedergibt. Die zyklomatische Zahl nach McCabe ist nur eine von vielen anwendbaren Metriken. Zum anderen kann die Komplexität einer Software auch mit Werkzeugunterstützung ermittelt werden, ob das wirklich besser ist, sei dahin gestellt.
Da ein vollständiges Testen eines Programms im Allgemeinen nicht möglich ist (vgl. zweiten Grundsatz des Testens im vorherigen Kapitel), müssen in der Praxis Prioritäten gesetzt werden, die im Idealfall auf besonders fehleranfälligen Systemteilen liegen. Was hat das alles jetzt aber mit diesem Metriken-Schrott zu tun? Die Fehleranfälligkeit einer Software korreliert mit ihrer Komplexität, welche wir anhand von Metriken ermitteln können.

2.1 Zyklomatische Komplexität

Diese Metrik („McCabe-Metrik“) misst die Komplexität eines Programms auf Basis des Kontrollflussgraphen. Die Definition der zyklomatischen Komplexität für einen Graphen G, einer Anzahl e der Kanten, einer Anzahl n der Knoten sowie einer Anzahl p der Endpunkte lautet

$$v(G) = e - n + p + 1$$

Dabei muss jedoch stets die Annahme, dass ein Programm nur einen Startpunkt hat, als Nebenbedingung berücksichtigt werden.
Nach McCabe ist eine zyklomatische Komplexität über 10 nicht tolerabel und erfordert eine Überarbeitung des Programmteils. Ein Messwert von 6 ist laut Definition akzeptabel, allerdings werden auch Messwerte im Bereich von 15 toleriert, wenn diese gut begründet sind. Für die Wartbarkeit ist die Verständlichkeit eines Programmteils wichtig. Eine hohe Komplexität kann das Nachvollziehen des Programmteils schwierig gestalten, sodass auch die Wartbarkeit darunter leidet. Wir erhalten ebenso eine Auskunft über dem Testaufwand, es gilt:

Zyklomatische Komplexität = **Anzahl unabhängiger Pfade**

Zyklomatische Komplexität – 1 = **Anzahl Entscheidungen** im Kontrollflussgraph

Die zyklomatische Komplexität ist zu dem eine obere Grenze für die Anzahl der benötigen Testfälle zur Erreichung einer hundertprozentigen Ausführung aller Anweisungen eines Programmteils.
Das große Problem dieser Metrik ist jedoch die Validität. Ein gut strukturiertes Programm kann dieselbe Komplexität haben, wie ein „Spaghetti“-Programm, welches dasselbe Problem löst. Ebenso ist sie ein schlechtes Maß für die Fehleranfälligkeit, da sie rein gar nichts über Sinn und Inhalt einer Programmzeile aussagt.

2.2 Weitere Softwaremetriken

Es folgt nun eine grobe Auflistung weiterer Softwaremetriken, die in der Vorlesung jedoch nur eine nebensächliche Rolle spielten.

Softwaremetrik	Beschreibung
Gewichtete Methoden pro Klasse (Weighted Method Complexity, WMC)	**Anzahl der Methoden in jeder Klasse**, gewichtet durch Komplexität $WMC = \sum C(i)$. Dabei sei $C(i)$ die Komplexität einer Methode i. Bei komplexen Objekten ist diese Metrik unter Umständen schwer verständlich.
Tiefe des Vererbungsbaums (Depth of Inheritance Tree, DIT)	**Maximale Tiefe der Generalisierungshierarchie**, also die Anzahl der Ebenen in einem Vererbungsbaum. Je tiefer der Baum, desto komplexer das Design (es müssen viele Klassen verstanden werden, um ein Blatt des Baums nachzuvollziehen)
Zahl der Kinder (Number of Children, NOC)	**Anzahl direkter Unterklassen.** Ein hoher NOC-Wert korreliert mit einer hohen Wiederverwendung. Wegen der hohen Zahl an Unterklassen wird eine Validierung der Basisklassen aufwändig.
Kopplung von Klassen (CBO)	Methoden von C benutzen Methoden/Variablen von D, also sind die Klassen C und D gekoppelt. CBO ist ein Maß für die Anzahl der Kopplungen. Ein hoher CBO-Wert sagt aus, dass die Klassen stark von einander abhängig sind, sodass die Änderung einer Klasse viele Klassen im Programm betrifft.
Reaktion einer Klasse (RFC)	**Maß für die Anzahl Methoden**, als Antwort auf Nachricht an umgebende Klasse ausführbar. Ein hoher RFC-Wert sagt aus, dass eine Klasse komplexer und fehleranfälliger ist.
Mangel an Zusammenhalt in Methoden (Lack of Cohesion of Methody, LCOM)	Anzahl gemeinsam genutzter **Instanzvariablen von Methoden** einer Klasse. Es handelt sich dabei um verschiedene Arten von Metriken, Informationen sind zu anderen Metriken lieferbar.

Fan-in / Fan-out	**Fan-in** einer Funktion *X* ist die Anzahl der Funktionen, die *X* aufrufen. Ist dieser Wert hoch, so ist *X* eng mit dem Rest des Systems verbunden. Änderungen an *X* können weitreichende Folgen haben. **Fan-out** ist die Anzahl der Funktionen, die von *X* aufgerufen werden. Ist dieser Wert hoch, so ist die Gesamtkomplexität von *X* hoch, da die Steuerungslogik von *X* die aufgerufenen Komponenten koordinieren muss.
Länge der Bezeichner	Maß über die durchschnittliche Länge von Bezeichnern, also die Namen von Variablen, Klassen, Methoden, usw. Je länger sie gewählt werden, desto aussagekräftiger sind sie, sodass das Programm verständlicher wird.
Tiefe der Verschachtlung	Maß der Tiefe der Verschachtlung von `if`-Bedingungen im Programm. Je tiefer sie ist, desto schwerer ist sie zu verstehen (daher auch fehleranfälliger).
Fog-Index	Maß durchschnittlicher Länge von Wörtern und Sätzen im Dokument. Je höher der Index eines Dokuments ist, desto schwerer ist es zu verstehen.

2.2.1 Einfaches Beispiel zu WMC, DIT und NOC

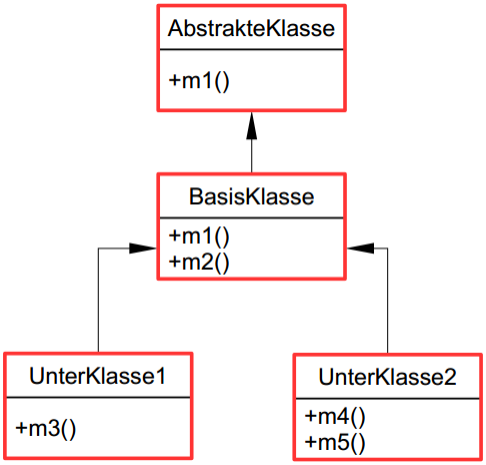


Abbildung 2.1: Wir nehmen an, dass für jede Methode i gilt, dass $C(i) = 1$

`AbstrakteKlasse`: WMC = 0, DIT = 0, NOC = 1

`BasisKlasse`: WMC = 2, DIT = 1, NOC = 2

`UnterKlasse1`: WMC = 1, DIT = 2, NOC = 0

`UnterKlasse2`: WMC = 2, DIT = 0, NOC = 0

2.2.2 Diskussionsfrage

Komplexitätsmaße sind ein Indikator für Fehleranfälligkeit und Wartbarkeit, **aber:** Welches Problem ergibt sich, wenn Programmierer am Komplexitätsmaß gemessen gemessen werden, um beide Eigenschaften zu steuern?

1. Die Komplexität des Codes entspricht der Komplexität des zu lösenden Problems. Letztere ist jedoch nur teilweise durch Programmierer steuerbar.

   Es kann allerdings ein Vergleich von mehreren Lösungen des gleichen Problems durch Metriken erfolgen. So wird die Qualität der Programmierung verglichen.

2. Wenn die Metrik bekannt ist, nach der der Programmierer bewertet wird, kann er den Code entsprechend optimieren ohne dessen Qualität zu verbessern.

Wir beschäftigen uns nun vor dem Hintergrund der Black Box Tests mit der Bildung von Äquivalenzklassen, dem zustandsbasierten sowie dem entscheidungstabellenbasierten Testen. Teilweise werden Inhalte aus SWT wiederholt, was aber auch nicht weiter schlimm sein sollte, da man im Bachelorstudium an der TU Dortmund durchaus traumatischere Fächer als SWT hat. Insbesondere handelt es sich dabei um die Unterschiede zwischen *White Box Tests*
Was bereits aus SWT bekannt sein sollte:

Black Box Tests: Die Tests werden ohne Kenntnis der Programmlogik an einem Testobjekt (z.B. einer Nutzeroberfläche) durchgeführt und das Ist-Ergebnis wird mit einem im Vorfeld bekannten Soll-Ergebnis verglichen.

White Box Tests: Wir werden uns den damit verbundenen Testverfahren im nächsten Kapitel näher widmen. Die Tests werden mit Kenntnis der Programmlogik durchgeführt. Auf diese Weise kann die Programmlogik modular (bestimmte Teile) getestet werden, um gezielter nach Fehlern zu suchen.

3.1 Äquivalenzklassenbildung

Die Grundidee sieht vor, die Definitionsbereiche der Ein- und Ausgabewerte in so genannte *Äquivalenzklassen* zu zerlegen (ÄK). Dabei gehen wir davon aus, dass alle Werte einer Klasse zu einem äquivalenten Verhalten[1] des Testobjekts führen. Die Werte der ÄK sollten aus sinnvollen Stichproben bestehen, damit die Tests bei aller Liebe zur Effektivität übersichtlich bleiben.
ÄK sollten eindeutig gekennzeichnet werden, d.h. aus ihrer Bezeichnung sollte vor allem hervorgehen, ob sie gültige oder ungültige Werte enthält. Außerdem ist es sinnvoll für jeden Parameter mindestens zwei ÄK – je eine mit gültigen, bzw. ungültigen Werten – zu definieren.
Bei n Parametern mit m_i Äquivalenzklassen ($i = 1 \,..\, n$)

$$\prod_{i=1\,..\,n} m_i \text{ Kombinationen für unterschiedliche Testfälle}$$

Bei der Bildung von ÄK für einen beliebigen Parameter muss darauf geachtet werden, dass alle **gültigen** ÄK gemeinsam den gesamten Definitionsbereich des Parameters abdecken – ist der Definitionsbereich frei von Lücken, so existiert nur eine einzige ÄK! Ungültige ÄK benötigen Eingaben jenseits des Definitionsbereichs. Werden die Wertebereiche aller ÄK aneinander gereiht, so darf es weder Lücken, noch Überschneidungen geben. Ferner werden an den Grenzen eines Definitionsbereiches auch Grenzwertanalysen durchgeführt, d.h. man sucht konkrete Werte, für die man ein konkretes Verhalten voraussagt (und entsprechend testet).

[1] Wenn ein Wert der ÄK einen Fehler aufdeckt, dann sollten alle Werte einen Fehler aufdecken und umgekehrt.

3.1.1 Beispiel (Auszug aus den Übungen)

Wir betrachten beispielhaft einen Auszug aus der ersten Aufgabe des Onlinetestats zum fünften Übungsblatt:

Artikelnummer: Die Artikelnummer sei eine fünfstellige Zahl.

Formaler notiert gilt also für eine Artikelnummer a, dass $10.000 \leq a < 100.000$, was in diesem Beispiel die einzige gültige ÄK ist. Der Definitionsbereich hat eine obere und eine untere Grenze, was uns zu zwei ungültigen ÄK führt:

1. $a < 10.000$
2. $a \geq 100.000$

Abschließend führen wir noch eine Grenzwertanalyse durch. Dazu betrachten wir die Ordnungsrelationen an den Rändern des Definitionsbereichs von a: Der linke Rand ($10.000 \leq a$) führt uns zu den Grenzwerten 9.999 (liegt außerhalb) und 10.000 (liegt innerhalb). Analog erhalten wir am rechten Rand ($a < 100.000$) die Grenzwerte 99.999 (innerhalb) und 100.000 (außerhalb).

Ein Test auf Basis von ÄK hat einen so genannten Überdeckungsgrad, der sich wie folgt bestimmen lässt:

$$\text{Überdeckungsgrad}_{\text{ÄK}} = \frac{\text{Anzahl getesteter ÄK}}{\text{Gesamtanzahl der ÄK}}$$

Jede ÄK sollte durch mindestens einen Testfall abgedeckt werden, pro Testfall gilt:

- **Mehrere gültige ÄK** für verschiedene Beschränkungen abdecken, **oder**
- **Genau eine ungültige ÄK**, einzelne Prüfung wegen Fehlermaskierung notwendig!

3.1.2 Vor- und Nachteile

Vorteile

Die Anzahl der Testfälle ist kleiner, als bei einer unsystematischen Fehlersuche. Außerdem eignen sich ÄK für Programme mit vielen Ein- und Ausgabebedingungen.

Nachteile

Es werden die Bedingungen für einzelne Ein- und Ausgabeparameter betrachtet, daher ist die Beachtung von Wechselwirkungen und Abhängigkeiten von Bedingungen sehr aufwändig.

3.2 Zustandsbasierte Tests

Bei vielen Systemen spielt der Einfluss des bisherigen Ablaufs des Systems eine wichtige Rolle bei der Berechnung nachfolgender Ausgaben. Diesem Umstand wird bei zustandsbasierten Testverfahren Rechnung getragen:

- Ein endlicher Automat besteht aus einer endlichen Anzahl von internen Konfigurationen **(Zustände)**.
- Der Zustand eines Systems beinhaltet implizit Informationen, die sich aus bisherigen Eingaben ergeben und nötig sind, um die Reaktion des Systems auf nachfolgende Eingaben zu bestimmen.
- **System:** Annahme von unterschiedlichen Zuständen beginnend vom Startzustand. (H. Balzert: Lehrbuch der Softwaretechnik, Bd. I, Spektrum, 2002)
  Die Übergänge von einem Zustand in den anderen werden durch Ereignisse, wie zum Beispiel Funktionsaufrufe, durchgeführt. Bei Zustandsänderungen sind Aktionen durchführbar, spezielle Zustände sind der **Start-** und der **Endzustand**.

3.2.1 Ziele eines zustandsbasierten Tests

- Nachweis der Konformität des Testobjekts zum Zustandsdiagramm **(Zustands-Konformanztest)**
- Zusätzlich Test unter nicht konformen Benutzungen **(Zustands-Robustheitstest)**

3.2.2 Arbeitsschritte des zustandsbasierten Testens

Ein vollständiger **zustandsbasierter Testfall** umfasst:

- Anfangszustand des Testobjekts
- Eingaben für das Testobjekt
- Erwartete Ausgaben, bzw. ein erwartetes Verhalten
- Erwarteter Endzustand

Für jeden **im Testfall erwarteten Zustandsübergang** müssen folgende Dinge festgelegt werden:

- Zustand vor dem Übergang
- Auslösendes Ereignis, das den Übergang bewirkt
- Erwartete Reaktion, ausgelöst durch den Übergang
- Nächster erwarteter Zustand

1. Erstellung des **Zustandsdiagramms**
   Beispiel, gegeben seien

   Drei Zustände:

   `empty: size() = 0;`
   `filled: 0 < size() < MAX();`
   `full: size() = MAX();`

   Zwei „Pseudo-Zustände“:

   `initial: Vor Erzeugung;`
   `final: Nach Zerstörung;`

   Acht Zustandsübergänge:

   `initial` → `empty` ; `empty` → `final`
   `empty` → `filled`; `filled` → `empty` (Zyklus!)
   `filled` → `full`; `full` → `filled` (Zyklus!)
   `filled` → `filled`; `full` → `full` (Zyklen!)

Wir erhalten somit folgendes Zustandsdiagramm:

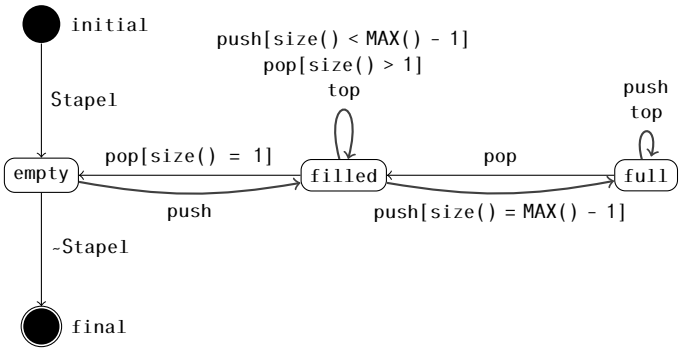


Abbildung 3.1: Die Bedingungen in den eckigen Klammern sind die so genannten Wächterbedingungen.

2. Prüfung auf **Vollständigkeit**
   Dazu legt man eine Zustandsübergangstabelle an. Insbesondere die Wächterbedingungen bezüglich eines Ereignisses sollten auf Vollständigkeit und Konsistenz geprüft werden. Ebenso sollten nicht spezifizierte Zustands-/Ereignis-Paare sollten hinterfragt werden.

3. Ableiten des **Übergangsbaums** für den **Zustands-Konformanztest**
   Der Übergangsbaum wird nach folgendem Algorithmus aufgebaut:

   (a) Anfangszustand: **Wurzel** des Baums

   (b) Für jeden möglichen **Übergang** vom Anfangszustand zum Folgezustand im Zustandsdiagramm:

   - Übergangsbaum erhält von der Wurzel aus einen **Zweig** zum Knoten: **Nachfolgezustand**
   - Notieren: Ereignis und Wächterbedingung am Zweig

   (c) Schritt (b) für jedes Blatt des Übergangsbaums wiederholen, bis eine der **Endbedingungen** eintritt:

   - Dem Blatt entsprechender Zustand: Auf „höherer Ebene“ einmal im Baum enthalten.
   - Dem Blatt entsprechender Zustand: Endzustand und hat keine weiteren Übergänge zu berücksichtigen.

   Jedes Blatt ist unabhängig von der davor liegenden Historie zu betrachten.

   Anmerkung: Zyklen werden dadurch höchstens einmal durchlaufen, garantiert einen endlichen Übergangsbaum, eine endliche Länge von Testfolgen und somit eine endliche Menge von Testfolgen.

   Wir überführen nun das Zustandsdiagramm in einen Übergangsbaum:

4. Erweitern des Übergangsbaums für den **Zustands-Robustheitstest**
   Die Robustheit wird unter spezifikationsverletzenden Benutzungen getestet. Für alle Nach-

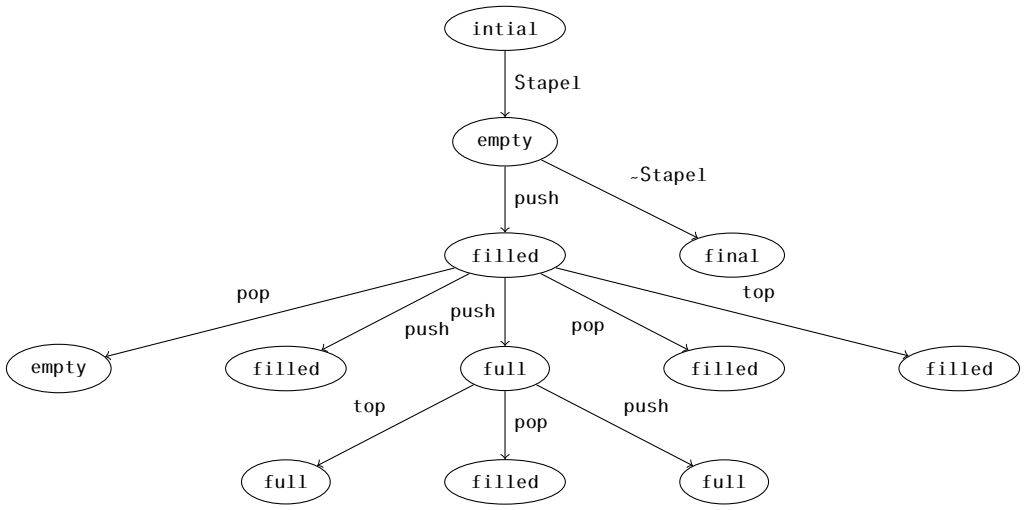
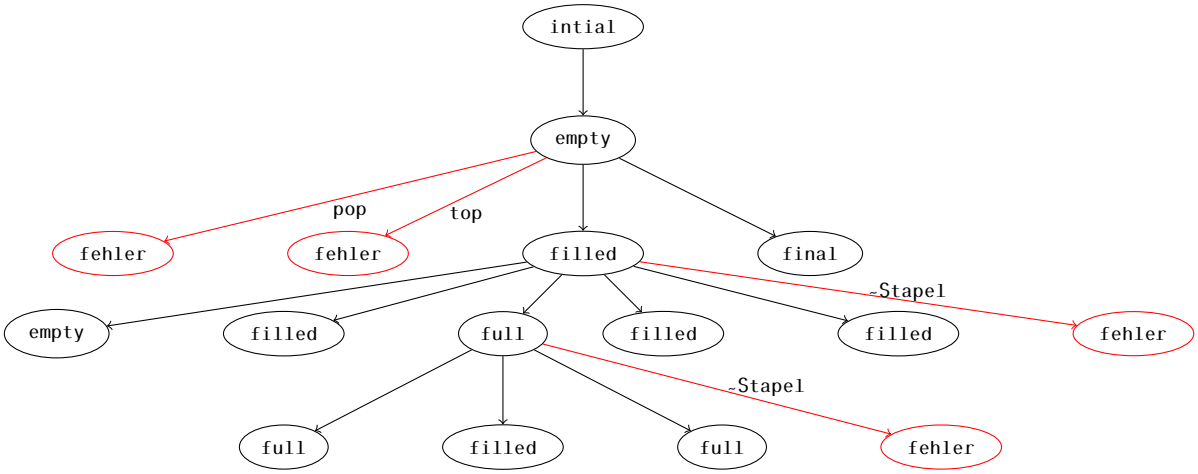


Abbildung 3.2: Wir verzichten hier auf eine Darstellung der Wächerbedingungen.

richten, für die aus dem betrachteten Knoten kein Übergang spezifiziert ist, wird der Übergangsbaum um einen neuen Fehler-Zustand erweitert.



5. Generieren der **Botschaftssequenzen** und Ergänzung der Botschaftsparameter
   Die Pfade von der Wurzel in die Blätter eines erweiterten Übergangsbaums nennt man **Funktions-Sequenzen**. Eine **Stimulierung** des Testobjekts mit entsprechenden Funktionsaufrufen deckt alle Zustände und Zustandsübergänge im Zustandsdiagramm ab. Dabei werden jedoch nicht unbedingt alle möglichen Variablenbelegungen beachtet. Man spricht in diesem Zusammenhang auch von einem **nicht kompletten** theoretisch möglichen Zustandsraum. Für Knoformanztests sind die Wächterbedingungen zu beachten – solche Konformanztests, die die Wächterbedingungen verletzen, sind sinnvoll für Robustheitstests!

6. **Ausführen der Tests** und **Überdeckungsmessung**
   Die Testfälle werden vor der **Ausführung** in ein **Testskript** gekapselt, welches unter Benut-

zung eines Testtreibers ausgeführt wird. Dabei ist dafür Sorge zutragen, dass die Zustände anhand zustandserhaltender Operationen ermittelt und protokolliert werden.

Das Minimalkriterium der **Überdeckungsmessung** lautet: Jeder Zustand wird mindestens einmal eingenommen.

$$\text{Überdeckungsgrad}_{\text{Zustand}} = \frac{\text{Anzahl getesteter Zustände}}{\text{Anzahl aller Zustände}}$$

Weitere Kriterien sind:

- Jeder Zustandsübergang wird mindestens einmal ausgeführt (Überdeckungsgrad analog definiert)
- Alle spezifikationsverletzenden Zustandsübergänge werden angeregt
- Jede Funktion wird mindestens einmal ausgeführt

Darüber hinaus muss bei hoch kritischen Anwendungen beachtet werden, dass alle Zustandsübergänge und Zyklen um Zustandsdiagramm vermerkt sind und in jeder beliebigen Reihenfolge mit allen möglichen Zuständen getestet werden (auch mehrfach hintereinander).

3.2.3 Bewertung

Wir gehen davon aus, dass ein Zustand eine Konstellation unterschiedlicher Werte von Variablen darstellt. Ein aufgespannter Zahlenraum ist sehr komplex, was eine Überprüfung einzelner Testfälle sehr aufwändig macht.
Zustandsbasierte Tests sind dort angebracht, wo die **Funktionalität** durch den jeweiligen **Zustand** des Testobjekts unterschiedlich beeinflusst wird. Dabei werden die anderen vorgestellten Testentwurfsverfahren nicht berücksichtigt, da sie nicht auf die Abhängigkeit des Verhaltens der Funktionen vom Zustand eingehen. Besonders eignen sich zustandsbasierte Tests für objektorientierte Systeme:

- Objekte können unterschiedliche Zustände annehmen.
- Methoden zur Manipulation der Objekte müssen entsprechend auf unterschiedliche Zustände reagieren.

3.3 Entscheidungstabellentest

Diese Tests sind anwendbar bei Systemanforderungen mit logischen Bedingungen sowie komplexen, vom System umzusetzenden Regeln in Geschäftsprozessen. Die Spezifikationen müssen untersucht werden, um die Eingabebedingungen und Aktionen des Systems zu ermitteln.
Eine Entscheidungstabelle enthält Kombinationen wahrer und falscher Werte für alle Eingabebedingungen sowie die daraus resultierenden Aktionen. Jede Spalte der Tabelle entspricht einer Regel im abgebildeten Geschäftsprozess, bei der eine eindeutige Kombination der Bedingungen

definiert ist. Der verwendete Standardüberdeckungsgrad bei Entscheidungstabellentests lautet: Wenigstens ein Testfall pro Spalte. Die Tabellen sind in vier Quadranten unterteilt:

Bedingungen: Mögliche Zustände von Objekten

Regeln: Kombinationen von Bedingungswerten

Aktionen: Aktivitäten. die abhängig von den Regeln auszuführen sind

Aktionszeiger: Belegungen der Bedingungen mit Aktionen

Tabelle 3.1: Im oberen linken Teil stehen die Bedingungen. Der obere rechte Teil beschreibt mit Bedingungsanzeigern die Regeln: N – nicht erfüllt, J – erfüllt, - – ohne Bedeutung. Unten links sind die Aktionen definiert, die unten rechts mit Aktionszeigern versehen sind: × – ausführen.

Textteil	**Regelteil**			
Bestellmenge > 0	N	J	J	J
Bestellmenge > Art-Lagermenge	-	J	N	N
Art-Lagermenge − Bestellmenge ≥ Art-Mindestmenge	-	-	N	J
Melde „Bestellmenge ungültig“	×			
Melde „Menge nicht ausreichend“		×		
Reduziere Lagermenge			×	×
Schreibe Nachbestellung			×	

3.3.1 Analyse

Eine Entscheidungstabelle heißt **vollständig**, wenn bei n Bedingungen alle 2^n Kombinationen enthalten sind (im oberen Teil). In **redundanzfreien** Tabellen führen verschiedene Bedingungen zu anderen Aktionen. Führen logische Beziehungen zwischen den Bedingungen zu konsistenten Aktionen, so heißt die Tabelle **widerspruchsfrei**. Aus diesen Definitionen leiten wir ab, dass Tabelle 3.1 redundanz- und widerspruchsfrei ist.

3.3.2 Bewertung

Vorteile

- Kombination von Bedingungen, die sonst nicht getestet werden
- ET-Technik zur Problemlösung anwendbar, wenn Abläufe von mehreren logischen Entscheidungen abhängen
- Logische Zusammenhänge systematisch formulierbar
- ET auf Redundanz, Widerspruchfreiheit und Vollständigkeit prüfbar
- Zwingen nicht zur Strukturierung eines Ablaufs

Nachteile

- Unübersichtlich bei vielen Bedingungen
- Zusammenhänge zwischen einzelnen Bedingungen nur implizit ausdrückbar

3.4 Allgemeine Bewertung von Black Box Tests

Allen Testverfahren liegen die Anforderungen und Spezifikationen des Systems und deren Zusammenwirken zugrunde. Fehlerhafte Anforderungen oder Spezifikationen werden von den Testverfahren nicht erkannt, da ein Testobjekt in diesem Fall fehlerhaft sein kann, auch wenn es sich der Spezifikation gemäß verhält. Außerdem ist es nicht möglich, fehlende (geforderte) Funktionen zu erkennen, wenn diese nicht spezifiziert wurden. Testfälle, die diese zusätzlichen Funktionen zur Ausführung bringen, werden nur zufällig ausgeführt. Im Mittelpunkt steht die Prüfung der Funktionalität des Testobjekts und die korrekte Funktion des Softwaresystems hat höchste Priorität, daher sind Black Box Tests unerlässlich.

Wir werden uns nun den Testverfahren zuwenden, die direkt am Code ausgeführt werden und somit die Logik des Programms testen, wobei zwischen kontroll- und datenflussbasierten Tests unterschieden wird. Dabei werden einige Themen aus SWT noch einmal kurz behandelt, aber nicht weiter vertieft.
Der White Box Test ist ein dynamisches Testverfahren, welches auf der Analyse der internen Struktur des Testobjekts basiert. Im Wesentlichen geht es dabei darum, fehleraufdeckende Stichproben möglicher Programmabläufe und Datenverwendungen zu suchen. Zur Herleitung der Testfälle und der Bestimmung der Vollständigkeit der Prüfung (Überdeckungsgrad) werden Informationen über die innere Struktur des Testobjekts herangezogen.

4.1 Kontrollflussbasierter Test

Dieses dynamische Testverfahren basiert zwar auf der Ausführung des Programms, baut aber auf einer Kenntnis der Programmstruktur auf. Der Kontrollflussgraph – das maßgebliche Analysemittel – verdeutlicht den Kontrollfluss im Programm. Die Auswahl der Testdaten muss dabei folgenden Bedingungen genügen:

- Möglichst **viele** Wege durch den Kontrollflussgraphen abdecken
- Möglichst **wenige** Testfälle gebrauchen

Kontrollflussbasierte Tests werden in folgende Varianten unterteilt:

- Art der verwendeten Kontrollflusswege oder -wegstücke
- Art und Weise der Überdeckung
- Angestrebter Überdeckungsgrad

4.1.1 Einige Begriffe

Kontrollflussgraph: Ein gerichteter Graph $G = (N, E)$ mit 2 ausgezeichneten Knoten *nstart* und *nfinal*, die die Anfangs-, bzw. Endanweisung des Programms darstellen.
Ein Knoten stellt eine Anweisung oder eine sequenzielle Anweisungsfolge dar. Eine Kante (auch Zweig genannt) aus der Menge der Kanten $E \subseteq N \times N$ beschreibt einen möglichen Kontrollfluss zwischen zwei Anweisungen.

Block: Eine nichtleere Folge von Knoten, die nur durch den ersten Knoten betretbar ist und von dort aus genau einmal deterministisch durchlaufbar ist.

Pfad/vollständiger Weg: Folge von Knoten und Kanten, die mit dem Startknoten beginnt und mit dem Endknoten aufhört.

Wege(T, P)**:** Menge vollständiger, endlicher Wege w des Kontrollflussgraphen, für die ein Testdatum t aus der Menge T der Testdaten existiert, welches den Weg w ausführt.

Entscheidungsknoten: Ein Knoten mit mindestens zwei Nachfolgeknoten.

Entscheidungskanten: Kanten, die ihren Ursprung in einem Entscheidungsknoten haben.

Zyklus: Weg im Kontrollflussgraphen mit mindestens zwei Knoten, der an demselben Knoten beginnt und endet.

Einfacher Zyklus: Zyklus, bei dem alle Knoten (außer Anfangs- und Endknoten) verschieden sind.

Entscheidungs-Entscheidungsweg: Wegstück, welches bei einem Entscheidungsknoten oder einem Anfangsknoten beginnt und alle folgenden Knoten und Kanten bis zum nächsten Entscheidungsknoten, bzw. bis zum Endknoten des Graphen enthält.

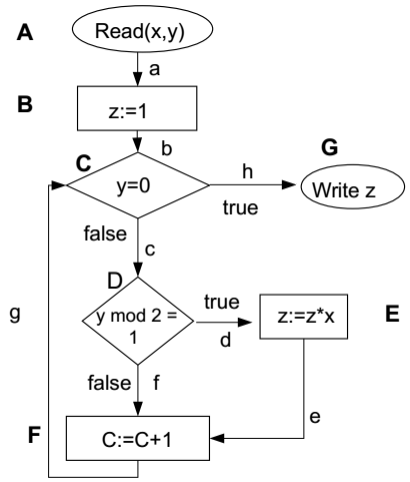


Abbildung 4.1: Dieses Beispiel aus der Vorlesung enthält folgende Entscheidungs-Entscheidungswege: $\{A,B,C\}$; $\{C,G\}$; $\{C,D\}$; $\{D,E,F,C\}$; $\{D,F,C\}$

Segment: Wegstück mit besonderen Eigenschaften:

- Erster Knoten ist der Anfangsknoten des Graphen, ein Entscheidungsknoten oder ein Vereinigungsknoten
- Letzter Knoten ist der Endknoten des Graphen, ein Entscheidungsknoten oder ein Vereinigungsknoten
- Alle anderen Knoten haben nur eine Eingangs- und Ausgangskante

In Abbildung 4.1 kommen wir auf folgende Segmente:

$$\{A,B,C\}; \{C,G\}; \{C,D\}; \{D,E,F\}; \{D,F\}; \{F,C\}$$

Zunächst werden alle Pfade durch den Kontrollflussgraphen bestimmt, die durch Testfälle zur Ausführung gebracht werden sollen. Aber mit welchen Eingaben werden diese Pfade erzwungen?

Um diese Frage zu beantworten, werden zunächst die Bedingungen an den kontrollflussbestimmenden Anweisungen betrachtet, anhand derer Aussagen über die Programmvariablen getroffen werden können.
Es ist jedoch auch zu beachten, dass auch Pfade ohne dazugehörige Testfälle erreicht werden können. Diese Pfade können nicht ausgeführt werden, weil sie zum Beispiel nicht erreichbare Programmstücke darstellen.

4.1.2 Relevante Fehlerarten

Durch kontrollflussbezogene Verfahren können bestimmte Fehler aufgedeckt werden:

Berechnungsfehler: Richtiger Kontrollflussweg im Programm ausgeführt, aber mindestens ein berechneter Variablenwert ist falsch.

Bereichsfehler: Falscher Kontrollflussweg im Programm ausgeführt, d.h. der Eingabebereich des vorliegenden Kontrollflussweges stimmt nicht mit dem Eingabebereich im korrekten Programm überein.

Unterbereichsfehler: Spezieller Bereichsfehler, „zu viel oder zu wenig“ Kontrollfluss, d.h. es fehlt eine Abfrage oder es gibt eine zusätzliche (falsche).

4.1.3 Überdeckungskriterien

Anweisungsüberdeckung: auch C_0-Überdeckung, alle Knoten

Eine Testdatenmenge T erfüllt die C_0-Überdeckung für ein Programm P genau dann, wenn es für jede Anweisung A in P mindestens ein Testdatum $t \in T$ gibt, das A ausführt. A wird unter T ausgeführt, genau dann, wenn der zu A gehörende Knoten k in mindestens einem Weg in $Wege(T)$ vorkommt. Das Testwirksamkeitsmaß TWM_0, also der Anweisungsüberdeckungsgrad wird wie folgt berechnet:

$$TWM_0 =_{\text{def}} \frac{\text{Zahl der unter } T \text{ überdeckten Anweisungen}}{\text{Zahl aller Anweisungen}}$$

Eine hundertprozentige Anweisungsüberdeckung ist nicht immer erreichbar. Es können beispielsweise Ausnahmebedingungen im Programm vor kommen, die während der Testphase mit erheblichem Aufwand oder gar nicht herzustellen sind. Außerdem besteht die Möglichkeit, dass nicht erreichbare Anweisungen im Code vorhanden sind.

Die Anweisungsüberdeckung ist nur ein shwaches Kriterium, da etwa leere Kanten, also solche, die einen oder mehrere Knoten überbrücken[1], nicht berücksichtigt werden. Folglich ist sie zwar ein notwendiges, aber kein hinreichendes Kriterium. Ein weiteres Manko ist die mangelnde Aussagekraft: Zwar werden nicht ausführbare Programmteile entdeckt, aber andere Fehler werden nur zufällig entdeckt. Für einen effektiven Programmtest ist also ein stärkeres Kriterium notwendig.

[1] else-Kanten mit leerem else-Teil, Rücksprünge zum Anfang einer repeat-Schleife, o.ä.

Entscheidungs-/Zweigüberdeckung auch C_1-Überdeckung, alle Zweige

Eine Testdatenmenge T erfüllt die C_1-Überdeckung für ein Programm P, genau dann, wenn es für jede Kante k im Kontrollflussgraphen von P mindestens einen Weg in $Wege(T, P)$ gibt, zu dem k gehört. Das Testwirksamkeitsmaß C_{Zweig} (Zweigüberdeckungsgrad) ist definiert als

$$C_{\text{Zweig}} =_{\text{def}} \frac{\text{Anzahl besuchter Zweige}}{\text{Zahl aller Zweige}}$$

Darüber hinaus gibt es hier noch ein weiteres Wirksamkeitsmaß TWM_1, bei dem im Gegensatz zu C_{Zweig} nicht die Zweige, sondern die Entscheidungskanten gezählt werden. Die Werte beider Maße können unterschiedlich sein.

$$TWM_1 = \frac{\text{Anzahl der überdeckten Entscheidungskanten}}{\text{Zahl aller Entscheidungskanten}}$$

Die Zweigüberdeckung kommt mit einer geringen Zahl von Eingaben aus und hat eine höhere Aussagekraft als C_0. Nicht-ausführbare Knoten und Zweige werden sicher entdeckt, alle anderen Fehler allerdings nur zufällig. Die durchlaufenen Programmteile sind gut erkennbar und dadurch gegebenenfalls optimierbar. Wie schon bei C_0 führt auch hier toter Code dazu, dass es selten zu einer hundertprozentigen Überdeckung kommt.

C_1 ist unzureichend zum Testen von Schleifen, besser geeignet ist die Grenze-Inneres-Überdeckung. Ebenfalls werden Abhängigkeiten zwischen den Zweigen nicht berücksichtigt. Tests komplexer, zusammengesetzter Bedingungen sind auch nicht möglich.

Mit einer hundertprozentigen Zweigüberdeckung kann man eine hundertprozentige Anweisungsüberdeckung garantieren, da beim Durchlauf aller Zweige alle Anweisungen überdeckt werden. Der Umkehrschluss ist jedoch nicht möglich!

Grenze-Inneres-Test , auch Grenze-Inneres-Überdeckung

Diese Tests beleuchten einzig und allein Schleifen zu testen. Alle Programmteile außerhalb von Schleifen werden nicht berücksichtigt. In der Praxis sollte dieses Verfahren daher höchstens als ergänzendes Kriterium verwendet werden. In einem solchen Testverlauf werden einzelne Schleifen unabhängig voneinander getestet, für verschachtelte Schleifen gibt es spezialisierte, in ihrer Stärke abgestufte Überdeckungsmaße.

Allgemein fordert die Grenze-Inneres-Überdeckung, dass jede Schleife genau einmal und mehr als einmal ausgeführt wird. Hinzu kommt ein Sonderfall für abweisende Schleifen (for und while): Es muss noch ein weiterer Testfall definiert werden, für den die Schleife gar nicht ausgeführt wird.

$$\text{GI-Überdeckungsgrad} = \frac{\text{Anzahl getesteter Schleifen}}{\text{Gesamtanzahl der Schleifen}}$$

Pfadüberdeckung auch C_∞-Überdeckung, alle Pfade

Die Pfadüberdeckung ist ein dynamisches Testentwurfsverfahren, das mindestens eine einmalige Ausführung eines jeden Pfades im Kontrollflussgraphen fordert.

$$\text{Pfadüberdeckungsgrad} = \frac{\text{Anzahl durchlaufener Pfade}}{\text{Gesamtanzahl der Pfade}}$$

Bei zyklischen Kontrollflussgraphen können potentiell unendlich viele beliebig lange Pfade entstehen, es ist jedoch gegebenenfalls eine obere Grenze für die mögliche Länge und Anzahl aus der Spezifikation oder den technischen Einschränkungen ableitbar. In der Praxis ist eine hundertprozentige Pfadüberdeckung oft nicht erreichbar, dennoch ist sie als theoretisches Vergleichsmaß wichtig.

Nichtsdestotrotz kann auch die Pfadüberdeckung nicht alle Fehler finden, weshalb ein alternatives Testparadigma als Ergänzung zu Hilfe genommen werden sollte.

Diskussionsfrage: Warum werden durch eine hundertprozentige Pfadüberdeckung nicht 100% der Fehler gefunden?

- Es werden zwar alle Pfade getestet, aber nicht mir allen möglichen Variablenbelegungen.
- Es wird nur vorhandene Funktionalitäten getestet und die, die möglicherweise noch fehlen.

4.2 Test der Bedingungen

Eine Schwäche der Zweigüberdeckung ist, dass mit bestimmten Testfällen ein richtiges von einem falschen Programm nicht unterscheiden kann. Dies kann etwa daran liegen, dass bei bedingten Anweisungen die Teilbedingungen nicht berücksichtigt werden, wie die folgende Abbildung zeigt.

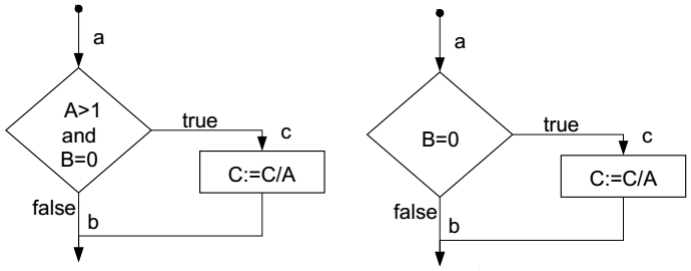


Abbildung 4.2: Es wird hier ein stärkeres Kriterium benötigt, welches auch die Teilbedingungen berücksichtigt.

In Programmen und Spezifikationen sind Bedingungen nichts anderes als Wahrheitswerte. Dabei unterscheidet man zwischen atomaren (Teil-)Bedingungen sowie zusammengesetzten Bedingungen. Atomare Bedingungen sind in der Regel Variablen vom Typ `boolean` oder Operationen mit einem Rückgabewert dieses Typs. Zusammengesetzte Bedingungen hingegen bilden eine Verknüpfung mehrerer atomarer Bedingungen. Auf beide Arten kann man Vergleichsoperatoren

anwenden. Eine (zusammengesetzte) Bedingung, die einen Programmablauf steuert, nennt man auch Entscheidung.

4.2.1 Bedingungsüberdeckung

Bedingungsüberdeckungen berücksichtigen Bedingungen in Schleifen und Auswahlstrukturen zur Definition von Tests. Die Idee dahinter ist, dass die einmalige Auswertung zusammengesetzter Bedingungen zu wahr und falsch nicht ausreichend ist, da die Teilbedingungen variieren können.

Einfache Bedingungsüberdeckung

Diese Bedingungsüberdeckung wird auch atomare Bedingungsüberdeckung oder C_2-Test genannt. Eine Testdatenmenge T erfüllt die C_2-Überdeckung genau dann, wenn es

- für jede Verzweigung im Programm mit zwei Ausgängen,
- für jeden (atomaren) Prädikatterm p des zur Verzweigung gehörenden booleschen Ausdrucks und
- zu jedem Wahrheitswert von p

ein Testdatum $t \in T$ gibt, bei dessen Ausführung p diesen Wahrheitswert annimmt. Eine Verzweigung mit mehr als zwei Ausgängen ist in ein äquivalentes Konstrukt mit zwei Ausgängen umformbar. Wir definieren das Testwirksamkeitsmaß C_2 wie folgt:

$$C_2 =_{\text{def}} \frac{\text{Anzahl wahrer Prädikate} + \text{Anzahl falscher Prädikate}}{2 \cdot \text{Anzahl Prädikate}}$$

Wie viele Testfälle benötigt man für eine einfache Bedingungsüberdeckung mit zwei Ausdrücken?
Falls eine Überdeckung erreicht werden kann, reichen zwei Testfälle – entweder $T_1 = ((0,0),(1,1))$ oder $T_2 = ((0,1),(1,0))$. Falls beide nicht erreicht werden können, weil jeweils mindestens eine Kombination nicht erreicht werden kann, bleiben ohnehin höchstens zwei Testfälle übrig.

Wie viele bei mehr als zwei Ausdrücken?
Falls eine Überdeckung erreicht werden kann, reichen n. Beweis per Induktion:

Induktionsanfang: s.o. ($n = 2$)

Induktionsschritt: Gilt für $n+1$ auf Basis von n: Die n Testfälle decken mindestens einen Wert für den $n+1$-ten Ausdruck ab. Für den anderen reicht ein weiterer Testfall.

Es ist zu beachten, dass es auch C_2-Testfälle geben kann, die nicht alle Zweige testen! Eine Testdatenmenge erfüllt die **Zweig-/Bedingungsüberdeckung** genau dann, wenn sie die C_2- **und** die C_1-Überdeckung erfüllt. Allerdings werden dann nur alle Zweige getestet, nicht alle Zweigkombinationen.

Mehrfache Bedingungsüberdeckung

Hier spricht man auch von der C_3- oder $C_2(M)$-Überdeckung. Eine Testdatenmenge erfüllt die Mehrfachbedingungsüberdeckung genau dann, wenn

- für jede Verzweigung im Programm mit zwei Ausgängen und
- für den zur Verzweigung gehörenden booleschen Ausdruck
- jede mögliche Kombination der Wahrheitswerte atomarer Prädikatterme ausgeführt wird.

Die Anzahl aller Testfälle bei n atomaren Ausdrücken lässt sich mit 2^n beziffern. Für den Überdeckungsgrad $MB\ddot{U}$ gilt

$$MB\ddot{U} = \frac{\text{Anzahl getesteter Kombinationen atomarer Ausdrücke}}{2^n}$$

Alle Kombinationen sind durch Testdaten nicht immer realisierbar. C_3-Tests weisen nicht auf Fehler in der Programmlogik hin, was die Beurteilung der Tests erschwert. Besser ist deshalb die **minimale Mehrfachbedingungsüberdeckung**.
Dabei handelt es sich um einen Kompromiss zwischen C_2 und C_3. Jedes Prädikat, egal ob atomar oder zusammengesetzt, wird zu wahr und falsch ausgewertet. Eine Testdatenmenge erfüllt die minimale Mehrfachbedingungsüberdeckung ($C_2(mM)$) genau dann, wenn

- für jede Verzweigung im Programm mit zwei Ausgängen und
- für den zur Verzweigung gehörenden booleschen Ausdruck
- folgende Kombinationen der Wahrheitswerte der enthaltenen atomaren Prädikatterme ausgeführt werden: Jede Kombination von Wahrheitswerten, für die eine Änderung des Wahrheitswertes eines Terms den gesamten Wahrheitswert ändert.

A	B	C	A ∧ B ∧ C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Abbildung 4.3: Beispiel aus den Folien (Abschnitt 2-4, Folie 82).

Eine hundertprozentige $C_2(mM)$-Überdeckung impliziert eine hundertprozentige Entscheidungsüberdeckung. Die Anzahl der Testfälle ist im Normalfall geringer als bei der Mehrfachbedingungsüberdeckung. Der Überdeckungsgrad $mMB\ddot{U}$ ist definiert als

$$mMB\ddot{U} = \frac{\text{Anzahl getesteter MM-Kombinationen}}{\text{Gesamtzahl MM-Kombinationen}}$$

4.2.2 Bewertung

Ein Problem ist die Messung der Überdeckung in Teilbedingungen. Programmiersprachen und Compiler verkürzen die Auswertung von booleschen Ausdrücken, sobald das Ergebnis feststeht. Außerdem ändern Compiler aus Effizienzgründen die Reihenfolge der Auswertung in Abhängigkeit von booleschen Operatoren. Wegen dieser Verkürzung ist eine Überdeckung oft nicht nachweisbar.

4.3 Datenflussbasierter Test

Wenn eine Anweisung ein falsches Ergebnis liefert, so kann dies zwei Ursachen haben. Entweder ist die Anweisung falsch, oder sie ist korrekt, aber referenzierte Werte werden im Vorfeld falsch berechnet.
Datenflussbasierte Tests nutzen wie alle White-Box-Verfahren die Programmstruktur aus. Die zugrunde liegende Hypothese ist die fehlerhafte Verwendung von Daten. Der Hintergedanke: Testen der Interaktion zwischen Anweisungen, die den Wert einer Variablen berechnen (definieren), und den Anweisungen, die diesen Variablenwert benutzen (referenzieren). Die verwendeten Testfälle werden unter der Berücksichtigung der Datenverwendung hergeleitet.
Das Ziel dieser Tests ist es, möglichst viele Fehler zu finden, ohne dabei eine vollständige Pfadüberdeckung erreichen zu müssen. Unterschieden werden die datenflussorientierten Verfahren anhand ihres Umfangs: Sollen alle Interaktionen getestet werden, oder nur ein Teil? Die Definition der Überdeckungsmaße orientiert sich an einem Kontrollflussgraphen, der um zusätzliche Informationen erweitert wird, dem so genannten Datenflussgraphen.

4.3.1 Definition: Datenflussgraph

zustand Wie bereits erwähnt ist ein Datenflussgraph ein Kontrollflussgraph, bei dem zu jedem Knoten k die Mengen *DEF(k)*, *UNDEF(k)* und *REF(k)* gehören.

- *DEF(k)*, Menge der Variablen x, für welche die Anweisungsfolge f, die zu k gehört, x einen Wert zuweist, der anschließend in f nicht mehr undefiniert wird.
- *UNDEF(k)*, Menge der Variablen x, für welche f die Variable x in einen undefinierten Zustand überführt, ohne x anschließend neu zu definieren.
- *REF(k)*, Menge der Variablen x, für welche f die Variable x referenziert, ohne dass x vorher in f undefiniert wird.

4.3.2 DR-Weg

Definition: Die Definition von x im Knoten k erreicht eine Referenz von x im Knoten l über den Weg w genau dann, wenn w im Graphen von k nach l führt und Variable x auf dem Weg nicht neu definiert oder undefiniert wird.

4.3.3 Datenflusskriterien

Es werden analog zu den kontrollflussbasierten Kriterien auch hier Kriterien definiert, denen eine Menge von Testpfaden genügen muss.

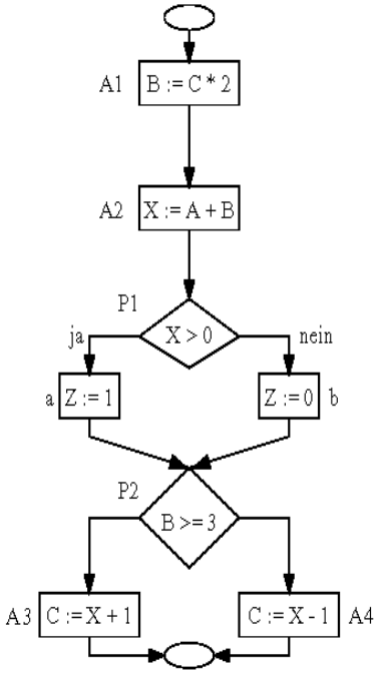


Abbildung 4.4: Wir werden uns bei Beispielen an dieser Grafik aus der Vorlesung orientieren.

Das Kriterium **„alle Definitionen“** ist die wenigstens einmalige Benutzung jeder Zuweisung. Eine Testdatenmenge T erfüllt das Kriterium genau dann, wenn für jede Variable x und jede Definition von x mindestens ein Weg in *Wege(T)* existiert, auf dem die Definition eine Referenz von x erreicht. In Abbildung 4.4 Wird das Kriterium vom Testpfad $T = (A1, A2, P1, a, P2, A3)$ erfüllt.
Ein weiteres Kriterium heißt **„alle DR-Interaktionen“**. Dessen Ziel ist es, alle Paare von Definitionen und Referenzen einer Variablen zu testen. Eine Testdatenmenge T erfüllt das Kriterium genau dann, wenn für jede Variable x, jede Definition von x und jede Referenz von x, die davon erreicht wird, mindestens ein Weg in *Wege(T)* existiert, auf dem die Definition eine Referenz von x erreicht. Der Graph in Abbildung 4.4 enthält zwei Wege, die das Kriterium erfüllen: $T_1 = (A1, A2, P1, a, P2, A3)$ und $T_2 = (A1, A2, P1, a, P2, A4)$.
Kommen wir nun zum letzten Kriterium der Vorlesung, **„alle Referenzen“**. Ziel ist es, alle ausgehenden Kanten eines Entscheidungsknotens zu berücksichtigen. Eine Testdatenmenge T erfüllt das Kriterium genau dann, wenn für jede Variable x, jede Definition von x im Knoten k, jede Referenz von x im Knoten l, die von der Definition in k erreicht wird, und für jeden Nachfolgerknoten m von l die Wegmenge *Wege(T)* mindestens ein Wegstück $u * m = \{k, \ldots, l, m\}$ enthält[2], wobei die Definition von x in k die Referenz von x in l über den Weg u erreicht. Wir betrachten nun wieder Abbildung 4.4:

[2] Hierbei: $u * m = \{k, \ldots, l, m\}$ ist die Knotenfolge mit $u = \{k, \ldots, l\}$, gefolgt vom Knoten m

- Nachfolgeknoten der Referenzen der Variable B: $(P1, A3, A4)$
  Nachfolgeknoten der Referenzen der Variable X: (a, b, end)
- Als Wege erhalten wir $T_1 = (A1, A2, P1, a, P2, A3)$ und $T_2 = (A1, A2, P1, b, P2, A4)$.

4.3.4 Variablen- und Objektverwendung

Wir haben mehrere Möglichkeiten, mit Variablen und Objekten umzugehen:

Wertzuweisung: Zustandsverändernd (definitional use), zum Beispiel `r = m` oder `r = 5`: *def(r)*

Benutzung in Ausdrücken: Zustandserhaltend (computational use), zum Beispiel `r = m mod n` oder `r = op1(m,n)`: *c-use(m,n)* und *def(r)*

Benutzung in Bedingungen: Zustandserhaltend (predicative use), zum Beispiel `while(r != 0)` oder `if(r != 0)`: *p-use(r)*

4.3.5 Bewertung

Folgende Fehler sind schlecht aufdeckbar:

- Fehlende Pfade
- Bereichsfehler durch falsch platzierte Anweisungen und falsche arithmetische Operatoren
- Berechnungsfehler bei speziellen Werten

4.3.6 Acht Irrtümer über Codeabdeckung

1. Man kann immer 100% Abdeckung erreichen.
2. Ein Abdeckungsmaß hat nur einen Namen.
3. Ein Name bezeichnet immer dasselbe Maß.
4. Es ist klar, wie Abdeckung gemessen wird.
5. Für die Abdeckung ist es egal, wie der Code formuliert ist.
6. Durch geschickte Programmierung kann man sich das Leben erleichtern.
7. Es genügt, Testfälle zur vollständigen Code-Abdeckung aus dem Code abzuleiten.
8. Codeabdeckung misst die Qualität des Codes.

4.4 Statische Analysen

Was habe wir bislang über das Testen gelernt? Zum einen, dass völliges Austesten nicht möglich ist. Für den Anteil gefundener Fehler muss eine gegenseitige Abstimmung mit dem Testaufwand gefunden werden. Kontroll- und datenflussbasierte Kriterien helfen dabei, garantieren aber keine hundertprozentige Fehlerfreiheit.
Es müssen also Verifikationsansätze gefunden werden, die nicht nur auf Ausführung des Programms beruhen. Dazu verwendet man statische Analysen, die vollautomatisch ablaufen, aber nur bestimmte Fehlerklassen aufdecken, z.B. die Analyse des Kontrollflussgraphen auf Anomalien. Ein weiterer Ansatz ist die formale Verifikation, die entweder vollautomatisch (Model-Checking) ablaufen kann und leicht bedienbar, aber nur eingeschränkt mächtig ist, oder teilautomatisiert (interaktives Theorembeweisen), was eine anspruchsvolle Bedienung mit sich bringt, aber beinahe uneingeschränkte Mächtigkeit besitzt.

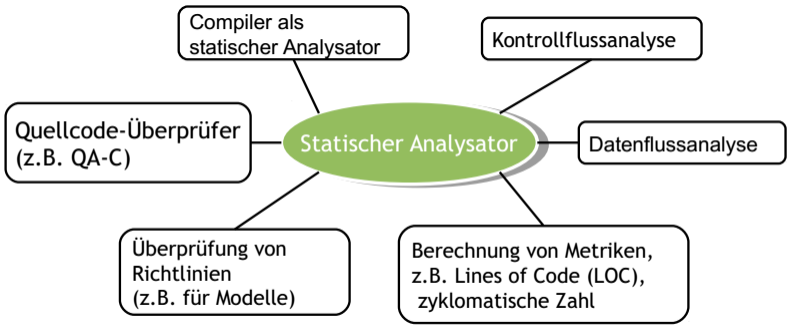


Abbildung 4.5: Einige Beispiele einer statischen Analyse.

Im Rahmen einer Kontroll- und Datenflussanalyse wird gezielt nach Anomalien im Programmtext gesucht. Eine Anomalie ist eine Unstimmigkeit, die zu einer Fehlerwirkung führen **kann**, aber nicht muss. Es sind jedoch nicht alle Fehlerzustände einfach nachweisbar.
Einfach, aber effektiv ist eine manuelle Analyse des Kontrollflussgraphen auf Anschaulichkeit. Eine bessere Anschaulichkeit macht es leichter, die Abläufe durch das Programmstück zu erfassen. Im Gegensatz dazu sind die Zusammenhänge und Abläufe in unübersichtlichen Graphen schwerer zu durchschauen.
Statisch feststellbare Unstimmigkeiten beim Ablauf eines Testobjekts nennt man **Kontrollflussanomalie**. Dazu gehören

- Sprünge aus Schleifen heraus
- Sprünge in Schleifen hinein
- Programmstücke mit mehreren Ausgängen

Dabei handelt es sich nicht zwangsweise um Fehlerzustände, aber die widersprechen den Grundsätzen strukturierter Programmierung und können fehlerträchtig sein. Den Kontrollflussgraphen

von Werkzeugen generieren zu lassen garantiert eine Abbildung, die dem Programmtext entspricht.
Zur **Analyse von Kontrollflüssen** benutzt man sogenannte Vorgänger-Nachfolger-Tabellen. Diese stellen die Beziehungen der Anweisungen dar (Ausführungsabfolge). Hat eine Anweisung keinen Vorgänger, so ist sie nicht erreichbar und ein Fehlerzustand ist erkannt. Ausnahmen in der Tabelle sind die erste und letzte Anweisung eines Programms, die logischerweise keinen Vorgänger, bzw. Nachfolger haben können, sowie Programmteile mit mehreren Eintritts- bzw. Austrittspunkten.
Die **Datenflussanalyse** basiert auf der Verwendung von Daten auf Pfaden durch das Programm. Datenflussanomalien können dabei folgende Formen haben:

- Referenzierende Verwendung einer Variablen ohne vorherige Initialisierung
- Nicht-Verwendung einer initialisierten Variablen

Wir kennen drei Datenflusszustände von Variablen (vgl. Definition: Datenflussgraph):

- Undefiniert (u)
- Definiert (d)
- Referenziert (r)

Damit einher gehen folgende Typen von Datenflussanomalien:

- ur-Anomalie: Undefinierter Wert einer Variablen wird gelesen
- du-Anomalie: Variable erhält Wert und wird ohne Verwendung ungültig
- dd-Anomalie: Eine Variable erhält einen Wert und ohne Verwendung einen zweiten Wert

Nicht jede Anomalie führt zu einem fehlerhaften Verhalten, etwa mit einer du-Anomalie kann ein Programm dennoch korrekt laufen.
Ein weiteres statisches Analyseverfahren ist die **symbolische Ausführung**. Es werden dabei symbolische statt konkreter Werte für Eingabevariablen verwendet, mit denen entsprechend gerechnet wird, zum Beispiel unter Verwendung algebraischer Eigenschaften der verwendeten Operatoren. Symbolische Tests decken eine Vielzahl normaler Testdaten ab, sodass ein vollständiges Testen prinzipiell möglich wird. Insbesondere werden Fehler entdeckt, bei denen für Teilmengen der Eingaben die Ergebnisse falsch berechnet werden. Im Gegensatz zu formaler Programmverifikation ist keine formale Programmspezifikation erforderlich. Allerdings benötigt man die formale Definition der verwendeten Programmiersprache. Bislang ist die symbolische Ausführung kein vollständiger Ersatz, sondern mehr eine Ergänzung für funktionsorientierte Tests. Während der Ausführung ist ein Theorembeweiser für die logischen Berechnungen nötig, allerdings ist kein vollständiger Theorembeweiser für mächtige Programmiersprachen vorhanden.

In diesem Kapitel liegt ein besonderer Fokus auf möglichen Strategien für Komponenten- und Integrationstests. Ein **Komponententest** ist ein systematischer Test erstellter Softwarebausteine nach der Programmierphase. Andere Bezeichnungen sind Modul-, Unit- oder Klassentests – je nachdem, wie man eine Softwareeinheit bezeichnen möchte.

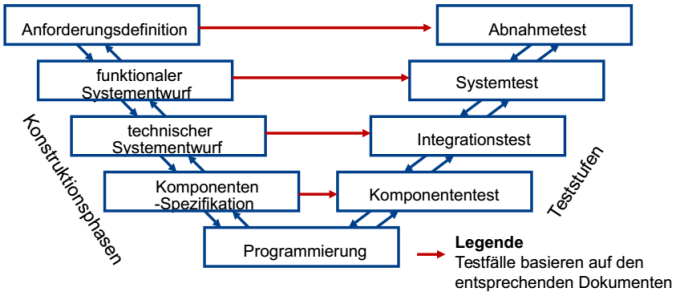


Abbildung 5.1: Aus dieser Grafik wird deutlich, wo die Komponenten- bzw. Integrationstests im V-Modell angesiedelt sind.

Die **Integrationstests** setzen voraus, dass übergebene Testobjekte bereits getestet und dabei aufgezeigte Fehlerzustände möglichst korrigiert wurden. Integration bedeutet in diesem Zusammenhang, dass der Tester eine Verbindung von Gruppen der getesteten Komponenten zu größeren Teilsystemen herstellt. Das maßgebliche Ziel ist das Aufdecken von Fehlerzuständen in Schnittstellen sowie im Zusammenspiel zwischen integrierten Komponenten. Einem Integrationstest sollte eine bestimmte **Integrationsstrategie** zugrunde liegen, in der festgelegt wird, in welcher Reihenfolge Einzelkomponenten integriert werden, damit die notwendigen Testarbeiten möglichst gering bleiben. Dabei sollte beachtet werden, dass Softwarekomponenten werden zu unterschiedlichen Zeiten fertig werden und zum Teil aus verschiedenen Projekten kommen. Wir werden im Folgenden einige Integrationsstrategien vorstellen.

5.1 Inkrementelles Testen

Bei dieser Strategie werden Schnittstellenfehler bei jedem „Dazubinden“ entdeckt, wodurch die Fehlerlokalisation vereinfacht wird. Außerdem werden zuerst ausgewählte Module gründlich getestet. Allerdings ist der Testaufwand höher, als bei nicht-inkrementellen Testen. Der Tester geht bei dieser Strategie wie folgt vor:

1. Beginn mit einem Modul X des Systems, wobei X das bisherige Teilsystem ist. Der Rest wird vorerst durch Treiber und Platzhalter ersetzt.

2. Füge ein Modul M zu bisherigem Teilsystem hinzu, wenn für M gilt:

   - M benutzt keine anderen Module oder
   - ein von M benutztes Modul gehört zum bisherigen Teilsystem oder

- *M* wird von einem Modul benutzt, das zum bisherigen Teilsystem gehört.
- Ersetze den Rest durch Treiber und Platzhalter.

3. Wiederhole Schritt 2 bis das ganze System abgedeckt ist.

Bei den Schritten 2 und 3 kann allerdings wenig parallel gearbeitet werden. Inkrementelle Integrationsstrategien können in zwei Richtungen verfolgt werden: **Top-Down** und **Bottom-Up**.

5.1.1 Top-Down-Integration

Der Test beginnt mit einer Komponente des Systems, die weitere Komponenten aufruft, dabei aber nicht selbst aufgerufen wird. Untergeordnete Komponenten werden durch Platzhalter ersetzt. Darauf aufbauend werden alle niedrigeren Systemschichten sukzessive in den Test integriert. Bereits getestete höhere Schichten werden als Treiber verwendet, was eine Stärke dieser Strategie ist. Durch die Abbildung der Ablaufumgebung durch bereits getestete Komponenten sind keine oder nur einfache Testtreiber nötig. Es kann sich jedoch als sehr aufwändig erweisen, untergeordnete und noch nicht integrierte Komponenten durch Platzhalter zu ersetzen.

5.1.2 Bottom-Up-Integration

Man beginnt mit elementaren Komponenten des Systems, die keine weiteren Komponenten aufrufen. Danach werden sukzessive größere Teilsysteme aus getesteten Komponenten unter fortlaufenden Tests der Integration zusammengesetzt. Es werden bei dieser Strategie keinerlei Platzhalter benötigt, es müssen jedoch übergeordnete Komponenten durch Testtreiber simuliert werden.

5.2 Ad-Hoc-Integration

Die Integration der Bausteine erfolgt in der Reihenfolge ihrer Fertigstellung. Nach dem Komponententest wird geprüft, ob die Komponente zu einer anderen vorhandenen und getesteten Komponente oder zu einem teilweise integrierten Subsystem passt. Wenn ja, werden bei de Teile integriert und der Integrationstest durchgeführt. Bei diesem Verfahren sind zwar Platzhalter und Treiber notwendig, es resultiert jedoch durch die frühe Integration eines Bausteins in seine passende Umgebung ein Zeitgewinn.

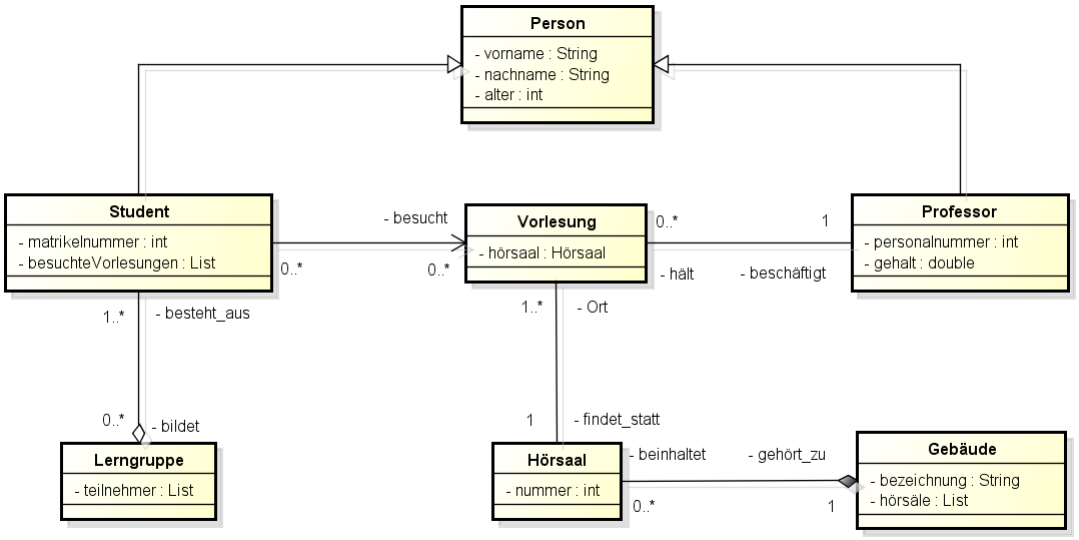
5.3 Big-Bang-Integration

Diese Strategie ist kein inkrementelles Verfahren. Nach dem alle Softwarebauteile entwickelt und getestet sind, wird alles auf einem zusammengeworfen. Im schlimmsten Fall wird dabei sogar auf vorgelagerte Komponententests verzichtet. Die Wartezeit bis zum „Big Bang“ ist natürlich verlorene Testdurchführungszeit, sodass beim letztendlichen Testen sogar ein Zeitmangel aufkommen kann. Mögliche Fehlerwirkungen treten geballt auf, was dazu führen kann, dass es lange dauern oder unter Umständen auch unmöglich sein kann, ein System zum Laufen zu kriegen. Außerdem ist dadurch die gezielte Lokalisation einzelner Fehler enorm erschwert.

Teil III

Übungsaufgaben

Aufgabe 1: UML-Klassendiagramm



Aufgabe 2: OCL-Begriffe

Klasseninvariante: Eine Bedingung, die **immer** von allen Instanzen einer Klasse erfüllt sein muss. Dies kann zum Beispiel eine Eingrenzung des Wertebereiches einer Variablen sein, oder die Bedingung, dass eine Variable in zwei Instanzen einer Klasse niemals denselben Wert annehmen darf, etwa die ID eines Datenbankeintrags. **Schlüsselwort:** `inv`

Vorbedingung: Eine Bedingung, die erfüllt sein muss, **bevor** eine Operation ausgeführt wird. Dies kann zum Beispiel ein bestimmter Wert sein, den eine Variable annehmen muss, bevor eine davon abhängige Operation ausgeführt werden kann. **Schlüsselwort:** `pre`

Nachbedingung: Eine Bedingung, die erfüllt sein muss, **nachdem** eine Operation ausgeführt wurde. Nachbedingungen sind insbesondere dann wichtig, wenn nachfolgende Aufrufe derselben oder anderer Operationen von den Ergebnissen des aktuellen Aufrufs abhängig sind. **Schlüsselwort:** `post`

Aufgabe 3: Vordefinierte Operatoren

- **logische Operatoren**

```
and, or, xor  // Konjunktion, Disjunktion, Kontravalenz (exklusives oder)
not, implies  // Negation, Implikation (Bsp: Aus Bedingung a folgt Bedingung b)
```

Wir wollen nun `implies` mithilfe der anderen logischen Operatoren ausdrücken. Nach der aus Logik und MafI1 bekannten Notation gilt für zwei Bedingungen A und B:

$$A \Rightarrow B \equiv \neg A \vee B$$

In OCL-Notation sieht dieser Ausdruck folgendermaßen aus:

```
A implies B = not A or B
```

- **Aufzählungstypen (enumeration types)**

```
variable = enumeration name :: enumeration value  // Diese besonderen Datentypen aus
     der UML erlauben die Definition von Literalen aus moegliche Werte. Der Zugriff
    auf die Werte erfolgt ueber ::
```

- **Operationen auf Objekten** (boolesche Operatoren)

```
=(o:T): Boolean   // true, wenn self gleich dem Objekt o ist
<>(o:T): Boolean  // true, wenn self ungleich dem Objekt o ist
<(o:T): Boolean   // true, wenn self kleiner als Objekt o ist
<=(o:T): Boolean  // true, wenn self kleiner gleich dem Objekt o ist
>(o:T): Boolean   // true, wenn self groesser Objekt o ist
>=(o:T): Boolean  // true, wenn self groesser gleich dem Objekt o ist
```

- **Operationen auf Collections**

1. **einfache Operationen**

```
-> size(): Integer    // Anzahl der Elemente in einer Collection
-> sum(): Real      // Summe ueber alle Elemente einer Collection
-> isEmpty(): Boolean // true, wenn Collection leer ist
-> notEmpty(): Boolean  // true, wenn Collection nicht leer ist (d.h.
    mindestens ein Element enthaelt)
```

2. **elementbezogene Operationen**

```
-> includes(o:T): Boolean // true, wenn Objekt o in Collection T enthalten ist
-> excludes(o:T): Boolean // true, wenn Objekt o nicht in Collection T
    enthalten ist
-> count(o:T): Integer    // zaehlt, wie oft o in T vorkommt
-> isUnique(expr:OclExpression) : Boolean // true, wenn der Ausdruck in den
    Klammern auf der Collection eindeutige Ergebnisse liefert
```

3. **Iterator-Ausdrücke**

```
-> select(expr:OclExpression): Collection(T)  // gibt alle Elemente einer
    Collection zurueck, fuer die der Ausdruck in den Klammern wahr ist
-> collect(expr:OclExpression): Collection(T) // sammelt alle Elemente aus
    einer oder mehreren Mengen, die die in dem Ausdruck spezifizierte Bedingung
     erfuellen. Unterschied zu select(...) ist, dass hier eine Multimenge
    zurueck gegeben werden kann
-> reject(expr:OclExpression): Collection(T)  // liefert eine Collection, die
    alle Elemente von self enthaelt, die nicht den Ausdruck in den Klammern
    erfuellen
-> forAll(expr:OclExpression): Boolean        // true, wenn Ausdruck fuer alle
    Elemente der Collection gilt
```

- **Operationen auf Klassen**

```
classifier.allInstances(): Set(T) // gibt alle Instanzen einer Klasse in einer Menge
     aus
```

Es soll nicht mehr als 10.000 Studenten geben:

```
context Student
inv: self.allInstances() -> size() <= 10000 // alle Zahlen ohne Dezimalpunkt oder -
    komma, sonst Kopf ab!
```

Aufgabe 4: Erste Anwendung von OCL

1. Das Alter eines Menschen ist immer positiv.

```
context Person
inv: self.alter > 0
```

2. Eine Veranstaltung können maximal so viele Gäste besuchen, wie der entsprechende Veranstaltungsort fassen kann.

```
context Veranstaltung
inv: self.veranstaltungsort.fassungsvermoegen >= self.besucher -> size()
```

3. Die Veranstaltungshalle *Westfalenhalle* muss für jede Veranstaltung mindestens 60 Karten für den freien Verkauf bereitstellen. Dann kann die Bedingung wie folgt ausgedrückt werden:

```
context Veranstaltungshalle
inv: self.name = 'Westfalenhalle' implies self.event -> forAll(v:Veranstaltung | v.
    Karten -> size() >= 60)
```

4. Aus wirtschaftlichen Gründen muss die *Westfalenhalle* sparen. Daher dürfen die Gehälter aller Arbeitnehmer der Westfalenhalle 60% des Budgets der Halle nicht übersteigen.

```
context Veranstaltungshalle
inv: self.name = 'Westfalenhalle' implies self.Arbeitnehmer -> collect(p:Personal |
    p.Gehalt) -> sum() <= 0.6 * self.Budget
```

5. Bei einem Flohmarkt darf es in der Veranstaltungshalle keine Bestuhlung geben.

```
context Veranstaltung
inv: self.Art = Veranstaltungsart::Flohmarkt implies self.Veranstaltungsort.
    Bestuhlung = false
```

Aufgabe 5: Mengen in OCL

1. **Bag:** Bezeichnet eine so genannte **Multimenge**, d.h. sie kann mehrere Elemente desselben Wertes enthalten.

   Set: Menge, in der jedes Element nur einmal vorkommen kann.

   OrderedSet: Wie ein Set, die Elemente sind aber geordnet.

   Sequence: Wie ein Bag, die Elemente sind aber geordnet.

2. Im Folgenden werden alle Kommentare durch führende // gekennzeichnet:

```
1 Set{1,2,3,4,5} -> sum() = 15 // Summe ueber alle Elemente
2 Set{1,2,3,4,5} -> size() = 5 // Groesse des Sets
3 Sequence{'h','a','l','l','o'} -> last() =  'o' // Letztes Element der Sequenz
4 Bag{'h','a','l','l','o'} -> last() =  undefined
5 Set{'h','a','l','l','o',' ','w','e','l','t'} -> one(e | e = 'a') = true // es kommt
    nur einmal 'a' vor
6 Set{'h','a','l','l','o',' ','w','e','l','t'} -> select(e | e = 'l') -> size() = 1 //
     keine Mehrfachvorkommen von 'l' in einem Set
7 Set{1,2,-1,-2} -> isUnique(e | e*e) = false // 1 * 1 = 1 = -1 * -1 und 2 * 2 = 4 =
    -2 * -2
```

Aufgabe 6: OCL Anwendung II

1. Manager verdienen mehr als 5.000 Euro, Angestellte und Küchenpersonal jedoch maximal 2.500 Euro.

```
context Manager inv: self.Gehaltsabfrage() > 5000.0
context Angestellter inv: self.Gehaltsabfrage() <= 2500.0
context Kuechenpersonal inv: self.Gehaltsabfrage() <= 2500.0
```

2. Jedes Ticket besitzt eine eindeutige ID.

```
context Ticket
inv: self.allInstances() -> isUnique(t | t.ID)
```

3. Es darf nur bei Flohmärkten Verkäufer geben, bei allen anderen Veranstaltungen sind alle Gäste Besucher! Zudem muss es bei Flohmärkten mindestens einen Verkäufer geben. (Lösung mit anderen Operatoren als aus Aufgabe 3)

```
context Veranstaltung
inv: if self.Art = Veranstaltungsart::Flohmarkt then self.Verkaeufer -> size() >= 1
   else self.Verkaeufer -> isEmpty()
```

   Lösung mit Operatoren aus Aufgabe 3:

```
context Veranstaltung
inv: (self.Art = Veranstaltungsart::Flohmarkt implies self.Verkaeufer -> size() >=
    1)
   and
   (not self.Art = Veranstaltungsart::Flohmarkt implies self.Verkaeufer -> isEmpty()
    )
```

4. Bei Flohmärkten dürfen pro Verkäufer in der *Westfalenhalle* nicht mehr als 5 Verkaufsstände aufgestellt werden.

```
context Veranstaltungshalle
inv: self.Name = 'Westfalenhalle' implies self.Event
-> select(e:Veranstaltung | e.Art = Veranstaltungsart::Flohmarkt)
-> forAll(e:Veranstaltung | e.Verkaeufer -> forAll(v : Verkaeufer | v.stellt_aus ->
    size() <= 5) )
```

   Betrachtet werden nur Veranstaltungen in der Wesfalenhalle, davon sind nur Flohmärkte relevant. Jeder Verkaeufer eines solchen Flohmarktes besitzt höchstens 5 Verkaufsstaende.

5. Eine Theateraufführung muss mindestens 10 Besucher haben (um nicht abgesagt zu werden)! Zudem darf dann in der Veranstaltungshalle die Bestuhlung nicht fehlen.

```
context Veranstaltung
inv: self.Art = Veranstaltungsart::Theaterauffuehrung
  implies self.Besucher -> size() >= 10 and self.Veranstaltungsort.Bestuhlung = true
```

6. Kultur soll bereits in der Jugend gefördert werden. Aus diesem Grund müssen Kinder unter 6 Jahren keinen Eintritt zu Klassikkonzerten oder Musicals zahlen, d.h. Tickets dieser Veranstaltungskategorien sind für sie kostenlos.

```
context EinzelTicket
inv: ((self.fuer.Art = Veranstaltungsart::Musical or self.fuer.Art =
    Veranstaltungsart::Klassikkonzert) and self.Besitzer.Alter < 6) implies self.
    Preis = 0.0
```

Aufgabe 7: Vordefinierte Operationen – Teil 2

- **weitere Operationen auf Collections**

  1. **boolesche Operatoren**

```
= (c:Collection(T)): Boolean    // true, wenn beide Collections identisch sind
<> (c:Collection(T)): Boolean    // true, wenn beide Collections sich in
    mindestens einem Element unterscheiden
```

  2. **Mengen-Operationen**

```
set -> union(c:Collection(T)): Collection(T)    // Die Vereinigung von einem
    Set und einer Collection liefert eine Collection zurueck. Die Set-
    Eigenschaften gehen dabei u.U. verloren.
set -> intersection(c:Collection(T)): Set(T)    // Die Schnittmenge eines Sets
    und einer Collection ist ein Set.
-> includesAll(c:Collection(T)): Boolean    // true, wenn jedes Element c der
    Collection in der zu Grunde liegenden Menge existiert
```

```
context Person
inv: self.oclIsTypeOf(Person)
inv: self.oclIsKindOf(Person)
```

```
context Student
inv: self.oclIsTypeOf(Student)
inv: self.oclIsKindOf(Person)
inv: self.oclIsKindOf(Student)
inv: not self.oclIsTypeOf(Person)
```

- **weitere Operationen auf Objekten**

  1. **Typ vs. Art**

```
self.oclIsTypeOf(typespec): Boolean    // true, wenn 'self' genau vom
    spezifizierten Typ (-> typespec) ist
self.oclIsKindOf(typespec): Boolean    // true, wenn 'self' vom spezifizierten
    Typ (-> typespec) oder einem davon abgeleiteten Typ ist (vgl.
    Klassendiagramm oben)
```

  2. **undefiniert vs. ungültig**

```
self.oclIsUndefined(): Boolean    // true, wenn 'self' 'invalid' oder 'null'
    ist
self.oclIsInvalid(): Boolean    // true, wenn 'self' 'invalid' ist
```

OCL

Aufgabe 1: OCL-Anwendung III

1. Die Anzahl von Personen eines Gruppentickets darf nicht negativ oder 0 sein!

```
context Gruppenticket
inv: self.Personen > 0
```

2. Für Tickets gelten besondere Bestimmungen: Gruppentickets werden erst ab 2 Personen ausgestellt, bei Familientickets müssen mindestens 2 Erwachsene und ein Kind gebucht werden!

```
context Gruppenticket
inv: self.Personen >= 2

context Familienticket
inv: self.Erwachsene >= 2 and Kinder >= 1
```

3. Das Personal kann sein Gehalt über Gehaltsabfrage() abfragen, dabei wird das aktuelle Gehalt zurück gegeben!

```
context Personal::Gehaltsabfrage():float
post: result = self.Gehalt
```

4. Wird das Gehalt des Personals um einen Betrag erhöht, so ist das neue Gehalt danach um genau diesen Betrag größer als das Gehalt zuvor. Damit die Gehaltserhöhung überhaupt stattfindet, muss der Betrag, um welchen das Gehalt erhöht werden soll mindestens 3% des aktuellen Gehalts betragen!

```
context Manager::erhoehtGehalt(p: Personal, Betrag: float):void
pre: Betrag >= (0.03 * p.Gehalt)
post: p.Gehalt = (p.Gehalt @pre + Betrag)
```

5. Bei dem Personal gelten die Bestimmungen, dass das Küchenpersonal über ein aktuell gültiges Gesundheitszeugnis verfügen muss.

```
context Kuechenpersonal
inv: self.Gueltigkeit_Gesundheitszeugnis >= today
```

6. Kündigt jemand vom Personal einer Veranstaltungshalle, so gehört er anschließend nicht mehr zu den Arbeitnehmern der Veranstaltungshalle. Damit derjenige überhaupt kündigen kann, muss er natürlich vorher bei dieser Veranstaltungshalle angestellt sein!

```
context Personal::kuendigt():void
pre: self.aktuellerArbeitgeber.Arbeitnehmer -> includes(self)
post: self.aktuellerArbeitgeber @pre Arbeitnehmer -> excludes(self)
```

EPK

Aufgabe 2: Grundlagen

1. Router konfigurieren: **Funktion**
   Werbebroschüre im Briefkasten: **Ereignis**
   DSL-Anschluss ist geschaltet: **Ereignis**
   Störstelle anrufen: **Funktion**
   DSL-Hardware versenden: **Funktion**

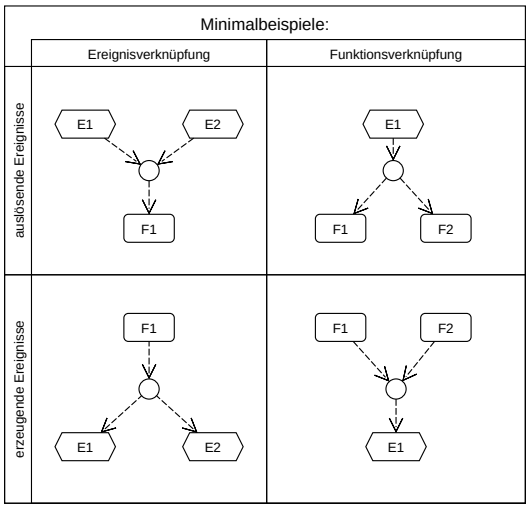
2. Konnektoren sind Notationselemente, die entweder einen Kontrollfluss aufspalten oder mehrere Kontrollflüsse zusammen führen. Es existieren drei Grundformen:

   - **AND**: Für bedingte und ggf. parallele Abläufe
   - **OR**: (Inklusives Oder) für parallele Abläufe
   - **XOR**: (Exklusives Oder) für bedingte Abläufe

   Die Konnektoren bei Teilungen und Verbindungen müssen immer zusammen passen. Wird der Kontrollfluss von einem **AND** geteilt, so muss er auch wieder durch ein **AND** zusammengefügt werden.

Aufgabe 3: Konnektoren

1. Wir beschränken uns auf die Angabe sehr allgemeiner Ketten:



2. Die Funktionsverknüpfung mit einem auslösenden Ereignis kann nur mit dem AND-Konnektor verwendet werden, denn gemäß Vorlesung kann es keine exklusive oder inklusive Entscheidung nach Ereignissen geben.

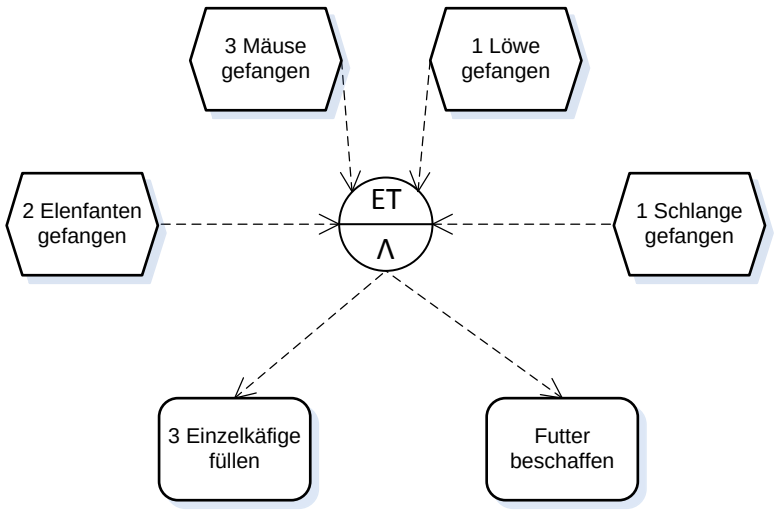
3. Ja, es ist möglich mehr als zwei Kontrollflüsse mit einem Konnektor zusammenzuführen.

Aufgabe 4: Erweiterte Konnektoren

1. Entscheidungstabelle:

Zwei Elefanten	Drei Mäuse	Ein Löwe	Eine Schlange	Drei Einzelkäfige
0	1	0	0	1
1	0	1	0	1
1	0	0	1	1

2. Modifiziertes Diagramm:



Aufgabe 5: OCL Anwendung IV

1. Bei allen Veranstaltungen sind Gruppentickets teurer als Einzeltickets.

```
context Veranstaltung
inv: self.Karten -> forAll(t1, t2: Ticket | t1.oclIsKindOf(Gruppenticket)
                        and t2.oclIsKindOf(Einzelticket) implies t1.Preis > t2.
    Preis)
```

2. Nach dem in einer Veranstaltungshalle eine Veranstaltung ausgerichtet wurde, werden vom zuständigen Manager direkt 10% der Ticketeinnahmen dem Budget der Veranstaltungshalle gutgeschrieben.

```
context Manager::richtetAus(v : Veranstaltung, d : Date):void
post: v.Veranstaltungsort.Budget = v.Veranstaltungsort.Budget @pre + v.Karten ->
    collect(t: Ticket | t.Preis) -> sum() * 0.1
```

3. Nach dem Wechsel zu einem neuen Arbeitgeber verdient ein Manager aus dem Bereich Personalwesen 10% mehr als bei seinem alten Arbeitgeber, wenn er bei dem neuen Arbeitgeber weisungsbefugt ist und in seiner neuen Abteilung mehr Mitarbeiter führen muss, als in seiner alten. Zudem befindet er sich durch die Weisungsbefugnis in der nächsthöheren Gehaltsklasse.

```
context Manager::wechseltArbeitgeber():void
post: self.Weisungsbefugt = true and self.Personalwesen = true and
      self.Mitarbeiter -> size() > self.Mitarbeiter @pre -> size() implies
      self.Gehalt = self.Gehalt @pre * 1.1 and
      self.bekleidet_Position.Gehaltsklasse =
      self.bekleidet_Position.Gehaltsklasse @pre + 1
```

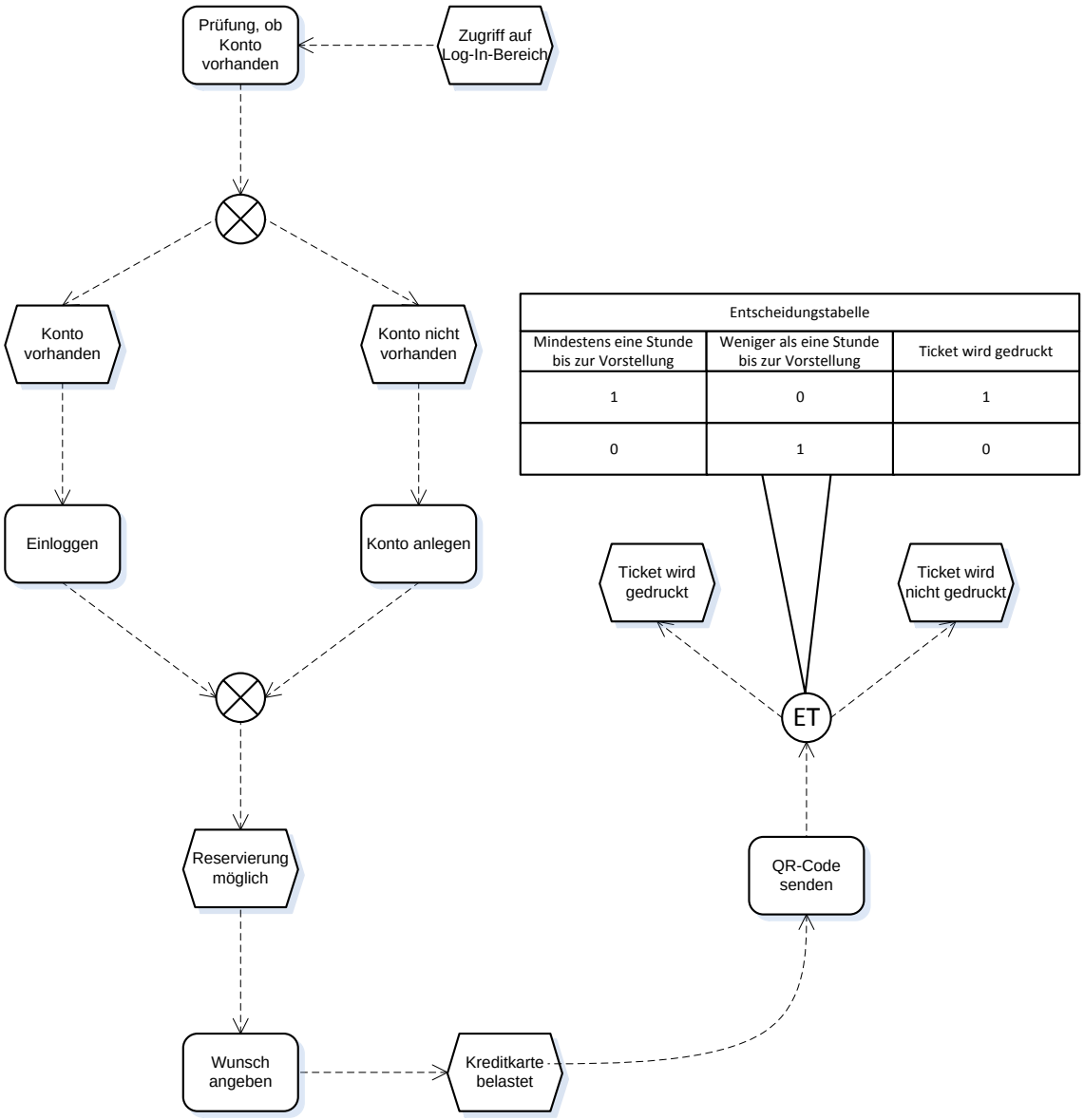
4. Die Veranstaltungshalle kann den Ticketverkauf einer Veranstaltung zu einem bestimmten Datum abfragen. Dabei wird die aktuelle Verkaufszahl zurück gegeben.

```
context Veranstaltungshalle::verkaufteTicktes(v: Veranstaltung, d: Date):Integer
post: result = v.Karten -> size()
```

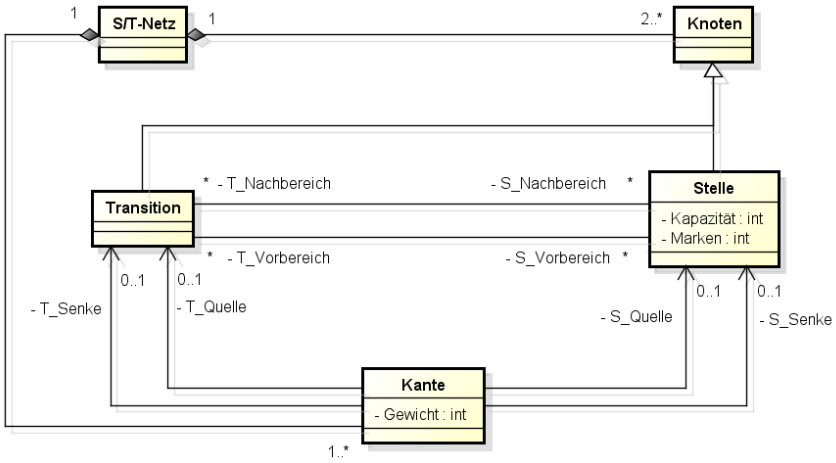
5. Um den Besuchern Abwechslung zu bieten, haben die Manager der Veranstaltungshallen beschlossen, im Jahr 2015 nicht mehr als 40 Flohmärkte und 100 Rockkonzerte auszurichten.

```
context Manager::richtetAus(v: Veranstaltung, d: Date):void
pre: ((d.year = 2015 and v.Art = Veranstatlungsart::Flohmarkt)
    implies self.verwaltet_aktuell.Event -> select(v1: Veranstaltung | v1.Art =
    Veranstaltungsart::Flohmarkt and v1.Datum.year = 2015) -> size() <= 40)
and
((d.year = 2015 and v.Art = Veranstatlungsart::Rockkonzert)
    implies self.verwaltet_aktuell.Event -> select(v2: Veranstaltung | v2.Art =
    Veranstaltungsart::Rockkonzert and v2.Datum.year = 2015) -> size() <= 100)
```

Aufgabe 6: Text zu Modell



Aufgabe 1: Petrinetze und Metamodellierung



Aufgabe 2: Metamodelle und OCL

Das Metamodell erlaubt Kanten ohne Quelle und ohne Senke sowie Kanten mit zwei Quellen und zwei Senken.

```
context Kante
inv: (self.T_Quelle -> union(self.S_Quelle) -> size() = 1) and
   (self.T_Senke -> union(self.S_Senke) -> size() = 1)
```

Das Metamodell erlaubt Quelle und Senke vom gleichen Typ.

```
context Kante
inv: (self.T_Senke -> notEmpty() implies self.S_Quelle -> notEmpty())
context Kante
inv: (self.T_Quelle -> notEmpty() implies self.S_Senke -> notEmpty())
```

Dem Metamodell fehlen Einschränkungen bezüglich Vorbereich und Nachbereich von Transitionen und Stellen.

```
context Kante
inv: self.S_Quelle -> notEmpty() implies
   (self.S_Quelle.T_Nachbereich -> includes(self.T_Senke) and
   self.T_Senke.S_Vorbereich -> includes(self.S_Quelle))
context Kante
inv: self.T_Quelle -> notEmpty() implies
   (self.T_Quelle.S_Nachbereich -> includes(self.S_Senke) and
   self.S_Senke.T_Vorbereich -> includes(self.T_Quelle))
```

Das Metamodell gibt keine Einschränkungen bezüglich Markenanzahl und Kapazität.

```
context Kante
inv: (self.S_Quelle -> union(self.T_Senke))
   -> forAll(s : Stelle | s.Marken <= s.Kapazitaet)
```

Dem Metamodell fehlt Einschränkung zum Kantengewicht.

```
context Kante
inv: self.Gewicht > 0
```

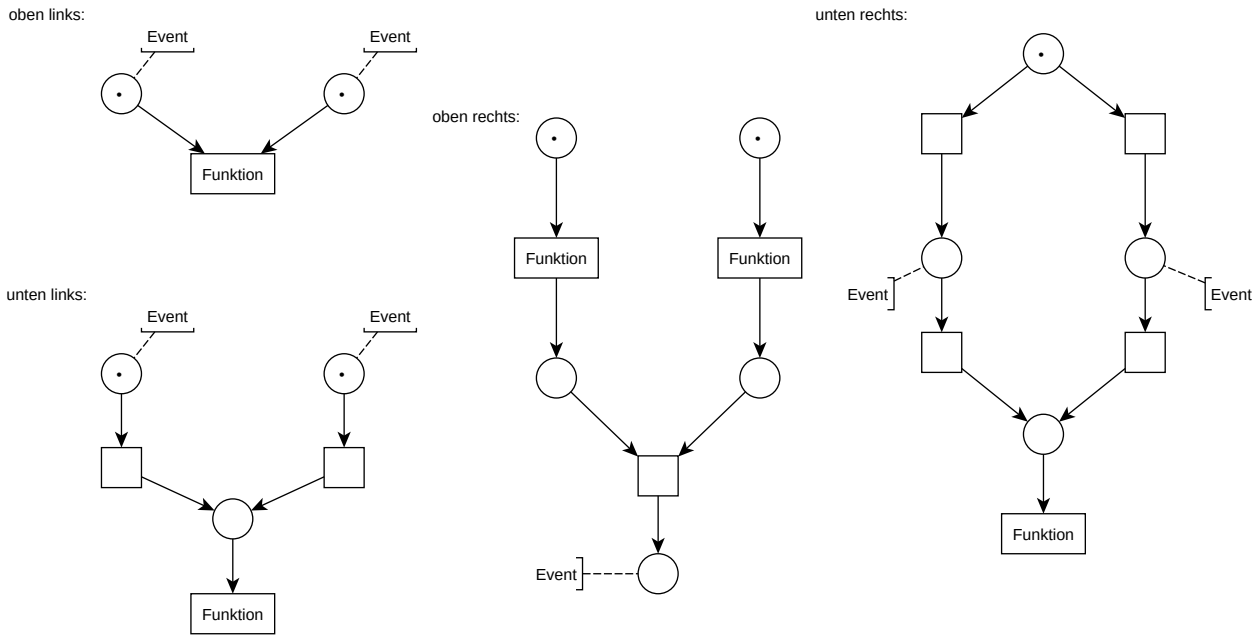
Aufgabe 3: Petrinetze – Netzeigenschaften

1. Mit umgangssprachlich kann ich dienen, ob es auch verständlich ist, ist eine andere Frage:

   - Eine Transition $t \in T$ heißt **aktivierbar im ganzen Netz:**
     Damit eine Transition als „aktivierbar“ gilt, muss das Netz eine Belegung annehmen können, in der die Transition aktiviert **ist**. Ob diese Belegung jemals angenommen wird, ist irrelevant; es muss lediglich die Möglichkeit bestehen.
   - Eine Transition $t \in T$ heißt **lebendig im ganzen Netz:**
     Wenn eine Transition in jeder erreichbaren Markierung aktivierbar (**nicht** aktiviert!) ist, so heißt sie „lebendig im ganzen Netz“.
   - Eine Transition $t \in T$ heißt **tot im ganzen Netz:**
     Wenn eine Transition in keiner erreichbaren Markierung aktiviert ist, so gilt sie als „tot im gesamten Netz“. Tot ist nicht die logische Negation von aktivierbar, nicht von lebendig!

2. Lebendige, bzw. tote S/T-Netze sind in gewisser Weise die Erweiterungen derselben Eigenschaften von Transitionen. Damit ein S/T-Netz lebendig ist, müssen in allen erreichbaren Markierungen alle Transitionen aktivierbar sein. Ist hingegen in jeder Belegung keine Transition aktiviert, so ist das Netz tot, da alle Transitionen tot sind.

Aufgabe 4: Petrinetze – Transformation



Aufgabe 5: Petrinetze – Netzeigenschaften

Aufgabenteil 1

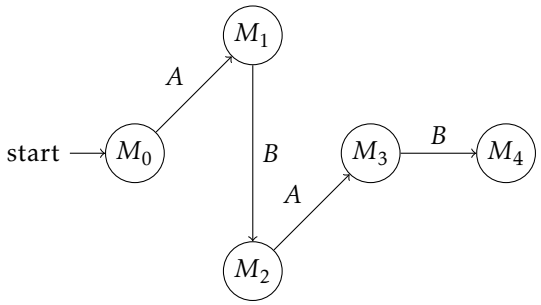
-	a)	b)	c)	d)
lebendig	nein	ja	nein	nein
deadlockfrei	ja	ja	nein	nein
tot	nein	nein	nein	nein
beschränkt	ja	nein	ja	ja
n-sicher mit n	1	-	2	1

Aufgabenteil 2

Aus dem gegebenen Petri-Netz ergibt sich folgende Erreichbarkeitstabelle:

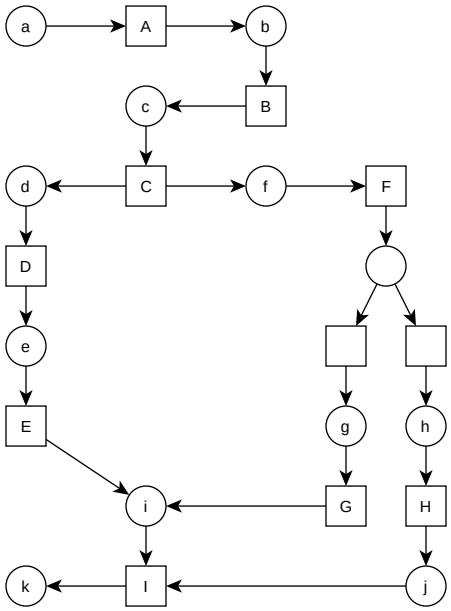
-	s_1	s_2	s_3	Schaltung(en)
M_0	1	0	0	A -> M_1
M_1	0	1	2	B -> M_2
M_2	1	0	2	A -> M_3
M_3	0	1	4	B -> M_4
M_4	1	0	4	keine Schaltung möglich

Anmerkung: Es wurde in der Vorlesung keine Syntax für Erreichbarkeitstabellen definiert!
Es ergibt sich folgender Erreichbarkeitsgraph:



Aufgabe 6: Petrinetze – Transformation

1. Nach Umwandlung der EPK ergibt sich folgendes S/T-Netz:



2. Deadlocks können im folgenden Fall auftreten:

   - Wenn eine Marke auf der Stelle nach Transition F liegt, kann nur eine der beiden nachfolgenden Transitionen schalten. Schaltet die Transition, die die Marke auf Stelle g ablegt, so haben wir keine Marke, die auf Stelle j abgelegt werden könnte.
   - Zur Aktivierung von Transition I müssen Marken auf den Stellen i **und** j liegen.
   - Der Deadlock würde also in Stelle i verursacht werden.

3. Der Deadlock kann umgangen werden, indem die Transition G die Möglichkeit bekommt, eine Marke auf Stelle j abzulegen. So kann Transition I auch aktiviert werden, wenn die „kritische Marke“ zuvor auf Stelle g abgelegt wurde.

4 Qualitätsmanagement und Metriken

Aufgabe 1: Qualitätsmanagement

1. Qualitätsstandards werden grundlegend wie folgt unterschieden. Natürlich lassen sich noch weitere (feinere) Unterscheidungen treffen.

   Produktstandards legen fest, welchen Anforderungen das fertige Produkt genügen muss. Dabei kann es sich sowohl um allgemeine Standards, als auch um firmeninterne Richtlinien handeln.

   Prozessstandards beschreiben die Herstellung der Software. Es handelt sich hier um Arbeitsabläufe bei der Entwicklung, Verwendung bestimmter Vorlagen bei der Dokumentation, etc.

2. Unterschiedliche Prüfverfahren:

   - verifizierend: Formal korrekte Beweise über die Korrektheit des Programms
   - analytisch: Kenngrößen und Metriken verwenden, um die Korrektheit zu zeigen
   - statisch: Tests ohne Programmausführung
   - dynamisch: White-Box- und Black-Box-Tests (Direkt im Code)

Aufgabe 2: Standards im Qualitätsmanagement

1. **ISO 9001:** Prozessqualitätsstandard, der allgemein Modelle zur Darstellung der Qualitätssicherung in Entwicklung, Produktion, Montage und Kundendienst beschreibt.

   CMMI: Das Capability Maturity Model Integrated gibt Hilfestellung bei der Bewertung und Verbesserung des Qualitätsmanagements.

2. Dokumentation, Überprüfbarkeit oder Transparenz von bestehenden Qualitätsprozessen: **ISO 9001**.

   Bewertung oder Verbesserung des Qualitätsmanagements: **CMMI**.

Aufgabe 3: Grundlagen Metriken

1. Fünf Beispiele für statische Metriken:

Zykomatische Komplexität	Die zyklomatische Komplexität einer Funktion betrachtet die Einfachheit des Codes. Ausgehend von der Annahme, dass die Funktion einen Eintrittspunkt hat, wird die Komplexität v für den Kontrollflussgraphen G der Funktion anhand der Anzahl e der Kanten, der Anzahl n der Knoten sowie der Anzahl p der Endpunkte berechnet: $v(G) = e - n + p + 1$
Fan-in / Fan-out	**Fan-in** einer Funktion X ist die Anzahl der Funktionen, die X aufrufen. Ist dieser Wert hoch, so ist X eng mit dem Rest des Systems verbunden. Änderungen an X können weitreichende Folgen haben. **Fan-out** ist die Anzahl der Funktionen, die von X aufgerufen werden. Ist dieser Wert hoch, so ist die Gesamtkomplexität von X hoch, da die Steuerungslogik von X die aufgerufenen Komponenten koordinieren muss.
Länge der Bezeichner	Maß über die durchschnittliche Länge von Bezeichnern, also die Namen von Variablen, Klassen, Methoden, usw. Je länger sie gewählt werden, desto aussagekräftiger sind sie, sodass das Programm verständlicher wird.
Tiefe der Verschachtlung	Maß der Tiefe der Verschachtlung von `if`-Bedingungen im Programm. Je tiefer sie ist, desto schwerer ist sie zu verstehen (daher auch fehleranfälliger).
Fog-Index	Maß durchschnittlicher Länge von Wörtern und Sätzen im Dokument. Je höher der Index eines Dokuments ist, desto schwerer ist es zu verstehen.

2. Ein grundlegendes Problem bei der Verwendung von Metriken ist die Tatsache, dass jede Metrik bestimmte Stärken und Schwächen hat. Es müssen also immer mehrere Metriken parallel verwendet werden, um eine möglichst objektives Bild der Qualität zu zeichnen. Diese Metriken müssen im Laufe der Entwicklung immer auf dem neusten Stand gehalten werden, was einen weitere Arbeitsaufwand mit sich bringt, der jedoch durch Werkzeugunterstützung minimiert werden kann.

3. Es ist schwierig, Zusammenhänge zwischen internen und externen Produkteingenschaften zu validieren, denn obwohl diese durch aus in Verbindung stehen, kann man nicht von guten internen Eigenschaften auf gute externe Eigenschaften schließen, vice versa ebenso wenig. So kann ein Code gut wartbar sein und eine optimale zyklomatische Komplexität aufweisen, aber dennoch ein falsches Ergebnis liefern, weil schlicht eine falsche Anforderung implementiert wurde.

Aufgabe 4: Zyklomatische Komplexität

Definition der zyklomatischen Komplexität für einen Graphen G, einer Anzahl e der Kanten, einer Anzahl n der Knoten sowie einer Anzahl p der Endpunkte:

$$v(G) = e - n + p + 1$$

1. Auswirkungen auf die Komplexität der Software von

   Sequenzen: Sequenzen verändern die Komplexität nicht, da die zusätzlichen Knoten immer mit zusätzlichen Kanten versehen werden müssen, sodass diese beiden Summanden in der Formel miteinander verrechnet werden.

   Verzweigungen: Verzweigungen steigern die Komplexität um 1 pro Verzweigung

   Sprünge: Sprünge verhalten sich analog zu Sequenzen

2. Kontrollflussgraph: Abbildung 4.1

Aufgabe 5: Anwendung von Metriken

1. Warum finden Softwaremetriken in der Industrie kaum Anwendung?

   - Eine Analyse des Codes anhand von Metriken treibt die *Kosten des Projekts* in die Höhe, da die Zeitaufwände sehr hoch werden können.

     Anmerkung: Für Softwareprojekte werden in der Vertriebsphase die benötigten „Stunden“ kalkuliert, also die voraussichtliche Entwicklungszeit inklusive Spezifikation, Implementierung, Tests und Inbetriebnahme. Dieses Stundenkonto wird auch als *Budget* bezeichnet. Jede Arbeit, die ein Entwickler für das Projekt leistet wird von diesem Budget abgezogen.
   - Werden *Programmteile* aus vergangenen Projekten *wiederverwendet*, so werden Anpassungen des Codes möglichst minimal gehalten, da davon ausgegangen wird, dass der *Code bereits getestet* wurde und fehlerfrei funktioniert. Würde eine Metrik genutzt werden, um das aktuelle Projekt zu bewerten, wäre das Bild verzerrt, da der wiederverwendete Code nicht unbedingt den aktuellen Richtlinien entspricht.

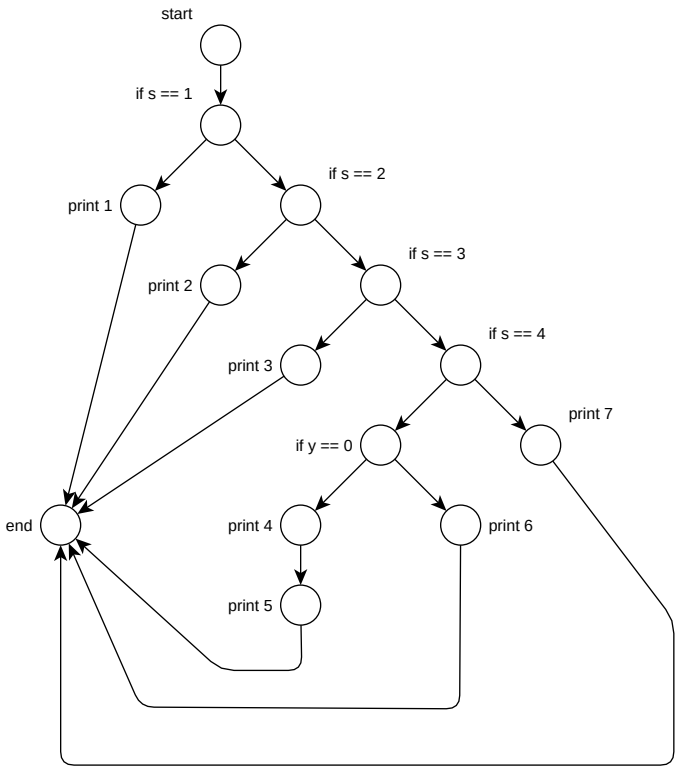


Abbildung 4.1: Die übergebene Variable `section` wurde hier mit `s` abgekürzt. Die zyklomatische Komplexität der Funktion beträgt 6.

- Die Bewertung einzelner Entwickler anhand von Metriken wird schwer, wenn ein Programmteil von einer ganzen Gruppe realisiert wird. Außerdem können einzelne Programmteile im Zuge der Entwicklung stark umgebaut werden, sodass man dafür Sorge zu tragen hätte, dass die *Bewertung durch die Metriken laufend aktualisiert* werden müsste.
- Einzelne Metriken haben Stärken und Schwächen. Für ein wirklich objektives Bild müssten *mehrere Metriken parallel* angewandt werden.

2. Wir befinden uns in der Welt der Datenbanken, also in einer Welt der Entitäten, Views und Abhängigkeiten. Außerdem nehmen wir an, dass in dieser Welt die Datenbanken objektorientiert implementiert werden, also jede Entität oder View durch Klassen repräsentiert wird. Dies führt uns zu den ersten drei Metriken:

   NOC: Über diese Metrik erhalten wir Auskunft, *wie viele Klassen von einer Entitätsklasse direkt abgeleitet sind*, sodass wir einen Index haben, der angibt, wie viele Klassen von Änderungen an der Oberklasse betroffen sein werden.

   DIT: Da NOC nur die direkten Unterklassen abbildet, führen wir noch die Metrik DIT ein, um ein „globaleres“ Bild der Vererbungshierarchie zu zeichnen. Die *Tiefe des Vererbungsbaums* gibt an, wie viele Ebenen in der Hierarchie existieren, allerdings erhalten

wir keinerlei Informationen über die Anzahl der Klassen in einer Ebene, sodass NOC und DIT zusammen verwendet werden sollten.

CBO: In Datenbanken gibt es immer Abhängigkeiten zwischen den einzelnen Entitäten. Wenn wir davon ausgehen, dass jedes Datum einer Datenbanktabelle durch ein Klassenattribut realisiert wird und andere Entitäten auf die in einer Tabelle hinterlegten Daten zugreifen, sollten diese Abhängigkeiten auch bei der Beurteilung der Softwarequalität eine Rolle spielen. Der CBO-Wert gibt an, *wie stark zwei Klassen aneinander gekoppelt sind*. Eine starke Kopplung zweier Klassen A und B sagt aus, wie viele Variablen und Methoden aus A von B genutzt werden und/oder umgekehrt.

Datenbankanwendungen müssen von Zeit zu Zeit auch gewartet und nach Release angepasst werden. Diese Aufgabe wird von der Service-Abteilung unseres Unternehmens übernommen. Da wir nicht wollen, dass die Service-Mitarbeiter den Entwicklern in Abwesenheit Kresse in die Tastatur streuen oder Essig in den Kaffee schütten, sollten wir garantieren, dass der Code gut wartbar ist. So kommen wir auf die vierte Metrik.

Länge der Bezeichner: Mit dieser Metrik kann der Code lesbar und verständlich gehalten werden, sodass auch komplexe Programmteile nicht ausführlich, sondern nur kurz und stichhaltig kommentiert werden müssen. Die Länge der Bezeichner ist das *arithmetische Mittel aller im Code vorkommenden Bezeichner*.

Zuletzt sollten wir auch den schlimmsten Feind des Entwicklers nicht außer Acht lassen: Den Kunden. Software sollte immer nutzergerecht dokumentiert werden, was die Kommunikation während der Entwicklung sowie die spätere Inbetriebnahme der Software erleichtert, sodass die fünfte Metrik angebracht wäre.

(Gunning) Fog Index: Gibt ein Bild über die (voraussichtliche) Verständlichkeit der Dokumentation des Produkts. Es handelt sich effektiv um die *Anzahl der zum Verständnis des Texts voraussichtlich notwendigen Jahre Schulbildung*. Bestimmt wird er durch:

$$0.4\left[\left(\frac{\#\text{Wörter}}{\#\text{Sätze}}\right)+100\left(\frac{\#\text{komplexe Wörter}}{\#\text{Wörter}}\right)\right]$$

Intuitiv gilt, dass ein Dokument mit niedrigem Fog Index vergleichsweise einfach verständlich ist.

Aufgabe 6: Zyklomatische Komplexität II

1. Der Code ergibt folgenden Graphen: Abbildung 4.2
2. Codebeurteilung mit anderen Metriken:
   - **Fan-in = 0**, bzw. nicht berechenbar, da kein weiterer Code angegeben ist, der die Methode aufrufen könnte.

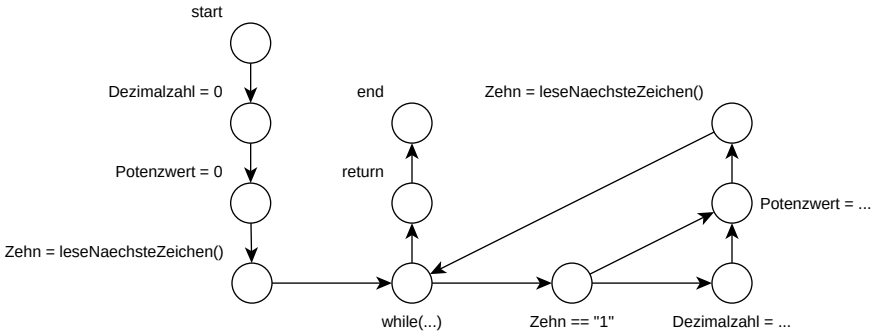


Abbildung 4.2: Wir kommen auf eine zyklomatische Komplexität von $v(G) = e - n + p + 2 = 12 - 11 + 2 = 3$ (Formel in der Vorlesung definiert).

Der Fan-in-Wert sagt aus, wie häufig im System eine Methode oder Funktion von anderen Methoden oder Funktionen aufgerufen wird.
Fan-out = 2 wegen der Methoden `leseNaechsteZeichen()` und `Math.pow(..)`.
Der Fan-out-Wert sagt aus, wie viele Funktionen oder Methoden von der betrachteten Funktion oder Methode aufgerufen werden.
Die Bestimmung beider Werte macht nur in großen Softwaresystemen Sinn, da aus ihnen hervor geht, wie wichtig eine Funktion oder Methode für die Funktionalität eines Programmteils (oder des gesamten Systems) ist. Im vorliegenden Fall könnte man also auch darauf verzichten.

- **NCSS:** Non-commented source statements
  - Zählung der Codezeilen ohne Kommentar- und Leerzeilen
  - Genaue Zählregeln erforderlich
  - Abhängig von Programmiersprache
  - Leicht messbar

  Wir definieren zunächst unsere Zählregeln:
  - Die Signatur der Methode zählt als Zeile
  - Schließende Klammern eines Anweisungsblocks zählen nicht als eigene Zeile, auch wenn sie in einer separaten Zeile stehen
  - Wenn eine Variable in getrennten Zeilen initialisiert und deklariert wird, so werden die Zeilen zu einer zusammengefasst.

  Nach diesen Regeln erhalten wir einen NCSS-Wert von 10. Dieser Zahlenwert missachtet mit den festgelegten Regeln allerdings die Komplexität des Problems, was ihn wenig aussagekräftig macht. Bei komplexen Problemen ist ein Wert von 10 vielleicht akzeptabel – eine Skala für NCSS geht nicht aus den Vorlesungsfolien hervor – das hier gelöste Problem (wir gehen davon aus, dass es sich bei der verwendeten Sprache um Java handelt), lässt sich allerdings auch mit einem NCSS-Wert von 2 lösen:

```java
int wandleInDezimalZahl(String binary) {
  return Integer.parseInt(binary, 2);
}
```

Es sei jedoch auch angemerkt, dass auch dieser Wert nicht wirklich aussagekräftig ist, da hier lediglich eine Methode aus dem Java-Standard aufgerufen wird, deren Komplexität bei einem NCSS-Wert von 2 ebenfalls nicht berücksichtigt wird.

- **NOC:** Diese Metrik ist hier nicht anwendbar, da das Codefragment lediglich eine Methode beinhaltet, von der keine Klasse abgeleitet werden kann. Außerdem handelt es sich dabei um eine objektorientierte Metrik, nicht um eine statische. Der Wert dieser Metrik gibt an, wie viele Unterklassen die betrachtete Klasse besitzt.
- **Länge l der Bezeichner:** Wir bestimmen zunächst die Längen aller Bezeichner und berechnen dann den arithmetischen Mittelwert:
  - `wandleDezimalzahl()`: 17
  - `Dezimalzahl`: 11
  - `Potenzwert`: 10
  - `leseNaechsteZeichen()`: 19
  - `INT_MAX`: 7
  - `Math`: 4
  - `pow(..)`: 3
  - `Zehn`: 4

$$l = \frac{17+11+10+19+7+4+3+4}{8} = \frac{75}{8} = 9,375$$

Nach Vorlesung gilt: Je höher der Wert ist, desto aussagekräftiger sind die Namen der Bezeichner. Lange Bezeichner haben keinen Einfluss auf die Funktionalität des Codes, allerdings machen sie ihn ungemein lesbarer, was wiederum dazu führt, dass knappe Kommentare genügen, um eine hohe Wartbarkeit des Codes zu garantieren.

Grundsätzlich ist es immer sinnvoll, die Bezeichner etwas länger zu wählen, wenn der Code dadurch selbsterklärender wird. In diesem Fall ist der Versuch jedoch etwas missglückt, denn

- der Bezeichner `wandleDezimalzahl()` sagt nicht unbedingt aus, dass ein Binär-String **in** eine Dezimalzahl umgerechnet wird (sinnvoller wäre hier vielleicht `wandleInDezimalzahl`).
- aus dem Bezeichner `leseNaechsteZeichen()` geht nicht hervor, dass der String (oder das **char**-Array, oder was auch immer als Eingabe verwendet wird) von rechts nach links durchlaufen wird, was in diesem Fall Grundvoraussetzung für die Korrektheit des Algorithmus ist, da die kanonische Darstellung einer Binärzahl vorsieht, dass die Exponenten von links nach rechts gelesen immer kleiner werden. Der Algorithmus hingegen erhöht den Exponenten in jeder Iteration um 1, was die Vermutung nahe legt, dass er von rechts nach links arbeitet.

Von diesen beiden Kritikpunkten abgesehen wird der Algorithmus durch die Wahl der Bezeichner verständlicher, da seine Funktion aus den restlichen Bezeichnern ohne weiteres hergeleitet werden kann.

Aufgabe 1: Grundlagen Äquivalenzklassen und Grenzwert

1. Abkürzungen in der Klausur kenntlich machen!: $KN \triangleq$ Kontonummer, Äquivalenzklassen sollten nahtlos sein.

   (a) `MIN_LONG` $< KN < 0$

   (b) $0 \leq KN < 1.000.000.000$, ggf. mit Nullen auffüllen

   (c) $1.000.000.000 \leq KN < 10.000.000.000$

   (d) $10.000.000.000 \leq KN <$ `MAX_LONG`

2. Sonder- und Randfälle ...

   (e) ... zu (a): $KN = 0$, $KN = -1$

   (f) ... zu (b) und (c): $KN = 999.999.999$, $KN = 1.000.000.000$

   (g) ... zu (c): $KN = 9.999.999.999$

   (h) ... zu (c) und (d): $KN = 10.000.000.000$

3. Mögliche Eingabewerte für Äquivalenzklassen:

   (a) Eingabewert sei −15, Soll-Ergebnis: „Error“

   (b) Eingabewert sei 15, Soll-Ergebnis: *DEpp bbbb bbbb* 0000 0000 15

   (c) Eingabewert sei 1.000.000.001, Soll-Ergebnis: *DEpp bbbb bbbb* 1000 0000 01

   (d) Eingabewert sei 10.000.000.000, Soll-Ergebnis: „Error“

   (e) Eingabewert sei 0, Soll-Ergebnis: *DEpp bbbb bbbb* 0000 0000 00
   Eingabewert sei −1, Soll-Ergebnis: „Error“

   (f) Eingabewert sei 999.999.999, Soll-Ergebnis: *DEpp bbbb bbbb* 0999 9999 99
   Eingabewert sei 1.000.000.000, Soll-Ergebnis: *DEpp bbbb bbbb* 1000 0000 00

   (g) Eingabewert sei 9.999.999.999, Soll-Ergebnis: *DEpp bbbb bbbb* 9999 9999 99

   (h) Eingabewert sei 10.000.000.000, Soll-Ergebnis: „Error“

Aufgaben 2 und 3: Kontrollflussbezogenes Testen

Testerfüllung

1. Flussdiagramm: Abbildung 5.1

2. Alle Entscheidungswege: $\{G, A, B\}$, $\{B, C, D\}$, $\{B, D\}$, $\{D, E, F\}$ und $\{D, F\}$

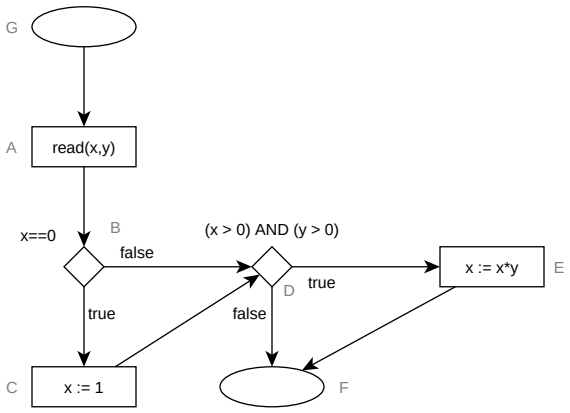


Abbildung 5.1: Flussdiagramm zu Aufgabe 2.1

3. Die Testmenge $\{(x = 0, y = 1), (x = -1, y = 5)\}$ deckt alle Pfade und Anweisungen ab. So werden vom ersten Tupel alle true-Zweige der if-Bedingungen ausgeführt und vom zweiten Tupel alle false-Zweige.

4. Die Testmenge erfüllt auch die Zweigüberdeckung – aus denselben Gründen, die bereits in Punkt 3 angeführt wurden.

5. Die Testmenge erfüllt die Entscheidungsüberdeckung, denn wenn alle Zweige überdeckt werden, werden auch alle Entscheidungen überdeckt.

6. Der Unterschied besteht in den Testwirksamkeitsmaßen:

$$C_{\text{Zweig}} := \frac{\#\text{besuchte Zweige}}{\#\text{Zweige}} \qquad TWM_1 := \frac{\#\text{überdeckte Entscheidungskanten}}{\#\text{Entscheidungskanten}}$$

7. Die einfache Bedingungsüberdeckung wird nicht erfüllt: Die atomaren Prädikate sind `x == 0`, `x > 0` und `y > 0`. Da `y > 0` niemals erfüllt ist, ist auch die einfache Bedingungsüberdeckung niemals erfüllt.

Testfälle

1. Kontrollflussgraph: Abbildung 5.2

2. Für Testfälle ist lediglich der Parameter `zeichen` relevant. Für eine vollständige Anweisungsüberdeckung genügen zwei Eingaben: `zeichen = { "1", "2"}`

3. Für eine minimale vollständige Zweigüberdeckung genügen die Testfälle `zeichen = { "1", "0", "2"}`

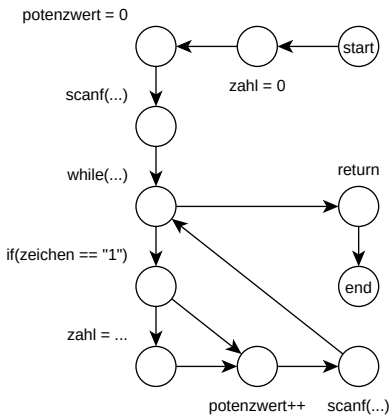


Abbildung 5.2: Kontrollflussgraph zu Aufgabe 3.1

Aufgabe 1: Grenze-Inneres-Überdeckung

1. Gesucht ist eine minimale Menge M von Tupeln der Form (a, b, c, d). M soll die minimal bestimmende Mehrfachbedingungsüberdeckung erfüllen, d.h. die Menge muss jede Kombination von Wahrheitswerten enthalten, die den **gesamten** Wahrheitswert des Terms ändert:

$$\texttt{while ((a<b) OR (c<d))} = \begin{cases} \textbf{true} & \text{, für Belegung } (0,1,0,1) \\ \textbf{false} & \text{, für Belegung } (0,0,0,0) \end{cases}$$

   Also erhalten wir eine Menge M mit $M = \{(0,0,0,0),(0,1,0,1)\}$

2. **Grenze-Inneres-Überdeckung:** Dynamisches, kontrollflussbasiertes Testverfahren, muss mindestens einen Testfall pro Schleife enthalten, bei dem diese
   - gar nicht, d.h. bei while- und for-Schleifen ist die Abbruchbedingung bei der ersten Auswertung wahr
   - genau einmal
   - mehr als einmal

   ausgeführt wird.

   Wir benötigen an dieser Stelle also drei Testfälle.

   Keine Ausführung: Die Abbruchbedingung wird beim ersten Aufruf als wahr ausgewertet, wenn wir den Testfall $(0,0,0,0)$ verwenden.

   Einmalige Ausführung: Eine einmalige Ausführung der Schleife wird durch den Testfall $(0,1,0,1)$ erreicht.

   Mehrmalige Ausführung: Für eine mehrmalige Ausführung müssen wir die Grenzen b und d erhöhen, also nutzen wir den Testfall $(0,5,0,5)$.

   Als minimale Testmenge erhalten wir dann $M_{\text{min}} = \{(0,0,0,0),(0,1,0,1),(0,5,0,5)\}$

3. Für das linke Programm werden drei Testfälle benötigt, da hier eine abweisende Schleife verwendet wird. Das rechte Programm hingegen benötigt nur zwei Testfälle, da die Schleife immer mindestens einmal ausgeführt wird.

Vor dem Refactoring	Nach dem Refactoring
keine Ausführung: `counter = 0` genau eine Ausführung: `counter = 1` mehr als eine Ausführung: `counter = 2`	genau eine Ausführung: `counter = 1` mehr als eine Ausführung: `counter = 2`

4. In beiden Fällen handelt es sich um abweisende Schleifen, sodass für beide jeweils drei Testfälle existieren müssen. Dieses Kriterium ist für die gegebenen Testdaten nur in der ersten Schleife erfüllt, der Überdeckungsgrad liegt also bei 1/2.

   Erste while-Schleife:

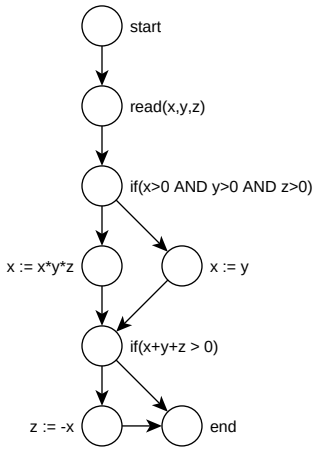
   - kein Durchlauf für $(5,5)$
   - ein Durchlauf für $(6,3)$
   - mehrere Durchläufe für $(8,2)$

5. Es wird ein Testfall benötigt, der in beiden Schleifen dafür sorgt, dass diese nicht ausgeführt werden. Dies gewährleistet der Testfall $(0,0)$. Darüber hinaus werden beide Schleifen mit $(6,3)$ genau einmal und mit $(3,9)$ mehrfach ausgeführt.

   Wir erhalten also als minimale Menge $M_{\min} = \{(0,0),(6,3),(3,9)\}$

Aufgabe 2: Kontrollflussbezogenes Tests – Testfälle II

Wir gehen davon aus, dass in den folgenden Mengen alle Tupel die Form (x,y,z) haben:



1. Anweisungsüberdeckung: $M_{\min} = \{(1,1,1),(-1,-1,-1)\}$
2. Einfache Bedingungsabdeckung: $M_{\min} = \{(1,1,0),(0,0,1)\}$
3. Mehrfachbedingungsabdeckung:
   $M_{\min} = \{(1,1,1),(1,1,-1),(1,-1,1),(1,-1,-1),(-1,1,1),(-1,1,-1),(-1,-1,1),(-1,-1,-1)\}$
4. Minimal bestimmende Mehrfachbedingungsabdeckung: $\{(1,1,1),(1,1-1),(1,-3,1),(-1,1,1)\}$
5. Es gibt keine Pfadabdeckung, da beide if-Bedingungen voneinander abhängig sind. Ist die erste erfüllt, so ist auch immer die zweite erfüllt

Aufgabe 3: Refactoring

Wir gehen wieder davon aus, dass in den folgenden Mengen alle Tupel die Form (x, y, z) haben:

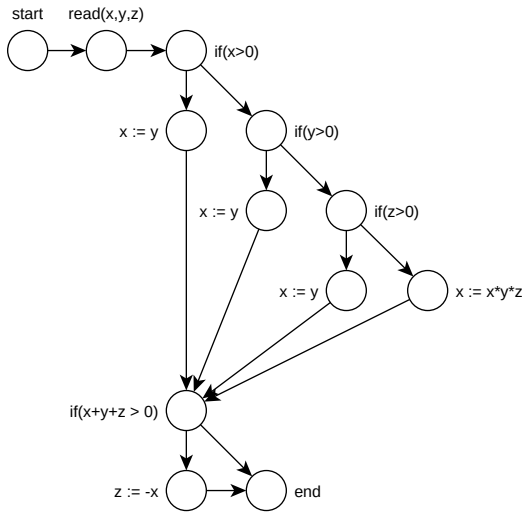


Abbildung 6.1: Zur Veranschaulichung betrachten wir den Kontrollflussgraphen des gegebenen Programms.

1. Anweisungsüberdeckung: $M_{\min} = \{(1,1,1),(1,1,-1),(1,-1,-1),(-1,-1,-1)\}$

   Es werden zwei weitere Testfälle benötigt, um alle Anweisungen zu überdecken, da die tiefere Verschachtelung der ersten if-Bedingung else-Zweige erzeugt hat, die nur mit weiteren Testfällen erreicht werden können.

2. Einfache Bedingungsabdeckung: $M_{\min} = \{(1,1,1),(1,1,-1),(1,-1,-1),(-1,-1,-1)\}$

   Die Testfälle müssen sicherstellen, dass alle Bedingungen erreicht werden.

3. Mehrfachbedingungsabdeckung: $M_{\min} = \{(1,1,1),(1,1,-1),(1,-1,-1),(-1,-1,-1)\}$

   Durch die Aufteilung müssen nun keine Kombinationen atomarer Prädikate betrachtet werden. Es ist jedoch dafür zu sorgen, dass jede Belegung mindestens einmal ausgewertet wird. Die Überdeckung fällt also mit C_2 zusammen.

4. Minimal bestimmende Mehrfachbedingungsabdeckung:
   $M_{\min} = \{(1,1,1),(1,1,-1),(1,-1,-1),(-1,-1,-1)\}$

   Die Überdeckung fällt ebenfalls mit C_2 zusammen.