~Mit Dank an Julian Rick für die Bereitstellung des hier verwendeten Materials.~

SOFTWAREKONSTRUKTION

~WS 2014/2015~

KAPITEL 1.1: MODELLBASIERTE SOFTWAREENTWICKLUNG

Im letzten Kapitel: Einführung zum Thema „Modellbasierte Software-Entwicklung“

- Geschichte der Software-Entwicklung (Entwicklung der Programmiersprachen)
- Herausforderungen (Steigende Komplexität, wie zu beherrschen? Wunsch nach mehr Abstraktion)
- Erläuterung des Modellbegriffs und Vorstellung von SW-Modellen (UML)
- Definition der Semantik von Modellen (welche Bedeutung soll durch das Modell ausgedrückt werden?)

Modellbasierte Softwareentwicklung

Was ist das? Modellbasierte Softwareentwicklung bezeichnet das Entwickeln von Software unter Zuhilfenahme von (UML-)Modellen. Diese Modelle werden dabei laufend aktualisiert und dienen als Bezugspunkte für alle Mitarbeiter eines Teams, egal ob Entwickler, QS-Mitarbeiter oder Projektleitung.

Das gibt es doch schon längst. Leider nur in der Theorie. Bei vielen Projekten finden zwar ebenfalls Modelle Verwendung, diese werden aber nur zu Beginn einmal erstellt und werden sehr schnell veraltet. Nachdem sie erstellt wurden kümmert sich niemand mehr darum um neue Designentscheidungen einzupflegen oder gefundene Fehler zu beheben.

Warum ist das so? Die Verwendung von Modellen wird oftmals mehr als nervige Pflicht gesehen, denn als etwas, was wirklich Nutzen bringt. Das liegt zum einen daran, dass es ein hoher Aufwand ist alle Modelle immer up-to-date zu behalten, zum anderen bieten Modelle meistens nur einen „optischen“ Nutzen, d. h. sie stellen Hierarchien oder Abläufe übersichtlicher als im Code dar.

Wo liegen die Probleme? Die Verwendung von Modellen muss eine zentrale Stelle in der Softwareentwicklung einnehmen. Es darf kein „nice-to-have“ sein, sondern die Modelle sollten der Dreh- und Angelpunkt eines Projektes werden. Dazu muss der Nutzen von Modellen hervorgehoben werden, beispielsweise eine automatische Codegenerierung aus den verwendeten Modellen heraus. Wir werden uns dazu in diesem Kapitel zuerst mit Anforderungen an die modellbasierte Softwareentwicklung sowie derer Grundlagen auseinandersetzen. In Kapitel 1.5 werden wir dann anhand des Eclipse Modelling Frameworks konkrete Beispiele betrachten.

Modelle dienen dazu, von technischen Feinheiten und Details zu abstrahieren und sich einen größeren Überblick über ein System oder Teilen davon zu verschaffen. Beispielsweise gibt es in der UML Anwendungsfalldiagramme zur groben Einteilung einer Anwendung in verschiedene Benutzungsfälle, Aktivitätsdiagramme zur Darstellung von Abläufen, Sequenzdiagramme, die immer noch ein gewisses Abstraktionsniveau bieten aber schon ziemlich nah an der Codeebene sind, Zustandsdiagramme zur Beschreibung von Zuständen und Übergängen (nützlich beispielsweise beim Entwurf von Automaten), u. a.

Modelle blenden für eine spezifische Sicht auf Dinge unnötige Details aus, sodass im Idealfall alle Mitglieder eines Teams verstehen sollten, welche Absicht hinter dem im Modell Gezeigten steckt. Betrachten wir folgenden Pseudo-C++-Code eines Clientsystems, welches sich per RPC (Remote Procedure Call) die Uhrzeit von einem Server holt. Das Beispiel ist konstruiert, da der Einsatz von RPC aber oftmals sehr komplex ist es praktisch, um die Abstraktion in einem Modell zu verdeutlichen:

```
// Instantiate RPC server object
RPCServer _rpc;
if(connectToServer(local_ip, &_rpc, size(buffer), AUTH_SECURE_LEVEL_3)
    !=
    ERROR_INVALID_CONNECTION)
{
        t_time timestamp = _rpc.fetchTime(LOCAL_TIME, 0, 0, false);

        setTime(convert(t_time, *timeType, CLIENT_ADDR);
}
```

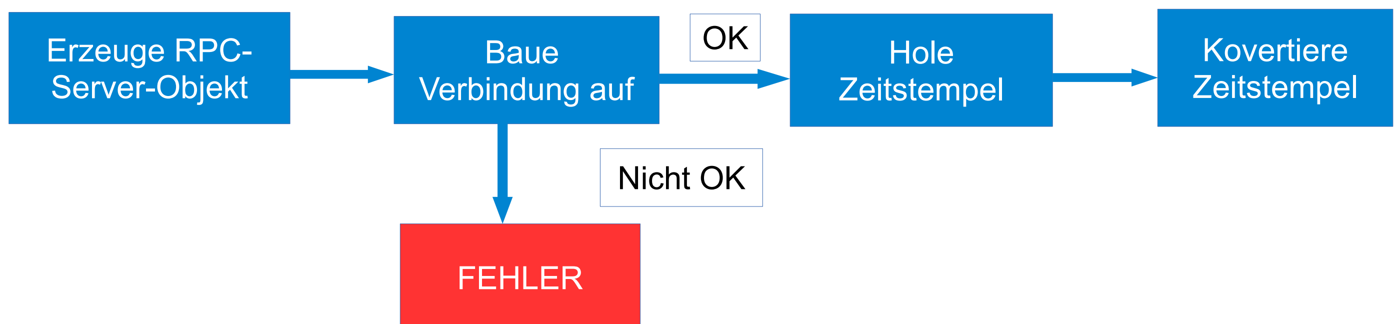
Versuchen wir nun dem Code etwas Sinn zu geben. Es wird ein RPC-Server Objekt erzeugt und versucht, eine Verbindung zu einem Server mit der IP *local_ip* aufzubauen. Weitere Argumente des Funktionsaufrufs sind eine Referenz auf das Serverobjekt, die Größe eines Buffers, damit der Server weiß wie lang seine Antworten sein dürfen (in Bytes) und eine Konstante, die etwas über die Sicherheitsstufe aussagt, mit der diese Verbindung betrieben wird.

Falls die Verbindung erfolgreich aufgebaut werden konnte, d. h. der Rückgabewert der *connectToServer*-Funktion ungleich der Konstanten *ERROR_INVALID_CONNECTION* ist, wird sich der aktuelle Zeitstempel per RPC vom Server geholt. Allerdings ist der Rückgabewert vom falschen Typ und muss deswegen in der *setTime*-Methode zuvor noch in einen anderen Typen konvertiert werden. Dieser andere Typ liegt irgendwo im Speicher und wird mittels eines Pointers angegeben.

Welche Werte genau die Pointer, Referenzen und Konstanten haben und ob obiges Beispiel sich überhaupt kompilieren lässt ist uninteressant, wichtig ist zu sehen, dass selbst dieses kurze Beispiel einiges an Komplexität mit sich bringt. Nun bestehen aktuelle Softwareprojekte aus vielen Millionen Codezeilen und die Komplexität wird dadurch nicht weniger. Schauen wir uns unser Beispiel noch mal an, was davon ist wirklich relevant, wenn wir die Funktionsweise einem Kollegen aus der QS-Abteilung erklären wollten?

- Wir legen ein RPC-Server Objekt an
- Wir versuchen eine Verbindung zum Server aufzubauen
- Wenn das geglückt ist holen wir uns den aktuellen Zeitstempel
- Danach müssen wir den Zeitstempel noch in ein passendes Format konvertieren

Unser Kollege aus der QS versteht weder etwas von der tiefergehenden Programmierung mit Pointer und Referenzen aus unserem Beispiel, noch ist es für seine Zwecke interessant. Er möchte einfach den Ablauf unseres Beispiels skizzieren. Und das könnte so aussehen:



Das ist eine abstrakte Zusammenfassung unseres Codebeispiels. Wir finden hier keinerlei Eigenheiten von C++, keine Informationen über Typen von Variablen oder irgendwelche Rückgabewerte. Unser „Modell“ ist auf jede beliebige Programmiersprache anwendbar. Damit sind wir bei einem wichtigen Punkt der modellbasierten Softwareentwicklung angekommen.

Modell-Driven-Architecture: Idee

Die Grundidee der MDA ist die Spezifikation von Softwarekomponenten unabhängig von technischen Umsetzungen. Dazu gibt es zwei Abstraktionslevel:

- Plattform-unabhängig: die Spezifikation enthält keinerlei Informationen über eine Zielplattform
- Plattform-spezifisch: die Spezifikation enthält Informationen über eine Zielplattform

Der Begriff Plattform ist dabei nicht zwingend im Sinne von Betriebssystem oder Prozessorarchitektur gemeint, sondern bezieht sich auf spezifische(re) Modelle einer Ebene, d. h. Modelle mit mehr Zusatzinformationen die so nicht mehr für alle Modelle gelten würden. Beispielsweise könnte man ein Modell zur Festplattenpartitionierung haben. In einem plattform-unspezifischen Modell hätte man hier nur eine Auflistung von Funktionen, ähnlich einem Interface. Im plattform-spezifischen Modell steht hier möglicherweise etwas wie SECTOR_ALIGNMENT=4k, ein Attribut welches eine Aussage darüber trifft, wie die Sektoren der Festplatte angeordnet werden sollen. Das gilt sicherlich nicht für alle Festplatten, also ist es plattform-spezifisch.

Das von unserem QS-Mitarbeiter angefertigte Modell hingegen ist plattform-unabhängig. Es enthält nur abstrakte Beschreibungen des Vorgangs. Falls es nun gewünscht wäre, das Modell plattform-spezifischer zu machen, wie könnte man dazu vorgehen?

Die MDA arbeitet mit der Idee, dass Übergänge von plattform-unabhängigen zu plattform-spezifischen Modellen vollautomatisiert geschehen sollten. Dazu ist eine Beschreibung der Transformationen notwendig, d. h. eine Art von Regeln, wie Modelle erweitert werden sollen um den gewünschte Grad an Spezifikation zu erhalten. Diese Transformationsbeschreibungen finden sich nicht im Modell selbst, denn da würden sie ja auch irritieren, sondern werden separat vorgehalten.

Welche Vorteile haben wir von unserem Modell?

Stellen wir uns vor, wir hätten das Modell nicht erstellt. Eines Tages wäre der Entwickler, der unseren RPC-Code geschrieben hat in den Ruhestand gegangen. Zwei Monate später macht die Komponente fürchterliche Probleme, und keiner weiß wieso.

Durch den Einsatz eines Modells sind wir mitnichten in der Lage Fehler sofort und effizient zu finden und zu beheben, es erhöht unsere Chancen aber deutlich. Wenn wir im Fehlerfall einen Blick in unser Modell werfen sehen wir zwei mögliche Ursachen:

- Die Verbindung zum Server konnte nicht aufgebaut werden
- Der Zeitstempel ist im falschen Format

Wir tappen also nicht komplett im Dunkeln. In der Realität ist es jedoch oft so, dass wertvolles Wissen über Abläufe, Fehlerbehandlungen, Abhängigkeiten, … lediglich in den Köpfen der Entwickler vorhanden sind oder in Altsystemen in Form von alter Software. Es besteht ein akuter Mangel an Dokumentation und Wissensspeicherung. Die plattform-unabhängige Modellierung der MDA umfasst folgende Punkte:

- **Pflege und Weiterentwicklung der Anwendungslogik**
  Im Modell: beispielsweise könnte es notwendig sein, die Fehlerbehandlung zu ändern; dann sollte der neue Stand auch im Modell sichtbar sein

- **Dokumentation und Evolution der Prozesse**
  Wichtig für Nachvollziehbarkeit und Fehlersuche. Beispielsweise hat der Code in der letzten Version noch funktioniert, mit den letzten Änderungen ist er kaputt gegangen. Ohne eine vollständige Dokumentation tappt man hier im Dunkeln

- **Integration dokumentierender Modelle in Transformationskette hin zur Anwendung**
  Die Modelle sollten in die Transformationskette eingebunden werden, d. h. sie sollen nicht einfach irgendwo herumliegen sondern aktiv zum Endergebnis (fertiges Produkt, ausführbarer Code) beitragen, beispielsweise durch automatische Codegenerierung

Es wird klar, dass durch die ständige Nutzung und Integration der Modelle diese ebenfalls auf einem aktuellen Stand gehalten werden. MDA beugt also der Degeneration von Dokumenten (und Modellen) vor und damit genau dem, was in den meisten Softwareprojekten eher früher als später passiert.

Welche weiteren Ziele hat die MDA?
Neben der Abstraktion von technischen Details ist es ebenfalls das Ziel der MDA Dinge wie *Portierbarkeit, Systemintegration und Interoperabilität* zu ermöglichen. Was steckt dahinter?

Portierbarkeit bezeichnet die Eigenschaft eines Systems, wie gut es sich von Zielumgebung A nach Zielumgebung B portieren lässt, d. h. wie viele Anpassungen notwendig sind, damit die Software ebenfalls auf Plattform B (fehlerfrei) läuft. Die Zielplattform kann dabei ein anderes Betriebssystem oder eine neue Major-Version einer benutzen Technologie sein (Beispiel: IPv4 und IPv6). Eine gute Portierbarkeit vereinfacht die parallele Entwicklung für mehrere Zielplattformen.

Systemintegration und Interoperabilität bezeichnet die Trennung von konkreter, technischer Repräsentation in Technologie von fachlichen Konzepten. Dadurch wird die Bildung von Schnittstellen erleichtert. In welchen technischen Details die Namensauflösung von IPv6-Adressen umgesetzt wurde ist also uninteressant, interessant ist lediglich die Vorgehensweise, d. h. welche Schritte (grob gefasst) werden unternommen, um eine Adresse aufzulösen. Ein weiteres Beispiel für Systemintegration und Interoperabilität ist unser Codebeispiel und das zugehörige Modell weiter oben.

Ein anderer Aspekt ist die *Verwendung offener Standards*. Das verringert zum einen das Risiko eines Vendor Lock-Ins und hilft zum anderen bestehende Anwendungen über einen Technologiezyklus zu retten. Programmiert man beispielsweise einen Webbrowser und nutzt dafür das HTTP-Protokoll ist das Risiko eines Vendor Lock-Ins sehr gering, da es sich um ein offen einsehbares Protokoll handelt. Ebenfalls ist die Anzahl an neuen Version von HTTP sehr überschaubar. Möchte man hingegen einen Videostreamingdienst anbieten und nutzt dafür proprietäre Protokolle läuft man Gefahr, dass man Teile seiner Software mit jedem Update anpassen muss. Im Laufe der Zeit hat man sein Produkt dann so sehr auf das verwendete Protokoll angepasst, dass es wirtschaftlich gesehen zu spät ist auf eine offene Lösung zu setzen.

Weitere Ziele der MDA

- Ein weiteres Ziel der MDA ist die *Automatisierung der Anwendungserstellung*, d. h. automatisches Prüfen spezifizierter Einschränkungen (wir erinnern uns: OCL) oder Codegenerierung. Das beschleunigt den gesamten Software-Prozess, da viel stupide Programmierarbeit automatisch umgesetzt wird und Entwickler nur noch an die interessanten Stellen müssen. Außerdem wird durch automatisches Testen die Software-Qualität stark gesteigert. Fehler fallen zudem früher auf und können kostengünstiger behoben werden.

- Die MDA hilft des Weiteren dabei, wichtiges *Wissen zu kapseln*, d. h. in Form von Transformationsvorschriften zu speichern und jederzeit einsetzen zu können. Das ermöglicht eine optimale Nutzung von Ressourcen, denn es muss nicht für jedes Projekt alles Wissen von null neu aufgebaut werden, und ebenfalls eine gleichzeitige Nutzung von

Ressourcen, in diesem Fall des Wissens.

- *Nutzung domänenspezifischer Modelle und Wiederverwendung von Architekturmetamodellen* zur Reduzierung von Time-to-Markets (Zeit bis zum Release eines Produkts) ohne die Notwendigkeit, dafür höhere Budgets zur Verfügung zu stellen (bspw. für mehr Entwickler, mehr Tester, …)
- *Fachlichkeit innerhalb des Software-Entwicklungszyklus:* es soll schnell auf Änderungen unterliegender Geschäftsprozesse und notwendige Änderungen aufgrund wechselnder technologischer Vorgaben reagiert werden können; das System soll nicht **starr** und **fest** sein sondern **flexibel** und **agil**

Nachdem wir uns mit den Zielen der MDA vertraut gemacht haben wollen wir uns nun mit der Meta-Modellierung in MDA auseinandersetzen. Zuvor führen wir noch folgende Begriffe ein:

Wir haben bereits plattform-spezifische und plattform-unabhängige Modelle kennengelernt. Diese werden in der MDA als *PIM* (Platform Independent Model) und *PSM* (Platform Specific Model) bezeichnet. Außerdem gibt es noch das *CIM*, das Computation Independent Model. Dieses spezifiziert Anforderungen an System und Umwelt. Was bedeutet das? Das CIM enthält Informationen darüber, wie das System genutzt werden kann (also: was ist möglich mit diesem System, was kann ich damit alles tun – eine Art Auflistung des Funktionsumfangs) und was dazu notwendig ist (also: welche Bedingungen müssen erfüllt sein, damit ich das System so nutzen kann wie beschrieben? Ist die notwendige Infrastruktur da? Beispiel: Für die Installation eines Produktes werden ein DNS-Server und ein Active Directory benötigt).

Modell und Metamodell der UML

Was ist ein *Modell* überhaupt? Ein Modell ist eine Abbildung eines real existierenden Systems auf verschiedene UML-Diagrammtypen: Klassendiagramm, Aktivitätsdiagramm, …
Und was ist ein *Metamodell*? Ein Metamodell ist ein Modell, welches ein Modell beschreibt. Beispielsweise ist die Spezifikation der UML-Diagrammtypen ein Metamodell. Ein einzelner Typ, beispielsweise ein Klassendiagramm ist dann ein Modell. Eine konkrete Repräsentation dieses Modells, beispielsweise eine Klasse „Mensa“ heißt **Instanz** eines Modells.

Wozu Metamodelle?
Metamodelle definieren die Elemente, die in einem Modell vorkommen dürfen. Beispielsweise Klassen, Pfeile, Stereotypen, …
Zum einen garantieren Metamodelle, dass sich nicht jeder etwas Eigenes ausdenkt, wie er beispielsweise UML-Klassendiagramme gestalten möchte (statt Pfeilen zwischen Klassen bunte Dreiecke). Zum anderen erlauben Metamodelle eine automatische Weiterverarbeitung von Modellen. Das kann sein:

- Validierung von Modellen
- Speicherung von Modellen
- Datenaustausch/Interoperabilität
- Definition von Transformationen

Besonders interessant sind dabei die beiden letzten Punkte. Wir haben unser RPC-Modell bewusst generisch gehalten (PIM) damit wir es bei Bedarf für anderen Zielplattformen benutzen können. Zur Anpassung unseres Modells für eine andere Zielplattform benötigen wir zusätzliche Informationen. Dann können wir unser Modell in ein PSM überführen. Die Transformationsregeln, welche die Transformation anhand des Modells und der Zusatzinformationen durchführen, werden

ebenfalls im Metamodell beschrieben. Transformationen können beliebig hintereinander geschaltet werden, sodass man Modelle beliebig ineinander transformieren kann, vorausgesetzt es gibt dazu Transformationsregeln (auch Mapping Rules) genannt. Diese Möglichkeit der Transformation ist ein Kernstück des MDA-Ansatzes. Erst durch Transformation entfaltet sich das wahre Potenzial von Modellen. Sie können nun an allen Stellen eingesetzt werden, wo sie benötigt werden und liegen nicht mehr einfach nur in einem Dokumentationsverzeichnis rum. Modelltransformation ist sozusagen eine Art „Programmierung mit Modellen". Natürlich können sie nicht den kompletten Programmieraufwand ersetzen, aber Modelle verhindern viel stupide Programmierarbeit (bspw. automatische Generierung von Code aus UML-Klassendiagramm, keine Notwendigkeit getter- und setter-Methoden von Hand zu implementieren) und bleiben durch Transformation **aktuell**. Ein weiterer positiver Punkt der Modelltransformation ist, dass viel Know-How in den Transformationsregeln steckt und nicht irgendwo im Code oder in Designdokumenten. Die Wiederverwendbarkeit von Modellen und Wissens wird so erhöht.

Welche Transformationsarten gibt es?
Es gibt *horizontale und vertikale* Transformation. Horizontale Transformation bezeichnet dabei die **inhaltliche Weiterentwicklung** eines Models. Das Modell bleibt auf demselben Abstraktionsniveau, erfährt aber Änderung seines Inhaltes.

Die vertikale Transformation bezeichnet die Transformation eines Modells auf **technologiespezifischer Ebene**. D. h. bei der vertikalen Transformation verliert ein Modell an Abstraktion, dafür wird es plattform-spezifischer und technisch detaillierter. Vertikale Transformation geht in der Metaebenen-Darstellung (siehe Foliensatz 1.2, Folie 26) von oben nach unten. Damit ein Modell vertikal transformiert werden kann, sind sein PIM und „weitere Informationen" notwendig. Die weiteren Informationen sind technischer Natur. In unserem Modell für das RPC-Beispiel könnten diese Informationen zum Beispiel Informationen über den genutzten Compiler oder die RPC-Version sein.

Modelltransformation kann für Modell-Modell-Transformation sowie für Modell-Code-Transformation genutzt werden. Letztere kann dabei u. U. auch als Modell-Modell-Transformation aufgefasst werden, da Code je nach Abstraktionsgrad (bspw. Hochsprache) auch als eine Art Modell aufgefasst werden kann (was es im Vergleich zu Assemblersprachen zweifelsohne ist). Modelltransformationen zwischen Modellen arbeiten dabei auf den zu den Modellen zugehörigen Metamodellen: hier stehen die Transformationsregeln.

Zur Beschreibungen von Abbildungen von Instanzen eines Metamodells in ein anderes Metamodell, d. h. F: $x \in MMA \rightarrow x \in MMB$ gibt es so genannte Modelltransformations-sprachen. Diese unterscheidet man zwischen deklarativ und imperativ. Deklarative Sprachen erlauben dabei Spezifikation von Regeln mit Vor- und Nachbedingungen (vgl. OCL), imperative Sprachen beschreiben Transformationen durch Sequenzen von Aktionen. Sie arbeiten also wie imperative Programmiersprachen wie bspw. C.

Welche Standards gibt es im Bereich von MDA?
Im Bereich von MDA kommen verschiedene Technologien und Standards zum Einsatz. Offensichtlich gehört die UML dazu, denn damit wird der größte Teil der Arbeit erledigt: Erstellen von Klassen-, Aktivitäts-, Sequenz-, Use-Case-, Zustands- und anderen Diagrammen. Es gibt allerdings noch weitere Werkzeuge die zu MDA dazugehören und durch die Kernaspekte wie Modelltransformation erst möglich werden.

Meta Object Facility
Die Meta Object Facility ist eine modellbasierte Sprache der Object Management Group (OMG) und dient zur Definition von Metamodellen. Von Metamodellen? Ja, bei der MOF handelt es sich

also um ein Meta-Metamodell: ein Modell, welches Metamodelle definiert. In der MOF sind unter anderem alle Standards des UML-2.0 Stacks definiert. Wie erinnern uns: das sind die verschiedenen Diagrammtypen von UML. Diese werden als Metamodelle bezeichnet. Eine Repräsentation eines dieser Metamodelle ist dann ein Modell. Die Kette sieht also folgendermaßen aus:

XML Metadata Interchange

Die XMI definiert eine Abbildung von MOF auf XML, d. h. mit Hilfe der XMI kann aus Meta-Modellen ein XML-Dokument erzeugt werden. Dieses kann dann an anderer Stelle wieder eingelesen werden, sodass alle Stellen über die gleiche Information verfügen. Der Austausch kann beispielsweise zwischen Transformatoren und Codegeneratoren stattfinden. Der Transformator transformiert zuerst das Modell vertikal bis zur für den Codegenerator notwendigen Abstraktion und packt es dann mittels XMI in ein XML-Dokument. Es wird schnell klar, dass XMI ein essentieller Bestandteil der MDA ist: XMI ermöglicht aktuelle Modelle und Austausch zwischen Komponenten. Ohne XMI wäre MDA kein so mächtiger Ansatz.

Kann man sich ein eigenes UML-Metamodell erstellen?

Prinzipiell ist das möglich, jedoch mit einem hohen Aufwand verbunden. Eine alternative Möglichkeit wäre nicht ein komplettes Metamodell zu erstellen sondern das bestehende zu erweitern. Das kann entweder über den direkten Zugriff auf das Metamodell geschehen. Dann wird aber uneingeschränkter Zugriff auf selbiges benötigt und man würde die existierende Semantik, d. h. die Bedeutung der einzelnen Komponenten, verletzen. Eine gute Alternative ist die Verwendung von UML Profiles. UML Profiles sind eine Erweiterung von UML und setzen auf UML auf. Es wird dabei nichts verändert sondern nur erweitert. UML Profiles sind vergleichbar mit einer abgeleiteten Java-Klasse, die nur weitere Funktionen implementiert, aber keine Funktionen der Vaterklasse überschreibt. Beispiele für UML Profiles gibt es einige: Profile für Echtzeitmodellierung, Profile zur Testmodellierung, Profile für Modellierung sicherheitskritischer Anwendungen (UMLsec), …

Welche Gründe kann es für den Wunsch nach UML-Erweiterung geben?

Anhand der genannten UML Profile kann man bereits erkennen, dass es Dinge gibt, die mit dem Standard UML-Stack nicht modelliert werden können. Wenn diese Dinge aber nicht modelliert werden können fehlt in Modellen benötigtes Fachwissen. Fehlendes Fachwissen wiederum erhöht Entwicklungskosten und –dauer.

Da UML von jeder fachlichen Domäne abstrahiert können solche Domänen also nicht beschrieben werden. Man möchte das oftmals aber dennoch tun. Wie kann man das anstellen? Alle Modelle mit riesigen Kommentaren, die Erklärungen beinhalten, zu versehen ist sicherlich eine schlechte Idee. Denn die Texte müssen immer wieder händisch angepasst werden. Besser wäre es, eine Technologie zu verwenden die maschinenlesbar und automatisierbar ist wie von MDA gefordert. UML Profiles eignen sich dazu gut.

Was ist ein UML Profile?

Ein UML-Profile ist eine Erweiterung von UML. Dabei werden Standard UML-Elemente zu konkreten Metatypen spezialisiert. Was bedeutet das? Den UML-Elementen wird keine neue (denn das wäre Verletzung der Semantik), aber eine speziellere Bedeutung gegeben. Die Bedeutung selbst hängt dabei vom Profile ab (Echtzeit-UML, Security-UML, Medical-UML, Logistical-UML,

was man eben haben möchte). Ein Profil kann zu einem Modell hinzugefügt werden und ist dann im gesamten Modell verfügbar, d. h. in allen Modellelementen.

Inhaltlich ist ein Profile ein Paket von Stereotypen und Tagged Values. Stereotypen sind dabei Bezeichner in spitzen Klammern, <<stereotype>>, die bestimmte Metatypen bezeichnen. Im Standard-UML gibt es in Klassendiagrammen beispielsweise den Stereotype <<interface>>. Eine Klasse, die mit <<interface>> ausgezeichnet ist dann keine Klasse, sondern ein Interface, also eine **spezialisierte** Klasse. An der eigentlichen Semantik geht dabei nichts kaputt. Tagged Values sind einfach Name-Wert-Paare, vergleichbar mit einer map<String, String>. Außerdem enthält ein Profile noch ein Klassendiagramm welches die Beziehung zwischen dem Stereotype und dem zu beschreibenden Element enthält (bspw. Stereotype und „Pfeil"). Die Semantik in UML-Profiles wird durch Text oder OCL constraints „definiert". Es handelt sich dabei um keine echte, maschinenlesbare Definition sondern dient als Verständnishilfe für einen menschlichen Betrachter.

Zum Abschluss wollen wir uns anschauen, wie automatische Codegenerierung aus Modellen in MDA umgesetzt werden kann. Dazu wird ein Generator benötigt. Der Generator arbeitet dabei ähnlich wie der Transformator, welcher die Transformationsregeln abarbeitet. Der Generator erhält jedoch mehr Eingaben. Dazu gehören:

- Das Modell, aus dem Code generiert werden soll
- Die Transformationslogik
- Parameter, die das Endergebnis beeinflussen; ohne diese Parameter wäre das Ergebnis immer das gleiche

Wozu braucht man die Parameter? Beispielsweise möchte man ausloten, ob eine Softwarekomponente mit 32- oder mit 64-bit Integern effizienter arbeitet. Dann passt man die Parameter entsprechend an und erhält durch den Codegenerator so sehr schnell unterschiedliche Programmversionen, die miteinander verglichen werden können. Ohne Codegenerator müsste man alle Stellen im Code heraus suchen und von Hand ändern.

Der Ablauf der Codegenerierung ist folgender: zuerst wird die Eingabespezifikation eingelesen. Das kann ein UML-Modell sein, ein Text (in maschinenlesbarer Form), ein von XMI erzeugtes .xml-Dokument, …
Danach werden die Parameter eingelesen und im Anschluss die Transformationsregeln angewandt. Ein Codegenerator nimmt viel Arbeit ab und ermöglicht effizientes Arbeiten, allerdings muss er zuerst programmiert werden. Da Codegeneratoren zu spezifisch sind (niedriges Abstraktionsniveau, vertikale Transformation!) gibt es keine generischen Generatoren die für alle Zwecke eingesetzt werden könnten.

KAPITEL 1.2: OBJECT CONSTRAINT LANGUAGE

Im letzten Kapitel: Modellbasierte Softwareentwicklung

- Elemente der modellbasierten Software-Entwicklung: MDA und Metamodellierung
- Welche Ideen liegen MDA zugrunde? - Modelle als Mittelpunkt des Entwicklungsprozesses
- Welche Vorteile bietet MDA? - Früheres Aufdecken von Fehlern, Wiederverwendung von Wissen, automatische Codegenerierung …
- Was macht MDA so besonders? - Modelltransformation mittles XMI und Transformationssprachen, Meta-Modelle, ...

Object Constraint Language

Was ist OCL? OCL steht für Object Constraint Language und ist eine Sprache mit der man Einschränkungen (Constraints) auf (UML-)Modellen spezifizieren kann. Dabei ähnelt OCL in gewisser Weise einer Programmiersprache, stellt jedoch einen Großteil der aus Programmiersprachen bekannten Elemente zur Kontrollflusssteuerung nicht bereit. Beispielsweise bietet OCL zwar den Konditionalausdruck „if-then-else-endif“ zur einfachen Fallunterscheidung, es besteht aber keine Möglichkeit Schleifen (for, while, …) zu simulieren.

Mit Hilfe von OCL können beispielsweise Vor- und Nachbedingungen, aber auch Invarianten auf Modellen spezifiziert werden. Die Java-Entsprechung von OCL sind Assertions: Assertions brechen die Ausführung eines Programms ab, wenn die von ihnen geforderte Bedingung nicht erfüllt ist. Im Gegensatz zu Assertions arbeitet OCL jedoch nicht mit konkreten Daten die zur Laufzeit in einem Programm vorhanden sind bzw. eingegeben werden, sondern setzt auf UML-Modellen auf.

Beispiel:
Gegeben seien eine Klasse „Mensa“, die folgende Methoden und Attribute enthält, die Enumerationsklasse „MensaStatus“ und die Klasse „Person“. Zwischen der Klasse Mensa und Person besteht eine „one-to-many“-Relation, d. h. in einer Mensa kann es mehrere Personen geben.

Die Mensa soll von 11-14 Uhr geöffnet haben (openingTime = 11, closingTime = 14). Pro Tag gibt es 2.000 Gerichte die verkauft werden können (foodCount).
Mittels OCL Constraints definieren.

Die Syntax eines OCL-Ausdrucks ist die folgende:

context <identifier>
<constraintType> [<constraintName>]: <boolean expression>

„context“ ist ein Schlüsselwort um das Modellelement zu markieren, um das es geht. Der nachfolgende „identifier“ spezifiziert das Modellelement. „constraintType“ ist ein Schlüsselwort aus „inv“ (Invariante), „pre“ (Vorbedingung) oder „post“ (Nachbedingung). Die boolean expression ist der eigentliche Constraint.

Fangen wir einfach an: das Ende der Öffnungszeit, d. h. die closingTime, soll hinter dem Beginn der Öffnungszeit, d. h. der openingTime, liegen:

context Mensa
inv: closingTime > openingTime

Damit alle Mensabesucher etwas zu essen bekommen müssen immer Gerichte vorhanden sein. Das wollen wir jetzt mittels OCL spezifizieren:

context Mensa
inv: foodCount > 0

Der dem Schlüsselwort „context“ folgende Name benennt das Modellelement, um das es in der Abfrage gehen soll. In diesem Fall ist das die Mensa. In der zweiten Zeile sagen wir mit dem Schlüsselwort „inv“, dass es sich um eine Invariante handeln soll. Die boolean expression schließlich ist eine einfache Prüfung, ob noch Gerichte da sind, d. h. die Anzahl der restlichen Gerichte > 0 ist.
Jeden Abend werden die übrig gebliebenen Gerichte des Tages weggeworfen. In diesem Fall wäre die Invariante verletzt. Besser wäre es einen Ausdruck zu formulieren, der garantiert, dass **während der Öffnungszeiten** genügend Gerichte vorhanden sind:

context Mensa
inv: status = MensaStatus::opened implies foodCount > 0

Was sagt dieser Ausdruck aus?

“Die Mensa ist immer geöffnet und das impliziert, dass Gerichte vorhanden sind”

Jetzt haben wir zwar eine Garantie, dass während der Öffnungszeiten immer Gerichte vorhanden sind, leider darf die Mensa jetzt aber auch nie schließen, da sie sonst den Constraint verletzen würde.
Schauen wir noch einmal in unser Klassendiagramm. Die Klasse Mensa besitzt eine Funktion „sellFood()“. Diese können wir mit einer Vorbedingung verbinden (Mensa ist geöffnet), sodass unser Ausdruck passender ist:

context: Mensa::sellFood()
pre: status = MensaStatus::opened

Was bedeutet unser Ausdruck jetzt?

„Wenn die Funktion sellFood() aufgerufen wird (bzw. wenn ein Gericht verkauft wird) muss die Mensa geöffnet sein“

Die Mensa muss jetzt also nicht mehr 24 Stunden geöffnet haben und kann wieder zwischendurch schließen. Außerdem können wir nur Gerichte verkaufen, während die Mensa geöffnet hat. Dann haben wir alles was wir berücksichtigen wollten. Oder? Nicht ganz: wir müssen noch prüfen, ob der foodCount > 0 ist. Das können wir ebenfalls in die Vorbedingung mit reinnehmen:

context Mensa::sellFood()
pre: status = MensaStatus::opened
and
foodCount > 0

Collection-Operationen:
Collection-Operationen arbeiten auf Collections. Eine Collection ist eine Menge. Man kann sich das Java-Äquivalent als Array vorstellen, als Vector, als Liste, Map, … Mit einer Collection-Operation kann man nun bestimmte Eigenschaften oder Teilmengen einer Collection herausbekommen. Die Collection-Operation, um das Größenattribut einer Menge zu ermitteln, ist beispielsweise „size()“. Schauen wir auf unser Mensabeispiel:

context Mensa:
inv: getCurrentTime() = 12 implies guests -> size() > 250

Was bedeutet das?

“Wenn es 12 Uhr ist dann ist die Anzahl der Gäste in der Mensa größer als 250“

Dass diese Invariante nie verletzt wird klar, wenn man der Mensa einmal zu besagter Uhrzeit einen Besuch abgestattet hat.

Wir wollen uns jetzt den Ausdruck etwas genauer anschauen, denn wir haben gleich zwei neue Elemente verwendet:

1. Die Collection-Operation *„size()“*: wir rufen sie mit dem Pfeiloperator (->) auf

   Hinweis: Collection-Operationen lassen sich natürlich nur auf Elemente anwenden, die auch Collections sind. Der Aufruf

   openingTime -> size()

   wäre also falsch.
2. Wir rufen die Methode „getCurrentTime()“ auf. Bisher haben wir nur mit den Attributen direkt gearbeitet oder Methoden nur in Vor-/Nachbedingungen genutzt (s. o.). Sie direkt aufzurufen ist aber auch legitim. Voraussetzung ist, dass sie einen Rückgabewert haben. Der Aufruf

   sellFood() > 5

   ist also ebenfalls falsch, denn „sellFood()“ gibt nichts zurück.

Wichtig ist, dass Methodenaufrufe und Aufrufe von Collection-Operationen keineswegs das gleiche sind. Methodenaufrufe erfolgen in der Punktschreibweise (class.method()) oder direkt, Collection-Operationen werden mit dem Pfeiloperator ausgeführt.

Mit Hilfe von Collection-Operationen können hilfreiche Einschränkungen auf Teilmengen vorgenommen werden. Wie muss man sich das vorstellen?

Um den Andrang zur Mittagszeit abzumildern hat sich die Mensaverwaltung dazu entschlossen, zwischen 12 und 13 Uhr nur Personen mit Matrikelnummern kleiner als 120000 in die Mensa zu lassen. Wie kann man so einen OCL-Ausdruck formulieren? In Pseudocode sähe es etwa so aus:

```
for i = 1 to n do
    if guest[i].matrikelnummer < 120000 then
        ok
    else
        not okay
```

Wir erinnern uns: in OCL gibt es keine Konstrukte um Schleifen zu simulieren. Wir müssen also irgendwie direkt an unsere Teilmenge der Gäste herankommen. Helfen kann uns dabei die Collection-Operation „select()“.

Die ***„select()“*-Operation** gibt alle Elemente einer Menge zurück, für die ein angegebener OCL-Ausdruck wahr ist. Als Beispiel:

```
context Numbers
inv: elements -> select (value % 2 = 0)
```

Was bedeutet dieser Ausdruck?

> „Nimm alle Elemente aus der Menge ‚Elements‘ und gib die zurück, deren Wert gerade ist (modulo 2 = 0)“

Mit diesem Beispiel ist es jetzt sehr einfach, unser Matrikelnummer Kriterium zu spezifizieren:

```
context Mensa
inv: getCurrentTime() >= 12
    and
    getCurrentTime() <= 13 implies guests -> select(matrikelnummer < 120000)
```

Alternativ können wir den Ausdruck auch mit der **Collection-Operation *„reject()“*** spezifizieren. „reject()“ arbeitet invers zu select:

```
context Mensa
inv: getCurrentTime() >= 12
    and
    getCurrentTime() <= 13 implies guests -> reject(matrikelnummer >= 120000)
```

Was ist passiert?

Da sich “reject()” invers zu “select()” verhält muss lediglich der Vergleichsoperator umgedreht werden, um die gleiche Menge zu erhalten. Würde man beim Einsatz von „reject()“ ebenfalls auf „matrikelnummer < 120000“ prüfen, würden Gäste mit einer Matrikelnummer kleiner als 120000 zur Mittagszeit keinen Einlass in die Mensa erhalten.

Die Collection-Operation *„collect()“*:
„collect()“ sammelt in einer Menge von jedem Element das spezifizierte Attribute in. Wie muss man sich das vorstellen?

```
context Mensa
inv: guests -> collect(name) -> includes(‘Hans’)
```

Was bedeutet dieser Ausdruck?

“In der Mensa ist immer eine Person mit dem Namen ‚Hans‘ zu Gast“

Analysieren wir den Ausdruck: auf das Attribut „guests“ der Klasse „Mensa“ rufen wir die „collect()“-Operation auf. Zuvor haben wir bereits „size()“ auf „guests“ aufgerufen, das hier ist nichts anderes.
„collect()“ gibt uns eine Menge der Namen der in der Mensa anwesenden Gäste zurück. Auf diese Menge wenden wir die Operation „includes()“ an. „includes()“ gibt genau dann *true* zurück, wenn ein Element in der Menge mit dem in „include()“ angegebenem Objekt übereinstimmt. In diesem Beispiel ist das der Name „Hans“. Wichtig dabei ist zu beachten, dass „collect()“ Multimengen zurückgibt, d. h. Elemente können mehrfach vorkommen. Möglicherweise gibt es also mehr als einen Hans in der Mensa, vielleicht heißen auch alle Gäste so. „includes()“ würde in jedem Fall *true* zurückgeben.

Die Collection-Operation *„forAll()“*:
„forAll()“ gibt genau dann *true* zurück, wenn **alle** Elemente einer Menge die übergebene Bedingung erfüllen. Als Beispiel:
Da es in letzter Zeit in der Mensa gehäuft zu Identitätsdiebstahl gekommen ist und einige Personen auf anderer Leute Rechnung gegessen haben, sind ab sofort alle Besucher verpflichtet, ihren Namen am Eingang zu nennen. Dabei darf kein Name mehrfach vorkommen, da sonst nicht sichergestellt werden kann, dass korrekt abgerechnet wird:

context Mensa
inv: guests -> forAll(guestA, guestB
| guestA <> guestB implies guestA.name <> guestB.name)

Was bedeutet dieser Ausdruck?

“Keine zwei Gäste dürfen den gleichen Namen haben”

Wichtig ist in diesem Beispiel, dass wir zuvor auf *guestA <> guestB* prüfen. „forAll()“ bildet das kartesische Produkt und würde sonst allen Gästen den Zutritt verwehren. Denn wenn *guestA = guestB* gilt, dann auch *guestA.name = guestB.name*.

Zum Schluss: Abschließend wollen wir noch Nachbedingungen in OCL abdecken. Wir erinnern uns: wir haben eine Vorbedingung spezifiziert um abzusichern, dass nur dann Gerichte verkauft werden, wenn die Mensa auch tatsächlich geöffnet hat. Wegen einer neuen Richtlinie muss ab sofort immer dokumentiert werden, wenn ein Gericht verkauft wurde. Es soll also der „foodCount“ um 1 reduziert werden, nachdem das Gericht serviert wurde. Das geht in OCL folgendermaßen:

context Mensa::sellFood()
post: foodCount = foodCount@pre – 1

Was ist in diesem Beispiel passiert?

Wieder beziehen wir uns auf die Methode „sellFood()“. Wir wollen eine Nachbedingung spezifizieren, also verwenden wir das Schlüsselwort „post“. Wir wissen, dass der „foodCount“

nachdem wir ein Gericht verkauft haben genau um 1 kleiner sein muss als vorher. Den vorherigen Wert von „foodCount“ bekommen wir aus der Vorbedingung. Diese wird mit „@pre“ referenziert. Der Rest des Ausdrucks ist dann einfache Arithmetik.

Ein Blick auf ein Folienbeispiel: auf Folie 45 wird eine Nachbedingung für die Methode „book()“ spezifiziert. Sie soll bedeuten, dass

- Nach dem Buchvorgang die passengers-Assoziation ein zusätzliches Objekt enthält und
- Ein Passenger-Objekt existiert, dessen Attribute mit den Parametern von „book()“ übereinstimmen

Es kann also sein, dass bereits ein Passagier gleichen Alters, mit dem gleichen Namen und dem gleichen Bedürfnis nach Unterstützung auf diesen Flug gebucht ist. In dem Fall würde *exists()* das Durchsuchen der *passenger*-Collection abbrechen, sobald dieser gefunden ist. Ob der soeben zugebuchte Passagier tatsächlich der Assoziation hinzugefügt wurde ist dann unklar. Besser wäre der OCL-Ausdruck folgendermaßen:

```
context Passenger::book(…)
post: flight.passengers -> size() – flight.passengers@pre -> size() = 1
        and
        flight.passengers ->
                select(p: Passenger | p.name = name and p.age = age and
                p.assistance = needsAssistance = assistance) -> size()
                –
flight.passengers@pre ->
        select(p: Passenger | p.name = name and p.age = age and p.assistance =
        needsAssistance = assistance) -> size()
        = 1
```

Mit diesem Ausdruck garantieren wir, dass wir in weiteres Objekt in unserer *passengers*-Collection haben. Dadurch, dass wir *select()* benutzen können wir außerdem auf die genauen Daten abfragen und so sicherstellen, dass es sich bei dem neu hinzugekommenen Passagier um jenen handelt, der soeben gebucht hat.

Zusammenfassung des Kapitels

In diesem Kapitel haben wir die Object Constraint Language (OCL) kennen gelernt. Wir haben die grundlegende Syntax von OCL-Ausdrücken gesehen und die verschiedenen „Ausdruckstypen“ – Invarianten, Vor- und Nachbedingungen. Wir haben gesehen, dass man in OCL-Ausdrücken sowohl klasseneigene Methoden (class.method()) nutzen kann als auch so genannte Collector-Operationen für Attribute, die Mengen darstellen (in unserem Mensabeispiel die Menge der Gäste).

Welche Vorteile bringt OCL?

Durch den Einsatz von OCL kann bereits sehr früh mit dem Testen begonnen werden. Fehler in Modellen und Designs fallen dadurch noch vor dem Implementierungsbeginn auf und können schnell und kostengünstig korrigiert werden. Außerdem lassen sich mit Hilfe von Tools aus den OCL-Ausdrücken automatisch Java-Assertions erzeugen, wodurch die Codequalität erheblich steigt, da das Programm sofort aussteigt wenn eine Assertion verletzt wird. Dadurch findet man Fehler sehr viel schneller.

KAPITEL 1.5: ECLIPSE MODELING FRAMEWORK

Im letzten Kapitel: Petrinetze

- Grundlage für Geschäftsprozessmodellierung und für Process Minig
- Analyse von Systemen – Simulation und Verifikation
- Definition der Semantik von Workflow-Netzen

Eclipse Modeling Framework

In Kapitel 1.1 haben wir uns mit der Model-Driven Architecture auseinandergesetzt. Wir haben gesehen welche Vorteile und Möglichkeiten die MDA bietet. Beispielsweise lassen sich aus Modellen automatisch TestCases erstellen, man kann Codegeneratoren bauen die Entwicklern viel stupide Programmierarbeit abnehmen und Modelle lassen sich untereinander transformieren sofern die dazu benötigten Transformationsregeln vorhanden sind. Dennoch waren die vorgestellten Techniken eher theoretischer Natur. In diesem Kapitel wollen wir uns ansehen, wie sich einige der MDA-Ideen in die Realität umsetzen lassen.

Zuerst wollen wir einige Begrifflichkeiten klären: wie bereits in Kapitel 1.2 erwähnt ist die MDA von der Object Management Group (OMG) spezifiziert. Das ist die gleiche Gruppe die auch die Unified Modeling Language (UML) spezifiziert, sodass die Verbindung zwischen MDA und UML auf der Hand liegt.

Außer MDA und UML stammt auch die Meta Object Facility von der OMG. Die MOF ist das Modell der verschiedenen Modellebenen aus Kapitel 1.2, d. h.:

Meta-Meta-Modell (MOF 2.0, Meta-Modell für Meta-Modelle)

Meta-Modell (UML 2.0-Stack, Spezifikation der Diagrammtypen, …)

Modell (Beispiel Klassendiagramm „Mensa")

Instanzen (Mensa der TU → reales Objekt)

Das Eclipse Modeling Framework um das es in diesem Kapitel geht, das EMF ist eine Realisierung des MOF-Konzeptes. Wie der Name schon andeutet ist es eine Realisierung im Eclipse-Tool. Die Implementation ist in Java erfolgt.

Bei dem EMF handelt es sich um ein Modellierungsframework. Was bedeutet das? Mit EMF lassen sich nicht nur einfach Diagramme erstellen, so wie es bei reinem UML der Fall wäre (z.B. Astah), sondern das Framework bringt eine Reihe weiterer Tools und Funktion zur Unterstützung mit. Dazu gehören unter anderem ein Tool zur Codegenerierung und ein einfacher Editor.

Der Codegenerator erlaubt zum einen, dass aus den angelegten Modellen fertige Java-Klassen erstellt werden können. Zum anderen kann mit dem Generator Code für einen EMF-Editor erzeugt werden. Was es damit auf sich hat wollen wir uns ansehen. Vorher betrachten wir, welche Möglichkeiten zum Erstellen eines Modells es im EMF gibt:

- Aus annotierten Java-Klassen:

  Das ist der „other way round“, d. h. aus fertigem Code wird ein Modell generiert
- Aus XML-Dokumenten;

  Wir erinnern uns: ein grundlegender Bestandteil der MDA ist das XML Metadata Interchange; dieses wandelt Modelle in XML um und ermöglicht so den Austausch zwischen verschiedenen Komponenten
- Aus anderen Modellierungstools: dann werden die Modelle direkt als UML importiert
- Direkt mit Hilfe des Editors

Aus welchen Bestandteilen besteht das EMF? Im Prinzip wurden schon alle Bestandteile vorgestellt, an dieser Stelle aber nochmal eine „formalere“ Auflistung:

- EMF.EMOF: EMOF steht für essential MOF

  Wie der Name schon andeutet handelt es sich bei EMOF nicht um eine komplette Realisierung der MOF, sodass mit dem EMF also nicht alles dargestellt werden kann, was mit MOF möglich ist. Dennoch ist EMOF für die meisten Zwecke ausreichend.
- EMF.Ecore

  EMF.Ecore ist der zentrale Bestandteil des EMF. Hier befinden sich:
  - Das Meta-Modell, um Modelle zu beschreiben (UML-Modelle)
  - Laufzeitunterstützung für Modelle
    - Benachrichtigung von Modellen bei Änderungen
  - Persistenzunterstützun durch Standard XML-Serialisierung
  - API zur generischen Veränderung von EMF-Modellen
- EMF.Edit

  Eine Sammlung generischer und wiederverwendbarer Klassen, um Editoren für EMF-Modelle zu erstellen.
- EMF.Codegen

  Ermöglicht die Generierung von Code der für einen EMF-Editor benötigt wird.

Sehen wir und die einzelnen Bestandteile genauer an:

EMF.EMOF:

Beim EMOF handelt es sich um einen Teil der MOF 2.0-Spezifikation. Wie bereits gesagt lässt sich damit nicht alles darstellen was mit MOF 2.0 möglich ist.

EMF.Ecore:

EMF.Ecore enthält neben dem Meta-Model, welches die erstellbaren Modelle beschreibt noch eine Laufzeitunterstützung für Modelle. Was bedeutet das?
Stellen wir uns vor wir haben eine GUI-Anwendung. Diese Anwendung basiert auf dem MVC-Pattern. Im Model arbeiten wir auf einer Datenbank. Alle Änderungen an den Daten werden vom Controller aus dem Model an das View (die GUI) weitergegeben. Während des Programmlaufes wird nun beispielsweise ein neuer Eintrag in der Datenbank angelegt. Der Controller ist als Listener im Model eingetragen und bekommt diese Änderung mit. Er gibt dem View Bescheid,

welcher einen Refresh macht und sich die neuen Daten holt und anzeigt.

Das ist das aus Java bekannte Observer-Pattern. Da es sich bei dem EMF um ein Eclipse-Tool handelt liegt der Schluss nahe, dass man sich an diesem Pattern orientiert hat. Tatsächlich ist es so, dass Modelle sich als Listener eintragen können und dann Änderung in anderen Modellen erfahren. Wieder erkennbar ist hier der MDA-Ansatz: alle Modelle sollen aktuell bleiben.

Neben diesem Updatemechanismus gibt es im EMF.Ecore außerdem eine Persistenzuntersützung. Diese verwendet die Standard XML-Serialisierung. Was bedeutet das?

Persistenz bedeutet „Datenerhaltung"/"Datenspeicherung". Ein Beispiel aus dem Hardware-Bereich: der Arbeitsspeicher eines PCs ist flüchtig. Schaltet man den Rechner aus und wieder an, dann ist alles, was im Arbeitsspeicher gespeichert war weg. Die Festplatte eines Rechners wiederum ist persistent. Schaltet man den Rechner aus und wieder ein sind noch alle Daten da. Selbst wenn man den Rechner 1 Jahr ausgeschaltet lässt sind danach immer noch alle Daten da. Die Datenerhaltung unterliegt gewissen physikalischen Prozessen, und es kann sein, dass nach langer Zeit tatsächlich Daten verloren gehen, aber das wollen wir hier nicht betrachten. Was ist XML-Serialisierung? Serialisierung ist ebenfalls ein Konzept aus Java. Lässt man eine Java-Klasse von der Klasse „Serializable" erben und teilt ihr eine Serializable-ID zu, dann werden alle Instanzen der Klasse beim Schließen des Programms gespeichert. Betrachten wir folgendes Beispiel:

```
class Mensa extends Serializable
{
        private int _foodCount;
        private int _openingTime;
        private int _closingTime;
}
```

Natürlich besitzt unsere Klasse auch noch Funktionen, diese sind aber momentan nicht relevant. Wichtig ist zu wissen, dass für jede Instanz die momentanen Werte in den Membervariablen gespeichert werden. Wenn wir also noch 900 Gerichte übrig haben und wir aus Versehen unsere Mensa-Anwendung schließen, dann zeigt uns die Anwendung beim nächsten Start noch immer 900 verfügbare Gerichte an.

Das Datenformat, in dem die Daten aus serialisierbaren Klassen gespeichert werden kann unterschiedlich sein. Möglich sind einfache Textdateien (.txt), Datendateien (.dat), oder: wie beim EMF der Fall – XML-Dateien. Das ist also XML-Serialisierung.

EMF.Edit:

Hierbei handelt es sich um eine Sammlung generischer, also wiederverwendbarer Klassen, die bei der Erstellung eines Editors unterstützen können. Warum müssen die Editoren zuerst erstellt werden? Da es ganz verschiedene Modelltypen gibt, benötigt man auch verschiedene Editoren. Man darf sich einen Editor dabei nicht wie einen reinen Texteditor vorstellen, denn dann würde mit Sicherheit einer ausreichen, sondern es handelt sich dabei um grafische Editoren, die je nach Anwendungszweck verschiedene Optionen/ Schaltflächen/ Auswahlmöglichkeiten bieten.

EMF.Codegen:

Damit man nicht den gesamten Code für einen Editor von Hand schreiben muss gibt es das Codegenerierungsframework. Das Framework baut dabei auf den generischen Klassen aus EMF.Edit auf.

Das Framework benutzt ein so genanntes „genmode", welches Informationen aus den Modellen enthält, aus denen Java-Klassen erstellt werden soll. Dabei überwacht das genmodel die Modelle, um stets aktuell zu sein. Im Gegenzug teilen die Modelle ihre Informationen ebenfalls mit dem genmodel.

GEF (Graphical Editing Framework)

Das GEF erlaubt die grafische Darstellung von Modellen. Das ist erstmal dasselbe, was auch Tools wie Astah erlauben, GEF bietet jedoch noch weitere Funktionalität: durch Maus- und Tastatureingaben kann man noch im Modell eine Simulation durchführen. Dabei werden die Eingaben des Benutzers verarbeitet, interpretiert und darauf reagiert. Es handelt sich also bei GEF-Modellen bereits um eine Art ausführbares Programm. Der Umfang dessen, was man tatsächlich simulieren kann reicht natürlich allein aufgrund des Abstraktionsgrades nicht an ein ausführbares Programm heran, dafür kann man aber alle Änderungen die verursacht wurden wiederholen oder rückgängig machen (undo/redo, gleicher Mechanismus wie in jedem Officeporgramm/ Bildbearbeitungsprogramm/ …). Von sich aus bieten Modell eine solche Simulationsunterstützung allerdings nicht. Die Aktionen, auf die das Modell reagieren soll müssen zuvor mit Codeausschnitten festgelegt werden.

Ebenso wie das EMF verwendet auch GEF das Model-View-Controller Pattern. Das erlaubt zum einen die selbstständige Aktualisierung der Modelle (mittels Listener und Reaktion auf Änderungen), zum anderen können die Schichten so getrennt und Funktionalität wiederverwendet werden. Die konkrete Umsetzung von MVC in GEF erfolgt über die so genannte „EditPartFactory": zu jedem Model gibt es einen EditPart. Dieser nimmt die Daten aus dem Model und reicht sie an die Komponente weiter, die für die graifsche Darstellung zuständig ist. Die EditPartFactory bzw. die einzelnen EditParts sind also eine Realisierung der Controllerschicht.

GMF (Graphical Modeling Framework)

Nachdem wir nun EMF und GEF vorgestellt haben, wollen wir die Kombination der beiden betrachten: das Graphical Modeling Framework, kurz GMF. Was ist das Ziel des GMF? Es sollen

- Modelle einfach erstellt oder importiert werden können, und das aus verschiedenen Quellen (XMI, Editor, ,Modellierungstools …)
- Modelle ohne weiteres zu tun automatische über Änderungen benachrichtigt und dann aktualisiert werden
- Das Ganze soll grafisch aufbereitet werden
- Modelle sollen gespeichert werden können
- Änderungen sollen wiederhergestellt oder rückgängig gemacht werden können
- …

Man könnte an dieser Stelle anmerken, dass viele der oben genannten Funktionen eigentlich selbstverständlich sein sollten, so zum Beispiel das Speichern von Modellen und das Rückgängigmachen von Änderungen. Zu beachten ist aber, dass das Ziel immer ist, die Anforderungen der MDA umzusetzen. Es reicht also nicht, Modelle einfach zu speichern oder

grafisch zusammen zu klicken. Es ist essentiell, dass die Semantik ebenfalls vorhanden ist, d.h. die Beziehungen von Elementen und Modellen untereinander (Stichwort MVC), es muss eine funktionsfähige XMI-Schnittstelle geben etc. MDA ist erstmal nur ein theoretischer Ansatz, deswegen gibt es keine komplette und fertige „MDA-Suite“ die man benutzen kann. Die Konzepte müssen zuerst realisiert werden, in diesem Fall in Eclipse.

Die Vorteile einer Kombination vom EMF und GEF sind eine gesteigerte Effektivität durch die automatische Aktualisierung der Modelle sowie durch die automatische Codegenerierung, die den Entwicklern mehr Zeit für wichtige Dinge gibt. Ein andere wichtiger Faktor ist allerdings der der Kosten. Wir haben in Kapitel 1.0 gesagt, dass ein prägender Faktor bei der Softwareentwicklung Budgets sind. Durch die Verwendung des GMF können Komponenten wiederverwendet werden und Wissen bleibt in Modellen erhalten (vgl. Kapitel 1.1). Es muss also nicht immer als „von 0“ programmiert werden. Grade bei größeren Projekten ist es von unschätzbarem Vorteil, wenn man bereits eine bewährte Basis hat. Man könnte auf andere Tools und Frameworks zurückgreifen, hat man jedoch bereits gute Erfahrungen gemacht, wird man natürlich auf die benutzte Technik zurückgreifen.

Ein Nachteil von EMF generell ist, dass nicht der komplette UML-Sprachschatz, sondern nur eine Teilmenge (Essential MOF) genutzt wird. Komplexe Sachverhalte lassen sich deswegen unter Umständen mit EMF nicht abbilden. In den meisten Fällen ist EMF aber ausreichend.

KAPITEL 2.0: QUALITÄTSMANAGEMENT

Im letzten Kapitel (gesamt):

- Einführung in die Softwareentwicklung/-Konstruktion: welche Probleme gibt es?
- Einführung in OCL: frühes und kostengünstiges Testen
- MDA: Modelle als Mittelpunkt der Softwareentwicklung
- EMF: Realisierung von MDA mittels Eclipse

Qualitätsmanagement

Was ist Qualitätsmanagement? Um ein Softwareprodukt zu entwickeln braucht man nicht nur Programmierer, die Ideen in Code gießen. Neben der eigentlichen Implementierung ist es wichtig, auch gewisse Standards einzuhalten. Wir haben uns in Kapitel 1.2 mit OCL auseinandergesetzt und festgestellt, dass man damit schnell und kostengünstig erste „Tests" auf Modellen definieren kann. Sicherlich ist es richtig, Software von Anfang an zu testen. Dieser Trend hat sich in den letzten Jahren abgezeichnet. Zwar wird durch den Aufbau einer Test-Infrastruktur zu Beginn eines Projektes der Zeitpunkt des eigentlichen Entwicklungsstarts heraus gezögert und die Entwicklung läuft danach langsamer ab, als wenn man gar nicht testen würde. Durch frühes und regelmäßiges Testen verringert sich allerdings die Zahl der Showstopper in der Entwicklung, d. h. es treten weniger Probleme auf, die eine Weiterentwicklung komplett verhindern bis das entsprechende Problem behoben ist, und nachdem das Produkt auf den Markt gebracht wurde sind weniger kritische Fehler zu erwarten. Natürlich ist kein Produkt fehlerfrei nur weil man es vor der Veröffentlichung oft und gründlich getestet hat, aber man kann viele Fehler vorzeitig aufspüren. Schauen wir uns kurz ein Beispiel an. Die Funktion multiplyWithTwo soll auf einen gegeben Wert n n-mal 1 addieren. Das entspricht einer Multiplikation mit 2:

```
public void multiplyWithTwo(int n)
{
        int result = 0;

        if(n = 0) return 0;

        result = result + 1 + multiplyWithTwo(n – 1);
}
```

Eine sehr einfache Funktion. Man könnte meinen, dass für so etwas kein Unit Test oder ein Test durch einen Mitarbeiter notwendig wäre. Aber schauen wir uns den Code genauer an: in der 3. Zeile haben wir eine if-Abfrage:

```
if(n = 0) …
```

Wenn wir genauer hinsehen bemerken wir, dass wir hier einen Semantikfehler haben. Was wir wollen ist if(n == 0), einen Vergleich auf 0. Tatsächlich haben wir aber eine Zuweisung: der Wert 0 wird an die Variable n zugewiesen. Dann wird 0 zurückgegeben und die Abarbeitung der Funktion terminiert. Egal welchen Wert wir also multiplyWithTwo übergeben, wir werden immer 0 als Ergebnis erhalten.

Wir sehen also, so einfach unser Code auch sein mag, es ist immer besser ihn zu testen. Ein Unit Test, vorausgesetzt er ist korrekt formuliert, würde uns beim ersten Durchlauf sofort auf das Problem aufmerksam machen. Jetzt mag man meinen unser Beispiel ist nicht sonderlich kritisch. Was ist aber, wenn die Funktion in einem größeren Kontext benutzt wird, beispielsweise um das Weihnachtsgeld aller Mitarbeiter einer Firma zu verdoppeln? Die Mitarbeiter werden sich bedanken, wenn sie eine Überweisung in Höhe von 0€ erhalten.

Testen ist allerdings nur ein Teilaspekt von Qualitätsmanagement und wir werden uns ab Kapitel 3 noch genauer damit auseinandersetzen. Ein anderer Aspekt ist das definieren und auch „leben“ von Prozessen und Arbeitsabläufen. Dazu gehören unter anderem Code Reviews, das Schreiben von Testprotokollen, das Abarbeiten selbiger, Dokumentieren von Fehlern, bspw. in einem System wie JIRA (Anwendung zur Fehlerverwaltung, ähnlich Bugzilla o. a.). Durch die Standardisierung von all diesen Abläufen soll gewährleistet werden, dass die Qualität des fertigen Produktes (aber auch des Zwischenproduktes) konstant bleibt bzw. gesteigert wird. Es könnte beispielsweise einen Ablauf für die Portierung einer Softwarekomponente geben. Stellen wir uns vor unsere (mittlerweile korrigierte) Funktion multiplyWithTwo kommt in einer Komponente calculateThirteenthSalary zum Einsatz. Jahrelang lief diese Komponente in einer Firma auf einem Windows Server 2003. Im Zuge einer IT-Modernisierung soll jetzt auf Windows Server 2012 gewechselt werden. Um die Komponente zur Weihnachtsgeldberechnung auf das neue System zu portieren wurde folgender Arbeitsablauf entwickelt (grob skizziert):

- Kompilieren des Codes mit aktuellem JDK (für Windows Server 2012)
  - OK, Ausführen der Unit Tests
    - OK, weiter mit nächstem Schritt
    - Nicht OK, suchen und beheben von Fehlern
  - Nicht OK, suchen und beheben von Fehlern (bspw. Kompilereinstellungen ändern)
- Ausführen des Binarys
  - Lässt sich Ausführen: OK, weiter mit nächstem Schritt
  - Nicht OK, suchen und beheben von Ursachen (bspw. nicht ausreichend Rechte zur Ausführung)
- Testen des Binarys mit Testdaten
  - Istwert = Sollwert, OK
  - Nicht OK, Anlegen eines Fehlereintrages (z. B. im JIRA) und zurück an Entwicklung

Dieser Prozess wird solange durchlaufen bis sich die Komponente schließlich fehlerfrei ausführen lässt. Fehlerfrei bedeutet in diesem Zusammenhang: es treten keine offenkundigen Fehler auf, d. h. das Programm stürzt nicht ab, die Sollwerte entsprechenden Istwerten, etc. Ob die Komponente komplett fehlerfrei arbeitet kann man nicht sagen.

Wir haben gesehen, dass Testen wichtig für die offensichtliche Qualität eines Produktes ist, d. h. für die Kernfunktionalität. Was bringt uns aber ein mächtiges Programm wenn die Bedienung grauenhaft ist. Oder wenn die Bedienung eines Onlinebanking-Programms wunderbar bequem ist, sämtliche Daten aber im Klartext über das Netzwerk geschickt werden. Und was bringt es uns, ein tolles Programm für Solaris SPARC entwickelt zu haben, sämtliche unserer Kunden aber ein Binary für Windows Server 2012 haben wollen?

Alle diese Aspekte fallen unter den Begriff Softwarequalität. Der Begriff bezeichnet dabei die so genannte „Gebrauchsqualtität“, das ist die Qualität während der Benutzung des Programms (stürzt nicht ab, ruckelt nicht ständig, Daten werden gespeichert wenn man auf den Speichern-Button klickt und landen nicht im Datennirvana, …), und die „äußere und innere Qualität“, das ist bspw. das Problem der Portierbarkeit (Bsp. von Solaris SPARC nach Windows Server 2012), die Usability, d. h. wie gut kommen Benutzer mit dem Programm klar, wie leicht/schnell können sie es erlernen, ist das Programm effizient (beschränkt es sich auf das wesentliche oder zeigt es vor jedem Klick eine 5 Sekunden Animation an – Zeiteffizienz), …

Es ist wichtig zu wissen, dass man nicht alle diese Ziele der Softwarequalität gleichzeitig maximieren kann. In einigen Bereichen ist immer ein gewisser Trade-Off erforderlich,

beispielsweise zwischen der Sicherheit eines Programms und der Benutzbarkeit. Ein aktuelles Beispiel:

Gesucht wird eine Messaging-Anwendung für Smartphones, welche einen hohen Bedienkomfort bietet, gleichzeitig aber eine Ende-zu-Ende-Verschlüsselung implementiert, so dass niemand anfallende Daten abgreifen kann. Für die gewünschte Verschlüsselung sind bei der Installation der Anwendung gewisse Schritte erforderlich: es müssen private Schlüssel generiert werden, und zur Initialisierung der Chatkontakte müssen Schlüssel ausgetauscht werden. Im Vergleich zu anderen Anwendungen, die einfach das gesamte Telefonbuch zu den Kontakten hinzufügen oder die Möglichkeit bieten, mit Fremden zu schreiben, ist das ein nicht zu vernachlässigender Aufwand, der viele Anwender dann davon abhält, die Anwendung zu nutzen.

Qualitätslenkung vs. -prüfung

Die bisher vorgestellten Beispiele entstammten größtenteils dem Bereich der Qualitätsprüfung. Qualitätsprüfung, oft auch Qualitätssicherung genannt, ist auf ein konkretes Produkt angewandtes Qualitätsmanagement. Hier wird getestet, Testprotokolle erstellt, Code Reviews durchgeführt, Audits, …

Qualitätslenkung ist der „abstraktere“ Teil des Qualitätsmanagements. Hier werden allgemeine Vorgehensweisen, Normen und Standards, Arbeitsabläufe, … festgelegt. Alles was hier definiert wird, gilt für alle Projekte. Qualitätslenkung wird auch „konstruktives QM“ genannt, d. h. hier wird etwas aufgebaut, nämliche eine Infrastruktur, ein Grundgerüst, auf dem dann das „analytische QM“, die Qualitätsprüfung, aufsetzen kann.

In der *Qualitätsprüfung* verwendete Techniken und Maßnahmen umfassen (Code-)Reviews, statische Analysen (mittels Tools, bspw. cppcheck für C++-Sourcen) oder Model Checking für den statischen Bereich, d. h. es wird kein Programm ausgeführt. Der dynamische Bereich umfasst das Testen anhand von Unit-, Integrations-, Regressions-, und Akzeptanztests oder die Simulation (beispielsweise mittels OCL) von Programmabläufen. Wir werden in Kapitel 3 einen ausführlichen Einblick in verschiedene Testtechniken und –Abläufe erhalten. An dieser Stelle soll diese kurze Vorstellung ausreichend sein.

Nun stellt sich die Frage, inwiefern die Qualität zugrundeliegender Prozesse (Arbeitsabläufe, …) die Qualität des fertigen Endproduktes beeinflusst. Wir haben diese Frage für den Bereich des Testens bereits weiter oben beantwortet. Wie sieht es aber in anderen Bereichen aus? Sicherlich ist es beispielsweise für die Qualität eines Produktes nicht förderlich, wenn plötzlich der Releasetermin um 2 Monate nach vorne geschoben wird. Auch wenig konstruktiv wird es sein, wenn die beteiligten Entwickler an 2 Projekten gleichzeitig mitarbeiten müssen und ständig aus der Entwicklung abgezogen werden. Eine erfahrene Projektleitung weiß um die Risiken im Entwicklungsprozess und wird versuchen, die Produktionsprozesse so optimal wie möglich zu gestalten, damit die Qualität des Endproduktes nicht zu weit von den Erwartungen abweicht.

Interessant ist zu wissen, dass seit dem Aufkommen dieser Art der Projektplanung, dem so genannte „Requirements Engineering“ (d. h. es wird zuerst viel Grundlagenarbeit geleistet, bevor mit der eigentlichen Arbeit begonnen wird) die durchschnittlichen Kosten für die Entwicklung von Software gesunken sind. Natürlich liegen die realen Kosten aufgrund von steigender Komplexität der Software, einer größeren Anzahl involvierter Mitarbeiter, … höher als vor einigen Jahren, mit dem alten Ansatz des „drauf-los-Programmierens“ war aber ein deutlich höherer Aufwand für Bugfixing und Nachtesten verbunden, eben weil man relativ planlos drauflos programmiert hat. Die Einführung von Prozessen und die stetige Verbesserung selbiger führt außerdem zu einer erhöhten Performanz und Produktivität in der Entwicklung: man kann auf erprobte Mechanismen zurückgreifen und muss nicht jedes Mal alles von neuem „erlernen“, entwickeln, …

Es gibt für den Bereich der Prozessqualität eine Reihe von Standards der International Standardization Organisation (ISO): ISO 900x

Die Grundlage dieser Standards ist „ISO 9000 – Qualitätsmanagementsysteme – Grundlagen und Begriffe“. Hier werden Grundlagen eines Qualitätsmanagementsystems erläutert. Der eigentlich interessante Teil der Reihe ist der Standard ISO 9001. Dieser definiert die „Infrastruktur“ eines Qualitätsmanagementsystems: dieses besteht aus einem Organisationshandbuch an dem sich alle in der Entwicklung befindlichen Projekte orientieren. Das Handbuch dokumentiert außerdem die Organisations-Qualitäts-Prozesse, d. h. das allgemeine Projekt-Qualitäts-Management. Es ist möglich, sich für ISO 9001 zertifizieren zu lassen, dann muss das Qualitätshandbuch von einer externen Stelle zertifiziert werden (eine Art ISO-TÜV).

Kann man eine Zertifizierung für ISO 9001 vorweisen, bedeutet das jedoch nicht sofort, dass die Produkte der Firma von besonderer Qualität sind. Da lediglich das Qualitätshandbuch überprüft wird sagt dies noch nichts über die entwickelten Produkte aus. Zur Ergänzung von ISO 9001 eignet sich das so genannte Capability Maturity Model Integrated (CMMI). CMMI beleuchtet ein wenig mehr die technischen Aspekte des Qualitätsmanagements. Es unterstützt bei der Einführung von Techniken und Methoden und der Definition von Standards, die dann im Qualitätsmanagement Verwendung finden. ISO 9001 hingegen befasst sich eher mit allgemeinen Arbeitsabläufen und Prozessen. CMMI ist dabei kein gleichwertiges Modell zu ISO 9001. Es deckt einige Bereiche nicht ab. Außerdem wird von Kunden meistens ISO 9001 gefordert.

KAPITEL 2.1: GRUNDLAGEN DER SOFTWAREVERIFIKATION

Im letzten Kapitel:

- Motivation für Qualitätsmanagement: sinkende Kosten, steigende Produktivität
- Unterteilung in konstruktives QM und analytisches QM: Qualitätslenkung und Qualitätsprüfung
- Zertifizierbares Qualitätsmanagement: ISO 9001

Grundlagen der Softwareverifikation

Wir haben uns bereits im letzten Kapitel ein wenig mit dem Testen von Code beschäftigt. Wir wollen uns jetzt genauer mit dem Erstellen und Durchführen von Tests beschäftigen. Betrachten wir zuerst ein Codebeispiel und erläutern daran einige Begrifflichkeiten:

```
public static int pricePerTicket = 5;
public static int ticketsLeft = 20;
public int availableMoney = args[1];

public void buyTicket()
{
	if(validateMoney() && ticketsLeft > 0)
	{
		availableMoney -= pricePerTicket;
		decreaseTickets();
	}
}

public bool validateMoney()
{
	if(availableMoney >= pricePerTicket)
		return true;

	return false;
}

public void decreaseTickets()
{
	if(ticketsLeft > 0)
		ticketsLeft--;
}
```

Bei unserem Beispiel handelt es sich um ein System zum Ticketverkauf. Zu Beginn wird der Wert aus args[1] an die Variable availableMoney zugewiesen und so das Kontoguthaben festgelegt. Die Variable pricePerTicket legt den Einzelpreis für ein Ticket fest, die Variable ticketsLeft zeigt an, wie viele Variablen noch vorhanden sind.

Der Code ist nicht sonderlich gut durchdacht und weist einige Fehler auf. Wir wollen diese nun betrachten:

Angenommen ein Entwickler benutzt die oben aufgeführte Klasse in seinem Code. Er möchte also das Ticketbuchungssystem realisieren. Statt die vordefinierten Funktionen zu benutzen arbeitet er allerdings direkt auf den Membervariablen der Klasse. Da diese als public deklariert sind ist das kein Problem. Unser Entwickler schreibt also folgenden Code:

```
for(int i = 0; i < ticketsToReserve; ++i)
{
	availableMoney -= pricePerTicket;
	ticketsLeft--;
}
```

Man kann sich schon denken was an diesem Codebeispiel ungünstig ist: es wird nicht auf das verbleibende Guthaben geprüft und ebenfalls nicht auf die Anzahl verbleibender Tickets. Falls die Variable ticketsToReserve mal sehr groß sein sollte, dann würden sowohl die Variable availableMoney als auch ticketsLeft danach einen negativen Wert haben.

Auch wenn unsere Ticketklasse schlecht geschrieben ist liegt in diesem Beispiel eine **Fehlerhandlung** des Entwicklers vor, der die Klasse benutzt. Eine Fehlerhandlung wird dabei bspw. durch eine schlechte Ausbildung oder fehlende vorgegebene Standards bei der Entwicklung verursacht. Es ist generell kein guter Stil außerhalb von Klassen mit internen Membervariablen zu hantieren. Würde es in der Firma des Entwicklers eine Vorgabe geben die besagt, dass man nur mit den öffentlichen Funktionen einer Klasse arbeiten soll, dann hätte diese Fehlerhandlung vermieden werden können.

Damit haben wir unseren ersten Begriff eingeführt. Wie geht es weiter? Durch die negative Belegung der Variablen nach Ausführung der Schleife befindet sich das System in einem **Fehlerzustand**. Ein Fehlerzustand ist dabei ein innerer Fehler der Software. Eine Person die den Code testet und den Fehlerzustand bemerkt wird nun einen Eintrag dazu in einem Fehlerverwaltungstool aufmachen. Dort muss die **Fehlerwirkung** eingetragen werden, eine Beschreibung des vorgefundenen Sachverhaltes, bspw:

> „Ich habe das Programm mit dem Wert ticketsToReserve = 100 gestartet und hatte nach Ausführung negative Werte in einigen Variablen"

Was ist denn aber genau ein Fehler? Ein Fehler ist die Nichterfüllung festgelegter Anforderungen, also eine Abweichung zwischen Soll- und Ist-Verhalten. Außerdem gibt es noch Mängel. Ein Mangel tritt dann auf, wenn gestellte Anforderungen oder berechtigte Erwartungen in Bezug auf beabsichtigten Gebrauch von Software nicht angemessen erfüllt werden. Handelt es sich bei unserem Beispiel tatsächlich um einen Fehler? Ja, denn mehr Tickets zu verkaufen als tatsächlich vorhanden sind kann bei einem realen Einsatz des Programms für große Probleme sorgen. Von einem Mangel könnte man sprechen, wenn man 20 Tickets gleichzeitig verkaufen möchte, aber durch einen Programmierfehler immer nur 5 Stück pro Durchlauf verkauft werden würden. Das würde die Ausführung des Programms einschränken, wenn letztendlich aber nicht die Anforderung „Tickets >= 0" verletzt werden würde, wäre es kein Fehler.

Wiederholen wir kurz die Vorzüge des Testens: Testen steigert die Qualität eines Softwareproduktes und senkt die Gesamtkosten für ein Projekt. Werden während des Testens Fehler aufgedeckt, dann können diese dokumentiert werden und die Mitarbeiter so daraus lernen, damit Fehlhandlungen verhindert werden können. Außerdem steigt bei positiven Testläufen, also Testläufen wo nur wenige oder gar keine Fehler auftreten, das Vertrauen in die Qualität eines Produktes. Testen ist während des Entwicklungsprozesses sehr kostenintensiv, insbesondere dann wenn kaum Testautomatisierung vorhanden ist, die Kosten für Bugfixing, nachdem ein Produkt auf den Markt gebracht wurde liegen aber noch deutlich über denen fürs Testen.

Schauen wir uns nun an, woraus so ein Testfall besteht:

- Eingabewert
- Soll-Ergebnis
- Vorbedingungen
- Nachbedingungen

Die Vor- und Nachbedingungen kennen wir bereits aus OCL. Das Soll-Ergebnis ist das erwartete Ergebnis. Wir wollen nun einen Pseudo-Unit-Test für unsere Funktion validateMoney() erstellen. In der Funktion wird geprüft, ob das vorhandene Guthaben größer oder gleich dem Preis für ein einzelnes Ticket ist. Wenn das gilt wird true zurück gegeben, ansonsten false. Ein möglicher Testfall:

- Eingabewert: /
- Soll-Ergebnis: false
- Vorbedingung: availableMoney = 0, pricePerTicket = 5
- Nachbedingung: availableMoney = 0, pricePerTicket = 5

Wenn kein Guthaben mehr da ist soll kein Ticket verkauft werden. Der Eingabewert für unseren Testfall ist leer, da die Funktion keinen Parameter erwartet. Noch ein Beispiel:

- Eingabewert: /
- Soll-Ergebnis: true
- Vorbedingung: availableMoney = 10, pricePerTicket = 5
- Nachbedingung: availableMoney = 10, pricePerTicket = 5

Wir haben in diesem Testfall ausreichend Guthaben um ein Ticket zu erwerben. Schauen wir uns die interessantere Funtkion buyTicket() an:

- Eingabewert: /
- Soll-Ergebnis: /
- Vorbedingung: availableMoney = 10, pricePerTicket = 5, ticketsLeft = 20
- Nachbedingung: availableMoney = 5, pricePerTicket = 5, ticketsLeft = 19

In diesem Beispiel haben wir ausreichend Guthaben und es sind ebenfalls noch Tickets vorhanden. Folgende Wertekombination funktioniert nicht:

- Eingabewert: /
- Soll-Ergebnis:
- Vorbedingung: availableMoney = 250, pricePerTicket = 5, ticketsLeft = 0
- Nachbedingung: availableMoney = 250, pricePerTicket = 5, ticketsLeft = 0

Wir haben hier zwar ausreichend Guthaben, aber es sind keine Tickets mehr zum Verkauf da.

Wir sehen an unserem Beispiel: obwohl es sehr überschaubar ist, benötigen wir eine große Zahl an Testfällen, um alle möglichen interessanten Wertkombinationen abzudecken (insbesondere in Grenzbereichen). Diese Werte können mit Hilfe eines Testorakels bestimmt werden. Was ist das?

Man kann entweder die Soll-Daten aus der Spezifikation des Testobjektes bestimmen. Man schaut sich dann an, was das Testobjekt modelliert, in unserem Beispiel bspw. die Anzahl der übrigen Tickets. Sicherlich macht hier nur ein Wertebereich von 0 bis n Sinn. Negative

Ticketanzahlen wollen wir nicht haben.

Eine andere Möglichkeit ist die Werte aus Prototypen zu ermitteln: dann benötigt man aber eine genaue formale Spezifikation des Testobjektes. Aus dieser werden dann mittels Werkzeugen Testfälle extrahiert.

Die dritte Möglichkeit ist, die gleiche Komponente mehrfach entwickeln zu lassen und mit den gleichen Testdaten zu füttern. Tritt bei einer Komponenten ein Fehlerzustand auf, dann kann man hieraus ebenfalls Testfälle ermitteln. Dieses Vorgehen ist allerdings sehr aufwändig und kostenintensiv und wird bei nicht-kritischer Softwareentwicklung kaum eingesetzt.

Es kann natürlich auch sein, dass Testfälle fehlerhaft formuliert wurden und das Programm eigentlich korrekt arbeitet. Man muss deswegen im Fehlerfall sowohl das Programm als auch den Testfall untersuchen um die genaue Ursache zu ermitteln.

Es gibt bei Testfällen grob 2 Arten von Tests: Positiv- und Negativtests. Positivtests gehen davon aus, dass das Programm korrekt funktioniert. Die Testfälle testen damit auf erwartete Eingaben. Positivtests werden oft von Entwicklern formuliert, da diese davon ausgehen, dass das, was sie programmiert haben, korrekt ist.

Negativtests hingegen testen auf erwartete Fehleingaben. Sie gehen also davon aus, dass das Programm nicht korrekt arbeitet. Negativtests werden oft von QS-Mitarbeitern durchgeführt. Eine Sonderform der Negativtests sind so genannte Robustheitstests: hier wird geprüft wie das Programm auf Eingaben reagiert, die sehr speziell sind und für die u. U. keine Ausnahmebehandlung spezifiziert wurde. Beispielsweise ist es bei Funktionen die Integerwerte akzeptieren sehr beliebt als Eingabe einen Wert MAX_INT einzugeben, die größte auf einem System darstellbare Integerzahl.

Aktivitäten des Testprozesses: der Testprozess besteht aus verschiedenen Stufen. Zu Beginn steht die Testplanung und Steuerung: wie sollen die Tests ausgeführt werden, was soll getestet werden?

Stufe zwei ist die Testanalyse und der Testentwurf: hier werden konkrete Testfälle ausgehend von der Testplanung spezifiziert.

In Stufe drei findet die tatäschliche Testrealisierung/-Durchführung statt.

Stufe vier wertet die Ergebnisse aus Stufe drei aus und bereitet sie auf: an dieser Stelle weiß der Tester dann, welche Tests erfolgreich waren und welche fehlgeschlagen sind und warum.

Letztendlich kommt der Abschluss der Testaktivitäten: war alles gut, dann ist der Testprozess beendet. Sind Fehler aufgetreten, dann müssen diese behoben und der Prozess von neuem durchlaufen werden.

Es gibt für jede Stufe des V-Modells einen entsprechenden Test. Für die Anforderungsdefinition ist das der Abnahmetests. Dieser ist nicht automatisiert und wird meistens durch den Kunden durchgeführt. Dem technischen Systementwurf entspricht der Integrationstests. Dieser lässt sich möglicherweise automatisiert durchführen. Der Komponentenspezifikation entspricht der Komponententest. Dabei handelt es sich um die Basistests, d. h. UnitTests. Diese testen nur einzelne, kleine Komponenten. Der Automatisierungsgrad ist sehr hoch.

In den nächsten Kapiteln wollen wir uns mit verschiedenen Testarten beschäftigen, unter anderem White- und Blackboxtests und Softwaremetriken. Letztere sind dabei keine dynamischen Tests sondern fallen unter den Bereich der Quellcodeanalyse.

KAPITEL 2.2: SOFTWAREMETRIKEN

Im letzten Kapitel:

- Grundlagen der Softwareverifikation: verifizieren zur Qualitätssteigerung
- Senkt Kosten und steigert Produktivität
- Anfangs hohe Investitionen, die sich aber zum Ende der Entwicklung rentieren

Softwaremetriken

Nachdem wir uns im letzten Kapitel ausführlich mit dynamischen Testverfahren beschäftigt haben werfen wir nun einen Blick auf den statischen Teil. Wir haben am Ende von Kapitel 3.1 bereits gesehen, dass es Verfahren gibt um Anomalien im Kontrollfluss eines Programms aufzudecken. Das Datenfluss-Testen welches ungenutzte Variablen und unwirksame Wertzuweisungen aufdeckt gehört ebenfalls zu den statischen Verfahren.

Ein weiteres Teilthema der statischen Testverfahren sind so genannte Softwaremetriken. Dabei handelt es sich nicht um „echte“ Tests mit denen man konkrete Fehler finden kann, sondern Softwaremetriken sagen etwas über die allgemeine Struktur und Komplexität eines Programms aus. Ein wohl bekanntes Beispiel ist die Zählung der in einem Programm enthaltenene Lines of Code (LOC). Es leuchtet ein, dass die LOC für komplexere Programme meistens größer ist als für einfachere.

Einwand: was ist, wenn man viel kommentiert hat, der Produktivcode aber eigentlich sehr kurz ist? Dann kann als Alternative zu LOC NCSS genommen werden. NCSS steht für Non-commented source statements, berücksichtigt damit nur genau die Zeilen, welche mindestens eine Anweisung enthalten. LOC und NCSS helfen nur dabei, die Komplexität eines Programms einzuschätzen. Es ist denkbar, dass ein schlechter Entwickler für ein verhältnismäßig einfaches Programm 10.00 LOCs benötigt, ein guter Entwickler nur 1.000 oder weniger. Die Komplexität eines Programms lässt sich aber ohnehin nicht fix feststellen. Es kommt immer darauf an, unter welchen Gesichtspunkten Software bewertet werden soll.

Wozu überhaupt Softwaremetriken? Wir haben im letzten Kapitel gesehen, dass dynamische Testverfahren sehr aufwändig sein können und die Ausbeute oftmals wenig zufrieden stellend ist. Wenn schon das teilweise Testen eines Programms so viel Arbeit macht ist ein komplettes Austesten höchstwahrscheinlich unmöglich. Man möchte deshalb wissen, für welche Komponenten einer Software sich das testen lohnt. Das geht zum einen durch Einschätzung seitens der Entwickler. Handelt es sich bei dem entwickelten Produkt beispielsweise um eine Synchronisationssoftware zwischen Client und Server mit ein wenig Ausgabe drumherum, dann werden die meisten Entwickler die Kommunikation zwischen Client und Server als die fehleranfälligste Komponente einstufen. Zum anderen kann mit Softwaremetriken entschieden werden, für welche Komponenten das Testen intensiviert werden sollte. Wie macht man das?

Naheliegend ist es, Softwaremetriken für das gesamte Projekt berechnen zu lassen, und die Werte für einzelne Komponenten zu vergleichen. Hohe bzw. anomale Werte können dann ein Hinweis darauf sein, dass an entsprechender Stelle ein Problem besteht bzw. entstehen kann. Als Beispiel? In einem Softwareprojekt wird für jeden Build für alle Methoden von Klassen der Fan-in und Fan-out berechnet. Fan-in/-out bezeichnen dabei die Anzahl der Methoden, die die Zielfunktion aufrufen, bzw. von ihr aufgerufen werden. Mit jedem Build steigt nun sowohl der Fan-in als auch der Fan-out einer Methode calculateALot() in einer Klasse. Diese Methode hat außerdem eine Signatur mit 7 Parametern. Spätestens jetzt sollten bei jedem Entwickler die Alarmglocken schrillen, denn was man hier hat ist eine übergroße Funktion mit viel zu viel Komplexität. Das Prinzip KISS (keep it simple, stupid) ist hier verletzt. Bevor die Situation sich weiter verschärft

sollte die in der Methode enthaltene Funktionalität jetzt in mehrere Methoden und Klassen aufgesplittet werden. Das senkt die Komplexität der einzelnen Methoden und macht den Code um ein vielfaches lesbarer. Außerdem sinkt die Gefahr, dass bei einer Änderung an der Methode andere Programmteile nicht mehr korrekt arbeiten.

In diesem Beispiel haben Softwaremetriken geholfen Handlungsbedarf zu erkennen bevor das Problem zu komplex wurde. Man stelle sich jetzt vor man hätte weiter gemacht wie bisher. Irgendwann, das Programm wurde bereits auf den Markt gebracht, tritt ein schwerwiegender Fehler in der Methode calculateALot() zu Tage. Jetzt die Situation zu analysieren und einen korrekt arbeitenden Fix herauszubringen verursacht immense Kosten in Entwicklung, Qualitätssicherung und Wartung. Betrachten wir nun ein weiteres Beispiel für Softwaremetriken.

Zyklomatische Komplexität

Die Softwaremetrik der zyklomatischen Komplexität berechnen die zyklomatische Zahl. Diese gibt an, wie viele linear unabhängige Pfade eines zyklischen Graphen, eines Kontrollflussgraphen, es gibt. D. h. die zyklomatische Zahl gibt an, wie viele verschiedene Möglichkeiten es gibt, durch ein Programm durchzulaufen. Damit ist die Zahl gleichzeitig obere Schranke für die Anzahl der Testfälle die erstellt werden müssen, um eine 100% Zweigüberdeckung zu erreichen. Ein Beispiel:

```
public int getNumber()
{
	return number;
}
```

Wie viele Möglichkeiten gibt es, durch diese Funktion zu laufen? Exakt eine. Noch ein Beispiel:

```
public int getNumber2() {
	if(number % 2 == 0) {
		switch(number) {
			case < 0:
			return 0;
			break;

			case = 0:
			return 1;
			break;

			case > 0 && < 10:
			[…]
		}
	}
	else {
		for(int i = 0; i < number; ++i) {
			checkForValidIndex(i);
			[…]
		}
	}
}
```

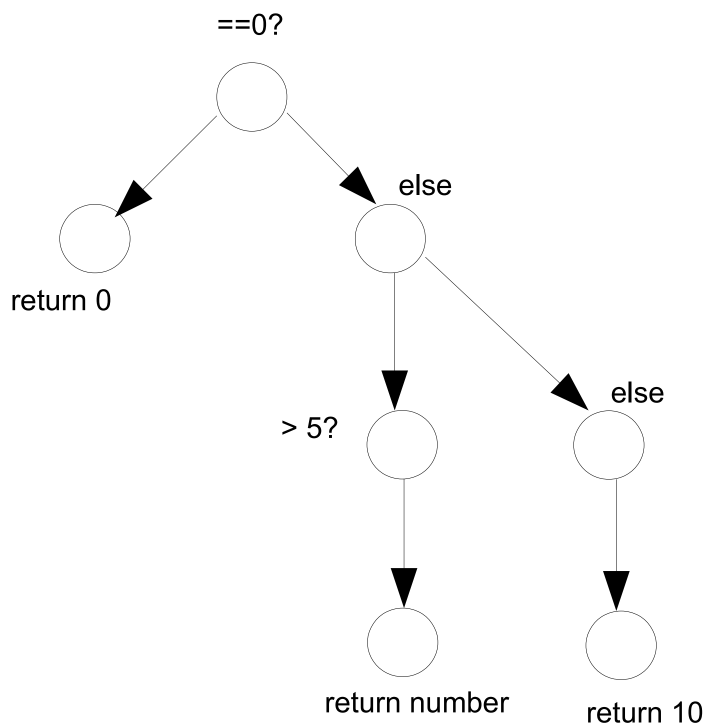
Was die Funktion genau tut ist uninteressant. Zu sehen ist jedenfalls, dass es auf den ersten und auf den zweiten Blick nicht klar wird, wie viele Pfade es durch die Funktion gibt. Für solche Beispiele, in denen Menschen an die Grenzen ihrer Vorstellungskraft stoßen, gibt die zyklomatische Zahl Hilfestellung. Die zyklomatische Komplexität ist dabei definiert als:

$v(g) = e - v + 2$ mit e = Anzahl der Kanten, v = Anzahl der Knoten

Ein einfaches Beispiel:

```
public int getNumber3()
{
	if(number == 0)
	{
		return 0;
	}
	else
	{
		if(number > 5)
			return 10;
		else
			return number;
	}
}
```

Als Baum:



Wir zählen: 6 Kanten, 7 Knoten. Dann gilt:

$v(G) = 6 - 7 + 2 = 1$

Was sagt uns dieses Ergebnis jetzt? Nach McCabe, dem Erfinder dieser Softwaremetrik ist eine zyklomatische Komplexität größer als 10 nicht akzeptabel, außer in einigen gut begründeten Ausnahmefällen. Ein Messwert im Bereich von 6 ist nach McCabe akzeptabel. Wenn wir also eine Komplexität von 1 haben können wir davon ausgehen, dass unsere Funktion überschaubar und verständlich ist (was sie wirklich ist).

Bei der zyklischen Komplexität gibt es das Problem der Validität: gegeben seien zwei Programmstücke für das gleiche Problem. Stück 1 arbeitet mit goTo-Befehlen, doppeltem und dreifachem Code (ohne dass dieser in eine Funktion ausgelagert wird) und verletzt auch sonst allen guten Programmierstil. Stück 2 ist wohl strukturiert und ohne Probleme lesbar und verständlich. Die zyklomatische Komplexität liefert für beide Stücke möglicherweise aber die gleiche Komplexität. Obwohl Stück 2 einfacher zu verstehen ist lässt die zyklomatische Komplexität diese Unterscheidung hier nicht zu. Es wir daher kritisiert, dass die Metrik keine viel bessere Aussagekraft als NCSS hat.

Metriken-Suite CK

Die Metriken-Suite CK bietet einige weitere Kriterien für die „Vermessung" von Quellcode. Folgende Metriken sind enthalten:

- Gewichtete Methoden pro Klasse: für jede Methode einer Klasse wird eine Gewichtung vorgenommen; einfache Methoden können Komplexität 1 haben (siehe unsere Funktion), komplexere und größere Methoden deutlich mehr
- Tiefe des Vererbungsbaums: je mehr Vererbung es in einem Softwareprojekt gibt, desto komplexer ist es; das begründet sich einfach darin, dass man zuerst eine lange Kette von Beziehungen zwischen Klassen nachvollziehen muss bevor man eine Blattklasse, d. h. eine Klasse die keine Kindklassen hat die von ihr erben, vollständig verstehen kann
- Zahl der Kinder: gibt an, wie viele Klassen von einer Klasse erben; ein hoher Wert ist ein Zeichen für Wiederverwendbarkeit, denn es deutet darauf hin, dass der Code in der Superklasse flexibel und gut durchdacht ist
- Kopplung von Klassen: Zwei Klassen A und B sind gekoppelt, wenn A viele Methoden oder Variablen aus B benutzt; eine hohe Kopplung ist ein Indiz dafür, dass eine Änderung einer Klasse viele andere Klassen betrifft; Kopplung ist zwar nicht ganz zu vermeiden, sollte aber so gering wie möglich gehalten werden; bei zu viel Kopplung treten des öfteren auch zyklische Abhängigkeiten auf (sehr dankbare Aufgabe, so etwas aufzulösen!)
- Reaktion einer Klasse: Die Reaktion einer Klasse gibt an, wie viele Methoden in Frage kommen um auf eine Nachricht einer umgebenden Klasse zu reagieren. Hoher Reaktionswert ist ein Indiz dafür, dass eine Klasse zu komplex ist. Als Beispiel: Eine Klasse A erhält die Nachricht „liefer mir die Nummer". Folgende Methoden kommen dazu in Frage:

  ```
  Int getNumber();
  Float getNumberPrecise();
  Double getNumberLong();
  Long getNumberBig();
  …
  ```

  Eine hohe Reaktion muss nicht sofort Indiz für eine zu hohe Komplexität sein, kann es aber.

- Mangel an Zusammenhalt in Methoden: bestimmt die Differenz zwischen den Methodenpaaren, die sich Attribute teilen und denen, die es nicht tun. Diese Metrik liefert unter Umständen kein brauchbares Ergebnis, denn was ist mit einer Klasse, die 10 Attribute und 10 Getter- und Settermethoden besitzt? Dann ist der LCOM-Wert sehr hoch, tatsächlich ist ein größerer Zusammenhalt der Methoden hier aber nicht sinnvoll

Ein weiteres Problem von Softwaremetriken ist, dass Entwickler versuchen könnten Code so zu schreiben, dass die Auswertung der Metriken „gnädiger" ausfällt, dadurch aber nichts an Qualitätszuwachs gewonnen ist. Meistens sinkt durch solche „Optimierungen" sogar noch die Lesbarkeit von Code.

KAPITEL 2.3: BLACK-BOX-TEST

Im letzten Kapitel:

- Softwaremetriken: Motivation (unterstützen bei der Fehlerfindung, weisen auf problematische Stellen hin)
- Einfache Metriken: LOC, NCSS, zyklomatische Zahl (alle gleich mächtig
- CK-Suite: Auswahl an weiteren Metriken, bewertet Eigenschaften von Klassen und Methoden

Black-Box-Tests

Bisher haben wir zum Erstellen von Testfällen die innere Struktur einer Komponente berücksichtigt, d. h. ihren Quellcode für Unit Tests, Anweisungs-, Zweig- und Pfadüberdeckung oder die Modellierung einer Komponente beim Einsatz von OCL. Bis auf OCL sind alle diese Verfahren aus „Entwicklersicht". D. h. ein Entwickler schreibt für (seinen meist eigenen) Code Testfälle. Meistens läuft das dann auf die so genannten Positivtests hinaus, d. h. der Entwickler will zeigen, dass sein Code auch tatsächlich das tut was er soll. Wir haben gesehen, dass es auch sinnvoll sein kann, Negativ- bzw. Robustheitstests zu formulieren. Ein QS-Mitarbeiter, dem so eine Aufgabe zufallen könnte, weiß in der Regel nichts über den Programmcode. Er kennt lediglich die Anforderungen an eine Komponente. Damit wären wir bei der Ausgangssituation für Black-Box-Tests.

Stellen wir uns vor wir sollen ein Auto testen. Wir wissen nichts über die intern verwendete Software, die mechanischen Eigenschaften der Schaltung oder der Lenkung und wir haben auch keinerlei Kenntnis von den Verbrennungsvorgängen im Motor. Glücklicherweise haben wir, bevor wir die Konstruktion des Autos in Auftrag gegeben haben, eine Liste von Anforderungen an das System erstellt. Unter anderem soll unser Auto vorwärts fahren können. Aber wie soll es das tun? Neben unserer Anforderungsliste haben wir eine **funktionale Spezifikation** des Autos. Eine funktionale Spezifikation im Bereich des Black-Box-Testens ist vergleichbar mit den in Schritt 4 einer algebraischen Spezifikation angegebenen Beschreibungen der Auswirkungen einer Komponente auf ein System. Beispielsweise findet sich folgende Stelle in unserer Spezifikation:

> „Komponente Gaspedal: wird das Gaspedal betätigt, es ist ein Gang eingelegt, der Motor ist gezündet und die Handbremse gelöst, dann bewegt sich das Auto nach vorne"

Modellieren wir nun eine boolesche Funktion *canAccelerate()*, die *true* zurückgibt, falls obige Bedingung wahr ist:

```
public bool canAccelerate()
{
	if(isGasPressed() &&
		isGearSet() &&
		isMotorIgnited() &&
		!isTightenedHandbrake())
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

Die genaue interne Logik der benutzten Funktionen interessiert uns dabei nicht. Wichtig ist, dass sie alle boolesche Werte zurückgeben. Mit unserer Modellierung können wir jetzt Testfälle für die Beschleunigung des Autos erstellen. Wir nutzen dazu den Äquivalenzklassenmechanismus.

Äquivalenzklasse

Eine Äquivalenzklasse kann man sich als Menge von Werten vorstellen. Alle Werte einer Menge teilen dabei gewisse Eigenschaften. Beispielsweise gibt es für natürliche Zahlen die Äquivalenzklasse der geraden und der ungeraden Zahlen. Für ganze Zahlen gibt es neben diesen beiden Klassen auch noch eine für positive und eine für negative Zahlen. Es müssen dabei nicht immer genau zwei Klassen sein. Ein anderes Beispiel: Werte für die Modulo-Rechnung mit 3:

- x mod 3 = 0
- x mod 3 = 1
- x mod 3 = 2

In diesem Beispiel also 3 Äquivalenzklassen.

Es gibt dabei mehrere Möglichkeiten, Äquivalenzklassen zu spezifizieren. Wie man sie spezifiziert kommt dabei auf die gegebene Einschränkung an, zum Beispiel:

- Beschränkung spezifiziert Wertebereich: bei einem Buffet in einem Restaurant darf jeder Gast zwischen 0 und 10 mal seinen Teller neu füllen:

  Wertebereich der Eingabe: 0 <= x <= 10
  Gültige Äquivalenzklasse: 0 <= x <= 10
  Ungültige Äquivalenzklasse: x > 10

- Beschränkung spezifiziert minimale und maximale Anzahl von Werten: bei einem Benefizlauf darf jeder Teilnehmer bis zu 50 Runden laufen. Minimalanforderung sind 10 Runden:

  Gültige Äquivalenzklasse: 10 <= x <= 50
  Ungültige Äquivalenzklassen: x < 10 und x > 50

- Beschränkung spezifiziert Menge von Werten, die unterschiedlich behandelt werden: bei einem Bücherverkauf sind Bücher aus den Genres „Fiktion“, „Geschichte“ und „Kochbuch“ reduziert:

  Gültige Äquivalenzklasse: Fiktion, Geschichte, Kochbuch
  Ungültige Äquivalenzklasse: ein beliebiges Element e, das nicht Element der Menge der reduzierten Genres ist, bspw. „Roman“ oder „Sprachkurs“

- Beschränkung spezifiziert Situation, die erfüllt sein muss: für die Zeitmessung beim Dortmunder Campuslauf benötigt jeder Teilnehmer einen Messchip, der in die Schnürsenkel gebunden werden muss:

  Gültige Äquivalenzklasse: Chip ist in Schnürsenkel gebunden
  Ungültige Äquivalenzklasse: Chip ist nicht in Schnürsenkel gebunden; eventuell ist er woanders befestigt oder wurde ganz vergessen

Kommen wir nun auf unser Beispiel mit der Beschleunigung zurück. Wenn wir obige Varianten von Beschränkungen betrachten sehen wir, dass sich Beschränkungen, die Situationen spezifizieren am besten zur Modellierung für unser Problem eignen. Diese Beschränkungen lassen sich nämlich in genau zwei Äquivalenzklassen aufteilen: Beschränkung eingehalten oder nicht eingehalten. Das entspricht in der booleschen Logik einem Wert von *true* bzw, *false*.

Stellen wir die Testfälle für unser Beispiel tabellarisch dar:

	isGasPressed	isGearSet	isMotorIgnited	isTightenedHandbrake
gÄK1	true	true	true	false
uÄK1	false	true	true	false
uÄK2	false	false	true	false
uÄK3	false	false	false	false
uÄK4	false	false	false	true

Die Belegung *(true, true, true, false)* entspricht hierbei der einzig gültigen Äquivalenzklasse. Die ersten drei Bedingungen müssen erfüllt sein, die letzte darf nicht stimmen (Handbremse angezogen). Dann kann das Auto vorwärts fahren.

Zwar ist dieses Beispiel sehr klein, dennoch ist die Tabelle verhältnismäßig groß. Die Anzahl der Testfälle kann jedoch minimiert werden: bspw. macht es keinen Sinn Repräsentanten undgültiger Äquivalenzklassen miteinander zu kombinieren. Das Gesamtergebnis wäre ebenfalls ungültig. Ebenfalls können alle Repräsentanten einer ÄK mit jedem Repräsentanten einer anderen AK in einem Testfall kombiniert werden, d. h. man kombiniert die Testfälle paarweise und nicht über die gesamte Tabelle. So kann man schnell sehen wann man ungültige Ergebnisse bekommt und braucht an der Stelle nicht mehr weiter testen.

Durch die Kombination von Repräsentanten aus ungültigen ÄKs kann es außerdem zur so genannten Fehlerüberdeckung kommen. Das bedeutet, dass man bei einem Auftreten mehrerer Fehler eventuell nur einen sieht. Als Beispiel:

velocity > 50; gear IN {g4, g5, g6}

Äquivalenzklassen:

velocity_gÄK1: velocity > 50
velocity_uÄK1: velocity <= 50
gear_gÄK1: gear IN (g4, g5, g6)
gear_uÄK1: NOT gear in (g4, g5, g6)

Testdaten:

velocity_uÄK1 und gear_uÄK1: velocity = 49, gear = backwards

Dass hier ein nicht erlaubter Gang verwendet wird fällt möglicherweise gar nicht auf, da der erste Fehler beim Geschwindigkeitswert auftritt.

Wir konnte mit Hilfe von Äquivalenzklassen relativ schnell geeignete Testfälle für die Autobeschleunigung erstellen. Leider berücksichtigen Äquivalenzklassen Beziehungen und Wechselwirkungen von Bedingungen kaum. Beispielsweise wurden die Vorbedingungen für die Beschleunigung (Handbremse locker, Motor gezündet, Pedal gedrückt, Gang eingelegt) zwar realisiert, aber wenig gewichtet. Wir werden später in diesem Kapitel noch geeignetere Verfahren finden, um so etwas zu modellieren.

Grenzwertanalyse

Erinnern wir uns kurz an unsere Methode zur Berechnung von Quadratzahlen aus einem früheren Kapitel zurück. Die Methode gab anstatt dem quadrierten Wert jedoch einfach nur den Betrag der Eingabe zurück. Für die Werte -1, 0 und 1 sah es dann danach aus, als würde die Funktion korrekt arbeiten. Bei einem anderen Wert allerdings versagte die Funktion. Was wir uns jetzt ansehen wollen geht ein wenig in diese Richtung: die **Grenzwertanalyse**

Die Idee hinter der Grenzwertanalyse ist, sich solche Werte anzugucken wo das Verhalten einer Komponente besonders fehleranfällig ist. Als Beispiel: das Unternehmen nextbike (Metropolrad Ruhr) bietet die Ausleihe von Fahrrädern an. Voraussetzung ist, dass man ein Konto bei der Firma hat. Auf diesem Konto muss immer ein gewisses Guthaben vorhanden sein um Räder ausleihen zu können. Ist das Guthaben größer als 0, dann kann man ein Fahrrad leihen. Es ist vorstellbar, dass der Entwickler der nextbike-Software anstelle eines Vergleiches auf größer („>") aus Versehen auf größer gleich („>=") prüft. Dann könnte man mit einem Guthaben von 0€ ebenfalls Räder leihen, was nicht im Sinne der Firma ist. Eine Grenzwertanalyse würde in diesem Fall Klarheit schaffen. Wie also sieht so eine Analyse aus?

Zur Grenzwertanalyse zieht man zuvor spezifizierte Äquivalenzklassen heran. Dann nimmt man sich den größten bzw. kleinsten Wert der Klasse und wählt ihn als Eingabe für eine Komponente. Ebenfalls wählt man einen „ein bisschen größeren" und einen „ein bisschen kleineren" Wert zur Überprüfung. Betrachten wir ein Beispiel für unser nextbike-Beispiel. Das Guthaben eines Kontos wird dabei immer in 1€ Schritten verbraucht, als Datentyp zur Modellierung eignet sich also Integer:

Gültige Äquivalenzklasse nextbike: Guthaben > 0
Ungültige Äquivalenzklasse nextbike: Guthaben <= 0

Grenze	Größer	Kleiner
1	2	0

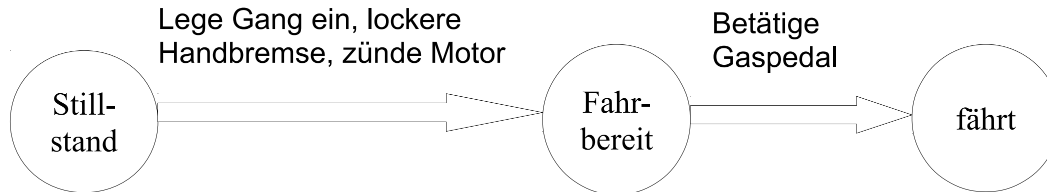
Nun können wir mit den ermittelten Grenzwerten unsere ÄK testen. Falls die Spezifikation tatsächlich aus Versehen größer mit größer gleich verwechselt hat, dann fällt uns das an dieser Stelle auf. Wir könnten ebenfalls mit negativen Werte arbeiten, das macht für dieses Beispiel aber keinen Sinn. Denkbar wäre es bspw. für die Modellierung eines Überziehmodells für Konten. Diese können bis zu einem gewissen Betrag (-x) überzogen werden. Testwerte für die Grenzwertanalyse wäre dann bspw:

- -x + 1
- -x
- -x – 1

Wir sehen, eine Grenzwertanalyse deckt Fehler möglicherweise besser auf als eine normale Verwendung von Äquivalenzklassen. Denn wenn ein Repräsentant x Element einer Äquivalenzklasse mit 1.000 Elementen ist, dann höchstwahrscheinlich auch alle Elemente im Bereich x – 400 und x + 400. Betrachten wir allerdings die Grenzbereiche, dann steigt die Wahrscheinlichkeit, Fehler zu entdecken. Dennoch ist auch die Grenzwertanalyse für abstraktere Testfälle nicht effizient genug. Was ist wenn wir abstrakte Datentypen haben? Beispielsweise einen Datentyp für einen *CONNECTION_STATE*. Bis auf *uninitialized* und *initialized* fällt einem da so schnell kein geeigneter Wert für eine Grenzwertanalyse ein. Also benötigen wir für so etwas andere Verfahren.

Zustandsbasierte Tests

Schauen wir uns nochmal unser Autobeispiel an. Wann kann es losfahren? Unser Auto kann losfahren, wenn die Handbremse gelockert, ein Gang eingelegt und der Motor gezündet ist. Dann muss nur noch das Gaspedal betätigt werden. Wir können das als Automaten modellieren:



Der Automat befindet sich, je nachdem was zuvor passiert ist, immer in einem definierten Zustand. In unserem Beispiel sind natürlich nicht alle Übergänge abgebildet. Die gleiche Idee steckt hinter dem zustandsbasierten Testen: ausgehend von einem Zustand werden gewisse Eingaben/Aktionen getätigt. Befindet sich der Automat am Testende im erwarteten Zustand, dann ist der Test erfolgreich verlaufen. Modellieren wir unsere Mensaklasse aus Kapitel 1:

Klasse Mensa

Zustandserhaltende Operationen
foodCount() : integer; // Anzahl vorhandener Gerichte
MAX() : integer; // maximale Gerichtanzahl (direkt nach Öffnung)
Zustandsverändernde Operationen
Mensa(Max: integer); // Konstruktor
~Mensa(); // Destruktor
sellFood(): void; // verkauft ein Gericht

Damit ergeben sich für unseren Automaten drei Zustände:

empty: foodcount() = 0, letztes Gericht verkauft
filled: 0 < foodCount() < MAX();
full: foodCount() = MAX;

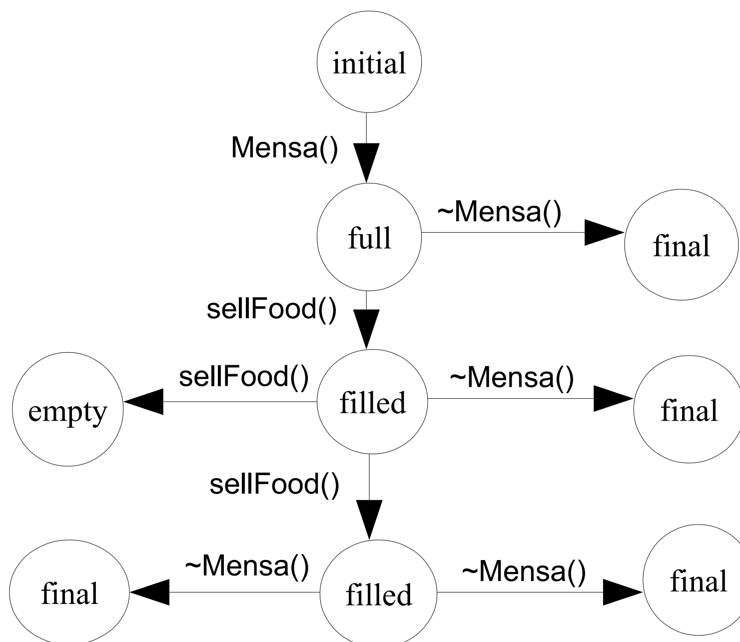
Es gibt zwei Testarten: Konformanz- und Robustheitstests. Konformanztests entsprechen dabei in etwa Positivtests, d. h. man versucht die Konformität eines Testobjektes zu zeigen. Robustheitstests arbeiten mit einer nicht-konformanten Benutzung, bspw. eine Belegung des Wertes *max* im Konstruktor mit einer negativen Zahl.

Was ist nun zu tun? Als erstes muss ein Zustandsdiagramm angelegt werden. Ich gebe direkt die Zustandsübergangstabelle an. Diese stellt alle möglichen (und unmöglichen) Übergänge dar und belegt sie mit einem Wert:

Zustand/Ereignis	initial	empty	filled	full
Mensa()	MAX	N/A	N/A	N/A
~Mensa()	N/A	final	final	final
sellFood()	N/A	N/A	empty, filled	filled

Die beiden Zustände initial und final sind Hilfszustände. Sie definieren einfach Beginn und Ende des Automaten, ähnlich wie in Aktivitätsdiagrammen. Betrachten wir die Tabelle genauer: wenn die Mensa geöffnet wird (Ereignis *Mensa()*), dann ist die Anzahl verfügbarer Gerichte maximal (= *MAX()*). Haben wir nicht mehr die maximale Anzahl (Zustand *filled*), dann kommen wir durch Verkauf eines weiteren Gerichtes entweder in den Zustand *empty* oder bleiben in *filled* (wenn Restanzahl > 0).

Als nächstes benötigen wir einen Übergangsbaum für unseren Tests. Der Übergangsbaum ist im Prinzip das gleiche wie der Automat und die Tabelle, nur dass man in ihm direkt mögliche Programmpfade ablesen kann:



Im nächsten Schritt muss der Übergangsbaum für den Zustands-Robustheitstest angepasst werden. Dazu müssen Fehlerzustände eingeführt werden für alle Botschaften (Ereignisse), für die in einem Knoten keine Übergänge spezifiziert sind. Beispielsweise ist für den Zustand *empty()* kein Übergang definiert, wenn als Ereignis *sellFood()* auftritt. Das macht auch keinen Sinn, also muss hier ein Fehlerzustand hin. Aus Platzgründen werde ich den erweiterten Übergangsbaum nicht darstellen, es sollte aber klar sein aus welchen Knoten man Fehlerzustände erreichen kann.

Was sind jetzt die Testfälle? Diese lassen sich in unserem Übergangsbaum einfach ablesen. Ein Pfad von der Wurzel bis zu einem Blatt beschreibt eine Funktionssequenz. Ein einfaches Beispiel:

(*) <initial> Mensa(2000) <full> ~Mensa() <final>

Bildet genau den Pfad im Übergangsbaum von der Wurzel zu ihrem Kindknoten “full” zum Kindknoten “final” ab. Allgemein benötigt man folgendes für die Generierung eines Testfalls:

- Anfangszustand des Testobjektes
  hier: initial
- Eingaben für das Testobjekt
  hier: eine der zustandsverändernden Operationen
- Erwartete Ausgaben bzw. das erwartete Verhalten
  bspw: wurde die Gerichtmenge nach Aufruf von sellFood() wirklich um eine Einheit

reduziert?

- Erwarteter Endzustand
  also: in welchem Zustand erwarte ich den Automaten nach Ausführung des Testfalls? Befindet er sich nach Beispiel (*) tatsächlich im Zustand final?

Für jeden **Zustandsübergang**, d. h. für jeden Wechsel von Zustand A in Zustand B während der Ausführung eines Testfalls muss folgendes gelten:

- Zustand vor dem Übergang:
  also: in welchem Zustand bin ich, bevor ich ein Ereignis auslöse?
- Auslösendes Ereignis, das den Übergang bewirkt
  bspw: Mensa(), ~Mensa(), sellFood()
- Erwartete Reaktion, ausgelöst durch den Übergang
  also: was bewirkt das Ereignis? Beispielsweise: die Anzahl der Gerichte wird um eine Einheit verringert
- Nächster erwarteter Zustand
  also: in welchem Zustand lande ich nach dem das Ereignis aufgetreten ist?

Schauen wir uns noch einen Beispieltestfall an:

<initial> new Mensa(2000) <full> sellFood() <filled> sellFood() <filled> […] sellFood() <empty> ~Mensa() <final>

Dieser Testfall beschreibt den normalen Mensaalltag: es werden so lange Gerichte verkauft, bis keine mehr da sind (Zustand *empty* erreicht) und die Mensa macht zu. Bezogen auf die für einen Testfall benötigten Komponente ergibt sich folgendes:

- Anfangszustand: initial
- Eingaben für das Testobjekt: Mensa(), ~Mensa(), sellFood()
- Erwartete Ausgaben bzw. Verhalten: foodCount = MAX, foodCount = N/A, foodCount = foodCount – 1, foodCount = 0
- Erwarteter Endzustand: final

Für die einzelnen Zustandsübergänge ergibt sich bspw. folgendes:

<u>Von initial nach full:</u>

- Zustand vor dem Übergang: initial
- Auslösendes Ereignis: new Mensa()
- Erwartete Reaktion: foodCount = MAX()
- Nächster erwarteter Zustand: full

<u>Von filled nach filled:</u>

- Zustand vor dem Übergang: filled
- Auslösendes Ereignis: sellFood()
- Erwartete Reaktion: foodCount = foodCount – 1
- Nächster erwarteter Zustand: filled

Von filled nach empty:

- Zustand vor dem Übergang: filled
- Auslösendes Ereignis: sellFood()
- Erwartete Reaktion: foodCount = 0
- Nächster erwarteter Zustand: empty

Wie bereits gesagt gibt es zwei Typen von Tests: Konformanz- und Robustheitstests. Für zweitere muss dabei nichts besonders beachtet werden. Für den ersten Testtyp ist es wichtig, dass in jedem Schritt die Wächterbedingungen der Zustände beachtet werden. Folgendes Beispiel ist ungültig:

<initial> new Mensa(2000) <empty>

Was ist daran falsch? Wir haben gesagt, dass der Zustand *full* bedeutet, dass *foodCount()* den Wert MAX liefert. Dieser ist auf jeden Fall größer als 0. *Empty* fordert aber, dass *foodCount() = 0* gilt. Damit ist die Bedingung verletzt, denn direkt nach der Mensaöffnung ist der *foodCount* niemals 0 (siehe Zustandsübergangstabelle).

Noch ein falsches Beispiel:

<initial> new Mensa(2000) <full> sellFood() <full>

Wir erstellen ein Mensaobjekt mit 2000 Gerichten. Der Automat befindet sich im Zustand full, da foodCount = MAX gilt. Wir verkaufen ein Gericht und landen wieder im Zustand full. Das kann aber nicht sein, da wir in sellFood() den foodCount dekrementieren. Also gilt

foodCountOld = foodCountNew + 1

Und damit

foodCountNew < MAX()

Also ist unser Testfall ungültig.

Bei Robustheitstests müssen wir die Wächterbedingungen nicht beachten. Ein fehlerhafter Testfall für Robustheitstest könnte so aussehen:

<initial> new Mensa(2000) <empty> <FEHLER>

Noch ein Beispiel:

<initial> new Mensa(2000) <full> sellFood() <filled> ~Mensa <final> sellFood() <FEHLER>

Nachdem das Mensaobjekt zerstört ist dürfen wir es nicht weiter verwenden. Der *sellFood()*-Aufruf am Ende ist daher illegal.

Prinzipiell können wir so viele Pfade des Übergangsbaum austesten wie gewünscht. Es ist allerdings klar, dass es keinen Sinn macht wirklich ALLE möglichen Variablenbelegungen auszutesten, sodass man nicht den kompletten möglichen Zustandsraum nutzen wird. Wir haben bereits in vorherigen Kapiteln gesehen, dass wir für ein vollständiges Austesten einer Funktion mit 3 Integerwerten über 90 Jahre benötigen.

Wie lassen sich unsere gefundenen Testfälle in Code gießen? Eine Möglichkeit, unsere Tests auszuführen ist mit Hilfe von Testskripten. Diese verwenden die zustandserhaltenden Funktionen

einer Klasse um den aktuellen Zustand zu prüfen. Beispiel:

```
// <initial>
Mensa m = new Mensa(2000);
// <full>
m.sellFood();
// <filled>
m.sellFood();
// <filled>
// Jetzt: Zustandsüberprüfung mittels foodCount; verändert den Zustand nicht
if(m.foodCount() == 1998) then „OK“ else throw LostFoodException;
```

Wie kann man zustandsbasierte Tests von ihrem Nutzen her einordnen? Sie eignen sich auf jeden Fall sehr gut zum Testen objektorientierter Systeme. Das haben wir an unserem Beispiel gesehen. Man kann jeden beliebigen Zustandsübergang testen. Dadurch ist eine komplette Überprüfung natürlich auch sehr aufwändig und vermutlich gar nicht effizient lösbar. Schauen wir uns nun ein weiteres Black-Box-Verfahren an: den Entscheidungstabellentests.

Entscheidungstabellentests

Entscheidungstabellentests sind vergleichbar mit Bedingungsüberdeckungen aus White-Box-Tests. Allerdings ist der Entscheidungstabellentest mächtiger, denn man kann ebenfalls Aktionen angeben, die im Erfolgs- bzw. Fehlerfall durchgeführt werden sollen. Wie sieht ein Entscheidungstabellentests also aus? Er besteht aus 4 Teilen:

Bedingungen	Regeln
Aktionen	Aktionszeiger

Die Bedingungen enthalten dabei die verschiedenen möglichen Bedingungen, bei unserem Autobeispiel also zum Beispiel „Handbremse ist fest“ oder „Handbremse gelockert“. Der Regelteil gibt an, welche Kombinationen von Wahrheitswerden berücksichtigt werden sollen. Sie werden dabei spaltenweise gelesen. Wir werden uns das in einem Beispiel ansehen. Die Aktionen sind eine textuelle Beschreibung der durchzuführenden Aktionen und der Aktionszeiger ist ein Indikator, ob eine Aktion abhängig von den Regeln ausgeführt werden soll.

Beispiel:

Handbremse gelockert	N	J	J	J	J	N	N
Gang eingelegt	N	N	J	J	N	N	J
Gaspedal betätigt	N	N	N	J	J	J	J
Bleibe stehen	X	X	X				
Fahre los				X			X
Motor aufheulen lassen					X	X	

Es sind dabei nicht alle Regeln erforderlich. Beispielsweise könnten wir die Regel (J, J, J) und (N, J, J) zusammenfassen, bzw. für „Handbremse gelockert ein „don’t care“ („-„) angeben: wenn sie nicht gelockert ist fahren wir eben etwas langsamer. Generell gilt, dass nur sinnvolle Kombinationen von Wahrheitswerten betrachtet werden müssen. Das ist der Vorteil von

Entscheidungstabellentests gegenüber Bedingungsüberdeckungen. Bei einer vollständigen Angabe aller Kombinationen spricht man auch von einer **vollständigen** Entscheidungstabelle. Die Tabelle heißt außerdem **redundanzfrei**, gdw. wenn verschiedene Bedingungen zu anderen Aktionen führen. Die ersten drei Spalten unserer Regeltabelle führen alle zur Aktion „bleibe stehen“. Die ET ist also nicht redundanzfrei. Schließlich gibt es noch den Begriff **widerspruchsfrei**. Das ist eine ET genau dann, wenn logische Beziehungen zu konsistenten Aktionen führen. In unserem Beispiel gibt es eine Beziehung zwischen Gang eingelegt und Gaspedal betätigt. Gäbe es einen Aktionsindikator der sagt, das Auto fährt wenn kein Gang eingelegt ist, aber das Gaspedal betätigt, dann wäre die ET nicht widerspruchsfrei.

Entscheidungstabellentests eignen sich gut um Zusammenhänge zwischen verschiedenen Bedingungen zu modellieren. Der Aufwand für das Erstellen einer ET ist außerdem „zum schnellen Testen“ überschaubar. Andererseits wird die Tabelle bei komplexeren Zusammenhängen sehr schnell sehr groß. Die Zusammenhänge zwischen einzelnen Bedingungen sind außerdem über die Regeln nur implizit ausdrückbar. Das verhält sich anders bei **Ursache-Wirkungs-Graphen**.

Ursache-Wirkungs-Graphen

Ursache-Wirkungsgraphen sind eine graphische Beschreibung von logischen Wirkzusammenhängen. Ein Graph wird aufgespannt durch:

- Ursachen:
  - Eingaben (Input von außen)
  - Dateiinhalte / Datenbanken (gespeicherter Input)
  - Initiale Systemzustände
- Wirkungen:
  - Ausgaben (aufgrund spezieller Eingaben)
  - Resultierende Systemzustände (ähnlich einem Automaten)
- Logische Verknüpfungen: logisches und, logisches oder und logisches nicht

Was ist die Motivation für UWGs? Beim Testen mit Grenzwertanalyse konnten wir immer nur einen Wert betrachten. Das reicht in der Praxis oftmals nicht, vielmehr kommt es auf die Kombination von Werten und deren Wechselwirkungen an.

Wie erstellt man einen UWG? Zuerst muss eine vorliegende Spezifikation in bearbeitbare Teile zerlegt werden, d. h. so dass sie sich mit einem UWG darstellen lassen. Danach müssen Ursachen und Wirkungen identifiziert werde. Ursachen lassen sich daran erkennen, dass sich ihnen boolesche Wahrheitswerte zuordnen lassen. Wirkungen sind in den meisten Fällen Ausgaben oder Systemtransformationen (Zustandswechsel). Wenn die Identifizierung abgeschlossen ist können Ursachen und Wirkungen mittels eines UWG graphisch dargestellt werden. Ein Beispiel: ein Ofen lässt sich mit drei Drehknöpfen bedienen. Der erste Knopf schaltet die Heizlampe im Ofen an. Der zweite Knopf dient zur Einstellung auf Umluft, der dritte Knopf wechselt auf Ober- und Unterhitze. Wir haben also drei Ursachen:

- Knopf 1 gedrückt
- Knopf 2 gedrückt
- Knopf 3 gedrückt

Die Wirkungen:

- Knopf 1 => schalte Heizlampe an
- Knopf 2 => schalte auf Umluft
- Knopf 3 => schalte auf Ober- und Unterhitze

Damit haben wir unsere Spezifikation bereits in bearbeitbare Teile zerlegt und wir haben Ursachen und Wirkungen erkannt. Gehen wir nun auf Schritte 3 und 4 ein, die bisher nur angerissen wurden. Wir müssen nun die logischen Beziehungen zwischen Ursachen und Wirkungen darstellen:

- W1 (Heizlampe an): K1
- W2 (Umluft): K1 ^ K2 ^ !K3
- W3 (Ober- und Unterhitze): K1 ^ K3 ^ !K2
- W4 (falsche Einstellung): (K1 ^ K2 ^ K3) v (!K1 ^ (K2 v K3))

W4 modelliert den Fall, dass alle drei Knöpfe betätigt sind, dann weiß der Ofen nicht, in welchen Modus er schalten soll, oder dass einer der „Modusknöpfe“ gedrückt ist (oder beide), nicht aber der 1. Knopf welcher die Heizlampe anschaltet. In diesem Fall kann nicht in einen Modus gewechselt werden.

Im 4. Schritt erstellen wir nun abschließend den Graphen mit den vorhandenen Mitteln:

- Eine Kante von Ursache i nach Wirkung j bedeutet: i verursacht j
- Eine durchgestrichene Kante bedeutet: wenn Wirkung j, dann Ursache i nicht vorhanden (nicht erlaubt)
- Mehrere Kanten $i_1 \ldots i_n$ nach j bedeuten: Wirkung j wenn mindestens eine Ursache i_x vorhanden
- Mehrere Kanten $i_1 \ldots i_n$ nach j bedeuten: Wirkung j wenn <u>alle</u> Ursachen vorhanden

Da es nur mit „und“ und „oder“-Verknüpfungen dazu kommen kann, dass sich Testfälle wiederholen oder sich widersprechen gibt es in UWGs Möglichkeiten, Testfälle auszuschließen:

- Einschränkung E: Kanten nach Bedingungen b1…bn bedeutet: höchstens eine der Bedingungen darf wahr sein (erfüllte Bedingungen <= 1)

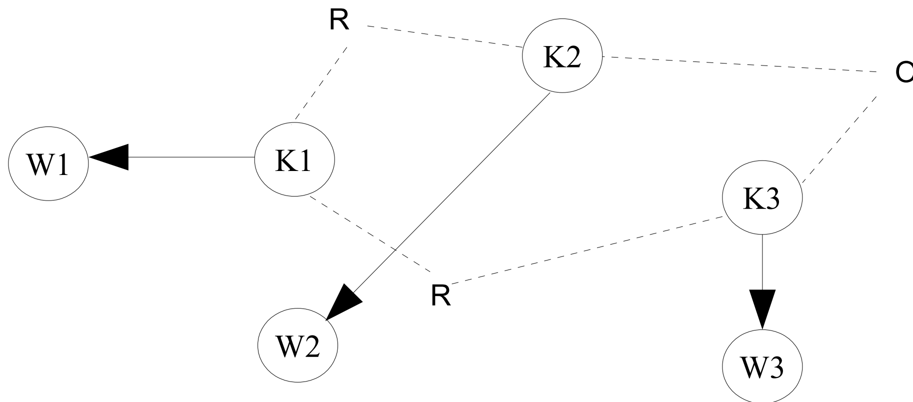
- Einschränkung I: Kanten nach Bedingungen b1…bn bedeutet: mindestens eine der Bedingungen muss wahr sein (erfüllte Bedingungen >= 1)

- Einschränkung O: Kanten nach Bedingungen b1…bn bedeuten: eine und nur eine Bedingung darf wahr sein (erfüllte Bedingungen = 1)

- Einschränkung R: Erfüllung einer mit R verbundenen Bedingung A erfordert Erfüllung von erforderlicher Bedingungen

- Einschränkung M: Erfüllung einer mit M verbundenen Bedingung A impliziert, dass Bedingung B nicht erfüllt ist (vgl. Fehlermaskierung Äquivalenzklassen, F. 24)
- Einschränkung IR: Erfüllung einer mit IR verbundenen Bedingung A bedeutet: Erfüllung weiterer Bedingung B ist irrelevant

Unser Graph:



Im nächsten Schritt müssen wir aus unserem Graphen eine Wertetabelle ableiten. Dazu wählen wir uns eine Wirkung, die erfüllt werden soll (also: Wi = 1) und suchen im Graphen alle Kombinationen von Ursachen, die diese Wirkung erzeugen. In der Wertetabelle legen wir dann für jede gefundene Ursachenkombination eine Spalte an. In der Spalte tragen wir die Werte aller anderen Wirkungen in diesem speziellen Fall ein. Als Beispiel:

- Wenn der Knopf K1 gedrückt ist, dann leuchtet die Heizlampe
- K1 ist also unsere Ursache
- W1 ist unsere Wirkung
- Wir durchsuchen den Graphen nach weiteren Kombinationen, die diese Wirkung erzeugen
- Es ist egal ob K2 oder K3 gesetzt sind oder keins von beiden; wenn in jedem Fall K1 = 1 gilt leuchtet die Lampe; also haben wir 2 weitere Kombinationen
- Nun müssen wir noch die Zustände der andere Wirkungen, also W2 und W3 bestimmen, für den Fall, dass W1 gilt
- Was ist dabei möglich? Wenn W1 gilt kann W2 gelten und W3 nicht, oder W3 und W2 nicht, oder keine Wirkung von beiden

Folgendermaßen sähen dann unsere Spalten aus:

	Testfall #1	Testfall #2	Testfall #3
K1	1	1	1
K2	0	1	0
K3	0	0	1
W1	1	1	1
W2	0	1	0
W3	0	0	1

Damit wir möglichst viele Fehler aufdecken setzen wir außerdem den „entscheidenden“ Wert in diesem Beispiel, d. h. K1 ebenfalls auf 0. Damit können wir aufdecken, ob unser Graph korrekt ist oder nicht:

	Testfall #4	Testfall #5	Testfall #6
K1	0	0	0
K2	0	1	0
K3	0	0	1
W1	0	0	0
W2	0	0	0
W3	0	0	0

Es gilt für Testfälle 4-6, dass W2 und W3 immer 0 sind, da W1 nie gesetzt ist.
Was erkennen wir hier dran? Würde die Wirkung W1 immernoch 1 sein, oder W2 oder W3 irgendwann 1, obwohl wir K1 auf 0 gesetzt haben, dann wüssten wir, dass unser Graph nicht richtig ist.

Zum Abschluss wollen wir noch ein etwas komplexeres Beispiel betrachten. Wir haben dazu keinen UWG gegeben, aber wir nehmen an, dass wir ihn haben. Das Verhalten der modellierten Komponente soll das folgende sein:

- Es handelt sich um ein Smartphone
- Wird der Bildschirm angeschaltet, dann befindet sich das Smartphone im Sperrmodus
- Es muss jetzt ein Code eingegeben werden
- Außerdem darf sich das Handy nicht im Notrufmodus befinden
- Außerdem muss das Handy mit einem Mobilfunknetz verbunden sein
- Wenn alles gilt und der Code korrekt ist, dann wird das Handy entsperrt

Zerlegen wir das Verhalten in Ursachen und Wirkungen:

- U1: Bildschirm angeschaltet
- U2: Mit Mobilfunknetz verbunden
- U3: Modus = Sperrmodus
- U4: Code korrekt eingegeben
- W1: Handy entsperrt
- W2: Handy gesperrt

Unsere Testfälle sehen folgendermaßen aus:

	Testfall #1	Testfall #2	Testfall #3	…	Testfall #i
U1	0	1	1	…	1
U2	0	0	1		1
U3	0	0	0		1
U4	0	0	0		1
W1	0	0	0	…	1
W2	1	1	1		0

Wir sehen, dass die Wertetabellen sehr schnell sehr groß werden. Damit der Vorteil der UWGs nicht aufgehoben wird, kann man die Tabellen minimieren. Eine Vorgehensweise dazu ist, Testfälle zusammenzufassen, die das gleiche Ergebnis haben. Beispielsweise können wir oben alle

Testfälle weglassen, in denen U1 = 0 gilt, also „Bildschirm ist ausgeschaltet“. Denn dann können wir niemals Wirkung W1 erreichen, und es kommt somit immer W1 = 0 heraus.

Mittels UWGs lassen sich fehlersensitive Ursachenkombinationen abbilden, d. h. Fehler die dann auftreten, wenn Bedingungen eintreten, die sich eigentlich ausschließen sollten (Beispiel Knopf für Umluft und Knopf für Ober- und Unterhitze gleichzeitig an). Damit eignen sich UWGs besser als Äquivalenzklassen zum Finden von Fehlern, da sie ähnlich wie bei der Grenzwertanalyse Augenmerk auf fehlerträchtige Eingaben legen. Dennoch sind UWGs mit einem hohen Aufwand verbunden. Es ist daher zu entscheiden ob das System diese Art der Überprüfung benötigt.

Insgesamt haben wir in Kapitel 2.3 gesehen, dass sich Black-Box-Tests mangels Einsicht in den Quellcode an funktionale Spezifikationen halten. Sind diese falsch, werden auch die Testfälle falsche Ergebnisse liefern, die dann allerdings als richtig erachtet werden – sie halten sich schließlich an die Spezifikation. Im Gegensatz zu White-Box-Tests erlauben Black-Box-Tests eine andere Herangehensweise an das Testen, da es hier oftmals darauf ankommt, fehlerträchtige Eingaben zu finden. Im White-Box-Bereich werden wir sehen, dass es darum geht die korrekte Arbeitsweise einer Komponente nachzuweisen.

KAPITEL 2.4: WHITE-BOX-TEST

Im letzten Kapitel:

- Überblick über dynamische Testverfahren: White-Box-Testen
- Nutzen zugrunde liegende Struktur: Quellcode
- Gute Testüberdeckung oftmals aufwändig
- Nicht alle Fehlerklassen auffindbar

White-Box-Test

Im letzten Kapitel haben wir bereits einige Pseudo-UnitTests für einen Beispielcode erzeugt. UnitTests überprüfen, ob eine Funktion so arbeitet wie man es von ihr erwartet. Das macht sie ziemlich nützlich um zu prüfen, ob die angedachte Funktionalität umgesetzt wurde, allerdings bieten sie keinswegs eine umfassende Absicherung dafür.

Warum? UnitTests arbeiten mit einer sehr kleinen Menge von Testwerten. Wie wir bereits im letzten Kapitel gesehen haben kann es aber Anwendungsfälle geben, die beim Erstellen von Funktionen nicht bedacht wurde (oftmals nicht berücksichtigte Grenzfälle). Anfällig für so etwas sind Schleifen. Möglicherweise funktionieren sie mit 2-3 Testdaten korrekt, danach aber nicht mehr. Schauen wir uns ein Beispiel an:

```
public int calculateSquare(int n) {
        return Math.abs(i);
}
```

Offensichtlich sieht man bereits beim Betrachten des Codes, dass die Funktion nicht das hält, was sie verspricht. Angenommen ein Entwickler, der mit der Funktion nicht vertraut ist entwirft jetzt einen UnitTest dafür:

```
AssertEquals(calculateSquare(-1), 1);
AssertEquals(calculateSquare(0), 0);
AssertEquals(calculateSquare(1), 1);
```

Der Test würde durchlaufen und Erfolg vermelden. Warum? Das Quadrat von -1 ist 1, von 0 die 0 und von 1 die 1. Also arbeitet die Funktion exakt für diese drei Testfälle korrekt. Würde man sie mit n = 2 testen würde sie bereits fehlschlagen, denn man erwartet bei Eingabe von 2 als Ergebniswert 4. Unglücklicherweise wurden die Testfälle aber genau so formuliert, dass es den Anschein macht als würde die Funktion korrekt arbeiten.

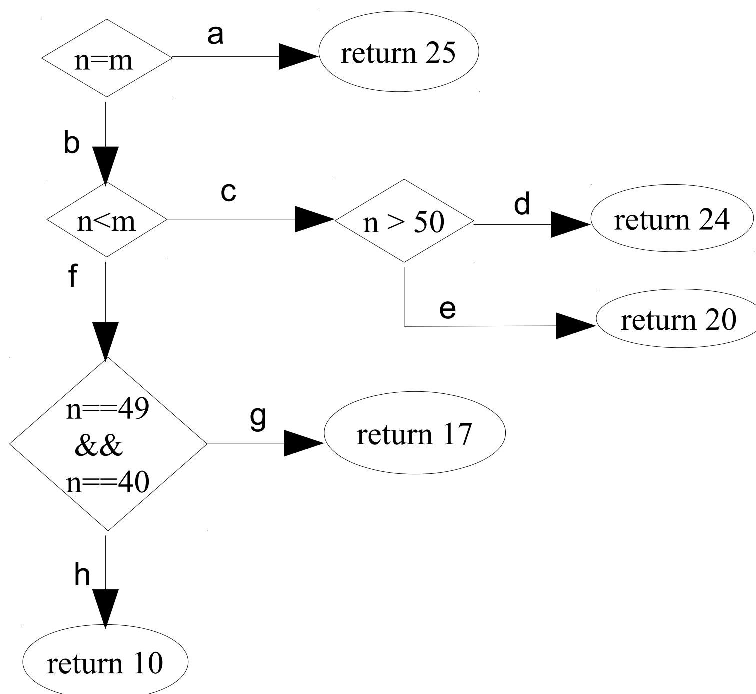
Um so etwas zu vermeiden sind verschiedene Techniken von Nöten welche wir jetzt diskutieren wollen. Zuerst betrachten wir jedoch ein anderes Beispiel:

```
public int doSomething(int n, int m)
{
        if(n == m)
        return 25;

        if(n < m)
        {
                if(m > 50)
                        return 24;
                else
                        return 20;
```

```
        }
        else
        {
                if(n == 49 && m == 40)
                        return 17;
                else
                return 10;
        }
}
```

Was auch immer diese Funktion letztendlich tut, man sieht, dass sie doch eine gewisse Komplexität hat. Wir haben aus unserem Debakel mit der calculateSquares-Funktion gelernt und wollen uns nicht mehr alleine auf UnitTests verlassen. Wie können wir also testen, ob die Funktion das tut, was sie tun soll? Ein erster Ansatz ist zu schauen, ob denn überhaupt alle Anwendungen durchlaufen werden. Dazu benötigen wir eine graphische Darstellung der Funktion:



Zur Überprüfung, ob alle Anweisungen überdeckt werden gibt es die so genannte „Anweisungsüberdeckung“ (C0). Dazu orientiert man sich an der inneren Struktur der Funktion und ermittelt geeignete Testwerte. Mit einer Anweisungsüberdeckung können nicht immer alle Anweisungen gleichzeitig überdeckt werden, insbesondere wenn Entscheidungsknoten in einer Funktion vorhanden sind. Welche Werte müssen wir für unser Beispiel wählen um alle Anweisungen abzudecken?

1. n = m = 0 für n == m
2. n = 0, m = 51 für n < m, m > 50
3. n = 0, m = 50 für n < m, m <= 50
4. n = 49, m = 40 für n > m, n == 49 && m == 40
5. n = 49, m = 39 für n > m, not(n == 49 && m == 40)

Obwohl unsere Funktion recht kurz ist benötigen wir 5 Varianten der Eingabewerte, um alle Anweisungen mindestens einmal zu überdecken. Was haben wir aber davon? Wir wissen jetzt, dass unsere Funktion keinen toten Code enthält. Ob die Funktion aber immer korrekte Ergebnisse liefert bleibt uns verborgen. Der Anweisungsüberdeckungstest eignet sich damit nur für einen ersten Einstieg in die White-Box-Tests.

Also, was gibt es für andere Überdeckung? Mächtiger als die Anweisungsüberdeckung ist die Zweigüberdeckung (C1). Wie der Name schon sagt versucht man dabei, alle Zweige, d. h. alle Kanten eines Kontrollflussgraphen mindestens einmal zu benutzen. Als Beispiel für unseren Funktion:

1. n = m = 0 überdeckt Kante a
2. n = 0, m = 1 überdeckt Kante b, c, d
3. n = 0, m = 51 überdeckt Kante b, c, e
4. n = 49, m = 40 überdeckt Kante b, f, g
5. n = 49, m = 49 überdeckt Kante b, f, h

Die Zweigüberdeckung hat eine etwas größere Aussagekraft als die Anweisungsüberdeckung, weil wir hiermit auch nicht ausführbare Zweige entdecken können. Dennoch sagt auch dieses Testverfahren wenig über eventuelle Fehler im Code aus. Wenn Fehler entdeckt werden, dann nur zufällig.

Sowohl die Anweisungs- als auch die Zweigüberdeckung eignen sich beide außerdem denkbar schlecht für das Testen von Schleifen. Warum? Bei den Verfahren wir jede Anweisung/jeder Zweig höchstens 1mal durchlaufen. Schleifen sind allerdings von Natur aus dazu konzipiert, etwas mehrfach auszuführen. Man könnte die Schleifen natürlich mit vielen verschiedenen Eingaben testen. Besser ist aber die so genannte Grenze-Inneres-Überdeckung (gi).

Was ist die Idee dahinter? Die GI sieht vor, Testfälle zu finden, so dass jede Schleife

- genau einmal
- mehr als einmal
- keinmal (bei ablehnenden Schleifen wie while und for)

durchlaufen wird. Damit sind alle drei Varianten einer Schleife abgedeckt. Betrachten wir ein neues Codefragment:

```
int n;
int m = 1;

for(int i = n; i < 50; ++i)
{
	m = m * 2;
}
```

Diese Schleife berechnet (50 – n)-mal das 2-fache vom aktuellen Wert von m. Wie sehen unsere drei Testfälle aus?

1. Kein Durchlauf: n = 50
2. Genau ein Durchlauf: n = 49
3. Mehr als ein Durchlauf: n = 0

Mit dem GI-Test können wir also prüfen ob die Schleife eine gewisse Anzahl oft durchlaufen wird. Wiederum nicht erkennbar ist, ob die Schleife an sich korrekt arbeitet, d. h. den richtigen Wert

berechnet. Selbst wenn wir sie n-mal durchlaufen lassen ist dadurch nicht gesagt, dass sie auch für den n+1-ten Durchlauf noch korrekt arbeitet.

Die bisher vorgestellten Verfahren waren wenig zufriedenstellend. Man muss einen recht großen Aufwand treiben um geeignete Testfälle zu finden, dennoch weiß man danach nichts darüber, ob eine Funktion korrekt arbeitet oder nicht. Gewiss kann man mit C0, C1 und GI Fehler aufdecken, aber es geschieht zufällig.

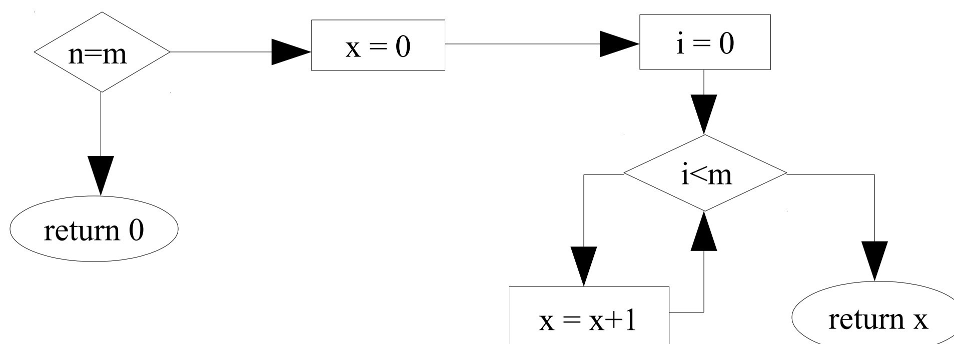
Wir wollen jetzt das Verfahren der Pfadüberdeckung betrachten. Geschrieben wir dieses als C_∞, wir schreiben abkürzend CU (C unendlich). Die CU fordert, dass jeder Pfad mindestens einmal ausgeführt wird. Wir erinnern uns: ein Pfad ist eine Folge von Knoten und Kanten, die mit dem Startknoten beginnt und mit dem Endknoten endet. Eine terminierende Abfolge von Anweisungen eines Programms also. Die CU ist dabei eine Kombination aus C0 und C1 und GI. Schauen wir uns noch ein Beispiel an:

```
public int function(int n, int m)
{
	if(n == m)
	{
		int x = 0;

		for(int i = 0; i < m; ++i)
		{
			x = x + 1;
		}

		return x;
	}
	else
	{
		return 0;
	}
}
```

Die graphische Darstellung zur Funktion:



Welche Testfälle gibt es für eine CU in diesem Beispiel?

1. n = 0, m = 1 für return 0 (else-Zweig)
2. n = m = 0 für return 0 (Schleife wird nicht durchlaufen)
3. n = m = 1 für return 1 (Schleife wird 1 mal durchlaufen)
4. n = m = 2 für return 2 (Schleife wird 2 mal durchlaufen)

   …

n. n. n = m = i für return i (Schleife wird i-mal durchlaufen)

Die Schleifendurchläufe können mit beliebig vielen Testfällen beliebig erhöht werden. Eine obere Schranke für die Durchläufe ist der MAX_INT Wert des zu Grunde liegenden Betriebssystems. Wenn wir also viele Durchläufe machen finden wir doch bestimmt auch viele, wenn nicht sogar alle Fehler? Leider nein, denn wir probieren nicht immer alle möglichen Variablenbelegungen durch. In diesem einfachen Beispiel tun wir das, sobald wir aber im Schleifenrumpf mit mehr als einer Variablen arbeiten wird es kompliziert. Außerdem kann die CI natürlich nur das testen was da ist. Vielleicht wollen wir ja neben dem Aufaddieren auf x ebenfalls in jedem Durchlauf n = n * n rechnen? Wenn wir vergessen haben das zu programmieren wird es natürlich nicht durchlaufen und damit nicht getestet.

Dennoch scheint die Pfadüberdeckung besser als die C0, C1 und GI zu sein, da sie alles kombiniert. Mit der CU lassen sich recht viele Kontrollabläufe testen und damit steigt die Wahrscheinlichkeit, Fehler zu finden. Wir wollen dennoch weitere Testverfahren betrachten.

Test der Bedingungen

Programme werden erst dann interessant, wenn es in ihnen Entscheidungspunkte gibt. Rein sequenzielle Programme sind leicht zu testen. Was passiert in Entscheidungspunkten? Die angegebenen Bedingungen werden ausgewertet, und zwar immer zu true oder false, völlig unabhängig vom verwendeten Datentyp:

- if(n == 5) wird zu booleschem Wert
- if(objA.equals(objB)) wird zu booleschem Wert
- if(_saveConnection) wird zu booleschem Wert

Dabei muss in der Abfrage nicht nur eine einzelne Bedingung stehen. Mehrere sind ebenfalls möglich:

```
if(_hasConnection && (!isEmpty(server_name) || connect() == SUCCESS_MSG))
```

ist ein Beispiel für eine komplexere Abfrage. Wie kann man so etwas auf Korrektheit prüfen? Man könnte den gesamten Ausdruck zu true oder false auswerten, indem man entsprechende Belegungen vorgibt. In unserem Beispiel kann man bspw. _hasConnection auf false setzen. Dann wird automatisch der komplette Ausdruck zu false, da wir eine Und-Konjunktion haben. Wenn wir so etwas tun geht uns aber viel Detailinformation verloren. Außerdem werden dadurch leicht Fehler übersehen. Möglicherweise liefert die Methode connect() ja immer SUCCESS_MSG, weil sie fehlerhaft implementiert ist. Mit einer Gesamtauswertung des Ausdrucks würde das nicht auffallen.

Die Idee bei der so genannten Bedingungsüberdeckung ist daher, alle atomaren Prädikatterme einzeln auszuwerten. Betrachten wir ein einfaches Beispiel:

```
if((n == 5) || (m < 10))
        return true;
else
        return false;
```

Die beiden Teilterme stehen jeweils in Klammern. Für die einfache Bedingungsüberdeckung (C2) muss jetzt jeweils ein Testdatum gefunden werden, so dass der Ausdruck einmal zu true und einmal zu false ausgewertet wird:

n	m	Gesamt
== 5	< 10	true
== 5	>= 10	true
!= 5	<10	true
!= 5	>= 10	false

Unglücklicherweise kann es passieren, dass wir bei der einfachen Bedingungsüberdeckung Testdaten so wählen, dass wir nicht alle Zweige überdecken. Als Beispiel:

1. Testfall: n = 5, m = 9
2. Testfall: n = 4, m = 10

Unser Gesamtausdruck wird nun einmal zu true (1. Testfall), und einmal zu false (2. Testfall) ausgewertet. Es werden ebenfalls alle Zweige überdeckt. Schreiben wir die Abfrage anders:

```
if(n == 5) // A
if(m < 10) // B
        return true;
else // C
        return false;
else // D
        return false;
```

Wie sieht es jetzt aus? Mit unseren beiden Testfällen durchlaufen wir die Zweige A, B und D. Zweig C verpassen wir, weil nach der Auswertung von n == 5 sofort abgebrochen wird. In Code steht dort nämlich jetzt kein Oder mehr, sondern ein Und:

```
if((n == 5) && (m < 10)) …
```

Unser C2-Test ist also nicht umfassend genug. Als Verbesserung des C2-Tests gibt es die so genannte Zweig-/Bedingungsüberdeckung, die eine Kombination aus C2 und C1 ist. Man kann aber zeigen, dass damit nicht zwingend alle Zweigkombinationen durchlaufen werden. Also muss ein weiteres Verfahren her: die Mehrfachbedingungsüberdeckung (C3):

Bei der Mehrfachbedingungsüberdeckung probiert man _jede_ mögliche Kombination von Prädikaten für die einzelnen atomaren Terme aus. Für einen komplexeren Ausdruck wird die Tabelle dann sehr groß:

Not(A ^ B) v (C ^ D)

A	B	C	D	Gesamt
true	true	true	true	true
true	true	true	false	false
true	true	false	true	false
true	true	false	false	false
true	false	true	true	true
true	false	true	false	true
true	false	false	true	true
true	false	false	false	true
false	true	true	true	true
false	true	true	false	true
false	true	false	true	true
false	true	false	false	true
false	false	true	true	true
false	false	true	false	true
false	false	false	true	true
false	false	false	false	true

Für noch komplexere Ausdrücke wird die Tabelle dementsprechend größer. Es gilt:

Größe der Tabelle = 2^Anzahl Variablen

Also bei 6 Variablen schon 2^6 = 64 Möglichkeiten. Das ist natürlich zu aufwendig. Als bessere Möglichkeit bietet sich daher die minimale Mehrfachbedingungsüberdeckung an. Die Idee dahinter ist simpel: anstatt alle Kombination zu betrachten werden nur diejenigen ausgewählt, bei denen die Änderung _eines_ booleschen Wertes eines Terms das Ergebnis des gesamten Terms verändert.

Jetzt haben wir eine Reihe von Testverfahren zur Ermittlung der Codeüberdeckung vorgestellt. Alle diese Verfahren arbeiten allerdings auf sehr einfachem Code. Problematisch wird es, wenn bspw. boolesche Ausdrücke in Variablen ausgelagert werden oder Objektorientierung wie in Java oder C++ zum Einsatz kommt. Programme laufen dann nicht mehr sequenziell ab sondern sind durch Beziehungen zwischen Objekten geprägt. Wir wollen daher nun einen Blick auf datenflussbasierte Tests werfen die versuchen, diesem Problem Herr zu werden.

Datenflussbasiertes Testen:

Beim datenflussbasierten Testen wird Augemerk auf die Interaktion zwischen Anweisungen gelegt. Es geht also nicht mehr so sehr um einzelne Anweisungen und dass sie erreicht werden, sondern wie sie sich gegenseitig beeinflussen. Was bedeutet in diesem Kontext beeinflussen? Als Beispiel:

```
int x = 5, a = 20;
a = a * 2;
x = x + a;
```

Die Variable a beeinflusst in diesem Beispiel die Variable x, da sie auf x addiert wird und somit ihren Wert ändert. Man sagt auch, x referenziert a bei der Zuweisung. Die initiale Belegung von x nennen wir Definition.

Das Datenfluss-Testen hat ebenfalls wie das Kontrollflusstesten das Ziel, möglichst viele Fehler zu finden ohne eine vollständige Pfadüberdeckung erreichen zu müssen. Wir haben gesehen, dass das sehr aufwändig ist und der Nutzen oftmals in keinem Verhältnis zum Aufwand steht.

Wir verwenden im Datenfluss-Testen einen Datenflussgraph, dieser ist ähnlich dem Kontrollflussgraph. Der Datenflussgraph besitzt aber zu jedem Knoten noch folgende Mengen:

- DEF(k): Menge der Variablen die innerhalb des Codeblocks von k einen Wert zugewiesen bekommen; dieser wird im gleichen Codeblock nicht wieder undefiniert
- UNDEF(k): Menge der Variablen, die innerhalb des Codeblocks von k undefiniert werden
- REF(k): Menge der Variablen innerhalb des Codeblocks von k die in k referenziert werden

Verdeutlichen wir die Mengen an einem Beispiel:

```
VAR_A = X;
VAR_B = Y;
VAR_C = VAR_B + VAR_A;
FREE(VAR_B);
```

Unsere Mengen sind:

- DEF(k) = {VAR_A, VAR_C}
- UNDEF(k) = {VAR_B}
- REF(k) = {X, Y, VAR_B, VAR_A}

DEF(k) enthält VAR_A und VAR_C, da diese definiert werden. VAR_B würde auch dazu gehören, wird aber später mittels FREE(VAR_B) undefiniert und darf deswegen nicht in die Menge. VAR_B ist dafür in UNDEF(k) enthalten. REF(k) schließlich enthält einfach alle Variablen, die irgendwann auf der rechten Seite einer Zuweisung stehen. Ein kleiner Trick:

Würde man VAR_B vor der Zuweisung zu VAR_C undefinieren, dann dürfte VAR_B ebenfalls nicht in REF(k) sein. Die Variable wäre dann undefiniert – uninitialisierter Speicher in einem realen System. Dann darauf zuzugreifen ist immer eine schlechte Idee.

Ebenfalls vermieden werden sollen lokale Datenflüsse. Was ist das? Ein lokaler Datenfluss ist eine Referenz gefolgt von einer Definition:

```
VAR_B = X; VAR_C = VAR_B;
```

Denn das hätte den gleichen Effekt wie:

```
VAR_C = X;
```

Falls doch so ein Fall auftritt, dann wird nach der Referenz der Codeblock gesplittet in zwei Teile.

DR-Wege: ein DR-Weg ist ein Weg von x in einem Knoten k zu x in einem Knoten l gdw. X auf dem Weg von k nach l weder neu definiert noch undefiniert wird. Die Variable wird also nicht angefasst. Sie darf allerdings als Referenz genutzt werden:

```
VAR_B = 5;
A = D;
C = 20;
D = C * A;
IF(VAR_B > 10) …
```

Ein DR-Weg wäre hier von Zeile 1 bis Zeile 5: VAR_B wird zwischendurch nicht veränder, denn das würde möglicherweise das Ergebnis verfälschen.

Alle-Definitionen:

Das Kriterium Alle-Definitionen besagt, dass das Resultat jeder Zuweisung wenigstens einmal benutzt werden muss. Damit ist es ein sehr wichtiges Kriterium, denn es verhindert, dass Variablen initialisiert/deklariert werden, aber nie benutzt. Ein guter Compiler erkennt solche ungenutzten Variablen und listet sie als Warnung an den Entwickler auf. Diese Variablen zu entfernen erhöht die Lesbarkeit des Codes und verkleinert den Code.

Alle DR-Interaktionen:

Dieses Kriterium ist sehr ähnlich zu all-Defs. Es besagt, dass es im Datenflussgraphen einen Weg geben muss, für den die Definition einer Variablen x die Referenz der Variablen x erreicht. Als Beispiel:

```
VAR_B = 5;
VAR_C = 10;
A = VAR_B + 1;
IF(A > 10) …
```

VAR_B erfüllt DR-Interaktion, denn die Variable taucht als Referenz in der Variablen A auf. VAR_C allerdings bleibt unbenutzt. Sie erfüllt DR-Interaktion *nicht*.

Alle Referenzen:

Alle Referenz besagt, dass die Variable, die durch die Referenz einer Variablen x einen Wert bekommen hat, danach noch einmal benutzt werden muss. Denn sonst würde es keinen Sinn machen die Zuweisung vorzunehmen. Als Beispiel:

```
VAR_B = 5;
VAR_C = 10;
VAR_A = VAR_B + 20;
VAR_D = VAR_C * 2;
IF(VAR_A > 10) …
```

VAR_B wird zu VAR_A zugewiesen. Danach wird VAR_A in einer if-Abfrage benutzt, der neue an VAR_A durch Benutzung von VAR_B zugewiesene Wert findet also Verwendung. VAR_C wiederum wird an VAR_D zugewiesen, VAR_D findet aber anschließend keine Verwendung mehr. Die Zuweisung war also überflüssig.

Alle vorgestellten Kriterien zum Datenfluss-Testen sollen vermeiden, dass Variablen initialisiert aber nicht benutzt, oder Zuweisungen getätigt und nicht benutzt werden. Dadurch wird der Code übersichtlich gehalten und es sammelt sich kein toter Code an, was wiederum beim Testen hilfreich ist – denn toten Code kann man nicht überdecken. Fällt er also weg ist man dieses Problem los. Man spricht bei der Initialisierung einer Variablen auch vom definitional use, def(x),

bei der Referenzierung einer Variablen vom Berechnungs-Referenz-/Computational-Use, kurz c-use(x), und bei der Benutzung in Abfragen von Entscheidungs-Referenz/Predicative use, kurz p-use(x). Als Beispiel:

```
VAR_B = 5; // def(VAR_B)
VAR_A = VAR_B + 5; // def(VAR_A), c-use(VAR_B)
if(VAR_A > 10 && VAR_B < 100) ... // p-use(VAR_A), p-use(VAR_B)
```

Das ist ein einfaches Beispiel für def, c-use und p-use. Wir können ebenfalls die bereits vorgestellten Kriterien in Verbindung mit Computational/Predicative-Usage nutzen. Dazu ein Beispiel:

```
1.    var A = 10, B = 5;
2.    if(A > 10) {
3.        var D = A + A; }
4.    else {
5.      var C = B * 2; }
6.    A = 20;
7.    if(A + 5 > 40) {
8.      B = 100; }
9.    A = 50;
```

Wenden wir nun das Kriterium All-Defs auf das Codebeispiel an. Dabei gilt, dass jede Definition einer Variablen, d. h. *def(x)*, mindestens einmal in einem *c-use(x)* oder *p-use(x)* verwendet werden muss, ohne dass sie erneut definiert wird. Erfüllt unser Codebeispiel das Kriterium? Fangen wir mit Variable B an:

- In Zeile 1: def(B)
- In Zeile 5: c-use(B)
- In Zeile 8: def(B)

Für Variable B ist das Kriterium also offensichtlich erfüllt. Wie sieht es mit A aus?

- In Zeile 1: def(A)
- In Zeile 2: p-use(A)
- In Zeile 3: c-use(A)
- In Zeile 6: def(A)
- In Zeile 7: p-use(A)
- In Zeile 9: def(A)

Auch Variable A erfüllt unser Kriterium. Würden wir in Zeile 7 beispielsweise auf *if(B + 5 > 40)* vergleichen, dann würde das *def(A)* in Zeile 6 direkt von dem *def(A)* in Zeile 9 gefolgt werden. Damit wäre das Kriterium all-defs dann nicht erfüllt.

Wie sieht es für das Kriterium alle DR-Interaktionen aus? Zur Erinnerung: alle DR-Interaktionen besagt, dass es einen Weg von *def(x)* zu *ref(x)* (d. h. p-use oder c-use) geben muss, ohne dass dazwischen erneut *def(x)* ausgeführt wird. Ein Beispiel:

```
1.    var A = 50, B = 10;
2.    if(A > 40) {
3.      B = B + 10;
4.      var C = 2 * B; }
5.    if(A <= 40) {
6.      var C = B + 5;
7.      B = 100; }
```

Das vorliegende Codebeispiel erfüllt alle DR-Interaktionen **nicht**. Warum? Schauen wir uns Variable B an:

- Zeile 1: def(B)
- Zeile 3: def(B)
- Zeile 4: c-use(B)
- Zeile 6: c-use(B)
- Zeile 7: def(B)

Weil bei alle DR-Interaktionen gefordert ist, dass die Eigenschaft für alle Paare von Definitionen/Referenzen gilt ist die Eigenschaft hier nicht erfüllt. Um sie wieder zu erfüllen bräuchten wir einen entsprechenden else-Zweig für die erste if-Abfrage. In diesem Zweig müsste B dann zuerst referenziert werden bevor es neu definiert wird.

Betrachten wir nun das Kriterium Alle Referenzen. Es gilt: wird die Definition einer Variablen X in einem Entscheidungs- oder Berechnungsknoten K referenziert, dann muss der in Knoten K berechnete Wert in einem Knoten L verwendet werden, so dass gilt: es gibt einen Pfad von *def(x)*nach L über K. Für Entscheidungsknoten gilt, dass der in einem Entscheidungsknoten referenzierte Wert dann in allen vom Entscheidungsknoten ausgehenden Zweigen Verwendung finden muss. Beispiel:

```
1.    var A = 50;
2.    if(A > 40) {
3.        var B = 200; }
4.    var C = A + 50;
5.    var D = C * 2;
```

Wie sieht es hier aus? Betrachten wir zuerst B-Referenzen:

- In Zeile 1: def(A)
- In Zeile 4: c-use(A)
- In Zeile 5: c-use(c-use(A))

Für B-Referenzen ist das Kriterium erfüllt, den die Referenz von A wird weiter verwendet. Gäbe es Zeile 5 nicht, dann wäre die Referenz von A in Zeile 4 sinnlos gewesen, da das Ergebnis nicht verwendet wird. Die Referenz auf C in Zeile 5 ist in diesem Beispiel also sinnlos. Wie sieht es mit E-Referenzen aus?

- n Zeile 2: p-use(A)

Wieder wird A nur in der Entscheidung selbst, nicht aber in den ausgehenden Zweigen verwendet. Wir sind an dieser Stelle darauf angewiesen, dass A im (momentan leeren) else-Zweig Verwendung findet.

Kriterium alle k-DR-Interaktionen. K-DR-Interaktionen sind das gleiche wie DR-Interaktionen, nur, dass man diesmal Referenzen folgt, und zwar k Schritte lang. Beispiel:

```
1.  var A = 50;
2.  var B = A * 2;
3.  var C = B * 5;
4.  var D = C + 2;
```

Normale DR-Interaktion von A:

- Zeile 1: def(A)
- Zeile 2: c-use(A)

k-DR-Interaktion mit k = 3:

- Zeile 1: def(A)
- Zeile 2: c-use(A)
- Zeile 3: c-use(B)

k-DR-Interaktion mit k = 4:

- Zeile 1: def(A)
- Zeile 2: c-use(A)
- Zeile 3: c-use(B)
- Zeile 4: c-use(C)

Das bedeutet also: A wird in Zeile 2 referenziert, B in Zeile 3, C in Zeile 4. Damit ist D von A abhängig, denn ohne A kann der Wert von D nicht berechnet werden.

Letztes Beispiel: Kontextüberdeckung. Bei der Kontextüberdeckung wird geprüft, woher Referenzen für eine Berechnung kommen. Es findet also bei Verwendung von Bedingungsknoten Verwendung. Als Beispiel:

```
var A = 50;
var B = 100;
var C = A + B;
```

In diesem Beispiel besteht kein Zweifel daran, woher die Werte von A und B kommen, da es keine Entscheidungen auf dem Weg gibt. Anders hier:

```
1.  var A = 50;
2.  var B = 100;
3.  if(A + B > 101) {
4.    var A = A * 2;
5.    var B = 700; }
6.  var C = A + B;
```

In diesem Beispiel gibt es zwei Möglichkeiten: entweder die Werte für A und B kommen aus Zeile 1 und 2, oder sie kommen aus Zeile 4 und 5, sofern die if-Bedingung erfüllt ist.
Ingsesamt kann man mit Datenfluss-Testen für abstraktere Programme, d. h. Programme in denen

Variablen referenziert und nicht offensichtlich benutzt werden, besser testen. Das Verfahren hat jedoch auch seine Schwächen: fehlende Pfade sind damit nicht aufdeckbar, da das Endergebnis nicht auf Korrektheit geprüft wird. Ebenso können Bereichsfehler, d. h. Fehler, die während der Berechnung passieren nicht aufgedeckt werden. Das Datenfluss-Testen befasst sich lediglich damit, dass Variablen benutzt werden, nicht aber was genau ihr Inhalt ist.

Was gibt es noch für Verfahren? Wir haben in diesem Kapitel viele Testverfahren kennen gelernt. Sie alle waren dynamischer Natur, d. h. basierend auf der „Ausführung“ von Code. Neben dynamischen Verfahren gibt es noch statische Verfahren. Diese basieren auf der Analyse von Quellcode, Kontroll- oder Datenflüssen. Statische Verfahren lassen sich gut automatisieren, bspw. auch um Codemetriken zu ermitteln. Sie decken jedoch nur eine bestimmte Klasse an Fehlern auf. Fehler, die man bspw. mit einem Grenze-Inneres-Test finden kann, wird man niemals mit statischen Methoden finden.

Beispiele für eine Analyse des Kontrollflussgraphen sind beispielsweise Sprünge aus einer Funktion, ohne dass für alle Kontrollpfade ein Wert zurückgeben wurde. Als Beispiel:

```
public int getX() {
        if(x == 5)
                return x;
}
```

Der C++-Compiler von Microsoft würde in diesem Beispiel anmerken, dass der else-Teil der Abfrage (hier weggelassen) keinen Rückgabewert liefert. Für den Fall, dass die Variable x nicht den Wert 5 hat wäre der Rückgabewert der Funktion dann undefiniert.

Ein anderes einfaches Beispiel sind Vorgänger-Nachfolger-Tabellen. Die Tabellen bilden den Ablauf eines Programms ab. Hat eine Anweisung keinen Vorgänger, dann wird sie nicht erreicht. Dann kann man sie entfernen. Ausnahme ist natürlich die erste Anweisung eines Programmteils, analog für die letzte Anweisung. Beispiel:

```
1.    | var A = 50, var C;
2.    | if(A > 10) {
3.    |     var B = 100;
4.    |   C = A * 5 + B; }
5.    | return C;
```

Vorgänger-Nachfolger-Tabelle für dieses Beispiel:

Anweisung	Vorgänger	Nachfolger
1	-	2
2	1	3, 5
3	2	4
4	3	5
5	2, 4	-

Statische Datenflussanalyse schließlich untersucht Code bspw. auf verwendete, aber nicht initialisierte Variablen – eine große Fehlerquelle, da uninitialisierte Variablen beliebige Werte haben können. Oftmals kann es hierbei (in C++) auch zu Abstürzen kommen, da auf nicht freigegebenen Speicher zugegriffen wird.
Folgende Datenfluss-Zustände gibt es:

- u: undefiniert
- d: definiert
- r: referenziert

Daraus ergeben sich folgende Anomalien:

- ur: eine Variable wird verwendet, ohne dass sie vorher initialisiert wurde
- du: eine Variable wird initialisiert, danach aber nicht mehr verwendet
- dd: eine Variable wird intialisiert und bei ihrer nächsten Verwendung wieder überschrieben; der erste Wert kommt nie zum Einsatz

Als Beispiel:

```
VAR_B = 5;
VAR_B = 10; // dd-Anomalie
VAR_A = VAR_C; // ur-Anomalie
VAR_B = VAR_A; // dd-Anomalie, du-Anomalie
```

Insbesondere die ur-Anomalie in Zeile 3 kann unerwünschte Nebenwirkungen haben. Fehlerhafte Ergebnisse in Berechnungen sind möglich, ebenso Abstürze des Programms.

Eine andere Möglichkeit ist die so genannte „symbolische Ausführung“ eines Programms. Dabei werden die konkreten Variablen im Code ersetzt. Dann kann mit dem „symbolischen Code“ „gerechnet“ werden. Die Auswertung kann dabei logisch-formalisierte Annahmen an Werte benutzen (also beispielsweise erlaubte Werte) oder auch algebraische Eigenschaften der verwendeten Operatoren (neutrales Element, etc.).

Der Vorteil beim symbolischen Testen ist, dass damit auch ein Großteil der normalen Testfälle abgedeckt wird. Im Gegensatz zu formaler Programmverifikation wie dem Model Checking ist außerdem keine formale Programmspezifikation notwendig, was die Hürden für eine symbolische Ausführung senkt. Ein Nachteil dieser Technik ist, dass man für die „Berechnung“ der Wert einen Theorembeweiser benötigt, da man nicht mit konkreten, sondern symbolischen Werten rechnet und daher allgemeingültige Aussagen über diese treffen muss. Es gibt allerdings keine vollständigen automatischen Theorembeweiser für die gängigen Hochsprachen.

Damit haben wir für dieses Kapitel einen guten Überblick über dynamische Testverfahren und eine kurze Einführung zu statischen Verfahren erhalten.

KAPITEL 2.5: TESTEN IM SOFTWARELEBENSZYKLUS

Im letzten Kapitel: Black-Box-Tests

- Unterschied zu White-Box-Tests: interne Struktur von Testobjekten unbekannt
- Techniken legen Augenmerk auf fehlerträchtige Eingaben; White-Box-Tests: Versuch, Korrektheit von Komponenten nachzuweisen
- Verschiedene Verfahren: Äquivalenzklassen, Grenzwertanalyse, Zustandstests (geeignet für objektorientierte Programme), Entscheidungswerttabelle, Ursache-Wirkungs-Graphen

Testen im Softwarelebenszyklus

Nachdem wir uns in den bisherigen Kapiteln zum Testen von Software sehr viel mit Details zu konkreten Techniken und Verfahren auseinandergesetzt haben wollen wir jetzt einen Blick auf den übergeordneten Ablauf des Testens im Softwarelebenszyklus werfen.

Was ist ein Softwarelebenszyklus? Der Zyklus beschreibt die Lebensdauer eines Softwareproduktes, angefangen bei der Planung bis hin zur Auslieferung an den Kunden und einen sich anschließenden Wartungszeitraum. Wir wissen bereits aus SWT um die verschiedenen Stufen des sequentiellen Entwicklungsmodells, das so genannte V-Modell. Beim V-Modell entsprechen dabei die auf der linken Seite stehenden Konstruktionsphasen jeweils einer Teststufe, zu finden auf der rechten Seite auf gleicher Höhe. Beispielsweise entspricht die Anforderungsdefinition dem Abnahmetest. Die Anforderungsdefinition steht dabei am Anfang eines Projektes. Hier wird festgelegt, was das Produkt können soll, welchen Einsatzzweck es hat, etc. Der Abnahmetest findet dann ganz am Ende statt. Oftmals wird er durch den Kunden, der das Produkt in Auftrag gegeben hat durchgeführt. Der Kunde testet dann das System und entscheidet, ob seine Anforderungen aus der Anforderungsdefinition alle zu seiner Zufriedenheit umgesetzt wurden. Beim Abnahmetest geht es daher nicht mehr darum, eventuelle Fehler aufzudecken wie bei niedrigeren Teststufen. Idealerweise treten im Abnahmetest keine Fehler mehr auf. Sie sollten alle bereits zuvor entdeckt und behoben worden sein. Natürlich kann es sein, dass der Kunde einen neuen Fehler entdeckt, der zuvor nicht aufgefallen ist. Das kann daher kommen, dass ein Kunde – anders als Entwickler und QS-Mitarbeiter – oftmals ein anderes Verständnis von Bedienkonzepten eines Systems hat und Produkte so bedient, wie es in vorherigen Teststufen nicht bedacht wurde. Erfahrungsgemäß haben Kunden ein Talent für so etwas.

Das Testen im Softwarelebenszyklus umfasst also Zusammenhänge zwischen Entwicklungs- und Testaktivitäten. Das beinhaltet Komponenten- (auch Modul- oder Unit Tests genannt), Integrations-, System- und Abnahmetests. Des Weiteren sind auch Regressions- und Akzeptanztests darin enthalten. Beim Testen werden neu entwickelte Produkte getestet ebenso wie Updateversionen. Dabei wird geprüft, in wie fern sich eine neue Version gegenüber einer alten verhält, ob nach einem Update noch alle gewünschte Funktionalität vorhanden ist, Komponenten weiterhin korrekt untereinander kommunizieren, etc.

Wie genau getestet wird hängt jeweils von den Präferenzen der Projektleitung und dem zu Grunde liegenden Entwicklungsmodell ab. Das in der Vergangenheit oft verwendete Wasserfallmodell bspw. sieht vor, dass Integrations- und Systemtests erst sehr spät im Entwicklungsprozess durchgeführt werden. Das führt allerdings dazu, dass der Testaufwand zuerst gering ist, ab einem bestimmten Zeitpunkt dann aber stark zunimmt. Ab diesem Zeitpunkt unterliegt das Testen dann einem strikten Zeitplan. Fallen nun ein oder mehrere Mitarbeiter aus, ist der Release Termin des ganzen Produktes gefährdet, da man möglicherweise nicht mehr rechtzeitig mit allen Tests fertig wird.

Im Gegensatz dazu sind iterativ-inkrementelle Entwicklungsmodellen wie bspw. Scrum darauf ausgelegt, nach jeder Iteration ein funktionsfähiges, getestetes Produkt vor sich zu haben. Bei Scrum werden beispielsweise so genannte Sprints durchgeführt. Diese Sprints dauern ca. 2

bis maximal 4 Wochen, in denen zuvor definierte Anforderungen umgesetzt werden. Ein Vorteil gegenüber dem Wasserfallmodell ist, dass man (zumindest in der Theorie) am Ende eines Sprints immer ein ausführbares Produkt hat. Beim Wasserfallmodell kann es hingegen vorkommen, dass Teilkomponenten eines Produktes über Wochen und Monate hinweg nicht oder nur fehlerhaft funktionieren bis sie im Zuge eines Bugfixings kurz vor Beginn der Testphase behoben werden.

In den iterativ-inkrementellen Entwicklungsmodellen ist es üblich, einen hohen Grad an Testautomatisierung anzuvisieren. Das geschieht meistens über den Einsatz von Unit Tests. Diese lassen sich automatisiert in jedem Build ausführen. Eine Auswertung der Ergebnisse kann ebenfalls automatisiert geschehen. Auf diese Weise hat man sofort einen Überblick, ob in einer Iteration neu hinzugekommener Code etwas kaputt gemacht hat oder nicht. Für die neue Funktionalität werden ebenfalls neue Tests angelegt, die dann ab dem nächsten Build ausgeführt werden.

Komponenten- /Unit-Test:

Komponenten- bzw. Unit Tests bilden die unterste Teststufe ab. Es werden dabei modulare (atomare) Bausteine eines Produkts getestet. Das sind entweder einzelne Funktionen oder Klassen. Unit Tests sollten **nicht** auf andere Klassen außer der zu testenden angewiesen sein. Dennoch kommt das leider immer wieder vor. Ein solches Verhalten kann ein Hinweis darauf sein, dass Klassen untereinander zu stark abhängig sind (vgl. auch Kapitel 3.2 – Softwaremetriken).

Ein einzelner Komponententest besteht dabei aus dem so genannten Testtreiber und Platzhaltern. Der Testtreiber ruft Dienste des zu testenden Objektes auf – also Methoden einer Klasse. Die Platzhalter simulieren Dienste, auf die das Testobjekt angewiesen ist. Beispielsweise gibt es eine Methode die überprüft, ob eine bestimmte Software installiert ist. Ein Indikator dafür kann sein, dass im Falle einer Installation ein bestimmter Registryschlüssel gesetzt ist. Ein Platzhalter würde nun diesen Schlüssel anlegen. Dann ruft der Testtreiber die entsprechende Methode auf und überprüft das Ergebnis. Es ist dabei wichtig, dass die angelegten Ressourcen **unbedingt** nach Durchlauf des Tests wieder weggeräumt werden – in diesem Beispiel der Registryschlüssel. Was passiert sonst beispielsweise?

```
public bool isInstalled()
{
        String key = ReadRegisty(…);
        return key.isEmpty();
}

void testIsInstalled()
{
        Assert(isInstalled() == false);
        createRegistryKey(…);
        Assert(isInstalled() == true);
}
```

Was würde hier passieren? Der erste *assert*-Aufruf in *testIsInstalled()* würde *true* liefern. Warum? Der Registryschlüssel ist nicht gesetzt, also liefert *isInstalled() false*. Das wird mit *false* vergleichen, was wiederum *true* liefert. Danach wird der Schlüssel angelegt und ein erneuter Vergleich durchgeführt, diesmal auf *true*. Da der Schlüssel gesetzt ist liefert *isInstalled()* diesmal *true*. Der Vergleich mit *true* liefert *true*, also alles in Ordnung. Im nächsten Durchlauf des Tests schlägt die erste Assertion allerdings fehl. Denn *isInstalled()* wird nun *true* liefern. Wieso? Wir haben vergessen, den zu Testzwecken angelegten Registryschlüssel wieder wegzuräumen.

Es kann unter Umständen lange dauern, einem solchen Fehler auf die Schliche zu kommen. Frameworks wie JUnit/CPPUnit bieten für Aufräumzwecke die Methode *tearDown()*, die nach jedem Testcase aufgerufen wird. Hier kann man dann alle Aufräumarbeiten hineinlegen. Das Ganze ist eine Realisierung des RAII-Prinzips (Resource Acquisiton is Initialization) welches in C++ weite Verbreitung findet (in Java wegen Garbage Collection nicht zwingend notwendig).

Test-driven development

Es gibt Ansätze, das so genannte test-driven development, zuerst Komponententests zu schreiben und dann erst den Programmcode. Auf diese Weise gießt man das gewünschte Verhalten einer Komponente in Code, bevor man ihr tatsächlich es Verhalten implementiert. Obwohl der Ansatz vielversprechend ist scheitert er oftmals daran, dass viele Entwickler die Einstellung haben „mal eben was zu programmieren“ ohne zuvor Tests dafür zu erstellen. Meistens ist es sogar so, dass auch nach dem Programmieren keine Testfälle geschrieben werden, sondern erst viel, viel später. Dann ist der Code aber unter Umständen nicht mehr verständlich und die Formulierung von Testfällen ist schwierig. Ebenso ist es dann nicht einfach den Grund für ein fehlerhaftes Testergebnis ausfindig zu machen.

Integrationstest

Wenn alle Komponententests erstellt sind und erfolgreich durchgeführt wurden ist es Zeit für die Integrationstests. Bei den Integrationstests werden dabei bereits fertige Komponenten „zusammengeschaltet“ und ihr Verhalten gegeneinander überprüft. Verhalten sie sich nicht korrekt, dann sind vermutlich Schnittstellen in der Kommunikation fehlerbehaftet. Es kann allerdings auch sein, dass einzelne Komponenten nicht korrekt arbeiten. Das bedeutet, dass der Testcase für diese Komponente ebenfalls falsch formuliert ist. Ein Beispiel hierfür ist der Verlust des Mars Climate Orbiters. Wenn dieser mit englischen Einheiten gerechnet hat, und auch die Unit Tests dieses Umrechnungsverhalten erwarten, dann ist auf Komponentenebene alles korrekt. Auf Integrationstestebene zeigt sich allerdings, dass die Umrechnungskomponente nicht korrekt arbeitet.

Da bei der Entwicklung eines Produktes nie alle Komponenten zum gleichen Zeitpunkt fertig werden benötigt man Strategien, um die für Integrationstests verantwortlichen Mitarbeiter sinnvoll beschäftigt zu halten. Dazu gibt es drei Vorgehen:

- Top-down Integration: hier beginnt man mit fertig gestellten Komponenten, die nicht von anderen aufgerufen werden, d. h. in der Aufrufhierarchie ganz oben stehen; die von ihr verwendeten, noch nicht fertig gestellten Komponenten werden mit Platzhaltern ersetzt
- Bottom-down Integration: hier wird mit den Komponenten begonnen, die keine anderen Komponenten aufrufen; übergeordnete, noch nicht fertig gestellte Komponenten müssen durch Testtreiber simuliert werden; des Weiteren zeigt das Beispiel des Mars Climate Orbiters, das ein „Vortesten“ der Basiskomponenten nicht zwangsläufig zu besseren Ergebnissen führt
- Big-Bang Integration: bei diesem Verfahren wird so lange gewartet, bis alle Komponenten vollständig implementiert sind; daraus ergibt sich, dass Mitarbeiter möglicherweise sehr lange untätig warten müssen; des Weiteren ist der verbleibende Zeitrahmen zum Testen sehr gering und ausfallende Mitarbeiter sowie andere unvorhersehbare Ereignisse führen möglicherweise zu einer Verschiebung des Release Termins

Systemtest

Der Systemtest unterscheidet sich vom Integrationstest dahingehend, dass er nicht darauf ausgelegt ist, funktionale Fehler in einem Produkt zu finden. Vielmehr geht es um nicht-funktionale Anforderungen wie Usability, Zuverlässigkeit des Produkts, wie effizient man damit arbeiten kann etc. Das System wird also aus der Perspektive des Kunden betrachtet.

Probleme können beim Systemtest dann auftreten, wenn Anforderungen in der Konstruktionsphase nicht genau definiert bzw. aufgeschrieben wurden. Es kann vorkommen, dass man sich auf etwas geeinigt hat, diese Vorstellung aber nur in den Köpfen der beteiligten Personen existiert und nicht in schriftlicher Form. Dann müssen Informationen über das Soll-Verhalten nachträglich zusammengetragen werden. Eine Gefahr dabei ist, dass sich die Vorstellungen der Personen im Laufe der Zeit verändert haben und sich u. U. nicht mehr miteinander vereinbaren lassen. Es ist deshalb sehr wichtig, ein Produkt von Anfang an so präzise wie möglich zu dokumentieren. Gerade wenn es um Kundenanforderungen geht ist das von besonderer Wichtigkeit, da eine Nichteinhaltung von Verträgen schnell zu juristischen Konsequenzen führen kann.

Abnahmetest

Der **Abnahmetest** schließlich ist wie bereits erwähnt ein Test, der vom Kunden durchgeführt wird. Diesen interessieren dabei die technischen Details der Implementierung nicht. Interessant ist höchstens, welche Dritttechnologien eingesetzt werden, beispielsweise in einem kryptographischen System (SSL, TLS, PFS, …). Der Kunde prüft, ob das System seinen zu Beginn festgelegten Anforderungen entspricht. Der Test findet dabei in einer Produktivumgebung statt. Das ist nicht zwingend das Produktivsystem des Kunden (denn wenn doch etwas schiefgehen sollte wäre das fahrlässig), aber eine dem System möglichst detailgetreu nachempfundene Umgebung.