Erstellung testbarer UML-Modelle zur Unterstützung der Automatisierung von Klassentests für objektorientierte Systeme

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers

Thomas Wierczoch

Berlin, den 26.03.2002

Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik
Prof. Dr.-Ing. Stefan Jähnichen
Franklinstr. 28/29
10587 Berlin

Betreuerin: Dehla Sokenou

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, die vorliegende Arbeit selbständig und eigenhändig angefertigt zu haben.

Berlin, den 26.03.2002 Thomas Wierczoch

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Objektorientierung nimmt innerhalb der Softwareentwicklung einen immer größeren Stellenwert ein. Gleichzeitig wächst die Komplexität der entwickelten Systeme sowie die Anforderungen hinsichtlich deren Sicherheit und Zuverlässigkeit.

Durch die ständig steigenden Qualitätsanforderungen an Software nimmt die Bedeutung des Tests innerhalb des Entwicklungsprozesses stetig zu. Um den Test bei gleichzeitig wachsender Komplexität der entwickelten Systeme zu einem effizienten und wirkungsvollen Mittel der Qualitätssicherung zu machen, wird dessen Automatisierung immer mehr zu einer Notwendigkeit. Eine Studie aus dem Jahr 1999 belegt, dass es bisher keine kommerziellen Werkzeuge gibt, die eine Automatisierung des Tests objektorientierter Systeme ausreichend unterstützen und eine konsequente Integration in den Entwicklungsprozess bieten [Jungmayr+99]. Dies wird in [Fraikin01] aktuell bestätigt.

Daraus ergeben sich verschiedene Probleme, sowohl für den Test als auch für den restlichen Entwicklungsprozess. Durch die mangelnde Integration des Tests entstehen z. B. viele Redundanzen zwischen den Informationen in Entwurfs- und Testwerkzeugen. Diese Redundanz sowie die zwangsläufig entstehende Verteilung von Informationen stellen ein nicht zu unterschätzendes Problem dar. Je stärker die Daten verteilt sind und je mehr Abhängigkeiten und Redundanzen zwischen diesen verteilten Daten existieren, umso mehr erhöht sich der Verwaltungsaufwand zur Einhaltung der Konsistenz der Daten. Die Konsistenz der dem Test zu Grunde gelegten Daten ist ein wesentlicher Faktor für die Qualität des Tests. Durch den erheblich größeren Arbeitsaufwand für die Einhaltung der Konsistenz bei redundanter und verteilter Datenhaltung müssen bei einem festgelegten finanziellen und zeitlichen Rahmen für den Test die eigentlichen Testaktivitäten entsprechend eingeschränkt werden. Außerdem erhöht sich die Wahrscheinlichkeit, dass die Konsistenz der Informationen nicht mehr vollständig gewährleistet werden kann. Dadurch wird die Aussagekraft des Tests erheblich reduziert und u. U. können Fehler in einem Testobjekt nicht mehr gefunden werden, die bei konsistenten Informationen entdeckt worden wären.

Eine konsequente Integration des Testprozesses in den restlichen Entwicklungsprozess reduziert eine redundante und verteilte Datenhaltung erheblich. Dadurch können die beschriebenen Probleme hinsichtlich der Konsistenz vermieden und somit die Qualität des Tests insgesamt deutlich erhöht werden.

In dieser Arbeit werden Techniken vorgestellt, welche die Erstellung testbarer UML-Modelle[1] ermöglichen, die als Grundlage für die Automatisierung des spezifikationsbasierten Klassentests verwendet werden können. Dabei wird für verschiedene Testmodelle und Testmuster für den spezifikationsbasierten Klassentest angegeben, welche Informationen in einem UML-Modell enthalten sein müssen und wie diese im UML-Modell darzustellen sind, um eine automatische Generierung ausführbarer Testfälle zu ermöglichen. Hierzu werden die verschiedenen von der UML zur Verfügung gestellten Diagrammarten und die in der UML enthaltene Spezifikationssprache OCL dahingehend untersucht, welche Konstrukte am Besten zur Darstellung der jeweiligen Informationen geeignet sind und es wird gezeigt, wie die Informationen in den jeweiligen Diagrammen integriert werden sollten.

Dabei ist es das Ziel, möglichst alle notwendigen Informationen im UML-Modell einzubinden, um den aufgezeigten Problemen, hinsichtlich der Redundanz und Konsistenz von Informationen, zu begegnen. Die Erstellung testbarer UML-Modelle bildet die Grundlage für eine in den objektorientierten Softwareentwicklungsprozess integrierte Automatisierung des Tests und stellt somit eine wesentliche Voraussetzung dar, um den steigenden Qualitätsanforderungen an Softwaresysteme bei gleichzeitig wachsender Komplexität gerecht zu werden.

In Kapitel 2 erfolgt eine Einführung in den Test objektorientierter Systeme. Es werden die verschiedenen Testaktivitäten erläutert sowie eine Einordnung von Fehlermodellen, Testmodellen und Testmustern in den Testprozess vorgenommen.

In Kapitel 3 wird die UML kurz vorgestellt. Dabei werden die zur Verfügung gestellten Diagrammarten, das der UML zu Grunde liegende Metamodell sowie die in die UML integrierte Spezifikationssprache OCL beschrieben. Für die einzelnen Aspekte wird deren Relevanz für den spezifikationsbasierten Klassentest untersucht.

Kapitel 4 beschreibt, wie ein testbares UML-Modell erstellt werden kann. Als Basis dafür dienen verschiedene Testmodelle und Testmuster. Für diese wird gezeigt, welche Informationen im Modell enthalten sein müssen, welche UML-Diagramme sich am besten zu ihrer Darstellung eignen und, falls notwendig, wie die OCL zu deren Spezifikation eingesetzt werden kann.

In Kapitel 5 wird die Anwendung des in dieser Arbeit erstellten Testprofils beispielhaft demonstriert.

Abschließend werden in Kapitel 6 die Ergebnisse zusammengefasst und ein Ausblick auf künftige Entwicklungen gegeben.

[1] Ein Modell wird als testbar bezeichnet, wenn man einen Algorithmus angeben kann, mit dessen Hilfe es möglich ist, einzig aus den in dem Modell enthaltenen Informationen, ausführbare Testfälle zu generieren [Binder99].

Kapitel 2

Test objektorientierter Systeme

2.1 Einführung

In den Anfangsjahren der Entwicklung objektorientierter Software wurde angenommen, dass die traditionellen Testverfahren, die sich für prozedurale Programmiersprachen bewährt hatten, auch für den Test objektorientierter Software eingesetzt werden können. Zusätzlich war man der Meinung, dass die der Objektorientierung zu Grunde liegenden Prinzipien der Vererbung und Datenkapselung und die damit verbundenen Möglichkeiten der Wiederverwendung von Softwarekomponenten den Aufwand für den Test erheblich reduzieren würden.

Inzwischen hat man erkannt, dass gerade die Datenkapselung, die Vererbung sowie auch die Polymorphie neue Probleme in Bezug auf den Test hervorbringen [Binder94a] [Binder99] [Jorgensen+94] [McGregor+01] [Liggesmeyer+96] [Pelkmann98]. Die Datenkapselung verschlechtert die Steuerbarkeit und Beobachtbarkeit des Verhaltens von Software, was die Testdurchführung und Testauswertung erheblich erschweren kann. Bei der Vererbung können Probleme beim Zusammenspiel von Methoden einer Klasse mit geerbten Methoden auftreten. Durch die Polymorphie kann mit wenigen Zeilen Programmtext ein sehr komplexes Laufzeitverhalten erzeugt werden, welches zu sehr schwer auffindbaren Fehlern führen kann. Um diesen veränderten Bedingungen gerecht zu werden, mussten traditionelle Testverfahren angepasst und neue Testverfahren speziell für den Test objektorientierter Software entwickelt werden.

Die beiden erwähnten Aspekte Steuerbarkeit und Beobachtbarkeit sind für den Test von erheblicher Bedeutung. Durch die Datenkapselung gilt dies insbesondere für den Test objektorientierter Systeme. Um ein Testobjekt systematisch testen zu können, ist es unbedingt erforderlich, dass man alle Testdaten[2] genau steuern und alle relevanten Ausgaben beobachten kann. Ist eine genaue Steuerung der Testeingaben nicht möglich, kann die Ursache für ein beobachtetes Verhalten nicht eindeutig festgestellt werden. Ist die Beobachtung des Verhaltens eines Testobjekts nicht möglich, kann nicht überprüft werden, wie bestimmte Eingaben durch das Testobjekt verarbeitet werden [Binder94] [Liggesmeyer+96].

Die Aufteilung des Tests in verschiedene Aktivitäten[3] wurde ebenfalls an die Gegebenheiten des objektorientierten Tests angepasst. Die grundsätzliche Orientierung der einzelnen Aktivitäten an der hierarchischen Struktur eines Systems ist aber erhalten geblieben. Für traditionelle Systeme wurde eine Aufteilung in die drei Aktivitäten Unittest,

[2] z.B. Eingabeparameter für eine Methode, Instanzvariablen der zu testenden Klasse, aber auch die entsprechenden Variablen der Testumgebung

[3] Der Begriff der Phase wird hier bewusst nicht verwendet, da der Testprozess kein stetiger, sondern ein iterativer Prozess ist. Dieser Umstand kann durch den Begriff der Aktivität besser dargestellt werden.

Integrationstest und Systemtest vorgenommen, um den Test auf den unterschiedlichen Ebenen eines Systems wiederzuspiegeln. Für den Test objektorientierter Software finden sich in der Literatur diese drei Aktivitäten mit einigen Veränderungen wieder.

Der Unittest beschäftigt sich danach, sowohl für traditionelle als auch für objektorientierte Software, mit dem Test der kleinsten unabhängig testbaren Einheit eines Systems. Für ein prozedurales System ist dies die Prozedur. Für objektorientierte Anwendungen werden in der Literatur allerdings unterschiedliche Auffassungen vertreten, welches Element diese kleinste Einheit darstellt. Viele Autoren sehen die Klasse als diese kleinste Einheit an [Liggesmeyer+96] [Beizer90] [McGregor+94] [Rüppel97]. Dagegen wird in [Jorgensen+94] die Meinung vertreten, dass die Methoden einer Klasse diese kleinsten Einheiten darstellen.

Der Integrationstest beschäftigt sich mit dem korrekten Zusammenwirken verschiedener Komponenten eines Systems. Für objektorientierte Systeme gibt es über diese Aktivität ebenfalls verschiedene Ansichten. Liggesmeyer et al. vertreten in [Liggesmeyer+96] die Auffassung, dass sich diese Aktivität relativ klar von den anderen beiden Aktivitäten abgrenzen lässt. Allerdings ist die Verwendung des Begriffs Integrationstest als einer klar abgegrenzten Aktivität für den Test objektorientierter Systeme problematisch. In [Binder99] und [McGregor+01] wird der Integrationstest eher als ein kontinuierlicher Prozess betrachtet, der sich über den gesamten Testprozess erstreckt. Barbey geht in [Barbey97] sogar soweit, zu sagen, dass es für objektorientierte Systeme gar keinen Klassentest gibt, sondern es sich dabei um viele verschiedene Formen des Integrationstests handelt.

Auf der obersten Ebene eines Systems ist der Systemtest angesiedelt. Hiermit werden Anforderungen überprüft, die das Verhalten eines Systems an den Schnittstellen zu seiner Umgebung festlegen. Für diese Aktivität gibt es keine großen Unterschiede zwischen dem Test traditioneller und objektorientierter Software. Lediglich die zu Grunde gelegten Spezifikationen unterscheiden sich. Darüber herrscht in der Literatur auch weitgehend Einigkeit.

In dieser Arbeit wird in Bezug auf die Testaktivitäten eine etwas veränderte Terminologie verwendet, die eine eindeutige Aufteilung der Testaktivitäten zulässt. Dazu wird der Testprozess in vertikale und horizontale Testaktivitäten unterteilt. Die vertikalen Aktivitäten untersuchen das Softwaresystem auf unterschiedlichen Abstraktionsebenen. Hier erfolgt die Aufteilung in die einzelnen Aktivitäten Klassentest, Teilsystemtest, Systemtest und die Integrationsstrategien. Für den spezifikationsbasierten Klassentest, der Gegenstand dieser Arbeit ist, wird nicht die Klasse als die kleinste unabhängig testbare Einheit betrachtet, sondern die Methode. Dennoch wird der Begriff Klassentest weiter beibehalten. Innerhalb dieser Aktivität wird jedoch eine differenziertere Sicht eingenommen, welche die Methode als eigenständig testbares Element berücksichtigt[4]. Der Begriff des Integrationstests wird hier nicht verwendet, da mit diesem Begriff eine Vermischung von zwei verschiedenen Aktivitäten erfolgt, die sich durch ihre unterschiedlichen Ziele und Strukturen nicht vereinbaren lassen, dem Teilsystemtest und den Integrationsstrategien. Während sich der Teilsystemtest ganz klar in die hierarchische Struktur eines Systems einordnen lässt, ist dies für die Integrationsstrategien nicht möglich. Dabei handelt es sich um die konsequente Anwendung der Erkenntnis, dass die Integration von Elementen eines Systems eigentlich ein über den gesamten Testprozess andauernder Prozess ist.

[4] Siehe Abschnitt 2.2.1

Die horizontalen Aktivitäten teilen den Testprozess entlang einer Zeitachse auf. Sie werden für jede vertikale Testaktivität in der folgenden Reihenfolge durchgeführt:

(1) Testobjektauswahl,

(2) Testfallermittlung,

(3) Testdatenerzeugung,

(4) Sollwertbestimmung,

(5) Testdurchführung,

(6) Testauswertung und

(7) Regressionstest.

Der Zusammenhang zwischen den vertikalen Testaktivitäten, inkl. der Integrationsstrategien, und den horizontalen Testaktivitäten ist in Abbildung 2.1 dargestellt.

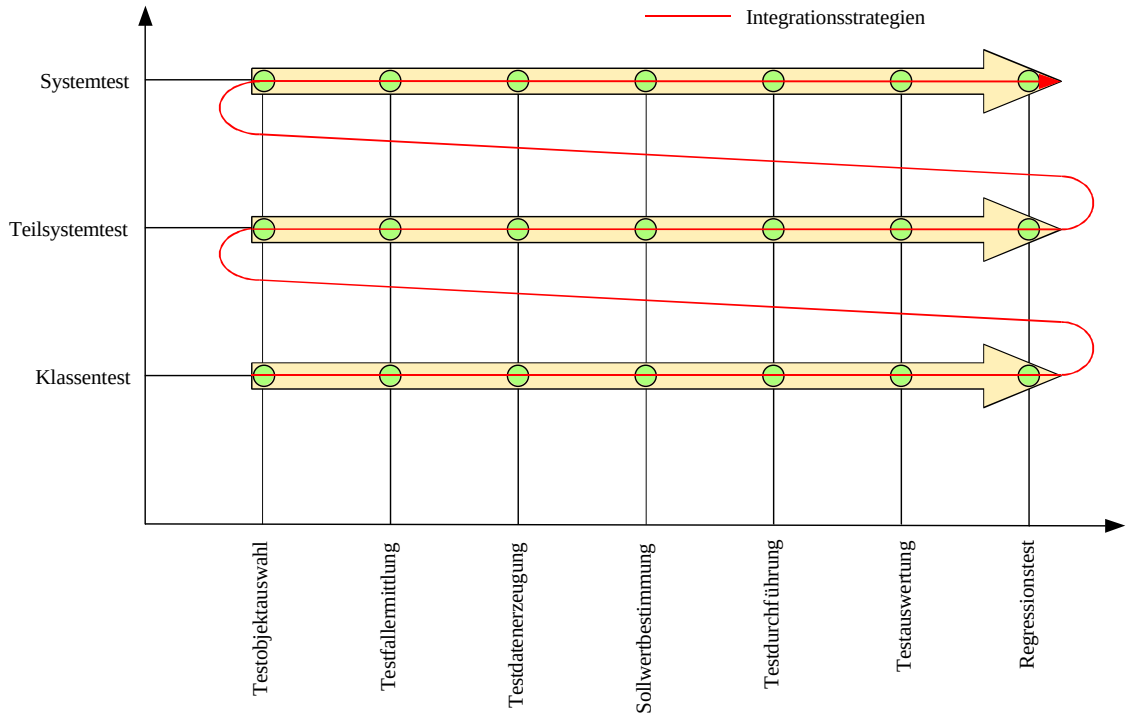


Abbildung 2.1 *Horizontale und vertikale Testaktivitäten*

Dabei ist zu beachten, dass dieser Prozess in der Realität nicht so strikt sequenziell abläuft, wie dargestellt. Zwischen den einzelnen Aktivitäten kann es sehr wohl zu Überschneidungen und Rückkopplungen kommen. Der Testprozess verläuft also iterativ und inkrementell. Eine genaue Beschreibung der einzelnen Testaktivitäten wird in den Abschnitten 2.2 und 2.3 gegeben.

2.2 Vertikale Testaktivitäten

Die vertikalen Testaktivitäten teilen den Testprozess entsprechend den verschiedenen Abstraktionsebenen eines Softwaresystems auf. Nach dem Klassentest auf der untersten Ebene folgt der Teilsystemtest und auf der obersten Ebene der Systemtest. Die Integrationsstrategien nehmen eine Sonderstellung ein. Es gibt für jede der vertikalen Aktivitäten Integrationsstrategien und selbst die Aufteilung in die vertikalen Aktivitäten stellt bereits eine solche Strategie dar.

2.2.1 Klassentest

Der Klassentest beschäftigt sich mit dem Test einer einzigen Klasse. Dabei werden alle von der Klasse selbst definierten und alle geerbten Merkmale[5] mit einbezogen [Liggesmeyer+96] [Binder99]. Das heißt, dass die Vererbungsbeziehungen mit Gegenstand des Klassentests sind. Allerdings wird in dieser Arbeit zumindest für nicht-öffentliche Methoden nicht die in der Literatur weit verbreitete Meinung vertreten, dass die Klasse die kleinste unabhängig testbare Einheit eines objektorientierten Systems ist [Binder94a] [Liggesmeyer+96] [Rüppel97]. Begründet wurde diese Ansicht u.a. damit, dass die einzelnen Methoden einer Klasse durch die gemeinsame Benutzung der Attribute der Klasse und durch gegenseitige Aufrufe nicht unabhängig voneinander getestet werden können. Für imperativ umgesetzte Systeme können solche Abhängigkeiten genauso bestehen. Dabei entsprechen z. B. die globalen Variablen eines Moduls den Attributen einer Klasse und gegenseitige Aufrufe zwischen den Prozeduren eines Moduls sind durchaus üblich und auch notwendig.

Ein weiteres Argument gegen die Klasse als kleinster unabhängig testbarer Einheit ist, dass durch den Test auf der Ebene der Klasse allein zumindest die nicht-öffentlichen Methoden nicht ausreichend getestet werden können [Harrold+92] [Binder99]. Begründet werden kann dies mit dem **Antidekompositionsaxiom** von Weyuker [Weyuker88]. Dieses Axiom ist in einer Menge von Axiomen enthalten, die die Angemessenheit von Testfallmengen für Programme und Komponenten beschreibt, und besagt Folgendes:

> Eine Testfallmenge, welche die Funktionalität eines Programms überdeckt, erreicht dies nicht notwendigerweise auch für die enthaltenen Komponenten.

Das die Komponenten einschließende Programm muss nicht die gesamte Funktionalität seiner enthaltenden Komponenten ausführen, so dass es gar nicht möglich ist, die Komponenten mit diesem Programm als Kontext ausreichend zu testen. Obwohl die Axiome von Weyuker ursprünglich für strukturelle Testverfahren entwickelt wurden, können sie auch für objektorientierte Testverfahren verwendet werden [Perry+90]. Das Antidekompositionsaxiom kann auf die unterschiedlichen Ebenen[6] eines objektorientierten Programms angewendet werden. Für das Verhältnis zwischen einer Klasse und den in ihr definierten Methoden bedeutet dies, dass die Funktionalität der einzelnen Methoden nicht ausreichend durch den Test der durch die Klasse zur Verfügung gestellten Funktionalität getestet werden kann. Dies gilt zumindest für die nicht-öffentlichen Methoden, die nur indirekt über die öffentlichen Methoden aufgerufen werden. Hier spielt das in Abschnitt 2.1 erwähnte Problem der Steuerbarkeit eine wesentliche Rolle.

Ein wichtiger Aspekt beim Test ist immer die Unterscheidung zwischen dem Testobjekt und der lediglich für die Durchführung des Tests notwendigen Testumgebung. Auch wenn für den Test einer einzelnen Methode immer ein Objekt der entsprechenden Klasse erzeugt werden muss, bleibt die Methode das Testobjekt. Der Rest der Klasse gehört zur Testumgebung. Wichtig für den unabhängigen Test einer Methode ist die genaue Spezifikation ihres Verhaltens mit Hilfe von Vor- und Nachbedingungen. Mit einer solchen Grundlage ist es möglich, eine Methode unabhängig davon zu testen, ob andere Methoden der Klasse die gleichen Attribute verändern. Überprüft wird beim Test einer

[5] Als Merkmale werden alle Attribute und Methoden einer Klasse bezeichnet. In der UML-Spezifikation werden diese von der Metaklasse `Feature` abgeleitet.
[6] Klassen vs. Methoden, Teilsystem vs. Klassen, System vs. Teilsysteme

Methode lediglich, ob die Methode bei Einhaltung der Vorbedingungen die Nachbedingungen einhält bzw. deren Verhalten bei einem Aufruf mit nicht eingehaltenen Vorbedingungen. Nicht getestet wird die gemeinsame Benutzung von Attributen. Dies wird auf der Ebene der Klassenfunktionalität überprüft.

In [Harrold+92] wird vorgeschlagen, jede Methode isoliert zu testen, also Stellvertretermethoden für die aufgerufenen Methoden zu implementieren. Das ist allerdings mit einem sehr hohen, in den meisten Fällen nicht vertretbaren, Aufwand verbunden. Binder geht in [Binder99] einen anderen Weg. Er schlägt eine Integrationsstrategie vor, mit deren Hilfe die Reihenfolge der Methodentests so festgelegt wird, dass die Anzahl der zu erstellenden Stellvertretermethoden stark verringert werden kann. Bereits getestete Methoden gelten dabei als hinreichend vertrauenswürdig und können für folgende Tests an Stelle von Stellvertretermethoden verwendet werden. Dazu sollen die Methoden in der folgenden Reihenfolge getestet werden:

(1) Konstruktoren

(2) lesende Methoden (Accessor Methods)

(3) Prädikate (Boolean Methods)

(4) schreibende Methoden (Modifier Methods)

(5) Iteratoren

(6) Destruktoren

Innerhalb dieser Methodengruppen sollen die Methoden entsprechend ihres Gültigkeitsbereiches überprüft werden. Zuerst werden die Methoden getestet, deren Gültigkeitsbereich auf die Klasse selbst beschränkt ist. Dann folgen die Methoden, auf die in der Klasse und in allen von dieser Klasse erbenden Klassen zugegriffen werden kann, und als letztes die Methoden ohne Zugriffsbeschränkungen[7]. Bei dieser Vorgehensweise spielen zwei Aspekte eine wesentliche Rolle: die Komplexität der einzelnen Methoden und die Benutzungsrelationen zwischen den Methoden. Allerdings kann auch diese Vorgehensweise das Problem der Abhängigkeiten zwischen verschiedenen Methoden nur eingeschränkt lösen. Diese Problematik wird in Kapitel 4 ausführlich behandelt.

Ein weiteres von Weyuker entwickeltes Axiom ist das **Antikompositionsaxiom**:

> Der adäquate, isolierte Test aller Komponenten eines Programms reicht nicht für den adäquaten Test des Programms selbst aus.

Auch dieses Axiom kann, wie das Antidekompositionsaxiom, auf die unterschiedlichen Ebenen eines objektorientierten Programms angewendet werden. Für den Klassentest bedeutet dies, dass der adäquate Test aller Methoden in Isolation keine hinreichende Bedingung für den adäquaten Test der Klasse selbst ist [Harrold+92] [Binder99].

Deshalb wird sowohl von Binder als auch von Harrold et al. neben dem Methodentest der Intra-Klassentest[8] eingeführt. Mit Hilfe des Intra-Klassentests sollen Fehler aufgedeckt werden, die durch die gemeinsame Nutzung von Instanzvariablen entstehen

[7] Die Methoden werden in einigen objektorientierten Programmiersprachen mit den Schlüsselwörtern `private`, `protected` und `public` deklariert.

[8] Binder spricht hier nur vom Klassentest. Um eine Verwechslung mit dem übergeordneten Begriff Klassentest zu vermeiden, wurde der Begriff Intra-Klassentest verwendet. In [Harrold+92] wird dieser Begriff explizit verwendet.

können. Hierbei wird also das korrekte Zusammenspiel der einzelnen Methoden überprüft. Ein Testfall besteht dabei häufig aus einer Nachrichtensequenz, die Nachrichten an verschiedene Methoden der zu testenden Klasse enthält. Binder bietet sowohl für den Methodentest als auch für den Intra-Klassentest verschiedene Testmuster inklusive verschiedener Integrationsstrategien an.

Ein weiterer Aspekt, der für den Klassentest von wesentlicher Bedeutung ist, ist die Vererbung. Eine Klasse wird immer aus den Merkmalen zusammengesetzt, die sie selbst definiert, und aus denen, die sie erbt[9]. Daraus ergeben sich Konsequenzen, die nur bei objektorientierten Systemen zu finden sind. So können z. B. Fehler aus der Oberklasse in die von ihr erbenden Unterklassen propagiert werden. Genauso ist es möglich, dass bei der Verwendung von geerbten Methoden im Kontext einer Unterklasse Fehler auftreten, die bei der Verwendung im Kontext der entsprechenden Oberklasse nicht aufgetreten sind. Dieses sind nur zwei Probleme, die im Zusammenhang mit der Vererbung auftreten können. Sie deuten allerdings schon die Komplexität dieses Beziehungstyps und die daraus für den Test entstehenden Konsequenzen an. Eine ausführlichere Beschreibung von Fehlern, die durch Vererbung entstehen können, ist in [Binder99] zu finden.

Die durch die Vererbung verfügbaren Möglichkeiten können aber auch dazu genutzt werden, den Test von Vererbungshierarchien zu vereinfachen. So können Testfälle einer Oberklasse unter bestimmten Umständen beim Test der geerbten Methoden in der Unterklasse wiederverwendet werden. Manchmal besteht sogar die Möglichkeit, dass die Testfälle der Oberklasse für den Test einer Unterklasse nicht erneut ausgeführt werden müssen. Welche Voraussetzungen für diese effizientere Gestaltung des Klassentests erfüllt werden müssen, wird in Kapitel 4 beschrieben.

Polymorphismus ist die Möglichkeit der Bindung einer Referenz auf unterschiedliche Elemente an ein Objekt [Binder99]. Es gibt zwei Formen des Polymorphismus, den statischen und den dynamischen Polymorphismus[10]. Beim statischen Polymorphismus erfolgt die Bindung zur Übersetzungszeit. Ein Beispiel hierfür sind C++-Templates. Beim dynamischen Polymorphismus wird die Bindung, und somit auch die Typüberprüfung, zur Laufzeit vorgenommen. Mit dieser Form des Polymorphismus wird z. B. eine Methode ermittelt, die an eine Nachricht gebunden wird.

Der auf der Vererbung basierende Polymorphismus stellt Möglichkeiten zur generischen, und damit sehr flexiblen Gestaltung von Klienten in einer Klienten/Server-Beziehung zur Verfügung. Dadurch kann ein Klient relativ unabhängig von seinen Servern gestaltet werden und ist damit besser zur Wiederverwendung geeignet. Aus dieser Flexibilität entstehen allerdings auch Probleme. Ist ein polymorpher Klient für eine Anwendung mit allen dort auftretenden Bindungen getestet worden, kann ein hinreichendes Vertrauen in dessen Korrektheit hinsichtlich der Zusammenarbeit mit den verwendeten Servern erreicht werden. An den Servern können zwar Änderungen unabhängig vom Klienten vorgenommen werden. Ein erneuter Test des Klienten mit den geänderten Serverklassen wird allerdings erforderlich. Wird die Klassenhierarchie der Server um neue Unterklassen erweitert, muss der Klient ebenfalls mit diesen neuen potentiellen Serverklassen getestet werden [Binder99].

[9] Eine Ausnahme bilden die Wurzelklassen, die die oberste Klasse in ihrer jeweiligen Vererbungshierarchie sind und keine Superklasse besitzen.

[10] Vgl. [Binder99], McGregor et al. verwenden in [McGregor+01] die Begriffe *Parametric Polymorphism* und *Inclusion Polymorphism.*

Ein weiteres Problem entsteht, wenn die Serverklassenhierarchie nicht sorgfältig entworfen wurde. Bei der Verwendung der einzelnen Serverklassen durch einen polymorphen Klienten kann es zu Fehlern kommen.

Durch die Verwendung von abstrakten Klassen ist es möglich, eine Schnittstelle zu definieren, ohne dabei eine konkrete Implementierung festzulegen. Da für abstrakte Klassen keine Objekte erzeugt werden können, ist es nicht möglich, diese direkt zu testen. Für den Test einer abstrakten Klasse muss eine konkrete Klasse erzeugt werden, welche die in der abstrakten Klasse definierte Schnittstelle implementiert. Ähnliches gilt für generische Klassen. Um diese testen zu können, müssen die Typparameter festgelegt und eine entsprechende Instanz erzeugt werden.

2.2.2 Teilsystemtest

Gegenstand des Teilsystemtests sind die Komponenten eines Systems, die nicht durch den Klassentest erfasst werden und die nur einen Teil der Funktionalität eines Gesamtsystems zur Verfügung stellen [Binder99]. Ein Teilsystem ist „eine unabhängige Gruppe von Klassen, die zusammenarbeiten, um eine bestimmte Menge von Aufgaben zu erfüllen.“ [Gamma+98] Unabhängig bedeutet, dass das Teilsystem als Ganzes ausführbar und testbar sein muss [Binder99]. Solche Komponenten können aus einer Menge zusammengehöriger Klassen[11] bestehen oder aber auch aus einzelnen Teilsystemen zusammengesetzt sein. Mit dem Teilsystemtest erfolgt die „Überprüfung des korrekten Zusammenwirkens von dienstanbietenden und dienstnutzenden Objekten unterschiedlicher Klassen, die nicht in einer Vererbungsbeziehung stehen. Die Integration einzelner Methoden einer Klasse sowie ihrer Oberklassen ist bereits Aufgabe des Klassentests.“ [Liggesmeyer+96]

Der Teilsystemtest beschäftigt sich mit Beziehungen, die nur für das ganze Teilsystem gelten, nicht aber für einzelne Komponenten des Teilsystems. Dazu zählen z. B. die korrekte Implementierung der Beziehungen zwischen Klassen, die in einem Klassendiagramm modelliert worden sind[12]. Ein weiterer Aspekt des Teilsystemtests ist die Überprüfung von Einschränkungen hinsichtlich erlaubter Nachrichtensequenzen für ein modales Teilsystem. Die Problematik verschiedener Modalitäten wird für die Ebene einzelner Klassen in Abschnitt 4.5.1 erörtert. Die entsprechende Definition ist auf die Ebene der Teilsysteme übertragbar.

Der erfolgreiche Teilsystemtest ist die Voraussetzung für die Integration der Teilsysteme zu einem Gesamtsystem und somit auch für den Systemtest.

2.2.3 Systemtest

Der Systemtest untersucht das Verhalten des gesamten Systems in Bezug auf die an das System gestellten Anforderungen. Die Anforderungen spezifizieren die an der Schnittstelle des Systems beobachtbaren Reaktionen auf Eingaben aus der Umgebung des Systems. Es gibt verschiedene Formen von Anforderungen: funktionale Anforderungen, Leistungsanforderungen, Kompatibilitätsanforderungen und Anforderungen an die Benutzungsschnittstelle. Dementsprechend gibt es verschiedene Testarten auf der Ebene des Systemtests: Funktionstest, Leistungs- und Stresstest, Kompatibilitätsüberprüfung und Überprüfung der Benutzungsschnittstelle.

---

[11] sog. Klassencluster [Binder99]

[12] Assoziationen, Aggregationen, Multiplizitäten, Teil-Ganzes-Beziehungen, Ist-Ein-Beziehungen

Beim funktionalen Test wird überprüft, ob alle in den funktionalen Anforderungen festgelegten Funktionen im System realisiert wurden und die Realisierung der Spezifikation dieser Funktionen entspricht. Mit dem Leistungstest soll untersucht werden, ob das System die in den Anforderungen festgelegten Mengen von Daten in der geforderten Zeit verarbeiten kann. Der Stresstest überprüft die Reaktion des Systems bei Überschreitung der Grenze der verarbeitbaren Datenmenge bei gleichzeitigem Entzug von Systemressourcen. Dies ist besonders wichtig bei sicherheitskritischen Anwendungen. Bei der Überprüfung der Kompatibilität wird die Verträglichkeit des Systems hinsichtlich verschiedener Kombinationen von Umgebungsbedingungen getestet. Dazu zählen bereits vorhandene Anwendungen, verschiedene Betriebssysteme bzw. Betriebssystemversionen, verschiedene Hardwarekonfigurationen[13] und Datenbanksysteme.

Da die Anforderungsdefinitionen auf dieser Abstraktionsebene unabhängig von dem zur Realisierung verwendeten Programmierparadigma sind, besteht grundsätzlich kein Unterschied bei den Teststrategien gegenüber dem Test traditioneller Systeme. Beachtet werden muss allerdings, dass die Spezifikation der funktionalen Anforderungen für objektorientierte Systeme in einer anderen Form erfolgt, als die konventioneller Systeme. Für objektorientierte Systeme werden die funktionalen Anforderungen mit Hilfe von Anwendungsfällen spezifiziert. Bei den anderen Anforderungsarten hingegen besteht kein Unterschied in der Repräsentation zwischen objektorientierten und traditionellen Systemen.

2.2.4 Integrationsstrategien

Die Integrationsstrategien besitzen eine besondere Stellung im Testprozess. Grundsätzlich lassen sie sich nicht in die an der hierarchischen Struktur eines objektorientierten Systems ausgerichteten vertikalen Testaktivitäten einordnen. Vielmehr beschäftigen sich Integrationsstrategien mit verschiedenen Modellen von Abhängigkeiten zwischen einzelnen Komponenten auf allen unterschiedlichen Abstraktionsebenen eines Systems [Binder99] [Winter00]. Ziel der Integrationsstrategien ist die Festlegung einer Reihenfolge, in der die Komponenten eines Systems zu testen sind. Durch Integrationsstrategien selbst werden keine Testfälle erzeugt. Die Einteilung in die drei vertikalen Aktivitäten Klassentest, Teilsystemtest und Systemtest stellt selbst eine Integrationsstrategie auf oberster Ebene dar. Dadurch wird festgelegt, dass ein System in Bezug auf seine Struktur *Bottom-up* zu testen ist, also von den kleinsten Einheiten, über die Teilsysteme, bis hin zum Gesamtsystem.

Unterhalb dieser Integrationsstrategie werden in und zwischen den einzelnen Aktivitäten ebenfalls wieder Integrationsstrategien angewendet, um die Testreihenfolgen auf den unterschiedlichen Ebenen des Systems festzulegen. Die Strategien, die sich mit der Integration innerhalb einer der vertikalen Aktivitäten befassen, legen die Reihenfolgen fest, in der die in der jeweiligen Abstraktionsebene als Testobjekte festgelegten Elemente zu testen sind. Für den Klassentest bedeutet dies sowohl die Auswahl der Reihenfolge, in der die Klassen grundsätzlich zu testen sind[14], als auch für jede Klasse, in welcher Reihenfolge deren Methoden geprüft werden sollen.

Die Integrationsstrategien, die zwischen den Abstraktionsebenen eines Systems wirken, legen die Reihenfolge fest, in der Elemente einer Abstraktionsebene zu einem Element

[13] unterschiedliche Prozessoren, Festplatten, Grafikkarten, etc.

[14] z.B. durch eine vererbungsbezogene Strategie, durch die die Klassendiagramme von den möglicherweise abstrakten Klassen an den Wurzeln der jeweiligen Vererbungshierarchie hin zu den konkreten Klassen an den Blättern getestet werden [Winter00].

der nächst höheren Ebene zusammengesetzt werden. So wird z. B. damit festgelegt, in welcher Reihenfolge Klassen in ein Teilsystem zu integrieren sind. In diesem Zusammenhang beschränkt sich die Sicht der Abstraktionsebenen eines Systems nicht nur auf die drei Ebenen Klasse, Teilsystem und Gesamtsystem. Auch innerhalb dieser drei Ebenen gibt es noch Abstufungen. So realisieren verschiedene Methoden einer Klasse gemeinsam eine von der Klasse zur Verfügung gestellte Funktionalität oder einzelne Teilsysteme können wiederum zu einem größeren Teilsystem kombiniert werden. In Tabelle 2.1 wird dies für die verschiedenen Ebenen eines objektorientierten Systems dargestellt. Daraus wird ersichtlich, dass die Integration bereits auf Klassenebene beginnt und erst mit der Einbettung des Systems in seine Arbeitsumgebung endet.

Komponente	System
Methode	Klasse
Klasse	Cluster
Cluster	Teilsystem
Teilsystem	System
System	Umgebung

Tabelle 2.1 *Ebenen des Integrationstests*

Die Auswahl einer Integrationsstrategie und die damit verbundene Festlegung einer Testreihenfolge hat einen großen Einfluss auf die Notwendigkeit der Erstellung von Stellvertreterkomponenten. Stellvertreterkomponenten müssen immer dann eingesetzt werden, wenn ein für den Test einer anderen Komponente benötigtes Element noch nicht fertig implementiert oder noch nicht getestet worden ist. Die Stellvertreterkomponente muss dann das Verhalten der durch sie ersetzten Komponente simulieren. Da die Erstellung solcher Stellvertreter mit einem sehr hohen Aufwand verbunden ist, soll in den meisten Fällen die Notwendigkeit für deren Einsatz möglichst minimiert werden. Overbeck gibt in [Overbeck94] eine Integrationsstrategie zur Ermittlung von Testreihenfolgen auf der Basis von Klassendiagrammen an, bei der die Notwendigkeit zur Erstellung von Stellvertreterkomponenten minimiert wird. Für die Auflösung von zyklischen Beziehungen bleibt der Einsatz von Stellvertretern aber unvermeidbar. Ein anderer Ansatz für eine Integrationsstrategie, die eine Minimierung von zu erstellenden Stellvertreterkomponenten gewährleistet, wird in [Kung+95] vorgestellt. Dieser basiert auf der Analyse von C++ Quellcode. In [Binder99] werden verschiedene Testmuster für die Integration vorgestellt, von denen die meisten unabhängig von der Abstraktionsebene der zu integrierenden Komponenten sind. Diese Muster beruhen lediglich auf den Beziehungen zwischen den zu integrierenden Komponenten und sind deshalb auf den verschiedenen Stufen eines Systems einsetzbar[15].

Der Einsatz von Integrationsstrategien kann sehr weit reichende Rückwirkungen auf die Gestaltung des gesamten Entwicklungsprozesses haben. Soll z. B. für den Klassentest eine *Bottom-up*-Integrationsstrategie angewendet werden, ist es sinnvoll, die Methoden der Klassen in einer Reihenfolge zu implementieren, welche die Abhängigkeiten zwischen den Methoden widerspiegelt und der Reihenfolge des Tests der Methoden entspricht. So kann mit dem Test der Methoden sehr frühzeitig begonnen werden, was die Fehlerauffindung erleichtert und beschleunigt und somit zu einer Effizienzsteigerung

[15] siehe Tabelle 2.1

und Kostensenkung führt. Dies setzt natürlich eine sehr gute Integration des Tests in den restlichen Entwicklungsprozess voraus.

2.3 Horizontale Testaktivitäten

Die horizontalen Testaktivitäten teilen jede vertikale Aktivität in einzelne Testaktivitäten ein, die notwendig sind um einen Test durchführen zu können. Es gibt die folgenden horizontalen Testaktivitäten:

(1) Testobjektauswahl

(2) Testfallermittlung

(3) Testdatenerzeugung

(4) Sollwertbestimmung

(5) Testdurchführung

(6) Testauswertung

(7) Regressionstest

Diese Einteilung ist unabhängig vom verwendeten Programmierparadigma, gilt also sowohl für den Test prozeduraler wie auch objektorientierter Programme. Da die einzelnen Aktivitäten voneinander abhängen, werden sie immer in dieser Reihenfolge durchgeführt. In bestimmten Fällen kann es dabei zu Rückkopplungen kommen. Dies geschieht genau dann, wenn während der Testauswertung festgestellt wird, dass nicht alle Aspekte eines Testobjekts überprüft wurden. Dann müssen entweder bestehende Testfälle verändert oder neue Testfälle entwickelt werden. Anschließend ist das Testobjekt mit diesen Testfällen zu testen.

2.3.1 Testobjektauswahl

Durch die Testobjektauswahl werden die Komponenten festgelegt, die getestet werden sollen. Die Entscheidung darüber wird durch die zur Verfügung stehenden Mittel[16] und den zu veranschlagenden Aufwand für eine Komponente beeinflusst. Ein weiterer Aspekt, der hierbei eine wesentliche Rolle spielt, stellen die Eigenschaften einer Komponente dar, die die Notwendigkeit eines Tests bestimmen. Sicherheitskritische Komponenten oder solche, die bei Fehlverhalten einen großen materiellen Schaden anrichten können, sollten in jedem Fall getestet werden. Das jeweils ausgewählte Testobjekt bildet den Kontext für die folgenden horizontalen Testaktivitäten.

2.3.2 Testfallermittlung

Während der Testfallermittlung wird festgelegt, welche Aspekte eines Testobjekts getestet werden sollen. Die ermittelten Testfälle bestimmen den Umfang und die Qualität des Tests. Es ist durchaus möglich, einen sehr umfangreichen Test mit vielen Testfällen zu erzeugen, ohne eine ausreichende Überprüfung des Testobjekts zu erreichen. Deshalb kommt der sorgfältigen Auswahl der Testfälle eine besondere Bedeutung innerhalb des Testprozesses zu.

Beim Klassentest besteht ein Testfall aus einer Nachrichtensequenz und abstrakten Wertebeschreibungen für den Zustand des Testobjekts und seiner Umgebung und den

[16] Zeit, finanzielle Mittel

Parametern der in der Nachrichtensequenz enthaltenen Nachrichten. Da ein vollständiger Test nur für triviale Testobjekte möglich ist, muss man sich in den meisten Fällen auf ausgewählte Testfälle beschränken. Hierbei spielt der Begriff der Äquivalenzklasse eine wesentliche Rolle. Dazu wird der Testdatenraum durch Äquivalenzklassenbildung[17] in verschiedene Partitionen, den sog. Äquivalenzklassen, aufgeteilt. Eine **Äquivalenzklasse** ist nach [Myers79] ein Teil des Testdatenraums, der so definiert wurde, dass, wenn irgendeines der Elemente einer Äquivalenzklasse einen Test besteht, davon ausgegangen werden kann, dass dies auch für alle anderen Elemente dieser Äquivalenzklasse gilt. Diese Einteilung hängt auch eng mit dem Begriff der **funktionalen Äquivalenz** zusammen, wobei hier davon ausgegangen wird, dass alle Elemente einer Äquivalenzklasse bei gleichem Ausgangszustand des Testobjekts ein identisches Verhalten hervorrufen[18]. In beiden Fällen erfolgt eine Zusammenfassung von mehreren Elementen des Testdatenraumes zu abstrakten Wertebeschreibungen, die eine effiziente Testfallermittlung ermöglichen.

Als Basis für eine systematische Testfallermittlung dienen Fehlermodelle, Testmodelle und Testmuster [Binder99]. Beim funktionalen Test dient zusätzlich die Spezifikation[19] des Testobjekts als Grundlage für die Testfallermittlung, beim strukturellen Test die Implementierung.

Ein **Fehlermodell** ist eine Annahme darüber, an welchen Stellen und unter welchen Umständen mit einer großen Wahrscheinlichkeit Fehler zu finden sind. **Testmodelle** sind Darstellungen der Implementierung der zu testenden Komponenten, die genug Informationen enthalten, um daraus ausführbare Testfälle erzeugen zu können. UML-Modelle sind eine solche Repräsentation der Implementierung, die aber in den meisten Fällen nicht genügend Informationen enthalten, um als Testmodelle fungieren zu können. Neben den Informationen für die Testfallerzeugung müsste ein Modell auch ausreichende Informationen für die restlichen horizontalen Aktivitäten enthalten. Die Darstellung der Vorgehensweise zur Erstellung testbarer UML-Modelle für den funktionalen Klassentest wird in Kapitel 4 behandelt. Ein **Testmuster** stellt eine allgemeine Lösung für ein spezifisches, häufig auftretendes Testproblem zur Verfügung, welches durch ein Fehlermodell beschrieben wird [Binder99]. Diese Problematik wird in Abschnitt 2.4 näher beschrieben.

2.3.3 Testdatenerzeugung

In den während der Testfallermittlung erzeugten Testfallspezifikationen enthaltene abstrakte Wertebeschreibungen werden durch die Testdatenerzeugung in primitive Wertebeschreibungen umgesetzt. Der Grund für die Erzeugung abstrakter Wertebeschreibungen durch die Testfallermittlung wurde bereits in Abschnitt 2.3.2 beschrieben. Abstrakte Wertebeschreibungen reichen für die Erzeugung ausführbarer Testfälle nicht aus, da ein Programm konkrete Daten erwartet. Für eine abstrakte Wertebeschreibung kann mehr als eine primitive Wertebeschreibung des entsprechenden Typs angegeben werden, die den Bedingungen der abstrakten Wertebeschreibung genügt. Für primitive Datentypen besteht eine abstrakte Wertebeschreibung aus der Angabe eines Wertebereichs, für Klassen aus komplexen Zustandsbeschreibungen, wie sie durch entsprechende Zustandsinvarianten in dem der Klasse zugeordneten Zustandsautomaten definiert sind.

[17] siehe [Myers79]

[18] z.B. Aktivierung des gleichen Zustandsübergangs mit Ausführung der gleichen Aktionen in einem Zustandsautomaten

[19] Enthält die Spezifikation des Testobjekts genügend Informationen für die Testfallerzeugung, kann sie als Testmodell verwendet werden [Binder99].

Die Aufgabe der Testdatenerzeugung besteht also in der Generierung von konkreten Daten für Parameter und Attribute, die den Bedingungen der abstrakten Wertebeschreibungen genügen.

2.3.4 Sollwertbestimmung

Durch die Sollwertbestimmung werden für jeden Testfall und die erzeugten Testdaten die erwarteten Ausgaben und der zu erwartende Zustand nach Ausführung des Testfalls für das Testobjekt und seine Umgebung[20] ermittelt. Dabei wird sich auf die für den jeweiligen Testfall relevanten Daten beschränkt [Fewster+99]. Diese Daten werden während der Testauswertung mit den vom Testobjekt während der Testdurchführung erzeugten Ausgaben und Zustände verglichen, um festzustellen, ob ein Fehler aufgetreten ist.

In [Binder99] übernimmt diese beiden Aufgaben ein so genanntes Orakel. Das Orakel wird als eine vertrauenswürdige Quelle für zu erwartenden Ergebnisse beschrieben, welches gleichzeitig einen Vergleichsmechanismus liefert, mit dem die realen mit den erwarteten Ausgaben verglichen werden können. Ein perfektes Orakel würde sich äquivalent zu der zu testenden Komponente verhalten. Es würde alle für die zu testende Komponente spezifizierten Eingaben akzeptieren und ein korrektes Ergebnis für diese Eingaben im Sinne der Spezifikation liefern. Die Entwicklung eines perfekten Orakels ist mindestens genauso kompliziert wie die der zu testenden Komponente und müsste ebenso getestet werden wie die Komponente selbst. Nach [Manna+78] ist es nicht möglich zu beweisen, dass ein Algorithmus in jedem Fall entscheiden kann, ob eine Ausgabe korrekt ist. Daraus folgt, dass es das perfekte Orakel nicht gibt. Für die Automatisierung des funktionalen Tests bedeutet dies, dass die Sollwertbestimmung nicht vollständig sein kann und an die aktuellen Testfälle und Testdaten angepasst werden muss.

2.3.5 Testdurchführung und Testauswertung

Auf der Basis der ersten vier horizontalen Testphasen kann ein ausführbares Programm generiert werden[21], welches den initialen Zustand des jeweiligen Testobjekts und der Testumgebung herstellt, die für den Testfall notwendigen Nachrichtensequenzen mit den entsprechenden Parametern erzeugt, die Sollwerte mit den aktuellen Ausgaben vergleicht und die Ergebnisse dokumentiert. Dieses Programm wird bei der Testdurchführung ausgeführt. Entsprechend der gefundenen Unterschiede zwischen Soll- und Istwerten muss eine Analyse stattfinden, welche die Ursache für diesen Unterschied aufdeckt. Grundsätzlich kann ein Fehler an den folgenden Stellen auftreten:

(1) Die Implementierung entspricht nicht der Spezifikation. Hier muss entschieden werden, ob die Spezifikation oder die Implementierung falsch ist.

(2) Es ist ein Fehler bei der Eingabe der Testdaten aufgetreten.

(3) Der Testfall wurde nicht entsprechend der Testfallspezifikation ausgeführt. Es ist z. B. möglich, dass eine in der Testfallspezifikation enthaltene

[20] dazu zählt u.a. der Zustand anderer Komponenten (Klassen, Teilsysteme, Systeme), Dateien, Datenbanken, etc.

[21] Ob für jeden Testfall ein eigenständig ausführbares Programm generiert wird, oder mehrere Testfälle mit einem einzigen generierten Programm ausgeführt werden, ist eine Frage der Realisierung durch das Testwerkzeug und wird in dieser Arbeit nicht betrachtet.

Nachricht nicht gesendet wurde. Hier muss überprüft werden, ob der Testtreiber der Testfallspezifikation entspricht.

(4) Es wurden falsche Sollwerte ermittelt. Da es das perfekte Orakel nicht gibt[22], besteht immer die Möglichkeit, dass das gewählte Orakel für einige Werte keine korrekten oder zu ungenaue Werte liefert.

(5) Der Vergleich zwischen den Soll- und den Istwerten ist fehlerhaft realisiert.

Durch die weitgehende Automatisierung des Tests sollte es möglich sein, die letzten vier der genannten Fehlerquellen auszuschließen oder zumindest zu minimieren.

Für die Testauswertung ist aber ebenso zu beachten, dass es zu einer Fehlermaskierung kommen kann, also ein aufgetretener Fehler nicht angezeigt wird. Dies kann z. B. durch eine falsche Realisierung des Soll-Ist-Vergleichs auftreten, wobei ein Unterschied zwischen Soll- und Istwert nicht angezeigt wird. Diese Probleme können nur durch eine entsprechende Überprüfung der eingesetzten Testumgebung gelöst werden. Wurde während des Tests ein Fehler in der verwendeten Testumgebung festgestellt, so muss dieser beseitigt und bereits mit dieser Umgebung ausgeführte Testfälle müssen wiederholt werden.

2.3.6 Regressionstest

Ein Regressionstest wird immer dann erforderlich, wenn sich die Implementierung einer bereits getesteten Komponente ändert, ohne dass ihre Spezifikation geändert wurde. Wird zusätzlich die Spezifikation geändert, müssen für diese Komponente neue Testfälle erzeugt oder bereits existierende Testfälle entsprechend angepasst und angewendet werden [McGregor+01]. Dieser Fall wird von McGregor et al. nicht dem Regressionstest zugerechnet.

In [Binder99] wird der Regressionstest allgemeiner als die Wiederverwendung von Testfällen definiert. Dieses schließt explizit den Test von Funktionalitäten einer Oberklasse im Kontext von ihr erbender Klassen unter der Wiederverwendung von Testfällen der Oberklasse ebenso mit ein wie den erneuten Test einer wiederverwendeten Komponente in ihrer neuen Umgebung.

In jedem Fall erfolgt der Regressionstest auf allen Ebenen eines Systems sowie während der verschiedenen Entwicklungsstufen innerhalb eines iterativen Entwicklungsprozesses. Eine Testsuite für eine Komponente entwickelt sich dabei sukzessive über den gesamten Zeitraum der Entwicklung immer weiter [Binder99]. Testfälle, die in einer veränderten Komponente nicht mehr ausgeführt werden können oder die auf Grund einer veränderten Spezifikation keine aussagekräftigen oder sogar falsche Ergebnisse liefern, und redundante Testfälle werden aus der Testsuite entfernt oder entsprechend angepasst. Für neue Funktionalitäten werden Testfälle hinzugefügt. Binder gibt eine genaue Vorgehensweise an, wie eine Testsuite im Laufe des Entwicklungsprozesses weiter entwickelt werden kann. Eine derart gepflegte Testsuite gestaltet den Test effizienter als eine Testsuite, die lediglich um neue Testfälle erweitert wird und somit viele falsche und redundante Testfälle enthält.

[22] Siehe Abschnitt 2.3.4

2.4 Fehlermodelle, Testmodelle und Testmuster

Der Softwaretest kann als ein Suchproblem charakterisiert werden, dessen Aufgabe es ist, aus der in den meisten Fällen enorm großen Menge von Eingabe- und Zustandskombinationen für ein Testobjekt die wenigen heraus zu finden, die einen Fehler auslösen [Binder99]. Die Basis zur Bewältigung dieses Problems können nur verschiedene Modelle für einzelne, den Softwaretest beeinflussende Aspekte sein.

Ein Modell ist eine vereinfachte, abstrakte Darstellung eines bestimmten, abgegrenzten Ausschnitts aus der Realität [Hesse+94]. Es kann sowohl dazu eingesetzt werden, ein Problem vor dessen Umsetzung [Rumbaugh+91] als auch ein bereits realisiertes System in seiner Funktionsweise besser verstehen zu können. Oft ermöglicht erst ein Modell ein Verständnis des modellierten Gegenstandes und somit eine systematische Bearbeitung dessen.

Die beiden wesentlichen, den Softwaretest beeinflussenden Aspekte, sind die Funktionalität des Testobjekts und das der Implementierung zu Grunde gelegte Programmierparadigma. Für die Funktionalität bedarf es dafür keiner weiteren Erläuterungen, da diese der Gegenstand des Tests selbst ist. Das der Implementierung zu Grunde gelegte Programmierparadigma, und bei noch genauerer Betrachtung sogar die ausgewählte Programmiersprache, hat ebenfalls Einfluss auf den Softwaretest. Jedes Programmierparadigma und jede Programmiersprache innerhalb eines Paradigmas, weisen Besonderheiten auf, die nur für sie zutreffen und durch die spezifische Fehlerquellen entstehen. Für die Objektorientierung wurden diese Besonderheiten bereits in den Abschnitten 2.1 und 2.2 vorgestellt. In [Binder99] erfolgt eine ausführliche Beschreibung möglicher Fehlerquellen in verschiedenen objektorientierten Programmiersprachen.

2.4.1 Fehlermodell

Ein Fehlermodell ist eine Annahme darüber, an welchen Stellen in einem Testobjekt mit der größten Wahrscheinlichkeit Fehler zu finden sind. Welches Fehlermodell als Grundlage für den Test verwendet wird, ist sowohl vom Testobjekt selbst als auch vom verwendeten Programmierparadigma, u.U. sogar von der verwendeten Programmiersprache abhängig. Fehlermodelle können grundsätzlich in zwei Kategorien unterteilt werden: nicht-spezifische und spezifische Fehlermodelle [Binder99]. Ein **nicht-spezifisches Fehlermodell** wird verwendet, um zu zeigen, dass das Testobjekt die gestellten Anforderungen erfüllt. Dabei reicht es aus, einen Fehler zu finden, um zu zeigen, dass das Testobjekt nicht konform zu einer überprüften Anforderung ist. Ziel dieses Tests ist es also nicht, möglichst viele Fehler zu finden, sondern nur zu zeigen, dass kein Fehler auftritt. Dafür reicht es aus, wenn jede Teilfunktionalität mindestens einmal ausgeführt wird. Für diese Art des Tests wird der Begriff **konformitätsgerichtetes Testen** verwendet. Diese Definition zeigt bereits, dass ein Test auf der Basis eines solchen Fehlermodells kein ausreichendes Vertrauen in die korrekte Umsetzung der Spezifikation schaffen kann. Ein fehlerhaftes Testobjekt kann ohne weiteres erfolgreich auf der Basis eines nicht-spezifischen Fehlermodells getestet werden.

Die gezielte Aufdeckung von Fehlern ist die Aufgabe eines **spezifischen Fehlermodells**, welches auf den Besonderheiten eines Programmierparadigmas oder einer bestimmten Form der Realisierung einer Funktionalität[23] basiert. Ein spezifisches Fehlermodell ist die Basis für den **fehlergerichteten Test**. Durch diesen Test sollen also

[23] wie z.B. Entwurfsmustern

möglichst viele, im Idealfall alle Fehler, in der getesteten Implementierung gefunden werden.

Für den Test objektorientierter Systeme müssen auf Grund ihrer Besonderheiten[24] gegenüber dem Test konventioneller Software neue Fehlermodelle entwickelt bzw. bereits existierende Fehlermodelle angepasst werden. Hier seien noch mal kurz die Begriffe Vererbung, Polymorphismus, Datenkapselung und abstrakte Klassen genannt[25].

2.4.2 Testmodell

Ein Testmodell ist ein Modell der zu realisierenden oder bereits implementierten zu testenden Funktionalität, welches genügend Informationen enthält, um ausführbare Testfälle für diese Funktionalität bzw. Implementierung automatisch erzeugen zu können [Binder99]. Alle anderen Modelle, die diese Anforderungen nicht erfüllen, werden von Binder als „Cartoons" bezeichnet. Diese sind inkonsistent, mehrdeutig und un vollständig. Sie können zwar auch als Grundlage zur Erstellung von Testfällen verwendet werden. Allerdings muss der Tester auf Grund der genannten Eigenschaften Interpretationen vornehmen. Das Gleiche gilt für den Programmierer. Daraus kann man ersehen, welche Probleme sich für den gesamten Entwicklungsprozess, inklusive des Tests, ergeben. Die Wahrscheinlichkeit, dass das Modell von mindestens einem der Beteiligten falsch interpretiert wird, ist relativ hoch. Kommt es sogar dazu, dass beide das Modell in der gleichen Art und Weise missverstehen, so können die daraus resultierenden Fehler während des Tests nicht aufgedeckt werden. Solche Modelle stellen also ein erhebliches Hindernis für einen effizienten und effektiven Entwicklungsprozess dar und beeinträchtigen die Qualität der erstellten Software in hohem Maße. Eine Automatisierung ist auf der Grundlage solcher „Cartoons" nicht möglich.

Für objektorientierte Systeme kann die Rolle des Testmodells durch ein UML-Modell realisiert werden. Auch wenn die UML-Spezifikation keine vollständigen UML-Modelle im Sinne eines Testmodells fordert und somit die Modelle sehr häufig noch nicht genügend Informationen enthalten, um als Testmodell fungieren zu können, bietet die UML die notwendigen Voraussetzungen dafür. Für den Klassentest wird in Kapitel 4 die Erstellung testbarer UML-Modelle erläutert.

2.4.3 Testmuster

Christopher Alexander definiert ein Muster wie folgt:

> „Jedes **Muster** beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen." [Alexander+77]

Diese Definition kann neben der Architektur auf viele verschiedene Themengebiete angewendet werden, u.a. auch auf den Entwurf und den Test von Software [Buschmann+98] [Gamma+98] [Binder99].

Testmuster werden in einem Schema mit mehreren Abschnitten beschrieben, von denen die wichtigsten im Folgenden kurz beschrieben werden. Eine vollständige und ausführliche Beschreibung ist in [Binder99] zu finden. Mit dem **Ziel** wird beschrieben, welche Arten von Testfällen mit dem Testmuster erzeugt werden. Der **Kontext** beschreibt, wel-

[24] vgl. Abschnitte 2.1 und 2.2
[25] siehe Abschnitt 2.2.1

ches Testproblem mit dem Muster gelöst wird und wann und auf welche Softwarekomponenten dieses Muster angewendet werden kann. Das **Fehlermodell** beschreibt die Art der Fehler, die mit diesem Testmuster aufgedeckt werden können und warum diese Fehler gefunden werden. Mit der **Strategie** wird beschrieben, wie die Testfälle entworfen und implementiert werden sollten. Der Abschnitt **Voraussetzungen** listet die Bedingungen auf, die erfüllt sein müssen, bevor das Muster angewendet werden kann. Das **Endekriterium** beschreibt die Bedingungen, die erfüllt sein müssen, damit der Test beendet werden kann. Im Abschnitt **Konsequenzen** werden die Vor- und Nachteile bei der Anwendung dieses Testmusters aufgezählt.

In Abbildung 2.2 sind die Beziehungen zwischen Fehlermodell, Testmuster und Testmodell dargestellt.

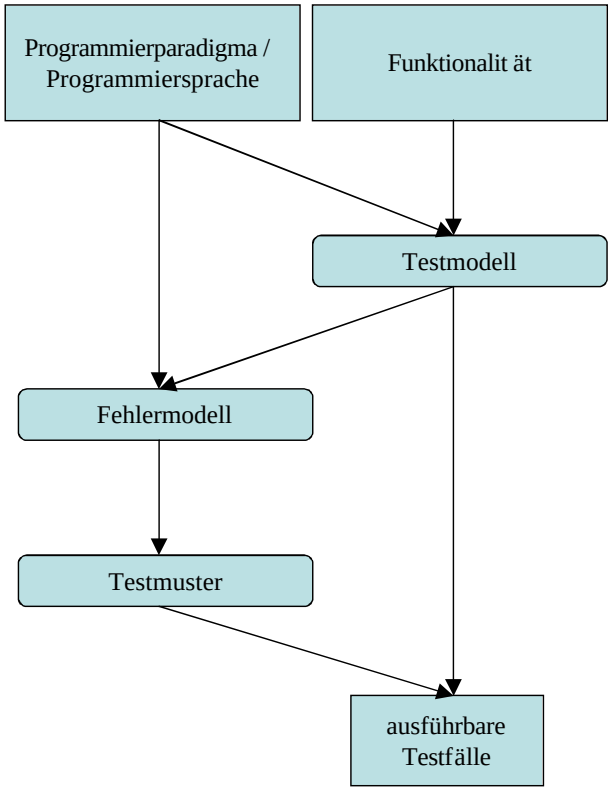


Abbildung 2.2 *Beziehungen zwischen Testmodell, Fehlermodell und Testmuster*

Das Programmierparadigma bzw. die Programmiersprache und eine bestimmte Funktionalität bilden die Grundlage, sowohl für das anzuwendende Fehlermodell, als auch für das Testmodell. Das Testmuster und das Testmodell sind die Basis für die Erzeugung ausführbarer Testfälle. Das Testmuster legt dabei fest, welche Aspekte wie getestet werden sollen. Das Testmodell muss die dafür notwendigen Informationen zur Verfügung stellen.

2.5 Die Automatisierung des Tests

Die wachsenden Anforderungen hinsichtlich der Sicherheit und Zuverlässigkeit von Softwaresystemen erhöhen die Bedeutung des Tests in immer stärkerem Maße. Die dabei gleichzeitig wachsende Komplexität der zu erstellenden Systeme lässt die Automatisierung des Tests immer mehr zu einer Notwendigkeit werden.

Testautomatisierung bedeutet in der Praxis in den meisten Fällen nur eine Automatisierung einzelner Testaktivitäten. Nur in seltenen Fällen kann eine umfassende

Automatisierung des Testprozesses erreicht werden. Dies liegt vor allem an einer unzureichenden Werkzeugunterstützung und wird durch ein Grundproblem vorhandener Testwerkzeuge verursacht. Diese sind an keine bestimmte Spezifikationssprache gebunden bzw. implizieren kein bestimmtes Testmuster [Rüppel97]. Dadurch sind sie zwar relativ einfach und flexibel einsetzbar, unterstützen den Testprozess aber nur in geringem Maße. Ohne eine zu Grunde gelegte Spezifikationssprache ist eine Automatisierung der Testfallermittlung sowie der Testdaten- und Sollwertbestimmung nicht möglich. Die mangelnde Unterstützung bestimmter Testmuster wiederum belässt die Verantwortung der Testfallauswahl dem Tester und macht damit die Qualität des Tests von dessen Erfahrungen abhängig [Rüppel97]. Dadurch ist eine Gewährleistung einer gleich bleibend hohen Qualität des Tests nur in einem sehr beschränkten Rahmen möglich.

Auf der anderen Seite sind Werkzeuge, die eine bestimmte Spezifikationssprache voraussetzen, in der Lage, die oben genannten Aktivitäten zu automatisieren. Allerdings werden in heutigen Testwerkzeugen nur sehr spezielle Spezifikationssprachen eingesetzt [Rüppel97]. Dadurch entsteht das Problem, dass die Spezifikation des Softwaresystems mit einer Sprache wie der UML erfolgt, während der Test in einer anderen, vom Testwerkzeug vorgegebenen Sprache spezifiziert wird. Dies macht eine Verteilung und auch Redundanz von Informationen notwendig, was einen stark erhöhten Verwaltungsaufwand nach sich zieht und für die Einhaltung der Konsistenz problematisch ist. Kann die Konsistenz nicht gewahrt werden, sinkt die Qualität des Tests in erheblichem Maße und die Effektivität sinkt durch die Suche nach fälschlicherweise gemeldeten Fehlern deutlich.

Die in [Rüppel97] vorgestellten Werkzeuge unterstützen, wenn überhaupt, nur bestimmte Testmuster und sind daher nur sehr begrenzt einsetzbar. Der Einsatz verschiedener Testwerkzeuge für die Unterstützung verschiedener Testmuster ist allerdings sehr aufwendig. Da für jedes Werkzeug andere Einschränkungen gelten, muss jeweils eine eigene Testumgebung geschaffen werden, die eine Einbettung in den jeweiligen Entwicklungsprozess ermöglicht.

Der von Rüppel vorgestellte Ansatz eines Frameworks zur flexiblen Konstruktion von Testwerkzeugen aus einzelnen Bausteinen geht dort einen anderen, vielversprechenderen Weg. Dabei sollen für den objektorientierten Test allgemein gültige Eigenschaften und Tätigkeiten in sog. Kernklassen zusammengefasst werden, die flexibel durch spezifische Komponenten erweitert werden können. Dieses Konzept ermöglicht die flexible Konstruktion von Testwerkzeugen, die an die jeweiligen Bedingungen angepasst sind, sowie eine stetige Erweiterung der Möglichkeiten durch die Umsetzung neuer Testmuster im Rahmen des Frameworks.

Es muss allerdings festgestellt werden, dass die Testwerkzeuge kaum Ansätze zur Integration in den Softwareentwicklungsprozess anbieten. Es werden verschiedene Testaktivitäten und Testmuster umgesetzt. Dies geschieht aber unabhängig von aktuellen Softwareentwicklungsstandards, wie der UML, und den dafür eingesetzten Werkzeugen.

In [Binder99] wird versucht, den Testprozess, unter Einbeziehung der UML als Standard für den Entwurf objektorientierter Systeme, stärker mit dem restlichen Entwicklungsprozess zu verbinden. Als Grundlage dafür sieht Binder die Erstellung testbarer Modelle zur Spezifikation eines Softwaresystems. Ein Modell wird dabei als testbar bezeichnet, wenn es genügend Informationen enthält, um allein aus diesen Informationen automatisch ausführbare Testfälle generieren zu können. Voraussetzung für die Erstellung eines testbaren Modells ist die Verwendung einer Spezifikationssprache, mit der

für den Test notwendige Informationen in einer eindeutigen Art und Weise beschrieben werden können. Die UML bietet, wie in Kapitel 3 beschrieben wird, eine überschaubare Menge an Basiselementen zur Spezifikation von Softwaresystemen an, deren Beziehungen in dem zu Grunde gelegten Metamodell beschrieben werden. Die Einbindung der formalen Spezifikationssprache OCL als Bestandteil der UML[26] ermöglicht die präzise Beschreibung von Bedingungen, die mit den graphischen Mitteln der UML nur sehr schwer oder gar nicht beschrieben werden können. Zusätzlich kann die Menge der Basiselemente der UML durch einen integrierten Erweiterungsmechanismus in einer genau definierten Form um anwendungsspezifische Elemente erweitert werden. Auf dieser Basis ist es möglich, testbare UML-Modelle zu erstellen. Allerdings stellt Binder auch fest, dass allein der Einsatz einer Spezifikationssprache, welche die entsprechenden Möglichkeiten bietet, als Ergebnis kein testbares Modell garantiert. Dieses trifft auch für die UML zu. Allerdings muss auch gesagt werden, dass dies nicht die Aufgabe der UML ist, die nur eine relativ allgemeine und damit sehr flexibel einsetzbare Notation zur Spezifikation anbieten soll.

Um den Klassentest zu automatisieren, müsste also ein UML-Modell erstellt werden, welches sämtliche Informationen enthält, um die verschiedenen horizontalen Testaktivitäten auf der Ebene des Klassentests automatisch durchführen zu können. Der in dieser Arbeit vorgestellte Ansatz greift die Idee eines flexibel einsetzbaren Frameworks von Rüppel auf und überträgt diese auf den Entwurf eines testbaren UML-Modells. Der Erweiterungsmechanismus der UML wird genutzt, um ein sog. Profil zu erstellen, welches verschiedene Stereotypen zur Verfügung stellt, die die jeweils notwendigen Informationen enthalten bzw. den Anwender dazu auffordern, bestimmte Informationen in einer festgelegten Weise anzugeben. Dabei ist eine Kombination verschiedener Stereotypen zur flexiblen Konstruktion von unterschiedlichen Testmustern möglich. Außerdem soll das Profil erweiterbar und somit an verschiedene Anforderungen hinsichtlich des Tests anpassbar sein. Das erstellte Profil und grundlegende Bedingungen für seinen Einsatz werden in Kapitel 4 beschrieben.

[26] Die OCL wird auch zur Spezifikation des Metamodells der UML verwendet.

Kapitel 3

Objektorientierter Entwurf auf der Basis der UML

3.1 Einführung

Die Unified Modeling Language (UML) ist eine graphische Sprache zur Darstellung, Spezifikation, Konstruktion und Dokumentation von Softwaresystemen [Booch+99] [Wahl98] [Kleiner98] [Roselli98] [Binder99].

Seit der Anerkennung als Standard für Modellierungssprachen durch die OMG 1997, wurde die UML stetig weiter entwickelt und an aktuelle Entwicklungen im Bereich der Softwareentwicklung angepasst. Dies war die Grundlage für ihre ständig wachsende Bedeutung und Verbreitung.

Die UML basiert auf einem Metamodell, in welchem die Semantik der einzelnen Elemente und deren Beziehungen spezifiziert sind[27]. Es stehen verschiedene Diagrammarten zur Verfügung, durch die jeweils eine andere Sicht auf ein Softwaresystem bzw. seine Komponenten dargestellt werden kann. Neben der Verwendung der grundlegenden Elemente der UML ist es auch möglich, die Sprache in einer definierten Form[28] um eigene Elemente zu erweitern.

Die Object Constraint Language (OCL) ist ebenfalls eine Modellierungssprache. Sie ist ein Bestandteil der UML[29] und wird verwendet, um Eigenschaften von modellierten Elementen und Beziehungen zwischen diesen Elementen auszudrücken, die mit Hilfe der graphischen Notation der UML nicht oder nur sehr schwierig darzustellen sind.

In den folgenden Abschnitten werden das Metamodell, die verschiedenen Diagramme sowie die OCL kurz erläutert. Für die Diagramme und die OCL wird zusätzlich deren Bedeutung für den spezifikationsbasierten Klassentest und dessen Automatisierung dargestellt.

Eine ausführliche Beschreibung der UML ist u.a. in [Booch+99], [Booch+99a] und [OMG01] zu finden. Eine detaillierte Beschreibung der OCL kann [OMG99] und [Warmer+99] entnommen werden.

---

[27] Die Definition der Semantik der einzelnen Elemente der UML lässt allerdings sehr viele Fragen offen, was insbesondere für den Test und seine Automatisierung problematisch ist. Die für den Klassentest relevanten Aspekte werden in Kapitel 4 dargelegt und entsprechende Lösungen dafür vorgestellt.

[28] mit Hilfe von „Stereotypen“

[29] seit UML-Version 1.1 (November 1997)

3.2 Das UML-Metamodell

Die Architektur der UML basiert auf einer vierschichtigen Metamodellstruktur. Diese enthält die folgenden Schichten: Meta-Metamodell, Metamodell, Modell und Benutzerobjekte. Die ersten drei Schichten der Struktur sind in Abbildung 3.1 vereinfacht dargestellt. Eine genaue Definition kann [OMG01] und [Booch+99a] entnommen werden.

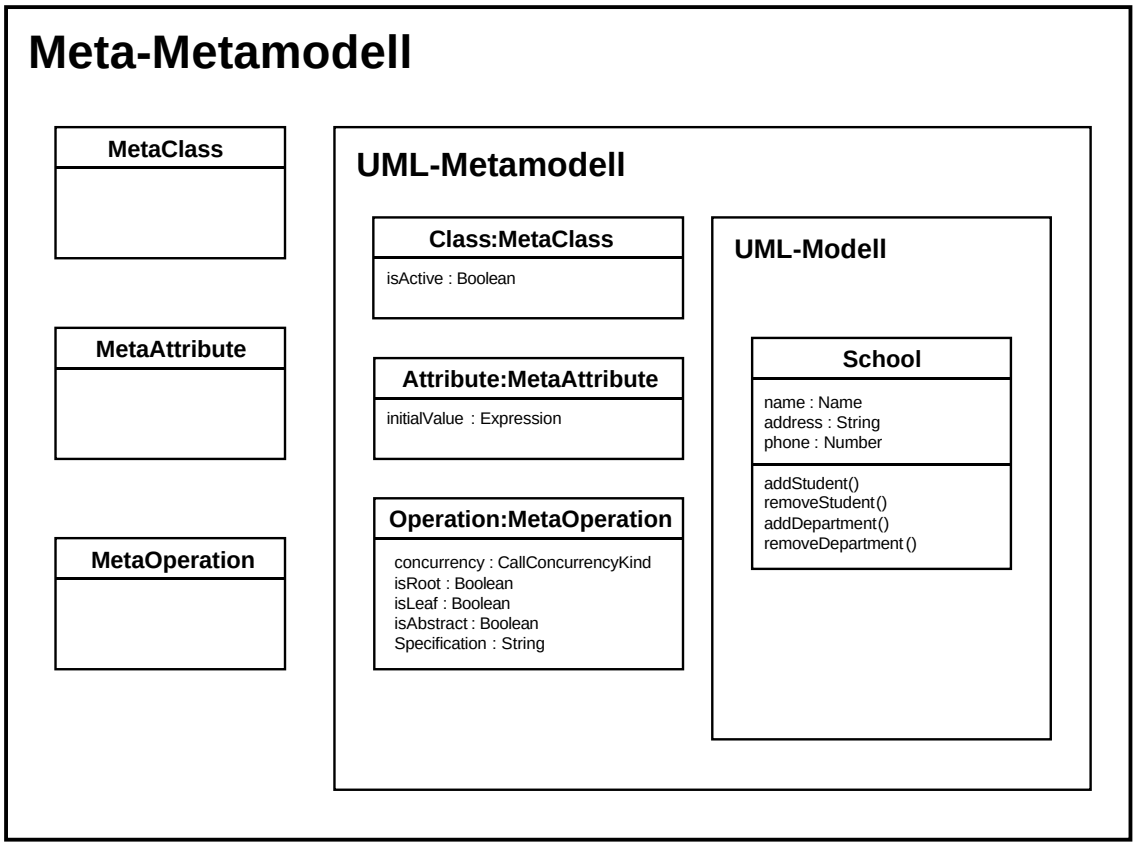


Abbildung 3.1 *UML-Metamodellarchitektur*

Die Aufgabe des Meta-Metamodells besteht in der Definition einer Sprache für die Erstellung von Metamodellen. Ein konkretes Metamodell ist eine Instanz des Meta-Metamodells. Mit Hilfe eines Metamodells wird wiederum eine Sprache zur Erstellung von Modellen definiert. Die UML wird mit genau einem solchen Metamodell beschrieben. Darin sind die Elemente der UML und deren Beziehungen untereinander festgelegt. Durch dieses Metamodell werden das Vokabular und die Regeln zur Konstruktion von UML-Modellen beschrieben. Jedes konkrete Modell ist eine Instanz des Metamodells, auf dem es basiert. Es beschreibt einen bestimmten Informationsbereich.

Für die Objektebene gilt die gleiche Struktur. Beispielsweise ist ein Metaobjekt der Klasse `Class` auf der Metamodellebene eine Instanz des Meta-Metaobjekts `MetaClass` auf der Meta-Metamodellebene. Die in Abbildung 3.1 dargestellte Klasse `School` ist ein Objekt auf Modellebene und eine Instanz des Metaobjekts `Class`.

Die vierte Ebene der Metamodellstruktur ist die Ebene der Benutzerobjekte[30]. Diese sind Instanzen von Objekten aus dem Modell.

[30] Diese Ebene ist in Abbildung 3.1 nicht dargestellt.

3.3 UML-Diagramme

In diesem Abschnitt werden die verschiedenen UML-Diagramme vorgestellt. Es erfolgt eine kurze Beschreibung und eine Einordnung der Diagramme innerhalb der UML. Weiterhin soll betrachtet werden, ob die Diagramme für den spezifikationsbasierten Klassentest und dessen Automatisierung von Bedeutung sind und welche Rolle sie dabei gegebenenfalls übernehmen können. Ein wichtiger Aspekt für diese Beurteilung ist das Abstraktionsniveau, auf dem das zu modellierende System oder ein Teilsystem durch das jeweilige Diagramm dargestellt wird.

Dabei muss die Frage beantwortet werden, ob die für den Test einer Klasse und dessen Automatisierung relevanten Informationen in dem jeweiligen Diagramm ganz oder teilweise dargestellt werden können und ob eine explizite und eindeutige Verknüpfung zwischen dem Diagramm oder einzelner darin enthaltener Elemente und einem für den Test relevanten Element möglich ist. So ist es z. B. notwendig und auch möglich, dass ein Zustandsdiagramm eindeutig einer Klasse zugeordnet wird.

Mit Hilfe der verschiedenen Diagrammarten in der UML kann eine Komponente aus unterschiedlichen Blickwinkeln betrachtet werden. In jedem Diagramm werden andere Aspekte in den Mittelpunkt gestellt. In Klassendiagrammen werden z. B. Vererbungs- und Assoziationsbeziehungen zwischen Klassen dargestellt. Diese vermitteln eine statische Sicht auf das System. In Interaktionsdiagrammen hingegen sind die Beziehungen zwischen den enthaltenen Elementen von dynamischer Natur. Dabei handelt es sich um die zwischen den Elementen versendeten Nachrichten. Durch diese Trennung der verschiedenen Aspekte eines Gesamtsystems werden eine strukturierte und verständliche Definition und Darstellung selbst komplexer Strukturen möglich. Allerdings erfordert die Verwaltung von verschiedenen Sichten auf ein einzelnes System einen erhöhten Aufwand zur Wahrung der Konsistenz zwischen den einzelnen Sichten. Hier bietet die UML selbst kaum Unterstützung an. Vorhandene Modellierungswerkzeuge können auf Grund fehlender theoretischer Konzepte nur bedingt weiterhelfen. Mit dem in dieser Arbeit entwickelten Ansatz wird diese theoretische Basis erweitert.

3.3.1 Anwendungsfalldiagramm

Anwendungsfalldiagramme werden dazu verwendet, die Umgebung einer Komponente sowie die Anforderungen dieser Umgebung an eine Komponente und deren für die Umgebung sichtbaren Reaktionen zu modellieren. Die Umgebung einer Komponente wird durch sog. Akteure, die sich außerhalb der zu modellierenden Komponente befinden, und deren Beziehungen zu dieser Komponente definiert. Bei den Akteuren kann es sich z. B. um Anwender oder um andere Soft- und Hardware-Systeme bzw. Komponenten handeln, die mit der modellierten Komponente kommunizieren. In Abbildung 3.2 wird ein Anwendungsfalldiagramm exemplarisch dargestellt.

Durch die Anforderungen wird festgelegt, welche Funktionen die Komponente aus Sicht der mit ihr kommunizierenden Akteure übernehmen soll. Dabei wird auch bestimmt, welchem Akteur welche Funktionen zur Verfügung stehen. Anwendungsfälle können hierarchisch strukturiert werden, wodurch eine schrittweise Verfeinerung und eine übersichtliche Darstellung möglich wird. In Anwendungsfalldiagrammen wird nicht darauf eingegangen, wie diese Anforderungen durch die Komponente realisiert werden.

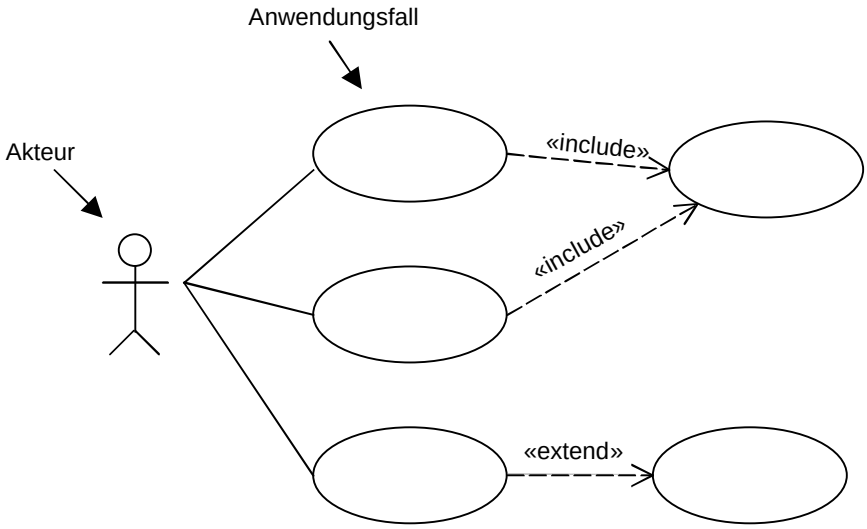


Abbildung 3.2 *Anwendungsfalldiagramm*

Anwendungsfalldiagramme ermöglichen eine Kommunikation über die zu erstellende Komponente auf einem hohen Abstraktionsniveau. Dadurch wird die Verständigung zwischen den verschiedenen Gruppen von Beteiligten[31] stark vereinfacht.

In der Literatur wird die Verwendung von Anwendungsfalldiagrammen häufig auf die oberen Abstraktionsebenen eines Systems beschränkt dargestellt [Roselli98] [Wahl98] [Weiß97]. Die Urheber der UML, Booch, Rumbaugh und Jacobson, gehen in [Booch+99] allerdings davon aus, dass Anwendungsfälle auch für die Beschreibung der Funktionalität von Teilsystemen und sogar von einzelnen Klassen verwendet werden können. Dabei sollen sie auch als Basis für die Erzeugung von Testfällen dienen können.

Testautomatisierung

Da die in Anwendungsfalldiagrammen dargestellten Informationen sehr abstrakt sind, Testmuster für den Klassentest aber auf viel konkreteren Strukturen aus Klassendiagrammen und Zustandsautomaten basieren, ist deren Einsatz als Informationsquelle für die automatische Erzeugung ausführbarer Testfälle nicht sinnvoll. Ein Aspekt dabei ist, dass mit dieser Art von Diagrammen lediglich Anforderungen an eine Komponente modelliert werden. Es könnte also lediglich ein konformitätsgerichteter Test unterstützt werden[32]. Die dabei erzeugten Testfälle würden bei einem systematischen Test durch die Testfälle der sehr viel spezielleren Testmuster für den Klassentest mit abgedeckt werden, brächten also keine zusätzlichen Informationen über das Testobjekt.

Der Einsatz von Anwendungsfalldiagrammen für die automatische Dokumentation der erzeugten Testfälle ist ebenfalls nicht sinnvoll. Durch die unterschiedliche Basis der für den Klassentest verwendeten Testmuster ist eine direkte Zuordnung sehr schwierig und in den meisten Fällen unmöglich. Die entsprechenden Informationen können besser an den Diagrammen annotiert werden, die die Basis für die Testmuster bilden.

3.3.2 Klassendiagramm

Klassendiagramme sind beim Entwurf objektorientierter Systeme von zentraler Bedeutung. Mit ihnen wird die statische Entwurfssicht auf ein System modelliert. In

[31] Anwender, Auftraggeber, Analysten, Modellierer, Entwickler
[32] siehe Abschnitt 2.4.1

Klassendiagrammen werden Klassen, Schnittstellen[33] und Kollaborationen sowie deren Beziehungen dargestellt, wie dies beispielhaft in Abbildung 3.3 gezeigt wird.

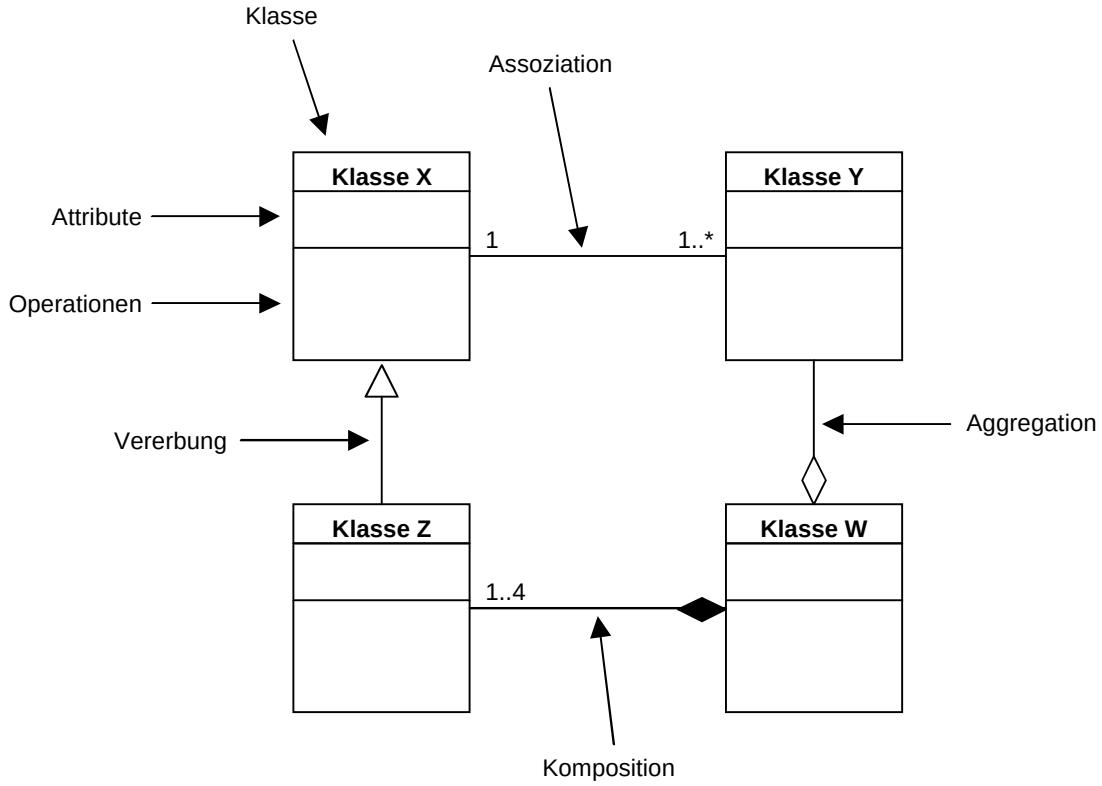


Abbildung 3.3 *Klassendiagramm*

Ein System besteht i. Allg. aus mehreren Klassendiagrammen. Jedes dieser Diagramme sollte genau eine Kollaboration darstellen und nur die für diese Kollaboration relevanten Elemente enthalten. Zur Erstellung von Klassendiagrammen werden von Booch et al. mehrere Schritte empfohlen. Dazu zählt unter anderem die Identifizierung aller an der Kollaboration beteiligten Elemente. Daran anschließend sollen Szenarien entwickelt werden, mit deren Hilfe eine Überprüfung hinsichtlich der Vollständigkeit und semantischen Korrektheit der ausgewählten Elemente in Bezug auf die durch sie zu realisierende Kollaboration erfolgen kann. Dieses kann hervorragend durch Anwendungsfalldiagramme geschehen, was noch einmal verdeutlicht, dass diese auch auf der Ebene von Klassen eingesetzt werden können[34].

Klassendiagramme besitzen auch deswegen eine so große Bedeutung innerhalb des objektorientierten Entwurfs, weil sie die direkte Verbindung zwischen dem Entwurf und der für die Realisierung eingesetzten Programmiersprache darstellen. Aus einem Klassendiagramm kann Quellcode generiert werden, dessen Struktur dem Klassendiagramm entspricht[35]. Dieser Schritt wird von heutigen Entwurfswerkzeugen sehr gut unterstützt. Dadurch ist es relativ einfach möglich, das Modell und die entsprechende Implementierung konsistent zu halten, was ein wesentlicher Faktor für den Erfolg eines Projektes ist.

[33] Metaklasse `Interface`
[34] siehe Abschnitt 3.3.1
[35] Dieser Schritt wird auch als „Forward Engineering“ bezeichnet.

Testautomatisierung

Klassendiagramme enthalten die grundlegenden Informationen über die Struktur einzelner Klassen und Schnittstellen sowie deren Beziehungen untereinander. Diese werden im „normalen“[36] Entwicklungsprozess sehr häufig zur Quellcodegenerierung verwendet und können somit als zentrale Basis für die Automatisierung des Klassentests angesehen werden.

Durch diese direkte Beziehung zwischen Klassendiagrammen bzw. den darin modellierten Elementen[37] und der Implementierung können für den Test relevante Informationen direkt an den entsprechenden Komponenten annotiert und für die Erzeugung der ausführbaren Testfälle, durch Erzeugung entsprechenden Quellcodes, genutzt werden.

Diese gegenüber dem konventionellen Einsatz von Klassendiagrammen weiterreichendere und präzisere Spezifikation kann zusätzlich Inkonsistenzen und Unvollständigkeiten in dem UML-Modell aufdecken. Durch die direkte Verknüpfung der für den Test erforderlichen Informationen mit den betroffenen Elementen wird zudem eine Verteilung von Informationen verhindert, die einen erhöhten Verwaltungsaufwand und mit hoher Wahrscheinlichkeit Inkonsistenzen zur Folge hätte.

3.3.3 Interaktionsdiagramme

Mit Hilfe eines Interaktionsdiagramms werden dynamische Aspekte einer Komponente modelliert. Es zeigt eine Interaktion, die sich aus den an der Interaktion beteiligten Elementen und den zwischen diesen Elementen versendeten Nachrichten zusammensetzt. Die Elemente eines Interaktionsdiagramms können verschiedene Arten von Instanzen sein, wie z. B. Instanzen von Klassen, Schnittstellen, Komponenten und Knoten. Wird eine Schnittstelle in einem Interaktionsdiagramm dargestellt, bedeutet dies, dass die modellierten Beziehungen für alle Objekte von allen Klassen, welche die modellierte Schnittstelle realisieren, gelten.

Ein Interaktionsdiagramm kann mit verschiedenen Komponenten eines UML-Modells verknüpft werden, so z. B. mit dem gesamten System, einem Teilsystem, einer Klasse, einem Anwendungsfall oder einer Kollaboration.

Es gibt zwei verschiedene Arten von Interaktionsdiagrammen, die semantisch äquivalent sind, das Sequenzdiagramm und das Kollaborationsdiagramm. Semantisch äquivalent bedeutet, dass ein Diagramm der einen Art jeweils in ein Diagramm der anderen Art transformiert werden kann, ohne dass dabei Informationen verloren gehen. Der Unterschied zwischen beiden Diagrammen liegt in der Art der Darstellung der Beziehungen zwischen den Elementen, wodurch der Schwerpunkt auf unterschiedliche Aspekte gelegt wird.

Ein **Sequenzdiagramm** zeigt sehr deutlich die zeitliche Reihenfolge zwischen den einzelnen versendeten Nachrichten sowie den Steuerungsfluss zwischen den beteiligten Komponenten. In Abbildung 3.4 wird ein Sequenzdiagramm exemplarisch dargestellt.

[36] also auch ohne eine so starke Integration des Tests in den restlichen Entwicklungsprozess
[37] Klassen, Interfaces, Attribute, Operationen, Parameter und deren Beziehungen untereinander

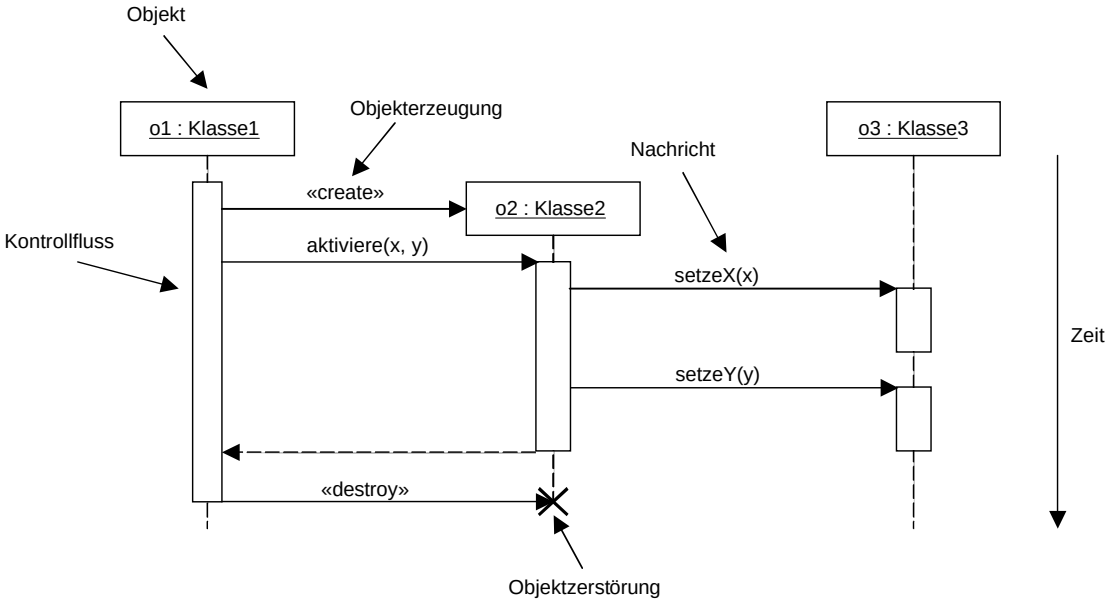


Abbildung 3.4 *Sequenzdiagramm*

In einem **Kollaborationsdiagramm** wird dagegen die strukturelle Organisation zwischen den beteiligten Elementen besser dargestellt. Dies wird aus der Darstellung in Abbildung 3.5 deutlich. Das dort dargestellte Kollaborationsdiagramm ist äquivalent zu dem in Abbildung 3.4 dargestellten Sequenzdiagramm.

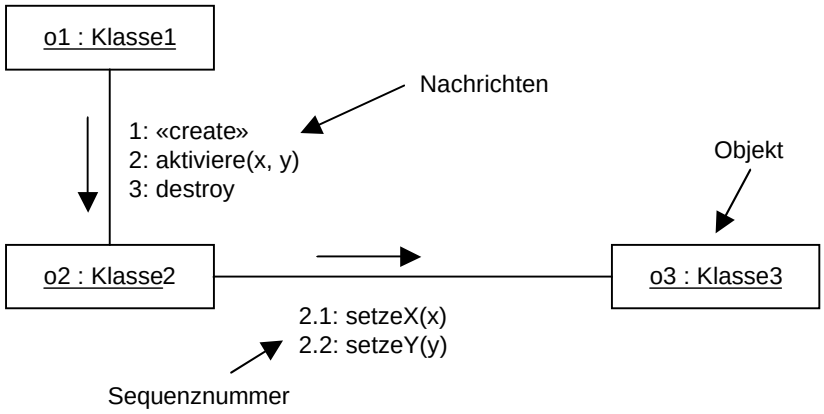


Abbildung 3.5 *Kollaborationsdiagramm*

Die Reihenfolge der Nachrichten und der Steuerungsfluss können diesem Diagramm zwar ebenfalls entnommen werden[38], nur ist dies für einen Betrachter schwieriger als bei Sequenzdiagrammen. Für Sequenzdiagramme gilt das Gleiche wie für die strukturelle Organisation zwischen den Elementen.

Ein weiterer Unterschied, der sich durch die verschiedene Art der Repräsentation ergibt, ist die Tatsache, dass Sequenzdiagramme nur für die Darstellung von relativ einfachen Wiederholungsanweisungen und Verzweigungsstrukturen geeignet sind. Komplexere Strukturen dieser Art lassen sich übersichtlicher in Kollaborationsdiagrammen darstellen.

[38] durch die entsprechende Nummerierung der Nachrichten

Testautomatisierung

Das ursprüngliche Ziel bei der Verwendung von Interaktionsdiagrammen ist die Darstellung von Beziehungen und Interaktionen zwischen mehreren Objekten zur Erfüllung einer gemeinsamen Aufgabe. Deshalb liegt der Schwerpunkt ihrer Verwendung für den Test eher im Teilsystemtest.

Allerdings besteht durchaus die Möglichkeit, Interaktionsdiagramme in Einzelfällen für die Spezifikation von sehr speziellen Methodensequenzen zu verwenden, die gezielt getestet werden sollen. Bei der Spezifikation eines solchen Interaktionsdiagramms kann für jeden Parameter entschieden werden, ob ein konkreter Wert festgelegt oder ein Platzhalter verwendet wird. Für die mit einem Platzhalter belegten Parameter kann dann ein Muster zur Testdatenerzeugung die konkreten Testdaten generieren. So kann eine Methodensequenz den jeweiligen Bedürfnissen entsprechend getestet werden. Sind alle Parameter aller im Interaktionsdiagramm spezifizierten Methodenaufrufe mit konkreten Werten belegt, so wird durch dieses Interaktionsdiagramm genau ein Testfall erzeugt, der ein ganz spezielles Szenario testet. Sind hingegen alle Parameter mit Platzhaltern belegt, so wird dadurch zum Ausdruck gebracht, dass die spezifizierte Methodensequenz unabhängig von den verwendeten Parametern als kritisch angesehen wird. In diesem Fall wird eine Menge von Testfällen erzeugt, die die Methodensequenz mit verschiedenen Parameterdaten testet.

3.3.4 Zustandsdiagramm

Ein Zustandsdiagramm dient der Beschreibung des Verhaltens einer Komponente. Diese Komponente kann das gesamte System, ein Teilsystem oder aber auch eine einzelne Klasse sein.

Die Spezifikation des Verhaltens erfolgt mit Hilfe von Zuständen, Zustandsübergängen und Ereignissen, durch die bei einer Instanz der zugehörigen Komponente bestimmte Aktionen ausgelöst werden[39]. Dies wird exemplarisch in Abbildung 3.6 dargestellt.

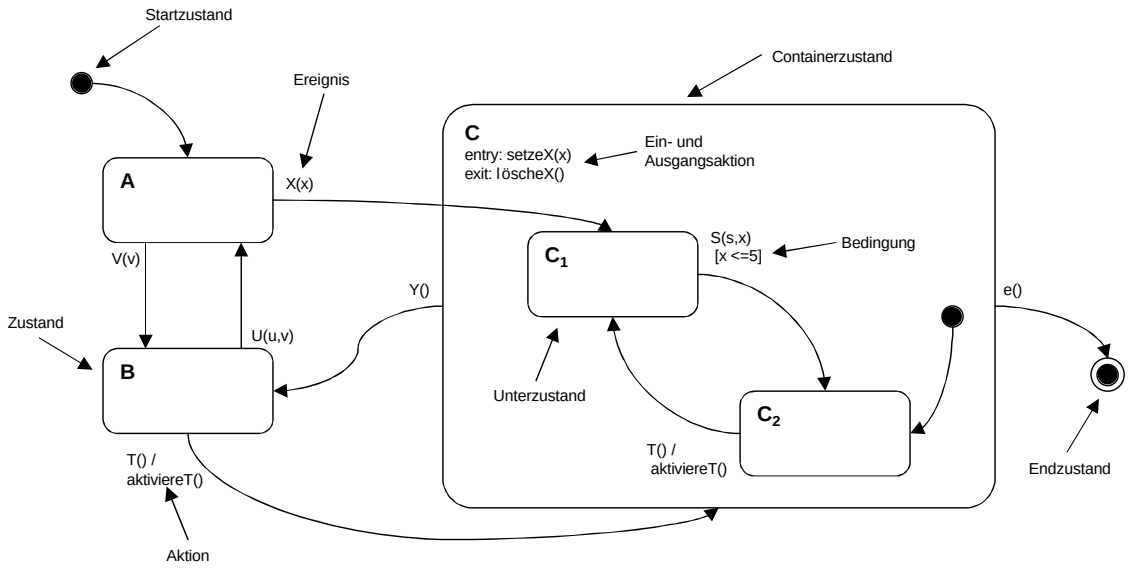


Abbildung 3.6 *Zustandsdiagramm*

[39] Zustandsübergänge, Versenden von Nachrichten, etc.

Von zentraler Bedeutung dabei ist die Definition eines Zustands. Binder gibt in [Binder99] drei Kategorien an, mit deren Hilfe ein Zustand charakterisiert werden kann: Sichtbarkeit, Bereich und Granularität. Die **Sichtbarkeit** stellt den grundlegenden Gegenstand des Zustandsmodells dar und kann abstrakt oder konkret sein. **Abstrakt** bedeutet, dass der Zustand die öffentliche Schnittstelle der zugehörigen Komponente beschreibt[40]. Ein **konkreter** Zustand beschreibt die interne Implementierung[41]. Der **Bereich** charakterisiert den Grad der Abstraktion und kann beschränkt oder rekursiv sein. **Beschränkt** bedeutet, dass nur abstrakte Werte von Zustandsvariablen[42] betrachtet werden. Bei einem **rekursiven** Bereich werden die Zustandsvariablen bis zu den primitiven Datentypen aufgelöst. Die **Granularität** bezeichnet die Einheit der Definition eines Zustandes. Diese kann zusammengesetzt oder primitiv sein. In einem **zusammengesetzten** Zustand[43] können mehrere unterschiedliche Wertekombinationen der Zustandsvariablen enthalten sein. Bei **primitiven** Zuständen stellt jede einzelne Wertekombination einen eigenen Zustand dar.

In der Praxis werden Zustandsautomaten in den meisten Fällen aus zusammengesetzten Zuständen gebildet. Dabei werden Wertekombinationen zusammengefasst, bei denen sich ein Objekt der entsprechenden Klasse funktional äquivalent verhält. Die Verwendung primitiver Zustände ist oft weder möglich noch sinnvoll, da der zu modellierende Zustandsraum zu komplex ist, um jede Wertekombination als eigenen Zustand darzustellen. Durch die funktionale Äquivalenz vieler Wertekombinationen wäre ein solcher Zustandsautomat hochgradig redundant und äußerst schwer zu interpretieren.

Ob für den Zustandsautomaten abstrakte oder konkrete bzw. beschränkte oder rekursive Zustände verwendet werden, hängt von den Eigenschaften der modellierten Klasse und von der Verwendung des Zustandsautomaten ab.

Neben den Zuständen sind in Zustandsautomaten Zustandsübergänge, Ereignisse, Aktionen, Bedingungen und Aktivitäten enthalten. Zustandsübergänge legen fest, welche Folgezustände von einem Startzustand aus erreichbar sind. An Zustandsübergängen annotierte Ereignisse und Bedingungen bestimmen, wann ein Zustandsübergang stattfindet. Tritt ein entsprechendes Ereignis ein und ist die Bedingung erfüllt, dann erfolgt ein Zustandsübergang. Ist dem Zustandsübergang keine Bedingung zugeordnet, erfolgt der Übergang sobald das Ereignis eintritt. Während eines Zustandsübergangs können verschiedene Aktionen ausgeführt werden.

Während sich eine Komponente in einem Zustand befindet, kann sie Aktivitäten ausführen. Der Unterschied zwischen Aktivitäten und Aktionen besteht darin, dass Aktionen atomare Einheiten sind, die nicht durch ein Ereignis unterbrochen werden können, während dies bei Aktivitäten möglich ist.

Ein weiteres wichtiges Konzept von Zustandsautomaten sind Unterzustände, durch die eine hierarchische Strukturierung ermöglicht wird, was die Übersichtlichkeit erheblich steigern kann. Es gibt sequentielle und parallele Unterzustände. Mit Hilfe von parallelen Unterzuständen ist es möglich, Nebenläufigkeit zu modellieren.

---

[40] Für die Beschreibung eines abstrakten Zustandes werden nur Attribute verwendet, auf die mit öffentlichen Zugriffsmethoden zugegriffen werden kann.

[41] Für die Beschreibung eines konkreten Zustands werden alle für die Implementierung sichtbaren Attribute verwendet. Das sind in der Regel alle Attribute einer Klasse.

[42] Attribute, die zur Beschreibung eines Zustandes verwendet werden (abhängig von der Sichtbarkeit).

[43] Dieser Begriff ist nicht zu verwechseln mit der UML-Metaklasse `CompositeState`, mit deren Hilfe eine hierarchische Struktur innerhalb eines Zustandsautomaten aufgebaut werden kann.

Zustandsdiagramme werden sehr häufig als das zentrale Diagramm zur Beschreibung des Verhaltens von Objekten einer Klasse verwendet, da sie mit einer einfachen Notation die Möglichkeit bieten, auch sehr komplexes Verhalten übersichtlich und verständlich zu modellieren.

Testautomatisierung

Zustandsdiagramme sind die Grundlage für den zustandsbasierten Test. Dieser ist geeignet für alle Klassen, die in Abhängigkeit von ihrem Zustand bei gleichen Eingaben ein unterschiedliches Verhalten[44] zeigen.

Für die Testautomatisierung kann ein Zustandsdiagramm sehr gut zur Testfallermittlung und Testdatenerzeugung verwendet werden. Allerdings muss der dargestellte Zustandsautomat dafür gegenüber der UML-Spezifikation zusätzliche Bedingungen erfüllen. Für die Sollwertermittlung und Testauswertung sind Zustandsdiagramme nur bedingt geeignet. Es lässt sich zwar sehr gut überprüfen, ob die korrekten Aktionen ausgeführt wurden, aber die Überprüfung der Ausgaben des Testobjekts kann nur relativ ungenau sein, da Zustandsautomaten in den meisten Fällen nur mit Hilfe von zusammengesetzten Zuständen spezifiziert werden. Somit ist nur eine Aussage möglich, ob sich ein Objekt im korrekten zusammengesetzten Zustand befindet. Innerhalb dieses Zustandes sind aber durchaus Fehler möglich, die so nicht aufgedeckt werden können.

3.3.5 Aktivitätendiagramm

Aktivitätendiagramme werden ebenfalls dazu verwendet, dynamische Aspekte zu modellieren. Sie können mit Klassen, Schnittstellen, Komponenten, Knoten, Anwendungsfällen und Kollaborationen verknüpft werden. Am häufigsten wird ein Aktivitätendiagramm allerdings mit einer Operation verknüpft.

In einem Aktivitätendiagramm wird der Kontrollfluss zwischen verschiedenen Aktivitäten dargestellt. Dies ist auch ein wesentlicher Unterschied zu den Interaktionsdiagrammen. Bei diesen wird ein Kontrollfluss zwischen verschiedenen Komponenten modelliert.

Eine Aktivität ist eine nichtatomare Operation, die durch ein Ereignis unterbrochen werden kann. Sie ist strukturiert und kann sowohl aus Aktivitäten als auch aus Aktionen zusammengesetzt sein. Eine Aktion ist eine atomare Operation, die nicht durch ein Ereignis unterbrochen werden kann. Sie wird also immer vollständig ausgeführt. Aktivitäten bestehen also im Endeffekt immer aus einer Menge von aufeinander folgenden oder nebenläufigen Aktionen. Eine Aktion ist nicht strukturiert, d.h. sie kann nicht mehr in Teilaktionen unterteilt werden. Aktivitäten und Aktionen sind durch Zustandsübergänge miteinander verbunden. Sobald eine Aktivität oder Aktion abgeschlossen ist, erfolgt der Übergang zu einer Folgeaktivität bzw. –aktion. In Abbildung 3.7 wird ein Aktivitätendiagramm exemplarisch dargestellt.

Ein weiteres Konzept in Aktivitätendiagrammen sind Verzweigungen. Diese spalten einen Zustandsübergang auf. An den entstehenden Zweigen werden Bedingungen annotiert, anhand derer festgestellt wird, welcher Zweig, also welche Folgeaktivität bzw. –aktion ausgeführt wird.

[44] Dazu zählen sowohl die Verwendung von unterschiedlichen Algorithmen wie auch Einschränkungen hinsichtlich erlaubter Nachrichten in den verschiedenen Zuständen.

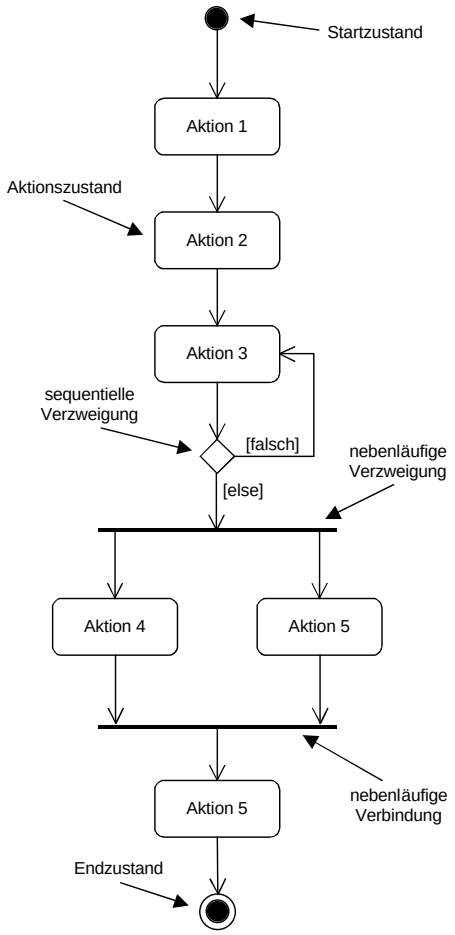


Abbildung 3.7 *Aktivitätendiagramm*

Nebenläufigkeit kann ebenfalls mit Hilfe von Aktivitätendiagrammen modelliert werden.

An Aktionen können sowohl abstrakter Text als auch konkrete Ausdrücke annotiert werden, die sogar der Syntax und Semantik der verwendeten Programmiersprache entsprechen können. Wird die zweite Möglichkeit konsequent zur Spezifikation aller Aktionen verwendet, so lässt sich aus einem Aktivitätendiagramm direkt ausführbarer Code generieren.

Testautomatisierung

Aktivitätendiagramme sind für die Verwendung beim Test nicht geeignet. Binder beschreibt in [Binder99] die Probleme, die sich für den Test aus der Definition von Zuständen über Aktivitäten und Aktionen[45] ergeben. Eine solche Zustandsdefinition ist nicht testbar, da für einen Zustand keine überprüfbare Zustandsinvariante angegeben werden kann, mit deren Hilfe festgestellt werden kann, ob sich ein Objekt in diesem Zustand befindet.

Für die automatische Erzeugung von Sollwerten sind Aktivitätendiagramme ebenfalls nicht geeignet. Werden an den Aktionen nur abstrakte Texte annotiert, können daraus keine konkreten Sollwerte berechnet werden. Werden dagegen mit den Aktionen konkrete ausführbare Ausdrücke verknüpft, kann zwar auf deren Basis eine ausführbare Operation generiert werden. Hier stellt sich allerdings die Frage, warum nicht gleich die

[45] Moore'sche Zustandsautomaten

generierte Operation als Implementierung verwendet wird. Hier geht es um die in Abschnitt 2.3.4 diskutierte Existenz eines perfekten Orakels. Wird eine aus einem Aktivitätendiagramm generierte Operation tatsächlich als Implementierung verwendet, die getestet werden soll, so kann das Aktivitätendiagramm nicht als Grundlage zur Ermittlung von Sollwerten verwendet werden. Die Operation würde gegen sich selbst getestet werden. Dadurch ließe sich aber kein einziger Fehler aufdecken, da sowohl die Implementierung als auch das Orakel eine identische Basis haben.

3.3.6 Implementierungsdiagramme

Implementierungsdiagramme werden dazu verwendet, verschiedene Aspekte der Implementierung zu modellieren. Dafür gibt es zwei Arten von Diagrammen: Komponentendiagramme und Deployment-Diagramme.

Komponentendiagramme dienen der Darstellung der statischen Implementierungssicht eines Systems. Mit ihrer Hilfe werden die Beziehungen zwischen verschiedenen physikalischen Bestandteilen eines ausführbaren Systems dokumentiert, wie z. B. ausführbaren Programmdateien, Bibliotheken, Tabellen einer Datenbank, Dateien und Dokumenten. Außerdem ist es möglich, die Struktur des Quellcodes darzustellen, d.h. die Zuordnung der Elemente der statischen Entwurfssicht[46] zu verschiedenen Quellcodedateien und deren Abhängigkeiten untereinander. Es können wiederum sowohl Beziehungen zwischen Quellcodedateien und den Bestandteilen des ausführbaren Systems als auch zwischen einzelnen Klassen und Schnittstellen und entsprechenden Teilen des ausführbaren Systems hergestellt werden, um die Zusammenhänge beim Übersetzungsprozess zu verdeutlichen. In Abbildung 3.8 wird ein Komponentendiagramm exemplarisch gezeigt.

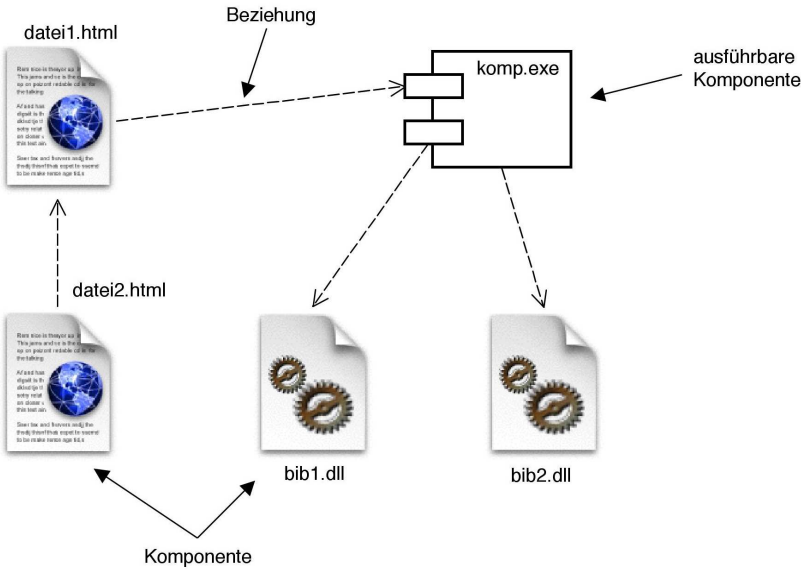


Abbildung 3.8 *Komponentendiagramm*

Die einzelnen Komponenten befinden sich zur Ausführungszeit auf verschiedenen sog. Knoten. Ein solcher Knoten stellt eine Hardwareeinheit dar, auf der Programme ausgeführt und Daten gespeichert werden können. Die Darstellung dieser Knoten und deren Beziehungen untereinander erfolgt in den **Deployment-Diagrammen**. Sie stellen die Struktur des Laufzeitsystems dar. Die Beziehungen zwischen den einzelnen Knoten

[46] Klassen, Schnittstellen

zeigen die physikalischen Kommunikationspfade zwischen diesen Hardwareeinheiten. In Abbildung 3.9 wird ein Deployment-Diagramm beispielhaft dargestellt.

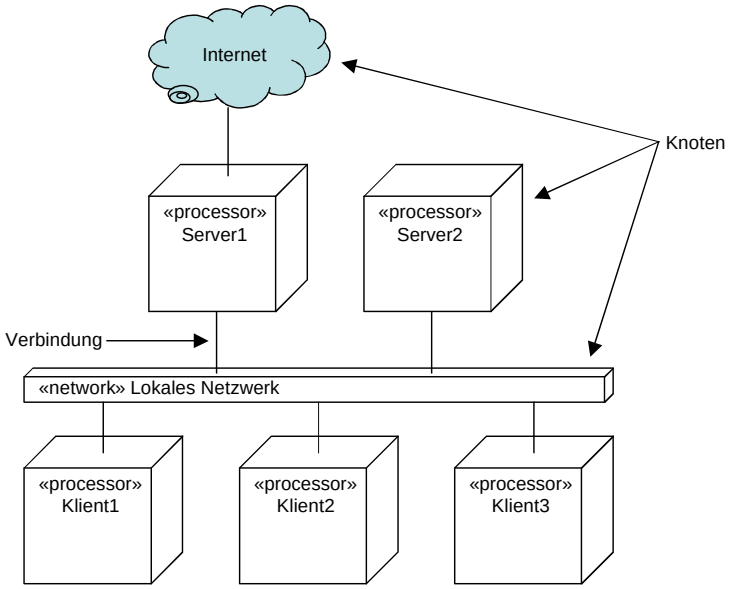


Abbildung 3.9 *Deployment-Diagramm*

Testautomatisierung

Für die automatische Quellcodegenerierung und Übersetzung eines ausführbaren Testprogramms enthalten diese Diagramme wichtige Informationen. Allerdings übersteigt deren Betrachtung den Rahmen dieser Arbeit, in der lediglich die Integration von Informationen zur Realisierung verschiedener Testmuster in ein UML-Modell untersucht werden soll.

3.4 Object Constraint Language (OCL)

Die Object Constraint Language (OCL) ist eine formale Sprache zur Spezifikation von Einschränkungen. Sie ist Bestandteil der UML[47] und wurde auch zur Spezifikation des UML-Metamodells selbst verwendet. Der Anwender, welcher UML-Modelle entwirft, kann die OCL verwenden, um Einschränkungen und Bedingungen zu spezifizieren, die mit den Mitteln der graphischen Notation der restlichen UML nicht oder nur sehr schwierig dargestellt werden können.

Die OCL ist eine Sprache zur Auswertung von Ausdrücken. Diese Ausdrücke haben keine Seiteneffekte, d.h. der Zustand eines Modells ändert sich nie auf Grund der Auswertung eines OCL-Ausdrucks. Es lassen sich allerdings sehr wohl Zustandsveränderungen beschreiben.

Die Möglichkeiten der Spezifikation, die durch die verschiedenen Diagramme der UML und deren Elemente angeboten werden, reichen oft nicht aus, um ein System genau genug zu spezifizieren. Auch wenn es grundsätzlich möglich ist, Invarianten sowie Vor- und Nachbedingungen in allen Diagrammen zu integrieren, reicht deren Spezifikation in natürlicher Sprache, wie sie oft angewendet wird, nicht aus, um vorhandene Beziehungen und Einschränkungen eindeutig zu definieren. Die entstehenden Mehrdeutigkeiten können zu einer Realisierung eines Systems führen, welche nicht den Anforderungen an

[47] seit UML-Version 1.1, September 1997

das zu entwickelnde System entspricht. Um diesem Problem zu begegnen, wurden formale Spezifikationssprachen entwickelt. Der Vorteil der OCL gegenüber anderen formalen Sprachen[48] besteht darin, dass sie trotz der zu Grunde gelegten mathematischen Theorie auch für Menschen mit einem weniger stark entwickelten mathematischen Hintergrund relativ leicht verständlich und einfach anwendbar ist.

Die OCL ist keine Programmiersprache. Es ist nicht möglich Programmlogik und Kontrollflüsse mit der OCL zu beschreiben. Es können keine Prozesse erzeugt werden. Die einzige Form von Operationen sind Abfragen, die einen Wert liefern. Allerdings handelt es sich bei der OCL um eine typisierte Sprache. Ausdrücke müssen den Typisierungsregeln der OCL folgen, d.h. es ist z. B. nicht möglich eine Zeichenkette mit einer Ganzzahl zu vergleichen.

Neben den in der OCL vordefinierten Typen repräsentiert jeder in einem UML-Modell definierte Bezeichner einen eigenen OCL-Typ. D.h., jede Klasse und jede Schnittstelle stellt einen eigenen Typ dar, deren Instanzen z. B. auch den Typregeln entsprechend miteinander verglichen werden können.

Die OCL kann für verschiedene Zwecke verwendet werden [OMG99]:

(1) Spezifikation von Invarianten für Klassen und Typen in einem Klassenmodell

(2) Spezifikation von Typinvarianten von Stereotypen

(3) Beschreibung von Vor- und Nachbedingungen von Operationen und Methoden

(4) Beschreibung von Bedingungen (z. B. an Zustandsübergängen in Zustandsdiagrammen)

(5) Spezifikation von Zustandsinvarianten in einem Zustandsdiagramm

Testautomatisierung

Der Einsatz der OCL hat eine wesentliche Bedeutung für den Klassentest und dessen Automatisierung. An vielen Stellen eines UML-Modells macht erst die Verwendung der OCL eine eindeutige Definition von Beziehungen, Einschränkungen und Bedingungen möglich. Ohne diese Eindeutigkeit ist eine Erzeugung von Testfällen, Testdaten und Sollwerten nur sehr schwer zu realisieren. OCL-Ausdrücke können in allen UML-Diagrammen integriert werden und sind damit auch sehr flexibel einsetzbar.

3.5 Erweiterungsmöglichkeiten der UML

Die grundlegenden Bestandteile der UML wurden so entworfen, dass mit ihrer Hilfe die Erfordernisse für den Entwurf eines breiten Spektrums an Softwaresystemen erfüllt werden können. Darüber hinaus bietet die UML Erweiterungsmöglichkeiten an, mit denen diese Basiselemente in einer definierten Art und Weise erweitert und an die jeweiligen Gegebenheiten angepasst werden können.

Mit der Beschränkung auf grundlegende, allgemein anwendbare Modellierungselemente bleibt die UML übersichtlich und leicht verständlich. Gleichzeitig ist sie durch die zur Verfügung gestellten Erweiterungsmechanismen flexibel adaptierbar für die Erfordernisse spezieller Anwendungen. Durch die genaue Definition der Erweiterungsmöglich-

[48] Objective-Z, VDM++

keiten bleibt ein Modell auch bei deren Verwendung verständlich und konsistent, vorausgesetzt, sie werden sehr überlegt und den Vorgaben entsprechend eingesetzt. Eine wesentliche Beschränkung der Erweiterungsmechanismen besteht darin, dass die Standardsemantik der UML nur erweitert werden darf. Eine Verletzung oder Negation dieser Grundregeln ist nicht erlaubt.

Es gibt vier Erweiterungsmechanismen innerhalb der UML:

(1) Stereotypen,

(2) Tag-Definitionen,

(3) Beschränkungen (Constraints) und

(4) Profile.

Mit Hilfe von **Stereotypen** können neue Modellierungselemente erzeugt werden. Diese basieren immer auf bereits existierenden Modellierungselementen und erweitern das Vokabular der UML. **Tag-Definitionen** werden dazu verwendet, die Eigenschaften eines bereits bestehenden Modellierungselements zu erweitern. Eine **Beschränkung** stellt eine Erweiterung der Semantik bereits bestehender Modellierungselemente dar. Es können neue Regeln hinzugefügt und bereits bestehende Regeln verändert werden. In einem **Profil** können die für einen bestimmten Anwendungsbereich definierten Stereotypen strukturiert modelliert und zur Verwendung bereitgestellt werden.

Ein Stereotyp ist ein Modellelement, welches zusätzliche, auf Tagdefinitionen basierende Eigenschaften, Beschränkungen und optional einer speziellen graphischen Repräsentation (Icon) definiert bzw. zusammenfasst. Bei der Verwendung von Stereotypen muss beachtet werden, dass diese auf der Ebene des UML-Metamodells angewendet werden. Wird also ein Modellelement mit einem Stereotypen markiert, so entsteht auf der Basis dieses Modellelements ein neues Modellelement im UML-Metamodell, welches zusätzlich zu seinen Standardeigenschaften die durch den Stereotypen definierten Eigenschaften erhält und für die Erstellung eines UML-Modells verwendet werden kann.

Eine Tagdefinition spezifiziert eine neue Eigenschaft, welche mit Modellelementen verknüpft werden kann. Die aktuellen Werte dieser Eigenschaften werden *Tagged Value* genannt. Sie können z. B. verwendet werden, um Organisations- oder Codegenerierungsinformationen darzustellen.

Durch Beschränkungen ist es möglich, eine zusätzliche Semantik für einzelne Modellelemente zu spezifizieren. Beschränkungen können mit Hilfe spezieller Sprachen, wie der OCL, beschrieben werden, aber auch mit Mitteln von Programmiersprachen, mathematischen Notationen oder natürlicher Sprache. Da Beschränkungen für beliebige Modellelemente spezifiziert werden können, ist es auch möglich, diese mit Stereotypen zu verknüpfen. Wird ein Modellelement mit einem derartigen Stereotypen gekennzeichnet, muss dieses Modellelement alle für diesen Stereotypen festgelegten Beschränkungen einhalten. In Abbildung 3.10 wird die Definition eines Stereotypen beispielhaft dargestellt.

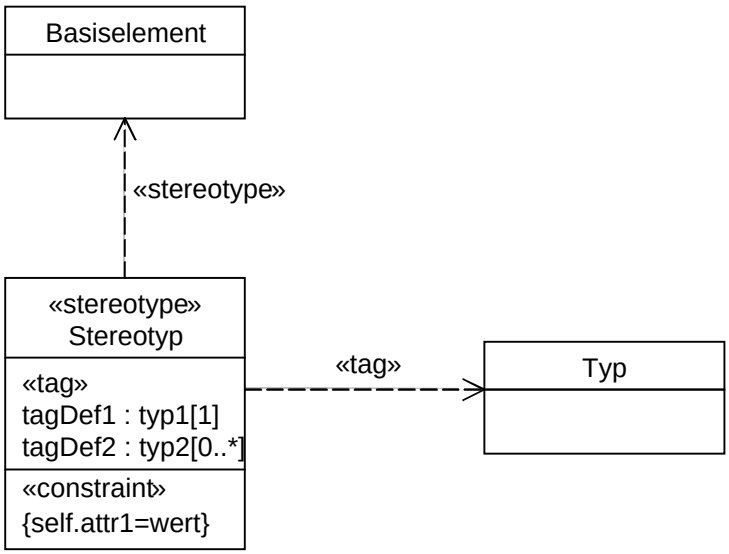


Abbildung 3.10 *Definition eines Stereotypen*

Testautomatisierung

Durch den Einsatz der erläuterten Erweiterungsmöglichkeiten der UML kann der Entwurfsprozess testbarer UML-Modelle wesentlich effizienter gestaltet werden. Es können spezielle, für die Erfordernisse des Tests angepasste, Stereotypen entworfen werden. Diese lassen sich dann mit Modellelementen eines UML-Modells verbinden, um ein bestimmtes Testmuster zu realisieren. Diese Herangehensweise stellt die Grundlage für den in dieser Arbeit vorgestellten Ansatz dar, welcher im folgenden Kapitel eingehend erläutert wird.

Kapitel 4

Erstellung testbarer UML-Modelle

4.1 Einführung

In Abschnitt 2.5 wurde bereits erläutert, dass eine wesentliche Schwäche verfügbarer Testwerkzeuge darin liegt, dass sie nur einzelne Testaktivitäten oder Testmuster unterstützen. Dadurch können sie nur in einem sehr begrenzten Bereich verwendet werden. Ein umfassender Test erfordert dann den Einsatz verschiedener Testwerkzeuge, was einen erheblichen Mehraufwand bei der Erzeugung und Verwaltung einer entsprechenden Testumgebung und der Gewährleistung der Konsistenz dieser Umgebung zur Folge hat.

Wird durch ein Testwerkzeug eine der Testaktivitäten Testfallermittlung, Testdatenerzeugung oder Sollwertbestimmung unterstützt, impliziert dies die Verwendung einer formalen Spezifikationssprache [Rüppel97]. Die von solchen Testwerkzeugen verwendeten Spezifikationssprachen sind aber meistens sehr spezifisch und nur in einem sehr begrenzten Kontext einsetzbar. Außerdem wird dadurch neben der Sprache zur Spezifikation des Softwaresystems eine zweite Spezifikationssprache in den Entwicklungsprozess eingeführt. Diese wird zur Spezifikation von Informationen verwendet, die abhängig oder sogar identisch gegenüber den zur Spezifikation des Softwaresystems verwendeten Informationen sind. Auch hier ergibt sich ein erhöhter Aufwand hinsichtlich Erstellung und Konsistenzgewährleistung für die Testumgebung. In jedem Fall verschlechtert sich die Qualität und Effizienz des Tests, wenn die Konsistenz der Informationen nicht sichergestellt werden kann.

Aus diesen Erkenntnissen ergibt sich die Notwendigkeit, den Testprozess stärker in den restlichen Entwicklungsprozess zu integrieren. Dabei spielt die Unterstützung durch entsprechende Testwerkzeuge, die eine bessere Integration zulassen als die zurzeit verfügbaren Werkzeuge, eine wesentliche Rolle. Hier stellt sich die Frage, an welcher Stelle und in welcher Form eine solche Integration in den Softwareentwicklungsprozess am sinnvollsten möglich ist. Die oben erwähnte Problematik der Verwendung mehrerer Spezifikationssprachen legt nahe, sich auf eine Spezifikationssprache zu beschränken. Da die für die Spezifikation des Tests verwendeten Sprachen sehr enge Grenzen für ihre Verwendung haben, können diese dabei nicht eingesetzt werden. Die UML dagegen ist eine sehr flexibel einsetzbare und erweiterbare Spezifikationssprache. Zusätzlich enthält sie mit der OCL eine formale Spezifikationssprache, die eine sehr weitreichende Unterstützung bei der Spezifikation von für den Test notwendigen Informationen anbietet. Außerdem ist die UML offiziell als Standard für die Spezifikation von Softwaresystemen anerkannt und auch in der Praxis etabliert. Eine darauf beruhende Modellierung der Testumgebung kann somit auch wesentlich einfacher eingeführt werden, was einen nicht zu vernachlässigenden Aspekt darstellt.

Die UML stellt die in Abschnitt 3.5 bereits vorgestellten Erweiterungsmechanismen zur Verfügung. Dadurch können die Ausdrucksmöglichkeiten der UML für spezielle Anwendungsgebiete erweitert und angepasst werden. Diese Möglichkeiten werden in dieser Arbeit verwendet, um eine Basis für die gemeinsame Spezifikation von Entwurfs- und Testinformationen in einem Modell zu schaffen. Durch diese Zusammenführung von sich ergänzenden und teilweise identischen Informationen können Redundanzen vermieden, die Konsistenz einfacher garantiert und im Softwaremodell vorhandene Inkonsistenzen leichter identifiziert und beseitigt werden. Testwerkzeuge sollten dann in der Lage sein, weitestgehend ohne menschliche Interaktionen aus den in dem Modell enthaltenen Informationen ausführbare Testfälle zu generieren.

Ein Modell, welches diese Bedingungen erfüllt, wird in [Binder99] als ein testbares Modell bezeichnet. Um die Erstellung eines solchen testbaren Modells möglichst effizient zu gestalten, wird in dieser Arbeit ein sogenanntes Profil erstellt. Ein Profil ist eine einfache Möglichkeit, anwendungsspezifische Erweiterungen der UML zusammenzufassen und zur Verfügung zu stellen. Die Erweiterungen werden innerhalb eines Profils durch Stereotypen definiert, welche die Eigenschaften und die Semantik der neuen UML-Elemente festlegen. Diese Möglichkeiten der UML wurden bereits in Abschnitt 3.5 kurz erläutert. Eine ausführliche Beschreibung ist in [Booch+99] und [Booch+99a] zu finden. Da die in dem in dieser Arbeit erstellten Profil enthaltenen Erweiterungen dem Anwendungsgebiet Test zuzuordnen sind, wird das Profil als **Testprofil** bezeichnet.

Mit den durch das Testprofil zur Verfügung gestellten Strukturen und Elementen können Anwender eigene Stereotypen definieren, die jeweils ein Testmuster repräsentieren. Diese Stereotypen enthalten dann die für die automatische Generierung von Testfällen notwendigen Informationen bzw. fordern die Einhaltung bestimmter Bedingungen bei der Modellierung von verschiedenen Elementen. Sie können auf einzelne Testobjekte[49] angewendet werden. Als Vorlage für die Definition von Testmustern innerhalb des Testprofils kann der Anwender z. B. die in [Binder99] vorgestellten Testmuster für den Klassentest verwenden. Aber auch andere Testmethoden können mit Hilfe des Testprofils in entsprechenden Stereotypen realisiert und dabei den jeweiligen Anforderungen angepasst werden. Erreicht wird dieser hohe Grad an Flexibilität durch die Übertragung des in [Rüppel97] vorgestellten Ansatzes eines Frameworks zur Konstruktion von Testwerkzeugen auf die Spezifikation der für den Test notwendigen Informationen. Dadurch kann das Problem der mangelnden Flexibilität bisheriger Testwerkzeuge hinsichtlich unterstützter Testmuster beseitigt werden.

In Abschnitt 4.2 wird zunächst ein Überblick gegeben, welchen Einfluss die Vererbung auf die Struktur objektorientierter Systeme besitzt und welche Auswirkungen dies insbesondere für den Klassentest solcher Systeme hat. Daran anschließend wird in Abschnitt 4.3 die grundsätzliche Struktur des Testprofils erläutert. In den beiden darauf folgenden Abschnitten erfolgt eine detaillierte Beschreibung der einzelnen Elemente des Testprofils und im abschließenden Abschnitt wird dessen Anwendung beispielhaft gezeigt.

4.2 Zusammenhang zwischen Vererbung und Test

In Abschnitt 2.2.1 wurde bereits gesagt, dass die Vererbung mit den daraus entstehenden Möglichkeiten der Polymorphie ein grundlegendes Merkmal der Objektorientierung

[49] Im Rahmen dieser Arbeit handelt es sich dabei um Methoden und Klassen.

ist und diese starke Auswirkungen auf den Test entsprechender Systeme hat. Durch die Vererbung wird der strukturierte und modulare Entwurf eines Systems wesentlich vereinfacht und es entstehen neue Möglichkeiten hinsichtlich der Wiederverwendung von Komponenten. Allerdings bringt die Vererbung auch Probleme mit sich, die in anderen Programmierparadigmen nicht anzutreffen sind. So kann es bei der Verwendung von geerbten Eigenschaften im Kontext der Unterklasse zu Fehlern kommen, die im Kontext der Oberklasse nicht aufgetreten sind. Dies gilt insbesondere dann, wenn einzelne Methoden der Oberklasse in der Unterklasse überschrieben wurden. Durch eine entsprechende Datenkapselung ist die gemeinsame Verwendung von geerbten und überschriebenen Methoden in einer Unterklasse kaum zu vermeiden. In [Jacobson+92] werden die zwei grundlegenden Ursachen dafür angegeben, warum eine geerbte Methode im Kontext der Unterklasse nicht korrekt funktioniert:

(1) Wenn die Unterklasse Instanzvariablen verändert, für die eine geerbte Methode bestimmte Werte erwartet und

(2) wenn geerbte Methoden überschriebene Methoden aufrufen.

Um diesen Problemen zu begegnen, wird die Zusicherung bestimmter Eigenschaften für die Beziehungen zwischen Ober- und Unterklassen gefordert. Diese können aus einer in der Literatur häufig unter dem Begriff *Liskovs Substitution Principle (LSP)* verwendeten Forderung abgeleitet werden. Danach soll es möglich sein überall dort, wo eine Instanz einer Klasse erwartet wird, diese durch eine Instanz irgendeiner ihrer Unterklassen zu ersetzen.

Im Folgenden werden die in der Literatur[50] aus dieser Forderung abgeleiteten zuzusichernden Eigenschaften aufgezählt:

(1) Die Klasseninvariante einer Unterklasse kann nur äquivalent oder stärker, aber nicht schwächer als die Klasseninvariante ihrer Oberklasse sein.

(2) Eine Vorbedingung einer überschriebenen Operation kann nur schwächer oder äquivalent, aber nicht stärker als die Vorbedingung der überschriebenen Operation sein.

(3) Die Nachbedingung einer überschriebenen Operation kann nur stärker oder äquivalent, aber nicht schwächer als die Nachbedingung der überschriebenen Operation sein.

Zur Illustration wird hier das in Abbildung 4.1 dargestellte Beispiel verwendet, welches aus [Warmer+99] entnommen wurde.

[50] [Binder99] [Warmer+99]

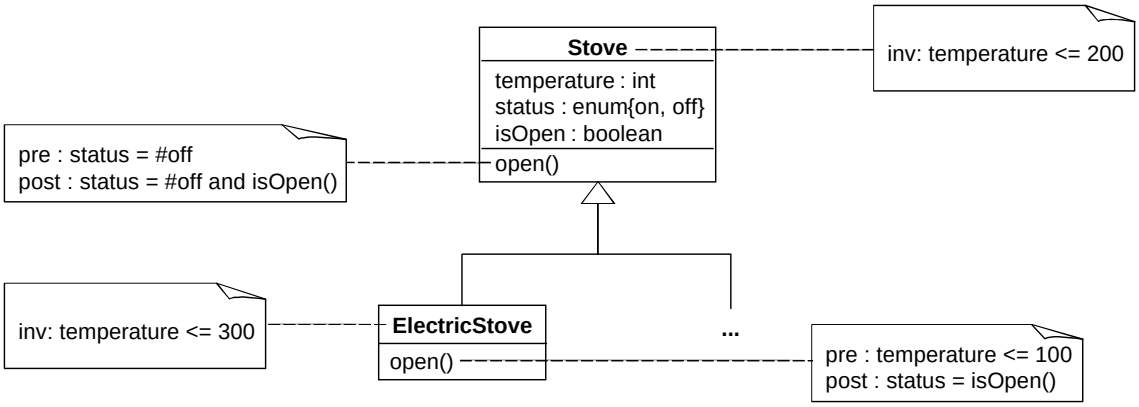


Abbildung 4.1 *Vererbung und Klasseninvarianten sowie Vor- und Nachbedingungen*

Für die allgemeinere Klasse `Stove` wird durch die Klasseninvariante festgelegt, dass die Temperatur eines Ofens nie 200°C überschreitet. Auf diese Information verlassen sich alle Klienten dieser Klasse. Die Klasseninvariante der abgeleiteten Klasse `ElectricStove` besagt dagegen, dass die Temperatur eines elektrischen Ofens nie 300°C überschreitet. Für alle Klienten der Klasse `Stove` ist dies problematisch, da sie nur mit Temperaturen von bis zu 200°C rechnen. Weder in dem modellierten Softwaresystem noch in der Realität ist es möglich, an Stellen, wo allgemein ein Ofen eingesetzt werden soll, einen elektrischen Ofen sicher zu verwenden. In der Realität sind dann u.U. die Sicherheitsabstände zu den den Ofen umgebenden Gegenständen so gering, dass sie für Temperaturen bis zu 200°C ausreichend sind, dies für 300°C aber nicht mehr gewährleistet ist. Innerhalb des Softwaresystems kann für einen Klienten, der ein Serverobjekt vom Typ `Stove` erwartet, dieses Objekt einen Zustand erreichen, den der Klient nicht mehr interpretieren kann. Dies kann genau dann passieren, wenn das Serverobjekt vom Typ `ElectricStove` ist.
Wichtig dabei ist, dass immer der gesamte mögliche Zustandsraum als Basis betrachtet wird. Dieser wird aus sämtlichen Instanzvariablen einer Klasse gebildet. Für das angegebene Beispiel sind das also `temperature`, `status` und `isOpen`. In Abbildung 4.2 werden für die beiden Klassen `Stove` und `ElectricStove` jeweils die Klasseninvarianten, die Vorbedingungen und die Nachbedingungen vergleichend dargestellt.

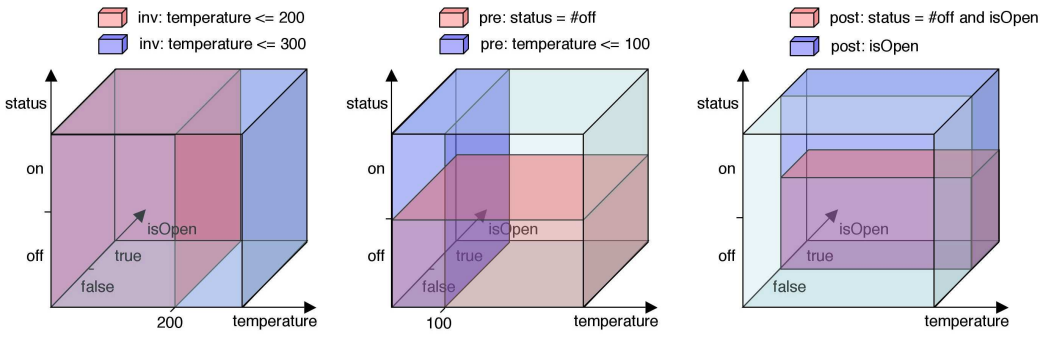


Abbildung 4.2 *Klasseninvariante, Vor- und Nachbedingungen der Klassen* `Stove` *und* `ElectricStove`

Im linken Teil ist das Verhältnis zwischen den beiden Klasseninvarianten zu sehen. Der Zustandsraum für die Klasse `Stove` ist eine echte Teilmenge des Zustandsraums der abgeleiteten Klasse `ElectricStove`, was nicht konform zum *LSP* ist. Dies kann zu den bereits erläuterten Problemen führen. In der Mitte sind die Vorbedingungen der

Methode `open` der Oberklasse und die der überschriebenen Methode `open` der Unterklasse dargestellt. Dadurch, dass die Vorbedingung der überschreibenden Methode der Unterklasse nur einen Teil des Zustandsraumes der Vorbedingung der Methode der Oberklasse abdeckt, kann die Methode der Unterklasse nicht überall dort aufgerufen werden, wo dies für die Methode der Oberklasse möglich ist. Dies stellt für Klienten der Klasse `Stove` ein Problem dar, da diese beim Aufruf der Methode von der Vorbedingung der Klasse `Stove` ausgehen und diese somit u.U. aufrufen, obwohl die Vorbedingung der Klasse `ElectricStove` dies nicht zulässt. Im rechten Teil von Abbildung 4.2 sind die beiden Nachbedingungen der jeweiligen Methoden `open` zu sehen, wobei zu erkennen ist, dass die Methode der Klasse `ElectricStove` Werte zurückgeben kann, die für die `open`-Methode der Klasse `Stove` nicht als Rückgabewerte möglich sind. Da Klienten der Klasse `Stove` nur zwingend die durch die Nachbedingung der `open`-Methode dieser Klasse abgedeckten Rückgabewerte interpretieren und verarbeiten können müssen, entsteht ein Problem, sobald ein Serverobjekt vom Typ `ElectricStove` ist und dessen `open`-Methode Werte zurückgibt, die nicht durch die Nachbedingung der `open`-Methode der Klasse `Stove` abgedeckt sind.
Für modale Klassen müssen analog Bedingungen für deren Zustandsautomaten erfüllt werden. In [Binder99] werden eine Reihe von Bedingungen für die Beziehungen zwischen dem Zustandsautomaten einer Oberklasse und dem Zustandsautomaten einer von dieser Oberklasse abgeleiteten Klasse angegeben. Diese Bedingungen sind neben weiteren Forderungen ein Bestandteil des von Binder entwickelten *FREE-State Model* und lassen sich teilweise aus den oben erläuterten Regeln ableiten. Das *FREE-State Model* wird ausführlich mit einigen Erweiterungen in Abschnitt 4.5.4 erläutert.

Wichtig ist, dass für einen umfassenden Test einer abgeleiteten Klasse diese immer sowohl gegen die eigene als auch gegen die Spezifikation ihrer Oberklasse(n) getestet werden muss [Binder99]. Sind dabei die oben erläuterten allgemeinen Bedingungen hinsichtlich der Klasseninvarianten sowie der Vor- und Nachbedingungen und die im Rahmen des *FREE-State Model* für modale Klassen aufgestellten Anforderungen erfüllt, so kann zumindest in Bezug auf die konformen Schnittstellen von einem sicheren Einsatz der Möglichkeiten polymorpher Strukturen ausgegangen werden. Außerdem ist es dann sogar möglich Testfälle, die für den Test einer Oberklasse erzeugt wurden, auch für den Test aller von dieser Klasse abgeleiteten Klassen zu verwenden, was die Effizienz des Tests erheblich steigern kann. Das Testprofil wurde so konstruiert, dass es bei seiner Anwendung gewährleistet, dass diese Bedingungen erfüllt werden.

4.3 Die Struktur des Testprofils

Ein Framework bietet für ein bestimmtes Anwendungsgebiet eine grundlegende Lösung an, in der die für dieses Anwendungsgebiet invarianten Bestandteile bereits realisiert sind. Zusätzlich werden Schnittstellen angeboten, an denen das Framework in einer definierten Art und Weise um für eine konkrete Anwendung spezifische Bestandteile erweitert werden kann [Rüppel97]. Diese Struktur ermöglicht es, dass sich ein Entwickler auf die anwendungsspezifischen Bestandteile konzentrieren kann, während er für die konstanten Teile des entsprechenden Anwendungsgebietes die durch das Framework zur Verfügung gestellten Lösungen nutzen kann. Ein weiterer Vorteil dieses Ansatzes ist die Gewährleistung der Konsistenz der durch das Framework zur Verfügung gestellten Strukturen.

Für eine Anwendung dieser Prinzipien auf ein Profil für den objektorientierten Test ist es also notwendig, die invarianten Bestandteile des Tests zu identifizieren und eine

generische Lösung für diese zu modellieren. Gleichzeitig müssen Schnittstellen in dem Profil angeboten werden, an denen es um spezifische Elemente erweitert werden kann. Die Trennung von invarianten und varianten Anteilen des Testprozesses war bereits Gegenstand von [Rüppel97]. Allerdings kann die dort getroffene Aufteilung nicht in der gleichen Form übernommen werden, da sie sich an der Realisierung von Testwerkzeugen orientiert. Dort werden z. B. allgemeine Modelle von Testfällen und Testfallsammlungen durch sogenannte Kernklassen repräsentiert, welche die invarianten Anteile des Tests realisieren. Testfälle und Testfallsammlungen sind für die Erstellung des Testprofils aber nicht relevant, da das Profil nur gewährleisten soll, dass die für bestimmte Testmodelle erforderlichen Informationen in einer definierten Form in einem UML-Modell enthalten sind. Auf dieser Ebene werden einzelne Testfälle nicht betrachtet, da die Testfallerzeugung selbst erst die Aufgabe des Testwerkzeugs ist. Die für das Testprofil wesentlichen invarianten Bestandteile des Tests ergeben sich aus den horizontalen Testaktivitäten sowie aus invarianten Bestandteilen von objektorientierten Testmustern.

Wie bereits erwähnt, soll das Testprofil ein hohes Maß an Flexibilität bieten, womit ein wesentliches Problem verfügbarer Testwerkzeuge, deren mangelnde Anpassbarkeit, beseitigt werden kann. Dies geschieht auf mehreren Ebenen. Auf der einen Seite fließen allgemeine Anforderungen des objektorientierten Prozesses in die Gestaltung des Testprofils mit ein, wodurch es in einem sehr breiten Anwendungsbereich einsetzbar wird. Auf der anderen Seite ist das Testprofil modular aufgebaut. Wie bereits in Abschnitt 3.5 erläutert, stellt ein Profil eine Menge von Stereotypen zu einem bestimmten Anwendungsbereich zur Verfügung. Im Testprofil werden grundsätzlich zwei Arten von Stereotypen unterschieden. Es gibt Stereotypen, die allgemeine Regeln zur Konstruktion von objektorientierten Testmustern definieren. Diese werden als **Testmusterstereotypen** bezeichnet. Außerdem gibt es die sog. **Komponentenstereotypen**, aus denen der Anwender im Rahmen der in den Testmusterstereotypen definierten Regeln die Testmuster nach seinen Erfordernissen zusammensetzen und konfigurieren kann.

Um eine wirkliche Anpassbarkeit gewährleisten zu können, wurde das Testprofil so gestaltet, dass es möglich ist, im Rahmen der vorgegebenen Strukturen zusätzliche Komponentenstereotypen zu erstellen und somit das Testprofil zu erweitern. Diese zusätzlichen Stereotypen können dann wie die bereits im Testprofil enthaltenen Komponentenstereotypen zur Konstruktion von Testmustern mit Hilfe der Testmusterstereotypen verwendet werden. Die grundlegende Struktur und die Konstruktion von Testmustern durch den Anwender sind in Abbildung 4.3 schematisch dargestellt.

Der Benutzer kann konkrete Testmuster konstruieren, indem er durch Vererbung neue Stereotypen von den abstrakten Testmusterstereotypen ableitet. Dabei muss er die in den Testmusterstereotypen definierten Regeln[51] anwenden. Diese Regeln beschreiben, welche Eigenschaften[52] für ein konkretes Testmuster definiert werden müssen. Für die Definition dieser Eigenschaften und die Festlegung ihrer spezifischen Ausprägungen für das jeweilige Testmuster verwendet der Benutzer die Komponentenstereotypen[53]. Vom Benutzer konstruierte Testmusterstereotypen können dann auf ein oder mehrere Testobjekte angewendet werden, wodurch festgelegt wird, dass dieses Testobjekt mit diesem speziellen Testmuster getestet wird.

[51] Beschränkungen (Constraints), in Abbildung 4.3 durch $C_1 - C_4$ dargestellt
[52] in Abbildung 4.3 durch $P_1 - P_4$ dargestellt
[53] In Abbildung 4.3 durch $K_1 - K_4$ dargestellt

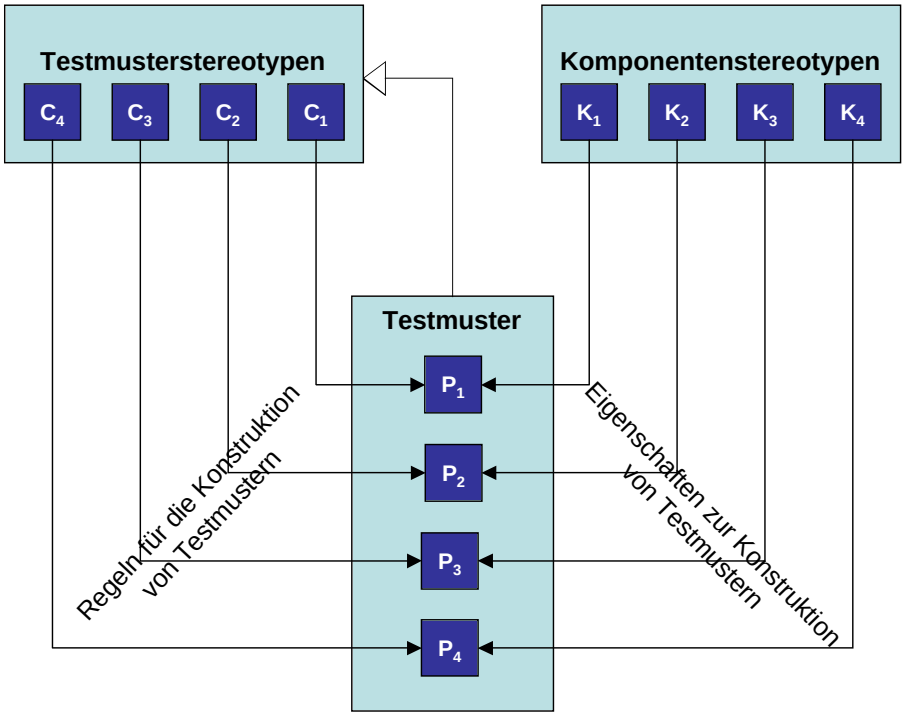


Abbildung 4.3 *Struktur und Konstruktionsprozess im Testprofil*

Ein Aspekt, der für den Klassentest einen sehr großen Einfluss bei der Konstruktion des gesamten Testprofils hat, ist die Sichtbarkeit der für ein Testmuster verwendeten Informationen. Dabei gibt es Informationen, die für die Klienten einer Klasse über die öffentliche Schnittstelle sichtbar sind. Zusätzlich gibt es Informationen, die nur für die Implementierung einer Klasse selbst sichtbar sind. In Abschnitt 3.3.4 wurde diese Klassifikation bereits für die Beschreibung von abstrakten und konkreten Zuständen[54] in Zustandsautomaten erläutert. Da für die Testautomatisierung die Konsistenz der zu Grunde gelegten Informationen von entscheidender Bedeutung ist, und starke Abhängigkeiten zwischen den in den verschiedenen UML-Diagrammen dargestellten Elementen bestehen, ist eine Berücksichtigung der Sichtbarkeit der für den Test zu verwendenden Daten bei der Konstruktion des Testprofils unumgänglich. Innerhalb des in dieser Arbeit erstellten Testprofils wird nur die Erzeugung abstrakter Testmuster betrachtet. Diese Testmuster verwenden nur über die öffentliche Schnittstelle einer Klasse zugängliche Informationen. Auf dieser Basis bleibt ein Test auf die Überprüfung öffentlicher Eigenschaften beschränkt. Die korrekte Übertragung der Werte von der öffentlichen Schnittstelle auf die einzelnen Instanzvariablen und umgekehrt wird dabei nicht betrachtet. Die internen Eigenschaften der Klasse, und somit auch die Zuordnung zwischen öffentlicher Schnittstelle und Instanzvariablen, werden zwar bei einem solchen Test mit ausgeführt und es können auch entsprechende Fehler aufgedeckt werden. Dies geschieht aber nur für die öffentlichen Eigenschaften systematisch. Die darüber hinaus gefundenen, auf den internen Eigenschaften beruhenden Fehler sind nur ein Nebenprodukt. Um auch die für die Implementierung sichtbaren Elemente systematisch testen zu können, ist eine entsprechende Anpassung des Testprofils notwendig. Die grundlegende Struktur des Testprofils kann dabei erhalten bleiben. Lediglich die für den Test im

[54] Diese Unterscheidung in *abstrakt* und *konkret* darf nicht mit dem Konzept *abstrakter* Modellelemente in der UML verwechselt werden. Während hier die Sichtbarkeit von Informationen gemeint ist, werden in der UML Modellelemente als *abstrakt* deklariert, von denen keine Instanzen erzeugt werden können.

UML-Modell zur Verfügung zu stellenden Informationen müssen entsprechend erweitert werden.

Im Rahmen dieser Arbeit wird davon ausgegangen, dass der Zugriff auf Instanzvariablen einer Klasse ausschließlich über entsprechende öffentliche Zugriffsmethoden der Klasse geschieht. Der abstrakte Zustand einer Instanz definiert sich also nur aus Rückgabe- und Ausgangsparametern von öffentlichen, zustandserhaltenden Methoden einer Klasse. Um den Test automatisieren zu können, muss an dieser Stelle eine weitere Einschränkung gemacht werden. Die Methoden, deren Rückgabe- und Ausgangsparameter zur Definition des abstrakten Zustands verwendet werden sollen, dürfen keine Eingangs- oder Durchgangsparameter besitzen. Ansonsten könnten Rückgabe- und Ausgangsparameter in Abhängigkeit von diesen Parametern bei gleichem internen Zustand verschiedene Werte liefern. Dies würde eine automatische Generierung von Sollwerten unmöglich machen[55].

Für die Definition des abstrakten Zustands ist es unerheblich, ob die verwendeten Rückgabe- und Ausgangsparameter direkt den Wert eines Attributs widerspiegeln oder aus diesen berechnet werden oder sogar eine ganz andere Quelle besitzen, wie z. B. einen in einer Datei gespeicherten Wert. Diese völlige Abstraktion und klare Trennung von den Attributen ist ein wesentlicher Unterschied zur Anwendung des Begriffs der abstrakten Schnittstelle in der Literatur. Dort werden zur Definition von Zustandsautomaten, Vor- und Nachbedingungen, Klasseninvarianten u.ä. durchgängig direkt Attribute verwendet, deren Wert in der Regel ausschließlich über öffentliche Methoden abgefragt werden kann. Für die Testbarkeit stellt dies ein wesentliches Hindernis dar. Ohne eine direkte Zuordnung von Attributen zu Rückgabe- und Ausgangsparametern von öffentlichen, zustandserhaltenden Methoden kann die Testbarkeit nicht gewährleistet werden.

Ein wesentlicher Aspekt für die Ausführbarkeit von Einschränkungen ist, dass diese keine Seiteneffekte besitzen, also zustandserhaltend sind. Dies ist notwendig, da Klienten direkt diese Einschränkungen verwenden, um z. B. festzustellen, ob die Vorbedingung einer Methode erfüllt ist. Dabei darf der Zustand eines Objekts nicht verändert werden. Da eine Einschränkung insgesamt keinen Seiteneffekt besitzen darf, gilt dies insbesondere auch für alle ihre Bestandteile. Das heißt, dass alle zur Definition der Einschränkung verwendeten Methoden zustandserhaltend sein müssen, was beim korrekten und konsequenten Einsatz des Testprofils zugesichert werden kann.

Die klare Trennung zwischen den an der öffentlichen Schnittstelle zugänglichen Informationen und dem konkreten Zustand, also den Werten von Instanzvariablen, hat zur Folge, dass der Zustand einer Instanz nicht direkt durch die Manipulation von Attributen verändert werden kann, sondern nur durch die Anwendung öffentlicher Methoden. Die Modellierung des Zusammenhangs zwischen einem primitiven Zustand[56] und einer konkreten Methodensequenz, die genau diesen primitiven Zustand erzeugt, übersteigt den Rahmen dieser Arbeit. Für den Test bedeutet dies, dass ein Testfall immer aus einer Methodensequenz bestehen muss, die ein Objekt neu erzeugt und dann eine Sequenz von Operationen darauf anwendet. Die Spezifikation eines primitiven Anfangszustands reicht hier nicht aus, da daraus keine entsprechende Methodensequenz automatisch generiert werden kann. Daraus folgt, dass der Test einer einzelnen Methode auf der Basis der öffentlichen Schnittstelle in dieser Arbeit nicht betrachtet wird. Durch eine Erweiterung des Testprofils um die Modellierung des Zusammenhangs eines primitiven,

[55] siehe Abschnitt 4.5.3
[56] siehe Abschnitt 3.3.4

abstrakten Zustands und einer ihn erzeugenden Methodensequenz, können auch methodenspezifische Testmuster sinnvoll angewendet werden. Das Gleiche gilt für eine Erweiterung auf der Ebene der für die Implementierung sichtbaren Informationen.

4.4 Testmusterstereotypen

Innerhalb des Testprofils werden die Testmusterstereotypen durch die im Paket `Test Pattern`[57] enthaltenen Stereotypen repräsentiert. Aus den im letzten Abschnitt genannten Gründen werden im Rahmen dieser Arbeit nur **klassenspezifische Testmuster** betrachtet. Repräsentiert werden diese durch den abstrakten Stereotypen «`ClassScopeTestPattern`» bzw. den davon abgeleiteten Stereotypen. Diese Stereotypen können auf Modellelemente der Metaklasse `Class` und `Interface` angewendet werden.

Durch die weitere Unterteilung des oben genannten Stereotypen in die ebenfalls abstrakten Stereotypen «`DirectClassScopeTestPattern`» und «`ReusableClassScopeTestPattern`» werden verschiedene Formen der Anwendung von Testmusterstereotypen auf die Testobjekte unterstützt. Grundsätzlich lassen sich von den «`ReusableClassScopeTestPattern`» abgeleitete Stereotypen fast so wie sie vom Benutzer definiert wurden auf mehrere Testobjekte anwenden. Sie können also weitestgehend wiederverwendet werden. Der Einsatz der von den Stereotypen «`DirectClassScopeTestPattern`» abgeleiteten Testmusterstereotypen erfordert hingegen für die Anwendung auf jedes Testobjekt eine erneute Festlegung der konkreten Werte der Eigenschaften des Stereotypen. Hier können also nur die im Testmusterstereotypen definierten Eigenschaften, also eine vordefinierte Testmusterstruktur, wiederverwendet werden, im Gegensatz zur zusätzlichen Wiederverwendung der konkreten Werte der Eigenschaften bei den «`ReusableClassScopeTestPattern`».

Realisiert werden die beiden Anwendungskonzepte direkt anzuwendender und wiederverwendbarer Testmusterstereotypen durch eine unterschiedliche Art der Deklaration der Eigenschaften der Testmuster in den Stereotypen durch Komponentenstereotypen. Um ein direkt anzuwendendes Testmuster zu erzeugen, muss der Benutzer bestimmte, durch in den abstrakten Testmusterstereotypen enthaltenen Regeln festgelegte, Eigenschaften des Testmusters als Tagdefinitionen deklarieren. Eventuelle Parameter dieser Eigenschaften, repräsentiert durch Tagdefinitionen in den jeweiligen Komponentenstereotypen, werden dann bei jeder Anwendung des Testmusterstereotypen auf ein Testobjekt durch konkrete Werte erneut definiert. Soll ein wiederverwendbares Testmuster erstellt werden, muss der Benutzer die gleichen Eigenschaften in dem konkreten Testmusterstereotypen festlegen. Allerdings erfolgt dies in diesem Fall durch die Markierung des zu erzeugenden Testmusterstereotypen mit den entsprechenden Komponentenstereotypen. Es werden also Stereotypen auf einen anderen Stereotypen angewendet. Dabei werden auch gleich die Tagdefinitionen der Komponentenstereotypen mit Tagwerten belegt. Durch die sofortige Festlegung der Tagwerte bei der Definition des Testmusterstereotypen wird erreicht, dass diese Werte bei jeder Anwendung auf ein neues Testobjekt bereits definiert sind und nicht jedes Mal wieder neu eingegeben werden müssen. Konkrete Anwendungsbeispiele für die beiden Formen von Testmusterstereotypen werden in Kapitel 5 gezeigt. Insgesamt sollte festgehalten werden, dass es sich bei der Aufteilung in direkt anzuwendende und wiederverwendbare Testmusterstereotypen lediglich um eine technische Frage der Anwendung der Stereotypen handelt. Diese

[57] siehe Anhang A.1

besitzt inhaltlich für den Test keine Relevanz, kann den Einsatz des Testprofils aber wesentlich vereinfachen.

Innerhalb der abstrakten Testmusterstereotypen sind, wie bereits erwähnt, Regeln für die Ableitung von konkreten Testmusterstereotypen definiert[58]. Diese Regeln bestimmen die Eigenschaften, die bei der Konstruktion von konkreten Testmusterstereotypen durch den Benutzer für diesen Stereotypen zu definieren sind. Es gibt testmusterspezifische und testobjektspezifische Eigenschaften. Mit Hilfe von **testmusterspezifischen Eigenschaften** werden die Struktur und die konkrete Ausprägung eines Testmusters beschrieben. **Testobjektspezifische Eigenschaften** beeinflussen den Test, hängen aber direkt von der Struktur des jeweiligen Testobjekts ab. Diese Eigenschaften müssen jedes Mal bei der Anwendung eines Testmusterstereotypen auf ein Testobjekt erneut mit konkreten Werten belegt werden, unabhängig davon, ob es sich dabei um direkt anzuwendende oder wiederverwendbare Testmusterstereotypen handelt.

Die Festlegung aller Eigenschaften in den Regeln konnte nur indirekt erfolgen. Die Regeln fordern nur die konkrete Definition der Eigenschaften durch den Benutzer. Eine direkte Definition der Eigenschaften über Tagdefinitionen innerhalb der abstrakten Testmusterstereotypen ließ sich nicht realisieren, da dies die durch die Struktur des Profils gewonnene Flexibilität wieder zunichte gemacht hätte. Tagdefinitionen besitzen zur Identifikation einen Namen. Da jeder von den abstrakten Testmusterstereotypen abgeleitete konkrete Stereotyp die Tagdefinitionen seiner Vorgänger erbt, besäßen alle diese konkreten Stereotypen Tagdefinitionen mit denselben Namen.

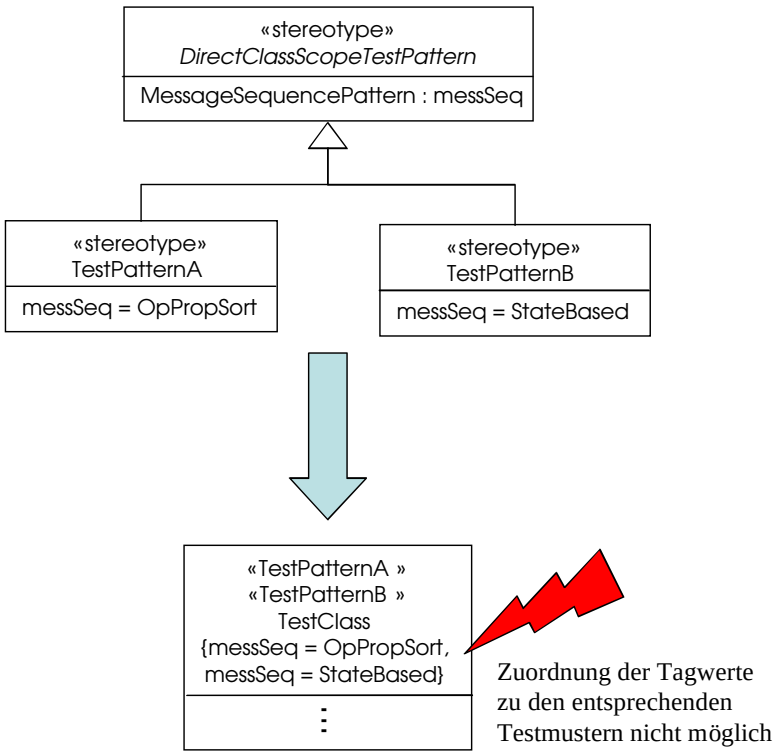


Abbildung 4.4 *Probleme mit Tagdefinitionen*

Die Folge einer solchen Vorgehensweise wäre, dass auf jedes Testobjekt nur jeweils ein Testmuster in Form eines Testmusterstereotypen angewendet werden könnte. Eine eindeutige Zuordnung der einzelnen Tagdefinitionen und ihrer Tagwerte zu den

[58] siehe statische Definitionen in Anhang A.1.2

entsprechenden Testmustern, wie in Abbildung 4.4 dargestellt, wäre nicht mehr möglich. Dies würde eine in der Praxis kaum annehmbare Einschränkung für einen systematischen Test bedeuten. Im UML-Metamodell ist auch festgelegt, dass die Namen der Tagdefinitionen innerhalb eines Gültigkeitsbereiches eindeutig sein müssen. Was der Benutzer auf Grund der indirekten Festlegung der in den konkreten Testmustern zu definierenden Eigenschaften zu beachten hat, wird in Kapitel 5 beschrieben.

Beim Test auf Klassenebene soll das korrekte Zusammenspiel von mehreren Methoden der zu testenden Klasse überprüft werden. Daraus ergibt sich, dass ein entsprechender Testfall immer aus einer Sequenz von Nachrichten besteht. Diese Nachrichtensequenz stellt den zentralen Gegenstand der Testfallermittlung, also der zweiten horizontalen Testaktivität, dar und bestimmt, welche Aspekte des Testobjekts untersucht werden. Um die unterschiedlichen Eigenschaften verschiedener Klassen angemessen testen zu können, sind verschiedene Arten zur Erzeugung von Nachrichtensequenzen notwendig, die diese spezifischen Eigenschaften berücksichtigen. Dementsprechend ist es erforderlich, dass für jedes konkrete klassenspezifische Testmuster sog. **Nachrichtensequenzmuster** festgelegt werden können, was durch eine Regel für klassenspezifische Testmusterstereotypen umgesetzt wird. Diese Regel fordert für die Konstruktion eines konkreten Testmusters die Spezifizierung einer Tagdefinition oder eines Stereotypen[59], mit deren Hilfe die für das Testmuster anzuwendenden Nachrichtensequenzmuster festgelegt werden können[60]. Um den unterschiedlichen Anforderungen hinsichtlich der Erzeugung von Nachrichtensequenzen Rechnung zu tragen, wurden die im Paket `Message Sequence Pattern` enthaltenen Stereotypen definiert[61].

Wie in Abschnitt 2.3.2 erläutert, besteht eine Testfallspezifikation neben der Festlegung einer Nachrichtensequenz auch aus abstrakten Wertebeschreibungen für testrelevante Elemente. Dabei handelt es sich häufig nur um die Angabe einer Menge von möglichen Werten, die für ein testrelevantes Element erlaubt sind. Da dies für die Erzeugung ausführbarer Testfälle zu ungenau ist, müssen aus der Menge der erlaubten Werte während der Testdatenerzeugung[62] konkrete Testdaten generiert werden. Diese Aufgabe wird im Rahmen des Testprofils durch sog. **Testdatenmuster** übernommen, welche durch im Paket `Test Data Pattern`[63] definierte Stereotypen repräsentiert werden. Für die Konstruktion von konkreten Testmustern existiert eine entsprechende Regel, die die Spezifizierung einer Tagdefinition oder eines Stereotypen fordert, mit deren Hilfe die für das Testmuster anzuwendenden Testdatenmuster festgelegt werden können[64].

Ein weiterer wichtiger Aspekt für die Erzeugung ausführbarer Testfälle ist die Art der Testauswertung. Hierbei geht es darum, wie Sollwerte mit aktuellen Istwerten eines Testfalls zu vergleichen sind. Auf den ersten Blick könnte man diese Problematik für trivial halten und meinen, dass die Istwerte identisch zu den Sollwerten sein müssten. In der Realität stellt sich dieses Problem jedoch als sehr vielschichtig dar, abhängig von verschiedenen Faktoren. Gesichtspunkte, die hier eine Rolle spielen, sind z. B. die Kritikalität und die Art des Testobjekts bzw. der zu vergleichenden Daten. Bei einem Testobjekt mit einer hohen Kritikalität müssen dessen Ausgaben genauer überprüft werden

[59] siehe Erläuterungen zum Unterschied zwischen direkt anwendbaren und wiederverwendbaren Testmustern auf Seite 51
[60] siehe jeweils Definition [1] in Anhang A.1.2.2 und A.1.2.3
[61] siehe Definitionen in Anhang A.2 und Erläuterungen in Abschnitt 4.5.1
[62] siehe Abschnitt 2.3.3
[63] siehe Anhang A.3
[64] siehe jeweils Definition [2] in A.1.2.2 und A.1.2.3

als bei eher unkritischen Objekten[65]. Oder für ein bestimmtes Testmuster, welches auf ein Testobjekt angewendet wird, soll ein anderes Auswertungsmuster verwendet werden als für ein anderes auf dieses Testobjekt angewendete Testmuster. Eine weitere Möglichkeit besteht darin, dass z. B. nur ein ganz bestimmter Teil der öffentlichen Schnittstelle einer Klasse nach einem Test ausgewertet werden soll, während der restliche Teil der Schnittstelle als nicht relevant angesehen wird. An diesen Beispielen lässt sich sehr gut erkennen, dass es sich bei der Testauswertung um einen sehr komplexen und zugleich sehr wichtigen Teil des Testprozesses handelt. Für die Definition der sog. **Auswertungsmuster** wurden im Testprofil die im Paket `Test Evaluation Pattern`[66] enthaltenen Stereotypen definiert. Die Struktur der Auswertungsmuster garantieren eine besonders flexible Konfiguration, die den genannten Anforderungen gerecht werden kann. Dabei gibt es die Unterscheidung zwischen globalen Auswertungsmustern und elementspezifischen Auswertungsmustern. **Globale Auswertungsmuster** beziehen sich auf kein konkretes Datenelement, wie z. B. Attribute oder Parameter, sondern betreffen davon unabhängige Aspekte, wie den Vergleich des Inhalts zweier Dateien oder die Überprüfung, ob die Ausführung eines Programms korrekt terminiert ist. Dadurch, dass keine direkte Beziehung zwischen globalen Auswertungsmustern und bestimmten Datenelementen besteht, werden diese direkt für ein Testmuster spezifiziert. **Elementspezifische Auswertungsmsuter** werden dagegen zur Überprüfung eines konkreten Datenelements verwendet. Aus diesem Grund muss eine Zuordnung zwischen solchen Auswertungsmustern und den entsprechenden Datenelementen erfolgen. Diese wird mit Hilfe des Stereotypen «`AbstractElementEvaluationPatternSet`» realisiert. Für die Konstruktion eines Testmusters existieren zwei Regeln, welche jeweils die Spezifizierung von Tagdefinitionen bzw. Stereotypen fordern, durch die globale bzw. elementspezifische Auswertungsmuster bestimmt werden können.

Die bisher vorgestellten Eigenschaften Nachrichtensequenzmuster, Testdatenmuster und Auswertungsmuster sind testmusterspezifische Eigenschaften. Zusätzlich wird eine testobjektspezifische Eigenschaft definiert. Dabei handelt es sich um die für den Test mit einem Testmuster einzubeziehenden Methoden einer Klasse. Nur die mit dieser Eigenschaft referenzierten Methoden einer zu testenden Klasse werden zur Erzeugung der Nachrichtensequenzen für die einzelnen Testfälle verwendet. Durch diese Eigenschaft wird eine relativ leichte Konfigurierbarkeit des Tests erreicht, indem z. B. noch nicht fertig implementierte Methoden vom Test ausgeschlossen werden können.

Der konkrete Testmusterstereotyp «`InheritedClassScopeTestPattern`» ermöglicht die genaue Konfiguration der Wiederverwendung von Testmustern und deren Testfällen, die bereits für den Test von Klassen oder Schnittstellen erzeugt und eventuell auch verwendet wurden, von denen die aktuell zu testende Klasse direkt oder indirekt durch Vererbung abgeleitet ist. Die Tagdefinition `reusedTestPattern` wird zur Referenzierung der Testmuster verwendet. Da ein Testmuster grundsätzlich auf mehrere Testobjekte angewendet werden kann, reicht eine Referenz darauf allerdings nicht aus, um auch die richtigen Testfälle für die Wiederverwendung genau spezifizieren zu können. Dazu wird zusätzlich eine Referenz auf das entsprechende Testobjekt benötigt. Im Testprofil wurde dies durch die Tagdefinitionen `baseElement` realisiert. Der Zusammenhang zwischen einem Testmuster, verschiedenen Testobjekten, auf die das Testmuster angewendet wird, und den daraus resultierenden Testfällen wird in Abbildung 4.5 dargestellt.

[65] Elemente einer Flugzeugsteuerung vs. Elemente aus einer Textverarbeitung
[66] siehe Definitionen in Anhang A.4.2 und Erläuterungen in Abschnitt 4.5.3

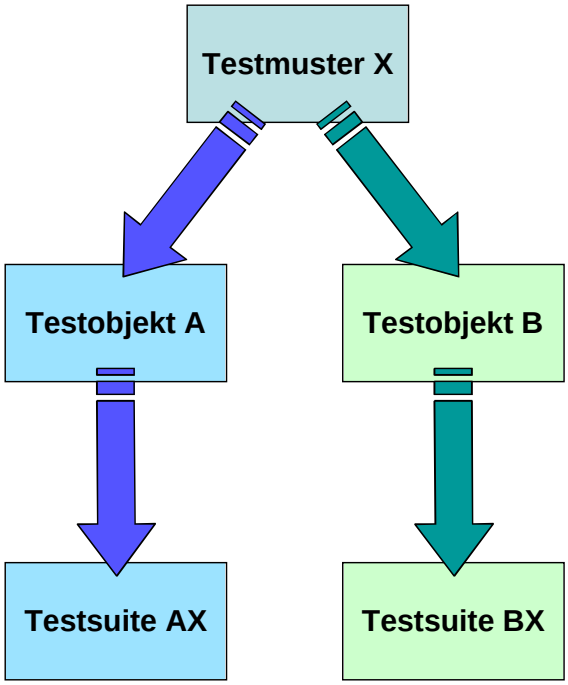


Abbildung 4.5 *Zusammenhang zwischen Testmuster, Testobjekten und Testfällen*

Für die Wiederverwendung von Testfällen müssen einige Voraussetzungen erfüllt werden. Grundsätzlich lassen sich klassenspezifische Testfälle einer Klasse `A` nur für Klassen wiederverwenden, die entweder direkt oder indirekt von der Klasse `A` erben. Die genauen Bedingungen, die für alle Klassen erfüllt werden müssen, wurden bereits in Abschnitt 4.2 genauer erläutert. Diese Forderungen werden durch entsprechende Einschränkungen, die im Stereotypen «`InheritedClassScopeTestPattern`» spezifiziert sind, realisiert. Für modale Klassen müssen zusätzliche Bedingungen erfüllt werden, die den Zustandsautomaten betreffen, der das Verhalten der zu testenden Klasse beschreibt. Diese Bedingungen werden durch das *XFREE State Model* beschrieben, welches in dieser Arbeit als Erweiterung des von Binder entwickelten *FREE State Model* entwickelt wurde. Die entsprechenden Einschränkungen werden in den im Paket `XFREE State Model` definierten Stereotypen spezifiziert und in Abschnitt 4.5.4 genauer erläutert.

4.5 Komponentenstereotypen

Neben den im letzten Abschnitt erläuterten Testmusterstereotypen werden im Testprofil Komponentenstereotypen definiert. Diese dienen der Konstruktion und Konfiguration von Testmustern in Form von konkreten Testmusterstereotypen im Rahmen der in den abstrakten Testmusterstereotypen formulierten Regeln. Das Testprofil kann an vielen Stellen um neue Komponentenstereotypen erweitert werden. So ist es z. B. möglich neue Nachrichtensequenzmuster, Testdatenmuster oder auch Muster zur Testauswertung zu definieren.

Die Komponentenstereotypen sind entsprechend der von ihnen abgedeckten Aspekte in verschiedenen Paketen innerhalb des Testprofils strukturiert. Nachfolgend werden die einzelnen Pakete mit einer kurzen Beschreibung ihres Zwecks aufgezählt.

(1) `Message Sequence Pattern` ➔ Muster zur Generierung von Nachrichtensequenzen

(2) `Test Data Pattern` ➔ Muster zur Erzeugung von konkreten Testdaten

(3) `Test Evaluation Pattern` ➔ Muster für die gezielte Konfiguration der Testauswertung

(4) `XFREE State Model` ➔ Spezifikation eines testbaren Zustandsautomaten

(5) `Method Properties` ➔ Definition von Eigenschaften von Methoden

(6) `Executable Expression` ➔ Definition von Eigenschaften von Ausdrücken, welche deren Ausführbarkeit zusichern

In den anschließenden Abschnitten werden die in diesen Paketen definierten Komponentenstereotypen näher beschrieben.

4.5.1 Message Sequence Pattern

Eine wesentliche Gemeinsamkeit klassenspezifischer Testmuster besteht darin, dass Testfälle immer aus einer Sequenz von Nachrichten bestehen, die das korrekte Zusammenwirken der durch die in der Sequenz enthaltenen Nachrichten aufgerufenen Methoden überprüfen soll. Nachrichtensequenzen stellen also einen invarianten Bestandteil klassenspezifischer Testmuster dar. Aus diesem Grund werden im Paket `Message Sequence Pattern` des Testprofils eine Menge von Komponentenstereotypen angeboten, die direkt zur Konstruktion von konkreten klassenspezifischen Testmustern verwendet werden.

Der grundlegende Stereotyp dieses Pakets ist der Stereotyp «`MessageSequence-Pattern`». Alle Nachrichtensequenzmuster müssen von diesem Stereotypen abgeleitet werden. Für jedes konkrete klassenspezifische Testmuster muss mindestens ein konkretes Nachrichtensequenzmuster angegeben werden, auf dessen Basis die Nachrichtensequenzen für das Testmuster erzeugt werden sollen[67]. Im Testprofil wird bereits eine Reihe von konkreten Nachrichtensequenzmustern zur Verfügung gestellt. Der Stereotyp «`UserDefinedSequence`» nimmt hierbei eine Sonderstellung ein. Bei der Verwendung dieses Stereotyps werden die Nachrichtensequenzen nicht wie bei allen vom Stereotypen «`SequenceGenerationPattern`» abgeleiteten Stereotypen auf der Basis von im UML-Modell enthaltenen Informationen automatisch erzeugt. Der Benutzer kann hier selbst ganz bestimmte Nachrichtensequenzen inklusive der aktuellen Parameter spezifizieren. Hierdurch kann der Benutzer ganz gezielt Nachrichtensequenzen testen, die durch die automatische Generierung nicht erzeugt wurden. Durch die weitere Unterteilung des Stereotypen «`SequenceGenerationPattern`» in die beiden abstrakten Stereotypen «`NonModalClassSequences`» und «`ModalClassSequences`» wird den unterschiedlichen Anforderungen hinsichtlich der Modalität der jeweiligen Klasse Rechnung getragen. Eine ausführliche Diskussion dieser Problematik ist weiter unten in diesem Abschnitt zu finden.

Durch den Stereotypen «`MethodPropertySorting`» wird ein verallgemeinertes Modell des in [Binder99] vorgestellten *Alpha-Omega Zyklus'* realisiert. Binder erzeugt mit diesem Testmuster auf der Basis der Eigenschaften *Komplexität* und *Sichtbarkeit* von Methoden eine Nachrichtensequenz, bei der die Nachrichten entsprechend der genannten Eigenschaften sortiert werden. Konstruktoren und Destruktoren bilden hierbei

[67] siehe Abschnitt 4.4

immer einen natürlichen Rahmen der Sequenz. Beim Stereotypen «`MethodPropertySorting`» wird die Abstraktion dadurch erreicht, dass verschiedene Eigenschaften von Methoden in beliebig vielen Stufen für die Sortierung der Nachrichten verwendet werden können. Realisiert wird dies durch die Tagdefinition `orderProperties`. Durch den damit verknüpften Stereotypen «`OrderProperty`» bzw. den von ihm abgeleiteten konkreten Stereotypen werden verschiedene Kriterien zur Verfügung gestellt, nach denen die Sortierung erfolgen kann. Durch den Stereotypen «`MethodGroups`» wird ein Teil des Konzepts des *Alpha-Omega Zyklus'* umgesetzt. Allerdings hat der Benutzer auch hier die Möglichkeit, die Reihenfolgen der einzelnen Methodengruppen[68] selbst zu spezifizieren. Der andere Teil des *Alpha-Omega Zyklus'* wird durch den Stereotypen «`MethodVisibility`» realisiert, der die Sortierung entsprechend der drei Möglichkeiten der Sichtbarkeit von Methoden, `public`, `proteced` und `private`, festlegt. Hier muss darauf geachtet werden, welche Sichtbarkeitskonzepte in der jeweiligen Programmiersprache realisiert sind. Gegebenenfalls muss hier das Testprofil durch die Definition entsprechender Stereotypen erweitert werden. Mit Hilfe des Stereotypen «`ImplementationBased`» kann eine Nachrichtensequenz in der Reihenfolge der gegenseitigen Benutzung der entsprechenden Methoden erzeugt werden. Diese Benutzungsrelationen können durch eine Analyse des Quellcodes ermittelt werden. Der Stereotyp «`Metric`» bzw. von ihm abgeleitete konkrete Stereotypen erlauben eine Sortierung entsprechend den Auswertungsergebnissen der jeweiligen Metrik. Für die Anwendung müssen allerdings erst noch konkrete Stereotypen abgeleitet werden, die eine spezielle Metrik realisieren.

Der mehrstufige Sortierungsprozess zur Erzeugung einer Nachrichtensequenz kann wie folgt beschrieben werden. Wurden verschiedene Eigenschaften mit der Tagdefinition `orderProperties` in der Reihenfolge P_1, P_2,... P_n als Sortierkriterien festgelegt, so werden zuerst alle in den Test einzubeziehenden Methoden nach Eigenschaft P_1 sortiert. Sind mehrere Methoden mit dieser Eigenschaft nicht unterscheidbar, bilden sie eine Untermenge. Die Methoden in den einzelnen Untermengen werden dann jeweils separat entsprechend Eigenschaft P_2 sortiert. Dies wird soweit fortgesetzt, bis nach allen Sortierkriterien sortiert wurde oder bis alle Untermengen jeweils nur noch eine Methode enthalten.

Ein wesentliches Merkmal des Verhaltens von Klassen bzw. deren Objekten sind die Einschränkungen, die für diese Klasse hinsichtlich der erlaubten Nachrichtensequenzen gelten. Diese Einschränkungen sind für die Erzeugung von Nachrichtensequenzen von entscheidender Bedeutung. Binder gibt in [Binder99] eine Klassifikation an, nach der Klassen entsprechend der für sie geltenden Einschränkungen in unterschiedliche Gruppen eingeteilt werden können. Dabei werden vier Arten von Klassen unterschieden: *nicht-modale, unimodale, quasi-modale* und *modale* Klassen. Binder ordnet Klassen nach den folgenden Gesichtspunkten diesen vier Gruppen zu:

(1) **Nicht-modale Klassen** besitzen keine Einschränkungen in Bezug auf erlaubte Nachrichtensequenzen. Dies trifft häufig auf Klassen zu, die grundlegende Datentypen implementieren.

(2) Die erlaubten Nachrichten für ein Objekt einer **unimodalen Klasse** hängen von vergangenen, an das Objekt gesendeten Nachrichten ab, nicht aber vom Zustand des Objekts. Als Beispiel nennt Binder eine Klasse, die eine

[68] Die zur Verfügung stehenden Methodengruppen werden durch den Stereotypen «`MethodType`» bestimmt.

Ampelsteuerung realisiert. Diese Klasse erlaubt z. B. eine Nachricht `RedLightOn` nur, wenn zuvor eine Nachricht `YellowLightOn` akzeptiert wurde.

(3) **Quasi-modale Klassen** besitzen Einschränkungen für erlaubte Nachrichten, die vom Zustand eines Objekts der Klasse abhängen. Container-Klassen, wie z. B. ein `Stack`, sind nach dieser Einteilung quasi-modal.

(4) **Modale Klassen** besitzen Einschränkungen, die sowohl vom Zustand als auch von vergangenen Nachrichten abhängen.

Die Unterteilung von Klassen mit Einschränkungen für erlaubte Nachrichtensequenzen in die Gruppen *unimodal*, *quasi-modal* und *modal* hat für den Test selbst allerdings keine Bedeutung. Für den Test ist lediglich wichtig, dass das Verhalten für alle diese Klassen durch ein testbares Modell beschrieben wird. In [Binder99] wird ein Modell vorgestellt, welches den Kriterien für ein testbares Modell genügt, das sog. *FREE State Model*. Dieses Modell dient als Basis für den Test und, insbesondere für die Erzeugung von Nachrichtensequenzen, für alle drei genannten Gruppen von Klassen. Die Unterschiede, die sich für diese Gruppen von Klassen bei der Modellierung ihres Verhaltens mit Hilfe des *FREE State Model* ergeben, sind für den Test selbst insofern nicht relevant, als dass dabei zwar sehr wohl unterschiedliche Testfälle erzeugt werden, die genau das Verhalten von Objekten der getesteten Klasse untersuchen, zur Erzeugung dieser Testfälle aber die gleichen Techniken angewendet werden. Dies wird auch bei genauer Betrachtung der von Binder für diese Arten von Klassen angegebenen Testmuster, den *Quasi-modal Class Test* und den *Modal Class Test*, deutlich. Die für beide Testmuster vorgeschlagenen Teststrategien sind nahezu identisch. Die einzigen, allerdings sehr geringen Unterschiede, ergeben sich genau für die Erstellung eines testbaren Zustandsautomaten als Basis des Tests. In jedem Fall soll der Zustandsautomat jedoch den Anforderungen des *FREE State Model* genügen. Die darauf angewendeten Techniken weisen keine Unterschiede auf. Außerdem soll die Teststrategie des *Modal Class Test* gleichermaßen auf *modale* und *unimodale* Klassen angewendet werden, was noch einmal unterstreicht, dass für den Test auch hier kein wesentlicher Unterschied besteht.

Die Definition von *unimodalen* Klassen auf der Basis vergangener Nachrichten besitzt lediglich eine theoretische Bedeutung, die bei einer konsequenten Umsetzung sogar dazu führen kann, dass ein allein auf einem solchen Modell basierender Test eine fehlerhafte Implementierung als korrekt akzeptieren würde. Wenn das Kriterium für einen bestandenen Test nur darin besteht, dass Nachrichten in der richtigen Reihenfolge akzeptiert werden, können fehlerhafte Implementierungen der entsprechenden Methoden wohl kaum aufgedeckt werden. Würde in dem von Binder genannten Beispiel der Ampelsteuerung die durch die Nachricht `YellowLightOn` aktivierte Methode anstatt der gelben die grüne Lampe einschalten, könnte dieser Fehler allein auf der Basis der Reihenfolge eingegangener Nachrichten nicht aufgedeckt werden. Außerdem sind Zustandsautomaten, bei denen der aktuelle Zustand in Abhängigkeit von in der Vergangenheit ausgeführten Methoden ermittelt werden soll, nicht testbar, da dies i. Allg. in der Realität nicht möglich ist. Soll ein solcher Zustandsautomat testbar gemacht werden, müsste die entsprechende Strategie mit Hilfe von Attributen realisiert werden. Damit würde die Modellierung wieder auf dem Zustand eines Objekts basieren und nur noch indirekt auf der Aufrufreihenfolge von Methoden.

Die Schlussfolgerung aus diesen Überlegungen ist die Zusammenfassung der drei Gruppen *unimodal, quasi-modal* und *modal*, so dass in dieser Arbeit nur noch die beiden Gruppen *nicht-modal* und *modal* unterschieden werden.

Für den Test nicht-modaler Klassen wird in [Binder99] das Testmuster *Nonmodal Class Test* zur Verfügung gestellt. Darin gibt Binder verschiedene Strategien zur Erzeugung von entsprechenden Nachrichtensequenzen an. Diese werden im Testprofil durch die beiden Stereotypen «`SystematicDefineUse`» und «`RandomDefineUse`» realisiert. Eine *Systematic-Define-Use*-Sequenz besteht dabei aus einer zustandsverändernden Methode, gefolgt von allen zustandserhaltenden Methoden. Bei einer *Random-Define-Use*-Sequenz wird eine Menge zustandsverändernder Methoden in eine zufällige Reihenfolge gebracht. Daran schließt sich eine Menge zufällig sortierter zustandserhaltender Methoden an.

Für die Erzeugung von Nachrichtensequenzen für *modale* Klassen auf der Basis eines Zustandsautomaten wird der Stereotyp «`ModalClassSequences`» zur Verfügung gestellt. Da es verschiedene Techniken zur Erzeugung einer zustandsbasierten Nachrichtensequenz gibt, ist dieser Stereotyp abstrakt. Die einzelnen Techniken können durch die Ableitung von konkreten Stereotypen realisiert werden, wie dies durch den Stereotypen «`NPlusStrategy`» beispielhaft gezeigt wird. Das entsprechende Testmuster wird in [Binder99] näher erläutert.

4.5.2 Test Data Pattern

Die Erzeugung konkreter Testdaten während der Testdatenerzeugung[69] ist eine wesentliche und auch invariante Aktivität des Testprozesses. Für die Umsetzung dieser Aktivität innerhalb des Testprofils werden die im Paket `Test Data Pattern` definierten Stereotypen verwendet.

Bei klassenspezifischen Testmustern wird für ein Testobjekt, also eine Klasse, zuerst mit Hilfe der Nachrichtensequenzmuster eine Menge von Nachrichtensequenzen generiert. Daran anschließend werden für jede Nachrichtensequenz durch Anwendung eines Testdatenmusters eine Menge von Testdatensätzen erzeugt. Dies wird in Abbildung 4.6 verdeutlicht.

Von dem abstrakten Stereotypen «`TestDataPattern`» werden alle konkreten Muster zur Testdatenerzeugung abgeleitet. Mit der Tagdefinition `usePolymorphism` kann festgelegt werden, ob für polymorphe Elemente auch Objekte von Unterklassen dieser Elemente erzeugt werden sollen. Besitzt z. B. eine Methode einen Parameter der Klasse `A`, für die zwei Unterklassen `B` und `C` spezifiziert sind, so kann mit dem Tagwert gesteuert werden, ob für diesen Parameter entweder nur Objekte der Klasse `A` oder auch Objekte der Klassen `B` und `C` erzeugt werden sollen. Durch die zweite Tagdefinition `extendSystematically` kann spezifiziert werden, ob die Erzeugung von polymorphen Objekten systematisch oder zufällig erfolgen soll. Wie die systematische Erzeugung von Objekten der Unterklassen genau erfolgen soll, muss endgültig durch ein Testwerkzeug spezifiziert werden, welches das Testprofil praktisch umsetzt. Dieser Aspekt ist nicht Gegenstand dieser Arbeit. Entsprechende Ansätze werden in [Overbeck94] [Winter00] und [Winter01] vorgestellt.

[69] siehe Abschnitt 2.3.3

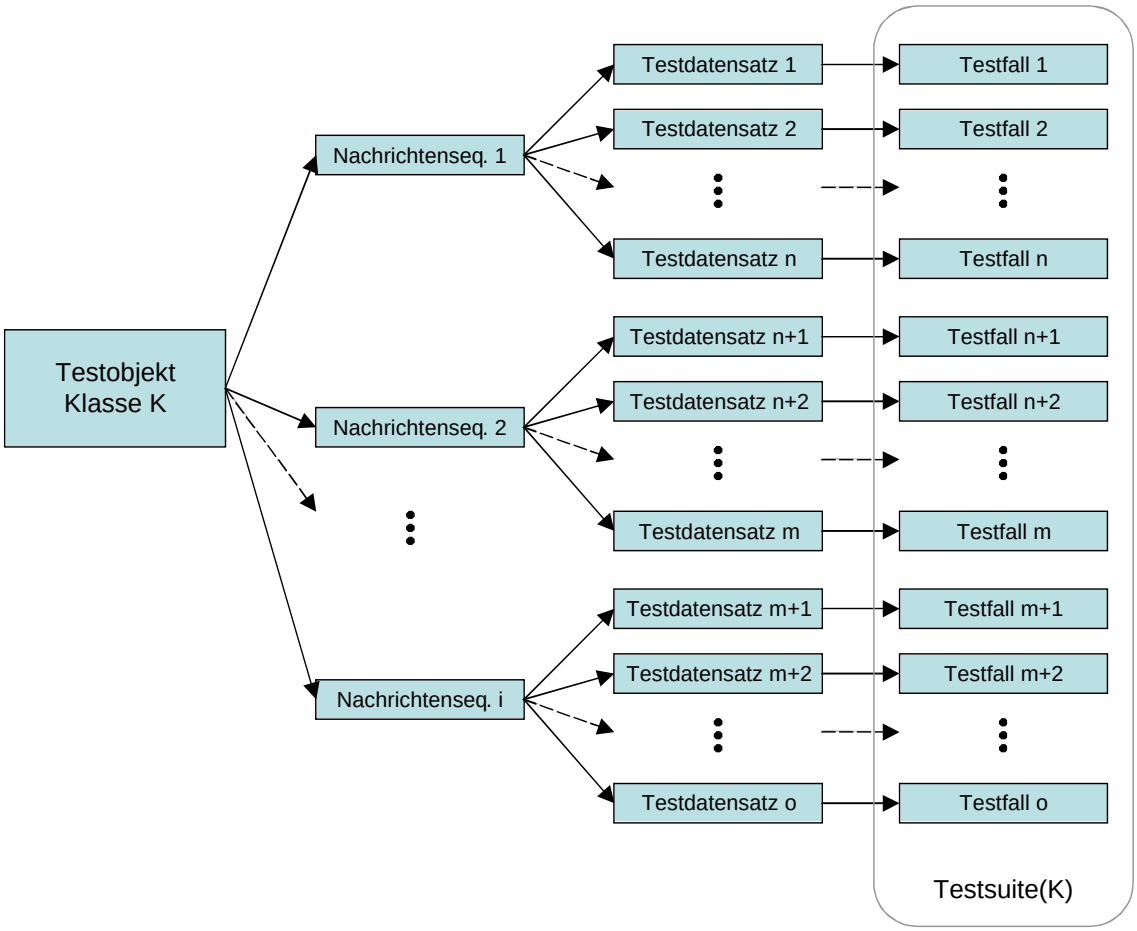


Abbildung 4.6 *Testdatenerzeugung für klassenspezifische Testmuster*

In dem Paket werden die beiden konkreten Stereotypen «`RandomTestData`» und «`InvariantBoundaries`» zur Verfügung gestellt. Durch das erste Testdatenerzeugungsmuster werden zufällige Testdaten erzeugt, wobei der Umfang des Tests durch die Anzahl der zu erzeugenden Testfälle festgelegt werden kann. Im Zusammenhang mit klassenspezifischen Testmustern ergibt sich hier wieder eine Interpretationsfreiheit, die durch ein Testwerkzeug beseitigt werden muss. Es könnte z. B. für jede erzeugte Nachrichtensequenz die festgelegte Anzahl von Testfällen generiert werden oder für alle Nachrichtensequenzen zusammen. Der Stereotyp «`InvariantBoundaries`» realisiert das in [Binder99] vorgestellte gleichnamige Muster, welches Binder als Basis für die Testdatenerzeugung in vielen anderen Testmustern verwendet. Mit der Tagdefinition `systematicDepth` kann festgelegt werden, bis zu welcher Tiefe bei komplexen Datentypen die Testdatenerzeugung systematisch nach dem Muster *Invariant Boundaries* erfolgen soll. Ab dieser Verschachtelungstiefe werden die Testdaten nur noch zufällig erzeugt. Auf diese Weise kann ebenfalls die Anzahl der erzeugten Testfälle gesteuert werden.

4.5.3 Test Evaluation Pattern

Ein weiterer wesentlicher Bestandteil des Tests ist die Testauswertung, also der Vergleich zwischen den erwarteten Sollwerten und den tatsächlich gelieferten Istwerten[70]. Für die Bereitstellung der für diese Testaktivität benötigten Informationen innerhalb

[70] siehe Abschnitt 2.3.5

eines UML-Modells wird im Rahmen des Testprofils das Paket `Test Evaluation Pattern` zur Verfügung gestellt.

Die Anforderungen, die unterschiedliche Testobjekte erfüllen müssen, können sehr verschieden sein und sogar für ein einzelnes Testobjekt zwischen verschiedenen Tests differieren. So muss die Wirkungsweise kritischer Testobjekte sehr viel genauer untersucht werden, indem z. B. die Ergebnisse von Fließkommaberechnungen einer höheren Genauigkeit bzw. einer niedrigeren Toleranz gegenüber den Sollwerten genügen müssen. Und auch die einzelnen Elemente der oft sehr komplexen Strukturen eines Testobjekts besitzen meistens sehr unterschiedliche Eigenschaften, für die jeweils andere Anforderungen an die Testauswertung bestehen. Um diesen vielfältigen Anforderungen für unterschiedliche Testobjekte und Testmuster gerecht zu werden, muss die Testauswertung sehr zielgenau und flexibel konfigurierbar sein. Dies kann durch die Struktur des Pakets `Test Evaluation Pattern` gewährleistet werden.

Um dies erreichen zu können, werden die beiden grundlegenden Stereotypen «`TestEvaluationPattern`» und «`AbstractElementEvaluationPatternSet`» definiert. Mit dem ersten bzw. den davon abgeleiteten Stereotypen werden verschiedene Muster für die Testauswertung zur Verfügung gestellt, also den individuellen Anforderungen einzelner Auswertungsobjekte angepasste Auswertungstechniken. Durch die Verwendung des Stereotypen «`AbstractElementEvaluationPatternSet`» können diese Techniken sehr flexibel und differenziert auf einzelne Objekte und Testmuster angewendet werden. In Abbildung 4.7 werden diese Möglichkeiten schematisch dargestellt. Die Klasse `Abgeleitet` ist das aktuelle Testobjekt. Darauf werden die beiden Testmuster `Testmuster1` und `Testmuster2` angewendet. Die beiden Auswertungsmuster `AM5` und `AM6` dienen der Überprüfung von globalen, nicht elementspezifischen Aspekten nach der Anwendung von `Testmuster1`. Deshalb besitzen sie keine Beziehung zu einem konkreten Datenelement. Solche globalen Aspekte können z. B. die korrekte Termination des Programms oder die Überprüfung des Inhalts einer Datei sein. Das Auswertungsmuster `AM2` überprüft für die Testfälle von `Testmuster2` die durch den Ausgangsparameter `out2` der Methode `C` zurückgegebenen Werte. Auswertungsmuster `AM1` wird sowohl für die Testfälle von `Testmuster1` als auch für die Testfälle von `Testmuster2` verwendet. Allerdings wird für `Testmuster1` nur Attribut `A` ausgewertet und für `Testmuster2` nur der Rückgabeparameter von Methode `C`.

Erreicht wird dieser hohe Grad an Flexibilität durch den Stereotypen «`AbstractElementEvaluationPatternSet`». Durch diesen Stereotypen erfolgt die Zuordnung von elementspezifischen Auswertungsmustern zu einzelnen Datenelementen.

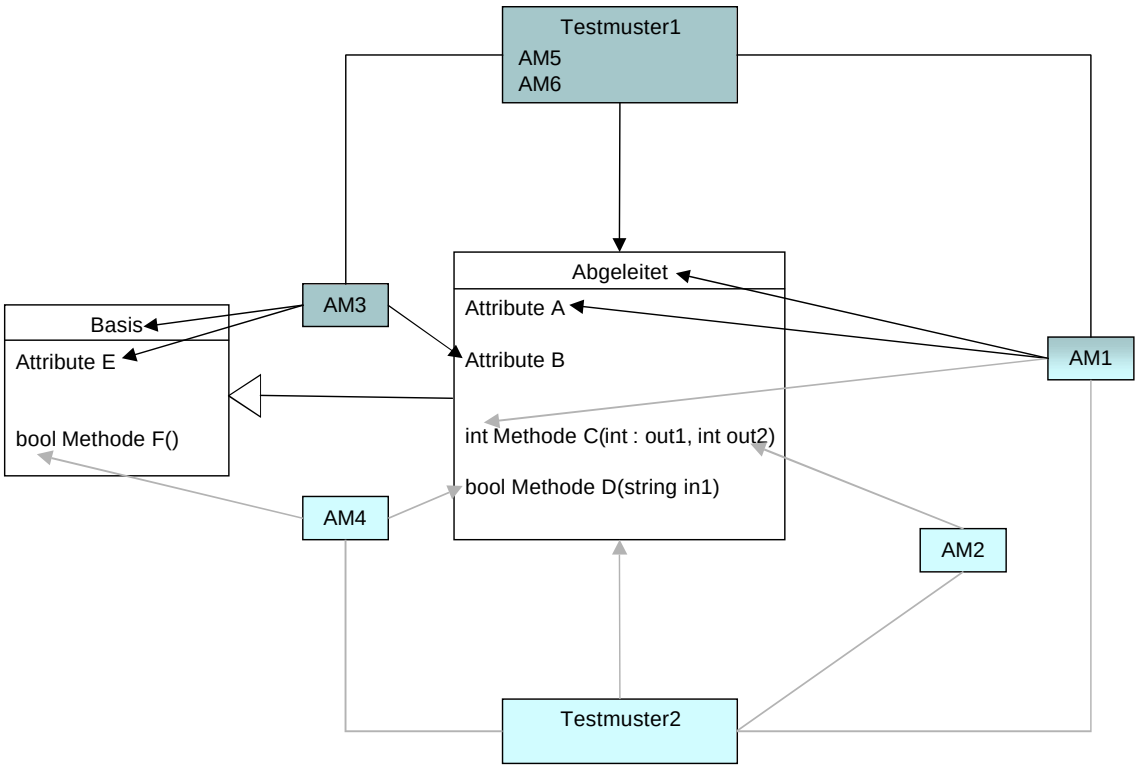


Abbildung 4.7 *Möglichkeiten der Konfiguration von Auswertungsmustern*

Die vom Stereotypen «`TestEvaluationPattern`» abgeleiteten Auswertungsmuster werden nach zwei verschiedenen Kriterien klassifiziert. Auf der einen Seite erfolgt eine Einteilung nach dem Kontext der Anwendung des Auswertungsmusters. Dabei gibt es globale Auswertungsmuster, die keinen direkten Bezug zu einem konkreten Datenelement besitzen. Diese werden durch den Stereotypen «`GlobalEvaluationPattern`» realisiert. Außerdem gibt es Auswertungsmuster, mit deren Hilfe einzelne Datenelemente, wie z. B. Parameter, überprüft werden können. Für diese Art von Auswertungsmuster wurde der Stereotyp «`AbstractElementEvaluationPattern`» definiert. Die zweite Kategorie teilt die Auswertungsmuster danach ein, ob für deren Anwendung durch ein externes Orakel erzeugte Sollwerte notwendig sind. Dafür wurden die beiden Stereotypen «`ExternalOracleEvaluationPattern`» und «`InternalEvaluationPattern`» definiert.

Als globale Auswertungsmuster werden die beiden Stereotypen «`NormalTermination`» und «`FileContentsEquality`» zur Verfügung gestellt. Der erste Stereotyp dient der Überprüfung, ob ein Programm normal terminiert, also nicht abgestürzt ist. Für diesen müssen keine Sollwerte erzeugt werden, da das Auswertungsmuster selbst den Sollwert darstellt. Aus diesem Grund erbt dieser Stereotyp zusätzlich vom Stereotypen «`InternalEvaluationPattern`». Beim zweiten Stereotypen werden zwei Dateien auf identischen Inhalt überprüft. Hierfür muss ein externes Orakel eine entsprechende Datei als Sollwert erzeugen. Deshalb wird dieser Stereotyp zusätzlich vom Stereotypen «`ExternalOracleEvaluationPattern`» abgeleitet.

Als elementspezifische Auswertungsmuster werden die Stereotypen «`StateEquality`», «`InlineEquality`», «`DeepEquality`» und «`UserDefinedOracleEvaluation`» zur Verfügung gestellt. Mit dem Auswertungsmuster «`StateEquality`» kann überprüft werden, ob sich ein Datenelement in einem bestimmten zusammengesetzten Zustand befindet. Der Vergleich kann nur erfolgen, wenn der

entsprechende Zustandsautomat dem *XFREE State Model* genügt, also mit dem entsprechenden Stereotypen, in diesem Fall «AbstractXFREEStateMachine», markiert ist. Auf der Basis dieser Zustandsautomaten, einer Nachrichtensequenz und dazugehörigen Testdaten kann sogar der Sollwert, also der entsprechende zusammengesetzte Zustand, automatisch ermittelt werden. Deshalb wird zur Sollwertermittlung kein externes Orakel benötigt, so dass der Stereotyp «StateEquality» ebenfalls vom Stereotypen «InlineEvaluationPattern» abgeleitet wird. Für die anderen drei elementspezifischen Auswertungsmuster müssen Sollwerte durch ein externes Orakel erzeugt werden. Deshalb werden diese vom Stereotypen «ExternalOracleEvaluationPattern» abgeleitet. Das durch den Stereotypen «InlineEquality» realisierte Auswertungsmuster vergleicht zwei Objekte des gleichen Typs mit Hilfe des Inline-Operators „=“, der dazu auf dem Typen der zu vergleichenden Ob jekte definiert sein muss. Das Auswertungsmuster «DeepEquality» vergleicht zwei Objekte des gleichen Typs derart mit einander, dass alle Instanzvariablen primitiver Typen direkt verglichen werden und auf komplexe Instanzvariablen rekursiv wieder das Auswertungsmuster angewendet wird. Mit dem Auswertungsmuster «UserDefinedOracle-Evaluation» kann der Benutzer mehrere Methoden spezifizieren, die zwei Objekte des entsprechenden Typs genau so vergleichen, wie es für den jeweiligen Test angemessen ist. Die Signatur solcher Vergleichsmethoden muss natürlich bestimmten Bedingungen genügen, die durch entsprechende Einschränkungen definiert wurden. So muss der Rückgabeparameter einer solchen Methode stets vom Typ `boolean` sein und es müssen zwei Eingangsparameter vom Typ der zu vergleichenden Objekte definiert sein. Nur so ist die automatische Ausführbarkeit gewährleistet.

Da die Sollwertermittlung durch ein externes Orakel sehr eng mit der Testauswertung verknüpft ist, wurde der Stereotyp «AbstractExternalOracle» ebenfalls in diesem Paket definiert. Er stellt das Grundgerüst zur Realisierung des externen Orakels zur Verfügung. Die automatische Erzeugung von Testfällen für den Klassentest basiert auf der Erzeugung verschiedener Reihenfolgen von an das Testobjekt gesendeten Nachrichten. Um für diese verschiedenen Nachrichtensequenzen, korrekte Sollwerte generieren zu können, muss ein externes Orakel in der Lage sein, genau die gleichen Nachrichten verarbeiten zu können wie das Testobjekt selbst. Dies soll an einem Beispiel verdeutlicht werden. Man stelle sich die folgende zu testende Klasse vor, die dazu dient, ein Datum zu speichern.

```
class Date {
     void setDay(int day);
     void setMonth(int month);
     void setYear(int year);
     int getDay();
     int getMonth();
     int getYear();
}
```

Für diese Klasse soll es möglich sein Tag, Monat und Jahr getrennt voneinander zu manipulieren. Als Orakel steht eine Klasse zur Verfügung, die ebenfalls zur Verwaltung eines Datums dient, bei der allerdings Tag, Monat und Jahr nur gemeinsam geändert werden können.

```
class Date {
     void setDate(int day, month, year);
```

```
        int getDay();
        int getMonth();
        int getYear();
    }
```

Für dieses Orakel ist es nicht möglich, automatisch Nachrichtensequenzen zu generieren, die jeder beliebigen Nachrichtensequenz des Testobjekts entsprechen, ohne den Zusammenhang zwischen der Orakelmethode `setDate` und den drei Methoden des Testobjekts `setDay`, `setMonth` und `setYear` zu modellieren. Ein einheitliches Konzept zur Modellierung solcher Zusammenhänge übersteigt den Rahmen dieser Arbeit. Aus diesem Grund wird hier ein pragmatischer Ansatz gewählt, bei dem der Benutzer für die Realisierung der Beziehungen zwischen dem Testobjekt und dem entsprechenden Orakel verantwortlich ist.

Um ein Orakel für eine Klasse zu erzeugen, wird von dieser Klasse durch Vererbung eine Klasse abgeleitet, die mit dem Stereotypen «`AbstractExternalOracle`» markiert ist. In dieser Orakelklasse müssen alle öffentlichen Methoden, die in der Oberklasse implementiert sind, überschrieben werden. Dabei soll das Verhalten der entsprechenden überschriebenen Methode so gut wie möglich simuliert werden. Eine vollständige Nachbildung des Verhaltens ist u.U. nicht möglich oder nicht erwünscht. So kann es sein, dass ein innerhalb einer solchen Methode angesteuertes externes Orakel[71] nur einen Teil der Funktionalität der zu testenden Methode abdeckt. In einem anderen Fall soll aus Kostengründen nur ein vereinfachtes Orakel realisiert werden. Die Unterschiede zwischen der zu testenden Methode und der dazugehörigen Orakelmethode müssen durch entsprechende Beziehungen zwischen den jeweiligen Vor- und Nachbedingungen der beiden Methoden modelliert werden. Wird durch die Orakelmethode nur ein Teil der Funktionalität der zu testenden Methode realisiert, müssen die Vor- und die Nachbedingung der Orakelmethode stärker sein als bei der zu testenden Klasse. In einem solchen Fall können automatisch ausführbare Testfälle nur im Rahmen der Vor- und Nachbedingungen des Orakels generiert werden. Für die restlichen Testfälle fehlen die Sollwerte und somit ist eine automatische Testauswertung nicht möglich. Sollen geerbte Methoden mit in den Test einbezogen werden, so müssen diese in der Orakelklasse ebenfalls überschrieben werden. Existiert bereits ein entsprechendes Orakel, in dem diese Methode bereits implementiert ist, so kann diese Implementierung wiederverwendet werden. Insgesamt ist bei der Realisierung des Orakelobjekts zu beachten, dass nur die öffentliche Schnittstelle der zu testenden Klasse realisiert werden soll.

Durch diese Art der Umsetzung einer Orakelklasse kann das Orakel einer ganzen Klasse aus einzelnen, externen Orakeln zusammengesetzt werden, was die Anwendung erheblich vereinfacht und sehr flexibel gestaltet.

4.5.4 XFREE State Model

Wie bereits in Abschnitt 3.3.4 beschrieben, wird ein Zustandsdiagramm dazu verwendet, das Verhalten von Instanzen von Klassen zu spezifizieren, die in Abhängigkeit von ihrem aktuellen Zustand ein unterschiedliches Verhalten zeigen sollen. Dort wurde auch bereits angedeutet, dass die in der UML-Spezifikation definierten Regeln die Erzeugung von Zustandsautomaten zulässt, die nicht testbar sind. In [Binder99] wird das sog. ***F**lattened **RE**gular **E**xpression (FREE) State Model* vorgestellt, welches zum Ziel hat, ein testbares Modell des Verhaltens von Instanzen einer Klasse zur Verfügung zu stellen.

[71] z.B. ein älteres System, oder eine Excel-Tabelle

Auch wenn durch dieses Modell viele Probleme gelöst werden, die sich durch die UML-Spezifikation für den Test ergeben, bleiben doch einige Fragen offen. Aus diesem Grund wird innerhalb des Testprofils eine erweiterte Version des *FREE State Model* realisiert, das *EXtended **FREE** State Model*. Die Umsetzung erfolgt durch die im Paket `XFREE State Model` definierten Stereotypen[72].

In diesem Abschnitt werden die Schwächen der UML-Spezifikation im Hinblick auf die Testbarkeit des Modells aufgezeigt, die mit dem *FREE State Model* vorgeschlagenen Lösungen diskutiert und die endgültige Umsetzung des *XFREE State Model* im Testprofil dargestellt. Die dabei spezifizierten Stereotypen präzisieren die Semantik der innerhalb der UML für die Definition eines Zustandsautomaten verwendeten Modellelemente dahingehend, dass der Anwender in die Lage versetzt wird, ein testbares Modell des Verhaltens von Instanzen einer Klasse zu erstellen. Die grundlegenden Stereotypen zur Konstruktion eines testbaren Zustandsautomaten sind die Stereotypen «`XFREEStateMachine`», «`XFREEState`» und «`XFREETransition`». Dabei muss ein Zustandsautomat vollständig aus diesen bzw. den davon abgeleiteten Stereotypen konstruiert werden. Dadurch wird sichergestellt, dass er den Bedingungen des *XFREE State Model* genügt und somit testbar ist.

Die entscheidende Voraussetzung für die Testbarkeit eines Zustandsautomaten besteht darin, dass für eine Instanz einer Klasse jederzeit eindeutig festgestellt werden kann, in welchem Zustand des spezifizierten Automaten sie sich gerade befindet [Binder99]. Dafür ist eine eindeutige Definition des Begriffs *Zustand* notwendig. In [OMG01] wird ein Zustand nur relativ allgemein als eine Situation definiert, während der einige Invarianten gelten. Dabei kann es sich sowohl um statische Situationen handeln, wobei ein Objekt auf externe Ereignisse wartet, als auch um dynamische Bedingungen wie der Ausführung einer Aktivität. Durch diese ungenaue Definition können sich, insbesondere für den Test und dessen Automatisierung, große Probleme ergeben. So werden z. B. bezüglich zustandsbeschreibender Invarianten keinerlei Einschränkungen bzw. Forderungen definiert, wodurch es zu mehrdeutigen Situationen kommen kann, die einen Test unmöglich machen[73].

Als eine Grundlage für eine testbare Definition des Begriffs *Zustand* sei hier noch einmal auf die in [Binder99] angegebene Klassifikation von Zuständen verwiesen, die in Abschnitt 3.3.4 vorgestellt wurde. Binder kommt dabei zu dem Schluss, dass Zustände, die zusammengesetzt und abstrakt sind, am besten für den Test geeignet sind. Begründet wird dies damit, dass bei der Betrachtung primitiver Zustände der Zustandsautomat so komplex wird, dass ein effektiver Test nicht mehr möglich ist. Außerdem bietet ein Zustandsautomat mit primitiven Zuständen gegenüber einem Automaten mit zusammengesetzten Zuständen auch keine zusätzlichen Informationen, sondern nur eine wesentlich erhöhte Redundanz. Deshalb lehnt Binder einen Test auf der Basis primitiver Zustände ab. Die Verwendung konkreter Zustände als Basis für den Test sieht Binder ebenfalls kritisch, da hierfür das Prinzip der Datenkapselung verletzt werden müsste und der Zustandsraum ebenfalls sehr schnell zu komplex für einen effektiven Test werden kann. Allerdings sollte auch, insbesondere für sehr kritische Komponenten, ein Test auf der Ebene der Implementierung möglich sein, was aber nicht Gegenstand dieser Arbeit ist, die sich auf den Test der öffentlichen Schnittstelle beschränkt. Die Struktur des Testprofils lässt allerdings grundsätzlich eine Erweiterung für den Test auf der Ebene der Implementierung zu.

[72] siehe Anhang A.5
[73] siehe Abbildung 4.9 auf Seite 68

Hier sei noch einmal auf den Unterschied zwischen den Begriffen *primitiv* und *konkret* bzw. *zusammengesetzt* und *abstrakt* hingewiesen. Diese betreffen jeweils die Einteilung von Zuständen nach den beiden unterschiedlichen Konzepten der *Granularität* und *Sichtbarkeit*. Es kann also primitive Zustände geben, die konkret, aber auch solche, die abstrakt sind. Der genaue Zusammenhang zwischen *Granularität* und *Sichtbarkeit* wird noch einmal in Tabelle 4.1 dargestellt.

Granularität / Sichtbarkeit	konkret	abstrakt
primitiv	**eine einzige Wertekombination** von Zustandsvariablen, die alle nur **für die Implementierung sichtbar** sind	**eine einzige Wertekombination** von Zustandsvariablen, die alle **an der öffentlichen Schnittstelle sichtbar** sind
zusammengesetzt	**mehrere Wertekombinationen** von Zustandsvariablen, die alle nur **für die Implementierung** sichtbar sind	**mehrere Wertekombinationen** von Zustandsvariablen, die alle **an der öffentlichen Schnittstelle sichtbar** sind

Tabelle 4.1 *Zusammenhang zwischen Granularität und Sichtbarkeit*

Zustandsvariablen sind Attribute, die zur Beschreibung von Zuständen von Instanzen einer Klasse verwendet werden. Diese bilden eine Teilmenge zu den sog. **Instanzvariablen**, welche die Menge aller Attribute einer Klasse repräsentieren. Hierbei ist natürlich jeweils wieder die Sichtbarkeit zu beachten, wodurch die Menge der jeweils einbezogenen Attribute verändert wird.

Die Definition der öffentlichen Schnittstelle einer Klasse direkt über deren Attribute ist für die Automatisierung des Tests kritisch, da sie implizit davon ausgeht, dass für jedes für die Definition verwendete Attribut eine entsprechende öffentliche, zustandserhaltende Methode implementiert ist, die als Rückgabeparameter genau den Wert des jeweiligen Attributs liefert. Dies ist allerdings nicht immer sinnvoll, wie das folgende Beispiel zeigt.

Man stelle sich eine Klasse vor, in der das Datum und die Uhrzeit durch eine Instanzvariable als Sekunden seit dem 01.01.1970 kodiert werden und in Abhängigkeit verschiedener zeitlicher Differenzen, wie z. B. einen Monat nach einem gespeicherten Termin, bestimmte Aktivitäten auszuführen sind. Eine Modellierung des entsprechenden Zustandsautomaten allein mit Hilfe der Repräsentation von Datum und Uhrzeit in Sekunden, also dem Wert der Instanzvariablen, ist nur sehr schwer möglich. Die Komplexität und Fehleranfälligkeit würden dabei stark ansteigen und die Verständlichkeit gleichzeitig erheblich sinken. Dies gilt sowohl für die Definition einzelner Zustände als auch für die an den Zustandsübergängen spezifizierten Bedingungen. Eine Implementierung, die einen solchen Automaten realisieren soll, wird ebenfalls mit diesen Problemen konfrontiert. Außerdem muss es auch für Klienten einer Klasse möglich sein, den aktuellen, abstrakten Zustand entsprechender Instanzen zu ermitteln. Die erhöhte Komplexität und Fehleranfälligkeit würden zusätzlich auf die Klienten einer solchen Klasse ausgeweitet werden. Die Modellierung und Implementierung würde sehr wahrscheinlich auf der Basis entsprechender Zugriffsoperationen erfolgen, welche die in der Instanzvariablen in Sekunden kodierte Repräsentation eines Zeitpunkts z. B. in die Werte Jahr, Monat, Tag, Stunde, Minute und Sekunde umsetzen. Für die Klienten sind nur diese Werte sichtbar, nicht aber der Wert der Instanzvariablen.

Aus diesen Überlegungen heraus erscheint es sinnvoll, einen abstrakten Zustand nicht direkt auf der Basis von Instanzvariablen zu beschreiben, sondern hierfür nur die Rückgabe- und Ausgangsparameter öffentlicher, zustandserhaltender Methoden der jeweiligen Klasse zu verwenden. Dies stellt auch eine direkte und konsequente Umsetzung des Konzepts der Datenkapselung dar. Außerdem wird die Schnittstelle einer Klasse korrekt so widergespiegelt wie sie von den Klienten der Klasse wirklich gesehen wird. Zusätzlich wird die Automatisierung des Tests wesentlich erleichtert. Auch hier gilt wieder die Einschränkung, dass nur Parameter von Methoden ohne Ein- und Durchgangsparameter die Basis des abstrakten Zustands sein können[74]. Ein Problem, welches bei der direkten Verwendung von Instanzvariablen für den Test und insbesondere für dessen Automatisierung entsteht, ist die Notwendigkeit einer Zuordnung von Instanzvariablen zu den entsprechenden Methoden, die den Wert der Instanzvariablen als Rückgabeparameter liefern. Wenn man bedenkt, dass es durchaus möglich ist, dass der Wert einer Instanzvariablen nicht als Rückgabeparameter, sondern als einfacher Ausgangsparameter zurückgegeben wird, ist sogar eine direkte Zuordnung einer Instanzvariablen zu einem Parameter einer entsprechenden Methode notwendig. Aus den oben genannten Gründen und da eine solche Zuordnung einen erheblichen Mehraufwand bei der Modellierung bedeuten würde, erfolgt im Rahmen des gesamten Testprofils die Definition der öffentlichen Schnittstelle direkt über die Rückgabe- und Ausgangsparameter von öffentlichen, zustandserhaltenden Methoden einer Klasse. Diese Methoden dürfen den Zustand des Systems nicht verändern, da der Zustand durch die Abfrage des Zustands selbst nicht verändert werden darf. Ansonsten wäre die erhaltene Information wertlos, weil sie nicht mehr aktuell wäre.

Für das *XFREE State Model* hat die Entscheidung hinsichtlich der Definition der öffentlichen Schnittstelle einer Klasse zur Folge, dass sämtliche Zustandsinvarianten sowie alle an den Zustandsübergängen annotierten Bedingungen, neben Konstanten, nur mit Rückgabe- und Ausgangsparametern öffentlicher, zustandserhaltender Methoden beschrieben werden dürfen.

Wie bereits erwähnt, ist es für die Modellierung eines testbaren Zustandsautomaten zwingend notwendig, dass zu jedem Zeitpunkt eindeutig festgestellt werden kann, in welchem der Zustände sich eine Instanz der entsprechenden Klasse befindet. Die entscheidende Bedingung hierfür ist, dass sich die durch die Zustandsinvarianten von zwei in einem Zustandsautomaten definierten Zuständen aufgespannten Zustandsräume entweder gegenseitig ausschließen oder der Zustandsraum des einen Zustandes eine echte Teilmenge des Zustandsraumes des anderen Zustandes ist. Sich teilweise überschneidende Zustände bzw. Zustandsräume sind also nicht erlaubt. In Abbildung 4.8 wird dies anschaulich dargestellt.

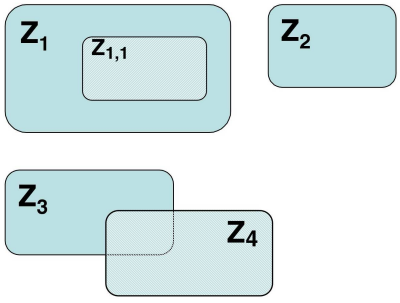


Abbildung 4.8 *Definition von Zuständen*

[74] siehe Erläuterungen zur Erzeugung von Orakelklassen in Abschnitt 4.5.3

Die Zustände Z_1, $Z_{1,1}$ und Z_2 in Abbildung 4.8 sind korrekt definiert, während die Zustände Z_3 und Z_4 nicht testbar sind. Durch die Einhaltung der genannten Beschränkungen lassen sich Probleme, wie sie in Abbildung 4.9 dargestellt werden, vermeiden.

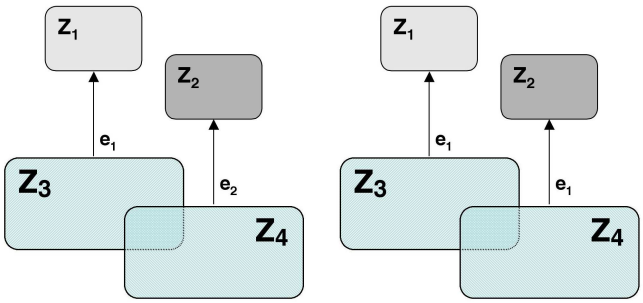


Abbildung 4.9 *Probleme bei Zustandsüberschneidungen*

In beiden in Abbildung 4.9 dargestellten Fällen ist es sowohl für eine Instanz der jeweiligen Klasse als auch für deren Klienten nicht möglich, jederzeit eindeutig den aktuellen Zustand festzustellen. Im linken Fall ließe sich deshalb für den sich überschneidenden Teil der beiden Zustände Z_3 und Z_4 nicht feststellen, ob die Nachrichten e_1 bzw. e_2 erlaubt sind. Und selbst, wenn man für den fraglichen Teil des Zustandsraumes sowohl e_1 als auch e_2 zulassen würde, bliebe das Problem, dass die Instanzen der Klasse innerhalb eines einzigen Zustandes ein unterschiedliches Verhalten zeigen würden. Dies widerspräche der Intention eines Zustandsautomaten. In der Praxis müssten die Zustandsinvarianten aller Zustände überprüft werden, um feststellen zu können, ob eine Nachricht erlaubt ist. Im rechten Beispiel kann für den sich überschneidenden Bereich von Z_3 und Z_4 gar nicht entschieden werden, welcher Zustandsübergang zu aktivieren ist.

Binder definiert für das *FREE State Model* weitere Bedingungen, die bei der Spezifikation der Zustandsinvarianten einzuhalten sind. Dabei gibt es den folgenden Zusammenhang zwischen der Klasseninvariante, welche den gesamten für die Instanzen der jeweiligen Klasse zulässigen Zustandsraum beschreibt, und den einzelnen Zustandsinvarianten:

> Wenn Z die Menge der durch alle Zustandsinvarianten eines Zustandsautomaten beschriebenen Wertekombinationen von Zustandsvariablen ist, und K die Menge aller für die Instanzen der Klasse zugelassenen Wertekombinationen von Instanzvariablen, so gilt $Z \equiv K$ und $\forall z \in Z \mid z \subseteq K$.

Jede einzelne Zustandsinvariante muss also stärker oder äquivalent zu der entsprechenden Klasseninvariante sein. Ist nur ein Zustand definiert, ist dessen Zustandsinvariante äquivalent zur Klasseninvariante. Dies ist allerdings wohl nur ein theoretischer Fall, da es sich dabei um eine *nicht-modale* Klasse handelt, deren Verhalten nicht sinnvoll durch einen Zustandsautomaten beschrieben werden kann. Innerhalb des *XFREE State Model* entsprechen die Begriffe Zustands- und Instanzvariable den Rückgabe- und Ausgangsparametern öffentlicher, zustandserhaltender Methoden, da diese für die Definition von Klassen- und Zustandsinvarianten verwendet werden

In der UML-Spezifikation ist definiert, dass, wenn ein Containerzustand aktiv ist, genau einer seiner Unterzustände ebenfalls aktiv ist[75]. Aus dieser Bedingung lässt sich eine weitere Einschränkung in Bezug auf Zustandsinvarianten ableiten.

[75] Dies gilt nur für Zustände, die nicht nebenläufig sind. Nebenläufigkeit wird im Rahmen dieser Arbeit aber nicht untersucht.

> Wenn Z die Menge der durch die Zustandsinvarianten aller direkten Unterzustände eines Containerzustandes S beschriebenen Wertekombinationen von Zustandsvariablen ist, und Z_s die Menge aller durch die Zustandsinvariante von S beschriebenen Wertekombinationen von Zustandsvariablen ist, so gilt $Z \equiv Z_s$ und $\forall z \in Z \mid z \subset Z_s$.

Die Summe der Zustandsinvarianten der direkten Unterzustände ist also äquivalent zur Zustandsinvariante des entsprechenden Containerzustandes und ein Containerzustand besitzt immer mindestens zwei direkte Unterzustände. Die zweite Bedingung entsteht dabei mehr aus praktischen Erwägungen, da ein direkter Unterzustand, wenn er der einzige direkte Unterzustand ist, äquivalent zu seinem Containerzustand ist. Eine solche Modellierung ist nicht sinnvoll und wird daher ausgeschlossen.

Zwischen den Zustandsinvarianten und den Vor- und Nachbedingungen der entsprechenden Methoden einer Klasse bestehen ebenfalls Abhängigkeiten. Die von Binder beschriebene Vorgehensweise zur Konstruktion von Zuständen aus den Nachbedingungen und der Klasseninvariante ist für die Anwendung in der Praxis problematisch. Binder schlägt dabei vor, die Zustände jeweils aus einzelnen, sich nicht überschneidenden und durch logische ODER-Operatoren verknüpften Teilausdrücke von Nachbedingungen der Methoden zu bilden. Dabei entstehen z. B. aus einer Nachbedingung *a or b* zwei Zustände. Hierbei wird allerdings der Zusammenhang zu den Vorbedingungen der Methoden überhaupt nicht beachtet. Außerdem erscheint es nicht als praktikabel und angemessen, das Verhalten von Instanzen einer Klasse durch Vor- und Nachbedingungen zu modellieren und daraus dann automatisch den Zustandsautomaten zu generieren. Dabei gibt man die Vorteile der Notation von Zustandsautomaten völlig aus der Hand. Wesentlich intuitiver und der Problemstellung der Konstruktion eines konsistenten Modells besser angepasst ist die umgekehrte Herangehensweise. Zuerst wird das Verhalten von Instanzen einer Klasse mit Hilfe eines Zustandsautomaten modelliert. Anschließend werden aus dem Automaten die Vor- und Nachbedingungen für die Methoden der Klasse ermittelt. Diese Vorgehensweise ist in Abbildung 4.10 dargestellt.

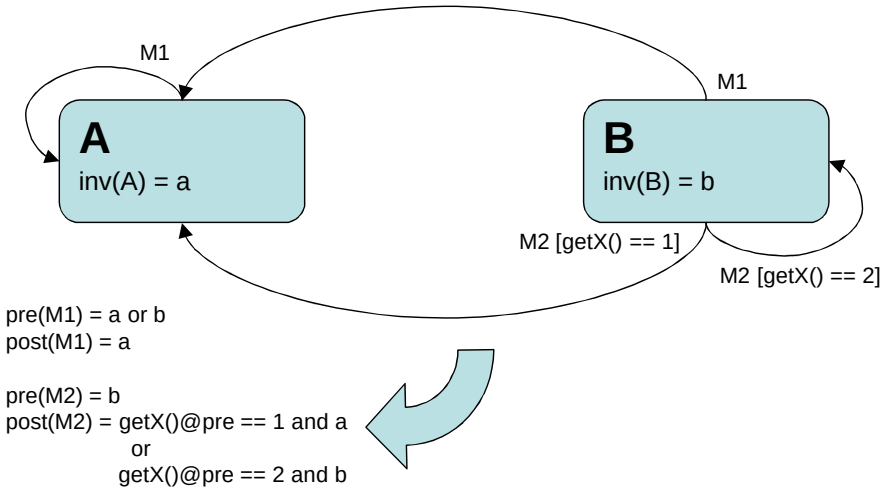


Abbildung 4.10 *Ermittlung von Vor- und Nachbedingungen aus einem Zustandsautomaten*

In dieser Abbildung kann man auch gleich die Zusammenhänge zwischen dem Zustandsautomaten und den Vor- und Nachbedingungen erkennen. Für jeden Zustandsübergang, an dem ein zu einer Methode gehöriges Ereignis annotiert ist, wird die Zustandsinvariante des Startzustandes mit einem ODER-Operator mit der restlichen

Vorbedingung verknüpft. Die Vorbedingung einer Methode besteht also aus der Summe aller Zustandsinvarianten der Zustände, in denen eine entsprechende Nachricht erlaubt ist. Die Nachbedingung einer Methode setzt sich entsprechend aus den Zustandsinvarianten von Zielzuständen von Zustandsübergängen zusammen, an denen ein mit der Methode verknüpftes Ereignis annotiert ist. Wichtig hierbei ist die Beachtung der an den jeweiligen Zustandsübergängen festgelegten Bedingungen, wie in Abbildung 4.10 dargestellt. Durch diese Vorgehensweise kann die Konsistenz zwischen den Methoden einer Klasse und dem das Verhalten dieser Klasse beschreibenden Zustandsautomaten sichergestellt werden. Die so gebildete Nachbedingung muss dann eventuell noch erweitert werden, wenn an den Zustandsübergängen Aktionen annotiert sind, die konkrete Auswirkungen auf den Zustand der Instanz besitzen. Wird z. B. der Wert der durch die Methode `getX()` zurückgegebenen Instanzvariable bei einem Zustandsübergang inkrementiert, so muss die Nachbedingung dies dokumentieren[76]. Bei der Erstellung der Vor- und Nachbedingungen muss nicht die vorgeschlagene Vorgehensweise angewendet werden. Wichtig ist nur, dass alle Vor- und Nachbedingungen konsistent zu dem modellierten Verhalten sind. Die Basis des Tests *modaler* Klassen ist in jedem Fall der jeweilige Zustandsautomat. Bei einer entsprechenden Spezifikation der Vor- und Nachbedingungen der Methoden einer modalen Klasse entstehen automatisch Redundanzen innerhalb des UML-Modells. Diese sind allerdings durchaus als sinnvoll zu betrachten. Sie erleichtern die Implementierung der Klasse dadurch, dass sie genau den durch diese Methode zu realisierenden Anteil des Verhaltens der gesamten Klasse modellieren. Dies bietet eine komprimierte Sicht für jede einzelne Methode. Während des Tests können außerdem Inkonsistenzen zwischen den Vor- und Nachbedingungen und dem durch den Zustandsautomaten modellierten Verhalten aufgedeckt werden.

Einen weiteren wichtigen Aspekt, den Binder in seinem Modell aufgreift, ist der Test von Konstruktoren und Destruktoren. Die in der UML-Spezifikation angegebene Notation der Start- und Endzustände[77] verkörpern entweder ein anderes Konzept oder sind unzureichend für den Test. Um dieses Problem zu beseitigen, führt Binder zwei spezielle Zustände ein: den Alpha- und den Omega-Zustand. Dabei verkörpert der **Alpha-Zustand** die Deklaration eines Objekts vor seiner Erzeugung und der **Omega-Zustand** wird nach der Zerstörung eines Objekts durch einen Destruktor erreicht. Für beide Zustände gilt, dass sie nicht durch eine Zustandsinvariante beschrieben werden können, da in beiden Fällen keine Instanz existiert. Die Verwendung des Startzustandes für die Modellierung des Verhaltens von Konstruktoren wirft verschiedene Probleme auf, auch wenn die UML-Spezifikation ihn dafür vorsieht. Da in der UML-Spezifikation definiert ist, dass ein Startzustand nur *maximal* einen ausgehenden Zustandsübergang[78] besitzen darf, ist die Modellierung eines Konstruktors, der Zustände mit unterschiedlichem Verhalten erzeugen kann, nicht möglich. Außerdem kann jeder Containerzustand einen Startzustand enthalten, während es für einen Zustandsautomaten nur genau einen Alpha-Zustand auf der obersten Ebene[79] geben kann. Der UML-Startzustand verkörpert also ein anderes Konzept, welches mit dem des Konstruktors nicht vereinbar ist. Um

[76] z.B. durch `getX()@pre == 1 and a and getX() == getX()@pre + 1`

[77] Ein Startzustand wird in der UML durch die Metaklasse `PseudoState` mit einem Tag `kind = #initial` modelliert. Ein Endzustand wird durch die Metaklasse `FinalState` repräsentiert.

[78] Die Semantik eines Startzustandes ohne ausgehenden Zustandsübergang wird in der UML-Spezifikation nicht erläutert.

[79] Diese oberste Ebene wird innerhalb eines Zustandsautomaten durch den durch das Tag `top` innerhalb der Metaklasse `StateMachine` referenzierten Zustand repräsentiert.

Letzteres angemessen modellieren zu können, wird innerhalb des Testprofils der Stereotyp «XFREEAlphaState» mit den erforderlichen Einschränkungen definiert[80].

Beim Vergleich zwischen UML-Endzustand und Omega-Zustand wird in dieser Arbeit eine etwas andere Auffassung vertreten als von Binder. Er ist der Meinung, dass wie beim Alpha-Zustand durch den Endzustand ein anderes Konzept realisiert wird als beim Omega-Zustand. Dies stimmt nicht und soll durch die Definition des Endzustandes in [OMG01] widerlegt werden:

> „When the final state is active, its containing composite state is *completed*, which means that it satisfies the completion condition. If the containing state is the top state, the entire state machine terminates, implying the termination of the entity associated with the state machine. If the state machine specifies the behavior of a classifier, it implies the „termination" of that instance."

Diese Definition macht genau die für die Modellierung des Konzepts des Omega-Zustands erforderliche Unterscheidung zwischen der obersten Ebene des Zustandsautomaten und dem Einsatz des Endzustandes auf tieferen Ebenen der Hierarchie. Dabei wird für die oberste Ebene explizit von der Termination der Instanz gesprochen, was der Zerstörung des Objekts entspricht. Außerdem verhindert die Definition der Metaklasse FinalState nicht die Erweiterungen, die für eine Präzisierung des Konzepts in Richtung Omega-Zustand notwendig sind. Deshalb wird innerhalb des Testprofils der Stereotyp «XFREEOmegaState» mit der Metaklasse FinalState als Basiselement mit den entsprechenden zusätzlichen Einschränkungen definiert[81].

Das Konzept des Omega-Zustands ist allerdings nur für Programmiersprachen von Relevanz, in denen Destruktoren als Programmierkonstrukt realisiert sind, die Zerstörung der Objekte also in der Verantwortung des Programmierers liegt. Binder vertritt zwar die Auffassung, wenn die Zerstörung von Objekten inhärent in der Laufzeitumgebung einer Programmiersprache integriert ist, dann muss der Omega-Zustand nicht explizit modelliert, aber dennoch getestet werden. In einem solchen Fall ist eine explizite Modellierung der Zustandsübergänge in den Omega-Zustand[82] aber gar nicht möglich. Es kann kein Ereignis angegeben werden, welches den Zustandsübergang auslösen würde und auch die Angabe einer Bedingung für den Zustandsübergang, wie z. B., dass keine Referenzen mehr auf das entsprechende Objekt existieren, sind nicht sinnvoll. Binders Forderung nach einem Test dieser impliziten Zustandsübergänge beachtet außerdem nicht die genaue Abgrenzung, die zwischen verschiedenen Testobjekten zu ziehen ist. In diesem konkreten Fall sind die Zustandsübergänge in den Omega-Zustand integraler Bestandteil der Laufzeitumgebung und als solche im Rahmen des Tests der Laufzeitumgebung zu testen. Bei deren Verwendung muss davon ausgegangen werden, dass sie entsprechend ihrer Spezifikation korrekt funktioniert. Diesen Mechanismus für jedes Programm explizit immer wieder zu testen, ist deshalb nicht sinnvoll. Wenn Binder mit seiner Forderung nach einem expliziten Test die Erfüllung der Voraussetzungen für einen Übergang in den Omega-Zustand meint, wie z. B., dass zu einem bestimmten Zeitpunkt keine Referenzen mehr auf ein Objekt existieren, so hat dies aber nichts mit dem Test der Zustandsübergänge in den Omega-Zustand zu tun.

---

[80] siehe Anhang A.5.2.8
[81] siehe Anhang A.5.2.9
[82] Zustandsübergänge in den Omega-Zustand werden von Binder als *Omega-Transition* bezeichnet.

Neben dem Konzept des Zustandes stellen Zustandsübergänge einen weiteren grundlegenden Baustein von Zustandsautomaten dar. Ein Zustandsübergang wird immer durch die folgenden Bestandteile beschrieben:

(1) Zustandsinvarianten von Anfangs- und Endzustand,

(2) einem Ereignis,

(3) einer optionalen Bedingung und

(4) einer optionalen Sequenz von Aktionen.

Ein Zustandsübergang besitzt immer genau einen Anfangs- und einen Endzustand. Das Ereignis stellt einen Stimulus dar, der einen Zustandsübergang auslösen kann. Wird mit einem Zustandsübergang kein Ereignis explizit verknüpft, so gilt eine Zuordnung des sog. impliziten Abschlussereignisses[83], welches genau dann ausgelöst wird, wenn alle entsprechend mit dem Startzustand verknüpften Aktivitäten beendet wurden. Die hierfür entscheidenden Aktivitäten können durch die in der UML definierten *Tags* `entry` und `doActivity` für einen Zustand spezifiziert werden, wobei das Letztere innerhalb des *XFREE State Model*, wie weiter unten in diesem Abschnitt erläutert wird, nicht verwendet werden darf. Wird für einen Zustandsübergang eine Bedingung definiert, so wird dieser nur aktiviert, wenn das zugehörige Ereignis eintritt und dann zu diesem Zeitpunkt die Bedingung erfüllt ist. Sind für einen Zustandsübergang Aktionen spezifiziert, so werden diese ausgeführt, sobald der Zustandsübergang aktiviert wird. Diese Bestandteile müssen bestimmten Bedingungen genügen, damit der zugehörige Zustandsautomat testbar ist. So darf z. B. die Zustandsinvariante des Endzustands des Zustandsübergangs nach der Ausführung der Aktionen nicht verletzt werden.

Eine weitere, wichtige zu erfüllende Bedingung besteht darin, dass der Zustandsautomat vollständig spezifiziert wird. Normalerweise werden in einem Zustandsautomaten nur die Ereignisse modelliert, die erlaubt sind. Dabei wird davon ausgegangen, dass alle nicht modellierten Ereignisse illegal sind. Da eine Klasse auch beim Auftreten eines solchen illegalen Ereignisses ein bestimmtes Verhalten zeigen soll, muss dieses Verhalten auch getestet und somit explizit modelliert werden. Zu diesem Zweck wurden im Paket `XFREE State Model` der Stereotyp «`IllegalTransition`» und die davon abgeleiteten Stereotypen «`SetFlagEvent`», «`RejectEvent`» und «`AbendEvent`» definiert. Wichtig für einen solchen Zustandsübergang ist, dass bei seiner Aktivierung der abstrakte Zustand der Instanz nicht verändert wird. In einem herkömmlichen Zustandsautomaten, in dem illegale Ereignisse nicht modelliert sind, wird implizit durch dieses Weglassen festgelegt, dass bei solchen Ereignissen der abstrakte Zustand nicht geändert wird. Durch die vom Stereotyp «`IllegalTransition`» abgeleiteten Stereotypen werden verschiedene Reaktionsmöglichkeiten der Klasse modelliert. Diese Menge kann durchaus durch die Spezifikation zusätzlicher Stereotypen erweitert werden. Das vollständige Ignorieren eines Ereignisses muss nicht explizit modelliert werden. Hier wird davon ausgegangen, dass für alle nicht modellierten Zustands-/Ereignispaare keinerlei Reaktion erfolgen soll. Diese Annahme kann gemacht werden, wenn sichergestellt ist, dass für alle illegalen Zustands-/Ereignispaare, für die eine Fehlerbehandlung realisiert werden soll, explizit entsprechende Zustandsübergänge modelliert worden sind. Da der abstrakte Zustand einer Instanz bei einem solchen

[83] engl. *completion event*

Zustandsübergang nicht verändert werden darf, muss es sich dabei um interne Zustandsübergänge handeln.

Für die vollständige Definition des Automaten fordert Binder außerdem, dass alle aus einer an einem Zustandsübergang annotierten Bedingung resultierenden Wahrheitswerte durch die Menge aller Bedingungen, die an Zustandsübergängen des gleichen Zustandes annotiert sind, abgedeckt werden müssen. In Abbildung 4.11 ist ein Beispiel dafür dargestellt.

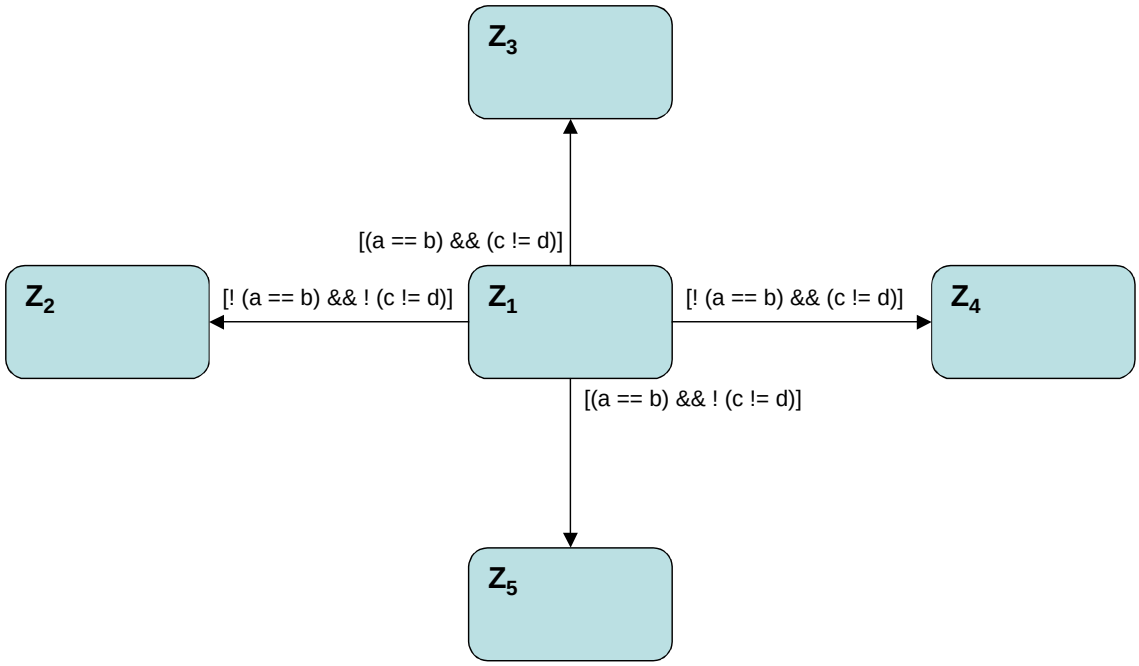


Abbildung 4.11 *Vollständige Definition von Zustandsübergangsbedingungen*

Binder bezieht seine Forderung allerdings auf die gesamte Menge der Zustandsübergänge und der damit verknüpften Bedingungen eines Zustandes. Das reicht allerdings nicht aus, da eine Bedingung an einem von einem Ereignis E_1 aktivierten Zustandsübergang in keinem Zusammenhang zu den Bedingungen steht, die an Zustandsübergängen mit einem anderen Ereignis E_2 annotiert sind. Diese werden nie zum gleichen Zeitpunkt ausgewertet. Im *XFREE State Model* wird die Forderung Binders dahingehend präzisiert, dass sie sich auf die Menge aller Zustandsübergänge mit dem gleichen Ereignis bezieht. Sollen nicht alle möglichen Kombinationen explizit modelliert werden, da in allen Fällen das gleiche Verhalten gefordert wird, können solche Bedingungen mit einem impliziten Zustandsübergang zusammengefasst werden. Realisiert wird dieses Konzept durch den Stereotypen «`ImplicitTransition`». Ein solcher Zustandsübergang besitzt keine Bedingung und wird erst aktiviert, wenn alle anderen ausgewerteten Bedingungen *falsch* waren. Somit stellt sie den sog. *else*-Fall dar. Dabei kann es in der Menge der Zustandsübergänge eines Zustandes mit dem gleichen Ereignis nur einen impliziten Zustandsübergang geben.

Beim Auftreten eines Ereignisses kann immer nur ein Zustandsübergang aktiviert werden. Besitzen mehrere Zustandsübergänge mit dem gleichen Anfangszustand das gleiche Ereignis als Stimulus, müssen die mit diesen Zustandsübergängen verknüpften Bedingungen ausgewertet werden, um feststellen zu können, welcher Zustandsübergang zu aktivieren ist. Werden dabei mehrere Bedingungen nach *wahr* ausgewertet, kann entsprechend der UML-Spezifikation trotzdem nur ein Zustandsübergang aktiviert werden. Welcher das ist, wird allerdings nicht festgelegt. Die Situation ist nicht deterministisch,

was für den Test nicht akzeptabel ist. Um dieser Schwäche der UML zu begegnen, wird im *FREE State Model* gefordert, dass sich die Bedingungen, die an den Zustandsübergängen eines Zustandes annotiert sind, gegenseitig ausschließen. Für alle Wertekombinationen der für die Definition der Bedingungen verwendeten Elemente darf nur eine Bedingung nach *wahr* auswerten. Bei Einhaltung dieser Forderung können viele Situationen nicht modelliert werden. Bei komplexen *if*-Konstrukten mit vielen *elseif*-Bedingungen kann man sich sehr gut Situationen vorstellen, bei denen sich die einzelnen Bedingungen gegenseitig überlappen. In Programmiersprachen entsteht dabei kein Konflikt, weil hier die Reihenfolge der Auswertung der einzelnen Bedingungen definiert ist, während dies für die UML nicht der Fall ist. Um dieses Problem für das *XFREE State Model* zu lösen, wurde im Testprofil das Konzept der Sequenzialität für die Auswertung von Bedingungen an Zustandsübergängen eingeführt. Realisiert wird dieses durch den Stereotypen «`OrderedTransition`». Die Zustandsübergänge eines Zustandes, die durch das gleiche Ereignis aktiviert werden und für die eine Bedingung definiert ist, müssen mit diesem Stereotypen markiert werden. Implizite Zustandsübergänge sind davon nicht betroffen, da sie sowieso immer erst dann aktiviert werden, wenn alle anderen Bedingungen nach *falsch* ausgewertet wurden. Durch die Verwendung dieses Stereotypen muss der Benutzer explizit eine eindeutige Auswertungsreihenfolge festlegen, was die Modellierung von Situationen zulässt, die mit dem *FREE State Model* nicht so einfach modelliert werden könnten.

Eine weitere Bedingung, die Binder für die Konstruktion eines testbaren Zustandsautomaten aufstellt, ist die alleinige Verwendung von Zuständen entsprechend der Definition von Mealy in [Mealy55]. Die Verwendung von Zuständen nach [Moore56] wird ausgeschlossen, da diese Zustände nicht testbar sind. Erwähnt wird diese Unterscheidung, da die UML-Spezifikation beide Typen von Zuständen zulässt. Beim Moore'schen Modell werden Zustände nicht durch eine Menge von Wertekombinationen definiert, wie bei Mealy, sondern mit den Zuständen werden Aktivitäten verknüpft. In der UML-Spezifikation werden solche Aktivitäten durch das *Tag* `doActivity` der Metaklasse `State` realisiert. Da nicht überprüft werden kann, ob ein Programm gerade eine bestimmte Aktivität ausführt, verbietet Binder diese für die Spezifikation eines testbaren Modells.

Ein grundlegendes Konzept der Objektorientierung besteht in der Vererbung. In Abschnitt 2.2.1 wurde bereits erläutert, warum es wichtig ist, auch die durch die Vererbung entstehenden Beziehungen zu testen. In Abschnitt 4.2 wurde außerdem dargelegt, welche Bedingungen für einen sicheren Einsatz polymorpher Strukturen bei nicht-modalen Klassen erfüllt sein müssen. Die Erfüllung dieser Bedingungen ist gleichzeitig die Voraussetzung dafür, dass Testfälle von ihren Oberklassen sinnvoll wiederverwendet werden können. Die Basis dafür stellt der Einsatz der Vererbung entsprechend dem *LSP* dar. Soll für modale Klassen ebenso ein sicherer Einsatz polymorpher Strukturen sowie eine Wiederverwendung von Testfällen ermöglicht werden, muss das *LSP* auf die Modellierung der entsprechenden Zustandsautomaten übertragen werden. Durch die UML-Spezifikation gibt es hierfür keinerlei Unterstützung. Im Rahmen des *FREE State Model* gibt Binder eine Vorgehensweise zur Konstruktion von Zustandsautomaten an, mit der es möglich sein soll, die Forderungen des *LSP* für den Zustandsautomaten zu erfüllen, und listet zusätzlich eine Reihe von Bedingungen auf, denen der Zustandsautomat genügen muss. Allerdings lassen die angegebenen Bedingungen an einigen Stellen Fragen offen, die durch eine genauere Spezifikation im Rahmen des *XFREE State Model* geklärt werden.

Die wichtigste Forderung, die Binder im Zusammenhang mit einer korrekten Anwendung der Vererbung für ein *FREE State Model* aufstellt, besagt, dass alle Ereignisse bzw. Zustandsübergänge, die in einer Oberklasse akzeptiert werden, auch in der Unterklasse akzeptiert werden müssen. Dieses lässt sich auch leicht aus der Forderung für nicht-modale Klassen ableiten, die auch für modale Klassen gilt, wonach die Vorbedingung einer Methode, die eine geerbte Methode überschreibt, äquivalent oder schwächer gegenüber der Vorbedingung der überschriebenen Methode sein muss. In diesem Zusammenhang sei noch einmal auf den engen Zusammenhang zwischen den Vor- und Nachbedingungen der Methoden und dem Zustandsautomaten einer Klasse hingewiesen[84].

In der Literatur wird die Beziehung zwischen Elementen, die für die Oberklasse und Unterklasse definiert werden, häufig so dargestellt, als ob es sich dabei um die Identität handelt [Binder99] [Warmer+99]. Dies ist aber nicht der Fall, denn in der UML besteht keinerlei Beziehung zwischen diesen Elementen. Es sind unterschiedliche Instanzen der gleichen Metaklasse. Man kann diese Elemente allerdings durch eine Zuordnung nach bestimmten Kriterien in eine Beziehung zueinander setzen. Für die Definition des *XFREE State Model* erfolgt nur eine Zuordnung von Zuständen der Unterklasse zu einem Zustand der Oberklasse. Die weiteren Definitionen basieren auf dieser Zuordnung. Für Zustände erfolgt diese Zuordnung entsprechend ihrer Zustandsinvarianten und ist wie folgt definiert:

(1) Alle Zustände z_u einer Unterklasse werden einem Zustand z_o einer ihrer direkten und indirekten Oberklassen zugeordnet, wenn die Zustandsinvariante $I(z_u)$ äquivalent oder stärker ist als $I(z_o)$.

Ein Zustandsautomat muss folgende Bedingungen erfüllen, um testbar zu sein:

(2) Für jeden Zustand z_o aller direkten und indirekten Oberklassen existiert eine nicht-leere Menge von Zuständen Z_u der Unterklasse, für die gilt, dass alle in Z_u enthaltenen Zustände z_u dem Zustand z_o der Oberklassen zugeordnet sind. Für genau einen Zustand $z_u \in Z_u$ gilt $I(z_u) \equiv I(z_o)$.

(3) Für jeden Zustandsübergang t_o der Oberklassen mit einem Anfangszustand s_o und einem Zielzustand z_o, welcher durch ein Ereignis e aktiviert wird und der durch die Bedingung g_o beschränkt wird, gilt Folgendes:

   (a) Für jeden Zustand z_u der Unterklasse aus der Menge der dem Zustand s_o zugeordneten Zustände gilt, dass die Summe der Bedingungen an allen Zustandsübergängen, deren Anfangszustand z_u oder ein dem Zustand z_u übergeordneter Zustand ist, die durch das Ereignis e aktiviert werden und als Zielzustand einen dem Zustand z_o zugeordneten Zustand besitzen, äquivalent oder schwächer ist als g_o.

   (b) Alle Zustandsübergänge t_u der Unterklasse, die von einem dem Zustand s_o zugeordneten Zustand ausgehen und durch ein Ereignis e aktiviert werden und für deren Bedingung $g_u \cap g_o \neq \varnothing$ gilt, besitzen einen Zielzustand, der dem Zustand z_o zugeordnet ist.

   (c) Ist der Zustandsübergang t_o ein legaler Zustandsübergang, er ist also nicht mit dem Stereotypen «`IllegalTransition`» markiert, so

[84] siehe Abbildung 4.10 auf Seite 69

muss dies auch für alle Zustandsübergänge t_u der Unterklasse gelten, die einen Anfangszustand und einen Zielzustand besitzen, die jeweils dem Zustand s_o bzw. dem Zustand t_o zugeordnet sind, die durch das Ereignis e aktiviert werden und für deren Bedingung $g_u \cap g_o \neq \varnothing$ gilt.

Bedingung (2) entspricht der sonst in der Literatur zu findenden Formulierung, dass kein Zustand aus dem Zustandsautomaten der Oberklasse im Zustandsautomaten der Unterklasse gelöscht werden darf. Diese Formulierung basiert auf der Zuordnung eines Zustands der Unterklasse zu einem Zustand der Oberklasse durch die Namen der Zustände. Der Name eines Zustandes hat für den Test allerdings nur rein syntaktische Bedeutung. Entscheidend ist die Semantik der Zustandsinvarianten. Deshalb basiert die Definition für das *XFREE State Model* auch auf diesen. Mit Bedingung (3)(a) wird sichergestellt, dass alle Ereignisse, welche von der Oberklasse akzeptiert werden, auch von der Unterklasse akzeptiert werden. In Abbildung 4.12 ist ein Beispiel dargestellt. Auf der linken Seite ist der Zustandsautomat einer Oberklasse und auf der rechten Seite ein entsprechender Zustandsautomat einer von dieser Oberklasse erbenden Klasse dargestellt.

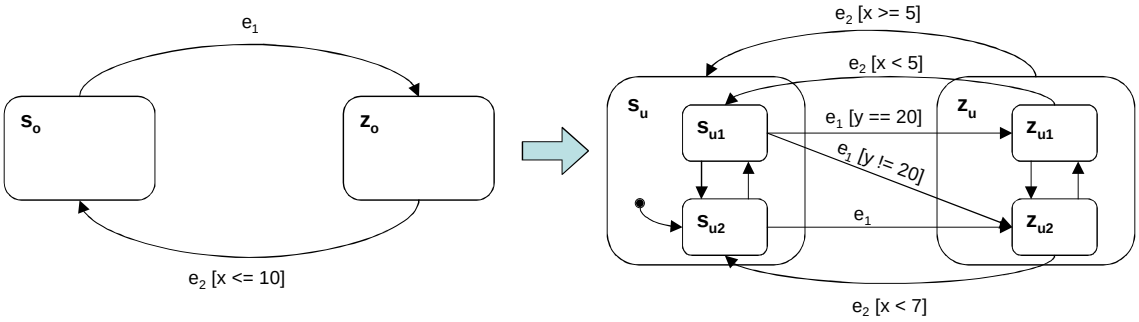


Abbildung 4.12 *Zusammenhang zwischen Zustandsübergängen bei der Vererbung*

Dabei werden die Zustände s_u, s_{u1} und s_{u2} dem Zustand s_o der Oberklasse zugeordnet, d.h. ihre Zustandsinvarianten sind äquivalent oder stärker als die Zustandsinvariante von Zustand s_o. Entsprechend der UML-Spezifikation ist immer genau ein Unterzustand eines Containerzustandes aktiv, wenn dieser Containerzustand selbst aktiv ist. Dies bedeutet, wenn der Zustand s_u aktiv ist, ist entweder der Zustand s_{u1} oder der Zustand s_{u2} ebenfalls aktiv. Für den Zustand z_u und seine Unterzustände gilt dies analog. Da in dem Zustandsautomaten der Oberklasse beim Auftreten des Ereignisses e_1 in dem Zustand s_o immer ein Zustandsübergang in den Zustand z_o erfolgt, muss für den Zustandsautomaten der Unterklasse gewährleistet werden, dass beim Auftreten von e_1 in dem Zustand s_u bzw. einem seiner Unterzustände[85] ein Zustandsübergang in den Zustand z_u, z_{u1} oder z_{u2} erfolgt. Da der Zustandsübergang $s_{u1} \rightarrow z_{u1}$ mit einer Bedingung verknüpft ist, muss es daher mindestens einen weiteren Zustandsübergang von s_{u1} nach z_u oder einem seiner Unterzustände geben. Wird dies nicht gewährleistet, würde also in dem dargestellten Beispiel der zweite Zustandsübergang $s_{u1} \rightarrow z_{u2}$ fehlen, würde das Ereignis e_1 im Zustand s_{u1} nicht akzeptiert werden. Wie man sich leicht vorstellen kann, stellt das ein Problem dar, wenn an einer Stelle in einem Programm, wo ein Objekt der Oberklasse erwartet wird, ein derartig unvollständiges Objekt auftritt. Der Klient verlässt sich auf die Definitionen der Oberklasse, die von der Unterklasse aber nicht eingehalten werden. Ebenso ist leicht einzusehen, dass man die Testfälle der Oberklasse nicht wiederverwenden

[85] s_u, s_{u1} und s_{u2} sind dem Zustand s_o zugeordnet.

kann, wenn die Spezifikation der Unterklasse genau so erfolgen sollte, dass es den Zustandsübergang $s_{u1} \rightarrow s_{u2}$ nicht gibt. Dann würden entsprechende Testfälle der Oberklasse Fehler melden, obwohl dies keine Fehler sind. Soll hingegen das *LSP* explizit bei der Spezifikation berücksichtigt werden und wurde der Zustandsautomat nur unvollständig spezifiziert, können die Testfälle der Oberklasse diese Fehler aufdecken.

Um zu erreichen, dass in dem Zustandsautomaten der erbenden Klasse beim Auftreten des Ereignisses e_1 im Zustand s_{u1} oder s_{u2} immer ein Zustandsübergang in einen dem Zustand z_o zugeordneten Zustand erfolgt, muss die Summe der Bedingungen der entsprechenden Zustandsübergänge in jedem Fall *wahr* ergeben, da an dem entsprechenden Zustandsübergang der Oberklasse keine Bedingung annotiert ist, was dem Wert *wahr* entspricht. Dies wird in dem angegebenen Beispiel durch die beiden Bedingungen *[y == 20]* und *[y != 20]* sichergestellt. Das Gleiche muss auch für den Zustand s_{u2} gelten, da es nichts nützt, wenn die Bedingung nur für einen der Unterzustände von s_u erfüllt ist. Ein Beispiel dafür, dass immer die Summe der Bedingungen von Zustandsübergängen eines Zustandes und von Zustandsübergängen der ihn enthaltenden Containerzustände zu betrachten ist, liefert der durch e_2 in der Oberklasse aktivierte Zustandsübergang. Dieser ist mit einer Bedingung verknüpft. In dem Automaten der Unterklasse reichen die entsprechenden Zustandsübergänge, die von den Zuständen z_{u1} bzw. z_{u2} ausgehen nicht aus, da die an diesen Übergängen annotierten Bedingungen stärker sind als die Bedingung an dem zugehörigen Zustandsübergang der Oberklasse. Tritt das Ereignis e_2 in einem dieser beiden Zustände ein und ist *[x == 8]*, dann wird der entsprechende Zustandsübergang nicht aktiviert. Dies wird durch den Zustandsübergang $z_u \rightarrow s_u$ mit seiner Bedingung *[x >= 5]* verhindert, da hierdurch die Lücke in dem Wertebereich bis *[x <=10]* vollständig abgedeckt wird.

Wichtig ist auch, dass der Zielzustand von durch die Vererbung bedingten Zustandsübergängen immer ein Zustand ist, der in der Menge der dem Zielzustand des zugehörigen Zustandsübergangs der Oberklasse zugeordneten Zuständen enthalten ist, wie von (3)(b) gefordert. Würde z. B. der Zustandsübergang, der von s_{u1} abgeht und mit der Bedingung *[y != 20]* verknüpft ist, als Zielzustand den Zustand s_u besitzen, entstünde wieder ein Problem. Ist e_1 ein sog. *CallEvent*, was mit einer Methode verknüpft ist, so wäre die Nachbedingung dieser Methode schwächer, als die der überschriebenen Methode der Oberklasse. Ein Klient der Oberklasse könnte sich nicht mehr auf deren Spezifikation verlassen, da diese von der Unterklasse verletzt würde.

Zu beachten ist, dass der Zustandsautomat der Oberklasse, und somit auch jener der Unterklasse, aus Gründen der Übersichtlichkeit nicht vollständig spezifiziert sind. Es fehlt eigentlich noch mindestens ein Zustandsübergang von z_o, der für das Ereignis e_2 den Fall *[x > 10]* spezifiziert.

Aus Abbildung 4.13 wird ersichtlich, dass für die Unterklasse ein Problem entsteht, wenn ein in der Oberklasse akzeptiertes Ereignis in der Unterklasse nicht mehr akzeptiert wird. In der Abbildung ist dies für die beiden Ereignisse e_1 und e_2 der Fall. Bei deren Auftreten in einem Objekt der Unterklasse würde eine Ausnahmebehandlung anstatt des erwarteten Zustandsübergangs erfolgen, worauf ein Klient der Oberklasse nicht vorbereitet wäre.

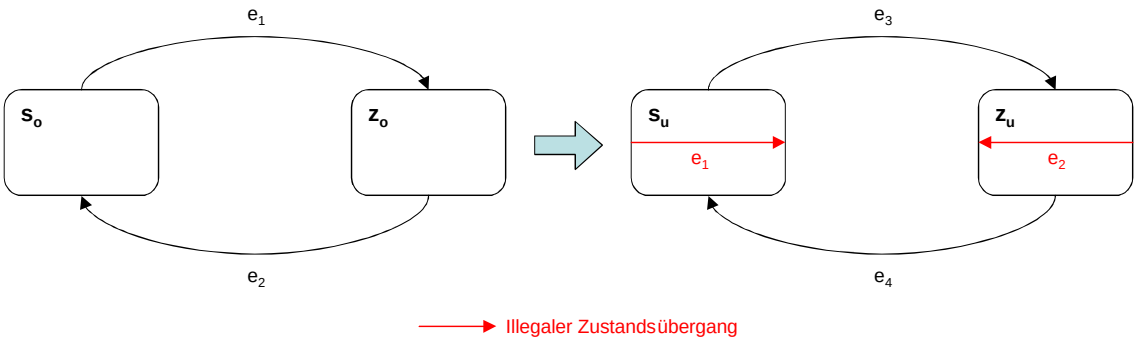


Abbildung 4.13 *Probleme mit illegalen Zustandsübergängen bei der Vererbung*

Ein wesentliches Merkmal des *FREE State Model* besteht in der Möglichkeit, neben den lokal in der zu testenden Klasse definierten auch die geerbten Eigenschaften systematisch zu testen. Zu diesem Zweck bietet Binder eine Form der Modellierung eines Zustandsautomaten für eine Klasse an, die sehr gut dazu geeignet ist, das gesamte Verhalten der Klasse, inklusive der geerbten Eigenschaften, übersichtlich darzustellen. Diese basiert auf einer flachen Repräsentation der Klassenhierarchie innerhalb eines Zustandsautomaten einer erbenden Klasse. In Abbildung 4.14 wird die Erstellung solcher Zustandsautomaten einer Klassenhierarchie gezeigt, in der jede Unterklasse alle nichtprivaten Eigenschaften seiner Oberklasse erbt, ohne sie erneut zu definieren, und zusätzlich selbst einige lokale Eigenschaften definiert[86]. In [Binder99] werden weitere Möglichkeiten dargestellt.

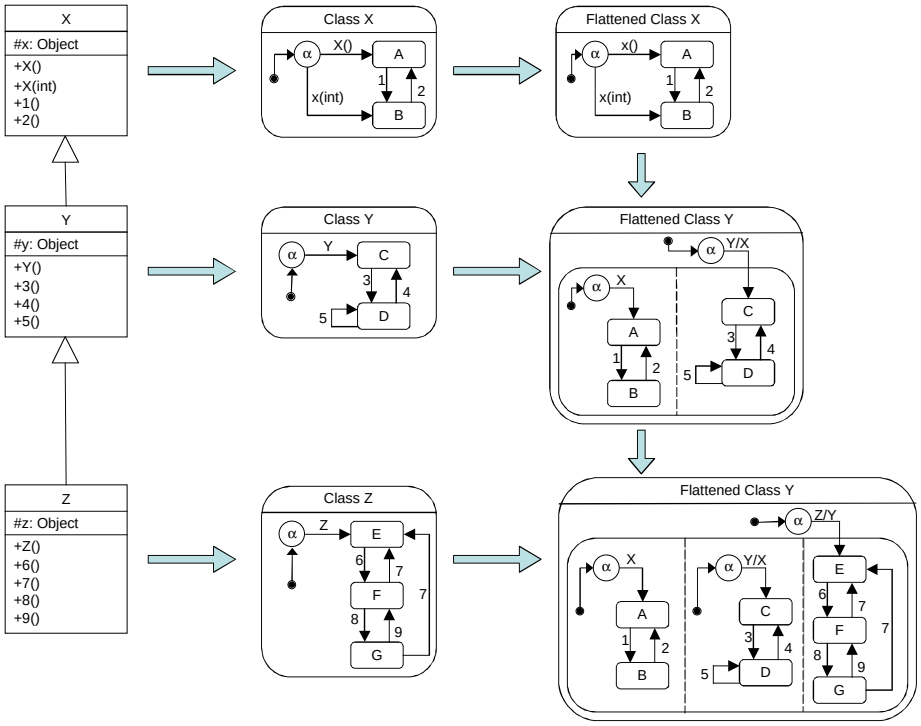


Abbildung 4.14 *Flache Sicht der Zustandsautomaten einer orthogonalen Klassenhierarchie*

[86] Dieser Fall wird von Binder als *orthogonal composition* bezeichnet.

Das in Abbildung 4.14 dargestellte Beispiel wurde aus [Binder99] entnommen und an die Erfordernisse des *XFREE State Model* angepasst. Durch die vorgenommenen Änderungen wird die Semantik der UML-Spezifikation besser umgesetzt als im Originalbeispiel von Binder. Auf der linken Seite ist die entsprechende Klassenhierarchie dargestellt. In der Mitte wurde jeweils das Verhalten der lokal in den drei Klassen definierten Eigenschaften spezifiziert. Dabei ist zu erkennen, dass die Klassen *Y* und *Z* jeweils orthogonale Zustände zu ihren Oberklassen definieren. Dies ist genau dann der Fall, wenn die lokal in der erbenden Klasse definierten Methoden keine der von ihren Oberklassen geerbten Eigenschaften verwenden. Auf der rechten Seite sind die flachen Zustandsautomaten dargestellt, die sich jeweils aus dem lokalen und den geerbten orthogonalen Automaten zusammensetzt. Innerhalb des Testprofils wird dieses Konzept durch die Stereotypen «`AbstractFlattenedXFREEStateMachine`», «`AbstractFlattenedXFREEState`» und «`AbstractOrthogonalXFREEState`» realisiert. Dabei muss der Wurzelzustand eines mit dem Stereotypen «`AbstractFlattenedXFREEStateMachine`» markierten Zustandsautomaten genau einen mit dem Stereotypen «`AbstractFlattenedXFREEState`» markierten Zustand enthalten. Diesem wiederum ist eine Menge von Unterzuständen zugeordnet, die mit dem Stereotypen «`AbstractOrthogonalXFREEState`» markiert sind und von denen jeder einen orthogonalen Zustandsautomaten der Vererbungshierarchie enthält. Ein lokaler Zustandsautomat einer Klasse ist genau dann orthogonal zu einem orthogonalen Zustandsautomaten seiner Oberklasse, wenn die folgenden beiden Bedingungen erfüllt sind:

(1) Die lokalen Methoden überschreiben keine der Methoden, die mit den Ereignissen verknüpft sind, die zur Definition des orthogonalen Zustandsautomaten der Oberklasse verwendet werden und

(2) die Vor- und Nachbedingungen aller lokal definierten Methoden verwenden zu ihrer Definition keine Zustandsvariablen, die zur Definition der Zustände des orthogonalen Zustandsautomaten der Oberklasse verwendet werden.

Ist die erste Bedingung nicht erfüllt, kann die zweite Bedingung ebenfalls nicht erfüllt sein, da die Vor- und Nachbedingungen der überschreibenden Methode in einer besonderen Beziehung zu den Vor- bzw. Nachbedingungen der überschriebenen Methode stehen[87]. Dagegen kann die erste Bedingung erfüllt sein, während die zweite Bedingung nicht erfüllt ist.

Durch die Verwendung eines eigenen Alpha-Zustandes für jeden orthogonalen Zustandsautomaten sowie die Ereignisse und Aktionen an deren Zustandsübergängen wird die stufenweise Aktivierung der Konstruktoren entlang der Vererbungshierarchie modelliert. Der lokale Alpha-Zustand ist direkt innerhalb des Wurzelzustands angeordnet. Die Erzeugung einer neuen Instanz der Klasse *Z* aus Abbildung 4.14 beginnt damit, dass eine Variable der Klasse *Z* deklariert wird. An dieser Stelle wird der Zustandsübergang vom Startzustand im Wurzelzustand zum lokalen Alpha-Zustand aktiviert. Danach befindet sich das Objekt im Alpha-Zustand. Der Alpha-Zustand ist kein realer Zustand, da noch keine Instanz erzeugt wurde. Erst mit einem Aufruf des lokalen Konstruktors *Z()* wird eine Instanz erzeugt und die Aktion für den Aufruf des Konstruktors der Oberklasse aktiviert. Durch den Zustandsübergang in den lokalen Zustand *E* werden die geerbten Alpha-Zustände aktiviert. Danach folgt die Abarbeitung des Konstruktoraufrufs der

[87] siehe Abschnitt 4.2

Oberklasse. Die Ausführungssemantik, wie sie in [OMG01] definiert ist, garantiert, dass die Zustandsübergänge in die geerbten Alpha-Zustände vor dem Aufruf des Konstruktors der Oberklasse erfolgen[88]. Die Abarbeitung der Konstruktornachricht *Y()* löst die Aktion zum Aufruf des Konstruktors *X()* aus und wechselt in den Zustand *C*. Dies wird solange durchgeführt, bis der Konstruktor der Wurzelklasse ausgeführt wurde. Erst dann ist die Instanz vollständig erzeugt.

Auf einen flachen Zustandsautomaten kann ein Muster zur zustandsbasierten Erzeugung einer Nachrichtensequenz nicht direkt angewendet werden, da diese auf einem normalen Zustandsautomaten basieren. Zu diesem Zweck muss der flache Zustandsautomat expandiert werden. Das dabei entstehende Ergebnis ist eine Produktmaschine. Algorithmen zur Berechnung von Produktmaschinen können in [Hopcroft+79], [Lewis+81] und [Holzmann91] gefunden werden. Für das in Abbildung 4.14 dargestellte Beispiel wird die Produktmaschine in Abbildung 4.15 dargestellt.

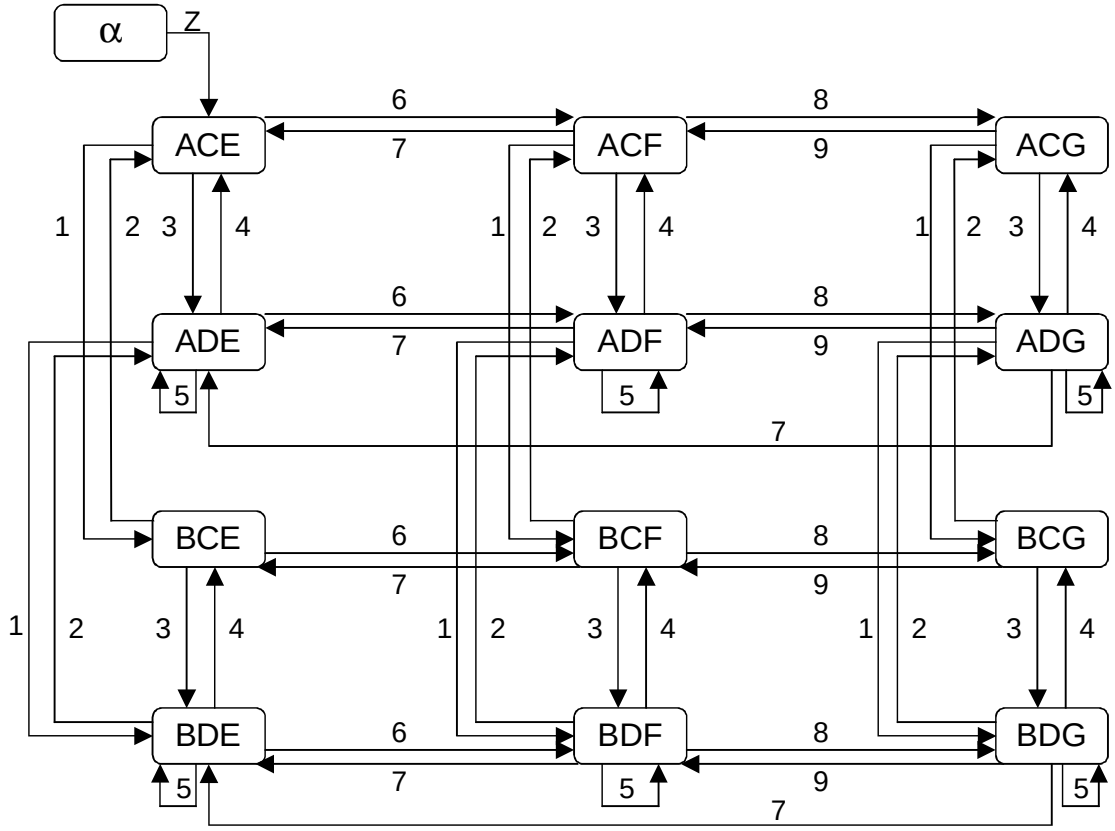


Abbildung 4.15 *Eine Produktmaschine als Ergebnis der Expansion eines flachen Zustandsautomaten*

Die Expansion eines flachen Zustandsautomaten ist ein sehr aufwendiger und fehleranfälliger Prozess und die dabei entstehende Produktmaschine ist nicht mehr so übersichtlich und gut verständlich, wie der flache Zustandsautomat. Da eine Automatisierung der Expandierung sehr gut möglich ist, kann diese Aufgabe durch das Testwerkzeug übernommen werden, welches das Testprofil realisiert. Der Benutzer muss mit dem flachen Zustandsautomaten also lediglich die Basis dafür zur Verfügung stellen.

[88] Die Art der Ereignisverarbeitung wird in [OMG01] als *run-to-completion step* bezeichnet.

4.5.5 Method Properties

Die in dem Paket `Method Properties` definierten Stereotypen dienen der Einteilung von Methoden nach verschiedenen Kriterien. So gibt es zustandserhaltende Methoden, realisiert durch den Stereotypen «`Accessor`» und die davon abgeleiteten Stereotypen, und zustandsverändernde Methoden. Diese werden durch den Stereotypen «`Modifier`» und die davon abgeleiteten Stereotypen repräsentiert. Konstruktoren werden dabei als zustandsverändernde Methoden betrachtet.

Mit dem Stereotypen «`SimpleAccessor`» werden Methoden markiert, die die Werte von Attributen unverändert über die Rückgabe- bzw. Ausgangsparameter zurückgeben. Durch den Stereotypen «`ComplexAccessor`» werden Methoden gekennzeichnet, die die Werte von Attributen verändert zurückgeben. Das heißt, vor der Zuweisung an einen Parameter werden verschiedene Aktionen durchgeführt. Für die Stereotypen «`SimpleModifier`» und «`ComplexModifier`» gilt dies analog, nur dass es hierbei um die Abbildung von Ein- und Durchgangsparametern auf die Attribute geht. Der Stereotyp «`Destructor`» spielt eine besondere Rolle, da dieser zwar den Zustand einer Instanz verändert, nach seiner Ausführung die Instanz aber zerstört ist.

Die Einteilung in diese Kategorien von Methoden spiegelt die in [Binder99] im Rahmen des *Alpha-Omega Zyklus'* vorgeschlagene Einteilung von Methoden zumindest teilweise wieder. Die dort vorgeschlagene Gruppe der *Iteratoren* wird hier unter den Stereotypen «`ComplexAccessor`» bzw. «`ComplexModifier`» zusammengefasst.

4.5.6 Executable Expression

Durch die Verwendung der im Paket `Executable Expression` definierten Stereotypen wird die Ausführbarkeit der entsprechenden Ausdrücke gewährleistet, was eine Voraussetzung für die Automatisierung des Tests ist. Durch den Stereotypen «`AbstractExecutableBooleanExpression`» werden logische Ausdrücke repräsentiert, zu deren Definition nur Elemente der öffentlichen Schnittstelle verwendet werden. Dazu gehören alle Rückgabe- und Ausgangsparameter von öffentlichen, zustandserhaltenden Methoden, die keine Ein- und Durchgangsparameter besitzen. Die Begründung für die Einschränkung hinsichtlich der Ein- und Durchgangsparameter kann Abschnitt 4.5.3 entnommen werden.

Mit Hilfe des Stereotypen «`AbstractInternalBooleanExpression`» wird zugesichert, dass der damit markierte Ausdruck so definiert ist, dass daraus automatisch ausführbarer Code generiert werden kann. Dazu müssen alle verwendeten Operatoren und Operationen auf den entsprechenden Typen definiert und über dessen öffentliche Schnittstelle zugänglich sein. Ist es nicht möglich, den Ausdruck so zu definieren, muss mit Hilfe des Stereotypen «`AbstractExternalBooleanWrapper`» eine Methode referenziert werden, die den entsprechenden Ausdruck auswertet. Diese Methode muss zustandserhaltend sein und eine bestimmte Signatur aufweisen. Der Rückgabeparameter muss vom Typ `boolean` sein und für jedes Element der öffentlichen Schnittstelle muss ein entsprechender Eingangsparameter definiert sein.

Diese boolschen Ausdrücke werden innerhalb des Testprofils zur Definition sämtlicher Vor- und Nachbedingungen, Klasseninvarianten, Zustandsinvarianten und Bedingungen an Zustandsübergängen verwendet, um die Automatisierung des Tests zu gewährleisten.

Kapitel 5

Anwendung des Testprofils

Nachdem in Kapitel 4 die Struktur des Testprofils dargelegt wurde, soll in diesem Kapitel dessen mögliche Anwendung exemplarisch an einem Beispiel gezeigt werden. Als Grundlage dient ein Beispiel aus [Binder99], womit ein Ballspiel modelliert werden soll. Aus Gründen der Übersichtlichkeit wurde das Beispiel leicht angepasst.

Bei dem Spiel geht es in der ersten Ausbaustufe um ein Ballspiel für zwei Personen, wie z. B. Squash oder Ping-Pong. In einem zweiten Schritt soll es so erweitert werden, dass es von drei Personen gespielt werden kann. Der Ablauf eines Spiels gestaltet sich folgendermaßen:

- Das Spiel wird gestartet.
- Beide Spieler betätigen eine Schaltfläche. Der Spieler, der die Schaltfläche zuerst betätigt hat, erhält den ersten Aufschlag.
- Beide Spieler versuchen den Ball immer wieder zurückzuschlagen, bis einer der beiden Spieler den Ball verfehlt.
- Verfehlt der aufschlagende Spieler den Ball, erhält der gegnerische Spieler den Aufschlag.
- Verfehlt der nicht-aufschlagende Spieler den Ball, wird der Punktestand des aufschlagenden Spielers um einen Punkt erhöht.
- Hat ein Spieler 21 Punkte erreicht, hat er das Spiel gewonnen.
- Das Spiel kann jederzeit neu gestartet oder auch beendet werden.

Das Verhalten des Spiels wird durch den in Abbildung 5.1 dargestellten Zustandsautomaten modelliert. Dieser Zustandsautomat entspricht dem *XFREE State Model* und ist somit testbar. Für die einzelnen Zustände wurden jeweils Zustandsinvarianten definiert, so dass jederzeit eindeutig der Zustand einer entsprechenden Instanz festgestellt werden kann. Dabei ist es wichtig, dass es keine sich teilweise überschneidenden Zustandsinvarianten gibt. Dies ist für den dargestellten Automaten sichergestellt. Für die von den beiden Zuständen `Player 1 Served` und `Player 2 Served` abgehenden Zustandsübergänge wurden jeweils über die Tagdefinition `evaluationPosition` des Stereotypen «`OrderedTransition`» die Auswertungsreihenfolge der annotierten Bedingungen festgelegt. Außerdem wurden die Wahrheitswerte der an den Zustandsübergängen annotierten Bedingungen entsprechend den Forderungen aus Abschnitt 4.5.4 vollständig spezifiziert.

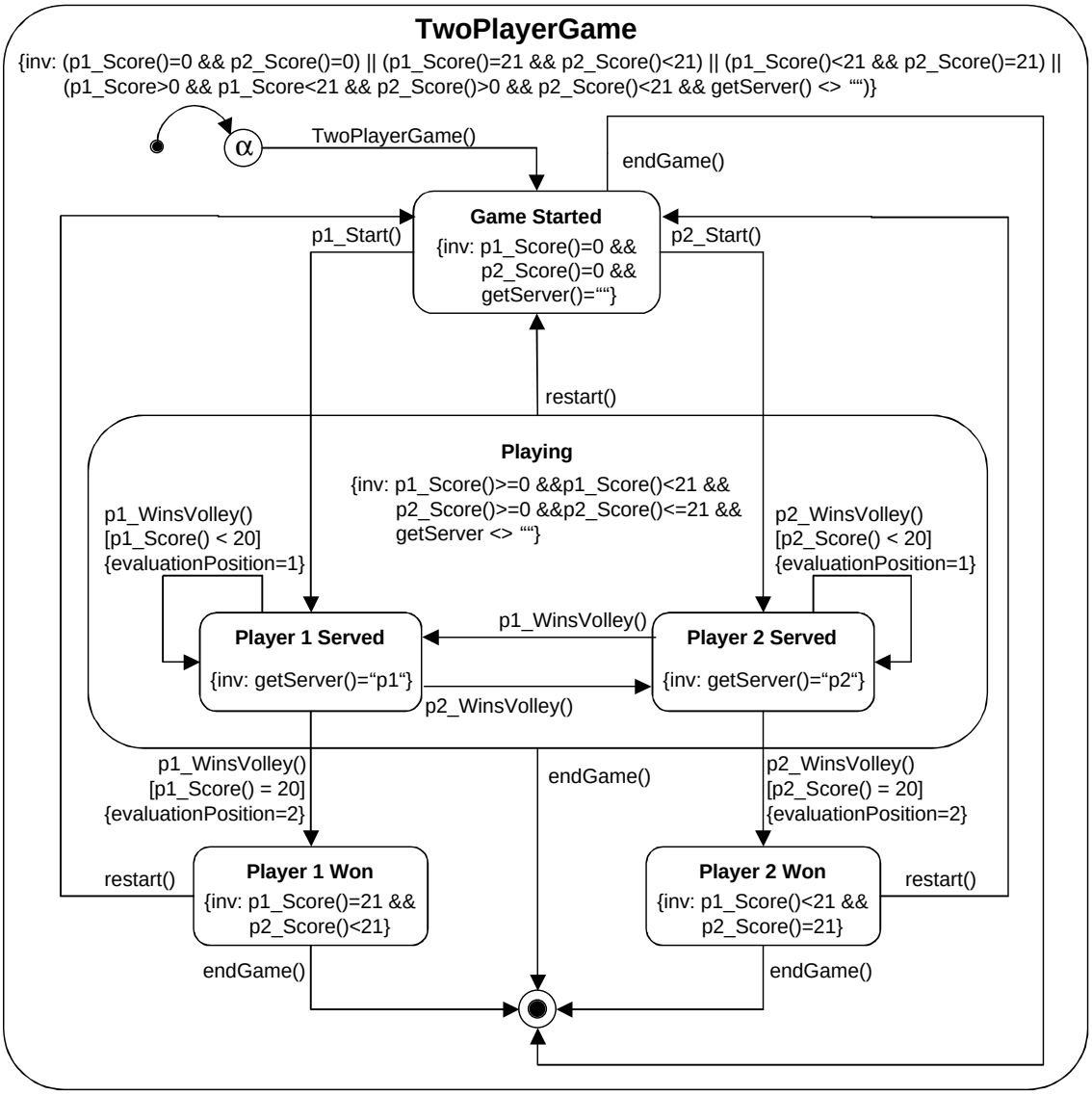


Abbildung 5.1 *Zustandsautomat der Klasse* `TwoPlayerGame`

Aus Gründen der Übersichtlichkeit werden die Stereotypen nicht explizit dargestellt. Wichtig dabei ist, dass die entsprechenden Einschränkungen der jeweiligen Stereotypen, hier «`AbstractXFREEStateMachine`», «`AbstractXFREEState`», «`XFREEAlphaState`» und «`AbstractXFREETransition`», eingehalten werden. An den einzelnen Zustandsübergängen auszuführende Aktionen wurden ebenfalls nicht modelliert, da sie keinen direkten Einfluss auf den Test besitzen.

In Abbildung 5.2 ist der gesamte Zustandsraum für die Klasse `TwoPlayerGame` dargestellt. Daran lassen sich sehr anschaulich die Zustandsinvarinaten überprüfen. Der Zustand `Playing` wurde nicht explizit dargestellt, ergibt sich aber genau aus der Vereinigung seiner beiden Unterzustände `Player 1 Served` und `Player 2 Served`. In der Grafik sind die explizit aus dem gültigen Zustandsraum der Klasse ausgeschlossenen Bereiche sehr gut zu erkennen.

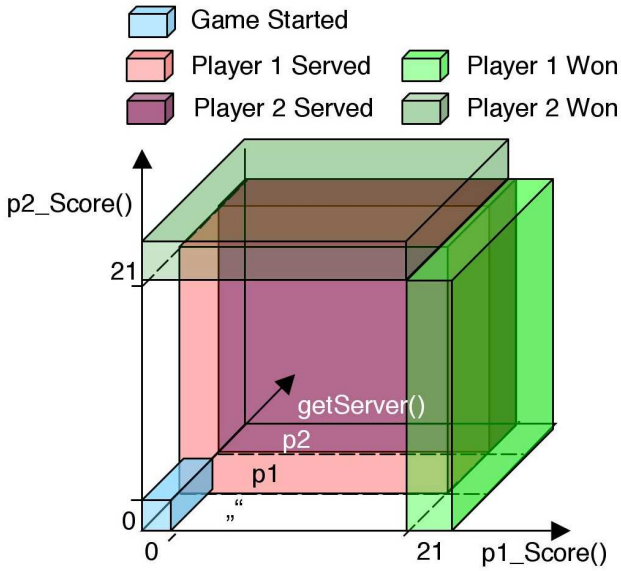


Abbildung 5.2 *Zustandsraum für die Klasse `TwoPlayerGame`*

Die zu dem Zustandsautomaten gehörige Klasse `TwoPlayerGame` ist in Abbildung 5.3 dargestellt.

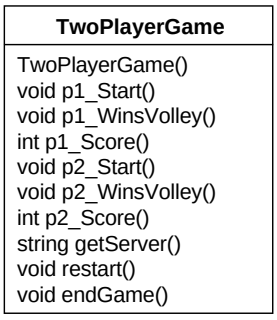


Abbildung 5.3 *Klassendiagramm für die Klasse `TwoPlayerGame`*

Sollen jetzt auf diese Klasse einzelne Testmuster angewendet werden, müssen zunächst die entsprechenden Elemente des Testprofils erweitert werden. Dabei muss durch Vererbung ein neues Testmuster von einem der Stereotypen «`DirectClassScopeTestPattern`» oder «`ReusableClassScopeTestPattern`» erzeugt werden.

Als Erstes wird die Erzeugung eines direkt anzuwendenden Testmusters erläutert und in Abbildung 5.4 dargestellt. Zur Erzeugung eines neuen Testmusters wird zuerst von dem Stereotypen «`DirectClassScopeTestPattern`» ein neuer konkreter Stereotyp «`NPlusTestPattern`» abgeleitet. Entsprechend der in dem Stereotypen «`DirectClassScopeTestPattern`» definierten Regeln zur Konstruktion eines neuen direkt anzuwendenden Testmusters[89] müssen nun die verschiedenen Tagdefinitionen zur Festlegung von Mustern für die Nachrichtensequenzerzeugung, Testdatenerzeugung und Testauswertung spezifiziert werden. Durch die Festlegung der Multiplizität für die einzelnen Tagdefinitionen wird bestimmt, wie viele Elemente jeweils bei der Anwendung des Testmusters verwendet werden dürfen. Für das Testmuster «`NPlusTestPattern`» kann danach ein Muster zur Erzeugung von Nachrichtensequenzen (`NPlus_MS`), eines für die Testdatenerzeugung (`NPlus_TD`) und mindestens

[89] siehe Anhang A.1.2.2

eine Zuordnung von elementspezifischen Mustern zur Testauswertung zu einem Modellelement (`NPlus_EP`) eingesetzt werden. Bei der Spezifizierung der einzelnen Tagdefinitionen ist darauf zu achten, dass deren Namen eindeutig sind. Dieses gilt für alle auf das gleiche Testobjekt anzuwendenden Testmuster bzw. den darin spezifizierten Tagdefinitionen. Bei einer Verletzung dieser Bedingung kann eine korrekte Zuordnung der einzelnen Tagwerte zu dem entsprechenden Testmuster nicht mehr gewährleistet werden[90].

Für die Anwendung des Testmusters auf ein konkretes Testobjekt, wie dies in Abbildung 5.4 dargestellt wird, müssen für die Festlegung von Tagwerten für die einzelnen Tagdefinitionen entsprechende Elemente definiert werden. Dies wird in dem Beispiel durch die Anwendung der Stereotypen «`NPlusStrategy`», «`Invariant-Boundaries`», «`StateEquality`», «`DeepEquality`» und «`AbstractElementEvaluationPatternSet`» auf die Klassen `NPlusSequences`, `PolySys3InvBound`, `StateEqualityEval`, `DeepEqualityEval` bzw. `StateEqualityAndDeepEquality` erreicht. Dabei werden durch die Belegung von in diesen Stereotypen spezifizierten Tagdefinitionen mit konkreten Tagwerten die Eigenschaften der entsprechenden Elemente festgelegt.

Zusätzlich zu diesen testmusterspezifischen Eigenschaften wird noch die testobjektspezifische Eigenschaft `NPlus_MO` zur Festlegung der in den Test einzubeziehenden Methoden und Operationen definiert. Bei der in Abbildung 5.4 dargestellten Anwendung des Testmusters wird zur Vereinfachung für die Festlegung dieser Eigenschaft das Schlüsselwort `all` verwendet. Dadurch wird ausgedrückt, dass alle Methoden der Klasse in den Test einbezogen werden sollen.

Für die Anwendung des direkt anzuwendenden Testmusters «`NPlusTestPattern`» auf ein anderes Testobjekt können für die Festlegung der einzelnen Tagdefinitionen neue Elemente mit entsprechend angepassten Eigenschaften definiert werden. Dies macht allerdings grundsätzlich nur für die Tagdefinitionen Sinn, bei denen für die entsprechenden Stereotypen wiederum Eigenschaften definiert sind, die neu belegt werden können. In dem dargestellten Beispiel gilt dies für die beiden Tagdefinitionen `NPlus_TD` und `NPlus_EP`.

[90] siehe Abschnitt 4.4 und insbesondere Abbildung 4.4

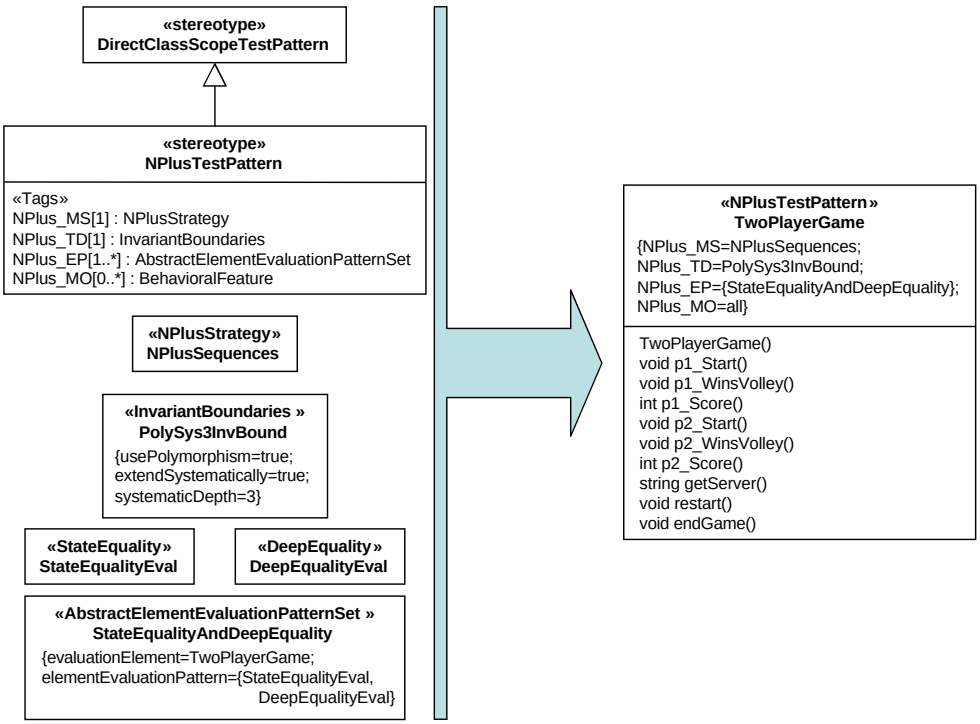


Abbildung 5.4 *Erzeugung und Anwendung eines direkt anzuwendenden Testmusters*

Soll dieses Testmuster mit genau den gleichen testmusterspezifischen Tagwerten, wie in Abbildung 5.4 dargestellt, auf mehrere Testobjekte angewendet werden, bietet sich die Definition eines entsprechenden wiederverwendbaren Testmusters an. Dies wird in Abbildung 5.5 beispielhaft gezeigt. Dabei wird durch Vererbung vom Stereotypen «`ReusableClassScopeTestPattern`» der neue Stereotyp, und damit das Testmuster «`NPlusTestPattern`» erzeugt. Auf diesen Stereotypen werden dann die entsprechenden Stereotypen zur Festlegung des Musters für die Nachrichtensequenzgenerierung und die Testdatenerzeugung sowie zur Zuordnung von elementspezifischen Auswertungsmustern zu einem Element angewendet. Dabei werden deren Tagdefinitionen konkrete Tagwerte zugewiesen, die dann für dieses Testmuster bei jeder Anwendung gelten. Dadurch ist es möglich, dieses Testmuster mit den gleichen Parametern auf mehrere Testobjekte anzuwenden, ohne diese jedes Mal erneut mit identischen Werten belegen zu müssen, wie dies bei den direkt anzuwendenden Testmustern der Fall wäre.

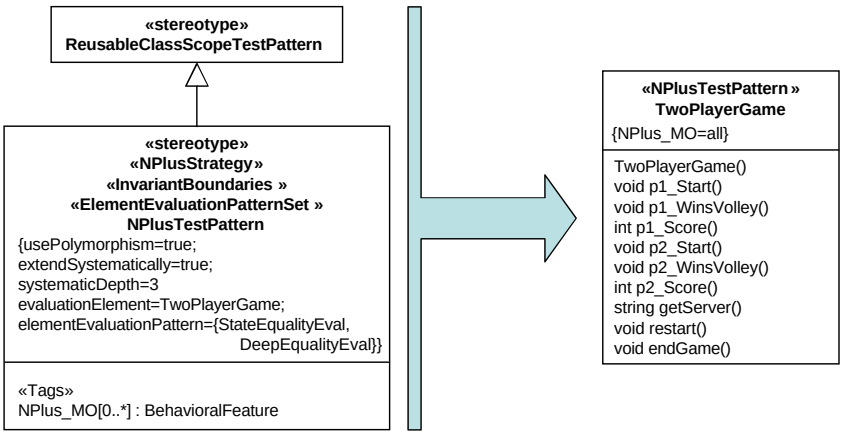


Abbildung 5.5 *Erzeugung und Anwendung eines wiederverwendbaren Testmusters*

Eine Voraussetzung für die Verwendung von wiederverwendbaren Testmustern ist die Eindeutigkeit der Namen der Tagdefinitionen der verschiedenen Komponentenstereotypen. Dies muss insbesondere bei der Erweiterung des Testprofils um weitere Komponentenstereotypen beachtet werden.

Um für den Test der Klasse `TwoPlayerGame` das für das Auswertungsmuster `DeepEquality` erforderliche externe Orakel zu erzeugen, muss von dem Testobjekt `TwoPlayerGame` durch Vererbung eine neue Klasse abgeleitet werden, die mit dem Stereotypen «`AbstractExternalOracle`» zu markieren ist. In dieser Orakelklasse müssen alle Methoden des Testobjekts überschrieben und das Verhalten der entsprechenden zu testenden Methoden so gut wie möglich simuliert werden.

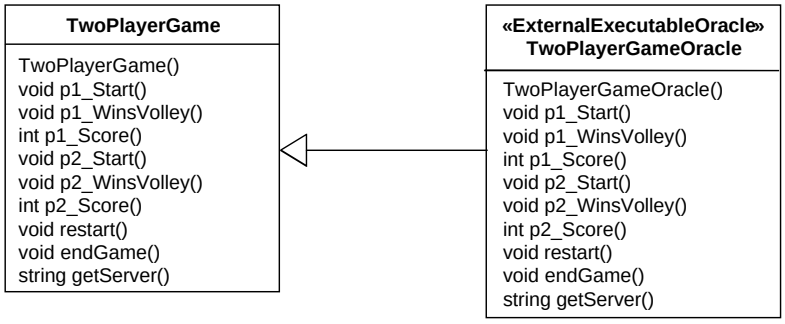


Abbildung 5.6 *Erzeugung der Orakelklasse für die Klasse* `TwoPlayerGame`

Die Vor- und Nachbedingungen der einzelnen Orakelmethoden sollen die Beziehung zwischen dem Verhalten der zu testenden Methode und dem Verhalten der Orakelmethode widerspiegeln. Kann die Orakelmethode nicht das vollständige Verhalten der zu testenden Methode simulieren, muss die Vor- bzw. die Nachbedingung entsprechend angepasst werden. Ausführbare Testfälle können nur für das Verhalten erzeugt werden, welches auch durch die Orakelmethoden abgedeckt wird.

Wird von der Klasse `TwoPlayerGame` durch Vererbung eine neue Klasse `ThreePlayerGame` erzeugt, so müssen für den Test die entsprechenden Bedingungen des Testprofils bei der Modellierung erfüllt werden. Die Klasse `ThreePlayerGame` soll das durch die Klasse `TwoPlayerGame` realisierte Ballspiel so erweitern, dass es durch drei Spieler gespielt werden kann. Dafür müssen einige neue Methoden definiert und die Methode `getServer` überschrieben werden. Dieses wird in Abbildung 5.7 dargestellt.

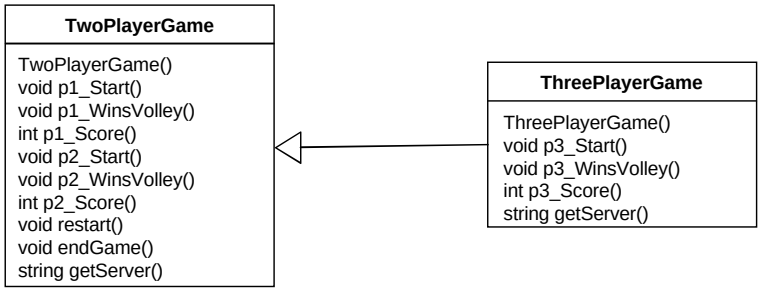


Abbildung 5.7 *Ableitung der Klasse* `ThreePlayerGame`

Bei der Modellierung des Verhaltens der Klasse `ThreePlayerGame` müssen die in Abschnitt 4.5.4 erläuterten Bedingungen eingehalten werden. Ein entsprechend modellierter Zustandsautomat wird in Abbildung 5.8 dargestellt.

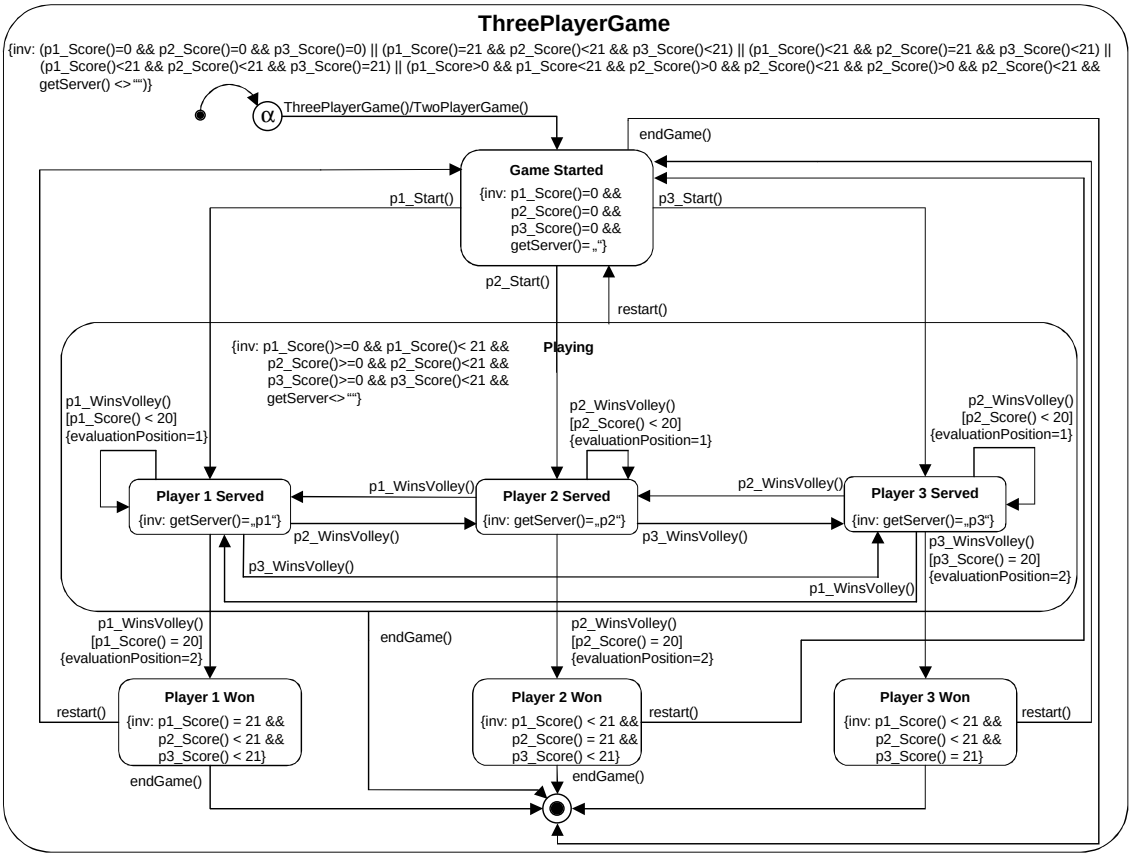


Abbildung 5.8 *Zustandsautomat der Klasse* `ThreePlayerGame`

In dem Zustandsautomaten für die Klasse `ThreePlayerGame` sind die Erweiterungen gegenüber dem Zustandsautomaten der Klasse `TwoPlayerGame` gut zu erkennen. So wurden die neuen Zustände `Player 3 Served` und `Player 3 Won` inklusive der notwenigen Zustandsübergänge eingefügt. Außerdem wurden die Zustandsinvarianten einiger Zustände entsprechend erweitert.

Für die Modellierung einer Orakelklasse, welche Sollwerte für die Klasse `ThreePlayerGame` erzeugt, muss von dieser Klasse durch Vererbung eine neue Klasse abgeleitet werden, die mit dem Stereotypen «`AbstractExternalOracle`» zu markieren ist. In dieser Orakelklasse müssen die lokal im Testobjekt definierten Methoden überschrieben werden. Für die Implementierung der Orakelmethoden für die Methoden, die durch die Klasse `ThreePlayerGame` von der Klasse `TwoPlayer-Game` geerbt werden, kann die Implementierung aus der Orakelklasse `TwoPlayerGa-meOracle` wiederverwendet werden. Die in dieser Orakelklasse lokal definierten Instanzvariablen müssen dann in der neuen Orakelklasse erneut definiert werden. Die Modellierung der Orakelklasse für die Klasse `ThreePlayerGame` ist in Abbildung 5.9 dargestellt. Die Definition von Instanzvariablen in den Orakelklassen wird hier nicht gezeigt. Die Realisierung der Wiederverwendung von Orakelmethoden und den entsprechenden Instanzvariablen aus den Orakelklassen der Oberklassen des Testobjekts kann durch ein Testwerkzeug, welches das Testprofil umsetzt, automatisch erfolgen. Die Möglichkeit der Mehrfachvererbung für die Ableitung von Orakelklassen wird nicht verwendet, da diese nicht in allen objektorientierten Programmiersprachen realisiert ist.

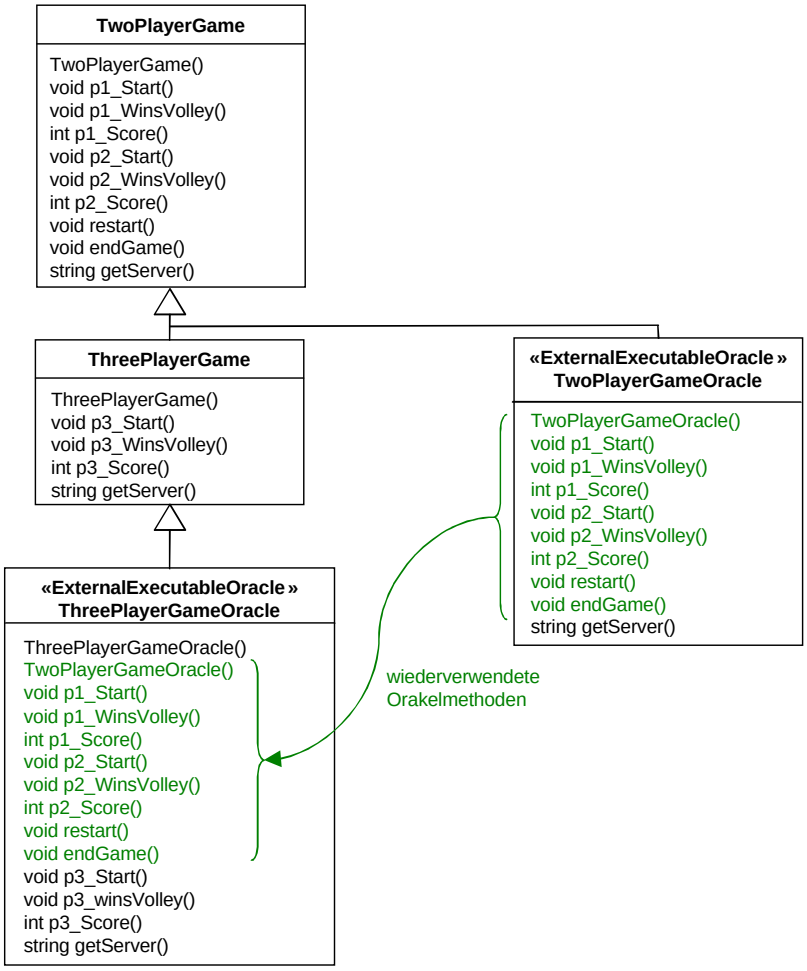


Abbildung 5.9 *Modellierung der Orakelklasse für die Klasse* `ThreePlayerGame`

Für die Wiederverwendung von Testfällen, die bereits für die Klasse `TwoPlayerGame` erzeugt worden sind, für den Test der Klasse `ThreePlayerGame` wird der Stereotyp «`InheritedClassScopeTestPattern`» auf die Klasse `ThreePlayerGame` angewendet. Dabei werden durch die Tagwerte der beiden Tagdefinitionen genau die Testfälle festgelegt, die für das Testobjekt wiederverwendet werden sollen. Dies wird in Abbildung 5.10 dargestellt.

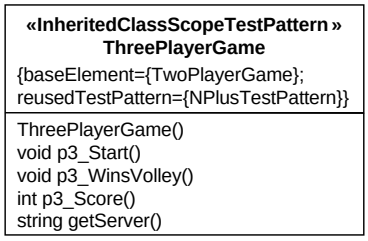


Abbildung 5.10 *Wiederverwendung von Testfällen*

In Abbildung 5.10 werden genau die Testfälle wiederverwendet, die für die Klasse `TwoPlayerGame` mit Hilfe des Testmusters `NPlusTestPattern` erzeugt wurden.

Kapitel 6

Zusammenfassung und Ausblick

In dieser Arbeit wurde ein UML-Profil zur Erstellung testbarer UML-Modelle für den objektorientierten Klassentest erstellt. Bestehende Schwächen der UML-Spezifikation auf diesem Gebiet wurden aufgezeigt und entsprechende Lösungen zur Beseitigung dieser Schwächen im Rahmen des Testprofils entwickelt. Außerdem stellt das Testprofil eine konsequente Weiterentwicklung bereits existierender Ansätze auf diesem Gebiet dar, welche die auch bei diesen Ansätzen noch vorhandenen Probleme systematisch beseitigt. Die konsequente Integration des Testprozesses in den restlichen Entwicklungsprozess, wie sie durch das Testprofil ermöglicht wird, bringt viele Vorteile gegenüber bisherigen Vorgehensweisen. So können durch die direkte Einbindung der für den Test notwendigen Informationen in ein UML-Modell, schon während des Entwurfs eines Systems, bereits in dieser Phase der Entwicklung Fehler in der Spezifikation aufgedeckt und beseitigt werden. Ein weiterer Vorteil, der sich gegenüber Ansätzen ergibt, die relativ unabhängig vom restlichen Entwicklungsprozess existieren, ist eine erhebliche Verringerung der Redundanz der zu verwaltenden Daten, da mit dem Testprofil die während des Entwurfs entwickelten Modelle lediglich derart erweitert bzw. konkretisiert werden, dass sie den Bedingungen der Testbarkeit genügen. Diese genauere Spezifikation kann sich zusätzlich positiv auf die Implementierung eines Systems auswirken, da Mehrdeutigkeiten und die daraus entstehenden Interpretationsspielräume in herkömmlichen UML-Modellen beseitigt werden. Durch die modulare Struktur des Testprofils kann eine flexible Konfiguration des Tests gewährleistet werden. Außerdem besteht die Möglichkeit, das Testprofil zu erweitern und somit an spezielle Anforderungen anzupassen.

In der jetzigen Ausbaustufe beschränkt sich das Testprofil auf den Test der öffentlichen Schnittstelle einer Klasse. Um auch die konkrete Implementierung einer Klasse testen zu können, ist eine entsprechende Erweiterung des Testprofils notwendig. Dazu wäre eine Einbeziehung der Attribute einer Klasse in die Konstruktion des Testprofils erforderlich. Auf dieser Ebene ließe sich auch ein Methodentest integrieren, da eine direkte Manipulation des Zustandes einer Instanz möglich ist und dafür somit keine Nachrichtensequenzen mehr erzeugt werden müssen. Außerdem sind Verbesserungen einzelner Komponenten des Testprofils durch die Verwendung anderer Ansätze zum Testen auf der Basis der UML denkbar. So wird in [Fraikin+00] ein Ansatz zum Testen auf der Basis von Sequenzdiagrammen vorgestellt, welcher z. B. für die Spezifikation von benutzerdefinierten Nachrichtensequenzen verwendet werden könnte.

Das entwickelte Testprofil stellt eine praktische Grundlage zur Erstellung testbarer UML-Modelle dar und ist gleichzeitig eine theoretische Basis für die Entwicklung eines Testwerkzeugs, welches die Konzepte des Testprofils umsetzt. Wenn auch einzelne Aspekte des Testprofils für den objektorientierten Softwareentwurf unabhängig vom Test

sinnvoll angewendet werden können, wie z. B. die für die Vererbung zu beachtenden Bedingungen, ist der Einsatz des gesamten Testprofils nur mit einem entsprechenden Testwerkzeug zweckmäßig, welches dessen Konzepte umsetzt und die Erzeugung ausführbarer Testfälle realisiert. Für ein solches Testwerkzeug könnten grundsätzlich zwei verschiedene Realisierungsmöglichkeiten verfolgt werden. So kann das Werkzeug darauf beruhen, dass bei der Erstellung des UML-Modells die Stereotypen genau so eingesetzt werden, wie dies beispielhaft in Kapitel 5 dargestellt wurde. Dabei wäre zu beachten, dass dann Elemente im UML-Modell spezifiziert werden müssten, die nur für den Test von Relevanz sind, wie z. B. die Stereotypen zur Definition der einzelnen Testmuster. Dagegen sind z. B. Vor- und Nachbedingungen von Methoden unabhängig vom Test ein erforderlicher Bestandteil eines UML-Modells, für die mit dem Testprofil nur zusätzlich für die Gewährleistung der Testbarkeit einzuhaltende Bedingungen angegeben werden. Die rein testrelevanten Komponenten könnten ein UML-Modell schnell unübersichtlich erscheinen lassen. Allerdings hat die Integration dieser Informationen in das UML-Modell den Vorteil, dass nur ein Werkzeug für die Modellierung benötigt wird.

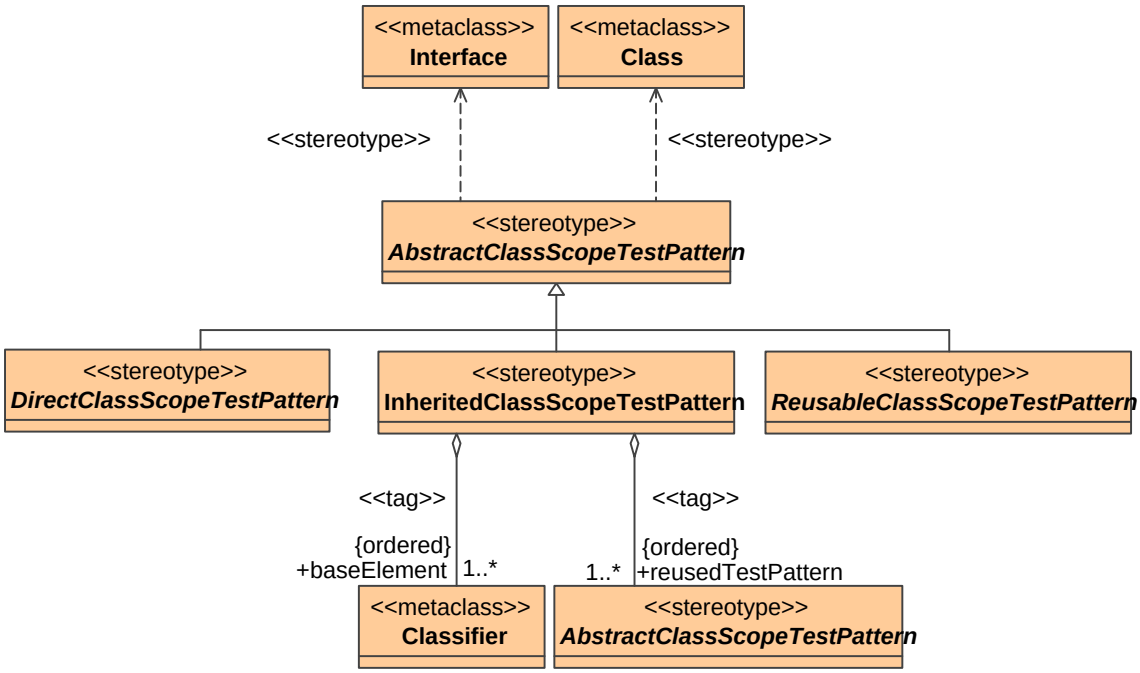
Bei dem zweiten Ansatz würden die rein testrelevanten Informationen direkt in dem Testwerkzeug verwaltet werden, welches direkt auf den Daten des Modellierungswerkzeugs aufsetzen würde. Hierdurch könnte eine komfortablere Art der Bearbeitung der entsprechenden Informationen realisiert werden. Die Definition von Stereotypen in den Modellierungswerkzeugen ist relativ umständlich und aufwendig.

Insgesamt sollte auf der Basis des erstellten Testprofils ein flexibel einsetzbares und konfigurierbares Testwerkzeug konstruiert werden können, welches die Automatisierung des Klassentests ohne die Restriktionen heutiger Testwerkzeuge ermöglicht.

Anhang A Test Profil

A.1 Test Pattern

A.1.1 Abstrakte Syntax



A.1.2 Beschreibungen und statische Definitionen

A.1.2.1 AbstractClassScopeTestPattern

Beschreibung

Die von diesem Stereotypen abgeleiteten, nicht-abstrakten Stereotypen repräsentieren Testmuster, die das Zusammenspiel der verschiedenen Methoden einer Klasse oder einer Schnittstelle überprüfen. Dabei wird nur die öffentliche Schnittstelle der Klasse / Schnittstelle getestet.

Statische Definitionen

[1] Mit Hilfe dieser Tagdefinition bzw. den aktuellen Tagwerten soll festgelegt werden, welche Methoden bzw. Operationen der entsprechenden Klasse in den Test mit einzubeziehen sind. Dadurch soll eine einfache Konfigurierbarkeit des Tests erreicht werden. Wird keine Methode bzw. Operation angegeben, wird der entsprechende Test nicht ausgeführt. Es können auch geerbte Methoden angegeben werden.

```
Alle von diesem Stereotypen abgeleiteten Stereotypen, die nicht
abstrakt sind, enthalten eine Tagdefinition mit Referenzen auf
Methoden oder Operationen. Diese Tagdefinition besitzt eine
Multiplizität von "0..*". Die mit der Tagdefinition
referenzierten Methoden sind in der mit diesem Stereotypen
markierten Klasse oder einer ihrer Oberklassen definiert.
```

[2] Für das Testobjekt, auf das das abstrakte Testmuster angewendet wird, ist genau eine ausführbare, abstrakte Klasseninvariante definiert.

```
self.constraint->select(body.
  oclIsKindOf(AbstractExecutableBooleanExpression)and
oclIsKindOf(invariant))->size = 1
```

[3] Sind für in den Test einzubeziehende Methoden Vor- oder Nachbedingungen definiert, so sind diese ausführbar.
'testingMethods' ist ein Stellvertreter für die durch den Benutzer zu definierende Tagdefinition für die in den Test einzubeziehenden Methoden.

```
self.testingMethods->forAll(constraint->
  select(oclIsKindOf(precondition)or
oclIsKindOf(postcondition))->
  forAll(body.
    oclIsKindOf(AbstractExecutableBooleanExpression)))
```

[4] Die Klasseninvariante und die Vor- und Nachbedingungen der Methoden der Klasse müssen konsistent zueinander sein.

```
Die abstrakten Vor- und Nachbedingungen aller in den Test
einzubeziehenden Methoden verletzen die abstrakte
Klasseninvariante nicht.
```

[5] Wird das Verhalten einer Klasse durch einen Zustandsautomaten spezifiziert, so müssen die Vorbedingungen der in den Test einzubeziehenden Methoden konsistent zu dem in dem Zustandsautomaten modellierten Verhalten sein.

```
self.behavior->
  select(oclIsKindOf(AbstractXFREEStateMachine))->size = 1
    implies
       'Die Summe der Vorbedingungen einer zu testenden Methode
       ist äquivalent zur Summe aller
       Zustandsübergangsvorbedingungen von legalen
       Zustandsübergängen, die durch ein mit der Methode
       verknüpftes Ereignis aktiviert werden. Eine
       Zustandsübergangsvorbedingung ist dabei das Produkt aus
       einer Zustandsinvarinate eines Zustands und der Summe
       aller Bedingungen an legalen Zustandsübergängen dieses
       Zustands, die durch ein mit der zu testenden Methode
       verknüpftes Ereignis aktiviert werden.
       Zustandsinvarianten von Zuständen, von denen keine
       Zustandsübergänge mit entsprechenden Ereignissen
       ausgehen, werden nicht beachtet.'
```

[6] Wird das Verhalten einer Klasse durch einen Zustandsautomaten spezifiziert, so müssen die Nachbedingungen der in den Test einzubeziehenden Methoden konsistent zu dem in dem Zustandsautomaten modellierten Verhalten sein.

```
self.behavior->
  select(oclIsKindOf(AbstractXFREEStateMachine))-> size = 1
    implies
       'Die Nachbedingung einer zu testenden Methode ist
       äquivalent zur Summe aus den Produkten von
       Zustandsübergangsvorbedingungen von legalen
       Zustandsübergängen, die durch ein mit der Methode
       verknüpftes Ereignis aktiviert werden, und den
       entsprechenden Zustandsinvarianten der Zielzuständer der
       Zustandsübergange. Eine Zustandsübergangsvorbedingung ist
       dabei das Produkt aus einer Zustandsinvarinate eines
       Startzustands und der Summe aller Bedingungen an legalen
       Zustandsübergängen dieses Zustands, die durch ein mit der
       zu testenden Methode verknüpftes Ereignis aktiviert
       werden.'
```

[7] Für den Test mit einem abstrakten Testmuster werden nur an der öffentlichen Schnittstelle der Klasse sichtbare Methoden einbezogen.' testingMethods' ist ein Stellvertreter für die durch den Benutzer zu definierende Tagdefinition für die in den Test einzubeziehenden Methoden.

```
testingMethods->forAll(visibilty = #public)
```

[8] Von einer zu testenden Klasse darf maximal eine Orakelklasse direkt durch Vererbung abgeleitet werden.

```
self.specialization->
  select(child.oclIsKindOf(AbstractExternalOracle))->size <= 1
```

A.1.2.2 DirectClassScopeTestPattern

Beschreibung

Von diesem Stereotypen können Stereotypen abgeleitet und konstruiert werden, die die Funktionalität einer Klasse oder einer Schnittstelle testen und dazu direkt auf das Testobjekt angewendet werden.

Statische Definitionen

[1] Mit Hilfe dieser Tagdefinition bzw. den aktuellen Tagwerten soll festgelegt werden, welche Muster zur Erzeugung der Testsequenzen verwendet werden sollen. Durch die Multiplizität von "1..*" wird erreicht, dass mehrere Muster zur Testsequenzerzeugung für ein Testmuster verwendet werden können.

```
Ein von diesem Stereotypen abgleieteter, nicht-abstrakter
Stereotyp enthält genau eine Tagdefinition mit Referenzwerten
von einem vom Stereotypen «MessageSequencePattern» abgeleiteten
Stereotypen, der ebenfalls nicht abstrakt ist. Diese
Tagdefinition besitzt eine Multiplizität im Bereich von "1..*".
```

[2] Mit Hilfe dieser Tagdefinition bzw. den aktuellen Tagwerten soll festgelegt werden, welche Muster zur Testdatenerzeugung verwendet werden sollen. Durch die Multiplizität von "1..*" wird erreicht, dass mehrere Muster zur Testdatenerzeugung für ein Testmuster verwendet werden können.

```
Ein von diesem Stereotypen abgleieteter, nicht-abstrakter
Stereotyp enthält genau eine Tagdefinition von einem vom
Stereotypen «TestDataPattern» abgeleiteten Stereotypen, der
ebenfalls nicht abstrakt ist. Diese Tagdefinition besitzt eine
Multiplizität im Bereich von "1..*".
```

[3] Mit Hilfe der Tagwerte werden die globalen Auswertungsmuster für dieses Testmuster festgelegt. Durch die Multiplizität von "0..*" wird erreicht, dass die Anwendung von globalen Auswertungsmustern optional ist.

```
Ein von diesem Stereotypen abgleieteter, nicht-abstrakter
Stereotyp enthält genau eine Tagdefinition mit Referenzwerten
von einem vom Stereotypen «GlobalEvaluationPattern»
abgeleiteten Stereotypen, der ebenfalls nicht abstrakt ist.
Diese Tagdefinition besitzt eine Multiplizität im Bereich von
"0..*".
```

[4] Mit Hilfe der Tagwerte werden die Elemente festgelegt, die für dieses Testmuster ausgewertet werden sollen. Außerdem werden jedem dieser Parameter die entsprechenden Auswertungsmuster zugeordnet. Durch die Multiplizität von "0..*" wird erreicht, dass die Anwendung von Auswertungsmustern auf einzelne Parameter optional ist.

```
Ein von diesem Stereotypen abgleieteter, nicht-abstrakter
Stereotyp enthält genau eine Tagdefinition mit Referenzwerten
auf Elemente, die mit einem vom Stereotypen
«AbstractElementEvaluationPatternSet» abgeleiteten Stereotypen
markiert sind, der ebenfalls nicht abstrakt ist. Diese
Tagdefinition besitzt eine Multiplizität im Bereich von "0..*".
```

[5] Sind in der Menge der angegebenen Auswertungsmuster auch mit dem Stereotypen «ExternalOracleEvaluationPattern» markierte Elemente enthalten, dann muss von der getesteten Klasse durch Ver-

erbung direkt eine Klasse abgeleitet werden, die mit dem Stereotypen «AbstractExternalOracle» markiert ist.

```
self.globalEvaluationPattern->union(
  self.getAllElementEvaluationPattern())->
    exists(oclIsKindOf(ExternalOracleEvaluationPattern) implies
      self.specialization->
        select(child.oclIsKindOf(AbstractExternalOracle))->
          size = 1
```

Zusätzliche Operationen

[1] Diese Operation liefert die Menge aller elementspezifischen Auswertungsmuster, die für dieses Testmuster angewendet werden sollen.'`self.elementEvaluationPatternSet`' ist ein Stellvertreter für die durch den Benutzer zu definierende Tagdefinition für die elementspezifischen Auswertungsmuster.

```
getAllElementEvaluationPattern() :
Collection(ElementEvaluationPattern)
getAllElementEvaluationPattern() =
self.elementEvaluationPatternSet->iterate(
    p : AbstractElementEvaluationPatternSet;
    result = Collection{} |
    result->union(p.elementEvaluationPattern))
```

A.1.2.3 ReusableClassScopeTestPattern

Beschreibung

Von diesem Stereotypen können Stereotypen abgeleitet und konstruiert werden, die die Funktionalität einer Klasse oder einer Schnittstelle testen und die vor ihrer Anwendung auf ein Testobjekt weitgehend fertig spezifiziert und damit wiederverwendbar sind.

Statische Definitionen

[1] Hiermit werden die Muster bestimmt, mit deren Hilfe die Methodensequenzen für die einzelnen Testfälle erzeugt werden.

```
Alle von diesem Stereotypen abgeleiteten Stereotypen, die nicht
abstrakt sind, müssen mit mindestens einem vom Stereotypen
«MessageSequencePattern» abgeleiteten Stereotypen markiert sein
und keiner dieser Stereotypen darf abstrakt sein.
```

[2] Hiermit werden die Muster bestimmt, mit deren Hilfe die Testdaten für dieses Testmuster erzeugt werden.

```
Alle von diesem Stereotypen abgeleiteten Stereotypen, die nicht
abstrakt sind, müssen mit mindestens einem vom Stereotypen
«TestDataPattern» abgeleiteten Stereotypen markiert sein und
keiner dieser Stereotypen darf abstrakt sein.
```

[3] Durch diese Stereotypen werden die für das Testmuster anzuwendenden globalen Auswertungsmuster festgelegt.

```
Sollen für ein Testmuster globale Auswertungsmuster angewendet
werden, so muss der abgeleitete Testmuster-Stereotyp mit den
entsprechenden vom Stereotypen «GlobalEvaluationPattern»
abgeleiteten nicht-abstrakten Stereotypen markiert werden.
```

[4] Durch diese Stereotypen werden die für dieses Testmuster auf einzelne Elemente anzuwendenden Auswertungsmuster festgelegt.

```
Sollen für ein Testmuster Auswertungsmuster auf einzelne
Elemente angewendet werden, so muss der abgeleitete
Testmusterstereotyp mit den entsprechenden vom Stereotypen
```

«AbstractElementEvaluationPatternSet» abgeleiteten nicht-abstrakten Stereotypen markiert werden.

[5] Für die mit dem Stereotypen «ExternalOracleEvaluationPattern» markierten Auswertungsmuster ist ein externes Orakel zur Erzeugung der Sollwerte notwendig.

```
Sind in der Menge der angegebenen Auswertungsmuster (sowohl
globale als auch elementspezifische Muster) auch mit dem
Stereotypen «ExternalOracleEvaluationPattern» markierte
Elemente enthalten, dann muss von der getesteten Klasse durch
Vererbung direkt genau eine Klasse abgeleitet werden, die mit
dem Stereotypen «AbstractExternalOracle» markiert ist.
```

A.1.2.4 InheritedClassScopeTestPattern

Beschreibung

Mit diesem Testmuster können Testmuster und deren Testfälle von Klassen wiederverwendet werden, von denen die zu testende Klasse direkt oder indirekt erbt.

Tagdefinitionen

baseElement: Klassen bzw. Schnittstellen, die mit Testmusterstereotypen markiert sind, die bzw. deren Testfälle für das aktuelle Testobjekt wiederverwendet werden sollen.

reusedTestPattern: Testmusterstereotypen, die in Zusammenhang mit einem konkreten Element, auf welches sie angewendet wurden (`baseElement`), für das aktuelle Testobjekt wiederverwendet werden sollen.

Statische Definitionen

[1] Das mit diesem Stereotypen markierte Element ist ein Untertyp von allen Elementen, von denen es Testmuster bzw. Testfälle erben soll.

```
baseElement->forAll(e | self.oclIsKindOf(e) and e <> self)
```

[2] In beiden Sequenzen muss die gleiche Anzahl von Elementen enthalten sein. Das ist die Mindestvoraussetzung dafür, dass jedes wiederverwendete Testmuster dem entsprechenden Element zugeordnet ist.

```
baseElement->size = reusedTestPattern.size
```

[3] Alle für die Wiederverwendung angegebenen Testmuster müssen für das zugehörige Element (`baseElement`, Zuordnung über jeweilige Position in den Sequenzen) als Stereotyp verwendet sein. Damit wird sichergestellt, dass die Testmuster eindeutig in Zusammenhang mit einer Klasse / Schnittstelle identifiziert werden können.
Da die für die Wiederverwendung bestimmten Testmuster immer klassenspezifische Testmuster sind, stellt diese Einschränkung gleichzeitig sicher, dass die als Basiselemente (`baseElement`) angegebenen Elemente nur Instanzen der Metaklasse `Class` oder `Interface` sind.

```
allReusedTestPatternMatch(1)
```

Zusätzliche Operationen

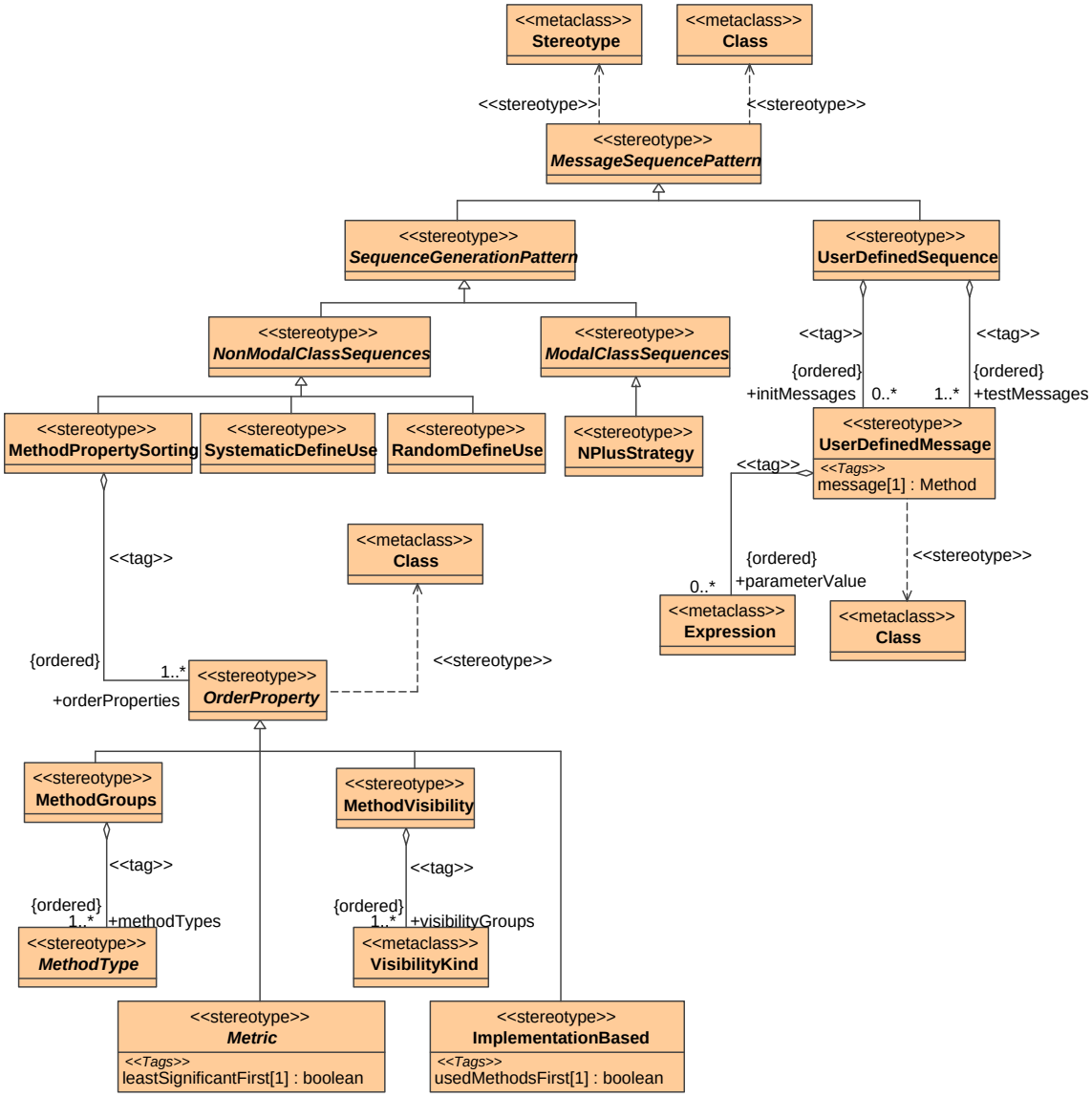
[1] Diese Operation ermittelt, ob alle zur Wiederverwendung angegebenen Testmuster (`reusedTestPattern`) als Stereotyp mit dem als zugehörig angegebenen Element (`baseElement`) verknüpft sind.

```
allReusedTestPatternMatch(i : int) : Boolean
result = if (reusedTestPattern->at(i).extendedElement <>
           baseElement->at(i)) then
           false
         else if (i < reusedTestPattern->size) then
           allReusedPatternMatch(i + 1)
         else
```

```
    true
endif
```

A.2 Message Sequence Pattern

A.2.1 Abstrakte Syntax



A.2.2 Beschreibungen und statische Definitionen

A.2.2.1 MessageSequencePattern

Beschreibung

Muster zur Erzeugung von Sequenzen von Nachrichten.

A.2.2.2 SequenceGenerationPattern

Beschreibung

Muster zur automatischen Erzeugung von Nachrichtensequenzen.

A.2.2.3 NonModalClassSequences

Beschreibung

Erzeugung von Nachrichtensequenzen zum Test nicht-modaler Klassen.

Statische Definitionen

[1] Die von diesem Stereotypen abgeleiteten Stereotypen beruhen darauf, dass das für die zu testende Klasse hinsichtlich erlaubter Nachrichtensequenzen keine Einschränkungen existieren.

```
Für die zu testende Klasse ist kein Zustandsdiagramm definiert.
```

A.2.2.4 MethodPropertySorting

Beschreibung

Erzeugung einer Sequenz von Nachrichten, bei der die Reihenfolge der Nachrichten durch eine Einteilung der zugehörigen Methoden in verschiedene Gruppen und einer Sortierung dieser Gruppen bestimmt wird.

Tagdefinitionen

orderProperties: Sequenz von Kriterien, nach denen Methoden mehrstufig sortiert werden können.

Statische Definitionen

[1] Jedes Sortierkriterium darf nur einmal verwendet werden.

```
orderProperties->forAll(op1, op2 | op1 <> op2 implies
  op1.stereotype.name <> op2.stereotype.name)
```

A.2.2.5 SystematicDefineUse

Beschreibung

Es wird jeweils eine Nachricht an eine zustandsverändernde Methoden gesendet, gefolgt von Nachrichten an alle nicht-zustandsverändernde Methoden.

A.2.2.6 RandomDefineUse

Beschreibung

Es werden Nachrichtensequenzen erzeugt, die jeweils aus einer zufälligen Sequenz von Nachrichten an alle zustandsverändernden Methoden, gefolgt von einer zufälligen Sequenz von Nachrichten an alle nicht-zustandsverändernden Methoden besteht.

A.2.2.7 ModalClassSequences

Beschreibung

Muster zur Erzeugung von Nachrichtensequenzen auf der Basis von Zustandsautomaten.

Statische Definitionen

[1] Für die zu testende Klasse muss ein testbarer Zustandsautomat definiert sein, der als Basis für die Erzeugung der Nachrichtensequenzen verwendet werden kann.

```
Für die zu testende Klasse ist genau ein Zustandsautomat
definiert, der mit einem vom Stereotypen «XFREEStateMachine»
abgeleiteten Stereotypen markiert ist.
```

A.2.2.8 NPlusStrategy

Beschreibung

Strategie zur Erzeugung von Nachrichtensequenzen auf der Basis eines Zustandsautomaten. Beschrieben wird die Strategie in [Binder99].

A.2.2.9 UserDefinedSequence

Beschreibung

Benutzerdefinierte Methodensequenz, mit der spezielle Situationen überprüft werden sollen.

Tagdefinitionen

initMessages: Nachrichten, die den Startzustand des Testobjekts für den Test erzeugen.

testMessages: Nachrichten, die den eigentlichen Test durchführen.

A.2.2.10 UserDefinedMessage

Beschreibung

Nachricht als Teil der benutzerdefinierten Nachrichtensequenz mit zugeordneten Testdaten.

Tagdefinitionen

message: Zu erzeugende Nachricht.

parameterValue: Ausdrücke, die als aktuelle Parameter übergeben werden.

Statische Definitionen

[1] Für jeden Eingangs- und Durchgangsparameter wird ein Ausdruck (`parameterValue`) angegeben, der als aktuelles Argument an den entsprechenden Parameter der Methode übergeben wird.

```
self.message.parameter->select(kind = #in or kind = #inout)->
  size = self.parameterValue->size
```

A.2.2.11 OrderProperty

Beschreibung

Kriterien nach denen verschiedene Methoden sortiert werden können.

A.2.2.12 MethodGroups

Beschreibung

Die Sortierung erfolgt stufenweise nach verschiedenen Typen von Methoden, die mit Hilfe von vom Stereotypen «`MethodType`» abgeleiteten Stereotypen für jede Methode festgelegt werden.

Tagdefinitionen

methodTypes: Sortierte Menge von Kriterien, nach denen die Methoden sortiert werden sollen.

Statische Definitionen

[1] Jeder Methodentyp darf nur einmal auftreten.

```
methodTypes->forAll(opt1, opt2 | opt1 <> opt2 implies
  opt1.name <> opt2.name)
```

A.2.2.13 Metric

Beschreibung

Bestimmung der Reihenfolge der Methoden nach einer bestimmten Metrik.

Tagdefinitionen

leastSignificantFirst: Die Methoden sollen entsprechend der Werte der Metrik in aufsteigender Reihenfolge sortiert werden.

A.2.2.14 MethodVisibility

Beschreibung

Sortierung von Methoden entsprechend ihres Gültigkeitsbereiches (`public`, `protected`, `private`).

Tagdefinitionen

visibilityGroups: Sortierte Reihenfolge von Sichtbarkeiten, nach denen die Methoden sortiert werden sollen.

A.2.2.15 ImplementationBased

Beschreibung

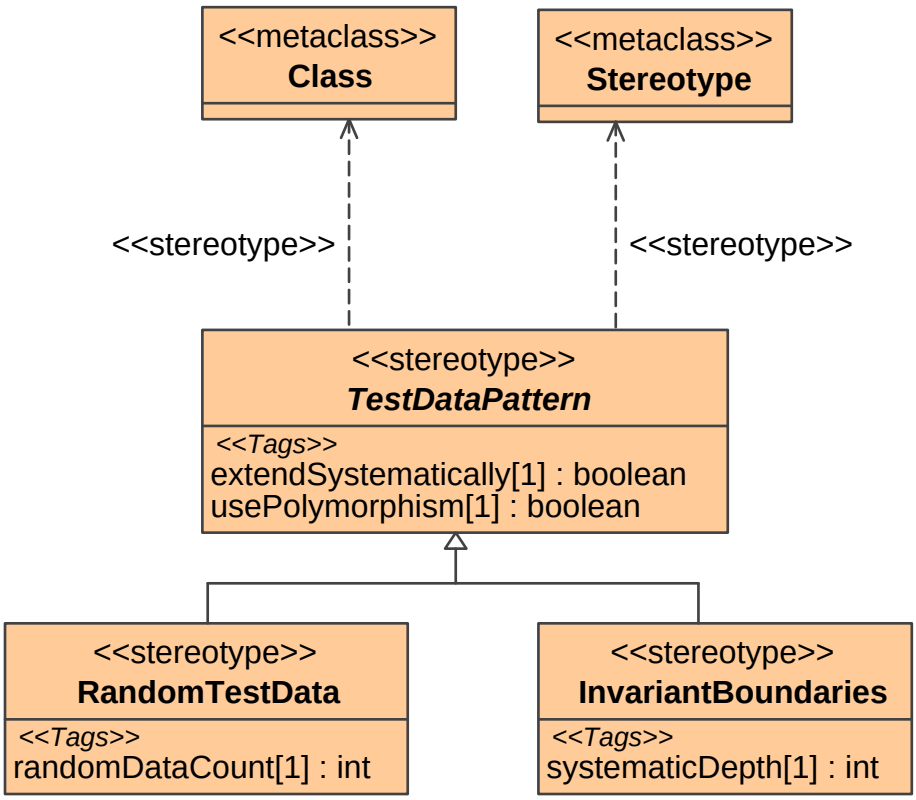
Sortierung der Methoden entsprechend der Benutzungshierarchie der Methoden untereinander.

Tagdefinitionen

usedMethodsFirst: Die Methoden, die von anderen Methoden verwendet werden, werden vor den sie verwendenden Methoden einsortiert.

A.3 Test Data Pattern

A.3.1 Abstrakte Syntax



A.3.2 Beschreibungen und statische Definitionen

A.3.2.1 TestDataPattern

Beschreibung

Diese Muster ermitteln für einen gegebenen Datenraum eine Menge von Datensätzen. Der Datenraum wird durch die mit dem Testmuster verknüpften Constraints definiert.

Tagdefinitionen

usePolymorphism: Hiermit wird spezifiziert, ob bei der Testdatenerzeugung für einzelne Elemente auch Objekte von Untertypen des Elements erzeugt werden sollen.

extendSystematically: Hiermit wird festgelegt, ob für jedes polymorphe Element systematisch oder zufällig Objekte der Unterklassen erzeugt werden sollen. Die systematische Erzeugung erhöht die Anzahl der generierten Testfälle.

A.3.2.2 RandomTestData

Beschreibung

Für einen gegebenen Datenraum wird zufällig eine festgelegte Anzahl von Datensätzen erzeugt.

Tagdefinitionen

RandomDataCount: Anzahl der zu erzeugenden Datensätze.

A.3.2.3 InvariantBoundaries

Beschreibung

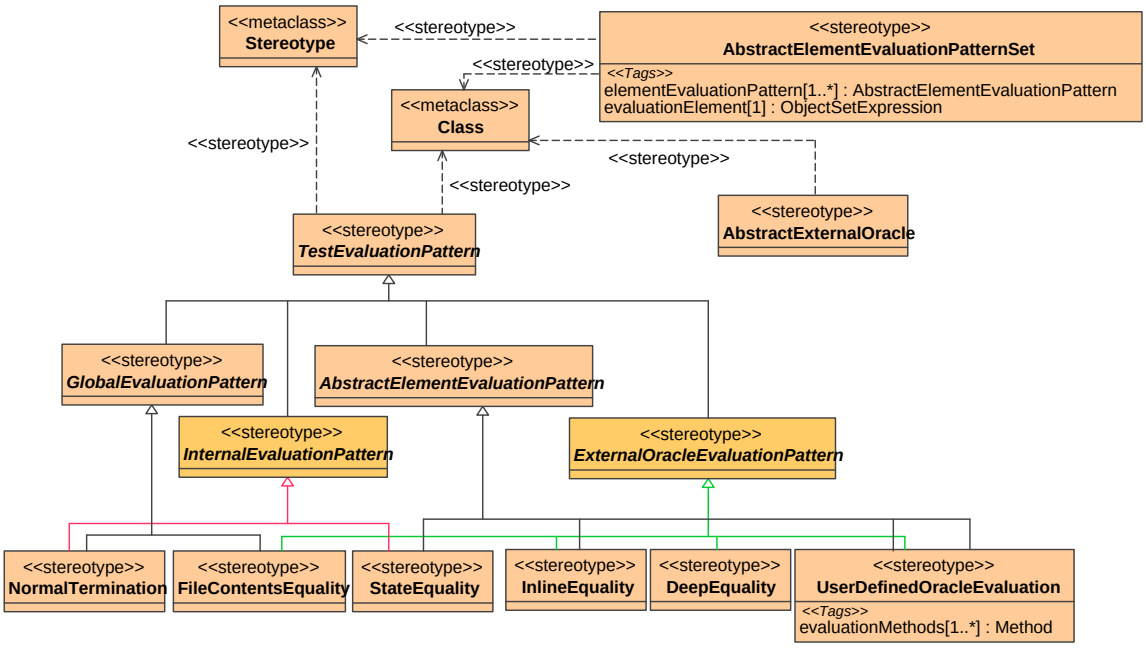
Muster, welches die Testdaten systematisch auf der Basis der Definition von Datenräumen, und insbesondere deren Grenzen, ermittelt.

Tagdefinitionen

systematicDepth: Für komplexe Datentypen muss festgelegt werden, bis zu welcher Tiefe das Testdatenmuster anzuwenden systematisch anzuwenden ist. Für Elemente die tiefer verschachtelt sind werden die Testdaten zufällig ermittelt.

A.4 Test Evaluation Pattern

A.4.1 Abstrakte Syntax



A.4.2 Beschreibungen und statische Definitionen

A.4.2.1 TestEvaluationPattern

Beschreibung

Stereotyp zur Definition verschiedener Muster zur Auswertung von Testfällen.

A.4.2.2 GlobalEvaluationPattern

Beschreibung

Muster zur Testauswertung, welches globale Kriterien untersucht.

A.4.2.3 AbstractElementEvaluationPattern

Beschreibung

Muster zur Testauswertung für ein einzelnes Element. Die Sollwerte für dieses Muster sind Instanzen des Datentypen des Modellelements, dem dieses Muster zugeordnet wurde.

A.4.2.4 InternalEvaluationPattern

Beschreibung

Für die von diesem Stereotypen abgeleiteten Auswertungsmechanismen wird kein externes Orakel benötigt. Die Sollwerte können direkt aus dem UML-Modell ermittelt werden oder das Auswertungsmuster enthält selbst den Sollwert.

A.4.2.5 ExternalOracleEvaluationPattern

Beschreibung

Für von diesem Stereotypen abgeleitete Auswertungsmechanismen ist ein externes Orakel notwendig, um die entsprechenden Sollwerte ermitteln zu können. Repräsentiert wird ein Orakel durch eine mit dem Stereotypen «ExternalAbstractOracle» markierte Klasse, die direkt durch Vererbung von der zu testenden Klasse abgeleitet wird.

Statische Definitionen

[1] Für die zu testende Klasse muss genau ein externes Orakel definiert sein. '`testingClass`' ist ein Stellvertreter für die zu testende Klasse.

```
testingClass.specialization->
  select(child.oclIsKindOf(AbstractExternalOracle))->size = 1
```

A.4.2.6 NormalTermination

Beschreibung

Überprüft, ob ein Testfall "normal" terminiert. Das heißt, es darf kein Programmabsturz auftreten. Ausnahmen dagegen sind erlaubt, wenn sie nicht zu einem Absturz führen, sie also aufgefangen und behandelt werden.

A.4.2.7 FileContentsEquality

Beschreibung

Mit diesem Auswertungsmuster kann der Inhalt von zwei Dateien miteinander verglichen werden.

A.4.2.8 StateEquality

Beschreibung

Basis des Vergleichs ist ein definierter Zustandsautomat.

Statische Definitionen

[1] Um auch den Zustand für verschachtelte Auswertungselemente ermitteln zu können, muss für die Typen aller auf den Pfaden liegenden Elemente ein Zustandsautomat vom Typ «AbstractXFREEStateMachine» definiert sein.

```
Für die Typen aller der in dem Ausdruck zur Ermittlung der
auszuwertenden Elemente verwendeten Elemente muss ein
Zustandsautomat vom Typ «AbstractXFREEStateMachine» definiert
sein.
```

A.4.2.9 InlineEquality

Beschreibung

Ein Element für dessen Typ ein Inline-Operator "=" definiert ist. Dies gilt z. B. für die vordefinierten Basistypen int, long, char, etc..

Statische Definitionen

[1] Auf dem Typen, des mit diesem Stereotypen markierten Elements, muss ein Inline-Operator mit dem Namen "=" definiert sein. Die Definition der Semantik dieses Operators ist nicht möglich, da hier keine Nachbedingung für diese Methode definiert werden kann. Ebenso kann keine OCL-Definition angegeben werden, dass der Operator in der verwendeten Programmiersprache als Inline-Operator angewendet werden kann. Die getrennten Ausdrücke für "Attribute"- und "Parameter"-Elemente ist notwendig, da die Funktion `oclAsType` nur auf einen Typ angewendet werden kann.

```
if self.oclIsKindOf(Attribute) then
  self.oclAsType(Attribute).type.allMethods->exists(m |
    m.name = "=" and
    m.isQuery = true and
```

```
        m.parameter->select(kind = return and type = Boolean)->
          size() = 1 and
        m.ownerScope = #classifier and
        m.parameter->select(kind = in)->size() = 2 and
        m.parameter->select(kind = in)->forAll(type =
          self.oclAsType(Attribute).type))
    else
      self.oclAsType(Parameter).type.allMethods->exists(m |
        m.name = "=" and
        m.isQuery = true and
                m.parameter->select(kind = return and type =
                  Boolean)->size() = 1 and
                m.ownerScope = #classifier and
                m.parameter->select(kind = in)->size() = 2 and
                m.parameter->select(kind = in)->forAll(type =
                  self.oclAsType(Parameter).type))
    endif
```

A.4.2.10 DeepEquality

Beschreibung

Für den Vergleich eines mit diesem Stereotypen markierten Elements wird eine Funktion verwendet, die von einem entsprechenden Testsystem erzeugt werden muss. Diese ermittelt die sogenannte "Deep Equality". Diese ist wie folgt definiert:

```
deepEquality(elem : Attribute) : Boolean
  if elem.type.oclIsKindOf(DataType) then
    result = (elem = self)
  else
    result = self.allAttributes->iterate(attr : Attribute;
                                         result : Boolean = true |
                                         result and
      attr.deepEquality(elem.allAttributes->select(name=attr.name)))
  endif
```

Für Elemente ohne Struktur (`DataType`) wird der "="-Operator zum Vergleich verwendet. Bei Elementen mit eigener Struktur wird für alle enthaltenen Attribute rekursiv die Funktion `deepEquality` aufgerufen.

A.4.2.11 UserDefinedOracleEvaluation

Beschreibung

Auswertung eines Testfalls durch eine benutzerdefinierte Methode.

Tagdefinitionen

evaluationMethods: Benutzerdefinierte Methoden mit deren Hilfe das Ergebnis eines Testfalls überprüft werden kann.

Statische Definitionen

[1] Die Auswertungsmethoden dürfen keine Seiteneffekte besitzen.

```
evaluationMethods->forAll(isQuery)
```

[2] Jede Auswertungsmethode besitzt genau einen Rückgabewert vom Typ `boolean`. Dieser gibt an, ob der Test erfolgreich war (`true`, kein Fehler gefunden) oder nicht (`false`, Fehler gefunden).

```
evaluationMethods->forAll(parameter->select(kind = #return and
type.oclIsTypeOf(boolean))->size() = 1)
```

[3] Jede Auswertungsmethode wird zur Überprüfung verschiedener Eigenschaften verwendet und erhält dafür als ersten aktuellen Parameter den Wert des verknüpften Elements. Dem zweiten formalen Eingangsparameter wird das vom Orakel gelieferte Objekt (Sollwert) übergeben.

```
evaluationMethods->forAll(parameter->select(kind = #in)->
  size = 2 and
parameter->forAll(oclIsTypeOf(owner.evaluationElement.type)))
```

A.4.2.12 AbstractElementEvaluationPatternSet

Beschreibung

Zuordnung von mehreren Auswertungsmustern zu einem Modellelement, welches mit diesen Auswertungsmustern überprüft werden soll.

Tagdefinitionen

elementEvaluationPattern: Menge der auf das mit diesem Stereotypen verknüpfte Element anzuwendenden Auswertungsmuster.

evaluationElement: Ein Ausdruck, der bei seiner Auswertung ein Modelelement ergibt, welches mit den zugeordneten Auswertungsmustern überprüft werden soll.

Statische Definitionen

[1] Es kann eine Instanz der zu testenden Klasse bzw. Schnittstelle (bzw. einer minimalen davon abgeleiteten Klasse) oder ein Parameter überprüft werden.

```
self.evaluationElement.oclIsKindOf(Class) or
self.evaluationElement.oclIsKindOf(Interface) or
self.evaluationElement.oclIsKindOf(Parameter)
```

[2] Ist das auszuwertende Element ein Parameter, dann muss es ein Rückgabe-, Durchgangs- oder Ausgangsparameter einer öffentlichen Methode sein.

```
self.evaluationElement.oclIsKindOf(Parameter) implies
  self.evaluationElement.oclAsType(Parameter).kind <> #in and
  self.evaluationElement.oclAsType(Parameter).
    behavioralFeature.visibility = #public
```

[3] Es können nur über die öffentliche Schnittstelle zugängliche Elemente ausgewertet werden.

```
Zur Ermittlung des auszuwertenden Elements werden in der
Tagdefinition ‘evaluationElement’ nur über die öffentliche
Schnittstelle zugängliche Elemente verwendet.
```

A.4.2.13 AbstractExternalOracle

Beschreibung

Klasse, die als Wrapper zu externen Orakeln fungiert, und durch die Sollwerte generiert werden können.

Statische Definitionen

[1] Es werden alle lokalen, öffentlichen Methoden der Oberklasse überschrieben.

```
self.generalization.parent.feature->
  select(oclIsKindOf(BehavioralFeature) and
visibility = #public)->forAll(f |
  self.feature.select(oclIsKindOf(BehavioralFeature))->
    includes(f))
```

[2] Eine Orakelklasse kann direkt nur von einer anderen Klasse erben.

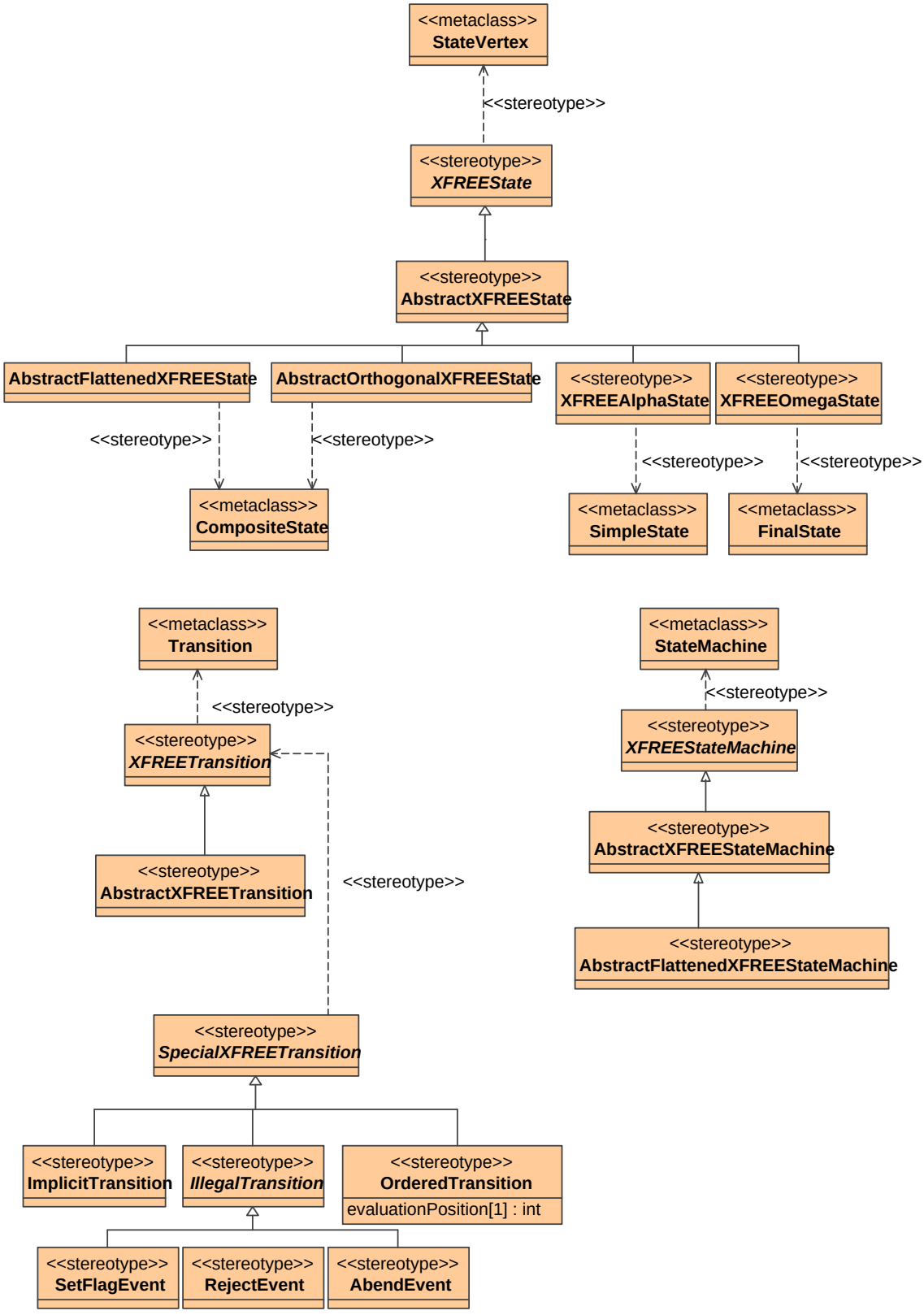
```
self.generalization->size = 1
```

[3] Die Orakelklasse soll möglichst das Verhalten der Oberklasse simulieren. Im Idealfall sind die Vor- und Nachbedingungen der jeweiligen Methoden in der Orakelklasse und der zu testenden Klasse identisch sein. Kann ein Orakel nicht die volle Funktionalität einer Methode simulieren, muss dies durch die Vor- und Nachbedingungen definiert sein. In einem solchen Fall sind die Vor- und Nachbedingungen einer Methode der Orakelklasse stärker als die der entsprechenden Methode zu testenden Klasse. Dann können ausführbare Testfälle nur entsprechend den Vor- und Nachbedingungen der Orakelklasse generiert werden. Für die nicht durch das Orakel abgedeckten Bereiche können zwar Testfälle generiert werden, eine automatische Testauswertung ist hier allerdings nicht möglich.

```
Die Vor- und Nachbedingungen aller überschriebenen Methoden ist
äquivalent oder stärker zu den Vor- und Nachbedingungen der
entsprechenden Methoden der Oberklasse.
```

A.5 XFREE State Model

A.5.1 Abstrakte Syntax



A.5.2 Beschreibungen und statische Definitionen

A.5.2.1 XFREEStateMachine

Beschreibung

Ein solcher Zustandsautomat genügt der Definition des *XFREE State Model* und ist somit testbar.

Statische Definitionen

[1] Das mit dem Zustandsautomaten spezifizierte Verhalten deckt den gesamten für die zugehörige Klasse definierten Zustandsraum ab.

```
self.top.constraint->
  select(oclIsKindOf(ExecutableBooleanExpression) and
  oclIsKindOf(stateInvariant)) = self.context.constraint->
    select(oclIsKindOf(ExecutableBooleanExpression) and
    oclIsKindOf(invariant))
```

[2] Besitzt die mit dem Automaten verbundene Klasse mindestens einen Destruktor, so ist genau ein Omegazustand (und kein anderer FinalState) innerhalb des Wurzelzustandes definiert. Dadurch, dass die Klasse einen Destruktor enthält, sind die Objekte der Klasse selbst für ihre Zerstörung (Übergang in den Omegazustand) verantwortlich. Deshalb muss der Omegazustand im Wurzelzustand modelliert werden.

```
self.context.oclAsType(Classifier).feature->
  select(oclIsKindOf(destroy)->size > 0 implies
    self.top.oclAsType(CompositeState).subvertex->
      select(oclIsKindOf(XFREEOmegaState))->size = 1 and
      self.top.oclAsType(CompositeState).subvertex->
        select(oclIsKindOf(FinalState))->size = 1
```

[3] Besitzt die mit dem Automaten verbundene Klasse keinen Destruktor, so ist im Wurzelzustand kein Endzustand (FinalState) definiert. Dadurch, dass die Klasse keinen Destruktor enthält, sind die Objekte der Klasse nicht selbst für ihre Zerstörung (Übergang in den Omegazustand) verantwortlich. Deshalb kann der Omegazustand im Wurzelzustand nicht modelliert werden.

```
self.context.oclAsType(Classifier).feature->
  select(oclIsKindOf(destroy)->size = 0 implies
    self.top.oclAsType(CompositeState).subvertex->
      select(oclIsKindOf(FinalState))->size = 0
```

[4] Es sollen alle definierten Ereignisse und Aktionen modelliert werden.

```
Jedes von der mit dem Zustandsautomaten verknüpften Klasse zu
verarbeitende Ereignis und jede für diese Klasse definierte
Aktion erscheint an mindestens einer Transition.
```

[5] Der Wurzelzustand enthält genau einen Alpha-Zustand.

```
self.top.oclAsType(CompositeState).subvertex->
  select(oclIsKindOf(XFREEAlphaState))->size = 1
```

[6] Es soll keine Zustände geben, die nicht erreicht werden können. Diese werden während der Existenz eines Objekts niemals aktiviert und sind somit nicht testbar.

```
Ausgehend vom im Wurzelzustand enthaltenen Alpha-Zustand sind
alle im Zustandsautomaten enthaltenen Zustände erreichbar.
```

A.5.2.2 AbstractXFREEStateMachine

Beschreibung

Ein solcher Zustandsautomat beschreibt das Verhalten von Instanzen einer Klasse, wie es über deren öffentliche Schnittstelle beobachtet werden kann.

[1] Der Wurzelzustand ist ein abstrakter Zustand. Durch die für den Stereotypen «AbstractXFREEState» definierten Einschränkungen wird sichergestellt, dass ein solcher Automat nur mit Hilfe von abstrakten Zuständen definiert wird.

```
self.top.oclIsKindOf(AbstractXFREEState)
```

[2] Durch den Zustandsautomaten wird der gesamte an der öffentlichen Schnittstelle der entsprechenden Klasse sichtbare Zustandsraum für die entsprechende Klasse abgedeckt.

```
self.top.constraint->
  select(oclIsKindOf(AbstractExecutableBooleanExpression) and
  oclIsKindOf(stateInvariant)) = self.context.constraint->
    select(oclIsKindOf(AbstractExecutableBooleanExpression) and
    oclIsKindOf(invariant))
```

[3] Es sollen alle definierten öffentlichen Ereignisse und Aktionen modelliert werden, damit sie getestet werden können.

```
Jedes von der mit dem Zustandsautomaten verknüpften Klasse zu
verarbeitende öffentliche Ereignis und jede für diese Klasse
definierte öffentliche Aktion erscheint an mindestens einem
Zustandsübergang.
```

A.5.2.3 AbstractFlattenedXFREEStateMachine

Beschreibung

Zustandsautomat, der die flache Hierarchie aus lokalen und geerbten Eigenschaften darstellt (genaue Definition siehe Abschnitt 4.5.4)

Statische Definitionen

[1] Der Wurzelzustand enthält genau einen Zustand, der die flache Hierarchie der Zustandsautomaten der Vererbungshierarchie darstellt.

```
self.top.oclAsType(CompositeState).subvertex->
  select(oclIsKindOf(AbstractFlattenedXFREEState))->size = 1
```

[2] Der Zustandsautomat muss mindestens das Verhalten aller direkten und indirekten Oberklassen modellieren.

```
Der Zustandsautomat muss alle legalen Zustandsübergänge der
Zustandsautomaten aller direkten und indirekten Oberklassen
akzeptieren.
```

A.5.2.4 XFREEState

Beschreibung

Ein solcher Zustand genügt der Definition des *XFREE State Model* und ist somit eindeutig definiert und testbar.

Statische Definitionen

[1] Alle in einem Containerzustand enthaltenen Zustände sind mit einem vom Stereotypen «XFREEState» abgeleiteten Stereotypen markiert.

```
self.oclIsKindOf(CompositeState) implies
  self.oclAsType(CompositeState).subvertex->
    forAll(oclIsKindOf(XFREEState))
```

[2] Jeder Zustand hat einen Namen.

```
self.name.notEmpty
```

[3] Die erste Bedingung stellt sicher, dass immer genau ein Zustand aktiv ist und dieser eindeutig ermittelt werden kann. Durch die zweite Bedingung wird sichergestellt, dass durch die Unterzustände ei-

nes Zustandes immer der gesamte durch ihn aufgespannte Zustandsraum abgedeckt wird, es also keinen undefinierten Zustand innerhalb eines Zustandes gibt.

```
self.oclIsKindOf(CompositeState) implies
   'Die durch die Zustandsinvarianten aller direkt in diesem
   Zustand enthaltenen Zustände aufgespannten Zustandsräume sind
   paarweise disjunkt zueinander.' and
   'Die Vereinigung der durch die Zustandsinvarianten aller
   direkt in diesem Zustand enthaltenen Zustände aufgespannten
   Zustandsräume ist äquivalent zum durch die Zustandsinvariante
   dieses Zustandes aufgespannten Zustandsraum.'
```

[4] Besitzt ein zusammengesetzter Zustand mindestens eine eingehende Transition (Zielzustand ist der zusammengesetzte Zustand selbst), so enthält er genau einen Pseudozustand vom Typ "`initial`", der genau eine ausgehende Transition besitzt.

```
self.oclIsKindOf(CompositeState) and self.incoming->notEmpty
implies
  let InitialPseudoState : PseudoState =
    self.oclAsType(CompositeState).subvertex->
      select(oclIsKindOf(PseudoState))->
        select(kind = #initial) in
          InitialPseudoState->size = 1 and
          InitialPseudoState.outgoing->size = 1
```

[5] Ist der Zustand ein History-Zustand, so besitzt dieser Zustand mindestens eine eingehende Transition und genau eine ausgehende Transition.

```
self.oclIsKindOf(PseudoState) and
   (self.oclAsType(PseudoState).kind = #deepHistory or
   self.oclAsType(PseudoState).kind = #shallowHistory) implies
        self.incoming->size > 0 and self.outgoing.size = 1
```

[6] Ein Zustand `s1` (außer den initialen Pseudozuständen und dem Wurzelzustrand) muss entweder selbst eine eingehende Transition besitzen, die als Quellzustand einen Zustand `s2` besitzt, der weder mit `s1` identisch ist, noch direkt oder transitiv in `s1` enthalten ist, oder ein direkt oder transitiv in `s1` enthaltener Zustand besitzt eine eingehende Transition, deren Quellzustand weder `s1` noch ein direkt oder transitiv in `s1` enthaltener Zustand ist. Damit kann sichergestellt werden, dass s1 zumindest durch eine Transition von einem Zustand "außerhalb" von s1 erreichbar ist. Dies ist ein notwendiges, aber kein hinreichendes Kriterium für die Erreichbarkeit aller Zustände vom Alphazustand aus.

```
not ((self.oclIsKindOf(PseudoState) and
  self.oclAsType(PseudoState).kind = #initial) or
    self.StateMachine.notEmpty) implies
      self.transitionFromOutsideExists(self, self)
```

[7] Alle von einem «`XFREEState`» ausgehenden Transitionen (inklusive eventuell vorhandener innerer Transitionen) sind mit dem Stereotypen «`XFREETransition`» oder einem davon abgeleiteten Stereotypen markiert.

```
allOutTransitions->forAll(oclIsKindOf(XFREETransition))
```

[8] Die Menge der ausgehenden und internen Transitionen eines Zustandes wird in disjunkte Untermengen aufgeteilt, deren Elemente jeweils das gleiche (bzw. das leere) Ereignis als Trigger besitzen (`tSet`) und und für die folgende Bedingungen gelten:

(a) Ist in einer Menge nur ein Element enthalten, dann ist der Guard dieser Transition leer.

(b) Ist in einer Menge mehr als ein Element enthalten, dann gelten folgende Bedingungen:

(i) Ist in der Menge keine implizite Transition enthalten, dann sind alle Wahrheitswerte jedes Guards durch die Guards der anderen Transitionen in der Menge vollständig abgedeckt.

(ii) Die Menge enthält höchstens eine implizite Transition.

(iii) Alle Transitionen der Menge (außer der impliziten Transition) sind mit dem Stereotypen «`OrderedTransition`» markiert.

(iv) Die Guards von zwei unterschiedlichen Transitionen sind nicht äquivalent.

(v) Die Auswertungspositionen für die einzelnen geordneten Transitionen («`OrderedTransition`») sind eindeutig.

```
allOutTransitions->forAll(t |
 let tSet : Set(Transition) = allTransitionsWithSameEvent(t) in
    (tSet->size = 1 implies tSet->forAll(guard->isEmpty)) and
    (tSet->size > 1 implies
      (tSet->select(oclIsKindOf(ImplicitTransition))->isEmpty
        implies
          'für alle Guards der in der Menge enthaltenen
          Transitionen sind alle Wahrheitswerte, die aus einem
          der Guards resultieren, ebenfalls in der Menge der
          Guards enthalten') and
      tSet->select(oclIsKindOf(ImplicitTransition))->
        size <= 1 and
      tSet->select(not oclIsKindOf(ImplicitTransition))->
        forAll(oclIsKindOf(OrderedTransition)) and
      tSet->forAll(t1, t2 |  t1 <> t2 implies
        'Die Guards von t1 und t2 sind nicht äquivalent.' and
        (t1.oclIsKindOf(OrderedTransition) and
        t2.oclIsKindOf(OrderedTransition)) implies
          t1.oclAsType(OrderedTransition).evaluationPosition <>
            t2.oclAsType(OrderedTransition).evaluationPosition)
```

[9] Handelt es sich um einen "normalen" Zustand, wird dieser durch eine ausführbare Zustandsinvariante beschrieben.

```
self.oclIsKindOf(State) and
not self.oclIsKindOf(AlphaState) and
not self.oclIsKindOf(OmegaState) implies
  self.constraint->select(oclIsKindOf(stateInvariant) and
  body.oclIsKindOf(ExecutableBooleanExpression))->size = 1
```

Zusätzliche Operationen

[1] Diese Operation ermittelt, ob der Zustand `s1` oder mindestens einer der direkt oder transitiv enthaltenen Zustände (`StateVertex`, sofern vorhanden) eine eingehende Transition besitzen, deren Quellzustand weder `s1` selbst noch ein direkt oder transitiv in `s1` enthaltener Zustand ist. Funktionsweise: Beim initialen Aufruf dieser rekursiven Operation sind `s1` und `s2` identisch und bezeichnen den Zustand für den die Eigenschaft überprüft werden soll. `s1` verweist über die gesamte Rekursion auf den zu überprüfenden Zustand. `s2` stellt jeweils den in `s1` direkt oder transitiv enthaltenen Zustand dar, für den überprüft wird, ob er eine eingehende Transition besitzt, die ihren Ursprung "außerhalb" von `s1` hat.
Zuerst wird überprüft, ob Transitionen, die `s2` selbst als Zielzustand besitzen, die geforderte Bedingung erfüllen. Ist dies der Fall gibt die Operation "`true`" zurück. Erfüllen keine dieser Transitionen die Bedingung, wird überprüft, ob es sich bei `s2` um einen zusammengesetzten Zustand handelt. In einem solchen Fall wird die Operation für alle in `s2` enthaltenen Unterzustände rekursiv aufgerufen und die Ergebnisse durch mit einem ODER-Operator miteinander verknüpft. Ist `s2` auch kein zusammengesetzter Zustand, so gibt die Operation den Wert "`false`" zurück.

```
transitionFromOutsideExists(s1 : StateVertex, s2 : StateVertex)
  : Boolean
result = if s2.incoming->size > 0 then
           s2.incoming->select(t |
             not ancestor(s1, t.source))->size > 0
         else if s2.oclIsKindOf(CompositeState) then
           s2.subvertex->iterate(s : StateVertex;
             result : Boolean = false |
```

```
        result or transitionFromOutsideExists(s1, s))
    else
      false
    endif
```

[2] Diese Operation ermittelt, ob der Zustand `s2` direkt oder transitiv in `s1` enthalten ist. Sie wurde aus der UML-Spezifikation mit der Änderung übernommen, dass beide Parameter hier vom Typ `StateVertex` sind (in der UML-Spezifikation vom Typ `State`). Diese Änderung war für die Verwendung in der Operation `transitionFromOutsideExists` notwendig.

```
ancestor(s1 : StateVertex, s2 : StateVertex) : Boolean
result = if (s1 = s2) then
           true
         else if (s1.container->isEmpty) then
           true
         else if (s2.container->isEmpty) then
           false
         else
           ancestor(s1, s2.container)
         endif
```

[3] Ermittelt alle von einem Zustand abgehenden Transitionen eines Zustandes, inklusive aller eventuell enthaltenen inneren Tansitionen.

```
allOutTransitions : Set(Transition)
result = if (self.oclIsKindOf(State)) then
           self.oclAsType(State)->innerTransition->
             union(self.outgoing)
         else
           self.outgoing
         endif
```

[4] Diese Operation liefert die Menge aller ausgehenden Transitionen (inklusive innerer Transitionen), die das gleiche Ereignis als Trigger besitzen, wie die übergebene Transition.

```
allTransitionsWithSameEvent(likeThisTransition : Transition) :
  Set(Transition)
result = if (likeThisTransition.trigger.isEmpty) then
           allOutTransitions->select(trigger.isEmpty)
         else
           allOutTransitions->select(trigger =
             likeThisTransition.trigger)
         endif
```

A.5.2.5 AbstractXFREEState

Beschreibung

Ein Zustand, der nur mit Hilfe von Rückgabe- und Ausgangsparametern öffentlicher, zustandserhaltender Methoden der entsprechenden Klasse definiert wird.

Statische Definitionen

[1] Für jeden „normalen“ Zusta nd ist genau eine ausführbare, abstrakte Zustandsinvariante definiert.

```
self.oclIsKindOf(State) implies
  self.constraint->select(body.
    oclIsKindOf(AbstractExecutableBooleanExpression)and
  oclIsKindOf(invariant))->size = 1
```

[2] Alle von einem abstrakten Zustand ausgehenden (inklusive eventuell vorhandener innerer Transitionen) Transitionen sind mit dem Stereotypen «`AbstractXFREETransition`» oder einem davon abgeleiteten Stereotypen markiert.

```
allOutTransitions->forAll(oclIsKindOf(AbstractXFREETransition))
```

[3] Handelt es sich um einen "normalen" Zustand, wird dieser durch eine abstrakte, ausführbare Zustandsinvariante beschrieben.

```
self.oclIsKindOf(State) and
  not self.oclIsKindOf(XFREEAlphaState) and
  not self.oclIsKindOf(XFREEOmegaState) implies
    self.constraint->select(oclIsKindOf(stateInvariant) and
      body.oclIsKindOf(AbstractExecutableBooleanExpression))->
        size = 1
```

A.5.2.6 AbstractFlattendXFREEState

Beschreibung

Zustand in dem die flache Hierarchie der geerbten Eigenschaften dargestellt wird.

Statische Definitionen

[1] Alle Unterzustände dieses Zustandes sind mit dem Stereotypen «`AbstractOrthogonalXFREEState`» markiert. Das heißt, sie stellen jeweils einen orthogonalen Zustandsautomaten der Vererbungshierarchie dar.

```
self.subvertex->
  forAll(oclIsKindOf(AbstractOrthogonalXFREEState))
```

[2] Der Zustand enthält orthogonale Unterzustände.

```
self.isConcurrent
```

[3] Ein solcher Zustand kann nur im Wurzelzustand definiert werden.

```
self.container.stateMachine.notEmpty
```

A.5.2.7 AbstractOrthogonalXFREEState

Beschreibung

Ein solcher Zustand stellt einen orthogonalen Zustandsautomaten einer Vererbungshierarchie dar.

Statische Definitionen

[1] Ein solcher Zustand ist immer ein direkter Unterzustand eines mit dem Stereotypen «`AbstractFlattenedXFREEState`» markierten Zustands.

```
self.container.oclIsKindOf(AbstractFlattenedXFREEState)
```

[2] Über den Alphazustand und dessen Zustandsübergänge wird der Aufruf der Konstruktoren der Oberklassen modelliert.

```
Wird in dem Zustand das Verhalten von einer Oberklasse
modelliert, enthält dieser Zustand einen Alphazustand.
```

A.5.2.8 XFREEAlphaState

Beschreibung

Repräsentiert die Deklaration eines Objekts vor seiner Definition (Konstruktion).

Statische Definitionen

[1] Der Alphazustand besitzt genau eine eingehende Transition, die als Quellzustand einen Pseudozustand vom Type "`initial`" hat.

```
self.incoming->size = 1 and
self.incoming.source.oclIsKindOf(PseudoState) and
self.incoming.source.oclAsType(PseudoState).kind = #initial
```

[2] Der Alphazustand hat mindestens eine ausgehende Transition.

```
self.outgoing->size >= 1
```

[3] Der Alphazustand enthält keine internen Transitionen.

```
self.internalTransition->isEmpty
```

A.5.2.9 XFREEOmegaState

Beschreibung

Repräsentiert den Zustand eines Objekts nach seiner Zerstörung.

Statische Definitionen

[1] Ein Omegazustand besitzt keine internen Transitionen. In der UML-Spezifikation sind für Objekte der Metaklasse "`FinalState`" interne Transitionen zugelassen.

```
self.internalTransition->isEmpty
```

A.5.2.10 XFREETransition

Beschreibung

Diese Transitionen und die an diesen Transitionen annotierten Elemente genügen der Definition des *XFREE State Model*.

Statische Definitionen

[1] Die Klasse, deren Verhalten durch den Zustandsautomaten beschrieben wird, muss das Signal verarbeiten können.

```
self.trigger.notEmpty implies
  self.trigger.oclIsKindOf(SignalEvent) implies
    self.trigger.oclAsType(SignalEvent).signal.reception->
      select(owner = getContextElement(self.source))->notEmpty
```

[2] Die Klasse, deren Verhalten durch den Zustandsautomaten beschrieben wird, muss die mit dem Ereignis verknüpfte Operation verarbeiten können.

```
self.trigger.notEmpty implies
  self.trigger.oclIsKindOf(CallEvent) implies
    self.trigger.oclAsType(CallEvent).operation.owner =
      getContextElement(self.source)
```

[3] Ein mit einer transition verknüpftes ChangeEvent muss ausführbar sein.

```
self.trigger.notEmpty implies
  self.trigger.oclIsKindOf(ChangeEvent) implies
    self.trigger.oclAsType(ChangeEvent).changeExpression.
      oclIsKindOf(ExecutableBooleanExpression)
```

[4] Ein mit einer Transition verknüpfter Guard muss ausführbar sein.

```
self.guard.notEmpty implies

self.guard.expression.oclIsKindOf(ExecutableBooleanExpression)
```

Zusätzliche Operationen

[1] Ermittelt das Kontextelement, dessen Verhalten mit dem Zustandsautomaten beschrieben wird, in dem `s` enthalten ist.

```
getContextElement(s : StateVertex) : ModelElement
result = if (s.oclIsKindOf(CompositeState)) and
           (s.oclAsType(State).StateMachine.notEmpty) then
           s.oclAsType(State).StateMachine.context
         else
           getContextElement(s.container)
```

```
endif
```

A.5.2.11 AbstractXFREETransition

Beschreibung

Modellierung von Transitionen in einem abstrakten Zustandsautomaten.

Statische Definitionen

[1] Eine mit einem CallEvent verknüpfte Operation muss über die öffentliche Schnittstelle sichtbar sein.

```
self.trigger.notEmpty implies
  self.trigger.oclIsKindOf(CallEvent) implies
    self.trigger.oclAsType(CallEvent).operation.
      visibility = #public
```

[2] Handelt es sich bei dem Ereignis um ein SignalEvent, so muss die Klasse das zu dem Ereignis gehörige Signal verarbeiten können und dies muss an der öffentlichen Schnittstelle deklariert sein.

```
self.trigger.notEmpty implies
  self.trigger.oclIsKindOf(SignalEvent) implies
    self.trigger.oclAsType(SignalEvent).signal.reception->
      select(visibility = #public)->size = 1
```

[3] Der Ausdruck eines ChangeEvents muss ausführbar sein.

```
self.trigger.notEmpty implies
  self.trigger.oclIsKindOf(ChangeEvent) implies
    self.trigger.oclAsType(ChangeEvent).changeExpression.
      oclIsKindOf(AbstractBooleanExecutableExpression)
```

[4] Der Ausdruck eines Guards muss ausführbar sein.

```
self.guard.notEmpty implies
  self.guard.expression.
    oclIsKindOf(AbstractBooleanExecutableExpression)
```

A.5.2.12 SpecialXFREETransition

Beschreibung

Modellierung von speziellen Eigenschaften von Transitionen innerhalb des *XFREE State Model*.

A.5.2.13 ImplicitTransition

Beschreibung

Diese Transitionen werden dazu verwendet, sogenannte 'else'-Fälle zu modellieren. Eine solche Transition wird immer dann aktiviert, wenn für die aktuelle Kombination aus Ereignis und Bedingung keine gültige Transition ermittelt werden kann. D.h. es kann für jede Menge von Transitionen mit dem gleichen Ereignis (inklusive dem leeren Ereignis) genau eine Transition mit diesem Stereotypen markiert werden. Durch die Verwendung dieser Transitionen wird die vollständige Definition des Zustandsautomaten ermöglicht.

Statische Definitionen

[1] Als Guard wird ein vordefinierter Guard verwendet, der mit dem Schlüsselwort '`else`' markiert ist. Dies ist der gleiche Mechanismus, wie er bei aus einem '`junction`'-Pseudozustand ausgehenden Transitionen verwendet wird. Die Art der Definition ist in der UML-Spezifikation sehr ungenau und wurde hier lediglich übernommen.

```
self.guard.expression = 'else'
```

A.5.2.14 IllegalTransition

Beschreibung

Eine Transition, mit deren Hilfe die Reaktion der Klasse bei einer Verletzung von Bedingungen modelliert wird.

Statische Definitionen

[1] Eine solche Transition ist eine interne Transitionen, da der abstrakte Zustand nicht geändert werden soll.

```
self.state.notEmpty
```

A.5.2.15 SetFlagEvent

Beschreibung

Es wird ein Fehlercode zurückgegeben.

Statische Definitionen

[1] An der Transition muss eine Aktion annotiert sein.

```
self.effect.notEmpty
```

[2] Die annotierte Aktion ist eine einzelne Aktion, die einen Wert an den Klienten zurück gibt.

```
self.effect.oclIsKindOf(ReturnAction) and
self.effect.actualArgument->notEmpty
```

A.5.2.16 RejectEvent

Beschreibung

Es wird eine Ausnahme erzeugt.

Statische Definitionen

[1] An der Transition muss eine Aktion annotiert sein.

```
self.effect.notEmpty
```

[2] Die annotierte Aktion ist eine einzelne Aktion, die ein asynchrones Signal, hier eine Ausnahme, sendet.

```
self.effect.oclIsTypeOf(SendAction) and
self.effect.oclAsType(SendAction).signal.oclIsKindOf(Exception)
```

A.5.2.17 AbendEvent

Beschreibung

Das Programm soll "abstürzen" (*abnormal termination*).

A.5.2.18 OrderedTransition

Beschreibung

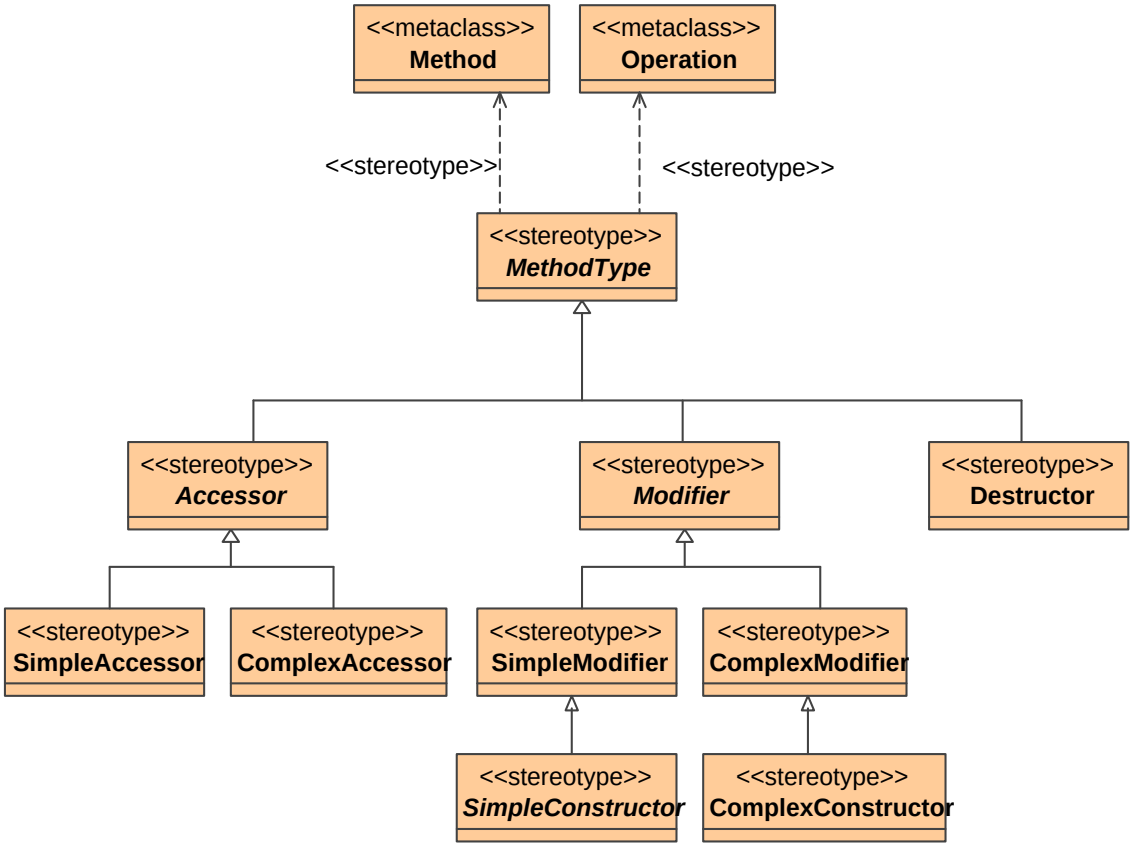
Alle Transitionen, die vom selben Zustand abgehen und den gleichen Trigger besitzen, müssen mit diesem Stereotypen markiert werden. Damit wird festgelegt (durch den Wert des Tags), in welcher Reihenfolge die Guards an diesen Transitionen ausgewertet werden. Die erste Transition, für die der Guard nach "wahr" auswertet, wird aktiviert. So kann zu jeder Zeit eindeutig eine Transition festgelegt werden, die aktiviert wird, auch wenn mehrere Guards gleichzeitig nach "wahr" ausgewertet werden würden.

Tagdefinitionen

evaluationPosition: Legt die Position innerhalb der Auswertungsreihenfolge fest. Dabei wird der Guard einer Transition zuerst ausgewertet, der den kleinsten Wert besitzt. Der Wert kann auch negativ sein, um eine leichtere Anpassbarkeit zu gewährleisten.

A.6 Method Properties

A.6.1 Abstrakte Syntax



A.6.2 Beschreibung und statische Regeln

A.6.2.1 MethodType

Beschreibung

Klassifikation von Methoden nach ihren Eigenschaften.

Statische Definitionen

[1] Alle Ein- und Durchgangsparameter werden für die Realisierung der Funktionalität der Methode gebraucht.

```
Alle Eingangs- und Durchgangsparameter der Methode werden auf
der rechten Seite von mindestens einer Nachbedingung der
Methode verwendet.
```

[2] Alle Ausgaben der Methode sollen eindeutig bestimmbar sein.

```
Alle Ausgangs-, Durchgangs- und Rückgabeparameter werden auf
der linken Seite von genau einer Nachbedingung verwendet.
```

A.6.2.2 Accessor

Beschreibung

Eine Methode, die den Systemzustand nicht ändert.

Statische Definitionen

[1] Die Methode verändert den Zustand des Systems nicht.

```
self.isQuery
```

[2] Die Nachbedingungen müssen wiederspiegeln, dass die Methode den Zustand des Systems nicht ändert.

```
Auf den linken Seiten der Nachbedingungen sind nur Ausgangs-,
Durchgangs- und Rückgabeparameter der Methode enthalten.
```

A.6.2.3 SimpleAccessor

Beschreibung

Eine Methode, die den Wert von Instanzvariablen des entsprechenden Objekts unverändert zurückgibt und dabei den Systemzustand nicht ändert.

Statische Definitionen

[1] Die Methode besitzt nur einen Rückgabeparameter und Ausgangsparameter.

```
self.parameter->select(kind = #return)->size = 1 and
self.parameter->select(kind <> #return and kind <> #out)->
  size = 0
```

A.6.2.4 ComplexAccessor

Beschreibung

Eine Methode, die "komplexe" Aktionen durchführt um die Aus- und Rückgabeparameter zu ermitteln und dabei den Systemzustand nicht ändert. "Komplex" bedeutet hier, dass der Wert eines Attributs nicht direkt zurückgegeben wird, sondern Berechnungen durchgeführt oder Operationen aufgerufen werden.

A.6.2.5 Modifier

Beschreibung

Eine Methode, die den Systemzustand ändern kann.

Statische Definitionen

[1] Die Methode kann den Zustand des Systems ändern.

```
not self.isQuery
```

A.6.2.6 SimpleModifier

Beschreibung

Eine Methode, die den Wert der Eingangsparameter unverändert an Instanzvariable der Klasse zuweist.

Statische Definitionen

[1] Die Methode besitzt nur Eingangsparameter.

```
self.parameter->select(kind <> #in)->size = 0 and
self.parameter->select(kind = #in)->size > 0
```

A.6.2.7 SimpleConstructor

Beschreibung

Eine Methode, die ein Objekt der Klasse erzeugt und initialisiert. Dabei wird das Objekt aus dem Alphazustand in einen "normalen" Zustand überführt.

A.6.2.8 ComplexModifier

Beschreibung

Eine Methode, die "komplexe" Operationen durchführt und dabei den Systemzustand ändert.

A.6.2.9 ComplexConstructor

Beschreibung

Eine Methode, die ein Objekt der Klasse erzeugt und initialisiert. Dabei wird das Objekt aus dem Alphazustand in einen "normalen" Zustand überführt.

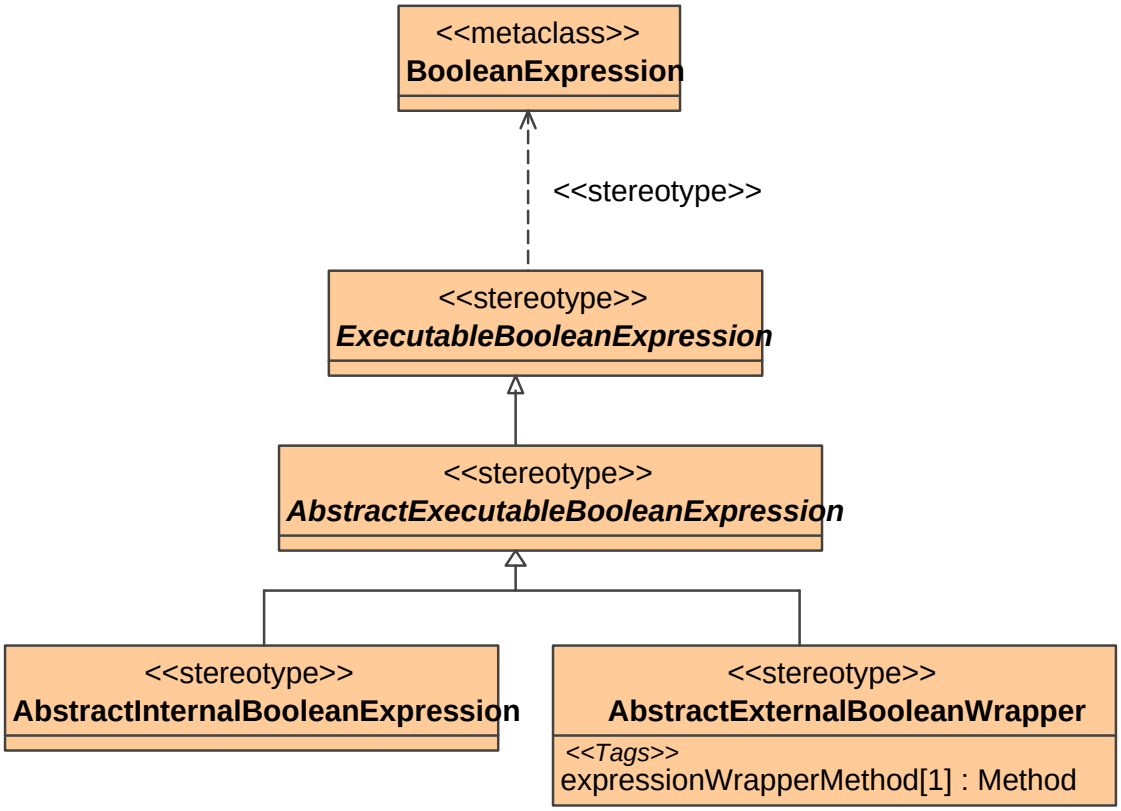
A.6.2.10 Destructor

Beschreibung

Eine Methode die ein Objekt der Klasse zerstört, es also in den Omegazustand überführt.

A.7 Executable Expression

A.7.1 Abstrakte Syntax



A.7.2 Beschreibungen und statische Definitionen

A.7.2.1 ExecutableBooleanExpression

Beschreibung

Der boolsche Ausdruck ist so definiert, dass entweder aus ihm ausführbarer Code generiert werden kann, oder es wird eine Wrappermethode angegeben, die den Ausdruck auswertet.

A.7.2.2 AbstractExecutableBooleanExpression

Beschreibung

Der Ausdruck ist so definiert ist, dass er ausführbar ist und nur die öffentliche Schnittstelle betrachtet wird.

Statische Definitionen

[1] Um testbar zu sein, muss eine Einschränkung ausführbar und öffentlich zugänglich sein.

```
Alle zur Definition verwendeten Elemente sind öffentlich
definiert und zustandserhaltend.
```

A.7.2.3 AbstractInternalBooleanExpression

Beschreibung

Ausdruck zur Beschreibung von Eigenschaften einer Klasse (Vorbedingungen, Nachbedingungen, Klasseninvarinaten, Zustandsinvarianten, Guards, etc) mit an deren öffentlicher Schnittstelle sichtbaren Elementen. Dieser muss so definiert sein, dass daraus ausführbarer Code generiert werden kann.

Statische Definitionen

[1] Der Test soll auf die öffentliche Schnittstelle der Klasse beschränkt werden. Dem entsprechend muss die Spezifikation so angepasst sein, dass nur die für Klienten über die öffentliche Schnittstelle der Klasse zugreifbaren Informationen verwendet werden. Die Einschränkung auf Methoden ohne Ein- und Durchgangsparameter muss gefordert werden, da ansonsten eine automatische Sollwertermittlung nicht möglich ist.

```
Für die Spezifikation werden nur Ausgangs- und
Rückgabeparameter von öffentlichen, zustandserhaltenden
Methoden verwendet, die keine Ein- und Durchgangsparameter
besitzen. Außerdem werden nur auf den entsprechenden Typen
definierte, öffentliche Operatoren und Operationen verwendet.
```

A.7.2.4 AbstractExternalBooleanWrapper

Beschreibung

Ist ein Ausdruck zu komplex, um ihn mit den Mitteln der *OCL* darstellen zu können, muss eine Methode spezifiziert werden, die als Wrapper zu einer konkreten Realisierung der Einschränkung dient.

Tagdefinitionen

expressionWrapperMethod: Wrappermethode, die die entsprechende Einschränkung überprüft und das Ergebnis als Rückgabewert (boolean) zurück gibt.

Statische Definitionen

[1] Die Wrappermethode besitzt einen Rückgabeparameter vom Typ Boolean.

```
let returnParameter : Sequence(Parameter) =
  self.constraintWrapperMethod.parameter->
    select(kind = #return) in
  returnParameter->size = 1 and
  returnParameter.type.oclIsKindOf(Boolean)
```

[2] Alle Parameter der Wrappermethode außer dem Rückgabeparameter sind Eingangsparameter.

```
self.expressionWrapperMethod.parameter->
  select(kind <> #return)->forAll(kind = #in)
```

[3] Die Wrappermethode besitzt die folgenden Eingangsparameter in der angegebenen Reihenfolge: Entsprechend der Definitionsreihenfolge aller zustandserhaltenden Operationen des verknüpften Typen sind jeweils zuerst ein Parameter für deren Rückgabeparameter und daran anschließend für alle Ausgangsparameter jeweils ein Parameter mit dem entsprechenden Typen enthalten.

```
self.expressionWrapperMethod.parameter->select(kind = #in)->
  collect(type) =
    getPublicClassifierSignature(getRelatedClassifier)
```

Zusätzlicher Operationen

[1] Operation erzeugt eine Sequenz von Typen für die Rückgabe- und Ausgangsparameter aller öffentlichen, zustandserhaltenden Operationen.

```
getPublicClassifierSignature(c : Classifier) :
  Sequence(Classifier);
result = c.feature->select(oclIsKindOf(BehavioralFeature) and
         isQuery and ownerScope = #public))->iterate(
           f : BehavioralFeature;
           result = Sequence{} |
           result.union(getPublicFeatureSignature(f)))
```

[2] Diese Operation erstellt für die übergebene Eigenschaft eine Sequenz, die den Typ des Rückgabeparameters (sofern vorhanden) und daran anschließend die Typen aller Ausgangsparameter enthält.

```
getPublicFeatureSignature(f : BehavioralFeature) :
  Sequence(Classifier);
result =
  f.parameter->select(kind = #return)->collect(type)->
    union(f.parameter->select(kind = #out)->collect(type))
```

[3] Diese Operation liefert den Typen (Classifier), mit dem der Ausdruck direkt oder indirekt verknüpft ist.

```
getRelatedClassifier() : Classifier;
result =
  if self.owner.(oclIsKindOf(Constraint)) then
    if self.owner.
      constrainedElement(oclIsKindOf(Classifier)) then
      self.owner.constrainedElement.oclAsType(Classifier)
    else if self.owner.
      constrainedElement(oclIsKindOf(State)) then
      getTopState(self.owner.constrainedElement.
        oclAsType(State)).stateMachine.context.
          oclAsType(Classifier)
    else if self.owner.
      constrainedElement(oclIsKindOf(BehavioralFeature)) then
      self.owner.constrainedElement.
        oclAsType(BehavioralFeature).owner
    endif
  else if self.owner.oclIsKindOf(Guard) then
    self.owner.oclAsType(Guard).transition.stateMachine.
      context.oclAsType(Classifier)
  endif
```

Literaturverzeichnis

[Alexander+77] Christpher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel: *A Pattern Language*. Oxford University Press, 1977.

[Barbey97] Stéphane Barbey: *Test selection for specification-based unit testing of object-oriented software based on formal specifications*. Ph.D, diss., Swiss Federal Insitute of Technology, Computer Science Department, Software Engineering Laboratory, 1997.

[Beizer90] Boris Beizer: *Software Testing Techniques.* 2. ed. New York: International thompson Computer Press, 1990.

[Binder94] Robert V. Binder: *Design for Testability in Object-Oriented Systems.* Communications of the ACM. 37(9): 87–101, September 1994.

[Binder94a] Robert V. Binder: *Testing Object-Oriented Systems: A Status Report*. American Programmer. 7(4): 22–28, April 1994

[Binder99] Robert V. Binder: *Testing Object-Oriented Systems*: Models, Patterns, and Tools. Addison Wesley, 1999.

[Bohner+96] Shawn A. Bohner, Robert S. Arnold: *Software Change Impact Analysis*. Los Alamitos, Calif.: IEEE Press, 1996.

[Booch+99] Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[Booch+99a] Grady Booch, James Rumbaugh, Ivar Jacobson: *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[Buschmann+98] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: *Pattern-orientierte Software-Architektur*. Addison Wesley, 1998.

[Fewster+99] Mark Fewster, Dorothy Graham: *Software Test Automation.* Addison Wesley, 1999.

[Fraikin+00] Falk Fraikin, Thomas Leonhardt: *Top-Down Testen auf der Basis von Sequenzdiagrammen*. Diplomarbeit, TU Darmstadt, 2000.

[Fraikin01] Falk Fraikin, *SeDiTeC – Testen auf der Basis von Sequenzdiagrammen*. Z. Softwaretechnik-Trends. 21(3), 2001.

[Gamma+98] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Entwurfsmuster*. Addison Wesley, 1998.

[Harrold+92] Mary Jean Harrold, John D. McGregor, Kevin J. Fitzpatrick: *Incremental Testing of Object-Oriented Class Structures.* Proceedings of the 14th International Conference on Software Engineering, Los Alamitos, Calif.: IEEE Computer Society Press, May 1992.

[Hesse+94] W. Hesse, G.Barkow, H. von Braun, H.-B. Kittlaus, G.Scheschonk: *Terminologie der Softwaretechnik, Ein Begriffssystem für die Analyse und Modellierung von Anwendungssystemen, Teil 2: Tätigkeits- und ergebnisbezogene Elemente*. Z. Informatik-Spektrum, Nr. 17 (1994): 96–105.

[Holzmann91] Gerald J. Holzmann: *Design and validation of computer protocols*. Prentice Hall, Inc., 1991.

[Hopcroft+79] John E. Hopcroft, Jeffrey D. Ullman: *Introduction to automata theory, languages, and computation*. Addison-Wesley, 1979.

[Jacobson+92] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, Gunnar Overgaard: *Object-oriented software engineering*. Addison-Wesley, 1992.

[Jorgensen+94] Paul C. Jorgensen, Carl Erickson: *Object-Oriented Integration Testing*. Communications of the ACM. 37(9): 30–38, September 1994.

[Jungmayr+99] Stefan Jungmayr, Mario Winter: *OO-Testwerkzeuge: Wunsch und Wirklichkeit*. TAV 13. Siemens AG, München, Februar 1999.

[Kleiner98] Max Kleiner: *Unified Modeling Language*. Z. M+K Computermarkt, Nr. 5 (1998): 40–42.

[Kung+95] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Cris Chen: *A Test Strategy for Object-Oriented Programs*. Proceedings of the 19th Annual International Computer Software and Applications Conference (COMPSAC'95), Los Alamitos, Calif.: IEEE Computer Society Press: 239–244, 1995.

[Lewis+81] Harry R. Lewis, Christos H. Papadimitriou: *Elements of theory of computation*. Prentice-Hall, 1981.

[Liggesmeyer+96] Peter Liggesmeyer, Peter Rüppel: *Die Prüfung von objektorientierten Systemen*. Z. OBJEKTspektrum, Nr. 6 (1996): 68–78.

[Liskov+94] Barbara A. Liskov, Jeannette M. Wing: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems. 16(6): 1811–1841, November 1994.

[Manna+78] Zohar Mann, Richard Waldinger: *The logic of computer programming*. IEEE Transactions on Software Engineering. SE-4(3): 199–229, May 1978.

[McGregor+94] John D. McGregor, T. D. Korson: *Integrating Object-Oriented Testing and Development Processes*. Communications of the ACM. 37(9): 59–77, September 1994.

[McGregor+01] John D. McGregor, David A. Sykes: *A Practical Guide to Testing Object-Oriented Software*. Addison Wesley, 2001.

[Mealy55] George H. Mealy: *A method for synthesizing sequential circuits*. Bell Systems Technical Journal 34: 1045–1079, September 1955.

[Meyer97] Bertrand Meyer: *Object-oriented software construction*. Prentice Hall, 1997.

[Moore56] Edward P. Moore: *Gedanken-experiments on sequential machines*. Automata studies: annals of mathematical studies (34), Princeton University Press, 1956.

[Myers79] Glenford J. Myers: *The art of software testing*. John Wiley & Sons, 1979

[OMG99] *Object Constraint Language Specification*. Version 1.3. Object Management Group (OMG). Internet: http://www.omg.org, 1999.

[OMG01] *Unified Modeling Language Specification*. Version 1.4. Object Management Group (OMG). Internet: http://www.omg.org, 2001.

[Overbeck94] Jan Overbeck: *Integration Testing for Object-Oriented Software*. Dissertation, TU Wien, 1994.

[Pelkmann98] Ute Pelkmann: *Testen von OO-Programmen – Problematik des Klassentests am Beispiel C++*. Z. Softwaretechnik-Trends. 18(2): 20–21, Mai 1998.

[Perry+90] Dewayne E. Perry, Gail E. Kaiser: *Adequate testing and object-oriented programming*. Journal of Object-Oriented Programming. 2(5): 13–19, January/February 1990.

[Poston96] Robert M. Poston: *Automating Specification-Based Software Testing*. IEEE Computer Society Press, 1996.

[Roselli98] Werner Roselli: *Softwaresysteme entwerfen mit UML*. Z. Elektronik, 4 (1998): 88–97.

[Rothermel+94] Gregg Rothermel, Mary Jean Harrold: *Selecting Regression Tests for Object-Oriented Software*. Proceedings, Conference on Software Maintenance, Los Alamitos, Calif.: IEEE Computer Society Press, October 1994.

[Rüppel97] Peter Rüppel: *Ein generisches Werkzeug für den objektorientierten Softwaretest*. Dissertation, TU Berlin, 1997.

[Rumbaugh+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorenson: *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Wahl98] Günter Wahl: *UML kompakt*. Z. OBJEKTspektrum. Nr. 2 (1998): 22–33.

[Warmer+99] Jos Warmer, Anneke Kleppe: *The Object Constraint Language*. Addison Wesley, 1999.

[Weiß97] Christoph Weiß: *Das Use-Case-Konzept*. Z. Softwaretechnik-Trends. 17(4): 26–31, November 1997.

[Weyuker88] Elaine Weyuker: *The evaluation of program-based software test data adequacy criteria*. Communications of the ACM. 31(6): 668–675, June 1988.

[Winter00] Mario Winter: *Ein interaktionsbasiertes Modell für den objektorientierten Integrations- und Regressionstest*. Z. Informatik Forschung und Entwicklung. 15 (2000): 121–132.

[Winter01] Mario Winter: *Test dynamisch gebundener Operationsaufrufe*. Z. Softwaretechnik-Trends. 21(1), Februar 2001.