Technische Universität Graz (TUG)
Fakultät für Informatik
Lehrstuhl Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Qualitätssicherung in verteilten Systemen

Testautomatisierung mittels Modellbasiertem Test

Diplomarbeit

vorgelegt von Philipp Haiden

8. Dezember 2011

Betreuer: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa
Kooperationspartner: NTE Naturenergie. Technology and Engineering GmbH

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..
date (signature)

Inhaltsverzeichnis

Abbildungsverzeichnis

Abstract

Softwareprojects are getting bigger, more complex and in consequence more error prone. So extensive testing of this systems is an essential success factor. Only when adequate quality and stability is achieved, a system will be commercially sucessful.

The project discussed in this thesis is a SOA-based application used to monitor and control facility systems. The system will be tested only by the software developers themselves as no dedicated quality assurance division exists. Both of this aspects (SOA and developers as testers) will be adressed in this thesis. The thesis will describe the problems faced while developing this system, what the solutions were found to address them. Also, it will be shown how an adequate quality could be achieved, by using test automation techniques - especially the so called testing method model based testing - in reference to the SOA aspect.

Zusammenfassung

Softwareprojekte werden immer größer, somit komplexer und in Folge dessen auch immer fehleranfälliger. Daher ist das ausgiebige Testen dieser Systeme ein essentieller Erfolgsfaktor derselbigen. Denn nur wenn eine ausreichende Qualität und Stabilität gewährleistet ist, wird ein System die benötigte Akzeptanz und Verbreitung finden, um wirtschaftlich rentabel zu sein.

Bei dem Projekt handelt es sich um ein SOA-basiertes Anlagenmonitoring- und Steuerungssystem, welches auch direkt vom Entwicklerteam getestet wird, da keine eigenständige Testabteilung existiert. Auf diese beiden Aspekte des Projektes wird in dieser Arbeit besonders eingegangen, denn beide stellen besondere Anforderungen an die Qualitätssicherung. Die Arbeit wird erörtern, mit welchen Problemen das Team während der Entwicklung konfrontiert wurde, und welche Lösungen bzw. Lösungsansätze erarbeitet wurden. Folgend wird aufgezeigt, wie die Produktqualität gewährleistet werden konnte, indem automatisierte Testtechniken - insbesondere der modellbasierte Test - eingesetzt wurden.

1 Einleitung

„Wenn Debugging der Vorgang ist, Fehler aus Software zu entfernen, dann ist Programmieren der Vorgang, Fehler einzubauen.“ (unbekannt)

1.1 Motivation

Ein sehr wichtiges Ziel in der Softwareentwicklung ist es, Software fehlerfrei zu erstellen. Korrektes Systemverhalten und hohe Qualität sind essentielle Faktoren, welche für Systeme, die wirtschaftlich rentabel sein sollen unerlässlich sind.

Softwaresysteme, welche fehlerhaft oder nicht entsprechend der gewünschten Qualitätsanforderungen funktionieren, führen für Unternehmen meist zu enormen direkten und auch indirekten wirtschaftlichen Verlusten, wie z.B. einer niedrigeren Reputation, welche mitunter noch weit schlimmere Auswirkungen zur Folge haben kann.

Allein für Europa wurden im Jahr 2009 die wirtschaftlichen Verluste durch Softwarefehler auf 150 Milliarden Euro geschätzt [23].

Für den Großteil der Unternehmungen ist es selbstverständlich, Systeme zu testen. Allerdings stellt sich heraus, dass das Vorgehen in den Unternehmen sehr unterschiedlich ist. Oftmals ist es für die Unternehmungen äußerst schwierig, ihre eigenen Testmethoden und Prozesse objektiv zu bewerten, da im Bereich Softwaretest kein eindeutiges Vokabular und keine exakten Definitionen anzutreffen sind. Des Weiteren zeigt sich, dass der Testprozess oftmals zu optimistisch bewertet wird [32][31].

Besonders für kleine Unternehmungen stellt der Aufbau einer angemessenen Qualitätssicherung oftmals eine große Herausforderung dar. Diese besitzen meist nicht die notwendigen Ressourcen, eine eigene Testabteilung zu etablieren, weshalb oftmals die Entwickler selbst den Softwaretest übernehmen müssen. Zusätzlich stehen diese auch unter dem Druck, möglichst schnell ein Produkt am Markt anzusiedeln, damit so bald als möglich Umsätze generiert werden. Weitere Faktoren, welche schlechte Qualitätssicherung unter diesen Gegebenheiten bedingen, sind [1]:

- Entwickler mit wenig Erfahrung, um Kosten zu sparen
- Großer Anteil an Technologierecherchen
- Einsatz von Technologien, welche von den Projektbeteiligten noch nicht richtig beherrscht werden
- Organisatorische Struktur definiert keine festen Kompetenzen

[1] Nur ein Auszug an Faktoren, da es natürlich noch viele weitere existieren

Alle diese Dinge führen dazu, dass besonders kleine Unternehmungen - jene Kategorie, der jedoch der Großteil der Softwareentwicklungshäuser angehört [45] - oftmals nicht in der Lage sind Produkte mit ausreichender Qualität zu erstellen.

Aufgrund des oben beschriebenen Sachverhalts wird sich diese Arbeit mit dem Thema Softwaretest in einem kleinem Startup-Unternehmen befassen, welches es sich zum Ziel gesetzt hat, eine SOA-basierte Plattform für Anlagenmonitoring und Steuerung umzusetzen und als Produkt am Markt zu etablieren.

1.2 Zielsetzung

Ziel der Arbeit ist es darzulegen, wie in einem kleinen Softwareteam (das Team besteht aus sechs Personen) ein Qualitätssicherungsprozess eingeführt werden konnte. Im Besonderen wird gezeigt, welche Probleme entstanden und gelöst wurden und wie dieser Qualitätssicherungsprozess in den eigentlichen Entwicklungsprozess integriert werden konnte. Schließlich wird die Arbeit auch noch aufzeigen, welche Testtechniken verwendet wurden, um den Testaufwand für das Team zu minimieren.

1.3 Abgrenzung

Die Arbeit konzentriert sich im Speziellen auf das Testen einer SOA-basierten verteilten Umgebung. Primär wird auf das Testen des Servers eingegangen, die Arbeit bezieht sich somit nicht auf das Testen des Clients (GUI-Test usw.).

Die Arbeit stellt nicht den Anspruch, die ideale Vorgehensweise bei der Einführung eines Testprozesses aufzuzeigen, vielmehr soll die Arbeit festhalten, mit welchen Problemen das Team während dieser Vorgehensweise konfrontiert war, und wie diese - zumindest teilweise - gelöst werden konnten. Somit kann die Arbeit als Hilfestellung für Teams in ähnlicher Ausgangslage gesehen werden und ihnen Anhaltspunkte bei der Einführung einer Qualitätssicherung liefern.

1.4 Aufbau dieser Arbeit

Der erste Teil der Arbeit beschäftigt sich mit den theoretischen Grundlagen und der Einführung in die Thematik.

Zu Beginn wird das Thema "Verteilte Systeme" und deren spezielle Anforderungen im Bezug auf das Testen erörtert. Weiters wird die Arbeit auf die Grundlagen des Softwaretestens an sich eingehen.

Der praxisorientierte Teil der Arbeit zeigt einleitend die grundlegende Architektur und Struktur des Softwareprojekts NTE.CLM, da die Qualitätssicherung eines Projekts sehr

stark von dieser geprägt ist. Folgend wird dann noch auf die Testtechniken, welche eingesetzt wurden eingegangen.

Die Kapitel Fazit und Ausblick stellen den Abschluss der Arbeit dar.

2 Verteilte Architekturen

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable." (Leslie Lammport)

2.1 Verteilte Systeme

Für den Begriff „Verteiltes System" gibt es mehrere Definitionen. Eine Definition nach George Coulouris lautet: „Ein verteiltes System (VS) ist ein System, in dem sich HW- und SW-Komponenten auf vernetzten Computern befinden und miteinander über den Austausch von Nachrichten kommunizieren". Andrew Tannenbaum definiert sie als „Zusammenschluss unabhängiger Computer, der sich für den Benutzer als ein einzelnes System präsentiert". Peter Löhr verwendet eine etwas allgemeinere Definition welche besagt, dass ein verteiltes System die Menge aller Prozesse (oder Prozessoren) ist, die über keinen gemeinsamen Speicher verfügen und deshalb über Nachrichten miteinander kommunizieren.

In der Praxis werden verteile Systeme oftmals mit Rechnernetzen gleichgesetzt. Dies entspricht aber nicht der eigentlichen Definition eines verteilten Systems. Ein verteiltes System versucht nämlich, den Umstand, dass es sich bei dem System um einen Rechnerverbund handelt, zu verstecken.

Verteilte Systeme sind äußerst komplex und somit aufwendig zu entwickeln. Trotz dieser Komplexität werden verteilte Architekturen immer häufiger eingesetzt. Der Grund dafür liegt wohl darin, dass heutige Geschäftsanwendungen immer öfter Anforderungen stellen, welche von verteilten Systemen ausgezeichnet bedient werden können. Des Weiteren hat die rasante technologische Entwicklung im Bereich der Netzwerke dazu beigetragen.

Laut [36][11] sind die Charakteristiken, die für den Einsatz eines verteilten Systems sprechen, folgende:

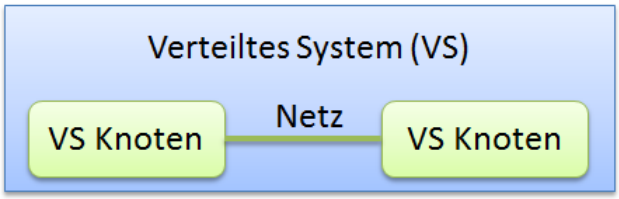

Abbildung 1: Verteiltes System

- Funktionale Aufteilung
  Verteilte Systeme stellen Services zur Verfügung. Jeder Service stellt eine bestimme Systemfunktionalität für die Außenwelt zur Verfügung. Damit lässt sich eine sehr gute funktionale Separation des Systems erreichen.

- Inhärente Verteilung
  Daten, Personen und Informationen sind inhärent im System verteilt. D.h. unter-

schiedliche Komponenten können Daten sammeln und bereitstellen. Wiederum andere Systeme verarbeiten und analysieren diese Daten, ohne von der erst genannten Komponente zu wissen.

- Ausfallsicherheit
  Wenn im System einzelne Knoten ausfallen, können andere Knoten einspringen und übernehmen. Zusätzlich können Backup-Systeme in das System integriert werden.
- Skalierbarkeit
  In das System können mit niedrigem Aufwand neue Ressourcen (z.B. weitere Servermaschinen) integriert werden, ohne dass das Systemverhalten sich nach außen hin ändert. Dadurch ist es möglich, das System an steigende Ressourcenbedarfe anzupassen.
- Wirtschaftlichkeit
  Systemressourcen können von den Systemkomponenten gemeinsam genutzt werden und somit mitunter Kosten gespart werden. Ein aktuelles und anschauliches Beispiel für diesen Punkt sind Cloud-Computing-Systeme.

2.2 Verteilte Anwendung

Die verteilte Anwendung ist die eigentliche Anwendunglogik, n Form mehrerer VA-Komponenten, welche auf verschiedenen VS-Knoten (Rechnern) laufen.

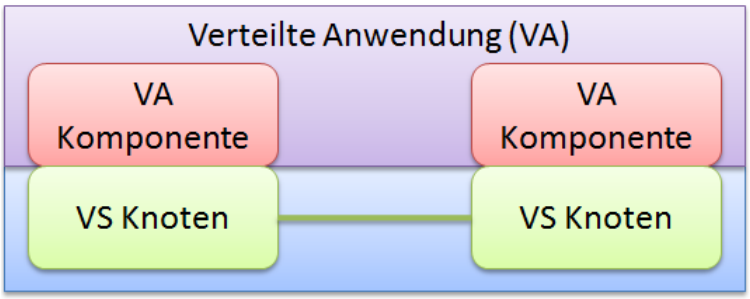

Abbildung 2: Verteilte Anwendung

2.3 Middleware

Es ist möglich, verteilte Systeme mit jeder beliebigen Kommunikationstechnologie - wie z.B. Sockets - zu realisieren. Die Entwicklung der Infrastruktur mittels einer solch grundlegenden Kommunikationstechnologie bietet einige Vorteile, z.B. hohe Performance und gute Steuerbarkeit. Doch der Aufwand dafür ist sehr hoch, sodass diese meist nur in Anwendungen mit sehr speziellen Anforderungen sinnvoll ist. Aufgrund dessen wird in der Praxis der Anwendungslayer auf einer bereits etablierten Middleware aufgesetzt.

Middleware ist ein Softwarelayer, welcher über dem Betriebssystem aber unterhalb der der eigentlichen Applikation angesiedelt ist [6].

Die Middleware ist ein Framework, welcher es Entwicklern ermöglicht, heterogene Systeme (unterschiedliche Plattformen und Technologien) mittels einer einheitlichen Infrastruktur zu verbinden. Beispiele für eine solche Middleware sind die Windows Communication Foundation (WCF), Common Object Request Broker Architecture (CORBA) und Java Messaging Service (JMS), um nur einige zu nennen.

Durch die Middleware wird sichergestellt, dass einheitliche Protokolle (z.B. SOAP) und Standards (z.B. WSDL) eingesetzt werden. Es wird sichergestellt, dass alle Systemkomponenten, welche sich an diese Standards halten, mit Systemkomponenten, welche diese Standards ebenfalls erfüllen, interagieren können.

Zusätzlich bietet die Middleware die Basisfunktionalität und Schnittstellen, welche für die Entwicklung einer verteilten Anwendung gebräuchlich sind:

- Sichere Nachrichtenübertragung (Datenverlust)
- Authentifizierung, Identitäten-Management und Sicherheitsmechanismen
- Transaktionen
- Routing
- Serialisierung
- uvm.

Durch den Einsatz von bestehender Middleware haben Entwickler die Möglichkeit, sich auf die Erstellung der eigentlichen Businesslogik zu konzentrieren, anstatt sich mit den grundlegenden Problemen verteilter Systeme auseinander setzen zu müssen.

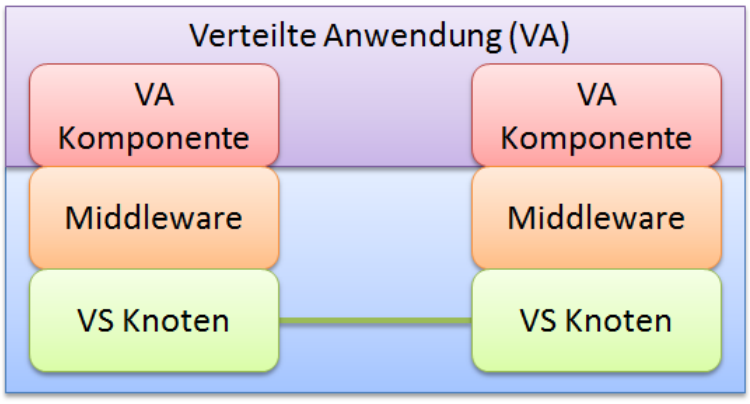

Abbildung 3: Middleware

2.4 Besondere Testanforderungen

Verteilte Systeme bieten viele Möglichkeiten. Doch die Architektur eines solchen Systems bringt auch viele Probleme - vor allem im Bezug auf das Testen - mit sich. Nachfolgend werden kurz die besonderen Herausforderungen, welche verteile Systeme bezüglich des Softwaretests mit sich bringen, dargelegt. Diese Eigenschaften sind sogenannte gemeine Eigenschaften, welche beinahe alle verteilten Systeme, unabhängig von ihrem speziellen Einsatzgebiet, inne haben [36] [11]

2.4.1 Heterogenität

Heterogenität im Bereich der verteilten Systeme bedeutet, dass die jeweiligen Komponenten des Systems sehr verschieden realisiert sein können. Die Komponenten können auf unterschiedlichen Plattformen laufen, welche mitunter weit voneinander entfernt sind, und sogar in verschiedenen Programmiersprachen implementiert sein.

Hier ergibt sich auch eines der größten Probleme der verteilten Architekturen bezüglich des Testens. Selbst wenn die einzelnen Komponenten des Systems an sich funktionieren, kann nicht sichergestellt sein, dass das Gesamtsystem auch korrekt arbeitet [2].

Es gibt keine Garantie, dass sich alle Komponenten an Kommunikationsstandards halten. Während des Tests einer Komponente können solche Dinge aber nur schwer geprüft werden, denn zu diesem Zeitpunkt besteht dafür oftmals noch nicht die Notwendigkeit/Möglichkeit. Diese Art des Softwaretests wird unter dem Begriff des Interoperabilität-Testens zusammengefasst.

2.4.2 Transparenz

Transparenz bedeutet im wesentlichen, dass es für Entitäten außerhalb des Systems nicht ersichtlich ist, dass das verteilte System aus einer Vielzahl an (heterogenen) Komponenten besteht. Stattdessen erscheint selbiges als ein einziges großes und einheitliches System [26]. Transparenz ist also das Verstecken von Interna vor der Außenwelt.

[2] Durch den Einsatz von Middleware im gesamten System kann dieses Problem zu einem großen Teil gelöst werden; Der Aspekt der Fehlerfreiheit im Gesamtsystem wird auch in 2.4.2 nochmals angesprochen

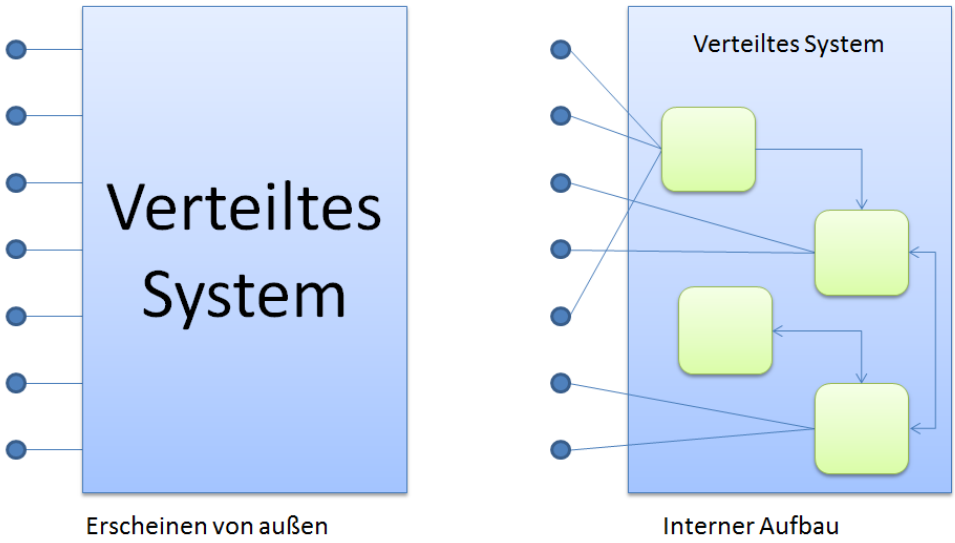

Abbildung 4: Transparenz in verteilten Systemen

Damit ergeben sich für den Softwaretest eines Systems weitreichende Folgen. Für systemexterne Entitäten ist das System eine Blackbox, welche einen gewissen internen Systemzustand besitzt und über Interfaces angesprochen werden kann.

Systemintern ist der Sachverhalt aber ganz und gar nicht so. Hier besteht das System aus einzelnen Komponenten, von denen keine Komponente den Systemzustand des Gesamtsystems kennt. Hier wird wiederum die Problematik beschrieben, die im vorhergehenden Punkt - der Heterogenität 2.4.1 - in ähnlicher Form angesprochen wurde.

Beim Test einer Komponente können nur die Komponenten selbst getestet werden. Das Verhalten des Gesamtsystems kann zu diesem Zeitpunkt noch nicht überprüft werden. Das Gesamtsystem kann erst getestet werden, wenn es das Gesamtsystem gibt. Primäres Mittel zum Test wird in diesem Bereich der Systemtest darstellen. Hier gibt es nun aber das Problem, dass Systemtests nur mit der externen Systemsicht erstellt werden, Systeminterna sind nicht mehr bekannt, was zu besonderen Problematiken beim Test führt.

Im Gegensatz zu 2.4.1 geht es hier nicht mehr um so grundlegende Fehler wie Kommunikationsprobleme zwischen den Komponenten, sondern ob das gewünschte Gesamtsystemverhalten nicht durch z.B. Deadlocks, Nicht-Determinismus bezüglich der Abarbeitungsreihenfolge usw. zerstört wird.

2.4.3 Fehlertoleranz

Fehlertoleranz bezeichnet die Fähigkeit eines Systems, auch mit einer begrenzten Anzahl fehlerhafter Komponenten seine spezifizierte Funktion zu erfüllen [21].

Im Rahmen von verteilten Systemen bedeutet dies im Wesentlichen, dass der Ausfall oder

das Fehlverhalten einer oder mehrerer Komponenten nicht zum Ausfall oder zu Fehlverhalten des Gesamtsystems führen darf. Logischerweise kann das Gesamtsystem jene Funktionalitäten, die im Aufgabenbereich der defekten Komponente liegen, nicht mehr erfüllen. Dennoch soll sich das Gesamtsystem unter Berücksichtigung des Fehlerfalls weiterhin logisch und konsistent verhalten.

Nach [13] werden vier Fehlerklassen definiert.

1. Aufall
   Ein Prozess reagiert nicht auf einen Event
2. Timeout
   Ein Prozess reagiert nicht innerhalb einer gewissen Zeitspanne
3. Absturz
   Ein Prozess stürzt ab und kann nicht weiter exekutiert werden
4. Byzantische Fehler
   Diese Fehlerklasse bezeichnet Fehler, bei welchen Komponenten in einem System an die Komponenten, mit welchen sie interagieren, verschiedene Resultate senden [24].

Fehlertoleranz durch Systemdesign zu verhindern ist äußerst schwierig, da es kaum möglich ist, alle Problemfälle eines Systems vorherzusehen. Daher ist die gängigste Methode, Fehlertoleranz zu erreichen, Ressourcen mehrfach bzw. unabhängig zur Verfügung zu stellen. Z.B. werden von einem Prozess mehrere Instanzen erzeugt, sodass, wenn eine Instanz nicht weiter ausgeführt werden kann, die nächste Instanz einspringen kann. Das System kann mitunter die erste Anfrage zwar nicht fertig bearbeiten, aber Folgeanfragen können wieder erfolgreich abgearbeitet werden.

Fehlertoleranz eines Systems durch Softwaretests nachzuweisen ist ein recht schwieriges Verfahren. Um die Fehlertoleranz zu überprüfen, werden im System künstlich Fehler erzeugt und das Systemverhalten beobachtet. Diese Methodik nennt man Fehlerinjektion. Eine gute und sinnvolle Integration dieser Methodik in den Entwicklungsprozess ist im Allgemeinen nicht einfach zu bewerkstelligen. Auch die Auswertung der Ergebnisse ist oft nicht trivial und aufwändig.

Weiter Informationen zu diesem Thema sind in [9] [19] [38] zu finden.

2.4.4 Nebenläufigkeit

Nebenläufigkeit bezeichnet im Wesentlichen die parallele Verarbeitung. Dies bedeutet, dass zur selben Zeit auf gemeinsam genutzte Ressourcen zugegriffen wird.

Da Ressourcen aber durch Zugriffe blockiert werden, muss im Softwaretest sichergestellt werden, dass Ressourcen auch wieder frei gegeben werden und somit nicht von einer Komponente dauerhaft belegt werden können, was mitunter zu einem Deadlock führen könnte.

2.4.5 Migration und Loadbalancing

Verteilte Systeme unterliegen im Allgemeinen einer ständigen Evolution. Bedingt ist diese dadurch, dass sich die externen Gegebenheiten ständig ändern, neue Komponenten müssen ins System integriert werden, Komponenten müssen auf andere Systemknoten ausgelagert werden, um die Performanceanforderungen weiterhin zu erfüllen usw.

Im Bereich des Softwaretests manifestierten sich diese Charakteristika eines verteilten Systems in Loadtests und Regressionstests.

Mithilfe der Regressionstests muss sichergestellt werden, dass nach Migrationstätigkeiten das System immer noch korrekt und wie gewünscht funktioniert. Regressionstests müssen unbedingt hoch automatisiert durchgeführt werden, da der Aufwand für diese ansonsten sehr schnell unüberschaubar werden kann.

Loadtests müssen zeigen, dass das System auch unter hoher Last noch immer angemessen funktioniert. Allgemein müssen die Antwortzeiten unterhalb einer bestimmten Obergrenze liegen, auch wenn tausende Anfragen das System erreichen. Um dies zu gewährleisten, ist der Loadtest unumgänglich, da während der Entwicklung nur durch Simulationstechniken ein solches Anfrageaufkommen erzeugt werden kann. Das aufsetzten einer Umgebung für diese Testmethodik muss gut durchdacht und geplant sein um ihren Zweck effizient erfüllen zu können.

2.4.6 Sicherheit

Sicherheit ist die Abwesenheit von Eigenschaften und Features, welche ein Risiko für den Operator einer Software oder Dritte darstellen, wenn diese mit bösartiger Absicht ausgenutzt werden [39].

Seit den ersten verteilten Systemen wurden spezielle Prüfmechanismen verwendet, um ungewolltes Eindringen oder andere Arten von Attacken zu unterbinden [27]. Verteilte Systeme sind für Softwareattacken besonders interessant, denn in solchen Plattformen sind meist sensible Daten von einer Vielzahl an Personen gespeichert. Aufgrund verschiedenster Intentionen sind diese Daten für gewisse Institutionen bzw. Personen äußerst attraktiv.

Die Sicherheit in verteilten Systemen zu testen und somit nachzuweisen ist sehr schwierig. Verteilte Systeme haben oftmals kein homogenes Authentifizierungssystem und Identitätsmanagement. Eine Vielzahl an Services, wobei für jeden einzelnen die Sicherheitsaspekte validiert werden müssen, treiben den Testaufwand sehr schnell in die Höhe.

Dass die Sicherheit von Software heute ein essentielles Qualitätskriterium darstellt, ist Softwareentwicklungshäusern bekannt. Dennoch steigt die Anzahl der Sicherheitslecks in Systemen kontinuierlich.

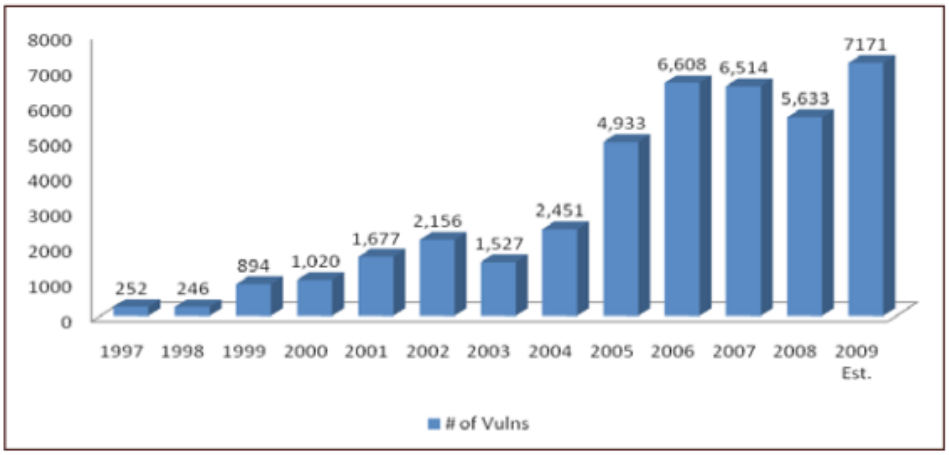

Abbildung 5: Anzahl der Sicherheitslecks in Softwaresystemen [30]

3 Softwaretest Grundlagen

In diesem Kapitel der Arbeit sollen nun die grundlegenden Begriffe aus dem Bereich des Softwaretests erläutert werden. Die hier dargelegten Definitionen und Erklärungen basieren im Wesentlichen auf den folgenden Referenzen: [7] [16] [15] [25]

3.1 Definition Test

Für den Begriff „Test“ oder „Softwaretest“ gibt keine exakte und allgemeingültige Definition. Einige der bekanntesten Definitionen lauten:

- „the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspects of the system or component“ (ANSII/IEEE Std. 610.12-1990)
- „Test [...] der überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen“ (Ernst Denert)
- „Testing is a process of gathering information by making observations and comparing them to expectations“ (Dale Emery, Elisabeth Hendrickson)
- „A test is an experiment designed to reveal information, or answer a specific question, about the software or system.“ (Dale Emery, Elisabeth Hendrickson)

3.2 Definition Testprozess

Eine einheitliche Definition für den Terminus Testprozess existiert nicht. Eine einfache Beschreibung zu diesem Begriff könnte folgendermaßen aussehen. Der Testprozess legt fest, welche Testtätigkeiten im Rahmen der Qualitätssicherung eines Softwareprodukts durchgeführt werden, wie diese durchgeführt werden und wann diese zu tätigen sind. Leider ist festzustellen, dass in vielen Unternehmungen dieser Testprozess sehr „hemdsärmlig“ d. h. unsystematisch und lückenhaft statt findet [31].

3.3 Klassifikation

3.3.1 Blackbox/Whitebox und Greybox Tests

Blackbox Bei einem Blackbox-Test sind die Interna der IUT nicht bekannt, sondern nur deren Schnittstelle/Spezifikation. Der Tester weiß, was er sich von dem IUT erwartet und erstellt anhand davon Testfälle, welche die Implementierung treiben.

Der Vorteil von Blackbox-Testen ist, dass der Tester relativ wenig Programmiererfahrung benötigt, da er keine Programminterna verstehen können muss, sondern nur geeignete Eingaben für das IUT in den Testfällen erzeugen muss (mitunter auch ohne jeglichen Programmieraufwand).

Blackbox-Testen hat konträr aber den Nachteil, dass (prinzipiell) keine Aussagen über die erreichte Testabdeckung gemacht werden können. Aufgrund dessen sind oftmals wesentlich mehr Testfälle als bei den folgenden Techniken notwendig, um mit einer gewissen Wahrscheinlichkeit die gewünschte Testabdeckung zu erreichen.

Whitebox Beim Whitebox-Testen sieht sich der Tester das IUT vor dem Test sehr genau an. Er analysiert den Code und entwirft anhand dieser Analyse die Testfälle.

Zwar benötigt der Tester nun wesentlich bessere Programmierkenntnisse, um die Codeanalysen durchführen zu können, aber er kann nun zielorientierter Testfälle erstellen. Daher sind generell wesentlich weniger Testfälle notwendig. Ein erfahrener Tester kann zudem noch problematische Codestellen identifizieren und diese besonders genau testen.

Greybox Die Greybox-Tests sind ein Kompromiss zwischen Whitebox- und Blackbox-Tests. Anstatt die IUT genau zu analysieren bzw. dies vollständig zu vernachlässigen, werden bei dieser Art des Testens nur äußerst grobe Analysen durchgeführt, um eine möglichst große Testabdeckung zu erreichen.

Eine solche Analyse könnte z.B. darin bestehen, die Eingabeparameter aufzudecken, welche in Kontrollflusselementen in der Implementierung verwendet werden, und für diese die Werte zu finden, sodass alle Codepfade erreicht werden. Oder auch, dass man die Eintrittsbedingungen denen die Methoden unterliegt z.B.

```
1 void function(int i)
2 {
3   if(i < 0) throw new Exception();
4   ...
5 }
```

bei der Erstellung der Testfälle mit einfließen lässt und somit einen Teil der Testfälle nicht erstellt, da sie sich als nicht notwendig erweisen.

3.3.2 Formale und nicht-formale Tests

Formale Testmethoden beschäftigen sich mit der strukturierten Testfallgenerierung anhand von Spezifikationen. Der Vorteil der formalen Methoden besteht darin, dass durch ihre genau spezifizierte Erzeugungsstruktur gewisse Qualitätsmerkmale mit Sicherheit erreicht werden können. Nicht-formale Testmethoden hingegen bezeichnen Methoden, bei welchen ein Tester dafür zuständig ist, die Testfälle zu erstellen. Daher entscheidet alleine

die Vorgehensweise des Testers darüber, ob die gewünschten Qualitätsmerkmale durch die Testfälle sichergestellt werden. [8]

Formale Testmethoden bedürfen aufgrund der notwendigen Spezifikation des Systems eines Mehraufwands weshalb auch heute nicht-formale Methoden häufig noch bevorzugt werden. Es zeigt sich aber, dass mehr und mehr auf formale Methoden - zu welchen z.B. modellbasiertes Testen oder auch automatisiertes Whitebox-Testen gehören - immer mehr Anwendung finden.

3.3.3 Statische und dynamische Tests

Statische Tests Zu den statischen Testverfahren gehören jene Methoden, bei denen das IUT nicht ausgeführt wird. Es wird also nicht versucht, korrektes Systemverhalten durch Testfälle, welche die IUT antreiben sicherzustellen, sondern das IUT wird analysiert, um die Ursache von Fehlern zu finden.

Statische Tests können händisch (prinzipiell auch ohne Computer) oder maschinell unterstützt durchgeführt werden. Zu ersterem zählen z.B. Code Reviews während zu letzterem z.B. das Erstellen von Code Metriken gehört.

Der Vorteil der statischen Tests ist, dass kein lauffähiges System notwendig ist und somit diese Technik in jeder Phase der Entwicklung durchgeführt werden kann.

Problematisch ist, dass diese Technik nicht einfach einzusetzen ist. Um diese statischen Analysen durchzuführen, die Ergebnisse folgerichtig zu interpretieren und dann auch noch basierend darauf Fehler in Programmen aufzudecken, bedarf es sehr viel Erfahrung und Verständnis. Aufgrund dessen wird diese Technik in der Praxis oftmals unterschätzt/vernachlässigt bzw. überhaupt nicht als Mittel des Softwaretests angesehen.

Dynamische Tests Dynamische Tests sind jene Art von Tests, die wohl jeder Entwickler mit dem Begriff Testen verbindet. Bei den dynamischen Tests wird das IUT von Testfällen getrieben. D.h. es werden Testeingaben für das System erzeugt und dieses damit getestet. Die realen Ausgaben des Systems werden mit erwarteten Ausgaben (dem sogenannten Testorakel) verglichen. Ein Testfall ist als erfolgreich zu interpretieren, wenn reale Ausgabe und Testorakel äquivalent sind.

Der Vorteil an dieser Methodik ist, dass sie recht einfach umzusetzen und auch automatisierbar ist. Daher können einmal erstellte Tests immer wieder verwendet werden, um das Systemverhalten zu validieren. Des Weiteren sind die Tests sehr gut reproduzierbar. Zusätzlich wird das IUT in einer „realen“ Umgebung betrieben, was bewirkt, dass auch Fehler, welche nicht unbedingt mit der implementierten Logik zusammenhängen, aufgedeckt werden können.

Dynamische Tests bergen aber auch Risiken und Probleme. Oft wird z.B. die Aussagekraft eines erfolgreichen Testfalls überschätzt (aufgrund weniger positiver Testergebnisse wird angenommen, dass das IUT funktioniert). Diese Testart kann gewisse Aspekte der Qualitätssicherung einfach nicht bedienen. Z.b. kann mit diesen Tests nicht überprüft werden, ob die Systemperformance adäquat ist. Einen Testfall für ein IUT, welches die Zahl π genauer berechnet als jemals zuvor (es kann kein Testorakel bestimmt werden) kann einfach nicht erstellt werden. Dann muss wieder auf statische Verfahren zurückgegriffen werden (formaler Programmbeweis, etc.).

3.4 Testbereiche

3.4.1 Funktional/Nicht-Funktional

Der funktionale Test gehört zu den dynamischen Tests. Er hat die Aufgabe sicherzustellen, dass ein System seine funktionalen Anforderungen korrekt implementiert. Funktionale Anforderungen ergeben sich aus der Fragestellung, **was** ein System können muss. Z.B.: Das System muss Messwerte dauerhaft speichern können.

Funktionales Testen ist sicherlich einer der wichtigsten Testbereiche.

Konträr zu den funktionalen gibt es die nicht-funktionalen Testbereiche. Nicht funktionale Testanforderungen ergeben sich aus den funktionalen Anforderungen. Zu der oben genannten funktionalen Anforderung wäre eine nicht-funktionale Anforderung z.B.: Das Speichern von n Messwerten muss innerhalb einer angemessenen Zeitspanne abgeschlossen sein.

Nicht-funktionale Anforderungen sind oftmals sehr viel schwieriger zu testen als die funktionalen Anforderungen, da sich dafür oftmals nur schwer konkrete Testfälle generieren lassen. Oftmals können diese Anforderungen nur anhand von Systemmonitoring (Antwortzeiten) und/oder Extrapolation (Last/Stresstest) gemacht werden. Reproduzier- und Nachvollziehbarkeit sind oftmals schwer bis gar nicht zu realisieren.

3.4.2 Performance

Der Performancetest gehört zu den nicht-funktionalen Tests und ist ein dynamischer Test. Er soll nachweisen, dass ein System mit einer adäquaten Performance läuft. Der Term „adäquate Performance“ ist nicht ganz einfach definierbar. Im Grunde genommen ist dieser Begriff für jedes System, welches eine qualitativ angemessene Performance einhalten muss, eigens zu spezifizieren. Für Webseiten wurden teilweise Versuche unternommen, einheitliche Standards für den Bereich Performance zu etablieren. Diese konnten sich aber niemals wirklich durchsetzen. Dennoch gibt es bei Applikationen einige gebräuchliche Schlüsselindikatoren, mittels welcher die Performance-Anforderungen eines Systems definiert und validiert werden können.

- Antwortzeit
  Unter der Antwortzeit versteht man die Zeitspanne zwischen einer Anfrage und deren Antwort. Besonderen Stellenwert haben Antwortzeiten schon immer im Bereich von Web-Applikationen. Die Antwortzeit ist ein äußerst wichtiger Performanceindikator, da diese direkt vom Benutzer wahrgenommen wird. Die Antwortzeit eines Systems gehört zudem zu jenen Merkmalen, welche recht einfach gemessen werden können. Auch statistische extrapolierende Auswertungen können anhand dieser Kennzahl recht gut umgesetzt werden.

- (Daten-) Durchsatz
  Der Durchsatz bestimmt, wie viel ein System innerhalb einer gewissen Zeitspanne verarbeiten kann. Dabei können verschiedene Aspekte wie z.B. Datenmenge, Systemevents usw. gemeint sein.

- Auslastung
  Darunter versteht man wie sehr ein System / eine Applikation die ihr zur Verfügung gestellten Ressourcen (Arbeitsspeicher, Rechenkapazität, Speicher, Bandbreite, etc.) nutzt. Niemand würde eine EMail-Applikation, welche 100% der CPU-Rechenleistung benötigt, als adäquat bezüglich ihrer Performance-Qualität einstufen.

- Verfügbarkeit
  Unter Verfügbarkeit versteht man das Verhältnis von jener Zeit, zu der die Applikation verfügbar ist, zu jener, in der die Applikation nicht genutzt werden kann (Systemausfälle, Applikationsabstürze, Server-Ausfallszeiten).

Die Performance einer Applikation ist eigentlich sehr wichtig, da diese bei Benutzern eines Systems sehr oft als sehr wichtiger Qualitätsfaktor gesehen wird. Schlechte Performance kann sehr schnell zu einem getrübten Bild oder schlimmer noch zu Frust bei den Anwendern führen. Dennoch wird dem Performancetest bis heute nicht der Stellenwert eingeräumt, welchen er eigentlich verdient hätte. Zwar wird dem Performanctest mittlerweile seine Berechtigung eingeräumt, aber dennoch werden Performancetests meist viel zu spät und bei Weitem zu unstrukturiert durchgeführt, um jenen Mehrwert zu erbringen, welchen er könnte.

Vor allem er späte Durchführungszeitpunkt führt zu sehr großen Problemen, da eine schlechte Performance oftmals mit einer Neukonzeption und Neuimplementierung eines Moduls/-Subsystems einhergeht, sofern Performanceprobleme nicht nur auf schlampiger/schlechter Programmierung beruhen.

3.4.3 Interoperabilität

Interoperabilität bezeichnet die Fähigkeit, heterogener Systeme, miteinander zu arbeiten. Tests in diesem Bereich sollen nachweisen, dass diese Fähigkeit auch gegeben ist. Meist bestand der Interoperabilitätstest darin, zu zeigen, dass bestimmte Kommunikationsstan-

dards, über welche die Systeme verbunden wurden, von den jeweiligen Systemen eingehalten wurden.

Heutzutage sind Tests bezüglich der Interoperabilität bei Weitem nicht mehr so relevant wie noch vor einigen Jahren. Dies liegt vor allem an den mittlerweile sehr guten und etablierten Middlewares 2.3 welche aktuell eingesetzt werden bzw. werden sollten. Bei Einsatz einer Middleware hat der Hersteller der Middleware generell dafür Sorge zu tragen, dass die Interoperabilität mit Systemen, welche die gleichen Standards umsetzen (solche Standards sind z.B.: WSDL, SOAP, WS*) gewährleistet ist.

Probleme bezüglich der Interoperabilität, sind durch den Einsatz einer Middleware aber nicht automatisch passe. Zwar ist sichergestellt, dass die Implementierung an sich interoperabel ist, durch fehlerhafte Konfiguration oder falsche Handhabung der Middleware kann es aber noch immer zu Kommunikationsproblemen mit anderen Systemen kommen. In diesem Fall ist aber kein eigener Interoperabilitätstest notwendig. Viel mehr kann der Nachweis der Interoperabilität im Rahmen eines Systemtest vorgenommen werden, da im Fehlerfall keine vollständigen Fehleranalysen notwendig sind, sondern nur Konfigurationsfehler und Anwendungsfehler entdeckt werden müssen.

3.4.4 Sicherheit

Softwaresicherheit bezeichnet die Fähigkeit eines Systems, auch dann innerhalb seiner Anforderungen weiterzuarbeiten, wenn es attackiert wird. "Software security is the ability of software to provide required function when it is attacked" [37].

Weiters kann nach [37] der Prozess des Sicherheitstestens in zwei Bereiche eingeteilt werden. Den Bereich des funktionalen Sicherheitstests und jenen des Sicherheitslücken-Tests.

Der funktionale Sicherheitstest dient dem Nachweis der korrekten Funktionsweise von Funktionalitäten, welche mit der Sicherheit eines Systems zu tun haben. Eine solche Funktionalität stellt z.B. das Zurücksetzen eines Passworts oder das Festlegen einer Benutzergruppe für einen Benutzer dar.

Der Sicherheitslückentest hingegen soll Schwachstellen im System finden, welche aufgrund von Design-Fehlern, Implementierungsfehlern und dergleichen auftreten. In diesem Bereich des Sicherheitstests versucht der Tester die Rolle eines Systemangreifers einzunehmen. Er attackiert also gewollt das System und versucht auf diese Art und Weise die Schwachstellen des selbigen aufzudecken.

Tests bezüglich der Sicherheit werden bis dato kaum automatisiert ausgeführt [28]. Die Sicherheit eines Systems mittels Testen nachzuweisen ist ein äußerst schwieriges Unterfangen. Während die funktionalen Aspekte der Softwaresicherheit mitunter noch recht gut abgedeckt werden können, stellt vor allem das Auffinden von Sicherheitslücken ein großes Problem dar, welches bisher nicht zufriedenstellend gelöst werden konnte.

3.5 Teststufen

3.5.1 Unittest

Der Unittest ist die erste Stufe der Tests. Bei dieser Teststufe wird die Funktionalität einer einzigen, separiert betrachteten Komponente getestet. Die separierte Betrachtungsweise ist in der Praxis oftmals nicht einfach umzusetzen, da Module - sofern es sich nicht um grundlegende Basismodule handelt - beinahe immer auf die Interaktion mit anderen Modulen angewiesen sind. Daher ist es auf der Stufe der Unittests notwendig, auf Mocking Frameworks zurückzugreifen, um Tests tatsächlich auf der Stufe der Unittests durchführen zu können. Setzt man diese nicht ein (z.B. beim Test einer Datenbank-Zugriffs-Schicht, ohne die eigentliche Datenbank durch einen Mock zu ersetzen) wird im Falle der Tests auf die Implementierung nicht nur die Komponente sondern auch die Interaktion zwischen Komponente und Datenbank getestet, wodurch der Test eher der Klasse der Integrationstest zuzuordnen ist.

Mocks sind Pseudoimplementierungen die das erwartete Verhalten von Komponenten simulieren, welche von der eigentlich zu testenden Komponente benutzt werden.

Unittests werden generell hochautomatisiert durchgeführt. Dazu gibt es eine Vielzahl an Unittest-Frameworks wie beispielsweise NUnit, JUnit und viele weitere. Diese hohe Automatisierung macht sie besonders wichtig im Bereich des Regressionstests.

Unittests sollten idealerweise als Blackbox-Tests konzipiert sein. Wenn Tests anhand von Codekenntnissen erstellt werden, werden sie anfällig dafür, bei Codeänderungen fehlerhaft zu werden. Das heißt, dass die Testfälle aufgrund von getroffenen Annahmen ihre Allgemeingültigkeit bezüglich der Spezifikation verlieren.

In der Praxis werden sie aber dennoch oft als Whitebox- oder Greyboxtests realisiert, da der Aufwand für Unittests ansonsten häufig bei Weitem zu hoch ist. Durch Kenntnisse der Modulinterna kann die Anzahl der notwendigen Eingabeparameter - und somit die Anzahl der Testfälle - deutlich reduziert werden, wobei weiterhin sichergestellt ist, dass das Modul ausreichend getestet wurde.

3.5.2 Integrationstest

Der Integrationtest hat zum Ziel, zu zeigen, dass bestimmte Systemmodule auf korrekte Art und Weise zusammenarbeiten.

Die Anzahl der notwendigen Testfälle in dieser Stufe steigt rapide mit der Zahl der vorhandenen Komponenten an. Aus diesem Grund werden Komponenten meist zusammengefasst und diese dann wieder als Komponente betrachtet. Durch diese Maßnahme wird eine Vereinfachung des Systems erreicht, welche den Testaufwand drastisch reduziert.

Der Integrationstest ist ebenfalls sehr gut automatisierbar und wird oftmals durch die selben Frameworks wie auch der Unittest realisiert.

Für das Erstellen der Integrationstests gibt es verschiedene Strategien. All diese Strategien orientieren sich bis zu einem gewissen Grad an einer Einteilung der zu testenden Komponenten in Schichten, wobei die unterste Schicht jene ist, in welcher Komponenten einzeln existieren, und die oberste Schicht das Gesamtsystem darstellt.

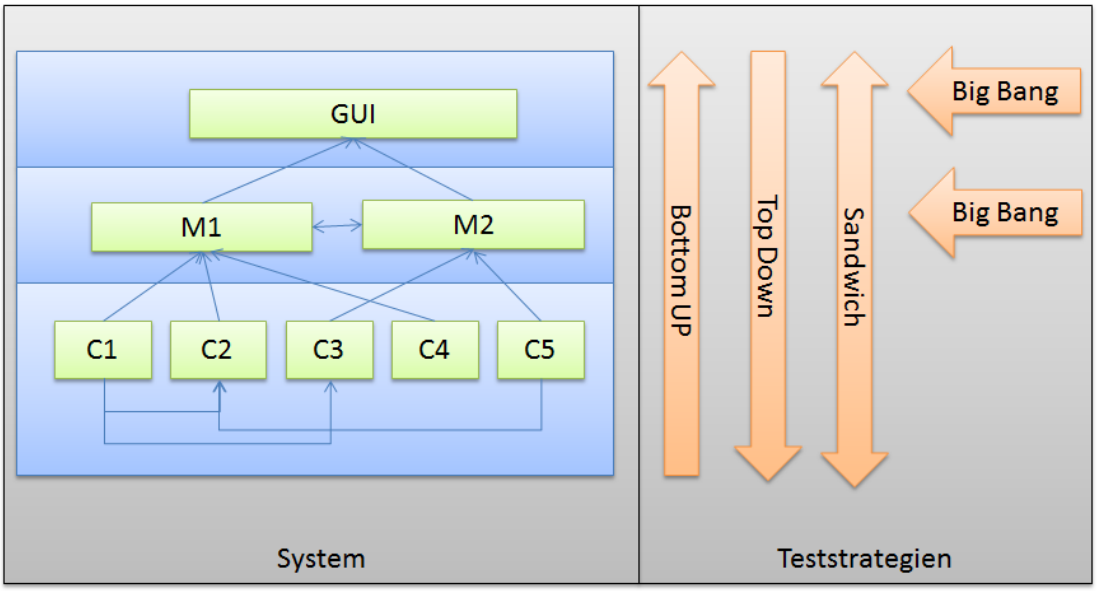

Abbildung 6: Integrationsteststrategien

- Bottom Up
  Bei der Bottom Up-Strategie wird zuerst die unterste Schicht im System betrachtet. Konnten die Integrationstests für diese Stufen erfolgreich abgeschlossen werden, werden die Komponenten dieser Schicht in der nächsten Schicht wiederum zu Komponenten zusammengefasst, für welche dann die Interaktion getestet wird. Da die nächst höhere Schicht erst dann getestet wird, wenn die Komponenten der unteren Schichten getestet sind und diese somit auch für die folgenden Schichten verfügbar sind, ist bei dieser Strategie keine Entwicklung von Mocks notwendig.

  Diese Strategie versucht von einer Ebene welche, sehr nahe bei den Unittests angesiedelt ist, sich auf eine Ebene, welche dem Systemtest sehr nahe ist, hochzuarbeiten.

  Der Vorteil dieser Vorgehensweise ist, dass Fehler in jener Ebene aufgedeckt werden, in welcher sie auch wirklich auftreten und daher die Fehlerlokalisierung recht einfach ist.

  Der Nachteil hingegen ist, dass Fehler, welche in den oberen Schichten entdeckt werden, mitunter Änderungen in den Komponenten unterer Schichten nach sich ziehen und somit einen Neubeginn der Integrationstests dieser Schichten zur Folge haben.

Aufgrund dessen, dass die nächste Schicht erst dann getestet werden kann, wenn die darunterliegenden Schichten getestet wurden, kann es dazu kommen, dass Fehler in den oberen Schichten erst sehr spät erkannt werden. Zusammen mit dem zuvor genannten Aspekt ist es leicht ersichtlich, dass sich hier sehr große Probleme bei bestimmten Projekten ergeben können.

- Top Down

  Top Down ist der konträre Ansatz zur Bottom Up Strategie. Hier wird zuerst der Integrationstest für die oberste Schicht durchgeführt. Komponenten aus den unteren Schichten, welche noch nicht vorhanden bzw. getestet sind, werden als Mocks realisiert.

  Dieser Ansatz hat den Vorteil, dass Probleme aus den oberen Schichten nun früher aufgedeckt werden. Fehler, die nun in Änderungen in den unteren Schichten resultieren, können ausgebessert werden, bevor diese überhaupt getestet wurden. Dies kann sich in einer Verringerung des Testaufwands manifestieren.

  Der Nachteil dieses Ansatzes ist, dass eine Entwicklung von Mocks notwendig ist, welche mitunter einen nicht unerheblichen Aufwand darstellt und zusätzlich können Fehler in eben diesen zu falschen Testresultaten und damit zu weiteren zusätzlichen Aufwänden führen.

- Sandwich

  Die Sandwich-Methode stellt eine Kombination aus der Top Down und der Bottom Up Strategie dar. Durch die Kombination sollen die Vorteile beider Methoden herausgehoben und die Nachteile minimiert werden.

  Eine Kombination aus beiden Verfahren könnte so aussehen, dass sobald ein Fehler in der obersten Schicht eines Systems gefunden wird - wenn das Top Down-Verfahren verwendet wird - nicht starr bis zur untersten Ebene weiter verfahren wird, sondern direkt ab diesem Zeitpunkt der Bottom Up Ansatz für diesen Teil des Systems eingesetzt wird, wenn der Fehler darauf schließen lässt, dass er aus Fehlern in den unteren Schichten resultiert.

  Ein weiterer Ansatz könnte z.B. das vorzeitige Abbrechen des Top Down-Verfahrens sein. Wenn in den oberen Schichten eines Systems keine Fehlerfälle auftreten, ist es mitunter nicht notwendig, darunterliegende Schichten weiter zu testen.

- Big Bang

  Bei der Big Bang-Strategie werden **alle** Komponenten zusammen als ein einziges System getestet. Sie ist somit beinahe schon einem Systemtest zuzuordnen. Diese Methode sollte nur dann Anwendung finden, wenn davon ausgegangen werden kann, dass das System bereits fehlerfrei ist und dies nur mehr nachgewiesen werden soll. Das heißt, es wird beim Integrationstest eine sehr optimistische Annahme über das System getroffen und der Integrationstest soll dazu dienen, diese Annahme zu untermauern.

Der Vorteil dieser Methode ist der deutlich reduzierte Testaufwand, welcher daraus resultiert, dass im Grunde nur mehr eine einzelne Schicht des Systems getestet wird und die Implementierung von Mocks gänzlich entfällt.

Der große Nachteil an dieser Methode liegt darin, dass eine Fehlerlokalisierung allerdings äußerst schwierig ist.

3.5.3 Systemtest

Der Systemtest ist der finale Test, wenn alle Komponenten zu einen gesamtem System zusammengefügt wurden. Der Systemtest dient dazu, Fehler bezüglich der funktionalen und nicht-funktionalen Anforderungen aufzudecken. Der Systemtest stellt zweifelsohne die wichtigste Teststufe dar. Eine ausgiebige Beschreibung dieses Teilgebiets im Thema Softwaretest würde den Rahmen dieser Arbeit bei Weitem sprengen. Daher sei dem interessierten Leser an dieser Stelle das Buch [33] nahegelegt, welches diese Teststufe sehr genau betrachtet.

3.5.4 Abnahmetest

Der Abnahmetest ist die finale Teststufe und folgt generell auf den Systemtest. Der Abnahmetest wird meist von den Kunden bzw. Auftraggebern eines Softwaresystems durchgeführt. Grundlage für diese Tests ist ein Lastenheft und eine Anforderungsspezifikation.

Der Abnahmetest kann eher dem Gebiet des Projektmanagements denn der eigentlichen Entwicklung eines Softwareprojekts zugeordnet werden, da dieser Test weit mehr der rechtlichen Absicherung dient, als der Erhöhung der Qualität eines Softwareprodukts. Da diese Arbeit ihren Fokus primär auf die entwicklerischen Tätigkeiten von Software legt, wird dieses Thema an dieser Stelle nicht weiter ausgeführt und sei nur der Vollständigkeit halber erwähnt.

3.6 Modellbasiertes Testen

3.6.1 Motivation

Um ein System zu testen, gibt es die unterschiedlichsten Möglichkeiten. Robinson zum Beispiel definiert die folgenden Systematiken [29] bzw. [40]:

- Unsystematisches Testen
  Der Tester schreibt manuell Testfälle, mit welchen er versucht, Fehler aufzudecken bzw. nachzuweisen, dass die die Implementierung die gegebenen Spezifikationen erfüllt. Dieser Ansatz ist mit einigen Problemen behaftet. Zum einen birgt jegliche

Änderung an den Spezifikationen die Gefahr, dass diese wiederum umfassende Änderungen an den bereits erstellten Tests mit sich bringt. Des Weiteren ist es bei dieser Vorgehensweise oftmals nicht möglich zu sagen, ob das System ausreichend getestet wurde.

- Testen mit Skripten
  Dieser Ansatz versucht, mithilfe von Skripten Testfälle zu generieren, auszuführen und zu validieren. Der Vorteil der vorhergehenden Methode ist, dass mit einem Skript eine Vielzahl an Testfällen erstellt werden kann, wobei der Aufwand für das Erstellen dieses Skripts wesentlich geringer ist als jener für das Erstellen der eigentlichen Testfälle.

  Dennoch leidet dieser Ansatz ebenfalls zu einen gewissem (reduzierten) Grad unter den zuvor genannten Problemen. Zwar muss nicht jeder einzelne Testfall bei Spezifikationsänderungen angepasst werden, dennoch müssen die Skripte, welche zur Testgenerierung gewartet werden. Die Skripte an sich können selbst wiederum fehlerhaft sein bzw. recht komplex werden, was wiederum Probleme für die Qualitätssicherung mit sich bringen kann.

Die beiden zuvor genannten Testansätze gehören zu relativ bekannten Testtechniken. Unit-Tests z.B. können meist dieser Kategorie zugeordnet werden.

Modelle sind in vielen Disziplinen (z.B.: Elektrotechnik, Maschinenbau, Regelungstechnik usw.) schon recht lange eine etablierte Technik (seit ca. den 60ern und 70ern) welche dort auch schon längere Zeit zum Test der realen Systeme, welche hinter diesen Modellen stehen eingesetzt werden. In der Softwareentwicklung hingegen hat sich die Beschreibung mittels Modellen erst in den 90-ern (UML) etabliert, und der Softwaretest, basierend auf diesen Modellen ist erst seit einigen Jahren zu einem der Schlagworte im Bereich der Qualitätssicherung für Softwareprojekte geworden.

Die grundlegende Intention des modellbasierten Testens ist es, ein abstraktes Modell einer „Implementation Unter Test“ (IUT) zu erstellen und basierend auf diesem Modell Testfälle, welche aus Eingaben (Test-Sequenzen) und erwarteten Ausgaben (Test-Orakel) bestehen, zu erstellen.

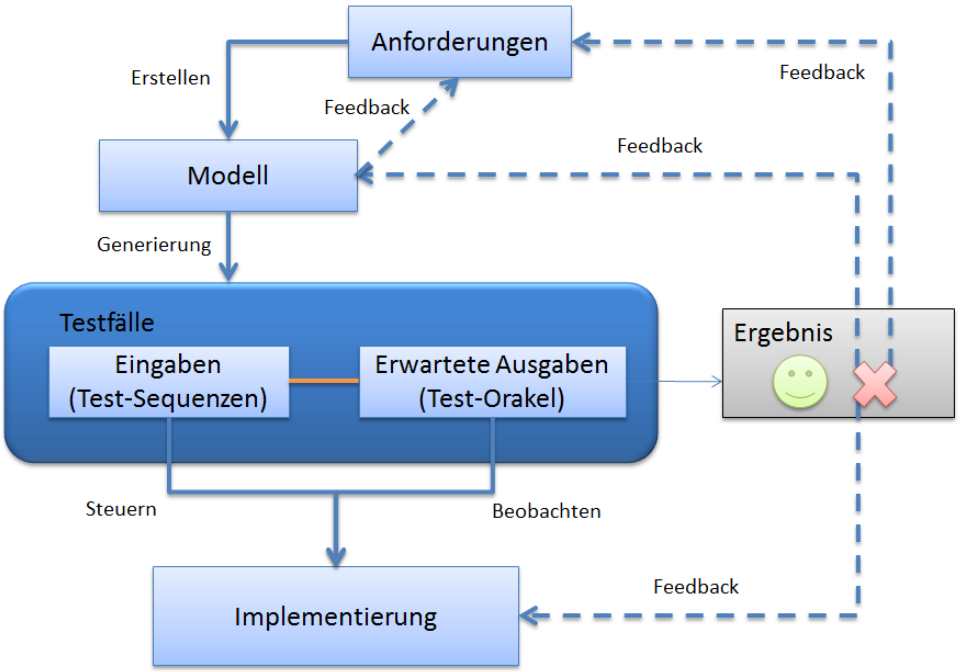

Abbildung 7: Modellbasiertes Testen Prozess

Die Ausgangsinformation für den modellbasierten Test stellen die Anforderungen des Systems dar. Unter den Anforderungen versteht man im Bereich des modellbasierten Tests primär die funktionalen Anforderungen. Weitere nicht-funktionale Aspekte wie die Systemsicherheit, Lastverhalten usw. stellen beim MBT eine untergeordnete Rolle dar. Das bedeutet aber nicht, dass diese von MBT nicht - zumindest bis zu einem gewissen Grad - behandelt werden können.

Anhand der funktionalen Anforderungen wird dann ein Modell gebildet. Das Modell wird in 3.6.2 genauer behandelt. Das Modell dient dann als Ausgangslage für die Testgenerierung. MBT-Tools können basierend auf dem abstrakten Modell automatisiert Testfälle erstellen. Dieser Punkt wird in 3.6.4 detailliert geschildert. Die Testfälle treiben/steuern dann die Implementierung. Die Ausgaben/Resultate, welche die Implementierung liefert, werden mit den Test-Orakeln verglichen und schlussendlich kommt so ein Endergebnis zustande, anhand welchem dann bestimmt wird, ob das System innerhalb der Spezifikationen arbeitet.

Grundsätzlich könnte man nun meinen, dass dieser Ansatz sehr ähnlich dem zuvor genannten Skript-basierten Testen ist. Zu einem gewissen Grad ist dieser Vergleich auch durchaus angemessen, denn in beiden Fällen wird eine Mechanik eingeführt, die das automatisierte Erstellen von Testfällen ermöglicht und in beiden Fällen ist eine Wartung der selbigen bei variierender Systemspezifikation notwendig. Dennoch besitzt MBT gewisse Vorteile gegenüber dem Skript-basierten Testen.

- Modelle dienen als Basis für das **systematische** Erstellen von Testfällen

- Das Modell bietet zusätzliche Information über das zu testende System und stellt somit ein Diskussionsgrundlage für selbiges dar
- Ein Modell ist im Gegensatz zu einem Testerzeugungsskript eine Abstraktion des Systems und somit wesentlich einfacher gegenüber Änderungen in der Spezifikation anpassbar
- Modelle sind eine generalisierte Darstellungsform und können somit als Ausgangsbasis für weiter (Test-) Techniken eingesetzt werden.

Der wohl wichtigste Punkt, welcher für das modellbasierte Testen spricht, ist die Erwartung, den Testaufwand mithilfe dieser Technik bis zu einem gewissen Maß senken zu können. Fallstudien wie [12] und [14] zeigen, dass diese Erwartungen mittlerweile durchaus erfüllt werden können, und dass obwohl modellbasiertes Testen und vor allem der reale Einsatz dieser Technik noch am Anfang steht.

Objektive Zahlen, die den Vorteil von MBT auch bewertbar machen, gibt es leider kaum. Jedoch hat Microsoft einige Zahlen bezüglich dem Vergleich zwischen modellbasiertem Testen und klassischen Testansätzen erhoben.

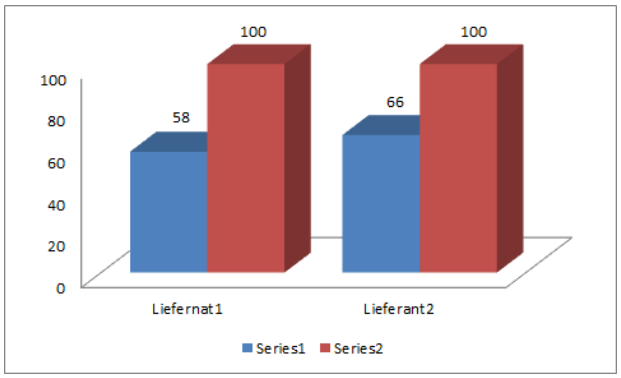

Abbildung 8: Vergleich Testaufwand von MBT zu traditionellem Testen

Die Grafik zeigt das Verhältnis des Aufwands von MBT-basiertem Testen (blau) zu traditionellen Testtechniken (rot). Bei dieser Studie wurden zwei Zulieferer herangezogen. Aufgabe der Zulieferer war, die Implementierung des SMB2 (Server Message Block) Protokolls. Es zeigt sich, dass durch Einsatz modellbasierter Techniken eine durchschnittliche Reduzierung von 38% zum traditionellen Testaufwand erreicht werden konnte.

3.6.2 Modell

Für den Begriff Modell gibt es eine Vielzahl an Definitionen, hier einige Beispiele [44] (zit.n. [35] [18] [1]):

- Ein Modell ist eine **abstrakte** Repräsentation von Struktur, Funktion oder Verhalten eines Systems

- Ein Modell beschreibt einen Aspekt eines konkret zu entwickelnden Systems
- A model of a system is a description of that system and its environment for some certain purpose

Oft werden Modelle im allgemeinen Verständnis mit grafischen Repräsentationen gleichgesetzt. Der Grund dafür liegt wohl darin, dass Modelle in der Praxis am häufigsten in dieser Form repräsentiert werden. Im Bereich der Softwareentwicklung stellt die wohl bekannteste Methode die grafische Modellierungssprache "Unified Modeling Language" kurz UML genannt, dar. Modelle müssen aber keineswegs eine grafische Repräsentationen darstellen. Modelle können selbst auch wieder mittels Programmcode dargestellt werden wie in 5 gezeigt wird.

Ein sehr wichtiges Merkmal von Modellen ist, dass sie eine **abstrakte** Repräsentation des Systems darstellen. Das heißt, ein Modell muss wesentlich einfacher als das eigentliche System sein.

"In terms of model-based testing, the necessity to validate the model implies that the model must be simpler than the SUT, or at least easier to check, modify and maintain. Otherwise, the efforts of validating the model would equal the efforts of validating the SUT." (Anm. d. Verf.: SUT steht für System Under Test) [40]

Übernommen aus [2] (zit.n. [22] [17]) ergibt sich damit weiters: "Ein Ziel eines Modellierers ist generell die Reduzierung der Komplexität des Modells gegenüber der Realität. Häufiger Trugschluss ist daher jeder Versuch der Gleichsetzung des Modells mit der Realität durch den Modellierer. Tatsächlich kann lediglich der Modellkontext bestimmt und optimiert werden. Damit wird die Zweckbindung des Modells bestimmt. Weiter kann das Modell hinsichtlich der Komplexität variiert werden. Im Grundsatz bleibt das Modell in allen Merkmalen außer der Verständlichkeit immer hinter der Realität zurück."

3.6.3 Modellprogramme

Modellprogramme sind eine spezielle Ausprägungsform eines Modells, welches z.B. in SpecExplorer Anwendung findet.

Ein Modell Programm $\mathcal{P}$ besteht aus folgenden Komponenten:

- Einem Set von Aktionen $\mathcal{M}$ wobei die Aktion als $m(v)/w$ definiert ist. v sind hier die Eingabeparameter, w die Ausgabeparameter
- Einem Set von Zustandsvariablen $\mathcal{V}$

Ein Zustand des Modellprogramms $\mathcal{P}$ ist durch die Werte der Zustandsvariablen festgelegt.

Wird nun in einem Zustand $\mathbf{s}$ die Aktion $m \in \mathcal{M}$ ausgeführt, sodass sich die Werte einer der Zustandsvariablen ändert, ist ein Folgezustand $\mathbf{t}$ von $\mathbf{s}$ erreicht.

Beispiel für ein einfaches Modellprogramm in SpecExplorer mit den bisher genannten Modellprogrammelementen:

```
1
2  class Model
3  {
4  #region State
5
6    int value = 0;        //Zustandsvariable
7
8  #endregion
9
10 #region Actions
11
12   void Foo()
13   {
14     value = value + 1; //Aktion
15   }
16
17 #endregion
18 }
```

Zusätzlich können zu den Aktionen noch Bedingungen $Pre_m[x]$ angegeben werden, wobei $\mathbf{x}$ die formalen Eingabeparameter von $m \in \mathcal{M}$ sind, unter welchen diese angewendet werden dürfen.

```
1
2    void Foo()
3    {
4      Condition.IsTrue(value < 10);
5      value = value + 1;
6    }
```

Warum Vorbedingungen unverzichtbar sind, wird in 5 gezeigt.

3.6.4 Endliche Zustandsautomaten

Ein **deklaratives** Modell (wie z.B. ein Modellprogramm) alleine reicht natürlich noch nicht aus, um systematisch Testfälle und die dazugehörigen Orakel für das System in Form von Testcode zu generieren.

Je nach Art des ursprünglichen Modells, muss dieses weiter verarbeitet werden. Bei dieser Verarbeitung geht es im Wesentlichen darum, eine deklarative Modellbeschreibung zu einer ablaufbasierten Beschreibung (wie z.B. einer Zustandsmaschine) zu transformieren. Das Ergebnis dieses Vorgangs ist aber wiederum ein Modell, welches aber einen anderen Aspekt des Systems beachtet. Für diese Verarbeitung gibt es verschiedene Paradigmen, von denen einige wären:

- Endliche Zustandsmaschinen
- Anfangs-/Endbedingungen
- Label Transition Systems

- Zeitliche Automaten
- Abstraktes Datentypen-Testen
- ...

Im Rahmen dieser Arbeit wird im Besonderen auf endliche Zustandsautomaten eingegangen, da diese zu den in der Praxis am häufigsten eingesetzten Verfahren gehören. Der endliche Zustandsautomat selbst ist wiederum ein abstraktes Modell des eigentlichen Systems, jedoch nun aus einer ablauforientierten Sichtweise. Der Zustandsautomat wird durch automatisierte Analyse des Ursprungsmodells generiert, dieser Vorgang wird Exploration genannt.

Eine deterministische Zustandsmaschine besteht aus folgenden Komponenten

- $\sum$ ist das Eingabealphabet (endliche, nicht leere Menge von Symbolen)
- Einer Menge von Zuständen **S**
- Einem Initial-Zustand $s_0 \in S$
- Der Transitionsfunktion δ deren Definition $\delta : S \times \sum \rightarrow S$ lautet
- Den Endzuständen **F**

Eine Zustandsmaschine ist deterministisch, wenn für für jeden Zustand $s \in S$ und für jede Eingabe $e \in \sum$ gilt, dass es **höchstens einen** Folgezustand **t** gibt.

Ziel der Exploration ist es, nun aus dem in 3.6.3 gezeigten Modellprogramm eine endliche Zustandsmaschine zu konstruieren, welche dann in der Lage ist, automatisiert Testfälle zu erstellen/exekutieren.

Exploration Die Exploration beginnt bei den initialen Zuständen s_0. In diesen Zuständen wird geprüft, welche Aktionen erfüllbar sind und diese werden dann angewendet. Eine Aktion ist erfüllbar, wenn gilt:

- Vorbedinung $Pre_m\,[v]\,|v \in \mathcal{V}$ ist wahr
- Der Aufruf der Aktion m(v) resultiert in dem Ausgabeparametern w.

Wenn die beiden genannten Bedingungen zutreffen, resultiert der Aufruf der Aktion in einem neuen Folgezustand. Die Prozedur wird für alle resultierenden Zustände wieder ausgeführt.

Die Methodik der Exploration kann man sehr gut an einem Beispiel verdeutlichen. Als Beispiel soll das Modellproramm aus 3.6.3 dienen.

Der Initialzustand s_0 lautet

```
1  value = 0
```

Die Vorbedingung **Condition.IsTrue(value < 10)** ist erfüllt, damit kann die Aktion **Foo** ausgeführt werden, da die Aktion zusätzlich noch den Ausgabeparameter w mit **value = value + 1** inne hat. Nach dem ersten Aufruf der Aktion **Foo** lautet der Zustand des Modells

```
value = 1
```

Sobald der Zustand

```
value = 10
```

erreicht ist, ist die Vorbedingung nicht mehr erfüllt, und somit ist keine Aktion mehr ausführbar, der Explorationsvorgang ist damit beendet und alle gültigen Zustände für das Modell wurden erstellt.

Lässt man die Exploration auf das Modellprogrammm ohne Eingangsbedingung laufen, kann die Exploration nicht beendet werden. Es wurde keine explizite Vorbedingung definiert, somit ist die Vorbedingung für die Aktion **Foo** jenes Modeprogramms stets erfüllt. Daher kann die Exploration unendlich viele Zustände generieren. Dieser Umstand führt zum sogenannten Problem der Zustandsexplosion. Der hier gezeigte Fall ist die extremste Form der Zustandsexplosion, nämlich dass unendlich viele Zustände generiert werden [3]. Selbst für einfache und triviale Programme tritt eine Zustandsexplosion sehr schnell auf. In realen Systemen ist eine Zustandsexplosion de facto unumgänglich. Aufgrund dessen ist modellbasiertes Testen ein nicht triviales Unterfangen. Ein Großteil der Tätigkeit während des modellbasierten Testens besteht darin, durch geeignete Techniken diese Zustandsexplosion zu verhindern. Im Golgenden sollen diese Techniken kurz aufgezeigt werden.

- Parameter Selektion
- Methoden Restriktion
- Zustands-Filterung
- Gerichtete Suche
- Zustands-Gruppierung

Auf die formale Definition dieser Methoden wird an dieser Stelle verzichtet. Der interessierte Leser kann diese jedoch in [41] nachlesen. Die praktische Anwendung dieser Techniken werden in 5 weitergeführt.

Dieses Kapitel hat sich mit den grundlegenden theoretischen Aspekten des Softwaretests und abschließend noch im Speziellen dem theoretischen Aspekten des Modellbasierten Tests gewidmet. Damit ist der theoretische Teil der Arbeit welche als Vorbereitung für den praktischen Teil angesehen werden kann, abgeschlossen.

[3] In diesem Fall werden aufgrund technischer Gegebenheiten nicht unendlich viele Zustände generiert, sondern nur so viele, dass ein Integer Überlauf erreicht wird, und somit der Zustand s_0 eintritt.

4 NTE.CLM

4.1 Zielsetzung

Ziel des Projektes ist die Entwicklung einer Plattform, welche für die Verwaltung von verbauten Anlagen (primär Solaranlagen) auf sowohl zentralisierte als auch dezentralisierte Art und Weise ermöglicht. Solche Systeme werden im Allgemeinen als Supervisory Control and Data Aquisition (SCADA) Systeme bezeichnet.

Die Verwaltung einer Anlage besteht aus folgenden Merkmalen:

- Überwachung
  Überwachung bedeutet in diesem Projektkontext das Abrufen aktueller Anlagenwerte und Verarbeitung der selbigen anhand zuvor bestimmter Systemeinstellungen.
- Historie erfassen
  Die erfassten Datenwerte müssen in einem Datenhaltungssystem dauerhaft hinterlegt werden, sodass eine mitunter im Nachhinein anfallende neue Auswertung des Anlageverhaltens bzw. der Nachweis eines bestimmten Anlagenverhaltens erbracht werden kann.
- Visualisierung
  Die erfassten Daten (sowohl historische als auch aktuelle Daten) müssen in einer Form dargestellt werden, sodass es für die Systemnutzer möglich ist, Aussagen über das Anlagenverhalten bzw. den Anlagenzustand zu treffen.
- Steuerung
  Das Verhalten der Anlage muss mittels Setzen von Steuervariablen, welche sich auf der Steuereinheit (siehe 4.2.3) befinden, variierbar sein. Als Variieren des Anlagenverhaltens ist z.B. das Abschalten der selbigen oder das Setzen einer Steuervariable im Regelkreis der Anlage zu verstehen.

4.2 Systembeschreibung

In einer sehr vereinfachten Darstellung ist das NTE.CLM-System eine Client-Server basierte Plattform.

Abbildung 9: Architektur auf höchster Abstraktionsstufe

Die obige Grafik soll zeigen, dass das Kommunikationsprinzip auf der obersten Ebene noch immer auf dem sehr weitläufig verwendeten und etablierten Client-Server Prinzip beruht.

Die reale Systemstruktur ist jedoch wesentlich komplexer und wird in den folgenden Punkten im Detail erklärt.

4.2.1 Client

Der Client ist die Schnittstelle des Benutzers zur CLM-Plattform. Der Client visualisiert die vom Server bereitgestellten Daten und leitet Benutzereingaben an diesen weiter.

Die Plattform wird von verschiedenen Client-Anwendungen angesprochen. Einerseits gibt es eine Desktop-Applikation, welche auf der Windows Presentation Foundation (WPF) basiert und für die komplexen Aufgaben im System gedacht ist. Unter diese komplexen Aufgaben fällt zum Beispiel das Konfigurieren von Diagrammen, welche angezeigt werden können, sowie das Erstellen von Anlagenschemata, welche die Anlage visuell darstellen.

Andererseits gibt es noch eine Webbrowser-basierte Applikation, welche für jene Aufgaben gedacht ist, welche nicht eine so große Interaktion mit dem Benutzer erfordern. Zu diesen Aufgaben gehören Beispielsweise das Anzeigen der zuvor erwähnten Diagramme oder auch das Anzeigen des Anlagenstandorts auf einer Landkarte. Diese Applikation wurde mit Microsoft Silverlight ®realisiert.

4.2.2 Server

Der Server ist kein Server im eigentlichen Sinne. Stattdessen ist der Server ein Rechnernetz (Applikationscluster/Verteilte Anwendung), welches für systemexterne Entitäten als einzelnes System auftritt (dies entspricht genau der Definition wie in 2.1), welches seine Funktionalitäten als Web-Services zur Verfügung stellt.

Der Applikationsserver besteht aus mehreren Komponenten von denen jede ihre eigene spezielle Aufgabe übernimmt. Zwischen den Komponenten wird über Nachrichtenaustausch kommuniziert. Über die Systemgrenzen hinweg wird mittels Nachrichten mit den diversen Clients kommuniziert.

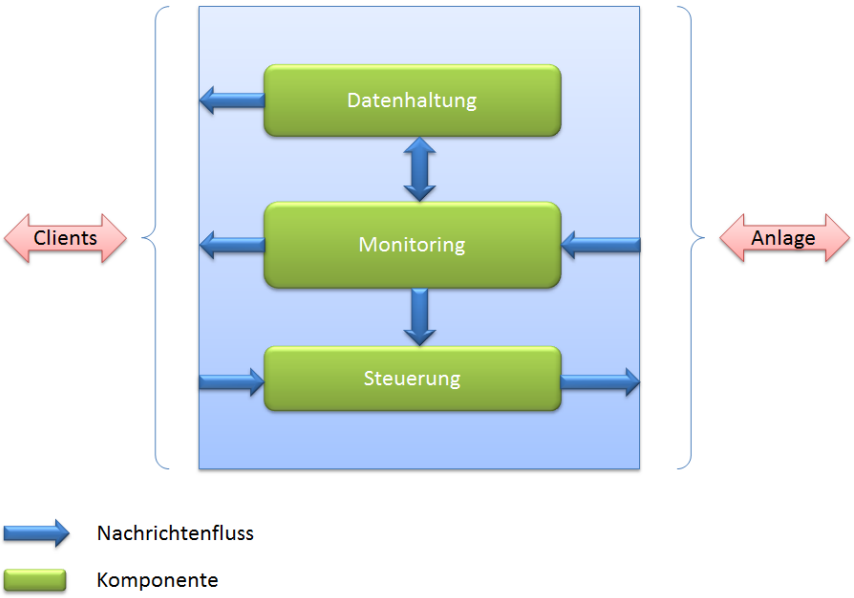

Abbildung 10: Grundlegende Serverarchitektur

4.2.3 Anlage

Die Anlage - genauer gesagt deren vorgeschaltetes Steuergerät (z.B. eine SPS) - ist die zweite Art von Client, welche im System existiert. Dieser Client ist für die Bereitstellung der Sensorwerte als Datenpunkte und das Umsetzen der Steuerbefehle, welche vom Server erteilt werden, zuständig.

Trotz der eingeschränkten Hardware, sind diese Steuergeräte in der Lage Web-Services anzusprechen. Daher ist kein eigener Kommunikationslayer notwendig, um mit diesen zu kommunizieren.

4.3 Projektanforderungen

Im Kurzen sollen folgend die wichtigsten Systemanforderungen des NTE.CLM-Projekts skizziert werden. Die folgende Auflistung stellt nur einen rudimentären Überblick dar, und soll dabei helfen, einen besseren Eindruck über das Projekt und insbesondere die wichtigen Aspekte bezüglich des Softwaretests zu bekommen.

4.3.1 Zuverlässigkeit

Eine sehr hohe Zuverlässigkeit ist für das NTE.CLM-System unerlässlich, da dieses zur Steuerung von Anlagen verwendet wird. Ein Ausfall des Systems könnte schwerwiegende Folgen nach sich ziehen. Zum einen können durch Ausfälle, während Steuerkommandos durch das System umgesetzt werden Beschädigungen an der Anlage auftreten. Weiters könnte durch längerfristige Ausfälle die Steuerungsfähigkeit des Systems nicht mehr wahrgenommen werden, was sehr schnell Auswirkungen auf die Vertrauenswürdigkeit des NTE.CLM-Systems nach sich ziehen würde. Weiters wäre mit einem sehr schnellen Schwinden der Kundenzufriedenheit zu rechnen. Wie bei den meisten Online-Systemen gilt, dass das System möglichst geringe Ausfallszeiten (meist kleiner 0.01%) einhalten muss.

4.3.2 Sicherheit

Sicherheit stellt für alle gängigen Web-Plattformen, auf welche von vielen Benutzern zugegriffen wird einen unverzichtbaren Leitgedanken dar. Auch für NTE.CLM ist diese Thematik ein sehr wichtiger Aspekt.

Anlagendaten werden im Allgemeinen als hoch sensible Daten angesehen, welche keinesfalls von Personen ohne die notwendige Autorisierung eingesehen werden dürfen. Das hat zur Folge, dass das NTE.CLM System Mandantenfähigkeit gewährleisten muss. Als mandantenfähig wird ein System bezeichnet, wenn dieses in der Lage ist, die Daten mehrerer Mandanten (Kunden bzw. Auftraggeber) zu verwalten, ohne dass diese die Daten der andren Mandanten einsehen können.

4.3.3 Performance

Performance ist für webbasierte Projekte ein weiteres sehr wichtiges Kriterium. Vor allem die hohe Anzahl an gleichzeitig im System aktiven Benutzern und die daraus resultierende Datenlast stellen für diese Art von Softwareprojekten immer wieder eine große Herausforderung dar.

Für die NTE.CLM-Plattform ist der Aspekt der Performance in zwei Bereichen besonders essentiell.

1. Datensammlung
   Ausreichende Performance im Bereich der Datensammlung stellt für die gesamte NTE.CLM-Plattform eines der wichtigsten Qualitätskriterien dar. Der Grund dafür liegt darin, dass das korrekte Systemverhalten der Plattform nicht gewährleistet werden kann, wenn die Plattform nicht in der Lage ist, die Daten genügend schnell abzurufen und zu speichern.

2. Datenbereitstellung
   Die Daten in einer angemessenen Zeitspanne auf den Client zu übertragen ist ebenfalls sehr wichtig, da angemessene Antwortzeiten unumgänglich sind, um eine ausreichende Kundenakzeptanz zu erlangen.

4.3.4 Nachvollziehbarkeit

Nachvollziehbarkeit bezeichnet im Kontext der NTE.CLM-Plattform, dass zu jedem beliebigen Zeitpunkt nachgewiesen werden kann, welche Aktionen in einem System zuvor getätigt wurden. Da die Plattform auch verwendet werden kann, um Anlagen zu steuern, kann es durch fehlerhafte Steuermaßnahmen zur Beschädigung oder Fehlverhalten und mitunter zu schweren Folgeschäden kommen.

Deshalb muss es einerseits für den Kunden möglich sein, nachzuvollziehen, welche Steuer-Aktionen von seinen Benutzern getätigt wurden, um diese zu kontrollieren und zu überprüfen. Zum anderen muss es möglich sein, nachzuweisen, dass das Fehlverhalten der Anlage nicht aus einem Fehlverhalten der NTE.CLM-Plattform, sondern durch fehlerhafte Steuerbefehle verursacht wurde.

Daher muss die Plattform mit entsprechenden Diagnose und Aufzeichnungsmechanismen ausgestattet sein.

4.4 Automatische Testfallerstellung

4.4.1 Problemstellung

Wie Anfangs bereits erwähnt, besteht das Team, welches für die Entwicklung der NTE.CLM-Plattform verantwortlich ist, aus sechs Entwicklern. Dezidierte Softwaretester konnten aufgrund bestimmter Gegebenheiten in der Unternehmung ursprünglich nicht eingestellt werden. Damit sind also die Entwickler selbst dafür verantwortlich, dass das System die erforderlichen Qualitätsmerkmale erfüllt.

Laut [33] sollte von den Entwicklern selbst nur der Unittest durchgeführt werden und bei kleineren Projekte auch noch der Integrationstest. Bei größeren Projekten sollte auch der Integrationstest von eigenen Testern durchgeführt werden. Der Systemtest jedoch sollte ganz klar von einer eigenen Testabteilung durchgeführt werden.

Allerdings ist es so, dass in der Praxis die von der Theorie vorgegebenen Prinzipien oftmals schlicht und ergreifend nicht einzuhalten sind - so auch in der ersten Entwicklungsphase der NTE.CLM- Plattform bei der es schlichtweg per Definition die Aufgabe der Entwickler war, das System vollständig, d.h. angefangen beim Unittest bis hin zum Systemtest, zu testen.

Dass Entwickler nicht selbst alles testen sollen, hat klare Gründe [16].

- Sie tendieren dazu nur das zu testen, was sie sich ohnehin vom System erwarten
- Ihnen fehlt einfach die Erfahrung die ein gut ausgebildeter Softwaretester mitbringt
- Entwickler tendieren dazu, Tests nur sporadisch zu erstellen
- Haben oftmals einfach nicht die Zeit, genügend zu testen, da bereits weitere Funktionalitäten auf ihre Umsetzung warten

Auch bei der Entwicklung des NTE.CLM Systems konnten so gut wie alle diese Probleme nach genauerer Analyse aufgezeigt werden. Ein Indikator für eine schlechte nicht ausreichende Qualitätssicherung war, dass die Tests für den Server - welche vom Server-Team erstellt wurden - positiv absolviert wurden, doch sobald das System vom Client angesprochen wurde, kam es meist nach nur wenigen Aufrufen der Services zu unerwartetem Systemverhalten bzw. zu unerwarteten Systemfehlern.

Dieser Umstand konnte primär auf den Punkt „Sie tendieren dazu, nur das zu testen was sie sich ohnehin vom System erwarten“ zurückgeführt werden. Die Tests waren derart aufgesetzt, dass die Aspekte welche der Entwickler genau beachtet hatte und von denen er sicherstellen wollte, dass sie funktionieren, getestet wurden, aber schon die kleinste Abweichung von diesen Aspekten (z.B.: ungültige Eingaben, Ablaufreihenfolgen welche der Entwickler während der Implementation nicht berücksichtigte) führten oft zu Problemen.

4.4.2 Motivation

Um den im vorherigen Punkt beschriebenen Problemen entgegenzuwirken, wurde im Rahmen dieser Arbeit und der damit einhergehen Entwicklung des NTE.CLM-Systems versucht Techniken für die automatische Testfallerstellung zu finden und im Rahmen einer unternehmerischen Tätigkeit effizient einzusetzen.

Zu diesem Zeitpunkt war noch keinerlei Information über Testautomatisierung in diesem Projektumfeld vorhanden. Also musste mit einer Recherche bezüglich der Testautomatisierung (wenn möglich mit Bezug auf SOA und Webservice-Systeme) begonnen werden. Da die Implementierungsplattform für das System C# war wurde zuerst Automatismen für diese Programmiersprache gesucht und dies führte zu zwei vielversprechenden Tools, Pex und SpecExplorer. Bei der Suche nach Testautomatisierung im Zusammenhang mit SOA und Webservice Architekturen fiel immer wieder der Begriff „Fehler-Injektion“. Basierend auf dieser ersten Grundrecherche wurden diese drei Testmethoden eingehender studiert.

Fehlerinjektion Die Fehlerinjektion ist eine Testtechnik, welche gezielt Fehler in einem System hervorruft und das Systemverhalten bei Eintreten dieser Fehler überwacht. Die Fehlerinjektion wird primär in drei Arten der Fehlerinjektion eingeteilt.

- Hardware - Fehlerinjektion
  Diese Art der Fehlerinjektion wird meist beim Test von Hardware eingesetzt oder um die Reaktion von Software auf Hardware Fehler zu testen [42].
- Software - Fehlerinjektion
  Hierbei werden Programmcodes verändert, sodass Fehlerfälle auftreten. Z.B. kann ein Softwaremodul durch eine fehlerhafte Implementierung ausgetauscht werden, oder Codemutationen innerhalb des Moduls eingefügt werden.
- Protokoll - Fehlerinjektion
  In diesem Fall werden Nachrichten, welche von Komponenten ausgetauscht werden so mutiert, dass Fehlerfälle simuliert werden.

Besonders im Falle von SOA-Architekturen wird auf diese Art des Testens zurück gegriffen. SOA Systeme bestehen aus einer großen Anzahl lose gekoppelter Komponenten, welche auf unterschiedlichen Layern agieren. Das Problem in solchen Systemen ist, dass ein Fehler in einer der Komponenten dazu führen kann, dass sich dieser Fehler durch darauf aufbauende Komponenten fortpflanzen und somit ganze Systemteile lahmlegen kann [20].

Fehlerinjektion ist also eine Testmethode welche primär darauf abzielt, den QOS sicherzustellen. Im Falle einer Web-Service Architektur bietet sich die Fehlerinjektion an, da diese in einem solchen System recht einfach einzuführen ist, da sowohl die Protokoll-Fehlerinjektion als auch die Software-Fehlerinjektion angewendet werden kann.

Pex - Automatisierte White-Box-Tests Microsoft Pex ist ein Framework, welcher automatisiertes White-Box-Testen ermöglicht. Pex generiert Testfälle, indem es für die Eingabeparameter einer Methode Testdaten generiert, welche anhand dynamischer symbolischer Evaluierung des zu Grunde liegenden Programmcodes determiniert werden. Pex versucht dabei eine möglichst kleine Testreihe, welche aber eine möglichst hohe Programmablaufabdeckung erreicht, zu generieren [10].

Die von Pex erzeugten Testreihen werden dann in Form von Unit-Tests für ein beliebiges C# Unit-Testing Framework hinterlegt und können dann bei Bedarf wieder ausgeführt werden. Automatisiertes White-Box-Testen ist eine sehr komplexe Thematik. Eine weiterreichende Erklärung dieses Systems würde den Rahmen dieser Arbeit bei weitem sprengen - vor allem da Pex im Rahmen dieser Arbeit **nicht** als geeigneter Kandidat für das Testen des NTE.CLM-Systems eingestuft wurde.

SpecExplorer - Modellbasiertes Testen Siehe 5.

4.4.3 Ergebnisse

Als geeignetste der zuvor genannten drei Alternativen wurde der modellbasierte Testansatz mit Hilfe von SpecExplorer eingestuft. Diese Einschätzung wurde durch mehrere Faktoren getroffen. Einer dieser Faktoren ist das Projekt an sich. NTE.CLM ist ein neu entwickeltes System, für welches als wichtigster Testzweck die Validierung der Businessfunktionalität identifiziert wurde.

Daher ist der Ansatz mittels Fehlerinjektion ausgeschieden, da diese wesentlich mehr den QOS-Aspekt bedienen soll, aber nicht geeignet ist, um die eigentliche Funktionalität eines Systems zu validieren. Auch der Pex Ansatz kann diesen Umstand anhand der durchgeführten Evaluierung nicht ausreichend bedienen, da Pex wesentlich mehr auf den Test einzelner Komponenten zugeschnitten ist. Mit Pex wäre es also möglich, einzelne Komponenten sehr gut zu testen, aber durch nachgewiesene Korrektheit einzelner Komponenten ist in keinster Weise sichergestellt, dass diese in Zusammenarbeit - was für die Businessfunktionalität in einem SOA-System aber notwendig ist - ebenfalls gegeben ist.

Ein weiteres Problem von Pex ist, dass es primär darauf abzielt, mit den generierten Testfällen eine möglichst hohe Programmablaufabdeckung zu erreichen, dabei aber logische Testaspekte völlig außer Acht gelassen werden.

Schlussendlich ist zu sagen, dass die Eigenschaften des modellbasierten Testens bei weitem am adäquatesten für das zu Grunde liegende Projekt erschienen. Zum einen ist der modellbasierte Test eine Blackbox-Testtechnik welche nur eine Schnittstelle des Systems benötigt, um dieses zu testen. Eine genau solche Situation herrscht bei den Web-Services, welche lediglich ihre Schnittstelle für ihr Umfeld bereitstellen. Weiters ermöglicht es die Testgenerierung anhand von Modellen dem Tester, jene logischen Strukturen des Systems zu validieren, welche als wichtig erachtet werden, da durch die entsprechende Modellierung der Testzweck recht genau bestimmt werden kann. Aus diesen Gründen wurde für das automatisierte Testen des NTE.CLM Systems der modellbasierte Testansatz gewählt.

5 Modellbasiertes Testen in NTE.CLM

5.1 SpecExplorer

SpecExplorer ist ein Plugin für Visual Studio. Dieses Plugin ermöglicht, die Erstellung von Modellen und das Testen der IUT anhand dieser Modelle indem automatisch Testfälle erstellt werden können. Im Rahmen dieser Arbeit wurde SpecExplorer in der Version 3.3 verwendet. Die aktuellste Version dieses Tools kann von `http://visualstudiogallery.msdn.microsoft.com/en-us/271d0904-f178-4ce9-956b-d9bfa4902745` heruntergeladen werden.

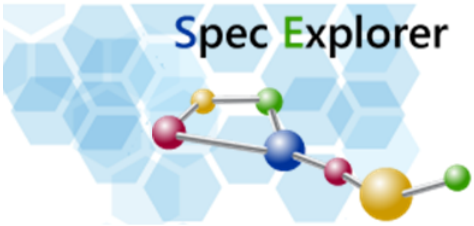

Abbildung 11: SpecExplorer Logo

SpecExplorer ist lediglich mit Visual Studio in der Version 2010 kompatibel. Ältere Versionen von Visual Studio werden mittlerweile nicht mehr unterstützt.

Nachdem SpecExplorer heruntergeladen wurde, kann es installiert werden (Visual Studio darf während des Installationsvorgangs nicht ausgeführt werden). Nach erfolgreicher Installation kann Visual Studio wieder gestartet werden. Dass SpecExplorer erfolgreich installiert wurde, kann man an folgenden Änderungen in Visual Studio erkennen.

- Neuer Visual C# Testprojekttyp "Spec Explorer Model"
- Neuer Toolbar-Eintrag Spec Explorer
- Neues Toolfenster Exploration Manager

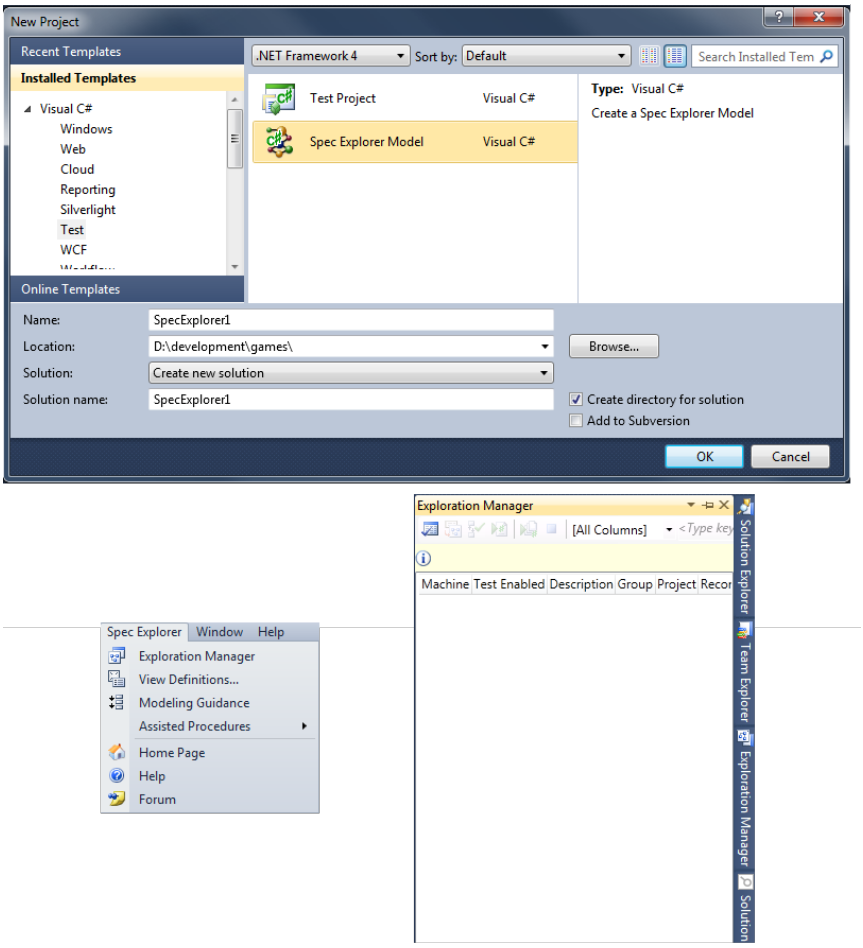

Abbildung 12: SpecExplorer IDE Elemente

5.2 Testprojekte einrichten

Der erste Schritt nach der Installation von SpecExplorer ist es, ein SpecExplorer Modell-Projekt zu erstellen. Zu diesem Zweck gibt es den SpecExplorer Model Wizard. Der Modelwizard hilft bei der initialen Erstellung eines Testprojektes für modellbasiertes Testen. Der Wizard erstellt dabei Demoprojekte mit einfachen Implementierungen, welche bereits direkt getestet werden. Diese Projekte können dann als Grundlage für das Testen des realen IUT herangezogen werden, indem sie entsprechend modifiziert werden.

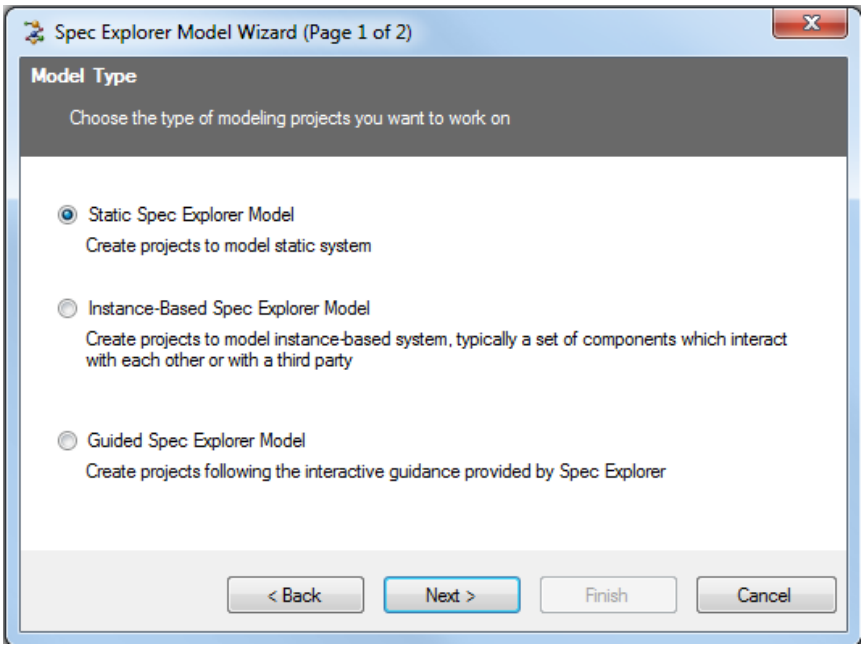

Abbildung 13: SpecExplorer Model Wizard

Der Modell Wizard kann drei verschiedene initiale Projektkonfigurationen erstellen.

5.2.1 Statisches SpecExplorer-Modell

Bei einem statischen Modell gilt, dass das System, welches getestet wird, im Grunde genommen lediglich statische Methoden anbietet. D.h., es gibt nur eine einzige Modellinstanz. Das heißt aber nicht, dass mit dieser Art von Modell nur statische Klassen getestet werden können. Vielmehr bedeutet die Verwendung eines statischen Modells, dass das Modell eine Betrachtungsweise des Systems heranzieht, bei welcher es nicht von Bedeutung ist, ob mehrere Modellinstanzen eines Modells miteinander interagieren oder nicht.

Ein sehr gutes Beispiel für eine solche Betrachtungsweise sind Web-Services. Für einen Client ist es irrelevant, ob mehrere Proxy-Instanzen zum Ansprechen des Server-Systems verwendet werden oder nicht. Das Verhalten des Serversystems gegenüber dem Client sollte unabhängig davon, ob nun mehrere oder ein Client-Proxy verwendet werden, immer gleich sein.

Da statische Modelle im Allgemeinen einfacher zu modellieren und übersichtlicher sind, ist diese Projektvorlage - sofern adäquat - zu bevorzugen. Dass ein Modell statisch ist, erkennt man daran, dass die Modellklasse ebenfalls statisch ist und kein Typebinding für das Modell anwendbar ist.

```
1 static class TestModelProgram
2 {
3   //State
4         ...
5   //Actions
```

```
6           ...
7   }
```

5.2.2 Instanzbasiertes SpecExplorer-Modell

Konträr zur Projektvorlage für statische Modelle gibt es eine für instanzbasierte Modelle. Bei dieser Projektvorlage ist es nun so, dass das Modell so gestaltet ist, dass mehrere Modellinstanzen bei der Exploration erzeugt werden, welche dann miteinander interagieren. Instanzbasierte sind nicht-statische Klassen und haben ein Typebinding, um auf die Implementierung zu verweisen.

```
1   [TypeBinding("ClassUnderTest")]
2   class TestModelProgram
3   {
4           [Rule(Action = "new␣ClassUnderTest()"]
5           TestModelProgram()
6           {
7             ...
8           }
9
10          [Rule(Action = "this.Foo()"]
11          int PerformFoo()
12          {
13            ...
14          }
15  }
```

```
1   class ClassUnderTest()
2   {
3           public ClassUnderTest()
4           {
5              ...
6           }
7
8           public int Foo()
9           {
10             ...
11          }
12  }
```

Das Typebinding ist für die Erstellung der Testfälle notwendig. Die aus dem Modell generierten Testfälle müssen die eigentliche Implementierung des zu testenden Systems ansprechen. Genau zu diesem Zweck gibt es das Typebinding. Es spezifiziert, wie die Beziehung zwischen Modell und der Implementierung auszusehen hat.

Während bei der Auswertung des Modells die Modellklasse herangezogen wird, werden für die Erstellung über das Typebinding die Regelaktionen des Modells auf Methoden der eigentlichen Implementierung umgewandelt und diese werden dann für die Testfalldurchführung verwendet.

5.2.3 Geführtes SpecExplorer-Modell

Bei dieser Projektvorlage handelt es sich um eine Vorlage mit einem Projekt für das Modell und einem Projekt für die Testfälle. Die Projekte an sich sind eigentlich leer (eine Datei für das Modell und eine Datei für die Testfälle wird angelegt, aber nicht ausgefüllt).

Zusätzlich dazu gibt es aber eine Liste mit Punkten, die abzuarbeitend sind. Wenn alle diese Punkte abgearbeitet wurden, hat man alle Nötigen Schritte durchgeführt, um Testfälle für eine IUT durchzuführen.

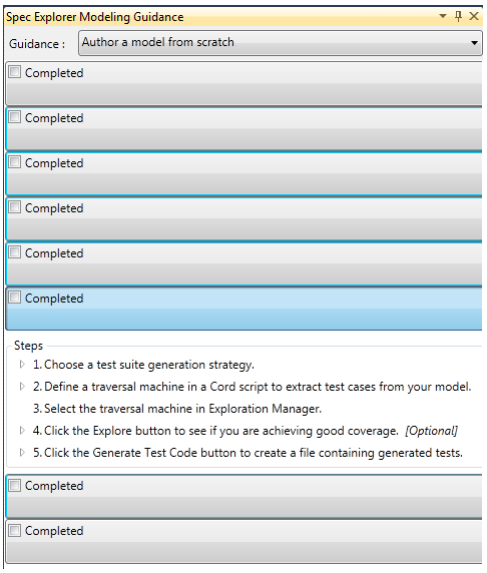

Abbildung 14: SpecExplorer-Punktliste für geführte Durchführung

5.2.4 Adapter

Bisher wurde immer davon gesprochen, dass die Testfälle, welche aus dem Modell generiert werden, das IUT treiben.

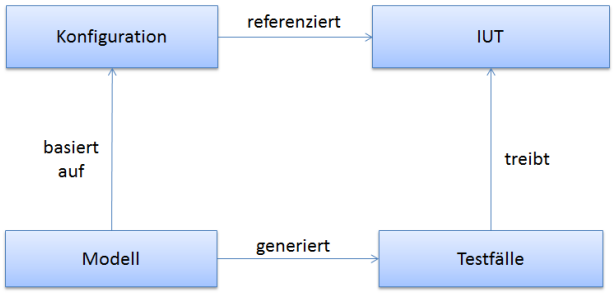

Abbildung 15: MBT Schema ohne Adapter

In realen Testumgebungen muss aber meist ein Adapter eingesetzt werden. Der Adapter ist ein Softwarelayer, welcher primär die Aufgabe hat, bestimmte Implementierungsaspekte

des IUT zu abstrahieren. Das Modell wird also nicht direkt auf das IUT bezogen, sondern vielmehr auf die Abstraktion des selbigen, also den Adapter

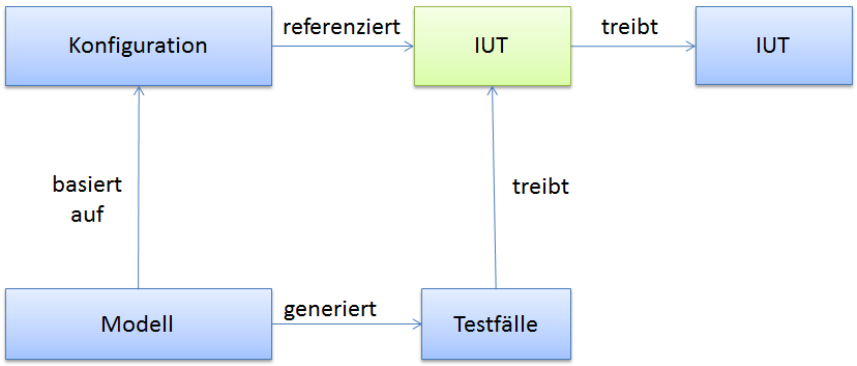

Abbildung 16: MBT Schema mit Adapter

Die Notwendigkeit eines Adapters ergibt sich daraus, dass mit einem Modell bestimmte Details der Implementierung nicht abgebildet werden können, oder bewusst vernachlässigt werden sollen.

Ein Beispiel für einen Aspekt, der nicht nachgebildet werden kann, ist die Vergabe einer ID in einem Datenbanksystem. Das Modell ist nicht in der Lage zu spezifizieren, welche ID die Datenbank für die nächste Entität, die in ihr abgelegt wird vergeben wird.

Hier kann der Adapter verwendet werden um das sogenannte "Mapped ID Pattern" anzuwenden. Wenn dieses Pattern Anwendung findet , wird sowohl für das Modell als auch für den Adapter auf gleiche Art und Weise spezifiziert, wie die nächste ID aussehen wird.

```
1
2  //Nicht im Model mitzaehlen da ansonsten Zustandsaenderung
3  //durch ID Vergabe eintreten wuerde
4  static class ModelIdCounter
5  {
6    static int idCounter = 1;
7
8          public static int GetNextId()
9          {
10     return idCounter++;
11         }
12 }
13
14 static class Model
15 {
16   [Rule]
17   static void CreateEntity()
18   {
19     int entityId = ModelIdCounter.GetNextId();
20     state.Insert(entityId);
21   }
22 }
```

```
1
2  static class Adapter
3  {
4    static Dictionary<int, Guid> adapterToRealIdMap = new Dictionary<int, Guid>();
```

```
5    static int idCounter = 1;
6    ClassUnderTest classToTest = new ClassUnderTest();
7
8    static void CreateEntity()
9    {
10       Guid entityId classToTest.CreateEntity(name, description);
11       int adapterId = idCounter++;
12       adapterToRealIdMap.Add(adapterId, entityId);
13   }
14 }
```

Durch diese Technik wird erreicht, dass im Modell die Vergabe von ID's für die Entitäten wieder modelliert werden kann. Die ID-Vergabe wird im Adapter auf ein äußerst einfaches Verfahren abstrahiert. Diese Mechanik kann dann im Modell gleich umgesetzt werden (in beiden Fällen wird einfach ein Zähler für die ID erhöht).

Der Adapter muss aber schlussendlich die „echte" Implementierung treiben. Daher speichert er sich die notwendigen Informationen, um einerseits die abstrakte ID in eine reale und umgekehrt konvertieren zu können.

Wie bereits erwähnt, wird der Adapter weiters verwendet um Aspekte des IUT zu abstrahieren. So eine Abstraktion könnte z.B. darin bestehen, dass es für die Betrachtung des Systems aus der Modellperspektive unerheblich ist, welche Eigenschaftswerte eine Entität hat.

Ein Beispiel für einen solchen Fall ist, wenn ein Modell unter dem folgendem Betrachtungswinkel erstellt wird. „Eine Entität, welche im System angelegt wurde, muss auch wieder gelöscht werden können". Für das Testen dieser Funktionalität ist es ohne Bedeutung, ob die Entität, welche erstellt wurde, korrekte (valide) Eigenschaften - wie z.B., dass die Eigenschaft Name der Entität ungleich der Leerzeichenfolge sein muss - hat.

Folgendes Beispiel zeigt eine solche Abstraktion durch den Adapter.

```

class Entity
{
  int Id { get; set; }
  string Name {get; set;}
  string Description { get; set; }
}
```

```

static class Model
{
  List<int> entities = new List<int>();

  [Rule(Action = "CreateEntity")]
  int CreateEntity()
  {
    int id = ModelIdCounter.GetNextId();
    entities.Add(id);
  }

  [Rule(Action = "DeleteEntity")]
  void DeleteEntity(int entity)
  {
```

```
16      entities.Remove(entity);
17    }
18  }
```

```
1
2  static class Adapter
3  {
4    static Dictionary<int, Entity> entities = new Dictionary<int, Entity>();
5
6    static int CreateEntity()
7    {
8      Entity toCreate = new Entity
9      {
10       Id = 0,
11       Name = CreateUniqueString(),
12       Description = CreateUniqueString()
13     };
14     Entity created  = classToTest.Create(toCreate);
15     entities.Add(created);
16     return created.Id;
17   }
18
19   static void DeleteEntity(int entityId)
20   {
21     Entity e = entities[entityId];
22     classToTest.Delete(e);
23     entities.Remove(e);
24   }
25 }
```

Der Adapter ist ein sehr einfacher und schlanker Abstraktionsmechanismus für das IUT. Dennoch gehört gerade dieses sehr einfache Design-Pattern, das hier eingesetzt wird, zu den Gründen warum modellbasiertes Testen so flexibel und damit mächtig wird. Denn dadurch wird es möglich, IUT's so weit zu abstrahieren dass die Modelle auch dazu benutzt werden können, um automatisch Testfälle zu erstellen. Der Adapter ermöglicht es, jede beliebige Komponente, welche ein Interface für die Ein- und Ausgabe hat mit Hilfe von SpecExplorer zu testen (dazu gehören z.B. auch eigenständige Anwendungen oder Softwaremodule welche nicht in C# implementiert wurden).

Das Erstellen eines Adapters ist einer der ersten Arbeitsschritte bei der Erstellung eines modellbasierten Testprojekts. Der Adapter gibt das Interface vor, welches dann in der Modellspezifikation verwendet wird. Je nach Zieldefinition des Modellprogramms können durchaus mehrere Adapter für ein und die selbe Systemkomponente notwendig sein.

5.3 Modellerstellung

Sind die Projekte eingerichtet und das IUT bzw. der Adapter vorhanden, kann mit der Modellerstellung begonnen werden. Im Rahmen dieser Arbeit soll anhand von zwei konkreten Services des NTE.CLM-Systems gezeigt werden, wie Modelle erstellt werden können.

Zuerst soll die Modellerstellung anhand des BusinessObjectDataServices dargestellt werden. Dieser Service ist die Abbildung des Server-Datenspeichers für höherwertige Objekte,

die sogenannten Businessobjekte. Ein BusinessObject ist im NTE.CLM-System ein Objekt welches folgende Eigenschaften besitzt.

- Es kann gesichert werden, d.h. es könnten Autorisierungen auf diese Objekte vergeben werden
- Es kann mit anderen Businessobjekten verknüpft werden
- Jedes Businessobjekt wird mit serverinternen Informationen verknüpft, welche der Außenwelt nicht zugänglich sind

Das Interface dieses Services ist wie folgt definiert:

```
1  [ServiceContract(Namespace = "http://ntesystems.at/clm/r3/",
2    Name = "IBusinessObjectDataService")]
3  public interface IBusinessObjectDataService
4  {
5      Dto.BusinessObject GetById(long id);
6      Dto.BusinessObject GetByIdWithType(long id, Dto.BusinessObjectType boType);
7      List<Shared.Dto.BusinessObject> GetByIdListWithType(List<long> idList,
8        Shared.Dto.BusinessObjectType boType);
9      List<Dto.BusinessObject> GetByType(Dto.BusinessObjectType type);
10     List<Dto.BusinessObject> GetAssignmentSources(Dto.BusinessObject obj);
11     List<Dto.BusinessObject> GetAssignmentSourcesWithType(Dto.BusinessObject obj,
12       Dto.BusinessObjectType sourcesType);
13     List<Dto.BusinessObject> GetAssignmentTargets(Dto.BusinessObject obj);
14     List<Dto.BusinessObject> GetAssignmentTargetsWithType(Dto.BusinessObject obj,
15       Dto.BusinessObjectType targetsType);
16     Dto.BusinessObject Create(Dto.BusinessObject newBusinessObject);
17     Dto.BusinessObject CreateAndAssignToTarget(Dto.BusinessObject newBusinessObject,
18       Dto.BusinessObject assignTo);
19     Dto.BusinessObject Update(Dto.BusinessObject toUpdate);
20     bool AddAssignment(Dto.BusinessObject obj, Dto.BusinessObject target);
21     bool RemoveAssignment(Dto.BusinessObject obj, Dto.BusinessObject target);
22     void Delete(Dto.BusinessObject obj);
23 }
```

Der Service ist für das Anlegen, Abrufen und das logische Verknüpfen von Basisdaten zuständig. Zu solchen Daten gehören z.B. Kunden und die mit ihnen verknüpften Anlagen. Dieser Service ist für modellbasiertes Testen sehr gut geeignet, weil er folgende Eigenschaften erfüllt

- Der Service ist reaktiv
  Als reaktiv wird ein System bezeichnet, wenn es mit seiner Umgebung interagiert.
- Der Service verändert den Zustand des Systems
  Der Systemzustand wird durch die gespeicherten Daten repräsentiert.

Die besondere Herausforderung beim Testen dieses Services ist, eine möglichst hohe Testabdeckung zu erreichen. Der Service ist nach reiner Betrachtung der Service-Schnittstelle nicht sonderlich komplex oder umfangreich und es scheint so, als wäre es nicht sonderlich aufwändig, die korrekte Implementierung mit der manuellen Erstellung von Testfällen abzudecken. Aber der Umstand, dass es insgesamt zehn verschiedene Business-Objekt-Typen gibt und dass diese auf bestimmte Art und Weise verknüpft werden können, treibt die Auf-

wände für die manuelle Testerstellung in die Höhe. Zum Zeitpunkt der Erstellung dieser Arbeit existierten folgende Typen von Business-Objekten:

- Kunde (Customer)
- Anlage (Facility)
- Anlagenschama (FacilitySchema)
- Steuereinheit (ControlUnit)
- Benutzer (User)
- Datenquelle (Datasource)
  - Steuereinheit-Datenquelle (ControlUnitDatasource)
  - Benutzerdefinierte Datenquelle (CustomDatasource)
  - System-Datenquelle (SystemDatasource)
- Regel (Rule)

Alleine für die Methode „Create“ müssten prinzipiell zehn Testfälle erstellt werden - einer für jeden Business-Objekt Typ - um eine vollständige Testabdeckung zu erreichen [4].

Zusätzlich kommt hinzu, dass die Daten nur auf bestimmte Art und Weise verknüpft werden dürfen. Z.B. darf eine Anlage nur einem Kunden zugewiesen werden. Konträr dazu darf eine Regel hingegen zu den Business-Objekt-Typen Kunde, Anlage, Steuereinheit zugewiesen werden. Für all diese gültigen Kombinationen Positivtests bzw. für die ungültigen Kombinationen Negativtests zu erstellen ist sehr aufwändig. Aus diesem Grund, war dieser Service einer der primären Kandidaten für das modellbasierte Testen.

In SpecExplorer wird ein Modellprogramm durch eine Modellprogrammklasse repräsentiert. [5]

Bei der Modellerstellung geht es darum, Regeln für die Aktionen zu definieren. Die möglichen Aktionen werden vom IUT bzw. dem dazugehörigen Adapter vorgegeben. Eine Regel dient nun dazu, zu spezifizieren, welche Änderungen im Modellzustand eintreten, wenn diese Aktion ausgeführt wird.

Jede Regel wird durch eine Methode im Modellprogramm repräsentiert, welche mit dem "Rule" Attribut versehen wird. Dieses Attribut ist wie folgt definiert

[4] Der Umstand, dass für eine annähernd komplette Testabdeckung auch noch die Parameter des zu erstellenden Business-Objekts variiert werden müssten, wird an dieser Stelle vernachlässigt, da es sich hier um eine abstraktere Systemsicht handelt

[5] Modelle in den Vorgängerversionen von SpecExplorer 2010 (v.3.0) wurden mittels einer eigens dafür entwickelten "Design by Contract" Sprache Spec# definiert. Mit Visual Studio 2010 und dem damit eingeführten .NET 4.0 Framework wurde "Design by Contract" integraler Bestandteil von C#. Daher können Konstrukte, wie z.B. die Modellprogramme nun direkt mittels C# erstellt werden

```
    [AttributeUsage(AttributeTargets.Constructor | AttributeTargets.Method |
      AttributeTargets.Property, AllowMultiple = false, Inherited = true)]
    public sealed class RuleAttribute : Attribute
    {
        public RuleAttribute();
        public string Action { get; set; }
        public ParameterExpansionPoint DefaultParameterExpansionPoint { get; set; }
        public string ModeTransition { get; set; }
    }
```

Das Attribut wird innerhalb einer Modellklasse also folgendermaßen angewendet

```

static class ModelProgram
{
  [Rule]
  public static float RuleMethod(int x) {...}
}
```

Zustäzlich kann bei einer Regel noch die Eigenschaft "Action" gesetzt werden. Dabei handelt es sich um einen String, welcher im Cord-Syntax angegeben werden muss. Diese Zeichenfolge kann verwendet werden, um genau zu spezifizieren, zu welcher Aktion diese Regel gehört. Die Definition

```
static class ModelProgram
{
  [Rule(Action = "ActionMethod(x)/result")]
  public static float RuleMethod(int x) {...}
}
```

ist zum zuvor gezeigten Modellprogramm äquivalent. Primär wird das „Action“-Attribut eingesetzt, wenn der Name der Regel nicht gleich dem Namen der Aktion, für welche diese Regel gilt, ist. Die Aktion wird in der sogenannten Cord-Syntax angegeben. Bei dieser Syntax sind Datentypen irrelevant. Falls die Methode einen Rückgabewert liefert, wird dies durch ein abschließendes "/result" spezifiziert.

An dieser Stelle soll nun dargelegt werden, wie bei der eigentlichen Modellerstellung vorgegangen wird. Zu diesem Zwecke wird dargelegt, wie für die beiden Service-Operationen

1. BusinessObject Create(BusinessObject bo)
2. List<BusinessObject> GetByType(BusinessObjectType type)

die notwendige Adapter- und Modellfunktionalität umgesetzt wurde.

Create Für die Create Operation wurden Adapter und Modell-Regel wie folgt umgesetzt. Adapter:

```
public static Dto.BusinessObject Create(Dto.BusinessObject bo)
{
    using(var proxy = ServiceProxyFactory.Create<IBusinessObjectDataService>())
    {
        var created = proxy.Create(bo); //Erzeuge DTO mit validen Zufallswerten
```

```
6           proxy.Close();
7           return created;
8       }
9   }
10
11  public static List<Dto.BusinessObject> GetByType(Dto.BuinessObjecType boType)
12  {
13      using(var proxy = ServiceProxyFactory.Create<IBusinessObjectDataService>())
14      {
15          var created = proxy.Create(bo); //Erzeuge DTO mit validen Zufallswerten
16          proxy.Close();
17          return created;
18      }
19  }
```

Modell

```
1   [Rule]
2   public static Dto.BusinessObject Create(Dto.BusinessObject boType)
3   {
4       Condition.Requires(boType == typeof(Dto.Customer));
5       // Was ist in der Regel noch zu tun???
6   }
7
8   [Rule]
9   public static List<Dto.BusinessObject> GetByType(Dto.BusinessObjectType boType)
10  {
11      Condition.Requrires(boType == typeof(Dto.Customer);
12      // Was ist in der Regel noch zu tun???
13  }
```

Dieser erste äußerst naive Ansatz, um modellbasierte Techniken einzusetzen war zur Gänze unbrauchbar. In diesem Beispiel ist jedes Prinzip, welches für modellbasierte Techniken einzuhalten ist, verletzt. Dennoch soll an dieser Stelle kurz auf dieses Problem eingegangen werden, da jene Konstrukte, vor allem wenn für einen Entwickler modellbasierte Konzepte Neuland darstellen, immer wieder auftreten.

Welche Aspekte führten nun zu obigen Konstrukten?

Keine konzeptionelle Trennung von Modell und IUT/Adapter Ein sehr häufiges Problem bei den ersten Versuchen mit modellbasierten Techniken ist die konzeptionelle Trennung von Modell und dem IUT/Adapter. Oftmals wird versucht, das Modell zu genau dem IUT/Adapter nachzubilden. Dieser Umstand tritt meist ein, da bei der Modellerstellung immer der Gedanke im Kopf ist, dass das Modell das IUT mit Testfällen treiben muss. Daher ist der Modellersteller immer versucht, das Modell mit selbigen zu verknüpfen. Aufgrund dessen ist es dann aber oftmals nicht mehr möglich, sinnvolle Regeln zu definieren.

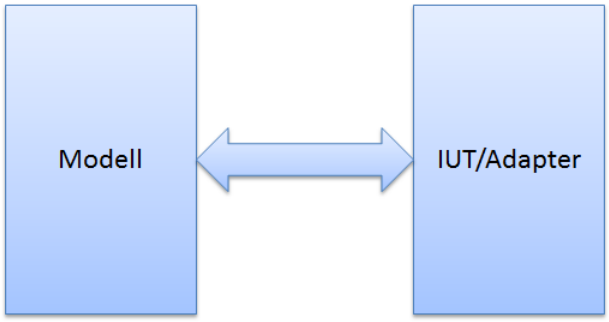

Abbildung 17: Falsches Modellkonzept

Es zeigt sich, dass sich diese Sichtweise bei den Entwicklern durch die klassischen Testkonzepte, wie sie z.B. dem Unit-Test festgesetzt hat. Daher stellt modellbasiertes Testen Entwickler vor eine spezielle Herausforderung, da dies ein gänzlich anderes Testkonzept darstellt und somit ein vollständiges Umdenken und Loslösen von bereits bekannten Vorgehensweisen erfordert.

Eine mögliche Sichtweise, um sich von diesem zuvor gezeigten Konzept zu lösen könnte so aussehen.

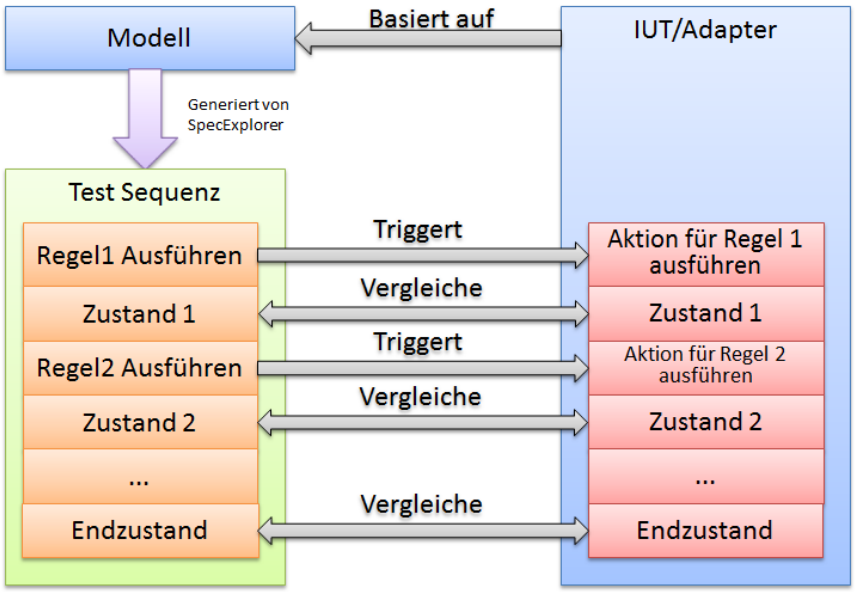

Abbildung 18: Mögliches Modellkonzept

Bei dieser Sichtweise ist es nun so, dass das Modell nicht mehr direkt mit dem IUT/Adapter verknüpft wird. Vielmehr ist das Modell hier eine wirklich losgelöste Abbildung des selbigen, welche nur die Funktionalität eines Systems modellieren soll, aber nicht mehr direkt die Abbildung des Systems darstellt.

Vielmehr verändert sowohl das Modell als auch die/der IUT/Adapter einfach entsprechend

ihrer Regeln bzw. ihrer Aktionen den internen Zustand [6].

In diesem Fall ist es sehr wichtig, den Aspekt heranzuziehen, dass es nicht die eigentliche Aufgabe des Modells ist, für die Erstellung von Testorakeln und den dazugehörenden Testfällen verantwortlich zu sein, sondern vielmehr auf abstrakte Art und Weise ein Systemverhalten abzubilden. Die eigentliche Generierung der Testorakel und Testfälle obliegt weit mehr den Tools, welche das Modell verarbeiten. Erst diese stellen eine Verbindung über die erzeugten Testfälle zwischen Modell und IUT/Adapter her.

Keine Abstraktionen vorgenommen Ein weiteres Problem im gezeigten Beispiel ist, dass keinerlei Abstraktionen vorgenommen wurden. Durch die fehlenden Abstraktionen, welche sich in diesem Fall dadurch manifestieren, dass der Adapter exakt der Service-Schnittstelle entspricht, ist es nicht möglich, die Regel zu definieren. Der Grund dafür ist, dass es nicht möglich ist, in der Regel des Modells jegliche Annahme über den Zustand des Zielsystems zu treffen. Während der Modellierung kann nicht spezifiziert werden, welche Werte die Eigenschaften (ID, Name, etc.) das erzeugte Businessobjekt annehmen wird.

5.3.1 Modellvalidierung

Dass das zuvor gezeigte Beispiel nicht korrekt ist, wird auch von SpecExplorer selbst während einer Validierung bestätigt. Die Validierung kann im Exploration Manager - einem, Element welches in der Visual Studio UI integriert wird und das primäre Werkzeug für die Verwendung von SpecExplorer darstellt [7] - eingeleitet werden.

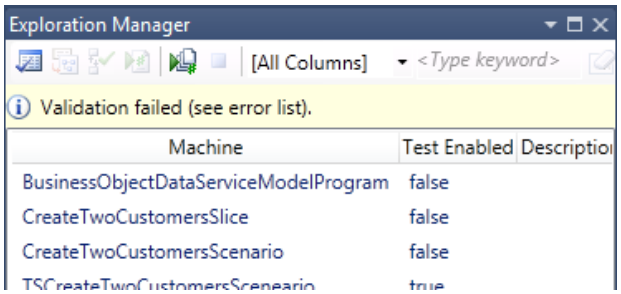

Abbildung 19: Exploration Manager Werkzeug Fenster

Die erste Schaltfläche dient dazu, die Validierung des Modells einzuleiten. Das Resultat der Validierung für das Beispiel liefert folgende Ausgabe.

[6] Auch die Rückgabewerte einer Funkion sind Teil des jeweiligen Zustands.

[7] Spec Explorer kann auch außerhalb von Visual Studio mittels Applikationen welche über die Kommandozeile angesteuert werden betrieben werden

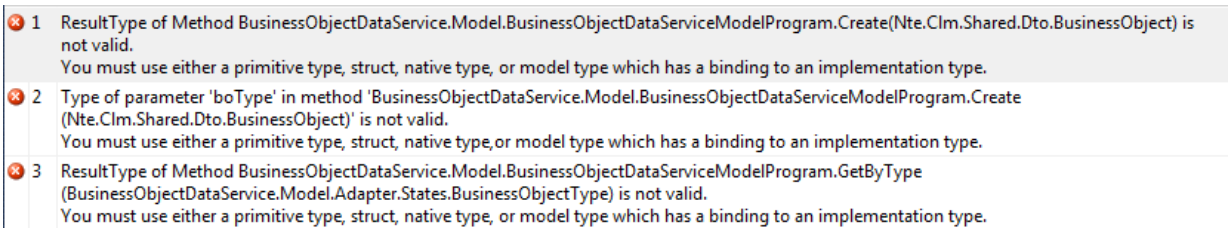

Abbildung 20: Ergebnis der Validierung

Im Wesentlichen sagt das Ergebnis aus, dass SpecExplorer die angegebenen Eingabe- und Ausgabeparameter des Modells nicht verarbeiten kann. Grund dafür ist, dass Parameter des Modells, welche SpecExplorer während der Exploration verarbeitet, gewissen Einschränkungen unterliegen. Sie müssen ein primitiver Datentyp (int, bool, etc.) eine Struktur [8], ein nativer Datentyp (z.B. DateTime) oder ein Modelltyp mit einem TypeBindnig sein. Die ersten drei Bedingungen bedürfen keiner weiteren Erklärung. Lediglich der letztere Fall mit dem TypeBinding stellt eine besondere Eigenheit von SpecExplorer dar.

TypeBinding Würden alle Parameter den ersten drei der zuvor genannten Einschränkungen unterliegen, könnten viele komplexe Modellszenarien von SpecExplorer nicht bedient werden. Daher gibt es das sogenannte TypeBinding. Die Defintion des TypeBindings nach MSDN lautet "Type bindings are used for translating rule-method parameters, which belong to model types, into action parameters, which belong to implementation types. When testing, a one-to-one mapping is established between instances returned by the implementation and model types instances appearing in assumptions derived from model exploration." [5]

Damit SpecExplorer "komplexe" Datentypen während der Modellevaluierung verarbeiten kann, wie es z.B. notwendig ist, wenn sie als Parameter oder Rückgabewerte verwendet werden, müssen für diese wiederum Modelle erstellt werden. Diese Modelle stellen den Modell-Typ dar. Der Adapter bzw. die IUT werden aber mit den tatsächlichen Typen der Implementierung betrieben. Daher ist es notwendig, dass SpecExplorer ein Mapping zwischen dem Modelltyp und dem Implementierungstyp bilden kann, genau das wird mit dem TypeBinding erreicht. SpecExplorer kann nun für die Auswertung und Exploration den Modelltyp verwenden, und den eigentlichen Implementierungstyp, der dann z.B. in den Testfällen verwendet wird. Mittels dem TypeBinding-Attribut kann SpecExplorer während der Auswertung des Modells den eigentlichen Implementierungstyp auflösen und somit während der Testfallerstellung verwenden.

TypeBindings sollten nur in Fällen eingesetzt werden, in denen sie wirklich unumgänglich sind, da sie die Erstellung des Modells sehr schnell wesentlich komplexer machen. TypeBindings sind generell nur notwendig, wenn komplexe Objekte - also tatsächlich Klassen, welche wiederum selbst Funktionalität abbilden - als Parameter an zu modellierende Pro-

[8] In C# stellen Strukuren Werte-Datentypen dar - diese können nicht "null" werden und müssen alle Felder initialisieren)

grammelemente übergeben werden müssen. Für das zuvor gezeigte Beispiel, in welchem dem Service lediglich Datentransferobjekte übergeben werden, sollte also nicht auf eine Modellierung selbiger und den damit notwendigen Einsatz eines TypeBinding zurück gegriffen werden. Stattdessen werden diese als einfache Datenhaltungsobjekte (Strukturen) gesehen, auch wenn sie im realen Programm auch Funktionalität inne haben.

SpecExplorer Container Das obige Beispiel enthält noch einen weiteren Fehler, der aus der Fehlerausgabe der Validierung noch nicht ersichtlich ist, aber dann bei der Exploration des Modells ersichtlich wird. Der Datentyp für den Rückgabewert der Methode "GetByType" ist als List<Dto.BussinessObject> definiert. Eine weitere Eigenheit von SpecExplorer ist, dass er nicht mit den Standard .NET Containern (Lists, Dictionary, etc.) arbeiten kann.

Für diese Beschränkung gibt es zwei Gründe:

1. Gleichheit
   Der Modellzustand in einem Modell wird oft mit Hilfe von Containern abgebildet. Die Vergleichs-Operationen der Standard-Container haben Eigenschaften, welche diese suboptimal für den Einsatz in einem Modellprogramm machen. .NET Datentypen können Gleicheit nicht garantieren. Eine Liste, die mit (100, 200) initialisiert ist - dieser Zustand der Liste soll als ListeStart definiert sein - , und auf welche dann Operationen wie das Anfügen des Elements und dann das Entfernen des selbigen, sodass auch in der finalen Liste wieder die Elemente wie nach der Initialisierung vorhanden sind - dieser Zustand soll als ListeEnde definiert sein, in diesem Fall ist nicht garantiert dass gilt ListeStart == ListeEnd. Ein reales Beispiel für diesen Fall ist das anwenden von LINQ-Operationen auf .NET collections. LINQ-Operationen/Transformationen verändert z.B. immer die Listenreferenz, weshalb die veränderlichen SpecExplorer Container diese auch nicht unterstützen.

2. Unveränderlichkeit
   Ein weiteres Problem ist, dass .NET-Container jederzeit verändert werden können. Diese Veränderlichkeit der Container erschwert SpecExplorer die Verarbeitung des Modells ungemein, denn wenn eine Liste von Objekten in einer Regel als Rückgabwert geliefert wird, kann diese Liste dazu verwendet werden, um den Modellzustand zu verändern und somit zu einem Teil des Modellzustands werden. Wären Standard-.NET-Container in diesem Fall erlaubt, wäre z.B. folgende Extremsituation prinzipiell möglich.

```
1
2   class Model
3   {
4     List<DataType> returnResult = null;
5
6     [Rule] public List<DataType> GetResult()
7     {
8         return returnResult;
9     }
```

```
10
11      public void OnTimer()
12      {
13        returnResult.Add(new DataType());
14      }
15    }
```

D.h., der Rückgabewert der Methode könnte außerhalb von Regelauswertungen mittels andere Methoden weiter verändert werden. Das würde bedeuten, dass während der Exploration das Resultat der Regel ungleich dem Resultat bei der nächsten Regelauswertung werden könnte, was eine sinnvolle Exploration des Modells unmöglich machen würde.

Aus diesen beiden Gründen definiert SpecExplorer eigene Container-Klassen, welche die oben genannten Nebeneffekte eliminieren. Im Folgenden eine kurze Übersicht über diese Container.

.NET Container	SpecExplorer Container	
	(veränderbar)	(unveränderbar)
List<T>	SequenceContainer<T>	Sequence<T>
Dictionary<T, T>	MapContainer<T, T>	Map<T,T>
HashSet<T>	SetContainer<T>	Set<T>

Tabelle 1: Zusammenhang SpecExplorer und .NET Container

Bei der Modellierung sollten die folgenden Regeln beachtet werden:

- Immer nur die SpecExplorer Container verwenden
- Für den Modellzustand die veränderbaren oder die unveränderbaren Container verwenden
- Für Parameter und Rückgabewerte die unveränderbaren Container verwenden

5.3.2 Ein valides Modell

In diesem Abschnitt soll nun anhand der zuvor genannten Aspekte gezeigt werden, wie man das invalide Beispiel von zuvor in ein valides umformt.

Zuerst müssen Stukturen gebildet werden, welche die Abbildung eines Business-Objekts darstellen. Diese sind wie folgt aufgebaut.

```
1 enum BussinessObjectTypeState
2 {
3   None              = 0,
4   Customer          = 1,
5   User              = 2,
6   ...
7 }
```

```
8
9  public struct BusinessObjectState
10 {
11   public int                Id        { get; set; }
12   public BusinessObjectType BoType    { get; set; }
13 }
```

Die Abbildung des Business-Objekts besteht aus lediglich zwei Eigenschaften, einer ID und dem Typ des Business-Objekts. Die ID wird benötigt, um die verschiedenen Instanzen der Business-Objekte unterscheiden zu können und damit die Testfälle den Adapter noch sinnvoll treiben können. Der Typ des Businessobjekts wird benötigt um den Aspekt unter welchem das Modell erstellt wurde sinnvoll abdecken zu können. Alle anderen Eigenschaften der Business-Objekte werden vollkommen vernachlässigt, es wurde also eine sehr **starke** Abstraktion des IUT vorgenommen.

Der Adapter ist nun folgendermaßen aufgebaut

```
1  public static class Adapter
2  {
3    static List<Tuple<int, BusinessObject>> modelStateRealDtoMapping =
4      new List<Tuple<int, BusinessObject>>();
5
6    public static void Create(BusinessObjectState state)
7    {
8      BusinessObject realDto = null;
9      if(state.BoType == BusinessObjectTypeState.Customer)
10     {
11       realDto = CreateCustomerDto();
12     }
13     else if(...)
14     {
15       ....
16     }
17
18     Customer c  = CreateCustomerDto();
19     var created = proxy.Create(c);
20     int modelId = AquireNextId();
21     modelStateRealDtoMapping.Add(Tuple.Create(modelId, created));
22     return new BusinessObjectState()
23     {
24       Id        = modelId,
25       BoType    = RuntimeTypeToTypeState(created),
26     };
27   }
28
29   public static List<BusinessObjectTypeState> GetByType(BusinessObjectTypeState bots)
30   {
31     BusinessObjectType boType    = TypeStateToType(bots);
32     List<BusinessObject> fetched = proxy.GetByType(boType);
33     List<BusinessObjectTypeState> result = new List<BusinessObjectTypeState>();
34     foreach(BusinessObject bo in fetched)
35     {
36       result.Add(new BusinessObjectTypeState()
37       {
38         Id          = modelState.Single((e) => e.Item2.Id == bo.Id).Item1,
39         BoType      = RuntimeTypeToTypeState(bo),
40       };
41     }
42
43     return result;
```

```
44    }
45
46    //Hilsfsmethoden
47    private static Dto.Customer CreateCustomerDto() {...}
48    private static Dto.User CreateUserDto() {...}
49    private static BusinessObjectTypeState RuntimeTypeToTypeState(BusinessObject bo) {...};
50    private static BusinessObjectType TypeStateToType(BuinessObjecTypeState bots) {...};
51    private static int AquireNextId() {...}
52 }
```

Der Adapter ist jenem aus dem allgemeinen Abschnitt über Adapter 5.2.4 sehr ähnlich und die Funktionsweise ist äquivalent. Der hier dargelegte Adapter ist nur der Vollständigkeit des Beispiels - es ist vor allem anfangs sehr wichtig zu sehen, wie alle Teile zusammen wirken - wegen noch einmal aufgeführt.

Das eigentliche Modellprogramm ist in einer sehr grundlegenden Umsetzung nun folgendermaßen aufgebaut.

```
1  public static class Model
2  {
3    static SequenceContainer<BusinessObjectState> existingObjects
4      = new SequenceContainer<BusinessObjectState>();
5
6    [Rule]
7    public static BusinessObjectState Create(BusinessObjectState bo)
8    {
9        Condition.IsTrue(existingObjects.Contains(bo) == false);
10       Condition.IsTrue(bo.BoType == BusinessObjectTypeState.Customer ||
11         bo.BoType == BusinessObjectTypeState.User);
12
13       BusinessObjectState state = new BusinessObjectState()
14       {
15           Id          = IdGenerator.AquireNextId(),
16           BoType      = bo.BoType
17       };
18
19       existingObjects.Add(state);
20       return state;
21   }
22
23   [Rule]
24   public static Sequence<BusinessObjectState> GetByType(BusinessObjectTypeState boType)
25   {
26       Condition.IsTrue(boType == BusinessObjectTypeState.Customer ||
27         boType == BusinessObjectTypeState.User);
28
29       return new Sequence<BusinessObjectState>(
30         existingObjects.Where((e) => e.BoType == boType).ToArray());
31   }
32 }
```

Das Modell-Programm besteht aus zwei Regeln. Die erste definiert das Verhalten des Systems, wenn ein BusinessObjekt erzeugt wird. Die Bedingungen der Regel sagen Folgendes über das System aus.

- Damit ein Business-Objekt erzeugt werden kann, muss es ein Objekt mit gültigem Datentyp sein (also „null“ z.B. ist nicht erlaubt)

- Das Objekt darf nicht schon einmal erzeugt worden sein

Sind diese beiden Bedingungen erfüllt, darf die Regel angewandt werden. Der zweite Teil der Regel spezifiziert, was dann vom System erwartet wird, wenn die Aktion durchgeführt wurde, und wie sich der Modellzustand nach Aktion dieser Regel verändert hat.

- Das neu erzeugte Objekt ist nun im System vorhanden (existingObjects.Add(state))
- Der erwartete Rückgabewert des Systems ist das neu erzeugte Objekt

Die zweite Regel ist folgendermaßen zu interpretieren

- Es muss ein bekannter Business-Objekt Typ angefragt werden (bei diesem Modell sind nur Customer und User abgebildet)
- Vom System müssen alle Objekte, welche diesen Type haben zurückgeliefert werden.

Das Modell spezifiziert also im Wesentlichen immer folgende Dinge

- Die Bedingungen wann eine Regel angewendet werden darf.
- Die Veränderung des Systemzustands
- Die Resultate von Systemaufrufen

Diese Modell spiegelt nur einen sehr kleinen Teil des BusinessObjectDataService wieder. Grundsätzlich ist es anzuraten, dass mit solchen sehr kleinen Modellen, welche nur **einen** bestimmten Aspekt des Systems abdecken, die Systemmodellierung begonnen wird. Erst wenn diese korrekt sind, sollten weitere Systemaspekte dem Modell angefügt werden. Dieses Vorgehen ist sinnvoll, da große fehlerhafte Modelle sehr schwierig zu korrigieren sind. Hingegen können additive Systemaspekte mit relative geringen Aufwand hinzugefügt werden. Einführend in dieses Kapitel wurde dargelegt, dass zwischen den verschiedenen Business-Objekt-Typen Verknüpfungen bestehen können. Diese Verknüpfungen werden durch die Service-Operation "CreateAndAssignToTarget(Dto.BusinessObject newBo, Dto.BusinessObject assignTo)" und z.B. GetAssignmentSorucesWithType repräsentiert. Um das Modell dahingehend zu modifizieren, dass diese Verknüpfungen berücksichtigt werden, ist es nun notwendig, neue Regeln für diese beiden Operationen einzuführen und den Modellzustand zu erweitern, sodass dieses Verhalten auch modelliert werden kann.

```
1  public static class Model
2  {
3    static MapContainer<BusinessObjectState, SequenceContainer<BusinessObjectState>>
4        assignmentSources = new MapContainer<BusinessObjectState,
5                            SequenceContainer<BusinessObjectState>>();
6
7    [Rule]
8    public static BusinessObjectState CreateAndAssignToTarget(BusinessObjectState newBos,
9                                                              BusinessObjectState existingBos)
10   {
11     Condition.IsTrue(existingObjects.Contains(existingBos));
12     Condition.IsTrue(existingObjects.Contains(newBos) == false);
13
14
15     Condition.IsTrue(// Erlaubte Business Objekt Verknuepfungen
```

```
16        (newBos.BoType == BusinessObjectTypeState.User &&
17          existingBos.BoType == BusinessObjectTypeState.Customer) ||
18        (newBos.BoType == BusinessObjectTypeState.Facility &&
19          existingBos.BoType == BusinessObjectTypeState.Customer) ||
20        (newBos.BoType == BusinessObjectTypeState.FacilitySchema &&
21          existingBos.BoType == BusinessObjectTypeState.Facility) ||
22        // ... weitere valide Verknuepfungen );
23
24      BusinessObjectState newBoState = new BusinessObjectState()
25      {
26          Id                          = IdGenerator.AquireNextId(),
27          BoType                      = newBos.BoType,
28      };
29
30      if(assignmentSources.ContainsKey(existingBos) == false)
31          assignmentSources.Add(existingBos, new SequenceContainer<BusinessObjectState>());
32      assignmentSources[existingBos].Add(newBoState);
33      existingObjects.Add(newBoState);
34
35      return newBoState;
36    }
37
38    [Rule]
39    public static Sequence<BusinessObjectState>
40      GetAssignmentSourcesWithType(BusinessObjectState target, BusinessObjectTypeState bots)
41    {
42      Condition.IsTrue(existingObjects.Contains(target));
43
44      if(assignmentSources.ContainsKey(target))
45          return new Sequence<BusinessObjectState>(
46            assignmentSources[target].Where((i) => i.BoType == bots).ToArray());
47      return new Sequence<BusinessObjectState>();
48    }
49  }
```

Der hier gezeigte Modellcode ist als additiver Code zu dem ursprünglichen Modell zu betrachten. Der Verknüpfungsaspekt wird in der Regel "CreateAndAssignToTarget" mittels der Bedingungen für diese Regel abgebildet. Alle möglichen Kombinationen werden als eine Und-Klausel abgebildet. Zusätzlich wurde der Modellzustand um eine Map erweitert, welche die erzeugten Verknüpfungen repräsentiert. Das Modell drückt nun aus, dass ein Objekt, welches mit dieser Regel erzeugt wird, ein neues Objekt ist, und somit den existierenden Objekten hinzugefügt werden muss, aber vom System auch eine Verknüpfung erzeugt werden muss.

Die Regel für "GetAssignmentSourcesWithType" besagt, dass die Operation für jedes existente Objekt ausgeführt werden kann, unabhängig davon, ob für selbiges Verknüpfungen existieren. Existieren welche, werden jene mit dem korrekten Typ zurückgegeben, ansonsten eine leere Liste.

Dieses Beispiel scheint auf den ersten Blick recht unspektakulär. Dennoch zeigt es, worin der riesige Vorteil von modellbasierten Techniken gegenüber klassischen Methoden liegt. Im Modell waren nur wenige und sehr einfache Änderungen notwendig um einen eigentlich sehr komplexen Systemsachverhalt abzubilden. Für diese Systemfunktionalität die notwendigen Anwendungsfälle zu finden und basierend darauf die Testfälle zu implementieren, stellt einen sehr hohen Aufwand dar. Der nächste Abschnitt wird dieses Thema genauer be-

trachten. Dort wird gezeigt, wie diese einfache Modifikation des Modells dazu führt, dass dutzende an sehr komplexen Testfällen für das IUT generiert werden können.

5.4 Exploration und Testen

Ist das Modell erstellt, folgt die Exploration. Die Exploration ist jener Vorgang, der das Modell weiter verarbeitet und somit die Grundlage für die Testfälle erstellt. Als Grundbasis für die Exploration dient also das Modell. Von diesem werden nun sogenannte Maschinen erzeugt. Damit eine Exploration vorgenommen werden kann, muss im ersten Schritt eine Konfiguration für die Exploration erstellt werden. Diese Konfiguration wird im sogenannten Cord-Skript abgelegt. In Cord-Skripten wird die gesamte Information, welche SpecExplorer als Eingabe benötigt, um das Modell wie gewünscht verarbeiten zu können, abgelegt. Cord ist als Abkürzung für Coordination (Koordination) gedacht.

5.4.1 Konfiguration

Jedes Cord-Skript benötigt **mindestens** eine Konfiguration, kann aber auch mehrere davon haben. Mit der Konfiguration werden grundlegende Einstellung bezüglich des Explorationsvorgangs und der Testerstellung festgelegt. Eine übliche Standardkonfiguration sieht folgendermaßen aus.

```
config Main
{
    action all SecurityWithBosAdapter;

    switch StepBound = 8192;
    switch StateBound = 8192;
    switch PathDepthBound = 4096;
    switch TestClassBase = "MbtTestBase";
    switch GeneratedTestPath = "..\\..\\Testing\\Nte.Clm.Security.TestSuite";
    switch GeneratedTestNamespace = "Nte.Clm.Security.TestSuite";
    switch TestEnabled = false;
    switch ForExploration = false;
}
```

Ein sehr wichtiges Element ist das Action-Element. Dieses deklariert alle Aktionen, die während des Explorationsvorgangs zur Verfügung stehen. Das Action-Element dient dazu, die verfügbaren Aktionen für die Exploration zu definieren. Die Aktionen können auf drei Arten eingebunden werden

1. Alle Aktionen von Adapter/IUT importieren und deklarieren
   Bei dieser Anweisung werden alle öffentlichen Methoden des Adapters als verfügbare Aktionen während der Exploration deklariert

   ```
   action all AdapterClass;
   ```

2. Einzelne Aktion von Adapter/IUT importieren und deklarieren
   Bei dieser Methode muss jede Methode, die als Aktion während der Exploration zur Verfügung stehen soll, angegeben werden

```
action static ReturnType AdapterClass.Method(ParameterType parameter);
```

3. Abstrakte Aktion deklarieren
   Diese Methode wird verwendet, wenn das IUT für welches Testfälle generiert werden sollen, noch gar nicht existiert. Bei einer abstrakten Deklaration versucht SpecExplorer zwar noch immer, im IUT eine Methode zu finden, die der Deklaration entspricht, gibt aber keine Fehler aus, wenn dem nicht so ist und nimmt einfach an, dass das IUT bei Ausführung der Testfälle eine Methode besitzen wird, welche der Deklaration der Aktion entspricht.

```
action abstract static ReturnType AdapterClass.Method(ParamterType parameter)
```

Zusätzlich zu den Aktionen werden in der Konfiguration eine Vielzahl an Parametern eingestellt. Diese Parameter werden durch die **switch** Anweisung gesetzt. Was diese bedeuten, ist in folgender Tabelle ersichtlich, welche auf [3] basiert.

Parameter Name	Standardwert	Valide Werte	Geltungsbereich	Bedeutung
StateBound	128	Int, None	Exploration, TFG	Legt fest, wie viele Zustände maximal während der Exploration erzeugt werden dürfen. Wird diese Zahl überschritten, wird die Exploration mit einem StateBoundError abgebrochen. Dieser Wert sollte auf maximal 10000 gesetzt werden.
StepBound	128	Int, None	Exploration, TFG	Legt fest, wie viele Zustandsübergänge maximal während der Exploration erzeugt werden dürfen. Wird diese Zahl überschritten, wird die Exploration mit einem StepBoundError abgebrochen. Dieser Wert sollte auf maximal 10000 gesetzt werden.
StepsPerStateBound	1024	Int, None	Exploration, LZT	Anzahl der Zustandsübergänge, welche von einem Zustand ausgehen können. Wird dieser Wert überschritten, wird die Exploration für den Pfad, welcher die Überschreitung hervorgerufen hat, abgebrochen. Dieser Wert sollte auf maximal 10000 gesetzt werden.
PathDepthBound	32	Int, None	Exploration, LZT	Maximale Länge eines Pfades im Graphen der Zustandsmaschine. Wird dieser Wert überschritten, wird die Exploration im Pfad, welcher die Überschreitung hervorgerufen hat abgebrochen.
GeneratedTestPath	"TestSuite"	Pfad (relativ)	TFG	Pfad, in welchem die die Codedatei mit den generierten Testfällen gespeichert wird. Dieser muss als relativer Pfad zu dem Projekt, in welchem sich das Cord-Skript befindet angegeben werden.
GeneratedTestNamespace	"TestSuite"	C# Namensraum	TFG	Namensraum für die generierte Testklasse
ForExploration	False	True/False	Exploration	Legt fest, ob die Maschine exploriert werden kann oder nicht. Wenn die Machine nicht explorierbar ist, wird sie im SpecExplorer-Werkzeugfenster nicht in der Liste der Maschinen angeführt.
TestEnabled	False	True/False	Exploration, TFG	Legt fest, ob die angegebene Maschine für die Erstellung von Testfällen bzw. für Laufzeittests verwendet werden kann.
				TFG... Testfallgenerierung, LZT... Laufzeittests

Tabelle 2: Cord-Konfigurationsparameter [9]

5.4.2 Maschinen

In SpecExplorer ist unter dem Terminus Maschine eine (endliche) Zustandsmaschine zu verstehen. Im Cord-Skript werden diese Maschinen definiert. Jeder Maschineneintrag repräsentiert für SpecExplorer die Anweisung eine Zustandsmaschine, für die im Eintrag angegebenen Bedingungen zu erzeugen.

Nachdem eine Grundkonfiguration - in diesem Fall Main - erstellt ist, kann die erste Maschine erzeugt werden. Die initiale Maschine, die generell erstellt wird, ist eine Maschine für das Modellprogramm. Diese kann mit folgendem Eintrag im Cord-Skript erstellt werden.

```
1 machine ModelProgramBusinessObjectDataService() : Main where ForExploration = true
2 {
3     construct model program from Main
```

[9] Diese Tabelle beinhaltet nicht alle möglichen Parameter, sondern jene, welche sehr häufig benötigt werden. Die vollständige Referenz ist zu finden unter http://msdn.microsoft.com/en-us/library/ee620405.aspx

```
4 }
```

Führt man die Exploration für diese Maschine nun aus, sieht der Graph der Zustandsmaschine folgenermaßen aus.

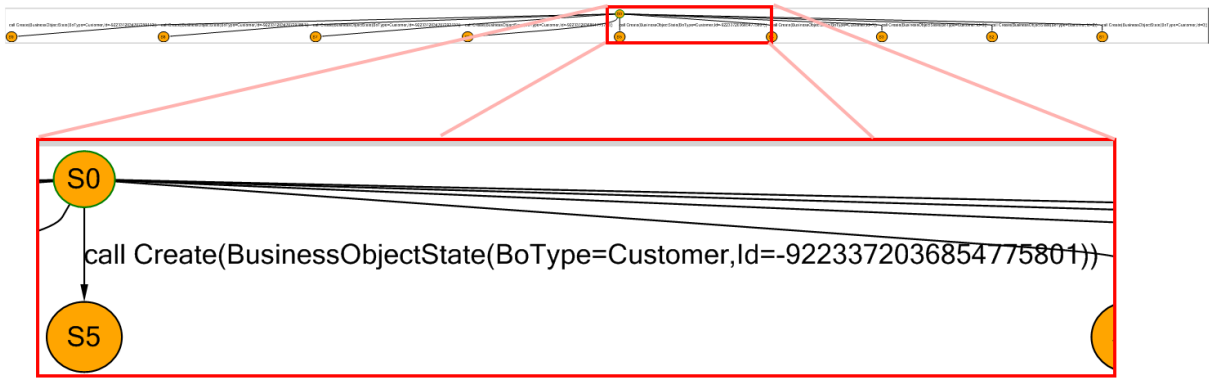

Abbildung 21: Zustandexplosion

Die resultierende Zustandsmaschine ist für eine weitere Verarbeitung unbrauchbar, da diese Zustandsmaschine sofort die eingestellten Grenzwerte bezüglich der erlaubten Anzahl der Zustände bzw. der Zustandstransitionen erreicht, obwohl nur eine einzige Regel ausgeführt wird [10]

Genau dieser Umstand macht den Explorationsvorgang zu einer besonderen Herausforderung. Das Modell ist korrekt, aber dennoch kann es noch nicht verwendet werden, um damit automatisiert Testfälle zu erstellen. Der Grund dafür liegt darin, dass SpecExplorer beim Auswerten der Bedingungen folgendes Verhalten inne hat. Erstens wendet SpecExplorer eine Regel immer wieder an, sofern ihre Bedingungen erfüllt sind. Des Weiteren versucht er für die Parameter der Regel **alle** gültigen Kombinationen zu erzeugen. Dieses Verhalten von SpecExplorer führt nun im obigen Beispiel dazu, dass immer wieder CreateBusinessObject aufgerufen wird bis die Grenzwerte für die Anzahl der Zustände bzw. für die Anzahl der Transitionen erreicht wird. Dieses Verhalten wird sich bei so ziemlich allen realen Modellen sofort zeigen.

Um diesem Verhalten entgegen zu wirken, gibt es die sogenannten Szenario-Kontrolltechniken. Diese dienen dazu, bestimmte Testszenarien zu erzeugen mit welchen dann der Zustandsraum deutlich eingeschränkt werden kann. Bei der Anwendung solcher Szenarien wird in der Literatur vom sogenannten Versuchszweck gesprochen. Ein solcher Versuchszweck könnte

[10] Für das gezeigte Beispiel wurde die Anzahl der erlaubten Zustände auf einen Wert von 10 gesetzt, was für eine reale Exploration bei Weitem zu niedrig ist. Würde man den Wert allerdings auf üblicherweise verwendete Werte, also größer/gleich 128 stellen, wäre der Gesamtgraph nur mehr als schwarze Linie zu erkennen.

für den obigen Service z.B. folgendermaßen lauten.

„Für **zwei** Kunden müssen von jedem im System vorhandenen Businessobjekttyp **zwei** Instanzen der selbigen angelegt und ab diesem Zeitpunkt auch wieder abfragbar sein.“.

Diese Art des Versuchszwecks ist eine sehr einfache aber äußerst effiziente Methode, die Zustandsexplosion zu unterbinden. Durch die Einschränkung der Anzahl der zu erzeugenden Instanzen wird die Zustandsexplosion vermieden, denn die endliche Anzahl der Instanzen bedingt, dass auch der Zustandsraum endlich wird. Das führt zwar dazu, dass das System nicht mehr vollständig getestet wird aber nun Aspekte, auf den man sich konzentriert weiterhin behandelt wird. Die Aspekte die mit obigem Versuchszweck bedient werden sind folgende:

- Zwei Kunden müssen gleichzeitig im System arbeiten können
- Die Änderungen eines Kunden dürfen sich nicht auf die Daten des anderen Kunden auswirken
- Objekte, die im System angelegt wurden, müssen auch wieder abgefragt werden können.

Die Vereinfachung des Versuchszwecks besteht also in folgenden Punkten

- Anstatt das gewünschte Verhalten für beliebig viele Kunden zu testen, wird angenommen, dass das System das Verhalten auch dann für beliebig viele Kunden aufweist, sofern das gewünschte Verhalten bereits mit zwei Kunden nachgewiesen werden kann
- Der Kunde kann von jedem Unterobjekt-Typ lediglich zwei Instanzen erzeugen.

Natürlich ist durch solche Maßnahmen die Testabdeckung eines Systems verringert, und durch die getroffenen Annahmen werden Aussagen bezüglich des Systems getroffen, welche im Grunde genommen nicht nachgewiesen wurden. Dies muss aber in Kauf genommen werden, da folgende Aussage unumstritten Gültigkeit hat.

„Nur einfache Programme lassen sich vollständig testen. Ein vollständiger Test typischer Softwareprodukte, bei dem beispielsweise alle Eingabe-, Ausgabewerte, ihre Kombination und ihre zeitlichen Abhängigkeiten getestet werden ist unmöglich.“ [34]

SpecExplorer bietet zwei Mechanismen für die Szenariokontrolle, die Parameterselektion und das Slicing.

Parameterselektion Mit der Parameterselektion kann man SpecExplorer dazu bewegen, für die Parameter einer Regel zusätzliche Bedingungen zu definieren, welche im Modell selbst nicht spezifiziert sind. Wenn man das Beispiel des vorhergehenden Modells heranzieht, erkennt man, dass für die ID-Eigenschaft des zu erzeugenden Business-Objekts alle möglichen (abstrusen) Werte vorkommen. Die Spezifikation der Service Schnittstelle [11] gibt

[11] Implizite Spezifikationen, dass bestimmte Systeminterna während der Modellierung bekannt sind

aber vor, dass diese Eigenschaft für das Erzeugen eines Objekts komplett irrelevant ist, da sie ohnedies immer überschrieben wird. Daher kann man den möglichen Zustandsraum für die Regel "CreateBusinessObject" so einschränken, dass die ID des BusinessObjekts immer auf "0" gesetzt wird.

Um eine Parameterselektion durchzuführen, gibt es in SpecExplorer mehrere Möglichkeiten.

- Gültigen Parameterraum bei Deklaration einer Aktion in einer Konfiguration spezifizieren
  In diesem Fall wird die Deklaration einer Aktion um zusätzliche Bedingungen innerhalb des Cord-Skripts erweitert. Im Wesentlichen findet in diesem Fall tatsächlich eine Erweiterung der Modell-Regel statt, erkennbar dadurch, dass die zusätzlichen Bedingungen als eingebetteter C# Code spezifiziert werden. Es ist möglich, in Cord-Skripten C# Code zu definieren, welcher dann während der Exploration zur Laufzeit dem Modell angefügt wird. C# Code kann mittels öffnendem Code Tag **"{."** und schließendem Äquivalent **".}"** eingefügt werden. Möchte man nun die ID-Eigenschaft für das Businessobjekt aus vorhergehendem Beispiel auf 0 restriktieren, kann dies durch folgende Deklaration im Cord-Skript erreicht werden.

```
1  config Restricted : Main
2  {
3    action static BusinessObjectState Adapter.CreateBusinessObject(BusinessObjectState bo)
4      where
5      {.
6        Condition.IsTrue(bo.Id == 0);
7        Condition.IsTrue(bo.Type = BusinessObjectTypeState.Customer);
8      .}
9  }
```

  Zwar könnten solche Bedingungen auch direkt im Modell ausgedrückt werden, jedoch ist es oft notwendig, die Einschränkungen, welche für bestimmte Szenarien gemacht werden nicht in das Modell zu ziehen. Bei dem hier gezeigten Beispiel könnte dies zwar durchaus noch angebracht sein, da das Verhalten des Modells durch diese Einschränkung nicht grundlegend verändert wird, dennoch ist es keine definierte Spezifikation des Modells, dass die ID beim Erzeugen eines Objekts 0 sein muss und ein Modell sollte nur jene Dinge modellieren, welche auch tatsächlich so gelten. Daher ist es auch in diesem Fall bereits sinnvoll, im Modell die Einschränkung nicht zu modellieren und diese erst mittels der Szenariokontrolle im Cord-Skript zu deklarieren, da das Modell somit genereller und näher am realen System ausgerichtet ist.

- Gültigen Parameterraum in einer Slice-Maschine spezifizieren
  In diesem Fall wird die Einschränkung für den Parameterraum nicht innerhalb einer Konfiguration vorgenommen, sondern in einer einzelnen Slice-Maschine. Was Slice-Maschinen sind, wird in 5.4.2 erläutert. Im Wesentlichen ist die Funktionsweise die selbe, nur der Geltungsbereich - in diesem Fall eine Maschine - und die Syntax sind unterschiedlich. In diesem Fall würde die Deklaration wie folgt aussehen.

```
machine Scenario() : Main
{
  let BusinessObjectState bos
    where
      {.
        Condition.IsTrue(bos.Id = 0);
        Condition.IsTrue(bo.Type = BusinessObjectTypeState.Customer);
      .}
  in
    CreateBusinessObject(bos);
}
```

- Konkreten Parameter in einer Slice-Maschine spezifizieren
  Die letzte Methode besteht darin, konkrete Instanzen für die Parameter anzugeben. Im Gegensatz zu den vorhergegehenden Beispielen wird hier aber nicht mit eingebettetem C# Code gearbeitet. Stattdessen wird der Parameter mittels Cord-Syntax definiert.

```
machine Scenario() : Main
{
  CreateBusinessObject(BusinessObject(Id = 0, BoType = Customer));
}
```

  Es ist aber nicht zwingend notwendig, alle Parameter einer Methode zu spezifizieren. Zu diesem Zweck gibt es den "_" Operator. Dieser besagt, dass für einen Paramter alle durch das Modell gültigen Möglichkeiten eingesetzt werden sollen.

```
machine Scenario() : Main
{
  CreateBusinessObjectAndAssignToTarget(BusinessObject(Id = 0,
    BoType = BusinessObjectTypeState.Facility), _);
}
```

  Dieses Beispiel legt fest, dass für den ersten Parameter eine konkrete Instanz bei der Exploration herangezogen werden soll, während für den zweiten Parameter alle möglichen Werte, welche der Modellspezifikation entsprechen, herangezogen werden sollen.

Slicing Das in diesem Abschnitt beschriebene Slicing darf nicht mit dem allgemein mit diesem Terminus verknüpften Program-Slicing verwechselt werden. "Program slicing is a method used by experienced computer programers for abstracting from programs. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior" [43].

Slicing hat in diesem Fall einen ähnlichen Aspekt, erfüllt aber nicht obige Definition. Im Wesentlichen stellt das Slicing hier eine Reduktion des Zustandsraums auf den Testzweck dar. D.h., wenn also der Testzweck lautet „Ein Objekt welches mit einem bestimmten Typ erzeugt wurde muss bei der Abfrage nach Objekten mit diesem Typ wieder geliefert werden“ ist es für den Testzweck nicht dienlich, auch die eine Methode, welche ein Business-Objekt

verändert mit einzubeziehen. Daher kann diese Methode aus der Maschine für diesen speziellen Testzweck ausgenommen werden, was dann zu einer Reduktion des Zustandsraums führt.

Um einen Slice zu erzeugen, werden in SpecExplorer Szenario-Maschinen eingesetzt. Diese Szenario-Maschinen sind endliche oder unendliche Zustandsmaschinen, welche mit der Ausgangszustandsmaschine kombiniert werden. Szenario-Maschinen definieren wann Regeln bzw. in welcher Reihenfolge sie aufgerufen werden dürfen. Folgendes Beispiel zeigt eine grundlegende Szenario-Maschine

```
machine Scenario() : Main() where ForExploration = true
{
  CreateBusinessObject;
  CreateBusinessObject;
  GetByType*;
}
```

Die Exploration liefert folgendes Ergebnis.

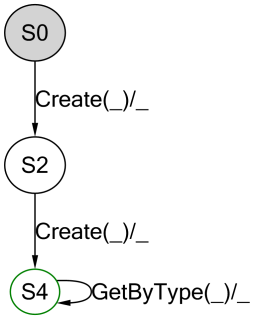

Abbildung 22: Explorationsergebnis einer Szenario-Maschine

Szenario-Maschinen sind nach der Exploration dadurch als solche erkennbar, dass sie grundsätzlich keine konkreten Werte für die Parameter besitzen, sondern nur den "_" Platzhalter haben, welcher ja besagt, dass alle möglichen Parameter eingesetzt werden sollen. SpecExplorer generiert in diesem Fall eine Szenario-Maschine, da keinerlei "Construct"-Anweisung innerhalb der Maschine vorkommen. Es wird also von der deklarierten Maschine tatsächlich nur die grafische Repräsentation selbiger erzeugt.

Diese Szenario-Maschine kann nun mit dem Modellprogramm kombiniert werden, um so eine neue Zustandsmaschine mit reduziertem Zustandsraum zu generieren, welche dann einem bestimmten Testzweck entspricht [12].

```
machine Slice() : Main where ForExploration = true
{
  ModelProgram || Scenario
}
```

[12] Nun wird wieder eine Zustandsmaschine erzeugt, da die Maschine ModelProgram eine "Construct"-Anweisung enthält

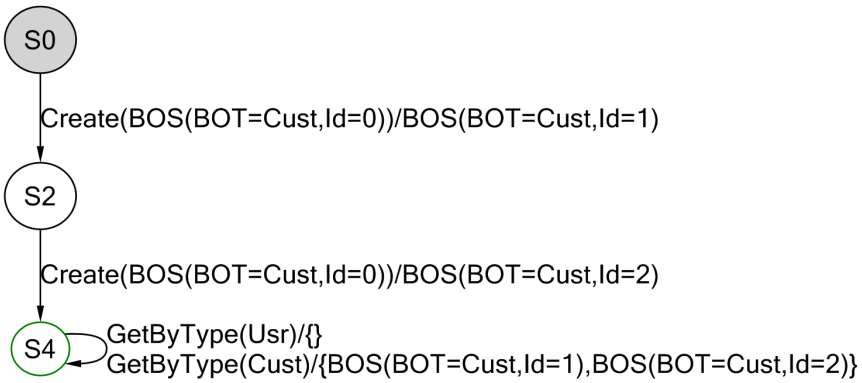

Abbildung 23: Slice

Diese Zustandsmaschine wurde durch eine synchrone parallele Komposition von Modellprogramm und Szenario-Maschine erreicht. SpecExplorer kennt eine Vielzahl an Operatoren um Zustandsmaschinen zu kombinieren, welche in folgendem Abschnitt näher beschrieben werden.

- Strikter Sequenz Operator ";"
  Die Operanden werden zu einer Sequenz verknüpft. Der Operator ist nicht-kommutativ und assoziativ.

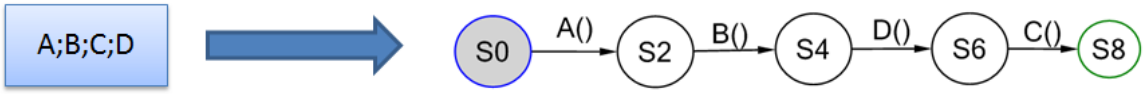

Abbildung 24: Strikter Sequenzoperator

- Loser Sequenz Operator "->"
  Bei einem schwachen Sequenzoperator ist es so, dass die angegebenen Operanden wieder zu einer Sequenz verbunden werden, aber SpecExplorer fügt zusätzlich alle möglichen Transitionen in dem Zustand, der zum Sequenzaufbau für die beiden Operanden erzeugt wird, ein. Der Operator ist nicht-kommutativ und nicht-assoziativ.

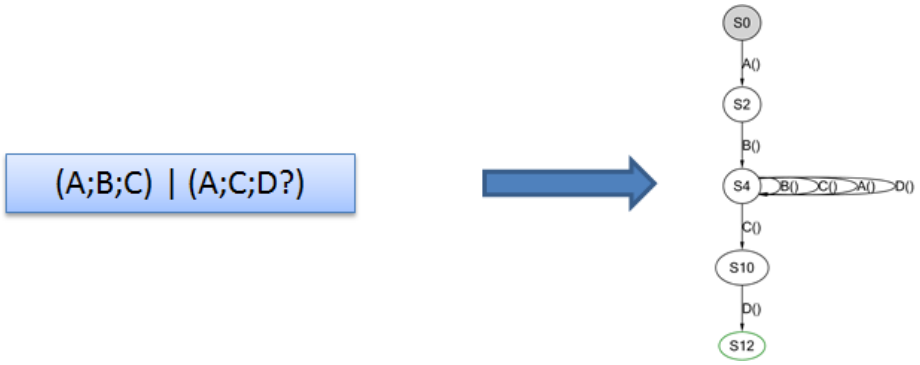

Abbildung 25: Loser Sequenzoperator

- Vereinigung "|"
  Bei der Vereinigung von zwei Zustandsmaschinen kommt jeder Pfad aus den beiden

Ursprungsmaschinen in der neuen Maschine wieder vor. Der Operator ist kommutativ und assoziativ.

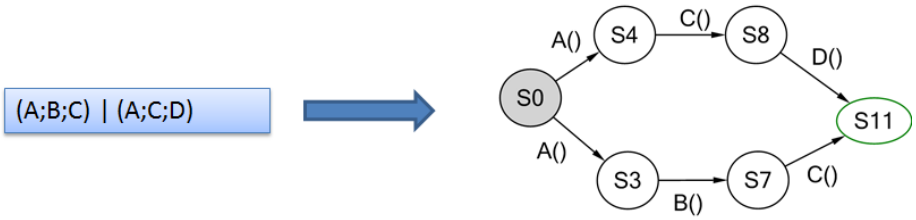

Abbildung 26: Vereinigung

- Optional Operator "?"
  Der Operand ist optional.

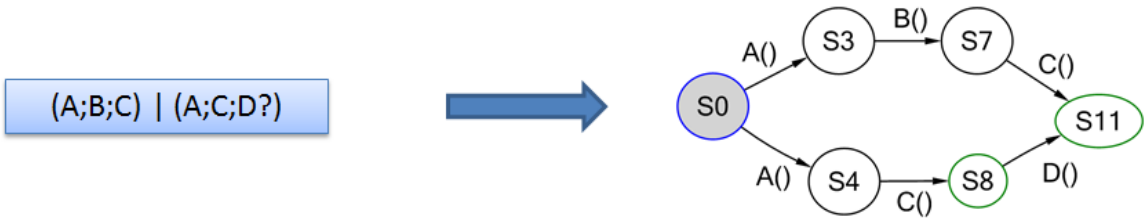

Abbildung 27: Optional Operator

Durch den Operator wurde nun auch der Zustand S8 zu einem Endzustand (Accepting State).

- 1..N Wiederholungen "+"

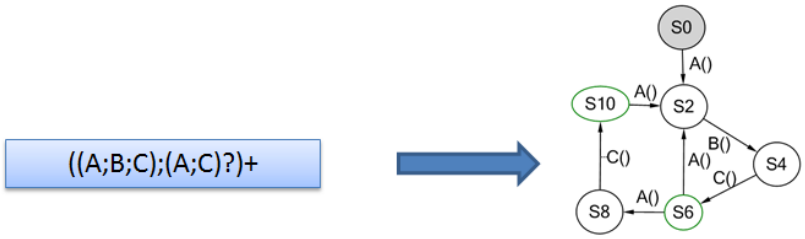

Abbildung 28: 1..N Wiederholungen

- 0..N Wiederholungen "*"

Abbildung 29: 0..N Wiederholungen

- N Wiederholungen **"{N}"**

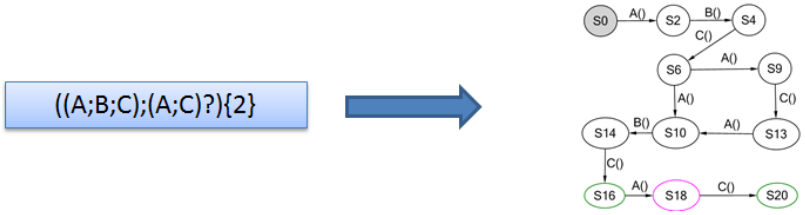

Abbildung 30: N Wiederholungen

- Mindestens M Wiederholungen **"{M,}"**

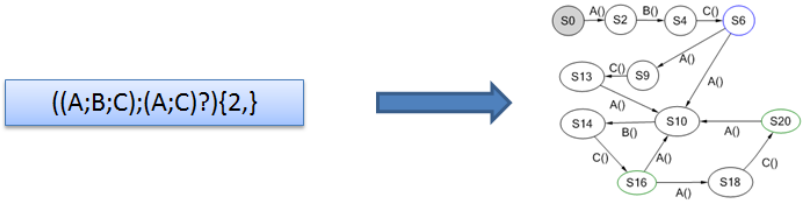

Abbildung 31: Mindestens M Wiederholungen

- M..N Wiederholungen **"{M,N}"**

- Jegliche Aktion **"_"**

Abbildung 32: Jegliche Aktion

- Jegliche Aktion in beliebiger Wiederholung **"..."**

Abbildung 33: Jegliche Aktion in beliebiger Wiederholung

- Negation **"!"**

Abbildung 34: Negation

- Permutation **"&"**

Es werden alle möglichen Kombinationen der Operanden erzeugt. Operator ist kommutativ.

Abbildung 35: Permutation

- Synchrone parallele Komposition "||"
  Dieser Operator erzeugt eine Zustandsmaschine, in welche nur jene Transitionen aufgenommen werden, welche für die beiden Ursprungszustandsmaschinen synchronisiert werden können. Synchronisation bedeutet in diesem Fall, dass die Transition in beiden Zustandsmaschinen im selben Schritt vorkommen müssen.

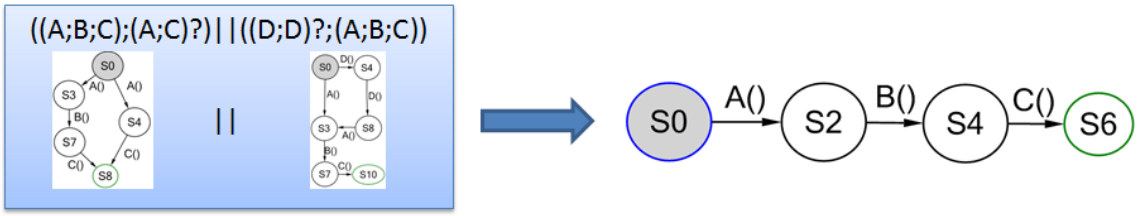

Abbildung 36: Synchrone Parallele Komposition

Im Beispiel ist nur die Synchronisation des Transitionspfades (A;B;C) möglich, dann werden die beiden Maschinen asynchron und keine weiteren Transitionen können mehr aufgenommen werden. Eine vereinfachte Darstellung der Funktionweise des Operators ist folgender Tabelle zu entnehmen

Zustand		Verfügbare Transitionen		
M0	M1	M0	M1	Synchrone Transitionen
S0	S0	A	A,D	A
S3	S3	B	B	B
S7	S7	C	C	C
S4	S3	C	B	

Tabelle 3: Synchrone parallele Kompositions-Arbeitsweise

- Verschachtelte parallele Komposition "|||"

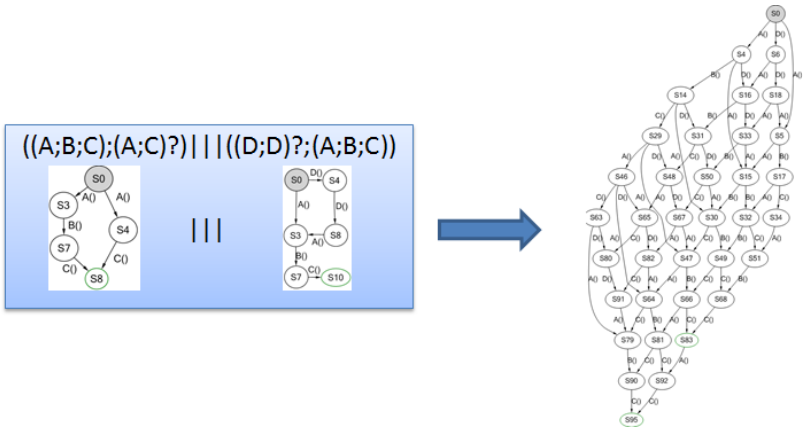

Abbildung 37: Verschachtelte parallele Komposition

Mit Hilfe dieser Operatoren ist es nun möglich, komplexe Szenario-Maschinen zu erzeugen. Durch Kombination dieser Maschinen mit einem Modellprogramm werden dann Slices für ein Szenario erzeugt. Die jeweiligen Maschinen-Typen (Szenario-, Modell-, Slice-Maschinen) sind von der technischen Sichtweise ein und dasselbe, nämlich Zustandsmaschinen die dann über bekannte Operatoren für diese Zustandsmaschinen kombiniert werden können. Diese Klassifizierung in Maschinen-Typen soll lediglich der gedanklichen Trennung des Entwicklers dienen.

Abschließend sollen an dieser Stelle noch die Maschinen aufgezeigt werden, welche für das bereits mehrfach erwähnte Szenario „Objekte, welche erzeugt wurden müssen auch wieder abgefragt werden können“ notwendig sind.

```
machine ModelProgramBusinessObjectDataService() : Main where ForExploration = true
{
    construct model program from Main
}

machine ScenarioCreateTwoCustomers() :Main where ForExploration = true
{
    Create; Create
}

machine ScenarioCreateBos() : Main where ForExploration = true
{
    (CreateAndAssignToTarget; GetByType){3}
}

machine ScenarioCombined() : Main where ForExploration = true
{
    ScenarioCreateTwoCustomers ; ScenarioCreateBos
}

machine SliceCombined() : Main where ForExploration = true
{
    ModelProgramBusinessObjectDataService  || ScenarioCombined
}
```

Das Resultat für dieses Szenario sieht dann wie folgt aus:

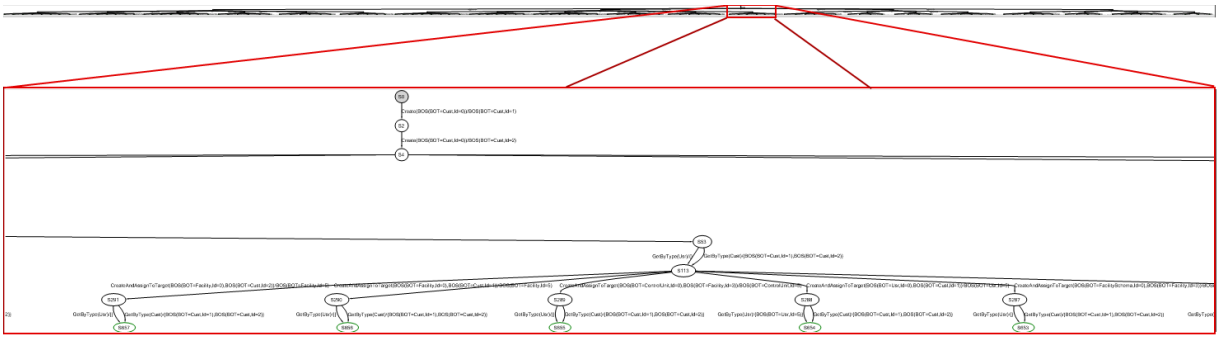

Abbildung 38: Explorationsergebnis bei testbarem Szenario

Der Zustandsautomat ist zwar noch immer sehr groß (735 Zustände, 880 Transitionen), aber im Gegensatz zu dem Explorationsergebnis, welches nicht für die Testfallgenerierung geschaffen war, da es unendliche groß werden konnte, ist dieses Ergebnis hier ein endlicher Zustandsautomat. Anstatt Zuständen, welche Explorationsgrenzen sprengen und daher zu BoundSteps führen in welchen die Exploration einfach abgebrochen wird, werden hier Accepting States erreicht.

Accepting State Als Accepting State werden jene Zustände bezeichnet, welche als gültige Modellzustände gesehen werden können. Für viele Modelle ist das im Grunde jeder Zustand der anhand gültiger Transitionen erreicht werden kann.

```
1 static int state = 0;
2 [Rule] public static void Foo()
3 {
4     Condition.IsTrue(state < 3);
5     state = state + 1;
6 }
```

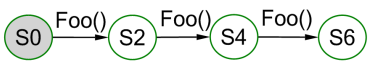

Abbildung 39: Accepting States

Die Exporation liefert eine Zustandsmaschine in welcher alle Zustände Accepting States sind, erkennbar durch den grünen Rahmen eines Zustands. D.h., egal in welchem Zustand die Testsequenz abgebrochen wird, in jedem dieser Fälle handelt es sich um einen erfolgreichen Testablauf.

Der Accepting State wird vor allem dazu verwendet die Testfallgenerierung zu steuern. Der Accepting State hat die Aufgabe jene Zustände im Modell zu spezifizieren, für welche der Transitionspfad welcher erzeugt wird, bis dieser Zustand erreicht wird, als ein gültiger und sinnvoller Testablauf angesehen werden kann. Um festzulegen, welche Zustände Accepting States sind, gibt es zwei Techniken. Anders ausgedrückt, alle Transitionspfade, welche einen Accepting State erreichen sind gültige Pfade.

Zum eine die Definition einer Accepting State - Bedingung im Modell. Diese kann folgendermaßen vorgenommen werden.

```
1  [AcceptingStateCondition]
2  static bool AcceptingState()
3  {
4      return state == 2;
5  }
```

Dies führt zu folgendem Explorationsergebnis.

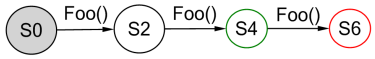

Abbildung 40: Exploration mit Accepting State Bedingung

Nur mehr der Zustand S4 in welchem gilt dass "state" gleich zwei ist, ist ein Accepting State. Daher ist der Transitionspfad S0 → S1 kein gültiger, da kein Accepting State erreicht wird. Daher wird für diesen Pfad auch kein kein Testfall erzeugt werden. Da die Modellregel aufgrund ihrer Bedingungen aber weiter angewendet werden kann, werden weitere Modellzustände generiert, welche dann die Klassifikation Error-State mit der Untereinordnung NonAcceptingEnd erhalten, da jene Transitionspfade die über Zustand S4 hinaus gehen nicht mehr in der Lage sind, einen Accepting State zu erreichen.

Die zweite Methode, um bestimmte Accepting States zu definieren, besteht wiederum darin, Szenario-Maschinen mit dem Modellprogramm zu kombinieren. Wenn man nun ein Szenario bildet [13]:

```
1 machine Slice() : Main where ForExploration = true
2 {
3     BusinessObjectDataServiceModelProgram || (Foo{3})
4 }
```

Durch das Szenario ist nun vorgegeben, dass "Foo" exakt drei Mal aufgerufen werden muss. Daher weiß SpecExplorer nun, dass die ersten beiden Aufrufe nicht in die Kategorie "Accepting State" fallen können, da dass Szenario erst beim dritten Aufruf von "Foo" erfüllt ist.

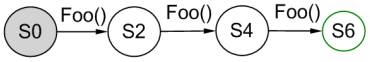

Abbildung 41: Accepting State durch Szenario definiert

5.4.3 Testfallgenerierung

Dieser Abschnitt beschäftigt sich nun mit dem finalen Schritt des modellbasierten Testens, der Testfallgenerierung. Als Grundlage für selbige dienen endliche Zustandsmaschinen,

[13] Hier wird eine implizite Szenario-Definition vorgenommen, da das Szenario nicht in eine eigene Maschine ausgelagert wurde

welche fehlerfrei sind und lediglich „normale“ Zustände oder Zustände welche "Accepting States" sind aufweisen.

In diesem Fall kann auf Grundlage dieser Szenario-Maschinen eine Test-Maschine erzeugt werden. Eine Testmaschine wird erzeugt, indem die Maschineneigenschaft „TestEnabled“ auf den Wert wahr gesetzt wird. Eine Testmaschine erzeugt dann aus den ursprünglichen Maschinen Testsequenzen. Im Gegensatz zu den ursprünglichen Maschinen haben diese Sequenzen keine Verzweigungen mehr. Es werden alle möglichen Pfade der ursprünglichen Maschine evaluiert und jeder dieser Pfade als eine Testsequenz abgebildet. Durch dies Transformation werden die Testsequenzen in der Testnormalform generiert was bedeutet, dass in den Tests keine Entscheidungspunkte mehr vorhanden sind.

```
machine TestSuiteSecuirtyWithBosSetInstanceRights() :
   Main where ForExploration = true,
   TestEnabled = true
{
    construct test cases
       where Strategy = "LongTests" for
       SliceSecuirtyWithBosTwoCustomersSetInstanceRights()
}
```

Abbildung 42: Testsequenz resultierend aus der Short-Strategie

In der Teststrategie kann spezifiziert werden, welche Strategie zur Erzeugung der Tests verwendet werden soll. Aktuell gibt es hier die beiden Auswahlmöglichkeiten "ShortTests" und "LongTests". Bei "ShortTests" erzeugt SpecExplorer eine große Anzahl kurzer Testsequenzen während bei der "LongTests" Strategie versucht wird, eine geringere Anzahl an Testsequenzen zu erzeugen, welche aber eine höhere Anzahl an Einzelschritten aufweisen. Abbildung **??** zeigt die resultierenden Testsequenzen, wenn die Strategie "ShortTests" ver-

wendet wird. Wenn die Strategie "LongTests" bei der Generierung der Testfälle für selbigen Slice verwendet wird, werden folgende Testsequenzen erzeugt.

Abbildung 43: Testsequenzen resultierend aus der Long-Strategie

Das hier gezeigte Beispiel verdeutlicht, dass die „LongTest “ - Strategie nicht zwangsläufig zu längeren Testsequenzen führt. Die Definitionen der jweiligen Strategien lauten:

"The "ShortTests" strategy generates a complete test case as soon as it finds an accepting state with a path that includes at least one transition step that has not already been tested. To do this, it performs a depth-first search of the exploration graph, starting from an initial state, adding edges in the path to the current sub-graph (called a "test case"). When an accepting end state (an accepting state with no outgoing uncovered edges) is reached, the current test case is cut and added to the output, and a new test case starts from an initial state. Observable choice nodes cause the algorithm to explore all outgoing paths in parallel until they are all cut. Each (controllable) loop is expanded to a single iteration. This strategy tends to generate more but shorter test cases." [4]

"The "LongTests" strategy is based on a general tour of the exploration graphs in which it tries to find the most complete (that is, longest) path from an initial state that covers as many transition steps as possible before ending with an accepting state that is as deep as possible. This strategy tends to generate fewer but longer test cases." [4]

Daher sind Testfälle welche mittels der "LongTest" - Strategie erzeugt werden im generellen länger, es gibt aber keinerlei Garantie dafür.

6 Ergebnisse, Auswirkungen und Konsequenzen

6.1 Entwickler als Tester

Wie bereits erwähnt, mussten bei der Entwicklung von NTE.CLM die Entwickler auch als Softwaretester fungieren. Eigens für diesen Zweck abgestellte Personen gab es nicht. Besonders in diesem Bereich konnte das modellbasierte Testen gute Dienste leisten.

Als starker Vorteil des modellbasierten Testens konnte in diesem Umfeld die von der Implementierung losgelöste Betrachtung des Systems hervorstechen. Der Grund für diese Loslösung besteht vor allem in zwei Aspekten des modellbasierten Testens. Zum einen benötigt das Erstellen von Modellen wesentlich mehr Zeit und ein wesentlich strukturierteres Vorgehen als das „simple“ Erstellen manueller Testfälle, was wiederum dazu führt, dass diese Testtätigkeit nicht - oder nur sehr schwer - direkt an die Implementierung angehängt werden kann. Dieser Umstand bedingt also, dass zwischen Implementierungszeitpunkt und Testzeitpunkt eine gewisse Zeitspanne liegt, welche dazu führt, dass der Entwickler Implementierungsdetails zum Testzeitpunkt nicht mehr (so gut) kennt, und die Tests somit nicht mehr an diesen ausgerichtet werden können.

Zum anderen ist das Erstellen des Modells eine grundsätzlich andere Tätigkeit als das Erstellen Use-Case getriebener Testfälle. Modelle versuchen Systemverhalten zu modellieren, und überlassen es einem Tool die Use-Case orientierteren Testfälle dazu zu generieren. Das Modellieren eines Systems ist ein Vorgang, welcher eine Mechanik darstellt, bei welcher es grundsätzlich nicht mehr möglich ist, auf spezielle Implementierungsdetails Rücksicht zu nehmen. Der Entwickler wird sozusagen gezwungen, eine Systemperspektive einzunehmen, welche mit jener während der Systemimplementierung nicht mehr vereinbar ist. Dieser Effekt wird weiter verstärkt durch die notwendigen Abstraktionen, die ein Modell mit sich bringt, um beherrschbar zu werden. Die notwendigen Abstraktionen bedingen einfach, dass gewisse Dinge, die ein Entwickler testen möchte einfach nicht betrachtet kann, da er sie im Modell einfach nicht miteinbeziehen kann, da es ansonsten zu groß werden würde.

Eine weiterer Vorteil des modellbasierten Tests ist es, dass der Testfallerstellungsvorgang in gewisser Hinsicht umgekehrt wird. Bei der manuellen Testfallspezifikation kennt der Entwickler die Spezifikation des Systems und versucht nun, Testfälle zu erstellen, welche innerhalb - oder bei Fehlerfalltests auch außerhalb - dieser Spezifikation agieren und mit diesen Fällen eine möglichst große Abdeckung bezüglich der Spezifikation zu erreichen. Bei modellbasierten Tests hingegen gibt der Entwickler die Spezifikation vor und SpecExplorer versucht entsprechend dieser Spezifikation Testfälle zu generieren, um diese vollständig abzudecken. Diese Vorgehensweise ist wesentlich natürlicher als die des klassischen manuellen Tests. Für Menschen ist es relativ einfach, Bedingungen aufzustellen, unter welchen ein System das gewünschte Verhalten aufweist. Allerdings ist es für Menschen sehr schwierig, den Parameterraum (die Zustände und Transitionsparameter), welcher diese Bedingungen erfüllt, zu evaluieren. Und genau hier liegt die Stärke von SpecExplorer. Mit Hilfe seiner

Bedingungs-Löser kann er **alle** Zustände und Parameter auflösen und somit gewährleisten, dass auch wirklich alle notwendigen Testsequenzen erzeugt werden, um eine vollständige Abdeckung zu erreichen. Weiters erzeugt SpecExplorer auch die möglichen Permutationen der Testsequenzen, ein Aspekt, welcher bei der manuellen Erstellung von Testfällen oftmals keine Betrachtung erfährt.

Als Konsequenz aus diesem Verhalten ergibt sich, dass beim Test Aspekte des Systems als ganzes betrachtet werden, anstatt dass Testfälle für Systemteile erstellt werden. Man begibt sich also wesentlich mehr in das Umfeld der Systemanalytik, welche eine wesentlich umfangreichere Betrachtung des Systems zulässt.

Diese Eigenschaft des modellbasierten Tests mit SpecExplorer haben dieses Testtool zu einem in diesem Szenario (Entwickler = Tester) unverzichtbaren Werkzeug für den Test des NTE.CLM-Systems gemacht. Die beiden Services (IBusinessObjectService, ISecuirtyService) für welche im Rahmen dieser Arbeit modellbasiertes Testen angewandt wurde, waren jene Services, die einen Großteil der manuellen Testfälle für sich beanspruchten. Etwa 40% aller manuell erstellten Testfälle entfielen auf diese (insgesamt bestand das System zu diesem Zeitpunkt aus elf Services). Von diesen 40% entfielen jeweils 50% auf einen der beiden Services. Somit waren diese beiden Services die beiden am besten getesteten Services des Systems (mit händisch erstellten Testabläufen. Die manuellen Testfälle machten zu diesem Zeitpunkt mehrere hundert Testfälle aus. Allein die Testfälle für den ISecurityService entsprachen einigen tausend Zeilen Code. Zusätzlich wurde die jeweils aktuelle Version des Systems bereits für mehrere Wochen wegen dem Betrieb in einer Testanlage und durch manuelle Systemtests betrieben. [14]). Dennoch wurden für beide Services bereits mit den ersten generierten Testfällen Implementierungsfehler selbiger durch SpecExplorer gefunden. Zwei Fehler wurden im IBusinessObjectDataService aufgedeckt, welche als „leicht“ klassifiziert wurden. Insgesamt sechs Fehler konnte im ISecurityService gefunden werden. Zwei davon konnten als „kritische“, einer als „schwerer“ und drei davon als „leichte“ Fehler klassifiziert werden. Die beiden kritischen Fehler hätten jedenfalls ein Produktupdate, zur Korrektur dieser Fehler bedingt.

Der Grund, dass diese Fehler nicht aufgedeckt wurden, liegt darin, dass nur ganz bestimmte Aufrufreihenfolgen der Servicemethoden dazu führten, dass diese Fehler auftraten, und diese Ablaufreihenfolgen nur dann zu dem Fehler führten, wenn vor dem Ablauf ein ganz bestimmter Systemzustand herrschte. Die Wahrscheinlichkeit durch manuelle Testfälle bzw. der benutzerorientierten Nutzung des Systems während des Systemtests genau auf diese Testsequenz ist de facto gleich 0. Folgende Frage hat in diesem Fall wohl ihre Berechtigung „Wie findet man Bugs, nach denen man nicht sucht?“. Im Fall von SpecExplorer gilt wohl: „Indem ich diese Aufgabe ein Tool erledigen lasse “.

Allein dieser Sachverhalt zeigt, dass modellbasiertes Testen einen echten Mehrwert darstellt und schlicht und ergreifend wesentlich dazu beitragen kann, die Qualität von Softwaresystemen in beachtlichem Ausmaße zu erhöhen.

[14] Genaue Zahlen dürfen im Rahmen dieser Arbeit nicht belegt werden

Wie in der Softwareentwicklung eigentlich immer, hat modellbasiertes Testen (mit SpecExplorer) aber nicht nur Vorteile sondern auch Nachteile.

Der wohl größte Nachteil ist die wesentlich höhere Komplexität, welche modellbasiertes Testen mit sich bringt, da es sich bei der Modellierung in gewisser Weise um eine vollständige Systemanalyse handelt. Die Qualität der schlussendlich generierten Testfälle hängt sehr stark von der Qualität des Modells ab. Wenn ein Modell, die IUT zu sehr abstrahiert, zu wenig abstrahiert oder wichtige Systemaspekte einfach vernachlässigt, werden auch die daraus generierten Testfälle diese Eigenschaften inne haben.

Weiters ist ein Problem, dass man den Tools mitunter teilweise zu sehr vertraut. Während der Entwicklung eines ersten Modells für den ISecurityService war das Modell fehlerhaft. Im Wesentlichen haben die Testfälle zwar das IUT mit den im Modell spezifizieren Regeln getrieben, aber die Regeln zur Auswertung der Resultate dieser Serviceaufrufe waren fehlerhaft, was zur Folge hatte, dass diese überhaupt nicht evaluiert wurden. D.h. die Serviceaufrufe wurden zwar durchgeführt, aber die Rückgabewerte der Aufrufe wurden nicht evaluiert und somit wurden die Testfälle stets als „Korrekt“ klassifiziert. Was nun also passierte war, dass ein sehr großes TestSuite fehlerfrei durchlief und deshalb die Korrektheit des Systems impliziert wurde. Tatsächlich war es aber so, dass die durch SpecExplorer generierten Testfälle nichts anderes testeten, als dass die Methoden immer wieder mit den gleich parametrisierten Serviceaufrufen aufgerufen werden können, ohne dass diese bei diesen Folgeaufrufen abstürzen, was natürlich nicht der Intention während der Modellerstellung entsprach.

Ein weiterer negativer Punkt ist, dass MBT mit SpecExplorer nur bedingt dazu geeignet ist, Fehlerfälle zu testen. Generell werden Fehlerfälle in C# mit Ausnahmen behandelt. SpexExplorer kennt einerseits weder Mechanismen, um Ausnahmen in einem Modell zu behandeln, noch Konstrukte um bewusst die im Modell gesetzten Bedingungen zu verletzen. Daher müssen auch alle Fehlerfälle welche getestet werden sollen, vollständig modelliert werden. Die Ausnahmen müssen zusätzlich noch im Adapter zu Fehlerrückgabewerten umgewandelt werden, damit diese im Modell behandelt werden können. All dies macht es sehr umständlich - mitunter sogar gänzlich unrentabel - Fehlertests mit SpecExplorer zu behandeln.

6.2 Verteilte Architektur

Abschließend soll nun das modellbasierte Testen in Bezug auf den Aspekt des verteilten Systems betrachtet werden. Ein Aspekt, der durch MBT - zumindest mit SpecExplorer - gut bedient werden kann ist die Heterogenität. Die Modelle werden zwar alle unter der gleichen Plattform entwickelt, dennoch kann SpecExplorer auch heterogene Systeme treiben. Aufgrund des Adapter-Musters, welches bei MBT aufgrund der technischen Voraussetzungen eigentlich immer Anwendung findet, können auch die verschiedensten Module miteinander getestet werden. Es ist dank dieser Vorgehensweise nicht einmal notwendig, dass sich die

Systeme in der selben Umgebung befinden. Die zu testenden Komponenten können also auf jeder beliebigen Plattform laufen und lassen sich dennoch gemeinsam testen.

Auch den Aspekt der Transparenz kann MBT sehr gut abdecken. MBT ist generell als Black-Box Technik anzusehen. Daher ist für das Testen eines Systems nur dessen öffentliche Schnittstelle und die Kenntnis der gewünschten Funktionsweise notwendig, also genau jene Teile eines Services welcher der Außenwelt ohnedies zugänglich gemacht werden. Jegliche Systeminterna und dergleichen sind für die Testfallerstellung mittels MBT völlig irrelevant. Auch das Zusammenspiel von Services lässt sich mit MBT relativ einfach modellieren und somit auch die komponentenübergreifende/serviceübergreifende Funktionalität sicherstellen (Hinweis: in **??** lässt sich ein Beispiel für ein solches Modell finden).

Ein großes Problem für die manuelle Testfallerstellung stellt die Migration dar, da bei Änderungen in der Schnittstelle alle Testfälle neu erstellt werden bzw. abgeändert werden müssen. Hier bietet MBT besondere Vorteile, da die Testfälle generiert werden. Dieser Sachverhalt soll an einem Beispiel erläutert werden. Der in dieser Arbeit dargelegte IBusinessObjectDataService existierte nicht immer in dieser Form. Anfangs gab es für jede Art eines Business-Objekts einen eigenen Service (IFaciltiyService, ICustomerService, IUserService, ...). Durch MBT wurde allerdings gezeigt, dass es mitunter ratsam wäre, all diese Services zu einem einzigen Service zu vereinen, da die Modelle all dieser Services immer gleich waren, außer, dass sie auf unterschiedlichen Objekt-Arten operierten. Aufgrund dieser Zusammenlegung wurde nun ein einziger Service erstellt, welcher das gleiche Interface wie jeder der anderen Services bietet, aber anstatt dass ein konkreter Typ von BusinessObjekt übergeben/geliefert wird, arbeitet dieser Service mit dem Basis Typ BusinessObject. Das im Hintergrund arbeitende Systemverhalten war aber immer noch das exakt gleiche, lediglich die Systemschnittstelle änderte sich. Diese Schnittstellenänderung reichte aber, um alle manuellen Tests obsolet zu machen, sodass diese ohne Überarbeitung nicht mehr nutzbar waren. Das Modell für diese Services war aber noch immer gültig und aufgrund der vorgenommenen Abstraktionen - es wurde im Modell schon immer mit BusinessObject statt mit konkreten Ausprägungsformen dessen gearbeitet - immer noch valide. Es war lediglich eine Anpassung des Adapters notwendig. Nach einer erneuten Generierung der Testfälle, waren die Testfälle welche mittels MBT erstellt wurden wieder valide gegenüber der neuen Systemschnittstelle. Während die Anpassung der manuellen Testfälle mehr als zwei Tage beanspruchte (bei einigen tausend Zeilen Code und somit einigen dutzend Testfällen) konnten die mehr als 40.000 Zeilen Testcode welche mittels MBT erzeugt wurden innerhalb eines halben Tages wieder in einen lauffähigen Status gebracht werden.

7 Fazit

Modellbasiertes Testen ist auch heute noch ein sehr junges Gebiet in der Softwareentwicklung. Aber Tools wie SpecExplorer zeigen, dass diese Art des Testens im Begriff ist, äußerst praktikabel und rentabel zu werden. Trotz der erhöhten Komplexität zeigt sich, dass die sehr großen Vorteile, welche MBT mit sich bringt diese bei Weitem aufwiegen.

Ziel dieser Arbeit war es, eine Methode für die automatisierte Testfallerstellung zu finden, welche sich in einem realen Projektumfeld einsetzen lässt. SpecExplorer konnte diese Anforderung mit einem zufriedenstellenden Ergebnis erfüllen. Das Tool wird kontinuierlich verbessert und gehört mittlerweile wohl zu einem der umfangreichsten im Bereich des modellbasierten Testens[15].

Wie bei vielen Dingen in der Softwareentwicklung ist aber auch MBT kein Allheilmittel. Es muss für jedes Projekt genau evaluiert werden, ob sich MBT dafür eignet oder nicht. Die Service-orientierte Architektur des NTE.CLM Systems war ein idealer Testkandidat für MBT und es zeigte sich auch, dass solche Architekturen auf jeden Fall sehr gute Kandidaten für diese Testtechnik sind.

Jedoch darf man selbst dann nicht der Falschannahme unterliegen, dass MBT ein vollständiger Ersatz für die manuelle Testfallerstellung oder alternative automatisierende Techniken darstellt. MBT ist vielmehr ein weiteres Mittel, um das Portfolio an Techniken bezüglich der Qualitätssicherung von Softwareprojekten zu erweitern. Denn bestimmte Validierungen - vor allem in Hinsicht auf Funktionalität - lassen sich auch heutzutage noch immer mit Hilfe von manuell erstellten Tests in einem recht guten Verhältnis zwischen Aufwand und Nutzen erzeugen. Automatisierte Testtechniken sollten vielmehr als Ergänzung angesehen werden, welche die manuellen Tests vervollständigen und dadurch eine weitere Qualitätssteigerung mit sich bringen.

[15] Einschätzung des Autors

8 Ausblick

Die hier dargelegte Arbeit beschäftigte sich mit der grundlegenden Verwendung von MBT in Form des Tools SpecEplorer in Verbindung mit einer serviceorientierten Architektur. Damit wurde die Basis für eine ausreichende Qualitätssicherung für das NTE.CLM System geschaffen. Dennoch gibt es viele Aspekte an diesem Thema, welche weitere Arbeit benötigen.

Zum einen gehört dazu nicht synchrones Systemverhalten. Alle hier gezeigten Service waren synchroner Natur, sodass auf eine Anfrage direkt auch die Antwort kam. Gegen Ende dieser Arbeit wurden aber die ersten Services erstellt, die asynchroner Natur waren. Diesbezüglich ist es auf jeden Fall interessant inwiefern SpecExplorer diese Anforderungen ebenfalls erfüllen kann (Recherchen zeigen, dass SpecExplorer auch dafür Mechaniken umgesetzt hat, diese konnten aber im Rahmen dieser Arbeit nicht evaluiert werden).

Weiters wäre es von großem Interesse, zu evaluieren, inwiefern sich MBT auf den unterschiedlichen Abstraktionsebenen eines Systems einsetzen lässt. In den hier gezeigten Beispielen wurde immer eine sehr abstrakte Systemsicht angewandt. Die Frage in diesem Fall ist: „Lassen sich auch grundlegende Dinge wie die Parameter Validierung von Daten-Transfer-Objekten - welche im Falle von NTE.CLM mit manuell erstellten Testfällen validiert wurde - ebenfalls in vertretbarem Aufwand durch MBT abdecken“

Ein weiterer interessanter Punkt stellt die Frage dar, ob MBT auch dafür genutzt werden kann, nicht funktionale Aspekte eines Systems zu testen. Hier stellt sich die Frage, ob mit Hilfe von MBT und der damit automatisierten Testfallgenerierung Möglichkeiten geschaffen werden könnten, um relativ reale Szenarien für Last-Test und Load-Tests zu schaffen.

9 Appendix

9.1 Modellprogramme

9.1.1 BusinessObjectService

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;
using Microsoft.Modeling;
using Nte.Clm.ModelStates.Common;

namespace BusinessObjectDataService.Model
{

    static class IdGenerator
    {
        static int idCounter = 0;

        public static int AquireNextId()
        {
            idCounter++;
            return idCounter;
        }
    }

    public static class BusinessObjectDataServiceModelProgram
    {

        #region State

        static SequenceContainer<BusinessObjectState> existingObjects
            = new SequenceContainer<BusinessObjectState>();
        static MapContainer<BusinessObjectState, SequenceContainer<BusinessObjectState>>
          assignmentSources =
          new MapContainer<BusinessObjectState,SequenceContainer<BusinessObjectState>>();

        #endregion

        [Rule]
        public static BusinessObjectState Create(BusinessObjectState bo)
        {
            Condition.IsTrue(bo.BoType != BusinessObjectTypeState.None);
            Condition.IsTrue(existingObjects.Contains(bo) == false);
            Condition.IsTrue(bo.BoType == BusinessObjectTypeState.Customer);

            BusinessObjectState state = new BusinessObjectState()
            {
                Id          = IdGenerator.AquireNextId(),
                BoType      = bo.BoType
            };

            existingObjects.Add(state);
            return state;
        }

        [Rule]
        public static BusinessObjectState
          CreateAndAssignToTarget(BusinessObjectState newBos,
                                  BusinessObjectState existingBos)
```

```
56          {
57              Condition.IsTrue(existingObjects.Contains(existingBos));
58              Condition.IsTrue(existingObjects.Contains(newBos) == false);
59
60              Condition.IsTrue(
61                  (newBos.BoType == BusinessObjectTypeState.User &&
62                    existingBos.BoType == BusinessObjectTypeState.Customer) ||
63                  (newBos.BoType == BusinessObjectTypeState.Facility &&
64                    existingBos.BoType == BusinessObjectTypeState.Customer));
65
66              BusinessObjectState newBoState = new BusinessObjectState()
67              {
68                  Id                          = IdGenerator.AquireNextId(),
69                  BoType                      = newBos.BoType,
70              };
71
72              if(assignmentSources.ContainsKey(existingBos) == false)
73                  assignmentSources.Add(existingBos,
74                    new SequenceContainer<BusinessObjectState>());
75              assignmentSources[existingBos].Add(newBoState);
76              existingObjects.Add(newBoState);
77
78              return newBoState;
79          }
80
81          [Rule]
82          public static Set<BusinessObjectState> GetByType(BusinessObjectTypeState boType)
83          {
84              Condition.IsTrue(boType == BusinessObjectTypeState.Customer ||
85                boType == BusinessObjectTypeState.User);
86              return new Set<BusinessObjectState>(existingObjects.Where(
87                (e) => e.BoType == boType).ToArray());
88          }
89
90          [Rule]
91          public static Set<BusinessObjectState>
92            GetAssignmentSourcesWithType(BusinessObjectState target,
93                                         BusinessObjectTypeState bots)
94          {
95              Condition.IsTrue(existingObjects.Contains(target));
96
97              if(assignmentSources.ContainsKey(target))
98                  return new Set<BusinessObjectState>(assignmentSources[target].Where(
99                    (i) => i.BoType == bots).ToArray());
100             return new Set<BusinessObjectState>();
101         }
102
103         [Rule]
104         public static Dto.BusinessObject Create(Dto.BusinessObject bo)
105         {
106             bo.Id = AquireNextId();
107             return bo;
108         }
109
110         [Rule]
111         public static List<BusinessObjectState> GetByType(BusinessObjectType boType)
112         {
113             Condition.IsTrue(boType != BusinessObjectType.None);
114
115             var withType = existingObjects.Where((eo) => eo.BoType == boType);
116             return new List<BusinessObjectState>(withType);
117         }
118
119     }
120 }
```

9.1.2 SecurityService

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Linq;

using Microsoft.Modeling;
using Nte.Clm.Security.Adapters.States;
using System.Diagnostics;

namespace Nte.Clm.Security.Models
{

    public static class IdGenerator
    {
        static int idCounter = 0;

        public static int AquireNextId()
        {
            idCounter++;
            return idCounter;
        }
    }

    static class SecuirtyWithBosModelProgram
    {
        public static int MaxObjectPerCustomer = 3;

        private static MapContainer<BusinessObjectState, SetContainer<BusinessObjectState>>
          customerObjects
          = new MapContainer<BusinessObjectState,SetContainer<BusinessObjectState>>();
        private static MapContainer<BusinessObjectState, SetContainer<BusinessObjectState>>
          userInstanceReadableObjects
          = new MapContainer<BusinessObjectState, SetContainer<BusinessObjectState>>();
        private static MapContainer<BusinessObjectState, SetContainer<BusinessObjectState>>
          userInstanceFullAccessObjects
          = new MapContainer<BusinessObjectState,SetContainer<BusinessObjectState>>();
        private static MapContainer<BusinessObjectState, SetContainer<BusinessObjectState>>
          userInstanceInacessibleObjects
          = new MapContainer<BusinessObjectState,SetContainer<BusinessObjectState>>();
        private static MapContainer<BusinessObjectState,
          MapContainer<BusinessObjectTypeState, AccessState>>
          userTypeAccessRights
          = new MapContainer<BusinessObjectState,
            MapContainer<BusinessObjectTypeState,AccessState>>();

        [Rule]
        public static BusinessObjectState
          CreateCustomerWithRoot(BusinessObjectState newCustomer)
        {
            Condition.IsTrue(newCustomer.Id == 0);
            Condition.IsTrue(newCustomer.BoType == BusinessObjectTypeState.Customer);
            Condition.IsFalse(customerObjects.ContainsKey(newCustomer));
            BusinessObjectState created = new BusinessObjectState()
            {
                Id              = IdGenerator.AquireNextId(),
                BoType          = BusinessObjectTypeState.Customer,
            };
            customerObjects.Add(created, new SetContainer<BusinessObjectState>());
```

```
59              return created;
60          }
61
62          [Rule]
63          public static BusinessObjectState CreateBusinessObject(BusinessObjectState bo,
64                                                                 BusinessObjectState target)
65          {
66              Condition.IsTrue(customerObjects.ContainsKey(target));
67              Condition.IsFalse(ObjectExists(bo));
68              Condition.IsTrue(bo.Id == 0);
69              Condition.IsTrue(
70                  (bo.BoType == BusinessObjectTypeState.Facility &&
71                    target.BoType == BusinessObjectTypeState.Customer) ||
72                  (bo.BoType == BusinessObjectTypeState.User &&
73                    target.BoType == BusinessObjectTypeState.Customer));
74
75              //"Harcoded" constraints to make scenario generation easier
76              Condition.IsTrue(customerObjects[target].Count < MaxObjectPerCustomer);
77
78              BusinessObjectState created               = new BusinessObjectState()
79              {
80                  Id                          = IdGenerator.AquireNextId(),
81                  BoType                      = bo.BoType,
82              };
83
84              BusinessObjectState targetsCustomer = GetTargetCustomer(target);
85
86              customerObjects[targetsCustomer].Add(created);
87
88              return created;
89          }
90
91          [Rule]
92          public static Set<BusinessObjectState> GetByType(BusinessObjectTypeState bots,
93                                                           BusinessObjectState user)
94          {
95
96              Condition.IsTrue(ObjectExists(user));
97              Condition.IsTrue(user.BoType == BusinessObjectTypeState.User);
98              Condition.IsTrue(bots != BusinessObjectTypeState.None);
99              Condition.IsTrue(userInstanceReadableObjects.ContainsKey(user));
100
101             BusinessObjectState domainCustomer = GetTargetCustomer(user);
102
103             Condition.IsTrue(domainCustomer.BoType == BusinessObjectTypeState.Customer);
104             Condition.IsTrue(customerObjects.ContainsKey(domainCustomer));
105
106             List<BusinessObjectState> accessibleObjects = new List<BusinessObjectState>();
107
108             foreach (BusinessObjectTypeState ts in
109               Enum.GetValues(typeof(BusinessObjectTypeState)))
110             {
111                 AccessState typeAccess;
112                 if (GetUserTypeAccess(user, bots, out typeAccess) == true)
113                 {
114                     switch (typeAccess)
115                     {
116                         case AccessState.Read:
117                         case AccessState.Full:
118                             if (ts != BusinessObjectTypeState.None && ts !=
119                               BusinessObjectTypeState.Customer)
120                                 accessibleObjects.AddRange(
121                                   customerObjects[domainCustomer].Where(
122                                   (o) => o.BoType == ts).AsEnumerable());
123                             break;
```

```
124                     }
125                 }
126             }
127
128             List<BusinessObjectState> instanceAccessible =
129             new List<BusinessObjectState>(userInstanceReadableObjects[user].Where(
130               (o) => o.BoType == bots).ToArray());
131             foreach (var state in instanceAccessible)
132             {
133                 if (accessibleObjects.Contains(state) == false)
134                   accessibleObjects.Add(state);
135             }
136
137             if (userInstanceInacessibleObjects.ContainsKey(user))
138             {
139                 foreach (var state in userInstanceInacessibleObjects[user])
140                 {
141                     accessibleObjects.Remove(state);
142                 }
143             }
144
145
146             return new Set<BusinessObjectState>(accessibleObjects.ToArray());
147         }
148
149         private static bool GetUserTypeAccess(BusinessObjectState user,
150                                               BusinessObjectTypeState bots,
151                                               out AccessState access)
152         {
153             if (userTypeAccessRights.ContainsKey(user))
154             {
155                 if (userTypeAccessRights[user].ContainsKey(bots))
156                 {
157                     access = userTypeAccessRights[user][bots];
158                     return true;
159                 }
160             }
161
162             access = AccessState.None;
163             return false;
164         }
165
166         [Rule]
167         public static void SetUsersRole(BusinessObjectState user,
168                                         BusinessObjectState customer,
169                                         RoleState role)
170         {
171             Condition.IsTrue(user.BoType == BusinessObjectTypeState.User);
172             Condition.IsTrue(customer.BoType == BusinessObjectTypeState.Customer);
173             Condition.IsTrue(ObjectExists(user));
174             Condition.IsTrue(ObjectExists(customer));
175             Condition.IsTrue(customerObjects.ContainsKey(customer));
176             Condition.IsTrue(customerObjects[customer].Contains(user));
177             Condition.IsTrue(role != RoleState.None);
178
179             if(userInstanceReadableObjects.ContainsKey(user))
180                 userInstanceReadableObjects[user].Clear();
181             else
182                 userInstanceReadableObjects.Add(user,
183                   new SetContainer<BusinessObjectState>());
184
185             if(userInstanceFullAccessObjects.ContainsKey(user))
186                 userInstanceFullAccessObjects[user].Clear();
187             else
188                 userInstanceFullAccessObjects.Add(user,
```

```
189                     new SetContainer<BusinessObjectState>());
190
191             if (role == RoleState.Administrator)
192             {
193                 userInstanceFullAccessObjects[user].Add(customer);
194                 userInstanceFullAccessObjects[user].Add(user);
195                 userInstanceFullAccessObjects[user].AddRange(customerObjects[customer]);
196                 userInstanceReadableObjects[user].Add(customer);
197                 userInstanceReadableObjects[user].AddRange(customerObjects[customer]);
198                 userInstanceReadableObjects[user].Add(user);
199             }
200             else if (role == RoleState.User)
201             {
202                 userInstanceReadableObjects[user].Add(customer);
203                 userInstanceReadableObjects[user].AddRange(customerObjects[customer]);
204                 userInstanceReadableObjects[user].Add(user);
205             }
206         }
207
208         [Rule]
209         public static void SetInstanceAccessRight(BusinessObjectState bo,
210                                                   BusinessObjectState user,
211                                                   AccessState access)
212         {
213             Condition.IsTrue(bo.BoType != BusinessObjectTypeState.None);
214             Condition.IsTrue(user.BoType == BusinessObjectTypeState.User);
215
216             BusinessObjectState customer = GetTargetCustomer(bo);
217             Condition.IsTrue(customerObjects.ContainsKey(customer));
218             Condition.IsTrue(customerObjects[customer].Contains(user));
219
220             if(userInstanceFullAccessObjects.ContainsKey(user) == false)
221               userInstanceFullAccessObjects.Add(user,
222                 new SetContainer<BusinessObjectState>());
223             if(userInstanceReadableObjects.ContainsKey(user) == false)
224               userInstanceReadableObjects.Add(user,
225                 new SetContainer<BusinessObjectState>());
226             if(userInstanceInacessibleObjects.ContainsKey(user) == false)
227               userInstanceInacessibleObjects.Add(user,
228                 new SetContainer<BusinessObjectState>());
229
230             if (access == AccessState.None)
231             {
232                 userInstanceFullAccessObjects[user].Remove(bo);
233                 userInstanceReadableObjects[user].Remove(bo);
234                 userInstanceInacessibleObjects[user].Add(bo);
235             }
236             else if (access == AccessState.Read)
237             {
238                 userInstanceInacessibleObjects[user].Remove(bo);
239                 userInstanceFullAccessObjects[user].Remove(bo);
240                 if (userInstanceReadableObjects[user].Contains(bo) == false)
241                   userInstanceReadableObjects[user].Add(bo);
242             }
243             else
244             {
245                 userInstanceInacessibleObjects.Remove(bo);
246                 if(userInstanceFullAccessObjects[user].Contains(bo) == false)
247                   userInstanceFullAccessObjects[user].Add(bo);
248                 if(userInstanceReadableObjects[user].Contains(bo) == false)
249                   userInstanceReadableObjects[user].Add(bo);
250             }
251
252         }
253
```

```
254         [Rule]
255         public static void SetTypeAccessRights(BusinessObjectTypeState boType,
256                                                BusinessObjectState user,
257                                                AccessState access,
258                                                BusinessObjectState securityDomain)
259         {
260             if(userTypeAccessRights.ContainsKey(user) == false)
261               userTypeAccessRights.Add(user,
262                 new MapContainer<BusinessObjectTypeState, AccessState>());
263 
264             Condition.IsTrue(user.BoType == BusinessObjectTypeState.User);
265             Condition.IsTrue(boType != BusinessObjectTypeState.None);
266             Condition.IsTrue(customerObjects.ContainsKey(securityDomain));
267             Condition.IsTrue(customerObjects[securityDomain].Contains(user));
268 
269             if(userTypeAccessRights[user].ContainsKey(boType) == true)
270                 userTypeAccessRights[user][boType] = access;
271             else
272                 userTypeAccessRights[user].Add(boType, access);
273         }
274 
275 
276         #region Helpers
277 
278         private static bool ObjectExists(BusinessObjectState bos)
279         {
280             foreach (var kvp in customerObjects)
281             {
282                 if(kvp.Key.Equals(bos) || kvp.Value.Contains(bos)) return true;
283             }
284 
285             return false;
286         }
287 
288         private static BusinessObjectState GetTargetCustomer(BusinessObjectState bos)
289         {
290             if(bos.BoType == BusinessObjectTypeState.Customer) return bos;
291 
292             foreach (var kvp in customerObjects)
293             {
294                 if(kvp.Value.Contains(bos) == true) return kvp.Key;
295             }
296 
297             return new BusinessObjectState()
298             {
299               Id = -1, BoType = BusinessObjectTypeState.None
300             };
301         }
302 
303         #endregion
304     }
305 }
```

9.2 Adapter

9.2.1 BusinessObjectDataServiceAdapter

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
```

```
4  using System.Linq;
5  using Microsoft.Modeling;
6  using Nte.Clm.Shared.Hosting.ServiceProxies;
7  using Nte.Clm.Shared.ServiceContracts.Domain;
8  using System.ServiceModel;
9  using Nte.Clm.ModelStates.Common;
10
11
12 namespace BusinessObjectDataService.Model.Sample
13 {
14     using Dto = Nte.Clm.Shared.Dto;
15
16     public static class BusinessObjectDataServiceAdapter
17     {
18         static int idCounter = 0;
19         private static List<Tuple<long, Dto.BusinessObject>> modelIdRealDtoMap
20           = new List<Tuple<long,Dto.BusinessObject>>();
21
22         public static BusinessObjectState Create(BusinessObjectState bos)
23         {
24             var newBo = CreateDto(bos);
25             var proxy =
26               ServiceProxyFactory.Instance.CreateProxy<IBusinessObjectDataService>();
27             var created = proxy.Create(newBo);
28
29             BusinessObjectState boState = new BusinessObjectState()
30             {
31                 Id = AquireNextId(),
32                 BoType = DtoTypeToBusinessObjectTypeState(created.GetType()),
33             };
34
35             modelIdRealDtoMap.Add(Tuple.Create(boState.Id, created));
36
37             return boState;
38         }
39
40         public static BusinessObjectState
41           CreateAndAssignToTarget(BusinessObjectState bos,
42                                   BusinessObjectState existing)
43         {
44             var newBo = CreateDto(bos);
45             var existingBo = modelIdRealDtoMap.Single((i) => i.Item1 == existing.Id).Item2;
46
47             var proxy                          =
48               ServiceProxyFactory.Instance.CreateProxy<IBusinessObjectDataService>();
49             using(var channel = proxy as IClientChannel)
50             {
51                 var created = proxy.CreateAndAssignToTarget(newBo, existingBo);
52                 BusinessObjectState state    = new BusinessObjectState()
53                 {
54                     Id = AquireNextId(),
55                     BoType = DtoTypeToBusinessObjectTypeState(created.GetType()),
56                 };
57
58                 modelIdRealDtoMap.Add(Tuple.Create(state.Id, created));
59                 return state;
60             }
61         }
62
63
64         public static Set<BusinessObjectState>
65           GetByType(BusinessObjectTypeState boType)
66         {
67             return null;
68         }
```

```
69
70          private static BusinessObjectTypeState
71            DtoTypeToBusinessObjectTypeState(Type t)
72          {
73              if(t == typeof(Dto.Customer))
74                return BusinessObjectTypeState.Customer;
75              return BusinessObjectTypeState.None;
76          }
77
78          private static Dto.BusinessObject CreateDto(BusinessObjectState bos)
79          {
80              if(bos.BoType == BusinessObjectTypeState.Customer)
81              {
82                  return CreateCustomerDto();
83              }
84              else if(bos.BoType == BusinessObjectTypeState.User)
85              {
86                  return CreateUserDto();
87              }
88              else
89                  return null;
90          }
91
92          private static Dto.BusinessObject CreateCustomerDto()
93          {
94              Dto.Customer customer = new Dto.Customer()
95              {
96                  Id              = 0,
97                  Name            = Guid.NewGuid().ToString(),
98                  Disabled        = false,
99                  Description     = Guid.NewGuid().ToString(),
100             };
101
102             return customer;
103         }
104
105         private static Dto.BusinessObject CreateUserDto()
106         {
107             Dto.User user           = new Dto.User()
108             {
109                 Id                  = 0,
110                 Name                = Guid.NewGuid().ToString(),
111                 Disabled            = false,
112                 Description         = Guid.NewGuid().ToString(),
113                 Password            = Guid.NewGuid().ToString(),
114             };
115
116             return user;
117         }
118
119         private static long AquireNextId()
120         {
121             idCounter++;
122             return idCounter;
123         }
124     }
125 }
```

9.2.2 SecurityWithBusinessObjectsAdapter

```
1 using System;
```

```
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using Nte.Clm.Shared.ServiceContracts.Domain;
6  using Nte.Clm.Shared.ServiceContracts.Administration;
7  using Nte.Clm.Shared.Dto;
8  using Nte.Clm.Shared.Hosting.ServiceProxies;
9  using Microsoft.Modeling;
10 using System.Net;
11 using Nte.Lms.Services.Contracts.Services;
12 using Nte.Clm.Security.Adapters.States;
13 using Nte.Clm.TestUtils;
14
15
16 namespace Nte.Clm.Security.Adapters
17 {
18     public static class SecurityWithBosAdapter
19     {
20         const string originalPassword = "password";
21
22         #region Fields
23         static CommonTestBase testBase = new CommonTestBase();
24
25         static IBusinessObjectDataService businessObjectService = null;
26         static ISecurityService securityService = null;
27
28         private static BusinessObjectState sysAdmin = new BusinessObjectState()
29         {
30             Id = 1,
31             BoType = BusinessObjectTypeState.User,
32         };
33
34         private static List<Tuple<BusinessObjectState, BusinessObject>>
35           stateDtoMapping = new List<Tuple<BusinessObjectState,BusinessObject>>();
36
37         private static int idCounter = 0;
38
39         private static User RootUser = new User()
40         {
41             Name = "SysAdmin",
42         };
43
44         #endregion
45
46         static SecurityWithBosAdapter()
47         {
48             SwitchServiceUser(RootUser);
49             ServicePointManager.ServerCertificateValidationCallback =
50               (sender, cert, chain, pe) => true;
51         }
52
53
54         public static BusinessObjectState
55           CreateCustomerWithRoot(BusinessObjectState customer)
56         {
57             ICustomerService customerProxy
58               = ServiceProxyFactory.Instance.CreateProxy<ICustomerService>();
59             IProductService  productsProxy
60               = ServiceProxyFactory.Instance.CreateProxy<IProductService>();
61             ILicensingService licensingProxy
62               = ServiceProxyFactory.Instance.CreateProxy<ILicensingService>();
63             ISecurityService securityProxy
64               = ServiceProxyFactory.Instance.CreateProxy<ISecurityService>();
65             IBusinessObjectDataService boService
66               = ServiceProxyFactory.Instance.CreateProxy<IBusinessObjectDataService>();
```

```
67
68              //Create LMS customer and check CLM Customer gets created
69
70              Lms.Dto.Customer lmsCustomer                = customerProxy.Create(
71                  new Lms.Dto.Customer()
72                  {
73                    Name = testBase.CreateUniqueString()
74                  }, internalCustomer: false);
75
76              var customers = boService.GetByType(BusinessObjectType.Customer);
77              Customer clmCustomer = boService.GetByType(BusinessObjectType.Customer).
78                Cast<Customer>().First(
79                  (c) => c.LmsGuid != null && c.LmsGuid == lmsCustomer.Id);
80
81              //Install a new Product
82              var product = productsProxy.CreateProduct(new Lms.Dto.Product()
83              {
84                Name = "CLM Test " + DateTime.Now.ToString()
85              });
86              var featureSet = productsProxy.CreateFeatureSet(new Lms.Dto.FeatureSet()
87              {
88                Name = "Test Feature Set [BASE]"
89              }, product);
90              var feature = productsProxy.CreateFeature(
91                  new Lms.Dto.Feature()
92                  {
93                      Name = "Teste Feature [BOD Access]",
94                      ClaimRight = Shared.Claims.ClaimRights.PosessProperty,
95                      ClaimType = Shared.Claims.ClaimTypes.GrantServiceAccess,
96                      ClaimResource = Shared.Claims.ClaimResources.CompleteServer
97                  }, featureSet);
98
99              //install a license for the customer
100
101             Dictionary<Lms.Dto.FeatureOwnership,
102               List<Lms.Dto.FeatureSetParameterValue>> featureOwnerhips =
103                 new Dictionary<Lms.Dto.FeatureOwnership,
104                   List<Lms.Dto.FeatureSetParameterValue>>();
105             featureOwnerhips.Add(
106                 new Lms.Dto.FeatureOwnership()
107                 {
108                     FeatureSetId                        = featureSet.Id,
109                     ProductLicenseCustomerId            = lmsCustomer.Id,
110                     ProductLicenseProductId             = product.Id,
111                     OwnerhipStart                       = DateTime.Now,
112                     OwnershipEnd                        = DateTime.Now.AddDays(2),
113                     LockAutomatically                   = false,
114                     IsLocked                            = false,
115                 }, new List<Lms.Dto.FeatureSetParameterValue>());
116
117             licensingProxy.AddLicense(product, featureOwnerhips, lmsCustomer);
118
119             BusinessObjectState createdState                = new BusinessObjectState()
120             {
121                 Id                                          = AquireNextId(),
122                 BoType                                      = DtoTypeToState(clmCustomer),
123             };
124
125             stateDtoMapping.Add(Tuple.Create(createdState, clmCustomer as BusinessObject));
126
127             return createdState;
128         }
129
130
131         public static BusinessObjectState
```

```
132           CreateBusinessObject(BusinessObjectState newBoState,
133                                BusinessObjectState targetState)
134       {
135           SwitchServiceUser(RootUser);
136
137           BusinessObject newbo = CreateBusinessObjectDto(newBoState);
138           BusinessObject targetBo = StateToBusinessObject(targetState);
139           var created = businessObjectService.CreateAndAssignToTarget(newbo, targetBo);
140
141           BusinessObjectState createdState         = new BusinessObjectState()
142           {
143               Id              = AquireNextId(),
144               BoType          = DtoTypeToState(created),
145           };
146
147
148           stateDtoMapping.Add(Tuple.Create(createdState, created));
149
150           return createdState;
151       }
152
153       public static void SetUsersRole(BusinessObjectState userState,
154                                        BusinessObjectState customerState,
155                                        RoleState roleState)
156       {
157           SwitchServiceUser(RootUser);
158
159           User user = (User)StateToBusinessObject(userState);
160           Customer customer = (Customer)StateToBusinessObject(customerState);
161           List<Role> roles = securityService.GetRoles(customer);
162           Role correctRole = RoleStateToRole(roleState, roles);
163           if(correctRole == null)
164             throw new Exception("RoleState cannot be mapped to a RoleDTO");
165           securityService.SetUsersRole(correctRole, user);
166           securityService.ClearAllUserPermissions(user, customer);
167       }
168
169       public static Set<BusinessObjectState> GetByType(BusinessObjectTypeState bots,
170                                                        BusinessObjectState serviceUser)
171       {
172           SwitchServiceUser(serviceUser);
173
174           BusinessObjectType boType = TypeStateToType(bots);
175           var fetched = businessObjectService.GetByType(boType);
176           List<BusinessObjectState> boStates = new List<BusinessObjectState>();
177           foreach (var f in fetched)
178           {
179               boStates.Add(new BusinessObjectState()
180               {
181                   Id                  = BusinessObjectToState(f).Id,
182                   BoType              = DtoTypeToState(f)
183               });
184           }
185
186           return new Set<BusinessObjectState>(boStates.ToArray());
187       }
188
189       public static void SetInstanceAccessRight(BusinessObjectState bo,
190                                                 BusinessObjectState user,
191                                                 AccessState access)
192       {
193           SwitchServiceUser(RootUser);
194
195           BusinessObject businessObject               = StateToBusinessObject(bo);
196           BusinessObject userObject                   = StateToBusinessObject(user);
```

```
197             PermissionType pt                               = AccessStateToPermission(access);
198
199             securityService.SetUserPermissions(userObject as User, businessObject, pt);
200         }
201
202         public static void SetTypeAccessRights(BusinessObjectTypeState bots,
203                                                BusinessObjectState user,
204                                                AccessState access,
205                                                BusinessObjectState securityDomainState)
206         {
207             SwitchServiceUser(RootUser);
208
209             BusinessObjectType boType = TypeStateToType(bots);
210             User u = (User)StateToBusinessObject(user);
211             PermissionType permission = AccessStateToPermission(access);
212             Customer securityDomain = (Customer)StateToBusinessObject(securityDomainState);
213
214             securityService.SetUserTypePermissions(u, boType, permission, securityDomain);
215         }
216
217         public static void ResetAdapter()
218         {
219             ResetIdCounter();
220             ResetMappings();
221         }
222
223         private static void ResetMappings()
224         {
225             stateDtoMapping.Clear();
226
227         }
228
229         private static void ResetIdCounter()
230         {
231             idCounter = 0;
232         }
233
234         #region Helpers
235
236         private static PermissionType AccessStateToPermission(AccessState state)
237         {
238             switch(state)
239             {
240                 case AccessState.Full:
241                   return PermissionType.Read |
242                     PermissionType.Update |
243                     PermissionType.Delete;
244                 case AccessState.Read:
245                   return PermissionType.Read;
246                 default: return PermissionType.None;
247             }
248         }
249
250         private static BusinessObject CreateBusinessObjectDto(BusinessObjectState bos)
251         {
252             if(bos.BoType == BusinessObjectTypeState.Customer)
253                 return testBase.CreateCustomerDto();
254             else if(bos.BoType == BusinessObjectTypeState.User)
255                 return testBase.CreateUserDto();
256             else if(bos.BoType == BusinessObjectTypeState.Facility)
257                 return testBase.CreateFacilityDto();
258             else
259                 return null;
260         }
261
```

```
262         private static Role RoleStateToRole(RoleState rs, List<Role> roles)
263         {
264             if(rs == RoleState.Administrator)
265               return roles.Single((r) => r.Name == "Administrators");
266             else if(rs == RoleState.User)
267               return roles.Single((r) => r.Name == "Users");
268             else return null;
269         }
270 
271 
272 
273 
274 
275         private static BusinessObjectType TypeStateToType(BusinessObjectTypeState bots)
276         {
277             if(bots == BusinessObjectTypeState.Customer)
278               return BusinessObjectType.Customer;
279             else if(bots == BusinessObjectTypeState.Facility)
280               return BusinessObjectType.Facility;
281             else if(bots == BusinessObjectTypeState.User)
282               return BusinessObjectType.User;
283             else return BusinessObjectType.BusinessObject;
284 
285         }
286 
287         private static BusinessObjectTypeState DtoTypeToState(BusinessObject obj)
288         {
289             if(obj.GetType() == typeof(Customer))
290               return BusinessObjectTypeState.Customer;
291             else if(obj.GetType() == typeof(Facility))
292               return BusinessObjectTypeState.Facility;
293             else if(obj.GetType() == typeof(User))
294               return BusinessObjectTypeState.User;
295             else
296               return BusinessObjectTypeState.None;
297         }
298 
299         private static BusinessObject StateToBusinessObject(BusinessObjectState bos)
300         {
301             return stateDtoMapping.Single((e) => e.Item1.Id == bos.Id).Item2;
302         }
303 
304         private static BusinessObjectState BusinessObjectToState(BusinessObject bos)
305         {
306             return stateDtoMapping.Single((e) => e.Item2.Id == bos.Id).Item1;
307         }
308 
309         private static void SwitchServiceUser(BusinessObjectState userState)
310         {
311             User realUser = (User)StateToBusinessObject(userState);
312             SwitchServiceUser(realUser);
313         }
314 
315         private static void SwitchServiceUser(User u)
316         {
317             businessObjectService =
318               ServiceProxyFactory.Instance.
319                 CreateProxy<IBusinessObjectDataService>(u.Name, originalPassword);
320             securityService =
321               ServiceProxyFactory.Instance.
322                 CreateProxy<ISecurityService>(u.Name, originalPassword);
323         }
324 
325         private static int AquireNextId()
326         {
```

```
327             idCounter++;
328             return idCounter;
329         }
330 
331         #endregion
332     }
333 }
```

9.3 Coord-Skripte

9.3.1 BusinessObjectDataService

```
1  // This is a Spec Explorer coordination script (Cord version 1.0).
2  // Here is where you define configurations and machines describing the
3  // exploration to be performed.
4 
5  using BusinessObjectDataService.Model;
6  using BusinessObjectDataService.Model.Sample;
7 
8  config Main
9  {
10     action all BusinessObjectDataServiceAdapter;
11 
12     switch StepBound = 128;
13     switch PathDepthBound = 128;
14     switch TestClassBase = "MbtTestBase";
15     switch GeneratedTestPath = "..\\BusinessObjectDataService.Model.TestSuite";
16     switch GeneratedTestNamespace = "BusinessObjectDataService.Model.TestSuite";
17     switch TestEnabled = false;
18     switch ForExploration = false;
19 }
20 
21 machine BusinessObjectDataServiceModelProgram() : Main where ForExploration = true
22 {
23     construct model program from Main
24     where namespace =
25       "BusinessObjectDataService.Model.BusinessObjectDataServiceModelProgram"
26 }
27 
28 
29 machine CreateTwoCustomersSlice() :
30   Main where ForExploration = true
31 {
32     Create; Create; GetByType;
33 }
34 
35 machine CreateTwoCustomersScenario() :
36   Main where ForExploration = true
37 {
38     BusinessObjectDataServiceModelProgram || CreateTwoCustomersSlice;
39 }
40 
41 machine TSCreateTwoCustomersSceneario() :
42   Main where ForExploration = true, TestEnabled = true
43 {
44     construct test cases for CreateTwoCustomersScenario()
45 }
```

9.3.2 SecurityService

```
using Nte.Clm.Security.Adapters;
using Nte.Clm.Security.Adapters.States;

/// Contains actions of the model, bounds, and switches.
config Main
{
    /// The two implementation actions that will be modeled and tested
    action all SecurityWithBosAdapter;
    //abstract action SecuirtyWithBosAdapter

    switch StepBound = 8192;
    switch StateBound = 8192;
    switch PathDepthBound = 4096;
    switch TestClassBase = "MbtTestBase";
    switch GeneratedTestPath = "..\\..\\Testing\\Nte.Clm.Security.TestSuite";
    switch GeneratedTestNamespace = "Nte.Clm.Security.TestSuite";

    switch TestEnabled = false;
    switch ForExploration = false;
}

machine ModelProgramSecurityWithBos() : Main where ForExploration = true
{
    construct model program from Main
} //this machine only good example for state explosion



//Scenario Control machines

machine ScenarioTwoCustomers() : Main where ForExploration = true
{
    CreateCustomerWithRoot;
    CreateCustomerWithRoot;

}

machine ScenarioUserAndFacility() : Main where ForExploration = true
{
    CreateBusinessObject(BusinessObjectState(Id=0,
      BoType=BusinessObjectTypeState.User), _);
    CreateBusinessObject(BusinessObjectState(Id=0,
      BoType=BusinessObjectTypeState.Facility), _);
}


machine ScenarioChangingRoles() : Main where ForExploration = true
{
    (SetUsersRole | GetByType)*
}

machine ScenarioTwoCustomersChangingRoles() : Main where ForExploration = true
{
    ScenarioTwoCustomers; ScenarioUserAndFacility; ScenarioChangingRoles
}

machine ScenarioSetInstanceRights() : Main where ForExploration = true
{
    //(SetInstanceAccessRight | GetByType)*
    SetInstanceAccessRight;
    GetByType;
    SetInstanceAccessRight;
```

```
63      GetByType;
64  }
65
66  machine ScenarioSetInstanceRightsComplete() : Main where ForExploration = true
67  {
68      (
69          ScenarioTwoCustomers;
70          CreateBusinessObject(BusinessObjectState(Id=0,
71            BoType=BusinessObjectTypeState.User), _);
72          SetUsersRole;
73          ScenarioSetInstanceRights;
74      )
75  }
76
77  //Sliced machines
78
79  machine SliceSecurityWithBosTwoCustomersChangingRoles() :
80    Main where ForExploration = true
81  {
82      ModelProgramSecurityWithBos || ScenarioTwoCustomersChangingRoles
83  }
84
85
86  machine SliceSecuirtyWithBosTwoCustomersSetInstanceRights() :
87    Main where ForExploration = true
88  {
89      ModelProgramSecurityWithBos || ScenarioSetInstanceRightsComplete
90  }
91
92  /////Test machines
93
94  machine TestSuiteSecurityWithBosTwoCustomersChangeRoles() :
95    Main where ForExploration = true, TestEnabled = true
96  {
97      construct test cases where Strategy = "ShortTests" for
98        SliceSecurityWithBosTwoCustomersChangingRoles()
99  }
100
101 machine TestSuiteSecuirtyWithBosSetInstanceRights() :
102   Main where ForExploration = true, TestEnabled = true
103 {
104     construct test cases where Strategy = "ShortTests"
105       for SliceSecuirtyWithBosTwoCustomersSetInstanceRights()
106 }
```

Literatur

[1] *MDA Specifications, UML-Profiles.* `http://www.omg.org/mda/specs.htm`. Version: 2006

[2] *Modellbildung.* `http://de.wikipedia.org/wiki/Modellbildung`. Version: 2010

[3] *SwitchClause.* `http://msdn.microsoft.com/en-us/library/ee620405.aspx`. Version: 2010

[4] *TestCasesConstruct - Spec Explorer 3.5.* `http://msdn.microsoft.com/en-us/library/ee620427.aspx`. Version: 2010

[5] *TypeBinding Attribute.* `http://msdn.microsoft.com/en-us/library/ee620507.aspx`. Version: 2010

[6] Bakken, D. E.: Middleware. In: *Encyclopedia of Distributed Computing* (2003). – Kluwer Academic Press

[7] Boehm, M. ; Jungkunz, B.: *Grundkurs IT-berufe: Die technischen Grundlagen verstehen und anwenden koennen.* Vieweg & Teubner, 2005

[8] Brost, M. ; Reuss, H. C. ; Zoellner, R.: Automatische Testfallgenerierung aus einer formalen Funktionsbeschreibung. In: *1. AutoTest Konferenz.* Stuttgart : FKFS, Oktober 2006

[9] Buchacker, K. ; Sieh, V. ; Alexander, F.: Framework for Testing the Fault-Tolerance of Systems Including OS and Network Aspects. In: *IEEE High-Assurance System Engineering Symposium, Boca*, 2001, S. 95–105

[10] Corporation, Microsoft: *Parameterized Unit Testing with Microsoft Pex.* `http://research.microsoft.com/en-us/projects/pex/pextutorial.pdf`. Version: 2010

[11] Coulouris, G. ; Dollimore, J. ; Kindberg, T.: *DISTRIBUTED SYSTEMS Concepts and Design.* Addison Wesley, 2001

[12] Craggs, I. ; Sardis, M. ; Heuillard, T.: AGEDIS Case Studies: Model-Based Testing in Industry, imbus AG, Dezember 2003 (1.st European Conference on Model Driven Software Engineering), S. 106–117

[13] Cristian, F.: Understanding fault-tolerant distributed systems. In: *Commun. ACM* 34 (1991), February, S. 56–78. – ISSN 0001–0782

[14] Ernits, J. ; Roo, R. ; Jacky, J. ; Veanes, M.: Model-Based Testing of Web Applications Using NModel. In: *Proceedings of the 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop.* Berlin, Heidelberg : Springer-Verlag GmbH, 2009 (TESTCOM '09/FATES '09). – ISBN 978–3–642–05030–5, S. 211–216

[15] Franz, K.: *Handbuch Zum Testen Von Web-applikationen: Testverfahren, Werkzeuge, Praxistipps*. Springer-Verlag GmbH, 2007. – ISBN 978–3–540–24539–1

[16] Frühauf, K. ; Ludwig, J. ; Sandmayr, H.: *Software-Prüfung - Eine Anleitung zum Test und zur Inspektion*. Bd. 6. vdf Hochschulverlag AG, 2007. – 168 S.

[17] Häusler, S. ; F., Poppen ; et.al., Hausmann K.: Modellierung von Komplexität und Qualität als Faktoren von Produktivität in Desing-Flows für integrierte Schaltungen, HDE Verlag, 2007 (edaWorkshop 07)

[18] Hitz, M. ; Kappel, G.: *UML@Work, Von der Analyse zur Realisierung*. dPunkt Verlag, 1999

[19] Hsueh, M. ; Tasi, T. K. ; Iyer, R. K.: Fault Injection Techniques and Tools. In: *Computer* 30 (1997), April, S. 75–82. – ISSN 0018–9162

[20] Juszczyk, Lukasz ; Dustdar, Schahram: Programmable Fault Injection Testbeds for Complex SOA, 2010 (Proceedings of the 8th International Conference on Service Oriented Computing (ICSOC 10))

[21] Klaus, E.: *Fehlertoleranzverfahren*. Springer-Verlag GmbH, 1990

[22] Kleuker, S.: Qualitaetsorientierte Analyse von Modellierungsansaetzen. In: *Tagungsband zum ersten Elmshorner Wirtschaftsinformatiktag 2009*, F. Zimmermann, 2009, S. 61–70

[23] Lambertz, K.: Softwarefehler verursacht das teuerste "Feuerwerk" aller Zeiten / Verifysoft Technology GmbH. 2009. – Forschungsbericht

[24] Lamport, L. ; Shostak, R. ; Pease, M.: The Byzantine Generals Problem. In: *ACM Trans. Program. Lang. Syst.* 4 (1982), July, S. 382–401. – ISSN 0164–0925

[25] Molyneaux, I.: *The art of application performance testing*. O'Reilly Media, 2009. – ISBN 978–0–596–52066–3

[26] Peschel-Findeisen, T.: *Nebenlaeufige und Verteilte Systeme*. MITP-Verlag, 2005

[27] Ravi, S. ; Pierangela, S.: Access Control: Principles and Practices. In: *IEEE Communications Magazine* 32 (1994), S. 40–48

[28] Rennhard, M.: Automatisiertes Software Security-Testing. In: *IT-Security* (2008)

[29] Robinson, H.: Intelligent test automation. In: *Software Testing & Quality* (2000), S. 24 – 32

[30] Scambray, J.: Applicaton Security Fundamentals. In: *testing experience - The Magazine for Professional Testers* (2009), June, S. 15–18

[31] Schmidberger, Rainer: Analyse von Testprozessen in der Industrie. In: *Software Engineering (Workshops)* Bd. 106, GI, 2007 (LNI). – ISBN 978–3–88579–200–0, S. 85–90

[32] Siegwart, K.: *Empirische Untersuchung zur analytischen Qualitätssicherung in der Industrie*, Universität Stuttgart, Diplomarbeit, 2004

[33] Sneed, H. M. ; Seidl, R.: *Der Systemtest - Anforderungsbasiertes Testen von Software-Systemen*. Bd. 1. Hanser Fachbuchverlag, 2008

[34] Spillner, A.: *Basiswissen Testen von Software*. W3l Verlag, 2010. – ISBN 978–3–86834–012–9

[35] Stahl, T. ; Völter, M.: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Bd. 1. dpunkt.verlag

[36] Tanenbaum, A. ; Steen, van M.: *Distriubuted Systems: Principles and Paradigms*. Prentice Hall International, 2001

[37] Tian-yang, Gu ; Yin-sheng, Shi ; You-yuan: Research on Software Security Testing. In: *World academy of science, Engineering and Technlogy* 69 (2010). `http://www.waset.org/journals/waset/v69/v69-118.pdf`

[38] Tixeuil, S. ; Hoarau, W. ; Silva, L.: An Overview of Existing Tools for Fault-Injection and Dependability Benchmarking in Grids / LRI - CNRS UMR 8623 & INRIA Grand Large, Universit Paris Sud XI, France. 2006. – Forschungsbericht

[39] Tuerpe, S.: Security Testing: Turning Practice into Theory. In: *Software Testing Verification and Validation Workshop*, 2008

[40] Utting, M. ; Legeard, B. ; James, M. E. (Hrsg.): *Practical model-based testing: a tools approach*. Bd. 1. MORGAN KUAFMANN, 2006. – ISBN 978–0123725011

[41] Veanes, M. ; Campbell, C. ; Grieskamp, W. ; Schulte, W. ; Tillmann, N. ; Nachmanson, L.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In: Hierons, R. (Hrsg.) ; Bowen, J. (Hrsg.) ; Harman, M. (Hrsg.): *Formal Methods and Testing* Bd. 4949. Springer-Verlag GmbH, 2008. – ISBN 978–3–540–78916–1, S. 39–76

[42] Wappler, U. ; Fetzer, C.: *Hardware Fault Injection Using Dynamic Binary Instrumentation: FITgrind*. 2006. – In Proceedings Supplemental Volume of EDCC-6

[43] Weiser, M.: Program Slicing*. In: *IEEE Transactions on Software Engineering* 10 (1984), Juli, S. 352–357

[44] Weismann, B.: *Architekturzentrierte Modellgetriebene Softwareentwicklung - Fallbeispiel und Evaluierung*, TU Wien, Diplomarbeit, 2006

[45] Zyl, van J.: SOFTWARE TESTING in a small company: a case study / Department of Computer Science University of Pretoria. 2010. – Forschungsbericht