Institut für Rechnergestützte Ingenieursysteme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 3145

Konzeption und Implementierung eines OTX Test Frameworks für das emotive ODF

Hai-Lang Thai

Studiengang:	Informatik
Prüfer:	Prof. Dr. Dieter Roller
Betreuer:	Dr. Jörg Supke
begonnen am:	01.02.2011
beendet am:	03.08.2011
CR-Klassifikation:	J.6

Danksagung

Ich danke unserem Gott und meinem Herrn und Retter Jesus Christus. Weiterer Dank geht an Herrn Prof. Dr. Dieter Roller, der diese Arbeit überhaupt erst möglich gemacht hat. Auch meinen beiden Betreuern Herr Dr. Jörg Supke und Herr Truong-An Nguyen möchte ich für ihre Unterstützung besonderen Dank aussprechen. Nicht zuletzt möchte ich meiner Familie und allen meinen Geschwistern für alle Ermutigung und Bestärkung danken.

Inhaltsverzeichnis

1. Einleitung

Wir schreiben das Jahr 2016. Der Ausbau an der neuen Mikrochip-Fabrik in Reutlingen, die weltweit eines der modernsten ist, wird fertig gestellt sein. Es ist ein Tag wie jeder andere, an dem das genannte Bosch-Werk in 24 Stunden bis zu einer Millionen Mikrochips herstellt und seinen bescheidenen Teil dazu beiträgt[1], die Weltbevölkerung mit einem Bestand von weltweit insgesamt einer Milliarden Transistoren pro Kopf zu beglücken.[2]

Diese ungeheuer anmutende Zahl an Transistoren pro Kopf kann sich unserem Vorstellungsvermögen nur entziehen und bleibt unserem Bewusstsein als eine nichts sagende Zahl verschlossen – und doch hinterlässt sie bei uns das mulmige und etwas ungewisse Gefühl nicht zu verstehen, was es für Konsequenzen mit sich trägt bzw. was das wirklich für uns bedeutet.

Nur eines steht fest: Unsere Welt verändert sich zusehends in rasant steigenden Größenordnungen, die wir nicht überblicken, geschweige denn kontrollieren können.

Es ist eine Ironie des Schicksals, dass der Mensch die Informatik gebraucht, um die Komplexität zu bewältigen, die auch, oder sogar vor Allem gerade durch eben jene kultiviert und multipliziert wurde. Herr EDSGER WYBE DIJKSTRA sagt dazu:

> *"As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them."*
>
> *(Dijkstra, 1972)*

Wie dem auch sei, die Computertechnologie hält erfolgreich Vormarsch in alle Bereiche unseres täglichen Lebens. Jeder Durchschnittsbürger in den westlichen Ländern darf sich über schätzungsweise 80 Mikrochips erfreuen[3], die ihm in seinem Umfeld in allen alltäglichen und nicht-alltäglichen Belangen begleiten, und sogar mit Rat und Tat zur Hilfe stehen. Der Nutzen und die Unterstützung, die der Mensch von diesen „intelligenten“ Halbleiterschaltungen erhält, sind immens und nicht von der Hand zu weisen. Es war das Jahr 1958 als der Erfinder und Entwickler JACK KILBY diese Technologie erstmals der Öffentlichkeit vorstellte und es sollte ein halbes Jahrhundert dauern, bis sämtliche Systeme, in denen der Mensch lebt und agiert, von ihr durchdrungen werden. Heute ist ein Leben ohne den Computer kaum mehr vorstellbar. Doch trotz oder gerade wegen aller Euphorie ist Vorsicht geboten, denn:

---

[1] Dies ist kein fiktives Szenario. Bosch-Werk soll bis spätestens 2016 bis zu 100 Millionen Chips am Tag produzieren können. (Siehe [20]: Stuttgarter Nachrichten)

[2] Vgl. Originaltext: “Today, there are nearly a billion transistors per human” (Siehe [18]: Palmisano)

[3] Vgl. [18]: Kandel

"Kein Produkt menschlicher Intelligenz kommt fehlerfrei zur Welt. Wir formulieren Sätze um, trennen Nähte wieder auf, setzen Pflanzen um, planen Häuser neu und reparieren Brücken. Warum sollte es uns mit Software anders gehen?"

(Wiener, 1994)

Jeder Softwareentwickler wird, je länger er sich mit dem Entwerfen und Implementieren von Software beschäftigt, sicher nicht die Erfahrung machen, dass er mit der Zeit fähig wird, Systeme mit weniger oder keinen Fehlern zu entwickeln, sondern muss und wird nur mehr und mehr zu der Erkenntnis der Tatsache des eben Zitierten gelangen.

Ist diese Einsicht erstmal vollzogen, muss die rationale Konsequenz und Forderung, nämlich Software sorgfältig und gründlich zu testen, unbedingt beachtet und durchgeführt werden. Mit anderen Worten: Da Fehler in Software ein ungeschriebenes Gesetz sind, ist das Testen von Software absolut unabdingbar. Leider stellt sich heraus, dass selbst das Testen von Software nicht völlig fehlerfrei bzw. vollständig durchgeführt werden kann und somit in der Praxis nie ein völlig fehlerfreies Produkt entsteht. Die Testphase der Entwicklung eines Produktes kann bis zu 50% der Zeit beanspruchen und muss oft selbst nach der Auslieferung in der Wartung von Software weitergeführt werden. Unter Software-Testern sagt man sich sinngemäß:

„The last bug was found, when the last user of the software stopped using it."

Die vorliegende Arbeit will sich mit dem beschriebenen Thema des Software Testens speziell im Bereich der Fahrzeugdiagnose beschäftigen. Denn auch gerade in der Automobilindustrie werden in jüngster Zeit Mikrochips en masse verbaut. So hat ein gut ausgestatteter Mittelklassewagen bis zu 80 Steuergeräte die zahlreiche Elektronikkomponenten des Fahrzeuges vernetzt und verwaltet. Und es scheint erst der Anfang zu sein – Für die nächsten Jahre ist in den Märkten für Fahrzeugdiagnose ein Wachstum von bis zu 50% zu erwarten.

Wieder muss darauf aufmerksam gemacht werden, dass dieser hohe Grad an Komplexität von Software getestet werden muss! Ich will dazu kurz eine Anekdote bringen, die mir letzte Woche während einer Autofahrt widerfahren ist. Ich fuhr mit einem Bekannten gerade auf der Autobahn als plötzlich der Wagen uns bei 130 km/h durch Warntöne offenbar etwas mitteilen wollte. Mein Bekannter (Fahrer und Besitzer des Fahrzeugs) ignorierte diese Warnhinweise zunächst, schaltete nach einigen Momenten dann doch das Parkpilot-System des Fahrzeugs aus und bemerkte mit einem kurzen Lächeln: *„Das Parkpilot-System nervt manchmal, wenn es anfängt zu regnen"*. Offenbar hatte es angefangen zu regnen, und statt den Scheibenwischer an zu stellen, hielt der Wagen es für notwendiger uns über die Gefahr eines Zusammenstoßes mit den unzähligen Regentropfen zu informieren. Wir waren nur froh, dass das Parkpilot-System noch nicht so ausgereift ist, dass es bei bevorstehendem Crash eine Vollbremsung unternimmt.

Was sich in unserem glücklichen Fall als harmlos und amüsant erweist, kann bei kritischen Elektronikkomponenten wie *ABS* oder *ESP* Katastrophen bewirken. Umso wichtiger ist es, das Testen

und die Fehlersuche vor allem im Bereich der Fahrzeugdiagnose so optimal und prozesssicher wie möglich zu gestalten.

Das Ziel dieser Arbeit ist es, ein geeignetes *Test Framework* für das *ODF (Open Diagnostic Framework)* zu entwickeln. *ODF* ist eine Entwicklungsumgebung, mit der sich Diagnoseabläufe für Fahrzeuge spezifizieren, realisieren, validieren, dokumentieren, debuggen, testen und ausführen lassen. Das *Test Framework* soll Funktionen bereitstellen, die ein automatisiertes und wiederholbares Testen von den mit *ODF* erstellten Diagnoseabläufen ermöglicht.
In **Kapitel 2** werden einige Grundlagen über das Testen allgemein und die Kern-Technologien die das *ODF* verwendet besprochen.
Auf Basis dieser Grundlagen wird in **Kapitel 3** ein *Testing*-Ansatz konzipiert und analysiert sowie eine Konzeption für *Test Framework* entwickelt.
Kapitel 4 dokumentiert den Überblick über den Aufbau und die technischen Details der Implementierung des Frameworks.
Anschließend führt **Kapitel 5** kurz in die Verwendung des Frameworks bzw. die Durchführung von Tests ein.
Zum Schluss werden in **Kapitel 6** noch einpaar Verbesserungen vorgeschlagen, sowie Grenzen und Möglichkeiten aufgezeigt.

2. Grundlagen

2.1. Workflows

"The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules."

(WFMC Introduction)

Die einleitenden Worte, *"automation of a business process"*, des Zitats (oben) aus der Workflow-Einführung von *WFMC* sprechen für sich - bei *Workflow-Management* geht es im Wesentlichen um eine Automatisierung von Geschäftsprozessen.

Wir wollen uns zunächst näher mit dem Begriff Geschäftsprozess auseinandersetzen. Grundsätzlich stellt ein Geschäftsprozess den Prozess dar, der für die Entwicklung bzw. Fertigstellung eines bestimmten Produkts durchlaufen wird. Hierbei muss es sich nicht nur um materielle Güter handeln, sondern das Produkt kann auch eine Dienstleistung sein. In dieser Hinsicht gibt es Geschäftsprozesse schon seit Menschengedenken - sei es ob die ersten Menschen sich aus Feigenblättern Schürzen flochten, oder sich die heutigen Menschen in humanitären Organisationen betätigen. Der einzige Unterschied besteht darin, dass der Mensch und seine Ideen bzw. Prozesse viel komplexer und komplizierter geworden sind.

Wie aus dem eingangs erwähnten Zitat auch hervorgeht, kann während eines Geschäftsprozesses ein Austausch oder eine Weiterleitung von Dokumenten, Daten oder Aufgaben zwischen vielen Beteiligten in einem komplexen System stattfinden. Während es beispielsweise vor 2000 Jahren noch sehr einfach, leicht verständlich und überschaubar war, sind Geschäftsprozesse von Unternehmen in der heutigen Zeit hochkompliziert und, wie der Mensch nunmal fehlt und irrt, noch dazu sehr fehleranfällig. Im Laufe der Zeit stellt sich immer mehr heraus, dass der Preis und die Qualität eines Produktes nicht nur von den zur Entwicklung benötigten Ressourcen abhängen, sondern hauptsächlich auch von der Effizienz und Effektivität seines Geschäftsprozesses. Im Falle einer Dienstleistung gilt die Gleichung *'Produkt = Geschäftsprozess'* sowieso. Wegen der zunehmenden Komplexheit und Fehlerhaftigkeit, aber auch überaus großen Wichtigkeit von Geschäftsprozessen, können/sollten diese geplant, durchgeführt, wiederholt bzw. wiederverwendet und im Laufe der Zeit optimiert werden. Das *Workflow-Management* deckt alle der eben genannten Aspekte ab und ermöglicht aber vor allem auch die zu Anfang erwähnte Automatisierung von Geschäftsprozessen.

Hierzu werden der gesamte Ablauf und die Struktur eines Geschäftsprozesses auf ein spezielles Prozessmodell abgebildet, welches von einem entsprechenden *Workflow-Management-System* verwaltet und ausgeführt werden kann.

Obwohl Prozessmodelle von Workflows in den meisten Fällen in *XML* spezifiziert werden, eignen sich Workflows sehr gut dazu als ein Graph dargestellt zu werden. Die graphische Modellierung von

Workflows ist sehr intuitiv und auch für Nicht-IT-Spezialisten leicht erlernbar. Workflows spielen demnach eine gewichtige Rolle darin, die sogenannte *IT-Bussiness Gap* zu schließen.
Um dies zu erreichen, müssen alle relevanten Aspekte eines Geschäftsprozesses im Prozessmodell klar spezifiziert sein. Die nötigen Arbeitsschritte eines Geschäftsprozesses werden in der Workflow-Terminologie *Aktivitäten* genannt. *Aktivitäten* stellen also ausführbare Einheiten dar, die von menschlichen oder maschinellen Ressourcen durchgeführt werden. Neben den *Aktivitäten* definiert ein Prozessmodell auch deren zeitlichen Ablauf, sowie den nötigen Datenaustausch zwischen ihnen, und nicht zuletzt die benötigten Ressourcen. Die graphische Repräsentation eines Workflows würde *Aktivitäten* als Knoten darstellen und den zeitlichen Ablauf der *Aktivitäten* auf Kanten, bzw. sogenannte *Kontroll-Flows* abbilden. Nach ***[2]: Leymann*** lässt sich ein Geschäftsprozess bzw. ein Workflow im Wesentlichen durch 3 Dimensionen (*What?, Who?, With?)* beschreiben.

- ***What?* (Prozesslogik):** Die What?-Dimension fragt nach der zeitlichen Abfolge der einzelnen Aktivitäten und den in diesem Zusammenhang benötigten Daten. Mit anderen Worten definieren hier der Kontrollfluss und der Datenfluss die sogenannte Prozesslogik des Workflow-Modells.
- ***Who?* (Organisation):** Die Who?-Dimension legt fest, wer oder was die Durchführung einer Aktivität übernimmt. Dies kann abhängig von der Organisations-Struktur eine bestimmte einzelne Person sein, aber auch Gruppen und Rollen.
- ***With?* (IT Infrastruktur):** Die With?-Dimension bestimmt die für die zu erledigende Aktivität zugrunde liegende IT-Infrastruktur - also mit welchen Mitteln (Programme, Tools,...) wird eine Aktivität durchgeführt.

Ist ein Workflow-Modell hinsichtlich dieser Dimensionen genügend spezifiziert, so kann von diesem in einem *Workflow-Management-System* eine Instanz erzeugt werden, die letztendlich ausgeführt werden kann. Da Workflows oft in sehr großen und komplexen Szenarien eingesetzt werden, in denen Geschäftsprozesse über verschiedene IT-Systeme oder sogar über Unternehmensgrenzen hinweg ablaufen, muss ein *Workflow-Management-System* hochgradig interoperabel sein. Deshalb werden die Schnittstellen solcher Systeme in der Praxis häufig durch *Web Services* oder vergleichbare Technologien definiert, die eine möglichst lose Kopplung bereitstellen. Zudem hat die *Workflow Management Coalition (WfMC)* ein *Workflow-Referenz-Modell* spezifiziert, um auf dieser Basis *Workflow-Management-Systeme* zu entwickeln, die unter Anderem auch untereinander interoperabel sind. In der nächsten Abbildung ist der grobe Aufbau des Referenz-Modells zu sehen.

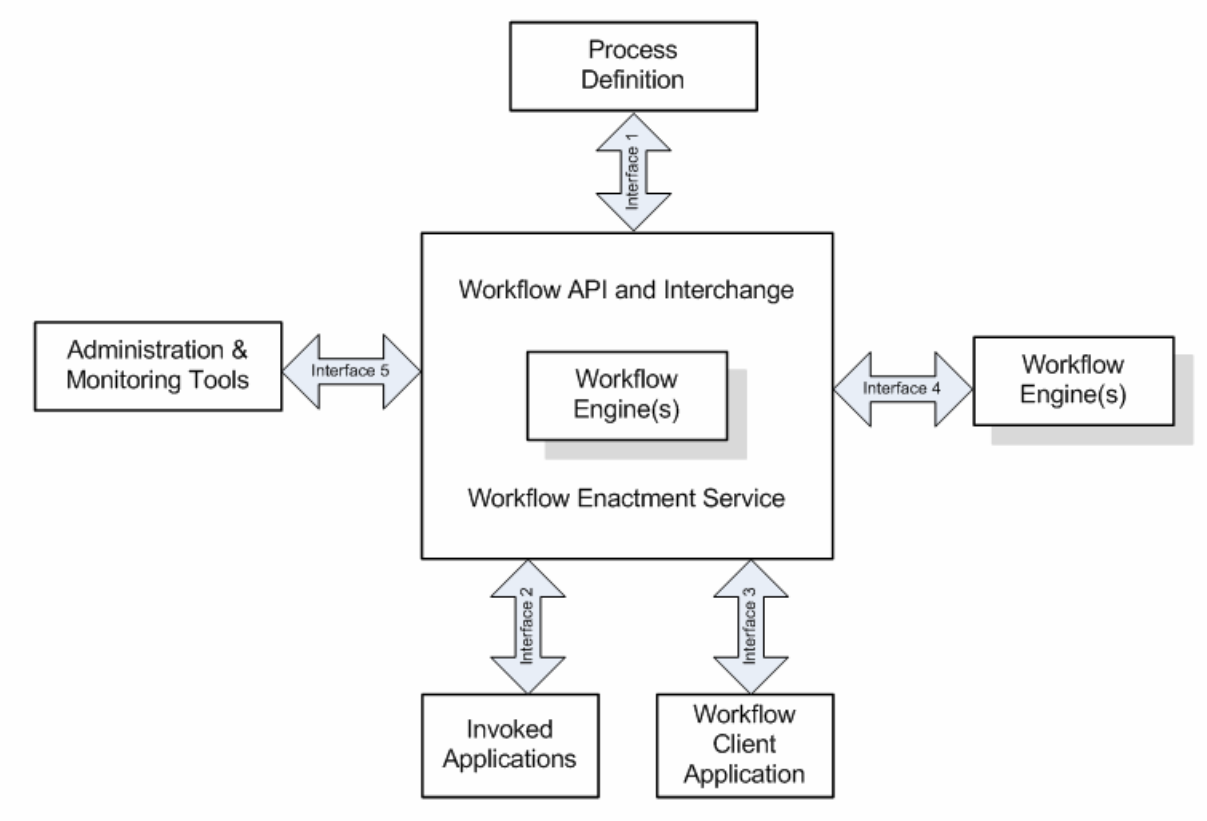

Abb. 1: Workflow-Reference-Modell. Quelle: [6]: WfMC

Die Funktionen der einzelnen Komponenten werden hier aufgelistet:

- Der **Workflow Enactment Service** stellt eine oder mehrere **Workflow Engine**s zur Verfügung. Dies ist die Laufzeitumgebung der ausführbaren Workflows - das Herzstück eines jeden *Workflow-Management-Systems*. Hier wird die Instanz eines Workflows ins Leben gerufen; hier werden alle An- und Abfragen verarbeitet; und hier wird die Koordination aller Partizipanten bewerkstelligt.
- Die **Process Definition** ist die Spezifikation eines Prozess-Modells. Es gibt hierzu viele verschiedene Tools, die die Entwicklung von Workflow-Modellen erleichtern. Workflow-Modelle werden in einer speziellen Workflow-Beschreibungssprache (häufig in einem *XML*-Derivat) spezifiziert.
- Über eine **Workflow Client Application** können sich menschliche Partizipanten ihre sogenannte *worklist* abrufen, die zu bearbeitende Aktivitäten (*work-items*) anzeigt. Der *Workflow Enactment Service* muss dafür sorgen, dass *Aktivitäten* als entsprechende *work-items* in der *worklist* der richtigen Partizipanten angezeigt werden, sobald diese von einer Workflow-Instanz aufgerufen werden.
- Die Schnittstelle zu **Invoked Application** dient dazu Programme aufzurufen, die bestimmte *Aktivitäten* von Geschäftsprozessen implementieren und ausführen (maschinelle Ressource als *Who?*-Dimension). Es ist aber auch möglich sich vom Workflow durch diese Schnittstelle ein Programm zur Bearbeitung einer *Aktivität* aufrufen zu lassen (menschliche Ressource als *Who?*-Dimension mit maschineller Ressource als *With?*-Dimension).

- Das Interface 5 stellt dem System **Administration & Monitoring Tools** zur Verfügung mit Hilfe derer laufende und abgeschlossene Workflow-Instanzen mit all ihren Logging-, Statistik-, und *Audit-Trail* -Daten verwaltet, beobachtet und analysiert werden können.
- Wegen der unbedingten Forderung nach Interopabilität müssen *Workflow-Management-Systeme* wohldefinierte Schnittstellen für und zu anderen **Workflow Engines** bereitstellen. (Interface 4)

2.2. Grundlegendes über das Testen in der Softwaretechnik

Bevor wir uns speziell mit dem Testen von Workflow-Sprachen beschäftigen wollen, lohnt es sich einen Blick auf grundlegende Konzepte für das Testen allgemein in der Softwaretechnik zu werfen. Ich beginne mit einleitenden Worten über einige Grundbegriffe für das Testen von Software.

Das Testen von Software dient zur Qualitätssicherung des entwickelten Produkts. Das heißt, dass sichergestellt werden soll, dass ein Produkt die an ihn gestellten **Anforderungen** erfüllt. Der Begriff **Softwarequalität** lässt sich nach der ISO-Norm 9126 auf folgende Dimensionen aufspannen: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit. Dies ist schon auf dem ersten Blick als ein sehr breites Feld erkennbar und wir wollen uns hier auf die Funktionalität und speziell auf dessen Teilmerkmal, nämlich die **Richtigkeit** von Software, beschränken. Mit anderen Worten soll der Fokus für das Testen von Software in dieser Arbeit vordergründig darauf gerichtet sein, zu prüfen, ob die Software das richtige bzw. das von ihm geforderte, spezifizierte Verhalten zeigt. Mängel bzw. ein **Fehler** in der Software sind demnach Abweichungen von einem erwarteten, spezifizierten Verhalten der Software. Ein Fehler liegt vor, wenn eine Diskrepanz zwischen einem **Ist-Verhalten**, welches beim Testen festgestellt wird, und seinem vorher festgelegten **Soll-Verhalten** vorliegt. Wir wollen zwischen dem Fehler (engl. *fault*) und die daraus testbare bzw. feststellbare resultierende **Fehlerwirkung** (engl. *failure*) unterscheiden. Verschiedene Fehler können zu ein und derselben Fehlerwirkung führen. D.h. auch, dass ein Test eine Fehlerwirkung feststellen kann, während der Fehler selbst nicht bekannt ist. Hier lässt sich dann auch die Grenze zum *Debugging*-Prozeß ziehen, der Fehler lokalisieren und beheben soll, während beim Testen Fehler bzw. deren Fehlerwirkung "nur" festgestellt oder aufgedeckt werden sollen.

Des Weiteren ist es möglich, dass Fehler andere Defekte im Programm kompensieren - wir sprechen dann von einer **Fehlermaskierung**. Die Fehlerwirkung bleibt aus und der Fehler wird erst offenbar, wenn Fehler, die die Maskierung verursachen, behoben wurden.

Es ist daher einzusehen, dass selbst ein umfassender Test nicht immer alle Fehler bzw. deren Fehlerwirkungen auffinden kann. Tatsächlich ist es in der Praxis nicht einmal möglich einen **vollständigen Test** durchzuführen, der alle Möglichkeiten und Eventualitäten mit einschließt. Dies würde einen Testdurchlauf mit allen möglichen Kombinationen aller möglichen Eingaben

bedeuten, der eine kaum einzugrenzende (in Bezug auf die Anzahl an Eingaben exponentiell wachsende) Anzahl an Testfällen nach sich ziehen würde. Nüchtern betrachtend muss man sagen, dass durch ein *Software-Test* keine völlige **Fehlerfreiheit** gewährleistet werden kann.

Wir wollen den eben genannten Begriff eines *Testfalles* klarer definieren. Ein *Testfall* ist eine konkrete Ausprägung der Ausführung der zu testenden Software mit festgelegten **Eingaben** und **Randbedingungen**. Ferner wird durch einen *Testfall* eben diese Ausprägung *(Ist-Verhalten)* gegen das gewünschte, spezifische Verhalten *(Soll-Verhalten)* der Software geprüft. Die festgelegten Eingaben und Randbedingungen sowie das gewünschte Verhalten bzw. Ergebnis werden in der so genannten **Spezifikation** des *Testfalls* festgelegt.

Testfälle werden des Öfteren in **Test-Suiten** strukturiert bzw. gruppiert, die eine Funktion oder bestimmte Arten von Funktionen testen. Schließlich sorgt die Ausführung von mehreren *Test-Suiten* in einem **Testlauf** für die **Automatisierung** des Testens.

Ein **Test Framework** muss außerdem je nach Bedürfnis dafür Sorge tragen, dass *Testfälle* (oder *Test-Suiten* oder spätestens bei *Testläufen*) ohne Einfluss aufeinander ablaufen können, sodass deren Ergebnisse reproduzierbar sind und generell die **Wiederholbarkeit** von Tests gegeben ist. Das Testen von Software lässt sich im Laufe des Softwareentwicklungsprozess auf mehreren Ebenen durchführen. Die nachstehende Abbildung stellt dies in dem allgemein bekannten *V-Modell* dar.

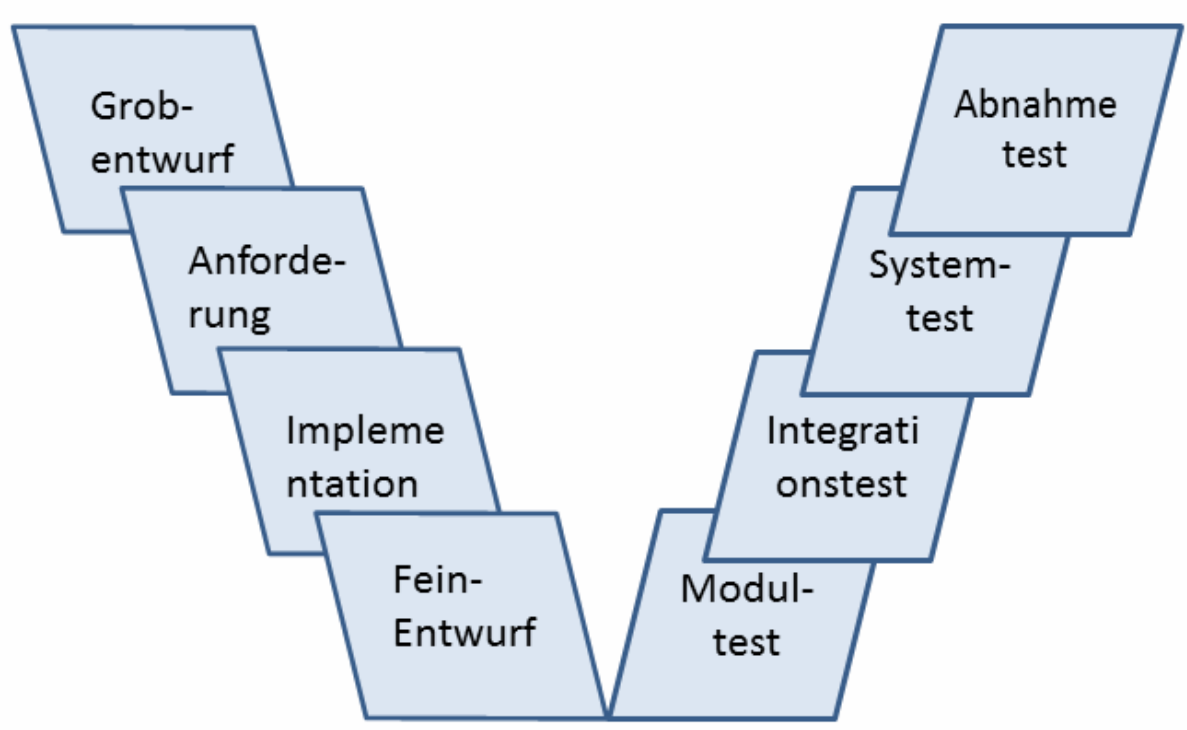

Abb. 2: V-Diagramm nach Böhm

Der **Modultest** oder auch **Unit-Test** wird meistens vom Entwickler durchgeführt um ein implementiertes Softwaremodul isoliert zu testen. Isoliert heißt, dass sämtliche Kommunikation mit dem Restsystem durch festgelegte statische Werte simuliert wird und das Testen auf diese Weise unabhängig und frei von Fehlereinflüssen anderer Module stattfinden kann.
Gefundene Fehlerwirkungen lassen so eine klare Zuordnung des Fehlers in das zu testende Modul zu. Ein solches Testen von kleinen Software-Einheiten (*Units*) ist viel überschaubarer und eine große Anzahl an Fehlern ist viel schneller zu finden und leichter zu beheben als etwa in einem systemweiten Test. Auf der anderen Seite werden im Modultest Fehler, die bei der Interaktion und aufgrund von Wechselwirkung zwischen vielen Komponenten entstehen, nicht berücksichtigt.
Diese Mängel werden in einem **Integrationstest** behoben. Hierbei wird davon ausgegangen, dass die einzelnen Softwaremodule für sich schon mehr oder weniger korrekt funktionieren. Teile der Software werden nun zu einem größeren Teilsystem aufgebaut und zusammen integriert getestet.
Ein **Systemtest** testet das integrierte Softwaresystem als Ganzes in einer möglichst produktionsnahen Umgebung. D.h., dass wenn möglich auf Testtreiber oder sonstige Platzhalter verzichtet wird, und möglichst dieselben Infrastrukturen (Sowohl Hardware- als auch Software -Komponenten) wie beim Kunden genutzt werden. Es wird evaluiert, ob die spezifizierten Anforderungen an das Produkt erfüllt wurden.
Der **Abnahmetest** ist mit einem Systemtest zu vergleichen, wird aber in einer Abnahmeumgebung beim Kunden oder komplett vom Kunden selbst durchgeführt. Dabei wird das abzunehmende Gesamtsystem daraufhin überprüft, ob aus Kundensicht die vertraglich festgelegten Anforderungen bzw. Abnahmekriterien erfüllt werden.

Im Folgenden wollen wir uns mit einigen bekannten Verfahren und Ansätzen beschäftigen, die sich vordergründig für *Unit-Tests* eignen. Die verschiedenen Ansätze lassen sich grob in zwei Kategorien, nämlich das *Black-Box* Verfahren und das *White-Box* Verfahren, einteilen.

2.3. Das Black-Box Verfahren

Bei dem *Black-Box* Verfahren werden ausschließlich nur die eingehenden und ausgehenden Daten der zu testenden *Unit* für einen Test mit einbezogen. *Input/Output* Daten werden mit den bezüglich des *Soll-Verhaltens* zu erwarteten Werten verglichen. Die Kenntnis der internen Logik der Software wird dabei nicht gebraucht und muss nicht einmal verfügbar sein (z.B. nur *Binary* ohne Quellcode). Was allein interessiert sind die Eingabe-Daten und die darauf berechneten Ausgabe-Daten, die mit den spezifizierten Werten verglichen werden.
Da ein *vollständiger Test* mit allen möglichen Eingaben praktisch nicht umsetzbar ist, müssen Verfahren verwendet werden, die die Anzahl der Testfälle stark eingrenzen. Eben zu diesem Zweck werden im Folgenden die *Äquivalenzklassenbildung* und die *Grenzwertanalyse* vorgestellt.

Äquivalenzklassenbildung

Ziel und Zweck der Äquivalenzklassenbildung ist die Verminderung der exponentiell explodierenden Anzahl an Testfällen bei einem vollständigen Test.

Das Verfahren der Äquivalenzklassenbildung teilt Eingabebereiche in Äquivalenzklassen auf. Dabei wird die Einteilung so vollführt, dass Eingabewerte, die in dieselbe Äquivalenzklasse fallen, ein gleiches Verhalten bzw. die gleiche Verarbeitung in dem zu testenden Programmstück auslösen sollen. Getestet wird dann nur je ein Repräsentant bzw. nur einige wenige Repräsentanten einer Äquivalenzklasse - und es kann mit einer tendenziell großen Wahrscheinlichkeit daraus geschlossen werden, dass alle oder die meisten anderen Elemente der jeweiligen Äquivalenzklasse ebenso dieses Testverhalten aufweisen. Die Testfälle beschränken sich von allen möglichen Kombinationen von Eingabewerten auf alle möglichen Kombinationen von Äquivalenzklassen, was in den meisten Fällen eine Verminderungen um unzählbar viele Größenordnungen bedeutet und so überhaupt erst einen *quasi-vollständigen* Test durchführbar macht.

Die *Partitionierung* oder Aufteilung in Äquivalenzklassen lässt sich aus der funktionalen Anforderung an die Software bzw. dessen Spezifikation herleiten. Dabei ist zu beachten, dass neben Eingabedaten, die der Spezifikation nach in einem gültigen Definitionsbereich liegen, auch unzulässige Eingabewerte berücksichtigt werden sollten. Es werden also ebenso Äquivalenzklassen mit ungültigen Werten gebildet, um das Verhalten mit unzulässigen Eingabedaten zu testen.

Es bleibt anzumerken, dass eine Äquivalenzklassenbildung aus der Spezifikation der Software keine 100%-ige Zusicherung dafür geben kann, dass alle Äquivalenzklassenelemente im Test mit der tatsächlichen Implementierung letztendlich das erwartete Verhalten zeigen. Der Entwurf und die Spezifikation der Software haben ein um mehrere Ebenen höheres Abstraktionsniveau als die Implementierung der Software, was zu einer in der Implementierung unter Umständen komplett verschiedene Verarbeitung von Eingabewerten führt, obwohl diese der Spezifikation nach in derselben Äquivalenzklasse liegen. Um dies zu vermeiden, müsste die Entwicklung der Äquivalenzklassen auf Grundlage der internen Ablauflogik des Programms erfolgen, was aber dann als ein White-Box Verfahren angesehen werden müsste.

Grenzwertanalyse

Die Grenzwertanalyse ist eine sinnvolle Verbesserung des eben vorgestellten Ansatzes der Äquivalenzklassenbildung. Wie der Name schon verlauten lässt, geht es hierbei darum, Grenzwerte bzw. Grenzbereiche der Äquivalenzklassen näher zu untersuchen. Die Praxis zeigt, dass vor allem Grenzwerte der Äquivalenzklassen durch fehlerhafte Implementierung nicht korrekt bzw. nicht wie erwünscht verarbeitet werden, weil sie dann in eine benachbarte Äquivalenzklasse fallen. Man stelle sich ein ganz simples Beispiel vor, in der eine Fallunterscheidung in der Implementierung statt mit *'kleiner gleich'* nur mit *'kleiner'* geprüft wurde. Das zusätzliche Testen von Testfällen mit Grenzwerten

der Äquivalenzklassen birgt einen gut realisierbaren Mehraufwand, der einen großen Teil von Fehlern aufdeckt.

2.4. Das White-Box Verfahren

Allgemein bezeichnet der Begriff *White-Box* ein Testverfahren, welches die Software unter Zuhilfenahme der inneren Logik und Struktur testet. Der bekannteste Ansatz ist, durch ein Verfahren festzustellen bzw. sicherzustellen, dass die Testabdeckung möglichst hoch ist, also möglichst viel Programmcode getestet wurde. Im Idealfall soll eine 100%ige sogenannte *Code-Coverage* erreicht werden.

Code coverage

Verschiedene Kriterien und Metriken werden hierbei definiert, um zu bestimmen in wie fern ein Programm hinsichtlich seines Quellcodes vollständig getestet wurde. Es gibt in der Literatur dazu vier verschiedenen Kriterien, die am folgenden Beispiel erklärt und nachvollzogen werden können.

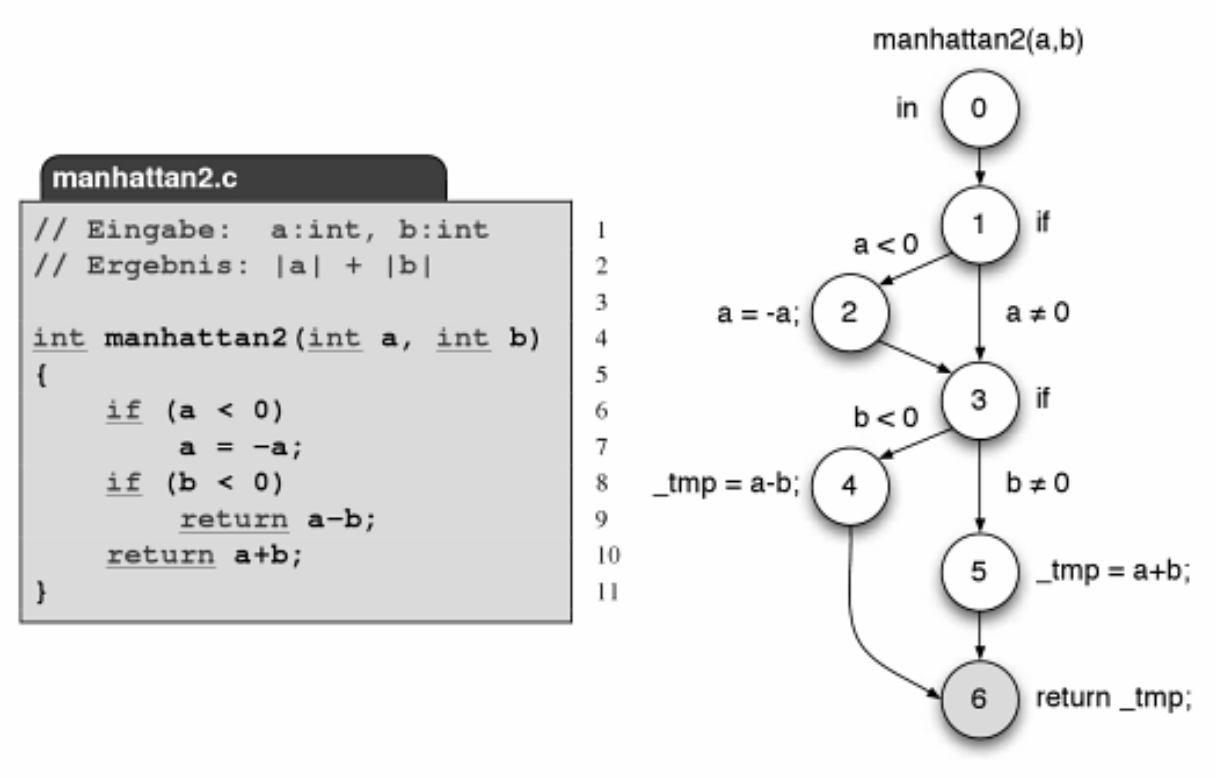

Abb. 3: Programmcode & Flussdiagramm. Quelle: [2]: Hoffmann, S.204

statement coverage

Kriterium: Jede Anweisung wird mindestens einmal ausgeführt.

Minimale Anzahl Testfälle für vollständige C0-Überdeckung: 2
Testfälle: manhattan2(-1, -1) => {0, 1, 2, 3, 4, 6}
manhattan2(-1, 0) => {0, 1, 2, 3, 5, 6}

branch coverage

Kriterium: Jeder Zweig wird mindestens einmal ausgeführt.

Minimale Anzahl Testfälle für vollständige C1-Überdeckung: 2
Testfälle: manhattan2(-1, -1) => {0, 1, 2, 3, 4, 6}
manhattan2(0, 0) => {0, 1, 3, 5, 6}

condition coverage

Erweiterung der ersten Bedingung bei Knoten 1 auf **((a<0) && (a<b))**

Einfache Bedingungsüberdeckung
Kriterium: Jedes atomare Wahrheitsprädikat einer Bedingung muss mindestens einmal *'true'* und einmal *'false*' getestet werden.

Minimale Anzahl Testfälle für vollständige C2-Überdeckung: 2

Testfälle	a < 0	b < 0	a < b	**(a<0) && (a<b)**	
manhattan2(0, 1)	false	false	true	false	**=> {0, 1, 3, 5, 6}**
manhattan2(-1, -2)	true	true	false	false	=> {0, 1, 3, 4, 6}

Minimale Mehrfachbedingungsüberdeckung
Kriterium: Jedes atomare oder zusammengesetzte Wahrheitsprädikat einer Bedingung muss mindestens einmal *'true'* und einmal *'false'* getestet werden.

Minimale Anzahl Testfälle für vollständige C2-Überdeckung: 2

Testfälle	a < 0	b < 0	a < b	**(a<0) && (a<b)**	
manhattan2(0, -1)	false	true	false	false	**=> {0, 1, 3, 4, 6}**
manhattan2(-1, 0)	true	false	true	true	**=> {0, 1, 2, 3, 5, 6}**

<table>
<tr><td></td><td>
<u>Mehrfachbedingungsüberdeckung</u>
Kriterium: Jede Kombination mit 'true' bzw. 'false' aller atomaren Wahrheitsvariablen einer Bedingung.

Minimale Anzahl Testfälle für vollständige C2-Überdeckung: nicht möglich
<table>
<tr><th><u>Testfälle</u></th><th>a < 0</th><th>b < 0</th><th>a < b</th><th>(a<0) && (a<b)</th><th></th></tr>
<tr><td>manhattan2(1, 0)</td><td>false</td><td>false</td><td>false</td><td>false</td><td>=> {0, 1, 3, 5, 6}</td></tr>
<tr><td>manhattan2(0, 1)</td><td>false</td><td>false</td><td>true</td><td>false</td><td>=> {0, 1, 3, 5, 6}</td></tr>
<tr><td>manhattan2(0, -1)</td><td>false</td><td>true</td><td>false</td><td>false</td><td>=> {0, 1, 3, 4, 6}</td></tr>
<tr><td>manhattan2(?, ?)</td><td>false</td><td>true</td><td>true</td><td>false</td><td>=> {0, 1, 3, 4, 6}</td></tr>
<tr><td>manhattan2(?, ?)</td><td>true</td><td>false</td><td>false</td><td>false</td><td>=> {0, 1, 3, 5, 6}</td></tr>
<tr><td>manhattan2(-1, 0)</td><td>true</td><td>false</td><td>true</td><td>true</td><td>=> {0, 1, 2, 3, 5, 6}</td></tr>
<tr><td>manhattan2(-1, -2)</td><td>true</td><td>true</td><td>false</td><td>false</td><td>=> {0, 1, 3, 4, 6}</td></tr>
<tr><td>manhattan2(-2, -1)</td><td>true</td><td>true</td><td>true</td><td>true</td><td>=> {0, 1, 2, 3, 4, 6}</td></tr>
</table>
</td></tr>
<tr><td>Path coverage</td><td>
Kriterium: Jeder Pfad wird mindestens einmal ausgeführt.

Minimale Anzahl Testfälle für vollständige C0-Überdeckung: 4
Testfälle: manhattan2(0, 0) => {0, 1, 3, 5, 6}
manhattan2(0, -1) => {0, 1, 3, 4, 6}
manhattan2(-1, 0) => {0, 1, 2, 3, 5, 6}
manhattan2(-1, -1) => {0, 1, 2, 3, 4, 6}
</td></tr>
</table>

Tabelle 1: Code Coverage Beispiel

<u>Testcase Generation</u>

Im Hinblick auf *Code Coverage* können automatisiert Testfälle generiert werden, die im optimalem Fall eine 100%ige *Code Coverage* bieten. Dabei wird der Programmcode in einem ersten Schritt meistens in ein spezifisches *Kontrollflussgraphen*-ähnliches Modell transformiert. Anschließend kann durch einen geeigneten Graphendurchlauf die nötigen Bedingungen an den Verzweigungen gesammelt werden.

Für *Path Coverage* etwa könnte man den Programmcode auf eine Baumstruktur abbilden und darauf eine komplette Tiefensuche durchführen, die die Bedingungen sammelt; jedes Mal wenn ein Blatt erreicht wird, werden die aktuellen Bedingungen von Blatt bis zur Wurzel festgehalten.

Zuletzt werden die Bedingungen in einem *Constraint*-System zusammengefasst und dadurch die passenden Werte bzw. Wertebereiche für die Eingabe-Daten der einzelnen Testfälle berechnet.

2.5. Testen von Workflows

Obgleich Workflows in dem Bereich des imperativen Programmierparadigmas einzuordnen sind, unterscheiden sie sich doch stark von klassischen imperativen Programmiersprachen. Wie im vorherigen Abschnitt dargestellt wurde, werden Workflows zumeist genutzt um wiederkehrende Abläufe/Prozesse zu organisieren und vor allem zu automatisieren. Es geht hier insbesondere nicht hauptsächlich darum mathematische Berechnungen durchzuführen und Ergebnisse zurückzuliefern, sondern - wenn überhaupt - Ergebnisse solcher Berechnungen zu sammeln, um diese, nach einem definierten Ablauf, zur Weiterverarbeitung an einen geeigneten Verarbeiter weiterzuleiten.
Die Semantik von Workflows spiegelt sich demnach im Allgemeinen also nicht in den Ergebnissen, die ein Workflow zurückliefert, wider, sondern in dessen Ablauf und den durchgeführten Aktivitäten. Möchte man die Korrektheit eines Workflows in Hinsicht auf eine geforderte Semantik verifizieren, reicht es folglich nicht aus, den Workflow in einem reinen *Black-Box* Verfahren auf seine Eingabe- und Ausgabe-Daten hin zu testen.
Tatsächlich ist es so, dass Ausgabeparameter eines Workflows oftmals nur eine Bestätigungsnachricht beinhalten oder gar gänzlich fehlen - und somit nicht selten gar keine Relevanz oder nicht genügend Indizien für eine Korrektheitsaussage über den Workflow haben. Dies ist insbesondere der Fall, wenn ein Ablauf ausgeführt wird, ohne dass Ergebnisse oder Teilergebnisse den Aufrufer interessieren und erwartet werden. Um dies zu veranschaulichen wollen wir ein simples Beispiel betrachten.

Selbst wenn auf dem ersten Blick aussagekräftige Ausgabeparameter definiert sind und das *Black-Box* Verfahren die Korrektheit des Workflows verifiziert, ist es möglich, dass die interne Logik anders arbeitet als es erwartet wird. Betrachten wir nachfolgenden abstrakten Workflow mit einem Testfall, dessen Eingabeparameter die Bedingungen *Condition 2* und *Condition 7* zu *'true'* auswertet; der Ausgabeparameter nimmt demnach den Wert *'success'* an. Der dann nach der Spezifikation imaginär ablaufende Pfad ist grün gekennzeichnet. Nun nehmen wir an, dass *Condition 1* und *Condition 2* falsch implementiert wurden und der tatsächliche Pfad so verläuft, wie die roten Pfeile es kennzeichnen. Der Testfall würde im *Black-Box* Verfahren als korrekt verifiziert. Der interne Ablauf ist allerdings fehlerhaft. Die Aktivität *Task 3* sollte ausgeführt werden; tatsächlich wurde aber stattdessen *Task 2* ausgeführt.

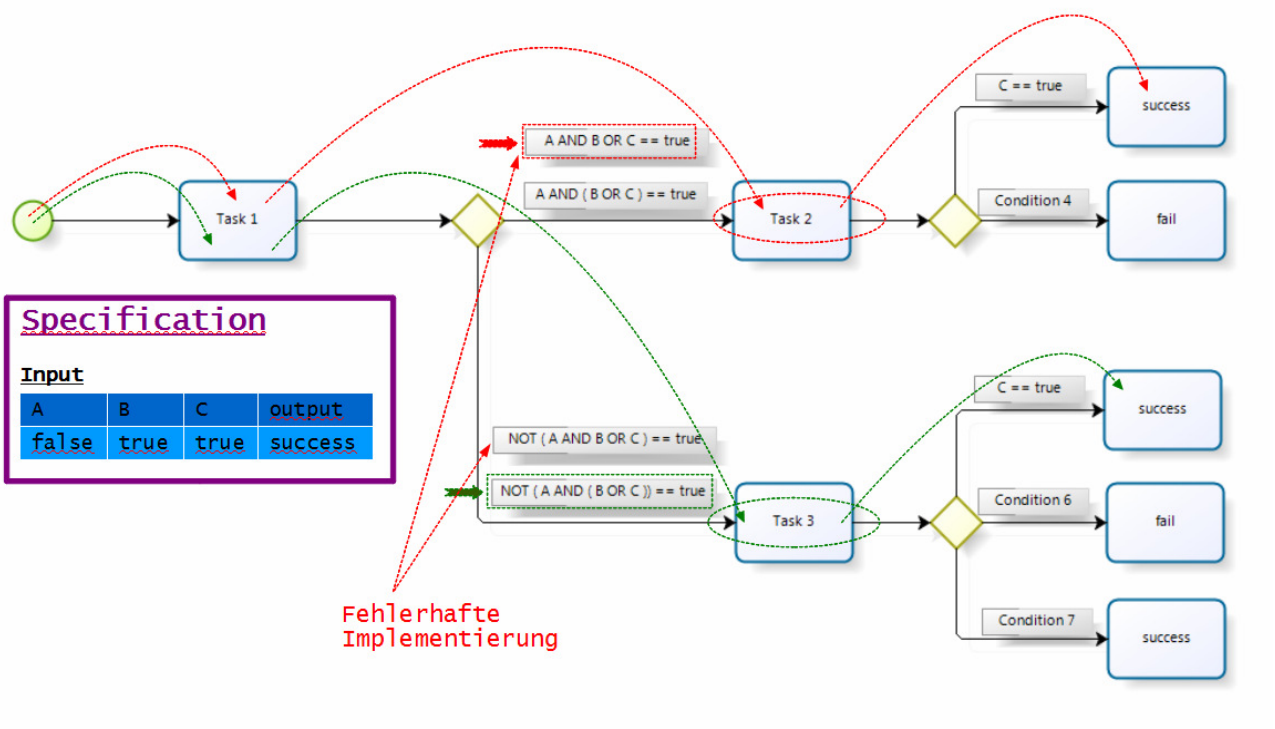

Abb. 4: Workflow, Black-Box Test

Wir können also leicht einsehen, dass es in vielen Fällen schwierig bis gar nicht möglich ist, durch ein paarweises Vergleichen der zu erwartenden Werten mit den Eingabe- und Ausgabe-Größen eines ausgeführten Workflows, die Korrektheit eines Workflows zu überprüfen.Man kann Eingabe- und Ausgabeparameter verifizieren und doch ist unter Umständen noch keine Aussage darüber getroffen, ob der Workflow tut, was er soll. Da es klar ist, dass ein reiner *Black-Box* Ansatz für das Testen von Workflows ungenügend ist, werde ich im Folgenden die aus der Literatur bekannten und aktuellen Ansätze eines *White-Box Testings* für Workflows vorstellen.

Code coverage & Test case generation

Im Hinblick auf *Code coverage* können automatisiert *Testcases* generiert werden, die im optimalen Fall eine auf eine spezifische Metrik bezogene 100%ige *Code Coverage* bieten. Hierbei wird die Workflow-Beschreibungssprache in einem ersten Schritt in ein simples graphenbasiertes Meta-Modell transformiert, um anschließend auf dem resultierenden Graphen durch eine Tiefensuche oder andere *Path Search* Algorithmen alle Pfade abzulaufen. Alle Bedingungen, die bei den Verzweigungen in einem Pfad auftreten, werden zu einem *Constraint*-System zusammengefasst und zu einem *Testcase* ausgewertet. Dabei kann es durchaus vorkommen, dass nicht alle Eingabe-Werte voll definiert werden können, da die entsprechende Variable oder nur ein Teil ihres Wertebereiches einen Einfluss auf den Verlauf des Pfades hat. Diese nicht relevanten, variablen Daten können manuell festgelegt werden oder automatisiert durch Zufallswerte generiert werden. Es ist vernünftig die erwarteten Ausgabewerte manuell zu spezifizieren, um die Korrektheit der Semantik des Workflows nicht auf Grundlage des eventuell fehlerhaft modellierten Workflows nachzuprüfen. Nichtsdestotrotz ist es auch möglich die Ausgabeparameter automatisch berechnen zu lassen und somit einen *Testcase* vollautomatisiert

generieren zu lassen. Selbstverständlich müssen bei solchen *Testcases* dessen generierte Ausgabeparameter der Semantik des Workflows entsprechend überprüft bzw. korrigiert werden.

Diese Art von *White-Box* Test - nämlich Testfälle auf Grundlage des Programmcodes zu erstellen - wird auch für das Testen von Programmen konventioneller Programmiersprachen eingesetzt; indem die generierten Testfälle im Anschluss an das *White-Box* Verfahren als Eingabe- und Ausgabe-Werte für einen nachfolgenden *Black-Box* Ansatz fungieren.
Wie wir schon festgestellt haben, eignet sich der *Black-Box* Ansatz für das Testen von Workflows in vielen Fällen jedoch nur unzureichend. Dies ändert sich auch dann nicht, wenn Eingabe- und Ausgabe-Parameter durch ein *White-Box* Verfahren generiert wurden. Die besprochene Testfall-Generierung stellt "nur" die Zusicherung, dass im optimalen Fall alle möglichen Pfade abgedeckt sind - sagt aber über die Korrektheit des Workflows erst dann etwas aus, wenn die Spezifikation der Ein- und Ausgabeparameter die gewünschte Semantik widerspiegeln. Eine vollautomatische Generierung dieser Daten kann niemals Fehler des zugrunde liegenden Programms aufdecken. Zumindest die Ausgabe-Daten eines Testes müssen manuell der Semantik entsprechend spezifiziert werden, um einen aussagekräftigen Test durchzuführen.
Es ist sicher ein großer Gewinn durch dieses Verfahren Eingabe-Daten zu erhalten, die eine gute *Code-Coverage* bereitstellen; allerdings erst dann effektiv, wenn die generierten Eingaben mit einem geeigneten Mittel kombiniert werden, welches die Korrektheit des inneren Ablaufs sicherstellt.

Kommunikations-Protokoll

In *"BPEL4WS Unit Testing: Framework and Implementation"*[4] stellen LI ET AL. einen interessanten *White-Box* Ansatz vor, der sich hauptsächlich auf die Interaktion zwischen dem *PUT (Process under Test)*, d.h. dem zu prüfenden Workflow, und seiner Partner/externen Services bezieht. Dass dieser Ansatz wahrscheinlich effektiv ist, lässt sich aus der logischen Konsequenz schließen, dass Workflows, wie schon beschrieben, oftmals Abläufe modellieren, die viele externe Services aufrufen bzw. mit ihnen interagiert.
In diesem Ansatz werden neben den Eingabe- und Ausgabe-Daten für einen Testfall zusätzlich noch eine Art Kommunikations- oder Aufrufprotokoll spezifiziert, welches bestimmt in welcher Reihenfolge Aufrufe an externe Partner stattfinden. Stimmen die Aufrufe bei der Ausführung des Workflows nicht mit dem Protokoll überein, ist der modellierte Workflow offensichtlich fehlerhaft. MAYER hat diesen Ansatz in *BPEL-Unit*[5] implementiert. Nachfolgend sehen wir die Struktur einer Spezifikation für einen Testfall.

[4] [7]: Li et. al., 2005
[5] Test Framework für BPEL (Siehe [4]: Mayer, 2006)

```
<testCases>
     <testCase name="Travel Test">
          <property name="useCase">245</property>

          <clientTrack>
               ...
          </clientTrack>

          <partnerTrack name="Airline">
               ...
          </partnerTrack>
     </testCase>
</testCases
```

Abb. 5: Testfall Specification BPEL-Unit

2.6. OTX - Open Test sequence eXchange (ISO 13209)

OTX (Open Test sequence eXchange) wurde als ein *XML*-basiertes, Plattform und Tester unabhängiges Austauschformat modelliert. Dieser in der ISO 13209 spezifizierter Standard wurde mit dem Ziel entwickelt, im Bereich der Fahrzeugdiagnose Testsequenzen mit einem hohen Abstraktionsniveau graphisch modellieren zu können, zu spezifizieren und auch auszuführen.

Da Testsequenzen für die Fahrzeugdiagnose im Grunde nichts anderes sind als Geschäftsprozesse oder allgemeine Prozesse, kann man sagen, dass *OTX* im Wesentlichen eine domänen-spezifische Workflow-Beschreibungssprache ist.

2.6.1. Hintergrund – Exkurs : Fahrzeugdiagnose

Um zu sehen welche Rolle *OTX* in der Fahrzeugdiagnose spielt, wollen wir an dieser Stelle einen kurzen Exkurs in die Fahrzeugdiagnose antreten, um uns zur Standortsbestimmung sowie Daseinsberechtigung von *OTX* eine grobe Übersicht zu verschaffen.

Zu diesem Zweck zeigt die folgende Abbildung einen typischen Testablauf in einem modernen und etablierten Applikations- und Diagnosesystem *(ASAM).*

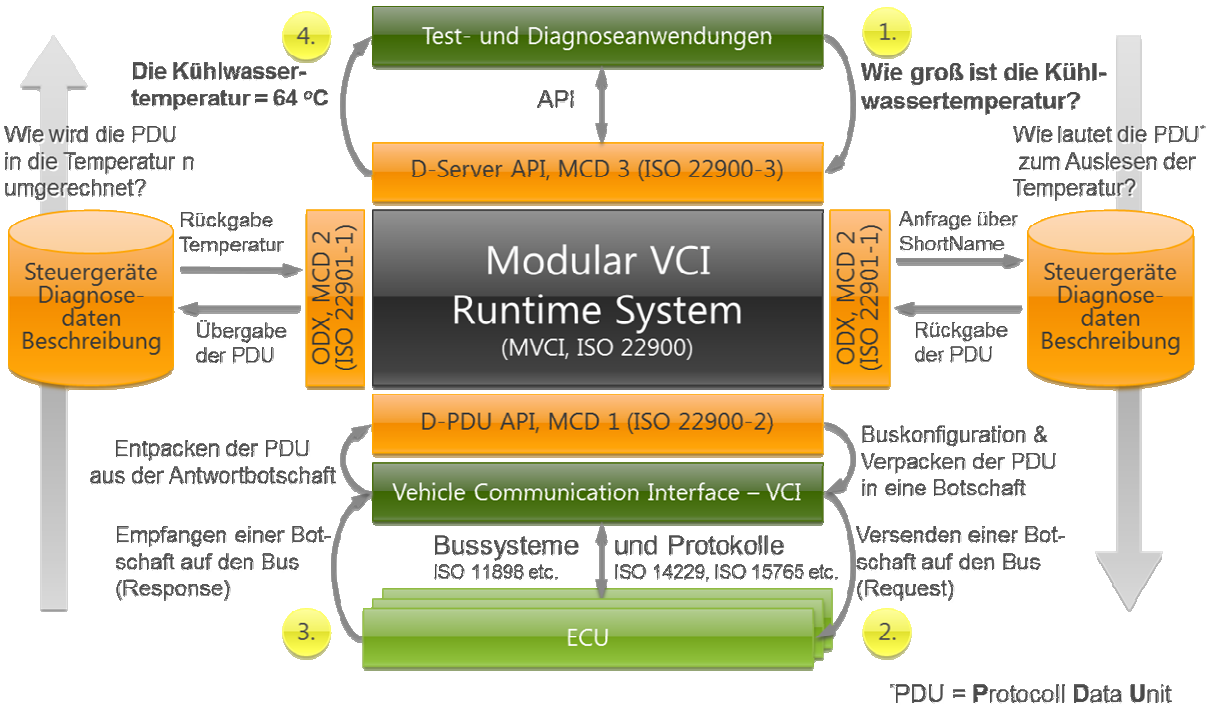

Abb. 6: State of the Art – Diagnoseablauf in ASAM-System [8]:

Was auf den ersten Blick etwas kompliziert aussehen mag, lässt sich schnell simplifizieren, indem diese Darstellung in 5 Komponenten aufgeteilt wird:

- **ECU's** *(Eletronic Control Unit)* stellen die Steuergeräte im Fahrzeug dar, mit Hilfe derer man die Elektronik im Fahrzeug ansteuern kann. Zusätzlich ist es möglich auf die im Fahrzeug gespeicherten Daten zuzugreifen bzw. sie zu modifizieren.
- Über externe Hardware-Testgeräte, den so genannten **VCI's**, wird mittels verschiedenen Bussystemen und Protokollen die Verbindung und Kommunikation zu diesen Steuergeräten hergestellt.
- Das **MVCI Runtime System** ist eine *Abstraction-Layer* über verschiedenen *VCI's*. Außerdem stellt das *MVCI* die typischen und standardisierten Funktionsaufrufe für das Versenden, Empfangen und Umrechnen in einer *API* zur Verfügung (*MCD3*-Schnittstelle) und ermöglicht so einen generischen Zugriff auf die Hardware(*MCD1*-Schnittstelle).
- Durch das standardisierte Diagnose-Datenaustauschformat **ODX** (Open Diagnostics data eXchange) werden die für die Diagnose relevanten fahrzeug- und steuergeräte-spezifischen Daten in eine Datenbank ausgelagert. Man erreicht durch diese Auslagerung eine Entkopplung der Diagnoseanwendung und den spezifischen Steuergerätevarianten in verschiedenen Fahrzeugmodellen verschiedenster Fahrzeugherstellern. Die *ODX*-Datenbank ist durch die *MCD2*-Schnittstelle mit dem *MVCI*-System verbunden.
- Ganz oben sitzt die **Diagnoseanwendung,** die sehr individuelle und hochspezialisierte Diagnoseabläufe implementiert.

Der Vorteil eines solchen Diagnoselaufzeitsystems liegt auf der Hand und sollte schon durch die eben aufgeführte Aufteilung hindurchgeschienen sein.
Auf Grundlage der Standards *ODX* und *D-PDU API* sorgt ein *MVCI*-System für die komplette interne Kommunikationslogik zwischen der Diagnose-Testanwendung bis hin zum Steuergerät. Komplexe Zusammenhänge von Diagnose.Kommunikationsprotokollen werden verborgen. Die Testanwendung muss sich nicht um Diagnose- und Transportprotokolle kümmern, d.h. der Transport-Layer und selbstverständlich auch die darunterliegenden Schichten des *OSI*-Schichtenmodells werden für den Tester komplett transparent und theoretisch irrelevant. In der vorangegangenen Abbildung sind die einzelnen Schritte eines Diagnose-Funktionsaufrufs dargestellt, die von einer Tester-Applikation ausgeführt werden müssten, wenn kein *MVCI Runtime System* vorhanden ist. Diese werden hier kurz noch mal erläutert.

1. Die Diagnoseanwendung stellt über die D-Server API eine Anfrage an das *MVCI*-System die Kühlwassertemperatur auszulesen.
2. Das System startet eine Abfrage an die Diagnosedatenbank (*ODX*), um die, an das Steuergerät zu sendende, Busbotschaft (auch *PDU* genannt) zu ermitteln. Ist die entsprechende, für die betreffende Steuergeräte-Variante spezifische *PDU* für das Auslesen der Temperatur gefunden, wird diese an das *VCI* weitergegeben. Anschließend sendet das *VCI* die *PDU* gepackt in einem spezifischen Diagnose-Request durch das Bussystem an das Steuergerät.
3. Das *VCI* erhält die Diagnose-Response und entpackt aus dieser wiederum eine *PDU*. Üblicherweise werden die Rückgabewerte in einem je nach Steuergerät spezifischen Hexdezimalwert codiert und können durch Informationen aus der *ODX*-Datenbank umgerechnet werden.
4. Das *MVCI*-System antwortet der Diagnoseanwendung mit der Botschaft: *“Kühlwassertemperatur = 64° C“*

Es gibt also im Wesentlichen drei Aufgabenbereiche, die ein *MVCI*-System in der Regel abdeckt.

1. Kommunikation mit Steuergerät durch Bussystem (*MCD1*-Schnittstelle).
2. Automatische Abfrage von *ODX*-Daten für Fahrzeugbaureihen zur Ermittlung der spezifischen Busbotschaften, sowie ggf. der Umrechnungsfunktionen (*MCD2*-Schnittstelle).
3. Laufzeitsystem, welches die einzelnen Komponenten koppelt und schließlich allen Mehraufwand trägt um Diagnoseabfragen zu realisieren. Dazu gehören, wie erwähnt, bspw. die zum Versenden, Empfangen und Umrechnen benötigten Algorithmen, die in einer *API* zur Verfügung gestellt wird (*MCD3*-Schnittstelle).

2.6.2. Motivation

OTX nun setzt da an, wo für gewöhnlich sich spezielle Diagnoseanwendungen befinden, die die *D-Server API* eines *MVCI*-Systems nutzen. *OTX* spezifiziert unter Anderem Diagnoseaufrufe an Steuergeräte, welche durch ein darunter liegendes *MVCI*-System realisiert werden können. Während *ODX* Daten beschreibt, die benötigt werden, um einen bestimmten Funktionsaufruf an ein bestimmtes Steuergerät durchzuführen, beschreibt *OTX* einen ganzen Diagnoseablauf samt den „verschiedenen Interaktionen zwischen einem Anwender (Entwicklungs-, Produktions- oder Werkstattpersonal), dem Diagnosetester, den Steuergeräten und ggf. der externen Messtechnik“[6]. ***Abb. 7: Abstrakter Diagnoseablauf*** verdeutlicht dies.

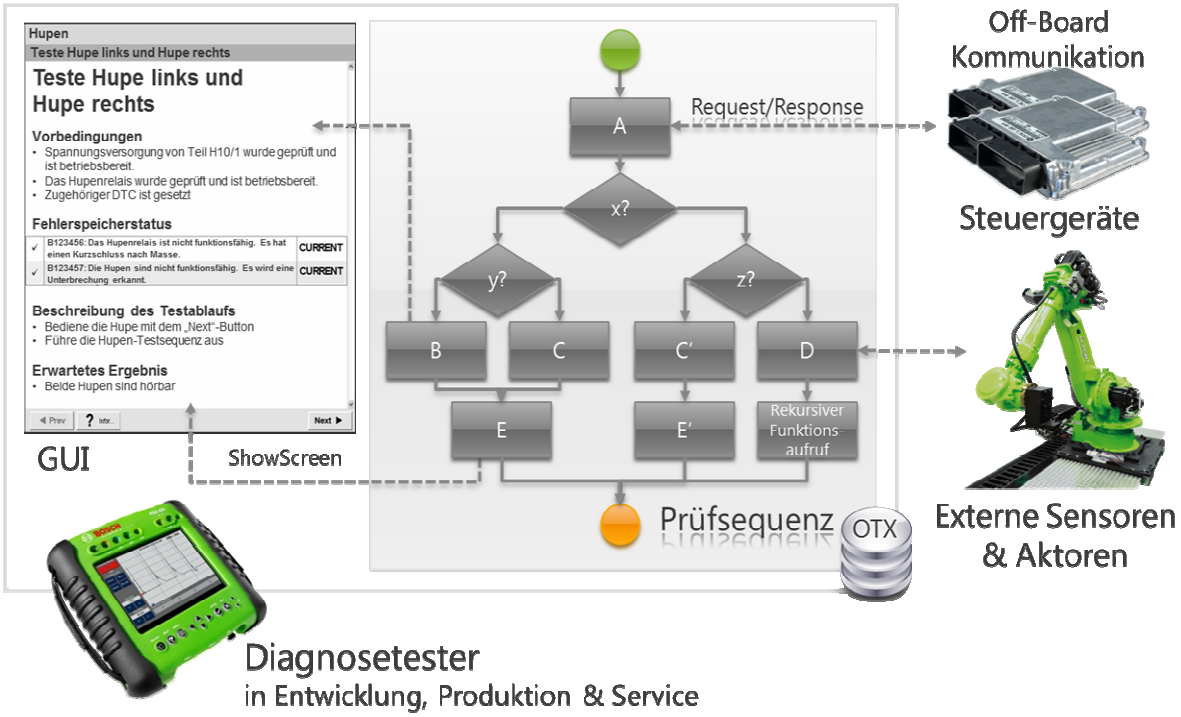

Abb. 7: Abstrakter Diagnoseablauf [8]:

<u>Vorteile durch OTX:</u>

- Spezifikation, Implementierung und Ausführung von Diagnoseabläufen.
- Simple und graphische Modellierung von prozesssicheren Diagnoseabläufen.
- Entwicklung von *OTX*-Abläufen von Fachmann für Fahrzeugdiagnose ohne tiefe Kenntnisse in der Softwareentwicklung.
- Automatische Generierung von Tester-Applikationen aus einem spezifizierten *OTX*-Ablauf . Code-Erzeugung nicht mehr manuell durch einen Entwickler.
- Agile Anpassungen an Diagnoseabläufen ohne viel zusätzlichem Entwicklungsaufwand möglich.

[6] [11]: **SUPKE, J.:** OTX - Hintergrund & Motivation. http://www.emotive.de (28.07.2011)[21]:

- Wissen über verifizierte, praxiserprobte und effektive Diagnoseabläufe können prozesssicher abgelegt werden und wieder verwendet werden.
- Verknüpfung von sämtlichen Diagnoseschritten mit Diagnosedaten, sowie Fehlerbeschreibungen und Ersatzteildatenbanken von Fahrzeugherstellern.
- Vollständige Funktionstests durch Benutzung verschiedener Diagnosedienste.
- Benutzung in allen Bereichen, sowohl der Entwicklung, der Produktion als auch des Services.

2.6.3. OTX Core - Basisbibliothek

Das Datenmodell von *OTX* wird im Wesentlichen durch die *OTX Core Basisbibiothek* beschrieben. In Form einer *XML* Schema-Definition (*XSD*) spezifiziert der *OTX Core* die Struktur von *OTX*-Dokumenten, sowie die für die allgemeine Ablauflogik zur Verfügung stehenden Elemente.
Die meisten der Hauptelemente sind die im klassischen Programmierparadigma wohlbekannten und bewährten Kontrollstrukturen, Deklarationen, Fehler- und Ereignisbehandlung, etc… und unterscheiden sich nicht maßgeblich von anderen Workflow-Beschreibungssprachen oder allgemein von anderen Programmiersprachen. Diese werden hier daher nur erwähnt und wenn nötig kurz beschrieben, um einen groben Eindruck über die Mächtigkeit – d.h. Möglichkeiten, aber auch Grenzen! - von *OTX* zu vermitteln.

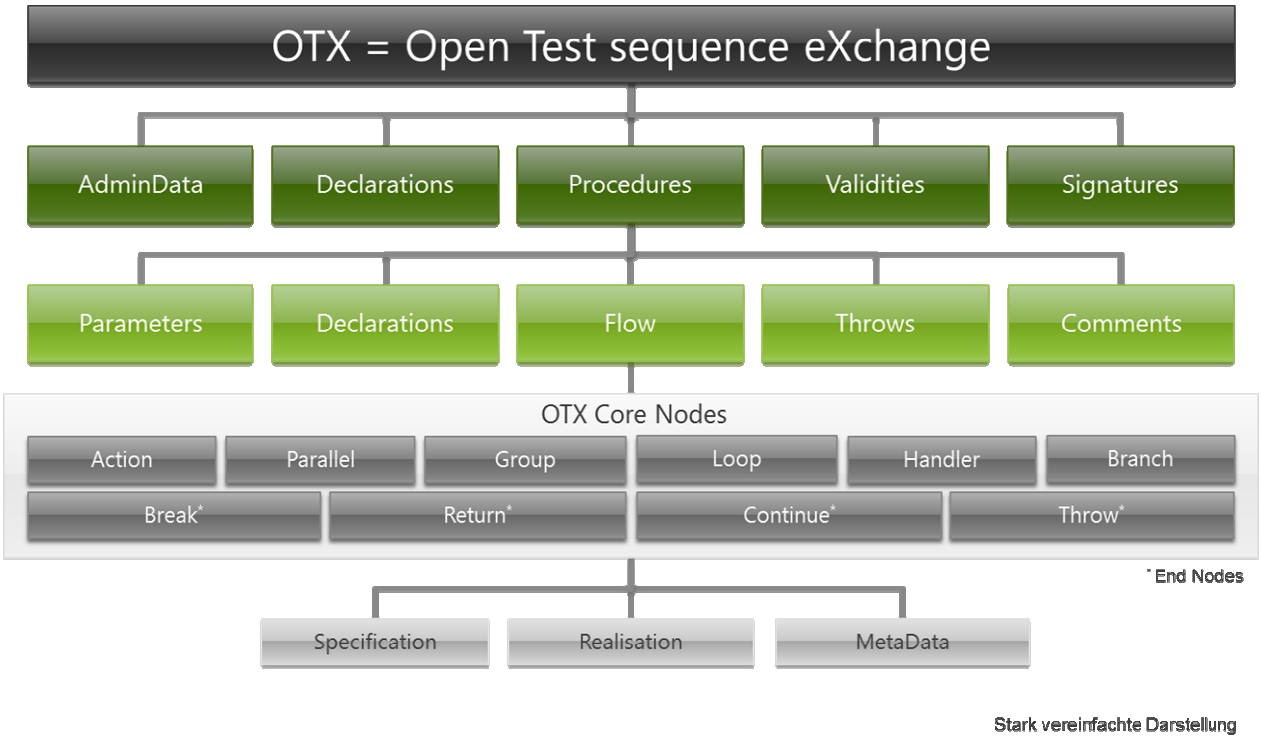

Abb. 8: Aufbau eines OTX-Dokumentes [8]:

Die Hauptelemente sind in ***Abb. 8: Aufbau eines OTX-Dokumentes*** zu sehen. Grob gesagt, besteht ein *OTX*-Dokument aus einer oder mehreren Prozeduren, die jedes einen Ablauf beschreiben. Der

Ablauf wird durch einen *Flow* repräsentiert, der aus einem oder mehreren Knoten besteht, die je eine atomare Aktivität oder zusammengesetzte Aktivitäten darstellen. Im Folgenden werden einige Erläuterungen gegeben, die erwähnenswert sind.
Ein **OTX-Dokument** besteht unter Anderem aus:

- **Deklarationen:** Konstanten, globale Variablen, *Kontext-Variablen*,…
- beliebig vielen **Prozeduren**, die intern oder von extern aufgerufen werden können.
- **Validities und Signaturen:** siehe ***Kapitel 2.6.5 Basiskonzepte***

Spezifizierte **Prozeduren** eines *OTX*-Dokumentes bestehen unter Anderem aus:

- **Parametern:** Übergabe- und Rückgabe- Parameter. Es wird zwischen drei Parameter-Typen unterschieden:
    - **InParameter**: Diese sind Übergabe-Parameter und werden wie Konstanten behandelt - können also nicht modifiziert werden.
    - **OutParameter**: Diese sind nur Rückgabe-Parameter. Es wird der Prozedur kein Wert übergeben.
    - **InOutParameter**: Es werden hier Referenzen übergeben, d.h. Werte-Zuweisungen werden direkt auf den Variablen des Aufrufers vollzogen.
- **Deklarationen:** Konstanten, lokale Variablen.
- einem **Flow**, der den Ablauf von Aktivitäten beschreibt.
- einem **Throws**, welches unbehandelte Fehlerausnahmen (*Exceptions*) weitergibt.

Der **Flow** einer Prozedur ist eine Sequenz von *OTX*-Knoten. Diese können Instanzen von entweder *Atomic*-Nodes oder *Compound*-Nodes sein. *Atomic*-Nodes sind einzelne Aktionen, während *Compound*-Nodes wiederum einen *Flow* beinhalten und somit beliebig tief verschachtelt Sequenzen von *OTX*-Knoten beherbergen können.

Atomic-Nodes:

- **Action Node:** Elemente die bestimmte Aufgaben/Aktionen durchführen. z.B.: *Assignment*, *ProcedureCall*, *ExecuteDiagService*, *MessageDialog*
- **Return, Continue, Break** haben die übliche allgemein bekannte Semantik.
- **Throw:** Eine explizite Ausnahme wird ausgelöst und die Fehlerbehandlung (*Handler*) wird ausgeführt.

Compound-Nodes:

- **Group:** Sequenzen von *OTX*-Knoten werden zur Übersicht und logischen Strukturierung zu einer *Group* zusammengefasst.
- **Loop:** Es gibt *ForLoop, ForeachLoop, WhileLoop* und *DoWhileLoop*, deren Semantik selbsterklärend sein sollte.

- **Parallel:** Macht es möglich Sequenzen parallel auszuführen.
- **Branch:** Repräsentiert die bekannte *If-Then-Else* Kontrollstruktur.
- **Handler:** Stellt einen *Try-Catch* Block für die Fehlerbehandlung dar.

2.6.4. Datentypen

OTX verfolgt ein streng typisiertes Programmierparadigma. Variablen, Konstanten und Parameter müssen zur *Compile-Zeit* statisch als bestimmte Datentypen deklariert sein. Der *OTX*-Standard spezifiziert dazu eine Reihe von *Basis-Datentypen*. Weitere benutzerdefinierte Datentypen können als komplexe Datentypen durch Erweiterungsbibliotheken zur Verfügung gestellt werden. Die *Basis-Datentypen* sind in ***Abb. 9: Datentypen von OTX*** zu sehen.

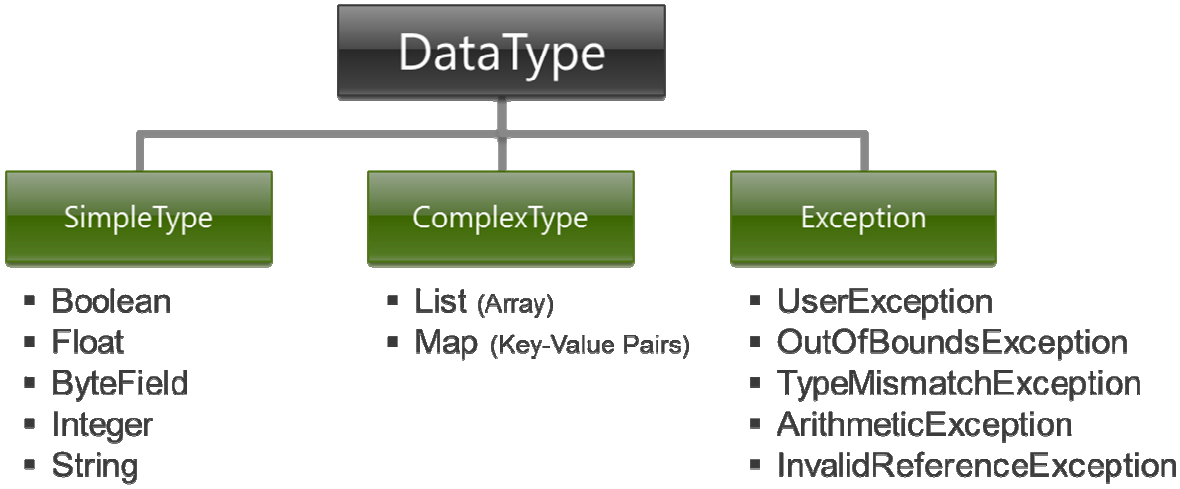

Abb. 9: Datentypen von OTX [8]:

2.6.5. Basiskonzepte

Es gibt in *OTX* einige Basiskonzepte, die *OTX* von herkömmlichen Workflow-Beschreibungssprachen unterscheidet und aufgrund von langjährigen Erfahrungen in der Fahrzeugdiagnose als domänen-spezifische Aspekte in *OTX* eingeführt wurden.

Specification / Realisation - Konzept:

OTX beschreibt die Entwicklung von Diagnoseabläufen als einen drei-stufigen Prozess. In der *Spezifikations-Phase* wird ein Ablauf grob modelliert, ohne dass implementierungs-technische Details festgelegt werden. Funktion und Inhalte können allein aus Namen oder aus einer Prosa-Beschreibung abgeleitet werden. In der *Realisierungs-Phase* beginnt man die Implementierung des Ablaufs. In dieser Zwischenstufe gibt es Elemente mit Realisierung (Implementierung) und auch solche, die ohne Realisierung nur spezifiziert sind.

Nach Beendigung der *Realisierungs-Phase* ist der Diagnoseablauf vollständig realisiert und kann – anders als in den vorangehenden Phasen, ohne Simulation - ausgeführt werden.

Kontext - Konzept:

Das Ziel von *OTX* ist unter Anderem möglichst generische Abläufe modellieren zu können, die „intelligent“ genug sind, um auf verschiedene Fahrzeug–Modelle, Fahrzeug-Typen oder sonstige Varianten reagieren zu können. Durch das Kontext-Konzept können dazu spezielle sogenannte *Kontext-Variablen* deklariert werden, die vom Laufzeitsystem erkannt werden, und mit den nötigen Informationen von der darüberliegenden Diagnoseanwendung gemappt werden. Diese Daten sind typischerweise Fahrzeugdaten, Benutzerdaten oder Umgebungsdaten, wie Fahrzeugmodell, Verkäufer, Identifikationsnummer, Motorisierung, Sonderausstattungsdaten, Benutzername, Benutzerrechte, Betriebssystemversion, verwendetes *VCI*, etc…[7]

Validity – Konzept:

Das *Validity*-Konzept vervollständigt das Kontext-Konzept insofern, dass Teile des *OTX*-Ablaufs ausgeblendet werden können. Dazu kann bestimmten *OTX*-Elementen ein *Validity*-Term zugeordnet werden, der als ein *Boolean*-Ausdruck entscheidet, ob dieses Element ausgeführt wird oder nicht. Vernünftigerweise setzt sich dieser Ausdruck unter Anderem auch vor Allem aus *Kontext-Variablen* zusammen.

Signature – Konzept:

Durch das *Signature*-Konzept lässt sich in einem `OTX`-Ablauf für einen Prozeduraufruf eine `Signature` spezifizieren. Diese `Signature` ist wie eine `Procedure` ohne Implementierung – d.h. es werden nur In- und Out- Parameter spezifiziert. Zur Laufzeit muss es eine oder mehrere `Procedures` geben, die diese `Signature` implementieren. Das Laufzeitsystem entscheidet dann, anhand eines *ValidFor-Terms*, welche `Procedure` aufgerufen wird.

2.6.6. Erweitungs-Bibliotheken

Neben dem zentralen Kern, dem *OTX-Core*, gibt es einige im Standard spezifizierte Erweiterungs-Bibliotheken die den *OTX-Core* durch spezifische Funktionen erweitern.

[7] [22]: **SUPKE, J.:** OTX – Basiskonzepte. http://www.emotive.de (28.07.2011)

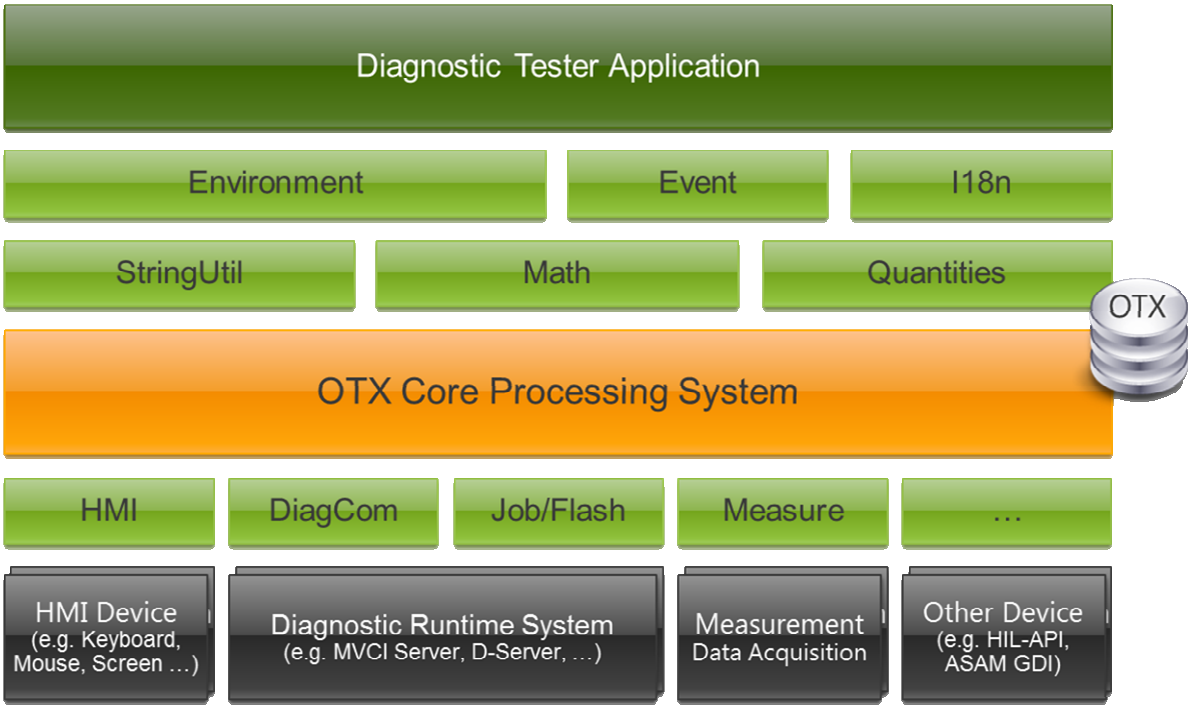

Abb. 10: Aufbau von OTX [8]:

In ***Abb. 10: Aufbau von OTX*** werden diese Bibliotheken dargestellt, die im Folgenden nur kurz beschrieben werden sollen.

- **Environment:** Diese Bibliothek ist für die Kommunikation mit der Umgebung, dem Betriebssystem oder anderen Anwendungen verantwortlich.
- **Event:** Diese Bibliothek spezifiziert alles rund um die Ereignisbehandlung von sowohl internen (z.B. `TimerEvent`) als auch externen Ereignissen (z.B. Mausklick).
- **I18n:** Auch die *Internalization* genannt, sorgt diese Bibliothek für die Übersetzung von allerlei Zeichen und Werten zur Anpassung an regionale Bedürfnisse.
- **StringUtil:** Bietet nützliche Hilfsfunktionen zur Verarbeitung von Zeichenketten an.
- **Math:** Erweitert den Core um weitere mathematische Hilfsfunktionen (z.B. *log, sin, exp*).
- **Quantities:** Wird für Berechnungen mit verschiedenen Einheiten benutzt, um regionale Unterschiede transparent zu machen.
- **HMI:** Stellt *UI*-Elemente zur Verfügung, die eine Interaktion mit einem menschlichen Akteur ermöglichen (Ein- und Ausgabe, Standarddialoge, etc…).
- **DiagCom:** Beinhaltet Elemente, die eine Schnittstelle zur *Offboard*-Kommunikation mit dem Fahrzeug aufbauen. In der Regel ist es, bei dem heutigem Stand der Technik, eine Schnittstelle zu einem *MVCI*-System.
- **Flash:** Ermöglicht durch spezielle Befehle eine autorisierte Programmierung von Steuergeräten.
- **Measure:** Bietet eine Schnittstelle für die externe Messtechnik.

2.7. Open Diagnostic Framework

Das *Open Diagnostic Framework* ist eine von der Firma *emotive GmbH* entwickelte Entwicklungsumgebung, mit der sich Diagnoseabläufe auf Basis von *OTX* spezifizieren, realisieren, validieren, dokumentieren, debuggen, testen und ausführen lassen. In erster Linie ist *ODF* also eine Implementierung des *OTX*-Standards und stellt mit *ODX* und einem Diagnoselaufzeitsystem (z.B. ein *ASAM MVCI-System*) eine komplette, prozesssichere Lösung für die ganze Prozesskette der Fahrzeugdiagnose dar.

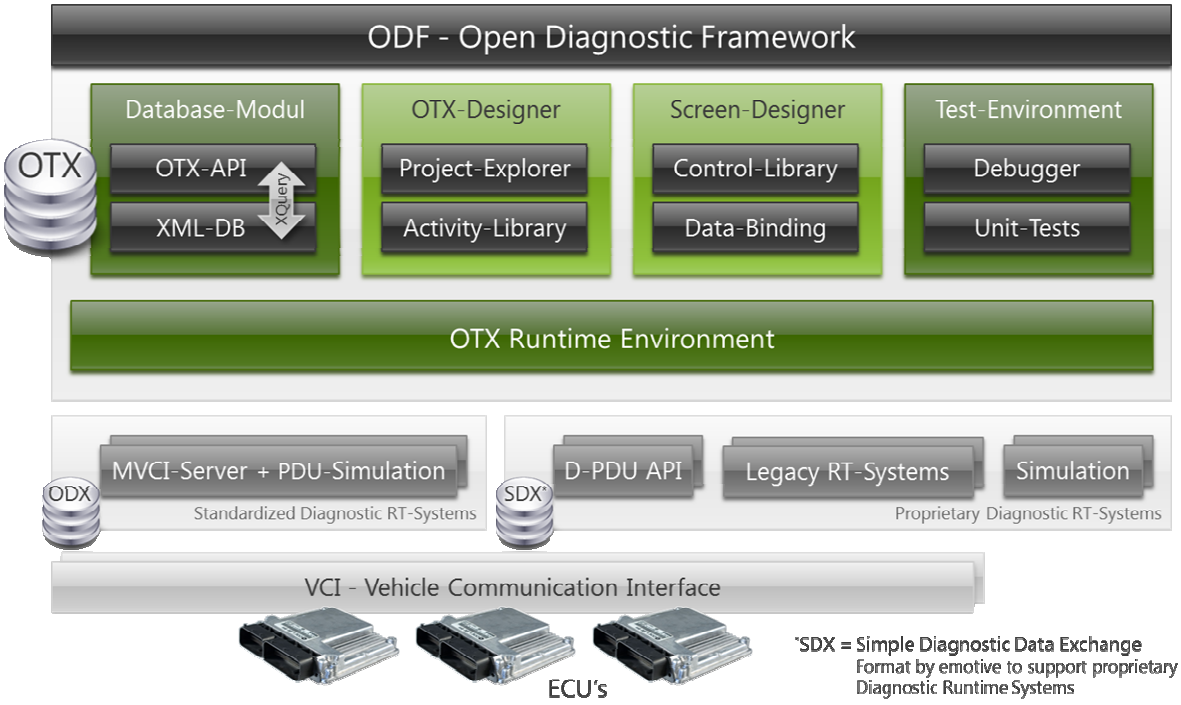

Abb. 11: Aufbau des ODF's [8]:

In ***Abb. 11: Aufbau des ODF's*** ist der Aufbau des *Open Diagnostic Frameworks* zu sehen. Von Anfang an wurde bei der Entwicklung wert darauf gelegt, dass das Framework modular aufgebaut ist, um einzelne Komponenten des Frameworks auch außerhalb von *ODF* in anderen Anwendungssystemen verwenden zu können. Wie man in der Abbildung sehen kann, setzt das Framework auf einem bestehenden Diagnose-Laufzeitsystem auf. Das Framework sorgt für eine Integration und Kommunikation mit der *Hersteller-API* verschiedener Diagnose-Laufzeitsystemen. Darüberhinaus gibt es fünf Komponenten: das *Database-Modul*, der *OTX-Designer*, der *Screen-Designer* und das *Test-Environment*.

Der **OTX-Designer** stellt das zentrale Werkzeug dar, um *OTX*-Abläufe bzw. *OTX*-Dokumente zu modellieren. Er besteht aus:

- einem **Workflow-Designer**, in dem ein Ablauf graphisch als ein Flussdiagramm dargestellt wird und dessen Eigenschaften und Daten durch Popup-Fenster editiert werden können,

- einem **Solution-Explorer**, der für die Navigation durch das aktuelle Projekt, sämtliche Elemente von der *Projekt-Root* bis zur *Procedure*, bis hin zu *Action-Nodes*, *Declarations*, *Parameters*, etc... bereithält,
- einer **Toolbox**, die sämtliche *OTX*-Elemente mit Symbol und Text darstellt, welche per *Drag&Drop* zu einem Diagnose-Ablauf im *Workflow-Designer* hinzufügen kann,
- und aus einer **Ausgabe**, die verschiedene Ausgabefenster zum Logging, Tracing und dergleichen anzeigt.

Die *OTX*-Daten, die durch den *Workflow-Designer* angezeigt und editiert werden können, werden über das *Database-Modul* sowohl geladen als auch persistent gemacht.

Das **Database-Modul** kann als Manager für die *OTX*-Daten angesehen werden. Dieser stellt die sogenannte *OTX-API* zur Verfügung und sorgt für ein performantes Schreiben und einen performanten Zugriff auf die zugrunde liegenden Daten in *XML*. Vor allem da *OTX*-Dokumente bzw. *OTX*-Projekte sehr groß werden können und dieser Speicheraufwand für den Arbeitsspeicher nicht realisierbar ist, bedient sich das *Database-Modul* einer *XML*-Datenbank eines Drittherstellers und lädt nur benötigte Daten in den Arbeitsspeicher.

Durch den **Screen-Designer** ist es möglich, sogenannte *Screens* (einfache *User-Interfaces*) – ähnlich wie beim *Forms-Designer* in Visual Studio - zu erzeugen und anzupassen. Zu diesem Zweck gibt es eine erweiterbare *Control-Library*, die vorgefertigte Steuer-Elemente anbietet. Die erstellten *Screens* dienen zur Ein- und Ausgabe für den Benutzer während der Laufzeit eines Diagnose-Ablaufs. Dazu findet ein *Data-Binding* zwischen Variablen des *OTX*-Ablaufs und entsprechenden Elementen der *Screens* statt. Durch den *Screen-Designer* bzw. durch die entwickelten Screens realisiert *ODF* die *HMI (Human Machine Interface)* Schnittstelle zwischen dem Anwender und dem Diagnoseablauf.

Die **OTX Runtime Environment** ist die Laufzeitumgebung für *OTX*-Abläufe. Nachdem ein *OTX*-Ablauf vollständig spezifiziert wurde und validiert wurde, kann er in der *ODF*-Entwicklungs-Umgebung oder - falls das Binär-Format schon vorliegt – mit Hilfe der stand-alone *OTX*-Runtime Bibliothek ausgeführt werden. Um einen *OTX*-Ablauf auszuführen, werden aus den *OTX*-Daten aus der Datenbank Programmcode (aktuell C#) *On-The-Fly* erzeugt, anschließend wird dieser in Binär-Format übersetzt und kann letztendlich vom Betriebssystem ausgeführt werden. ***Abb. 12: Ablauf der OTX Runtime*** zeigt die besprochene Prozesskette für die Ausführung eines *OTX*-Ablaufs, der als *OTX*-Format vorliegt.

Abb. 12: Ablauf der OTX Runtime [8]:

Die *OTX-Runtime* sorgt, während der Laufzeit eines Ablaufs, außerdem auch für die Kommunikation mit den darunterliegenden Diagnoselaufzeitsystemen (z.B. ein *MVCI*-System). Desweiteren verwaltet sie diverse Prozeduraufrufe an andere *OTX*-Abläufe samt definierten Sichtbarkeiten und Zugriffsrechten.

Das **Test-Environment** stellt Tools zur Verfügung, die die Fehlersuche und das Testen in der Entwicklung von *OTX*-Abläufen unterstützen. Es umfasst zwei Komponenten:

- einen Debugger, mit Hilfe dessen man noch während der Modellierung die Fehlersuche durchführen kann,
- und ein *Unit-Test Framework*, welches fertig gestellte Abläufe testen soll und vor allem die Softwarequalität gewährleisten und erhalten soll (Qualitätssicherung) - das genannte Framework soll Ergebnis dieser Arbeit sein.

3. Modellierung

Diese Kapitel widmet sich der Konzeption und dem Entwurf des Test Frameworks. Wir beginnen mit einer Liste der Anforderungen an ein Test Framework.

3.1. Anforderungen

- Test eines *OTX* Workflows als isolierte Unit, unabhängig von Änderungen an externen Service-Partnern.
- Test von *OTX* Workflows mit u.U. für den Ablauf nicht relevanten oder keinen Eingabe- und Ausgabe- Daten.
- Das Testen eines *OTX* Workflows soll keine Änderungen bzw. Zusätze zur Workflow-Quelldatei nach sich ziehen lassen.
- Erstellung von Testfällen und automatisierte Ausführung einer Menge von Testfällen zur Wiederholbarkeit von Tests und der Qualitätssicherung des Workflows.
- Testfälle müssen voneinander unabhängig prüfbar sein.
- Verwaltung von unter Umständen einer sehr großen Zahl von Testfällen (~10.000).
- Einfache, graphisch unterstützte Spezifizierung von Testfällen durch integrierte Modellierung am Workflow selbst - d.h. im *OTX-Editor* integriert.
- Synchronisierung der Testdaten eines Testfalls mit einem *OTX*-Ablauf, der nach der Erstellung des Testfalls modifiziert wurde.
- Geeignete Darstellung der Testergebnisse
    - Abb. 12.1: Bei Tests mit unter Umständen großen Mengen an Testfällen soll Misserfolg/Erfolg auf einen Blick erkennbar sein.
    - Abb. 12.2: **Optional:** Lokalisierung des Fehlers und Verlinkung zum Fehlerort sowie Fehlerbeschreibung.
- **Optional:** Darstellung von *Code Coverage* Metriken.
- Das Framework wird in C# unter der *.NET*-Umgebung mit der Version 3.5 implementiert.
- Der für die Ausführung der Testfälle bzw. Workflows erzeugter Programmcode sollte ebenfalls kompatibler C#-Code der *.Net*-Version 3.5 sein. Später soll es möglich sein, generisch auch andere Sprachen wie *J#, Visual Basic*, etc. zu unterstützen.

3.2. Use Cases

Die Funktionen, die das Test-Framework bereithalten soll, werden in diesem Kapitel durch ein *Use Case-Diagramm* und Prosa-Beschreibungen spezifiziert. ***Abb. 13: Use Case-Diagramm*** bildet alle *Use-Cases* graphisch ab. Die daran anschließende Tabelle erläutert die einzelnen Use-Cases.

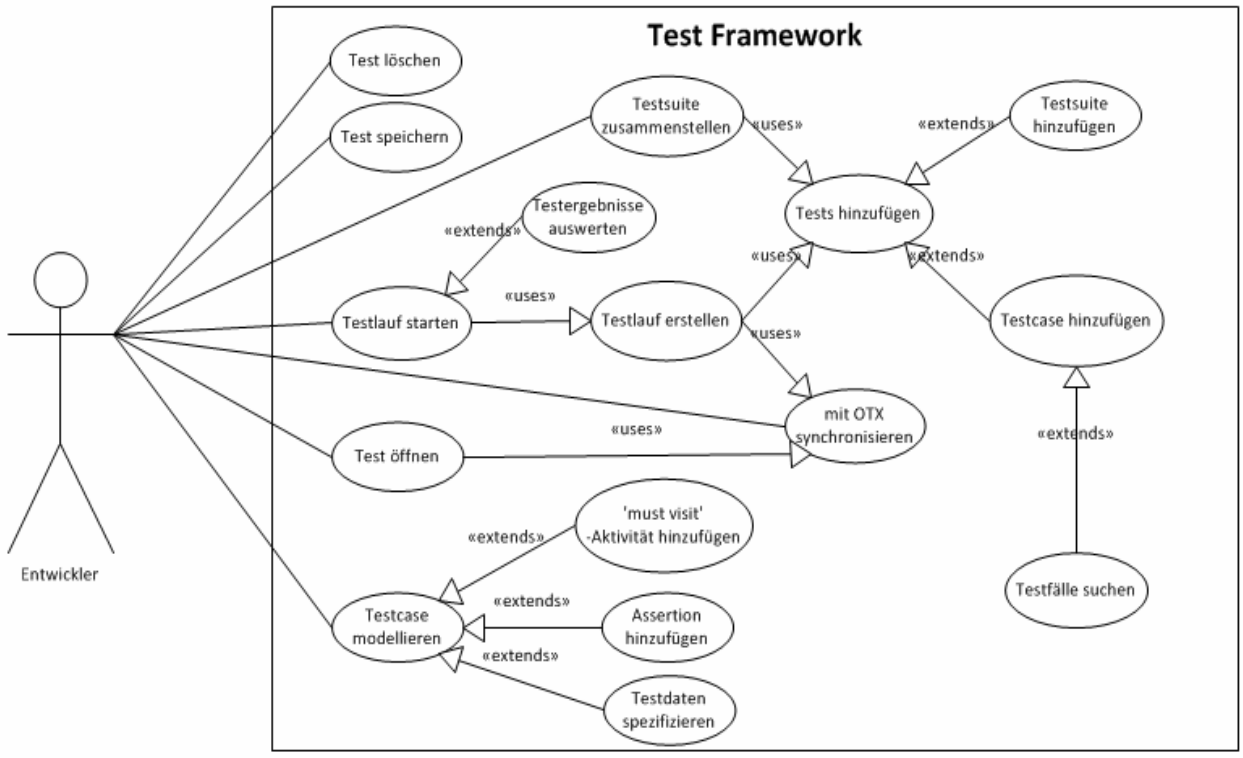

Abb. 13: Use Case-Diagramm

Anwendungsfall	Beschreibung
Test öffnen	Dieser Anwendungsfall soll eine Generalisierung für das Öffnen von *Testcases, Testsuites* und Testruns beschreiben. • **Testcase:** Nachdem ein *OTX*-Ablauf im *OTX*-Designer geöffnet wurde, kann ein *Testcase* aus der Menge aller dem Ablauf zugehörigen *Testcases* zur Bearbeitung geöffnet werden. Die Anreicherung von Testdaten durch den *OTX-Designer* geschieht am aktuell geöffneten *Testcase*. Beim Öffnen von *Testcases* muss außerdem eine Synchronisierung mit den zugehörigen Daten des *OTX*-Ablaufs erfolgen. • **Testsuite:** Zur Verwaltung und zum Editieren von *Testsuites* kann der *Testsuite Manager* geöffnet werden. • **Testrun:** Ein vorher durchgeführter *Testrun* soll auch später wieder geöffnet werden können, um dessen Testergebnisse abzurufen oder den Testlauf wiederholt auszuführen.
Test löschen	Dieser Anwendungsfall soll eine Generalisierung für das Löschen von *Testcases, Testsuites* und *Testruns* beschreiben. Beim Löschen von *Testcases* und *Testsuites* muss beachtet werden, dass vorhandene Referenzen angepasst werden müssen.

Test speichern	Dieser Anwendungsfall soll eine Generalisierung für das Speichern von *Testcases, Testsuites* und *Testruns* beschreiben. Das Speichern geschieht in einem für das Test Framework speziell spezifizierten *XML*-Derivat. Jeder *Testcase* bzw. *Testsuite* oder *Testrun* wird separat in einer einzelnen Datei in einem dedizierten Verzeichnis des Dateisystems vom Betriebssystem gespeichert.
Testcase modellieren	Die Modellierung von Testcases findet integriert im *OTX-Designer* statt. Durch Kontextmenüs soll es möglich sein einer ausgewählten Aktivität *Assertions* hinzuzufügen oder sie als *must visit*-Aktivität zu kennzeichnen. Weitere Testdaten wie z.B. Eingansparameter oder erwartete Ausgabeparameter sollen durch eine weitere Ansicht unterhalb des dargestellten *OTX*-Ablaufs eingegeben werden können.
must visit-Aktivität hinzufügen	Eine Aktivität kann durch ein Kontextmenü im OTX-Designer als *must visit* gekennzeichnet werden. Dies wird für den aktuell geöffneten *Testcase* durchgeführt.
Assertion hinzufügen	Durch das Kontextmenü einer Aktivität kann dem aktuell geöffneten *Testcase* eine *Assertion* hinzugefügt werden.
Testdaten spezifizieren	Durch eine geeignete Ansicht unterhalb des *Workflow-Designers* sollen alle Testdaten angezeigt und editiert, sprich spezifiziert werden können.
Mit Otx synchronisieren	Die Testfallspezifikation kann automatisch mit den Daten des *OTX*-Ablaufs synchronisiert werden. Änderungen am *OTX*-Ablauf müssen am *Testcase* berücksichtigt bzw. aktualisiert werden.
Testsuite zusammenstellen	Mit Hilfe eines *Testsuite Managers* lassen sich *Testsuites* verwalten und zusammenstellen. Dieser soll zwei Ansichten (*Testsuites* und *Testcases*) bieten, mit Hilfe derer *Testcases* einfach per *Drag&Drop* zu *Testsuites* hinzugefügt werden können.
Testlauf erstellen	Um Tests durchzuführen werden *Testsuites* bzw. *Testcases* für einen Testlauf ausgewählt.
Tests hinzufügen	Tests können *Testsuites* oder *Testcases* sein. Diese werden einem Testlauf zur Ausführung oder einem *Testsuite* zur Sammlung und Organisation von Tests hinzugefügt.
Testsuite hinzufügen	Eine *Testsuite* kann einem Testlauf zur Ausführung oder wiederum

	durch eine Referenz einer anderen Testsuite hinzugefügt werden.
Testcase hinzufügen	Ein *Testcase* kann einem Testlauf zur Ausführung oder durch eine Referenz einer *Testsuite* hinzugefügt werden.
Testcases suchen	Der Anwender kann projektweit in verschiedenen Kategorien nach *Testcases* filtern bzw. suchen.
Testlauf starten	Nach der Erstellung bzw. Zusammenstellung eines Testlaufs kann dieser gestartet werden. Die Test-Laufzeitumgebung generiert dazu aus den *OTX*-Daten einen ausführbaren Code und führt alle Tests nacheinander durch.
Testlauf auswerten	Während der Ausführung eines Testlaufs werden die Testergebnisse sofort angezeigt. Nach Beendigung des Testlaufes kann der Anwender durch die Testergebnisse navigieren und auch ein *Test-Report* erstellen lassen.

Tabelle 2: Use-Case Beschreibung

3.3. Analyse und Konzeption eines geeigneten Testing Ansatzes für OTX Workflows

Wie bereits besprochen, genügt ein einfacher *Black-Box* Ansatz mit Vergleich von Ist- und Soll-Ausgabewerten nicht, da die Ausgabegrößen oft nicht aussagekräftig genug sind. Auch ist ein *White-Box* Ansatz zur automatischen Generierung von Testfällen nicht effektiv, wenn nicht zusätzlich durch ein geeignetes Verfahren die Korrektheit des Ablaufs eines Workflows verifiziert werden kann.

Die Prüfung der Interaktionen eines Workflows mit anderen Prozessen durch eine Art Kommunikationsprotokoll, welches der Semantik des Workflows nach spezifiziert wird, deckt den Bereich der Interaktionen bzw. Kommunikation des Workflows ab. Da Workflow-Systeme vor allem im Hinblick darauf erfunden worden sind, Aktivitäten und Unter- oder Neben-Prozesse in einem Ablauf zu integrieren und zu automatisieren; und somit meistens Workflows verwalten, deren Semantik im Wesentlichen aus dem Ablauf seiner Interaktionen mit den Partnern bestehen; ist dieser Ansatz in den meisten Fällen schon ausreichend und gut geeignet um die Korrektheit des Workflows zu gewährleisten - obgleich ein Workflow aus mehr als "nur" seinen Interaktionen besteht. Wir wollen uns dieses Konzept des Testens der Interaktionen eines Workflows im Hinterkopf behalten und es zu einem späteren Zeitpunkt in unsere Konzeption mit einfließen lassen; wenn auch ein anderer Implementierungsansatz gewählt wird als bei MEYER (*BPELUnit Framework*)[8].

[8] Test Framework für BPEL (Siehe [4]: Mayer, 2006)

Doch zunächst wollen wir uns noch einmal dem einfachen *Black-Box* Ansatz zuwenden und dem eine genauere Betrachtung schenken, um das Problem besser zu verstehen und zu sehen welche Lösung es dafür gibt. Das erste Problem besteht darin, dass Eingabeparameter beim Aufrufen eines *OTX* Workflows fehlen oder nicht genügen um daraus Aussagen über die Korrektheit des Workflows zu gewinnen. Dies liegt daran, dass der Ablauf eines Workflows nicht nur von den Eingabeparametern abhängt, sondern auch von dem eingehenden Datenfluss von externen Abläufen. Werden für das *Black-Box* Verfahren als Input nur die Eingabeparameter des Workflows festgelegt, so ist der Testfall unterspezifiziert und Aussagen über das korrekte Verhalten des Workflows natürlicherweise eingeschränkt. Die Lösung liegt nahe: Sämtliche eingehende Daten werden als Input für die *Black-Box* miteinbezogen. Werden sämtliche Eingangsgrößen spezifiziert so ist der Ablauf eines Workflows vollständig bestimmt und determiniert. In ***Abb. 14: Workflow ohne Eingabe-*** und Ausgabeparameter wird ein Workflow in *BPMN*[9] dargestellt, der keine Eingabeparameter besitzt. Erst mit der Einbeziehung der restlichen Eingangsgrößen ist ein Black-Box-Test überhaupt erst möglich.

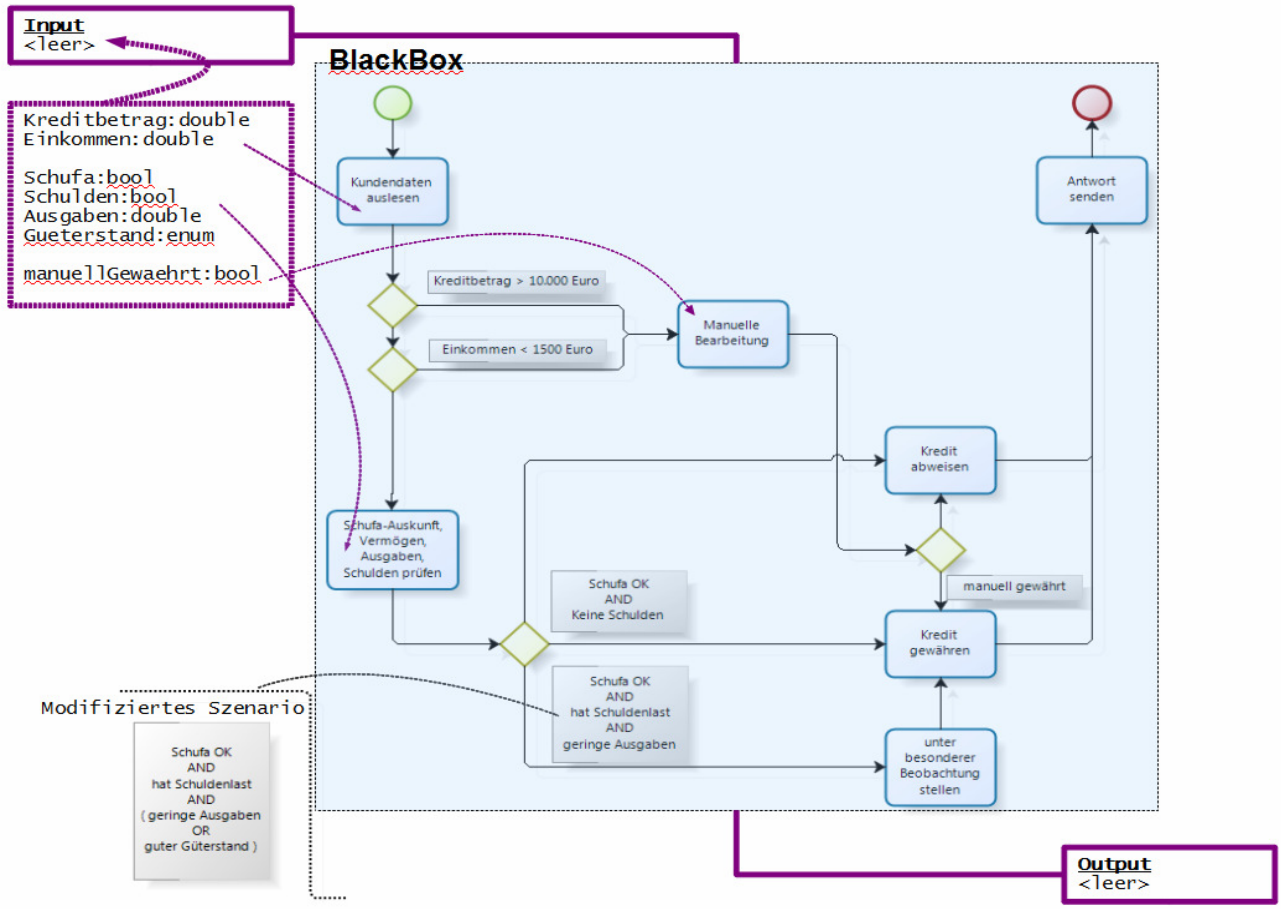

Abb. 14: Workflow ohne Eingabe- und Ausgabeparameter

[9] Business Process Modeling Notation, eine abstrakte Workflow Beschreibungssprache.

Beispielsweise werden von dem Aufruf an den externen Prozess `Kundendaten auslesen` die Rückgabewerte *'Kreditbetrag'* und *'Einkommen'* zurückgeliefert. Von diesen zwei Eingangsgrößen hängt der Ablauf und somit die Semantik des Workflows stark ab. Um die Korrektheit des Workflows durch Vergleichen von *IST-* und *SOLL-Werten* zu prüfen, müssen unter Anderem diese beiden Eingangsgrößen selbstverständlich mit berücksichtigt werden.
Um alle Eingangsgrößen eines Workflows statisch schon vor der Ausführung festzulegen – d.h. um diese mit in die Test-Spezifikation aufzunehmen und zu prüfen, eignet sich das Konzept von *Mock-Objekten* besonders gut.
Im folgenden Abschnitt wird erklärt, wozu *Mock-Objekte* verwendet werden und wie sie für das Test-Framework genutzt werden können.

3.3.1. Simulierung des Inbound Datenstroms - Mock Objekte

Mock-Objekte sind in der Softwaretechnik vor allem im Bereich *Unit-Testing* unverzichtbar. Um das *Unit-Testing* eines Moduls abgekapselt und unabhängig von Änderungen/Fehlern außerhalb eben dieses Moduls zu ermöglichen, muss die Kommunikation mit anderen Modulen emuliert/simuliert werden. *Mock-Objekte* implementieren die Schnittstelle von Modulen vollständig, indem sie jedoch statische Daten verwenden, die vor Ausführung spezifiziert werden müssen. Ein *Mock-Objekt* (engl. to mock = etwas vortäuschen) simuliert also ein aufzurufendes Modul, indem es die für einen Testfall spezifizierten Eingabe- und Rückgabewerte nutzt, um einen Aufruf "vorzutäuschen".
Die Nutzung von *Mock-Objekten* hat folgende Vorteile:

- Keine Abhängigkeiten des *Units* zu anderen Modulen. Fehler in anderen Modulen haben keinen Einfluss auf den *Unit-Test*.
- *Unit-Test* eines Moduls ist möglich, ohne dass erforderliche Module vollständig implementiert oder überhaupt existieren müssen.
- Spezifikation von statischen Rückgabewerten des *Mock-Objektes*, um ein bestimmtes Verhalten des zu testenden Moduls zu prüfen. Speziell auch um sonst schwer auszulösendes Verhalten zu testen (z.B. Ausnahmefehlerbehandlung).
- Da nur statische Daten zurückgegeben werden, wird quasi keine Zeit für den Aufruf gebraucht.

Diese Vorteile sind ebenso für das Testen von Workflows gültig. Das Konzept von *Mock-Objekten* lässt sich für Workflows leicht realisieren. Jeder Aufruf, den ein Workflow an einen externen Dienst sendet, wird quasi vom *Test Framework* abgefangen und es werden statische Rückgabewerte zurückgegeben. Man könnte *Mock-Objekte* auch realisieren, indem ein tatsächlich generiertes *Mock-*Modul, Eingabewerte entgegennimmt und beispielsweise aus einer Tabelle oder Datenbank die entsprechenden Ausgabewerte zurückliefert. Das *Test Framework* würde dann alle Aufrufe zu den

entsprechenden *Mock-Objekten* umleiten. Allerdings hat dies den Nachteil, dass Rückgabewerte nun dynamisch berechnet werden und es somit schwieriger wird ein bestimmtes Verhalten in dem zu testenden Workflow auszulösen (siehe oben: 3. Punkt bei den Vorteilen).
Ein anderer Ansatz würde die Rückgabewerte bei der Spezifikation eines Testfalles mit einbeziehen. Jeder Testfall muss demnach zusätzlich zu den Eingabe- und Ausgabe-Parametern des Workflows auch die Rückgabewerte externer Aufrufe spezifizieren. Die Spezifizierung von Eingabegrößen für externe Aufrufe ist in dem Fall nicht nötig, da die Rückgabewerte für diesen Testfall statisch schon festgelegt sind und nicht von Eingabegrößen abhängig sind - eben darum ist ein gezieltes Triggern von einem gewünschtem Verhalten des Workflows möglich. Besonders Randfälle und Ausnahmefehler (*Exceptions/Faults*) werden auf diese Weise testbar gemacht.
Zusätzlich ist damit die Möglichkeit gewonnen worden, schon vor Ausführung des Workflows durch diese Rückgabewerte sämtlich eingehenden Datenfluss des Workflows festzulegen, um damit den Input beim *Black-Box Testing* zu füllen.

Wie wir einsehen können, lösen *Mock-Objekte* unser beschriebenes Problem mit den unterspezifizierten Eingangsgrößen für ein Workflow im Testgang. Wir wollen als nächstes ein verwandtes Problem betrachten.
Ein Workflow kapselt oft viele Aktivitäten und Prozesse in Teilabläufe bzw. Teilfunktionen zu einem Gesamtablauf. Prüft man in einem *Black-Box* Verfahren den Gesamtablauf, ist unter Umständen. noch nichts über die richtige Verarbeitung in den Teilabläufen ausgesagt. Dies spiegelt sich auch darin wieder, dass die Ausgabeparameter eines Workflows oftmals nicht genügend Informationen liefern, um damit die Korrektheit des Workflows prüfen zu können. In ***Abb. 14: Workflow ohne Eingabe- und Ausgabeparameter*** (siehe oben) sehen wir einen Workflow, der gar keine Ausgabeparameter zurückgibt. Ein einfacher Black-Box-Test könnte somit keine Aussagen über die Korrektheit treffen.
Das Problem in diesem Beispiel liegt darin, dass die eigentliche Funktionalität des Workflows gekapselt wurde. Die relevante Information, nämlich dass der Kredit gewährt oder abgewiesen wurde, wird durch die Aktivität *Antwort senden* schon von dem Workflow selbst verarbeitet und nicht als Ergebnis in den Ausgabeparametern des Workflows zurückgeliefert. Natürlich ließe sich dieses Problem umgehen, indem Workflows prinzipiell immer so modelliert werden, dass Eingabe- und Ausgabe- Schnittstellen genau den Eingabe- und Ausgabemengen der erwünschten Semantik-Funktion entsprechen. Weitere Verarbeitung würde in einem separaten Workflow behandelt. In ***Abb. 15: Workflow, abgekapselte Funktion*** wird die angesprochene Kapselung des Workflows in ***Abb. 14: Workflow ohne Eingabe- und Ausgabeparameter*** aufgebrochen. Die hauptsächliche Funktion, nämlich einen Kredit zu prüfen, wird von der restlichen Verarbeitung abgetrennt und kann so in einem *Black-Box* Verfahren geprüft werden.

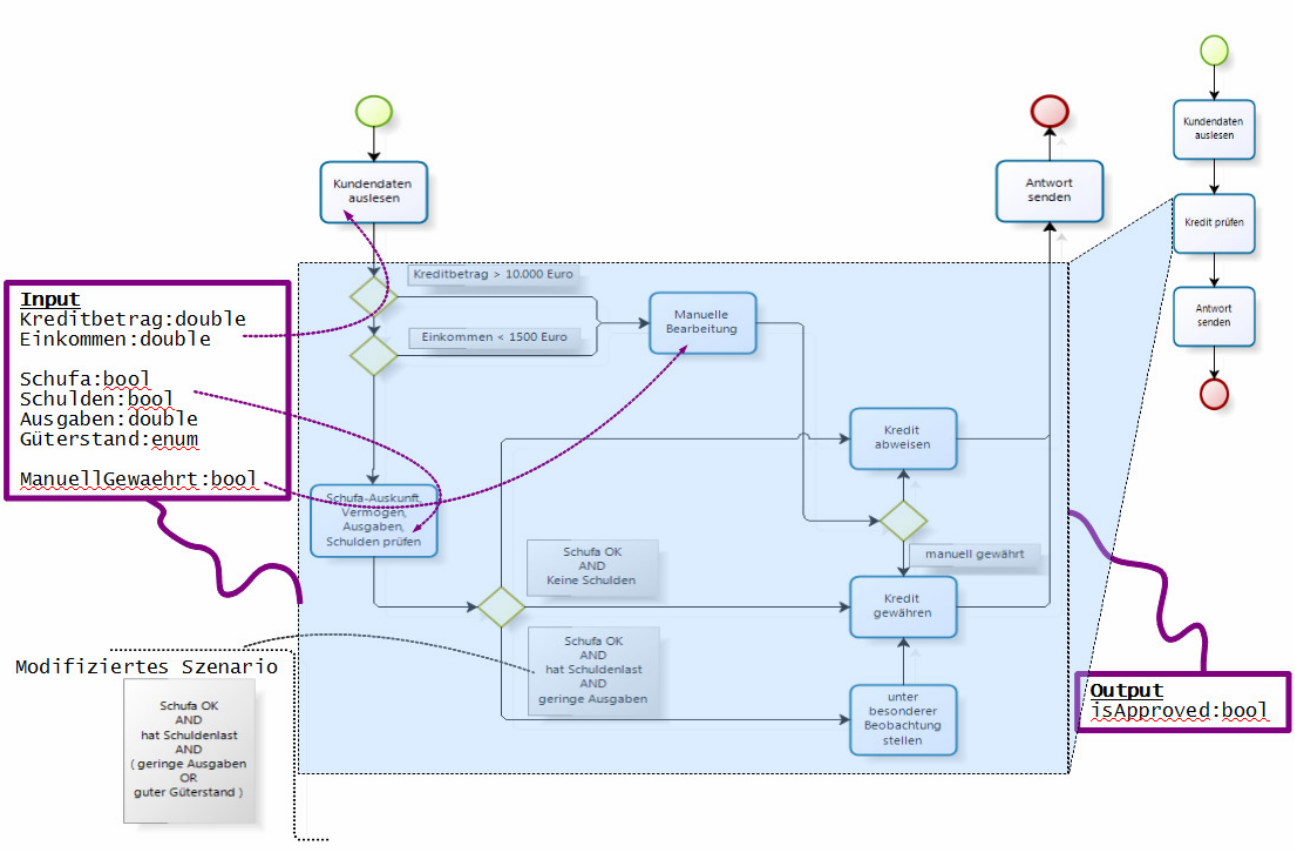

Abb. 15: Workflow, abgekapselte Funktion

Der Gesamtprozess wird wie in der ***Abb. 15: Workflow, abgekapselte Funktion*** zu sehen ist, von zwei verschiedenen Workflows implementiert. Allerdings ist dies nicht immer so erwünscht und ist eine Einschränkung für den Entwickler. Es ist unrealistisch - d.h. praktisch nicht durchsetzbar, dass der Entwickler in Hinsicht darauf modelliert, wie es geeignet wäre um Tests durchzuführen. Es ist nicht die Aufgabe des Entwicklers, sondern die des *Test Frameworks* dafür zu sorgen, dass ein Workflow beliebiger Struktur testbar ist. Gesucht wird also ein Konzept, welches *Unit-Tests* von Workflows ermöglicht, auch ohne dass die Funktionalität eines Workflows durch seine Eingabe- und Ausgabe-Parameter widergespiegelt wird. Da Eingabe- und Ausgabe- Parameter des Workflows also nicht ausreichen und das *Black-Box Testing* seiner Definition nach nur Eingabe- und Ausgabe- Parameter in den Testvorgang einbezieht, können wir im Folgenden nicht mehr von einem reinen *Black-Box Test* sprechen.

Um beispielsweise die Funktionalität der Kreditprüfung in dem besprochenen Workflow zu testen, muss offensichtlich auf die interne Variable *'isApproved'* zugegriffen werden, dessen Wert von den Aktivitäten *Kredit gewähren* bzw. *Kredit abweisen* gesetzt wird. Das Test-Framework muss demnach die interne Struktur und Logik des Workflows kennen und innerhalb des Workflows einen Mechanismus zur Prüfung ansetzen. Wir werden im nächsten Abschnitt sehen, dass so genannte *Assertions innerhalb eines Workflows* diesen Dienst erfüllen.

3.3.2. Assertions innerhalb eines Workflows

Der Begriff *Assertion* (lat./engl. für Aussage; Behauptung) wurde in der Informatik zum ersten Mal von ROBERT FLOYD 1967 in seinem Artikel *Assigning Meanings to Programs* gebraucht. Floyd schreibt in diesem Artikel darüber, wie sich die Korrektheit von Flussdiagrammen durch Zusicherungen (*Assertions*) beweisen lässt. Später erweitert TONY HOARE diesen Ansatz um das sogenannte *Hoare-Kalkül* und legt damit den Grundstein für die formale Verifikation von Software.
Wenn auch für den Beweis der vollkommenen Korrektheit eines Programms viele Zusicherungen notwendig sind, die z.T. Intelligenz und Rafinesse erfordern und somit auch viel Aufwand bedeuten, kann die Korrektheit von Teilfunktionen des Programms recht leicht durch einzelne *Assertions* sichergestellt werden.
Im modernen Softwaretest werden *Assertions* häufig genutzt, um sicherzustellen, dass der Wert eines Datenobjektes mit einem erwarteten Wert übereinstimmt. Beispiel:

```
GOLogic = 2 + 2 ;
assert( GOLogic == 5 );
```

In diesem kurzen Code-Abschnitt wird durch die *assert*-Anweisung zugesichert, dass zu genau diesem Zeitpunkt der Wert der Variable 'GOLogic' dem Wert *'5'* entspricht. Sollte diese Zusicherung bei der Ausführung nicht erfüllt werden, so muss der Test als fehlgeschlagen ausgewertet werden.
Kehren wir nun zurück zu unserem Beispiel der Kreditprüfung. Wie besprochen, benötigt das Test-Framework einen Mechanismus, um auf die innere Struktur und Logik des Workflows zuzugreifen. Durch *Assertions*, die innerhalb des Workflows an Kontrollflüssen gesetzt werden und in der Laufzeit ausgewertet werden, lassen sich beliebige Teilfunktionen eines Workflows testen. In ***Abb. 16: Workflow mit Assertion*** sehen wir wie die Variable `isApproved` durch eine *Assertion* geprüft wird. Die *Assertion* stellt sicher, dass ein Kredit je nach Testfall entweder gewährt oder abgewiesen wird.

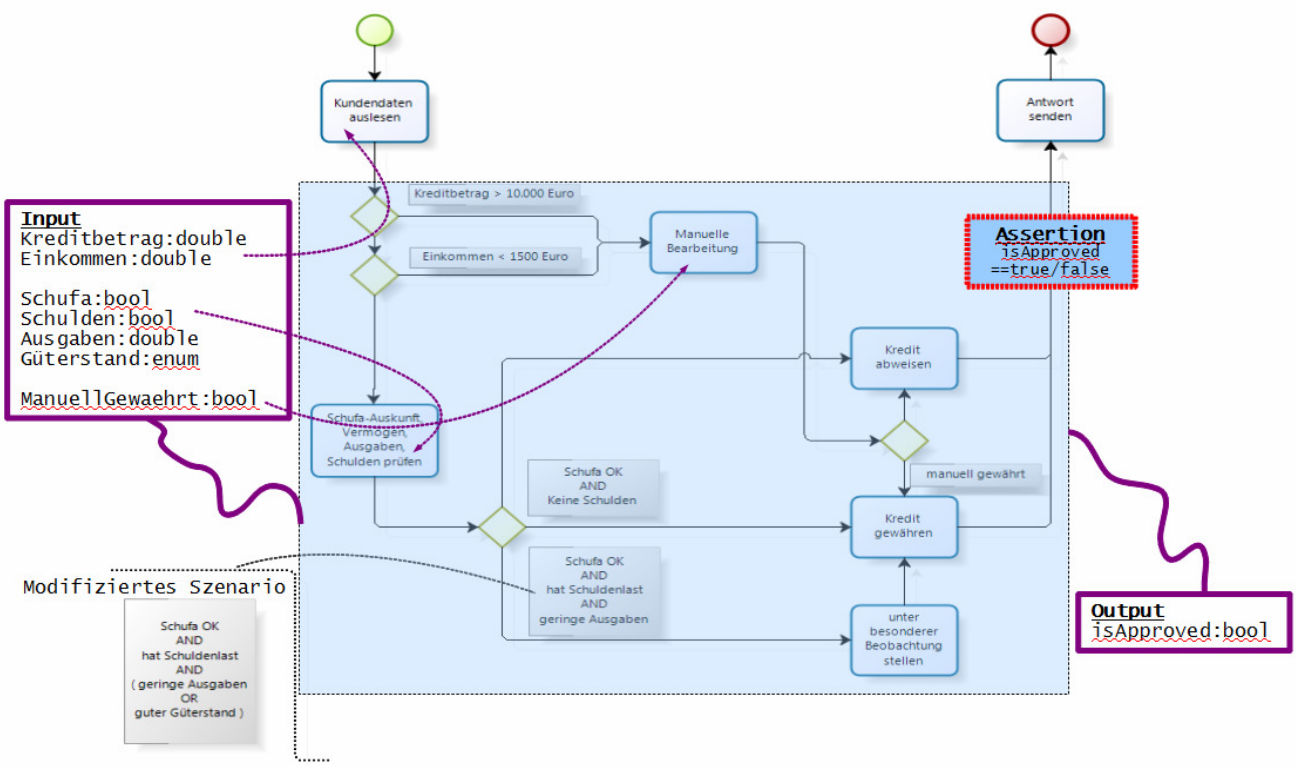

Abb. 16: Workflow mit Assertion

In einem weiteren Beispiel wollen wir mithilfe eines konkreten Testfalls nur die Teilfunktion der *automatischen Kreditprüfung* desselben Workflows sicherstellen. Die Testspezifikation ist in den magentafarben-umrandeten Kästen zu sehen.

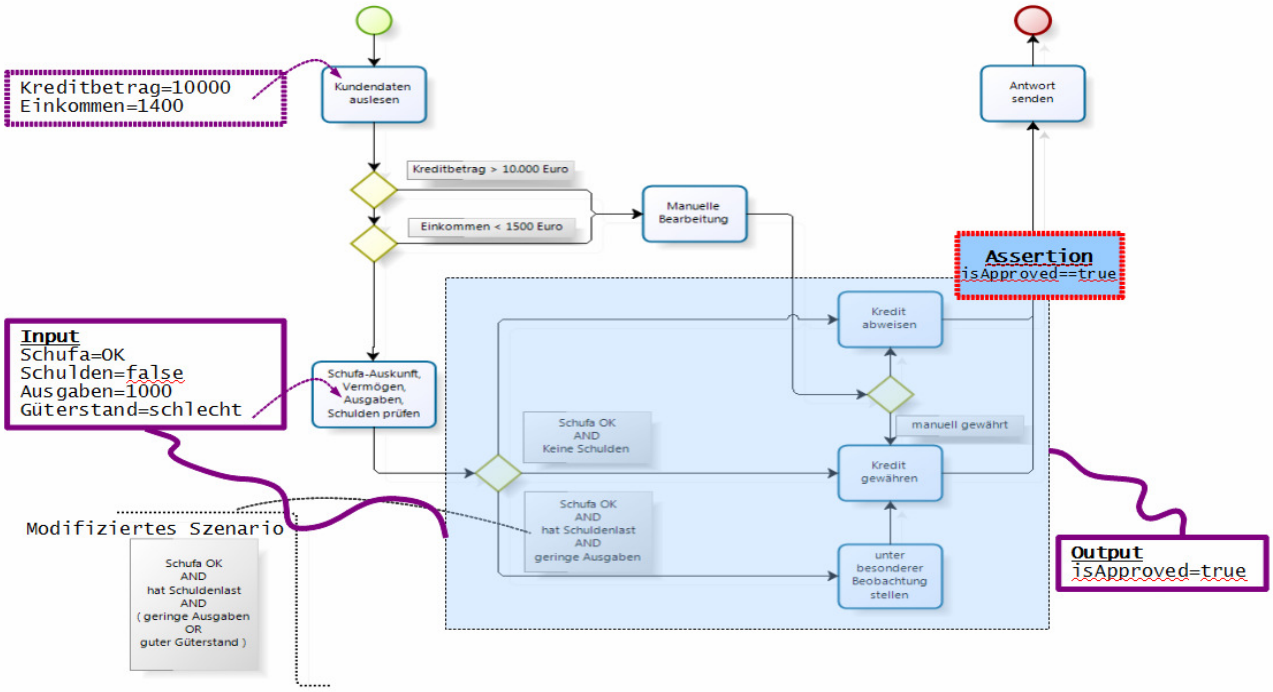

Abb. 17: Workflow, Prüfung einer Teilfunktion

Anhand der Input-Daten lässt sich der Ablauf des Workflows nachverfolgen und erkennen, dass die Aktivität *Kredit gewähren* ausgeführt wird, die die Variable `isApproved` auf *'true'* setzt. Die nachfolgende Assertion stellt sicher, dass zu diesem Zeitpunkt tatsächlich `'isApproved==true'` gilt und der Workflow mit dieser Zusicherung weiter ausgeführt wird. Unter der Annahme, dass der Workflow korrekt implementiert wurde, wird die *Assertion* in diesem Fall positiv ausgewertet und unser Testfall wird als erfolgreich beendet. Der Entwickler kann noch weitere Testfälle spezifizieren und verifizieren lassen, bis alle Testfälle abgedeckt sind und gesichert ist, dass der Workflow die Kreditprüfung korrekt ausführt. In einem Optimierungsprozess soll nun die automatische Kreditprüfung verbessert werden, indem der Güterstand des Kreditnehmers mitberücksichtigt wird. Betrachten wir das modifizierte Szenario, welches die *Condition* eines Pfades (siehe Abbildung oben) um den Status des Güterstandes erweitert. Wir können leicht nachvollziehen, dass der Ablauf unter der gegebenen Spezifikation auch in unserem modifizierten Szenario derselbe ist. Die *Assertion* gilt und der Testfall läuft erfolgreich ab. Nehmen wir nun jedoch an, dass bei der besprochenen *Condition* ein Implementierungsfehler unterlaufen ist und die Klammern in der Condition versehentlich weggelassen wurden. Wir halten fest:
korrekte Implementierung

```
' Schufa OK AND hat Schuldenlast AND ( geringe Ausgaben OR guter Güterstand )'
```

Implementierung mit Fehler:

```
' Schufa OK AND hat Schuldenlast AND   geringe Ausgaben OR guter Güterstand  '
```

Unter der gegebenen Spezifikation würden die *Conditions* der unteren beiden Pfade zu *'false'* ausgewertet und der Pfad zu der Aktivität *Kredit abweisen* würde ausgeführt werden. Die Aktivität *Kredit abbweisen* setzt die Variable `isApproved` auf *'false'* und die nachfolgende *Assertion* wird negativ ausgewertet. Der Testfall wird somit als fehlgeschlagen beendet und deutet damit auf einen Fehler in der Implementierung hin.
Durch diese Testfallspezifikation, die sowohl vor als auch nach dem Optimierungsprozess dieselbe geblieben ist, konnte ein Implementierungsfehler während der Änderung des vorher korrekt ablaufenden Prozesses gefunden werden. Zusammenfassend lässt sich sagen, dass wir durch geeignete Testfallspezifikationen (Inputwerte und *Assertion*) eine Qualitätssicherung der Software, bzw. des Workflows, erreichen können.
Erwähnenswert ist beiläufig noch, dass dieser Test mit einem *Black-Box-Test* von eben nur dieser Teilfunktion vergleichbar ist. Wir sehen die *Black-Box* als blau-schattierten Bereich mit Input- und Output- Parametern in den magentafarben-umrandeten Kästen. Mit Hilfe von internen *Assertions* in Workflows und das Konzept von *Mock-Objekten* lässt sich also ein quasi *Black-Box-Test* von Teilfunktionen des zu testenden Workflows simulieren. Wobei sich die bestehende Tatsache natürlich nicht ändert, dass wir hier pro forma einen *White-Box-Test* auf unseren Workflow durchführen.

Wo dieses Beispiel noch gut überschaubar ist und ein umfassender Test zum Beweis der Korrektheit noch recht einfach durchgeführt werden kann, ist Softwaretest im Allgemeinen eine komplexe Angelegenheit. Durch das Konzept der *Assertions* können an beliebiger Stellt Zusicherungen gesetzt werden, die die Korrektheit von Funktionen des Programms sicherstellen. Allerdings erfordern komplexe Programme oftmals intelligente und raffinierte *Assertions* um die Korrektheit festzustellen. Im ***Abschnitt 5.1: Erstellung von Testfällen - Best Practices*** werden einpaar universelle Anleitungen und Tipps für grundlegende, immer wiederkehrende, Testsituationen gegeben. Letzten Endes bleibt das Testen von Software aber doch komplex und ist eine Profession für sich.

3.3.3. Prüfung des Kontrollflusses durch 'must visit'-Aktivitäten

Wir haben bisher das Konzept betrachtet durch *Assertions* sicherzustellen, dass bestimmte Variablen bzw. Ausgabegrößen bestimmten Soll-Werten entsprechen. In gewisser Weise haben wir uns somit im Bereich des Datenflusses des Workflows bewegt. Fürs Weitere wollen wir betrachten in wie weit sich durch Zusicherungen am Kontrollfluss des Workflows dessen Korrektheit verifizieren lässt.
Das Konzept ist intuitiv leicht verständlich und Testfälle lassen sich einfach und gut modellieren. Zu den zu spezifizierenden Eingabegrößen des Workflows werden zusätzlich noch Aktivitäten (bzw. der Pfad) angegeben, die in dem Testfall ausgeführt werden müssen. Wir wollen hierzu wieder unser Kredit-Prüfung-Szenario hernehmen. In der Abbildung sehen wir die für unseren Testfall spezifizierten eingehenden Input-Daten des Workflows, sowie die Aktivitäten, die für diesen Testfall ausgeführt werden müssen. Die rot-gestrichelten Markierungen im Workflow zeigen die zu besuchenden *must visit*-Aktivitäten sowie den daraus eindeutig resultierenden, auszuführenden Pfad. Sollte bei dem Testdurchlauf des Workflows eines dieser *must visit*-Aktivitäten nicht ausgeführt werden, so muss der Testfall als fehlgeschlagen bewertet werden.

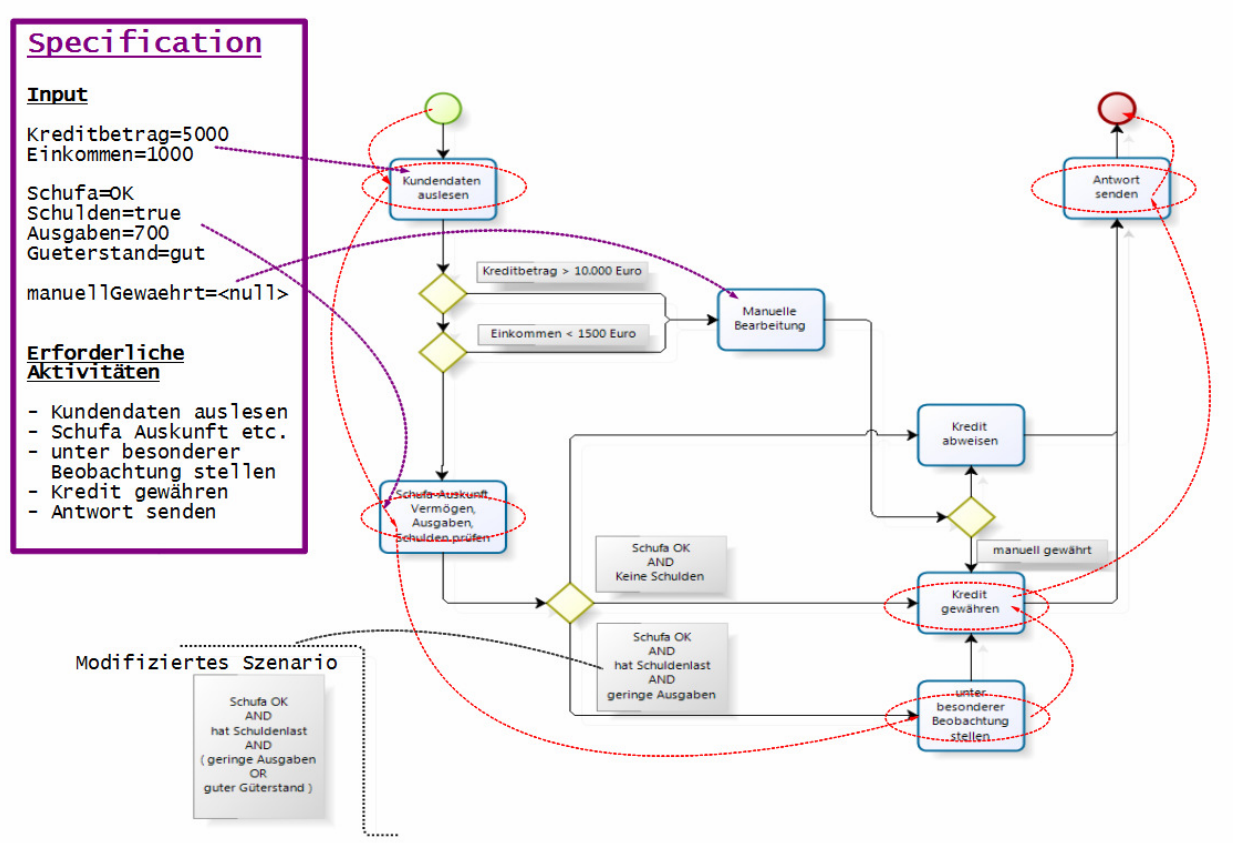

Abb. 18: Workflow mit must visit-Aktivitäten

Die Behauptung ist, dass die Testfallmodellierung mit *must visit*-Aktivitäten in den meisten Fällen genauso mächtig ist wie eine Spezifikation des zu durchlaufenden Pfades. Mit anderen Worten soll es möglich sein durch das Konzept der *must visit*-Aktivitäten, genau den Pfad zu spezifizieren, den ein Workflow durchläuft. Wenn wir uns also mit Pfaden beschäftigen, heißt es, dass wir uns vor allem mit Verzweigungen auseinandersetzen müssen. Pfade unterscheiden sich dadurch, dass sie verschiedene Verzweigungen durchlaufen. Können wir durch *must visit*-Aktivitäten Testfälle so modellieren, dass sie alle Verzweigungen beliebiger Pfade abdecken können, so ist unsere Behauptung erfüllt. Falls jede Verzweigung eine Aktivität ausführt, so ist es klar. Wir können die Spezifizierung aller Kanten (Zweige) eines beliebigen Pfades durch die Spezifizierung einer jeweils in dem Zweig befindlichen *must visit*-Aktivität ersetzen.

Die Frage ist vielmehr, wie es sich mit Kanten verhält, die keine Aktivitäten besitzen. In der Nachstehenden Abbildung sehen wir die Modellierung von vier Testfällen. Im Testfall a) sehen wir links die Spezifikation des auszuführenden Pfades nach der Verzweigung rot markiert. Rechts ist die dazu äquivalente Spezifikation von *must visit*-Aktivitäten. Wir wollen an dieser Stelle festlegen, dass rot markierte Aktivitäten ausgeführt werden müssen und nicht markierte Aktivitäten nicht ausgeführt werden dürfen. Dann können wir sehen, dass wir in den Fällen a), b) und c) die Spezifikation des durchlaufenden Pfades auf eine äquivalente Spezifikation mit *must visit*-Aktivitäten reduzieren können - d.h. es gibt eine eindeutige Zuordnung zwischen dem zu durchlaufenden Pfad und einer Menge von Aktivitäten, die genau denselben Pfad eindeutig spezifizieren. Das Testszenario d) macht mehr Schwierigkeiten. Es gibt keine Möglichkeit durch *must visit*-Aktivitäten zu unterscheiden,

ob die Kante *cond2* oder *cond3* durchlaufen wird. Das Problem liegt darin, dass die Pfade, die jeweils eines dieser Kanten durchlaufen, paarweise keinen Unterschied in der Ausführung von Aktivitäten zueinander zeigen - der Pfad über *cond2* ist bezogen auf die Ausführung von Aktivitäten äquivalent zu einem Pfad über *cond3*. Dieser Workflow macht so gesehen nicht viel Sinn - da die beiden Zweige keinen Unterschied machen und eigentlich als nur ein Zweig mit der Bedingung `'cond2 OR cond3'` implementiert werden könnten. Dennoch ist es vorstellbar, dass solch ein Workflow vorab schon auf diese Weise modelliert wird, weil zukünftig noch Aktivitäten hinzugefügt werden sollen. Um das Testszenario d) zu realisieren, müssen wir das Konzept erweitern und zusätzlich zu der *must visi*-Aktivität `Task 3` noch eine *Assertion* `'assert(false)'` setzen, die den Zweig mit *cond3* ausschließt.

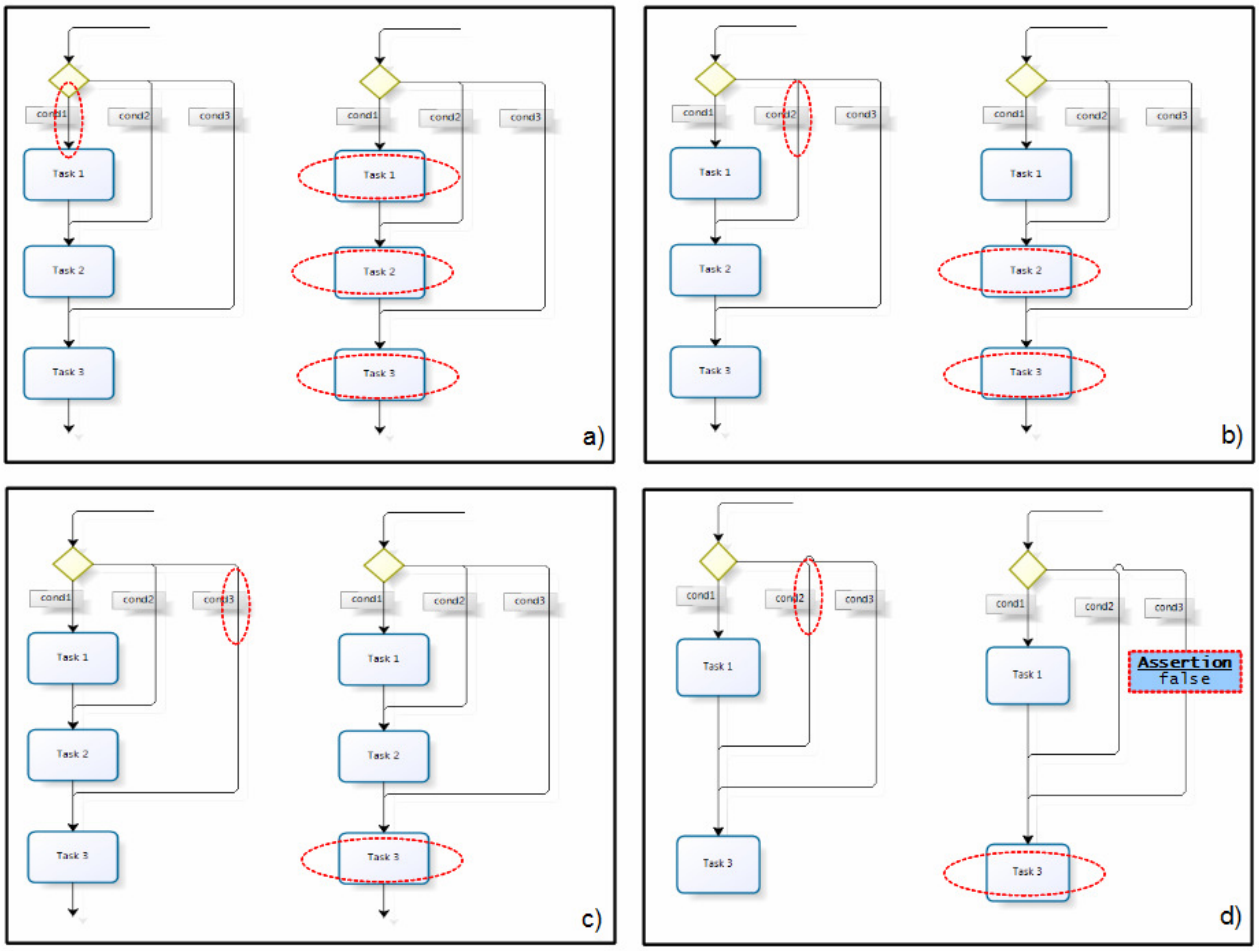

Abb. 19: Steuerung des Kontrollflusses durch must visit-Aktivitäten

Wir sehen also, dass wir durch das Konzept von *must visit*-Aktivitäten die Möglichkeit haben, den Kontrollfluss eines Workflows zu verifizieren. Natürlich stellt sich die Frage wieso der Umweg über Aktivitäten gegangen wird und nicht gleich der Pfad selbst spezifiziert wird. Dies hat implementierungstechnische Gründe und wird später klarer erklärt, wenn es um die Implementierung geht. An dieser Stelle sei nur gesagt, dass es notwendig ist, *Assertions* an eine Aktivität zu binden, da

eine *Assertion* nicht innerhalb eines *OTX*-Ablaufs spezifiziert wird, sondern extern in einer separaten Testfall-Spezifikation – es ist nun einzusehen, dass eine Bindung an ein *OTX*-Element notwendig ist, um später die *Assertion* an der richtigen Position auszuführen.

Wir haben oben gefordert, dass alle nicht markierten Aktivitäten nicht ausgeführt werden dürfen - bzw. dass alle Aktivitäten spezifiziert werden müssen, die auf dem auszuführenden Pfad liegen. Diese Forderung an eine Testfall-Spezifikation stellt mit hoher Zuverlässigkeit das korrekte Verhalten eines Workflows fest, allerdings ist so eine vollständige Spezifikation des Pfades nicht immer erwünscht und vor allem bei sehr großen Workflows praktisch kaum durchsetzbar. Oftmals reicht es für einen Testfall sicherzustellen, dass bestimmte Kern-Aktivitäten ausgeführt werden, um die semantisch korrekte Funktionalität zu gewährleisten. Wir lassen die Forderung, dass nicht markierte Aktivitäten nicht ausgeführt werden dürfen, für das Grundkonzept wieder fallen und bemerken, dass Kanten oder Aktivitäten, die nicht ausgeführt werden sollen, auch explizit durch eine *Assertion* `'assert(false)'` sichergestellt werden können. Durch solche expliziten *Assertions* lassen sich Uneindeutigkeiten, wie sie in der nächsten Abbildung zu sehen sind, vermeiden.

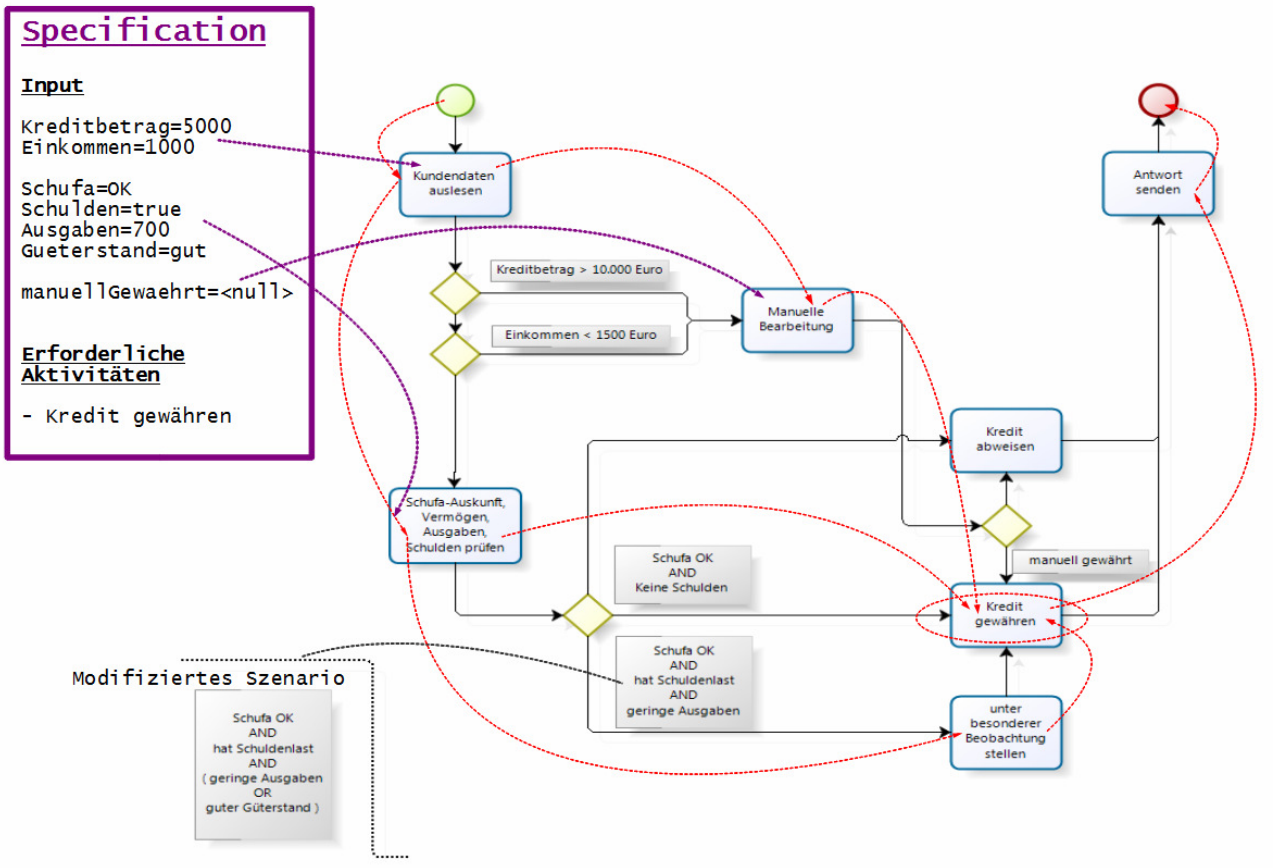

Abb. 20: Testfall mit Merhdeutigkeit

Wir sehen in der Abbildung einen Testfall, in dem es vor allem wichtig ist, dass unter den gegebenen Input-Daten der Kredit gewährt wird. Wie zu sehen, gibt es drei verschiedene Pfade, die alle zum gewünschten Ergebnis führen. Allerdings wäre durch eine solche Spezifikation die korrekte interne

Verarbeitung noch nicht völlig gewährleistet. Ein solcher Test hat Ähnlichkeiten mit einem *Black-Box-Test*. Es macht konkret für diesen Testfall beispielsweise keinen Unterschied ob der interne Ablauf des Workflows eine manuelle oder automatische Kreditprüfung verfolgt. Eine *Assertion* '`assert(false)`' an der richtigen Stelle würde die manuelle Bearbeitung ausschließen.

3.3.4. Weitere Konzepte/Komponenten

OTX-Exceptions

In der folgenden Tabelle sind die Standard Ausnahmefehler (*Exceptions*) aus dem *OTX-Core* aufgelistet, die in einem *OTX*-Ablauf auftreten können.

Durch das *Test Framework* soll ermöglicht werden, diese Ausnahmen an beliebiger Stelle des *OTX*-Ablaufs auszulösen, um das Verhalten der Ausnahmefehlerbehandlung *(Exception-Handling*) des Workflows gezielt zu testen. Es ist theoretisch auch vorstellbar, die Ausnahmefehlerbehandlung eines Workflows zu testen, indem man die in den *OTX*-Ablauf eingehenden Daten in einem Testfall genau so spezifiziert, sodass die zu prüfende Ausnahmefehlerbehandlung eintritt. Jedoch ist es bei komplexen Abläufen unter Umständen ein mühsames Geschäft, zurückzuverfolgen, welche Testdaten eine bestimmte Ausnahmefehlerbehandlung auszulösen vermögen. Allein der Name *Ausnahmefehlerbehandlung* drückt schon aus, dass es sich hierbei um seltene Ausnahmen handelt, die eigentlich in einem korrekten Ablauf mit vernünftigen Eingabedaten gar nicht auftreten sollten – es ist somit umso mehr erschwert die notwendigen Testdaten zu finden, um das Verhalten der gewünschten Ausnahmebehandlung zu testen. Aus diesem Grund sollte das Test-Framework es unterstützen, Fehlerausnahmen zu Testzwecken an beliebigen Stellen eines Workflows auszulösen.

In meinem Ansatz werde ich dies ähnlich realisieren wie bereits bei den *Assertions*: Es soll möglich sein ein sogenanntes *Throw-Element* an eine beliebige Aktivität zu binden. Dieses *Throw-Element* ist nichts anderes als die Spezifikation bzw. der Name einer im Standard definierten *OTX-Exception*, die geworfen werden soll, sobald die Ausführung der gebundenen Aktivität beendet wurde.

Nicht jede Aktivität in *OTX* kann jede *OTX-Exception* auslösen. Um die Programmkomplexität klein zu halten, soll die Testfall-Modellierung jedoch jede Kombination von *OTX-Exceptions* und Aktivitätstyp zulassen. Es macht zwar keinen Sinn etwa ein *Throw-Element* für *DiagComExceptions* an beispielsweise eine *Assignment*-Aktivität zu binden, stellt jedoch keine Beschränkung des Test-Frameworks dar, die Fehlerbehandlung eines *OTX*-Ablauf vollständig zu testen.

Da der *OTX*-Standard bei *ProcedureCalls* auch eine *throws*-Klausel spezifiziert – sprich: unbehandelte Ausnahmen in der aufgerufenen Prozedur werden einfach eine Ebene höher an den Aufrufer weitergegeben -, sollen auch *ProcedureCall*-Aktivitäten nicht von diesem *throws*-Mechanismus ausgeschlossen sein. Kurz: Es wird möglich sein ein *throw-Element* mit beliebiger *Exception*-Spezifikation an jede beliebige Aktivität zu binden.

Events

Ähnlich wie die Ausnahmefehlerbehandlung soll auch das *Event-Handling* gezielt testbar sein. Man stelle sich vor, in einem *OTX*-Ablauf werde ein *Event-Handler* durch ein zeitliches Event (z.B. wöchentlich an einem bestimmtem Tag) angestoßen. Dieser *Event-Handler* könnte dann nur zu genau diesem zeitlichen *Event* getestet werden.

Betrachten wir ein anderes Szenario, in dem ein *OTX*-Ablauf auf Benutzereingaben wartet (z.B. Tastatureingabe, oder Mausklick), welche durch *Events* realisiert werden. Um nun Testfälle automatisiert ablaufen zu lassen, muss es möglich sein in einem solchen Ablauf diese Benutzereingabe-Events zu simulieren bzw. durch das *Test Framework* auszulösen.

Analog zum *throw-Element* für Ausnahmefehler soll es also auch möglich sein ein *throw-Element* für Events an Aktivitäten zu binden. Der spezifizierte Event soll direkt nach der Ausführung der Aktivität geworfen werden.

DiagCom

Die *DiagCom*-Bibliothek stellt Elemente für die Kommunikation mit dem darunterliegenden Diagnoselaufzeitsystem (z.B. *MVCI*) zur Verfügung. Aktivitäten, welche die *DiagCom*-Schnittstelle betreffen, sollen durch Aufrufe an *Mock-Objekte* simuliert werden. Dieses Konzept lässt sich analog zu einem *ProcedureCall* realisieren. Rückgabeparameter von Aufrufen an *DiagCom* werden mit statischen Daten gefüllt, die im Testfall spezifiziert wurden.

Kontext-Konzept

Kontext-Variablen sollen durch die in einem Testfall spezifizierten Werte festgelegt werden, statt diese mit Daten aus der Diagnoseanwendung zu befüllen. So ist es möglich für den als generisch modellierten *OTX*-Ablauf verschiedene Umgebungen zu simulieren, ohne diese Umgebungen explizit für jeden Test herstellen zu müssen.

Validity-Konzept

Validities stellen (wie in Abschnitt 2.6.5 bereits besprochen) in *OTX* eine Möglichkeit dar, bestimmte Aktivitäten oder Elemente für die tatsächliche Ausführung ein- oder auszuklammern. *Validities* werden durch einen *Boolean*-Term ausgewertet, die sich in der Praxis in den meisten Fällen auf Kontext-Variablen beziehen. Solche *Validities* können gut durch das vorgestellte Konzept der *must visit*-Aktivitäten getestet werden. In einem Testfall werden zuerst die entsprechenden Kontext-Variablen spezifiziert, nun können all diejenigen *Validities*, die der Semantik nach korrekterweise eigentlich *'true'* auswerten sollten , als *must visit*-Aktivitäten spezifiziert werden. Der Testfall stellt sicher, dass in einem bestimmten Kontext, bestimmte Elemente ausgeführt werden – dies ist genau die Funktion die das *Kontext*- und das *Validities*- Konzept realisieren.

Signature-Konzept

Signatures werden dazu eingesetzt, um in einem Ablauf unter verschiedenen Testumgebungen dementsprechend auch verschiedene Implementierungen eines *ProcedureCalls* aufzurufen. Dieses lässt sich testen, indem

1. die gewünschte Testumgebung durch die Spezifizierung der *Kontext-Variablen* in einem Testfall simuliert wird (wie soeben oben im *Kontext*-Konzept besprochen), und
2. dementsprechend die dazugehörige Implementierung durch die Spezifizierung eines geeigneten *Mock-Objektes* in eben diesem Testfall simuliert wird.

3.4. Konzept/Überblick

In diesem Abschnitt werden die erarbeiteten Aspekte und die wichtigsten Punkte noch einmal zusammengefasst.

3.4.1. Testspezifikation

- Die Testspezifikation soll in einem vom *OTX* Workflow separatem *TCS-Dokument (TestCaseSpecification-Dokument)* festgehalten werden.
- Die Spezifikation von Eingabe- und Ausgabe- Größen des Workflows werden durch das Konzept von *Mock-Objekten* realisiert. Wie besprochen sind keine Eingabe-Parameter für externe Aufrufe notwendig, sondern es werden zu den jeweiligen Aktivitäten nur die Rückgabewerte spezifiziert.
- *must visit*-Aktivitäten werden in einer einfachen Liste mit ihrem eindeutig referenzierbaren Namen spezifiziert.
- *Assertions* innerhalb des Workflows werden durch boolesche Ausdrücke realisiert und müssen eine eindeutige Referenz zu einer Aktivität haben.

3.4.2. Testorganisation

- Um Testfälle zu strukturieren, werden sie zu *Testsuites* organisiert.
- Damit gibt es die Möglichkeit eine Menge an Testfällen zusammenzufassen, die eine bestimmte Funktion oder ähnliche Funktionen des Workflows testen.
- Eine *Testsuite* enthält somit eine Menge von Testfällen. Es ist dadurch möglich Testfälle wiederholt und automatisiert prüfen zu lassen.
- In einem Testlauf (*Testrun*) können mehrere *Testsuites* sowie Testfälle automatisiert durchgeführt werden.

3.4.3. Testdurchführung

Bei der Ausführung sind folgende Punkte zu beachten:

- Um den *OTX*-Workflow ausführbar zu machen, wird C#-Code erzeugt. Das *TCS-Dokument* muss während der Code-Erzeugung in den *OTX*-Workflow integriert werden.
  - Anstelle von externen Prozeduraufrufen werden für Out- bzw. Inout- Parameter konstante Werte gesetzt, die in dem *TCS-Dokument* spezifiziert sind.
    - Alternativ: Aufruf an *dummy*-Prozedur (*Mock-Object),* der keinen Code ausführt, sondern sofort die besagten konstanten Werte zurückgibt.
  - *Assertions* werden im C#-Code nach einer Aktivität geprüft.
  - Fehlgeschlagene *Assertions* brechen die Ausführung nicht ab.
  - Unbehandelte *OTX-Exceptions* müssen auf Testfall-Ebene abgefangen werden. Die Ausführung von anderen Testfällen darf nicht von einer unbehandelten Fehlerausnahme des aktuellen Testfalls tangiert werden.
  - Nach Ausführung einer *must visit*-Aktivität wird die Aktivität in der Liste der *must visit*-Aktivitäten als "erfüllt" markiert.
  - Am Ende der Ausführung des Workflows muss die Liste der *must visit*-Aktivitäten geprüft werden.
  - Die Ergebnisse der *Assertions* sowie der *must visit*-Aktivitäten müssen gesammelt und gespeichert werden. Die Ergebnisse müssen auch nach Ausführung des Testfalls zur Präsentation bereitstehen.
- Es muss möglich sein eine *Testsuite* oder mehrere *Testsuiten* in einem Testdurchlauf automatisiert durchlaufen zu lassen.
- Dabei ist zu beachten, dass die Ausführung der Testfälle keinen Einfluss aufeinander haben darf.
- Insbesondere unbehandelte Fehler, die beim Testdurchlauf von Workflows auftreten, dürfen den Rest des Testdurchlaufs nicht abbrechen oder in irgendeiner Weise beeinflussen. *Faults/Exceptions* müssen vom *Test Framework* abgefangen werden.

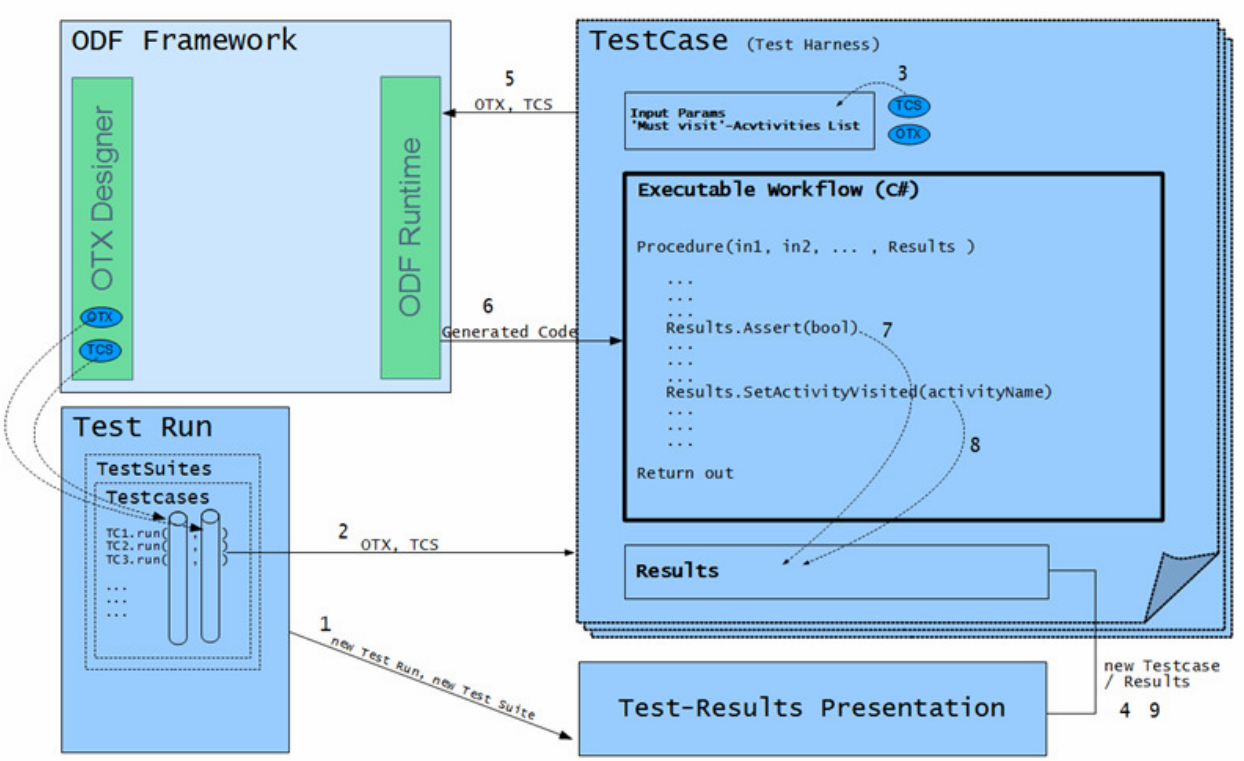

Abb. 21: Grobe Darstellung des Konzepts

In der obigen Abbildung ist eine grobe Übersicht zu sehen, die die Durchführung von Tests beschreibt. Ein vorher mit dem Designer spezifizierter Testfall bzw. mehrere Testfälle in einem oder vielen Test-Suiten werden in einem Test-Run durchgeführt. Das dafür zuständige *Test Run-Modul* sammelt hierzu die *OTX*- und *TCS*- Daten und erzeugt die für das Testen des Workflows benötigte Test-Umgebung (*Test-Harness*). Die einzelnen Schritte zur Ausführung eines Testfalles werden nachfolgend erläutert.

1. Daten über den *Test Run* und des auszuführenden *Testsuites* werden dem Modul *Test-Results Presentation* zur Darstellung des Testlaufs übergeben.
1. Das *Test Run*-Modul erzeugt und initialisiert eine *TestCase*-Klasse, die die Test-Umgebung für die Ausführung des Workflows bereitstellt. Dabei werden das entsprechende *OTX*- und *TCS*-Dokument übergeben.
2. Aus dem *TCS*-Dokument werden die Input-Parameter für das Starten des Workflows und die *must visit*-Liste ausgelesen.
3. Daten über den Testfall werden an *das Test-Results Presentation-Modul* übergeben.
4. Das *OTX*- und *TCS*- Dokument wird zur Code-Generierung an die *ODF Runtime* übergeben.
5. Aus dem *OTX*- und *TCS*- Dokument wird eine C#-Klasse generiert, die den Workflow abbildet. Sie wird anschließend von der *TestCase*-Klasse ausgeführt.
6. Eine *Assertion* wird während der Ausführung ausgewertet, indem die `assert()`–Methode des referenzierten *Results-Objektes* der entsprechenden *TestCase*-Klasse aufgerufen wird.
7. Auch das Konzept von *must visit*-Aktivitäten wird durch eine Methode des *Results-Objekts* realisiert. Hierzu wird die Methode `SetActivityVisited()` aufgerufen, um die entsprechende Aktivität als besucht zu markieren.

8. Änderungen am *Results-Objekt* können zur sofortigen Darstellung der Ergebnisse an das *Test-Results Presentation*-Modul weitergegeben werden.
9. Nach Ausführung des Workflows wird die Liste der *must-visi'*-Aktivitäten ausgewertet.
10. Die Referenz auf die ausführbare Workflow Klasse sollte gelöscht werden, um Speicher freizugeben.
11. Der Testfall ist abgeschlossen; der nächste Testfall wird durchgeführt.

3.4.4. Darstellung

Testfallmodellierung

- Die Spezifizierung eines Testfalls sollte unmittelbar am Workflow selbst geschehen. Das heißt, dass der Entwickler keinen Quellcode schreiben muss, sondern bei der Modellierung mit einem erweiterten *OTX-Editor* arbeitet. Evtl. sollte es einen *Modus-Switch* geben, der speziell die Modellierung von Testfällen unterstützt.
- Zur Spezifikation eines Testfalles gehören, wie gehabt, die Eingabe- und Ausgabe- Größen des Workflows, *Assertions* und *must-visit*-Aktivitäten. *Assertions* und *must visit*-Aktivitäten werden als zusätzliche *Properties* den bestehenden Aktivitäten im Workflow angereichert. Die Erstellung von *Assertions* sollte mit einem Ausdruckseditor unterstützt werden.
- Für *must visit*-Aktivitäten reicht es, als *Property* ein einzelnes Flag zu setzen. Praktisch wäre auch ein *Modus-Switch* (durch *Hotkey* oder Klick auf ein Button), der es erlaubt Aktivitäten durch einen einfachen Mausklick zur der Liste der *must visit*-Aktivitäten hinzuzufügen bzw. zu entfernen.
- Eingehende Daten können auch als *Properties* an den entsprechenden Aktivitäten modelliert werden. Es ist jedoch vorteilhaft zusätzlich alle Eingabe- und Ausgabe- Daten gesammelt als eine Übersicht in einer editierbaren Tabelle festzuhalten. So ist auf einem Blick ersichtlich, welche Testdaten der jeweilige Testfall spezifiziert, ohne dass auf die entsprechenden Aktivitäten einzeln zugegriffen werden muss, um deren *Properties* anzuzeigen. Die Füllung der Testdaten in einer Tabelle ist auch schneller. Des Weiteren ist schnell ersichtlich, ob evtl. noch Parameter fehlen, oder falsch eingegeben wurden. Das Ausfüllen der Tabelle kann natürlich noch zusätzlich dadurch unterstützt werden, dass die Aktivität des gerade zu bearbeitenden Eintrags graphisch hervorgehoben wird - denn es ist nicht immer sofort klar, zu welcher Aktivität der Parameter gehört.

Testergebnisse

- Es muss davon ausgegangen werden, dass u.U. eine große Zahl an *Testsuites*, mit wiederum einer großen Zahl an Testfällen, in einem Lauf getestet werden. Und einzelne Testfälle

enthalten wiederum eine Menge an Zusicherungen. Die Darstellung der Testergebnisse sollte übersichtlich sein und es sollte schnell erkennbar sein, welche Tests erfolgreich verliefen und welche davon fehlschlugen.

- Es bietet sich hier eine Baumstruktur an, die als Wurzel den Testdurchlauf enthält. Kinder dieser Wurzel sind *Testsuites*, die wiederum Testfälle als Kinder besitzen. Auf der untersten Ebene unter den Testfällen befinden sich die Zusicherungen an die jeweiligen Testfälle.
- Fehlschlag oder Erfolg von Zusicherungen oder Testfällen oder *Testsuites* werden durch ein rotes bzw. grünes Symbol dargestellt. Darüber hinaus sollten unbehandelte Fehler in einem Testfall auch dargestellt werden.

4. Implementierung

4.1. Grobarchitektur

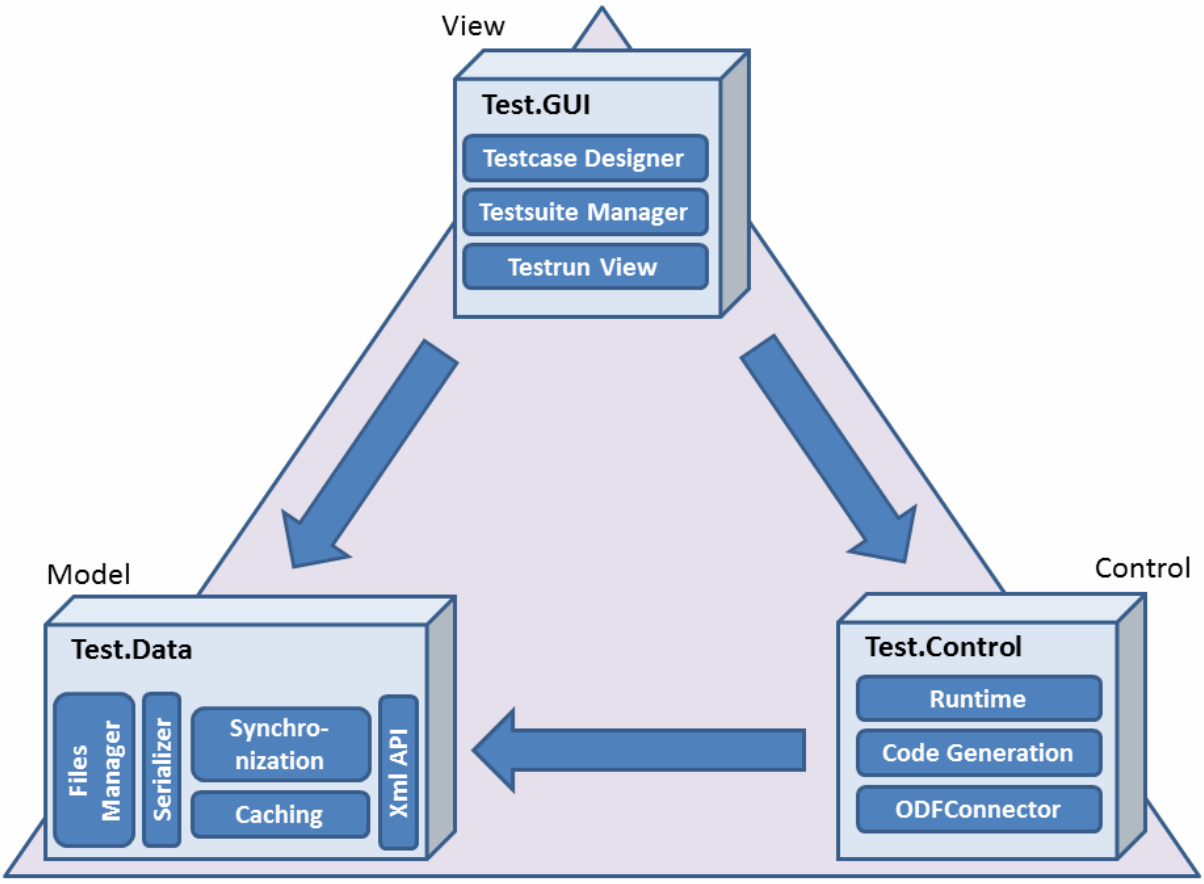

Abb. 22: Test Framework Architecture

Das Framework ist in drei Teile aufgeteilt. Da in C# unter Visual Studio entwickelt wird, sind diese drei Teile in Visual-Studio zu je einem Projekt zugeordnet – d.h. letzen Endes werden diese als drei dll's bzw. *dotNET*-Bibliotheken assembliert werden. Der Aufbau lehnt sich an das klassische *MVC* Architektur-Entwurfsmuster (Model-View-Control).
In ODF integriert, werden für das *Test Framework* die folgenden Namen bzw. *Namespaces* verteilt: *Emotive.Odf.Test.Data, Emotive.Odf.Test.Runtime* und *Emotive.Odf.Test.Gui.*

Test.Data (Model)

Test.Data ist das **Datenmodell** des Frameworks. Hier wird die Datenstruktur spezifiziert. Außerdem sorgen die vier Module in diesem Teil für den Zugriff, die Modifikation, die Persistenz und die automatische Aktualisierung aller für das Testen von *OTX*-Abläufen benötigten Testdaten.

- Die **Xml-API** bietet hierzu die Schnittstelle für sämtliche Funktionen an.

- Der **Files Manager** stellt die Verbindung zum Dateisystem des laufenden Betriebssystems her. Er ist für das Lesen und Schreiben der serialisierten Daten zuständig, und sorgt zudem für die Projektverwaltung und eine geeignete, festgelegte Ordnerstruktur.
- Der **Serializer** serialisiert bzw. deserialisiert C#-Klassen zum spezifizierten Datenformat und umgekehrt.
- Das **Caching**-Modul hält Testdaten im Speicher um Such- bzw. Filter- Anfragen, Referenzenanpassungen, etc… einfach und effektiv umzusetzen.
- Darüber sitzt das **Synchronization**-Modul, welches sämtliche Funktionen bereithält, um die Testdaten mit den dazugehörigen Daten des *OTX*-Ablaufs synchron zu halten.

Test.Control (Control)

Die **Runtime** stellt die Laufzeitumgebung für die Durchführung der Tests an den Diagnoseabläufen dar. Sie beinhaltet die komplette Steuerungslogik um Testläufe durchzuführen und auszuwerten. Beim Durchlauf eines Tests werden dazu aus den Testdaten und den *OTX*-Daten die nötige Testumgebung als sogenannte **Test Harness**[10] initialisiert und der Programmcode zur Ausführung von Tests durch das **Code-Generation** Modul dynamisch generiert. Die *Test Harness* stellt alle Daten des aktuell auszuführenden Testfalles zur Verfügung und bietet zudem sonstige Funktionen und Mechanismen zur Prüfung von *Assertions* oder *must visit*-Aktivitäten, etc. Ergebnisse eines Testlaufes gehen über die *Xml-API* an *Test.Data* – sie werden aber auch sogleich am *User Interface* dargestellt (siehe unten *Observer-Pattern*).

Der **ODFConnector** stellt die Schittstelle für das *ODF* zur Verfügung. Die gesamte Kommunikation zwischen *ODF* und *Test Framework* läuft über dieses Modul ab.

Test.Gui (View)

Die *Test.Gui* ist die Präsentationsschicht und beinhaltet sämtliche *User Interfaces*. Zu nennen wären da

- der **Testcase Designer**, der z.T. im *Workflow-Designer* (auch *Otx-Editor)* integriert ist und das Interface zur Testfall Modellierung bereithält.
- der **Testsuite Manager**, der eine einfache und effektive Zusammenstellung von *Testsuites* unterstützt.
- der **Testrun View**, der einen Testlauf mit Ergebnissen übersichtlich darzustellen hat.

Observer-Pattern

Für die Aktualisierung aller *Views* bzw. *User-Interfaces* wird das **Observer-Pattern** implementiert. *Test.Gui* wird als *Observer (Beobachter*) bzw. *Listener/Subscriber* von *Test.Data* dem *Event-Publisher* gehandelt. Veränderungen im Datenmodell werden also durch *Events* sofort im *User-Interface* dargestellt.

[10] engl. für Testgeschirr

4.2. Test.Data

4.2.1. Serializer & Xml-API

Die zu einem *OTX*-Ablauf spezifizierten Testdaten werden als ein *XML*-Dokument persistent gemacht. Dabei eignet sich *XML* als sowohl Menschen- als auch Maschinen- lesbare Sprache zu solchen Zwecken besonders gut, da sie eine mächtige und vor allem erweiterbare und selbstbeschreibende Sprache ist, die in der modernen Softwareentwicklung allgemein bekannt, akzeptiert und weitläufig Verwendung findet.

Das Datenmodell des Frameworks gründet auf der *Xml*-Datenstruktur. Sämtliche Testdaten zu einem *OTX*-Ablauf werden letzten Endes in der *Xml*-Struktur abgelegt. Die *Xml-API* stellt hierzu alle Funktionen zur Verfügung, die zum Lesen, Ändern und Schreiben der Daten nötig sind. Desweiteren kapselt sie aber auch die Funktionalität der Synchronisation zwischen Testdaten und *OTX*-Daten, sowie die Funktionalitäten der Datei- und Projekt- Verwaltung, die hintergründig der *File Manager* erledigt.

Das Transformieren der, im Speicher als C#-Klassen vorliegenden, Testdaten wird auch *Serialisierung* genannt und wird von Haus aus vom *dotNET*-Framework unterstützt. Aus einer *Xml-Schema* Definition (Beschreibung) lassen sich automatisiert C#-Klassen generieren, die der spezifizierten Datenstruktur des *Xml-Schemas* entsprechen. Um *Xml-Daten* in diese Klassen einzulesen (*Deserialisierung*), müssen nur die vom dotNET-Framework bereitgestellten Funktionen aufgerufen werden – das gleiche gilt für den umgekehrten Fall, der *Serialisierung*.

Wir befassen uns nun mit der grundlegenden Datenstruktur. Um die *Xml*-Sprache an die Bedürfnisse des Test Frameworks anzupassen bzw. zu erweitern, ist es nötig eine *Xml-Schema* Definition zu spezifizieren. An dieser Stelle will ich die *Xml-Schema* Definition nur einiger Hauptelemente vorstellen und erläutern.

Testcase

Wir schauen uns als Erstes die Definition des übergeordneten *Testcase* an:

```
<xsd:complexType name="TestCase">
  <xsd:complexContent>
    <xsd:extension base="TestResultBase">
      <xsd:sequence>
        <xsd:element name="Id" type="xsd:string"/>
        <xsd:element name="WorkflowRef" type="xsd:string"/>
        <xsd:element name="WorkflowName" type="xsd:string"/>
        <xsd:element name="ContextVariables" type="tf:ContextVariables"/>
        <xsd:element name="InParameters" type="tf:Parameters"/>
        <xsd:element name="ExpectedOutParameters" type="tf:Parameters"/>
        <xsd:element name="OutParameterAssertions" type="tf:Assertions"/>
        <xsd:element name="Mocks" type="tf:Mocks"/>
        <xsd:element name="Activities" type="tf:Activities"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Abb. 23: Xml-Schema Definition, Testcase

Ein *Testcase* beinhaltet alle notwendigen Testdaten, um ein *OTX*-Ablauf bzw. - im Terminus von *OTX* – eine *Procedure* auszuführen und zu testen. Wichtig sind folgende Elemente:

- **WorkflowRef:** zur eindeutigen Referenzierung des Workflows bzw. der *Procedure* (ein *OTX*-Ablauf ist durch das *Otx-Package*, das *Otx-Document* und die *Otx-Procedure* eindeutig identifizierbar),
- **ContextParameters:** Testwerte für globale Kontextvariablen in einem *OTX*-Dokument.
- **InParameters:** Testwerte für die beim Prozeduraufruf übergebenden *Parametern*,
- **ExpectedOutParameters:** Die vom Prozeduraufruf zurückgegebenen, erwarteten Werte,
- **Mocks:** Simulationen von Prozeduraufrufen innerhalb der auszuführenden Prozedur,
- **Activities:** Aktivitäten, die besucht werden müssen (*must visit*-Aktivität) oder die an sie gebundene *Assertions* besitzen.

Activities

```
<xsd:complexType name="Activity">
  <xsd:complexContent>
    <xsd:extension base="TestResultBase">
      <xsd:sequence>
        <xsd:element name="Assertions" type="tf:Assertions"/>
        <xsd:element name="Events" type="tf:ThrowEvents"/>
        <xsd:element name="Exceptions" type="tf:ThrowExceptions"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Abb. 24: Xml-Schema Definition, Activity

Da in den Anforderungen verlangt wird, die Testdaten für ein *OTX*-Ablauf separat von dem *OTX*-Dokument abzulegen ist es nötig *Assertions* an ein geeignetes *OTX-Element* zu binden, um sie zur Testlaufzeit zur gewünschten Zeit bzw. an der geforderten Stelle zu testen. *Activities* beinhalten je eine oder mehrere *Assertions*. Das gesagte gilt gleichermaßen auch für Events und *Exceptions*.

Assertions

```
<xsd:complexType name="Assertion">
  <xsd:complexContent>
    <xsd:extension base="TestResultBase">
      <xsd:sequence>
        <xsd:element name="Id" type="xsd:string"/>
        <xsd:element name="Expression" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Abb. 25: Xml-Schema Definition, Assertion

Assertions besitzen zur Referenzierung eine eindeutige *Id*. Außerdem spezifizieren sie ein *Expression-Element*, welches aus einer Zeichenkette besteht, die ein gültiger, auswertbarer *Boolean*-Ausdruck sein muss. Dieser Ausdruck wird auch bei der Codegenerierung benutzt und wird bei der Testlaufzeit vom Compiler ausgewertet. Die Prüfung der *must visit*-Aktivitäten werden durch spezielle *Assertions* realisiert, deren *Boolean*-Ausdruck immer *'true'* ist. Und auch die Prüfung der ausgehenden Parameter des Ablaufs mit den erwarteten Werten wird intern durch *Assertions* realisiert. Diese speziellen *MustVisitAssertions* und *OutParamAssertion* werden als *Extension* von der `Assertiontype` abgeleitet.

Mocks & ProcedureCall

```
<xsd:complexType name="Mocks">
  <xsd:sequence>
    <xsd:element name="ProcedureCall" type="tf:ProcedureCall" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ProcedureCall">
  <xsd:complexContent>
    <xsd:extension base="TestBase">
      <xsd:sequence>
        <xsd:element name="OutParameters" type="tf:Parameters"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Abb. 26: Xml-Schema Definition, Mocks

Simulationen von Prozeduraufrufen werden durchgeführt, indem Ausgabeparameter mit in der Designzeit festgelegten Testwerten belegt werden. Der `Mocks`-Typ listet dazu alle Prozeduraufrufe auf, die wiederum im Element **Outparameters** die Parameter bzw. ihre Rückgabewerte festlegen.

Parameters

```
<xsd:complexType name="Parameters">
  <xsd:sequence>
    <xsd:element name="Param" type="tf:Parameter" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Parameter" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="TestBase">
      <xsd:sequence>
        <xsd:element name="Datatype" type="tf:dataTypes"/>
        <xsd:element name="Modifier" type="tf:paramModifier"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Abb. 27: Xml-Schema Definition, Parameters

Die Werte, der in den *OTX*-Ablauf eingehenden Daten, werden in dem *Parameter-Element* gespeichert. Auch der dazugehörige Datentyp und die *Modifier (in, out, inout)* werden hier festgehalten.

```
<xsd:complexType name="IntegerType">
      <xsd:complexContent>
          <xsd:extension base="Parameter">
              <xsd:sequence>
                  <xsd:element name="Value" type="xsd:long"/>
                  <xsd:element name="InitValue" type="xsd:long"/>
              </xsd:sequence>
          </xsd:extension>
      </xsd:complexContent>
  </xsd:complexType>
```

Abb. 28: Xml-Schema Definition, IntegerType

Abb. 28: Xml-Schema Definition, IntegerType zeigt als Beispiel die Datenstruktur für einen *Integer*-Wert mit Initial-Wert (siehe *OTX*-Standard: *initValue*).

Base-Elemente

```
<xsd:complexType name="TestBase">
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string"/>
    <xsd:element name="Description" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="TestResultBase">
  <xsd:complexContent>
    <xsd:extension base="TestBase">
      <xsd:sequence>
        <xsd:element name="TestResults" type="tf:TestResults" minOccurs="0" />
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="TestResults">
  <xsd:sequence>
    <xsd:element name="Result" type="tf:ResultEnum" />
    <xsd:element name="ResultString" type="xsd:string"  />
    <xsd:element name="ExecutionDate" type="xsd:date"  />
    <xsd:element name="Duration" type="xsd:double" />
  </xsd:sequence>
</xsd:complexType>
```

Abb. 29: Xml-Schema Definition, Base-Elemente

Alle für das Framework spezifizierten *Xml*-Typen werden als *extension* vom Typ `TestBase` modelliert. Das heißt, dass sie alle die Elemente *Name* und *Description* von `TestBase` erben. Dies ist vernünftig, da alle Typen, einen Namen und eine Beschreibung besitzen müssen bzw. können.
Die Typen `Testcase`, `Activity`, `Assertion` und noch einige andere werden von `TestResultBase` abgeleitet, welches wiederum von `TestBase` ableitet. Diese sind Typen, die in

einem Testdurchlauf ein Ergebnis zurückgeben – nämlich *'success'*, *'fail'*, oder *'skip'*. Dementsrechend besitzen sie neben *Namen* und *Description* noch weitere Elemente, wie z.B. *Result*, zur Speicherung der Ergebnisdaten. Eine *Extension* in der *Xml-Schema Definition* wird in C# durch eine Subclass der *extension base* abgebildet.

Dieser Abschnitt umfasst nur einen Teilausschnitt der *Xml*-Datenstruktur. Ein vollständigeres Bild über die Datenstruktur sowie dessen Interaktion mit dem restlichen Datenmodell liefert das Klassendiagramm auf der nächsten Seite. Es ist zu sehen, dass die aus der *Xml-Schema* generierten C#-Klassen um viele weitere Felder und Methoden erweitert sind, um die Funktionalitäten der *Xml-API* zu realisieren. Das *dotNET*-Framework bietet dazu an bei der Klassendeklaration das Keyword *partial* zu verwenden, um Klassen über mehrere Dateien hinweg zu implementieren. Die automatisch generierten C#-Klassen sind standardmäßig allesamt mit *partial* gekennzeichnet.

Hinweis (betrifft alle Klassendiagramme dieser Arbeit): Wenn auch die Klassendiagramme in dieser Arbeit allesamt eine „nur" vereinfachte Darstellung der tatsächlichen Implementierung geben, setzen sie den Fokus doch auf die Hauptelemente und Funktionen des Frameworks und stellen so eine gute Übersicht und Zusammenfassung über die technische Funktionsweise des Frameworks dar. Die Klassendiagramme erheben somit also keinen Anspruch auf Vollständigkeit (Beispielsweise sind in der nächsten Abbildung nicht alle Datentypen von *OTX* abgebildet. Auch sind eine Menge privater Methoden, die für das Verständnis des Grundkonzepts unerheblich sind, ausgeblendet.).

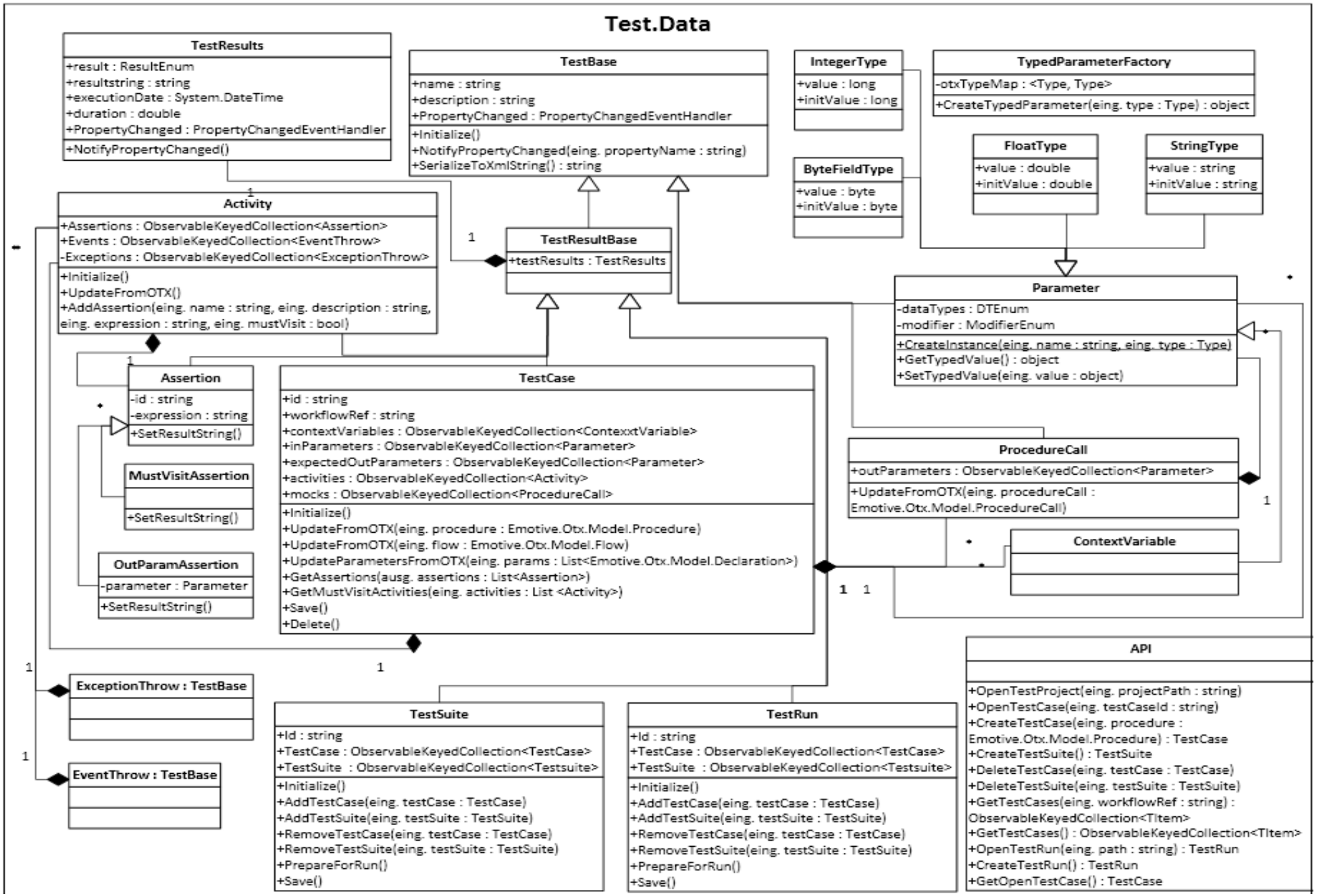

Abb. 30: UML Klassendiagramm, Test.Data pt.1

4.2.2. Files Manager / Caching

In dieser Abbildung ist die Fortsetzung des Klassendiagramms von *Test.Data* abgebildet. Hier sehen wir vor allem die Funktionalität des *Files Managers* sowie des *Caching*-Mechanismus.

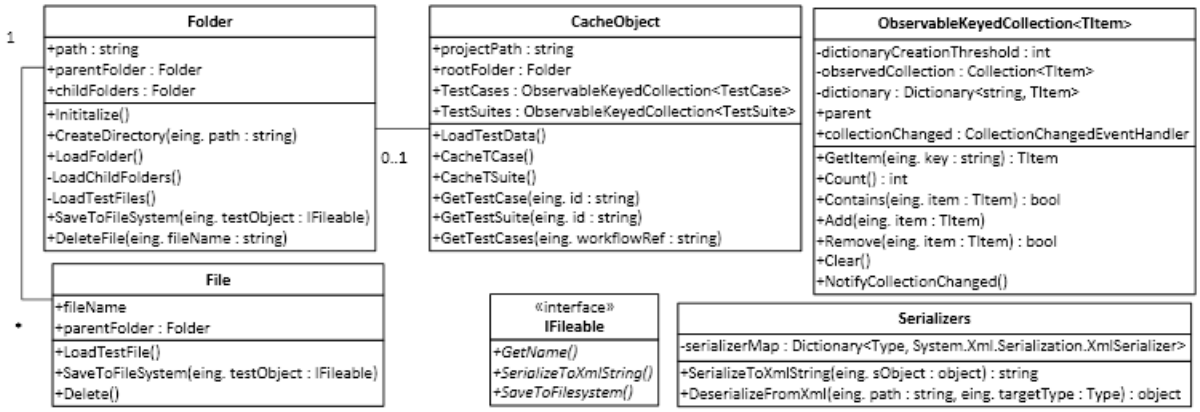

Abb. 31: UML Klassendiagramm, Test.Data pt.2

Files Manager

Der *Files Manager* ist für die Datei- und Projekt-Verwaltung zuständig. Für jedes *OTX-Projekt* wird dazu ein dediziertes Verzeichnis erstellt, der die gesamten Testdaten enthält. Es gibt zudem spezielle Unterverzeichnisse für *Testcases, Testsuites* und *Testruns* (Testlauf). Die Verzeichnisstruktur ist wie folgt aufgebaut:

- **Root:** ~ProjectHome\TestUnits\
- **Testsuites:** ~ProjectHome\TestUnits\Testsuites\
- **Testruns:** ~ProjectHome\TestUnits\Testruns\
- **Testcases:** ~ProjectHome\TestUnits\Testcases\${OtxPackage-Name}\${OtxDocument-Name}\${OtxProcedure-Name}\

Die Verzeichnisstruktur wird (falls nötig) beim Öffnen eines Projektes, bzw. zur Erstellung eines Testfalls automatisch angelegt. Beim Laden eines Projektes wird die Verzeichnisstruktur im Speicher durch die Klassen `Folder` und `File` nachgebildet. Jeder Klassentyp der im Dateisystem persistent gemacht werden kann, muss mit einer `File`-Instanz gelinkt sein und das Interface `IFileable` implementieren, welches Informationen und Funktionen zur Serialisierung und zum Speichern zugreifbar macht. Soll eine Instanz mit `IFileable`-Implentierung im Dateisystem abgelegt werden, muss lediglich die Methode `SaveToFilesystem` aufgerufen werden. Die Erstellung der Verzeichnisstruktur und alles weitere erledigt der *Files Manager*.

Caching

Aufgrund der Referenzen von *Testsuites* und *Testruns* auf *Testcases* und zur Realisierung der Such- und Filteralgorithmen werden alle projektweiten *Testcases* zum jetztigen Stand der Entwicklung vollkommen in den Speicher geladen und durch die `CacheObject`-Klasse zugreifbar gemacht. Die ***Abb. 32: UML Sequenzdiagramm, LoadTestData*** zeigt einen Ladevorgang während des Öffnens eines Testprojektes. Hier werden die einzelnen Schritte zum Laden der Testdaten in das `CacheObject` in einem Sequenzdiagramm graphisch veranschlaulicht.

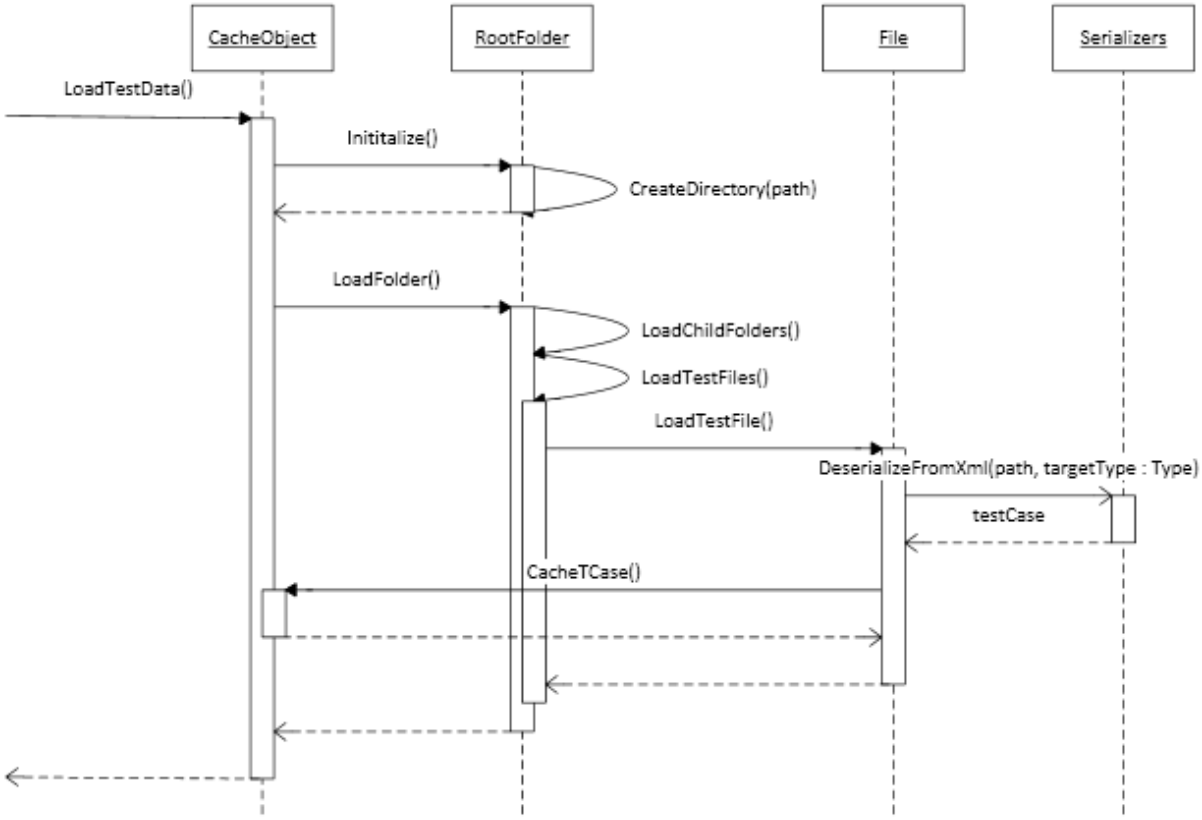

Abb. 32: UML Sequenzdiagramm, LoadTestData

ObservableKeyedCollection

Um auf der einen Seite auf die Menge der Testdaten effizient zugreifen zu können und auf der anderen Seite aber auch einen möglichst geringen Speicherverbrauch zu erreichen, wurde für das Test Framework eine geeignete Datenstruktur entwickelt. Das *dotNET*-Framework generiert für *Xml*-Strukturen die einen Container für eine beliebige Anzahl eines Elementes modellieren, standardmäßig eine Array-Datenstruktur. Array-Zugriffe sind wie bekannt sehr schnell. Für die Suche nach einem Element, der einen bestimmten Namen hat, ist eine Array-Struktur jedoch weniger effizient (*worst case: O(n)*).

Da im Test Framework unter Umständen eine sehr hohe Anzahl an Testfällen (>10.000) verarbeitet werden müssen und auf diese sowie andere Datentypen über ein Bezeichner (*Name* oder *Id*) zugegriffen wird, eignen sich *Hash*-Datenstrukturen besser. Der Zugriff über einen Bezeichner kann

dadurch mit O(1) erfolgen. `ObservableKeyedCollection` implementiert eine generische Klasse, die eine Menge von Datentypen einer Sorte sammelt, d.h. referenziert. Darüber hinaus müssen die gesammelten Datentypen das Interface `IKeyInterfaced` implementieren. Dieses Interface stellt die Methode offen, die verwendet wird, um auf den Bezeichner eines Datentyps zuzugreifen. Instanzen dieses Datentyps werden dann mit diesem Bezeichner als *Key* und einem *Hash* in einer *Dictionary*-Klasseninstanz gespeichert, die privat in der `ObservableKeyedCollection` gehalten wird. Auf diesem Weg ist es möglich den Bezeichner bzw. *Key* – meistens *Id* oder *Name* – durch die bereitgestellte Interface-Methode in dem Datentyp selbst festzulegen.

Da `ObservableKeyedCollection` aber auch für Datentypen genutzt wird, von denen in einer *Collection* nur wenige Instanzen gesammelt werden, ist es nicht ratsam den Mehraufwand sprich den erhöhten Speicherbedarf eines *Dictionary's* zu betreiben. Zum Beispiel wird in der Praxis an eine *Activity* oft nur eine *Assertion* und in den meisten Fällen sicher unter fünf *Assertions* gebunden – oder ein Prozeduraufruf wird in den seltensten Fällen mehr als zehn Parameter haben. Aus diesem Grund wird die interne Verwaltung nicht standardmäßig mit einem *Dictionary* vollzogen, sondern es lässt sich für eine `ObservableKeyedCollection`-Instanz eine *DictionaryCreationThreshold* festlegen, die die numerische Grenze für die Erstellung eines *Dictionary's* beschreibt.

Eine weitere Funktion der `ObservableKeyedCollection` ist die *observable*-Eigenschaft. Diese Funktion lässt einen Beobachter bzw. *Subscriber* der *Collection* zu, der benachrichtigt wird, sobald Schreib-Aktionen (z.B. *Add* oder *Delete*) an der *Collection* stattgefunden haben. Diese Implementierung des *Observer*-Patterns ist vor allem für die Aktualisierung des *User-Interfaces* gedacht und wird in ***Kapitel 4.4: Test.GUI*** näher erläutert.

4.2.3. Synchronization

In diesem Abschnitt geht es um die Synchronisation der Testdaten, d.h. in erster Linie die Testfallspezifikation, mit den Daten der zugeordneten *OTX-Prozedur*. Um einen erfolgreichen Testdurchlauf zu gewährleisten, müssen die Testdaten sowie die *OTX-Daten* sozusagen miteinander kompatibel bzw. synchron sein. Zum Beispiel müssen Testwerte für Parameterübergaben vom Datentyp her, mit den in *OTX* spezifizierten Datentypen übereinstimmen. Oder, der bool'sche Ausdruck einer *Assertion* muss gültig, bzw. auswertbar sein. Das *Test Framework* sorgt direkt bei der Erstellung dieser Daten für eine Prüfung bzw. Synchronisation. Allerdings kann eine nachträgliche Änderung von *OTX-Daten* zu einer Inkonsistenz führen, die unter Umständen den Testfall nicht mehr ablauffähig macht. Noch schlimmer wäre der denkbare Fall, in dem eine gemachte Veränderung zwar den Ablauf nicht stört, aber zu falschen Ergebnissen führt.

Wir sehen also, dass es sehr wichtig ist, Testdaten und *OTX-Daten* konsistent zu halten. Wie erwähnt soll das *Test Framework* schon bei der Erstellung von Testfällen die Konsistenz wahren. Weiter ist es

vernünftig eine Prüfung und Aktualisierung bei jedem Öffnen eines vorhandenen Testfalls bzw. vor jeder Ausführung eines Testlaufs durchzuführen.
Der Prüfungs- und Aktualisierungs- Prozess durchläuft bei seiner Durchführung rekursiv den gesamten *OTX-Baum* eines Ablaufs und prüft bzw. vergleicht die Daten. Die folgende Tabelle fasst zusammen, welcher Handlungsbedarf je im Falle eines bestimmten inkonsistenten Datenbefundes notwendig ist.

Befund	**Handlung** (immer am Testcase - keine Modifikation des OTX-Dokumentes!)
1. OTX: Aktivität fehlt.	*Lässche Aktivität samt Assertions löschen.* *Optional: Markiere als inaktiv.*
2. TestCase: Aktivität fehlt.	*Ok. Aktivität hat noch keine Assertions.*
3. OTX: Name der Aktivität geändert.	*Äquivalent zu Fall 1 und 2*
4. OTX: Parameter fehlt.	*Lösche Parameter.*
5. Testcase: Parameter fehlt.	*Füge Parameter hinzu .*
6. inkompatibler Parametertyp	*Generiere Parameter neu.*
7. Testcase: Expression der Assertion ungültig.	*Setze Expression = 'true'.*
8. Bei jeglicher Inkosistenz	*Verfahre nach 1.-7. und markiere zusätzlich mit Changed-Flag*

Tabelle 3: Synchronisierung mit OTX

Die Synchronisation wird in den betreffenden Klassen durch die Methode `UpdateFromOTX` implementiert (siehe dazu vorangegangene ***Abb. 30:UML Klassendiagramm, Test.Data pt.1***).

4.3. Test.Control

Die nachfolgende Abbildung stellt das Klassendiagramm von *Test.Control* dar.

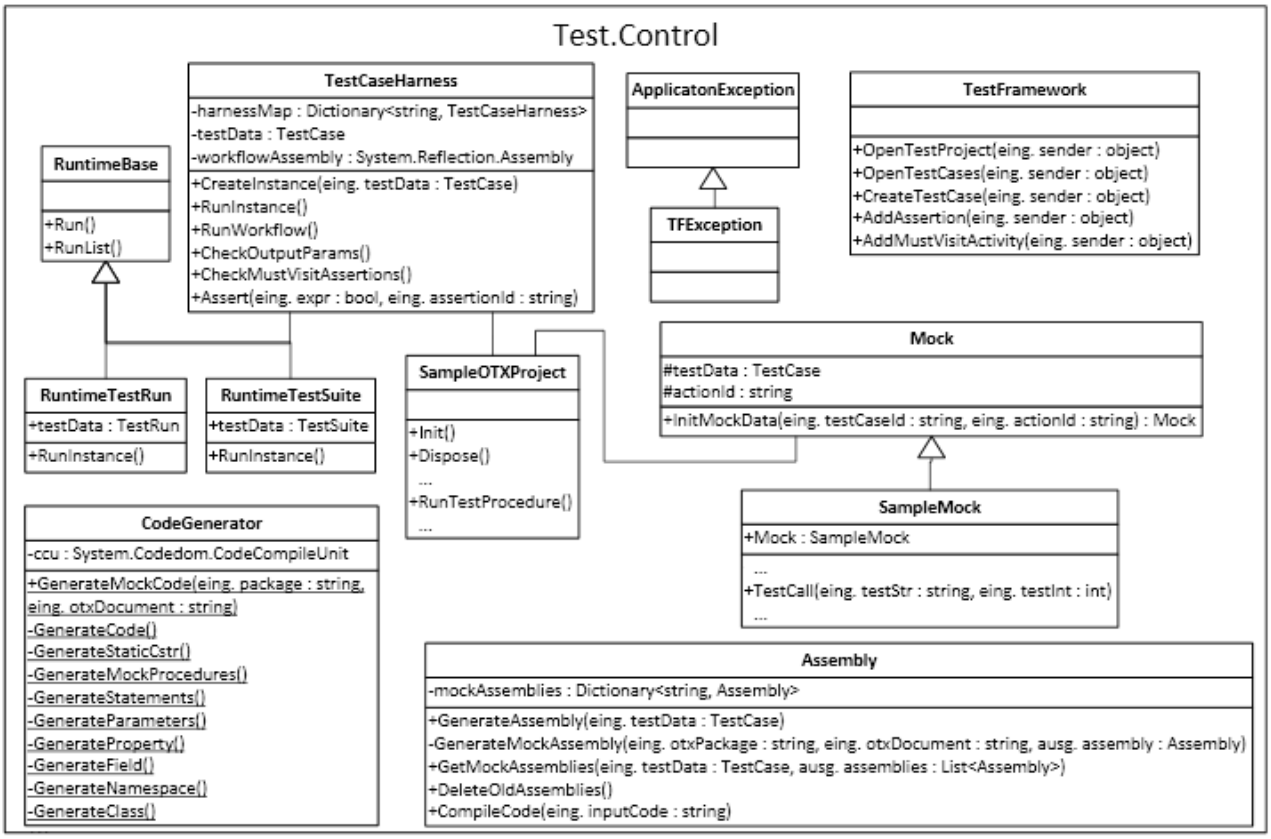

Abb. 33: UML Klassendiagramm, Test.Control

4.3.1. ODFConnector

Der *ODFConnector* stellt die Verbindung zwischen dem *ODF* und dem *Test Framework* her. Da die *Testcase*-Modellierung mit dem *Workflow-Designer* des *ODF's* integriert ist, muss dem *OTX-Designer* entsprechende Funktionen bereitgestellt werden.

Die Klasse `Test.Control.TestFramework` stellt diesen Connector dar, der diese Funktionen offen legt. In dem Klassendiagramm von `Test.Control` können wir die gelisteten Funktionen besehen. Betrachten wir einige dieser Funktionen. Durch die Methode `OpenTestProject` werden alle Testdaten für das aktuelle *OTX-Projekt* geladen. Nachdem beispielsweise ein *OTX*-Ablauf geöffnet wurde, kann über ein Kontextmenu im *Workflow-Designer* ein neuer *Testcase* durch die Methode `CreateNewTestCase` erzeugt werden. Anschließend könnte etwa eine *Assertion* an eine Aktivität durch `AddAssertion` gebunden werden. Die folgende Abbildung veranschaulicht den Prozess der Erstellung und Bindung einer *Assertion* an eine Aktivität.

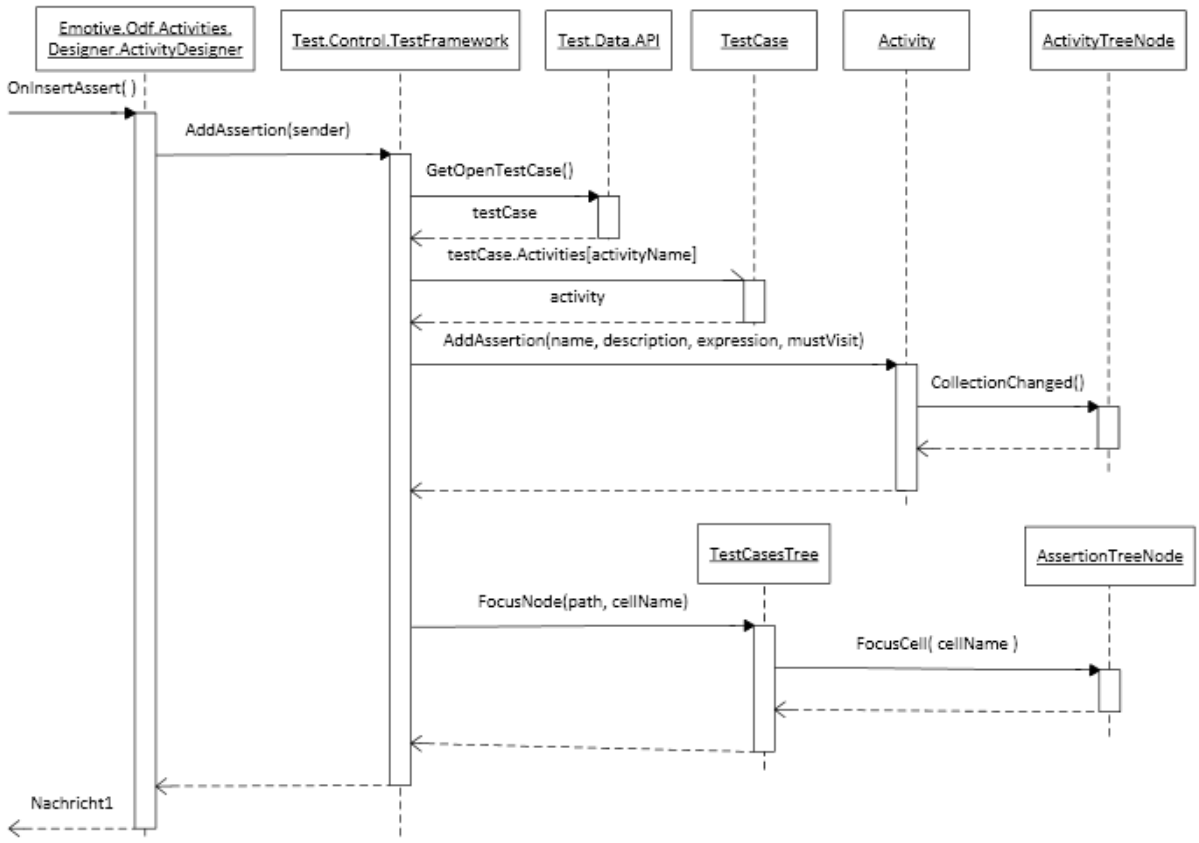

Abb. 34: UML SequenzDiagramm, AddAssertion

Einige erwähnenswerte Schritte werden erläutert:

1. Der `ActivityDesigner` des *ODF* ruft die `AddAssertion`-Methode auf. Die Prozessverarbeitung obliegt von hier an dem *Test Framework*.
2. Es kann zu einem Workflow mehrere Testfälle geben. `GetOpenTestCase`( ) liefert den aktuell geöffneten *Testcase*.
3. `testCase.Activities[activityName]` liefert die gesuchte Aktivität durch den Zugriff auf die `ObservableKeyedCollection` über den *Namen* der Aktivität.
4. Nachdem eine *Assertion* zur Aktivität hinzugefügt wurde, müssen durch ein `CollectionChanged()` *Event* die betroffenen *User Interfaces* (in unserem Fall: `ActivityTreenode`) benachrichtigt werden.
5. Nachdem das User Interface die Aktualisierung vorgenommen hat, gibt es einen neuen `AssertionTreenode`, der sofort den Fokus erhalten soll, damit der Benutzer die Testdaten der neu erstellten *Assertion* festlegt.

4.3.2. Test-Laufzeitumgebung (Runtime)

Die Laufzeitumgebung des *Test Frameworks* implementiert das Umfeld und die Logik für die automatisierte Ausführung von gesammelten Testfällen bzw. Testsuiten. Nachdem durch das User Interface die auszuführenden Tests ausgewählt und markiert wurden, werden sie in einem *Testrun* (Testlauf) zusammengefasst und der Laufzeitumgebung zur Ausführung übergeben.

Die Ausführung eines Testlaufes muss in einem neuen getrennten *Thread* gestartet werden, da eine Live-Aktualisierung der Testergebnisse im User-Interface sichtbar sein soll und die Ausführung den Rest der Applikation (*ODF*) nicht beeinträchtigen soll. Vielmehr sollte es dem Anwender möglich sein nebenher weiterzuarbeiten. Des Weiteren ist es nötig den Testlauf in einer neuen *App-Domain* zu starten, weil es nicht möglich ist, einzelne Klassen oder *Assemblies* zu entladen, sondern nur Ganze *App-Domains*. Da für jeden *Testcase* der C#-Code des entsprechenden *OTX*-Ablaufs dynamisch generiert wird und zur Ausführung als eine *Assembly* geladen werden muss, würde dies auf lange Sicht zur ungenutzten Belegung großer Mengen an Arbeitsspeicher führen.

Nach der Code-Erzeugung und einigen Initialisierungsprozessen führt die Laufzeitumgebung nacheinander alle enthaltenen *Testsuites* bzw. *Testcases* aus. Für die Ausführung der *Testcases* hält die Laufzeitumgebung die nötigen Testdaten bereit, sowie einige Methoden zur Prüfung von *Assertions*, Ausgabeparametern und *must visit*-Aktivitäten. Darüber hinaus muss die Ausführungslogik auch die Behandlung von *Exceptions* berücksichtigen, die vom *OTX*-Ablauf nicht behandelt wurden. Durch die bereitgestellten *Mock-Objekte* können Prozeduraufrufe umgelenkt und durch statische Testdaten simuliert werden. In der Fachsprache spricht man auch von einem *Test-Harness* (Test-Geschirr), der all diese Funktionen und nötigen Testdaten kapselt und bereitstellt. Die Klasse `TestCaseHarness` implementiert eine solche eben genannte Komponente und ist somit das Herzstück des Test-Laufzeitsystems. In ***Abb. 35:UML-Sequenzdiagramm, Runtime*** wird der grobe Ablauf eines Testlaufs im Laufzeitsystem dargestellt.

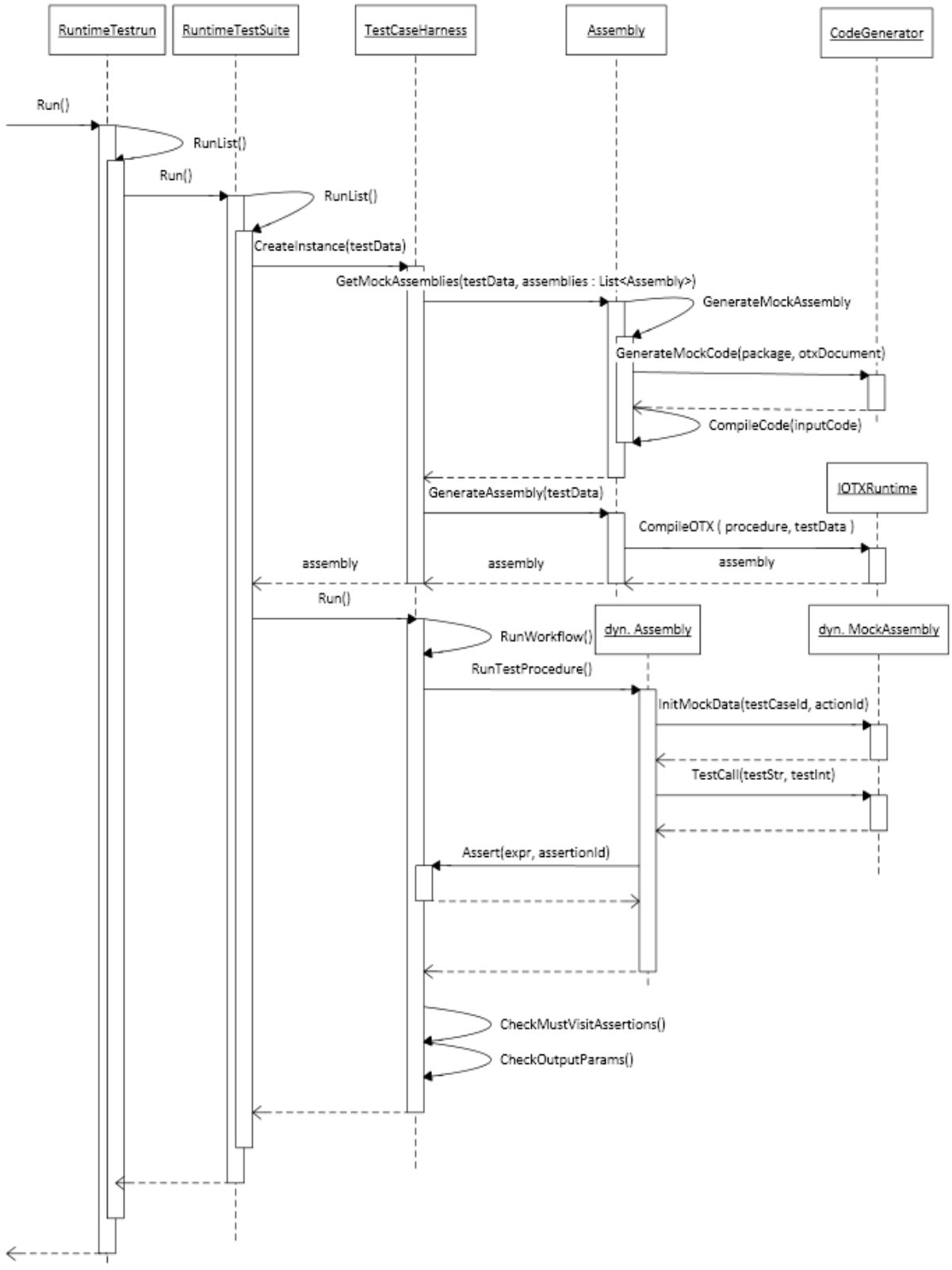

Abb. 35: UML-Sequenzdiagramm, Runtime

Wir können die Laufzeit der *TestCaseHarness* in drei Phasen aufteilen:

1. **Pre-Run:** Erstellung, Initialisierung und Code-Generierung
2. **Run:** Tatsächliche Ausführung des Testablaufs
3. **Post-Run:** Prüfung der Testdaten

1. Phase: Pre-Run

Die Erstellung des *TestCaseHarness* erfolgt durch die statische Methode `CreateInstance`. Diese ruft einige Initialisierungsmethoden auf. Anschließend wird der C#-Code für den *OTX*-Ablauf inklusive der *Assertion-, Event-, Exception-*, und *ProcedureCall-* Codezusätze sowie der Code für die *Mock-Objekte* generiert – Zu diesem Zweck werden die Methoden `GetMockAssemblies` und `GenerateAssembly` aufgerufen. Die Code-Generierung wird gesondert im ***Abschnitt 4.3.3: Code-Generierung*** behandelt.

2. Phase: Run

Nach der Initialisierung und Code-Generierung kann der Testablauf ausgeführt werden. Da der auszuführende C#-Code erst zur Laufzeit dynamisch generiert und geladen wird, muss über *Reflection*-Methoden des Namespaces `System.Reflection` auf die dynamisch generierte und geladene *Assembly* zugegriffen werden. Durch den von *dotNET*-Framework bereitgestellten *Reflection*-Mechanismus lassen sich Informationen über Methoden, Felder, etc. von Klassen zur Laufzeit abrufen. Außerdem ist es möglich mit der Kenntnis über die Struktur der Klasse auf Methoden und Felder, etc. zuzugreifen***. In Code Listing 1: RunWorkflow*** wird über Reflection die Methode `RunTestProcedure` der generierten *Assembly* aufgerufen.

```
private void RunWorkflow()
{
    Object workflow = WorkflowAssembly.CreateInstance(TestCaseData.WorkflowRefTokens.Packa
        BindingFlags.Public, null, null, System.Globalization.CultureInfo.CurrentCulture,

    workflow.GetType().GetMethod("RunTestProcedure").Invoke(workflow, new object[] { });
}
```

Code Listing 1: RunWorkflow

Bei der Codeerzeugung wird immer die Methode mit dem Namen `RunTestProcedure` generiert, die sozusagen den Einstiegspunkt für die *Assembly* darstellt, und die die eigentliche Prozedur – d.h. den zu testenden *OTX*-Ablauf - mit den spezifizierten Test-Parametern aufruft.

Steht während der Ausführung des *OTX*-Ablaufs die Prüfung einer *Assertion* an, wird die Methode `Assert` des *TestCaseHarness* mit dem *Boolean*-Ausdruck und einer *AssertionId* aufgerufen.
Die *AssertionId* wird benötigt, um die entsprechende *Assertion* im Datenmodell zu referenzieren und das Ergebnis der Auswertung festzuhalten. Das *User Interface* wird als *Event-Listener* (Beobachter) der *Assertion* automatisch aktualisiert.
Falls der *Boolean*-Ausdruck als '*false*' ausgewertet wird, wird eine *TFException* (*Test Framework - Exception*) geworfen, um die weitere Ausführung des aktuellen *OTX*-Ablaufs zu unterbrechen. Das Framework fängt die geworfene *TFException* ab und fährt mit dem nächsten *Testcase* fort. Das Code-Listing der `Assert`-Methode ist in ***Code Listing 2: Assert*** zu sehen.

```
public void Assert(bool expr, String assertionId)
{
    Emotive.Odf.Test.Data.Assertion assertion = assertions[assertionId];
    assertion.SetResultString();
    if (assertion.IsSkip)
    {
        return;
    }

    if (expr)
    {
        assertion.TestResults.Result = ResultEnum.success;
        assertion.TestResults.NotifyPropertyChanged(TestResults.PropertyNames.Result.ToString());
    }
    else
    {
        assertion.TestResults.Result = ResultEnum.fail;
        assertion.TestResults.NotifyPropertyChanged(TestResults.PropertyNames.Result.ToString());

        //Set Results for Parent Activity
        ((TestResultBase)assertion.Parent).TestResults.Result = ResultEnum.fail;
        ((TestResultBase)assertion.Parent).TestResults.ResultString = assertion.TestResults.Result
        ((TestResultBase)assertion.Parent).TestResults.NotifyPropertyChanged(TestResults.PropertyN

        throw new TFException(assertion.TestResults.ResultString);
    }
}
```

Code Listing 2: Assert

Kommt es bei der Ausführung des OTX-Ablaufs zu einem Prozeduraufruf, so wird statt eines anderen OTX-Ablaufs eine Simulation durch eine generierte Prozedur im Mock-Objekt ausgeführt. Lediglich die Rückgabeparameter werden mit den Testdaten belegt – mehr tut diese Mock-Prozedur nicht. ***Code Listing 5: Generated Mock C#-Code (siehe S.73)*** stellt eine solche Mock-Prozedur dar.

3. Phase: Post-Run

Nach der Ausführung des *OTX*-Ablaufs müssen die Ausgabeparameter der Prozedur geprüft werden. Die Methode `CheckOuputParams` führt hierzu einen Vergleich der Ausgabeparameter mit den im Testcase spezifizierten, erwarteten Werten durch.

Zum Schluss werden noch die *must visit*-Aktivitäten daraufhin geprüft, ob sie tatsächlich ausgeführt wurden. *must visit*-Aktivitäten werden, wie erwähnt, durch spezielle an sie gebundene *must visit-Assertions* realisiert, die immer '*true*' auswerten. Wie der Name schon suggeriert sind dies *Assertion*s, die besucht bzw. ausgeführt werden müssen. Durch die Methode *CheckMustVisitAssertions* prüft das Framework nach kompletter Ausführung des *OTX*-Ablaufs, ob alle *must visit*-Assertions tatsächlich ausgeführt wurden und wirft bei negativer Auswertung einen Fehler.

Exception-Handling

Das *TestcaseHarness* muss außerdem für die Behandlung bzw. das Abfangen von Ausnahmefehlern sorgen. Ein unbehandelter Ausnahmefehler darf nicht die Ausführung des *Testruns* bzw. andere *Testcases* unterbrechen. In ***Code Listing 3: Exception*** Handling ist zu sehen, wie dies durch einen *Try-Catch-Block* realisiert wird. Die Ausnahmefehler werden abgefangen (*catch*) und die entsprechenden Fehlerinformationen werden in den Test-Ergebnissen (*TestResults*) festgehalten.

```
Initialize();
try
{
    RunWorkflow();
    CheckMustVisitAssertions();
    CheckOutputParams();
}
catch (TFException ex)
{
    TestResults.Result = ResultEnum.fail;
    TestResults.ResultString = ex.Message;
    TestResults.NotifyPropertyChanged(TestResults.PropertyNames.Result.ToString());
    return;
}
catch (Exception ex)
{
    if (
            ex is Emotive.Otx.Data.UserException
        || ex is Emotive.Otx.Data.OutOfBoundsException
        || ex is Emotive.Otx.Data.TypeMismatchException
        || ex is Emotive.Otx.Data.ArithmeticException
        || ex is Emotive.Otx.Data.InvalidReferenceException
        || ( ex.InnerException != null || ex.InnerException is TFException )
        )
    {
        TestResults.Result = ResultEnum.fail;
        TestResults.ResultString = ex.InnerException.Message;
    }
    else
    {
        TestResults.Result = ResultEnum.error;
        TestResults.ResultString = ex.Message;
    }

    TestResults.NotifyPropertyChanged(TestResults.PropertyNames.Result.ToString());
    return;
}
```

Code Listing 3: Exception Handling

4.3.3. Code-Generierung

Um während der Laufzeit aus dem *OTX*-Dokument und den Testdaten dynamisch Code zu generieren werden die vom *dotNET*-Framework unter dem Namespace `System.CodeDOM` bereitgestellten Klassen genutzt.

Mit Hilfe von *CodeDOM* lässt sich durch verschiedene Methoden ein Code-Baum erzeugen, dessen Knoten und Blätter verschiedene Code-Elemente darstellen und letztendlich den gewünschten Programm-Code wiederspiegeln. Ist der Code-Baum vollständig aufgebaut, kann mit Hilfe eines Graphen-Durchlaufs der Programm-Code erzeugt werden. Da dieser Baum quasi ein generischer Repräsentant des erwünschten Codes ist, ist es mit *CodeDOM* möglich Code für verschiedene Sprachen erzeugen zu lassen. Mit einer Reihe von *CodeDOM*-Methoden können alle möglichen Sprachelemente erzeugt werden. In ***Code Listing 4: CodeDOM, Assignment*** etwa sehen wir ein Code-Sample, um beispielsweise ein *Assignment-Statement* für ein *Mock-Objekt* zu erzeugen.

```
CodeAssignStatement assign = new CodeAssignStatement();
assign.Left = new CodeVariableReferenceExpression(name);
assign.Right = new CodeCastExpression(type, new CodePropertyReferenceExpression(
                                        new CodeIndexerExpression(
                                            new CodePropertyReferenceExpression(
                                                new CodeIndexerExpression(
                                                    new CodePropertyReferenceExpression(
                                                        new CodePropertyReferenceExpression(
                                                            new CodeThisReferenceExpression()
                                                            , "TestCaseData")
                                                        , "MocksDic")
                                                    , new CodePropertyReferenceExpression(
                                                        new CodeThisReferenceExpression()
                                                        , "ActionId"))
                                                , "OutParametersDic")
                                            , new CodePrimitiveExpression(name))
                                        ,"TypedValue") );
```

Code Listing 4: CodeDOM, Assignment

Der aus diesem Abschnitt generierte Code entspricht einem *Assign-Statement* in ***Code Listing 5: Generated Mock C#-Code)***, welches eine Prozedur eines generierten *Mock-Objektes* darstellt.

```
public void TestCall(long TestCallIn, out long TestCallOut, ref string TestCallInOut)
{
    TestCallOut = ((long)(this.TestCaseData.MocksDic[this.ActionId].
        OutParametersDic["TestCallOut"].TypedValue));

    TestCallInOut = ((string)(this.TestCaseData.MocksDic[this.ActionId].
        OutParametersDic["TestCallInOut"].TypedValue));
}
```

Code Listing 5: Generated Mock C#-Code

In ***Code Listing 6: Generated Otx C#-Code*** ist der generierte *OTX*-Code eines sehr simplen Ablaufs zu sehen, welcher eine einzige Aktivität, nämlich einen Prozeduraufruf, ausführt. Der *OTX*-Ablauf tut demnach weiter nichts als nur die Prozedur `Testcall` aufzurufen. Die Testdaten des *Testcases* spezifizieren zusätzlich eine *Assertion* und ein *Throw-Element* mit dem Typ `Emotive.Otx.Data.ArithmeticException`.

Nach dem Prozeduraufruf `TestCall` ist dementsprechend die `Assert`-Anweisung gefolgt von einer `throws` Anweisung zu sehen. Wir sehen: Ein *Throw-Element* wird also durch eine C#-throws Anweisung implementiert.

Um den eigentlichen *OTX*-Ablauf `Workflow1Procedure` nun auszuführen, wird dieser vom Test Framework nicht direkt aufgerufen, sondern über die Methode `RunTestProcedure`, die vorher und nachher noch für die Belegung und Rückbelegung der Eingabe- und Ausgabe- Parameter sorgt.

Die Erzeugung des Programmcodes für die *Mock-Objekte* wird durch die Methode `GenerateMockCode` der Klasse `Test.Control.CodeGenerator` angestoßen. Die Codezusätze für das Testen eines *OTX*-Ablaufs werden integriert während der Codeerzeugung eines *OTX*-Dokumentes im *ODF* mit eingebunden.

```
public static void RunTestProcedure()
{
    // Get TestCase Harness, that holds TestFramework Data and Methods
    Emotive.Odf.Test.Runtime.TestCaseHarness testCaseHarness =
        Emotive.Odf.Test.Runtime.TestCaseHarness.HarnessMap["62aa2274-dd7d-4beb-b4ad-51e6f47c4842"];
    //
    //
    string InParameterDeclaration1;
    InParameterDeclaration1 = ((string)(testCaseHarness.InParameters["InParameterDeclaration1"].TypedValue));
    //
    long OutParameterDeclaration1;
    //
    string InOutParameterDeclaration1;
    InOutParameterDeclaration1 = ((string)(testCaseHarness.InParameters["InOutParameterDeclaration1"].TypedValue));
    //
    // Call Test Procedure
    Package.ODFProject3.Workflow1Procedure(
        InParameterDeclaration1, out OutParameterDeclaration1, ref InOutParameterDeclaration1);
    //
    testCaseHarness.OutParameters["OutParameterDeclaration1"].TypedValue = OutParameterDeclaration1;
    testCaseHarness.OutParameters["InOutParameterDeclaration1"].TypedValue = InOutParameterDeclaration1;
}

public static void Workflow1Procedure(string InParameterDeclaration1, out long OutParameterDeclaration1, ref string InOutParameterDeclaration1)
{

    // Action - ProcedureCall0 - id_285edcae_9f0c_466a_9a58_b164935324f5

    procedureCall = ((Package.ODFProject3.TestCall)
            (Emotive.Odf.Test.Runtime.Package.ODFProject3.Mock["62aa2274-dd7d-4beb-b4ad-51e6f47c4842", "ProcedureCall0"].TestCall));
    if ((procedureCall != null))
    {
        ((Package.ODFProject3.TestCall)(procedureCall))(5, out TestCallOutInt, ref InOutParameterDeclaration1);
    }

    Emotive.Odf.Test.Runtime.Assert.IsTrue(
        InParameterDeclaration1 == InOutParameterDeclaration1, "62aa2274-dd7d-4beb-b4ad-51e6f47c4842.45cbce0b-928c-4881-b234-438d9bb3fff0");
    throw new Emotive.Otx.Data.ArithmeticException("Test Exception");
}
```

Code Listing 6: Generated Otx C#-Code

4.4. Test.GUI

Die nachfolgende Abbildung stellt das Klassendiagramm von `Test.GUI` dar.

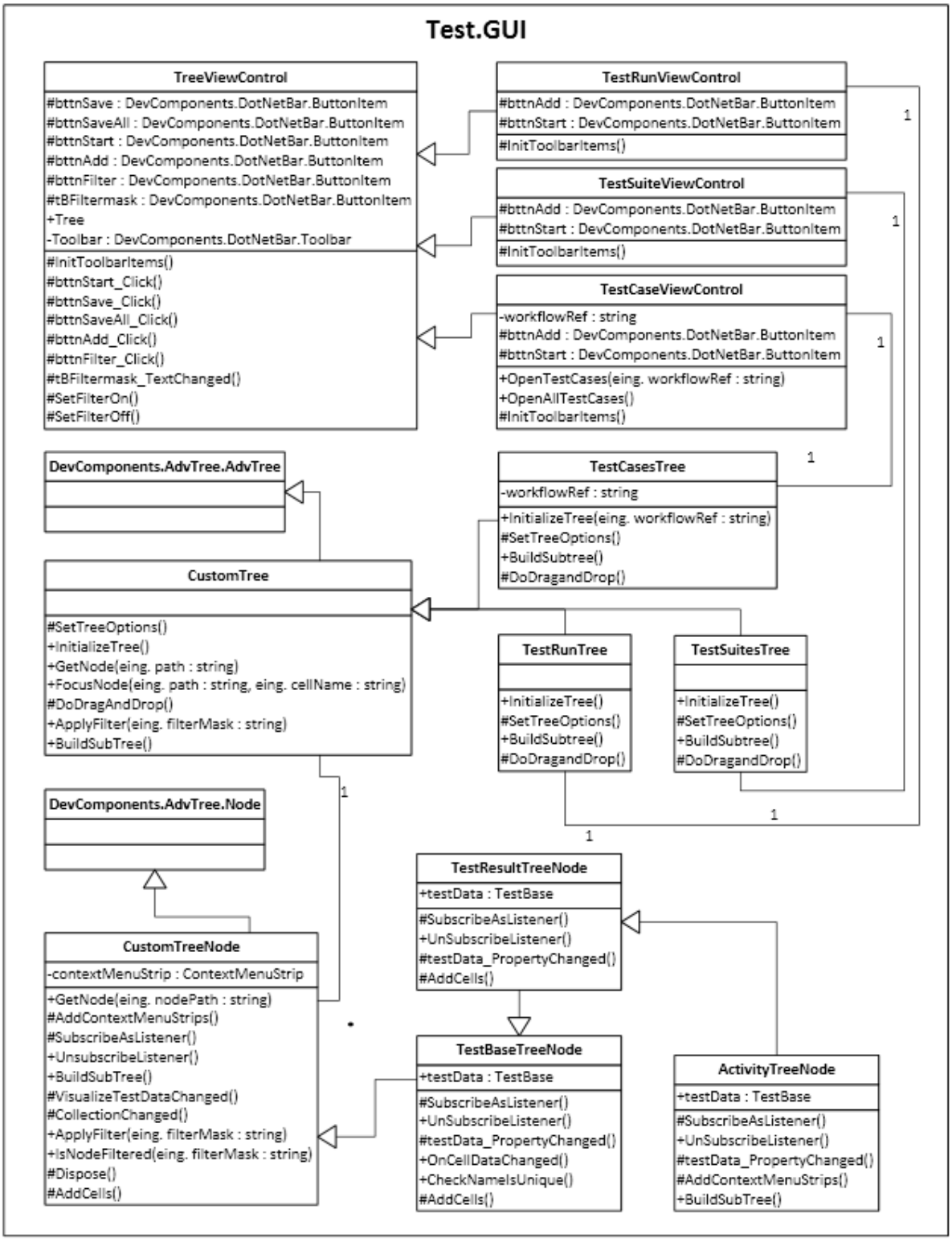

Abb. 36: UML-Klassendiagramm, Test.GUI

4.4.1. Workflow/Testcase-Designer

Um dem Anwender eine einfache und intuitive *Testcase*-Modellierung zu ermöglichen, ist letztere mit dem *Workflow-Designer* integriert bzw. verknüpft. *Assertions* können beispielsweise direkt im Designer durch ein Kontext-Menü einer Aktivität hinzugefügt werden.

Die Testdaten werden über Methoden der Klasse `Test.Control.TestFramework` (alias *ODFConnector*) hinzugefügt. Zusätzlich zum *Workflow-Designer* gibt es noch das *TestCaseViewControl* zur detaillierten Bearbeitung der Testdaten. In diesem *Control* werden alle Testfälle zum geöffneten *OTX*-Ablauf in einer Baumstruktur angezeigt. Jeder Testfall-Knoten hat eine Unterbaumstruktur zur Anzeige und Bearbeitung von Eingangsparametern, erwarteten Ausgangsparametern, Rückgabeparametern von Prozeduraufrufen, *Assertions*, etc.

Die Baumstruktur `TestCasesTree` verwendet die `AdvTree`-Komponente der Firma *DevComponents*, die zusätzliche Funktionen gegenüber dem standardmäßigen *TreeView-Control* von *dotNet* bietet. Für unseren *Testcase-Designer* werden vor allem die zusätzlichen Spalten zur Darstellung und Editierung der Testdaten verwendet. Die Baumstruktur besteht hauptsächlich aus Knoten die allesamt von `DevComponents.AdvTree.Node` ableiten müssen. Um eine einfache und ausreichende Individualisierung von Knoten verschiedener Art zu ermöglichen, wird für jede Art von Knoten eine eigene Klasse implementiert, die von `DevComponents.AdvTree.Node` ableitet. Alle Knoten die Testdaten enthalten sollen, sind von `TestBaseTreeNode` abgeleitet, welches selbstverständlich wiederum von `DevComponents.AdvTree.Node` abgeleitet ist und standardmäßig Funktionen zur Anzeige und Bearbeitung zur Verfügung stellt. In dem Klassendiagramm für `Test.GUI` (siehe oben) können wir einige Knoten-Klassen sehen. Aufgrund der großen Anzahl der verschiedenen Arten von Knoten (z.B. `TestcaseNode`, `ParameterNode`, `AssertionNode`, `ActivityNode`, etc…) wird nur ein Teil von ihnen abgebildet. Tatsächlich gibt es noch etliche weitere Klassen, die von `TestBaseTreeNode` bzw. von `TestResultBaseNode` ableiten. In ***Abb. 37: UML Sequenzdiagramm, OpenTestCases*** ist der Ablauf für den Aufbau eines *TestCasesTree* dargestellt. In diesem Beispiel werden alle *Testcases* eines *OTX*-Ablaufs samt ihrer Unterbaumstruktur aufgebaut. Ein `TestCaseTreeNode` stellt je einen *Testcase* dar.

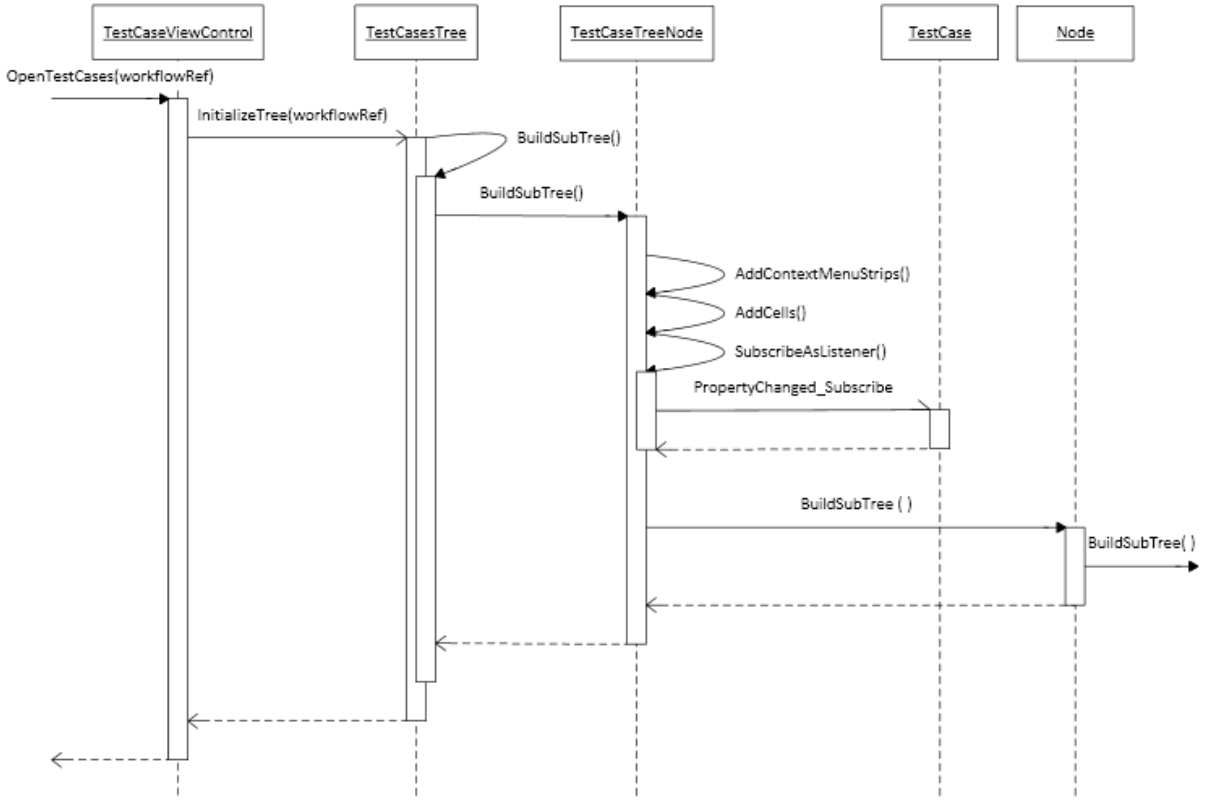

Abb. 37: UML Sequenzdiagramm, OpenTestCases

Durch die Methode `BuildSubTree` wird rekursiv der Aufbau der ganzen Unterbaumstruktur ausgeführt. Es werden dabei noch einige Initialisierungsprozesse für das Laden des Kontextmenüs und der Testdaten gestartet. Wichtig ist auch die Methode `SubscribeAsListener`, welche die Beobachterfunktion des Knotens auf die Testdaten aktiviert und das erwähnte *Observer-Pattern* implementiert. Sobald Testdaten aktualisiert werden, werden die Knoten im User Interface ebenso aktualisiert. Es ist hierbei äußerst wichtig für ein `Subscribe` auch eine entsprechende `Unsubscribe`-Funktion zu implementieren und diese aufzurufen sobald das Objekt nicht mehr genutzt wird. Da eine *Subscription* immer bedeutet, dass ein Verweis vom beobachteten Objekt auf den Beobachter erzeugt wird, würde das *Beobachter-Objekt* nie vom *Garbage-Collector* gesammelt und zerstört werden, solange dieser Verweis noch besteht. Ein „geschlampertes“ *Unsubscribe* führt also dazu, dass ein nicht benutztes Objekt weiterhin Speicher belegt.

4.4.2. Testsuite Manager

Um *Testsuites* komfortabel zusammenzustellen gibt es den *Testsuite Manager*, der aus einem *TestcaseViewControl* und einem *TestsuiteViewControl* besteht. Das *TestsuiteViewControl* enthält einen *TestsuiteTree* welches ähnlich aufgebaut ist wie ein *TestcaseTree* und hier nicht näher

vorgestellt wird. Das *TestsuiteViewControl* stellt alle projektweit vorhandenen *Testsuites* dar und stellt Funktionen zur Verfügung *Testsuites* zu erstellen, zu verwalten und zu löschen.
Über das *TestcaseViewControl* lassen sich *Testcases* schnell und einfach finden und können dann per *Drag&Drop* einer *Testsuite* hinzugefügt werden. Das *TestcaseViewControl* hält dazu eine Filtereingabe bereit, womit projektweit nach verschiedenen Kriterien gefiltert werden kann. So ließe sich etwa die Anzeige anpassen, sodass nur alle *Testcases* einer bestimmten *Otx-Package* oder eines bestimmten *Otx-Documents* erscheinen. Vom *Testsuite Manager* aus können beliebige *Testsuites* und *Testcases* ausgewählt und in einem Testlauf ausgeführt werden.

4.4.3. Testrun - Darstellung der Test-Ergebnisse

Der Testlauf bzw. *Testrun* wird ebenfalls durch eine Baumstruktur implementiert, da dies einen sehr übersichtlichen und effektiven Überblick über den Testlauf ermöglicht. Die *TestRun-TestSuite-Testcase-Assertion* Hierarchie lässt sich wunderbar durch einen Baum darstellen. Testergebnisse werden in einer zweiten Spalte festgehalten, wobei deren Darstellungslogik auch sehr gut in einer Baumstruktur umgesetzt werden kann. Sollte ein *Testcase* fehlschlagen, so gilt die *Testsuite* und weiter auch der *Testrun* als fehlgeschlagen. Dies kann in der Baumstruktur gut wiedergegeben werden und es ist wie gefordert auf einem Blick schnell ersichtlich, ob ein *Testrun* bzw. eine *Testsuite* erfolgreich war oder nicht. Es müssen nicht alle Informationen über alle *Testcases* angezeigt werden. Jene *Testcases* die ohnehin erfolgreich durchlaufen sind, interessieren bei einem Fehler-Test nicht. Will ein Anwender genauere Ergebnisse über eine *Testsuite* oder einem *Testcase* erhalten, so kann er durch die Baumstruktur navigieren.
Die Darstellung eines Testlaufs übernimmt das *TestRunViewControl*. Dieser beherbergt einen *TestrunTree*, der den gleichen Aufbau hat wie der *TestcaseTree*. Speziell zu erwähnen sind die *TestResultTreeNodes*, von denen alle Knoten des TestrunTrees abgeleitet sind, die Ergebnisse zurückliefern. Zum Beispiel ist eine *AssertionNode* ein *TestResultTreeNode* und hat als SubTyp eines *TestBaseTreeNode* selbstverständlich auch einen Verweis auf ein `Test.Data.TestBase`-Objekt, welches in der Tat sogar ein `Test.Data.TestResultBase`- bzw. ein `Test.Data.Assertion`-Objekt ist. Über den Verweis werden die Testdaten, aber auch Testergebnisse abgerufen und angezeigt. Die *Assertion-Node* ist als `TestBaseNode` standardmäßig Beobachter des *PropertyChanged*-Event des *Assertion-Objektes* und kann auf alle Datenänderungen des *Assertion-Objektes* reagieren. Während der Ausführung eines *Testcases* aktualisiert die Test-Laufzeitumgebung nach der Prüfung einer *Assertion* dessen Ergebnisse am *Assertion-Objekt* von `Test.Data`. Da der *AssertionNode* Beobachter ist, kann er die Testergebnisse sofort anpassen und der *TestRunTree* kann die Ergebnisse direkt anzeigen. Bei der Ausführung eines Testlaufes werden also nacheinander die Ergebnisse von *Assertions, Testcases*, etc. angezeigt. Um dies zu realisieren ist es jedoch nötig die Ausführung des Testlaufes in einem neuen *Thread* zu starten.

5. Evaluierung

5.1. Erstellung von Testfällen - Best Practices

5.1.1. Assertions

Betrachten wir für den Anfang einen einfachen Workflow, der die Geldausgabe an einem Bankautomaten steuert. Der Workflow bekommt die *PIN-Nummer* eines Benutzers als Parameterübergabe. Die Aktivität *PIN*-Prüfung prüft die *PIN*-Nummer auf ihre Gültigkeit und gibt als Antwort einen *Boolean*-Wert zurück. Sollte die Antwort positiv sein, wird der Zugriff auf das Konto gewährt, solange der Kontostand nicht negativ ist - andernfalls wird der Zugriff verweigert.

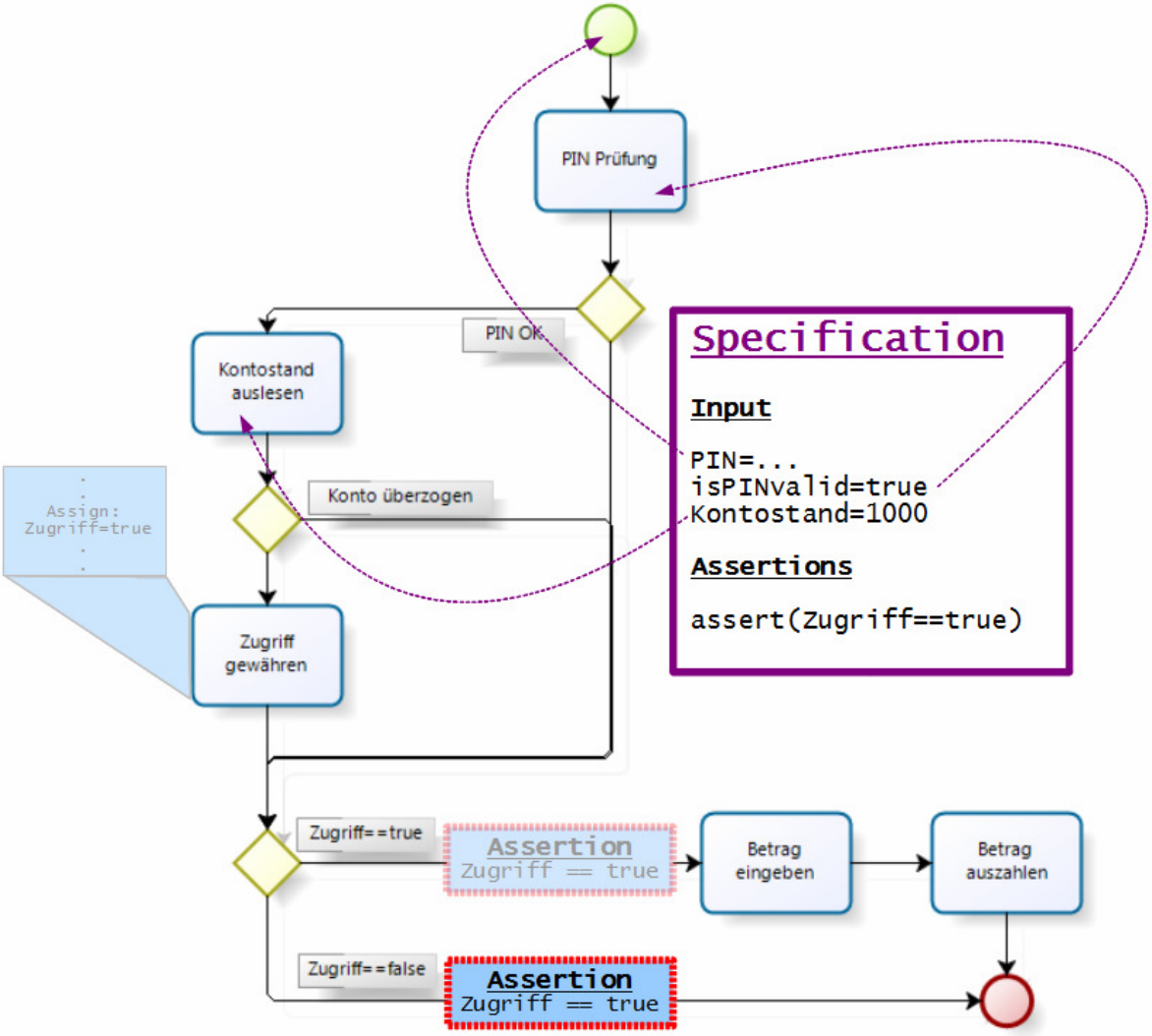

Abb. 38: Best Practice, Assertions 1

Um sicherzustellen, dass Geld nur ausgegeben wird, wenn der Zugriff gewährt wurde (Sollverhalten), würde man auf dem ersten Blick naiverweise die Zusicherung `assert( Zugriff==true )` auf den gleichen Pfad setzen, in dem das Geld ausgezahlt wird (siehe ausgegraute Assertion in obiger Abbildung). Logisch: Geld wird ausgezahlt, wenn `'Zugriff==true'` zugesichert wird.
Dass eine solche Zusicherung nicht sehr effektiv ist, sollte auf dem zweiten Blick ersichtlich sein: Der Pfad auf dem unsere Zusicherung `'Zugriff==true'` liegt, wird genau dann ausgeführt, wenn die *Condition* ohnehin dieselbe Bedingung, nämlich `'Zugriff==true',` positiv auswertet. Eine solche Zusicherung in so einem Kontext ist logisch korrekt und gilt immer, d.h. für jeden Testfall - hat somit aber keinen effektiven Nutzen.
Für einen effektiven Test sollte die gegenteilige Zusicherung `'assert(Zugriff==false)'` gesetzt werden. Nachfolgende ***Abb. 39: Best Practice, Assertions 2*** zeigt den Testfall, der das besprochene Sollverhalten des Workflows verifiziert. Wir bereiten sämtliche Input-Daten des Workflows auf ein solches Szenario vor, in dem der Zugriff - semantisch gesehen! - verweigert werden sollte. In unserem Beispiel wird eine falsche *PIN-Nummer* übergeben. Sollte nun wider Erwarten, aufgrund fehlerhafter Implementierung des Workflows, trotzdem der Pfad ausgeführt werden, in dem der Zugriff gewährt wurde und das Geld ausgezahlt wird, wird die Zusicherung `'assert( Zugriff==false)'` negativ ausgewertet und der Testfall schlägt somit fehl. Tatsächlich sichert der Testfall in obiger ***Abb. 38: Best Practice, Assertions 1*** ein anderes Sollverhalten des Workflows zu. Der Testfall sichert den komplementären Fall zu - nämlich, dass der Geldausgabe-Prozess ausgeführt wird, wenn der Zugriff gewährt werden soll.

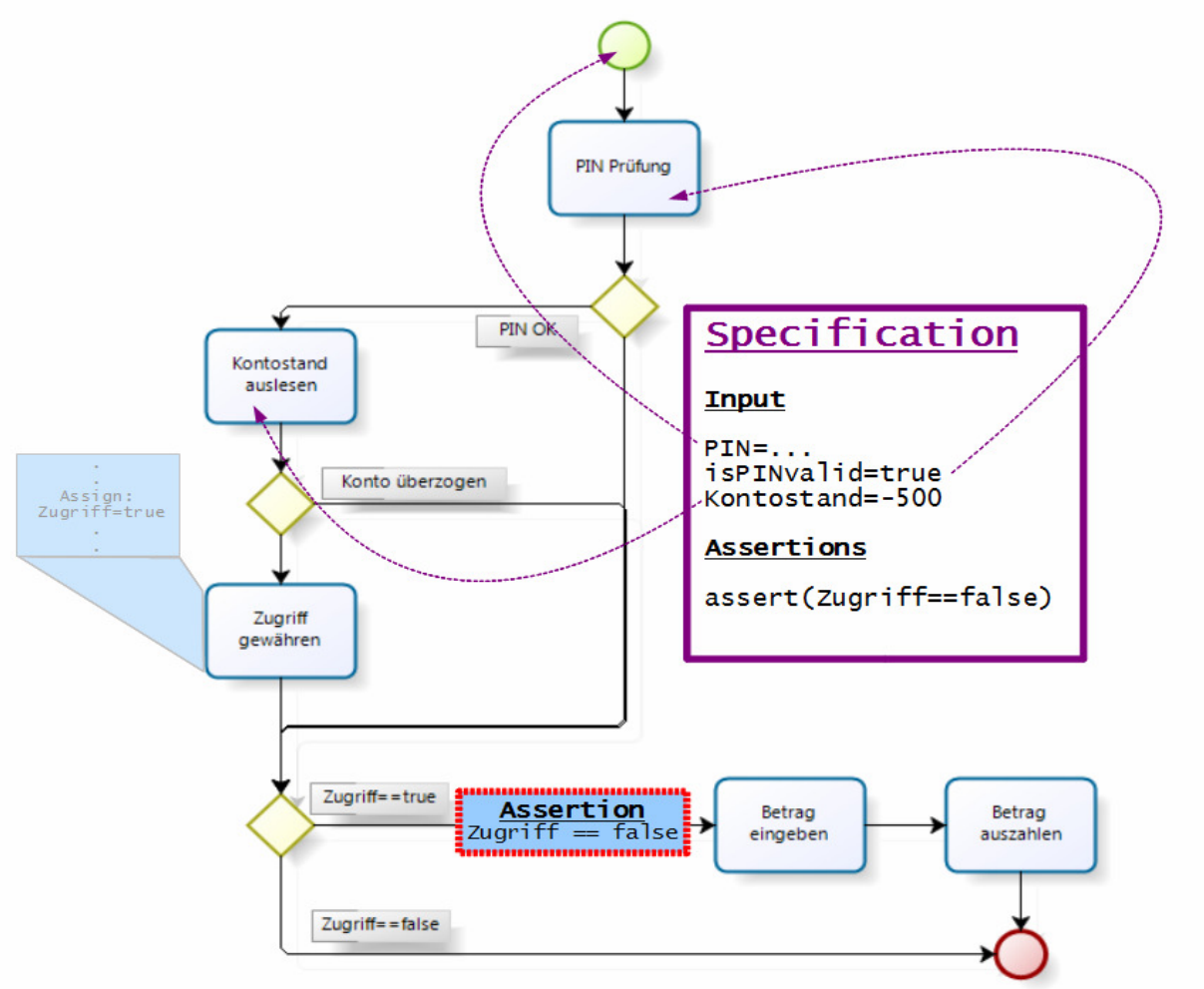

Abb. 39: Best Practice, Assertions 2

Ein anderer Ansatz die Korrektheit des Workflows zu testen, wäre eine Assertion zu stellen, die zusichert, dass die *PIN-Nummer* gültig ist, wenn Geld ausgezahlt wird. Nachfolgende Abbildung zeigt eine Testspezifikation (*Testfall A*), die ein solches Sollverhalten prüft.

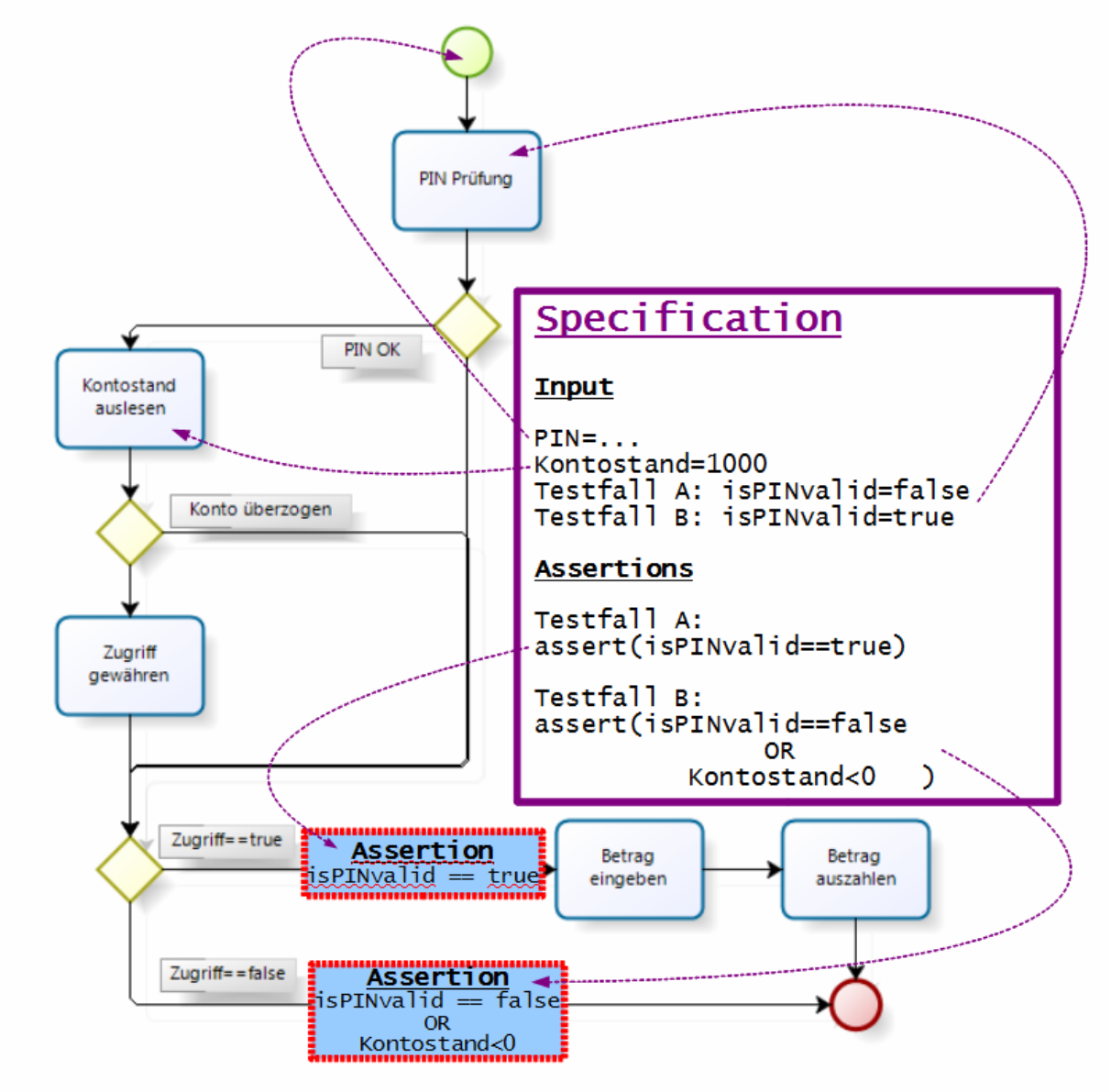

Abb. 40: Best Practice, Assertions 3

Die Assertion von *Testfall A* stellt sicher, dass die *PIN-Nummer* gültig ist, wenn Geld ausgezahlt wird. Wir testen dies, indem in der Testspezifikation die *PIN-Prüfung* einen negativen Wert für *'isPINvalid'* zurückgibt - sprich die *PIN-Nummer* ist ungültig. Wird der Zugriff von einem fehlerhaften Workflow trotz ungültiger PIN-Nummer gewährt, so schlägt der Testfall wegen der `Zusicherung 'isPINvalid == true'` auf dem folgendem Pfad fehl. Ähnlich dazu lässt sich der *Testfall B* auswerten, welcher zusichern soll, dass kein Geld ausgezahlt wird, wenn die *PIN-Nummer* gültig ist oder der Kontostand negativ ist. Betrachtet man die vorgestellten Testfälle genauer, so fällt auf, dass diese genau so spezifiziert wurden, dass *Assertions* auf Pfade liegen, die korrekterweise (d.h. die Semantik betreffend) nicht ausgeführt werden sollten; und ferner, dass diese *Assertions*, sollte einer dieser Pfade fälschlicherweise doch ausgeführt werden, aufgrund der Testspezifikation und des Ablaufs immer negativ ausgewertet werden.

Um die Modellierung von Testfällen zu vereinfachen, lassen sich in solchen und ähnlichen Szenarien diese *Assertions* einfach durch Zusicherungen der Art `'assert(false)'` ersetzen. Die Bedeutung der Zusicherung `'assert(false)'` ist jene, dass sie immer negativ ausgewertet wird und ein Testfall somit automatisch fehlschlägt, sobald ein Pfad mit einer solchen *Assertion* erreicht wird. Das solche *Assertion* durchaus sinnvoll sind, stellt nachfolgender Testfall dar.

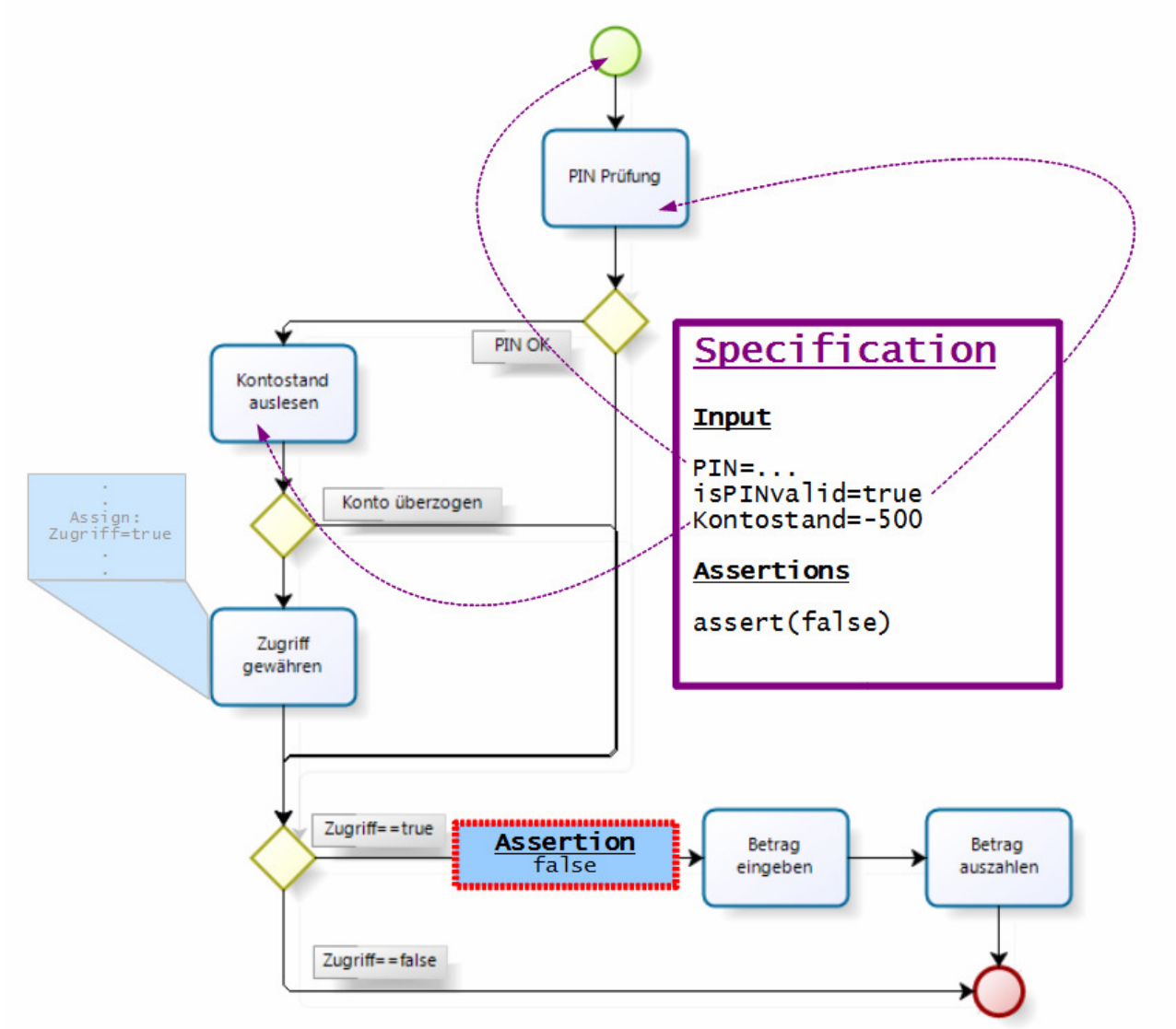

Abb. 41: Best Practice, Assertions 4

Es ist bemerkenswert, dass dieser Testfall viel leichter nachzuvollziehen ist, als die vorherigen Testfälle, obwohl er das gleiche Sollverhalten des Workflows überprüft. Der Testspezifikation nach sollte der Workflow den Geldausgabeprozess nicht ausführen. Sollte dies aufgrund von Fehlern doch passieren, so sorgt die Zusicherung `'assert(false)'` im selben Pfad für einen negativen Testausgang. Der andere Fall mit positivem Kontostand und gültiger *PIN-Nummer* muss die

Zusicherung `'assert(false)'` auf den unteren Pfad mit der Condition `'Zugriff==false'` setzen. Die folgende Abbildung fasst das Besprochene nochmal allgemein zusammen.

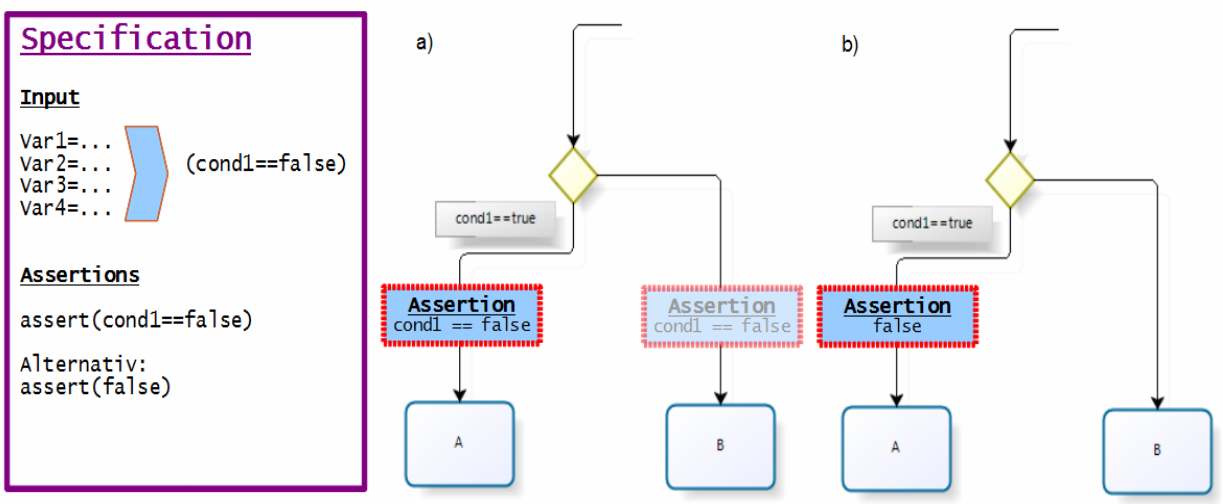

Abb. 42: Best Practice, Assertions 5

Wir sehen in dieser Abbildung nur einen Teilausschnitt eines Workflows, der die eingehenden Variablen `Var1`, `Var2`, `Var3`, `Var4` in einem beliebigen, uns nicht sichtbaren und nicht zu interessierenden, Berechnungsvorgang schon vorher verarbeitet hat.

Das zu prüfende Sollverhalten ist, dass *Aktivität A* nur dann ausgeführt wird, wenn `'cond1==true'` wahr ist. Wir modellieren dazu die komplementären Testfälle mit den nötigen Input-Daten, die die *Condition cond1* zu *'false'* auswerten sollen, um zu prüfen, dass in keinen dieser Testfälle die *Aktivität A* ausgeführt wird. Wir gehen nun davon aus, dass *cond1* zu *'false'* ausgewertet wird. Unabhängig davon, welche Verzweigung ausgeführt werden soll, müsste demnach die *Assertion* `'assert(cond1==false)'` halten. Wir können die *Assertion* `'assert(cond1==false)'` also an beiden Verzweigungen setzen, wie es unter Punkt a) in der Abbildung zu sehen ist. Die *Assertion* in der Verzweigung mit der *Condition* `'cond1==false'` (also die Verzweigung die zur *Aktivität B* führt) kann auch weggelassen werden, da die Zusicherung mit der *Condition* der Verzweigung übereinstimmt und somit immer erfüllt wird. Sollte in einem dieser Testfälle die Verzweigung der *Aktivität A* ausgeführt werden, wird die *Assertion* negativ ausgewertet, der Testfall schlägt fehl und wir wissen, dass sich ein Fehler im Workflow befindet.

Wir können von der *Assertion* in a), die, sofern sie geprüft wird, immer negativ ausgewertet wird, abstrahieren, und ebenso gut mit `'assert(false)'` ersetzen. Dieser Testfall ist unter b) zu sehen und ist völlig äquivalent zu dem Testfall unter a). Wie gehabt spezifizieren wir die Testfälle so, dass *Aktivität A* nicht ausgeführt wird und setzen dann eine `'assert(false)'` *Assertion* auf die Verzweigung mit der *Aktivität A*. Wir sichern in den Testfällen im Prinzip zu, dass der Workflow bestimmte Zweige nicht ausführt - in unserem Fall darf die Verzweigung mit der *Aktivität A* nicht ausgeführt werden.

Eine weitere Abstraktion lässt uns also feststellen, dass *Assertions* der Art `'assert(false)'` ein Test-Konzept für Workflows realisieren, welches durch Prüfung des durchlaufenden Pfades die Korrektheit des Workflows testet. Die Zusicherung `'assert(false)'` stellt sicher, dass der Pfad, auf dem die *Assertion* liegt, nicht ausgeführt wird. Wollten wir einen Workflow auf seine Korrektheit prüfen, indem für bestimmte Eingangsdaten des Workflows der erwünschte Kontrollfluss (oder Pfad) des Workflows mitspezifiziert wird, so kann dies durch *Assertions* der Art `'assert(false)'` erreicht werden, indem alle anderen Pfade, die nicht ausgeführt werden sollen, mit eben solchen *Assertions* zugesichert werden. In unserem Standardbeispiel der Kreditprüfung ließe sich beispielsweise durch zwei *Assertions* die Ausführung des rot markierten Pfades zusichern.

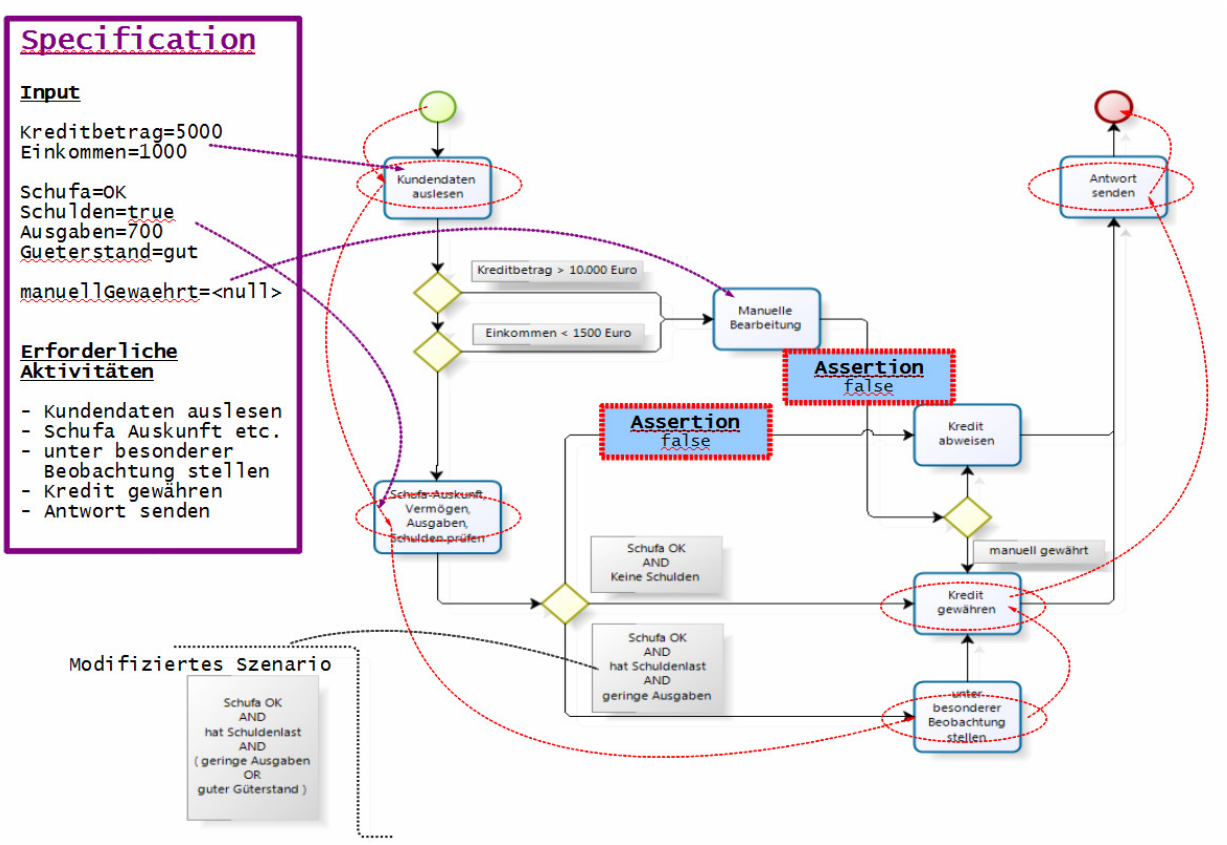

Abb. 43: Best Practice, Assertions 6

5.1.2. *must visit*-Aktivitäten

Es ist oft einfacher und weniger Arbeit, den Pfad anzugeben, der ausgeführt werden soll, als alle anderen Pfade durch *Assertions* auszuschließen. Es ist auch intuitiv und leichter überschaubar Aktivitäten anzugeben, die zu einem spezifischen Testfall ausgeführt werden müssen. In vielen Fällen ist es möglich schon durch die Angabe von wenigen Aktivitäten, die oft auch die Hauptfunktionen des Ablaufs darstellen, das korrekte Verhalten für einen Testfall sicherzustellen. Wie wir in den

nachfolgenden Beispielen sehen werden, gibt es auch da einpaar Dinge, die man sich bewusst machen und beachten sollte.

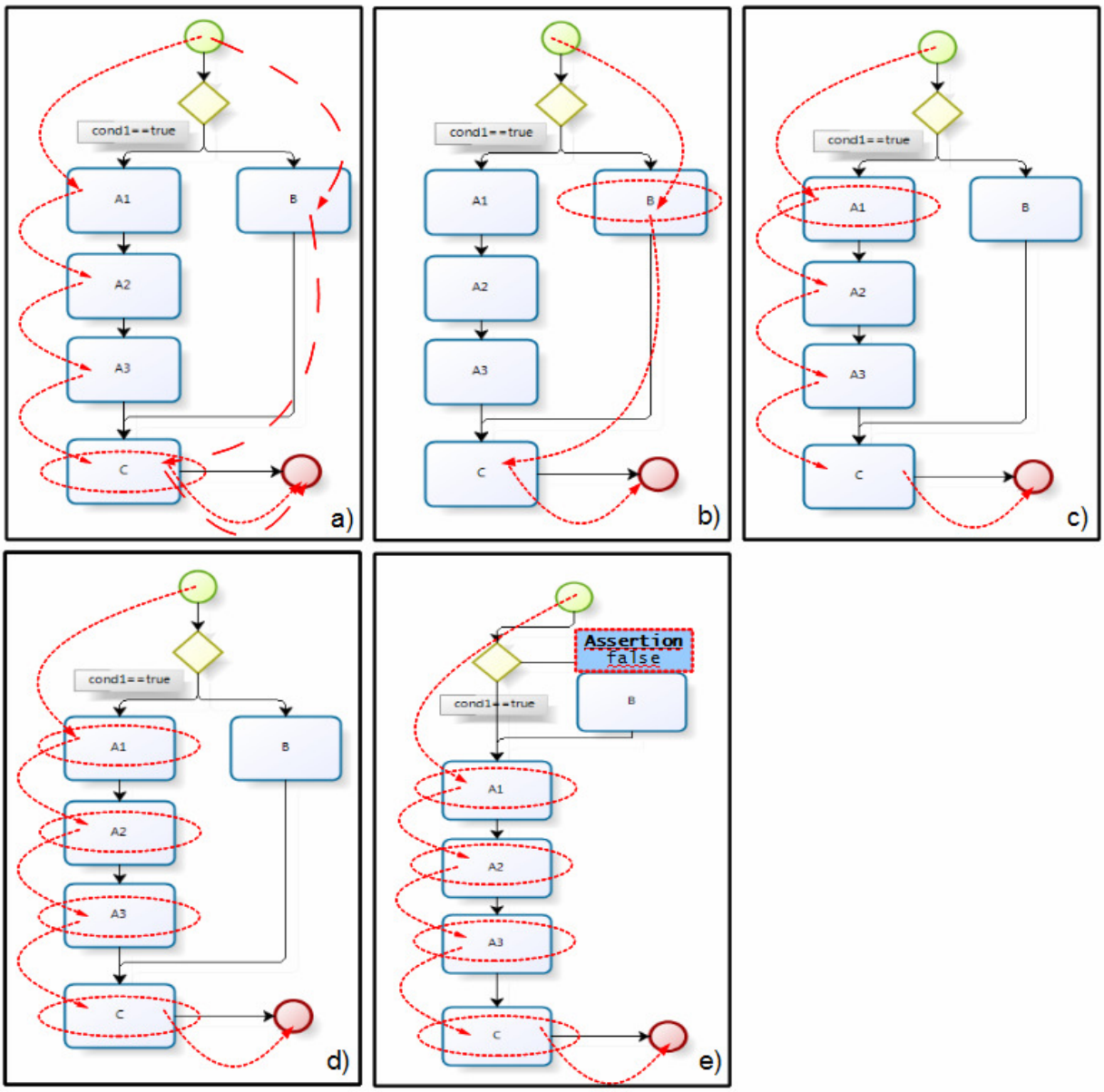

Abb. 44: Best Practice, Assertions 7

Im Testfall a) wird nur die *Aktivität C* als *must visit* spezifiziert. Der Pfad der durchlaufen werden kann, ist in diesem Fall nicht eindeutig. Will man einen eindeutigen Pfad festlegen, so muss wie in b) oder c) bei einer Verzweigung für mindestens einen Zweig mindestens eine Aktivität markiert sein. Man könnte auch den Weg wählen generell immer alle Aktivitäten eines Pfades mitzuspezifizieren, wie es in d) zu sehen ist. In den meisten Fällen ist der Pfad dann eindeutig bestimmt. Es gibt jedoch Ausnahmen wie wir in e) sehen können. Der zu durchlaufende Zweig nach dem *Gateway* hat keine markierbare Aktivität. Wir müssen zusätzlich durch eine *Assertion* den Pfad über *Aktivität B* ausschließen.

6. Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, ein Framework zu entwickeln und zu implementieren, welches geeignete Funktionalitäten bietet, um die mit *ODF* entwickelten *OTX*-Abläufe automatisiert zu testen.
Das vorgestellte *Test Framework* realisiert dazu ein *Unit-Testing*, das durch ein mit dem *ODF* integrierten *Testcase-Designer* spezifiziert wird. Die *Testcase-Modellierung* geschieht quasi „direkt" im Workflow selbst und ist dadurch intuitiv und leicht verständlich. Alle Testdaten werden in einem speziell für das Framework angepassten *XML*-Derivat gespeichert und werden somit in einem zukunftsträchtigen Austauschformat aufgehoben. Durch einen *Testsuite-Manager* können *Testsuites* und *Testcases* in einer hierarchischen Struktur organisiert werden und vom Test-Laufzeitsystem in einem Testlauf automatisiert ausgeführt werden. Diese Testläufe können zur späteren Abrufung von Testergebnissen gespeichert werden und zur Qualitätssicherung der Software auch wiederholt ausgeführt werden. Bei Konsistenzproblemen zwischen *OTX*-Daten und *Test-Daten* findet dabei eine automatische Synchronisation statt.
Bei der Modellierung des Datenmodells wurde darauf Wert gelegt geeignete Datenstrukturen zu verwenden, die große Datenmengen effizient verarbeiten können, da es unter Umständen eine sehr große Anzahl an *OTX*-Abläufen und daraus folgend eine noch größere Anzahl an *Testcases* zu verwalten gibt.
Allerdings werden noch alle projektweiten Testdaten gänzlich in den Arbeitsspeicher geladen. Es ist in Zukunft notwendig ein **Lazy-Loading** Konzept zu implementieren bzw. eventuell eine **Xml-Datenbank** einzubinden, um den Speicherplatz im Arbeitsspeicher nicht zu überanspruchen.
Bis zum Zeitpunkt des Verfassens dieser Arbeit wurde das *Event*-Konzept von *OTX* durch *ODF* noch nicht funktionalisiert. Erst nach der Vervollständigung der Implementierung von *Events* kann auch das *Test Framework* das Testen mit *Events* unterstützen. Wie dem auch sei, das Konzept zur Unterstützung von *Events* wurde bereits besprochen und kann nachträglich implementiert werden.
Für die Dokumentation von ausgeführten Testläufen wäre außerdem eine automatische Erzeugung eines ausdruckbaren **Test-Report**s in Form von *Html* oder ähnliches sehr vorteilhaft. Ein *Test-Report* könnte neben den Testergebnissen und diversen Statistiken den Anteil von getestetem Code anzeigen. Dazu müsste ein Konzept für die Bestimmung der **Code Coverage** entwickelt und implementiert werden.
Um die Testfallerstellung zu erleichtern und um eine gute *Code Coverage* zu erreichen, kann das Test Framework um die Funktion einer automatischen **Testcase Generation** erweitert werden. So können automatisiert Testfälle mit entsprechenden Testdaten erzeugt werden, die eine gute oder sogar vollständige *Code Coverage* bieten.
Zuletzt möchte ich noch den Bezug auf die **Formale Verifikation** von Software nehmen, der sicherlich noch Stoff für interessante Arbeiten aufbietet. Durch einige grundlegende Techniken der *formalen Verifikation* ist es möglich die Korrektheit von Programmen mathematisch zu beweisen.

Durch das *Hoare-Kalkül* bzw. die *Hoare-Regeln* können für einzelne Anweisungen und kombiniert auch für Anweisungsblöcke Zusicherungen getroffen werden, die eine bestimmte Funktion verifizieren. Diese Regeln können für *OTX-Sprachelemente* angepasst werden und zu einem *Hoare-System* zusammengefasst werden, sodass dadurch eine formale Verifizierung von *OTX*-Abläufen möglich wird.

Solche Zusicherungen könnten durch *Assertions* dem *OTX*-Ablauf angereichert werden und würden so die Funktion des Ablaufs prüfen. Ein mit solchen Regeln geführter Beweis über die Korrektheit einer Funktion eines *OTX*-Ablaufs würde die 100%ige Korrektheit eben dieser Funktion sicherstellen. Da ein solcher Beweis wie besprochen jedoch in den meisten Fällen nicht trivial bis quasi undurchführbar ist, hat dieser an sich interessante Ansatz nicht viel praktischen Nutzen.

Literaturverzeichnis

[1]: **DIJKSTRA, E.W.:** The Humble Programmer. Commun. ACM **15**: 859-866 (1972)

[2]: **HOFFMANN, DIRK W.:** Software-Qualität. Springer Berlin, Heidelberg, 2008

[3]: **LEYMANN, F.; ROLLER, D.; SCHMIDT, M.-T.:** Web services and business process management. IBM Systems Journal, Vol. 41, No.2, 2002

[4]: **MAYER, P.**: Design and Implementation of a Framework for Testing BPEL Compositions. Gottfried Wilhelm Leibniz Universität, Hannover, 2006

[5]: **WIENER, L.R.:** Digitales Verhängnis, Gefahren der Abhängigkeit von Computern und Programmen. Addison-Wesley, München, 1994

[6]: **WFMC:** Workflow: An Introduction. Workflow Management Coalition.

[7]: **LI, Z.; SUN, W.; JIANG Z. B.; ZHANG X.:** BPEL4WS Unit Testing: Framework and Implementation. Proceedings of the IEEE International Conference on Web Services (ICWS), 2005

[8]: **SUPKE, J.**: Diagnosesysteme im Automobil. Seminarunterlagen der Technischen Akademie Esslingen, 2008.

[9]: **LÜBKE, D.; SINGER L.; SALNIKOW A.:** Calculating BPEL Test Coverage through Instrumentation. Leibniz Universität, Hannover

[10]: **DONG, W.**: Test Case Generation Method for BPEL-based Testing. Chinese Academy of Science, Beijing, 2009

[11]: **ZAKARIA Z.; ATAN R.; AZIM A.; GHANI A.; SANI N. F.:** Unit Testing Approaches for BPEL: A Systematic Review. Asia-Pacific Software Engineering Conference, 2009

[12]: **LINK J.**: Unit-Tests mit Java. dpunkt.verlag GmbH, Heidelberg, 1.Auflage 2002

[13]: **SPILLNER A.; LINZ T.:** Basiswissen Softwaretest. dpunkt.verlag GmbH, Heidelberg, 2.Auflage 2004

[14]: **ALONSO, G.; CASATI, F.; KUNO, H.; MACHIRAJU, V.:** Web Services: Concepts, Architectures and Applications. Springer Berlin, 2004

[15]: **AALST, W. V. D.; HEE, K. V.:** Workflow Management, Models, Methods, and Systems. The MIT Press, Cambridge, Massachusetts London, England, 2002

[16]: **GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J., DESIGN PATTERNS**: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional; 1995

[17]: **LEYMANN, F.; ROLLER, D.:** Production workflow: concepts and techniques. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2000.

Internetquellen

[18]: **KANDEL, DUNJA**: Lars Thomsen über die Zukunft von RFID im Alltag. http://www.rfid-im-blick.de/200907131520/Sprechende-Joghurtbecher-und-intelligente-Spiegel.html (28.07.2011)

[19]: **PALMISANO, SAM J.**: A Smart Transportation System: Improving Mobility for the 21st Century. http://www.ibm.com/smarterplanet/us/en/transportation_systems/article/palmisano_itsa_speech.html (28.07.2011)

[20]: **STUTTGARTER NACHRICHTEN**: Köhler eröffnet Boschs Chipfabrik in Reutlingen. http://www.stuttgarter-nachrichten.de/inhalt.koehler-eroeffnet-boschs-chipfabrik-in-reutlingen.d6852d03-d64c-4157-80be-63b0fbdfd906.html (28.07.2011)

[21]: **SUPKE, J.:** OTX - Hintergrund & Motivation. http://www.emotive.de (28.07.2011)

[22]: **SUPKE, J.:** OTX – Basiskonzepte. http://www.emotive.de (28.07.2011)

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ABS	**Anti-Blockier-System**
ESP	**Elektronisches Stabilitäts-Programm**
ODF	**Open Diagnostic Framework**
WfMC	**Workflow Management Coalition**
WfMS	**Workflow Management System**
ISO	**International Standards Organisation**
ASAM	**Association for Standardization of Automation and Measuring Systems**
BPEL	**Business Process Execution Language**
ECU	**Electric Control Unit**
MVCI	**Modular Vehicle Communication Interface**
VCI	**Vehicle Communication Interface**
API	**Application Programming Interface**
ODX	**Open Diagnostic Exchange**
OSI	**Open Systems Interconnection**
PDU	**Protocol Data Unit**
XSD	**XML Schema Definition**
XML	**eXtensible Markup Language**
ODF	**Open Diagnostic Framework**
UI	**User Interface**
HMI	**Human Machine Interface**
BPMN	**Business Process Modeling Notation**
MVC	**Model-View-Control**
PIN	**Personal Identification Number**

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

< Ort, Datum >