Bachelorthesis

Entwurf und prototypische Implementierung einer serverunabhängigen und flexibel zur Laufzeit parametrierbaren Testkomponente für eine bestehende Clientanwendung

Vorgelegt am: 21. August 2020

Von: Pit Friedrich
Moritzstraße 18
09212 Limbach-Oberfrohna

Studiengang: Wirtschaftsinformatik

Studienrichtung: Wirtschaftsinformatik

Seminargruppe: WI17

Matrikelnummer: 4003323

Praxispartner: DELTA proveris AG
Ludwig-Richter-Straße 3
09212 Limbach-Oberfrohna

Gutachter: Dipl.-Ing. (BA) Knut Polifka (DELTA proveris AG)
Prof. Dr. Rainer Penzel (Staatliche Studienakademie Glauchau)

Inhalt

Abbildungsverzeichnis IV

Tabellenverzeichnis V

Abkürzungsverzeichnis VI

1 Einleitung 1

1.1 Problemstellung und Relevanz 1

1.2 Zielstellung 2

1.3 Aufbau der Arbeit 2

2 Architektur von Software 4

2.1 Allgemeines 4

2.2 Model-View-Controller 4

2.3 Model-View-Presenter 5

2.4 Model-View-ViewModel 6

2.5 Synchronisation der Daten und Darstellung 7

2.5.1 Synchronisation mit dem Observer-Pattern 7

2.5.2 Synchronisation in JavaFX 8

3 Softwarequalitätssicherung durch Testen 10

3.1 Testen im Softwarelebenszyklus 10

3.2 Teststufen 11

3.2.1 Modul- / Komponententest 11

3.2.2 Integrationstest 12

3.2.3 Systemtest 13

3.2.4 Abnahmetest 13

3.3 Entwicklertest 13

3.4 Testbarkeit der MVVM-Architektur 14

3.4.1 Unittest 14

3.4.2 Integrationstest 14

3.4.3 Einsatz von Testdoubles 15

3.4.4 Strategien zum Erzeugen von Testdaten 17

3.4.4.1 Manuell 17

3.4.4.2 Zufallsbasiert 17

3.4.4.3 Kopieren von Serverdaten 17

3.4.5 Strategien zur Erstellung von Mockobjekten 18

3.4.5.1 Manuell 18

3.4.5.2 Proxy-Entwurfsmuster ... 18
3.4.5.3 Bytecode-Manipulation ... 20
3.4.6 Mocking-Frameworks ... 20
3.4.6.1 Zustandsverifikation vs. Verhaltensverifikation ... 20
3.4.6.2 EasyMock ... 21
3.4.6.3 jMock ... 23
3.4.6.4 Mockito ... 23

4 Die MVVM-Architektur in der DELTA Workbench ... 26
4.1 View ... 27
4.2 Modell ... 27
4.2.1 Funktion der Modelle ... 27
4.2.2 Modelltypen ... 28
4.3 Viewmodell ... 29
4.4 Dependency Injection von Modellen und Services ... 31
4.5 Unittest-Framework ... 31

5 Konzeption einer Testkomponente ... 34
5.1 Anforderungsanalyse ... 34
5.2 Serverunabhängige Simulation ... 37

6 Implementierung der Testkomponente ... 39
6.1 Mockmodus-Schalter ... 39
6.2 Viewmodell-Events ... 39
6.3 Modellverarbeitung ... 40
6.4 Verarbeitung eines Modellobjekts ... 43
6.5 Steuerung aktiver Features ... 44
6.6 Erweiterungs- und Optimierungsmöglichkeiten ... 45
6.6.1 Bereitstellung der Testdaten in strukturierter Form ... 45
6.6.2 Zwischenspeicherung der Mockdatenbank ... 46
6.6.3 Manuelle Auswahl des Mockdatensatzes ... 46

7 Zusammenfassung ... 47
Literaturverzeichnis ... 49
Anhangsverzeichnis ... 51

Abbildungsverzeichnis

Abbildung 1 Model-View-Controller 4
Abbildung 2 Model-View-Presenter 6
Abbildung 3 Model-View-ViewModel 7
Abbildung 4 Observer-Entwurfsmuster 8
Abbildung 5 Allgemeines V-Modell 10
Abbildung 6 Proxy-Entwurfsmuster 18
Abbildung 7 Dynamic Proxy Class API 20
Abbildung 8 Beipielhafte Verwendung von EasyMock 22
Abbildung 9 Beispielhafte Verwendung von jMock 23
Abbildung 10 Beispielhafte Verwendung von Mockito 24
Abbildung 11 Architektur des Leasman – Ist und Soll 26
Abbildung 12 Schematische Darstellung der MVVM-Architektur in der DWB . 27
Abbildung 13 Schematische Darstellung der Viewmodell-Hierarchie 30
Abbildung 14 Modell und Stub-Modell 32
Abbildung 15 Schematische Darstellung der Mockmodell-Implementierung .. 32
Abbildung 16 Beispielhafte Verwendung eines DWB-Mockmodells 33
Abbildung 17 Mockup des Testmoduls 36
Abbildung 18 Konflikt der verschiedenen Mockdatensätze 37
Abbildung 19 Abbildung der Viewmodellhierarchie durch die Testkomponente 41
Abbildung 20 Tabelle für die Parametrierung der Mockmodelle 42
Abbildung 21 Inspektion des Modellobjekts 43
Abbildung 22 Parametrierbarkeit der Leasman-Features 44
Abbildung 23 Exemplarischer Vergleich XML – YAML 45
Abbildung 24 Datendopplung im Mockmodus 46

Tabellenverzeichnis

Tabelle 1 Beispiel der Verhaltensdefinition in BDD 12
Tabelle 2 Übersicht Testdoubles 16
Tabelle 3 Übersicht Modell-Interfaces 28
Tabelle 4 Übersicht der Viewmodelle 29
Tabelle 5 Verfügbare Operationen der (Mock-)Modelle 43

Abkürzungsverzeichnis

AAA	Arrange – Act – Assert
API	Application Programming Interface
BDD	Behavior-Driven Development
BLS	Business Logic Server (Leasman Geschäftslogik)
DWB	DELTA Workbench (Leasman Client)
EJB	Enterprise Java Bean
GUI	Graphical User Interface (grafische Benutzeroberfläche)
IDE	Integrated Development Environment (integrierte Entwicklungsumgebung)
JDK	Java Development Toolkit
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MO	Model Object
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
OOP	Objektorientierte Programmierung
RCP	Rich Client Platform
RMI	Remote Method Invocation
SWT	Standard Widget Toolkit
TO	Transportobjekt
UI	User Interface (Benutzerschnittstelle)
XML	eXtensible Markup Language
YAML	YAML Ain’t Markup Language

1 Einleitung

1.1 Problemstellung und Relevanz

Das Testen von Software ist ein umfangreiches Verfahren, welches viel Zeit in Anspruch nimmt und ein planvolles Vorgehen erfordert. Daher sind Strategien, Paradigmen und Tools zum Testen von Softwareanwendungen entstanden. Auf Entwicklerseite wird meistens codenah getestet, wobei man sich hier auf den Teststufen des Komponenten- und Integrationstests befindet. Bei den Tests werden nur einzelne Komponenten oder das Zusammenspiel einiger Komponenten und nicht das ganze System betrachtet. Da diese Komponenten in den meisten Fällen Abhängigkeiten zu anderen Komponenten besitzen und eine Ausführung der im Test betrachteten Objekte Nebeneffekte produzieren können, wird auf das Konzept von Mocks zurückgegriffen. Mocks sind Stellvertreterobjekte, die das Verhalten des echten Teilsystems imitieren und keine Veränderung am Zustand des Systems vollziehen. Auch bei einer fehlenden Implementierung der Abhängigkeiten des Testobjekts kann bis zur Fertigstellung der Komponente mithilfe eines Mocks getestet werden.

Beim Testen einer 3-Schichten-Architektur kann das Konzept von Mocks eingesetzt werden, um die Interaktion des Clients mit dem Server zu simulieren. Wenn die Tests mit dem realen Server stattfinden, lassen sich manche Testsituationen nur schwer oder gar nicht herstellen. Eine lange Ladezeit des Servers ließe sich nur mit der Manipulation des Servercodes bewerkstelligen. Ebenso lassen sich Fehler auf Serverseite nur provozieren, wenn beim Aufruf einer Methode falsche Parameter übergeben werden. Dies könnte mithilfe eines Mocks umgangen werden, der so konfiguriert werden kann, dass er den Anforderungen eines gewünschten Testfalls entspricht ohne Anpassungen am Server nachvollziehen zu müssen.

Dieses Konzept der Mockobjekte kommt auch bei der Cliententwicklung des Leasman zur Anwendung. Leasman ist eine Geschäftsanwendung für den Bereich Leasing und Finanzierung. Es verfolgt das Konzept einer 3-Schichten-Architektur, die Client, Server und Datenbank umfasst. Die Architektur des Clients wiederum folgt dem Model-View-ViewModel-Paradigma, bei der eine strikte Trennung von Daten und deren Darstellung erfolgt. Dieses Konzept erlaubt eine isolierte Testbarkeit der Viewmodelle, welche den Zustand der Oberfläche abbilden. Die Viewmodelle besitzen eine Abhängigkeit zu den Modellen, welche die Daten für die Darstellung beschaffen. Wenn ein Viewmodell getestet wird, wird ihm ein Mockmodell statt der realen Implementierung als Abhängigkeit zur Verfügung gestellt. Somit kann der Test ohne das Vorhandensein eines Servers stattfinden. Die Mockmodelle stellen konstante Daten bereit, die von den Viewmodellen entsprechend interpretiert und von der Oberfläche dargestellt werden.

Um eine Implementierung testen zu können, müssen Testdaten erstellt werden, die eine für den Test adäquate Konstellation abbilden. Dies erfordert Zeit und einen hohen fachlichen Kenntnisstand von dem zu entwickelnden Modul. Um bspw. einen Leasingvertrag in den für den Test erforderlichen Zustand zu bringen, muss der Entwickler genau wissen, welche Schritte dafür notwendig sind.

Da die Daten in einer zentralen Datenbank hinterlegt sind und mehrere Personen unter Umständen Tests mit denselben Daten durchführen, können sich Daten zwischen zwei Testdurchführungen ändern, was ungewollt zu verschiedenen Ergebnissen führen kann. Testdaten, die hingegen nicht von einem Server bezogen werden, können beliebig angepasst werden, ohne dass Validierungen der Geschäftslogik berücksichtigt werden müssen. Es stehen allein die Daten und deren Darstellung im Fokus der Tests.

1.2 Zielstellung

Eine Parametrierung der Mockobjekte im Quelltext erfordert Zeit und u. U. ein Neustart der Clientanwendung. Für Leasman existieren für verschiedene Kunden eigene Projekte mit spezifischen Parametrierungen. Will man nun eine Implementierung in einer kundenspezifischen Umgebung testen, ist dafür der Wechsel des Projekts notwendig. Daher wäre es vorteilhaft, wenn dies umgangen werden könnte und eine dynamische Parametrierung zur Laufzeit stattfinden könnte.

Mit dieser Arbeit soll untersucht werden, ob sich das Prinzip und die Vorteile von der Nutzung von Mockobjekten auch auf manuelle Tests und den eigenständigen Betrieb eines Clients ohne Server übertragen lassen. Dabei wird ein neues Modul konzipiert, welches den Entwickler bei der Implementierung und beim Test unterstützen soll und somit den Workflow verbessern kann. Diese Testkomponente soll einen Schalter bereitstellen, der ein Umschalten der Anwendung zwischen Server- und Mockmodus erlaubt. Weiterhin soll es möglich sein, mehrere Testdaten manuell zu erstellen oder in einer strukturierten Form auf der Festplatte ablegen zu können, die dann bei Bedarf von dem Testmodul geladen werden können. So können bei einem manuellen Test die Testdaten einfacher geändert werden. Ebenso wird die Möglichkeit geschaffen die Daten am Viewmodell selbst zu inspizieren und ihre Werte einzusehen und ggf. Änderungen vorzunehmen. Diese Arbeit soll die Vor- und Nachteile von Komponenten- und Integrationstests mit Mocks aufzeigen und die Möglichkeiten analysieren, diese flexibel zur Laufzeit zu parametrieren.

1.3 Aufbau der Arbeit

Zu Beginn soll auf die Architekturmuster für interaktive Systeme im Allgemeinen und grafische Benutzeroberflächen im Besonderen eingegangen werden. Hier wird ein

Vergleich angestellt mit besonderem Fokus auf die Wart- und Testbarkeit der verschiedenen Paradigmen. Anschließend werden die verschiedenen Teststufen im Softwarelebenszyklus genauer betrachtet und die in dieser Arbeit betrachteten Tests eingeordnet. Es soll auch ein Blick auf die Rolle der verschiedenen Arten von Stellvertreterobjekten bei Softwaretests geworfen werden. Dabei wird auch ein Vergleich angestellt zwischen drei verschiedene Bibliotheken für die Erstellung von Mockobjekten. Die Architektur der DELTA Workbench soll im Folgekapitel beleuchtet werden, insbesondere mit Blick auf den Einsatz von Unittests und Mockmodellen als Stellvertreterobjekte. Dabei wird auch der Unterschied zwischen dem eigenen Mockingkonzept und den im Vorkapitel vorgestellten Mocking-Bibliotheken verdeutlicht. Darauf aufbauend wird eine Testkomponente spezifiziert, die das Mocking-Konzept auf die manuelle Testebene überträgt und eine dynamische Parametrierung der Mockobjekte zur Laufzeit ermöglicht. Dabei werden die Vorteile aber auch die Grenzen einer solchen Komponente aufgezeigt. Zum Schluss wird die Implementierung der entstandenen Testkomponente näher erläutert und ein Ausblick auf das Verbesserungs- und Erweiterungspotential gegeben.

2 Architektur von Software

2.1 Allgemeines

Die Konzeption einer Softwarearchitektur ist ein komplexer Vorgang, der einer intensiven Planung bedarf und eine tiefe Auseinandersetzung mit den gestellten Anforderungen an das zu implementierende System erfordert. Ähnlich wie die Entwurfsmuster im Software Engineering haben sich auch Architekturmuster und Best Practices in der Planung und Umsetzung von Softwarearchitekturen entwickelt. In den folgenden Abschnitten werden einige Architekturmuster für interaktive Systeme erläutert.

2.2 Model-View-Controller

Das Model-View-Controller-Architekturmuster (MVC) stellt das älteste der hier aufgeführten Patterns für grafische Oberflächen (GUI) und interaktive Systeme im Allgemeinen dar. Es wurde ursprünglich in den 1970er Jahren von Trygve Reenskaug für die Programmiersprache Smalltalk entworfen.[1] Der Hauptgedanke hinter diesem Konzept ist die Trennung zwischen Darstellung und der Datenhaltung einer Softwarearchitektur. Die Daten werden auf Modellebene geladen, verwaltet und gespeichert. Die View-Komponente repräsentiert die Struktur des Layouts und die Darstellung der Daten. Hier ist lediglich die Sicht des Anwenders, ohne jegliche Logik zu finden. Um eine Kommunikation dieser beiden voneinander getrennten Schichten zu erreichen, wird die Controller-Komponente eingesetzt. Innerhalb dieses Architekturkonzepts findet der Datenfluss in zwei Richtungen statt. Zum einen wird bei einem Update des Modells die Sicht aktualisiert. Zum anderen Seite delegiert der Controller die Eingaben des Anwenders aus der Viewkomponente in das Modell (siehe Abbildung 1).

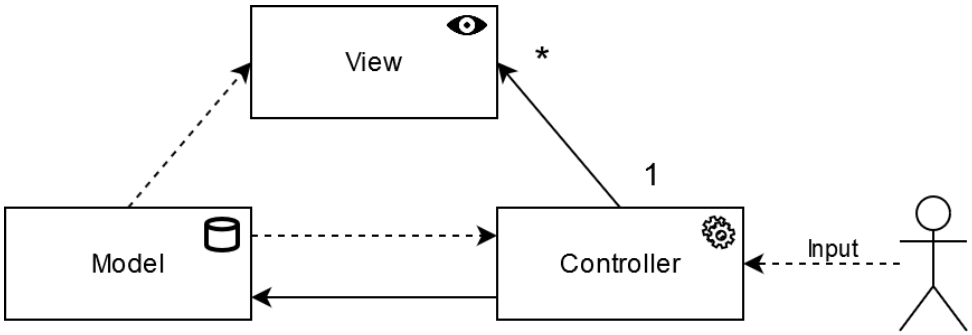


Abbildung 1 Model-View-Controller

(eigene Darstellung in Anlehnung an GOLL, 2014, S. 379)

[1] vgl. FOWLER; RICE, 2011, S. 330

Die häufigen Änderungen der Viewkomponente im Gegensatz zur Modellseite, begründen eine Trennung der Verantwortlichkeiten innerhalb dieses Architekturmusters. Damit bei einer Änderung der Daten das Layout nicht angepasst werden muss, findet diese Trennung statt. Daraus folgt auch, dass die gleichen Daten mit unterschiedlichen Layouts zur gleichen Zeit dargestellt werden können.[2]

Es erfolgt grundsätzlich eine Trennung von Datenhaltung, Verarbeitung, Präsentationslogik und Benutzereingaben in den verschiedenen Komponenten. Die Viewkomponente und der Controller bilden strenggenommen die Benutzeroberfläche. Während die Interaktion mit dem Benutzer auf der Viewebene stattfindet, werden diese im Controller interpretiert und validiert. Das Modell erhält wiederum die bereinigten Daten des Controllers und speichert diese. Das Modell kann somit bei einer Änderung die Sicht aktualisieren.

Somit bildet dieses Muster eine Dreieckskonstellation, bei der die verschiedenen Komponenten voneinander abhängen. Sowohl View als auch Controller besitzen eine Referenz auf das Modell. Der Controller kann die Daten aktualisieren und die View stellt immer den aktuellen Stand der Daten dar. Zwischen View und Controller besteht eine bidirektionale Beziehung. Die Viewkomponente gibt Eingabedaten an den Controller weiter und der Controller kann wiederum die Sicht aktualisieren.

Es gibt dennoch unterschiedliche Ansätze innerhalb dieser Architektur. Auf der einen Seite existiert das passive Modell, bei dem der Controller die View über Änderungen benachrichtigt. Auf der anderen Seite liegt bei einem aktiven Ansatz die Verantwortung für eine Benachrichtigung der Viewkomponente beim Modell. Bei diesem Ansatz lässt sich weiterhin die Art der Beschaffung der Daten für die View unterscheiden.[3]

- *Push*: Bei dieser Vorgehensweise sendet das Modell die geänderten Daten über den Aufruf einer Methode an seine registrierten Views
- *Pull*: Das Modell benachrichtigt die Viewkomponente über eine Datenänderung. Die Beschaffung erfolgt von ihr selbst per Methodenaufruf am Modell.

2.3 Model-View-Presenter

Der Model-View-Presenter (MVP) unterscheidet sich durch die geringere Kopplung der Schichten und des Datenflusses innerhalb dieser Architektur. Während beim MVC Abhängigkeiten zwischen allen Ebenen bestehen, hat die Viewkomponente im MVP keine Kenntnis vom Modell und das Modell nicht von der View (siehe Abbildung 2). Der gesamte Datenaustausch erfolgt über den Presenter als Zwischenschicht, welcher

[2] vgl. GOLL, 2014, S. 289

[3] vgl. GOLL, 2014 S. 377 - 383

die Daten von der Viewkomponente erhält, ggf. umformt und an das Modell zum Speichern übergibt. Der Presenter ruft somit alle Funktionen am Modell auf, auch um Daten zu laden und die aktualisierten Daten an die Viewkomponente zu senden. Mit dieser Abwandlung des MVC-Musters findet eine Verschiebung der Steuerungskomponente in die Logikschicht und weg von der Darstellungsebene statt. Somit stellt diese Architektur eine strengere Trennung der Verantwortlichkeiten dar und erhöht die isolierte Testbarkeit der einzelnen Komponenten.[4]

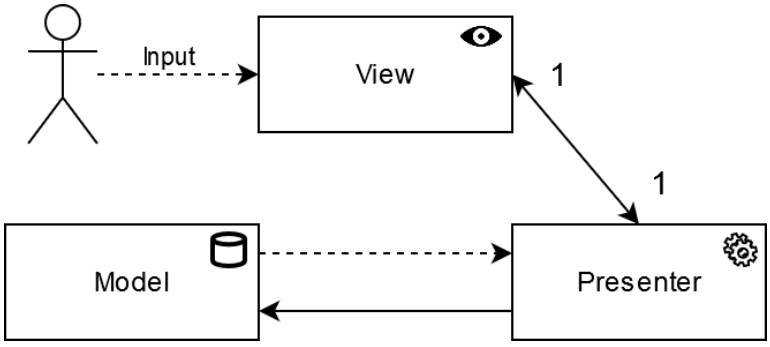


Abbildung 2 Model-View-Presenter

(eigene Darstellung in Anlehnung an KOURAKLIS, 2016, S. 6)

2.4 Model-View-ViewModel

Das Hauptziel des MVVM-Architekturmusters ist es, wie beim MVP, die Viewkomponente und das Modell vollständig zu entkoppeln, um damit eine hohe Wart- und Testbarkeit zu ermöglichen. Während in den ersten beiden Mustern die Logikschicht und die View ähnliche Verantwortlichkeiten besitzen, repräsentiert das Viewmodell den Zustand und die Logik der Sicht. Das Modell und die View sind im Vergleich zu den ersten beiden Architekturmustern schlank und stellen im Fall der Sichtebene nur das Layout und das Mapping zum Viewmodell bereit. Das Modell liefert nur die Methoden zum Beschaffen und Ändern der Daten. Die Daten selbst werden dann im Viewmodell hinterlegt und verwaltet.

Charakteristisch für das MVVM-Muster ist auch der Informationsfluss innerhalb der Architektur (siehe Abbildung 3). Die View kennt das Viewmodell, aber das Viewmodell nicht die View. Dies führt dazu, dass verschiedene Layouts an dasselbe Viewmodell andocken können (im Unterschied zum MVP). Ebenso verhält es sich mit dem Viewmodell und dem Modell. Ersteres ruft Methoden des Modells auf und muss dieses somit kennen. Letzteres wiederum ist zustandslos und kennt das Viewmodell nicht.[5]

[4] vgl. KOURAKLIS, 2016, S. 6 - 7

[5] vgl. KOURAKLIS, 2016, S. 7 - 11

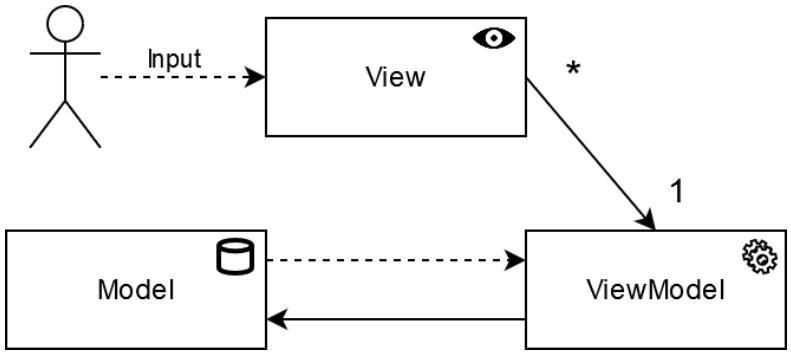


Abbildung 3 Model-View-ViewModel

(eigene Darstellung in Anlehnung an online: MICROSOFT, 2012)

2.5 Synchronisation der Daten und Darstellung

2.5.1 Synchronisation mit dem Observer-Pattern

Um die Synchronizität der Oberflächenelemente mit den Daten sicherzustellen, muss die Viewkomponente über Datenänderungen auf der Modellebene benachrichtigt werden. Allerdings sollen auch die Abhängigkeiten so gering wie möglich gehalten werden. Um dies zu erreichen findet das ‚Inversion of Control'-Prinzip Anwendung. Dieses erlaubt die Reduzierung der Abhängigkeiten zwischen zwei Komponenten, indem der Kontrollfluss in eine Richtung aber der Datenfluss in beide Richtungen möglich ist.[6] Die Umsetzung des Prinzips erfolgt in den bisher genannten Architekturmustern mithilfe des Beobachter-Entwurfsmusters.[7]

Das Beobachter-Muster oder auch Observer-Pattern genannt wird für die Synchronisierung von Komponenten eingesetzt. Es besteht aus Beobachtern (Observer) und einem Beobachteten (Subject). Der Observer stellt die passive Komponente dar und reagiert auf Änderungen des Subjects. Somit übergibt Ersterer die Kontrolle vollständig an das Subject (siehe Abbildung 4). Ohne dieses Muster müsste die Viewkomponente in regelmäßigen Abständen das Viewmodell auf geänderte Daten befragen (Polling). Dies ist ineffizient und würde dem Synchronizitätsgedanken widersprechen. Dieses Entwurfsmuster erlaubt eine 1-zu-n-Beziehung zwischen Observer und Subject. Der Observer kann lediglich ein Subject beobachten aber das Subject beinhaltet eine Liste von Observern, die bei einer Datenänderung benachrichtigt werden. Somit ist es beispielsweise möglich, dass innerhalb der MVVM-Architektur verschiedene Views an ein Viewmodell gekoppelt werden können. Um eine deutlichere Vorstellung von diesem Entwurfsmuster zu erhalten wird es in der Literatur auch häufig Publish-Subscribe-Muster (kurz: PubSub)

[6] vgl. GOLL, 2014, S. 28 - 29

[7] vgl. online: FOWLER, 2004 (06.08.2020)

genannt.[8] Der Publisher (Subject) ‚veröffentlicht' Benachrichtigungen, die von der Subscribern (Observer) empfangen werden können. Eine beliebige Anzahl von Subscribern können die Benachrichtigungen von einem Publisher empfangen.

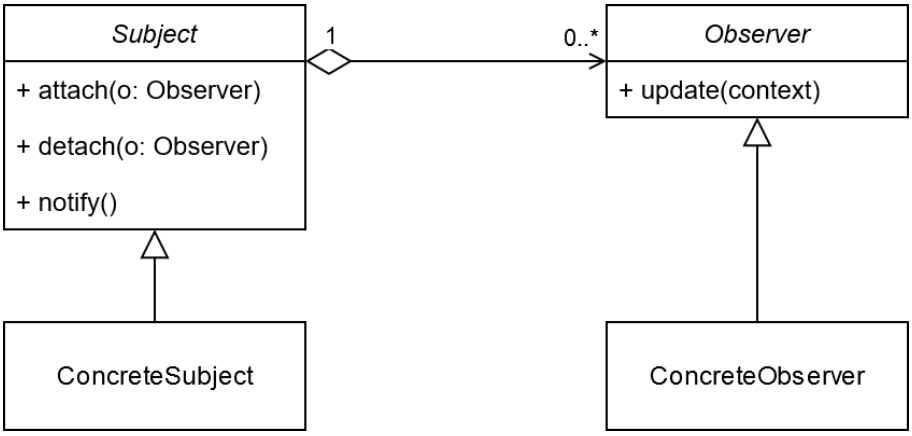


Abbildung 4 Observer-Entwurfsmuster

(eigene Darstellung in Anlehnung an GAMMA; u.a., 1995, S. 295)

2.5.2 Synchronisation in JavaFX

In JavaFX kann die Synchronizität der Oberfläche mit den Daten über Propertys und den dazugehörigen Listenern sichergestellt werden. Propertys sind Klassen, die ein Objekt oder den Wert eines primitiven Datentyps beinhalten. Wird dieser Wert das erste Mal ungültig gesetzt, sendet das Property ein Invalidierungsereignis aus, an das sich ein Invalidierungslistener hängen kann. Ändert sich der Wert des eines Propertys, sendet das Property ein Änderungsereignis aus, das wiederum mittels Changelistener abgefragt werden kann. Weiterhin existieren für Collections vom Typ *ObservableList* und *ObservableMap* spezielle Listener, die auf eine Änderung eines Elements der Collection reagieren können.[9]

Um Propertys miteinander zu synchronisieren, gibt es eine vereinfachte Verwendung der Listener. Mit einem *Binding* kann der Wert eines Propertys immer auf den Wert eines anderen Propertys gesetzt werden. Das ist sowohl in eine Richtung als auch in beide Richtungen mit einem bidirektionalen Binding möglich. Alle Steuerelemente (Buttons, Labels, Textfelder, etc.) sind Klassen, die zugrundeliegende Propertys besitzen. Mittels Binding können sie an den Zustand des Viewmodells gebunden werden, um somit immer dessen Zustand wiederzugeben. Durch ein bidirektionales Binding ist auch der umgekehrte Fall möglich: den Zustand des Viewmodells mit dem

[8] vgl. GAMMA; U. A., 1995, S. S. 293

[9] vgl. EPPLE, 2016, S. 19 - 23

der Oberfläche zu synchronisieren.[10] Dieses Property- und Bindingkonzept basiert auf dem Observer-Designpattern. Propertys sind allerdings Observer und Subject zugleich, die sie sich gegenseitig ‚beobachten' können, um ein bidirektionales Binding und damit eine Wertesynchronisation zu ermöglichen.

[10] vgl. EPPLE, 2016, S. 25 - 31

3 Softwarequalitätssicherung durch Testen

3.1 Testen im Softwarelebenszyklus

Das Testen von Software ist ein wesentlicher Bestandteil des Softwareentwicklungsprozesses. Es beinhaltet alle Aufgaben, die sich mit dem Auffinden und Erkennen von Fehlern befassen. Ob eine Software tatsächlich fehlerfrei ist, kann mit Tests nie vollkommen sicher festgestellt werden. Oft sind es Ausnahmesituationen, die nicht erfasst und geprüft werden, da ein Softwaresystem ab einem gewissen Umfang eine hohe Komplexität aufweist. Beim Testen wird die Software überprüft, ob sie den gestellten Anforderungen entspricht. Dabei kann die Qualität eingeschätzt werden und bei einer Feststellung von Fehlern, Korrekturen veranlasst werden. Das Auffinden eines Fehlers im Quellcode und die darauffolgende Fehlerkorrektur liegt im Aufgabenbereich des Entwicklers und wird auch als ‚Debugging' bezeichnet.[11]

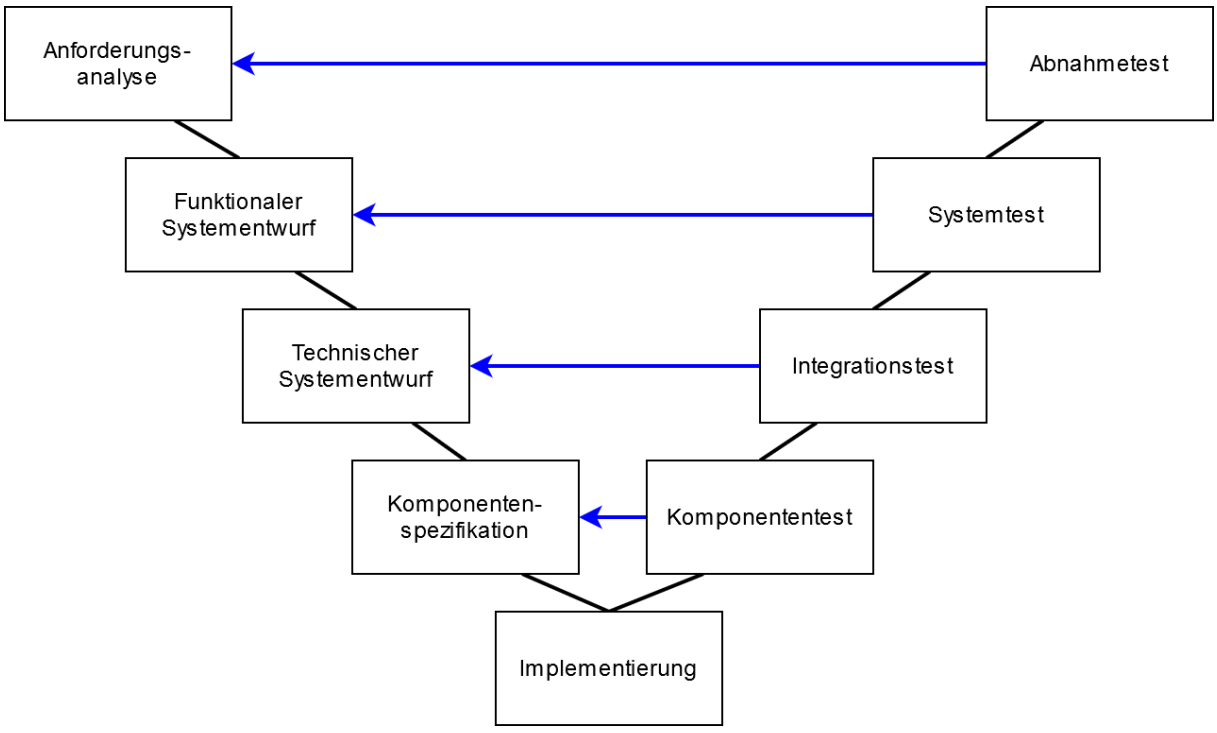


Abbildung 5 Allgemeines V-Modell

(eigene Darstellung in Anlehnung an SPILLNER; LINZ, 2019, S. 55)

Es existieren unterschiedliche Vorgehensmodelle zum Abbilden des Softwareentwicklungsprozesses. Dabei unterscheidet man zwischen sequenziellen und iterativ-inkrementellen Modellen. Die bekanntesten Vertreter des ersten Typs sind

[11] vgl. SPILLNER; LINZ, 2019, S. 7 - 13

das Wasserfallmodell und das allgemeine V-Modell. Ersteres beinhaltet eine einzige Testphase, die nach der Implementierung stattfindet. Beim V-Modell hingegen gibt es zu jeder Entwurfs- und Implementierungsphase eine korrespondierende Testphase (siehe **Abbildung 5**). In der jeweiligen Testphase erfolgt die Verifizierung der entsprechenden Umsetzungsphase. Durch die unterschiedlichen Tests, die meist auch in der Verantwortung verschiedener Personen liegen, erhöht sich die Flexibilität und es lässt sich angemessener auf Fehler reagieren. Zu den iterativ-inkrementellen Modellen, die hier allerdings nicht näher erläutert werden sollen, zählen z.B. das Spiralmodell aber auch agile Vorgehensmodelle wie Scrum und eXtreme Programming.[12] In den nachfolgenden Abschnitten werden die Teststufen des allgemeinen V-Modells näher erläutert.

3.2 Teststufen

3.2.1 Modul- / Komponententest

Der Modul- bzw. Komponententests überprüft die kleinsten autonomen Einheiten einer Software. Er ist die erste Teststufe, die unmittelbar nach der Implementierung stattfinden kann. Aufgrund der starken Nähe zum Quellcode, wird dieser Test in den meisten Fällen vom Entwickler selbst durchgeführt. Bei objektorientierten Programmiersprachen sind die zu testenden Komponenten Klassen und Methoden. Die Methoden werden ggf. mit Testdaten als Übergabeparameter versorgt. Anschließend wird die Methode ausgeführt und das tatsächliche mit dem erwarteten Ergebnis verglichen. Stimmt der Erwartungswert mit dem Ergebnis des Tests überein, gilt er als bestanden.[13]

Diese Vorgehensweise wird auch Arrange – Act – Assert, oder kurz: AAA genannt. Man erzeugt im Arrange-Schritt eine Testsituation, bspw. durch die Instanziierung von Klassen, die Vorbereitung von Parametern für den Methodenaufruf oder die Erstellung von Fake-Objekten. Im nächsten Schritt (Act) wird das Testobjekt (die Methode) ausgeführt. Dies erzeugt eine Veränderung des Testobjekts oder einfach die Produktion eines Wertes. Dieser kann im letzten Schritt (Assert) mit dem erwarteten Zustand bzw. Wert verglichen werden.[14]

Diese drei Schritte kommen auch im Behavior Driven Development (BDD) zur Anwendung, welches ein eigenes Vokabular für das Überprüfen von Komponenten bietet. Hier wird auch nicht mehr von ‚Tests' sondern von ‚Verhalten' (Behavior)

[12] vgl. SPILLNER; LINZ, 2019, S. 53 - 60

[13] vgl. SPILLNER; LINZ, 2019, S. S. 63 - 64

[14] vgl. WESTPHAL, 2006, S. 9 - 20

gesprochen, da in einem Test das korrekte Verhalten einer Komponente überprüft wird. Eine Intention für die Verwendung von BDD ist, dass es die Lücke zwischen fachlicher Konzeption und technischer Umsetzung versucht zu schließen, indem es ein einfach verständliches, gemeinsames Vokabular bereitstellt. Weiterhin lassen sich die Verhaltensdefinitionen zu ganzen Sätzen zusammenfassen.[15] Ein kleines Beispiel zur Demonstration ist in Tabelle 1 zu finden.

Bezeichner	Eigenschaft	Beispiel
Given	Initialer Zustand	Es befindet sich Guthaben auf dem Bankkonto.
When	Eintretendes Ereignis	Der Kunde möchte 50 Euro abheben.
Then	Sicherstellung eines Ergebnisses	Das Bankkonto wird um 50 Euro belastet.

Tabelle 1 Beispiel der Verhaltensdefinition in BDD

3.2.2 Integrationstest

Der Integrationstest kann nach dem Komponententest stattfinden. Hier wird das Zusammenspiel einzelner Komponenten getestet. Wenn hier Fehler auftreten, liegt das an der Interaktion der Komponenten, da sie im ersten Schritt bereits isoliert getestet wurden und als fehlerfrei gelten. Im Fokus stehen hier die Schnittstellenformate und der Datenaustausch zwischen einzelnen Komponenten. Folgende Fehlerzustände werden beim Integrationstest unterschieden:[16]

- Eine Komponente sendet keine oder falsche Daten, sodass die empfangende Komponente diese nicht korrekt verarbeiten kann
- Die übertragenen Daten zwischen Komponenten werden anderes als erwartet interpretiert
- Die Daten werden zum falschen Zeitpunkt zwischen den Komponenten ausgetauscht

Die Grenze zwischen dem Komponenten- und Integrationstest ist unscharf. Wann der eine zum anderen wird ist oft anwendungsfallbezogen und abhängig von der erforderlichen Granularität der zu testenden Komponenten. Ob bereits bei einem Test

[15] vgl. NORTH, 2006

[16] vgl. SPILLNER; LINZ, 2019, S. 75

von zwei interagierenden Klassen von einem Integrationstest gesprochen werden kann, ist umstritten. Häufig findet in diesen Fällen auch die Bezeichnung ‚Mikrointegrationstest' Anwendung.[17]

3.2.3 Systemtest

Beim Systemtest wird die Umsetzungsqualität der spezifizierten Anforderungen an das Gesamtsystem überprüft. Das Gesamtsystem bestehend aus Hardware und Software sollte möglichst äquivalent zum Produktivsystem sein. Somit erfolgt diese Teststufe aus Kundensicht. Als Ergebnis vollständig umgesetzte Anforderungen, Fehler und Mängel dokumentiert. Da hier normaler keine Kenntnisse über den Quelltext erforderlich sind, kann der Systemtest den Blackbox-Tests zugerechnet werden.[18]

3.2.4 Abnahmetest

Der Abnahmetest liegt in der Verantwortung des Kunden. Dies kann der direkte Auftraggeber, die zukünftigen Benutzer oder der Systembetreiber sein. Beim Abnahmetest kann es sich um ein formales Review der durchgeführten Tests und deren Ergebnisse oder um eigene Tests zur Verifikation der Abnahmekriterien handeln. Von den potenziellen Nutzern können auch Akzeptanztests durchgeführt werden, da diese zukünftig mit der Software arbeiten sollen.[19]

3.3 Entwicklertest

Der Test durch den Entwickler ist keine eigenständige Teststufe. Hier ist lediglich die Person benannt, die den Test durchführt. Wenn der Entwickler seinen eigenen Code testet, ist es möglich, dass Fehler bereits zu einem frühen Zeitpunkt im Softwareentwicklungsprozess entdeckt und behoben werden können. Es ist auch vorteilhaft, dass der Entwickler sein eigenes Werk testet, da er über ein detailliertes Knowhow der Implementierung verfügt, was bei der Fehlerbehebung und der Zusammenarbeit mit den Testkollegen hilfreich sein kann.

Der Nachteil daran ist, dass der Entwickler beim Test über keine ausreichende Distanz zu seiner Implementierung verfügt und somit keine passenden oder kritischen Testfälle durchführt. Deshalb ist es immer besser eine andere Person mit einem ausreichenden Abstand zum Quellcode mit dem Testen zu betrauen. Ein weiterer Nachteil das Testen einer anderen Person als dem Entwickler zu überlassen ist, dass sie sich für das

[17] vgl. WESTPHAL, 2006, S. 177 - 178

[18] vgl. SPILLNER; LINZ, 2019, S. 80 - 81

[19] vgl. SPILLNER; LINZ, 2019, S. 82 - 83

Testen der Komponente oder des (Teil-)Systems sich Wissen darüber aneignen muss. Daher ist es sinnvoll, dass Entwickler und Tester eng zusammenarbeiten [20].

Der Entwickler hat die Möglichkeit bei der Lokalisierung der Fehlerursache einen Debugger benutzen, der Auskunft über die Fehlerursache geben kann. Meist ist ein Debugger in der Entwicklungsumgebung (IDE) integriert, durch den er beim Ausführen der Software die verschiedenen Zustände des Systems anzeigen und untersuchen kann. Sinnvoll ist dabei auch die schrittweise Ausführung des Programms und die Anzeige der Veränderung von Variablen von bestimmten Klassen über den Zeitraum der Ausführung.

3.4 Testbarkeit der MVVM-Architektur

3.4.1 Unittest

Wie bereits in 2.4 geschildert, ist der größte Vorteil der MVVM-Architektur die lose Kopplung zwischen View und Präsentationslogik (Viewmodell). Es besteht bei Letzterem allerdings eine Abhängigkeit zur Modellebene, was den Komponententest erschwert. Das Wesen von Unittests ist der isolierte Test einer Komponente bzw. eines Moduls. Der Test zielt somit auf die Überprüfung einer funktionalen Einheit eines Systems in Isolation ab. So können Fehler lokal ermittelt werden und es werden keine Nebeneffekte erzeugt. Weiterhin schränken reduzierte Abhängigkeiten die Effekte auf die zu testende Komponente ein. Da eine Software aber selten aus einzelnen, isolierten Komponenten besteht, sollten die Abhängigkeiten einzelner Klassen künstlich reduziert oder vereinfacht werden. Dies kann mit Test-Doppelgängern erfolgen.[21] Diese imitieren das Verhalten einer Abhängigkeit des Testobjekts, ohne eine komplexe Logik oder eine Interaktion mit anderen Komponenten zu beinhalten.

3.4.2 Integrationstest

Das Viewmodell und die View können während der Laufzeit verschiedene Zustände annehmen, die auch von den Daten des Modells abhängig sind: bspw. ist ein Steuerelement nicht sichtbar, wenn der Wert eines Property null ist. Daher ist es bei einem Integrationstest sinnvoll, dass dieser mit verschiedenen Datenkonstellationen durchgeführt werden kann. Wie diese Konstellationen oder Testdaten im Allgemeinen erzeugt werden können, wird in 3.4.4 näher beleuchtet.

Da ein Integrationstest innerhalb der MVVM-Architektur auch die View betrifft, ist ein automatisierter Integrationstest nur schwer möglich. Hierfür existieren Bibliotheken wie

[20] vgl. SPILLNER; LINZ, 2019, S. 51

[21] vgl. online: FOWLER, 2006 (14.08.2020)

TestFX[22], die Unittests von JavaFX-UI-Komponenten automatisieren. Die Verwendung dieses Tools könnte für Integrationstests erweitert werden, um alle Ebenen der MVVM-Architektur abzudecken. Dazu benutzt TestFX eine Robot API, die die Controls ansteuern und mit ihnen interagieren kann. Auch Verifizierungen des Zustands dieser Controls sind möglich, um die Erwartungen des Tests zu überprüfen.[23] Bei einem Integrationstest zwischen View und Viewmodell sind die Bindings von Interesse und wie die View diese darstellt. Dabei sollten aber nur selbst entwickelte Controls getestet werden, da JavaFX-eigene Steuerelemente als vertrauenswürdig gelten können und somit keiner Tests bedürfen.

3.4.3 Einsatz von Testdoubles

Wie bereits in 3.4.1 geschildert, können Abhängigkeiten des Testobjekts mittels Testdoubles reduziert werden. Das einfachste ist das *Dummy*-Objekt, welches keine Funktion enthält und nur eingesetzt wird, um die Abhängigkeiten des Testobjekts zu erfüllen. Wenn der Test aber auf eine Interaktion des Testobjekts mit der Abhängigkeit abzielt, kommen *Stubs* zum Einsatz. Diese geben konstante Daten zurück und beinhalten keinerlei Logik, sodass sich das Stubobjekt bei jedem Test gleich verhält. Methoden ohne Rückgabewert haben einen leeren Methodenkörper.

Ein Mockobjekt ist eine Erweiterung des Stubs. Während Stubobjekte auschließlich konstante Daten produzieren, erweitern Mocks dieses Konzept um eine integrierte Testfunktionalität. Hier lässt sich das Verhalten dynamisch festlegen. Zum Beispiel kann konfiguriert werden, dass eine Methode einen Fehler produziert. Außerdem kann die Anzahl der Methodenaufrufe oder die erwarteten aufgetretenen Fehler festgelegt werden. Im nächsten Schritt erfolgt die Ausführung des Testobjektes, d.h. der Aufruf der zu testenden Methode, welches mit dem Mockobjekt interagiert. Im letzten Schritt werden die Erwartungswerte mit den Ergebnissen verglichen. Die Mocks stellen eine *verify*-Methode bereit, die ihren tatsächlichen mit dem erwarteten Zustand vergleicht.

Statt die Daten an einen Server zu schicken bzw. von diesem zu holen, muss ein Ersatz-Modell zum Einsatz kommen, welches konstante Daten bereitstellt. Somit lässt sich das Viewmodell isoliert testen, ohne auf die Korrektheit des tatsächlichen Modells achten zu müssen.[24]

[22] https://github.com/TestFX/TestFX

[23] vgl. unveröffentlicht: FRIEDRICH, 2019, S. S. 6

[24] vgl. ACHARYA, 2014, S. 26 - 43

Bezeichnung	Funktionen
Dummy	▪ Stellt keine Funktionen zur Verfügung (Null-Objekt) ▪ Keine aktive Verwendung in Tests ▪ Für Instanziierung der zu testenden Klasse benötigt
Stub	▪ Existieren ausschließlich für Softwaretests ▪ Liefert festgelegte Daten an die aufrufende Methode ▪ Kann Informationen über Anzahl der Methodenaufrufe aufzeichnen
Spy	▪ Überwacht die Interaktionen mit dem Testobjekt ▪ Aufzeichnung der Methodenaufrufe
Mock	▪ Kombination aus Stub und Spy ▪ Integrierte Testfunktionalität (Selbstvalidierung)
Fake / Simulator	▪ Echte Logik statt konstanter Daten

Tabelle 2 Übersicht Testdoubles

Die Tabelle 2 zeigt die unterschiedlichen Kategorien von Testdoubles. Eine Bedingung für die Verwendung von Testdoubles ist ein sauberes Softwarekonzept. Es muss sichergestellt sein, dass das Liskovsche Substitutionsprinzip umgesetzt ist. Dieses besagt, dass die Ableitungen einer Klasse zu jeder Zeit durch ihre Basisklasse austauschbar sein müssen.[25] D.h. in diesem Fall, dass das Modell zwar als Instanz im Viewmodell existieren muss, dieses aber immer durch eine Modellbasisklasse bzw. eine Ableitung davon ausgetauscht werden kann.

Darüber hinaus ist es gut Praxis, dass Objekte ihre Abhängigkeiten nicht selbst erzeugen, sondern diese von außen bekommen. Dieses Prinzip wird Dependency Injection genannt und lässt sich mit zahlreichen Strategien realisieren: bspw. über einen Setter, der die Modellinstanz neu setzt, oder einen Service Locator, der eine Registry mit allen Modellen verwaltet, die das Viewmodell bei Bedarf beziehen kann. Es ist auch möglich dies mit einer Fabrikmethode umzusetzen, die neue Modelle anhand einer übergebenen Spezifizierung erstellt.[26]

[25] vgl. MARTIN, 2014, S. 111

[26] vgl. MARTIN, 2014, S. 128 - 130

Zur Erstellung von Mockobjekten gibt es verschiedene Ansätze und Strategien und es haben sich (nicht nur) in der Java-Entwicklung einige Frameworks herauskristallisiert, die den Entwickler bei dieser Aufgabe unterstützen oder die Erzeugung von Mocks vollständig übernehmen. Allerdings ist es auch möglich Mocks selbst zu erstellen.

3.4.4 Strategien zum Erzeugen von Testdaten

3.4.4.1 Manuell

Bei einem manuellen Ansatz werden die Testdaten vom Entwickler selbst festgelegt. Er hinterlegt sie fest im Quellcode und die Anwendung verwertet diese bei einem Test entsprechend. Dies ist oft mit einem hohen Aufwand verbunden, da sich der Entwickler sinnvolle, kontextabgängige, für den Testfall passende Daten einfallen lassen muss.

3.4.4.2 Zufallsbasiert

Bei einem zufallsbasierten Ansatz sind die Daten von Testausführung zu Testausführung verschieden, aber für den jeweiligen Testfall dennoch gültig. Hier können Tools wie *JavaFaker*[27] genutzt werden, die einen umfangreichen Pool von Testdaten zur Verfügung stellen. Diese Daten sind nach Themengebieten kategorisiert, bspw. Emailadressen, Namen, Firmen, Telefonnummern, Produkte, Lebensmittel, Farben, usw. Dies eignet sich aber eher für automatische Unittests statt für manuelle Tests, da die Daten nicht berechenbar sind, auch wenn sie für den jeweiligen Anwendungsfall gültig sind.

3.4.4.3 Kopieren von Serverdaten

Werden hingegen Daten aus einem Produktiv- oder Testsystem übernommen, können auch realistische Testfälle durchgeführt werden. Durch das Kopieren sind sie auch nicht den Änderungen ausgesetzt, denen sie auf dem Server widerfahren. Würde direkt mit den Serverdaten getestet werden, gibt es keine Garantie dafür, dass die Daten zwischen zwei Tests gleichbleiben, da sie sich durch manuelle Tests verschiedener Entwickler und Tester ändern können. Ein weiterer Vorteil dieses Ansatzes ist, dass die kopierten Testdaten beliebig angepasst werden können ohne dass das zu serverseitigen Validierungsproblemen führt. Um diese Daten im Quellcode verfügbar zu machen, müssen sie beim Kopieren in strukturierter Form auf der Festplatte abgelegt werden. Anschließend werden sie beim Test geladen. Strukturierte Formate können bspw. XML, JSON und YAML sein. Alle drei Formate haben ihre Vor- und Nachteile in der Maschinen- bzw. Menschenlesbarkeit (siehe 6.6.1).

[27] https://github.com/DiUS/java-faker

3.4.5 Strategien zur Erstellung von Mockobjekten

3.4.5.1 Manuell

Eine Bedingung hierfür ist, dass das Mockobjekt und die tatsächliche Implementierung ein gemeinsames Interface teilen, um als Abhängigkeiten der zu testenden Klasse austauschbar zu sein. Während das „echte“ Modell mit dem Server kommuniziert, gibt die Mockimplementierung konstante Daten zurück. Welche Daten das konkret sind, legt der Entwickler selbst fest. Dies ist zugleich ein Nachteil, da die Daten sehr umfangreich sein können.

3.4.5.2 Proxy-Entwurfsmuster

Proxys im Allgemeinen werden als Stellvertreter für Objekte eingesetzt. Dies kann notwendig werden, wenn bspw. die Erstellung des echten Objektes zu ineffizient ist, und stattdessen mit einem Objekt gearbeitet wird, welches diese laufzeitintensive Operation umgeht und erst durch einen expliziten Zugriff ausführt. Beispiel: Die *Eclipse Resource API* der Eclipse Entwicklungsumgebung (IDE) verwaltet alle Ressourcen (Projekte, Ordner und Dateien) im Workspace der IDE. Diese können geöffnet, geschlossen, bearbeitet und gelöscht werden. Damit Eclipse bei ungeöffneten Ressourcen nicht unnötigerweise auf die Festplatte zugreifen muss, arbeitet die API mit Ressourcen-Proxys, die nur Metainformationen beinhalten. Der Durchgriff auf die ‚echte‘ Ressource erfolgt dann explizit durch einen Methodenaufruf am Proxy.[28]

Weiterhin können Proxys das reale Objekt vor unautorisierten Zugriffen schützen, indem sie die Autorisierungsinformationen prüfen, bevor sie Methodenaufrufe an das echte Objekt delegieren. Der Proxy implementiert das gleiche Interface wie das echte Objekt und delegiert die Methodenaufrufe unter bestimmten Voraussetzungen an es weiter (siehe Abbildung 6).

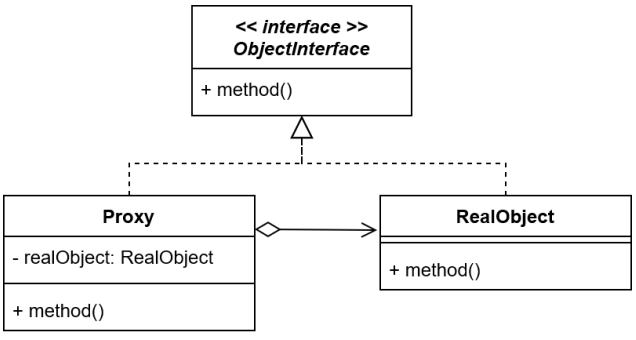


Abbildung 6 Proxy-Entwurfsmuster

(eigene Darstellung in Anlehnung an GAMMA, u.a., 1995, S. 211)

[28] vgl. online: ARTHORNE, 2002 (08.08.2020)

Die ‚Gang of Four' unterscheidet zwischen vier Arten von Proxys, die für verschiedene Situationen einsetzbar sind:[29]

- *Remote Proxy* ist ein Stellvertreterobjekt für eine Instanz auf einer entfernten Java Virtual Machine (JVM). Bei einem Methodenaufruf werden die Daten serialisiert, über das Netzwerk übertragen und auf Empfängerseite deserialisiert. Der Proxy verhält sich allerdings, als würden die Methodenaufrufe lokal erfolgen. Beispiel: Remote Enterprise Java Beans (EJB).
- *Virtual Proxy* ist ein Stellvertreter für ein Objekt dessen Instanziierung zu teuer ist. Wie bereits eingehend geschildert, wird das tatsächliche Objekt erst bei einer expliziten Anforderung erstellt.
- *Protection Proxy* prüft die Zugriffe auf ein Objekt anhand von Autorisierungsinformationen, um sie falls notwendig zu verhindern.
- *Smart Reference Proxy* kann die Methodenaufrufe abfangen und Funktionen des echten Objekts verändern oder ersetzen. Beispielsweise wird diese Art Proxy für das Transaktionsmanagement des Spring-Frameworks genutzt. Diese Verwendung von Proxys bietet sich auch für die Erstellung von Mockobjekten an.

Damit das Proxy-Muster angewendet werden kann, muss das echte Objekt ein Interface implementieren, welches die Methoden definiert, die gemockt werden sollen. Soll zu einer Vielzahl von Interfaces eine Mockimplementierung erstellt werden, muss auch hier manuell eine zusätzliche Implementierung des Interfaces erfolgen. Somit unterscheidet sich diese Vorgehensweise zur vorherigen nur dahingehend, dass nicht alle Methoden gefaked werden müssen. Methodenaufrufe, die sich von der realen Implementierung nicht unterscheiden sollen, können einfach delegiert werden. Gemockte Aufrufe hingegen, werden nur eingeschränkt bzw. überhaupt nicht delegiert und stattdessen mit Stellvertretungslogik gefüllt.

Um diesen Nachteil zu umgehen, gibt es im JDK seit der Version 1.3 die *Dynamic Proxy Class API*, die eine Erstellung dynamischer Proxys zur Laufzeit ermöglicht. Er fängt den Aufruf aller Methoden ab und kann Funktionalitäten zum Methodenaufruf hinzufügen oder diesen vollständig ersetzen. Auch hier ist es möglich konstante Daten zurückzugeben, statt die tatsächliche Methode aufzurufen und mit dem Server zu kommunizieren. Allerdings sollte beachtet werden, dass alle Methoden der

[29] vgl. GAMMA; U. A., 1995, S. 208 - 211

Klassenhierarchie überschrieben werden, einschließlich der Object-Methoden *toString*, *equals, finalize* und *hashCode*.[30]

Die Hauptbestandteile der Dynamic Proxy Class API sind das Interface *InvocationHandler* und die statischen Methoden der Klasse *Proxy*. Ersteres wird implementiert, um das Verhalten des Proxys beim Aufruf der Methoden festzulegen. Letzteres wird genutzt, um den Proxy zu erstellen. Zur Instanziierung ist das Interface des tatsächlichen Objekts, der InvocationHandler und ein Classloader nötig (siehe **Abbildung 7**).

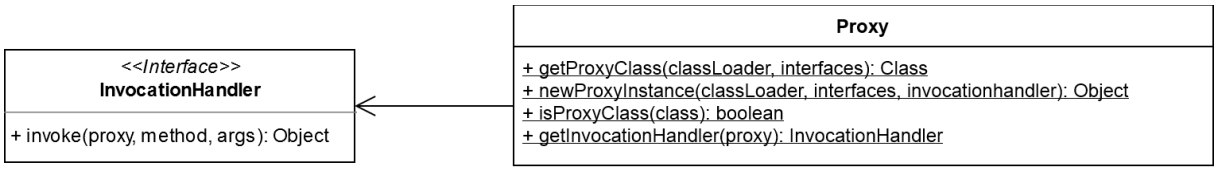


Abbildung 7 Dynamic Proxy Class API

3.4.5.3 Bytecode-Manipulation

Es ist allerdings nicht immer gegeben, dass zu mockende Objekte ein Interface besitzen. Hier stößt auch die Dynamic Proxy Class API an ihre Grenzen, da diese ausschließlich mit Interfaces funktioniert. Frameworks wie *Mockito* benötigen hingegen keine Interfaces, um einen Mock aus einer Klasse zu erstellen. Mockito verwendet hier die Bibliothek *ByteBuddy*, die den Bytecode zur Laufzeit manipuliert und somit eine „leere“ Implementierung der Originalklasse erzeugt und diese mit Testfunktionalitäten ausstattet (siehe 3.4.6.4).

3.4.6 Mocking-Frameworks

3.4.6.1 Zustandsverifikation vs. Verhaltensverifikation

Martin Fowler unterscheidet beim letzten Schritt eines Unittests (Assert) zwischen Zustands- und Verhaltensverifikation. Bei Unittests ohne den Einsatz von Mockobjekten wird in der Regel nur der Zustand des Testobjekts und ggf. von seinen Kollaborateuren nach dem Ausführen des Testobjekts überprüft. Bei der Verhaltensverifikation kommen die Mocks ins Spiel. Im Arrange-Schritt des Unittests wird ein erwartetes Verhalten für die Mockobjekte festgelegt. Im Assert-Schritt wird das erwartete gegen das tatsächliche Verhalten geprüft. Als Verhalten ist hier gemeint,

[30] vgl. online: VITZ, 2020 (20.07.2020)

welche Methoden, in welcher Reihenfolge, wie oft am Mock aufgerufen werden und welche Rückgabewerte dieser produziert.[31]

Es haben sich zahlreiche Mockingframeworks herauskristallisiert, die hier im Ansatz beleuchtet werden sollen. Dabei soll auch der Fokus insbesondere Auf das Problem des *Stubbing* gelegt werden. Dies ist meist die umfangreichste Aufgabe bei der Erstellung von Mockobjekten, da die Stub- bzw. Testdaten manuell erstellt werden müssen. Damit eine realistische Konstellation dieser Daten erstellt werden kann, ist außerdem fachliches Wissen über die zu testende Komponente erforderlich. Ebenso ist es sinnvoll, Testdaten zu erstellen, die Grenzfälle abbilden, um ein breites Spektrum von Anwendungsfällen abzubilden und entlang der Grenzen erfolgreiche Tests durchführen zu können. Ebenso kann es von Interesse sein, Testdaten zu nutzen, bei denen der Test bewusst fehlschlägt, um auch Negativfälle einzubeziehen.

In den kommenden Absätzen sollen folgende Frameworks an einem Beispiel untersucht werden: EasyMock, jMock und Mockito. Dabei soll der gleiche Testfall mit den verschiedenen Frameworks durchgeführt werden. Der Fall hat dabei folgende Konstellation: Es wird ein Modell erstellt, welches Fahrzeugdaten von einem Server bezieht. Dieses wird von einem Viewmodell referenziert, welches beim Laden die load-Operation des Modells aufruft. Im Unittest fungiert das Viewmodell als Testobjekt, das Modell als Kollaborateur bzw. Mock.

Alle hier aufgeführten Frameworks nutzen die Möglichkeiten der Bytecodemanipulation, um Proxys von Klassen erstellen zu können. Dabei verwenden jMock und Mockito die Bibliothek ByteBuddy, EasyMock hingegen verwendet CGLib. Die Manipulation überschreibt das Verhalten der echten Klasse zur Laufzeit und ermöglicht ein dynamisches Stubbing und eine Aufzeichnung der Methodenaufrufe. Alle drei Frameworks werden nach aktuellem Stand gewartetet und weiterentwickelt.

3.4.6.2 EasyMock

EasyMock existiert seit 2001 und damit das älteste der hier vorgestellten Frameworks. Es stellt eine Schnittstelle zur Erstellung von Mocks für Java und .NET-Anwendungen bereit.[32] Es verwendet einen Record-Replay-Mechanismus zu ‚Aufnahme' bzw. Festlegung des erwarteten Verhaltens des Mockobjekts. Erfolgt ein Aufruf der ‚Replay'-Methode wird die Aufzeichnung beendet und der Mock wird für die Interaktion mit dem Testobjekt vorbereitet. Beim Ausführen des Testobjekts speichert EasyMock die tatsächlichen Aktionen des Mocks. Im Assert-Schritt des Tests kann zusätzlich zur Zustandsverifikation des Testobjekts eine Verhaltensverifikation (erwartetes =

[31] vgl. online: FOWLER, 2007 (29.07.2020)

[32] vgl. online: EasyMock, 2020 (10.08.2020)

tatsächliches Verhalten) des Mocks erfolgen. Sind beide Überprüfungen erfolgreich ist der Test bestanden.[33] Eine Demonstration der Verwendung von EasyMock kann der Abbildung 8 entnommen werden.

```
@Test
public void testWithEasyMock() throws Exception {
    // Arrange
    IFahrzeugModel mockModel = EasyMock.mock(IFahrzeugModel.class);
    FahrzeugViewModel viewModel = new FahrzeugViewModel(mockModel);
    FahrzeugMO mo = new FahrzeugMO();
    mo.setId(1L);
    mo.setBezeichnung("Audi A8");
    mo.setAkz("C-XX 123");

    // Erwartetes Verhalten des Mocks festlegen + Stubbing
    EasyMock.expect(mockModel.load(1L)).andReturn(mo);
    EasyMock.replay(mockModel);

    // Act
    viewModel.onActionLoad(true);

    // Assert
    assertEquals(Long.valueOf(1L), viewModel.getModelObject().getId());
    assertEquals("Audi A8", viewModel.getModelObject().getBezeichnung());
    assertEquals("C-XX 123", viewModel.getModelObject().getAkz());

    // Erwartetes Verhalten des Mocks == Tatsächliches Verhalten?
    EasyMock.verify(mockModel);
}
```

Abbildung 8 Beipielhafte Verwendung von EasyMock

EasyMock bietet die Möglichkeit unterschiedliche Arten von Mockobjekten zu erstellen, die auch teilweise kombiniert werden können:[34]

- *Default Mock*: die Reihenfolge der Methodenaufrufe wird standardmäßig nicht verifiziert.
- *Strict Mock*: diese Form des Mocks führt eine Überprüfung der Aufrufreihenfolge durch.
- *Nice Mock*: das Standardverhalten der mit EasyMock erzeugten Mocks ist, dass alle Methoden, deren Verhalten nicht explizit festgelegt ist, beim Aufruf eine Exception produzieren. Ein ‚netter' Mock hingegen gibt immer den NULL-Wert der aufgerufenen Methode zurück (null, 0 oder false).
- *Partial Mock*: ein partieller Mock erlaubt das Verhalten bestimmter Methoden zu überschreiben, während alle anderen Methoden des echten Objekts unberührt bleiben.

[33] vgl. online: FOWLER, 2007 (29.07.2020)

[34] vgl. online: EasyMock, 2020 (10.08.2020)

3.4.6.3 jMock

Das Framework jMock gibt es seit 2004 und ermöglicht es ebenso wie EasyMock eine einfache Instanziierung von Mockobjekten. Dabei wird zuerst ein Kontext erstellt, der wiederum das Mockobjekt erzeugt. Im Anschluss wird ein dynamisch konfigurierbares Erwartungsobjekt in den Kontext gelegt (siehe Abbildung 9).

```
@Test
public void testWithJMock() throws Exception {
    // Arrange
    Mockery context = new Mockery();
    IFahrzeugModel mockModel = context.mock(IFahrzeugModel.class);
    FahrzeugViewModel viewModel = new FahrzeugViewModel(mockModel);
    FahrzeugMO mo = new FahrzeugMO();
    mo.setId(1L);
    mo.setBezeichnung("Audi A8");
    mo.setAkz("C-XX 123");

    // Erwartetes Verhalten des Mocks festlegen + Stubbing
    context.checking(new Expectations() {{
            oneOf(mockModel).load(1L);
            will(returnValue(mo));
    }});

    // Act
    viewModel.onActionLoad(true);

    // Assert
    assertEquals(Long.valueOf(1L), viewModel.getModelObject().getId());
    assertEquals("Audi A8", viewModel.getModelObject().getBezeichnung());
    assertEquals("C-XX 123", viewModel.getModelObject().getAkz());

    // Erwartetes Verhalten des Mocks == Tatsächliches Verhalten?
    context.assertIsSatisfied();
}
```

Abbildung 9 Beispielhafte Verwendung von jMock

Dabei spielt die Reihenfolge der Erwartungsdefinitionen eine wichtige Rolle, da diese exakt der Reihenfolge der Methodenaufrufe am Mockobjekt bei der Testdurchführung entsprechen muss. Im letzten Schritt erfolgt auch hier u.a. die Verhaltensverifikation des Mockobjekts.[35]

3.4.6.4 Mockito

Mockito existiert effektiv seit dem Jahr 2008. Das Framework basierte anfangs auf EasyMock und ist das zurzeit am häufigsten eingesetzte Framework zur Erstellung von Mockobjekten in Java.[36] Das Stubbing der Methodenaufrufe erfolgt ähnlich dem von EasyMock. Allerdings ist die Deklaration der gestubbten Methoden an die natürliche Sprache angelehnt und kann somit u.U. auch von Personen ohne Programmierkenntnisse verstanden werden (siehe Abbildung 10). Mockito stellt sogenannte *ArgumentMatchers* für das Stubbing zur Verfügung. Dies sind Platzhalter für Werte, die nicht fest angegeben werden: bspw. `anyInt()` liefert einen beliebigen

[35] vgl. online: jMock, 2019 (10.08.2020)

[36] vgl. online: Mockito - Webseite, 2020 (10.08.2020)

ganzzahligen Wert, `argThat(isValid())` hingegen produziert einen zufälligen gültigen Wert. Somit kann sichergestellt werden, dass das Ergebnis des gestubbten Methodenaufruf immer dasselbe ist, auch wenn der Methode verschiedene Parameter übergeben werden. Die Reihenfolge der Methodenaufrufe muss auch explizit festgelegt werden, da dies für Mockito standardmäßig keine Rolle spielt. Dafür muss der Entwickler den Mock einem Inorder-Verifikator übergeben, der die dafür nötigen Funktionalitäten bereitstellt.

Weiterhin ermöglicht Mockito die Erzeugung von Spy-Objekten. Im Gegensatz zu Mocks arbeiten Spys mit den echten Objekten. Diese können verwendet werden, wenn die Erstellung von Mocks nicht erforderlich ist, da die echten Objekte keine Nebeneffekte produziert bzw. keine Kommunikation mit einem Server durchführt. Die Spys ermöglichen aber dennoch eine Verhaltensverifikation.

```
@Test
public void testWithMockito() throws Exception {
    // Arrange
    IFahrzeugModel mockModel = Mockito.mock(IFahrzeugModel.class);
    FahrzeugViewModel viewModel = new FahrzeugViewModel(mockModel);
    FahrzeugMO mo = new FahrzeugMO();
    mo.setId(1L);
    mo.setBezeichnung("Audi A8");
    mo.setAkz("C-XX 123");

    // Erwartetes Verhalten des Mocks festlegen + Stubbing
    Mockito.when(mockModel.load(1L)).thenReturn(mo);

    // Act
    viewModel.onActionLoad(true);

    // Assert
    assertEquals(Long.valueOf(1L), viewModel.getModelObject().getId());
    assertEquals("Audi A8", viewModel.getModelObject().getBezeichnung());
    assertEquals("C-XX 123", viewModel.getModelObject().getAkz());

    // Erwartetes Verhalten des Mocks == Tatsächliches Verhalten?
    Mockito.verify(mockModel).load(1L);
}
```

Abbildung 10 Beispielhafte Verwendung von Mockito

Methoden können bei ihrem Aufruf die ihr übergebenen Parameter verändern. Um diesen Wert im Test überprüfen zu können, stellt Mockito die Verwendung von *ArgumentCaptors* zur Verfügung, die den Parameter ‚einfangen' und es ermöglichen deren Zustand nach der Ausführung des Testobjekts zu verifizieren. Mit dem Release der Version 3 ermöglicht Mockito auch das Mocking von statischen und finalen Methoden.[37] Private Methoden hingegen sind nur unter Zuhilfenahme von PowerMock mockbar, das eine API für Mockito bereitstellt, welche diese Funktionalitäten erfüllt. Es sollte allerdings genau untersucht werden, ob es tatsächlich notwendig ist solche Methoden zu mocken. Oft kann auch eine Änderung des Softwaredesigns Abhilfe schaffen.

[37] vgl. online: Mockito 3.4.6 API, 2020 (10.08.2020)

Zusammenfassend ist Mockito das umfangreichste der hier vorgestellten Frameworks. Es bietet zahlreiche Möglichkeiten die Abhängigkeiten eines Testobjekts zu ersetzen. EasyMock und jMock stellen hingegen die grundlegenden Funktionen bereit, um Mockobjekte zu erzeugen und somit für einfache Anwendungsfälle durchaus geeignet.

4 Die MVVM-Architektur in der DELTA Workbench

Leasman ist eine Geschäftsanwendung für den Bereich Leasing und Finanzierung. Sie bietet die Funktionalitäten, um den Lebenszyklus eines Leasingvertrags von der Angebotserstellung, über die Bestellung des Fahrzeugs, der Vertragsaktivierung, bis hin zur Rechnungsstellung und Abwicklung des Vertrags abzubilden. Die Anwendung wurde ursprünglich in Sybase PowerBuilder entwickelt. Da Software ein immaterielles Gut ist, das einem Alterungsprozess unterliegt, unterläuft Leasman seit Jahren einer technologischen Portierung von einer 2-Tier- hin zu einer 3-Tier-Architektur. Während in der PowerBuilder-Implementierung Präsentations- und Geschäftslogik miteinander verwoben sind, sollen in der portierten Software diese Komponenten strikt getrennt sein. Die Geschäftslogik befindet sich dann auf Serverseite (Tier 2) und wird Business Logic Server (BLS) genannt. Der Anwender interagiert mit dem Desktopclient (Tier 1), welcher wiederum Methoden am BLS aufruft. Um eine modulweise Portierung zu ermöglichen, wird die Eclipse Rich Client Platform (RCP) genutzt, die sowohl PowerBuilder- als auch portierte Module beinhaltet. Die in Eclipse mitgelieferte Oberflächenbibliothek Standard Widget Toolkit (SWT) wurde anfangs zur Entwicklung des Clients genutzt, später wurde JavaFX als GUI-Framework eingesetzt.

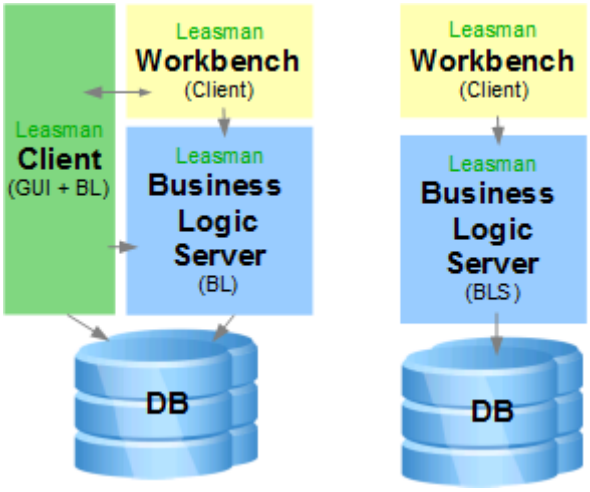


Abbildung 11 Architektur des Leasman – Ist und Soll
(unveröffentlicht: DELTA PROVERIS AG, 2020, S. 13)

Mit dem Umstieg auf JavaFX wurde auch gleichzeitig das Model-View-Viewmodell Architekturparadigma eingesetzt. Es findet eine Trennung von Darstellung, der Präsentationslogik und der Datenschicht statt (siehe Abbildung 11). Durch die Einführung der MVVM-Architektur und der damit verbundenen losen Kopplung der Komponenten entsteht auch die Möglichkeit der Nutzung von Unittests für die Viewmodelle (siehe Abbildung 12). Insgesamt erhöht dieser Umstieg die Test- und Wartbarkeit der DELTA Workbench. Weiterhin wurde mit diesem Umstieg auch die

Möglichkeit der Nebenläufigkeit genutzt. Der Aufruf von BLS-Methoden braucht nun nicht länger nacheinander, sondern kann in separaten Threads stattfinden. In den nachfolgenden Abschnitten soll nun die Architektur näher beleuchtet werden.

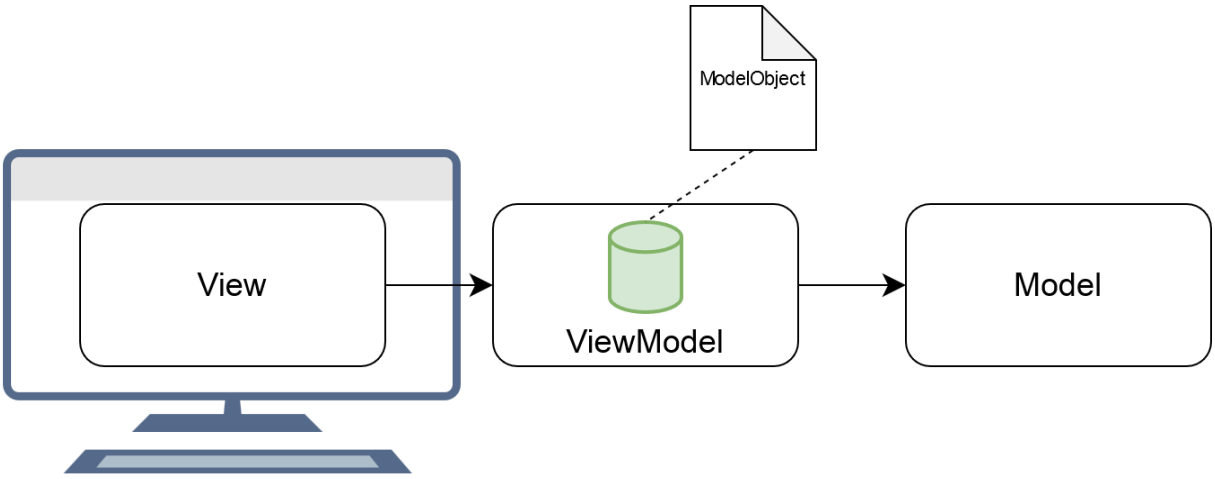


Abbildung 12 Schematische Darstellung der MVVM-Architektur in der DWB

4.1 View

Auf der obersten Ebene befindet sich die View, mit der der Anwender interagiert. Diese wird in den meisten Fällen durch eine FXML-Datei erstellt, die die Struktur des Layouts abbildet. Diese wird zur Laufzeit von der JavaFX-Bibliothek geladen und daraus wird ein sogenannter Scenegraph erstellt. Dieser repräsentiert die Hierarchie aller Steuerelemente auf der Sichtebene. Aufgrund dieser Abbildung als XML-Datei kann ein Entwurf der Oberfläche mit einem Design-Tool erfolgen. Dafür stellt das Unternehmen Gluon den SceneBuilder bereit. Die View kann um eine Controller-Klasse ergänzt werden. Der Name ist etwas irreführend, da es sich hierbei nicht um einen Controller im MVC-Sinne handelt, sondern lediglich um eine ‚Code-Behind'-Klasse, in der das Binding mit dem Viewmodell erfolgt.

4.2 Modell

4.2.1 Funktion der Modelle

Auf der letzten Ebene sind die Modelle zu finden, die Methoden via Remote Method Invocation (RMI) am Server aufrufen und ggf. Daten von diesem erhalten. Die Methoden im Modell fungieren als Wrapper für den Aufruf der Servermethoden, da diese lediglich Transportobjekte (TO) bereitstellen, die Felder mit einfachen Datentypen bzw. Collections enthalten. Damit ein Binding zwischen der View und dem Viewmodell funktioniert, müssen Propertys eingesetzt werden. Propertys sind Container-Objekte für Variablen, die ein abhörbares Event aussenden, sobald sich der

beinhaltete Wert ändert. Diese Events können von Listenern konsumiert werden, um auf eine Änderung des Modellwertes zu regieren, bspw. durch eine Aktualisierung der Oberfläche. Daher findet beim Aufruf der Modellmethoden auch ein Mapping der TO-Felder in Propertys statt. Diese werden in Modellobjekten (MO) gekapselt und vom Modell zurückgegeben.[38]

4.2.2 Modelltypen

Viele Modelle bieten ähnliche Funktionalitäten zum Laden, Speichern und Löschen von Daten. Durch die Möglichkeiten der Objektorientierung ist eine Hierarchie von Interfaces für die Modelle entstanden, um eine einheitliche Interaktion zu ermöglichen (siehe Tabelle 3). Ein Klassendiagramm, welches die Hierarchie illustriert ist in Anhang 1 zu finden.

Modellinterface	Funktionen
IModel	▪ Marker-Interface ▪ Keine eigenen Methoden
ILoadableModel	▪ Lädt einen Datensatz ▪ Bestimmung des Datensatzes anhand eines Parameters
ILoadableListModel	▪ Lädt eine Liste von Datensätzen ▪ Bestimmung der Datensätze anhand eines Parameters
IParametrizedPersistentModel	▪ Einfügen und Aktualisierung eines Datensatzes ▪ Sperren und Entsperren eines Datensatzes ▪ Bestimmung des Datensatzes anhand eines Parameters
IPersistentModel	▪ Gleiche Funktionalität wie ParametrizedPersistentModel ▪ Keinen Parameter
IMasterDetailModel	▪ Lädt Liste von Datensätzen ▪ Löschen eines Datensatzes
IPeriodMasterDetailModel	▪ Lädt zeitabhängige Datensätze ▪ Zeitliches Beenden eines Datensatzes

Tabelle 3 Übersicht Modell-Interfaces

[38] vgl. unveröffentlicht: DELTA PROVERIS AG, 2020, S. 23 - 24

Weiterhin existieren noch spezielle Modell-Interfaces für Suchen und die Historien von Daten, sowie Modelle, die Methoden für häufige Interaktionen mit einem Geschäftsobjekt bereitstellen. Diese werden aus Platzgründen nicht näher erläutert.

4.3 Viewmodell

In der Viewmodell-Ebene ist eine ähnliche Hierarchie und Abstraktion wie bei den Modellen zu finden, um Boilerplate-Code in der Modulentwicklung zu unterbinden (siehe Tabelle 4).

Bezeichnung	Funktionen	Korrespondierendes Modell-Interface
Viewmodel	• Basis-Viewmodell • Alle Grundfunktionalitäten • Titel, Icon, Kind-Viewmodelle, …	-
TypedViewModel	• Beinhaltet Modellobjekt	-
ModelbaseViewModel	• Beinhaltet Modell	IModel
LoadableViewModel	• Lädt Daten • Kopiert diese in das MO	ILoadableModel
LoadableListViewModel	• Lädt Listen • Kopiert eine Liste von Modellobjekten	ILoadableListModel
ParametrizedViewModel	• Einfügen, Aktualisieren von Daten	IParametrizedPersistentModel
PersistentViewModel	• Einfügen, Aktualisieren von Daten • Kein Parameter zum Ermitteln der Daten	IPersistentModel
MasterDetailViewModel	• Lädt Masterliste • Modellobjekt ist aktiver Detailsatz	IMasterDetailModel
PeriodMasterDetailViewModel	• Lädt zeitabhängige Daten in Masterliste • Detailsatz kann zeitlich beendet werden	IPeriodMasterDetailModel

Tabelle 4 Übersicht der Viewmodelle

Die Klassenhierarchie der Viewmodelle ist auszugsweise in Anhang 2 zu finden. Das abstrakte Standard-Viewmodell beinhaltet alle Funktionalitäten, die alle Viewmodelle besitzen sollen, unter anderem einen Titel und ein Icon. Weiterhin enthält es eine Liste mit Kind-Viewmodellen, damit eine Hierarchie aufgebaut werden kann. Durch diese Eltern-Kind-Beziehung ist es möglich Operationen innerhalb der Hierarchie zu delegieren, um bestimmte Methoden nicht für jedes Viewmodell einzeln aufrufen zu müssen. Bsp.: bei einem Viewmodell mit Ladefunktion wird diese Operation nicht nur für das direkt zugehörige Modell, sondern auch für alle Kinder mit Ladefunktion aufgerufen. Das gleiche Verhalten findet sich auch bei Viewmodellen mit Speicher-, Sperr- und Entsperrfunktion. Es werden hier die Viewmodell-spezifischen Methoden für das Speichern, Aktualisieren und Löschen in der Hierarchie nach unten delegiert. Weiterhin ist es möglich einem Viewmodell ein Host-Viewmodell zuzuweisen. Dieses ist erforderlich, wenn das Viewmodell seine Daten nicht selbst bezieht, sondern warten muss, bis sein Host-Viewmodell seine Daten geladen hat. Diese hierarchischen Beziehungen sind beispielhaft in Abbildung 13 abgebildet.

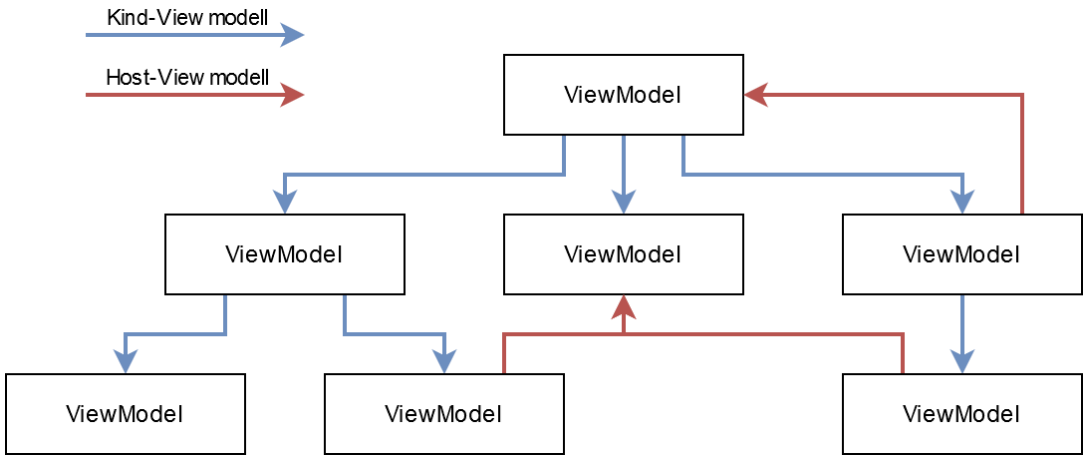


Abbildung 13 Schematische Darstellung der Viewmodell-Hierarchie

Durch diese Abstraktion, der damit verbundenen Wiederverwendbarkeit von Funktionalitäten und der Anwendung der MVVM-Architektur entsteht ein Framework, mit dem die Fachmodulentwicklung standardisiert und vereinfacht werden soll. Services, die nicht in das MVVM-Schema integriert ist, aber dennoch wiederkehrende Funktionen bereitstellen, ergänzen das Framework. Beispielsweise ist es möglich mit dem Preference Service anwendungsspezifische Einstellungen auf der Festplatte des Nutzers zu speichern oder der Formatierungsservice wandelt Datenobjekte in formatierte Zeichenketten um, die oft benötigt werden.

4.4 Dependency Injection von Modellen und Services

Sowohl Modelle als auch Services sind zustandslos, d.h. sie stellen Methoden zur Verfügung, die am Zustand des Modells nichts ändern. Auch von außen ist der Zustand nicht änderbar. Daraus folgt, dass es zur Laufzeit lediglich eine Instanz eines Modells oder Services geben brauch. Die gegenwärtige Architektur der DWB erlaubt das Erzeugen von Services und Modellen über Fabrikmethoden. Diese erzeugen beim Aufruf der *create*-Methode eine neue Instanz oder beziehen eine existierende Instanz aus einem Cache in Form einer Hashmap. Diese bildet Schlüssel-Wert-Paare ab und ermöglicht die Wertermittlung in einer konstanten Zugriffszeit. Der Erzeugungsmechanismus der Objekte erfolgt mittels eines Classpath-Scanners. Dieser erhält den Typ des Interfaces eines Modells oder Services und sucht im Classpath nach der ersten zugehörigen Implementierung des Interfaces. Ein zweiter Scanner findet ausschließlich Modelle bzw. Services, die anhand des Namens des übergebenen Interfaces und des Präfixes „Mock“ eine Implementierung erzeugt bzw. zurückgibt, die keine Kommunikation mit dem BLS vollzieht, sondern konstante Daten liefert.

Somit lässt sich einfach durch den Austausch der Classpath-Scanner-Strategie an der Mock- bzw. Service-Factory die zu instanziierende Implementierung austauschen. Durch dieses Design ist ein globales Umschalten zwischen BLS- und Mockmodus zur Laufzeit möglich. Diese Architektur kann damit die Basis der in dieser Arbeit entwickelten Komponente bereitstellen.

4.5 Unittest-Framework

In der gegenwärtigen Implementierung der DWB-Architektur ist ein manueller Ansatz zur Erstellung von Mockobjekten gewählt (siehe 3.4.5.1), die strenggenommen Stubobjekte sind. Es muss für jedes Modellinterface eine reale und eine Stub-Implementierung erstellt werden (siehe Abbildung 14). Die reale Implementierung ruft Methoden am BLS auf, welche Transportobjekte (TO) zurückgibt. Diese werden in Modellobjekte (MO) gepackt und von der Modellmethode zurückgegeben. Die Stub-Implementierung erstellt hingegen ein Modellobjekt, dessen Daten manuell gefüllt werden müssen. Dies ist oft mit einem hohen Aufwand verbunden, da Modellobjekte viele Propertys beinhalten können.

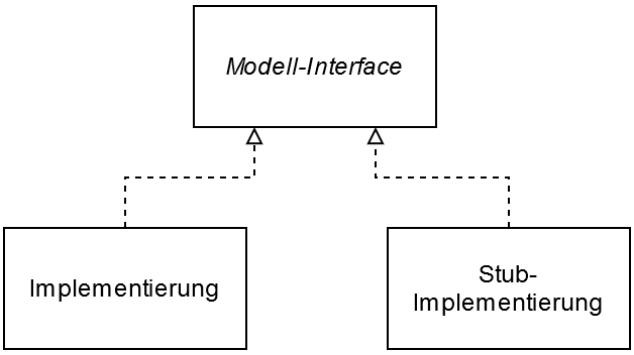


Abbildung 14 Modell und Stub-Modell

Die Stub-Implementierung kann um die Mockfunktionalität erweitert werden. Dafür bietet das Testframework die Möglichkeit zu jeder Modellart in der Hierarchie, um eine Klasse für den Mock zu erweitern (siehe Abbildung 15, Beispiel im Anhang 3). Diese Mocks ermöglichen die Festlegung der Erwartungen an die Anzahl der Methodenaufrufe oder ob die Methode eine Exception produziert. Weiterhin ermöglichen diese Mockmodelle eine Verzögerung bei der Ausführung der jeweiligen Operation einzustellen. Durch diese Möglichkeit lässt sich bspw. eine Exception beim Laden provozieren, um somit die Fehlerbehandlung des Clients zu überprüfen. Außerdem lässt sich hier auch zeigen, wie das versetzte Laden bei den Host-Viewmodellen funktioniert. Die konfigurierten Ladezeiten summieren sich, da das Client-Viewmodell wartet, bis sein Host-Viewmodell seine Operation beendet hat.

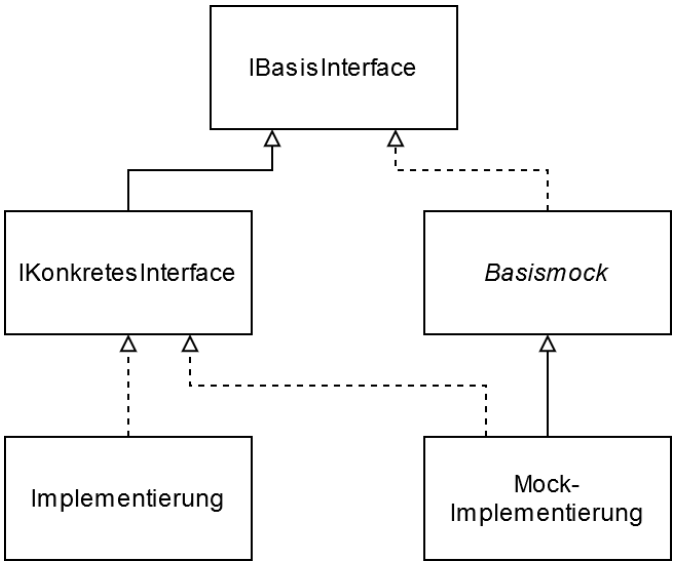


Abbildung 15 Schematische Darstellung der Mockmodell-Implementierung

Wie der Abbildung 16 zu entnehmen ist funktioniert auch das Setup des Tests und dessen Ablauf ähnlich dem der vorgestellten Drittanbieter-Frameworks. Ein wesentlicher Unterschied ist beim Stubbing zu finden. Während dies bei EasyMock,

Mockito und jMock ad-hoc geschieht, ist beim DWB-Mockmodell eine separate Implementierung des Modell-Interfaces nötig, die die Testdaten initialisiert. Ein Vorteil das Stubbing auszulagern ist, dass die Testfälle übersichtlich bleiben und durch die Initialisierung der Daten nicht unnötig aufgebläht werden. Allerdings führt dies auch dazu, dass die doppelte Anzahl an Modellklassen nötig ist, da für jedes konkrete Interface zwei Implementierungen erstellt werden müssen.

```
@Test
public void testWithDefault() {
    // Arrange
    MockFahrzeugModel mockModel = (MockFahrzeugModel) ModelFactory.create(IFahrzeugModel.class);
    FahrzeugViewModel viewModel = new FahrzeugViewModel(mockModel);

    // Erwartetes Verhalten des Mocks festlegen
    // Stubbing wird im MockFahrzeugModel selbst vorgenommen
    mockModel.setExpectedLoadCalls(1);

    // Act
    viewModel.onActionLoad(true);

    // Assert
    assertEquals(Long.valueOf(1L), viewModel.getModelObject().getId());
    assertEquals("Audi A8", viewModel.getModelObject().getBezeichnung());
    assertEquals("C-XX 123", viewModel.getModelObject().getAkz());

    // Erwartetes Verhalten des Mocks == Tatsächliches Verhalten?
    mockModel.verify();
}
```

Abbildung 16 Beispielhafte Verwendung eines DWB-Mockmodells

.

5 Konzeption einer Testkomponente

5.1 Anforderungsanalyse

Die Testkomponente, die im Zuge dieser Arbeit erstellt wird soll zeigen, dass sich eine Clientanwendung, die das MVVM-Pattern implementiert, von einzelnen Modulen bis hin zu anwendungsweiten Einstellungen bei einem Test dynamisch parametrieren lässt. Dabei soll es möglich sein die Anwendung zwischen einem Mock- und einem BLS-Modus umzuschalten. Der BLS-Modus ist die Standardkonfiguration der Anwendung in denen auf Modellebene mit dem BLS kommuniziert wird. Hier werden Daten vom Server geholt und zurückgesendet. Dieses Verhalten entspricht einer klassischen 3-Tier-Architektur. Schaltet man nun in den Mockmodus um, wird die Kommunikation mit dem BLS abgeschnitten. Jetzt findet kein Aufruf von BLS-Methoden mehr statt, sondern es werden lediglich konstante Daten zurückgegeben.

Der Mechanismus, der ein Umschalten zwischen den beiden Modi erlaubt, ist aktuell bereits implementiert. Dieser wird genutzt, um bei automatischen Unittests mit Mockmodellen, statt mit den eigentlichen Modellen zu testen. Beim Initialisieren einer Testsuite wird an der Modell-Factory und der Service-Factory die Classpath-Scanner-Strategie geändert. Dann wird bei jedem Beschaffen eines Modells bzw. Services die Mockimplementierung gefunden und zurückgegeben.

Wird ein Interface in den Fabrikmethoden erstmalig referenziert, wird es instanziiert und in einer Hashmap gecacht. Dies ist sinnvoll, da Modelle zustandslos sind. Bei einer erneuten Referenzierung wird das Modell oder der Service aus dem Cache geholt, wodurch eine Performanceverbesserung erreicht wird.

Das bestehende Verhalten soll auch für manuelle Unit- und Integrationstests genutzt werden können. Dabei soll an dieser Stelle untersucht werden, ob mit einem neuen Modul innerhalb der DWB eine Möglichkeit zur dynamischen Parametrierbarkeit geschaffen werden kann. Dynamisch soll hier bedeuten, dass für eine Neueinstellung der Parameter kein Schließen und erneutes Öffnen des zu testenden Moduls oder ein Neustart der Anwendung erfolgen muss. Das bisherige Vorgehen ist ein unnötiger und zeitaufwendiger Prozess.

Dieses Testmodul soll parallel zu den Fachmodulen geöffnet werden können, um die Eigenschaften des aktiven Moduls zu inspizieren und zu parametrieren. Dazu soll es möglich sein dieses als eine Art Sidebar zu nutzen. Der Nutzer soll allerdings auch in der Lage sein das Testmodul zu verschieben und ggf. in ein separates Fenster auszulagern. Dieses Verhalten ist mit der Eclipse RCP in dieser Form auch möglich und Bedarf daher keiner speziellen Anpassung für das vorliegende Problem.

Das Testmodul muss über einen gut sichtbaren Schalter verfügen, bei dem es möglich ist zwischen BLS- und Mockmodus umzuschalten. Bei einem Schließen des Moduls muss der zuletzt aktive Modus gespeichert werden. Dies kann mithilfe des *Application-Viewmodells* geschehen, das den Zustand der Anwendung repräsentiert. Der aktive Modus kann als simples Boolean-Property hinterlegt werden. Ebenso benötigt das Testmodul einen Schalter, bei dem das aktive Modul neu geladen werden kann, wenn es eine Ladefunktion besitzt. Dies ist insofern notwendig, da bei einer stattgefundenen Parametrierung ein erneutes Laden erforderlich ist, um die Auswirkungen der Datenänderung sichtbar zu machen. Wenn das Wurzel-Viewmodell eines Moduls kein ladbares Viewmodell ist, muss in der Viewmodell-Hierarchie solange gesucht werden (Breitensuche), bis eines gefunden, oder der Baum vollständig traversiert worden ist. In allen bisher portierten Modulen ist es jedoch der Fall, das ein ladbares Viewmodell bereits spätestens auf der zweiten Ebene zu finden ist.

Eine wichtige Funktionalität des Testmoduls ist die Inspektion der Viewmodell-Hierarchie eines Moduls bzw. des Viewmodells selbst. Um die Hierarchie darstellen zu können, ist es möglich die Treeview von JavaFX zu nutzen. Diese ermöglicht die Visualisierung von Hierarchien und stellt eine Ein-/Ausklappfunktion zur Verfügung. Bei der Auswahl eines Elementes (Viewmodell) des Baumes soll es möglich sein, die dahinterliegenden Daten zu analysieren. Dies ist allerdings erst ab der Stufe des typisierten Viewmodell möglich, weswegen dahingehend eine Prüfung erfolgen muss. Die Daten sollen als Liste darstellbar sein mit einer evtl. Möglichkeit zur Manipulation.

Ein weiterer wichtiger Aspekt ist die Möglichkeit der Einstellung einer Verzögerung beim Laden. Dies ist nützlich, um die Ladeanzeige für eine selbstgewählte Zeitspanne sichtbar zu machen. Hier kann auch ein verkettetes Laden sichtbar gemacht werden, wenn ein Viewmodell auf das Fertigstellen das Laden seines Host-Viewmodells warten muss. Die Festlegung der Verzögerungszeit soll über ein Textfeld in Millisekunden geschehen. Das Gleiche ist auch für andere Modelloperationen, wie Speichern, Sperren, Entsperren und Löschen vorgesehen. Weiterhin soll in es möglich sein in der Testumgebung einen Fehler durch den BLS zu provozieren, der vom Client entsprechend behandelt werden muss. Hier soll im Mock-Modus möglich sein, einen künstlichen Serverfehler erzustellen, um die adäquate Fehlerbehandlung des Clients sichtbar zu machen.

Eine Kernfunktionalität von Leasman ist es unterschiedliche Konfigurationen und freigeschaltete Funktionen mithilfe von Features abzubilden. Diese sind für jede Serverkonfiguration gesetzt und lassen sich clientseitig nicht einfach testen. Daher ist es mithilfe eines Mockmodells möglich die freigeschalteten Features selbst einzustellen und ihre Wirkung auf der Oberfläche nachzuvollziehen und zu überprüfen. Daher ist es eine weitere Anforderung an eine Testkomponente die Features nicht nur

im Quellcode, sondern dynamisch, über die grafische Oberfläche einstellen zu können. Da es sich bei den Features nicht um gebundene Propertys, sondern um einfache Objekte handelt, ist es nicht möglich bei einem geöffneten Modul eine Änderung der freigeschalteten Funktionalitäten festzustellen. Für dieses Szenario wird es weiterhin nötig sein, das Fachmodul neu zu öffnen.

Es soll für ein Viewmodell eine Möglichkeit geben, die dahinterliegenden Daten des Modellobjekts einzusehen. Um die Testdaten verfügbar zu machen, ist jedes Mockmodell von einer Mock-Datenbank abgeleitet, die eine Liste mit Modellobjekten beinhaltet. Diese Hinterlegung der Testdaten am Viewmodell selbst soll mit einer Listenauswahl in der Testkomponente ermöglicht werden, bei der der aktive Datensatz für das geöffnete Modul geändert werden kann. Bei einem Umschalten muss darauf geachtet werden, dass das Viewmodell neugeladen wird. Ein einfaches Ersetzen der Daten ist nicht möglich, da solche Änderungen als „manuelle“ Anpassung interpretiert wird und beim Verlassen das Fachmodul zum Speichern auffordert.

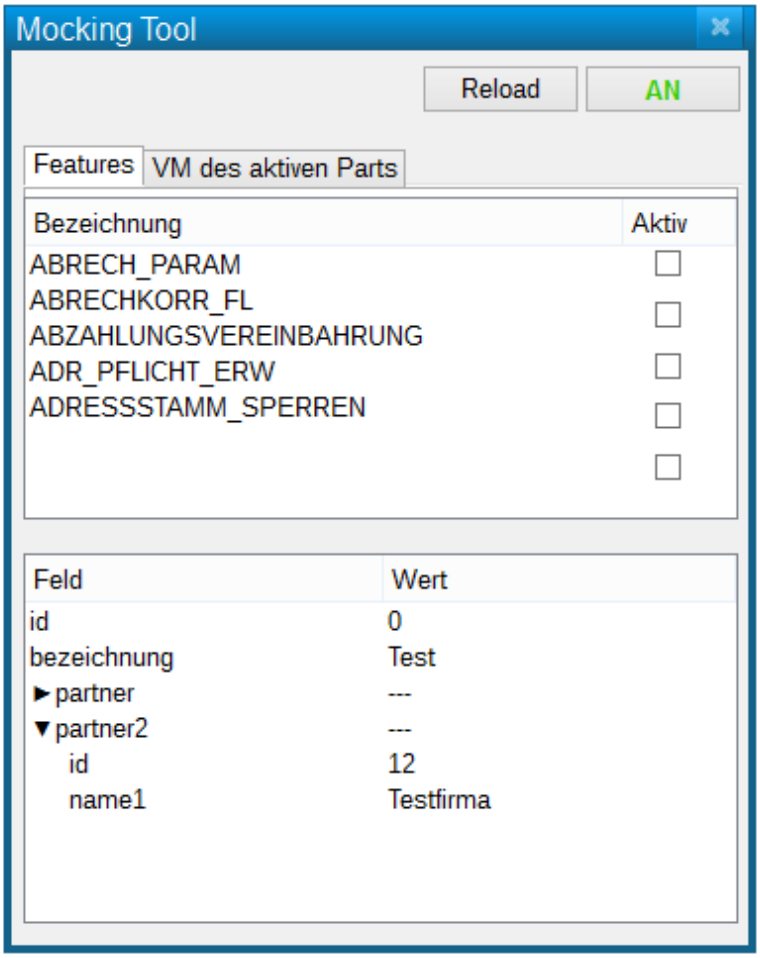


Abbildung 17 Mockup des Testmoduls

Das Testmodul soll kompakt gehalten werden und so wenig wie möglich Platz im Fenster einnehmen, wenn ein Fachmodul des Leasman geöffnet ist. Deshalb soll es sich initial auf der rechten Seite als Sidebar öffnen, aber es soll sich verschieben und in ein separates Fenster auslagern lassen. Die Tabelle mit der Einstellungsmöglichkeit der freigeschalteten Features und die Viewmodellübersicht sollen über Tabs wahlweise sichtbar sein (siehe Abbildung 17).

Die Struktur des Modellobjekts eines Viewmodells wird als Treetable abgebildet und soll immer sichtbar sein. Das Testmodul soll bei der Auswahl eines Viewmodells in der Übersicht die Tabelle mit den Daten zum Modellobjekt füllen. Ebenso soll die Parametrierbarkeit des Viewmodells (Exceptions und Verzögerungen) immer sichtbar sein (nicht in Mockup enthalten), aber auch hier nur mit Daten gefüllt sein, wenn ein Viewmodell ausgewählt ist.

5.2 Serverunabhängige Simulation

Damit die Anwendung unabhängig vom Server betrieben werden kann, muss sie sich auch wie der Server verhalten. Einem entscheidenden Problem, dem hier begegnet werden muss ist die fehlende Kongruenz der Daten mehrerer Mockmodelle. Bsp.: Fachmodul ‚Vertragsanzeige'. Im ersten Schritt des Wizards befindet sich die Vertragssuche, in der Verträge gesucht, gefiltert und abschließend einer ausgewählt werden kann, der dann im zweiten Wizardschritt angezeigt wird. Sowohl die Vertragssuche, also auch die Vertragsanzeige selbst, haben ihr eigenes Viewmodell, welches wiederum ein Modellinterface referenziert.

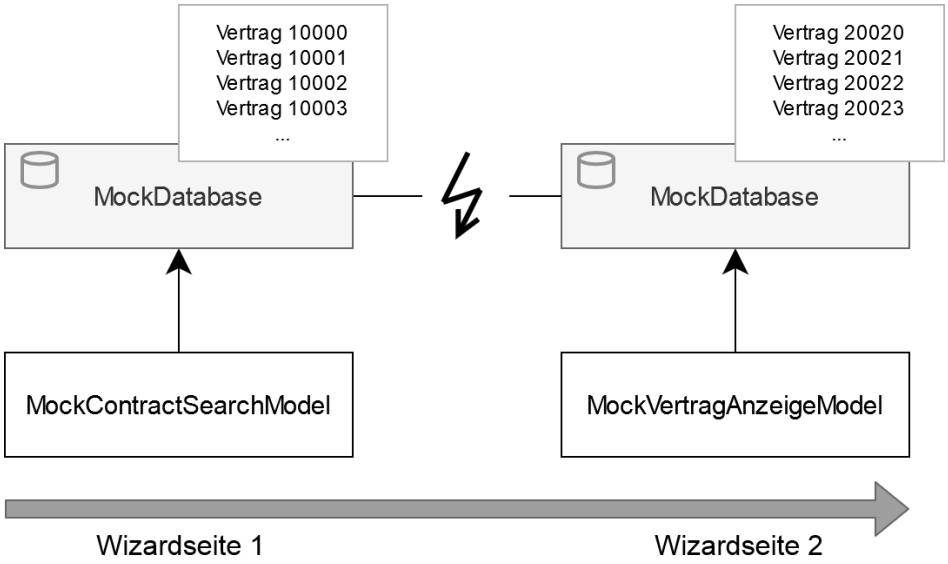


Abbildung 18 Konflikt der verschiedenen Mockdatensätze

Jedes Mockmodell beinhaltet eine eigene Datenbank (einfache Liste mit Datensätzen), wenn es von der passenden Mockmodell-Basisklasse abgeleitet ist. Da diese Dummydaten vom Entwickler selbst festgelegt werden, ist es sehr wahrscheinlich, dass die Daten der beiden Mockdatenbanken nicht zusammenpassen. Also kann in diesem Fall kein Vertrag aus der Vertragssuche in der Vertragsanzeige gefunden und damit auch nicht dargestellt werden (siehe Abbildung 18). Damit dies möglich ist müssen die Daten kontextabhängig zentralisiert werden. Das heißt im konkreten Fall, dass es eine Sammlung von Vertragsnummern geben soll, die in den Mockmodellen genutzt werden dürfen.

6 Implementierung der Testkomponente

6.1 Mockmodus-Schalter

Die Umsetzung kann prinzipiell in seine Teilprobleme unterteil werden. Es muss ein neues Plugin für die DWB erstellt werden, welches alle erforderlichen Funktionalitäten und ein neues Modul zur dynamischen Parametrierung beinhaltet. Das neue Plugin wird im Debugging-Feature freigeschaltet, damit es den Entwicklern und ggf. der Abteilung für Qualitätssicherung zur Verfügung gestellt werden kann, aber nicht an die Kunden ausgeliefert wird. Es soll sich lediglich um eine Komponente zur internen Verwendung handeln, die keine funktionalen und technischen Änderungen an der DWB-Architektur vornimmt. Dieses Modul soll vor allem zur Entwicklungsunterstützung und zum der Testen der Software genutzt werden können.

Im Kernmodul der DWB-Architektur existiert ein Applikationsviewmodell, welchen den anwendungsweiten Zustand speichert und globale Aktionen steuert. Um die Aktivität des Mockmodus abzubilden, kann ein neues Property hinterlegt werden. Da das Applikationsviewmodell ein Singletonobjekt ist, kann dieser Zustand auch nur einmalig zur Laufzeit existieren und bleibt somit nach dem Schließen der Testkomponente vorhanden.

Im Viewmodell der Testkomponente wird ein Listener an das Mockmodus-Property des Applikationsviewmodells gehangen, der je nach Zustand des Flags (an / aus) die Classfinding-Factory an der Modellfactory umschaltet. Diese produziert wiederum je nach Zustand Modelle oder Mockmodelle. Wird bei aktivem Mockmodus ein Fachmodul geöffnet, werden die Daten aus den Mockdatenbanken der Mockmodelle angezeigt.

Es ist allerdings nicht möglich den Modus zu wechseln, während ein Fachmodul geöffnet ist. Das liegt darin begründet, dass die Viewmodelle bei ihrer Instanziierung ihre Modelle aus der Modellfactory beziehen und bei einer Umschaltung keine erneute Instanziierung der Viewmodelle und damit der darin enthaltenen Modelle erfolgt. Es ist zwar prinzipiell möglich durch den Viewmodellbaum zu iterieren und alle Modelle zur Laufzeit zur ändern. Es werden in den Viewmodellen aber auch Modelle bzw. Viewmodelle referenziert, die nicht in diese Hierarchie eingereiht sind und somit beim Ersetzen der Modelle nicht beachtet würden. Somit wäre keine volle Serverunabhängigkeit mehr gegeben.

6.2 Viewmodell-Events

Für das Testmodul ist es notwendig, dass eine Kommunikation mit dem Viewmodell eines anderen Moduls stattfinden kann. Da es die Viewmodell-Hierarchie aktiven

Moduls abbilden soll, muss das Viewmodell der Testkomponente das aktive Modul bzw. sein Viewmodell kennen. Dies kann folgendermaßen erreicht werden: e4 stellt einen Eventbroker zur Verfügung, der innerhalb der Rich Client Platform Events aussendet, die an jeder beliebigen anderen Stelle konsumiert werden können. Somit ist eine Kommunikation von Komponenten und Modulen möglich, ohne eine Abhängigkeit zwischen ihnen zu schaffen. Dieser Broker soll nun beim Öffnen eines Moduls und der damit verbundenen Instanziierung des Viewmodells ein Ereignis aussenden, welches von ihm an anderer Stelle verarbeitet werden kann. Das Event erhält einen eindeutigen Bezeichner (Topic), der hier auf VIEWMODEL/CREATION festgelegt wird. Weiterhin kann dem Event ein ‚Payload', also Daten für Informationen über das eingetretene Ereignis, angehängt werden. Dazu wird hier das instanziierte Viewmodell übergeben. Das Event wird dann im Testmodul konsumiert und das erstellte Viewmodell kann dem Ereignis entnommen und verarbeitet werden.

Im analogen Fall, wenn ein Modul geschlossen wird, soll ebenso ein Event produziert werden, damit sich ein schließendes Viewmodell entsprechend abmelden kann. Dieses Event wird VIEWMODEL/DISPOSAL heißen und bekommt ebenso das Viewmodell als Payload übergeben. Es kann allerdings auch der Fall auftreten, dass mehrere Module geöffnet sind und diese sich an der Testkomponente anmelden. Damit es zu keinen Konflikten kommt, muss das Viewmodell der Testkomponente eine Liste von Viewmodellen verwalten, die geöffnet sind und eine Instanz des aktuell aktiven Viewmodells. Damit muss auch auf das Umschalten des aktiven Moduls (das den Fokus erhält) regiert werden können, damit das aktive Viewmodell im Testmodul immer gesetzt ist. Um dies zu ermöglichen stellt e4 ein Partlistener-Interface zur Verfügung, dass verschiedene Ereignisse aus dem Lebenszyklus des Parts (Moduls) deklariert, auf die reagiert werden kann. Für den hier geschilderten Anwendungsfall muss lediglich die Methode `IPartListener::partActivated` implementiert werden, die beim Ändern des aktiven Parts aufgerufen wird. Dieser Listener wird dann mittels des EPartService registriert. Damit diese Registrierung nicht mehrfach stattfindet, muss darauf geachtet werden, dass der Listener beim Schließen der Testkomponente wieder entfernt wird.

6.3 Modellverarbeitung

Sobald ein Viewmodell zur Liste der Root-Viewmodelle der offenen Module hinzugefügt wird, findet eine Analyse statt. Das Testmodul iteriert durch die Viewmodellhierarchie und erzeugt eine Datenstruktur, die diese Hierarchie abbildet und alle erforderlichen Informationen zu den Viewmodellen speichert. Diese Datenstruktur wird mit einer JavaFX TreeView synchronisiert, um dem Entwickler bzw.

Tester eine adäquate Darstellung bereitzustellen (siehe Abbildung 19). Folgende Informationen werden zu jedem Viewmodell der Hierarchie gespeichert:

- *Viewmodell*: eine Referenz auf das konkrete Viewmodell selbst
- *Wrapped-Viewmodell*: Viewmodelle können lediglich eine ‚Hülle' für andere Viewmodelle sein. Hier wird das ‚verpackte' Viewmodell hinterlegt, sofern vorhanden
- *Load*: Konfigurationsmöglichkeit der Ladeverzögerung und -exception
- *Delete*: Konfigurationsmöglichkeit der Löschverzögerung und -exception
- *Lock*: Konfigurationsmöglichkeit der Sperrverzögerung und -exception
- *Unlock*: Konfigurationsmöglichkeit der Entsperrverzögerung und -exception
- *Insert*: Konfigurationsmöglichkeit der Einfügeverzögerung und -exception
- *Update*: Konfigurationsmöglichkeit der Aktualisierungsverzögerung und -exception

Ein Problem ergibt sich aus der zweiten Information, dem Wrapped-Viewmodell. Dieses kann sich zur Laufzeit ändern, bspw. durch das Umschalten eines Wizardschritts. Damit muss dafür gesorgt werden, dass sowohl die Datenstruktur als auch die TreeView immer mit dem Zustand des aktiven Moduls synchronisiert wird. Das einhüllende Viewmodell beinhaltet ein Property, dass das eingehüllte Viewmodell repräsentiert.

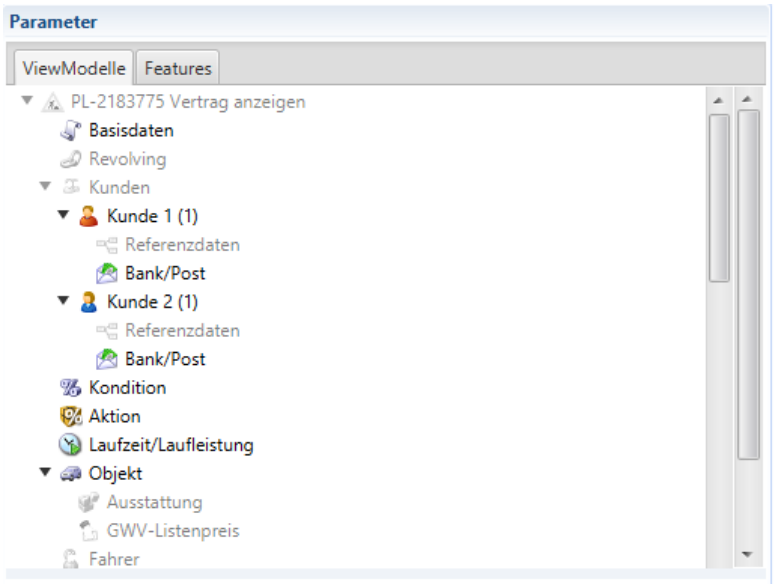


Abbildung 19 Abbildung der Viewmodellhierarchie durch die Testkomponente

Damit kann durch ein einfaches Binding die Synchronizität der Datenstruktur sichergestellt werden. Um die TreeView immer aktuell zu halten wurde ein *RecursiveTreeTreeItem*[39] für diesen speziellen Anwendungsfall eingeführt. Dieses hängt sowohl an das Property des Items, sowie alle Kinder rekursiv Listener, die entsprechend auf eine Änderung der Datenstruktur reagieren und die TreeView aktualisieren.

Bei der Auswahl eines solchen Treeitem sollen auch die viewmodellspezifischen Informationen und Parametrierungsmöglichkeiten sichtbar gemacht werden (siehe Abbildung 20). Dazu nutzt das Testmodul eine Tabelle, die den Tester die Möglichkeit einräumt, Verzögerungen und Exceptions bei bestimmten Modelloperationen einzustellen. Dazu werden die Informationen aus der o.g. Liste verwendet und bei einer Änderung der Parametrierung im dazugehörigen Mockmodell aktualisiert.

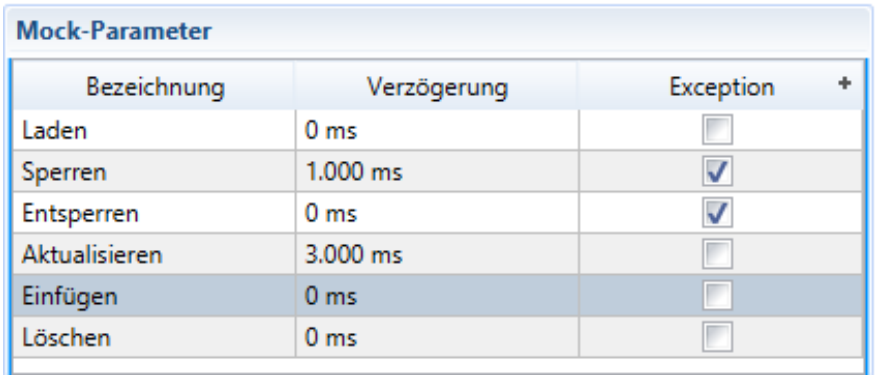


Abbildung 20 Tabelle für die Parametrierung der Mockmodelle

Dabei unterscheidet die Verarbeitung, um welche Modellstufe es sich handelt, da nicht alle Operationen bei jedem Typ vorhanden sind. Tabelle 5 stellt die vorhandenen und parametrierbaren Methoden in den Mockmodelltypen dar.

Bezeichnung der Basisklasse	Methoden
MockLoadableModel	Load
MockParametrizedPersistentModel	Load, Lock, Unlock, Insert, Update
MockPersistentModel	Load, Lock, Unlock, Insert, Update
MockMasterDetailModel	Load[40], Lock, Unlock, Insert, Update, Delete

[39] vgl. online: MAUKY, 2015 (16.08.2020)

[40] Bei den Masterdetail-Modellen wird die einfache load-Methode nicht verwendet, sondern loadMasterList. Von der Intention und der Parametrierung ist es aber das gleiche

MockPeriodMasterDetailModel	Load[40], Lock, Unlock, Insert, Update, Delete, LoadSubset, Finish

Tabelle 5 Verfügbare Operationen der (Mock-)Modelle

Die Parameter müssen immer mit dem Mockmodell synchronisiert werden, d.h. ändert der Tester den Wert in der Tabelle, müssen die Parameter in den korrespondierenden Mockmodellen aktualisiert werden. Da die Modelle durch die Modell-Factory gecacht werden, bleiben die Einstellungen auch nach dem Schließen der Testkomponente und des zu testenden Moduls gesetzt. Daher muss auch bei einem erneuten Öffnen der Testkomponente und Selektieren des Viewmodells die Konfigurationen aus den Mockmodellen ausgelesen und in die Tabelle geschrieben werden. Findet allerdings in dieser Zeit eine Umschaltung zwischen Mock- und Servermodus statt, löscht die Modell-Factory die Modelle und die in ihnen enthaltenen Konfigurationen aus dem Cache.

6.4 Verarbeitung eines Modellobjekts

Der Zustand eines Viewmodells in der DWB wird zum großen Teil durch das Modellobjekt bestimmt. Dieses stellt eine Ansammlung von Propertys dar, die in der Regel von der Modellebene beschafft werden. Es kann nun zum Debugging hilfreich sein, welchen aktuellen Zustand ein Modellobjekt aufweist.

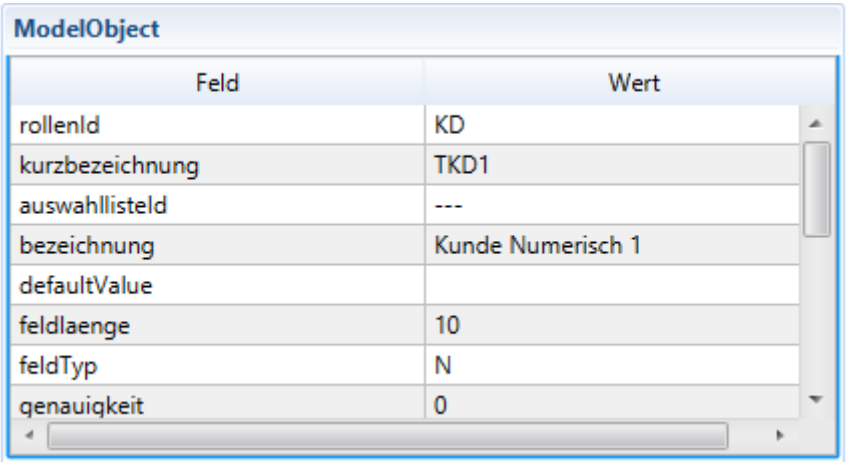


Abbildung 21 Inspektion des Modellobjekts

Dazu kann sich mit der Möglichkeit der Reflexion beholfen werden. Dies ermöglicht die Struktur von Klassen zu analysieren und die Werte ggf. zu ändern. Dies wird in diesem konkreten Fall auf das Modellobjekt angewandt, in dem alle Felder und deren gegenwärtiger Wert bei der Auswahl eines Viewmodells ermittelt werden. Diese werden in Form einer Tabelle dargestellt (siehe Abbildung 21). Da Modellobjekte

verschachtelt sein können, also ein MO ein anderes beinhaltet, durchläuft der Algorithmus zum Ermitteln der Felder und Werte das Modellobjekt rekursiv. Eine Änderungsmöglichkeit der Werte ist momentan nicht vorgesehen, da dies vom Fachmodul als Datenänderung interpretiert wird und zum Speichern auffordert wird.

6.5 Steuerung aktiver Features

Um die vorhandenen und aktiven Features abzubilden und parametrierbar zu gestalten, wird eine Tabelle genutzt, die den Bezeichner und die Aktivität anzeigt (siehe Abbildung 22). Durch eine Checkbox lässt sich ein Feature an- und abschalten. Da sich freigeschaltete Features zur Laufzeit nicht ändern, sind diese auch nicht als Propertys in den Viewmodellen hinterlegt, sondern werden über das Parametermodell bei der Instanziierung des Viewmodells abgefragt. Somit lassen sich die unmittelbaren Auswirkungen einer Aktivierung bzw. Deaktivierung eines Features bei geöffneten Modulen nicht nachvollziehen. Dazu muss das Fachmodul erneut geöffnet werden, um die Wirksamkeit der Änderungen festzustellen. Ein Screenshot vom vollständigen Testmodul und dem geöffneten Fachmodul ‚Vertragsanzeige' befindet sich im Anhang 4.

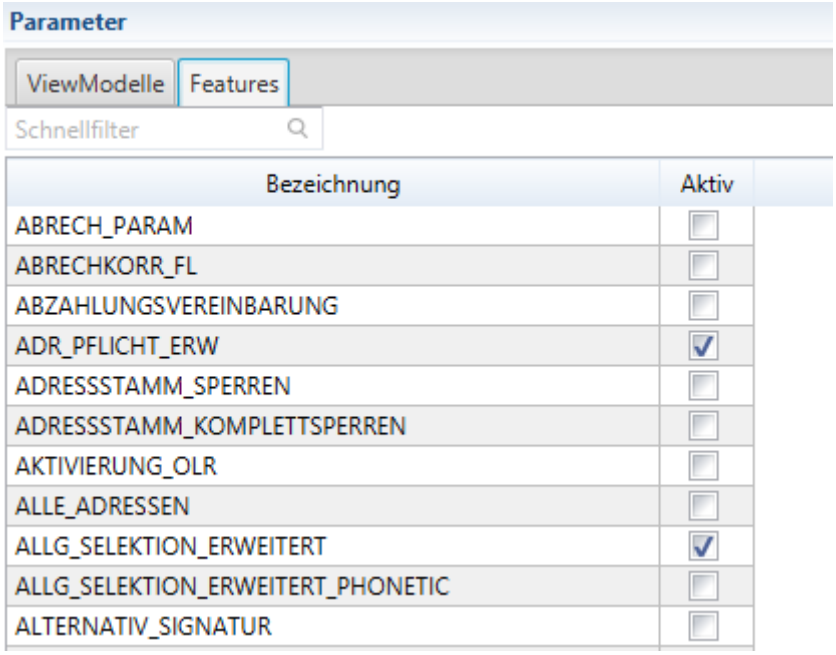


Abbildung 22 Parametrierbarkeit der Leasman-Features

6.6 Erweiterungs- und Optimierungsmöglichkeiten

6.6.1 Bereitstellung der Testdaten in strukturierter Form

Die gegenwärtige Beschaffenheit der Mock-Architektur erlaubt nur die Testdaten in der Mockimplementierung des Modells zu hinterlegen. Dadurch sind sie allerdings schwer auffindbar und schlecht lesbar. Besser wäre es sie in einer strukturierten Form auf der Festplatte zu speichern und zur Laufzeit vom Mockmodell laden zu lassen. Damit dies umsetzbar ist muss es möglich sein die Daten aus den Modellobjekten zu extrahieren und in einer strukturierten Form zu serialisieren.

Allerdings ist ein Modellobjekt von sich aus nicht serialisierbar, da die Propertys wiederum nicht serialisierbar sind. Das liegt darin begründet, dass es keine standardisierte Logik gibt Propertys in eine sequenzielle Form zu bringen. Denn es soll nur der aktuelle Wert gespeichert werden und keine evtl. an dem Property hängenden Listener. Daher müsste eine eigene Serialisierungslogik für die Modellobjekte entworfen werden.

Das geläufigste Format für eine Speicherung ist das XML-Format, dass aber bei einer großen Datenmenge unübersichtlich werden kann. Es soll je nach Anwendungsfall möglich sein, die Testdaten in der Datei vor einem Test anzupassen. Dies wird erschwert, wenn das Format nicht einfach zu lesen ist. Weiterhin existiert das JSON-Format, dass allerdings eher für die Webentwicklung als Datenaustauschformat dient und somit keine praktikable Anwendung für dieses Problem bietet. Ein weiteres Format ist YAML, dass einen guten Kompromiss zwischen Maschinen- und Menschenlesbarkeit bietet. In Abbildung 23 wird ein Beispiel mit den Dateiformaten XML und YAML gezeigt. JSON wurde aus Platzgründen weggelassen.

Abbildung 23 Exemplarischer Vergleich XML – YAML

6.6.2 Zwischenspeicherung der Mockdatenbank

Aufbauend darauf, kann die Serialisierung der Daten eines Mockobjektes auch einer weiteren Optimierung dienen. In der aktuellen Implementierung der Mockmodelle ist es so, dass die Mockimplementierung die Daten der Mockdatenbank initialisiert. Wird eine Ladeaktion am Viewmodell aufgerufen führt dies wiederum zum Aufruf der Ladeoperation am Mockmodell. Dieses ermittelt anhand des übergebenen Parameters (sofern vorhanden) den erforderlichen Datensatz aus der Mockdatenbank oder gibt einen Fehler zurück. Ist die Operation allerdings erfolgreich, dann wird nicht das Modellobjekt aus der Datenbank in das Viewmodell kopiert, sondern es wird der Wert jedes Properties kopiert (tiefe Objektkopie). Das führt dazu, dass bei einem erfolgreichen Ladevorgang eine doppelte Datenhaltung vorzufinden ist – im Viewmodell und im Mockmodell (siehe Abbildung 24). Bei wenigen Daten ist das unproblematisch, aber wenn mehrere Tausend Datensätze geladen werden, dann kann dies zu Performanceeinbußen führen.

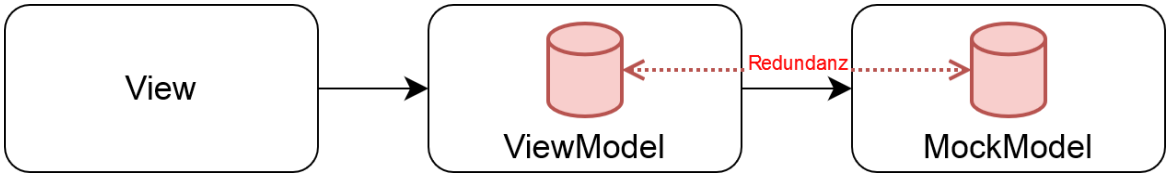


Abbildung 24 Datendopplung im Mockmodus

Deshalb sollten die Daten nur beim Aufruf der Modellmethoden im Hauptspeicher sein. Wenn keine Interaktionen des Viewmodells mit dem Modell stattfinden, ist es hier günstiger die Daten als temporäre Datei auf der Festplatte zu speichern. Da auch hier das Problem mit der fehlenden Serialisierbarkeit des Modellobjekts besteht, kann die Logik zum Extrahieren der Daten wiederverwendet werden. Hier ist es allerdings nicht nötig die Daten in strukturierter Form zu speichern, sondern als Binärdatei.

6.6.3 Manuelle Auswahl des Mockdatensatzes

Bei der Initialisierung der Mockmodelle ist es mögliche mehrere Testdatensätze festzulegen. Wenn nun ein Aufruf der Ladeoperation am Modell stattfindet, wird der Datensatz zum passenden Parameter zurückgegeben. Es ist allerdings auch sinnvoll andere Testdaten zur Laufzeit festzulegen unabhängig vom Ladeparameter. Dazu müsste das Standardverhalten (Ermittlung des Datensatzes über den Parameter) überschrieben werden und der Wert zurückgegeben werden, der zur Auswahl des Testers passt. Die Auswahl kann über eine Combobox erfolgen, die die Liste der möglichen Datensätze beinhaltet. Bei einer Auswahl wird das Viewmodell neugeladen und der ausgewählte Datensatz in das Viewmodell geladen.

7 Zusammenfassung

In dieser Arbeit wurde das gezeigt, dass sich das Konzept von Mockobjekten in automatischen Unittests auch auf manuelle Tests bis hin zu einer dynamisch parametrierbaren Simulation einer komplexen Geschäftsanwendung übertragen lässt. Zu Beginn wurden verschiedene Architekturentwurfsmuster für grafische Benutzeroberflächen miteinander verglichen. Dabei wurde ein besonderer Fokus auf die Testbarkeit der unterschiedlichen Paradigmen gesetzt. Es konnte gezeigt werden, dass je weniger Abhängigkeiten zwischen den Komponenten der Architektur bestehen, desto besser ist die Testbarkeit der einzelnen Bestandteile. Dies liegt darin begründet, dass Komponententests einzelne Komponente in Isolation testen und keine bzw. wenige Nebeneffekte produzieren sollen. Das lässt sich am besten mit dem Model-View-Viewmodel-Architekturmuster realisieren, da es die geringsten Abhängigkeiten der Komponenten untereinander aufweist.

Im Anschluss wurde die Wichtigkeit von Tests im Softwareentwicklungsprozess näher beleuchtet. Dabei wurden einige Vorgehensmodelle bei der Softwareentwicklung und deren Position von Tests in diesen beschrieben. Anschließend wurden die verschiedenen Teststufen des allgemeinen V-Modells näher erläutert, mit Fokus auf den Komponententest und die Möglichkeit Abhängigkeiten von Komponenten künstlich zu reduzieren, um eine isolierte Testbarkeit zu erreichen. Hier kommen die verschiedenen Arten von Testdoubles zum Einsatz, die Abhängigkeiten von Komponenten imitieren. Es wurden die Unterschiede zwischen Dummys, Stubs, Spys, Mocks und Fake-Objekten erläutert und in welchen Situationen sie zum Einsatz kommen. Darauf aufbauend wurde der Unterschied zwischen der Zustandsverifikation in einfachen Unittests und der Verhaltensverifikation in mock-basierten Unittests eingegangen.

Im Anschluss wurde die Architektur der DELTA Workbench in Ansätzen beschrieben und gezeigt, dass eine serverunabhängige Tertbarkeit gegeben ist. Es wurde insbesondere auf die Umsetzung der MVVM-Architektur und der damit verbundenen Möglichkeit, die Modelle auszutauschen. Darauf aufbauend sollte untersucht werden, inwiefern sich die Austauschbarkeit und die Parametrierung der Modelle zur Laufzeit realisieren lässt.

Es wurde ein eigenständiges Modul innerhalb der DWB entworfen und implementiert, welches den Client vom Server abkoppelt und Mock- statt Serverdaten verwendet. Es ermöglicht zudem eine Inspektion der Viewmodelle und eine dynamische Parametrierung der Mockmodelle. Dazu wird die Viewmodell-Hierarchie des zu untersuchenden Moduls analysiert und in adäquater Form dargestellt. Zudem wird

dem Tester eine Einstellungsmöglichkeit gegeben, mit der er schwer zu testende Anwendungsfälle simulieren kann, bspw. Antwortverzögerung oder Fehler des Servers. Somit können Testfälle durchgeführt werden, die in einer realen Umgebung nicht durchführen lassen und zudem zu keinen Änderungen in der Datenbank führen, da die Dummydaten clientseitig verwaltet werden.

Um eine vollständige Simulation der Anwendung zu ermöglichen, müssen zusammenhängende Daten auch zusammenpassen. Dies wurde exemplarisch am Fachmodul ‚Vertragsanzeige' demonstriert. Es wurde gezeigt, dass jedes Mockmodell seine eigene Liste von Testdatensätze beinhaltet und es daher keine Garantie gibt, dass zwei Mockmodelle passende Datensätze enthalten. Daher müssen zumindest die Ladeparameter der Testdatensätze zentral gehalten werden und nur diese dürfen beim Implementieren der Testfälle genutzt werden.

Zum Schluss wurden noch einige Verbesserungs- und Erweiterungsmöglichkeiten erörtert, um die Testbarkeit und Parametrierung zu erleichtern. Unter anderem konnte gezeigt werden, dass es vorteilhafter ist die Testdaten nicht im Quelltext, sondern in strukturierter Form auf der Festplatte zu hinterlegen, da sie leichter zu finden und einen besseren Überblick bieten. Außerdem wurde darauf hingewiesen, dass nach einem Laden der Daten aus der Mockdatenbank die Daten doppelt vorhanden sind. Einmal als Datensatz der Mockdatenbank selbst und als Arbeitskopie im Viewmodell. Daher kann es bei großen Datenmengen aus performancegründen besser sein, die Mockdatenbank auf der Festplatte zwischenzuspeichern. Außerdem könnte eine Möglichkeit zum Umschalten des Datensatzes implementiert werden, die den Datensatz unabhängig vom Ladeparameter ermittelt. Somit hätte der Tester volle Kontrolle darüber, welche Daten er angezeigt bekommt.

Literaturverzeichnis

ACHARYA, Sujoy: Mockito essentials – A practical guide to get you up and running with unit testing using Mockito. Birmingham, UK, 2014

ARTHORNE, John: How You've Changed! – Responding to resource changes in the Eclipse workspace, 2002. In: https://www.eclipse.org/articles/Article-Resource-deltas/resource-deltas.html (08.08.2020)

DELTA proveris AG: DWB Entwicklerdokumentation, 2020 (unveröffentlicht)

EASYMOCK: User Guide, 2020. In: https://easymock.org/user-guide.html (10.08.2020)

EPPLE, Anton: JavaFX 8 – Grundlagen und fortgeschrittene Techniken. 1., korrigierter Nachdruck. Heidelberg, 2016

FOWLER, Martin: Observer Synchronization – Synchronize multiple screens by having them all be observers to a shared area of domain data., 2004. In: https://martinfowler.com/eaaDev/MediatedSynchronization.html (06.08.2020)

FOWLER, Martin: TestDouble, 2006. In: https://martinfowler.com/bliki/TestDouble.html (14.08.2020)

FOWLER, Martin: Mocks Aren't Stubs, 2007. In: https://martinfowler.com/articles/mocksArentStubs.html (29.07.2020)

FOWLER, Martin; RICE, David: Patterns of enterprise application architecture. 17. print. Boston, Massachusets, 2011

FRIEDRICH, Pit: Integration UI-basierter Unittests mit JavaFX in ein bestehendes Testframework, 2019 (unveröffentlicht)

GAMMA, Erich u. a.: Design patterns – Elements of resuable object-oriented software. England, 1995

GOLL, Joachim: Architektur- und Entwurfsmuster der Softwaretechnik – Mit lauffähigen Beispielen in Java. 2., aktualisierte Aufl. Wiesbaden, 2014

jMock, 2019. In: http://jmock.org (10.08.2020)

KOURAKLIS, John: MVVM in Delphi – Architecting and building model view ViewModel applications. Place of publication not identified, 2016

MARTIN, Robert C.: Agile software development, principles, patterns, and practices. 1. ed., Pearson new international ed. Harlow, 2014

MAUKY, Manuel: JavaFX: Data Model for TreeTable, 2015 (16.08.2020)

MICROSOFT: The MVVM Pattern, 2012 (10.08.2020)

Mockito - Webseite, 2020. In: https://site.mockito.org/ (10.08.2020)

Mockito 3.4.6 API, 2020. In: https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html (10.08.2020)

NORTH, Dan: Behavior Modification – The evolution of behavior-driven development. In: Better Software Magazine, Jg. 2006, 2006, H. 3, S. 27 - 31

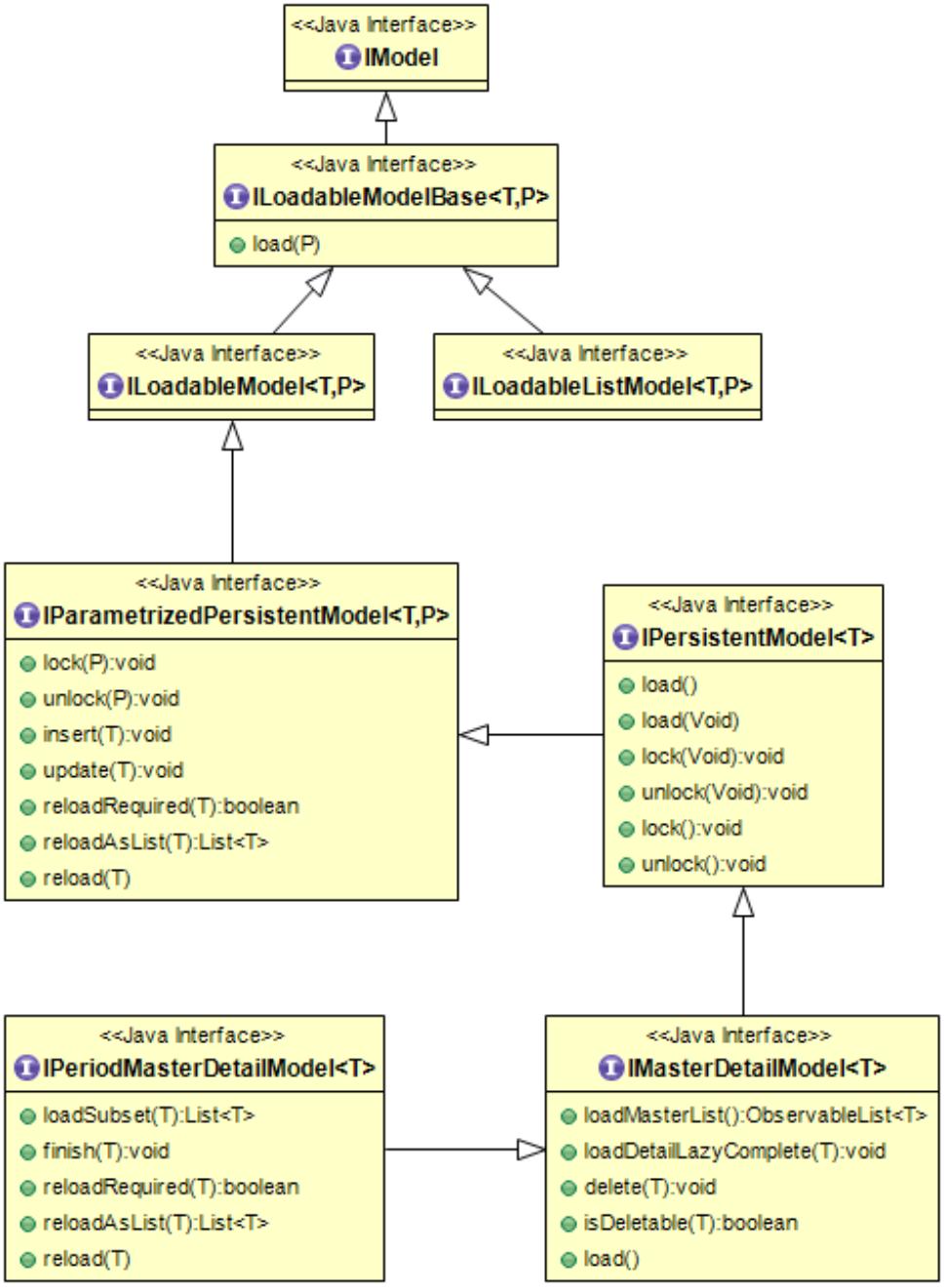
SPILLNER, Andreas; LINZ, Tilo: Basiswissen Softwaretest – Aus- und Weiterbildung zum Certified Tester : Foundation Level nach ISTQB-Standard. 6., überarbeitete und aktualisierte Auflage, 2019

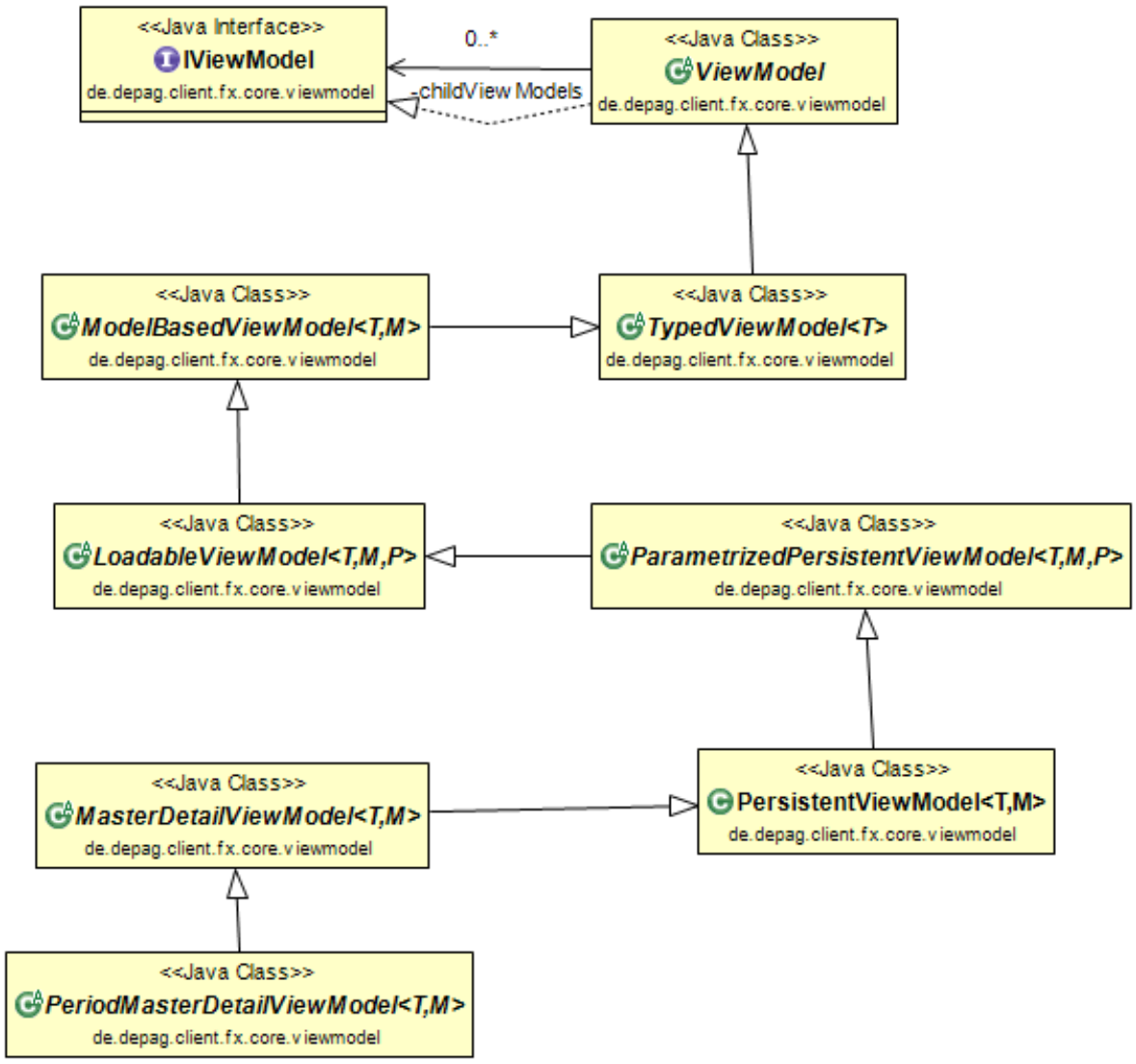
VITZ, Michael: Dynamische Proxys mit dem JDK umsetzen – Der dynamische Stellvertreter, 2020. In: https://www.innoq.com/de/articles/2020/03/java-dynamic-proxy/ (20.07.2020)

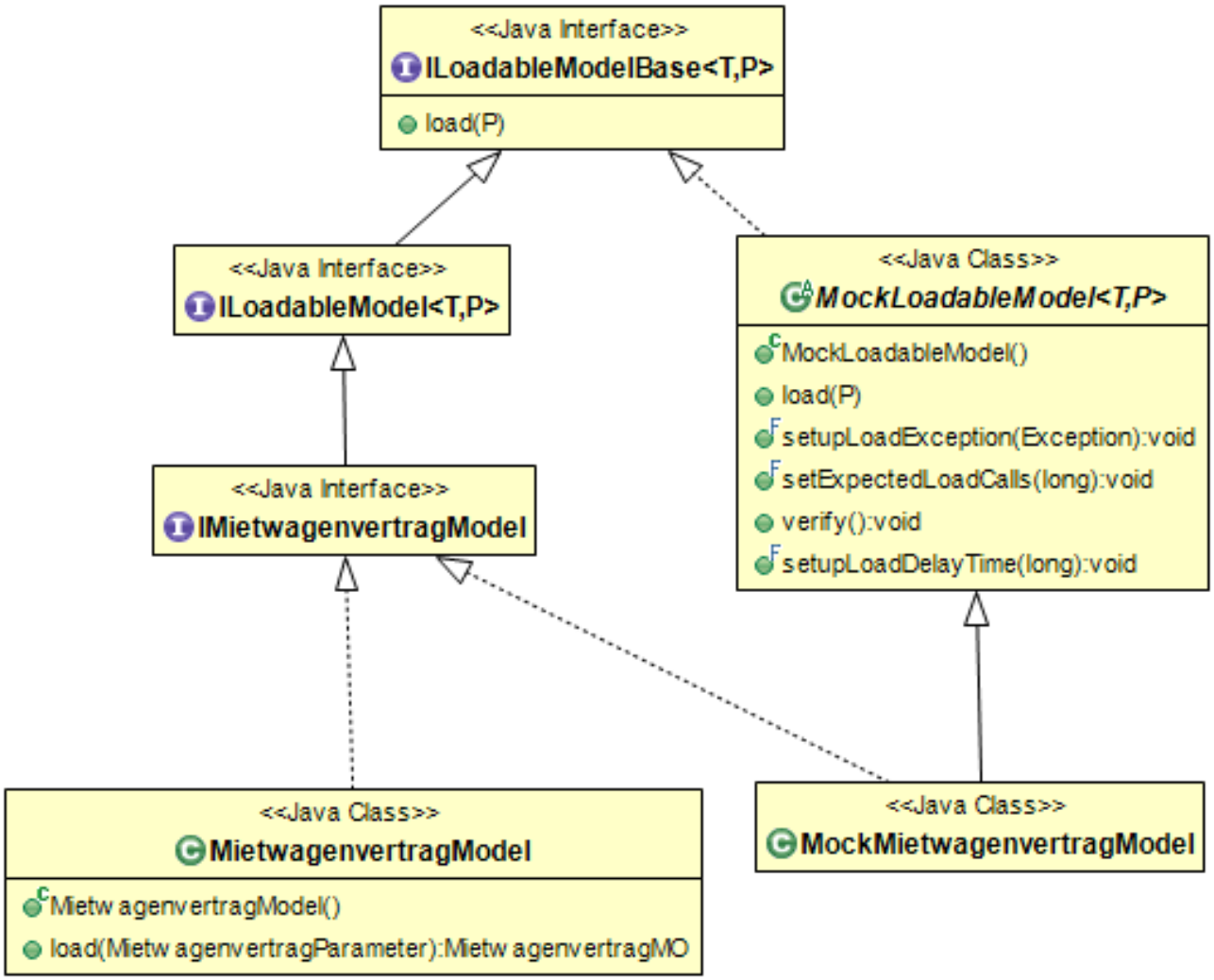
WESTPHAL, Frank: Testgetriebene Entwicklung - mit JUnit & FIT – Wie Software änderbar bleibt. 1. Aufl. Heidelberg, 2006

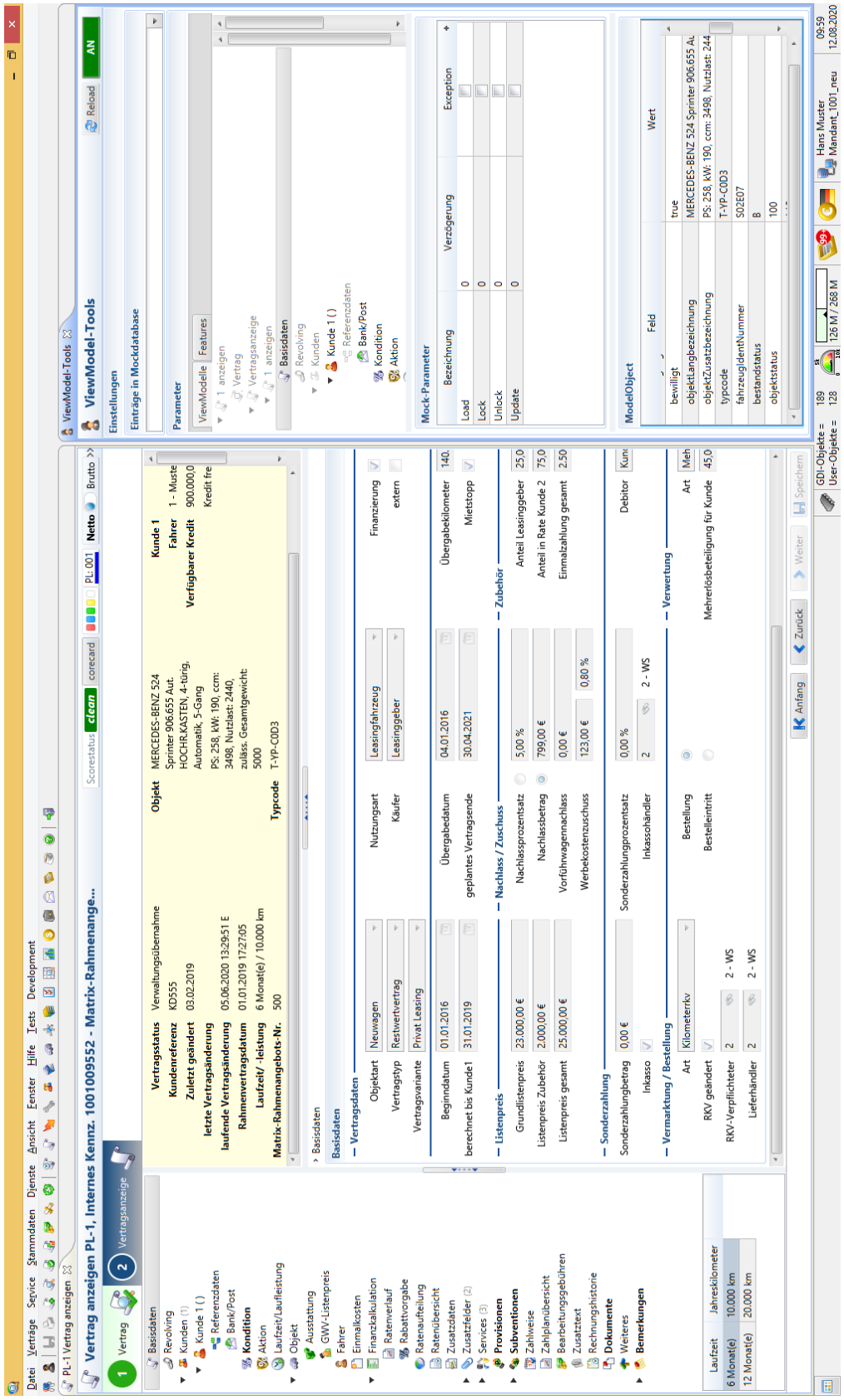
Anhangsverzeichnis

Anhang 1 Modell-Hierarchie der DELTA Workbench 52
Anhang 2 Viewmodell-Hierarchie der DELTA Workbench..... 53
Anhang 3 Zusammensetzung eines Modells / Mockmodells 54
Anhang 4 Screenshot der Testkomponente 55


<<Java Interface>>
IModel
<<Java Interface>>
ILoadableModelBase<T,P>
load(P)
<<Java Interface>>
ILoadableModel<T,P>
<<Java Interface>>
ILoadableListModel<T,P>
<<Java Interface>>
IParametrizedPersistentModel<T,P>
lock(P):void
unlock(P):void
insert(T):void
update(T):void
reloadRequired(T):boolean
reloadAsList(T):List<T>
reload(T)
<<Java Interface>>
IPersistentModel<T>
load()
load(Void)
lock(Void):void
unlock(Void):void
lock():void
unlock():void
<<Java Interface>>
IPeriodMasterDetailModel<T>
loadSubset(T):List<T>
finish(T):void
reloadRequired(T):boolean
reloadAsList(T):List<T>
reload(T)
<<Java Interface>>
IMasterDetailModel<T>
loadMasterList():ObservableList<T>
loadDetailLazyComplete(T):void
delete(T):void
isDeletable(T):boolean
load()


<<Java Interface>>
IViewModel
de.depag.client.fx.core.viewmodel
0..*
childView Models
<<Java Class>>
ViewModel
de.depag.client.fx.core.viewmodel
<<Java Class>>
ModelBasedViewModel<T,M>
de.depag.client.fx.core.viewmodel
<<Java Class>>
TypedViewModel<T>
de.depag.client.fx.core.viewmodel
<<Java Class>>
LoadableViewModel<T,M,P>
de.depag.client.fx.core.viewmodel
<<Java Class>>
ParametrizedPersistentViewModel<T,M,P>
de.depag.client.fx.core.viewmodel
<<Java Class>>
MasterDetailViewModel<T,M>
de.depag.client.fx.core.viewmodel
<<Java Class>>
PersistentViewModel<T,M>
de.depag.client.fx.core.viewmodel
<<Java Class>>
PeriodMasterDetailViewModel<T,M>
de.depag.client.fx.core.viewmodel


<<Java Interface>>
ILoadableModelBase<T,P>
load(P)
<<Java Interface>>
ILoadableModel<T,P>
<<Java Class>>
MockLoadableModel<T,P>
MockLoadableModel()
load(P)
setupLoadException(Exception):void
setExpectedLoadCalls(long):void
verify():void
setupLoadDelayTime(long):void
<<Java Interface>>
IMietwagenvertragModel
<<Java Class>>
MietwagenvertragModel
Mietw agenvertragModel()
load(Mietw agenvertragParameter):Mietw agenvertragMO
<<Java Class>>
MockMietwagenvertragModel


Vertrag anzeigen PL-1, Internes Kennz. 1001009552 - Matrix-Rahmenange...
ViewModel-Tools
Einstellungen
Einträge in Mockdatabase
Parameter
Mock-Parameter
ModelObject
Vertragsdaten
Listenpreis
Nachlass / Zuschuss
Zubehör
Sonderzahlung
Verwertung
Vermarktung / Bestellung
Laufzeit
Jahreskilometer
6 Monat(e)
10.000 km
12 Monat(e)
20.000 km

Ehrenwörtliche Erklärung

„Ich erkläre hiermit ehrenwörtlich“,

1. dass ich meine Bachelorthesis mit dem Thema „Entwurf und prototypische Implementierung einer serverunabhängigen und flexibel zur Laufzeit parametrierbaren Testkomponente für eine bestehende Clientanwendung“ ohne fremde Hilfe angefertigt habe,

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe und

3. dass ich meine Bachelorthesis bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

_____ _____

Ort, Datum Unterschrift