VAWi.de
Virtueller Weiterbildungsstudiengang
Wirtschaftsinformatik Master of Science

Automatisierte Tests in agilen Softwareprojekten erklärt an einem Beispiel

Projektarbeit

Im Virtuellen Weiterbildungsstudiengang Wirtschaftsinformatik

Verfasser: Tobias Tatsch
Matrikelnummer: 1723304
2. Fachsemester
Dompropststr. 36
91056 Erlangen

Betreuer: Prof. Dr. Adelsberger
Universität Duisburg-Essen
Wirtschaftsinformatik
Universitätsstr. 7
45141 Essen

Abgabe: 16.08.2013
Sommersemester 2013

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

CSS3	Cascading Style Sheets 3
DAO	Data Access Object
HMTL5	Hypertext Markup Language 5
TDD	Test-Driven Development

Code-Ausschnitte

1 Einleitung

Je später im Verlauf eines Softwareprojektes ein Fehler gefunden wird, desto teurer ist dessen Behebung. Dies ist ein wichtiger Grund für die stark gestiegenen Kostenanteil der Software gegenüber dem Anteil der Hardware an einem Gesamtsystem. In Softwareprojekten, die mit Hilfe von traditionellen wasserfallartigen Vorgehensmodellen durchgeführt werden, wird die Software erst nach Abschluss der Realisierungsphase getestet. Somit werden Fehler erst sehr spät gefunden. Vgl. [Lig09, S. 1]

Bei agilen Vorgehensmodellen wird die Software nicht in einer langen Realisierungsphase entwickelt, sondern in mehreren kleinen Perioden von zwei bis sechs Wochen. Nach jeder Periode werden die bis dahin realisierten Funktionen getestet. Dadurch werden Fehler früher gefunden und können kostengünstiger behoben werden. Jedoch muss in jeder Iteration sichergestellt werden, dass neu hinzugekommene bzw. veränderte Funktionen keine Regressionsfehler verursachen. Der Aufwand für das Testen der Anwendung steigt somit von Iteration zu Iteration. Die Automatisierung von Tests kann diesen Aufwand reduzieren und so ein effizienteres Testen ermöglichen. Vgl. [Lig09, S. 192ff]

Ziel der Arbeit ist es, anhand eines Anwendungsbeispiels zu prüfen, ob die Ansätze aus der Literatur für die Testautomatisierung tatsächlich umgesetzt werden können. Dazu soll die Projektarbeit folgende Fragen beantworten:

1. Welche Testarten gibt es und wie werden diese voneinander abgegrenzt?
2. Wie kann die Realisierung dieser Tests in einem agilen Softwareentwicklungsprozess integriert werden?
3. Gibt es für die Automatisierung der Tests Frameworks?

2 Testautomatisierung

2.1 Vorgehensmodell

Um die in der Einleitung gestellten Fragen beantworten zu können, wird vorab Recherchiert welche Testarten für die Automatisierung grundsätzlich geeignet sind. Die gefundenen Testarten werden im Abschnitt 2.4 erläutert. Nachdem geklärt ist welche Testarten es gibt, wird recherchiert wie die Automatisierung der Tests in einen agilen Softwareentwicklungsprozess integriert werden kann. Eine weitere wichtige Voraussetzung für die Automatisierung ist das vorhanden sein von Frameworks mit denen automatisierte Tests effizient implementiert werden können. Dementsprechend werden anschließend Frameworks für die Automatisierung von Tests untersucht. Abschließend wird an einem Anwendungsbeispiel gezeigt, wie ein Test mit Hilfe eines Frameworks automatisiert werden kann. Die Anwendungsbeispiele sind über die web-basierte Versionsverwaltung *GitHub*[1] veröffentlicht. Der Quellcode sowie eine kompilierte Version der Anwendungsbeispiele können über die folgende URL heruntergeladen werden.

https://github.com/iface06/factoryCanteen

Eine Installationsanleitung befindet sich in der Datei *readme.md*[2].

2.2 Anforderungen an die Beispielanwendung

Kernstück der Projektarbeit sind die Anwendungsbeispiele. Damit die Anwendungsbeispiele möglichst praxisnah sind, werden diese auf Basis der Anforderungen für ein

[1] https://github.com/iface06/factoryCanteen
[2] http://git.io/XyLCTw

Informationssystem realisiert. Die Anforderungen stammen aus einer Übungsaufgabe, die in der Vorlesung *Objektorientierte Systementwicklung in Java* im Wintersemester 2012/13 gestellt wurde. Im Folgenden werden die Anforderungen an das Informationssystem erläutert.

2.2.1 User Stories zur Beispielanwendung

Eine gängige Methode zur Beschreibung der Anforderungen in agilen Vorgehensmodellen sind User Stories. Sie dienen dazu die Anforderungen an eine Software aus Sicht des Kunden zu formulieren und zu sammeln. Als Vorlage dient hierfür:

> Ich als <*Rolle der Person/Benutzer*> möchte <*Anforderung*>, um <*Begründung*>.

Vgl. [Mye04, S. 98ff]

User Story 1 Ich als Küchenchef möchte, dass die Anwendung einen Speiseplan für den Zeitraum von drei Wochen erstellt. Damit soll sichergestellt werden, dass der Einkauf ggf. Mengenrabatte ausnutzen kann. Bei der Erstellung kann immer davon ausgegangen werden, dass die benötigen Zutaten in ausreichender Menge verfügbar sind.

User Story 2 Ich als Küchenchef möchte, dass pro Werktag drei Gerichte angeboten werden. Ein vegetarisches, ein Fleischgericht und eine weitere Alternative (Vegetarisch, Fleisch oder Fisch), um sicherzustellen, dass unserer Gäste immer die Wahl zwischen einem vegetarischen Gericht und einem Fleichgericht haben.

User Story 3 Ich als Küchenchef möchte, dass es mindestens einmal in der Woche ein Fischgericht gibt, am besten Freitags, um die Freitags-Fisch-Tradition aufrecht zu erhalten.

User Story 4 Ich als Küchenchef möchte, dass innerhalb der drei Wochen es keine Wiederholungen von Gerichten gibt, um so ein abwechslungsreiches Angebot zu bieten, wodurch die Zufriedenheit unserer Gäste hoffentlich steigt.

User Story 5 Ich als Küchenchef möchte, dass die Hitliste-Datei bei der Auswahl der Gerichte berücksichtigt wird. Dadurch soll die Zufriedenheit unserer Gäste weiter gesteigert werden.

User Story 6 Ich als Gast möchte, dass der Speiseplan der aktuellen Woche in unserem Intranet veröffentlicht wird, um mich vor meinen Besuch in der Kantine über das Angebot zu informieren.

Anhand der User Stories lässt sich bereits ein UML-Klassendiagramm erstellen, mit denen die Business-Objekte und ihre Beziehungen dargestellt werden können. Die Abb. 2.1 zeigt die Business-Objekte und ihre Beziehungen zueinander. Ein Menü besteht aus fünf Tagen mit jeweils 3 angebotenen Gerichten. Jedes Menü ist einer Kalenderwoche zugeordnet.

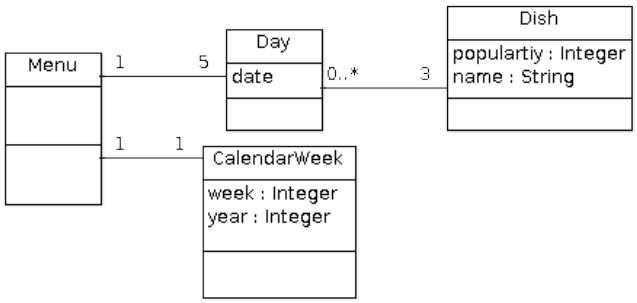

Abb. 2.1: 3-Schichten-Model (eig. Darstellung)

In Abb. 2.2 werden die Anwendungsfälle gezeigt, die mit der Anwendung unterstützt werden sollen. Der Anwendungsfall *Speiseplan erstellen* ergibt sich aus der User Story 1 bis 5. Der Anwendungsfall *Aktuellen Speiseplan einsehen* ergibt sich aus der User Story 6.

Abb. 2.2: Anwendungsfälle der Beispielanwendung (eig. Darstellung)

2.2.2 Akzeptanztests zur Beispielanwendung

Zu den User Stories werden sogenannte Akzeptanztests formuliert. Mit diesen Tests soll der Kunde prüfen können ob die User Stories wie erwartet realisiert wurden. Für die User Stories 1 bis 6 wurden folgende Akzeptanztests formuliert.

Akzeptanztest User Story 1 Nach dem Ausführen der Funktion *Speiseplan erstellen*, ist jeweils ein Speiseplan pro Woche für die nächsten drei aufeinander folgenden Wochen für die Gäste hinterlegt.

Akzeptanztest User Story 2 Die Anwendung soll für jeden Werktag (Montag - Freitag) drei Gerichte anbieten. Eines davon muss ein Fleischgericht sein, ein weiteres muss vegetarisches sind und das Dritte soll als Alternative angeboten werden.

Akzeptanztest User Story 3 Jeden Freitag offeriert die Anwendung ein Fischgericht. Jedoch auch ein Fleischgericht und ein vegetarisches Gericht.

Akzeptanztest User Story 4 Es dürfen sich keine Gerichte innerhalb der drei Wochen wiederholen.

Akzeptanztest User Story 5 Bei der Auswahl der Gerichte wird immer das beliebtere Gericht ausgewählt. Jedoch dürfen die vorherigen Kriterien nicht unberücksichtigt bleiben. Letztendlich muss der Speiseplan vollständig sein ggf. werden dazu auch weniger beliebte Gerichte angeboten.

Akzeptanztest User Story 6 Nach dem Aufruf der Intranetseite wird der Speiseplan der aktuellen Kalendarwoche angezeigt.

2.3 Architektur der Beispielanwendung

Auf Basis der Anforderungen wurde ein geeigneter Ansatz für die Architektur der Anwendung gesucht. In dem Buch [Mar09a] beschreibt Robert C. Martin, dass ein umfangreiches Design (Big Design Up Front) nicht sinnvoll ist. Sollte im späteren

Verlauf des Projektes eine Anpassung der Architektur erforderlich sein, wird diese ggf. nicht umgesetzt, da der Aufwand zu groß wäre. Dadurch besteht die Gefahr, dass dem Kunden nicht die optimale Lösung geliefert werden kann. Sein Vorschlag ist daher, mit einer einfachen Architektur zu starten. Die im Verlauf des Projektes erweitert werden kann. Eine gute Architektur im Sinne des Autors ermöglicht die Entkopplung der einzelnen Module voneinander. Vgl. [Mar09a, S. 166ff]

Krik Knoernschild beschreibt in seinem Buch [Kno12], dass Anwendungen in denen Module eng gekoppelt sind nur schwer erweitert und gewartet werden können. Daraus folgert er, dass die Anwendung durch Anpassungen im Projektverlauf fehleranfällig wird. Die enge Koppelung von Modulen soll sich darüber hinaus negativ auf die Realisierung von automatisierten Tests auswirken. Hintergrund ist, dass Module unabhängig von anderen Modulen testbar sein sollen. Dies ist einer Anwendung nicht möglich, in der die einzelnen Module eng verdrahtet sind. Vgl. [Kno12, S. 48ff]

2.3.1 Schichtenmodell

Ein beliebtes Vorgehen von Softwareentwicklern ist das Aufteilen einer Anwendung in verschiedene Schichten. Durch das 3-Schichten-Modell in Abb. 2.3 wird der Aufbau der Anwendung dargestellt.

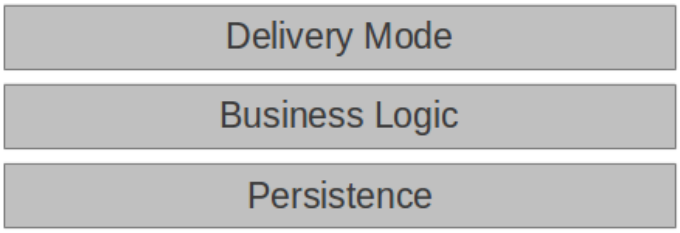

Abb. 2.3: 3-Schichten-Model (eig. Darstellung)

Über den Delivery Mode werden die Funktionen der Business Logic den Benutzern bereitgestellt. Die Bereitstellung der Funktionen kann z. B. über ein webbasiertes oder desktopbasiertes User Interface (UI) und/oder einer Webservice Schnittstelle erfolgen. Hinsichtlich der Beispielanwendung ist der Delivery Mode eine webbasierte UI. In der Business Logic Schicht werden alle fachlichen Anforderungen aus den Use Cases als Funktionen der Anwendung implementiert. Die Daten, die in der Business-

Logic-Schicht verarbeitet werden, werden über die Persitence-Schicht bereitgestellt. In dieser Schicht wird das Speichern und Laden der Business Objects realisiert. Vgl. [Kno12, S. 162ff]

In den drei Schichten der Beispielanwendung kommen verschiedene Patterns zum Einsatz, um so eine Architektur zu realisieren in der die einzelnen Module entkoppelt sind.

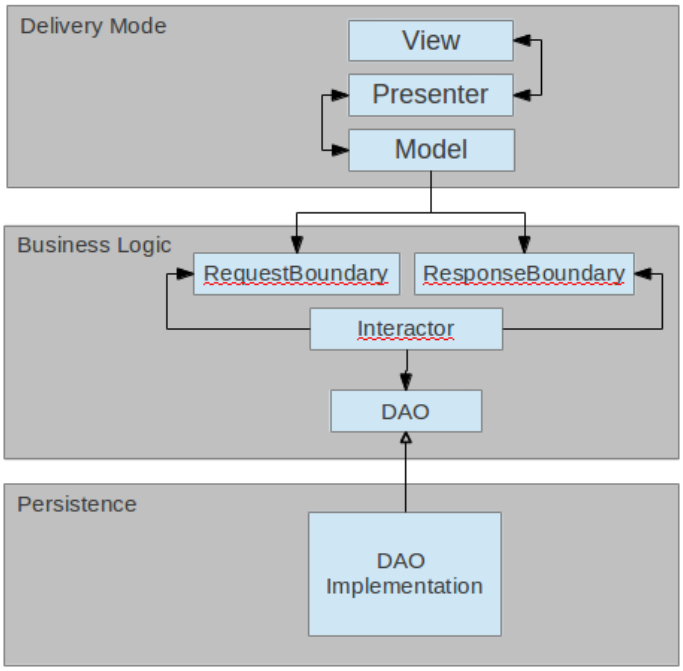

Abb. 2.4: Erweitertes 3-Schichten-Model (eig. Darstellung)

In der Delivery-Mode-Schicht kommt das Model-View-Presenter-Pattern (MVP) zum Einsatz. Die View repräsentiert die Oberflächenelemente (z. B. Textfelder). Der Presenter enthält die Daten welche dem Benutzer über die View angezeigt werden. Werden die Daten durch den Benutzer geändert werden diese Änderungen im Presenter hinterlegt. Das Model hat die Aufgabe, die Eingaben des Benutzers anschließend an die entsprechende Funktion der Business Logic zur Verarbeitung zu übergeben. In der Business-Logic-Schicht wird das Interactor-Konzept von Robert C. Martin verwendet. Mit diesem Konzept kann die Business Logic-Schicht von der Persistence-Schicht und vom Delivery Mode entkoppelt werden, wodurch die Tests der Business Logic z. B. mit Testdaten verknüpft werden können. Ein weiterer Vorteil ist, dass kein Webserver oder Datenbankserver gestartet werden muss, um die Tests

ausführen zu können. Mit dem Data Access Object Pattern wird die Business-Logic-Schicht von der Persistence-Schicht entkoppelt. Jeder Interactor definiert über ein DAO Interface die Funktionen mit denen die benötigten Daten aus der Persistence-Schicht geladen werden können. Die Persistence-Schicht implementiert diese DAOs. Vgl, [Kno12, S. 162ff]

2.3.2 Interactor-Konzept

Das Interactor Konzept ist für die Architektur der Anwendung ein zentraler Bestandteil. Deshalb wird das Konzept an dieser Stelle nochmal genauer erläutert. In dem Buch [Mar12] beschreibt der Autor, dass mit dem Command-Pattern, die vom Kunden geforderten Funktionen, realisiert werden können. Jede Funktion wird in einer Command-Klasse mit einer Methode *execute* implementiert. Die Funktionen die implementiert werden sollen, können durch das Verdichten der User Stories zu Use Cases ermittelt werden. Für die Beispielanwendung entstehen so zwei Use Cases.

1. Use Case: Speiseplan Erstellung
2. Use Case: Anzeige aktueller Speiseplan

Der erste Use Case besteht aus den Anforderungen der User Stories 1 bis 5 und den dazugehörigen Acceptance Tests 1 bis 5. Der zweite Use Case besteht aus der User Stroy 6 und dem Acceptance Test 6. Vgl.[Mar12, S. 151ff]

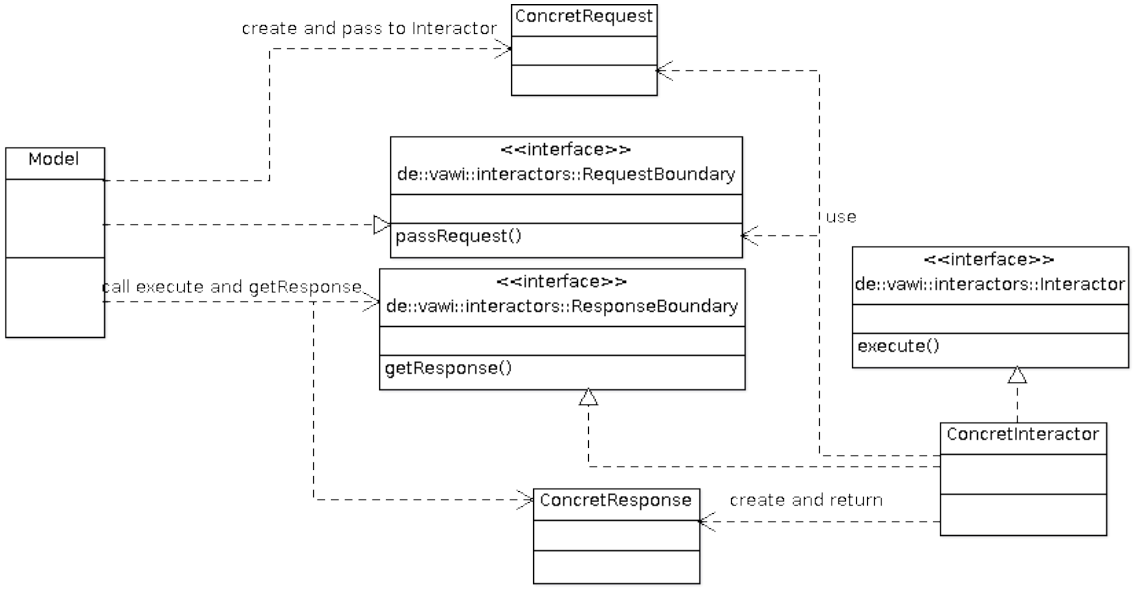

Abb. 2.5: Implementierung des Interactor Konzept (eig. Darstellung)

Die Abb. 2.5 zeigt die Umsetzung des Interactor-Konzepts. Für jeden Use Case wird eine Interactor-Klasse entwickelt. Der Interactor wird von einer Klasse (Model) auf Seite des Delivery Mode instantiiert. Die Parameter, welche der Interactor für die Ausführung benötigt, werden in einem Request-Objekt gebündelt an den Interactor übergeben. Anschließend wird die Methode *execute* des Interactors aufgerufen. Das Ergebnis wird in ein Response-Objekt geschrieben. Dieses kann von dem Aufrufer für die Anzeige auf der UI genutzt werden. Ggf. muss der Aufrufer die Daten aus dem Response-Objekt für die UI aufbereiten.

2.3.3 Technologien

Die Beispielanwendung wird als Webanwendung entwickelt. Hierzu werden auf Server- und Client-Seite verschiedene Technologien eingesetzt.

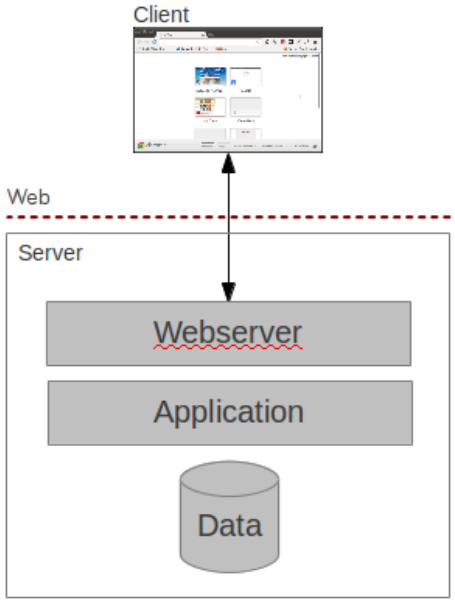

Abb. 2.6: Architektur Webanwendung (eig. Darstellung)

Server-seitig wird die Anwendung mit Java entwickelt. Java ist u. a. für die Programmierung von unternehmensinternen Anwendungen sehr verbreitet. Darüber hinaus gibt es es ein weitreichendes Angebot an Frameworks für die Realisierung von Webanwendungen. Vgl. [ea10, S. 244ff]. Die Oberfläche wird mit HTML5 und CSS3 sowie JavaScript realisiert. Die Wahl der Frameworks und APIs die server- und client-seitig eingesetzt werden, hängt von der Unterstützung hinsichtlich der Testautomatisierung ab. Deshalb wird die Wahl der Frameworks für die Realisierung der Webanwendung erst im Kapitel 2 durchgeführt.

2.4 Testarten

Welche Testarten gibt es, die für die Automatisierung in Frage kommen? Die Fachliteratur liefert keine eindeutige Antwort, jedoch ließen sich in verschiedenen Quellen Testarten finden, die für die Automatisierung geeignet erscheinen.

Mike Cohan beschreibt in seinem Buch „Succeeding with Agile“ die verschiedenen Testarten einer Teststrategie. Die Abb. 2.7 zeigt die Testarten. Diese bauen aufeinander auf. Jede Ebene testet einen anderen Aspekt der Anwendung. Zusammen genommen ergeben alle Testarten die Teststrategie mit der Softwareentwicklern den

Anforderungen des Qualitätsmanagement und dem Kunden gerechte werden wollen. Vgl. [Mar11b, S. 114]

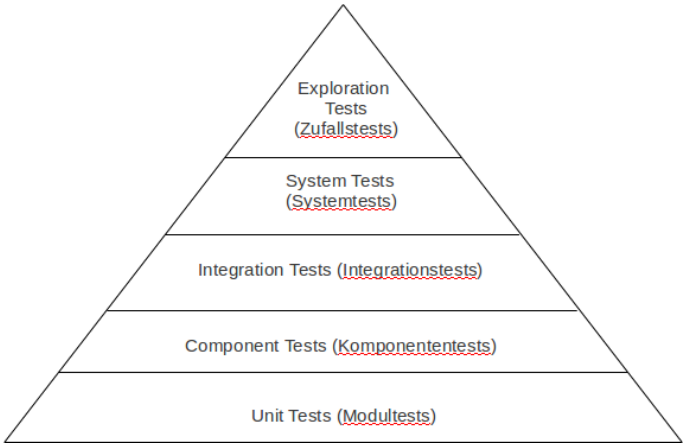

Abb. 2.7: Testpyramide (angelehnt an: [Mar11b, S. 114])

Im Buch „The Art of Software Testing“ werden Unit Tests als Basis verstanden. Sie seien jedoch nicht ausreichend, da diese lediglich einzelne Funktionseinheiten testen aber nicht deren Zusammenspiel. Die Zusammenarbeit von Funktionseinheiten soll in Tests höherer Ordnung getestet werden. Vgl. [Mye04, S. 90ff]

Beide Bücher ordnen Testarten verschiedenen Ebenen zu. Die Testarten stimmen ebenfalls überein. Somit werden die Testarten aus der Pyramide von Mike Cohan hinsichtlich ihrer Automatisierbarkeit untersucht.

2.4.1 Unit Tests (Modultests)

Unit Tests bilden in der Testpyramide das Fundament. Beim Programmieren unterlaufen dem Softwareentwickler Fehler. Diese können frühzeitig durch Unit Tests entdeckt werden. Darüber hinaus gilt die Implementierung von Unit Tests als einfacher Weg, um automatisierte Tests in ein Projekt zu integrieren. Vgl. [Rot07, S. 256ff]

Definition

In dem Buch „Software Qualität“ beschreibt Prof. Dr.-Ing. Peter Liggesmeyer Unit Tests wie folgt:

„In der klassischen Entwicklung wird in der Regel eine abgeschlossene Funktionseinheit als kleinste unabhängig prüfbare Einheit betrachtet. Für die Entwicklung von Software bedeutet das in Abhängigkeit der Mechanismen, die die verwendete Programmiersprache zur Verfügung stellt, dass bei einem geeigneten Entwurf z. B. Prozeduren, Funktionen oder Unterprogramme getrennt voneinander getestet werden können.“ [Software-Qualität, S. 411]

Das Buch „Working Effectivley with Legacy Code“ werden Unit Tests wie folgt definiert:

„Common to most conceptions of unit tests is the idea that say are tests in isolation of individual components of software. What are components? The definition varies, but in unit testing, we are usually concerned with the most atomic behavioral units of a system. In procedural code, the units are often functions. In object-oriented code, the units are classes“ [Fea11, S. 12]

Somit kann festgehalten werden, dass im Kontext einer objektorientierten Programmiersprache mit Hilfe von Unit Tests die Funktionalität einer Klasse getestet werden kann.

In dem Buch [Mar09a, S. 132ff] beschreibt Robert C. Martin, die F.I.R.S.T Eigenschaften, die ein Unit Test erfüllen soll.

Schnell Unit Tests sollen schnell sein, weil die Entwickler diese häufig ausführen sollen, um sicherzustellen, dass Änderungen keine anderen Funktionen beeinträchtigt haben. Wenn diese jedoch langsam sind, besteht die Gefahr, dass die Tests nicht nach jeder Änderung ausgeführt werden. Wodurch Fehler nicht frühzeitig entdeckt werden und der Code ggf. nicht angemessen überarbeitet wird.

Unabhängig Die Unit Tests sollen unabhängig von einander ausgeführt werden können. Es darf somit keine Abhängigkeiten zwischen den einzelnen Tests geben. Tests müssen auch hinsichtlich der Reihenfolge in der sie ausgeführt werden unabhängig sein. Grund ist, dass dadurch die Fehleranalyse erschwert wird, da die Ursache für das Fehlschlagen eines Tests ggf. in einem ganz anderem Unit Test liegt.

Wiederholbar Die Unit Tests sollen in jeder Umgebung wiederholbar sein. Damit das Fehlschlagen eines Tests nicht damit entschuldigt werden kann, dass sich die Umgebung geändert hat.

Selbst-Auswertend Die Unit Tests sollen durch ein Boolean auswertbar sein. Mittels *true* oder *false* soll entschieden werden können ob ein Test fehlgeschlagen ist oder nicht. Dadurch wird sichergestellt, dass der Softwareentwickler nicht manuell die Ausgaben der Anwendung auswerten muss, in dem er z. B. Konsolenausgaben miteinander vergleicht. Manuelle Prüfungen beanspruchen gegenüber automatisierten Prüfungen mehr Zeit.

Rechtzeitig Die Unit Tests sollen vor dem eigentlichen Produktiv-Code geschrieben werden. Das nachträgliche Entwickeln des Unit Tests kostet in der Regel mehr Zeit als den Test vorab zu entwickeln. Grund hierfür ist, dass z. B. Abhängigkeiten für die Tests gebrochen werden müssen und so eine Überarbeitung des Produktiv-Codes notwendig ist.

Frameworks

Für die Realisierung von Unit Tests existieren in vielen Programmiersprachen wie Java, JavaScript, C#, C++, PHP usw. Frameworks. In Java sind vor allem JUnit[3] und TestNG[4] bekannt. Beide Frameworks sind Open-Source Projekte und in gängigen Entwicklungsumgebungen wie Netbeans, Eclipse und Intelij Idea integriert. Sie unterscheiden sich jedoch im Funktionsumfang. TestNG bietet neben den Funktionen für Unit Tests, Funktionen für das Anbinden von externen Datenquellen. Diese Funktionen sind für Tests oberhalb der Unit-Test-Ebene interessant. JUnit hingegen bietet u. a. solche Möglichkeiten nicht. Da Unit Tests allerdings isoliert von anderen Systemen (z. B. Datenbanken oder Dateisystemen) entwickelt werden sollen, wird für die Beispielanwendung JUnit eingesetzt.

Wie bereits erwähnt ist JUnit in den gängigen Entwicklungsumgebungen inte-

[3] http://junit.org/
[4] http://testng.org/doc/index.html

griert. Die Beispeilanwendung enthält einen Source-Packages-Ordner[5] (*/src*) sowie einen Test-Packages Ordner[6](*/test*). Im Test-Package-Ordner werden die Testklassen zu den Klassen im Source-Package-Ordner angelegt. Die Testklassen werden wie die zu testende Klassen benahmt und um den Suffix „Test“ erweitert. Z. B. wird für die Klasse *CalendarWeek*[7] die Test-Klasse *CalendarWeekTest*[8] angelegt. Die Test-Klasse sollte darüber hinaus im gleichen Package liegen wie die zu testende Klasse.

Im Folgenden wird ein einfacher Test mit JUnit gezeigt:

```
1  package de.vawi.factoryCanteen;
2
3  import org.junit.Test;
4  import static org.junit.Assert.*;
5
6  public class HelloJunitTest {
7
8      @Test
9      public void testHelloJunitWorld() {
10         String hello = "Hello JUnit!";
11         assertEquals("Hello JUnit!", hello);
12
13     }
14 }
```

Code-Ausschnitt 2.1: Einfache Testklasse mit JUnit

Eine JUnit-Test-Klasse benötigt den Import *org.junit.Test*. Über diesen Import werden die verschiedene Annotationen bereitgestellt, mit denen z. B. eine Methode als Test deklariert werden kann. Der statische Import der Assertion ermöglicht den direkten Aufruf der statischen Methoden innerhalb der Assert-Klasse. Diese Methoden werden für die Prüfung der Ergebnisse verwendet. Mit der Annotation *@Test* wird eine Methode als Test-Methode deklariert. Beim Ausführen der Test-

[5] https://github.com/iface06/factoryCanteen/tree/master/src
[6] https://github.com/iface06/factoryCanteen/tree/master/test
[7] http://git.io/NdJ5Zg
[8] http://git.io/jjyv2g

Klasse werden alle Methoden mit einer Annotation *@Test* ausgeführt. In der Methode *testHelloJunitWorld* wird die Methode *assertEquals* aufgerufen. Diese prüft, ob das erwartete Ergebnis *Hello JUnit!* in der Variablen *hello* steht. Damit wird sichergestellt, dass die Methode der Eigenschaft Selbstauswertend entspricht. Neben *assertEquals* bietet JUnit weitere Assert-Methoden[9], mit denen überprüft werden kann, ob eine Funktion das erwartete Ergebnis liefert.

Integration in agilen Sofwareentwicklungsprozess

Mit Hilfe von JUnit sind Entwickler technisch in der Lage Unit Tests effizient zu realisieren. Das Entwickeln von Unit Tests sollte den Softwareentwickler nicht das Gefühl geben, zusätzlichen und unnötigen Code zu produzieren. Dazu ist es wichtig, dass die Ziele, welche mit den Unit Tests verfolgt werden, für alle nachvollziehbar sind und das alle Projektbeteiligte Unit Tests als Weg sehen, diese Ziele zu erreichen. Ein weiterer wichtiger Punkt ist, dass alle Softwareentwickler ein gemeinsames Verständnis haben, was Unit Tests sind und wie diese mit Hilfe eines Frameworks implementiert werden können. Dazu sind Schulungen notwendig, sowie das Erarbeiten eines gemeinsamen Vorgehens. Vgl. [Rot07, 259]

Ein Vorgehen um die Implementierung von Unit Tests in die tägliche Arbeit zu integrieren ist Test-driven Development (TDD). TDD wurde von Kent Beck entwickelt. Bei diesem Vorgehen werden die Tests vor dem Produktiv-Code entwickelt. Als Erstes überlegt sich der Softwareentwickler im Vorfeld, welche Anforderung realisiert werden soll. Die Anforderung muss dann in kleine Teilaufgaben zerlegt werden. Durch die Zerlegung der Anforderung in kleine Aufgaben, kann eine hohe Testabdeckung erreicht werden. Anschließend realisiert der Entwickler jede Teilaufgabe in drei Phasen. Diese werden in Abbildung Abb. 2.8 dargestellt. Vgl. [Bec02, Einleitung]

[9] https://github.com/junit-team/junit/wiki/Assertions

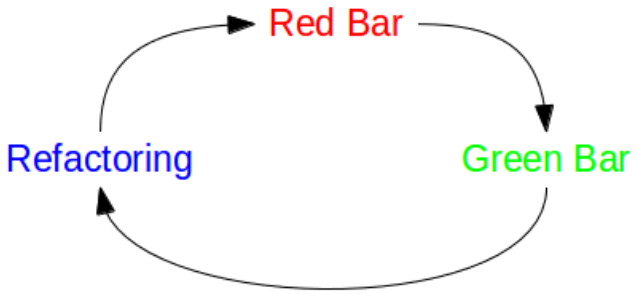

Abb. 2.8: TDD Zyklus (angelehnt an: [Fre12, S. 6])

Red Bar In dieser Phase wird der Unit Test geschrieben. Anschließend wird der Test ausgeführt. Erwartungsgemäß schlägt der Test fehl, da noch kein Code geschrieben wurde, der den Test erfüllt. Wobei Fehlschlagen auch bedeutet, das der Test nicht kompiliert. Vgl. [Bec02, S. 11]

Green Bar In der zweiten Phase wird der Test zum kompilieren gebracht und eine möglichst einfache Lösung geschrieben, die den Test erfüllt. Vgl. [Bec02, S. 11]

Refactoring In der letzten Phase wird sowohl der Produktiv-Code als auch der Test-Code überarbeitet. Dazu werden Code-Duplicationen und zu lange Methoden überarbeitet. Die Basisregeln für *sauberen Code* liefert u. a. das Buch „Clean Code“ von Robert C. Martin.[10] Vgl. [Bec02, S. 11]

Anwendungsbeispiel

Nachdem die Phasen und Regel von TDD vorgestellt wurden, wird im Folgenden gezeigt, wie mit Hilfe von TDD eine Funktion aus einer Anforderung realisiert werden kann. Dazu wird eine kleine Aufgabe aus dem Use Case *Speiseplan erstellen* extrahiert und test-driven realisiert.

Die Speisen, die über den Speiseplan angeboten werden, sollen beim beim Programmstart aus der Datei *hitliste.csv*[11] importiert werden. Ein Eintrag in dieser Datei sieht wie folgt aus:

[10] [Mar09a]
[11] http://git.io/hizC2w

```
1 1,"Bohneneintopf Mexiko","v"
```

Code-Ausschnitt 2.2: Zeile aus der Datei hitliste.csv

Der erste Wert ist die Beliebtheit, der zweite der Namen und letzte Wert gibt den Typ der Speise an. Dabei steht *v* für Vegetarisch, *m* für Fleisch und *f* für Fisch. Um beim Importvorgang jedem Dish Objekt den richtigen Typ zuordnen zu können, soll das Enum DishCategory[12] anhand der drei Kurzzeichen *v*, *m* und *f* die dazu korrespondierende Enumeration zurückgeben. Die Implementierung dieser Funktion wird in folgende Schritte zerlegt:

1. Wenn die Abkürzung *v* übergeben wird, gib *Vegetarisch* zurück
2. Wenn die Abkürzung *m* übergeben wird, gib *Fleisch* zurück
3. Wenn die Abkürzung *f* übergeben wird, gib *Fisch* zurück
4. Wenn eine Falsche Abkürzung übergeben wird, soll eine RuntimeException geworfen werden

Erster Test Vorab wird eine Klasse *DishCategoryTest*[13] mit dem gleichen Package wie *DishCategory*[14] in dem Test-Ordner angelegt. Mit dem ersten Test wird vorab nur die Instanziierung und der Aufruf der Methode, die das erwartete Ergebnis zurück liefern soll, entwickelt. Dazu schreibt der Softwareentwickler wie er die Methode aufrufen möchte und welche Parameter übergeben werden.

```
1     @Test
2     public void test(){
3         DishCategory category = DishCategory.byAbbreviation("v");
4     }
```

Code-Ausschnitt 2.3: Erster Unit Test für die Instanziierung

Das Enum *DishCategory* soll eine statische Methode anbieten, die das Kurzzeichen aus der Zeile der CSV-Datei entgegen nimmt. Als Rückgabewert erwartet der Entwickler dann die dazu passende Enumeration.

[12] http://git.io/RqTRTg
[13] http://git.io/Fz1OhQ
[14] http://git.io/RqTRTg

```
1 public enum DishCategory {
2         MEAT, FISH, VEGETARIAN;
3
4         public static DishCategory byAbbrevation(String abbr){
5                 return null;
6         }
7 }
```

Code-Ausschnitt 2.4: Produktiv-Code zum ersten Test

Beim Ausführen des Tests schlägt dieser mit einem Kompilierfehler fehl. Somit befinden wir uns am Ende der ersten Phase *Red Bar* und gehen in die *Green Bar* Phase über. Der Entwickler darf gemäß der ersten Regel jetzt Produktiv-Code schreiben. Dazu erstellt er das Enum *DishCategory* und deklariert die Methode *byAbbreviation(String category)* sowie die drei Enumerationen *VEGETARIAN*, *MEAT* und *FISH*. Als Return-Wert kann erstmal *null* zurückgegeben werden. Anschließend führt er den Test erneut aus. Der Test kompiliert und schlägt nicht mehr fehl. Somit wechselt der Entwickler in die dritte Phase *Refactoring*. In dieser Phase soll der Test sowie der Produktiv-Code überarbeitet werden. Da es weder im Produktiv-Code noch im Test aktuell etwas zu überarbeiten gibt, kann mit der ersten Aufgabe aus der Liste begonnen werden.

Wenn die Abkürzung ein v ist gib Vegetarisch zurück Der erste Test kann dazu einfach kopiert und umbenannt werden. Anschließend wird der Test um den Aufruf einer Assertion-Methode erweitert. Als Parameter wird die erwartete Enumeration *VEGETARIAN* übergeben und die zurückgegeben Kategorie.

```
1 @Test
2 public void testFindFishByAbbreviation(){
3     DishCategory category = DishCategory.byAbbreviation("v");
4     assertEquals(DishCategory.VEGETARIAN, category);
5 }
```

Code-Ausschnitt 2.5: Test für die Rückgabe VEGETARIAN

Anschließend wird der Test ausgeführt. Dieser Schlägt fehl, da bislang die Methode *byAbbreviation* nur *null* zurück gibt. Nachdem der Test also fehlgeschlagen ist, darf wieder Produktiv-Code geschrieben werden. Allerdings nicht mehr als notwendig, um den Test erfolgreich ausführen zu können (3. Regel). Somit muss die Methode *byAbbreviation* anstatt *null* nun die Enumeration *VEGETARIAN* zurückgeben.

```
1 public enum DishCategory {
2         MEAT, FISH, VEGETARIAN;
3
4         public static DishCategory byAbbrevation(String abbr){
5                 return DishCategory.VEGETARIAN;
6         }
7 }
```

Code-Ausschnitt 2.6: Enum DishCategory

Der Test läuft mit dieser Änderung erfolgreich durch. Nachdem nur eine Zeile im Enum hinzugefügt wurde, gibt es nichts zu überarbeiten. In der Testklasse jedoch kann der erste Test für das Instantiieren gelöscht werden, da dieser in der neuen Testmethode mit abgedeckt ist. Anschließend kann mit der zweiten Aufgabe fortgefahren werden.

Wenn die Abkürzung ein m ist gib Fleisch zurück Dazu kann der vorherige Test wieder kopiert werden, da sich nur der Parameter und der erwartete Wert in der *assertEquals* Methode ändert.

```
1 @Test
2 public void testFindMeatByAbbreviation(){
3     DishCategory category = DishCategory.byAbbreviation("m");
4     assertEquals(DishCategory.MEAT, category);
5 }
```

Code-Ausschnitt 2.7: Test für die Rückgabe MEAT

Anschließend kann der Test gestartet werden. Erwartungsgemäß schlägt der Test fehl und der Entwickler kann wieder Produktiv-Code schreiben. Dieser könnte wie folgt aussehen:

```
1  public enum DishCategory {
2          MEAT, FISH, VEGETARIAN;
3
4          public static DishCategory byAbbrevation(String abbr){
5                  if(abbr.equals("v")){
6                          return DishCategory.VEGETARIAN;
7                  } else if(abbr.equals("m")) {
8                          return DishCategory.MEAT;
9                  } else {
10                         return null;
11                 }
12         }
13 }
```

Code-Ausschnitt 2.8: Klasse DishCategory für den MEAT Test

Der Test läuft nach der Erweiterung der Methode erfolgreich durch und es kann mit der Überarbeitung begonnen werden. Die Methode ist auf den ersten Blick übersichtlich und einfach. Allerdings besteht die Gefahr, dass wenn weitere Kategorien hinzukommen, z. B. *DESSERT*, die Methode durch das wachsende *If-Else-Kontrollkonstrukt* unübersichtlich wird und so die Erweiterbarkeit und Wartbarkeit erschwert wird. Eine mögliche Lösung ist die Folgende:

```
1  public enum DishCategory {
2          MEAT("m"), FISH("f"), VEGETARIAN("v");
3
4          private String abbreviation;
5
6          DishCategory(String abbreviation){
7                  this.abbreviation = abbreviation;
8          }
9
10         public static DishCategory byAbbrevation(String abbr){
11                 for(DishCategory kategorie : values()) {
12                 if (kategorie.abkuerzung.equals(abbr)) {
13                         return kategorie;
14                     }
15             }
16                 return null;
17         }
18 }
```

Code-Ausschnitt 2.9: Klasse DishCategory nach der Überarbeitung im MEAT Test

Im Fall, dass neue Kategorien hinzukommen, muss das Enum um eine Enumeration erweitert werden. Die Logik der Methode *byAbbreviation* muss nicht angepasst werden. Nach der Überarbeitung werden die Tests erneut durchgeführt, um so sicherzustellen, dass sich durch seine Änderungen keine Fehler entstanden sind. Dies ist ein wichtiger Vorteil des TDD-Ansatzes. Der Entwickler kann jederzeit den Code überarbeiten, ohne Angst haben zu müssen, dass er dabei bereits implementierte Funktionalität zerstört und dies nicht bemerkt.

Nach der Überarbeitung und dem erfolgreichen Ausführen der Tests kann nun mit der letzten Aufgabe fortgefahren werden.

Wenn die Abkürzung ein f ist gib Fisch zurück Dazu wird wieder eine Testmethode implementiert die den Fall analog zu den vorherigen Tests abdeckt.

```
1     @Test
2     public void testFindFishByAbbreviation(){
3         DishCategory category = DishCategory.byAbbreviation("f")
              ;
4         assertEquals(DishCategory.FISH, category);
5     }
```

Code-Ausschnitt 2.10: Test für die Rückgabe MEAT

Der Test schlägt jedoch nicht fehl, da bereits durch den vorherigen Test, die *byAbbreviation* Methode so überarbeitet wurde, dass das Enum einfach um neue Kategorien erweitert werden kann. Somit ist die dritte Aufgabe erledigt.

Im Fall, dass eine unbekannte Abkürzung übergeben wird, kann das Programm nicht weiterarbeiten, da eine Speise mit unbekannter Kategorie nicht beim Erstellen des Speiseplans berücksichtigt werden kann. Da das Enum jedoch nicht entscheiden kann, wie mit dem Import weiter verfahren werden kann, soll eine Exception geworfen werden. Gemäß [Mar09b, S. 106] soll eine *Unchecked Exception* geworfen werden, die dann vom Import gefangen werden kann. Der Vorteil einer *Unchecked Exception* ist, dass es zu keiner Abhängigkeit zwischen den Klassen kommt und der Entwickler frei entscheiden kann, ob die Exception behandelt werden kann oder ob der Fehler so schwerwiegend ist, dass die Anwendung beendet werden muss. Vgl. [Mar09a, S. 103ff]

```
1 @Test(expected = DishCategory.UnknownDishCategoryException.class
      )
2 public void testUnknownDishCategory(){
3     DishCategory category = DishCategory.byAbbrevation("x");
4 }
```

Code-Ausschnitt 2.11: Test für für unbekannte Kategorie

Für das Testen von Exceptions bietet JUnit die Möglichkeit der Test-Annotation mitzuteilen welche Exception erwartet wird. Wird diese Exception nicht geworfen, so schlägt der Test fehlt. Beim Ausführen des Tests schlägt dieser fehl, weil noch keine Exception geworfen wird. Anschließend wird die Methode *byAbbrevation* im

Enum *DishCategory* erweitert.

```
1  public enum DishCategory {
2          MEAT("m"), FISH("f"), VEGETARIAN("v");
3
4          private String abbreviation;
5
6          DishCategory(String abbreviation){
7                  this.abbreviation = abbreviation;
8          }
9
10         public static DishCategory byAbbrevation(String abbr)
11                 throws UnknownDishCategory{
12
13                 for (DishCategory kategorie : values()) {
14                 if (kategorie.abkuerzung.equals(abbr)) {
15                 return kategorie;
16                     }
17             }
18                 throw new UnknownDishCategoryException();
19         }
20
21         public static class UnknownDishCategoryException
22                 extends RuntimeException{}
23 }
```

Code-Ausschnitt 2.12: Klasse DishCategory mit UnknownDishCategoryException

Nachdem das Werfen der Exception implementiert ist wird der Test erneut ausgeführt. Die Abkürzung x ist unbekannt, weshalb die *UnknownDishCategoryException* geworfen wird. Der Test fängt die geworfene Exception und prüft, ob es sich um die erwartete Exception handelt. Da dies der Fall ist der Test erfolgreich.

Somit sind alle Aufgaben für die Implementierung der *DishCategory* erledigt. Das Beispiel wurde bewusst sehr einfach gewählt, da nicht die Lösung, sondern der Lösungsweg im Vordergrund steht.

2.4.2 Component Tests (Komponententests)

Die nächste Ebene der Testpyramide ist die der Component Tests. Auf dieser Ebene werden die Akzeptanztests automatisiert, die mit den Stakeholdern zusammen mit den User Stories erfasst wurden.

Definition

Für die Definition werden wieder die Beschreibungen aus verschiedenen Fachbüchern herangezogen. Robert C. Martin definiert Akzeptanztests als Tests mit denen die Softwareentwickler feststellen können, ob eine Anforderung aus Kundensicht vollständig realisiert wurde. Vgl. [Mar11b, S. 116]

Glenford J. Myers sieht dies ähnlich:

> „With acceptance tests, the customer validates an expected result from the application. A deviation from the expected result is considered a bug and is reported to the development team.“ [Mye04, S. 127]

Aus diesen beiden Beschreibungen kann festgehalten werden, dass Akzeptanztests zusammen mit den Stakeholdern erstellt werden. Anhand der schriftlich Tests können die Stakeholder testen, ob eine User Story korrekt realisiert wurde. Gleichzeitig können die Softwareentwickler während der Realisierung prüfen, ob sie die Anforderungen korrekt umgesetzt haben.

Darüber hinaus können die Akzeptanztests als Regressionstests verwendet werden. Mit Regressiontests soll sichergestellt werden, dass durch das Hinzufügen neuer Funktionen, bereits implementierte Funktionen nicht beeinträchtigt wurden. Dies ist vor allem für Projekte mit einem agilen Softwareentwicklungsprozess interessant, da Regressionstests am Ende jeder Iteration durchgeführt werden. Der Aufwand für Regressiontests steigt somit von Iteration zu Iteration. Durch die Automatisierung der Akzeptanztests kann dem steigenden Aufwand begegnet werden. Vgl. [Fre12, S. 7ff]

Integration im agilen Sofwareentwicklungsprozess

Akzeptanztests müssen aus der Sicht der Stakeholder geschrieben werden. Dies sollte bei der detaillierte Anforderungserhebung pro Iteration durchgeführt werden. Die Automatisierung der Akzeptanztests erfolgt anschließend durch einen Softwareentwickler. Vgl. [Mar11b, S. 105]

In dem Buch „Growing Object-Oriented Software, Guided by Tests“ von Steve Freeman und Nat Pryce beschreiben die Autoren, dass jeder TDD Zyklus (Reb Bar, Green Bar, Refactoring) mit dem Fehlschlagen eines Akzeptanztests startet. Abbildung Abb. 2.9 zeigt den erweiterten TDD Zyklus.

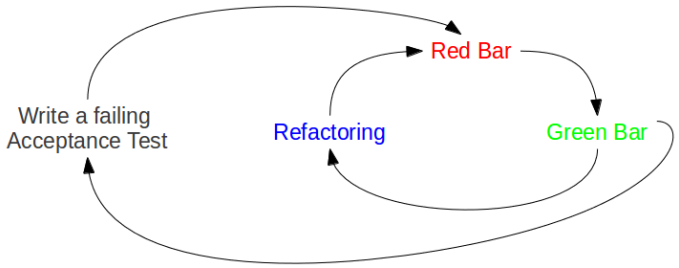

Abb. 2.9: Akzeptanztest Zyklus (angelehnt an: [Fre12, S. 6])

Nachdem Automatisieren eines Akzeptanztests, schlägt dieser beim ersten Ausführen fehl. Das liegt daran, dass die Funktionen noch nicht implementiert wurden, welche den Akzeptanztest erfüllen. Die einzelnen Funktionen werden anschließend test-driven realisiert. Der TDD-Zyklus kann mehrmals durchlaufen werden, um die benötigten Funktionen zu entwickeln. Anschließend wird der automatisiert Akzeptanztest wieder ausgeführt. Sobald der Akzeptanztest erfolgreich war, ist die Anforderung realisiert. Ist er nicht erfolgreich wird wieder mit Hilfe der Unit Tests und TDD weiterentwickelt.

Frameworks

Für die Automatisierung von Akzeptanztest gibt es unterschiedliche Vorgehensweisen. Laut Freeman sollen Akzeptanztest als End-To-End-Tests realisiert werden. Ein End-To-End-Test nutzt für den Aufruf der zu testenden Funktion die gleiche Schnittstelle wie der Benutzer. Im Fall der Beispielanwendung z. B. die webbasierte

UI. Für die Prüfung wiederum verwendet der End-To-End-Test die Antwort der Anwendung. Im Falle der Beispielanwendung ist dies die in der UI gezeigte Ergebnis (z. B. der Speiseplan der aktuellen Woche). Neben der eigentlichen Komponente wird somit auch die Zusammenarbeit aller Schichten der Architektur getestet. Wenn alle Akzeptanztest erfolgreich ausgeführt wurden ist die Anwendung voll Funktionsfähig und kann ausgeliefert werden. Vgl. [Fre12, S. 8ff]

Robert C. Martin beurteilt Akzeptanztest, welche die UI einer Anwendung nutzen kritisch. Grund dafür ist, dass die UI sehr häufig geändert wird, wodurch die Akzeptanztests ständig angepasst werden müssten. Daher empfiehlt er, dass die zu testende Funktionalität nicht über den Delivery Mode getestet werden soll. Vgl. [Mar11b, S. 200ff]

Bei der Analyse der User Stories und deren Akzeptanztest aus Abschnitt 2.2 für die Beispielanwendung, stellte sich heraus, dass zusätzliche Informationen an die Benutzeroberfläche übertragen werden müssten, um die Akzeptanztests als End-To-End-Tests realisieren zu können. Es müsste beispielsweise der Typ einer Speise übertragen werden, um testen zu können, ob es jeden Tag ein Fleisch und ein vegetarisches Gericht gibt. Das gleiche gilt für die Freitags-Fisch-Regel. Deshalb werden die Akzeptanztests im Anwendungsbeispiel nicht als End-To-End-Test realisiert.

Als Nächstes wird nach einem geeignetem Framework für Akzeptanztests gesucht.

Cucumbar[15] Cucumbar ist ein Framework für die Realisierung von Acceptance Tests. Die Akzeptanztests werden in einer Textdatei im *Given/When/Then Style* geschrieben. Die einzelnen Schritte des Tests werden mit der Skriptsprache Ruby programmiert und kann über die Systemkonsole ausgeführt werden. Die Ergebnisse werden auf der Konsole ausgegeben.

FITnesse[16] Mit Hilfe dieses Frameworks werden mit einer Syntax, die der eines MediaWiki ähnelt, Tests geschrieben und anschließend über eine Java-Klasse automatisiert. Die Tests können über eine Wiki-ähnliche Webseite gestartet werden. Die Ergebnisse des Tests werden dann für den Stakeholder auf einer Webseite angezeigt.

[15] http://cukes.info/
[16] http://fitnesse.org

jBehave[17] Das dritte Akzeptanztest-Framework jBehave ist dem Cucumber Framework sehr ähnlich. Der Acceptance Test wird ebenfalls in der Textdatei im *Given/When/Then Style* geschrieben. Zu jeder Textdatei wird in einer Java-Klasse die einzelnen Schritte des Tests programmiert. Die Java-Klasse kann dann mittels eines JUnit Tests ausgeführt werden. Die Ergebnisse werden in verschiedene Formaten ausgegeben. Dazu gehören HTML, Text-Dateien und XML. Somit können die Ergebnisse z. B. über eine Website bereitgestellt werden.

Alle drei Frameworks werden als Open Source Lösung angeboten und haben eine umfassende Dokumentation wodurch eine schnelle Einarbeitung ermöglicht ist. Einen leichten Vorteil hinsichtlich der Einarbeitung hat jBehave, weil der Softwareentwickler in der gewohnten Java-Umgebung bleibt und keine weitere Syntax oder Skritsprache lernen muss. Darüber hinaus können die Akzeptanztests, die mit jBehave entwickelt wurden, über eine JUnit Test gestartet und ausgewertet werden. Dadurch wird die Integration in die Beispielanwendung erleichtert. Deshalb wird für die Automatisierung der Akzeptanztests für die Beispielanwendung das Framework jBehave eingesetzt.

Anwendungsbeispiel

Das Akzeptanztest Framework jBehave ist nicht in Netbeans integriert, weshalb die Bibliothek des Frameworks[18] manuell eingebunden werden müssen. Danach kann eine Story-Datei und eine dazugehörige Klasse angelegt werden. Die beiden Dateien sollten sich im gleichen Package befinden und den gleichen Namen haben. In der Beispielanwendung wurde die Story *CreateMenuForPeriode.story*[19] und die *CreateMenuForPeriode.java*[20] angelegt. Die Story-Datei enthält folgenden Test:

[17] http://jbehave.org
[18] http://jbehave.org/reference/stable/dependencies.html
[19] http://git.io/WSDdWg
[20] http://git.io/JfUDBw

```
Narrative:
This acceptance test should demonstrate that the creation of a
menu for the given periode comply with all given rules.


Scenario: Take care that menu contains required dishes
Given a menu contains 3 meals per day for the next 5 days
When the periode is 3 weeks
Then expect 15 days with 45 dishes
Then each day with 3 dishes
Then each day contains a vegetarian dish, a dish with meat and a
    third alternativ
Then each dish is not offered repeatedly in the periode
```

Code-Ausschnitt 2.13: jBehave Story für die Speiseplanerstellung

Die Logik des Tests befindet sich in der Datei *CreateMenuForPeriodeTest.java*[21]. Die Klasse *CreateMenuForPeriodeTest* ist sehr lang, deshalb wird hier nur die Deklarationen der Methoden gezeigt und auf die wesentlichen Aspekte eingegangen.

```
public class CreateMenuForPeriodeTest {

    @Given("a menu contains $numberOfMealsPerDay meals per day
        for the next $numberOfDayPerWeek days")
    public void pressCreateButton(int numberOfMealsPerDay, int
        numberOfDayPerWeek) {
    }

    @When("the periode is $weeks weeks")
    public void offerThreeMenusForTheNextThreeWeeks(int weeks) {
    }

    @Then("expect $days days with $expectedNumberOfMeals dishes"
        )
```

[21] http://git.io/eId8yQ

```
12      public void menusContainsEnoughtMealsForThePeriode(int days,
            int expectedNumberOfMeals) {
13      }
14
15      @Then("each day with 3 dishes")
16      public void requiredNumberOfDishesPerDayAreOffered() {
17      }
18
19      @Then("each day contains a vegetarian dish, a dish with meat
            and a third alternativ")
20      public void menusContainsEnoughtMealsForThePeriode() {
21      }
22
23      @Then("each dish is not offered repeatedly in the periode")
24      public void eachDishIsNonRecurring() {
25
26      }
27  }
```

Code-Ausschnitt 2.14: Klasse: CreateMenuForPeriodeTest

Diese enthält für jede Zeile, die mit *Given*, *When* oder *Then* beginnt, eine Methode in der Test-Klasse. Die Verknüpfung zwischen Methode und Zeile erfolgt durch eine gleichnamige Annotation. Ein Satz der mit *Given* beginnt wird über die Annotaion *@Given* mit der Methode in der Test-Klasse verknüpft. Der Annotation wird als Parameter die Zeile aus der Story-Datei übergeben. Die Werte auf die geprüft werden soll werden in dem Satz durch Platzhalter ersetzt. Die Platzhalter werden mit dem Zeichen $ gekennzeichnet. Die Werte werden beim Aufruf der Test-Methode als Parameter übergeben.

Im Folgenden wird der Ablauf eines Tests beschrieben. Als erstes wird die Methode mit der Annotation *@Given* aufgerufen. In dieser Methode werden die benötigten Daten für den Test zusammengestellt. Dies bedeutet konkret, dass die *PeriodeConfi-*

guartion[22] initialisiert und die Speisen importiert werden müssen. In der Klasse *PeriodeConfiguartion* sind die Einstellungen für eine Planungsperiode enthalten. Zum Beispiel wie viel Wochen eine Periode umfasst, wie viele Speisen pro Tag angeboten werden sollen usw.

```
1         @Given("a menu contains $numberOfMealsPerDay meals per
              day for the next $numberOfDayPerWeek days")
2     public void pressCreateButton(int numberOfMealsPerDay, int
          numberOfDayPerWeek) {
3         periode = new PeriodeConfiguration();
4         periode.setNumberOfDaysPerWeek(numberOfDayPerWeek);
5         periode.setNumberOfMealsPerDay(numberOfMealsPerDay);
6         DishesImport importer = new DishesImport();
7         importer.importFiles();
8     }
```

Code-Ausschnitt 2.15: Mehtode pressCreateButton

Anschließend ruft der Acceptance Test die zu testende Funktionalität über die Interactor-Klasse auf. Mit der Klasse *CreateMenuForPeriodeTest* wird der Use Case *Speiseplan Erstellung* getestet. Somit wird die Klasse *CreateMenusInteractor* in der Testklasse instantiiert und die Methode *execute* aufgerufen. Dies erfolgt in der Methode *offerThreeMenusForTheNextThreeWeeks*.

```
1     @When("the periode is $weeks weeks")
2     public void offerThreeMenusForTheNextThreeWeeks(int weeks) {
3                 CreateMenusInteractor interactor = new
                      CreateMenusInteractor(createRequestBoundary()
                      );
4         interactor.setDao(DaoFactory.INSTANCE.makeCreateMenuDao
              ());
5         interactor.setMenuCreator(new MenuCreator());
6         interactor.execute();
7         offers = interactor.getResponse();
```

[22] http://git.io/vXZFsQ

```
8       }
```

Code-Ausschnitt 2.16: Methode offerThreMenusForTheNextThreeWeeks

Die Methoden mit der Annotation *@Then* prüfen anschließend auf Basis der zurückgegeben Offers die Anforderungen aus den User Stories 1 bis 5 und deren Akzeptanztests.

```
1  @Then("each day with 3 dishes")
2      public void requiredNumberOfDishesPerDayAreOffered() {
3                  //see on http://git.io/eId8yQ
4      }
5
6      @Then("each day contains a vegetarian dish, a dish with meat
           and a third alternativ")
7      public void menusContainsEnoughtMealsForThePeriode() {
8                  //see on http://git.io/eId8yQ
9      }
10
11     @Then("each dish is not offered repeatedly in the periode")
12     public void eachDishIsNonRecurring() {
13         //see on http://git.io/eId8yQ
14     }
```

Code-Ausschnitt 2.17: Then Methoden der Klasse CreateMenuForPeriodeTest

Mit Hilfe von jBehave lassen sich somit die Akzeptanztests automatisieren. Der zeitliche Aufwand dafür sollte jedoch nicht unterschätzt werden. Die Zeitersparnis beim Regressionstest zu jedem Iterationsende ist jedoch ebenfalls nicht unerheblich.

2.4.3 Integration Tests (Integrationstests)

Die vorletzte Ebene der Testpyramide sind die Integration Tests. Diese Testmethode wird in den Literatur sehr unterschiedlich bewertet.

Definition

Mit Integration Tests soll Code getestet werden, der nicht unter eigener Kontrolle steht. Darunter fallen fremde und firmeneigene Bibliotheken, die nicht vom Projektteam geändert bzw. angepasst werden können. Bei großen Anwendungen, die aus mehreren Komponenten bestehen, soll mit den Integrationstests ggf. die Zusammenarbeit der einzelnen Komponenten getestet werden, jedoch nicht die Business Logic. Dies geschieht bereits in den Unit- und Komponententests. Vgl. [Fre12, S.10] und [Mar11b, S. 117]

In dem Buch [Mye04] werden Integrationstest als nicht notwendig erachtet, da durch Unit- und Component Tests die Zusammenarbeit der verschieden Module (Units) bereits getestet sind. Vgl. [Mye04, S. 95]

Hinsichtlich der Integration von Fremdcode, kann mit den Integrationstests sichergestellt werden, dass eine neue Version des Fremdcodes weiterhin die erwarteten Ergebnisse liefert. Vgl. [Mar09a, S. 117ff]

Zusammenfassend kann festgehalten werden, dass mit Integrationstests die Kommunikation zwischen den Komponenten einer Software getestet werden kann. Dies ist vor allem für große Anwendungen mit mehreren Komponenten sinnvoll. Wenn Fremdcode zum Einsatz kommt, sollte mit den Integrationstests die genutzten Funktionen des Fremdcodes getestet werden, um sicherzustellen, dass diese wie erwartet funktionieren, auch nach einem Update des Fremdcodes.

Integration in agilen Sofwareentwicklungsprozess

Integrationstests für Fremdcode können durch sogenannte Learning Tests in den Softwareentwicklungsprozess integriert werden. Die Softwareentwickler können diese Tests während der Einarbeitung in die Bibliotheken bzw. Frameworks entwickeln. In den Tests werden die Methoden des Fremdcodes genauso aufgerufen wie später im Produktiv-Code. Dadurch wird die Einarbeitung effizienter und die Vorteile durch das Testen des Fremdcodes können genutzt werden. Wichtig ist, dass nur die benötigten Funktionen getestet werden und nicht die komplette Bibliothek, da dies sehr aufwendig sein kann. Vgl. [Mar09a, S. 118ff]

Frameworks

Für Integration Tests werden in der Regel die gleichen Frameworks wie für Unit Tests und Component Tests verwendet. Vgl. [Mar09a, S. 117] Somit werden in der Beispielanwendung die Integrationstests mit JUnit realisiert.

Anwendungsbeispiel

In der Beispielanwendung wird mit Kalenderwochen gearbeitet, da jeder Speiseplan einer Kalenderwoche und einem Jahr zugewiesen wird. Dazu wird eine Funktion benötigt, die zu einem gegebenen Datum, die Kalenderwoche und das Jahr liefert. Eine andere Funktion soll zu einer Kalenderwoche alle Werktage von Montag bis Freitag liefern. Die beiden Funktionen können mit Hilfe des Joda Time Frameworks[23] realisiert werden. Mit Hilfe eines Learning Tests kann sich der Softwareentwickler in die Bibliothek einarbeiten und gleichzeitig einen Integrationstest für diese entwickeln. Der Learning Test für die Beispielanwendung wird mit JUnit realisiert.

```
1  public class JodaTimeLearningTest {
2      private DateTime date;
3
4      @Before
5      public void before() {
6          date = new DateTime().withDate(2013, 4, 20);
7      }
8
9      @Test
10     public void testGetCalendarWeekFromDate() {
11         int calendarWeek = date.getWeekOfWeekyear();
12         assertEquals(16, calendarWeek);
13     }
14
15     @Test
16     public void testGetYearFromDate() {
```

[23] http://joda-time.sourceforge.net/

```
17          int year = date.getYear();
18          assertEquals(2013, year);
19      }
20
21      @Test
22      public void testGetDaysOfCalendarWeek() {
23          List<Date> workingDays = new ArrayList<>();
24          for (int day = DateTimeConstants.MONDAY;
25                  day <= DateTimeConstants.FRIDAY;
26                  day++) {
27              workingDays.add(date.withDayOfWeek(day).toDate());
28          }
29
30          assertWeekDays(workingDays);
31      }
32
33      private void assertWeekDays(List<Date> workingDays) {
34          assertTrue(workingDays.size() == 5);
35          int expectedDay = DateTimeConstants.MONDAY;
36          for (Date date : workingDays) {
37              assertEquals(expectedDay, new DateTime(date).
                    getDayOfWeek());
38              expectedDay++;
39          }
40      }
41  }
```

Code-Ausschnitt 2.18: Learning Test für das Joda Time Framework

Im ersten Test wird das Joda Time Framework für das Ermitteln der Kalenderwoche auf Basis eines übergebenen Datums genutzt. Die Assert-Methode testet, ob die richtige Kalenderwoche zurückgegeben wird. Im zweiten Test wurde getestet, wie mit dem Framework das Jahr zu der Kalenderwoche ausgelesen werden kann. Anschließend wurde im dritten Test die Funktion entwickelt, welche die Arbeitstage

einer Kalenderwoche zurück gibt.

Sollte eine neue Version des Joda Time Frameworks eingebunden werden, kann mit dem Test überprüft werden, ob die neue Version des Frameworks immer noch wie erwartet arbeitet. Durch das Vorgehen ist der zeitliche Aufwand für das Entwickeln von Integrationstest sehr gering, da diese mit der obligatorischen Einarbeitung in das Framework zusammenfällt.

2.4.4 System Tests (Systemtests)

Diese Tests gehören zu den schwierigsten und oft falsch verstandenen Testmethoden. Das liegt vor allem daran, dass die Systemtests nicht die Geschäftlogik testen sollen, sondern nur die Kommunikation der einzelnen Systemteile. Sie gehören zu der letzten Ebene der automatisierbaren Tests aus der Testpyramide. Vgl. [Mye04, S. 96ff]

Definition

Laut Robert C. Martin sollen Systemtests nur die Zusammenarbeit der verschieden Teile eines Systems testen. Vgl. [Mar11b, S. 118]

Myers entspricht ebenfalls in seinem Buch „The Art of Software Testing“ dieser Vorstellung. Er stellt fest, dass es keine generelle Methodik für Systemtests gibt. Es muss je nach Projekt entschieden werden was im Rahmen der Systemtests getestet werden soll. Die Durchführung von Systemtests erfolgt durch Systemtest-Experten. Vgl. [Mye04, S. 96 - 104]

Zusammenfassend kann festgehalten werden, dass mit Systemtests sichergestellt werden soll, dass alle Schichten einer Anwendung korrekt mit einander verknüpft sind. Darüber hinaus gibt es weitere Testmethoden die ggf. nicht automatisiert werden können (z. B. Usability Tests). Die korrekte Verknüpfung der verschieden Schichten und darin enthaltenen Module kann automatisiert werden.

Integration in agilen Sofwareentwicklungsprozess

Die Integration der Systemtests muss explizit vorgesehen werden. Sie lassen sich nicht wie die Unit-, Komponenten- und Integrationtests in die Entwicklung integrie-

ren. Somit muss für jede Anwendung geprüft werden, welche Systemtests sinnvoll sind. Dazu können die Kategorien für die Testfallentwicklung aus [Mye04, S. 96 - 104] hilfreich sein. Die Implementierung der automatisierbaren Systemtests sollte durch den Softwarearchitekten bzw. den leitenden Softwareentwickler implementiert werden. Vgl. [Mar11b, S. 118]

Frameworks

Um die Performance und die Zusammenarbeit der verschiedenen Systemteile zu testen, müssen Systemtests über den Delivery Mode auf die Anwendung zugreifen. Der Delivery Mode kann entweder die UI oder die Webservice-Schnittstelle sein.

Für das Testen der Zusammenarbeit der verschieden Systemteile eignen sich bei Anwendung mit Benutzeroberfläche Tools für UI Testing. Bei der Auswahl des UI Testing Tools soll berücksichtigt werden, dass es sich problemlos in die Entwicklungsumgebung integrieren lässt. Damit ist gemeint, dass die Softwareentwickler das Tool in ihrer gewohnten Umgebung (Programmiersprach(en) und IDE) einsetzen können. Ein UI Testing Tool das direkt im Web-Application-Framework integriert ist wird deshalb bevorzugt. Die folgenden Frameworks wurden untersucht.

1. Google Web Toolkit[24]
2. Play Framework[25]
3. Apache Wicket[26]
4. Restlet[27] und AngularJS[28]

Die ersten drei Frameworks bieten Funktionen für UI-Tests, somit sind sie hinsichtlich der Testautomatisierung für die Entwicklung der Beispielanwendung geeignet. Die Frameworks bieten die Möglichkeit sogenannte Rich Internet Applications (RIA) zu realisieren. Mit dem Google Web Toolkit werden die Web-Oberflächen ähnlich

[24] http://www.gwtproject.org/
[25] http://www.playframework.com/
[26] http://wicket.apache.org/
[27] http://restlet.org/
[28] http://angularjs.org/

wie Desktop-Anwendung programmiert. Vorteil ist, dass der Softwareentwickler nur Basiswissen über HTML, CSS und JavaScript benötigt. Die anderen beiden Frameworks unterstützen ebenfalls die Entwicklung von Webanwendungen, allerdings setzen beide Frameworks ein breiteres Wissen hinsichtlich der Webtechnologien voraus. Alle drei Frameworks haben eine umfangreiche Dokumentation und es gibt zahlreiche Bücher zu den Frameworks. Eine Anforderung an die Beispielanwendung ist, dass die Webanwendung ohne größeren Installationsaufwand gestartet werden kann. Dazu wurde ein Framework gesucht, dass einen eignen internen Webserver anbietet. Das Framework Restlet bietet einen solchen internen Webserver. Der Unterschied zu den vorherigen Frameworks ist, dass es sich bei Restlet um ein Framework handelt, welches nur die Implementierung eines REST-basierten Webservice unterstützt und keinerlei Unterstützung hinsichtlich der UI-Entwicklung bietet. Auf den ersten Blick war damit Restlet nicht geeignet für die Realisierung der Beispielanwendung. Jedoch stellte sich bei weiterer Recherche heraus, dass mit Hilfe eines clientseitigen JavaScript-Frameworks die Nachteile aufgehoben werden können. AngularJS ist ein clientseitiges JavaScript Framework, mit dem REST-basierte Webservices aufgerufen werden können. Die empfangenen Daten können dann mit JavaScritp, HTML und CSS dargestellt werden. Darüber hinaus sind die beiden Frameworks Karma[29] und Jasmine[30] integriert. Diese bieten die Möglichkeit, Unit-Tests für clientseitigen Code und End-To-End-Tests zu realisieren. Somit kann sowohl die UI als auch die Datenanbindung an Restlet getestet werden. Für die Beispielanwendung wird Restlet und AngularJS eingesetzt, da Restlet einen eignen Webserver enthält und AngularJS für die Oberflächenentwicklung ein sehr interessanter Ansatz[31] ist.

Anwendungsbeispiel

Ob der Speiseplan korrekt erstellt wird oder ob die Methode den aktuellen Speiseplan zurück liefert, wird bereits mit den Komponententests sichergestellt. Im Folgenden wird der Test für das Laden des aktuellen Speiseplans beschrieben.

Die JavaScript-Test-Datei *scenario.js* liegt im Projektordner unter *view/test/e2e*.

[29] http://karma-runner.github.io/
[30] http://pivotal.github.io/jasmine/
[31] http://www.youtube.com/watch?v=C7ZI7z7qnHU

Um den Test ausführen zu können, muss die Beispielanwendung gestartet werden. Anschließend kann unter *view/test/scripts* mittels dem e2e-test.sh (bzw. Windows e2e-test.bat) der Test gestartet werden. Voraussetzung ist, dass der Browser Google Chrome[32], der Webserver nodejs[33] und die Test-Laufzeitumgebung Karma[34] installiert sind.

Die Testdatei *scenario.js*[35] beinhaltet folgenden Test:

```
1  describe('Find', function() {
2
3      beforeEach(function() {
4          browser().navigateTo('/web/index.html');
5      });
6
7      it('Menu for Calendar Week', function() {
8                  expect(binding('menu.calendarWeek.week')).toBe(
                       moment().week().toString());
9                  expect(binding('menu.calendarWeek.year')).toBe(
                       moment().year().toString());
10                 expect(element('tbody.offers tr').count()).toBe
                       (3);
11
12     });
13 });
```

Code-Ausschnitt 2.19: Systemtest mit angularJs und Karma

Mit der Funktion, die an *beforeEach* übergeben wird, öffnet der Browser die Website der Anwendung. Anschließend wird die an *it* übergebene Methode ausgeführt. In dieser Methode werden die Tests aufgerufen. Die ersten beiden Zeilen der Methode prüfen ob die aktuelle Kalenderwoche angezeigt wird. In der dritten Zeile wird geprüft ob der Speiseplan drei Zeilen beinhaltet. So kann überprüft werden, dass der Delivery Mode korrekt mit der Business-Logic-Schicht und diese wiederum mit der

[32] https://www.google.com/intl/en/chrome/browser/
[33] http://nodejs.org/
[34] http://karma-runner.github.io/0.10/intro/installation.html
[35] http://git.io/ed8lDA

Persistence-Schicht verknüpft ist.

Der Aufwand für Systemtests ist sehr hoch, allerdings ist nur mit den Systemtests eine vollständige Testabdeckung über alle Schichten der Anwendung möglich. Weshalb sie nicht vernachlässigt werden sollten. Im Unterabschnitt 2.4.2 wurden die End-To-End Tests vorgestellt. Diese testen auf Basis der formulierten Akzeptanztests vom Delivery Mode ausgehend die Funktionen der Business Logic. Diese wären somit eine Alternative zu den Systemtests, da sie die Verknüpfung der einzelnen Schicht implizit testen und in den Softwareentwicklungsprozess ggf. besser implementiert werden können. Jedoch würde die Entwicklung der Component Tests aufwändiger werden.

3 Fazit

Zu den Fragen aus der Einleitung werden abschließend die wichtigsten Erkenntnisse aus der Projektarbeit zusammengefasst.

Welche Testarten gibt es und wie werden diese voneinander abgegrenzt? Die Testarten aus der Testpyramide können grundsätzlich automatisiert werden. Besonders die Unit-, Compoment- und Integration-Tests lassen einfach mit Hilfe von Frameworks wie JUnit und jBehave automatisieren.

Für die Entwicklung von automatisierten Tests ist es wichtig, dass die einzelnen Module und Komponenten unabhängig voneinander getestet werden können. Dies wird durch eine Softwarearchitektur erreicht, die eine lose Koppelung zwischen Modulen und Komponenten ermöglicht. Die folgenden Fachbücher liefern Konzepte und Design Patterns mit denen eine solche Architektur entwickelt werden kann.

1. Agile Software Development von Robert C. Martin ([Mar12])

2. Java Application Architecture von Krik Knoerenschild ([Kno12]

Mit TDD kann die Entwicklung von Unit Tests in einen agilen Softwareerstellungsprozess integriert werden. Wichtig ist, dass mit Unit Tests alle öffentlichen Funktionen einer Klasse, isoliert von anderen Klassen, getestet werden. Methoden die nicht öffentlich sind (private, protected) werden durch verschiedene Testfälle getestet. Darüber hinaus muss berücksichtigt werden, dass TDD eine andere Herangehensweise zur Lösung von programmatischer Aufgabenstellungen ist. Dementsprechend muss der Einarbeitungsaufwand berücksichtigt werden. Für die Einarbeitung in das Thema können folgende Fachbücher empfohlen werden:

1. Clean Code von Robert C. Martin ([Mar09a])

2. Test-Driven Development von Kent Beck ([Bec02])

3. Working Effectively with Legacy Code von Michael C. Feathers ([Fea11])

4. Growing Object-Oriented Software, Guided by Example von Steve Freeman ([Fre12])

Auf der Ebene der Component Tests können die Akzeptanztests automatisiert werden. Mit diesen kann sichergestellt werden, dass die Anforderungen der Stakeholder vollständig implementiert wurden. Gleichzeitig wird getestet, ob die einzelnen Klassen wie geplant zusammenarbeiten.

Wird für die Entwicklung einer Software Fremdcode integriert, so kann dieser mit Integration Tests getestet werden. Dadurch wird sichergestellt, dass der Fremdcode wie erwartet funktioniert. Wird im späteren Verlauf des Projekts eine neue Version des Fremdcodes eingebunden, kann mit Integration Test geprüft werden, ob die neue Version ebenfalls wie erwartet funktioniert. Das hat einen sehr positiven Einfluss auf die Wartbarkeit der Anwendung, weil schnell und sicher Updates von Fremdcode eingespielt werden können.

Eine besondere Herausforderung ist meiner Meinung nach die Automatisierung von System Tests. Tests welche die Zusammenarbeit der Schichten einer Anwendung testen lassen sich mit zeitlich hohen Aufwand automatisieren. Trotz des Aufwands ist nur so mit automatisierten Tests sichergestellt, dass alle Schichten der Anwendung wie erwartetet zusammenarbeiten. Jedoch kann mit einer erheblichen Zeitersparnis bei Regressionstests gerechnet werden.

Wie kann die Realisierung dieser Tests in einem agilen Softwareentwicklungsprozess integriert werden? Die Integration von Unit- und Component Tests ist durch den beschriebenen Test-Zyklus in Unterabschnitt 2.4.2 sehr effizient. Ausgehend von einem fehlgeschlagenen Component Test werden mit TDD die einzelnen Klassen entwickelt. Anschließend wird der Component Test wiederholt. Im Fall, dass er fehlschlägt, wird mit Hilfe der Unit Tests nach dem Fehler gesucht. Schlägt der Test nicht fehl, ist die geforderte Funktion realisiert.

Die Integration Tests lassen sich ebenfalls sehr gut in die Erstellung der Software integrieren. Wird mit Hilfe von Fremdcode eine bestimmte Funktion realisiert, muss der Softwareentwickler sich in der Regel in die Funktionsweise des Fremdcodes einarbeiten. Bei der Einarbeitung kann er Learning Tests nutzen. In diesen Tests prüft der Entwickler, ob der Fremdcode die erwarteten Ergebnisse liefert. Da die Learning Tests während der Einarbeitung entwickelt werden und diese auch unterstützen ist meiner Ansicht nach der zeitliche Aufwand für deren Entwicklung vernachlässigbar.

Für die Integration von System Tests gibt es meiner Meinung nach zwei Möglichkeiten. Die Erste ist die explizite Integration in den Softwareentwicklungsprozess. Die System Tests werden dann von den Softwarearchitekten oder den leitenden Softwareentwickler programmiert und angepasst. Die zweite Möglichkeit ist die implizite Integration durch die Component Tests, die vom Delivery Mode ausgehend, die Funktionen der Business Logic aufrufen und testen. Damit wird implizit die Zusammenarbeit der Module und Schichten getestet. Jedoch sollte berücksichtigt werden, dass sich dadurch der Aufwand für die Entwicklung der Component Tests erhöht.

Gibt es für die Automatisierung der Tests Frameworks? Es gibt sehr viele Frameworks, mit sehr unterschiedlichen Ansätzen für die Automatisierung von Test. Bei der Entwicklung der Componenten Tests gibt es die größten Unterschiede. Hier gibt es *Story-basierte*, *tabellen-basierte* und *End-To-End-Test* Ansätze. Die Wahl welcher Ansatz der Richtige ist kann meiner Ansicht nach nicht abschließend beantwortet werden. Dies hängt letztendlich vom Projekt und den Softwareentwicklern ab. Jedoch sollte sich das Framework gut in die Entwicklungsumgebung integrieren lassen, damit die Einarbeitung möglichst kurz gehalten werden kann.

Literaturverzeichnis

[Bec02] Kent Beck. *Test Driven Development*. Addision-Wesley Professional, 1 edition, 2002.

[ea10] Laudon et al. *Wirtschaftsinformatik*. Pearson Education, München, 2010.

[Fea11] Michael C. Feathers. *Working Effectively with Legacy Code*. Person Education, Boston, 1 edition, 2011.

[Fre12] Steve Freeman. *Growing Object-Oriented Software, Guided by Tests*. The Pragmatic Bookshelf, Boston, 6 edition, 2012.

[Kno12] Kirk Knoernschild. *Java Application Achritecture*. Pearson Education, Boston, 1 edition, 2012.

[Lig09] Prof. Dr.-Ing. Peter Liggesmeyer. *Software-Qualität, Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Heidelberg, 2 edition, 2009.

[Mar09a] Robert C. Martin. *Clean Code*. Pearson Education, Boston, 2 edition, 2009.

[Mar09b] Robert C. Martin. *Clean Code A Handbook of Agile Software Craftsmanship*. Pearson Education, Boston, 2009.

[Mar11a] Robert C. Martin. Clean architecture, 22. November 2011.

[Mar11b] Robert C. Martin. *The Clean Coder*. Pearson Education, Boston, 2011.

[Mar12] Robert C. Martin. *Agile Software Development*. Pearson Education, Boston, 2012.

[Mye04] Glenford J. Myers. *The Art of Software Testing*. John Wiley & Sons, Inc, 2 edition, 2004.

[Rot07] Johanna Rothman. *Manage It*. The Pragmatic Bookshelf, Dallas, 2 edition, 2007.

Eidesstattliche Erklärung

Ich erkläre hiermit gemäß §16 der Prüfungsordnung für den Projektarbeit VAWi, dass ich die vorliegende Masterarbeit mit dem Titel „Automatisierte Tests in agilen Softwareprojekten erklärt an einem Beispiel“ selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt, Zitate kenntlich gemacht und die Arbeit noch keiner anderen Stelle zu Prüfungszwecken vorgelegt habe.

Erlangen, 16.08.2013

Tobias Tatsch