BITMARCK®

Implementierung eines eigenen Testframeworks für die „Freien Anträge“ der Webanwendung bitGo_GS

Dokumentation der betrieblichen Projektarbeit
für die Abschlussprüfung zum Fachinformatiker Fachrichtung Anwendungsentwicklung

Auszubildender
Patrick Grunau
Dohlenhorst 4b
22453 Hamburg

Ausbildungsbetrieb
BITMARCK Technik GmbH
Hammerbrookstraße 38
20097 Hamburg

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Diese Dokumentation zur betrieblichen Projektarbeit ist im Rahmen meiner Ausbildung zum Fachinformatiker in der Fachrichtung Anwendungsentwicklung bei der BITMARCK Technik GmbH entstanden.
Alle Begriffe, die kursiv gekennzeichnet sind, werden im Glossar (siehe Anhang 8.1.) erklärt. Während der Projektarbeit habe ich auch auf externe Quellen zugegriffen, diese habe ich im Literaturverzeichnis (siehe Anhang 8.3.) aufgeführt. Die Dokumentation habe ich mit Microsoft Word 2013 erstellt. Für die Erstellung der Präsentation werde ich Microsoft PowerPoint 2013 verwenden.

Neben der BITMARCK Technik GmbH und vier weiteren Business-Units besteht die BITMARCK-Unternehmensgruppe aus einer Holding. Die Unternehmensgruppe ist ein Full-Service-Anbieter im IT-Markt der gesetzlichen Krankenversicherung (GKV) und beschäftigt aktuell rund 1.400 Mitarbeiter.
Die BITMARCK Technik GmbH hat ihren Hauptsitz in Hamburg und bietet eine hohe Fachkompetenz im Bereich der GKV.
Neben dem größten Kunden, der DAK-Gesundheit (DAK-G), gehören viele weitere Krankenkassen zum Kundenstamm der BITMARCK Unternehmensgruppe.

Während meiner Ausbildungszeit wurde ich in der Abteilung GKV-Services eingesetzt.
GKV-Services gehört dem Bereich Softwareentwicklung innerhalb der BITMARCK Technik GmbH an und betreut diverse Anwendungen der Kunden, die überwiegend in *Java* und *Cobol* entwickelt werden.

2. Projektbeschreibung

Die Webanwendung *bitGo_GS* ist eine Online-Geschäftsstelle, die in nächster Zukunft von vielen verschiedenen Krankenkassen für ihre Versicherten eingesetzt wird. Jede dieser Krankenkassen hat ihre eigene *bitGo_GS*-Anwendung, die sie mit einem *Content Management System* (CMS) administriert. Das CMS ermöglicht es den Redakteuren der Krankenkassen, die Anwendung nach individuellen Wünschen anzupassen und/oder mit neuen Funktionen zu versehen. Der wichtigste Anwendungsfall hierbei ist das Erstellen von Webformularen, die es dem Versicherten ermöglichen, sich zu einem konkreten fachlichen Thema mit seiner Krankenkasse online auszutauschen. Man spricht in diesem Zusammenhang von sogenannten „Freien Anträgen“, da die jeweilige Krankenkasse den Inhalt und die Gestaltung dieser Formulare selbst bestimmen kann. Ein solcher Antrag - z.B. ein Antrag auf Kinderkrankengeld - kann dann von dem Versicherten ausgefüllt und online an die Krankenkasse übermittelt werden.

Vorteile der Anwendung *bitGo_GS* sind unter anderem, dass die Versicherten nicht mehr von den Öffnungszeiten der Kassen abhängig sind und ein umständlicher Versand von Unterlagen auf dem Postweg an die Krankenkasse entfällt.

Nachdem der Versicherte einen Antrag abgeschickt hat, wird dieser mittels eines *Webservice* Aufrufs an das Kern-System *iskv_21c* geleitet, wo der Antrag anschließend validiert und das Ergebnis über den *Webservice Response* zurückgeschickt wird.

Da *bitGo_GS* sich noch in der Weiterentwicklung befindet, müssen immer wieder neue „Freie Anträge“ erstellt werden. Um eine hohe Qualität der Anwendung zu gewährleisten, ist es notwendig, jeden dieser Anträge regelmäßig in vielen verschiedenen Konstellationen zu testen. Bei der mühsamen und zeitintensiven Erstellung der entsprechenden Testfälle soll mein entwickeltes Testframework die Projektmitarbeiter zukünftig unterstützen.

2.1. Projektbegründung

Ein wichtiger Bestandteil der Qualitätssicherung einer Software ist, diese ausreichend zu testen.
Insbesondere die Anwendung *bitGo_GS* muss ausführlich getestet werden, da bei dieser Anwendung nicht nur die jeweilige Krankenkasse, sondern auch die Versicherten der Kasse zu den Kunden zählen. Die Kunden erwarten eine stabile und zuverlässige Software.

Ein Softwaretest hilft dabei, die Erfüllung der Anforderungen von *bitGo_GS* zu prüfen.
Durch das Testframework kann nicht nur getestet werden ob die „Freien Anträge“ fehlerfrei funktionieren, sondern z.B. auch ob bei falschen Angaben durch einen Anwender die richtige Fehlermeldung erscheint.

Die „Freien Anträge“ sind der wesentlichste Bestandteil der Anwendung. Damit stellen sie auch den Hauptbestandteil der zu testenden Objekte dar. Hier fällt der größte Teil der Aufwände im Bereich *Unit-Testing* an. Und da sich die prinzipiellen Arbeitsabläufe bei „Freien Anträgen“ immer wiederholen, bietet es sich an, hier mittels meines Frameworks die Erstellung von Testfällen zu optimieren. Durch die Optimierung ergibt sich eine Zeitersparnis.

Die erstellten Testfälle können durch mein Testframework alle auf einmal gestartet werden, diese werden dann automatisch nacheinander ausgeführt (Testautomatisierung).

Momentan werden die Ergebnisse von Testläufen nicht dokumentiert, dies soll mit meinem Projekt nun automatisch geschehen. Dadurch kann gegeben falls auf Testläufe aus der Vergangenheit zurückgegriffen werden.

2.2. Projektziel

Das Ziel meiner Projektarbeit ist, ein Testframework zu entwickeln, mit dessen Hilfe Testfälle für „Freie Anträge“ der Anwendung *bitGo_GS* effizient erstellt werden können. Hierbei sollen immer wiederkehrende Aufwände beim Erstellen von Tests durch das *Framework* übernommen werden, um redundante Tätigkeiten zu vermeiden. Außerdem soll die Erstellung von Testfällen möglichst auch ohne tiefere Kenntnisse der Programmiersprache *Java* möglich sein, um im besten Fall diese Tätigkeit auch von Personen ohne Entwickler-Know-how durchführen lassen zu können.

Bei der Test-Erstellung soll es möglich sein, nicht nur einen einzelnen *Webservice* anzusprechen und dessen Antwort auszuwerten, sondern durch aktive Überprüfung der Transaktions-Ergebnisse (z. B. durch Abfragen des *Backends*) die Richtigkeit der Prozesse zu verifizieren. Dementsprechend soll nach dem Test eine weitere Überprüfung des Testfallergebnisses möglich sein.

Neben der Möglichkeit, eine komplette Testreihe zu starten, muss auch die Option bestehen, einzelne Tests auszuführen.

Nach jedem Testlauf soll ein Report generiert werden, in dem alle Testfälle aufgeführt sind. Eine zusätzliche Statistik gibt Auskunft darüber, wie viele der Tests fehlerfrei durchgeführt wurden und bei wie vielen Fehler auftraten.

Einzelne Tests sollen, z. B. weil sie sich noch in der Entwicklung befinden, ignoriert werden können. Dazu soll eine Konfigurationsmöglichkeit geschaffen werden.

2.3. Zielgruppe

Die Zielgruppe des Testframeworks sind die Mitarbeiter des *bitGo_GS* Teams innerhalb der BITMARCK Technik GmbH. In diesem Team sind neben den Entwicklern der Anwendung auch fachliche Mitarbeiter, die ebenso wie die Entwickler in der Lage sein sollen, Testfälle erstellen zu können.

3. Analyse

3.1. Ist-Zustand

Momentan werden für jeden Testfall mühsam einzelne *JUnit* Klassen entwickelt. Es ist nicht möglich mehrere Testfälle in eine Klasse zu schreiben. Aktuell können nur Entwickler mit Zugriff auf das Entwicklungssystem neue Testfälle erstellen und diese ausführen.
Zudem ist es aktuell nicht möglich, einen Report über die Testergebnisse zu generieren, so dass der Entwickler manuell jeden Testfall auswerten muss.

3.2. Soll-Konzept

Um die folgenden muss-Anforderungen an mein Projekt zu identifizieren, habe ich ein Fachgespräch mit einem Entwickler aus dem *bitGo_GS* Team geführt.
Das Testframework soll als ein weiteres Modul in die Anwendung *bitGo_GS* eingebettet werden. Mit Hilfe dieses Moduls, soll es möglich sein alle Tests zu starten oder einzelne Tests anzustoßen. Für jeden einzelnen „Freien Antrag“ soll eine Datei existieren, in der Daten über den zu testenden *Webservice* und weitere wichtige Daten, um die Testfälle ausführen zu können, enthalten sind.

Weiterhin soll für jeden einzelnen Testfall in der Datei eine Beschreibung existieren sowie die Möglichkeit, diesen Testfall bei der Ausführung des Testframeworks ignorieren zu können.
Am wichtigsten ist natürlich, dass in der Datei angegeben werden kann, welches Ergebnis am Ende des Testfalls erzielt werden soll. Dieses schließt ein, dass natürlich eine Validierung des Ergebnisses stattfinden muss.

Wenn am Ende alle Testfälle durchgelaufen sind, wird eine Datei erzeugt, die einen Report über alle ausgeführten Testfälle und dessen Ergebnisse enthält.

3.3. Kosten-Nutzen-Analyse

Die Kosten-Nutzen-Analyse ist ein Werkzeug der betriebswirtschaftlichen Analyse, welches zur Gegenüberstellung von Kosten und Nutzen verwendet wird. Hierbei werden die für das Projekt angefallenen Kosten, wie zum Beispiel *Personentage* (PT) und eventuelle andere Anschaffungen summiert und dem Nutzen gegenübergestellt.

3.3.1. Kosten

Die Kosten für die Entwicklung in der BITMARCK Technik GmbH, werden in *Personentagen* angegeben. Ein *Personentag* entspricht dabei dem Aufwand von 7,8 Stunden.
Da von diesem Projekt nur die BITMARCK Technik GmbH betroffen ist, werden die Kosten auch nur für die BITMARCK Technik GmbH berechnet. Somit ist die BITMARCK Technik GmbH sowohl Kunde als auch Auftraggeber für dieses Projekt.
In diesem Fall rechne ich mit dem internen Stundensatz der BITMARCK Technik GmbH, welcher laut der Abteilung Controlling 520,- €/PT beträgt.

$$\frac{520\ €/PT}{7{,}8\ Stunden} \approx 66{,}67\ €/Stunde$$

Die in das Projekt investierte Zeit beträgt 70 Stunden. Zusammen mit dem Stundensatz der BITMARCK Technik GmbH werden die Kosten für meine Bemühungen in dem Projekt berechnet.

$$66{,}67\ €/Stunde * 70\ Stunden = 4.666{,}90\ €$$

Neben meinen Aufwänden an dem Projekt, muss noch die Arbeitszeit eines Kollegen aus dem *bitGo_GS* Team berechnet werden. Der Kollege hat mit mir drei Stunden ein Fachgespräch über mein Projekt geführt, da der Kollege zur BITMARCK Technik GmbH gehört, wird auch hier der interne Stundensatz verwendet.

$$66{,}67\ €/Stunde * 3\ Stunden = 200{,}01\ €$$

Zum Abschluss des Projekts habe ich drei Kollegen des *bitGo_GS* Teams mein Testframework in Form einer Schulung übergeben. Diese Schulung hatte einen Zeitaufwand von drei Stunden. Die Kosten der Kollegen müssen ebenfalls mit Hilfe des internen Stundesatz berechnet werden.

$$(66{,}67\ €/Stunde * 3\ Stunden) * 3\ Kollegen = 600{,}03\ €$$

Um nun die Gesamtkosten berechnen zu können muss mein Aufwand mit dem der Kollegen zusammengerechnet werden.

$$4.666{,}90\ € + 200{,}01\ € + 600{,}03\ € = 5.466{,}94\ €$$

Da ich mein Projekt mit Hilfe von der schon vorhandenen Hardware und Software umgesetzt habe sind für diesen Posten keine Kosten entstanden.

3.3.2. Nutzen

Der Nutzen meines Testframeworks liegt in der Zeitersparnis pro neu implementierten Testfall im Vergleich zum alten Verfahren. Ein Vergleich mit der Umsetzung eines identischen Testfalls mit Hilfe meines Testframeworks und danach mit Hilfe des „alten“ Verfahrens ergab folgende Ergebnisse:

- „Altes“ Verfahren: 35 Minuten
- Projekt Testframework: 15 Minuten

Daraus lässt sich nun die Zeitersparnis berechnen.

$$35\ Minuten/Testfall - 15\ Minuten/Testfall = 20\ Minuten/Testfall$$

Um eine tatsächliche Ersparnis berechnen zu können, muss noch festgestellt werden wie viele Testfälle pro Jahr umgesetzt werden. Nach einer Anfrage an das *bitGo_GS* Team, wurde mir berichtet, dass pro Jahr neun Anträge umgesetzt werden sollen und pro Antrag 30 Testfälle erstellt werden. Damit lässt sich die Anzahl der Testfälle pro Jahr berechnen:

$$9\ Anträge/Jahr * 30\ \ Testfälle/Antrag = 270\ Testfälle/Jahr$$

Jetzt berechne ich zu den gesamten Testfällen noch die zuvor errechnete gesparte Zeit hinzu. Da ich die Zeit in Stunden benötige, um sie in Relation mit den internen Stundensatz der BITMARCK Technik GmbH zu stellen, wird das Ergebnis durch 60 geteilt:

$$270\ Testfälle/Jahr * 20\ Minuten/Testfall = 5.400\ Minuten/Jahr = \ 90\ Stunden/Jahr$$

Die jährlichen Ersparnisse des Projektes pro Jahr berechne ich mit Hilfe des internen Stundensatzes der BITMARCK Technik GmbH:

$$90\ Stunden/Jahr * 66{,}67\ €/Stunde = \ 6000{,}30\ €/Jahr$$

3.3.3. Gegenüberstellung von Kosten und Nutzen

Die folgende Tabelle zeigt eine Gegenüberstellung von den berechneten Kosten und dem monetären Nutzen. Da die Eingliederung meines Projektes in die Anwendung *bitGo_GS* erst im nächsten Jahr erfolgen soll, kann der Nutzen erst ab den Jahr 2017 einberechnet werden.

Die Kosten sind für dieses Projekt nur einmalig zu berechnen, da es mit dem umgesetzten Stand in Betrieb genommen werden soll, weitere Kosten für dieses Projekt sind nicht zu erwarten.

	2016	2017	2018
Kosten	5.466,94 €	0,00 €	0,00 €
Nutzen	0,00 €	6.000,30 €	6.000,30 €
Bilanz	**-5.466,94 €**	**533,36 €**	**6533,66 €**

Tabelle 1: Gegenüberstellung von Kosten und Nutzen

3.3.4. Amortisation

Der Zeitraum, der benötigt wird, um mit der Einsparung der umgesetzten Anforderungen die Summe aller Kosten decken, wird als Amortisation bezeichnet.
Diese lässt sich mit folgender Formel berechnen:

$$\frac{Kosten}{Nutzen/Jahr} = Amortisation$$

$$\frac{5.466{,}94\ €}{6.000{,}30\ €} \approx \ 0{,}91\ Jahre$$

Die Amortisationszeit beträgt ungefähr 0,91 Jahre. Dies bedeutet, dass in etwa 11 Monaten sind die Kosten dieses Projekts gedeckt sind.

4. Designphase

Bei dem Design eines Projektes wird ein Konzept entworfen, auf dessen Basis die Software umgesetzt werden soll.
Die Anforderungen an das Testframework legen folgendes Design nahe.
Da die Testfälle durch Parameter definiert werden sollen bietet es sich an, diese in Form einer Datei an das Testframework zu übergeben, dadurch müssen keine *Java* Klassen für jeden einzelnen Testfall erstellt werden.
Somit ist kein Wissen über *Java* Klassen erforderlich, um einen Testfall zu erstellen.
Alle wesentlichen Informationen, wie beispielsweise welcher *Webservice* bzw. „Freier Antrag“ angesprochen werden soll und wie die Testkonstellation (Aufrufparameter, Erwartungswerte) genau aussehen sollen, sind in den Dateien enthalten, die wir Parameterdateien nennen.

Die Logik des Testablaufs wird im meinem *Framework* abgebildet. Ein einzelner Testfall besteht dabei im Allgemeinen aus dem Aufruf eines mehr oder weniger beliebigen *Webservices* gefolgt von einer anschließenden Analyse des Rückgabe-Objekts. Um zu vermeiden, dass in der Klasse des Testablaufs schon alle *Webservice* Objekte und dessen Methoden bekannt und hinterlegt sein müssen, mache ich mir die Programmiertechnik der *Reflection* zu Nutze.
Dies ermöglicht mir die *Webservice* Aufrufe generisch zu halten. Wenn also eine Änderung oder eine Neuimplementierung von *Webservices* erfolgt, muss mein *Framework* nicht angepasst werden.

Um dieses Projekt realisieren zu können, verwende ich externe Bibliotheken, die ich gemäß den beschriebenen Anforderungen an das Projekt ausgesucht habe und die ich im Folgenden kurz vorstellen möchte:

- TestNG
- GSON
- Hamcrest

4.1. Verwendung von TestNG

TestNG ist ein *Framework* zum Testen von *Java* Programmen. Meine Wahl fiel auf dieses *Framework*, weil es einen sogenannten „DataProvider“ enthält. Durch diesen können die für den Testlauf notwendigen Daten zur Verfügung gestellt werden. Weiterhin kann mit Hilfe von TestNG ein Report über alle ausgeführten Testfälle erstellt werden.

4.2. Verwendung von GSON

Die Bibliothek GSON ermöglicht die *Serialisierung* und *Deserialisierung* von *JSON* Dateien. *JSON* Dateien sollen in diesem Projekt als Parameterdateien dienen. Mit der Hilfe von GSON werden die *JSON* Dateien eingelesen und die Parameter werden dem „DataProvider“ zur Verfügung gestellt.

4.3. Verwendung von Hamcrest

Das *Framework* Hamcrest bietet verschiedene *Matcher* um ein Testergebnis mit dem erwartenden Ergebnis zu vergleichen.

5. Realisierungsphase

In diesem Abschnitt der Dokumentation werde ich darlegen wie ich das Projekt entwickelt habe. Nach dem ich meine Entwicklungsumgebung und die Implementierung der neuen

Packages vorgestellt habe, werde meine Realisierung veranschaulichen, vom Start eines Tests bis zu Verifizierung eines Tests.

5.1. Entwicklungsumgebung

Ich verwende als Entwicklungsumgebung (engl. IDE – Integrated Development Environment) die IntelliJ IDEA in der Version 2016.2.1.

5.2. Implementierung der neuen Packages

Ich habe mich entschieden, ein neues *Package* in unserem Projekt zu erstellen. Dieses *Package* habe ich „bitgo-test“ genannt. In diesem *Package* habe ich drei weitere Verzeichnisse untergeordnet, die ich in der nachfolgenden Tabelle erklären werde:

Name	Beschreibung
test	Enthält den Logikteil der Anwendung.
ws.antraege	Hier liegt für jeden Antrag die Konfiguration (z.B. Objektreferenz des *Webservice* oder Name der Parameterdatei).
resources	Enthält alle Parameterdateien.

Tabelle 2: Packages

5.3. Vor dem Start eines Testfalls

Für jeden Antrag muss eine Klasse erstellt werden die von der Klasse „AbstractBaseWSClientTest“ erbt, diese Klasse enthält die Logik für den kompletten Testablauf. Ich habe als Bespiel die Klasse „ZuzahlungsbefreiungAntragITest“ (siehe Abbildung 1) gewählt, hier ist unteranderem ein *Array* mit einem *Webservice* enthalten der später benötigt wird sowie der Pfad zur Parameterdatei mit den Testfällen.

Bevor ein Testfall überhaupt getestet werden kann muss noch ein Login in den geschützten Bereich von *bitGo_GS* geschehen. Dies geschieht mittels meiner Login Methode, die sich in der *Java* Klasse „AbstractAuthWSClientTest“ befindet. Die benötigten Benutzerdaten werden in der Klasse „TestDataProvider“ abgelegt.
Dieser Arbeitsablauf wird von der Methode „setUpClass()“ erledigt. Diese besitzt die *Annotation* „@BeforeClass“. Das bedeutet, dass beim Start des Tests diese Methode automatisch aufgerufen wird.

5.4. Implementierung des „DataProvider“ und der Parameterdatei

Wie in der Designphase (Punkt 4) beschrieben, gibt es einen „DataProvider“. Dieser wurde wie in der Abbildung (siehe Abbildung 2) gezeigt implementiert. Als erstes benötigt der „DataProvider“ eine Liste von Parametern die gesetzt werden sollen diese werden in der *ArrayList* „list“ abgelegt.

Die *ArrayList* hat den Typen „WSTestparameter“, dieser Typ hat drei Klassen die jede verschiedene Variablen enthalten.
Folglich eine Aufstellung der Variablen:

Klasse Parameter		
Typ	**Name**	**Beschreibung**
String	description	Beschreibung eines Testfalls.
String	webserviceName	Name des *Webservice* der verwendet wird.
String	webserviceMethodName	Methode die im *Webservice* verwendet werden soll.
String []	methodParamClasses	Typen der Parameter des gewählten Antrags.
Object []	methodParamValues	Wert der Parameter des gewählten Antrags.
String	expectedExceptionClass	*Exception* die beim Verarbeiten des Testfalls erwartet wird.
String	expectedResultClass	Klasse die nach dem Verarbeiten des Testfalls erwartet wird.
Object	expectedResultValue	Wert der nach dem Verarbeiten des Testfalls erwartet wird.
MethodParameter []	methodOnResult	Siehe Tabelle 4
String	matcherName	Namen des *Matchers* siehe Punkt 5.7

Tabelle 3: Die Klasse „Parameter“

Klasse MethodParameter		
Typ	**Name**	**Beschreibung**
String	methodName	Methode die auf das Ergebnis angewendet werden soll.
String []	methodParamClasses	Klassen der Variablen die, die Methode braucht.
Object []	methodParamValues	Werte der Variablen die, die Methode braucht.
String	expectedExceptionClass	*Exception* die beim Verarbeiten des Testfalls erwartet wird.
String	expectedResultClass	Klasse die nach dem Methodenaufruf erwartet wird.
Object	expectedResultValue	Wert der nach dem Methodenaufruf erwartet wird.
String	matcherName	Namen des *Matchers* siehe Punkt 5.7

Tabelle 4: Die Klasse „MethodParameter“

Klasse Ignore		
Typ	**Name**	**Beschreibung**
String	ignore	Beschreibung weshalb dieser Testfall ignoriert wird. Standard Wert ist „null“ (Test wird nicht ignoriert).

Tabelle 5: Die Klasse „Ignore“

Ein Objekt der Klasse „WSTestparameter“ repräsentiert einen einzigen Testfall. Für die Parameterdateien habe ich als Datentyp *JSON* gewählt. Dieser Typ ist sehr kompakt und übersichtlich. In einer Parameterdatei ist ein *Array* enthalten, in dem mehrere Testfälle stehen können. In *JSON* wird ein *Array* mit einer „[“ geöffnet und mit einer „]“ wieder geschlossen.

Ein einzelner Testfall wird als ein Objekt dargestellt. In *JSON* wird ein Objekt mit einer „{“ geöffnet und wird mit einer „}“ geschlossen. In einem Testfall Objekt sind noch weitere Objekte integriert. Diese sind vom Typen „Parameter“ (siehe Tabelle 3),
„MethodParameter“ (siehe Tabelle 4) oder „Ignore“ (siehe Tabelle 5).
Zusammenfassend besteht eine Parameterdatei aus einer Liste von Objekten des Typen „WSTestparameter“. Mit Hilfe der *Deserialisierung* entstehen hieraus wieder *Java* Objekte.

Eine Bespiel Parameterdatei ist im Abbildungsverzeichnis (siehe Abbildung 3) zu sehen.

Nun muss der „DataProvider“ über den angegebenen Pfad die Parameterdatei finden. Anschließend, wird mit Hilfe des *Framework* GSON, die *Deserialisierung* der Parameterdatei durchgeführt.
Wenn dieser Vorgang abgeschlossen ist, kann mit der Ausführung des Tests begonnen werden.

5.5. Implementierung des Testablaufs

Der eigentliche Test wird mit der Methode „runTest()“ gestartet, diese hat die *Annotation* „@Test“ dadurch erkennt meine IDE IntelliJ, dass es sich um einen ausführbaren Test handelt. Dieser Methode werden außerdem die benötigten Parameter, die der „DataProvider“ ermittelt hat, zum ausführen des Tests übergeben.

Als nächstes muss der richtige *Webservice* ermittelt und danach die richtige Methode auf dem *Webservice* aufgerufen werden. Die Auswahl des *Webservice* erledigt die Methode „getWebserviceByName“ (siehe Abbildung 4), diese bekommt einen *Webservice* Namen übergeben und ermittelt mit Hilfe der *Webservice* Liste (diese liegt in der Klasse des Antrags) das richtige Objekt aus.
Wenn der richtige *Webservice* gefunden ist, wird mit Hilfe der *Reflection* die Methode des *Webservice* aufgerufen.
Nach dem die Methode ausgeführt wurde, wird das Ergebnis mit Hilfe der *Matcher* überprüft.

5.6. Implementierung der Ergebnisverifizierung

Zum Schluss wird die Methode „compare()“ (siehe Abbildung 5) aufgerufen. Hier habe ich die Schnittstellen des *Frameworks* Hamcrest zu Hilfe genommen.
In der Klasse „Parameter“ und „MethodParameter“ können *Matcher* (Für eine Übersicht der *Matcher* siehe Punkt 5.7.) angegeben werden.
Wenn in der Klasse Parameter die Variable „matcherName“ gesetzt wurde, bedeutet dies, dass der *Webservice* als Antwort ein Objekt liefert der mit den *Matchern* verifiziert werden soll.
Sollte die Klasse „MethodParamter“ nicht „null“ sein so wird hier mittels *Reflection* auf dem Ergebnis eine Methode ausgeführt die den richtigen Wert liefert, der verifiziert werden soll.
Zum Schluss erstellt das *Framework* TestNG automatisch einen Report (siehe Abbildung 6), der in einem Internet Browser angezeigt werden kann.

5.7. Matcher für die Verifizierung

Die *Matcher* sind in der Klasse „WSTestfaelle“ zu finden, in dieser Klasse habe ich die *Matcher* von dem *Framework* Hamcrest erweitert, so dass sie unseren Anforderungen entsprechen. Bei jedem *Matcher* wird das Ergebnis des Testlaufs mit einem Erwartungswert verglichen.

Hier eine kleine Übersicht der *Matcher* :

Name des *Matchers*	Beschreibung
isA	Prüft ob die Klasse des Objekts gleich ist.
hasSizeTest	Prüft ob die Größe der Liste der Angabe entspricht.
listNotEmptyTest	Prüft ob die Liste nicht leer ist.
listEmptyTest	Prüft ob die Liste leer ist.
isTest	Prüft ob die Werte sich gleichen.
greaterThanTest	Prüft ob die Zahl größer ist.
lessThanTest	Prüft ob die Zahl kleiner ist.
notNullTest	Prüft ob der Wert nicht „null“ ist.
nullTest	Prüft ob der Wert „null“ ist.

Tabelle 6: Die Matcher

6. Qualitätssicherung

Bei der Anwendung *bitGo_GS* handelt es sich um eine Webanwendung an der noch intensiv weiterentwickelt wird. Um die Kollegen meines Teams nicht bei der Weiterentwicklung zu stören habe ich mir eine eigne Entwicklungsumgebung lokal auf meinen Rechner eingerichtet. So konnten die Kollegen weiter arbeiten ohne, dass sie von mir gestört wurden.

Für alle Tests habe ich auf eine *iskv_21c* Anwendung zurückgegriffen, die für das Team *bitGo_GS* freigegeben wurde, dementsprechend konnte ich schon erstellte Testszenarien einbinden.

7. Abschluss des Projekts

7.1. Übergabe an das Team bitGo_GS

Nach der Implementierung des Projekts, habe ich den Mitarbeitern mittels einer Live Demo gezeigt, wie ein Testfall zu implementieren ist. Des Weiteren habe ich demonstriert wie nur ein einzelner Antrag oder alle Testfälle auf einmal getestet werden können. Um eine korrektes Ausfüllen der Parameterdatei zu gewährleisten, habe ich zusätzlich eine Musterdatei mit übergeben.

7.2. Dokumentation des Projekts

Um Weiterentwicklungen am meinem Projekt zu erleichtern, habe ich meinen Quellcode mit Hilfe von *Javadoc* und zusätzlichen Kommentaren dokumentiert.

7.3. Soll-Ist-Abgleich

In der folgenden Tabelle habe ich die geplante Projektzeit der tatsächlich angefallenen Zeit gegenübergestellt.

	Soll	Ist
Analysephase	**8**	**8**
Ist-Zustand	2	2
Soll-Konzept	3	3
Anforderungen	1	1
Kosten-Nutzen-Analyse	2	2
Designphase	**8**	**7**
Design des Testframeworks	8	7
Realisierungsphase	**30**	**30**
Implementieren des Testframeworks	25	25
Einlesen von *JSON* Dateien implementieren	5	5
Testphase und Übergabe	**14**	**15**
JSON Dateien mit Testdaten erstellen	5	5
Durchführen von Tests	4	4
Fehlerbehebungen	3	3
Übergabe an das *bitGo_GS* Team	2	3
Dokumentation	**10**	**10**
Summe	**70**	**70**

Tabelle 7: Soll-Ist-Abgleich

Der oberen Darstellung ist zu entnehmen, dass während der Projektphase einige Änderungen in der Planung stattfanden.
So habe ich beim Design des Projekts Zeit gespart, da mir nach der Analyse des Ist-Zustandes und des Soll-Konzeptes viel schneller als erwartet ein passendes Design für das Projekt eingefallen ist.

Ich habe gegenüber der Planung im Projektantrag zwei Änderungen vorgenommen.
Zum einem habe ich den Punkt „Übergabe an das *bitGo_GS* Team“ als neuen Punkt aufgenommen. Dabei habe ich eine Einführung in das Testframework für die Kollegen gegeben, die ein wenig länger dauerte, als geplant.

Die Testdaten werden erst später in der Testphase benötigt, deswegen habe ich den Punkt „*JSON* Dateien mit Testdaten erstellen“ in die Testphase verschoben.

Die Änderungen in den Bereichen waren klein und glichen sich aus. Somit wurde die Fertigstellung meines Projekts innerhalb des vorgegebenen Zeitrahmens zu keinem Zeitpunkt gefährdet.

7.4. Fazit

Da ich seit letztem Jahr in dem Team von *bitGo_GS* mitarbeite, fiel es mir nicht ganz so schwer, dieses Projekt in *bitGo_GS* zu integrieren. Ich habe in dem Zeitraum auch schon viel über die Anwendung gelernt, jedoch kannte ich die „Freien Anträgen“ zu diesem Zeitpunkt noch nicht. Dieses Projekt gab mir somit die Möglichkeit mich in diesen Bereich der Anwendung einzuarbeiten.

Ich hatte zu der Zeit zwar schon einige meiner Anwendungen mit Unterstützung von *JUnit* getestet, hatte aber noch keine Erfahrung mit alternativen Test-*Frameworks* wie z. B. TestNG, was ich im Rahmen dieses Projekts nachholen konnte.

Faszinierend für mich war es, die spezielle *Java*-Technologie der *Reflection* kennen zu lernen, mit der ich bis zu diesem Zeitpunkt noch keine Berührung hatte. Darüber hinaus habe ich gelernt, wie wichtig es ist, sich Gedanken zu machen wie eine Software umzusetzen ist und welche *Frameworks* gegeben falls bei der Umsetzung unterstützen können.

Zusammenfassend bin ich mit dem Ergebnis der Projektarbeit sehr zufrieden und habe eine Menge während der Umsetzung dazugelernt.

7.5. Ausblick

Im Laufe des Projekts wurde mir bewusst, wie wichtig das Testen unserer Anwendung eigentlich ist. Die Anwendung *bitGo_GS* befindet sich, wie bereits erwähnt, noch in der Weiterentwicklung und viele Funktionalitäten sind aktuell noch in Planung. Auch bei diesen zukünftigen Funktionalitäten besteht Testbedarf, der mit meinem Testframework abgedeckt werden könnte.

Beim Testen meines Projektes ist aufgefallen, dass bei vielen Testfällen als Ergebnis eine lange Liste von Objekten zurückgegeben wird. Momentan ist es noch nicht möglich, dass mein Testframework über die Liste iterieren kann und nebenbei prüfen kann ob ein bestimmtes Objekt in dieser Liste vorhanden ist. Eine Umsetzung dieser Funktionalität ist bereits in Planung.

Bisher ist es noch nicht möglich, bei der Beschreibung eines Testfalls für einen Parameter eine Liste von Werten vorzugeben und den Testfall für jeden dieser Werte durchführen zu lassen. Dementsprechend könnte noch umgesetzt werden, dass in der Parameterdatei für einen Testfall mehrere Werte für eine Variable angegeben werden können und der Testfall darauf wiederholt ausgeführt wird.

Eine Überlegung wäre es mein Projekt für alle *Java* Anwendungen der BITMARCK Technik GmbH zur Verfügung zu stellen. Dies ist momentan nicht möglich, weil mein *Framework* noch zu sehr auf die Anwendung *bitGo_GS* zugeschnitten ist.

8. Anhang

8.1. Glossar

Begriff	Erklärung
Annotation	Element der Programmiersprache Java, erlaubt die Einbindung von Metadaten in den Quellcode.
Array	Ist ein Feld mit mehreren Elementen desselben Typs.
ArrayList	Ist in der Programmiersprache Java eine Sammlung mehrerer Elemente desselben Typs
Backend	Bezeichnet eine Software-Anwendung die auf einem Server läuft und Daten verwaltet.
bitGo_GS	= **BIT**MARCK-**G**eschäftsprozesse **O**nline **G**eschäft**s**stelle Anwendung für die mein Projekt entwickelt wird.
Cobol	= **Co**mmon **B**usiness **O**riented **L**anguage Eine Programmiersprache die früher öfter eingesetzt wurde.
Content Managment System	Eine Software zur Verwaltung, Bearbeitung und Organisation von Inhalten (Content).
Deserialisierung	Umwandlung eines Datenstroms (z.B. Datei) in Objekte. Gegenteil der Serialisierung.
Exception	Fehlermeldung in der Programmierung.
Framework	Ein Programmiergerüst zur Unterstützung der Softwareentwicklung.
iskv_21c	BITMARCK-Kernsystem für das Krankenkassenmanagment.
Java	Eine objektorientierte Programmiersprache
Javadoc	Ein Dokumentationswerkzeug, welches aus kommentierten Java Quelltexten eine HTML basierte Dokumentation erstellt.
JSON	= **J**ava**S**cript **O**bject **N**otation Ist ein kompaktes Datenformat mit lesbarer Textform.
JUnit	Ist ein Framework zum Testen von Java Programmen.
Matcher	Vergleichsoperatoren
Package	Enthält Java Klassen.
Personentage	Bezeichnet die Arbeitsleistung einer Person.
Reflection	Ermöglicht zur Laufzeit in der objektorientierten Programmierung Informationen über Klassen oder Instanzen abzufragen.
Response	Bezeichnet die Antwort eines Webservice.
Serialisierung	Umwandlung eines Objekts in eine Datei. Gegenteil der Deserialisierung.
Unit-Testing	Auf Deutsch Modultest. Wird angewendet um Einzelteile (Module) von Software zu testen.
Webservice	Ist eine Softwareanwendung die im Netzwerk bereitgestellt wird und Anfragen bearbeitet.

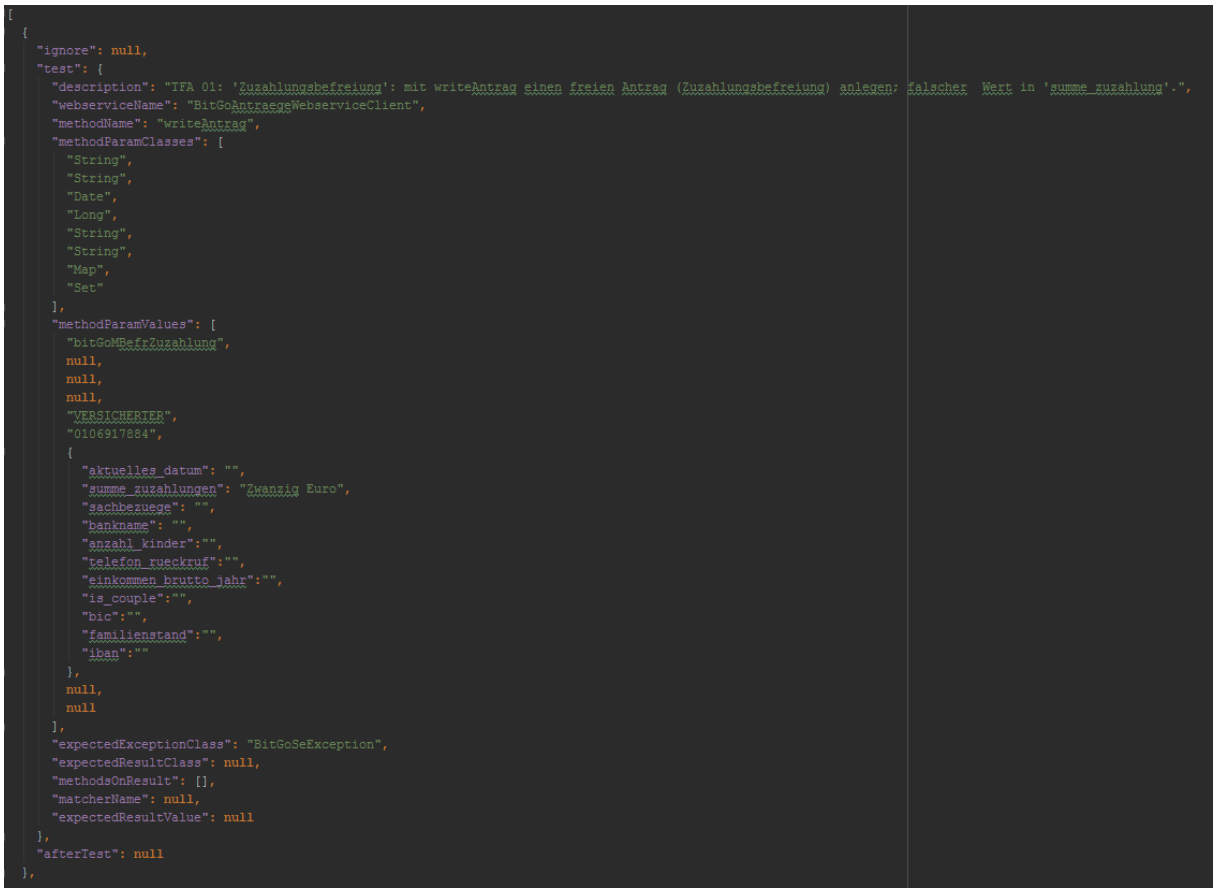
8.2. Abbildungsverzeichnis

```java
public class ZuzahlungsbfreiungAntragITest extends AbstractBaseWSClientTest{


        private final String PATHTODATAFILE = "de/bitmarck/bitgo/gs/mwf/ws/antraege/bitGoGSAntraege/AntragZuzahlungsbefreiung.json";

        @Inject
        private BitGoAntraegeFacade bitGoAntraegeFacade;

        @Override
        protected Object[] getWebservices() {
            Object[] oArray = {bitGoAntraegeFacade};
            return oArray;
        }
        @Override
        protected String getPathToDatafile() { return PATHTODATAFILE; }

        @Override
        protected boolean doLogin() { return true; }

        @Override
        protected boolean doLogout() { return true; }

        @Override
        protected boolean withAttachement() { return false; }

}
```

Abbildung 1: Die Klasse „ZuzahlungsbefreiungAntragITest“

```
@DataProvider(name = "fileDataProvider")
public Iterator<Object[]> parameters() throws Exception {
    //Testparameter aus Datei lesen
    ArrayList<WSTestparameter> list = new ArrayList<~>();
    //Reader für die JSOn Datei
    BufferedReader reader = null;
    try {
        //Pfad zur JSON Datei
        ClassPathResource cpr = new ClassPathResource(getPathToDatafile());
        //Lese die JSON Datei
        reader = new BufferedReader(new InputStreamReader(cpr.getInputStream()));
        //Neue Instanz vom Framework GSON
        Gson gson = new Gson();
        //TypeTokens für die Variablen
        Type typeOfT = new TypeToken<ArrayList<WSTestparameter>>() {
        }.getType();
        //Deserialisierung
        list = gson.fromJson(reader, typeOfT);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (com.google.gson.JsonSyntaxException e) {
        e.printStackTrace();
    } finally {
        try {
            if (reader != null) {
                reader.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    //Daten in Liste von Object-Arrays wandeln
    ArrayList<Object[]> outList = new ArrayList<~>();
    //Für jedes WSTestparameter Objekt (Darstellung von einem Testfall) in der Liste
    for (WSTestparameter dataSet : list) {
        Object[] objArray = new Object[5];
        objArray[0] = (dataSet.getIgnore() == null) ? "" : dataSet.getIgnore().description;
        //Wenn es kein Test Abschnitt gibt in dem Objekt dann werfe eine Exception...
        if (dataSet.getTest() == null) {
            throw new Exception("Json-Eingabedaten ohne \"test\"-Abschnitt");
        } else {
            //...sonst hole dir die Beschreibung für den Testfall
            objArray[1] = dataSet.getTest().description;
        }
        //Hole die restlichen Daten für den Testfall
        objArray[2] = dataSet.getTest();
        objArray[3] = dataSet.getAfterTest();
        objArray[4] = dataSet.getIgnore();
        outList.add(objArray);
    }
    return outList.iterator();
}
```

Abbildung 2: Der „DataProvider“



```
[
  {
    "ignore": null,
    "test": {
      "description": "TFA 01: 'Zuzahlungsbefreiung': mit writeAntrag einen freien Antrag (Zuzahlungsbefreiung) anlegen; falscher  Wert in 'summe zuzahlung'.",
      "webserviceName": "BitGoAntraegeWebserviceClient",
      "methodName": "writeAntrag",
      "methodParamClasses": [
        "String",
        "String",
        "Date",
        "Long",
        "String",
        "String",
        "Map",
        "Set"
      ],
      "methodParamValues": [
        "bitGoMBefrZuzahlung",
        null,
        null,
        null,
        "VERSICHERTER",
        "0106917884",
        {
          "aktuelles_datum": "",
          "summe_zuzahlungen": "Zwanzig Euro",
          "sachbezuege": "",
          "bankname": "",
          "anzahl_kinder":"",
          "telefon_rueckruf":"",
          "einkommen_brutto_jahr":"",
          "is_couple":"",
          "bic":"",
          "familienstand":"",
          "iban":""
        },
        null,
        null
      ],
      "expectedExceptionClass": "BitGoSeException",
      "expectedResultClass": null,
      "methodsOnResult": [],
      "matcherName": null,
      "expectedResultValue": null
    },
    "afterTest": null
  },
```

Abbildung 3: Parameterdatei des Antrags „Zuzahlungsbefreiung“

```
//Suche den angegebenen Webservice
private Object getWebserviceByName(String webserviceName) throws ClassNotFoundException{
    //Wenn der WebserviceName nicht gegeben wurde werfe eine Exception
    if(webserviceName != null && !webserviceName.equals("")) {
        for (Object o : getWebservices()) {
            if (webserviceName.equals(o.getClass().getName())) {
                return o;
            }
        }
    }
    throw new ClassNotFoundException("Service konnte nicht gefunden werden.");
}
```

Abbildung 4: Die Methode „getWebserviceByName()“

```java
//Benutze die Hamcrest Matcher zum vergleichen
private void compare(Object result, Object expected, String matcherName, String expectedClass) {
    //Wenn das erwartete Objekt eine Zahl sein soll und das Resultat keine Collection ist,
    // dann wandel die Zahl in ein Double damit wird verglichen
    if (expected instanceof Number && !(result instanceof Collection)) {
            result = ((Number) result).doubleValue();
    }
    if (matcherName.equals("greaterThan")) {
        WSTestfaelle.greaterThanTest(result, expected);
    } else if (matcherName.equals("is")) {
        WSTestfaelle.isTest(result, expected);
    } else if(matcherName.equals("isA")) {
        WSTestfaelle.isA(result, expectedClass);
    } else if (matcherName.equals("lessThan")) {
        WSTestfaelle.lessThanTest(result, expected);
    } else if (matcherName.equals("null")) {
        WSTestfaelle.nullTest(result);
    } else if (matcherName.equals("notNull")) {
        WSTestfaelle.notNullTest(result);
    } else if (matcherName.equals("hasItem")) {
        WSTestfaelle.hasItemTest((Collection) result, expected);
    } else if (matcherName.equals("hasSize")) {
        WSTestfaelle.hasSizeTest((Collection) result, ((Number) expected).intValue());
    } else if (matcherName.equals("listNotEmpty")) {
        WSTestfaelle.listNotEmptyTest((Collection) result);
    } else if (matcherName.equals("listEmpty")) {
        WSTestfaelle.listEmptyTest((Collection) result);
    } else {
        //Wenn Matcher falsch geschrieben wurden wird ein Fehler geworfen
        throw new NoSuchElementException("Der Matcher ist nicht bekannt.");
    }
}
```

Abbildung 5: Die Methode „compare()“

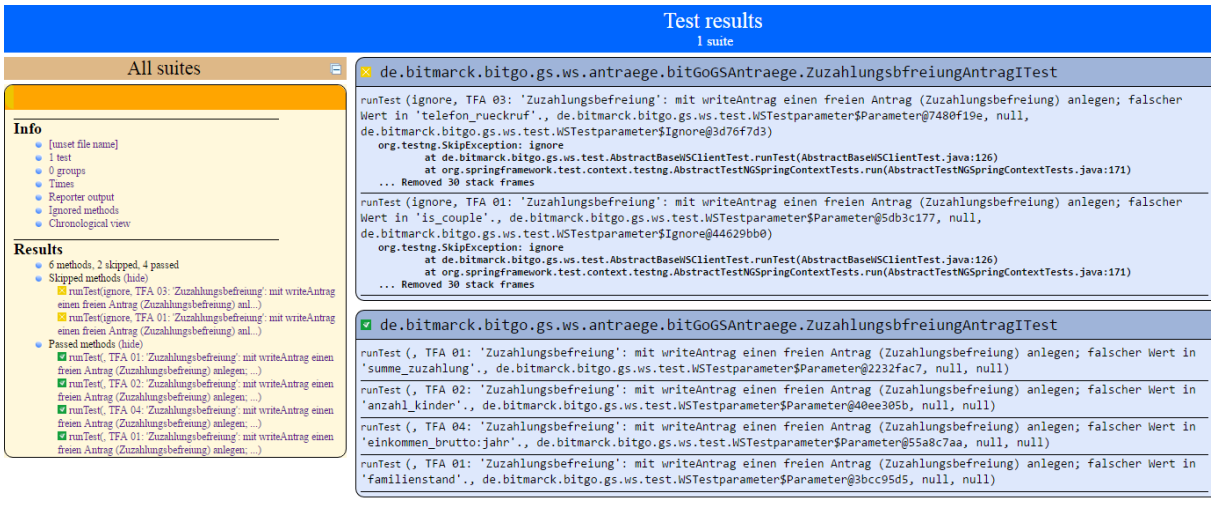


Abbildung 6: Der Report

8.3. Literaturverzeichnis

- https://de.wikipedia.org/wiki/Serialisierung (konsultiert am 09.12.16)
- https://de.wikipedia.org/wiki/Framework (konsultiert am 09.12.16)
- https://de.wikipedia.org/wiki/JavaScript_Object_Notation (konsultiert am 09.12.16)
- https://de.wikipedia.org/wiki/Reflexion_(Programmierung) (konsultiert am 09.12.16)
- https://de.wikipedia.org/wiki/JUnit (konsultiert am 09.12.16)