FH Vorarlberg
University of Applied Sciences

Entwicklung eines Test Case Management Systems für Gantner Instruments Messsysteme

Bachelorarbeit
zum Erlangen des akademischen Grades

Bachelor of Science in Engineering (BSc)

Fachhochschule Vorarlberg
Informatik - Software and Information Engineering

Betreut von
Dipl.-Ing. Dr. techn. Ralph Hoch

Vorgelegt von
Marco Prescher

Dornbirn, am 20. Mai 2023

Kurzreferat

Entwicklung eines Test Case Management Systems für Gantner Instruments Messsysteme

Die Entwicklung technischer Systeme ist ein komplexer und kostspieliger Prozess. Daher ist es wichtig, dass die Produkte vor der Auslieferung an den Kunden gezielt und sorgfältig getestet werden. Dies wird durch knappe Ressourcen und Termindruck oftmals vernachlässigt, oder nur unzureichend durchgeführt.

Durch ein Test Case Management System ist es möglich, dass Features vor dem Release eines Produktes getestet und abgenommen werden müssen. Das wiederum ermöglicht, dass neu implementierte Features gründlich getestet werden und somit zu einer hohen Qualität des Produktes beitragen. Dadurch kann stabile Hardware sowie effiziente und gut strukturierte Software an den Kunden ausgeliefert werden.

Ziel dieser Arbeit ist es, ein Test Case Management System zu entwickeln, mit dem es möglich ist Features von Messsystemen der Firma Gantner Instruments zu beschreiben. Zusätzlich soll das Test Case Management System in der Lage sein, Testergebnisse von einem bereits bestehenden Test Automation System zu empfangen und diesen Features zuzuordnen.

Das in dieser Arbeit entwickelte Test Case Management System ermöglicht eine präzise Beschreibung von Features und Testergebnissen sowie deren Speicherung in einer Datenbank. Über eine standardisierte API können Daten aus dem System abgefragt und von externen Systemen verarbeitet werden. Durch sorgfältige Dokumentation von Features und Testergebnisse können Statistiken und Reports erstellt werden, die wiedergeben, welche Features erfolgreich getestet wurden. Dadurch ist eine Überwachung des Testfortschritts möglich und es kann anhand von Zielvorgaben festgestellt werden, welche Features Release-fähig sind und in das Produkt mit übernommen werden können.

Abstract

Development of a Test Case Management System for Gantner Instruments Measurement Systems

The development of technical systems is a complex and costly process. It is therefore important that the products are tested specifically and carefully before delivery to the customer. Due to scarce resources and deadline pressure, this is often neglected, or only carried out inadequately.

A test case management system makes it possible for features to be tested and approved before a product is released. This in turn enables newly implemented features to be thoroughly tested, thus contributing to the high quality of the product. As a result, stable hardware as well as efficient and well-structured software can be delivered to the customer.

The goal of this thesis is to develop a Test Case Management System, which makes it possible to describe features of measurement systems of the company Gantner Instruments. In addition, the test case management system should be able to receive test results from an already existing test automation system and assign them to features.

The Test Case Management System developed in this thesis allows a precise description of features and test results as well as their storage in a database. Using a standardized API, data can be retrieved from the system and processed by external systems. By carefully documenting features and test results, statistics and reports can be generated that reflect which features were successfully tested. This makes it possible to monitor the progress of testing and to determine which features are ready for release and can be incorporated into the product on the basis of targets.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface

JSON JavaScript Object Notation

TCMS Test Case Management System

TDD Test Driven Development

EF Entity Framework

ORM Object Relational Mapping

OAS OpenAPI Spezifikation

HTTP Hypertext Transfer Protocol

MVP Minimum Viable Product

UI User Interface

DDD Domain Driven Design

DHCP Dynamic Host Configuration Protocol

GI Gantner Instruments

TAS Test Automation System

IDE Integrated Development Environment

ID Identifier

SQL Structured Query Language

DTO Data Transfer Object

DI Dependency Injection

1 Einleitung

> *"Time is money."*
>
> — Benjamin Franklin (Franklin (1748))

In der Softwareentwicklung kann es durch Fehler und Mängel zu Verzögerungen und dadurch in weiterer Folge zu Verlusten kommen. Um dieses Risiko zu minimieren, plant die Firma *Gantner Instruments* (GI) ein *Test Case Management System* (TCMS) einzuführen. Diese Arbeit beschäftigt sich mit der Planung und Entwicklung eines TCMS und dem Vergleich verschiedener vorhandener Lösungen. Das entwickelte TCMS ist dabei speziell auf die Anforderungen von GI angepasst.

1.1 Motivation

Die Entwicklung technischer Systeme ist ein komplexer Prozess, der eine hohe Qualität für das zu entwickelnde System erfordert, um den Anforderungen der Kunden gerecht zu werden. Damit diese Qualität auch gewährleistet werden kann, müssen Fehler sowie Mängel identifiziert und ausgebessert werden. Ein TCMS bietet eine Lösung, um diesen Prozess zu vereinfachen und effektiver zu gestalten. Durch die Verwendung eines TCMS können Angestellte aus verschiedenen Abteilungen, wie z.B. der Software-, Hardware-, Support- oder Marketingabteilung, die Qualität des Produkts gemeinsam verbessern.

1.2 Problemstellung

Durch die angesprochene Komplexität technischer Systeme ist bekannt, dass Fehler, die erst spät im Entwicklungsprozess entdeckt werden, viel kostspieliger zu beheben sind als Fehler, die frühzeitig identifiziert und behoben werden. (Westland (2002))

Infolgedessen suchen Unternehmen nach Lösungen, um den Testprozess effektiver zu gestalten und Fehler früher im Entwicklungszyklus zu identifizieren. TCMS bietet eine solche Lösung, indem es Entwicklern und Testern ermöglicht, Testfälle effizient zu planen und zu verwalten sowie Testergebnisse zu erfassen, darzustellen und zu analysieren. Obwohl TCMS in der Industrie weit verbreitet sind und einige fertige Lösungen auf dem Markt erhältlich sind, gibt es jedoch nicht immer eine Lösung die einfach in bestehende Systeme oder Entwicklungsprozesse integriert werden kann. Zusätzlich können spezielle firmeninterne Anforderungen oftmals nicht abgedeckt werden. Diese Probleme stellen Hindernisse dar, die die Einführung von TCMS in einem Unternehmen erschweren.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, ein TCMS zu entwickeln und in die bestehende Infrastruktur von GI zu integrieren. Zudem wird untersucht was ein TCMS auszeichnet und was dieses enthalten soll. Hiermit ergeben sich drei relevante fragen:

- Wie kann man Features, Testfälle und Testimplementierungen beschreiben?
- Wie diese in Datenbanken zur weiteren Verarbeitung speichern?
- Wie können Testläufe mit diesen Beschreibungen verknüpft werden?

Insbesondere möchten wir die folgenden Ziele erreichen:

- Die Vor- und Nachteile eines TCMS zu identifizieren, analysieren und dokumentieren.
- Empfehlungen für die erfolgreiche Implementierung und Integration eines TCMS in der Softwareentwicklung zu geben, einschließlich der Identifizierung bewährter Praktiken.

Durch die Erfüllung dieser Ziele trägt diese Arbeit bei, das Verständnis für die Verwendung eines TCMS in der Produktentwicklung zu verbessern und Unternehmen dabei zu unterstützen, den Testprozess zu optimieren und die Qualität ihrer Produkte zu verbessern.

2 Stand der Technik

Dieses Kapitel gibt einen Überblick über den aktuellen Stand der Technik von Testarten, TCMS sowie deren Verwendung und Integration mit bestehenden Systemen. Weiters werden Technologien, auf denen das in dieser Arbeit entwickelte TCMS aufbaut, beschrieben.

2.1 Testarten

Während der Entwicklung eines Produkts kommen unterschiedliche Testarten zum Einsatz. Atlassian (2023) beschreibt sieben unterschiedliche Testverfahren die verschiedene Aspekte eines Produkts testen. In *Software Testing/Qualitätssicherung - Alle Methoden und Tools* (2023) wird beschreiben, dass dabei zwischen *Funktionalen* und *Nicht-Funktionalen* Tests unterschieden wird. Zu der funktionalen Testfamilie zählen beispielsweise Unit-Tests und Integrationstests. Nicht funktionale Testarten sind beispielsweise Leistung-, Last-, Stress- oder Usability-Tests.

Einige der am häufigsten vorkommenden Testarten sind:

- Unit-Tests
- Integrationstests
- Funktionstests
- Leistungstests

2.1.1 Unit-Tests

Unit-Tests sind Tests die beispielsweise Methoden einer Klasse mit unterschiedlichen Parametern testen. Sie sind automatisierbar und können von einer Continuos-Integration-Pipeline durchgeführt werden. Diese Tests ermöglichen eine kontinuierliche Überprüfung einzelner Methoden im Entwicklungsprozess und unterstützen das frühzeitige Finden von Fehlern.

2.1.2 Integrationstests

Integrationstests sind Tests, die sicherzustellen, dass verschiedene Module oder Services problemlos miteinander interagieren können. Durch diese Tests kann die Funktionalität einzelner Teile der Anwendung überprüft werden, wie beispielsweise die Interaktion mit einer Datenbank oder der Zusammenarbeit von Microservices.

2.1.3 Funktionstests

Funktionstests werden verwendet, um ausschließlich Ergebnisse einer gegebenen Funktion zu überprüfen. Der Unterschied zum Unit-Test ist, dass Funktionstests Spezifikationen überprüfen und Unit-Tests Code. Beispielsweise werden Spezifikation vor der Testausführung festgelegt und diese dann mit den Ergebnissen verglichen.

Während Integrationstests beispielsweise nur prüfen, ob Datenbankabfragen generell möglich sind, wird bei einem Funktionstest ein bestimmter Wert aus der Datenbank abgerufen und dieser mit den angegebenen Spezifikationen geprüft.

2.1.4 Leistungstests

Leistungstests prüfen das Verhalten eines Systems unter verschiedenen Lastprofilen. Häufig wird Zuverlässigkeit, Geschwindigkeit, Skalierbarkeit und Reaktionsfähigkeit einer Anwendung getestet. Außerdem können Leistungstests mögliche Engpässe in einer Anwendung identifizieren.

2.2 Black-Box und White-Box Testing

Sowohl *Black-Box* als auch *White-Box* Tests werden in der Software Entwicklung häufig verwendet, um Fehler zu identifizieren und die Qualität eines Produkts zu evaluieren. Dabei gibt es zwischen den beiden wichtige Unterschiede wie Abbildung 2.1 veranschaulicht.

Bei White-Box Testing handelt es sich um Unit-Tests die den Code, die Struktur und das Desgin des zu testeten Produkts überprüfen. Wobei der Inhalt des Codes für den Tester einsehbar ist. White-Box Testing bezieht sich dabei nur auf die interne Funktion der Software.

Bei Black-Box Testing wird die interne Struktur, das Design und die Implementierung nicht berücksichtigt. Hier werden nur die Ausgaben oder Reaktionen von dem System geprüft. Black-Box Testing bezieht sich hiermit nur auf die externe Funktion der Software. (Nidhra und Dondeti (2012, Seite 12))

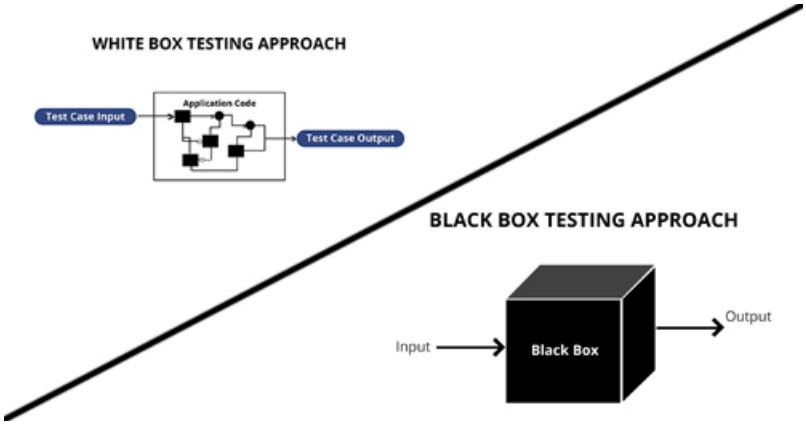

Abbildung 2.1: Black-Box/White-Box Testing (Quelle: Khandelwal (2019))

2.3 Test Driven Devlopment

Test Driven Development (TDD) ist ein Konzept, bei dem Tests zuerst geschrieben werden und erst anschließend eine passende Implementierung erstellt wird. Diese beinhaltet genau soviel, dass der Test erfolgreich durchgeführt werden kann. Abbildung 2.2 zeigt einen TDD Zyklus. TDD bietet daher mehrere Vorteile (Ammann und Offutt (2016)):

- Geschriebener Code kann überarbeitet oder verschoben werden, ohne dass die Gefahr besteht, Funktionalität zu beschädigen.
- Die Tests selbst werden durch die Implementierung getestet.
- Die Anforderungen können mit geringerem Aufwand umgesetzt werden, da nur die benötigte Funktion geschrieben wird.

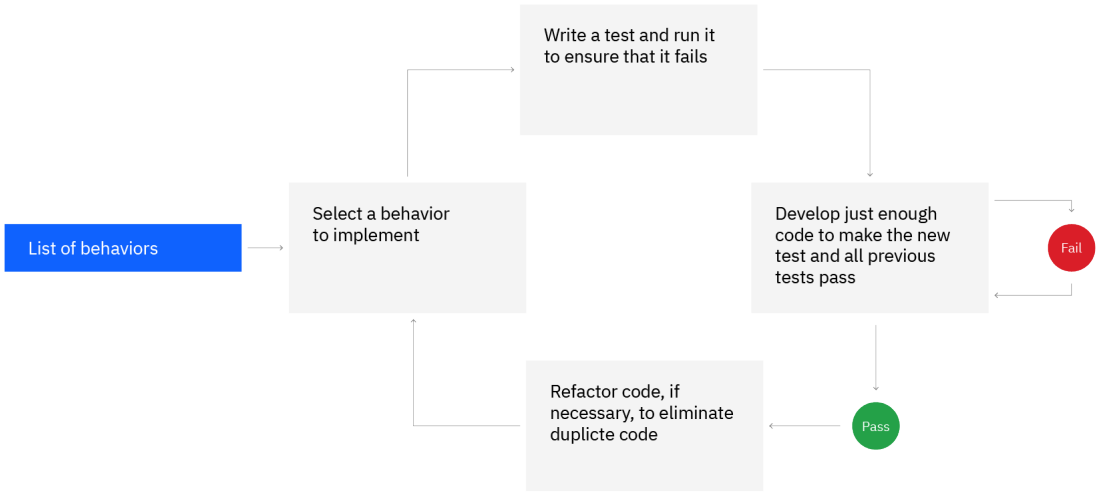

Abbildung 2.2: Test Driven Development cycle (Quelle: *Test-driven development - IBM Garage Practices* (2023))

2.4 Test Case Management System

Ein Test Case Management System (TCMS) ist eine Software, mit dem Test-Teams für ein bestimmtes Projekt oder eine Anwendung Testfälle verwalten, organisieren und analysieren können. Es hilft bei der Planung, Überwachung und Dokumentation von Tests und ermöglicht es, Testfälle sicher und effizient zu verwalten.

Ein TCMS ist ein wichtiges Werkzeug, um einen strukturierten und effektiven Testprozess zu gewährleisten und den Qualitätsstandard einer Anwendung zu verbessern. Es verfügt über Basisfunktionen wie Testfall-Erstellung, Testfall-Verwaltung, Testfall-Ausführung und Ergebnisberichterstattung. Weiters kann ein solches System auch eine integrierte Umgebung für die Zusammenarbeit von verschiedene Abteilungen in einer Firma bereitstellen.

Zusätzlich zu den Basisfunktionen sind weitere wichtige Merkmale (Lead (2023)):

- Attraktive Benutzeroberfläche und benutzerfreundliches Design
- Nachvollziehbarkeit
- verbesserte Zeitplanung und Organisation für Releases durch Reports
- Überwachung und Metriken
- Flexibilität

2.4.1 Allgemeine Vorteile von Test Case Management Systemen

Einer der Hauptvorteile eines TCMS besteht darin, dass sie den Testprozess verbessern. Sie unterstützen auch die Kontrolle der Gesamtkosten, indem sie Testautomatisierung nutzen, um einen reibungslosen Ablauf zu gewährleisten.

Einige Vorteile von TCMS bezüglich der Testausführungsläufe sind (Lead (2023)):

- Sie geben einen besseren Überblick über das zu testende System, halten den gesamten Prozess auf Kurs und koordinieren die Testaktivitäten.
- Sie helfen bei der Feinabstimmung des Testprozesses, indem sie die Zusammenarbeit, Kommunikation und Auswertung unterstützen.
- Sie dokumentieren Aufgaben, Fehler sowie Testergebnisse und vereinfachen den Prozess, indem sie alles in einer einzigen Anwendung erledigen.
- Sie sind skalierbar und können eingesetzt werden, wenn die Testaktivitäten umfangreicher und komplexer werden.

2.4.2 Überblick von bestehenden Test Case Management Systeme

Auf dem Markt gibt es eine Vielzahl fertiger und direkt verwendbaren TCMS, die die Verwaltung von Tests vereinfachen.

Einige bekannte TCMS sind in Abbildung 2.3 dargestellt und hier aufgelistet:

- *TestRail – Orchestrate Testing. Elevate Quality.* (2023)

  Webbasiertes TCMS, ist zentralisiert und hat folgende Features:

  - Test Planung, Verwaltung und Ausführung
  - Echtzeit Berichterstattung und Analysen
  - Rückverfolgbarkeits- und Abdeckungsberichte für Anforderungen, Tests und Fehler

- *PractiTest - Test Management Platform to Manage all Your QA Efforts* (2023)

  Webbasiertes TCMS und hat folgende Features:

  - Test Planung, Verwaltung und Ausführung
  - Echtzeit Test Status Update
  - Berichterstattung
  - Dashboards
  - Integrationsmöglichkeit mit Automation Tools

- *Zephyr Test Management Products | SmartBear* (2023)

  Webbasiertes TCMS, kann in JIRA integriert werden und hat folgende Features:

  - Test Planung, Verwaltung und Ausführung
  - Test Automatisierung
  - Echtzeit Visualisierung von Projektstatus

- *Tricentis qTest for Unified Test Management* (2023)

  Webbasiertes TCMS und hat folgende Features:

  - Test Planung, Verwaltung und Ausführung
  - Migrationsmöglichkeit von alten Test Management Lösungen
  - Integrationsmöglichkeit mit Jenkins, Azure Pipelines, Bamboo oder jedem anderen CI/CD-Tool
  - Anpassbare Dashboards, um über alle Releases, Projekte oder Programme im gesamten Unternehmen zu berichten
  - Berichte per E-Mail oder URL teilen

Welches Tool am besten geeignet ist, hängt dabei von der jeweiligen Anwendung und den spezifischen Anforderungen ab.

Abbildung 2.3: Liste von aktuell führenden Test Case Management Systemen/Tools (Quelle: *7 Best Test Management Tools* (2023))

2.5 Verwendete Technologien

Dieser Abschnitt gibt eine Übersicht über die verwendeten Technologien, die für die Implementierung des in dieser Arbeit entwickelten TCMS eingesetzt worden sind.

2.5.1 MariaDB

MariaDB ist eine weit verbreitete relationalen Open-Source-Datenbanken. Sie wurde auf einem Fork von MySQL basierend von den ursprünglichen Entwicklern von MySQL entwickelt. Der Fokus von MariaDB liegt auf Leistung, Stabilität und Offenheit. Zu den jüngsten Erweiterungen gehören Clustering mit Galera Cluster 4, Kompatibilitätsfunktionen mit der Oracle-Datenbank und temporäre Datentabellen, mit denen

Daten zu jedem beliebigen Zeitpunkt in der Vergangenheit abgefragt werden können. (*MariaDB documentation* (2023))

2.5.2 Docker

Docker ist eine Plattform mit der Entwickler einfach und schnell Container erstellen, bereitstellen, ausführen, aktualisieren und verwalten können. Abbildung 2.4 zeigt die Funktionsweise von Docker. Container sind eigenständige, ausführbare Einheiten die unabhängig vom OS deployed werden können. Container vereinfachen die Entwicklung und Bereitstellung von verteilten Anwendungen. Entwickler können Container auch ohne Docker erstellen doch Docker macht die Containerisierung schneller, einfacher und sicherer. Telepresence ist die neuste Erweiterung und bietet einen einfachen Weg mit Kubernetes zu entwickeln. (Ghosh (2020))

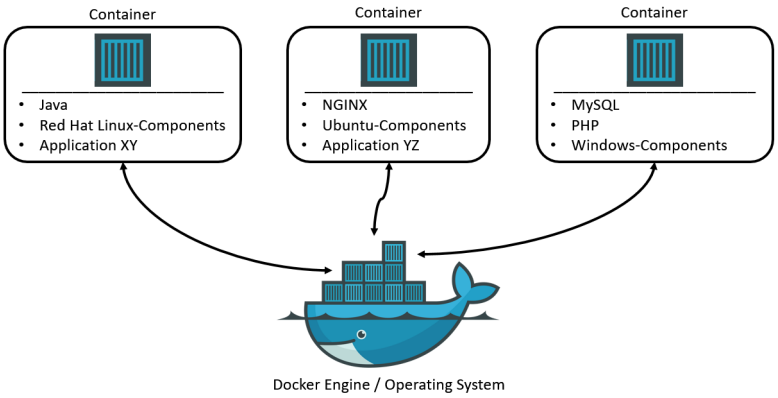

Abbildung 2.4: Funktionsweise von Docker (Quelle: Böllhoff (2022))

2.5.3 Microsoft .NET Core

Microsoft .NET Core ist ein Open-Source und plattformübergreifendes Framework, um Applikationen auf Android, Apple, Linux und Windows Betriebssystemen zu entwickeln. Microsoft .NET Core unterstützt mehrere Programmiersprachen wie C#, F# und Visual Basic. Zudem bietet .NET einen Paketmanager, um einfach und effizient Bibliotheken von Drittanbietern zu verwenden. Die Version .NET 8 bietet

einige Neuerungen wie beispielsweise Performance-focused Typen, die die Leistung einer Anwendung erheblich verbessern sollen. (BillWagner (2023))

2.5.4 Microsoft ASP.NET Core

Microsoft ASP.NET Core ist im Gegensatz zu .NET Core ein Framework um Webanwendungen zu entwickeln. ASP.NET stellt zusätzlich Frameworks zur Verfügung wie beispielsweise Blazor mit dem man einfach und effizient eine Interaktive Webbenutzeroberfläche mit C# entwickeln kann. ASP.NET unterstützt die Entwicklung und Implementierung von *Application Programming Interface* (API) und Microservices. Zudem ist ASP.NET laut *TechEmpower Web Framework Performance Comparison* (2023) schneller als jedes andere beliebte Web-Framework. (BillWagner (2023))

2.5.5 Entity Framework Core

Entity Framework (EF) Core ist eine *Object Relational Mapping* (ORM) Technik für Microsofts .NET Core Technologie. Die Technologie ist Open-Source, erweiterbar und plattformübergreifend. EF Core unterstützt LINQ-Abfragen, Änderungsverfolgung sowie Schemamigration. Weiteres unterstützt EF Core mehrere Datenbanken wie beispielsweise MySQL, PostgreSQL, Azure CosmosDB etc. und auch MariaDB. Die aktuelle Version von EF Core ist 7.0 und verfügt über neue Funktionen wie z.B. das Mapping und Abfragen auf *JavaScript Object Notation* (JSON) Spalten. Dabei ist es möglich einzelne Parameter von einem JSON Objekt abzufragen und zu ändern. Quellcode 1 zeigt wie das erfolgt. (BillWagner (2023))

```
1  var jeremy = await context
2  .Authors
3  .SingleAsync(author => author.Name.StartsWith("Jeremy"));
4
5  jeremy.Contact = new() {
6      Address = new("2 Riverside", "Trimbridge", "TB1 5ZS", "UK"),
7      Phone = "01632 88346"
8  };
9
10 await context.SaveChangesAsync();
```

Quellcode 1: Update von einem Parameter einer JSON Spalte

2.5.6 OpenAPI

Die OpenAPI-Initiative hat die *OpenAPI Spezifikation* (OAS) entwickelt, die eine API Beschreibung standardisiert. Die OAS ist eine Sprache für *Hypertext Transfer Protocol* (HTTP) APIs und bietet eine standardisierte Beschreibung für diese. Mithilfe von einem OpenAPI-Code-Generator (*OpenAPI Generator* (2023)) kann direkt Client-Code für verschiedene Technologien wie z.B. Typescript generiert werden. (*OpenAPI* (2023))

3 Anforderungen

Das Hauptziel dieser Arbeit ist die Entwicklung eines TCMS für GI. Dabei steht Zuverlässigkeit, Geschwindigkeit und Skalierbarkeit im Vordergrund. Zudem soll darauf geachtet werden, dass Ressourcen bzw. Entitäten, wie beispielsweise Testfälle, für mehrere Testpläne verwendet werden können, um damit mehrfache Einträge in der Datenbank zu verhindern.

Das zu entwickelnde TCMS hat folgende Anforderungen (Requirements):

- RE 1 - Produkt und Testsysteme beschreiben und darstellen
- RE 2 - Features beschreiben und darstellen
- RE 3 - Features zu einem Produkt zuordnen
- RE 4 - Testfälle beschreiben und darstellen
- RE 5 - Testfälle zu einem Feature zuordnen
- RE 6 - Testimplementierungen beschreiben und darstellen
- RE 7 - Testimplementierungen zu Testfällen und Testsysteme zuordnen
- RE 8 - Testläufe von Testimplementierungen zu einem Testsystem zuordnen
- RE 9 - Resultate von bestimmten Testläufen abzufragen

3.1 Minimum Viable Product

Bei einem *Minimum Viable Product* (MVP) handelt es sich um die erste funktionierte Iteration eines Produkts das nur Kernfunktionalitäten beinhaltet. In unserem Fall ist das ein funktionierendes TCMS, welches die oben angeführten Anforderungen abdeckt und von der Datenbankabfrage bis zur API unterstützt.

Der Vorteil ist, dass das Produkt so schnell wie möglich zum Kunden gelangt und Entwickler:innen Kundenfeedback bekommen. Die Kunden von dem zu entwickelnden

TCMS sind in diesem Fall die Mitarbeitenden der Firma Gantner Instruments. (Alliance (2017))

3.2 Testen der Funktionalitäten

Damit der Kunde die Funktionalitäten des TCMS auch *User Interface* (UI) unterstützt testen kann, wird ein webbasiertes UI von *Swagger* zur Verfügung gestellt, siehe Abbildung 3.1.

Swagger ist ein Werkzeug mit denen API-Beschreibungen erstellt werden können. Zudem kann Swagger mithilfe des standardisiertem OAS automatisch ein UI generieren. (*API Documentation & Design Tools for Teams | Swagger* (2023))

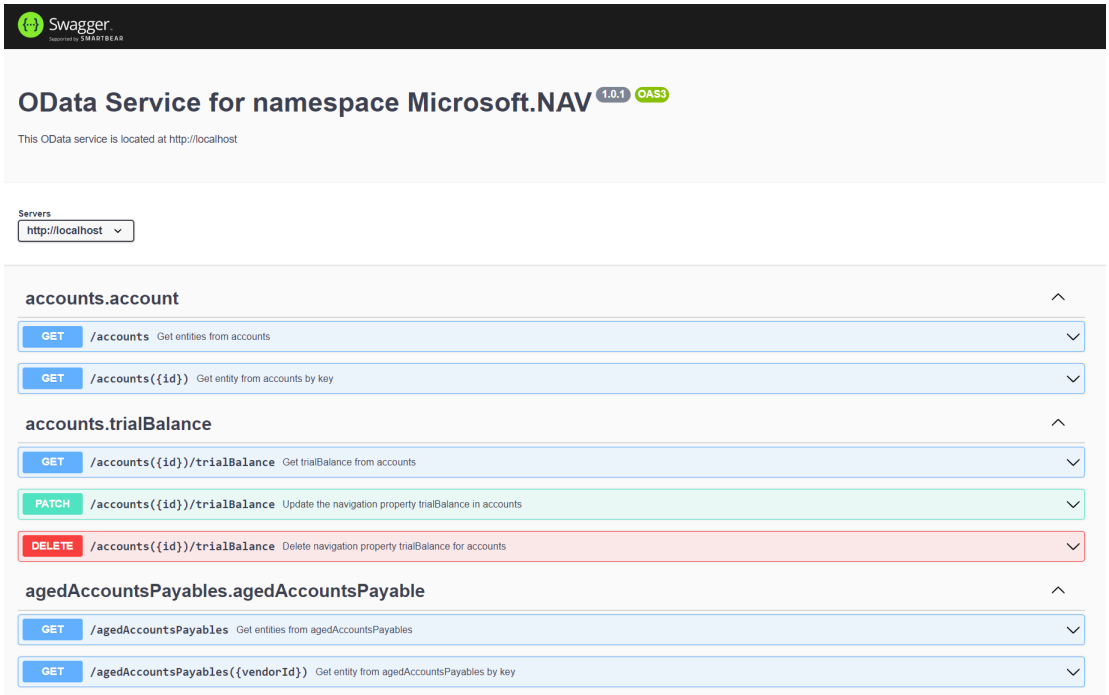

Abbildung 3.1: Beispiel UI von Swagger (Quelle: *Host the Swagger UI for your BC OpenAPI spec in your BC container* (2023))

3.3 Vorgehensweise

Die ermittelten Anforderungen wurden in User-Stories aufgeteilt. Nachfolgend sind vier User-Stories aufgeführt, die aufzeigen, welche Funktionalitäten das TCMS beinhaltet, um die oben gelisteten Anforderungen abzudecken.

3.3.1 Projekt und Datenbank Aufsetzen

Diese User-Story beinhaltet noch keine Implementierung von Funktionalitäten, sondern beschäftigt sich mit dem Aufsetzten des Projektes und der Datenbank.

Die Akzeptanzkriterien sind wie folgend:

- Projekt Struktur in die Architekturschichten aufteilen
- Datenbank mit Docker aufsetzen
- *Structured Query Language* (SQL) Statements möglich

Diese User-Story erfüllt nicht direkt eine oben angeführte Anforderung, bietet aber die Basis für die weiteren User-Stories.

3.3.2 Erstellen und Suchen von Test-Environments

Als Nutzer:in möchte ich Test-Environments erstellen und suchen können, sodass ich Testpläne zuordnen kann.

Die Akzeptanzkriterien sind wie folgend:

- Erstellen von Test-Environments über das Swagger-UI
- Suchen von Test-Environments über das Swagger-UI
- Zuordnungsmöglichkeit von Testplänen über das Swagger-UI

Diese User-Story erfüllt die Anforderungen RE 1 und RE 3.

3.3.3 Erstellen und Suchen von Testplänen, Testfällen und Testimplementationen

Als Nutzer:in möchte ich Testpläne, Testfälle und Testimplementationen erstellen und suchen können, sodass ich diese entsprechend zuordnen kann.

Die Akzeptanzkriterien sind:

- Erstellen von Testplänen, Testfällen und Testimplementationen über das Swagger-UI
- Suchen von Testplänen, Testfällen und Testimplementationen über das Swagger-UI
- Zuordnungsmöglichkeit von Testfällen und Testimplementationen über das Swagger-UI

Diese User-Story erfüllt die Anforderungen RE 2, 4, 5, 6 und RE 7.

3.3.4 Erstellen und Suchen von Testläufe

Als Nutzer:in möchte ich Testläufe erstellen und suchen können, sodass ich Reports generieren kann.

Die Akzeptanzkriterien sind wie folgend:

- Erstellen von Testläufen über das Swagger-UI
- Suchen von Testläufen über das Swagger-UI
- Zuordnungsmöglichkeit von Testläufen über das Swagger-UI

Diese User-Story erfüllt die Anforderungen RE 8 und RE 9.

4 Konzept

Im vorherigen Kapitel wurden die Anforderungen des zu entwickelnden TCMS festgelegt. Dieses Kapitel gibt einen Überblick über die konzipierte Architektur und gibt Einblick in die Planung des TCMS.

4.1 Architektur

Grundlegend repräsentiert eine Softwarearchitektur die Organisation und den Aufbau eines Systems, was die Architektur zu einer der wichtigsten Bestandteile in der Softwareentwicklung macht. Die Wahl der Softwarearchitektur eines Softwareprojektes hat demnach hohen Einfluss auf den späteren Verlauf des Projektes was auch bei einer Umstellung der Architektur sehr kostenintensive Auswirkung hat. Eine gut strukturierte Softwarearchitektur hat dementsprechend einige Vorteile (Richards und Ford (2020)):

- Übersichtlichkeit
- bessere Wartbarkeit
- Erweiterbarkeit
- Anpassungsfähigkeit
- Skalierbar
- reduziert Kosten und verhindert Code-Duplikation
- erhöht die Qualität der Software
- hilft bei komplexen Problemstellungen
- reduziert den Time-to-Market Faktor durch effizienteres entwickeln

4.1.1 Backend

In dieser Arbeit haben wir uns für eine Schichtenarchitektur mit *Domain Driven Design* (DDD) entschieden, da sich diese schon öfters bewährt hat und es gute Erfahrungen damit gab. Der Aufbau dieser Architektur unterteilt sich in vier Schichten, siehe Abbildung 4.1:

- Präsentationsschicht

  Diese repräsentiert APIs, die Anfragen entgegennimmt und dementsprechend Antworten liefert. Die Anfragen werden an die Applikationsschicht weitergeleitet.

- Applikationsschicht

  Diese Schicht interagiert mit der Präsentationsschicht und der Infrakstrukturschicht. Abhängig von den aus der Präsentationsschicht weitergeleiteten Anfragen werden Domain-Objects erzeugt und an die Infrakstrukturschicht weitergeleitet.

- Domainschicht

  Beinhaltet die Entitäten von der Domain und ist das Zentrum der Architektur. Diese Schicht wird von der Applikationsschicht und von der Infrastrukturschicht aufgerufen und beinhaltet die Business-Logik.

- Infrastrukturschicht

  Hier wird mit der Applikationsschicht, der Domainschicht und der Datenbank interagiert.

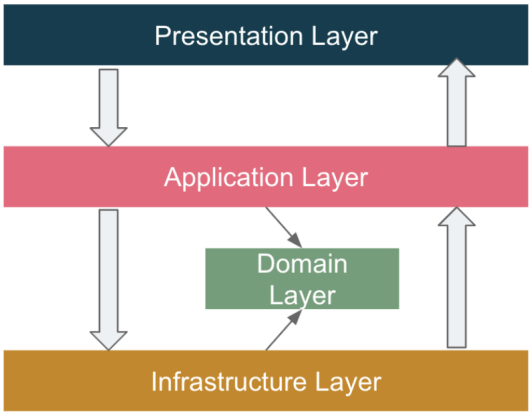

Abbildung 4.1: Übersicht einer Schichtenarchitektur (Quelle: Zhang (2023))

DDD ist ein Designansatz, bei dem die Entitäten des Systems, abhängig von dem Input der Domainexperten, modelliert werden. Das erleichtert es den Teammitgliedern die Arbeit der anderen besser zu verstehen. Diese Verwendung trägt auch zur *ubiquitous language* bei, die alle Teammitglieder bei Modell- und Entwurfsdiskussionen verwenden können. (Vernon (2013))

Abbildung 4.2 zeigt alle Komponenten, die bei DDD zum Einsatz kommen. In dieser Arbeit fokussieren wir uns speziell auf die folgenden Komponenten:

- Entities
- Value-Objects
- Aggregates
- Repositories
- Services

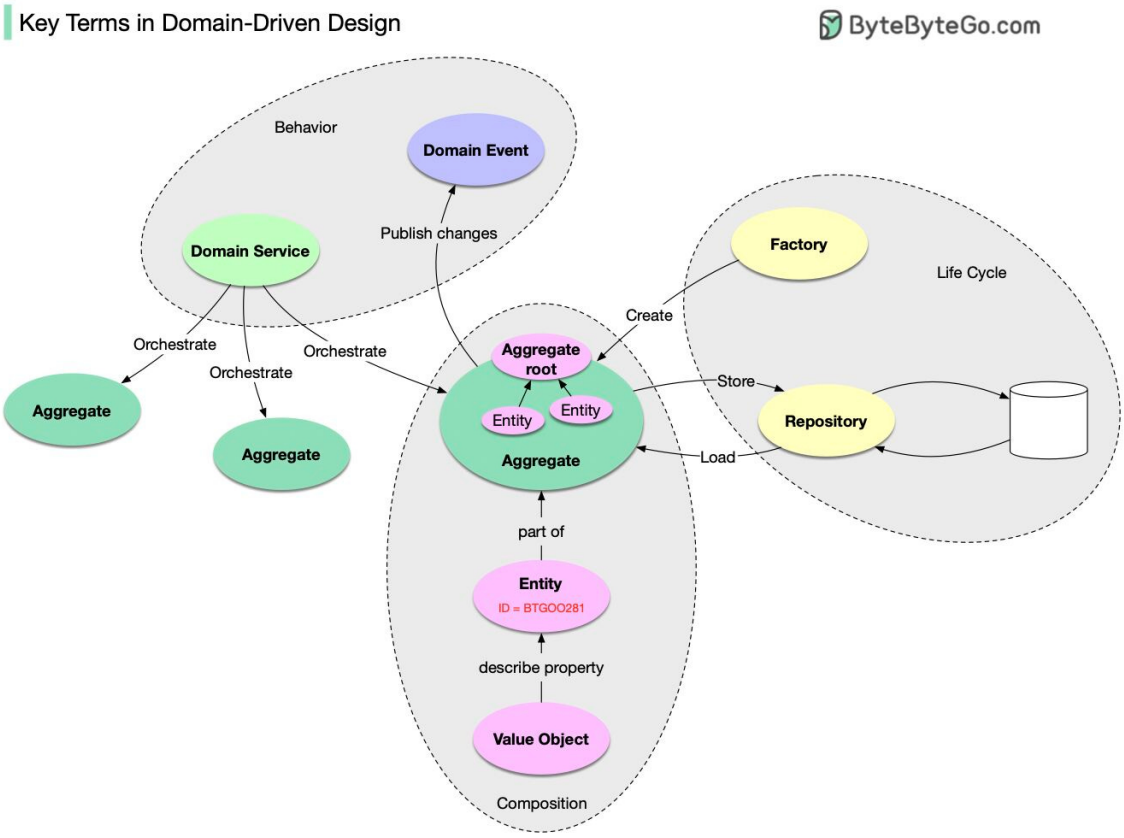

Abbildung 4.2: Komponenten von DDD (Quelle: Xu (2023))

4.2 Planung

In diesem Abschnitt geht es um die Modellierung des Domain-Models des TCMS. Vor der Modellierung wurde eine Übersicht von dem schon bestehenden *Test Automation System* (TAS) verschafft, dargestellt in Abbildung 4.3. Durch diese Übersicht wurden die Interaktionen mit dem zu entwickelnden TCMS ermittelt. Dabei greift das TAS über die Technologie *Automise* über eine von TCMS bereitgestellte API darauf zu und erstellt Testläufe. In Unterabschnitt 4.2.1 erfolgt eine detaillierte Beschreibung wie das TAS Testläufe erstellt.

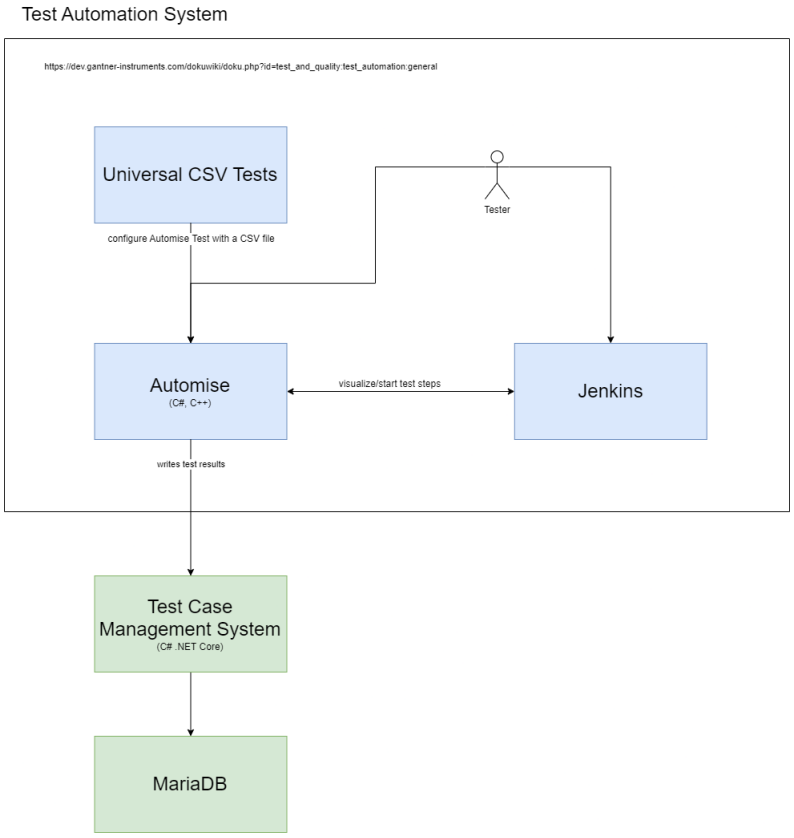

Abbildung 4.3: TAS von GI

4.2.1 Domain-Model

Zunächst wurde das Domain-Model entwickelt, um die Entitäten und deren Beziehungen darzustellen. Dabei wurde exakt darauf geachtet, dass die in Kapitel 3 ermittelten Anforderungen vollständig abgedeckt werden.

Die in Abbildung 4.4 abgebildeten Entitäten betreffen folgende Anforderungen:

- RE 1 - Produkt und Testsysteme beschreiben

  Das Produkt wurde als *Environment/Platform* (Test-Environment) modelliert. Ein Beispiel hierfür wäre ein Gerät der Firma. Das *Testsystem* ist ein Aufbau mit Geräten (Messsystemen), die geprüft werden. Beispielsweise ist das eine Prüfwand mit mehreren Geräten mit denen Tests durchgeführt werden. Jedes Produkt hat eigene Testsysteme.

- RE 2 - Features beschreiben und darstellen

  Feature wurde als *Testplan/Feature* modelliert. Ein Testplan besteht aus mehreren Testfällen (Entität Testcase) und bildet ein Feature von einem Gerät ab. Ein Beispiel für ein Feature ist die Netzwerk-Connectivity von einem Gerät.

- RE 3 - Features zu einem Produkt zuordnen

  Die erstellten Testpläne können zu Produkten zugeordnet werden. Somit können Testpläne für andere Produkte wiederverwendet werden.

- RE 4 - Testfälle beschreiben und darstellen

  Testfall wurde als *Testcase* modelliert. Ein Testfall besteht aus mehreren Testimplementierungen (Entität Testimplementation). Diese Testimplementierungen sind für jeden Testfall einzigartig und können nicht wiederverwendet werden. Ein Testfall ist beispielsweise eine Überprüfung, ob ein Gerät eine *Dynamic Host Configuration Protocol* (DHCP) Adresse erhalten hat.

- RE 5 - Testfälle zu einem Feature zuordnen

  Die Testfälle können zu Features zugeordnet werden. Somit können Testfälle für andere Features wiederverwendet werden.

- RE 6 - Testimplementierungen beschreiben und darstellen

  Eine Testimplementierung wurde als *Testimplementation* modelliert. Testimplementierungen sind Beschreibungen wie ein Testfall getestet wird und ob dieser manuell oder automatisch ausgeführt wird. Testimplementierungen enthalten auch Informationen, ob sie erfolgreich beim letzten Testlauf getestet worden sind. Eine Testimplementierungen ist die Beschreibung mit welchen Konfigurationen ein Testfall bei einem Gerät getestet wird.

- RE 7 - Testimplementierungen zu Testfällen und Testsystemen zuordnen

  Testimplementierungen könne zu Testfällen und zu Testsystemen zugeordnet werden. Durch die Zuordnung von Testimplementierungen zu Testsystemen ist es möglich die Testimplementierungen von einem Produkt darzustellen. Somit können beispielsweise Statistiken und Reports generiert werden, die einen Überblick über jede Testimplementierungen von einem Produkt geben.

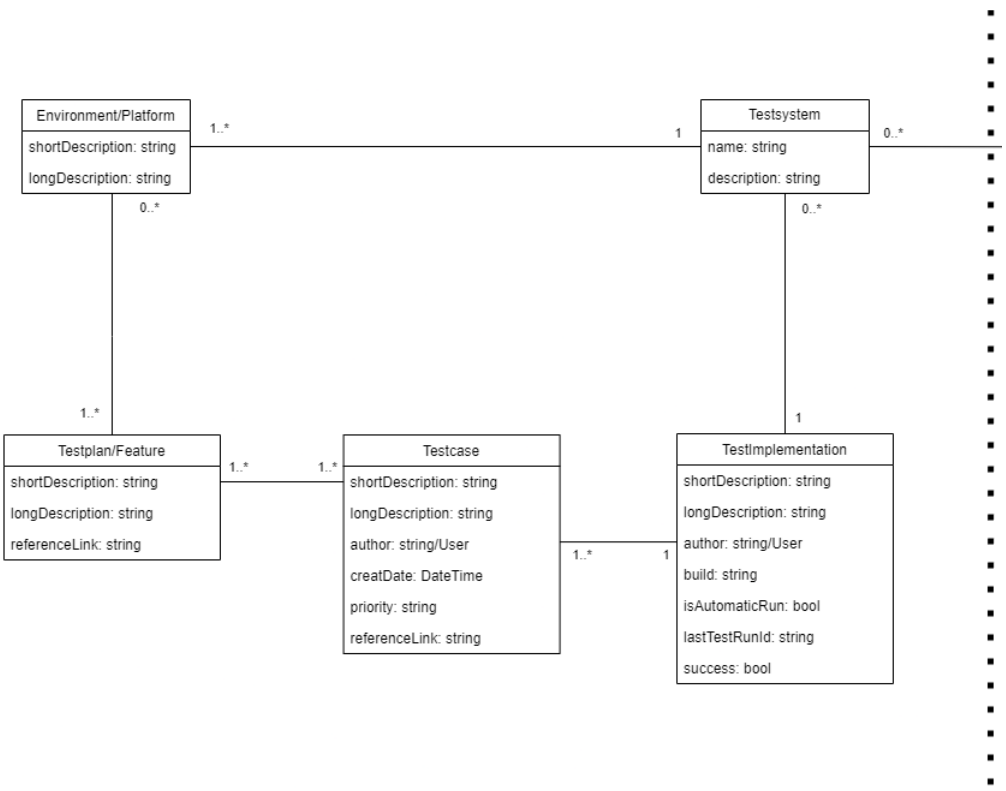

Abbildung 4.4: Domain-Model (Teil 1) basierend auf den Anforderungen von Kapitel 3

Die in Abbildung 4.5 abgebildeten Entitäten betreffen folgende Anforderungen:

- RE 8 - Testläufe von Testimplementierungen zu einem Testsystem zuordnen

  Ein Testlauf wurde als *TestRun* modelliert. Testläufe werden automatisch von dem bereits bestehenden TAS der Firma erstellt und beschrieben. Diese Testläufe enthalten Informationen, welche Testimplementationen mit welchen Konfigurationen und mit welcher Hardware getestet wurde. Testläufe werden zum Zeitpunkt des Erstellens zu den jeweiligen Testsystemen, mit denen diese durchgeführt worden sind, zugeordnet.

- RE 9 - Resultate von bestimmten Testläufen abzufragen

  Durch die Zuordnung von Testläufen zu Testsystemen können Resultate zu jedem Testsystem abgefragt werden.

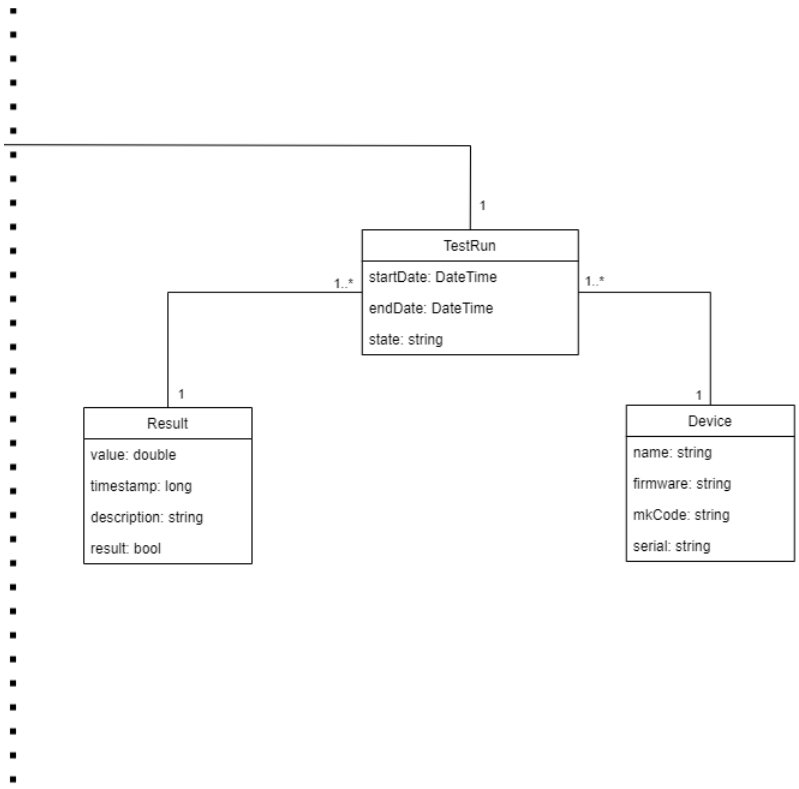

Abbildung 4.5: Domain-Model (Teil 2) basierend auf den Anforderungen von Kapitel 3

Die in Abbildung 4.4 und Abbildung 4.5 gezeigten Entitäten wurden auf Basis der Kommunikation mit Domainexperten modelliert.

5 Technische Umsetzung

Dieses Kapitel beschreibt die Implementierung der definierten User-Stories mit den gewählten Technologien und wie diese mit der entworfenen Architektur umgesetzt wurden. Zuerst wird der allgemeine Ablauf des verwendeten Scrum-Prozesses beschrieben und im Anschluss detailliert auf die User-Stories eingegangen.

5.1 Scrum

Scrum ist eine Projektmanagement-Methode zur Unterstützung der agilen Softwareentwicklung. Bei dieser Methode geht es darum, dass ein Team Teilaufgaben von einer komplexen Aufgabenstellung in kleinen Schritten, sogenannte *Sprints*, angeht. (Rubin (2012))

Für die Implementierung der User-Stories wurde der in Abbildung 5.1 gezeigte Prozess angewendet.

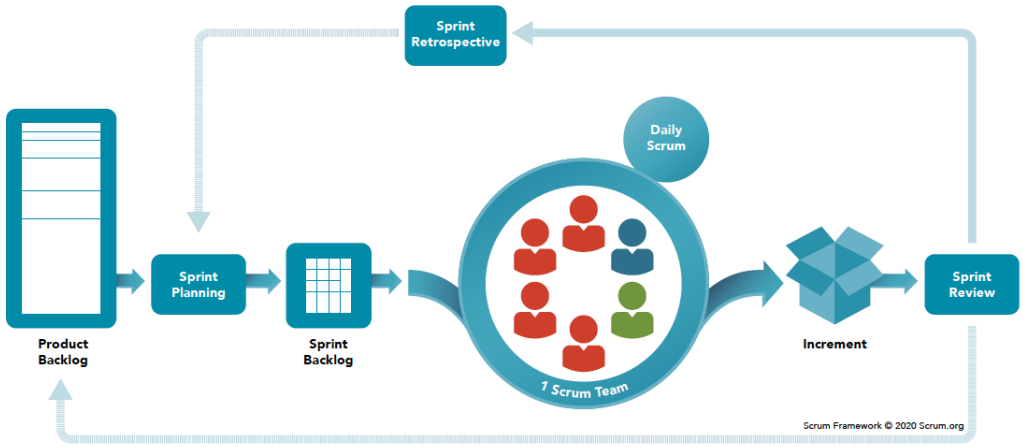

Abbildung 5.1: Scrum Prozess (Quelle: *The Home of Scrum* (2023))

5.2 Implementierung der User-Stories

Dieser Abschnitt zeigt den Ablauf der Implementierung der User-Stories. Die Implementierung ist auf vier User-Stories aufgeteilt, die in der folgenden Reihenfolge implementiert wurden:

- Projekt und Datenbank Aufsetzen
- Erstellen und Suchen von Test-Environments

  Um die Durchgängigkeit der Architektur zu veranschaulichen, wird bei dieser User-Story detailliert auf jede Schicht eingegangen. Eine detaillierte Beschreibung erfolgt in Unterabschnitt 5.2.2. Dabei wurden die User-Stories in folgender Reihenfolge in die Schichten implementiert:

  - Domainschicht
  - Infrastrukturschicht
  - Applikationsschicht
  - Präsentationsschicht

- Erstellen und Suchen von Testplänen, Testfällen und Testimplementationen

  In dieser User-Story gehen wir detailliert auf die Tabellen Konfigurationen mit EF Core ein. Eine detaillierte Beschreibung erfolgt in Unterabschnitt 5.2.3.

- Erstellen und Suchen von Testläufe

  In dieser User-Story wird gezeigt, wie das TAS von GI mit dem TCMS interagiert, siehe Unterabschnitt 5.2.4.

Für die Umsetzung des TCMS kommt die Version .NET 7.0 zum Einsatz. Als *Integrated Development Environment* (IDE) wurde *Rider* von *JetBrains* verwendet.

5.2.1 Projekt und Datenbank Aufsetzen

Damit eine Grundlage für die folgenden User-Stories vorhanden ist, wurde die Struktur des Projektes in vier Teilprojekte aufgeteilt. Dabei repräsentiert jedes Teilprojekt eine Schicht der verwendeten Architektur.

Für das Aufsetzen der Datenbank wurde das von Docker bereitgestellte Tool *Docker Desktop* verwendet. Docker Desktop ist eine Anwendung die das Erstellen und Verwalten von Container vereinfacht. (*Docker* (2022))

Quellcode 2 zeigt das Erstellen und Ausführen eines MariaDB Containers. Nachdem der Container hochgefahren ist, kann mit dem MariaDB Server gearbeitet werden.

```
1 docker run --name mariadb-container
2 -e MYSQL_ROOT_PASSWORD=admin -p 3306:3306 -d mariadb:latest
```

Quellcode 2: Starten eines MariaDB Containers

Quellcode 3 zeigt das Erstellen eines Datenbankschemas namens *tcms_ mariadb*. In dieser werden die Tabellen vom TCMS erzeugt und verwaltet.

```
1 CREATE DATABASE tcms_mariadb;
2 USE tcms_mariadb;
```

Quellcode 3: Erstellen und verwenden eines Datenbankschemas

Um eine Verbindung zur Datenbank herzustellen, liegt im Projekt ein JSON-File, das die benötigten Konfigurationen bereitstellt, siehe Quellcode 4. Diese werden beim Start des Projektes in der *Startup*-Klasse gelesen und verwendet, siehe Quellcode 5. Weiters zeigt Quellcode 5, dass die Funktion *AddDbContext* den Typ *EFContext* bekommt. Diese Klasse kümmert sich um die Konfiguration aller verwendeten Tabellen, die verwendet werden, siehe Quellcode 18.

Quellcode 15, 16 und 17 im Anhang zeigen die *Startup* und *Program*-Klasse, die für die Initiierung der Datenbank und Tabellen, der Dependency Injection (DI) Container und Generierung der Swagger-UI zuständig sind.

```
{
    [...]
    "ConnectionStrings": {
        "MariaDbConnectionString":
        "server=localhost;port=3306;database=tcms_mariadb;user=root;password=admin;"
    }
}
```

Quellcode 4: Ausschnitt der Konfigurationen für die Verbindung zur MariaDB Datenbank

```
[...]
services.AddDbContext<EFContext>(dbContextOptions => {
    var connectionString = _config
        .GetConnectionString("MariaDbConnectionString");

    dbContextOptions
        .UseMySql(connectionString,
            ServerVersion.AutoDetect(connectionString));
});
[...]
```

Quellcode 5: Ausschnitt der *Startup*-Klasse, der das Registrieren der Datenbank zeigt

5.2.2 Erstellen und Suchen von Test-Environments

Zuerst erfolgte die Implementierung der Domain-Models, *Test-Environment* und *Testsystem*, in der Domainschicht. Im Anschluss wurden die Interfaces für die Infrastrukturschicht definiert, die ebenfalls in der Domainschicht liegen. Somit kann die Infrastrukturschicht einfach und ohne Abhängigkeiten ausgetauscht werden.

Als nächster Schritt wurde das definierte Interface (Siehe: Quellcode 6) in der Infrastrukturschicht implementiert.

```
public interface ITestEnvironmentRepository
{
    string NextIdentity();
    Task<TestEnvironment> FindById(string id);
    Task<List<TestEnvironment>> FindByShortDescription(string shortDescription);
    Task<List<TestEnvironment>> GetAll();
    [...]
}
```

Quellcode 6: Test-Environment Aggregate Interface, für die Infrastrukturschicht

Mithilfe der ORM Technologie EF Core können Objekte ohne SQL Statements in die Datenbank geschrieben, gelesen, gelöscht und aktualisiert werden. Quellcode 7 zeigt wie ein Test-Environment mit einem *Identifier* (ID) von der Datenbank ausgelesen werden kann.

Damit wir das von Quellcode 7 ausgelesene Test-Environment-Objekt weiter verwenden können, wurde ein *TestEnvironmentService* in der Applikationsschicht implementiert. Dabei wird für den Service sowohl ein Interface definiert als auch eine implementierende Klasse *TestEnvironmentService* realisiert.

```
[...]
public async Task<TestEnvironment> FindById(string id)
{
    return await _context.TestEnvironments
        .Include(te => te.TestSystems)
        .Include(tp => tp.TestEnvironmentPlans)
        .FirstOrDefaultAsync(testEnvironment => testEnvironment.DomainId == id);
}
[...]
```

Quellcode 7: Ausschnitt der *TestEnvironmentRepository*-Klasse, die ein Test-Environment-Objekt mithilfe einer ID von der Datenbank ausliest

Der in Quellcode 8 gezeigte Code bekommt das Test-Environment-Objekt von der Infrastrukturschicht zurück und wandelt es in ein *Data Transfer Object* (DTO) um. Das DTO wird anschließend von der Applikationsschicht in die Präsentationsschicht weitergeleitet.

```
[...]
public async Task<TestEnvironmentDTO> FindById(string id)
{
    var testEnvironment = await _testEnvironmentRepository.FindById(id);
    return new TestEnvironmentDTO
    {
        Id = testEnvironment.DomainId,
        ShortDescription = testEnvironment.ShortDescription,
        LongDescription = testEnvironment.LongDescription,
        [...]
    };
}
[...]
```

Quellcode 8: Ausschnitt der *TestEnvironmentService*-Klasse

Der in Quellcode 9 gezeigte Code bekommt das umgewandelte Test-Environment-Objekt (DTO) von der Applikationsschicht zurück und schickt es dem Client weiter.

```
[...]
[HttpGet("{id}")]
public async Task<ActionResult<TestEnvironmentDTO>>
FindTestEnvironmentById(string id)
{
    var testEnvironmentDTO = await _testEnvironmentService.FindById(id);
    return Ok(testEnvironmentDTO);
}
[...]
```

Quellcode 9: Ausschnitt der *TestEnvironmentController*-Klasse

Der oben gezeigte Ablauf beschreibt die Suche nach einem Test-Environment. Beim Erstellen passiert ähnliches, nur dass der Client ein DTO über den bereitgestellten Endpunkt der API schickt. Die *TestEnvironmentController*-Klasse empfängt das DTO und schickt es der Applikationschicht weiter, diese Schicht wandelt es von einem DTO in ein Domain-Objekt um und schickt es weiter zur Infrastukturschicht, die es dann persistiert.

Der in Quellcode 10 gezeigte Code zeigt das Schreiben eines Test-Environment-Objekts in die Datenbank.

```
[...]
public async Task Add(TestEnvironment testEnvironment)
{
    await _context.TestEnvironments.AddAsync(testEnvironment);
    await _context.SaveChangesAsync();
}
[...]
```

Quellcode 10: Ausschnitt der *TestEnvironmentRepository*-Klasse

5.2.2.1 Swagger-UI

Dadurch, dass *Swagger* verwendet wird, wird beim Start des Projektes die *TestEnvironmentController*-Klasse gescannt und dafür eine Weboberfläche für die definierten Endpunkte generiert (Siehe: Abbildung 5.2).

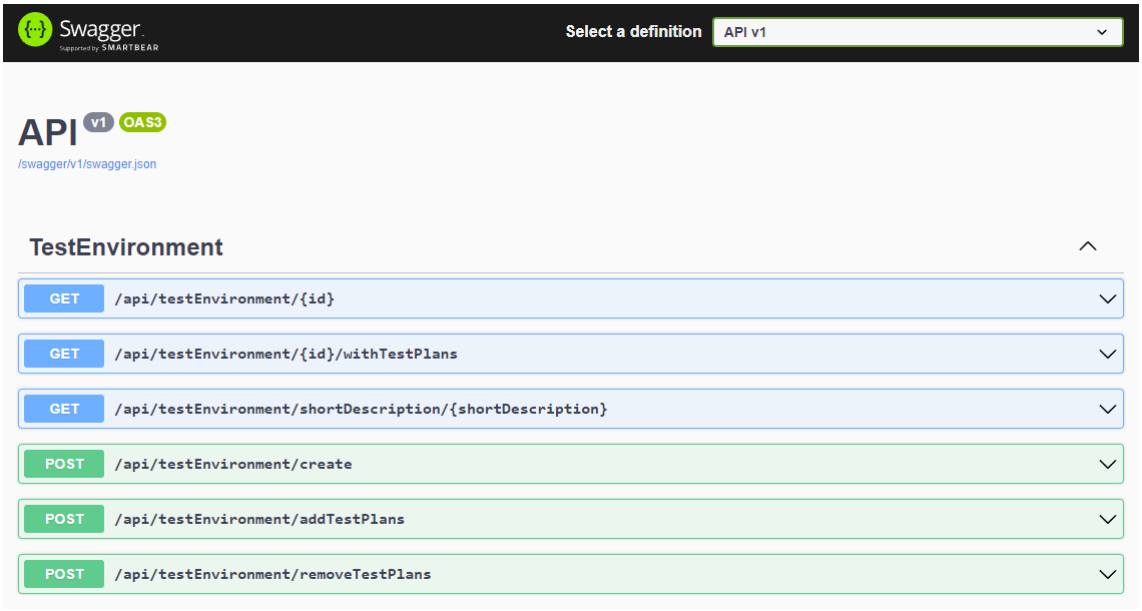

Abbildung 5.2: Test-Environment UI von Swagger

5.2.3 Erstellen und Suchen von Testplänen, Testfällen und Testimplementationen

Der Ablauf der Implementierung für die User-Story *Erstellen und Suchen von Testplänen, Testfällen und Testimplementationen* erfolgte analog zu dem oben beschriebenen Verfahren.

Zuerst wurden die Domain-Models erstellt, dann das Repository, gefolgt von dem Service und dem Controller. Zudem werden beim Start des Projektes die *Controller*-Klassen gescannt und eine Weboberfläche generiert. Somit ist das Erstellen und Suchen von Testplänen, Testfällen und Testimplementationen auf dem Swagger-UI verfügbar.

5.2.3.1 Tabellen Konfiguration mit EF Core

Damit beispielsweise die Testpläne einem Test-Environment zugewiesen werden können, muss eine *Junction* Tabelle erstellt werden, da zwischen Testplan und Test-Environment eine Many-to-many Beziehung herrscht.

Dadurch, dass eine *Junction* Tabelle benötigt wird, wurde die *TestEnvironmentPlan*-Klasse erstellt, die lediglich die Domain-IDs beider Domain-Klassen, *TestEnvironment* und *TestPlan*, beinhaltet (Siehe: Quellcode 11). Damit die zwei Domain-Klassen später in den EF Core Tabellenkonfigurationen als Many-to-many Beziehung konfigurierbar sind, ist die *TestEnvironmentPlan*-Klasse als *Property* in beiden Klassen vorhanden.

```
public class TestEnvironmentPlan
{
    public string TestEnvironmentDomainId {get; private set;}
    public string TestPlanDomainId {get; private set;}

    public TestEnvironmentPlan(
        string testEnvironmentDomainId,
        string testPlanDomainId
    ) {
        TestEnvironmentDomainId = testEnvironmentDomainId;
        TestPlanDomainId = testPlanDomainId;
    }
}
```

Quellcode 11: *TestEnvironmentPlan*-Klasse

Der in Quellcode 12 gezeigte Code konfiguriert einen Teil der Tabellen *TestEnvironment* und *TestPlan*. Dadurch wird festgelegt, dass die Tabellen *TestEnvironment* und *TestPlan* eine Many-to-one Beziehung zu der *Junction* Tabelle haben.

```
[...]

builder // TestEnvironment - many to one - TestEnvironmentPlan
    .HasMany(e => e.TestEnvironmentPlans)
    .WithOne()
    .HasPrincipalKey(e => e.DomainId)
    .IsRequired();

[...]

builder // TestPlan - many to one - TestEnvironmentPlan
    .HasMany(e => e.TestEnvironmentPlans)
    .WithOne()
    .HasPrincipalKey(e => e.DomainId)
    .IsRequired();

[...]
```

Quellcode 12: Ausschnitt aus der *TestEnvironmentConfiguration* und *TestPlanConfiguration*-Klasse

Der in Quellcode 13 gezeigte Code konfiguriert einen Teil der *Junction* Tabelle *TestEnvironmentPlan*. Dadurch wird festgelegt, dass die *TestEnvironmentPlan* Tabelle eine Many-to-one Beziehung zu den Tabellen *TestEnvironment* und *TestPlan* hat.

```
1  [...]
2
3  builder.HasKey(e => new {e.TestEnvironmentDomainId, e.TestPlanDomainId});
4
5  builder // TestEnvironmentPlan - many to one - TestEnvironment
6      .HasOne<TestEnvironment>()
7      .WithMany(e => e.TestEnvironmentPlans)
8      .HasPrincipalKey(e => e.DomainId)
9      .IsRequired();
10
11 builder // TestEnvironmentPlan - many to one - TestPlan
12     .HasOne<TestPlan>()
13     .WithMany(e => e.TestEnvironmentPlans)
14     .HasPrincipalKey(e => e.DomainId)
15     .IsRequired();
16
17 [...]
```

Quellcode 13: Ausschnitt aus der *TestEnvironmentPlanConfiguration*-Klasse

Dadurch ergibt sich eine Many-to-Many Beziehung und es wird ermöglicht, dass Testpläne zu einem Test-Environment zuweisbar sind.

Um ein Test-Environment und dazugehörige Testpläne abzufragen, wurde eine *TestEnvironmentManager*-Klasse in der Applikationsschicht implementiert. Diese Klasse hat Zugriff auf die *TestEnvironmentRepository*- und *TestPlanRepository*-Klassen. Quellcode 19 im Anhang zeigt, wie dies erfolgt.

5.2.4 Erstellen und Suchen von Testläufe

Der Ablauf der Implementierung erfolgte analog zu den vorherigen User-Stories. Die EF Core Konfiguration besteht aus folgenden Beziehungen:

- Many-to-one zu der *Testsystem* Domain-Klasse

  Dadurch wird ermöglicht, dass alle Testläufe eines Testsystems von einem Test-Environment abgefragt werden können.

- One-to-many zu der *ResultDetails* ValueObject-Klasse

- One-to-many zu der *DeviceDetail* ValueObject-Klasse

Quellcode 14 zeigt wie ein *Dictionary* mit EF Core konfiguriert wird. Dabei wird das *Dictionary* als JSON in eine JSON-Spalte geschrieben. Beim Auslesen wird das JSON-Objekt wieder in ein *Dictionary* deserialisiert.

```
1  [...]
2
3  builder
4      .Property(p => p.ResultDetailsMap)
5      .HasConversion<string>(
6          d => JsonConvert.SerializeObject(d),
7          s => JsonConvert.DeserializeObject<Dictionary<string, ResultDetails>>(s) ??
8              new Dictionary<string, ResultDetails>()
9  );
10
11 [...]
```

Quellcode 14: Ausschnitt aus der *TestRunConfiguration*-Klasse

Die Testimplementationen sind, wie die Testläufe, in einer Many-to-one Beziehung zu der *Testsystem*-Klasse. Dadurch ist es möglich, anhand des Status der Testimplementationen, Statistiken und Reports zu erstellen, die angeben in welchem Zustand ein Feature (Testplan) eines Gerätes (Test-Environment) ist und ob es Release bereit ist.

5.2.4.1 Automatische Erstellung von Testläufen über das TAS

Über bereitgestellte APIs ermöglicht das TCMS dem TAS von GI Testläufe automatisch zu erstellen. Ein Ablauf eines TAS sieht wie folgt aus:

- Ein Testlauf wird beim Start des TAS erstellt.

  Jeder Test des TAS beinhaltet eine ID von einer Testimplementation.

- Nach Beendigung eines Tests benachrichtigt das TAS, über eine bereitgestellte API, das TCMS

  Der Testlauf der beim Start erstellt wurde, wird nun aus der Datenbank ausgelesen. Die Resultate, die das TAS mit der zugehörigen Testimplementation ID sendet, wird in das *Dictionary*, welches von der *Testlauf*-Klasse, hinzugefügt. Der Testlauf wird anschließend in der Datenbank aktualisiert.

- Nach Beendigung des gesamten Testablaufs wird das Enddatum beim Testlauf eingetragen.

Testläufe können auch manuell über die Swagger-UI erstellt werden.

6 Evaluierung und Ausblick

Abschließend befasst sich dieses Kapitel mit den Verbesserungsmöglichkeiten der technischen Umsetzung des TCMS Backends. Weiters wird ein kurzer Ausblick in die Zukunft des TCMS gegeben.

6.1 Verbesserungsmöglichkeiten

In diesem Abschnitt werden potenzielle Verbesserungsmöglichkeiten für die umgesetzte Lösung erläutert. Obwohl die Umsetzung die Ziele dieser Arbeit erreicht hat, ist es wichtig mögliche Optimierungen und Verbesserungen der bestehenden Lösung zu veranschaulichen, da dadurch potenzielle Unschönheiten, Codeverdoppelungen etc. aufgedeckt werden können und sich dadurch der Wert des Projektes steigert. Folgend gibt es zwei Möglichkeiten Codeverdoppelungen und Unschönheiten zu vermindern.

6.1.1 Basis Klassen

Bei der Implementierung der User-Stories wurde ersichtlich, dass einige Funktionalitäten, wie z.B. Quellcode 6 zeigt, für mehrere Domain-Objekte wie beispielsweise *TestEnvironment*, *TestPlan* etc. gleich ist. Um das zu verhindern, könnten Basis-Klassen in der Infrastruktur- und Applikationsschicht verwendet werden. Beispielsweise könnten die Klassen *TestEnvironmentRepository* und *TestPlanRepository* Basisfunktionalitäten Generisch von einer Basis-Klasse vererben. Somit könnte ein großer Teil der Codebase verkleinert werden.

6.1.2 Automapper

Bei der Entwicklung der *Controller*-Klassen ist aufgefallen, dass die Umwandlung von DTOs in Domain-Objekte und umgekehrt oft mehrfach in verschiedene Funktionalitäten gleich funktioniert. Eine Third-Party Bibliothek names *AutoMapper* könnte die Umwandlungen um einiges vereinfachen. *AutoMapper* (2023)

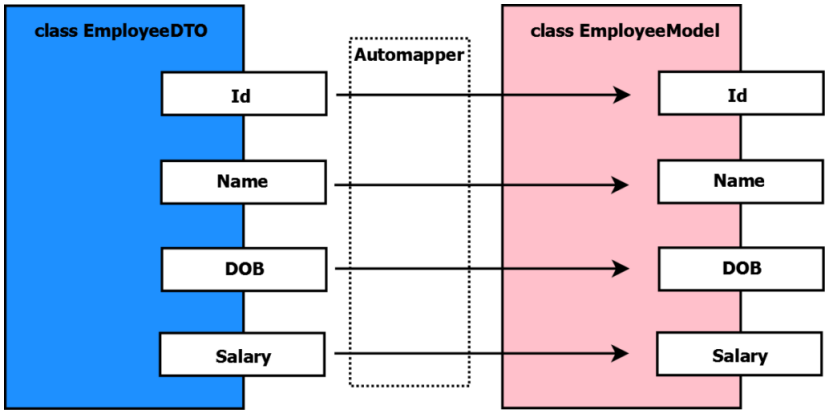

Abbildung 6.1: Funktionsweise von *AutoMapper* (Quelle: Sanjay (2020))

Abbildung 6.1 zeigt wie DTOs zu Domain-Objekte mit *AutoMapper* umgewandelt werden können. Somit wird die Übersichtlichkeit des Codes verbessert und eventuell Codeverdoppelung verhindert.

6.2 Ausblick

Die technische Umsetzung des TCMS Backends in Zusammenarbeit mit *Gantner Instruments* war der erste Schritt zur Realisierung eines Vollständigem *Test Case Management Systems*.

Aus der Sicht des Backends werden in Zukunft die oben aufgelisteten Verbesserungsmöglichkeiten implementiert und gegebenenfalls weitere Features eingebaut. Weiters wird ein Frontend mit den Technologien *React* und *TypeScript*, auf Basis dieser Arbeit, entwickelt.

Dadurch wird es die Möglichkeit geben über eine Weboberfläche Statistiken und Reports über gesamte Systeme und Geräte erstellen zu lassen, die wertvollen Einblick geben, welche Features getestet und Release bereit sind.

7 Zusammenfassung

Ziel dieser Arbeit war es ein funktionierendes TCMS Backend zu implementieren. Anhand der Kommunikation mit den Domainexperten sind die aufgelisteten Kriterien in Kapitel 3 auf Seite 22 entstanden. Durch diese Kriterien wurde dann entsprechend ein Domain-Model erstellt, das die angegebenen Kriterien vollständig abgedeckt hat. Folgend darauf wurden in Abschnitt 3.3 auf Seite 24 die User-Stories für die Umsetzung der gegebenen Anforderungen geplant.

Mit den beschriebenen Technologien in Abschnitt 2.5 auf Seite 18 erfolgte eine dementsprechende Umsetzung eines TCMS Backends. Angefangen mit der User-Story *Erstellen und Suchen von Test-Environments*, ist die vierschichtige Architektur implementiert worden. Darauffolgend konnten, mithilfe dieser Architektur, mit wenig Aufwand die restlichen User-Stories implementiert werden.

Betrachtet man die Ergebnisse der technischen Umsetzung[1] kann für die vierschichtigen Architektur und den angewendeten Technologien eine Empfehlung ausgesprochen werden. Entwickler:innen die Erfahrung mit den genannten Technologien und der Architektur haben können damit einfach und effizient ein eigenes TCMS Backend entwickeln.

Abschließend lässt sich sagen, dass die vorgenommenen Ziele erfolgreich umgesetzt worden sind und die anfangs gestellten relevanten Fragen, unterstützend mit der Umsetzung, beantwortet sind.

[1] https://github.com/iiNomad23/FHV_BachelorThesis

Literatur

7 Best Test Management Tools (2023). URL: https://www.practitest.com/test-management-tools/ (besucht am 18. 04. 2023).

Alliance, Agile (27. Juni 2017). *What is a Minimum Viable Product (MVP)?* Agile Alliance |. URL: https://www.agilealliance.org/glossary/mvp/ (besucht am 05. 05. 2023).

Ammann, Paul und Jeff Offutt (13. Dez. 2016). *Introduction to Software Testing*. Google-Books-ID: 58LeDQAAQBAJ. Cambridge University Press. 367 S. ISBN: 978-1-316-77312-3.

API Documentation & Design Tools for Teams | Swagger (2023). URL: https://swagger.io/ (besucht am 05. 05. 2023).

Atlassian (2023). *Die unterschiedlichen Arten von Softwaretests*. Atlassian. URL: https://www.atlassian.com/de/continuous-delivery/software-testing/types-of-software-testing (besucht am 15. 04. 2023).

AutoMapper (2023). URL: https://automapper.org/ (besucht am 17. 05. 2023).

BillWagner (2023). *.NET documentation*. URL: https://learn.microsoft.com/en-us/dotnet/ (besucht am 19. 04. 2023).

Böllhoff, Patrick (6. Jan. 2022). *Kubernetes vs Docker: eine Kooperation statt Konkurrenz*. Section: DevOps. URL: https://kruschecompany.com/de/kubernetes-vs-docker/ (besucht am 05. 05. 2023).

Docker (10. Mai 2022). *Docker: Accelerated, Containerized Application Development*. URL: https://www.docker.com/ (besucht am 01. 05. 2023).

Franklin, Benjamin (1748). *Advice to a young tradesman*. Bd. 1748. Philadelphia.

Ghosh, Saibal (3. Okt. 2020). *Docker Demystified: Learn How to Develop and Deploy Applications Using Docker*. Google-Books-ID: 650AEAAAQBAJ. BPB Publications. 243 S. ISBN: 978-93-89845-87-7.

Host the Swagger UI for your BC OpenAPI spec in your BC container (2023). URL: https://tobiasfenster.io/host-the-swagger-ui-for-your-bc-openapi-spec-in-your-bc-container (besucht am 05. 05. 2023).

Khandelwal, Abhik (12. Okt. 2019). *Difference between Black Box and White Box Testing | Testing Types*. TestingGenez. URL: https://testinggenez.com/black-box-and-white-box-testing/ (besucht am 01. 05. 2023).

Lead, The QA (2023). *Articles Archives*. The QA Lead. URL: https://theqalead.com/topics/ (besucht am 19. 04. 2023).

MariaDB documentation (2023). MariaDB.org. URL: https://mariadb.org/documentation/ (besucht am 19. 04. 2023).

Nidhra, Srinivas und Jagruthi Dondeti (30. Juni 2012). „BLACK BOX AND WHITE BOX TESTING TECHNIQUES -A LITERATURE REVIEW“. In: *International journal of embedded systems and applications*. DOI: 10.5121/ijesa.2012.2204. URL: https://www.scinapse.io/papers/2334860424 (besucht am 01. 05. 2023).

OpenAPI (2023). OpenAPI Initiative. URL: https://www.openapis.org/ (besucht am 19. 04. 2023).

OpenAPI Generator (2023). URL: https://openapi-generator.tech/ (besucht am 19. 04. 2023).

PractiTest - Test Management Platform to Manage all Your QA Efforts (2023). URL: https://www.practitest.com/ (besucht am 18. 05. 2023).

Richards, Mark und Neal Ford (28. Jan. 2020). *Fundamentals of Software Architecture: An Engineering Approach*. Google-Books-ID: xa7MDwAAQBAJ. Ö'Reilly Media, Inc." 422 S. ISBN: 978-1-4920-4342-3.

Rubin, Kenneth S. (2012). *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. Google-Books-ID: HkXX65VCZU4C. Addison-Wesley Professional. 501 S. ISBN: 978-0-13-704329-3.

Sanjay (7. Juni 2020). *Implement Automapper in ASP.NET Core 3.1 - Quick & Easy Guide | Pro Code Guide*. Running Time: 325 Section: .NET Core. URL: https://procodeguide.com/programming/automapper-in-aspnet-core/ (besucht am 17. 05. 2023).

Software Testing/Qualitätssicherung - Alle Methoden und Tools (2023). Software Testing/Qualitätssicherung - Alle Methoden und Tools. URL: https://q-centric.com/ (besucht am 15. 04. 2023).

TechEmpower Web Framework Performance Comparison (2023). www.techempower.com. URL: https://www.techempower.com/benchmarks/#section=data-r21&hw=ph&test=plaintext (besucht am 01. 05. 2023).

Test-driven development - IBM Garage Practices (2023). URL: https://www.ibm.com/garage/method/practices/code/practice_test_driven_development/ (besucht am 01. 05. 2023).

TestRail – Orchestrate Testing. Elevate Quality. (25. Jan. 2023). URL: https://www.testrail.com/ (besucht am 17. 04. 2023).

The Home of Scrum (2023). Scrum.org. URL: https://www.scrum.org/index (besucht am 12. 05. 2023).

Tricentis qTest for Unified Test Management (2023). Tricentis. URL: https://www.tricentis.com/products/unified-test-management-qtest/ (besucht am 18. 04. 2023).

Vernon, Vaughn (6. Feb. 2013). *Implementing Domain-Driven Design*. Google-Books-ID: X7DpD5g3VP8C. Addison-Wesley. 656 S. ISBN: 978-0-13-303988-7.

Westland, J. Christopher (1. Mai 2002). „The cost of errors in software development: evidence from industry". In: *Journal of Systems and Software* 62.1, S. 1–9. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(01)00130-3. URL: https://doi.org/10.1016/S0164-1212(01)00130-3 (besucht am 08. 05. 2023).

Xu, Alex (19. Apr. 2023). *EP32: REST vs. GraphQL*. URL: https://blog.bytebytego.com/p/ep32-how-does-grpc-work (besucht am 18. 05. 2023).

Zephyr Test Management Products | SmartBear (2023). URL: https://smartbear.com/test-management/zephyr/ (besucht am 17. 04. 2023).

Zhang, Yang (2023). *Domain Driven Design implemented by functional programming*. Thoughtworks. URL: https://www.thoughtworks.com/insights/blog/microservices/ddd-implemented-fp (besucht am 06. 05. 2023).

Anhang

```
1  public class Startup
2  {
3      private readonly IConfiguration _config;
4
5      public Startup(IConfiguration config) { _config = config; }
6
7      // This method gets called by the runtime.
8      // Use this method to add services to the container.
9      public void ConfigureServices(IServiceCollection services)
10     {
11         // Register database with the DI container
12         services.AddDbContext<EFContext>(dbContextOptions => {
13             var connectionString = _config
14                 .GetConnectionString("MariaDbConnectionString");
15
16             dbContextOptions
17                 .UseMySql(connectionString,
18                     ServerVersion.AutoDetect(connectionString));
19         });
20
21         // Register repositories with the DI container
22         services.AddTransient<ITestCaseRepository, TestCaseRepository>();
23         services.AddTransient<ITestPlanRepository, TestPlanRepository>();
24         [...]
25
26         // Register services with the DI container
27         services.AddTransient<ITestCaseService, TestCaseService>();
28         services.AddTransient<ITestPlanService, TestPlanService>();
29         [...]
30
31         services.AddControllers();
32         services.AddSwaggerGen(c => {
33                 c.SwaggerDoc("v1",
34                     new OpenApiInfo { Title = "API", Version = "v1" });
35         });
36     }
37
38 [...]
```

Quellcode 15: *Startup*-Klasse Teil 1

```
[...]

    // This method gets called by the runtime.
    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseSwagger();
            app.UseSwaggerUI(c =>
                c.SwaggerEndpoint(
                    "/swagger/v1/swagger.json",
                    "API v1"
                )
            );
        }

        app.UseHttpsRedirection();
        app.UseRouting();
        app.UseAuthorization();
        app.UseEndpoints(endpoints => { endpoints.MapControllers(); });
    }
}
```

Quellcode 16: *Startup*-Klasse Teil 2

```
public class Program
{
    public static void Main(string[] args)
    {
        // Use this line of code when we are going production
        // CreateHostBuilder(args).Build().Run();

        // Remove the code block below when we are going production
        var host = CreateHostBuilder(args).Build();
        using (var scope = host.Services.CreateScope())
        {
            var services = scope.ServiceProvider;
            var dbContext = services.GetRequiredService<EFContext>();
            dbContext.Database.EnsureDeleted();
            dbContext.Database.EnsureCreated();
        }

        host.Run();
    }

    private static IHostBuilder CreateHostBuilder(string[] args)
    {
        return Host
            .CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder => {
                webBuilder.UseStartup<Startup>();
            });
    }
}
```

Quellcode 17: *Program*-Klasse, die die *Startup*-Klasse verwendet

```
public class EFContext : DbContext
{
    public DbSet<TestCase> TestCases { get; set; }
    public DbSet<TestPlan> TestPlans { get; set; }
    public DbSet<TestImplementation> TestImplementations { get; set; }
    public DbSet<TestRun> TestRuns { get; set; }
    public DbSet<TestEnvironment> TestEnvironments { get; set; }
    public DbSet<TestSystem> TestSystems { get; set; }
    [...]

    public EFContext(DbContextOptions<EFContext> options) : base(options) { }
    public EFContext() { }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        ApplyChildConfigurations(modelBuilder);

        modelBuilder.ApplyConfiguration(new TestCaseConfiguration());
        modelBuilder.ApplyConfiguration(new TestPlanConfiguration());
        modelBuilder.ApplyConfiguration(new TestImplementationConfiguration());
        modelBuilder.ApplyConfiguration(new TestRunConfiguration());
        modelBuilder.ApplyConfiguration(new TestEnvironmentConfiguration());
        [...]
    }

    private static void ApplyChildConfigurations(ModelBuilder modelBuilder)
    {
        modelBuilder.ApplyConfiguration(new TestSystemConfiguration());
        modelBuilder.ApplyConfiguration(new TestEnvironmentPlanConfiguration());
        [...]
    }
}
```

Quellcode 18: *EFContext*-Klasse, die EF Core verwendet, um die Tabellen zu erstellen. Diese Klasse wird, bei der Registrierung der Datenbank, in Quellcode 15 als Typ angegeben.

```
[...]

public async Task<TestEnvironmentWithTestPlansDTO> FindByIdWithTestPlans(string id)
{
    var testEnvironment = await _testEnvironmentRepository.FindById(id);

    var testEnvironmentPlanIds = testEnvironment.TestEnvironmentPlans
        .Select(item => item.TestPlanDomainId)
        .ToArray();

    var testPlans = await _testPlanRepository.FindByIdSet(testEnvironmentPlanIds);

    var testPlanDTOs = testPlans
        .Select(tp => new TestPlanDTO
            {
                Id = tp.DomainId,
                ShortDescription = tp.ShortDescription,
                LongDescription = tp.LongDescription,
                ReferenceLink = tp.ReferenceLink
            }
        )
        .ToList();

    var testEnvironmentDTO = new TestEnvironmentDTO
    {
        Id = testEnvironment.DomainId,
        ShortDescription = testEnvironment.ShortDescription,
        LongDescription = testEnvironment.LongDescription,
        TestSystems = testEnvironment.TestSystems
            .Select(ts => new TestSystemDTO
                {
                    Name = ts.Name,
                    Description = ts.Description,
                }
            )
            .ToList()
    };

    var testEnvironmentWithTestPlansDTO = new TestEnvironmentWithTestPlansDTO
    {
        TestEnvironmentDTO = testEnvironmentDTO,
        TestPlanDTOs = testPlanDTOs
    };

    return testEnvironmentWithTestPlansDTO;
}

[...]
```

Quellcode 19: Ausschnitt der *TestEnvironmentManager*-Klasse die das Interface *ITestEnvironmentManger* implementiert

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dornbirn, am 20. Mai 2023 Marco Prescher