FH Vorarlberg
University of Applied Sciences

Entwicklung eines Test Case Management Systems für GI-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 1. Juli 2023

Kurzreferat

Integration eines Feature-orientierten Testsystems in den Entwicklungszyklus technischer Systeme

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 Zeitdruck und Deadlines oftmals vernachlässigt, oder nur unzureichend durchgeführt. Mit einem gut strukturierten Testplan kann dieses Risiko allerdings minimiert werden, und die Durchführung der Tests mit dem Produktentwicklungszyklus integriert werden. Dadurch kann stabile Hardware sowie effiziente und gut strukturierte Software ohne große Verzögerung an den Kunden ausgeliefert werden.

Durch Methodiken wie zum Beispiel Feature-orientierte Entwicklung ist es möglich, dass bestimmte Features vor dem Release der Produkte abgenommen und getestet werden müssen. Das wiederum ermöglicht, dass neu implementierte Features gründlich getestet werden und somit zu einer hohen Qualität beitragen.

Dies kann erreicht werden, indem Features strukturiert in einer Datenbank eingepflegt werden. Ein auf diesen Featurebeschreibungen basierendes System kann die automatische Testdurchführung unterstützen, gezielt Tests für einzelne Features durchführen und deren Abnahme beschleunigen. Dadurch kann der Testaufwand verringert und den Entwicklern ein fokussiertes Feedback vermittelt werden.

Abstract

Integrating a feature-oriented testing system into the development cycle of technical systems

The development of technical systems is a complex and costly process. Therefore, it is important that products are thoroughly tested before delivery to the customer. However, this is often neglected or done insufficiently due to time pressure and deadlines. A well-structured test plan can minimize this risk and integrate the testing process into the product development cycle. This allows stable hardware and efficient, well-structured software to be delivered to the customer without significant delays.

Using techniques such as feature-oriented development, certain features must be accepted and tested before the product release. This in turn allows newly implemented features to be thoroughly tested and contribute to high quality.

This can be achieved by structuring features in a database. A system based on these feature descriptions can support automatic testing, conduct targeted tests for individual features, and accelerate their acceptance. This reduces testing effort and provides focused feedback to the developers.

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

Danksagung

Ich möchte mich aufrichtig bei Ralph Hoch, der Firma Gantner Instruments und allen Mitarbeitenden bedanken, die mir bei der Vollendung dieser Arbeit geholfen haben. Ihre Unterstützung und Expertise waren von unschätzbarem Wert und haben zum erfolgreichen Abschluss dieses Projekts beigetragen. Vielen Dank für Ihre harte Arbeit und Ihr Engagement. Ich schätze Ihre Zusammenarbeit sehr.

1 Einleitung

"Time is money."
— Franklin (1748)

In der Softwareentwicklung kann es durch Fehler und Mängel zu Verzögerungen und dadurch zu Verlusten kommen. Deswegen wurde in Zusammenarbeit mit der Firma *Gantner Instruments* (GI) geplant ein *Test Case Management System* (TCMS) zu entwickeln. Diese Arbeit fokussiert sich daher auf die Implementierung von einem TCMS und vergleicht verschiedene vorhandene Lösungen.

1.1 Motivation

Die Entwicklung technischer Systeme ist ein komplexer Prozess, der eine hohe Qualität erfordert, um den Anforderungen der Kunden gerecht zu werden. Damit diese Qualität auch gewährleistet wird, 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 von Abschnitt 1.1 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 somit auch zu verfolgen. Obwohl TCMS in der Industrie weit verbreitet sind und es einige fertige Lösungen gibt, gibt es jedoch nicht immer die perfekte Lösung um ein bestehendes TCMS für das eigene Projekt anzuwenden. Insbesondere gibt es Bedenken hinsichtlich der Anwendbarkeit und Integration von einem schon bestehendem TCMS, da dieses System auch mit internen Tools kommunizieren können muss. Diese Probleme stellen Hindernisse dar, die die Einführung von TCMS in einem Unternehmen erschweren. Diese Arbeit beschäftigt sich mit diesen Problemen und untersucht, wie ein TCMS effektiv eingesetzt werden kann.

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es, die Verwendung von TCMS zu untersuchen und zu bewerten. Zudem ein auf unser eigenes Produkt angepasstes TCMS zu entwickeln. Hiermit ergeben sich drei relevante fragen:

- Wie kann man Features, Testfälle und Testimplementierungen beschreiben?
- Wie in Datenbank schreiben und lesen?
- Wie mit Testläufen verknüpfen?

Insbesondere möchten wir die folgenden Ziele erreichen:

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

Durch die Erfüllung dieser Ziele wird diese Arbeit dazu beitragen, das Verständnis für die Verwendung von einem 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 je nachdem andere Aspekte eines Produkts testen. In *Software Testing/Qualitätssicherung - Alle Methoden und Tools* (2023) wird beschreiben, dass dabei zwischen *Funktionalen* und *Nicht-Funktionalen* Tests unterschieden werden. Zu der funktionalen Testfamilie zählen beispielsweise Unit-Tests und Integrationstests. Nicht funktionale Testarten sind beispielsweise Leistung, Last und Stresstests sowie Usability-Tests zu den nicht funktionalen Testfamilie gehören.

Ein paar der am häufigsten vorkommenden Testarten sind:

- Unit-Tests
- Integrationstests
- Funktionstests
- Leistungstests

Um eine Testabdeckung sicherzustellen und den allgemeinen Testprozess zu optimieren werden TCMS eingesetzt. Diese Systeme bieten eine zentrale Plattform zur Verwaltung von Testfällen, wo Tests, die für ein Release oder Produkt erforderlich sind, erstellt und dokumentiert werden können (Siehe Abschnitt 2.4).

Ammann und Offutt (2016)

2.1.1 Unit-Tests

Im Allgemeinen sind Unit-Tests Tests die beispielsweise Methoden einer Klasse mit unterschiedlichen Parametern testet. Sie sind automatisierbar und können von einer Continuos-Integration-Pipeline durchgeführt werden. Diese Tests ermöglichen eine kontinuierliche Überprüfung der Funktionsfähigkeit des Produkts 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 integriert, um ausschließlich Ergebnisse einer gegebenen Funktion zu überprüfen. Dabei 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 zu überprüfen. 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 als auch die Qualität des Produkts zu evaluieren. Dabei gibt es zwischen den beiden wichtige Unterschiede.

Verbessering der Erklärung von white box testing

Bei White-Box Testing handelt es sich grundsätzlich um Unit-Tests die den Code und die Struktur 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)

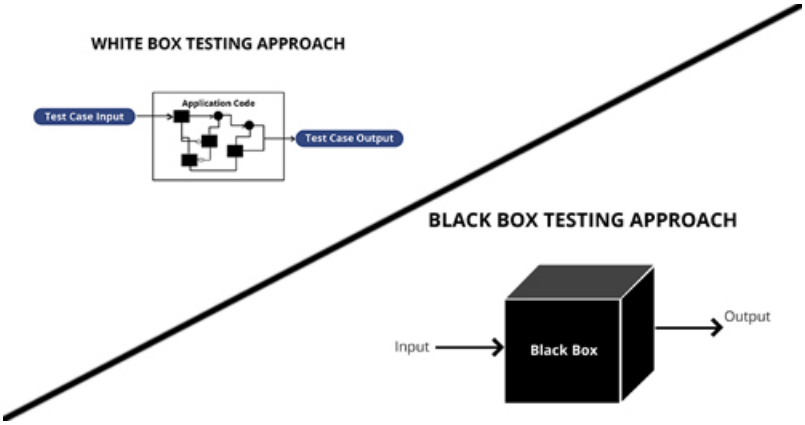

Abbildung 2.1: Black-Box/White-Box Testing (Quelle: Khandelwal (2019))

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 die genau soviel beinhaltet, dass der Test erfolgreich durchgeführt werden kann. TDD bietet daher mehrere Vorteile:

- 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.

Ammann und Offutt (2016)

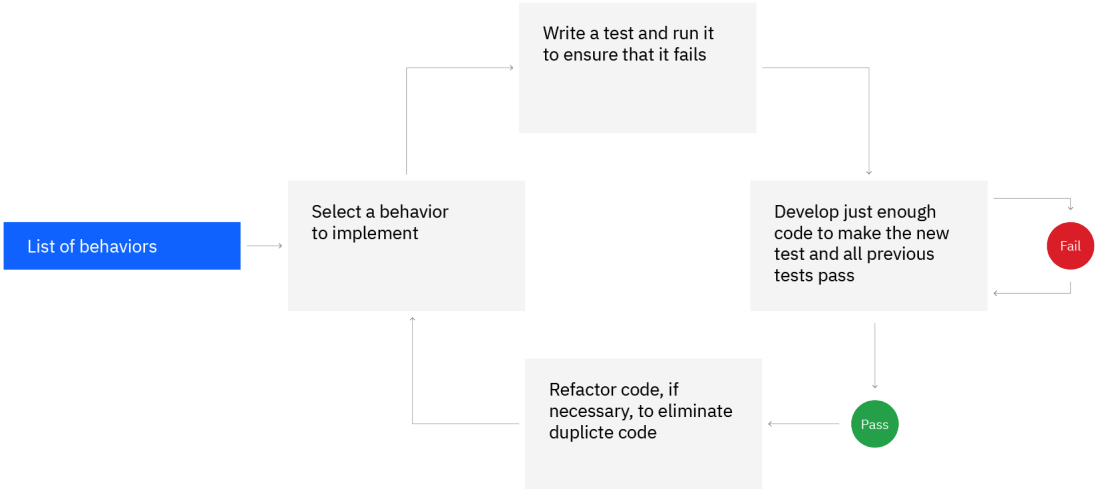

Abbildung 2.2: Test Driven Development cycle (Quelle: *Test-driven development - IBM Garage Practices* (2023))

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:

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

Lead (2023)

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 die Testautomatisierung nutzen, um einen reibungslosen Ablauf zu gewährleisten.

Einige Vorteile von TCMS bezüglich der Testausführungsläufe sind:

- 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 und 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.

Lead (2023)

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 vereinfacht.

Einige bekannte TCMS sind:

- *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

- *Tricentis Test Management for Jira* (2023)

  Webbasiertes TCMS und hat folgende Features:

  - Test Planung, Verwaltung und Ausführung
  - Echtzeit Test Status Update
  - Berichterstattung

- *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 natürlich von der jeweiligen Anwendung 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. 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)

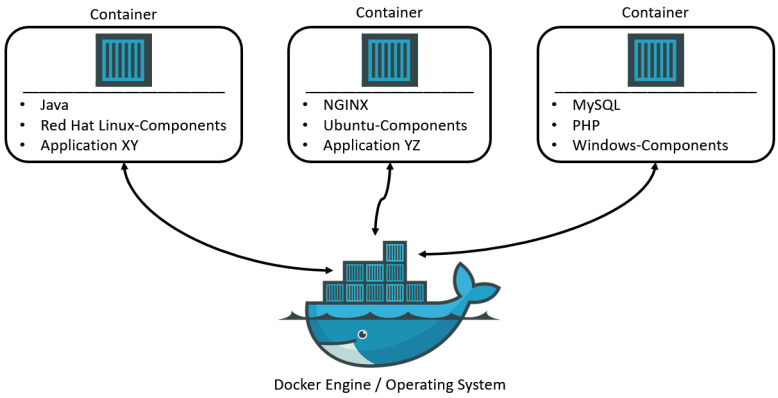

Abbildung 2.4: Funktionsweise von Docker (Quelle: Böllhoff (2022))

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 neue Version .NET 8 bietet einige Neuerungen wie beispielsweise Performance-focused Typen, die die Leistung einer Anwendung erheblich verbessern soll. .NET 8 wird offiziell im November 2023 veröffentlicht.

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 verfügt über Vorteile wie beispielsweise Blazor mit dem man einfach und effizient eine Interaktive Webbenutzeroberfläche mit C# entwickeln kann. Dieses Framework unterstützt die Entwicklung und Implementierung von *Application Programming Interface* (API) und Mikroservices. 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 (Unterabschnitt 2.5.3). Die Technologie ist Open-Source, erweiterbar und plattformübergreifend. EF Core unterstützt LINQ-Abfragen, Änderungsverfolgung, Aktualisierung 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.

```
var jeremy = await context
.Authors
.SingleAsync(author => author.Name.StartsWith("Jeremy"));

jeremy.Contact = new() {
    Address = new("2 Riverside", "Trimbridge", "TB1 5ZS", "UK"),
    Phone = "01632 88346"
};

await context.SaveChangesAsync();
```

Quellcode 1: Update von einem Parameter einer JSON Spalte

BillWagner (2023)

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, durch diese APIs standardisiert beschrieben werden können. 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

Der Hauptteil dieser Arbeit ist es ein funktionierendes TCMS mit den genannten Technologien zu entwickeln. Dabei steht Zuverlässigkeit, Geschwindigkeit und Skalierbarkeit im Vordergrund. Zudem soll geachtet werden, dass Ressourcen bzw. Entitäten wie beispielsweise Testfälle für mehrere Testpläne verwendet werden können damit mehrfache Einträge in der Datenbank verhindert werden.

Das zu entwickelnde TCMS hat folgende Anforderungen:

- Produkt und Testsysteme beschreiben
- Features beschreiben
- Features zu einem Produkt zuordnen
- Testfälle beschreiben
- Testfälle zu einem Feature zuordnen
- Testimplementierungen beschreiben
- Testimplementierungen zu Testfällen und Testsysteme zuordnen
- Testläufe von Testimplementierungen zu einem Testsystem zuordnen
- Resultate von bestimmten Testläufen abzufragen

3.1 Minimum Viable Product

Ein *Minimum Viable Product* (MVP) ist ein minimal brauchbares Produkt. Dabei handelt es sich um die erste funktionierte Iteration eines Produkts das nur Kernfunktionalitäten beinhaltet. In unserem Fall ist das ein Funktionierendes TCMS backend von der Datenbank abfrage bis hin zur API und sollte die oben angeführten Anforderung abdecken.

Der Vorteil ist, dass das Produkt so schnell wie möglich zum Kunden gelangt und das Produkt Kundenfeedback bekommt. Die Kunden von dem zu entwickelnden TCMS sind in diesem Fall die Mitarbeitenden der Firma Ganter Instruments.

Alliance (2017)

3.2 Testen der Funktionalitäten

Damit der Kunde die Funktionalitäten des TCMS auch oberflächlich Testen kann, wird ein Tool namens Swagger verwendet um ein Webbasiertes *User Interface* (UI) bereitzustellen. Swagger ist grundsätzlich ein Werkzeug mit denen Teams API-Beschreibungen erstellen können. Zudem kann Swagger mithilfe des standardisiertem OAS automatisch ein UI generieren.

API Documentation & Design Tools for Teams | Swagger (2023)

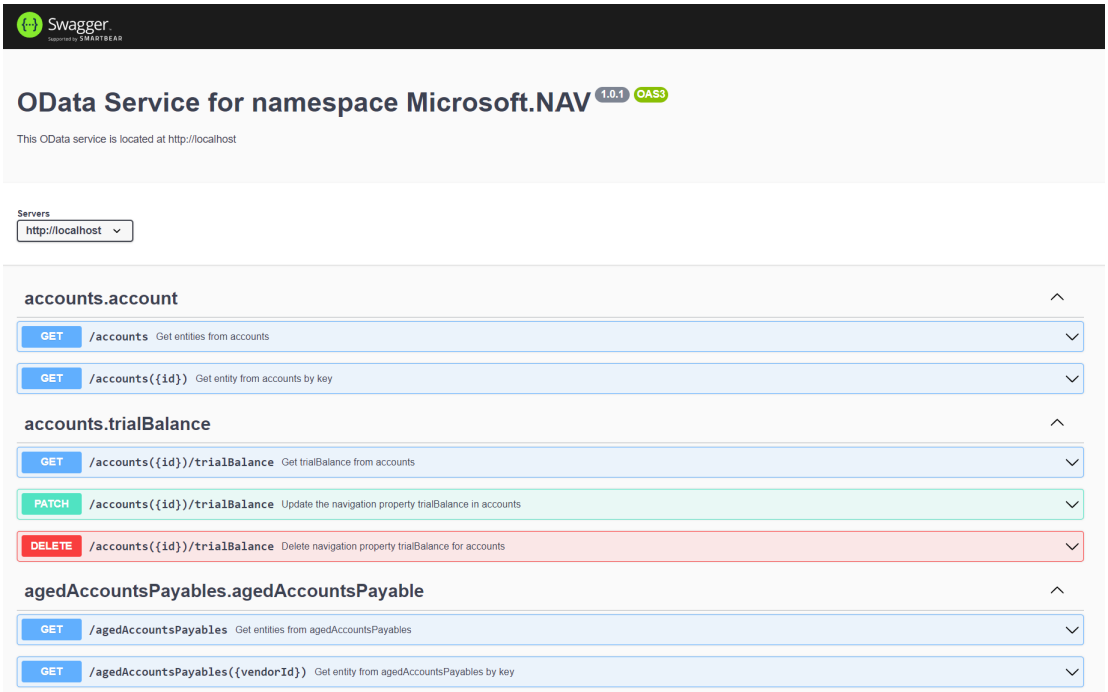

Abbildung 3.1: Beispiel UI von Swagger (Quelle: *Host the Swagger UI for your BC OpenAPI spec in your BC container* (2023))

3.3 Vorgehensweise

Damit das zu Entwickelnde TCMS die eben genannten Anforderungen erfüllen kann werden diese in User-Stories aufgeteilt. Nachfolgend sind vier User-Stories aufgeführt, die erklären, welche Funktionalitäten das TCMS beinhaltet, um die 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 Architektur Schichten aufteilen
- Datenbank mit Docker aufsetzen
- Datenbank manuell beschreibbar

3.3.2 Erstellen und suchen von Test-Environments

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

Die Akzeptanzkriterien sind wie folgend:

- Erstellen von Test-Environments über Swagger UI
- Suchen von Test-Environments über Swagger UI
- Hinzufügen und Entfernen von Testplänen über Swagger UI

3.3.3 Erstellen und suchen von Testplänen, Testfällen und Testimplementationen

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

Die Akzeptanzkriterien sind wie folgend:

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

3.3.4 Erstellen und suchen von Testläufe

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

Die Akzeptanzkriterien sind wie folgend:

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

4 Konzept

Im vorherigen Kapitel wurden die Anforderungen des zu entwickelnden TCMS festgelegt. Dieses Kapitel gibt nun einen Überblick über die Architektur, die dafür verwendet wird und vertieft sich in die Entwicklungsphasen des TCMS.

4.1 Architektur

Grundlegend repräsentiert eine Softwarearchitektur die Organisation und den Aufbau eines Systems, zudem ist die Architektur eines der wichtigsten Komponente bei der Softwareentwicklung. Die Wahl der Softwarearchitektur ist zu Beginn eines Softwareprojektes sehr relevant, da diese hohen Einfluss auf den späteren verlauf des Projektes haben wird und es sehr kostenintensiv ist diese umzustellen. Eine gut strukturierte Software Architektur bringt dementsprechend einige Vorteile:

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

Richards und Ford (2020)

4.1.1 Backend

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

- Präsentationsschicht

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

- Applikationsschicht

  Diese Schicht interagiert mit der Präsentationsschicht und koordiniert Applikation Aktivitäten. Hier werden abhängig von den gelieferten Daten dementsprechende Domain Objekte erzeugt und der Infrastruktur 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 ganze Business-Logik.

- Infrastrukturschicht

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

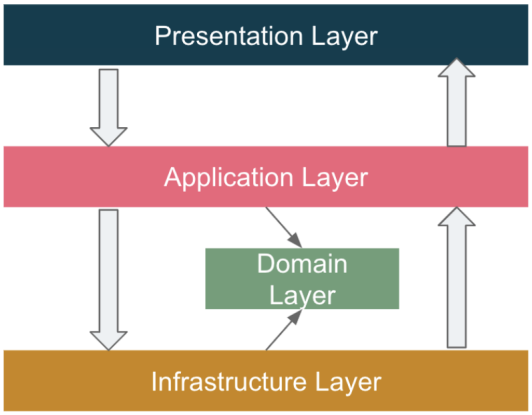

Abbildung 4.1: Übersicht der Schichten (Quelle: Zhang (2023))

DDD ist ein Design Ansatz, der sich auf den Input der Domainexperten konzentriert und somit die Entitäten des Systems dadurch 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)

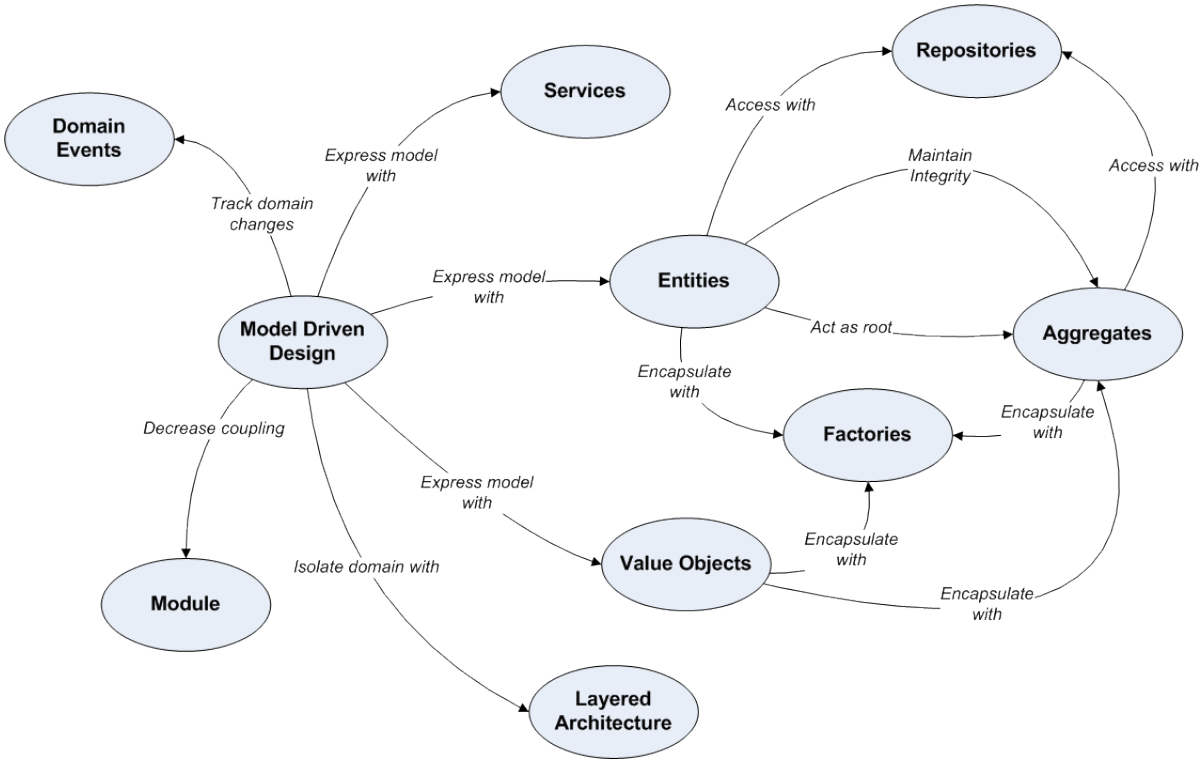

Abbildung 4.2: Komponenten von DDD (Quelle: *Domain Driven Design - Building blocks Domain Driven Design* (2023))

4.2 Vorentwicklungsphase

In dieser Phase handelt es sich hauptsächlich um die Planung der Implementierung des TCMS. Zuerst wurde eine Übersicht von dem schon bestehenden *Test Automation System* (TAS) erzeugt, das zeigt wie es mit dem TCMS interagiert. Dabei greift das TAS über die Technologie Automise über eine von TCMS bereitgestellte API darauf

zu und erstellt Testläufe. Bei der Beschreibung von Abbildung 4.5 wird darauf genauer eingegangen (Siehe Unterabschnitt 4.2.1).

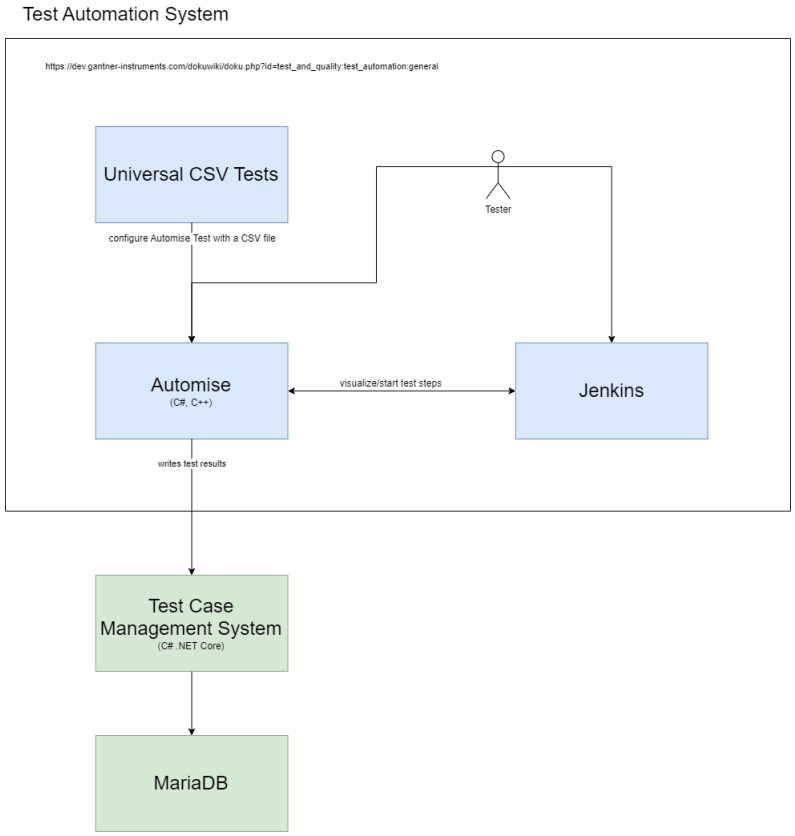

Abbildung 4.3: TAS von GI

Durch die von Kapitel 3 gegebenen Anforderungen und dem bestehendem TAS wurde dementsprechend geplant.

4.2.1 Domain Model

Zunächst wurde das Domain-Model entwickelt, um die Entitäten und ihre Beziehungen darzustellen. Dabei wurden exakt, die von den Anforderungen geforderten Features abgebildet.

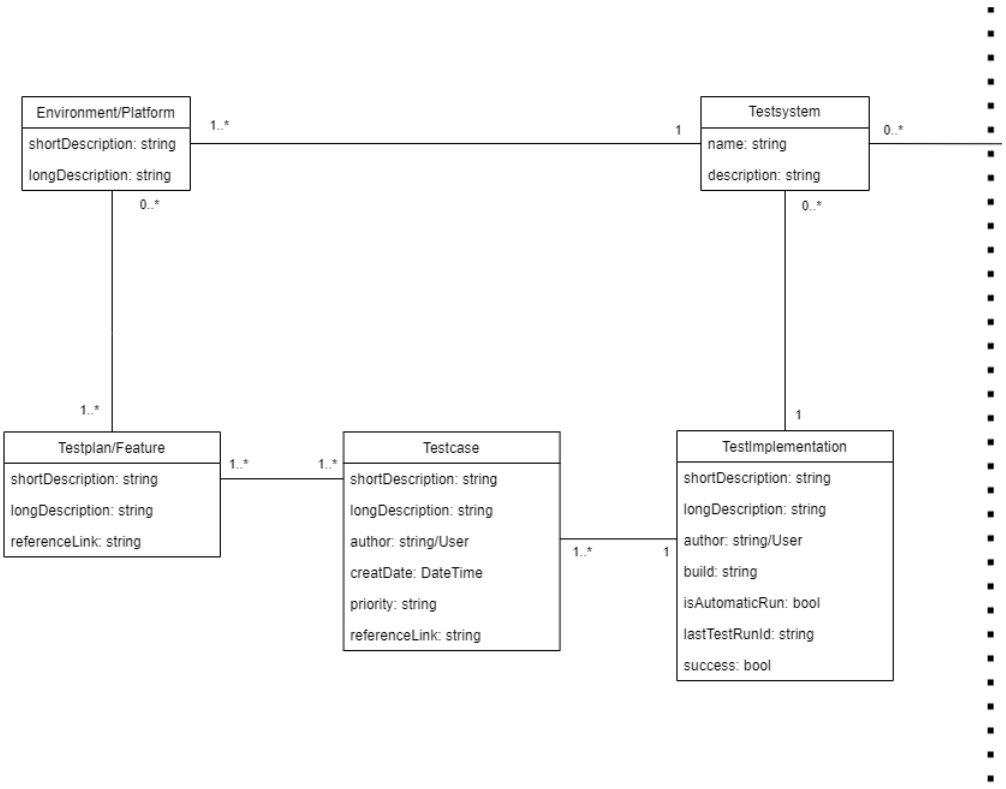

Abbildung 4.4: Domain-Model (Teil 1) aus den Anforderungen von Kapitel 3

Bei den in Abbildung 4.4 abgebildeten Entitäten handelt es sich um die folgenden Anforderungen:

- Produkt und Testsysteme beschreiben

  Das Produkt wurde als Test-Environment Entität modelliert, dass kann beispielsweise ein Gerät von der Firma sein. Das Testsystem bildet beispielsweise eine Prüfwand mit mehreren Geräten ab mit denen Tests durchgeführt werden. Jedes Produkt hat eigene Testsysteme.

- Features beschreiben

  Feature wurde als Testplan Entität modelliert. Ein Feature besteht aus mehreren Testfällen und bildet ein Feature von einem Gerät ab wie beispielsweise Netzwerk Connectivity.

- Features zu einem Produkt zuordnen

Die erstellten Testpläne können nach dem Erstellen zu Test-Environments zugeordnet werden. Somit können Testpläne für andere Environments wiederverwendet werden.

- Testfälle beschreiben

  Ein Testfall besteht aus mehreren Testimplementationen. Diese Testimplementationen sind für jeden Testfall unique und können nicht wiederverwendet werden. Ein Testfall ist beispielsweise eine überprüfung ob das Gerät eine *Dynamic Host Configuration Protocol* (DHCP) Adresse erhalten hat.

- Testfälle zu einem Feature zuordnen

  Die Teställe können dann zu Features zugeordnet werden. Somit können Testfälle wiederverwendet werden.

- Testimplementierungen beschreiben

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

- Testimplementierungen zu Testfällen und Testsysteme zuordnen

  Testimplementierungen könne zu Testfällen und zu Testsysteme zugeordnet werden. Durch die Zuordnung von Testimplementierungen zu Testsystemen ist es einfacher die Testimplementierungen von einer Environment darzustellen. Somit kann beispielsweise ein Report generiert werden, der einen Überblick über jede Testimplementation von einem Environment gibt.

Bei den in Abbildung 4.5 abgebildeten Entitäten handelt es sich um die folgenden Anforderungen:

- Testläufe von Testimplementierungen zu einem Testsystem zuordnen

  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

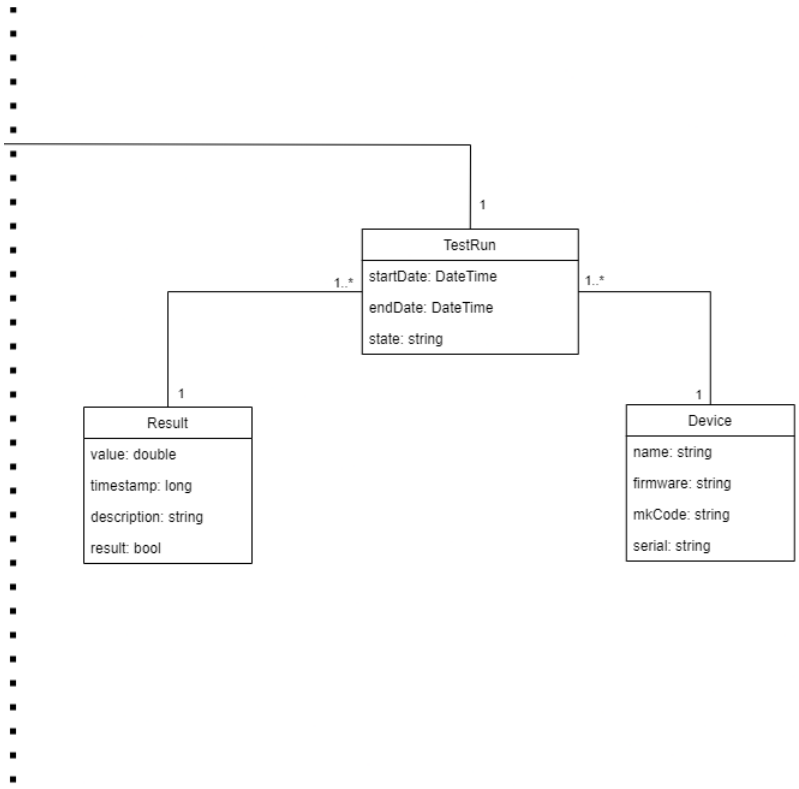

Abbildung 4.5: Domain-Model (Teil 2) aus den Anforderungen von Kapitel 3

getestet wurde. Testläufe werden zum Zeitpunkt des Erstellens zu den jeweiligen Testsystemen zugeordnet, mit denen diese durchgeführt worden sind.

- Resultate von bestimmten Testläufen abzufragen

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

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

4.3 Datenbank Server

Der Datenbank Server wird mit Docker aufgesetzt. Über die bereitgestellten APIs des TCMS wird dann über die schichten der verwendeten Architektur in die Datenbank

geschrieben und gelesen werden (Siehe Unterabschnitt 4.1.1). Wie dies implementiert wird, wird im folgenden Kapitel beschrieben.

5 Technische Umsetzung

Damit das Ziel dieser Arbeit, ein TCMS zu entwickeln, erreicht wird, werden die im Kapitel 3 auf Seite 25 definierten User-Stories mithilfe der erwähnten Technologien aus Abschnitt 2.5 umgesetzt. Dabei wird auf den Scrum-Prozess und im Anschluss auf die User-Stories detailliert eingegangen.

5.1 Scrum

Scrum ist grundsätzlich eine Projektmanagement-Methode um in der Softwareentwicklung agil zu arbeiten. Bei dieser Methode geht es darum, dass ein Team Aufgaben von einer Aufgabenstellung in kleinen Schritten, sogenannte Sprints, angeht.

Rubin (2012)

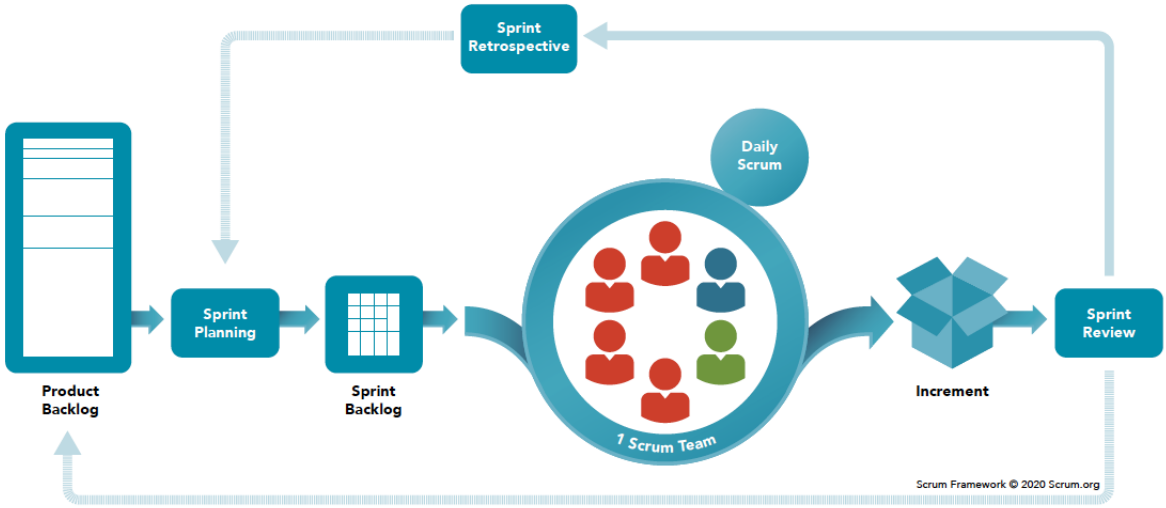

Abbildung 5.1: Scrum Prozess (Quelle: *The Home of Scrum* (2023))

Für die Implementierung der User-Stories wurde der oben gezeigte Prozess angewendet (Abbildung 5.1).

5.2 Implementierung der User-Stories

Dieser Abschnitt zeigt den Ablauf der Implementierung des TCMS dieser Arbeit. Gesamt ist die Implementierung auf vier User-Stories aufgeteilt, die in folgender 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 in jede Schicht eingegangen (Siehe: Unterabschnitt 5.2.2 auf Seite 38). Dabei werden die User-Stories in folgender Reihenfolge in die Schichten implementiert:

  - Domainschicht
  - Infrastrukturschicht
  - Applikationsschicht
  - Präsentationsschicht

- Erstellen und suchen von Testplänen, Testfällen und Testimplementationen

  Um zu zeigen, wie EF Core Tabellen konfiguriert, werden bei dieser User-Story detailliert auf die Konfigurationen eingegangen (Siehe: Unterabschnitt 5.2.3 auf Seite 41).

- 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 auf Seite 43).

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

5.2.1 Projekt und Datenbank Aufsetzen

Damit eine Grundlage für die folgenden User-Stories vorhanden ist, wurde die Struktur des Projektes in vier Teil Projekte aufgeteilt. Dabei ist jedes Projekt die Projektion einer Schicht der verwendeten Architektur (Siehe: Unterabschnitt 4.1.1).

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

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

Quellcode 2: Starten eines MariaDB Containers

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

```
CREATE DATABASE tcms_mariadb;
USE tcms_mariadb;
```

Quellcode 3: Erstellen und verwenden eines Datenbankschemas

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

```
{
    [...]

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

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

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 wird in Quellcode 5 gezeigt, dass die Funktion *AddDbContext* den Typ

```
[...]

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

EFContext bekommt. Diese Klasse kümmert sich um alle Tabellen Konfigurationen, die verwendet werden (Siehe: Quellcode 18).

Quellcode 15, 16 und 17 im Anhang auf Seite 52 zeigt die *Startup* und *Program* Klasse, die im Zusammenhang der oben gezeigten Ausschnitte, alles nötige initiiert.

5.2.2 Erstellen und suchen von Test-Environments

Zuerst wurde mit den Domain-Models, *Test-Environment* und *Testsystem*, in der Domainschicht angefangen. 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.

```
public interface ITestEnvironmentRepository
{
    string NextIdentity();
    Task<TestEnvironment> FindById(string id);
    Task<List<TestEnvironment>> FindByShortDescription(string shortDescription);
    Task<List<TestEnvironment>> GetAll();
    Task Add(TestEnvironment testEnvironment);
    Task Remove(string id);
    Task Update(TestEnvironment testEnvironment);
}
```

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

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

```
[...]

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

    [...]

    return testEnvironment;
}

[...]
```

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

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

Damit wir das von Quellcode 7 ausgelesene Test-Environment-Objekt weiter verwenden können, wird ein Test-Environment-Service in der Applikationsschicht einwickelt. Dabei wird für den Service zuerst wieder ein Interface definiert, welches die *TestEnvironemntService* Klasse implementiert.

```
[...]

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 *TestEnvironemntService* Klasse

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 dann in der Präsentationsschicht weiter verarbeitet.

```
[...]

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

[...]
```

Quellcode 9: Ausschnitt der *TestEnvironemntController* Klasse

Der in Quellcode 9 gezeigte Code bekommt das umgewandelte Test-Environment-Objekt (DTO) von der Applikationsschicht zurück und schickt es dem Client weiter. Zu guter Letzt werden die *TestEnvironemntService* und *TestEnvironmentRepository* Klassen in der *EFContext* Klasse (Siehe: Quellcode 15) für den *Dependency Injection* (DI) Container registriert.

Der gezeigte Ablauf war die Suche eines Test-Environments. Beim Erstellen passiert ähnliches, nur dass der Client ein DTO über den bereitgestellten Endpunkt der API schickt (Siehe: Abbildung 5.2). Die *TestEnvironemntController* 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.

```
[...]

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

[...]
```

Quellcode 10: Ausschnitt der *TestEnvironmentRepository* Klasse

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

in die Datenbank.

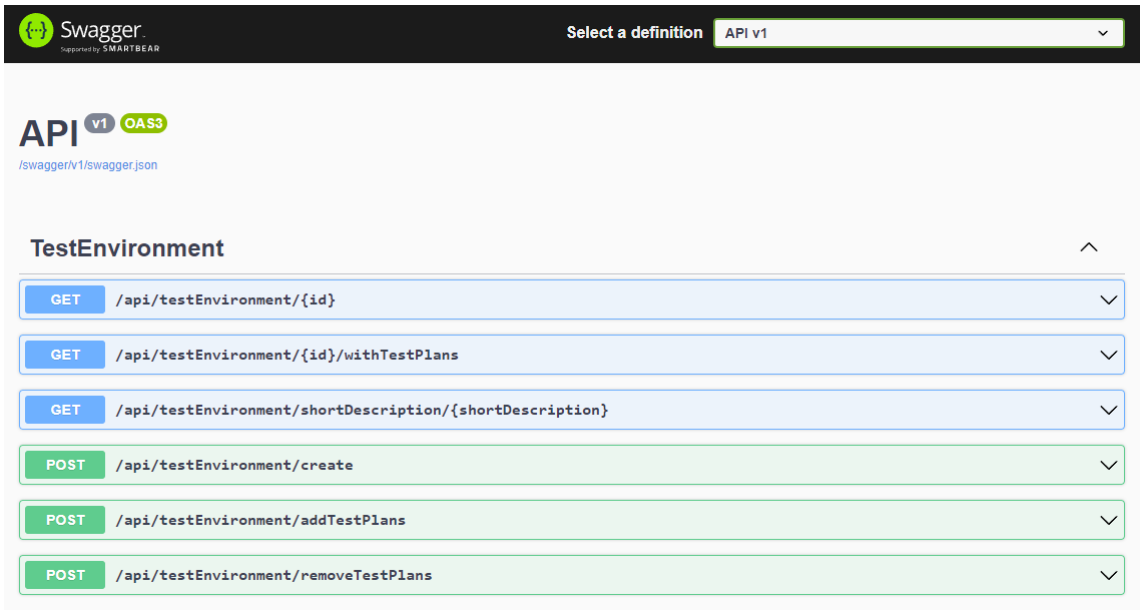

Abbildung 5.2: Test-Environment UI von Swagger

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

5.2.3 Erstellen und suchen von Testplänen, Testfällen und Testimplementationen

Wie bei der vorherigen User-Story ist der Ablauf der Implementierung für die Testpläne, Testfälle und für die Testimplementationen der gleiche.

Zuerst wurden die Domain-Models erstellt, dann das Repository, folgend mit dem Service und als Letztes wurde der Controller implementiert. Zudem werden bei Start des Projektes wieder die *Controller* Klassen gescannt und dafür eine Weboberfläche generiert.

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

```
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

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.

```
[...]

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

[...]

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

[...]
```

Quellcode 12: Ausschnit aus der *TestEnvironmentConfiguration* und *TestPlanConfiguration* Klasse

Der in Quellcode 12 gezeigte Code konfiguriert ein Teil der Tabellen *TestEnvironment* und *TestPlan*. Damit wird bestimmt, dass die Tabellen *TestEnvironment* und *TestPlan*

eine Many-to-one Beziehung zu der *Junction* Tabelle haben.

```
[...]

builder.HasKey(e => new {e.TestEnvironmentDomainId, e.TestPlanDomainId});

// TestEnvironmentPlan - many to one - TestEnvironment
builder
    .HasOne<TestEnvironment>()
    .WithMany(e => e.TestEnvironmentPlans)
    .HasPrincipalKey(e => e.DomainId)
    .IsRequired();

// TestEnvironmentPlan - many to one - TestPlan
builder
    .HasOne<TestPlan>()
    .WithMany(e => e.TestEnvironmentPlans)
    .HasPrincipalKey(e => e.DomainId)
    .IsRequired();

[...]
```

Quellcode 13: Ausschnit aus der *TestEnvironmentPlanConfiguration* Klasse

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

Somit wurde mit EF Core eine Many-to-Many Beziehung hergestellt. Nun ist es möglich Testpläne einer Test-Environemnt zuzuweisen.

Um ein Test-Environment mit den hinzugefügten Test-Pläne abzufragen, wurde eine *TestEnvironmentManager* Klasse in der Applikationsschicht implementiert. Dieser Manager hat Zugriff auf die *TestEnvironmentRepository* und *TestPlanRepository* Klassen. Quellcode 19 im Anhang auf Seite 56 zeigt, wie dies erfolgt.

5.2.4 Erstellen und suchen von Testläufe

Der Ablauf der Implementierung ist gleich wie bei 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 effizient abgefragt werden können.

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

```
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: Ausschnit aus der *TestRunConfiguration* Klasse

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

Die Testimplementationen sind, wie die Testläufe, in einer Many-to-one Beziehung zu der *Testsystem* Klasse. Dadurch ist es möglich, durch den 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 es dem TAS von GI Testläufe automatisch zu erstellen. Ein Ablauf eines TAS sieht folgend 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*, von der *Testlauf* Klasse, hinzugefügt. Der Testlauf wird daraufhin in der Datenbank aktualisiert.

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

Um manuell Testläufe zu erstellen, gibt es ein UI, dass mit *swagger* generiert wurde.

6 Evaluierung und Ausblick

In diesem Kapitel werden die erzielten Ergebnisse der technischen Umsetzung des TCMS Backends noch einmal reflektiert. Weiters werden Verbesserungsmöglichkeiten und ein kurzer Ausblick auf zukünftige Features für das TCMS gegeben.

Ziel dieser Arbeit war es ein funktionierendes TCMS Backend zu implementieren. Anhand der Kommunikation mit den Domain-Experten sind die aufgelisteten Kriterien in Kapitel 3 auf Seite 23 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 25 die User-Stories für die Umsetzung der gegebenen Anforderungen geplant.

Mit den beschriebenen Technologien in Abschnitt 2.5 auf Seite 19 erfolgte eine dementsprechende Umsetzung eines TCMS Backends. Angefangen mit der User-Story *Erstellen und suchen von Test-Environments* ist die vierschichtige Architektur implementiert worden. Mithilfe dieser Architektur konnten dann 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.

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.

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

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)

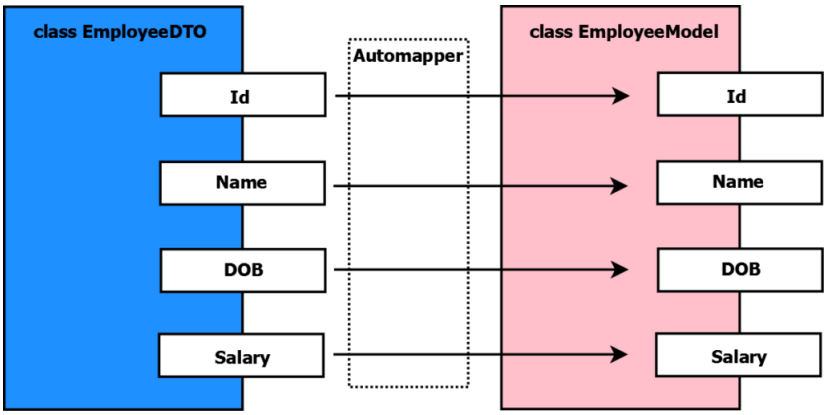

Abbildung 6.1: Funktionsweise von *AutoMapper* (Quelle: Sanjay (2020))

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

6.3 Zusammenfassung

Abschließend lasst sich zusammenfassen...

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).

Domain Driven Design - Building blocks Domain Driven Design (2023). URL: https://uniknow.github.io/AgileDev/site/0.1.8-SNAPSHOT/parent/ddd/core/building_blocks_ddd.html (besucht am 06. 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).

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).

Tricentis Test Management for Jira (2023). Atlassian Marketplace. URL: https://marketplace.atlassian.com/apps/1228672/tricentis-test-management-for-jira (besucht am 17. 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).

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

```
public class Startup
{
    private readonly IConfiguration _config;

    public Startup(IConfiguration config) { _config = config; }

    // This method gets called by the runtime.
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        // Register database with the DI container
        services.AddDbContext<EFContext>(dbContextOptions => {
            var connectionString = _config
                .GetConnectionString("MariaDbConnectionString");

            dbContextOptions
                .UseMySql(connectionString,
                    ServerVersion.AutoDetect(connectionString));
        });

        // Register repositories with the DI container
        services.AddTransient<ITestCaseRepository, TestCaseRepository>();
        services.AddTransient<ITestPlanRepository, TestPlanRepository>();
        [...]

        // Register services with the DI container
        services.AddTransient<ITestCaseService, TestCaseService>();
        services.AddTransient<ITestPlanService, TestPlanService>();
        [...]

        services.AddControllers();
        services.AddSwaggerGen(c => {
                c.SwaggerDoc("v1",
                    new OpenApiInfo { Title = "API", Version = "v1" });
        });
    }

[...]
```

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 1. Juli 2023 Marco Prescher