Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

Die testgetriebene Entwicklung eines Spielzeiten-Planers für den Jugendfußball

Bachelorarbeit

vorgelegt von

Florian Ohmes

28. November 2024

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Erstgutachter: Dr. Jens Bendisposto
Zweitgutachter: Dr. Markus Brenneis

Selbstständigkeitserklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 28. November 2024

Florian Ohmes

Zusammenfassung

Diese Bachelorarbeit dokumentiert die testgetriebene Entwicklung eines webbasierten Spielzeitenplaners für den Jugendfußball. Das Projekt wurde mit Spring Boot entwickelt und kombiniert den Ansatz des Test-Driven Development (TDD) mit den Prinzipien der Onion-Architektur. Es soll als ein Echtwelt-Beispiel für die TDD-Methode dienen.
Der Entwicklungsprozess umfasst die Ausarbeitung der Web-, Service- sowie der Persistenzschicht. Mithilfe von `MockMvcTests` und der Java-Bibliothek `Jsoup` ist jede einzelne Seite strukturiert entwickelt worden, von den einzelnen Bereichen über grundlegende UI-Komponenten, wie Formulare, bis hin zu dynamischen Inhalten, die mithilfe von Thymeleaf gerendert werden. Außerdem sind die Hauptaufgaben der Web-Steuereinheiten (Controller) mithilfe geeigneter Test überprüft worden. In der Service-Schicht sind Tests für zentrale Funktionen für das Speichern von Benutzeranfragen, das Bereitstellen von Daten sowie das Berechnen der Scores und Spielzeiten geschrieben worden. In der Persistenzschicht ist schließlich sichergestellt worden, dass das Speichern, Laden und Filtern der entsprechenden Entitäten ordnungsgemäß funktioniert.
Der Spielzeitenplaner richtet sich an Fußballlehrende im Amateur- bzw. im Jugendbereich und Sinn und Zweck der Anwendung ist es, eine faire und nachvollziehbare Verteilung der Spielminuten basierend auf strukturierten, individuellen Trainingsbewertungen zu ermöglichen. Außerdem soll er das Trainerteam dabei unterstützen, begründet und datenbasiert Entscheidungen bezüglich der Spielzeiten der Spieler zu treffen und diese transparent zu kommunizieren, um potenzielle Konflikte zwischen Spielern, Eltern und Trainerteam zu vermeiden.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Softwareentwicklung steht heutzutage vor vielfältigen Herausforderungen, da moderne Software eine Vielzahl an Anforderungen erfüllen muss. So sollte diese beispielsweise skalierbar, flexibel und sicher wie auch qualitativ hochwertig und wartbar sein. Darüber hinaus sollte sie auf die Wünsche der Kunden zugeschnitten sein, denn schließlich sind diese die Hauptnutzenden und wollen durch sie einen Mehrwert erlangen.
Dabei hat sich Test-Driven Development (TDD) als eine methodische Herangehensweise an Softwareentwicklung herauskristallisiert. Sie bietet einen strukturierten und qualitätsorientierten Ansatz, der heutzutage immer häufiger praktiziert wird. Test-Driven Development kann zur Qualität und Robustheit moderner Software beitragen, indem Anforderungen präzise definiert und Fehler frühzeitig erkannt werden können. Außerdem erhöht es die Modularität und damit verbunden auch die Wartbarkeit von Software.
Die im Rahmen dieser Arbeit erschaffene Software – der Spielzeitenplaner – ist nach den Prinzipien des Test-Driven Development entwickelt worden und soll als Echtwelt-Beispiel für die Methode dienen. Diese Arbeit wiederum soll den Prozess der testgetriebenen Entwicklung des Spielzeitenplaners dokumentieren und TDD-Neulingen somit einige Anregungen bieten, wie moderne Software strukturiert entwickelt werden kann. Dabei werden an zahlreichen Stellen konkrete Tests gezeigt und deren Sinn und Zweck erläutert.
Doch was ist nun eigentlich der Spielzeitenplaner und welchen Nutzen soll er welcher Zielgruppe bringen? Der Spielzeitenplaner ist eine Softwarelösung für den Jugendfußball im Amateurbereich. Er soll Fußballlehrende dabei unterstützen, begründet sowie datenbasiert Entscheidungen bezüglich der Spielzeiten der einzelnen Spieler zu treffen.
Nicht selten kommt es zwischen Trainerteam und Spielern und/oder Eltern zu Diskussionen über die Einsatzzeit eines Akteurs in einem Fußballspiel. Dabei gibt es eine Diskrepanz zwischen der Wahrnehmung des Spielers und seinen Eltern sowie der des Trainerteams. Erstere sind der Meinung, ihr Sohn hätte mehr Spielzeit verdient als er im Spiel tatsächlich bekommen hat und bringen demzufolge Gründe hervor, warum ihre Ansicht gerechtfertigt ist. Das Trainerteam hingegen vertritt unter Umständen eine andere Ansicht, da sie den Spieler über Wochen hinweg im Training gesehen und beurteilt haben und dementsprechend zu einem anderen Ergebnis kommen.
Mit Blick auf die zuvor beschriebene Diskussion über die Spielzeit eines Spielers stellen sich gleich mehrere Fragen: Wie viel Spielzeit hat jeder einzelne Spieler verdient? Wie viele Spielminuten können für einen bestimmten Spieler als `fair` erachtet werden? Und was bedeutet `fair` in diesem Zusammenhang überhaupt und wie lässt sich eine faire Aufteilung der Spielzeit ermitteln?
Diese Fragen können mit dem Spielzeitenplaner beantwortet werden. Mit ihm ist es möglich, die eigene Mannschaft im Bezug auf die Spielzeiten der einzelnen Spieler zu verwalten. Dazu können Fußballlehrende eigene Kriterien erstellen anhand derer Bewertungen durchgeführt werden sollen. Nach jedem Training wird jeder Spieler im Hinblick auf jedes einzelne Kriterium bewertet – auch `Recap` genannt. Diese Bewertungen werden gespeichert und als Grundlage für die Berechnung eines Gesamt-Scores verwendet. An Spieltagen können Trainerteams dann die Spielzeiten planen, indem aufgrund der berechneten Scores Kader,

Startelf und Wechsel bestimmt werden.
Das Erstellen von Kriterien und Bewerten der Spieler nach jedem Training fördert die Objektivität bei der Bestimmung der Spielzeiten sowie die Fähigkeit, begründet Entscheidungen zu treffen und diese zu vertreten. Darüber hinaus kann ein solch strukturierter Ansatz – wenn Entscheidungen denn transparent kommuniziert und den Beteiligten erklärt werden – zu mehr Verständnis auf Spieler- und Elternseite führen. Schließlich macht es die Situation für den Spieler greifbarer und er weiß, wie bzw. in welchen Bereichen er sich verbessern kann und muss, um auf mehr Spielzeit zu kommen.
In der hier vorliegenden Arbeit sollen nun zunächst einmal einige theoretische Grundlagen erläutert werden – Test-Driven Development sowie fußballerische Grundlagen, ehe im Anschluss der grundsätzliche Aufbau und die Funktionsweise des Spielzeitenplaners erklärt wird. Es folgt die testgetriebene Entwicklung der Anwendung, wobei in jeder Architekturschicht – Web, Service und Persistenz – genau beschrieben und erläutert werden soll, wie dort testgetrieben entwickelt worden ist und konkrete Beispiele aus dem Projekt des Spielzeitenplaners gezeigt werden. Abschließend werden der Entwicklungsprozess des Projektes sowie die gewählte Methode (TDD) kritisch reflektiert.

2 Theoretische Grundlagen

Dieser Arbeit sowie der Entwicklung des Projektes des Spielzeitenplaners liegen einige theoretische Konzepte, Methoden und Modelle zugrunde: zum einen der Ansatz des Test-Driven Development aus der Softwareentwicklung, zum anderen einige fußballerische Konzepte und Richtlinien, die in den folgenen Kapiteln kurz erläutert werden sollen. Außerdem richtet sich die Struktur des Projektes an der von Jeffrey Palermo in 2008 vorgeschlagenen `Onion-Architektur` [Pal08], die an dieser Stelle jedoch nicht explizit erläutert werden soll.

2.1 Test-Driven Development (TDD)

Test-Driven Development (TDD) – zu deutsch: testgetriebene Entwicklung – ist die Bezeichnung für eine methodische Herangehensweise der Softwareentwicklung. Die dieser Methode zugrunde liegenden Ideen, Prinzipien und Praktiken sowie ihr Nutzen sind bereits 2003 von Kent Beck ausführlich beschrieben worden [Bec03]. Für detaillierte Ausführungen kann die zuvor genannte Monographie genutzt werden. Im Folgenden soll dennoch kurz auf diese wichtige theoretische Grundlage, die eine zentrale Rolle bei der Entwicklung dieses Projektes spielt, eingegangen werden.
Ein Zyklus des Test-Driven Development setzt sich aus den folgenden Phasen zusammen: `Red`, `Green` und `Refactor`. Diese Phasen sind der Reihe nach zu durchlaufen und der Zyklus so lange iterativ zu wiederholen, bis das Software-Projekt mit all seinen gewünschten Funktion schließlich vollständig realisiert ist.
Die Phase `Red` ist die erste Phase, deren Ziel es ist, einen Test zu schreiben, der fehlschlägt – in der IDE also rot angezeigt wird. Der Inhalt des Tests beschreibt eine gewünschte Funktionalität, die in den folgenden Phasen dann im Produktiv-Code implementiert werden soll. Der Test schlägt also erwartungsgemäß fehl, da die Funktionalität zu Beginn noch nicht vorhanden ist. Die Farbe Rot bezieht sich dabei nicht nur auf fehlschlagende Tests, sondern auch auf eben jene Code-Schnipsel, die zunächst gar nicht kompilieren. Unter Umständen ist es also beispielsweise möglich, dass eine Klasse erst einmal erstellt, ein Konstruktor implementiert oder eine Funktion definiert werden muss, bevor mit dem Schreiben des Tests fortgefahren werden kann.
Auf die `Red`-Phase folgt die `Green`-Phase. Ist ein Test implementiert und fehlgeschlagen, so kann ein Stück Code geschrieben werden, dass gerade groß genug ist, sodass es diesen Test bestehen – also grün werden – lässt. Becks Fokus liegt dabei auf einem schnellen und unkomplizierten Umsetzen des gewünschten Features, bei dem jegliche Programmier-Sünden erlaubt sind. Diese können und sollen dann zu einem späteren Zeitpunkt ausgemerzt werden. Diese Herangehensweise kann als eine strenge Handhabung der TDD-Praktik bezeichnet werden. Im Gegensatz dazu ist es bei einer lockeren Handhabung des TDD durchaus denkbar, ein Stück Code direkt auf die beabsichtigte Weise zu implementieren, zum Beispiel im Falle routinemäßiger Aufgaben, die von Entwickelnden aufgrund ihrer Erfahrung bereits antizipiert werden können.

Die dritte Phase ist `Refactor`. Hier geht es um die Überarbeitung und Verbesserung des vorliegenden (Produktiv-)Codes, ohne dabei die Tests zu beschädigen. Nach einem `Refactor`-Schritt sind also nach wie vor alle bereits bestehenden Tests grün. Typischerweise werden beim `Refactoring` Duplikationen im Code entfernt, Code-Schnipsel in Methoden ausgelagert oder ähnliche Schritte zur qualitativen Code-Verbesserung durchgeführt.
Test-Driven Development verfolgt das Ziel, sauberen Code zu produzieren, der funktioniert. Es stellt sicher, dass Software den definierten Anforderungen entspricht und fördert ein stärkeres Bewusstsein für den Zweck des vorliegenden Codes. Darüber hinaus kann TDD als eine Art Frühwarnsystem fungieren, das Alarm schlägt, wenn ein Test fehlschlägt, denn auf diese Weise können Fehler im vorliegenden Code frühzeitig erkannt und behoben werden. Außerdem kann TDD die Modularität der Software erhöhen und diese letztendlich wartbarer machen, denn durch das Testen werden kleine, unabhängige Einheiten geschaffen mit klar definierten Verantwortlichkeiten, Schnittstellen sowie Ein- und Ausgaben.

2.2 Fußballerische Grundlagen und Konzepte

Um Sinn und Zweck sowie die Funktionsweise der in dieser Arbeit vorliegenden Anwendung vollumfänglich verstehen zu können, sind neben softwaretechnischen Grundlagen auch das Wissen über grundlegende Konzepte des Fußballs sowie Richtlinien und Bestimmungen des Jugendfußballs notwendig.
Der Fußballverband Niederrhein – kurz: FVN – ist einer der 21 Verbände des Deutschen Fußballbundes (DFB). Er ist unter anderem für die Organisation eines geregelten Spielbetriebs im Amateurfußball am Niederrhein verantwortlich. Dies beinhaltet sämtliche Alters- aber auch Leistungsklassen im Senioren- und Juniorenbereich. Jedes Jahr werden auf der Webseite des FVN sogenannte Durchführungsbestimmungen veröffentlicht, die den Rahmen für die kommende Saison bilden [Fus24]. Dort ist beispielsweise die Dauer eines Fußballspiels für jede Altersklasse festgelegt.
Für die C-Jugend, die in der Saison 2024/2025 aus den Jahrgängen 2010 und 2011 besteht, beträgt die Spielzeit insgesamt 70 – eine Halbzeit also 35 – Minuten. Gespielt wird mit elf Spielern pro Mannschaft, weitere fünf Spieler dürfen im Verlauf eines Spiels ein- und wieder ausgewechselt werden. Die elf Spieler einer Mannschaft, die zu Beginn des Spiels auf dem Platz stehen, bilden die sogenannte Startelf. Die restlichen Spieler werden auch als Reservespieler, Reserve oder einfach nur Bank – in Anlehnung an die Sitzgelegenheit, auf der die Spieler Platz nehmen – bezeichnet.
Startelf und Reservespieler bilden zusammen den Kader. Er setzt sich daher aus all denjenigen Spielern zusammen, die vom Trainerteam für ein Spiel nominiert werden, und kann von Spiel zu Spiel variieren, je nach Gesundheitszustand oder Trainingsstand der einzelnen Spieler oder aber aufgrund privater Termine der Akteure. Für weitere Bezeichnungen und allgemeine Fußball-Regeln sind die vom Deutschen Fußball-Bund veröffentlichten Fußball-Regeln zu studieren [Deu24].
Des Weiteren ist festzustellen, dass jede Mannschaft mit einer bestimmten Formation spielt. Die Formation spiegelt die räumliche Anordnung der Spieler auf dem Platz wider und hat zum Ziel, gewisse Symbiose-Effekte zwischen den einzelnen Spielern hervorzurufen sowie

für eine ausgeglichene Aufteilung der Akteure auf dem Platz zu sorgen. Außerdem können auf Basis der gewählten Formation spezifische Taktiken gelehrt und angewendet werden, die für die hier vorliegende Arbeit jedoch nicht von Relevanz sind.
Beispiele für beliebte Formationen sind `4-2-3-1`, `4-3-3` oder `3-5-2`. Dabei werden im Namen die Anzahlen der Spieler nach Positionsgruppen sortiert und durch einen Bindestrich getrennt angegeben. `4-2-3-1` bedeutet also, dass die Abwehr aus vier Spielern – der Viererkette – besteht, der Sturm hingegen aus nur einem Spieler. Während sich die erste Zahl auf die Anzahl der Abwehrspieler bezieht und die letzte Zahl die Anzahl der Stürmer referenziert, bilden die restlichen Zahlen in der Mitte des Ausdrucks die Anzahl der Mittelfeldspieler, im Falle des `4-2-3-1` zwei defensive und drei offensive Mittelfeldspieler. Der Torwart bleibt bei der Bezeichnung einer Formation stets unerwähnt, da er immer vorhanden sein muss und immer nur aus einer Person besteht.
Innerhalb einer Formation nimmt jeder Spieler eine bestimmte Position ein. Eine Formation kann somit auch als eine Liste von elf Positionen interpretiert werden. Auch wenn im Kindesalter noch großer Wert auf eine ganzheitliche fußballerische Ausbildung gelegt wird, so ist es ab dem Jugendalter üblich, Spieler positionsspezifisch auszubilden. Jede Position bringt zum Teil sehr unterschiedliche Anforderungen mit sich, weshalb nicht jeder Spieler auf jeder Position spielen kann. Gängige – oder grundlegende – Positionen und ihre Bezeichnungen sind beispielsweise der Torwart (`TW`), der Innenverteidiger (`IV`), der linke/rechte Außenverteidiger (`LV/RV`), das zentrale defensive Mittelfeld (`ZDM`), das linke/rechte Mittelfeld (`LM/RM`), das zentrale offensive Mittelfeld (`ZOM`) und der Stürmer (`ST`). Positionsbezeichnungen werden üblicherweise in Großbuchstaben angegeben und versuchen, die Rolle der Position widerzuspiegeln.
Im Rahmen des Spielzeitenplaners haben Nutzende die Möglichkeit, eine eigene Formation und Positionen – basierend auf den oben erläuterten Konventionen – zu erstellen.

3 Aufbau und Funktionsweise des Spielzeitenplaners

Der Spielzeitenplaner ist grundsätzlich in vier verschiedene Bereiche eingeteilt: Team, Recap, Spielzeiten planen und Einstellungen. Diese Bereiche und die damit verbundenen Grundfunktionen des Spielzeitenplaners sollen im Folgenden kurz beschrieben und erläutert werden.
Als Erstes ist der Team-Bereich zu nennen, der sich im Wesentlichen aus zwei Teilen zusammensetzt: der Team-Seite und der Spieler-Seite. Auf Ersterer lässt sich ein Teamname festlegen und speichern oder ändern. Außerdem wird eine Liste mit allen Spielern im Team und den dazugehörigen Daten (Name, Position, Trikotnummer, etc.) angezeigt (siehe Abbildung 1).
In der aktuellen Version des Spielzeitenplaners wird sich zunächst auf nur ein Team beschränkt, da ein Großteil aller Fußballlehrenden nicht in mehreren Teams zugleich aktiv ist. Eine Unterstützung mehrerer Teams ist jedoch in zukünftigen Versionen bereits eingeplant. Darüber hinaus sind Persistenz- und Service-Schicht bereits im Hinblick auf die Verwendung mehrerer Teams entsprechend ausgelegt worden oder es sind nur kleinere Änderungen notwendig, um diese Funktionalität künftig unterstützen zu können.

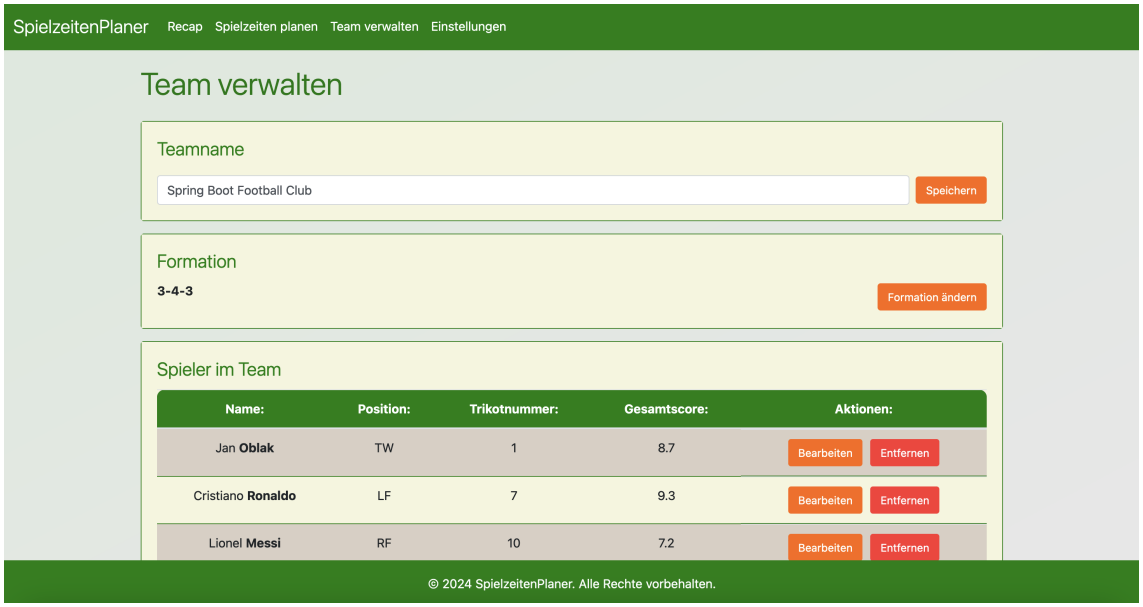

Abbildung 1: Screenshot der Team-Seite

Für jeden Spieler gibt es die Möglichkeit, diesen entweder zu bearbeiten oder zu löschen. Mit einem Klick auf den Löschen-Button wird der entsprechende Spieler gelöscht, durch den Klick auf den Bearbeiten-Button hingegen gelangt man zur Spieler-Seite.
Diese enthält sowohl die individuellen Spieler-Daten des aktuell ausgewählten Spielers wie auch eine Anzeige der Spieler-Scores. Hier können Vor-, Nachname, Position und Trikotnum-

mer des ausgewählten Spielers geändert werden. Wählt man auf der Team-Seite den Button zum Erstellen eines neuen Spielers, so wird die Spieler-Seite mit einem leeren Formular aufgerufen, sodass ein neuer Spieler erstellt und anschließend gespeichert werden kann.
Der Bereich Einstellungen enthält – wie der Name bereits suggeriert – einige grundsätzliche Einstellungen, die insbesondere für den Bereich des Planens der Spielzeiten von Bedeutung sind. Unter anderem besteht hier die Möglichkeit, eine eigene Formation zu erstellen. Dafür sind die Angabe eines Namens – zum Beispiel *4-4-2* oder *3-4-3* – sowie die Bezeichnungen der einzelnen Positionen notwendig (siehe Abbildung 2). Dabei werden diese geordnet von hinten nach vorne (Torwart, Verteidigung, Mittelfeld, Sturm) sowie von links nach rechts (linker Flügel, Zentrum, rechter Flügel) angegeben.

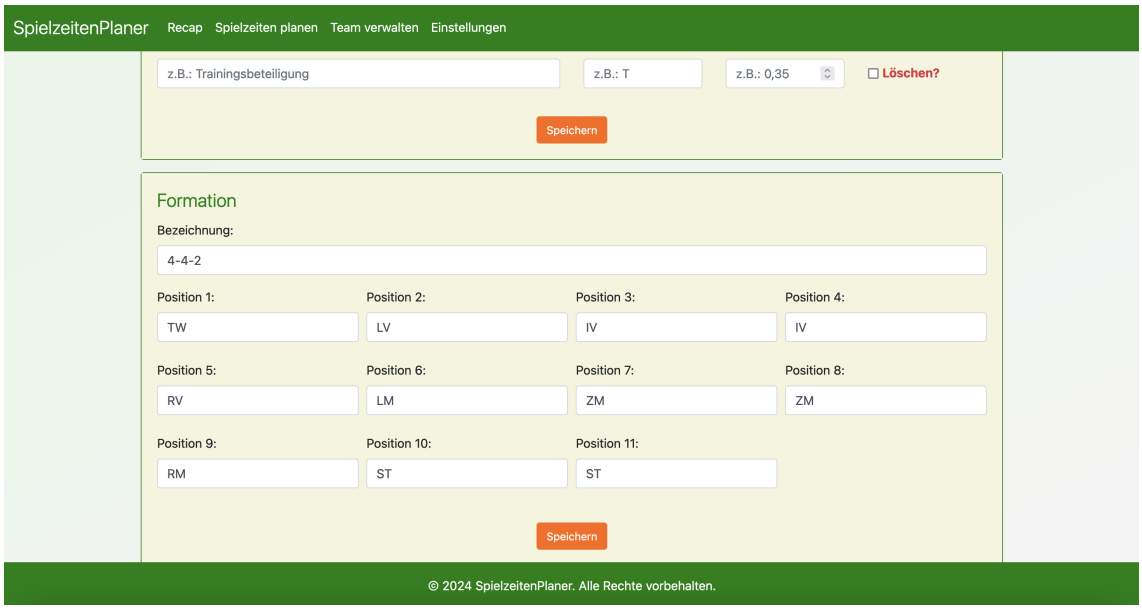

Abbildung 2: In den Einstellungen lässt sich die Formation bearbeiten/speichern

Über dem Abschnitt zur Formation befindet sich der Kriterien-Abschnitt. Hier können Kriterien erstellt, bearbeitet und gelöscht werden. Diese sind von zentraler Bedeutung bei der Bewertung der Spieler im Recap-Bereich. Für die Erstellung eines Kriteriums wird ein Name bzw. eine Bezeichnung, eine Abkürzung (ein bis zwei Buchstaben) und eine Gewichtung benötigt. Die Summe aller Gewichte sollte stets eins ergeben, ein einzelnes Gewicht im Bereich zwischen null und eins liegen. Die wohl gebräuchlichsten Kriterien sind zum Beispiel die Trainingsbeteiligung und die Leistung.
Schließlich gibt es noch die Scores-Einstellungen, die ganz oben auf der Seite zu finden sind. In diesem Bereich lässt sich der Zeitraum festlegen, auf dessen Grundlage die Scores für die einzelnen Kriterien berechnet werden. Für das Kriterium der Trainingsbeteiligung gibt es zusätzlich noch besondere Einstellungen: Zum einen kann zwischen einer kurzfristigen und langfristigen Trainingsbeteiligung unterschieden, zum anderen können spezifische Gewichte für die Kurz- und Langfrist festgelegt werden.

Folgendes Beispiel soll zur Verdeutlichung des Sachverhaltes herangezogen werden: Ein Trainerteam entscheidet sich dazu, dass die Trainingsbeteiligung 50 Prozent des Gesamt-Scores ausmachen soll. Innerhalb der Trainingsbeteiligung wird dann nochmals festgelegt, dass die letzten drei Wochen für die Kurzfrist herangezogen werden sollen und die letzten acht Wochen für die Langfrist. Da das Trainerteam etwas mehr Wert auf eine langfristige Teilnahme am Training legt, werden die Gewichte auf 0.4 für die Kurzfrist und 0.6 für die Langfrist festgelegt. Somit berechnet sich der Score für dieses Kriterium zu 60 Prozent aus der langfristigen Trainingsbeteiligung und zu 40 Prozent aus der Kurzfrist. So kann ein Spieler, der grundsätzlich immer am Training teilnimmt, ein kurzfristiges Fehlen aufgrund von Krankheit oder schulischer Verpflichtungen durch einen hohen Score in der Langfrist korrigieren.
Der dritte große Bereich der Anwendung ist das Recap. Im Wesentlichen geschieht hier Folgendes: Nach jedem Training wird eine Bewertung eines jeden Spielers zu jedem Kriterium vorgenommen. Die Bewertungen werden gespeichert und zur Ermittlung der Scores für jedes Kriterium sowie des Gesamtscores herangezogen. Letzterer wiederum ist von wesentlicher Relevanz bei der Planung der Spielzeiten – also der Spielminuten – für das kommende Spiel. Um zur Recap-Seite zu gelangen, werden in einem ersten Schritt all diejenigen Spieler ausgewählt, die am Training teilgenommen haben. Diese Information wird dann auf der eigentlichen Bewertungsseite genutzt, um eine Vorsortierung der Spieler vorzunehmen.

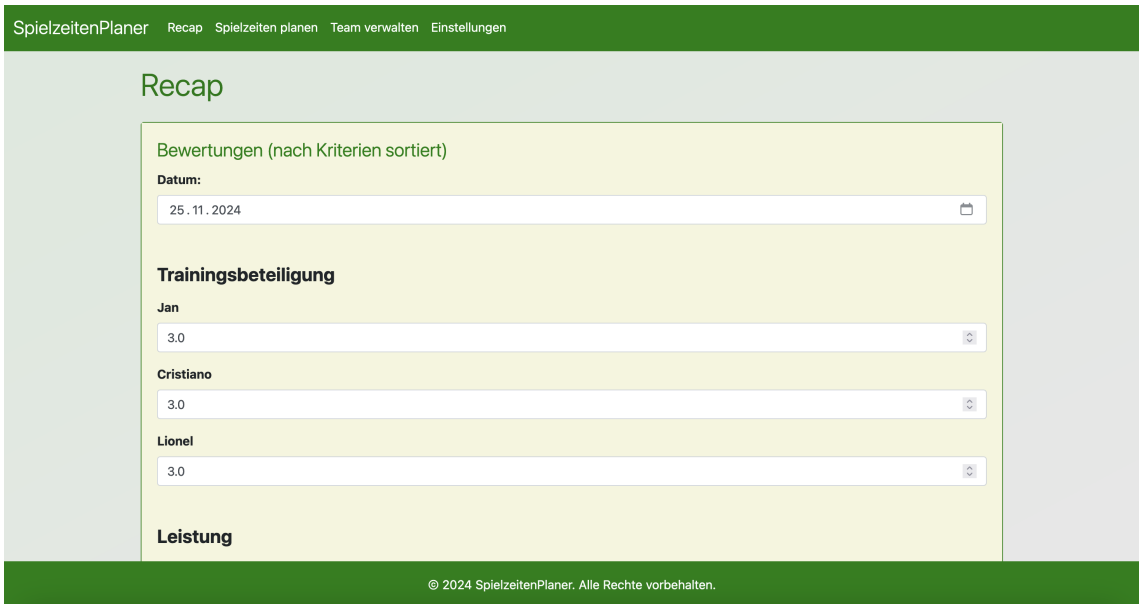

Abbildung 3: Auf der Recap-Seite wird jeder Spieler bewertet

Grundsätzlich ist die Recap-Seite nach den vorhandenen Kriterien gegliedert, das bedeutet, dass für jedes Kriterium eine Liste mit Spielern angezeigt wird, für die dann eine Bewertung abgegeben wird (siehe Abbildung 3). Die Bewertung erfolgt auf einer Skala von eins bis fünf, wobei eine Drei den Durchschnitt bildet, eine Eins eine deutlich unterdurchschnittliche

Bewertung darstellt und eine Fünf die bestmögliche Bewertung ist. Dementsprechend ist eine Vier als tendenziell überdurchschnittlich und eine Zwei als tendenziell unterdurchschnittlich zu betrachten.
Da der jeweilige Wert serverseitig als Double abgebildet wird, ist es den Nutzenden möglich, weitere Abstufungen vorzunehmen, beispielsweise eine 2.5 oder 4.5 zu vergeben. Für eine schnelle und effiziente Bewertung ist es jedoch ratsam, bei den Bewertungen eins, zwei, drei, vier und fünf zu bleiben. Für einen Spieler, der beim Training nicht anwesend ist, wird standardmäßig eine 0.0 vergeben. Die Null-Bewertungen werden dann serverseitig herausgefiltert und nicht gespeichert, da ein häufiges Fehlen sonst nicht nur den Score der Trainingsbeteiligung, sondern auch alle anderen Scores verringern würde, was einer gleich mehrfachen Abwertung gleichkäme.
Schließlich ist dann noch der Bereich der Spielzeitenplanung zu nennen. Hier können die Nutzenden vor einem Spiel planen, wie viele Minuten Spielzeit jeder einzelne Spieler basierend auf dem Gesamtscore erhalten soll. Die Spielzeitenplanung gestaltet sich als ein mehrstufiges Verfahren, durch das die Nutzenden vom Spielzeitenplaner geleitet werden.
In einem ersten Schritt werden zunächst alle Spieler ausgewählt, die für das kommende Spiel zur Verfügung stehen, die also nicht krank oder verletzt sind oder aufgrund von privaten Terminen und anderen Gründen abgesagt haben. Aus den verfügbaren Spielern wird dann in einem zweiten Schritt ermittelt, welche Spieler es in den Kader geschafft haben und welche nicht. Die vorgeschlagene Aufteilung kann dabei übernommen oder aber manuell durch die Nutzenden überarbeitet werden, sodass das Trainerteam die Kontrolle über die Spielzeitenplanung behält.

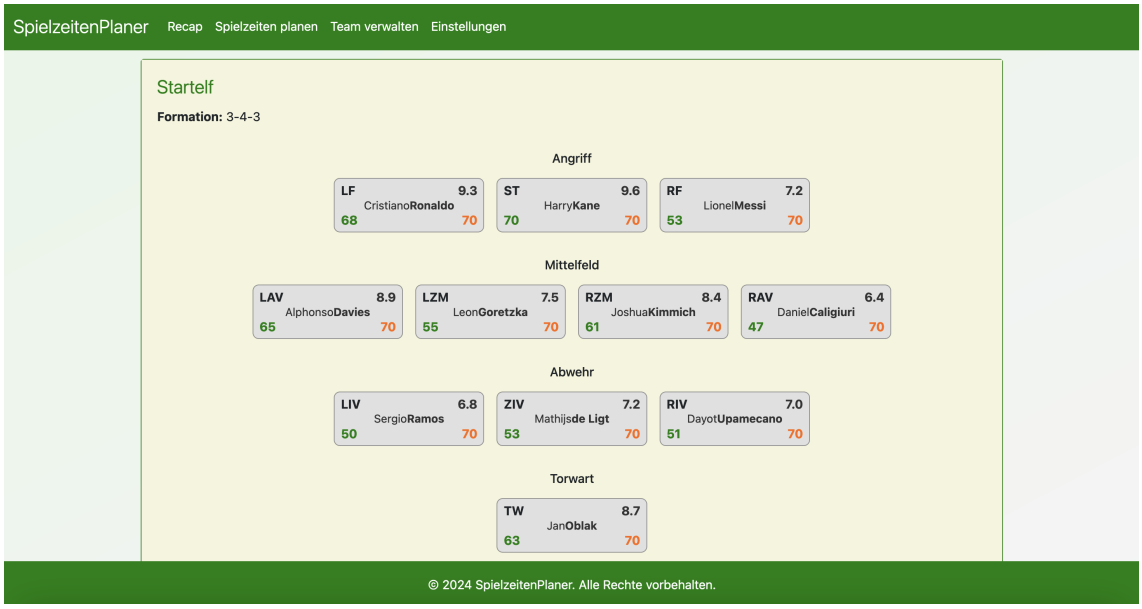

Abbildung 4: Startelf inklusive berechneter und geplanter Spielminuten

Nachdem der Kader feststeht und das entsprechende Formular durch die Benutzenden abgeschickt worden ist, wird in einem dritten Schritt aus dem Kader die Startelf bestimmt. Auch bei diesem Schritt können die Nutzenden Einfluss nehmen, indem Positionen getauscht oder Spieler von der Bank in die Startelf gesetzt werden. Ist die Startelf ermittelt, kommt es zum letzten Schritt der Spielzeitenplanung: dem Eintragen der Wechsel. Dieser letzte Planungsschritt ist essenziell für die Bestimmung der Spielzeiten, denn mit dem Feststehen der Wechsel bzw. der Wechsel-Zeitpunkte steht ebenfalls fest, welcher Spieler wie viele Minuten auf dem Platz steht (siehe Abbildung 4).
Wie bei den vorherigen Planungsschritten besitzen die Nutzenden auch hier die volle Kontrolle: sowohl Einwechsel- wie Auswechselspieler aber auch die konkrete Spielminute kann bestimmt werden. Die voraussichtliche Anzahl der Spielminuten für jeden einzelnen Spieler wird auf Basis der gespeicherten Wechsel berechnet und auf der Seite angezeigt. Außerdem wird vom Spielzeitenplaner eine erwartete Spielzeit berechnet. Diese errechnet sich maßgeblich aufgrund des Gesamtscores eines Spielers und stellt diejenige Spielzeit dar, die basierend auf den Bewertungen des entsprechenden Spielers als fair erachtet wird.
Nun können Nutzende so lange wie nötig Anpassungen vornehmen – das heißt neue Wechsel eintragen, Wechsel löschen oder die Wechselzeitpunkte anpassen – bis die voraussichtliche Anzahl der Spielminuten eines jeden Spielers ungefähr mit der Anzahl der erwarteten Spielminuten übereinstimmt. Ein weiteres Mal ist es dem Trainerteam selbst überlassen zu entscheiden, ob erwartete und voraussichtliche Spielzeit beispielsweise bis auf fünf, zehn oder fünfzehn Minuten übereinstimmen müssen, dennoch sollten die beiden Kennzahlen so eng wie möglich beieinander liegen, da große Abweichungen die Sinnhaftigkeit die Spielzeitenplanung infrage stellen. Ist schließlich eine faire Aufteilung der Spielzeiten unter allen beteiligten Akteuren gefunden, so kann die Spielzeitenplanung als erfolgreich abgeschlossen betrachtet und der Einsatz der Startelf sowie die Wechsel wie in der Anwendung geplant durchgeführt werden.
Die Entwicklung des Spielzeitenplaners befindet sich zum aktuellen Zeitpunkt noch in der Anfangsphase. Dennoch konnten erste Features der Anwendung bereits mit einigen wenigen Fußballlehrenden getestet werden. Die Rückmeldungen waren dabei durchaus positiv. Durch das regelmäßige Bewerten der Spieler nach dem Training konnten die Eindrücke der Fußballlehrenden zeitnah festgehalten werden. Die Akkumulation sämtlicher Bewertungen eines Spielers spiegelte sich im berechneten Gesamt-Score wieder, der den Trainerteams einen konkreten Anhaltspunkt gab, wie Trainingsbeteiligung, Leistung und Verhalten in den vorangegangenen Wochen einzuordnen ist.
Das Nutzen der `Recap`-Funktionalität sowie die Spielzeitenplanung machte den Bewertungs- und Planungsprozess gewissermaßen sichtbar und half dabei eine möglichst faire Aufteilung der Spielminuten unter den verschiedenen Spielern zu finden. Außerdem lieferte die Anwendung den Nutzenden eine Datengrundlage, auf Basis derer sie Entscheidungen treffen und schließlich auch begründet vertreten konnten. Erste Gespräche mit Spielern bezüglich der Spielzeit verliefen positiv, da diesen transparent kommuniziert werden konnte, warum eine gewisse Spielzeit gerechtfertigt ist, aber gleichzeitig auch aufgezeigt wurde, in welchen Bereichen sie sich verbessern können, um in Zukunft auf mehr Spielzeit zu kommen.
Während der Erprobung der Anwendung konnten gleichzeitig aber auch noch Verbesserungspotenziale ermittelt werden. So wäre es von Vorteil, wenn die Bedienung im Bereich

der Spielzeitenplanung noch etwas intuitiver gestaltet werden könnte, zum Beispiel durch eine `Drag and Drop`-Funktionalität bei der Festlegung der Startelf. Darüber hinaus könnte die Spielzeitenplanung noch effektiver gestaltet werden, indem automatisch Wechsel vorgeschlagen werden, die dann von den Nutzenden direkt übernommen werden können, wenn gewünscht.
Schließlich kann und sollte noch darüber nachgedacht werden, wie `Recaps` noch effizienter durchgeführt werden können. Angenommen eine Mannschaft besteht aus 25 Spielern und es werden vier Kriterien zur Bewertung herangezogen, so ergeben sich für ein Trainerteam bereits 100 kleine Bewertungsschritte über die nach dem Training nachgedacht werden muss. Denkbar wäre hier, Bewertungen nicht nach jedem Training, sondern beispielsweise ein Mal wöchentlich durchzuführen oder aber nicht alle Spieler zu bewerten, sondern nur Punkte für die bei einem Training positiv wie negativ herausragenden Spieler zu vergeben. Eine solche Bewertungsstrategie würde jedoch zu Lasten der Objektivität und Genauigkeit der Bewertungen führen. Hier muss in Zukunft ein geeignetes Mittelmaß zwischen Effizient Genauigkeit gefunden werden.

4 Die testgetriebene Entwicklung des Spielzeitenplaners

Nachdem sowohl die theoretischen Grundlagen wie auch der Aufbau und die Funktionsweise des Spielzeitenplaners geklärt und erläutert worden sind, kann nun im folgenden Kapitel näher auf die testgetriebene Entwicklung des Spielzeitenplaners eingegangen werden. Dabei soll für jede Architekturschicht dokumentiert werden, wie dort testgetrieben entwickelt worden ist. Außerdem sollen konkrete Beispiele aus dem Projekt gezeigt, beschrieben und erläutert werden.

4.1 Das systematische Testen der WebPages

Im Folgenden soll sich zunächst auf die Entwicklung der Benutzeroberfläche fokussiert werden. Das Testing des Web-Interfaces ist grundsätzlich zweigeteilt: Zum einen werden die konkret ausgelieferten Webseiten mit ihren Inhalten und HTML-Elementen überprüft, zum anderen gibt es spezielle Testklassen für die Controller, mithilfe derer eine grundsätzliche Funktionsüberprüfung der Web-Steuereinheiten erfolgt. Die Testklassen für Ersteres sind im Verzeichnis `de/bathesis/spielzeitenplaner/templates` zu finden. Eine solche Testklasse ist grundsätzlich folgendermaßen aufgebaut:

```
@WebMvcTest(Controller.class)
class PageTest {
    @Autowired
    MockMvc mvc;
    ...
    Document page;
    ...
    @BeforeEach
    void setUpPage() throws Exception {
        ...
        page = RequestHelper
                  .performGetAndParseWithJSoup(mvc, "/path");
    }
    ...
}
```

Durch die Verwendung der `WebMvcTest`-Annotation wird ein spezieller Testkontext initialisiert, der sich nur auf das Laden und Konfigurieren von Komponenten der Web-Schicht konzentriert [VMw24]. Darüber hinaus wird in Klammern notiert, welche spezifische Controller-Klasse für den Test benötigt wird, sodass nur diese für den Test geladen und bereitgestellt wird. Überlicherweise enthält jede WebPage-Testklasse eine mit `BeforeEach` annotierte `setUp`-Methode, die vor jedem Test ausgeführt wird. Da das Vorgehen vor

jedem Test gleich ist, bietet sich die Extraktion dieser Logik an, um die einzelnen Tests übersichtlicher und wartbarer zu gestalten.
Das Ziel der `setUp`-Methode ist es, mithilfe des MockMvc eine Anfrage über einen bestimmten Pfad zu simulieren, das Ergebnis mit Jsoup zu parsen (siehe `RequestHelper.java`) und schließlich in einer Instanzvariable – hier `page` genannt – zu speichern. Auf diese Weise kann ein Abbild derjenigen Seite erzeugt werden, die im Browser zurückgegeben wird. Diese kann dann detaillierten Testungen unterzogen werden.
Ein wichtiger Bestandteil dabei bildet die Java-Bibliothek Jsoup. Sie wurde für das Arbeiten mit HTML entwickelt und ermöglicht daher sowohl das Parsen, wie auch Extrahieren, Manipulieren oder Korrigieren von HTML-Dokumenten [Hed24]. Für die dieser Arbeit zugrunde liegenden Testungen werden vor allem die erstgenannten Funktionen – also das Parsen und Extrahieren von HTML bzw. HTML-Schnipseln – benötigt.
Wie Jsoups `parse`-Funktion in diesem Projekt verwendet wurde, ist zuvor bereits bei den Ausführungen zur `setUp`-Methode erwähnt worden, das Extrahieren von HTML-Schnipseln lässt sich wie folgt realisieren: Mithilfe der `select`-Funktion wird aus einem HTML-Dokument oder einem HTML-Element ein HTML-Schnipsel herausgelöst, das den Anforderungen einer `CSS-Query` entspricht, die der Funktion als Parameter übergeben worden ist.
Über die Komplexität der `CSS-Query` kann gesteuert werden, wie detailliert die Struktur der HTML-Seite getestet werden soll. So kann mit `‘‘h2’’` zum Beispiel einfach eine Überschrift zweiter Ebene herausgefiltert werden, mit `‘‘.card .card-body h2.card-title’’` hingegen präziser nach einer `H2` gesucht werden, die die Klasse `card-title` besitzt und sich innerhalb des Bodies einer `Card` befindet.
Das Testing jeder einzelnen Webseite erfolgt im Wesentlichen nach ein und demselben Konzept, das im Folgenden geschildert werden soll. Zunächst werden die sogenannten Essentials – oder auch Basics – getestet, es wird also überprüft, ob die Seite die korrekte Überschrift besitzt sowie ob grundlegende Elemente – wie die Navigationsleiste und der Footer – angezeigt werden. Im Anschluss erfolgt eine detaillierte Überprüfung der einzelnen Bereiche der jeweiligen Seite und ihrer Elemente, also beispielsweise, ob der Bereich Teamname auf der Teamseite korrekt angezeigt wird oder ob das Formular zum Ändern des Teamnamens korrekt angezeigt wird (siehe `TeamPageTest.java`).
Abschließend wird getestet, ob entsprechende Daten, die durch den Controller bzw. das Model bereitgestellt werden, wie beabsichtigt mithilfe von Thymeleaf in die Seite gerendert werden. Durch diesen strukturierten Ansatz wird gewährleistet, dass alle wesentlichen Elemente und Funktionen der Webseite umfassend getestet werden.
Mit Beginn der Arbeiten an diesem Projekt wurden zunächst sämtliche HTML-Prototypen bzw. die ihnen zugrunde liegenden HTML-Templates testgetrieben entwickelt. Ein Beispiel dafür, auf das sich im Folgenden bezogen werden soll, ist die sogenannte `PlayerPage`, der das HTML-Template `player.html` als Basis dient, und die zugehörige Testklasse `PlayerPageTest.java`.

Ein einfacher Test, durch den die Anwesenheit der korrekten H1-Überschrift erzwungen werden kann, ist `test_01`:

```
@Test
...
void test_01() {
    String expectedTitle = "Spieler bearbeiten/hinzufügen";
    String pageTitle = RequestHelper
              .extractTextFrom(playerPage, "h1");
    assertThat(pageTitle).isEqualTo(expectedTitle);
}
```

Hier wird mittels der Utilities-Klasse `RequestHelper.java` der Text des H1-Tags der `PlayerPage` extrahiert. Dann wird überprüft, ob dieser mit dem erwarteten Text übereinstimmt. Ist dies nicht der Fall, schlägt der Test fehl. Wenn auf der Seite überhaupt kein solches Element vorhanden ist, schlägt der Test ebenfalls fehl, denn dann ist das Ergebnis des Parsens durch Jsoup schlichtweg ein leerer String. Auf diese Weise kann also gewährleistet werden, dass auf einer Seite die korrekte Überschrift angezeigt wird.
Für die Navigationsleiste wurde auf der `PlayerPage` – wie auch auf allen anderen Seiten – getestet, ob diese vorhanden ist:

```
@Test
...
void test_02() {
    Elements navbar = RequestHelper.extractFrom(playerPage, "nav");
    assertThat(navbar).isNotEmpty();
}
```

Dafür wird das `nav`-Element der Seite extrahiert und durch eine geeignete Assertion sichergestellt, dass diese nicht leer ist. Analog zur Navigationsleiste kann ebenfalls mit dem Footer verfahren werden. Die eigentliche Überprüfung, ob die `Essentials` den Anforderungen entsprechen, ist ausgelagert und in der `FragmentsTest.java` unter `src/test/java/de/bathesis/spielzeitenplaner/templates/fragments` zu finden.
Dort wird im Detail getestet, ob die einzelnen Elemente korrekt strukturiert sind, sie den gewünschten Text enthalten und die erwarteten Funktionen unterstützen – im Falle der Navigationsleiste beispielsweise, ob die Links zu den Unterseiten `Team`, `Recap`, `Spielzeiten planen` sowie `Einstellungen` funktionieren. Das entsprechende Fragment – also der wiederkehrende HTML-Schnipsel einer Webseite, der ausgelagert worden ist – ist in der `basics.html` unter `src/main/resources/templates/fragments` gespeichert. Dort wird es mittels des `th:fragment`-Tags als Solches gekennzeichnet und benannt.

Ist dies geschehen, so kann es im Folgenden dann wiederverwendet werden, indem es mithilfe der Nutzung des `th-replace`-Tags im HTML-Template und Thymeleaf in die entsprechende Seite hineingerendert wird. Ein konkretes Beispiel für die Verwendung eines `Thymeleaf-Fragments` in diesem Projekt ist der Footer:

```
<div th:fragment="footer">
    <footer class="footer fixed-bottom">
        <p>
            &copy; 2024 SpielzeitenPlaner. Alle Rechte vorbehalten.
        </p>
    </footer>
</div>
```

Dieser ist mit dem `th:fragment`-Tag versehen und als `‘‘footer’’`, benannt. Innerhalb der `welcome.html` kann er dann einfach mithilfe des einzeiligen Codeschnipsels `<div th:replace=‘‘~{fragments/basics :: footer}’’></div>` eingefügt werden.
Durch diese Herangehensweise müssen die Funktionalitäten der `Essentials` nicht auf jeder Seite explizit getestet werden – bei zehn verschiedenen Seiten wären das immerhin 30 Tests, die aber stets nur das Gleiche testen würden – sondern lediglich das Vorhandensein gewisser Elemente. Wenn sich nun eine Beschriftung ändert, die Struktur angepasst werden soll oder ein neuer Bereich – zum Beispiel `Statistik` – hinzukommt, müssen die entsprechenden Tests nur an einer Stelle angepasst werden, die Testklassen für die einzelnen Webseiten bleiben unberührt, da hier – wie zuvor bereits beschrieben – lediglich das Vorhandensein geprüft wird.
Navigationsleiste und Footer sind im Rahmen des Spielzeitenplaners als feste Bestandteile jeder Seite eingeplant, um Letzteren eine Rahmung zu geben und Nutzenden die Navigation durch die einzelnen Bereiche der Anwendung zu erleichtern. Für den Fall, dass Navigationsleiste oder Footer jedoch komplett entfernt werden sollen, kann über die Verwendung des `Thymeleaf Layout Dialect` nachgedacht werden. Dieser ermöglicht es, Layout-Templates zu erstellen und zu definieren, die dann wiederum von anderen Templates verwendet werden können [Bor24].
So kann die grundsätzliche Struktur einer Seite von ihrem konkreten Inhalt getrennt werden, was die Modularisierbarkeit fördert und die Wiederverwendbarkeit erhöht. Für das konkrete Testing bedeutet dies, dass ein Layout-Template ein Mal gesondert getestet wird, das Testing der einzelnen Seiten sich vollkommen auf den Inhalt konzentrieren kann.
Doch wie bereits zuvor beschrieben ist nicht nur das Testen der `Essentials` ein wichtiger Bestandteil der `WebPages`-Tests, sondern auch die Überprüfung der einzelnen Bereiche einer Seite und ihrer Elemente. Den Kern der Spieler-Seite bilden die Bereiche `Spieler-Daten` und `Spieler-Scores`. Durch das Entwickeln dreier Tests, die stets einen anderen Aspekt überprüfen, kann ein solcher Bereich im Produktivcode erzwungen und Schritt für Schritt geformt werden, bis er schließlich seine gegenwärtige Form erreicht hat.

In einem ersten Schritt kann ein Test geschrieben werden, der die grundsätzliche Struktur des Bereichs festlegt:

```
@Test
...
void test_04() {
    String expectedCardTitle = "Spieler-Daten";
    List<String> expectedAttributes = List.of(
        "Vorname", "Nachname", "Trikotnummer", "Position"
    );

    String cardTitle = RequestHelper.extractTextFrom(
        playerPage, ".card.player-data .card-body .card-title"
    );
    String playerInfo = RequestHelper.extractTextFrom(
        playerPage, ".card.player-data .card-body .player-info"
    );

    assertThat(cardTitle).isEqualTo(expectedCardTitle);
    assertThat(playerInfo).contains(expectedAttributes);
}
```

Um diesen Test bestehen zu lassen, ist die Etablierung der Grundstruktur in der `player.html` erforderlich, wie dem Commit `5e52549` im Detail entnommen werden kann.
In einem zweiten Schritt kann dann die Anwesenheit des Spieler-Formulars erzwungen werden, das zum einen dazu dient, die Daten eines Spielers anzuzeigen, zum anderen aber auch das Ändern bereits gespeicherter Informationen unterstützt. Wie `test_05` der `PlayerPageTest.java` zu entnehmen ist, wird dort geprüft, ob es ein Formular mit der ID `playerForm` gibt sowie ob dieses die für die Verarbeitung der Daten notwendigen Elemente – wie Input-, Label-Felder und einen Button – besitzt. Außerdem wird an dieser Stelle ebenfalls gefordert, dass das Formular die `Post`-Methode verwenden und die Anfrage über `/team/savePlayer` verschickt werden soll sowie einen Button vom Typen `submit` enthalten muss, wie den folgenden Zeilen zu entnehmen ist:

```
Elements playerForm = RequestHelper.extractFrom(playerPage,
    "form#playerForm[method=\"post\"][action=\"/team/savePlayer\"]"
);
String buttonLabel = RequestHelper.extractTextFrom(playerForm,
    "button[type=\"submit\"]"
);
```

Durch solche spezifischen `CSS-Queries` können präzise Anforderungen an das Formular gestellt werden und sichergestellt werden, dass die notwendigen Funktionalitäten korrekt

implementiert werden. Im hier vorliegenden Fall der Änderung von Spielerdaten kann bzw. muss parallel im Controller-Testing eine entsprechende Route und Methodenunterstützung implementiert werden (siehe Kapitel 3.2).
Nachdem nun die grundlegende Struktur des Spielerdaten-Bereichs etabliert und das Spieler-Formular vorhanden ist, kann abschließend in einem dritten Schritt die korrekte Anzeige der konkreten Spieler-Daten überprüft werden:

```
@Test
...
void test_07() {
    List<String> expectedValues = List.of(
        Integer.toString(player.getId()),
        player.getFirstName(), player.getLastName(),
        player.getPosition(),
        Integer.toString(player.getJerseyNumber())
    );

    List<String> values = RequestHelper.extractFrom(
        playerPage, "form#playerForm input"
    ).eachAttr("value");

    assertThat(values)
        .containsExactlyInAnyOrderElementsOf(expectedValues);
}
```

Die `expectedValues` entsprechen den Attributen des `player`, der als Instanzvariable in der Testklasse definiert ist. Mithilfe des `RequestHelpers` und Jsoups `eachAttr`-Methode können die tatsächlich angezeigten Werte der Input-Felder in einer Liste gespeichert und anschließend überprüft werden. Damit der Test jedoch ordnungsgemäß funktioniert, muss der mit `@MockBean` annotierte `playerService` noch konfiguriert werden. Mithilfe von `when(playerService.loadPlayer(player.getId())).thenReturn(player)` geschieht dies innerhalb der `setUp`-Methode entsprechend.
Um den Test nun bestehen zu lassen und die gewünschte Funktionalität zu implementieren, muss Folgendes geschehen: Die Spieler-Daten, die vom entsprechenden Service an den Controller weitergegeben und durch Letzteren im Model bereitgestellt werden, müssen mit dem jeweiligen Input-Feld verknüpft werden. Die Existenz jener Input-Felder wurde bereits im vorherigen Test gefordert, nicht aber ihr spezifischer Inhalt. Mithilfe von Thymeleaf lassen sich die entsprechenden Daten komfortabel in das jeweilige Template bzw. die jeweilige Seite hineinrendern:

```
<form id="playerForm" ... th:object="${playerForm}">
    ...
```

```
        <input type="text" ... th:field="*{firstName}" ...>
        ...
        <input type="text" ... th:field="*{lastName}" ...>
        ...
    </form>
```

Hier wurde `th:object` verwendet, um das Formular mit dem Formular-Objekt `playerForm` aus dem Model zu paaren. Thymeleafs `th:field` bindet außerdem jedes einzelne Input-Feld an das entsprechende Attribut, also `firstName`, `lastName`, `position` und `jerseyNumber`. Dies ist nicht nur für eine spätere Validierung und der damit verbundenen Ausgabe der Formular-Fehler von Vorteil und notwendig, sondern sorgt darüber hinaus dafür, dass die Daten des aktuellen Spielers auf der `PlayerPage` angezeigt werden.
Zusammengefasst lässt sich noch einmal sagen, dass sich durch gezieltes, umfangreiches Testing unterschiedlicher Aspekte die wesentlichen Funktionalitäten und Bausteine einer Webseite testgetrieben entwickeln lassen: Von der grundlegenden Strukur einer Seite bis hin zum Aufbau ihrer spezifischen Bereiche, die ihnen innewohnenden Elemente zur Datenerfassung, Interaktion und Kommunikation – also zum Beispiel Formulare – sowie die Anzeige konkreter und gespeicherter Daten und Inhalte.
Diese grundlegenden Testprinzipien lassen sich auf die Entwicklung jeder einzelnen Webseite übertragen und an ihre spezifischen Anforderungen anpassen, beispielsweise bei der Spielerbewertung auf der `RecapPage` (siehe `RecapPageTest.java`) oder der Anzeige und Bestätigung des Kaders in der Spielzeitenplanung (siehe `KaderPageTest.java`).

4.2 Controller-Tests: Das Überprüfen der Hauptaufgaben der Web-Steuereinheiten

Eng verbunden mit mit dem Testing der `WebPages` ist auch die Überprüfung der Web-Steuereinheiten, also der entsprechenden Controller. Wie bereits im vorangegangenen Kapitel erwähnt, können konkrete Inhalte nur in die Seite eingefügt werden, wenn diese im Model vorhanden sind. Dies stellt unter anderem eine Aufgabe des Controllers dar. Doch neben der Datenaufbereitung und -bereitstellung ist er außerdem auch für die Verarbeitung von Benutzeranfragen, das Verwalten der Anwendungslogik, die Fehlerbehandlung und das grundlegende Routing zuständig. Alle soeben genannten Aufgaben sollen in diesem Kapitel unter dem Aspekt der testgetriebenen Entwicklung des Spielzeitenplaners eingehend beleuchtet werden.
Begonnen werden soll mit dem letzten Punkt – dem grundlegenden Routing, das den Startpunkt sämtlicher Controller-Tests darstellt. Denn wie bereits in Kapitel 3.1 festgehalten, sind sämtliche Testungen der Elemente und Strukturen einer ausgelieferten Webseite unter Gebrauch eines `MockMvc`-Objektes nur möglich, wenn zuvor ein entsprechendes Routing etabliert und ein geeignetes Request-Mapping stattgefunden hat.
Der wohl simpelste Test zur Überprüfung einer Route kann im Commit 3aec5fe betrachtet werden. Alles, was zum Bestehen des Tests zur Erreichbarkeit der Team-Seite benötigt

wird, ist eine mit `@Controller` annotierte Klasse und eine mit `@GetMapping(‘‘/team’’)` beschriftete Handler-Methode, die wiederum den Namen eines Templates zurückgibt – in diesem Fall die `team.html`, die im Verzeichnis `src/main/ressources/templates` existieren muss.
Im weiteren Verlauf der Entwicklung des Projektes ist die `team()`-Handler-Methode dann in einen eigens für diesen Bereich angelegten `TeamController` ausgelagert worden. Des Weiteren ist mit der Einführung des `TeamService` das gewünschte Verhalten – hier die Rückgabe des Team- Objektes, das den Teamnamen enthält – gemockt und eine Überprüfung des `view`-Namen ergänzt worden, sodass sich final der folgende Testablauf ergibt:

```
@Test
...
void test_01() throws Exception {
    when(teamService.load())
      .thenReturn(new Team(142, "Holstein Kiel"));

    RequestHelper.performGet(mvc, "/team")
                 .andExpect(status().isOk())
                 .andExpect(view().name("team/team"));
}
```

Sobald die entsprechende Seite erreichbar ist, kann mit der testgetriebenen Entwicklung ihrer Struktur – wie in Kapitel 3.1 verdeutlicht – begonnen werden. Neben dem grundlegenden Routing ist der Controller aber ebenfalls für die Aufbereitung und Bereitstellung der durch die Service-Schicht zur Verfügung gestellten Daten verantwortlich. Eine Überprüfung dieser Verantwortlichkeit lässt sich wie folgt realisieren:

```
@Test
@DisplayName("Das Model für die Team-Seite ist korrekt befüllt.")
void test_02() throws Exception {
    // Erstellen eines Team-Objektes zu Testzwecken
    // Mocking des Team-Services

    // Erstellen einiger Test-Spieler
    // Mocking des Player-Services

    // Erstellen der Total-Scores & Mocking

    RequestHelper.performGet(mvc, "/team")
                 .andExpect(model().attribute("teamForm", teamForm))
                 .andExpect(model().attribute("players", players))
                 .andExpect(model().attribute(
```

```
                    "totalScores", totalScores
                ));
    }
```

Aus Gründen der Übersichtlichkeit ist hier auf eine vollständige Darstellung des Tests verzichtet worden, der genaue Wortlaut bzw. der genaue Code ist der `TeamControllerTest.java` zu entnehmen. Den Kern dieses Tests bilden der mithilfe des `MockMvcs` simulierte Get-Request und die spezifische Überprüfung der HTTP-Antwort durch `andExpect`. Durch das zuvor konfigurierte Mocking der Services erhalten Entwickelnde die Kontrolle über die zur Verfügung gestellten Daten. Unter Verwendung von `model().attribute(‘‘attributeName’’, ‘‘expectedValue’’)` kann dann gezielt gesteuert werden, welche Werte `teamForm`, `players` und `totalScores` annehmen sollen, denn nur wenn sie dem `expectedValue` entsprechen, wird der gegebene Test bestehen. Für `players` beispielsweise bedeutet das, dass das Attribut genau diejenige Liste von Spielern als Wert annehmen muss, die durch die `loadPlayers`-Methode des `PlayerService` zurückgegeben wird.
Des Weiteren wird im Rahmen der Entwicklung des Spielzeitenplaners zwischen Formular- und Domänen-Objekten unterschieden, eine Mapper-Klasse ist für die entsprechende Konvertierung von der einen in die andere Darstellung verantwortlich. Für den Teamnamen bedeutet dies, dass dieser aus dem Team-Objekt in die `TeamForm` überführt wird, sodass er schließlich im Model bereitgestellt werden kann. Diese Aufteilung fördert die Trennung der Verantwortlichkeiten – Web-UI von Geschäftslogik – und erhöht damit auch die Wartbarkeit des Codes, da zukünftige Änderungen am Team-Formular von der Geschäftslogik losgelöst durchgeführt werden können.
Wie bisher gezeigt fokussieren sich die beiden vorangegangenen Tests auf das Anfordern von Ressourcen auf dem Server mittels eines GET-Requests und die damit verbundene Datenaufbereitung und -bereitstellung. Im Gegensatz dazu steht der POST-Request, der für das Erstellen oder Verändern einer Ressource auf dem Server verantwortlich ist.
Konkret für die `TeamPage` bedeutet dies, dass nicht nur das Aufrufen der Team-Seite sowie die Anzeige des Teamnamens und der Spieler im Team eine wichtige Funktion darstellt, die der Spielzeitenplaner gewährleisten sollte, sondern auch die Möglichkeit, den Teamnamen zu bearbeiten und zu ändern, Spielerinformationen zu aktualisieren oder sich nicht mehr im Team befindliche Akteure zu löschen. Für die Unterstützung manipulierender Benutzeranfragen ist auch hier zunächst einmal die Etablierung einer grundlegenden Route vonnöten:

```
@Test
@DisplayName("Es werden Post-Requests über /team/teamname akzeptiert.")
void test_05() throws Exception {
    mvc.perform(post("/team/teamname").param("name", "Spring Boot FC"))
        .andExpect(status().is3xxRedirection())
        .andExpect(view().name("redirect:/team"));
}
```

Das grundsätzliche Prinzip solcher Anfragen in diesem Projekt ist es, für jeden solcher Requests eine individuelle Route anzulegen, die dann von einer speziellen Handler-Methode eines zuständigen Controllers verarbeitet wird. Im Anschluss an eine erfolgreiche Anfrage wird dann auf eine Get-Route weitergeleitet. Im Falle der `TeamPage` bedeutet dies Folgendes: Analog zum `GetMapping` wird hier parallel zum Testcode eine mit `@PostMapping(''/teamname'')` annotierte Handler-Methode geschrieben, die `''redirect:/team''` retourniert. Letzteres stellt sicher, dass nach abgeschlossener Verarbeitung zur Team-Seite umgeleitet wird.
In einem zweiten Schritt muss dann sichergestellt werden, dass die zuständige Service-Methode korrekt aufgerufen wird:

```
@Test
...
void test_06() throws Exception {
    String teamName = "Spring Boot FC";
    TeamForm teamForm = new TeamForm();
    teamForm.setName(teamName);

    mvc.perform(post("/team/teamname").param("name", teamName));

    verify(teamService).save(teamForm);
}
```

Eine wie zuvor beschriebene Überprüfung kann mithilfe von `Mockitos verify`-Methode realisiert werden. Durch die letzte Zeile des `test_06` kann zum einen sichergestellt werden, dass die `save`-Methode des `TeamService` – also die für das Speichern des Teamnamens zuständige Service-Methode – durch den `TeamController` aufgerufen wird, zum anderen aber auch überprüft werden, ob diese mit dem richtigen Parameter aufgerufen wird. Letzteres ist besonders wichtig bei schichtbasierten Softwarearchitekturen – wie der Onion-Architektur, um zu gewährleisten, dass Objekte zwischen den Schichten korrekt übergeben werden.
Für die `changeTeamName`-Methode des `TeamControllers` bedeutet dies konkret, dass gewährleistet wird, dass der Teamname korrekt in das `TeamForm`-Objekt integriert wird. Dies geschieht üblicherweise automatisch durch Spring, indem eine Instanz des `TeamForm`-Objekts als Parameter der Methode verwendet wird, sodass die Werte der Anfrage automatisch an das Formular-Objekt gebunden werden. Anschließend muss die `teamForm` als Parameter an die `save`-Methode des `TeamServices` übergeben werden, der sich dann wiederum um die Konvertierung zum Domänen-Objekt und das Speichern kümmern kann.
Doch neben Routing und Verwalten der Anwendungslogik ist ein Controller – insbesondere bei POST-Requests – auch noch für die Validierung der Benutzereingaben zuständig. Beim Teamnamen muss daher überprüft werden, dass das entsprechende Input-Feld nicht leer bzw. blank ist und eine Länge von 100 Zeichen nicht überschreitet, wie der `TeamForm.java` zu entnehmen ist.
Für ein weiteres, ein wenig umfangreicheres Beispiel kann abermals die `PlayerPage` herangezogen werden – genauer gesagt das Formular zur Verwaltung der Spieler-Daten: Hier

müssen Vorname, Nachname, Position und Trikotnummer geeignet validiert werden. Folgender Test der `TeamControllerTest.java` soll veranschaulichen, wie eine solche Validierung schrittweise erzwungen bzw. kontrolliert werden kann:

```
@Test
...
void test_12() throws Exception {
    String response = mvc.perform(post("/team/savePlayer")
                                    .param("firstName", "")
                                    .param("lastName", "")
                                    .param("position", ""))
                         .andExpect(model()
                             .attributeErrorCount("playerForm", 5))
                         .andReturn()
                             .getResponse().getContentAsString();
    Elements errors = Jsoup.parse(response).select(".error");
    assertThat(errors).hasSize(4);
}
```

Der hier gezeigte Test ist nach folgendem Prinzip aufgebaut: Zunächst wird wieder ein POST-Request mit geeigneten Parametern simuliert, ehe der `AttributeErrorCount` des zu betrachtenden Objektes – in diesem Fall die `PlayerForm` – überprüft wird. Im Anschluss wird dann noch kontrolliert, ob potenzielle Fehlermeldungen auch tatsächlich auf der Seite angezeigt werden. Dazu wird die durch die Anfrage zurückgegebene Antwort mit `Jsoup` geparst sowie alle `div`-Container mit der Klasse `error` extrahiert und im Bezug auf ihre Größe inspiziert.
Den Basisfall stellt hier ein leeres Spieler-Formular dar. Der Anfrage werden somit jeweils ein leerer String als Parameter für den Vornamen, Nachnamen und die Position hinzugefügt. Damit der `AttributeErrorCount` für die `PlayerForm` nun die erwartete Größe besitzt, müssen die folgenden Anpassungen im Produktivcode vorgenommen werden: In der entsprechenden Handler-Methode des Controllers muss die `PlayerForm` mit `@Valid` annotiert werden, damit die Benutzereingabe auch wirklich validiert wird. Des Weiteren muss direkt nach der `PlayerForm` ein weiterer Parameter vom Typ `BindingResult` eingefügt werden, in dem das Ergebnis der auf die Eingabe angewendeten Validierung gespeichert und an das Formular-Objekt gebunden wird.
Diese Schritte alleine reichen jedoch nicht aus, um den Test bestehen zu lassen. Damit das `BindingResult` nicht einfach leer bleibt, muss die eigentliche Validierung noch konfiguriert werden. Dies geschieht innerhalb der Formular-Klasse `PlayerForm.java` mithilfe geeigneter Validierungs-Annotationen. Dort wird festgelegt, dass Attribute wie der Vor- und Nachname oder die Position nicht blank sein dürfen sowie Letztere zwischen einem und fünf Zeichen lang sein und die Trikotnummer zwischen eins und 99 liegen muss.
Im oben gezeigten Basisfall mit einem leeren Spieler-Formular werden also bereits sämtliche `@NotBlank`-Annotation geprüft sowie die `@Size`-Annotation der Position und die

`@NotNull`-Annotation der Trikotnummer. Im Anschluss wird dann noch in der Variable `error` gespeichert, ob für jedes Eingabefeld ein entsprechender Fehler-Container auf der Seite vorhanden ist und ein Fehler angezeigt wird – die beiden Fehler bezüglich der Position werden dabei in demselben Container angezeigt, da sie sich jeweils auf dasselbe Attribut beziehen.
Im weiteren Verlauf der Entwicklung der Validierungsfunktionalitäten sind dann noch gezielt Anfragen simuliert worden, die die noch fehlenden Möglichkeiten abdecken: Eine Benutzeranfrage mit einer Trikotnummer kleiner eins sowie einer Nummer größer 99, wie `test_13()` sowie `test_14()` zu entnehmen ist. Diese gewährleisten, dass sowohl die `Max`- wie auch die `Min`-Validierung der `jerseyNumber` ordnungsgemäß implementiert sind.
Auf die hier gezeigte Weise kann also die Validierung von Benutzereingaben und die Fehlerausgabe testgetrieben entwickelt werden. Das grundlegende Prinzip lässt sich prinzipiell auf andere Formulare übertragen, sogar verschachtelte Formulare sind umsetzbar, wie in der `RecapForm.java` und der `FormAssessment.java` gezeigt. Dort wird zum einen das Recap-Formular an sich validiert, das bedeutet das Datum des Recaps darf weder leer sein noch in der Zukunft liegen, aber auch die Liste der Bewertungen soll validiert werden. Für jedes `FormAssessment` ist daher zusätzlich noch festgelegt, dass der Wert nicht `null` sowie zwischen null und fünf liegen muss.

4.3 Das Testen der Service-Schicht

Nachdem im vorherigen Kapitel ausführlich das Testing des Web-Interfaces behandelt worden ist, soll nun eine weitere wichtige Komponente in den Fokus rücken: die Service-Schicht. Diese kann unabhängig von den anderen Komponenten entwickelt werden. Im Falle des hier vorliegenden Projektes – der Entwicklung des Spielzeitenplaners – ist die Service-Schicht schrittweise aufgebaut worden.
Dabei wurde sich an verschiedenen `use cases` orientiert, die die funktionalen Anforderungen der Anwendung hervorheben und die wiederum eine Interaktion der Nutzenden mit der Web-Oberfläche als Startpunkt besitzen. Ein Beispiel für einen in diesem Projekt vorliegenden `use case` wäre beispielsweise das Löschen eines Spielers durch die Nutzenden. Durch einen Klick auf den `Löschen`-Button auf der `TeamPage` wird das entsprechende Formular an den Server geschickt. Dort wird es von einer entsprechend konfigurierten Handler-Methode eines Controllers in Empfang genommen und verarbeitet. Im Zuge dessen wird die entsprechende Service-Methode – in diesem Fall die `deletePlayer`-Methode des `PlayerService` – aufgerufen, die den Fall dann bearbeitet. Schließlich ruft diese dann eine entsprechende Repository-Methode auf, die wiederum für die Löschung der Spieler-Daten in der Datenbank zuständig ist.
Innerhalb dieser Aufrufkette interagiert eine Komponente also gewissermaßen mit der nächsten. Bezogen auf die Onion-Architektur bedeutet dies: die Anfrage wird von der äußeren Web-Schicht an die inneren Schichten – also die Service-Schicht und das Domain-Model – weitergeleitet und diese delegieren die entsprechdende Aufgabe dann wieder an die (äußere) Persistenzschicht. Auf diese Weise können sich Entwickelnde vom einen Ende der Software-Zwiebel hindurch zum anderen arbeiten.

Ein konkretes Beispiel aus der Entwicklung des Spielzeitenplaners ist dem Commit 0a64ca3 zu entnehmen. Durch die Implementierung der Funktionalitäten in der Web-Schicht – siehe dazu die Änderungen an der `TeamControllerTest.java` – wird unter anderem das Vorhandensein der `deletePlayer`-Methode und damit verbunden auch die Existenz des `PlayerService` an sich gefordert. Um den Test also bestehen zu lassen und schließlich committen zu können ist die Erstellung einer `PlayerService.java` und einer darin enthaltenen `deletePlayer`-Methode zwingend erforderlich.
Sobald die Service-Klasse vorhanden ist, kann sie auch testgetrieben entwickelt werden. Im Folgenden soll nun zunächst das Vorgehen bei den sogenannten Basisoperationen geschildert werden, ehe dann auf komplexere Testfälle eingegangen werden soll. Das zuvor genannte Beispiel für die `deletePlayer`-Methode lässt sich einfach und unkompliziert wie folgt testgetrieben entwickeln:

```
@Test
...
void test_01() {
    Integer playerId = 17;
    playerService.deletePlayer(playerId);
    verify(playerRepository).deleteById(playerId);
}
```

Hier wird also überprüft, ob der `PlayerService` die Löschanfrage korrekt weiterverarbeitet. In diesem Fall bedeutet dies, dass diese an das zuständige Repository delegiert wird. Dabei ist besonders wichtig, dass das richtige Repository – hier also das `PlayerRepository` – verwendet wird und die entsprechende Methode mit dem aus der ursprünglichen Anfrage gesendeten Objekt aufgerufen wird.
Um diesen Test so schreiben zu können, muss das `PlayerRepository` zunächst mithilfe von `Mockito` gemockt werden, um die Kontrolle über sämtliche Aktionen des Repository zu erhalten, und der `PlayerService` mit dem soeben gemockten Repository ordnungsgemäß initialisiert werden, was wiederum einen geeigneten Konstruktor in der `PlayerService.java` voraussetzt. In diesem Zuge wird dann auch das Interface `PlayerRepository.java` innerhalb der Service-Schicht etabliert und seine Implementierung, die mit `@Repository` annotierte `PlayerRepositoryImpl.java`, in der Persistenzschicht. Ihre testgetriebene Entwicklung ist in Kapitel 3.5 beschrieben. Sind die zuvor genannten Voraussetzungen implementiert, so kann der `deletePlayer`-Methode der Aufruf `playerRepository.deleteById(id)` hinzugefügt werden, wobei `id` als Argument entgegen genommen wird (siehe dazu `PlayerService.java nach Commit e4bc878`).
Die Methode zum Löschen eines Spielers ist ein Beispiel für eine Basisoperation, bei der eine Anfrage an die entsprechende Repository-Methode weitergeleitet wird. Hier steht also die Delegation der Aufgabe im Vordergrund, während das Repository sich um die Löschlogik kümmert. Für die Verifizierung der Übergabe des korrekten Parameters wurde `deleteById(playerId)` auf `verify(playerRepository)` aufgerufen. Dies ist in diesem Fall auch völlig ausreichend, sollten jedoch mehrere Methodenaufrufe mit unterschiedli-

chen Argumenten auftreten, komplexe Objekte – wie beispielsweise ein Spieler mit seinen verschieden Attributen – übergeben oder aber Eigenschaften innerhalb des Methodenaufrufs verändert werden, so ist der Gebrauch eines `ArgumentCaptors` durchaus sinnvoll, wie `test_01` der `RecapServiceTest.java` veranschaulicht:

```
@Test
@DisplayName("Die Bewertungen werden gespeichert.")
void test_01() {
    List<Assessment> assessments = List.of(
            // Hier manuell erstellte Bewertungen hinzufügen
    );
    ArgumentCaptor<Assessment> assessmentCaptor =
        ArgumentCaptor.forClass(Assessment.class);

    recapService.submitAssessments(assessments);

    verify(assessmentRepository, times(4))
        .save(assessmentCaptor.capture());

    List<Assessment> savedAssessments =
        assessmentCaptor.getAllValues();

    assertThat(savedAssessments).isEqualTo(assessments);
}
```

In dem oben gezeigten Test wird der `ArgumentCaptor` dazu benutzt, die verschiedenen Bewertungen, mit der die `save`-Methode aufgerufen wird, einzufangen. Bevor dies mithilfe `assessmentCaptor.capture()` geschehen kann, muss die spezifische Klasse, für die der `Captor` programmiert ist, zunächst bei der Initialisierung des `assessmentCaptors` konfiguriert werden. Nach dem Erfassungsvorgang können alle gespeicherten `Assessments` dann abgerufen und in einer Liste gespeichert, um schließlich weiteren Überprüfungen mithilfe von `assertThat` unterzogen werden zu können.
Während bisher Gesagtes vor allen Dingen das Delegieren eines Aufrufs an die Persistenzschicht und die Überprüfung der Übergabe von Parametern in den Vordergrund stellt, soll nun im Folgenden auf die Überprüfung von Rückgabewerten eingegangen werden. Eine in diesem Projekt häufig getestete Methode ist die `load`-Methode, die Daten aus der Datenbank anfragt und diese dem Web-UI zur Verfügung stellt. Beispielsweise liefert die `loadCriteria`-Methode des `SettingsService` die aktuell gespeicherten Kriterien, `loadFormation` stellt die aktuell genutzte Formation bereit und `loadPlayers` gewährleistet die Versorgung der Webschicht mit den aktuellen Spielern im Team. Letztere wird wie folgt getestet:

```
@Test
...
void test_02() {
    List<Player> players = TestObjectGenerator.generatePlayers();
    when(playerRepository.findAll()).thenReturn(players);

    List<Player> loadedPlayers = playerService.loadPlayers();

    verify(playerRepository).findAll();
    assertThat(loadedPlayers)
        .containsExactlyInAnyOrderElementsOf(players);
}
```

Nachdem der `TestObjectGenerator` eine Liste von (Test-)Spielern bereitgestellt hat, kann das `PlayerRepository` so konfiguriert werden, dass es diese Liste von Spielern zurückgibt. Sämtliche Repositories werden für die Service-Tests gemockt, um die Service-Schicht von der Datenbank losgelöst testen zu können. In dem folgenden `Act`-Schritt des Tests wird die `loadPlayers`-Methode dann ausgeführt und die Rückgabe in einer lokalen Variable gespeichert. So kann dann abschließend getestet werden, ob die `findAll`-Methode des `PlayerRepository` aufgerufen wurde – die Überprüfung der Delegation eines Requests ist ja bereits aus den vorangegangenen Tests bekannt – und ob die geladenen Spieler auch wirklich der gewünschten Liste entsprechen.
Dabei wird die `containsExactlyInAnyOrderElementsOf`-Methode verwendet, um sicherzustellen, dass sich auch wirklich nur die erwarteten Spieler in der Liste befinden und keine weiteren. Außerdem spielt die Reihenfolge keine Rolle, damit der Test auch weiterhin besteht, sollten die Spieler nach Namen, Score, Trikotnummer oder Position geordnet werden. Zusammenfassend kann gesagt werden, dass beim Delegieren und Koordinieren von Anfragen ein besonderes Augenmerk zum einen auf den korrekten Methodenaufruf, zum anderen aber auch die Überprüfung der Übergabe der korrekten Parameter sowie die Richtigkeit des Rückgabewertes gelegt werden sollte, denn diese sind für eine ordnungsgemäße Funktionsweise der Anwendung unerlässlich.
Darüber hinaus ist es wichtig Methoden der Service-Schicht zu testen, die sich nicht eindeutig einer Klasse aus dem Domain-Model zuordnen lassen oder aber eine Verknüpfung zwischen mehreren Entitäten herstellen. Im Bezug auf den Spielzeitenplaner ist dies zum Beispiel der Fall in der `SpielzeitenService.java`, in der aus allen verfügbaren Spielern ein Kader oder aus dem Kader wiederum eine Startelf ermittelt werden soll, und in der `PlayerService.java`, in der für einen Spieler für jedes Kriterium anhand mehrerer Bewertungen ein Score berechnet wird.
Als anschauliches Beispiel soll im Folgenden die `calculatePlannedMinutes`-Methode des `SpielzeitenService` dienen. Diese soll anhand einer Spieler- sowie einer Wechsel-Liste die sich durch die einzelnen Wechsel ergebenen Spielminuten für jeden Spieler ermitteln. Den Basisfall stellt hier die Situation dar, in der keine Wechsel vorgenommen werden. Dann beträgt die Spielzeit jedes Startelf-Spielers 70 Minuten und die jedes Reservespielers exakt

0 Minuten. Ein strukturierter Test, der eben genau diese Situation prüft, ist `test_05()` der `SpielzeitenService.java`.
Über den Basisfall hinaus gibt es dann noch weitere Testfälle, in denen die ordnungsgemäße Funktionsweise der Service-Methode getestet werden kann und muss, beispielsweise wenn ein einfacher Wechsel vorgenommen wird, `Spieler Y` also für `Spieler X` in einer bestimmten Spielminute eingewechselt wird. Damit wird zum ersten Mal die korrekte Berechnung der Spielminuten getestet, die nicht 70 oder 0 betragen, also über den Basisfall hinausgehen. Diese Situation wird in `test_06()` simuliert.
Nach und nach lassen sich komplexere Testsituationen konstruieren, um die Methode robuster zu machen. Der folgende `test_07()` prüft die ordnungsgemäße Funktionsweise, wenn ein Spieler aus- und wieder eingewechselt wird:

```
@Test
...
void test_07() {
    List<Substitution> substitutions = List.of(
        new Substitution(1, 20, "Player16 Last16", "Player8 Last8"),
        new Substitution(3, 50, "Player8 Last8", "Player16 Last16")
    );
    List<Integer> expected = List.of(
        70, 70, 70, 70, 70, 70, 70, 40, 70, 70, 70, 0, 0, 0, 0, 30
    );

    List<Integer> plannedMinutes = spielzeitenService
                    .calculatePlannedMinutes(squad, substitutions);

    assertThat(plannedMinutes).hasSize(squad.size());
    assertThat(plannedMinutes).isEqualTo(expected);
}
```

Hier werden zunächst die Liste der Wechsel sowie das erwartete Ergebnis vorbereitet. Im Anschluss wird dann die zu testende Methode aufgerufen und ihre Rückgabe zur folgenden Überprüfung gespeichert. Mithilfe geeigneter `Assertions` kann schließlich sichergestellt werden, dass die Methode das richtige Ergebnis liefert.
Im weiteren Verlauf der Entwicklung können nun noch weitere, komplexere Situationen geprüft werden, um die Methode schließlich in ihre gegenwärtige Form zu bringen. In `test_08()` ist beispielsweise eine Situation mit mehreren Wechseln, mehreren Spielern und unterschiedlichen Wechselzeitpunkten kreiert worden, die so von einem Trainerteam bei der Planung der Spielzeiten realisiert werden könnte. Auch hier wird mithilfe geeigneter `Assertions` gewährleistet, dass die geplanten Spielminuten wie erwartet ermittelt werden.

4.4 Realitätsnahe Datenbank-Tests mit Testcontainers

In den vorangegangenen Kapiteln wurde gezeigt, wie innerhalb der Web- und Service-Schicht testgetrieben entwickelt werden kann. Dabei wurde zunächst die Benutzeroberfläche entwickelt und damit verbunden auch die Web-Steuereinheiten – die Controller. Diese rufen unter anderem die für die Benutzeranfragen zuständigen Service-Methoden auf. Die einzelnen Services verwenden wiederum geeignete Repository-Interfaces und ihre Methoden als Schnittstelle zur Persistenzschicht, deren konkrete Implementierungen für das Laden der entsprechenden Daten aus der Datenbank verantwortlich sind. Das folgende Kapitel soll sich demnach der testgetriebenen Entwicklung dieser Repositories – und damit verbunden der Persistenzschicht im Allgemeinen – widmen.
Im Allgemeinen ist zu sagen, dass für sämtliche Datenbank-Tests die `Testcontainers`-Bibliothek in Kombination mit `Docker` verwendet wird. Eine solche Konfiguration bietet gleich mehrere Vorteile: Erstens werden so realistische Tests mit einer echten Instanz der für das Projekt ausgewählten Datenbank ermöglicht. Die Test-Datenbank kann somit exakt so wie die Produktiv-Datenbank konfiguriert werden, wodurch möglichst realistische Bedingungen simuliert werden.
Zweitens sorgt die Verwendung von `Testcontainers` dafür, dass jeder Test in einer bereinigten Umgebung stattfindet, um zu verhindern, dass sie einander beeinflussen und das Ergebnis verfälschen. Drittens wird das Hoch- und Runterfahren sowie das Setup des `Docker-Containers` automatisiert und von `Testcontainers` übernommen.
Doch um `Testcontainers` mit seinen zuvor benannten Vorteilen nutzen zu können, muss auch ein gewisser Preis gezahlt werden, denn mit der Verwendung der Java-Bibliothek gehen wiederum auch einige Nachteile einher. Die realistischen Tests mit einer Test-Datenbank benötigen zusätzliche Systemressourcen und das Starten sowie Stoppen der Test-Container benötigt Zeit, besonders dann, wenn Images gezogen werden müssen. Außerdem benötigt `Testcontainers` eine funktionierende Docker-Installation, was wiederum eine weitere Abhängigkeit darstellt.
Daher muss bei der Konzeption der Datenbank-Tests stets abgewogen werden, ob nun die Vor- oder Nachteile überwiegen und damit verbunden die spezifischen Anforderungen des jeweiligen Projektes geprüft werden, um schließlich die bestmögliche Lösung zu wählen. Im Falle des Spielzeitenplaners überwiegen die zuvor beschriebenen Vorteile, denn es soll Wert auf realitätsnahe Datenbank-Tests gelegt werden, Nachteile wie der erhöhte Ressourcenbedarf oder die längeren Testlaufzeiten haben hier keine große Relevanz.
Grundsätzlich sind alle Datenbank-Testklassen, die im Verzeichnis `src/test/java/de/bathesis/spielzeitenplaner/database` zu finden sind, auf die gleiche Art und Weise aufgebaut. Zunächst einmal muss die jeweilige Testklasse mit drei verschiedenen Annotationen versehen werden: (1.) `@DataJdbcTest`, durch die Spring eine spezielle Testumgebung für die Persistenzschicht konfiguriert – der Web-Layer zum Beispiel wird komplett deaktiviert – um ressourcenschonender arbeiten zu können, (2.) `@AutoConfigureTestDatabase(replace = AutoConfigureTestDatabase.Replace.NONE)`, die dafür sorgt, dass Spring die Produktiv-Datenbank nicht automatisch durch eine `In-Memory`-Datenbank ersetzt, sondern die durch `Testcontainers` konfigurierte Test-Datenbank benutzt, und (3.) `@Testcontainers`, die für

die Verwaltung der innerhalb der Klasse definierten Container zuständig ist (`Start`, `Stop` und `CleanUp`).
In dem nun folgenden Schritt wird ein solcher Container definiert. Dies geschieht durch folgendes Statement:

```
@Container
private static PostgreSQLContainer<?> postgresContainer =
        new PostgreSQLContainer<>("postgres:15-alpine")
                .withDatabaseName("testdb")
                .withUsername("testuser")
                .withPassword("testpassword");
```

Es wird ein `Postgres`-Container erstellt und konfiguriert, indem die `PostgreSQL`-Version, der Name der Test-Datenbank sowie Benutzername und Passwort festgelegt werden. Auch weitere Konfigurationsmöglichkeiten können hier hinzugefügt werden, um die Datenbank den Ansprüchen entsprechend zu gestalten. Die `@Container`-Annotation sorgt dafür, dass der `Postgres`-Container als solcher erkannt wird und das zuvor erwähnte Starten und Stoppen automatisiert werden kann.
Damit `Spring` sich nun auch wirklich mit der Test-Datenbank – und nicht mit der Produktiv-Datenbank – verbindet, müssen die Eigenschaften der ursprünglichen Datenbankverbindung, die in der `application.yaml` festgelegt sind, während der Tests überschrieben werden. Dies geschieht mithilfe der mit `@DynamicPropertySource`-annotierten `configureProperties`-Methode. Innerhalb dieser werden in der `DynamicPropertyRegistry` die nötigen Verbindungseigenschaften – darunter die dynamisch generierte `JdbcUrl` sowie `Username` und `Password` der durch den `postgreSQLContainer` definierten Test-Datenbank – festgelegt. Ist dies geschehen, so wird die Produktiv-Datenbank für die Dauer der Tests erfolgreich durch die Test-Datenbank substituiert.
Bevor nun konkret mit dem Schreiben einzelner Tests begonnen werden kann, müssen die notwendigen Repositories bereitgestellt werden. Da eine strikte Einhaltung der Onion-Architektur klar definierte Schnittstellen zwischen der inneren Schicht der Domäne und äußeren Schichten – wie der Persistenzschicht – erfordert, müssen diese bereits bei der Entwicklung der Service-Schicht definiert werden. Auch das in den Tests verwendete Mocking ist ohne die Existenz der Klasse gar nicht möglich. Im gleichen Zuge ist dann auch direkt eine Klasse für die jeweilige Repository-Implementierung erstellt worden, auch wenn diese bis zu diesem Zeitpunkt noch leer bleibt. Lediglich die `@Repository`-Annotation muss verwendet werden, da sonst der `contextLoads()`-Test fehlschlägt, weil Spring die relevante `Bean` nicht finden kann.
Zur Veranschaulichung dieses Sachverhalts soll nun die `DatabaseAssessmentTest.java` dienen. Konkret auf die `Assessment-Repositories` bezogen ergeben sich folgende `AssessmentRepository.java` und `AssessmentRepositoryImpl.java`, wie den verknüpften Links zu entnehmen ist.

Nun muss ein `SpringDataAssessmentRepository` – kurz: `SpringRepository` – erstellt werden, das wiederum ein `Interface` ist und gewissermaßen eine Brücke zwischen der Anwendung und der Datenbank bildet. Es erbt von `CrudRepository` und ermöglicht die sogenannten `CRUD-Operationen` – also `Create`, `Read`, `Update` und `Delete` – bezogen auf die Entität `Assessment`. Dieses `SpringRepository` wird dann mithilfe der `@Autowired`-Annotation und eines Konstruktors in der `DatabaseAssessmentTest.java` in das `AssessmentRepository` injiziert. Dieses Vorgehen erfordert wiederum einen Konstruktor in der `AssessmentRepositoryImpl.java`, der zu diesem Zeitpunkt erstellt werden kann.
Um die Persistenzschicht und ihre Daten klar von den anderen Schichten abzutrennen, verwendet das `SpringRepository` eine eigene `Assessment`-Klasse, die im Verzeichnis `src/main/java/de/bathesis/spielzeitenplaner/database/entities` verortet ist. Sie kann zunächst als einfacher `Record` mit den entsprechenden Attributen aus der Domain-Klasse implementiert werden. Zur Verdeutlichung der zuvor beschriebenen Schritte sowie der Trennung der Service- und Persistenzschicht und ihre Schnittstellen kann Abbildung 5 herangezogen werden.

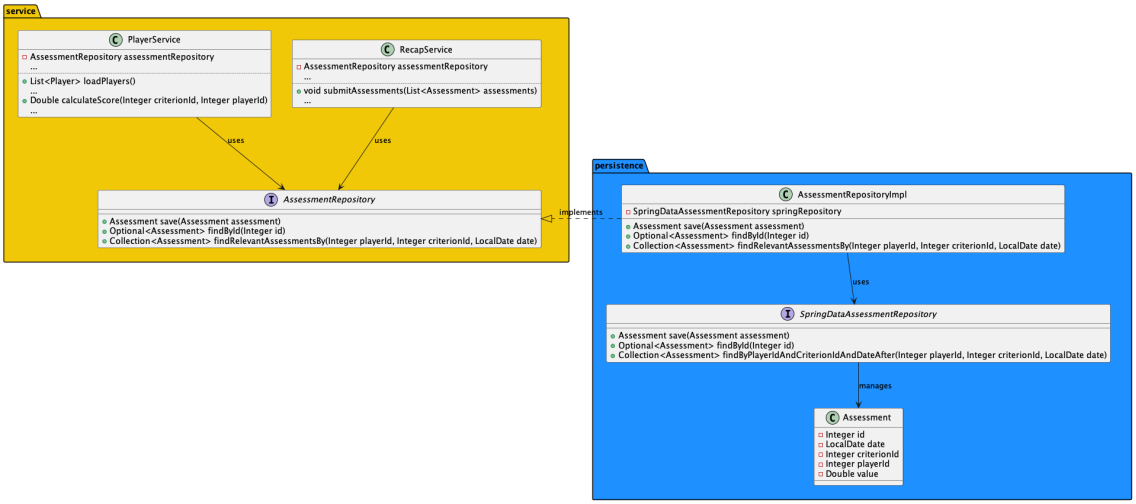

Abbildung 5: Modellierung von Service- und Persistenzschicht am Beispiel der Assessment.java

Sind all die zuvor beschriebenen Schritte vollzogen, so kann mit dem Schreiben konkreter Tests begonnen werden. Zunächst einmal ist es wichtig, dass das Speichern und Laden einer Entität wie gewünscht funktioniert. Diese beiden Operationen sind untrennbar miteinander verbunden, denn um zu überprüfen, ob eine Entität korrekt in der Datenbank gespeichert wurde, muss diese zunächst geladen werden. Gleiches gilt für Umkehrrichtung: Um zu überprüfen, ob eine Entität korrekt geladen wurde, muss diese zunächst in die Datenbank eingefügt werden. Der Test, der feststellt, ob eine Bewertung korrekt gespeichert und geladen wird, kann wie folgt geschrieben werden:

```
@Test
void test_01() {
    Assessment assessment = new Assessment(
        null, LocalDate.of(2022, 12, 31), 1, 1, 3.0
    );

    Assessment saved = assessmentRepository.save(assessment);
    Assessment loaded = assessmentRepository
        .findById(saved.getId()).get();

    assertThat(loaded).isEqualTo(saved);
}
```

Zuerst wird ein zu speicherndes Test-Objekt erstellt. Dieses wird dann mittels der `save`-Methode in der Datenbank gespeichert und mit `findById` geladen, ehe das gespeicherte mit dem geladenen Objekt abgeglichen und auf Gleichheit geprüft wird. Damit dieser Test nun bestehen kann, muss die `Docker`-Anwendung – das hier zur Entwicklung genutzte Endgerät verwendet die `Docker Desktop-App` – auf dem testenden Gerät laufen. Darüber hinaus können dem `SpringDataAssessmentRepository` die entsprechenden Methodensignaturen – gemeint sind `Assessment save(Assessment assessment)` und `Optional<Assessment> findById(Integer id)` – hinzugefügt werden, um größere Kontrolle über deren Implementierung zu erhalten. Dieser Schritt ist in diesem Fall jedoch nicht zwingend notwendig, da die Methoden bereits über das `CRUD`-Repository zur Verfügung gestellt werden.
Schließlich müssen dann auch die im Test genutzten Repository-Mehoden zum Speichern und Laden einer Bewertung implementiert werden. Dies gestaltet sich wie folgt: Im Falle der `save`-Methode erfolgt eine Übersetzung des `DomainAssessment` zum `DatabaseAssessment`, das dann der `save`-Methode des `SpringDataAssessmentRepository` zum Speichern übergeben werden kann. Anschließend wird das gespeicherte Objekt für die Rückgabe vorbereitet, indem es zurück in ein entsprechendes Domänen-Objekt konvertiert wird. Der `AssessmentMapper` bzw. die für diese Konvertierung zuständigen Methoden müssen an dieser Stelle ebenfalls implementiert werden. Analog dazu wird die `findById`-Methode implementiert, indem die Anfrage an die `findById`-Methode des `SpringDataAssessmentRepository` delegiert wird und dessen Rückgabe zum Domain-Objekt konvertiert und zurückgegeben wird.
Außerdem sollte sichergestellt werden, dass die entsprechende Tabelle in der Datenbank existiert. Ist dies nicht der Fall, wird eine `BadSqlGrammarException` geworfen, da `JDBC` versucht, die Entität zur Datenbank hinzuzufügen, die entsprechende Relation jedoch nicht existiert. Dieser `Exception` kann beispielsweise entgegengewirkt werden, indem mithilfe von `Flyway` ein entsprechendes Migrationsschema erstellt wird. Dazu sollte die Anwendung nach Erstellung des Migrationsschemas wenigstens einmal gestartet werden, sodass `Flyway` die Migration durchführen kann.
Schließlich muss der Instanzvariable `id` der `Assessment.java` – gemeint ist die `DatabaseAssessment`-Klasse im Verzeichnis `database/entities` – noch die `@Id`-Annotation hinzugefügt werden, damit `JDBC` weiß, dass es sich hierbei um dem Primärschlüssel der

Tabelle handelt und Anfragen somit ordnungsgemäß ausführen kann.
Neben `save` und `findById` stellt die `findAll`-Methode ebenfalls ein nützliches Werkzeug zum Laden mehrerer Entitäten aus der Datenbank dar, das im Rahmen des Spielzeitenplaners häufig Gebrauch findet. Der Ablauf des Tests ist dem der `findById`-Methode ähnlich und ist zum Beispiel unter dem Namen `test_02` in der `DatabasePlayerTest.java` zu finden. Dort werden zunächst mehrere Spieler gespeichert und anschließend mithilfe der `findAll()`-Methode des `CRUD`-Repositories geladen. Abschließend wird dann geprüft, ob die zuvor gespeicherten Elemente durch die `findAll`-Methode gefunden werden.
Dank `JDBCs Derived Queries` ist es möglich, über die standardmäßig implementierten Basismethoden hinaus auch eigene, spezifischere Methoden zu schreiben, die anhand des Methodennamens sowie aufgrund bestimmter Regeln und Konventionen entsprechende Anfragen ableiten und ausführen können. Im Kontext des Spiezeitenplaners beispielsweise ist es für die Berechnung der Scores hilfreich, die gespeicherten Bewertungen nach mehreren Kriterien – wie dem Namen, Kriterium und Datum – zu filtern, sodass die Score-Berechnung auch nur die gewünschten Daten als Grundlage nimmt. Ein `Test` für die so entstandene `findRelevantAssessmentsBy`-Methode kann wie folgt aussehen:

```
@Test
...
void test_02() {
    Integer playerId = 1;
    Integer criterionId = 1;
    Assessment assessment1 = new Assessment(
        null, LocalDate.of(2022, 12, 31), criterionId, playerId, 5.0
    );
    Assessment assessment2 = new Assessment(
        null, LocalDate.of(2022, 12, 29), criterionId, playerId, 3.0
    );
    Assessment assessment3 = new Assessment(
        null, LocalDate.of(2022, 12, 31), criterionId, 2, 2.0
    );
    Assessment saved1 = assessmentRepository.save(assessment1);
    Assessment saved2 = assessmentRepository.save(assessment2);
    Assessment saved3 = assessmentRepository.save(assessment3);

    Collection<Assessment> found =
        assessmentRepository.findRelevantAssessmentsBy(
            playerId, criterionId, LocalDate.of(2022, 12, 1)
    );

    assertThat(found).containsExactly(saved1, saved2);
    assertThat(found).doesNotContain(saved3);
}
```

Auch in diesem Fall ist es zunächst notwendig einige Vorbereitungen zu treffen: Benötigte sowie geeignete Test-Daten müssen generiert und in der Datenbank gespeichert werden, ehe diese dann in einem `Act`-Schritt abgerufen und gefiltert werden. Ob dieser Schritt wie erwartet funktioniert, wird schließlich überprüft: Im Falle des zuvor vorgestellten Tests sollten nur `assessment1` und `assessment2` bzw. `saved1` und `saved2` gefunden werden, `assessment3` bzw. `saved3` hingegen nicht, da diese Bewertung eine andere als die gewünschte `playerId` besitzt.
Nach diesem Prinzip kann die Methode unter weiteren, verschiedenen Bedingungen getestet werden, um diese robuster zu machen. Zum Beispiel können die `Assessments` so gewählt werden, dass sie überprüfen, ob das Filtern nach Datum korrekt funktioniert. Gleiches gilt für das Filtern nach Kriterium. Des Weiteren ist auch die Überprüfung von Randfällen sinnvoll, um sicherzustellen, dass die Methode auch unter weniger optimalen Bedingungen das richtige Ergebnis liefert.

5 Kritische Reflexion des Entwicklungsprozesses

Im folgenden Kapitel soll sich kritisch mit dem Entwicklungsprozess des Spielzeitenplaners und der gewählten Methode des Test-Driven Developments auseinandergesetzt werden.
Zunächst ist festzuhalten, dass TDD ein mächtiges Werkzeug zur Entwicklung moderner Software ist: Mit ihr kann die Qualität des Codes verbessert und die Anfälligkeit für Fehler in der Software verringert werden. Die kleinen, testbaren Einheiten, die durch konsequentes Anwenden des TDD-Prinzips entstehen, sind sauber, modular, gut wartbar und dienen gleichzeitig als eine Art Dokumentation des Entwicklungsprozesses.
Doch Test-Driven Development bringt auch einige Einschränkungen und Stolpersteine mit sich, die auch im Rahmen der Entwicklung des Spielzeitenplaners deutlich geworden sind und im Folgenden thematisiert werden sollen. Als Erstes ist hier der Faktor Zeit zu nennen. Dieser ist nicht unerheblich und in der heutigen schnelllebigen Welt immer mehr von Relevanz. Test-driven Development benötigt schlichtweg mehr Zeit als die gewöhnliche Entwicklung – insbesondere zu Beginn des Entwicklungsprozesses, da zunächst erst ein geeigneter Test formuliert werden muss, bevor dann die konkrete Implementierung realisiert werden kann.
Die Gegenpole TDD vs. gewöhnliche Entwicklung können relativ treffend mithilfe einer Analogie aus der Leichtathletik beschrieben werden: der 100-Meter-Lauf vs. Marathon. Während es beim 100-Meter-Lauf im Prinzip darum geht, so schnell wie möglich ans Ziel zu kommen, ist beim Marathon eine langfristige, beständige Strategie gefragt, um ans Ziel zu kommen und schließlich Erfolg zu haben. Anstatt den Wert auf Schnelligkeit zu legen, erfordert TDD eher ein moderates, beständiges Tempo: Ähnlich wie beim Marathon wird in kleinen, stetigen und zu bewältigen Schritten gearbeitet, bis man schließlich am Ziel angekommen ist. Außerdem ist es nicht das Ziel, schnell funktionierenden Code zu produzieren, sondern langfristig stabil und wartbare Software zu erschaffen. Darum kann der Fortschritt bei TDD manchmal langsamer erscheinen, wovon man sich als Entwickelnde nicht entmutigen lassen sollte.
Wenn der zuvor beschrieben Faktor Zeit jedoch nicht in ausreichendem Maße vorhanden ist, weil beispielsweise Deadlines unbedingt eingehalten werden müssen, oder der Fokus auf einer schnellen Fertigstellung ohne Beachtung der langfristigen Wartbarkeit liegt, sollte eher über eine andere Methode der Softwareentwicklung nachgedacht oder die Prioritäten überdacht werden.
Neben dem Faktor Zeit ist außerdem die Tatsache zu nennen, dass die testgetriebene Entwicklung einen `Mind Shift` der Entwickelnden erfordert, da der gewöhnliche Entwicklungsprozess auf den Kopf gestellt wird, indem die Tests zuerst geschrieben werden. Dieses Umdenken ist für TDD-Anfänger unter Umständen schwierig aufrechtzuerhalten, insbesondere wenn die Idee für die konkrete Implementierung einer Funktionalität bereits in den Köpfen der Entwickelnden vorhanden ist, während das Schreiben des Tests zugleich noch unklar ist.
Im Rahmen der Entwicklung des Spielzeitenplaners kam es insbesondere beim Entwickeln gewisser Funktionalitäten der Service-Schicht zu solchen Situationen, in denen sich das Schreiben von Tests schwierig gestaltete, während die Idee für den Produktivcode bereits

vorhanden war. Hier bestand die Gefahr, dass das TDD-Prinzip eben doch wieder umgekehrt wird, indem der Produktivcode als Anhaltspunkt für das Schreiben eines Tests genommen wurde.
Weitere wichtige Aspekte, die eng mit den zuvor genannten Argumenten der Zeit und des Umdenkens verbunden sind, sind Geduld, Disziplin und Vertrauen. Test-driven Development benötigt Geduld, die bereits genannten kleinen Fortschritte beharrlich umzusetzen und nicht in alte Muster zurückzufallen. Des Weiteren ist für diese Methode ein hohes Maß an Disziplin notwendig, denn der `Red-Green-Refactor`-Zyklus muss stets eingehalten werden, es sind keine Abkürzungen erlaubt, auch wenn ein Test für eine bestimmte Funktionalität auf den ersten Blick trivial erscheint.
Darüber hinaus ist es in stressigen Situationen – wie beispielsweise engen Deadlines oder unvorhergesehenen Verzögerungen im Entwicklungsablauf – verlockend, den TDD-Prozess zu umgehen, auch hier ist Disziplin notwendig, um die Vorteile des Ansatzes auch unter Druck nutzen können. Auch im Rahmen des Entwicklungsprozesses des Spielzeitenplaners ist es zu Verzögerungen gekommen, weshalb insbesondere zum Ende der Entwicklung der Zeitdruck und damit verbunden auch der Druck nach einer schnellen Lösung zunahm.
Außerdem braucht die testgetriebene Entwicklung Vertrauen auf Seiten der Entwickelnden. Das Vertrauen, dass die Methode bei konsequenter Anwendung zu den gewünschten Ergebnissen führt, auch wenn diese zu einem gewissen Zeitpunkt in der Entwicklung noch nicht absehbar sind. Das Vertrauen und die Akzeptanz, dass Tests immer wieder und so lange fehlschlagen, bis alle zum Bestehen nötigen Details implementiert sind, da dies eben genau von TDD beabsichtigt ist.
Und schließlich ist das Vertrauen in TDD eben auch angebracht, wenn eine neue Abhängigkeit notwendig geworden ist und eingeführt wird. So mussten beim Spielzeitenplaner beispielsweise mit Einführung des `PlayerService` die entsprechenden Abhängigkeiten – insbesondere in der Web-Schicht – gemockt werden, um wieder zu einem konsistenten Gesamtzustand zu gelangen. Dabei fungierten die Tests als ein Warnsystem, das Alarm schlägt, weil bestehende Tests unter Umständen noch nicht auf die sich verändernde Architektur abgestimmt sind. Hier ist also Vertrauen in die Tests und damit verbunden in den TDD-Ansatz notwendig sowie die stetige Reflexion über die geschriebenen Tests und ihren Zweck.

6 Fazit

Im Großen und Ganzen ist zu konstatieren, dass das Projekt `Spielzeitenplaner` gezeigt hat, wie eine Anwendung ganzheitlich und auf verschiedenen Ebenen testgetrieben entwickelt werden kann.
Für die einzelnen `WebPages`, deren testgetriebene Entwicklung zu Beginn im Vordergrund stand, ist ein Konzept entwickelt worden, nach dem jede einzelne Seite strukturiert getestet wird: Zunächst wird durch eine Reihe von Tests gewährleistet, dass `Essentials` wie Navigationsleiste und Footer oder aber auch die korrekte Überschrift einer Seite vorhanden sind und den gewünschten Anforderungen entsprechen.
Im Anschluss erfolgt eine Testung der einzelnen Bereiche einer Seite, beispielsweise ob der Bereich `Spieler-Daten` auf der Spieler-Seite korrekt strukturiert ist. Dies beinhaltet unter anderem auch Tests, die die Anwesenheit von Formularen erzwingen, die in diesem Projekt zum einen der Anzeige bereits gespeicherter Daten dienen, den Nutzenden zum anderen aber auch die Möglichkeit bieten, Anfragen zum Speichern oder Bearbeiten an den Server zu senden. Darüber hinaus ist mithilfe geeigneter Tests ebenfalls sichergestellt worden, dass Daten, wie Spielerinformationen, Teamname oder Formation, die im Model bereitstehen, wie gewünscht in die entsprechende Seite hineingerendert werden.
Für das Testing der einzelnen `WebPages` erwies sich `Jsoup` als nützliches Werkzeug zum Parsen und Extrahieren von HTML-Elementen aus einer Seite. Über die Komplexität der `CSS-Query` konnte dabei gezielt gesteuert werden, wie detailliert die jeweilige Seite im Hinblick auf ihre Struktur getestet werden soll.
Im Bereich des Controller-Testing ist gezeigt worden, wie die Web-Steuereinheiten mit ihren verschiedenen Aufgaben testgetrieben entwickelt und überprüft werden können. Dabei ist grundsätzlich zwischen GET- und POST-Requests unterschieden worden. Bei Ersteren wird zunächst mithilfe eines geeigneten Tests das grundlegende Routing überprüft und etabliert sowie sichergestellt, dass das richtige HTML-Template zurückgegeben wird. In einem zweiten Schritt kann dann überprüft werden, ob die geforderten Daten durch den Controller bei der Service-Schicht angefragt werden und das entsprechende Model geladen werden.
Die grundsätzliche Vorgehensweise bei POST-Requests unterschiedet sich ein wenig von der, die bei GET-Requests angewendet wird. In einem solchen Fall wird zwar auch zunächst erst das grundlegende Routing überprüft und eingerichtet, in einem weiteren Schritt jedoch überprüft, ob die richtige Methode des für die Anfrage zuständigen Services aufgerufen wird. Schließlich müssen die übermittelten Daten beim Abschicken eines Formulars noch durch den Controller validiert werden. Um diese Funktionalität umzusetzen, sind für jeden Fall gezielte Anfragen konstruiert worden, die eine korrekte Validierung überprüfen.
Durch das konsequente Anwenden der Onion-Architektur ist gleichzeitig aber auch die Existenz der einzelnen Services bzw. der Service-Schicht an sich erzwungen worden. Im Rahmen der Service-Tests sind grundlegende Operationen zur Delegation von Benutzeranfragen getestet worden, beispielsweise die `loadPlayers`-Methode des `PlayerService`, die der Web-Schicht alle gespeicherten Spieler zur Verfügung stellt, oder die `submitAssessments`-Methode des `RecapService`, die die vom Trainerteam vorgenommenen Bewertungen zur

Speicherung an das zuständige Repository weiterleitet.
Dabei ist es für eine ordnungsgemäße Funktionsweise der Anwendung unerlässlich, dass Benutzeranfragen mit ihren Daten korrekt weitergeleitet werden, daher wurde hier nicht nur getestet, ob die das Repository und die richtige Methode aufgerufen wird, sondern auch, ob Parameter wie erwartet übergeben werden. Während die Überprüfung der Parameterübergabe insbesondere bei zu speichernden Daten essenziell ist, muss bei Methoden, die Daten aus der Datenbank laden und diese bereitstellen insbesondere der Rückgabewert der Methode unter die Lupe genommen werden.
Darüber hinaus sind sowohl im Bereich der Score-Berechnung wie auch in der Spielzeitenplanung durch geeignete Testfälle geprüft worden, ob für eine bestimmte Eingabe das erwartete Ergebnis erzeugt wird. So kann sichergestellt werden, dass die einzelnen Services den Anforderungen entsprechen und die gewünschten Funktionalitäten bereitstellen.
Schließlich ist noch das Testing der Persistenz-Schicht veranschaulicht worden. Für eine realitätsnahe Testumgebung erwies sich die Java-Bibliothek `Testcontainers` als vorteilhaft, um das Verhalten der Produktivdatenbank möglichst genau simulieren zu können. Grundsätzlich sind die Datenbank-Testklassen nach dem gleichen Prinzip aufgebaut: Zunächst ist stets getestet worden, dass das Laden und Speichern einer Entität in der Datenbank korrekt implementiert ist. Neben den Standardmethoden zum Speichern und Laden von Entitäten sind darüber hinaus noch spezifischere Methoden mithilfe von `JDBCs Derived Queries` realisiert worden. Eine davon ist zum Beispiel eine Methode des `AssessmentRepository`, die Bewertungen nach Spieler, Kriterium und Datum filtert. Auch diese ist durch geeignetes Testing entwickelt worden, um sicherzustellen, dass die Methode für eine bestimmte Eingabe die erwartete Rückgabe liefert.
Rückblickend auf die Zeit der Entwicklung des Spielzeitenplaners bleibt festzuhalten, dass TDD ein mächtiges Werkzeug zur Entwicklung modularer Software ist. Jedoch soll dies keineswegs darüber hinwegtäuschen, dass es während des Entwicklungsprozesses auch schwierige Phasen gibt, in denen Geduld, Disziplin und Vertrauen gegenüber der testgetriebenen Herangehensweise an Softwareentwicklung auf die Probe gestellt werden. Insbesondere zeitlicher Druck oder die Schwierigkeit, in gewissen Situationen einen geeigneten Test zu schreiben, stehen der erfolgreichen Softwareentwicklung entgegen und können die konsequente Anwendung des TDD erschweren.
Abschließend ist festzuhalten, dass das Projekt `Spielzeitenplaner` nicht nur gezeigt hat, wie eine Anwendung von Beginn bis Ende testgetrieben entwickelt werden kann, sondern auch eine Lösung für die eingangs beschriebenen bekannten Probleme im Jugendfußball sein kann. Indem Kriterien erstellt und Spieler nach Trainings anhand dieser bewertet werden, wird einesystematische Reflexion im Trainerteam angeregt. Die Berechnung von Scores – basierend auf den Bewertungen – führt dazu, dass Spielzeiten für jeden einzelnen Spieler geplant und so eine möglichst faire Aufteilung gewährleistet wird. Diese systematische Spielzeitenplanung – kombiniert mit einer transparenten Kommunikation – kann möglichen Diskussionen mit Spielern und Eltern vorbeugen und dem Spieler transparent aufzeigen, wie er sich verbessern bzw. was er tun kann, um auf mehr Spielzeit zu kommen.
Die Anwendung in seiner gegenwärtigen Form stellt eine Art Grundversion mit den wichtigsten Features dar. Sie kann in den kommenden Jahren weiter ausgebaut und um einige nützliche Features erweitert werden. Diese beinhalten die Möglichkeit, mehrere Teams

zu erstellen, falls Fußballehrende Teil von mehr als nur einem Team sind, die dauerhafte Speicherung der tatsächlichen Spielzeiten über eine Saison hinweg, sodass nicht nur von Spiel zu Spiel geplant, sondern die Spielzeit auch über eine ganze Saison hinweg betrachtet werden kann oder aber eine `Drag and Drop`-Funktionalität bei der Anpassung der Startelf, um diesen Schritt für den Benutzer intuitiver zu gestalten.

Literatur

[Bec03] Beck, Kent: *Test-Driven Development: By Example*. Boston, MA, USA : Addison-Wesley, 2003

[Bor24] Borowiec, Rafał: *Thymeleaf Page Layouts*. https://www.thymeleaf.org/doc/articles/layouts.html. Version: 2024. – 19. November 2024

[Deu24] Deutscher Fussball-Bund (DFB): *Fussball-Regeln 2024/2025*. https://assets.dfb.de/uploads/000/309/175/original_308598-Regelheft24-25.pdf?1725278229. Version: 2024. – 19. November 2024

[Fus24] Fussballverband Niederrhein (FVN): *Durchfuehrungsbestimmungen für den Spielbetrieb der Juniorenspielklassen auf Kreisebene für die Saison 2024/2025*. https://fvn.de/media/duf__best_kreis.pdf. Version: 2024. – 19. November 2024

[Hed24] Hedley, Jonathan: *Use CSS selectors to find elements*. https://jsoup.org/cookbook/extracting-data/selector-syntax. Version: 2024. – 19. November 2024

[Pal08] Palermo, Jeffrey: *The Onion Architecture: part 1*. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/. Version: 2008. – 20. November 2024

[VMw24] VMware: *Annotation Interface WebMvcTest*. https://docs.spring.io/spring-boot/api/java/org/springframework/boot/test/autoconfigure/web/servlet/WebMvcTest.html. Version: 2024. – 19. November 2024