

Programmentwurf

Dart Counter

im Rahmen der Prüfung zum
Bachelor of Science (B.Sc.)

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Robin Purschwitz

Abgabedatum:	25. Mai 2023
Bearbeitungszeitraum:	12.12.2022 - 25.05.2023
Matrikelnummer, Kurs:	2415691, TINF20B2
Gutachter der Dualen Hochschule:	Dr. Lars Briem

Inhaltsverzeichnis

Abkürzungsverzeichnis	III
Abbildungsverzeichnis	IV
Tabellenverzeichnis	V
Quellcodeverzeichnis	VI
1 Einführung	1
1.1 Übersicht über die Applikation	1
1.2 Wie startet man die Applikation	1
1.3 Wie testet man die Applikation	2
2 Clean Architecture	3
2.1 Was ist Clean Architecture	3
2.2 Analyse der Dependency Rule	5
2.3 Positiv-Beispiel: Dependency Rule	6
2.4 Positiv-Beispiel: Dependency Rule	9
2.5 Analyse der Schichten	10
3 SOLID	12
3.1 Analyse Single-Responsibility-Principle (SRP)	12
3.2 Open Closed Principle (OCP)	15
3.3 Interface Segregation Principle (ISP)	16
4 Weitere Prinzipien	18
4.1 Analyse GRASP: Geringe Kopplung	18
4.2 Analyse GRASP: Hohe Kohäsion	20
4.3 Don't Repeat Yourself (DRY)	21
5 Unit Tests	23
5.1 10 Unit Tests	23
5.2 ATRIP: Automatic	25
5.3 ATRIP: Thorough	26
5.4 ATRIP: Professional	29
5.5 Code Coverage	31
5.6 Fakes und Mocks	34

6 Domain Driven Design	36
6.1 Ubiquitous Language	36
6.2 Entities	38
6.3 Value Objects	39
6.4 Repositories	39
6.5 Aggregates	39
7 Refactoring	40
7.1 Code Smells	40
7.2 2 Refactorings	44
8 Entwurfsmuster	48
8.1 Entwurfsmuster: Erbauer	48
8.2 Entwurfsmuster: Strategie	52

Literaturverzeichnis

VII

Abkürzungsverzeichnis

SRP	Single Responsibility Principle
OCP	Open-Closed Principle
ISP	Interface Segregation Principle
OCP	Open-Closed Principle
DRY	Don't Repeat Yourself
DDD	Domain Driven Design
IDE	Integrated Development Environment

Abbildungsverzeichnis

2.1	Clean Architecture Schichten[1]	4
2.2	POM Application layer	6
2.3	POM Adapters layer	7
2.4	Dart Game Handlers	8
2.5	Adapters und Domain Class in UML	9
2.6	Dart Klasse der Domain Schicht	10
2.7	HandleDart Klasse der Application Schicht	11
3.1	Leg Klasse	13
3.2	HandleLeg Klasse	14
3.3	MessagesDuringMatch Interface UML	16
3.4	Handle Interface UML	17
4.1	MatchHistory geringe Kopplung Positiv Beispiel	18
4.2	HandleThrow geringe Kopplung Negativ Beispiel	19
4.3	UserCommunicationService hohe Kohäsion Positiv Beispiel	20
4.4	HandleMatch und HandleSet DRY Positiv Beispiel (vorher)	21
4.5	HandleMatch und HandleSet DRY Positiv Beispiel (nacher)	22
5.1	UserInput UML	26
5.2	vollständig abgedeckte Methode <i>isValidDart</i>	27
5.3	UserCommunicationService UML	27
5.4	UserCommunicationService abgedeckter Code	29
5.5	<i>getUserInput</i> Test-Methode der Unit Test Klasse <i>UserCommunicationServiceTest</i>	31
5.6	Überblick über die Test Coverage	33
5.7	Übersicht der Mock-Klassen	34
6.1	Match UML	38
7.1	Extract Method in der <i>fillPlayerScoreAtRoundBeginMap</i> -Methode	44
7.2	UML vor Refactor	44
7.3	UML nach Refactor	45
7.4	UML nach Refactor	46
8.1	MatchBuilder	49
8.2	Implementierung der Handle Klasse im UML	52

Tabellenverzeichnis

5.1	Unit Tests 1-4	23
5.2	Unit Tests 5-8	24
5.3	Unit Tests 9-10	25
6.1	Ubiquitous Language	37

Quellcodeverzeichnis

5.1	UserInputTest isValidDart-Methode	26
5.2	UserCommunicationServiceTest Klasse	27
5.3	<i>createLeg</i> -Funktion der Unit Test Klasse <i>PlayerAverageCalculatorTest</i>	29
5.4	<i>players</i> -Liste der Unit Test Klasse <i>PlayerAverageCalculatorTest</i>	30
7.1	getPlayerAverageOfLeg-Methode vor Refactoring	40
7.2	getPlayerAverageOfLeg-Methode nach Refactoring	40
7.3	Methode <i>isMatchSetWon</i> der Elternklasse	42
7.4	Methode <i>isMatchSetWon</i> der HandleSet	43
7.5	Code vor Refactor	45
8.1	Match-Klasse	49
8.2	Handle-Klasse	52

1 Einführung

1.1 Übersicht über die Applikation

Dart ist ein beliebtes Geschicklichkeitsspiel, das sowohl als professioneller Sport als auch als geselliges Freizeitspiel gespielt wird. Ziel des Spiels ist es, Punkte zu sammeln, indem man Pfeile (*Darts*) auf ein kreisförmiges Dartboard wirft. Ein Standard-Dartboard ist in 20 nummerierte Sektionen unterteilt, wobei jeder Bereich unterschiedliche Punktzahlen vergibt. Darüber hinaus gibt es Doppel (Double) und Dreifach (Triple Felder), die den Wert der getroffenen Sektion verdoppeln bzw. verdreifachen. Das Feld dreifache 20 (*triple 20*) gibt die meisten Punkte. Es gibt verschiedene Spielvarianten, wobei 501 und 301 die bekanntesten sind. Bei diesen Varianten beginnen die Spieler mit einer festgelegten Punktzahl und müssen versuchen, ihre Punktzahl exakt auf Null zu reduzieren. Das Spiel erfordert Präzision, gute Hand-Auge-Koordination und strategisches Denken. Professionelle Dart-Turniere werden weltweit ausgetragen und ziehen Tausende von Zuschauern an. Der Dart-Counter ist eine Anwendung, die es einer Gruppe aus Spielern ermöglicht dieses Spiel zu spielen. Zwar müssen die Spieler die Darts immer noch händisch auf eine Dartscheibe werfen, jedoch können die Spieler so ihre Punkte zählen und wissen, wie viele Punkte ihnen noch zu einem Checkout fehlen. Darüber hinaus können die Spieler einen ständigen Überblick über ihren aktuellen Average (Durchschnitt) behalten und am Ende eines Legs (z. B. einer Runde 501) ihre Checkout-Quote (Trefferquote) sehen. Am Ende eines Gesamten Matches, welches aus mehreren Sets bestehen, welche wiederum aus mehreren Legs bestehen, haben die Spieler die Möglichkeit, ihren Spielverlauf in einer Textdatei zu speichern.

1.2 Wie startet man die Applikation

Zum starten der Applikation wird eine Java-Laufzeitumgebung und ein JDK (Version 19) benötigt. Zusätzlich wird eine Integrated Development Environment (IDE) zum Ausführen der Anwendung benötigt. Mit dieser IDE kann auch die Anwendung gebaut

werden, um das Ausführen ohne IDE zu ermöglichen. Gestartet werden muss die Klasse '*DartCounterV2/src/main/java/de/p3lina/Main.java*'. Alle Interaktionen mit der Anwendung finden dann in dem Terminal der IDE statt, mit welcher ausschließlich mit der Tastatur interagiert werden kann. Um ein eigentliches Match zu spielen, werden zuerst Spielinformationen, wie z. B. Anzahl der Spieler, Spielernamen, Startpunktanzahl etc. benötigt. Diese können, wie bereits erwähnt mit der Tastatur spezifiziert werden.

1.3 Wie testet man die Applikation

Das Testen der Applikation erfordert die Installation von Maven. Zum Testen können die Test-Klassen, welche sich in jedem Modul (application, domain etc.) unter '*src/main/test*' befinden, ausgeführt werden. Die Test-Klassen können mit der IDE ausgeführt werden. Um nicht alle Tests einzeln ausführen zu müssen, besteht auch die Möglichkeit mit dem Terminal in das Wurzelverzeichnis zu navigieren und den Befehl '*mvn test*' auszuführen.

2 Clean Architecture

In diesem Kapitel steht die Clean Architecture und deren zentrale Aspekte im Fokus. Zuerst erfolgt eine Analyse der Dependency Rule, einer Schlüsselregel der Clean Architecture. Untersucht werden dabei die Auswirkungen dieser Regel auf die Softwarearchitektur, ergänzt durch positive und negative Anwendungsbeispiele.

Im Anschluss daran richtet sich der Fokus auf die Struktur der Clean Architecture, wobei die einzelnen Schichten detailliert analysiert werden. Besondere Aufmerksamkeit gilt dabei den Schichten "Domain und Application", deren Rolle und Bedeutung innerhalb der Clean Architecture ausführlich diskutiert werden. Diese Analysen ermöglichen ein tiefgreifendes Verständnis der Clean Architecture und ihrer praktischen Anwendung.

2.1 Was ist Clean Architecture

Die **Clean Architecture**, auch bekannt als die **Onion Architecture**, ist ein Software-Entwurfsprinzip, das von Robert C. Martin entwickelt wurde. Sie zielt darauf ab, eine klare und getrennte Struktur in Software-Systemen zu schaffen, um Wartbarkeit, Testbarkeit und Flexibilität zu verbessern und somit den Rahmen für eine effiziente und zukunftssichere Softwareentwicklung zu schafft.

Die Struktur der Clean Architecture teilt eine Anwendung in konzentrische Schichten auf. Jede Schicht erfüllt bestimmte Arten von Aufgaben und weist klar definierte Abhängigkeiten auf. Dies ist ein entscheidender Aspekt, um eine hohe Entkopplung und Kohäsion innerhalb des Systems zu erreichen. Von innen nach außen sind die Schichten wie folgt strukturiert:

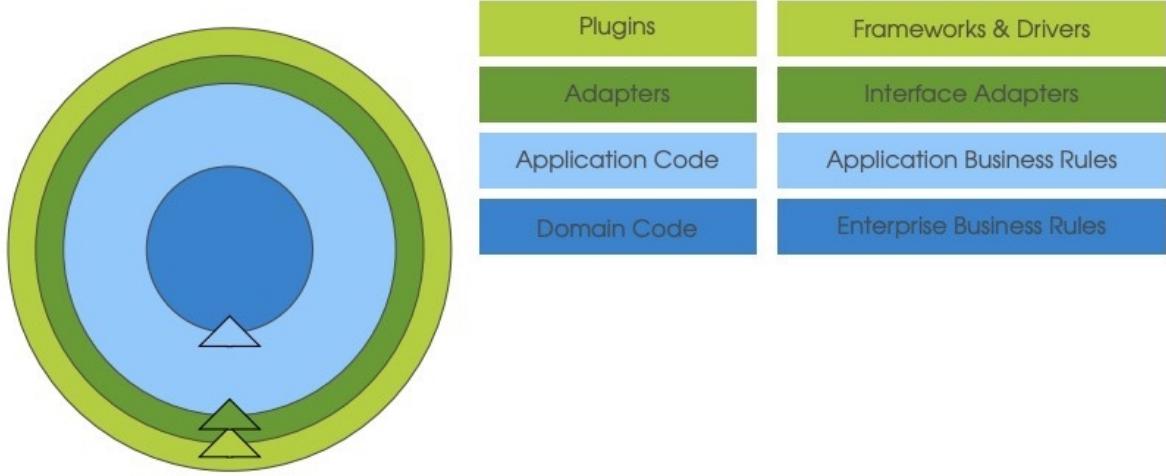


Abbildung 2.1: Clean Architecture Schichten[1]

Domain-Schicht: Die innerste Schicht, die die Geschäftslogik und Geschäftsregeln einer Anwendung enthält. Sie hat keine Abhängigkeiten von den äußeren Schichten und repräsentiert die fundamentalen Konzepte der Anwendung, unabhängig von spezifischen technologischen Details. Die Unabhängigkeit dieser Schicht ermöglicht eine hohe Testbarkeit und Flexibilität.

Anwendungs-Schicht: Diese Schicht enthält spezifische Geschäftslogik, die sich auf bestimmte Anwendungsfälle bezieht. Sie ist von der Domain-Schicht abhängig und kann mit ihr interagieren, kennt aber keine Details über äußere Schichten. Die Anwendungsschicht orchestriert die Interaktion zwischen der Domain-Schicht und den äußeren Schichten und stellt so eine Trennung der Belange sicher.

Adapter-Schicht: Sie dient der Übersetzung von Daten zwischen den für die inneren und äußeren Schichten geeigneten Formaten. Sie kann Datenbankcode, Benutzeroberflächen-Code oder sogar Code für externe Dienste enthalten. Sie fungiert als Brücke zwischen der inneren Logik und der Außenwelt.

Plugin-Schicht: Die äußerste Schicht, die spezifische Technologien wie Datenbanken, Webserver oder Frameworks umfasst. Sie interagiert mit den inneren Schichten durch Ports und Adapter. Diese Architektur ermöglicht eine einfache Ersetzbarkeit und Erweiterung von technologischen Komponenten.

Zusammenfassend lässt sich sagen, dass die Clean Architecture ein Ansatz ist, der darauf abzielt, die Unordnung und Komplexität in Softwareprojekten zu reduzieren, indem klare Grenzen und Regeln für die Struktur und Organisation des Codes vorgegeben werden. Sie ermöglicht es Entwicklern, Systeme zu erstellen, die widerstandsfähig gegenüber technologischen Änderungen sind und die sich im Laufe der Zeit leicht anpassen und erweitern lassen.

2.2 Analyse der Dependency Rule

Die Dependency Rule ist das zentrale Prinzip der Clean Architecture und bestimmt die Richtung der Abhängigkeiten zwischen den verschiedenen Schichten. Sie besagt, dass Abhängigkeiten immer von den äußeren Schichten zu den inneren Schichten gerichtet sein sollten, was bedeutet, dass der Code in den inneren Schichten frei von spezifischen technologischen Implementierungsdetails bleiben kann.

In der Praxis führt diese Regel dazu, dass der Code in den inneren Schichten unabhängig von spezifischen Technologien in den äußeren Schichten entwickelt und getestet werden kann. Dies erleichtert die Wartung des Codes und ermöglicht eine höhere Testabdeckung, da die Geschäftslogik unabhängig von spezifischen Technologien getestet werden kann.

Darüber hinaus fördert die Dependency Rule die Entkopplung der Schichten und trägt zur Flexibilität des Systems bei. Durch die eindeutige Trennung der Verantwortlichkeiten und die Richtung der Abhängigkeiten können Änderungen in den äußeren Schichten vorgenommen werden, ohne die inneren Schichten zu beeinflussen. Das System bleibt dadurch anpassungsfähig und widerstandsfähig gegenüber technologischen Änderungen.

Die Dependency Rule unterstützt auch die Anwendung der SOLID-Prinzipien, insbesondere das Single Responsibility Principle (SRP) und das Open-Closed Principle (OCP). Durch die klare Trennung der Verantwortlichkeiten zwischen den Schichten können diese ihre spezifischen Aufgaben erfüllen und gleichzeitig offen für Erweiterungen, aber geschlossen für Modifikationen bleiben. Dies führt zu einer verbesserten Modularität und Austauschbarkeit der Komponenten.

Abschließend kann man sagen, dass die Dependency Rule nicht nur dazu beiträgt, die Wartbarkeit und Testbarkeit der Software zu verbessern, sondern auch eine klare und

verständliche Struktur in der Codebasis fördert. Sie trägt zur Reduzierung der Komplexität bei, indem sie klare Grenzen und Regeln für die Struktur und Organisation des Codes vorgibt.

Die Clean Architecture und ihre Dependency Rule zielen insgesamt darauf ab, die Langlebigkeit und Widerstandsfähigkeit von Softwareprojekten zu verbessern. Sie ermöglicht es Entwicklern, Systeme zu erstellen, die sich an verändernde Anforderungen und Technologien anpassen können, ohne dass umfangreiche Überarbeitungen erforderlich sind. Durch die klare Strukturierung und Organisation des Codes können Entwickler effizienter arbeiten und gleichzeitig die Qualität ihrer Software sicherstellen.

Zusammenfassend lässt sich sagen, dass die Clean Architecture und insbesondere die Dependency Rule wesentliche Bestandteile einer modernen und zukunftssicheren Softwareentwicklung sind. Sie bieten Entwicklern ein solides Fundament, auf dem sie hochwertige, wartbare und flexible Softwarelösungen erstellen können, die den Anforderungen von heute und morgen gerecht werden.

Im Folgenden werden zwei Beispiele für die Einhaltung der Dependency Rule gezeigt.

2.3 Positiv-Beispiel: Dependency Rule

Im Projekt wurden die unterschiedlichen Schichten der Clean Architecture mit Modulen realisiert, welche von Maven verwaltet werden. Das hat den Vorteil, dass die inneren Schichten nicht auf die äußeren Schichten zugreifen können, sofern dieses Verhalten nicht explizit definiert ist.

```
<dependency>
    <groupId>de.p3lina</groupId>
    <artifactId>domain-3</artifactId>
    <version>1.0</version>
</dependency>
```

Abbildung 2.2: POM Application layer

```
<dependency>
    <groupId>de.p3lina</groupId>
    <artifactId>application-2</artifactId>
    <version>1.0</version>
</dependency>
<dependency>
    <groupId>de.p3lina</groupId>
    <artifactId>domain-3</artifactId>
    <version>1.0</version>
</dependency>
```

Abbildung 2.3: POM Adapters layer

In diesen beiden Code Ausschnitten ist zu sehen, wie die anderen Schichten in den Modulen importiert werden.

Die Application Schicht importiert ausschließlich die Domain Schicht, da nur die Domain Schicht weiter innen im Schichtenmodell ist.

Die Adapters Schicht importiert die Application Schicht und die Domain Schicht.

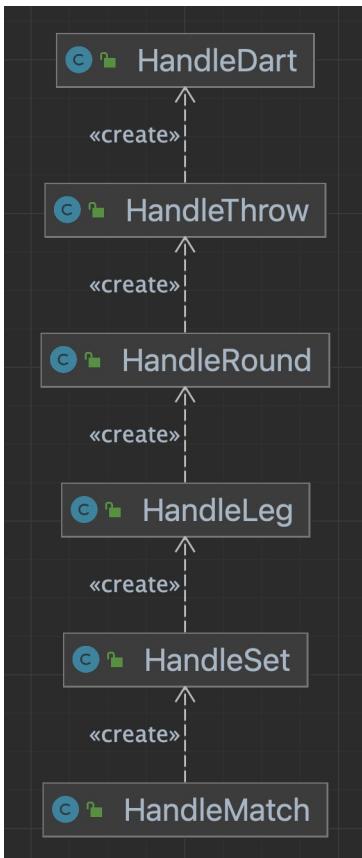


Abbildung 2.4: Dart Game Handlers

Als positiv Beispiel können die Handle Klassen für ein Dart Spiel verwendet werden. Diese dienen zum Verwalten der unterschiedlichen Teile eines Dart Spiels. Das Verhalten der Klassen ist prinzipiell immer gleich:

Die Klassen HandleMatch, HandleSet und HandleLeg bilden eine Struktur zur Organisation eines Dartspiels. Sie repräsentieren verschiedene Phasen des Spiels und sind für das Verwalten dieser Phasen zuständig.

HandleMatch ist für die Gesamtstruktur des Spiels verantwortlich. Sie erstellt ein Match-Objekt, das das gesamte Spiel repräsentiert. Während des Spiels wird jedes einzelne Set, das gespielt wird, von der HandleMatch Klasse an das Match-Objekt angefügt.

Diese Sets werden jedoch nicht direkt von HandleMatch erstellt. Stattdessen wird die Arbeit an die HandleSet-Klasse delegiert. HandleSet erstellt ein Set und verarbeitet alle

damit verbundenen Daten. Jedes gespielte Leg, ein kleinerer Abschnitt innerhalb des Sets, wird von HandleSet zum Set-Objekt hinzugefügt.

Bevor das Leg jedoch zum Set hinzugefügt wird, wird es von der HandleLeg-Klasse verarbeitet. HandleLeg ist dafür zuständig, ein Leg zu erstellen und zu verwalten.

Die Struktur geht noch tiefer: Jedes Leg besteht aus mehreren Runden, die von der HandleRound-Klasse verwaltet werden. HandleRound erstellt und verwaltet eine Runde, in der mehrere Würfe, sogenannte Throws, stattfinden können. Jeder dieser Throws wird von der HandleThrow-Klasse erstellt und verwaltet.

Schließlich kommt die HandleDart-Klasse ins Spiel. Sie kümmert sich um die tatsächlichen Würfe mit dem Dart, den kleinsten Einheiten im Spiel.

So entsteht eine klare Hierarchie und Struktur: Ein Match besteht aus mehreren Sets, die aus mehreren Legs bestehen. Jedes Leg beinhaltet mehrere Runden und in jeder Runde werden mehrere Würfe, also Throws, durchgeführt. Jeder einzelne Wurf mit dem Dart wird auch verwaltet. Jede dieser Ebenen wird von einer eigenen Handle-Klasse verwaltet. Durch diese strukturierte Aufteilung ist es möglich, ein Dart-Spiel effizient und organisiert zu verwalten.

Diese Klassen liegen in der Application Schicht und erstellen jeweils nur Objekte von Klassen der selben Schicht, womit die Dependency Rule eingehalten wird.

2.4 Positiv-Beispiel: Dependency Rule



Abbildung 2.5: Adapters und Domain Class in UML

In diesem Beispiel nutzt eine Klasse in der Adapter-Schicht ein Interface aus der Domain-Schicht. Dies ermöglicht es beispielsweise der Anwendungsschicht, ein Objekt dieser Klasse aus der Adapter-Schicht zu übergeben, da die Anwendungsschicht das Interface aus der Domain-Schicht kennt.

Der Vorteil dieser Vorgehensweise besteht darin, dass die Adapter-Schicht und die Domain-Schicht über das gemeinsame Interface miteinander kommunizieren können. Dadurch wird die Trennung zwischen den verschiedenen Schichten gewahrt und die Abhängigkeiten zwischen ihnen werden reduziert.

In der Anwendungsschicht kann ein Objekt der Adapter-Klasse erzeugt werden, da die Anwendungsschicht das Interface aus der Domain-Schicht kennt. Dies ermöglicht eine flexible Handhabung der Klassen und erleichtert die Integration von Komponenten aus verschiedenen Schichten.

2.5 Analyse der Schichten

In diesem Abschnitt werden 2 Klassen aus den jeweiligen Schichten analysiert.

2.5.1 Schicht: Domain

C 🔒 Dart		
m	↳ Dart(PossibleDarts)	
p	↳ dart PossibleDarts	
p	↳ doubleNumber boolean	
p	↳ points int	

Abbildung 2.6: Dart Klasse der Domain Schicht

Die Dart Klasse ist eine Klasse der Domain Schicht. Sie enthält die Attribute *dart* des Typs *PossibleDarts*, *doubleNumber* des Typs *boolean* und *points* des Typs *int*. Erstellt werden kann ein Dart-Objekt mit Hilfe eines *PossibleDarts*-Wertes. *PossibleDarts* ist ein

Enum der selben Schicht, welches alle möglichen Dart-Würfe enthält, zusammen mit der Punkt-Anzahl, die dieser Wurf erzielt und der Information, ob dieser Wurf ein Doppel ist oder nicht, da dies relevant für den Checkout ist.

Diese Klasse befindet sich in der Domain Schicht, da sie ein zentraler Bestandteil eines Dart-Spiels ist, von fast allen Schichten bekannt werden muss und sich deutlich seltener ändert als Klassen der anderen Schichten.

2.5.2 Schicht: Application

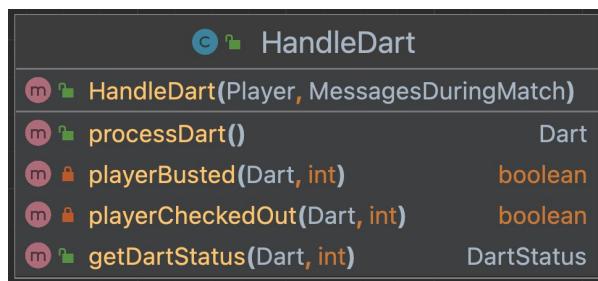


Abbildung 2.7: HandleDart Klasse der Application Schicht

Diese Klasse der Application-Schicht dient zum Verwalten eines Dart-Wurfes. Das Herzstück dieser Klasse ist die *processDart*-Methode. Diese wird vom überliegenden Teil des Dart Spiels, in diesem Fall der HandleThrow-Klasse oder genauer, der *processThrow*-Methode, aufgerufen und gibt ein Objekt der Dart-Klasse aus der Domain-Schicht zurück. Die anderen Methoden der Klasse dienen dazu, einen Spezialfall abzuprüfen, nämlich ob der Wurf eines Spielers ein checkout ist oder ob er damit überworfen (*engl. busted*) hat. Diese Klasse befindet sich in der Application-Schicht, da sie genau Regeln des Ablaufes eines Dart-Spiels implementiert. Solche Regeln können öfter geändert, angepasst und erweitert werden, weshalb diese Klasse in der Application-Schicht liegt.

3 SOLID

SOLID ist ein Akronym, das fünf grundlegende Prinzipien des objektorientierten Designs zusammenfasst. Diese Prinzipien dienen dazu, die Struktur und Flexibilität von Software zu verbessern und die Codequalität zu erhöhen.

1. Single Responsibility Principle (**SRP**): Eine Klasse sollte nur eine einzige Verantwortlichkeit haben und für eine spezifische Aufgabe zuständig sein.
2. Open-Closed Principle (**OCP**): Klassen sollten offen für Erweiterungen, aber geschlossen für Modifikationen sein, indem sie neue Funktionen hinzufügen, ohne den bestehenden Code zu ändern.
3. Liskov Substitution Principle (**LSP**): Subtypen sollten sich genauso verhalten wie ihre Basistypen, um eine nahtlose Austauschbarkeit zu gewährleisten.
4. Interface Segregation Principle (**ISP**): Schnittstellen sollten spezifisch auf die Bedürfnisse der Clients zugeschnitten sein, sodass sie nur von den Methoden abhängig sind, die sie tatsächlich benötigen.
5. Dependency Inversion Principle (**DIP**): Abhängigkeiten sollten auf abstrakte Konzepte oder Schnittstellen, nicht auf konkrete Implementierungen, ausgerichtet sein, um die Flexibilität und Austauschbarkeit von Komponenten zu fördern.

3.1 Analyse Single-Responsibility-Principle (SRP)

Das Single Responsibility Principle (SRP) besagt, dass eine Klasse nur eine einzige Verantwortlichkeit haben sollte. Sie sollte für eine spezifische Aufgabe oder Funktion zuständig sein. Dadurch wird sichergestellt, dass die Klasse nur für einen bestimmten Aspekt der Funktionalität verantwortlich ist und sich nicht mit mehreren unterschiedlichen Aufgaben befasst. Das SRP ermöglicht eine bessere Lesbarkeit, Wartbarkeit und Testbarkeit des Codes, da jeder Verantwortungsbereich in einer separaten Klasse organi-

siert ist. Durch die Einhaltung des SRP können Änderungen oder Erweiterungen in einem Bereich vorgenommen werden, ohne dass andere Bereiche der Klasse betroffen sind.

3.1.1 Positiv Beispiel

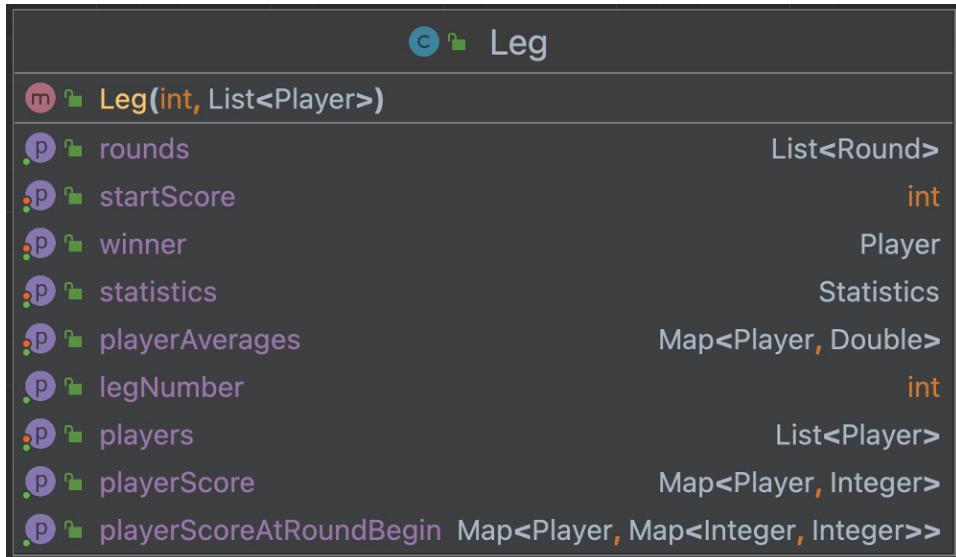


Abbildung 3.1: Leg Klasse

Diese Klasse hält das SRP ein, da sie nur die Aufgabe hat, Informationen über ein Leg eines Dart Spiels zu speichern. Zur Initialisierung des Legs werden 2 Informationen benötigt. Die Spieler, welche am Leg teilnehmen und die Anzahl der zu erreichenden Punkte. Während der Initialisierung wird das Attribut *playerscore* deklariert und mit einer *Map* initialisiert. Die *Map* ermöglicht eine Zuordnung zwischen den Spielern und ihren Scores. Im *statistics*-Objekt werden die Statistiken (Averages und checkout Quote) pro Spieler gespeichert. Zuvor war es nur möglich averages (im *playerAverages*-Objekt) zu speichern. Dieses Attribut kann in den nächsten Updates der Applikation gelöscht werden. Die *legNumber* speichert die Nummer des Legs, welche beispielsweise für die Ausgabe in der Konsole (*Leg Nummer 3 wird nun gespielt...*) verwendet. *playerScoreAtRoundBegin* speichert den Score des Spielers zu Beginn einer Runde. Dieses Attribut wird benötigt, um die Checkout Quote zu berechnen. Das *rounds*-Attribut speichert die verschiedenen Runden des Legs und das *winner*-Attribut speichert den Gewinner eines Legs.

3.1.2 Negativ Beispiel

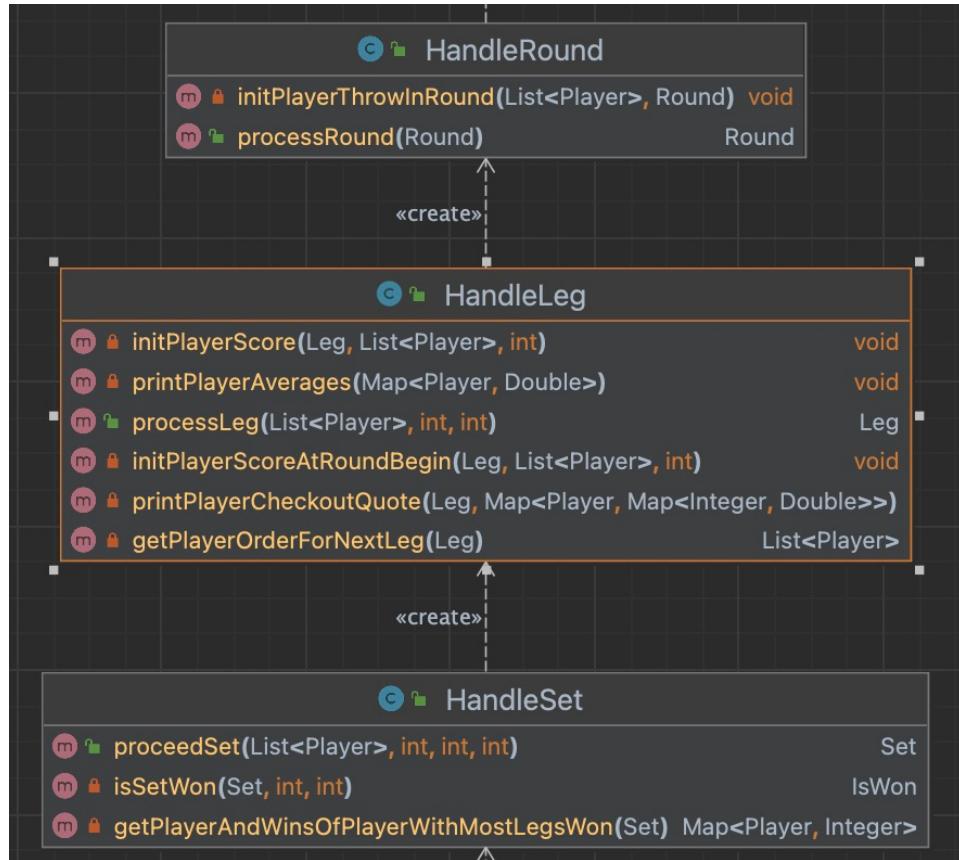


Abbildung 3.2: HandleLeg Klasse

Die HandleLeg-Klasse hält das SRP nicht vollständig ein, da sie eigentlich nur Logik für das Abarbeiten eines Legs beinhalten sollte. Die Logik des Legs befindet sich, wie auch bei den anderen Handle-Klassen, in der HandleLeg-Klasse, welche ein Leg-Objekt zurückgibt. Die Existenz der init-Methoden in der Klasse könnte auch schon als Argument genommen werden, um zu begründen, dass die Klasse das SRP nicht einhält, da diese Methoden auch in eine Klasse *Setup* oder *Init* ausgelagert werden könnten. Die beiden print-Methoden verletzen das SRP jedoch deutlicher, da diese wenig mit dem Abarbeiten des Leg-Prozesses zu tun haben und ausgelagert werden können.

3.2 Open Closed Principle (OCP)

3.2.1 Positiv Beispiel

In der Leg-Klasse wird ein Statistics-Objekt verwendet, um verschiedene statistische Metriken zu speichern. Dies beinhaltet sowohl Durchschnittswerte, gespeichert in der *averages*-Map, als auch Checkout-Informationen, die in der *checkout*-Map gespeichert sind. Der Einsatz dieses Statistics-Objekts fördert die Einhaltung des Open-Closed-Prinzips (OCP).

Da das Statistics-Objekt für die Speicherung aller statistischen Werte verantwortlich ist, können neue Statistiken hinzugefügt oder vorhandene geändert werden, ohne die Leg-Klasse selbst zu modifizieren.

Sollte also eine Erweiterung um zusätzliche Statistiken erforderlich sein, würde dies lediglich eine Anpassung oder Erweiterung des Statistics-Objekts bedeuten. Die Leg-Klasse bleibt dadurch unverändert, wodurch das Risiko von Fehlern oder unerwarteten Seiteneffekten in diesem Teil des Systems minimiert wird. Dies unterstreicht, wie das Statistics-Objekt dazu beiträgt, das Open-Closed-Prinzip in der Anwendung einzuhalten.

3.2.2 Negativ Beispiel

Als negativ Beispiel können die verschiedenen Handle-Klassen betrachtet werden. Wenn beispielsweise ein weiterer Spielmodus hinzugefügt werden soll, muss jede Handle-Klasse angepasst werden. Beispielsweise ist das Kriterium für einen Checkout ein anderer bei bspw. Around the Clock als beim klassischen 501. Zum tracken der Punkte reicht auch keine normale Map mehr, sondern es wird eine separate Datenstruktur benötigt. Alle diese Änderungen würden dazu führen, dass die Handle-Klassen verändert werden müssten. Dies verstösst gegen das Open-Closed Principle (OCP).

3.3 Interface Segregation Principle (ISP)

3.3.1 Positiv Beispiel

Interface Segregation Principle (ISP)

Interface Segregation Principle (ISP): Das Interface Segregation Principle (ISP) ist ein Prinzip in der objektorientierten Programmierung, das besagt, dass keine Klasse aufgrund ihrer Schnittstellen von Methoden abhängig sein sollte, die sie nicht verwendet. Es gehört zu den fünf Prinzipien des SOLID-Ansatzes für das Software-Design. Mit anderen Worten, es ist besser, viele spezifische Schnittstellen zu haben als eine allgemeine; dies verhindert, dass eine Änderung in einer nicht verwendeten Methode einer Klasse zu einem unerwünschten Seiteneffekt in einer anderen Klasse führt. Das ISP hilft, den Code wartbarer und einfacher zu verstehen zu machen, indem es verhindert, dass Klassen mit ungenutzten Methoden überladen werden. Kurz gesagt, es fördert die Entkopplung von Software-Modulen und verbessert deren Flexibilität und Wiederverwendbarkeit.

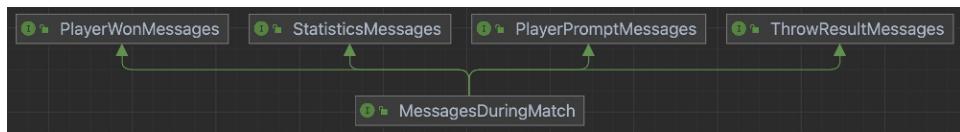


Abbildung 3.3: `MessagesDuringMatch` Interface UML

Das Interface `MessagesDuringMatch` ist nicht monolithisch gestaltet, sondern in kleinere Interfaces unterteilt, die jeweils unterschiedliche Nachrichten während des Spielverlaufs repräsentieren. Diese Struktur entspricht dem Prinzip der Interface Segregation Principle (ISP). Im Falle, dass eine Klasse nur einen bestimmten Nachrichtentyp, wie zum Beispiel Statistikausgaben, benötigt, ermöglicht diese Struktur die Implementierung lediglich des relevanten Interfaces. Auf diese Weise wird eine effiziente und zielgerichtete Architektur erreicht.

3.3.2 Negativ Beispiel



Abbildung 3.4: Handle Interface UML

Das Handle-Interface verstößt gegen das ISP, da es Methoden enthält, die nicht von allen Implementierungen benötigt werden. Konkret werden zwei Methoden bereitgestellt, wobei die erste Methode in allen Implementierungen genutzt wird, die zweite jedoch lediglich in zwei von sechs Implementierungen. Dies führt zu einer unnötigen Abhängigkeit für die vier Implementierungen, die die zweite Methode nicht benötigen. Eine Lösung dieses Problems könnte in der Aufteilung des Handle-Interfaces in zwei separate Interfaces bestehen, wobei jedes Interface genau eine der beiden Methoden enthält. Alternativ könnte das Interface auch vollständig aufgelöst und durch andere Mechanismen ersetzt werden, um den Anforderungen des ISP gerecht zu werden.

4 Weitere Prinzipien

4.1 Analyse GRASP: Geringe Kopplung

Geringe Kopplung ist ein Prinzip von GRASP, das auf die Minimierung der Abhängigkeiten zwischen Klassen oder Modulen abzielt. Es ist eine grundlegende Strategie zur Steigerung der Modularität und Wiederverwendbarkeit von Software. Bei geringer Kopplung sind einzelne Komponenten weniger auf die internen Eigenschaften und das Verhalten anderer Komponenten angewiesen. Dies erleichtert das Verständnis des Systems, da jede Komponente einzeln verstanden werden kann. Zudem wird dadurch die Wartbarkeit verbessert, da Änderungen an einer Komponente weniger wahrscheinlich Auswirkungen auf andere haben. Geringe Kopplung erleichtert auch das Testen von Komponenten, da diese unabhängig voneinander getestet werden können.

4.1.1 Positiv Beispiel

MatchHistory		
m	MatchHistory()	
m	getMatchHistoryString(Match)	String
m	saveMatchHistory(Match, MessagesOutsideMatch)	void
m	getThrowString(Throw, Player, Leg, Round)	String
m	getRoundString(Round, Leg)	String
m	legLegCheckoutQuoteString(Leg)	String
m	getSetString(Set)	String
m	getLegString(Leg)	String
m	isCheckoutThrow(Leg, Player, Round)	boolean
m	getPlayerAverages(Leg)	String
m	breakLineIfNotEqual(int, int)	String

Abbildung 4.1: MatchHistory geringe Kopplung Positiv Beispiel

Die Klasse *MatchHistory* ist ein gutes Beispiel für geringe Kopplung in der Softwareentwicklung. Sie hat keine Abhängigkeiten zu anderen Klassen und kann somit unabhängig existieren und betrieben werden. Durch ihre Eigenständigkeit kann sie in jedem Moment des Programms ausgeführt oder ignoriert werden. Zum Beispiel kann der Benutzer am Ende des Programms entscheiden, ob die Match History gespeichert werden soll oder nicht. Ihre Funktionalität ist durch öffentliche Methoden wie *getMatchHistoryString(Match)* und *saveMatchHistory(Match, MessagesOutsideMatch)* sowie private Methoden definiert, die spezifische Aufgaben erfüllen. Alle diese Methoden sind speziell auf die Verarbeitung und Speicherung von Match-Daten zugeschnitten, und benötigen keine Kenntnisse oder Interaktionen mit anderen Teilen des Systems. Dies trägt dazu bei, die Kopplung zu minimieren und die Wartbarkeit und Erweiterbarkeit des Codes zu verbessern.

4.1.2 Negativ Beispiel

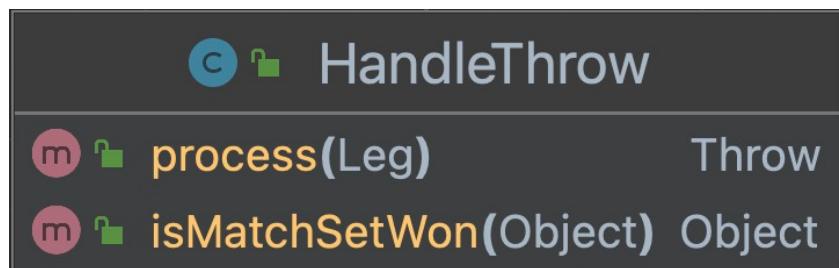


Abbildung 4.2: HandleThrow geringe Kopplung Negativ Beispiel

Die *HandleThrow* Klasse zeigt einen deutlichen Mangel an geringer Kopplung. Sie ist stark an mehrere andere Klassen gebunden, einschließlich *Player*, *Leg*, *MessagesDuringMatch*, *Throw*, *HandleDart*, *Dart* und *DartStatus*. Insbesondere führt die *process()* Methode Änderungen am Zustand von *Player*, *Leg* und *Dart*-Instanzen durch und ist somit stark an das Innendrama dieser Klassen gekoppelt. Änderungen an diesen Klassen könnten dazu führen, dass auch *HandleThrow* angepasst werden muss. Außerdem verlässt sie sich auf die *HandleDart* Klasse für das Verarbeiten eines Dart-Wurfs, was zusätzliche Abhängigkeiten schafft. Insgesamt fehlt dieser Klasse die Unabhängigkeit und Flexibilität, die durch geringe Kopplung erreicht werden könnte, was potenzielle Probleme bei der Wartung und Erweiterung des Codes verursachen könnte. Es könnte Abstraktion auf die *HandleThrow*-Klasse eingesetzt werden. Dieses Prinzip bietet erhebliche Vorteile in Bezug auf Kapselung und Informationssicherheit. Anstatt direkt auf die Zustände der

Player, *Leg* und *Dart* Klassen zuzugreifen, könnte *HandleThrow* durch Implementierung von Interfaces oder abstrakten Klassen mit diesen kommunizieren.

4.2 Analyse GRASP: Hohe Kohäsion

Hohe Kohäsion bezieht sich auf das GRASP-Prinzip, das besagt, dass Klassen oder Module klar definierte, eng miteinander verbundene Verantwortlichkeiten haben sollten. Dieses Prinzip zielt darauf ab, dass die innerhalb einer Klasse oder eines Moduls gruppierten Funktionen stark miteinander verbunden sein sollten, um die Stabilität, Zuverlässigkeit und Verständlichkeit des Systems zu verbessern. Hohe Kohäsion erleichtert auch die Wartung und Erweiterung des Systems, da die Auswirkungen von Änderungen innerhalb einer Klasse auf andere Klassen minimiert werden. Sie fordert auch die Wiederverwendbarkeit von Klassen, da diese spezifische und gut definierte Aufgaben haben.

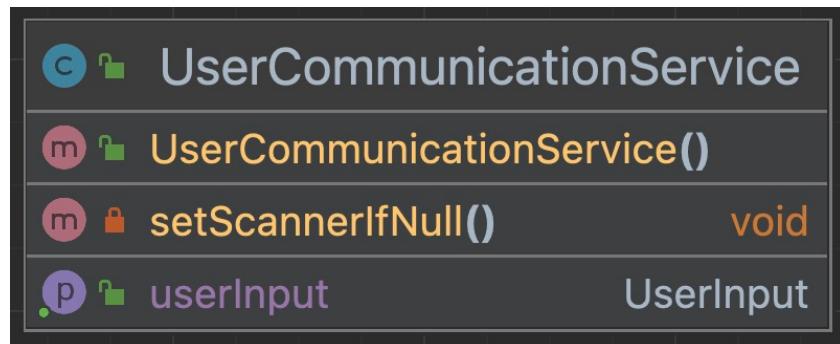


Abbildung 4.3: UserCommunicationService hohe Kohäsion Positiv Beispiel

Die Klasse *UserCommunicationService* illustriert exemplarisch das Konzept der hohen Kohäsion in der objektorientierten Softwareentwicklung. Ihr alleiniger Zweck besteht in der Verarbeitung von Benutzereingaben, welche sie in ein entsprechendes *UserInput*-Objekt umwandelt und zurückgibt. Zusätzlich initialisiert diese Klasse, falls nicht bereits geschehen, einen global verfügbaren *Scanner* in einer dafür bestimmten Speicherklasse. Dieses Design wurde in Anbetracht zukünftiger Erweiterungen gewählt, bei denen weitere Klassen den Scanner nutzen könnten. Zudem optimiert es die aktuelle Version im Hinblick auf Testbarkeit, indem es die Simulation von Benutzereingaben mittels Mocking erleichtert.

4.3 Don't Repeat Yourself (DRY)

Don't Repeat Yourself (DRY) ist ein grundlegendes Prinzip in der Softwareentwicklung, das die Eliminierung von Redundanz fördert. Das Ziel ist, dass jede Information oder jedes Verhalten im System nur an einer Stelle definiert sein sollte. Dies verringert die Fehleranfälligkeit, da bei Änderungen oder Korrekturen nur eine Stelle im Code angepasst werden muss. Es verbessert auch die Wartbarkeit, da weniger Code zu warten ist und das System einfacher zu verstehen ist. DRY fördert zudem die Konsistenz im System, da die Wahrscheinlichkeit von Inkonsistenzen durch mehrfache Definitionen reduziert wird.

Das folgende Beispiel bezieht sich auf den Commit **270423b**

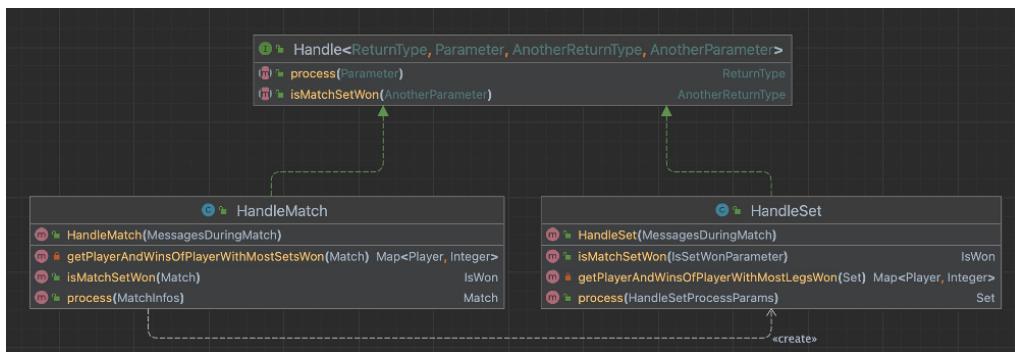


Abbildung 4.4: HandleMatch und HandleSet DRY Positiv Beispiel (vorher)

Wie in Abbildung 4.4 zu sehen ist, enthalten die Klassen *HandleMatch* und *HandleSet* die nahezu gleichen Methoden *isMatchSetWon* und *getPlayerAndWinsOfPlayerWithMostSets/LegsWon*. Diese beiden Methoden sind bei beiden Klassen nahezu identisch, womit knapp 50 Zeilen an Code-Duplikat existieren.

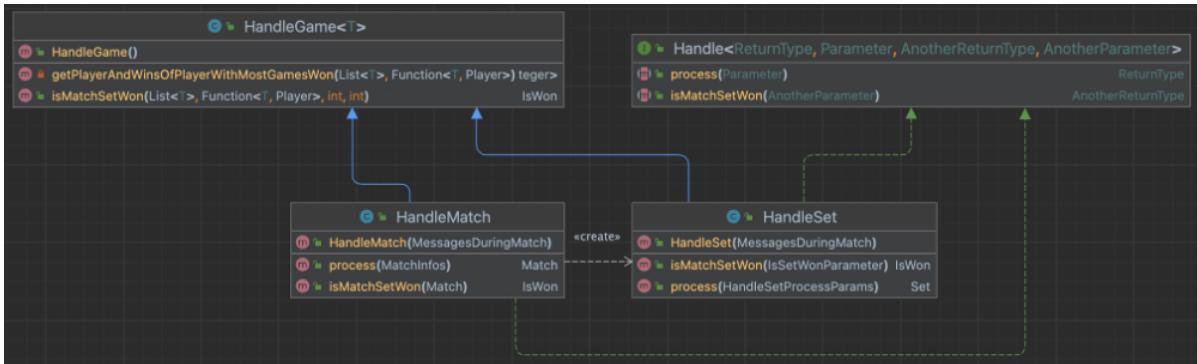


Abbildung 4.5: HandleMatch und HandleSet DRY Positiv Beispiel (nacher)

Abbildung 4.5 zeigt, wie das Problem der doppelten Codepassagen gelöst wurde. Eine neue Klasse wurde implementiert, die als Elternklasse für die beiden vorherigen Klassen dient. Diese übergeordnete Klasse enthält die beiden Methoden, die vorher in beiden Klassen doppelt vorhanden waren.

Mit der Entfernung dieser doppelten Codepassagen lässt sich die Anzahl der Codezeilen um 50 Zeilen reduzieren. Dies trägt zur Lesbarkeit des gesamten Codes bei. Zudem zentralisiert diese Lösung die entsprechenden Methoden an einem Ort. Das erleichtert die Wartung, da bei Änderungen nur an einer Stelle Anpassungen vorgenommen werden müssen.

5 Unit Tests

5.1 10 Unit Tests

Unit Test	Beschreibung
1 processTest: HandleDartTest	Der Test <i>processTest()</i> überprüft, ob die <i>process()</i> -Methode der Klasse <i>HandleDart</i> die Eingabe des Benutzers korrekt verarbeitet und die erwarteten Dartpunkte zurückgibt. Der Test wird fünfmal wiederholt, um mehrere Eingaben zu simulieren und die Konsistenz der Ergebnisse zu überprüfen.
2 playerDartStatusTest: HandleDartTest	Der Test <i>playerDartStatusTest()</i> überprüft, ob die Methode <i>getDartStatus()</i> der Klasse <i>HandleDart</i> den korrekten Dartstatus basierend auf dem aktuellen Punktestand und dem geworfenen Dart zurückgibt. Der Test verwendet verschiedene Dart-Eingaben und überprüft, ob der zurückgegebene Dartstatus den erwarteten Werten entspricht.
3 getPlayerAverageOf RoundTest:Player AverageCalculatorTest	Der Test <i>getPlayerAverageOfRoundTest()</i> überprüft, ob die Methode <i>getPlayerAverageOfRound()</i> des <i>PlayerAverageCalculator</i> die durchschnittlichen Punkte eines Spielers in einer Runde korrekt berechnet und zurückgibt.
4 getPlayerAverageOf LegTest:PlayerAverage CalculatorTest	Der Test <i>getPlayerAverageOfLegTest()</i> überprüft, ob die Methode <i>getPlayerAverageOfLeg()</i> des <i>PlayerAverageCalculator</i> die durchschnittlichen Punkte eines Spielers über alle Runden in einem <i>Leg</i> korrekt berechnet und zurückgibt.

Tabelle 5.1: Unit Tests 1-4

Unit Test	Beschreibung
5 getPlayersAveragesOfLegTest:PlayerAverageCalculatorTest	<p>Der Test <i>getPlayersAveragesOfLegTest()</i> überprüft, ob die Methode <i>getPlayersAveragesOfLeg()</i> des <i>PlayerAverageCalculator</i> die durchschnittlichen Punkte aller Spieler in einem Leg als Map zurückgibt und ob die berechneten Durchschnittswerte den erwarteten Werten entsprechen.</p>
6 getPlayerCheckoutQuoteOfLegTest:PlayerCheckoutQuoteCalculatorTest	<p>Der Test <i>getPlayerCheckoutQuoteOfLegTest()</i> überprüft, ob die Methode <i>getPlayerCheckoutQuoteOfLeg()</i> des <i>PlayerCheckoutQuoteCalculator</i> den Checkout-Anteil (Prozentsatz des erfolgreichen Checkouts im Vergleich zu den möglichen Checkouts) eines bestimmten Spielers in einem Leg korrekt berechnet und zurückgibt. Der Test überprüft, ob der berechnete Anteil mit dem erwarteten Wert übereinstimmt.</p>
7 getPlayersCheckoutQuoteOfLegTest:PlayerCheckoutQuoteCalculatorTest	<p>Der Test <i>getPlayersCheckoutQuoteOfLegTest()</i> prüft, ob die Methode <i>getPlayersCheckoutQuoteOfLeg()</i> des <i>PlayerCheckoutQuoteCalculator</i> eine Map zurückgibt, die die Checkout-Prozentsätze aller Spieler in einem Leg enthält. Der Test überprüft, ob die berechneten Anteile für jeden Spieler mit den erwarteten Werten übereinstimmen.</p>
8 getUserInputTest:UserCommunicationServiceTest	<p>Der Test <i>getUserInputTest()</i> überprüft, ob die Methode <i>getUserInput()</i> des <i>UserCommunicationService</i> die Benutzereingabe korrekt abruft und als <i>UserInput</i>-Objekt zurückgibt. Dabei wird eine vordefinierte Zeichenkette als simulierter Benutzereingabe verwendet, und der Test vergleicht, ob die Rückgabewerte der Methode für jede Zeile der simulierten Eingabe den erwarteten Werten entsprechen.</p>

Tabelle 5.2: Unit Tests 5-8

Unit Test	Beschreibung
9 isValidDartValid CaseTest:UserInputTest	Der Test <i>isValidDartValidCaseTest()</i> überprüft, ob die Methode <i>isValidDart()</i> der Klasse <i>UserInput</i> den Wert <i>true</i> zurückgibt, wenn die Eingabe ein gültiger Dartwert ist. In diesem Fall wird überprüft, ob <i>SBull</i> als gültiger Dartwert erkannt wird.
10 prepareUserDartInput SBullTest:UserInputTest	Der Test <i>prepareUserDartInputSBullTest()</i> überprüft, ob die Methode <i>prepareUserDartInput()</i> der Klasse <i>UserInput</i> die Benutzereingabe <i>25</i> korrekt in den Dartwert <i>SBull</i> umwandelt und als <i>UserInput</i> -Objekt zurückgibt. Dabei wird überprüft, ob das umgewandelte Objekt den erwarteten Wert <i>SBull</i> hat.

Tabelle 5.3: Unit Tests 9-10

5.2 ATRIP: Automatic

1. Einfache Ausführung: Die Unit Tests werden ausgeführt und die Ergebnisse werden in der Konsole ausgegeben.
2. Automatisches Ablaufen: Durch die Verwendung eines global zugänglichen Scanners, kann die Systemeingabe vom Test gesetzt werden und durch den öffentlichen Scanner immer wieder weitere Zeilen aufgerufen werden.
3. Selbstüberprüfung: Die Unit Tests liefern durch Asserts, wie *AssertEquals* oder *AssertTrue* immer nur das Ergebnis Bestanden / Nicht Bestanden.

5.3 ATRIP: Thorough

Positiv Beispiel:

In dieser Klasse wurden nur die Methoden *isValidDart* und *prepareUserDartInput* aktiv

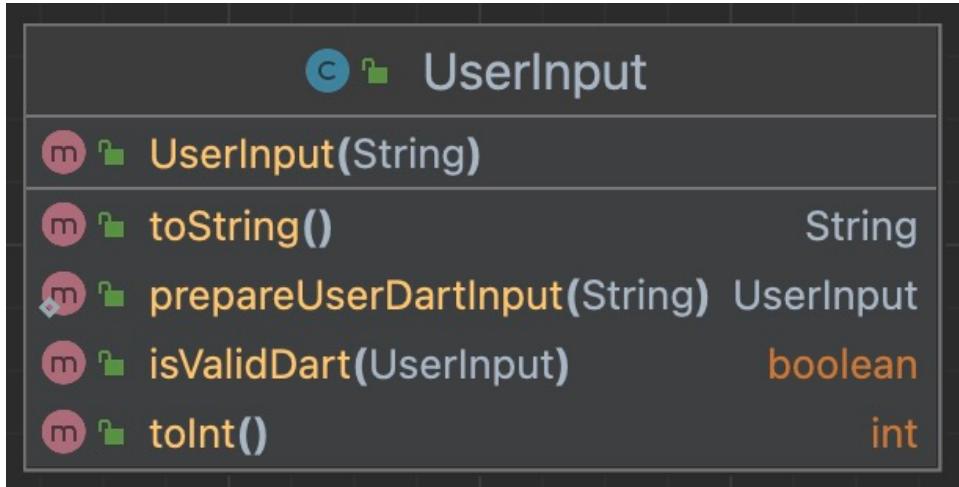


Abbildung 5.1: UserInput UML

getestet. Es wurde sich dafür entschieden, da nur diese kritische und eventuell fehlerbehaftete Teile des Systems sind. Die Methoden *toInt* und *toString* sind hingegen kleine Hilfsmethoden, deren falsche Implementierung einen ebenso negativen Effekt auf die Anwendung hätte, jedoch sind diese Vergleichbar mit einer *Getter*-und *Setter*-Methode und sind somit trivial.

```

1  @Test
2      public void isValidDartValidCaseTest() {
3          UserInput input = new UserInput("SBull");
4          assertTrue(input.isValidDart(input));
5      }
6
7  @Test
8  public void isValidDartInvalidCaseTest() {
9      UserInput input = new UserInput("NotInPossibleDarts");
10     assertFalse(input.isValidDart(input));
11 }

```

Listing 5.1: UserInputTest *isValidDart*-Methode

Dieser Code Ausschnitt zeigt zwei Test-Methoden für die *isValidDart*-Methode. Dadurch werden beide Fälle der Methode abgedeckt: Ein gültiger Dart und ein ungültiger Dart.

```
public boolean isValidDart(UserInput userInput) {
    try {
        PossibleDarts.valueOf(userInput.toString());
        return true;
    }catch(IllegalArgumentException exc) {
        return false;
    }
}
```

Abbildung 5.2: vollständig abgedeckte Methode *isValidDart*

Wie auf diesem Bild zu sehen ist, wurde die Methode *isValidDart* vollständig abgedeckt. Dies bedeutet, dass alle möglichen Pfade der Methode durchlaufen wurden.

Negativ Beispiel:

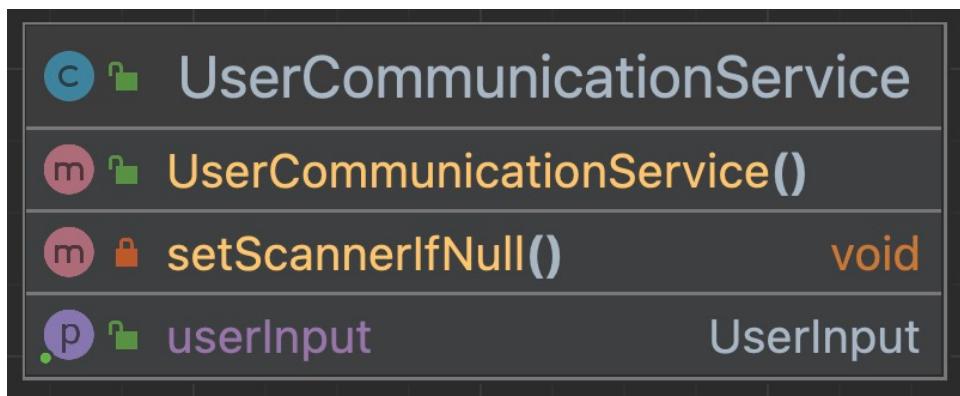


Abbildung 5.3: UserCommunicationService UML

```

1  @BeforeEach
2      public void setup() {
3          userCommunicationService = new ↴
4              ↴ UserCommunicationService();
5      }
6  @Test

```

```
7 public void getUserInputTest() {  
8     String testInput = "T20\nS20\nD20\n";  
9     InputStream in = new ↵  
    ↴ ByteArrayInputStream(testInput.getBytes());  
10  
11     System.setIn(in);  
12  
13     ScannerStore.setScanner(new Scanner(System.in));  
14  
15  
16     String[] input = testInput.split("\n");  
17     for(String inputLine : input) {  
18         UserInput userInput = ↵  
    ↴ userCommunicationService.getUserInput();  
19         assertEquals(inputLine, userInput.toString());  
20     }  
21  
22  
23     System.setIn(System.in);  
24 }
```

Listing 5.2: UserCommunicationServiceTest Klasse

```

public UserInput getUserInput() {
    setScannerIfNull();
    Scanner scanner = ScannerStore.getScanner();
    try {
        String s = scanner.nextLine();
        UserInput userInput = new UserInput(s);
        return userInput;
    } catch(Exception exc) {
    }
    return new UserInput("-1");
}

private void setScannerIfNull() {
    if(ScannerStore.getScanner()==null) {
        ScannerStore.setScanner(new Scanner(System.in));
    }
}

```

Abbildung 5.4: UserCommunicationService abgedeckter Code

Dieses Beispiel bezieht sich auf die Klasse *UserCommunicationService*, welche für die Benutzerinteraktion verantwortlich ist. Sie verfügt über eine Methode zur Erfassung der Benutzereingaben und zur Umwandlung dieser in ein *UserInput*-Objekt. Obwohl der Code innerhalb des *try*-Blocks normalerweise keine Probleme verursachen sollte, muss dennoch ein Verhalten der Anwendung definiert sein, das im Fehlerfall ausgeführt werden kann. Sollte es unerwartet zu einem Fehler kommen und dieses Verhalten ausgeführt werden, würde der Test fehlschlagen und das Verhalten der Anwendung wäre unvorhersehbar. Daher erfüllt dieser Test nicht das Kriterium Thorough aus dem ATRIP-Prinzip, da er einen wesentlichen Teil des Systems, auf dem die Anwendung basiert, nicht abdeckt.

5.4 ATRIP: Professional

Positiv Beispiel:

```

1 private Leg createLeg(List<Player> players, Round round) {
2     Leg leg = new Leg(1, players);
3     leg.addRound(round);
4     return leg;

```

```
5 }
```

Listing 5.3: *createLeg*-Funktion der Unit Test Klasse *PlayerAverageCalculatorTest*

In der Unit Test Klasse *PlayerAverageCalculatorTest* wurde die Methode *createLeg* implementiert. Diese Methode wird in der *setup()*-Methode aufgerufen. Durch die Auslagerung des Codes in diese Methode, kann der Code für zukünftige Tests wiederverwendet werden.

```
1 players = Arrays.asList(new Player("Player1"), new ↴  
↳ Player("Player2"));
```

Listing 5.4: *players*-Liste der Unit Test Klasse *PlayerAverageCalculatorTest*

Zuvor wurden 2 *Player*-Objekte initialisiert, doch um die Erweiterbarkeit der Klasse zu verbessern wurden diese 2 Objekte in eine Liste aus Spielern gespeichert. Dadurch kann in Zukunft die Anzahl der Spieler erhöht werden, ohne dass der Code der Unit Test Klasse angepasst werden muss.

Negativ Beispiel:

```
@Test
public void getUserInputTest() {
    String testInput = "T20\nS20\nD20\n";
    InputStream in = new ByteArrayInputStream(testInput.getBytes());

    System.setIn(in);

    ScannerStore.setScanner(new Scanner(System.in));

    String[] input = testInput.split( regex: "\n");
    for(String inputLine : input) {
        UserInput userInput = userCommunicationService.getUserInput();
        assertEquals(inputLine, userInput.toString());
    }

    System.setIn(System.in);
}
```

Abbildung 5.5: *getUserInputTest*-Methode der Unit Test Klasse *UserCommunicationServiceTest*

Als Negativ Beispiel könnte die Methode *getUserInputTest* der Unit Test Klasse *UserCommunicationServiceTest* dienen. Diese Methode ist für einen Entwickler, der nicht an der Entwicklung der Anwendung beteiligt war, nicht verständlich. Dies liegt daran, dass alle Operationen in einer Klasse geschehen. Durch die Einführung von Methoden, die den Code in kleinere Teile aufteilen, könnte die Lesbarkeit des Codes und die Wiederverwendbarkeit von Code-Teilen verbessert werden.

5.5 Code Coverage

Im Rahmen des Testens der Applikation wurde eine gezielte Vorgehensweise bei der Erstellung der Unit-Tests verfolgt. Es wurde dabei berücksichtigt, dass nicht für jede

Klasse Tests notwendig sind, sondern ein Fokus auf jene gelegt wurde, die für den Verlauf der Anwendung als relevant erachtet wurden.

Die Konzentration lag insbesondere auf Klassen oder Methoden, die durch ihre Komplexität oder Wichtigkeit in der Anwendung hervorstechen. Ein Beispiel hierfür ist die Berechnung des Player-Average, welche eine komplexere Berechnung in vielen Teilen darstellt und daher explizit getestet wurde.

Aus diesem Grund wurden ausschließlich Tests für die Application-Layer entwickelt, da sie eine entscheidende Rolle für den Verlauf der Anwendung spielt. Die in dieser Schicht enthaltenen Klassen und Methoden haben direkte Auswirkungen auf den Anwendungsverlauf, weshalb ihre korrekte Funktion von großer Bedeutung ist.

Trotz alleinigen Testens der Application-Layer, blieb die Domain-Layer nicht unberachtet. Viele der Klassen und Methoden dieser Schicht wurden passiv durch die Unit-Tests in der Application-Layer abgedeckt. Dies hat dazu geführt, dass die Domain-Layer eine Line-Coverage von knapp 50% erreicht hat.

Zusammengefasst lässt sich sagen, dass durch diese Vorgehensweise eine gute und effiziente Testabdeckung erreicht wurde, die einen guten Überblick über den Zustand und die Qualität der Anwendung gibt.

Element ▾	Class, %	Method, %	Line, %	Branch, %
de	46% (15/32)	31% (51/160)	37% (229/610)	41% (62/150)
p3lina	46% (15/32)	31% (51/160)	37% (229/610)	41% (62/150)
application	53% (8/15)	38% (23/60)	32% (122/380)	42% (62/146)
handle	14% (1/7)	17% (5/29)	12% (24/200)	27% (19/68)
MatchHistory	0% (0/1)	0% (0/12)	0% (0/72)	0% (0/26)
PlayerAverageCalculate	100% (1/1)	100% (3/3)	86% (25/29)	75% (9/12)
PlayerCheckoutQuoteC	100% (3/3)	100% (7/7)	97% (45/46)	75% (18/24)
ScannerStore	100% (1/1)	100% (2/2)	100% (2/2)	100% (0/0)
UserCommunicationSet	100% (1/1)	100% (2/2)	80% (8/10)	100% (2/2)
UserInput	100% (1/1)	80% (4/5)	85% (18/21)	100% (14/14)
domain	41% (7/17)	28% (28/100)	46% (107/230)	0% (0/4)
i18n	0% (0/1)	0% (0/2)	0% (0/22)	100% (0/0)
Messages	0% (0/1)	0% (0/2)	0% (0/22)	100% (0/0)
messages	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
MessagesDuringMatch	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
MessagesOutsideMatch	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
PlayerPromptMessage	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
PlayerWonMessages	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
StatisticsMessages	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ThrowResultMessage	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
Dart	100% (1/1)	75% (3/4)	85% (6/7)	100% (0/0)
DartStatus	100% (1/1)	100% (2/2)	100% (4/4)	100% (0/0)
Handle	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
HandleLegProcessParallel	0% (0/1)	0% (0/7)	0% (0/10)	100% (0/0)
HandleSetProcessParallel	0% (0/1)	0% (0/9)	0% (0/13)	100% (0/0)
IsSetWonParameter	0% (0/1)	0% (0/7)	0% (0/10)	100% (0/0)
IsWon	0% (0/1)	0% (0/4)	0% (0/7)	100% (0/0)
Leg	100% (1/1)	55% (11/20)	65% (17/26)	100% (0/0)
Match	0% (0/1)	0% (0/9)	0% (0/14)	100% (0/0)
MatchInfos	0% (0/1)	0% (0/6)	0% (0/11)	100% (0/0)
Player	100% (1/1)	100% (2/2)	100% (3/3)	100% (0/0)
PossibleDarts	100% (1/1)	75% (3/4)	97% (67/69)	100% (0/0)
Round	100% (1/1)	100% (4/4)	100% (6/6)	100% (0/0)
Set	0% (0/1)	0% (0/8)	0% (0/14)	0% (0/4)
Statistics	0% (0/1)	0% (0/4)	0% (0/4)	100% (0/0)
Throw	100% (1/1)	50% (3/6)	57% (4/7)	100% (0/0)
ThrowStatus	0% (0/1)	0% (0/2)	0% (0/3)	100% (0/0)

Abbildung 5.6: Überblick über die Test Coverage

Abbildung 5.6 gibt einen Überblick über die Abdeckung der implementierten Tests.

5.6 Fakes und Mocks

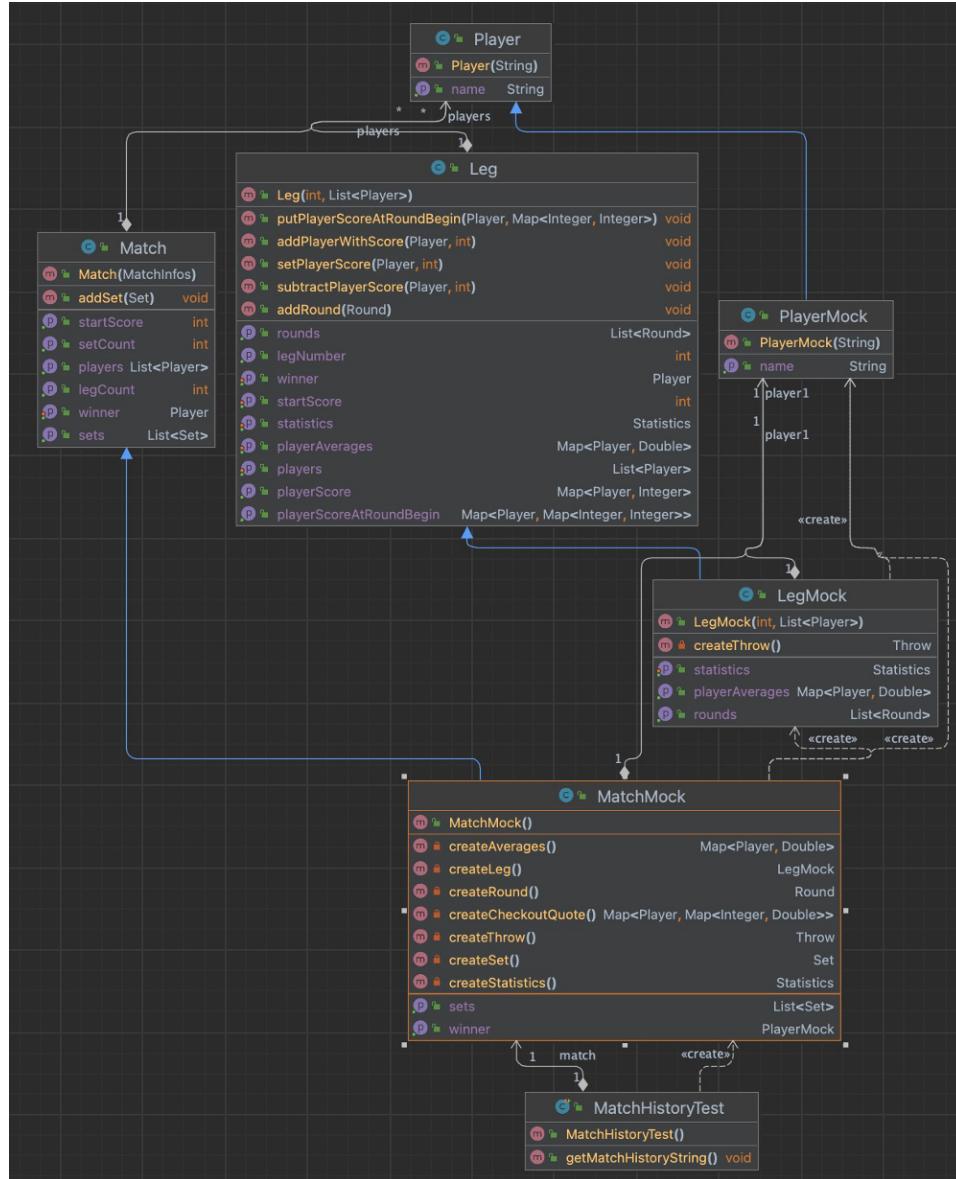


Abbildung 5.7: Übersicht der Mock-Klassen

In der Unit Test Klasse `MatchHistoryTest` werden zwei Fake- bzw. Mock-Objekte verwendet, um die Funktionalität der `getMatchHistoryString` Methode in der `MatchHistory` Klasse zu testen. Die zwei Mock-Objekte, `PlayerMock` und `MatchMock`, simulieren das Verhalten realer Objekte innerhalb eines isolierten Testumfelds.

PlayerMock dient dazu, das Verhalten eines *Player*-Objekts zu imitieren, insbesondere im Hinblick auf die Rückgabe eines spezifischen Spielernamens. Auf der anderen Seite stellt *MatchMock* ein komplexeres Szenario dar, in dem ein vollständiges Dartspiel mit Sets, Legs, Runden und Würfen simuliert wird.

Der Einsatz dieser Mock-Objekte ist wichtig, da es das Testen der *getMatchHistoryString* Methode ermöglicht, ohne von den tatsächlichen Implementierungen der *Player* und *Match* Klassen abhängig zu sein. Dies ist besonders nützlich, da die realen Klassen, vor allem die *Match*-Klasse, komplex sind.

Diese Mock-Objekte erlauben das Erstellen von vorhersehbaren Testszenarien. In diesem Fall kann garantiert werden, dass der *getMatchHistoryString* Methode immer das gleiche Match mit den gleichen Spielern übergeben wird, wodurch der erwartete Ausgabestring konstant bleibt.

6 Domain Driven Design

6.1 Ubiquitous Language

Es wurde sich bereits vor der Implementierung des Projektes gedanken über die Ubiquitous Language gemacht, wie in dem Commit **0005472** zu sehen ist. Für diese Vorgehensweise wurde sich bewusst entschieden, da die Namensgebung aller Klassen, Objekte, Variablen, Nachrichten usw. auf den Namen der Ubiquitous Language basieren.

Beispiele der Ubiquitous Language können auf der nächsten Seite gesehen werden.

Bezeichnung	Bedeutung	Begründung
1 Match	Ein Match ist das gesamte Spiel zwischen mehreren Spielern. Ein Spiel besteht aus mindestens einem Set. Ein Set besteht aus mindestens einem Leg. Ein Leg besteht aus mindestens einer Runde.	Vor allem für Dart-Neulingen könnten Dart Begriffe nicht geläufig sein und daher könnte ein Neuling davon ausgehen, dass ein Match gleich ein Leg ist.
2 Score	Der Score ist der Punktestand eines Spielers in einem Leg. Im klassischen Dart liegt er somit zwischen 0 und 501 Punkten.	Für den Score muss auch ein bestimmtes Wort festgelegt werden, damit die Verwendung von verschiedenen Begriffen, wie z.B. Points nicht zu Verwirrungen führen.
3 Dart	Ein Dart ist ein Wurf auf die Dartscheibe. Ein Dart besteht aus einem Wert des Darts (z. B. 60) und der Information, ob dieser Dart ein Doppelfeld getroffen hat, da dies wichtig ist für das Beenden eines Legs.	Es musste ein Begriff für einen einzelnen Dart festgelegt werden, damit er nicht mit dem Throw verwechselt wird.
4 Throw	Ein Throw besteht aus drei geworfenen Darts auf die Dartscheibe.	Dieser Begriff wurde eingeführt, damit er nicht mit dem Einzelnen Dart-Wurf (Dart) verwechselt wird.

Tabelle 6.1: Ubiquitous Language

6.2 Entities

Eine Entity sei definiert über Identität, Lebenszyklus und Verhalten.

Entity Match:

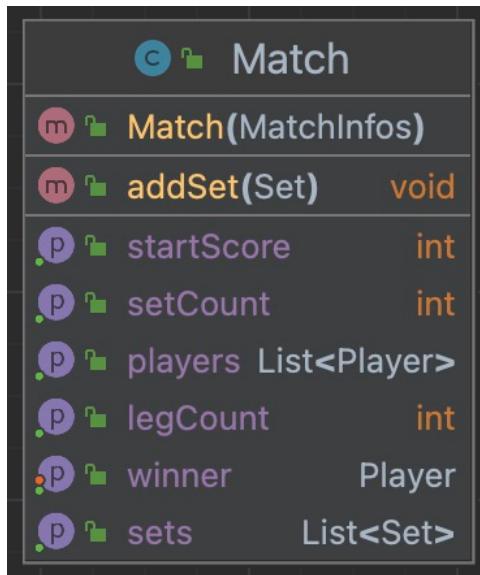


Abbildung 6.1: Match UML

- **Identität:** Jedes Match ist eindeutig und kann anhand einer Identität unterschieden werden. Auch wenn die Attribute (Spieler, Sets, etc.) des Matches sich ändern könnten, bleibt die Identität des Matches gleich. Die Identität könnte implizit durch eine Kombination der Attribute oder explizit durch eine eindeutige ID repräsentiert werden, falls nötig.
- Ein Match hat einen klaren Lebenszyklus und einen Zustand, der sich im Verlauf der Applikation ändert. Es wird initialisiert, Sets werden hinzugefügt, und schließlich wird ein Gewinner festgelegt.
- Das Match hat Methoden, die das Verhalten darstellen, wie ein Match manipuliert oder geändert werden kann.

6.3 Value Objects

In dieser Anwendung wurden keine vollwertigen Value Objects verwendet, da der Einsatz dieser nicht notwendig ist. Value Objects werden oft für elementare Werte verwendet, wie Geld oder eine Adresse, wobei solche Konzepte in der Anwendung nicht vorkommen. Jedoch könnte ein abgewandeltes Objekt der Dart-Klasse als Value Object angesehen werden, da ein Dart Wurf insbesondere über sein Attribut *points* definiert wird und somit zwei Dart Objekte mit der gleichen points-Anzahl als gleich angesehen werden können.

6.4 Repositories

Repositories sind besonders nützlich, wenn wiederholte Zugriffe auf bestimmte Objekte erforderlich sind, da sie das Definieren von spezifischem Suchverhalten für diese Objekte ermöglichen.

In dieser Anwendung wird immer nur am Ende eines Legs oder Sets auf andere Objekte zugegriffen. Zum Beispiel wird am Ende eines Legs auf die vorherigen Legs zugegriffen, um einen eventuellen Gewinner des Sets zu ermitteln. Da diese Zugriffe so selten geschehen, ist der Einsatz eines Repositories überflüssig und würde nur unnötig die Komplexität erhöhen.

6.5 Aggregates

Aggregates in Domain Driven Design (DDD) sind Gruppen von assoziierten Objekten, die als Einheit behandelt werden, um die Konsistenz und Integrität der Geschäftsdaten zu gewährleisten. Jedes Aggregate hat eine sogenannte Root-Entität, die als Einstiegspunkt für die Interaktion mit dem Aggregate dient und die Kontrolle über den Lebenszyklus der inneren Objekte hat. Zugriffe auf innere Objekte eines Aggregats erfolgen in der Regel über die Root-Entität. So wird sichergestellt, dass Geschäftsregeln eingehalten werden und die Datenkonsistenz gewährleistet ist. Aggregates helfen, die Komplexität im Design zu reduzieren und die Isolation zwischen verschiedenen Teilen des Systems zu fördern. In diesem Projekt wurde kein spezifisches Aggregate implementiert, da der Entwurf so gewählt wurde, dass jedes Objekt für seine eigene Konsistenz zuständig ist.

7 Refactoring

7.1 Code Smells

Code Smell 1 (Long Method):

```

1 public Double getPlayerAverageOfLeg(Player player, Leg leg) {
2     double playerAveragePL;
3     double sumPlayerAveragesPerRound = 0.0;
4     int totalRoundCount = ↴
5         ↴ (leg.getRounds().get(leg.getRounds().size()-1).getRoundNumber());
6     for(Round round : leg.getRounds()) {
7         Double playerAverageOfRound = ↴
8             ↴ getPlayerAverageOfRound(round, player);
9         if(playerAverageOfRound== -1.0) {
10             totalRoundCount--;
11             continue;
12         }
13         sumPlayerAveragesPerRound += playerAverageOfRound;
14     }
15     if(totalRoundCount==0) {
16         return 0.0;
17     }
18     playerAveragePL = sumPlayerAveragesPerRound / ↴
19         ↴ totalRoundCount;
20     return playerAveragePL;
21 }
```

Listing 7.1: getPlayerAverageOfLeg-Methode vor Refactoring

```

1 public Double getPlayerAverageOfLeg(Player player, Leg leg) {
2     int totalRoundCount = getRoundNumberOfLastRound(leg);
3     if(totalRoundCount==0) {
```

```

4         return 0.0;
5     }
6     double sumPlayerAveragesPerRound = ↴
7         ↪ getSumOfPlayerAveragesOfRounds(leg.getRounds(), ↴
8             ↪ player);
9
10    return sumPlayerAveragesPerRound / totalRoundCount;
11
12}
13
14
15
16 private Double getSumOfPlayerAveragesOfRounds(List<Round> ↴
17     ↪ rounds, Player player) {
18     double sumPlayerAveragesPerRound = 0.0;
19     for(Round round : rounds) {
20         Double playerAverageOfRound = ↴
21             ↪ getPlayerAverageOfRound(round, player);
22         if(playerAverageOfRound===-1.0) {
23             break;
24         }
25         sumPlayerAveragesPerRound += playerAverageOfRound;
26     }
27     return sumPlayerAveragesPerRound;
28 }
```

Listing 7.2: getPlayerAverageOfLeg-Methode nach Refactoring

Die Methode *getPlayerAverageOfLeg* wurde so überarbeitet, dass sie klarer und einfacher zu verstehen ist. Dies wurde durch die Extraktion von Teilen des Codes in separate, private Hilfsmethoden erreicht, die jeweils eine spezifische Aufgabe erfüllen.

Die Methode **getRoundNumberOfLastRound**: Diese Methode nimmt als Eingabe ein Leg und gibt die Rundennummer der letzten Runde zurück. Sie kapselt den Teil des ursprünglichen Codes, der die Gesamtzahl der Runden ermittelt.

Die Methode **getSumOfPlayerAveragesOfRounds**: Diese Methode nimmt als Eingabe eine Liste von Round-Objekten und einen Player und gibt die Summe der Durchschnittswerte des Spielers pro Runde zurück. Sie kapselt den Teil des ursprünglichen Codes, der durch die Runden iteriert, um den Durchschnitt des Spielers für jede Runde zu berechnen.

Die Hauptmethode **getPlayerAverageOfLeg** wurde nun so vereinfacht, dass sie leichter zu verstehen ist. Sie ruft nun einfach die Hilfsmethoden auf und führt eine Berechnung durch, um den Durchschnittswert des Spielers für das Leg zu ermitteln. Dies bietet die Vorteile Wartbarkeit, Lesbarkeit, Testbarkeit und Wiederverwendbarkeit.

Code Smell 2 (Duplicated Code):

Die Klassen *HandleMatch* und *HandleSet* weisen beide eine Methode auf, die prüft, ob ein Spieler entweder ein komplettes Spiel oder ein Set gewonnen hat. Diese Methoden sind in ihrer Struktur und Logik sehr ähnlich. Der einzige Unterschied besteht darin, dass die Methode in *HandleMatch* die unterschiedlichen Sets und deren Gewinner berücksichtigt, während die Methode in *HandleSet* die verschiedenen Legs eines Sets und deren Gewinner in Betracht zieht. Ansonsten sind beide Methoden nahezu identisch. Dies stellt einen deutlichen Fall von doppeltem Code dar, durch dessen Vermeidung etwa 50 Zeilen Code eingespart und die Lesbarkeit, Wiederverwendbarkeit sowie Testbarkeit des Codes verbessert werden könnten.

Zur Lösung dieses Problems mit doppeltem Code wurde die Klasse *HandleGame* eingeführt. Diese Klasse enthält die duplizierten Methoden und dient als Elternklasse für *HandleMatch* und *HandleSet*.

```

1 public IsWon isMatchSetWon(List<T> games, Function<T, Player> ↵
    ↵ getWinner, int gameCount, int playerCount) {
2     IsWon isWon = new IsWon();
3     Map<Player, Integer> ↵
        ↵ playerAndPlayerWinsWithMostGamesWon = ↵
        ↵ getPlayerAndWinsOfPlayerWithMostGamesWon(games, ↵
        ↵ getWinner);
4     Player currentWinner = (Player) ↵
        ↵ playerAndPlayerWinsWithMostGamesWon.keySet().toArray()[0];
5     int currentWinnerGamesWon = ↵
        ↵ playerAndPlayerWinsWithMostGamesWon.get(currentWinner);
6     int gamesNeedToWin = gameCount / playerCount + 1;

```

```
7         if (currentWinnerGamesWon < gamesNeedToWin) {  
8             return isWon;  
9         }  
10        isWon.setPlayer(currentWinner);  
11        return isWon;  
12    }
```

Listing 7.3: Methode isMatchSetWon der Elternklasse

Die Methode in Listing 7.3 wurde wie bereits erwähnt in die Elternklasse HandleGame ausgelagert, womit dieser duplizierte Code gespart wurde.

```
1 public IsWon isMatchSetWon(IsSetWonParameter ↴  
2     ↴ isSetWonParameter) {  
3     Set set = isSetWonParameter.getSet();  
4     int legCount = isSetWonParameter.getLegCount();  
5     int playerCount = isSetWonParameter.getPlayerCount();  
6     return super.isMatchSetWon(set.getLegs(), ↴  
7         ↴ Leg::getWinner, legCount, playerCount);  
8 }
```

Listing 7.4: Methode isMatchSetWon der HandleSet

Die Methode in Listing 7.4 ist nun die neue Methode in HandleSet. Die Klasse HandleSet erbt von der Klasse HandleGame, womit die gleiche Funktionalität, wie vor dem Refactoring erreicht wurde.

7.2 2 Refactorings

Refactoring 1 (Extract Method):

Dieses Beispiel bezieht sich auf Commit f50cfec.

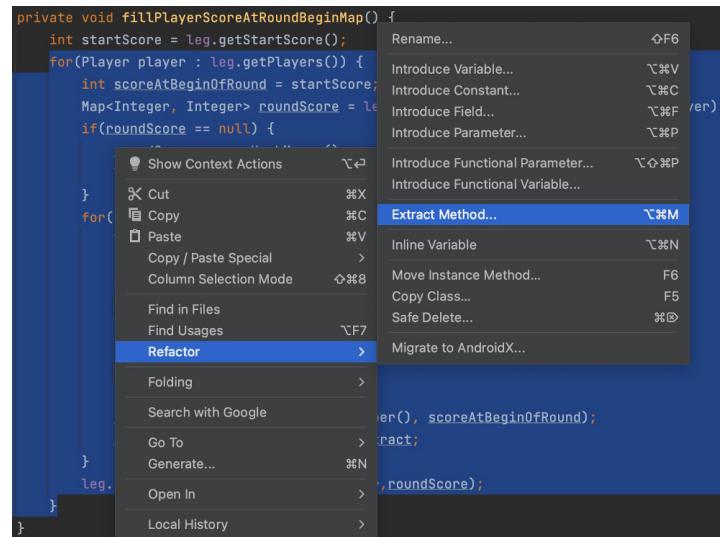


Abbildung 7.1: Extract Method in der *fillPlayerScoreAtRoundBeginMap*-Methode

Wie bereits in Abbildung 7.1 in Teilen zu sehen ist, ist die vorgestellte Methode sehr lang und damit unübersichtlich, weswegen sich entschieden wurde, diese Methode in mehrere kleinere, übersichtlichere Methode aufzuteilen.

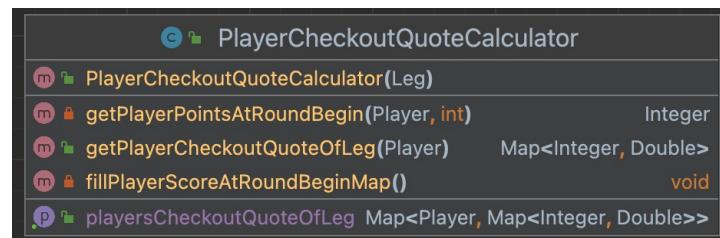


Abbildung 7.2: UML vor Refactor

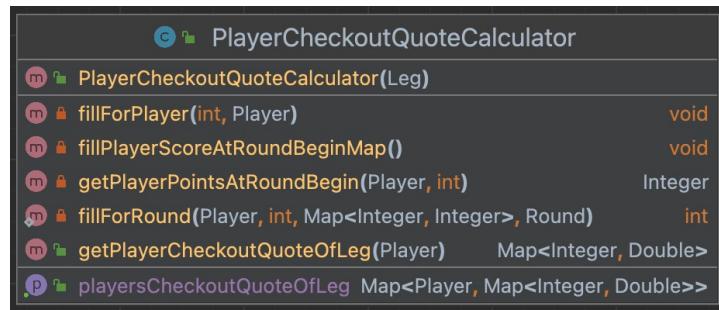


Abbildung 7.3: UML nach Refactor

Wie in Abbildung 7.2 und Abbildung 7.3 zu sehen, wurden 2 neue Methoden hinzugefügt, welche jeweils Verantwortung für einen Teil der ursprünglichen Methode übernehmen. Die Methode *fillPlayerScoreAtRoundBeginMap* ist nun übersichtlicher und leichter zu verstehen.

Refactoring 2 (Replace Conditional with Polymorphism):

Dieses Beispiel bezieht sich auf Commit 479ad5e

```

1 public static UserInput prepareUserDartInput(String ↵
    ↵ userInputString) {
2     if(userInputString.length()==0) {
3         return new UserInput(userInputString);
4     }
5     if(Character.isDigit(userInputString.charAt(0)) && ↵
        ↵ userInputString.length()==1 && ↵
        ↵ Integer.parseInt(userInputString)==0) {
6         return new UserInput("Zero");
7     }
8     if(!Character.isDigit(userInputString.charAt(0))) {
9         return new UserInput(userInputString.substring(0, ↵
            ↵ 1).toUpperCase() + ↵
            ↵ userInputString.substring(1));
10    }
11    if(userInputString.equals("25")) {
12        return new UserInput("SBull");
13    }

```

```

14     if(userInputString.equals("50")) {
15         return new UserInput("DBull");
16     }
17     return new UserInput("S" + userInputString);
18 }

```

Listing 7.5: Code vor Refactor

Wie in Listing 7.5 zu sehen ist, verwendet die Methode 5 if-Statements und einen Default Fall. Dieses if-Statement könnte im Verlauf der Applikation erweitert werden oder es könnten Statements gelöscht oder erweitert werden. Eine hohe Anzahl von if-Zweigen erhöht die Komplexität einer Methode, was das Lesen, Verstehen und Pflegen des Codes erschwert.

Wenn neue Fälle berücksichtigt werden sollen, erfordert dies zusätzliche Arbeit und birgt das Risiko, Fehler zu erzeugen. Zudem kann die Einfügung neuer Bedingungen an irgendeiner Stelle im if-Konstrukt unerwartete Auswirkungen auf die restliche Logik haben.

Zusätzlich kann die Testbarkeit der Methode leiden, da jeder if-Zweig individuell getestet werden muss. Dies erhöht die Anzahl der benötigten Testfälle und die Zeit, die für das Schreiben und Ausführen der Tests benötigt wird. Deswegen wurde sich entschieden, dieses if-Statement durch Polymorphismus zu ersetzen.

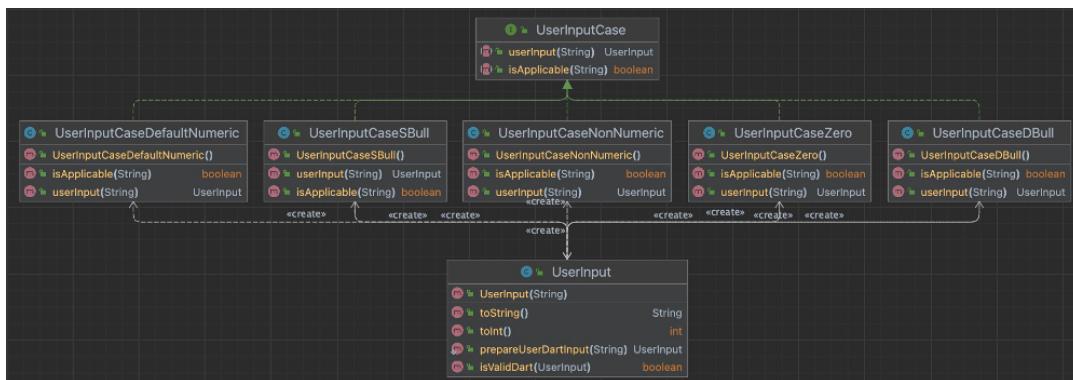


Abbildung 7.4: UML nach Refactor

Wie nun in Abbildung 7.4 zu sehen ist, wurden 6 neue Klassen eingeführt. *UserInputCase* ist dabei das Interface, auf dem die anderen Fälle aufbauen und welches die benötigten

Methoden für Zutrefflichkeit und Konsens bereitstellt. Diese 5 Klassen implementieren das Interface und überschreiben die Methoden. Die Klasse *UserInputCaseDefault* ist dabei beispielsweise der Default Fall, welcher die Logik des Default Falls aus der ursprünglichen Methode übernimmt.

8 Entwurfsmuster

8.1 Entwurfsmuster: Erbauer

Die Erzeugung eines Match-Objektes wurde mit Hilfe eines Erbauers realisiert. Dies ermöglichte eine erhöhte Lesbarkeit und Verständlichkeit: Durch den Einsatz des Builder-Musters wurde der Code lesbarer und einfacher zu verstehen, insbesondere da die Match-Klasse viele Parameter erwartet. Der Einsatz ermöglichte zudem mehr Flexibilität. Bei Verwendung des Konstruktors hätten entweder separate Konstruktoren für verschiedene Kombinationen von Parametern bereitstellen oder Parameter auf einen Standardwert gesetzt werden müssen, wenn sie nicht angegeben worden wären. Zuletzt ermöglichte der Einsatz des Erbauers eine Trennung von Erstellungslogik und Geschäftslogik. Das Builder-Muster ermöglicht es, die Erstellungslogik des eher komplexen Match-Objekts von seiner Geschäftslogik zu trennen. Das macht den Code übersichtlicher und einfacher zu warten und zu erweitern.

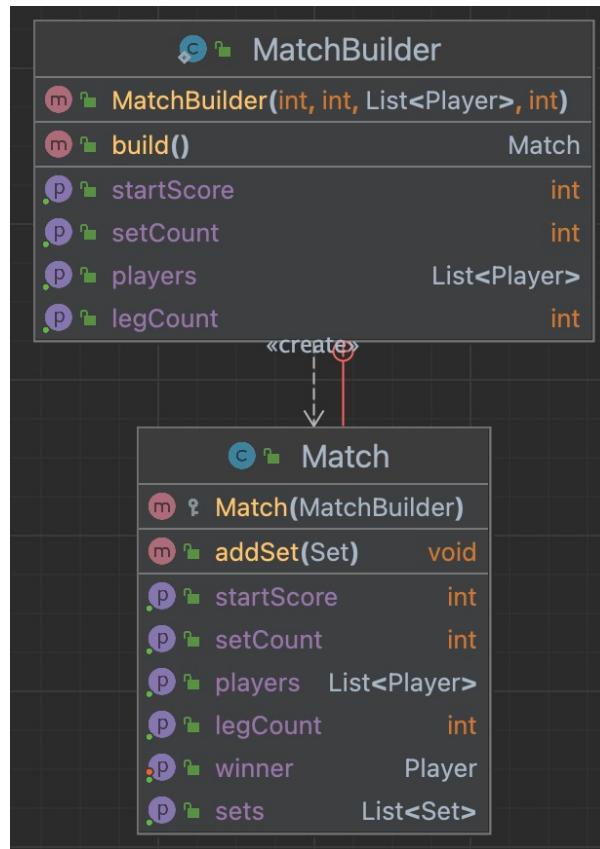


Abbildung 8.1: MatchBuilder

```
1 package de.p3lina.domain;
2
3
4 import java.util.ArrayList;
5 import java.util.List;
6 public class Match {
7
8     protected Match(MatchBuilder matchBuilder) {
9         this.players = matchBuilder.getPlayers();
10        this.startScore = matchBuilder.getStartScore();
11        this.setCount = matchBuilder.getSetCount();
12        this.legCount = matchBuilder.getLegCount();
13        this.sets = new ArrayList<>();
14    }
15}
```

```
16     private List<Set> sets;
17     private int legCount;
18     private int setCount;
19     private List<Player> players;
20     private int startScore;
21     private Player winner;
22
23     public void addSet(Set set) {
24         this.sets.add(set);
25     }
26
27     public List<Set> getSets() {
28         return sets;
29     }
30
31     public int getLegCount() {
32         return legCount;
33     }
34
35     public int getSetCount() {
36         return setCount;
37     }
38
39     public List<Player> getPlayers() {
40         return players;
41     }
42
43     public int getStartScore() {
44         return startScore;
45     }
46
47     public Player getWinner() {
48         return winner;
49     }
50
51     public void setWinner(Player winner) {
```

```
52         this.winner = winner;
53     }
54
55     public static class MatchBuilder {
56         private int legCount;
57         private int setCount;
58         private List<Player> players;
59         private int startScore;
60         public MatchBuilder(int legCount, int setCount, ↴
61             ↪ List<Player> players, int startScore) {
62             this.legCount = legCount;
63             this.setCount = setCount;
64             this.players = players;
65             this.startScore = startScore;
66         }
67         public Match build() {
68             return new Match(this);
69         }
70         public List<Player> getPlayers() {
71             return players;
72         }
73         public int getStartScore() {
74             return startScore;
75         }
76         public int getSetCount() {
77             return setCount;
78         }
79         public int getLegCount() {
80             return legCount;
81         }
82     }
```

Listing 8.1: Match-Klasse

8.2 Entwurfsmuster: Strategie

Das Strategie-Entwurfsmuster bietet erhebliche Flexibilität, da es ermöglicht, das Verhalten von Objekten zur Laufzeit dynamisch zu ändern. Es fördert die Wiederverwendbarkeit und Austauschbarkeit von Algorithmen und erleichtert die Entkopplung von spezifischen Verhaltensweisen von den Klassen, die verwendet werden. Dadurch wird der Code sauberer und einfacher zu warten.

```

1 public interface Handle<ReturnType, Parameter, ↴
   ↴ AnotherReturnType, AnotherParameter> {
2
3
4     ReturnType process(Parameter something);
5     AnotherReturnType isMatchSetWon(AnotherParameter something);
6 }
```

Listing 8.2: Handle-Klasse

Umgesetzt wurde das Entwurfsmuster mit Hilfe der Handle-Klasse, die den Handle-Klassen eine Methode *process* bereitstellt, die den jeweiligen Teil des Dartspiels durchlaufen lassen. Dieses Interface wird dann von allen Handle-Klassen implementiert.

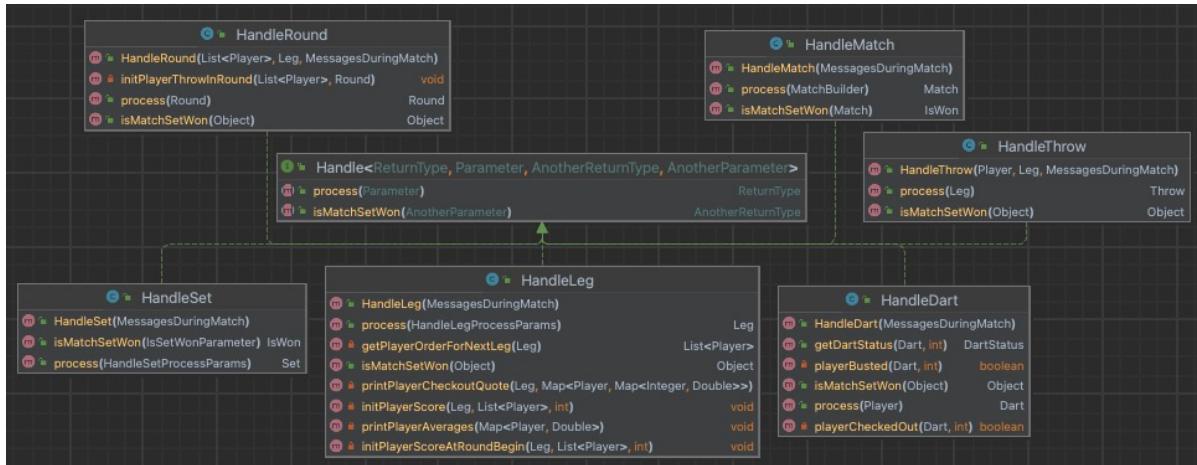


Abbildung 8.2: Implementierung der Handle Klasse im UML

Literaturverzeichnis

- [1] Briem, L. *Clean Architecture Folien*. https://moodle.dhbw.de/pluginfile.php/242767/mod_resource/content/1/clean-architecture.pdf. o.O. (Einsichtnahme: 28.05.2023).