Tournament Generator

Advanced Software Engineering Projekt

Für die Prüfung zum

Bachelor of Science

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Vincenzo Ciullo, Dennis Wiebe

Abgabedatum	31.05.2021
Matrikelnummer	8683078 / 6735868
Kurs	TINF18B5
Dozent	Maurice Müller

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Aufgabenstellung

Bei vielen Spielen mit mehreren Spielern bietet es sich an ein Turnier zu erstellen anstatt ohne ein System gegeneinander zu spielen. Deswegen haben wir uns die Aufgabe gestellt einen Turniergenerator in der Konsole zu programmieren. Hierfür haben wir uns für die Programmiersprache C# entschieden.

Dazu muss geklärt werden welche Arten von Turnieren es gibt und wie ein solches Turnier abläuft. In unserem Fall gibt es die Möglichkeit direkt einen Turnierbaum mit der Anzahl der Spieler zu erzeugen (sofern die Spieleranzahl eine Potenz von 2 ist). Dabei kann auch frei entschieden werden, ob das Turnier in einem sogenannten Doppel-KO-System gespielt werden soll. Die zweite Möglichkeit besteht darin zunächst eine Gruppenphase zu ermöglichen und anschließend aus den besten der Gruppen einen Turnierbaum zu erstellen.

Das Ziel ist es, dass die geschriebene Software langlebig ist und dass von guter Softwarequalität die Rede ist.

In den nachfolgenden Kapiteln werden die einzelnen Punkte, um das Ziel zu erreichen nähergebracht.

2 Aufbau Projekt (Klassen)

Damit das Projekt übersichtlicher gestaltet ist, wurde es in mehrere Ordner aufgeteilt. Im Nachfolgenden werden diese Ordner mit den dazugehörigen Klassen aufgezeigt.

Dabei handelt es sich um den finalen Stand der Anwendung. Dieser wurde über die Zeit öfter angepasst. Die einzelnen Ordner die als nächstes beschrieben werden sind:

- MainEngine
- GroupEngine
- TreeEngine
- Entities
- Core
- Adapters
- DataholderClasses
- Log
- Excel

Unit Test Klassen werden ebenfalls separat in einem Ordner gehalten und im Kapitel *Unit Tests* beschrieben.

MainEngine

Klasse	Program
Beschreibung	Einstiegspunkt des Programms
Verwendung	Main-Methode zum Starten der Anwendung

Klasse	ChooseHelper
Beschreibung	Helferklasse beim Benutzen der Software
Verwendung	1. Gibt Auskunft über Anzahl bisher eingegebener Spieler sowie ob Gruppe oder Turnierbaum möglich sind 2. Gibt zu Spielen ein Vorschlag ob mit oder ohne Gruppenphase empfohlen wird.
Attribute/ Properties	int AmountOfPlayers

Klasse	TournamentGenerator
Beschreibung	Hauptklasse, welche die Generation startet
Verwendung	1. Erstellt Spieler anhand der Benutzereingaben und prüft diese 2. Übergibt Spieler an einen Turnierbaum oder einer Gruppenphase
Attribute/ Properties	int AmountOfPlayers List<Player> AllPlayers bool WithGroupPhase bool WithWildCards const int MINIMUM_PLAYERS_COUNT const int MINIMUM_GROUP_PLAYERS_COUNT

GroupEngine-Ordner

Klasse	GroupPhase
Beschreibung	Ist für die Gruppenphase des Turniers zuständig
Verwendung	1. Nimmt Spieler-Liste entgegen und erstellt Gruppen anhand der Anzahl der Spieler 2. Spielt die einzelnen Spiele (Matches) in der Gruppe 3. Wertet die Gruppe aus damit die besten ausgewählt werden können
Attribute/ Properties	IList<Group> Groups IList<Players> Players List<Players> RemainingPlayers IList<Match> AllMatches GroupFactory _groupFactory

TreeEngine-Ordner

Klasse	TournamentTree
Beschreibung	Basisklasse für einen Turnierbaum
Verwendung	1. Startet Turnierbaum-Generator Je nach Eliminationsverfahren
Attribute/ Properties	IElimination _elimination

Interface	IElimination
Beschreibung	Stellt eine Schnittstelle für Eliminationsverfahren dar
Verwendung	1. Startet Elimination 2. Eliminiert Spieler, die verloren haben 3. Zeigt den Turnierbaum an

Klasse	SingleElimination
Beschreibung	Standard Turnierbaum im KO-Verfahren
Verwendung	1. Implementiert IElimination 2. Startet Elimination anhand von einer Liste von Spielern im Single Elimination-Verfahren
Attribute/ Properties	IList<Player> Players bool FirstTree ILog log

Klasse	DoubleElimination
Beschreibung	Turnierbaum im Doppel-KO-System
Verwendung	1. Implementiert IElimination 2. Startet Elimination anhand von einer Liste von Spielern im Double Elimination-Verfahren
Attribute/ Properties	IList<Player> Winners IList<Player> Losers bool FirstTree bool FirstLosers ILog log

Entities-Ordner

Klasse	Match
Beschreibung	Stellt ein Spiel innerhalb eines Turniers dar
Verwendung	1. Repräsentiert ein Match zwischen 2 Spielern
Attribute/ Properties	Player PlayerOne Player PlayerTwo

Klasse	Group
Beschreibung	Stellt eine Gruppe von Spieler eines Turniers dar
Verwendung	1. Repräsentiert eine Gruppe von mehreren Spielern
Attribute/ Properties	int GroupId IList<Player> Players

Core-Ordner

Klasse	MatchFactory
Beschreibung	Stellt den Application Code von einem Match dar
Verwendung	1. Fügt Spieler zu einer Gruppe hinzu 2. Sortiert Spieler je nach Punkten

Klasse	GroupFactory
Beschreibung	Stellt den Application Code von einer Gruppe dar
Verwendung	1. Spielt ein Match 2. Validiert ein Match 3. Vertauscht Heim und Gast Spiel 4. Überprüft Eingabe von Ergebnissen

Adapters-Ordner

Klasse	MatchAdapter
Beschreibung	Adapter für Spiele
Verwendung	1. Konvertiert Punkte von Spielern zur Weiterverwendung von int zu string

Klasse	GroupAdapter
Beschreibung	Adapter für Gruppen
Verwendung	1. Wird noch nicht verwendet, kann aber für zukünftige Konvertierungen benötigt werden

DataHolderClasses-Ordner

Klasse	Player
Beschreibung	Stellt einen Spieler eines Turniers dar
Verwendung	1. Repräsentiert einen Spieler mit Name und ID
Attribute/ Properties	Name PlayerName Identification PlayerID int Wins int Ties int Points int GoalDifference bool isWildCard

Klasse	Name
Beschreibung	Stellt einen Namen eines Spielers des Turniers dar
Verwendung	1. Repräsentiert den Namen eines Spielers
Attribute/ Properties	string Title

Klasse	Identification
Beschreibung	Stellt die ID eines Spielers des Turniers dar
Verwendung	1. Repräsentiert die ID eines Spielers
Attribute/ Properties	int Id

Klasse	Component
Beschreibung	Stellt eine Klasse dar, die nützliche Methoden bereitstellt
Verwendung	1. Mischt Spieler (Player) 2. Mischt Spiele (Match) 3. Prüft dass keine Frei-Lose gegeneinander spielen 4. Erstellt ein Log eines Turniers

Log-Ordner

Interface	ILog
Beschreibung	Stellt eine Schnittstelle für Logs dar
Verwendung	1. Fügt einen Eintrag zu einem Log hinzu 2. Erstellt ein Log

Klasse	TournamentLog
Beschreibung	Erstellt ein Log eines Turnier als Textdatei
Verwendung	1. Implementiert ILog
Attribute/ Properties	List<String> LogEntries

Excel-Ordner

Klasse	CellFinder
Beschreibung	Klasse, um für den Export Zellen zu finden
Verwendung	1. Liefert eine Zelle aus der Excel-Datei, die dann beschriftet werden kann.

Klasse	CellWriter
Beschreibung	Klasse, um Inhalte in eine Zelle zu schreiben
Verwendung	1. Schreibt den gewünschten Inhalt in eine Zelle der Excel-Datei

Klasse	ExcelExporter
Beschreibung	Führt den Export nach Excel durch
Verwendung	1. Wird für den Export nach Excel verwendet

Klasse	ExcelExportFactory
Beschreibung	Klasse, die für das Erstellen der Excel-Export-Klassen verwendet wird
Verwendung	1. Erzeugt die Klassen, die den Excel-Export durchführen (Gruppe, Single-, DoubleElimination)

Interface	IExcelExport
Beschreibung	Schnittstelle für den ExcelExport
Verwendung	1. Wird implementiert, wenn die Klasse einen ExcelExport bereitstellen muss.

Klasse	RowsCounter
Beschreibung	Klasse für den ExcelExport
Verwendung	1. Zählt die Reihen, die für den Export angelegt werden müssen.

Klasse	RowsCreator
Beschreibung	Klasse für den ExcelExport
Verwendung	1. Erzeugt alle Reihen, die in der Excel-Datei benötigt werden.

Klasse	TournamentBracketLog
Beschreibung	ExcelExport für SingleElimination
Verwendung	1. Implementiert IExcelExport 2. Exportiert einen SingleElimination Turnierbaum
Attribute/ Properties	int startPosition int? nextStartPosition List<TournamentBracketLogRound> Rounds

Klasse	TournamentBracketLogRound
Beschreibung	Stellt eine Runde für den ExcelExport dar
Verwendung	1. Wird für den ExcelExport verwendet
Attribute/ Properties	int Round List<TournamentBracketLogRoundMatch> Matches

Klasse	TournamentBracketLogRoundMatch
Beschreibung	Stellt ein Spiel für den ExcelExport dar
Verwendung	1. Wird für den ExcelExport verwendet
Attribute/ Properties	string PlayerOne string PlayerTwo bool FirstPlayerWin

Klasse	TournamentDoubleKoLog
Beschreibung	ExcelExport für DoubleElimination
Verwendung	1. Implementiert IExcelExport 2. Exportiert einen DoubleElimination Turnierbaum
Attribute/ Properties	List<TournamentBracketLogRound> WinnerRounds List<TournamentBracketLogRound> LoserRounds List<TournamentBracketLogRoundMatch> FinalMatches

Klasse	TournamentGroupLog
Beschreibung	ExcelExport für Gruppenphase
Verwendung	1. Implementiert IExcelExport 2. Exportiert die Gruppenphase
Attribute/ Properties	List<TournamentGroupLogMatch> AllMatches Dictionary<int, List<Player>> Groups

Klasse	TournamentGroupLogMatch
Beschreibung	Stellt ein Gruppenspiel für den ExcelExport dar
Verwendung	1. Wird für den ExcelExport verwendet
Attribute/ Properties	string PlayerOne string PlayerTwo int PlayerOnePoints int PlayerTwoPoints

3 Unit Tests

In diesem Kapitel werden Unit Tests für das Softwareprojekt erstellt. Die Anzahl der Testklassen ist dabei 7. Diese Klassen haben aber meist mehrere Tests innerhalb der Klassen, sodass insgesamt **13** Tests vorhanden sind.

3.1 Tests für CheckIfAmountOfPlayersIsPowerOfTwo

Für die oben genannte Methode wurden zwei Tests geschrieben. Die erste Methode überprüft, ob korrekt ermittelt werden kann, dass die Anzahl der Spieler eine Potenz von 2 ist. Die zweite Methode ermittelt den anderen Fall. Es wird also eine Spieleranzahl angegeben, bei der es sich um keine Potenz handelt.

Hierfür wird für beide Methoden ein vereinfachtes Objekt des Turniergenerators erzeugt. Dieser wird mit vier und mit drei Spielern befüllt. In der ersten Methode wird TRUE als Ergebnis erwartet, da hier die vier Spieler im Turniergenerator enthalten sind. In der zweiten Methode, die mit drei Spielern im Generator, ist das erwartete Ergebnis FALSE.

Diese Tests werden durchgeführt, da es sich bei der getesteten Methode um eine sehr wichtige Methode für den korrekten Ablauf eines Turniers handelt. Bei einem Turnier kann ein Turnierbaum, also eine KO-Phase, nur richtig erzeugt werden, wenn es sich bei der Anzahl der Spieler um eine Potenz von 2 handelt, da in der KO-Phase immer die Verlierer aus den Spielen ausscheiden und die Gewinner sich für die nächste Runde qualifizieren. Ist die Anzahl der Spieler keine Potenz von 2, kann niemals ein Gewinner ermittelt werden, da nie ein Finale mit 2 Spielern erreicht werden kann. Deswegen muss diese Methode immer funktionieren, was durch die

Tests sichergestellt ist, da diese auf einen Fehler laufen, wenn die Methode irgendwann mal fälschlicherweise geändert werden sollte.

Die zwei Tests sind im Commit 6071bfa.

Des Weiteren wurde der Test auf ein Fake-Objekt geändert. Diese Änderung ist unter Commit 177149c zu finden.

3.2 Tests für den Excel-Export

Eine große Funktionalität des Turniergenerators ist die Möglichkeit, den Turnierverlauf nach Excel zu exportieren. Dabei wird jeweils für die Gruppen und für die KO-Phase des Turniers ein Sheet für eine Excel-Datei erzeugt und mit den nötigen Informationen befüllt. Damit die Exports immer funktionieren und nicht durch mögliche zukünftige Commits kaputt gemacht werden, wurden für die Exports ebenfalls Tests geschrieben.

Dabei wurde für den Export von Gruppen und den Export der KO-Phase eine eigene Methode geschrieben, da sich die Exports voneinander unterscheiden. Damit die Funktionalität der einzelnen Export-Methoden getestet werden können, wurden vor dem eigentlichen Test einige Objekte erzeugt, die einen Turnierverlauf simulieren. Für die Gruppen ist es dabei nicht wichtig, wie das Ergebnis der einzelnen Spiele lautet, weswegen hier immer ein 1:1 ausgewählt worden ist. Der Export findet unabhängig vom Ergebnis statt. Nachdem die benötigten Informationen in die Export-Objekte befüllt worden sind, wird noch ein vereinfachtes Excel-Dokument erzeugt, welches beim Export benötigt wird. Erst dann kann der eigentliche Export der Gruppen stattfinden. Nachdem dieser durchgelaufen ist, finden die eigentlichen Tests

statt. Hierfür werden nun einige Zellen des Dokuments ermittelt und mit den erwarteten Ergebnissen verglichen.

Ähnlich wurde auch für den Export der KO-Phase vorgegangen, jedoch sind für diesen Export andere Daten notwendig, die vor dem Test erstellt werden müssen. Die Tests dieser Methode befassen sich mit der korrekten Positionierung der ersten Elemente der einzelnen Runden. Diese muss immer gleich sein, damit auch ein richtiger Baum erzeugt werden kann.

Die Tests befinden sich im Commit dd04538.

3.3 Test für GetLetterByNumber-Methode

Für die GetLetterByNumber-Methode wurde ebenfalls ein Test geschrieben. Diese Methode ist sehr wichtig für den Excel-Export, da durch diese Methode die Spalten ermittelt werden, die für den Export wichtig sind. Anhand der Rückgabe dieser Methode werden einzelne Elemente im Export in die Excel-Datei geschrieben. Da die Methode nach dem Refactoring (siehe *Refactoring*) eine Exception werfen kann, ist es für diesen Test notwendig, dass ein weiteres Attribut über der Testmethode verwendet wird: „ExpectedException“. Hier wird der Typ der erwarteten Exception angegeben, sodass der Test nicht fehlschlägt, wenn diese Exception geworfen wird. Der Test erfolgt dann anhand einer Zahl, bei der der erwartete Buchstabe angegeben wird und einer zu hohen Zahl, bei der die Exception geworfen wird.

Der Test befindet sich im Commit c257e19.

3.4 Tests für Klasse Component

In dieser Klasse befinden sich Methoden, die in anderen Klassen häufig verwendet werden. Dazu gehören Spiele mischen (ShuffleMatches), Spieler mischen (ShufflePlayers) und überprüfen, dass keine Frei-Lose gegeneinander spielen (NoFreeWinsAgainstEachOther). Für diese 3 Methoden werden Tests in der ComponentTest Klasse geschrieben.

Der Commit, der für die nachfolgenden Test ausgeführt wurde, ist: a542213.

Aufgrund der Länge der Tests und der daraus folgenden Unübersichtlichkeit, wurden diese Tests erneut angepasst. Die Anpassung befindet sich im Commit 683cf21. In der Anpassung wurden FakeObjekte erstellt, welche die Tests deutlich übersichtlicher gestalten, ohne den Test an sich zu verändern.

3.4.1 ShuffleMatches

Dieser Test dient dazu zu überprüfen ob die Methode die Reihenfolge von Matches in einer Liste verändert, also vermischt. Dazu werden zunächst 16 Spieler angelegt und mit den 16 Spielern werden 8 Matches einer Gruppenphase erstellt. Dann wird die Reihenfolge der Matches mit der ursprünglichen Reihenfolge verglichen.

Diese spezielle Anpassung ist im Commit 9f4cf3f.

Vorher waren es nur 4 Matches, die überprüft wurden, das ist aber keine gute Anzahl, da die Ursprüngliche Kombination der Matches eintreten kann und somit der Test fehlschlägt, weil keine Aussage über das Mischen getroffen werden kann. Zwar ist es mit 8 Matches immer noch möglich, aber die Wahrscheinlichkeit, dass es auftritt ist viel geringer. Zum Überprüfen des Mischverfahrens sollten definitiv nicht

weniger als 4 Matches verwendet werden, da sonst sehr schnell die Ursprüngliche Kombination wahrscheinlich wird.

Falls die Reihenfolge der Matches eine andere, als die ursprüngliche Reihenfolge ausweist gilt der Test als erfolgreich.

3.4.2 ShufflePlayers

So wie Shuffle Matches muss auch die Möglichkeit gegeben sein die Spieler beispielsweise innerhalb eines Turnierbaums zu Beginn zu mischen. Analog zu ShuffleMatches wird hier überprüft, ob das Mischen eine veränderte Reihenfolge der Spieler aufweist. Dazu werden 8 Spieler erstellt und die Spieler in ein Single Elimination Turnierbaum angelegt. In diesem Turnierbaum werden diese Spieler dann gemischt.

Der Test gilt als erfolgreich, wenn die 8 Spieler in einer anderen Reihenfolge vorhanden sind, als sie erstellt wurden. Ähnlich zu den Matches im vorherigen Kapitel sollten nicht weniger Spieler für den Test verwendet werden, da beispielsweise bei 2 Spielern das Mischen zu 50 % die ursprüngliche Reihenfolge aufweist.

3.4.3 NoFreeWinsAgainstEachOther

Diese Methode überprüft, ob in einem Turnierbaum Frei-Lose gegen Frei-Lose spielen. Dazu erhält er eine Liste und gibt einen bool Wert zurück. Der Test soll dazu dienen ob die Methode bei Beispielhaften Eingaben den erwarteten Wert True oder False ausgibt. Dazu werden 2 Spieler angelegt und 2 Frei-Lose. Dann folgen 2 Listen, bei der einmal Frei-Lose miteinander spielen und einmal nicht. Bei Ersterem soll False die Ausgabe sein und bei der zweiten Liste True.

Der Test ist erfolgreich, wenn diese beiden Werte angenommen werden und somit den erwarteten entsprechen.

3.5 Tests für die Klasse Match

Ein Match hat stets einen PlayerOne und einen PlayerTwo. Dabei sind der erste Spieler der Heim-Spieler und der zweite Spieler der Gast-Spieler. Bei Hin- und Rückrunde in einer Gruppenphase müssen die Rollen getauscht werden. Dazu werden 2 Spieler angelegt und aus den beiden Spielern ein Match gebildet. Danach wird die Methode ChangeHomeAndAway aufgerufen. Der Test ist dann erfolgreich, wenn der PlayerOne des Match Objektes der erstellte PlayerTwo entspricht und umgekehrt.

Der Umfang dieses Tests muss nicht auf mehrere Matches erweitert werden, da dieser Test Repräsentant für jeden weiteren Match ist.

Eine weitere Funktionalität eines Matches ist es, die Eingabe eines Ergebnisses eines Matches zu überprüfen. Die Eingabe kann True oder False entsprechen, je nachdem ob die Eingabe der korrekten Syntax entspricht oder nicht. Dazu wird erneut ein Match mit 2 Spielern erzeugt. Anschließend wird dreimal ein falscher Input für die Methode CheckInputOfMatch verwendet und einmal ein korrekter Input. Bei den falschen Inputs wird jeweils ein anderer Fehler überprüft. Dazu gehören es werden genau zwei Werte benötigt, der erste Wert entspricht keiner Zahl und der zweite Wert entspricht keiner Zahl. Damit sollten alle möglichen Fehler abgedeckt sein. Der Test gilt als Erfolg, wenn bei den 3 falschen Eingaben ein False als Ergebnis ausgegeben wird und bei der richtigen Eingabe ein True.

Die beiden Tests sind im Commit 66a11aa.

3.6 Tests für Single Elimination

Der erste Test prüft eine essenzielle Methode, die EliminateLosingPlayer Methode. Diese wirft ausgeschiedene Spieler aus dem Turnier und somit aus der Liste der Spieler des Turniers. Für den Test werden 4 Spieler erstellt und in ein SingleElimination Turnier hinzugefügt. In dem Turnier werden dann 2 Spieler als Verlierer definiert und mithilfe dieser die Methode EliminateLosingPlayer aufgerufen. Anschließend wird überprüft, ob die Verlierer nicht mehr in der Liste der Spieler des Turniers vorhanden sind. Trifft dies zu, ist der Test erfolgreich. Weitere Spieler würden den Test nur unübersichtlicher gestalten.

Eine weitere Methode, die getestet werden sollte, ist die CreateTree Methode. Diese baut den Turnierbaum in der Konsole auf und dient dazu dem Benutzer zu zeigen, wie der Baum aussieht. Exemplarisch werden hier ebenfalls 4 Spieler verwendet und ein SingleElimination Turnier erstellt. Zum Testen wird der gewünschte Baum mit einem String aufgebaut. Anschließend wird die Methode CreateTree aufgerufen und geschaut ob es dem gewünschten String entspricht. Falls dies der Fall ist, ist der Test erfolgreich. Weitere Spieler würden erneut nur die Komplexität des Tests erhöhen und nicht weitere Funktionalitäten testen.

Die beiden Tests sind im Commit 5fc1462.

Da in beiden Tests jeweils Spieler erzeugt werden, wurde im Commit d81f718 eine Anpassung vorgenommen. Diese beschreibt eine Anpassung des FakeObjekts FakeSingleElimination. Die beiden Tests sind nun viel Übersichtlicher und verwenden die FakeSingleElimination anstatt eigene Spieler zu erzeugen und ein SingleElimination Objekt zu erstellen.

3.7 Test für Double Elimination

Einer der wichtigsten Methoden, die in der DoubleElimation Klasse vorhanden ist, im Vergleich zur Single Elimination, ist die MoveLosingPlayer Methode. Dabei werden ausgeschiedene Spieler vom Winner bracket in das Loser Bracket verschoben.

Dazu werden 4 Spieler erstellt und anschließend ein DoubleElimination Objekt mit einer Liste der eben erstellten Spieler. Dann wird die Methode MoveLosingPlayer aufgerufen. Dabei wird eine List von Spielern benötigt, sodass 2 von den 4 erstellten Spielern gewählt werden. Bei mehr Spielern würde keine weitere Funktionalität überprüft werden, da nur die Liste der beispielhaften Spieler länger und somit der Test unübersichtlicher wird. Anschließend folgen die Asserts.

Der Test ist erfolgreich, wenn die Property Winner aus double Elimination genau zwei Spieler beinhaltet sowie die Property Loser genau zwei Spieler enthält. Des Weiteren wird mit einem Asser überprüft ob die richtigen Spieler in den jeweiligen Properties vorhanden sind.

Der Commit der diesen Test beinhaltet ist 668af6a.

Auch hier wurde eine weitere Anpassung vorgenommen, welche ein FakeObjekt beinhaltet. Es wurde die Klasse FakeDoubleElimination erstellt, sodass nicht mehr im Test Spieler erzeugt werden müssen. Bei weiteren Tests von Double Elimination kann das FakeObjekt weiter angepasst werden, sodass doppelter Code in Zukunft verhindert werden kann. Der Commit ist e51b10d.

4 Programming Pinciples

Hier geht es darum zu analysieren, ob sich der Code an Programmier-Prinzipien hält. Darunter wird exemplarisch anhand SOLID, GRASP und DRY der Code betrachtet.

4.1 SOLID

In SOLID steht jeder Buchstabe für ein Prinzip. Im Nachfolgenden wird also auf SRP, OCP, LSP, ISP und DIP eingegangen.

4.1.1 Single-Responsibility Principle

SRP steht für Single Responsibility Principle und bezieht sich auf Module, Klassen, Methoden und Variablen. Die eben genannten sollen nach diesem Prinzip jeweils nur für eine Aufgabe zuständig sein.

Ein Teil der Umsetzung des Single-Responsibility Principle ist unter Commit cbf9eca zu finden. Der Excel-Export wurde so aufgeschlüsselt, dass für jede kleinere Funktionalität eine eigene Klasse erstellt worden ist. Diese Klassen sind klar benannt und dadurch weiß der Programmierer später auch, was die einzelnen Klassen können.

In der Tournament Generator Klasse wird bei der Methode CreateAllPlayers viel mehr getan als nur Spieler erzeugt. Das bedeutet, dass hier nicht das Prinzip des SRP benutzt wird. Im Commit 4e4d97a wurde das verbessert. Es wurden zwei weitere Methoden angefertigt, welche die Funktionalität der Klasse CreateAllPlayers verlagert. Es wird in der Methode nur noch ein Name entgegengenommen und Spieler erstellt. Falls falsche Eingaben getätigt werden, übernimmt dies die neue

Methode ValidateInputOfPlayerName und falls Frei-Lose erstellt werden sollen, ist die neu hinzugefügte Methode CreateWildCards zuständig.

Trotz der beiden kleinen Refactorings ist nicht im gesamten Projekt das Prinzip eingehalten. Beispielsweise wird beim Erstellen eines Baums die Methode StartTreeGenerator in der Klasse SingleElimination verwendet. Diese umfasst knapp 80 Zeilen Code und könnte etwas ausgelagert werden.

4.1.2 Open-Closed Principle

OCP steht für Open-Closed Principle und bezieht sich auf Klassen / Module. Diese sollen nämlich sowohl offen für Erweiterung sein als auch verschlossen für Veränderungen.

TournamentGroupLog

Eine Möglichkeit für das OCP ist unter Commit 1b3ce09 zu finden. Die Klasse TournamentGroupLog erwartete bisher die einzelnen Variablen, die für das Erzeugen eines TournamentGroupLogMatch-Objektes nötig sind. Sollte es irgendwann also möglich sein, dass ein Match mit drei Spielern gespielt werden kann, müsste diese Klasse angepasst werden. Dadurch, dass nun gleich ein TournamentGroupLogMatch erwartet wird, ist dies nicht mehr so.

ITournamentTree

In der Domäne gibt es die Möglichkeit einen Turnierbaum zu erstellen. Dies kann zu einem Standard Baum führen, in dem man direkt ausscheidet, sobald man verliert oder ein Doppel Ko System, in dem man zwei Niederlagen benötigt. In beiden Fällen handelt es sich um einen Turnierbaum. In der Vorherigen Namensgebung wurde durch den Namen Tournament Tree (für den Standard Baum) und den Namen

DoubleKO keine Verknüpfung dieser beiden Komponenten ersichtlich. Es wurde nun ein Interface erstellt (ITournamentTree -> später IElimination siehe Entwurfsmuster). Der vorherige Tournament Tree wurde in SingleElimination und die Klasse DoubleKO wurde in DoubleElimination umbenannt. Falls Erweiterungen geplant werden, können diese über das Interface an beiden Klassen ausgeführt werden. Diese Änderung ist am Commit 6387fd8 zu sehen.

4.1.3 Liskov Substitution Principle

LSP steht für Liskov Substitution Principle und hat den Vorteil einer besseren Abstraktion. Es geht darum, dass eine abgeleitete Klasse an jeder beliebigen Stelle ihre Basisklasse ersetzen können soll, ohne, dass es zu unerwünschten Nebeneffekten kommt. In unserem Fall wird es am Beispiel SingleElimination und DoubleElimination deutlich. Double Elimination könnte von Single Elimination erben oder umgekehrt, wie im Beispiel Rechteck und Quadrat. Allerdings ist dies keine gute Lösung, da je nachdem immer daran gedacht werden müsste, weitere Methoden zu überschreiben. Aus diesem Grund wurde ein Interface ITournamentTree erstellt, von dem beide Klassen erben, um eine bessere Abstraktion darzustellen.

4.1.4 Interface Segregation Principle

ISP steht für Interface Segregation Principle und bezieht sich wie der Name schon auf Interfaces. Es geht darum, dass nicht ein großes Interface für mehrere Klassen verwendet wird, sondern dieses große Interface auf kleinere aufgeteilt wird. In unserer Anwendung gibt es nicht viele Interfaces, jedoch ist das Prinzip anhand ITournamntTree(IElimination siehe Entwurfsmuster) verständlich nahezubringen. Das Interface hat zwei Klassen die, dieses Interface verwenden nämlich SingleElimination für einen ganz normal Turnierbaum und DoubleElimination für einen Turnierbaum mit

DoppelKO-System. In Beiden werden alle Methoden, die im Interface vorhanden sind, benutzt, sodass eine Aufteilung zunächst kein Sinn ergibt. Sollten weitere Klassen folgen, die ebenfalls dieses Interface benutzen, sollte das In Erwägung gezogen werden. Beispielsweise wenn in der Zukunft ein sehr selten genutztes Format TrippleElimination genutzt wird, kann man evtl. das Interface aufteilen in mehrere Interfaces.

4.1.5 Dependency Inversion Principle

DIP steht für „Dependency Inversion Principle" und beschäftigt sich mit der Abhängigkeit von Modulen in der Software. Dies kann in unserem Fall an folgendem Beispiel verwendet werden. In den Klassen SingleElimination und DoubleElimination gibt es eine Property vom Typ TournamentLog. Diese Klasse wird verwendet, um nach dem Durchlauf des Turniers ein Log zu erstellen und in einer Textdatei zu speichern. Jedoch wird der Log bisher direkt in der Klasse instanziiert, was eine Verletzung des DIP bedeutet. Um diese Verletzung zu verhindern, muss eine erzeugte Instanz des TournamentLog-Objektes in den Konstruktor der Single- und DoubleElimination-Klassen übergeben werden. Dies kann durch Dependency Injection erreicht werden. Des Weiteren wurde noch folgende Änderung vorgenommen: Damit es später möglich ist, auch eine andere Log-Klasse zu verwenden, erwarten die Konstruktoren jetzt nicht mehr ein Objekt der Klasse TournamentLog, sondern ein Objekt, welches von der Schnittstelle ILog erbt.

Die Änderung hierfür ist unter Commit ad28743 zu finden.

4.2 GRASP

GRASP steht für „General Responsibility Assignment Software Patterns” und beschreibt Prinzipien oder Muster für Zuständigkeiten. So wird die Fragenstellung, welche Klasse für was zuständig sein sollte, durch dieses Softwaredesign-Pattern beantwortet.

4.2.1 Information Expert

Ein Beispiel für Information Expert aus unserem Code ist die Klasse Group, zu finden im Ordner GroupEngine. Information Expert beschreibt, dass für eine Aufgabe der zuständig ist, der bereits das meiste Wissen für diese Aufgabe hat. In diesem Fall ginge es um das Sortieren der Spieler innerhalb einer Gruppe. Ein Objekt der Klasse Group kennt die Punkte und Tordifferenzen der Spieler der Gruppe und somit kann sich die Gruppe selbst um das Sortieren der Spieler als Rangliste kümmern. So findet sich in der Klasse Group die Methode, SortPlayers(), welches alle Spieler in einer Gruppentabelle sortiert.

4.2.2 Creator

Ein Beispiel für das Creator-Prinzip ist ebenfalls bereits in unserem Code zu finden. Objekte, welche vom Interface IExcelExport erben, sollten auch dann nur erzeugt werden, wenn tatsächlich ein Export benötigt wird. So wird in der Klasse ExcelExporter nur dann ein Objekt für die benötigte Exportart erzeugt, wenn auch tatsächlich ein Export aufgerufen wird (da es sein kann, dass ein Turnier ohne Gruppen gespielt wird, ist somit nicht immer ein Export für Gruppen notwendig).

4.2.3 Controller

Ein Controller ist die erste Schnittstelle nach der GUI – in unserem Fall der Code, welche Konsoleneingaben auswertet. Ein Beispiel für einen Controller, in Bezug auf das Delegieren von Aufgaben, in unserem Code sind die Klassen „SingleElimination“ und „DoubleElimination“. Hierbei handelt es sich um Klassen, welche sich um das Durchführen des Turnierbaums kümmern. Am Ende eines Turniers, also sobald ein Gewinner feststeht, hat der Anwender die Möglichkeit, das Turnier abzuspeichern. Hierfür hat er zwei Möglichkeiten: Das Abspeichern in einer Excel-Datei und das Abspeichern als Log in einer Textdatei. Für beide Fälle gibt es eigene Klassen und die Single- bzw. DoubleElimination-Klasse delegiert das Abspeichern nach dem Turnier an die jeweiligen Klassen.

4.2.4 Low Coupling

Ein Beispiel für Low Coupling in unserem Code ist in den Klassen „Player“, „Match“ und „Group“ zu finden. Dadurch, dass wir die Klassen so gewählt haben, ist es für uns einfach, diese auch mehrfach und in einem anderen Kontext zu verwenden. So besteht eine Gruppe (Group) zum Beispiel aus mehreren Spielern (Player) und zwei Spieler sind Teil eines Spiels (Match). Die Klasse Match wird jedoch nur in der Gruppenphase verwendet, da es hier eine Angabe von Punkten benötigt, um ein Spiel erfolgreich zu beenden. Da bei uns Low Coupling herrscht, können wir so auch die Player-Klasse verwenden, wenn das Turnier in die Single- oder DoubleElimination geht. Hier werden zwei Spieler (Player) pro Spiel angegeben und der Anwender muss hier keine Werte mehr angeben, sondern nur, welcher Spieler gewonnen hat. So wird die Klasse Player in einem Match in der Gruppenphase verwendet, aber auch in der KO-Phase, wenn sie gegen ein anderes Objekt der

Klasse Player spielt. Diese erhöhte Verwendbarkeit ist ein klarer Vorteil des Low Coupling Modells.

4.2.5 High Cohesion

Ein klarer Vorteil von High Cohesion ist die Tatsache, dass der Code übersichtlicher und strukturierter wird. Ein Beispiel hierfür sind die Klassen, welche für den Excel-Export verwendet werden. Diese befinden sich im Ordner Excel und sind klar betitelt, sodass der Programmierer gleich wissen sollte, was welche Klasse tut. Die Klassen, auf die hier verwiesen wird sind die folgenden: CellFinder, CellWriter, ExcelExporter, RowsCounter und RowsCreator. CellFinder kümmert sich um das Finden einer Zelle in der Excel-Datei, CellWriter um das Schreiben eines Inhalts in eine Zelle der Excel-Datei. ExcelExporter hat nur den Export nach Excel als Aufgabe. RowsCounter zählt die Anzahl der Reihen, welche für einen Export angelegt werden müssen und RowsCreator legt diese Zeilen dann an.

Dadurch, dass die Klassen so klar strukturiert sind, ist es für zukünftige Entwickler viel einfacher etwas zum Beispiel am Excel-Export zu verändern oder zu korrigieren, da sofort klar ist, in welcher Klasse sich der Export befindet.

4.3 DRY

DRY steht für Don´t Repeat yourself und beschäftigt sich mit dupliziertem Code. Das bedeutet, wenn vollständig an das Prinzip gehalten wird, dann präsentiert jeglicher Code eine einzigartige Funktionalität, die nicht ohne den jeweiligen Code schon vorhanden ist.

Im Kapitel Refactoring wurde exemplarisch bereits eine Stelle, in der duplizierter Code vorhanden ist, verbessert. Des Weiteren wurden Methoden, die in mehreren

Klassen benötigt wurden, abstrahiert und in eine Component-Klasse hinzugefügt. Diese enthält beispielsweise Methoden wie ShufflePlayer, welche häufiger in der Anwendung benötigt werden.

Die Methode CheckIfAmountOfPlayerIsPowerOfTwo wurde zweimal definiert. Einmal in TournamentGenerator und einmal in der ChooseHelper Klasse. Da zweiteres nur eine Exktraktion von Funktionalitäten aus TournamentGenerator war, erbt ChooseHelper nun von dieser Klasse, sodass der Code nicht doppelt vorhanden ist. Der Commit dazu ist b271d90.

In den Klassen SingleElimination und DoubleElimination wird jeweils zwei Mal derselbe Code geschrieben, wenn ein Logging erstellt werden soll. Damit dies nicht mehr der Fall ist, wurde eine Methode in der Component-Klasse geschrieben, die genau diesen duplizierten Code beinhaltet. Dadurch, dass beide Klassen von dieser Component-Klasse Erben, wurde der Doppelte Code an diesem Abschnitt verringert. Der Commit dazu ist d4a3f48.

Jedoch ist immer noch an manchen Stellen duplizierter Code. Ein Beispiel hierfür ist die Eingabe wer ein Gewinner aus einem Spiel ist. Dies ist jeweils teilweise in SingleElimination und in DoubleElimination implementiert, da der Ablauf dazu unterschiedlich ist.

5 Refactoring

Refactoring wird angewandt, um die Codequalität zu verbessern. Das bedeutet, dass der Code einfacher lesbar wird, flexibler genutzt werden kann oder die Struktur an die Problemdomäne angepasst wird. Im Nachfolgenden wird exemplarisch ein Code Smell untersucht und mithilfe von Refactoring verbessert.

5.1 Zyklomatische Komplexität

Für das Refactoring wurde eine Methode der TournamentLog-Klasse betrachtet. Die GetLetterByNumber-Methode gibt ausgehend von einer Zahl einen Buchstaben zurück, welcher durch die Zahl repräsentiert wird. Bei einer 1 wird also ein A zurückgegeben, bei einer 2 ein B usw. Bisher wurde diese Methode mit einem switch-case gelöst. Für jede mögliche Zahl gibt es also einen case, durch den dann der passende Buchstabe zurückgegeben wird. Eine Analyse des Programmcodes zeigt, dass die zyklomatische Komplexität der Methode bei 28 liegt.

Bei der zyklomatischen Komplexität handelt es sich um eine Codemetrik, mit der die Komplexität des Codeteils (Methode, allgemeines Stück Programmcode, …) gemessen wird. Ein hoher Wert der Komplexität bedeutet, dass der Codeteil nur schwer von einem Menschen begreifbar ist. Idealerweise sollte der Wert nicht größer als 10 sein. Eine zyklomatische Komplexität von 28 ist also viel zu hoch.

Die nachfolgende Abbildung zeigt den bisherigen Ablaufplan der Methode.

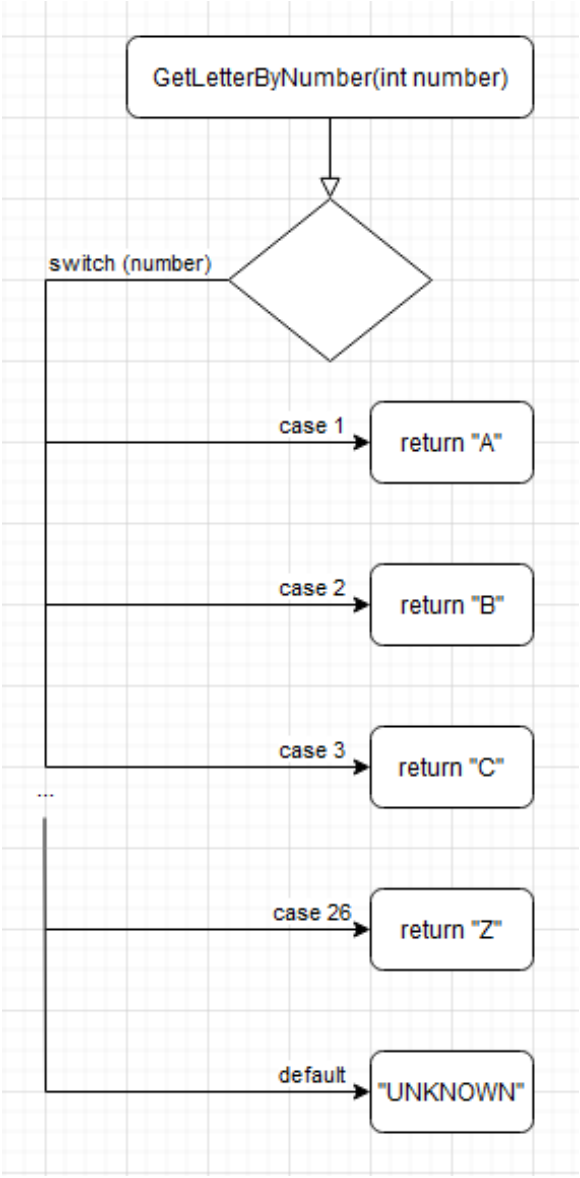

Abbildung 1: Bisheriger Ablaufplan der Methode

In der Abbildung sind die verschiedenen cases des switch-Statements zu sehen. Die Stelle zwischen „case 3“ und „case 26“ ist aus Platzgründen durch drei Punkte abgekürzt. Damit sind die cases 4 – 25 gemeint.

Die Methode wurde umgebaut, sodass diese verständlicher wird. Die nachfolgende Abbildung zeigt den neuen Programmablauf der Methode.

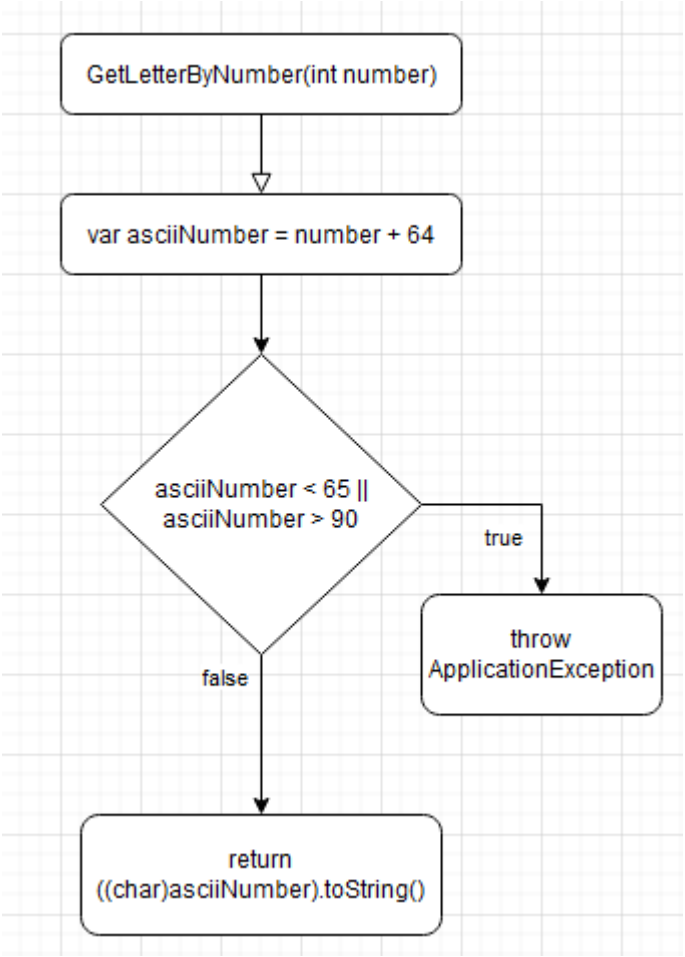

Abbildung 2: Ablaufplan der Methode nach Umbau

Durch den Umbau der Methode wurde eine zyklomatische Komplexität von 3 erreicht. Die Methode ist nun viel besser lesbar und vor allem viel verständlicher für den Menschen, der sich die Methode anschaut.

Die Änderung ist am Commit 3733d4e nachvollziehbar.

5.2 Duplicated Code

Doppelter Code ist der am häufigsten vorkommende Code Smell. Auch in unserem Software Projekt ist doppelter Code vorhanden der beseitigt werden sollte. Die Klasse DoubleElimination verwendet zwei unterschiedliche Methoden um einen Turnierbaum anzuzeigen. Eine Methode, um den Gewinner Baum anzuzeigen und eine Methode um den Verlierer Baum anzuzeigen. Beides könnte aber auch in einer Methode umgesetzt werden, wenn ein Parameter angegeben wird, der eine Liste von Spielern entgegennimmt.

Da SingleElimination sehr ähnlich zu DoubleElimination ist, befindet sich hier ebenfalls die CreateTree-Methode, da sie ursprünglich aus dem Interface ITournamentTree stammt.

Des Weiteren ist aufgefallen, dass in beiden Klassen die Methode NoFreeWinsAgainstEachOther() geschrieben wurde. Dies ist eine Methode die prüft, ob Frei-Lose (Wildcards) nicht gegeneinander spielen in der ersten Runde, damit diese nicht in die nächsten Runden gelangen können.

In der nachfolgenden Abbildung ist ein UML-Klassendiagramm mit den beiden erwähnten Klassen sowie dem Interface und einer Component Klasse welche bisher nur 2 Methoden beinhaltet, die Spiele und Spieler mischt. Da man diese Methoden häufiger benötigt, wurde diese Klasse erstellt.

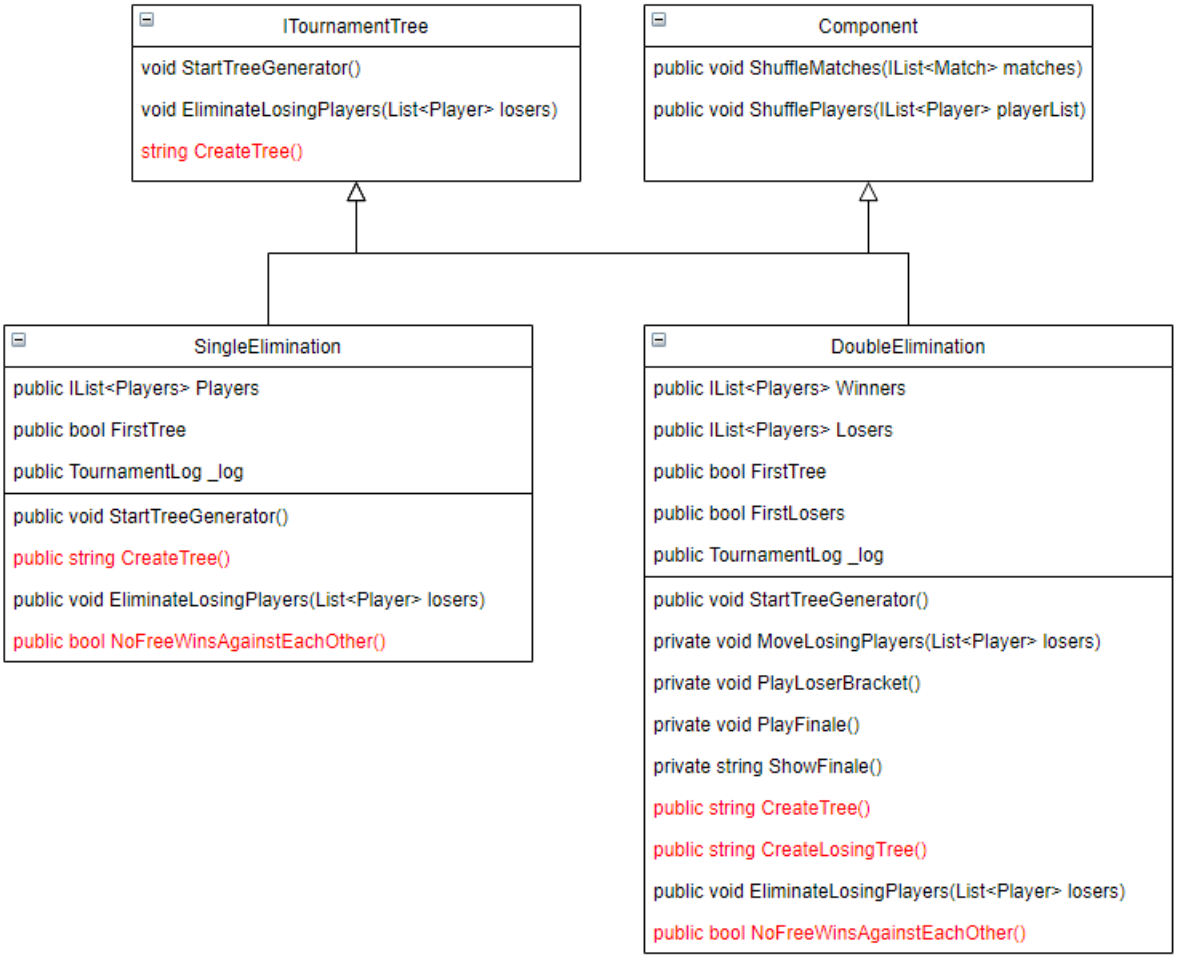

Abbildung 3: Doppelter Code anhand TournamentTree

Als Erstes wurde die Methode NoFreeWinsAgainstEachOther() ausgelagert in die Component Klasse. Dazu wurde ein Parameter hinzugefügt, der eine Liste von Spielern entgegennimmt. Damit ist in beiden Klassen eine Methode weniger definiert.

Damit CreateTree für den Gewinner Baum und den Verlierer Baum benutzt werden kann, wurde ein Parameter hinzugefügt, der eine Liste von Spieler repräsentiert. Damit können entweder die Spieler des Gewinner Baums oder des Verlierer Baums verwendet werden. Zur besseren Übersichtlichkeit wurde noch ein Optionaler Parameter zur Methode hinzugefügt, der einen String darstellt. Mit diesem String

kann in der Anwendung beim Erstellen des Baums mitgegeben werden, ob es sich um einen Gewinner Baum handelt oder um einen Verlierer Baum.

Der zuständige Commit ist: b51687e

In der Nachfolgenden Abbildung sind die Änderungen zu sehen, welche Vorhandenen Doppelten Code vermindert hat.

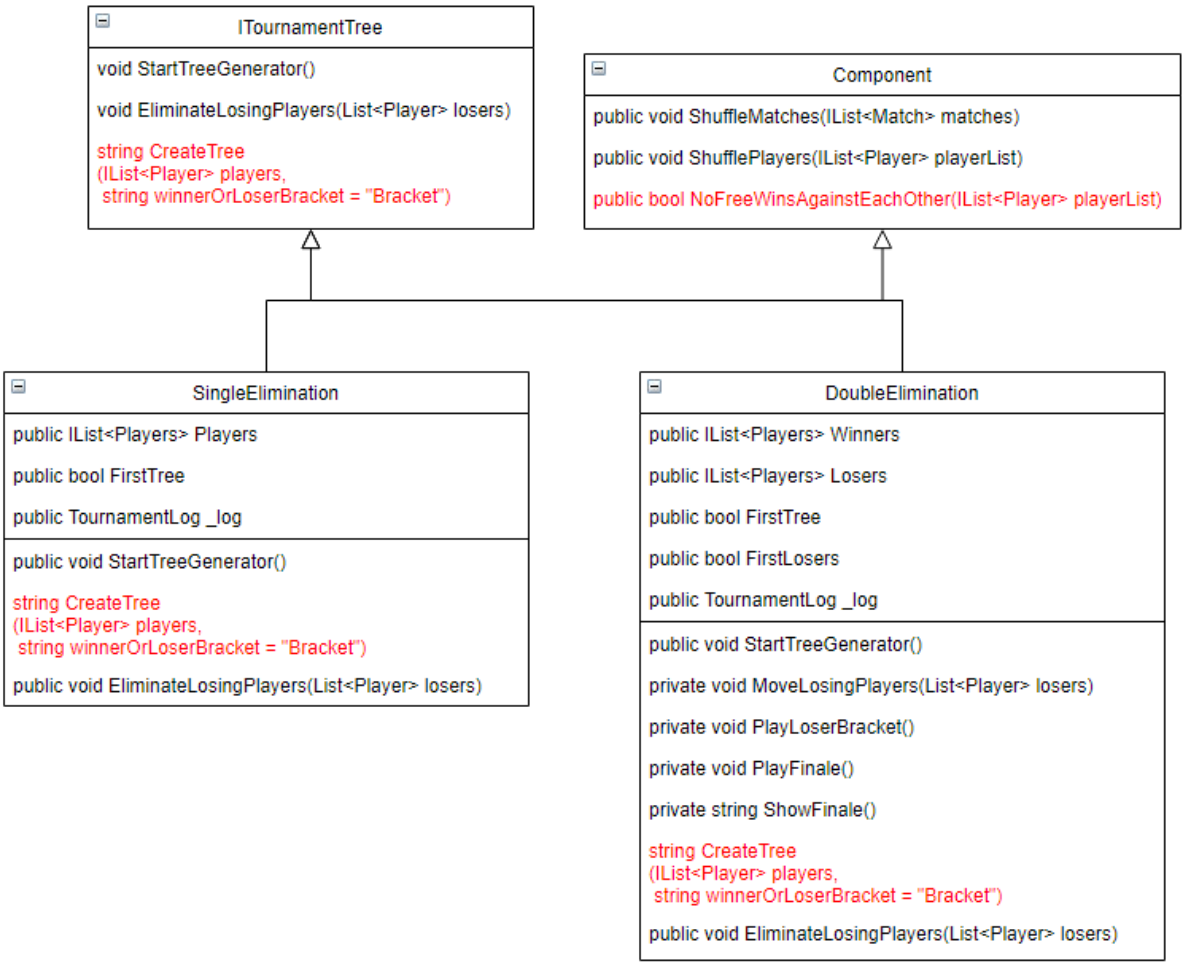

Abbildung 4: Doppelten Code vermindert in TournamentTree

6 Legacy Code

Legacy Code bedeutet, dass der vorhandene Code schwer wartbar und schwer änderbar ist. Das entsteht beispielsweise dadurch, dass einfach alter Code übernommen wird, Code verändert oder erweitert wird, oder einfach keine Tests für den Code geschrieben werden.

6.1 Legacy-Code anhand einer Export-Methode

Auch für diesen Punkt wird wieder eine Export-Methode nach Excel betrachtet, die Methode für den Export des Turnierbaums. Diese Methode hat etwas weniger als 200 Zeilen und ist für Leute, die die Methode nicht erstellt haben, relativ undurchsichtig und schwer zu verstehen. Die zyklomatische Komplexität der Methode beträgt 9, was eigentlich ein akzeptabler Wert ist, durch die Länge der Methode wirkt diese aber dennoch unübersichtlich und schwer zu verstehen.

Mithilfe der „Extract Method“-Technik wurde dieser Code so angepasst, dass aus der gesamten Methode möglichst kleine Methoden erstellt worden sind, welche so kompakt sind, dass diese auch für neue Programmierer an der Software leicht verständlich sind. Die neu erstellten Methoden wurden alle auf private gestellt, sodass diese nicht von außen aufgerufen werden können, sondern nur in der Klasse selbst. Die zyklomatische Komplexität der Hauptmethode konnte so auf 1 gesenkt werden. Da es für diese Methode bereits UnitTests gibt (siehe *Tests für den Excel-Export*), kann die Änderung durch den erfolgreichen Test als sinnvoll und funktionierend angesehen werden.

Die Anpassung des Legacy-Codes sind in Commit 29713d9 erkennbar.

6.2 Legacy Code Group & GroupPhase

Bei der Klasse Group und der Klasse GroupPhase werden in beiden Fällen im Konstruktor Listen erstellt. Dadurch entstehen Abhängigkeiten die lieber gebrochen werden sollten. Dazu wird die Technik Parametrize Constructor verwendet. Es muss an keiner anderen Stelle der Code verändert werden, da die alte Funktionalität vollständig erhalten bleibt.

In der Klasse Group wird eine Liste von Spielern im Konstruktor angelegt. Um die Abhängigkeit zu brechen, wurde ein weiterer Konstruktor erstellt, der eine Liste von Spielern als Parameter zusätzlich hat. Im alten Konstruktor wurde lediglich die ID der Gruppe mitgegeben. Der alte Konstruktor verweist dabei auf den neu erstellten Konstruktor.

Die Änderung ist im Commit b1919d9 zu finden.

Bei der GroupPhase ist ein ähnlicher Fall. Hier wurden im Konstruktor lediglich die Spieler an eine Property mit einem Parameter übergeben und die Gruppen wurden im Konstruktor initialisiert. Um diese Abhängigkeit zu brechen, wurde wieder ein neuer Konstruktor erstellt, der einen zusätzlichen Parameter entgegennimmt, welche die Gruppen beinhaltet. Der alte Konstruktor behält die Funktionalität bei und verweist lediglich auf den neu erstellten Konstruktor

Die Änderung ist im Commit 6793d34 zu finden.

Mit diesen Änderungen ist man in der Lage beispielsweise FakeObjekte zu erstellen und diese direkt an die neu erstellten Konstruktoren zu übergeben. Das hat den Vorteil schneller und besser Tests durchzuführen.

7 Entwurfsmuster

Entwurfsmuster werden verwendet, um Lösungen für wiederkehrende Probleme zu liefern. Sie helfen komplexe Softwaresysteme zu verstehen und zu beherrschen sowie die Kommunikation zwischen den Entwicklern zu beschleunigen und zu vereinfachen. Die Muster sind in Kategorien aufgeteilt. Diese sind Erzeugungsmuster, Strukturmuster und Verhaltensmuster. Im Nachfolgenden wird exemplarisch auf zwei Entwurfsmuster eingegangen.

7.1 Factory Pattern

Das Factory Pattern ist ein Entwurfsmuster in der Softwareentwicklung. Es gehört zu den Erzeugungsmustern und bedeutet, dass ein Objekt durch den Aufruf einer Methode statt dem Konstruktor erzeugt wird. Dies wurde für die Excel-Export-Klassen umgesetzt. Hierfür wurde zuerst ein Interface IExcelExport implementiert. Von diesem Interface erben die drei ExcelExport-Klassen. In der neuen ExcelExportFactory-Klasse gibt es eine Methode Create, welche ein IExcelExport-Objekt zurückgibt. Dadurch, dass die ExcelExport-Klassen von diesem Interface erben, können diese auch in der Methode zurückgegeben werden. Im Code, welcher für den eigentlichen Export aufgerufen wird, wird jetzt einfach statt den Konstruktoren der einzelnen Klassen für jede Klasse einmal die Methode der Factory aufgerufen. Dies ist im Commit 05d632d zu sehen. Die folgende Abbildung zeigt dieses Vorhaben in einem UML-Klassendiagramm.

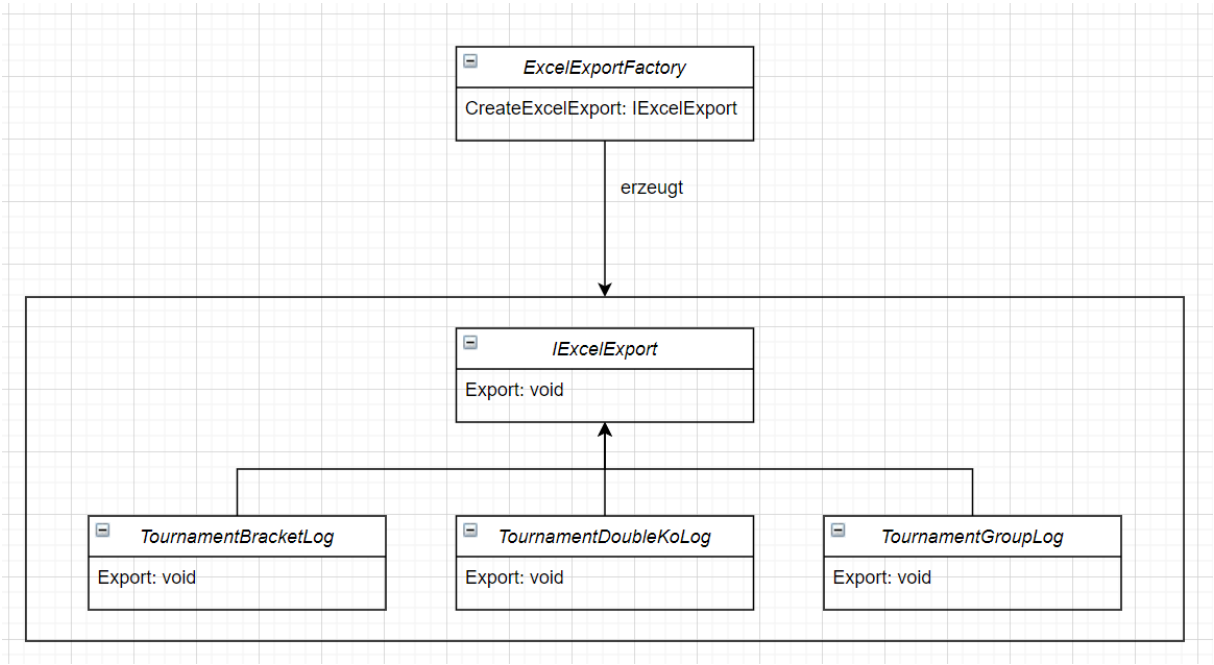

Abbildung 5: Aufbau des Factory Pattern

7.2 Entwurfsmuster Strategie

Anhand von SingleElimination und DoubleElimination kann das Strategy-Pattern gut angewandt werden. Ganz zu Beginn waren im Projekt lediglich zwei Klassen eine für SingleElimination und eine für das DoppelKO-System. Anschließend wurde ein Interface namens ITournamentTree für diese beiden Klassen erstellt, da sie offensichtlich beides Turnier Bäume darstellen. Dazu ist folgendes UML-Klassendiagramm erstellt worden.

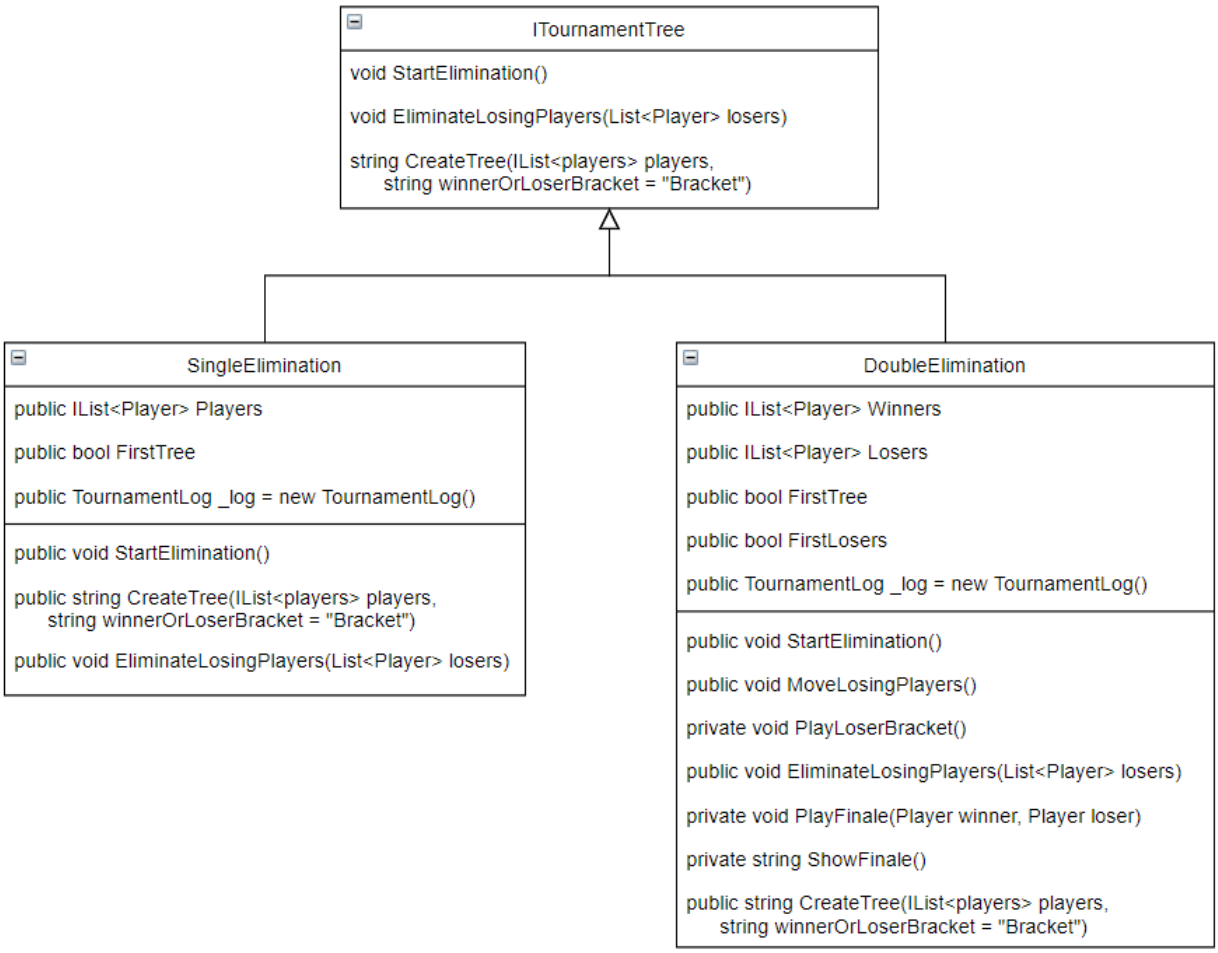

Abbildung 6: Strategie-Muster vorher

Das ist aber immer noch nicht gut umgesetzt, da man hierfür das Strategie-Muster anwenden kann. Dazu wurde das Interface ITournamentTree in IElimination geändert. Des Weiteren wurde eine neue Klasse namens TournamentTree erstellt. Diese neue Klasse besitzt eine private Instanz Variable, welche ein IElimination darstellt. Je Nachdem welches Eliminationsverfahren verwendet wird, kann diese Variable entweder SingleElimination oder DoubleElimination sein. Man kann der Klasse TournamentTree beim Konstruktor das Eliminations-System übergeben oder durch Aufrufen der SetElimination Methode. Tournament Tree ruft dann jeweils die StartEliminate Methode der beiden anderen Klassen mit StartTreeGenerator auf. Dazu ist folgendes UML-Klassendiagramm erstellt worden.

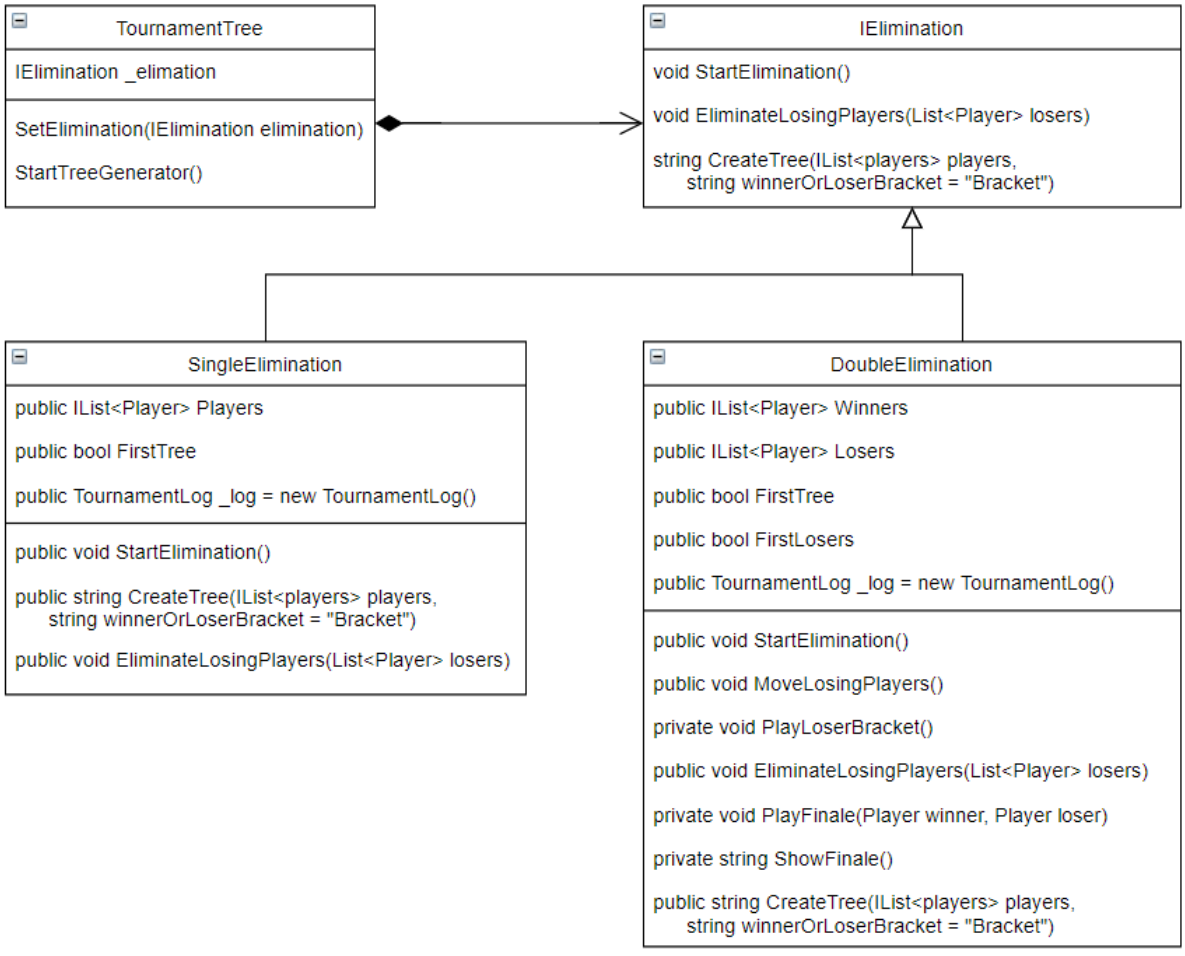

Abbildung 7: Strategie-Muster nachher

Mithilfe dieser Änderung ist das Auswählen des Turnier-Baums deutlich übersichtlicher gestaltet. Im TournamentGenerator wird nämlich bevor der Benutzer eine Wahl trifft ein TournamentTree-Objekt erstellt. Sobald dann die Wahl in Abfragen getroffen ist, muss nur mit der SetElimination-Methode das Eliminationsverfahren ausgewählt werden. Anschließend kann dann das zu Beginn erstellte TournamentTree-Objekt die Methode StartTreeGenerator ausführen.

Der Commit für dieses Entwurfsmuster ist b4509e5.

8 Domain Driven Design

Damit die Software gut verstanden und benutzt werden kann, muss eine Sprache verwendet werden, die die Entwickler sowie die Experten über die Domäne verstehen. Hierzu muss zunächst die Domäne betrachtet werden, um diese dann auf den Quellcode zu implizieren.

In der Domäne geht es um die Generierung eines Turniers. Dazu treten Spieler gegeneinander an, bis ein Sieger feststeht. Auf dem Weg dahin können Spieler in der Gruppenphase in Gruppen gegeneinander in Spielen antreten. Dabei spielen dann anschließend die besten Spieler gegeneinander in einem Turnierbaum an. Es gibt auch die Möglichkeit diesen Turnierbaum direkt, ohne eine Gruppenphase zu erstellen. Spieler, die dann im Turnierbaum verlieren, scheiden aus, außer man verwendet ein Doppel KO-System. Hier müssen Spieler ein weiteres Mal verlieren bis sie aus dem Turnier ausscheiden. In Diesem Doppel KO-System gibt es ein Gewinnerbaum für die, die noch nicht verloren haben und einen Verliererbaum, welcher für diejenigen sind, die bereits einmal verloren haben.

Damit ist die Domäne für unsere Software abgedeckt. In der nachfolgenden Tabelle ist der Übergang von der Domäne in den Quellcode. Wir haben uns dafür entschieden, den Code so nah wie möglich an der Domäne zu gestalten, um somit eine verständliche „Ubiquitous Language“ zu schaffen. Jedoch haben wir uns für die englische Variante im Quellcode entschieden.

Domäne	Quellcode
Spieler	Player
Spiel	Match
Gruppenphase	GroupPhase
Gruppen	Group
Turnierbaum	Tournament Tree
Gewinnerbaum	Winner-Bracket
Verliererbaum	Loser-Bracket
Einzelausscheidung	SingleElimination
Doppel KO-System	DoubleElimination
Frei-Lose/ Wildcard (Sicherer Sieg)	Wildcard
Spieler scheiden aus	EliminateLosingPlayer()
Turniergenerator	Tournament Generator

Anhand dieser Beispiele wird deutlich, dass versucht wird die Begriffe der Domäne genauso in den Quellcode zu integrieren. So entsteht keine Kluft im gegenseitigen Verständnis und es wurden keine eigenen Begriffe in der Software verwendet. Aufgrund der Tatsache, dass man in diesem Projekt Domänen-Experte und gleichzeitig Entwickler der Software ist, konnte direkt darauf geachtet werden, wie die Domäne im Quellcode verständlich implementiert wird.

Value Objects sind einfache Objekte, welche keine eigene Identität besitzen. Es wird nur durch seine Werte und Eigenschaften beschrieben. Ein Beispiel in unserem Code ist die Klasse Player. In unserem Fall ist ein Player, also ein Spieler im Turnier, gleich zu einem anderen Player-Objekt, wenn sie im Namen und in der ID übereinstimmen. Dadurch, dass ein Player jedoch mehr als diese zwei Properties hat, kann er nicht direkt als Value Object angesehen werden. Hierfür erfolgt folgende Änderung: Es werden neue Klassen für den Namen und die ID des Spielers erzeugt

– Name und Identification. Die Klasse Name erhält hier einen String und die Klasse Identification einen int. Diese werden genau durch die Properties beschrieben, sodass es sich bei diesen beiden Klassen um Value Objects handelt. Dies kann im Commit 6da8a2b nachvollzogen werden.

Ein Beispiel für eine Entity ist ebenfalls die Klasse Player. In einem früheren Stadium der Software war es so, dass ein Spieler nur anhand eines Namens erzeugt wurde. Dies führte zu Problemen, wenn ein Turnier mit Personen gespielt wurde, welche den gleichen Namen haben. So war nie klar, welches Player-Objekt jetzt zu welcher Person gehört. Eine Änderung aus Commit 656c790 führte dazu, dass Player-Objekte beim Erzeugen nun eine automatisch generierte ID erhalten, sodass diese nun von Spielern, die denselben Namen haben, auseinandergehalten werden können. Ein Player ist nun also durch seine ID und seinen Namen eindeutig. Des Weiteren ist die ID an sich auch immer eindeutig, da sie hochgezählt wird und bei jedem Spieler einzeln gesetzt wird.

Für Aggregates haben wir einige Beispiele im Code. So ist die Klasse Player ein Aggregate für die Klassen Name und Identification und die restlichen Properties von Player. Ohne diese Klasse wäre es zum Beispiel sehr schwierig für die Group-Klasse die Informationen der Spieler wie zum Beispiel Name, Tore, Punkte zu erhalten und korrekt zu sortieren. Des Weiteren ist die Zusammenkunft von Group-Objekt, alle Spieler der Gruppe und den in der Gruppe gespielten Matches ein Aggregate, welches als „Gruppenphase“ betitelt werden könnte. Die Gruppenphase besteht aus den eben genannten Entitäten, sodass diese die Komplexität reduziert.

Ein Beispiel für ein Repository wäre in unserem Fall die Klasse GroupPhase. Wie bereits beschrieben wird eine Gruppenphase durch alle Gruppen, die Spieler in den

Gruppen und den Spielen in den Gruppen beschrieben. Hierbei kümmert sich die Klasse GroupPhase darum, dass die Gruppenphase korrekt abläuft, also dass alle Spiele gespielt werden um am Ende die besten Spieler aus den Gruppen in die nächste Phase des Turniers kommen. Eine weitere Klasse, welche die Gruppenphase bearbeitet, gibt es nicht im Code, sodass GroupPhase eindeutig als Repository gesehen werden kann.

9 Clean Architecture

Clean Architecture wird angewandt, um eine nachhaltige Architektur zu schaffen. Das macht den Code langlebiger und verständlicher. Um dies zu ermöglichen gibt es verschiedene Strukturen, die eingehalten werden müssen. Die Struktur der Clean Architecture besteht aus Abstraction Code, Domain Code, Application Code, Adapters und Plugins. Dabei ist Abstraction Code die innerste Schicht und Plugins die äußerste Schicht. Wichtig ist dabei, dass die Inneren Schichten nichts von den Äußeren Wissen und die Abhängigkeiten immer von außen nach innen gerichtet sind.

Im Nachfolgenden werden zwei Beispiele von Clean Architecture anhand unserer Software Tournament Generator gezeigt.

9.1 Clean Architecture anhand der Klasse Match

Die Klasse Match stellt ein Spiel eines Turniers da. Damit für diese Klasse das Konzept der Clean Architecture zählt, müssen einige Änderungen erfolgen. Die Klasse Match gehört zum Teil der Entities, der innersten Schicht. Hier ist es üblich, dass die Klasse eine einfache Datenstruktur beherrscht. Dies ist bereits gegeben, da in der Klasse aber noch verschiedene Methoden aufgerufen werden, werden diese Methoden in die zweite Schicht – die Schicht der Use-Cases gezogen. Hierfür wird eine neue Klasse MatchFactory erzeugt, in welcher die Methoden eingefügt werden. Die Klasse Match darf hierbei keinen Verweis auf die neue Klasse haben, da es sonst zu zirkulären Verweisen kommt, was nicht der Clean Architecture entspricht. Dadurch, dass die Methoden aus der Klasse Match nun in die Klasse MatchFactory

gewandert sind, wird bei einer Änderung der Methoden die Klasse Match nicht mehr angepasst, sodass es keine Auswirkungen auf diese Klasse hat.

Da es für ein Match nicht die Möglichkeit gibt, dass dieses im Nachhinein noch in der Software angezeigt wird, ist eine Methode in der Adapterschicht für Match nicht nötig. Deswegen wurde eine Klasse MatchAdapter hergestellt, welche eine Methode für die Klasse TournamentGroupLogMatch bereitstellt. So kann gezeigt werden, dass das Prinzip der Adapterschicht dennoch verstanden wird.

Ein TournamentGroupLogMatch wird für die Anzeige im Excel-Export benötigt. Für die Anzeige ist es nicht wichtig, dass die Ergebnisse der einzelnen Spiele als integer gespeichert werden, sodass in der Adapterklasse eine Konvertierung zu string stattfindet.

Die Änderungen sind in Commit 1a9443c zu finden.

9.2 Clean Architecture anhand der Klasse Group

Die Klasse Group stellt eine Gruppe eines Turniers da. In einem Turnier, in dem nicht direkt ein Turnierbaum erstellt wird, oder werden kann, können auch mehrere Gruppen vorhanden sein. In den Gruppen sind dann jeweils mehrere Spieler, von denen die besten in einem Turnierbaum gegeneinander spielen.

Damit hierbei das Konzept der Clean Architecture angewandt werden kann, muss die Klasse geändert werden. Die Gruppe bisher hatte auch die Möglichkeit Spieler hinzuzufügen und die Spieler zu sortieren. Diese Methoden wurden in eine andere Klasse namens GroupFactory verschoben, sodass Group nur noch eine einfache Datenstruktur aufweist. Die Klasse Gruppe gehört also nun zur Schicht der Entität (Domain Code) und die GroupFactory zur Application Code Schicht.

Wichtig ist hier zu sehen, dass Group für sich allein stehen kann und keine Abhängigkeit zur Application Code Schicht entsteht. Sondern die Abhängigkeit von außen nach innen zeigt.

Die nächste Schicht ist die Adapter Schicht. Hierzu wurde ein Group Adapter erstellt. Aufgrund der Tatsache, dass jedoch noch keine Funktionalität dieses Adapters benötigt wird, wurde dieser auch noch nicht implementiert. Falls in der Zukunft eine GUI erstellt wird, indem die Gruppen grafisch dargestellt werden, könnte man sich vorstellen, dass der Adapter benötigt wird, indem eine Konvertierung zu den Komponenten der grafischen Oberfläche gebaut wird. Wichtig ist in diesem Punkt, dass die Arbeit dann erledigt werden soll, wenn sie einen Wert hat.

Die Änderung zu diesem Kapitel befinden sich im Commit 2a6b7d5.

10 API-Design

Eine API wird benutzt, um eine Menge an Funktionen unabhängig von ihrer Implementierung zu definieren. Dabei gibt es Programmiersprachen APIs wie Bibliotheken oder Frameworks und Remote-APIs. Der Einsatz von APIs kann dabei viele Vorteile mit sich bringen wie Stabilität oder Portabilität.

Wie gut eine API ist, wird durch die Qualitätsmerkmale deutlich. Diese sind Benutzbarkeit, Effizienz, Zuverlässigkeit sowie Vollständigkeit und Korrektheit. Eine API soll also für andere leicht verständlich sein, weshalb auch eine gute Dokumentation sehr wichtig ist.

Unsere Anwendung beschäftigt sich mit dem Erstellen eines Turniers, damit Benutzer mithilfe der Anwendung auf schnellem Wege ein Turnier mit anderen Spielern zu absolvieren, ohne weiteren Aufwand oder weitere Kenntnisse über die Struktur eines Turniers zu haben. Das Ziel ist es nicht diese in andere Software zu integrieren bzw. Remote darauf zugreifen zu können.

Deshalb ergibt sich nicht die Möglichkeit eine sinnvolle API bereitzustellen und dazugehörige Methoden oder HTTP Endpunkte zu definieren.