Transfer-Dokumentation-Report (TDR)

**Algorithmen und Datenstrukturen**



**Eingereicht bei**

Petra Hieber

**Verfasst von**

Michaelsen, Robin - 638-0-04073

Härtel, Otto - 638-0-04059

638-0-04059

BiB8aE

Eigenständigkeitserklärung

Ich habe die vorliegende Arbeit im Rahmen des Projekt-Kompetenz-Studiums „2018/2021“ selbstständig verfasst und keine anderen als die angegebenen Quellen, Tools und Hilfsmittel benutzt.

Alle Aussagen in der Arbeit, die wörtlich oder sinngemäß aus gedruckten oder elektronischen Veröffentlichungen oder aus anderen Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die vorliegende Arbeit oder Teile daraus sind noch nicht Leistungsnachweis einer   
vorangegangenen Prüfung gewesen bzw. sind entsprechend als Quelle gekennzeichnet.

Erlangen, den 29.04.2021

Michaelsen, Robin

Erlangen, den 29.04.2021

Härtel, Otto

Erlangen, den 29.04.2021

Rösch, Rene

Inhaltsverzeichnis

[Eigenständigkeitserklärung I](#_Toc70574934)

[Inhaltsverzeichnis II](#_Toc70574935)

[Abkürzungsverzeichnis IV](#_Toc70574936)

[Abbildungsverzeichnis V](#_Toc70574937)

[Tabellenverzeichnis VI](#_Toc70574938)

[1 Einleitung 1](#_Toc70574939)

[2 Funktionalität und Umgebung 1](#_Toc70574940)

[3 Implementierung 1](#_Toc70574941)

[3.1 Verwendete Patterns 1](#_Toc70574942)

[3.1.1 Observer Pattern 1](#_Toc70574943)

[3.1.2 Model View Presenter Pattern 2](#_Toc70574944)

[3.1.3 Data Access Object 4](#_Toc70574945)

[3.1.4 Object Relational Mapping 4](#_Toc70574946)

[3.2 Verwendete Algorithmen 4](#_Toc70574947)

[3.2.1 Compare 4](#_Toc70574948)

[3.2.2 Copy 4](#_Toc70574949)

[3.2.3 Equals 4](#_Toc70574950)

[3.2.4 Logger 4](#_Toc70574951)

[3.3 Beispielhafte Hauptfunktion 6](#_Toc70574952)

[4 Grafische Benutzeroberfläche (in MVP nur der View) 7](#_Toc70574953)

[4.1 Dialoge 7](#_Toc70574954)

[4.2 Fenster 8](#_Toc70574955)

[4.3 Menüs 9](#_Toc70574956)

[4.4 Spezielle Designelemente in der vorliegenden Applikation 9](#_Toc70574957)

[5 Schluss 11](#_Toc70574958)

[Anhang 13](#_Toc70574959)

[Literaturverzeichnis 15](#_Toc70574960)

[Literaturverzeichnis 15](#_Toc70574961)

Abkürzungsverzeichnis

Abb. Abbildung

Abbildungsverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

Tabellenverzeichnis

**Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.**

**//IMMER SICH FRAGEN WURDE DAS WAS? WIE? WARUM?  Designschritte**

**//Überleitungen und Zusammenfassungen & Unternehmerischer Nutzen / Transfer (S.17 / 22 im Balzert)**

# Einleitung

//Robin

# Funktionalität und Umgebung

//Rene

* Java 11.0.9
* Apache Netbeans 12.0
* Apache Derby 10.14.2.0
* Hibernate 5.4.25

//Anwendung allgemein: Verwaltung von Bundesliga 1 bis 3 inkl. Verwaltung der dazugehörigen Clubs

# Implementierung

Im Folgenden wird nun auf eine Auswahl von, in der Applikation verwendeten Patterns eingegangen. Dabei wird nur auf eine Auswahl der verwendeten Patterns eingegangen, die v.a. in der Benutzeroberfläche (GUI), der Persistenz oder der Datenverarbeitung eingesetzt wurden.

## Verwendete Patterns

### Observer Pattern

Beim Beobachter- Muster (observer pattern) handelt es sich um ein objektorientiertes Verhaltensmuster (behavioral pattern). Ziel dieses Musters ist es, dass von einem Objekt abhängige Objekte automatisch benachrichtigt werden, wenn sich das eine Objekt ändert, um ggf. ihre Daten zu aktualisieren. Dies wird durch eine Registrierung der abhängigen Objekte bei dem gewünschten Objekt erreicht. Dieses Muster wird häufig in Verbindung mit GUI- Frameworks und Benutzeroberflächen eingesetzt. Beobachter (Observer) (Objekte die informiert werden wollen) registrieren sich dafür bei den gewünschten Objekten. In Java existiert eine Klasse (Observable, zuständig für Re-/Deregistrierung von Observern, sowie deren Benachrichtigung) und eine Schnittstelle (Observer, für Beanschriftung bei Änderungen an den jeweiligen Observable) um das Observer- Muster umzusetzen.[[1]](#footnote-1)

In der vorliegenden Applikation wurde das Observer Pattern in Verbindung mit dem verwendeten MVP- Pattern eingegangen, auf welchem in unterem Abschnitt noch genauer eingegangen wird.

### Model View Presenter Pattern

Als allgemeine Programmstruktur Programmarchitektur wurde das Model- View- Presenter Pattern gewählt, welches ein Derivat des Model- View- Controller Pattern darstellt.[[2]](#footnote-2)

Das MVC- Pattern wiederum nutzt u.a. das Observer-Pattern. [[3]](#footnote-3)

Eine Anwendung wird durch das MVC- Muster grundlegend in drei Komponenten bzw. Subsysteme aufgeteilt:3

* Model
* View
* Controller

Die Anwendungs- bzw. Geschäftslogik sowie deren Daten wird durch das Model repräsentiert. Die grafische Benutzeroberfläche wird durch die View dargestellt. Die Hauptfunktion des Controllers ist das Steuern der Applikation (u.a. Benutzerinterkation und Veränderung des Models). So fungiert der Controller als Bindeglied und Bekanntmacher zwischen Model und View. Dabei kann ein Model ggf. von mehreren Controllern verändert werden. Zwischen Model und View wird das Observer- Pattern implementiert, so dass sich die jeweiligen Views als Beobachter bei den gewünschten Models registrieren und so bei einer Zustandsänderung des Models benachrichtigt werden, wodurch sie wiederum ihre Darstellung anpassen können.3

Falls eine Applikation komplex aufgebaut ist, kann diese auch aus mehrere kleinere fachliche Sub- Komponenten zusammengesetzt werden. Diese Sub- Komponenten (oftmals eigene View- Controller- Beziehungen) können eigene Verantwortlichkeiten bzgl. der Darstellung oder der Interaktion besitzen. Eine weitere Variante des MVC- Pattern stellt das Model- View- Presenter (MVP) Pattern dar.[[4]](#footnote-4)

Beim MVP- Pattern herrscht eine strenge Trennung zwischen Model und View. Das Model enthält wie beim MVC- Pattern die Geschäftslogik sowie die Daten und kennt weder den View noch den Presenter. Außerdem übernimmt der Presenter die alleinige Steuerung des Models. Der View enthält keinerlei Steuerungslogik. Er akzeptiert nur Benutzereingabe und stellt die gewünschten Daten entsprechend dar. Die Verbindung des Models und des Views erfolgt über den Presenter. Dieser enthält die steuernde Logik, nimmt Benutzereingabe der View entgegen, ändert Daten am Model und leitet diese Änderungen wieder für die Darstellung an den View weiter. Dadurch übernimmt der Presenter die Rolle des Mediators zwischen Model und View. Die View besitzt jedoch keinen Zugriff auf den Presenter oder das Model. Der größte Unterschied zum MVC- Muster besteht darin, dass es beim MVP- Muster keine Assoziation zwischen Model und View gibt.[[5]](#footnote-5)

Beim MVP – Muster kann der View unterschiedliche Ausprägungen besitzen. Einerseits kann der View komplett passiv sein, in dem er bei Benutzerinterkation einen Methodenaufruf beim Presenter veranlasst, der wiederum über Schnittstellen am View und Model Daten aktualisiert und zur Darstellung aufbereitet. Anderseits kann der View auch so implementiert sein, dass er die Möglichkeit besitzt Interaktionen oder Ereignisse selbstständig zu verarbeiten.[[6]](#footnote-6)

In der vorliegenden Applikation wurde eine Kombination aus MVC und MVP- Muster angewendet. Es existierte eine strikte Trennung zwischen Model und View ähnlich wie im MVP- Muster vorhanden ist, jedoch werden die Benutzereingabe bzw. Interaktion wie im MVC- Muster direkt von der, im vorliegenden Falle, als Presenter bezeichneten Instanz wahrgenommen, ohne einen expliziten Aufruf in der jeweiligen View. Zwischen Views und Presenter existiert in der vorliegenden Applikation eine 1:1 Beziehung. Zudem bleibt der Presenter dem dazugehörigem View verborgen. Der Presenter übernimmt als Action- Listener die gesamte Steuerung der Applikation, in dem er u.a. auf Benutzereingaben reagiert, Benutzeroberflächenelemente hinzufügen, aktivieren oder auch deaktivieren kann oder das jeweilige Model ändern kann. Ähnlich wie in dem Buch von Balzert H (S.67f.) beschrieben kann ein Presenter je nach gewünschter Aktion auch die Erzeugung weiterer Presenter bzw. Views und deren Verknüpfung oder das Wechseln des Models übernehmen. Die Datenkonsitenz der Applikation wird durch die Implementierung des Observer- Musters vom MainPresenter, der jeweils als Observer bei allen Ligen hinzugefügt wird, (ähnlich wie im MVC- Pattern (siehe Balzert S.65)) gewährleistet. Jedoch werden einzelne Views nicht direkt mit dem Liga- Model als Observer verknüpft, sondern nur der MainPresenter dient als Observer für das Liga- Modell. Das Liga- Modell wird bei weiteren erzeugten View- Presneter Beziehungen jeweils als Referenz übergeben. Bei Änderungen des Liga- Models wird dadurch nur der MainPresenter benachrichtigt, der die Daten des aktuell anzuzeigenden Modells an einer Stelle aktualisiert und veranlasst den anzuzeigenden View neu zu zeichnen.

### Data Access Object

//Rene

### Object Relational Mapping

//Robin + Rene

## Verwendete Algorithmen

//Theorie (kleiner Absatz)

### Compare

//Robin

### Copy

//Otto

### Equals

//Robin

### Logger

//Rene

Beim

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

Dies geschieht über die Implementierung der ActionListener- Schnittstelle in den jeweiligen Presentern, welche jeweils eine ActionPerformed Methode überschreiben. Zwischen Views und Presenter existiert in der vorliegenden Applikation eine 1:1 Beziehung. Zudem bleibt der jeweiligen View dem dazugehörigen Presenter verborgen. Da der Presenter die jeweiligen Views kennt, fügt sich dieser selbst als Action- Listener für z.B. jeweilige Buttons oder auch Click- Ereignisse in Tabellen oder Listen hinzu. In der jeweiligen View wird jeder Button mit einem passenden ActionCommand versehen, welche anschließend im Presenter in der ActionPerformed Methode verarbeitet wird, um die jeweils korrekte Aktion auszuführen. Der Presenter übernimmt als Action- Listener die gesamte Steuerung der Applikation, in dem er u.a. Benutzeroberflächenelemente hinzufügen, aktivieren oder auch deaktivieren kann. Ähnlich wie in dem Buch von Balzert H (S.67f.) beschrieben kann ein Presenter je nach gewünschter Aktion auch die Erzeugung weiterer Presenter bzw. Views und deren Verknüpfung oder das Wechseln des Models übernehmen. Ein Beispiel für das Wechseln des Models in der vorliegenden Applikation ist u.a. das Umschalten der Liga (z.B. Liga 1 zu Liga 2) über Knöpfe im MainView, welches wiederum den MainPresenter veranlasst das anzuzeigende Datenmodell (Liga) zu ändern, welches anschließend durch den jeweils ausgewählten View in der gewünschten Form (z.B. Tabelle, Spielplan, Clubs) dargestellt wird. Ein Wechsel von Presenter- View Beziehungen liegt u.a. im MainPresenter vor, wenn von der Tabellen- View in den Spielplan- View gewechselt wird. Hierbei wird der View angepasst und der dazu passende Presenter erzeugt. In der vorliegenden Applikation wird mit einem einzigen Datenmodell (Liga) als Grundlage gearbeitet. In diesem Datenmodell sind Clubs als auch Spiele enthalten. Dieses Datenmodell wird beginnend im MainPresenter an die jeweiligen (Sub)- Presenter- View- Beziehungen als Referenz weitergereicht, die ggf. vom MainPresenter ausgewählt bzw. angesteuert werden (z.B. ClubPresenter und ClubView). Die Datenkonsitenz der Applikation wird durch die Implementierung des Observer- Patterns vom MainPresenter, der jeweils als Observer bei allen Ligen hinzugefügt wird, (ähnlich wie im MVC- Pattern (siehe Balzert S.65)) gewährleistet. Jedoch werden einzelne Views nicht direkt mit dem Liga- Model als Observer verknüpft, sondern nur der MainPresenter dient als Observer für das Liga- Modell. Bei Änderungen des Liga- Models wird dadurch nur der MainPresenter benachrichtigt, der die Daten des aktuell anzuzeigenden Modells an einer Stelle aktualisiert und veranlasst den anzuzeigenden View neu zu zeichnen. Dieses System funktioniert über verschiedene View- Presenter- Beziehungen hinweg. Dies ist möglich, da jeweils nur das anzuzeigende Liga- Model aktualisiert wird und kaskadierend an die bestehende View- Presenter- Auswahl weiter gereicht wird.

Es wurde bewusst diese Art der Datenkonsistenz zwischen Model und View gewählt, da die Applikation einen komplexen Aufbau aus verschiedensten Presenter – View – Beziehungen aufweist, die wiederum weitere Presenter- View – Beziehungen erzeugen können. So ist die Datenkonsistenz über alle Views zu jedem Zeitpunkt gegeben.

## Beispielhafte Hauptfunktion

//Otto

//Hauptfunktion: Spiel hinzufügen (manuell) 🡪 dafür in der jeweiligen Liga wechsel in den Spielplan View 🡪 ändert Ansicht

🡪 Auf Knopf drücken Spiel hinzufügen 🡪 action command wird an PlanPresenter gesendet 🡪 dieser verarbeitet actioncommand mittels switch case und erzeugt neues Fenster inkl. Passenden Presente für Spielhinzufügen (PlanAddGameView & PlanAddGamePresenter) und übergibt dem Presenter den View, die gerade ausgewählte Liga, und den gerade existierenden MainView (da PlanAddGameView ein JDialog ist und dieser ein Vaterfenster braucht) MainView wird nochmals extra dem PlanAddGameView übergeben, da dieser als JDialog einen ParentView braucht

🡪 PlanAddGamePresenter passt beim Initialisieren der anzuzeigenden View die jeweiligen anzuzeigenden Daten, an die jeweilige An, die den Presenter aufruft (um nur spiele zwischen Teams aus bestimmten verschiedenen Ligen zu ermöglichen)

🡪 auf den jeweiligen ComoBoBoxen für die einzelnen Ligen und den einzelnen Club wird der PlanAddGamePresenter als Itemlistener hinzugefügt 🡪 bei Änderung der Liga wird er so benachrichtigt und passt die anzuzeigenden Clubs in der JComboBox und in den zwei jeweiligen JLabels des PlanAddGameView an, die übersichtshalber, die beiden Clubs anzeigen, die gegeneinander spielen des PlanAddGameView an

🡪 wenn Club ausgewählt und Datum eingeben wird bei Druck auf Speil hinzufügen knopf 🡪 Aktion weiter geleitet an PlanAddGamePresenter 🡪 dieser überprüft ob Clubs unterschiedlich sich, ob das datum dem korrekten Format entspricht 🡪 Falls alle eingaben stimmen 🡪 Game Objekt Angelegt mit Parametern (Team A, Team B, Datum , Liga A, Liga B) die jeweils involvierten Ligen zu aktualisieren

//Prüfung im Model (generierung der GameIDs)

//Falls dieses erfolgreich 🡪 kommt Message Dialog mit erfolgreicher Bestätigung, Falls es hierbei zu einem Fehler kommt 🡪 Message Dialog mit Fehlermeldung und spiel wird nicht angelegt

# Grafische Benutzeroberfläche (in MVP nur der View)

Im folgenden wird genauer auf die Gestaltungsweise der grafischen Benutzeroberfläche mit Hilfe der theoretischen Grundlagen von Frau Prof. Balzert eingegangen.

Die grafische Benutzeroberfläche (GUI, Graphical User Interface) entspricht der View- Komponente im MVC bzw. MVP- Pattern. Sie erzeugt Fenster, Menüs sowie die Dialogführung. Zu einer GUI gehören jeweils eine E/A- Komponente (Gestaltung der Information) sowie eine Dialogkomponente (Bedienungsabläufe). Diese grafische Benutzeroberfläche wird mittels eines GUI- Systems (in dem vorliegenden Anwendungsfall Windows) verwaltet. [[7]](#footnote-7)

* 1. Dialoge

Für die Interaktion zwischen einem Dialogsystem und einem Benutzer sowie die Erreichung eines bestimmten Zieles damit wird ein Dialog verwendet. Dabei existieren zwei Arten von Dialogen: Primär- und Sekundärdialog. Primärdialoge dienen der direkten Aufgabenerfüllung und werden bei Fertigstellung der zu bearbeitenden Aufgabe beendet. Sekundärdialoge dienen zum Darstellen und Anzeigen von Zusatzinformationen. Dialoge können außerdem unterschiedliche Modi besitzen. So gibt es modale Dialoge (Schließung vor Öffnung eines weiteren Applikationsfensters notwendig) und nicht modale Dialoge (Schließung vor Öffnung eines weiteren Applikationsfensters nicht notwendig).[[8]](#footnote-8)

In der vorliegenden Applikation stellt das Hauptfenster (Klasse MainView) den Primärdialog dar von welchem aus weitere Dialoge angesteuert bzw. erzeugt werden können. Die Applikation besitzt wenig Sekundärdialoge (z.B. Kader-View- bzw. Dialog oder auch den PlanAddGame- View bzw. Dialog). Die Sekundärdialoge, die existieren sind zudem alle nicht- modal. Dies wurde gewählt, damit die Datenkonsitenz zu jedem Zeitpunkt der Applikation gegeben ist.

Dialoge können einer objektorientierten (zuerst Auswahl des zu bearbeitenden Objektes, anschließend Auswahl der anzuwendenden Funktion auf das Objekt) oder funktionsorientierten (vice versa zur objektorientierten Bedienung) Bedienung folgen. Oftmals werden in der Praxis Kombinationen aus diesen Bedienungsarten verwendet. [[9]](#footnote-9)

In der vorliegenden Applikation wird beispielsweise eine objektorientierte Bedienung im MainView- bzw. MainPresenter bei der Auswahl der Liga bzw. des Liga- Models (über die vorhandenen Liga 1 bis Liga 3 Buttons) gewählt. Eine funktionsorientierte Beziehung ist bei dem Dialog zum Hinzufügen von Spielen vorzufinden. Hierbei wird zuerst die auszuführende Funktion ausgewählt (Spiel hinzufügen), welches wiederum das öffnen einen neuen Sekundär-Dialoges veranlasst. In diesem Sekundär- Dialog werden anschließend die Objekte, in diesem Fall die zwei Clubs, die gegeneinander spielen, ausgewählt.

* 1. Fenster

Das Fenster ist das zentrale Element der Dialoggestaltung. Unter Windows werden folgende Fenstertypen differenziert:[[10]](#footnote-10)

* Sekundärfenster (Durchführung Sekundäraktivitäten (Sekundärdialoge), Optionseingabe)
* Primärfenster (Durchführung Hauptaktivitäten (Primärdialoge), Anwendungsfenster)

Das Anwendungsfenster, aus welchem sich heraus bei Bedarf weitere Fenster öffnen lassen können, ist das wichtigste Primärfenster. Der Arbeitsbereich, der Menübalken sowie der Titelbalken sind mindestens im Primärfenster enthalten. Bei einer Schließung des Primärfensters erfolgt außerdem das Beenden aller offenen Sekundärfenster.[[11]](#footnote-11)

In der vorliegenden Applikation besteht das Primärfenster der Applikation (MainView realisiert durch ein JFrame) aus einem Titelbalken (Titel: Bundesligaverwaltung), einem zwei reihigem Menübalken, bestehend aus den Auswahlknöpfen für Liga 1, Liga 2, Liga 3 sowie den Auswahlknöpfen für Tabelle, Spielplan und Clubs und dem Arbeitsbereich, der je nach der getroffenen Auswahl die Darstellung bzw. die anzuzeigenden Daten im Arbeitsbereich anpasst.

Dialogfenster (dialog box, Benutzereingabe meist als modaler Dialog) und Mitteilungsfenster (message box, nur Benutzermittelungen keine Interaktionselemente zur Datenselektion- oder Manipulation) sind u.a. verwendbare Arten von Sekundärfenster in Windows.[[12]](#footnote-12)

In der vorliegenden Applikation wurden Sekundärfenster als Dialogfenster z.B. beim hinzufügen eines neuen Spiels zwischen zwei Teams verwendet. Dies wurde durch mittels der Implementierung eines JDialog realisiert. Nach vollständiger Auswahl und Eingabe aller erforderlichen Parameter werden diese überprüft. Je nach positiver oder negativer Überprüfung der Eingabedaten wird ein entsprechender MessageDialog (mittels JOptionPane) mit entsprechenden Hinweisen angezeigt.

* 1. Menüs

Eine übersichtliche und zumeist vorbestimmte Auswahl von Menüoptionen sind häufig Bestandteile von Menüs. Menüs können als Aktionsmenü (Verzweigung in andere Menüs oder Auslösen von Anwendungsfunktionen) oder als Eigenschaftsmenü (Einstellung von Parametern zum Beeinflussung des Verhaltens der Anwendung, auch mehr Selektion möglich) in Applikationen vorkommen.[[13]](#footnote-13)

Es gibt u.a. folgende Menüarten: [[14]](#footnote-14)

* Menübalken (besitzt alle Menütitel, ist immer sichtbar, Anwendungsfenster hat immer einen Menübalken, Menübalken sind nicht in Dialog- oder Mitteilungsfenster vorhanden, Menübalken in Unterfenster möglich)
* Pop-up Menüs (wird an aktueller Mauszeigerposition erzeugt, Bezugspunkt ist Objekt, auf das es aktiviert wird, nur sichtbar, wenn geöffnet, Menüoptionen gelten nur für selektiertes Objekt)

In der vorliegenden Applikation wurde auf klassische Menüleiste, wie in den meisten Windows- Applikationen vorzufinden ist verzichtet. Stattdessen wurden beispielsweise im Hauptfenster (MainView) zwei Menüreihen zur Auswahl der anzuzeigenden Daten (über Liga 1 bis Liga 3 selektierbar) und der passenden Ansicht (Tabelle, Spielplan, Clubs) zu den Daten genutzt. Dies ermöglicht sowohl eine große Flexibilität in der Auswahl als auch eine einfache Bedienung. Pop-up Menüs wurden in der Applikation beispielsweise beim Bearbeiten oder Löschen Clubs oder Spielern verwendet.

* 1. Spezielle Designelemente in der vorliegenden Applikation

In jedem GUI- System stehen verschiedene Elemente zur Interaktion mit bzw. Steuerung der Applikation bereit. Dabei können diese Elemente bzgl. ihres Aussehens und ggf. bzgl. ihrer Funktionsweise vom jeweiligen Betriebssystem abhängig sein, auf welchem die Applikation ausgeführt wird.[[15]](#footnote-15)

Im Folgenden wird auf eine Auswahl eingesetzter grafischer Interaktionselemente genauer eingegangen, die in der vorliegenden Applikation (laufend unter dem Betriebssystem Microsoft Windows) verwendet wurden.

**Tabellen bzw. Listenelemente** stellen Erweiterungen eines Listenfeldes (list box) dar. Für die Darstellung der Daten in der Tabellenform existieren verschiedene Darstellungsvarianten:[[16]](#footnote-16)

* beliebige Positionierung von (Mini)-Piktogramm mit Beschriftung im Listenelemente
* jeder Eintrag in Liste bestehend aus Beschriftung und Minipiktogramm spaltenweise sortiert
* Report (pro Zeile ein Eintrag, jeder Eintrag bestehend aus mehreren Spalten, Spaltenbreite anpassbar)

In der vorliegenden Applikation verwendeten Tabellen wurden mit der Java eigenen JTable realisiert. Diese stellt die Daten in der Form des oben beschriebenen Reportes dar, wofür sie ein eigenes Model besitzt. Dieses Model wird in der View- Klasse initialisiert und im dazugehörigem Presenter mit Daten befüllt. Die verwendeten JTables sind nicht an sich editierbar. Dies wurde aus Gründen der Datenkonsistenz so implementiert. Eine Bearbeitung von Tabelleninhalten kann nur bei speziell ausgewählten Tabellen (Club oder Spieler-Tabelle) über eigene Menüs erfolgen.

Eine platzsparende Variante der Liste ist das **Dropdown- Listenfeld** bzw. die Klappliste, bei welcher ein Aufklappen der Liste vor der Selektion eines Listenelementes notwendig ist. Nach der Selektion ist die Liste wieder unsichtbar und das selektierte Element wird permanent angezeigt. Dabei ist eine Überdeckung anderer Interaktionselemente durch die aufgeklappte Liste möglich. Es kann eine Vorauswahl in dem Drop- Down Listenfeld getroffen werden.15

In der vorliegenden Applikation wurden Dropdown- Listenfelder in Form von JComboBoxen beispielsweise beim Ersteller neuer Spiele im PlanAddGameView verwendet. Hierbei ist sowohl die auswählbare Liga als auch das jeweils auswählbare Team über JComboBoxen realisiert. Die jeweils selektierbaren Teams richten sich dabei nach der ausgewählten Liga. Diese Interaktionselemente wurde an dieser Stelle bewusst gewählt, um den Benutzer, durch vorgegebene Auswahlmöglichkeiten in den JComboBoxen eine einfache Bedienung der Applikation zu ermöglich und zu gleich die Wahrscheinlichkeit für fehlerhafte Benutzereingaben, die z.B. durch einen direkten Input durch Textfelder entstehen könnten, zu reduzieren.

# Schluss

//Robin

//bei uns erklären: wir nutzen für uns eine Kombination aus MVC und MVP wir nutzen die strikte Trennung von Model und View aus dem MVP Pattern, jedoch die Verarbeitung von Benutzereingaben, die durch den Presenter geschenen aus dem MVC Pattern  für die Datenkonsitenz nutzen wir das Observerpattern (MainPresenter registriert sich beim LigaModel  ist das Hauptmodel an Liga ist alles dran)  Quelle die die Anwendung des Observer Patterns im MVC Bereich beschreibt

 Ziel: keine Beziehung zwischen Model und View & App- Weite Datenkonsitenz über mehrere View- Presenter- Beziehung durch ein einheitliches Hauptmodels (durch Aktualisierung des MainPresenters bei Änderung des Liga Models)

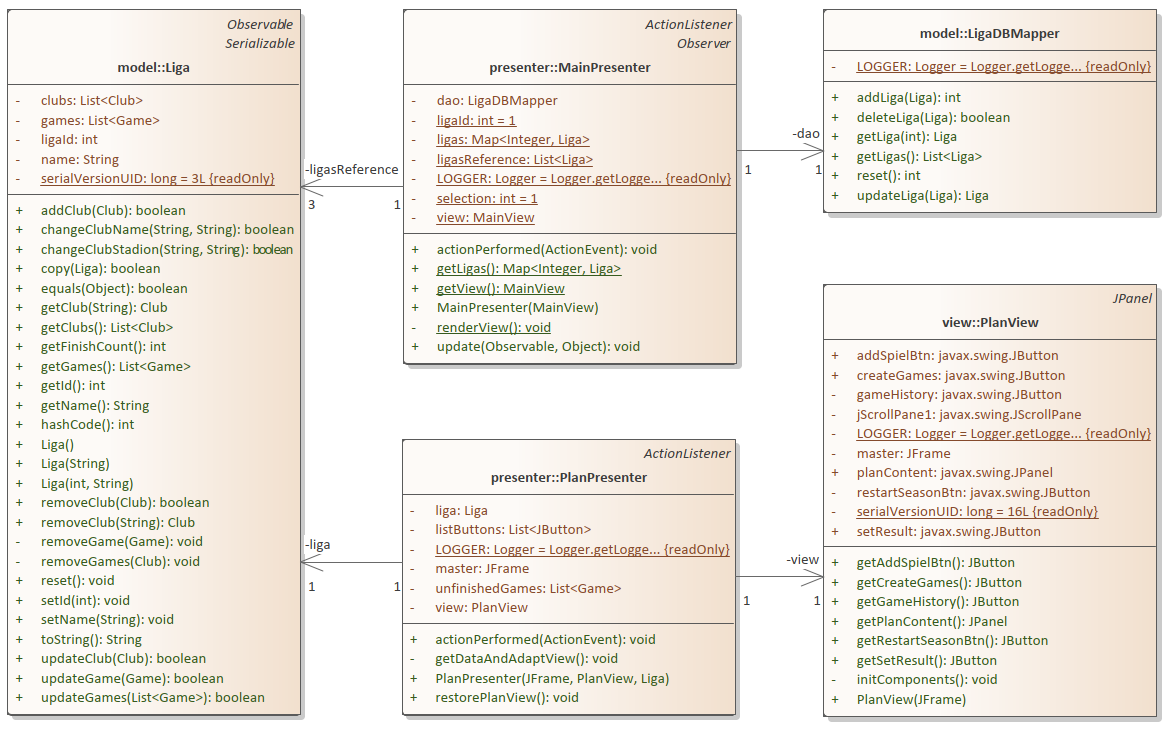
 warum geben wir wie was mit  manche Fenster Dialoge  brauchen ein Vater Fenster  Hauptinstanz die Alles Kontrolliert: MainPresenter  hier allgemeine Anzeige im Hauptfenter (und damit auch weitere Intialisierung von View- Presenter Beziehungen)  & Aktualisierung der Daten über das Observer pattern  app Funktioniert und weitere Fenster bleiben offen und nur Datenaktualisierung (Bsp.: Hinzufügen von Spielern (im ClubView), weil wir bei Aktualisierung im MainPresenter nicht dessen Zustand ändern d.h. alle offenen Presnter- View Beziehungen, die durch den MainPresenter initlaiisert wurden bleiben so bestehen  das gleiche gilt auch für alle nachfolgenden Presenter- View Beziehungen  sinnvoll wegen der Komplexittät der Applikation (models können von mehreren Controllern aktualisiert werden, und von mehreren Views dargestellt werden)

 bei uns: wir arbeiten mit einem einheitlichen Datenmodell (der ausgwählten Liga)  deswegen auch nur eine einheitliche, zentrale Update Schnittstelle

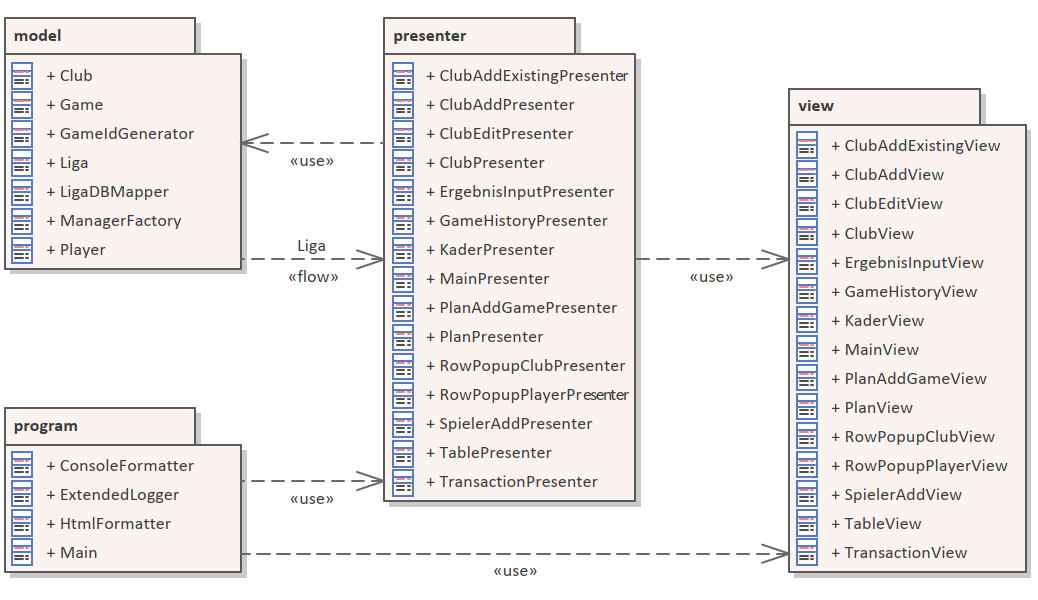
 siehe S. 449 Subsystem der Benutzeroberfläche

Anhang

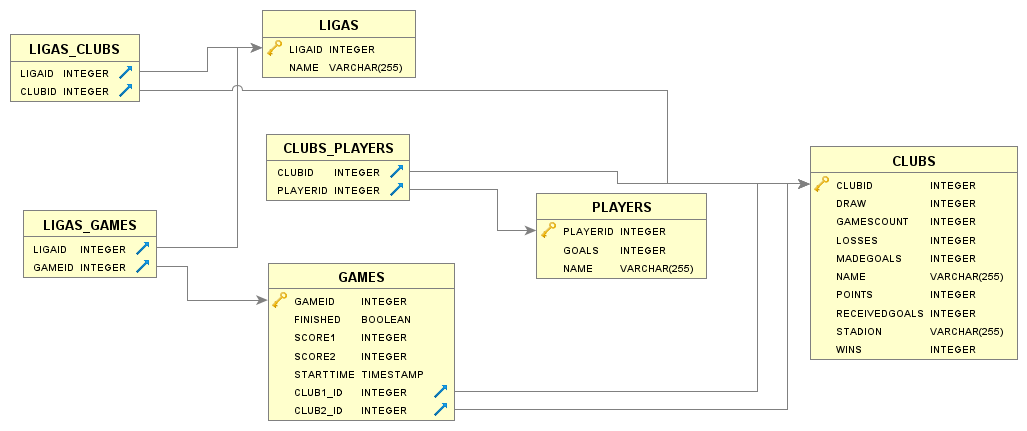
Anhang 1 Klassendiagramm (Eigene Darstellung)



Anhang 2 Paketdiagramm (Eigene Darstellung)



Anhang 3 Datenbank-Diagramm (Eigene Darstellung)



Literaturverzeichnis

1 Balzert, Heide (Lehrbuch der Objektmodellierung; 1999): Lehrbuch der Objektmodellierung: Analyse und Entwurf, Spektrum Akademischer Verlag, Heidelberg, Berlin, 1999

2 Balzert, Helmut (Lehrbuch der Softwaretechnik; 2011): Lehrbuch der Softwaretechnik, Spektrum Akademischer Verlag, Heidelberg, 2011

3 Gharbi, Mahbouda; Koschel, Arne; Rausch, Andreas; Starke, Gernot (Basiswissen für Softwarearchitekten; 2018): Basiswissen für Softwarearchitekten, dpunkt Verlag, Heidelberg, 2018

Literaturverzeichnis

1 Balzert, Heide (Lehrbuch der Objektmodellierung; 1999): Lehrbuch der Objektmodellierung: Analyse und Entwurf, Spektrum Akademischer Verlag, Heidelberg, Berlin, 1999

2 Balzert, Helmut (Lehrbuch der Softwaretechnik; 2011): Lehrbuch der Softwaretechnik, Spektrum Akademischer Verlag, Heidelberg, 2011

3 Gharbi, Mahbouda; Koschel, Arne; Rausch, Andreas; Starke, Gernot (Basiswissen für Softwarearchitekten; 2018): Basiswissen für Softwarearchitekten, dpunkt Verlag, Heidelberg, 2018

1. Vgl. Balzert, H.; Lehrbuch der Softwaretechnik; 2011; S.54-56 [↑](#footnote-ref-1)
2. Vgl. Balzert, H.; Lehrbuch der Softwaretechnik; 2011; S. 68 [↑](#footnote-ref-2)
3. Vgl. Balzert, H.; Lehrbuch der Softwaretechnik; 2011; S. 62f. [↑](#footnote-ref-3)
4. Vgl. Balzert, H.; Lehrbuch der Softwaretechnik; 2011; S. 67f. [↑](#footnote-ref-4)
5. Vgl. Gharbi, M./ Koschel, A./ Rausch, A., et al.; Basiswissen für Softwarearchitekten; 2018; S. 79 [↑](#footnote-ref-5)
6. Vgl. Balzert, H.; Lehrbuch der Softwaretechnik; 2011; S. 458 [↑](#footnote-ref-6)
7. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 194 [↑](#footnote-ref-7)
8. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 195 [↑](#footnote-ref-8)
9. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 197-199 [↑](#footnote-ref-9)
10. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 199 [↑](#footnote-ref-10)
11. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 200 [↑](#footnote-ref-11)
12. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 201 [↑](#footnote-ref-12)
13. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S.202f. [↑](#footnote-ref-13)
14. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 203 [↑](#footnote-ref-14)
15. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 215 [↑](#footnote-ref-15)
16. Vgl. Balzert, H.; Lehrbuch der Objektmodellierung; 1999; S. 218 [↑](#footnote-ref-16)