WYŻSZA SZKOŁA INFORMATYKI I UMIEJĘTNOŚCI
Wydział Informatyki i Zarządzania
Kierunek: Informatyka

Piotr Paziak
nr albumu: 30067

Praca Inżynierska
Symulator Rzeczywistych Rozgrywek Sportowych

Praca napisana pod kierunkiem
dr Dariusza Doliwy

Rok akademicki 2019/2020

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie oraz nie korzystałem ze źródeł innych niż wymienione w pracy.

…….

(czytelny podpis)

Spis treści

1. Wstęp

Już od najdawniejszych czasów ludzie interesowali się różnego rodzaju zmaganiami sportowymi. Organizowano rozmaite zawody sportowe, igrzyska, turnieje a na początku XX wieku powstały cyklicznie rozgrywki ligowe. Do najpopularniejszych rozgrywek ligowych można zaliczyć: piłkę nożną, koszykówkę, hokej, baseball czy piłkę siatkową. Kibice obok ekscytacji związanej z oglądaniem spotkań sportowych zaczęli interesować się również statystykami prowadzonymi podczas rozgrywanych spotkań. Podczas meczu odnotowywano liczbę zdobytych punktów, bramek, asyst przez danego zawodnika i jego drużynę. Zaczęto wyliczać różnego rodzaju średnie, które pozwalały wybrać najlepszego zawodnika ligi, stworzenie najlepszej drużyny sezonu czy ustalenie, która drużyna ma najlepsza obronę lub atak. Zaczęły też powstawać rożnego rodzaju gry planszowe oraz karciane, które poprzez opis parametrów danego zawodnika i drużyny umożliwiały rozegranie kilku osobom rozgrywki zbliżonej do tej rzeczywistej.

Pod koniec XX wieku rozwój komputerów i oprogramowania pozwolił na tworzenie aplikacji, które miały na celu symulować dane rozgrywki. Tworzono rozmaite gry symulujące rozgrywki piłkarskie lub inne sporty nie tylko drużynowe. Sam jestem miłośnikiem komputerowych symulatorów rozgrywek piłkarskich tak zwanych futbol menadżerów. Gram w nie od wczesnych lat dziewięćdziesiątych po dzień dzisiejszy. Zawsze zastanawiałem się jak działają i jak są skonstruowane. Jednocześnie zawsze interesowałem się piłką nożną. Oglądałem wszystkie turnieje Mistrzostw Świata i Europy w tej dyscyplinie, śledziłem różne ligi europejskie i puchary. Tak narodziła się pasja do programowania, która w połączeniu z pasją do piłki nożnej była głównym czynnikiem, który zmotywował mnie do stworzenia symulatora rzeczywistych rozgrywek ligowych.

Wyzwanie jakim jest stworzenie takiego symulatora pozwoliła mi połączyć pasję do piłki nożnej z nauką programowania, co było kolejnym krokiem na drodze osobistego rozwoju oraz pomogło mi na uzyskanie awansu zawodowego. Jednocześnie umożliwiło mi to pogłębienie wiedzy w zakresie programowania obiektowego i zastosowania tej wiedzy w praktyce.

2. Cel Pracy

Celem niniejszej pracy jest zaprojektowanie i stworzenie aplikacji, która umożliwiłaby symulowanie rozgrywek ligowych w tym przypadku rozgrywek piłki nożnej. Najważniejszym aspektem projektu a zarazem najtrudniejszym było odwzorowanie realistycznych zachowań drużyn i zawodników w odniesieniu do rzeczywistości.

Na rynku istnieje kilka grup symulatorów (managerów). Pierwsza z nich realistycznie odwzorowuje rozgrywki ligowe różnych krajów oraz pucharów krajowych i zagranicznych. Druga grupa to symulatory typu fantazy w których uczestnicy mogą mieć te same drużyny z powtarzającymi się zawodnikami w ich składach.

W rzeczywistości istnieją rozgrywki piłkarskie pod nazwą „Champions League" skupiające najlepsze drużyny z Europy. Jednakże nie jest to liga w dosłownym słowa tego znaczeniu. System rozgrywek jest bardzo podobny do tego jaki jest stosowany w „Mistrzostwach Świata" w piłce nożnej drużyn narodowych. Polega on na tym, że wszystkie drużyny rozlosowywane są do grup składających się z czterech zespołów, następnie rozgrywają mecze każdy z każdym tak aby ostatecznie wyłonić dwie najlepsze drużyny z grupy. W kolejnym etapie rozgrywek wyłonione drużyny spotykają się w systemie pucharowym tak aby drogą eliminacji dwie najlepsze spotkały się w finale. System „Champions League" rożni się od systemu rozgrywek w „Mistrzostwach Świata" tylko tym, że drużyny grają mecz i rewanż zamiast pojedynczego meczu.

Rezultatem tych rozgrywek jest 6 meczów w fazie grupowej i 7 meczów w fazie pucharowej co daje łącznie 13 meczów pod warunkiem dotarcia drużyny do finału. W efekcie prowadzi to do tego, że tylko dwie drużyny rozgrywają po 13 spotkań, połowa po 6 spotkań (przegrani w fazie grupowej) a pozostałe (drużyny które odpadły w fazie pucharowej) od 8 do 11 meczów. Rodzi to pewne wątpliwości czy taki system rozgrywek wyłania naprawdę najlepszą drużynę.

Dlatego też celem tego projektu jest utworzenie ligi składających się z 16 najlepszych drużyn europejskich tak aby umożliwić spotkanie każdy z każdym w systemie mecz rewanż. Umożliwi to rozegranie każdej drużynie 30 spotkań i wyłonienie najlepszej z nich w możliwie najsprawiedliwszy sposób. Realizacja takiego symulatora pozwoli sprawdzić jak będą kształtowały się rozgrywki kiedy w jednej lidze zostaną zgromadzone najlepsze drużyny i najlepsi zawodnicy.

Z powodu wyżej wymienionych czynników projekt został napisany w technologii wykorzystującej obiektowość, która najbardziej nadaje się do takich rozwiązań. Ważnym celem jest także zaprojektowanie przyjaznego graficznego interfejsu użytkownika (GUI), który w łatwy i intuicyjny sposób umożliwi obsługę symulatora osobie z niego korzystającej.

3. Wybór technologii

3.1. Przegląd technologii

W tym rozdziale skupię się na przeglądzie dostępnych technologii na rynku, które umożliwiają wytwarzanie oprogramowania. Przedstawię podział języków ze względu na kryterium przydatności oraz uargumentuję dlaczego jedna grupa jest lepsza od innej do wykonania tego projektu. Dodatkowo postaram się opisać biblioteki graficzne dedykowane do poszczególnych języków, które umożliwiają w krótkim czasie wytwarzanie graficznego interfejsu użytkownika (GUI).

Języki programowania możemy podzielić na kilka grup. Najpopularniejsze to:

- funkcyjne
- skryptowe
- proceduralne
- obiektowe

3.1.1. Języki funkcyjne

Przedstawicielami języków funkcyjnych jest Haskell, Lisp, Erlang czy F#. Programy napisane w językach funkcyjnych składają się jedynie z funkcji. Główny program jest funkcją, która przyjmuje argumenty i zwraca obliczoną wartość. Może składać się on z innych funkcji, które mogą wywoływać jeszcze inne funkcje. Funkcja może przyjmować jako parametr funkcję, jak również zwracać inna funkcję. W programowaniu funkcyjnym zamiast zmiennych występują stałe a zamiast pętli stosowana jest rekurencją. Najważniejszą cechą tych języków jest to, iż nie pozwalają na żadne efekty uboczne. W czystym programowaniu funkcyjnym, raz zdefiniowana funkcja dla tych samych danych wejściowych zawsze zwróci ten sam wynik. Języki funkcyjne zostały zaprojektowane z myślą o zastosowaniach współbieżnych (Erlang) oraz pod kątem tworzenia systemów rozproszonych wymagających długotrwałej pracy i odporności na awarie. Jak również często stosowane są do tworzenia sztucznej inteligencji (Lisp).

3.1.2. Języki skryptowe

Drugą grupą są języki skryptowe. Jak sama nazwa wskazuje są to języki, które obsługują skrypty. Najpopularniejsze to JavaScript, Pyton, PHP, Ruby on Rails, Perl czy Node.JS. Grupa ta odróżnia się od pozostałych tym, że kod napisany w tych językach może być tworzony nawet w notatniku i nie musi być kompilowany do pliku wykonywalnego - zrozumiałego dla komputera. Proces kompilacji w językach skryptowych odbywa się za każdym razem w momencie uruchomienia programu. Zaletą takiego programowania jest szybsze pisanie aplikacji a programy tworzone w

tych technologiach mają zwykle małą objętość. Języki skryptowe często stosuje się do automatyzowania różnego rodzaju zadań najczęściej administracyjnych. Skrypty wykorzystywane są również do sterownia przebiegiem gry, kontrolują wtedy fabułę, dialogi czy sterują zachowaniem wirtualnych postaci. Języki takie jak PHP czy JavaScript wykorzystywane są w technologiach internetowych (HTML) do dynamicznej interakcji z użytkownikiem poprzez reagowanie na zdarzenia, zapewnieniu walidacji danych z formularzy czy dynamicznym manipulowaniu obiektami na stronach internetowych.

3.1.3. Języki proceduralne

Kolejną grupą, której przedstawicielem jest język C, są języki proceduralne. Głównym założeniem tych technologii jest to, że program główny składa się z wielu procedur (podprogramów) lub funkcji realizujących określone zadania, które mogą być wywoływane wile razy. Zadaniem procedur jest modyfikacja zmiennych globalnych albo też modyfikacja wartości lub struktury przekazanej w parametrze, natomiast funkcja oprócz modyfikacji parametrów może je również zwracać. Możliwe jest również zagnieżdżanie odwołań do procedur dzięki czemu można korzystać z rekurencji. C jest językiem strukturalnym i nie wspiera programowania obiektowego, jednakże programowanie obiektowe jest w nim możliwe. Dodatkowo język C jest wśród języków wysokopoziomowych „najbliżej" maszyny, co czyni go bardzo wygodnym narzędziem do tworzenia oprogramowania dla systemów, czy mikrokontrolerów. Język C jest bardzo wydajny i lekki przez co chętnie jest wykorzystywany do programowania systemów wbudowanych stosowanych w urządzeniach domowych takich jak: pralki, lodówki poprzez zegarki, kończąc na elementach samochodów takich jak: wyświetlacze, czujniki czy systemy zapewniające bezpieczeństwo. Język C został również wykorzystany do napisania jąder takich systemów jak Linux, Windows, Android czy biblioteki graficznej OpenGL.

3.1.4. Języki obiektowe

Ostatnią grupą stanowią języki obiektowe. Do najpopularniejszych technologii z tej grupy należą: C++, Java, C#. Najważniejszym pojęciem związanym z programowaniem obiektowym jest klasa, która stanowi szablon na podstawie, którego tworzone są obiekty (instancje klas). Można powiedzieć, że klasa jest to typ złożony zdefiniowany przez użytkownika. Dodatkowo klasa w odróżnieniu od struktury posiada metody(funkcje) umożliwiające wpływanie na zachowanie obiektu. Podstawowymi założeniami paradygmatu obiektowego są cztery cechy:

- Abstrakcja - polega na ukrywaniu lub pomijaniu mało istotnych informacji a skupieniu się na wydobyciu informacji, które są niezmienne i wspólne dla pewnej grupy obiektów.

- Dziedziczenie - Podstawowym zastosowaniem dziedziczenia jest ponowne wykorzystanie kodu. Jeśli dwie klasy wykonują podobne zadania, możemy utworzyć dla nich wspólną klasę bazową, do której przeniesiemy identyczne metody oraz atrybuty.
- Enkapsulacja – czyli hermetyzacja polega na ukrywaniu implementacji obiektu przed użytkownikiem i zapewnia , że obiekt nie może zmieniać w nieoczekiwany sposób stanu wewnętrznego innych obiektów.
- Polimorfizm – uwolnienie się od typów czyli traktowanie różnych danych w jednolity sposób. Rozróżniamy polimorfizm statyczny (czasu kompilacji) i polimorfizm dynamiczny (czasu wykonania)

Zastosowanie języków obiektowych jest bardzo szerokie. Wykorzystuje się je do pisania gier, aplikacji desktopowych, internetowych jak również mobilnych czy nawet systemów wbudowanych (C++).

Odnosząc się do powyższej charakterystyki technologii językowych stwierdzam, że najodpowiedniejszą grupą do stworzenia symulatora rzeczywistych rozgrywek ligowych w oparciu o graficzny interfejs użytkownika jest grupa języków obiektowych. Języki funkcyjne jak wcześniej napisałem mają główne zastosowanie w systemach rozproszonych czy projektowaniu sztucznej inteligencji. Języki skryptowe są w ostatnich czasach bardzo często wykorzystywane przy wytwarzaniu aplikacji internetowych. Języki proceduralne nadają się do programowania tego typu aplikacji, jednakże przewagą języków obiektowych nad językami proceduralnymi jest fakt, że taki sposób programowania jest bardziej zgodny rzeczywistością a ludzki mózg jest lepiej przystosowany do takiego podejścia podczas przetwarzania informacji. Podstawowe założenia paradygmatu obiektowego takie jak abstrakcja czy enkapsulacja przyczyniają się do zwiększenia czytelności natomiast dziedziczenie i polimorfizm są zwykle używane w celu redukcji zbędnego kodu. Dodatkowo w ostatnich latach języki obiektowe zostały wyposażone w potężne darmowe biblioteki graficzne , które ułatwiają i przyśpieszają znacząco tworzenie kodu.

3.2. Przegląd Języków Obiektowych

3.2.1. C++ i Qt

Język C++ to potężne narzędzie łączące w sobie trzy różne kategorie programowania: programowanie proceduralne charakterystyczne dla języka C, obiektowe wyrażone przez dodanie klas oraz programowanie uogólnione znajdujące wyraz w szablonach języka C++. Charakteryzuje się wysoką wydajnością kodu wynikowego, bezpośrednim dostępem do zasobów sprzętowych i funkcji systemowych, łatwością tworzenia i korzystania z bibliotek (napisanych w C++, C lub innych językach), niezależnością od konkretnej platformy sprzętowej lub systemowej (co gwarantuje wysoką przenośność kodów źródłowych) oraz niewielkim środowiskiem uruchomieniowym.

C++ został zaprojektowany przez Bjarne Stroustrupa jako rozszerzenie języka C o obiektowe mechanizmy abstrakcji danych i silną statyczną kontrolę typów. Zachowanie zgodności z językiem C na poziomie kodu źródłowego pozostaje jednym z podstawowych celów projektowych kolejnych standardów języka. W latach 90. XX wieku język C++ zdobył pozycję jednego z najpopularniejszych języków programowania ogólnego przeznaczenia.

Qt jest to wieloplatformowy framework służący do tworzenia aplikacji desktopowych, wbudowanych (embedded) i mobilnych, które mogą być uruchamiane na wszystkich podstawowych systemach (Windows, Linux, iOs, Android, BlackBerry i inne). Qt używa standardów C++ z rozszerzeniami takimi jak sygnały i sloty. Posiada również własny zestaw szablonów kontenerów (QVector, QList, QMap i inne). Qt wyposażony jest w Qt Creator narzędzie do szybkiego tworzenia graficznego interfejsu użytkownika (GUI). Ponadto Qt posiada bardzo dobrze napisaną dokumentację oraz liczne przykłady i tutoriale ułatwiające pracę z tym środowiskiem. Biblioteki Qt, oprócz obsługi interfejsu użytkownika, zawierają także niezależne od platformy systemowej moduły obsługi procesów, plików, sieci, grafiki trójwymiarowej (OpenGL), baz danych (SQL), języka XML, lokalizacji, wielowątkowości, zaawansowanej obsługi napisów oraz wtyczek.

Bardzo łatwy sposób tworzenia GUI przy pomocy kreatora, rozbudowana dokumentacja z licznymi przykładami, jak również bardzo pomocny edytor powodują, że jest to najbardziej intuicyjne środowisko programistyczne z jakim do tej pory się spotkałem.

3.2.2. Java i Swing

Java jest językiem programowania który umożliwia wytwarzanie aplikacji na wiele platform. Każdy program napisany przez programistę kompilowany jest do kodu bajtowego i dzięki wirtualnej maszynie może działać na wielu systemach operacyjnych takich jak Windows, Linux i Mac OS. Java odziedziczyła wiele cech po C i C++ z których zapożyczono dużą cześć składni i słów kluczowych, natomiast zarządzanie pamięcią zajmuje się maszyna wirtualna (JVM) z języka Smalltalk. Java została stworzona przez Jamesa Goslinga z firmy Sun Microsystems.

Podobnie jak w C++ Java wykorzystuje cztery główne paradygmaty programowania obiektowego: abstrakcję, dziedziczenie, enkapsulację i polimorfizm. W odróżnieniu od języka C++ Java nie udostępnia wielodziedziczenia ale w zamian za to wprowadza interfejsy, których zadaniem jest wyeliminowanie konfliktów podczas przekazywania właściwości przez klasy nadrzędne (śmiertelny rąb). W Javie wszystkie obiekty oprócz typów prostych (int, float) dziedziczą po klasie nadrzędnej Object. Implementacja jej podstawowych zachowań i właściwości umożliwia ich porównywanie, identyfikację, kopiowanie oraz niszczenie. Kolejną różnicą miedzy tymi językami jest zarządzanie pamięcią (Memory Management). W C++ całość spoczywa w rękach programisty poprzez tworzenie i niszczenie wskaźników natomiast w Javie kontroluje to system przy pomocy narzędzia zwanego Garbage Collection. Java w przeciwieństwie do C++ który stosunkowo umożliwia

programowanie niskopoziomowe dostarcza szeroki zakres klas dla różnych usług wysokiego poziomu.

Swing jest biblioteką graficzną służącą do wytwarzania aplikacji okienkowych w języku Java. Biblioteka powstała w 1997 roku i jest nową ulepszoną wersją biblioteki AWT. Najistotniejszą różnicą pomiędzy tymi bibliotekami jest to że AWT pobiera wszystkie kontrolki z systemu co miało być zgodne z główną zasadą - niezależnością od platformy.. Natomiast Swing rysuje wszystkie komponenty od początku, dzięki czemu aplikacje wygląda identycznie na wszystkich platformach, na których jest uruchamiana. Minusem tego rozwiązania jest zmniejszenie wydajność aplikacji, jednakże przy obecnym rozwoju technologii (coraz szybsze komputery) jest praktycznie niewyczuwalne.

Budowa aplikacji z użyciem obu tych bibliotek polega na tworzeniu graficznego interfejsu użytkownika (GUI) z mniejszych składników takich jak przyciski, pola tekstowe, rozwijane listy czy etykiety. Całość umieszczana jest w panelach (JPanel), które umieszczane są w obiekcie nadrzędnym – ramce (JFrame). Do interakcji z użytkownikiem wykorzystywane są zdarzenia, które implementując interfejs ActionListener pozwalają na obsługę wciśniętego przycisku, odczytu położenia kursora na ekranie oraz wielu innych zdarzeń.

3.2.3. C# i WPF

C# to odpowiedź Microsoftu na Javę. Jest to obiektowy język programowania ogólnego przeznaczenia z bezpiecznymi typami. Język został zaprojektowany w latach 1998-2001 a jego głównym architektem trzymającym pieczę nad językiem jest Anders Hejlsberg(twórca Turbo Pascala). Projektanci tego języka starali zachować balans między prostotą, ekspresyjnością i wydajnością. Nazwa języka (podobnie jak nazwa C++ który poprzez wykorzystanie operatora inkrementacji ++ oznaczającego zwiększenie o jeden w odniesieniu do języka C) zawierająca symbol # przypominająca dwa operatory inkrementacji sugeruje, że C# jest następcą C++.

Podobnie jak Java wykorzystuję dużą część składni i słów kluczowych z C++. Jeżeli chodzi o obiektowość i główne paradygmaty programowania to jest to bardzo zbliżone do Javy. (zamiast dziedziczenia wielokrotnego wprowadzono interfejsy). Dodatkowo C# wprowadza nowe elementy składowe klas takie jak: właściwości i indeksery. delegaty i zdarzenia to odpowiedniki i rozwinięcie wskaźników na funkcje z C++. Hierarchia dziedziczenia opiera się na istnieniu jednej klasy Object po której dziedziczą wszystkie inne klasy. W odróżnieniu od Javy również typy proste takie jak (int, double) są strukturami posiadające odpowiednie dla siebie metody takie jak: ToString, Equals czy GetType. W języku C# pamięcią automatycznie zarządza system wykonawczy. Środowisko uruchomieniowe CLR(Common Language Runtime) wyposażone jest w moduł usuwania nieużywanych obiektów(Garbage Collector) co zwalnia programistę (tak jak w Javie a w przeciwieństwie do C++) od pamiętania o własnoręcznej dezalokacji pamięci. Jednakże w celach optymalizacji wydajności

kodu możliwe jest stosowanie wskaźników w specjalnych blokach oznaczonych jako niebezpieczne(unsafe).

W przeszłości język C# przeznaczony był w zasadzie do tworzenia aplikacji przeznaczonych na systemy Windows. Jednakże w ostatnich czasach poprzez rozwój takich narzędzi jak Xmarin czy ASP.NET możliwe jest tworzenie wieloplatformowych aplikacji mobilnych jak również aplikacji internetowych. C# znajduje również zastosowanie w popularnym silniku Unity, który jest wykorzystywany do tworzenia gier na PC, konsole i urządzenia mobilne.

Windows Presentation Foundation (WPF) znany wcześniej jako "Avalon" to graficzny podsystem (podobny do WinForms) stworzony do generowania interfejsu użytkownika stworzony jako część .NET Framework #.0 w 2006 roku. WPF do renderownia elementów pulpitu w grafice wektorowej takich jak okna wykorzystuje bibliotekę Direc3D. Pozwala to na wyświetlanie bardziej złożonych elementów grafiki z wykorzystaniem procesorów graficznych (GPU) jednocześnie zmniejszając obciążenie procesora komputera (CPU). Rezultatem tego jest szybsze odświeżanie ekranu co przekłada się znacząco na poprawę wydajności działania aplikacji graficznych oraz animacji.

Aplikacje tworzone przy pomocy WPF wykorzystują wzorzec model-view-view-model (MVVM) w celu oddzielenia logiki biznesowej (tworzonej w c#) od widoku i składają się z trzech kluczowych elementów:

- warstwa widoku (plik z kodem XAML przypominającym strukturą HTML)
- warstwa kontrolera (code-behind – odpowiedzialnych za obsługę widoków)
- warstwa modelu (odpowiedzialna za realizację logiki biznesowej i przechowywanie danych)

Dzięki takiemu podejściu możliwe jest tworzenie aplikacji desktopowych przez projektantów nie będących ekspertami od C#. Tworzenie aplikacji w WPF polega podobnie jak w Swingu i Qt na budowaniu aplikacji z „cegiełek” tak zwanych kontrolek (Button, Label, CheckBox) i umieszczaniu ich w oknie (Window). Dodatkowo do dyspozycji mamy pojemniki służące do grupowania kontrolek (Border, Grid, StackPanel, DockPanel). Następnym udogodnieniem jest wprowadzenie styli, które pozwalają na definiowanie kolorów, czcionek, wyrównania i innych graficznych atrybutów dla wszystkich kontrolek co pozwala zaoszczędzić czas kiedy zaistnieje potrzeba zmiany wyglądu aplikacji.

3.2.4. Podsumowanie

Podsumowując wyżej wymienione technologie można dojść do wniosku, że każda z nich doskonale nadaję się do stworzenia symulatora rzeczywistych rozgrywek ligowych. Różnice pomiędzy samą składnią języków nie są duże (zwłaszcza między

C# a Javą) jak również wytwarzanie samych aplikacji graficznych podlega podobnym zasadom.

W pracy zawodowej jako programista do wytwarzania aplikacji używam języka C# i technologii .NET, która doskonale sprawdza się w rozwiązaniach biznesowych. Technologie związane z Javą używałem sporadycznie w większości przypadków jako projekty uczelniane. Język C++ to mój pierwszy język obiektowy. Jest językiem najtrudniejszym z wszystkich trzech a jego główną wadą albo zaletą jest brak automatycznego zarządzania pamięcią. Sprawia to wiele trudności na początku, jednakże opanowanie tej umiejętność pozwala na wytwarzanie aplikacji dużo wydajniejszych niż w Javie czy C#. Dodatkowo framework w postaci Qt z intuicyjnym QtCreatorem i znakomitą dokumentacją sprawił że został on wybrany do stworzenia tego projektu.

4. Projekt Systemu

W tym rozdziale przedstawię projekt systemu jego mechanikę działania oraz tworzenie struktury aplikacji. Zaprezentowane zostaną wszelkie wykorzystane mechanizmy, które umożliwiają działanie programu jak również funkcje i logika odpowiedzialna za przebieg meczów oraz całych rozgrywek.

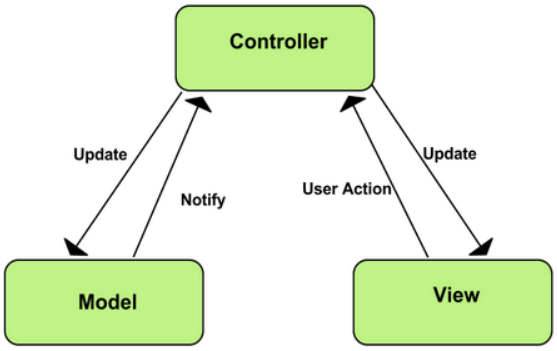
4.1. Pomocne Narzędzia

Zanim przejdę do omawiania funkcji sterujących aplikacją, muszę objaśnić kilka zagadnień bez których poznania ciężko będzie zrozumieć strukturę oraz mechanizmy które, są odpowiedzialne za działanie systemu.

Cały projekt został stworzony przy pomocy Qt Creatora – zintegrowanego środowiska programistycznego (IDE) działającego na różnych platformach, stworzonego z myślą o programistach piszących aplikacje w języku C++ korzystające z Qt. Składa się on z edytora kodu w C++ z funkcją uzupełniania kodu, komunikatami o błędach oraz ostrzeżeniami podczas pisania i narzędziami do szybkiej nawigacji w strukturze projektu. Dodatkowo środowisko zostało wzbogacone o Qt Designer – narzędzie do tworzenia formularzy metodą „przeciągnij i upuść”. Qt Creator oferuje wizualny debuger , narzędzia do budowania projektów i zarządzania nimi, a także możliwość generowania kodu.

4.1.1. Wykorzystanie Wzorca MVC

Framework Qt oparty jest na wzorcu „model-widok-kontroler” (MVC), który ułatwia stworzenie aplikacji, gdzie klasy danych(model) są odseparowane od klas związanych z prezentacją interfejsu użytkownika (widok). Kontroler jest odpowiedzialny za synchronizację danych z widoku do modelu. Kiedy użytkownik dokonując jakiejś zmiany na widoku np. naciska przycisk następuje wywołanie odpowiedniej metody w kontrolerze, która wprowadza zmiany w modelu. Kontroler pełni rolę pośrednika pomiędzy widokiem a modelem.



Rys. 1 Wzorzec MVC

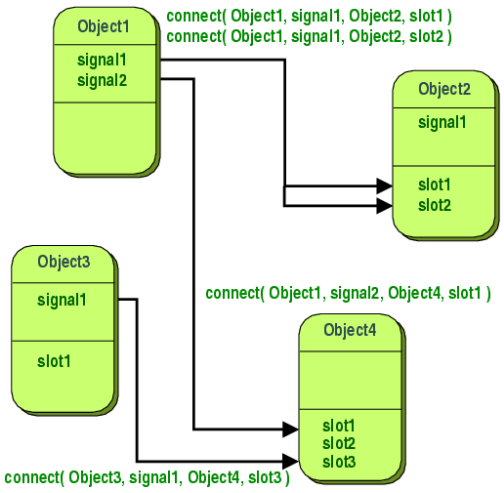
Takie podejście wymusza na programiście tworzenie osobnych klas odpowiedzialnych za model danych oraz klas odpowiedzialnych za prezentację widoku. Oddzielenie modelu od widoku zmniejsza złożoność każdego z tych elementów. Model i widok utrzymywane są w zupełnie inny sposób – zmiany są powodowane przez kompletnie inne czynniki dlatego łatwiej nad nimi panować, gdy nie są połączone. Kolejna zaleta separacji to możliwość stworzenia kilku różnych, spójnych widoków na te same dane.

„Architektura MVC obejmuje obiekty trzech rodzajów. Model to obiekt aplikacji, View odpowiada prezentacji widocznej na ekranie, a Controller określa jak interfejs ma reagować na działanie użytkownika. Przed wprowadzeniem architektury MVC w interfejsach użytkownika wszystkie te obiekty były zwykle połączone ze sobą. MVC rozdziela je co, zwiększa elastyczność i możliwości powtórnego wykorzystania projektu.” – Banda Czterech [Gamma95].

4.1.2. Sygnały i Sloty

Sygnały i sloty służą do komunikacji między obiektami. Mechanizm sygnałów i slotów (gniazd) jest centralną cechą Qt i prawdopodobnie częścią, która najbardziej różni się od funkcji zapewnianych przez inne frameworki. Sygnały i sloty są możliwe dzięki systemowi metaobiektu Qt.

Sygnał to komunikat widoczny w definicji klasy jako deklaracja pustej funkcji void. Ma on listę parametrów, ale nie ma ciała. Sygnał jest częścią interfejsu klasy. Wygląda jak funkcja, ale jest wywoływany w inny sposób: jest emitowany przez obiekt danej klasy.



Rys. 2 Sygnały i Sloty

Slot również najczęściej jest pustą funkcją składową. Można ją wywołać w tradycyjny sposób lub niebezpośrednio przez system QMetaObject. Sygnał jednego obiektu może zostać podłączony do slotów jednego lub więcej obiektów, o ile takie obiekty istnieją, a lista parametrów sygnału jest kompatybilna z listą slotu. Składnia instrukcji connect (podłącz) jest następująca:

```
connect(ui->down, SIGNAL(clicked(bool)), this, SLOT(volumeDown()));
```

Rys. 3 Połączenie sygnał – slot

Pierwszy parametr to wskaźnik na wysyłający obiekt, drugi nazwa sygnału z opcjonalną listą argumentów, trzeci wskaźnik na odbierający obiekt, czwarty nazwa slotu z opcjonalną listą argumentów.

4.1.3. Inteligentne wskaźniki

Język C++ nie posiada automatycznego mechanizmu odśmiecania pamięci (tzw. Garbage Collection), jednak istnieje kilka sposobów na automatyzację zarządzania pamięcią zajmowaną przez obiekty. Do najważniejszych takich sposobów należy zaliczyć inteligentne wskaźniki oraz zliczanie referencji. Qt posiada wiele typów inteligentnych wskaźników, stworzonych do realizacji różnych celów.

Klasę nazywamy inteligentnym wskaźnikiem, jeśli przesłania operator()* i operator ->(), czyli operatory wyłuskania wskaźnika. Pozwala to instancjom zachowywać się jak wskaźniki wbudowane. Klasy te prawie zawsze są szablonami, więc w definicjach musimy podawać odpowiednie typy w postaci argumentów szablonowych. Najłatwiej znaleźć przesłonięte wersje tych operatorów w iteratorach i właśnie w inteligentnych wskaźnikach. Nazwa „inteligentne” wynika z

niestandardowego zachowania się wskaźników podczas konstrukcji , destrukcji i przypisywaniu.

Klasa QScopedPointer to inteligentny wskaźnik, który automatycznie usuwa obiekt docelowy wskaźnika, gdy tylko wskaźnik wyjdzie poza zakres. Przypomina to std::auto_ptr. Jest to wskaźnik który zezwala na posiadanie tylko jednego egzemplarza wskaźnika na dany obiekt. Zakres wskaźnika wyraźnie zaznacza czas życia właściciela obiektu.

Klasa QSharedPointer również jest inteligentnym wskaźnikiem usuwającym obiekt docelowy, jednak dozwolone jest tworzenie kopii, a QSharedPointer przechowuje licznik referencji. Współdzielony obiekt na stercie jest niszczony dopiero wtedy, gdy zniszczony zostanie ostatni wskaźnik na niego

4.1.4. QStackedWidget

Klasa *QStackedWidget* jest wbudowaną klasą Qt i jest wykorzystywana do tworzenia interfejsu użytkownika podobnego do tego zapewnianego przez *QTabWidget*. Jej głównym zadaniem jest tworzenie listy widgetów poprzez dodawanie kolejnych elementów do stosu. Polega to na dodawaniu referencji istniejących widoków do obiektu klasy *QStackedWidget* przy pomocy odpowiednich metod. (Rys. 9-11). W ten sposób otrzymujemy obiekt zawierający listę widoków, którymi możemy manipulować w dowolny sposób(wyświetlać, usuwać, dodawać widoki). Klasa ta zawiera 8 metod, 2 sloty oraz 2 zdefiniowane sygnały.

- int indexOf (QWidget*) – metoda, zwraca indeks podanego widżetu na tej liście.
- int addWidget(QWidget*) – metoda, dodaje widget na koniec listy.
- int insertWidget (int, QWidget*) – metoda, dodaje widget pod wskazany indeks.
- void removeWidget(QWidget*) – metoda, usuwa widget z listy.
- int count() – metoda, zwraca liczbę widgetów z listy.
- QWidget *widget (int) – metoda, zwraca widget z podanego indeksu z listy widgetów.
- int currentIndex () – metoda, zwraca numer indexu aktualnie wyświetlanego na ekranie.
- QWidget * currentWidget () – metoda, pobiera aktualnie wyświetlany widget.
- void setCurrentIndex (int) – slot, zamienia aktualnie wyświetlany widget na ekranie na widget o podanym indeksie.
- void setCurrentWidget (QWidget*) – slot, zmienia aktualnie wyświetlany widget na ekranie na podany widget() (podany widget musi znajdować się na liście widgetów).
- void currentChanged(int) – sygnał, emitowany jest kiedy widget o podanym indeksie ulegnie zmianie.

- void widgetRemoved(int) – sygnał, emitowany jest kiedy widget o podanym indeksie jest usuwany.

4.1.5. Wstrzykiwanie zależności

Wstrzykiwanie zależności (ang. Dependency Injection) jest to wzorzec projektowy polegający na przekazywaniu gotowych instancji obiektów, które udostępniają swoje metody i właściwości obiektom, które z nich korzystają. W podejściu tym obiekt nie tworzy obiektów, które są wykorzystywane wewnątrz, tylko pobiera wcześniej utworzony obiekt jako parametr konstruktora a następnie inicjalizuje nim swoje składowe. Pozwala to na pozbycie się zależności pomiędzy komponentami i otwarcie kodu na rozszerzenia.

```
class PlayerSupportFunction
{
private:
    double gkeeper,defence, midfield,
           attack, tackle, technick,
           shot, pass, header, speed,
           overall;
    Player *player;
public:
    PlayerSupportFunction(Player *player){
        this->player = player;
    }
    void setSupportFunction();
    double setGoalKeeperRatings();
    double setCBRatings();
    double setRBLBRatings();
    double setDMRatings();
    double setCMRatings();
    double setRMLMRatings();
    double setAMRatings();
    double setRWLWRatings();
    double setSSRatings();
    double setCFRatings();
};
```

Rys.4 Wstrzykiwanie zależności

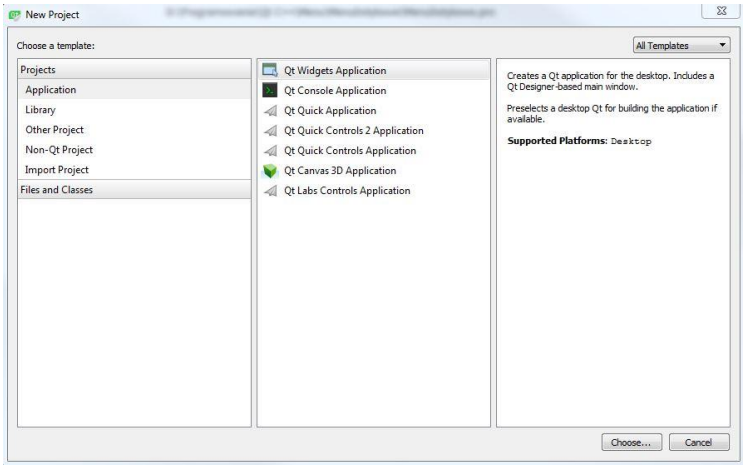
Na rysunku powyżej przedstawiono przykład wstrzykiwania zależności wykorzystywanej w aplikacji. Klasa *PlayerSuportFunction* pobiera przez konstruktor obiekt klasy *Player* i inicjalizuje nim własne pole tej klasy. Następnie w metodzie void setSuportFunction() inicjalizowane są pozostałe składowe klasy *PlayerSuportFunction* przy pomocy obiektu klasy ‘Player’. Dzięki temu pozostałe metody służące do wyliczania umiejętności poszczególnych graczy mają dostarczone parametry danego gracza.

4.2. Tworzenie Projektu

Jak już napisałem wcześniej framework Qt oparty jest na wzorcu „model-widok-kontroler", co w naturalny sposób wymusiło podział projektu na klasy danych(model), klas związanych z prezentacją interfejsu użytkownika (widok) oraz klas kontrolerów.

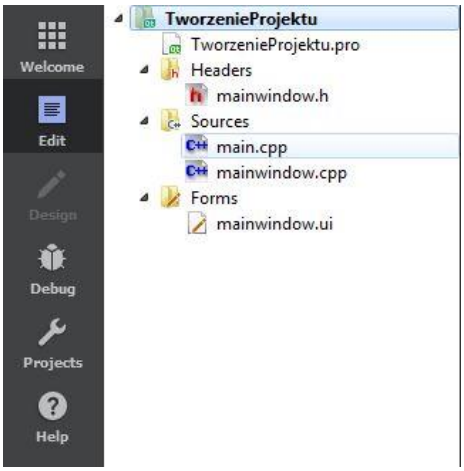
4.2.1. Nowy Projekt

Po uruchomieniu Qt Creatora i wybraniu opcji New Project dostajemy do dyspozycji okno dialogowe , w którym mamy do wyboru szablon projektu w jakim będziemy rozwijać naszą aplikację. W tym wypadku jest to Aplikations - Qt Widgets Application.



Rys. 5 Nowy projekt

Następnie w celu konfiguracji projektu będziemy musieli przejść przez kilka okien dialogowych. Po nadaniu nazwy dla projektu, wybraniu odpowiednich tzw. „Kit Selections", typu okna startowego, jego nazwy oraz skonfigurowaniu repozytorium dostajemy gotowy projekt startowy.

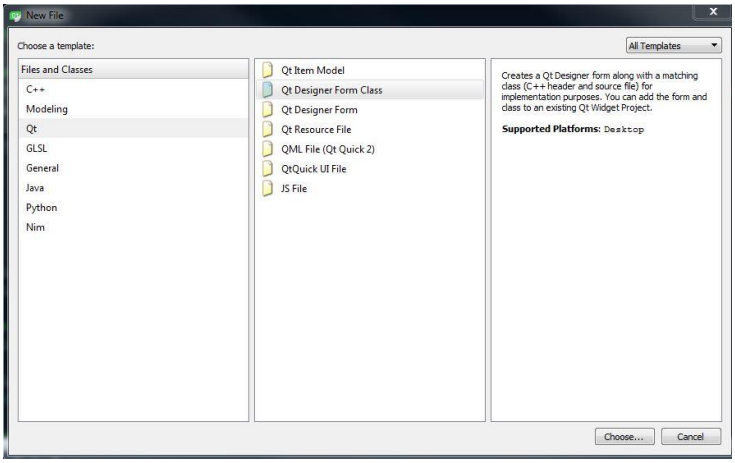


Rys. 6 Struktura Projektu Startowego

Projekt składa się z pliku z rozszerzeniem .pro, gdzie przechowywane są informacje o wersji Qt, jakie moduły zostały dołączone do projektu, spis plików źródłowych (.cpp), nagłówkowych (.h) oraz form (.ui). Poniżej znajdują się trzy katalogi, które służą do przechowywania trzech wyżej wymienionych plików. Plikiem startowym całej aplikacji jest main.cpp, to w nim wywoływane jest okno startowe aplikacji MainWindow, które posiada wskaźnik na niewidoczny w projekcie plik ui_mainwindow.h. Jest to plik generowany automatycznie przez Qt Designera i zawiera kod odpowiadający wszystkim elementom, które tworzymy na formie mainwindow.ui. Widok jest jednocześnie generowany do pliku xml, w którym możemy manipulować kontrolkami znajdującymi się na widoku, jak i samym widokiem. Plik ui_mainwindow.h zawiera metodę setupUI(QMainWindow *MainWindow), która pobiera wskaźnik na obiekt mainwindow w celu ustawienia wszystkich elementów formy na layoucie okna głównego. Do tak skonfigurowanego projektu możemy dodawać kolejne okna dialogowe jak również klasy C++ odpowiedzialne za operacje na modelach danych.

4.2.2. Tworzenie Widoku

Klasy widoku nie są widoczne w samym projekcie, generowane są automatycznie i zapisywane w osobnym katalogu. Dostęp do widoków możemy uzyskać poprzez kliknięcie na odpowiedni plik w katalogu Forms. Uzyskujemy w ten sposób dostęp do narzędzia zwanego Qt Designer, który umożliwia dodawanie do formy layoutów, przycisków, kontenerów, kontrolek odpowiadających za wprowadzanie i wyświetlanie danych oraz ich dowolną konfigurację i stylizację.



Rys. 7 Dodawanie Formy do projektu

Projekt można rozbudować o dowolną ilość form. Po wygenerowaniu nowej formy Qt Creator automatycznie generuje plik ui_nazwawapliku.h – jest to kod widoku. Dodatkowo generuje plik nagłówkowy (.h) oraz plik źródłowy (.cpp) i zapisuje w odpowiednich katalogach. Plik z rozszerzeniem .cpp staje się tutaj kontrolerem to w jego ciele będzie można łączyć logikę aplikacji i wyświetlać dane na widok(formę).

4.2.3. Tworzenie Modelu

Klasy modeli odpowiedzialne za manipulowaniem na danych tworzone są w podobny sposób jak klasy widoków podczas dodawania formy do projektu. Różnica polega na tym, że zamiast szablonu Qt wybieramy C++. W ten sposób tworzymy klasy modeli, które zapisywane są odpowiednio do katalogów Headers i Source.

4.3. Budowa i Struktura Aplikacji

W poprzednim podrozdziale zostało opisane ogólne tworzenie nowego projektu oraz podział plików i katalogów projektu w zależności od pełnionej roli. W tej części skupię się na pokazaniu, jak zbudowany jest ten konkretny projekt. Jakie zależności i interakcje zachodzą pomiędzy klasami widoków i modeli.

4.3.1. Start Aplikacji

Plikiem startowym całej aplikacji jest plik main.cpp. Plik ten znajduje się w katalogu Sources, nie posiada on pliku nagłówkowego natomiast zawiera funkcjęint int main(), która jest główną funkcją całej aplikacji. Funkcja int main() wywołuje obiekt klasy QApplication, która zarządza przepływem sterowania i głównymi ustawieniami aplikacji GUI. Dla każdej aplikacji GUI używającej Qt istnieje dokładnie jeden obiekt QApplication, bez względu na to czy aplikacja ma jedno czy nawet kilka okien otwartych w danym momencie.

```
#include "intro.h"
#include "container.h"
#include <QApplication>
#include <QSharedPointer>

QSharedPointer<Container>container(new Container());

int main(int argc, char *argv[])
{
    QCoreApplication::addLibraryPath(".");

    QTime time = QTime::currentTime();
    qsrand((uint)time.msec());

    QApplication a(argc, argv);
    container->loadDefaultData();

    a.setWindowIcon(QIcon("Button/ball.png"));
    Intro w;
    w.show();
    return a.exec();
}
```

Rys. 8 Plik main.cpp

Dodatkowo funkcja int main() zawiera klasę QTime, która zapewnia funkcję zegara. Obiekt QTime zawiera czas zegarowy, tj. liczbę godzin, minut, sekund i milisekund od północy. Może odczytywać aktualny czas z zegara systemowego i mierzyć przedział czasu, który upłynął. Zapewnia funkcje porównywania czasów i manipulowania nim poprzez dodanie liczby milisekund. QCoreApplication::addLibraryPath(".") zwraca ścieżkę do pliku wykonywalnego aplikacji. Linia kodu a.setWindowIcon(QIcon("Button/ball.png")); ustawia ikonę okna głównego aplikacji. Następnie wywoływany jest obiekt klasy *Intro* poprzez metodę show() i następuje uruchomienie GUI widoku intro.ui.

Widok intro.ui jest widokiem początkowym, który po naciśnięciu przycisku przenosi nas do widoku mainmenu.ui, który jest z kolei głównym widokiem całej aplikacji. Przejście z widoku intro.ui do maimenu.ui odbywa się poprzez utworzenie obiektu klasy MainMenu, następnie wywoływana jest metoda statyczna klasy *Intro,* która zamyka widok intro.ui, tak aby na koniec wywołać obiekt klasy *MainMenu* zawierający widok mainmenu.ui.

```
MainMenu mainmenu;
Intro::close();
mainmenu.exec();
```

.

Rys. 9 Uruchomienie widoku MainMenu

Takie podejście w uruchomianiu okien nie jest najlepszym sposobem. Sprawdza się podczas uruchamiania okien modalnych, które pojawiają się na wierzchu okna macierzystego, a następnie po wykonaniu potrzebnych operacji jest zamykane.
Kiedy zależy nam na płynnym przemieszczaniu między pełnoekranowymi widokami lepszym sposobem jest wykorzystanie klasy QStackedWidget oraz sygnałów i slotów do przemieszczania się między widokami.

4.3.2. QStackedWidget, Sygnały i Sloty w praktyce.

MainMenu jako widok główny aplikacji zawiera w swojej strukturze odwołania pośrednie lub bezpośrednie do pozostałych widoków aplikacji. Bezpośrednio przy pomocy przycisków umieszczonych na górze ekranu możemy przejść do takich widoków jak *Wybór Drużyny*, *Menu Gry* , *Menu Opcji* czy *Abaut*. Koncepcja taka polega na tym, że w obiekcie klasy MainMenu (właściwie jego widoku Ui_mainMenu) tworzony jest obiekt klasy QstackedWidged do którego dodawany jest w indeksie zerowym layout widoku *MainMenu*. Następnie na kolejnych indeksach dodawane są kolejne widoki (*About*, *Opcji*, *Wyboru Drużyny*, *Menu Gry*). W ten sposób otrzymujemy kontener, który przechowuje wskaźniki lub referencje do odpowiednich obiektów.

```
QStackedWidget *stackedWidget = new QStackedWidget(mainMenu);
stackedWidget->setCurrentIndex(0);
```

Rys 10. Utworzenie obiektu klasy QstackedWidged w Klasie Ui_MainMenu

```
ui->stackedWidget->insertWidget(1, &about);
ui->stackedWidget->insertWidget(2, &options);
ui->stackedWidget->insertWidget(3, choseTeam);
ui->stackedWidget->insertWidget(4, userDataDialog);
ui->stackedWidget->insertWidget(5, gameDialog);
```

Rys. 11 Dodawanie referencji oraz wskaźników obiektów do listy

Przemieszczanie pomiędzy widokami następuję po naciśnięciu przycisku znajdującego się na górze ekranu. Przykładowo po wciśnięciu przycisku ‘Nowa Gra’ wywoływana jest metoda setCurrentIndex() obiektu stackedWidget – z parametrem 3, co oznacza ustawienie widoku ‘Wybór Drużyny ’ jako głównego layoutu. Dodatkowo wywoływana jest funkcja klasy *Container*, która zmienia tło widoku. (klasa *Container* zostanie opisana później) .

```
QPixmap bkgnd("Stadiony/realfuture.jpg");
container->functions->setBackground(this, bkgnd);
 ui->stackedWidget->setCurrentIndex(3);
```

Rys. 12 Zmiana wyświetlanego layoutu oraz jego tła.

Możliwe jest również utworzenie kolejnego kontenera widoków na nowym layoucie. Takie rozwiązanie zastosowane jest w widoku ‘Opcje’, gdzie utworzony jest nowy niezależny kontener klasy *QStackedWidget* do którego dodawane są kolejne referencje do innych widoków. Jest to dobry sposób na odciążenie głównego widoku od zbytniej odpowiedzialności oraz porządkuje kod.

Kolejnym rozwiązaniem jest zastosowanie *Sygnałów i Slotów* jest to mechanizm służący do komunikacji między obiektami. Przykładowym zastosowaniem tej funkcjonalności może być sytuacja kiedy znajdujemy się w widoku *Menu Gry*, który nie posiada przycisków widoku *Menu Głowne* i chcemy dostać się do widoku *Opcji*. W takim wypadku wymagane jest kilka kroków które umożliwią komunikację między wyżej wymienionymi widokami.

```
connect(gameDialog, SIGNAL(dialogClicked()), this, SLOT(bakToMainMenu()));
```

Rys. 13 Utworzenie połączenia pomiędzy widokami.

W pierwszym kroku tworzone jest w klasie *MainMenu* połączenie pomiędzy widokiem *Menu Gry* (*gameDialog*) a *Menu Głowne* (*this*). Następnie tworzona jest metoda backToMainMenu() w której za pomocą metody setCurentIndex() klasy QstackedWidget ustawiany jest indeks na pozycje zero (*Menu Główne*) . Jest to tak zwana funkcja powrotu.

```
signals:
    void dialogClicked();
```

Rys. 14 Ustawienie sygnału w klasie 'GameDialog'

W kolejnym kroku w klasie *GameDialog* ustawiany jest sygnał dialogClicked(), tak aby po naciśnięciu przycisku *Menu Główne* została wywołana metoda on_mainmenu_clicked(), która wyemituje sygnał dialogClicked(). Wówczas uruchamiana jest funkcja connect, uruchamiającą sygnał wywołujący slot, który zawiera metodę backToMainMenu().

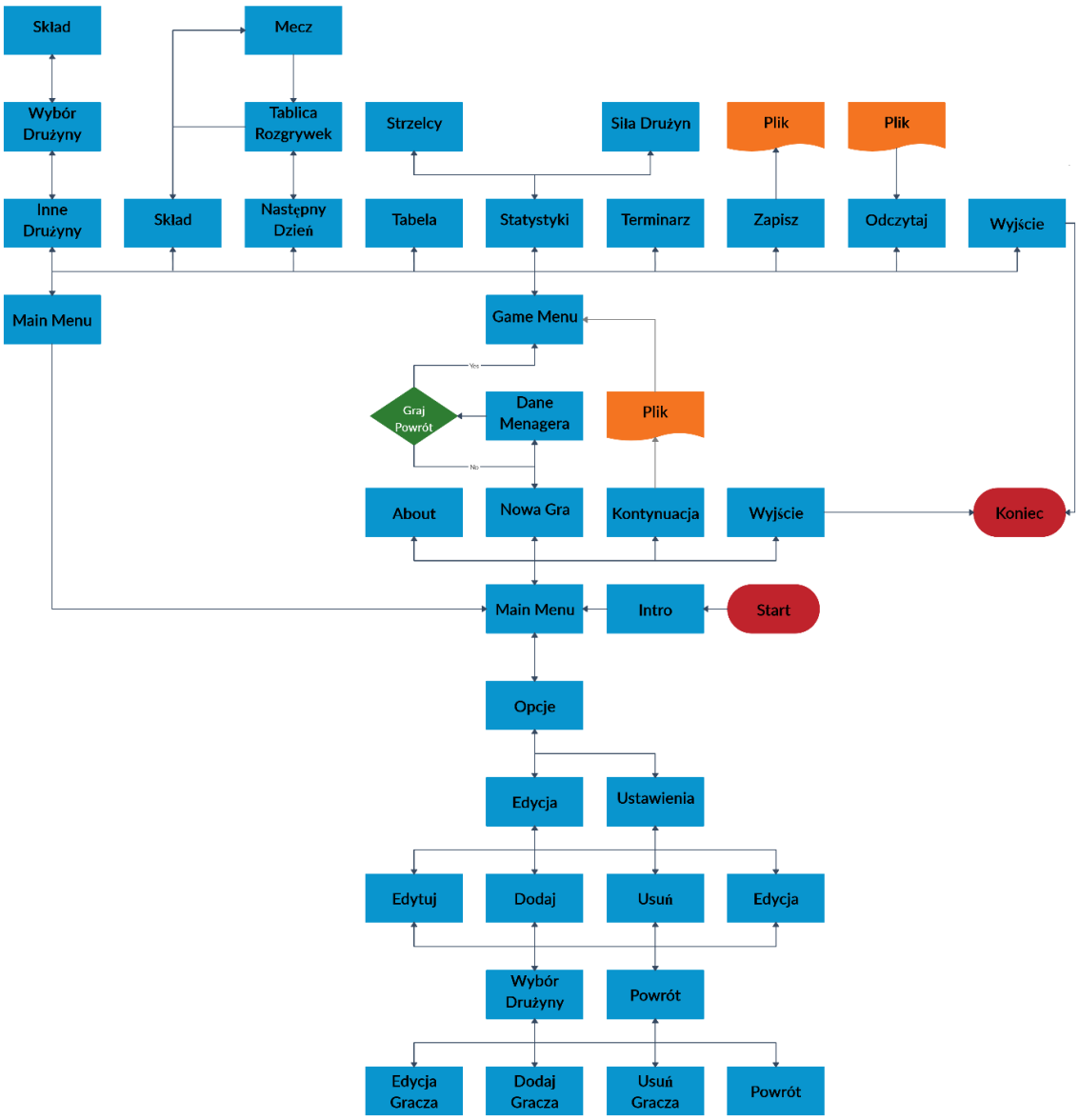
```
void GameDialog::on_mainmenu_clicked()
{
    emit dialogClicked();

    container->music_player->stop();
    container->loadDefaultData();
}
```

Rys.15 Wywołanie sygnału dialogClicked()

W tym podrozdziale zostało zaprezentowane działanie w praktyce wcześniej omówionych mechanizmów takich jak *QStackedWidget* oraz *Sygnały i Sloty*. Takie podejście zastosowane jest w całym projekcie zapewniając płynniejsze działanie aplikacji, lepsze zarządzanie pamięcią oraz poprawia czytelność kodu. Schemat budowy aplikacji zaprezentowany jest w rozdziale - 4.3.3. Schemat blokowy aplikacji.

4.3.3. Schemat blokowy aplikacji



Rys. 16 Schemat blokowy aplikacji

4.4. Klasy Aplikacji

4.4.1. Klasa C++ vs Qt

Standardowa klasa C++ składa się z konstruktora destruktora składowych prywatnych, publicznych oraz dodatkowo protected jeżeli klasa jest klasą bazową dla klasy pochodnej. Klasa Qt jest bardziej złożona i zawiera więcej elementów. Pierwszym istotnym elementem, którego nie posiada klasa C++ jest makro Q_OBJECT. Qt posiada specjalny kompilator 'Meta-Object-Compiler' (moc), który obsługuje rozszerzenia C++ Qt. Kiedy napotka jedną lub więcej deklaracji klas, które zawierają makro Q_OBJECT, generuje plik źródłowy C++ (np. moc_options.cpp) zawierający kod meta-obiektu dla tych klas. Między innymi wymagany jest kod meta-obiektu dla mechanizmu sygnałów i slotów. Plik źródłowy C++ wygenerowany przez moc musi zostać skompilowany i powiązany z implementacją klasy.

```
class Versus
{

public:

    Versus(int fix);
    ~Versus() {}

    void pairList();
    void pairProgress();
    void setTeamsPairs();
    int getPair(int i){return pair[i];}
    int getTeamsPaired(int i){return teams[i];}

private:

    QList<QPair<int, int> > list;
    int fixture;
    int pair[8];
    int teams[16];
};
```

```
class Options : public QDialog
{
    Q_OBJECT

public:
    explicit Options(QWidget *parent = 0);
    ~Options();

private slots:
    void on_mainmenu_clicked();
    void on_edit_clicked();
    void bakToMainMenu();

signals:
    void dialogClicked();

private:
    Ui::Options *ui;
    Edit edit;
};
```

Rys.17 Porównanie klasy C++ i Qt

Najistotniejszą jednak różnicą jest to, że każda klasa Qt dziedziczy publiczne składowe klasy *QDialog*. Klasa *QDialog* jest podstawową klasą okien dialogowych.

Okno dialogowe to okno najwyższego poziomu używane najczęściej do krótkoterminowych zadań i krótkiej komunikacji z użytkownikiem. *QDialogs* mogą być modalne lub niemodalne.

Kolejną różnicą jest występowanie domyślnie słowa kluczowego explicit, które zabrania używania konstruktora jednoargumentowego do niejawnych konwersji. Sam konstruktor pobiera wskaźnik na obiekt klasy QWidget umożliwiający manipulowanie elementami graficznymi rodzica . Dodatkowo klasa Qt wyposażona jest w identyfikatory signals i slots, które umożliwiają deklarowanie i definiowanie sygnałów i gniazd.

4.4.2. Klasy Modelów

Za sterownie logiką aplikacji odpowiadają klasy modeli, które w odróżnieniu od klas kontrolerów stworzone zostały jako czyste klasy języka C++. Klasy te odpowiedzialne są za wyliczanie parametrów odpowiadających między innymi za przebieg meczu, serializację danych, umiejętności zawodników oraz statystyki drużyn.

Klasa *Container* inicjalizowana jest jako obiekt globalny przy pomocy inteligentnego wskaźnika w pliku main.cpp [Rys.77]. Zapewnia to szybki dostęp do jej składowej z każdego miejsca w programie wystarczy wywołać ją w pliku .cpp (extern Container *container). Klasa *Container* spełnia w aplikacji dwie funkcje. Pierwszą z nich jest przechowywanie wskaźników na inne klasy modeli. Jest to bardzo wygodny sposób ułatwiający szybki dostęp do wszystkich funkcji aplikacji. Drugą funkcjonalność zapewniają 4 metody.

```
class Container
{

public:
    Container();
    ~Container();
    QVector<QVector<Player>> player;
    QList<Team> teams;
    League *league;
    MusicPlayer *musicPlayer;
    Functions *functions;
    MatchSimulation *matchSimulation;
    Serialization *serialization;
    Versus *versus;
    ShowTeamStats *showTeamStats;
    UserData *userData;
    ArtificialIntelligence *artificialInteligence;
    MatchAlgorithms *matchAlgoritm;
    Hovered *hovered;
    Formations *formations;
    SetMultimedia *setMultimedia;

    void loadDefaultData();
    void saveData();
    void loadData();
    void memeoryHarvester();

};
```

Rys. 18 Klasa 'Container'

- void loadDefaultData() – metoda, wywoływana jest przy starcie aplikacji [Rys.77]. Jej głównym zadaniem jest utworzenie wszystkich obiektów wymaganych do prawidłowego działania systemu. Obiekty te odpowiadają za wypełnienie danymi początkowymi wszystkich drużyn oraz zawodników.
- void saveData() – metoda, zapisuje stan aplikacji w trakcie jej działania.
- void loadData() – metoda, przywraca stan aplikacji po jej ponownym wznowieniu
- void memoryHarvester() – metoda, służąca do czyszczenia pamięci.

Klasa *ArtificialIntelligence* używana jest przez komputer do podejmowania decyzji w trakcie rozgrywanego meczu. Składa się z 4 metod ustawiających oraz 3 metod zarządzających drużyną sterowaną przez AI.

- void setId(int) - ustala którą drużyną steruje AI
- void setAway(bool) – ustawia komputer jako drużynę grającą na wyjeździe
- void setHome(bool) – ustawia komputer jako gospodarza
- void setDefaultPressing(int) – przywraca początkowy pressing drużyny komputera
- void setStrategy(int,int,int) – metoda ta pobiera trzy parametry w czasie meczu: stan zegara, wynik gospodarza i gościa i w zależności od wyniku oraz minuty meczu zwiększa lub zmniejsza pressing drużyny sterowanej przez AI
- void changePlayer() – zmienia zawodnika w trakcie meczu w zależności od tego, który jest najbardziej zmęczony – bierze pod uwagę pozycję gracza oraz ilość dokonanych zmian
- void changePlayerSimulation(int, bool) – podobnie jak metoda wyżej tylko jest wykonywana w trakcie symulacji meczu AI vs AI

Metody klasy ArtificialIntelligence wykorzystywane są w klasie 'LiveMatch' w metodzie computerStrategy.

Klasa *Formations* składa się z 3 metod z możliwością dodania dodatkowych metod w celu zwiększenia wyboru dostępnych formacji. Klasa ta jest odpowiedzialna za nadawanie odpowiednich kolorów zawodników grających w odpowiedniej formacji.

- void formation442(QTableWidget *, QStringList)
- void formation433(QTableWidget *, QStringList)
- void formation352(QTableWidget *, QStringList)

Każda metoda pobiera wskaźnik na obiekt QTableWidget oraz QStringList i w zależności od wybranej metody ustawia kolory dla poszczególnych formacji. Przykładowo metoda formation442 nadaje pierwszemu wierszowi obiektu QTableWidget kolor, żółty, czterem kolejnym zielony, następnym czterem niebieski i dwóm ostatnim czerwony.

Rys. 19 Efekty użycia klasy Formations

Klasa *Functions* jest tak zwaną ‘klasą użyteczności’ (ang. Utility Class). W jej skład wchodzi 11 metod służących głównie do ustawiania grafiki na wybranych elementach oraz dwie metody związane z generowaniem liczb i czasu.

- int randInt(int, int) – pobiera dwie liczby, które stanowią dolny i górny zakres przedziału z którego losowana jest liczba pseudolosowa.
- void delay(double) - ustawia opóźnienie, które podajemy jako parametr w milisekundach.
- void setIcon(QPixmap*, QLabel*, int, int) – metoda ustawia ikonę na etykiecie. Pierwsze dwa parametry to w wskaźniki na ikonę, oraz etykietę na której ma być ustawiony. Pozostałe dwa parametry określają rozmiar obrazka.
- void setIcon(QPixmap*, QButton*, QString) - przeciążona wersja funkcji setIcon(). Różnica polega na tym, że obrazek ustawiany jest na przycisku. Dodatkowo pobierany jest tekst który jest wyświetlany po ustawieniu kursora nad przyciskiem.
- void setIcon(QPixmap*, QLabel*, int, int, int) – kolejne przeciążenie metody setIcon(). W odróżnieniu od pierwszej wersji, ostatnim parametrem jest liczba milisekund, która powoduje opóźnienie wyświetlenia obrazka.

- void setHighLight(QRect, QPushButton*, QString, QString) – jej zadaniem jest odpowiednie podświetlenie obrazka po najechaniu kursorem. W tym celu, jako pierwszy parametr pobierana jest pozycja przycisku na ekranie, drugi parametr to wskaźnik na przycisk oraz dwa parametry które w zależności od tego czy kursor jest nad przyciskiem czy nie ustawiają odpowiednie właściwości podświetlenia.
- void showFrame(QTableWidget*, QRect) – pobiera wskaźnik na obiekt tabeli oraz jego pozycję i w zależności czy kursor jest na pozycji tabeli wyświetla ją lub ukrywa.
- void showTeamStats(ShowTeamStats *, QRect , int) – pobiera wskaźnik na obiekt klasy ShowTeamStats i dodatkowo id drużyny. Wyświetla go lub ukrywa ze statystykami odpowiedniej drużyny w zależności od tego na której drużynie ustawiony jest kursor.
- void showPlayerStats(ShowPlayerStats *, QRect rect, int) – podobnie jak metoda opisana powyżej tylko wyświetla statystyki zawodników.
- void setBackground(QMainWindow *, QPixmap) – pobiera wskaźnik na obiekt QMainWindow oraz obrazek i ustawia je jako tło.
- void setBackground(QDialog, QPixmap) - pobiera wskaźnik na obiekt QDialog oraz obrazek i ustawia je jako tło.
- void setLabelTextColor(QLabel*, int) - ustawia kolor tekstu na etykiecie w zależności od drugiego parametru.
- void setLabelTextColor(QLabel*,QString) – przeciążona wersja metody setLabelTextColor() – ustawia kolor tekstu na etykiecie w zależności od drugiego parametru.

Klasa *League* używana jest do sterowania przebiegiem sezonu. Składa się z 17 metod, które podzielone są na 6 metod zwracających, 6 metod ustawiających oraz 5 metod służących do zarządzania kalendarzem rozgrywek.

- int getDay()const - zwraca dzień miesiąca
- int getMonth()const - zwraca miesiąc.
- int getYear()const – zwraca rok.
- int getFixture()const – zwraca numer kolejki.
- void getDays() – zwraca kolejny dzień rozgrywek.
- void getMatchDays() – zwraca numer dnia meczowego.
- void setDay(int) – ustawia dzień miesiąca.
- void setMonth(int) – ustawia miesiąc.
- void setYear(int) – ustawia rok.
- void setFixture(int) – ustawia kolejkę.
- void setDays(int) – ustawia kolejny dzień rozgrywek.
- void setMatchDays(int) – ustawia następny dzień meczowy.
- void nextDay() – inkrementuje dzień o jeden.
- void calendar() – metoda zarządzająca kalendarzem. Kontroluje ilość dni w miesiącu, zmienia miesiące oraz lata.

- void setFixture() – zwiększ kolejkę o jeden.
- void setDays() – inkrementuje dzień rozgrywek o jeden
- void setMatchDays() – inkrementuje następny dzień meczowy.

Klasa *MatchAlgorithms* – jest jedną z najbardziej rozbudowanych klas w całej aplikacji. Składa się z 33 metod: 7 metod ustawiających, 18 metod zwracających oraz 8 metod zarządzających przebiegiem meczu.

- void setClock(int) – metoda ustawia zegar meczowy.
- void setAtributes(int , int) - metoda zeruje atrybuty drużyny gospodarza i drużyny przyjezdnej takie jak: wynik, strzały, strzały celne oraz siłę drużyn.
- void setParametrs() – ustawia parametry drużyn takie jak: pressing, energia oraz siła drużyn.
- void shotOrNot() – metoda ustala która drużyna ma oddać strzał. Ustalane jest to na podstawie różnicy pomiędzy siłami drużyny gospodarza i drużyny przyjezdnej. Przykładowo jeżeli drużyna gospodarzy ma siłę 90 a gości 85 to różnica wynosi 5. Jeżeli byłoby odwrotnie to współczynnik wyniósłby -5. Następnie losowana jest liczba z zakresu 1-100. Dla współczynnika 5 czyli korzystnego dla gospodarza przeprowadzony jest w warunku switch-case następujący przypadek:

```
case 5:
        if(range >= 86 and range <= 100)
            strikesHome();
        if(range >= 0 and range <= 8)
            strikesAway();

break;
```

Rys. 20 Przypadek korzystny dla gospodarza.

W tym przypadku szanse na oddanie strzału wynoszą 14 do 8 dla gospodarza ponieważ jego zakres liczby wynosi od 86-100 natomiast drużyny przyjezdnej od 0-8. Jeżeli zostanie wylosowana liczba pomiędzy 8 a 86 żadna z drużyn nie oddaje strzału.

```
case -5:
        if(range >= 91 and range <= 100)
            strikesHome();
        if(range >= 0 and range <= 14)
           strikesAway();
break;
```

Rys. 21 Przypadek niekorzystny dla gospodarza.

W tym przypadku szanse na oddanie strzału dla gospodarza wynoszą 9 do 14, ponieważ drużyna gospodarzy posiada niekorzystny współczynnik wynikający z różnicy sił. Szanse są jednak minimalnie powiększone przez fakt że drużyna gospodarza rozgrywa mecz na własnym boisku. Metoda ta jest wywoływana 90-97 razy podczas każdego meczu.

- void strikesAway() – zwiększa licznik strzałów drużyny przyjezdnej o jeden oraz wybiera czy strzelec ma być wybierany w metodzie getStriker(int, bool) lub getStrikerLive(int, bool) – jest to uzależnione od tego czy rozgrywany jest mecz gracz vs AI czy AI VS AI
- void strikesHome() – zwiększa licznik strzałów drużyny gospodarzy o jeden oraz wybiera czy strzelec ma być wybierany w metodzie getStriker(int, bool) lub getStrikerLive(int, bool) – jest to uzależnione od tego czy rozgrywany jest mecz gracz vs AI czy AI VS AI
- void strikerValue(int) – metoda ustala który z zawodników danej drużyny ma największe szanse na oddanie strzału po tym jak jego drużynie w wyniku metody shotOrNot() zostanie przyznany strzał. Polega to na tym, że zawodnikom drużyny przyznawane są losowo punkty w zależności od umiejętności zawodnika w ataku. Przykładowo dla najlepszych zawodników posiadających umiejętności powyżej 95 losowane są liczby z przedziału 83-99. Im umiejętności danego zawodnika są słabsze tym przedział losowanych liczb jest zwiększany i szanse na wylosowanie wysokiej liczby maleją. Liczby te zapisywane są dla każdego zawodnika drużyny w specjalnej zmiennej i przechowywane, aż do następnej szansy oddania strzału.
- void getStrikerLive(int, bool) – metoda pobiera imię, nazwisko i pozycję zawodnika który ma najwięcej punktów przyznanych w wyniku wykonania metody strikerValue(int). Następnie losuje liczbę z zakresu 1-100 i dodaje ją do punktów uzyskanych z metody strikerValue(int). W kolejnym kroku sprawdza pozycję gracza i jeśli graczem jest środkowy obrońca to dodaje do sumy punktów umiejętności gry głową a jeżeli z innej pozycji to umiejętności strzeleckie. Jeżeli suma punktów wynosi ponad 160 to zawodnik zdobywa bramkę. Wszystkie sumy poniżej tej wartości kończą się strzałem niecelnym, obronionym przez bramkarza, zablokowanym, poprzeczką bądź słupkiem. Dodatkowo metoda zwiększa ilość goli strzałów, strzałów celnych oraz ustawia gole zdobyte przez zawodników. Jest również odpowiedzialna za ustawianie komentarzy w zależności od zaistniałej sytuacji.
- void getStriker(int, bool) - jest to metoda bardzo podobna do getStrikerLive(int, bool). Różnica polega na tym, że jest ona wykorzystywana do meczów AI vs AI i nie ustawia komentarzy.
- void setStatistic() – metoda ustawia pomeczowe statystyki dla drużyn. W zależności od wyniku meczu przyznaje punkty, dodaje drużynom bramki zdobyte i stracone, dodaje zwycięstwa, remisy bądź porażki.
- void clearVariables() – zeruje zmienne klasy.
- setLiveMatch(bool) – ustawia czy mecz jest symulowany czy rozgrywany na żywo.
- int getHomeScore() – zwraca wynik drużyny gospodarza.
- int getAwayScore() – zwraca wynik drużyny przyjezdnej.
- int getHomeTeamPower() – zwraca siłę drużyny gospodarza.
- int getAwayTeamPower() – zwraca siłę drużyny przyjezdnej.

- int getHomeShots() – zwraca ilość strzałów oddanych przez drużynę gospodarza.
- int getAwayShots() – zwraca ilość strzałów oddanych przez drużynę przyjezdną.
- int getHomeTarget() - zwraca ilość strzałów celnych oddanych przez drużynę gospodarza.
- int getAwayTarget() - zwraca ilość strzałów celnych oddanych przez drużynę przyjezdną.
- int getDifference() – zwraca różnicę sił pomiędzy drużynami.
- QString getComentary() – zwraca komentarz po przeprowadzonej akcji.
- QString getAction() – zwraca rodzaj przeprowadzonej akcji.
- QString getStrikersName() – zwraca nazwisko strzelca.
- QString getTeamName() – zwraca nazwę drużyny.
- QString getHomeStriker() – zwraca nazwisko strzelca drużyny gospodarza.
- QString getAwayStriker() – zwraca nazwisko strzelca drużyny przyjezdnej.
- getStartDisplay(bool),setStartDisplay(),getHomeGoalFlag(), setHomeGoalFlag(bool), getAwayGoalFlag(), setAwayGoalFlag(bool) – metody służące do ustawiania i zdejmowani flag wykorzystywanych w innych metodach.

Klasa *MatchSimulation* - służy do symulowania meczów rozgrywanych pomiędzy AI. Posiada tylko 1 metodę.

- void matchProgres() – metoda ta w pętli trwającej 90 kroków wywołuje wszystkie niezbędne metody klasy ‘MatchAlgorithm’ oraz ‘ArtificialIntelligence’ umożliwiające przeprowadzenie symulacji meczu.

Klasa *MusicPlayer* odpowiedzialna jest za sterowanie dźwiękiem w całej aplikacji. Wykorzystuje ona klasę QMediaPlayer, która służy do odtwarzania multimediów. Można jej używać do odtwarzania takich treści jak piosenki, filmy czy radio internetowe. Klasa ‘MusicPlayer’ zawiera 12 publicznych metod, które stanowią podstawowe funkcje playera:

- void setPlayer() – metoda ta ustawia główne funkcje odtwarzacza, takie jak ustawienie listy odtwarzania, ustawienie początkowego indeksu odtwarzania czy głośności.
- void start() – rozpoczyna odtwarzanie utworu
- void stop() – zatrzymuje odtwarzanie utworu (wraca do początku po wznowieniu)
- void pause() - wstrzymuje odtwarzanie utworu
- void volumeTurnUp() - podgłaśnia utwór o 10%

- void volumeTurnDown() – ścisza utwór o 10%
- void previous() - cofa o jeden utwór do tyłu
- void forward() – przełącza o jeden utwór do przodu
- void smoothVolumeDown() – wycisza stopniowo muzykę
- void smoothVolumeUp() – podgłaśnia stopniowo muzykę
- int getVolume() – zwraca głośność utworu.
- void setTrack(int) – ustawia wybrany utwór.

Obiekt klasy *MediaPlayer* posiada swój wskaźnik w klasie 'Container', dlatego może być wykorzystywana w każdym miejscu aplikacji bez tworzenia jej instancji. Jest integralną częścią odtwarzacza utworów wbudowanego w widok 'Menu Gry'. Często jest też używana, kiedy użytkownik przełącza się między widokami w celu zmiany utworu oraz płynnego wyciszenia lub podgłośnienia muzyki.

Klasa *Player* wykorzystywana jest w aplikacji do tworzenia zawodników oraz zarządzania nimi w trakcie rozgrywek. Składa się z 53 metod: 7 metod zarządzających, 23 metod pobierających oraz 23 metod ustawiających.

- int getOverall(in, int) - metoda, która wylicza umiejętności gracza na danej pozycji. Bierze pod uwagę takie parametry jak rodzaj formacji oraz to czy gracz znajduje się na właściwej dla siebie pozycji.
- int getEnergyOverall(int, int) - podobne działanie jak metoda powyżej. Dodatkowo umiejętności gracza pomniejszane są o zmęczenie.
- int getRealOveral() - wylicza umiejętności gracza na jego nominalnej pozycji nie pobiera żadnych innych parametrów.
- void staminaDecreaser(QVector<Player> &, int) - wylicza ubytek energii każdego zawodnika na boisku. Bierze pod uwagę takie parametry jak pressing drużyny oraz pozycję na której znajduje się zawodnik.
- void playerRecovery(QVector<Player>&) - metoda służąca do regeneracji zawodników po odbytym meczu. Bierze pod uwagę wiek gracza. Im starszy zawodnik tyn regeneracja przebiega wolniej.
- void setPlayerSupportFunction() - metoda która wywołuje metody klasy 'PlayerSuportFunction', która jest wykorzystywana do obliczeń pomocniczych w metodach getOverall(in, int), getEnergyOverall(int, int) oraz getRealOveral().
- static void swapPlayer(Player &, Player &) – zamienia pozycjami dwóch zawodników.
- int getId()const - zwraca numer indeksu danego gracza.
- QString getFname()const - zwraca imię zawodnika.
- QString getLname()const - zwraca nazwisko zawodnika.
- QString getNationality()const - zwraca narodowość zawodnika.
- QString getPosition()const - zwraca pozycję gracza.
- double getGkeeper()const - zwraca umiejętności bramkarskie zawodnika.
- double getMidfield()const - zwraca umiejętności gry zawodnika w pomocy.

- double getAttack() const - zwraca umiejętności gry zawodnika w ataku.
- double getTackle()const - zwraca umiejętności gry zawodnika w odbiorze piłki.
- double getTechnik()const - zwraca umiejętności techniczne zawodnika.
- double getShot()const - zwraca umiejętności strzeleckie zawodnika.
- double getPas()const - zwraca umiejętności w celności podań zawodnika.
- double getHeader()const - zwraca umiejętności gry głową zawodnika.
- double getSpeed()const - zwraca szybkość zawodnika.
- double getEndurance()const - zwraca wytrzymałość zawodnika.
- double getStamina()const - zwraca energię gracza.
- int getGoals()const - zwraca ilość zdobytych goli przez zawodnika.
- int getAge()const - zwraca wiek zawodnika.
- int getNumber()const - zwraca numer zawodnika.
- int getShotTemp()const - zwraca tymczasowy współczynnik strzału zawodnika wykorzystywany w metodzie strikerValue() klasy 'MatchAlgorithm'.
- int getFirstTeam()const - sprawdza czy zawodnik jest w pierwszym składzie.
- QPixmap getFoto()const - zwraca fotografię zawodnika.
- QString getFotoPatch()const - zwraca ścieżkę do fotografii zawodnika.
- void setId(int) - ustawia numer indeksu gracza.
- void setFname(QString) - ustawia imię gracza.
- void setLname(QString) - ustawia nazwisko gracza.
- void setNationality(QString) - – ustawia narodowość gracza.
- void setPosition(QString) - ustawia pozycję gracza.
- void setAge(int) - ustawia wiek gracza.
- void setGkeeper(double) - ustawia umiejętności bramkarskie gracza.
- void setDefence(double) - ustawia umiejętności gry zawodnika w defensywie.
- void setMidfield(double) - ustawia umiejętności gry zawodnika w pomocy.
- void setAttack(double) - ustawia umiejętności gry zawodnika w ataku.
- void setTackle(double) - ustawia umiejętności zawodnika w odbiorze piłki.
- void setTechnik(double) - ustawia umiejętności techniczne zawodnika.
- void setShot(double) - ustawia umiejętności strzeleckie zawodnika.
- void setPas(double) - ustawia umiejętności w celności podań zawodnika.
- void setHeader(double) - ustawia umiejętności gry głową zawodnika.
- void setSpeed(double) - ustawia szybkość zawodnika.
- void setEndurance(double) - ustawia wytrzymałość zawodnika.
- void setGoal() - inkrementuje ilość goli zawodnika.
- void setGoal2(int) – ustawia ilość goli zawodnika.
- void setStamina(double) - ustawia energię gracza.

- void setFirstTeam(int) - ustawia zawodnika w pierwszym składzie lub jako rezerwowego.
- void setFoto(QPixmap) – ustawia fotografię zawodnika.
- void setFotoPath(QString) – ustawia ścieżkę do fotografii zawodnika.

Klasa *PlayerSupportFunction* wykorzystywana jest jako klasa pomocnicza dla klasy 'Player' w celu wyliczania parametrów zawodników. Składa się z 11 metod wyliczających umiejętności zawodników.

- void setSupportFunction() – inicjalizuje zmienne klasy zmiennymi klasy 'Player'. Użyta została technika 'wstrzykiwania zależności'.
- double setGoalKeeperRatings() - oblicza umiejętności na pozycji bramkarza.
- double setCBRatings() - oblicza umiejętności na pozycji środkowego obrońcy.
- double setRBLBRatings() - oblicza umiejętności na pozycji prawego lub lewego obrońcy.
- double setDMRatings() - oblicza umiejętności na pozycji defensywnego pomocnika.
- double setCMRatings( ) - oblicza umiejętności na pozycji środkowego pomocnika.
- double setRMLMRatings() - oblicza umiejętności na pozycji prawego lub lewego pomocnika.
- double setAMRatings() - oblicza umiejętności na pozycji ofensywnego pomocnika.
- double setRWLWRatings() - oblicza umiejętności na pozycji prawy lub lewego skrzydłowego.
- double setSSRatings() - oblicza umiejętności na pozycji cofniętego napastnika.
- double setCFRatings() - oblicza umiejętności na pozycji napastnika.

Klasa *Serialization* służy do serializacji, deserializacji oraz do wczytywania początkowych ustawień aplikacji. Składa się z 10 metod: 5 do serializacji oraz 5 do deserializacji.

- void saveSquad(const QString &, QVector<Player> &) - metoda pobiera ścieżkę do zapisu danych oraz kontener obiektów klasy 'Player' w celu serializacji ich we wskazanym miejscu.
- void loadSquad(const QString &load_path, QVector<Player> &player) - metoda pobiera ścieżkę do odczytu danych oraz kontener klas 'Player' w celu deserializacji danych ze wskazanego miejsca.
- void loadSquadGame(const QString &, QVector<QVector<Player> > &, QList<Team> &) - metoda pobiera ścieżkę do odczytu danych, kontenera

kontenerów klasy ‘Player’ oraz listę klas ‘Team’ w celu deserializacji danych ze wskazanego miejsca.

- void saveSquadGame(const QString &, QList<Team>) - metoda pobiera ścieżkę do zapisu danych oraz listę obiektów klasy ‘Team’ w celu serializacji ich we wskazanym miejscu.
- void saveTeam(const QString &, QList<Team>) - metoda pobiera ścieżkę do zapisu danych oraz listę obiektów klasy ‘Team’ w celu serializacji ich we wskazanym miejscu.
- void loadTeam(const QString &, QList<Team> &) - metoda pobiera ścieżkę do odczytu danych oraz liste klas ‘Team’ w celu deserializacji danych ze wskazanego miejsca.
- void saveLeague(const QString &, League *) - metoda pobiera ścieżkę do zapisu danych oraz wskaźnik na obiekt klasy ‘League’ w celu serializacji danych.
- void loadLeague(const QString &, League *) - metoda pobiera ścieżkę do odczytu danych oraz wskaźnik na obiekt klasy ‘League’ w celu deserialzacji danych.
- void saveUserData(const QString &, UserData *) - metoda pobiera ścieżkę do zapisu danych oraz wskaźnik na obiekt klasy ‘UserData’ w celu serializacji danych.
- void loadUserData(const QString &, UserData *) - metoda pobiera ścieżkę do odczytu danych oraz wskaźnik na obiekt klasy ‘UserData’ w celu deserialzacji danych.

Klasa *SetMultimedia* - przeznaczona jest do zapełniania innych kontenerów danymi, które wykorzystywane są w późniejszym działaniu aplikacji. Składa się z 7 metod ustawiających i 6 metod zwracających:

- void setFoto() – ładuje fotografie zawodników
- void setTeamTshirt() – ładuje uniformy drużyn
- void setTeamIcon() – ładuje herby drużyn
- void setMusicPlayersButton() – ładuje przyciski odtwarzacza muzycznego
- void setSavedTeamDataPath() – ładuje zapisany przez użytkownika stan aplikacji
- void setDefaultDataPath() – ładuje początkowy stan aplikacji.
- void setStadiumsData() – ładuje fotografie stadionów oraz ich dane
- QPixmap getButtons() – zwraca przycisk odtwarzacza muzycznego o podanym indeksie
- QPixmap getLogo() – zwraca logo drużyny o podanym indeksie
- QString getSavedPlayersList() -
- QString get SavedSLData() –
- QString getDefaultPlayers() –
- QString getDefaultDataTeams() –

Metody tej klasy wykorzystywane są w klasie ‘Container’. W metodach loadDefaultData, loadData oraz saveData do serializacji i deserializacji danych jak również w innych miejscach aplikacji do ustawiania plików graficznych.

Klasa *Stadium* służy do zarządzania stadionami. Składa się z 6 metod, 3 metod ustawiających oraz 3 metod zwracających.

- void setName(QString) – ustawia nazwę stadionu.
- void setCapacity(int) – ustawia pojemność stadionu.
- void (QPixmap) – ustawia fotografię stadionu.
- QString getName() – pobiera nazwę stadionu.
- int getCapacity() - zwraca pojemność stadionu.
- QString getFoto() – zwraca fotografię stadionu.

Klasa *Team* używana jest w aplikacji do tworzenia drużyn oraz zarządzania nimi. Składa się z 68 metod: 25 metod zwracających, 21 metod ustawiających oraz 22 metod zarządzających.

- QString getName()const – zwraca nazwę drużyny
- QString getCity()const – zwraca nazwę miasta.
- QString getSeparator() – zwraca separator.
- QString getShortName() - zwraca skróconą nazwę drużyny.
- int getRound()const – zwraca numer rundy.
- int getPoint()const – zwraca punkty drużyny.
- int getWin()const – zwraca zwycięstwa drużyny.
- int getDraw()const – zwraca remisy drużyny.
- int getLost()const – zwraca przegrane drużyny.
- int getGoalScored()cons – zwraca gole zdobyte przez drużynę.
- int getGoalLost()const – zwraca gole stracone przez drużynę.
- int getDiffer()const – zwraca różnicę goli zdobytch i straconych przez drużynę.
- int getStrength()const – zwraca siłę drużyny.
- int getFormation()const – zwraca formację drużyny.
- int getScore()const – zwraca wynik drużyny.
- int getTeamId()const – zwraca id drużyny.
- int getPressing()const – zwraca poziom presingu drużyny.
- int getTackle()const – zwraca umiejętności drużyny w odbiorze piłki.
- int getAttackLevel()const – zwraca poziom ataku drużyny.
- QVector<Player>getPlayer()const – zwraca zawodników drużyny.
- QVector<Player>&getPlayer() – zwraca zawodników drużyny.
- Stadium &getStadium() – zwraca stadion drużyny.
- QPixmap getLogo() – zwraca logo drużyny.
- QPixmap getTshirt() – zwraca logo uniformów drużyny.
- QList<QString> getStrikerList() - zwraca listę strzelców z ostatniego meczu.

- void setName(QString&) – ustawia nazwę drużyny.
- void setCity(QString&) – ustawia miasto drużyny.
- void setSeparator() – ustawia separator.
- void setRound(int) – inkrementuje rundę.
- void setPoint(int) – dodaje punkty drużyny.
- void setWin() – inkrementuje zwycięstwa drużyny.
- void setDraw() – inkrementuje remisy drużyny.
- void setLost() - inkrementuje porażki drużyny.
- void setGoalScored(int) – dodaje gole do goli zdobytych drużyny.
- void setGoalLost(int) – dodaje gole do goli straconych drużyny.
- void setDiffer(int) – ustawia różnicę goli zdobytych i straconych przez drużynę.
- void setStrength(int) – ustawia siłe drużyny.
- void setFormation(int) – ustawia formację drużyny.
- void setScore(int) – ustawia wynik drużyny.
- void setPressing(int) – ustawia presing drużyny.
- void setAttackLevel(int) – ustawia poziom ataku drużyny.
- void setPlayer(const Player &) – dodaje zawodnika do listy zawodników.
- void setStadium(const Stadium &) - dodaje stadion do drużyny.
- void setLogo(QPixmap) – ustawia logo drużyny.
- void setTshirt(QPixmap) – ustawia uniformy drużyny.
- void setStrikerList(const QString) – dodaje zawodnika do lisy strzelców ostatniego meczu.
- int teamStrengthEnergy(QVector<Player>&, int) – oblicza siłę drużyny na podstawie wybranej formacji.
- int teamGoalkeeper(QVector<Player>&) – oblicza siłę bramkarza.
- int teamDefence(QVector<Player>&, int) – oblicza siłę obrony drużyny na podstawie wybranej formacji.
- int teamMidfield(QVector<Player>&, int) – oblicza siłę pomocy drużyny na podstawie wybranej formacji.
- int teamAttack(QVector<Player>&, int) – oblicza siłę ataku drużyny na podstawie wybranej drużyny.
- int teamSpeed(QVector<Player>&) – oblicza średnią prędkość drużyny.
- int teamTechnick(QVector<Player>&) – oblicza umiejętności techniczne drużyny.
- int teamDefensive(QVector<Player>&, int) – oblicza umiejętności drużyny w defensywie.
- int teamOffensive(QVector<Player>&, int) – oblicza umiejętności drużyny w ofensywie.
- int teamStrengthEnergy(QVector<Player>&, int) – oblicza siłę drużyny na podstawie formacji oraz utraconej energii zawodników w trakcie meczu.
- int teamGoalkeeperEnergy(QVector<Player>&) – oblicza umiejętności bramkarza na podstawie utraconej energii w trakcie meczu.

- int teamDefenceEnergy(QVector<Player>&, int) – oblicza siłę drużyny w defensywie na podstawie formacji oraz utraconej energii.
- int teamMidfieldEnergy(QVector<Player>&, int) – oblicza siłę drużyny w pomocy na podstawie formacji oraz utraconej energii.
- int teamAttackEnergy(QVector<Player>&, int) – oblicza siłę ataku w pomocy na podstawie formacji oraz utraconej energii.
- int teamSpeedEnergy(QVector<Player>&) – oblicza prędkość drużyny na podstawie utraconej energii.
- int teamTechnickEnergy(QVector<Player>&) – oblicza umiejętności techniczne drużyny na podstawie utraconej energii.
- int teamTackleEnergy(QVector<Player>&, double) – oblicza umiejętności w odbiorze piłki drużyny na podstawie utraconej energii oraz pressingu.
- int teamDefensiveEnergy(QVector<Player>&, int) – oblicza umiejętności gry w defensywie drużyny na podstawie utraconej energii.
- int teamOffensiveEnergy(QVector<Player>&, int) – oblicza umiejętności gry w ofensywie drużyny na podstawie utraconej energii.
- int teamEnergy(QVector<Player>&) – oblicza ile energii pozostało drużynie.
- void clearStrikerList() – czyści listę strzelców po zakończonym meczu.
- void clearTeamStats(QList<Team>&) – czyści statystyki drużyny po zakończonym sezzonie.

Klasa *UserData* zarządza danymi użytkownika. Składa się z 10 metod, 5 metod zwracających oraz 5 metod ustawiających.

- Qstring getFname() – zwraca imię użytkownika.
- Qstring getLName() – zwraca nazwisko użytkownika.
- Qstring getNationality() – zwraca narodowość użytkownika.
- Int getAge() – zwraca wiek użytkownika.
- Int getId() – zwraca id drużyny użytkownika
- void setFname(QString) – ustawia imię użytkownika.
- void setLname(QString) – ustawia nazwisko użytkownika.
- void setNationality(QString) – ustawia narodowość użytkownika.
- void setAge(int) – ustawia wiek użytkownika.
- void setId(int) – ustawia id drużyny użytkownika.

Klasa *Versus* używana jest do rozlosowania spotkań pomiędzy wszystkimi 16 drużynami metodą każdy z każdym, mecz i rewanż co daje 30 spotkań dla każdej z 16 drużyn. W jej skład wchodzi 5 metod: 3 metody zarządzające i 2 metody zwracające.

- void pairList() – metoda ta wypełnia listę parami zespołów które będą rywalizowały ze sobą. Ustawiane jest 240 par na cały sezon – 8 par na jedną z 30 kolejek.
- void pairProgress() – zwiększa pozycję każdej z par o osiem, tak aby pierwsza para z 1 kolejki była pierwszą parą w drugiej kolejce.

- void setTeamPairs() – przypisuje drużynę do odpowiednie pary i pozycji w parze.
- int getPair(int) – zwraca parę o wskazanym numerze.
- int getTeamsPaired(int) – zwraca pozycję drużyny z wskazanej pary.

4.4.3. Klasy Kontrolerów

Klasy kontrolerów odpowiedzialne są za sterownie aplikacją, pełnią one funkcje mediatorów pomiędzy klasami modeli służących do zarządzania logiką aplikacji a klasami widoku, które to z kolei wyświetlają te przetworzone dane. Każda klasa kontrolera posiada wskaźnik do swojej klasy widoku, który umożliwia manipulowanie jej kontrolkami i przekazywanie danych z klas modelowych na widok.(UI::MainMenu *Ui) Dodatkowo klasy kontrolerów posiadają identyfikatory signals i slots, których metody zapewniają komunikację pomiędzy innymi kontrolerami.

Klasa *About* – w aplikacji pełni tylko funkcję wyświetlania informacji o twórcy aplikacji. Posiada tylko jedną metodę.

- void setGraphics() – ustawia przy pomocy wskaźnika na widok kolor tła poszczególnych banerów na przeźroczysty.

Klasa ‘AddPlayer’ - jest częścią edytora, w który wyposażona jest aplikacja. Służy do dodawania nowych graczy do wybranej drużyny. Posiada 3 metody i 2 sloty.

- void setGraphic() – metoda, ustawia style graficzne poszczególnych elementów, które znajdują się na widoku.
- void squad() – metoda, odpowiedzialna za przekazanie danych z modelu zawierającego informacje o drużynie do widoku.
- void setSpinBoxGraphics() – metoda, ustawia grafikę na kontrolkach typu ‘QSpinBox’.
- void on_editchooseteam_clicked() – slot, odpowiedzialny za powrót do poprzedniego menu.
- void on_add_clicked() – slot, odpowiadyialnz za dodanie nowego gracza do wybranego zespołu. Po wypełnieniu pól po prawej stronie formularza i kliknięciu przycisku, zawodnik dodawany jest do tabeli znajdującej się w centralnej części ekranu. Następnie jego parametry zapisane są do pliku dzięki wykorzystaniu metody saveSquad klasy ‘Serialization’.

Klasa *ChoseTeam* – jest to menu wyboru odpowiedzialne za wyświetlenie informacji o wskazanej drużynie oraz za zapisanie informacji o wyborze drużyny jakiej dokonał użytkownik. Posiada 4 metody, 17 slotów oraz 1 zdefiniowany sygnał.

- void setImages() – metoda, ustawia ikony drużyn na przyciskach. Wykorzystana jest tutaj funkcja setIcon() klasy ‘Functions’.
- void mouseTracking() – metoda, włącza śledzenie wskaźnika myszy na przyciskach. Działanie to umożliwia wyświetlanie innych okien dialogowych kiedy ustawimy kursor myszy na wybranym przycisku.

- void setAnimation() – metoda, ustawia w odpowiedniej pozycji okna dialogowe odpowiedzialne za prezentacje informacji o drużynie.
- void setUserData(int) - metoda, zapisuje id wybranej drużyny przez użytkownika oraz wywołuje sygnał dialogClickedGo(), który jest odpowiedzialny za przejście do kolejnego widoku.
- highLight() - slot którego zadaniem jest wyświetlenie okien dialogowych odpowiedzialne za prezentacje informacji o drużynie po najechaniu kursorem myszy na wybrane logo drużyny. Wykorzystuje metodę setAniomation().
- pozostałe 16 slotów odpowiedzialne są za przekazanie id drużyny wybranej przez użytkownika i przekazanie go do metody setUserData().

Klasa *ContextMenu* jest używana w aplikacji jako okno dialogowe w trakcie działania aplikacji służące do wyświetlania informacji dla użytkownika. Nie posiada żadnych metod, pobiera przez konstruktor ciąg znaków, które następnie wyświetla w postaci komunikatu.

Klasa *Data* jest to menu wyboru, które umożliwia wyświetlenie listy najskuteczniejszych strzelców oraz najmocniejszych drużyn. Posiada 1 metodę oraz 2 sloty.

- void setGraphics() – metoda, ustawia style graficzne poszczególnych elementów, które znajdują się na widoku.
- void on_topscorers_clicked() – slot, przełącza do widoku ‘TopScorer’.
- void on_stats_clicked() – slot, przełącza do widoku ‘Stats’.

Klasa *Edit* jest klasą wyboru wbudowanego edytora pozwalającą na poruszanie się pomiędzy trzema widokami ‘AddPlayer’, ‘EditPlayer’ oraz ‘RemovePlayer’, które służą kolejno do dodawania, edytowania bądź usuwania gracza. Posiada 5 slotów oraz jeden zdefiniowany sygnał.

- void on_option_clicked() – slot wywołuje sygnał dialogClicked() który przenosi użytkownika do widoku ‘Options’.
- void on_add_clicked() – slot przenosi użytkownika do widoku ‘RemovePlayer’. przenosi użytkownika do widoku ‘AddPlayer’.
- void on_remove_clicked() – slot przenosi użytkownika do widoku ‘RemovePlayer’.
- void on_edit_clicked() – slot przenosi użytkownika do widoku ‘EditPlayer’.
- void backToMainMenu() – slot przenosi użytkownika do widoku ‘Options’
- void dialogClicked() – sygnał wywoływany w slocie on_option_clicked() który przenosi użytkownika do widoku ‘Options’.

Klasa *EditPlayer* jest cześcią edytora w którego wyposażona jest aplikacja. Służy do edytowania graczy wybranej drużyny. Posiada 2 metody, 6 slotów oraz 1 zdefiniowany sygnał.

- void SetGraphics() – metoda, ustawia style graficzne poszczególnych elementów, które znajdują się na widoku.
- void squad() – metoda, odpowiedzialna za przekazanie danych z modelu zawierającego informacje o drużynie i wyświetlenie w tabeli znajdującej się w centralnej części widoku.
- void on_editchooseteam_clicked() – slot, przenosi użytkownika do widoku 'EditChooseTeam'.
- void on_add_clicked() – slot, odpowiada za edycje wybranego gracza z wybranego zespołu. Po wypełnieniu pól po prawej stronie formularza i kliknięciu przycisku, zawodnik dodawany jest do tabeli znajdującej się w centralnej części ekranu. Następnie jego parametry zapisane są do pliku dzięki wykorzystaniu metody saveSquad klasy 'Serialization'.
- void edit() – slot, odpowiedzialny jest za wyświetlenie danych z modelu na widok w polach edycyjnych (QLabel) oraz sliderach (QSlider) po kliknięciu odpowiedniego indeksu zawodnika znajdującego się w tabeli.
- void setSpinBoxGraphics() – slot, ustawia grafikę na kontrolkach typu 'QSpinBox'.
- void take() – slot, pobiera indeks zawodnika klikniętego w tabeli ze składem
- void put() – slot, zamienia dwóch wybranych zawodników miejscami w składzie.

Klasa *Fixture* odpowiedzialna jest za wyświetlanie kolejnych kolejek meczowych wraz z parami zespołów które będą między sobą rozgrywać mecz. Posiada 2 metody oraz 2 sloty.

- void setGraphic() – metoda, ustawia style graficzne poszczególnych elementów, które znajdują się na widoku.
- void setFixture() – metoda, wykorzystuję klasę modelu 'Versus' do rozlosowania kolejnej kolejki sezonu i wyświetlenie danych na widok.
- void next() – slot służy do zwiększania numeru kolejki i wyświetlenia jej na widoku.
- void previous() - slot służy do zmniejszania numeru kolejki i wyświetlenia jej na widoku.

Klasa *GameDialog* – jest to menu wyboru odpowiedzialne za wyświetlenie między innymi takich informacji jak: skład drużyny wybranej przez użytkownika, podgląd innych drużyn, statystyki czy terminarz rozgrywek. Posiada również

wbudowany w widok odtwarzacz muzyczny. Składa się z 2 metod, 18 slotów oraz 1 zdefiniowany sygnał.

- void setIco() – metoda, ustawia grafikę przycisków na widoku oraz ikonki odtwarzacza muzycznego.
- void mouseTracking() – metoda, włącza śledzenie pozycji myszy wszystkich przycisków na widoku.
- void on_table_clicked() – slot, przełącza na widok 'Table'.
- void on_squad_clicked() – slot, przełącza na widok 'UserSquad'
- void on_exit_clicked() – slot, kończy działanie aplikacji.
- void on_teams_clicked() – slot, przełącza na widok 'SelectShowSquad'.
- void on_stats_clicked() – slot, przełącza na widok 'Data'.
- void on_fixture_clicked() – slot przełącza na widok 'Fixture'.
- void on_nextday_clicked() – slot przełacza na widok 'Match'.
- void on_mainmenu_clicked() slot – wraca do widoku początkowego 'MainMenu'.
- void on_save_clicked() – slot zapisuje stan aplikacji.
- void on_load_clicked() – slot przywraca stan aplikacji.
- void highLight() - slot podświetla przyciski kiedy znajduje się nad nim kursor myszy.
- void start() – slot włącza utwór w odtwarzaczu muzycznym.
- void stop() - slot wyłacza utwór w odtwarzaczu muzycznym.
- void pause() – slot wstrzymuje odtwarzanie muzyki w odtwarzaczu muzycznym.
- void forward() – slot przełacza o jeden utwór do przodu.
- void previous() - slot przełacza o jeden utwór do tyłu.
- void volumeUp() – slot podgłaśnia utwór muzyczny o 10%.
- void volumeDown() – slot zcisza utwór muzyczny o 10%.

Klasa *Intro* pełni funkcję planszy startowej wyświetlając obrazek startowy. Posiada tylko 1 slot.

- void on_go_clicked() – slot, przełącza na widoku 'MainMenu'.

Klasa *LiveMatch* odpowiedzialna jest za prezentacje rozgrywanego meczu w trybie rzeczywistym. Jej zadaniem jest umożliwienie użytkownikowi interakcji (wraz z klasą *UserSquad*) z zespołem poprzez zmianę ustawienia drużyny, zwiększanie pressingu czy zmianę zmęczonych zawodników. Jednocześnie wyświetla szereg różnych statystyk dotyczących przebiegu meczu takich jak, czas, ilość oddanych strzałów, strzały celne czy posiadanie piłki. Posiada też specjalne pole tekstowe na którym wyświetlane są komentarze z przebiegu meczu. Składa się z 10 metod oraz 3 slotów.

- matchProgress() – metoda odpowiada za przebieg meczu. W pętli składającej się z 90 kroków wywołuje metody z klasy ‘MatchAlgorithms’ odpowiedzialne za przebieg meczu.
- setParameters() – metoda, ustawia zegar na widoku oraz wywołuję metodę possessionStatusBar().
- void setGraphic() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void displayMatchStats() – metoda, wyświetlająca statystyki z meczu pochodzące z klas modelu na widok.
- void shotsStatusBar() – metoda, wyświetla ilość oddanych strzałów drużyn rozgrywających mecz.
- void shotsOnTargetStatusBar() – metoda, wyświetla ilość strzałów celnych drużyn rozgrywających mecz.
- void possessionStatusBar() – metoda, oblicza i wyświetla na widok posiadanie piłki.
- void computerStrategy() – metoda, odpowiedzialna za strategię komputera podczas meczu. Wykorzystuje metody klasy ‘ArtificialInteligence’.
- void comentary() – metoda, odpowiedzialna za wyświetlanie komentarzy na widoku.
- void on_back_clicked() – slot, przełącza do widoku ‘MatchInterface’.
- void on_pause_clicked() – slot ,wstrzymuje mecz.
- void on_squad_clicked() – slot, przełącza do widoku ‘UserSquad’

Klasa *MainMenu* jest zarówno główną klasą aplikacji jak również menu wyboru. Zawiera obiekt klasy *QStackedWidget* który zawiera referencje do 5 kolejnych widoków oraz 4 zdefiniowane połączenia umożliwiające pośrednie przełączanie między widokami. Posiada 2 metody oraz 11 slotów.

- void setGraphics() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void mouseTracking() – metoda, włącza śledzenie pozycji myszy wszystkich przycisków na widoku.
- void on_mainMenu_clicked() – slot, przełacza z powrotem do widoku ‘MainMenu’.
- void on_newGame_clicked() – slot, przełącza do widoku ‘ChooseTeam’.
- void on_abaut_clicked() – slot, przełącza do widoku ‘About’.
- void on_option_clicked() – przełacza do widoku ‘Option’.
- void on_load_clicked() – slot, wczytuje dane początkowe I przełącza do widoku ‘GameMenu’.
- void highLight() – slot, podświetla przyciski kiedy znajduje się nad nim kursor myszy.
- void on_quit_clicked() – slot, kończy działanie aplikacji.
- void backToMainMenu() – slot, umożliwiający powrót z widoku ‘GameMenu’ do widoku ‘MainMenu’.

- void backToChoseTeam() – slot, umożliwiający powrót z widoku ‘UserDataDialog’ do widoku ‘ChooseTeam’.
- void goToGameDialog() – slot, umożliwiający przejście z widoku ‘UserDataDialog’ do widoku ‘GameDialog’.
- void goToUserDataDialog() - slot, umożliwiający przejście z widoku ‘ChooseTeam’ do widoku ‘UserDataDialog’.

Klasa *Match* jest rodzajem interaktywnego kalendarza wykorzystywanego do przechodzenia przez kolejne dni sezonu. Składa się z 2 metod oraz 1 slota.

- void setGraphics() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void setting() – metoda, wykorzystuję klasę ‘Versus’ odpowiedzialną za rozlosowanie kolejnych kolejek meczowych oraz wyświetlenie jej na widok.
- void on_nextday_clicked() - slot, wykorzystuje klasę ‘League’ do kontrolowania dni meczowych oraz sterowaniem przebiegiem całego sezonu jak również klasę ‘Player’ odpowiedzialną za regeneracje zawodników poszczególnych drużyn

Klasa *MatchInterface* – służy do wyświetlania wyników rozegranych spotkań w danej kolejce. Dodatkowo wyświetla nazwiska strzelców oraz minutę w której został zdobyty gol. Składa się z 3 metod oraz 1 slota.

- void setGraphics() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void setting() – metoda, wykorzystuje klasę ‘Versus’ oraz ‘League’ do rozlosowania par zespołów w danej rundzie oraz wyświetlenie ich na widoku.
- void setData(int, int, QLabel *, QLabel *, QListWidget *, QListWidget *) – metoda wypisuje strzelców bramek na widok oraz wyświetla wyniki drużyn.
- void confrontation(int, int) - metoda, odpowiedzialna za rozgrywanie meczów. Jej zadaniem jest wyodrębnienie meczów rozgrywanych pomiędzy AI a użytkownikiem vs AI. Kiedy mecz rozgrywa się pomiędzy AI używana jest klasa modelu ‘MatchSimulation’ natomiast kiedy mecz rozgrywa użytkownik wywoływany jest kontroler ‘LiveMatch’ wraz z widokiem.
- void on_back_clicked() – slot, powraca do porzedniego widoku.

Klasa *Options* jest menu wyboru. Składa się z 3 slotów oraz posiada 1 zdefiniowany sygnał. Zawiera również obiekt klasy QStackedWidget, który posiad referencję do klasy ‘Edit’.

- void on_mainmenu_clicked() – slot, wykorzystuje sygnał dialogClicked() by powrócić do widoku ‘MainMenu’.
- void on_edit_clicked() – slot, przełącz do widoku ‘Edit’.
- void bakToMainMenu() – slot, umożliwiający powrót z widoku ‘Edit’ do ‘Options’.

Klasa *RemovePlayer* jest cześcią edytora w którego wyposażona jest aplikacja. Służy do usuwania graczy wybranej drużyny. Posiada 2 metody, 2 sloty oraz 1 zdefiniowany sygnał.

- void setGraphics() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void squad() – metoda, odpowiedzialna za przekazanie danych z modelu zawierającego informacje o drużynie i wyświetlenie w tabeli znajdującej się w centralnej części widoku.
- void on_editchooseteam_clicked() – slo,t wykorzystuje sygnał dialogClicked() aby wrócić do widoku ‘EditChooseTeam’.
- void on_remove_clicked() – slot, usuwa wybranego zawodnika i zapisuje zmiany.

Klasa *SelectShowSquad* – jest to menu wyboru odpowiedzialne za wyświetlenie informacji o wskazanej drużynie. Posiada 1 metodę oraz 16 slotów.

- void setImages () – metoda, ustawia grafikę elementów znajdujących się na widoku.
- 16 slotów odpowiedzialnych jest za przekazanie informacji do klasy ‘ShowSquads’ w celu wyświetlenia informacji o wybranej drużyny.

Klasa *ShowPlayerStats* jest to okno dialogowe, które wyświetla dane zawodników na widoku ‘ChoseTeam’ po skierowaniu kursora na odpowiednią ikonę. Posiada 3 metody.

- void setPlayerId(int) – metoda pobiera id wskazanej drużyny.
- void setGraphics() - metoda ustawia grafikę elementów znajdujących się na widoku.
- void showStats() – metoda wyświetla podstawowe dane o zawodnikach wskazanej drużyny w formie tabeli.

Klasa *ShowsSquad* - służy do wyświetlania informacji wybranej drużyny. Składa się z 4 metod oraz 1 slotu.

- void setGraphic() – metoda, ustawia grafikę elementów znajdujących się na widoku.

- void squad() – metoda, wyświetla informacje o składzie drużyny w formie tabeli.
- void formationId(int) – metoda, ustawia kolory formacji.
- void updateTeamStats() – metoda, która pobiera informacje o drużynie z modelu w celu wyświetlenia ich na widoku.
- void personalData() – slot, który umożliwia wyświetlanie szczegółowych danych wybranego zawodnika.

Klasa *ShowTeamSquad* jest to okno dialogowe, które wyświetla dane drużyny na widoku 'ChoseTeam' po skierowaniu kursora na odpowiednią ikonę. Posiada 3 metody.

- void setTeamId(int) – metoda, pobiera id wskazanej drużyny.
- void setGraphics() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void showStats() – metoda, wyświetla podstawowe dane o drużynie wskazanej na widoku 'ChoseTeam'.

Klasa *Stats* – wyświetla listę najmocniejszych drużyn posortowanych malejąco. Składa się z 2 metod.

- void setGraphic() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void table() – metoda, wyświetla nazwy drużyn oraz ich siłę. Do obliczenia siły drużyny brani są zawodnicy z pierwszego składu, nie bierze pod uwagę zmęczenia zawodników.

Klasa *Table* jest odpowiedzialna za wyświetlanie aktualnego dorobku punktowego i bramkowego oraz innych statystyk wszystkich drużyn. Sklada się z 3 metod i 1 predykatu.

- void setGraphic() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void table() – metoda, wyświetla klasyfikacje wszystkich drużyn z ligi: herb, nazwa drużyny, runda, zwycięstwa, remisy, porażki, bramki zdobyte, bramki stracone, różnice bramek, punkty.
- void setLogoAtTable(QList<Team>&) – metoda, ustawia herby wszystkich drużyn z tabeli.
- worseTeam(const Team &, const Team &) – predykat, wykorzystywany do sortowania drużyn w tabeli. Do sortowania brane są w pierwszej kolejności punkty drużyny, następnie różnica bramek i w końcu bramki zdobyte.

Klasa *TopScorer* wyświetla listę 10 najlepszych strzelców posortowaną malejąco. Składa się z 3 metod oraz 1 predykatu.

- void setGraphic() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void table() – metoda, wyświetlająca listę 10 najlepszych strzelców.
- int matcher(const Player&) – metoda, paruje zawodników z ich drużynami.
- int findeScorer(const Player &) – predykat, odpowiedzialny za znalezienie ilości zawodników którzy zdobyli przynajmniej jedną bramkę.

Klasa *UserDataDialog* odpowiedzialna jest za zapisanie informacji o użytkowniku. Składa się z 1 metody, 2 slotów oraz 2 zdefiniowanych sygnałów.

- void setGraphic() - metoda ustawia grafikę elementów znajdujących się na widoku.
- void on_play_clicked() – slot zapisuje dane użytkownika i wykorzystując sygnał dialogClickedGo przechodzi do widoku ‘GameDialog’
- void on_teamchoice_clicked() – slot wykorzystuje sygnał dialogClicked wraca do widoku ‘ChooseTeam’.
- void dialogClicked() – sygnał umożliwiający powrót do widoku ‘ChooseTeam’.
- void dialogClickedGo() – sygnał umożliwiający przejście do widoku ‘GameDialog’.

Klasa *UserSquad* odpowiedzialna jest za wyświetlanie składu drużyny, formacji drużyny oraz innych statystyk. Umożliwia interakcję użytkownika z drużyną poprzez możliwość dokonywania zmian zawodników, zmianę formacji, czy też ustawienia pressingu lub sposobu atakowania drużyny. Składa się z 9 metod oraz 8 slotów.

- void setGraphic() – metoda, ustawia grafikę elementów znajdujących się na widoku.
- void squad() – metoda, wyświetla skład drużyny w formie tabeli.
- void formationPitch(int) – metoda, odpowiedzialna za graficzne rozmieszczenie zawodników na boisku.
- void formationColor(int) – metoda, ustawiająca kolory formacji w zależności od wybranej opcji.
- void updateTeamStats() – metoda, pobiera informacje o drużynie z modelu w celu wyświetlenia ich na widoku.
- void changePlayer() – metoda odpowiedzialna za zmianę zawodników miejscami w składzie drużyny.
- void highlightPlayer(QTableWidget *, QLabel *, int) metoda podświetla danego zawodnika na mapie rozmieszczenia zawodników.
- void pressingValue(int) – metoda wyświetla odpowiednią nazwę presingu w zależności od położenia suwaka.

- void attackValue(int) - metoda wyświetla odpowiednią nazwę presingu w zależności od położenia suwaka.
- void formation442() – slot ustawia atrybuty drużyny na podstawie wybranej formacji. Używa metod formationColor(), updateTeamStats() oraz squad().
- void formation433() - slot ustawia atrybuty drużyny na podstawie wybranej formacji. Używa metod formationColor(), updateTeamStats() oraz squad().
- void formation352() - slot ustawia atrybuty drużyny na podstawie wybranej formacji. Używa metod formationColor(), updateTeamStats() oraz squad().
- void take() – slot pobiera indeks zawodnika kliknięty w tabeli ze składem.
- void put() – slot zamienia dwóch wybranych zawodników miejscami w składzie.
- void personalData() - slot który umożliwia wyświetlanie szczegółowych danych wybranego zawodnika.
- void pressingControler() – slot ustawia presing drużyny no podstawie położenia suwaka.
- void attackControler() - slot ustawia atak drużyny no podstawie położenia suwaka.

4.4.4. Klasy Widoków

Klasy widoków generowane są automatycznie poprzez narzędzie *QtDesignera*. Proces ten został opisany w rozdziale *‚4.2.1. Nowy Projekt'*. Cały projekt zawiera 26 form na podstawie których zostało wygenerowane 26 widoków:

- Ui_Abaut
- Ui_AddPlayer
- UI_ChooseTeam
- UI_ContextMenu
- UI_Data
- UI_Edit
- UI_EditChooseTeam
- UI_EditPlayer
- UI_Fixture
- UI_GameDialog
- UI_Intro
- UI_LiveMatch
- UI_MainMenu
- UI_Match
- UI_MatchInterface
- UI_Options

- UI_RemovePlayer
- UI_SelectShowSquad
- UI_ShowPlayerStats
- UI_ShowSquads
- UI_ShowTeamsStats
- UI_Stats
- UI_Table
- UI_TopScorer
- UI_UserDataDialog
- UI_UserSquad

5. Prezentacja Projektu

Podczas projektowania aplikacji przyjęto jej zgodność z systemem *„Windows 7”* oraz rozdzielczością Full Hd (1920 x 1080). W przypadku wykorzystywania innych systemów lub rozdzielczości mogą wystąpić różnice w sposobie wyświetlania elementów i czcionek.

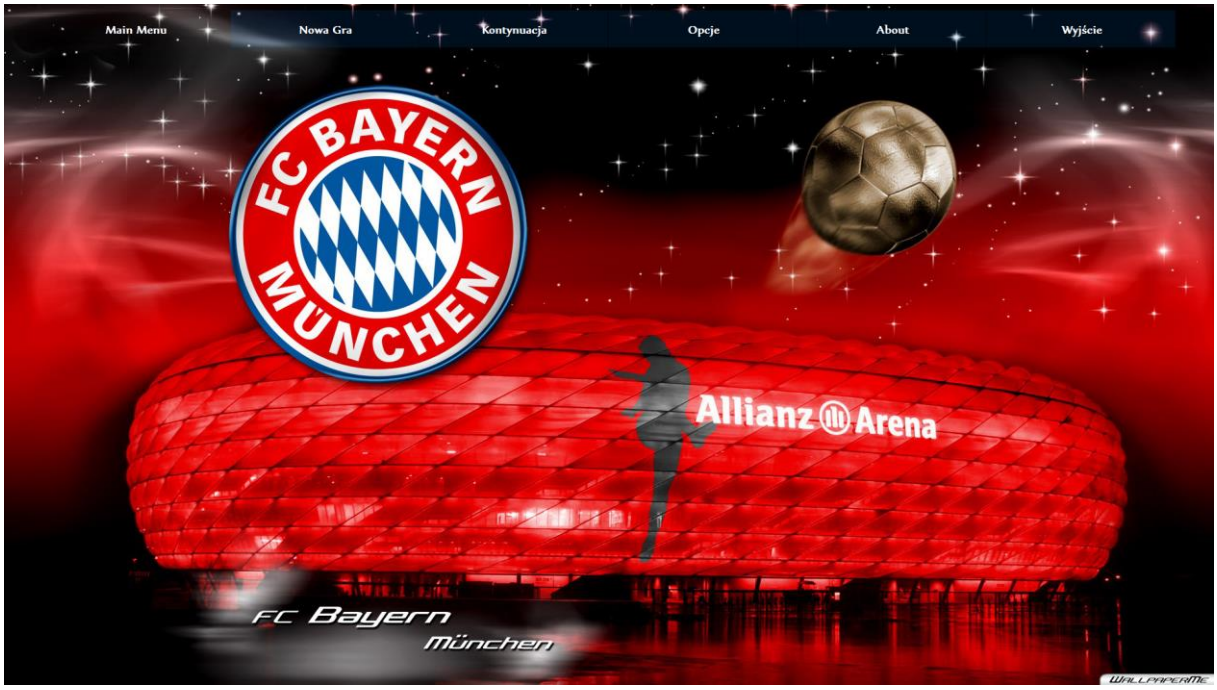
5.1. Ekran Powitalny

Po uruchomieniu aplikacji pojawia się *Ekran Powitalny* z jednym przyciskiem na środku, w dolnej części ekranu. Po naciśnięciu przycisku aplikacja przechodzi do *Menu Startowego.*

Rys. 22 Widok klasy UI_Intro

5.2. Menu Główne

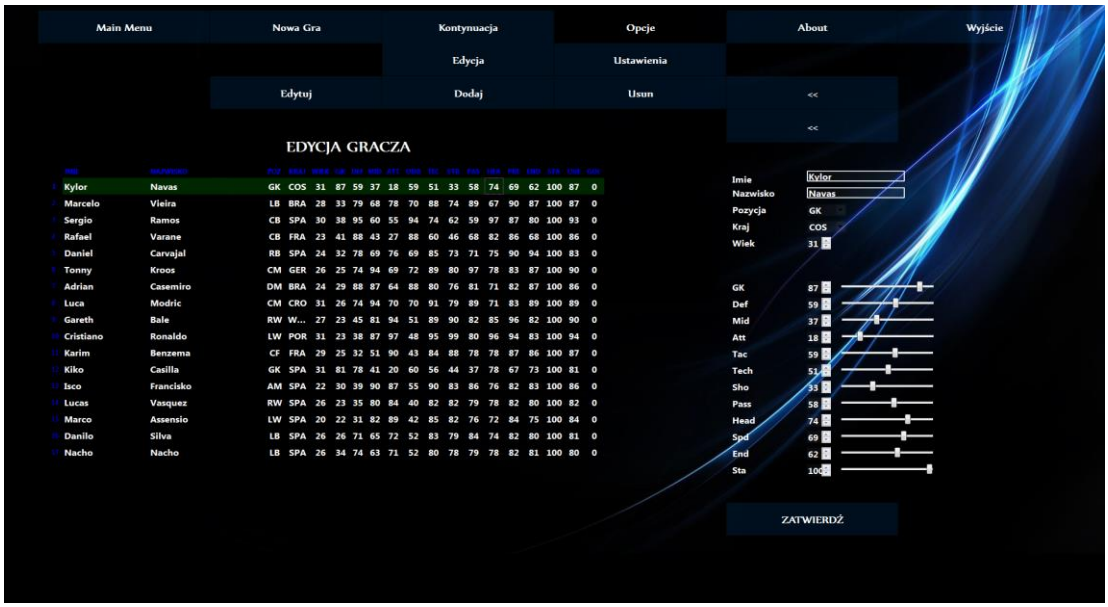
Menu Startowe zawiera pięć przycisków: *Nowa Gra*, *Kontynuacja*, *Opcje*, *About* oraz Koniec. Przycisk *Nowa Gra* uruchamia widok *Wybór Drużyny*. *Kontynuacja* umożliwia wznowienia ostatniego zapisanego stanu gry. *Opcje* przechodzi do *Menu Opcji*. *About* wyświetla informacje na temat aplikacji oraz jej twórcy. *Koniec* kończy działanie programu.



Rys. 23 Widok Klasy UI_MainMenu

5.3. Menu Opcji

Menu Opcji posiada dwa przyciski *Edycja* oraz *Ustawienia.* Po naciśnięciu przycisku *Edycja* przechodzimy do kolejnego menu *Edycja* zawierającego cztery przyciski *Edytuj, Dodaj, Usuń* oraz przycisku powrotu. Naciśnięcie każdego z nich powoduje przejście do menu *Wybór Drużyny* (widok *EditChooseTeam*) a następnie odpowiednio do widoków *EditPlayer*, *AddPlayer*, *RemovePlayer*. Widoki te odpowiedzialne są za edycję, dodawanie oraz usuwanie zawodników z poszczególnych drużyn.



Rys. 24 Widok Klasy UI_EditPlayer

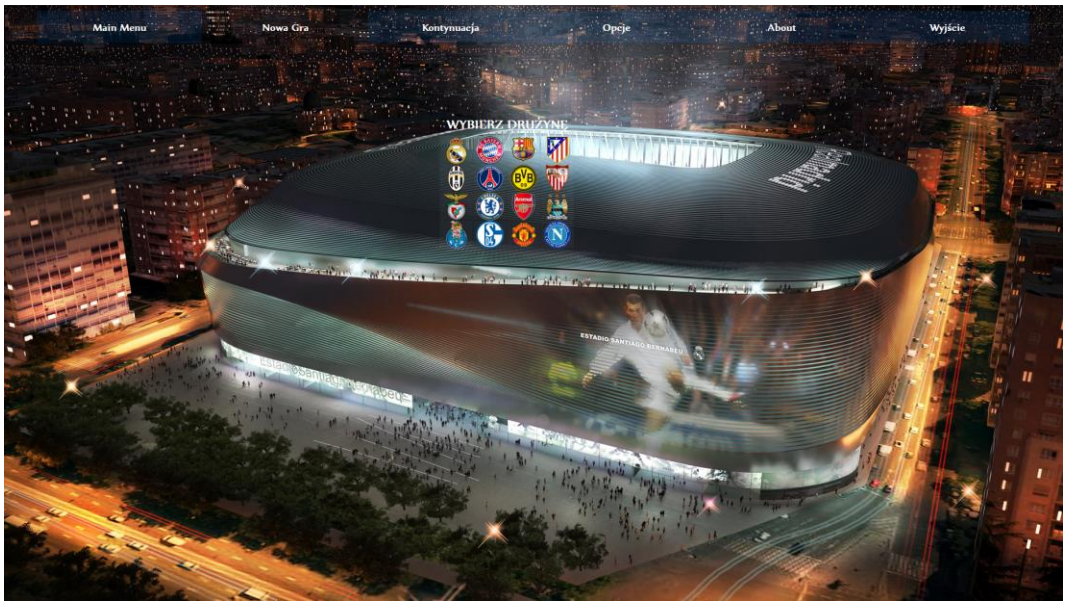
5.4. About

Widok *„About”* jest widokiem informacyjnym zawierającym w centralnej części planszę informacyjną o autorze aplikacji, uczelni oraz promotorze.

Rys. 25 Widok klasy UI_About

5.5. Menu Wybór Drużyny

W centralnej części Menu *Wybór Drużyny* znajduję się 16 herbów drużyn do wyboru. Po najechaniu kursorem na dany herb pojawiają się dwie tabelki. Tabela po lewej stronie zawiera listę zawodników wraz z ich średnią oceną umiejętności.

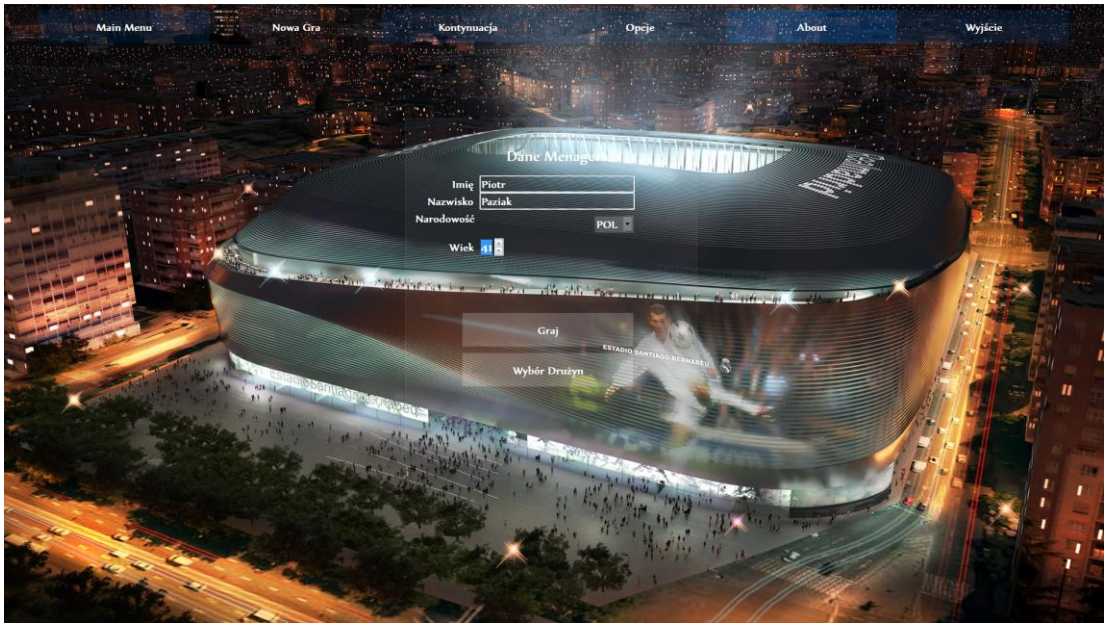


Rys. 26 Widok Klasy UI_ChooseTeam

Tabela po prawej stronie zawiera herb drużyny oraz pięć elementów opisujących siłę drużyny: umiejętności bramkarza, obrony, pomocy, ataku oraz średnią ocenę całego zespołu. Po naciśnięciu herbu wybranej drużyny użytkownik przechodzi do *Menu Użytkownika*.

5.6.. Menu Użytkownika

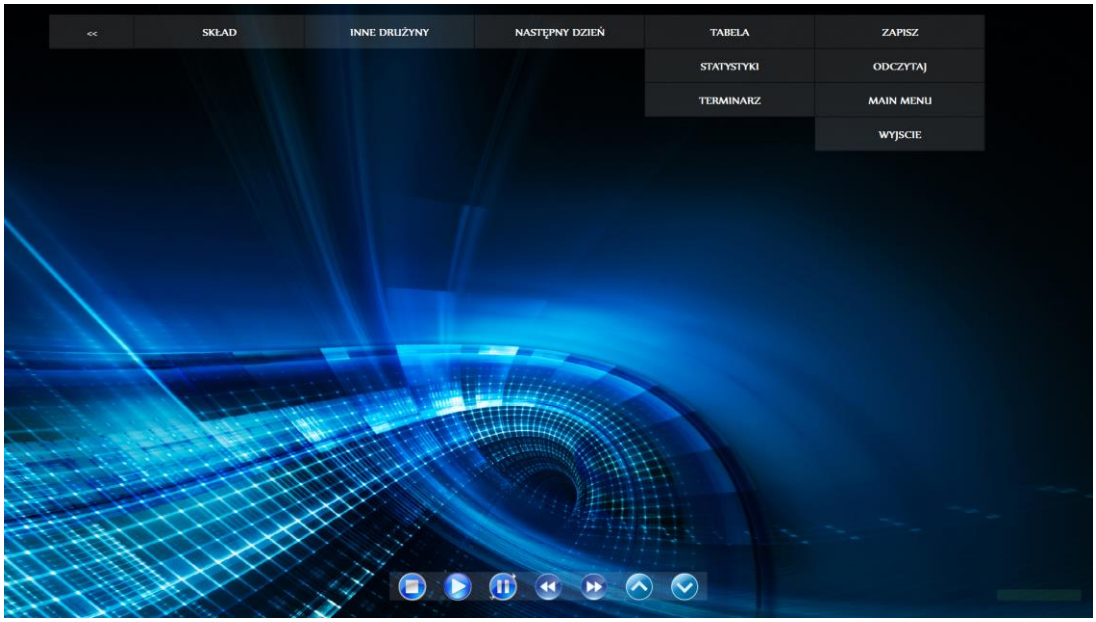
„Menu Użytkownika” to okno modalne, które umożliwia użytkownikowi wprowadzenie informacji o sobie. Składa się ono z dwóch pól typu *edit text* umożliwiających podanie swojego imienia i nazwiska, *combo box* do ustalenia narodowości oraz *spin box* do podania wieku. Ponadto okno zawiera dwa przyciski. Pierwszy *„Graj”* przenosi użytkownika do *Menu Gry*, drugi *Wybór Drużyny* cofa do menu *Wybór drużyny.*



Rys. 27 Widok Klasy UI_UserData

5.7. Menu Gry

Menu Gry w górnej części ekranu posiada jedenaście przycisków. Przyciski powrotu pozwalający powrócić do *Menu Gry* kiedy znajdujemy się w innym z powyższych menu. Przycisk *Skład* – przenosi użytkownika do menu *Zarządzanie Drużyną*. *Inne Drużyny* uruchamia menu *Wybór Drużyny*, które pozwala na wybranie drużyny oraz zobaczenie jej składu oraz innych statystyk. *Następny Dzień* – przenosi do menu *Kalendarz* odpowiadającego za zarządzaniem czasem rozgrywek. *Tabela* – wyświetla tabelę bieżących rozgrywek. *Statystyki* – umożliwia wyświetlenie statystyk drużyn oraz najlepszych strzelców.



Rys. 28 Widok UI_GameMenu

Przycisk *Terminarz* umożliwia zobaczenia kolejnych kolejek ligowych. *Zapisz* – zapisuje aktualny stan rozgrywek. *Odczytaj* – przywraca ostatnio zapisany stan rozgrywek. *Main Menu* – wraca do *Menu Główne*. *Wyjście* – kończy działanie aplikacji. W dolnej części ekranu znajdują się przyciski umożliwiające sterownie wbudowanym odtwarzaczem muzycznym oraz pasek wyświetlający poziom głośności.

5.8. Menu Skład

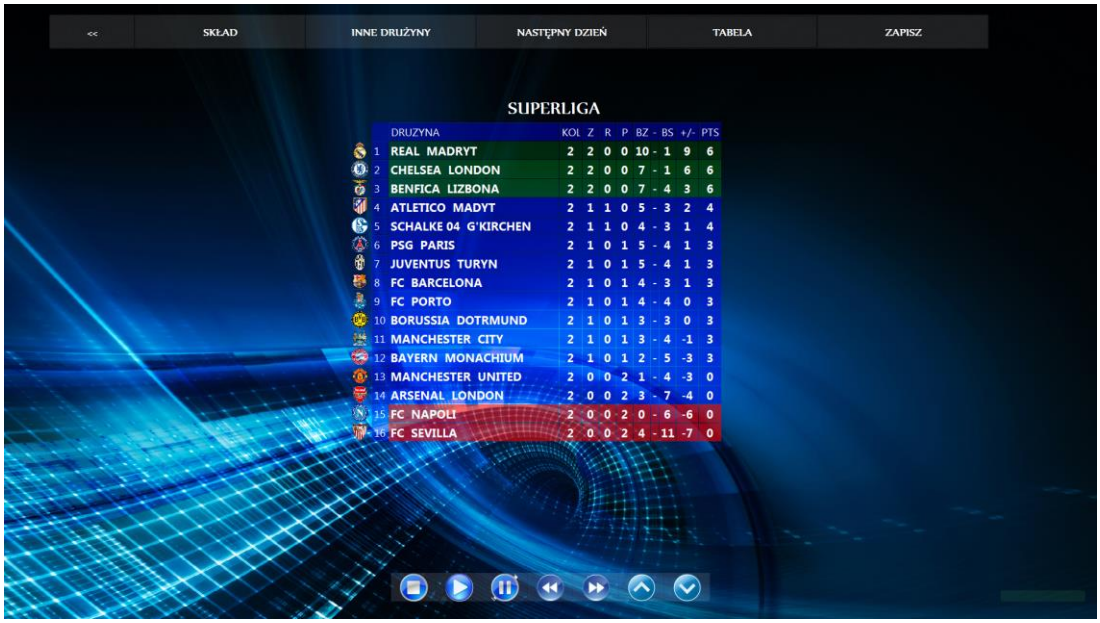
Menu Skład odpowiedzialne jest za zarządzanie drużyną. W górnej części widoku znajdują się herby, nazwa prowadzonej drużyny, imię oraz nazwisko użytkownika. Po lewej stronie usytuowana jest tabela zawierająca skład drużyny. Tabela składa się patrząc od lewej: numer zawodnika, imię, nazwisko, pozycja, kraj pochodzenia, wiek, energia zawodnika, umiejętności, oraz liczba strzelonych goli. Wiersze tabeli w zależności od formacji posiadają inne kolory: żółty - bramkarz, zielone - defensywa, niebieskie - linia pomocy oraz fioletowe – ofensywa, brak koloru – zawodnicy rezerwowi. Kolejna tabelka zawiera szczegółowe dane zawodnika, który został wybrany z tabeli składu drużyny. Obok tabelki szczegółów znajduje się panel meczowy który zawiera rozmieszczenie zawodników na boisku wraz z ich nazwiskami. W lewej części ekranu usytułowane są trzy przyciski odpowiedzialne za zmianę formacji drużyny. Poniżej przycisków formacji usytułowany jest panel wyświetlający aktualnie wybraną formację.

Rys. 29 Widok UI_UserSquad

Poniżej panelu meczowego znajdują się kolejno 3 panele. Pierwszy odpowiedzialny jest za wyświetlenie zdjęcia aktualnie wybranego zawodnika. Drugi panel zawiera dwa suwaki, które kolejno odpowiedzialne są za pressing oraz sposób ataku drużyny. Trzeci panel odpowiedzialny jest za wyświetlenie statystyk drużyny. Uwzględnia statystyki drużyny na maksymalnym poziomie (kolor biały) jak również kiedy drużyna jest zmęczona (kolor czerwony). Po prawej stronie znajduje się pasek energii drużyny.

5.9. Menu Tabela

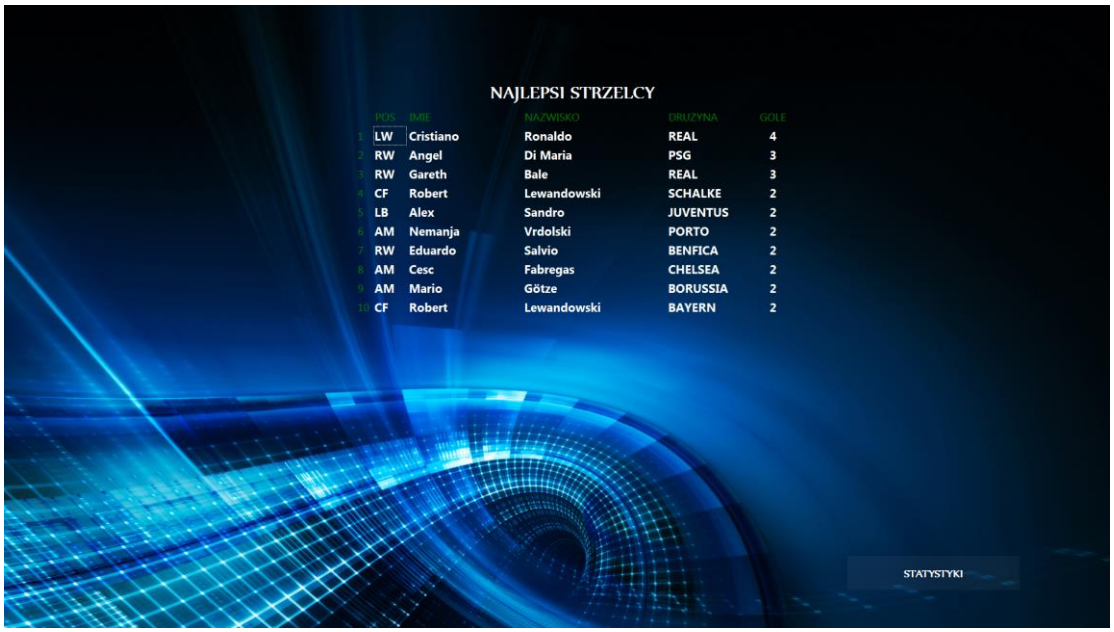
Menu Tabela służy do wyświetlania pozycji drużyn aktualnych rozgrywek ligowych. Tabela zawiera patrząc od lewej: herb drużyny, pozycję w tabeli, nazwę drużyny, ilość rozegranych kolejek, zwycięstwa, remisy, porażki, bramki zdobyte, bramki stracone, bilans bramkowy oraz punkty. O pozycji zajmowanej przez drużynę decydują w pierwszej kolejności zdobyte punkty następnie bilans bramkowy a na końcu zdobyte bramki.



Rys 30. Widok klasy UI_Table

5.10. Menu Najlepsi Strzelcy

Menu Najlepsi Strzelcy to 10 najlepszych strzelców ligi. Tabela zawiera patrząc od lewej strony: pozycję zawodnika w tabeli, pozycję zawodnika na boisku, imię zawodnika, nazwisko zawodnika, drużynę zawodnika oraz ilość zdobytych goli.

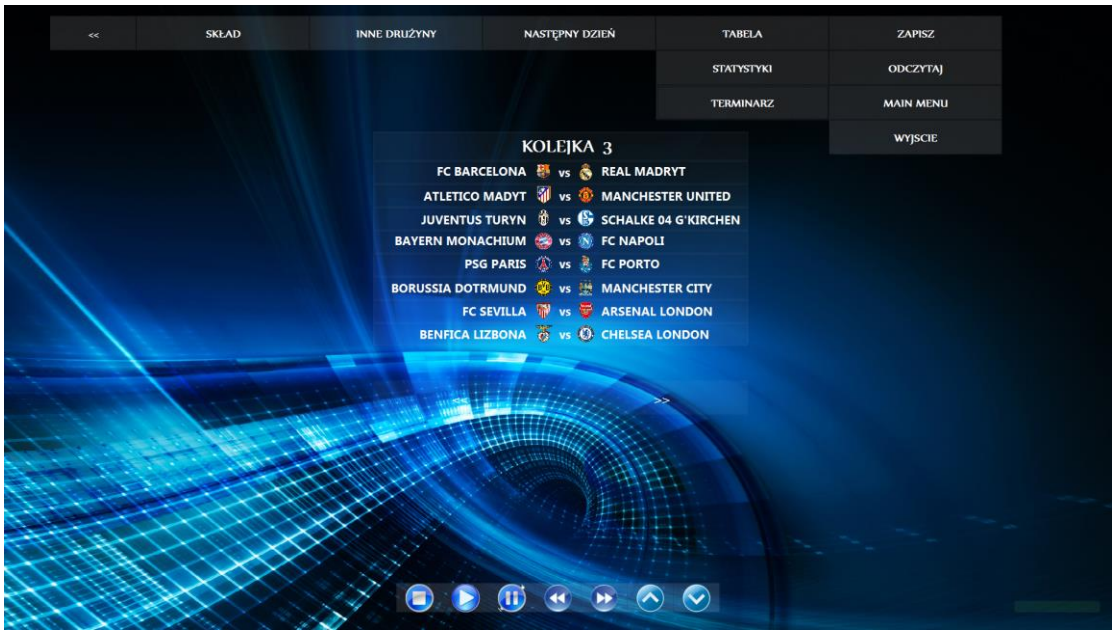


Rys. 31 Widok klasy UI_TopScorer

5.11. Menu Terminarz

Menu Terminarz służy do sprawdzania par meczowych z dowolnie wybranej 1 z 30 kolejek. Menu jest zorganizowane w formie tabeli, która w górnej części zawiera

numer aktualnej kolejki ligowej oraz 8 wierszy. Każdy z wierszy zawiera 2 drużyny z ich herbami, pierwsza drużyna jest gospodarzem spotkani, a druga gościem. Pod tabelą znajdują się dwa przyciski nawigacyjne umożliwiające zmianę kolejki.

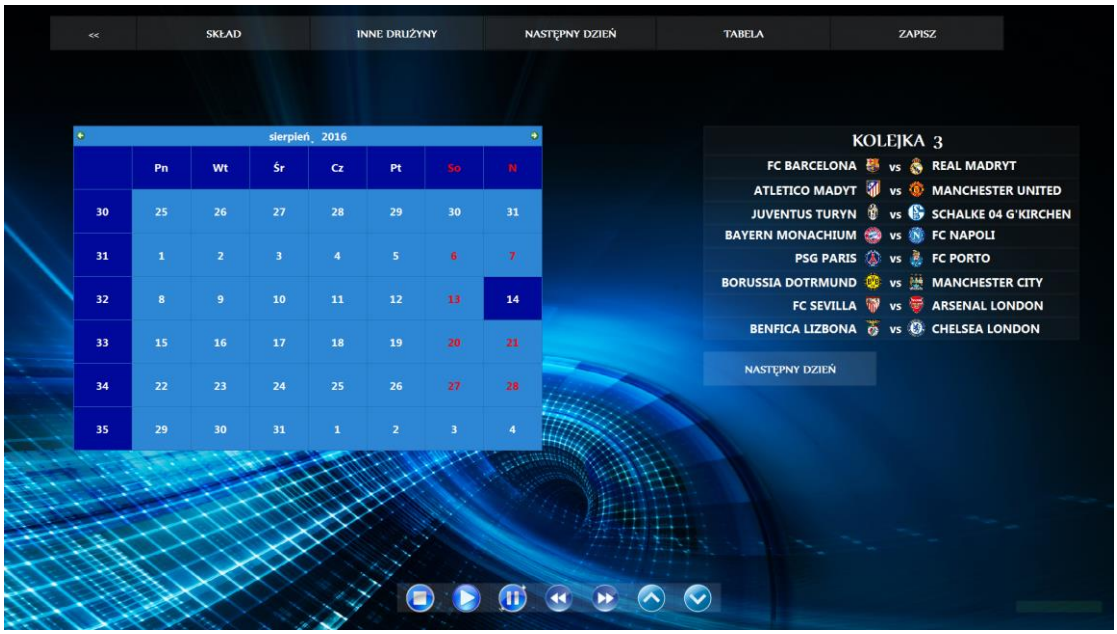


Rys. 32 Widok klasy Ui_Fixture

5.12. Menu Kalendarz

Menu Kalendarz odpowiada za zarządzanie czasem rozgrywek. W lewej części widoku znajduje się kalendarz pokazujący aktualny dzień rozgrywek. Grna część kalendarza wyposażona jest w strzałki nawigacyjne służące do zmiany miesiąca. Miesiąc oraz rok można również zmienić przy pomocy list rozwijanych

umieszczonych pośrodku paska nawigacyjnego. W lewej części ekranu znajduję się terminarz rozgrywek z aktualną kolejką oraz parami meczowymi. Poniżej terminarza usytułowany jest przycisk *Następny Dzień* który, odpowiada za przejście do następnego dnia rozgrywek.



Rys. 33 Widok klasy UI_Match

5.13. Menu Wyniki Meczów

Menu Wyniki Meczów – odpowiedzialne jest za wyświetlanie wyników spotkań oraz strzelców bramek. W jego górnej, centralnej części znajduje się pasek postępu rozgrywanych meczów oraz numer aktualnej kolejki. Poniżej znajduje się 8 paneli w których wyświetlane są wyniki rozegranych spotkań. Każdy panel zawiera parę meczową, wynik spotkania oraz listy strzelców bramek wraz z minutami, w których

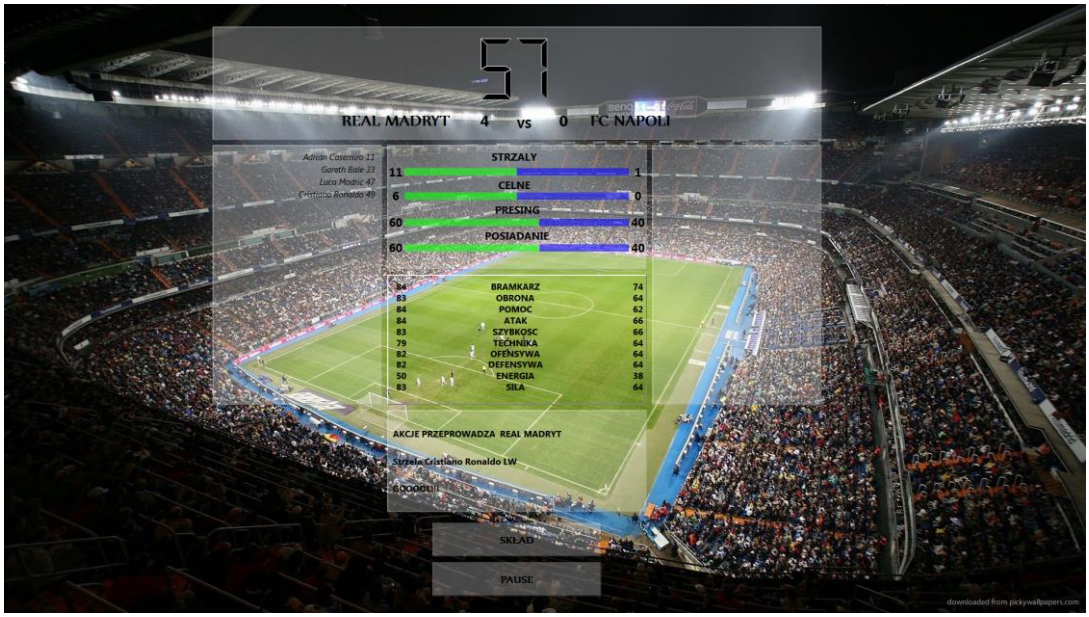
padły gole. W prawym dolnym rogu znajduje się przycisk *Wyjście*, którego naciśnięcie powoduje przeniesienie użytkownika do *Menu Kalendarza.*

Rys. 34 Widok klasy UI_MatchInterface

5.14. Menu Mecz

Menu Mecz odpowiedzialne jest za przedstawienie przebiegu meczu. Dzięki niemu użytkownik jest w stanie śledzić przebieg spotkania oraz ma realny wpływ na jego wynik czy zachowanie zawodników. Menu składa się z 6 paneli oraz 2 przycisków. Panel górny zawiera zegar meczowy, który odpowiedzialny jest za wyświetlanie upływającego czasu spotkania. Poniżej znajduje się informacja jakie

drużyny rozgrywają mecz oraz aktualny wynik. Dwa skrajne środkowe panele służą do wyświetlania listy strzelców goli wraz z czasem, w których zostały zdobyte. Górny środkowy panel odpowiedzialny jest za prezentacje statystyk meczowych zarówno poprzez liczby, jak również w sposób graficzny. Na statystyki składają się strzały, strzały celne, presing drużyny oraz posiadanie piłki.



Rys. 35 Widok klasy UI_LiveMatch

Dolny środkowy panel zawiera statystyki drużyny, które pomniejszają się wraz z upływem czasu spotkania. Statystyki te zawierają dane licząc od góry: bramkarza, obrony, linii pomocy, ataku, szybkości drużyny, techniki, siły ofensywnej i defensywnej drużyny, pozostałej energii oraz aktualnej siły drużyny. Dolny panel służy do wyświetlania komentarzy które opisują aktualną sytuację na boisku. Przycisk *Skład* przenosi użytkownika do *Menu Skład* umożliwiając zmianę zmęczonych zawodników oraz dokonanie odpowiednich zmian w ustawieniu drużyny. Przycisk *Pause* zatrzymuje mecz na dowolną ilość czasu.

Spis ilustracji

Literatura

[1] Alan Ezust, Paul Ezust , *C++ i Qt – Wprowadzenie do wzorców projektowych - Wydanie II*

[2] Stephen Prata, *Język C++ Szkoła programowania – Wydanie VI*

[3] Joseph Albahari, Ben Albahari, *C# 6.0 w pigułce – wydanie VI*

[4] Erich Gamma, Richald Helm, Ralph Johnson, John Vilssides,

Wzorce Projektowe – Elementy oprogramowania obiektowego wielokrotnego użytku.

[5] Wikipedia – *Programowanie obiektowe, Języki programowania*