Dokumentacja Techniczna BomberMan

Sylwia Nowak A2

Table of Contents

Wstęp

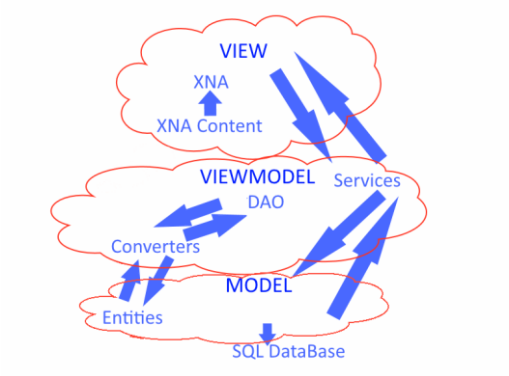
Przedstawiona dokumentacja techniczna opisuje architekturę, klasy oraz schemat algorytmów przygotowanych do implementacji gry Bomberman. Celem gry jest umożliwienie jednemu graczowi zarejestrowania się w aplikacji lub zalogowanie się na istniejące konto, przejrzenie najlepszych wyników wszystkich graczy, zagranie w grę, zapisanie gry oraz załadowanie gry. Gracz ma możliwość zmieniania ustawień gry np. muzyki, animacji, obsługi klawiszy. Celem gry jest osiągnięcie jak największego wyniku punktowego. Punkty są przyznawane za eliminowanie przeciwników oraz prędkość z jaką graczowi udało się ukończyć dany poziom. Wyeliminowanie ostatniego przeciwnika na planszy zalicza poziom i ładuje następny, odpowiednio trudniejszy. Zebrane na jednym poziomie punkty przechodzą na kolejne, aż do samej przegranej (tj. śmierci gracza). Mający dostatecznie dużą liczbę punktów gracz zostaje w chwili przegranej wpisany na listę najlepszych wyników.

Technologie

Zalecany framework to .Net 4.0, język C#. Dodatkowe biblioteki, jakie wykorzystuje aplikacja to XNA Framework, EntityFramework, AutoMapper i biblioteka umożliwiająca wykorzystywanie plików z rozszerzeniem „gif” we frameworku XNA – GifAnimation. Wersje bibliotek powinny być kompatybilne z użytą wersją .Net. Środowisko developerskie VisualStudio 2012 lub starszy.

Architektura

Projekt BomberManGame zawiera w sobie 6 solucji o nazwach BomberMan, BombermanContent, BomberManViewModel, BomberManModel, GifAnimation, GifAnmiation.Pipeline. Ostatnie dwie solucje odpowiadają za operacje związane na gif’ach i są solucjami udostępnionymi przez twórców biblioteki GifAnimation. Solucje BomberMan oraz BomberManContent odpowiadają za widok aplikacji. Tworzą instancję projektu Game XNA. BomberManContent odpowiada za przechowywanie plików multimedialnych oraz obrazków. BomberMan jest główną solucją całego projektu. Generuje widok aplikacji, integruje z BomberManViewModel odpowiedzialnym za przekazywanie informacji z widoku do modelu projektu. BomberManModel wykorzystując EntityFramework łączy się z lokalną bazą na komputerze (lub też ją tworzy, jeżeli taka nie istnieje) i przekazuje obiekty z bazy do Entities lub odwrotnie. Obiekty przekazywane z Model do ViewModel są mapowane przy pomocy biblioteki AutoMapper i tak zmapowane obiekty DAO (DataAccessObjects) są obiektami używanymi do komunikacji między View a ViewModel. AutoMapper korzysta z Converter’ów znajdujących się w ViewModel.



Powyżej schemat komunikacji między solucjami.

Podział katalogów :

```
\---BomberManGame
   |  BomberManDataBase.sln
   |  UpgradeLog.htm
   |
   +---BomberMan
   |  +---BomberMan
   |  |  |  App.config
   |  |  |  Game.ico
   |  |  |  GameManager.cs
   |  |  |  GameThumbnail.png
   |  |  |  packages.config
   |  |  |  Program.cs
   |  |  |  Utils.cs
   |  |  |
   |  |  +---Common
   |  |  |  |  Component.cs
   |  |  |  |  Engine.cs
   |  |  |  |  Screen.cs
   |  |  |  |  Configutation.cs
   |  |  |  |
   |  |  |  +---Components
   |  |  |  |  |  MovingComponent.cs
   |  |  |  |  |
   |  |  |  |  +---MovingComponents
   |  |  |  |  |      Rocket.cs
   |  |  |  |  |      Star.cs
   |  |  |  |  |
   |  |  |  |  \---StateComponents
   |  |  |  |          Block.cs
   |  |  |  |          Button.cs
   |  |  |  |          CheckBox.cs
   |  |  |  |          Label.cs
   |  |  |  |          ProgressBar.cs
   |  |  |  |          TextInput.cs
   |  |  |  |
   |  |  |  \---Engines
   |  |  |          BoardEngine.cs
   |  |  |          PlanetEngine.cs
   |  |  |          RocketsEngine.cs
   |  |  |          SpecialElementsEngine.cs
   |  |  |          StarsEngine.cs
   |  |  |
   |  |  \---Screens
   |  |      |  GameScreen.cs
   |  |      |  HelpMenuScreen.cs
   |  |      |  HighScoresScreen.cs
   |  |      |  LoginScreen.cs
   |  |      |  Menu.cs
   |  |      |
   |  |      \---Menus
   |  |              LoadGameScreen.cs
```

```
|   |           MainMenuScreen.cs
|   |           SettingsScreen.cs
|   |
|   \---BomberManContent
|       |   BomberManContent.contentproj
|       |   high_scores.png
|       |
|       +---Fonts
|       |       Input.spritefont
|       |
|       +---Images
|       |   +---Common
|       |   +---Game
|       |   +---Help
|       |   +---HighScores
|       |   +---LoadGame
|       |   +---Login
|       |   +---MainMenu
|       |   \---Settings
|       +---Music
+---BomberManModel
|   |   App.config
|   |   BomberManContext.cs
|   |   packages.config
|   +---Entities
|   |       BoardElement.cs
|   |       BoardElementLocation.cs
|   |       entities.cd
|   |       Game.cs
|   |       Oponent.cs
|   |       OponentLocation.cs
|   |       User.cs
|
+---BomberManViewModel
|   |   App.config
|   |   DataManager.cs
|   |   packages.config
|   +---Converters
|   |       BoardElementDAOToBoardElementEntity.cs
|   |       BoardElementEntityToBoardElementDAO.cs
|   |       BoardElementLocationDAOToBoardElementLocationEntity.cs
|   |       BoardElementLocationEntityToBoardElementLocationDAO.cs
|   |       GameDAOToGameEntity.cs
|   |       GameEntityToGameDAO.cs
|   |       OponentDAOToOponentEntity.cs
|   |       OponentEntityToOponentDAO.cs
|   |       OponentLocationDAOToOponentLocationEntity.cs
|   |       OponentLocationEntityToOponentLocationDAO.cs
|   |       UserDAOToUserEntity.cs
|   |       UserEntityToUserDAO.cs
|   |
|   +---DataAccessObjects
|   |       BoardElementDAO.cs
|   |       BoardElementLocationDAO.cs
|   |       ClassDiagram1.cd
|   |       GameDAO.cs
|   |       OponentDAO.cs
|   |       OponentLocationDAO.cs
|   |       UserDAO.cs
|   |
|   +---Services
|   |       BoardService.cs
|   |       ClassDiagram1.cd
|   |       GameService.cs
|   |       OponentService.cs
|   |       UserService.cs
|   |
|
+---GifAnimation
|   |   GifAnimation.cs
|   |   GifAnimation.csproj
```

```
|   |   GifAnimationContentTypeReader.cs
|   \---Properties
|           AssemblyInfo.cs
|
+---GifAnimation.Pipeline
|   |   GifAminationImporter.cs
|   |   GifAnimation.Pipeline.csproj
|   |   GifAnimationContent.cs
|   |   GifAnimationContentTypeWriter.cs
|   |   GifAnimationProcessor.cs
|   |   Quantizer.cs
|
```

Biblioteki

EntityFramework

Entity Framework jest narzędziem typu ORM (Object Relational Mapping), pozwalającym odwzorować relacyjną bazę danych za pomocą architektury obiektowej. Istnieją 3 sposoby na stworzenie modelu danych.

- Database First - podejście to stosujemy gdy mamy już gotową bazę danych. Za pomocą ADO.NET Entity Data Model Designer dodajemy istniejące w bazie tabele, widoki oraz procedury składowane.
- Model First - za pomocą tego podejścia nie musimy pisać żadnego kodu SQL. Wystarczy, że stworzymy model danych w ADO.NET Entity Data Model Designer. Na podstawie stworzonego modelu tworzona jest struktura bazy danych.
- Code First - pierwsze piszemy kod klas, na podstawie którego tworzony jest model danych oraz struktura bazy danych.

W projekcie stosowany jest model CodeFirst a obiektami odpowiadającymi tabelom w bazie SQL są klasy znajdujące się w namespace BomberManModel.Entities.

AutoMapper

AutoMapper to biblioteka, która jest w stanie przepisać wartości obiektu pewnego typu, do analogicznych właściwości w drugim obiekcie, który posiada inny typ. W przypadku gdy pola różnią się nazwami, istnieje możliwość stworzenia mapowania uwzględniającego różnice między oboma klasami. Biblioteka obsługuje również typy zagnieżdżone, dzięki czemu możliwe jest mapowanie nawet bardzo złożonych struktur.

Narzędzia typu AutoMapper wykorzystujemy głównie w sytuacji, gdy tworzymy rozbudowane aplikacje, w których możemy rozróżnić warstwy danych oraz prezentacji, tak jak w omawianym projekcie. W projekcie stosujemy AutoMappera do przepisywania Entities na DAO. View oraz ViewModel komunikują się obiektami DAO co jest bezpieczniejsze niż przekazywanie obiektów typu Entity. Do konwertowania obiektów stosowane są zawsze własne konwertery z namespace BomberManViewModel.Converters, to rozwiązanie jest bezpieczniejsze i umożliwia większą kontrolę nad danymi wędrującymi z modelu do widoku i odwrotnie.

GifAnimation

GifAnimation to oddzielna biblioteka, dokładniej silnik napisany w C# dla XNA umożliwiający łatwe i szybkie wstawianie obiektów z rozszerzeniem gif do widoku aplikacji. Stosowanie biblioteki ma swoje uzasadnienie tylko dla gif'ów z wieloma klatkami. W projekcie istnieje zastosowanie biblioteki podczas wyświetlania komunikatów o wygranej lub porażce w grze. Dzięki temu możliwe jest wyświetlenie rozbudowanej, gotowej animacji.

Wzorce projektowe

Kompozyt

Kompozyt to strukturalny wzorzec projektowy, którego celem jest składanie obiektów w taki sposób, aby klient widział wiele z nich jako pojedynczy obiekt. Argumentem za wzorcem jest rozbudowana siatka, po której porusza się gracz. Dla użytkownika wygląda jak spójna całość, każdy obiekt znajdujący się na planszy jest oddzielną instancją klasy Component.

Most

Most to strukturalny wzorzec projektowy, który pozwala oddzielić abstrakcję obiektu od jego implementacji. Zaleca się stosowanie tego wzorca aby:

- odseparować implementację od interfejsu,
- poprawić możliwości rozbudowy klas, zarówno implementacji, jak i interfejsu (m.in. przez dziedziczenie),
- ukryć implementację od klienta, co umożliwia zmianę implementacji bez zmian interfejsu.

W projekcie warto odseparować implementację update'owania widoku Componentów od definicji metody. Obiekty dziedziczące po Component posiadają własne sposoby generowania widoku w zależności od wymiarów okna czy czasu gry.

Stan

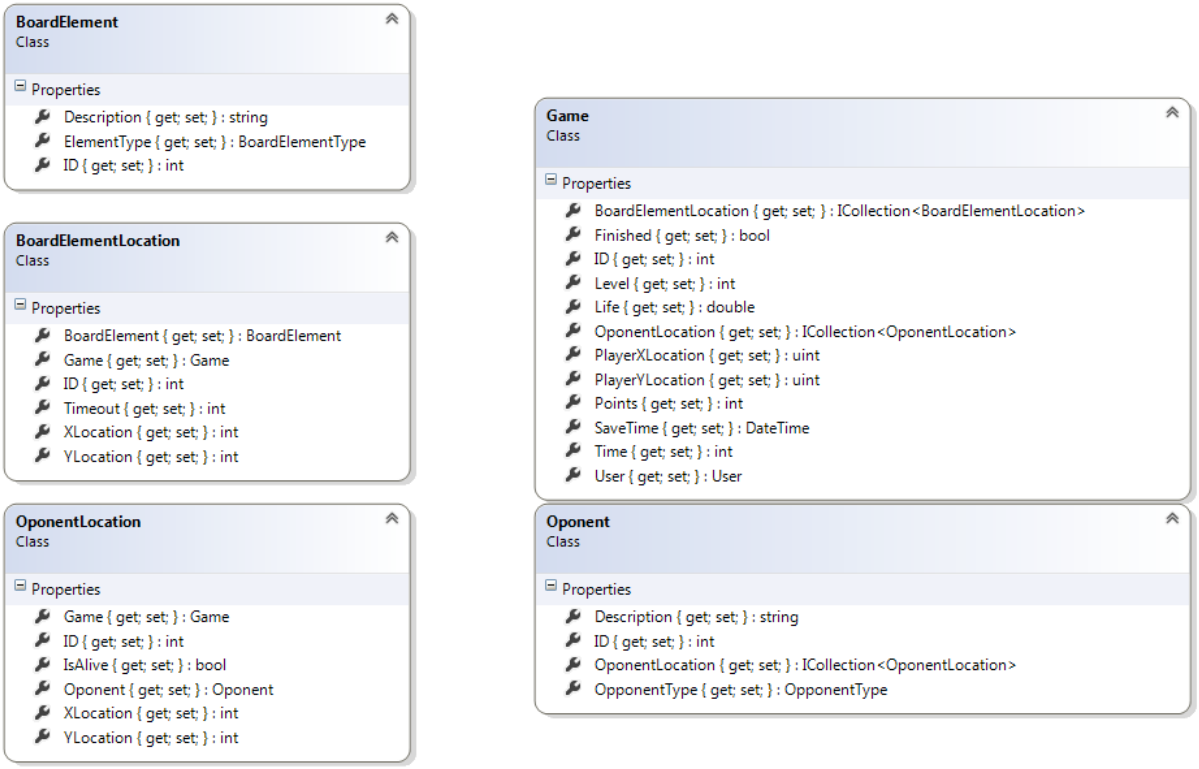
Stan – czynnościowy wzorzec projektowy, który umożliwia zmianę zachowania obiektu poprzez zmianę jego stanu wewnętrznego. Innymi słowy – uzależnia sposób działania obiektu od stanu w jakim się aktualnie znajduje. W projekcie, niektóre elementy związane z planszą będą zmieniały swój widok i zachowanie w zależności od czasu gry. Wzorzec umożliwia panowanie nad zmianami takimi jak śmierć przeciwnika, wybuch bomby, zużycie bonusu.

Model dziedziny

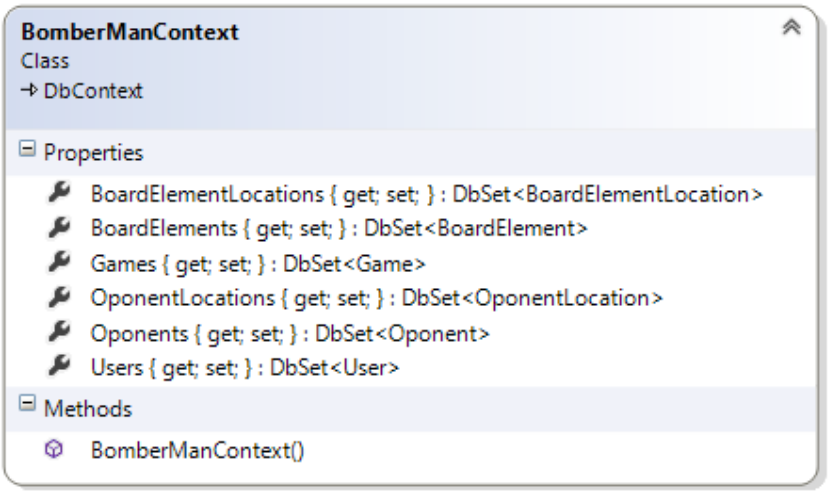
Baza Danych

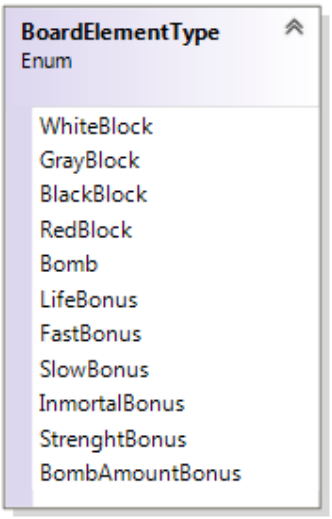
Informacje związane ze stanem gry (punkty, data, dokładne rozmieszczenie bonusów, bomb oraz przeciwników na planszy, zużycie życia oraz dodatkowych mocy) przechowywane są w bazie danych. Użytkownik tworząc konto dodaje nowy rekord do tabeli Users. Logując się, w tabeli wyszukiwany jest rekord z loginem użytkownika, który jest unikalny. Porównywane jest zahashowane i posolone hasło z rekordem w bazie. W razie sukcesu użytkownik dostaje dostęp do gry, w przeciwnym przypadku wyświetlany jest komunikat związany z błędem logowania.

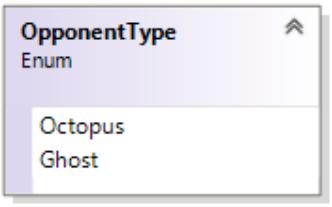
Poniżej widok namespace BomberManModel.Entities



Rozróżniamy obiekty na planszy na Oponentów i BoardElement. Każdy z tych elementów ma swój własny typ. Dodatkowo na planszy mamy zawsze jednego gracza. Wszystkie potrzebne informacje przechowywane są w tabeli Game. Tabele łączące Oponentów i Elementy z Game to OponentLocation i BoardElementLocation. Obie tabele zawierają informacje o współrzędnych obiektu w świecie planszy tzn. jeżeli plansza ma wymiary 12x14 a obiekt współrzędne XLocation = 0, YLocation = 0 to znajduje się w lewym górnym rogu planszy, czyli na polu (0,0). Oponenci dodatkowo mają informacje o tym czy żyją. Podczas nadpisywania stanu gry nie wpisujemy już nieżyjących Oponentów, czyścimy również rekordy bazy z Oponentów, którzy już nie żyją. BoardElementy mają swoją żywotność wyrażoną w milisekundach w polu Timeout. Jeżeli przy zapisie pojawi się element z Timeoutem < 0 to usuwamy rekord z bazy danych. Informacje o opisach komponentów potrzebne do wyświetlenia w widoku Help uzyskujemy z bazy. Poniżej diagram Contextu bazy oraz rodzaje elementów i oponentów. W Game mamy pole Time, do którego będziemy dodawać czas jaki zawodnik poświęcił na dany poziom. Przy rozpoczęciu poziomu Time jest równy zero. Time trzymamy domyślnie w milisekundach.



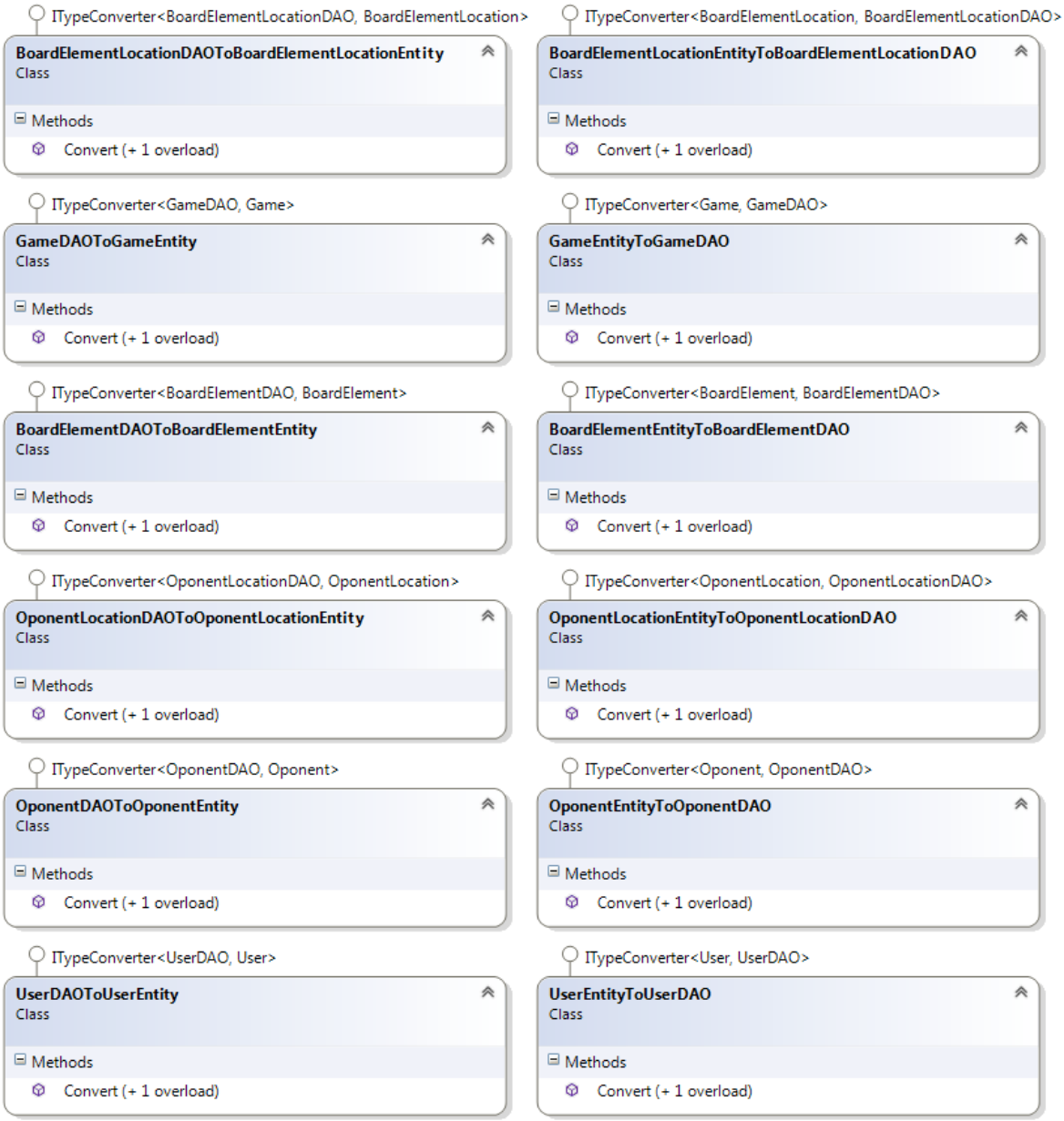




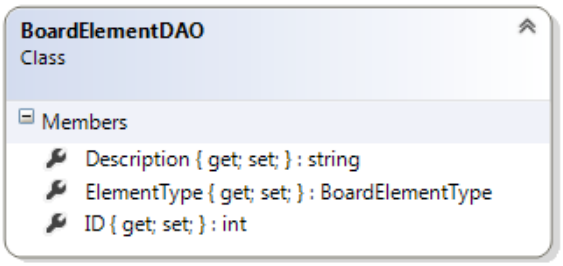
BoardElementType zawiera elementy zarówno związane z kolorem pola jak i elementy pojawiające się na polach planszy. WhiteBlock to jednostkowe pole planszy, po którym może poruszać się każdy (gracz, ośmiornica, duch). Szare pola to pola zniszczalne przez bomby, dodatkowo ośmiornica może przejść przez pole szare (jako jedyna) ale zajmuje jej to 2 razy więcej czasu niż zwykły ruch. Wszyscy uczestnicy poruszają się z taką samą prędkością. BlackBlock to niezniszczalny element planszy. Schowany za nim gracz nie jest widoczny dla przeciwników. RedBlock to pole planszy, które jest niszczone przez bombę. Kto stoi na polu czerwonym umiera. Pola czerwone po określonym czasie zamieniają się w pola białe. Pole RedBlock pojawia się tylko w zastępie WhiteBlock. Wszystkie Bonusy proporcjonalnie do poziomu przyznają zgodnie z nazewnictwem odpowiednich własności (na stałe – LifeBonus lub czasowo – pozostałe).

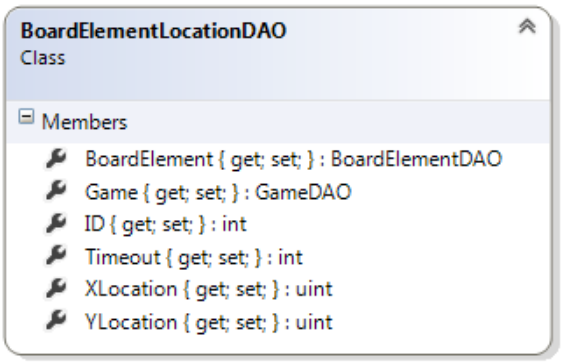
ViewModel

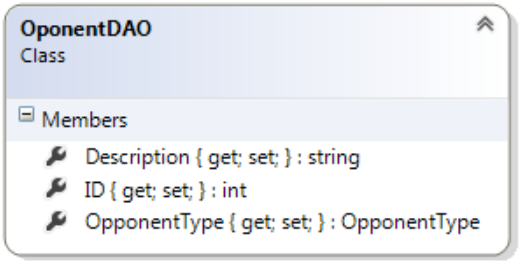
Do łączenia widoku prezentowanego graczowi z warstwą danych służy warstwa ViewModel (BomberManViewModel). W tej części konwertujemy obiekty uzyskane z widoku na obiekty Entity i odwrotnie. Używamy do tego konwerterów znajdujących się w namespace BomberManViewModel.Converters. Poniżej lista dostępnych konwerterów.

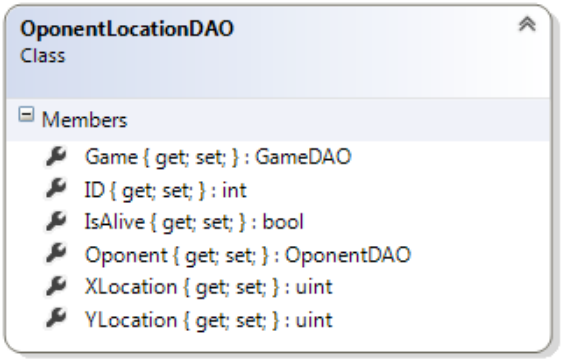


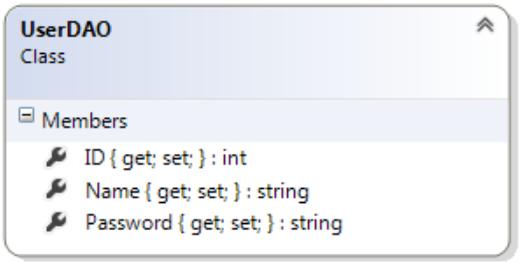
Obiekty DataAccessObjects są przekazywane do View i z niego odbierane. Umożliwia to zabezpieczenie przed przesyłaniem pełnych obiektów z bazy i niekontrolowanym edytowaniem Entity. Poniżej diagram klas zawartych w namespace BomberManViewModel.DataAccessObjects.

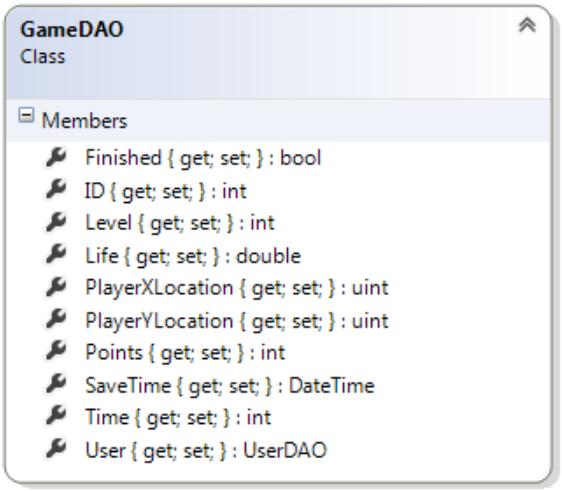




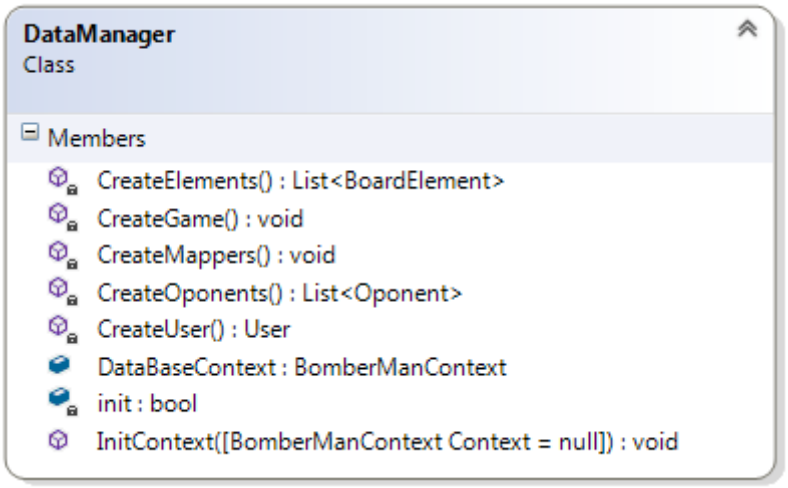




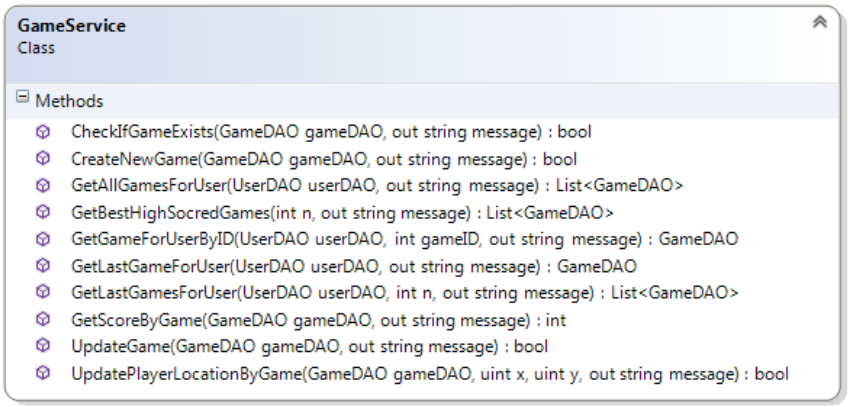


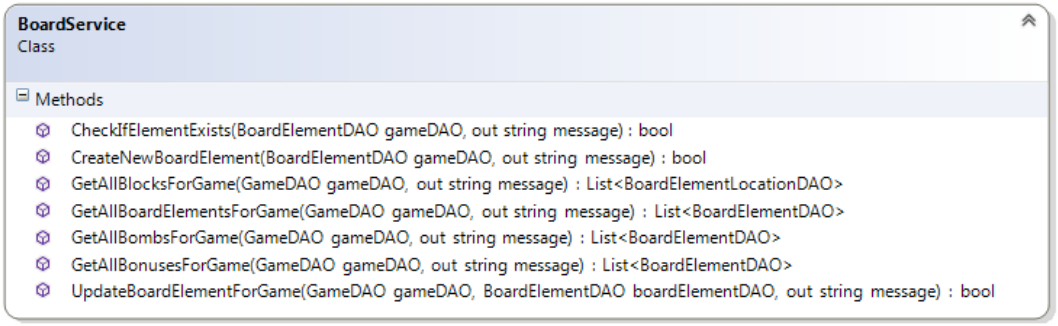


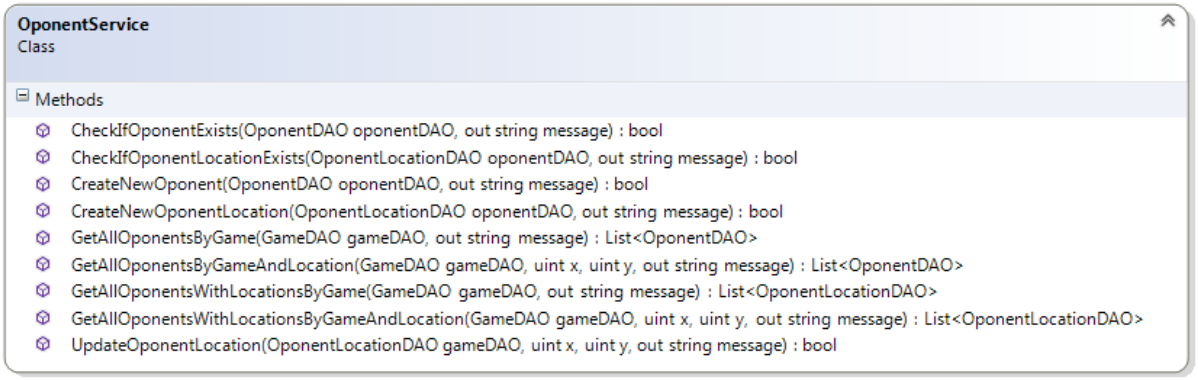
Klasa odpowiedzialna za zarządzanie Contextem bazy oraz mapowaniem obiektów to DataManger. W niej istnieje metoda tworząca bazę o ile nie zostanie wykryta już istniejąca baza zgodna z naszym modelem. Przy tworzeniu bazy wstawiane są rekordy Opponent oraz BoardElement. Wszystkie opisy wybierane są z Resources. Poniżej diagram klasy DataManager.

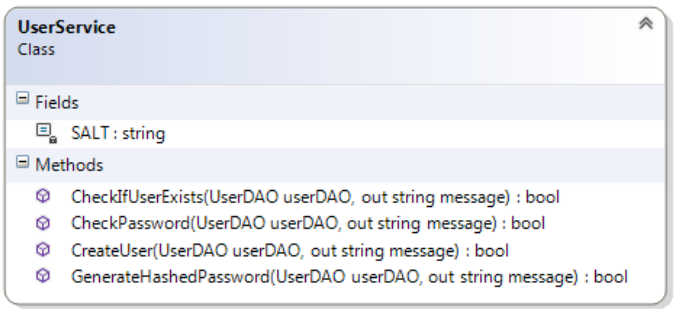


Widok chcąc uzyskać informacje z modelu wywołuje odpowiednie metody statyczne z wybranego serwisu z namespace BomberManViewModel.Services. Wszystkie elementy przekazywane z widoku lub otrzymywane z serwisów nie są obiektami typu Entity. Widok przekazuje zawsze obiekty DAO do wybranego serwisu otrzymując z zależności od oczekiwanego wyniku obiekt lub typ prosty wraz z parametrem out String będącym ustawianym na wiadomość, którą chcemy wyświetlić w razie niepowodzenia. Domyślnie w parametrze out będzie przekazywany null. Poniżej diagram wszystkich serwisów dostępnych dla widoku.





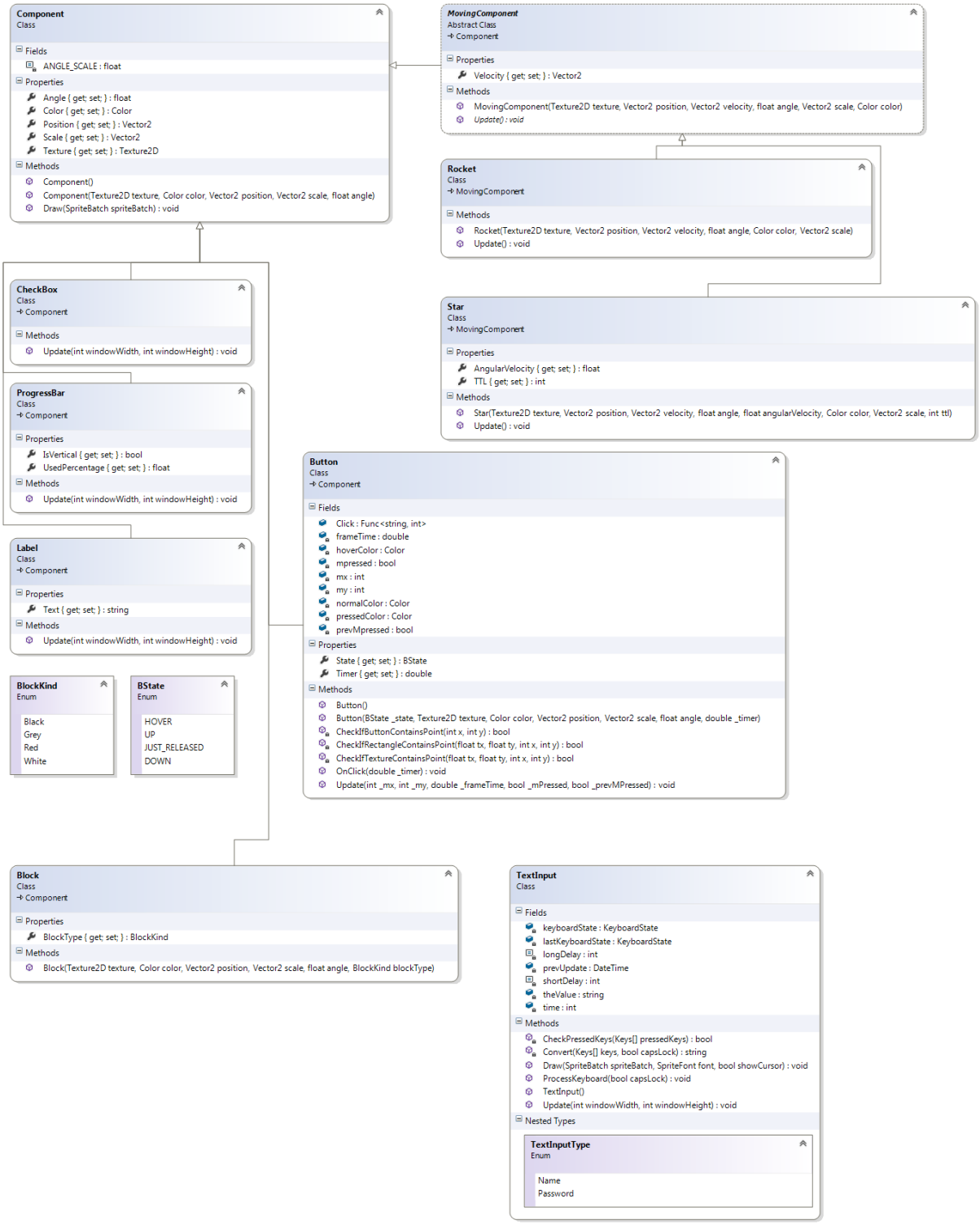




View

Widok jest tworzony w oparciu o bibliotekę XNA. Generowanie opiera się zawsze na dwóch metodach void Update(…) oraz void Draw(…) z różnymi parametrami. Wszystkie elementy pojawiające się w oknie programu można sprowadzić do jednostkowych obiektów z namespace BomberMan.Commons.Components lub do obiektu Component z namespace BomberMan.Commons. Jednostkowe obiekty dziedziczą bezpośrednio po klasie Component lub

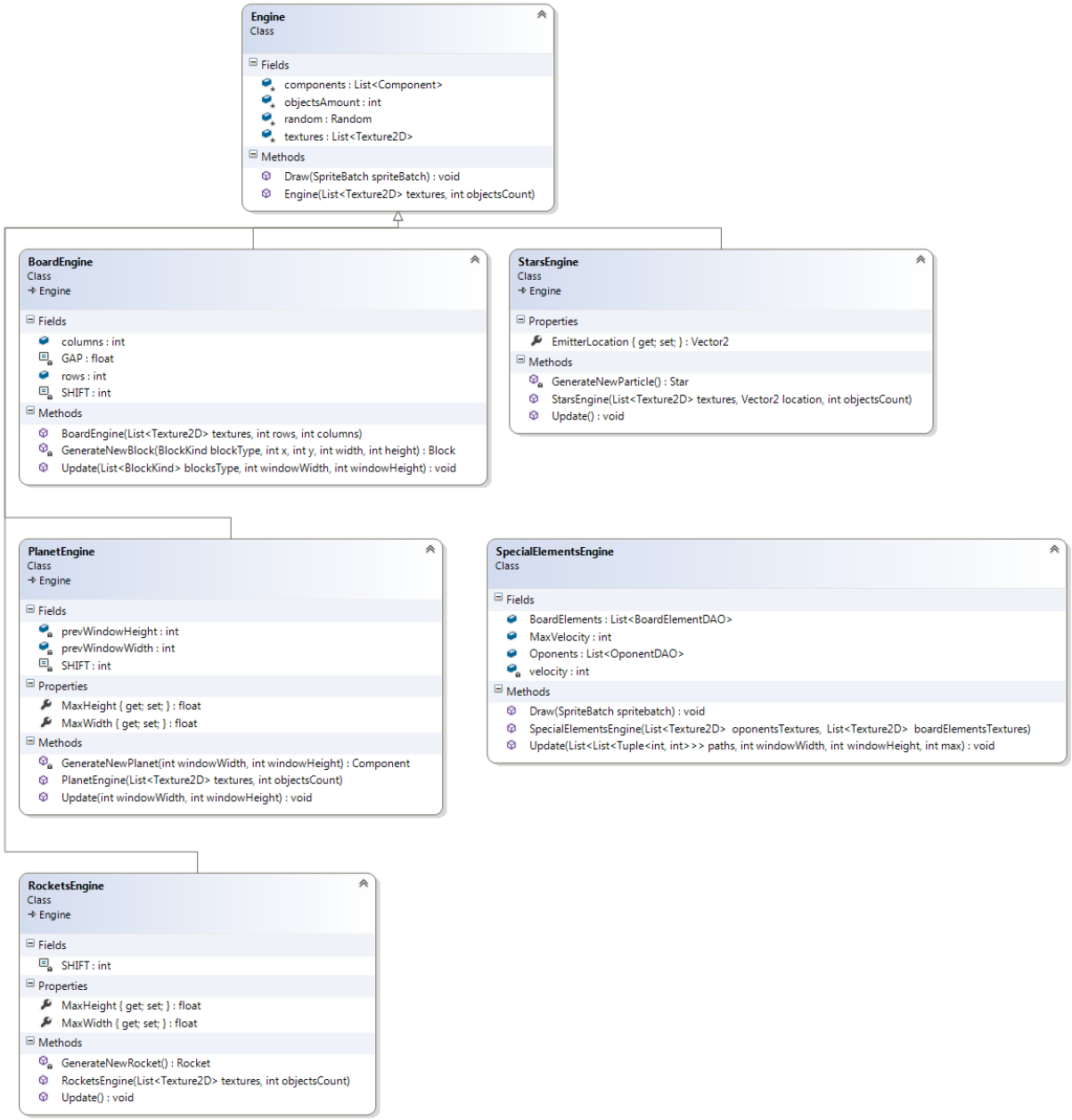
pośredniej klasie abstrakcyjnej MovingComponent. Istnieje dodatkowy jednostkowy obiekt nie dziedziczący po żadnej z wymienionych – TextInput odpowiedzialny za wpisywanie hasła lub loginu do odpowiednich pól w screenach. Poniżej przedstawiony schemat zależności między podstawowymi komponentami wyświetlanymi w oknie programu.



Oczywiście podczas generowania widoku wyświetlane są również instancje klas Texture2D, które są zawarte w bibliotece XNA. MovingComponents (Rocket, Star) różnią się od zwykłego Component (posiadającego parametry charakteryzujące jego położenie na ekranie, texturę, kąt obrotu, skalę oraz kolor) właściwością prędkość. Dzięki temu w metodzie update można przesuwać

obiekt zgodnie z przetrzymywaną prędkością we właściwości Velocity. Obiekty dziedziczące bezpośrednio po klasie Component charakteryzują się tym, że ich położenie, ewentualnie rozmiar, zmienia się jedynie podczas zmiany rozmiaru okna. Instancje Obiektów Rocket i Star generowane są dla wszystkich widoków gry, pozostałe elementy pojawiają się w zależności od wyświetlanego widoku. Klasa Button symuluje działanie przycisku (akcje hover, click, release), Block odpowiada jednostkowemu blokowi planszy, pozostałe klasy odpowiadają obiektom zgodnie z nazewnictwem.

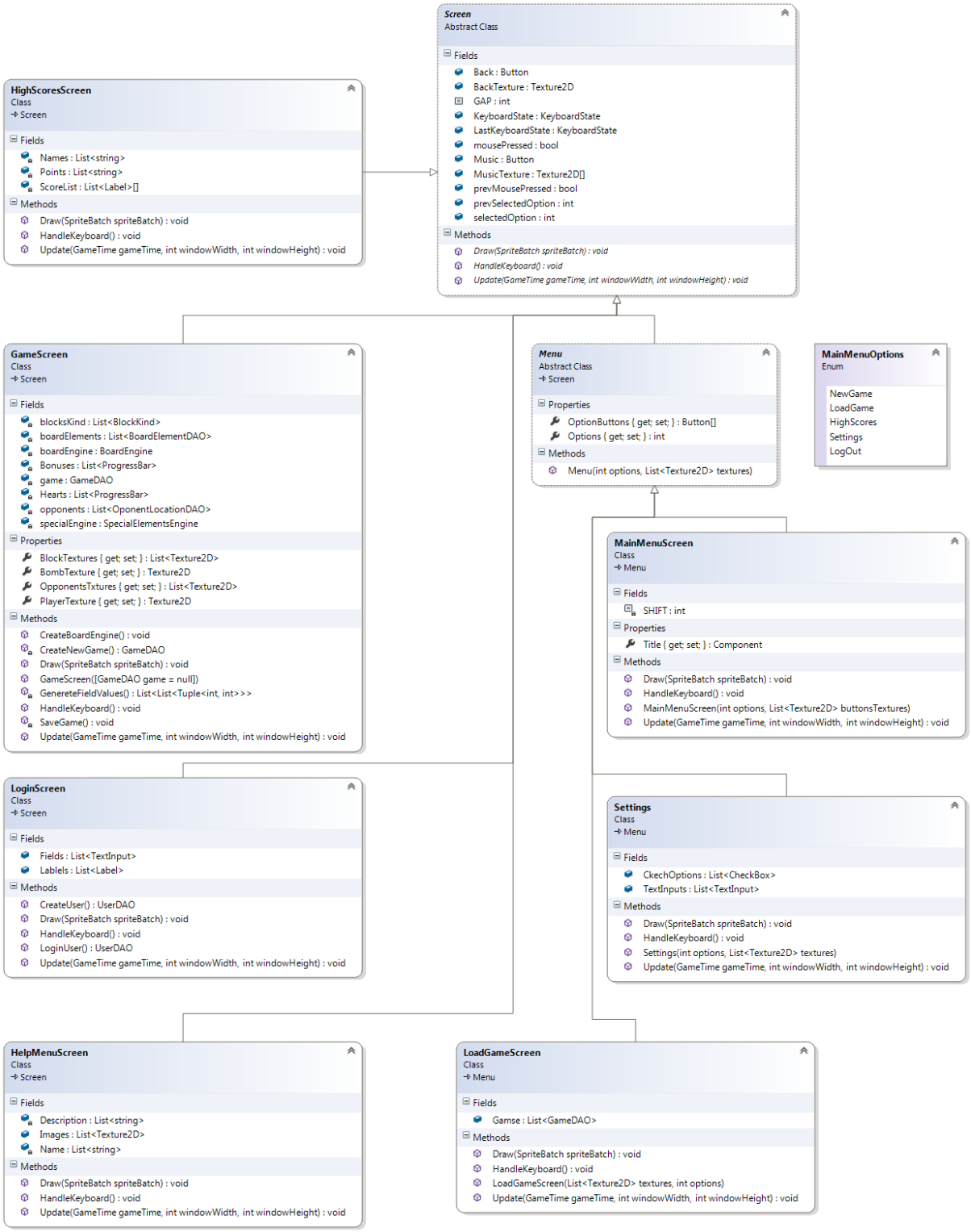
Podstawowe jednostki do wyświetlania na ekran są przechowywane bezpośrednio w Enginach generujących częściowe widoki składające się z zadanych elementów lub Screenach pełniących funkcje okienek w aplikacji. Poniżej przedstawione zależności pomiędzy klasami dziedziczącymi po klasie Engine.



Zgodnie z nazewnictwem PlanetEngine to silnik generujący zadaną ilość instancji Component, które będą zawierać textury przedstawiające planety. Zgodnie z przyjętą zasadą każdy Engine posiada metodę void Draw(SpriteBatch) dziedziczoną po klasie Engine, która rysuje wszystkie komponenty z

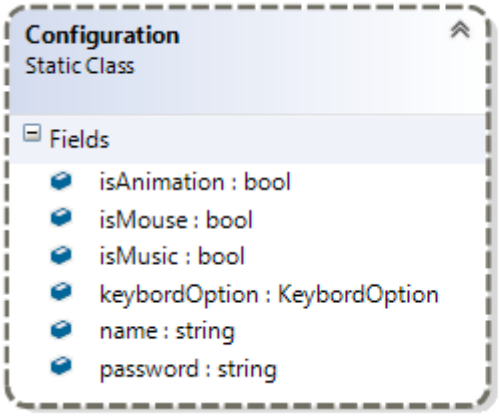
listy components. Metody Update dla wszystkich silników wywołują w swoich metodach Update dodatkowo medody Update dla każdego elementu z listy comopnents. Metoda Draw w klasie Engine w swojej implementacji również dla każdego elementu z listy Componentów wywojuje metodę Draw. Dodatkowo RocketEngine odpowiada za generowanie Rocketów, BoardEngine za utworzenie całej planszy ale tylko jednostkowych Blocków bez Elementów oraz Oponentów. StarsEngine odpowiada za efekty zostawiania śladu podczas ruchu myszką po ekranie. SpecialElementEngine nie dziedziczy po klasie Engine jednakże jest również silnikiem. Przechowuje instancje obiektów z DAO, otrzymane z ViewModel i na podstawie listy Oponentów i Elementów wyświetla najpierw specjalne elementy na odpowiednim miejscu siatki (w zależności od danych w XPosition i YPosition), następnie wykonuje to samo dla listy Oponentów. Przy tak zachowanej kolejności będziemy mieć pewność, że zawsze Block będzie narysowany na samym spodzie, następnie elementy odpowiadające BoardElementDAO i na końcu postacie. Nigdy bonusy nie przysłonią nam postaci.

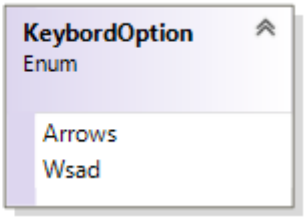
Screeny to najwyżej postawione jednostki tworzące widoki. Posiadają zarówno Enginy wspierające generowanie dużych ilości obiektów Component jak i własne obiekty jednostkowe. Zmiana Screena odpowiada pojawianiu się nowego widoku. Pierwszym pojawiającym się zawsze Screenem jest LoginScreen umożliwiający zalogowanie bądź zarejestrowanie się gracza. Po udanej weryfikacji bądź założeniu konta Screen zostaje podmieniony na MainMenuScreen. Gracz widzi dostępne opcje do wyboru : . Opcja New Game odsyła nas do widoku GameScreen gdzie rozpoczyna się pierwszy poziom gry. Load Game zmienia widok na LoadGameScreen, który umożliwia wybranie z listy zapisanych gier wybranie gry, którą chce się kontynuować. Nigdy nie trzymamy w tej liście gier, które się zakończyły. Użytkownik może bezpowrotnie usunąć zapisany stan gry. Opcja High Scores odsyła do listy najlepszych dziesięciu graczy. Opcja Settings pozwala zmienić ustawienia gry a Log Out wylogować użytkownika i przejść do panelu logowania. Poniżej przedstawione diagramy wszystkich dostępnych Screenów z namespace BomberMan.Screens wraz z klasą macierzystą Screen.



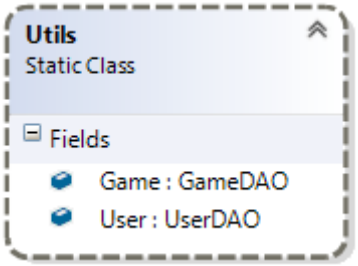
Każdy widok posiada metody Draw, Update oraz HandleKeybord odpowiadające odpowiednio za rysowanie obiektów z klasy, zmienianie położeń, rozmiarów bądź innych parametrów elementów podczas Update oraz obsługę klawiszy w zależności od wybranej konfiguracji w Configurations. Jeżeli Screen posiada Engine lub jednostkowy obiekt to wywołuje odpowiednio w metodach Draw oraz Update metody Draw oraz Update dla tych elementów. Screeny posiadają odpowiednio przycisk Back, który odsyła do MainMenuScreen lub w przypadku MainMenuScreen do

panelu LoginScreen wylogowując tym samym użytkownika. Każdy Screen ma możliwość wyłączenia na szybko dźwięku bez wchodzenia w opcje. Istnieje specjalny przycisk do zmiany konfiguracji dźwięku. Wszystkie ustawienia związane z aplikacją a nie danym poziomem przechowywane są w klasie Configurations. Poniżej diagram klasy.

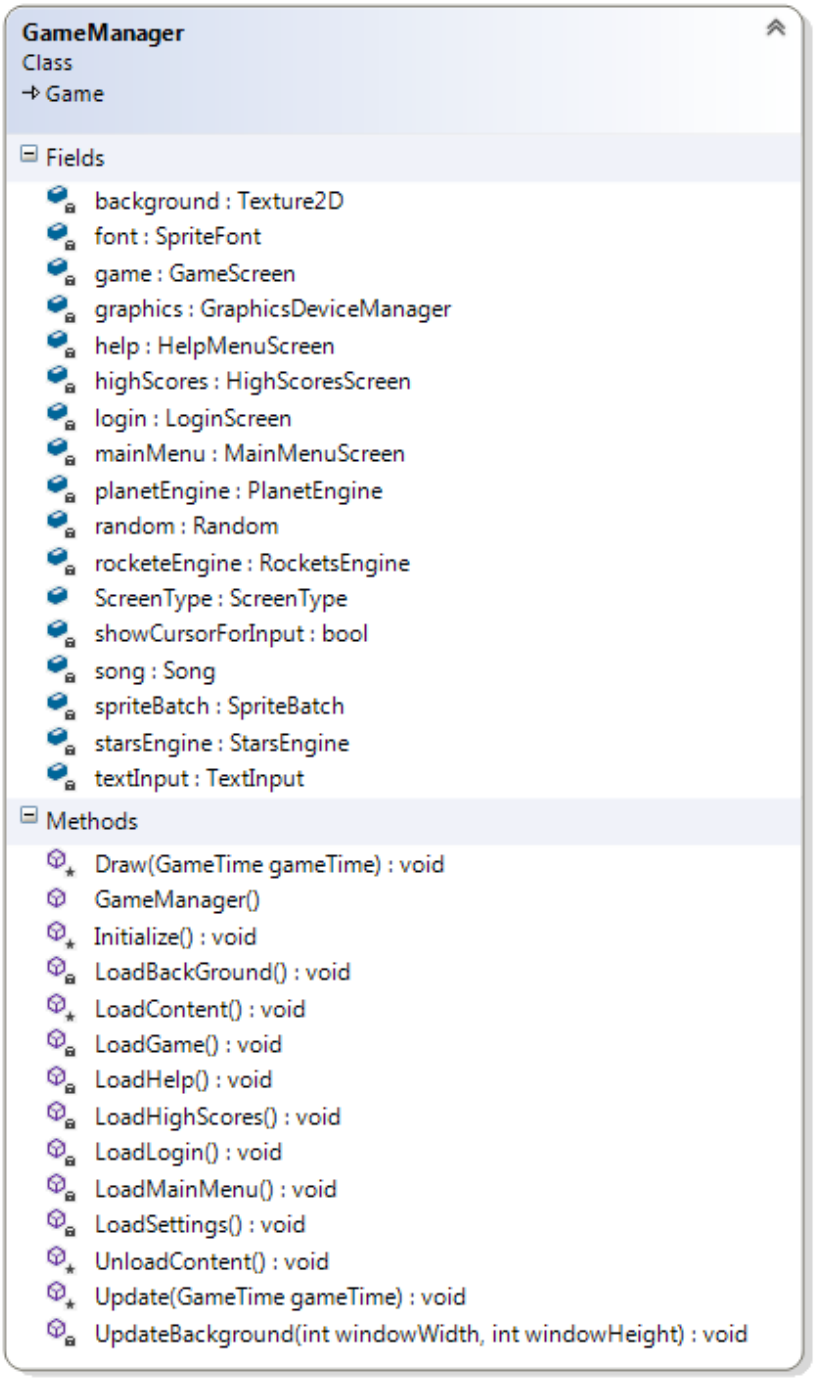


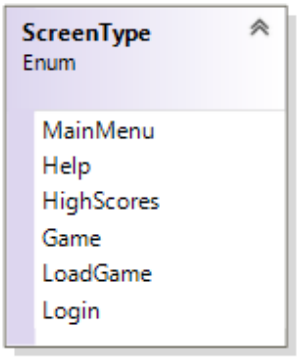


Screeny mają możliwość wywoływania metod z dostępnych serwisów z ViewModel. W momencie zakończenia gry lub zmienienia widoku (pauza) aktualne dane zapisywane są do klasy Utils. Podobnie podczas zapisania uczestnika po zalogowaniu do aplikacji zapisywany jest obietk UserDAO do Utils. Analogicznie ze zmianami na przechowywanym GameDAO. Poniżej diagram klasy Utils.



Główną klasą zajmującą się przełączaniem widoków, ładowaniem Textur, muzyki i nie tylko z projektu BomberManContent zajmuje się GameManager. Jego zadaniem jest odpowiednie updatowanie widoków w zależności od aktualnego Screena oraz przechowywanie załadowanych mediów z Contentu dla wspólnego tła dla wszystkich screenów. GameManager zarządza pojawiającymi się planetami, efektem śladu dla myszki oraz latającymi statkami kosmicznymi. Trzyma instancje wszystkich screenów, dzięki polu statycznemu screenType z każdego poziomu aplikacji możliwe jest zakomunikowanie zmiany screena by GameManager mógł wybrać Screena do rysowania i updatowania. Instancja GameManagera jest tworzona podczas funkcji Main w klasie Program. Poniżej diagram klasy GameManager.





Algorytmy

GenerateFieldValues – GameScreen:

Algorytm zastosowany do generowania dla zadanej pozycji przeciwnika (ox, oy) i gracza (px , py) (w świecie planszy), tablicy BlockKind[n][m] ścieżki przez którą ma iść przeciwnik, żeby dopaść gracza.

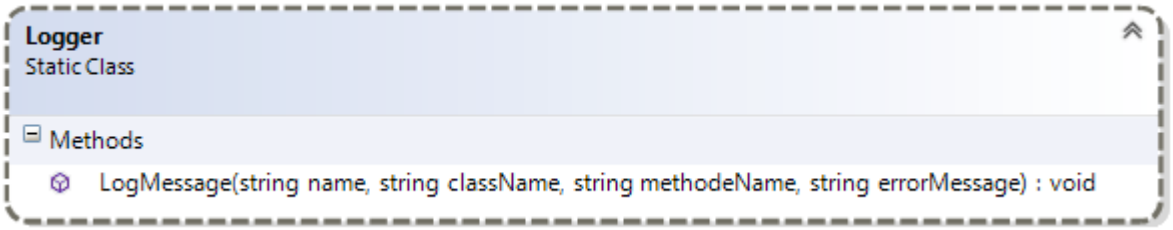
Całość jest stosowana dla wszystkich przeciwników na planszy, którzy jeszcze żyją.

Algorytm opiera się na algorytmie A*. Algorytm dla obu przeciwników przyjmuje za wierzchołki grafu wszystkie pola planszy. Wierzchołki numerowane są od 0 do n*m, numery nadawane są wierszami.

```
GenerateFieldValues()
{
	Dla każdego przeciwnika wylicz :
	{
		List<Tuple<Tuple<int,int>,bool>> Vertices = wierzchołki planszy, pole
bool = false
		oznacza nie przejrzane do tej pory
		List<int> Wagi = lista MaxInt
		Wagi<nr wierchołka startowego> = 0
		foreach (wiechołek)
		{
			if(blok jest biały)
			{
				waga = 0;
			}
			if(przeciwnik to ośmiornica)
			{
				if(blok jest szary)
				{
					waga = 1;
				}
			}
		}
		zastosuj algorytm a* dla wierchołkow podanych powyżej i wag. Zgodnie z
konwencją numeracji.
	}
	return List<List<Tuple<int,int>>>()
}
```

Logger

Wszystkie klasy w namespace BomberManViewModel.Services podczas wykonywania metod i natrafienia na błędy logują wszystkie errory w postaci [Date][User_Name][Class.Name][Method Name][error message]. Logi są generowane do pliku textowego Logger.txt w katalogu BomberManViewModel. Metody z serwisów wywołują statyczną metodę klasy Logger. Poniżej diagram klasy.



SOLID

Zgodnie z konwencją SOLID klasy zostały zaprojektowane tak by

- *Nigdy nie powinno być więcej niż jednego powodu do modyfikacji klasy*
- *Metody były otwarte na rozszerzenia*
- *Funkcje, które używają wskaźników lub referencji do klas bazowych, były w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.*
- *Klient nie był zmuszany do implementacji metod, których nie użyje*
- *Klasy dziedziczyły po klasach abstrakcyjnych*

Najlepszym przykładem dobrego zastosowania Solid są klasy dziedziczące po Screen i Menu. Obie klasy są klasami abstrakcyjnymi. Z góry wiadomo, że każdy Screen musi obsługiwać możliwość używania klawiszy do wybierania opcji/ poruszania się po planszy itp. stąd metoda HandleKeybord(). Aby móc dostosowywać rozmiar Screena przy jakimkolwiek resize okna aplikacji należy zadbać by każdy Screen miał metodę, która wywoływać będzie GameManager w swojej metodzie Update() stąd obowiązkowa jest implementacja metody Update(GameTime time, int windowWidth, int windowHeight) przez każda klasę będącą Screenem. Parametr time pojawia się, ponieważ w każdym przypadku wyliczamy czas od kliknięcia, do kliknięcia itp. Tak samo z metoda Draw, która musi być implementowana, a jest na tyle szczególna, że każda klasa ma swoją własną różną implementację (wywoływane są metody Draw dla różnych komponentów).Wprowadzenie kolejnej abstrakcyjnej klasy - Menu jest całkowicie uzasadnione. Część Screenów potrzebowała dodatkowych pól reprezentujących przyciski, jednakże nie każdy screen musi posiadać listę przycisków w takiej formie. Używanie w metodach obiektów typu Screen nie zaburza zamienienia obiektu na np. ScreenSettings. Wszystkie metody są otwarte na rozszerzenia. Nie jest problemem wprowadzenie dodatkowych pól czy metod w klasach o ile jest to uzasadnione.