Left 4 Dead

Gra zręcznościowa

Łukasz Hryniewicki, Kamil Kapliński

PS8

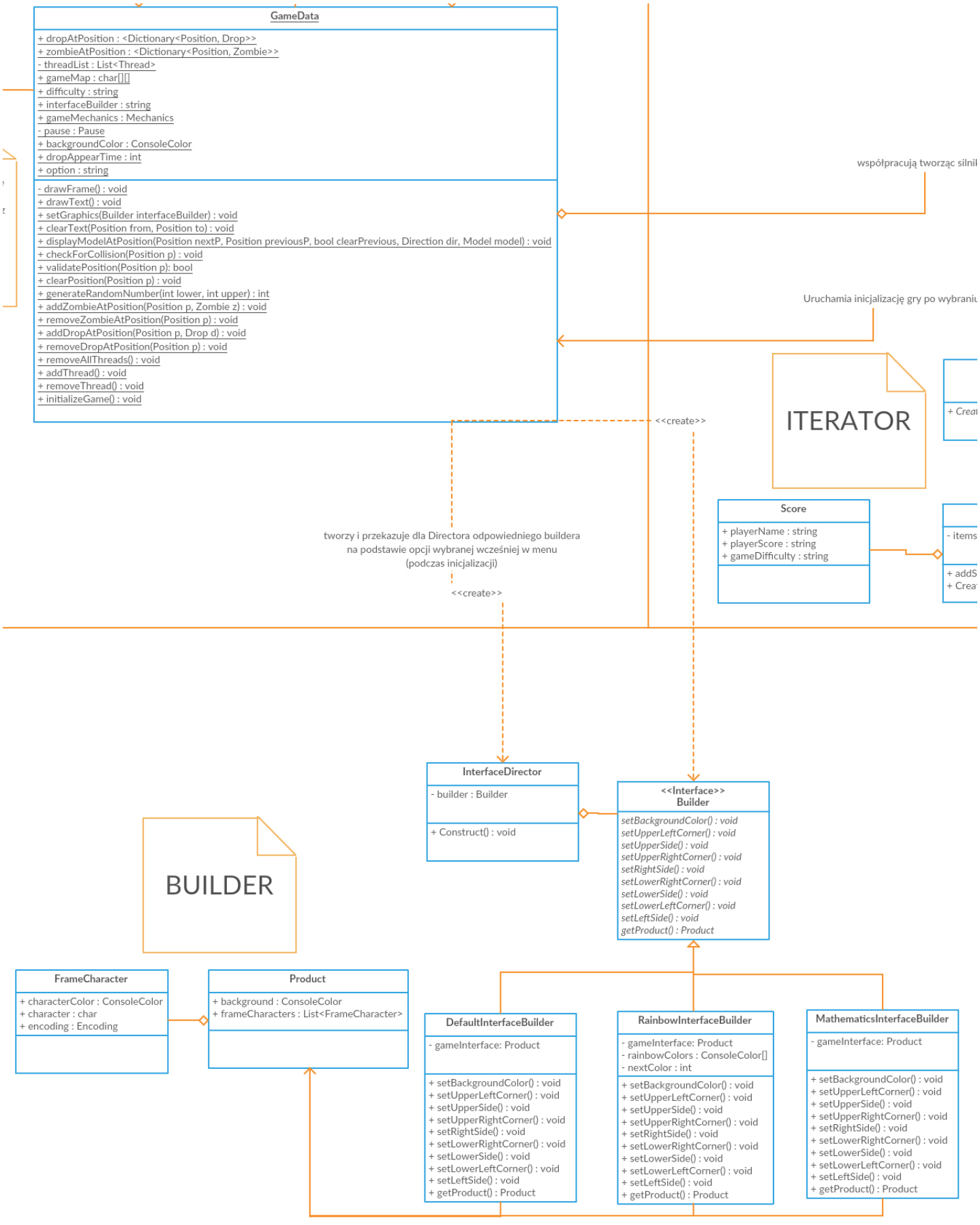
16.01.2019

Zaawansowane Techniki Programistyczne, semestr V, zima 2019, Wydział Informatyki, Politechnika Białostocka

1. Diagram klas.

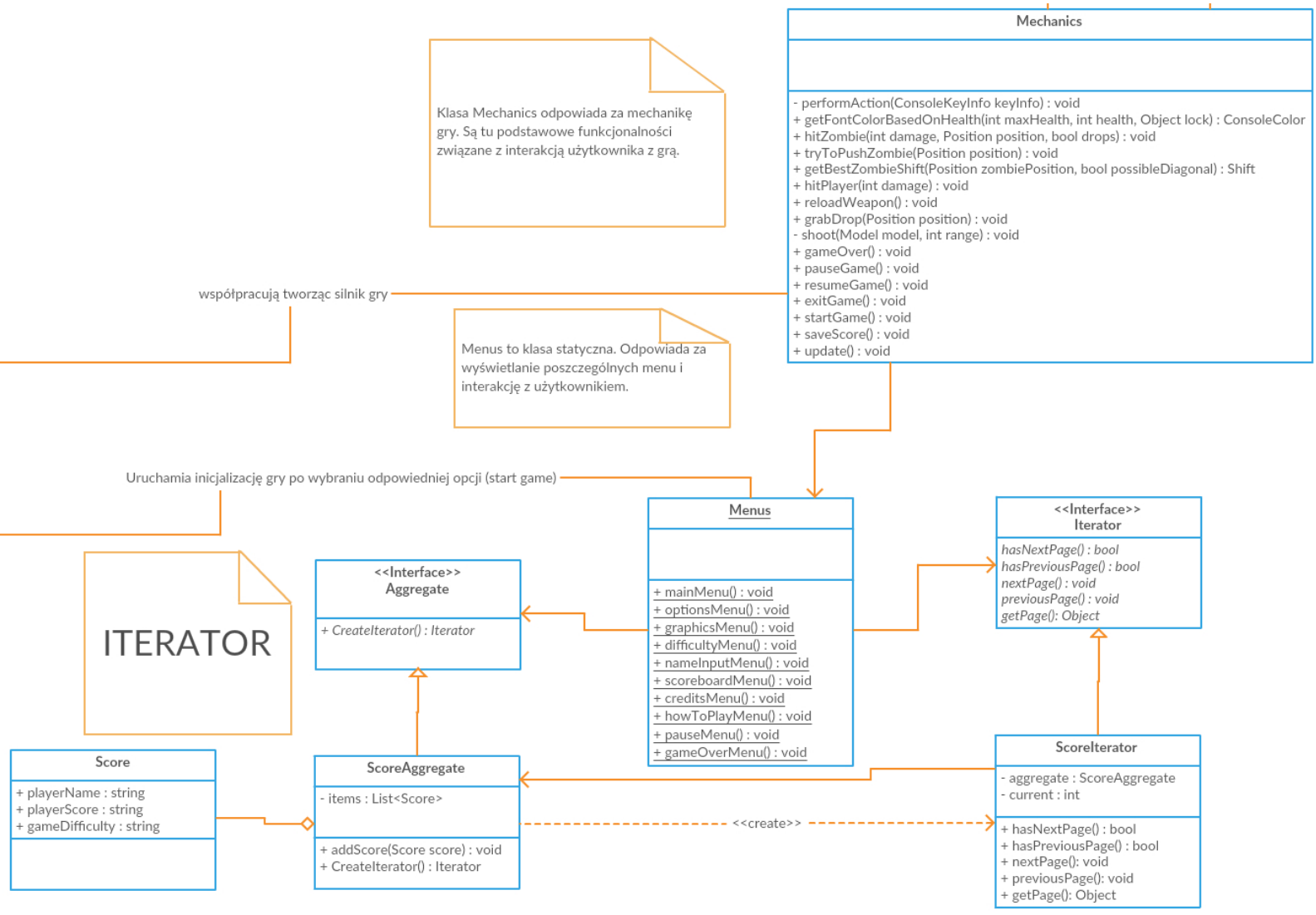
Cały diagram klas zamieszczamy w folderze z dokumentacją.

GameData to najważniejsza klasa w programie. Udostępnia informacje dla innych klas. Posiada dane o modelach występujących w grze, wątkach, obiektach synchronizujących pracę wątków, metody dostępowe do konsoli, itp. Współpracuje z klasą *Mechanics* tworząc silnik gry. Odpowiada za inicjalizację gry, podczas której m.in. tworzy obiekt poziomu trudności oraz builder'a interfejsu gry, wykorzystując informacje o ustawieniach wybranych w menu głównym przez użytkownika.

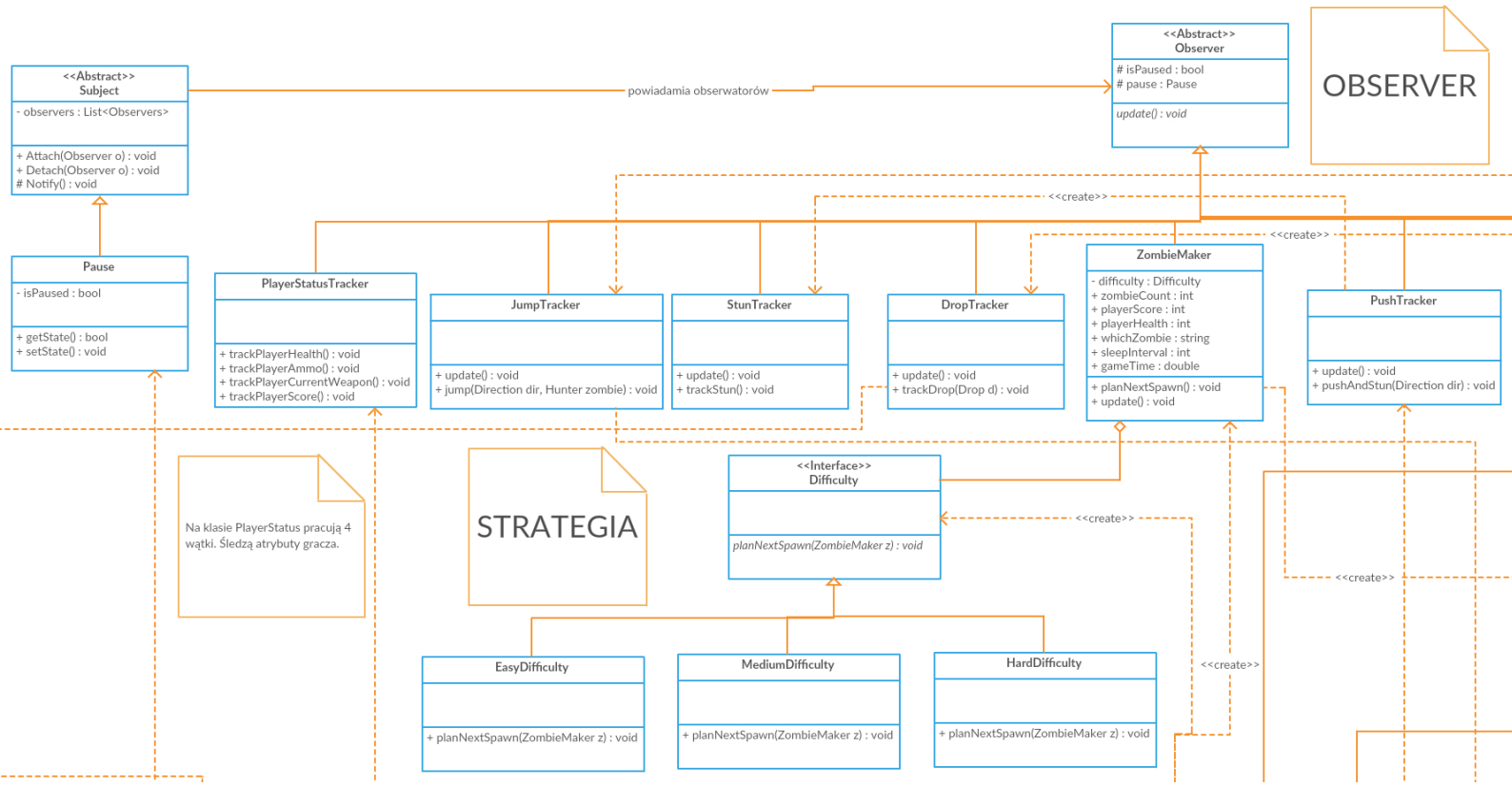

GameData
+ dropAtPosition : <Dictionary<Position, Drop>>
+ zombieAtPosition : <Dictionary<Position, Zombie>>
- threadList : List<Thread>
+ gameMap : char[][]
+ difficulty : string
+ interfaceBuilder : string
+ gameMechanics : Mechanics
- pause : Pause
+ backgroundColor : ConsoleColor
+ dropAppearTime : int
+ option : string
- drawFrame() : void
+ drawText() : void
+ setGraphics(Builder interfaceBuilder) : void
+ clearText(Position from, Position to) : void
+ displayModelAtPosition(Position nextP, Position previousP, bool clearPrevious, Direction dir, Model model) : void
+ checkForCollision(Position p) : void
+ validatePosition(Position p): bool
+ clearPosition(Position p) : void
+ generateRandomNumber(int lower, int upper) : int
+ addZombieAtPosition(Position p, Zombie z) : void
+ removeZombieAtPosition(Position p) : void
+ addDropAtPosition(Position p, Drop d) : void
+ removeDropAtPosition(Position p) : void
+ removeAllThreads() : void
+ addThread() : void
+ removeThread() : void
+ initializeGame() : void
współpracują tworząc silnik
Uruchamia inicjalizację gry po wybraniu
<<create>>
ITERATOR
Score
+ playerName : string
+ playerScore : string
+ gameDifficulty : string
tworzy i przekazuje dla Directora odpowiedniego buildera
na podstawie opcji wybranej wcześniej w menu
(podczas inicjalizacji)
<<create>>
InterfaceDirector
- builder : Builder
+ Construct() : void
<<Interface>>
Builder
setBackgroundColor() : void
setUpperLeftCorner() : void
setUpperSide() : void
setUpperRightCorner() : void
setRightSide() : void
setLowerRightCorner() : void
setLowerSide() : void
setLowerLeftCorner() : void
setLeftSide() : void
getProduct() : Product
BUILDER
FrameCharacter
+ characterColor : ConsoleColor
+ character : char
+ encoding : Encoding
Product
+ background : ConsoleColor
+ frameCharacters : List<FrameCharacter>
DefaultInterfaceBuilder
- gameInterface: Product
+ setBackgroundColor() : void
+ setUpperLeftCorner() : void
+ setUpperSide() : void
+ setUpperRightCorner() : void
+ setRightSide() : void
+ setLowerRightCorner() : void
+ setLowerSide() : void
+ setLowerLeftCorner() : void
+ setLeftSide() : void
+ getProduct() : Product
RainbowInterfaceBuilder
- gameInterface: Product
- rainbowColors : ConsoleColor[]
- nextColor : int
+ setBackgroundColor() : void
+ setUpperLeftCorner() : void
+ setUpperSide() : void
+ setUpperRightCorner() : void
+ setRightSide() : void
+ setLowerRightCorner() : void
+ setLowerSide() : void
+ setLowerLeftCorner() : void
+ setLeftSide() : void
+ getProduct() : Product
MathematicsInterfaceBuilder
- gameInterface: Product
+ setBackgroundColor() : void
+ setUpperLeftCorner() : void
+ setUpperSide() : void
+ setUpperRightCorner() : void
+ setRightSide() : void
+ setLowerRightCorner() : void
+ setLowerSide() : void
+ setLowerLeftCorner() : void
+ setLeftSide() : void
+ getProduct() : Product

Klasa *Mechanics* odpowiada za mechanikę gry. Działa ona tylko podczas rozpoczętej gry. Dzięki niej gracz wchodzi w interakcję z otoczeniem podczas rozgrywki. Korzystają z niej również zombie.

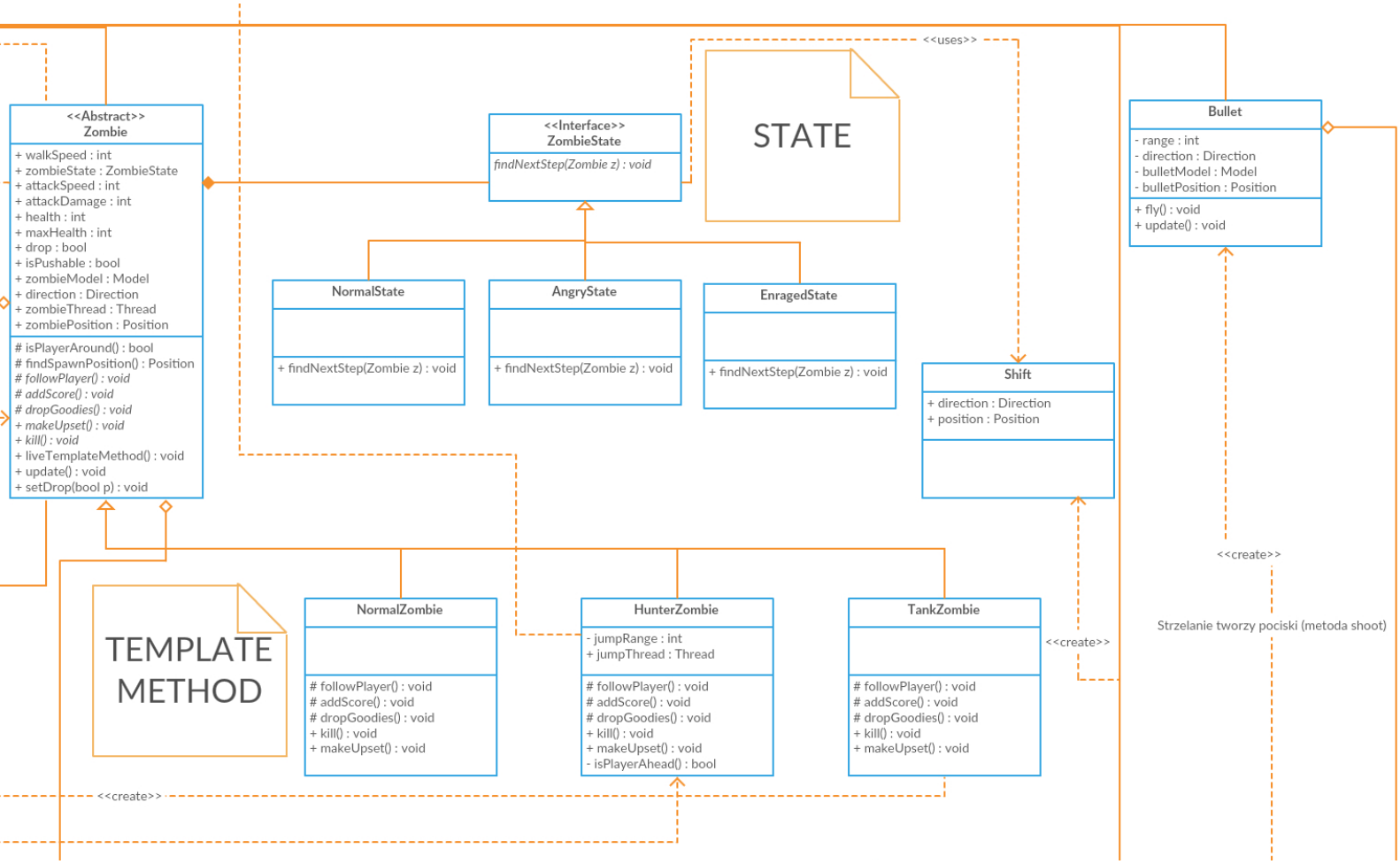
Klasa *Menus* posiada metody umożliwiające poruszanie się po menu gry. Dostępne są opcje graficzne, poziomu trudności, samouczek, tabela wyników, a także menu, które można wywoływać podczas rozgrywki (np. menu pauzy) – wtedy takie menu wywołuje podczas gry gracz z klasy Mechanics. Po wybraniu przycisku „START GAME", wywołuje metodę inicjalizującą grę z klasy *GameData*.



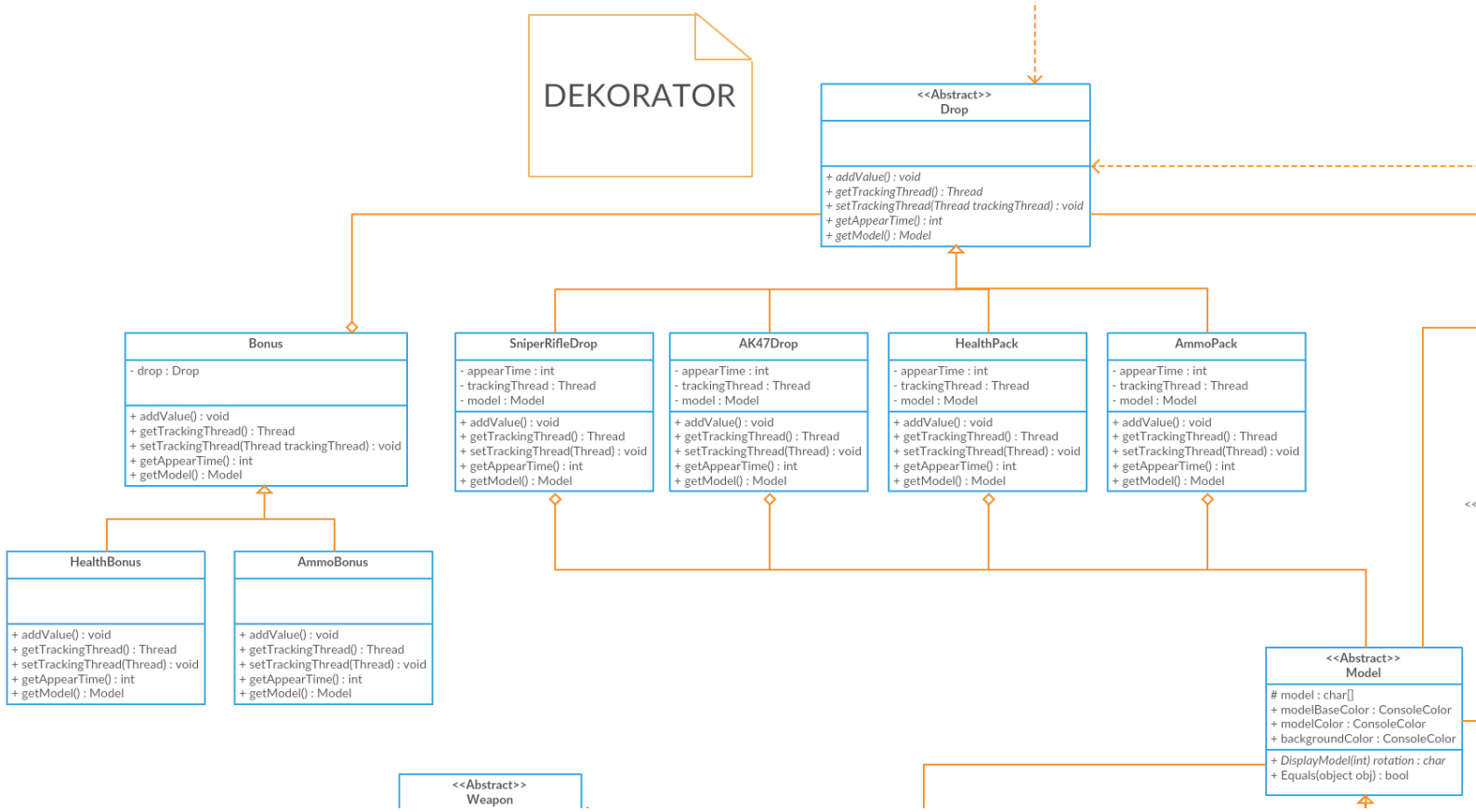
GameData daje również początek obiektowi klasy *Pause* oraz *PlayerStatusTracer*. *Pause* odpowiada za informowanie wątków o wstrzymaniu rozgrywki. *PlayerStatusTracker* śledzi atrybuty gracza, takie jak amunicja, punkty zdrowia, itp. *ZombieMaker* tworzy zombie, jednak wybór tworzonego przeciwnika ustalany jest na podstawie wybranego poziomu trudności (*Difficulty)*. Trackery to klasy, na które kontrolują stan obiektów w rozgrywce. Przykładowo *StunTracker* odpowiada za czas ogłuszenia gracza.



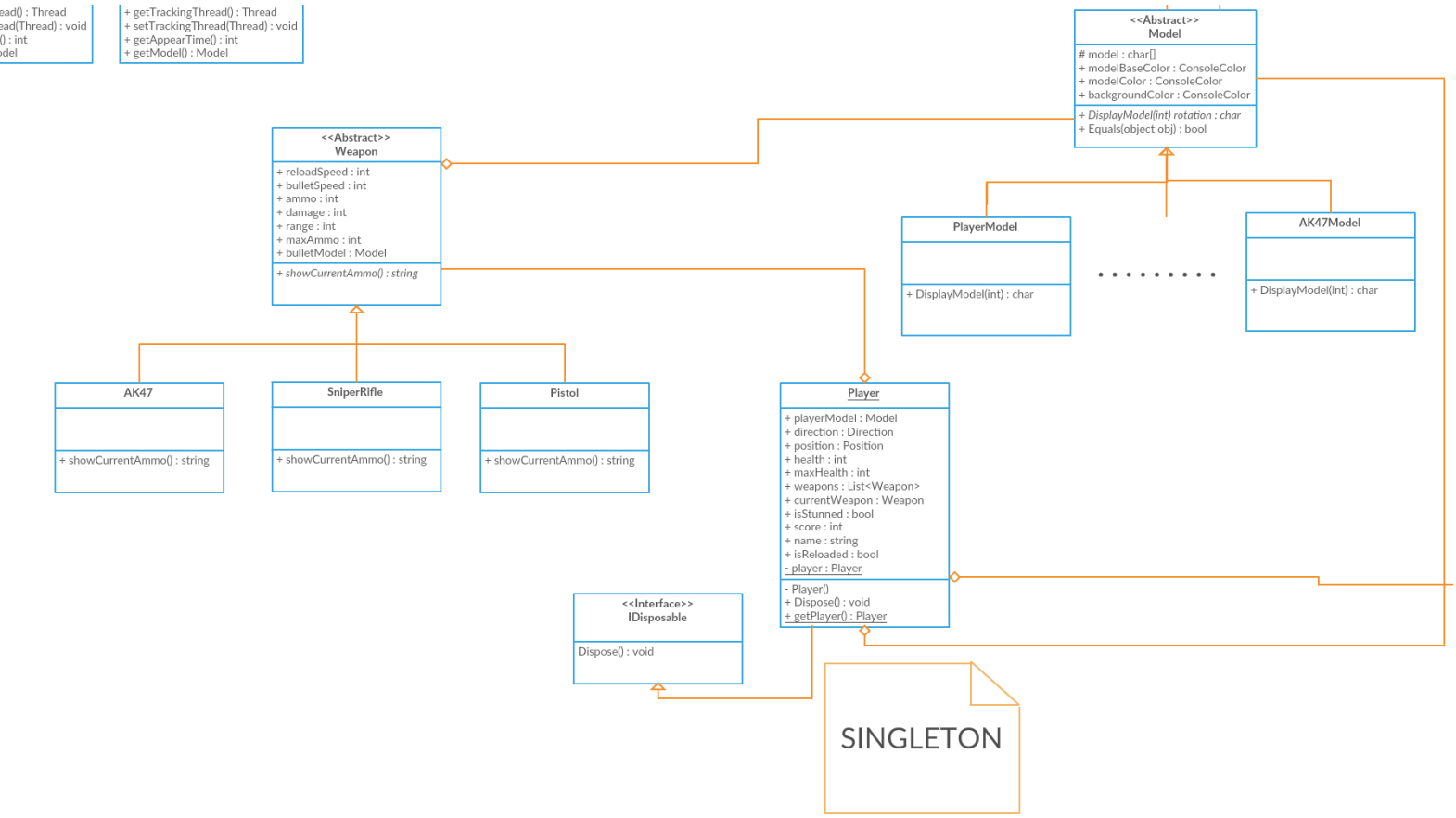
W grze występują trzy rodzaje zombie. Dziedziczą po klasie abstrakcyjnej *Zombie*. Każdy zombie ma inne umiejętności. O zachowaniu przeciwników decyduje *ZombieState*. Stan potworów zmienia się w zależnosci od ich punktów zdrowia. *Shift* to przesunięcie, kolejna pozycja zombie. Jest częścią logiki śledzenia postaci gracza przez zombie. *Bullet* to klasa reprezentująca pociski wylatujące z broni gracza.



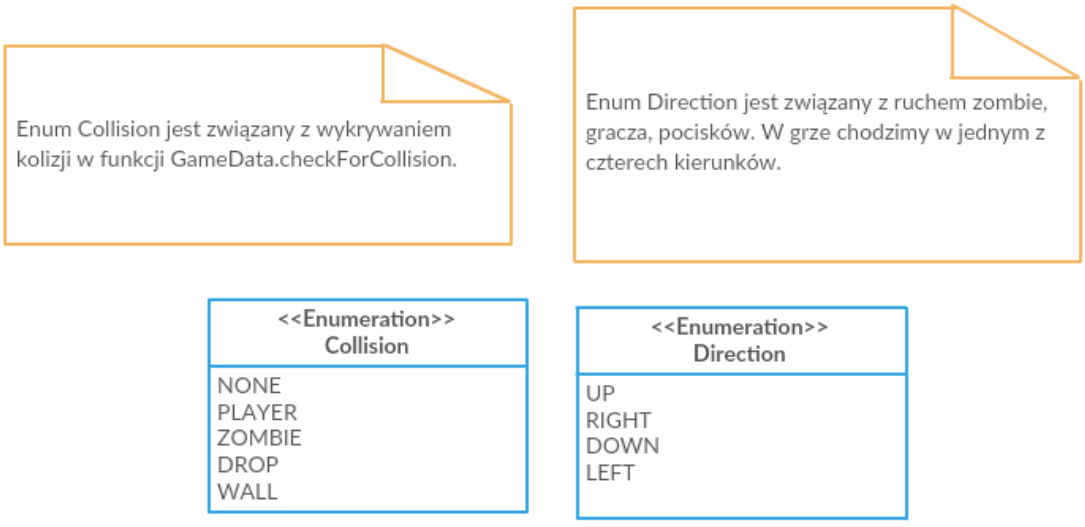
Abstrakcyjna klasa *Drop* symbolizuje przedmioty upuszczane przez zombie po ich śmierci. Mogą upuszczać bronie oraz paczki zdrowia i amunicji. Przedmioty moga posiadać bonusy. Klasa *Model* przechowuje tekstową reprezentację obiektu pojawiającego się w konsoli podczas rozgrywki wraz z kolorem tła i czcionki rysowanego obiektu.



W grze mamy do wyboru trzy bronie (klasa *Weapon*). Są to: pistolet, AK47 i snajperka. Klasa *Player* posiada dane o graczu takie jak życie, amunicja, obecnie zaekwipowana broń, itp.



Typy wyliczeniowe występujące w programie:

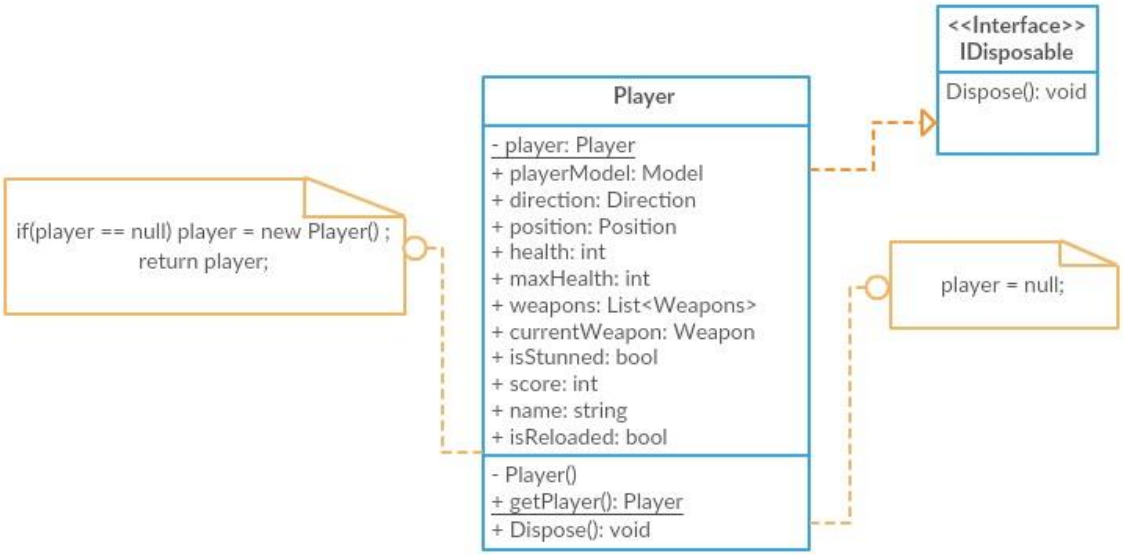


2. Opis użytych wzorców.

SINGLETON

Cel użycia: Jeden obiekt gracza na całą rozgrywkę. Obiekt gracza powinien być tworzony dopiero w momencie rozpoczęcia rozgrywki.

- Klasa *Player* to singleton.
- Zastosowaliśmy późną inicjalizację. Inicjalizacja gry jako pierwsza wywołuje *GetPlayer()*.
- Statyczna metoda *GetPlayer()* posiada mechanizm synchronizacji – dostęp do gracza ma wiele wątków.
- Po zakończeniu rozgrywki ustawiamy referencję *player* na null. Zaimplementowaliśmy do tego interfejs IDisposable (metoda *Dispose*) wystepujący w .NET.



BUILDER

Cel użycia: użytkownik powinien móc wybrać odpowiadający mu wygląd interfejsu w opcjach. Struktura interfejsu podczas rozgrywki jest niezmienna. Zmianie ulegać powinna jedynie kolorystyka tła oraz wygląd obramowania mapy.

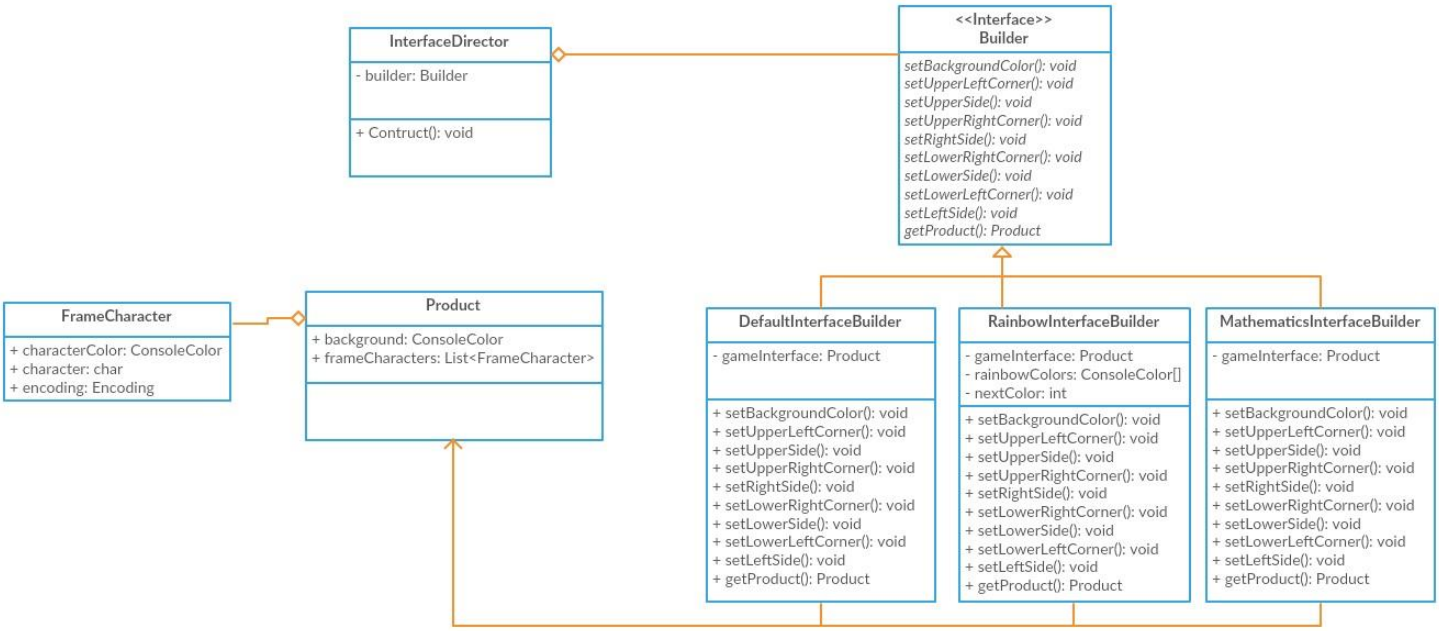
- *InterfaceDirector* generuje scenariusz tworzenia obiektu.

- *InterfaceBuilder* deklaruje metody tworzące produkt.

- *DefaultInterfaceBuilder*, *RainbowInterfaceBuilder*, *MathematicsInterfaceBuilder* to konkretni budowniczy. Każdy z nich jest spokrewniony, służą do tego samego.

- Produkt składa się z koloru tła oraz znaków budujących ramkę.

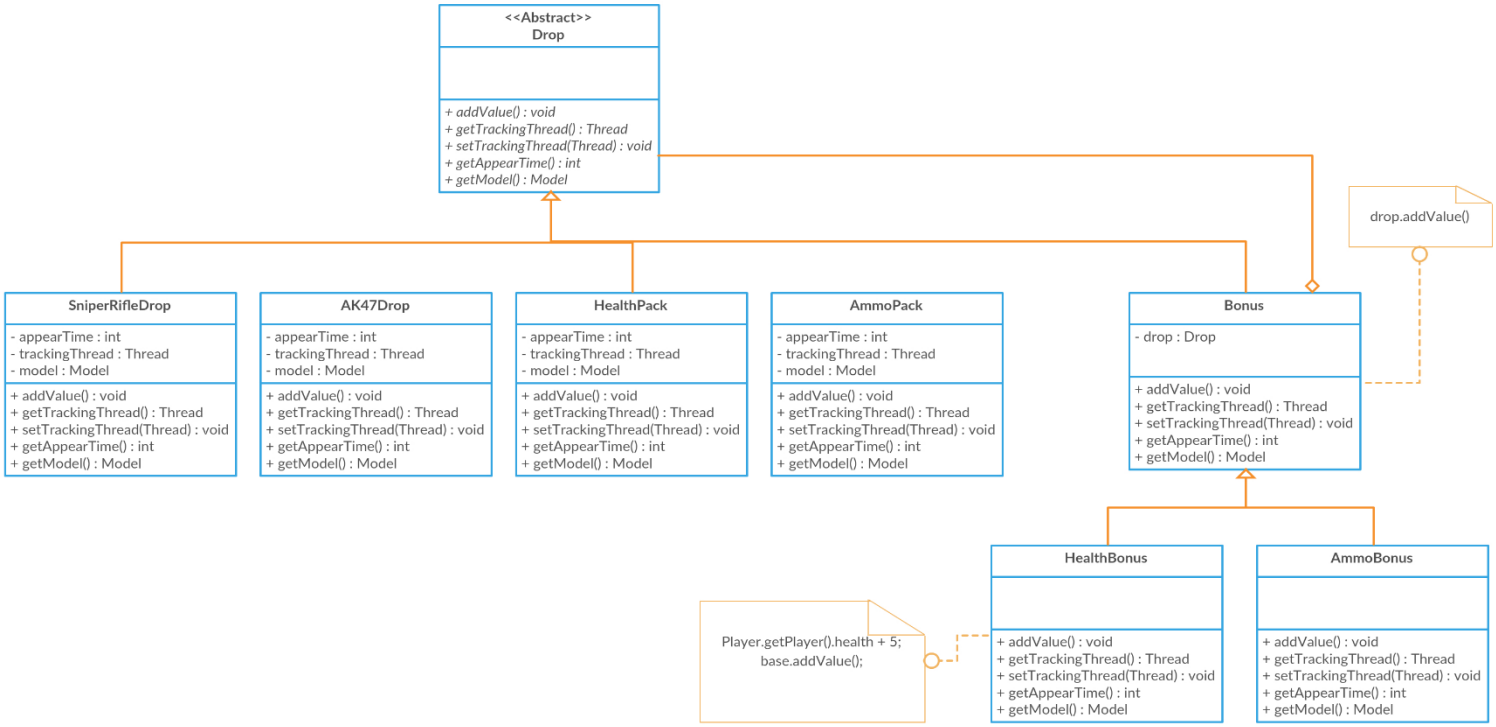
- Ramka składa się ze znaków o określonym kolorze oraz kodowaniu. Przykładowo, domyślnie ramka jest zrobiona z poprzeczek w formacie ASCII. Z drugiej strony *RainbowInterfaceBuilder* oferuje kolorową ramkę w formacie Unicode



DEKORATOR

Cel użycia: tworzenie przedmiotów upuszczanych przez zombie posiadających różne atrybuty. Liczy się elastyczność i innowacja. Zupełna dowolność w tworzeniu przedmiotów.

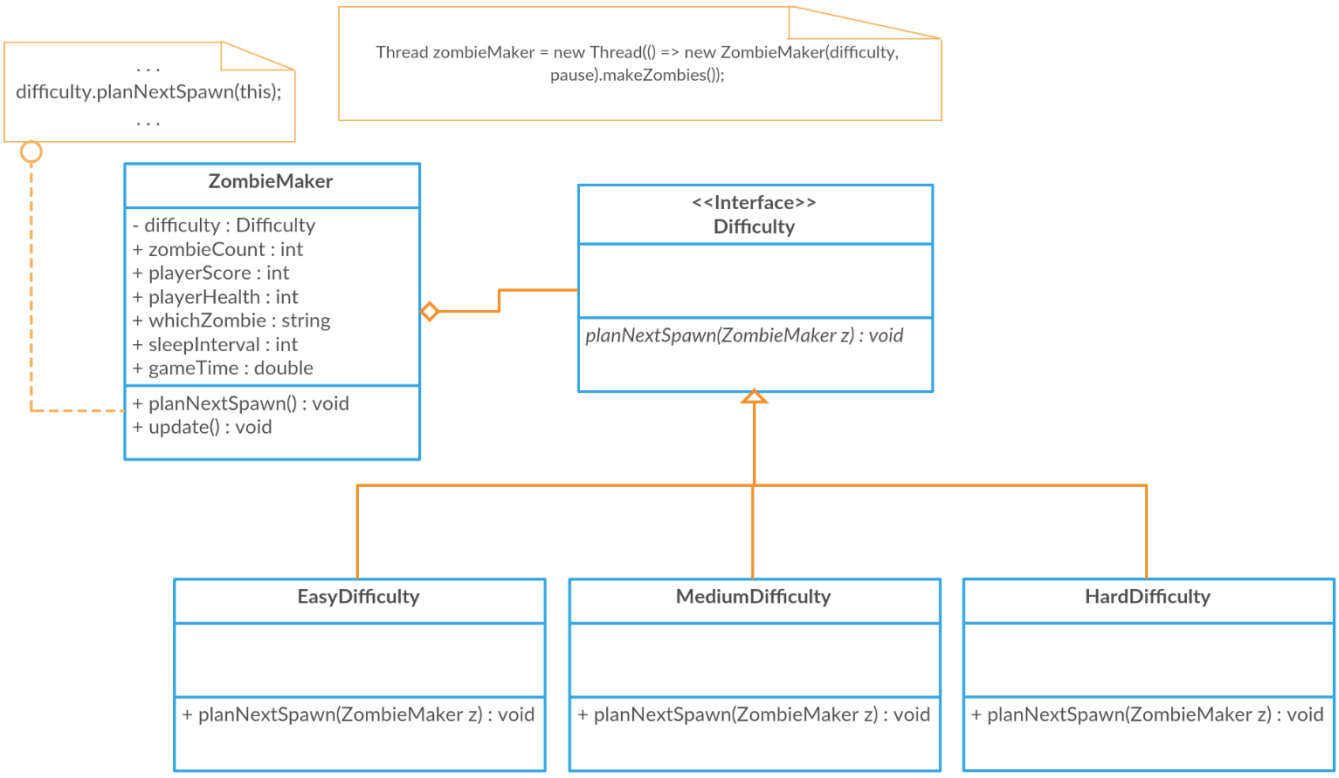
- *Drop* to komponent, deklaruje wspólne metody dla dekoratora oraz fizycznych obiektów.
- *SniperRifleDrop, AK47Drop, HealthPack, AmmoPack* to fizyczne przedmioty (obiekty).
- *Bonus* to dekorator. Opakowujemy w niego przedmioty.
- Metoda *AddValue()* dodaje opowiedni atrybut dla gracza. Mogą to być np. punkty zdrowia.
- Gracz będzie rozróżniał modele na planszy dzięki *GetModel()* – bonusy modyfikują wygląd przedmiotu.



STRATEGIA

Cel użycia: gracz ma do wyboru trzy poziomy trudności. Różne strategie pojawiania się zombie w zależności od atrybutów gracza. Manipulacja liczbami pseudolosowymi.

- *ZombieMaker* to kontekst. Odpowiada za tworzenie konkretnych zombie.
- Kod wyboru zombie wydzielony do klas *EasyDifficulty*, *MediumDifficulty*, *HardDifficulty* – *PlanNextSpawn()*.
- Zastosowaliśmy metodę pull.
- *EasyDifficulty* – na mapie pojawiają się tylko *NormalZombie*, stały interwał czasu pomiędzy pojawieniami.
- *MediumDifficulty* – na mapie pojawiają się *NormalZombie* i *TankZombie*, losowy interwał (niezależny od wyniku i zycia gracza), wybór zombie na podstawie wyniku gracza.
- *HardDifficulty* – na mapie pojawiają się *NormalZombie*, *TankZombie* i *HunterZombie*, interwał zależny od wyniku gracza, wybór zombie zależny od punktów zdrowia i wyniku gracza.

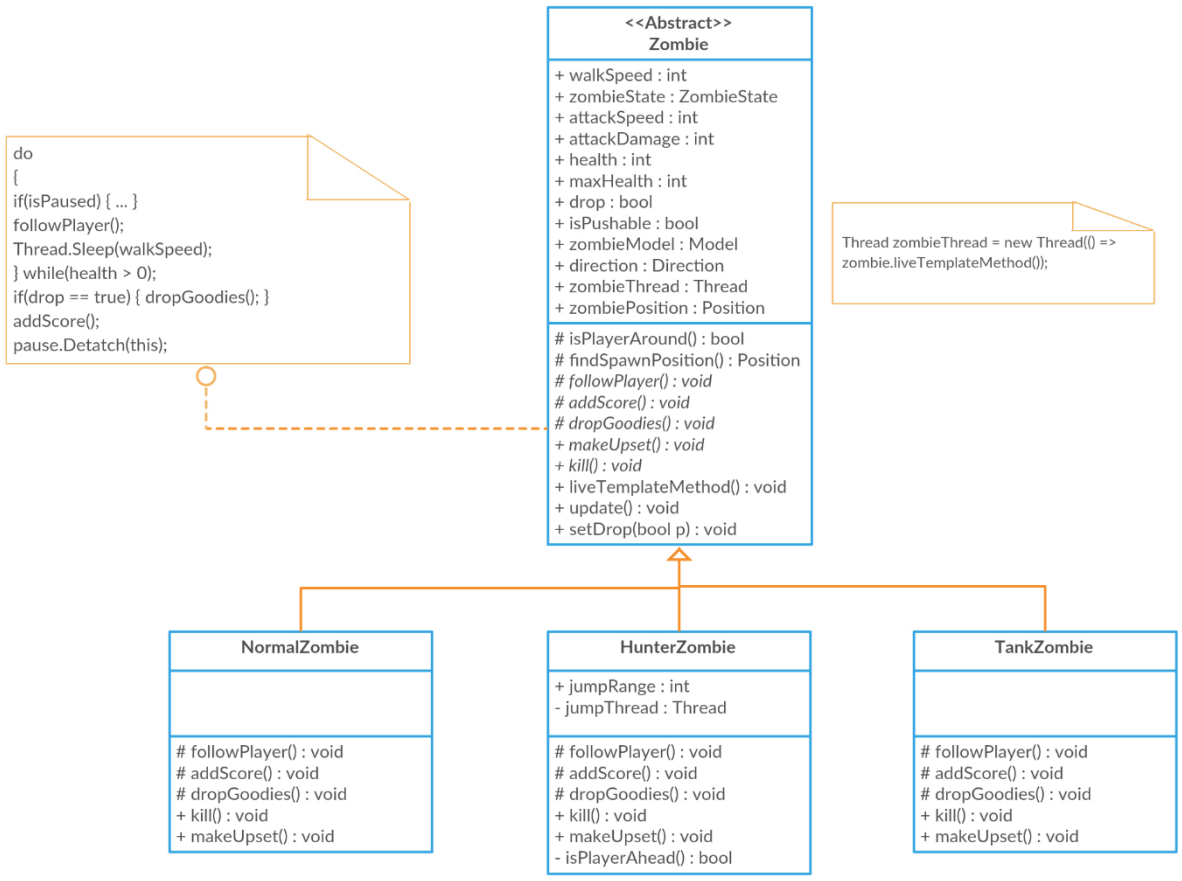


TEMPLATE METHOD

Cel użycia: zachowanie każdego zombie jest unikalne. Każdy z nich dąży do tego samego – chce zabić gracza. Cykl życia zombie jest jednakowy.

- *Zombie* to abstrakcyjna klasa zawierająca metodę szablonową *LiveTemplateMethod()*.

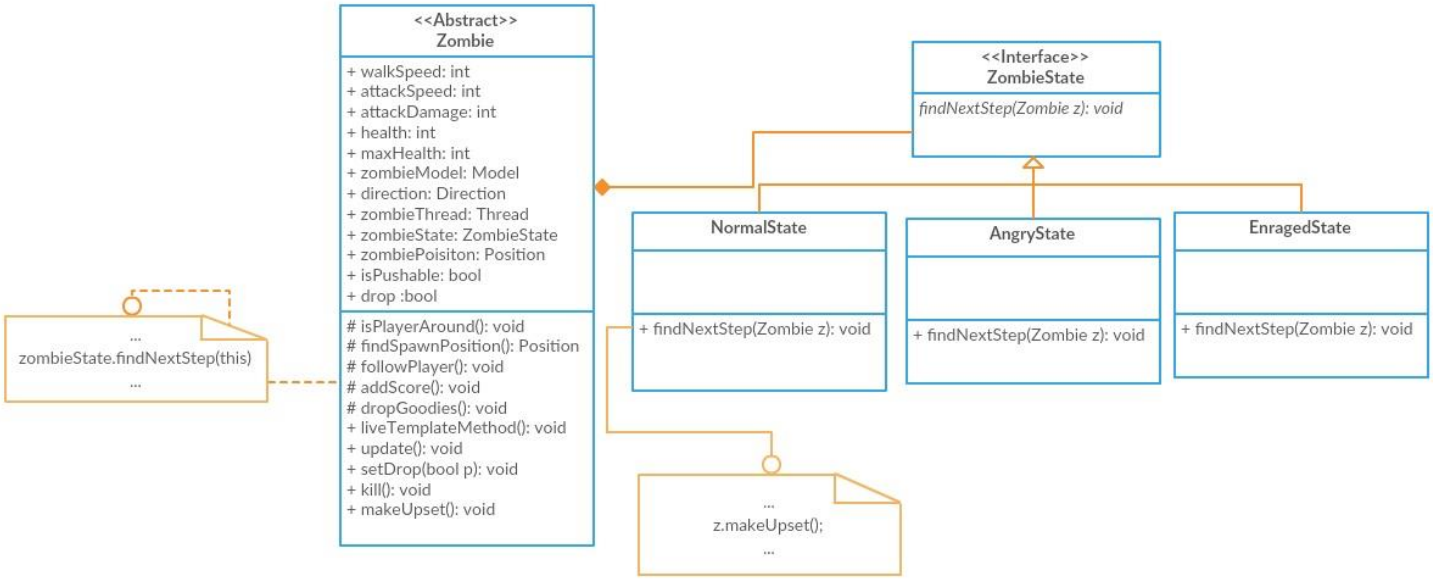
- Abstrakcyjne metody nadpisywane w klasach dziedziczących to: *FollowPlayer()*, *DropGoodies()*, *AddScore()*. Każdy zombie wchodzi w interakcję z graczem inaczej, upuszcza inne przedmioty, jest warty różną liczbę punków po zabiciu.



STAN

Cel użycia: zmiana zachowania zombie pod wpływem utraty zdrowia.

- Interfejs *ZombieState* deklaruje metodę *FindNextStep()*, odpowiadającą za szukanie kolejnego kroku w celu odnalezienia gracza.
- *NormalState*, *AngryState*, *EnragedState* to konkretne stany.
- Wewnątrz metody *FollowPlayer()*(wywołuje się ona w metodzie szablonowej) klasy *Zombie* znajduje się wywołanie metody *FindNextStep()*.
- Właściwa zmiana stanu następuje w obiekcie typu *Zombie* (obiekcie kontekstu). Jeżeli życie zombie spadnie poniżej 50%, wywołuje się metoda *MakeUpset()* w klasie *NormalState* zmieniająca sposób szukania gracza oraz atrybuty zombie.
- *HunterZombie* i *NormalZombie* mogą przejść tylko do stanu *AngryState*. Tylko *TankZombie* przechodzi do stanu *EnragedState*.



OBSERVER

Cel użycia: wszystko jest wątkiem. Umożliwienie wstrzymywania i wznawiania rozgrywki w dowolnym momencie. Mechanizm pauzy.

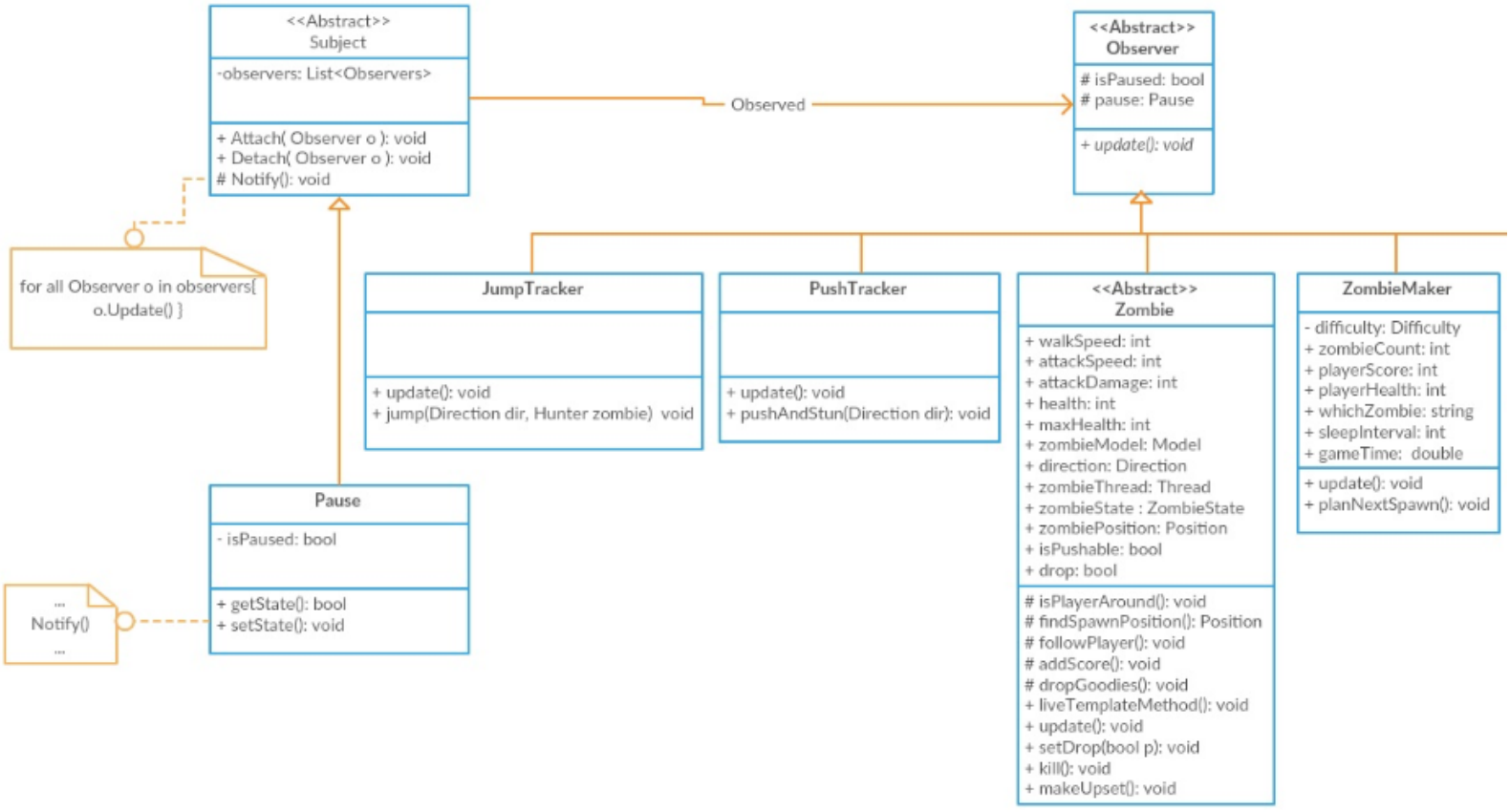
- Abstrakcyjna klasa *PauseObserver* reprezentuje wątki.

- Abstrakcyjna klasa *PauseSubject* udostępnia metody umożliwiające rejestrację i wyrejestrowanie się.

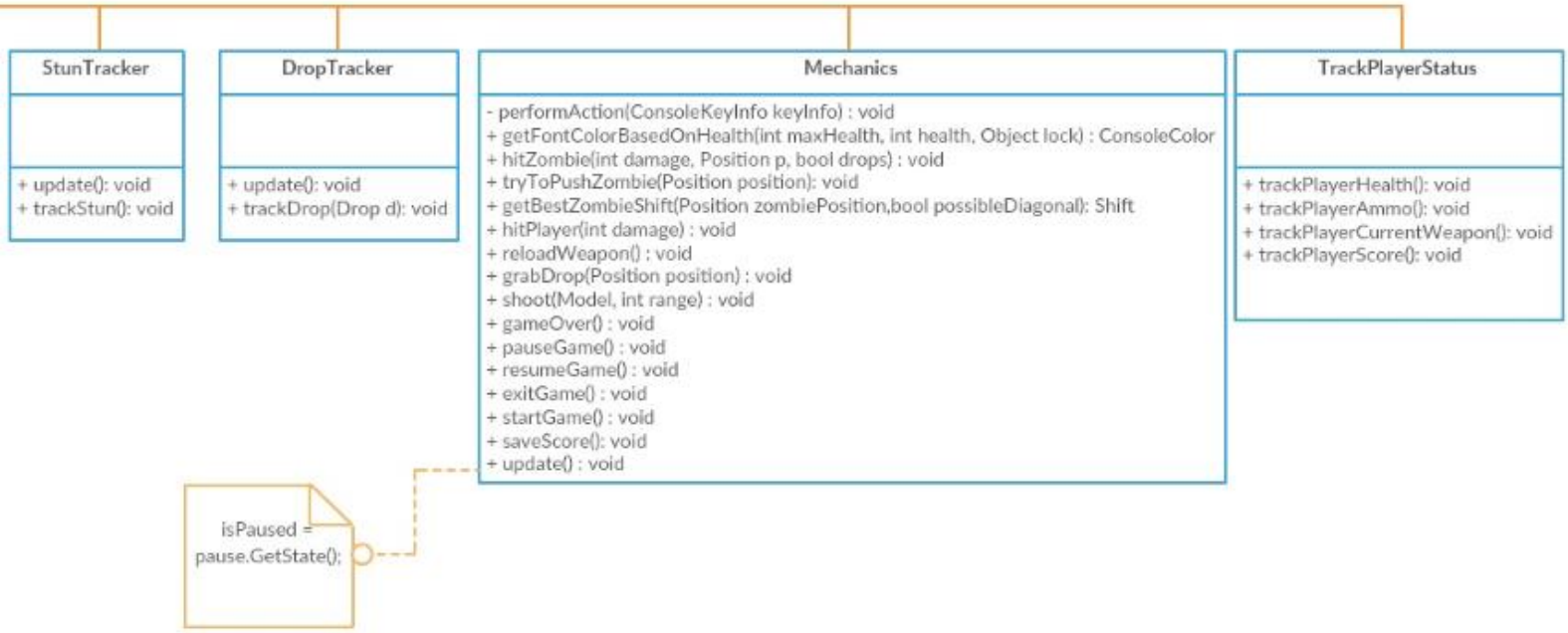
- Obserwujący przechowują flagę informującą o pauzie. Zawierają referencję do obiektu obserwowanego.

- Automatyczne wywołanie *Notify()* przy zmianie stanu obiektu. Obserwatorzy powiadamiani dzięki metodzie *Update()*.

- Zmianę stanu obiektu obserwowanego powoduje metoda *PauseGame()* klasy *Mechanics*. Zatem obiekt obserwowany jest odpowiedzialny za zmianę stanu.



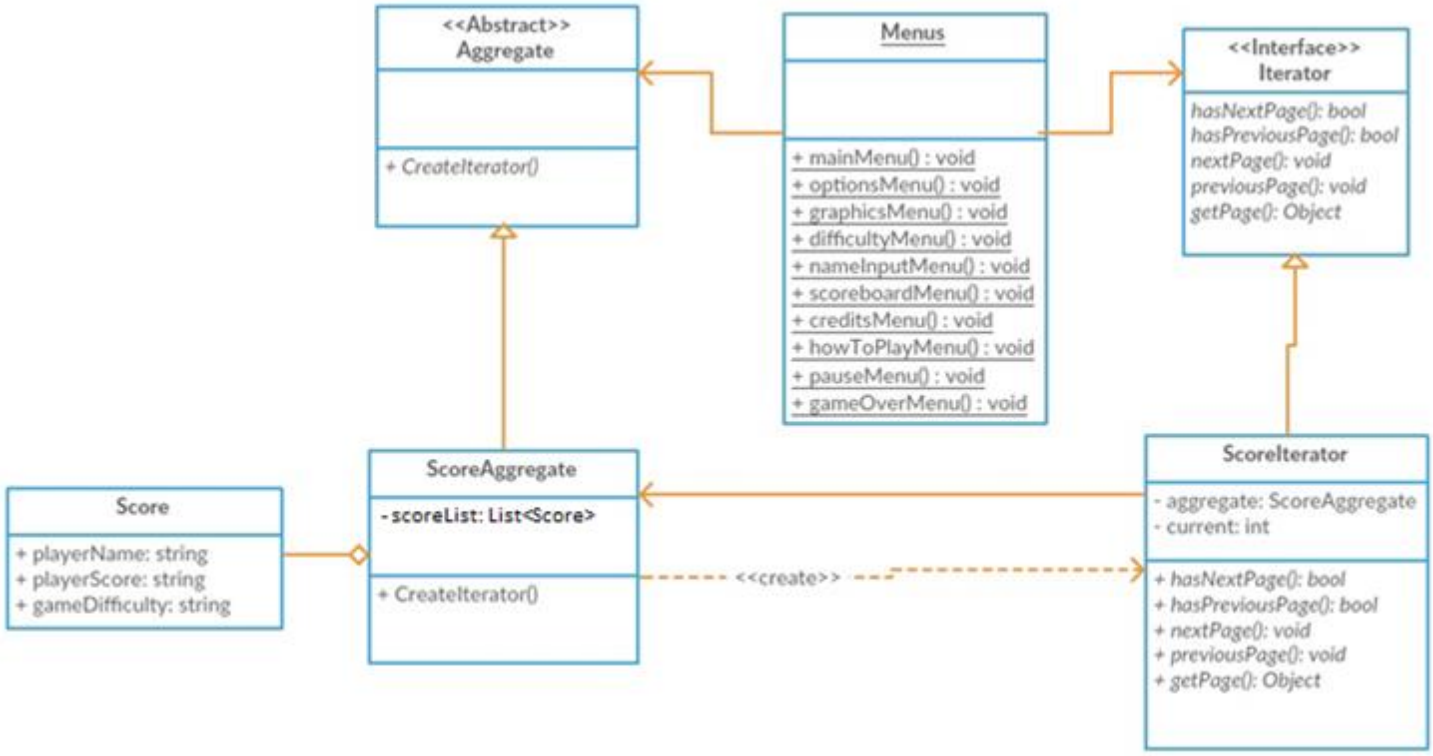
Poniższe klasy to też obserwatorzy:



ITERATOR

Cel użycia: lista wyników graczy. Okno konsoli nie mieści pełnej liczby wyników w miarę wzrostu liczby rozgrywek. Potrzeba swobodnego chodzenia po liście.

- Interfejs *IPageIterator* deklaruje metody do stronnicowania listy wyników.
- Iterator zewnętrzny.
- *ScoreIterator* implementuje funkcjonalności dla struktury *ScoreAggregate*.
- *ScoreAggregate* implementuje interfejs *IAggregate*. *CreateIterator()* tworzy i zwraca *IPageIterator*.
- Lista wyników składa się z obiektów klasy *Score*. Pojedynczy wynik gry zawiera: imię gracza, liczba punktów, poziom trudności.
- Na pojedynczej stronie może znajdować się maksymalnie 10 wyników. Metoda *GetPage()* zwraca 10 wyników lub mniej.



3. Użycie rozwiązania specyficznego dla wykorzystywanej technologii.

W programie znajduje się dosyć dużo kolekcji. Wykorzystujemy słowniki (hashmapy) do mapowanie pozycji na mapie na konkretnego zombie. To samo robimy z przedmiotami upuszczanymi przez przeciwników. Gracz zawiera listę broni, które obecnie posiada i na której musimy pracować podczas interakcji z upuszczanymi przedmiotami.

Dosyć duży problem stanowiło poruszanie się po kolekcjach. Przykładowo pocisk, który trafia zombie, musi pobrać go ze słownika i zabrać mu punkty zdrowia. Problem w tym, że pocisk i zombie mają własne pozycje, a słownik porównuje klucze po referencjach. Możliwe jest nadpisanie komparatora *IEqualityComparer* tak, aby porównywał klucze po współrzędnych pozycji, a nie po referencjach. Problem w tym, że implementując taki komparator należy nadpisać metodę *GetHashCode()*. O tym nie będziemy się rozpisywać, ważne jest jedno – nie należy modyfikować obiektów kluczy – zmienia się wtedy ich hashcode i porównania stają się nieprawidłowe, a pozycje zombie nieustannie się zmieniają.

Innym problemem było zrelizowanie algorytmu metody *AddValue()* klasy *Drop*. Przykładowo, jeżeli gracz podnosi amunicję, musimy najpierw sprawdzić czy posiada bronie inne niż pistolet (sam pistolem ma nieskończoność amunicji), aby dodać do tej broni naboje. Wszystko sprowadza się do tego, że poruszając się po kolekcjach musielibyśmy pisać skomplikowane pętle z instrukcjami warunkowymi.

Zintegrowany język zapytań (Language Integrated Query, **LINQ**) w .NET oferuje elastyczne rozwiązanie tego problemu.
Źródło: https://docs.microsoft.com/pl-pl/dotnet/csharp/programming-guide/concepts/linq/getting-started-with-linq.
Najpierw pobieramy pozycję z kolekcji kluczy słownika, a następnie pobieramy konkretnego zombie. Klucze porównujemy przekazując predykat (wnętrze metody *Where*).

```
3118    public bool hitZombie(Position position, int damage, bool drops = true)
3119    {
3120        Zombie zombie;
3121        lock (GameData.consoleAccessObject)
3122        {
3123            lock(GameData.zombieAtPositionDictionaryAccess)
3124            {
3125                position = GameData.zombieAtPosition.Keys.Where(p => p.Equals(position)).FirstOrDefault();
3126                zombie = GameData.zombieAtPosition[position];
3127            }
3128            lock (GameData.zombieHealthAccess)
3129            {
3130                zombie.health -= damage;
3131                zombie.zombieModel.modelColor = getFontColourBasedOnHealth(zombie.health, zombie.maxHealth, GameData.zombieHealthAccess);
3132                GameData.displayModelAtPosition(position, position, zombie.zombieModel, zombie.direction);
3133                if (zombie.health <= 0)
3134                {
3135                    zombie.setDrop(drops);
3136                    GameData.removeZombieAtPosition(position);
3137                    return true;
3138                }
3139            }
3140            return false;
3141        }
3142    }
```

Fragment metody *AddValue()*. Podnoszenie paczki z amunicją. Przed dodaniem amunicji do broni, sprawdzamy czy gracz ją posiada (*Exists)*, następnie pobieramy ją z kolekcji (*Find*) i dodajemy naboje:

```
if (Player.getPlayer().weapons.Exists(w => w.ToString() == "AK47"))
{
    Weapon ak47 = Player.getPlayer().weapons.Find(w => w.ToString() == "AK47");
    if (ak47.ammo + 3 <= ak47.maxAmmo)
    {
        ak47.ammo += 3;
    }
    else
    {
        ak47.ammo = ak47.maxAmmo;
    }
}
else if (Player.getPlayer().weapons.Exists(w => w.ToString() == "Sniper Rifle"))
{
    Weapon sniperRifle = Player.getPlayer().weapons.Find(w => w.ToString() == "Sniper Rifle");
    if (sniperRifle.ammo + 1 <= sniperRifle.maxAmmo)
    {
        sniperRifle.ammo += 1;
    }
    else
    {
        sniperRifle.ammo = sniperRifle.maxAmmo;
    }
}
```

Dzięki LINQ uzyskaliśmy elastyczność operowania na kolekcjach oraz oszczędność kodu.

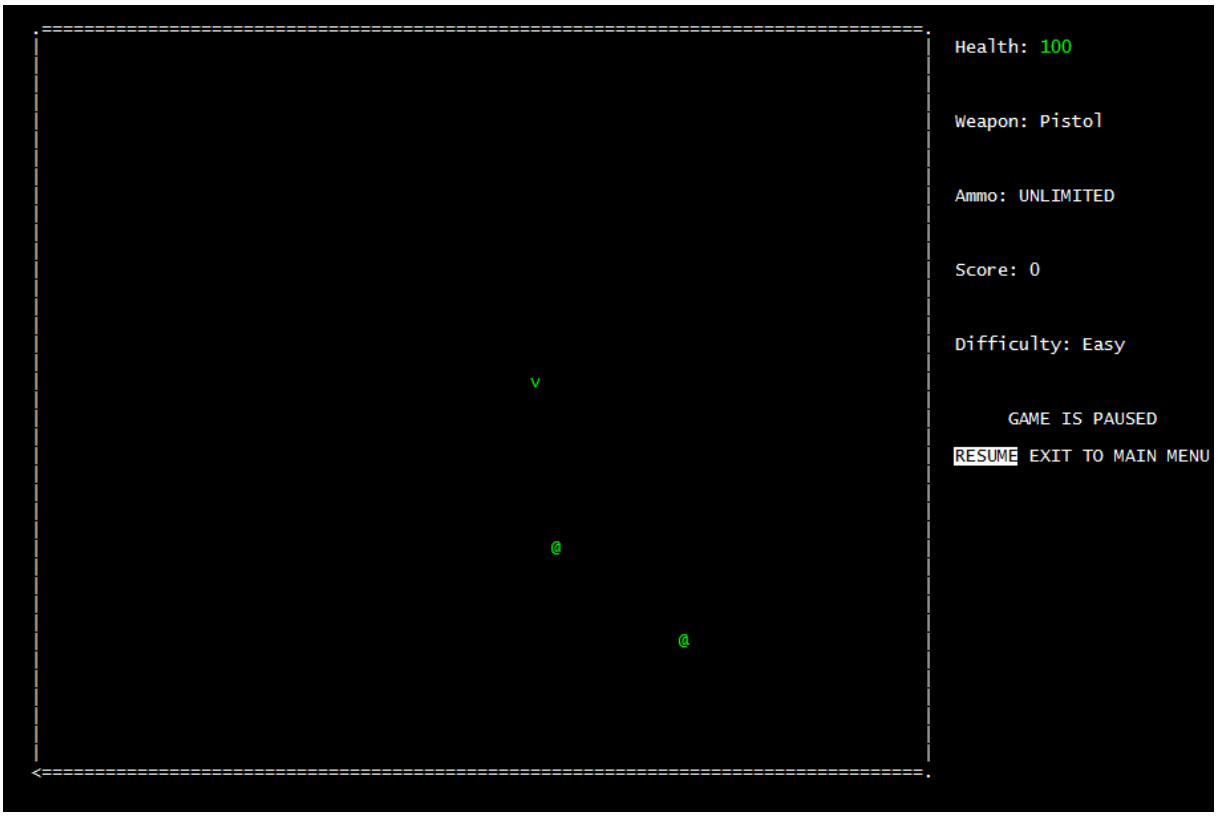
5. Krótka instrukcja użytkownika.

Menu główne:

- Start: rozpoczęcie rozgrywki, wprowadzenie imienia gracza.
- Scoreboard: tabela wyników.
- How To Play: krótki samouczek.
- Options: wybór poziomu trudności i opcji graficznych.
- Credits: informacje od twórców gry.
- Quit: wyjście z programu.

Po menu poruszamy się strzałkami. Aby powrócić do poprzedniego okna należy wcisnąć escape.

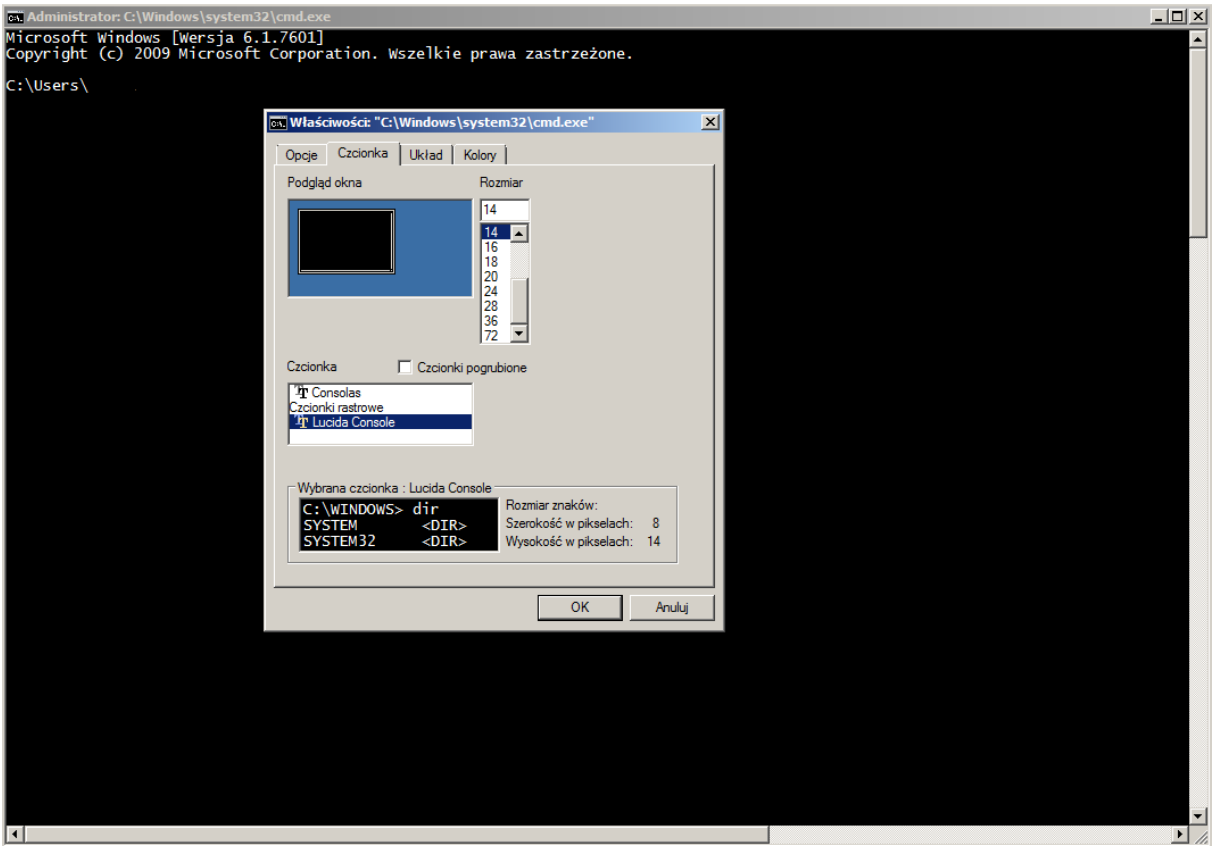
Interfejs podczas rozgrywki:



- Health: punkty zdrowia, gdy są równe 0 gracz umiera.
- Weapon: obecnie zaekwipowana broń.
- Ammo: amunicja w obecnie zaekwipowanej broni.
- Score: wynik gracza.
- Difficulty: poziom trudności wybrany w opcjach.

6. Krótka instrukcja instalacji.

Gra nie wymaga instalacji. Wystarczy uruchomić plik z rozszerzeniem .exe. Prawidłowe działanie gry wymaga zainstalowania odpowiedniej czcionki i jej zmiany w konsoli systemowej.



Program był testowany na czcionkach MS Gothic oraz SimSun-ExtB. Przy nieodpowiedniej czcionce matematyczne opcje graficzne mogą nie działać poprawnie.

Linki do pobrania czcionek:

MS Gothic: https://www.ffonts.net/MS-Gothic.font.download

SimSun-ExtB: https://fontzone.net/font-details/simsun-extb

Instalacja i dodanie czcionki do cmd: https://www.howtogeek.com/howto/windows-vista/stupid-geek-tricks-enable-more-fonts-for-the-windows-command-prompt/.

7. Analiza możliwości rozbudowy programu.

Dalszy rozwój programu powinien iść w kierunku rozbudowy zarówno opcji graficznych jak i urozmaicenia rozgrywki pod względem różnorodności przeciwników, zachowania zombie, arsenału broni, upuszczanych przedmiotów, czy poziomów trudności. Builder powinien być wzbogacony o inne opcje graficzne. Można stworzyć budowniczych zajmujących się tworzeniem mapy, po której porusza się gracz. Obecnie ruch gracza ogranicza jedynie ramka stanowiąca granicę mapy. Plansza może składać się z korytarzy, może tworzyć jakiegoś rodzaju labirynt. Obecnie modele są teksturą typu char. Modele powinny się składać z większej liczby znaków. Gracz mógłby zmieniać wygląd swojej postaci. Stworzylibyśmy do tego kolejny builder. Można dodać kolejne bonusy do przedmiotów upuszczanych przez zombie. Mogą to być bonusy szybkości, brak czasu przeładowania broni, większe obrażenia zadawane dla przeciwników itp. Dodanie kolejnych rodzajów zombie wiązałoby się z wymyśleniem nowych stanów, aby osiągnąć unikalność każdego z zombie. Dobrze by było zbudować stan, który odliczałby czas życia zombie. Po określonym interwale (jeżeli zombie nadal nie jest zabity) zombie wpadałby w szał - *EnragedState* lub inny stan. Należy także urozmaicić poziomy trudności. Algorytmy mogą również wyznaczać pozycję pojawiania się kolejnego zombie. Przykładowo kiedy za dobrze nam idzie, zombie powinny pojawiać się bliżej nas. Tablica wyników jest obecnie nieco nieczytelna ze względu na brak możliwości sortowania. Można stworzyć wewnętrzny iterator dla struktury *ScoreAggregate*, który sortowałby listę wyników po poziomie trudności i punktach gracza.