Dokumentacja programu KillKardashians

**Dawid Holewa**

1. Overview

Aplikacja KillKardashians to prosta gra platformowa, w której gracz potrafi poruszać się postacią oraz strzelać pociskami. Aby przejść grę, należy pokonać wszystkich przeciwników.

Kod całego programu znajduje się na gicie: <https://github.com/nambonumestinvisibili/KillKardashians>

1. Opis użytkowania
   1. **Instalacja (na Windowsie)**

W terminalu:

**git clone** [**https://github.com/nambonumestinvisibili/KillKardashiansExecutable**](https://github.com/nambonumestinvisibili/KillKardashiansExecutable)

może być potrzebna kompilacja komendą **java -jar KillKardashians.jar** (KillKardashians.jar znajduje się w folderze KillKardashiansExecutable-master. Musi znajdować się w tym samym folderze co resources.)

* 1. **Jak grać?**

Po uruchomieniu gry, gracz ma do dyspozycji klawisze a, w, d, dzięki którym może poruszać się odpowiednio w lewo, w górę i w prawo. Po naciśnięciu spacji, gracz wystrzeli pociski, potrafiące zabić przeciwników. Różni przeciwnicy potrzebują różnej ilości pocisków, by zniknąć. Gracz ma do przejścia 3 poziomy. Po przejściu trzech poziomów, pojawi się okno dialogowe z informacją o wygranej. Przeciwnicy mają zdolność strzelania pociskami, gdy gracz znajduje się blisko nich. Gracz ma trzy szanse i po utracie wszystkich, wyświetla się okienko dialogowe z zapytaniem, czy gracz chce grać od nowa, czy aplikacja powinna się zamknąć.

1. Opis architektury programu

3.1 **Klasy**:

- ***Main***: ustawia okienko w odpowiedni miejscu, wywołuje też GameStarter

- ***GameStarter***: reprezentuje okienko, w którym dostajemy informację, jak grać. Umożliwia odpalenie właściwej gry poprzez przycisk play

- ***Sprites***: jest używany jako interfejs dla wszystkich klas, które będą posiadały zdolność poruszania się.

- ***Missiles***: Abstrakcyjna klasa dla pocisków gracza i pocisków przeciwników. Mówi, jak pociski mają wyglądać i jak się poruszać.

- ***PlayerMissile*** i ***ObstacleMissile***: różnią się kolorem oraz stroną, w która mają się poruszać. Pociski gracza poruszają się w stronę na którą wskazuje ostatnio wciśnięty przycisk na klawiaturze, natomiast pociski przeciwników zawsze kierują się w stronę gracza.

- ***Obstacle***: Również abstrakcyjna klasa. Określa wygląd przeciwników oraz sposób poruszania się: przeciwnicy potrafią poruszać się tylko po platformach

- ***Obstacle*** ***High***, ***Medium*** i ***Low***: różnią się swoim przypisanym obrazkiem, ilością pocisków, jaką muszą przyjąć, by zniknąć, oraz szybkością, z jaką poruszają się po platformach

- ***Wall***: reprezentuje platformy, po których będzie można się poruszać.

- ***Player***: posiada wiele informacji o graczu: startową pozycję, szybkość poruszania się, czy wciśnięte są klawisze akcyjne, ilość szans, stronę w którą jest obecnie skierowany gracz. Określa aktualną pozycję i możliwości poruszania się gracza po naciśnięciu odpowiedniego klawisza, biorąc pod uwagę środowisko (np. jeśli gracz znajduje się w powietrzu, to nie potrafi wyżej podskoczyć (*set()*)). Manipuluje również szybkością gracza (*maintainSpeed()* zapewnia, że gracz nie będzie się zbyt długo „ślizgał” po platformie). Zawiera też funkcję *afterHit()* która potrafi usunąć obiekty, z którymi zderzył się gracz: np. pociski.

- ***Factory***: jest to klasa, której każda składowa jest statyczna. Odpowiada ona za przetwarzanie danych z pliku do budowy planszy, po której poruszają się obiekty typu Sprite (funkcja *createAllGameObjects()*). Posiada takie metody, które potrafią przygotować planszę do nowego levelu (*handleNextLevel* usuwa wszystkie obiekty grywalne w GamePanel).

- ***KeyChecker***: dziedziczy po KeyAdapter: wyłapuje klawiaturowe akcje i odpowiada za wykonanie odpowiednich czynności po naciśnięciu i puszczeniu przycisku.

- ***GamePanel***: W konstruktorze uruchamiany jest TimerTask, który gra rolę main game loop. Wykonują się tam takie czynności jak: posprzątanie starej i utworzenie nowej planszy (wraz z obiektami (dzięki Factory)); kontrolowany jest ruch każdego obiektu (dzięki temu, że każdy Sprite posiada *.set()*). Sprawdzane są kolizje między pociskami gracza i przeciwnikami, GamePanel odpowiada też za strzelanie przez przeciwników pociskami oraz za sprawdzanie warunków do zakończenia gry.

- ***WinObserver***, ***LoseObserver***, ***Observer***: Observer to interfejs dla Win- I Lose-, które mają wywoływać przede wszystkim okna dialogowe z informacjami dla użytkownika.

**3.2 Ruch i Kolizje:**

Każdy obiekt reprezentowany jest jako Rectangle, kolizje są wykrywane poprzez przecinanie się obiektów takiego typu (każdy Sprite, Player i Wall posiada swój własny Rectangle). Ruch obiektów jest różnie rozpatrywany: Obstacles poruszają się w lewo/prawo, tak długo jak kolidują ze swoją platformą, w przeciwnym razie zmieniają kierunek. Player porusza się w lewo/prawo kiedy chce, niezależnie od pocisku, lecz zawsze spada, tak długo jak nie zacznie kolidować z obiektem typu Wall. Pociski Playera/Obstacles kolidują tylko z Obstacles/Playerem, po kolizji lub wyjściu poza planszę znikają.

**3.3 Połączenia między klasami**

GamePanel łączy prawie wszystkie klasy ze sobą. GamePanel posiada wiele ArrayList, których elementami są wszystkie obiekty typu Sprites i Wall i Observers. Posiada również referencję do Playera. „Statyczna” klasa Factory nie potrzebuje odwołań, jej składowe przyjmują GamePanel jako argument. Main wywołuje GameStarter, a z kolei GameStarter GamePanel. W ten sposób wyczerpujemy wszystkie możliwe klasy, może lepiej widać to na diagramie:

*\*tutaj diagram\**

* 1. **Motywacja podziału na klasy**

Starałem się zorganizować podział na klasy tak, by zgadzały się z zasadami SOLID, np. Każda klasa ma tylko jedną odpowiedzialność (np. GamePanel realizuje rozgrywkę całej gry, Factory odpowiada za przygotowanie planszy).

1. Wzorce Projektowe:

Wzorce Projektowe zaimplementowane w programie:

**- MVC:**

MVC opiera się na kontrolerze, modelu i rendererze.

Rolę kontrolera pełni klasa **KeyChecker**: wyłapuje akcje związane z klawiaturą i przetwarza je na odpowiednie czynności (zmienia stan pól w klasie Player). Po części rolę kontrolera pełni też klasa Player, ponieważ jej składowe są w stanie powiedzieć, czy zlecone polecenie może zostać wykonane.

Rolę modelu pełni klasa **Player**, jeśli zlecono polecenie i jest ono poprawne, jej składowe zmieniają swoje współrzędne, bądź zostają wystrzelone pociski. Model nie potrafi sam siebie narysować, ani zmienić swojego własnego stanu.

Rolę renderera pełni klasa **GamePanel**. Ponieważ jest ona jednocześnie „main game loop”, nakazuje klasie Player weryfikację swojej pozycji na podstawie zmiany stanu przez KeyChecker. Po wszystkich wykonanych zmianach (pozycje zmieniają również inne obiekty, lecz samoistnie, nie dzięki poleceniom użytkownika). Ostatnią instrukcją jest repaint(), która renderuje aktualny obraz z aktualnymi pozycjami, dzięki czemu użytkownik wie, jaki jest obecny stan, a to pozwala mu na dalszą rozgrywkę.

**- Factory Method**

Ten wzorzec jest zaimplementowany w klasie Factory, rozwiązuje problem tworzenia obiektów bez potrzeby operowania na konkretnych klasach. Nasza klasa dzięki otrzymanym danym (u nas to ciąg znaków w csv, gdzie każda cyfra odpowiada innemu obiektowi np.: 0 – nic, 1 – player, 2 – wall, itd…) potrafi sama wygenerować dowolną planszę z dowolną liczbą różnych obiektów różnych klas, co uwalnia nas od samodzielnej konstrukcji poziomów.

- **Observer**

Ten wzorzec wykorzystałem do rozwiązania problemu zakończenia gry. Klasa, która będzie wykorzystywała obserwatorów to GamePanel. GamePanel posiada listę swoich obserwatorów. W TimerTasku sprawdza się warunki, które odpowiadają za zakończenie gry z nie- lub powodzeniem. Jeśli się okaże, że warunki są spełnione, uruchamia się notifyObserver z odpowiednim parametrem w postaci odpowiedniego obserwatora, co powoduje jego aktywację poprzez onNotify(), a to w konsekwencji uruchomienie okienek dialogowych z informacją. Wydaje się, że przy prostocie tej gry jest to dość niepotrzebne, bo bez tego kod był bardziej czytelny i zwięzły. Jednak, gdybyśmy chcieli rozwinąć grę o na przykład osiągnięcia (jak na przykład: Pozbycie się przeciwników w określonym czasie albo zdobycie maksymalnej ilości coinsów), to obserwator, który powiadamiany by był po spełnieniu odpowiednich warunków i wyświetlałby powiadomienia, nadałby się do tego idealnie.

- **FlyWeight**

Gdybyśmy chcieli, żeby gra była bardziej realistyczna, moglibyśmy ją urozmaicić o różne rodzaje podłoży: na przykład zamiast samotnego Wall, podłoża typu Grass, River, Stone itp. Każdy z różnych rodzajów podłoża miałby różne właściwości, które wpływałyby na rozgrywkę (na przykład chodzenie po kamieniu byłoby szybsze niż po lodzie). Oprócz tego każdy z nich potrzebowałby osobnego obrazka. Umieszczenie takich samych danych dla każdego kafelka, byłoby bardzo nieefektywne. Zamiast tego moglibyśmy stworzyć klasę, która jest „bankiem obrazków” i dla danego kafelka umieścić w nim referencję do obrazka w banku (robimy tak na przykład nadając każdemu rodzajowi przeciwnika tylko referencję do obrazka znajdującego się już w klasie Bank, a nie tworzymy nowego obrazka przy każdej konstrukcji nowego obiektu).