

OPIS KLAS

Coin Hunter



Uniwersytet Jagielloński, Wydział Matematyki i Informatyki
Inżyniera Oprogramowania

Spis treści

1. Wprowadzenie	2
2. Klasy na tle zasady SOLID	3
2.1. SRP (Single Responsibility Principle - Zasada Jednej Odpowiedzialności)...	3
2.2. OCP (Open-Closed Principle - Zasada Otwarte-Zamknięte)	3
2.3. LSP (Liskov Substitution Principle - Zasada Podstawienia Liskov).....	4
2.4. ISP (Interface Segregation Principle - Zasada Segregacji Interfejsów)	4
2.5. DIP (Dependency Inversion Principle - Zasada Odwrócenia Zależności).....	4
3. Opis klas i funkcji	5
3.1. EntityFactory	5
3.2. EnemyFactory	5
3.3. CoinFactory	6
3.4. AbstractEntity	6
3.5. Enemy	6
3.6. Coin	7
3.7. GenerateEntity.....	7
3.8. Player	7
3.9. Controller.....	8
3.10. GameEngine	8

1. Wprowadzenie

Niniejszy dokument przedstawia klasy, które znajdują się w aplikacji „*Coin Hunter*” wraz z ich opisem. Ku lepszemu zdefiniowaniu działania klas opisano również zależności jakie dane klasy prezentują na tle zasady SOLID. Zasada SOLID została również opisana w pliku „Metodologia, architektura i wzorce”, w folderze „Documentation”.

2. Klasy na tle zasady SOLID

Na początku warto wspomnieć o wzorcu projektowym „Fabryka Abstrakcyjna”, który został zrealizowany w projekcie „*Coin Hunter*” poprzez interfejs `EntityFactory` oraz implementujące go klasy `CoinFactory` i `EnemyFactory` zwracające obiekty `Coin` i `Enemy` będące pochodnymi klasy `AbstractEntity`.

2.1. SRP (Single Responsibility Principle - Zasada Jednej Odpowiedzialności)

Klasa powinna mieć tylko jedną odpowiedzialność (nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy).

Każda z klas projektu ograniczona jest do jednej odpowiedzialności, np:

EnemyFactory

tworzy obiekt `Enemy`.

Enemy

zawiera tylko metody dotyczące stanu obiektu `Enemy` jak:

- `ChangePosition()` - zmiana wartości pola `position`,
- `checkCollision(number)` - sprawdzenie czy obiekt ma kolizję z obiektem na pozycji danej argumentem
- `remove()` - usunięcie obiektu

Player

zawiera tylko metody dotyczące stany obiektu `Player`, np:

- `grabCoin()` - zmiana ilości zgromadzonych monet
- `updatePlayerPosition()` - aktualizacja pozycji obiektu `Player`
- `getHit()` - przyjęcie ataku przez playera

2.2. OCP (Open-Closed Principle - Zasada Otwarte-Zamknięte)

Klasy (encje) powinny być otwarte na rozszerzenia i zamknięte na modyfikacje.

Program otwarty jest na dodawanie nowych obiektów (entities), w celu dodania nowego entity. Wystarczy stworzyć klasę obiektu (dziedziczącą po `AbstractEntity`), jeżeli dodatkowo chcemy, aby obiekt był nowego typu (innego niż generowane przez `CoinFactory` lub `EnemyFactory`), możemy stworzyć nową implementację interfejsu `EntityFactory`.

2.3. LSP (Liskov Substitution Principle - Zasada Podstawienia Liskov)

Funkcje, które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.

W projekcie „Coin Hunter”, klasa GenerateEntity przyjmuje w konstruktorze i generuje obiekty dowolnej klasy dziedziczącej po AbstractEntity.

2.4. ISP (Interface Segregation Principle - Zasada Segregacji Interfejsów)

Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny.

Interfejsy w projekcie: AbstractFactory, AbstractEntity.

2.5. DIP (Dependency Inversion Principle - Zasada Odwrócenia Zależności)

Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych - zależności między nimi powinny wynikać z abstrakcji.

W projekcie „Coin Hunter”:

- Interfejs EntityFactory i implementujące go CoinFactory i EnemyFactory,
- Klasa abstrakcyjna AbstractEnemy i dziedziczące po niej klasy Coin i Enemy.

3. Opis klas i funkcji

W grze zdefiniowana jest funkcja globalna:

- `const setRandomInterval (intervalFunction, minDelay, maxDelay)`

Funkcja wykorzystywana jest w metodzie `generateEntity()` klasy `GenerateEntity` do generowania obiektów gry (`Coin`, `Enemy`) w odstępach czasowych zależnych od wartości argumentów `minDelay`, `maxDelay` oraz pewnej losowej wartości. Argument `intervalFunction` jest funkcją wywoływaną we wspomnianych odstępach czasowych, w przypadku opisywanej aplikacji jest to funkcja generująca obiekty `Coin` bądź `Enemy`.

Funkcja zapożyczona została z pakietu open-source:

<https://github.com/jabacchetta/set-random-interval>

W grze zdefiniowane następujące klasy:

3.1. EntityFactory

Klasa abstrakcyjna reprezentująca fabryki obiektów w grze.

Pola:

- **protected MAX_DELAY:** number - argument dla funkcji `setRandomInterval`,
- **protected MIN_DELAY:** number - argument dla funkcji `setRandomInterval`.

Metody:

- **public getMaxDelay()** - zwraca wartość `MAX_DELAY`,
- **public getMinDelay()** - zwraca wartość `MIN_DELAY`,
- **abstract CreateEntity()**.

3.2. EnemyFactory

Dziedziczy po `EntityFactory`. Fabryka obiektów `Enemy`.

Konstruktory:

- **constructor(number, number)** - przyjmuje i ustala dwie wartości: `MAX_DELAY` i `MIN_DELAY`.

Metody:

- **public CreateEntity()** - zwraca obiekt `Enemy`.

3.3. CoinFactory

Dziedziczy po [EntityFactory](#). Fabryka obiektów [Coin](#).

Pola:

- **constructor(number, number)** - przyjmuje i ustala dwie wartości: MAX_DELAY i MIN_DELAY.

Metody:

- **public CreateEntity()** - zwraca obiekt Coin.

3.4. AbstractEntity

Klasa abstrakcyjna reprezentująca obiekty w grze.

Pola:

- **protected position:** number,
- **protected entityDiv:** HTMLDivElement
- **protected ownSpeed:** number.

Konstruktory:

- **constructor()** – ustawia pole position na wartość początkową wspólną dla wszystkich obiektów.

Metody:

- **public changePosition()** - zmienia pozycję obiektu,
- **public remove()** - usuwa obiekt z okna gry,
- **public getPosition()** - zwraca wartość pola position,
- **public getEntityDiv()** - zwraca wartość pola entityDiv,
- **abstract checkCollision(number).**

3.5. Enemy

Dziedziczy po [AbstractEntity](#). Klasa obiektów Enemy

Konstruktory:

- **constructor(number)** – przyjmuje i ustala wartość pola ownSpeed.

Metody:

- **public checkCollision(number)** - sprawdza czy obiekt koliduje z obiektem Player z wartością pola position daną argumentem.

3.6. Coin

Dziedziczy po [AbstractEntity](#). Klasa obiektów Coin.

Konstruktory:

- **constructor(number)** – przyjmuje i ustala wartość pola ownSpeed.

Metody:

- **public checkCollision(number)** - sprawdza czy obiekt koliduje z obiektem Player z wartością pola position daną argumentem.

3.7. GenerateEntity

Generator obiektów Entity, wykorzystujący fabrykę Entity - generuje obiekty zwracane przez fabrykę podaną w konstruktorze z losowym opóźnieniem (mieszczącym się w zdefiniowanych ramach)

Pola:

- **private factory:** EntityFactory.

Konstruktory:

- **constructor(EntityFactory)** - przyjmuje obiekt klasy implementującej EntityFactory.

Metody:

- **generateEntity(AbstractEntity[])** - funkcja generuje obiekty tworzone przez obiekt w polu factory z użyciem globalnej funkcji setRandomInterval i umieszcza je w kolejce danej argumentem.

3.8. Player

Implementuje obiekt gracza.

Pola:

- **private playerDiv:** HTMLElement,
- **private position:** number,
- **private lives:** number,
- **private immune:** Boolean,
- **private distance:** number,
- **private coins:** numer.

Konstruktory:

- **constructor()** – ustala wartości początkowe pozycji, przebytego dystansu, zdobytych monet oraz liczbę żyć.

Metody:

- **public getPosition()** - zwraca aktualną pozycję gracza (oś Y),

- **public getLives()** - zwraca aktualną liczbę żyć,
- **public takeLife()** - odejmuje graczowi jedno życie,
- **public getDistance()** - zwraca przebyty przez gracza dystans,
- **public increaseDistance()** - zwiększa dystans przebyty przez gracza o jedną jednostkę,
- **public getCoins()** - zwraca ilość zdobytych przez gracza monet,
- **public jump()** - trigger animacji, zmiana pozycji gracza w czasie.

3.9. Controller

Klasa obsługująca kontrolę gry za pomocą klawiatury.

Pola:

- **private keyInput: { up: boolean }** - obiekt oznaczający stan wciśnięcia dowolnego z klawiszy odpowiedzialnych za skok gracza.

Konstruktory:

- **constructor()** - dodaje eventListener obserwujący wciśnięcie i puszczenie klawiszy klawiatury przez użytkownika.

Metody:

- **public isKeyUp()** - zwraca boolean oznaczający stan wciśniętego klawisza (pole w obiekcie keyInput),
- **public keyUpHandler(e)** - metoda obsługująca event puszczenia klawisza, sprawdzająca czy został wciśnięty klawisz odpowiedzialny za skok gracza oraz blokująca domyślnie zachowanie po naciśnięciu (w celu np. zapobieganiu scrollowania strony),
- **public keyDownHandler(e)** - analogicznie jak wyżej, dla wciśnięcia klawisza.

3.10. GameEngine

Silnik gry zawierający główną pętlę odświeżającą stan obiektów w grze.

Pola:

- **private controller:** Controller,
- **private player:** Player,
- **private coins:** Coin[] (tablica obiektów Coin),
- **private enemies:** Enemy[] (tablica obiektów Enemy),
- **private coinGeneratorInterval:** {clear: () => void} - (obiekt zwracany przez funkcję randomizującą interwały w grze, które są używane w generatorach obiektów implementujących klasę AbstractEntity),
- **private enemyGeneratorInterval:** {clear: () => void}.

Metody:

- **private coinsUpdate(number)** - obsługuje pozycję monet - jej aktualizację oraz sprawdzanie względem hitboxa playera,
- **private enemiesUpdate(number)** - analogicznie jak wyżej, dla obiektów Enemy,
- **private update(number)** - funkcja aktualizująca stan obiektów w grze, wywoływana w pętli głównej gry przy każdym odświeżeniu jej stanu,
- **private gameLoop(number)** - główna pętla gry,
- **private countDistance()** - funkcja aktualizująca przebyty przez gracza dystans w grze,
- **private startProcedure()** - wyświetla licznik czasu pozostałego do rozpoczęcia rozgrywki po kliknięciu "PLAY",
- **public init()** - inicjalizacja obiektów gry,
- **private endGame()** - zakończenie generacji obiektów gry, wyświetlenie ekranu końca gry oraz wyemitowanie do bazy danych wyników gracza (niżej opisana sendResultToDb),
- **private sendResultToDb(number, number)** - wyemitowanie wyników rozgrywki do bazy danych.