Projekt: Tower Defense

Dokumentacja Techniczna i Wzorce Projektowe

1. PODZIAŁ PRACY W ZESPOLE

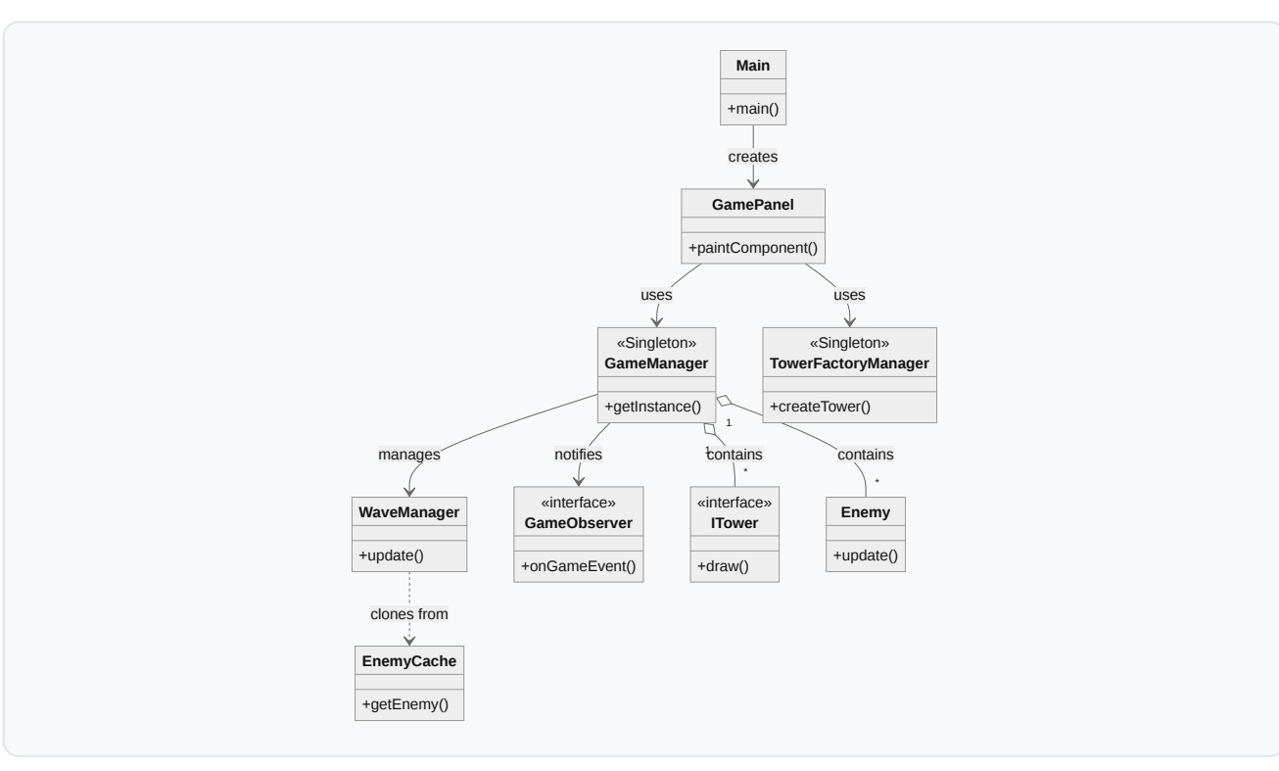
Autor	Zrealizowane zadania i moduły
Piotr Czyżewski	• Implementacja wzorca **Decorator** (TowerDecorators.java) - system ulepszeń wież z wizualnymi wskaźnikami. • Implementacja wzorca **Factory Method** (AllFactories.java) - mechanika tworzenia różnych typów wież (Łucznik, Armata, Snajper, Laser). • Walidacja pozycji wież w TowerFactoryBase.createTowerWithValidation().
Mikołaj Staniaszek	• Implementacja wzorca **Command** (IGameCommand, BuyTowerCommand, StartWaveCommand) - enkapsulacja akcji UI. • Implementacja głównego panelu gry (GamePanel.java) z pętlą renderowania 60 FPS. • System obsługi myszy i klawiatury, menu ulepszeń wież (PPM). • Integracja interfejsu użytkownika z systemem komend przez mapę commands.
Bartosz Składanowski	• Implementacja wzorca **Observer** (AllObservers.java) - system reagowania na zdarzenia gry. • Klasy: StatisticsObserver, SoundObserver, AchievementObserver, LoggerObserver. • Współpraca przy wzorcu **Prototype** - definicja interfejsu Prototype i klasy Enemy. • System powiadomień o osiągnięciach z animacjami (AchievementNotification w GamePanel).
Maciej Socik	• Implementacja wzorca **Singleton** (GameManager, TowerFactoryManager) - zarządcy stanu gry. • Implementacja wzorca **Prototype** - cache przeciwników (EnemyCache.java) z mechanizmem klonowania. • Zarządzanie falami przeciwników (WaveManager.java) z progresywnym skalowaniem trudności. • System zdarzeń (GameEvent, GameEventType) dla wzorca Observer. • Konfiguracja ścieżek na mapie i zarządzanie kolekcjami thread-safe (CopyOnWriteArrayList).

2. SPECYFIKA ROZWIĄZAŃ TECHNOLOGICZNYCH

Projekt został zrealizowany w języku **Java (JDK 17+)** z wykorzystaniem biblioteki **Swing** oraz **AWT**.

- **Wątkowość (Threading):** Gra wykorzystuje osobny wątek dla pętli logicznej (game loop) działającej w 60 FPS (GamePanel.run()), oddzielony od wątku renderowania UI (EDT - Event Dispatch Thread).
- **Thread-Safety:** Użyto kolekcji `CopyOnWriteArrayList` dla list `enemies`, `towers` i `projectiles` w GameManager, aby uniknąć `ConcurrentModificationException` podczas iteracji i modyfikacji w różnych wątkach.
- **Graphics2D:** Renderowanie oparte na `Graphics2D` z włączonym anti-aliasingiem (`RenderingHints.VALUE_ANTIALIAS_ON`) i transformacjami (scaling, translate) dla responsywnego UI.
- **Event-Driven Architecture:** System zdarzeń oparty na enum `GameEventType` (ENEMY_KILLED, TOWER_BUILT, etc.) i klasie `GameEvent` przesyłającej dane do Observerów.
- **Pełny ekran:** Wykorzystanie `GraphicsDevice.setFullScreenWindow()` dla trybu full-screen z automatycznym skalowaniem contentu.

3. ARCHITEKTURA SYSTEMU



Wzorce 1 & 2: Singleton i Observer

1. Singleton

Cel użycia: Gwarancja istnienia tylko jednej instancji kluczowych zarządców (GameManager, TowerFactoryManager) i globalny punkt dostępu do nich z dowolnego miejsca w kodzie.

Role w wzorcu Singleton:

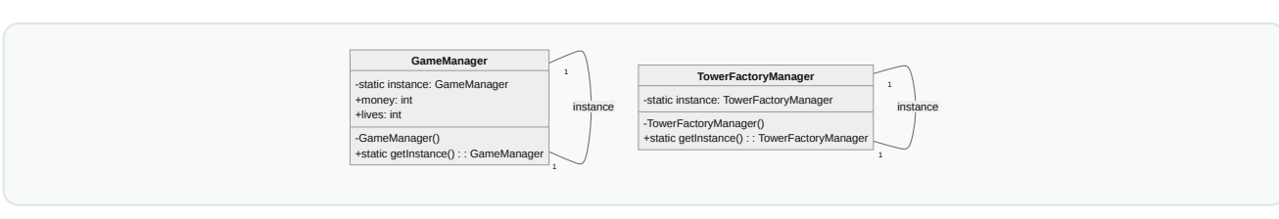
Singleton: GameManager, TowerFactoryManager
Instance (pole statyczne): GameManager.instance, TowerFactoryManager.instance
getInstance() (metoda dostępowa): GameManager.getInstance(), TowerFactoryManager.getInstance()
Konstruktor prywatny: private GameManager(), private TowerFactoryManager()
Client: GamePanel, WaveManager, Main (wszyscy klienci wywołujący getInstance())

Lokalizacja i użycie:

Definicja: GameManager.java (linie 33-36), AllFactories.java (linie 130; 139-144)
Użycie:
• GamePanel.java (linie 11-13): Inicjalizacja referencji do singletonów.
• WaveManager.java (linia 2): Pobranie instancji GameManager.
• Tower.java (linia 28): Dostęp do listy wrogów przez GameManager.

Wektor zmian:

Ułatwia dodawanie nowych globalnych usług (np. AudioManager, SaveManager) bez modyfikacji przekazywania referencji przez konstruktory. Nowe singletony można dodać bez zmiany istniejącego kodu.



src/GameManager.java (linie 33-36)

```
public static GameManager getInstance() {
    if (instance == null) instance = new GameManager();
    return instance;
}
```

2. Observer

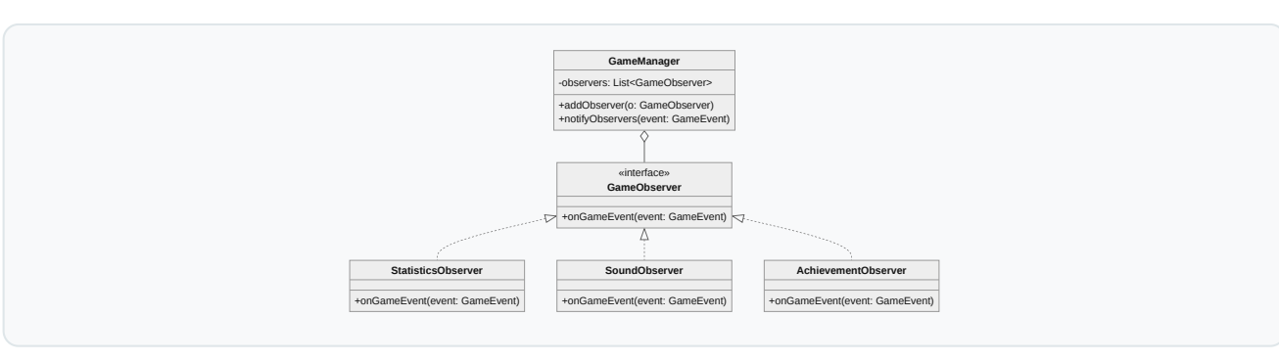
Cel użycia: Reakcja na zdarzenia gry (śmierć wroga, nowa fala, zmiana pieniędzy) w niezależnych modułach (Dźwięk, UI, Osiągnięcia, Logi) bez silnego wiązania kodu.

Role w wzorcu Observer:

Subject (Podmiot): GameManager
Observer (Interfejs obserwatora): GameObserver
ConcreteObserver: StatisticsObserver, SoundObserver, AchievementObserver, LoggerObserver, GamePanel
attach() (rejestracja): GameManager.addObserver()
notify() (powiadamianie): GameManager.notifyObservers(GameEvent)
Event (dane zdarzenia): GameEvent z GameEventType

Lokalizacja i użycie:

Definicja: GameObserver.java (interfejs), AllObservers.java (concrete observers), GameEvent.java (enum + klasa)
Użycie - rejestracja: Main.java (linie 16-21): dodawanie obserwatorów do GameManager.
Użycie - powiadamianie:
• GameManager.takeDamage() (linie 66-75): powiadomienie o zmianie żyć.
• GameManager.enemyKilled() (linie 77-79): powiadomienie o nagrodzie.
• WaveManager.startWave() (linie 6-31): wywołanie metody waveStarted w linii 31.



Wzorzec 3: Factory Method

CEL UŻYCIA

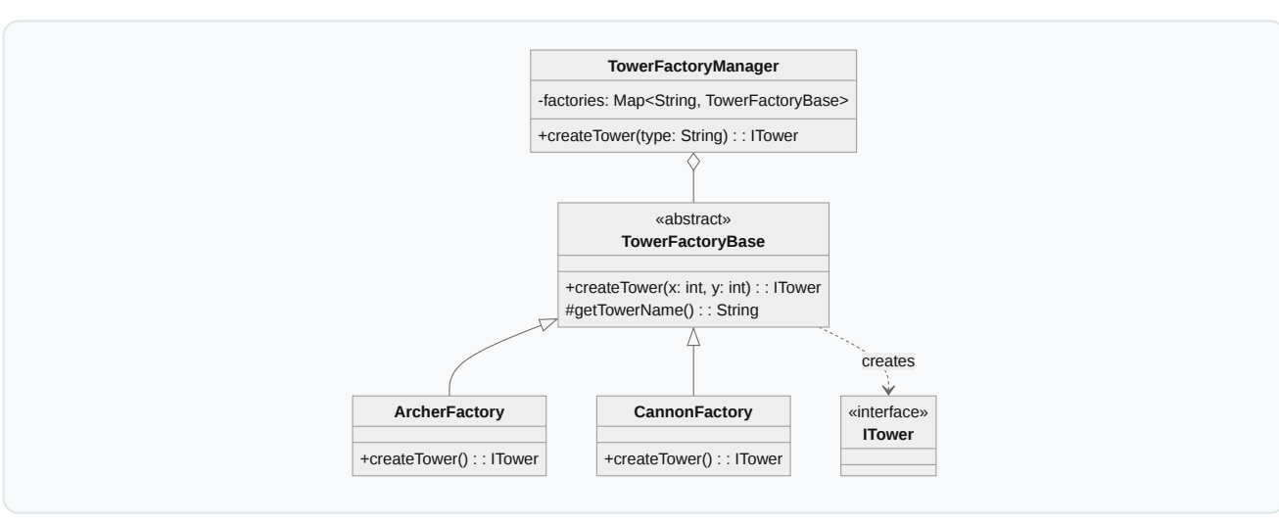
Oddzielenie procesu tworzenia obiektów (wież) od miejsca ich użycia. Pozwala na dodawanie nowych typów wież poprzez dodanie nowej fabryki, bez modyfikacji kodu klienta (GamePanel).

Role w wzorcu Factory Method:

Creator (abstrakcyjna fabryka): TowerFactoryBase
ConcreteCreator (konkretne fabryki): ArcherFactory, CannonFactory, SniperFactory, LaserFactory
Product (interfejs produktu): ITower
ConcreteProduct (konkretny produkt): Tower
factoryMethod() (metoda fabrykująca): createTower(int x, int y)
Client (klient fabryki): TowerFactoryManager, GamePanel

Lokalizacja i użycie:

Definicja: AllFactories.java
• TowerFactoryBase (linie 6-27) - klasa abstrakcyjna.
• Konkretne fabryki: Archer (38-59), Cannon (60-82), Sniper (84-104), Laser (107-128).
• TowerFactoryManager (linie 130-180) - zarządza fabrykami.
Użycie:
• GamePanel.handleClick() (linie 167-253): tworzenie wieży wybranego typu.
• TowerFactoryManager.registerDefaultFactories() (linie 146-151): rejestracja dostępnych fabryk.



Wektor zmian:

Dodanie nowej wieży (np. FlameThrowerFactory) wymaga tylko: (1) stworzenia klasy extends TowerFactoryBase, (2) zarejestrowania w TowerFactoryManager.registerDefaultFactories(). Manager automatycznie obsłuży nowy typ bez zmian w GamePanel.

src/AllFactories.java (linie 38-58)

```
class CannonFactory extends TowerFactoryBase {
    @Override
    public ITower createTower(int x, int y) {
        return new Tower(x, y, getTowerName(),
            getBaseRange(), getCooldown(),
            getBaseDamage(), getTowerColor());
    }

    @Override
    protected String getTowerName() { return "Armata"; }
    @Override
    protected int getBaseDamage() { return 60; }
}
```

Wzorzec 4: Decorator

CEL UŻYCIA

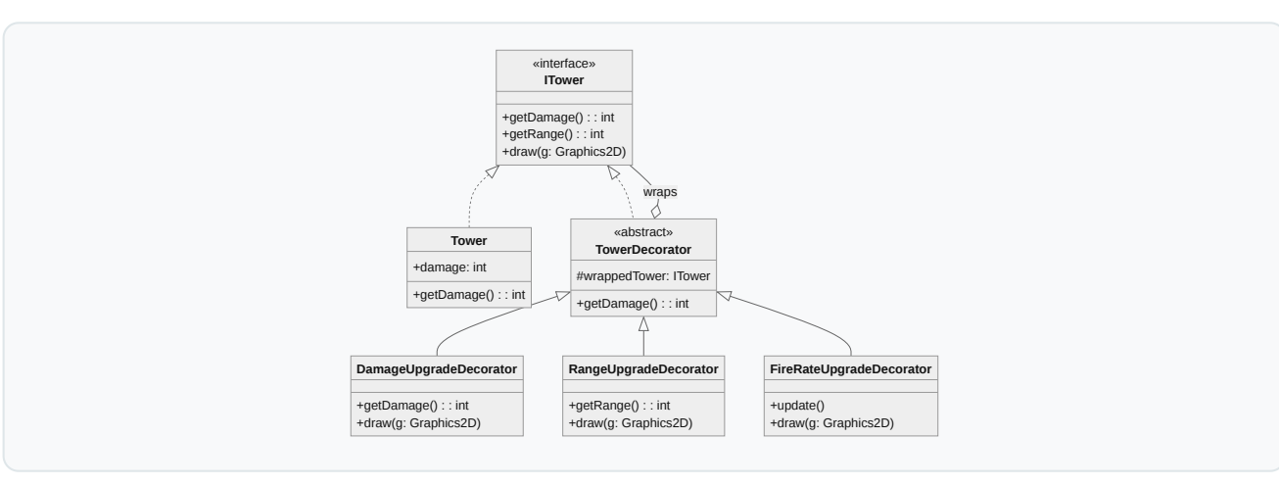
Dynamiczne dodawanie funkcjonalności (ulepszenia statystyk, zmiana wyglądu) do obiektów wież w trakcie działania programu, bez konieczności tworzenia wielu podklas dziedziczących.

Role w wzorcu Decorator:

Component (interfejs komponentu): ITower
ConcreteComponent (bazowy komponent): Tower
Decorator (abstrakcyjny dekorator): TowerDecorator
ConcreteDecorator: DamageUpgradeDecorator, RangeUpgradeDecorator, FireRateUpgradeDecorator
wrappedTower (opakowywany obiekt): pole protected ITower wrappedTower w TowerDecorator
Client: GamePanel.applyUpgrade() (tworzy dekoratory)

Lokalizacja i użycie:

Definicja: TowerDecorators.java
• TowerDecorator (linie 6-17) - bazowy dekorator
• Konkretne dekoratory: Damage (19-35), Range (37-56), FireRate (58-83)
Użycie:
• GamePanel.applyUpgrade() (linie 256-290): tworzenie dekoratorów na podstawie wyboru gracza.
• GamePanel.handleClick() (linie 217-230): wykrywanie kliknięcia PPM na wieży i otwieranie menu.



Wektor zmian:

Pozwala na łatwe tworzenie nowych typów ulepszeń (np. PoisonDecorator, SlowDecorator) lub tymczasowych efektów (buffów) poprzez "opakowywanie" obiektów. Można łączyć dekoratory (np. wieża z +damage i +range jednocześnie).

src/TowerDecorators.java (linie 19-33)

```
class DamageUpgradeDecorator extends TowerDecorator {
    public DamageUpgradeDecorator(ITower tower) {
        super(tower);
    }

    @Override
    public int getDamage() {
        // Bazowe obrażenia + bonus 25
        return wrappedTower.getDamage() + 25;
    }

    @Override
    public void draw(Graphics2D g) {
        wrappedTower.draw(g);
        // Złota ramka jako wizualny wskaźnik ulepszenia
        g.setColor(new Color(255, 215, 0));
        g.drawRect(getX() - 15, getY() - 15, 30, 30);
    }
}
```

Wzorzec 5: Prototype

CEL UŻYCIA

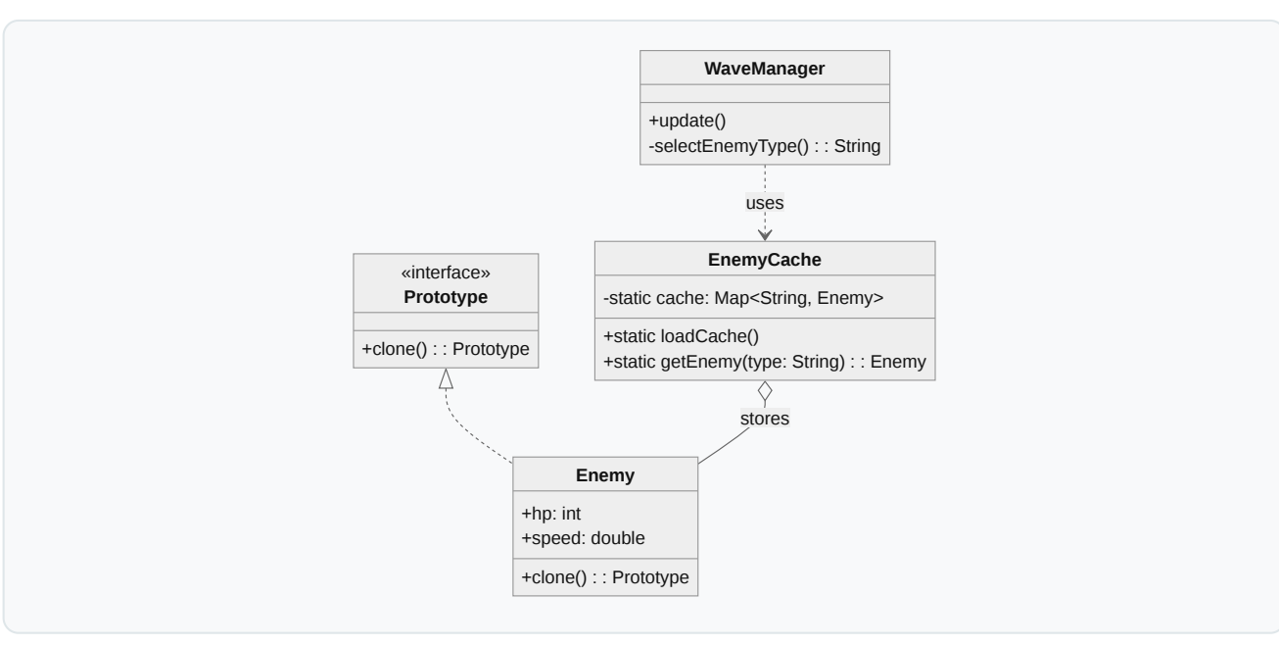
Optymalizacja tworzenia wielu obiektów przeciwników o podobnych parametrach. Definicje wrogów są ładowane raz do pamięci podręcznej (Cache), a następnie klonowane zamiast wielokrotnie tworzone przez konstruktor.

Role w wzorcu Prototype:

Prototype (interfejs prototypu): interface Prototype extends Cloneable
ConcretePrototype (konkretny prototyp): Enemy
clone() (metoda klonująca): Enemy.clone() (nadpisuje Prototype.clone())
PrototypeRegistry (rejestr prototypów): EnemyCache
Client: WaveManager.update() (pobiera sklonowane obiekty)

Lokalizacja i użycie:

Definicja:
• Prototype.java (interfejs).
• Enemy.java (linie 20-23): implementacja clone().
• EnemyCache.java: cache prototypów i metoda getEnemy().
Użycie:
• Main.java (linia 14): EnemyCache.loadCache().
• WaveManager.update() (linie 39-42): pobranie sklonowanego wroga.
• WaveManager.selectEnemyType() (linie 68-91): wybór typu wroga.



Wektor zmian:

Dodanie nowych typów przeciwników wymaga tylko rozszerzenia EnemyCache.loadCache() o nową definicję. Można tworzyć warianty sezonowe (zimowe, letnie) bez duplikacji kodu. System łatwo rozszerzalny o system "modów" gry z różnymi zestawami wrogów.

src/EnemyCache.java (linie 20-26)

```
public static Enemy getEnemy(String type) {
    Enemy prototype = cache.get(type);
    if (prototype == null) {
        prototype = cache.get("NORMAL"); // Fallback
    }
    return (Enemy) prototype.clone();
}
```

Wzorzec 6: Command

CEL UŻYCIA

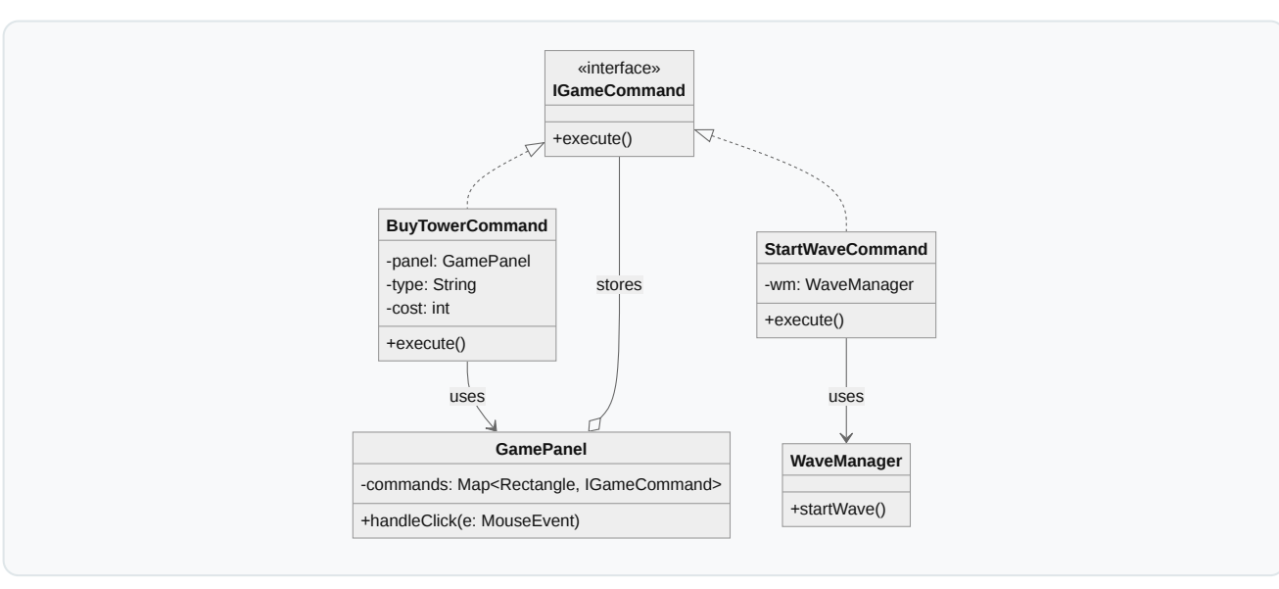
Enkapsulacja akcji użytkownika (kupno wieży, start fali) w obiekty komend. Ułatwia to obsługę GUI, pozwala na parametryzację przycisków i umożliwia łatwe rozszerzenie o funkcje undo/redo.

Role w wzorcu Command:

Command (interfejs komendy): IGameCommand
ConcreteCommand (konkretne komendy): BuyTowerCommand, StartWaveCommand
execute() (metoda wykonująca): IGameCommand.execute()
Receiver (odbiorcy): GamePanel (dla BuyTowerCommand), WaveManager (dla StartWaveCommand)
Invoker (wywoływacz): GamePanel.handleClick() - wykonuje komendy z mapy commands
Client (konfiguruje komendy): GamePanel konstruktor (linie 69-82)

Lokalizacja i użycie:

Definicja:
• IGameCommand.java (interfejs)
• BuyTowerCommand.java (linie 1-13)
• StartWaveCommand.java (linie 1-6)
Użycie - konfiguracja:
• GamePanel konstruktor (linie 69-79): mapowanie przycisków do komend.
Użycie - wykonanie:
• GamePanel.handleClick() (linie 167-253): pętla wykonująca komendę przypisaną do przycisku.



Wektor zmian:

Łatwe dodawanie nowych akcji UI (np. SellTowerCommand, PausGameCommand) bez modyfikacji logiki obsługi kliknięć. Możliwość implementacji historii komend dla undo/redo. Komendy można serializować i zapisywać (replay system).

src/BuyTowerCommand.java

```
class BuyTowerCommand implements IGameCommand {
    private GamePanel panel;
    private String type;
    private int cost, range;

    public BuyTowerCommand(GamePanel panel, String type,
                           int cost, int range) {
        this.panel = panel;
        this.type = type;
        this.cost = cost;
        this.range = range;
    }

    @Override
    public void execute() {
        // Ustawia wybraną wieżę w panelu
        panel.setSelectedTower(type, cost, range);
    }
}
```

Instrukcje

INSTRUKCJA UŻYTKOWNIKA

Cel gry: Przetrwanie 20 fal przeciwników poprzez budowanie i ulepszanie wież obronnych. Wrogowie podążają ustaloną ścieżką - jeśli dotrą do końca, tracisz życie.

Sterowanie i Interfejs:

- **LPM (Lewy Przycisk Myszy):** Wybór wieży ze sklepu (dół ekranu) i postawienie jej na mapie. Pole musi być wolne (nie może być ścieżką ani inną wieżą).
- **PPM (Prawy Przycisk Myszy):** Kliknięcie na istniejącą wieżę otwiera **Menu Ulepszeń** z trzema opcjami:
  - Atak (+25 damage) - 100$
  - Zasięg (+50 range) - 80$
  - Szybkość (4x fire rate) - 120$
- **Przycisk "START WAVE":** Rozpoczyna kolejną falę wrogów (dostępny tylko w fazie przygotowawczej).

Typy Wież:

- **Łucznik (50$):** Szybka wieża podstawowa (damage: 25, range: 120, cooldown: 600ms)
- **Armata (120$):** Silne pojedyncze strzały (damage: 60, range: 150, cooldown: 1500ms)
- **Snajper (250$):** Długi zasięg, wolny ogień (damage: 150, range: 300, cooldown: 2500ms)
- **Laser (80$):** Bardzo szybkie, słabe strzały (damage: 15, range: 100, cooldown: 300ms)

Skróty Klawiszowe:

- `S` - Włącz/Wyłącz panel statystyk (zabici wrogowie, wydane pieniądze, najwyższa fala).
- `L` - Włącz/Wyłącz logi systemowe (historia zdarzeń w grze).
- `M` - Wycisz/Włącz dźwięk (symulowane w konsoli).
- `A` - Pokaż panel osiągnięć (9 osiągnięć do odblokowania).
- `ESC` - Wyjście z gry.

Nowość - Zimowa Mapa:

Od 11 fali mapa zmienia się na zimową z nowym tłem i nowymi przeciwnikami:

- **Ice:** Podstawowy zimowy wróg (HP: 120, speed: 1.8)
- **Frost Giant:** Lodowy olbrzym (HP: 500, speed: 0.7) - pojawia się od fali 15
- **Blizzard:** Szybki mały wróg (HP: 60, speed: 4.2)

INSTRUKCJA INSTALACJI

Wymagania:

- Zainstalowane środowisko **Java Runtime Environment (JRE)** w wersji **17 lub nowszej**.
- System operacyjny: Windows 10/11, macOS 10.14+, lub Linux z GUI.
- Rozdzielczość ekranu: minimum 1280x720 (zalecane 1920x1080).

Kompilacja ze źródeł (opcjonalna):

```
javac -d bin src/*.java
java -cp bin Main
```

Uruchomienie z pliku JAR:

1. Pobierz plik `TowerDefense.jar`.
2. Uruchom terminal (konsolę) w folderze z grą.
3. Wpisz komendę:
   ```
   java -jar TowerDefense.jar
   ```
4. Alternatywnie: Kliknij dwukrotnie plik .jar (jeśli skojarzenia plików są poprawne).

Uwaga: Gra uruchamia się w trybie pełnoekranowym. Użyj ESC aby wyjść.