



POLSKO-JAPOŃSKA AKADEMIA
TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Dominik Stec

12623

**Generyczna wtyczka automatyzująca wdrażanie
wzorców projektowych w zintegrowanych środowiskach
programistycznych**

Praca magisterska napisana
pod kierunkiem

dr inż. Mariusz Trzaska

Warszawa, sierpień, 2024

Streszczenie

W niniejszej pracy opisano tematykę wzorców projektowych stosowanych we współczesnych językach programowania. Przybliżono zalety i wady ich implementacji oraz przedstawiono teoretyczne aspekty wzorców. Skonstruowano prototyp generujący zindywidualizowane szablony kodu z wzorcami projektowymi gotowymi do integracji. Wstępna konfiguracja generowanego kodu wzorca poprzez mechanizm parametryzowania generatora ułatwia integrację z już istniejącym kodem. Podane podczas konfiguracji parametry zwiększają zgodność obu implementacji. Przeanalizowano sposoby integracji gotowych szablonów z już istniejącymi rozwiązaniami projektowymi. Porównano proponowane rozwiązanie pod względem innowacyjności z podobnymi, już istniejącymi rozwiązaniami. Pozytywnym skutkiem generowania szablonów wzorców jest mniejsze zużycie zasobów koniecznych na ich wdrożenie w projektach informatycznych. W prezentowanej pracy opisano: implementację, funkcjonalność oraz narzędzia wykorzystane do konstrukcji prototypu. Autor pracy stara się pokazać korzyści i możliwości płynące z automatyzacji procesu wdrażania wzorców projektowych w zintegrowanych środowiskach programistycznych.

Słowa kluczowe:

wzorce projektowe, wtyczka, zintegrowane środowisko programistyczne, automatyzacja, parametryzacja, szablon, integracja

Podziękowania

Autor pracy pragnie podziękować kadrze dydaktycznej, personelowi oraz studentom budującym środowisko skupione wokół Polsko-Japońskiej Akademii Technik Komputerowych, za przekazaną wiedzę, atmosferę panującą na uczelni i otrzymane wsparcie, czego efektem jest m. in. możliwość prezentacji niniejszej pracy magisterskiej.

Spis treści

1	WSTĘP	5
1.1	PROBLEM	5
1.2	MOTYWACJA	6
1.3	KONTEKST	6
1.4	REZULTAT	7
1.5	ORGANIZACJA PRACY	7
2	ISTNIEJĄCE ROZWIĄZANIA.....	9
2.1	ROZWIĄZANIA OPARTE O SZTUCZNĄ INTELIGENCJĘ	9
2.1.1	<i>ChatGPT i Tutor Me</i>	9
2.1.2	<i>GitHub Copilot</i>	11
2.2	WTYCZKI.....	14
2.2.1	<i>Go Design Pattern Generator</i>	14
2.2.2	<i>IntelliJ IDEA Design Patterns</i>	16
2.3	PRACE NAUKOWE.....	18
2.4	ZAKRES BADAŃ	19
2.5	PORÓWNANIE ROZWIĄZAŃ	19
3	PROPOZYCJA ROZWIĄZANIA.....	21
3.1	WZORCE PROJEKTOWE	21
3.1.1	<i>Zalety</i>	22
3.1.2	<i>Wady</i>	23
3.1.3	<i>Przyjęte rozwiązanie</i>	23
3.2	WZORCE BEHAWIORALNE	24
3.2.1	<i>Charakterystyka</i>	24
3.2.2	<i>Obserwator</i>	26
3.3	WZORCE STRUKTURALNE	29
3.3.1	<i>Charakterystyka</i>	29
3.3.2	<i>Dekorator</i>	30
3.3.3	<i>Adapter</i>	33
3.4	WZORCE KREACYJNE.....	36
3.4.1	<i>Charakterystyka</i>	36
3.4.2	<i>Metoda Wytwórcza</i>	37
4	NARZĘDZIA I TECHNOLOGIE	41
4.1	KOTLIN	41
4.2	JAVA.....	42
4.3	INTELLIJ PLATFORM PLUGIN SDK	43
4.4	INTELLIJ PLATFORM PLUGIN TEMPLATE	44
4.5	INTELLIJ IDEA CE.....	45
4.6	GIT	46
5	OPIS IMPLEMENTACJI PROTOTYPU	48
5.1	KONFIGURACJA SZABLONU	48
5.2	ROZWIĄZANIA IMPLEMENTACYJNE	50
5.2.1	<i>Klasy AnAction</i>	50

5.2.2	<i>Klasy setters DialogWrapper</i>	53
5.2.3	<i>Klasy main DialogWrapper</i>	54
5.3	DIAGRAM AKTYWNOŚCI	62
5.4	DIAGRAM SEKWENCJI	64
6	PRZYPADKI UŻYCIA	66
6.1	OBSERWATOR	66
6.1.1	<i>Konfiguracja</i>	66
6.1.2	<i>Generator</i>	69
6.1.3	<i>Integracja</i>	71
6.2	DEKORATOR	72
6.2.1	<i>Konfiguracja</i>	73
6.2.2	<i>Generator</i>	75
6.2.3	<i>Integracja</i>	78
6.3	ADAPTER	80
6.3.1	<i>Konfiguracja</i>	80
6.3.2	<i>Generator</i>	82
6.3.3	<i>Integracja</i>	83
6.4	FABRYKA	84
6.4.1	<i>Konfiguracja</i>	84
6.4.2	<i>Generator</i>	85
6.4.3	<i>Integracja</i>	86
6.5	DODATKOWE FUNKCJONALNOŚCI	87
6.5.1	<i>Notyfikacje</i>	87
6.5.2	<i>Wybór ścieżki</i>	88
6.5.3	<i>Walidacja formularza</i>	88
6.5.4	<i>Wczytywanie pól i metod</i>	89
7	PODSUMOWANIE	92
	SPIS LISTINGÓW	93
	SPIS RYSUNKÓW	95
	PRACE CYTOWANE	97

1 Wstęp

Wzorce projektowe to sprawdzone sposoby konstrukcji oprogramowania, które mają na celu spełnienie określonych wymagań dla tworzonych fragmentów kodu. Bazą dla implementowanych wzorców projektowych są szablony kodu oraz specyficzny sposób konstrukcji architektury projektu. Podczas procesu tworzenia oprogramowania często napotyka się problemy, które zostały już rozwiązane przez innych programistów w podobnym kontekście. Zamiast szukać własnego, autorskiego rozwiązania problemu, warto skorzystać z już istniejących rozwiązań. Wzorce projektowe stanowią takie gotowe rozwiązania, które dodatkowo wielokrotnie sprawdziły się jako najlepsze rozwiązanie dla podobnych grup problemów programistycznych.

1.1 Problem

Implementacja wzorca odbywa się według szablonu, który:

- należy zintegrować z już istniejącym kodem,
- może wymagać dodatkowo przebudowy architektury projektu.

W przypadku tworzenia implementacji projektu od początku lub tworzenia od początku wybranego fragmentu istniejącej aplikacji możliwe jest wdrożenie wzorca projektowego na wstępie, a potem w dalszym kroku dopasowywanie implementacji do narzuconych ram istniejącej implementacji oraz wymagań konkretnego wzorca.

Stosowanie wzorców projektowych wymaga od dewelopera:

- identyfikacji i określenia napotkanego w kodzie problemu,
- znajomości wzorca, który taki problem rozwiązuje,
- umiejętnego zaimplementowania wzorca w projekcie.

Etap identyfikacji problemu wraz z etapem znalezienia odpowiedniego wzorca rozwiązującego zidentyfikowany problem muszą zostać rozeznane przez programistę. Trudno jest zastąpić te etapy pracy nad projektem przez inne narzędzia, które wyręczałyby deweloperów. Powód stanowi duża złożoność i głębia analizy zagadnienia. Proces dopasowania szablonu z kodem wzorca do tworzonej implementacji programu jest podobnie do poprzednich etapów trudny do automatyzacji z powodu dużego zróżnicowania możliwych sposobów konstrukcji oprogramowania rozwiązujących zadany problem

Etapem wdrożenia wzorców projektowych, który poddaje się automatyzacji jest proces tworzenia szablonu z kodem dla wybranego wzorca. Podczas procesu programowania możliwe jest pominięcie tego etapu i podjęcie próby integracji szablonu z istniejącym kodem w pamięci i jako jeden etap. Pomija się oddzielne tworzenie implementacji szablonu. W takim podejściu, w przypadku bardziej złożonych implementacji problematyczne może być:

- skomplikowanie kodu,
- dopasowywanie szablonu wzorca do istniejącego kodu, które wymaga choćby częściowej implementacji szablonu,

- fakt, że mniej wprawni programiści mogą mieć problem ze zrozumieniem mechanizmów działania wzorca podczas próby jego integracji z kodem z pominięciem tworzenia szablonu.

Taki stan rzeczy może powodować wzrost kosztów realizacji projektów ze względu na wydłużony czas ich trwania. Etap tworzenia szablonu wzorca pomimo swoich zalet stanowi proces żmudny i czasochłonny. Niniejsza praca proponuje rozwiązanie problemu tworzenia szablonów wzorców projektowych poprzez generowanie gotowych szablonów kodu z wzorcami na podstawie ich nazwy. Dodatkowo istnieje możliwość podania specyficznych dla nich samych parametrów, które usprawniają proces integracji z już istniejącym projektem.

Niniejsza praca nie skupia się na innych etapach wdrażania wzorców projektowych, takich jak:

- rozpoznanie kontekstu w jakim należałoby wdrożyć wzorzec,
- dopasowanie wzorca do dziedziny zastosowań,
- dopasowanie szablonu wzorca do implementacji,
- wyszukanie i analiza już istniejących technologicznych rozwiązań dla wypunktowanych tu etapów.

1.2 Motywacja

Motywację dla podjętej pracy stanowi pragnienie usprawnienia procesu implementacji wzorców projektowych. Nie w każdym projekcie lub kontekście zasadne jest stosowanie wzorców. W nieuzasadnionych przypadkach mogłyby niepotrzebnie skomplikować kod programu i stać się rozwiązaniem na wyrost oczekiwaniom.

W przypadku uzasadnionego zastosowania wzorców projektowych ich implementacja jest procesem bardziej czasochłonnym niż klasyczne podejście do programowania nieuwzględniające stosowanie wzorców. Główną motywację dla podjętych prac stanowi zastosowanie rozwiązania, które usprawni proces tworzenia implementacji, opierającego się o wzorce projektowe. Dodatkową motywację stanowi opracowanie rozwiązania, które uprości zrozumienie sposobu działania i kontekstu zastosowania wzorców projektowych. Oczekiwania wobec przyjętego rozwiązania to skrócenie czasu implementacji, lepsze zrozumienie mechanizmów działania wzorców i łatwiejszy sposób integracji wzorców z już istniejącym projektem.

1.3 Kontekst

Kontekst tematyki zarysowanej w niniejszej pracy związany jest z tworzeniem oprogramowania w języku *Java*, szczególnie w odniesieniu do funkcjonalności zaprezentowanego prototypu. Poruszane kwestie teoretyczne są bardziej uniwersalne i referują do szerokiej tematyki zastosowania wzorców projektowych. W pracy skupiono się szczególnie na sposobach wdrażania wzorców w projektach informatycznych.

Język *Java* jest językiem w pełni obiektywnym, umożliwiającym wdrożenie wielu rodzajów wzorców projektowych. Wokół środowiska języka *Java* skupionych jest wiele dedykowanych technologii, usprawniających proces tworzenia oprogramowania. Przykładem może być zintegrowane

środowisko programistyczne (*IDE*) stanowiące funkcjonalną platformę o szerokim spektrum zastosowań programistycznych do tworzenia, edycji i kompilacji kodu.

Środowisko firmy *JetBrains* o nazwie *IntelliJ IDEA* w wersji *Community Edition* o otwarto-źródłowym (*open source*) kodzie jest przykładem *IDE*, które cechuje się mechanizmem rozszerzalnej funkcjonalności poprzez zastosowanie elastycznego systemu wtyczek. Rozszerzenie podstawowej funkcjonalności oferowanej przez środowiska *IntelliJ IDEA* umożliwia instalacja nowych wtyczek do tego programu. W ten sposób znacząco rozszerza się zakres standardowego tworzenia, edycji i kompilacji tworzonego kodu w *IDE*.

Realizowany prototyp został osadzony w kontekście środowiska języka *Java* jako elastyczna wtyczka do programu *IntelliJ IDEA*. Projekt ma charakter generyczny i umożliwia tworzenie szablonów wzorców projektowych na podstawie ustawionych na wejściu parametrów inicjalizacyjnych podczas tworzenia implementacji. W zarysowanym kontekście możliwa jest automatyzacja procesu wdrażania wzorców projektowych.

1.4 Rezultat

Powstała praca ma na celu uproszczenie zastosowania wzorców projektowych na etapie integracji szablonoego kodu źródłowego wzorca z istniejącą w projekcie implementacją. Opisane uproszczenie możliwe jest dzięki automatyzacji procesu generowania kodu wybranych wzorców. Takie podejście skutkuje dodatkowo:

- lepszym poznaniem kontekstu w jakim należy zastosować wybrany wzorzec,
- większą znajomością mechanizmów działania wzorca,
- poprawioną umiejętnością integracji konkretnego wzorca z już istniejącym kodem.

Pobocznym rezultatem realizacji niniejszej pracy jest prototyp w postaci elastycznej wtyczki do platformy *IntelliJ IDEA*, rozwiązujący opisany problem tworzenia oprogramowania w oparciu o wzorce projektowe poprzez generowanie gotowych szablonów wzorców.

Takie podejście umożliwia wywołanie kreatora konfiguracji dla generowania wybranego wzorca poprzez skrót klawiaturowy podczas procesu tworzenia oprogramowania. Możliwe jest proste przejście od procesu tworzenia implementacji do generowania kodu wzorca i powrót do dalszej implementacji. Nowo wygenerowany kod najczęściej wymaga dopasowania do już istniejącej implementacji.

Kreator parametryzacji dla nowo generowanego kodu wzorca wymaga podania cech specyficznych dla utworzenia implementacji. Celem jest zapewnienie w jak największym stopniu zgodności nowej implementacji z już istniejącą w kodzie aplikacji. Efekt stanowi łatwiejsze dopasowanie nowego, wygenerowanego kodu z kodem istniejącym w projekcie.

1.5 Organizacja pracy

W Rozdziale 2. opisano już istniejące rozwiązania nakreślonego problemu, dotyczącego tworzenia szablonów wzorców projektowych i możliwości automatyzacji procesu ich integracji z

istniejącą implementacją. Opisano różnice pomiędzy przyjętym w pracy rozwiązaniem, a rozwiązaniami już istniejącymi.

W Rozdziale 3. podjęto teoretyczne rozważania nad rozwiązaniem opisanego w pracy problemu zastosowania wzorców projektowych w projektach programistycznych. Opisano wzorce projektowe, diagramy klas, zalety i wady stosowania wzorców oraz propozycję rozwiązania dla podjętej w pracy tematyki.

W Rozdziale 4. przybliżono wykorzystane w celu konstrukcji prototypu wtyczki technologie informatyczne. Skonstruowany prototyp obrazuje rozwiązanie zarysowanej w pracy problematyki w sposób praktyczny.

W Rozdziale 5. opisano na podstawie wybranych fragmentów kodu implementację skonstruowanego prototypu wtyczki. Określono funkcjonalność wybranych fragmentów kodu, przedstawiono diagram aktywności, sekwencji oraz wykorzystanie systemów pomocniczych.

W Rozdziale 6. przedstawiono działanie prezentowanego systemu na podstawie opisanych przypadków użycia. Podzielono funkcjonalność oferowanego rozwiązania na sekcje i opisano je w kontekście wybranych przypadków. Opis funkcjonalności dotyczy funkcji kluczowych oraz pomocniczych.

2 Istniejące rozwiązania

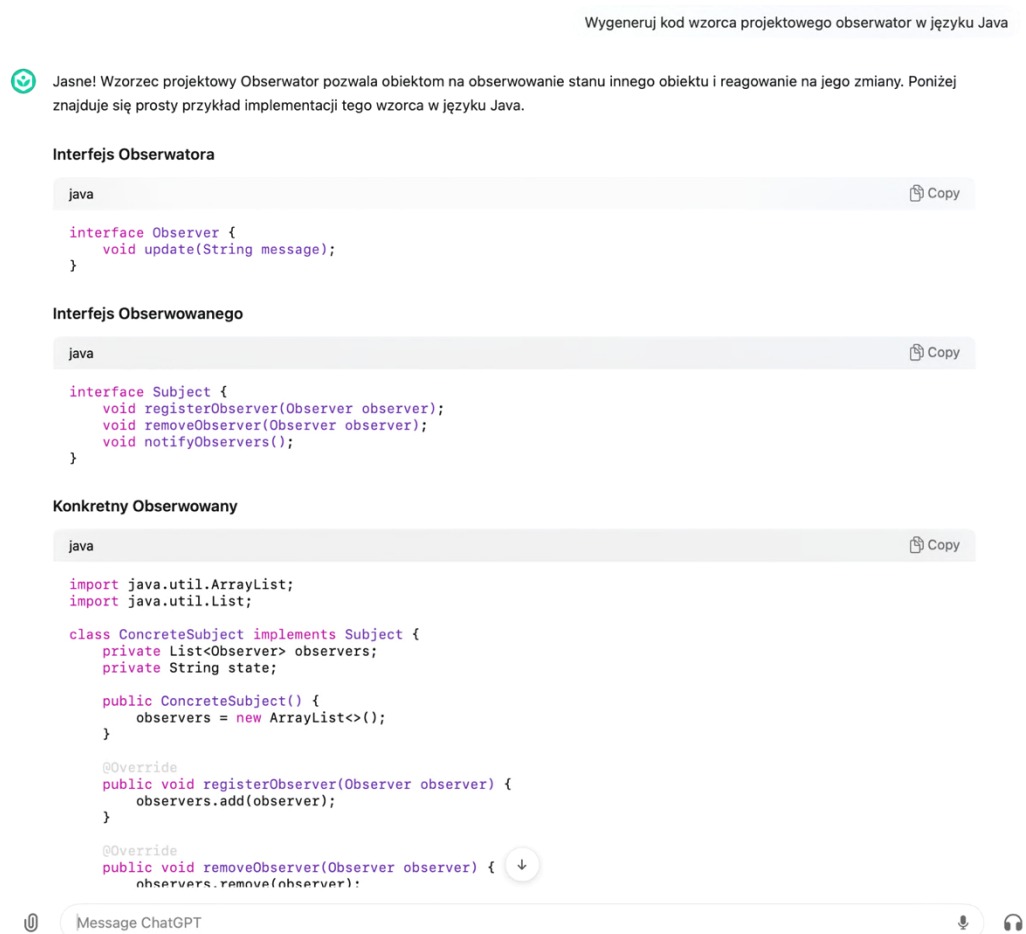
W rozdziale podjęto analizę istniejących i wykorzystywanych rozwiązań technologicznych pod kątem podjętej w niniejszej pracy problematyki zastosowania wzorców projektowych w projektach informatycznych.

2.1 Rozwiązania oparte o sztuczną inteligencję

Sztuczna inteligencja stanowi stosunkowo młodą dziedzinę informatyki, cechującą się bardzo dużą dynamiką wzrostu w różnych dziedzinach zastosowań. Podrozdział opisuje wybrane zastosowania sztucznej inteligencji w kontekście wzorców projektowych.

2.1.1 ChatGPT i Tutor Me

ChatGPT [1] to rodzaj sztucznej inteligencji w dziedzinie dużych modeli językowych (*LLM*). Jest ukierunkowany na generowanie tekstu mającego przypominać ludzką wymowę. Posiada szeroką gamę zastosowań, jednym z nich jest wykorzystywanie *ChatGPT* do generowania kodów programów



Rysunek 1. Fragment nr 1 kodu źródłowego wzorca projektowego Obserwator wygenerowany w systemie Tutor Me. Źródło: opracowanie własne.

w miejsce tekstu przypominającego ludzkie dialogi. Ten rodzaj sztucznej inteligencji jest trenowany ogromną ilością danych internetowych, na bazie których jest w stanie symulować tekstowe wypowiedzi, w rzeczywistości trudne do odróżnienia od wypowiedzi ludzkich.

Możliwe jest skierowanie żądania (*prompt*) do *ChatGPT* mającego na celu wygenerowanie kodu dla wybranego wzorca projektowego w dowolnym języku programowania.

Jedną z wielu usług powstałych na bazie *ChatGPT* jest m.in. *Tutor Me*, będący dziełem serwisu edukacyjnego *Khan Academy* [2]. To odgałęzienie korzysta z modelu danych agregujących dane z różnych dziedzin nauki, w tym dane naukowe z dziedziny inżynierii oprogramowania. W celu porównania wyników pracy sztucznej inteligencji w kontekście wzorców projektowych i proponowanego w projekcie rozwiązania skierowano do programu *Tutor Me* zapytanie w celu wygenerowania kodu wzorca projektowego obserwator w języku *Java*. *Rysunek 1* i *Rysunek 2* przedstawiają odpowiedź systemu AI na polecenie wygenerowania kodu źródłowego wzorca projektowego *Obserwator*.

Konkretny Obserwator

```
java
class ConcreteObserver implements Observer {
    private String name;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        System.out.println(name + " received update: " + message);
    }
}
```

Przykładowe użycie

```
java
public class ObserverPatternDemo {
    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();

        Observer observer1 = new ConcreteObserver("Observer 1");
        Observer observer2 = new ConcreteObserver("Observer 2");

        subject.registerObserver(observer1);
        subject.registerObserver(observer2);

        subject.setState("State 1");
        subject.setState("State 2");

        subject.removeObserver(observer1);

        subject.setState("State 3");
    }
}
```

W tym przykładzie:

- *Observer* to interfejs definiujący metodę *update*, którą muszą implementować wszystkie obserwatory.
- *Subject* to interfejs definiujący metody do rejestrowania, usuwania i powiadamiania obserwatorów.
- *ConcreteSubject* to klasa implementująca *Subject*, która przechowuje listę obserwatorów i powiadamia ich o zmianach stanu.
- *ConcreteObserver* to klasa implementująca *Observer*, która reaguje na aktualizacje wysyłane przez *ConcreteSubject*.

Czy masz pytania dotyczące tej implementacji?

Message ChatGPT

Rysunek 2. Fragment nr 2 kodu źródłowego wzorca projektowego Obserwator wygenerowany w systemie Tutor Me. Źródło: opracowanie własne.

Dużą zaletą rozwiązania bazującego na *ChatGPT* jest elastyczność uzyskiwanych konfiguracji. Możliwe jest wygenerowanie dowolnego wzorca w dowolnym języku programowania. Korzystne są także opisy dla generowanych implementacji ułatwiające ich zrozumienie.

Wadą generowanych w ten sposób implementacji jest niski poziom adaptacji z już istniejącym oprogramowaniem. Opisywane rozwiązanie wymaga ręcznego kopiowania wygenerowanego kodu do adaptowanego projektu. Trudnością jest modyfikacja tak wygenerowanych implementacji w celu dopasowania lub wstępnego przystosowania ich do integracji z projektem. Aby wygenerować implementację, która miałaby być bardziej przystępna do wdrożenia w tworzonej rozwiązaniu należałoby skonstruować precyzyjnie dobrane zapytanie do *Tutor Me*, co stanowi zadanie trudne i czasochłonne. Otrzymany wynik jest niepewny, ponieważ pomimo sprecyzowanego zapytania możliwe jest otrzymanie zróżnicowanych implementacji.

2.1.2 GitHub Copilot

GitHub Copilot jest rodzajem sztucznej inteligencji z rodziny dużych modeli językowych. W przeciwieństwie do *ChatGPT*, *GitHub Copilot* został wytrenowany danymi, na które składały się przede wszystkim kody źródłowe programów z serwisu *GitHub*. Repozytorium *GitHub* jest największym i najpopularniejszym [3] internetowym repozytorium kodów źródłowych programów, które mogą być w tym serwisie zamieszczane jako publicznie dostępne na licencji *otwarto-źródłowej (open-source)*. Licencje *otwarto-źródłowe* w odniesieniu do programowania zapewniają bezpłatny dostęp do kodu programu z możliwością dowolnych jego modyfikacji i użycie przez licencjobiorcę. Trenowanie modelu językowego sztucznej inteligencji *GitHub Copilot* przy użyciu znacznych ilości kodów źródłowych sprawiło, że jest to narzędzie pomocne w procesie tworzenia oprogramowania.

GitHub Copilot jest dostępne dla użytkowników m.in. w formie wtyczek do najpopularniejszych zintegrowanych środowisk programistycznych (*IDE*). W takim środowisku wydanym przez firmę *Microsoft* o nazwie *Visual Studio Code* możliwa jest instalacja wtyczki *GitHub Copilot* w formie *chat-a* oraz w formie asystenta dla procesu tworzenia oprogramowania.

Asystent *GitHub Copilot* analizuje proces wpisywania linii kodu przez programistę i na podstawie tworzonych implementacji proponuje swoje uzupełniające kod programisty implementacje. Deweloper może zaakceptować uzupełnienie lub wprowadzić modyfikację do proponowanego rozwiązania poprzez doprecyzowanie zapytania, może także propozycję odrzucić lub zaakceptować i samodzielnie zmodyfikować kod. Cały proces współpracy z asystentem odbywa się w sposób dynamiczny podczas tworzenia kodu programu.

W celach porównawczych proponowanego w projekcie rozwiązania z usługą oferowaną przez *GitHub Copilot* w kontekście generowania kodu wzorców projektowych skierowano zapytanie do systemu *GitHub Copilot* o wygenerowanie wzorca projektowego *Obserwator* podczas procesu tworzenia oprogramowania w środowisku *Visual Studio Code*. *Listing 1* przedstawia odpowiedź systemu AI na wysłane zapytanie.

Listing 1. Kod źródłowy wzorca projektowego Obserwator wygenerowany przez AI GitHub Copilot.

```
// Observer interface
interface Observer {
    void update(String message);
}

// Subject interface
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// ConcreteObserver class
class ConcreteObserver implements Observer {
    private String name;
    private String message;

    public ConcreteObserver(String name) {
        this.name = name;
    }

    @Override
    public void update(String message) {
        this.message = message;
        display();
    }

    public void display() {
        System.out.println(name + " received message: " + message);
    }
}

// ConcreteSubject class
class ConcreteSubject implements Subject {
    private List<Observer> observers;
    private String message;

    public ConcreteSubject() {
        observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }
}
```

```

        public void setMessage(String message) {
            this.message = message;
            notifyObservers();
        }
    }

    // Client class
    class Client {
        public static void main(String[] args) {
            ConcreteSubject subject = new ConcreteSubject();
            ConcreteObserver observer1 = new ConcreteObserver("Observer 1");
            ConcreteObserver observer2 = new ConcreteObserver("Observer 2");
            ConcreteObserver observer3 = new ConcreteObserver("Observer 3");

            subject.registerObserver(observer1);
            subject.registerObserver(observer2);
            subject.registerObserver(observer3);

            subject.setMessage("Hello World!");
        }
    }

    // Main class
    public class ObserverPattern {
        public static void main(String[] args) {
            Client.main(args);
        }
    }
}

```

Analogicznie do rozwiązania oferowanego przez *ChatGPT* w *Rozdziale 2.1.1.* w przypadku *GitHub Copilot* wadę stanowi konieczność skierowania do systemu sztucznej inteligencji precyzyjnie skonstruowanego zapytania. Cel stanowi możliwość wygenerowania kodu wzorca projektowego dopasowanego do już istniejącej implementacji. *GitHub Copilot* generuje ogólny kod wzorca, który należy dopasować do już istniejącego oprogramowania. Konstrukcja zapytania, które zmieniłoby generowanie kodu w sposób, który ułatwiłby integrację z już istniejącym rozwiązaniem jest zadaniem trudnym, czasochłonnym i w specyficznych przypadkach może okazać się mało adekwatnym z już istniejącym rozwiązaniem ze względu na możliwe duże zróżnicowanie generowanych wyników przez system sztucznej inteligencji.

Proponowane w realizowanym projekcie rozwiązanie ma bardziej precyzyjne możliwości dopasowania generowanego kodu do istniejącej implementacji ze względu na zastosowany mechanizm wstępnej parametryzacji procesu generowania kodu. Dodatkowo istnieje mechanizm automatycznego wykrywania zmiennych i metod zaimplementowanych w adaptowanym kodzie. Proponowane rozwiązanie ma bardziej ścisłe i celne możliwości adaptacji do istniejącego kodu w przeciwieństwie do *GitHub Copilot*. System AI to rozwiązanie, które ma znacznie szersze możliwości generowania kodu, ale w zakresie bardziej ogólnym.

Kolejną korzyścią proponowanego rozwiązania jest możliwość automatycznego tworzenia nowych, wymaganych plików dla już istniejącej implementacji. W przypadku *GitHub Copilot* nadal

istnieje konieczność utworzenia nowych plików źródłowych i kopiowania do nich nowej zawartości z kodem.

Dużą zaletą systemu *GitHub Copilot* jest możliwość szybkiego podglądu proponowanej zawartości z kodem przed jego zatwierdzeniem w projekcie. Po akceptacji możliwe jest natychmiastowe edytowanie wygenerowanego kodu. To rozwiązanie sprawdza się lepiej przy konstrukcji pojedynczych algorytmów, natomiast w przypadku generowania całych zestawów powiązanych funkcjonalnie plików z kodem wymaga więcej czasu na konfigurację. W przypadku wzorców projektowych mamy do czynienia właśnie z wieloma powiązаныmi ze sobą funkcjonalnie plikami źródłowymi.

Niewątpliwą przewagą *GitHub Copilot* nad proponowanym rozwiązaniem jest możliwość generowania kodu w każdym języku programowania, a więc elastyczność zastosowania jest bardzo wysoka. W realizowanym prototypie możliwa jest łatwiejsza adaptacja jedynie do programów napisanych w języku *Java*.

Kolejną korzyścią dla programisty jest możliwość konstrukcji zapytań (*prompt*) skierowanych do *GitHub Copilot* w językach narodowych. Możliwe jest dzięki temu bardziej precyzyjne konstruowanie zapytań. Wraz ze wzrostem złożoności zapytania w języku narodowym może jednak maleć precyzja translacji na Język angielski, który jest podstawowym semantycznym źródłem dla generowania wyników w systemie *GitHub Copilot*. Konwencja zakłada pisanie oprogramowania w Języku angielskim, więc modele uczenia maszynowego *GitHub Copilot* zostały wytrenowane głównie w tym języku. Złożone zapytania w Języku polskim mogą skutkować nietrafionymi wynikami działania algorytmu. Analogiczne do Języka polskiego precyzyjne konstrukcje językowe po angielsku pozostają w dalszym ciągu zadaniem trudnym i czasochłonnym.

2.2 Wtyczki

W podrozdziale opisano istniejące, podobne rozwiązania do oferowanego w niniejszej pracy w postaci wtyczek do programu *IntelliJ IDEA*.

2.2.1 Go Design Pattern Generator

W oficjalnym repozytorium firmy *JetBrains* z wtyczkami dla programu *IntelliJ IDEA* istnieje podobne rozwiązanie do proponowanego w pracy. W postaci instalowanej dodatkowo wtyczki, możliwe jest generowanie kodu wzorców projektowych w języku programowania *Golang* [4]. Narzędzie umożliwia kreację czterech rodzajów wzorców projektowych [stan na 31.07.2024 r.].

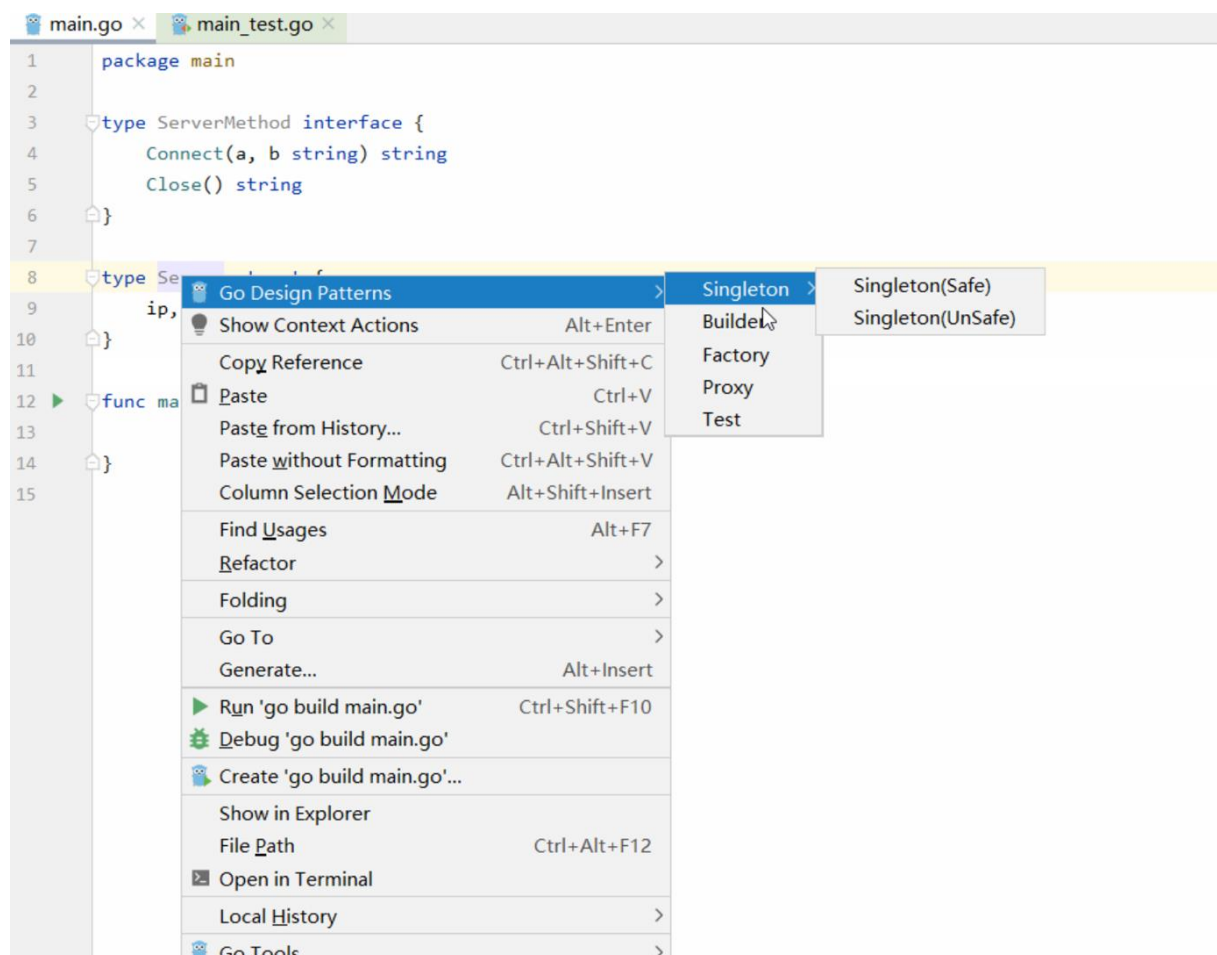
Zasadnicza różnica pomiędzy językiem *Java*, a językiem *Golang* polega na odmiennym paradygmacie programowania. W przypadku języku *Java* zastosowanie ma głównie paradygmat programowania obiektowego [5], choć możliwe jest także programowanie w paradygmatach:

- funkcyjnym,
- generycznym,
- asynchronicznym,
- imperatywnym.

W języku *Golang* stosuje się na pierwszym miejscu paradygmat imperatywny [6] z możliwością programowania w paradygmatach:

- funkcyjnym,
- generycznym,
- strukturalnym,
- współbieżnym.

Język *Golang* w bardzo niewielkim stopniu umożliwia programowanie w paradygmacie obiektowym w przeciwieństwie do języka *Java*. Proponowane rozwiązanie generowania wzorców projektowych dla języka *Java* bazuje na obiektowym paradygmacie programowania i dekomponuje generowany wzorzec projektowy na wiele plików w kodem źródłowym. Każdy z wygenerowanych plików stanowi oddzielną klasę lub interfejs. Podobne podejście w języku *Golang* jest utrudnione, dlatego zastosowany we wtyczce dla języka *Golang* sposób generowania wzorców projektowych jest odmienny od rozwiązania proponowanego w niniejszej pracy.

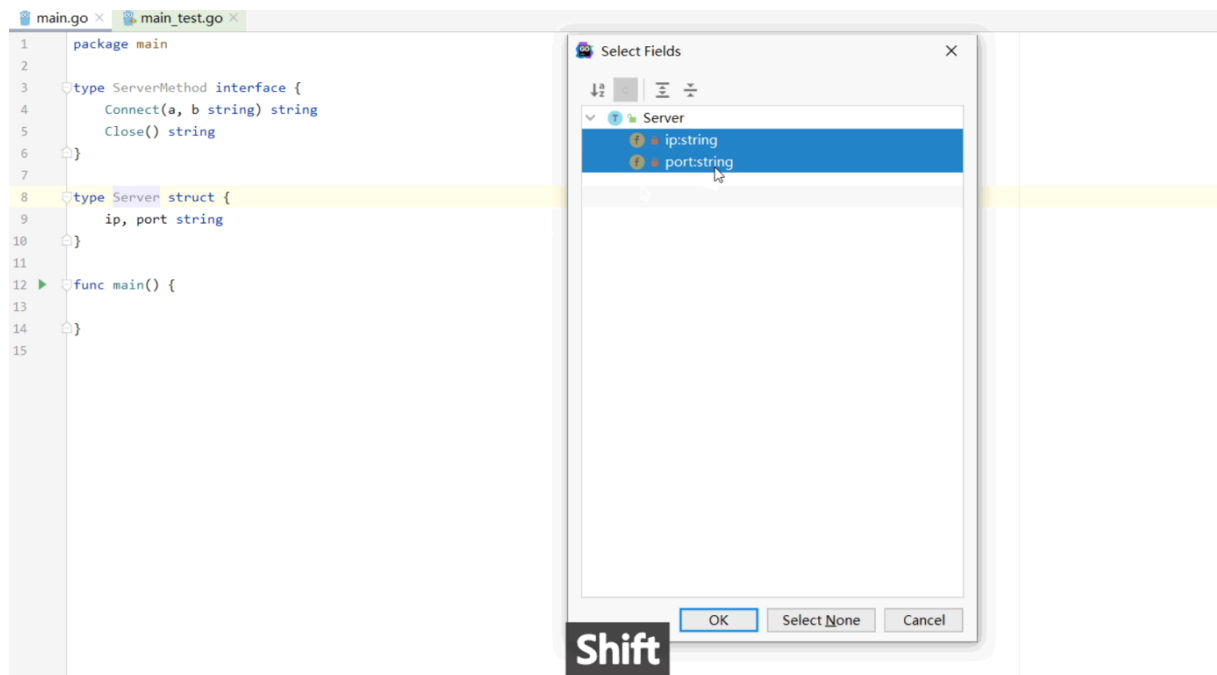


Rysunek 3. Interfejs nr 1 wtyczki generującej wzorce projektowe w języku Go. Źródło: [7]

Podstawowa różnica uwidacznia się w odmiennym sposobie docelowego generowania kodu wzorca, który polega na generowaniu całej implementacji do pojedynczego pliku. Kolejną różnicą jest sposób generowania oprogramowania na podstawie już istniejącego kodu. W prezentowanym w pracy rozwiązaniu, obok takiego podejścia istnieje możliwość generowania kodu wzorca na bardziej abstrakcyjnym i niezależnym od pojedynczego pliku z kodem poziomem. Zastosowane podejście

zapewnia bardziej uniwersalny charakter tworzonego kodu względem pojedynczej klasy na korzyść całego projektu, ponieważ obejmuje szerszy zakres zastosowania.

Podobieństwo opisanego dla języka *Golang* rozwiązania z podejściem przyjętym w niniejszej pracy uwidacznia się w zastosowanym mechanizmie interakcji z użytkownikiem poprzez zastosowanie formularzy. W przypadku języka *Golang* formularze zostały bardziej uproszczone. Kolejnym podobieństwem jest możliwość szybkiego wygenerowania kodu wzorca podczas tworzenia implementacji, które nie powoduje rozproszenia uwagi programisty, zapewniając ciągłość procesu kodowania. Sposób działania wtyczki dla języka *Go* został zobrazowany na *Rysunku 3* i *Rysunku 4*.

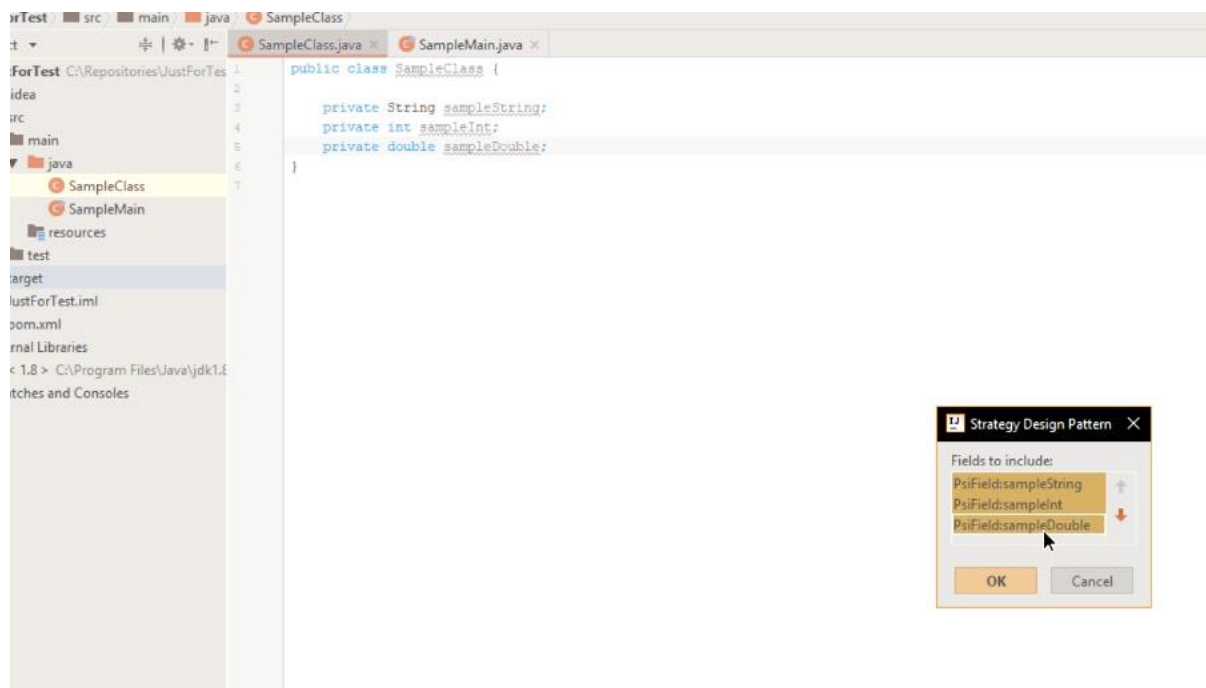


Rysunek 4. Interfejs nr 2 wtyczki generującej wzorce projektowe w języku Go. Źródło: [7]

2.2.2 IntelliJ IDEA Design Patterns

Wtyczka do programu *IntelliJ IDEA* o nazwie *Design Patterns* [8] stanowi jedną z dwóch alternatyw umożliwiających generowanie kodu wzorców projektowych jako zewnętrzne rozszerzenie do tego programu. W *Rozdziale 2.2.1.* opisano pierwszą z alternatyw.

We wtyczce możliwe jest generowanie kodu podczas tworzenia implementacji bez odrywania uwagi programisty od tego zadania. Wtyczka udostępnia prosty formularz w celu indywidualizacji kodu jaki ma zostać wygenerowany. Dodatkową funkcjonalność stanowi nawiązanie do już istniejącego oprogramowania podczas procesu generowania wzorców, która zwiększa kompatybilność nowego kodu z kodem już istniejącym. Formularz wyświetlany podczas korzystania z wtyczki jest mocno uproszczony i ma za zadanie dopasowanie do już istniejących w kodzie deklaracji, na podstawie których następuje generowanie kodu w dalszym etapie. Na *Rysunku 5* przedstawiono interfejs wbudowany we wtyczkę *Design Patterns*.



Rysunek 5. Interfejs wtyczki generującej wzorce projektowe w języku Java. Źródło: [9]

Jako brakującą funkcjonalność w udostępnionej społeczności wtyczce można uznać brak dekompozycji generowanej implementacji na wiele klas. Powstały kod umieszczony zostaje w jednej klasie. Na bazie tej klasy zostało wysłane żądanie generowania kodu poprzez wyświetlony formularz. Formularz jest wykorzystywany przed procesem generowania kodu w celu ściślejszego dopasowania nowego oprogramowania do już istniejącej implementacji. W celu jeszcze lepszego dopasowania nowych wyników do istniejących rozwiązań formularz mógłby zawierać więcej opcji konfiguracyjnych. Wynik działania oprogramowania wtyczki stanowi wygenerowany, przykładowy kod źródłowy pokazany na Rysunku 6.

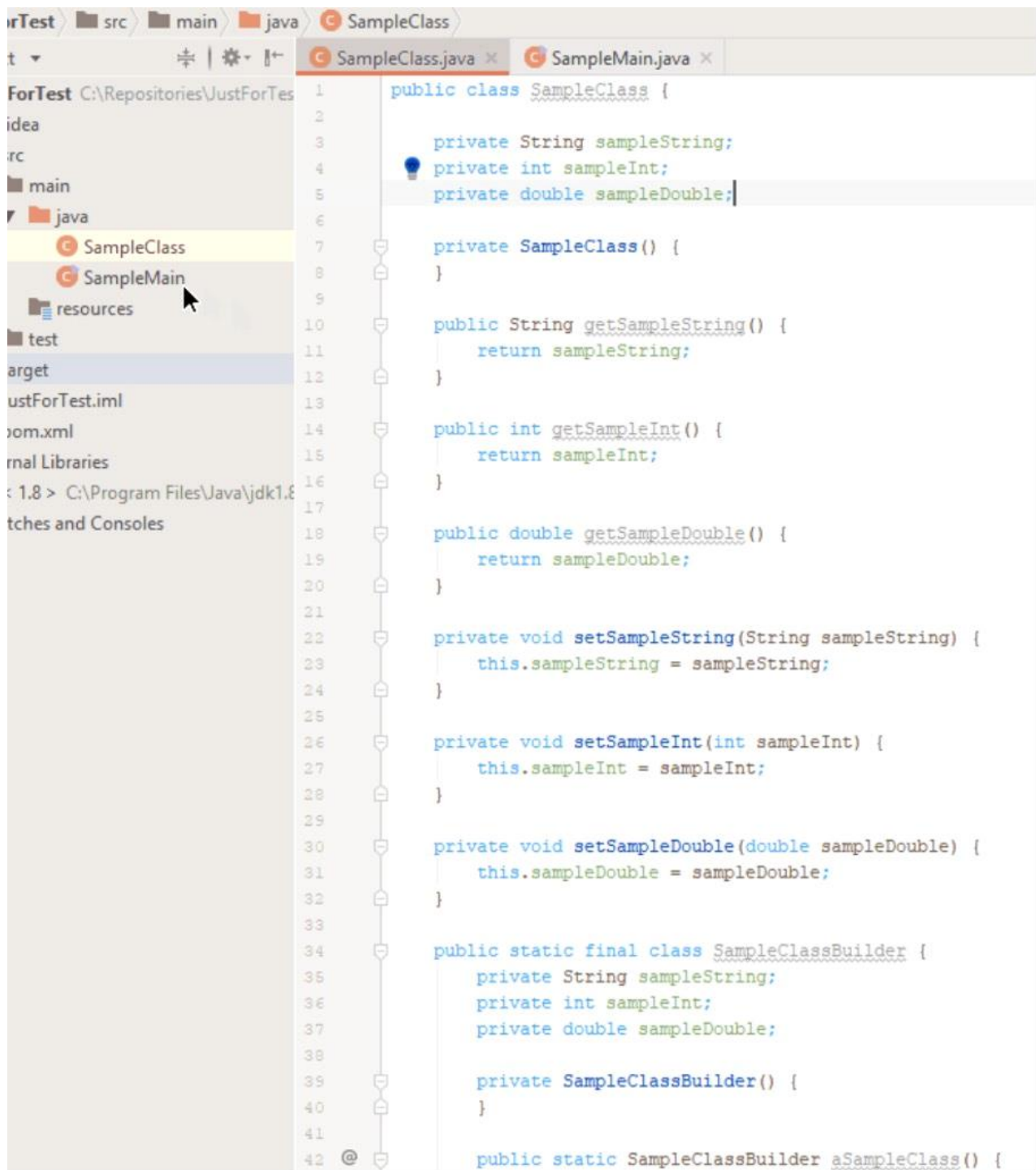
Podobieństwa pomiędzy proponowanym w pracy rozwiązaniem, a opisaną wtyczką *Design Patterns* są następujące:

- możliwość generowania kodu podczas pisania bieżącej implementacji,
- mechanizm wczytywania różnych rodzajów deklaracji w tworzonym kodzie na rzecz wykorzystania przez generator.

Różnice pomiędzy rozwiązaniami uwidaczniają się w:

- braku dekompozycji generowanego kodu na większą liczbę klas,
- braku formularzy uwzględniających dopasowanie w większym stopniu nowego kodu do już istniejącego.

Zastosowane w pracy rozwiązanie stara się wyjść naprzeciw opisanym brakom, przy jednoczesnym zachowaniu pożądaných podobieństw poprzez zastosowanie mechanizmów kompensujących te braki.



Rysunek 6. Kod źródłowy wzorca projektowego wygenerowany przez wttyczkę w języku Java. Źródło: [9]

2.3 Prace naukowe

W zakresie prac naukowych istnieje rozwiązanie odpowiadające na wymagania odnośnie do wspomagania procesu tworzenia implementacji poprzez zastosowanie wzorców projektowych. W Polsko-Japońskiej Akademii Technik Komputerowych w 2020 r. powstała praca magisterska „Narzędzia wspomagające implementację wzorców projektowych w zintegrowanych środowiskach programistycznych” [10] opisująca podobny problem.

Podstawowa różnica pomiędzy przytoczoną pracą magisterską a rozwiązaniem proponowanym w prezentowanej pracy dotyczy progu wejścia dla zainicjowania narzędzia w celu rozpoczęcia generowania wzorców projektowych w realizowanym projekcie. Próg wejścia jest tutaj zdefiniowany jako czynności niezbędne, aby możliwe było rozpoczęcie korzystania z prototypu oraz przygotowanie wymaganych danych wejściowych. Prezentowane w przytoczonej pracy rozwiązanie zakłada istnienie implementacji, na bazie której nastąpi generowanie kodu. Takie rozwiązanie podwyższa próg wejścia. Prezentowane w mojej pracy rozwiązanie zaspokaja potrzebę jak najniższego progu wejścia dla użycia prototypu w celu wygenerowania wzorców projektowych.

W zaprezentowanej przeze mnie pracy rozwiązanie problemu wspomagania programisty podczas tworzenia oprogramowania w zakresie wdrażania wzorców projektowych stanowi brak wymaganej wstępnej implementacji przed wykorzystaniem prototypu. Rozwiązanie alternatywne stanowi niezbędna konieczność konfiguracji wstępnych, wymaganych ustawień poprzez formularz konfiguracyjny przed procesem generowania.

W moim rozwiązaniu możliwe jest także generowanie kodu na podstawie już istniejącej implementacji w celu ściślejszego zintegrowania nowego kodu z kodem już istniejącym. Powiązanie obydwu kodów opiera się wyłącznie na podstawowych deklaracjach pól i metod w klasach nie wymagając dodatkowych, wstępnych konstrukcji programistycznych. W dodatku opisane powiązanie pozostaje opcjonalne w użyciu i nie jest obligatoryjne w celu zainicjowania procesu generowania wzorców projektowych podczas procesu tworzenia implementacji. W tym sensie prezentowane przeze mnie rozwiązanie posiada niższy próg wejścia niż zaprezentowane w przytoczonej pracy magisterskiej.

2.4 Zakres badań

Szukając zasadności realizacji prezentowanej pracy nie uwzględniono rozwiązań przyjętych w innych niż *JetBrains IntelliJ IDEA* zintegrowanych środowiskach programistycznych. Inne rozwiązania dotyczą zakresem wspomagania programisty w implementacji wzorców projektowych. W prezentowym rozwiązaniu skupiono się na analizie rozwiązań mających zastosowanie wyłącznie w zintegrowanym środowisku programistycznym firmy *JetBrains*.

2.5 Porównanie rozwiązań

Wspierana przez *ChatGPT* usługa generowania implementacji przewyższa rozwiązanie oferowane w niniejszym projekcie pod kątem elastyczności w doborze języka programowania. Kod wygenerowany przez *ChatGPT* łatwiej zrozumieć dzięki dodatkowym opisom.

Oferowane rozwiązanie przewyższa *ChatGPT* pod kątem adaptacji wygenerowanych implementacji do istniejącego rozwiązania. Mechanizm parametryzacji kodu przed procesem jego generowania we wtocze jest łatwiejszy w zastosowaniu praktycznym niż tworzenie precyzyjnych zapytań do *ChatGPT*. Dodatkowo kreowane rozwiązanie nie wymaga kopiowania kodu pomiędzy programami, gdyż generowanie kodu ma miejsce bezpośrednio w realizowanym projekcie eliminując problem niskiej przenośności.

System *GitHub Copilot* umożliwia wygenerowanie wielu rodzajów algorytmów w każdym języku programowania z natychmiastową możliwością edycji. W porównaniu do proponowanego

rozwiązania *GitHub Copilot* ma większe trudności w generowaniu gotowych zestawów plików źródłowych składających się na konkretny wzorzec projektowy. W większości wzorców projektowych funkcjonalność oferowana przez zastosowanie wzorca jest zapewniona przez połączenie wielu plików z kodem w jedną funkcjonalną całość. Niemożliwe jest także tworzenie nowych plików w strukturze projektu, to zadanie spoczywa na programiście.

Wtyczka *Go Design Pattern Generator* zapewnia odmienne podejście do generowania wzorców projektowych, bardziej dopasowane do języka *Golang*. Język firmy *Google* opiera się na odmiennych paradygmatach programowania niż język *Java*. Główna różnica polega na próbie implementacji wzorców w paradygmacie imperatywnym w opozycji do paradygmatu obiektowego.

Wtyczka *Design Patterns* dla języka *Java* zapewnia prosty interfejs do komunikacji z programistą w celu wyszczególnienia powiązań istniejących w implementowanym kodzie, z kodem który ma zostać wygenerowany. Wygenerowany kod swoim zakresem obejmuje pojedynczą klasę i brakuje bardziej złożonych mechanizmów komunikacji z programistą w celu lepszego dopasowania kodów źródłowych.

W celu uproszczenia rozwiązania istniejącego w wybranym opracowaniu naukowych, postanowiono zminimalizować próg wejścia i wymagania wstępne, niezbędne do zainicjowania procesu generowania wzorców projektowych poprzez prototyp.

Fundamentalną kwestią jest brak wsparcia ze strony społeczności programu *IntelliJ IDEA* dla rozwoju narzędzia ułatwiającego implementację wzorców projektowych, dedykowanemu językowi *Java*. Nie istnieje także inicjatywa dla budowy takiego narzędzie ze strony firmy *JetBrains*.

Proponowane w pracy rozwiązanie stara się podjąć przytoczone kwestie i zaproponować rozwiązanie problemów je dotyczących.

3 Propozycja rozwiązania

Niniejszy rozdział wychodzi naprzeciw zarysowanej w pracy problematyce zastosowania wzorców projektowych w realizowanych projektach informatycznych. Automatyzacja etapu tworzenia szablonów wzorców umożliwia bardziej elastyczne podejście podczas procesu tworzenia oprogramowania. Większa elastyczność uwarunkowana jest poprzez:

- mniejszy próg wejścia do stosowania wzorców,
- zwiększenie wydajności programistów,
- mniejsze koszty tworzenia oprogramowania,
- poprawę jakości oprogramowania,
- zwiększone możliwości adaptacji nowego kodu do już istniejącego,
- mniejsze ryzyko wydania błędnie działającego oprogramowania.

W rozdziale skupiono się na teoretycznych podstawach przytoczonej tematyki.

3.1 Wzorce projektowe

Definicja wzorców projektowych [11] odnosi się do procesu tworzenia oprogramowania. Podczas tego procesu napotyka się problemy natury:

- funkcjonalnej,
- strukturalnej,
- behawioralnej.

Problemy funkcjonalne próbują odpowiedzieć na pytanie odnośnie do funkcji lub celu w jakim działa dany fragment oprogramowania, innymi słowy: dlaczego tworzy się dany fragment kodu?

Problemy strukturalne dotyczą sposobu konstrukcji programistycznych zawartych w kodzie celem realizacji określonej funkcjonalności, innymi słowy: w jaki sposób należy zbudować oprogramowanie w celu realizacji wymaganej funkcjonalności?

Problemy behawioralne skupiają się na zachowaniu danych fragmentów kodu podczas pracy programu. Zalicza się do tego opisu sposób zmiany stanów lub danych programu podczas realizacji określonych celów, innymi słowy: jak mają zachowywać się zaimplementowane funkcjonalności pod wpływem danych na wejściu w celu kreacji danych wyjściowych?

Opisane problemy wg. dzisiejszego stanu wiedzy i rozwoju technologii informacyjnych zostały już rozwiązane w bardzo wielu przypadkach użycia. Stało się to możliwe dzięki wielokrotnym próbom w jaki programiści podchodzili do tych samych problemów. Wiele tych prób okazało się podobnych w konkretnych kontekstach. Obok podobieństwa tych rozwiązań kluczowa stała się skuteczność i trafność w podejściu do rozwiązania danego problemu. Zbiór tych:

- podobnych,
- właściwych,
- skutecznych

praktyk programistycznych rozwiązujących określony problem programistyczny określono mianem wzorców projektowych. Stanowią zbiór zasad wg. których należy konstruować oprogramowanie pod kątem: funkcjonalnym, strukturalnym i behawioralnym, aby możliwe było realizowane zdefiniowanych założeń programistycznych w sposób możliwie najlepszy i sprawdzony. Proponowane rozwiązania są poparte praktyką, stosowane i zalecane przez wielu programistów.

Wzorce projektowe [12] wg. przyjętej definicji to sprawdzone rozwiązania dla powtarzających się problemów, które pojawiają się podczas projektowania oprogramowania. Zostały one opracowane na podstawie doświadczeń wielu programistów i inżynierów oprogramowania. Wzorce projektowe są abstrakcyjnymi szablonami, które mogą być stosowane w różnych kontekstach programistycznych, aby rozwiązywać określone problemy projektowe.

3.1.1 Zalety

Do wybranych zalet płynących ze stosowania wzorców projektowych należą:

- wprowadzony przez wielu kompetentnych i doświadczonych praktyków sposób rozwiązania danego problemu, świadczący o autorytecie stojącym za tą ideą,
- potwierdzony przez społeczność programistów sposób na jak najlepsze podejście do określonego problemu implikujący pewność, że w przypadkach wielu możliwych zastosowań wzorców korzyści płynące z ich stosowania zostały potwierdzone,
- modułowy sposób konstrukcji oprogramowania umożliwiający wielokrotne użycie tego samego kodu dzięki fragmentaryzacji jego struktury i dekompozycji funkcjonalności kodu na fragmenty struktury, zwiększający jego dostępność i użyteczność,
- ułatwiona komunikacja w zespole deweloperskim ze względu na ten sam sposób działania wzorca, jego integrację z zastaną implementacją, szablonowość budowy wzorca. Łatwiejszą komunikację umożliwiają te same punkty odniesienia w kodzie. Wspólna jest zasada użycia wzorca i jego konstrukcja,
- łatwiejsze utrzymanie kodu programu, będące konsekwencją schludnych zasad konstrukcji wzorca wg. przyjętego szablonu. Widoczność wzorca wpleczonego w projekt ułatwia zrozumienie kodu i jego utrzymanie,
- szerszy potencjał na rozbudowę projektu i rozszerzenie funkcjonalności programu. Z powodu wdrażania najlepszych rozwiązań spośród wielu możliwości, czyli stosując wzorce projektowe zapewnia się, że taki kod będzie łatwiej rozbudować, ponieważ konstrukcja oprogramowania, które jest rozszerzalne stanowi jedną z podstawowych zasad tworzenia dobrej jakości kodu. Mówi o tym zasada *otwarte-zamknięte* ze zbioru zasad „*SOLID*” [13]. Zasady „*SOLID*” są powiązane z zasadami konstrukcji wzorców projektowych,
- mniejsze ryzyko wytworzenia wadliwego oprogramowania. Podczas rozwoju implementacji z pominięciem sprawdzonych wzorców, szczególnie w kontekstach, w których użycie wzorca jest zasadne, działanie na własną rękę podczas tworzenia kodu, może spowodować późniejsze

problemy z poprawnością działania tworzonych funkcjonalności. Za przyczynami powstawania błędów stoją próby rozbudowy albo modyfikacji tak tworzonych implementacji.

3.1.2 Wady

Do wad stosowania wzorców projektowych należą:

- zbyt skomplikowanie prostego kodu. Nie w każdym przypadku należy stosować wzorce projektowe. Nie warto stosować ich w przypadku łatwych i oczywistych rozwiązań,
- trudność w zrozumieniu zasady działania wzorców szczególnie wśród nowych adeptów programowania, ze względu na dużą złożoność wzorców,
- usztywnienie architektury realizowanego projektu, poprzez narzucenie struktur i konstrukcji, które są trudne do modyfikacji lub rozbudowy i zmniejszają elastyczność kodu podczas prób jego rozwoju. Dotyczy to głównie prostszych projektów, w których możliwe są do zastosowania odmienne od wzorców metody w celu realizacji wymaganych założeń,
- zwiększona ilość kodu, który należy napisać w celu implementacji wzorców, może spowodować trudności w jego zrozumieniu albo dalszej rozbudowie utrudniając w ten sposób zarządzanie kodem,
- podejście do programowania, w którym stosowanie wzorców jest w sposób merytorycznie nieuzasadniony nadużywane, powoduje zbyt skomplikowanie kodu w kontekstach, w którym można zastosować prostsze rozwiązania, nie oparte o wzorce projektowe,
- w niektórych przypadkach rozwoju albo modyfikacji kodu, w którym zastosowano wzorce może nastąpić zmniejszenie elastyczności tworzonej implementacji. Z racji swojej natury wzorce, które opierają się na stosunkowo sztywnych szablonach narzucających sposób tworzenia implementacji programiście mogą zmniejszyć potencjał odnośnie modyfikacji i rozwoju analizowanego kodu. W takim przypadku należy rozważyć jednocześnie i porównać korzyści i problemy, które mogą być konsekwencją wdrożenia wzorca.

3.1.3 Przyjęte rozwiązanie

Ze względu na fakt, że wzorce projektowe stosowane są w niektórych przypadkach podczas tworzenia oprogramowania oraz pozostają dobrowolnym wskazaniem dla dewelopera, w przyjętym rozwiązaniu postanowiono, że wybór zastosowania wzorca zależy od gestii programisty.

Ze względu na wysoką złożoność zagadnienia, jakim jest rozpoznanie kontekstu użycia wzorca i decyzja czy użyć wzorca czy nie, analiza tego zagadnienia musi pozostać po stronie programisty.

Decyzja o użyciu wzorca w danym momencie tworzenia implementacji, wymusza konstrukcję szablonu wzorca i jego integrację z już istniejącym kodem. Zadanie to w prezentowanej pracy postanowiono ułatwić, wyręczając programistę z etapu tworzenia szablonu wzorca. Dodatkową

korzyścią jest fakt, że programista nie jest zmuszony do łączenia procesu budowy szablonu i procesu jego integracji z kodem w jednym kroku, często bezpośrednio na już tworzonej implementacji. Złączenie tych procesów w jeden może utrudnić programowanie, ze względu na duże skomplikowanie zagadnienia, powodując dłuższy czas wdrożenia wzorca i potencjalnie powstałe błędy. Proponowane rozwiązanie automatyzuje proces konstrukcji szablonu wzorca, generując wymagany kod, wyręczając programistę i rozbijając etap wdrażania wzorca na etap tworzenia szablonu i etap jego integracji, przy zachowaniu korzyści płynących z automatyzacji procesu, który:

- przyspiesza wdrożenie wzorca,
- zmniejsza ryzyko popełnienia błędu,
- ułatwia zrozumienie kontekstu zastosowania wzorca,
- zmniejsza złożoność zagadnienia, atomizując proces implementacji wzorca na mniejsze etapy.

Sam etap integracji szablonu z istniejącym kodem stanowi zadanie, które stoi przed programistą ze względu na bardzo dużą złożoność i trudność w automatyzacji całego procesu. Automatyczne generowanie szablonu jest zadaniem, które umożliwia wstępną konfigurację generowanej zawartości w celu łatwiejszej późniejszej integracji z istniejącym kodem. Takie podejście możliwe jest poprzez generowanie indywidualnych szablonów danego wzorca w zależności od wymagań istniejącego kontekstu zastosowania w bieżącej implementacji. Tworzenie zindywidualizowanego szablonu następuje poprzez ustawienie wstępnych parametrów, czyli wskazań w jaki sposób dany wzorzec ma zostać wygenerowany. Niniejsze rozwiązanie nie wyręcza programisty w całym etapie integracji wzorca w realizowanym projekcie, jednak znacznie ten proces ułatwia, dzięki zastosowaniu indywidualnych szablonów. Dodatkowo usprawnia proces lepszego zrozumienia zasad działania wzorca w odniesieniu do konkretnego, dedykowanego projektu zamiast tylko ogólnego rozeznania zasad jego działania pod kątem szerokiego zakresu wielu rodzajów projektów.

3.2 Wzorce behawioralne

Wzorce behawioralne [14] albo operacyjne skupiając się na modelu komunikacji pomiędzy obiektami. Istnieje wiele specjalistycznych odmian tych wzorców w zależności od modelu komunikacji.

Wzorce behawioralne dotyczą także sposobu konstrukcji algorytmów w taki sposób, aby możliwa była implementacja komunikacji pomiędzy klasami i obiektami. Wymiana informacji pomiędzy nimi umożliwia także podział zadań pomiędzy te obiekty. W rzeczywistości nie trzeba skupiać się na analizie i śledzeniu złożonych przepływów sterowania tylko na powiązaniach występującymi pomiędzy obiektami komunikującymi się między sobą i pełniącymi przez te obiekty zadaniami.

Możliwa staje się kontrola nad stanem poszczególnych obiektów, zapewniając niezmiennosc stanu z możliwością komunikowania się poprzez nadawanie lub odbieranie informacji. Dodatkowo możliwe jest sterowanie aktywnością obiektów poprzez ich aktywację lub dezaktywację.

3.2.1 Charakterystyka

Wzorzec *Łańcuch Zobowiązań* tworzy łańcuch lub szereg powiązanych ze sobą obiektów, gdy informacja jest przekazywana wzdłuż łańcucha każde z ogniw decyduje o swoim zachowaniu w

odpowiedzi na komunikat oraz czy przekazać informację dalej wzdłuż łańcucha. Wzorzec ma zastosowanie np. w funkcjonalnościach wymagających zastosowania tej samej logiki dla szeregu różnych obiektów.

Wzorzec *Polecenie* hermetyzuje wywołanie metod w osobny obiekt, który odpowiada za wywołanie tak hermetyzowanych metod. Dzięki temu możliwe jest:

- wywoływanie metod z opóźnieniem, w wymaganym czasie,
- cofanie efektów działania metod nawet po ich wywołaniu,
- wywoływanie szeregu różnego rodzaju metod, również z możliwościami wycofywania zmian,
- zmiana logiki metod poprzez ich parametryzowanie, także dynamicznie w trakcie działania programu.

Zastosowanie wzorca często ma miejsce jako warstwa pośrednia pomiędzy graficznym interfejsem użytkownika, a logiką biznesową aplikacji, aby uniknąć każdorazowej aktualizacji stanu logiki biznesowej na każdą interakcję użytkownika z *GUI*. Realizacja żądań z *GUI* jest separowana i agregowana w zbiory żądań, aby we właściwym momencie wywołać je na rzecz wewnętrznej warstwy logiki biznesowej.

Wzorzec *Iterator* umożliwia przechodzenie po elementach zbioru (obiektu) zapewniając mu interfejs charakterystyczny dla obiektów iterowanych, jednocześnie nie naruszając wewnętrznej struktury obiektu i nie wymuszając jego konwersji na typ klas iterowanych.

Wzorzec *Mediator* ogranicza powiązania komunikacyjne pomiędzy obiektami. W środowiskach o złożonych zależnościach między obiektami wprowadzenie obiektu mediatora, który sam przejmuje i zarządza komunikacją między obiektami umożliwia separację obiektów od siebie nawzajem i zmniejsza złożoność zależności komunikacyjnych między nimi. Łatwiejsze staje się utworzenie nowego typu komunikacji wraz z jego integracją z już istniejącym systemem.

Wzorzec *Pamiętka* umożliwia zapamiętanie stanu obiektu jako migawki, następnie możliwe są dalsze zmiany stanu i w pewnym momencie w przyszłości przywrócenie stanu obiektu z migawki na zasadzie powrotu do przeszłości. Stany obiektu zapamiętywane są w samym obiekcie, którego dotyczą, a nie przez zewnętrzny obiekt. Takie rozwiązanie upraszcza model wzorca, dzięki czemu możliwe są operacje przywracania stanów z przeszłości w działających programach np. poprzez opcję *cofnij* w procesorach tekstu.

Wzorzec *Strategia* umożliwia wymienne stosowanie różnych algorytmów przez obiekt w zależności od żądania jaki rodzaj algorytmu ma zostać użyty w zadanym kontekście. Algorytmy są hermetyzowane, dzięki czemu możliwa jest ich modyfikacja lub dodawanie nowych algorytmów bez ingerencji w obiekt je wykonujący.

Wzorzec *Stan* podobnie jak wzorzec *Strategia* hermetyzuje logikę algorytmów, aby możliwa była ich wymiennność. Podstawową różnicę stanowi powiązanie ich wymiany ze zmianą stanu obiektu. W przypadku zmiany stanu obiektu na wewnętrznie określony następuje dopasowanie zachowania obiektu do aktualnego stanu. Obiekt może zachowywać się tak jakby reprezentował różne klasy w

zależności od swojego stanu. Wzorzec często stosuje się w celu redukcji wielu instrukcji *if-else* w tworzonym kodzie.

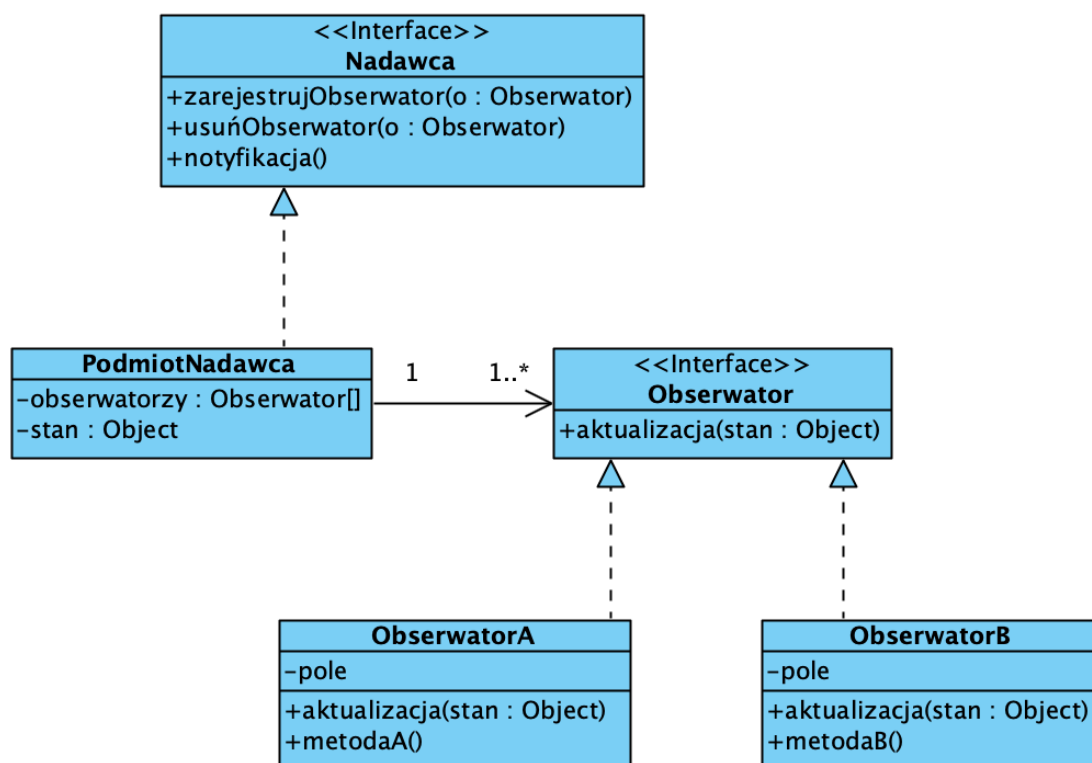
Wzorzec *Metoda Szablonowa* ma zastosowanie w przypadku, gdy istnieje wiele wariantów podobnych do siebie implementacji klas, których zadeklarowanie i użycie następuje w osobnej i pojedynczej klasie. W tej *klasie-rodzicu* względem pozostałych *klas-dzieci* tworzony jest szablon kodu uwspólniający i uogólniający podobne algorytmy z *klas-dzieci*. Logika z *klas-dzieci* jest podstawiana w szablonie, w zależności od wymagań funkcjonalnych określających która *klasa-dziecko* ma zostać w danym momencie podstawiona. Dzięki Metodzie szablonowej możliwe jest dodawanie nowych lub modyfikacja istniejących algorytmów w pojedynczej klasie celem rozszerzenia jej funkcjonalności bez zmiany jej struktury. Podstawiana jest wymagana i wybrana w danym momencie implementacja z grupy do szablonu w celu jej wywołania.

Wzorzec *Odwiedzający* potrafi przejść przez zbiór podobnych obiektów i zażądać wywołania tej samej metody na rzecz każdego z nich. Logika nie jest umieszczana wewnątrz odwiedzanych obiektów tylko w *Odwiedzającym*. Logika metody będzie różnić się w zależności od konkretnego, pojedynczego obiektu i będzie rozpoznawana poprzez przekazanie samego obiektu do obiektu *Odwiedzającego*. Żaden obiekt poza *Odwiedzającym* nie będzie zawierał implementacji tej logiki, jedynie wspólną, pustą metodę. Klasa *Odwiedzającego* potrafi wywołać tą metodę z właściwą logiką dla danego obiektu rozpoznając ją tylko na podstawie samego przekazania obiektu, bez instrukcji warunkowych *if-else*. Wzorzec umożliwia rozszerzenie funkcjonalności podobnych grup obiektów, na zewnątrz tych obiektów, bez implementacji logiki wewnątrz samych obiektów.

3.2.2 Obserwator

Wzorzec projektowy *Obserwator* umożliwia propagację informacji lub zmianę stanu jednego obiektu do zbioru określonych obiektów, oczekujących tej zmiany. Pojedynczy obiekt nadający określany jako podmiot lub nadawca posiada listę wielu obiektów odbiorców. Po nadaniu przez podmiot informacji otrzymają ją wszystkie obiekty odbiorcze. Wzorzec działa w relacji *jeden-do-wielu* między obiektem nadawcą, a odbiorcami. Obiekty, na rzecz których istnieje potrzeba otrzymywania informacji od nadawcy mogą zostać dodane do listy subskrybentów i otrzymywać publikowane powiadomienia. Gdy obiekty subskrybujące otrzymają powiadomienie mogą przetworzyć otrzymaną informację lub zmienić swój stan. Istnieje możliwość wycofania subskrypcji przez obiekt, wówczas notyfikacje nie będą do takiego obiektu wysyłane, natomiast będą wysyłane do pozostałych subskrybentów.

Wzorzec *Obserwator* ma zastosowanie w przypadkach, gdy istnieją obiekty, które muszą się ze sobą komunikować i są zależne od siebie. Gdy jeden obiekt nadaje informację pozostałe obiekty po jej otrzymaniu zmieniają swój stan. Szczególnie dotyczy to sytuacji, gdy informacja jest nadawana dynamicznie, nieregularnie lub bez określonego terminu. Dotyczy to także sytuacji, w których obiekty oczekujące muszą otrzymać informację, aby podjąć dalsze przetwarzanie danych w systemie lub rozpocząć nowe działanie.



Rysunek 7. Diagram klas UML wzorca projektowego Obserwator. Źródło: opracowanie własne.

Na diagramie UML zaprezentowanym na Rysunku 7, odnoszącym się do wzorca projektowego *Obserwator* Klasa *PodmiotNadawca* przechowuje referencję do obiektu typu *Obserwator*. Interfejsy *Nadawca* i *Obserwator* stanowią kontrakty określające w jaki sposób obiekty je implementujące mają się ze sobą komunikować. Klasa *PodmiotNadawca* przechowuje listę zmiennych typu *Obserwator* reprezentującą zarejestrowane obserwatory. Gdy zmienna stan reprezentująca na diagramie ogólny typ danych *Object* ulegnie zmianie, nastąpi wywołanie metody *notyfikacja*, która w swoim ciele wywoła metodę *aktualizacja(stan)* na rzecz każdej przechowywanej zmiennej typu *Obserwator*. Wywołanie metody *aktualizacja(stan)* w obiekcie typu *PodmiotNazwa* spowoduje, że zmienna stan zostanie odebrana przez obiekty *ObserwatorA* i *ObserwatorB*. Obiekty obserwatorów mogą od tego momentu rozpocząć dalsze przetwarzanie stanu zgodnie ze swoją logiką.

Listing 2 przedstawia przykładową implementację wzorca projektowego *Obserwator*. *Listing* prezentuje interfejs *Nadawcy* i *Obserwatora* oraz klasy je implementujące. Na implementację składa się klasa *Nadawcy* i dwie klasy *Odbiorców* tego samego komunikatu. Komunikat wysyła klasa *Nadawcy*. Rozwiązanie symuluje prosty wycinek działania stacji meteorologicznej, która wysyła komunikaty o temperaturze do dwóch podległych jej stacji odbiorczych

Listing 2. Przykładowa implementacja wzorca projektowego Obserwator w języku Java.

```
// Obiekt Obserwowany
interface Nadawca {
    void zarejestrujObserwator(Obserwator obserwator);
    void usuńObserwator(Obserwator obserwator);
    void notyfikacja();
}

// Obserwator
interface Obserwator {
    void aktualizacja(Object stan);
}

// Nadawca
import java.util.ArrayList;
import java.util.List;

class Pogoda implements Nadawca {
    private List<Obserwator> obserwatorzy;
    private Object stan;
    public Pogoda() {
        obserwatorzy = new ArrayList<>();
    }

    @Override
    public void zarejestrujObserwator (Obserwator obserwator) {
        obserwatorzy.add(observer);
    }

    @Override
    public void usuńObserwator (Obserwator obserwator) {
        obserwatorzy.remove(observer);
    }

    @Override
    public void notyfikacja () {
        for (Obserwator obserwator: obserwatorzy) {
            obserwator.aktualizacja(stan);
        }
    }

    // Metoda symulująca zmianę stanu pogody
    public void zmieńStan(Object stan) {
        this.stan = stan;
        notyfikacja();
    }
}

// Nadawca A
class StacjaMeteoA implements Obserwator {
    private Object stan;

    @Override
    public void aktualizacja(Object stan) {
        this.stan = stan;
        wyświetl();
    }

    public void wyświetl () {
```

```

        System.out.println("Aktualna pogoda w A: " + stan);
    }
}
// Nadawca B
class StacjaMeteoB implements Obserwator {
    private Object stan;

    @Override
    public void aktualizacja(Object stan) {
        this.stan = stan;
        wyswietl();
    }

    public void wyswietl () {
        System.out.println("Aktualna pogoda w B: " + stan);
    }
}
// Klient
public class ObserwatorDemo {
    public static void main(String[] args) {
        Pogoda pogoda = new Pogoda();

        StacjaMeteoA stacjaMeteoA = new StacjaMeteoA();
        StacjaMeteoB stacjaMeteoB = new StacjaMeteoB();

        pogoda.zarejestrujObserwator(stacjaMeteoA);
        pogoda.zarejestrujObserwator(stacjaMeteoB);

        pogoda.zmieńStan("26 C");
    }
}
//output:
//aktualna pogoda w A: 26 C
//aktualna pogoda w B: 26 C

```

3.3 Wzorce strukturalne

Wzorce strukturalne [15] odpowiadają za budowanie złożonych konstrukcji, struktur z bardziej podstawowych elementów takich jak klasy i obiekty. Struktury te pełnią jednolitą funkcjonalność przyczyniając się do większej elastyczności i efektywności niż pojedyncze klasy i obiekty. Ich powiązanie w strukturalne zależności umożliwia realizowanie funkcjonalności niedostępnych dla pojedynczych klas i obiektów.

Wzorce strukturalne klasowe mają u podstaw dziedziczenie interfejsów i klas w celu składania bardziej złożonych struktur, natomiast wzorce obiektowe w tym celu składają obiekty.

3.3.1 Charakterystyka

Wzorzec *Most* umożliwia rozbięcie obszernej monolitycznej klasy na zestaw kilku mniejszych klas, których implementacje będą zawierały poszczególne funkcjonalności z monolitu. Za pomocą mechanizmu kompozycji obiektów można połączyć zgrupowane wg pokrewnej funkcjonalności klasy tworząc most pomiędzy nimi. Most umożliwia zachowanie pełnej funkcjonalności z monolitu na rzecz krótszej implementacji i mniejszej liczby przypadków lub odmian funkcjonalnych, które normalnie

wymagały implementacji dla każdej odmiany. Ta skłonność uwidacznia się szczególnie podczas próby dodania nowej funkcjonalności do monolitu. Monolit wymagałby implementacji nowej cechy dla wszystkich przypadków, natomiast *Most* umożliwia po prostu dodanie tej funkcjonalności, podczas gdy mechanizm kompozycji zapewnia grupie klas z nową funkcjonalnością zachowanie odmiany dla wszystkich możliwych przypadków. Możliwe jest niezależne dodawanie nowych klas z nową funkcjonalnością dla każdej ze zgrupowanych klas po obu stronach Mostu.

Wzorzec *Kompozyt* możliwy jest do zastosowania tylko dla struktur klas o układzie hierarchicznym, z których można zbudować drzewo. Węzły i liście w drzewie stanowią klasy, które są ze sobą powiązane tak jak były w hierarchii. Cechą takiej struktury jest możliwość rekurencyjnego wywoływania tej samej metody dla wszystkich elementów, możliwe jest więc pominięcie złożonej struktury obiektów składających się na drzewo i skupienie tylko na wywołanej metodzie, oddziałującej na wszystkie te obiekty. Kompozyt można zastosować np. w budowie interfejsu użytkownika albo w mapowaniu systemu plików.

Wzorzec *Fasada* tworzy nowy, uproszczony interfejs, który opakowuje złożony funkcjonalnie system. Fasada odpowiada za komunikację pomiędzy klientem, a takim rozbudowanym systemem zapewniając klientowi prostą obsługę pomimo jego skomplikowania. Wzorzec może mieć wiele różnych, nieskomplikowanych interfejsów, dzięki którym klient może korzystać ze złożonego systemu w sposób uproszczony i bez zawiłych powiązań kodu klienta z systemem, czyli bez zmniejszania elastyczności kodu.

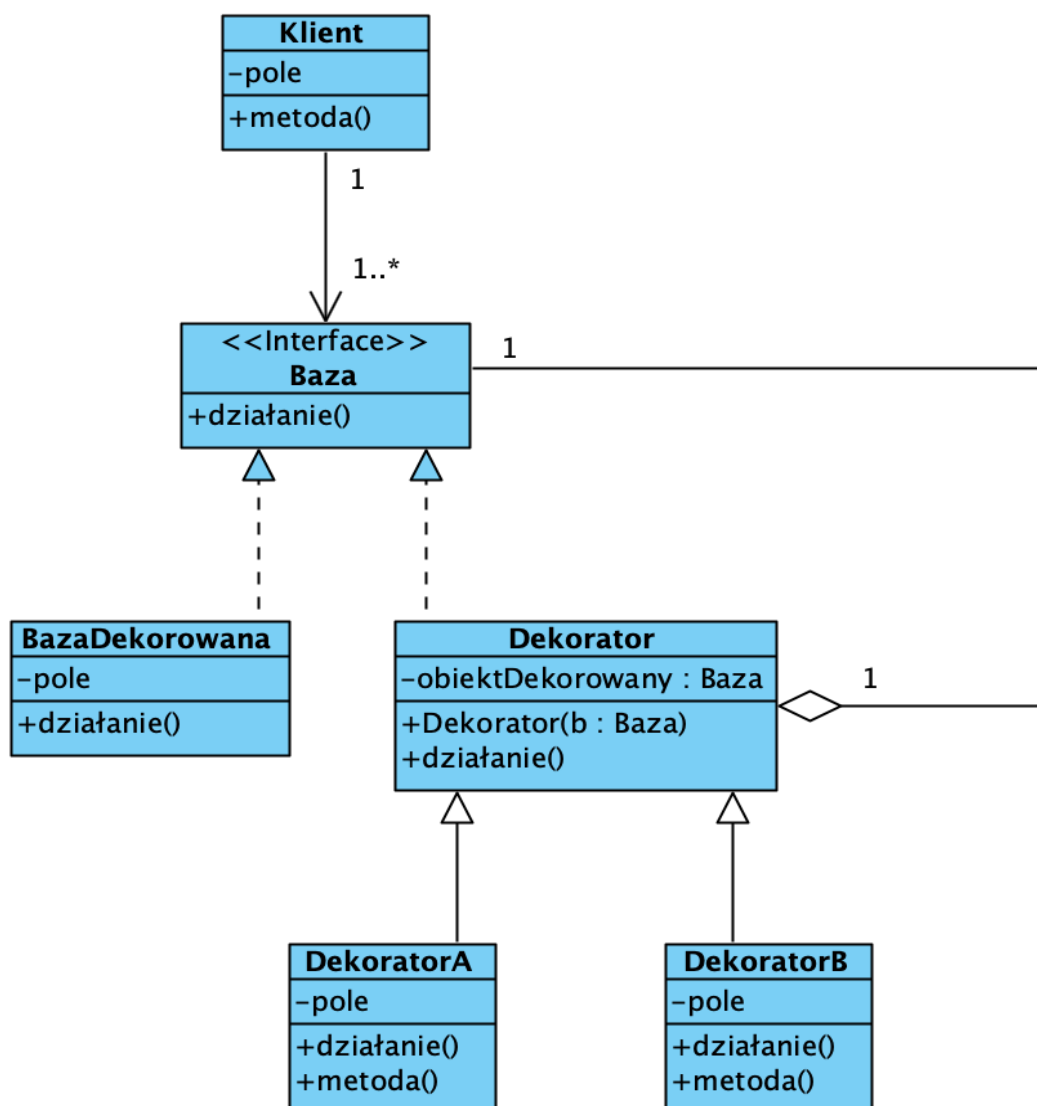
Wzorzec *Pyłek* ma zastosowanie wszędzie tam, gdzie tworzona implementacja zawiera dużą liczbę wymagających pod względem zasobów obiektów, które mają dodatkowo te same cechy. W takim przypadku tworzy się jeden obiekt charakteryzujący wiele tych obiektów oraz tablicę, która zawiera referencję do nowego obiektu oraz informację o lokalizacjach, w których ten obiekt ma zostać użyty. Nie jest wymagane tworzenie nowych obiektów tego samego typu w każdej lokalizacji, zamiast tego korzysta się z obiektów wirtualnych, czyli referencji do jednego obiektu, na którego wskazuje tablica wraz z jego lokalizacją. Z obiektu, który ma cechy niezmiennicze i te ulegające zmianom tworzy się obiekt, który zawiera tylko cechy niezmiennicze, szczególnie jeśli są to cechy wymagające dużo zasobów. Taki odchudzony obiekt nazywa się pyłkiem. Referencję do *pyłka* umieszcza się w tablicy lub klasie wraz z cechami, które ulegają zmianom, które wcześniej usunięto z obiektu *pyłka*, a które teraz reprezentują *pyłek* w podanej referencji i z podanymi cechami.

Wzorzec *Pośrednik* lub *Proxy* umożliwia stworzenie obiektu, który będzie reprezentantem innych obiektów, inicjalizacja których byłaby kosztowna pod względem zasobów czasu, wydajności lub wymagałaby stosownych uprawnień dostępu. Pośrednik może dotyczyć m. in. obiektów zdalnych i wirtualnych. Interakcja z rzeczywistymi obiektami odbywa się za pomocą pośrednika.

3.3.2 Dekorator

Wzorzec projektowy *Dekorator* stanowi alternatywę dla mechanizmu dziedziczenia klas. Zachowuje korzyści płynące z dziedziczenia i eliminuje jego wady. Dziedziczenie ma charakter statyczny i nie udostępnia możliwości ingerencji w podklasę w trakcie działania programu. Dodatkowo w wielu językach programowania w tym w Języku *Java* nie występuje dziedziczenie wielokrotne. Możliwe jest dziedziczenie tylko z jednej nadklasy. Wzorzec *Obserwator* zapewnia elastyczny

mechanizm rozszerzenia funkcjonalności klasy, modyfikacji jej działania lub zmiany stanu poprzez istnienie zależności pomiędzy obiektem dekorowanym, a dekoratorem. Obiekt dekorowany może być rozszerzony przez pojedynczy dekorator, po czym może wystąpić drugie rozszerzenie za pomocą drugiego dekoratora itd. Liczba dekoratorów nie jest w góry ograniczona. Wizualnie możliwe jest przedstawienie dekoratora jako obudowę dla klasy dekorowanej. Każdy kolejny dodany dekorator obudowuje już istniejącą klasę dekorowaną i otaczającą ją *dekoratory*. Każdy dekorator może zmienić funkcjonalność implementacji, którą obudowuje przed przekazaniem sterowania do obudowywanej warstwy lub po przekazaniu. Kolejne modyfikacje stanu w układzie wielu dekoratorów następują warstwa po warstwie, od tej całkowicie zewnętrznej, po kolejne kierując się ku środkowi, który stanowi obiekt klasy, która jest dekorowana. Kolejno wprowadzane zmiany w układzie klas *Dekoratora* przebiegają w sposób rekurencyjny.



Rysunek 8. Diagram klas UML wzorca projektowego Dekorator. Źródło: opracowanie własne.

Możliwość zastosowania wielu różnych dekoratorów symuluje mechanizm dziedziczenia wielokrotnego, ponieważ każdy dekorator dostarcza własne rozszerzenie funkcjonalności dla klasy bazowej. Zwiększona elastyczność tak tworzonej implementacji jest zapewniona przez możliwość wymiany dekoratorów w trakcie działania programu. Kolejną korzyść stanowi zachowanie zgodności typów dekoratorów z typem klasy dekorowanej, więc z punktu widzenia klienta interfejs do komunikacji z klasą bazową pozostaje zachowany, niezależnie od liczby i rodzaju dekoratorów. Klient nie musi zmieniać swojej implementacji podobnie jak klasa dekorowana.

Na diagramie *UML* zaprezentowanym na *Rysunku 8*, odnoszącym się do wzorca projektowego *Dekorator* klasa *Klient* używa referencji do obiektu typu *Baza*, który wykorzystuje w celu interakcji ze wzorcem. Klasa *Dekorator* agreguje obiekt typu *Baza*. Interfejs *Baza* jest wspólny dla:

- Klienta,
- Klasy dekorowanej,
- Dekoratorów.

Dzięki wspólnemu interfejsowi *Baza* możliwa jest interakcja klienta z całym systemem dekoratorów za pomocą tych samych metod, które posiada klasa dekorowana. Klasa *BazaDekorowana* stanowi klasę dekorowaną. Posiada ona własną logikę, która będzie modyfikowana przez klasy dekoratorów. Klasy dekoratorów: A i B dziedziczą po klasie *Dekorator* i nadpisują metodę *działanie()* w celu dopasowania funkcjonalności klasy *BazaDekorowana* do własnej logiki. Klasa *Dekorator* posiada zmienną instancyjną odwołującą się do obiektu dekorowanego, tą zmienną dziedziczą dekoratory A i B. Dekoratory A i B mogą:

- rozbudowywać stan obiektu dekorowanego,
- dodawać nowe metody,
- rozszerzać istniejące metody z interfejsu *Baza*.

Listing 3. Przykładowa implementacja wzorca projektowego Dekorator w języku Java.

```
// Baza
interface Kawiarnia {
    String opis();
}

// BazaDekorowana
class Espresso implements Kawiarnia {
    @Override
    public String opis() {
        return "Espresso";
    }
}

// Dekorator
abstract class Dekorator implements Kawiarnia {
    protected Kawiarnia kawiarnia;

    public Dekorator(Kawiarnia kawiarnia) {
        this.kawiarnia = kawiarnia;
    }

    @Override
```



```

        public String opis() {
            return kawiarnia.opis();
        }
    }
    // DekoratorA
    class Cukier extends Dekorator {
        public Cukier(Kawiarnia kawiarnia) {
            super(kawiarnia);
        }

        @Override
        public String opis() {
            return kawiarnia.opis() + ", Cukier";
        }
    }
    // Klient
    public class DekoratorDemo {
        public static void main(String[] args) {
            // Zamówienie espresso bez dodatków
            Kawiarnia kawiarnia = new Espresso();
            System.out.println(kawiarnia.opis()) //Espresso

            // Zamówienie espresso z cukrem
            kawiarnia = new Cukier(kawiarnia);
            System.out.println(kawiarnia.opis()) // Espresso, Cukier
        }
    }
}

```

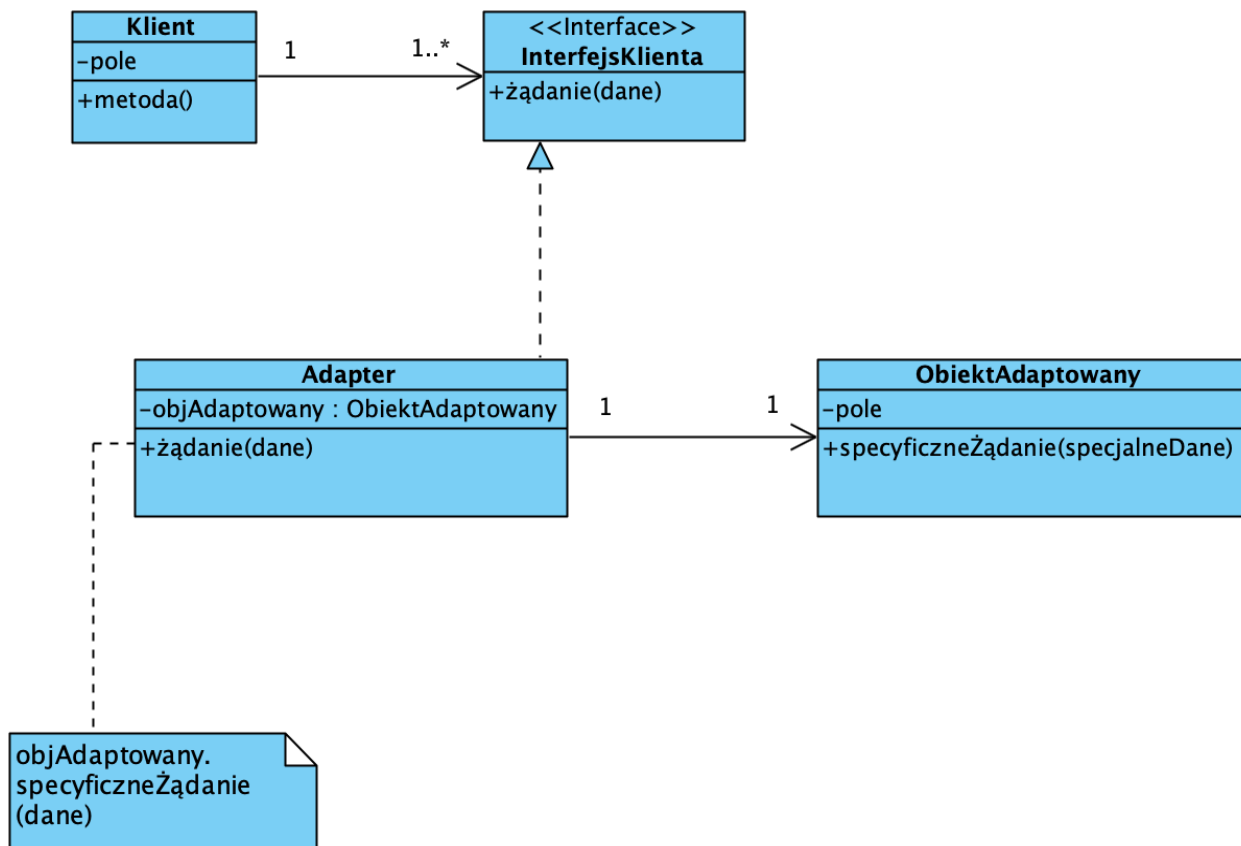
Na *Listingu 3* przedstawiono przykładową implementację wzorca projektowego *Dekorator*. Pokazaną strukturę projektu tworzy interfejs dla klasy bazowej, klasa bazowa, klasa abstrakcyjna dekoratora i klasa dekoratora. Klasa bazowa stanowi klasę dekorowaną. Klasa dekoratora dziedziczy po klasie abstrakcyjnej. Klasa bazowa implementuje interfejs. Pokazana logika odnosi się do prostej symulacji składania zamówienia w kawiarni. Klasa bazowa odpowiada za rodzaj podawanej kawy, a klasa dekoratora za dodatek w formie cukru do kawy. Wynik działania programu pokazuje korelację działania logiki klasy dekorowanej i dekoratora.

3.3.3 Adapter

Wzorec projektowy *Adapter* umożliwia dopasowanie do siebie dwóch obiektów w taki sposób, aby możliwa była interakcja pomiędzy nimi. Przyczyną braku interakcji pomiędzy obiektami jest niekompatybilny interfejs w procesie komunikacji. *Adapter* umożliwia komunikację klasy klienta, która korzysta z adaptera i klasy obiektu adaptowanego, który ma interfejs niemożliwy do użycia w procesie komunikacji przez klienta. Zastosowanie wzorca skutkuje wystawieniem interfejsu, który po stronie klienta jest możliwy do użycia w procesie interakcji. Po stronie obiektu adaptowanego następuje przysłonięcie bazowego interfejsu przez interfejs adaptera w taki sposób, że adapter za pomocą swojego interfejsu przenosi jego wywołanie na interfejs obiektu adaptowanego. Adapter zawiera logikę, która dopasowuje wywołanie jego interfejsu przez klienta do interfejsu obiektu adaptowanego. Dzięki takiemu podejściu możliwa staje się komunikacja pomiędzy kodem klienta, a obiektem adaptowanym za pośrednictwem adaptera. *Adapter* współdziała zarówno w kodem klienta jak i obiektu adaptowanego.

Adapter ma zastosowanie w przypadku konieczności użycia jakiejś klasy, której interfejs jest niekompatybilny z istniejącą implementacją np. zewnętrznej biblioteki. Użycie adaptera wyklucza

konieczność dopasowania kodu klienta albo często niemożliwą do realizacji konieczność zmiany interfejsu biblioteki.



Rysunek 9. Diagram klas UML wzorca projektowego Adapter. Źródło: opracowanie własne.

Na diagramie *UML* zaprezentowanym na *Rysunku 9*, odnoszącym się do wzorca projektowego *Adapter* Klasa *Klient* przechowuje referencję do obiektu typu *InterfejsKlienta*, natomiast klasa *Adapter* przechowuje referencję do obiektu typu *ObiektAdaptowany*. Klasa *Klient* wywołuje metodę *żądanie(dane)* przekazując dane poprzez interfejs klienta. Interfejs implementuje klasa *Adapter*, która przyjmuje dane klienta i wewnątrz ukrytych mechanizmów swojej logiki konwertuje otrzymane dane na dane możliwe do propagacji poprzez metodę *specyficzneŻądanie(specjalneDane)*. Metoda *specyficzneŻądanie()* pochodzi z klasy *ObiektAdaptowany*, do którego referencja jest przechowywana w obiekcie typu *Adapter*.

Przedstawiony mechanizm umożliwia zastosowanie tego samego *Adaptora* dla dowolnej klasy podrzędnej *ObiektuAdaptowanego*. Dodatkowo dzięki temu, że klient korzysta z interfejsu możliwe staje się użycie wielu rodzajów adapterów dla różnych typów klas obiektów adaptowanych bez zmiany kontraktu odnośnie do sposobu użycia przez klienta wystawionego interfejsu. Kod klienta nie wymagałby modyfikacji po zmianie adaptera.

Listing 4. Przykładowa implementacja wzorca projektowego Adapter w języku Java.

```
// Interfejs oczekiwany przez klienta
interface Odtwarzacz {
    void graj(String nazwaPliku);
}
// Istniejący interfejs, który chcemy zaadaptować
interface OdtwarzaczHiFi {
    void grajHiFi(String nazwaPliku);
}
// Implementacja istniejącego interfejsu dla plików HiFi
class MarkowyOdtwarzacz implements OdtwarzaczHiFi {
    @Override
    public void grajHiFi(String nazwaPliku) {
        System.out.println("graj HiFi: " + fileName);
    }
}
// Adapter implementujący interfejs Odtwarzacz
class AdapterOdtwarzaczaHiFi implements Odtwarzacz {
    OdtwarzaczHiFi odtwarzaczHiFi = new MarkowyOdtwarzacz();

    @Override
    public void graj(String nazwaPliku) {
        // logika innowacyjnego konwertera audio z mp3 na FLAC
        odtwarzaczHiFi.grajHiFi(nazwaPliku)
    }
}
// Klient używający AdapterOdtwarzaczaHiFi do odtwarzania zwykłych utworów
// w jakości HiFi
class StudioKlienta implements Odtwarzacz {
    AdapterOdtwarzaczaHiFi adapterOdtwarzaczaHiFi;

    @Override
    public void graj(String nazwaPliku) {
        // granie formatu mp3 w HiFi
        adapterOdtwarzaczaHiFi = new AdapterOdtwarzaczaHiFi();
        adapterOdtwarzaczaHiFi.graj(nazwaPliku)
    }
}
// Użycie wzorca
public class AdapterDemo {
    public static void main(String[] args) {
        StudioKlienta studioKlienta = new StudioKlienta();

        studioKlienta.graj("utwór w mp3"); // graj HiFi: utwór w mp3
    }
}
```

Listing 4 prezentuje przykładową implementację wzorca projektowego Adapter. Na rozwiązanie składają się dwa interfejsy i trzy klasy. Klasa adaptera implementuje ten sam interfejs co klasa klienta, natomiast drugi interfejs implementuje klasa adaptowana, niedostępna dla klienta. Wynikiem działania przedstawionej konfiguracji jest możliwość wywołania logiki z klasy adaptowanej przez klienta za pomocą adaptera. Prezentowany przykład przedstawia prostą symulację konwersji utworu muzycznego z formatu *mp3* do formatu *HiFi*. Adaptację klasy grającej utwór *HiFi* dla klienta odtwarzającego tylko utwór *mp3* umożliwia logika konwersji zawarta w adapterze.

3.4 Wzorce kreacyjne

Wzorce kreacyjne [16] odpowiadają na potrzebę tworzenia szerokiej gamy obiektów w sposób zadany przez wymagania funkcjonalne.

Proces tworzenia obiektów i jego mechanizmy pozostają ukryte przed programistą. Do cech, które przyczyniają się do zwiększenia elastyczności stosowania wzorców kreacyjnych należą mechanizmy odpowiedzialne za:

- rodzaj tworzonych obiektów,
- system odpowiedzialny za tworzenie obiektów,
- sposób tworzenia obiektów,
- moment, w którym obiekty mają zostać utworzone.

Szczegóły opisanych cech pozostają ukryte wewnątrz struktur wzorców. Zastosowanie wzorców zapewnia dostarczenie wymaganego obiektu o określonej poprzez udostępniony, publiczny interfejs konfiguracji. Możliwe jest konfigurowanie potrzebnego obiektu w sposób:

- statyczny, zakodowany na sztywno podczas tworzenia implementacji,
- dynamiczny w trakcie działania programu.

3.4.1 Charakterystyka

Wzorzec Budowniczy:

- ukrywa przed klientem implementację związaną z tworzeniem obiektu,
- dzieli proces tworzenia obiektu na wiele etapów,
- rozciąga etapy tworzenia obiektu w czasie,
- zapewnia interfejs składający się z wielu metod, za pomocą których na każdym etapie tworzenia obiektu można dopasować konfigurację obiektu.

W efekcie otrzymuje się obiekt dopasowany do wymagań w danym kontekście, niezależnie od liczby etapów jego tworzenia, czasu trwania procesu, odmienności i zróżnicowania każdego ze stworzonych obiektów.

Istnieje podobieństwo między wzorcem *Budowniczy*, a wzorcem *Metoda Wytwórcza*. W przydadku wzorca *Fabryka (Metoda Wytwórcza)* etap tworzenia obiektu jest jednorazowy, a w przypadku wzorca *Budowniczy* proces jest wieloetapowy. Wzorzec *Budowniczy* wykorzystuje się także w celu eliminacji stosowania złożonych i rozbudowanych konstruktorów.

Wzorzec *Prototyp* umożliwia tworzenie kopii istniejących obiektów, zawierających kompletny ich stan. Na stan składają się zarówno zmienne publiczne jak i prywatne. Sama implementacja logiki klonowania jest zawarta w obiekcie, który ma zostać sklonowany. Taki obiekt nazywa się prototypem służącym za szablon do konstrukcji pochodnych obiektów. Wzorzec stosuje się, gdy istnieje potrzeba utworzenia kopii istniejących obiektów, które dodatkowo mają złożoną strukturę wewnętrzną.

Wzorzec *Singleton* jest stosowany, gdy:

- istnieje potrzeba konstrukcji obiektu, do którego możliwy jest dostęp z każdego kontekstu implementacji, czyli obiektu o dostępie globalnym,
- każdorazowe odwołanie się do obiektu z każdego kontekstu aplikacji nie spowoduje utworzenia nowego obiektu, tylko zwróci obiekt już istniejący. Stan takiego obiektu stanowią dane współdzielone przez program.

Wzorzec *Singleton* hermetyzuje dostęp do konstruktora obiektu i udostępnia statyczną metodę dostępową dla klientów. Metoda zwraca statyczną zmienną typu klasy *singletona*.

Singleton na zastosowanie tak gdzie wymagany jest dostęp do zasobów współdzielonych np.

- dostęp do globalnej konfiguracji aplikacji,
- menadżer połączenia z bazą danych,
- klient zarządzający logowaniem do aplikacji,
- zarządzanie połączeniem sieciowym.

Wadą zastosowania wzorca *Singleton* stanowi złamanie zasady pojedynczej odpowiedzialności.

Wzorzec *Fabryka Abstrakcyjna* jest wykorzystywana do tworzenia powiązanych ze sobą grup lub rodzin obiektów. Ma zastosowanie, gdy:

- system musi pozostać niezależny od sposobu w jaki tworzone będą obiekty,
- nowe rodziny obiektów powinny być dodawane za pomocą nowych rodzajów fabryk,
- tworzone obiekty są ze sobą powiązane albo mają współpracować razem ze sobą.

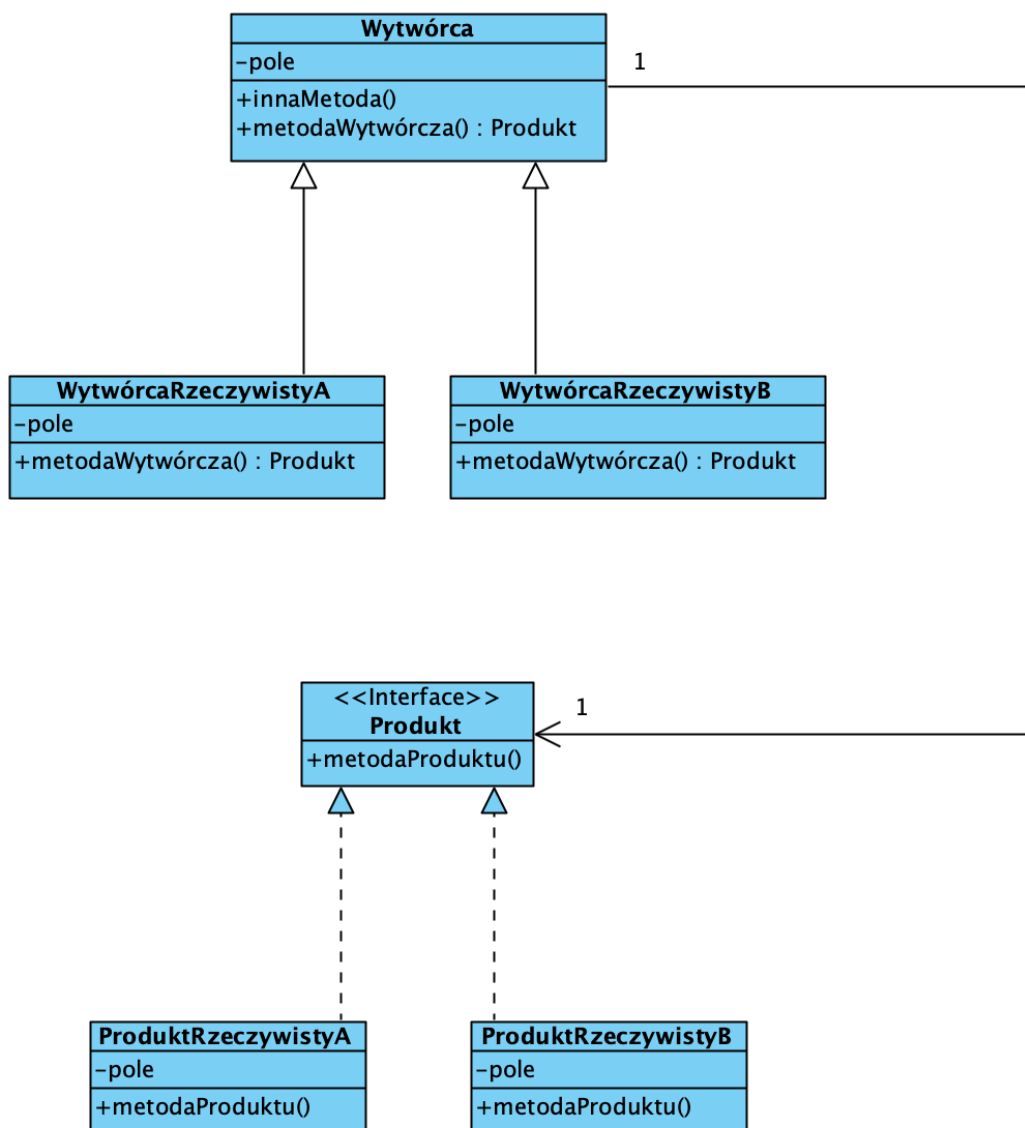
Wzorzec zapewnia interfejs do konstrukcji określonej grupy obiektów. Klient nie musi znać klas tworzonych obiektów. Za tworzenie konkretnych obiektów w ramach grupy może posłużyć sprzężenie *Fabryki Abstrakcyjnej* ze wzorcem *Metoda Wytwórcza*.

3.4.2 Metoda Wytwórcza

Wzorzec *Metoda Wytwórcza (Fabryka)* zapewnia interfejs do wywołania metody na rzecz obiektu, który w zależności od aktualnego kontekstu podlega wymianom. Z punktu widzenia klienta nie jest istotne jaki typ obiektu jest w danym momencie podstawiony. Istotne jest wywołanie bardziej abstrakcyjnej metody z nadklasy i efekt jej działania. Wzorzec ukrywa przed klientem mechanizm dopasowywania typu obiektu, na rzecz którego wywołana zostanie metoda klienta. Zastosowanie *Metody Wytwórczej* zapewnia elastyczny mechanizm rozszerzania funkcjonalności kodu o nowe typy obiektów, bez konieczności ryzykownych i kosztownych modyfikacji istniejącego kodu.

Wzorzec stosuje się w celu utworzenia różnych rodzajów obiektów, jednak o tym samym implementowanym interfejsie. Interfejs stanowi wzorzec dla zgrupowania rodziny obiektów w *Metodzie Wytwórczej*, czyli decyduje o jej rodzaju i typie tworzonych obiektów. Implementacja tworzonych obiektów spoczywa na podklasach. Łatwe staje się dodawanie nowych podklas w celu rozszerzenia zbioru tworzonych obiektów.

Może istnieć wiele rodzajów *Metod Wytwórczych*, szczególnie w sprzężeniu z wzorcem *Fabryka Abstrakcyjna*, jednak tworzone w konkretnej *Metodzie Wytwórczej* obiekty mają wspólny interfejs, więc stanowią wspólną grupę różnych obiektów o podobnych cechach.



Rysunek 10. Diagram klas UML wzorca projektowego *Metoda Wytwórcza*. Źródło: opracowanie własne.

Na diagramie UML zaprezentowanym na *Rysunku 10*, odnoszącym się do wzorca projektowego *Metoda Wytwórcza* klasa *Wytwórca* przechowuje referencję do obiektu typu *Produkt*. Interfejs *Produkt* grupuje *ProduktRzeczywistyA* oraz *ProduktRzeczywistyB* w jedną rodzinę klas. Klasa *Wytwórca* posiada zaimplementowane pole typu *Produkt* powodując istnienie zależności. *WytwórcaRzeczywistyA* nadpisuje metodę odpowiedzialną na tworzenie produktów z nadklasy i zwraca obiekt typu *ProduktRzeczywistyA*. Analogiczne zachowanie dotyczy klasy *WytwórcaRzeczywistyB* dla obiektu typu *ProduktRzeczywistyB*.

Listing 5. Przykładowa implementacja wzorca projektowego Metoda Wytwórcza w języku Java.

```
// Produkt
interface Zwierze {
    void dajGłos();
}

// Konkretne Produkty
class Pies implements Zwierze {
    @Override
    public void dajGłos() {
        System.out.println("pies");
    }
}

class Kot implements Zwierze {
    @Override
    public void dajGłos() {
        System.out.println("kot");
    }
}

// Wytwórca
abstract class FabrykaZwierzat {
    // Metoda wytwórcza
    public abstract Zwierze stwórzZwierze();
}

// Konkretni wytwórcy
class FabrykaPsów extends FabrykaZwierzat {
    @Override
    public Zwierze stwórzZwierze() {
        return new Pies();
    }
}

class FabrykaKotów extends FabrykaZwierzat {
    @Override
    public Zwierze stwórzZwierze() {
        return new Kot();
    }
}

// Użycie wzorca
public class MetodaWytwórczaDemo {
    public static void main(String[] args) {
        FabrykaZwierzat fabrykaPsów = new FabrykaPsów();
        Zwierze pies = fabrykaPsów.stwórzZwierze();
        pies.dajGłos(); // pies

        FabrykaZwierzat fabrykaKotów = new FabrykaKotów();
        Zwierze kot = fabrykaKotów.stwórzZwierze();
        kot.dajGłos(); // kot
    }
}
```

Listing 5 przedstawia przykładową implementację wzorca projektowego *Metoda Wytwórcza*. Na rozwiązanie składa się interfejs produktu i dwie klasy go implementujące oraz klasa abstrakcyjna fabryki i dwie klasy ją dziedziczące. W przykładzie jedna klasa implementująca jest funkcjonalnie

związana z jedną klasą dziedziczącą tworząc grupę. Prezentowana logika tworzy dwie grupy funkcjonalne. Jedną grupę stanowi fabryka tworząca obiekty zwierząt - kotów, a druga obiekty zwierząt - psów. Prezentowana funkcjonalność przedstawia mechanizm tworzenia konkretnych, wskazanych obiektów za pomocą fabryki.

4 Narzędzia i technologie

W rozdziale zarysowano aspekty wykorzystanych w realizowanym projekcie technologii.

4.1 Kotlin

Język programowania *Kotlin* [17] wywodzi się z rodziny języków opartych na *Wirtualnej Maszynie Java (JVM)*. *Wirtualna Maszyna Java* odpowiada za wykonanie kodu bajtowego języka *Java*. Kod bajtowy powstaje po etapie kompilacji kodu źródłowego języka *Java*. Aby skompilować kod dla *JVM* możliwe jest użycie innego niż *Java* języka, o ile ten język wywodzi się z rodziny opartej na *JVM*. Do rodziny tego typu języków zalicza się:

- *Clojure* – implementacja języka *Lisp* dla *JVM*,
- *AspectJ* – realizuje paradygmat programowania aspektowego,
- *Free Pascal* – implementacja języka *Pascal* dla *JVM*,
- *Groovy* – język skryptowy, obiektowy, rozszerza składnię *Javy*,
- *Scala* – język funkcyjny, obiektowy,
- *JRuby* – implementacja języka *Ruby* dla *JVM*,
- *Jython* – implementacja języka *Python* dla *JVM*.

Istnieje kilka innych propozycji języków programowania dla *JVM*, jednak pozostają one mniej znane, niszowe, o węższym zakresie, mniej powszechne lub wysoce specjalistyczne.

Wykonanie kodu języka opartego na *JVM* odbywa się etapami:

1. utworzenie kodu źródłowego wybranym i zgodnym z *JVM* języku programowania,
2. kompilacja kodu źródłowego do kodu bajtowego *Maszyny Wirtualnej Javy*,
3. Wykonanie kodu bajtowego przez *Maszynę Wirtualną Javy*.

Uruchomienie implementacji w tak przedstawiony sposób posiada następujące korzyści:

- *Maszyna Wirtualna Javy* może zostać zainstalowana w dowolnym systemie operacyjnym, o ile istnieje implementacja dla wybranego systemu. Może wykonywać kod bajtowy na procesorze, na którym zainstalowany jest ten system. Konkretna wersja *JVM* dedykowana jest konkretnemu procesorowi albo konkretnej konfiguracji sprzętowej. Analogicznie działa dowolny kompilator, który kompiluje kod dla wybranego procesora i nie jest kompatybilny z innym procesorem,
- kod źródłowy może zostać napisany w dowolnym, zgodnym z *JVM* językiem programowania. Języki można wymieniać ze sobą i pozostają ze sobą kompatybilne. Implementacje zostają skompilowane do kodu bajtowego *JVM*, który będzie identyczny dla tej samej logiki, chociaż byłaby napisana w odmiennych językach *JVM*,
- kod źródłowy napisany w określonym środowisku ma charakter przenośny i może zostać przetransferowany i skompilowany w dowolnym innym środowisku, na którym działa *JVM*. Za wykonie kodu na procesorze odpowiada określona *JVM*, natomiast kod bajtowy może być przenoszony z jednego systemu na inny bez ryzyka utraty zgodności z wybranym procesorem, na którym działa system i *JVM*.

Zastosowanie *JVM* pomiędzy kodem źródłowym, a procesorem w miejsce klasycznego kompilatora zapewnia dodatkowy poziom abstrakcji zwiększając przenośność i elastyczność procesu kompilacji i interpretacji.

Język *Kotlin* jest dialektem języka *Java* stworzonym i rozwijanym przez firmę *JetBrains*. Możliwa jest automatyczna konwersja języka *Kotlin* do Języka *Java* wspierana przez produkty firmy. Zintegrowane środowisko programistyczne *IntelliJ IDEA*, dla którego zrealizowano prototyp w prezentowanej pracy zostało stworzone przez firmę *JetBrains*.

Pod względem funkcjonalnym język *Kotlin* posiada bardziej rozbudowaną dokumentację dot. tworzenia wtyczek do produktu *IntelliJ IDEA*. Dzięki specyficznej składni możliwe są np. konstrukcje wiążące ze sobą formularze z graficznego interfejsu użytkownika z przechwytywaniem danych z przesłanych formularzy i ich dynamicznym wiązaniem do zmiennych w celu dalszego przetwarzania. Takie podejście zapewnia konstrukcja zgodna z *Kotlin UI DSL Version 2* [18]. W praktyce efekt dynamicznego wiązania danych w formularzach byłby trudny do realizacji w czystym języku *Java*.

W realizowanej pracy postanowiono wybrać język *Kotlin* do implementacji prototypu ze względu na:

- wygodę konstrukcji specyficznych rozwiązań programistycznych podczas programowania wtyczek, trudnych do osiągnięcia w języku *Java*,
- bogatszą oficjalną dokumentację dotyczącą tworzenia wtyczek do programu *IntelliJ IDEA*,
- wyspecjalizowaną społeczność, skupioną wokół języka *Kotlin* i konstrukcji oprogramowania dla wtyczek,
- wsparcie firmy *JetBrains* dla języka *Kotlin*, przez co język *Kotlin* jest bardziej natywny dla środowiska *IntelliJ IDEA*.

4.2 Java

Język programowanie *Java* ze względu na w pełni obiektowy paradygmat programowania stanowi użyteczne środowisko do wdrażania implementacji wzorców projektowych. Implementacja biblioteki standardowej języka *Java* zawiera różne zaimplementowane wzorce projektowe:

- biblioteka *Swing* wspierająca tworzenie komponentów graficznego interfejsu użytkownika (*GUI*) posiada klasę *JButton* odpowiedzialną za tworzenie przycisków w interfejsie. Klasa *ActionListener* pozwala na nasłuchiwanie zdarzeń na przycisku. W nomenklaturze wzorców projektowych klasa *JButton* stanowi podmiot, natomiast klasa *ActionListener* stanowi obserwator. W opisanym przypadku zastosowano wzorec *Obserwator*,
- pakiet *java.io* odpowiedzialny za obsługę strumieni wejścia-wyjścia posiada w swojej implementacji wzorec projektowy *Dekorator*. Klasa *InputStream* stanowi klasę dekorowaną i jest odpowiedzialna za przechwytywanie strumieni wejścia. Klasa ma implementację najbardziej ogólną i może identyfikować wiele rodzajów strumieni wejściowych, jednak bez ich uszczegółowienia. Dekoratorem opisaney klasy jest przykładowo klasa *FilterInputStream*, z kolei jej dekoratorem jest klasa *BufferedInputStream* zapewniająca bufor dla strumienia

wejściowego z metodami oznaczania danych w buforze i resetowania. *Dekoratory* rozszerzają podstawową funkcjonalność klasy dekorowanej,

- metoda statyczna *sort()* z klasy *Arrays* zawiera wbudowany wzorzec projektowy *Metoda Szablonowa*. Możliwe jest sortowanie różnych elementów, wzorzec dostarczy odpowiedni algorytm, jednak warunkiem jest implementacja interfejsu *Comparable* i wskazanie w metodzie *compareTo()* warunku sortowania. Pozostałą logikę wymaganą do posortowania elementów dostarczy *Metoda Szablonowa*,
- interfejs *java.util.Iterator* zapewnia logikę do implementacji wzorca projektowego *Iterator* dla klas implementujących ten interfejs i jego metody. Obiekty takich klas za pomocą metod z interfejsu *Iterator* mają zapewniony mechanizm iteracji bez wnikania w strukturę obiektów iterowanych,
- pakiet *java.lang.reflect* zapewnia implementację wspomagającą użycie wzorca projektowego *Pośrednik*. W celu podstawienia odpowiednich klas przez pośrednika każda z tych klas, które stanowią kandydaturę do wyboru przez pośrednika powinna zaimplementować interfejs *InvocationHandler*. Należy zmapować każdy z utworzonych obiektów na nazwę metody obsługiwanej przez pośrednika. *Pośrednik* po wywołaniu odpowiedniej metody zwróci odpowiadający metodzie obiekt z jego unikalną implementacją. Klient może wywoływać metody z mapowanymi obiektami tylko poprzez pośrednika. Zwracane obiekty są tego samego typu, ale mają odmienną logikę.

Ze względu na użyteczność języka *Java* pod kątem implementacji różnego rodzaju wzorców projektowych oraz jej wysoką popularność wśród społeczności [19] skupionej wokół tego języka w prezentowanym w niniejszej pracy prototypie postanowiono, że generowany przez wtyczkę kod dla wybranego wzorca projektowego będzie napisany w języku *Java*.

4.3 IntelliJ Platform Plugin SDK

Platforma IntelliJ [20] od firmy *JetBrains* stanowi oprogramowanie *otwarto-źródłowe (open source)*. Zapewnia wsparcie dla produktów firmy. Platforma skupia się wokół rozwoju zintegrowanego środowiska programistycznego (*IDE*) i pochodzących od niego produktów. Oprogramowanie udostępniono na licencji *open-source Apache*. W rozwoju produktów opartych na bazie platformy może mieć udział każdy zainteresowany tym tematem programista.

Platforma IntelliJ zapewnia kompletną infrastrukturę do tworzenia implementacji na rzecz zintegrowanych środowisk programistycznych firmy *JetBrains*. Posiada charakter wieloplatformowy, opiera się na budowie komponentowej i działa na *Wirtualnej Maszynie Javy*.

Do produktów firmy opartych na jej platformie należą przykładowo środowiska:

- *PhpStorm* – dla języka *PHP*,
- *PyCharm* – dla języka *Python*,
- *Rider* – dla platformy *.NET*,
- *RubyMine* – dla języka *Ruby*,

- *RustRover* – dla języka *Rust*,
- *WebStorm* – dla języka *JavaScript* i *TypeScript*,
- *GoLand* – dla języka *Go*,
- *CLion* – dla języka *C* i *C++*.

Na bazie platformy *IntelliJ* powstało środowisko programistyczne *Android Studio* do tworzenia aplikacji mobilnych w języku *Java* lub *Kotlin*, wydane przez podmiot nie związany z firmą *JetBrains*.

Rozwój możliwych do rozbudowy funkcjonalności, oferowanych ze strony platformy obejmuje następujące moduły:

- narzędzia do kreacji graficznego interfejsu użytkownika m. in. okien narzędzi, widoków, wyskakujących elementów menu, okien dialogowych, widoków drzew i list,
- manipulator edytorem tekstowym wraz z abstrakcyjnymi implementacjami kolorowania składni, składania i uzupełniania kodu, edycją składni, tekstu sformatowanego,
- edytor obrazów,
- otwarte interfejsy *API* do rozwijania *IDE* w tym m. in. modelu projektu, systemu kompilacji, debugowania z obsługą punktów przerwań niezależną od języka, stosu wywołań, oknami podglądu i ewaluację wyrażeń,
- interfejs struktury programu (*PSI*), umożliwiający m. in. analizę plików, kreację rozbudowanych modeli syntaktycznych i semantycznych kodu, budowanie indeksów, szybką nawigację po plikach, typach i symbolach oraz po zawartości wyskakujących okienek, znajdowanie zastosowań kodu z sugestią jego uzupełnienia, inspekcję kodu, przepisywanie kodu w celu szybkich poprawek lub refaktoryzacji. Model *PSI* oferowany przez platformę posiada wsparcie dla wielu języków programowania i ma rozszerzalny charakter.

Produkty zbudowane na bazie platformy *IntelliJ* zapewniają swoją elastyczność i rozszerzalność, pozostając otwarte na dalsze modyfikacje i rozwój. Mechanizm dodawania istniejących wtyczek do produktów firmy *JetBrains* i tworzenia własnych jest w pełni obsługiwany przez platformę. Możliwa jest dystrybucja wtyczek dla innych użytkowników poprzez oficjalne repozytorium *JetBrains Marketplace*. Możliwe jest także wykorzystanie nieoficjalnych repozytoriów. Od wersji *IntelliJ IDEA CE 2021.1*, dostępna jest opcja dodawania do środowiska *CE* wtyczek o zamknięto-źródłowym kodzie.

Platforma *IntelliJ* nie posiada oddzielnej implementacji w postaci zewnętrznego projektu. Całość funkcjonalności oferowanych przez platformę pokrywa się z kodem programu *IntelliJ IDEA Community Edition* [21] będącym bezpłatną wersją *IntelliJ IDEA Ultimate*. Różnica pomiędzy wersjami dotyczy istnienia wtyczek i funkcjonalności o zamkniętym kodzie źródłowym w wersji *Ultimate*.

W realizowanej pracy, w celu realizacji założonych wymagań wykorzystano wybrane funkcjonalności oferowane przez *IntelliJ Platform Plugin SDK*.

4.4 IntelliJ Platform Plugin Template

IntelliJ Platform Plugin Template stanowi repozytorium w serwisie *GitHub* [22]. Projekt w repozytorium posiada implementację w pełni skonfigurowanego środowiska wtyczki do programu *IntelliJ IDEA Community Edition*. Stanowi bazę dla dalszych konfiguracji projektu i jego rozwoju.

Wykorzystanie istniejącego, wstępnie skonfigurowanego środowiska z implementacją wtyczki, znacząco usprawnia proces tworzenia nowej wtyczki do programu *IntelliJ IDEA*. Zmniejsza się próg wejścia dla inicjalizacji tego procesu oraz minimalizuje nakłady związane z debugowaniem aplikacji. Koszty mogą rosnąć w wyniku błędów powstałych podczas tworzenia implementacji na własną rękę. Mają tendencję malejącą przy wykorzystaniu wstępnej konfiguracji projektu i początkowego przygotowywania środowiska dla dalszych etapów rozwoju wtyczki.

Konfiguracja wstępnego projektu wtyczki opiera się na języku *Gradle*. Surowy projekt zapewnia szkielet wtyczki oraz dodatkowo konfigurację procesu *continuous integration* i linków z referencją do odpowiednich stron w dokumentacji. W celu zaadaptowania projektu do własnych celów należy wykonać polecenie *Użyj tego szablonu* poprzez przycisk na stronie projektu w serwisie *GitHub*. Od tego momentu możliwe jest klonowanie repozytorium z własnych zasobów *GitHub* w celu rozpoczęcia implementacji. Sklonowane z serwisu *GitHub* repozytorium należy wczytać do programu *IntelliJ IDEA* jako projekt dla dalszych modyfikacji.

W realizowanym projekcie rozwój funkcjonalności prototypu zrealizowano na podstawie implementacji *IntelliJ Platform Plugin Template*, wykorzystując zalety płynące z tego rozwiązania.

4.5 IntelliJ IDEA CE

Zintegrowane środowisko programistyczne *IntelliJ IDEA* w bezpłatnej, *otwarto-źródłowej* wersji *Community Edition* [23] wspiera programowanie w języku *Java* lub *Kotlin*. Posługuje się rozbudowanym zestawem użytecznych narzędzi i nowoczesnych, innowacyjnych rozwiązań. Środowisko zapewnia także wsparcie dla:

- *frameworka Spring* [24], który znacznie przyspiesza i ułatwia produkcję oprogramowania w języku *Java* i *Kotlin*. Wspiera mechanizm wstrzykiwania zależności, odwrócenia sterownia, modułową budowę,
- platformy *Jakarta EE* [25] będącą projektem *open source*, wspierającym tworzenie aplikacji biznesowych. Zapewnia: przetwarzanie rozproszone, rozbudowane usługi internetowe, natywne wsparcie dla chmury, wstrzykiwanie zależności, rozwój usług związanych z integracją dla rozwiązań sztucznej inteligencji i uczenia maszynowego oraz różne, inne usprawnienia. *Jakarta EE* jest nadzbiorem bibliotek dla standardu *Java SE*, zawiera w sobie funkcje *Java SE* i dodatkowo rozbudowuje standard o nowe funkcjonalności. *Jakarta EE* jest rozwijana przez fundację *Eclipse*,
- profilowania aplikacji, umożliwiającego monitorowanie wydajności jej działania,
- obsługi zapytań poprzez protokół *HTTP*, *gRPC*, *GraphQL*, *WebSockets*,
- zarządzania relacyjnymi lub obiektowymi bazami danych poprzez zestaw użytecznych narzędzi.

Bazowe mechanizmy środowiska *IntelliJ IDEA CE* zapewniają:

- inteligentny edytor kodu wspierający programowanie w języku *Kotlin* lub *Java*, oparty o sztuczną inteligencję asystent tworzenia kodu, analizator wstrzykniętego kodu, automatyczne uzupełnianie kodu, inspekcje kodu na bieżąco, sugerowane na żywo szablony, automatyczną refaktoryzację,

- programowanie poprzez grupę w sposób zdalny,
- debugger, terminal, dekompiletor,
- wsparcie konteneryzacji,
- wsparcie dla przepływu pracy, narzędzi do budowania projektu, uproszczonej wstępnej konfiguracji projektu, prostego uruchamiania aplikacji, procesu testowania w tym ciągłego automatycznego testowania,
- wykorzystanie systemu kontroli wersji,
- narzędzia do nawigacji i różnych form przeszukiwania,
- konfigurowalne motywy graficzne.

Wersja *IntelliJ IDEA Ultimate* posiada więcej wbudowanych narzędzi. Na potrzeby realizacji prototypu w niniejszej pracy postanowiono wykorzystać środowisko *IntelliJ IDEA* w wersji *Community Edition* ze względu na w pełni otwarty kod źródłowy oraz wystarczającą funkcjonalność. Środowisko *IntelliJ IDEA* wymieniane jest w wielu internetowych zestawieniach [26] świadcząc o wysokiej popularności wśród programistów. Ze względu na:

- otwarto-źródłowy kod,
- popularność środowiska,
- wspierany system wtyczek,
- bogatą i zróżnicowaną funkcjonalność

postanowiono wykorzystać środowisko *IntelliJ IDEA CE* do realizacji opisywanego projektu.

4.6 Git

Git [27] stanowi otwarto-źródłowe oprogramowania z grupy systemów kontroli wersji (*VCS*). Zostało wydane na licencji *GNU GPL v2*. System kontroli wersji pozwala śledzić zmiany dokonywane w kodzie programu podczas procesu implementacji. Możliwe jest tworzenie, synchronizacja i łączenia zmian dokonywanych w tym samym kodzie przez wielu programistów. *Git* jest systemem rozproszonym. Istnieje możliwość tworzenia oprogramowania na wielu różnych komputerach-węzłach. *Git* cechuje szybkość działania, mała waga, prosta obsługa i dość rozbudowana funkcjonalność. Może być użyteczny zarówno dla zespołu jak i pojedynczego programisty.

W *Git* możliwe jest dzielenie procesu kodowania na etapy. Każdy etap stanowi migawkę (*snapshot*) aktualnego stanu implementacji. Zbiory migawek grupuje się w gałęzie (*branch*). Gałęzie z danym stanem implementacji można kopiować i rozwijać niezależnie od siebie. Możliwe jest scalanie gałęzi (*merge*) w celu łączenia różnych implementacji w jedną wspólną. Za pomocą *Git* można przywrócić stan implementacji do dowolnego zapamiętanego momentu z przeszłości oraz przełączać implementację wraz w gałęziami.

Kod źródłowy zapamiętany w systemie *Git* można udostępniać na zewnętrznych, internetowych repozytoriach np. *GitHub*, *GitLab*, *BitBucket* oraz współdzielić z klientami tych serwisów. Możliwe jest tworzenie implementacji w sposób zdalny, wspólnie ze społecznością na całym świecie. Istnieje wiele popularnych, powszechnie używanych projektów tworzonych przez społeczność *open source* w skali globalnej. Przykładem jest lista najpopularniejszych projektów w serwisie *GitHub* w danym dniu [28].

Git posiada bardziej rozbudowaną funkcjonalność, możliwe jest m. in.:

- identyfikacja użytkowników pod kątem wprowadzonych w kodzie zmian,
- wyświetlanie różnic pomiędzy różnymi wersjami plików,
- oznaczanie różnych wersji kodu własnymi *tagami* oraz tworzenie *aliasów* dla złożonych poleceń systemu *Git*,
- obsługę konfliktów scalania w sposób automatyczny bądź ręczny,
- osadzanie jednego repozytorium *Git* w innym jako *submoduł*,
- automatyzacja zarządzaniem kontrolą wersji za pomocą mechanizmu *hooków* i *skryptów*,
- obsługa dużych plików np. obrazów, filmów oraz możliwość pobrania tylko części rozbudowanego repozytorium,
- zapewnianie bezpieczeństwa danych oraz *kryptografii*,
- integracja z graficznymi interfejsami w celu wizualizacji stanu *repozytorium* oraz integracja z różnymi zintegrowanymi środowiskami programistycznymi.

Na potrzeby realizacji niemniejszej pracy wykorzystano tylko podstawowe funkcjonalności systemu *Git*. Cel stanowi wsparcie, usprawnienie i lepsza kontrola nad procesem tworzenia oprogramowania dla prototypu.

5 Opis implementacji prototypu

Prezentowany rozdział zawiera opis implementacji prototypu. Opisano proces konfiguracji szablonu projektu, strukturę projektu, strukturę klas oraz diagramy aktywności i sekwencji. Całość uzupełniono o logi z systemu kontroli wersji *Git* odnoszące się do projektu.

5.1 Konfiguracja szablonu

Proces konfiguracji szablonu projektu rozpoczęto od jego inicjalizacji poprzez serwis *GitHub*. Serwis umożliwia wykonanie odgałęzienia (*fork*) szablonu. Potrzebne jest nadanie własnej nazwy takiemu odgałęzieniu projektu.

Kolejny etapem stanowi proces importowania projektu do zintegrowanego środowiska programistycznego *IntelliJ IDEA CE*. Po zaimportowaniu w konfiguracji projektu dokonano następujących zmian:

- we właściwościach projektu ustawiono wersję kompilacji w języku *Java* zgodną z *Software Development Kit (SDK) 11*,
- w pliku konfiguracyjnym *gradle.properties* przełączono flagę *kotlin.stdlib.default.dependency* na *true*, ze względu na wymagalność użycia zależności biblioteki standardowej podczas tworzenia implementacji w języku *Kotlin*,
- usunięto domyślną zawartość katalogu *resources* z wyjątkiem pliku konfiguracyjnego *plugin.xml*,
- w katalogu *resources* dodano pliki z ikonami oraz czcionkami na potrzeby ustawień graficznego interfejsu użytkownika,
- w pliku konfiguracyjnym *plugin.xml* przedstawionym na *Listingu 6* w katalogu *resources* nadano nazwę projektu, określono jego dostawcę (*vendor*), skonfigurowano wyskakujące okna informacyjne typu *Balloon* oraz określono klasy, które mają zostać wykonane w wyniku zadanych przez użytkownika akcji w *GUI*.

Listing 6. Zawartość pliku konfiguracyjnego *plugin.xml*.

```
<!-- Plugin Configuration File. Read more:
https://plugins.jetbrains.com/docs/intellij/plugin-configuration-
file.html -->
<idea-plugin>
  <id>com.github.dominikstec.designpatternify</id>
  <name>Design-Patternify</name>
  <vendor>dominik-stec</vendor>

  <depends>com.intellij.modules.platform</depends>
  <depends>org.jetbrains.kotlin</depends>

  <extensions defaultExtensionNs="com.intellij">

    <notificationGroup id="DesignPatternify.NotificationGroup"
                      displayType="BALLOON" />
  </extensions>
</idea-plugin>
```



```

</extensions>

<!-- Create a group for UI actions. -->
<actions>
    <group
        id="DesignPatternify.ActionGroup"
        text="Design Patternify"
        popup="true"
        icon="/icons/ic_extension.svg">
        <add-to-group group-id="ToolsMenu" anchor="first" />
        <!-- DP Observer -->
        <action id="designpatternify.gui.ActionObserver"
class="com.github.dominikstec.designpatternify.gui.ActionObserver"
        description="Generuje kod wzorca projektowego
obserwator" text="Design Pattern Observer" icon="/icons/ic_extension.svg"
>
            <keyboard-shortcut first-keystroke="ctrl alt 1"
keymap="$default" />
        </action>
        <!-- DP Decorator -->
        <action id="designpatternify.gui.ActionDecorator"
class="com.github.dominikstec.designpatternify.gui.ActionDecorator"
text="Design Pattern Decorator"
        description="Generuje kod wzorca projektowego
dekorator" icon="/icons/ic_extension.svg" >
            <keyboard-shortcut first-keystroke="ctrl alt 2"
keymap="$default" />
        </action>
        <!-- DP Adapter -->
        <action id="designpatternify.gui.ActionAdapter"
class="com.github.dominikstec.designpatternify.gui.ActionAdapter"
text="Design Pattern Adapter"
        description="Generuje kod wzorca projektowego
adapter" icon="/icons/ic_extension.svg" >
            <keyboard-shortcut first-keystroke="ctrl alt 3"
keymap="$default" />
        </action>
        <!-- DP Factory -->
        <action id="designpatternify.ui.ActionFactory"
class="com.github.dominikstec.designpatternify.gui.ActionFactory"
        text="Design Pattern Factory" description="Generuje
kod wzorca projektowego fabryka"
        icon="/icons/ic_extension.svg" >
            <keyboard-shortcut first-keystroke="ctrl alt 4"
keymap="$default" />
        </action>
    </group>

</actions>
</idea-plugin>

```

5.2 Rozwiązania implementacyjne

Opisywany prototyp umożliwia generowanie czterech rodzajów wzorców projektowych:

- *Obserwator*,
- *Dekorator*,
- *Adapter*,
- *Fabryka*.

W strukturze projektu w celu realizacji logiki konfiguracji i generowania plików źródłowych dla pojedynczego wzorca wymagane są trzy klasy. Ich implementacja umożliwia spełnienie wymagań funkcjonalnych systemu. Sposób konstrukcji klas dla każdego wzorca jest podobny. Drobne wyjątki wynikają ze specyfiki danego wzorca. Cały projekt prototypu buduje dwanaście klas.

5.2.1 Klasy AnAction

Interakcja z wtyczką rozpoczyna się od wywołania akcji dla wybranego wzorca. Przed wywołaniem wzorca istnieje możliwość wskazania klasy w projekcie, z której zostaną wczytane deklaracje występujących w niej metod lub pól. Deklaracje zostaną przekazane dalej w przepływie sterowania. Na podstawie tych metod i pól nastąpi generowanie kodu wzorca. Dzięki temu łatwiejsza jest integracja wygenerowanego kodu z kodem już istniejącym. Opisany mechanizm nie dotyczy wzorca projektowego *Fabryka*. Funkcjonalność fabryki nie jest związana z istnieniem metod lub pól w wybranej klasie, w istniejącym w projekcie.

W przypadku klasy *AnAction* dla wzorca projektowego *Fabryka* przedstawionej na *Listingu 7* implementacja jest uproszczona. Po wywołaniu akcji sterowanie zostaje przekazane od razu klasie *MainFactory*.

Listing 7. Kod źródłowy Klasy ActionFactory wzorca projektowego Fabryka.

```
package com.github.dominikstec.designpatternify.gui

import com.github.dominikstec.designpatternify.gui.dialogs.MainFactory
import com.intellij.openapi.actionSystem.AnAction
import com.intellij.openapi.actionSystem.AnActionEvent

class ActionFactory : AnAction() {
    override fun actionPerformed(event: AnActionEvent) {
        MainFactory(event).show()
    }
}
```

W przypadku pozostałych trzech wzorców logika implementacji uwzględnia wczytywanie pól i metod ze wskazanej klasy.

W przypadku wzorca *Adapter* wskazanie klasy zawierającej właściwe metody jest obligatoryjne, natomiast w przypadku wzorca *Obserwator* i *Dekorator* jest opcjonalne. Brak wskazania poprawnej metody lub klasy we wzorcu *Adapter* jest sygnalizowane użytkownikowi.

Implementacja wychwyci czy adaptowana klasa może być dziedziczona, czy nie jest abstrakcyjna i czy ma publiczny identyfikator dostępu lub przypisany dostęp pakietowy. Funkcjonalność notyfikacji użytkownikowi błędu wczytywania klasy zaprezentowano na *Listing 8*.

Listing 8. Fragment kodu źródłowego klasy ActionAdapter odpowiedzialny za notyfikację błędu wczytywania klasy użytkownikowi.

```
//select class from project as parent for generation
//class must be selected
if (execute) {

    // can be class adaptive ?
    val ac = children.filter({ it.toString().contains("PsiClass:") })
        .map({ it.text.substring(0, it.text.indexOf("{")) })
        .filter({ (it.contains("private").not()) &&
(it.contains("protected").not()) && it.contains("abstract").not() })

    accessClass = ac.isNotEmpty()

    //get class name
    children.filter({ it.toString().contains("PsiClass:") })
        .forEach({ className = it.toString().removePrefix("PsiClass:") })

    // class can be adaptive
    if (accessClass) {...} else {

        //notification if class was not properly
        Messages.showMessageDialog(
            event.project,
            "Klasa ${className} nie może być adaptowana.",
            "Nie wskazano właściwej klasy",
            Messages.getInformationIcon()
        )

    }
} else {
    //notification if class was not point
    Messages.showMessageDialog(
        event.project,
        "Wskaż klasę do adaptowania.",
        "Nie wskazano klasy",
        Messages.getInformationIcon()
    )
}
```

Mechanizm wczytywania zaimplementowanych w danej klasie metod, którego przykład dla wzorca *Adapter* przedstawia *Listing 9* przechodzi przez następujące fazy:

- wczytuje wszystkie zmienne typu *PsiElement* konteneryzujące metody ze wskazanej klasy,
- eliminuje zduplikowane nazwy metod,
- eliminuje metody objęte komentarzem w celu ich dezaktywacji,
- konwertuje konteneryzowane nazwy metod i ich argumentów z typu *PsiElement* na typ *String*,
- usuwa opisowe komentarze umieszczone za deklaracją metody,
- usuwa metody statyczne oraz metody nie deklarowane jako publiczne lub nie o dostępie pakietowym,

- eliminuje średnik kończący deklarację,
- usuwa nadmiarowe białe znaki występujące pomiędzy nazwą metody i jej argumentami,
- skleja nazwę metody z jej argumentami jako typ *String* i takie zestawy metod eksportuje jako listę.

Listing 9. Fragment implementacji klasy `ActionAdapter` przetwarzającej nazwy metod i ich argumentów dla wzorca projektowego `Adapter`.

```
//get list of PSI element with PsiIdentifier:getValue1
//and PsiParameterList:(String val2, String val3)
val methodNameArgsMap = LinkedHashMap<String, String>()
for(p in setPsi) {
    val word = p.childrenOfType<PsiElement>()
    var splitName = ""
    var splitArgs = ""
    for(w in word) {
        val word = w.originalElement.toString()
        if(word.contains("PsiIdentifier")) {
            splitName = word.split(":").get(1)
        }
        else if(word.contains("PsiParameterList")) {
            splitArgs = word.split(":").get(1)
        }
        if(splitName != "" && splitArgs != "") {
            methodNameArgsMap[splitName] = splitArgs
            splitName = ""
            splitArgs = ""
        }
    }
}

//bug hotfix: if comment appears in results then delete it
val hotfixSet = HashSet<String>()
for (item in set) {
    var part = listOf<String>()
    if (item.contains("//")) {
        part = item.split(";")
    } else if (item.contains("/*")) {
        part = item.split("*/")
    }
    if (part.isEmpty().not()) {
        for (p in part) {
            if (p.contains("//").not() &&
                p.contains("/*").not()) {
                hotfixSet.add(p)
            }
        }
    } else {
        hotfixSet.add(item)
    }
}

//substring methods key words declaration
val splitMethods = HashSet<String>()
for(s in hotfixSet) {
    val begin = 0
    val end = s.indexOf("{")
    try{
```

```

        val split = s.substring(begin, end)
        if(split.contains("private") ||
            split.contains("protected") ||
            split.contains("static")) {
            continue
        } else {
            splitMethods.add(split)
        }
    } catch (exc: IndexOutOfBoundsException) {
        continue
    }
}

```

Wczytane i przetworzone nazwy metod wraz z argumentami z klasy *ActionAdapter* zostają przekazane do klasy *SettersAdapter* w celu dalszego przetwarzania.

Implementacja logiki w klasie *ActionAdapter* dot wczytywania metod i generowania wzorca projektowego *Adapter* jest analogiczna w klasie *ActionObserver* oraz *ActionDecorator* dla wzorców kolejno: *Obserwator* i *Dekorator*, ale nie dotyczy metod tylko pól. Dodatkowo przekazywanym do dalszego przetwarzania parametrem z klas typu *AnAction* jest nazwa klasy, z której następuje wczytywanie metod lub pól.

5.2.2 Klasy setters DialogWrapper

Po wskazaniu klasy bazowej, z której nastąpi wczytywanie zadeklarowanych pól lub metod użytkownikowi wyświetlane jest okno dialogowe. Możliwe jest zatwierdzenie lub anulowanie wybranych pól lub metod. Wybór możliwy jest za pomocą obiektów typu *checkbox* reprezentujących wybrane pola lub metody. W *Rozdziale 5.2.1.* pokazano proces wyszukiwania w kodzie tych pól i metod. *Listing 10* zawiera implementację odpowiedzialną za wyświetlenie okna dialogowego z wyborem wczytanych metod w klasie *SettersAdapter*.

Przetawione podejście ułatwia proces integracji wygenerowanego kodu z już istniejącym kodem. Proces ułatwia wskazanie tylko niezbędnych pól lub metod podlegających dalszemu przetwarzaniu przez system.

Listing 10. Metoda z klasy SettersAdapter odpowiedzialna za wyświetlenie okna dialogowego z wyborem wczytanych metod.

```

override fun createCenterPanel(): JComponent {

    super.setOKButtonText("DALEJ")
    super.setCancelButtonText("anuluj")
    super.setUndecorated(true)
}

```

```

    val dialogPanel = JPanel(GridLayout(7,2, ))

    //label for fields
    val fieldLabel = JLabel("Jakie metody klasy ${className.capitalize()}
    będą adaptowane?")
    val breakLineFieldTop = JLabel("")
    breakLineFieldTop.preferredSize = Dimension(300, 10)
    val currentFont = fieldLabel.getFont()
    val boldFont = Font(currentFont.fontName, Font.BOLD,
currentFont.size)
    fieldLabel.setFont(boldFont)

    //fields label positioning
    fieldLabel.verticalAlignment = SwingConstants.TOP
    fieldLabel.horizontalAlignment = SwingConstants.LEFT
    dialogPanel.add(fieldLabel, BorderLayout.NORTH, -1)
    dialogPanel.add(breakLineFieldTop, BorderLayout.SOUTH, -1)

    //checkboxes for fields
    for(f in methodList) {
        val cbox = JCheckBox(f, true)
        cbox.preferredSize = Dimension(15,15)
        cbox.verticalAlignment = SwingConstants.BOTTOM
        cbox.horizontalAlignment = SwingConstants.LEFT
        cboxFieldsHandlers.add(cbox)
        dialogPanel.add(cbox)
    }

    //<br/>
    val breakLineFieldBottom = JLabel("")
    breakLineFieldBottom.preferredSize = Dimension(300, 30)
    dialogPanel.add(breakLineFieldBottom, BorderLayout.SOUTH, -1)

    //<br/>
    val breakLineMethodBottom = JLabel("")
    breakLineMethodBottom.preferredSize = Dimension(300, 10)
    dialogPanel.add(breakLineMethodBottom, BorderLayout.SOUTH, -1)

    return dialogPanel
}

```

Potwierdzenie wyboru zaznaczonych obiektów typu *checkbox* z metodami lub polami z klasy bazowej tworzy instancję obiektu odpowiedzialnego za konfigurację wygenerowanego kodu. Wybrane dane zostaną przekazane do tej instancji obiektu. W przypadku anulowania okna dialogowego nastąpi wyjście z mechanizmów wtyczki do bieżącego widoku w *IntelliJ IDEA*.

Opisana logika wyboru obiektów typu *checkbox* jest analogiczna dla klas we wzorcach: *Obserwator* i *Dekorator*, jednakże dotyczy pól, a nie metod.

5.2.3 Klasy main DialogWrapper

Inicjalizacja instancji klasy *MainAdapter* powoduje wyświetlenie okna dialogowego stanowiącego konfigurator dla generatora kodu wzorca. W przypadku konfiguratora dla wzorców: *Obserwator*, *Dekorator*, *Adapter* i *Fabryka*, każdy z konfiguratorów będzie inny od pozostałych. Wynika to z wymagań i unikalnej, odmiennej struktury kodu generowanego dla każdego wzorca.

W przypadku wzorca projektowego *Adapter* na konstrukcję okna dialogowego, którego implementację zawiera *Listing 11* składają się:

- przycisk wyboru ścieżki dla generowania nowych plików w formie osobnego okna ze strukturą katalogów,
- pola tekstowe stałe w oknie dialogowym, określające nazwę interfejsu adaptera oraz nazwę adaptera,
- pole tekstowe generowane dynamicznie w liczbie zależnej od przekazanej ilości metod z obiektu typu *SettersAdapter* w kroku poprzedzającym wyświetlenie okna konfiguratora. Generowanie pól tekstowych odnosi się do liczby mapowanych metod z ich nazw źródłowych na nazwy metod z adaptera. Proces wczytywania metod został opisany w *Rozdziale 5.2.2.*,
- przyciski: generuj i anuluj,
- notyfikacje o wybranej ścieżce dla generowania kodu,
- implementacja graficznego interfejsu okna dialogowego.

Listing 11. Fragment implementacji klasy MainAdapter odpowiedzialnej za wyświetlenie okna konfiguratora dla generatora wzorca projektowego Adapter.

```
@Nullable
override fun createCenterPanel(): JPanel {

    super.setOKButtonText("GENERUJ")
    super.setCancelButtonText("anuluj")
    super.setUndecorated(true)

    //get project root directory path
    val manager = ModuleManager.getInstance(e.project!!)
    val modules: Array<Module> = manager.modules
    val root =
ModuleRootManager.getInstance(modules.first()).sourceRoots.first()

    //layout type and dimensions
    val rowSize = methodArgsMap.size + 3
    dialogPanel = JPanel(GridLayout(rowSize, 3, 20,20))
    dialogPanel.size = Dimension(400,300)

    // Create a border
    val grayBorder: Border = BorderFactory.createLineBorder(Color(20, 40,
60), 15)
    dialogPanel.border = grayBorder

    //path button
    val btnLabel = JLabel("Gdzie umieścić kod wzorca?")
    val btnLabelBold = JLabel(">> Wybierz pakiet źródłowy:")
    // Get the current font or set a default one
    var currentFont: Font = btnLabelBold.getFont()
    var boldFont = Font(currentFont.fontName, Font.BOLD,
currentFont.size)
    // Apply the bold font to the label
    btnLabelBold.setFont(boldFont)
    dialogPanel.add(btnLabel)
    dialogPanel.add(btnLabelBold)

    val btn = JButton(">> Wybór Pakietu <<")
```

```

btn.addActionListener {
    var rootPath = ""
    val descriptor = FileChooserDescriptor(
        false,
        true,
        false,
        false,
        false,
        false
    )
    var path: String
    try {
        path = FileChooser.chooseFile(descriptor, e.project!!,
root)!!.path
    } catch (ex: Exception) {
        path = ""
        okPathClick = false
    }
    if(path != "") runCancel = true
    val start = path.indexOf("src/")
    if (start == -1) rootPath = ""
    val end = path.length
    if (start != -1) rootPath = path.substring(start + 4, end)
    if (rootPath.contains("/")) {
        rootPath = rootPath.replace("/", ".", true)
    }
    sourcePath = rootPath
    if (rootPath != "") notificationInfo(e.project!!, "Wybrano
pakiet: " + rootPath)
}

dialogPanel.add(btn)

//text field for adapter interface
var txtLabel = JLabel("Jaka ma się nazywać interfejs adaptera?")
var txtLabelBold = JLabel(">> Podaj nazwę interfejsu:")
// Get the current font or set a default one
currentFont = txtLabelBold.getFont()
boldFont = Font(currentFont.fontName, Font.BOLD, currentFont.size)
// Apply the bold font to the label
txtLabelBold.setFont(boldFont)
dialogPanel.add(txtLabel)
dialogPanel.add(txtLabelBold)
var adapterInterfaceField = JPasswordField()
currentFont = adapterInterfaceField.getFont()
boldFont = Font(currentFont.fontName, Font.BOLD, currentFont.size)
// Apply the bold font to the label
adapterInterfaceField.setFont(boldFont)
// Request focus for the text field
SwingUtilities.invokeLater({
adapterInterfaceField.requestFocusInWindow() })

//handle adapter interface name
inputAdapterNamesHandler["interface"] = adapterInterfaceField

dialogPanel.add(adapterInterfaceField)

//text field for adapter interface
txtLabel = JLabel("Jak ma się nazywać adapter?")
txtLabelBold = JLabel(">> Podaj nazwę klasy:")

```



```

// Get the current font or set a default one
currentFont = txtLabelBold.getFont()
boldFont = Font(currentFont.fontName, Font.BOLD, currentFont.size)
// Apply the bold font to the label
txtLabelBold.setFont(boldFont)
dialogPanel.add(txtLabel)
dialogPanel.add(txtLabelBold)
var adapterField = JbTextField()
currentFont = adapterField.getFont()
boldFont = Font(currentFont.fontName, Font.BOLD, currentFont.size)
// Apply the bold font to the label
adapterField.setFont(boldFont)

//handle adapter name
inputAdapterNamesHandler["class"] = adapterField

dialogPanel.add(adapterField)

//extract type of return value from method
val idxArr: ArrayList<Int> = ArrayList()
for(s in setterMethods) {
    val part = s.tokenizeByWhitespace()
    var idx = 0
    for (p in part) {
        if(p.contains("("))
            break
        idx++
    }
    idxArr.add(idx-1)
}
val methodTypeArr: ArrayList<String> = ArrayList()
var t = 0
setterMethods.forEach{
methodTypeArr.add(it.tokenizeByWhitespace().get(idxArr[t]).dropWhitespaces()); t++ }

var i = 0
methodArgsMap.forEach {

    //text field for adapter interface
    txtLabel = JLabel("Jaka metoda ma być adapterem dla:")
    txtLabelBold = JLabel(">>  ${methodTypeArr[i]}
${it.key}${it.value}  >>")
    // Get the current font or set a default one
    currentFont = txtLabelBold.getFont()
    boldFont = Font(currentFont.fontName, Font.BOLD,
currentFont.size)
    // Apply the bold font to the label
    txtLabelBold.setFont(boldFont)
    dialogPanel.add(txtLabel)
    dialogPanel.add(txtLabelBold)

    // text field declaration
    adapterField = JbTextField()
    currentFont = adapterField.getFont()
    boldFont = Font(currentFont.fontName, Font.BOLD,
currentFont.size)
    // Apply the bold font to the label

```

```

        adapterField.setFont(boldFont)

        // adapter text fields handler
        inputTextHandlers.add(adapterField)

        dialogPanel.add(adapterField)
        i++
    }

    dialogPanel.background = Color(20, 40, 60)

    return dialogPanel
}

```

Po wpisaniu wymaganych i logicznie poprawnych danych w oknie dialogowym oraz wciśnięciu przycisku generuj następuje bindowanie podanych danych do zmiennych wykorzystywanych w generatorze. Wywołana zostaje metoda *doOKAction()* w odpowiedzi na zdarzenie generowania. W metodzie *doOKAction()* następuje walidacja podanych w polach tekstowych danych. Od pomyślnej walidacji zależy uruchomienie generatora plików z kodem.

Po uruchomieniu procesu generowania plików z kodem pierwszy etap stanowi utworzenie pustych plików o zadanych nazwach w podanej lokalizacji. Kolejny etap stanowi wypełnienie utworzonych pustych plików zawartością. Zawartość to szablon implementacji wybranego wzorca projektowego w języku *Java*. Fragment implementacji klasy *MainAdapter* odpowiedzialnej za generowanie zawartości pustego pliku źródłowego przedstawiono na *Listing 12*.

Listing 12. Fragment implementacji klasy MainAdapter odpowiedzialnej za generowanie zawartości pliku adaptera we wzorcu projektowym Adapter.

```

private fun createAdapterContent(project: Project) {
    val addFileContent = Runnable {
        ApplicationManager.getApplication().assertWriteAccessAllowed()

        var path = ""
        if (sourcePath != "") path = "${sourcePath}."

        //for first line abstract class
        var code = ""
        if (sourcePath == "") {
            code = ""
            public class ${adapter} implements ${iAdapter} {
                private ${setterClassName}
                ${toCamelCase(setterClassName)};

                public ${adapter} (${setterClassName}
                ${toCamelCase(setterClassName)}) {
                    this.${toCamelCase(setterClassName)} =
                ${toCamelCase(setterClassName)};
                }

                public interfaceMethods();
            }
        }
    }
}

```

```

        """.trimIndent()
    }
    //for next lines of abstract classes
    else code = """
        package ${sourcePath};

        public class ${adapter} implements ${iAdapter} {
            private ${setterClassName}

${toCamelCase(setterClassName)};

            public ${adapter} (${setterClassName}
${toCamelCase(setterClassName)}) {
                this.${toCamelCase(setterClassName)} =
${toCamelCase(setterClassName)};
            }

            public interfaceMethods();
        }
        """.trimIndent()

    val method: List<String>
    //if more then 1 method
    if (methods.contains("\n")) {
        method = methods.split("\n")
    }
    //if only one method
    else {
        method = listOf("")
    }

    val names: ArrayList<String> = ArrayList()
    val args: ArrayList<String> = ArrayList()
    for (m in methodArgsMap) {
        names.add(m.key)
        args.add(m.value)
    }

    //extract methods arguments names
    val argNameList: ArrayList<String> = ArrayList()
    argNameList.clear()
    for (a in args) {
        if (a.dropWhitespaces().length == 2) {
            argNameList.add("(")
        } else {
            var r = a.replace("(", "")
            r = r.replace(")", "")
            var argName = "("
            if (r.contains(",")) {
                val multi = r.split(",")
                var i = 0
                for (m in multi) {
                    if (i == multi.size - 1)
                        argName +=
m.tokenizeByWhitespace().get(1).dropWhitespaces()
                    else argName +=
m.tokenizeByWhitespace().get(1).dropWhitespaces() + ","
                    i++
                }
            }
            argNameList.add(argName + ")")
        }
    }
}

```

```

        var p = a.tokenizeByWhitespace().get(1)
        p = p.replace(")", "")
        argName += p.dropWhitespaces()
    }
    argName += ")"
    argNameList.add(argName)
}

var toAddCode = ""

//if more then 1 method
if (method[0] != "") {
    var i = 0
    for (m in method) {
        //without return statement
        if(m.contains("void")) {
            //first iter
            if (i == 0) {
                toAddCode += "@Override\n"
                toAddCode += ("\tpublic $m" + "" {
                    |
                    ${"\t\t"}${toCamelCase(setterClassName)}.${names[i]}${argNameList[i]};
                    |${"\t"}
                    """.trimMargin() + "\n")
                    toAddCode.trimIndent()
                    //iter between first and last
                } else if (i < method.size - 1) {
                    toAddCode += "\t@Override\n"
                    toAddCode += ("\tpublic $m" + "" {
                        |
                        ${"\t\t"}${toCamelCase(setterClassName)}.${names[i]}${argNameList[i]};
                        |${"\t"}
                        """.trimMargin() + "\n")
                        toAddCode.trimIndent()
                        //last iter
                    } else {
                        toAddCode += "\t@Override\n"
                        toAddCode += ("\tpublic $m" + "" {
                            |
                            ${"\t\t"}${toCamelCase(setterClassName)}.${names[i]}${argNameList[i]};
                            |${"\t"}
                            """.trimMargin()
                            toAddCode.trimIndent()
                        }
                        i++
                    }
                    //with return statement
                } else {
                    //first iter
                    if (i == 0) {
                        toAddCode += "@Override\n"
                        toAddCode += ("\tpublic $m" + "" {
                            |
                            ${"\t\t"}return
                            |
                            ${toCamelCase(setterClassName)}.${names[i]}${argNameList[i]};
                            |${"\t"}
                            """.trimMargin() + "\n")
                            toAddCode.trimIndent()
                            //iter between first and last
                        } else if (i < method.size - 1) {
                            toAddCode += "\t@Override\n"

```

```

        toAddCode += ("\tpublic $m" + "" {
            |    ${"\t\t"}return
${toCamelCase(setterClassName)}.${names[i]}${argNameList[i]};
            |${"\t"}
        ""}.trimMargin() + "\n")
        toAddCode.trimIndent()
        //last iter
    } else {
        toAddCode += "\t@Override\n"
        toAddCode += ("\tpublic $m" + "" {
            |    ${"\t\t"}return
${toCamelCase(setterClassName)}.${names[i]}${argNameList[i]};
            |${"\t"}
        ""}.trimMargin()
        toAddCode.trimIndent()
    }
    i++
}
}
//if only one method
} else {
    //without return statement
    if(methods.contains("void")) {
        toAddCode += "@Override\n"
        toAddCode += ("\tpublic $methods" + "" {
            |
${"\t\t"}${toCamelCase(setterClassName)}.${names[0]}${argNameList[0]};
            |${"\t"}
        ""}.trimMargin() + "\n")
        toAddCode.trimIndent()
        //with return statement
    } else {
        toAddCode += "@Override\n"
        toAddCode += ("\tpublic $methods" + "" {
            |    ${"\t\t"}return
${toCamelCase(setterClassName)}.${names[0]}${argNameList[0]};
            |${"\t"}
        ""}.trimMargin() + "\n")
        toAddCode.trimIndent()
    }
}

code = code.replace("public interfaceMethods();", toAddCode,
true)

//get project root directory path
val manager = ModuleManager.getInstance(e.project!!)
val modules: Array<Module> = manager.modules
val root =
ModuleRootManager.getInstance(modules.first()).sourceRoots

//convert source path format for find as psi file
var slashPath: String
if (sourcePath.contains(".")) {
    slashPath = sourcePath.replace(".", "/", true)
    slashPath = "/${slashPath}"
} else if (sourcePath.contains(".").not() && sourcePath != "") {
    slashPath = "/${sourcePath}"

```

```

    } else {
        slashPath = sourcePath
    }

    //get file from relative to root directory path
    val dir = PsiManager.getInstance(project)
        .findFile(root.first().findFileByRelativePath(slashPath + "/"
+ adapter + ".java"))!!
    val uniqueVirtualFile = dir!!.containingFile.virtualFile
    val documentFile =
FileDocumentManager.getInstance().getDocument(uniqueVirtualFile)

    //write to file
    documentFile!!.insertString(0, code)

}
WriteCommandAction.runWriteCommandAction(project, addFileContent)
}

```

Idea generatora polega na utworzeniu abstrakcyjnego szablonu dla określonej klasy wybranego wzorca. Następuje modyfikacja szablonu poprzez zamianę słów-kluczy z wygenerowanego, abstrakcyjnego szablonu na docelowe słowa-klucze. Słowa-klucze ustalane są na podstawie wpisów z okna dialogowego konfiguratora. Odnoszą się również do wybranych metod lub pól z klasy bazowej w projekcie, na rzecz której wywołano generowanie wzorca.

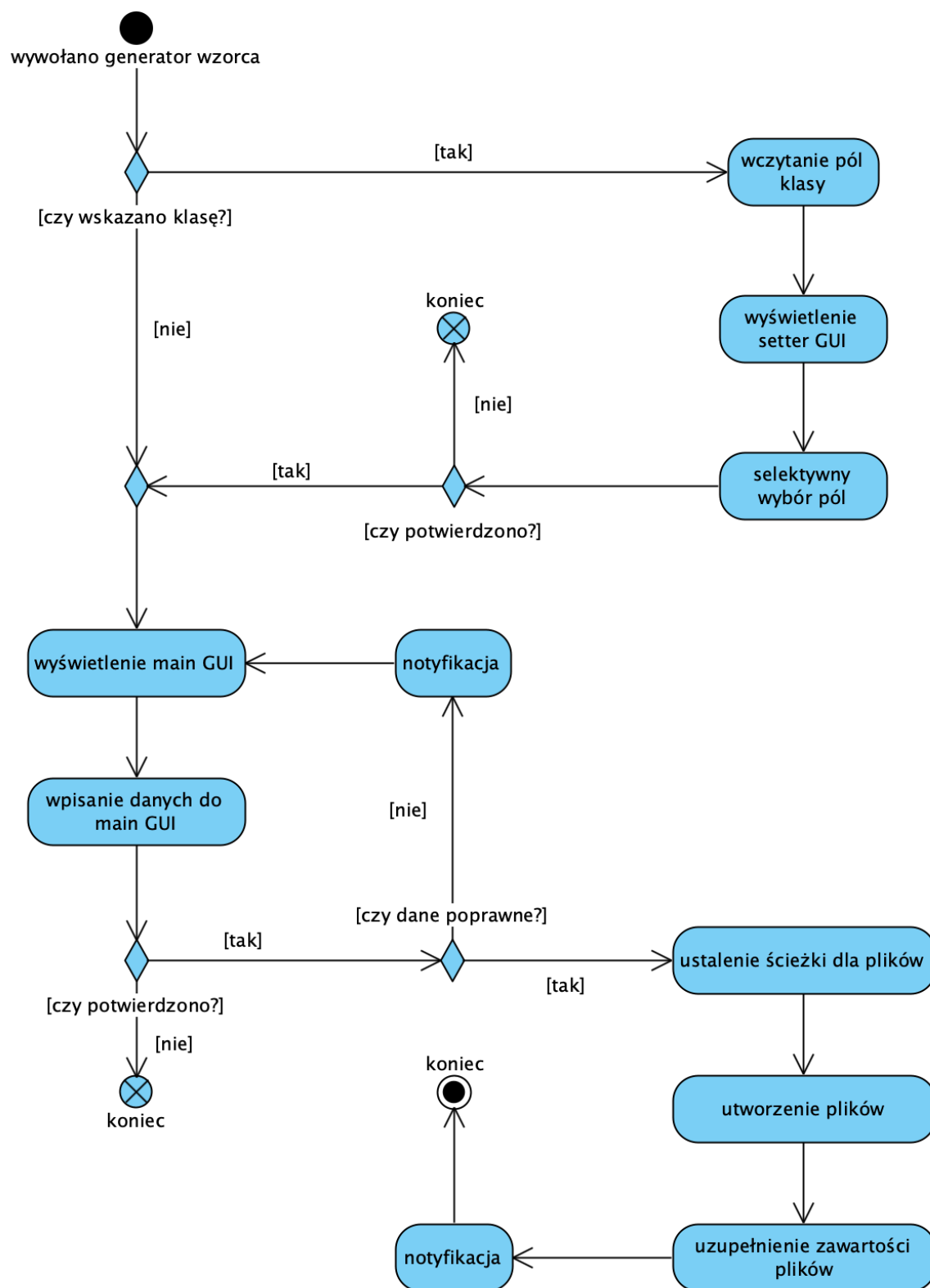
Opisany mechanizm konfiguratora i generatora w klasie *MainAdapter* jest analogiczny dla klas: *MainObserver*, *MainDecorator* i *MainFactory*, odnoszących się kolejno do wzorców projektowych: *Obserwator*, *Dekorator* i *Fabryka*.

5.3 Diagram aktywności

Przepływ pracy w systemie rozpoczyna się od wyboru rodzaju wzorca projektowego, dla którego kod ma być wygenerowany. W przypadku wskazania klasy bazowej, na podstawie której generowanie ma się odbywać wyświetlone zostanie *setter GUI*. Istnieje możliwość selektywnego wyboru pól lub metod przez użytkownika na potrzeby dalszego przetwarzania. W przypadku anulowania operacji następuje wyjście z systemu. W przypadku akceptacji wyświetlone zostanie *main GUI* z możliwością wpisywania ustawień konfigurujących proces generowania plików wzorca. W przypadku anulowania procesu następuje wyjście z systemu. W przypadku akceptacji wpisane dane podlegają walidacji. W razie błędnej walidacji użytkownik jest informowany, a *main GUI* wyświetlane jest ponownie w celu naniesienia korekty. W przypadku poprawnej walidacji ustalana jest ścieżka do generowania plików i nowe pliki zostaną utworzone. Po utworzeniu ich zawartość zostanie uzupełniona kodem wybranego wzorca projektowego zależnie od podanych w *GUI* kryteriów. Na końcu użytkownik jest informowany o sukcesie procesu. Następuje koniec przepływu sterowania i wyjście w system.

Diagram aktywności zaprezentowany na *Rysunku 11* opisano na podstawie wzorca projektowego *Dekorator*. Diagram aktywności dla wzorca projektowego *Obserwator* jest analogiczny, natomiast dla wzorca *Adapter* różnica polega na wczytywaniu metod zamiast pól w *setter GUI*. W przypadku wzorca projektowego *Fabryka* należy całkowicie pominąć elementy na diagramie

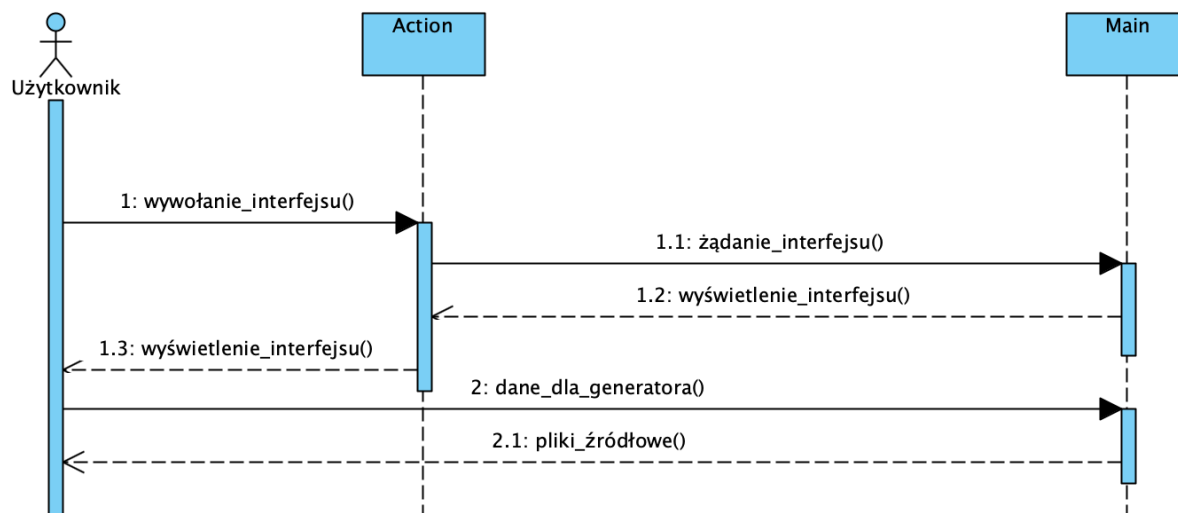
odpowiadające i powiązane z generowaniem *setter GUI*. Pozostały przepływ pracy jest analogiczny do przedstawionego na diagramie.



Rysunek 11. Diagram aktywności dla przepływu pracy na przykładzie wzorca projektowego Dekorator. Źródło: opracowanie własne.

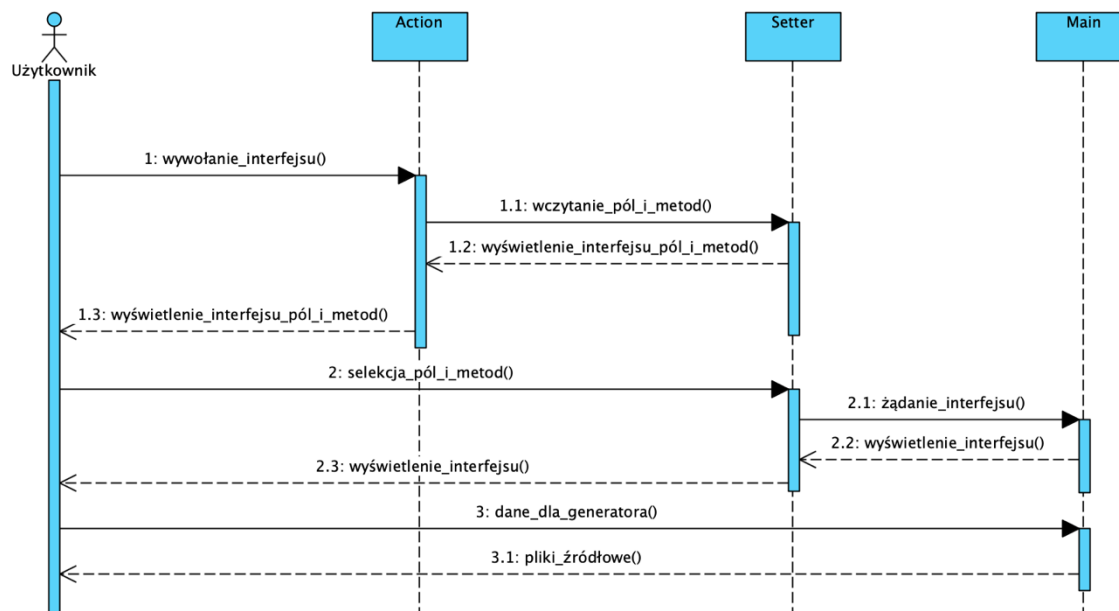
5.4 Diagram sekwencji

Na diagramie sekwencji opisano przepływ wiadomości przekazywanych pomiędzy klasami systemu. Diagram sekwencji zawiera charakterystyczną dla siebie linię czasu. Dzięki temu możliwa staje się wizualizacja przepływu komunikatów w czasie.



Rysunek 12. Diagram sekwencji dla przepływu informacji w systemie bez udziału klasy typu Setter. Źródło: opracowanie własne.

Na Rysunku 12 zobrazowano diagram sekwencji bez udziału klasy wczytującej pola i metody z klasy bazowej. Klasa bazowa stanowi punkt wejścia do systemu generatora. Na podstawie wczytanych pól i metod zmienia się sposób generowania plików źródłowych.



Rysunek 13. Diagram sekwencji dla przepływu informacji w systemie z udziałem klasy typu Setter. Źródło: opracowanie własne.

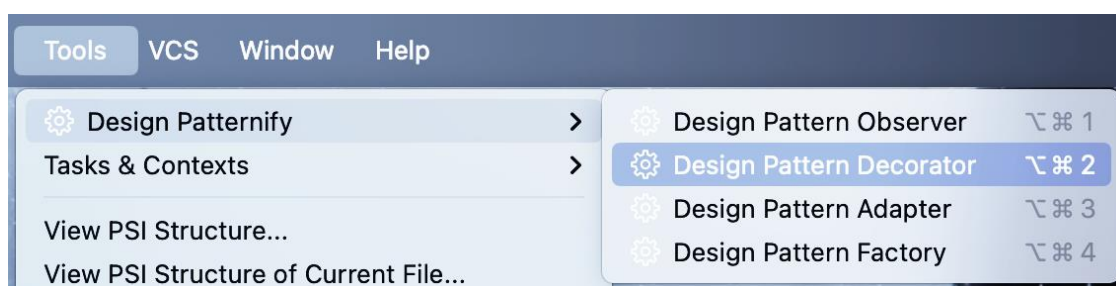
Generowanie plików bez udziału klasy bazowej polega na wysłaniu żądania o wyświetlenie graficznego interfejsu dla użytkownika. Interfejs referuje do wybranego przez użytkownika wzorca projektowego. Żądanie o wyświetlenie interfejsu przechwytuje klasa typu *Action*. Następuje przekazanie żądania do klasy typu *Main*. Klasa typu *Main* generuje interfejs dla użytkownika. Użytkownik wypełnia interfejs i wysyła jako odpowiedź do klasy *Main*. Na podstawie podanych w formularzu GUI danych następuje generowanie odpowiednich plików źródłowych. Wygenerowanie plików kończy przepływ sterowania w systemie. Przepływ informacji w systemie odbywa się w sposób synchroniczny.

Na *rysunku 13* zaprezentowano diagram sekwencji opisujący proces generowania plików źródłowych z udziałem klasy bazowej. Pokazano mechanizm wczytywania pól i metod z klasy bazowej. Proces wczytywania deklaracji z klasy bazowej został umiejscowiony w czasie jako początkowy. Po żądaniu wyświetlenia *GUI* przez użytkownika, następuje przekazanie informacji do klasy typu *Action*. Klasa typu *Action* wczytuje wszystkie zadeklarowane w klasie bazowej pola i metody i przekazuje je do klasy typu *Setter*. Klasa *Setter* zwraca na podstawie otrzymanych danych formularz umieszczony w *GUI* użytkownikowi. Użytkownik dokonuje selekcji pól i metod oraz zatwierdza formularz z wyborem. Dane z formularza zostają przekazane do klasy typu *Setter*. Na podstawie otrzymanych danych klasa generuje żądanie o formularz do klasy typu *Main*. Formularz dotyczy danych wymaganych do procesu generowania plików z kodem źródłowym. Interfejs z formularzem zostaje wygenerowany użytkownikowi. Użytkownik wypełnia formularz i po zatwierdzeniu nowe dane zostaną wysłane do klasy typu *Main*. Klasa typu *Main* przetwarza otrzymane od użytkownika informacje. Na podstawie otrzymanych danych klasa *Main* generuje odpowiednie pliki z kodem źródłowym dla użytkownika. Przepływ informacji w systemie odbywa się w sposób synchroniczny. Wygenerowanie plików kończy przepływ sterowania w systemie.

6 Przypadki użycia

W niniejszym rozdziale prezentowanej pracy zawarto opis funkcjonalności oferowanej przez prototyp oraz przykładowe przypadki użycia. Prezentowane przypadki użycia odnoszą się do przepływu niektórych informacji pomiędzy fabryką pojazdów, a salonami ich sprzedaży w celu zapewnienia skutecznej współpracy i koordynacji prac obu jednostek oraz w celu spełnienia wymagań klienta zainteresowanego ofertą sprzedaży pojazdów.

Aby wywołać proces generowania wybranego wzorca w programie *IntelliJ IDEA* należy skorzystać w menu Narzędzia, następnie wybrać menu wtyczki i wybrać wzorzec. Możliwe jest także skorzystanie ze skrótu *alt + ctrl + 1 ... 4*, lub *option + cmd + 1 ... 4*, w zależności od rodzaju wzorca, który ma zostać wybrany. *Rysunek 14* przedstawia wywołane menu wtyczki z narzędzi programu.



Rysunek 14. Menu wtyczki z możliwością wyboru wzorca projektowego do generowania. Źródło: opracowanie własne.

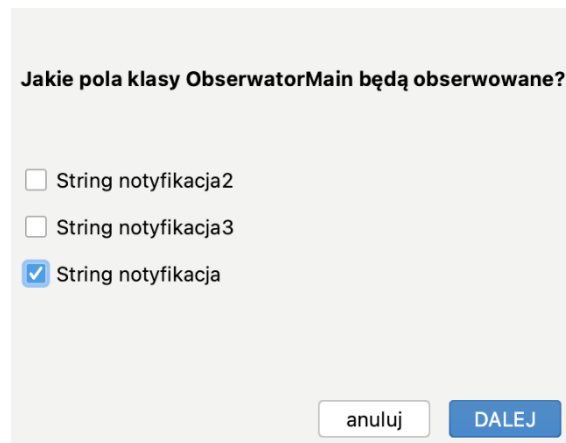
6.1 Obserwator

Fabryka pojazdów często wysyła zaktualizowane informacje do wybranych salonów. Salon musi zdecydować, czy chce dołączyć do wybranej grupy odbiorców otrzymującej charakterystyczne dla niej informacje z fabryki. Salon ma możliwość wypisania się z grupy odbiorców informacji z fabryki. Fabryka wysyłając nową informację kieruje ją do wszystkich salonów zapisanych na liście odbiorców.

W celu zapewnienia działania opisanego w tym przypadku użycia mechanizmu przepływu informacji pomiędzy fabryką, a salonami sprzedaży aut zastosowano wzorzec projektowy Obserwator.

6.1.1 Konfiguracja

Klasa *ObserwatorMain* zawiera implementacje trzech zmiennych z informacjami jakie fabryka aut chce przekazać do salonów. Po uruchomieniu konfiguratora wtyczki dla wzorca projektowego *Obserwator* przedstawionego na *Rysunku 15* ze wskazaniem klasy *ObserwatorMain* jako klasy bazowej zostaną wychwycone trzy zadeklarowane zmienne.



Rysunek 15. Okno dialogowe z wyborem wczytanych z klasy zmiennych dla wzorca projektowego *Obserwator*. Źródło: opracowanie własne.

Wybór potrzebnych zmiennych i zatwierdzenie okna dialogowego spowodują wyświetlenie kolejnego okna dialogowego. Nowe okno to konfigurator dla procesu generowania wzorca projektowego *Obserwator*.

W oknie konfiguratora wybrano polecenie określenia pakietu źródłowego dla generowania plików. Po otwarciu nowego okna z menu katalogów w ścieżce źródłowej projektu utworzono nowy katalog o nazwie *obserwator*. Wskazano katalog i zatwierdzono wybór. Efektem potwierdzenia operacji był powrót do okna dialogowego konfiguratora i wyświetlenie notyfikacji z potwierdzeniem wyboru ścieżki źródłowej przedstawionej na *Rysunku 16*.

i DESIGN PATTERNIFY
Wybrano pakiet: obserwator

Rysunek 16. Notyfikacja potwierdzająca wybór ścieżki do generowania plików wzorca projektowego *Obserwator*. Źródło: opracowanie własne.

Listing 13 Wyrażenie lambda z logiką wyboru ścieżki dla plików przed ich wygenerowaniem.

```
return panel {
    row("Gdzie umieścić kod wzorca?") {
        label(">> Wybierz pakiet źródłowy:").bold().align(AlignX.RIGHT)
        button(">> Wybór Pakietu <<", { _ ->
            val descriptor = FileChooserDescriptor(
                false,
                true,
                false,
                false,
                false,
                false)
            var path: String
            try {
                path = FileChooser.chooseFile(descriptor, e.project!!,
root)!!.path
            } catch (ex: Exception) {
                path = ""
            }
        })
    }
}
```

```

        okPathClick = false
    }
    if(path != "") runCancel = true
    val start = path.indexOf("src/")
    if(start == -1) rootPath = ""
    val end = path.length
    if(start != -1) rootPath = path.substring(start + 4, end)
    if (rootPath.contains("/")) {
        rootPath = rootPath.replace("/", ".", true)
    }
    sourcePath = rootPath
    if(rootPath != "") notificationInfo(e.project!!, "Wybrano
pakiet: " + rootPath)
    }).align(AlignX.FILL)
}

```

Na *Listingu 13* pokazano implementację okna dialogowego odpowiedzialnego za wyświetlenie GUI ze strukturą katalogów. W oknie możliwy jest wybór katalogu, do którego zostaną wygenerowane pliki wzorca. Logika pokazuje wczytanie i formatowanie wybranej ścieżki wewnątrz wyrażenia lambda, które zostanie zwrócone w wyniku zdarzenia kliknięcia przycisku typu *button*.

Rysunek 17. Wypełnione okno dialogowe konfiguratora dla wzorca projektowego Obserwator. Źródło: opracowanie własne.

Po powrocie do okna dialogowego konfiguratora przedstawionego na *Rysunku 17* wpisano nazwę klasy nadawcy informacji. W prezentowanym przypadku użycia nadano nazwę *Fabryka*. Kolejne pole tekstowe wymagające wypełnienia określa nazwy klas odbiorców. Na potrzeby prezentowanego przykładu nadano nazwy: *SalonWarszawa*, *SalonGdansk*, *SalonKrakow*. Ostatnie pole tekstowe określa zmienne w klasach, które mają podlegać obserwacji i być propagowane od odbiorcy do nadawcy. W prezentowanym przykładzie obserwowane jest już pole notyfikacja, którego wybór

przedstawiono na *Rysunku 17*. W przykładzie w oknie dialogowym konfiguratora dodano jeszcze dwa dodatkowe pola: *kosztorys* i *gwarancja*.

Zatwierdzenie okna dialogowego konfiguratora przyciskiem *generuj* uruchomi proces generowania wymaganych plików z kodem źródłowym.

6.1.2 Generator

Powodzenie operacji generowania plików z kodem źródłowym zostanie potwierdzone notyfikacją przedstawioną na *Rysunku 18*.

 **DESIGN PATTERNIFY**
Dodano pliki do projektu.

*Rysunek 18. Notyfikacja potwierdzająca dodanie plików do bieżącego projektu.
Źródło: opracowanie własne.*

W wyniku działania generatora kodu dla wzorca projektowego Obserwator w podanej konfiguracji opisanej w *Rozdziale 6.1.1*, zostało utworzone sześć plików z kodem źródłowym w języku *Java*.

Listing 14. Kod źródłowy klasy Fabryka wygenerowany w języku Java dla wzorca projektowego Obserwator.

```
package obserwator;

import java.util.ArrayList;
import java.util.List;

public class Fabryka implements IPublisher {
    private List<ISubscriber> subscribers;
    String kosztorys;
    String gwarancja;
    String notyfikacja;

    public Fabryka() {
        subscribers = new ArrayList<ISubscriber>();
    }

    @Override
    public void subscribe(ISubscriber s) {
        subscribers.add(s);
    }

    @Override
    public void unsubscribe(ISubscriber s) {
        subscribers.remove(s);
    }

    @Override
    public void notification() {
        for(ISubscriber subscriber : subscribers) {
```

```

        subscriber.info(kosztorys, gwarancja, notyfikacja);
    }
}

public void pushNotification(String kosztorys, String gwarancja,
String notyfikacja) {
    this.kosztorys = kosztorys;
    this.gwarancja = gwarancja;
    this.notyfikacja = notyfikacja;

    notification();
}
}

```

Klasę *Fabryka* przedstawiono na *Listingu 14*. Implementacja zawiera zależności w postaci pól typu *String* o nazwie: *notyfikacja*, *kosztorys* i *gwarancja*. Klasa posiada zaimplementowane metody do dodawania i usuwania subskrybentów oraz metodę do wysyłania informacji do subskrybentów w oparciu o zaimplementowane zależności.

Listing 15 pokazuje implementację interfejsów składających się na kompletną strukturę wzorca projektowego *Obserwator*

Listing 15. Kod źródłowy interfejsów składających się na wzorec projektowy Obserwator.

```

package obserwator;

public interface IPublisher {
    public void subscribe(ISubscriber s);
    public void unsubscribe(ISubscriber s);
    public void notification();
}

package obserwator;

public interface ISubscriber {
    public void info(String kosztorys, String gwarancja, String
notyfikacja);
}

```

Klasy będące odbiorcami wiadomości przekazywanych przez klasę nadawcę, posiadają zaimplementowane takie same jak nadawca pola typu *String*. Dodatkowo posiadają metodę przechwytyjącą zdarzenie wysłania komunikatu przez nadawcę oraz dane komunikatu. Wygenerowaną implementację klasy *SalonKrakow* przedstawiono na *Listingu 16*.

Listing 16. Kod źródłowy klasy SalonKrakow wygenerowany w języku Java dla wzorca projektowego Obserwator.

```

package obserwator;

public class SalonKrakow implements ISubscriber {
    String kosztorys;
    String gwarancja;
    String notyfikacja;

    private Fabryka fabryka;
}

```

```

public SalonKrakow(Fabryka fabryka) {
    this.fabryka = fabryka;
    fabryka.subscribe(this);
}

public void info(String kosztorys, String gwarancja, String
notyfikacja) {
    this.kosztorys = kosztorys;
    this.gwarancja = gwarancja;
    this.notyfikacja = notyfikacja;
}
}

```

Kod źródłowy klas *WarszawaSalon* i *GdanskSalon* jest analogiczny do kodu klasy *SalonKrakow*, dlatego został pominięty. Te klasy stanowią pozostałych odbiorców wiadomości.

6.1.3 Integracja

Wygenerowany szablon wzorca projektowego *Obserwator* należy zintegrować z już istniejącą implantacją i uruchomić. W tym celu do klasy *ObserwatorMain* przedstawionej na *Listingu 17* dodano brakujące pola typu *String*: *kosztorys* i *gwarancja*. Utworzono obiekt typu *Fabryka* i trzy obiekty reprezentujące salony. Powiązano obiekty salonów z obiektem fabryki i wykonano metodę notyfikacji salonów przez obiekt fabryki z ustalonymi parametrami.

Listing 17. Kod źródłowy klasy ObserwatorMain odpowiedzialnej za konfigurację obiektów wzorca projektowego Obserwator.

```

package obserwator;

public class ObserwatorMain {

    String notyfikacja = "Fabryka informuje o produkcji nowego modelu
auta";
    String notyfikacja2 = "notyfikacja nr 2";
    String notyfikacja3 = "notyfikacja nr 3";

    Fabryka fabryka = new Fabryka();

    SalonWarszawa salonWarszawa = new SalonWarszawa(fabryka);
    SalonGdansk salonGdansk = new SalonGdansk(fabryka);
    SalonKrakow salonKrakow = new SalonKrakow(fabryka);

    String kosztorys = "kosztorys";
    String gwarancja = "gwarancja";

    public void start() {
        fabryka.pushNotification(kosztorys, gwarancja, notyfikacja);
    }

}

```

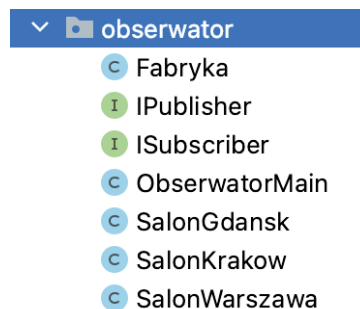
W metodzie *info(...)* pokazanej na *Listingu 18*, zawartej w klasach: *SalonWarszawa*, *SalonGdansk* i *SalonKrakow* należy dopisać logikę przetwarzania informacji otrzymanej od obiektu typu *Fabryka*. Na potrzeby prezentacji do klas reprezentujących salony dodano logikę wypisania otrzymanych danych na standardowym wyjściu.

W wyniku wywołania metody *start()* na rzecz obiektu typu *ObserwatorMain* w terminalu pokazanym na *Rysunku 19* zostaną wypisane przetworzone przez odbiorców dane.

```
Salon Warszawa: Fabryka informuje o produkcji nowego modelu auta, dołączono: kosztorys, gwarancja
Salon Gdansk: Fabryka informuje o produkcji nowego modelu auta, dołączono: kosztorys, gwarancja
Salon Krakow: Fabryka informuje o produkcji nowego modelu auta, dołączono: kosztorys, gwarancja
```

Rysunek 19. Dane wyświetlone w terminalu IDE po uruchomieniu kodu wzorca projektowego Obserwator. Źródło: opracowanie własne.

Na całość rozwiązania w prezentowanym przykładzie składa się siedem plików z kodem w języku *Java* przedstawionych na *Rysunku 20*, jednak liczba plików możliwych do wygenerowania nie jest niczym z góry ograniczona i zależy od wymaganej konfiguracji.



Rysunek 20. Pakiet z plikami źródłowymi w języku Java dla wzorca projektowego Obserwator. Źródło: opracowanie własne.

*Listing 18. Metoda *info(...)* z klasy *SalonGdansk* z dopisaną logiką przetwarzania otrzymywanych danych.*

```
public void info(String kosztorys, String gwarancja, String
notyfikacja) {
    this.kosztorys = kosztorys;
    this.gwarancja = gwarancja;
    this.notyfikacja = notyfikacja;

    System.out.println("Salon Gdansk: " + notyfikacja + ", dołączono: "
+ kosztorys + ", " + gwarancja);
}
```

6.2 Dekorator

Salony sprzedaży pojazdów umożliwią klientom konfigurowanie aut według własnych kryteriów w celu dopasowania oferty do potrzeb klienta. Własna konfiguracja auta składa się z kolejnych, występujących po sobie etapów. W pierwszej kolejności wybierana jest podstawowa wersja pojazdu. Kolejnym etapem może być wybór systemów elektronicznych wspierających jazdę pojazdem. Taka konfiguracja, może zostać rozszerzona o rodzaj materiałów wykorzystanych we wnętrzu auta podczas procesu produkcji. Końcowym etapem konfiguracji może być wybór metalizowanego lakieru

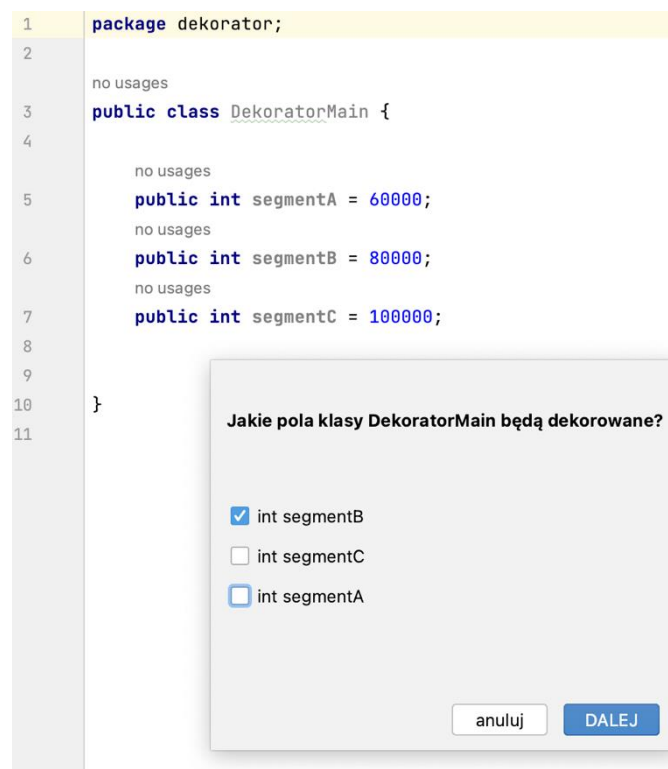
o rzadko występującym kolorze. Każda wprowadzona modyfikacja niesie ze sobą określone koszty dla klienta.

W celu wdrożenia opisanego mechanizmu stopniowego uzupełniania konfiguracji pojazdu o nowe funkcjonalności zastosowano wzorec projektowy *Dekorator*.

6.2.1 Konfiguracja

Klasa *DekoratorMain* posiada zaimplementowane trzy pola typu *Integer* z przypisaną wartością. Przypisane wartości odnoszą się do bazowej ceny modelu auta. Różnica w cenie polega na odmiennych modelach aut należących do różnych typów segmentów. Wybrany segment auta wraz z jego ceną stanowi punkt wejścia dla dalszej konfiguracji podzespołów i cech auta przez użytkownika zgodnie z jego wymaganiami.

W wyniku wywołania konfiguratora dla wzorca projektowego *Dekorator* wyświetlono okno dialogowe przedstawione na *Rysunku 21*. Punkt wejścia dla procesu inicjalizacji stanowi wczytanie zadeklarowanych pól w klasie *DekoratorMain*. Klasa zawiera implementacją pól z przypisanymi wartościami wejściowymi dla procesu generowania kodu. Okno dialogowe umożliwia wybór zmiennych stanowiących dane wejściowe dla logiki generatora wzorca *Dekorator*.



Rysunek 21. Początkowa implementacja klasy *DekoratorMain* z deklaracją pól wychwyconych przez wtyczkę w celu selektywnego wyboru. Źródło: opracowanie własne.

W wyniku zatwierdzenia okna dialogowego z wybraną zmienną nastąpi wyświetlenie kolejnego okna dialogowego z konfiguracją dla generatora plików.

W konfiguratorze pokazanym na *Rysunku 22* wybrano istniejący pakiet źródłowy *dekorator* dla generowania nowych plików. W polu tekstowym określającym liczbę i rodzaj warstw wyspecyfikowano trzy warstwy: elektronika, materiały i lakier. Każda z kolejnych warstw stanowi we wzorcu *Dekorator* otoczenie dla poprzednich tworząc warstwową strukturę. Kolejne warstwy zawierają w sobie wszystkie poprzednie, jednak komunikacją zachodzi tylko pomiędzy sąsiednimi warstwami.

Dla każdej z warstw określono różne klasy wchodzące w skład każdej warstwy:

- dla warstwy elektronika określono klasy: *Kamera*, *LED-y*, *ESR*,
- dla warstwy materiał określono klasy: *Skora*, *ECOSkora*, *Syntetyk*,
- dla warstwy lakier określono klasy: *Metalik*, *Mat*.

Kolejne pole tekstowe stanowi grupa wspólnych dla każdej warstwy zmiennych. Wspólne zmienne stanowią pola w klasie, które będą podlegały każdorazowo modyfikacji wewnątrz każdej warstwy, podczas przepływu sterowania z warstwy do warstwy. W prezentowanym przykładzie zmienną, która będzie podlegała modyfikacji wczytano z klasy *DekoratorMain* przed wyświetleniem okna konfiguratora. Wczytane pole stanowi wspólną, dekorowaną przez każdą z warstw zmienną. Pole tekstowe z deklaracją dodatkowych, dekorowanych zmiennych pozostawiono puste zgodnie z wymaganiami w prezentowanym przykładzie. Możliwe jest odmienne zachowanie polegające na wczytaniu zmiennych z klasy wejściowej i dodanie kolejnych zmiennych w oknie konfiguratora. Wybór konfiguracji zależy od specyfiki projektu i wymagań logiki wzorca, który chce się zintegrować z projektem.

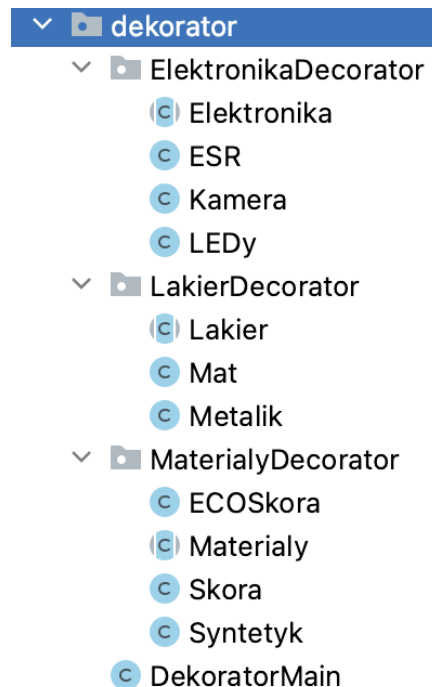
Ostatnie pole tekstowe stanowią metody, które mają być wspólne dla każdej z warstw wzorca *Dekorator*. W prezentowanym przykładzie podano metodę, która będzie wyświetlała bieżącą cenę po wybraniu dodatków.

Gdzie umieścić kod wzorca?	>> Wybierz pakiet źródłowy:	>> Wybór Pakietu <<
Jakie mają być kolejne warstwy?	>> Podaj nazwy warstw:	Elektronika Materiały Lakier
Jakie dekoratory umieścić w kolejnych warstwach?	>> Podaj nazwy klas:	Kamera LEDy ESR Skora ECOSkora Syntetyk Metalik Mat
Jakie pola mają być dekorowane?	>> Podaj nazwy zmiennych z typem:	
Jakie metody mają być wspólne?	>> Podaj deklaracje metod:	void wyswietlCene()

Rysunek 22. Wypełnione okno dialogowe konfiguratora dla wzorca projektowego *Dekorator*. Źródło: opracowanie własne.

6.2.2 Generator

W wyniku działania generatora wzorca projektowego *Dekorator* powstały trzy pakiety z klasami pokazane na *Rysunku 23*. Dwa pierwsze pakiety zawierają po trzy klasy, natomiast trzeci pakiet zawiera dwie klasy.



Rysunek 23. Struktura pakietów i klas dla wzorca projektowego *Dekorator*. Źródło: opracowanie własne.

Pierwszą i najbardziej wewnętrzną warstwę we wzorcu *Dekorator* stanowi implementacja klas w pakiecie *elektronikaDecorator*. Pakiet zawiera jedną klasę abstrakcyjną i trzy klasy dekoratorów o analogicznej implementacji różniące się nazwą klasy. W argumencie konstruktora zawarta jest referencja do klasy z wewnętrznej warstwy, stanowiącego punkt wejścia dla przetwarzania danych. *Listing 19* pokazuje implementację klasy abstrakcyjnej *Elektronika* należącej do najbardziej wewnętrznej warstwy budującej strukturę wzorca *Dekorator*.

Listing 19. Kod źródłowy klasy abstrakcyjnej najbardziej wewnętrznej warstwy składających się na wzorec projektowy *Dekorator*.

```
package dekorator.ElektronikaDecorator;

public abstract class Elektronika {
    public int segmentB;

    public abstract void wyswietlCene();
}
```

Listing 20 pokazuje implementację klasy *LED-y* rozszerzającej klasę abstrakcyjną *Elektronika* i należącą do najbardziej wewnętrznej warstwy budującej strukturę wzorca *Dekorator*.

Listing 20. Kod źródłowy klasy LED-y wygenerowany w języku Java dla wzorca projektowego Dekorator.

```
package dekorator.ElektronikaDecorator;

import dekorator.ElektronikaDecorator.Elektronika;

public class LEDy extends Elektronika{
    public LEDy (DekoratorMain dekoratormain) {

        this.segmentB = dekoratormain.segmentB;

    }

    public void wyswietlCene() { }

}
```

Środkową warstwę we wzorcu Dekorator stanowi implementacja zawarta w klasach pakietu *materialyDecorator*. Implementację klas w pakiecie odróżnia inna nazwa dla każdej klasy. Listing 21 pokazuje implementację klasy abstrakcyjnej *Materialy* należącej do środkowej warstwy budującej strukturę wzorca *Dekorator*. Klasa abstrakcyjna dziedziczy po klasie *Elektronika*.

Listing 21. Kod źródłowy klasy abstrakcyjnej środkowej warstwy składającej się na wzorec projektowy Dekorator.

```
package dekorator.MaterialyDecorator;

import dekorator.ElektronikaDecorator.Elektronika;

public abstract class Materialy extends Elektronika{
    public Elektronika elektronika;

    public int segmentB;

    public abstract void wyswietlCene();

}
```

Listing 22. Kod źródłowy klasy Syntetyk wygenerowany w języku Java dla wzorca projektowego Dekorator.

```
package dekorator.MaterialyDecorator;

import dekorator.ElektronikaDecorator.Elektronika;
import dekorator.MaterialyDecorator.Materialy;

public class Syntetyk extends Materialy{
    public Syntetyk (Elektronika elektronika) {
        this.elektronika = elektronika;
    }
}
```

```
public void wyswietlCene() { }
```

Listing 22 pokazuje implementację klasy *Syntetyk* rozszerzającej klasę abstrakcyjną *Materialy* i należącej do środkowej warstwy budującej strukturę wzorca *Dekorator*.

Do ostatniej i najbardziej zewnętrznej warstwy w strukturze wzorca projektowego *Dekorator* należy dołączyć implementacje klas w pakiecie *lakierDecorator*. Na pakiet składa się jedna klasa abstrakcyjna i dwie klasy, których implementacje różnią się tylko nazwą klasy. *Listing 23* pokazuje implementację abstrakcyjnej klasy *Lakier* dziedziczącej po klasie *Materialy*.

Listing 23. Kod źródłowy klasy abstrakcyjnej najbardziej zewnętrznej warstwy składającej się na wzorzec projektowy *Dekorator*.

```
package dekorator.LakierDecorator;

import dekorator.MaterialyDecorator.Materialy;

public abstract class Lakier extends Materialy{
    public Materialy materialy;

    public int segmentB;

    public abstract void wyswietlCene();
}
```

Listing 24. Kod źródłowy klasy *Metalik* wygenerowany w języku Java dla wzorca projektowego *Dekorator*.

```
package dekorator.LakierDecorator;

import dekorator.MaterialyDecorator.Materialy;
import dekorator.LakierDecorator.Lakier;

public class Metalik extends Lakier{
    public Metalik (Materialy materialy) {
        this.materialy = materialy;
    }

    public void wyswietlCene() { }
}
```

Listing 24 pokazuje implementację klasy *Metalik* rozszerzającej klasę abstrakcyjną *Lakier* i należącej do zewnętrznej warstwy budującej strukturę wzorca *Dekorator*.

Powiązanie wygenerowanych klas pomiędzy sobą w strukturę warstwową następuje w konstruktorze każdej klasy.

6.2.3 Integracja

Proces integracji wygenerowanego kodu z istniejącą implementacją rozpoczyna się od importu brakujących referencji w najbardziej wewnętrznej warstwie. Zaimportowane klasy należą do zewnętrznych implementacji względem wygenerowanego kodu wzorca. Stanowią punkt wejścia dla procesu inicjalizacji logiki wzorca. Odnoszą się do najbardziej wewnętrznej warstwy wzorca.

W pierwszej warstwie stanowiącej klasę *LED-y* przedstawioną na *Listingu 25* w metodzie *wyswietlCene()* zaimplementowano logikę wyświetlania bieżącej ceny auta w klasie *LED-y*. Cena całkowita stanowi sumę ceny auta w klasie *B* oraz ceny dodatkowego oświetlenia *LED*. Logikę zwiększenia ceny umieszczono w konstruktorze.

Listing 25. Uzupełniona metoda i konstruktor w klasie LED-y odpowiedzialnej za wyświetlanie ceny auta z dodatkiem LED.

```
package dekorator.ElektronikaDecorator;

import dekorator.DekoratorMain;
import dekorator.ElektronikaDecorator.Elektronika;

public class LEDy extends Elektronika{
    public LEDy (DekoratorMain dekoratormain) {

        this.segmentB = dekoratormain.segmentB;
        this.segmentB += 10000;

    }

    public void wyswietlCene() {
        System.out.println("Cena auta segmentu B + LEDy = " +
this.segmentB);
    }

}
```

W klasie *Syntetyk* przedstawionej na *Listingu 26* pobierającej w konstruktorze referencję do klasy *Elektronika* uzupełniono konstruktor logiką aktualizacji ceny za materiał syntetyczny w aucie z oświetleniem *LED*. W dalszym kroku uzupełniono metodę wypisującą bieżącą cenę auta.

W klasie *Metalik* uzupełniono implementację w sposób analogiczny jak w klasie *Syntetyk* z *Listingu 26*.

Listing 26. Uzupełniona metoda i konstruktor w klasie Syntetyk odpowiedzialnej za wyświetlenie ceny auta z dodatkiem LED i materiałem syntetycznym.

```
package dekorator.MaterialyDecorator;

import dekorator.ElektronikaDecorator.Elektronika;
import dekorator.MaterialyDecorator.Materialy;

public class Syntetyk extends Materialy{
    public Syntetyk (Elektronika elektronika) {
        this.elektronika = elektronika;
        this.segmentB = this.elektronika.segmentB + 12000;
    }

    public void wyswietlCene() {
        System.out.println("Cena auta segmentu B + LEDy + syntetyk = " +
this.segmentB);
    }
}
```

Listing 27. Kod klasy DekoratorMain odpowiedzialnej za powiązanie obiektów dekorowanych i dekoratorów.

```
package dekorator;

import dekorator.ElektronikaDecorator.LEDy;
import dekorator.LakierDecorator.Metalik;
import dekorator.MaterialyDecorator.Syntetyk;

public class DekoratorMain {

    public int segmentA = 60000;
    public int segmentB = 80000;
    public int segmentC = 100000;

    public void start() {

        LEDy ledy = new LEDy(this);
        ledy.wyswietlCene();

        Syntetyk syntetyk = new Syntetyk(ledy);
        syntetyk.wyswietlCene();

        Metalik metalik = new Metalik(syntetyk);
        metalik.wyswietlCene();
    }
}
```

Implementację wzorca wdrożono w klasie *DekoratorMain* pokazanej na *Listingu 27*. Zgodnie z logiką wzorca obiekty reprezentujące wewnętrzne warstwy przekazano do warstw zewnętrznych.

Warstwy wewnętrzne stanowią obiekty dekorowane, zaś warstwy zewnętrzne względem nich dekoratory. W każdej warstwie wyświetlono aktualną cenę pojazdu.

Rysunek 24 przedstawia wynik wyświetlony w terminalu *IDE* po uruchomieniu skonfigurowanej implementacji wzorca projektowego *Dekorator*.

```
Cena auta segmentu B + LEDy = 90000
Cena auta segmentu B + LEDy + syntetyk = 102000
Cena auta segmentu B + LEDy + syntetyk + metalik = 111000
```

Rysunek 24. Dane wyświetlone w terminalu *IDE* po uruchomieniu kodu wzorca projektowego *Dekorator*. Źródło: opracowanie własne.

6.3 Adapter

Komunikacja pomiędzy fabryką pojazdów, a salonami sprzedaży jest uwarunkowana formatem przesyłanych danych. Fabryka pojazdów generuje dane w formacie *XML* i w takiej formie je wysyła. Salony sprzedaży pojazdów mają mocno zróżnicowane systemy odbioru informacji z fabryki. Część salonów wykorzystuje format *XML*, jednak pozostałe salony używają formatu *JSON* lub *CSV*. Fabryka pojazdów nie ma możliwości przesłania informacji do salonu, który wspiera format *JSON* lub *CSV* w sposób bezpośredni.

W celu zapewnienia komunikacji pomiędzy fabryką pojazdów, a wszystkimi rodzajami salonów sprzedaży, niezależnie od wspieranego przez nie interfejsu do odbioru informacji wykorzystano wzorec projektowy *Adapter*.

6.3.1 Konfiguracja

Punktem wejścia dla wzorca *Adapter* jest klasa *FabrykaXML* przedstawiona na *Listingu 28*, która zapewnia możliwość pobrania wymaganych przez salon samochodowy danych w formacie *XML*.

Listing 28. Kod klasy *FabrykaXML* odpowiedzialnej za wysyłanie danych w formacie *XML* do salonów samochodowych.

```
package adapter;


public class FabrykaXML {
    String xml = "dane XML z fabryki";

    public String pobierzXML() {
        return xml;
    }

    public void ustawXML(String xml) {
        this.xml = xml;
    }
}
```


W celu adaptacji metody *pobierzXML()* do formatu umożliwiającego pobranie danych przez salony, które wspierają wyłącznie format *JSON* lub *CSV* uruchomiono konfigurator wzorca projektowego *Adapter*.

W oknie dialogowym wyboru metod pokazanym na *Rysunku 25* wybrano metodę, która zostanie poddana adaptacji. W prezentowanym przykładzie wybrano metodę odpowiedzialną za pobieranie danych w formacie *XML*.



Jakie metody klasy *FabrykaXML* będą adaptowane?


☐ `public void ustawXML(String xml)`

☒ `public String pobierzXML()`

anuluj DALEJ

Rysunek 25. Okno dialogowe z wyborem metod, które mają zostać poddane adaptacji we wzorcu projektowym *Adapter*. Źródło: opracowanie własne.

W wyniku wyboru potrzebnych metod wyświetlone zostanie drugie okno dialogowe konfiguratora pokazane na *Rysunku 26*. W oknie wyboru pakietu źródłowego dla plików wzorca *Adapter* wybrano pakiet *adapter*. W pakiecie istnieje już klasa *FabrykaXML*. Jako nazwę interfejsu dla adaptera fabryki wysyłającego dane w formacie *JSON* wybrano nazwę *IFabrykaJSON*. Dla klasy adaptera fabryki wysyłającej dane w formacie *JSON* wybrano nazwę *FabrykaJSON*. Na nazwę metody adaptera, która ma adaptować metodę *pobierzXML()* z klasy *FabrykaXML* wybrano deklarację *String pobierzJSON()*.



Gdzie umieścić kod wzorca?	>> Wybierz pakiet źródłowy:	>> Wybór Pakietu <<
Jaka ma się nazywać interfejs adaptera?	>> Podaj nazwę interfejsu:	IFabrykaJSON
Jak ma się nazywać adapter?	>> Podaj nazwę klasy:	FabrykaJSON
Jaka metoda ma być adapterem dla:	>> String pobierzXML() >>	String pobierzJSON()

anuluj GENERUJ

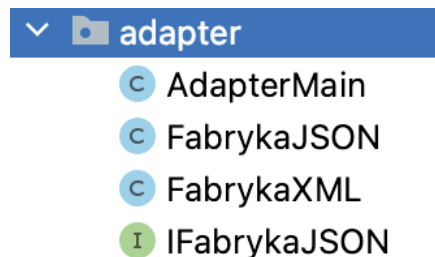
Rysunek 26. Wypełnione okno dialogowe konfiguratora dla wzorca projektowego *Adapter*. Źródło: opracowanie własne.

W wyniku zatwierdzenia konfiguracji przyciskiem *generuj* nastąpi uruchomienie generatora plików.

W celu zestawienia konfiguracji dla salonów samochodowych wymagających danych formatu *CSV* należy wpisać w pola tekstowe analogiczne wartości jak dla formatu *JSON*, ale referujące do formatu *CSV*.

6.3.2 Generator

Wygenerowana struktura katalogów dla plików projektu pokazana na *Rysunku 27* składa się z: klasy fabryki, klasy i interfejsu adaptera dla fabryki oraz klasy uruchomieniowej dla wzorca Adapter.



*Rysunek 27. Pakiet z plikami źródłowymi w języku Java dla wzorca projektowego Adapter.
Źródło: opracowanie własne.*

Klasa *FabrykaXML* przedstawiona na *Listingu 28* zawiera logikę odpowiedzialną za generowania informacji z formacie *XML* oraz możliwość pobierania tej informacji przez salony samochodowe.

Interfejs *IFabrykaJSON* zawiera deklarację metody *pobierzJSON()* adaptującej metodę *pobierzXML()* z klasy *FabrykaXML*.

Klasa *FabrykaJSON* pokazana na *Listingu 29* implementuje interfejs *IFabrykaJSON* oraz stanowi adapter dla salonów samochodowych umożliwiając pobranie danych z fabryki w formacie *JSON*.

Listing 29. Kod źródłowy klasy FabrykaJSON stanowiącej adapter łączący klasę FabrykaXML z salonami samochodowymi wymagającymi danych formatu JSON.

```
package adapter;

public class FabrykaJSON implements IFabrykaJSON {
    private FabrykaXML fabrykaxml;

    public FabrykaJSON(FabrykaXML fabrykaxml) {
        this.fabrykaxml = fabrykaxml;
    }

    @Override
    public String pobierzJSON() {
        return fabrykaxml.pobierzXML();
    }
}
```

6.3.3 Integracja

W celu integracji wygenerowanego kodu źródłowego dla wzorca projektowego *Adapter* z już istniejącą implementacją należy w pierwszej kolejności dostosować logikę klasy *FabrykaJSON*. Kolejnym krokiem jest stworzenie implementacji wdrażającej wzorzec projektowy.

Modyfikacja klasy *FabrykaJSON* pokazana na *Listingu 30* polega na implementacji logiki konwertującej format danych *XML* do formatu *JSON*. W prezentowym przykładzie modyfikację należy wprowadzić w metodzie *pobierzJSON()*.

Analogiczne modyfikacje należy zaimplementować w metodzie adaptera dla formatu *CSV*. Implementacja powinna być zgodna z logiką konwersji dla formatu *CSV*.

Listing 30. Modyfikacja metody pobierzJSON() w klasie FabrykaJSON zgodnie z logiką wzorca projektowego Adapter.

```
package adapter;

public class FabrykaJSON implements IFabrykaJSON {
    private FabrykaXML fabrykaxml;

    public FabrykaJSON(FabrykaXML fabrykaxml) {
        this.fabrykaxml = fabrykaxml;
    }

    @Override
    public String pobierzJSON() {
        fabrykaxml.ustawXML("dane JSON z fabryki");
        return fabrykaxml.pobierzXML();
    }
}
```

Działanie adaptera polega na przekazaniu obiektu, który ma być adaptowany do konstruktora obiektu adaptera. W prezentowanym przykładzie przekazano obiekt typu *FabrykaXML* do konstruktora obiektu typu *FabrykaJSON*. W kolejnym kroku wywołano metodę adaptera, która pobiera dane z fabryki w formacie *JSON* po konwersji z formatu *XML*. *Listing 31* prezentuje kod klasy *AdapterMain* odpowiedzialnej za uruchomienie logiki wzorca projektowego *Adapter*.

Listing 31. Kod klasy AdapterMain odpowiedzialnej za uruchomienie logiki wzorca projektowego Adapter.

```
package adapter;

public class AdapterMain {

    public void start() {
        FabrykaXML fabrykaXML = new FabrykaXML();
        System.out.println(fabrykaXML.pobierzXML());

        FabrykaJSON fabrykaJSON = new FabrykaJSON(fabrykaXML);
        System.out.println(fabrykaJSON.pobierzJSON());
    }
}
```

Analogiczną logikę należy zaimplementować w adapterze dla formatu *CSV*, aby otrzymać dane w tym formacie po procesie integracji.

W wyniku działania programu w terminalu środowiska uruchomieniowego pokazanym na *Rysunku 28* otrzymano wyniki działania integracji wzorca projektowego *Adapter*.

```
dane XML z fabryki
dane JSON z fabryki
```

Rysunek 28. Dane wyświetlone w terminalu IDE po uruchomieniu kodu wzorca projektowego Adapter. Źródło: opracowanie własne.

6.4 Fabryka

Fabryka pojazdów produkuje różne rodzaje silników spalinowych. Każdy silnik posiada swoją specyficzną konfigurację. Salon sprzedaży zainteresowany specyfikacją konkretnego rodzaju silnika powinien mieć możliwość wyboru takiego silnika wraz z dotyczącą go specyfikacją w celu zaprezentowania jego cech klientowi.

W celu zapewnienia szybkiej budowy reprezentacji wymaganego silnika oferowanego przez producenta spośród szerokiej gamy innych rodzajów silników, postanowiono skorzystać ze wzorca projektowego *Fabryka*.

6.4.1 Konfiguracja

Wzorzec projektowy *Fabryka* nie posiada wbudowanego mechanizmu wczytywania pól lub metod z klasy bazowej, z którą miałyby nastąpić integracja generowanego kodu z już istniejącym.

Gdzie umieścić kod wzorca?	>> Wybierz pakiet źródłowy:	>> Wybór Pakietu <<
Jakiego typu obiektów dotyczy fabryka?	>> Podaj nazwę interfejsu:	Silnik
Jakie metody ma mieć fabryka?	>> Podaj deklaracje metod:	void wyswietlMoc() void wyswietlPojemnosc()
Jakie obiekty ma produkować fabryka?	>> Podaj nazwy klas:	Diesel Benzyna Hybryda

anuluj GENERUJ

Rysunek 29. Wypełnione okno dialogowe konfiguratora dla wzorca projektowego Fabryka. Źródło: opracowanie własne.

Po etapie wywołania konfiguratora wzorca Fabryka pokazanego na *Rysunku 29* wyświetlone zostanie okno dialogowe z wyborem ścieżki. Do ścieżki nastąpi generowanie nowych plików. Należy podać nazwę interfejsu wspólnego dla wszystkich typów generowanych obiektów. Wymagana jest deklaracja metod wspólnych dla wszystkich typów silników. Ostatnie pole stanowi lista deklaracji typów silników, które będą generowane.

Uzupełniona i zatwierdzona przyciskiem *generuj* konfiguracja uruchomi proces generowania plików.

6.4.2 Generator

Wynikiem działania generatora są pliki źródłowe z kodem implementującym interfejs. Interfejs deklaruje metody wspólne dla klas należących do jednej rodziny. W prezentowym przykładzie klasy zgrupowane w jedną rodzinę stanowią o rodzaju silnika. Klasy reprezentujące rodzaje silników implementują ten sam interfejs, jednak logika zawarta w każdej metodzie pochodzącej z interfejsu jest odmienna dla każdej klasy i charakterystyczna dla niej. *Listing 32* prezentuje kod źródłowy klasy *Hybryda* implementującej interfejs *Silnik*, której implementacja jest zgodna z logiką wzorca *Fabryka*.

Listing 32. Kod klasy Hybryda implementującej w sposób zgodny z logiką wzorca Fabryka interfejs Silnik.

```
package fabryka;

public class Hybryda implements Silnik {

    @Override
    public void wyswietlMoc() {
        System.out.println("TODO Hybryda.wyswietlMoc()");
    }
    @Override
    public void wyswietlPojemnosc() {
        System.out.println("TODO Hybryda.wyswietlPojemnosc()");
    }
}
```

Listing 33. Implementacja klasy SilnikFactory odpowiedzialnej za generowania obiektów silników zgodnie z logiką wzorca projektowego Fabryka.

```
package fabryka;

public class SilnikFactory {

    public enum SilnikType {
        DIESEL, BENZYNA, HYBRYDA
    }

    public Silnik createSilnik(SilnikType type) {
        if (type == null) {
            return null;
        }
    }
}
```

```

        else if (type.equals(SilnikType.DIESEL)) {
            return new Diesel();
        } else if (type.equals(SilnikType.BENZYNNA)) {
            return new Benzyna();
        } else if (type.equals(SilnikType.HYBRYDA)) {
            return new Hybryda();
        }
        return null;
    }
}

```

Sterowanie kreacją obiektów typu *Silnik* zawarte jest w klasie *SilnikFactory* zaprezentowanej na *Listingu 33*. Metoda *createSilnik()* z argumentem określającym typ silnika zwraca obiekt określonego typu silnika.

W prezentowanym przykładzie na całość implementacji wzorca projektowego *Fabryka* składa się sześć klas przedstawionych na *Rysunku 30*.



Rysunek 30. Pakiet z plikami źródłowymi w języku Java dla wzorca projektowego *Fabryka*. Źródło: opracowanie własne.

6.4.3 Integracja

W klasie *FabrykaMain* pokazanej na *Listingu 34* zawarto implementację kreacji różnych typów silników. Kod opiera się o istniejący w logice wzorca *Fabryka* zbiór klas typu *Silnik*. Na rzecz każdego stworzonego obiektu wywołano te same metody. Zwrócone przez metody wyniki były odmienne i uzależnione od rodzaju silnika, który je implementuje.

Listing 34. Reprezentacja klasy *FabrykaMain* pokazująca sposób implementacji wzorca projektowego *Fabryka*.

```

package fabryka;

public class FabrykaMain {

    public void start() {
        Silnik diesel = new
        SilnikFactory().createSilnik(SilnikFactory.SilnikType.DIESEL);
        Silnik beznyna = new
        SilnikFactory().createSilnik(SilnikFactory.SilnikType.BENZYNNA);
        Silnik hybryda = new
        SilnikFactory().createSilnik(SilnikFactory.SilnikType.HYBRYDA);
        diesel.wyswietlMoc();
    }
}

```

```

        diesel.wyswietlPojemnosc();
        beznyna.wyswietlMoc();
        beznyna.wyswietlPojemnosc();
        hybryda.wyswietlMoc();
        hybryda.wyswietlPojemnosc();
    }
}

```

```

TODO Diesel.wyswietlMoc()
TODO Diesel.wyswietlPojemnosc()
TODO Benzyna.wyswietlMoc()
TODO Benzyna.wyswietlPojemnosc()
TODO Hybryda.wyswietlMoc()
TODO Hybryda.wyswietlPojemnosc()

```

Rysunek 31. Dane wyświetlone w terminalu IDE po uruchomieniu kodu wzorca projektowego Fabryka.
Źródło: opracowanie własne.

Wyniki działania wywołanych metod są uzależnione od typu silnika, na rzecz którego zostały wywołane. Rysunek 31 pokazuje wyniki wywołania tych samych metod na rzecz różnych obiektów w IDE.

6.5 Dodatkowe funkcjonalności

W prezentowym podrozdziale opisano dodatkowe funkcje oferowane przez wtyczkę, wspierające etap wypełniania okna dialogowego podczas konfiguracji.

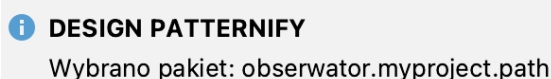
6.5.1 Notyfikacje

W wyniku interakcji użytkownika z systemem wtyczki na kluczowe zdarzenia generowane przez użytkownika wyświetlane są notyfikacje, które informują o aktualnym stanie systemu. Notyfikacje wyświetlane są w postaci wyskakujących okienek typu *Balloon* charakterystycznych dla środowiska *IntelliJ IDEA*. Okna są niewielkich rozmiarów, wyświetlane są w prawym, dolnym rogu graficznego interfejsu programu, znikają samoistnie po określonym czasie, możliwe jest anulowanie ich wyświetlania samodzielnie. Notyfikacje wyświetlane są w momencie:

- wyboru ścieżki dla katalogów w projekcie, do której nastąpi generowanie plików wzorca. Użytkownik jest informowany o wybranej ścieżce. Proces opisano w *Rozdziale 6.5.2.*,
- zatwierdzenia formularza z konfiguracją dla generatora plików. W wyniku poprawnego procesu generowania plików użytkownik jest informowany o powodzeniu procesu. Notyfikację pokazano na *Rysunku 19*,
- zatwierdzenia niepoprawnie wypełnionego formularza w oknie dialogowym konfiguratora. Proces opisano w *Rozdziale 6.5.3.*,
- braku metod mogących być wywołane wewnątrz logiki adaptera, w oknie konfiguratora dla wzorca projektowego Adapter. Proces opisano w *Rozdziale 6.5.4.*

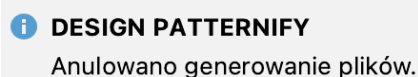
6.5.2 Wybór ścieżki

Wybór ścieżki do wygenerowania plików w konfiguratorze następuje po wybraniu przycisku *wybierz pakiet*. W przypadku braku wyboru przycisku pliki zostaną wygenerowane do głównego katalogu projektu. W środowisku *IntelliJ IDEA* domyślnym katalogiem jest katalog *src*. Wybór własnej ścieżki otworzy okno z możliwością nawigacji po strukturze katalogów, tworzenia nowych katalogów i wyboru ścieżki docelowej dla plików spośród istniejących katalogów. Na *Rysunku 32* pokazano notyfikację o nazwie wybranej przez użytkownika ścieżki do wygenerowania plików.



Rysunek 32. Notyfikacja informująca o wybranej przez użytkownika ścieżce do wygenerowania plików. Źródło: opracowanie własne.

Utworzenie nowego katalogu i zatwierdzenie go jako wybranej ścieżki docelowej dla plików, po czym anulowanie okna dialogowego z wyborem spowoduje, że nowo utworzony katalog zostanie usunięty. Wyświetlona zostanie stosowna notyfikacja o anulowaniu konfiguracji pokazana na *Rysunku 33*.



Rysunek 33. Notyfikacja informująca o anulowaniu okna konfiguracji dla dowolnego wzorca. Źródło: opracowanie własne.

Wybór istniejącego, niepustego katalogu na ścieżkę docelową dla generowania plików, zatwierdzenie wyboru ścieżki, a następnie anulowanie okna konfiguratora spowoduje, że istniejący katalog z zawartością nie zostanie usunięty.

6.5.3 Walidacja formularza

Podczas próby zatwierdzenia pustego formularza konfiguracji użytkownik otrzyma notyfikację o błędzie pokazaną na *Rysunku 34*. Informacja odnosić się będzie do pierwszego pola w formularzu. Domyślnie pole będzie zlokalizowane jako pierwsze, na samej górze okna. Informacja o wymaganym i niewypełnionym polu zostanie wyświetlona w *IDE*. W przypadku konfiguratora dla wzorca projektowego *Obserwator* będzie to pole o nazwie *nadawca*.



Rysunek 34. Notyfikacja informująca o niewypełnionym, wymaganym polu nadawcy w formularzu dla wzorca projektowego Obserwator. Źródło: opracowanie własne.

W przypadku pominięcia obligatoryjnych pól tekstowych w formularzu zostaną wyświetlone analogiczne notyfikacje. Warunek wymagalności jest podyktowany charakterystyką danego wzorca. W przypadku pól opcjonalnych notyfikacja nie zostanie wyświetlona.

Obligatoryjne pola tekstowe dla wzorca projektowego *Obserwator* stanowią:

- nazwa klasy nadawcy komunikatu,
- nazwy klas odbiorców komunikatów,
- deklaracje zmiennych będących w stanie obserwacji przez odbiorców, ale tylko w przypadku braku wczytanych pól z kodu klasy bazowej, poddanej procesowi integracji. W przypadku wczytania danych z klasy bazowej, poddanej integracji z generowanym kodem pole w formularzu konfiguratora, dotyczące deklaracji zmiennych jest opcjonalne.

Obligatoryjne pola tekstowe dla wzorca projektowego *Dekorator* stanowią:

- nazwy warstw składające się na strukturę wzorca,
- nazwy klas dekoratorów dla każdej warstwy,
- nazwy zmiennych odpowiedzialnych za przechowywanie stanu w warstwach, ale tylko w przypadku braku wczytania zmiennych z klasy bazowej, poddanej procesowi integracji. Wczytanie danych z klasy bazowej, poddanej integracji z generowanym kodem zmieni status wymagalności pola deklaracji zmiennych w formularzu konfiguratora na opcjonalny.

Obligatoryjne pola tekstowe dla wzorca projektowego *Adapter* stanowią:

- nazwa interfejsu dla klasy adaptera,
- nazwa klasy adaptera,
- deklaracje wszystkich metod w formularzu mających być adapterami.

Obligatoryjne pola tekstowe dla wzorca projektowego *Fabryka* stanowią:

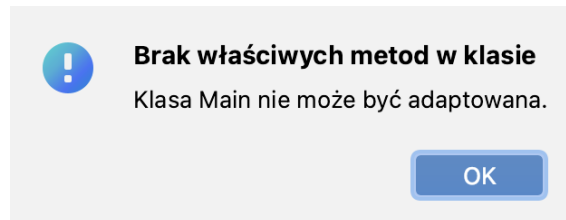
- nazwa interfejsu wspólnego dla klas budujących fabrykę,
- deklaracje nazw metod wspólnych dla obiektów fabryki,
- nazwy klas elementów składających się na fabrykę.

Nieudana próba zatwierdzenia formularza spowodowana brakami w wymaganych polach, a następnie anulowanie formularza i ponowne wywołanie formularza spowoduje, że wprowadzona zawartość pól tekstowych zostanie zapamiętana. Stan pól zostanie analogicznie zapamiętany, także po zatwierdzeniu błędnego formularza i otrzymaniu notyfikacji. Podany mechanizm ma ułatwić weryfikację błędnie wprowadzonych pól w celu spełnienia wymagań procesu walidacji.

6.5.4 Wczytywanie pól i metod

Prezentowana wtyczka posiada wbudowany mechanizm wczytywania pól lub metod z podanej klasy. Taka klasa zostanie poddana procesowi integracji z generowanym kodem. Wczytywanie pól jest opcjonalne. W uzasadnionych przypadkach mechanizm znacznie ułatwia proces integracji nowego kodu z już istniejącym.

Wyjątek stano wzorzec *Adapter*. Wymagane jest wskazanie klasy z metodami, które mają zostać poddane procesowi adaptacji. Metody zadeklarowane jako *static*, *protected* lub *private* nie będą wczytane przez konfigurator z klasy adaptowanej. Powód stanowi brak możliwości ich adaptacji. Takie deklaracje są niezgodne z logiką wzorca i nie zostaną uwzględnione przez konfigurator. *Rysunek 35* wyświetla notyfikację dla użytkownika o błędnie wskazanej klasie lub braku wskazanej klasy z metodami kompatybilnymi dla wzorca *Adapter*.



Rysunek 35. Notyfikacja o braku wymaganych metod w klasie na potrzeby generowanie kodu dla wzorca projektowego Adapter. Źródło: opracowanie własne.

Listing 35. Testowe zestawienie deklaracji pól i metod z komentarzami w celu sprawdzenia działania mechanizmu identyfikacji pól i metod przez prototyp.

```
public class Main {  
  
    int pole1;  
    //    int pole2;  
    double pole3;  
    /*    String pole4; */  
    String metoda1() {return ""};  
  
    //    String metoda2() {};  
    void metoda3() {}; // komentarz  
    void metoda4() {};  
  
    public static void main(String[] args) {  
  
    }  
}
```

Wtyczka posiada funkcjonalność rozpoznawania metod i pól, których implementacja została objęta komentarzem. Użycie komentarzy względem pól lub metod skutkuje ich automatycznym pominięciem. Implementacja nie zostanie wczytana w celu dalszego przetwarzania. *Listing 35* prezentuje przykładową konfigurację pól i metod objętych komentarzami w klasie *Main*.

W wyniku wywołania okna konfiguratora wzorca projektowego *Adapter* na bazie klasy *Main* do konfiguratora pokazanego na *Rysunku 36* zostaną wczytane tylko metody nie objęte komentarzem.

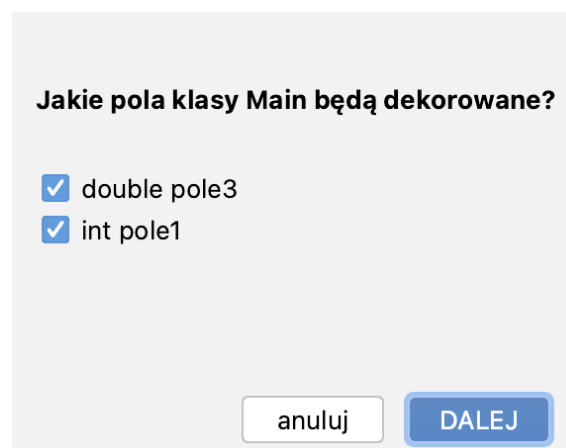
W wyniku wywołania okna konfiguratora wzorca projektowego *Dekorator* na bazie klasy *Main* do konfiguratora pokazanego na *Rysunku 37* zostaną wczytane tylko pola nie objęte komentarzem.



Jakie metody klasy Main będą adaptowane?

- ☒ String metoda1()
- ☒ void metoda3()
- ☒ void metoda4()

Rysunek 36. Wychwycone i pominięte przez formularz metody do integracji przez wzorec projektowy Adapter. Źródło: opracowanie własne.



Jakie pola klasy Main będą dekorowane?

- ☒ double pole3
- ☒ int pole1

Rysunek 37. Wychwycone i pominięte przez formularz pola do integracji przez wzorec projektowy Dekorator. Źródło: opracowanie własne.

7 Podsumowanie

Stosowanie wzorców projektowych czyni kod bardziej elastycznym podczas prób jego rozwoju i modyfikacji. Nie w każdym przypadku jednak należy stosować wzorce projektowe. W uzasadnionych przypadkach wdrożenie wzorców projektowych pozwala zminimalizować nakłady potrzebne na procesy utrzymania kodu.

Prezentowana praca skupia się na lepszym poznaniu mechanizmów rządzących wdrażaniem wzorców projektowych. Szczególnym aspektem jest proces integracji kodu wzorca z już istniejącą implementacją. W celu realizacji wymagań zastosowano automatyzację procesu generowania szablonów wzorców. W ten sposób proces integracji jest bardziej zrozumiały i szybszy.

Prezentowana praca różni się od innych podobnych tego typu rozwiązań. Odmienność warunkuje możliwość parametryzacji generowanego kodu. Uzyskany efekt umożliwia jak najlepsze dopasowanie wygenerowanego szablonu z kodem do implementacji istniejącej w projekcie.

Skutkiem wdrożenia prezentowanego rozwiązania jest prototyp. Prototyp ma postać elastycznej wtyczki do programu *IntelliJ IDEA*. Ułatwia automatyzację procesu wdrażania wzorców projektowych w projektach programistycznych. Wtyczka umożliwia integrację nowo wygenerowanego kodu z już istniejącą implementacją. Nowo wygenerowany kod jest poddany procesowi wstępnej konfiguracji przed procesem generowania. W opisany sposób możliwa jest szybsza i bardziej dopasowana konfiguracja implementacji dla procesu automatyzacji wdrażania wzorców projektowych. Proces ułatwia ściślejszą integrację kodu wzorca z istniejącym projektem.

W prezentowanej pracy opisano charakterystykę wybranych wzorców projektowych. Cel stanowi lepsze poznanie podjętej tematyki. Dodatkowo opisano:

- narzędzia informatyczne wykorzystane do konstrukcji prototypu,
- implementację prototypu,
- funkcjonalność oferowaną przez prototyp.

Spis listingów

Listing 1. Kod źródłowy wzorca projektowego Obserwator wygenerowany przez AI GitHub Copilot.	12
Listing 2. Przykładowa implementacja wzorca projektowego Obserwator w języku Java.	28
Listing 3. Przykładowa implementacja wzorca projektowego Dekorator w języku Java.	32
Listing 4. Przykładowa implementacja wzorca projektowego Adapter w języku Java.	35
Listing 5. Przykładowa implementacja wzorca projektowego Metoda Wytwórcza w języku Java.	39
Listing 6. Zawartość pliku konfiguracyjnego plugin.xml.	48
Listing 7. Kod źródłowy Klasy ActionFactory wzorca projektowego Fabryka.	50
Listing 8. Fragment kodu źródłowego klasy ActionAdapter odpowiedzialny za notyfikację błędu wczytywania klasy użytkownikowi.	51
Listing 9. Fragment implementacji klasy ActionAdapter przetwarzającej nazwy metod i ich argumentów dla wzorca projektowego Adapter.	52
Listing 10. Metoda z klasy SettersAdapter odpowiedzialna za wyświetlenie okna dialogowego z wyбором wczytanych metod.	53
Listing 11. Fragment implementacji klasy MainAdapter odpowiedzialnej za wyświetlenie okna konfiguratora dla generatora wzorca projektowego Adapter.	55
Listing 12. Fragment implementacji klasy MainAdapter odpowiedzialnej za generowanie zawartości pliku adaptera we wzorcu projektowym Adapter.	58
Listing 13. Wyrażenie lambda z logiką wyboru ścieżki dla plików przed ich wygenerowaniem.	67
Listing 14. Kod źródłowy klasy Fabryka wygenerowany w języku Java dla wzorca projektowego Obserwator.	69
Listing 15. Kod źródłowy interfejsów składających się na wzorec projektowy Obserwator.	70
Listing 16. Kod źródłowy klasy SalonKrakow wygenerowany w języku Java dla wzorca projektowego Obserwator.	70
Listing 17. Kod źródłowy klasy ObserwatorMain odpowiedzialnej za konfigurację obiektów wzorca projektowego Obserwator.	71
Listing 18. Metoda info(...) z klasy SalonGdansk z dopisaną logiką przetwarzania otrzymywanych danych.	72
Listing 19. Kod źródłowy klasy abstrakcyjnej najbardziej wewnętrznej warstwy składających się na wzorec projektowy Dekorator.	75
Listing 20. Kod źródłowy klasy LED-y wygenerowany w języku Java dla wzorca projektowego Dekorator.	76
Listing 21. Kod źródłowy klasy abstrakcyjnej środkowej warstwy składającej się na wzorec projektowy Dekorator.	76
Listing 22. Kod źródłowy klasy Syntetyk wygenerowany w języku Java dla wzorca projektowego Dekorator.	76
Listing 23. Kod źródłowy klasy abstrakcyjnej najbardziej zewnętrznej warstwy składającej się na wzorec projektowy Dekorator.	77
Listing 24. Kod źródłowy klasy Metalik wygenerowany w języku Java dla wzorca projektowego Dekorator.	77
Listing 25. Uzupełniona metoda i konstruktor w klasie LED-y odpowiedzialnej za wyświetlanie ceny auta z dodatkiem LED.	78

Listing 26. Uzupełniona metoda i konstruktor w klasie Syntetyk odpowiedzialnej za wyświetlenie ceny auta z dodatkiem LED i materiałem syntetycznym.	79
Listing 27. Kod klasy DekoratorMain odpowiedzialnej za powiązanie obiektów dekorowanych i dekoratorów.	79
Listing 28. Kod klasy FabrykaXML odpowiedzialnej za wysyłanie danych w formacie XML do salonów samochodowych.	80
Listing 29. Kod źródłowy klasy FabrykaJSON stanowiącej adapter łączący klasę FabrykaXML z salonami samochodowymi wymagającymi danych formatu JSON.	82
Listing 30. Modyfikacja metody pobierzJSON() w klasie FabrykaJSON zgodnie z logiką wzorca projektowego Adapter.	83
Listing 31. Kod klasy AdapterMain odpowiedzialnej za uruchomienie logiki wzorca projektowego Adapter.	83
Listing 32. Kod klasy Hybryda implementującej w sposób zgodny z logiką wzorca Fabryka interfejs Silnik.	85
Listing 33. Implementacja klasy SilnikFactory odpowiedzialnej za generowania obiektów silników zgodnie z logiką wzorca projektowego Fabryka.	85
Listing 34. Reprezentacja klasy FabrykaMain pokazująca sposób implementacji wzorca projektowego Fabryka.	86
Listing 35. Testowe zestawienie deklaracji pól i metod z komentarzami w celu sprawdzenia działania mechanizmu identyfikacji pól i metod przez prototyp.	90

Spis rysunków

Rysunek 1. Fragment nr 1 kodu źródłowego wzorca projektowego Obserwator wygenerowany w systemie Tutor Me. Źródło: opracowanie własne.	9
Rysunek 2. Fragment nr 2 kodu źródłowego wzorca projektowego Obserwator wygenerowany w systemie Tutor Me. Źródło: opracowanie własne.	10
Rysunek 3. Interfejs nr 1 wtyczki generującej wzorce projektowe w języku Go.	15
Rysunek 4. Interfejs nr 2 wtyczki generującej wzorce projektowe w języku Go. Źródło: [7].....	16
Rysunek 5. Interfejs wtyczki generującej wzorce projektowe w języku Java.	17
Rysunek 6. Kod źródłowy wzorca projektowego wygenerowany przez wtyczkę w języku Java. Źródło: [9]	18
Rysunek 7. Diagram klas UML wzorca projektowego Obserwator. Źródło: opracowanie własne.	27
Rysunek 8. Diagram klas UML wzorca projektowego Dekorator. Źródło: opracowanie własne.	31
Rysunek 9. Diagram klas UML wzorca projektowego Adapter. Źródło: opracowanie własne.	34
Rysunek 10. Diagram klas UML wzorca projektowego Metoda Wytwórcza. Źródło: opracowanie własne.....	38
Rysunek 11. Diagram aktywności dla przepływu pracy na przykładzie wzorca projektowego Dekorator. Źródło: opracowanie własne.	63
Rysunek 12. Diagram sekwencji dla przepływu informacji w systemie bez udziału klasy typu Setter. Źródło: opracowanie własne.	64
Rysunek 13. Diagram sekwencji dla przepływu informacji w systemie z udziałem klasy typu Setter. Źródło: opracowanie własne.	64
Rysunek 14. Menu wtyczki z możliwością wyboru wzorca projektowego do generowania. Źródło: opracowanie własne.	66
Rysunek 15. Okno dialogowe z wyborem wczytanych z klasy zmiennych dla wzorca projektowego Obserwator. Źródło: opracowanie własne.	67
Rysunek 16. Notyfikacja potwierdzająca wybór ścieżki do generowania plików wzorca projektowego Obserwator. Źródło: opracowanie własne.	67
Rysunek 17. Wypełnione okno dialogowe konfiguratora dla wzorca projektowego Obserwator. Źródło: opracowanie własne.	68
Rysunek 18. Notyfikacja potwierdzająca dodanie plików do bieżącego projektu. Źródło: opracowanie własne.....	69
Rysunek 19. Dane wyświetlone w terminalu IDE po uruchomieniu kodu wzorca projektowego Obserwator. Źródło: opracowanie własne.	72
Rysunek 20. Pakiet z plikami źródłowymi w języku Java dla wzorca projektowego Obserwator. Źródło: opracowanie własne.	72
Rysunek 21. Początkowa implementacja klasy DekoratorMain z deklaracją pól wychwyconych przez wtyczkę w celu selektywnego wyboru. Źródło: opracowanie własne.....	73
Rysunek 22. Wypełnione okno dialogowe konfiguratora dla wzorca projektowego Dekorator. Źródło: opracowanie własne.	74
Rysunek 23. Struktura pakietów i klas dla wzorca projektowego Dekorator. Źródło: opracowanie własne.....	75
Rysunek 24. Dane wyświetlone w terminalu IDE po uruchomieniu kodu wzorca projektowego Dekorator. Źródło: opracowanie własne.	80
Rysunek 25. Okno dialogowe z wyborem metod, które mają zostać poddane adaptacji we wzorcu projektowym Adapter. Źródło: opracowanie własne.....	81

Rysunek 26. Wypełnione okno dialogowe konfiguratora dla wzorca projektowego Adapter. Źródło: opracowanie własne.	81
Rysunek 27. Pakiet z plikami źródłowymi w języku Java dla wzorca projektowego Adapter. Źródło: opracowanie własne.	82
Rysunek 28. Dane wyświetlone w terminalu IDE po uruchomieniu kodu wzorca projektowego Adapter. Źródło: opracowanie własne.	84
Rysunek 29. Wypełnione okno dialogowe konfiguratora dla wzorca projektowego Fabryka. Źródło: opracowanie własne.	84
Rysunek 30. Pakiet z plikami źródłowymi w języku Java dla wzorca projektowego Fabryka. Źródło: opracowanie własne.	86
Rysunek 31. Dane wyświetlone w terminalu IDE po uruchomieniu kodu wzorca projektowego Fabryka. Źródło: opracowanie własne.	87
Rysunek 32. Notyfikacja informująca o wybranej przez użytkownika ścieżce do wygenerowania plików. Źródło: opracowanie własne.	88
Rysunek 33. Notyfikacja informująca o anulowaniu okna konfiguracji dla dowolnego wzorca. Źródło: opracowanie własne.	88
Rysunek 34. Notyfikacja informująca o niewypełnionym, wymaganym polu nadawcy w formularzu dla wzorca projektowego Obserwator. Źródło: opracowanie własne.	88
Rysunek 35. Notyfikacja o braku wymaganych metod w klasie na potrzeby generowanie kodu dla wzorca projektowego Adapter. Źródło: opracowanie własne.	90
Rysunek 36. Wychwycone i pominięte przez formularz metody do integracji przez wzorzec projektowy Adapter. Źródło: opracowanie własne.	91
Rysunek 37. Wychwycone i pominięte przez formularz pola do integracji przez wzorzec projektowy Dekorator. Źródło: opracowanie własne.	91

Prace cytowane

- [1] <https://platform.openai.com/docs/models/gpt-4o> [dostęp: 24.08.2024]
- [2] <https://pl.khanacademy.org/about> [dostęp: 24.08.2024]
- [3] <https://ieeexplore.ieee.org/abstract/document/7816479/references#references> [dostęp: 24.08.2024]
- [4] <https://github.com/DaskJhonson/go-design-pattern-generator/tree/master> [dostęp: 24.08.2024]
- [5] <https://docs.oracle.com/en/java/> [dostęp: 24.08.2024]
- [6] <https://go.dev/doc/> [dostęp: 24.08.2024]
- [7] <https://github.com/DaskJhonson/go-design-pattern-generator/tree/master> [dostęp: 27.08.2024]
- [8] <https://plugins.jetbrains.com/plugin/10856-design-patterns> [dostęp: 24.08.2024]
- [9] <https://github.com/OrPolyzos/design-patterns-plugin> [dostęp: 27.08.2024]
- [10] <https://users.pja.edu.pl/~mtrzaska/Files/PraceMagisterskie/200717-Jaszczuk.pdf> [dostęp: 24.08.2024]
- [11] <http://wiki.c2.com/?PeopleProjectsAndPatterns> [dostęp: 24.08.2024]
- [12] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Wzorce Projektowe, Helion S.A., ISBN: 9788328386099, 2021
- [13] Robert C. Martin: Clean Code. A Handbook of Agile Software Craftsmanship, Financial Times Prentice Hall, ISBN: 9780132350884, 2008
- [14] E. Freeman, E. Freeman, K. Sierra, B. Bates: Head First Design Patterns, O'Reilly Media, ISBN: 9780596007126, 2004

- [15] <https://refactoring.guru/pl/design-patterns/book> [dostęp: 24.08.2024]
- [16] Joshua Kerievsky: Refaktoryzacja do Wzorców Projektowych, Helion S.A., ISBN: 8373619305, 2004
- [17] <https://kotlinlang.org/docs/home.html> [dostęp: 24.08.2024]
- [18] <https://plugins.jetbrains.com/docs/intellij/kotlin-ui-dsl-version-2.html> [dostęp: 24.08.2024]
- [19] <https://www.tiobe.com/tiobe-index/> [dostęp: 24.08.2024]
- [20] <https://plugins.jetbrains.com/docs/intellij/welcome.html> [dostęp: 24.08.2024]
- [21] <https://github.com/JetBrains/intellij-community> [dostęp: 24.08.2024]
- [22] <https://github.com/JetBrains/intellij-platform-plugin-template> [dostęp: 24.08.2024]
- [23] <https://www.jetbrains.com/idea/> [dostęp: 24.08.2024]
- [24] <https://spring.io/> [dostęp: 24.08.2024]
- [25] <https://jakarta.ee/> [dostęp: 24.08.2024]
- [26] <https://www.linkedin.com/pulse/10-best-java-ides-editors-2024-hadisur-rahman-thbgf> [dostęp: 24.08.2024]
- [27] <https://git-scm.com/> [dostęp: 24.08.2024]
- [28] <https://github.com/trending> [dostęp: 24.08.2024]