POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Jakub Jaszczuk
19085

Narzędzia wspomagające implementację wzorców projektowych w zintegrowanych środowiskach programistycznych

Praca magisterska napisana
pod kierunkiem:

dr inż. Mariusz Trzaska

Warszawa, czerwiec, 2020

Streszczenie

Wzorce projektowe są uznanymi sposobami na rozwiązanie określonych problemów projektowych. Fakt ich schematyczności leży u podstaw analizy wykonanej w ramach niniejszej pracy.

Praca zawiera przedstawienie i ocenę obecnie dostępnych narzędzi służących wspomaganiu implementacji wzorców projektowych. Ponadto przedstawia teorię dotyczącą wzorców projektowych w ogólności oraz bardziej szczegółowo w przypadku wybranych wzorców, szczególnie istotnych w kontekście niniejszej pracy. Wynikiem analizy wybranych wzorców projektowych jest prototyp narzędzia wspomagającego ich implementację, które przyjmuje formę wtyczki do zintegrowanego środowiska programistycznego. Oferuje on możliwość rozszerzenia jego funkcjonalności przez użytkowników i społeczność zgodnie z pojawiającymi się potrzebami i pomysłami.

Wnioski zawarte w podsumowaniu stanowią ocenę użyteczności prototypu narzędzia wspomagającego implementację wzorców projektowych w zintegrowanych środowiskach programistycznych. Dodatkowo praca przedstawia możliwości rozwoju prototypu w odniesieniu do jego aktualnej funkcjonalności, zalet oraz wad.

Słowa kluczowe: wzorce projektowe, zintegrowane środowisko programistyczne, wtyczka

Spis treści

1. Wstęp

Wzorce projektowe, będąc sprawdzonymi rozwiązaniami powtarzalnych problemów projektowych, są szczególnie istotnym zagadnieniem w dziedzinie programowania obiektowego. Znajomość wzorców jest szczególnie ceniona przez pracodawców i coraz częściej wymagana nawet na stanowisku młodszego programisty. Odpowiednie zrozumienie wzorców projektowych, umiejętne ich stosowanie oraz łączenie przez programistów, często przesądza o jakości kodu projektu, co w konsekwencji przekłada się na niższe koszty jego utrzymania i rozwoju.

1.1. Wspomaganie implementacji wzorców projektowych w zintegrowanych środowiskach programistycznych

Obecnie, pomimo często występujących w kontekście wzorców projektowych pojęć, takich jak schematyczność i powtarzalność, brak jest popularnego narzędzia wykorzystującego wspomnianą schematyczność w celu wspomagania implementacji wzorców projektowych. Narzędzie oferujące taką funkcjonalność powinno ponadto być maksymalnie „blisko" programisty, tak aby jego używanie było naturalną czynnością w procesie tworzenia kodu. Z tego właśnie względu prototyp stworzony w ramach niniejszej pracy przyjmuje formę wtyczki do zintegrowanego środowiska programistycznego. Prototyp ponadto umożliwia rozszerzenie jego funkcjonalności, wykorzystując fakt, że programiści będąc wysoce wyspecjalizowanymi użytkownikami, posiadają odpowiednie umiejętności by stworzyć rozszerzenie prototypu, tym samym dostosowując go do swoich potrzeb.

1.2. Cel pracy

Celem niniejszej pracy jest ocena zasadności stosowania i rozwijania narzędzi wspomagających implementację wzorców projektowych w zintegrowanych środowiskach programistycznych.

1.3. Rozwiązanie przyjęte w pracy

Prototyp powstały w ramach niniejszej pracy wspomaga implementację wzorców projektowych w języku Java. Z tego też względu pełni on rolę wtyczki do zintegrowanego środowiska programistycznego, jakim jest IntelliJ IDEA. Prototyp powstał w oparciu o IntelliJ Platform, a jego kod źródłowy napisany jest w języku Java. W procesie jego tworzenia wykorzystane zostało narzędzie kontroli wersji Git, wraz z usługa hostingową przeznaczoną dla repozytoriów Gita jaką jest serwis GitHub, a także Gradle, służący automatyzacji budowy aplikacji.

1.4. Rezultaty pracy

Rezultaty niniejszej pracy to:

- ocena użyteczności narzędzi wspomagających implementację wzorców projektowych w zintegrowanych środowiskach programistycznych,
- propozycje kierunków rozwoju wspomnianych narzędzi w przyszłości.

Rezultatem pobocznym jest prototyp narzędzia wspomagającego implementację wzorców projektowych w zintegrowanych środowiskach programistycznych, przyjmujące formę wtyczki do IntelliJ IDEA.

1.5. Organizacja pracy

Niniejsza praca w rozdziale drugim przedstawia dostępne narzędzia wspomagające implementację wzorców projektowych, uwzględniając podobieństwa i różnice wedle powstałego prototypu.

Kolejne rozdziały zawierają teorię dotyczącą wzorców projektowych, ze szczególnym uwzględnieniem wzorców istotnych w kontekście powstałego prototypu.

Następny rozdział przedstawia narzędzia wykorzystane do realizacji prototypu wraz z argumentacją ich wyboru.

Rozdział piąty przybliża architekturę i implementację wtyczki wspomagającej implementację wzorców projektowych.

Rozdział kolejny prezentuje działanie prototypu dla każdego z pięciu wzorców projektowych, których implementację wspomaga.

Dokument kończy podsumowanie, które zawiera ocenę zasadności stosowania i rozwijania narzędzi wspomagających implementację wzorców projektowych w zintegrowanych środowiskach programistycznych, bazującą na prototypie powstałym w ramach pracy. Podsumowanie dodatkowo zawiera propozycje możliwych kierunków rozwoju prototypu w przyszłości.

2. Stan sztuki

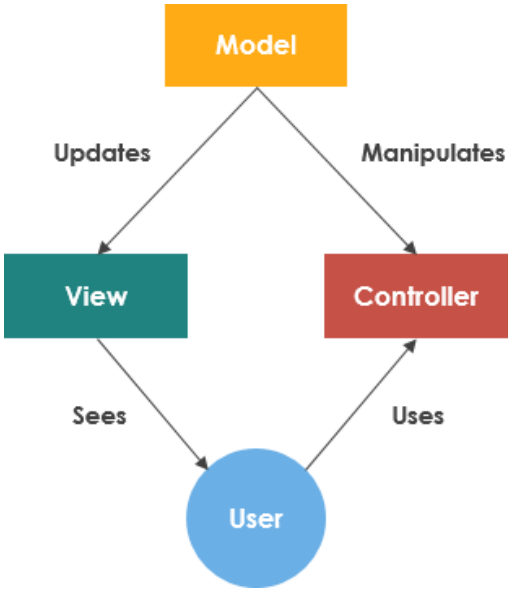
Niniejszy rozdział zawiera przedstawienie istniejących rozwiązań, których funkcjonalność pozwala na automatyczną generację kodu w kontekście wzorców projektowych.

2.1. JHipster

Pierwszy narzędziem, o którym warto wspomnieć w dziedzinie automatycznej generacji kodu, jest JHipster [10]. Jest to właściwie ogromnie rozbudowana platforma oferująca wiele możliwości, począwszy od generowania kodu aż po integrację wygenerowanej aplikacji z szeregiem uznanych i powszechnie używanych narzędzi oraz technologii, do których należą między innymi:

- Angular,
- React,
- Vue,
- Spring Boot,
- Gradle,
- Maven,
- Kafka,
- Docker,
- AWS

JHipster jednak w porównaniu od prototypu wtyczki opisanej w tej pracy działa na innym poziomie abstrakcji w dziedzinie automatycznego generowania kodu. Natomiast w kontekście wzorców projektowych, jako że wykorzystując JHipster warstwę backendową można wygenerować w oparciu o framework Spring Boot, w wygenerowanym kodzie odnajdziemy wzorzec Model-Widok-Kontroler, którego diagram interakcji przedstawiono na rysunku 1.



Rysunek 1. Diagram interakcji wzorca MVC. Źródło: [25]

Wzorzec ten jest często mylnie nazywany wzorcem projektowym, podczas gdy w książce „Wzorce Projektowe" Ericha Gammy i in. [1] jeden z podrozdziałów konsekwentnie nazywa Model-Widok-Kontroler architekturą i traktuje o wzorcach projektowych, które w tymże wzorcu architektonicznym są stosowane. JHipster pomimo bogatego zestawu możliwości nie może być rozważany jako alternatywa prototypu wtyczki opisanego w tej pracy.

2.2. FreeBuilder

Kolejnym narzędziem [11] dostępnym na rynku, które wspomaga implementację wzorców projektowych, a konkretnie jednego wzorca – budowniczego, jest biblioteka FreeBuilder, przeznaczona dla języka Java. W celu użycia FreeBuildera należy stworzyć interfejs lub klasę abstrakcyjną, która będzie posiadała informacje o polach, jakie ma zawierać docelowy obiekt. Informacje te są pozyskiwane przez bibliotekę na podstawie deklaracji metod. Samo użycie FreeBuildera polega na oznaczeniu interfejsu lub klasy abstrakcyjnej, którego budowniczego chcemy wygenerować, adnotacją `@FreeBuilder`. Następnie niezbędne jest utworzenie wewnętrznej klasy budowniczego, który będzie dziedziczył po automatycznie wygenerowanej klasie zawierającej aplikacyjny interfejs programistyczny. Za pomocą utworzonej przez użytkownika wewnętrznej klasy można konfigurować automatycznie wygenerowaną klasę budowniczego, w celu na przykład ustawienia domyślnych wartości jego konstruktora. Przykładowy interfejs z adnotacją `@FreeBuilder` przedstawiono na listingu 1.

Listing 1. Przykładowy interfejs z adnotacją `@FreeBuilder`. Źródło: [11]

```
@FreeBuilder
public interface Person {
  /** Returns the person's full (English) name. */
  String name();
  /** Returns the person's age in years, rounded down. */
  int age();
  /** Returns a human-readable description of the person. */
  String description();
  /** Builder class for {@link Person}. */
  class Builder extends Person_Builder {
    public Builder() {
      // Set defaults in the builder constructor.
      description("Indescribable");
    }
    @Override Builder age(int age) {
      // Check single-field (argument) constraints in the setter method.
      checkArgument(age >= 0);
      return super.age(age);
    }
    @Override public Person build() {
      // Check cross-field (state) constraints in the build method.
      Person person = super.build();
      checkState(!person.description().contains(person.name()));
      return person;
    }
  }
}
```

FreeBuilder, oprócz klasy budowniczego, wygeneruje również klasę implementującą interfejs stworzony wcześniej przez użytkownika, która będzie posiadała implementację metod `equals`, `toString` i `hashCode`. Dodatkowo wygenerowana zostanie klasa będąca częściową implementacją

(według oficjalnej dokumentacji, oznacza to taką, która może naruszać ograniczenia, przykładowo poprzez brak wymaganych pól) wcześniejszego interfejsu przeznaczoną do wykorzystania w testach jednostkowych. Ponadto FreeBuilder między innymi:

- obsługuje kolekcje (także te z biblioteki Guava),
- wspiera procesor JSON języka Java – Jackson,
- wspiera GWT,
- oferuje metody mutujące kolekcje i mapy, których argumentem może być znane w języku Java wyrażenie lambda,
- udostępnia metodę `buildPartial`, która pozwala na tworzenie za pomocą budowniczego niepełnych obiektów, które łamią nałożone ograniczenia. Metoda ta nie powinna być stosowana w kodzie produkcyjnym, lecz może okazać się bardzo użyteczna przy pisaniu testów [11].

Niewątpliwą zaletą FreeBuildera jest ograniczenie ilości nadmiarowego kodu, który powstaje przy implementacji wzorca budowniczy, a także dość bogaty zbiór interesujących funkcji odnoszących się do budowniczego oraz wsparcie kolekcji czy GWT. Jednak z drugiej strony jego użycie jest ograniczone tylko do jednego wzorca projektowego, co czyni go narzędziem o wąskim spektrum zastosowań. Z tego właśnie powodu FreeBulder mógłby być co najwyżej alternatywą dla fragmentu funkcjonalności prototypu przedstawionego w niniejszej pracy, związanej ze wzorcem budowniczy.

2.3. Design Patterns

Rozwiązaniem najbardziej zbliżonym do prototypu wtyczki opisanej w tej pracy jest wtyczka Design Patterns [12] dostępna w oficjalnym repozytorium wtyczek firmy JetBrains, czyli właściciela IntelliJ IDEA. Główna funkcja Design Patterns jest tożsama z funkcją prototypu, to znaczy służy automatycznemu generowaniu kodu implementującego wybrany przez użytkownika wzorzec projektowy. Pomimo tego podobieństwa można wyszczególnić też szereg różnic pomiędzy dwiema wtyczkami, które wpływają na wygodę i łatwość użytkowania, liczność zastosowań oraz, co najważniejsze, perspektywę rozwoju i rozszerzenia funkcjonalności. Różnice na korzyść prototypu, które należy wymienić to:

- wzbogacony interfejs użytkownika, bardziej „naturalne” umiejscowienie interfejsu służącego do uruchomienia głównych funkcji wtyczki,
- instrukcja użycia wtyczki dla każdego z obsługiwanych wzorców wraz z odnośnikami do witryn z informacjami o nich,
- inny zbiór obsługiwanych wzorców,
- interfejs programistyczny aplikacji dający możliwość rozszerzenia funkcjonalności wtyczki o własną implementację kolejnych wzorców. Jest to szczególnie ważne w przypadku chęci dostosowania działania wtyczki do oczekiwań programistów, którzy będąc wysoce wyspecjalizowaną grupą użytkowników, zyskują istotne narzędzie do praktycznie dowolnego rozszerzenia funkcjonalności wtyczki w przyszłości.

2.4. Podsumowanie

JHipster, pomimo ogromnej skali i dużej liczby wspieranych technologii, nie posiada funkcji pozwalającej automatyczną generację kodu w kontekście wzorców projektowych. FreeBuilder natomiast realizuje wspomnianą funkcję w sposób wyczerpujący i uwzględnia wszelakie potrzeby użytkowników. Czyni to jednak tylko w kontekście jednego wzorca - wzorca budowniczy. Z dostępnych na rynku rozwiązań jedynie niszowa wtyczka Design Patterns, która w momencie

powstawania niniejszej pracy została pobrana jedynie ponad pięć tysięcy razy z oficjalnego sklepu firmy JetBrains, pozwala na automatyczną generację kodu w kontekście wzorców projektowych. Jest ona jednak narzędziem niszowym, posiadającym szczątkową dokumentację, która ogranicza się do lakonicznego opisu funkcjonalności wtyczki. Ponadto narzędzie Design Patterns nie posiada żadnego interfejsu pozwalającego na rozszerzenie jego funkcjonalności.

Istniejące rozwiązania pokazują, że obecnie na rynku nie istnieje ani jedno popularne i (lub) rozbudowane narzędzie pozwalające na automatyczną generację kodu w kontekście wzorców projektowych. Tym samym utworzenie narzędzia realizującego wspomnianą funkcję, a przede wszystkim pozwalającego na dowolne rozszerzenie jego funkcjonalności, jest uzasadnione.

3. Wzorce projektowe i ich rola w procesie wytwarzania oprogramowania

Niniejszy rozdział przedstawia opis zasad projektowych SOLID i teorię dotyczącą wzorców projektowych. Ponadto podano definicję wzorca projektowego, podział wzorców projektowych oraz opis wzorców projektowych szczególnie istotnych w kontekście tej pracy.

3.1. SOLID

SOLID to mnemonik, pod którym kryje się pięć zasad mówiących o tym, jak pisać dobrej jakości kod zorientowany obiektowo. Kod napisany zgodnie z tymi regułami jest między innymi tańszy w utrzymaniu, łatwiej rozszerzalny i bardziej zrozumiały. Jednym z największych propagatorów SOLID jest Robert C. Martin, znany jako *Wujek Bob*, autorytet w dziedzinie tworzenia oprogramowania. Zdaniem autora tej pracy dobra znajomość i rozumienie każdej z pięciu zasad jest kluczowe w kontekście wzorców projektowych. Analizując, czy też zaznajamiając się, ze wzorcami projektowymi, można zastanawiać się, dlaczego zostały one wymyślone w pewien konkretny sposób. Argumenty będące odpowiedzią na te pytanie są zawarte w literaturze w opisach danych wzorców projektowych. Jednak często fundamentami dla tych argumentów są reguły SOLID, które brzmią następująco [9]:

- zasada jednej odpowiedzialności – „klasa powinna mieć tylko jeden powód do zmiany”,
- zasada otwarte-zamknięte – „podmioty oprogramowania (klasy, moduły, funkcje, itp.) powinny być otwarte na rozszerzenie, ale zamknięte na modyfikacje”,
- zasada podstawienia Liskov – „podtypy muszą być substytucyjne dla ich typów podstawowych”,
- zasada segregacji interfejsów – należy unikać interfejsów o niskiej kohezji, co oznacza, że lepszym rozwiązaniem jest więcej niż jeden interfejs o wysokiej kohezji niż jeden o niskiej kohezji,
- zasada odwrócenia zależności – „moduły wysokiego poziomu nie powinny zależeć od modułów niskiego poziomu. Oba powinny zależeć od abstrakcji” oraz „abstrakcje nie powinny zależeć od szczegółów. Szczegóły powinny zależeć od abstrakcji”.

3.2. Ogólna charakterystyka wzorców projektowych

W literaturze można spotkać wiele definicji wzorców projektowych. Do najważniejszych z nich należą następujące:

- „wzorce projektowe to powtarzające się rozwiązania problemów projektowych, które widzisz w kółko” [6],
- „wzorce projektowe stanowią zbiór zasad opisujących sposób wykonywania określonych zadań w dziedzinie tworzenia oprogramowania” [8],
- „wzorzec rozwiązuje powtarzający się problem projektowy, który pojawia się w konkretnych sytuacjach projektowych i przedstawia rozwiązanie” [7],
- „wzorce identyfikują i określają abstrakcje, które są powyżej poziomu pojedynczych klas i instancji lub składników” [1].

Autorzy ostatniej definicji określają również wzorce projektowe zawarte w swojej książce jako „[…] opisy komunikujących się obiektów i klas przeznaczonych do rozwiązywania ogólnych problemów projektowych w określonym kontekście". Ponadto wymieniają cztery kluczowe elementy, z których składają się wzorce projektowe. Są to:

- nazwa wzorca – najlepiej, gdyby były to pojedyncze słowa opisujące wzorzec projektowy. Dobranie właściwej nazwy jest bardzo istotne, ponieważ umożliwia tworzenie słownika, który usprawnia komunikację, systematyzuje wiedzę i ułatwia dokumentowanie,
- problem – opis przeszkody, którą wzorzec pomaga przezwyciężyć, określający kontekst w jakim sprawdzi się zastosowanie danego wzorca projektowego,
- rozwiązanie – nie powinno skupiać się na opisie implementacji solucji pojedynczego problemu, ale powinno przedstawiać szablon opisujący zastosowania i zależności elementów, które się na niego składają,
- konsekwencje – opis skutków zastosowania wzorca projektowego, powinien on zawierać informacje o wadach i zaletach wzorca, także w kontekście przyszłości projektu.

W książce *Head first design patterns* znajdziemy natomiast następującą, zwartą definicję wzorca: „wzorzec jest rozwiązaniem problemu w kontekście" [2]. Autorzy książki w tej definicji wyróżniają:

- kontekst – powtarzająca się sytuacja, w której należałoby użyć wzorca,
- problem – opisany za pomocą celu, który chcemy osiągnąć, by rozwiązać problem, uwzględniwszy wszystkie ograniczenia,
- rozwiązanie – szablon rozwiązania problemu.

Jako że z przytoczonych definicji jasno wynika, że istotną, jeśli nie najistotniejszą, funkcją wzorców projektowych jest przedstawianie rozwiązań dotyczących powszechnych problemów, bardzo ważny jest usystematyzowany schemat opisu wzorców. Dzięki temu korzystanie z katalogów wzorców projektowych, które służą zdobywaniu wiedzy o wzorcach, ich porównywaniu oraz analizowaniu jest zdecydowanie szybsze i łatwiejsze. *Banda Czterech* w swojej książce [1] wymienia następujące elementy opisu wzorców projektowych:

- nazwa i kategoria wzorca – nazwa jest elementem wzorca projektowego, który jest kluczowy przy tworzeniu słownika. Kategoria służy podziałowi wzorców projektowych (wyróżniamy trzy główne kategorie, które zostaną przedstawione w dalszej części tekstu),
- przeznaczenie – krótki opis celu osiąganego przez zastosowanie wzorca,
- inne nazwy,
- uzasadnienie – przykładowa sytuacja, zawierająca opis trudności wraz ze sposobem na jej rozwiązanie,
- warunki stosowania – określają, kiedy wzorzec może zostać z powodzeniem zastosowany,
- struktura – diagram klas wzorca. Dodatkowo może zawierać inne diagramy klas, obiektów, interakcji,
- elementy – opis funkcji poszczególnych klas i (lub) obiektów będących elementami wzorca,
- współdziałanie – opis relacji pomiędzy poszczególnymi elementami wzorca,
- konsekwencje – opis skutków zastosowania wzorca projektowego, zarówno tych dobrych, jak i złych,
- implementacja – opis jak największej liczby aspektów implementacji wzorca, powinien zawierać szereg porad, sugestii oraz poruszać między innymi kwestie możliwych problemów i trudności,
- przykładowy kod – kod będący implementacją wzorca,
- znane zastosowania – przykłady zastosowań znane z prawdziwych systemów,
- powiązane wzorce – opis zależności danego wzorca z innymi wybranymi wzorcami.

Ze względu na to, że już w książce *Bandy Czterech* [1] opisane zostały dwadzieścia trzy wzorce, a ogólna ich liczba stale rośnie, konieczna jest klasyfikacja wzorców. Dzięki niej, wiedza na temat wzorców jest łatwiejsza do przyswojenia. Dodatkowo usprawnia to poszukiwania i porównania odpowiednich wzorców projektowych. Najpopularniejszym podziałem wzorców, jest podział na trzy rodzaje [1]:

- kreacyjne (konstrukcyjne) – dotyczą tworzenia obiektów, pozwalają na uniezależnienie systemu od sposobu instancjonowania klas,
- strukturalne – dotyczą budowania większych struktur obiektów i klas,
- behawioralne (operacyjne) – skupiają się na opisie zachowań klas i obiektów.

Inną znaną klasyfikacją jest podział wzorców ze względu na ich zasięg:

- klasowe – dotyczą zależności pomiędzy klasami,
- obiektowe – dotyczą zależności pomiędzy obiektami.

3.3. Wzorce wykorzystane w pracy

Niniejszy podrozdział zawiera teorię dotyczącą wzorców szczególnie istotnych w kontekście tej pracy. Są to wzorce, których implementację umożliwia prototyp wtyczki. Zostały one wybrane subiektywnie, a decydował o tym potencjał automatyzacji ich implementacji. Wzorce, które zostały wytypowane, według autora charakteryzują się co najmniej jedną z następujących cech: schematycznością implementacji, pewnymi stałymi jej elementami, powtarzalnymi i (lub) parametryzowalnymi fragmentami kodu implementacji.

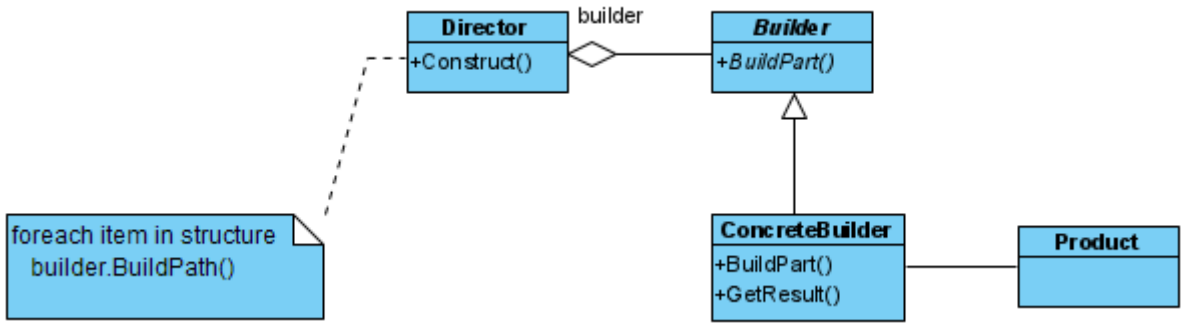
3.3.1 Budowniczy

Często instrukcja dotycząca instancjonowania klasy jest zawarta w niej samej w postaci jej konstruktora. Taka praktyka sprawdza się do momentu, gdy na przykład:

- obiekt, który chcemy stworzyć jest złożony,
- proces tworzenia pożądanego obiektu jest skomplikowany, często wieloetapowy,
- reprezentacja obiektu nie zawsze jest taka sama [3].

W takich sytuacjach z pomocą przychodzi nam wzorzec budowniczy, należący do obiektowych wzorców kreacyjnych, którego przeznaczeniem jest oddzielenie konstrukcji obiektu od jego reprezentacji [1]. Wzorzec ten często działa w połączeniu ze wzorcem fabryka [4].

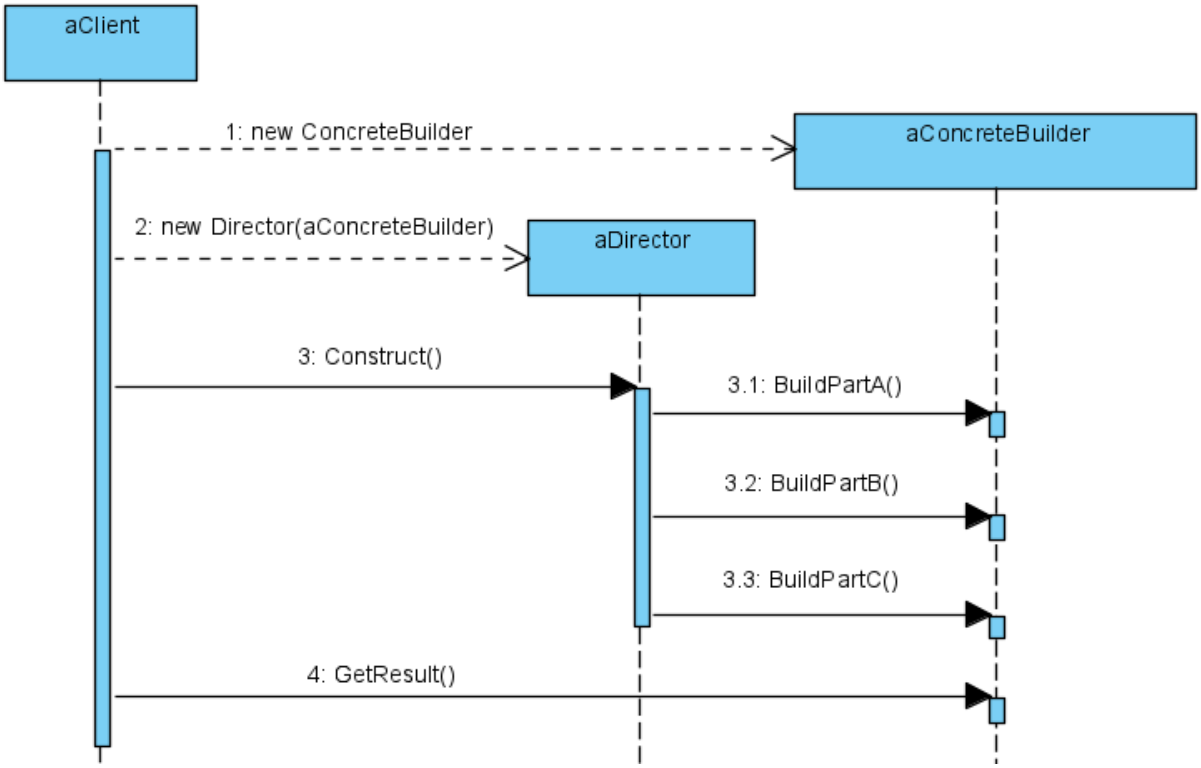
Strukturę wzorca budowniczy przedstawiono na rysunku 2.



Rysunek 2. Struktura wzorca budowniczy. Źródło: [1]

Najważniejszymi elementami struktury są:

- Budowniczy (**Builder**) – jest to miejsce, w którym odbywa się inicjalizacja składników produktu,
- Zarządca (**Director**) – jest to miejsce przechowujące instrukcje procesu inicjalizacji,
- Produkt (**Product**) – złożona instancja klasy, która inicjalizowana jest przy użyciu Buildera [4].



Rysunek 3. Scenariusz użycia wzorca budowniczy. Źródło: [1]

Scenariusz użycia wzorca budowniczy przedstawiono na rysunku 3 i rozpoczyna się on od stworzenia przez klienta obiektu Zarządca i sparametryzowania go za pomocą obiektu Konkretny Budowniczy. We wspomnianym wcześniej połączeniu wzorca budowniczy ze wzorcem fabryka, często to właśnie fabryka dostarcza odpowiedniego obiektu klasy Konkretny Budowniczy [3]. Innym przykładem współpracy tych dwóch wzorców może być wykorzystanie budowniczego wewnątrz fabryki [4]. W kolejnym kroku klient wywołuje metodę `build` Zarządcy, w której to Zarządca w odpowiedniej kolejności wywołuje metody Budowniczego. Po zakończeniu procesu inicjalizacji produktu klient pobiera go od obiektu Builder.

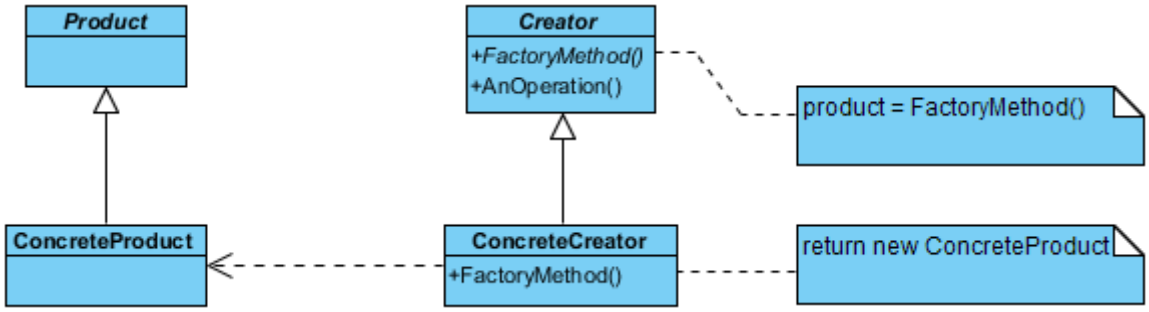
Zastosowanie wzorca budowniczy w konsekwencji pozwala na [3]:

- łatwiejsze dodawanie nowych wewnętrznych reprezentacji. Można to zrobić poprzez dodanie nowego Konkretnego Budowniczego,
- zmniejszenie rozmiarów klasy Produktu,
- większą modularność, osiąganą dzięki wydzieleniu poszczególnych implementacji do Konkretnych Budowniczych,
- pełną kontrolę nad procesem tworzenia Produktu.

3.3.2 Metoda wytwórcza

Wzorzec projektowy metoda wytwórcza należy do klasowych wzorców konstrukcyjnych i określa on interfejs służący do tworzenia obiektów, przenosząc przy tym odpowiedzialność za wybór konkretnej klasy do podklas [1]. Innymi słowy metoda wytwórcza pozawala podklasom zdecydować o instancji klasy, którą utworzą. Dzieje się tak dlatego, że interfejs do tworzenia obiektów nie jest świadomy tego, jaki produkt zostanie stworzony.

Strukturę wzorca metoda wytwórcza przedstawiono na rysunku 4.



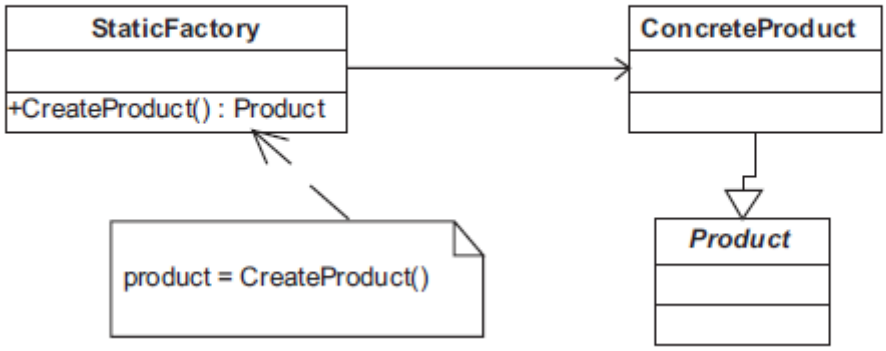
Rysunek 4. Struktura wzorca metoda wytwórcza. Źródło: [1]

Najważniejszymi elementami tej struktury są:

- Produkt (**Product**) – jest to interfejs, który implementują wszystkie Konkretne Produkty,
- Konkretny Produkt (**Concrete Product**) – jest to implementacja interfejsu Produktu,
- Kreator (**Creator**) – zawiera deklarację metody wytwórczej, zwracającej Produkty. Może zawierać jej domyślną implementację oraz sam może ją wywoływać,
- Konkretny Kreator (**Concrete Creator**) – przesłania lub, w przypadku braku domyślnej implementacji, implementuje metodę wytwórczą, która w tym miejscu zwraca Konkretne Produkty.

Tłumacząc nazwę wzorca projektowego metoda wytwórcza bezpośrednio z języka angielskiego, można wzorzec ten nazwać metodą fabryczną i to właśnie ze słowem fabryka powiązane są dwa ważne pojęcia. Pierwsze z nich to wzorzec projektowy fabryka abstrakcyjna, udostępniający interfejs do tworzenia rodzin powiązanych ze sobą obiektów. Drugie, szczególnie istotne w kontekście tej pracy, to prosta fabryka, która zdaniem autorów książki *Head first design patterns* [2] powinna być klasyfikowana jako idiom programistyczny, czyli zwyczajowy sposób kodowania rozwiązania pewnego, najczęściej prostego problemu w jednym lub kilku językach programowania. Jednak w wielu postach, wpisach na blogach oraz książkach klasyfikowana jest jako wzorzec projektowy. Prosta fabryka, podobnie jak inne fabryki, pozwalają na kapsułkowanie tworzenia konkretnych instancji klas. Od metody wytwórczej odróżnia ją przede wszystkim brak możliwości oddelegowania decyzji o użyciu konkretnej implementacji do podklas [2].

Strukturę prostej fabryki przedstawiono na rysunku 5.



Rysunek 5. Struktura prostej fabryki. Źródło: [4]

Najważniejszymi elementami te struktury są:

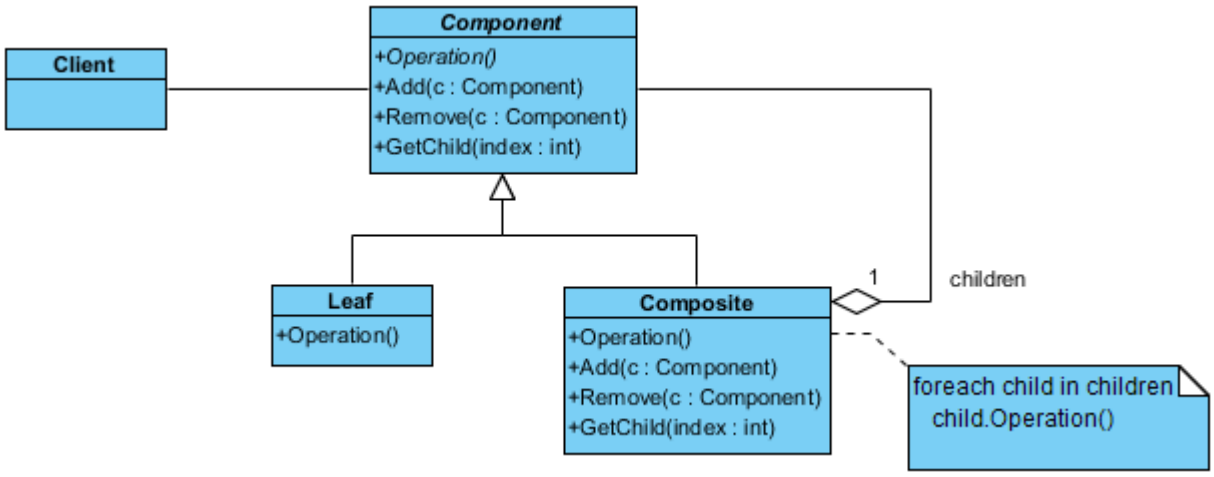
- Produkt (**Product**) – jest to wspólny interfejs dla wszystkich Konkretnych Produktów,
- Konkretny Produkt (**Concrete Product**) – jest to implementacja interfejsu Produktu,
- Fabryka (**Static Factory**) – odpowiada za tworzenie Produktów. Ponadto powinno to być jedyne miejsce, w którym znajdą się odniesienia do Konkretnych Produktów [2].

Prosta fabryka sprawdzi się świetnie, gdy proces tworzenia instancji klasy jest skomplikowany, wymaga wielu operacji i jest rozrzucony w wielu miejscach w kodzie programu. Zastosowanie jej pozwoli na łatwiejsze utrzymanie kodu, ponieważ w przypadku jakiejkolwiek zmiany we wspomnianym procesie będzie wymagało zmiany wyłącznie w jednym miejscu. Innym przykładem zastosowania prostej fabryki może być sytuacja, w której nie chcemy decydować w czasie kompilacji, której implementacji abstrakcji użyć do wykonania pewnej logiki, tylko pozwolić programowi zdecydować o tym w trakcie jego działania.

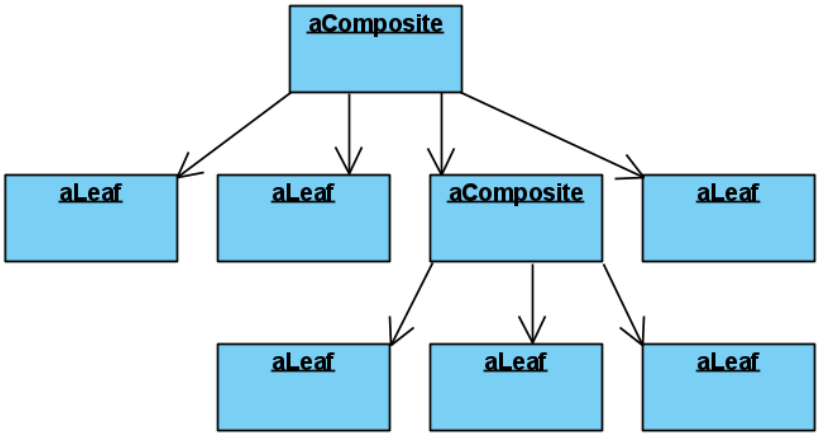
3.3.3 Kompozyt

Wzorzec projektowy kompozyt należy do obiektowych wzorców strukturalnych i umożliwia reprezentowanie hierarchii typu część-całość za pomocą drzew [2]. Innymi słowy oznacza to możliwość tworzenia kolekcji obiektów, z których każdy może być traktowany jako kompozyt lub indywidualny byt. Przekładając to na nomenklaturę struktury drzewiastej, elementy kompozytu mogą być liśćmi lub węzłami posiadającymi potomków [5]. W sytuacjach, gdy programista tworzy komponent, który może być indywidualnym bytem bądź kolekcją bytów, kompozyt pozwala na traktowanie ich w jednakowy sposób [1].

Strukturę wzorca kompozyt przedstawiono na rysunku 6, a przykładową strukturę kompozytu na rysunku 7.



Rysunek 6. Struktura wzorca kompozyt. Źródło: [1]



Rysunek 7. Przykładowa struktura kompozytu. Źródło: [1]

Najważniejszymi elementami struktury są:

- Komponent (**Component**) – deklaruje interfejs dla wszystkich obiektów (zarówno liści jak i kompozytów), a także może implementować domyślne zachowania [2]. Daje dostęp do elementów podrzędnych i umożliwia zarządzanie nimi (opcjonalnie może to dotyczyć również elementów nadrzędnych [1]),
- Liść (**Leaf**) – reprezentuje elementy nieposiadające dzieci i definiuje ich zachowanie poprzez implementację metod wspieranych przez Kompozyt [2],
- Kompozyt (**Composite**) – reprezentuje elementy posiadające dzieci, definiuje ich zachowanie i jest miejscem, w którym dzieci są przechowywane [2].
- Klient (**Client**) – manipuluje elementami kompozytu za pomocą interfejsu Komponentu [1].

Użycie wzorca kompozyt polega na wywoływaniu przez Klienta metod udostępnionych przez Komponent. Gdy metoda zostanie wywołana na obiekcie typu Liść, ten wykona ją bezpośrednio. Jeżeli natomiast zostanie wywołana na obiekcie typu Kompozyt, zazwyczaj następuje oddelegowanie jej wywołania do dzieci. Dodatkowo może wykonać pewne akcje przed i (lub) po oddelegowaniu.

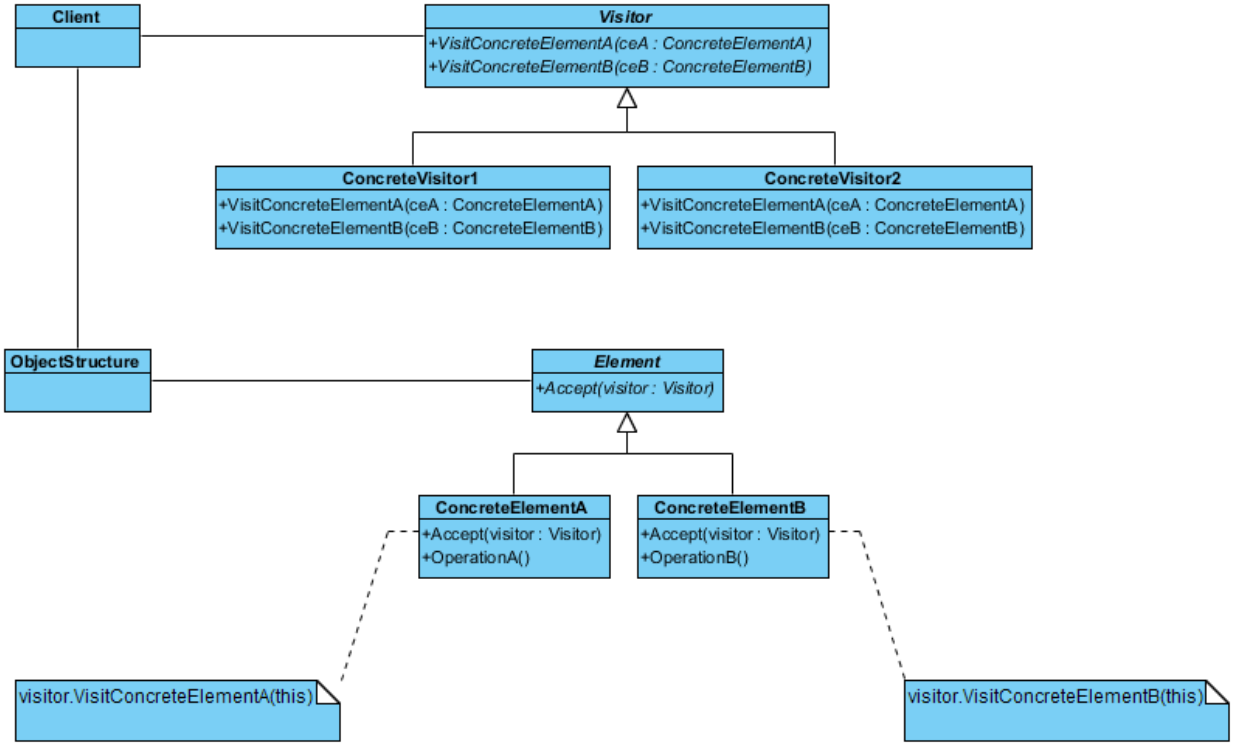
Przy implementacji wzorca kompozyt należy pamiętać o następujących zagadnieniach:

- struktura służąca do przechowywania komponentów może być dowolna, przy jej wyborze należy kierować się wydajnością,
- w przypadku pewnych projektów znaczenie może mieć kolejność elementów podrzędnych Kompozytu. Kontrola nad ich kolejnością może odbywać się z wykorzystaniem wzorca Iterator,
- w przypadku częstego przeszukiwania i (lub) przechodzenia po kompozycie, należy rozważyć wykorzystanie pamięci podręcznej.

3.3.4 Wizytator

Wzorzec obiektowy wizytator (inaczej odwiedzający) należy do obiektowych wzorców operacyjnych i umożliwia wykonywanie działań na elementach struktury obiektów, a co najważniejsze umożliwia dodawanie i modyfikację tychże działań bez zmian wspomnianych elementów [1]. Jest to możliwe dzięki rezygnacji z typowego kodowania działań w metodach danej klasy, na rzecz przekazania referencji do innych klas, które udostępniają pożądane funkcje i umożliwieniu przekazanym klasom definiowania zachowań klas bazowych [4]. Warto jednak nadmienić, że takie podejście nie powinno być stosowane, gdy enkapsulacja jest priorytetem [2].

Strukturę wzorca wizytator przedstawiono na rysunku 8.

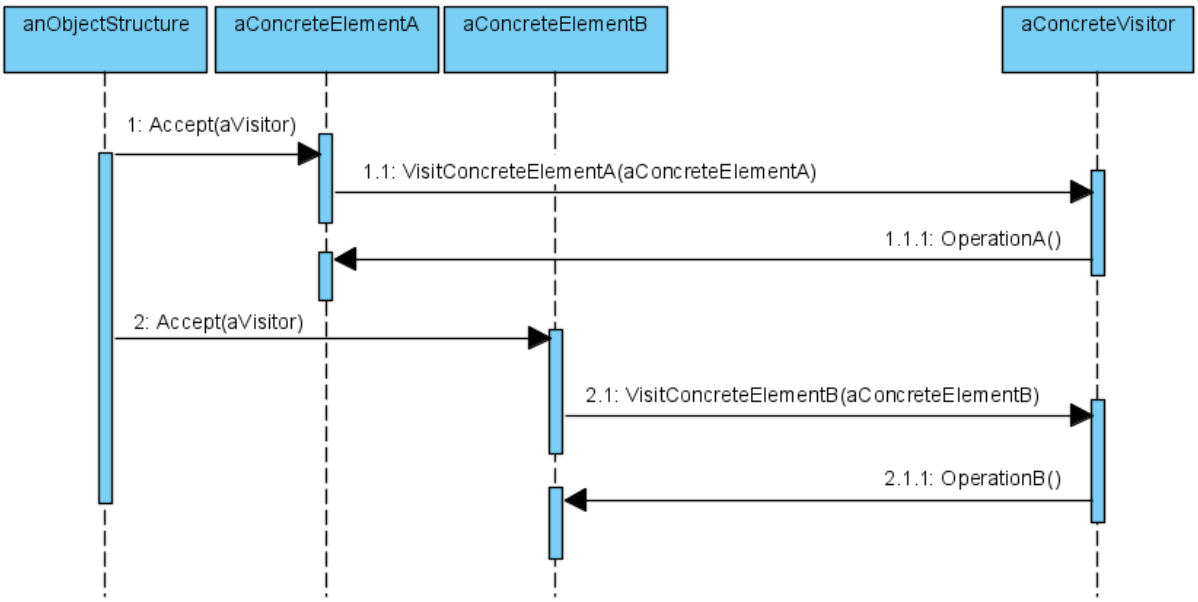


Rysunek 8. Struktura wzorca wizytator. Źródło: [1]

Głównymi jej elementami są [1]:

- Wizytator (**Visitor**) – zawiera deklaracje metod dla każdej z klas dziedziczących po klasie abstrakcyjnej Element. Nazwy metod opisują odwiedzany Konkretny Element. Ponadto należy podkreślić, że Konkretny Element jest przekazywany jako argument odpowiadającej mu metody,

- Konkretny Wizytator (**Concrete Visitor**) – jest miejscem implementacji wszystkich metod zadeklarowanych w Wizytatorze. Dodatkowo pełni rolę kontekstu, co można wykorzystać do przechowywania stanu,
- Element – zawiera deklarację metody `Accept`, której parametr jest typu Wizytator,
- Konkretny Element (**Concrete Element**) – zawiera implementację metody `Accept`, której parametr jest typu Wizytator,
- Struktura Obiekt (**Object Structure**) – reprezentuje pewną strukturę obiektów, może być kolekcją lub na przykład opisanym wcześniej kompozytem.



Rysunek 9. Scenariusz użycia wzorca wizytator. Źródło: [1]

Scenariusz użycia wzorca wizytator przestawiono na rysunku 9 i polega on na odwiedzeniu każdego elementu struktury obiektów przez Klienta za pomocą zainicjowanego wcześniej Konkretnego Wizytatora. Każdy z elementów przyjmuje Konkretnego Wizytatora za pomocą metody `accept`, w której wywołuje odpowiadającą jego klasie metodę `visit`, przekazując do niej samego siebie [1].

Przykładem praktycznego zastosowania wzorca wizytator może być sytuacja, w której chcemy dodać pewne zachowanie do klas zewnętrznej biblioteki czy frameworka, między innymi, gdy:

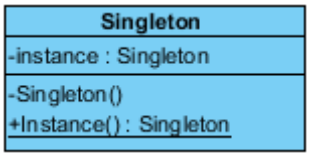
- nie mamy dostępu do kodu źródłowego,
- nie mamy możliwości modyfikacji kodu źródłowego,
- nie chcemy modyfikować kodu źródłowego na przykład ze względu na ograniczenia licencyjne.

W takiej sytuacji wystarczy, że utworzymy podklasę klasy, której zachowanie chcemy rozszerzyć i zaimplementujemy w niej metodę `accept` [5]. Oprócz wspomnianego wcześniej przypadku, gdy enkapsulacja jest priorytetem, wzorzec wizytator nie sprawdzi się, jeżeli struktura obiektów kompozytu czy też kolekcji często się zmienia.

3.3.5 Singleton

Wzorzec projektowy singleton należy do obiektowych wzorców konstrukcyjnych i zapewnia, że klasa będąca singletonem będzie posiadała tylko jedną instancję, do której będzie zapewniony globalny dostęp, przy jednoczesnym braku publicznego konstruktora tejże klasy [1, 4]. Jest on przydatny w sytuacjach, gdy programista potrzebuje dokładnie jednego obiektu danego typu, na przykład: pamięci podręcznej, obiektów zawierających globalną konfigurację, loggerów. Dzięki temu jest w stanie uniknąć problemów takich jak: zbyt duże zużycie zasobów czy niedeterministyczne efekty działania części lub całości programu [2].

Strukturę wzorca singleton przedstawiono na rysunku 10.



Rysunek 10. Struktura wzorca singleton. Źródło: [1]

Głównym jej elementem jest:

- **Singleton** – umożliwia klientom dostęp do jedynego egzemplarza klasy. Może też odpowiadać za jego instancjonowanie, a także zawierać wiele innych zmiennych i metod ogólnego użytku [1, 2].

Użycie wzorca singleton polega na wywołaniu przed klienta statycznej metody `getInstance`, która zapewnia dostęp do jedynej instancji klasy przechowywanej w zmiennej statycznej `uniqueInstance`. Zastosowanie metody `getInstance` pozwala na leniwe instancjonowanie unikalnego egzemplarza [2].

Przy implementacji wzorca singleton głównym wyzwaniem jest zapewnienie powstania dokładnie jednego jego egzemplarza (szczególnie w kontekście wielowątkowości). Cel ten zostaje osiągnięty poprzez utworzenie prywatnego konstruktora Singletonu, a także odpowiednią implementację metody `getInstance`. Jeżeli implementacja metody `getInstance` uwzględnia leniwe instancjonowanie klasy Singleton, istnieje ryzyko utworzenia więcej niż jednego egzemplarza (w przypadku wywołania tejże metody przez dwa lub więcej wątków w zbliżonym czasie). Receptą na to jest synchronizacja metody `getInstance`, której główną wadą jest wysoki koszt wydajnościowy. Jeżeli wydajność jest aspektem krytycznym, można w zamian zastosować [2]:

- wczesne ładowanie unikalnego egzemplarza,
- blokadę z podwójnym zatwierdzeniem.

Singleton, pomimo obecności w książce *Bandy Czterech* [1] i wielu innych, jest często nazywany antywzorcem. Na wielu forach i blogach programiści podają następujące wady używania wzorca singleton [13]:

- ukrywanie zależności w kodzie aplikacji. Używanie singletonów zamiast przekazywanie odpowiednich zależności może świadczyć o głębszych problemach aplikacji, utrudnia refaktoryzację kodu,
- powodowanie problemów w testowaniu. Singleton często powoduje wzrost współzależności modułów, czego efektem może być utrudnienie tworzenia testowych instancji singletonu. Dodatkowo singletony często przechowują stan, co powoduje utratę izolacji testów, co jest niedopuszczalne szczególnie w przypadku testów jednostkowych.

W opinii autora tej pracy powyższe wady bez wątpienia występują, jednak bardzo często spowodowane są niewłaściwym użyciem wzorca singleton. Błędne użycie tego wzorca jest zazwyczaj wymuszone przez inne, głębsze problemy aplikacji, które to pozwala w uproszczony sposób ominąć, co ma swoje późniejsze implikacje między innymi w postaci wymienionych wyżej trudności. Do częstego błędnego użycia singletonu przyczynić mógł się fakt, że jest on wymieniony w książce *Bandy Czterech* [1] na równi z innymi wzorcami, będąc jednocześnie wzorcem rzadkim. Należy pamiętać, że wzorzec ten ma bardzo ograniczony zbiór zastosowań, które zazwyczaj skupiają się wokół problemu rywalizacji o zasoby. Sytuacja odmienia się w przypadku stosowania frameworków, takich jak Spring, używających Inversion of Control. Wtedy to framework zarządza singletonem (co zmniejsza odpowiedzialność programisty) i modyfikuje jego zachowanie odpowiednio do potrzeb, na przykład w testach jednostkowych.

4. Opis narzędzi zastosowanych w pracy

Niniejszy rozdział zawiera opisy technologii i narzędzi wykorzystanych w procesie tworzenia prototypu wtyczki wspomagającej implementację wzorców projektowych.

4.1. Java

Termin Java [14, 15] oznacza zarówno język programowania, jak i platformę. Java jako język programowania jest silnie typowalnym językiem wysokiego poziomu, który opisują następujące słowa kluczowe:

- prosty,
- zorientowany obiektowo,
- rozproszony,
- wielowątkowy,
- dynamiczny,
- niezależny od architektury,
- przenośny,
- wydajny,
- stabilny,
- bezpieczny.

Pliki źródłowe zawierające kod Javy są kompilowane do kodu bajtowego, który jest uruchamiany za pomocą wirtualnej maszyny Javy. Java jako platforma składa się z dwóch komponentów:

- wirtualnej maszyny Javy,
- interfejsu programowania aplikacji Java.

Wirtualna maszyna Javy [16], której głównym zadaniem jest wspomniane wcześniej wykonywanie kodu bajtowego, charakteryzuje się mnogością platform, na których można ją uruchomić. Dzięki temu Java spełnia ideę WORA (Write-Once-Run-Anywhere, czyli „napisz raz, uruchamiaj gdziekolwiek"). Interfejs programowania aplikacji Java udostępnia zestaw bibliotek, które dostarczają programistom ogromną liczbę funkcji a w konsekwencji ułatwiają programowanie w języku Java.

Java jest udostępniana w dwóch dystrybucjach:

- JRE (Java Runtime Environment) – środowisko uruchomieniowe, zawierające między innymi zestaw bibliotek i wirtualną maszynę Javy, niezbędne do uruchomienia aplikacji Java,
- JDK (Java Development Kit) – zestaw narzędzi deweloperskich, zawierający JRE oraz między innymi kompilator kodu Javy, niezbędny do tworzenia aplikacji Java.

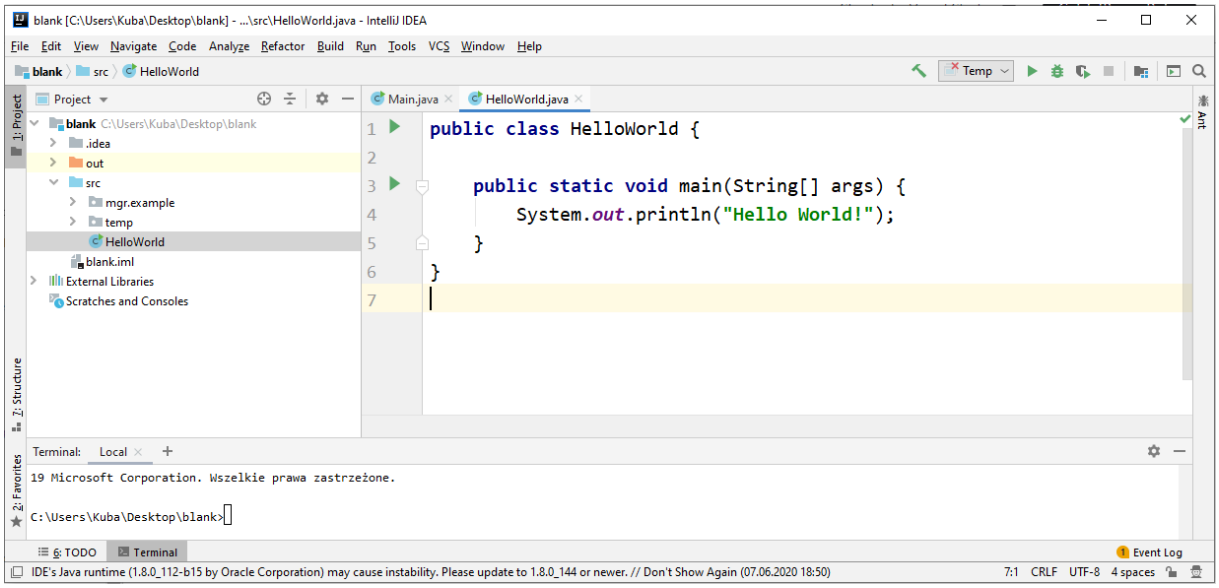
Zastosowanie Javy do realizacji prototypu wtyczki wspomagającej implementację wzorców projektowych, jest ściśle powiązane z wyborem zintegrowanego środowiska programistycznego, którego funkcjonalność będzie ona rozszerzać. Za wykorzystaniem Javy przemawia również kilkuletnie doświadczenie autora niniejszej pracy jako programisty Java.

4.2. IntelliJ IDEA

Intellij IDEA [17] jest to zintegrowane środowisko programistyczne przeznaczone dla języków programowania używających wirtualnej maszyny Javy, oferowane przez firmę JetBrains. Posiada ono szereg narzędzi, które wspomagają pracę programisty i służą między innymi do statycznej analizy kodu, autouzupełniania i wielu innych. Intellij IDEA jest dostępny na platformach Windows, macOS oraz Linux i występuje w trzech edycjach:

- Intellij IDEA Community Edition – wersja społecznościowa,
- Intellij IDEA Ultimate – wersja komercyjna, posiadająca funkcjonalność wersji społecznościowej i dodatkowo między innymi wsparcie technologii webowych i innych języków programowania, takich jak Typescript czy Javascript,
- Intellij IDEA Edu – wersja edukacyjna z wbudowanymi lekcjami języków Java, Kotlin i Scala, a także funkcjami dla nauczycieli.

Bardzo ważną cechą Intellij IDEA w kontekście tej pracy, jest bogata biblioteka wtyczek i możliwość rozszerzenia funkcjonalności Intellij IDEA za ich pomocą. Istotne dla wyboru zintegrowanego środowiska programistycznego wykorzystanego tej pracy jest kilkuletnie doświadczenie autora jako użytkownika IntelliJ IDEA. Okno zintegrowanego środowiska programistycznego IntelliJ IDEA przestawiono na rysunku 11.



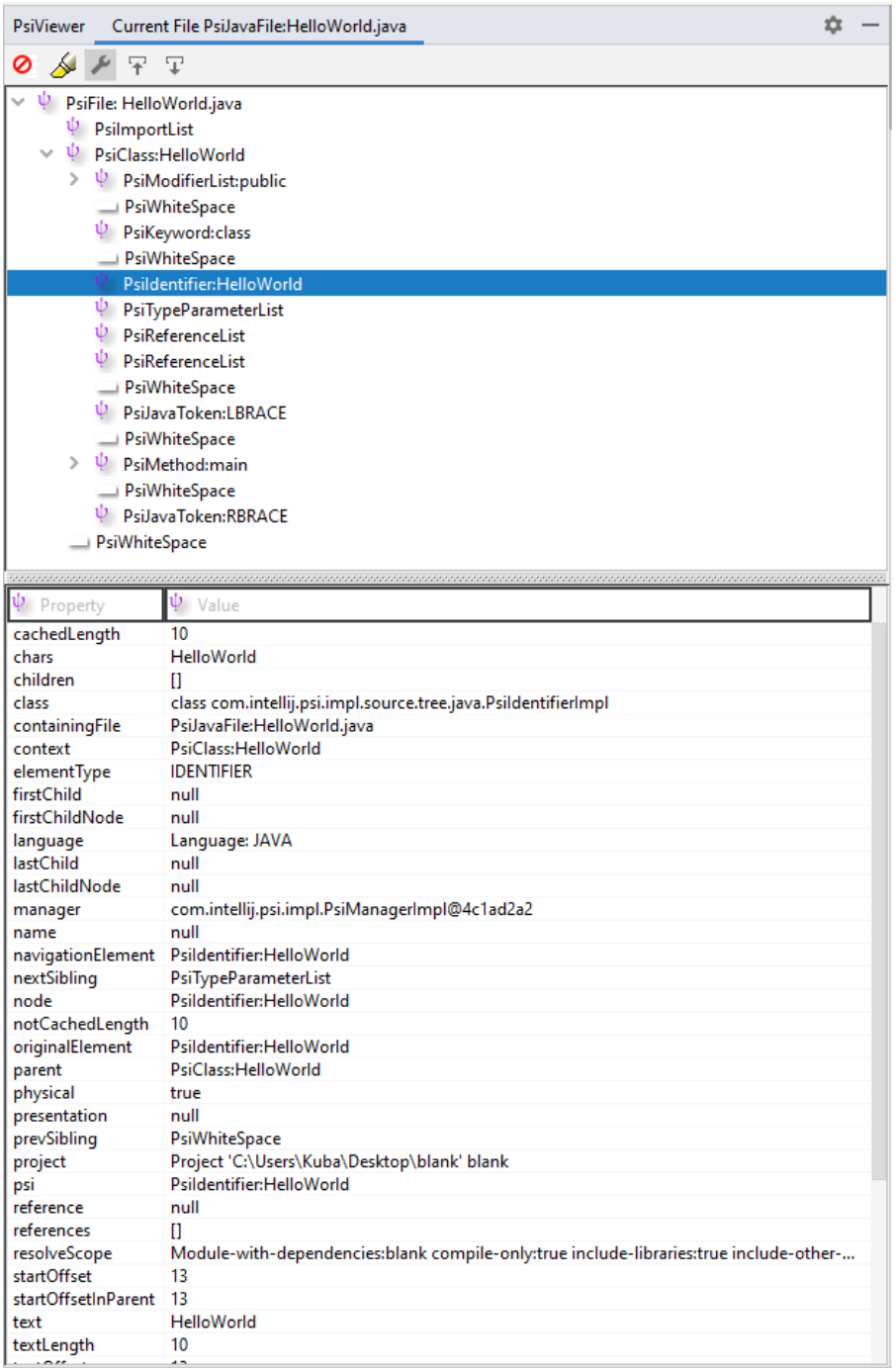
Rysunek 11. Okno zintegrowanego środowiska programistycznego IntelliJ IDEA

4.3. IntelliJ Platform

IntelliJ Platform [18] to otwarta platforma, która jest wykorzystywana przez produkty firmy JetBrains a także Android Studio od firmy Google. Do najważniejszych narzędzi jakie zapewnia ta platforma można zaliczyć:

- edytor obrazów i tekstu razem z abstrakcyjną implementacją autouzupełniania, kolorowania składni i wielu innych,

- interfejs programistyczny aplikacji wspierający funkcje zintegrowanych środowisk programistycznych,
- zestaw narzędzi do tworzenia wysokopoziomowego interfejsu użytkownika, który obejmuje między innymi okna i menu podręczne,
- PSI (Program Structure Interface), czyli interfejs, który analizuje pliki, buduje a następnie indeksuje model kodu, dzięki czemu twórcy aplikacji uzyskują dostęp do wielu funkcji, jak na przykład możliwość poruszania się po strukturze plików i typów. Przykładową strukturę PSI przedstawiono na rysunku 12.



Rysunek 12. Struktura PSI dla klasy `HelloWorld` z rysunku 11

Wtyczki do Intellij IDEA tworzone są w oparciu o platformę IntelliJ Platform a sam proces ich tworzenia może odbywać się w oparciu o [19]:

- Gradle – narzędzie automatyzujące budowę aplikacji (innym znanym narzędziem tego typu jest Apache Maven). W oficjalnej dokumentacji IntelliJ Platform można znaleźć rekomendację użycia Gradle w pracy nad wtyczkami, między innymi ze względu na ułatwioną zmianę parametrów wtyczek czy integrację ze zdalnym repozytorium wtyczek,
- DevKit – wtyczkę wbudowaną do IntelliJ IDEA.

4.4. Git

Git [20] jest to rozproszony system kontroli wersji, który powstał w 2005 roku. Impulsem do jego powstania było cofnięcie pozwolenia na darmowe używanie BitKeepera (innego systemu kontroli wersji) w pracach nad projektem otwartego oprogramowania jakim było jądro Linuxa.

Git [21, 22] w odróżnieniu od większości systemów kontroli wersji korzysta z migawek. Oznacza to, że tworząc obraz stanu projektu przechowuje on informacje na temat stanu wszystkich plików w danym momencie, a nie wyłącznie informacje na temat zbioru plików i zmian jakim one podlegały. Ze względu na to, że Git jest systemem rozproszonym wiele operacji wykonywanych podczas pracy z nim odbywa się z wykorzystaniem lokalnej kopii zdalnego repozytorium. Co ważne, w trakcie zwyczajowej prac Git w większości dodaje nowe informacje do swojej bazy, dlatego też ryzyko trwałego uszkodzenia projektu, bez możliwości przywrócenia jego działającej wersji jest bardzo ograniczone. Ponadto Git dla każdego przechowywanego elementu oblicza sumę kontrolną z wykorzystaniem funkcji skrótu SHA-1, dzięki czemu niemożliwym jest dokonanie zmiany jakiegokolwiek obiektu w taki sposób, aby Git to pominął.

Uzupełnieniem wykorzystania Gita jest użycie usługi hostingowej przeznaczonej dla repozytoriów Gita. Zastosowanie takiej usługi umożliwia stworzenie zdalnego repozytorium, które pełni rolę kopii zapasowej i pozwala na współpracę wielu programistów. W pracy nad prototypem wtyczki do IntelliJ IDEA wspomagającej implementację wzorców projektowych rolę usługi hostingowej pełnił serwis GitHub. Został on wybrany ze względu na to, że jego główne zadanie ograniczało się do pełnienia roli kopii zapasowej a autor tej pracy posiada wcześniejsze doświadczenie w używaniu tej usługi hostingowej.

5. Propozycja wtyczki do IntelliJ IDEA wspomagającej implementację wzorców projektowych

Jako że wiele definicji wzorców projektowych cechuje się powtarzalnością, proponowanym w tej pracy rozwiązaniem mającym usprawnić implementację wzorców jest wtyczka do IntelliJ IDEA, której opis zawarty jest w niniejszym rozdziale.

5.1. Koncepcja

Sam program generujący kod, w tym przypadku języka Java, można zaimplementować pod wieloma postaciami, na przykład jako aplikację desktopową czy webową. Rozwiązanie opisane w tej pracy przyjęło formę wtyczki do IntelliJ IDEA ze względu na:

- fakt, że wtyczka jest „najbliżej" programisty, który korzysta ze zintegrowanego środowiska programistycznego. Rozszerza ona możliwości IDE, jednocześnie będąc bardzo łatwą do zainstalowania (wystarczy ją pobrać z oficjalnego sklepu i kliknąć zainstaluj) i użycia (brak konieczności uruchamiania jakiejkolwiek innej aplikacji, dostępna jest w każdym momencie, w którym programista tworzy kod),
- możliwość korzystania z gotowych platform, przeznaczonych do tworzenia wtyczek do poszczególnych IDE, dzięki którym sam proces powstawania wtyczek jest prostszy i szybszy,
- popularność samego IntelliJ IDEA, z którego korzysta już 62% programistów pracujących z językami opartymi o wirtualną maszynę Javy [23].

Zadaniem wtyczki opisanej w tej pracy jest przede wszystkim wyręczenie programisty z konieczności pisania powtarzalnego, trywialnego, lecz niezbędnego kodu. Cel ten został zrealizowany poprzez konieczność napisania przez programistę określonej dla wybranego wzorca minimalnej implementacji, na podstawie której tworzony jest kod będący jej uzupełnieniem i (lub) jej rozszerzeniem. Wymagania dotyczące minimalnej implementacji są jasno opisane i dostępne dla użytkowników wtyczki. W pewnych przypadkach, oprócz minimalnej implementacji, potrzebne jest podjęcie decyzji dotyczących sposobu i (lub) zawartości implementacji wzorca. Decyzje te są podejmowane przez użytkownika wtyczki za pomocą przeznaczonego do tego interfejsu użytkownika. Ponadto sama wtyczka jest w pełni rozszerzalna i umożliwia wykorzystanie fragmentów lub całości swojej funkcjonalności w innych wtyczkach będących jej rozszerzeniem.

5.2. Architektura i strumień prac

Kod wtyczki opisanej w tej pracy realizuje trzy podstawowe zadania, jakimi są:

- wyświetlanie podstawowych informacji o wybranym wzorcu projektowym wraz z linkami do popularnych stron internetowych zawierających bardziej rozbudowany opis wzorca oraz informacje o sposobie działania wtyczki w kontekście wybranego wzorca projektowego (minimalna implementacja, informacja o kodzie jaki zostanie wygenerowany przez wtyczkę),
- generowanie kodu wybranego wzorca projektowego,
- udostępnianie aplikacyjnego interfejsu programistycznego, dającego możliwość rozszerzenia wtyczki.

W niniejszej pracy architektura wtyczki zostanie przedstawiona zgodnie z podstawowymi zadaniami, które realizuje prototyp.

Zanim jednak to nastąpi konieczne jest wprowadzenie pojęcia akcji jako sposobu na wywołanie kodu wtyczki do IntelliJ IDEA. Implementacja akcji polega na rozszerzeniu klasy `AnAction` i zaimplementowaniu metody `actionPerformed`, która jest wywoływana po uruchomieniu akcji. Tak przygotowaną akcję, należy połączyć z elementem interfejsu użytkownika, który będzie ją wyzwalał, takim jak opcja menu kontekstowego czy element paska narzędzi. Struktura akcji w interfejsie użytkownika przypomina kompozyt, co oznacza, że są one zorganizowane w grupach, które mogą zawierać inne grupy lub pojedyncze elementy.

5.2.1 Readme

Readme, którego polskim tłumaczeniem jest „Czytajto" oznacza plik dołączany do programów, zawierający zazwyczaj informacje na temat licencji, praw autorskich, dokumentację czy instrukcje obsługi programu. Ze względu na to elementy kodu wtyczki, odpowiedzialne za realizację zadania wyświetlania podstawowych informacji o wybranym wzorcu projektowym, zawierają Readme w swoich nazwach.

Klasą bazową modułu Readme jest klasa abstrakcyjna `DesignPatternsReadme`, którą przedstawiono na listingu 2.

Listing 2. Klasa abstrakcyjna `DesignPatternsReadme`

```
package base;

import …

public abstract class DesignPatternsReadme extends AnAction {

   private final String title;
   private final String descriptionFileName;
   private final String linksFileName;

   public DesignPatternsReadme() {
      title = getTitle();
      descriptionFileName = getDescriptionFileName();
      linksFileName = getLinksFileName();
   }

   public abstract String getTitle();

   public abstract String getDescriptionFileName();

   public abstract String getLinksFileName();

   @Override
   public void actionPerformed(@NotNull AnActionEvent anActionEvent) {
      new ReadmeDialogWrapper(title, descriptionFileName, linksFileName).show();
   }
}
```

Rozszerza ona klasę `AnAction` i w metodzie `actionPerformed` wywołuje dialog zawierający informację na temat wybranego wzorca projektowego. Wymaga ona od klas ją rozszerzających implementacji następujących metod:

- `getTitle` – poprzez implementację tej metody definiowany jest tytuł wyświetlanego dialogu,
- `getDescriptionFileName` – poprzez implementację tej metody wskazywany jest plik z katalogu resources zawierający opis wybranego wzorca. Ze względu na to, że dialog do

wyświetlania opisów używa klasy `JLabel` z pakietu Swing, która potrafi interpretować formatowanie HTML, plik wskazany w tej metodzie powinien być plikiem HTML. Dzięki użyciu HTML łatwiej jest odpowiednio ustrukturyzować opisy i zachować ich niezbędną czytelność,

- `getLinksFileName` – poprzez implementację tej metody wskazywany jest plik, zawierający linki do popularnych stron internetowych, zawierających bardziej rozbudowany opis wybranego wzorca i nazwę, która będzie wyświetlana zamiast pełnego adresu URL strony internetowej.

5.2.2 Generator

Klasą bazową modułu Generator jest klasa abstrakcyjna `DesignPatternsGenerator`, będąca rozszerzeniem klasy `AnAction`, którą przedstawiono na listingu 3.

Listing 3. Klasa abstrakcyjna `DesignPatternsReadme`

```
package base;

import …

public abstract class DesignPatternsGenerator extends AnAction {

   private static final long ERROR_BALLOON_FADEOUT_TIME = 7000;

   private final PsiFileValidator psiFileValidator;
   private final EditorValidator editorValidator;

   protected DesignPatternsGenerator() {
      psiFileValidator = new PsiFileValidator(getSourceLanguage());
      editorValidator = new EditorValidator();
   }

   public abstract String getSourceLanguage();

   public abstract DesignPatternsCodeGenerator getCodeGenerator(@NotNull
AnActionEvent anActionEvent);

   @Override
   public final void actionPerformed(@NotNull AnActionEvent anActionEvent) {
      try {
         DesignPatternsCodeGenerator designPatternsCodeGenerator =
getCodeGenerator(anActionEvent);
         WriteCommandAction.runWriteCommandAction(anActionEvent.getProject(),
designPatternsCodeGenerator::generateCode);
         showBalloon(anActionEvent, MessageType.INFO,
designPatternsCodeGenerator.getOnSuccessMessage());
      } catch (ValidationException validationException) {
         showBalloon(anActionEvent, MessageType.ERROR,
validationException.getMessage());
      }
   }

   protected <T> List<T> selectItemsFromCollection(Collection<T> collection,
String listTitlePrefix, String listItemName) {
      SelectionListDialogWrapper<T> selectionListDialogWrapper = new
SelectionListDialogWrapper<>(collection, listTitlePrefix, listItemName);
```

```
      selectionListDialogWrapper.show();

      return selectionListDialogWrapper.getSelectedItemList();
   }

   protected PsiClass getSourcePsiClass(AnActionEvent anActionEvent) {
      PsiFile psiFile = anActionEvent.getData(CommonDataKeys.PSI_FILE);
      Editor editor = anActionEvent.getData(CommonDataKeys.EDITOR);

      psiFileValidator.validate(psiFile);
      editorValidator.validate(editor);

      PsiElement elementAt =
psiFile.findElementAt(editor.getCaretModel().getOffset());
      return PsiTreeUtil.getParentOfType(elementAt, PsiClass.class);
   }

   protected PsiDirectory getTargetPsiDirectory(Project project) {
      final FileChooserDescriptor fileChooserDescriptor =
getFileChooserDescriptor();
      final VirtualFile targetDirectory =
FileChooser.chooseFile(fileChooserDescriptor, project, null);

      if (Objects.isNull(targetDirectory) || !targetDirectory.isDirectory()) {
         Messages.showInfoMessage(InfoMessagesUtil.INCORRECT_SELECTION,
InfoMessagesUtil.TITLE);
      } else {
         return PsiManager.getInstance(project).findDirectory(targetDirectory);
      }

      return null;
   }

   private FileChooserDescriptor getFileChooserDescriptor() {
      final FileChooserDescriptor descriptor =
FileChooserDescriptorFactory.createSingleFolderDescriptor();
      descriptor.setTitle(StringsUtil.SELECT_TARGET_DIRECTORY_TITLE);
      descriptor.setDescription(StringsUtil.SELECT_TARGET_DIRECTORY_DESCRIPTION);

      return descriptor;
   }

   private void showBalloon(AnActionEvent anActionEvent, MessageType messageType,
String message) {
      StatusBar statusBar = WindowManager.getInstance()
            .getStatusBar(anActionEvent.getProject());

      JBPopupFactory.getInstance()
            .createHtmlTextBalloonBuilder(message, messageType, null)
            .setFadeoutTime(messageType.equals(MessageType.ERROR) ?
ERROR_BALLOON_FADEOUT_TIME : -1)
            .createBalloon()
            .show(RelativePoint.getCenterOf(statusBar.getComponent()),
Balloon.Position.atRight);
   }
}
```

Wymaga ona od klas ją rozszerzających implementacji następujących metod:

- `getSourceLanguage` – poprzez implementację tej metody wskazywany jest język jaki został użyty do stworzenia minimalnej implementacji. Jest to informacja niezbędna, ponieważ wtyczka rozpoczyna swoje działanie zmierzające do wygenerowania kodu od walidacji między innymi minimalnej implementacji pod kątem zgodności z zadeklarowanym językiem programowania. Sama metoda i konieczność deklarowania języka programowania została zaimplementowana, aby w przyszłości stworzyć możliwość dodawania kolejnych języków programowania, celem walidacji minimalnej implementacji. Należy pamiętać, że jedynym językiem obsługiwanym w opisanym w tej pracy prototypie jest Java.
- `getCodeGenerator` – w tej metodzie wskazywana jest implementacja interfejsu `DesignPatternsCodeGenerator`, który między innymi wymusza implementację metody `generateCode`, realizującą właściwe generowanie kodu wybranego wzorca.

Klasą bazową, którą rozszerzają wszystkie specyficzne dla konkretnego wzorca projektowego generatory kodu Javy, jest `JavaCodeGenerator`. Klasa ta udostępnia klasom ją rozszerzającym między innymi fabrykę Javowych elementów drzewa PSI dla aktualnego projektu, dzięki której możliwe jest między innymi tworzenie nowych metod czy pól a także serwis katalogów Javowych, umożliwiający na przykład pozyskanie informacji o pakiecie odpowiadającemu wskazanemu katalogowi. Ponadto klasa `JavaCodeGenerator` posiada szereg metod pozwalających na generowanie kodu Javy niepowiązanego z implementacją żadnego konkretnego wzorca projektowego, które umożliwiają między innymi tworzenie getterów czy konstruktorów z parametrami.

W metodzie `actionPerformed` klasy abstrakcyjnej `DesignPatternsGenerator` realizowany jest proces generowania kodu, w którym możemy wyróżnić:

- inicjalizację specyficznego dla wybranego wzorca generatora kodu, w trakcie której odbywa się wspomniana wcześniej walidacja. Dodatkowo właśnie w trakcie inicjalizacji generatora kodu, pozyskiwane są informacje niezbędne do jego poprawnego działania. Są to na przykład decyzje użytkownika dotyczące zawartości implementacji wybranego wzorca, które podejmuje on wykorzystując przeznaczony do tego interfejs czy też informacje uzyskane dzięki analizie minimalnej implementacji (mogą one dostarczać informacji o strukturze i zależnościach pomiędzy elementami minimalnej implementacji),
- właściwy proces generowania kodu, który opiera się o wykorzystanie odpowiednich szablonów, uzupełnianych przez generator kodu określonymi danymi (pozyskiwane są one głównie podczas inicjalizacji generatora kodu). Przykładowy szablon metody `add` generowanej przy wzorcu kompozyt, przedstawiono na listingu 4,
- informowanie użytkownika o błędach jakie wystąpiły w trakcie procesu generowania kodu lub o pomyślnym jego zakończeniu.

Listing 4. Przykładowy szablon metody `add`

```
void add(%s %s) {
    children.add(%s);
}
```

5.2.3 Rozszerzenie

W celu rozszerzenia wtyczki IntelliJ IDEA konieczne jest zdefiniowanie extension point, czyli punktów rozszerzeń. Istnieją dwa rodzaje punktów rozszerzeń [24]:

- interface extension point, który wykorzystywany jest w sytuacjach, gdy chcemy rozszerzyć wtyczkę o kod. Sposób ten polega na zdefiniowaniu interfejsu, który będzie implementowany przez klasę zawartą we wtyczce będącej rozszerzeniem,

- bean extension point, który wykorzystywany jest w sytuacjach, gdy chcemy rozszerzyć wtyczkę o dane. Sposób ten polega na zdefiniowaniu klasy, której instancja zostanie stworzona w oparciu o dane dostarczone przez wtyczkę będącą rozszerzeniem.

Wtyczka opisana w tej pracy definiuje trzy punkty rozszerzeń:

- bean extension point o nazwie `extensionKey`,
- interface extension point o nazwie `extensionAction`,
- interface extension point o nazwie `extensionReadmeAction`.

Rozszerzenie extensionKey ma za zadanie dostarczyć klucz, na podstawie którego wtyczka wspomagająca implementację wzorców projektowych, będzie mogła zidentyfikować pozostałe punkty rozszerzeń, należące do tej samej wtyczki rozszerzającej. Aby było to możliwe konieczne jest odpowiednie wykorzystanie klucza extensionKey jako elementu nazw klas, będących implementacjami interfejsów zdefiniowanych na potrzeby pozostałych punktów rozszerzeń. Punkt rozszerzenia extensionKey jest też wykorzystywany do dynamicznego generowania interfejsu użytkownika, służącego uruchamianiu implementacji punktów rozszerzeń extensionAction i extensionActionReadme zdefiniowanych we wtyczce rozszerzającej.

Rozszerzenie `extensionAction` wymaga zdefiniowania implementacji interfejsu `ExtensionDesignPatternsGenerator`, który przedstawiono na listingu 5.

Listing 5. Interfejs `ExtensionDesignPatternsGenerator`

```
package extension;

import …

/**
 * IMPORTANT!
 * Extension point implementation class naming convention -
{extensionKey}DesignPatternsGenerator.java
 * Example: SingletonDesignPatternsGenerator.java,
FactoryDesignPatternsGenerator.java
 */
public interface ExtensionDesignPatternsGenerator {
   Runnable writeCommandAction(@NotNull AnActionEvent anActionEvent);
}
```

Metoda `writeCommandAction` zwraca instancję implementacji interfejsu `Runnable`, która jest uruchamiana w celu wygenerowania kodu.

Rozszerzenie `extensionReadmeAction` wymaga zdefiniowania implementacji interfejsu `ExtensionDesignPatternsReadme`, który przedstawiono na listingu 6.

Listing 6. Klasa abstrakcyjna `ExtensionDesignPatternsReadme`

```
package extension;

import …
/**
 * IMPORTANT!
 * Extension point implementation class naming convention -
{extensionKey}DesignPatternsReadme.java
 * Example: SingletonDesignPatternsReadme.java, FactoryDesignPatternsReadme.java
 */
public interface ExtensionDesignPatternsReadme {
```

```
    DialogWrapper getReadmeDialogWrapper();
}
```

Metoda `getReadmeDialogWrapper` zwraca obiekt typu `DialogWrapper`, który jest bazową klasą okien dialogowych IntelliJ IDEA. Implementacja `extensionPointAction` ma za zadanie dostarczyć okno dialogowe, które zostanie wyświetlone w celu dostarczenia informacji na temat sposobu działania generatora wtyczki rozszerzającej.

W ramach tej pracy powstała również wtyczka będąca rozszerzeniem wtyczki wspomagającej implementację wzorców projektowych. Wtyczka ta zgodnie ze zdefiniowanymi punktami rozszerzeń posiada:

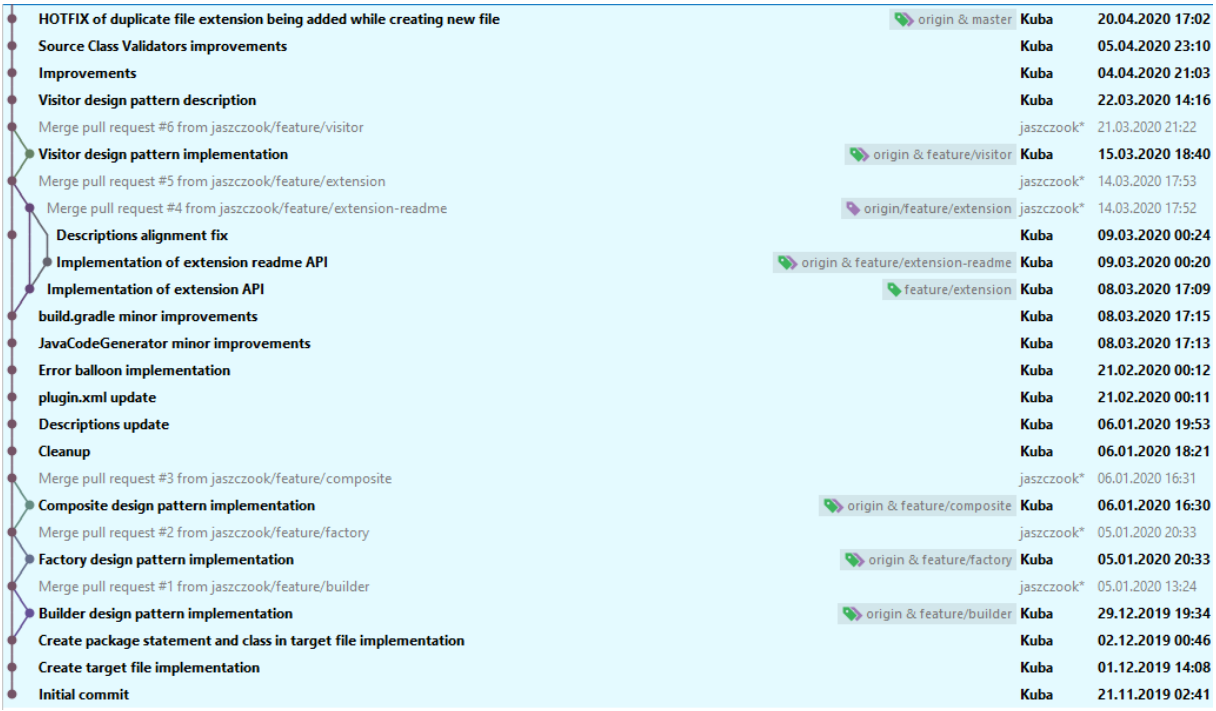
- definicję wartości klucza dla rozszerzenia `extensionKey`,
- implementację interfejsu `ExtensionDesignPatternsGenerator` dla roszerzenia `extensionAction`,
- implementację interfejsu `ExtensionDesignPatternsReadme` dla roszerzenia `extensionReadmeAction`.

Aby zdefiniowanie punktów rozszerzeń było możliwe, niezbędna była odpowiednia konfiguracja wtyczki rozszerzającej. W tym celu w pliku konfiguracyjnym build.gradle znajduje się odpowiednie wskazanie na plik jar uprzednio zbudowanej wtyczki wspomagającej implementację wzorców projektowych.

5.2.4 Strumień prac

Jako że jednym z narzędzi wykorzystywanych w do tworzenia wtyczki wspomagającej implementację wzorców projektowych jest narzędzi kontroli wersji Git, niezbędne było wybranie sposobu pracy z gałęziami Gita.

Branching oznacza duplikację obiektów będących pod kontrolą systemu kontroli wersji, dzięki czemu możliwe jest zrównoleglenie prac. Istnieje co najmniej kilka sposobów na pracę z gałęziami, jednak mają one zastosowanie przede wszystkim przy tworzeniu aplikacji w wieloosobowych zespołach, gdzie aplikacja ma określony wieloma wymaganiami cykl życia. Wpływ na wybranie odpowiedniego strumienia prac ma na przykład plan kolejnych wydań czy konieczność posiadania wielu wersji kodu produkcyjnego. Ze względu na fakt, że przy rozwoju wtyczki pracował tylko jeden programista, strategia pracy z gałęziami gita ograniczyła się do tworzenia feature branchy, czyli gałęzi dedykowanych rozwojowi określonych funkcji. Po zakończeniu rozwoju danej funkcji, feature branch jest scalany z główną gałęzią master. Dzięki temu, przez cały czas trwania pracy nad wtyczką, kod dostępny na głównej gałęzi był stabilny i poprawnie działający. Przy tworzeniu prototypu przedstawionego w niniejszej pracy, feature branche były tworzone tylko przy rozwoju dużych, rozbudowanych (w odniesieniu do skali wtyczki) funkcji. W przypadku konieczności wprowadzenia krytycznych poprawek czy pomniejszych udoskonaleń trafiały one bezpośrednio do głównej gałęzi master. Rysunek 13 przedstawia fragment historii pracy nad prototypem.

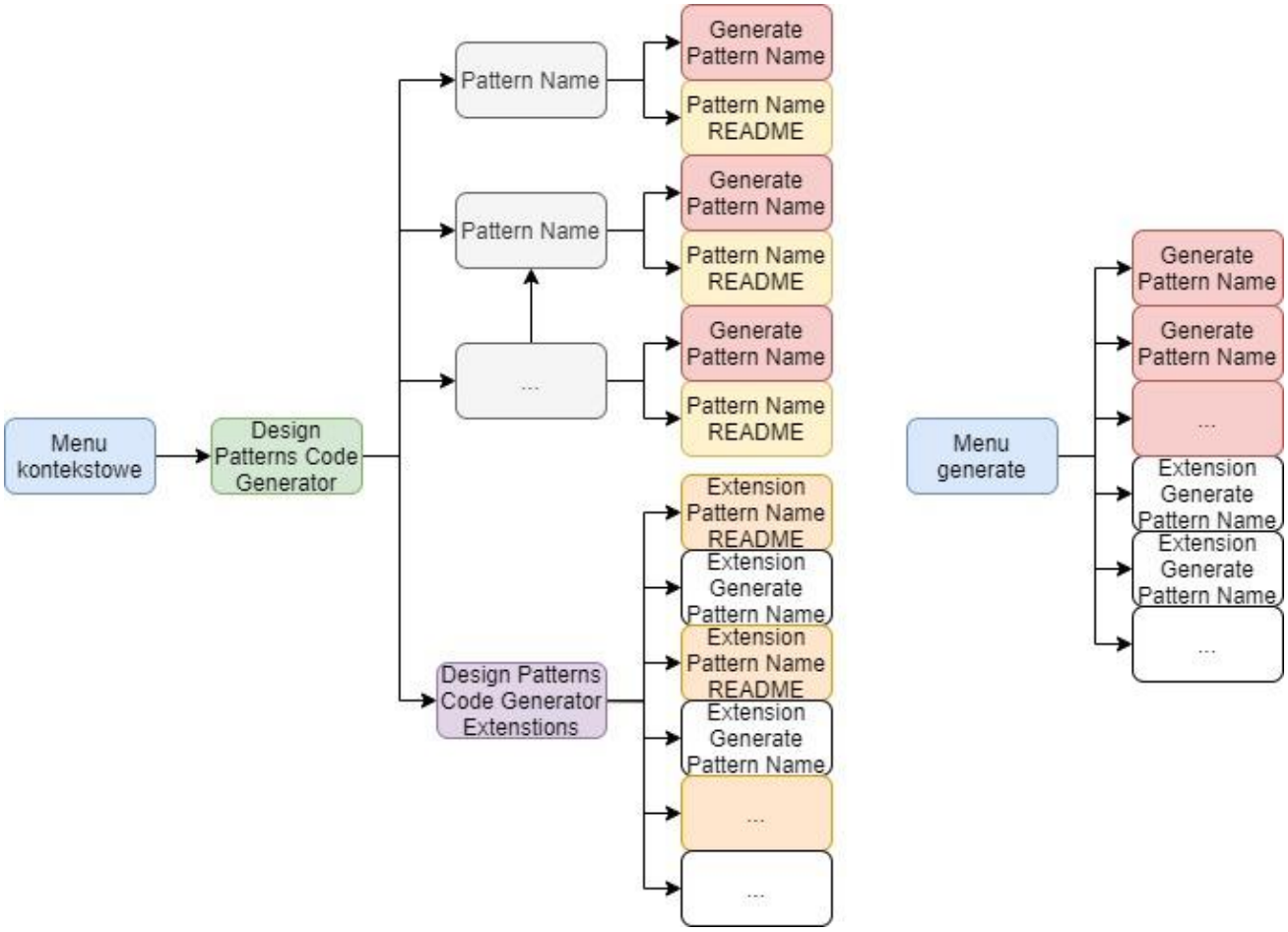


Rysunek 13. Fragment historii pracy nad wtyczką

5.3. Interfejs użytkownika

Jedną z największych zalet wtyczki do zintegrowanego środowiska programistycznego jest wspomniana wcześniej „bliskość” programisty. Ze względu na to konieczne było zaimplementowanie interfejsu użytkownika, który swoją prostotą i łatwą dostępnością uwydatni tę zaletę.

Schemat interfejsu użytkownika przedstawiono na rysunku 14.

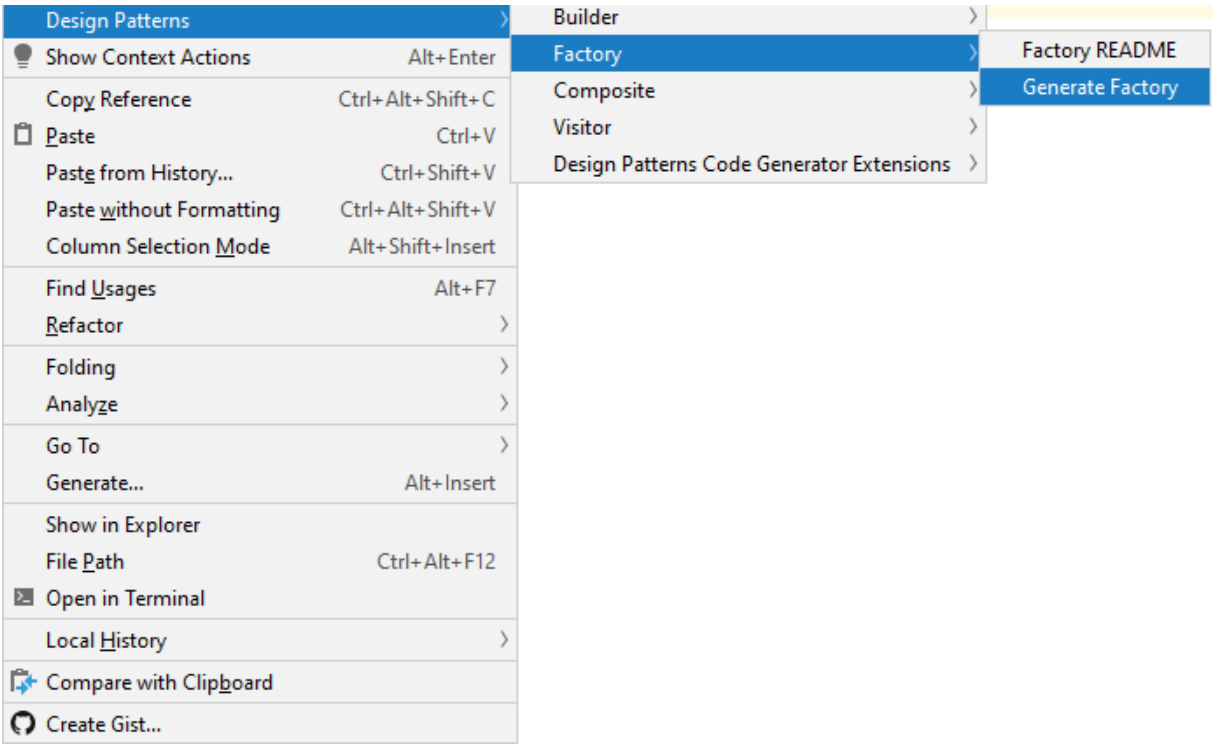


Rysunek 14. Diagram interfejsu użytkownika

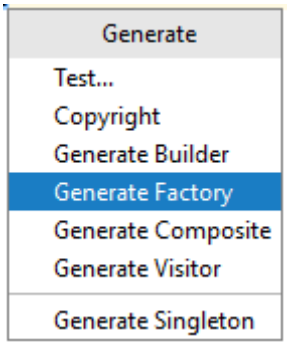
Głównymi jego elementami są:

- Menu kontekstowe – menu dostępne w oknie edytora pod prawym przyciskiem myszy, które zaprezentowano na rysunku 15,
- Design Patterns Code Generator – grupa akcji, obejmująca wszystkie udostępnione przez wtyczkę akcje,
- Menu generate – jest to domyślnie zdefiniowana grupa akcji, dostępna również pod skrótem klawiaturowym, którą zaprezentowano na rysunku 16. Obejmuje między innymi takie funkcje IDE jak generowanie getterów i seterów czy metody `toString`. Jako że głównym wtyczka wspomagająca implementację wzorców projektowych jest generowanie kodu, menu generate jest naturalnym miejscem, w którym taka funkcja powinna się znaleźć,
- Pattern Name – grupa akcji, obejmująca akcje dla jednego wzorca projektowego,
- Pattern Name README – akcja wywołująca funkcję Readme dla wybranego wzorca projektowego,
- Generate Pattern Name – akcja wywołująca funkcję Generate dla wybranego wzorca projektowego,
- Design Patterns Code Generator Extensions – grupa akcji, której zawartość jest generowana dynamicznie w zależności od wtyczek rozszerzających,
- Extension Pattern Name README – akcja wywołująca funkcję Readme dla wybranego wzorca projektowego, której implementacja znajduje się we wtyczce rozszerzającej,

- Extension Generate Pattern Name – akcja wywołująca funkcję Generate dla wybranego wzorca projektowego, której implementacja znajduje się we wtyczce rozszerzającej.

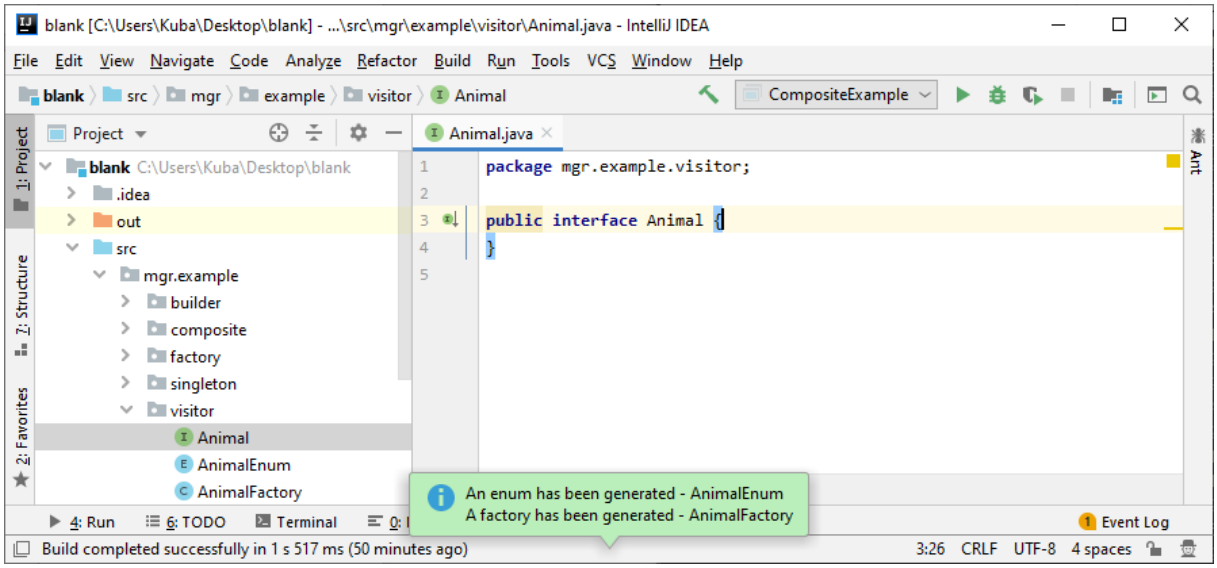


Rysunek 15. Menu kontekstowe z rozwiniętą opcją generowania kodu wzorca metoda wytwórcza



Rysunek 16. Menu generate z zaznaczoną opcją generowania kodu metoda wytwórcza

Pozostałe elementy interfejsu, takie jak listy wyboru opcji, to dobrze znane okna dialogowe, bazujące na bibliotece Swing. Do wyświetlania informacji na temat ewentualnych błędów, na przykład walidacji, czy też informowania o poprawnym zakończeniu gracy generatora, wykorzystano popularne w IntelliJ IDEA dymki, które zaprezentowano na rysunku 17.



Rysunek 17. Okno edytora z dymkiem zawierającym komunikat o poprawnym zakończeniu pracy generatora dla wzorca metoda wytwórcza

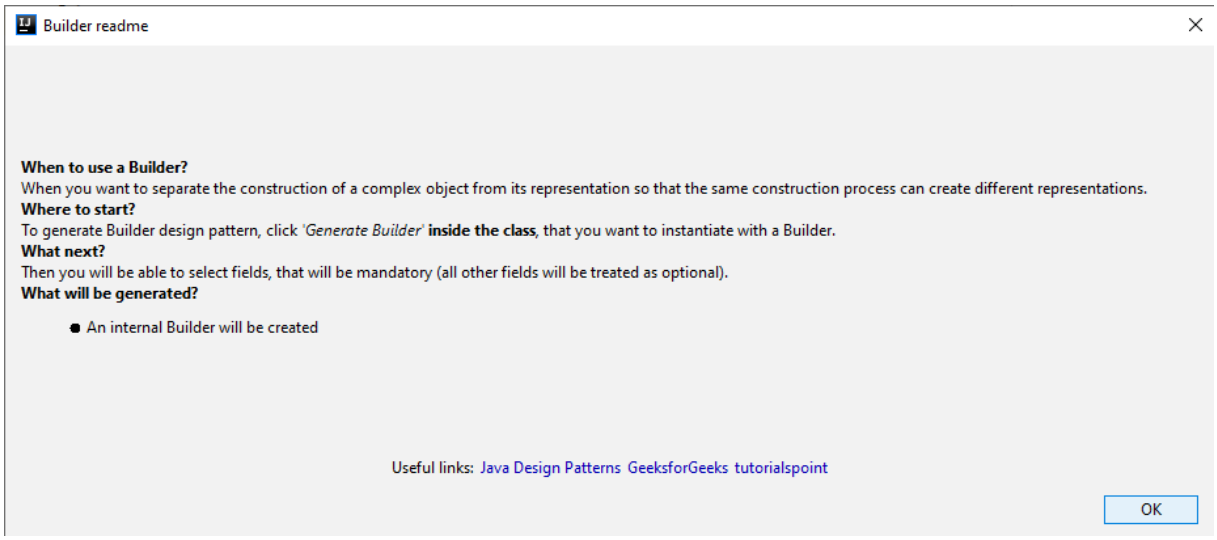
6. Prototyp wtyczki do IntelliJ IDEA wspomagającej implementację wzorców projektowych

Ten rozdział zawiera prezentację działania wtyczki wspomagającej implementację wzorców projektowych na przykładach zaimplementowanych wzorców, którymi są:

- budowniczy,
- metoda wytwórcza,
- kompozyt,
- wizytator,
- singleton.

6.1. Budowniczy

Wywołanie funkcji README wtyczki do implementacji wzorców projektowych dla wzorca budowniczy, skutkuje wyświetleniem okna dialogowego przedstawionego na rysunku 18.



Rysunek 18. Okno README dla wzorca budowniczy

Okno to zawiera takie informacje jak:

- krótki opis wskazujący w jakiej sytuacji należy zastosować wzorzec budowniczy,
- opis minimalnej implementacji,
- krótki opis wyborów jakich należy dokonać w celu wygenerowania wzorca,
- krótki opis wygenerowanego kodu,
- linki do witryn internetowych zawierających więcej informacji na temat wzorca budowniczy.

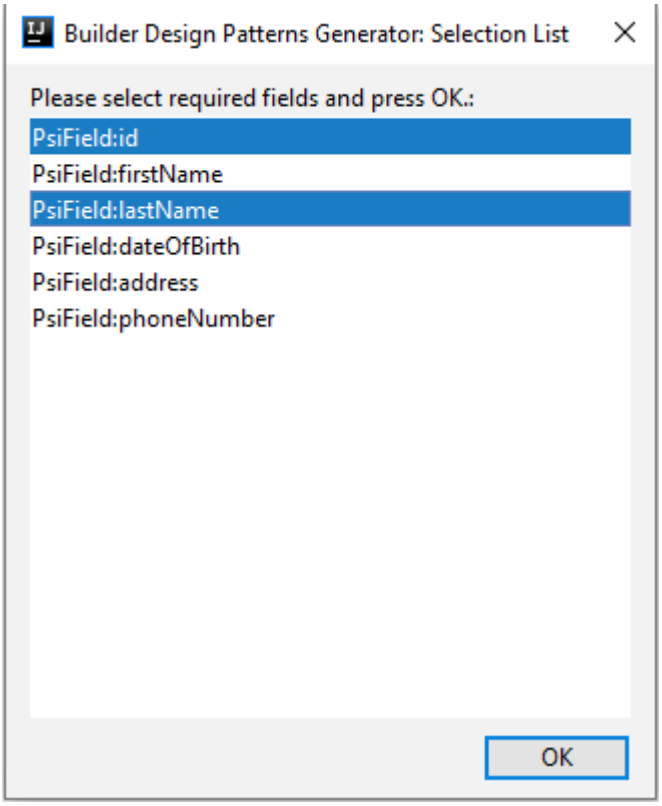
Aby móc uruchomić generator wzorca budowniczy, należy w ramach minimalnej implementacji stworzyć klasę, której instancje chcemy tworzyć wykorzystując budowniczego. Klasa ta będzie pełniła

rolę produktu. Oczywiście dobrze, gdyby klasa ta miała co najmniej jedno pole, chociaż wskazana jest większa ich liczba. Przykład minimalnej implementacji dla wzorca budowniczy przedstawiono na listingu 7.

Listing 7. Przykład minimalnej implementacji dla wzorca budowniczy

```
package mgr.example.builder;

import java.util.Date;

public class Client {
    private long id;
    private String firstName;
    private String lastName;
    private Date dateOfBirth;
    private String address;
    private String phoneNumber;
}
```

Następnie po uruchomieniu generatora użytkownik zostanie poproszony o wskazanie z listy tych pól, które będą obowiązkowe, czyli będą one parametrem konstruktora budowniczego. Okno wyboru obowiązkowych pól dla wzorca budowniczy zaprezentowano na rysunku 19.



Rysunek 19. Okno wyboru obowiązkowych pól dla wzorca budowniczy

Po dokonaniu wyboru, wygenerowany zostaje kod, który zaprezentowano na listingu 8.

Listing 8. Wygenerowany kod dla wzorca budowniczy

```
package mgr.example.builder;

import java.util.Date;

public class Client {
    private final long id;
    private final String firstName;
    private final String lastName;
    private final Date dateOfBirth;
    private final String address;
    private final String phoneNumber;

    private Client(ClientBuilder clientBuilder) {
        this.id = clientBuilder.id;
        this.firstName = clientBuilder.firstName;
        this.lastName = clientBuilder.lastName;
        this.dateOfBirth = clientBuilder.dateOfBirth;
        this.address = clientBuilder.address;
        this.phoneNumber = clientBuilder.phoneNumber;
    }

    public long getId() {
        return id;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public Date getDateOfBirth() {
        return dateOfBirth;
    }

    public String getAddress() {
        return address;
    }

    public String getPhoneNumber() {
        return phoneNumber;
    }



    public static class ClientBuilder {
        private final long id;
        private final String lastName;
        private String firstName;
        private Date dateOfBirth;
        private String address;
        private String phoneNumber;
```

```
        public ClientBuilder(long id, String lastName) {
            this.id = id;
            this.lastName = lastName;
        }

        public ClientBuilder withFirstName(String firstName) {
            this.firstName = firstName;
            return this;
        }

        public ClientBuilder withDateOfBirth(Date dateOfBirth) {
            this.dateOfBirth = dateOfBirth;
            return this;
        }

        public ClientBuilder withAddress(String address) {
            this.address = address;
            return this;
        }

        public ClientBuilder withPhoneNumber(String phoneNumber) {
            this.phoneNumber = phoneNumber;
            return this;
        }

        public Client build() {
            return new Client(this);
        }
    }
}
```

Wygenerowany kod zawiera:

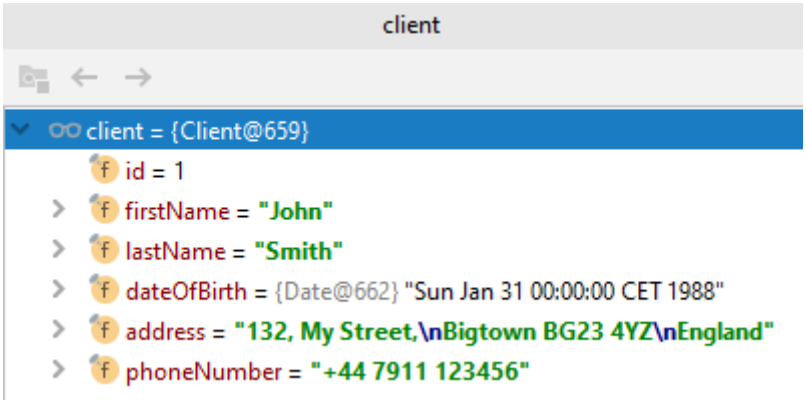
- prywatny konstruktor klasy produktu – ze względu na fakt, że chcemy, aby produkt był instancjonowany przy użyciu budowniczego, nie może on posiadać innego konstruktora niż prywatny,
- gettery wszystkich pól produktu
- wewnętrzną statyczną klasę budowniczego – fakt, że budowniczy jest klasą wewnętrzną a nie klasą w osobnym pliku, zapewnia budowniczemu dostęp do prywatnego konstruktora produktu. Ponadto, jako że budowniczy jest klasą wewnętrzną, w omawianej implementacji wzorca budowniczy zrezygnowano z klasy Nadzorcy, przez co cała odpowiedzialność za proces tworzenia instancji produktu przechodzi na klienta, który to powinien mieć świadomość jak wspomniany proces poprawnie przeprowadzić,
- prywatne pola budowniczego – odpowiadają one polom produktu,
- publiczny konstruktor budowniczego – jego parametrami są pola wybrane przez użytkownika. Wymusza to na kliencie podanie wartości wspomnianych pól, przez co utworzenie produktu bez tych wartości nie jest możliwe,
- publiczne metody `with` – ich celem jest ustawienie wartości pól budowniczego. Ponadto zwracają one `this`, czyli odniesienie do obecnego obiektu, dzięki czemu uzyskujemy płynny interfejs pozwalający na łączenie metod,
- publiczna metoda `build` – tworzy ona instancję produktu, wykorzystując odpowiednio ówcześnie zainicjowanego budowniczego.

Utworzony przez wtyczkę kod, można użyć w sposób zaprezentowany na listingu 9.

Listing 9. Przykład użycia wygenerowanego budowniczego

```
package mgr.example.builder;

import java.text.ParseException;
import java.text.SimpleDateFormat;

public class BuilderExample {
    public static void main(String[] args) throws ParseException {
        Client client = new Client.ClientBuilder(1, "Smith")
                .withFirstName("John")
                .withDateOfBirth(new
SimpleDateFormat("dd/MM/yyyy").parse("31/01/1988"))
                .withAddress("132, My Street,\n" +
                        "Bigtown BG23 4YZ\n" +
                        "England")
                .withPhoneNumber("+44 7911 123456")
                .build();
    }
}
```

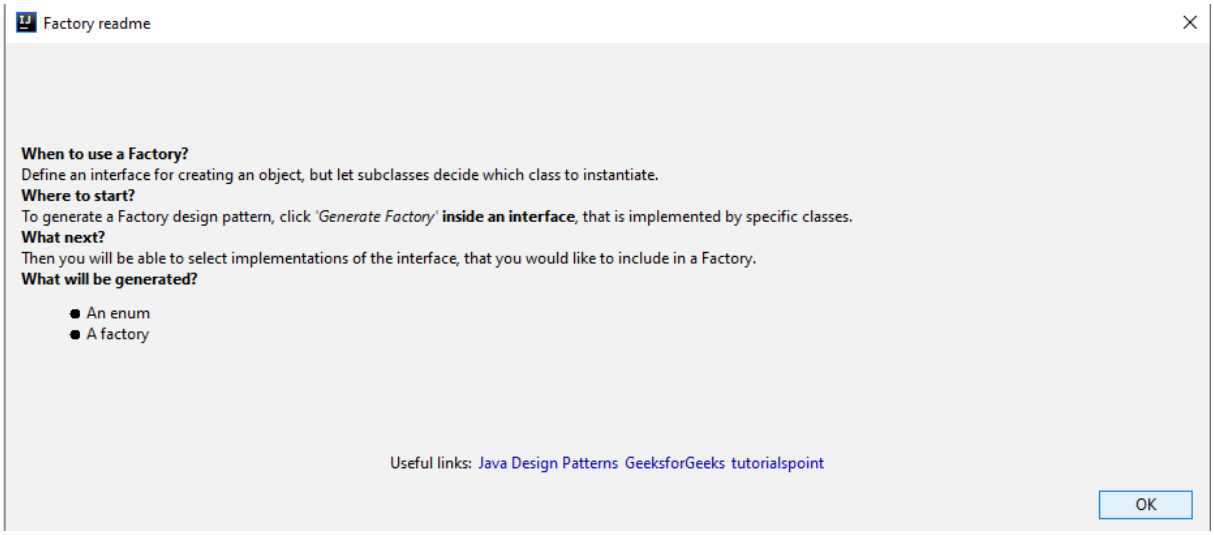
Obiekt utworzony w przykładowym użyciu wzorca budowniczy przedstawiono na rysunku 20.



Rysunek 20. Obiekt utworzony w przykładowym użyciu wzorca budowniczy

6.2. Metoda wytwórcza

Wywołanie funkcji README wtyczki do implementacji wzorców projektowych dla wzorca metoda wytwórcza skutkuje wyświetleniem okna dialogowego przedstawionego na rysunku 21.



Rysunek 21. Okno README dla wzorca metoda wytwórcza

Okno to zawiera następujące informacje:

- krótki opis wskazujący w jakiej sytuacji należy zastosować wzorzec metoda wytwórcza,
- opis minimalnej implementacji,
- krótki opis wyborów jakich należy dokonać w celu wygenerowania wzorca,
- krótki opis wygenerowanego kodu,
- linki do witryn internetowych zawierających więcej informacji na temat wzorca metoda wytwórcza.

Aby móc uruchomić generator wzorca metoda wytwórcza, należy w ramach minimalnej implementacji stworzyć interfejs, którego implementacje chcemy tworzyć wykorzystując metodę wytwórczą. Interfejs ten będzie pełnił rolę produktu, a jego implementacje będą pełniły rolę konkretnych produktów. Przykład minimalnej implementacji dla wzorca metoda wytwórcza przedstawiono na listingach 10, 11 oraz 12.

Listing 10. Przykład minimalnej implementacji dla wzorca metoda wytwórcza – interfejs `User`

```
package mgr.example.factory;

public interface User {
    void logIn();
    void withdrawMoney();
    void logOut();
}
```

Listing 11. Przykład minimalnej implementacji dla wzorca metoda wytwórcza – klasa `BusinessUser`

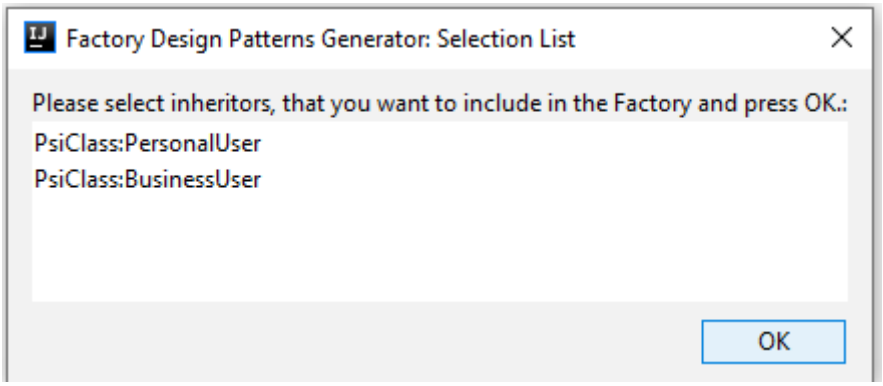
```
package mgr.example.factory;

public class BusinessUser implements User {
    @Override
    public void logIn() {
        System.out.println("Logging in business user.");
    }
```

```
    @Override
    public void withdrawMoney() {
        System.out.println("Withdrawing money from company account.");
    }

    @Override
    public void logOut() {
        System.out.println("Logging out business user.");
    }
}
```

Listing 12. Przykład minimalnej implementacji dla wzorca metoda wytwórcza – klasa `PersonalUser`

```
package mgr.example.factory;

public class PersonalUser implements User {
    @Override
    public void logIn() {
        System.out.println("Logging in personal user.");
    }

    @Override
    public void withdrawMoney() {
        System.out.println("Withdrawing money from private account.");
    }

    @Override
    public void logOut() {
        System.out.println("Logging out personal user.");
    }
}
```

Następnie po uruchomieniu generatora użytkownik zostanie poproszony o wskazanie z listy tych Konkretnych Produktów, które będą mogły być instancjonowane przy użyciu wygenerowanej metody wytwórczej. Okno wyboru konkretnych produktów zaprezentowano na rysunku 22.



Rysunek 22. Okno wyboru Konkretnych Produktów

Po dokonaniu wyboru, wygenerowany zostaje kod, który przedstawiono na listingach 13 i 14.

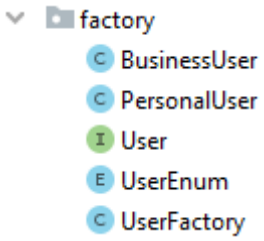
Listing 13. Wygenerowany kod dla wzorca metoda wytwórcza – typ wyliczeniowy `UserEnum`

```
package mgr.example.factory;

public enum UserEnum {BUSINESS_USER, PERSONAL_USER}
```

Listing 14. Wygenerowany kod dla wzorca metoda wytwórcza – klasa `UserFactory`

```
package mgr.example.factory;

public class UserFactory {
    public User getUser(UserEnum userEnum) {
        switch (userEnum) {
            case PERSONAL_USER: {
                return new PersonalUser();
            }
            case BUSINESS_USER: {
                return new BusinessUser();
            }
            default: {
                throw new IllegalArgumentException("Unknown ENUM:
".concat(userEnum.toString()));
            }
        }
    }
}
```

Strukturę pakietu po wygenerowaniu kodu dla wzorca metoda wytwórcza zaprezentowano na rysunku 23.



Rysunek 23. Struktura pakietu po wygenerowaniu kodu dla wzorca metoda wytwórcza

Wygenerowany kod zawiera:

- typ wyliczeniowy – zawierający wszystkie wybrane przez użytkownika implementacje interfejsu będącego częścią minimalnej implementacji. Wygenerowany typ wyliczeniowy zapobiega sytuacji, w której Fabryka nie potrafi stworzyć odpowiedniej instancji implementacji interfejsu, ze względu na brak dopasowania argumentu przekazanego do metody odpowiedzialnej za utworzenie wspomnianej instancji do konkretnej klasy będącej pożądaną implementacją interfejsu,
- klasę fabryki – posiadającą metodę, która zwraca implementację interfejsu bazując na argumencie typu wyliczeniowego przekazanym do tej metody. Ciało tej metody zawiera blok `switch`, który odpowiada za wybór właściwej implementacji. Alternatywą mogłoby być użycie konstrukcji `if else`. W przypadku braku dopasowania, co pomimo zastosowania typu wyliczeniowego jest możliwe na przykład na skutek zmian w kodzie wprowadzonych przez użytkownika już po użyciu wtyczki, rzucany jest wyjątek.

Utworzony przez wtyczkę kod, można użyć w sposób zaprezentowany na listingu 15.

Listing 15. Przykład użycia wygenerowanej metody wytwórczej

```
package mgr.example.factory;

public class FactoryExample {
    public static void main(String[] args) {
        User user = new UserFactory().getUser(UserEnum.BUSINESS_USER);
        user.logIn();
        user.withdrawMoney();
        user.logOut();
    }
}
```

Wynik działania programu prezentującego przykład użycia przedstawiono na listingu 16.

Listing 16. Wynik działania programu prezentującego przykład użycia

```
Logging in business user.
Withdrawing money from company account.
Logging out business user.
```

Łatwo zauważyć, że implementacja wzorca metoda wytwórcza utworzona przez wtyczkę nawiązuje do prostej fabryki a nie do metody wytwórczej. Autor pracy podjął taką decyzję ze względu na fakt, że prosta fabryka bywa uznawana za idiom programistyczny, a nie pełnoprawny wzorzec projektowy, jest prostsza do zautomatyzowania, a jej zautomatyzowana implementacja będzie bardziej odpowiadała faktycznej implementacji pożądanej przez użytkownika, przez co ten nie będzie musiał wprowadzać późniejszych dostosowań i poprawek. Dodatkowo w opinii autora idiom programistyczny jakim jest prosta fabryka może być częściej wykorzystywany przy użyciu wtyczki niż metoda fabryczna, ponieważ metoda fabryczna jest wzorcem przeznaczonym do stosowania w bardziej specyficznych sytuacjach.

6.3. Kompozyt

Wywołanie funkcji README wtyczki do implementacji wzorców projektowych dla wzorca kompozyt, skutkuje wyświetleniem okna dialogowego, przedstawionego na rysunku 24.

Rysunek 24. Okno README dla wzorca kompozyt

Okno to zawiera takie informacje jak:

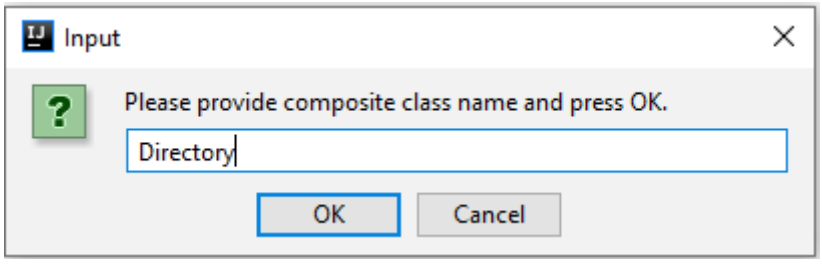
- krótki opis wskazujący w jakiej sytuacji należy zastosować wzorzec kompozyt,
- opis minimalnej implementacji,
- krótki opis wygenerowanego kodu,
- linki do witryn internetowych zawierających więcej informacji na temat wzorca kompozyt.

Aby móc uruchomić generator wzorca kompozyt, należy w ramach minimalnej implementacji stworzyć klasę abstrakcyjną. Będzie ona pełniła rolę Komponentu, posługując się nomenklaturą schematu opisanego w części teoretycznej niniejszej pracy. Przykład minimalnej implementacji dla wzorca kompozyt przedstawiono na listingu 17.

Listing 17. Przykład minimalnej implementacji dla wzorca kompozyt

```
package mgr.example.composite;

public abstract class File {
}
```

Następnie po uruchomieniu generatora użytkownik zostanie poproszony o nazwy klasy, która będzie pełniła rolę Kompozytu (jeżeli użytkownik nie poda żadnej nazwy, zostanie jej nadana domyślna). Okno wyboru nazwy Komponentu zaprezentowano na rysunku 25.



Rysunek 25. Okno wyboru nazwy Kompozytu

Po dostarczeniu minimalnej implementacji i uruchomieniu odpowiedniej opcji wtyczki, wygenerowany zostaje kod, który przedstawiono na listingach 18 i 19.

Listing 18. Wygenerowany kod dla wzorca kompozyt – klasa abstrakcyjna `File`

```
package mgr.example.composite;

public abstract class File {

    public void add(File file) {
        throw new UnsupportedOperationException();
    }

    public int count() {
        throw new UnsupportedOperationException();
    }

    public File getChild(int index) {
        throw new UnsupportedOperationException();
    }
```

```
    public void remove(File file) {
        throw new UnsupportedOperationException();
    }
}
```

Listing 19. Wygenerowany kod dla wzorca kompozyt – klasa `Directory`

```
package mgr.example.composite;

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

public class Directory extends File {
    private List<File> children = new ArrayList<>();

    public void add(File file) {
        children.add(file);
    }

    public int count() {
        return children.size();
    }

    public void remove(File file) {
        children.remove(file);
    }

    public File getChild(int index) {
        return children.get(index);
    }
}
```

Wygenerowany kod zawiera:

- domyślne implementacje metod typowych dla kompozytu, znajdujące się w klasie Komponentu, Metodami typowymi dla kompozytu są metody:
    - `add`,
    - `count`,
    - `remove`,
    - `getChild`.

  Domyślna implementacja zawiera rzucenie wyjątku, ze względu na fakt, że nie każda z metod zawartych w Komponencie, nie tylko wygenerowanych przez wtyczkę, ale też napisanych przez programistę, ma sens dla Liścia i (lub) Kompozytu. Metody, które wspomniany sens mają powinny zostać we właściwy sposób zaimplementowany. W przeciwnym przypadku, kiedy implementacja danej metody jest pozbawiona sensu dla Liścia lub Kompozytu, brak implementacji danej metody powoduje odziedziczenie jej implementacji po klasie Komponentu, która powinna rzucić wyjątek, co też czyni [2],
- klasę Kompozytu – rozszerza ona klasę Komponentu,
- prywatne pole będące kolekcją dzieci, znajdujące się w Kompozycie – do implementacji kolekcji została użyta lista oparta o tablicę, ze względu uniwersalność i na bogaty interfejs dostępu do elementów, które zawiera. Możliwe jest wykorzystanie w tym miejscu innej struktury danych, takiej jak tablica, drzewo czy inny rodzaj listy. Wybór powinien być podyktowany wydajnością w kontekście przewidywanych scenariuszy użycia. Dopuszczalna

jest także sytuacja, w której kolekcja dzieci znajduje się w klasie Komponentu. Takie rozwiązanie sprawdzi się lepiej jedynie w sytuacjach, gdy w kompozycie znajdzie się względnie mało elementów podrzędnych [1],

- publiczną implementację domyślnych metod typowych dla kompozytu.

Utworzony przez wtyczkę kod, można użyć w sposób zaprezentowany na listingach 20, 21, 22, 23 i 24.

Listing 20. Przykład użycia wzorca kompozyt, którego implementacja zawiera wygenerowany przez wtyczkę kod – klasa abstrakcyjna `File`

```
package mgr.example.composite;

public abstract class File {

    public void add(File file) {
        throw new UnsupportedOperationException();
    }

    public int count() {
        throw new UnsupportedOperationException();
    }

    public File getChild(int index) {
        throw new UnsupportedOperationException();
    }

    public void remove(File file) {
        throw new UnsupportedOperationException();
    }

    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

Listing 21. Przykład użycia wzorca kompozyt, którego implementacja zawiera wygenerowany przez wtyczkę kod – klasa `Image`

```
package mgr.example.composite;

public class Image extends File {
    String name;
    int height;
    int width;

    public Image(String name, int height, int width) {
        this.name = name;
        this.height = height;
        this.width = width;
    }

    public void print() {
        System.out.println(toString());
    }

    @Override
```

```
    public String toString() {
        return "Image{" +
                "name='" + name + '\'' +
                ", height=" + height +
                ", width=" + width +
                '}';
    }
}
```

Listing 22. Przykład użycia wzorca kompozyt, którego implementacja zawiera wygenerowany przez wtyczkę kod – klasa `TextFile`

```
package mgr.example.composite;

public class TextFile extends File {
    String name;
    String text;

    public TextFile(String name, String text) {
        this.name = name;
        this.text = text;
    }

    public void print() {
        System.out.println(toString());
    }

    @Override
    public String toString() {
        return "TextFile{" +
                "name='" + name + '\'' +
                ", text='" + text + '\'' +
                '}';
    }
}
```

Listing 23. Przykład użycia wzorca kompozyt, którego implementacja zawiera wygenerowany przez wtyczkę kod – klasa `Directory`

```
package mgr.example.composite;

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

public class Directory extends File {
    String name;
    private List<File> children = new ArrayList<>();

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

    public void add(File file) {
        children.add(file);
    }

    public int count() {
```

```
        return children.size();
    }

    public void remove(File file) {
        children.remove(file);
    }

    public File getChild(int index) {
        return children.get(index);
    }

    public void print() {
        System.out.println(toString());
        for (File file : children) {
            file.print();
        }
    }

    @Override
    public String toString() {
        return "Directory{" +
                "name='" + name + '\'' +
                '}';
    }
}
```

Listing 24. Przykład użycia wzorca kompozyt, którego implementacja zawiera wygenerowany przez wtyczkę kod – klasa `CompositeExample`

```
package mgr.example.composite;

public class CompositeExample {
    public static void main(String[] args) {
        Directory mainDirectory = new Directory("Main");
        Directory imagesDirectory = new Directory("Images");
        Directory textDirectory = new Directory("Text");

        imagesDirectory.add(new Image("Image 1", 200, 300));
        imagesDirectory.add(new Image("Image 2", 1500, 1500));
        textDirectory.add(new TextFile("Text 2", "Example text"));
        mainDirectory.add(imagesDirectory);
        mainDirectory.add(textDirectory);

        mainDirectory.print();
    }
}
```

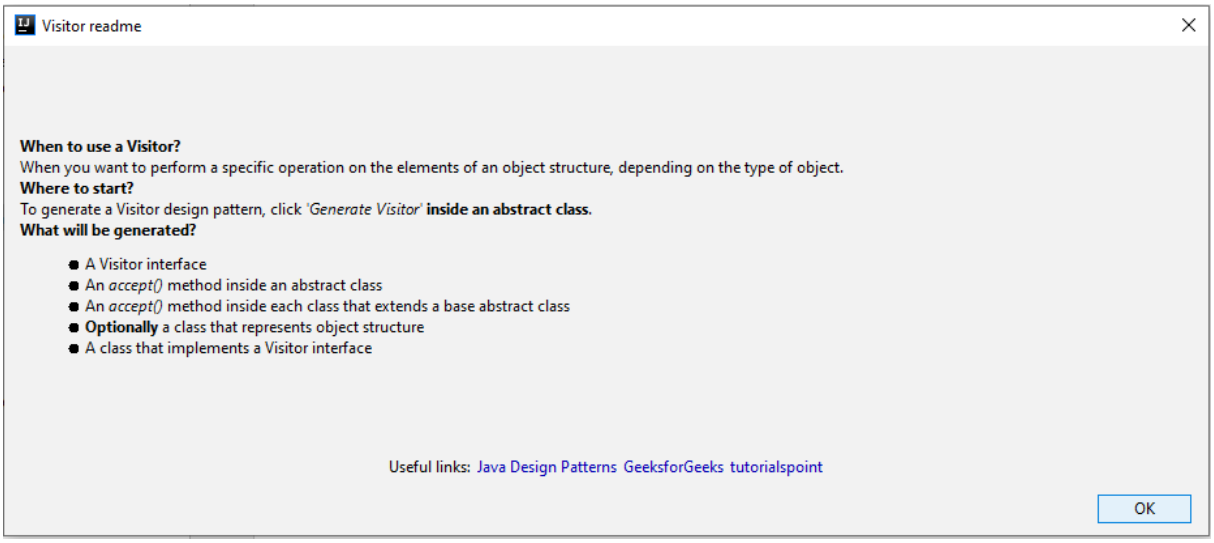
Wynik działanie programu prezentującego przykład użycia przedstawiono na listingu 25.

Listing 25. Wynik działania programu prezentującego przykład użycia

```
Directory{name='Main'}
Directory{name='Images'}
Image{name='Image 1', height=200, width=300}
Image{name='Image 2', height=1500, width=1500}
Directory{name='Text'}
TextFile{name='Text 2', text='Example text'}
```

6.4. Wizytator

Wywołanie funkcji README wtyczki do implementacji wzorców projektowych dla wzorca wizytator, skutkuje wyświetleniem okna dialogowego, przedstawionego na rysunku 26.



Rysunek 26. Okno README dla wzorca wizytator

Okno to zawiera takie informacje jak:

- krótki opis wskazujący w jakiej sytuacji należy zastosować wzorzec wizytator,
- opis minimalnej implementacji,
- krótki opis wygenerowanego kodu,
- linki do witryn internetowych zawierających więcej informacji na temat wzorca wizytator.

Aby móc uruchomić generator wzorca wizytator należy w ramach minimalnej implantacji stworzyć klasę abstrakcyjną, która jest rozszerzona przez co najmniej jedną klasę. Posługując się nomenklaturą schematu przedstawionego w części teoretycznej niniejszej pracy, klasa abstrakcyjna będzie pełniła rolę Elementu, a każda z jej konkretnych implementacji będzie Konkretnym Elementem. Przykład minimalnej implementacji dla wzorca wizytator przedstawiono na listingach 26, 27 i 28.

Listing 26. Przykład minimalnej implementacji dla wzorca wizytator – klasa abstrakcyjna `Animal`

```
package mgr.example.visitor;

public abstract class Animal {
}
```

Listing 27. Przykład minimalnej implementacji dla wzorca wizytator – klasa `Cat`

```
package mgr.example.visitor;

public class Cat extends Animal {
}
```

Listing 28. Przykład minimalnej implementacji dla wzorca wizytator – klasa `Dog`

```
package mgr.example.visitor;

public class Dog extends Animal {
}
```

Następnie użytkownik będzie miał możliwość wyboru, czy chce, aby wtyczka wygenerowała klasę reprezentującą Strukturę Obiektu. Jeżeli potwierdzi wybór, będzie mógł podać nazwę wspomnianej klasy (jeżeli użytkownik nie poda żadnej nazwy, zostanie jej nadana domyślna). Po dostarczeniu minimalnej implementacji i uruchomieniu odpowiedniej opcji wtyczki, wygenerowany zostaje kod, który zaprezentowano na listingach 29, 30, 31, 32, 33 i 34.

Listing 29. Wygenerowany kod dla wzorca wizytator – klasa abstrakcyjna `Animal`

```
package mgr.example.visitor;

public abstract class Animal {
    public abstract void accept(AnimalVisitor animalVisitor);
}
```

Listing 30. Wygenerowany kod dla wzorca wizytator – klasa `Cat`

```
package mgr.example.visitor;

public class Cat extends Animal {
    public void accept(AnimalVisitor animalVisitor) {
        animalVisitor.visit(this);
    }
}
```

Listing 31. Wygenerowany kod dla wzorca wizytator – klasa `Dog`

```
package mgr.example.visitor;

public class Dog extends Animal {
    public void accept(AnimalVisitor animalVisitor) {
        animalVisitor.visit(this);
    }
}
```

Listing 32. Wygenerowany kod dla wzorca wizytator – interfejs `AnimalVisitor`

```
package mgr.example.visitor;

public interface AnimalVisitor {
    void visit(Cat cat);

    void visit(Dog dog);
}
```

Listing 33. Wygenerowany kod dla wzorca wizytator – klasa `AnimalVisitorImplementation`

```
package mgr.example.visitor;

public class AnimalVisitorImplementation implements AnimalVisitor {
    @Override
```

```
    public void visit(Cat cat) {
        System.out.println("Visiting CAT!");
    }

    @Override
    public void visit(Dog dog) {
        System.out.println("Visiting DOG!");
    }
}
```

Listing 34. Wygenerowany kod dla wzorca wizytator – klasa `Zoo`

```
package mgr.example.visitor;

import java.util.List;

public class Zoo {
    private List<Animal> children;

    public Zoo(List children) {
        this.children = children;
    }

    public void accept(AnimalVisitor animalVisitor) {
        for (Animal child : children) {
            child.accept(animalVisitor);
        }
    }
}
```

Wygenerowany kod zawiera:

- publiczną abstrakcyjną metodę `accept` – należy ona do klasy abstrakcyjnej Elementu i wymusza implementację metody `accept` w każdej klasie Konkretnego Elementu,
- publiczny interfejs pełniący rolę Wizytatora wraz z deklaracjami metod `visit` dla każdego Konkretnego Elementu – metody `visit` odróżnia od siebie typ parametru przekazywanego do nich,
- publiczną klasę implementującą interfejs Wizytatora, będącą Konkretnym Wizytatorem – posiada on implementacje metod interfejsu Wizytatora z pustymi ciałami metod, które powinny zostać uzupełnione przez użytkownika.
- publiczną metodę `accept` w każdym z Konkretnych Elementów – metoda ta przyjmuje Wizytatora jako argument i wywołuje jego metodę `visit`, przekazując do niej aktualny Konkretny Element.

Utworzony przez wtyczkę kod, można użyć w sposób przedstawiony na listingu 35 (metody `visit` w klasie będącej implementacją Wizytatora zostały uzupełnione o proste wyświetlanie tekstu charakterystycznego dla typu argumentu, przekazanego do metody):

Listing 35. Przykład użycia kodu wygenerowanego dla wzorca wizytator

```
package mgr.example.visitor;

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

public class VisitorExample {
```

```
    public static void main(String[] args) {
        AnimalVisitor animalVisitor = new AnimalVisitorImplementation();
        List<Animal> animals = new ArrayList<>();
        animals.add(new Cat());
        animals.add(new Cat());
        animals.add(new Dog());
        new Zoo(animals).accept(animalVisitor);
    }
}
```

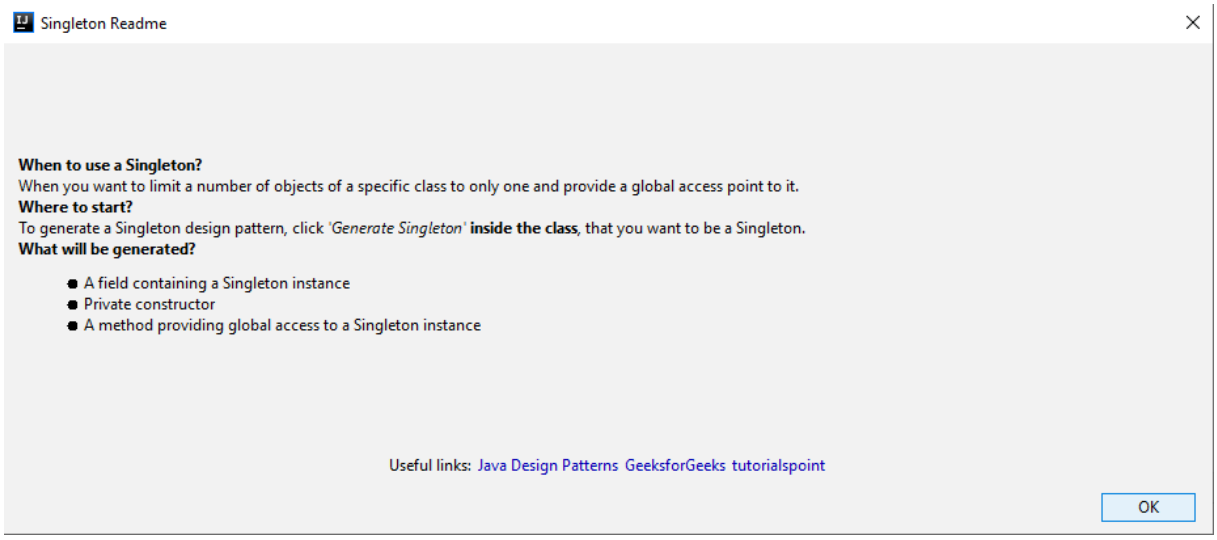
Wynik działania programu prezentującego przykład użycia przedstawiono na listingu 36.

Listing 36. Wynik działania programu prezentującego przykład użycia

```
Visiting CAT!
Visiting CAT!
Visiting DOG!
```

6.5. Singleton

Niniejszy podrozdział prezentuje działanie wtyczki będącej rozszerzeniem wtyczki wspomagającej implementację wzorców projektowych. Wywołanie funkcji README wtyczki będącej rozszerzeniem skutkuje wyświetleniem okna dialogowego, zaprezentowanego na rysunku 27.



Rysunek 27. Okno README dla wzorca singleton

Okno to zawiera takie informacje jak:

- krótki opis wskazujący w jakiej sytuacji należy zastosować wzorzec wizytator,
- opis minimalnej implementacji,
- krótki opis wygenerowanego kodu,
- linki do witryn internetowych zawierających więcej informacji na temat wzorca singleton.

Aby móc uruchomić generator wzorca singleton, należy w ramach minimalnej implementacji stworzyć klasę. Klasa ta zostanie w efekcie działania wtyczki przekształcona w singleton. Przykład minimalnej implementacji dla wzorca singleton przedstawiono na listingu 37.

Listing 37. Przykład minimalnej implementacji dla wzorca singleton

```
package mgr.example.singleton;

public class DatabaseConnection {
    private long sid = 10000000;

    public long getSid() {
        return sid;
    }
}
```

Po dostarczeniu minimalnej implementacji i uruchomieniu opcji generowania kodu, wygenerowany zostaje kod zaprezentowany na listingu 38.

Listing 38. Wygenerowany kod dla wzorca singleton

```
package mgr.example.singleton;

public final class DatabaseConnection {
    private static final DatabaseConnection INSTANCE = new DatabaseConnection();
    private long sid = 10000000;

    private DatabaseConnection() {
    }

    public static DatabaseConnection getInstance() {
        return INSTANCE;
    }

    public long getSid() {
        return sid;
    }
}
```

Wygenerowany kod zawiera:

- modyfikator `final` klasy będącej minimalną implementacją – zastosowanie tego modyfikatora uniemożliwia dziedziczenie klasy będącej Singletonem,
- prywatne statyczne finalne pole `INSTANCE` – reprezentuje ono jedyną instancję klasy będącej singletonem. Wspomniany w części teoretycznej niniejszej pracy problem dotyczący bezpieczeństwa w kontekście wielowątkowości, został rozwiązany poprzez wczesną inicjalizację `INSTANCE`. Stosując wczesną inicjalizację polegamy absolutnie na wirtualnej maszynie Javy, która zapewnia, że jedyna instancja Singletonu zostanie stworzona, zanim jakikolwiek wątek zażąda dostępu do niej [2],
- prywatny konstruktor – dzięki zastosowaniu prywatnego konstruktora, nie jest możliwe utworzenie jakiejkolwiek innej instancji Singletonu poza jedyną przechowywaną w polu `INSTANCE`. Jeżeli przed wygenerowaniem kodu wzorca singleton w wybranej klasie znajdował się inny konstruktor, wtyczka usunie go i pozostawi jedynie prywatny,
- publiczna statyczna metoda `getInstance` – zwraca ona jedyną instancję klasy będącej singletonem.

Utworzony przez wtyczkę kod, można użyć w sposób przedstawiony na listingu 39.

Listing 39. Przykład użycia wygenerowanego singletonu

```
package mgr.example.singleton;

public class SingletonExample {
    public static void main(String[] args) {
        long sid = DatabaseConnection.getInstance().getSid();
        System.out.println("Sid: " + sid);
    }
}
```

Wynik działania programu prezentującego przykład użycia zaprezentowano na listingu 40.

Listing 40. Wynik działania programu prezentującego przykład użycia

```
Sid: 10000000
```

7. Podsumowanie

W wyniku analizy zrealizowanej w niniejszej pracy, dotyczącej wspomagania implementacji wzorców projektowych w zintegrowanych środowiskach programistycznych, powstało narzędzie temu służące. Na podstawie wspomnianej analizy, a także na podstawie procesu tworzenia wtyczki i samego jej użytkowania, można nakreślić możliwe perspektywy jej rozwoju, a także sformułować szereg wniosków.

Wtyczka służąca automatyzacji implementacji wzorców projektowych w obecnej formie sprawdza się zdecydowanie najlepiej w przypadku wzorców o prostej strukturze lub wręcz w odniesieniu do idiomów programistycznych. Wraz ze wzrostem zależności pomiędzy elementami struktury wzorca, a także wzrostem samej liczby elementów, automatyzacja staje się bardziej skomplikowana i konieczne jest podejmowanie coraz to większej liczby decyzji dotyczących konkretnego wariantu implementacji. Rozwiązaniem tego problemu mogłoby być wprowadzenie bardziej rozbudowanego interfejsu użytkownika, który prowadziłby użytkownika przez proces generowania kodu tak, aby finalny produkt generatora odpowiadał jego oczekiwaniom. Rozbudowany interfejs mógłby służyć także określaniu przez użytkownika elementów, które chciałby, aby zostały wygenerowane. Doskonałym uzupełnieniem tej możliwości są wtyczki rozszerzające, ponieważ pozwalają one na przykład na dopisanie własnej implementacji elementów wybranego wzorca, z których generowania programista zrezygnował. Interfejs służący rozszerzeniu możliwości wtyczki, który obecnie oddelegowuje praktycznie cały proces generowania kodu do wtyczki rozszerzającej, powinien wówczas zostać przemodelowany tak, aby można było oddelegować poszczególne fragmenty procesu generowania kodu dla wybranego wzorca.

Kolejnym aspektem jest użyteczność wtyczki w pracy programistów, która często polega na utrzymaniu kodu i pracy z kodem zastanym. W takiej sytuacji, gdy ilość nowego kodu wytwarzanego przez programistę jest niewielka, a do głównych jego zadań należy analiza i naprawa kodu zastanego, użyteczność wtyczki jest znikoma i ogranicza się praktycznie do idiomów programistycznych. Bazując na doświadczeniu autora niniejszej pracy, przy wspomnianym charakterze obowiązków programisty, idiomy wykorzystywane są częściej niż pełnoprawne wzorce projektowe. Dodatkowo idiomy posiadając prostszą i mniej zależną od kontekstu strukturę, można łatwiej wdrożyć przy okazji na przykład zmian w kodzie zastanym.

Wtyczka wspomagająca implementację wzorców projektowych pozwala zaoszczędzić jej użytkownikowi czas, który musiałby przeznaczyć na bardzo szczegółową analizę implementacji pożądanego wzorca. Taka analiza pozwala często na określenie punktów decyzyjnych, co zapewniłaby rozbudowa interfejsu użytkownika. Niewątpliwą wadą wtyczki zaprezentowanej w niniejszej pracy jest fakt, że pomimo skrócenia czasu niezbędnego na analizę wzorca, dalej to na programiście spoczywa konieczność znajomości katalogu dostępnych wzorców i sytuacji, w których powinny zostać zastosowane. Rozwiązaniem tego problemu mogłoby być rozszerzenie możliwości wtyczki o analizę kodu i sugerowanie zastosowania odpowiedniego wzorca projektowego. Najprostszym przykładem takiego scenariusza może być klasa, której konstruktor posiada cztery i więcej parametrów. W takiej sytuacji wtyczka powinna zasugerować zastosowanie wzorca budowniczy. Aby określić scenariusze, w których wtyczka podpowiadałby użycie określonego wzorca, można by było zdefiniować pewne klasy problemów, przy rozwiązaniu których pożądane byłoby zastosowanie określonego wzorca. Następnie, analizując rozwiązania wspomnianych problemów na przykład przez niedoświadczonych programistów, możliwe byłoby tworzenie reguł określających sytuacje, gdy wtyczka sugerowałaby zastosowanie odpowiedniego wzorca.

Niniejsza praca dowodzi, że wspomaganie implementacji wzorców projektowych w zintegrowanych środowiskach programistycznych jest możliwe. Korzystanie z narzędzi tego typu jest zasadne, choć osiągane rezultaty są zależne od czynników, takich jak między innymi złożoność

struktury wzorca. Przedstawiono także perspektywy rozwoju, dzięki którym możliwe byłoby spopularyzowanie przedstawionej wtyczki, poprzez rozszerzenie jej użyteczności w pracy programisty. Ponadto na podstawie zaproponowanych perspektyw rozwoju można wywnioskować jak złożonym i skomplikowanym narzędziem byłaby wtyczka, która posiadałaby pożądaną przez programistów elastyczność, a także oferowałaby maksymalną użyteczność niezależne od kompetencji programisty.

Prace cytowane

[1]. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Wzorce projektowe*, Helion S.A., 2010.

[2]. E. Freeman, E. Freeman, K. Sierra, B. Bates, *Head first design patterns*, O'Reilly Media, Inc., Sebastopol, 2004.

[3]. P. Kuchana, *Software architecture design patterns in Java*, Auerbach Publications, 2004.

[4]. Ch. G. Lasater, *Design patterns*, Wordware Publishing, Inc., Plano, Texas, 2007.

[5]. J. W. Cooper, *Java design patterns: a tutorial*, Addison Wesley Longman, Inc., Reading, Massachusetts, 2000.

[6]. S. R. Alpert, K. Brown, B. Woolf, *The design pattern Smalltalk companion*, Addison Wesley, Reading, Massachusetts, 1998.

[7]. F. Buschmann, R. Meunier, *Pattern oriented software architecture: a system of patterns*, John Wieley and Sons, New York, 1996.

[8]. W. Pree. *Design patterns for object-oriented software development*, Addison Wesley, Reading, Massachusetts, 1994.

[9]. R. C. Martin, M. Martin, *Agile principles, patterns, and practices in C#*, Pearson Education, Inc., Westford, Massachusetts, 2006

[10]. *JHipster*, *https://www.jhipster.tech/*, dostęp: czerwiec 2020

[11]. *FreeBuilder*, https://github.com/inferred/FreeBuilder, dostęp: czerwiec 2020

[12]. *DesignPatterns*, https://github.com/OrPolyzos/design-patterns-plugin, dostęp: czerwiec 2020

[13]. A. Miller, Pure Danger Tech, *Patterns I Hate #1: Singleton*, dostęp: czerwiec 2020

[14]. *About the Java Technology*, https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html, dostęp: czerwiec 2020

[15]. *The Java Language: A White Paper*, https://tech-insider.org/java/research/acrobat/9503.pdf, dostęp: czerwiec 2020

[16]. J. Kubryński, *Co każdy programista Java powinien wiedzieć o JVM*, https://bottega.com.pl/pdf/materialy/jvm/jvm1.pdf, dostęp: czerwiec 2020

[17]. *IntelliJ IDEA overview*, https://www.jetbrains.com/help/idea/discover-intellij-idea.html, dostęp: czerwiec 2020

[18]. *What is the IntelliJ Platform?*, https://www.jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html, dostęp: czerwiec 2020

[19]. *Creating Your First Plugin*, https://www.jetbrains.org/intellij/sdk/docs/basics/getting_started.html, dostęp: czerwiec 2020

[20]. *Krótka historia Git*, https://git-scm.com/book/pl/v2/Pierwsze-kroki-Kr%C3%B3tka-historia-Git, dostęp: czerwiec 2020

[21]. *Wprowadzenie do kontroli wersji*, https://git-scm.com/book/pl/v2/Pierwsze-kroki-Wprowadzenie-do-kontroli-wersji, dostęp: czerwiec 2020

[22]. *Podstawy Git*, https://git-scm.com/book/pl/v2/Pierwsze-kroki-Podstawy-Git, dostęp: czerwiec

[23]. *JVM Ecosystem Report 2020*, https://snyk.io/wp-content/uploads/jvm_2020.pdf, dostęp: czerwiec 2020

[24]. *Plugin Extension Points*, https://www.jetbrains.org/intellij/sdk/docs/basics/plugin_structure/plugin_extension_points.html, dostęp: czerwiec 2020

[25]. *How to Model MVC Framework with UML Sequence Diagram?*, https://www.visual-paradigm.com/guide/uml-unified-modeling-language/how-to-model-mvc-with-uml-sequence-diagram/, dostęp: czerwiec 2020

Spis rysunków

Spis listingów