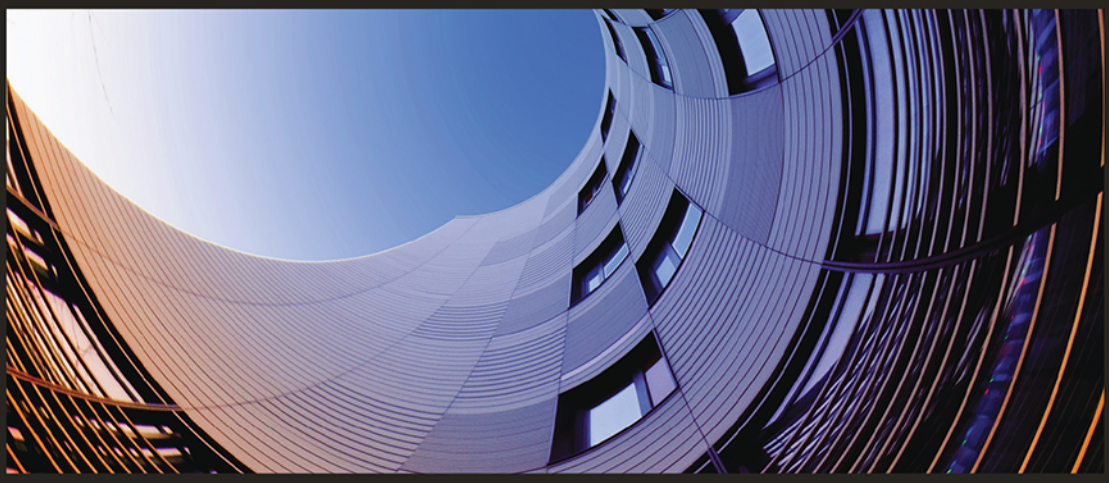
Helion

<packt>



Wzorce projektowe dla programistów Javy

Udoskonal swoje umiejętności projektowania oprogramowania

MIROSLAV WENGNER

Tytuł oryginału: Practical Design Patterns for Java Developers: Hone your software design skills by implementing popular design patterns

Tłumaczenie: Piotr Rajca

ISBN: 978-83-289-0772-0

First published in the English language under the title 'Practical Design Patterns for Java Developers – (9781804614679)'.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/wzoprj
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 2

CZĘŚĆ 2. Implementowanie standardowych wzorców projektowych w języku Java

CZĘŚĆ 3. Inne ważne wzorce projektowe i antywzorce

Kreacyjne wzorce projektowe

Rozdział 3

W ostatnich dziesięcioleciach społeczność informatyczna doświadczyła dramatycznego przejścia od wcześniejszych odizolowanych systemów do rozwiązań rozproszonych lub hybrydowych. Te odmienne podejścia ujawniają nowe możliwości rozwoju oprogramowania.

Wydaje się, że rozwiązania rozproszone spełniają potrzeby migracji starszych systemów, ale rzeczywistość może okazać się inna. Wymagana refaktoryzacja może powodować dodatkowe problemy wynikające z konieczności podziału obowiązków lub refaktoryzacji ściśle powiązanej logiki i reguł biznesowych oraz wielu nieznanych ukrytych logik, które są odkrywane zbyt późno, aby na nie zareagować.

W tym rozdziale zajmiemy się kreacyjnymi wzorcami projektowymi. Wzorce te odgrywają istotną rolę w konstruowaniu oprogramowania. Są one bardzo przydatne, gdyż pozwalają zapewnić łatwość utrzymania bazy kodu oraz jego czytelność. Kreacyjne wzorce projektowe starają się przestrzegać wszystkich wcześniej wymienionych zasad lub podejścia **„nie powtarzaj się”** (**DRY**). W tym rozdziale przyjrzyjmy się bliżej konkretnym wzorcom w następującej kolejności:

- stosowanie wzorca Metoda wytwórcza,
- hermetyczne tworzenie dodatkowych fabryk za pomocą wzorca Fabryka abstrakcyjna,
- tworzenie innej konfiguracji instancji przy użyciu wzorca Budowniczy,
- unikanie wielokrotnie złożonej konfiguracji przy użyciu wzorca Prototyp,
- sprawdzanie istnienia tylko jednej instancji przy wykorzystaniu wzorca Singleton,
- przyspieszanie działania dzięki użyciu przygotowanych instancji przy wykorzystaniu wzorca Pula obiektów,
- kontrolowanie instancji na żądanie za pomocą wzorca Leniwa inicjalizacja,
- redukowanie instancji obiektów przy wykorzystaniu wzorca Wstrzykiwanie zależności.

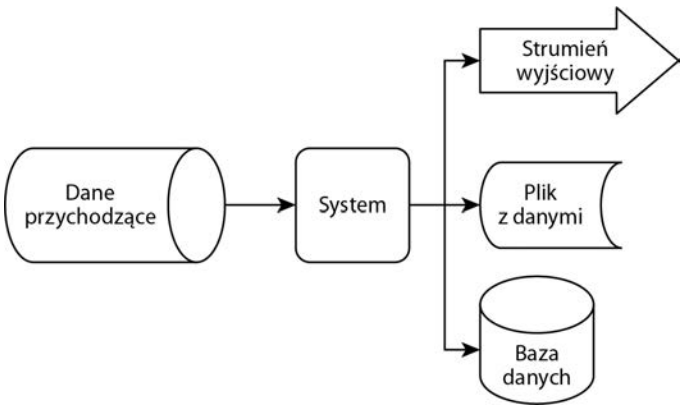
Pod koniec tego rozdziału będziesz dysponował solidną wiedzą na temat pisania łatwego w utrzymaniu kodu do tworzenia obiektów, które mogą być przechowywane zarówno na stercie, jak i na stosie wirtualnej maszyny Javy.

Wymagania techniczne

Pliki źródłowe z kodem przykładów prezentowanych w tym rozdziale można znaleźć w przykładach do książki dostępnych na serwerze wydawnictwa Helion, pod adresem *https://ftp.helion.pl/przyklady/wzoprj.zip*.

Wszystko zaczyna się od klas, które stają się obiektami

W Javie każdy obiekt musi być najpierw opisany przez klasę. Rozważmy pokrótce typowy teoretyczny scenariusz aplikacji. Takie scenariusze są często podzielone na części przedstawione na rysunku 3.1.



Rysunek 3.1. Ogólny obraz powszechnie stosowanego procesu obsługi danych w aplikacjach

Strumień wejściowy danych przychodzących (czyli przepływ informacji) został zaakceptowany przez aplikację. Aplikacja przetwarza dane wejściowe i tworzy wynik. Wynik jest przechowywany i poddawany wymaganemu ukierunkowaniu przez system.

Taki system ma możliwość spełnienia kilku różnych procesów w określonych warunkach. Wyniki są przechowywane na kilka sposobów, w bazie danych lub w pliku, lub ewentualnie zapisywane w określonym strumieniu wyjściowym, takim jak strona internetowa, w celu wyświetlenia informacji użytkownikom.

System działa jako rezerwuar napływających informacji, przetwarza je i przechowuje w bazie danych, a następnie dostarcza wyniki. Przez większość czasu wszystko jest ściśle powiązane i wzajemnie ze sobą połączone.

Powiązania występowały na kilku różnych poziomach, a projektant oprogramowania tego nie zauważał. Ścisła zależność występowała pomiędzy klasami, obiektami, a nawet pakietami. Pod wieloma względami możliwe było korygowanie problematycznej wydajności aplikacji za pomocą mocniejszego sprzętu. Ewolucja systemu postępowała mniej więcej tak jak statystyczna obserwacja prawa Moore'a, które zostało opublikowane w 1965 roku.

Prawo Moore'a mówiło, że każdego roku liczba komponentów w układach scalonych się podwaja. Prawo to zostało zmodyfikowane w 1975 roku i głosiło, że liczba komponentów podwaja się co *dwa* lata. Chociaż debata na temat aktualności prawa Moore'a może okazać się kontrowersyjna, obecne trendy (i szybkość, z jaką potrzebne są aktualizacje sprzętu) pokazują, że nadchodzi czas na kolejną rewizję. Przyspieszenie modernizacji sprzętu (i tak już szybkiej) może nie być konieczne na całym świecie, ponieważ może nie mieć żadnego wpływu na szybkość przetwarzania informacji. Ta obserwacja odnosi się do wymagań związanych z możliwościami aplikacji i koncentruje się bardziej na jakości i złożoności zaimplementowanych algorytmów.

Ciągłe zwiększanie szybkości tworzenia obiektów może nie być możliwe ze względu na ograniczenia fizyczne, ponieważ takie informacje muszą być fizycznie przechowywane w pamięci. Oznacza to, że w nadchodzących dekadach możemy spodziewać się zwiększonego nacisku na poprawę wydajności oprogramowania i jego projektowania. Aby uzyskać przejrzystość logiki aplikacji, musi być jasne, jak ona działa, a ponadto w jaki sposób zasila kluczowe obszary JVM, czyli stos metod i stertę, a następnie jak wygląda wykorzystanie wątków przez obszary stosu (jak pokazano wcześniej na rysunku 2.2).

Ze względu na obecne trendy tworzenia aplikacji, które koncentrują się na odwzorowywaniu, przekształcaniu lub zarządzaniu dużymi ilościami danych, warto studiować, rozumieć i uczyć się, jak radzić sobie z typowymi scenariuszami. Chociaż czas książki Bandy Czworga (ang. *Gang of Four*, w skrócie: **GoF**) już minął, ewolucja jest nieunikniona, a wyzwania pozostają. W wielu przypadkach, przy zachowaniu odpowiedniej abstrakcji, wykorzystanie kreacyjnych wzorców projektowych wciąż może być sensownym rozwiązaniem. Tworzenie obiektów i instancji klas oraz wypełnianie zamierzonych części JVM może drastycznie wpłynąć na koszty obliczeń i wydajności, a także wymusić przejrzystość logiki biznesowej.

W następnym podrozdziale przedstawię różne możliwości tworzenia obiektów. Przyjrzymy się również niedawno dodanym możliwościom i konstrukcjom składniowym Javy, które powinny zmniejszyć wielkość pisanego kodu źródłowego. Zacznijmy od jednego z najpopularniejszych wzorców projektowych.

Tworzenie obiektów na podstawie danych wejściowych przy użyciu wzorca Metoda wytwórcza

Głównym celem tego wzorca projektowego jest scentralizowanie tworzenia instancji klasy określonego typu. Wzorzec pozostawia decyzję o utworzeniu dokładnego typu klasy klientowi, który może ją podjąć w czasie wykonywania kodu. Wzorzec projektowy **Metoda wytwórcza** (ang. *Factory method*) został opisany w książce Bandy czworga.

Uzasadnienie

Wzorzec Metoda wytwórcza wymusza separację kodu i jego odpowiedzialność za tworzenie nowych instancji klasy, co oznacza, że taka metoda zapewnia oczekiwany rezultat. Wzorzec ten ukrywa hierarchię klas aplikacji opartą na uogólnionej abstrakcji i wprowadza wspólny interfejs. W przejrzysty sposób oddziela logikę tworzenia instancji od reszty kodu. Dzięki wspólnemu interfejsowi klient zyskuje swobodę decydowania o utworzeniu konkretnej instancji klasy w czasie wykonywania kodu.

Wzorzec ten jest często używany na wczesnych etapach tworzenia aplikacji, ponieważ jest łatwy do refaktoryzacji i zapewnia wysoki poziom przejrzystości.

Chociaż wzorzec Metoda wytwórcza może wprowadzić pewną złożoność, jest on prosty do zrozumienia i zaimplementowania.

Przykłady zastosowania w JDK

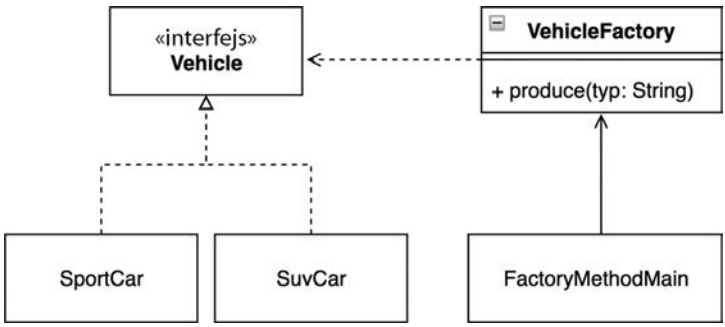
Wzorzec Metoda wytwórcza jest często wykorzystywany w frameworku Java Collection do konstruowania obiektów pożądanego typu. Implementacja tego frameworka należy do pakietu `java.util` w module `java.base`. Pakiet ten zawiera różne implementacje interfejsów `Set`, `List` i `Map`. Chociaż typ `Map` jest ważnym elementem frameworka Java Collection, nie dziedziczy on interfejsu `Collection`, ponieważ w celu przechowywania krotek elementów, kluczy i wartości implementuje interfejs `Map.Entry`. Każda implementacja interfejsów `Set`, `List` i `Map` zapewnia przeciążoną metodę wytwórczą `of` służącą do tworzenia instancji danego typu.

Klasa `Collections` jest klasą użytkową. Zawiera kilka metod wytwórczych służących do tworzenia określonych kolekcji, takich jak lista pojedynczych elementów, mapa lub zbiór. Innym użytecznym przykładem zastosowania wzorca Metoda wytwórcza jest klasa użytkowa `Executors`, należąca do pakietu `java.util.concurrent` w module `java.base`. Klasa `Executors` definiuje metody statyczne, takie jak `newFixedThreadPool`.

Przykładowy kod

Wyobraźmy sobie prosty przykład, który można łatwo zastosować w rzeczywistym świecie przy użyciu odpowiedniej abstrakcji. Celem jest zaprojektowanie aplikacji, która śledzi produkcję pojazdów. Najprawdopodobniej firma oferuje różne typy pojazdów. Każdy pojazd może być reprezentowany przez własny obiekt. Aby graficznie przedstawić nasz zamiar i jednocześnie zachować przejrzystość, na rysunku 3.2 przedstawiłem diagram **UML** (ang. *Unified Modeling Language* — zunifikowany język modelowania) klas tworzonego rozwiązania.

Planowana metoda wytwórcza zamierza tworzyć dwa różne typy pojazdów, a aplikacja używa jej, by tworzyć pojazdy wybranego typu na bieżąco wedle żądania (patrz przykład 3.1).



Rysunek 3.2. Przykład rejestracji produkcji pojazdów

Przykład 3.1. VehicleFactory produkuje pojazdy z tej samej „rodziny" na podstawie argumentów wejściowych

```
public static void main(String[] args) {
    System.out.println("Wzorzec Metoda wytwórcza: Vehicle Factory 2");
    var sportCar = VehicleFactory.produce("sportowe");
    System.out.println("auto sportowe: " + sportCar);
    sportCar.move();
}
```

A oto wyniki wykonania tego programu:

```
Wzorzec Metoda wytwórcza: Vehicle Factory 2
auto sportowe:[typ=porsche 911]
SportCar, typ:'porshe 911', jedzie
```

Zamiast implementować tworzenie takich typów pojazdów w wielu miejscach, tworzymy metodę wytwórczą. Abstrakcja tej metody obejmuje cały proces konstruowania pojazdu i udostępnia tylko jeden punkt wejścia, który pozwala klientowi utworzyć pojazd żądanego typu (jak pokazałem w przykładzie 3.2). Metoda wytwórcza implementuje tylko jedną statyczną metodę, więc sensowne jest zdefiniowanie jej konstruktora jako prywatnego, ponieważ tworzenie instancji tej klasy nie jest pożądane.

Przykład 3.2. Klasa VehicleFactory udostępnia statyczną metodę wytwórczą służącą do tworzenia instancji klas implementujących interfejs Vehicle

```
final class VehicleFactory {
    private VehicleFactory(){}
    static Vehicle produce(String type){
        return switch (type) {
            case "sportowe" -> new SportCar("porsche 911");
            case "suv" -> new SuvCar("skoda kodiaq");
            default -> throw new IllegalArgumentException("""
                    nie zaimplementowano typu:'%s'
                    """.formatted(type));

        };
    }
}
```

Przedstawione w tym kodzie wyrażenie `switch` może korzystać z dopasowywania wzorców w celu uproszczenia kodu zamiast tradycyjnego podejścia, czyli klauzul `case` z etykietami. Aplikacja zapewnia możliwość tworzenia wielu implementacji pojazdu (patrz przykład 3.3):

Przykład 3.3. Każdy rozważany pojazd dziedziczy metody abstrakcji poprzez implementację interfejsu Vehicle

```
interface Vehicle {
    void move();
}
```

Ze względu na inne ulepszenie syntaktyczne platformy Java — rekordy — możliwe jest wybranie poziomu hermetyzacji klasy z uwzględnieniem zasad SOLID. Zależy to od tego, jak bardzo architekt oprogramowania zamierza pozwolić instancji pojazdu na zmianę swojego stanu wewnętrznego. Przyjrzyjmy się najpierw standardowemu podejściu do definiowania klas w Javie (patrz przykład 3.4):

Przykład 3.4. SuvCar umożliwia dodawanie wewnętrznych pól służących do przechowywania stanu, który można zmieniać

```
class SuvCar implements Vehicle {
    private final String type;
    public SuvCar(String t){
        this.type = t;
    }

    @Override
    public void move() {...}
}
```

Architekt oprogramowania może także zastosować nowy rodzaj klas — rekordy — w celu utworzenia niezmiennych instancji wybranego typu pojazdu, które automatycznie będą dysponować implementacjami metod `hashCode`, `equals` oraz `toString`.

Przykład 3.5. Typ SportCar jest uważany za niezmienny

```
record SportCar(String type) implements Vehicle {
    @Override
    public void move() {
        System.out.println("""
            SportCar, type:'%s', move""".formatted(type));
    }
}
```

Niedawno wprowadzona do Javy możliwość definiowania rekordów zmniejsza potencjalną bazę szablonowego kodu, a jednocześnie umożliwia implementację wewnętrznej funkcjonalności (zgodnie z informacjami podanymi w poprzednim rozdziale, w podrozdziale pt. „Rekordy (Java SE 16, JEP-395)”).

Wniosek

Metoda wytwórcza ma pewne ograniczenia. Najważniejszym z nich jest to, że może być używana tylko dla określonej rodziny obiektów. Oznacza to, że wszystkie klasy muszą dysponować podobnymi właściwościami lub jakimiś wspólnymi podstawami. Odchylenie od klasy bazowej może wprowadzić dramatycznie silne powiązania pomiędzy kodem a aplikacją.

Kwestią do rozważenia może być sposób implementacji samej metody wytwórczej; może ona być metodą statyczną bądź instancyjną (jak opisałem w poprzednim rozdziale, odpowiednio w punktach pt. „Obszar stosu" i „Obszar sterty"). Wybór konkretnego rozwiązania zależy od projektanta oprogramowania.

Tworzony jest obiekt należący do jednej rodziny. Przeanalizujmy, jak radzić sobie z problemem tworzenia obiektów należących do wielu różnych rodzin z wykorzystaniem metod wytwórczych mających wspólną właściwość.

Tworzenie obiektów z różnych rodzin przy użyciu wzorca Fabryka abstrakcyjna

Ten wzorzec wprowadza abstrakcję fabryki, która nie narzuca konieczności definiowania konkretnych klas (lub klas wymagających tworzenia swojej instancji). Klient żąda odpowiedniej fabryki, która tworzy instancję obiektu zamiast próbować go samodzielnie utworzyć. Wzorzec **Fabryka abstrakcyjna** (ang. *Abstract factory*) został opisany w książce Bandy czworga.

Uzasadnienie

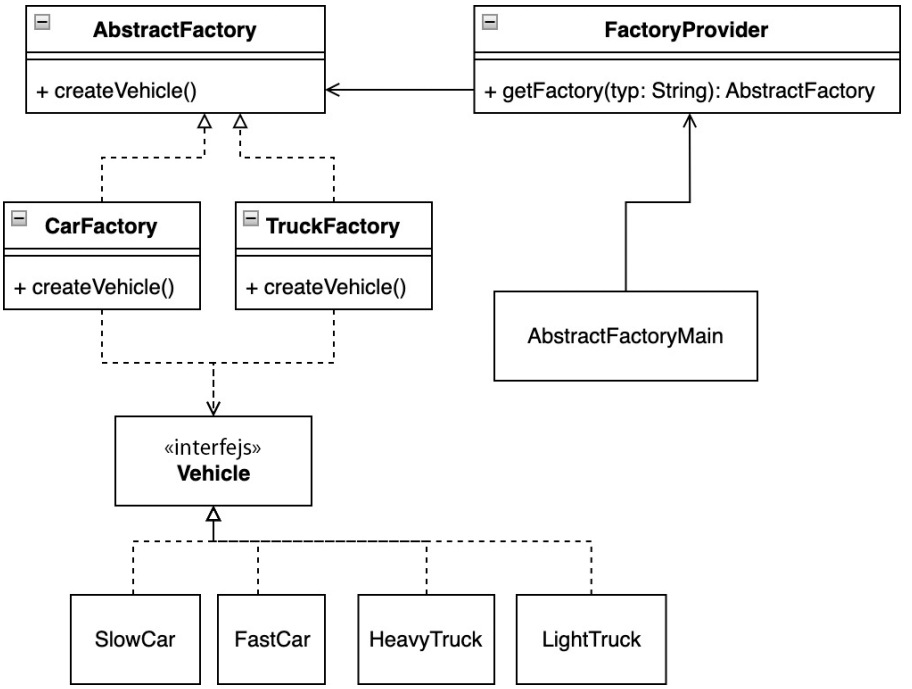
Modularyzacja aplikacji może być poważnym wyzwaniem. Projektanci oprogramowania mogą unikać dodawania kodu do klas, aby zachować ich hermetyzację. Motywacją jest oddzielenie logiki tworzenia instancji klasy od kodu aplikacji, aby istniała możliwość dostarczenia odpowiedniej fabryki służącej do tworzenia wymaganych obiektów. Fabryka abstrakcyjna zapewnia znormalizowany sposób tworzenia instancji pożądanej fabryki i dostarczania jej do klienta w celu dalszego użycia. Klient używa zwróconej fabryki do tworzenia obiektów. Fabryka abstrakcyjna zapewnia interfejs do tworzenia zarówno fabryk, jak i obiektów bez określania ich klas. Wzorzec ten niejawnie wspiera zasady SOLID i łatwość utrzymania kodu poprzez izolowanie logiki klas wykorzystujących i tworzących ten wzorzec. Aplikacja jest niezależna od sposobu, w jaki jej produkty są tworzone, konstruowane i reprezentowane.

Przykłady zastosowania w JDK

Zastosowanie wzorca Fabryka abstrakcyjna można znaleźć w JDK w pakiecie `java.xml` należącym do modułu `java.xml`. Konkretnie rzecz biorąc, jest on używany w reprezentacji i implementacji klasy abstrakcyjnej `DocumentBuilderFactory` i jej statycznej metody `newInstance`. W jego przypadku fabryka korzysta z usługi wyszukiwania, aby znaleźć wymaganą implementację odpowiedzialną za tworzenie dokumentów XML.

Przykładowy kod

Trzeba zauważyć, że choć pojazdy mają pewne wspólne cechy, ich produkcja wymaga różnych rodzajów procesów (patrz rysunek 3.3):



Rysunek 3.3. Wytwarzanie różnych typów pojazdów przy wykorzystaniu wzorca Fabryka abstrakcyjna

W takich przypadkach tworzymy wiele fabryk odpowiedzialnych za tworzenie konkretnych obiektów. Chociaż klasy te należą do różnych rodzin, mają wspólne właściwości. Ważną cechą jest to, że każda fabryka może zaimplementować własną sekwencję inicjalizacji, jednocześnie współdzieląc ogólną logikę. W poniższym przykładzie musimy użyć właściwej instancji `CarFactory`, aby utworzyć obiekt `SlowCar` (patrz przykład 3.6).

Przykład 3.6. Klient decyduje, jakiego rodzaju pojazd jest potrzebny

```
public static void main(String[] args) {
    ...
    AbstractFactory carFactory = FactoryProvider.getFactory("auto");
    Vehicle slowCar = carFactory.createVehicle("wolne");
    slowCar.move();
}
```

A oto wyniki:

```
Wzorzec Fabryka abstrakcyjna: użycie fabryki do tworzenia pojazdów
wolne auto, jedzie
```

W tym przypadku kluczowym elementem jest dostawca fabryki; rozróżnia on, która fabryka jest tworzona na podstawie argumentów wejściowych (patrz przykład 3.7). Dostawca fabryki jest zaimplementowany jako klasa narzędziowa, więc może to być klasa sfinalizowana, mająca prywatny konstruktor, gdyż tworzenie instancji tej klasy nie jest konieczne. Oczywiście implementacja może się różnić w zależności od wymagań.

Przykład 3.7. Klasa FactoryProvider definiuje sposób konfiguracji i tworzenia instancji konkretnej fabryki rodzin obiektów

```
final class FactoryProvider {
    private FactoryProvider(){}
    static AbstractFactory getFactory(String type){
        return switch (type) {
            case "auto" -> new CarFactory();
            case "ciężarówka" -> new TruckFactory();
            default -> throw new IllegalArgumentException("""
                this is %s""".formatted(type));
        };
    }
}
```

Wszystkie fabryki należące do grupy zwracanych fabryk mają wspólną logikę lub wspólne możliwości (patrz przykład 3.8), dzięki czemu są zgodne z zasadą *nie powtarzaj się* (DRY).

Przykład 3.8. Klasa AbstractFactory zapewnia wspólną logikę lub metody, które mogą wymagać implementacji przez konkretną fabrykę

```
abstract class AbstractFactory {
    abstract Vehicle createVehicle(String type);
}
```

Poszczególne fabryki mogą implementować dodatkową logikę w celu rozróżnienia, który produkt powinien zostać dostarczony, jak pokazałem na przykładzie zamieszczonej poniżej implementacji klasy `TruckFactory` (patrz przykład 3.9) i podobnej klasy `CarFactory`.

Przykład 3.9. Klasa TruckFactory reprezentuje konkretną implementację abstrakcyjnej klasy AbstractFactory

```
class TruckFactory extends AbstractFactory {
    @Override
    Vehicle createVehicle(String type) {
        return switch(type) {
            case "ciężkie" -> new HeavyTruck();
            case "lekkie" -> new LightTruck();
            default -> throw new IllegalArgumentException
            ↪("nie zaimplementowano");
        };
    }
}
```

Wniosek

Wzorzec Fabryka abstrakcyjna zapewnia spójność między produktami. Korzystanie z „superfabryki" może powodować niestabilność w środowisku uruchomieniowym klienta — żądany produkt może zgłaszać wyjątek lub błąd z powodu nieprawidłowej implementacji, ponieważ w momencie jego tworzenia nie były znane niezbędne informacje. Jednocześnie wzorzec Fabryka abstrakcyjna promuje testowalność. Fabryka abstrakcyjna może swobodnie reprezentować wiele innych interfejsów, które zostały dostarczone wraz z jej implementacją. Wzorzec ten zapewnia wspólny sposób radzenia sobie z produktami bez wprowadzania zależności od ich implementacji, co może poprawić separację zagadnień w kodzie aplikacji. Wzorzec ten może wykorzystywać interfejsy lub klasy abstrakcyjne. Klient staje się niezależny od sposobu konstruowania i tworzenia obiektów.

Korzyści z hermetyzacji fabryk i separacji kodu mogą być postrzegane jako ograniczenie. Fabryka abstrakcyjna musi być kontrolowana przez jeden parametr lub więcej parametrów niezbędnych do prawidłowego zdefiniowania zależności. Aby poprawić łatwość utrzymania kodu wymaganych fabryk, warto rozważyć wykorzystanie klas zapieczętowanych — nowej możliwości języka Java, opisanej w poprzednim rozdziale, w podrozdziale pt. „Klasy zapieczętowane (Java SE 17, JEP-409)".

Klasy zapieczętowane mogą mieć pozytywny wpływ na stabilność bazy kodu. W następnym podrozdziale przyjrzyjmy się, jak można dostosować proces tworzenia obiektów.

Tworzenie złożonych obiektów przy użyciu wzorca Budowniczy

Wzorzec **Budowniczy** (ang. *Builder*) pomaga oddzielić konstruowanie złożonego obiektu od jego reprezentacji w kodzie, dzięki czemu ten sam proces konstruowania może być ponownie wykorzystany do tworzenia różnych konfiguracji typu obiektu. Wzorzec projektowy Budowniczy został zidentyfikowany dosyć wcześnie i opisany w książce Bandy czworga.

Uzasadnienie

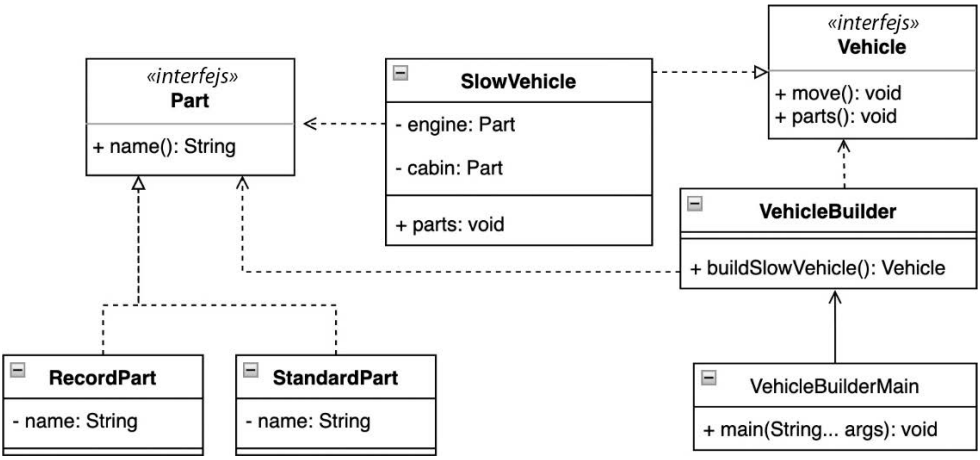
Główną motywacją stojącą za wzorcem Budowniczy jest konstruowanie złożonych instancji bez zanieczyszczania konstruktora. Pomaga to oddzielić lub nawet podzielić proces tworzenia na określone kroki. Konstruowanie obiektów jest przejrzyste dla klienta i pozwala na tworzenie różnych konfiguracji tego samego typu. W przypadku tego wzorca konstruktor jest reprezentowany przez oddzielną klasę. Może to pomóc w przejrzystym rozszerzaniu konstruktora w zależności od potrzeb. Wzorzec ten pomaga hermetyzować i wymuszać przejrzystość procesu tworzenia instancji w zgodzie z opisanymi wcześniej zasadami projektowania SOLID.

Przykłady zastosowania w JDK

Wzorzec Budowniczy jest powszechnie używany w JDK. Doskonałym przykładem jest tworzenie sekwencji znaków reprezentujących łańcuch znaków. Na przykład klasy `StringBuilder` i `StringBuffer` znajdują się w pakiecie `java.lang` modułu `java.base`, który jest domyślnie widoczny dla każdej aplikacji Java. Budowniczy łańcuchów udostępnia wiele przeciążonych metod łączących, które akceptują różne typy danych wejściowych. Takie dane wejściowe są łączone z już utworzoną tablicą bajtów. Inny przykład można znaleźć w pakiecie `java.net.http`, a chodzi o interfejs `HttpRequest.Builder` i jego implementację lub interfejs `Stream.Builder` dostępny w pakiecie `java.util.stream`. Jak wspomniałem wcześniej, wzorzec projektowy Budowniczy jest stosowany bardzo często. Na uwagę zasługują klasy `Locale.Builder` i `Calendar.Builder`, które wykorzystują metody ustawiające do przechowywania wartości produktu końcowego. Obie można znaleźć w pakiecie `java.util` modułu `java.based`.

Przykładowy kod

Obiekt budowniczego, który jest kluczowym elementem wzorca Budowniczy, przechowuje wymagane wartości pól podczas tworzenia instancji typu `Vehicle`, a dokładniej referencje do obiektów (patrz rysunek 3.4).



Rysunek 3.4. Jak w niewidoczny sposób utworzyć nowy pojazd, używając do tego celu wzorca projektowego Budowniczy

Ogólnym zadaniem prezentowanej w tym przykładzie implementacji wzorca Budowniczy jest tworzenie pojazdów (patrz przykład 3.10):

Przykład 3.10. Wzorzec konstruktora można wdrożyć na kilka sposobów, w zależności od wymagań

```
public static void main(String[] args) {
    System.out.println("Wzorzec Budowniczy: tworzenie aut");
```

```
    var slowVehicle = VehicleBuilder.buildSlowVehicle();
    var fastVehicle = new FastVehicle.Builder()
                        .addCabin("kokpit")
                        .addEngine("silnik")
                        .build();

    slowVehicle.parts();
    fastVehicle.parts();
}
```

A oto wyniki:

```
Wzorzec Budowniczy: tworzenie aut
SlowVehicle, silnik: RecordPart[name=silnik]
SlowVehicle, kokpit: StandardPart{name='kokpit'}
FastVehicle, silnik: StandardPart{name='silnik'}
FastVehicle, kokpit: RecordPart[name=kokpit]
```

Wzorzec Budowniczy można implementować na różne sposoby. Jednym z nich jest hermetyzacja i ukrycie całej logiki budowniczego i dostarczenie produktu bezpośrednio, bez ujawniania szczegółów implementacji (patrz przykład 3.11):

Przykład 3.11. VehicleBuilder ukrywa logikę działania w celu dostarczenia konkretnej instancji

```
final class VehicleBuilder {
    static  Vehicle buildSlowVehicle(){
        var engine = new RecordPart("silnik");
        var cabin = new StandardPart("kokpit");
        return new SlowVehicle(engine, cabin);
    }
}
```

Ewentualnie budowniczy może być elementem klasy, której instancja ma zostać utworzona. W takim przypadku można zdecydować, który element powinien zostać dodany do nowo utworzonej instancji (patrz przykład 3.12).

Przykład 3.12. Budowniczy FastVehicle.Builder jest reprezentowany jako klasa statyczna, a użycie go do dostarczenia końcowego wyniku wymaga utworzenia instancji tej klasy

```
class FastVehicle implements Vehicle {
    final static class Builder {
        private Part engine;
        private Part cabin;
        Builder(){}
        Builder addEngine(String e){...}
        Builder addCabin(String c){...}
        FastVehicle build(){
            return new FastVehicle(engine, cabin);
        }
    }
    private final Part engine;
```

```
        private final Part cabin;
        ...
        @Override
        public void move() {...}
        @Override
        public void parts() {...}
    }
```

Oba przedstawione podejścia są zaimplementowane zgodnie z zasadami SOLID. Wzorzec Budowniczy jest dobrym przykładem zasad abstrakcji, polimorfizmu, dziedziczenia i hermetyzacji (APDH) i zapewnia doskonałe możliwości refaktoryzacji, rozszerzania lub walidacji właściwości.

Wniosek

Wzorzec Budowniczy pomaga egzekwować zasadę pojedynczej odpowiedzialności poprzez oddzielenie złożonego tworzenia od logiki biznesowej. Poprawia również czytelność kodu i zapewnia zgodność z zasadą *nie powtarzaj się* (DRY), ponieważ proces tworzenia obiektów jest rozszerzalny i zrozumiały dla użytkownika. Budowniczy jest powszechnie stosowanym wzorcem projektowym, ponieważ redukuje „zapach kodu" i zanieczyszczenie konstruktorów. Poprawia również możliwości testowania kodu. Baza kodu pomaga uniknąć wielu konstruktorów o wielu różnych reprezentacjach, z których niektóre nigdy nie będą używane.

Innym zagadnieniem, które warto rozważyć podczas implementacji wzorca Budowniczy jest wykorzystanie sterty lub stosu JVM — a dokładniej tworzenie reprezentacji wzorca, dla których pamięć będzie przydzielana statycznie lub dynamicznie.

Decyzja ta jest zwykle podejmowana przez samych projektantów oprogramowania i nie zawsze konieczne jest ujawnianie zastosowanego procesu tworzenia obiektów. W następnym podrozdziale przedstawię, jak prosto można klonować obiekty.

Klonowanie obiektów przy wykorzystaniu wzorca Prototyp

Wzorzec **Prototyp** (ang. *Prototype*) rozwiązuje trudności związane z tworzeniem nowych obiektów, których proces konstruowania jest złożony. Jest on stosowany w przypadku klas, dla których proces tworzenia obiektów jest zbyt uciążliwy i niepożądany, gdyż może prowadzić do niepotrzebnego tworzenia klas podrzędnych. Prototyp jest bardzo popularnym wzorcem projektowym i został opisany w książce Bandy czworga.

Uzasadnienie

Wzorzec projektowy Prototyp staje się bardzo przydatny, gdy trzeba utworzyć duże i złożone obiekty, a zastosowanie metody wytwórczej lub fabryki są niepożądanym rozwiązaniem. Nowo utworzona instancja jest klonowana na podstawie jej rodzica,

ponieważ rodzic ten pełni rolę prototypu. Instancje są od siebie niezależne i mogą być dostosowywane. Logika tworzenia instancji nie jest dostępna dla klienta i nie może być przez niego w żaden sposób modyfikowana.

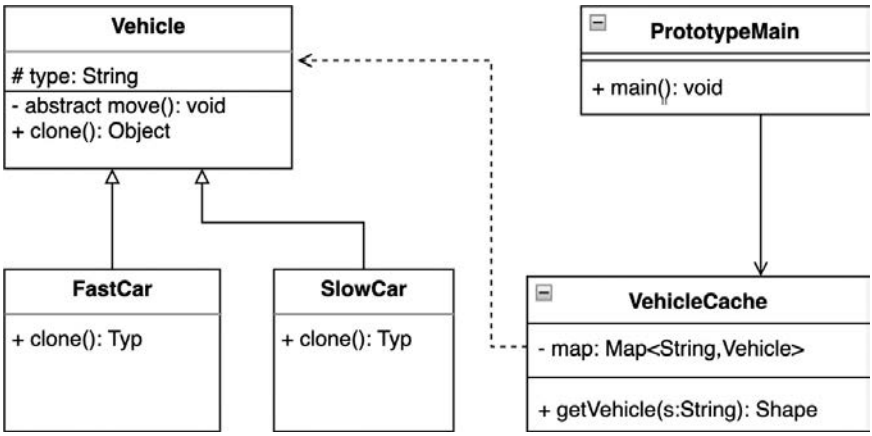
Przykłady zastosowania w JDK

W pakietach JDK można znaleźć wiele przykładów użycia wzorca projektowego Prototyp. Elementy frameworka Collection implementują metodę `clone` wymaganą przez dziedziczony interfejs `Cloneable`. Na przykład wykonanie metody `ArrayList.clone()` tworzy płytką kopię elementów listy, pole po polu.

Inną implementacją tego wzorca projektowego jest klasa `Calendar` dostępna w pakiecie `java.util` modułu `java.base`. Klon przesłoniętej metody jest również używany w implementacji samej klasy `Calendar`, ponieważ pomaga uniknąć niechcianych modyfikacji już skonfigurowanego obiektu. Przykłady użycia tego wzorca można znaleźć w metodach `getActualMinimum` i `getActualMaximum`.

Przykładowy kod

Gdy produkowanych jest tylko kilka modeli pojazdów, nie ma potrzeby, by fabryki i konstruktory bezustannie tworzyły nowe obiekty, co w praktyce mogłoby prowadzić do nieporęcznego działania kodu, wynikającego z możliwości zmian wartości wewnętrznych właściwości. Wyobraźmy sobie wczesny etap produkcji pojazdów, gdzie w celu śledzenia postępów w każdej nowej iteracji konieczne jest zachowanie równości (patrz rysunek 3.5).



Rysunek 3.5. Tworzenie nowych instancji przy użyciu wzorca Prototyp

W takich przypadkach łatwiej jest stworzyć dokładną kopię już zaprojektowanego pojazdu, wykorzystując do tego celu jego prototyp (patrz przykład 3.13).

Przykład 3.13. Nowy pojazd można sklonować na podstawie już istniejącej instancji

```
public static void main(String[] args) {
    System.out.println("Wzorzec Prototyp: prototyp pojazdów");
    Vehicle fastCar1 = VehicleCache.getVehicle("szybkie-auto");
    Vehicle fastCar2 = VehicleCache.getVehicle("szybkie-auto");
    fastCar1.move();
    fastCar2.move();

    System.out.println("equals : " + (fastCar1.equals(fastCar2)));
    System.out.println("fastCar1:" + fastCar1);
    System.out.println("fastCar2:" + fastCar2);
}
```

A oto wyniki:

```
Wzorzec Prototyp: prototyp pojazdów
szybkie auto, jedzie
szybkie auto, jedzie
equals : false
fastCar1:FastCar@816f27d
fastCar2:FastCar@87aac27
```

Instancje mogą być odtwarzane (odpowiednio klonowane) na żądanie. Klasa abstrakcyjna `Vehicle` stanowi podstawę dla każdej nowej implementacji prototypu i udostępnia szczegóły operacji klonowania (patrz przykład 3.14).

Przykład 3.14. Klasa abstrakcyjna Vehicle musi implementować interfejs Cloneable i wprowadzać implementację metody klonowania

```
abstract class Vehicle implements Cloneable{
    protected final String type;


    Vehicle(String t){
        this.type = t;
    }

    abstract void move();

    @Override
    protected Object clone() {
        Object clone = null;
        try{
            clone = super.clone();
        } catch (CloneNotSupportedException e){...}
        return clone;
    }
}
```

Każda implementacja pojazdu wymaga rozszerzenia klasy bazowej `Vehicle` (patrz przykład 3.15).

Przykład 3.15. Charakterystyczna dla interfejsu implementacja klasy Vehicle — klasa SlowCar wraz z metodą move

```
class SlowCar extends Vehicle {
    SlowCar(){
        super("wolne auto");
    }
    @Override
    void move() {...}
}
```

Wzorzec Prototyp wprowadza wewnętrzną pamięć podręczną, w której zapisywane są dostępne prototypy klasy `Vehicle` (patrz przykład 3.16). Proponowana implementacja udostępnia metodę statyczną, która zapewnia, że pamięć podręczna działała jako narzędzie. W tym rozwiązaniu konstruktor powinien być prywatny.

Przykład 3.16. Klasa VehicleClone przechowuje referencje do już przygotowanych prototypów, które można sklonować

```
final class VehicleCache {

    private static final Map<String, Vehicle> map =
        Map.of("szybkie-auto", new FastCar(), "wolne-auto", new SlowCar());

    private VehicleCache(){}
    static Vehicle getVehicle(String type){
        Vehicle vehicle = map.get(type);
        if(vehicle == null) throw new IllegalArgumentException
        ↪("niedozwolone:" + type);
        return (Vehicle) vehicle.clone();
    }
}
```

Przykłady pokazują, że klient za każdym razem korzysta z identycznej kopii podstawowego prototypu. Kopię tę można jednak dostosować do bieżących wymagań.

Wniosek

Wzorzec Prototyp jest przydatny w przypadku stosowania ładowania dynamicznego lub gdy chcemy uniknąć zwiększania złożoności bazy kodu poprzez wprowadzanie niepotrzebnych abstrakcji, czyli **definiowania klas pochodnych** (ang. *subclassing*). Nie oznacza to, że klony nie muszą implementować interfejsu, ale klonowanie może zmniejszyć wymagania dotyczące ujawniania szczegółów klas, jak również zapewnić, że proces tworzenia instancji nie stanie się zbyt skomplikowany. Wzorzec ten odpowiednio hermetyzuje skomplikowaną logikę tworzenia instancji, która nie ma być w żaden sposób modyfikowana. Projektanci oprogramowania powinni być świadomi możliwości, że taka baza kodu może łatwo przekształcić się w kod zastany (ang. *legacy code*). Jednocześnie wzorzec Prototyp może odsuwać w czasie konieczność wprowadzania zmian w bazie kodu i nadawać tym zmianom charakter iteracyjny.

Tworzenie wielu instancji danej klasy nie zawsze jest pożądane, a czasami nawet trzeba go unikać. W następnym podrozdziale opiszę, jak zagwarantować, że podczas wykonywania kodu będzie istnieć tylko jedna unikalna instancja danej klasy.

Zapewnianie istnienia tylko jednego obiektu przy użyciu wzorca projektowego Singleton

Obiekt określny jako *signleton* zapewnia przejrzysty i globalny dostęp do swojej instancji i gwarantuje, że będzie istnieć tylko jedna taka instancja. Wzorzec **Singleton** został zidentyfikowany bardzo wcześnie przez wymagania branżowe i został opisany w książce Bandy czworga.

Uzasadnienie

Klient lub aplikacja chce mieć pewność, że w czasie wykonywania będzie istnieć tylko jedna instancja danej klasy. Aplikacja może wymagać wielu instancji, z których wszystkie korzystają z jednego unikalnego zasobu. Fakt ten wprowadza niestabilność, ponieważ każdy z tych obiektów może uzyskać dostęp do takiego zasobu. Singleton gwarantuje tylko jedną instancję, która zapewnia globalny punkt dostępu dla wszystkich klientów w pożądanym zakresie działającej wirtualnej maszyny Javy.

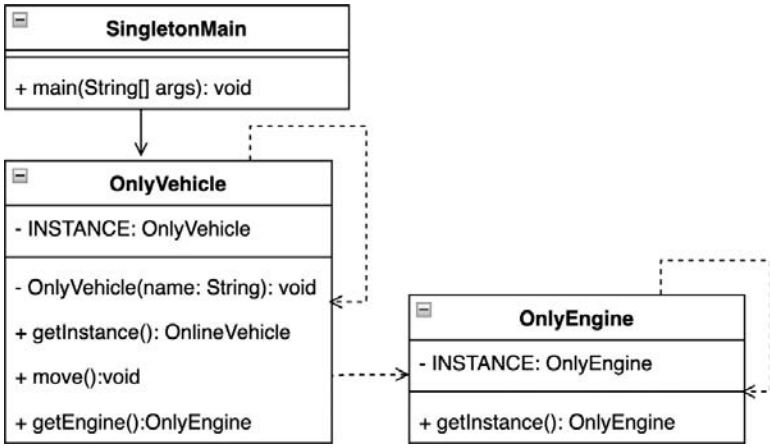
Przykłady zastosowania w JDK

Najlepszym przykładem użycia singletonu jest działająca aplikacja Java, a dokładniej środowisko uruchomieniowe. Znajduje się ona w klasie `Runtime`, a jej metoda `getRuntime` znajduje się w pakiecie `java.lang` modułu `java.base`. Metoda ta zwraca obiekt powiązany z bieżącą aplikacją Javy. Instancja środowiska uruchomieniowego umożliwia klientowi dodanie do działającej aplikacji na przykład metod, które zostaną wykonane podczas jej zamykania.

Przykładowy kod

Poniższy przykład sugeruje aplikację, która uruchamia tylko jeden samochód i jego silniki (patrz rysunek 3.6).

Innymi słowy, oznacza to, że w JVM musi być dostępna tylko jedna instancja określonego typu silnika i pojazdu (patrz przykład 3.17).



Rysunek 3.6. Sposób reprezentacji silnika przy użyciu wzorca projektowego Singleton

Przykład 3.17. W trakcie realizacji kodu istnieje tylko po jednej instancji klas OnlyEngine i OnlyVehicle

```
public static void main(String[] args) {
    System.out.println("Wzorzec Singleton: tylko jeden silnik");
    var engine = OnlyEngine.getInstance();
    var vehicle = OnlyVehicle.getInstance();

    vehicle.move();
    System.out.println("""
        OnlyEngine:'%s', tożsamy z silnikiem pojazdu:'%s'""""
            .formatted(engine, (vehicle.getEngine().equals(engine))));
}
```

A oto wyniki wykonania tego programu:

```
Wzorzec Singleton: tylko jeden silnik
OnlyVehicle, jedzie
OnlyEngine:'OnlyEngine@6b884d57', tożsamy z silnikiem pojazdu:'true'
```

Istnieje kilka różnych sposobów na zapewnienie unikalności obiektu. Implementacja klasy `OnlyEngine` przedstawia możliwą implementację wzorca Singleton, w której instancja klasy jest tworzona na żądanie w sposób określny jako „leniwy" (przykład 3.18). Klasa `OnlyEngine` implementuje ogólny interfejs silnika — `Engine`. Jej implementacja zapewnia statyczną metodę `getInstance` stanowiącą przezroczysty punkt wejścia.

Przykład 3.18. Klasa OnlyEngine sprawdza, czy istnieje jej instancja — leniwa inicjalizacja

```
interface Engine {}
class OnlyEngine implements Engine {
    private static OnlyEngine INSTANCE;
    static OnlyEngine getInstance(){
        if(INSTANCE == null){
            INSTANCE = new OnlyEngine();
```

```
        }
        return INSTANCE;
    }
    private OnlyEngine(){}
}
```

Innym sposobem implementacji wzorca Singleton jest utworzenie statycznego pola należącego do samej klasy i udostępnienie potencjalnemu klientowi punktu wejścia, na przykład o nazwie `getInstance` (przykład 3.19). Należy zauważyć, że w takim przypadku konstruktor klasy powinien zostać zdefiniowany jako prywatny.

Przykład 3.19. Klasa OnlyVehicle udostępnia swoją instancję jako pole statyczne będące składową tej klasy

```
class OnlyVehicle {
    private static OnlyVehicle INSTANCE = new OnlyVehicle();
    static OnlyVehicle getInstance(){
        return INSTANCE;
    }
    private final Engine engine;
    private OnlyVehicle(){
        this.engine = OnlyEngine.getInstance();
    }
    void move(){
        System.out.println("OnlyVehicle, jedzie");
    }
    Engine getEngine(){
        return engine;
    }
}
```

Implementacja wzorca Singleton korzystająca z leniwej inicjalizacji może stać się wyzwaniem w środowisku wielowątkowym, w którym metoda `getInstance` musi być zsynchronizowana, aby możliwe było uzyskanie unikalnej instancji. Jedną z możliwości jest utworzenie singletona jako klasy wyliczeniowej (patrz przykład 3.20).

Przykład 3.20. OnlyEngineEnum — implementacja singletona przy użyciu klasy wyliczeniowej

```
enum OnlyEngineEnum implements Engine {
    INSTANCE;
}
...
private OnlyVehicle(){
    this.engine = OnlyEngineEnum.INSTANCE;
}
...
```

Wniosek

Singleton jest stosunkowo trywialnym wzorcem projektowym, choć może się skomplikować w przypadku używania w środowisku wielowątkowym, gdyż w tych przypadkach

konieczne będzie zagwarantowanie, że będzie istnieć tylko jedna instancja danej klasy. Wzorzec ten może stanowić wyzwanie podczas egzekwowania zasady pojedynczej odpowiedzialności, ponieważ klasa jest w rzeczywistości odpowiedzialna za instancjonowanie samej siebie. Wzorzec Singleton zapewnia także klientom globalny dostęp do przydzielonych zasobów, zapobiegając przypadkowej inicjalizacji lub zniszczeniu obiektu. Wzorzec ten powinien być używany z rozwagą, ponieważ powoduje powstawanie ściśle powiązanego kodu — w formie wymaganych instancji klas — co z kolei może utrudniać testowanie. Co więcej, wzorzec ten utrudnia dziedziczenie, gdyż sprawia, że rozszerzenie klasy singletona jest niemal niemożliwe.

Tworzenie instancji nie zawsze jest dobrym podejściem. Przyjrzyjmy się, jak to zrobić na żądanie.

Poprawianie wydajności dzięki wykorzystaniu wzorca Pula obiektów

Wzorzec **Pula obiektów** (ang. *Object pool*) tworzy gotowe do użycia obiekty i ogranicza czas ich inicjalizacji. Wymagane instancje mogą być odtwarzane na żądanie. Pula obiektów może stanowić bazę warunków, na podstawie których mogą być tworzone nowe instancje, lub ograniczać ich tworzenie.

Uzasadnienie

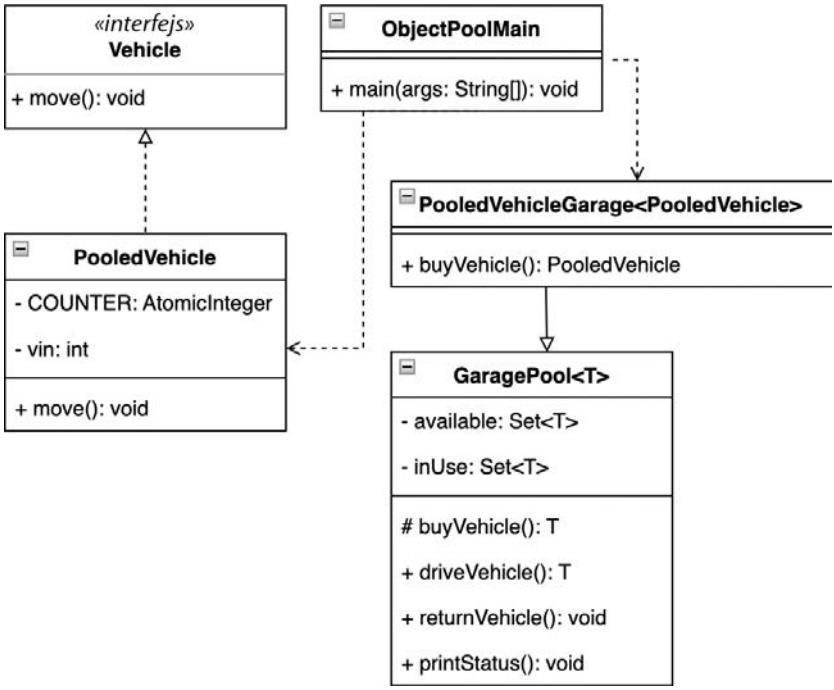
Zamiast ciągłego tworzenia nowych obiektów w bazie kodu pula obiektów zapewnia hermetyzowane rozwiązanie do zarządzania wydajnością aplikacji lub klienta poprzez udostępnianie już zainicjalizowanego obiektu gotowego do użycia. Wzorzec ten oddziela logikę konstruowania obiektów od kodu biznesowego i pomaga zarządzać zasobami oraz wydajnością aplikacji. Może nie tylko pomóc w obsłudze cyklu życia obiektu, ale także w walidacji tych obiektów podczas ich tworzenia lub usuwania.

Przykłady wykorzystania wzorca w JDK

Dobrym przykładem zastosowania wzorca Pula obiektów jest interfejs `ExecutorService` należący do pakietu `java.util.concurrent` i jego implementacja dostarczona przez klasę `Executors` pakietu `util`, która obsługuje odpowiednie instancje wykonawców, jak na przykład metoda `newScheduledThreadPool`.

Przykładowy kod

Bieżący przykład wprowadza scenariusz, w którym salon samochodowy posiada określoną liczbę aut, które kierowcy mogą prowadzić (patrz rysunek 3.7).



Rysunek 3.7. Salon samochodowy zaimplementowany z użyciem wzorca projektowego Pula obiektów

W przypadku gdy samochód nie jest dostępny, salon udostępnia logikę zakupu nowego, aby wszyscy kierowcy byli zajęci (patrz przykład 3.21).

Przykład 3.21. Pobieranie pojazdów z puli przechowywanej przez salon samochodowy redukuje koszty tworzenia obiektów

```
public static void main(String[] args) {
    System.out.println("Wzorzec Pula obiektów: salon samochodowy");
    var garage = new PooledVehicleGarage();
    var vehicle1 = garage.driveVehicle();
    ...
    vehicle1.move();
    vehicle2.move();
    vehicle3.move();
    garage.returnVehicle(vehicle1);
    garage.returnVehicle(vehicle3);
    garage.printStatus();
    var vehicle4 = garage.driveVehicle();
    var vehicle5 = garage.driveVehicle();
    vehicle4.move();
    vehicle5.move();
    garage.printStatus();
}
```

A oto wyniki wykonania tego kodu:

```
Wzorzec Pula obiektów: salon samochodowy
PooledVehicle, jedzie, vin=1
PooledVehicle, jedzie, vin=2
PooledVehicle, jedzie, vin=3
zwrócono pojazd, vin:1
zwrócono pojazd, vin:3
Pojazdy dostępne w salonie=2[[1, 3]] inUse=1[[2]]
PooledVehicle, jedzie, vin=3
PooledVehicle, jedzie, vin=1
Pojazdy dostępne w salonie=0[[]] inUse=3[[2, 1, 3]]
```

Podstawowym elementem tego wzorca jest abstrakcja puli, ponieważ zawiera ona całą logikę wymaganą do zarządzania encjami. Jedną z opcji jest utworzenie abstrakcyjnej klasy puli salonu (patrz przykład 3.22), która zawiera wszystkie mechanizmy synchronizacji. Mechanizmy te są wymagane, aby uniknąć potencjalnej niestabilności i niespójności kodu.

Przykład 3.22. Klasa abstrakcyjnej puli pojazdów udostępnia całą logikę konieczną do prawidłowego zarządzania elementami

```
abstract class AbstractGaragePool<T extends Vehicle> {
    private final Set<T> available = new HashSet<>();
    private final Set<T> inUse = new HashSet<>();
    protected abstract T buyVehicle();
    synchronized T driveVehicle() {
        if (available.isEmpty()) {
            available.add(buyVehicle());
        }
        var instance = available.iterator().next();
        available.remove(instance);
        inUse.add(instance);
        return instance;
    }
    synchronized void returnVehicle(T instance) {...}
    void printStatus() {...}
}
```

Pula salonu ogranicza możliwe typy instancji. Klasa jest powiązana z abstrakcją `Vehicle` (patrz przykład 3.23). Jej interfejs określa wspólne funkcje używane przez klienta. W poniższym przykładzie klienta reprezentuje implementacja `AbstractGaragePool`.

Przykład 3.23. Interfejs Vehicle, który musi zostać zaimplementowany w klasie PooledVehicle

```
interface Vehicle {
    int getVin();
    void move();
}
```

Oprócz implementacji metod klasa `PooledVehicle` udostępnia prywatny licznik (przykład 3.24). Licznik ten należy do klasy, więc jest oznaczony jako statyczny i sfinalizowany. Przechowuje on liczbę instancji przechowywanych w puli salonu.

Przykład 3.24. Implementacja klasy PooledVehicle zawiera także informacje o liczbie utworzonych instancji

```
class PooledVehicle implements Vehicle{
    private static final AtomicInteger COUNTER = new AtomicInteger();

    private final int vin;
    PooledVehicle() {
        this.vin = COUNTER.incrementAndGet();
    }

    @Override
    public int getVin(){...}

    @Override
    public void move(){..}
}
```

Wniosek

Poprawa wydajności klienta pomaga zredukować kosztowny czas tworzenia obiektów (jak widzieliśmy na przykładzie 3.21). Pule obiektów są również bardzo przydatne w przypadkach, gdy wymagane są tylko obiekty używane przez krótki czas, ponieważ pomagają zmniejszyć fragmentację pamięci przez niekontrolowane tworzenie obiektów. Warto wspomnieć o implementacji wewnętrznego wzorca pamięci podręcznej, jaki został zastosowany na przedstawionym przykładzie salonu samochodowego.

Chociaż wzorzec Puli obiektów jest dość wydajny, to ogromny wpływ na jego wydajność może nieć właściwy wybór struktury kolekcji.

Warto rozważyć jeszcze jedno zagadnienie — wpływ zastosowania tego wzorca na proces odśmiecania i zagęszczania pamięci ze względu na analizę aktywnych obiektów. Jest on pozytywny, ponieważ zastosowanie tego wzorca może zmniejszać liczbę obiektów do przeanalizowania.

Nie zawsze jednak przechowywanie wszystkiego w pamięci w celu późniejszego ponownego wykorzystania będzie konieczne. Przyjrzyjmy się, jak opóźnić proces inicjalizacji obiektów i unikać w ten sposób zanieczyszczania pamięci.

Inicjalizacja obiektów na żądanie przy wykorzystaniu wzorca Leniwa inicjalizacja

Celem tego wzorca jest odroczenie tworzenia obiektów żądanej klasy do momentu, gdy klient faktycznie ich zażąda.

Uzasadnienie

Chociaż na przestrzeni lat wielość dostępnej pamięci operacyjnej w komputerach drastycznie wzrosła, z poprzedniego rozdziału dowiedziałeś się, że JVM przydziela dla sterty określony konkretny rozmiar pamięci. Gdy sterta zostanie wyczerpana, a JVM nie będzie w stanie zaalokować żadnego nowego obiektu, spowoduje to błąd braku pamięci. Leniwa obsługa może mieć całkiem pozytywny wpływ na takie zanieczyszczenie sterty. Rozwiązanie to jest czasami nazywane również ładowaniem asynchronicznym ze względu na opóźnienie tworzenia obiektów. Wzorzec **Leniwa inicjalizacja** (ang. *Lazy initialization*) jest dość powszechnie stosowany w aplikacjach internetowych, gdzie strona internetowa jest generowana na żądanie, a nie podczas procesu inicjalizowania aplikacji. Znajduje on także zastosowanie w aplikacjach, w których koszt obsługi odpowiedniego obiektu jest wysoki.

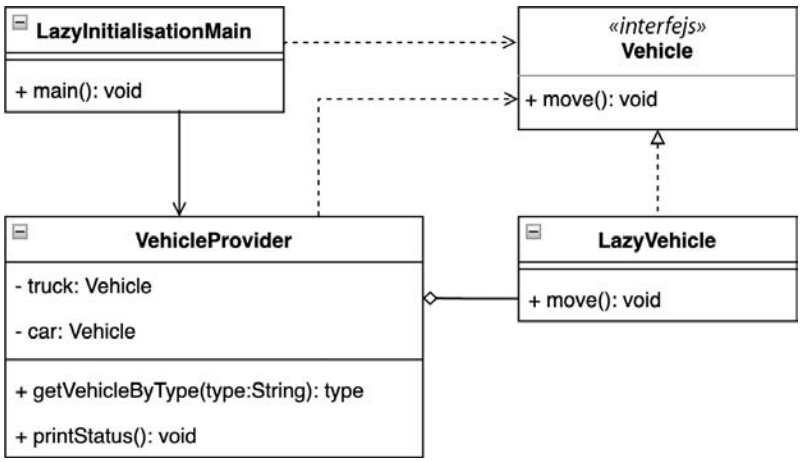
Przykłady zastosowania w JDK

Leniwą inicjalizację można zademonstrować na przykładzie dynamicznego ładowania przez klasę `ClassLoader` klas, które nie zostały dołączone w czasie uruchamiania aplikacji. Klasy mogą być ładowane aktywnie lub odraczane w zależności od używanej polityki. Niektóre klasy, takie jak `ClassNotFoundException`, są ładowane niejawnie przez moduł `java.base`. Obsługują one implementację klasy znajdującą się w pakiecie `java.lang` i jej metodę `forName`. Implementacja metody jest dostarczana przez wewnętrzny interfejs API. Leniwie zainicjowana klasa może być powodem tego, że aplikacja wymaga więcej czasu na uruchomienie. Na przykład typy wyliczeniowe, klasy `Enum`, są specjalnym typem statycznych klas sfinalizowanych, które działają jako stałe i są ładowane aktywnie.

Zgodnie z informacjami podanymi w poprzednim rozdziale operator -> odnosi się do kroków ładowania do mechanizmu ładowania klasy i wypełniania odpowiedniego obszaru metod.

Przykładowy kod

Podstawową ideą przykładu leniwej inicjalizacji jest to, że utworzony pojazd jest inicjowany na żądanie lub, w przypadku gdy jest już utworzony, klient otrzymuje jego referencję (patrz rysunek 3.8).



Rysunek 3.8. Sposób tworzenia pojazdów na żądanie przy wykorzystaniu wzorca projektowego Leniwa inicjalizacja

Takie pojazdy istnieją tylko wtedy, gdy klient na prawdę ich potrzebuje. W takich przypadkach tworzone są konkretne instancje pojazdu. Jeśli pojazd jest już obecny w kontekście dostawcy, wówczas taka instancja jest ponownie wykorzystywana (patrz przykład 3.25).

Przykład 3.25. Przykładowa implementacja aut pobieranych z puli obiektów

```
public static void main(String[] args) {
    System.out.println("Wzorzec Inicjalizacja leniwa: leniwe tworzenie
    ↪pojazdów");
    var vehicleProvider = new VehicleProvider();
    var truck1 = vehicleProvider.getVehicleByType("ciężarówka");
    vehicleProvider.printStatus();
    truck1.move();
    var car1 = vehicleProvider.getVehicleByType("auto");
    var car2 = vehicleProvider.getVehicleByType("auto");
    vehicleProvider.printStatus();
    car1.move();
    car2.move();
    System.out.println("ca1==car2: " + (car1.equals(car2)));
}
```

A oto wyniki:

```
Wzorzec Inicjalizacja leniwa: leniwe tworzenie pojazdów
leniwie utworzono ciężarówkę
stan, ciężarówka:LazyVehicle[type=ciężarówka]
stan, auto:null
LazyVehicle, jedzie, typ:ciężarówka
leniwie utworzono auto
stan, ciężarówka:LazyVehicle[type=ciężarówka]
stan, auto:LazyVehicle[type=auto]
LazyVehicle, jedzie, typ:auto
LazyVehicle, jedzie, typ:auto
ca1==car2: true
```

Implementacja klasy `VehicleProvider` stosuje pola prywatne. W razie konieczności pola te przechowują referencję do żądanego pojazdu. Dostawca hermetyzuje logikę decyzji doboru pojazdu oraz tworzenia instancji. Możliwe implementacje mogą wykorzystywać instrukcję `switch` o klasycznej postaci (patrz przykład 3.26), wyrażenia `switch` itd. Warto zauważyć, że w tym przykładzie klasa `VehicleProvider` wymaga instancji w zakresie pakietu, więc jej konstruktor jest pakietem prywatnym i nie jest dostępny dla innych pakietów.

Przykład 3.26. Klasa VehicleProvider ukrywa przed klientem możliwą złożoność logiki tworzenia obiektów

```
final class VehicleProvider {
    private Vehicle truck;
    private Vehicle car;
    VehicleProvider() {}
    Vehicle getVehicleByType(String type){
        switch(type){
            case "auto":
                ...
                return car;
            case "ciężarówka":
                if(truck == null){
                    System.out.println("leniwie utworzono ciężarówkę");
                    truck = new LazyVehicle(type);
                }
                return truck;
            default:
                ...
        }

    }
    void printStatus(){...}
}
```

Aby wymusić rozszerzalność klas, których obiekty potencjalnie mogą być inicjalizowane w leniwy sposób, każdy z nich implementuje abstrakcję `Vehicle` (patrz przykład 3.27).

Przykład 3.27. Abstrakcja pojazdu i możliwa implementacja LazyVehicle przy użyciu rekordu w celu wymuszenia niezmienności

```
interface Vehicle {
    void move();
}
record LazyVehicle(String type) implements Vehicle{
    @Override
    public void move() {
        System.out.println("LazyVehicle, jedzie, typ:" + type);
    }
}
```

Takie podejście wymusza ciągłą ewolucję pojazdu bez konieczności wprowadzania skomplikowanych zmian w logice dostawcy.

Wniosek

Wzorzec projektowy Leniwa inicjalizacja może pomóc w zmniejszeniu wymagań pamięciowych aplikacji. Jego niewłaściwe użycie może powodować jednak niepożądane opóźnienia, wynikające z faktu, że obiekty mogą być zbyt skomplikowane do tworzenia, a ich uruchomienie może zajmować znaczną ilość czasu.

W następnym podrozdziale pokażę, jak wstrzyknąć logikę do klienta reprezentowanego przez nowo utworzoną instancję pojazdu.

Zmniejszanie zależności przy wykorzystaniu wzorca Wstrzykiwanie zależności

Wzorzec projektowy **Wstrzykiwanie zależności** (ang. *Dependency injection*) oddziela inicjalizację klasy (działającej jako usługa) od klienta (który korzysta z tej usługi).

Uzasadnienie

Wzorzec Wstrzykiwanie zależności jest szeroko stosowany tam, gdzie istnieje potrzeba oddzielenia implementacji określonego obiektu (usługi) od obiektu docelowego (klienta) korzystającego z jego udostępnionych usług, metod itp. Kiedy ma zostać utworzona instancja klienta, usługi są już dostępne. Wzorzec ten pozwala wyeliminować wszelkie zależności umieszczone na stałe w kodzie. Same usługi są tworzone zupełnie niezależnie od procesu tworzenia klienta. Oznacza to, że te dwa elementy są luźno powiązane, co pozwala na zachowanie zgodności z zasadami SOLID. Istnieją trzy sposoby implementacji wstrzykiwania zależności:

- **Wstrzykiwanie zależności do konstruktora**: Zamierzone usługi są udostępniane klientowi poprzez przekazanie ich w wywołaniu konstruktora.
- **Metoda wstrzykująca**: Klient udostępnia odpowiednią metodę w swoim interfejsie. Taka metoda dostarcza zależności do klienta. Obiekt dostawcy używa metody do wstrzyknięcia usługi (lub usług) do klienta.
- **Wstrzykiwanie zależności przy użyciu pola**: Ten typ wstrzykiwania zależności jest wykonywany przy użyciu metody przypominającej metodę ustawiającą (ang. *setter*). Takie metody ustawiające operują na odpowiednim polu klienta. Klient może również udostępnić takie pole w formie publicznej właściwości.

Przykłady zastosowania w JDK

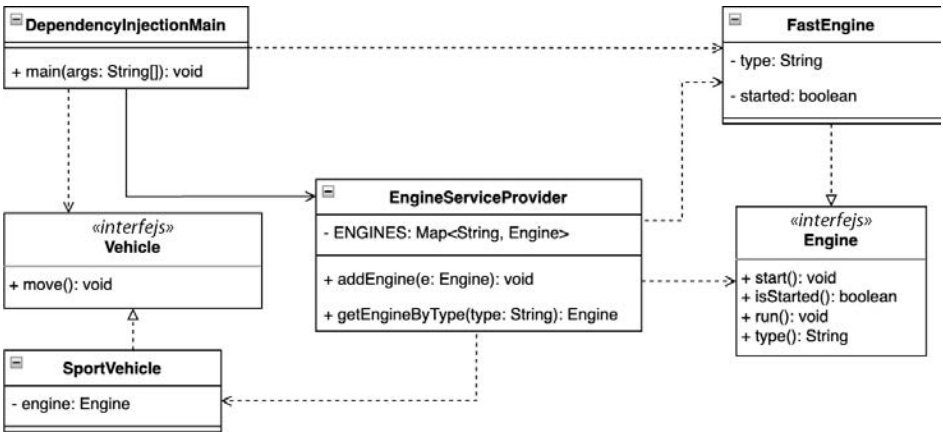
Dobrym przykładem zastosowania wzorca projektowego Wstrzykiwanie zależności jest klasa `ServiceLoader`, którą można znaleźć w module `java.base` i wchodzącym w jego skład pakiecie `java.util`. Instancja `ServiceLoader` próbuje znaleźć usługi w trakcie wykonywania kodu podczas uruchamiania aplikacji. Usługa jest reprezentowana przez dobrze określony

interfejs, który jest implementowany przez odpowiedniego dostawcę lub dostawców usług. Kod aplikacji jest w stanie rozróżnić pożądanego dostawcę w trakcie wykonywania. Warto zauważyć, że `ServiceLoader` działa z klasyczną konfiguracją ścieżki klas, jak również może być używany z systemem modułów platformy Java (który opisałem w rozdziale 2., pt. „Odkrywanie platformy Java pod kątem wzorców projektowych").

W przeszłości wstrzykiwanie zależności było możliwością charakterystyczną dla środowiska Java EE, a nie klasycznego JDK. Oznacza to, że funkcja ta była dostępna na platformie Java. Ze względu na ewolucję JDK funkcje wstrzykiwania zależności zostały przeniesione do projektu *Jakarta Dependency Injection*. Nowo utworzony projekt ma własny cykl wydawniczy i rozwojowy, który nie jest zależny od JDK. Niemniej wstrzykiwanie zależności jest dobrze znane ze względu na użycie adnotacji `@Inject`, `@Named`, `@Scope` lub `@Qualifier`. Adnotacje te umożliwiają przekształcenie klasy w obiekt zarządzany w czasie wykonywania kodu, w którym można wskazać implementację żądanego dostawcy.

Przykładowy kod

Przedstawiony tu przykład pokazuje uproszczoną implementację wzorca Wstrzykiwanie zależności i ma za zadanie dostarczyć jedynie ogólnych informacji o rozwiązaniach tego typu. Jest to bardzo prosta implementacja; w rzeczywistości przedstawia jedynie zarys tego, jak wspomniane wcześniej API działa za kulisami. Wyobraźmy sobie scenariusz, w którym zamierzona instancja pojazdu wymaga silnika (patrz rysunek 3.9).



Rysunek 3.9. Wstrzykiwanie silnika jako usługi do instancji pojazdu

Konkretna logika konstruowania silnika jest oddzielona od kodu związanego z pojazdem (patrz przykład 3.28).

Przykład 3.28. Utworzenie instancji FastEngine niezależnie od pojazdu, a następnie dodanie silnika do tworzonego obiektu SportVehicle

```
1. public static void main(String[] args) {
2.     System.out.println("Wzorzec Wstrzykiwanie zależności:
       ↪pojazd i silnik");
3.     EngineServiceProvider.addEngine(new FastEngine("sportowy"));
```

```
4.      Engine engine = EngineServiceProvider.getEngineByType("sportowy");
5.      Vehicle vehicle = new SportVehicle(engine);
6.      vehicle.move();
7. }
```

A oto wyniki:

```
Wzorzec Wstrzykiwanie zależności: pojazd i silnik
FastEngine, uruchomiony
FastEngine, działa
SportCar, jedzie
```

Instancja `FastEngine` zostaje użyta, gdy jest już w pełni gotowa (zainicjowana, zweryfikowana itd.). Pojazd pożądanego typu można utworzyć niezależnie, bez żadnej zależności od logiki silnika. Instancja silnika jest dostarczana do `SportVehicle` przy wykorzystaniu klasy `EngineServiceProvider` (patrz przykład 3.29).

Przykład 3.29. Klasa EngineServiceProvider rejestruje utworzone usługi nadające się do wielokrotnego użycia

```
final class EngineServiceProvider {
    private static final Map<String, Engine> ENGINES = new HashMap<>();
    ...
    static Engine getEngineByType(String t){
        return ENGINES.values().stream()
            .filter(e -> e.type().equals(t))
            .findFirst().orElseThrow(IllegalArgumentException::new);
    }
}
```

Klasa `SportVehicle` implementuje interfejs `Vehicle` (patrz przykład 3.30), odzwierciedlając podejście „otwarte-zamknięte", o którym wspominałem wcześniej w ramach opisu zasad projektowania SOLID (patrz przykład 3.30).

Przykład 3.30. SportVehicle implementuje interfejs Vehicle wraz z dodatkową wewnętrzną logiką dla dostarczonej instancji Engine

```
interface Vehicle {
    void move();
}
class SportVehicle implements Vehicle{
    private final Engine engine;
    SportVehicle(Engine e) {...}
    @Override
    public void move() {
        if(!engine.isStarted()){
            engine.start();
        }
        engine.run();
        System.out.println("SportCar, jedzie");
    }
}
```

Zauważ, że chociaż instancja określonego typu silnika (patrz przykład 3.31) (`FastEngine`) jest tworzona gdzie indziej (patrz przykład 3.28, wiersz 3), jej obecność jest wymagana podczas tworzenia instancji `SportVehicle` (patrz przykład 3.28, wiersz 5).

Przykład 3.31. Interfejsy silnika implementowane przez klasę FastEngine i dostarczane przez EngineServiceProvider

```
interface Engine {
    void start();
    boolean isStarted();
    void run();
    String type();
}
class FastEngine implements Engine{
    private final String type;
    private boolean started;
    FastEngine(String type) {
        this.type = type;
    }
    ...
}
```

Kluczowym obiektem w opisywanym przykładzie jest `EngineServiceProvider`. Zwraca on referencję do już utworzonych pożądanych instancji silnika i udostępnia ją w kodzie biznesowym. Oznacza to, że każdy klient, który do działania potrzebuje silnika, taki jak instancja `SportVehicle`, uzyska dostęp do właściwej instancji za pośrednictwem łącza udostępnionego przez `EngineServiceProvider`.

Przedstawiony trywialny przykład można łatwo przekształcić w inny przy użyciu instancji klasy `ServiceProvider`. Niezbędne zmiany w kodzie są minimalne (patrz przykład 3.32).

Przykład 3.32. ServiceLoader zapewnia dostępną implementację interfejsu Engine używanego do tworzenia instancji typu SportVehicle

```
public static void main(String[] args) {
    System.out.println("Wzorzec Wstrzykiwanie zależności: pojazd i silnik");
    ServiceLoader<Engine> engineService = ServiceLoader.load(Engine.class);
    Engine engine = engineService.findFirst().orElseThrow();
    Vehicle vehicle = new SportVehicle(engine);
    vehicle.move();
}
```

W standardowym wykorzystaniu ścieżki klas platforma Java wymaga, aby dostawcy usług byli zarejestrowani w podkatalogu *services* w katalogu *META-INF*. Nazwą pliku jest nazwa pakietu i interfejsu usługi, a sam plik zawiera listę dostępnych dostawców usług.

System modułów platformy Java upraszcza etapy konfiguracji. Odpowiednie moduły zapewniają (i udostępniają implementacje usług) modułom docelowym, o czym wspomniałem w rozdziale 2., pt. „Odkrywanie platformy Java pod kątem wzorców projektowych”.

Wniosek

Wzorzec Wstrzykiwanie zależności zapewnia, że klient nie dysponuje żadnymi informacjami o procesie tworzenia używanej usługi. Klient uzyskuje dostęp do usługi za pośrednictwem wspólnych interfejsów. Rozwiązanie to poprawia możliwości testowania bazy kodu. Sprawia także, że testowanie bazy kodu jest łatwiejsze. Wstrzykiwanie zależności jest wzorcem projektowym powszechnie stosowanym przez różne frameworki, takie jak Spring i Quarkus. Quarkus wykorzystuje specyfikację Jakarta Dependency Injection. Wzorzec Wstrzykiwanie zależności jest zgodny z zasadami programowania obiektowego SOLID i APDH, ponieważ zapewnia abstrakcję interfejsów. Kod nie zależy od implementacji, ale komunikuje się z usługami za pośrednictwem interfejsów. Wzorzec ten wymusza także zgodność z zasadą *nie powtarzaj się*, ponieważ nie jest wymagane ciągłe inicjowanie usługi.

Podsumowanie

Wzorce projektowe odgrywają bardzo ważną rolę w projektowaniu aplikacji. Pomagają w przejrzystej centralizacji logiki tworzenia obiektów z zachowaniem podstawowych zasad programowania obiektowego. Przykłady pokazały, że każdy wzorzec może mieć wiele implementacji. Wynika to z faktu, że decyzja o implementacji może zależeć od innych czynników związanych z architekturą oprogramowania. Czynniki te uwzględniają wykorzystanie sterty i stosu JVM, czas działania aplikacji lub hermetyzację logiki biznesowej.

Niejawne używanie wzorców projektowych umożliwia zachowanie zgodności z zasadą *nie powtarzaj się*, co ma pozytywny wpływ na rozwój aplikacji i zmniejsza zanieczyszczenie bazy kodu. Poprawiają się możliwości testowania aplikacji, a architekci oprogramowania zyskują podstawy do potwierdzania obecności oczekiwanych obiektów wewnątrz JVM. Staje się to szczególnie ważne w razie zidentyfikowania problemu logicznego, który może być wyjątkiem lub nieoczekiwanymi wynikami. Dobrze napisana baza kodu pomaga dość szybko dotrzeć do przyczyny problemu, być może nawet bez konieczności debugowania.

W tym rozdziale pokazałem, jak utworzyć obiekt określonej rodziny przy użyciu niezauważalnego wzorca Metoda wytwórcza. Wzorzec Fabryka abstrakcyjna pokazał, jak w sposób wspierający hermetyzację można tworzyć różne rodzaje fabryk. Nie wszystkie wymagane informacje są zawsze dostępne w danym momencie, a wzorzec Budowniczy przedstawił sposób radzenia sobie z takim wyzwaniem w kontekście konstruowania złożonych obiektów. Wzorzec Prototyp zaprezentował podejście polegające na ukrywaniu przed klientem logiki tworzenia obiektów, natomiast wzorzec Singleton zapewnia gwarancję, że w trakcie wykonywania kodu będzie istnieć tylko jedna instancja danej klasy. Wzorzec projektowy Pula obiektów pokazał, jak poprawić wykorzystanie pamięci w czasie wykonywania kodu, a wzorzec Leniwa instancja pokazał, jak odroczyć tworzenie obiektu do momentu, gdy faktycznie będzie potrzebny. Wzorzec Wstrzykiwanie zależności zademonstrował możliwość ponownego wykorzystania instancji podczas tworzenia nowych obiektów.

Wzorce projektowe nie tylko zapewniają przejrzystość tworzenia nowych instancji, ale w wielu przypadkach mogą również przyczynić się do podjęcia decyzji prowadzących do wybrania prawidłowej struktury kodu (struktury, która odzwierciedla wymaganą logikę biznesową). Ostatnie trzy wspomniane wzorce projektowe rozwiązują nie tylko kwestię tworzenia obiektów, ale także ich ponownego używania w celu efektywnego wykorzystania pamięci. Przedstawione tu przykłady pokazały, w jaki sposób można zaimplementować wzorce kreacyjne oraz zademonstrowały cele każdego z nich i różnice pomiędzy nimi.

W następnym rozdziale zajmiemy się strukturalnymi wzorcami projektowymi. Wzorce te pomogą nam zorganizować nasz kod w oparciu o najczęściej występujące scenariusze. A zatem przejdźmy dalej.

Pytania

1. Jakie wyzwania pomagają rozwiązać kreacyjne wzorce projektowe?
2. Które wzorce mogą być pomocne w obniżeniu kosztów tworzenia obiektów?
3. Jaki jest główny powód stosowania wzorca projektowego Singleton?
4. Który wzorzec projektowy pomaga zmniejszyć zanieczyszczenie konstruktorów?
5. Jak ukryć przed klientem złożoną logikę tworzenia obiektów?
6. Czy można zmniejszyć ilość pamięci zużywanej przez aplikację na tworzenie obiektów?
7. Jaki wzorzec projektowy jest przydatny do tworzenia obiektów określonej rodziny?

Dalsza lektura

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku*, Helion, 2010.
- Robert C. Martin, *Design Principles and Design Patterns*, Object Mentor, 2000.
- Gordon E. Moore, *Cramming more components onto integrated circuits*, Electronics Magazine, 19.04.1965.
- Poradniki Oracle: Generics: *https://docs.oracle.com/javase/tutorial/java/generics/index.html*.
- Framework Quarkus: *https://quarkus.io/*.
- Framework Spring: *https://spring.io/*.
- Projekt Jakarta Dependency Injection: *https://jakarta.ee/specifications/dependencyinjection/*.
- Robert C. Martin, *Czysty kod. Podręcznik dobrego programisty*, Helion, 2019.
- Joshua Bloch, *Java. Efektywne programowanie*, Wydanie III, Helion, 2018.

Wzorce projektowe: niezbędnik najlepszych programistów Javy!

Właściwe stosowanie wzorców projektowych jest umiejętnością, którą bardzo cenią twórcy oprogramowania. Wzorce pozwalają na szybkie opracowanie złożonych zagadnień i umożliwiają tworzenie kodu nadającego się do wielokrotnego użycia. Taki kod jest przy tym wydajny, łatwy w testowaniu i utrzymaniu. Z pewnością wzorce projektowe są świetnym rozwiązaniem dla inżyniera, który chce doskonalić umiejętności projektowania oprogramowania.

Książka stanowi zbiór praktycznych informacji dotyczących najpopularniejszych wzorców projektowych. Lekturę rozpoczniesz od zapoznania się z możliwościami korzystania z nich na platformie Javy. Dowiesz się także, jakich zasad trzeba przestrzegać, aby zapewnić czytelność i łatwość utrzymania kodu Javy. W kolejnych rozdziałach znajdziesz praktyczne wskazówki i przykłady dotyczące stosowania wzorców kreacyjnych, strukturalnych, operacyjnych, a także wzorców współbieżności. Nauczysz się używać ich do rozwiązywania problemów często spotykanych podczas projektowania oprogramowania. Końcowy rozdział został poświęcony antywzorcom, a zawarte w nim informacje pomogą Ci w ich identyfikacji i podjęciu najlepszych środków zaradczych.

W książce:

- jakie problemy można rozwiązać za pomocą wzorców projektowych w Javie
- jakie możliwości ma programowanie współbieżne
- jak wzorzec projektowy Obserwator buduje relację „jeden-do-wielu” między instancjami
- jakie problemy pomaga rozwiązać wzorzec Odwiedzający
- jak kontrolować zasoby przy użyciu wzorca Pula wątków
- jak zaradzić problemom spowodowanym przez antywzorce

Miroslav Wengner jest głównym inżynierem w OpenValue i współtwórcą OpenJDK. Pasjonuje się tworzeniem odpornych systemów rozproszonych i zapewnianiem jakości produktów. Bierze też udział w tworzeniu rozwiązań odpornych i skalowalnych. Chętnie występuje na branżowych konferencjach (JavaOne, Devoxx itp.).

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0772-0
9 788328 907720
Cena: 67,00 zł