O mnie

Jestem informatykiem amatorem. Choć pracowałem przez pewien czas w *branży* to jednak nie uważam się za zawodowca. Programuje także amatorsko, obecnie wyłącznie w javie (także pod androida).
Amatorstwo nie oznacza jednak *amatorszczyzny*. Dla mnie to same zalety: nie jestem ograniczony wymaganiami pracodawcy, terminami, narzuconymi technologiami. Mogę uczyć się we własnym tempie, wykorzystywać dowolne technologie po uprzednim ich przetestowaniu.

Wstęp

Gdy rozpoczynamy naukę jakiegoś języka programowania zaczynamy od najprostszych przykładów typu *Witaj świecie*. Dowiadujemy się więcej o typach danych, strukturach sterujących, operatorach, klasach. Są to absolutne podstawy - poziom podstawowy. Powoli przechodzimy wyżej - interfejsy, biblioteki dostarczane wraz z jdk. Proces ten nie jest skokowy, raczej przebiega linearnie i nie można między nimi wyznaczyć wyraźnej granicy.
I to właściwie wystarczy aby napisać dowolny, działający program. Znamy już podstawowe konstrukcje, biblioteki ułatwiające jego tworzenie. Wiele pokoleń programistów borykało się jednak z problemami innej natury, które zaliczają się do kategorii stylu programowania. I wypracowali wiele rozwiązań funkcjonujących do dzisiaj. Są to z jednak strony konwencje nazewnicze, obiektowość, czyste funkcje etc.,etc. z drugiej, będące tematem tej książki wzorce projektowe.
Wzorce projektowe do dobrze zdefiniowane standardowe rozwiązanie znanych już problemów - problemów projektowych, choć to pojęcie bardzo szerokie. Dziedziczyć, czy komponować? interfejsy, czy klasy abstrakcyjne? Kiedy pierwszy raz spotkałem się ze wzorcami projektowymi byłem zauroczony. Być może czasami nadkłada się pracy na początku, ale potem oszczędza bardzo dużo czasu, o ile rozsądnie wykorzystamy te wzorce. Typowym przykładem to nadużywania wzorca singleton - zauroczenie, zauroczeniem a rozsądek rozsądkiem. To też wzorzec.
Dzięki wzorcom projektowym możemy zaoszczędzić czas na tworzenie projektu, a co za tym idzie jego koszty. Jednak nie tylko chodzi o wymierne korzyści. To tak jak się komponuje muzykę - obowiązują zasady: elegancja, prostota, czytelność. Te trzy elementy to tak naprawdę jedność. Żadne nie istniej bez innych.
Wykorzystanie wzorców sprzyja także ponownemu wykorzystaniu kodu. Su-

gerują skoncentrowanie się na abstrakcji, czyli takiemu projektowaniu, aby jak najbardziej odejść od konkretnych rozwiązań. Potem taką abstrakcję można wykorzystać w danym lub innym projekcie.
Poszukiwałem materiałów do tej książki ze szczególnym uwzględnieniem przykładów, natykałem się na takie, które są dość daleko oddalone od rzeczywistości, przynajmniej dla mnie - macierze, zbiory figur matematycznych. Były oczywiście też bardziej praktyczne - samochody, pracownicy, książki. Dla moich celów utworzyłem wymyśloną fabrykę marmolady - pracowałem kiedyś w takiej, więc w miarę się orientuję. Tak fabryka posiada wiele możliwości projektowych w części przeze mnie wykorzystanych.
W książce tej omówię trzy podstawowe typy wzorców: konstrukcyjne, strukturalne i czynnościowe. Funkcją pierwszych jest tworzenie obiektów, ich konfiguracja i inicjalizacja. Wzorce strukturalne opisują powiązania pomiędzy klasami i obiektami, ostatnie z kolei zachowanie powiązanych ze sobą obiektów.
Wspomnę także o antywzorcach, stylu programowania, którego należy unikać.

Część I
Wzorce konstrukcyjne

Obiekty programów w jawie konstruujemy na podstawie utworzonych wcześniej klas. Jednak nie zawsze wygląda to tak prosto. Na przykład współczesne komputery posiadają już wiele rdzeni lub procesorów, umożliwiających przetwarzanie współbieżne. Niesie to ze sobą problemy natury konstrukcyjnej - jak zablokować możliwość wielokrotnego utworzenia obiektu, uchronić się przed niespójnością obiektu lub jego prawidłową inicjalizacją. Tym właśnie zajmują się wzorce konstrukcyjne.

1 Singelton

Wyobraźmy sobie, że mamy bazę danych w naszej fabryce marmolady, która używana jest przez kilku klientów - technologa tworzącego receptury, pracowników produkcji, korzystających z tych receptur, magazynierów i niezmierzoną rzeszę innych. Przechowuje ona wszelkie informacje potrzebne przy produkcji od wprowadzenia receptury, przez produkcję po wysłanie towaru do klienta. Ta baza powinna być jedna, nie można dopuścić do sytuacji, gdy

nagle tworzonych jest kilka jej egzemplarzy i zaczynają żyć własnym życiem, koszmarnie mieszając w systemie. Musimy więc tak skonstruować klasę opisującą bazę, aby można było utworzyć tylko jeden jej egzemplarz tak, aby jakakolwiek zmiana jej stanu w dowolnym miejscu widoczna była od razu wszędzie, aby jednoczesna zmiana tego samego elementu była niemożliwa. Temu celowi służy wzorzec singelton.
Ogranicza on tworzenie obiektu klasy i zapewnia, że tylko jeden taki istnieje na maszynie wirtualnej jawy. Jego realizacja jest stosunkowo prosta, choć istnieją pewne niebezpieczeństwa, które łatwo można zignorować.
Po pierwsze niekiedy niepotrzebne używamy tego wzorca. Pamiętajmy, że obiekt singeltona istnieje od momentu pierwszego wywołania do zakończenia działania programu, niezależnie od tego, czy jest on potrzebny, czy nie. Zajmuje więc zasoby, często bardzo cenne.
Drugi problem wiąże się z przetwarzaniem wielowątkowym, gdy wiele wątków próbuje jednocześnie utworzyć obiekt. Może więc powstać wiele obiektów, które powinny pozostać singletonami[1].
Istnieje wiele obiektów, które powinny być pojedyncze, aby nie spowodować niespójności aplikacji np.: jej kontekst, obiekt sterownika, bazy danych, konsola wejścia lub wyjście[2]
Jest kilka sposobów tworzenia singeltona. Najprostszy to użycie zmiennej statycznej:

SingeltonSimple.java

```
package eu.godlesie.jgdlws;
public class SingeltonSimple {
    public static SingeltonSimple sineltonSimple = new
        SingeltonSimple();
}
```

Jak wiemy może istnieć tylko jedna zmienna statyczna w całym programie. Powyższe rozwiązanie działa dopóty używamy wyłącznie zmiennej *singelton*. Ale, poniższy fragment kodu jest całkowicie poprawny:

```
SingeltonSimple s1 = SingeltonSimple.singelton;
SingeltonSimple s2 = new SingeltonSimple();
```

Mamy teraz dwie zmienne wskazujące na dwie różne instancje klasy Singelton. Aby temu zapobiec należy utworzyć prywatny konstruktor tej klasy zapobiegający tworzeniu obiektu operatorem *new*:

SingeltonWithPrivateConstructor.java

```
package eu.godlesie.jgdlws;
```

```
public class SingeltonWithPrivateConstructor {
    public static SingeltonWithPrivateConstructor singelton = new
        SingeltonWithPrivateConstructor();
    //prywatny konstruktor
    private SingeltonWithPrivateConstructor() {};
}
```

Wszystko byłoby w porządku, gdyby nie zalecenie projektowe, aby nie dawać dostępu do pól klasy. Dlatego należy do powyższej klasy dodać metodę statyczną zwracającą pole singelton, a samą zmienną zmienić na prywatną:

SIngeltonGetInstance.java

```
package eu.godlesie.jgdlws;
public class SingeltonGetInstance {
    private static final SingeltonGetInstance SINGELTON = new
        SingeltonGetInstance();
    private SingeltonGetInstance() {};
    public static SingeltonGetInstance getInstance() {
   return SINGELTON;
    }
}
```

Problem z tą wersja singletona polega na tym, że dany obiekt będzie tworzony natychmiast po załadowaniu klasy przez maszynę wirtualną javy nawet gdy obiekt nie będzie nigdy wywoływany. Należy zastosować tzw. leniwą inicjalizację:

SingeltonLazy.java

```
package eu.godlesie.jgdlws;
public class SingeltonLazy {
    private static SingeltonLazy singelton = null;
    private SingeltonLazy() {};
    public static SingeltonLazy getInstance() {
        if (singelton == null) {
            singelton = new SingeltonLazy();
        }
        return singelton;
    }
}
```

I zasadniczo to już wystarczy. Ostatnim etapem tworzenia poprawnego singeltona to uwzględnienie środowiska wielowątkowego jeśli jest taka konieczność. Istnieje prawdopodobieństwo że dwa wątki jednocześnie żądające obiektu

utworzy go dwukrotnie. Aby temu zapobiec należy zsynchronizować singeltona. Są dwa sposoby. Pierwszy prostszy, lecz obarczony kosztem takiej synchronizacji, drugi - bardziej rozwlekły - zwany też *podwójnym blokowaniem*: Pierwszy sposób:

SingeltonSynchronized.java

```
package eu.godlesie.jgdlws;
public class SingeltonSynchronized {
    private static SingeltonSynchronized singelton = null;
    private SingeltonSynchronized() {}
    public static synchronized SingeltonSynchronized getInstance() {
        if (singelton == null) {
            singelton = new SingeltonSynchronized();
   }
   return singelton;
    }
}
```

I druga metoda *podwójnego blokowania*:

SingeltonDoubleCheck.java

```
package eu.godlesie.jgdlws;
public class SingeltonDoubleCheck {
    private static SingeltonDoubleCheck singelton = null;
    private SingeltonDoubleCheck() {};
    public static SingeltonDoubleCheck getInstance() {
        if (singelton == null) {
            synchronized(SingeltonDoubleCheck.class) {
                if (singelton == null) {
                    singelton = new SingeltonDoubleCheck();
      }
            }
   }
   return singelton;
    }
}
```

Istnieje ostatni sposób, sposób dla bardzo leniwych i przeznaczony dla najprostszych zastosowań. Jak wiemy klasa typu wyliczeniowego Enum jest z zasady statyczna i finalna, więc zachowuje się de facto jako singelton. Jest ona też zabezpieczona na wypadek wielowątkowości. Oto przykład:

SingeltonEnum.java

```
package eu.godlesie.jgdlws;
public enum SingeltonEnum {
    INSTANCE;
    //jakas zmienna
    int value;
    //tutaj moze byc konstruktor
    SingeltonEnum() { this.value = 2; }
    public int getValue() { return this.value; };
}
```

```
   public static void main(String... args) {
      SingeltonEnum singelton = SingeltonEnum.INSTANCE;
      System.out.println(singelton.getValue());
   }
```

2 Fabryka

Wzorzec ten, znany także pod nazwą **metody fabrycznej** związany jest z kolekcją klas dziedziczących po jakiejś ogólnej klasie, a więc powiązanych ze sobą zasadą podobieństwa. Często nie wiemy także czy utworzyliśmy wszystkie klasy potomne, czy też niektóre z nich z czasem staną się niepotrzebne. Jako przykład, załóżmy, że produkujemy marmoladę (klasa bazowa) i mamy pewien katalog produktów (klasy potomne). Chcemy napisać program drukujący nalepki w zależności od produkowanego asortymentu.
Zacznijmy od napisania klasy bazowej. Jaki będzie jej rodzaj (klasa normalna, abstrakcyjna czy interfejs) zależy od konkrentej implementacji i osobistych preferencji. Dobre praktyki programowania obiektowego zalecają jednak utworzenie klasy abstrakcyjne bądź interfejsu.

Marmolada.java

```
package eu.godlesie.jgdlws;
public abstract class Marmolada {
    public abstract String getNazwa();
    public abstract int getOwocProcent();
    public abstract int getCukierProcent();
    @Override
    public String toString() {
        return getNazwa() +
                "\nProcent owowcu: " + getOwocProcent() +
```

```
                "\nProcent cukru: " + getCukierProcent();
    }
}
```

Nadpisana metoda *toString()* to tylko przykład jak wydrukować nalepkę, reszta metod zwraca nazwę i zawartość dżemu. Następnem etapem jest utworzenie konkretnych klas dla poszczególnych produktów:

Truskawkowa.java

```
package eu.godlesie.jgdlws;
public class Truskawkowa extends Marmolada{
    private final String nazwa = "Truskawkowa";
    private final int owocProcent;
    private final int cukierProcent;
    //konstruktor
    public Truskawkowa() {
   this.owocProcent = 30;
   this.cukierProcent = 25;
    }
    @Override
    public String getNazwa() { return this.nazwa; }
    @Override
    public int getOwocProcent() { return this.owocProcent; }
    @Override
    public int getCukierProcent() { return this.cukierProcent; }
}
```

Malinowa.java

```
package eu.godlesie.jgdlws;
public class Malinowa extends Marmolada {
    private final String nazwa = "Malinowa";
    private final int owocProcent;
    private final int cukierProcent;
    //konstruktor
    public Malinowa() {
        this.owocProcent = 13;
        this.cukierProcent = 45;
    }
    @Override
    public String getNazwa() { return this.nazwa; }
    @Override
    public int getOwocProcent() { return this.owocProcent; }
```

```
    @Override
    public int getCukierProcent() { return this.cukierProcent; }
}
```

Na razie wszystko wygląda klasycznie. Wzorzec rozpoczyna się w tym momencie, przez utworzenie *fabryki* wraz z jego *metodą fabryczną*:

MarmoladaFabryka.java

```
package eu.godlesie.jgdlws;
public class MarmoladaFabryka {
    public enum Typ{
        MALINOWA, TRUSKAWKOWA;
    }
    public static Marmolada getMarmolada(Typ typ) {
        Marmolada marmolada = null;
        switch (typ) {
            case MALINOWA:
                marmolada = new Malinowa();
                break;
            case TRUSKAWKOWA:
                marmolada = new Truskawkowa();
                break;
        }
        return marmolada;
    }
}
```

Fabryka została przygotowana pora więc na realizację:

FabricMethod.java

```
package eu.godlesie.jgdlws;
public class FabricMethod {
    public static void main(String[] args) {
        Marmolada truskawkowa =
            MarmoladaFabryka.getMarmolada(MarmoladaFabryka.Typ.TRUSKAWKOWA);
        System.out.println(truskawkowa);
        Marmolada malinowa =
            MarmoladaFabryka.getMarmolada(MarmoladaFabryka.Typ.MALINOWA);
        System.out.println(malinowa);
    }

}
```

Zaletą takiego podejścia jest przede wszystkim to, że możemy swobodnie dodawać nowe klasy pochodne klasy Marmolada uzupełniając jedynie typ wyliczeniowy i konstrukcję **switch**. Zaznaczam jednak, że typ wyliczeniowy i instrukcja switch to tylko sposób realizacji - można to zrobic dowolnie inaczej.
Przy okazji realizujemy pewną ważną zasadę programowania obiektowego - odwracanie zależności, czyli przekazywanie implementacji w stronę abstrakcji. Klienta realizującego program, w tym akurat przypadku metodę main nie interesuje jak metoda fabryczna jest realizowana. Ważne że działa.

3 Fabryka abstrakcyjna

Wzór ten jest rozszerzeniem **metody fabrycznej** i stanowi tak naprawdę *fabryką fabryk*. Zasadniczą zmianą w tym wzrocu jest usunięcie z kodu instrukcji warunkowych *if-else* (bądź instrukcji wyboru *switch*). Ponadto będziemy mogli zrezygnować z typu wyliczeniowego.
Zaczniemy podobnie jak w przypadku metody fabrycznej od abstrakcyjnej klasy bazowej:

Marmolada.java

```
package eu.godlesie.jgdlws;

public abstract class Marmolada {
    public abstract String getNazwa();
    public abstract int getOwowcProcent();
    public abstract int getCukierProcent();

    @Override public String toString() {
        return getNazwa() +
                "\nprocent owowcu: " + getOwowcProcent() +
                "\nprocent cukru: " + getCukierProcent();
    }
}
```

Dalej też jest podobnie - dowolna ilość podklas konkretnych:

Truskawkowa.java

```
package eu.godlesie.jgdlws;

public class Truskawkowa extends Marmolada{
    private final String nazwa;
```

```
    private final int owoc;
    private final int cukier;

    public Truskawkowa(int owoc, int cukier) {
        this.nazwa = "Truskawkowa";
        this.owoc = owoc;
        this.cukier = cukier;
    }

    @Override
    public String getNazwa() {
        return this.nazwa;
    }

    @Override
    public int getOwowcProcent() {
        return this.owoc;
    }

    @Override
    public int getCukierProcent() {
        return this.cukier;
    }

}
```

Malinowa.java

```
package eu.godlesie.jgdlws;

public class Malinowa extends Marmolada{
    private final String nazwa;
    private final int owoc;
    private final int cukier;

    public Malinowa(int owoc, int cukier) {
        this.nazwa = "Malinowa";
        this.owoc = owoc;
        this.cukier = cukier;
    }

    @Override
    public String getNazwa() {
```

```
        return this.nazwa;
    }

    @Override
    public int getOwowcProcent() {
        return this.owoc;
    }

    @Override
    public int getCukierProcent() {
        return this.cukier;
    }

}
```

Tu następuje pierwsza zmiana. Tworzymy interfejs dla dka głównej fabryki:

MarmoladaFabrykaAbstrakcyjna.java

```
package eu.godlesie.jgdlws;

public interface MarmoladaFabrykaAbstrakcyjna {
    public Marmolada createMarmolada();
}
```

Interfejs ten potrzebny będzie przy tworzeniu fabryk konkretnych:

TruskawkowaFabryka.java

```
package eu.godlesie.jgdlws;

public class TruskawkowaFabryka implements
    MarmoladaFabrykaAbstrakcyjna {
    private final int owoc;
    private final int cukier;

    public TruskawkowaFabryka(int owoc, int cukier) {
        this.owoc = owoc;
        this.cukier = cukier;
    }

    @Override
    public Marmolada createMarmolada() {
        return new Truskawkowa(owoc, cukier);
    }
```

```
}
```

MalinowaFabryka.java

```
package eu.godlesie.jgdlws;

public class MalinowaFabryka implements
    MarmoladaFabrykaAbstrakcyjna {
    private final int owoc;
    private final int cukier;

    public MalinowaFabryka(int owoc, int cukier) {
        this.owoc = owoc;
        this.cukier = cukier;
    }

    @Override
    public Marmolada createMarmolada() {
        return new Truskawkowa(owoc, cukier);
    }


}
```

Ostatnim krokiem jest utworzenie klasy głóœnej fabryki:

MarmoladaFabryka.java

```
package eu.godlesie.jgdlws;

public class MarmoladaFabryka {
    public static Marmolada
        getMarmolada(MarmoladaFabrykaAbstrakcyjna fabryka) {
        return fabryka.createMarmolada();
    }
}
```

Pozostaje nam tylko przetestować utworzoną fabrykę abstrakcyjną:

FabrykaAbstrakcyjna.java

```
package eu.godlesie.jgdlws;
public class FabrykaAbstrakcyjna {
```

```
    public static void main(String[] args) {
        Marmolada truskawkowa = MarmoladaFabryka.getMarmolada(new
            TruskawkowaFabryka(25, 30));
        Marmolada malinowa = MarmoladaFabryka.getMarmolada(new
            MalinowaFabryka(36,50));
        System.out.println(truskawkowa);
        System.out.println(malinowa);
    }

}
```

[1].

4 Budowniczy (Builder)

Ten wzorzec został wprowadzony w celu rozwiązania pewnych problemów ze wzorami projektowymi **Fabryki** i **Fabryki Abstrakcyjnej**. Gdy obiekt zawiera wiele atrybutów, wzorzec **Builder** rozwiązuje problem z dużą liczbą opcjonalnych parametrów i niespójnym stanem[1].
Czasem zachodzi potrzeba tworzenia wielu obiektów tej samej klasy lecz każdy z nich może mieć różne, opcjonalne, wewnętrzne stany. Pierwszym, co przychodzi nam do głowy jest metoda przeciążania konstruktorów z coraz większą liczbą parametrów. Takie konstruktory zwane są teleskopowymi, ze względu na coraz bardziej wydłużającą się jego sygnaturę:

NalesnikTeleskop.java

```
package eu.godlesie.jgdlws;
public class NalesnikTeleskop {
    int wielkosc;
    String nadzienie;
    String sos;
    NalesnikTeleskop() {
        this.wielkosc = 24;
        this.nadzienie = "standardowe";
        this.sos = "lagodny";
    };
    NalesnikTeleskop(int wielkosc, String nadzienie) {
        this.wielkosc = wielkosc;
        this.nadzienie = nadzienie;
        this.sos = "lagodny";
    };
```

```
    NalesnikTeleskop(int wielkosc, String nadzienie, String sos) {
        this.wielkosc = wielkosc;
        this.nadzienie = nadzienie;
        this.sos = sos;
    }
}
```

O ile takie rozwiązanie jest poprawne nie tylko nie wygląda zbyt elegancko, ale jest przede wszystkim nieczytelne. Zbyt wiele takich konstrukcji wprowadza zamęt i trzeba bardzo uważać na kolejność wywoływanych parametrów.
Innym, lepszym sposobem na poradzenie sobie z taką nieczytelnością są *gettery* i *settery*.

NalesnikGetSet.java

```
package eu.godlesie.jgdlws;
public class NalesnikGetSet {
    private int wielkosc;
    private String nadzienie;
    private String sos;
    public void setWielkosc(int wielkosc) { this.wielkosc =
        wielkosc;}
    public int getWielkosc() { return this.wielkosc; }
    public void setNadzienie(String nadzienie) {
        this.nadzienie = nadzienie;
    }
    public String getNadzienie() { return this.nadzienie; }
    public void setSos(String sos) {
        this.sos = sos;
    }
    public String getSos() { return this.sos; }
}
```

W klasie tej występuje niejawnie konstruktor domyślny inicjujący wszystkie pola wartościami domyślnymi - w przypadku *int* jest to 0, w przypadku *String* pusty łańcuch. Inicjacja obiektu będzie wyglądać następująco:

```
    Nalesnik nalesnik = new Nalesnik();
    nalesnik.setWielkosc(24);
    nalesnik.setNadzienie("miesne");
    nalesnik.setSos("pikantny");
```

Wygląda świetnie i często wystarcza do tworzenia prostych klas. Problem

pojawia się w momencie współbieżności. Do momentu wywołania ostatniej metody *set* obiekt jest w niespójnym stanie - inny wątek może zakłócić tworzenie obiektu.
I tutaj z pomocą przychodzi wzorzec projektowy budowniczego **Builder**.

Nalesnik.java

```
package eu.godlesie.jgdlws;
public class Nalesnik {
    private final int wielkosc;
    private final String nadzienie;
    private final String sos;

    //konstruktor nalesnika
    private Nalesnik(Builder builder) {
        this.wielkosc = builder.wielkosc;
        this.nadzienie = builder.nadzienie;
        this.sos = builder.sos;
    }
    public static class Builder {
        //parametr wymagany
        private final int wielkosc;
      //parametry opcjonalne
        private String nadzienie = "miesne";
        private String sos = "lagodny";
      //konstruktor
        public Builder(int wielkosc) {
            this.wielkosc = wielkosc;
        }
      //metody ustawiajace
        public Builder nadzienie(String wartosc) {
            nadzienie = wartosc; return this;
        }
        public Builder sos(String wartosc) {
            sos = wartosc; return this;
        }
      //metoda tworzaca
        public Nalesnik build() {
            return new Nalesnik(this);
        }
    }
    // tylko gettery do pobrania wartosci
    // ...
    }
```

Przyjrzyjme się tej klasie dokładniej. Klasa główna posiada sfinalizowane pola stanu. Nie muszą być one finalne (niezmienne), tym sposobem jednak zapewniamy zalecaną niezmienność obiektu utworzonego.
W następnej kolejności piszemy wewnętrzną, statyczną i publiczną klasę **Builder**. Oczywiście może nazywać się dowolnie, nazwa taka zapewnia czytelność. Definiujemy w niej ponownie wszystkie pola klasy głównej, przy czym te, które są obowiązkowe tworzymy jako finalne, te opcjonalne jako zwykłe oraz inicjujemy ich wartości domyślne. W ciele klasy **Builder** muszą znaleźć się także metody ustawiające wartości domyślne (w tym przypadku **nadzienie()** i **sos()**), zwracające jednak obiekt klasy **Builder**. Zwieńczeniem klasy jest metoda **buid()** zwracająca typ klasy głównej (**Nalesnik**)

5 Prototyp

W naszej fabryce marmlolady produkuje się nie tylko główne marmoladę truskawkową i malinową. Są najróżniejsze ich odmiany różniące się jedynie drobnymi szczegółami, np. odbiorcą (mogą być różne nalepki), lub procentem cukru (np. BIO). A niekiedy nasze klasy mogą być bardzo duże i kosztowne podczas tworzenia. Skoro mamy więc już podstawowy obiekt (właśnie prototyp) możemy skopiować (choć właściwsze określenie to - sklonować) go do nowego i zmodyfikować jedynie jej niektóre elementy. W pierwszej kolejności zmodyfikujemy naszą klasę abstrakcyjną:

Marmolada.java

```
package eu.godlesie.jgdlws;

public abstract class Marmolada implements Cloneable {
    public abstract String getNazwa();
    public abstract int getOwowcProcent();
    public abstract int getCukierProcent();
    public abstract void setNazwa(String nazwa);
    public abstract void setOwocProcent(int owoc);
    public abstract void setCukierProcent(int cukier);

    @Override public String toString() {
        return getNazwa() +
                "\nprocent owowcu: " + getOwowcProcent() +
                "\nprocent cukru: " + getCukierProcent();
    }
    @Override
    public Marmolada clone() throws CloneNotSupportedException {
```

```
        return (Marmolada)super.clone();
    }
}
```

Pojawiły się w niej trzy nowe elementy. Po pierwsze klasa ta implementuje interfejs *Cloneble*, który pozwala na klonowanie obiektów. Drugim elementem jest nadpisanie metody *clone()* klasy *Object*. Zrobimy to z dwóch powodów. Metoda *clone()* jest zabezpieczona (*protected*), a więc niewidoczna z pakietu, dlatego tworzymy ją jako publiczną (*public*). Po drugie operacja ta bez problemu klonuje obiekty pola proste (np. *int*, *String*, *double*) i niezmienialne. Jednak gdy używamy pól typu *ArrayList* musimy użyć tzw klonowania głębokiego:

Prcawnicy.java

```
   //definicja klasy ...
   private List<String> listaPracownikow = new ArrayList<>();
   //Wypelnienie listy
   @Override
   public Object clone() throws CloneNotSupportedException{
      List<String> temp = new ArrayList<>();
      for(String s : listaPracownikow){
         temp.add(s);
      }
      //Klas
      return new pracownik(temp);
   }
```

Trzecim elementem, który musimy wziąć pod uwagę to uwzględnienie wyjątku *CloneNotSupportedException*. Umożliwia to reakcję na sytuację, gdy klasa nie może zostać sklonowana. Co prawda poprzez implementacje interfejsu *Cloneable* tak sytuacja nie zaistnieje, ale kompilator o tym nie wie.
Zmienić się musi także konkretna klasa - Dodajemy odpowiednie settery, aby można było w przyszłości zmienić sklonowany obiekt:

Truskawkowa.java

```
package eu.godlesie.jgdlws;
public final class Truskawkowa extends Marmolada{
    private String nazwa;
    private int owoc;
    private int cukier;

    public Truskawkowa(int owoc, int cukier) {
```

```
        this.nazwa = "Truskawkowa";
        this.owoc = owoc;
        this.cukier = cukier;
    }

    @Override
    public String getNazwa() {
        return this.nazwa;
    }

    @Override
    public int getOwowcProcent() {
        return this.owoc;
    }

    @Override
    public int getCukierProcent() {
        return this.cukier;
    }

    @Override
    public void setNazwa(String nazwa) {
        this.nazwa = nazwa;
    }

    @Override
    public void setOwocProcent(int owoc) {
        this.owoc = owoc;
    }

    @Override
    public void setCukierProcent(int cukier) {
        this.cukier = cukier;
    }
}
```

Na koniec przetestujmy nasz prototyp:

Prototyp.java

```
package eu.godlesie.jgdlws;
public class Prototyp {
    public static void main(String[] args) throws
        CloneNotSupportedException {
```

```
        Marmolada truskawkowa = MarmoladaFabryka.getMarmolada(new
            TruskawkowaFabryka(12, 30));
        System.out.println(truskawkowa);

        Marmolada bio = truskawkowa.clone();
        bio.setNazwa("Truskawkowa BIO");
        bio.setCukierProcent(0);
        bio.setOwocProcent(80);
        System.out.println(bio);
    }
}
```

Część II
Wzorce strukturalne

Opisuje struktury powiązanych ze sobą obiektów i dotyczą składania klas i obiektów w większe struktury. Dzielą się one na dwa rodzaje:

- klasowe wzorce strukturalne - oparte są na wykorzystaniu dziedziczenia do składania interfejsów i implementacji,
- obiektowe wzorce strukturalne - opisują sposoby składania obiektów w celu obsługi nowych funkcji.[3]

6 Adapter

Zadaniem tego wzorca jest przekształcenie jednego interfejsu w drugi. To tak naprawdę konwerter, który przekształca dane wejściowe w wyjściowe posługując się do tego specjalnym mechanizmem. Dlaczego nie robić tego bezpośrednio? Istnieją sytuacje, w których format danych wejściowych się zmienia. Np. klient zamawiający naszą Marmoladę zmieni formularz elektronicznego zamówienia. Zmiana wszystkich miejsc w których występuje konwersja jest wysoce narażona na błędy. Poza tym możemy pozyskać nowego klienta, który ma kompletnie inny system zamówień. W pierwszym przypadku wystarczy zmodyfikować tylko istniejący adapter, w drugim napisać drugi i posługiwać się nim w ramach potrzeb.
Jako przykład rozważmy następującą sytuację: Mamy dwóch klientów zamawiających taką samą marmoladę truskawkową. Przy czym jeden zamawia

pewną ilość słoików o wadze 450 gram (i podaje tylko ich ilość), drugi natomiast chce wiaderka o pojemności 3 kg. My natomiast na podstawie wielkości zamówienia potrzeby produkcji potrzebujemy ilości potrzebnych produktów, aby zrealizować konkretne zamówienie. Zauważmy, że klient w każdej chwili może zmienić swoje wymagania np. na słoik 430 gramów lub wiaderko 5 kg. Na początek stwórzmy interfejs reprezentujący dostarczone zamówienie. Te zamówienia będą potem przekształcane na konkretną recepturę :

Zamowienie.java

```
package eu.godlesie.jgdlws;
public interface Zamowienie {
    public double getKilogramy();
}
```

Teraz możemy stworzyć kilka (w naszym przypadku dwie) konkretne klasy implementujące nasz interfejs

Sloiki450.java

```
package eu.godlesie.jgdlws;

public class Sloiki450 implements Zamowienie{
    private final double waga;
    public Sloiki450(int iloscSloikow) {
        this.waga = iloscSloikow * 0.45;
    }

    @Override
    public double getKilogramy() {
        return this.waga;
    }
}
```

Wiaderko3.java

```
package eu.godlesie.jgdlws;

public class Wiaderko3 implements Zamowienie{
    private final double waga;
    public Wiaderko3(int ilosc) {
        this.waga = ilosc * 3;
    }

    @Override
```

```
    public double getKilogramy() {
        return this.waga;
    }

}
```

Zaczynamy część drugą część projektu. I znowu interfejs reprezentujący naszą recepturę:

Receptura.java

```
package eu.godlesie.jgdlws;
public interface Receptura {
    public double getKilogramy();
}
```

No doszliśmy do właściwego adaptera. Takich adapterów może być wiele, a zależności od okoliczność.

ZamowieniaNaKilogramy.java

```
package eu.godlesie.jgdlws;

public class ZamowieniaNaKilogramy implements Receptura{
    private final double kilogramy;
    private final Zamowienie zamowienie;

    public ZamowieniaNaKilogramy(Zamowienie zamowienie) {
        this.zamowienie = zamowienie;
        this.kilogramy = zamowienie.getKilogramy();
    }

    @Override
    public double getKilogramy() {
        return kilogramy;
    }

}
```

Przetestujmy nasz adapter. Warto zauważyć, że w każdej chwili możemy dopisać zarówno nowe implementacje naszego zamówienia (np mniejszego słoika lub wiaderka) czy też adapter na różne receptury nie modyfikując praktycznie nic w kodzie klienckim:

AdapterClass.java

```
package eu.godlesie.jgdlws;
public class AdapterClass {
    public static void main(String[] args) {
        Receptura sloiki = new ZamowieniaNaKilogramy(new
            Sloiki450(1000));
        System.out.println(sloiki.getKilogramy() + " kg");
        Receptura wiaderko = new ZamowieniaNaKilogramy(new
            Wiaderko3(256));
        System.out.println(wiaderko.getKilogramy() + " kg");
    }

}
```

7 Kompozyt

To kolejny wzorzec strukturalny, stosowany wtedy, gdy musimy reprezentować hierarchię obiektów, oraz gdy trzeba na niej całej wykonać jakieś operacje. Inaczej mówiąc wszystkie elementy tej struktury mogą być potraktowane w taki sam sposób. Hierarchia ta tworzy drzewo, w którym występują węzły i tzw. liście. Liście te pojedyncze obiekty, węzły to elementy zawierające liście i / lub kolejne węzły.
Wyobraźmy sobie, że w naszej fabryce marmolady mamy już zbiór produktów, które chcemy zaprezentować swoim nowym klientom. Produkty te należą do pewnych kategorii, podkategorii itd. Przykładowo - główny podział może być na konfitury BIO i ńormalne". W ramach tych dwóch kategorii jest podział na

Wzór złożony jest jednym ze wzorów projektowania strukturalnego i jest używany, gdy musimy reprezentować całą hierarchię. Kiedy musimy stworzyć strukturę w taki sposób, że obiekty w strukturze muszą być traktowane w ten sam sposób, możemy zastosować złożony wzór projektu.
Rozumiemy to za pomocą prawdziwego przykładu - diagram to struktura składająca się z obiektów, takich jak okrąg, linie, trójkąt itp., A kiedy wypełnimy rysunek kolorem (np. Czerwony), ten sam kolor zostanie również zastosowany do obiektów w rysunek. Tutaj rysunek składa się z różnych części i wszystkie mają te same operacje. Sprawdź artykuł Wzór złożony dla różnych komponentów złożonego wzoru i przykładowego programu[1].

8 Proxy

Intencją wzorca proxy jest „Zapewnienie zastępczego lub zastępczego dla innego obiektu w celu kontrolowania dostępu do niego". Sama definicja jest bardzo jasna, a wzór proxy jest używany, gdy chcemy zapewnić kontrolowany dostęp do funkcjonalności.
Powiedzmy, że mamy klasę, która może uruchomić pewne polecenie w systemie. Teraz, jeśli go używamy, jest dobrze, ale jeśli chcemy przekazać ten program do aplikacji klienckiej, może to mieć poważne problemy, ponieważ program kliencki może wydać polecenie usunięcia niektórych plików systemowych lub zmienić niektóre ustawienia, których nie chcesz. Sprawdź post wzorca proxy dla przykładowego programu ze szczegółami implementacji[1].

9 Flyweight

Wzór konstrukcji Flyweight jest używany, gdy musimy stworzyć wiele obiektów klasy. Ponieważ każdy obiekt zużywa miejsce w pamięci, które może mieć kluczowe znaczenie dla urządzeń o małej pamięci, takich jak urządzenia mobilne lub systemy wbudowane, wzorzec konstrukcji z wagą można zastosować w celu zmniejszenia obciążenia pamięci poprzez udostępnianie obiektów. Implementacja puli ciągów w java jest jednym z najlepszych przykładów implementacji wzorca Flyweight. Zapoznaj się z artykułem Flyweight Pattern dla przykładowego programu i procesu wdrażania[1].

10 Fasada

Wzór elewacji służy do ułatwienia aplikacjom klienckim łatwej interakcji z systemem. Załóżmy, że mamy aplikację z zestawem interfejsów do korzystania z bazy danych MySql / Oracle i generowania różnych typów raportów, takich jak raport HTML, raport w formacie PDF itp. Będziemy więc mieli inny zestaw interfejsów do pracy z różnymi typami baz danych. Teraz aplikacja kliencka może korzystać z tych interfejsów, aby uzyskać wymagane połączenie z bazą danych i generować raporty. Ale kiedy złożoność wzrasta lub nazwy zachowań interfejsu są mylące, aplikacja kliencka będzie miała trudności z zarządzaniem. Możemy więc zastosować wzór fasadowy i udostępnić interfejs opakowania na istniejącym interfejsie, aby pomóc aplikacji klienckiej. Zapoznaj się z postem na temat elewacji, aby poznać szczegóły implementacji i przykładowy program[1].

11 Most

Gdy mamy hierarchie interfejsów zarówno w interfejsach, jak iw implementacjach, to wzorzec projektowy mostu jest używany do oddzielenia interfejsów od implementacji i ukrycia szczegółów implementacji z programów klienckich. Podobnie jak wzór adaptera, jest to jeden ze wzorów projektowania strukturalnego.
Implementacja wzorca projektu mostu podąża za koncepcją preferowania kompozycji nad dziedziczeniem. Sprawdź post Bridge Pattern, aby poznać szczegóły implementacji i przykładowy program[1].

12 Dekorator

Wzorzec projektowy dekoratora służy do modyfikowania funkcjonalności obiektu w czasie wykonywania. W tym samym czasie inne instancje tej samej klasy nie będą miały na to wpływu, więc pojedynczy obiekt otrzyma zmodyfikowane zachowanie. Wzorzec projektanta dekoratora jest jednym ze wzorów konstrukcyjnych (takich jak Wzór adaptera, Wzór mostu, Wzór złożony) i wykorzystuje klasy abstrakcyjne lub interfejs z kompozycją do wdrożenia.
Używamy dziedziczenia lub kompozycji w celu rozszerzenia zachowania obiektu, ale odbywa się to w czasie kompilacji i ma zastosowanie do wszystkich instancji klasy. Nie możemy dodawać żadnych nowych funkcji, aby usunąć jakiekolwiek istniejące zachowanie w czasie wykonywania - to jest, gdy wzorzec Decorator pojawia się na zdjęciu. Sprawdź pocztę Decorator Pattern, aby zobaczyć przykładowy program i szczegóły implementacji[1].

Część III
Wzorce czynnościowe

Opisuje zachowanie i odpowiedzialność współpracujących ze sobą obiektów

13 Template Method

Metoda szablonu jest behawioralnym wzorcem projektowym i służy do tworzenia kodu pośredniego i odroczenia niektórych kroków implementacji do podklas. Metoda szablonowa definiuje kroki do wykonania algorytmu i może zapewnić domyślną implementację, która może być wspólna dla wszystkich lub niektórych podklas.

Załóżmy, że chcemy dostarczyć algorytm do budowy domu. Czynności, które należy wykonać, aby zbudować dom, to: budowa fundamentów, budowa filarów, ścian budynków i okien. Ważne jest to, że nie możemy zmienić kolejności wykonywania, ponieważ nie możemy zbudować okien przed zbudowaniem fundamentu. W tym przypadku możemy stworzyć metodę szablonową, która wykorzysta różne metody do budowy domu. Sprawdź szablon Pattern Method Pattern, aby zobaczyć szczegóły implementacji z przykładowym programem[1].

14 Mediator

Wzorzec projektowy mediatora służy do zapewnienia scentralizowanego medium komunikacyjnego między różnymi obiektami w systemie. Wzorzec projektowania mediatora jest bardzo pomocny w aplikacji korporacyjnej, w której wiele obiektów współdziała ze sobą. Jeśli obiekty oddziałują ze sobą bezpośrednio, komponenty systemu są ze sobą ściśle powiązane, co sprawia, że łatwość konserwacji jest wyższa i nie jest elastyczna, aby można ją było łatwo rozszerzyć. Wzorzec mediatora koncentruje się na dostarczeniu mediatora między obiektami do komunikacji i pomocy we wdrażaniu utraconego sprzężenia między obiektami.
Kontroler ruchu lotniczego jest doskonałym przykładem wzoru mediatora, w którym pokój kontroli lotniska działa jako mediator do komunikacji między różnymi lotami. Mediator działa jako router między obiektami i może mieć własną logikę zapewniającą sposób komunikacji. Sprawdź post Mediator Pattern, aby poznać szczegóły implementacji z przykładowym programem[1].

15 Łańcuch odpowiedzialności

Wzór łańcucha odpowiedzialności jest wykorzystywany do uzyskania luźnego połączenia w projektowaniu oprogramowania, gdy żądanie klienta jest przekazywane do łańcucha obiektów w celu ich przetworzenia. Następnie obiekt w łańcuchu sam zadecyduje, kto będzie przetwarzał żądanie i czy żądanie musi zostać wysłane do następnego obiektu w łańcuchu, czy nie.
Wiemy, że możemy mieć wiele bloków catch w kodzie bloku try-catch. Tutaj każdy blok catch jest rodzajem procesora przetwarzającego ten wyjątek. Tak więc, gdy wystąpi wyjątek w bloku try, zostanie on wysłany do pierwszego bloku catch do przetworzenia. Jeśli blok catch nie jest w stanie go przetworzyć, przekazuje żądanie do następnego obiektu w łańcuchu, tj. Następnego bloku catch. Jeśli nawet ostatni blok catch nie jest w stanie go przetworzyć,

wyjątek jest generowany poza łańcuchem do programu wywołującego.

Logikę maszyny wydającej bankomaty można zaimplementować za pomocą schematu łańcucha odpowiedzialności, sprawdź połączony post[1].

16 Obserwator

Wzorzec projektowy obserwatora jest przydatny, gdy interesuje Cię stan obiektu i chcesz otrzymywać powiadomienia o każdej zmianie. W układzie obserwatora obiekt obserwujący stan innego obiektu jest nazywany obserwatorem, a obserwowany obiekt jest nazywany obiektem.

Java zapewnia wbudowaną platformę do implementacji wzorca Observer za pośrednictwem interfejsu klasy java.util.Observable i java.util.Observer. Jednak nie jest powszechnie używany, ponieważ implementacja jest naprawdę prosta i przez większość czasu nie chcemy rozszerzać klasy tylko do implementacji wzorca Observer, ponieważ java nie zapewnia wielu dziedziczeń w klasach.
Java Message Service (JMS) używa wzorca Observer wraz z wzorcem Mediator, aby umożliwić aplikacjom subskrybowanie i publikowanie danych w innych aplikacjach. Sprawdź post wzorca Observer, aby poznać szczegóły implementacji i przykładowy program[1].

17 Strategia

Wzorzec strategii jest używany, gdy mamy wiele algorytmów dla konkretnego zadania, a klient decyduje o rzeczywistej implementacji, która ma być używana w czasie wykonywania.
Wzorzec strategii jest również znany jako wzorzec zasad. Definiujemy wiele algorytmów i pozwalamy aplikacji klienckiej przekazać algorytm, który ma zostać użyty jako parametr. Jednym z najlepszych przykładów tego wzorca jest metoda Collections.sort (), która pobiera parametr Komparator. W oparciu o różne implementacje interfejsów Komparatora, Obiekty są sortowane na różne sposoby.
Zapoznaj się z postem dotyczącym strategii dla szczegółów implementacji i przykładowego programu[1].

18 Polecenia

Wzorzec polecenia służy do implementacji utraconego sprzężenia w modelu żądanie-odpowiedź. We wzorcu poleceń żądanie jest wysyłane do wywołującego, a wywołujący przekazuje go do hermetyzowanego obiektu polecenia. Obiekt polecenia przekazuje żądanie do odpowiedniej metody odbiornika, aby wykonać określoną akcję.
Powiedzmy, że chcemy udostępnić narzędzie systemu plików z metodami otwierania, zapisywania i zamykania pliku i powinno ono obsługiwać wiele systemów operacyjnych, takich jak Windows i Unix.
Aby zaimplementować nasze narzędzie systemu plików, musimy przede wszystkim utworzyć klasy odbiorników, które faktycznie wykonają całą pracę. Ponieważ kodujemy w kategoriach interfejsów Java, możemy mieć interfejs FileSystemReceiver i jest to klasa implementacji dla różnych smaków systemu operacyjnego, takich jak Windows, Unix, Solaris itp. Sprawdź post wzorca poleceń dla szczegółów implementacji z przykładowym programem[1].

19 Stan

Wzorzec projektu stanu jest używany, gdy obiekt zmienia swoje zachowanie na podstawie stanu wewnętrznego.
Jeśli musimy zmienić zachowanie obiektu na podstawie jego stanu, możemy mieć zmienną stanu w bloku Object i użyć warunku if-else, aby wykonywać różne akcje w zależności od stanu. Wzorzec stanu służy do zapewnienia systematycznego i luźno powiązanego sposobu osiągnięcia tego poprzez implementację kontekstu i stanu.
Sprawdź stanowisko State Pattern, aby zobaczyć szczegóły implementacji z przykładowym programem[1].

20 Visitor

Wzór odwiedzin jest używany, gdy musimy wykonać operację na grupie podobnych obiektów. Za pomocą wzorca odwiedzającego możemy przenieść logikę operacyjną z obiektów do innej klasy.

Na przykład zastanów się nad koszykiem na zakupy, w którym możemy dodać inny rodzaj przedmiotów (elementy), a gdy klikniemy przycisk kasy, obliczy całkowitą kwotę do zapłaty. Teraz możemy mieć logikę obliczeń w klasach przedmiotów lub możemy przenieść tę logikę do innej klasy przy użyciu wzorca odwiedzającego. Zaimplementujmy to w naszym przykładzie

wzorca odwiedzającego. Sprawdź stanowisko wzorca odwiedzającego, aby poznać szczegóły implementacji[1].

21 Interpretator

służy do definiowania reprezentacji gramatycznej języka i zapewnia tłumacza do radzenia sobie z tą gramatyką.
Najlepszym przykładem tego wzorca jest kompilator Java, który interpretuje kod źródłowy Java w kodzie bajtowym zrozumiałym dla JVM. Google Translator jest również przykładem wzorca interpretatora, w którym dane wejściowe mogą być w dowolnym języku i możemy uzyskać dane wyjściowe interpretowane w innym języku[1].

22 Iterator

Wzór iteratora w jednym z wzorców behawioralnych i służy do zapewnienia standardowego sposobu przechodzenia przez grupę obiektów. Wzorzec iteratora jest szeroko stosowany w Java Collection Framework, gdzie interfejs Iterator zapewnia metody przechodzenia przez kolekcję.
Wzorzec iteratora to nie tylko przechodzenie przez kolekcję, ale możemy również zapewnić różne rodzaje iteratorów w oparciu o nasze wymagania. Wzorzec iteratora ukrywa rzeczywistą implementację przechodzenia przez kolekcję, a programy klienckie używają metod iteratora. Sprawdź post Iterator Pattern, na przykład szczegóły programu i implementacji[1].

23 Memento

Wzór projektu memento jest używany, gdy chcemy zapisać stan obiektu, abyśmy mogli go później przywrócić. Wzorzec memento jest wykorzystywany do zaimplementowania tego w taki sposób, że zapisane dane stanu obiektu nie są dostępne poza obiektem, co chroni integralność zapisanych danych stanu. Wzór pamiątkowy jest realizowany za pomocą dwóch obiektów - inicjatora i dozorcy. Pomysłodawcą jest obiekt, którego stan wymaga zapisania i odtworzenia, i używa klasy wewnętrznej do zapisania stanu obiektu. Wewnętrzna klasa nazywa się Memento i jest prywatna, więc nie można uzyskać do niej dostępu z innych obiektów[1].

Część IV
Antywzorce projektowe

Część V
Wzorce architektoniczne

Część VI
Inne Wzorce

24 DAO

Wzorzec projektowania DAO służy do oddzielenia logiki trwałości danych od oddzielnej warstwy. DAO to bardzo popularny wzór, gdy projektujemy systemy do pracy z bazami danych. Pomysł polega na oddzieleniu warstwy usług od warstwy dostępu do danych. W ten sposób wdrażamy Separation of Logic w naszej aplikacji[1].

25 Wstrzykiwanie zależności

Dependency Injection pozwala nam usunąć zależne od siebie na stałe zależności i sprawić, że nasza aplikacja będzie luźno połączona, rozszerzalna i możliwa do utrzymania. Możemy zaimplementować wstrzykiwanie zależności w Javie, aby przenieść rozdzielczość zależności z czasu kompilacji na środowisko wykonawcze. Spring Framework jest zbudowany na zasadzie wstrzykiwania zależności[1].

26 MVC

Wzór MVC jest jednym z najstarszych wzorów architektonicznych do tworzenia aplikacji internetowych. MVC to skrót od Model-View-Controller[1].

Spis treści

Literatura

[1] Pankaj Kumar. Java design patterns – example tutorial.

[2] Rohit Joshi. *Java Design Patterns. Reusable Solutions to Common Problems.*

[3] Ralph Johnson John Vlissides Erich Gamma, Richard Helm.