Design Principles

Spis treści

Ten warsztat będzie lekko inny niż wszystkie, bo będzie o wiele bardziej teoretyczny. Pokażemy tutaj zagadnienia, które należy przyswoić i pamiętać. Dużo z nich będzie wymagało od Ciebie przypomnienia sobie wielokrotnie zanim dobrze wejdą w pamięć (czy tylko ja tak mam? 😁).

Czym są Design Principles

Design Principles (zasady projektowania) są abstrakcyjnymi zasadami, których należy przestrzegać podczas pisania oprogramowania. (Wiem, że dalej Ci to nic nie mówi ☺). Inaczej mówiąc są praktykami, które powinny być stosowane podczas projektowania i wytwarzania aplikacji, jeżeli chcemy powiedzieć, że mamy dobry design.

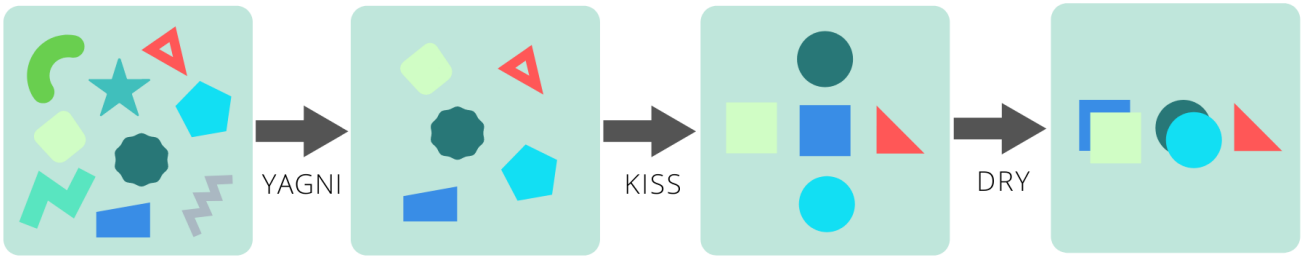
Design Principles są niezależne od języka programowania i są bardziej abstrakcyjnymi stwierdzeniami

niż **Design Patterns**, które poruszymy w następnej kolejności. Abstrakcyjnymi w tym rozumieniu, że **Design Patterns** mówią o wzajemnej zależności klas, podczas gdy **Design Principles** są generalnymi dobrymi praktykami przy pisaniu kodu.

Przykładami Design Principles mogą być:

- enkapsulacja - pola definiujemy jako prywatne i wystawiamy metody dostępowe,
- nasze klasy powinny być zależne od interfejsów, a nie od konkretnych implementacji,
- composition over inheritance - uzależniajmy się od innych klas definiując je jako pola w klasie niż rozszerzając inne klasy.

Poniżej chciałbym omówić często spotykane i istotne **Design Principles**, oczywiście nie są to wszystkie jakie można znaleźć w internecie.



Obraz 1. YAGNI → KISS → DRY design principles

DRY

DRY (Don't Repeat Yourself Principle). Wiki. Zasada ta mówi, że:

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

Czyli tłumacząc na nasz, jeżeli piszemy jakąś logikę w kodzie, to ma mieć ona tylko jedno źródło prawdy w naszej aplikacji. Nie piszemy obok metody, która w sumie robi to samo i jest kopią metody z innego miejsca bo nie chciało nam się tego posprzątać 😄.

Jak to się przekłada w praktyce na pisanie kodu w Javie? Używajmy interfejsów, zmiennych `public static final`, klas abstrakcyjnych jeżeli mamy jakieś fragmenty kodu, które mogą być wydzielone do wspólnego miejsca:

```
public abstract class Animal {
    public void eat() {
        System.out.println("eating");
    }
}
```

```
public class Dog extends Animal {
    public void bark() {
        System.out.println("Barking!");
    }
```

```
}
```

```
public class Cat extends Animal {
    public void meow() {
        System.out.println("Meowing!");
    }
}
```

Zarówno kot i pies jedzą, zatem logikę odpowiadającą za jedzenie wyciągamy do wspólnego miejsca, które jest dostępne dla obu tych klas. Natomiast kot nie szczeka, a pies nie miauczy, więc metody dotyczące szczekania i miauczenia mogą się znaleźć w dedykowanych klasach.

Nie musi być to zawsze dziedziczenie, później opiszę zasadę **composition over inheritance**. Możemy równie dobrze stworzyć metodę statyczną i umieścić ją w dedykowanej klasie, a następnie wykorzystać ją ze wszystkich miejsc, w których jest potrzebna. Ważne, żeby "logika', która jest realizowana przez tę metodę statyczną była tylko w jednym miejscu, które może być reużyte.

W projektach często pojawia się duża ilość wartości, które również mogą zostać reużyte przy wykorzystaniu zmiennych `public static final`.

```
class Constants {
    public static final Integer MAX_LIST_SIZE = 1000;
    public static final String OWNER = "my_owner";
    public static final Double FACTOR = 1.9;
}
```

Dzięki temu, że wyciągnęliśmy zmienne do jednego miejsca w kodzie, łatwiej jest odnaleźć źródło wartości na wypadek wprowadzania poprawek w kodzie. Jeżeli te wartości byłyby rozsiane po wszystkich plikach, bardzo łatwo byłoby popełnić błąd. Zwróć też uwagę, że wartości te są `final` żebyśmy przez przypadek nie przypisali wartości do tych referencji gdzieś w kodzie.

Głównym celem **DRY** jest ułatwienie utrzymania kodu. Jeżeli zmieniamy jakąś funkcjonalność to stosując **DRY** mamy ten komfort, że modyfikujemy ją tylko w jednym miejscu. Jeżeli poprawiamy jakiś błąd to działa to w ten sam sposób, unikamy duplikatów i zmniejszamy ryzyko popełnienia błędu. Jeżeli kod jest rozsiany po wielu miejscach, zawsze o czymś albo zapomnimy, albo do projektu przyjdzie ktoś nowy i zwyczajnie nie będzie wiedział o wszystkich miejscach w kodzie, które należy zmienić.

Oczywiście ważne jest to, żeby zachować przy tym zdrowy rozsądek i nie próbować na siłę wyłączać części wspólnych. Dostaniemy wtedy przesuszony kod - **OVERDRIED** 😄. Zwyczajnie może się zdarzyć taka sytuacja, że kod stanie się czytelniejszy jeżeli dokonamy jakiegoś powtórzenia. We wszystkim trzeba znaleźć złoty środek.

Widziałem kiedyś w jednym projekcie 2 klasy realizujące jakąś skomplikowaną logikę, która była powielona w tych dwóch klasach. Rzecz polega na tym, że ich nazwy różniły się jednym znakiem, przy czym jedna z tych klas miała błąd w nazwie. Coś jak `Successful` i `Sucessful`. Zrozumienie co się dzieje w takim kodzie było bardzo ciężkie, bo raz przy pisaniu ktoś używał jednej klasy, a w innych miejscach w kodzie była używana ta druga.

> Co zrobić jak spotkamy coś takiego w praktyce? Najlepiej poświęcić czas i posprzątać, ale nie jest to niestety takie proste, bo najpierw musimy dobrze zrozumieć co robimy i najlepiej jest najpierw mieć napisane testy 😄.

KISS

KISS - *Keep It Simple, Stupid* - które w wolnym tłumaczeniu można zrozumieć jako "Zrób to prosto, idioto", link. Zasada ta mówi, że nasz kod ma być prosty. Jeżeli spojrzymy na tę zasadę na przykładzie metod - zamiast pisać rozległą dużą metodę, która obsługuje wiele przypadków, podzielmy ją na mniejsze. W większości przypadków dodanie kolejnej metody praktycznie nie wpłynie na performance, a zyskamy na tym czytelność kodu. Metody, które mają po setki lub nawet tysiące linijek są praktycznie niemożliwe do zrozumienia. Jednocześnie pisząc wszystko w jednej metodzie bardzo możliwe, że zaczniemy się gdzieś powtarzać zamiast reużywać fragment kodu wyciągnięty do innej metody i w ten sposób wracamy do **DRY**.

> Wyobraź sobie teraz metodę, która ma ponad 2000 linijek. Pomyśl teraz jak trudno jest zrozumieć co tak właściwie ta metoda robi.

YAGNI

YAGNI - *You aren't gonna need it* - zasada, która mówi o tym, że musimy bardzo ostrożnie podchodzić do tworzenia kodu, który w przyszłości nie będzie potrzebny. Inaczej mówiąc, musimy bardzo zwracać uwagę na pisanie kodu "na wszelki wypadek", jeżeli nie jest on potrzebny do działania programu w danym momencie. Po napisaniu takiego kodu może się okazać, że nasza ocena co do przydatności takiego kodu była błędna i napisany fragment kodu może wymagać całkowitej modyfikacji lub nawet usunięcia. W takim momencie firma, która zleca pisanie oprogramowania traci pieniądze, co jest zjawiskiem niepożądanym. Zasadę tą można rozumieć jako odkładanie pisania kodu na później, do momentu gdy napisanie go będzie absolutnie konieczne do poprawnej realizacji jakiejś funkcji.

Composition over Inheritance

Composition over Inheritance - Kompozycja ponad dziedziczenie. Wspomniałem o tym krótko podczas bootcampu, nie zagłębiałem się natomiast w temat, ze względu na to, że wydaje mi się, że na etapie realizacji naszego bootcampu było na to za wcześnie 😄. Niezależnie natomiast od tego, czy realizowałeś/realizowałaś z nami bootcamp, chciałbym teraz przejść do omówienia **composition over inheritance**.

Zasada ta mówi, że podczas tworzenia systemów informatycznych powinniśmy starać się używać kompozycji zamiast dziedziczenia. Można również spotkać stwierdzenia mówiące, że projektując klasy powinniśmy dążyć do tego, aby mogły one osiągać zachowania polimorficzne, a następnie powinniśmy takie klasy reużywać stosując kompozycję. Czyli klasa powinna mieć zdefiniowane pola w postaci interfejsów, a następnie inne klasy powinny te interfejsy implementować. Obiekt stworzony na podstawie definicji klasy, która implementuje taki interfejs, będzie natomiast odpowiedzialny za faktyczną implementację metody zdefiniowanej w takim interfejsie.

Dziedziczenie

Zakładam na tym etapie, że wiesz już czym jest dziedziczenie, więc nie będziemy w to wchodzić. O tym czym jest dziedziczenie rozmawialiśmy podczas bootcampu, poruszaliśmy temat słówek `extends`, `implements`. Wspominaliśmy również o relacji `is-a` oraz `has-a`. Przypomnijmy tylko, że dziedziczenie jest przykładem relacji `is-a`.

Chciałbym natomiast skupić się na zaletach i wadach dziedziczenia:

Zalety

- Zapewnia strukturę hierarchiczną, dzięki czemu możemy rozłożyć nasz model na "warstwy", starając się w ten sposób osiągnąć lepszą organizację klas,
- Pozwala na reużycie kodu z klasy rodzica, wracamy tutaj do **DRY**.

Wady

- Gdyby się zastanowić głębiej nad enkapsulacją to można dojść do wniosku, że dziedziczenie ją łamie, gdyż szczegóły implementacji klasy rodzica niejako wyciekają do klasy dzieci,
- Bardzo usztywnia strukturę, Java przykładowo może dziedziczyć tylko z jednej klasy, więc na tym polu jesteśmy zablokowani,
- Małe zmiany w klasie rodzica są przerzucane automatycznie do klas dzieci, co może niekoniecznie być nam potrzebne,
- Zmienne i metody z klasy rodzica mogą być widoczne w klasie dziecka, nawet jeżeli nie są w niej używane.

Kompozycja

Kompozycja natomiast jest relacją typu `has-a`. Stosując kompozycję mówimy, że jedna klasa zawiera w sobie instancje innych klas. Czyli stosując kompozycję mówimy, że samochód jest skomponowany z elementów takich jak np. koła, drzwi, kierownica itp. Poniżej omówmy sobie zalety i wady takiego podejścia.

Zalety

- Stajemy się mniej zależni od innych klas niż w przypadku dziedziczenia,
- Możemy zastosować pełną enkapsulację, gdyż nie wprowadzamy tutaj pojęcia rodzica, ani dziecka, więc dziecko nie widzi informacji z rodzica.

Wady

- Często wymaga to napisania większej ilości kodu niż w przypadku dziedziczenia,
- Zrozumienie działania kodu może być trudniejsze, ponieważ w przypadku kompozycji opieramy się na referencjach. Pod daną referencją może znajdować się wiele obiektów. Obiekty te są tworzone dopiero podczas działania programu, więc może być ciężej zrozumieć, który obiekt zostanie faktycznie wykorzystany.

Jeżeli realizowałeś/realizowałaś z nami bootcamp albo poprzednie warsztaty, przypomnij sobie projekt "Kalkulator Kredytu Hipotecznego". Operowaliśmy tam na serwisach, czyli na klasach, które realizują określony rodzaj funkcjonalności. Klasy te implementowały interfejsy, natomiast wzajemne powiązania między klasami były oparte właśnie o interfejsy.

Jeżeli natomiast nie realizowałeś/realizowałaś z nami bootcampu albo poprzednich warsztatów. Tworzyliśmy tam projekt, w którym wiązaliśmy ze sobą klasy przy wykorzystaniu kompozycji, stosowaliśmy do tego interfejsy. Tworzyliśmy serwisy, które były odpowiedzialne za realizację jakiejś funkcjonalności, np. obliczenie wysokości raty kredytu w danym miesiącu. W skrócie wyglądało to np. tak:

```
public class AmountsCalculationServiceImpl implements AmountsCalculationService {

    private final ConstantAmountsCalculationService constantAmountsCalculationService;

    private final DecreasingAmountsCalculationService decreasingAmountsCalculationService;

    public AmountsCalculationServiceImpl(
        final ConstantAmountsCalculationService constantAmountsCalculationService,
        final DecreasingAmountsCalculationService decreasingAmountsCalculationService
    ) {
        this.constantAmountsCalculationService = constantAmountsCalculationService;
        this.decreasingAmountsCalculationService = decreasingAmountsCalculationService;
    }

    // reszta kodu
}
```

Klasa `AmountsCalculationServiceImpl` implementuje interface `AmountsCalculationService`. Service `AmountsCalculationServiceImpl` nie dziedziczy z żadnego innego serwisu, tylko korzysta z funkcjonalności innych serwisów przy zastosowaniu kompozycji. Pola w serwisie `AmountsCalculationServiceImpl` są interfejsami (`ConstantAmountsCalculationService` i `DecreasingAmountsCalculationService` to interfejsy) i są użyte w formie kompozycji. W innym miejscu w kodzie definiujemy jakie faktycznie klasy mają implementować interfejsy `ConstantAmountsCalculationService` oraz `DecreasingAmountsCalculationService` i przekazujemy te implementacje do wywołania konstruktora `AmountsCalculationServiceImpl`.

W ten sposób stajemy się elastyczni, bo jak przestanie nam pasować sposób, w jaki zaimplementowaliśmy np. interface `DecreasingAmountsCalculationService`, to piszemy drugą implementację tego interfejsu i przekazujemy drugą implementację tego interfejsu do konstruktora `AmountsCalculationServiceImpl`. W samej klasie `AmountsCalculationServiceImpl` nie musimy nic zmieniać.

Jeżeli natomiast reużywalibyśmy kod przy wykorzystaniu dziedziczenia i chcielibyśmy coś zmienić w klasie bazowej to mamy problem. Może się okazać, że z naszej klasy rodzica korzysta już 5 innych klas i są one wszystkie ze sobą na sztywno powiązane. W przypadku wprowadzania zmian, dzieci mogą je dostać niezależnie od tego czy tego chcą czy nie.

Dlatego właśnie mówi się **composition over inheritance**.

Możesz również spotkać się ze stwierdzeniem **coupling** (używa się tego w kontekście słowa: *powiązanie*). Jeżeli obiekty są ze sobą połączone przy wykorzystaniu dziedziczenia, mówi się, że są **strongly coupled**. Jeżeli natomiast są połączone wykorzystując kompozycję to mówi się, że są **loosely**

coupled. Wynika to z tego, że jedna zmiana w klasie rodzicu zmienia wiele klas dziedziczących (dlatego **strongly**), w przypadku kompozycji takie zjawisko nie ma miejsca. Jeżeli chcesz doczytać o **composition over iheritance** to polecam ten link oraz ten link.

Nazewnictwo

Jeżeli na tym etapie rozumiesz już różnice między **Aggregation** a **Composition**, widzisz, że mówimy **composition over inheritance**, a w przykładach jest pokazana agregacja. Podejście **composition over inheritance** skupia się wokół tego jak korzystają z siebie wzajemnie klasy, a nie gdzie tworzone są implementacje interfejsów. Jeżeli natomiast dołożymy do tego **DIP**, który zostanie wyjaśniony w dalszej części, to wyjaśni się, czemu podane przykłady są napisane w ten sposób. Także doczytaj tę notatkę do końca ☺.

Filozofii ciąg dalszy

Oczywiście nie da się jednoznacznie powiedzieć, zawsze stosuj kompozycję, albo zawsze stosuj dziedziczenie. To jest trochę tak jakby ktoś przyszedł i powiedział, że zawsze preferuj jedzenie pierogów zamiast kebaba. A w życiu jest tak, że raz się jesz pierogi, a raz kebaba. Tak samo w tym przypadku. Są takie sytuacje, że ewidentnie ma sens zastosowanie dziedziczenia, bo mamy przykład `Animal`, `Cat`, `Dog`. Nie będziemy przecież wtedy pisać, że `Cat` zawiera `Animal`. Nie wciskamy na siłę kompozycji tylko dlatego, że ta zasada mówi, żeby ją preferować. Jeżeli potrzebujemy zastosować dziedziczenie to je stosujmy.

W praktyce natomiast, w większości przypadków stosuje się kompozycję.

Jeżeli dobrze rozumiemy oba mechanizmy, rozumiemy też, że dziedziczenie niesie za sobą pewne konsekwencje. Należy dwa razy zastanowić się jak ten kod może nam się w przyszłości rozjechać, jeżeli planujemy użyć dziedziczenia.

SOLID

Następnie chciałbym przejść do omówienia zasad **SOLID**. Są to zasady, których pierwsze litery tworzą skrót **SOLID**, stąd ich grupa jest nazywana w ten sposób.

Zasady **SOLID** często są poruszane na rozmowach rekrutacyjnych. Tak tylko wspominamy o tym ☺.

Obraz 2. SOLID principles

Wszystkie z reguł **SOLID** określają wskazówki, które mają nam ułatwić zarządzanie systemami

informatycznymi z punktu widzenia napisanego kodu. To, że oprogramowanie żyje jest nieuniknione. Możemy natomiast napisać je w taki sposób, żeby zarządzanie takimi zmianami było ułatwione. Czyli w sumie to pomagamy sami sobie.

Single Responsibility Principle

Single Responsibility Principle - **SRP** - klasa powinna mieć jedną, ale to tylko jedną odpowiedzialność. W internecie znajdziemy takie określenie jak:

> A class should have only one reason to change.

Cytat ten jest wzięty z książki: *Martin, Robert C. (2003). Agile Software Development, Principles, Patterns, and Practices.*

Reason to change można rozumieć jako funkcjonalność, za jaką jest odpowiedzialna dana klasa. Jeżeli jest odpowiedzialna za więcej niż jedną rzecz, istnieje więcej niż jeden powód, żeby tę klasę zmieniać.

Trzymanie się tej zasady wprowadza swojego rodzaju modularność. Klasy zaczynają być odpowiedzialne tylko za jedną konkretną rzecz (np. komunikację z bazą danych, obliczenie wysokości pozostałej raty kredytu itp.).

Oczywiście możemy pisać swój kod w taki sposób, aby jedna klasa była odpowiedzialna za wiele rzeczy jednocześnie. Spowoduje to natomiast bałagan w zależnościach naszej klasy. Jeżeli klasa robi więcej niż jedną rzecz i jednocześnie jest zależna od innych klas, tworzymy w takim przypadku dodatkowe relacje między klasami, które nie byłyby nam potrzebne jeżeli trzymamy się **SRP**.

Wyobraź sobie klasę, która jest odpowiedzialna jednocześnie za kupienie ziemniaków, zrobienie obiadu, nakarmienie dzieci i pójście na siłownię. Każda z czynności wymaga dodania zależności do innych klas, czyli musimy mieć interfejs, który pomoże nam zrobić obiad, inny interfejs, który pomoże nakarmić dzieci, kolejny, który obsłuży przypadek kupienia ziemniaków i następny, który pójdzie za nas na siłownię. (Jakby co to mówię, że w życiu tak nie ma 😅). Ilość zależności w takiej klasie automatycznie puchnie. Jeżeli rozdzielimy taką klasę zgodnie z **SRP**, zmniejszymy ilość zależności i jednocześnie uprościmy kod.

Open/Closed Principle

Open/Closed - **OCP** - Zasada **Open/Closed** mówi, że klasy lub obiekty oraz metody powinny być otwarte na rozszerzanie, ale zamknięte na modyfikacje.

Oznacza to, że powinniśmy pisać nasze klasy uwzględniając to, że w przyszłości taka klasa może wymagać aktualizacji. W praktyce przekłada się to na to, że możemy w takiej klasie dodawać więcej pól lub metod, natomiast nie możemy usuwać starych pól i starych metod ani modyfikować ich w żaden sposób, po to aby kod po rozszerzeniu funkcjonalności nadal działał. Uwzględnianie przypadków, które mogą pojawić się w przyszłości pomaga lepiej zaprojektować klasę, którą akurat piszemy.

Stosując tę zasadę zapobiegamy powstawaniu błędów regresji. Są to błędy, które wprowadzamy w momencie gdy musimy zmodyfikować funkcjonalność, z której korzystają już klienci na produkcji.

Pomocne może też być ustalenie, które funkcjonalności mogą w przyszłości ulegać zmianie w stosunku

do tych, co do których jesteśmy pewni, że nie będą one ulegać zmianom. Zauważysz natomiast, że w wielu z omawianych kwestii, często należy kierować się "zdrowym rozsądkiem".

W praktyce ta zasada dosyć jest często naruszana ze względu na np. dynamicznie zmieniające się wymagania biznesowe. W takim przypadku należy przeprowadzić testy regresji - czyli ponownie sprawdzić działające już funkcjonalności, aby upewnić się, że funkcjonalność, która została zmieniona nie została przez nas popsuta w wyniku wprowadzania modyfikacji.

Liskov Substitution Principle

W ramach ciekawostki najpierw powiem, że **Liskov** pochodzi od nazwiska Barbary Liskov. Zauważyłem, że dużo osób myśli, że zasada ta pochodzi od męskiego nazwiska, więc wyjaśniam, link 😄.

Liskov Substitution Principle - **LSP**- zasada mówi, że jeżeli operujemy na klasach bazowych i klasach pochodnych, (czyli np. w ramach dziedziczenia mamy superklasę i subklasę, albo inaczej nazywając mamy rodzica i dziecko), to klasy pochodne powinny być w stanie zastępować swoje klasy bazowe bez zmiany zachowania kodu. Zaraz wyjaśnimy to sobie w kodzie.

Zasada ta jest jednocześnie powiązana z omówionymi wcześniej **SRP** principle i **Open/Closed** principle. Prawdopodobne naruszenie poprzednich zasad spowoduje również naruszenie tej zasady. Wynika to z tego, że jeżeli klasa robi więcej niż jedną rzecz i zaczniemy z tej klasy dziedziczyć to podklasy dziedziczące mają małe szanse aby sensownie zaimplementować dziedziczone zachowania, bo jest ich więcej niż jedno.

O stosowaniu dziedziczenia pomiędzy obiektami często myślimy w ten sposób, że jeżeli obiekty spełniają relację `is-a`, to znaczy, że możemy dziedziczyć. Jednakże stwierdzenie to jest często dosyć mylące. Naturalne dla nas wydaje się, że `Cat` is-a `Animal`, `Car` is-a `Vehicle`, `Chair` is-a `Furniture`. Potrafimy też rozróżnić, że nie działa to w odwrotną stronę, czyli `Animal` is not a `Cat`, `Car` is not a `Vehicle`, dlatego, że obok tych relacji możemy mieć w kodzie zapisane, że `Monkey` is-a `Animal` lub `Spider` is-a `Animal`. Jasne wtedy staje się, że `Animal` is not a `Cat`. Są jednak takie przykłady, gdzie to, co wydaje się nam naturalne, nie jest zgodne z tą zasadą (**LSP**).

Klasyczny przykład, który bardzo często można znaleźć w Internecie to "kwadrat jest prostokątem". Pamiętamy z matematyki ze szkoły, że nie każdy prostokąt jest kwadratem, natomiast każdy kwadrat jest prostokątem. Czyli można pomyśleć, że mamy tutaj relację `is-a`. Napiszmy sobie przykładowy kod:

Klasa Rectangle

```
@Getter
@Setter
@AllArgsConstructor
public class Rectangle {
    protected double a;
    protected double b;

    public double area() {
        return a * b;
    }
}
```

Klasa Square

```
public class Square extends Rectangle {

    public Square(double a) {
        super(a, a);
    }

    @Override
    public void setA(double a) {
        this.a = a;
        this.b = a;
    }

    @Override
    public void setB(double b) {
        this.a = b;
        this.b = b;
    }
}
```

W takim przypadku, `Square` rozszerza klasę `Rectangle` i słusznie nadpisuje metody `setA()` oraz `setB()`. Stwórzmy poniżej instancję klasy `Square` i przypiszmy ją do zmiennej typu `Rectangle`.

```
Rectangle rectangle = new Square(10);
```

Załóżmy dalej, że ktoś inny chce użyć zmiennej `rectangle`, aby zmienić długości boków tego prostokąta. W końcu kwadrat jest prostokątem 😄. Napiszmy zatem taki kod:

```
rectangle.setA(4);
rectangle.setB(2);
System.out.println(rectangle.area());
```

Skoro operujemy na prostokącie, to spodziewamy się, że otrzymamy wartość pola `8`. Natomiast wartość tego pola będzie wynosiła `4`. Zachowanie jest niespodziewane, gdyż operujemy na prostokącie, jednocześnie przykład jest na tyle prosty, że możemy dość szybko znaleźć przyczynę.

Można powiedzieć, że teoretycznie ma to działać dokładnie w ten sposób, bo przecież mamy w tym przypadku kwadrat, a nie prostokąt.

Kiedy dziedziczymy musimy pamiętać, że klasy pochodne powinny być w stanie zastępować swoje klasy bazowe bez zmiany zachowania kodu. Dziedzicząc musimy zatem pamiętać o tym jak nasze klasy się zachowują i czy rzeczywiście możemy używać ich wymiennie. W przypadku `Square` i `Rectangle` tak nie jest.

Interface Segregation Principle

Interface Segregation Principle - **ISP** - zasada segregacji interfejsów. Zasada ta mówi, że klasa implementująca interface nie powinna być zmuszona do polegania na interfejsie, którego nie używa w całości. Czyli interface powinien zawierać tylko minimalny zestaw metod, które są niezbędne żeby zapewnić funkcjonalność. Jednocześnie zestaw tych metod powinien być ograniczony tylko do tej jednej

funkcjonalności. Trochę analogicznie do **SRP**.

Jeżeli spojrzymy teraz na przykład owieczek i ryb. Stwórzmy interface `Animal`, do tego oczywiście stworzymy klasy `Sheep` i `Fish`. Interface `Animal` nie powinien definiować metody `swim()`, bo nie każde zwierze musi umieć pływać (mam nadzieję, że owca nie umie bo nam się przykład rozsypie 😄). Dlatego też w interface `Animal` nie definiujemy na siłę metody `swim()`, bo jeżeli klasa `Sheep` zaimplementuje ten interface to po co jej metoda `swim()`. Odwrotnie działa to tak samo, nie da się ostrzyc ryby. Możemy natomiast założyć, że każde zwierze musi coś jeść i jakoś się przemieszczać. Albo chociaż jakoś na boki poruszać.

Interface Animal

```
public interface Animal {
    void eat();
    void move();
}
```

Klasa Sheep

```
public class Sheep implements Animal {
    @Override
    public void eat() {
        System.out.println("Sheep eating!");
    }

    @Override
    public void move() {
        System.out.println("Sheep moving!");
    }

    public void getAHaircut() {
        System.out.println("I'm loosing my hair!");
    }
}
```

Klasa Fish

```
public class Fish implements Animal {
    @Override
    public void eat() {
        System.out.println("Fish eating!");
    }

    @Override
    public void move() {
        System.out.println("Fish moving!");
    }

    public void swim() {
        System.out.println("I'm swimming!");
    }
}
```

Jeżeli którąkolwiek z metod `getAHaircut()` lub `swim()` umieścilibyśmy w interface `Animal`, wtedy każda klasa implementująca ten interface musiałaby zaimplementować tę metodę. Dlatego właśnie w

interfejsach powinny być definiowane tylko niezbędne funkcjonalności.

Dependency Inversion Principle

Dependency Inversion Principle - **DIP** - zacznijmy od cytatu link:

> High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces).
>
> Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

Oznacza to, że moduły wysokiego poziomu (zaraz napiszę jak jest to rozumiane) i moduły niskiego poziomu powinny być ze sobą łączone w taki sposób, aby wprowadzanie zmian w modułach niskopoziomowych nie wymagało wprowadzania zmian w modułach wysokopoziomowych. Innymi słowami, zarówno moduły wysoko jak i niskopoziomowe nie powinny zależeć bezpośrednio od siebie, ale powinny zależeć od abstrakcji, jaką są interfejsy.

Moduły wysokiego poziomu są rozumiane jako klasy, które aranżują interakcję modułów niskiego poziomu. Moduły niskiego poziomu natomiast są np. konkretnymi klasami wykonującymi konkretne zadania, np. oblicz wynik dodawania 2 liczb.

Spróbujmy to ugryźć innymi słowami. **DIP** stara się przekazać, że uzależnianie się w klasach od interfejsów jest bezpieczniejsze niż zależność od konkretnych klas. Przekłada się to na mniejsze usztywnienie kodu, na mniejszą sztywną zależność między klasami. Ułatwia to następnie zarządzanie takim kodem, jego modyfikacje oraz późniejsze testowanie.

Jak ułatwia to testowanie? Wyobraź sobie, że klasa `A` jest zależna od klasy `B`. Chcemy przetestować klasę `A` i jednocześnie wymusić konkretne zachowanie na klasie `B` (w praktyce taka sytuacja określana jest **mockowaniem**, ale jeszcze nie będziemy o tym mówić).

```
class A {
    private B b;

    // jakiś kod
}
```

Jeżeli klasa `A` jest bezpośrednio zależna od klasy `B` to nie mamy możliwości wymienić takiego zachowania w sprytny sposób (przynajmniej na tym etapie nauki). Jeżeli natomiast klasa `A` będzie zależna od interfejsu `C` i ten interface `C` będzie implementowany przez klasę `B`, to możemy w teście określić, że chcemy żeby na potrzeby testu interface `C` był implementowany przez klasę `TestB`, która zachowa się dokładnie tak jak chcemy. Analogiczny przykład został pokazany w warsztacie o testach jednostkowych.

```
class A {
    private C c;

    // jakiś kod
```

```
}
```

```
interface C {
    // jakiś kod
}
```

```
class B implements C {
    // normalna implementacja
}
```

```
class TestB implements C {
    // implementacja na potrzeby testów
}
```

Dzięki temu uzyskujemy większą elastyczność, bo w uniezależniamy klasy od siebie. Jeżeli teraz z innego powodu niż testy potrzebujemy przekazać inną implementację klasy `B`, która ma być wykorzystana w klasie `A`, to nie musimy wcale modyfikować kodu w klasie `A`, wystarczy, że wymienimy implementację interfejsu `C`, która ma być wykorzystana w klasie `A`.

Jeżeli brałeś/brałaś udział w Bootcampie albo warsztatach to przypomnij sobie, że dokładnie w ten sposób był pisany projekt "Kalkulator Kredytu Hipotecznego".

Inversion of Control

Wiem, że i tak już jest ciężko z nowymi terminami, ale żeby ta notatka była kompletna, musimy wyjaśnić terminy, które i tak spotkasz na swojej drodze i które są często używane wymiennie. A jak już wiesz, jak coś jest używane wymiennie to się myli.

Inversion of Control - **IoC** - jest traktowany jako **Design Pattern** (wzorzec projektowy) (o wzorcach projektowych będzie w następnej części warsztatu, ale ten temat bardziej pasuje tutaj). Często również można spotkać w Internecie stwierdzenie, że jest to **Design Principle**, ważne natomiast żebyśmy wiedzieli o co w tym chodzi.

Wzorzec ten zaleca odwrócenie jakiegokolwiek rodzaju kontroli podczas projektowania programu zorientowanego na obiekty po to żebyśmy osiągnęli luźne powiązania pomiędzy klasami. Kontrola odnosi się w tym przypadku do jakiegokolwiek rodzaju odpowiedzialności jakie ma dana klasa oprócz jej głównej odpowiedzialności (która jest najczęściej wyrażona w metodach). Odpowiedzialnością inną niż główna może być np. tworzenie obiektów zależnych klas lub łączenie ze sobą utworzonych obiektów zależnych klas.

W **IoC** chodzi o odwrócenie takiej kontroli. Fragment kodu zostanie umieszczony w dalszej części. Teraz natomiast zastanówmy się jak mógłby wyglądać taki przykład w życiu codziennym.

Załóżmy, że normalnie robimy sobie sami jedzenie, a później je jemy. W przypadku **IoC**, ktoś robił by jedzenie przynosił nam, a my byśmy je później jedli. Dzięki temu możemy się skupić tylko na jedzeniu (czyli wykonaniu jednej czynności, a nie dwóch).

Inny przykład to noszenie kurtki, najpierw wybieramy kurtkę a potem ją nosimy. W przypadku **IoC** ktoś wybrałby za nas kurtkę, a my byśmy ją tylko nosili, czyli moglibyśmy się skupić na jednym zadaniu jakim jest noszenie kurtki.

Podejście **IoC** pomaga osiągnąć stan luźnego powiązania między klasami (**loose coupling**), co ułatwia późniejsze zarządzanie kodem.

Przykłady

Poniżej chciałbym omówić przykład, który pomoże zrozumieć istotę **IoC**.

Wyobraź sobie, że piszesz aplikację konsolową, która przyjmuje od użytkownika jakieś parametry i następnie wykonuje na ich podstawie jakieś operacje. Przykład takiej aplikacji może być realizowany przez poniższy kod:

```
public class ConsoleReadingApp {

    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Enter your username: ");
        if (console.hasNext()) {
            String read = console.nextLine();
            System.out.println("username: " + read);
        }

        System.out.print("Enter your password: ");
        if (console.hasNext()) {
            String read = console.nextLine();
            System.out.println("password: " + read);
        }

        System.out.print("THE END");
    }
}
```

W tym przypadku to metoda `main()` wymusza przepływ programu i to ona wymusza sekwencję interakcji użytkownika. Przykładem **IoC** w takiej sytuacji byłaby taka sama aplikacja, tylko że napisana w systemie **Android**, gdzie mielibyśmy formularz do wpisania **username** i **password**. W takim przypadku to system **Android** kontrolowałby przepływ programu, a nie my bezpośrednio. Innymi słowy to **Android** wywoływałby nasze metody, w których określalibyśmy co ma się stać w momencie gdy użytkownik wpisze wartość `username`, albo gdy wciśnie guzik na ekranie. To nie my byśmy to kontrolowali, bo przekazaliśmy tę kontrolę systemowi **Android**.

Dependency Injection

Dependency Injection - **DI** - o **DI** mówi się, że jest to **Design Pattern** (wzorzec projektowy), który jest implementacją **IoC**. **DI** jest techniką, w której obiekt otrzymuje od innego obiektu swoje gotowe przygotowane wcześniej zależności. Przypomnij sobie, że wcześniej jak omawialiśmy **IoC**, albo **DIP** nigdzie nie powiedzieliśmy gdzie takie obiekty są tworzone. Mówiliśmy jedynie o sposobach powiązania zależności i sposobie definiowania przepływu w aplikacji. Wyobraźmy sobie teraz taki przykład.

```
class A {
    private C c;

    public A() {
        this.c = new B();
    }
}
```

```
interface C {
    // jakiś kod
}
```

```
class B implements C {
    // normalna implementacja
}
```

W przykładzie powyżej zależymy od abstrakcji (czyli od interfejsu C - **DIP**), ale przepływem sterujemy my, bo w konstruktorze jawnie tworzymy implementację obiektu B. W zasadzie to nas to usztywnia, jeżeli będziemy chcieli zmienić implementację interfejsu C, to musimy modyfikować kod w klasie A. Jak można napisać to inaczej?

```
class A {
    private C c;

    public A(final C c) {
        this.c = c;
    }
}
```

```
interface C {
    // jakiś kod
}
```

```
class B implements C {
    // normalna implementacja
}
```

```
class Factory {
    void create() {
        C c = new B();
        A a = new A(c);
    }
}
```

Sytuacja taka określana jest **injection** (wstrzyknięcie) ze względu na to, że do klasy A wstrzykujemy obiekt B, który został utworzony w innym miejscu. W przykładzie powyżej usuwamy tworzenie zależności z naszej klasy i przenosimy ten kod do klasy, która jest odpowiedzialna za powiązanie takich

zależności ze sobą.

Wiem, że na tym etapie to wszystko wydaje się skomplikowane i zagmatwane. Uwierz mi, że z czasem to zacznie mieć dużo sensu, szczególnie jak zaczniemy poznawać frameworki, np. **Spring**.

Znowu odwołam się do Bootcampu albo warsztatów dla tych, którzy brali udział. Ponownie do projektu "Kalkulator Kredytu Hipotecznego". Pamiętasz już, że klasy zależały tam od interfejsów. Przypomnij sobie teraz, w którym miejscu następowało połączenie wszystkich klas ze sobą. Konfiguracja ta była zrobiona w metodzie `main()`. Czyli cała odpowiedzialność łączenia obiektów ze sobą była wyniesiona poza klasy, po to aby mogły się one skupić na tym co mają robić. Sama konfiguracja mogła również zostać wyniesiona poza metodę `main()`, natomiast nie chciałem więcej komplikować w tamtym momencie.

IoC Container

Czytając o **IoC** natkniemy się na stwierdzenie **IoC Container**. **IoC Container** oznacza framework (nie konkretny framework tylko rodzaj frameworka), który implementuje automatyczne **DI**. Framework taki zarządza wtedy tworzeniem obiektów oraz wstrzykiwaniem zależności między klasami. Jest to o tyle istotne, że my będziemy używać takiego podejścia jak przejdziemy do omówienia frameworka **Spring**.

Podsumowując DIP, IoC oraz DI

Ochłoń w tym momencie trochę, bo na pewno terminy te zdążyły się pomieszać. A jeżeli uważasz, że nie, to spróbuj nie patrząc w ekran wytłumaczyć je sobie na głos. I jak poszło 😀? Zapamiętanie tego wymaga czasu i jest to kompletnie naturalna sprawa, więc spokojnie, wszystko z czasem stanie się jasne.

Chciałbym natomiast podsumować prostymi słowami, że terminy te tak na prawdę nie są jednym i tym samym, ale są często używane w ten sposób (jak jedno i to samo), ze względu na to jak blisko są ze sobą związane.

- **DIP** mówiło nam o tym, że mamy zależeć od abstrakcji (interfejsów), a nie od konkretnych implementacji klas. Nie mówimy tutaj o tym, kto ma tym zarządzać.
- **IoC** mówiło nam o tym, że odwracamy kontrolę i to nie my kontrolujemy przebieg działania aplikacji, tylko np. framework **Android** albo **Spring**.
- **DI** mówiło nam o tym, że nasz obiekt otrzymał gotowy obiekt, który został mu wstrzyknięty. Jest to rodzaj odwrócenia kontroli, bo nie nasz obiekt zdecyduje o tym jaki gotowy obiekt ma być wykorzystany, tylko wykorzystywane jest przekazanie gotowca z zewnątrz.
- **IoC Container** określało framework, który za nas tworzył obiekty, które zostały wstrzyknięte, przykładowo dzięki **IoC** nie musimy tworzyć klasy, która powiąże ze sobą obiekty w projekcie.

Podsumowanie

Pamiętaj, że pokazane tutaj zasady to są wskazówki, a nie prawa fizyki jak pierwsze prawo dynamiki Newtona. 😀 Mają nam one pomóc uporządkować nasz kod, a nie zmusić nas do stosowania tych zasad kiedy tylko się da. Piszę o tym, bo należy pamiętać, że to są wypracowane przez lata wskazówki, ale

musimy stosować je z głową.

Zdaję sobie sprawę, że wiele z tych kwestii może się dosyć mocno pomieszać. Jest to często wiedza typowo pamięciowa, ale jest dosyć istotna, bo służy do tego, żeby ułatwiać nam życie w pracy. Zobaczysz też, że jak będziesz nabierać doświadczenia to wszystkie przedstawione tutaj kwestie będą stawały się coraz bardziej intuicyjne.