Programowanie obiektowe
Projektowanie obiektowe

Spis treści

1 Pojęcia

1.1 Abstrakcja

W programowaniu obiektowym za abstrakcję przyjmuje się opis oraz ograniczenie cech obiektu ze świata rzeczywistego do cech istotnych, kluczowych z punktu widzenia programisty. Tzn. spłycanie mniej lub bardziej czegoś abstrakcyjnego/konkretnego do pewnego opisu, tym opisem jest klasa. Tłumacząc na kod cechą obiektu będzie pole klasy, a zachowaniem metoda.

1.2 Agregacja

Agregacja (inaczej zawieranie się, gromadzenie) – sytuacja, w której tworzy się nową klasę, używając klas już istniejących (często nazywa się to "tworzeniem obiektu składowego"). Nowa klasa może być zbudowana z dowolnej liczby obiektów (obiekty te mogą być dowolnych typów) i w dowolnej kombinacji, by uzyskać żądany efekt. Przykładem agregacji jest klasa Lampa oraz klasa Żarówka. Lampa może zawierać żarówki, ale osobno obiekty tych klas również mogą istnieć.

1.3 Aktor

Aktor to zewnętrzny obiekt modelujący określoną rolę zewnętrznego użytkownika systemu. Aktor może operować na innych obiektach, tzn. komunikować się z nimi za pomocą interfejsu , ale sam nie podlega operacjom ze strony innych obiektów.

1.4 Analiza czasownikowo-rzeczownikowa

Analiza czasownikowo-rzeczownikowa jest etapem tworzenia oprogramowania polegająca na zbieraniu wymagań od klienta lub burzy mózgów co do wymagań wobec fragmentu programu, najczęściej klasy. Opis klasy jest dzielony na części, gdzie czasownikami są metody, a rzeczownikami są dane (składowe klasy).

1.5 Antywzorzec śrutówka

Antywzorzec śrutówka (ang. shotgun anti-pattern), to wzorzec opisujący rodzaj niskiej spójności. Śrutówka to broń, która wystrzeliwuje mnóstwo małego śrutu i ma zwykle duży rozrzut. W rozwoju oprogramowania ta metafora oznacza, że określony aspekt dziedziny lub jeden pomysł jest wysoce rozdrobniony i rodzielony między wiele modułów.

1.6 Delegacja

Delegacja to określenie sytuacji w obiektowej strukturze danych, gdy operacje, które można wykonać na danym obiekcie, są własnością innego obiektu (są "oddelegowane" do innego obiektu). W nieco innym znaczeniu (UML) delegacją jest nazywana sytuacja, kiedy obiekt po otrzymaniu komunikatu wysyła komunikat do innego obiektu. Często określenie "delegacja" dotyczy sytuacji, kiedy jakiś złożony atrybut obiektu jest także obiektem i jest wystąpieniem innej klasy. Delegacja jest uważana także za alternatywę lub szczególny przypadek dziedziczenia.
Przykładem może być klasa Samochód, która nie może dziedziczyć po częściach samochodowych takich jak Silnik czy Karoseria ze względu na dużą liczbę klas, po których musiałaby dziedziczyć, a może tylko po jednej. Wyjściem z tego problem u jest zawarcie w sobie klasy Silnik, Kierownica itd., a następnie odwołuje się do nich (deleguje) przez metody tych klas, np. engine.run(). Innym przykładem może klasa Lampa, która zwiera obiekt żarówka typu Żarówka. Żeby włączyć żarówkę używamy metody switchOn() na obiekcie lampy, a on deleguje nas do metody switchOn() żarówki.

1.7 Dziedziczenie

Dziedziczenie (czyli związek "jest") wyraża związek pomiędzy klasą ogólną i klasą specjalną. Klasa ogólna to klasa bazowa, a specjalna to pochodna. I jest to mechanizm programowania obiektowego, który pozwala

tworzyć klasy na podstawie innej klasy, wtedy taka klasa przejmuje jej pola i metody oraz rozszerza o własne pola i metody. Słowem kluczowym dziedziczenia jest *extends*

1.8 Garbage collector

Garbage Collector (odśmiecacz pamięci) mechanizm niektórych języków programowania, który przejmuje odpowiedzialność za zarządzanie pamięcią w programie od programisty. Odpowiada za zwalnianie dynamicznej przydzielonej pamięci w momencie kiedy dane nie są już potrzebne. GC JVM działa w oparciu o różne algorytmy, ale w głównej mierze jest oparty o widoczność obiektów na różnych pokoleniach sterty. GC znakuje obiekty, które nie są w użyciu, a następnie je usuwa. Więcej do przeczytania tutaj i tutaj .

1.9 G.R.A.S.P. General Responsibility Assignment Sofware Principles

G.R.A.S.P. to zbiór 9 zasad określających, jaką odpowiedzialność powinno się przypisywać określonym obiektom i klasom w systemie. Każda zasada wyjaśnia inny problem dotyczący ustalania odpowiedzialności obiektu.

1.9.1 Creator

Problem: kiedy obiekt powinien tworzyć inny obiekt?

Rozwiązanie: przypisz klasie B odpowiedzialność tworzenia obiektów klasy A, gdy:

- klasa B zawiera (lub agreguje) obiekty klasy A
- B zapamiętuje A
- B „blisko" współpracuje z A
- B ma dane inicajlizujące potrzebne przy tworzeniu A

Opis: tworzenie obiektów jest dość częstym zajęciem programów obiektowych. Odpowiednie przypisanie odpowiedzialności w tym zakresie powoduje, że projekt obiektowy jest lepszy, ponieważ zmniejsza się ilość powiązań między obiektami i niepotrzebnych wywołań. Ta zasada jest też przydatna, gdy jest paru kandydatów do wyboru – wtedy na podstawie ilości spełnianych warunków można wytypować najlepszego.

1.9.2 Information Expert

Problem: jak ustalać zakres odpowiedzialności obiektu?

Rozwiązanie: programista powinien delegować nową odpowiedzialność do klasy zawierającej najwięcej informacji potrzebnych do zrealizowania nowej funkcjonalności.

Opis: jest to najogólniejsza z zasad, brzmi wręcz banalnie. Tak naprawdę nie powinna być lekceważona, ponieważ w małym systemie może być ponad setka klas i podczas projektowania trzeba rozdzielić zdania pomiędzy nie. Pochopne podejmowanie decyzji może spowodować, że klasa aby wykonać zadanie będzie musiała przedostać się przez spory graf obiektów, aby uzyskać odpowiednie informacje.

1.9.3 Controller

Problem: jaka część systemu odpowiedzialna za logikę biznesową programu powinna komunikować się z interfejsem użytkownika?

Rozwiązanie: przypisz tą odpowiedzialność klasie, która odpowiada jednemu z poniższych opisów:

- klasa opisuje system jako całość, główny obiekt, urządzenie na którym system jest uruchomiony, większy podsystem lub „facade controller"
- klasa reprezentująca przypadek użycia systemu, w którym występuje dana operacja

Opis: na początek wyjaśnienie słownictwa: operacja systemu odpowiada funkcji systemu (np. logowanie, licytowanie przedmiotu), a kontroler to obiekt, który nie należy do UI, a jest odpowiedzialny za obsługę operacji. Zasada ta porusza bardzo szeroki temat oddzielenia interfejsu użytkownika od logiki aplikacji, w szczególności implikuje takie wzorce jak MVC czy MVP. Należy zauważyć, że na powyższej liście nie ma „okien", „widoków", czy „dokumentów", te obiekty powinny po przechwyceniu zdarzenia (zazwyczaj wygenerowanego przez użytkownika) przenieść sterowanie do klasy kontrolera.

1.9.4 Low Coupling (niska liczba powiązań)

Problem: jak ograniczyć zakres zmian w systemie w momencie zmiany fragmentu systemu?

Rozwiązanie: deleguj odpowiedzialności tak, aby zachować jak najmniejsza liczbę powiązań pomiędzy klasami.

Opis: klasa, która ma dużo powiązań, jest zależna od wielu innych klas, co powoduje:

- wiele lokalnych zmian spowodowanych zmianami w klasach powiązanych
- klasy trudniejsze do zrozumienia w izolacji
- zmniejszony „reuse" (możliwość ponownego wykorzystania), ponieważ najczęściej trzeba jednocześnie wykorzystać część klas powiązanych

1.9.5 High Cohesion (wysoka spójność/zwartość)

Problem: jak ograniczać zakres odpowiedzialności obiektu?

Rozwiązanie:przypisuj odpowiedzialności do obiektu tak, aby spójność była jak największa.

Opis: każdy obiekt powinien skupiać się tylko na jednej odpowiedzialności. Nie należy tworzyć obiektów odpowiedzialnych za dużą cześć logiki aplikacji. W przypadku dużej klasy należy podzielić odpowiedzialność na kilka mniejszych klas.

1.9.6 Polymorphism

Problem: co zrobić, gdy odpowiedzialność różni się w zależności od typu?

Rozwiązanie: przydziel zobowiązania, przy użyciu polimorfizmu, typom dla których to zachowanie jest różne.

Opis: zbiór obiektów może mieć zbiór zachowań wspólnych oraz zbiór zachowań odmiennych. Dla zachowań odmiennych powinniśmy wykorzystać mechanizm polimorfizmu. Podobny efekt można uzyskać korzystając z konstrukcji if-else jednak nie jest to dobra praktyka, ponieważ w przypadku zmian trzeba odszukać i zaktualizować wszystkie miejsca w programie zawierające te konstrukcje. Prowadzi to do powstania kodu trudnego i drogiego w utrzymaniu.

1.9.7 Protected variations

Problem: jak projektować obiekty, by ich zmiana nie wywierała szkodliwego wpływu na inne elementy?

Rozwiązanie: zidentyfikuj punkty przewidywanego zróżnicowania czy niestabilności i przypisz odpowiedzialności do wspólnego stabilnego interfejsu.

Opis: należy tak zaprojektować system, aby jak najprostsze było wymienianie elementów systemu na alternatywne. Podczas zmian powinna zostać dostarczona inna implementacja interfejsu, dlatego interfejsy powinny być tworzone dla punktów niestabilności, czyli miejsc w programie, które mogą wymagać różnych zachowań.

1.9.8 Indirection

Problem: komu przydzielić zobowiązanie, jeśli zależy nam na uniknięciu bezpośredniego powiązania między obiektami?

Rozwiązanie: przypisz te odpowiedzialności do nowego pośredniego obiektu. Obiekt ten będzie służył do komunikacji innych klas/komponentów/usług/pakietów tak, że nie będą one zależne bezpośrednio od siebie.

Opis: obiekty powinny być jak najmniej ze sobą powiązane, aby zmniejszyć ilość powiązań można stworzyć klasę pośrednią, która będzie się komunikować z pozostałymi obiektami. Klasa pośrednia może odpowiadać paru innym klasom przez co jedna klasa nie musi mieć żadnych informacji na temat innych klas. Przykładem takiego zachowania może być odczyt danych z pliku lub z bazy danych, klasa która potrzebuje te informacje nie musi wiedzieć skąd je wziąć oraz czy pochodzą z bazy danych czy z pliku. Takimi rzeczami zajmuje się klasa pośrednia.

1.9.9 Protected Variations

Problem: jak projektować obiekty, by ich zmiana nie wywierała szkodliwego wpływu na inne elementy?

Rozwiązanie: zidentyfikuj punkty przewidywanego zróżnicowania czy niestabilności i przypisz odpowiedzialności do wspólnego stabilnego interfejsu.

Opis: należy tak zaprojektować system, aby jak najprostsze było wymienianie elementów systemu na alternatywne. Podczas zmian powinna zostać dostarczona inna implementacja interfejsu, dlatego interfejsy powinny być tworzone dla punktów niestabilności, czyli miejsc w programie, które mogą wymagać różnych zachowań.

1.10 Hermetyzacja

Hermetyzacja (inaczej enkapsulacja) - jeden z wielu mechanizmów programowania obiektowego polegający na ukrywaniu danych lub metod klasy tak, aby były dostępne one z zewnątrz tylko przez ustalony interfejs, który często posiada mechanizm sprawdzania poprawności wprowadzonych danych. Z tą zasadą wszystkie pola i niektóre metody powinny być dostępne wyłącznie dla klasy, która je zawiera. Używanie słowa kluczowego *protected* jest łamaniem tej reguły, dlatego z zasady nie powinno się go używać na polach klasy. Jedynym z wyjątków, w którym można użyć protected jest utworzenie metody, która będzie mogła mieć odmienne zachowanie w klasach pochodnych, a sama ma być niewidoczna z zewnątrz.

1.11 Interfejs

Pierwszym znaczeniem interfejsu (zwanym też API (od ang. application programming interface) jest przedstawienie reguł jakie zachodzą pomiędzy programami lub ich modułami. Inaczej, jest to zbiór klas, metod i podprogramów zdefiniowanych na poziomie kodu źródłowego i możliwego do wykorzystania przez innego programistę. Przykładem może być Java Swing (biblioteka graficzna Javy), która dostarcza interfejs pozwalający pisać aplikacje okienkowe.
Drugim znaczeniem jest interfejs klasy, jest to abstrakcyjny typ danych mówiący co klasa, która go implementuje powinna robić (zestaw wymagań), ale nie jak. Co znaczy, że posiada wyłącznie deklaracje metod. Wszystkie metody w interfejsie są automatycznie publiczne. Przykładem interfejsu może być

```
public interface Comparable<T> {
    int compareTo(T other);
}
```

klasy, które go implementują mogą skorzystać ze statycznej metody sort klasy Arrays. Klasy mogą implementować dowolną liczbę interfejsów, wtedy muszą mieć implementacje wszytskich wymaganych metod.

1.12 Karta CRC

Karta CRC (Class, Responsibilities, Collaborations) to tabela podzielona na pewne części służąca do wyznaczenia zobowiązań klas obiektów oraz dla definicji związków między klasami. Takiej karty używa się jako metody, mającej na celu wyjaśnienie konceptualnych klas obiektów w systemie oraz ich asocjacji.

Class name:	
Superclass: Subclass(es):	
Responsibilities:	Collaboration:

Tabela 1.1: Szablon karty CRC

Class name: Hub	
Superclass: Building Subclass(es):	
Responsibilities:	Collaboration:

Tabela 1.2: Przykładowa karta CRC

1.13 Klasa abstrakcyjna

Klasa abstrakcyjna jest abstrakcyjnym typem danych. Oraz typem, który nie może mieć utworzonych instancji klas. Przez słowo abstrakcyjny rozumie się reprezentację takiego fragmentu rzeczywistości, który jest niemożliwy do przełożenia na realną konstrukcję, inaczej można powiedzieć, że nie jest konkretny. Przykładem klas abstrakcyjnych może być Zwierze, Pojazd, Figura czy Osoba. Każdy z tych przypadków ne ma określonej konstrukcji (bo jak opisać np. pojazd? można by powiedzieć, że pojazdem jest samochód, ale wtedy mówimy o konkretnym pojeździe, a nie samym pojeździe), ma jedynie zbiór cech oraz zachowań i to bardzo ogólnych.
W Javie klasę abstrakcyjną oznacza się słowem kluczowym *abstract*. Taka klasa może zawierać abstrakcyjne metody, tzn. takie nie posiadają implementacji. Klasa, która dziedziczy po klasie abstrakcyjnej może pozostawić niezaimplementowanie niektóre lub wszystkie metody nadklasy jednak wtedy musi być abstrakcyjna, a klasa, która zaimplementuje wszystkie metody nie musi być abstrakcyjna.

1.14 Kompozycja

Kompozycja (inaczej posiadanie) jest szczególnym przypadkiem agregacji, tj. tak zwana silna forma agregacji. Różnica pomiędzy agregacją a kompozycją jest taka, że w przypadku agregacji obiekty mogą istnieć oddzielnie, a w kompozycji jeden obiekt determinuje istnienie innych obiektów. Przykładem związku kompozycji jest klasa Blok oraz klasa Mieszkanie. Obiekty tych klas są nierozerwalnie powiązane ze sobą, usunięcie bloku spowoduje usunięcie mieszkań.

1.15 Konstruktor

Konstruktor to metoda wywoływana przy tworzeniu obiektu, której zadaniem jest utworzenie i inicjalizacja obiektu. Konstruktor musi mieć taką samą nazwę jak klasa oraz tak jak inne metody można go przeciążać, nie zwraca wartości. Różnice pomiędzy konstruktorem a zwykłą metodą są takie, że można go wywołać tylko przez operator *new*. Istnieje specjalna wersja konstruktora zwanym domyślnym. Istnieje on kiedy klasa nie posiada żadnego zadeklarowanego konstruktora albo zadeklarowany jest konstruktor bezparametrowy. W takim wypadku konstruktor ustawia wszystkie pola na wartości domyślne, czyli dane liczbowe będą wynosić 0, wartości logiczne będą ustawione na false, a zmienne obiektowe na null. Konstruktory także mogą wywołać inny konstruktor tej samej klasy przy użyciu

```
public class Window {
    private String windowTitle;
    private int width;
    private int height;

    public Window(String windowTitle, int width, int height) {
    this.windowTitle = windowTitle;
    this.width = width;
    this.height = height;
    }

    public Window(int width, int height) {
    this("window", width, height);
    }

    public Window() {}
}
```

1.16 Luźne powiązania

W rozwoju oprogramowania należy dążyć do luźnego powiązania między modułami. Oznacza to, że należy budować systemy, w których każdy moduł ma (lub wykorzystuje) tylko niewielką lub zerową wiedzę na temat definicji z innych, odrębnych modułów. Kluczem do luźnego powiązania są interfejsy. Interfejs obejmuje deklarację publicznie dostępnych operacji klasy bez zobowiązywania się do udostępniania konkretnej implementacji.

1.17 Metoda

Metoda to inaczej funkcja znajdująca się wewnątrz klasy i tak jak normalne funkcje może być przeciążana. Metoda, która ma taką samą nazwę jak nazwa klasy jest konstruktorem. Istnieje relacja pomiędzy funkcją a metodą - każda metoda jest funkcją, ale nie każda funkcja jest metodą. W Javie ze względu na pełną obiektowość języka każda funkcja jest metodą.

1.18 Parsowanie

Tłumaczenie jednego zapisu danych na inny dzięki procesowi analizy składniowej. Przykładem parsowania może być tłumaczenie ciągu znaków z postaci heksadecymalnej do dziesiętnej, czy transformacja zapisu matematycznego z postaci infiksowej do postaci pre lub postfiksowej.

1.19 Polimorfizm

Polimorfizm to możliwość odwoływania się przez obiekty (klasy bazowej) do wielu różnych typów (klas pochodnych). Dzięki regule "jest" w mechanizmie dziedziczenia, którą można sformułować także jako zasadę zamienialności. Zasada ta głosi, że wszędzie tam, gdzie można użyć obiektu nadklasy, można użyć obiektu podklasy.

```
public abstract class Employee {
    public abstract void work();
}

public class Secretary extends Employee {
    @Override
     public void work() { System.out.println("wykonywanie telefonow"); }
}

public class Charwoman extends Employee {
    @Override
    public void work() { System.out.println("sprzatanie"); }
}

Employee e1 = new Secretary();
e1.work();
```

```
Employee e2 = new Charwoman();
e2.work();
```

1.20 Rodzaje symulacji

Symulacje można podzielić ze względu na:

- przewidywalność:
  - deterministyczne
  - stochastyczne
- upływ czasu:
  - czas ciągły
  - czas dyskretny
  - symulacja zdarzeń dyskretnych
- sposób symulacji:
  - oparty o modele analityczne
  - agent-based

1.21 Testy jednostkowe

Test jednostkowy - sposób sprawdzania poprawności działania metod lub klas w programowaniu obiektowym lub procedur w programowaniu proceduralnym. Najważniejszy z aspektów testowania oprogramowania. Jest to kod, który wykonuje inny kod. Kod który jest testowany jest podzielony na mniejsze części - jednostki, które testowane są z osobna. Przykładem mogą być testy sprawdzające czy metoda equals klasy Complex zwróci spodziewane wartości.

```
@Test
public void eqTest() {
    double a = 1; double b = 5;
    Complex c1 = new Complex(a, b);
    Complex c2 = new Complex(a, b);
    Assert.assertTrue(c1.equals(c2));
}

@Test
public void notEqTest() {
    double a = 1; double b = 5;
    Complex c1 = new Complex(a, b);
    Complex c2 = new Complex(b, a);
    Assert.assertFalse(c1.equals(c2));
}
```

1.22 Wersjonowanie semantyczne

Określenie kolejności powstawania nowych wersji oprogramowania, pozwala na odróżnienie wersji między sobą. Postać takiego przedstawienia wersji to X.Y.Z, gdzie:

- **X - major** - prowadza zmiany, które nie są kompatybilne wstecz
- **Y - minor** - mniejsza poprawka lub wprowadzenie nowej funkcjonalności która jest kompatybilna wstecz
- **Z - patch** - łatka naprawiająca błędy

1.23 Wysoka spójność

Zgodnie z ogólnymi zaleceniami z obszaru rozwoju oprogramowania każdy element oprogramowania (synonimy: moduł, komponent, jednostka, klasa, funkcja) powinien mieć wysoką spójność. W ogólnym ujęciu spójność jest wysoka, jeśli moduł wykonuje dobrze zdefiniowane zadanie.

1.24 Zależność

Zależność to najsłabsza z relacji, która występuje pomiędzy klasami. Jedna klasa korzysta z innej klasy (wie o niej) przez krótką chwilę. Przykładem może być klasa Printer z metodą print(Book), która korzysta z obiektu klasy Book wyłącznie w trakcie drukowania.

```
public class Book {
}

public class Priner {
    public void print(Book book);
}
```

1.25 Zasada DRY

Don't Repeat Yourself , z pol. nie powtarzaj się - jedna z zasad czystego kodu mówiąca o wydzielaniu powtarzającej się części kodu do osobnej klasy/metody. Dzięki takiej praktyce jakakolwiek modyfikacja kodu nie stanowi dużego problemu ponieważ znajduje się w jednym miejscu, a także wzrasta czytelność kodu.

1.26 Zasada KISS

Reguła KISS (ang. Keep It Simple, Stupid), dosłownie *nie komplikuj, głupku*, mówia ona o tym, że kod powinien być tak prosty jak to tylko możliwe. Jej istotą jest dążenie do utrzymania eleganckiej i przejrzystej struktury, bez dodawania niepotrzebnych elementów.

1.27 Zadada odwrócenia zależności

Dependency Inversion Principle - wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych – zależności między nimi powinny wynikać z abstrakcji.

1.28 Zasada otwarte - zamknięte

Elementy systemu takie, jak klasy, moduły, funkcje itd. powinny być otwarte na rozszerzenie, ale zamknięte na modyfikacje. Oznacza to, iż można zmienić zachowanie takiego elementu bez zmiany jego kodu.
Jednym ze sposobów na realizację tej zasady jest dziedziczenie. Dzięki dziedziczeniu można dodawać nowe funkcje do klas bez modyfikowania tych ostatnich. Ponadto istnieje wiele wzorców obiektowych wspomagających tę zasadę. Takie wzorce to np. Strategia i Dekorator.

1.29 Zasada podstawienia Liskov

„Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów."

Innymi słowy, klasa dziedzicząca powinna rozszerzać możliwości klasy bazowej, a nie całkowicie zmieniać jej funkcjonalność. Sposób korzystania z klasy potomnej powinien być analogiczny do wywoływania klasy bazowej.

1.29.1 Przekładaj kompozycję nad dziedziczenie

Kiedy mowa o zasadzie podstawienia Liskov często pojawia się wzmianka o dylemacie prostokąta i kwadratu, który dziedziczy po tym pierwszym. W takim przypadku widać naruszenie tej zasady, inne moduły będą mogły ustawić odmienną szerokość i wysokość, a zabronienie tej możliwości to nie jest rozwiązanie problemu. W takiej sytuacji warto skorzystać z kompozycji. Kwadrat jak i prostokąt mogłyby dziedziczyć po wspólnej klasie Kształt, a żeby nie łamać zasady DRY można posłużyć się obiektem prostokąta w klasie Kwadratu.

1.30 Zasada segregacji interfejsów

Klasa udostępnia tylko te interfejsy, które są niezbędne do zrealizowania konkretnej operacji. Klasy nie powinny być zmuszane do zależności od metod, których nie używają. Klasa powinna udostępniać drobnoziarniste interfejsy dostosowane do potrzeb jej klienta. Czyli, że klienci nie powinni mieć dostępu do metod których nie używają.

1.31 Zasada SRP

Single Responsibility Principle, z pol. zasada jednej odpowiedzialności - jedna z zasad czystego kodu, która zakłada, że klasa lub moduł powinny mieć i tylko jeden powód do zmiany. Klasa powinna mieć jedną odpowiedzialność - jeden powód do zmiany. Przypisuj odpowiedzialności do obiektu tak, aby spójność była jak największa.

1.32 Zasady SOLID

- **S**ingle responsibility (pol. pojedyńcza odpowiedzialność) - klasa powinna mieć tylko jedną odpowiedzialność (nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy)
- **O**pen – Closed (pol. otwarte - zamknięte) - klasy powinny być otwarte na rozszerzenia i zamknięte na modyfikacje
- **L**iskov Substitution (pol. podstawianie Liskov) - metody które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów
- **I**nterface Segregation (pol. segregacja interfejsów) - wiele dedykowanych interfejsów jest lepsze niż jeden ogólny
- **D**ependency Inversion (pol. odwracanie zależności) - wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych - zależności między nimi powinny wynikać z abstrakcji

1.33 YAGNI

Zawsze implementuj rzeczy, gdy naprawdę ich potrzebujesz - nigdy wtedy, gdy tylko przewidujesz, że mogą ci się przydać.

Ta zasada jest ściśle powiązana z wcześniej opisaną regułą KISS. „Nie będziesz tego potrzebować" (ang. *You Aren't Gonna Need It!* lub *You Ain't Gonna Need It! - YAGNI* oznacza wypowiedzenie wojny spekulatywnemu uogólnianiuu i przekombinowaniu. Zgodnie z tą regułą nie należy pisać kody, który w danym momencie jest niepotrzebny, ale może przydać się w przyszłości.

2 Pytania

2.1 Jakie są różnice pomiędzy podejściem proceduralnym a obiektowym?

Różnica pomiędzy tymi paradygmatami programowania polega na przetrzymywaniu zmiennych oraz ich przepływie. W podejściu proceduralnym zmienne trzymane są w jednym miejscu (najczęściej głównej funkcji), a ich przetwarzaniem zajmują się podprogramy (funkcje lub procedury). Podprogramy pobierają i przekazują

wszystkie dane (czy też wskaźniki do nich) jako parametry wywołania. Natomiast w podejściu obiektowym jest wprowadzone pojęcie abstrakcji, dzięki której tworzy się własne typy danych (klasy), które są powiązane ze sobą w ściśle określony sposób. Instancje klas, czyli obiekty komunikują się wzajemnie z celu wykonania określonych zadań za pomocą interfejsów.

2.2 Jaka jest różnica pomiędzy klasą a obiektem?

Klasa jest zdefiniowanym przez programistę typem danych, obiekt to instancja klasy. Klasa opisuje fragment rzeczywistości przez abstrakcję (jest zbiorem cech - pól i zachowań - metod), a obiekt jest jest rzeczywistym odzwierciedleniem, na którym można już wykonywać pewnie działania (metody).

2.3 Jakie modyfikatory dostępu do składowych klasy są dostępne w języku Java?

- **Private** - pola i metody z tym modyfikatorem są wyłącznie dostępne wewnątrz klasy
- **Protected** - pola i metody są widoczne w pakiecie i we wszystkich podklasach
- **Public** - pola, metody, klasy i interfejsy są widoczne wszędzie
- **Domyślny** - nie jest oznaczony żadnym słowem kluczowym. Dostęp do pól, metod, klas i interfejsów jest możliwy tylko dla klas będących w tym samym pakiecie

Zasięg\Modyfikator	Private	Protected	Public	Domyślny
Wewnątrz klasy	tak	tak	tak	tak
Pakiet	nie	tak	tak	tak
Pakiet i podklasy	nie	tak	tak	tak
Inne pakiety	nie	nie	tak	nie
Inne pakiety i podklasy	nie	tak	tak	nie

2.4 Do czego służy metoda *finalize* w języku Java?

Metoda *finalize* jest w pewnym sensie destruktorem obiektu znanego z języka C++. Wywoływana jest przez garbage collector na obiekcie w trakcie jego niszczenia i służy do zwalniania zewnętrznych zasobów takich jak pliki czy połączenia internetowe. Również nie ma pewności co do wywołania tej metody przed zakończeniem programu, dlatego nie należy polegać na niej co do przywracania zasobów.

```
@Override
protected void finalize() throws Throwable {
    try {
        ... // cleanup subclass state
    } finally {
        super.finalize();
    }
}
```

2.5 Z jakiego mechanizmu języka Java skorzystasz aby hermetyzować zachowania?

Hermetyzowanie (inaczej enkapsulowanie) można dokonać za pomocą odpowiednich metod dostępu. Metoda private powinna być użyta na polach klasy oraz niektórych metodach, tzn. takich których mechanizm nie musi być wykorzystywany przez inne obiekty. Oraz metody public do uwidocznienia interfejsu danej klasy. Dostępy: domyślny oraz protected powinny być używane jak najmniej, za to powinno używać się getterów i setterów do uzyskiwania dostępu do pól.

2.6 Jaka jest różnica pomiędzy interfejsem języka Java a klasą abstrakcyjną?

- Klasa abstrakcyjna może rozszerzać jedną klasę i implementować dowolną liczbę interfejsów, a interfejs może rozszerzać dowolną liczbę interfejsów, w przypadku interfejsu nie ma mowy o implementowaniu innych interfejsów

- Klasa abstrakcyjna może posiadać metody abstrakcyjne jak i zwykłe. Interfejs natomiast posiada wyłącznie metody abstrakcyjne (nie używa się tam słowa kluczowego *abstract*), wszystkie metody interfejsu są publiczne
- Klasa abstrakcyjna może posiadać pola jak każda zwykła klasa, a interfejs może posiadać wyłącznie pola, które są statyczne, publiczne i final

2.7 Czy w języku Java jest możliwe dziedziczenie wielobazowe?

Nie. Klasa może dziedziczyć wyłącznie po jednej klasie. Może natomiast implementować dowolną liczbę interfejsów. Zaś interfejs może rozszerzać (dziedziczyć) dowolną liczbę interfejsów, nie mówimy wtedy o implementacji.

```
public class Animal {
    private double age;
    public Animal(double age) {this.age = age;}
}

public class Bat extends Animal { // wylacznie dziedziczenie po jednej
    private String name;
    public Bat(double age, String name) { super(age); this.name = name; }
    public Bat() { super(0); this.name = ""; }
    public void fly() { System.out.println("LECE"); }
}


public interface InterfaceA {
    public String getString();
}

public interface InterfaceB {
    public int getInt();
}

public interface InterfaceC extends InterfaceA, InterfaceB {
    public double getDouble();
}

public interface InterfaceD {
    public boolean getBoolean();
}

public class ClassA {
    private String reallyUnnecessaryVariable = "XD";
}

public ClassB extends ClassA implements InterfaceC, InterfaceD {
    public String getString() { return "Hello World"; }
    public int getInt() { return 0; }
    public double getDouble() { return 0.0; }
    public boolean getBoolean() { return false; }
}
```

2.8 Jak wiele interfejsów może implementować klasa w języku Java?

Klasa może implementować dowolną liczbę interfejsów (patrz wyżej). W przypadku implementacji wielu interfejsów należy zdefiniować metody wszystkich interfejsów, które implementuje.

2.9 Czym jest klasa finalna?

Klasa finalna, to klasa po której nie można dziedziczyć. Do zadeklarowania klasy finalnej należy użyć słowa kluczowego *final* przed *class*.

2.10 Wyjaśnij jaka jest różnica pomiędzy kompozycją a agregacją.

Kompozycja jest szczególnym przypadkiem agregacji. W przypadku agregacji obiekty mogą istnieć niezależnie od siebie, a w przypadku kompozycji jeden obiekt składa się z innych obiektów, a jego zniszczenie powoduje zniszczenie obiektów w nim zawartych.

3 Wzorce projektowe

3.1 O wzorcach projektowych

Wzorzec projektowy (ang. design pattern) – w inżynierii oprogramowania, uniwersalne, sprawdzone w praktyce rozwiązanie często pojawiających się, powtarzalnych problemów projektowych. Pokazuje powiązania i zależności pomiędzy klasami oraz obiektami i ułatwia tworzenie, modyfikację oraz pielęgnację kodu źródłowego. Jest opisem rozwiązania, a nie jego implementacją. Wzorce projektowe stosowane są najczęściej w projektach wykorzystujących programowanie obiektowe.

3.2 Wzorce kreacyjne

Kreacyjne wzorce projektowe izolują reguły tworzenia obiektów od reguł określających sposób używania obiektów (oddzielenie w kodzie programu kodu tworzącego obiekty od kodu, który używa obiekty)

3.2.1 Singleton

Wzorzec projektowy, którego celem jest ograniczenie możliwości tworzenia obiektów danej klasy do jednej instancji. Klasa, która jest singletonem zawiera w sobie statyczne pole, które jest jej instancją, dzięki czemu uniemożliwia tworzenie dodatkowych instancji. Konstruktor takiej klasy jest chroniony albo prywatny i jest dostępny wyłącznie przez metodę, dzięki której można utworzyć jedyną instancję klasy.

3.2.2 Builder

Jest wzorcem, gdzie proces tworzenia obiektu podzielony jest na kilka mniejszych etapów, a każdy z nich może być implementowany na wiele sposobów. Dzięki takiemu rozwiązaniu możliwe jest tworzenie różnych reprezentacji obiektów w tym samym procesie konstrukcyjnym. Rozdziela również sposób tworzenia obiektu od ich reprezentacji. Wzorzec Builder można podzielić na dwa rodzaje, pierwszy - Budowniczy - zna szczegóły budowy obiektu, a drugi - Architekt - wie co należy zrobić - w jakiej kolejności.

3.2.3 Abstract Factory

Wzorzec ten ma za zadanie udostępnienie jednego interfejsu, za którym mogą stać różne klasy po nim dziedziczące. Metody tych klas mają za zadanie tworzenie (produkowanie) nowych obiektów, które najczęściej będą również interfejsami związanymi z obiektami klas po nich dziedziczących. Koniec końców w tym wzorcu chodzi o to, żeby za jednym interfejsem można było osadzić różne wersje klas produkujących za pomocą tych samych metod różne obiekty.

3.2.4 Prototype

Prototyp jest wzorcem, opisującym mechanizm tworzenia nowych obiektów poprzez klonowanie jednego obiektu macierzystego. Mechanizm klonowania wykorzystywany jest wówczas, gdy należy wykreować dużą liczbę obiektów tego samego typu lub istnieje potrzeba tworzenia zbioru obiektów o bardzo podobnych właściwościach.

3.2.5 Factory Method

Wzorzec metody wytwórczej dostarcza abstrakcji do tworzenia obiektów nieokreślonych, ale powiązanych typów. Umożliwia także dziedziczącym klasom decydowanie jakiego typu ma to być obiekt. Wzorzec składa się z dwóch ról: produktu Product definiującego typ zasobów oraz kreatora Creator definiującego sposób

ich tworzenia. Wszystkie typy produktów (ConreteProduct1, ConreteProduct2 itp.) muszą implementować interfejs Product. Z kolei ConcreteCrator dostarcza mechanizm umożliwiający stworzenie obiektu produktu danego typu.

3.3 Wzorce strukturalne

Wzorce strukturalne opisują struktury powiązanych ze sobą obiektów.

3.3.1 Wraper

Wzorzec Wraper (znany także pod nazwą Adapter) służy do przystosowania interfejsów obiektowych, tak aby możliwa była współpraca obiektów o niezgodnych interfejsach. Szczególnie przydaje się przypadku wykorzystania gotowych bibliotek o interfejsach niezgodnych ze stosowanymi w aplikacji. W świecie rzeczywistym adapter to przejściówka, np. przejściówka do wtyczki gniazdka angielskiego na polskie

3.3.2 Decorator

Wzorzec służący do ulepszania i rozbudowywania obiektów. Załóżmy, że chcemy dodać do jakiejś klasy metodę (niech jeszcze będzie to abstrakcja lub metoda wirtualna, a klasa ta będzie nadklasą dla innych klas). Dzięki dekoratorowi nie musimy tutaj nic modyfikować, możemy nadać obiektowi nowe zachowanie podczas działania programu. Dodatkowo wzorzec ten zapobiega przed eksplozją klas, spowodowaną dużą liczbą rozszerzeń danej klasy (np. kombinacji rożnych składników na pizzy może być mnóstwo). Dekorator musi mieć taki sam interfejs jak obiekt, który będziemy dekorować (obiekty dekorujące są tego samego typu co obiekty dekorowane). Do dekoratora przekazujemy dekorowany obiekt. W przeciwieństwie do dziedziczenia, obiekty dostają nowe funkcjonalności dynamicznie (w trakcie działania programu), a nie na etapie kompilacji. Klient wcale nie musi wiedzieć o działaniu wzorca.

3.3.3 Facade

Fasada służy do ujednolicenia dostępu do złożonego systemu poprzez udostępnienie uproszczonego i uporządkowanego interfejsu programistycznego. Fasada zwykle implementowana jest w bardzo prosty sposób – w postaci jednej klasy powiązanej z klasami reprezentującymi system, do którego klient chce uzyskać dostęp.

3.3.4 Composite

Wzorzec projektowy, którego zadaniem jest łączenie obiektów w strukturę tak, że reprezentują hierarchię części-całość, unifikując dostęp do kolekcji jak i pojedynczego obiektu. Umożliwia to klientom jednolite traktowanie pojedynczych obiektów i również ich kompozycji.

3.3.5 Bridge

Oddziela interfejs danych od interfejsu realizacji określonego zadania. Często też służy jako podział interfejsu układu sterującego od układu sterowanego (np. pilot - urządzenie). Ten wzorzec umożliwia stworzenie warstwy opisu i oddzielnej warstwy realizacji np. pod różne platformy systemowe.

3.3.6 Flyweight

Wzorzec ten umożliwia wydzielenie części wspólnej dla wielu obiektów, które niewiele się różnią albo są identyczne i przechowywanie jej w jednym oddzielnym obiekcie, do którego obiekt główny odwołuje się poprzez wskaźnik lub referencję. Wszystko opiera się na współdziałaniu, polegającym na modyfikacji atrybutów tego obiektu (przekazujemy do niego tylko to co uległo zmianie). Efektem takiej pracy jest znaczne zmniejszenie zapotrzebowania naszej aplikacji na pamięć (zamiast całej masy podobnych, jak nie identycznych, obiektów, przechowujemy tylko jedna instancję). Jedyną wadą stosowania tego wzorca jest spadek wydajności aplikacji.

3.3.7 Proxy

Jego celem jest utworzenie obiektu, który kontroluje (zastępuje) inny obiekt. Wzorzec ten może być wykorzystany np. do inicjacji zawartego wewnątrz obiektu dopiero w momencie próby wykonania na nim jakiejś operacji, przyczyną takiego stanu rzeczy może być fakt, że obiekt nie zawsze będzie potrzebny a jego utworzenie jest czasochłonne. Innym zastosowaniem może być ograniczenie dostępu i przechwytywanie zadań obiektu, którego zestępuje.

3.4 Wzorce behawioralne

Opisują zachowanie i odpowiedzialność współpracujących ze sobą obiektów.

3.4.1 Visitor

Głównym celem wzorca Odwiedzającego jest dodawania nowych funkcjonalności do istniejącego obiektu bez modyfikowania klasy na których ten obiekt operuje. Implementacja w skrócie wygląda tak, że klasa Visitor deklaruje metodę visit(), której argumentem jest jakiś element struktury oraz elementy struktury deklarują metodę accept(), która za argument przyjmuję klasę Visitor, metoda accept() wywołuje metodę visit() w klasie Visitor w zależności od typu przekazanego argumentu.

3.4.2 Chain of Responsibility

Umożliwia on tworzenie listy zadań (czynności), które będą wykonywane kolejno do momentu, gdzie nie będzie już więcej zadań do zrealizowania. Każde pod zadanie jest reprezentowane przez oddzielną klasę, która dziedziczy po wspólnym interfejsie. Interfejs ten agreguje interfejsy nowych zadań dodanych do listy. Łańcuch zobowiązań może być zamknięty lub otwarty. W przypadku jego zamknięcia zadania będą realizowane do momentu przerwania przez użytkownika.

3.4.3 Template Method

Jego zadaniem jest zdefiniowanie metody, będącej szkieletem algorytmu. Algorytm ten może być następnie dokładnie definiowany w klasach pochodnych. Niezmienna część algorytmu zostaje opisana w metodzie szablonowej, której klient nie może nadpisać. W metodzie szablonowej wywoływane są inne metody, reprezentujące zmienne kroki algorytmu. Mogą one być abstrakcyjne lub definiować domyślne zachowania. Klient, który chce skorzystać z algorytmu, może wykorzystać domyślną implementację bądź może utworzyć klasę pochodną i nadpisać metody opisujące zmienne fragmenty algorytmu.

3.4.4 Strategy

Jego implementacja umożliwia wybranie klasy dziedziczącej po danym interfejsie, która to odpowiada za sposób realizacji zadania lub zadań tegoż obiektu. Co znaczy, że definiuje rodzinę wymiennych algorytmów i kapsułkuje je w postaci klas. Wszystkie klasy dziedziczące realizują to samo zadanie, ale w odmienny sposób.

3.4.5 State

Celem tego wzorca jest zaimplementowanie w głównej klasie możliwości wyboru interfejsu, z którego klasa ta będzie korzystała, czyli może zmienić stan (zachowanie) wewnętrznego obiektu. Taka implementacja może mieć miejsce np. przy wyborze trybu rysowania obiektów w programie graficznym.

3.4.6 Observer

Obserwator składa się z obiektu, który nazywamy „obiektem obserwowanym" oraz pewnej liczy obiektów obserwujących. Występuje tutaj relacja typu jeden do wielu. Obiekt obserwowany jest zarządcą danych, informuje on wszystkich swoich obserwatorów o zmianach w danych, które zawiera.

3.4.7 Mediator

Jego celem jest udostępnienie interfejsu pośredniczącego w przekazywaniu informacji pomiędzy różnymi nie połączonymi bezpośrednio z sobą obiektami klas (zna metody wszystkich klas, którymi zarządza), które dziedziczą po wspólnym interfejsie. Klasa pośrednicząca zawiera tablicę interfejsów klas.

3.4.8 Momento

Celem tego wzorca jest zaimplementowanie możliwości stworzenia kopii danych przechowywanych w obiekcie lub obiektach klasy. Pamiątka stosowana jest do implementacji cofania i przywracania zmian często spotykanych w programach graficznych czy też tekstowych. Pamiątka może też być swego rodzaju systemem pozwalającym np. zapisywać różne stany gry na dysku twardym, lub przywracać fabryczne ustawienia programu.

4 Unified Modeling Language

4.1 Czym jest UML

Unified Modeling Language (UML, zunifikowany język modelowania) to rozbudowany język modelowania systemów i tworzenia specyfikacji. W programowaniu obiektowych jest wykorzystywany do przedstawiania klas oraz zależności pomiędzy nimi.

4.2 Diagram klas

4.2.1 Przedstawienie klasy i interfejsu

Klasa w UML jest przedstawiona jako tabela z trzema wierszami. Na samej górze jest pogrubiona nazwa klasy, w środku przedstawione są pola, a na dole metody.

NazwaKlasy
modyfikator nazwaPola: typ
modyfikator nazwaMetody(typArgumentu1, typArgumentu2, ...): typ zwracany

Tabela 4.1: Przedstawienie klasy w diagramach klas

Klasa abstrakcyjna przedstawiona jest jak zwykła klasa z tą różnicą, że jej nazwa jest pochylona.

NazwaAbstrakcyjnejKlasy
modyfikator nazwaPola: typ
modyfikator nazwaMetody(typArgumentu1, typArgumentu2, ...): typ zwracany modyfikator *nazwaAbstrakcyjnejMetody(typArgumentu1, typArgumentu2, ...): typ zwracany*

Tabela 4.2: Przedstawienie klasy abstrakcyjnej w diagramach klas

W interfejsie nad nazwą interfejsu znajduje się napis «*interface*». Modyfikator pola i metody zawsze będzie public.

«interface» **NazwaInterfejsu**
+ NAZWAPOLA: typ
+ nazwaMetody(typArgumentu1, typArgumentu2, ...): typ zwracany

Tabela 4.3: Przedstawienie interfejsu w diagramach klas

4.2.2 Zmienne i metody

Zasady dotyczące przedstawiania zmiennych i metod:

- static - za pomocą podkreślenia
- final - wyłącznie wielkie litery
- abstract - pochylenie (wyłącznie metody)

4.2.3 Modyfikatory

Modyfikator	Znak
private	-
protected	#
public	+
package (defualt)	~

Tabela 4.4: Modyfikatory dostępu w diagram klas

4.2.4 Związki

Rysunek 4.1: Asocjacja z zaznaczaną rolą po jednej stronie

Rysunek 4.2: Zależność

Rysunek 4.3: Agregacja

Rysunek 4.4: Kompozycja

Rysunek 4.5: Dziedziczenie

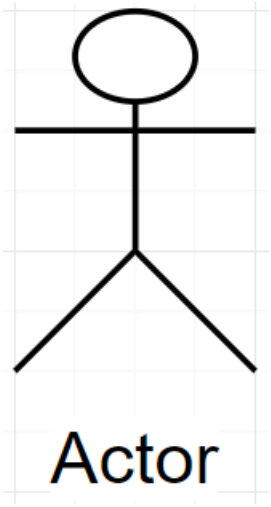
Rysunek 4.6: Implementacja interfejsu

4.3 Diagram przypadków użycia

Diagram przypadków użycia (ang. use case) – graficzne przedstawienie przypadków użycia, aktorów oraz związków między nimi, występujących w danej dziedzinie przedmiotowej. Diagram przypadków użycia w języku UML służy do modelowania funkcjonalności systemu. Tworzony jest zazwyczaj w początkowych fazach modelowania. Diagram ten stanowi tylko przegląd możliwych działań w systemie, szczegóły ich przebiegu są modelowane za pomocą innych technik (np. diagramów stanu lub aktywności). Diagram przypadków użycia przedstawia usługi, które system świadczy aktorom, lecz bez wskazywania konkretnych rozwiązań technicznych.

4.3.1 Postać aktora

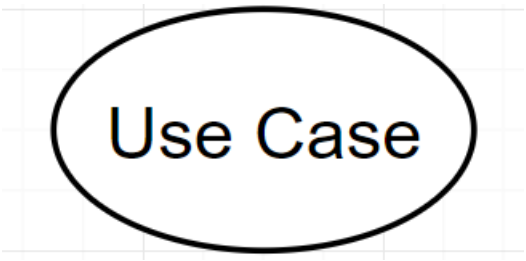
Aktor (ang. actor) jest rolą, którą pełni użytkownik w stosunku do systemu oraz przypadków użycia. Aktor reprezentuje spójny zbiór ról, które są odgrywane przez użytkowników przypadku użycia w czasie interakcji z tym przypadkiem. Aktorem może być człowiek, urządzenie lub inny system. Aktor nie musi być fizycznym obiektem. Istotne jest, by pełnił określoną funkcję wobec systemu i przypadku użycia, którego używa. Aktor reprezentuję rolę, w którą człowiek, urządzenie bądź inny system może się wcielić w trakcie współpracy z modelowanym systemem.



Rysunek 4.7: Postać aktora w diagramie przypadków użycia

4.3.2 Przypadek użycia

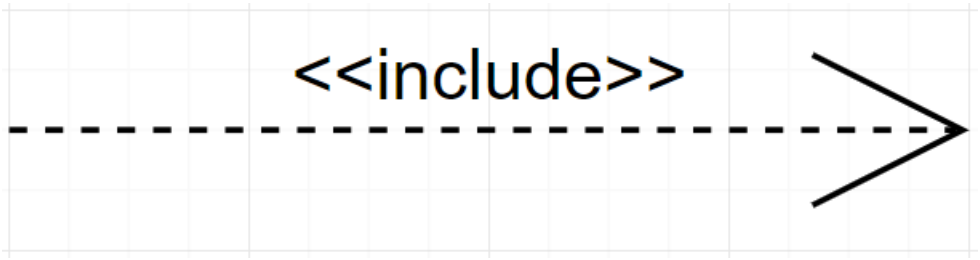
Przypadek użycia (ang. use case) to zbiór scenariuszy powiązanych ze sobą wspólnym celem użytkownika. Przypadek użycia jest graficzną reprezentacją wymagań funkcjonalnych. Definiuje zachowanie systemu bez informowania o wewnętrznej strukturze i narzucania sposobu implementacji. Przypadek użycia pozwala na zdefiniowanie przyszłego, spodziewanego zachowania systemu. Dostarcza także kwant funkcjonalności dostępnej dla użytkownika. Przypadki użycia są stosowane w całej analizie systemu i mają za zadanie dostarczyć wyniki, z których użytkownik będzie mógł skorzystać, i które go zainteresują. Istotny jest fakt, że przypadek użycia musi być w interakcji chociaż z jednym aktorem. Wyjątek stanowi sytuacja, gdy przypadek użycia jest połączony związkiem rozszerzenia lub zawierania z innym przypadkiem użycia.



Rysunek 4.8: Przypadek użycia

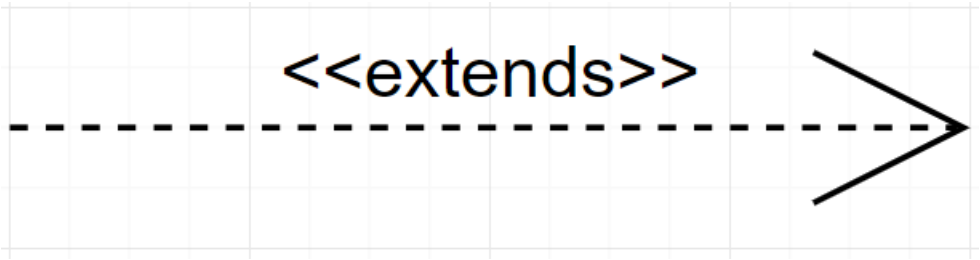
4.3.3 Związek zawierania i rozszerzania

Strukturalne związki, przedstawiane na diagramach przypadków użycia, opisują zależności między elementami modelu usług, określając: całość (bazowy przypadek użycia) i część (zawierany lub rozszerzający przypadek użycia) oraz hierarchię (poprzez związek generalizacji). Związek zawierania (ang. include) polega na rozszerzaniu funkcjonalności bazowego przypadku użycia o zachowanie innego przypadku użycia. Istotny jest fakt, że związek zawierania zawsze skierowany jest grotem w stronę zawieranego przypadku użycia.



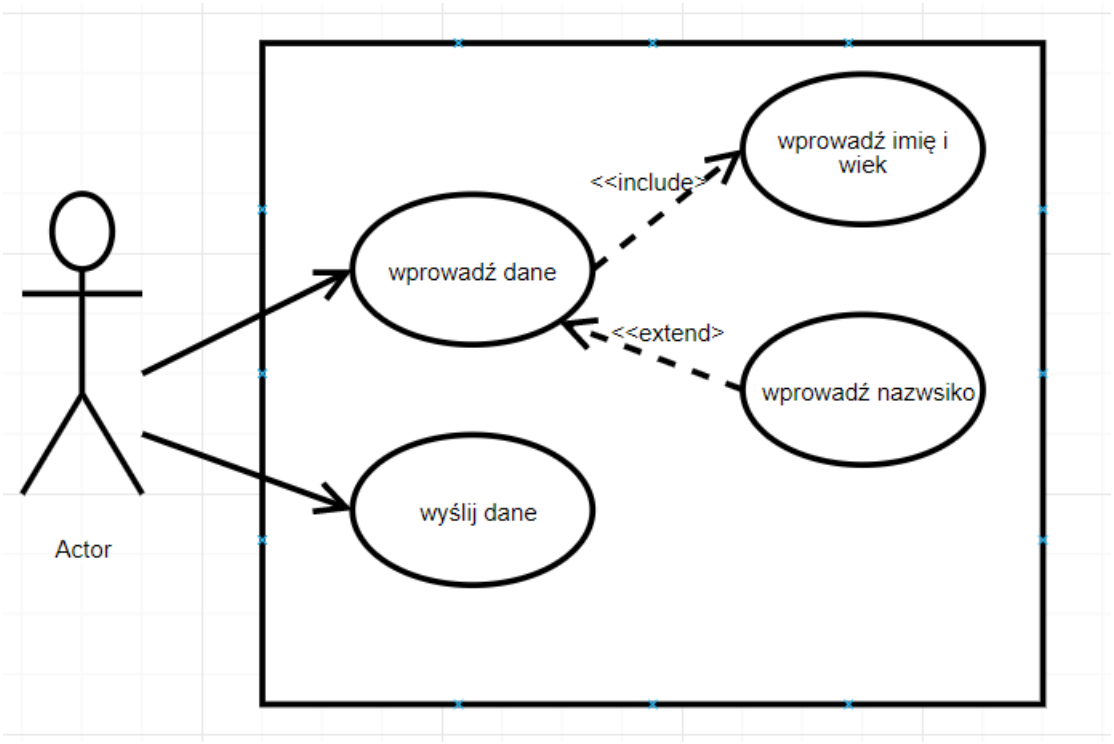
Rysunek 4.9: Związek zawierania

Inną sytuację przedstawia związek rozszerzenia (ang. extend), który wskazuje, że dany przypadek użycia opcjonalnie rozszerza funkcjonalność bazowego przypadku użycia. Funkcjonalność bazowego przypadku użycia jest rozszerzana o inny przypadek użycia po spełnieniu określonego warunku. Warunek taki może być zapisany w notce dołączonej do zależności.



Rysunek 4.10: Związek rozszerzania

4.3.4 Przykładowy diagram przypadków użycia



Rysunek 4.11: Przykładowy diagram przypadków użycia

4.4 Diagram sekwencji

Diagram sekwencji (ang. sequence diagram) służy do prezentowania interakcji pomiędzy obiektami wraz z uwzględnieniem w czasie komunikatów, jakie są przesyłane pomiędzy nimi.

4.4.1 Układ diagramu

Oś pozioma przedstawia instancje klasyfikatorów biorące udział w interakcji, natomiast pionowa chronologiczne ułożenie komunikatów, gdzie czas jest „skierowany" w dół.

4.4.2 Podstawowe pojęcia

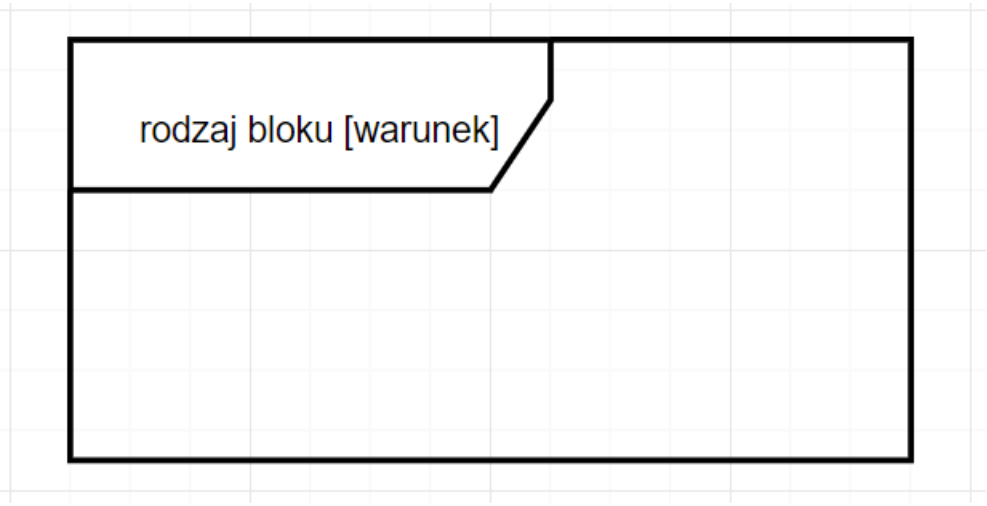
- Instancja klasyfikatora - abstrakcyjna kategoria modelowania, która uogólnia kolekcję instancji o tych samych cechach
- Linia życia - powiązana z konkretną instancją klasyfikatora linia na diagramie sekwencji wskazująca okres istnienia tej instancji
- Komunikat - specjacja wymiany informacji między obiektami, zawierająca zlecenia wykonainia określonej operacji
- Ośrodek sterowania - specyfikacja wykonywania czynności, operacji lub innej jednostki zachowania w ramach interakcji - w praktyce realizacja funkcjonalności

4.4.3 Rodzaje klasyfikatorów

- Aktor
- Przypadek użycia
- Klasa
- Sygnał
- Pakiet
- Interfejs
- Komponent

4.4.4 Bloki i ich rodzaje

- alt (od alternative) - określający warunek wykonania bloku operacji, odpowiadający instrukcji if-else ; warunek umieszcza się wówczas wewnątrz bloku w nawiasach kwadratowych
- opt (od optional) - reprezentujący instrukcję if (bez else)
- par (od parallel) - nakazujący wykonać operacje równolegle
- critical - blok atomowy, oznaczający obszar krytyczny
- loop - definiujący pętlę typu for (o określonej z góry liczbie iteracji) lub while (wykonywanej dopóki pewien warunek jest prawdziwy)
- break - wykonanie fragmentu i zakończenie interakcji
- seq - słaba sekwencja (podobnie do współbieżności) dotyczy zdarzeń z kilku linii
- ignore/consider - ignore(komunikat1, komunikat2, ...) oznacza, że na diagramie nie pokazano wymienionych komunikatów, choć mogą wystąpić

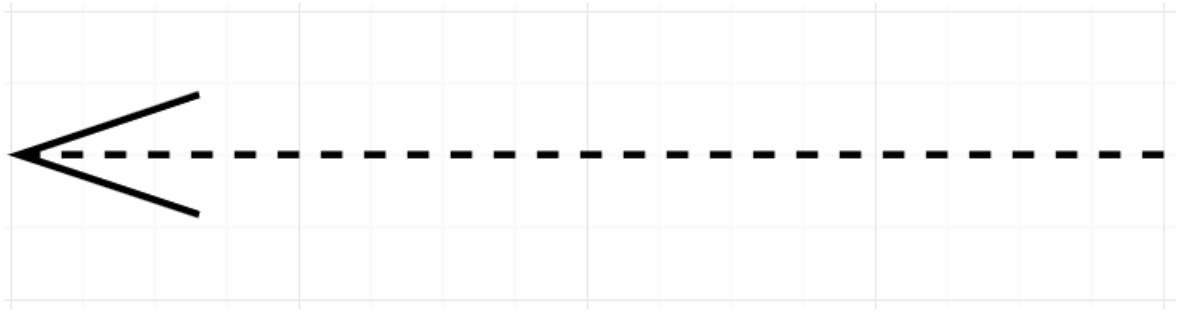


Rysunek 4.12: Blok

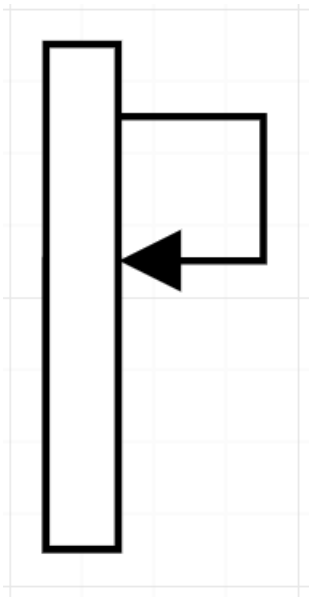
4.4.5 Rodzaje komunikatów

Rysunek 4.13: Komunikat synchroniczny

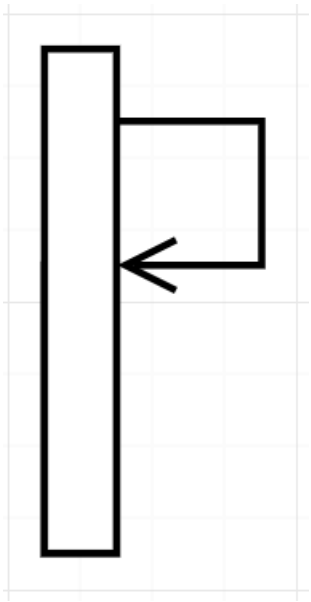
Rysunek 4.14: Komunikat asynchroniczny



Rysunek 4.15: Komunikat zwrotny

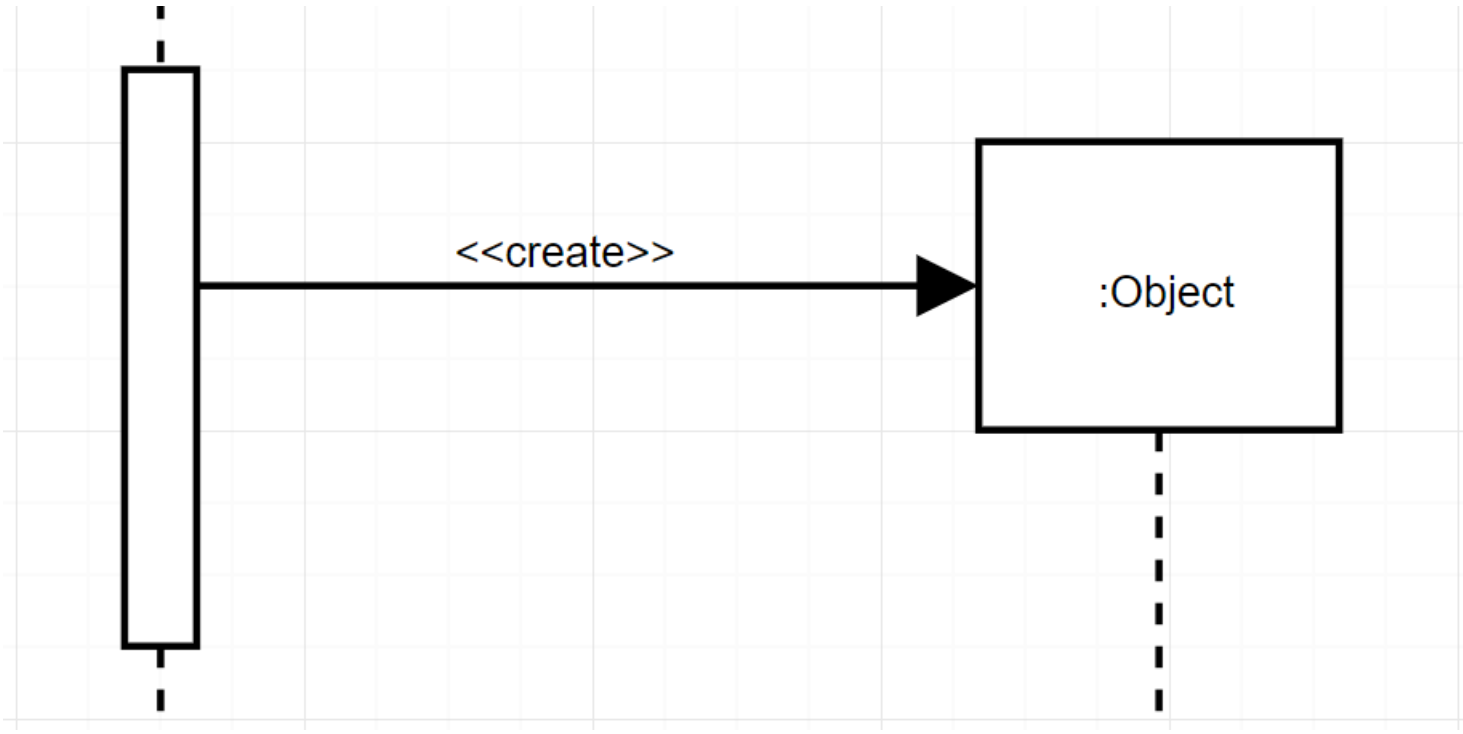


Rysunek 4.16: Samowywołanie synchroniczne

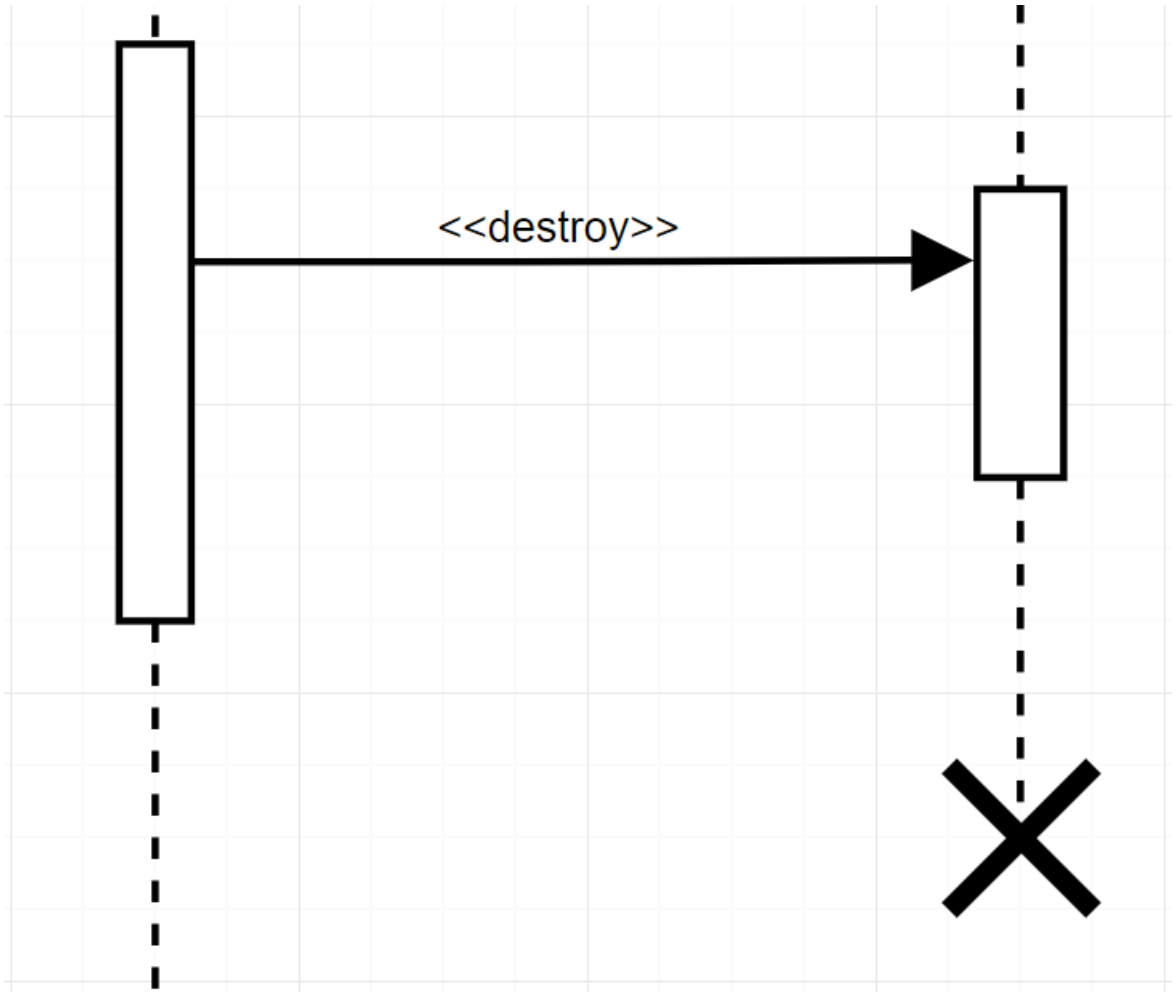


Rysunek 4.17: Samowywołanie asynchroniczne

4.4.6 Tworzenie i niszczenie obiektów



Rysunek 4.18: Tworzenie - stereotyp «create»



Rysunek 4.19: Niszczenie - stereotyp «destroy»

4.4.7 Komunikat warunkowy

Określa warunek jaki musi być spełniony, aby operacja wskazywana przez komunikat została wykonana.

4.4.8 Iteracja

Wielokrotne, policzalne powtórzenie pojedynczego komunikatu

4.5 Diagram obiektów

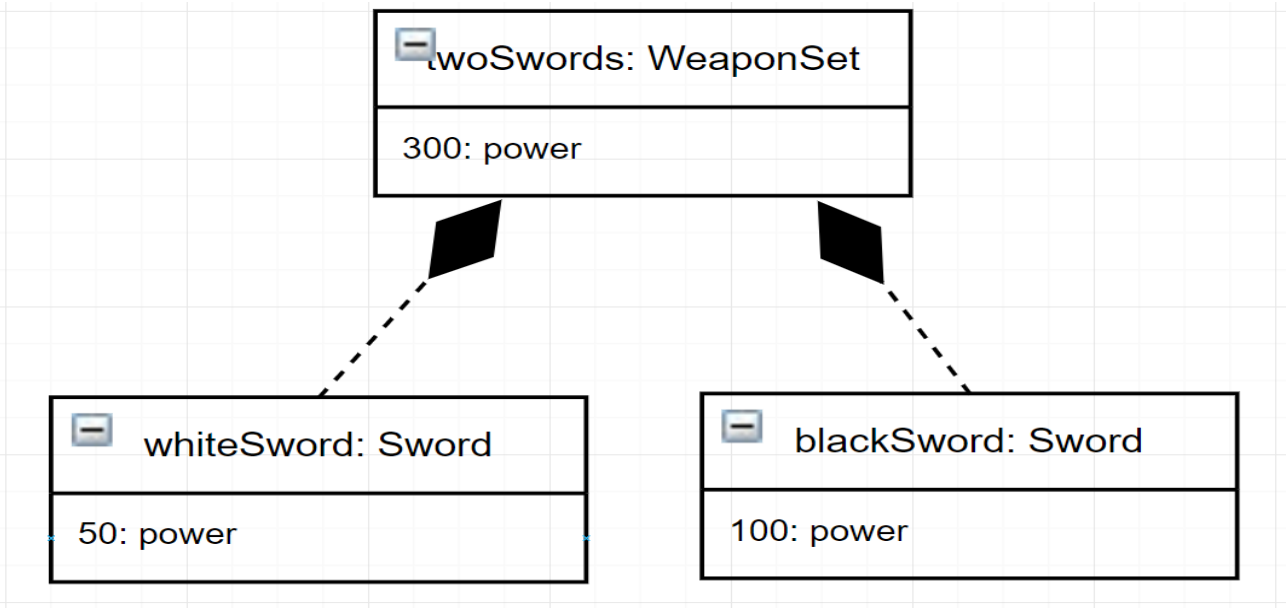
Na diagramie obiektów przedstawia się obiekty, czyli konkretne instancje klas i związki między nimi. Diagram ten wyobraża statyczny rzut pewnych egzemplarzy elementów występujących na diagramie klas. Podobnie jak diagram klas, odnosi się do statycznych aspektów perspektywy projektowej lub procesowej. Korzystając z niego bierze się jednak pod uwagę przypadki rzeczywiste lub prototypowe. Każdy obiekt może być opisany przy pomocy trzech elementów: tożsamości, stanu i zachowania. Tożsamość jest cechą wyróżniającą obiekt, przedstawiając go jako indywidualną jednostkę. Określa się ją przy pomocny unikatowej i indywidualnej cechy obiektu, która nigdy nie ulega zmianie.

4.5.1 Przedstawienie obiektu

nazwa obiektu : nazwa klasy
nazwa pola$_1$: wartość pola$_1$ nazwa pola$_2$: wartość pola$_2$ ... nazwa pola$_n$: wartość pola$_n$

Tabela 4.5: Przedstawienie obiektu na diagramie obiektów

4.5.2 Przykładowy diagram obiektów



Przykładowy diagram obiektów

4.6 Diagram aktywności

Diagram czynności (ang. activity diagram) jest diagramem interakcji, który służy do modelowania dynamicznych aspektów systemu. Jego zasadniczą funkcją jest przedstawienie sekwencji kroków, które są wykonywane przez modelowany fragment systemu.

4.6.1 Aktywność

Aktywność (ang. activity) jest sparametryzowanym zachowaniem systemu, przedstawionym w postaci uporządkowanych, podrzędnych elementów, z których najważniejszym jest akcja. Aktywność służy do zaprezentowania tych behawioralnych aspektów systemu, które mogą zostać zdekomponowane za pomocą czynności. Aktywność może stanowić oddzielny diagram czynności.

Rysunek 4.21: Aktywność z dwoma parametrami wejściowymi i jednym wyjściowym

4.6.2 Czynność

Czynność (ang. action) jest wykonywalnym węzłem aktywności, który reprezentuje transformację lub proces modelowanego systemu. Wynikiem działania czynności może być zmiana stanu systemu lub zwrócenie wyniku (ang. reference). Czynność nie podlega dekompozycji.

Rysunek 4.22: Czynność

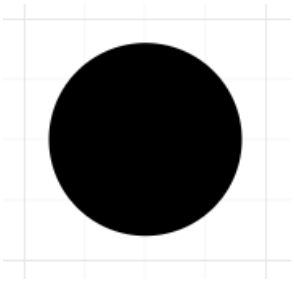
4.6.3 Przepływ sterujący i obiektu

Przepływ sterujący (ang. control flow) to element, który prezentuje przejście pomiędzy węzłami diagramu czynności.
Przepływ obiektu (ang. data flow), zwany także przepływem danych to element, który prezentuje przejście pomiędzy węzłami diagramu widoku interakcji.

Rysunek 4.23: Przepływ sterujący i obiektu (jest taki sam)

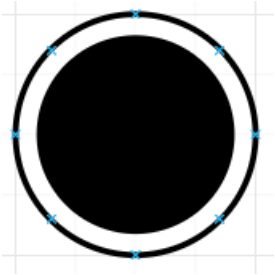
4.6.4 Węzeł początkowy i końcowy

Węzeł początkowy (ang. activity initial node) to element rozpoczynający aktywność. Aktywność może mieć tylko jeden węzeł początkowy.



Rysunek 4.24: Węzeł początkowy

Węzeł końcowy (ang. activity finale node) to element kończący aktywność. Aktywność może mieć więcej niż jedno zakończenie.

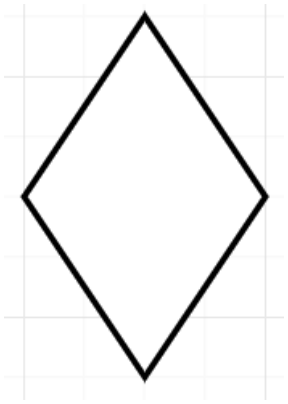


Rysunek 4.25: Węzeł końcowy

4.6.5 Węzeł decyzyjny i połączenia

Węzeł decyzyjny (ang. decision node) to element, który umożliwia dokonanie wyboru pomiędzy kilkoma możliwościami. Umieszczenie węzła decyzyjnego na diagramie oznacza, że nie ma jednej ścieżki wykonywania poszczególnych aktywności (istnieją ścieżki alternatywne). Węzeł decyzyjny może mieć jedno wejście dla

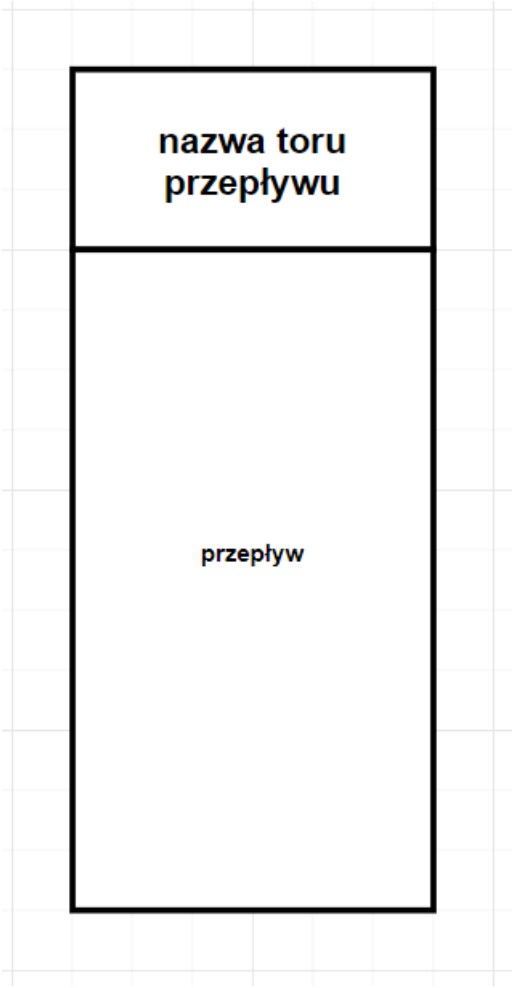
przepływu sterującego i minimalnie dwa wyjścia przepływów. Na wyjściach z węzła decyzyjnego znajdują się wykluczające się warunki dozoru. Istotne jest, by warunki dozoru uwzględniały wszystkie możliwości, tak by nie dopuścić do zatrzymania przepływu na węźle.
Język UML dopuszcza, by jedno wyjście z węzła było opisane warunkiem [else], który oznacza ścieżkę, jaka powinna być wybrana w przypadku niespełnienia warunku dozoru wszystkich pozostałych wyjść.
Węzeł połączenia (ang. merge node) to węzeł, w którym następuje scalenie kilku alternatywnych przepływów. Węzeł ten nie jest elementem synchronizującym.



Rysunek 4.26: Węzeł decyzyjny i połączenia

4.6.6 Partycje

Partycja aktywności (ang. activity partition) jest to część diagramu czynności, która grupuje czynności charakteryzujące się podobnymi cechami. Partycja aktywności w UML 1.x nosiła nazwę torów (ang. swimlines). Partycję aktywności można stosować zarówno pionowo, jak i poziomo, co umożliwia grupowanie czynności w wydajniejszy sposób.



Rysunek 4.27: Partycja

4.7 Diagram stanów

Diagram stanów określa ciągi stanów przyjmowanych przez obiekt w odpowiedzi na zdarzenia zachodzące w czasie jego życia, a także reakcje obiektu na te zdarzenia.

4.7.1 Elementy diagramu stanów

- stan - zaokrąglony prostokąt
- przejście - strzałka z ostrym grotem
- stan
  - początkowy - kropka
  - końcowy - kropka z okręgiem
  - zniszczenia - iks

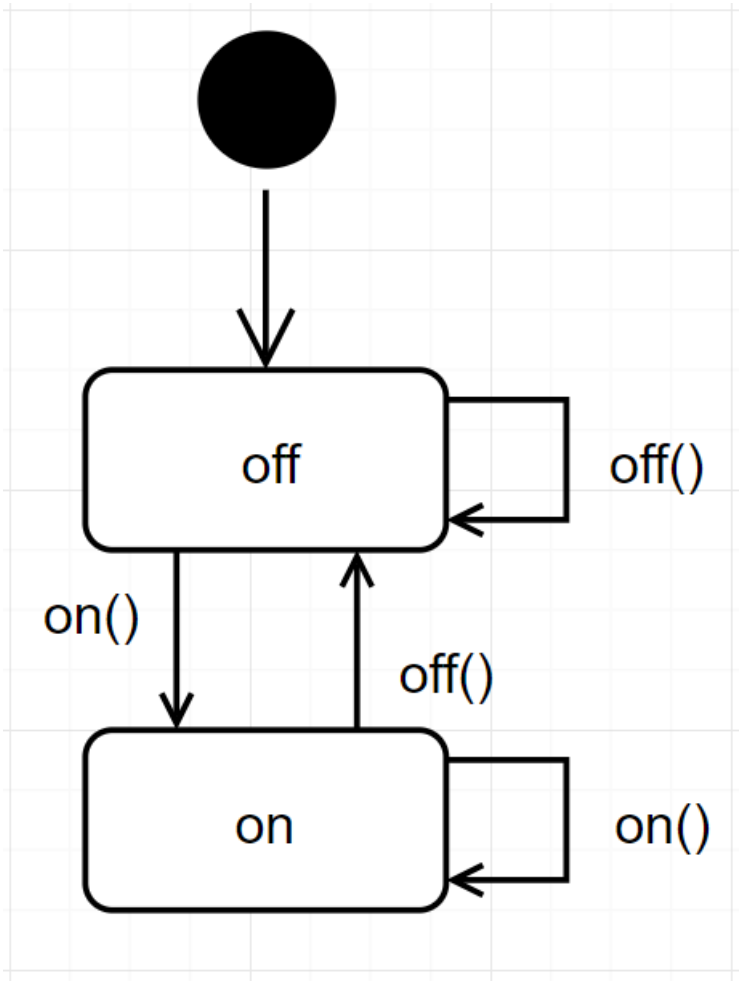
4.7.2 Format przejścia

[<zdarzenie>] [[<warunek doz.>]] [/<akcja>]

np. Urlopuj(dni)[dni<=wolnych]/wyliczDatePowrotu(dni)

Lub prostszy format będący nazwą metody, która może zmienić jego stan, np. pracuj(), przestańPracować(), zwolnijSię(), itp..

4.7.3 Przykładowy diagram stanów



Rysunek 4.28: Przykładowy diagram stanów

5 Przykładowe kody

5.1 Obiekty i skrzynki

[...] W grze, którą projektujemy będą różne rodzaje obiektów, każdy z nich będzie określany przez właściwości takie jak: waga, wartość, nazwa. Dodatkowo będą istnieć obiekty „skrzynki", w których będzie można umieścić inne obiekty, w tym inne skrzynki. [...]

```
public class Obiekt {
    private int waga;
    private int wartosc;
    private String nazwa;

    public Obiekt() {
        waga = 0;
        wartosc = 0;
        nazwa = "nic";
    }

    public Obiekt(int waga, int wartosc, String nazwa) {
        this.waga = waga;
        this.wartosc = wartosc;
        this.nazwa = nazwa;
    }

    public int getWaga() {
        return waga;
    }

    public void setWaga(int waga) {
        this.waga = waga;
    }

    public int getWartosc() {
        return wartosc;
    }

    public void setWartosc(int wartosc) {
        this.wartosc = wartosc;
    }

    public String getNazwa() {
        return nazwa;
    }

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

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

public class Skrzynka extends Obiekt {
    private List<Obiekt> obiekty;

    public Skrzynka() {
        super(0,0,"pusta skrzynka");
        obiekty = new ArrayList<>();
    }

    public Skrzynka(int waga, int wartosc, String nazwa) {
        super(waga, wartosc, nazwa);

        obiekty = new ArrayList<>();
    }

    public Skrzynka(int waga, int wartosc, String nazwa, List<Obiekt> obiekty) {
        super(waga, wartosc, nazwa);
```

```
        this.obiekty = obiekty;

        for (Obiekt obiekt: obiekty) // dla kazdego obiektu z listy
            aktualizujWartosci(obiekt);
    }

    public void dodajObiekt(Obiekt obiekt) {
        if (obiekt == this) // zeby nie mozna dodac skrzynki do siebie samej
            return;

        obiekty.add(obiekt);
        aktualizujWartosci(obiekt);
    }

    private void aktualizujWartosci(Obiekt obiekt) { // waga i wartosc skrzynki wzrasta jak si
    dodaje do niej obiekt
        super.setWaga(super.getWaga() + obiekt.getWaga());
        super.setWartosc(super.getWartosc() + obiekt.getWartosc());
    }

}
```

```
public class Main {

    public static void main(String[] args) {
        Obiekt miecz = new Obiekt(10, 800, "miecz");
        Obiekt tarcza = new Obiekt(20, 900, "tarcza");
        Obiekt kolczuga = new Obiekt(40, 1400, "kolczuga");

        Obiekt zlotaSkrzynka = new Skrzynka(5, 400, "zlota skrzynka");

        ((Skrzynka) zlotaSkrzynka).dodajObiekt(miecz);
        ((Skrzynka) zlotaSkrzynka).dodajObiekt(tarcza);
        ((Skrzynka) zlotaSkrzynka).dodajObiekt(kolczuga);

        System.out.println(zlotaSkrzynka.getWaga());
        System.out.println(zlotaSkrzynka.getWartosc());

        Skrzynka bezdennaSkrzyniaCzarnegoMaga = new Skrzynka(0, 5000, "skrzynia czarnego maga");

        bezdennaSkrzyniaCzarnegoMaga.dodajObiekt(zlotaSkrzynka);

        System.out.println(bezdennaSkrzyniaCzarnegoMaga.getWaga());
        System.out.println(bezdennaSkrzyniaCzarnegoMaga.getWartosc());
    }
}
```

5.2 Jednostki i oddziały

[...] W grze, którą projektujemy będą różne rodzaje jednostek, każda z nich będzie określana przez właściwości takie jak: nazwa, liczba punktów życia, liczba punktów many. Dodatkowo istnieć będą oddziały, w których będzie można umieścić dowolne jednostki, w tym inne oddziały. [...]

```
public class Jednostka {
    private String nazwa;
    private int punktyZycia;
    private int punktyMany;

    public Jednostka() {
        nazwa = "pusta jednostka";
        punktyZycia = 0;
        punktyMany = 0;
    }

    public Jednostka(Jednostka innaJednostka) {
        this.nazwa = innaJednostka.nazwa;
        this.punktyZycia = innaJednostka.punktyZycia;
        this.punktyMany = innaJednostka.punktyMany;
```

```
    }

    public Jednostka(String nazwa, int punktyZycia, int punktyMany) {
        this.nazwa = nazwa;
        this.punktyZycia = punktyZycia;
        this.punktyMany = punktyMany;
    }

    public String getNazwa() {
        return nazwa;
    }

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

    public int getPunktyZycia() {
        return punktyZycia;
    }

    public void setPunktyZycia(int punktyZycia) {
        this.punktyZycia = punktyZycia;
    }

    public int getPunktyMany() {
        return punktyMany;
    }

    public void setPunktyMany(int punktyMany) {
        this.punktyMany = punktyMany;
    }
}
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class Oddzial extends Jednostka {
    List<Jednostka> jednostki;

    public Oddzial() {
        super("pusty oddzial",0,0);
        jednostki = new ArrayList<>();
    }

    public Oddzial(String nazwa) {
        super(nazwa, 0,0);
        jednostki = new ArrayList<>();
    }

    public Oddzial(String nazwa, List<Jednostka> jednostki) {
        super(nazwa,0,0);
        this.jednostki = jednostki;

        for (Jednostka jednostka: jednostki)
            aktualizujInformacje(jednostka);
    }

    public void dodajJednostke(Jednostka jednostka) {
        if (jednostka == this) // zeby nie mozna oddzialu do samego siebie
            return;

        jednostki.add(jednostka);
        aktualizujInformacje(jednostka);
    }

    private void aktualizujInformacje(Jednostka jednostka) {
        super.setPunktyMany(super.getPunktyMany() + jednostka.getPunktyMany());
        super.setPunktyZycia(super.getPunktyZycia() + jednostka.getPunktyZycia());
    }
```

```
}

public class Main {

    public static void main(String[] args) {
        Jednostka magicznyHusarz1 = new Jednostka("magicznyHusarz1", 100, 50);
        Jednostka magicznyHusarz2 = new Jednostka("magicznyHusarz2", 200, 100);
        Jednostka magicznyHusarz3 = new Jednostka("magicznyHusarz3", 300, 150);

        Jednostka oddzialMagicznychHusarzy = new Oddzial("oddzialMagicznychHusarzy");

        System.out.println("Przed dodaniem: ");
        System.out.println("nazwa: " + oddzialMagicznychHusarzy.getNazwa());
        System.out.println("pkt. many: " + oddzialMagicznychHusarzy.getPunktyMany());
        System.out.println("pkt. zycia: " + oddzialMagicznychHusarzy.getPunktyZycia());

        ((Oddzial) oddzialMagicznychHusarzy).dodajJednostke(magicznyHusarz1);
        ((Oddzial) oddzialMagicznychHusarzy).dodajJednostke(magicznyHusarz2);
        ((Oddzial) oddzialMagicznychHusarzy).dodajJednostke(magicznyHusarz3);

        System.out.println("\nPo dodaniu magicznych husarzy: ");
        System.out.println("nazwa: " + oddzialMagicznychHusarzy.getNazwa());
        System.out.println("pkt. many: " + oddzialMagicznychHusarzy.getPunktyMany());
        System.out.println("pkt. zycia: " + oddzialMagicznychHusarzy.getPunktyZycia());
    }
}
```

Spis rysunków

Spis tabel

Źródła

[1] algorytm.org/wzorce-projektowe. algorytm.org/wzorce-projektowe/.

[2] altcontroldelete.pl/tag/wzorce/. altcontroldelete.pl/tag/wzorce/7.

[3] lukasz-socha.pl. lukasz-socha.pl/php/wzorce-projektowe-spis-tresci/.

[4] obliczeniowo.com.pl. obliczeniowo.com.pl/827.

[5] Psk - projektowanie systemów komputerowych. zasoby.open.agh.edu.pl/ 09sbfraczek/.

[6] C. S. Horstmann. *Czysty kod. Podręcznik dobrego programisty*. Helion, 2016-09-26.

[7] R. C. Martinn. *Java. Podstawy. Wydanie X*. Helion, 2010-02-19.

[8] S. Roth. *Czysty kod w C++17. Oprogramowanie łatwe w utrzymaniu*. Helion, 2018-07-20.