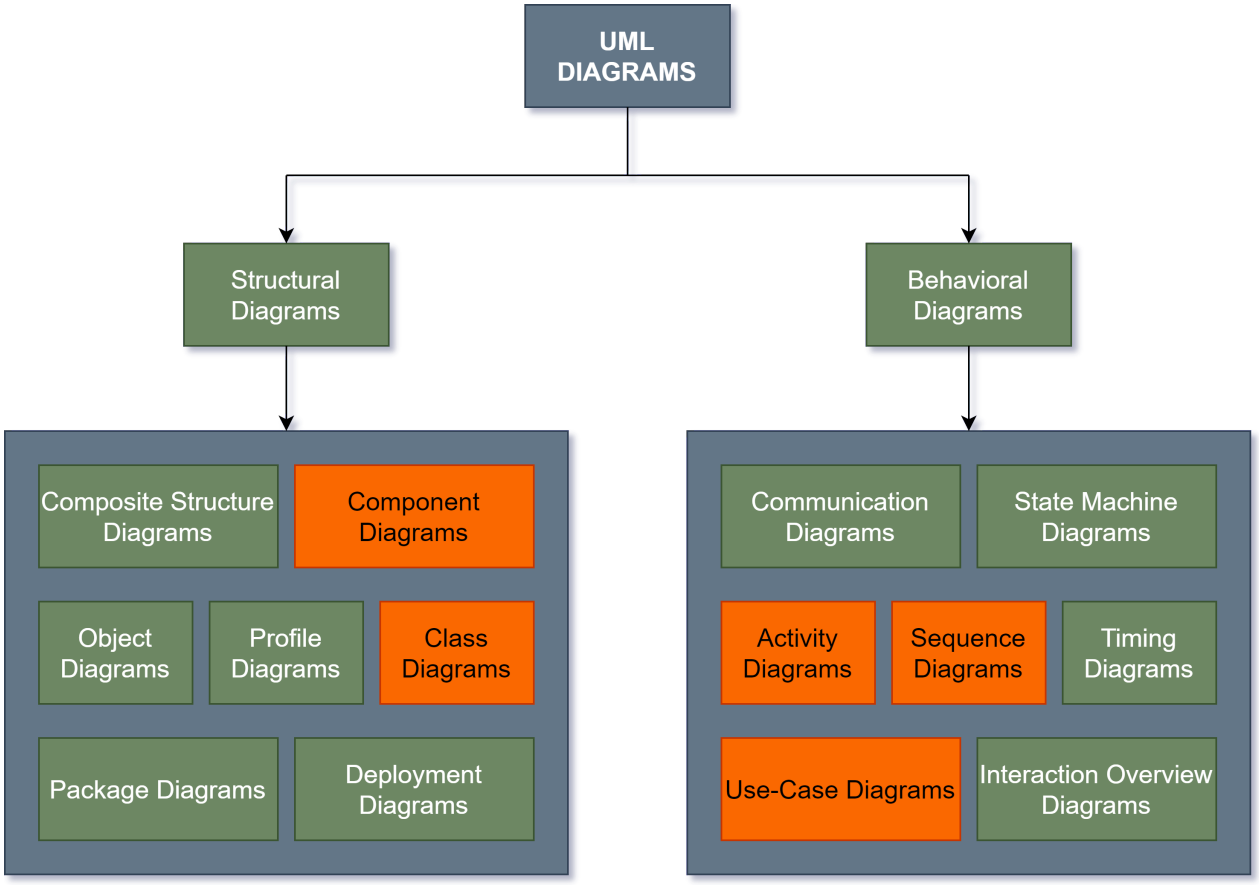
UML

Spis treści

Praktyczne przykłady

Zacznijmy od tego, że w możliwe, że praktyce nie zobaczysz większości z wymienionych niżej diagramów. Praca na dużej ilości diagramów jest często domeną bardzo dużych firm produkujących oprogramowanie. Prawdopodobne jest, że będziesz pracować w projekcie, gdzie nie będzie praktykowało się tworzenia obszernych dokumentacji, uzasadniając to tym, że kod jest najlepszą dokumentacją (bo jest najbardziej aktualny 😄), a dokumentację zawsze trzeba aktualizować w jakiś sposób.

Dlatego też pokażemy tylko kilka z wymienionych diagramów, te które zostaną omówione zaznaczono na grafice poniżej. Skupimy się tylko na typach diagramów, które zostały na grafice oznaczone kolorem pomarańczowym.



Obraz 1. UML Rodzaje Diagramów

Diagram Klas i Interfejsów

Klasa

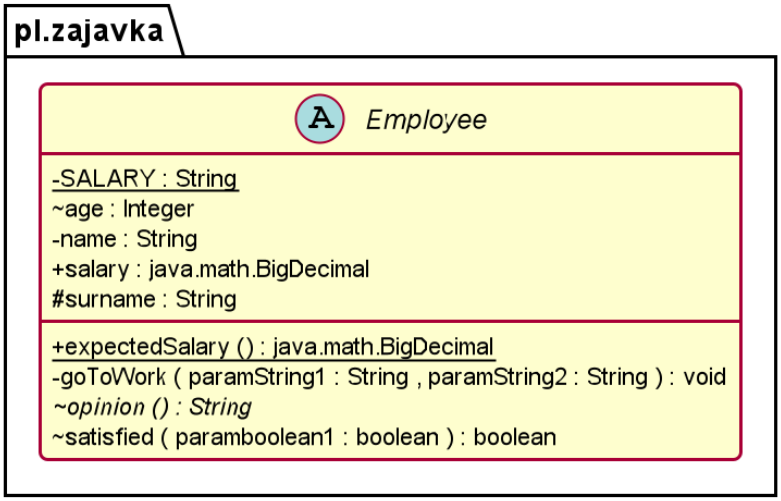
Wprowadzam ten diagram jako pierwszy żeby zaznaczyć, że chyba jest to najczęściej używany diagram podczas wizualizacji struktury naszego kodu. Służy do pokazania struktury klas i zależności między nimi. Pozwala na graficzny opis klasy z uwzględnieniem jej pól i metod, dokładając do tego modyfikatory dostępu. Teoretycznie na podstawie takiego diagramu możemy wygenerować strukturę naszego kodu, tzn. klasy z polami i metodami, ale bez implementacji metod. W praktyce, częściej widziałem próby generowania takiego diagramu z kodu niż odwrotnie. Poniżej umieszczam kod na podstawie którego stworzymy diagram klasy **UML**. Poniższy przykład będzie oczywiście zawierał różne dziwne kombinacje pól i metod aby pokazać różne aspekty diagramu **UML** dla takiej klasy.

Klasa Employee

```
package pl.zajavka;

import java.math.BigDecimal;

public abstract class Employee {

    private static final String SALARY = "5999";

    public Employee(final Integer age, final String surname) {
        this.age = age;
        this.surname = surname;
```

```
    }

    private String name;

    Integer age;

    protected String surname;

    public BigDecimal salary;

    abstract String opinion();

    private void goToWork(String name, String surname) {

    }

    boolean satisfied(boolean force) {
        return force;
    }

    public static BigDecimal expectedSalary() {
        return BigDecimal.valueOf(6900.20);
    }

}
```

Oraz diagram klasy `Employee`.



Obraz 2. UML Diagram klasy Employee

Zwróć teraz uwagę na następujące kwestie.

Klasa umieszczona jest w paczce `pl.zajavka`. Klasa reprezentowana jest przez prostokąt podzielony na kilka części. W pierwszej z nich znajduje się nazwa klasy. Może się tam również znaleźć informacja czy klasa jest abstrakcyjna, stąd tutaj mamy literkę **A**. Jeżeli klasa nie jest abstrakcyjna tylko konkretna - będziemy tutaj mieli literkę **C** i napis nie byłby napisany kursywą. Następna sekcja to pola w klasie. Ostatnia sekcja to metody w klasie.

Elementy, które są podkreślone oznaczają pola lub metody statyczne. Dlatego właśnie `SALARY` oraz `expectedSalary()` są podkreślone - bo są statyczne.

Elementy pisane kursywą są abstrakcyjne - dlatego metoda `opinion()` jest pisana kursywą. Zwróć uwagę, że nazwa klasy też jest kursywą - bo klasa jest abstrakcyjna.

Każde z pól lub metod jest poprzedzona jakimś symbolem, oznaczają one kolejno:

- (-) - pole lub metoda prywatna,
- (~) - pole lub metoda package-private,
- (#) - pole lub metoda protected,
- (+) - pole lub metoda public.

Teraz skupmy się na formacie zapisu. W przypadku pól, po symbolach modyfikatora dostępu pojawia się nazwa pola, a po dwukropku jest typ. W przypadku metod pojawia się nazwa metody oraz nawiasy sugerujące, że jest to metoda. W nawiasach pojawiają się parametry w konwencji przyjętej przy określaniu pól klasy, czyli `nazwaZmiennej:typ`. Z racji, że w metodach parametry mogą mieć dowolne nazwy, na diagramie jest to nazwane inaczej niż w kodzie. Za nawiasami sugerującymi, że jest to metoda umieszczony jest dwukropek i typ zwracany z metody.

Zauważ, że na diagramie nie ma konstruktorów oraz, że nie ma oznaczeń, że pole jest `final`.

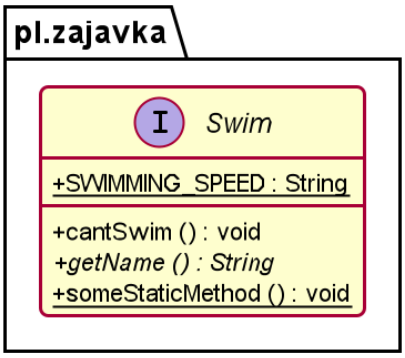
Interface

Na początek napiszmy przykład z kodem:

Interface Swim

```
package pl.zajavka;

public interface Swim {

    String SWIMMING_SPEED = "50";

    default void cantSwim() {
        // ciało metody
    }

    String getName();

    static void someStaticMethod() {
        // ciało metody
    }
}
```

Następnie diagram, który przedstawia powyższy interface.



Obraz 3. UML Diagram interfejsu Swim

Interface jest oznaczany w podobny sposób. Zwróć uwagę, na literkę **I** przy nazwie interfejsu. Nazwa jest pisana kursywą - można to rozumieć w ten sposób, że Interface jest abstrakcyjny, nie możemy utworzyć jego instancji. Podobnie jak w przypadku klasy, mamy podział na 3 sekcje - nazwa interfejsu, pola i metody. Oznaczenia są takie same jak w przypadku klas.

Zwróć uwagę, że bloki reprezentujące klasę czy interfejs wyglądają inaczej niż wcześniej, gdzie pokazywane były przykłady diagramu klasy czy diagramu interfejsu (wcześniej były takie białe). Wynika to z tego, że każde narzędzie do generowania diagramów **UML** przyjmuje trochę inny wygląd oraz oznaczenia. Wspomniałem dlatego, że nie należy się uczyć specyfikacji na pamięć, a co za tym idzie konkretnych znaczków na pamięć, bo nawet pomiędzy różnymi narzędziami mamy różnice w oznaczeniach. Przykładowo zamiast +, -, #, czy ~ możemy mieć kwadraciki, trójkąciki i romby w różnych kolorach.

Jeżeli wiemy już w jaki sposób można opisać klasę i interface, możemy przejść do przedstawienia relacji.

Relacje

Przykładem relacji między klasami jest dziedziczenie lub kompozycja. Relacje są reprezentowane przez symbole strzałek, które zostały pokazane wcześniej, przy czym każda strzałka oznacza inny rodzaj relacji. Aby pokazać przykłady relacji i przejść do ich omówienia, posłużymy się przykładem pokazanym poniżej.

Z racji, że już na tym etapie wiemy już (a przynajmniej powinniśmy) jak w kodzie Java wygląda dziedziczenie i implementacja interfejsu, nie będziemy przytaczać tych przykładów z kodem. Pokażemy natomiast przykłady w kodzie dotyczące **Aggregation** oraz **Composition**.

Prosty przykład

Klasa Human

```
public class Human {

    private String name;

    private Heart heart;

    private List<Dog> dogs;

    public Human(final String name, final List<Dog> dogs) {
        this.name = name;
        this.heart = new Heart();
        this.dogs = dogs;
    }

}
```

W przykładzie powyżej, serce (Heart) nie może istnieć bez człowieka (Human). Pies (Dog) natomiast może. Jednocześnie pies jest tutaj zapisany w formie relacji jeden do wielu, tzn. jeden właściciel może mieć wiele psów.

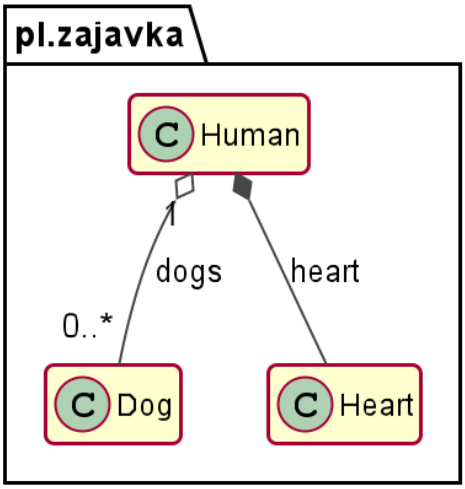
Klasa Heart

```
public class Heart {

    void pump() {
        // ciało metody
    }
}
```

Klasa Dog

```
public class Dog {}
```

Diagram **UML**, który powstanie na podstawie powyższego kodu może wyglądać w ten sposób:



Obraz 4. UML Aggregation i Composition

Zauważ, że tym razem diagram nie ma wypisanych pól i metod, wynika to z tego, że chcemy położyć nacisk na relację pomiędzy encjami. Na jednym diagramie umieściliśmy jednocześnie **Aggregation** oraz **Composition**. Oznaczenia `1` oraz `0..*` oznaczają, że jeden właściciel, może mieć zero lub wiele psów. O liczebności relacji powiemy później.

Oczywiście moglibyśmy również cały czas stosować oznaczenie w postaci **Association**, gdyż jest ono generalizacją **Agregacji** i **Kompozycji**.

Troszeczkę bardziej złożony przykład

Zacznijmy od napisania kodu, który będzie zawierał klasy "wydmuszki", klasy bez żadnej logiki.

```
public class Customer {

    private String name;
    private String surname;

    private ShoppingCart shoppingCart;
}
```

```
public class DiscountItem implements Item {}
```

```
public interface Item {}
```

```
public class NormalCustomer extends Customer {}
```

```
public class NormalItem implements Item {}
```

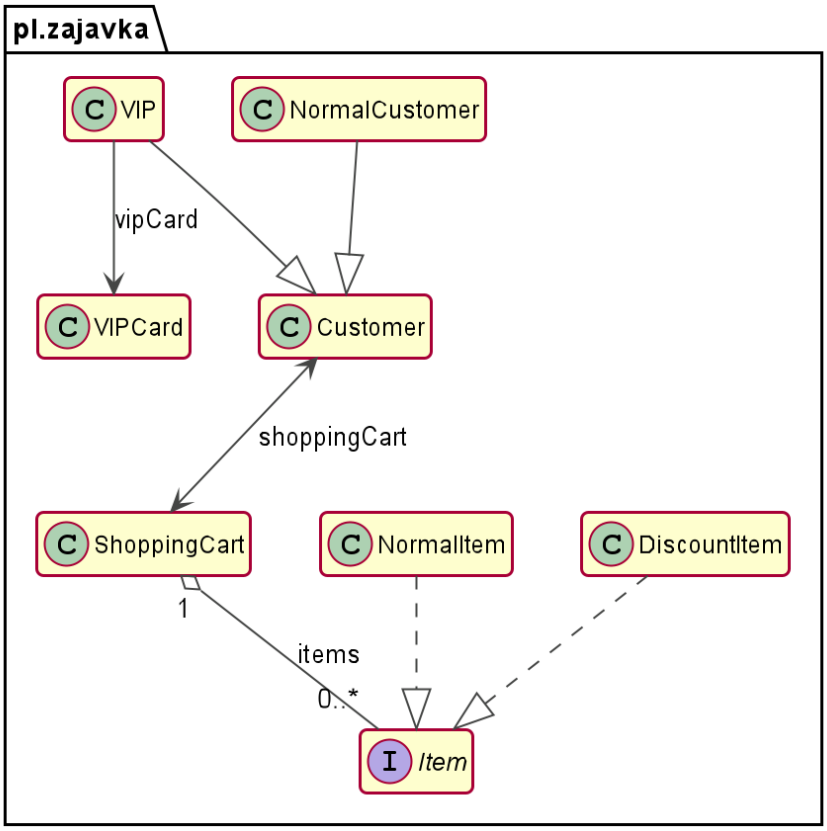
```
public class ShoppingCart {

    private Customer customer;
```

```
    private List<Item> items;
}
```

```
public class VIP extends Customer {

    private VIPCard vipCard;
}
```

```
public class VIPCard {}
```

Jeżeli teraz na podstawie pokazanego wyżej kodu spróbujemy narysować diagram relacji, będzie on wyglądał np. w ten sposób:



Obraz 5. UML Relations

- Klasy `NormalItem` oraz `DiscountItem` implementują interface `Item` - implementacja interfejsu,
- Klasy `VIP` oraz `NormalCustomer` rozszerzają klasę `Customer` - dziedziczenie,
- Klasa `VIPCard` jest zależna od klasy `VIP`, inaczej mówiąc klasa `VIP` korzysta z klasy `VIPCard` - asocjacja. Równie dobrze moglibyśmy w tym miejscu pokazać kompozycję, czyli wypełniony romb na końcu,
- Klasa `ShoppingCart` zawiera 0 lub więcej obiektów `Item` - agregacja.
- Klasa `Customer` zawiera klasę `ShoppingCart` - asocjacja, jednocześnie klasa `ShoppingCart` również wie o klasie `Customer`, możemy albo dać wtedy strzałkę w obu końcach linii, albo nie dawać jej wcale.

Jeżeli mówimy o asocjacji (np. Klasa **VIP** ma zdefiniowane pole **VIPCard**), to zwróć uwagę, że strzałka

jest przy klasie `VIPCard`. Jeżeli natomiast mówimy o agregacji, czyli `ShoppingCart` agreguje `Item`, to zobacz, że romb jest przy właścicielu tej relacji. Tak samo będzie w przypadku kompozycji.

Liczebność

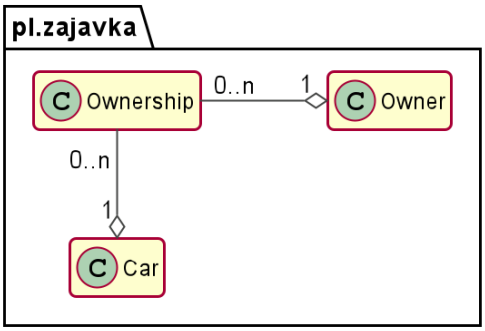
Widzisz, że na diagramie pojawia się oznaczenie `1` oraz `0..*`. Zapis taki oznacza liczebność w relacji, czyli ile encji może wystąpić po każdej ze stron. Możemy wyróżnić liczebności takie jak:

- 0..* - oznacza to, że obiekt klasy A może mieć wiele obiektów klasy B,
- 0..1 - oznacza to, że obiekt klasy A może mieć zero lub jeden obiektów klasy B,
- 1..* - oznacza to, że obiekt klasy A może mieć jeden lub wiele obiektów klasy B,
- 1 - oznacza to, że obiekt klasy A może mieć dokładnie jeden obiekt klasy B.

W związku z liczebnościami, mamy też typy relacji, które są wymienione poniżej:

- Jeden do jednego (**one-to-one**) - Jedno mieszkanie może mieć jedną kuchnię, wtedy po obu stronach linii możemy napisać `1`, ale najczyściej w takim przypadku zwyczajnie się tego nie pisze 😄,
- Jeden do wielu (**one-to-many**) - Jedno mieszkanie może mieć wiele łazienek, wtedy po jednej stronie linii możemy napisać `1`, a po drugiej `0..*`,
- Wiele do wielu (**many-to-many**) - Jeden właściciel może mieć wiele mieszkań, ale jedno mieszkanie może też mieć wielu właścicieli, wtedy po obu stronach możemy napisać `0..n` lub `1..n`.

W momencie gdy będziemy rozmawiać o sytuacjach wiele do wielu, w praktyce wprowadza się klasę, która sama w sobie oznacza taką relację. Czyli możemy mieć np. `Samochód` i `Właściciela`. Samochód może mieć wielu właścicieli, a właściciel może mieć wiele samochodów. W takim przypadku możemy wprowadzić klasę `Ownership`. Jak taka sytuacja może być przedstawiona na diagramie **UML**?



Obraz 6. UML relacja many-to-many

W powyższym przypadku zarówno `Owner` jak i `Car` będą przechowywały kolekcję swoich `Ownership`, czyli kolekcję swoich właścicielstw 😉.

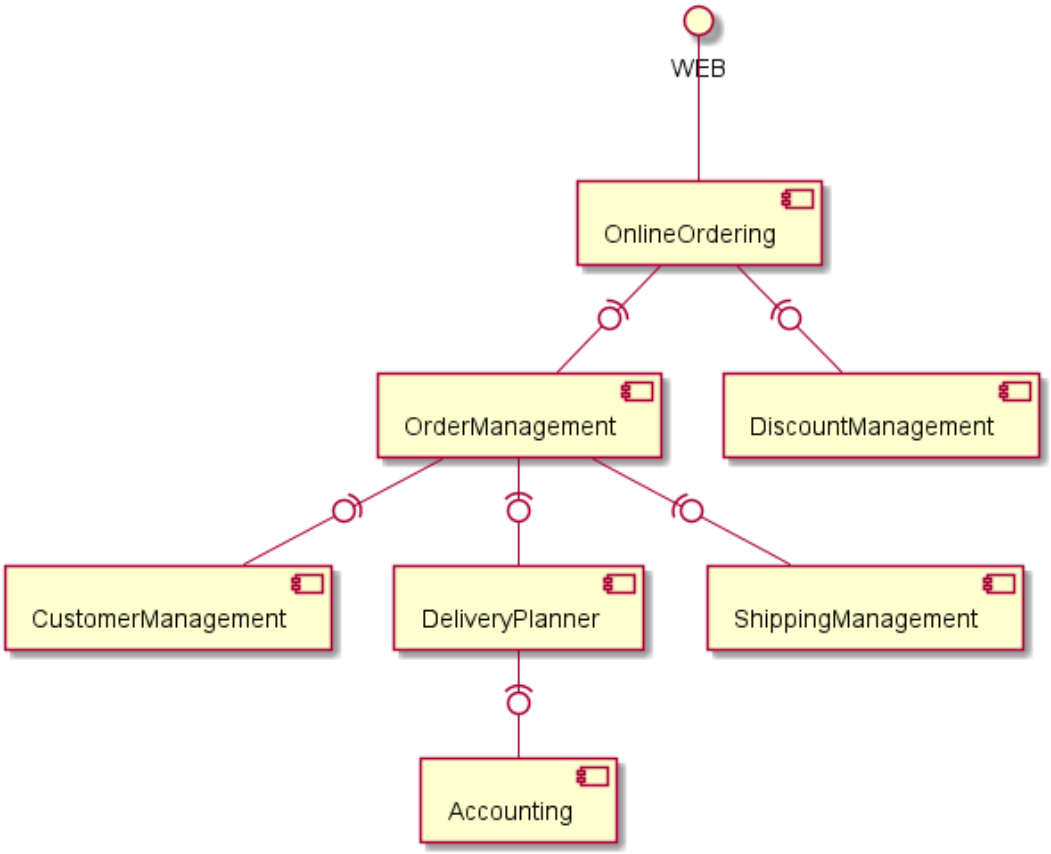
Diagram Komponentów

Wiemy już, że Diagram Klas pozwala nam zobaczyć projekt z bliska, z poziomu klas i interfejsów. Jednakże jest jeszcze diagram, który pozwala spojrzeć na projekt z dalszej odległości. Jeżeli projekt jest podzielony na moduły (komponenty), to możemy narysować diagram komponentów. Przykładowy symbol komponentu był pokazywany przy omawianiu bloków składowych **UML**.

Komponent można rozumieć jako podprogram naszego programu. Czyli cały realizowany program (albo projekt) składa się z wielu podprogramów albo podprojektów, które mogą się ze sobą komunikować (w jaki sposób dowiemy się później, na razie wystarczy nam tyle, że mogą ☺). Sam program również może składać się z modułów, ale nie poruszamy tego tematu. Komponenty mają też to do siebie, że można je w całości wymienić.

Komponent wystawia i konsumuje interfejsy, to dzięki nim jest w stanie komunikować się z innymi komponentami. Przez to, że komponenty komunikują się między sobą przy pomocy intefejsów, żaden z nich nie zastanawia się co i jak inne mają w środku. Jeżeli interfejsy są zgodne to można taki komponent w całości wymienić. Byle spełnione było to zachowanie w środku.

Poniżej możesz znaleźć przykładowy (oczywiście uproszczony) diagram komponentów przykładowego sklepu internetowego.



Obraz 7. UML Diagram Komponentów

To my rysując diagram decydujemy na jakim poziomie rozumiemy komponenty i na jakim poziomie będziemy je rysować. Możemy równie dobrze przyjąć, że paczka w projekcie jest komponentem i

chcemy narysować jak komunikuje się ona z innymi paczkami.

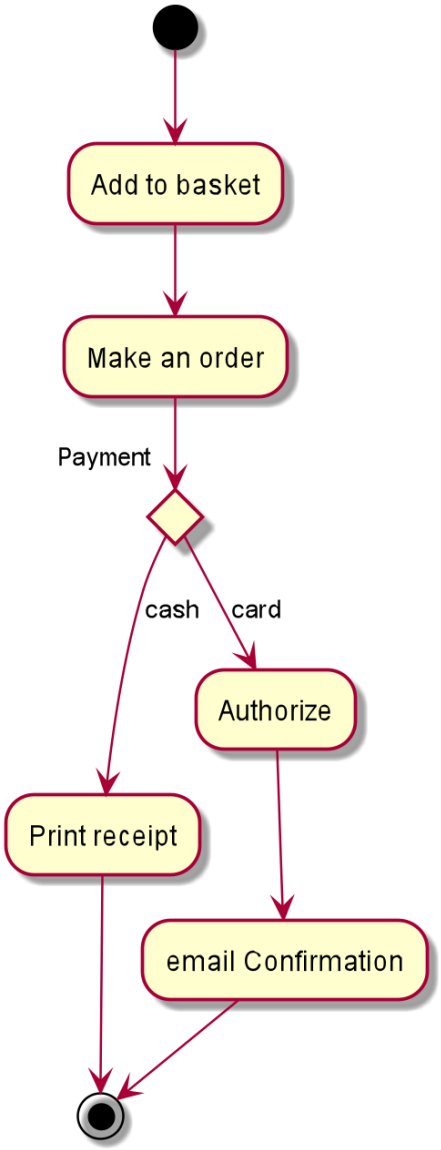
Oprócz samego bloku komponentu, który został pokazany wcześniej pojawią się też na takim diagramie łączniki. Tak samo jak mieliśmy linie i strzałki w poprzednich diagramach, natomiast tutaj są one lekko inne. Komponent aby mógł efektywnie funkcjonować, musi wystawiać interface albo sam go konsumować (po co komu komponent, z którym nie można się porozumieć). Wystawiane interfejsy oznaczane są jako kreska z okręgiem na końcu. Konsumowane interfejsy (czyli te, których komponent wymaga) są oznaczone jako kreska z łukiem.

Diagram Aktywności

Activity Diagram jest diagramem, który służy do abstrakcyjnego zobrazowania kroków w procesie, który opisujemy. Wcześniejsze diagramy skupiały się na statycznej konstrukcji systemu, na jego modelowaniu. Ten diagram skupia się na przedstawieniu zachowania/procesu jaki jest realizowany w ramach systemu informatycznego bądź aplikacji.

W ramach diagramu aktywności przedstawiamy kolejne kroki w procesie, zaznaczając gdzie jest początek takiego procesu, a gdzie jego koniec. W trakcie trwania procesu możemy rozdzielać jego przebieg na gałęzie, w zależności od tego czy jakieś warunki są spełnione, czy też nie. Do tego służył nam znaczek rombu.

Spójrzmy zatem na przykładowy diagram:



Obraz 8. UML Activity Diagram

W przedstawionym przykładzie mamy opisany uproszczony proces składania zamówienia w sklepie. Możemy wybrać, czy płatność w sklepie ma zostać zrealizowana przy wykorzystaniu gotówki czy karty. Jeżeli wybierzemy gotówkę, to kolejnym krokiem jest wydrukowanie paragonu. Jeżeli natomiast zdecydujemy się na kartę, to musimy dokonać autoryzacji (czyli wpisać pin) i dostaniemy wtedy powiadomienie mailem (bo tak ktoś wymyślił 😀). Cały proces kończy się w tym samym miejscu.

Na podstawie przedstawionego diagramu widać, że służy on do opisania kroków w procesie w sposób abstrakcyjny. Daje też nam możliwość rozgałęzienia procesu na zasadzie **if-else**. Diagram ten może być użyty jeżeli chcemy pokazać jak działa proces nie powołując się w żaden sposób na fragmenty aplikacji, fragmenty kodu albo moduły systemu. Oczywiście można dodać tutaj notatki mówiące, co jest za co odpowiedzialne, ale w bardzo prostym wariancie, opisujemy abstrakcyjne kroki procesu i jego zachowanie.

Kolejny diagram jest bardziej złożony i istnieje pewne prawdopodobieństwo, że to właśnie jego będziesz oglądać w praktyce częściej.

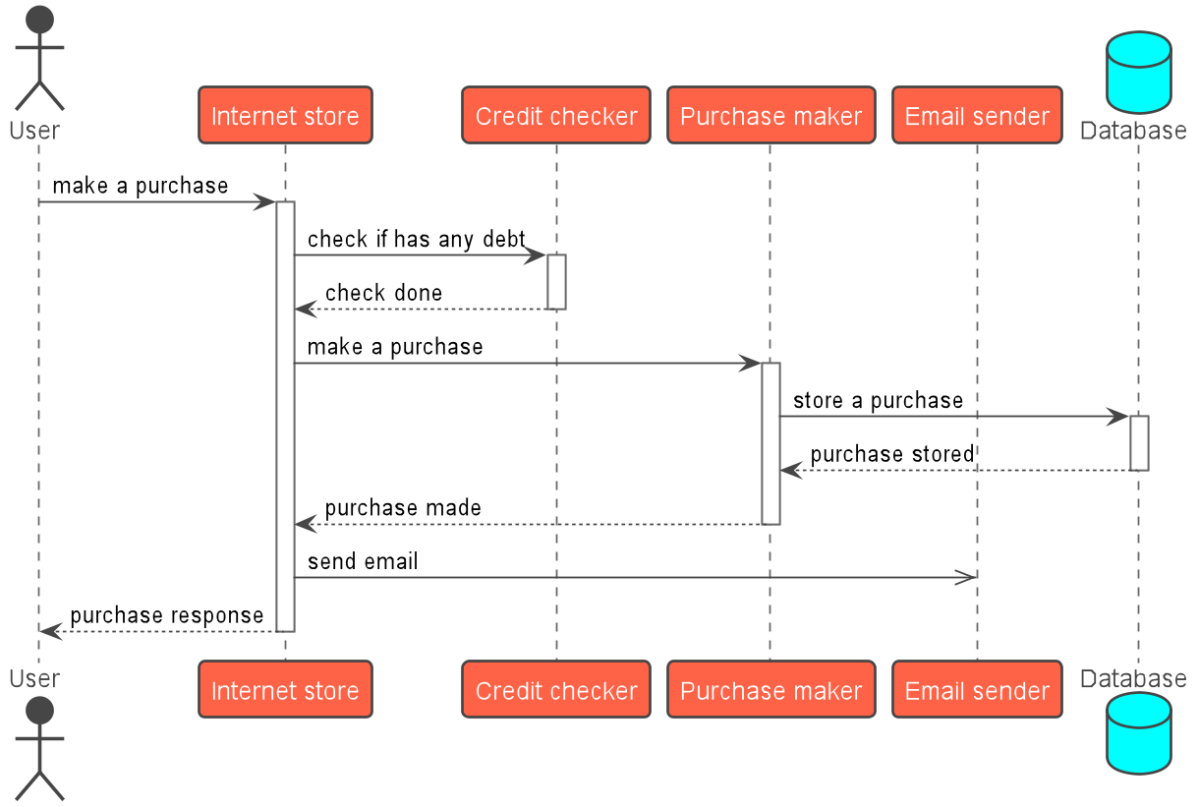
Diagram Sekwencji

Sequence Diagram jest również często używanym diagramem. W praktyce może być on często używany gdy chcemy zobrazować interakcję pomiędzy wieloma systemami, pokazując kiedy i w jakiej sekwencji systemy komunikują się ze sobą. Diagram taki może być również zastosowany do interfejsów, gdzie będziemy pokazywali sekwencję wykonania kolejnych metod. Diagram ten jest bardzo przydatny gdy chcemy pokazać sekwencję komunikacji.

Wcześniej zostało już pokazane, że na diagramie sekwencji możemy wyróżnić taki element jak **Lifeline**, czyli linię życia. Na niej możemy pokazać czas życia interfejsu (albo obiektów), jeżeli obrazujemy komunikację między interfejsami (albo obiektami). Możemy na tej linii pokazać również czas trwania interakcji między systemami (tutaj nie piszę czas życia, bo system nie kończy swojej pracy po zakończeniu komunikacji, tylko pracuje dalej). Na diagramie takim możemy też często zobaczyć aktora, czyli osobę lub inny system, który inicjuje przedstawianą sekwencję.

Prostokąty na linii życia oznaczają czas podczas którego dany obiekt lub aktor był aktywny, czyli kiedy wykonywana była przez aktora lub obiekt jakaś operacja. W tym czasie mógł też oczekiwać na wynik innej operacji.

Spójrzmy zatem na przykładowy diagram:



Obraz 9. UML Sequence Diagram

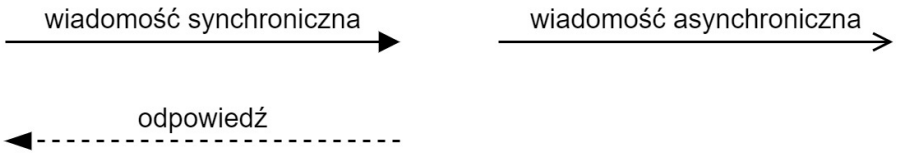
Diagram może służyć jako bardzo uproszczony schemat sekwencji zakupu produktu w sklepie internetowym. Początkowo Użytkownik dokonuje zakupu w sklepie internetowym. Następnie usługa (`Internet Store`), która jest odpowiedzialna za złożenie takiego zamówienia komunikuje się z usługą sprawdzającą (`Credit Checker`), czy klient nie ma ukrytych długów. Po uzyskaniu odpowiedzi usługa `Internet Store` składa zamówienie w usłudze robiącej faktyczny zakup (`Purchase Maker`), która musi zapisać takie zamówienie w bazie danych. Gdy to się udaje, zostaje wysłany email z wiadomością po

pomyślnym zakupie, a klient widzi odpowiedź w przeglądarce, że zakup się powiódł. Widzisz już też na pewno, że im wyżej coś jest na diagramie, oznacza to, że wydarzyło się wcześniej.

Pokazane zostały 3 rodzaje komunikatów:

- linia ciągła z grotem wypełnionym - wiadomość wysłana **synchronicznie**, inaczej mówiąc, wiadomość, w której czekamy na odpowiedź,
- linia przerywana z grotem wypełnionym - odpowiedź na wiadomość wysłaną synchronicznie. Czekamy na odpowiedź, a ten symbol obrazuje wysłanie odpowiedzi,
- linia ciągła z pustym grotem - wiadomość wysłana **asynchronicznie**, inaczej mówiąc, wysyłamy wiadomość i nie interesuje nas odpowiedź. Inaczej nazywa się to "fire and forget".

Poniżej narysujmy te strzałki ponownie:



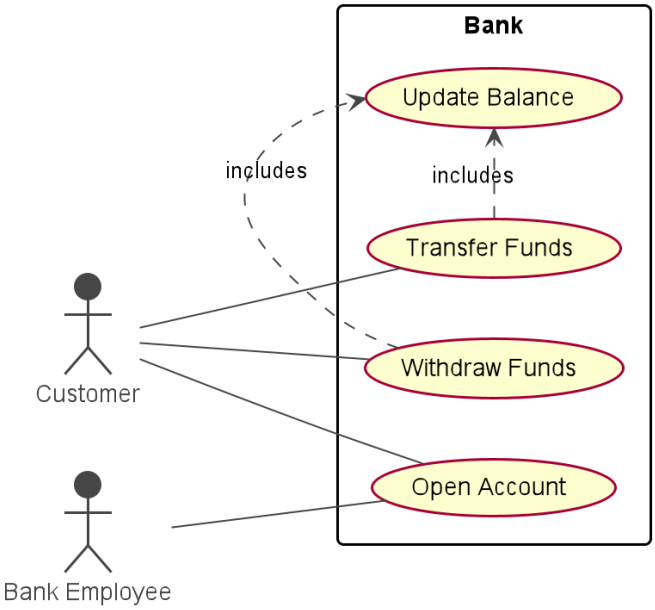
Obraz 10. UML Rodzaje wiadomości

Zwróciłeś/zwróciłaś uwagę, że grot strzałki, która reprezentuje asocjację na diagramie klas nie jest identyczny z grotem strzałki w notatkach (inaczej wygląda wypełnienie tych grotów)? Jeżeli jednocześnie zwróciłeś/zwróciłaś uwagę na to, że w przypadku wiadomości asynchronicznej grot strzałki na diagramie wygląda już tak samo jak grot strzałki pokazanej w notatkach to z jednej strony gratuluję szczegółowości (w tej pracy się przyda), a z drugiej strony pamiętaj, że wszystkie te diagramy są generowane przez pewne narzędzia. Nawet jeżeli pokazałem w notatkach, że strzałka powinna wyglądać w jakiś sposób, to zdarza się, że narzędzia robią różne rzeczy po swojemu i tutaj mamy tego przykład. Ale żeby podtrzymać swoją wiarygodność to zobacz co się kryje pod tym linkiem Wikipedia.

Diagram Przypadków Użycia

Use-Case diagram służy do przedstawienia interakcji pomiędzy systemem, a jego użytkownikami (aktorami). Jest to bardzo wysokopoziomowy diagram, który pomaga lepiej zrozumieć wymagania jakie powinien spełniać system informatyczny. Diagram ten opisuje co potrafi zrobić system, oraz którzy aktorzy są w stanie korzystać z konkretnych fragmentów systemu. Diagram taki często powstaje na bardzo wczesnych etapach projektu aby pomóc zwizualizować co tak na prawdę użytkownicy mają w tym systemie robić, jakich operacji mają dokonać.

Aktor jest reprezentacją roli w takim systemie. Aktorem może być typowy użytkownik - człowiek, ale również inny system informatyczny.



Obraz 11. UML Use Case Diagram

Na powyższym diagramie widać, że mamy 2 aktorów, `Użytkownik` i `Pracownik Banku`. Aby Użytkownik mógł otworzyć Konto, musi przy tym uczestniczyć Pracownik Banku. Użytkownik może wykonać transfer swoich środków oraz je wypłacić. Obie z tych operacji wiążą się z aktualizacją stanu konta.

Podsumowanie

Możesz już powiedzieć, że znasz podstawy **UML**. Mogą Ci się one bardzo przydać w codziennej pracy. Jeżeli trafisz do środowiska bardziej formalnego, może się okazać, że będziesz z tych diagramów korzystać dosyć często. A jeżeli będzie to środowisko mniej formalne, **UML** bardzo często będzie Ci się przydawał w momencie gdy zajdzie potrzeba wyrażenia swoich pomysłów rysując ☺.