Podstawowe pojęcia obiektowości:

- **Metoda** - Procedura, funkcja lub operacja przypisana do klasy obiektów i dziedziczona przez jej podklasy. Metoda jest abstrakcją programistyczną tej samej kategorii co procedura lub procedura funkcyjna. Metoda różni się od procedury tym, że działa w środowisku obiektu (wykorzystując jego wewnętrzne informacje, przede wszystkim wartości atrybutów) po wysłaniu do niego komunikatu zawierającego jej nazwę. Zakłada się, że (podobnie do wołania procedury) komunikat składa się z nazwy metody oraz parametrów. Z koncepcyjnego punktu widzenia miejscem przechowywania metody jest odpowiednia klasa; oznacza to, że taka metoda może być zastosowana do dowolnego obiektu będącego (bezpośrednim lub pośrednim) wystąpieniem tej klasy. Można również rozważać metody przechowywane wewnątrz samych obiektów i wywoływane na podobnych zasadach. Zwykle pojęciu metody przypisuje się niezbyt duży stopień skomplikowania ("mała procedura"), nie angażowanie zbyt dużych zasobów z zewnątrz klasy oraz brak efektów ubocznych na środowisku spoza wnętrza obiektu. Nie są to jednak formalne ograniczenia, lecz raczej zalecenia dla programistów, które z różnych względów nie zawsze muszą i nie zawsze mogą być przestrzegane.

- **Klasa** - Pojęcie klasy jest używane w trzech dość bliskich znaczeniach:
  - Zbiór obiektów o zbliżonych własnościach (niezbyt ściśle - patrz: ekstensja klasy); to znaczenie jest (naiwnie) preferowane przez osoby o orientacji matematycznej, którzy mylnie kojarzą pojęcie klasy z zakresem znaczeniowym (denotacją) pewnego pojęcia/nazwy, lub z klasą abstrakcji pewnej relacji równoważności.
  - Byt semantyczny rozumiany jako miejsce przechowywania takich cech grupy podobnych obiektów, które są dla nich niezmienne (np. zestawu atrybutów, nazwy, metod, ograniczeń dostępu).
  - Wyrażenie językowe specyfikujące budowę obiektów, dozwolone operacje na obiektach, ograniczenia dostępu, wyjątki, itd.

  Klasy są zwykle powiązane poprzez hierarchię (lub inną strukturę) dziedziczenia.

- **Abstrakcja** - Eliminacja, ukrycie lub pominięcie mniej istotnych szczegółów rozważanego przedmiotu lub mniej istotnej informacji; wyodrębnianie cech wspólnych i niezmiennych dla pewnego zbioru bytów oraz wprowadzanie pojęć lub symboli oznaczających takie cechy. Abstrakcja jest podstawową zasadą obiektowości. Oprócz klasycznych procedur, modułów i typów, abstrakcję wspomagają takie pojęcia jak klasy, dziedziczenie, metody, hermetyzacja, późne wiązanie i polimorfizm.

- **Konstruktor** - Operator lub metoda tworząca obiekty; nazwa konstruktora to nazwa klasy.

- **Interfejs** - Ogólnie, środki służące do komunikacji pomiędzy modułami systemu lub komunikacji systemu z użytkownikiem. W innym znaczeniu (OMG, ODMG) interfejs jest synonimem specyfikacji klasy. Interfejs do obiektu składa się z sygnatur wszystkich metod, które mogą być w stosunku do niego użyte. Operacje na obiekcie mogą odbywać się poprzez wysłanie do niego komunikatu zgodnego z dowolną z sygnatur należących do jego interfejsu. W niektórych koncepcjach (np. MS Repository) obiekty mogą mieć wiele różnych interfejsów, w zależności od roli pełnionej przez obiekty (patrz: rola (1)).

- **Wersyfikacja semantyczna** - określenie kolejności powstawania nowych wersji oprogramowania, pozwala na odróżnienie wersji między sobą. Zazwyczaj jest liczbą naturalną (np. numerowanie wersji od 1 lub według roku powstania), liczbą rzeczywistą lub zestawieniem kilku liczb naturalnych. W ostatnim przypadku kolejne liczby oddziela się zazwyczaj kropką, a ich znaczenie jest następujące:

  - *Major* (numer główny),
  - *Minor* (numer dodatkowy),
  - *Release* (numer wydania),
  - *czasami Build* (w przypadku tzw. nightly-build).

Poszczególne składniki są kolejno inkrementowane podczas zmian w programie. Często projekty nie używają wszystkich składników, albo nazywają je inaczej (np. jądro Linuksa).

MajorVersion.MinorVersion.Patch

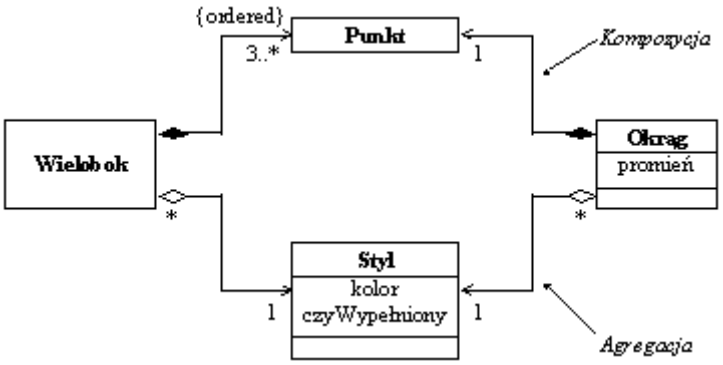
MajorVersion – Dodanie nowej funkcjonalności, API traci wsteczną kompatybilność.

MinorVersion – Dodanie nowej funkcjonalności, API jest wstecznie kompatybilne.

Patch – Drobna poprawa błędów, API się nie zmienia.

- **Agregacja** - Związek pomiędzy klasami obiektów (szczególny przypadek asocjacji), modelujący stosunek całości do jej części (np. stosunek samochodu do silnika). Obiekty są powiązane związkiem agregacji, jeżeli jeden z nich można uważać za część drugiego, zaś cykl i czas życia tych obiektów są jednakowe. Pojęcie agregacji w modelowaniu jest niejasne i rodzące mnóstwo wątpliwości semantycznych i pragmatycznych. Nie istnieje powszechnie akceptowana definicja agregacji, zaś wątpliwości co do jej znaczenia są zasadnicze. Np. Peter Coad podaje jako przykład agregacji związek pomiędzy organizacją i jej urzędnikami; dla odmiany James Rumbaugh twierdzi, że firma nie jest agregacją jej pracowników. Komu wierzyć? Wątpliwości powstają nawet w tak banalnym przypadku, jak cytowany wyżej przykład samochodu i silnika. Silnik może być towarem w sklepie nie związanym z żadnym samochodem lub może być przełożony z jednego samochodu do drugiego; wobec czego jest samodzielnym obiektem. Ponadto powstaje pytanie, czy chodzi o konkretny samochód i konkretny silnik, czy też o typ samochodu i typ silnika; w tym drugim przypadku większość analityków nie zgodziłaby się z założeniem, że silnik jest częścią samochodu. Podane przykłady pokazują powody, dla których formalna definicja agregacji grzęźnie w mętnych dywagacjach, z których trudno wyprowadzić jasne i naturalne zasady użycia tego pojęcia.
  Dodatkowy mętlik wynika z wykorzystywania pojęcia agregacji do rozwiązywania pewnych technicznych problemów związanych z ograniczeniami modelu obiektowego. Np. popularne wyjaśnienie technik obejścia braku wielodziedziczenia mówi, że można je "odwzorować przez agregację". Np., jeżeli klasa emerytowanych pracowników dziedziczy zarówno od klasy Emeryt jak i od klasy Pracownik, wówczas "odwzorowanie wielodziedziczenia przez agregację" oznacza, że obiekt emerytowanego pracownika zawiera jako swoją "część" inny obiekt grupujący informację o cechach osoby jako emeryta. Mówi się, że obiekty pracownika i emeryta pozostają w związku agregacji; emeryt "jest częścią" pracownika (sic). Tego rodzaju pseudowyjaśnienia, dodatkowo splątane z równie mętną interpretacją pojęcia "delegacji", powodują nie lada zamieszanie w głowach tych, którzy próbują dokładnie zrozumieć istotę pojęcia agregacji i jego stosowalność w konkretnych sytuacjach. Dodać należy, że żaden z istniejących obiektowych języków, systemów i standardów nie wprowadza agregacji jako wyróżnionego, odrębnego pojęcia programistycznego, odwołuje się więc ono wyłącznie do ludzkiej wyobraźni.
  Autorzy UML podejmują próbę uporządkowania pojęcia agregacji. Wprowadzili oni mocniejszą formę agregacji, nazywając ją kompozycją. Związek kompozycji oznacza, że dana część może należeć tylko do jednej całości. Taka część nie może istnieć bez całości, pojawia się i jest usuwana razem z całością. Przy takim zróżnicowaniu pojęć pozostaje jednak problem, co dokładnie oznacza "słaba" forma agregacji i jakie są pragmatyczne reguły jej użycia.

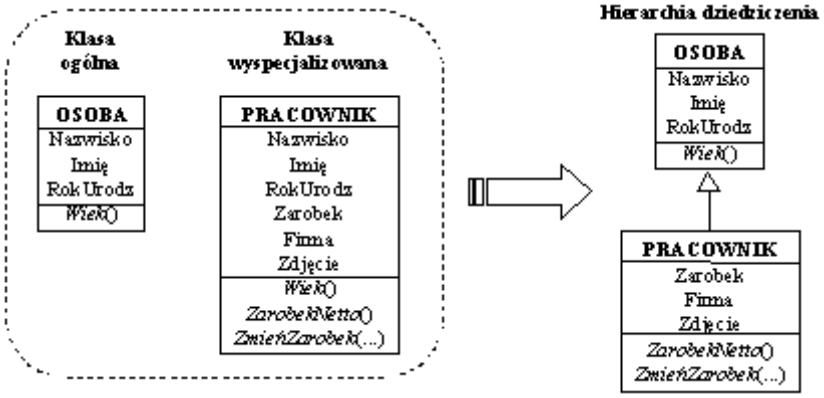
- **Kompozycja** - Mocna forma agregacji wprowadzona w UML. Związek kompozycji oznacza, że dana część może należeć tylko do jednej całości. Co więcej, część nie może istnieć bez całości, pojawia się i jest usuwana razem z całością. Klasycznym przykładem związku kompozycji jest zamówienie i pozycja zamówienia: pozycja zamówienia nie występuje oddzielnie (poza zamówieniem), nie podlega przenoszeniu od jednego zamówienia do innego zamówienia i znika w momencie kasowania zamówienia.



Przykłady kompozycji i agregacji

Powyższy rysunek ilustruje zastosowanie agregacji (pusty w środku romb) i kompozycji (zaczerniony romb). Każde wystąpienie obiektu Punkt należy albo do obiektu Wielobok, albo do obiektu Okrąg; nie może należeć do dwóch obiektów naraz. Wystąpienie obiektu Styl może być dzielone przez wiele obiektów Wielobok i Okrąg. Usunięcie obiektu Wielobok powoduje kaskadowe usunięcie wszystkich związanych z nim obiektów Punkt, natomiast nie powoduje usunięcia związanego z nim obiektu Styl.

- **Dziedziczenie** - Związek pomiędzy klasami obiektów określający przekazywanie cech (definicji atrybutów, metod, itd.) z nadklasy do jej podklas. Np. obiekt klasy Pracownik dziedziczy wszystkie własności (definicje atrybutów, metody) określone w ramach klasy Osoba. Dziedziczenie jest podstawowym mechanizmem sprzyjającym ponownemu użyciu. Istnieje wiele form dziedziczenia, wśród nich dziedziczenie oparte na klasach *(class-based inheritance)*, dziedziczenie oparte na prototypach lub delegacja, dziedziczenie dynamiczne, wielokrotne dziedziczenie, itd. Poniższy rysunek przedstawia wyodrębnienie hierarchii dziedziczenia z dwóch klas, OSOBA i PRACOWNIK.



Wyodrębnienie hierarchii dziedziczenia

- **Polimorfizm** - Termin używany w dwóch nieco różnych znaczeniach (które są często mylone): (1) w terminologii obiektowej: możliwość istnienia wielu metod o tej samej nazwie, powiązana z możliwością wyboru konkretnej metody podczas czasu wykonania (dynamicznym wiązaniem); (2) w terminologii teorii typów i języków polimorficznych (np. w ML): umożliwienie definiowania funkcji lub procedur, których argumenty i wynik mogą posiadać jednocześnie wiele typów; np. funkcji sort, która może posortować (wg pewnego algorytmu) kolekcję elementów dowolnego typu (o ile określona zostanie metoda porównania elementów). Polimorfizm parametryczny jest często kojarzony z sytuacja określaną jako przypadek typu *(typecase)*, czyli możliwością dynamicznego testowania typu danej wartości lub zmiennej. Powyższe dwa znaczenia pojęcia polimorfizmu są względem siebie ortogonalne: możliwy jest polimorfizm metod w języku nietypowanym (np. Smalltalk), oraz polimorfizm typów w języku nie wprowadzającym wiązania dynamicznego (np. ML).

- **Hermetyzacja** - Zamknięcie pewnego zestawu bytów programistycznych w "kapsułę" o dobrze określonych granicach; oddzielenie abstrakcyjnej specyfikacji tej kapsuły (obiektu, klasy, modułu, etc.) od jej implementacji; ukrycie części informacji zawartej w tej kapsule dla operacji z zewnątrz obiektu. Hermetyzacja jest podstawową techniką abstrakcji, tj. ukrycia wszelkich szczegółów danego przedmiotu lub bytu programistycznego, które na

danym etapie rozpatrywania (analizy, projektowania, programowania) nie stanowią jego istotnej charakterystyki. Hermetyzacja jest starą zasadą inżynierii oprogramowania (sformułował ją D. Parnas w 1975 r.) i znalazła swoje odbicie w takich pojęciach programistycznych jak moduł, abstrakcyjny typ danych i obiekt/klasa (co za tym idzie, niektórzy autorzy wyróżniają trzy rodzaje hermetyzacji).

W obiektowości dość popularnym stereotypem jest koncepcja ortodoksyjnej hermetyzacji (Smalltalk), w której wszelkie operacje, które można wykonać na obiekcie, są określone przez metody przypisane do obiektu; bezpośredni dostęp do atrybutów obiektu jest niemożliwy. Oznacza to często, że każdy atrybut obiektu musi być wyposażony w dwie metody: CzytajWartość i ZmieńWartość; takie założenie przyjmuje np. standard CORBA. Alternatywą jest hermetyzacja ortogonalna (C++, Eiffel), gdzie dowolny atrybut obiektu (i dowolna metoda) może być prywatny (niedostępny z zewnątrz) lub publiczny. Atrybut publiczny może być obsłużony przez operatory generyczne, takie jak odzyskanie referencji do atrybutu (czyli wiązanie), podstawienie i dereferencja. Zwolennicy ortodoksyjnej hermetyzacji argumentują, że zapewnia ona rzekomo brak możliwości zrobienia na obiektach danej klasy czegokolwiek, co nie zostało przewidziane przez projektanta tej klasy. Tę argumentację łatwo obalić, gdyż udostępnienie atrybutu poprzez metody CzytajWartość i ZmieńWartość jest semantycznie równoważne udostępnieniu danego atrybutu do przetwarzania poprzez w/w operatory generyczne.

Ortodoksyjna hermetyzacja jest ponadto semantycznie i koncepcyjnie niespójna, jeżeli założymy, że atrybuty mogą być powtarzalne, czyli mogą być kolekcjami o nieznanym i nieograniczonym rozmiarze; np. atrybut WypożyczoneKsiążki dla obiektów Student. W takich przypadkach zwolennicy ortodoksyjnej hermetyzacji sugerują, że potrzebne są metody takie jak DajPierwszy, DajNastępny, CzyOstatni (zaimplementowane np. w postaci iteratora). W takim przypadku powstaje pytanie, co mają zwrócić metody DajPierwszy i DajNastępny. Odpowiedź na to pytanie jest prosta: muszą one zwrócić referencje do (kolejnych) wartości tego atrybutu. Udostępnienie tych referencji na zewnątrz oznacza wyłom w koncepcji ortodoksyjnej hermetyzacji; np. takie referencje mogą być użyte w operacji podstawienia, przekazane jako call-by-reference parametr do procedury, itd. Dodatkowo, schemat iteracyjny DajPierwszy, DajNastępny, CzyOstatni (lub podobny) ustala określony porządek przetwarzania elementów, co oznacza, że kolekcje takie jak zbiory i wielozbiory stają się fikcją (oraz związane z nimi ewentualne metody optymalizacyjne). Poważnym argumentem na niekorzyść ortodoksyjnej hermetyzacji są języki zapytań, których semantyka bazuje na bezpośrednim wiązaniu nazw występujących w zapytaniu z wartościami atrybutów. Z tego względu C.J.Date uważa, że koncepcja hermetyzacji powinna być w ogóle odrzucona. Date w swoich wnioskach idzie za daleko; wystarczy bowiem zrezygnować z ortodoksyjnej hermetyzacji na rzecz hermetyzacji ortogonalnej, takiej, jak w C++, Eiffel, Modula-2, i innych językach (czyli wrócić do oryginalnego pomysłu D.Parnasa). Hermetyzacja ortogonalna nie prowadzi do jakichkolwiek sprzeczności z językami zapytań. Poniższy rysunek jest ilustracją hermetyzacji ortogonalnej.

Wewnętrzna struktura obiektu

Zewnętrzna struktura obiektu

- **Aktor** - W metodykach analizy i projektowania (np. w modelu przypadków użycia): obiekt modelujący określoną rolę zewnętrznego użytkownika systemu. Aktor może operować na innych obiektach, ale sam nie podlega operacjom ze strony innych obiektów. W językach tzw. aktorowych: fragment programu o dużym stopniu autonomii, działający równolegle (asynchronicznie i niezależnie) oraz posiadający własny stan i sterowanie. Synonimy: aktywny obiekt, agent, aktywny agent.

- **Aktywny obiekt (aktor)** - Obiekt posiadający własny program o sterowaniu inicjowanym i biegnącym niezależnie i równolegle w stosunku do przebiegu innych programów/procesów. Synonimy: aktor, agent, aktywny agent.

- **Klasa abstrakcyjna** - Klasa zawierająca własności (np. metody) dziedziczone przez jej podklasy, ale nie posiadająca bezpośrednich wystąpień obiektów. Stanowi ona wyższy poziom abstrakcji przy rozpatrywaniu pewnego zestawu obiektów. Np. klasa OSOBA może być klasą abstrakcyjną, o ile zdefiniowane są jej podklasy PRACOWNIK, STUDENT, EMERYT, itd. Z technicznego punktu widzenia klasę abstrakcyjną można uważać za nazwaną grupę cech, które są "wyciągnięte przed nawias" *(factoring out)* z pewnego zestawu klas. Klasa abstrakcyjna jest najczęściej używana do zdefiniowania wspólnego interfejsu dla pewnej liczby podklas. Klasa abstrakcyjna może mieć (nie musi) metody (operacje) abstrakcyjne, tj. takie, które są zdefiniowane w tej klasie, ale których implementacja znajduje się (jest oczekiwana) w jej bezpośrednich lub pośrednich podklasach. Pojęcie klasy abstrakcyjnej jest jednym z podstawowych dla obiektowości, wzmacniającym zarówno mechanizmy abstrakcji pojęciowej, jak i możliwości ponownego użycia. Możliwość definiowania klas abstrakcyjnych uważa się za wyróżnik obiektowości danego języka lub systemu.

- **Delegacja** - 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. Jest także określana jako dziedziczenie w ramach wystąpień obiektów, co kojarzy ją z pojęciem prototypu. Należy uprzedzić, że pojęcie delegacji nie zawsze jest jasne, ponieważ opiera się na dość powierzchownych cechach specyficznych dla modelu obiektowego przyjętego w danym języku, co powoduje dość różne jego rozumienie. Np. w języku Smalltalk (gdzie "wszystko jest obiektem") trudno wyróżnić istotną merytorycznie podstawę definicji tego pojęcia.

- **Analiza czasownikowo-rzeczownikowa –** specjalizacja, zbieranie wymagań od klienta; czasownik – metody, rzeczownik – dane (obiekty, składowe)

- **Zasada DRY –** nie powtarzamy kodu – korzystamy z napisanego wcześniej i/lub go modyfikujemy (wykorzystanie polimorfizmu)

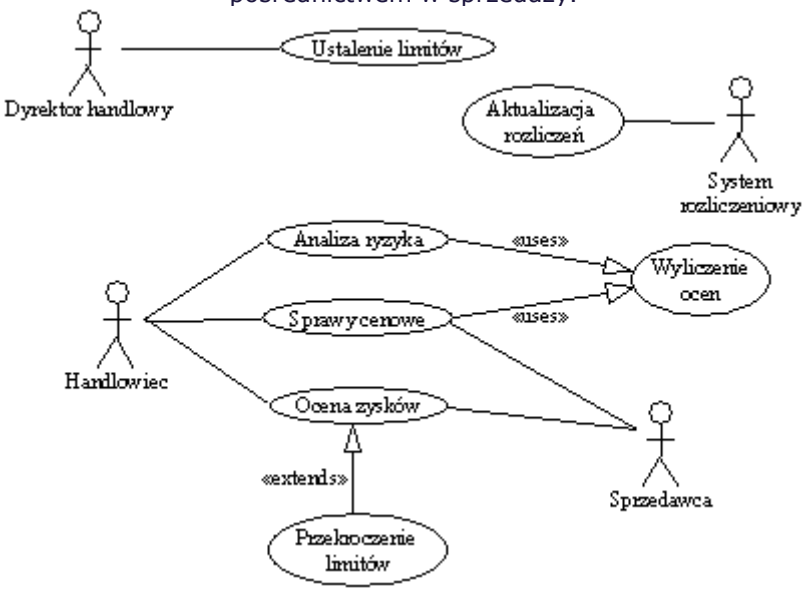
- **Zasady SRP –** obiekt jest odpowiedzialny tylko za jedną czynność

- **Karta CRC** - Karta papierowa posiadająca trzy pola nazwane: klasa, odpowiedzialność, współpraca (kolaboracja). Karta ta jest rekwizytem prostej metody ("burzy mózgów"), mającej na celu wyjaśnienie kluczowych abstrakcji i mechanizmów w systemie. Kartę taką zapełnia się dla każdej klasy. Wewnątrz karty wpisuje się nazwę klasy, określa się związek tej klasy z procesami zachodzącymi w danej dziedzinie przedmiotowej (odpowiedzialność), oraz określa się związek tej klasy z innymi klasami (współpraca); patrz rysunek poniżej.

Nazwa klasy **Zamówienie**	
Odpowiedzialność	*Współpraca*
Sprawdź, czy pozycja jest na składzie	***Pozycja zamówienia***
Określ cenę	***Pozycja zamówienia***
Sprawdź możliwości płatnicze klienta	***Klient***
Wyślij na adres klienta	

Przykładowa karta CRC

- **Diagram przypadków użycia** - Diagram pokazujący aktorów i przypadki użycia. Poniższy rysunek przedstawia diagram przypadków użycia dla pewnej firmy zajmującej się pośrednictwem w sprzedaży.



Przykładowy diagram przypadków użycia

Diagram przypadków użycia zawiera znaki graficzne oznaczające aktorów (ludziki) oraz przypadki użycia (owale z wpisanym tekstem). Te oznaczenia połączone są liniami i strzałkami odwzorowującymi powiązania poszczególnych aktorów z poszczególnymi przypadkami użycia oraz przypadki użycia z innymi przypadkami użycia. Strzałki oznaczone "extends" prowadzą od przypadku użycia, który (w niektórych sytuacjach) rozszerza dany przypadek użycia. Strzałki oznaczone "uses" prowadzą do bloku ponownego użycia, czyli takiego przypadku użycia, który może być wykorzystany w wielu przypadkach użycia. Patrz też: przypadek użycia.

- **Słowo kluczowe „this."** – referencja do obiektu, na rzecz którego wykonywana jest metoda

- **Adnotacja @Override** - informuje kompilator o tym, że dana metoda powinna przesłaniać inną metodę w klasie bazowej. Jeśli warunek ten nie będzie spełniony możesz spodziewać się błędu kompilacji.

- **Zasada segregacji interfejsów (Interface Segregation Principle)** - 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.

- **Zasada spójności (Single Responsibility Principle)** - Nie powinno być więcej niż jednego powodu do modyfikacji klasy. Przypisuj odpowiedzialności do obiektu tak, aby spójność była jak największa.

- **Zasada otwarte-zamknięte (Open-Close Principle)** - 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.

- **Information Expert Principle -** Programista powinien delegować nową odpowiedzialność do klasy zawierającej najwięcej informacji potrzebnych do zrealizowania nowej funkcjonalności. Niezbędne jest wcześniejsze określenie, jakie dane są niezbędne.

- **Low Coupling Principle** - Deleguj odpowiedzialności tak, aby zachować jak najmniejsza liczbę powiązań pomiędzy klasami.

- **Zasada odwrócenia zależności (Dependency Inversion Principle)** - Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych – zależności między nimi powinny wynikać z abstrakcji.

- **Zasada podstawienia Liskov (Liskov substitution principle) –** 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. Inaczej mówiąc, klasa dziedzicząca powinna tylko rozszerzać możliwości klasy bazowej i w pewnym sensie nie zmieniać tego, co ona robiła już wcześniej. Mówiąc jeszcze inaczej — jeśli będziemy tworzyć egzemplarz klasy potomnej, to niezależnie od tego, co znajdzie się we wskaźniku na zmienną, wywoływanie metody, którą pierwotnie zdefiniowano w klasie bazowej, powinno dać te same rezultaty.

Pytania:

Jakie są różnice pomiędzy podejściem proceduralnym a obiektowym?

Programowanie strukturalne

1. Programowanie strukturalne jest paradygmatem programowania, zalecającym podział programu na moduły, komunikujące się poprzez dobrze określone interfejsy. Jest rozszerzeniem koncepcji programowania proceduralnego, zalecającego dzielenie kodu na procedury, wykonujące ściśle określone zadania. Procedury nie powinny korzystać z parametrów globalnych, ale przekazywać wszystkie potrzebne dane jako parametry do procedury.
2. Programowanie strukturalne wykorzystuje do rozwiązywania zadań dobrze określone struktury algorytmiczne: sekwencja, selekcja, iteracja i rekursja; unika natomiast stosowania instrukcji skoku.
3. Programowanie strukturalne oddzielnie definiuje dane, oddzielnie funkcje.

Programowanie obiektowe

1. Programowanie obiektowe wprowadza pojęcie obiektu, w którym dane i procedury są ze sobą ściśle powiązane. Program obiektowy korzysta z obiektów, komunikujących się ze sobą w celu wykonania określonych zadań.
2. Cechy typowe dla programowania obiektowego
    - abstrakcja - zredukowanie właściwości opisywanego obiektu do najbardziej podstawowych,
    - hermetyzacja danych - dostęp do składowych jest ograniczony za pomocą dobrze określonego interfejsu,
    - dziedziczenie - mechanizm umożliwiający wywodzenie nowych klas z klas już istniejących, wraz z przejmowaniem ich metod,
    - polimorfizm - wielopostaciowość, pozwala na wybór metody spośród różnych wersji w zależności od kontekstu.

Jaka jest różnica pomiędzy klasą a obiektem?

W programowaniu obiektowym klasa jest częściową lub całkowitą definicją dla obiektów. Definicja obejmuje dopuszczalny stan obiektów oraz ich zachowania. Obiekt, który został stworzony na podstawie danej klasy nazywany jest jej instancją. Klasy mogą być typami języka programowania - przykładowo, instancja klasy Owoc będzie mieć typ Owoc. Klasy posiadają zarówno interfejs, jak i strukturę. Interfejs opisuje, jak komunikować się z jej instancjami za pośrednictwem metod, zaś struktura definiuje sposób mapowania stanu obiektu na elementarne atrybuty. W skrócie – obiekt to instancja klasy.

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

1. Modyfikator public

Słowo kluczowe public jest modyfikatorem dostępu, który pozwala na najbardziej swobodny dostęp do elementu, który poprzedza. public może być używane przed definicjami klas, pól w klasach, metod czy typów wewnętrznych. Zakładając, że klasa poprzedzona jest public i element w tej klasie jest także public, jest on dostępny dla wszystkich.

2. Modyfikator protected

Modyfikator protected ma znaczenie w przypadku dziedziczenia. Elementy poprzedzone tym modyfikatorem dostępu są udostępnione dla danej klasy i jej podklas. Dodatkowo elementy oznaczone modyfikatorem protected dostępne są dla innych klas w tym samym pakiecie. Modyfikatora protected nie można stosować przed klasami.

3. Brak modyfikatora dostępu

Brak modyfikatora dostępu również ma znaczenie. W przypadku gdy pominiemy modyfikator dostępu wówczas dana klasa czy element jest dostępna wyłącznie wewnątrz tego samego pakietu. Jest to podzbiór uprawnień, które nadaje modyfikator protected.

4. Modyfikator private

Słowo kluczowe private jest najbardziej restrykcyjnym modyfikatorem dostępu. Może być stosowane wyłącznie przed elementami klasy, w tym przed klasami wewnętrznymi. Oznacza on tyle, że dany element (klasa, metoda, czy pole) widoczny jest tylko i wyłącznie wewnątrz klasy.

5. Modyfikator packed

Zbiór obiektów współpracujących ze sobą.

Do czego służy metoda finalize w języku Java?

Metoda finalize jest wołana przez specjalny wątek Finalizera, który sprawdza czy obiekt spełnia warunki do bycia zabitym, jeśli tak to woła metodę finalize obiektu po czym sam Finalizer zapomina o obiekcie i ginie. Nie zwalnia zasób w maszynie wirtualnej – zwalnia je poza nią.

Jakiego mechanizmu języka Java skorzystasz, aby hermetyzować zachowania?

Aby hermetyzować zachowania trzeba skorzystać z modyfikatorów dostępu. Sprowadza się to do użycia private dla wszystkich pól i metod, które powinny być używane "wewnątrz". Pozostałe elementy, które stanowią interfejs komunikacji oznaczamy słowem kluczowym public. Brak modyfikatora dostępu czy protected mają znaczenie w przypadku bardziej złożonych relacji pomiędzy obiektami.

Jaka jest różnica pomiędzy interfejsem języka java a klasą abstrakcyjną?

Różnice między klasą abstrakcyjną, a interfejsem to, np. że dana klasa może dziedziczyć tylko po jednej klasie, a może implementować wiele interfejsów. Druga rzecz, to w klasie abstrakcyjnej można definiować ciała metod, a w interfejsach występują tylko sygnatury metod. Interfejsów używa się zazwyczaj do rozwiązań polimorficznych, za to klasy abstrakcyjne do agregowania kodu i jego uwspólniania między klasami. Mówiąc inaczej - interfejs powinien mówić co można z daną klasą zrobić, a klasa abstrakcyjna powinna być narzędziem do budowy hierarchii klas.

Czy w języku Java jest możliwe dziedziczenie wielobazowe?

Dziedziczenie może być wielopoziomowe, jednak w języku Java zawsze bezpośrednio możemy dziedziczyć od jednej klasy.

Jak wiele interfejsów może implementować klasa w języku Java?

Klasa może implementować nieskończenie wiele interfejsów, ale nie powinno być ich zbyt wiele dla czystości samego kodu. W takim przypadku należy skorzystać np. z delegacji.