1. Z czego składa się oprogramowanie?

Oprogramowanie *(software*) – pojęcie podstawowe (do oprogramowania włącza się niekiedy, oprócz np. plików binarnych i z kodem źródłowym, także dokumentację, pliki konfiguracyjne, przykładowe dane itp.)

2. Czym zajmuje się inżynieria oprogramowania?

Inżynieria oprogramowania (IO, *software engineering*) – dziedzina inżynierii związana z wytwarzaniem oprogramowania we wszystkich fazach jego cyklu życia

Jako dziedzina inżynierii IO rozważa w kontekście praktycznym rozmaite aspekty wytwarzania oprogramowania (techniczny, organizacyjny, finansowy itp.)

Cele inżynierii oprogramowania:

- → Dostarczenie zasad organizacji procesu wytwarzania oprogramowania (*software development*) w oparciu o istniejące teorie, modele, metody i narzędzia (podejście może być tutaj mniej lub bardziej formalne)
- → W końcowym efekcie proces wytwarzania oprogramowania ma w sposób efektywny doprowadzić do powstania produktu wysokiej jakości
- → Inżynieria oprogramowania jest częścią szerszej dyscypliny – inżynierii systemów *(system engineering),* w jej aspekcie dotyczącym oprogramowania

Inżynieria oprogramowania stara się zidentyfikować i opisać podstawowe fazy tworzenia i funkcjonowania oprogramowania, a także wskazać model optymalnego przebiegu tych faz

3. Wymień cechy dobrego oprogramowania

Cechy dobrego oprogramowania wg Sommerville:

- → poprawność, zgodność z wymaganiami użytkowników
- → łatwość pielęgnacji (konserwacji), dokonywania zmian
- → niezawodność *(availability, reliability),* bezpieczeństwo (w obu znaczeniach – *safety, security*)
- → wydajność, efektywne wykorzystanie zasobów
- → łatwość stosowania, ergonomiczność

4. Podaj podstawowe czynności związane z wytwarzaniem oprogramowania

Podstawowe czynności związane z wytwarzaniem oprogramowania:

- → Określanie wymagań i specyfikacji
- → Projektowanie
- → Implementacja
- → Testowanie – walidacja (atestowanie) i weryfikacja
- → Konserwacja (pielęgnacja)

5. Scharakteryzuj zadany model cyklu życia oprogramowania (jeden z poniższych), spróbuj

zilustrować model pokazując np. układ podstawowych czynności z p.4 dla zadanego
modelu; podaj podstawowe wady i zalety modelu:

a) model spiralny
b) model kaskadowy
c) model ewolucyjny (iteracyjny)
   - z prototypowaniem
   - z wytwarzaniem odkrywczym
   - z wytwarzaniem przyrostowym
d) model komponentowy

Model spiralny:

Model spiralny został zaproponowany jako ogólny model iteracyjnego rozwoju systemów oprogramowania

Wyróżnia się w nim cztery etapy, przez które przechodzą poszczególne fazy (czynności) w trakcie wytwarzania oprogramowania (czynnościami jest m.in. określanie wymagań, projektowanie i tworzenie kolejnych prototypów):

→ planowanie, określanie celu, w kontakcie z klientem
→ analiza alternatyw oraz szacowanie i minimalizacja ryzyka
→ realizacja, testowanie, wdrożenie
→ walidacja – ocena wraz klientem, także pod kątem planu kolejnej fazy

Zaletą tego modelu jest jawne uwzględnienie ryzyka

Model kaskadowy:

Jeden z podstawowych modeli cyklu życia oprogramowania

W modelu kaskadowym kolejne etapy procesu rozwoju oprogramowania następują po sobie w ściśle określonym porządku:

→ Określenie wymagań (*requirements*)
→ Projektowanie systemu (*system design*)
→ Implementacja i testowanie modułów (podsystemów)
→ Testowanie połączeń modułów i całości systemu
→ Użytkowanie i pielęgnacja (konserwacja, *maintenance)*

Każda następna faza rozpoczyna się dopiero po (często formalnym) zakończeniu fazy poprzedzającej

Zaletą modelu kaskadowego jest zidentyfikowanie podstawowych faz rozwoju oprogramowania i
uporządkowanie procesu tworzenia oprogramowania (ułatwia to planowanie i zarządzanie wykonaniem)

Wadą modelu kaskadowego jest rygorystyczne określenie następstwa faz (co może utrudniać realizację), w konsekwencji, jeżeli pewne błędy zostają popełnione w fazie określania wymagań lub projektowania, a wykryte w fazie testowania lub użytkowania, koszt ich usunięcia okazuje się bardzo wysoki

Realizacja kierowana dokumentami (*document driven*) jest wariantem modelu kaskadowego, w którym przejście od jednej fazy do następnej odbywa się po zaakceptowaniu zbioru dokumentów

Model ewolucyjny:

Celem modelu ewolucyjnego jest poprawienie modelu kaskadowego poprzez rezygnację ze ścisłego, liniowego następstwa faz

Pozostawia się te same czynności, ale pozwala na powroty, z pewnych faz do innych faz poprzedzających

Tym samym umożliwia się adaptowanie do zmian w wymaganiach i korygowanie popełnionych błędów (oba zjawiska występują w niemal wszystkich praktycznie wykonywanych projektach – stąd model ewolucyjny jest bardziej realistyczny od kaskadowego)

Model ewolucyjny wymaga dodatkowych strategii dla uporządkowania procesu wytwarzania oprogramowania

Prototypowanie:

Prototypowanie jest techniką w ramach wytwarzania ewolucyjnego, w której pojawia się nowa faza tworzenia oprogramowania, poza wymienionymi dotychczas – faza tworzenia prototypu

Prototyp jest niepełnym systemem, spełniającym cześć wymagań, przeznaczonym do przetestowania rozwiązań wykorzystanych do jego wytworzenia

Z założenia prototyp nie wchodzi w skład ostatecznego systemu (ostateczny system budowany jest od podstaw po zaakceptowaniu rozwiązań zastosowanych w prototypie)

Prototypowanie, będące w powszechnym użyciu w innych dziedzinach produkcji, w informatyce posiada swoją specyfikę:

- → najczęściej wykorzystywane jest w fazie uzgadniania wymagań – prototypy służą do wykrystalizowania i ostatecznego ustalenia wymagań klienta
- → w trakcie tworzenia oprogramowania wiele wysiłku wkłada się w ponowne wykorzystanie raz utworzonego kodu – stąd typowe dla prototypowania podejście z porzucaniem prototypów nie jest zalecane
- → do stworzenia prototypu także trzeba wykorzystać jakiś model rozwoju oprogramowania

Wytwarzanie odkrywcze:

Wytwarzanie odkrywcze (*exploratory development)* jest wariantem modelu ewolucyjnego, w którym iteracje dotyczą całego cyklu wytwarzania oprogramowania

Istotą wytwarzania odkrywczego jest stała współpraca z klientem, który otrzymuje kolejne, coraz bogatsze wersje systemu i na ich podstawie określa i uszczegóławia swoje wymagania

Wytwarzanie odkrywcze dobrze radzi sobie z występującym powszechnie faktem zmiany wymagań przez klientów

Wytwarzanie odkrywcze polega na stałej modyfikacji kodu, bez odrzucania dotychczas wytworzonego oprogramowania (przeciwnie do prototypowania)

Wytwarzanie odkrywcze musi poradzić sobie z dwoma istotnymi problemami:

- → jak często dokonywać rewizji aktualnego stanu kodu (problem ważny dla efektywnego zarządzania wytwarzaniem programów)
- → jak nie dopuścić do utraty jednolitej struktury przez ciągle modyfikowany kod

W przypadku stosowania wytwarzania odkrywczego konieczne jest wypracowanie strategii zarządzania wersjami, np. takiej w której istnieją wersje będące drobnymi poprawkami w stosunku do poprzednich oraz istotne zmiany, przy których odrzuca się znaczne części poprzedniej wersji kodu (która może być wtedy traktowana jako prototyp)

Wytwarzanie przyrostowe:

Kolejnym wariantem modelu ewolucyjnego jest wytwarzanie przyrostowe (*incremental development*)

W wytwarzaniu przyrostowym najpierw następuje określenie wymagań, po czym całość systemu dzielona jest na kolejne „przyrosty” (*increments*), tworzone każdorazowo, dające się testować, rozrastające się wersje systemu (pierwsze wersje zazwyczaj ujmują podstawowe funkcjonalności systemu)

Problemem podstawowym wytwarzania przyrostowego jest określenie „przyrostów”, tak aby były one istotnymi fragmentami oprogramowania, a mimo to każdą z wersji dawało się niezależnie testować i oceniać

Model komponentowy:

W modelu komponentowym idee ponownego użycia kodu posunięte są najdalej

Po fazie określania wymagań następuje faza analizy możliwości wykorzystania istniejących, gotowych komponentów i ewentualna faza modyfikacji wymagań, w konsekwencji zastosowania komponentów

W fazie projektowania uwzględnia się już znalezione komponenty oraz ewentualnie nowe, związane z techniczną realizacją (implementacją)

Projekt oprogramowania wykonywany jest tak, aby te spośród wytwarzanych elementów, które się do tego nadają, mogły być ponownie wykorzystane jako komponenty

W fazie wytwarzania kodu zwraca się szczególną uwagę na interfejsy pomiędzy modułami/komponentami

Testowanie jest w dużej mierze testowaniem integracji poszczególnych komponentów

Mimo zalet związanych z wykorzystaniem gotowych, przetestowanych modułów, wytwarzanie oprogramowania w oparciu o komponenty (*component based software development*) stwarza specyficzne trudności:

- → wymagania narzucane przez gotowe komponenty mogą być niezgodne z wymaganiami klientów
- → modyfikacje kodu mogą być utrudnione przez brak kontroli nad pochodzącymi z zewnątrz komponentami

6. Jakie znasz rodzaje (poziomy) narzędzi CASE, podaj przykłady narzędzi każdego

poziomu?

CASE – tłum. Inżynieria oprogramowania wspomagana komputerowo. Narzędzia:

- → niskiego poziomu – edytory i kompilatory (GCC, Dev-C++)
- → wspomagające poszczególne fazy – analizy, projektowania, implementacji, testowania – Ms. Visio, SmartDraw
- → zintegrowane środowiska wytwarzania – Ms. Visual Studio, Borland Developer Studio, Borland Together

7. Określ fazy procesu określania wymagań, na czym polega zarządzanie procesem

określania wymagań?

Określanie wymagań może odbywać się na różnych etapach realizacji projektu:

- → w fazie przeprowadzania studium wykonalności projektu (*feasibility study*)
- → po podjęciu decyzji o realizacji projektu, ale przed podpisaniem kontraktu o jego realizacji (lub ogłoszeniem przetargu)
- → po podpisaniu kontraktu

Na każdym z etapów inny będzie zakres wymagań, poziom szczegółowości opisu itp.

Zakres czynności wykonywanych w różnych fazach zależy od charakteru projektu i wielu innych czynników często związanych z marketingiem i zarządzaniem

Zagadnienia związane z marketingiem i zarządzaniem (np. pisanie ofert przetargowych i studiów wykonalności, zdobywanie kontraktów, pozyskiwanie klientów i utrzymywanie odpowiednich relacji z nimi) są w przemyśle informatycznym w wielu aspektach takie same jak w innych dziedzinach produkcji i usług

Proces określania wymagań dla systemu informatycznego można podzielić na następujące fazy:

- → faza ustalania wymagań (odkrywania wymagań)
- → faza specyfikacji wymagań (tworzenia opisu wymagań)
- → faza atestacji (walidacji, *validation*) wymagań

Fazy powyższe mogą być powtarzane wielokrotnie na różnych etapach określania wymagań, wraz z rosnącym zakresem i poziomem szczegółowości wymagań i proponowanych modeli dla systemu

Za każdym razem zakładać będziemy, że w określaniu wymagań uczestniczy klient (znający dziedzinę zastosowań) i wykonawca (odpowiedzialny za aspekty informatyczne)

Obie strony, klient i wykonawca, muszą porozumieć się co do wielu elementów, przy czym klient często nie rozumie specyfiki funkcjonowania programów wykonawca nie zna specyfiki dziedziny zastosowań

Klient musi ustalić swoje wymagania w kontekście możliwości i ograniczeń charakterystycznych dla systemów informatycznych

Wykonawca musi dostosować funkcjonowanie programów do standardów i konwencji dziedziny zastosowań

8. Omów klasyfikację wymagań na funkcjonalne i niefunkcjonalne oraz na użytkownika i systemowe

Z punktu widzenia rodzaju wymagań można je podzielić na:

- → wymagania funkcjonalne – dotyczące tego co ma realizować system; jakie ma spełniać funkcje, jakich dostarczać usług, jak zachowywać się w określonych sytuacjach
- → wymagania niefunkcjonalne – dotyczące tego jak system powinien realizować swoje zadania; np. wymagania dotyczące koniecznych zasobów, ograniczeń czasowych, niezawodności, bezpieczeństwa, przenośności, współpracy z określonymi narzędziami i środowiskami, zgodności z normami i standardami, a także przepisami prawnymi, w tym dotyczącymi tajności i prywatności itp.

Wymagania funkcjonalne powinny być:
kompletne – opisywać wszystkie usługi żądane od systemu
spójne – nie zawierać stwierdzeń sprzecznych

Wymagania niefunkcjonalne dla wielu systemów są co najmniej równie ważne jak wymagania funkcjonalne (np. szybkość działania wyszukiwarki może być równie ważna jak precyzja wyszukiwania)

Niekiedy wymagania niefunkcjonalne na pewnym poziomie szczegółowości stają się wymaganiami funkcjonalnymi na innym poziomie (np. wymagania dotyczące bezpieczeństwa – stopień bezpieczeństwa może przekształcić się w konieczność realizowania pewnych funkcji związanych z bezpieczeństwem)

Z punktu widzenia poziomu szczegółowości wymagania można podzielić na:

- → wymagania użytkownika (ogólna definicja wymagań) – zestaw życzeń użytkownika co do funkcjonowania systemu, wyrażony głównie z użyciem języka dziedziny zastosowań
- → wymagania systemowe (specyfikacja wymagań) – szczegółowy opis funkcji, usług i ograniczeń systemu, wyrażony z szerszym zastosowaniem kategorii informatycznych

9. Podaj rodzaje wymagań niefunkcjonalnych, jakie miary można wiązać z poszczególnymi kategoriami wymagań?

Aby przekształcić zbyt ogólne wymagania niefunkcjonalne stosuje się wiele miar dla różnych kategorii wymagań, np.:

- → dla szybkości działania: liczba transakcji na sekundę, czas reakcji na zdarzenie, czas odświeżenia strony
- → dla łatwości użycia: czas konieczny na przeszkolenie obsługi, liczba stron (okien) pomocy

- → dla niezawodności: średni czas do awarii, prawdopodobieństwo niedostępności
- → dla odporności: procent zdarzeń powodujących awarię, prawdopodobieństwo zniszczenia danych w trakcie awarii
- → dla przenośności: procent instrukcji zależnych od np. systemu operacyjnego lub bazy danych, liczba uwzględnionych systemów operacyjnych (baz danych)

10. Przedstaw techniki stosowane przy odkrywaniu wymagań

Pomocnymi technikami w odkrywaniu wymagań są:

- → poznanie całości otoczenia systemu (poprzez obserwacje, zaznajomienie z odpowiednimi dokumentami itp.)
- → wykorzystanie istniejących systemów realizujących podobne funkcje
- → obserwacje i wywiady z przyszłymi użytkownikami systemu
- → stosowanie scenariuszy wykorzystania systemu (przypadków użycia, *uses cases*)
- → modelowanie systemu
- → tworzenie prototypów systemu
- → rozważanie punktów widzenia (viewpoints)

11. Omów czym są tzw. punkty widzenia (oraz ściśle z nimi powiązane pojęcia aktorów lub ról w UML) w procesie odkrywania wymagań

Przydatną techniką przy odkrywaniu wymagań jest rozważanie tzw. punktów widzenia (*viewpoints*)

Punkt widzenia określa dowolną osobę, której w jakiś sposób dotyczy funkcjonowanie systemu, ewentualnie element szeroko rozumianego otoczenia wpływający na wymagania systemu

Punkty widzenia można podzielić następująco:

- → bezpośrednie – związane z ludźmi bezpośrednio korzystającymi z systemu bądź obsługującymi system
- → pośrednie – związane z ludźmi pośrednio zainteresowanymi funkcjonowaniem systemu (kierownictwo, osoby odpowiedzialne za bezpieczeństwo)
- → związane z dziedziną – standardy, przepisy organizacyjne, itp.

Punkty widzenia są źródłem rozmaitych oczekiwań, wizji i konkretnych wymagań w stosunku do systemu

Wygodnym sposobem porządkowania informacji uzyskanych z różnych punktów widzenia jest tworzenie scenariuszy – opisów możliwych sekwencji zdarzeń związanych z funkcjonowaniem systemu, które można następnie grupować w zbiory odpowiadające realizacji konkretnych funkcji lub grup funkcji systemu zwanych przypadkami użycia

Przypadek użycia oznacza interakcję z całym systemem lub jego podsystemem prowadzącą do pewnego konkretnego rezultatu

Pojedynczy przypadek użycia obejmuje zazwyczaj pewną ilość scenariuszy związanych z wariantami sposobu korzystania z systemu, zdarzeniami nietypowymi, itp.

Dla określenia wymagań istotne znaczenie mogą mieć właśnie przypadki nietypowe (awarie sprzętu, błędy użytkowników), które będą testowały istotne cechy systemu pod kątem niezawodności i bezpieczeństwa działania

Do zapisu przypadków użycia stosuje się często notację graficzną, np. diagramy przypadków użycia UML

W diagramach przypadków użycia UML pojawiają się:

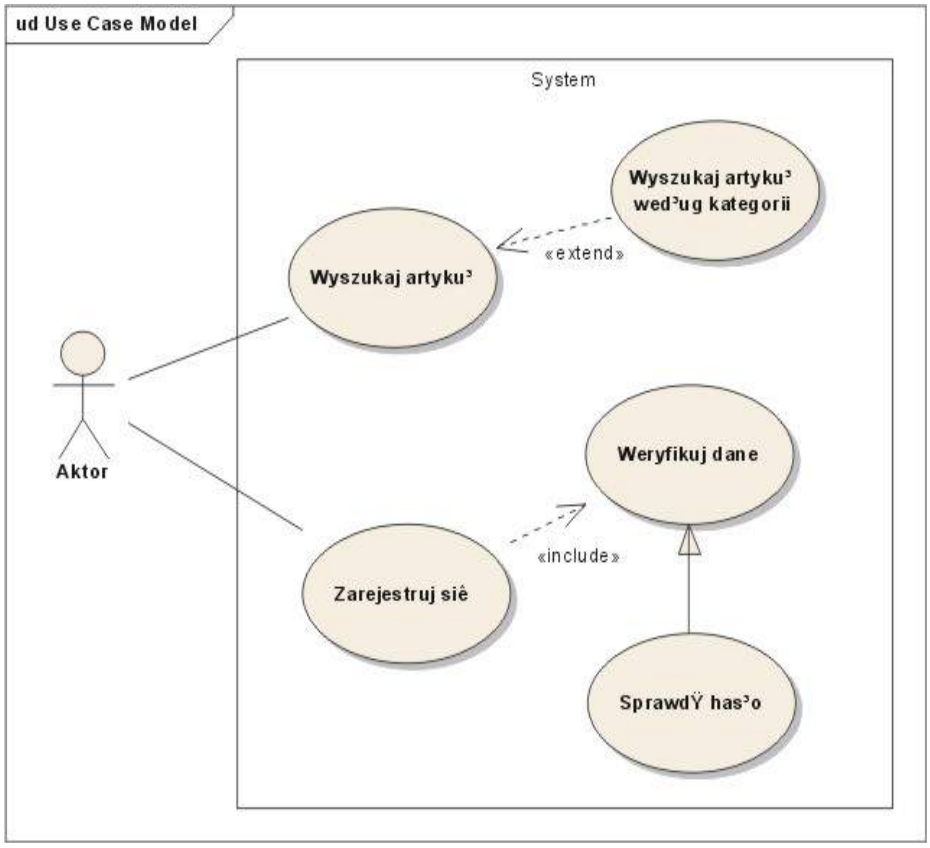
- → aktorzy – reprezentujący osoby, grupy osób lub elementy otoczenia systemu wchodzące w interakcje z systemem i odgrywający pewną rolę
- → przypadki użycia – oznaczające zbiór scenariuszy związanych z konkretnym wykorzystaniem systemu
- → klasyfikatory (np. pakiety) – do których przynależą konkretne przypadki użycia
- → powiązania – łączące aktorów z przypadkami użycia, w których odgrywają swe role

12. Czym są przypadki użycia, a czym scenariusze?

Scenariuszy – opis możliwych sekwencji zdarzeń związanych z funkcjonowaniem systemu, przypadek użycia oznacza interakcję z całym systemem lub jego podsystemem prowadzącą do pewnego konkretnego rezultatu. Pojedynczy przypadek użycia obejmuje zazwyczaj pewną ilość scenariuszy związanych z wariantami sposobu korzystania z systemu, zdarzeniami nietypowymi, itp.

13. Narysuj przykład diagramu przypadków użycia dla wybranego przez siebie systemu;

spróbuj umieścić na rysunku jak najwięcej znanych ci elementów diagramów przypadków
użycia, omów znaczenie tych elementów



Aktor – określa osobę, która chce skorzystać z systemu
System – nazwa systemu, prostokąt określa granice
Wyszukaj artykuł według kategorii – przypadek użycia rozszerzający
Wyszukaj artykuł, zarejestruj się – przypadek użycia bazowy
Weryfikuj dane – przypadek użycia zawierany
Sprawdź hasło – przypadek użycia uszczegóławiający
ud Use Case Model – nagłówek

14. Podaj przykład tekstowego opisu przypadku użycia dla wybranego przez siebie systemu;

nadaj opisowi przypadku użycia standardowy wygląd

Nazwa: Dokonaj rezerwacji

Inicjator: Rezerwujący
Cel: Zarezerwować pokój w hotelu
Główny scenariusz:

1. Rezerwujący zgłasza chęć dokonania rezerwacji
2. Rezerwujący wybiera hotel, datę, typ pokoju
3. System podaje cenę pokoju
4. Rezerwujący prosi o rezerwację
5. Rezerwujący podaje swoje potrzebne dane
6. System dokonuje rezerwacji i nadaje jej identyfikator
7. System podaje Rezerwującemu identyfikator rezerwacji i przesyła go mailem

Rozszerzenia:

3. Pokój niedostępny.
    a. System przedstawia inne możliwości wyboru
    b. Rezerwujący dokonuje wyboru

3b. Rezerwujący odrzuca podane możliwości
    a. Niepowodzenie
    itd.

15. Podaj przykład (lub wyłącznie schemat) specyfikacji dla pojedynczej funkcji (metody)

Przykładem konwencji zapisu wymagań w języku naturalnym może być stosowanie następującego formularza wymagającego podania dla każdej funkcji systemu:

Nazwa: oblicz_szybkość

Rola: oblicza szybkość obrotów wentylatora, na podstawie aktualnej temperatury elementu chłodzonego i otoczenia oraz poprzednich pomiarów

Opis działania: funkcja pobiera wartości odczytów temperatury i oblicza szybkość wentylatora na podstawie wzoru .....

Elementy współpracujące: pamięć trwała ...., czujniki temperatury – interfejs ...

Dane wejściowe: temperatury aktualne i poprzednie

Źródła danych: pamięć – poprzednie wartości, interfejs z czujnikami – aktualne wartości

Dane wyjściowe: prędkość obrotowa wentylatora w obr/s

Efekty uboczne: dodanie aktualnych odczytów do historii pomiarów

Warunki początkowe: historia pomiarów: rzeczywista lub zainicjowana

Warunki końcowe: wartość prędkości poprawnie obliczona, zaktualizowana historia

16. Jaka jest rola formalnych specyfikacji wymagań?

Sformalizowane specyfikacje posługują się specjalnymi językami specyfikacji wykorzystującymi pojęcia i notacje matematyczne

Pojęcia służą do przedstawienia stanów systemu oraz zmian tych stanów

Języki formalne są często specyficznymi opisami interfejsów funkcji systemu, które pozwalają na określenie poprawności funkcji systemu jako operacji zmieniających stan systemu

Formalna specyfikacja wymagań następuje zazwyczaj po stworzeniu wstępnego projektu systemu

Zalety:

- → sprecyzowanie określeń i usunięcie sprzeczności w wymaganiach
- → zagwarantowanie poprawności oprogramowania tworzonego na ich podstawie (ważne dla systemów o zakładanych wysokich: niezawodności i bezpieczeństwie)

Wady:

- → trudność uzyskania (zwłaszcza dla dużych i złożonych systemów oraz wszelkich systemów gdzie wymagania są słabo określone lub mogą się zmieniać)
- → znaczny czas konieczny do wypracowania (wada istotna w czasach silnej konkurencji)
- → nieprzydatność dla pewnych elementów systemów (np. graficzne interfejsy użytkownika)

Im bardziej szczegółowa i im bardziej formalna jest specyfikacja wymagań tym więcej obejmuje szczegółów z fazy projektowania i implementacji

17. Podaj elementy wzorcowego dokumentu definiującego wymagania dla systemu

Przyjmuje się że opis wymagań powinien zawierać następujące elementy (Sommerville):

- → Przedmowa (docelowi czytelnicy opisu, historia wersji)
- → Wstęp (motywacja stworzenia systemu, ogólny opis systemu i środowiska w którym ma funkcjonować)
- → Słownik (słownik pojęć użytych w opisie wymagań)
- → Definicja wymagań użytkownika (opis usług i wymagań niefunkcjonalnych zrozumiały dla klientów, współpraca z innymi systemami)
- → Ogólna architektura systemu (podsystemy, ponownie wykorzystywane komponenty)
- → Specyfikacja wymagań systemowych (szczegóły wymagań funkcjonalnych i niefunkcjonalnych)
- → Modele systemu (rozmaite modele systemu ukazujące go z różnych perspektyw, ilustracje graficzne)
- → Opis ewolucji systemu (przewidywany rozwój systemu związany z np. zmianami sprzętu, organizacyjnymi lub tp.)
- → Dodatki (dodatkowe informacje szczegółowe np.: opis sprzętu, wykorzystywanych baz danych itp.)
- → Indeks (spisy tabel, ilustracji, rozmaite indeksy – osób, pojęć itp.)

Określony w wymaganiach pożądany sposób działania systemu doskonale pasuje do ustalenia sposobu testowania systemu

18. Czym jest i w jaki sposób jest przeprowadzana walidacja wymagań

Walidacja (atestowanie, *validation*) wymagań oznacza sprawdzanie czy uzyskane ostatecznie wymagania odpowiadają pierwotnym życzeniom i oczekiwaniom klienta

Sposobami walidacji są:

- → przegląd wymagań (pod kątem ich zgodności z oczekiwaniami klienta a także zupełności, niesprzeczności, sprawdzalności realizacji itp.)
- → tworzenie prototypów na podstawie wymagań
- → tworzenie planów testów na podstawie wymagań

W sytuacjach kiedy określanie wymagań jest trudne ich walidacja stanowi ważny element procesu rozwijania oprogramowania

19. Przedstaw (wraz z krótką charakterystyką) rodzaje modeli systemu

Modelowanie można określić jako próbę uchwycenia w kategoriach informatycznych i wyrażenia za pomocą specjalnej notacji graficznej najważniejszych cech rozwijanego systemu oraz jego otoczenia

Modele systemu można podzielić na dwie podstawowe grupy:

- → modele zachowania systemu (aspekt dynamiczny) – istotny jest zestaw działań realizowanych przez system, które należą zazwyczaj do dwóch grup: zmiana stanu systemu, przetwarzanie danych wejściowych w dane wyjściowe
- → modele struktury systemu (aspekt statyczny) – pokazują statyczne elementy składowe systemu i ich wzajemne zależności

Istnieją rodzaje modeli ukazujące oba aspekty systemu

Modelami zachowania ukierunkowanymi na zmiany stanów systemu są modele maszyn stanowych UML

20. Omów możliwe sposoby użycia języka UML w procesie wytwarzania oprogramowania

Język UML (*Unified Modeling Language*) jest rozbudowanym językiem modelowania systemów oraz tworzenia specyfikacji

Z teoretycznego punktu widzenia UML może być traktowany jako sposób uściślenia pojmowania bytów występujących w programach (zwłaszcza obiektowych)

Z praktycznego punktu widzenia UML może być stosowany jako wyłącznie notacja ułatwiająca zrozumienie konkretnych systemów komputerowych

Język UML może być w procesie wytwarzania oprogramowania wykorzystywany na kilka różnych sposobów:

→ można stosować UML jako pomocniczą notację ułatwiającą zrozumienie struktury i funkcjonowania programu
→ można używać UML jako sposobu zapisu szczegółowego projektu systemu
→ można próbować stosowania UML jako ścisłego języka opisu tworzonego programu, tak aby możliwe było automatyczne wygenerowanie ostatecznego kodu w konkretnym języku programowania

21. Podaj podstawowe rodzaje diagramów UML, scharakteryzuj różne perspektywy spojrzenia na system informatyczny i przyporządkuj im odpowiednie typy diagramów UML (pojedynczy rodzaj diagramów może być stosowany dla różnych perspektyw)

Model UML systemu jest wyrażany w szeregu diagramów przedstawiających rozmaite części i aspekty modelu

Diagramy różnią się:

→ rodzajem – różne typy diagramów odpowiadają różnym sposobom widzenia systemu
→ stopniem szczegółowości – każdy diagram tworzony jest z konkretnym celem w konkretnej fazie rozwijania oprogramowania; inny poziom szczegółowości zawierać będzie konsultowany z użytkownikami diagram z fazy określania wymagań, a inny diagram mający być szczegółową specyfikacją elementu systemu, przeznaczony do automatycznej generacji kodu na jego podstawie

Obecna specyfikacja UML wyróżnia 13 rodzajów diagramów w następującej hierarchii:

→ diagramy struktury:
  ▪ diagramy klas, obiektów, komponentów, struktur złożonych, pakietów, wdrożenia
→ diagramy zachowania:
  ▪ diagramy przypadków użycia, maszyny stanowej, czynności
  ▪ diagramy interakcji - przeglądu interakcji, sekwencji, komunikacji, czasowe

Autorzy UML rozróżniają pięć perspektyw spojrzenia na system informatyczny i przyporządkowują im odpowiednie rodzaje diagramów UML:

→ perspektywa przypadków użycia – zachowanie systemu obserwowane z zewnątrz; diagramy przypadków użycia, interakcji, stanów i czynności
→ perspektywa projektowa – klasy, interfejsy, kooperacje; diagramy klas, obiektów, interakcji, stanów i czynności
→ perspektywa procesowa – wątki i procesy, współbieżność i synchronizacja; diagramy te same jak w perspektywie projektowej ze szczególnym uwzględnieniem klas i obiektów aktywnych

- → perspektywa implementacyjna – komponenty i pliki, zarządzanie konfiguracją; diagramy komponentów, interakcji, stanów i czynności
- → perspektywa wdrożeniowa – węzły, fizyczna realizacja sprzętowa; diagramy wdrożenia, interakcji, stanów i czynności

22. Czym jest standardowa inżynieria (“inżynieria wprzód”, *forward engineering*) a czym inżynieria odwrotna (*reverse engineering*)

Inżynieria wprzód (ang. forward engineering) to proces w którym najpierw następuje dokładne modelowanie i projektowanie systemu, a dopiero następnie jego implementacja. Tworzenie kodu odbywa się na podstawie możliwie kompletnych modeli.

Inżynieria odwrotna (ang. reverse engineering) to proces badania produktu (urządzenia, programu komputerowego) w celu ustalenia jak dokładnie działa, a także w jaki sposób i jakim kosztem został wykonany. Zazwyczaj prowadzony w celu zdobycia informacji niezbędnych do skonstruowania odpowiednika lub prezentacji działania. Innym zastosowaniem jest porównanie lub zapewnienie współdziałania z własnymi produktami. Przykład: Samba – serwer udostępniania plików i folderów protokołu Microsoft SMB dla Linuksa.

23. W jakich sytuacjach przydatne są diagramy stanu (rozróżnij diagramy maszyny stanowej zachowania i maszyny stanowej protokołu)

Modele stanu dla obiektów mają praktyczne znaczenie w przypadku obiektów, które mogą znajdować się w kilku stanach i których zachowanie jest różne w zależności od stanu w którym się znajdują

Diagram stanu dla obiektu staje się ilustracją jego cyklu życia (*object lifecycle)*

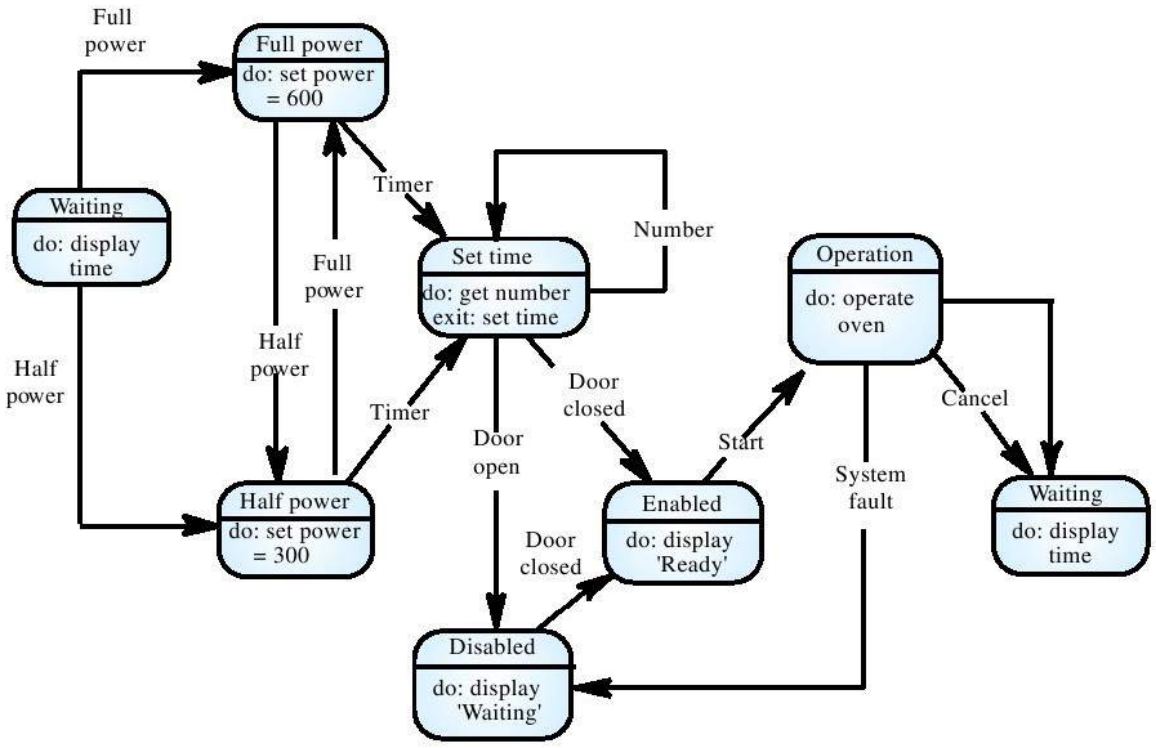
Przykładem notacji służącej do modelowania zmian stanów systemu są diagramy maszyny stanowej UML. W wersji 2.0 UML wprowadza się rozróżnienie między:

- → maszynami stanu modelującymi zachowanie (*behavioral state machines)* będącymi zmodyfikowaną wersją diagramów opracowanych przez Harela w roku 1987 (rozmaite diagramy stanów są w użyciu już od lat 60tych XX wieku) – tworzą model zachowania danego systemu (mikrofalówka, bankomat etc.)
- → maszynami stanu reprezentującymi dozwolone sposoby użycia egzemplarzy klasyfikatorów (np. obiektów jako egzemplarzy klas) (*protocol state machines)* – np. protokół dostępu do bazy danych

Maszyny stanowe protokołów są jednym ze sposobów ilustracji interfejsów klas

24. Podaj przykład diagramu UML maszyny stanowej zachowania dla prostego systemu (niekoniecznie informatycznego), omów elementy występujące na diagramie,

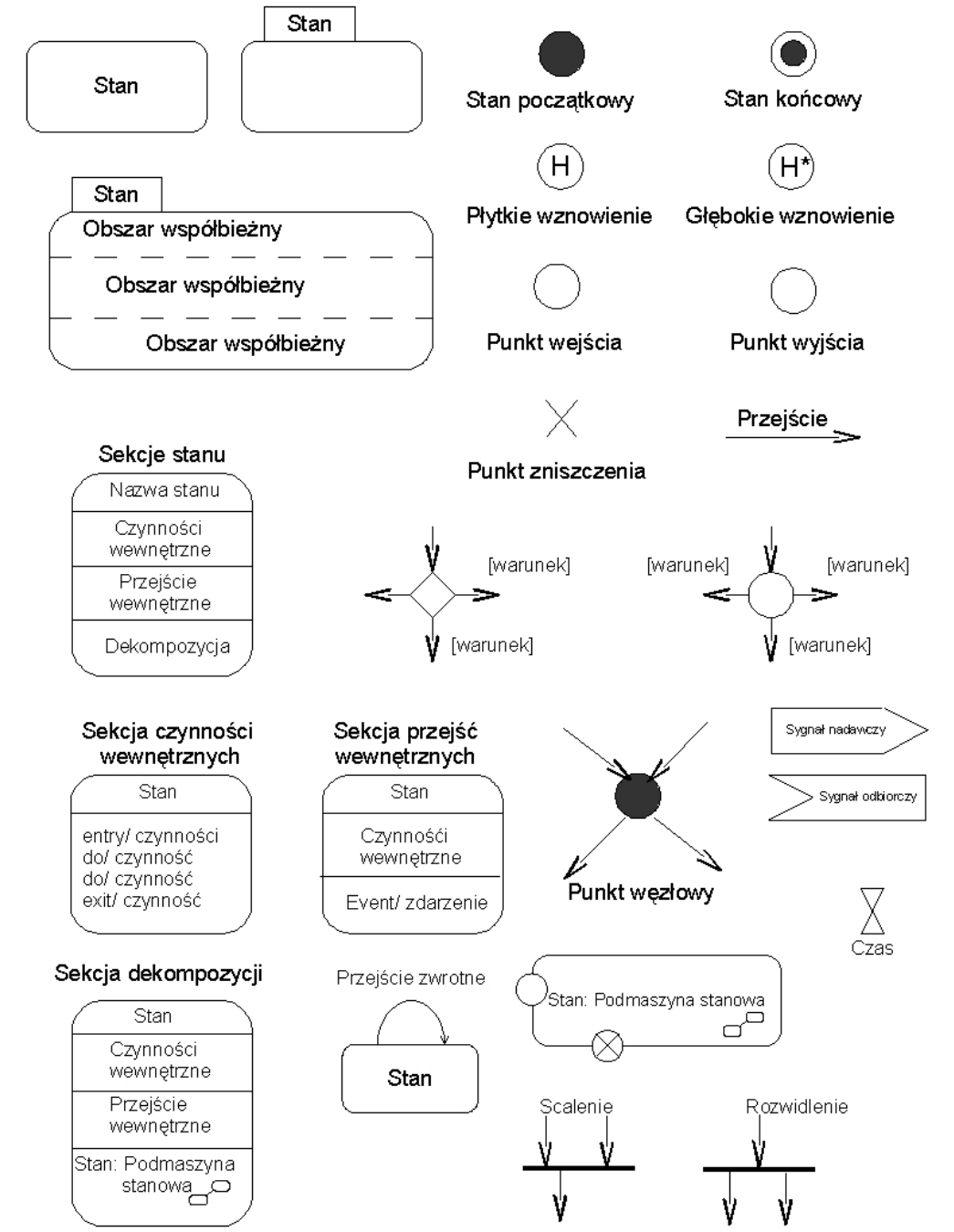
scharakteryzuj także elementy diagramów maszyny stanowej zachowania nie występujące
na przykładowym diagramie (dla każdego elementu, występującego lub nie na
przykładowym diagramie, przedstaw postać ogólną, możliwe warianty itp.; ważna jest
poprawność przykładowego diagramu i opisu elementów – mniej ważna sensowność
systemu)



W diagramach zachowania opis przejścia może mieć postać:

wyzwalacz [warunek sprawdzający]/czynność

- → wyzwalacz (*trigger*) jest tym co uruchamia przejście
- → czynność (*activity*) opisuje, co realizowane jest w trakcie przejścia
- → warunek sprawdzający *(*warunek dozoru, *guard)* musi być prawdziwy, aby doszło do przejścia (dzięki temu można wprowadzać sterowanie za pomocą warunków)



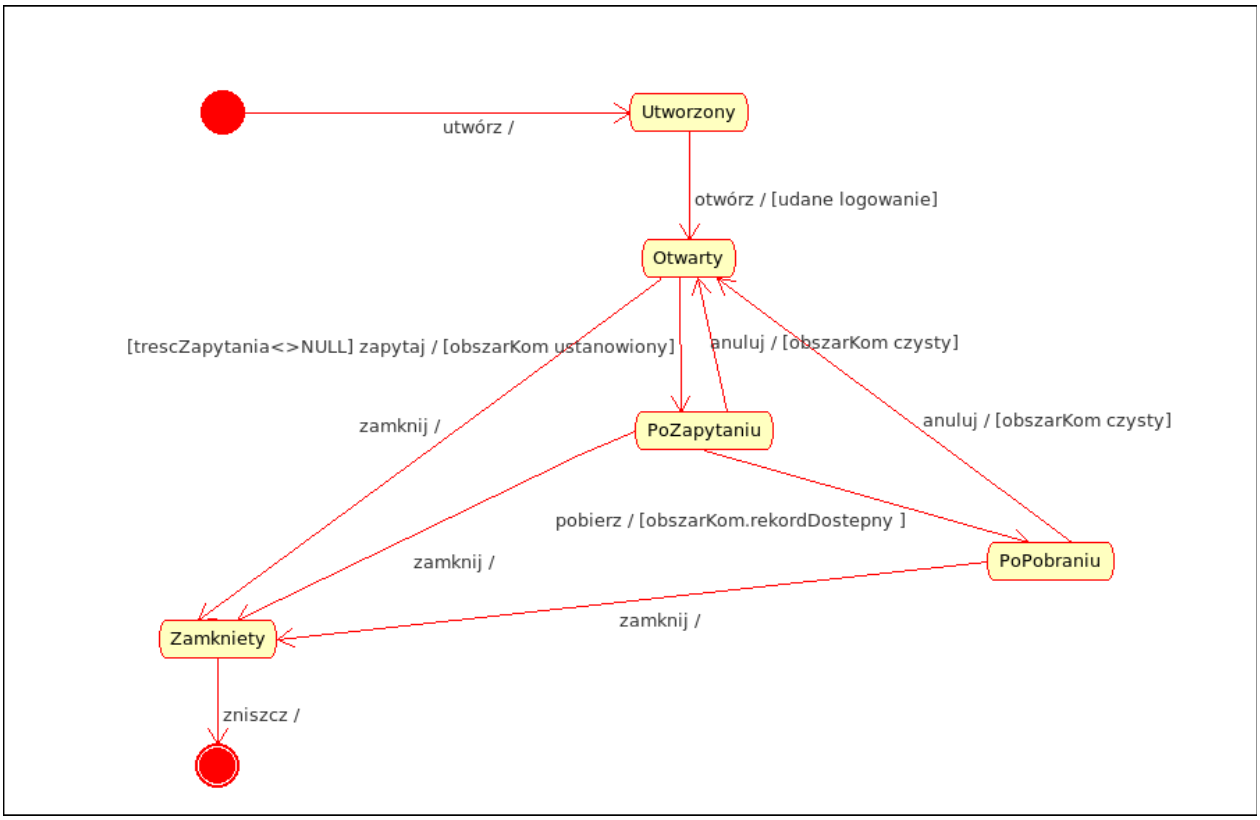
25. Podaj przykład diagramu UML maszyny stanowej protokołu dla obiektów pewnej klasy,

omów elementy występujące na diagramie, scharakteryzuj także elementy diagramów

maszyny stanowej protokołu nie występujące na przykładowym diagramie (dla każdego

elementu, występującego lub nie na przykładowym diagramie, przedstaw postać ogólną,
możliwe warianty itp.; ważna jest poprawność przykładowego diagramu i opisu
elementów – mniej ważna sensowność projektu klasy)

Elementy – j/w



26. Omów metodologię strukturalną tworzenia oprogramowania (w tym dekompozycję
funkcjonalną jako sposób analizy systemu); jakie są podstawowe zalety i wady
metodologii strukturalnej, czy metodologia ta może mieć zastosowanie także dzisiaj?

Metody strukturalne modelowania powstały w latach 70. XX wieku, jako element procesu rozwijania oprogramowania dostosowany do powszechnego w tych czasach programowania proceduralnego.

W metodach strukturalnych dominuje sposób analizy „od ogółu do szczegółu” (*topdown*)

Podstawowym sposobem podejścia do analizy jest dekompozycja funkcjonalna – próbuje się uchwycić podstawową funkcję systemu (jej dane wejściowe i dane wyjściowe), a następnie rozbić ją na sekwencję funkcji składowych, realizujących kolejne etapy przetwarzania

Proces dekompozycji przeprowadza się, aż do znalezienia funkcji nadających się do bezpośredniego kodowania

Zaletą dekompozycji funkcjonalnej jest jej stosunkowa prostota, szybkość realizacji i możliwość przeprowadzenia we współpracy z klientem

Wadami metodologii strukturalnej są:

- → silne uzależnienie elementów modelu od siebie (często operujących na pojedynczej strukturze danych)
- → niska elastyczność powstających modeli (specyfikacje kładą często nacisk na cechy specyficzne dla konkretnego projektu)
- → słabe uwzględnienie ponownego wykorzystania kodu (specyficzne wymagania projektu utrudniają korzystanie z gotowych komponentów)
- → trudność testowania systemu, który nie nadaje się do użycia dopóki wszystkie elementy nie są gotowe

Na skutek swoich specyficznych cech metodologia strukturalna uznawana jest dzisiaj za element pomocniczy przydatny np.:

- → jako sposób na szybkie, wspólnie z klientem, stworzenie wstępnego modelu systemu w fazie określania wymagań
- → jako sposób na projektowanie małych i średnich podsystemów o złożonym przetwarzaniu i prostej strukturze z jednym wejściem i jednym wyjściem
- → jako sposób tworzenia algorytmów

27. Podaj przykład diagramu czynności UML dla wybranego przez siebie systemu, omów
elementy występujące na diagramie; scharakteryzuj także elementy diagramów czynności
nie występujące na przykładowym diagramie (dla każdego elementu, występującego lub
nie na przykładowym diagramie, przedstaw postać ogólną, możliwe warianty itp.; ważna
jest poprawność przykładowego diagramu i opisu elementów – mniej ważna sensowność
projektu systemu)

Diagramy czynności są jednym z trzech podstawowych sposobów modelowania zachowania systemów w ramach specyfikacji UML

Diagramy czynności UML nadają się do opisu przepływu danych i sterowania, organizacji pracy (workflow), procesów biznesowych, a także proceduralnego sposobu przetwarzania (dane wejściowe, przetwarzanie, dane wyjściowe)

Diagramy czynności mogą być uważane za wariant, wprowadzający istotne rozszerzenia i modyfikacje, klasycznych schematów blokowych

Istotną cechą diagramów czynności jest fakt, że umożliwiają one modelowanie współbieżnie wykonywanych procesów

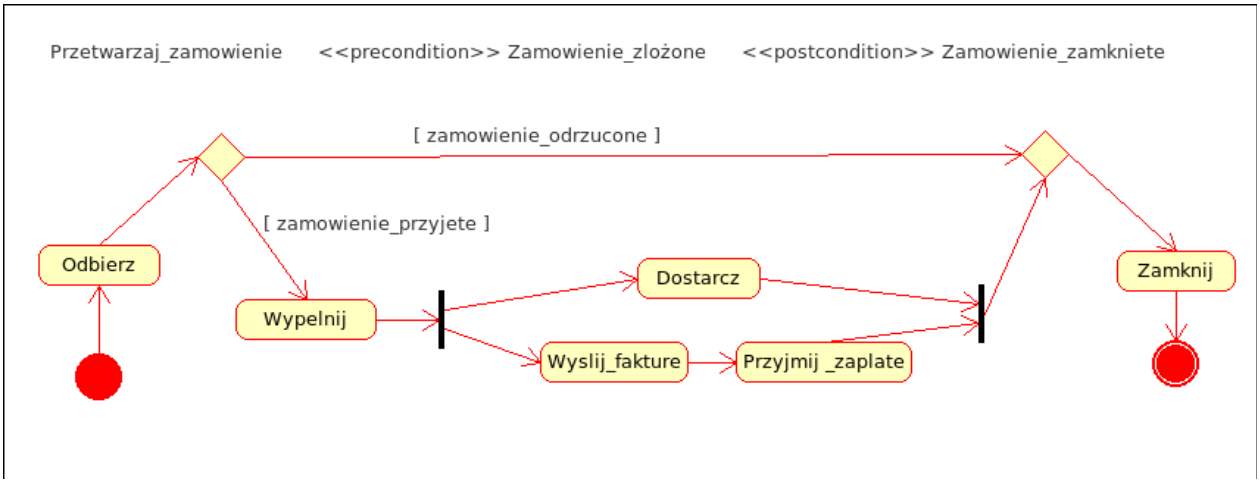
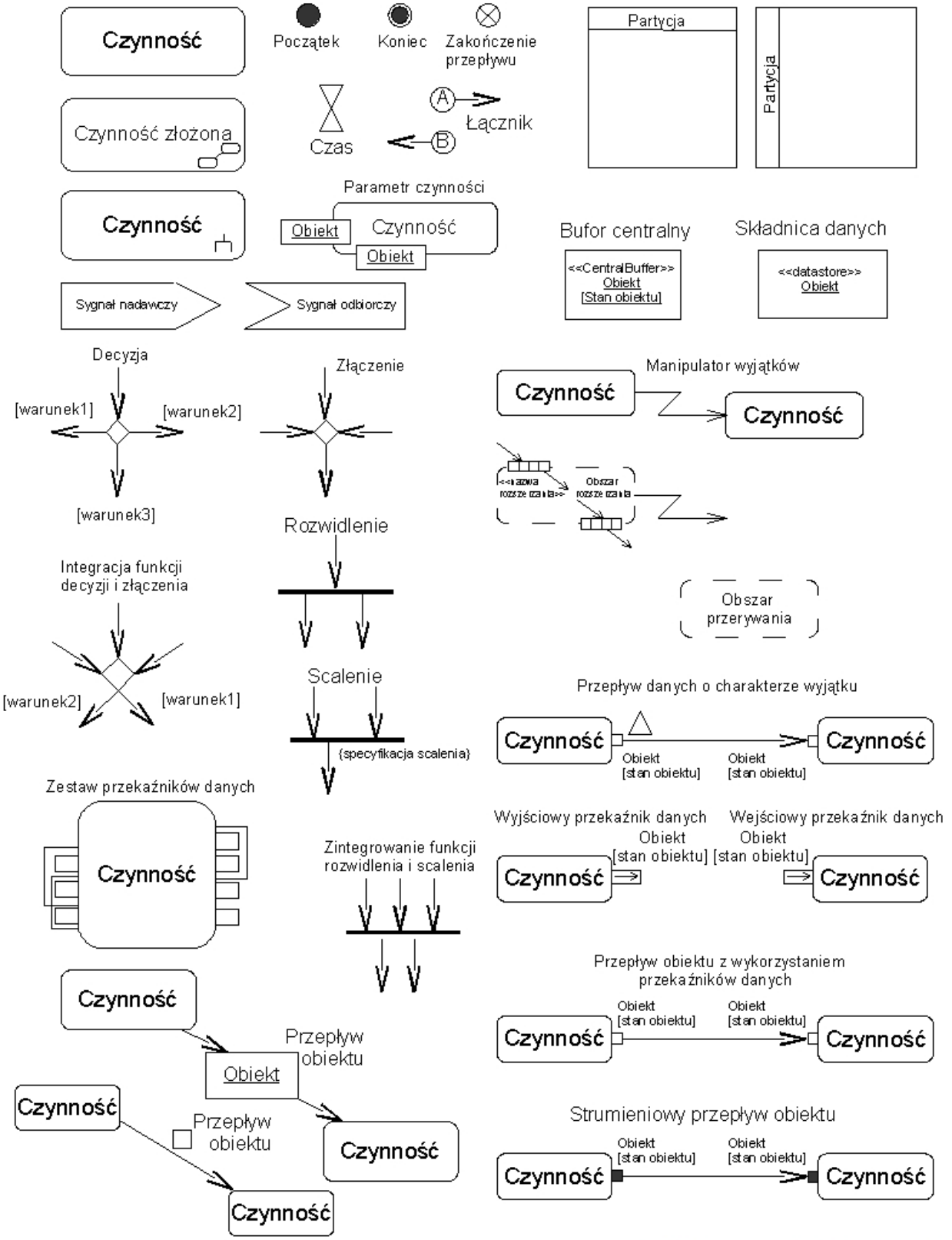

Przetwarzaj_zamowienie
<<precondition>> Zamowienie_złożone
<<postcondition>> Zamowienie_zamkniete
[zamowienie_odrzucone]
[zamowienie_przyjete]
Odbierz
Wypelnij
Dostarcz
Wyslij_fakture
Przyjmij _zaplate
Zamknij

Diagram czynności, przegląd notacji

Portal www.UML.com.pl



28. Jakie rodzaje węzłów sterowania znajdują się na diagramach czynności; jaka jest ich rola;

podaj przykład diagramu czynności z kilkoma rodzajami węzłów sterowania

węzły sterowania pozwalające wyrazić strukturę przepływów:

- → (*initial)* – węzeł rozpoczynający przepływy w momencie wywołania czynności
- → *(fork)* – rozdziela przepływ na współbieżne przepływy
- → *(join)* – łączy i synchronizuje wchodzące przepływy
- → *(decision)* – dokonuje wyboru między różnymi wychodzącymi przepływami
- → *(merge)* – łączy różne przepływy pochodzące z wyboru
- → (tylko jeden może być realizowany)
- → *(flow final)* – zakończenie pojedynczego przepływu
- → *(activity final)* – zakończenie czynności (wszystkich przepływów)

elementy – j/w

29. Krótko scharakteryzuj, koncentrując się głównie na różnicach, następujące elementy

strukturalne oprogramowania: podsystemy, moduły, komponenty, pakiety, klasy

Na najwyższym poziomie architektury system składa się z elementów, które bywają nazywane:

- → podsystemami – jeśli chce się zwrócić uwagę na podział całego systemu ze względu na funkcjonalność oraz na niezależność działania składników
- → modułami – jeśli chce się zwrócić uwagę na niezależne tworzenie składników, ich wymienność i wzajemne interakcje (wymianę komunikatów)
- → komponentami – jeśli chce się zwrócić uwagę na montowanie z gotowych składników (a także ścisłe określenie interfejsów)

Podsystem – część programu wydzielona ze względu na funkcjonalność oraz niezależność działania składników

Moduł – część programu tworzona niezależnie, komunikująca się z resztą systemu

Komponentem nazywany jest fragment oprogramowania nadający się do niezależnego montowania w większe programy. Komponent jest fragmentem nadającym się bezpośrednio do użycia, co zakłada że jest tworzony i kompilowany niezależnie od reszty programu (biblioteka w postaci skompilowanej jest komponentem, w postaci źródłowej nie). Ze względu na wymaganie współpracy z innymi komponentami tworzącymi całość systemu najważniejszą charakterystyką komponentu jest precyzyjnie zdefiniowany interfejs, określający usługi świadczone przez komponent. Komponent powinien realizować taki zestaw usług, aby dobrze nadawać się do ponownego użycia.

Klasa jest bytem precyzyjnie definiowanym w obiektowych językach programowania.

Pojedynczy pakiet składa się z pewnej liczby klas.

Różnica pomiędzy klasami i pakietami z jednej strony, a komponentami z drugiej, jest w głównej mierze różnicą perspektywy – komponent jest postrzegany z

perspektywy klienta, składającego oprogramowanie z przygotowanych modułów, natomiast klasy i pakiety są postrzegane przez programistę tworzącego kod obiektowy

30. Przedstaw i krótko omów modele sterowania systemami oraz typowe style (modele) architektury systemów

Modele sterowania:

- → centralny: wywołan i powrotów, zarzadcy i pracowników (głównie w przetwarzaniu współbieznym)
- → sterowany zdarzeniami: rozgłoszeniowy, sterowania przerwaniami (głównie RTS)

Przykładami stylów w architekturze są:

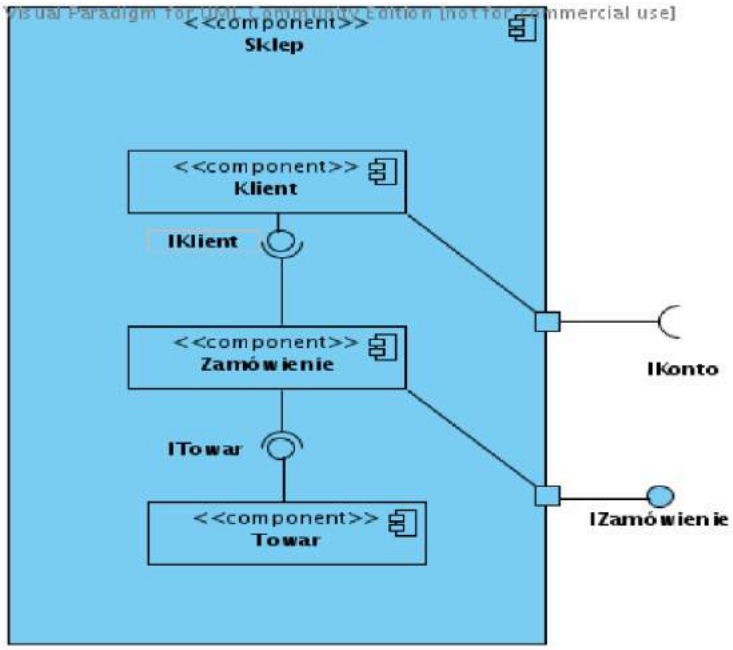
- → model z centralną bazą danych
- → model warstwowy
- → model klient-serwer
- → model przetwarzania potokowego
- → model obiektowy

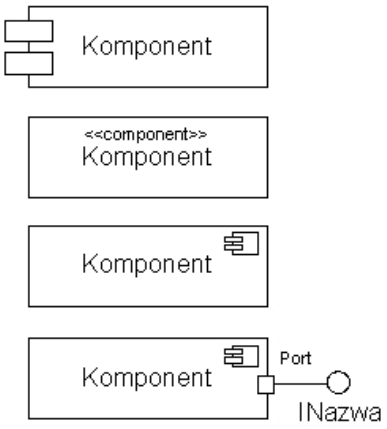
31. Podaj przykład diagramu komponentów UML dla wybranego przez siebie systemu, omów elementy występujące na diagramie; scharakteryzuj także elementy diagramów komponentów nie występujące na przykładowym diagramie (dla każdego elementu, występującego lub nie na przykładowym diagramie, przedstaw postać ogólną, możliwe warianty itp.; ważna jest poprawność przykładowego diagramu i opisu elementów – mniej ważna sensowność projektu systemu)

Z punktu widzenia tworzenia oprogramowania komponent może być rozumiany podobnie jak klasa (klasa traktowana jako abstrakcyjny typ danych – czyli poprzez realizowany interfejs, z ukrytą implementacją)

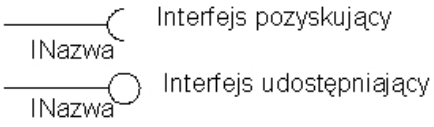
Z tego względu na diagramach komponentów UML 2.0 oznacza się je podobnym symbolem co klasy na diagramach klas – prostokątem, z tym, że w prostokącie umieszcza się specjalny symbol – prostokąt z dwoma wypustkami (dawniej w UML 1.0 oznaczający komponent)

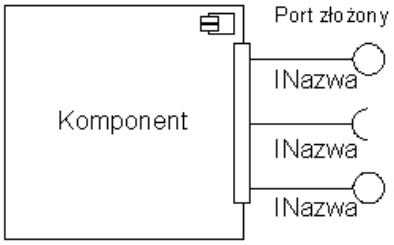
Komponent natomiast jest czymś innym niż fizyczny nośnik oprogramowania (plik binarny, konfiguracyjny lub tp.) – te ostatnie w UML 2.0 nazywane są artefaktami (choć często artefakty są fizycznymi pojemnikami na komponenty)

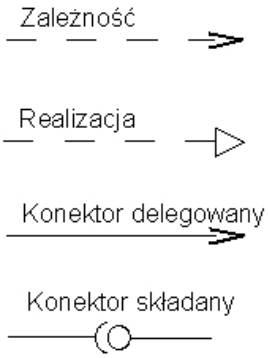


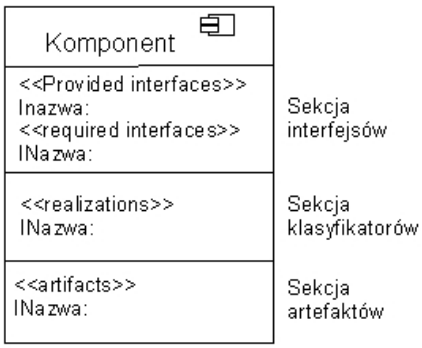


Stereotypy komponentów:
<<subsystem>>
<<table>>
<<executable>>
<<OS>>
<<library>>
<<service>>









32. Co nazywamy obiektem, co klasą i czym jest oprogramowanie w pełni obiektowe?

Oprogramowaniem w pełni obiektowym nazywane będzie oprogramowanie, w którym wszystkie byty istniejące w trakcie wykonywania programu są albo obiektami, albo składowymi obiektów lub klas.

Obiektem nazywany będzie byt istniejący w trakcie wykonywania programu posiadający tożsamość, stan i zachowanie

Klasą nazywana będzie częściowa lub całkowita definicja obiektów

Z punktu widzenia programu w trakcie jego wykonania, klasa może być także określana jako zbiór obiektów o tych samych stanach i zachowaniu (co jednak nie obejmuje np. klas abstrakcyjnych)

33. Omów krótko (najlepiej podając także proste przykłady) trzy podstawowe cechy oprogramowania obiektowego

Ukrywanie informacji (obudowywanie, *encapsulation*) – obiekty ukrywają jak najwięcej informacji o swoim stanie i zachowaniu (zwłaszcza informacji dotyczących implementacji), udostępniając na zewnątrz tylko minimum konieczne do korzystania z ich usług

Dziedziczenie (*inheritance*) – obiekty tworzy się na podstawie definicji zawartych w klasach, z tym że klasy mogą wykorzystywać inne klasy do dostarczenia fragmentów definicji (korzystając przy tym z klasyfikowania na typy i podtypy), ułatwia to programowanie i zwiększa stopień ponownego użycia kodu

Polimorfizm (*polymorphism*) – pewne nazwy stosowane w kodzie mogą odpowiadać kilku różnym bytom (np. obiektom, metodom), decyzji o wyborze odpowiedniego bytu dokonuje się automatycznie w trakcie kompilacji lub w trakcie wykonania programu

34. Scharakteryzuj techniki ustalania struktury klas dla systemów obiektowych

Spośród wielu technik analizy prowadzących do identyfikacji klas na uwagę zasługują m.in.:

Analiza zmienności i wariantów – jeżeli przy realizacji danej funkcji pojawiają się warianty (instrukcje warunkowe) lub jeśli przewidujemy, że pewien fragment kodu może ulec modyfikacji (np. przez wprowadzenie nowych rodzajów obiektów) należy zadać sobie pytanie czy nie byłoby możliwe odseparowanie realizowanej funkcji od wariantów i zmienności, tzn. ukrycie (obudowanie) odkrytej zmienności za interfejsem klasy, tak aby funkcja zawsze była taka sama, natomiast zmieniał się tylko typ obiektów realizujących interfejs

Delegowanie odpowiedzialności – projektując realizację interfejsu należy rozważyć, czy nie istnieje inny typ obiektu (klasa), który lepiej zrealizuje daną funkcję interfejsu i oddelegować wykonanie tej funkcji do nowej klasy

Być może najważniejszym elementem, który należy uwzględnić przy poszukiwaniu optymalnej dla danego systemu hierarchii klas są istniejące już systemy realizujące podobne funkcje do systemu projektowanego oraz wnioski płynące z analizy ich funkcjonowania

Poszczególne dziedziny zastosowań wypracowują własne typowe zestawy klas (okienkowe interfejsy użytkownika; systemy wspomagające podstawowe funkcje

przedsiębiorstwa: księgowość, kadry, płace itp.; systemy edycji i przetwarzania dokumentów)

Rozróżnienie rodzajów dziedziczenia

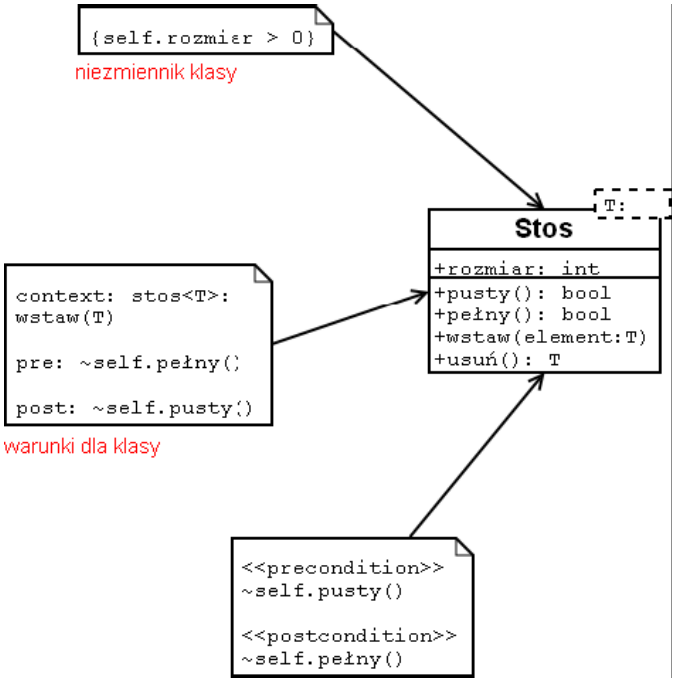
Po stworzeniu wstępnego projektu klas istotna jest jego weryfikacja mająca na celu uzyskanie możliwie najprostszej, czytelnej i funkcjonalnej hierarchii klas, nadającej się dodatkowo do modyfikacji, rozszerzeń oraz powtórnego użycia konkretnych klas

35. Scharakteryzuj klasy jako abstrakcyjne typy danych oraz przedstaw idee projektowania kontraktowego (*design by contract*); jak można umieszczać logiczne warunki wynikające z projektowania kontraktowego na diagramach klas UML? Podaj przykład diagramu UML dla pojedynczej klasy umieszczając na nim elementy związane z projektowaniem kontraktowym

Klasa powinna by realizacją dobrze zdefiniowanego, abstrakcyjnego typu danych (łączy typy danych i funkcje na nich operujące). Powinna mieć precyzyjny interfejs i być przystosowana do wielokrotnego użycia. Z każdą klasą wiąże się zbiór precyzyjnie określonych asercji – zdań zawsze prawdziwych dla każdego obiektu. Asercje dzielą się na:

- → warunki początkowe (*precondition*) – dla każdej metody
- → warunki końcowe (*postcondition*) – dla każdej metody
- → niezmienniki (*invariants*) – dla każdej klasy

Na schemacie UML asercje umieszczamy w notatkach:



Warunki początkowe określają za co odpowiedzialny jest wywołujący operację – awaria wynikła z niespełnienia warunku początkowego przy wywołaniu operacji nie obciąża twórcy klasy

Warunki końcowe określają za co odpowiedzialny jest twórca kodu klasy

Precyzyjne ustalenie kto za co odpowiada (będące rodzajem jednoznacznej umowy między twórcą i użytkownikiem klasy) nie tylko rozstrzyga kwestie sporne, ale także nakłada obowiązek dokonywania sprawdzeń (uwaga: tylko na jedną ze stron)

36. W jaki sposób projektowanie kontraktowe uściśla pojęcie wyjątku, a w jaki precyzuje warunki poprawności dziedziczenia, tak aby spełniona była zasada podstawialności (*substitution principle*)?

W projektowaniu kontraktowym wyjątek definiowany jest jako zdarzenie uniemożliwiające poprawne zakończenie operacji, mimo spełnienia jej warunku początkowego

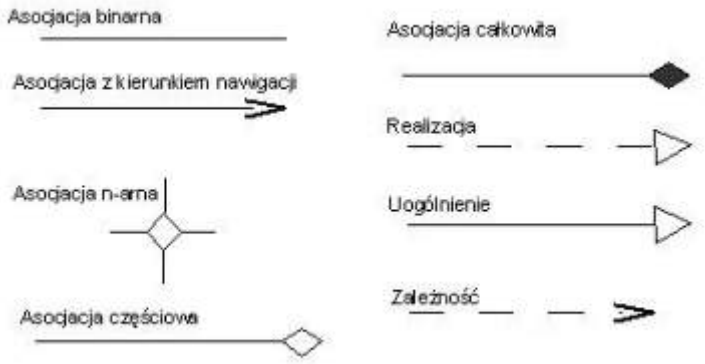
Projektowanie kontraktowe może istotnie pomóc w zagwarantowaniu poprawności kodu stosującego dziedziczenie. Ze względu na to, że każdy obiekt klasy pochodnej powinien móc być użyty w miejsce obiektu klasy podstawowej, metody klasy pochodnej muszą:

→ mieć takie same lub słabsze warunki początkowe (wymagać tyle samo lub mniej od użytkownika)

→ mieć takie same lub mocniejsze warunki końcowe (wymagać tyle samo lub więcej od siebie)
→ mieć takie same lub mocniejsze niezmienniki (wymagać tyle samo lub więcej od siebie)

37. Czym różnią się między sobą następujące relacje pomiędzy klasami: powiązania (*associations*), kompozycje (*compositions*), generalizacje(uogólnienia)/specjalizacje (*generalizations/specializations*), zależności (*dependencies*); jak zaznacza się je na diagramach UML i w jaki sposób można dokonać implementacji powyższych związków w konkretnych językach programowania

Związki pomiędzy klasami są oznaczane liniami, które mogą, opcjonalnie, mieć strzałkę wskazującą kierunek relacji. Końce związków mogą także być oznaczone krotnościami: mówią one o tym, ile obiektów danej klasy może brać udział w danej relacji. Na przykład istnieje jeden katalog, ale może się nim posługiwać dowolna liczba pracowników biblioteki (co oznaczamy 0...*)



Powiązania/asocjacje (associations)

Asocjacja to związek strukturalny, który wskazuje, że obiekty jednej klasy są połączone z obiektami innej klasy. Asocjacja pomiędzy dwoma klasami oznacza, że przejście od obiektu jednej z klas do obiektu drugiej oraz odwrotnie jest możliwe. Jeśli połączenie dotyczy dokładnie dwu klas, to taką asocjację nazywamy dwuargumentową. Na diagramie asocjacja jest przedstawiona jako linia ciągła

Kompozycje/agregacje (compositions/aggregations)

Agregacja to związek typu „całość-część”, w którym jedna z klas reprezentuje „całość”, zaś pozostała klasa lub klasy reprezentują „część”. Innymi słowy klasa reprezentująca całość składa sie z pozostałych klas. Agregacja to szczególny przypadek asocjacji. Prosta postać agregacji ma znaczenie pojęciowe, jak również nie wyznacza zależności między czasem życia „całości” a czasem życia „części”

Generalizacje (uogólnienia)/specjalizacje (generalizations/specializations)

Generalizacja w ujęciu UML odpowiada ogólnie przyjętej definicji. Jest to związek występujący między bardziej ogólnym elementem (nadklasa, rodzic) a bardziej szczegółowym elementem (podklasa, dziecko) w pełni zgodnym z nadrzędnym i zawierającym ponadto dodatkowe informacje czy własności. Generalizacja posiada predefiniowane komplementarne pary ograniczeń, overlapping i disjoint (względem rozłączności podklas) oraz complete i incomplete (względem wyróżnienia wszystkich podklas). Przydatnym elementem notacji jest dyskryminator pozwalający określić ze względu na jaką własność ma miejsce związek generalizacji np.: drzewo → dąb, brzoza, sosna (dyskryminator = gatunek)

Zależności (dependencies)

Zależność to związek użycia. Zmiany dokonane w specyfikacji jednego elementu klasy (czyli obiektu), np. należącego do klasy Okno, może mieć wpływ na inny element, który używa tego pierwszego, ale nie koniecznie odwrotnie. Z zależności należy korzystać wtedy, kiedy chce się podkreślić, że jeden element używa drugiego. Najczęściej mamy do czynienia z zależnościami pomiędzy klasami, dla pokazania, że jedna klasa używa drugiej jako argumentu w sygnaturze operacji. Zmiany wprowadzone w danej klasie mogą mieć wpływ na operacje innej klasy.

38. Omów rozróżnienie dziedziczenia na dziedziczenie interfejsu, implementacji oraz interfejsu i implementacji; jakie mechanizmy w konkretnych językach programowania wspierają wymienione rodzaje dziedziczenia

Dziedziczenie jest relacją między klasami, zachodzącą wtedy gdy jakaś klasa (klasa pochodna) korzysta w swej definicji z definicji innej klasy (klasy bazowej, podstawowej). Można wyobrażać sobie, że obiekty klasy pochodnej zawierają w sobie elementy klasy podstawowej

Podstawowe rozróżnienie rodzajów dziedziczenia:

- → interfejsu – dziedziczonymi elementami są wyłącznie sygnatury funkcji składowych klasy
- → interfejsu i implementacji – dziedziczone są dane składowe (stan), sygnatury i implementacja metod
- → implementacji (nie zalecane) – dziedziczone jest wszystko jak wyżej, ale klientom na zewnątrz nie udostępnia się ani danych składowych (dobrze), ani funkcji składowych z interfejsu (kontrowersyjne)

39. Jakie zmiany powinna wnosić klasa pochodna w stosunku do klasy podstawowej, aby uzasadnić wprowadzenie dziedziczenia do kodu; w jakich sytuacjach stosować dziedziczenie, a w jakich składanie (kompozycję) obiektów?

Mając pewną klasę warto próbować dokonać specjalizacji i stworzyć podklasy:

- → gdy pojawiają się różniące się metodami rodzaje obiektów i wystarczająco duża liczba metod jest inna dla wyróżnionych rodzajów (podklas)

→ gdy specjalizacja umożliwi ponowne użycie klasy bazowej w innych kontekstach (klasa bazowa uwolniona od pewnych konkretów, przejdzie na wyższy poziom abstrakcji)

Klasa pochodna powinna wnosić co najmniej jedną z poniższych zmian:

→ nowe dane lub funkcje składowe

→ redefinicja istniejących funkcji składowych (redefinicja może oznaczać dodanie implementacji lub zmianę widzialności metody)

→ zmiana niezmienników klasy

Dziedziczenie jest podobne do składania obiektów, należy je stosować wtedy gdy:

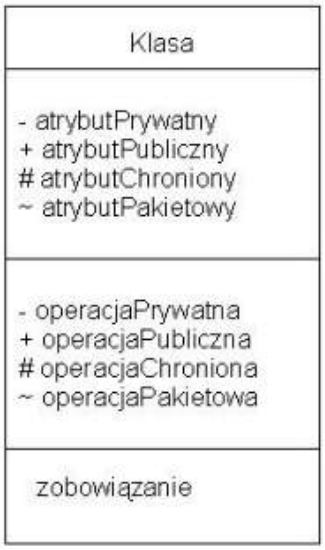
→ chcemy używać obiektów klas pochodnych, dla których typ określać będzie klasa podstawowa (np. będziemy posługiwać się wskaźnikiem do obiektu typu klasy podstawowej)

→ chcemy wprowadzić nową klasę, która częściowo modyfikuje zachowanie lub własności innej klasy, przy czym większość atrybutów i funkcji starej klasy pozostaje bez zmian (zwłaszcza jeśli chcemy pozostawić możliwość stosowania starej klasy)

→ nie spodziewamy się, aby obiekt klasy pochodnej miał kiedykolwiek w trakcie swojego istnienia dokonać zmiany atrybutów i zachowania związanych z klasą podstawową (w przeciwnym przypadku – gdy spodziewamy się wymiany składowych związanych z klasą podstawową na inne składowe – należy użyć bardziej elastycznej kompozycji obiektów)

40. Podaj przykład symbolu klasy z diagramów UML z pełną specyfikacją (zawierającą wszystkie możliwe charakterystyki) atrybutów (danych składowych) klasy; jaka jest postać ogólna specyfikacji atrybutu, omów jej poszczególne elementy

42. Podaj przykład symbolu klasy z diagramów UML z pełną specyfikacją (zawierającą wszystkie możliwe charakterystyki) metod (funkcji składowych) klasy; jaka jest postać ogólna specyfikacji metody, omów jej poszczególne elementy



Klasa – nazwa klasy

budowa atrybutu:

widoczność nazwa: typ wielokrotność = wartość_domyślna {listawłasności}

widoczność (-/+/#/~)

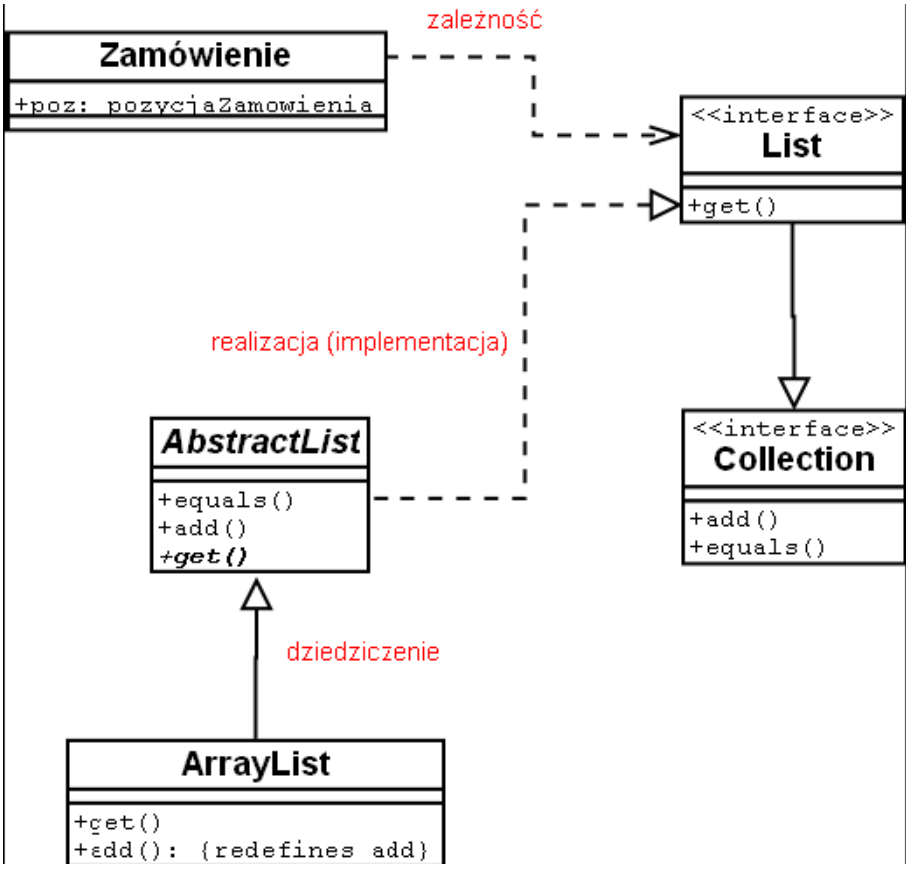
typ – typ danych

wielokrotność – określa liczbę (1, 0..1, n..m, 1..*, *, itd.)

budowa operacji:

widoczność nazwa (parametry) : zwracany_typ {listawłasności}
parametry: kierunek nazwa: typ = wartość_domyślna
kierunek (In, out, inout)

41. Przedstaw przykład diagramu UML, na którym pojawią się powiązania (w tym kompozycje) między klasami; umieść jak najwięcej symboli podających szczegóły powiązań między klasami (ilustrujących charakterystyki atrybutów klas)



43. Przedstaw przykład diagramu UML dla klasy parametryzowanej i jej konkretnej realizacji

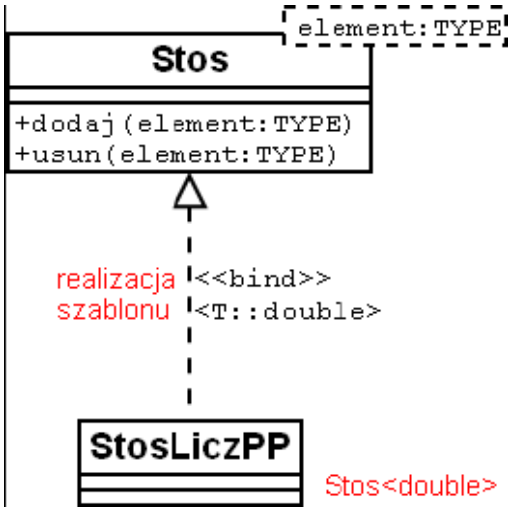
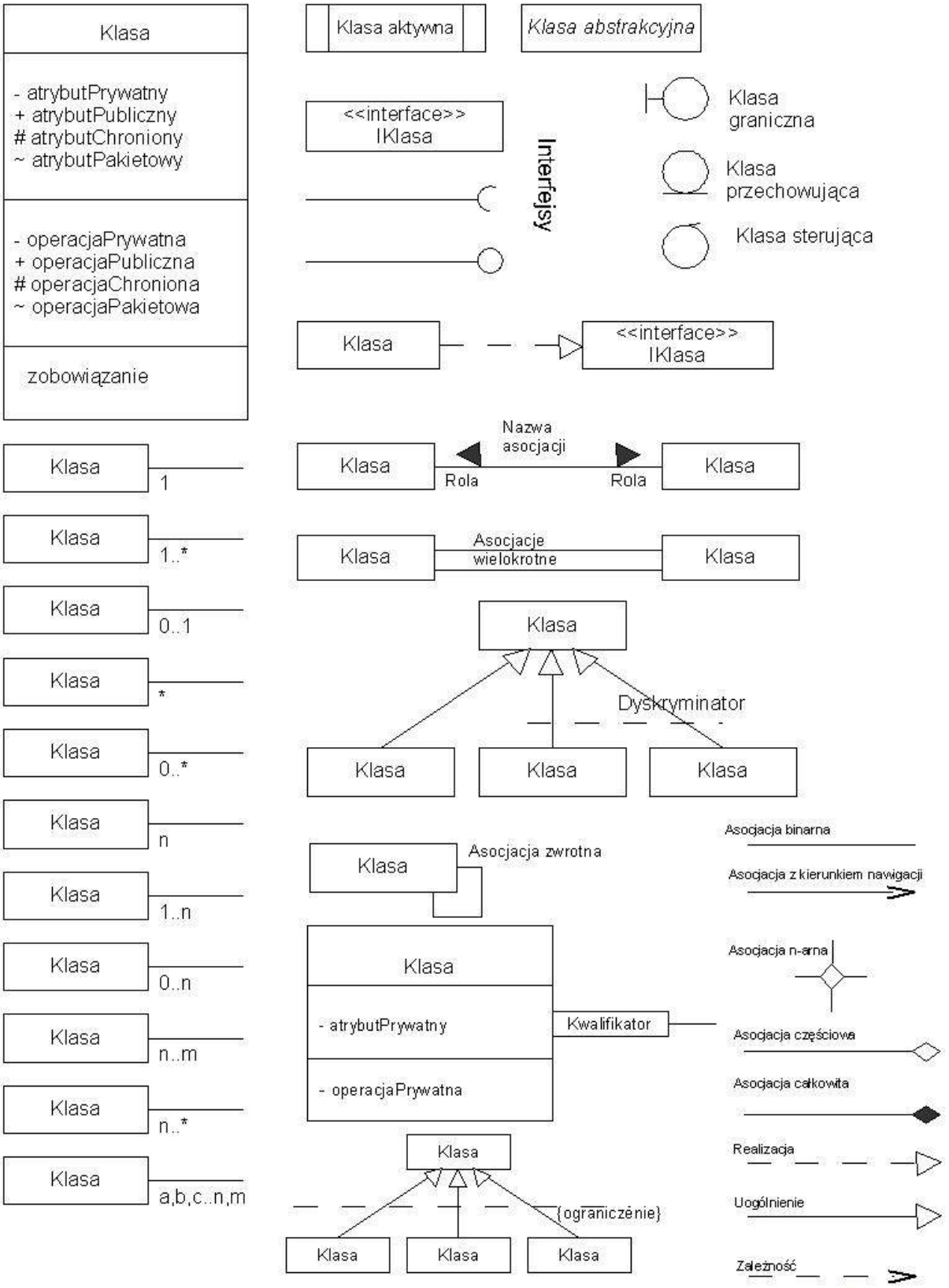


Diagram klas, przegląd notacji

Portal www.UML.com.pl



44. Wymień typowe pomocnicze funkcje składowe związane z funkcjonowaniem obiektów w
programach napisanych w podstawowych językach obiektowych

Typowe funkcje pomocnicze w niektórych nowoczesnych językach obiektowych (np. C#) mogą zostać zastąpione przez tzw. settery i gettery. Charakterystyczne funkcje pomocnicze dla klas to element *get()* czy *void set(element)*, ustawiające wartości prywatnych zmiennych klasy. Implementacja w języku C# polega na ustawieniu getterów i setterów dla zmiennych prywatnych np.

```
private int _x;

public int x
{
	get { return this._x; }
	set { this._x = value; }
}
```

45. Podaj przykład diagramu obiektów UML dla wybranego przez siebie fragmentu systemu,
omów elementy występujące na diagramie; scharakteryzuj także elementy diagramów
obiektów nie występujące na przykładowym diagramie (dla każdego elementu,
występującego lub nie na przykładowym diagramie, przedstaw postać ogólną, możliwe
warianty itp.; ważna jest poprawność przykładowego diagramu i opisu elementów – mniej
ważna sensowność projektu systemu)

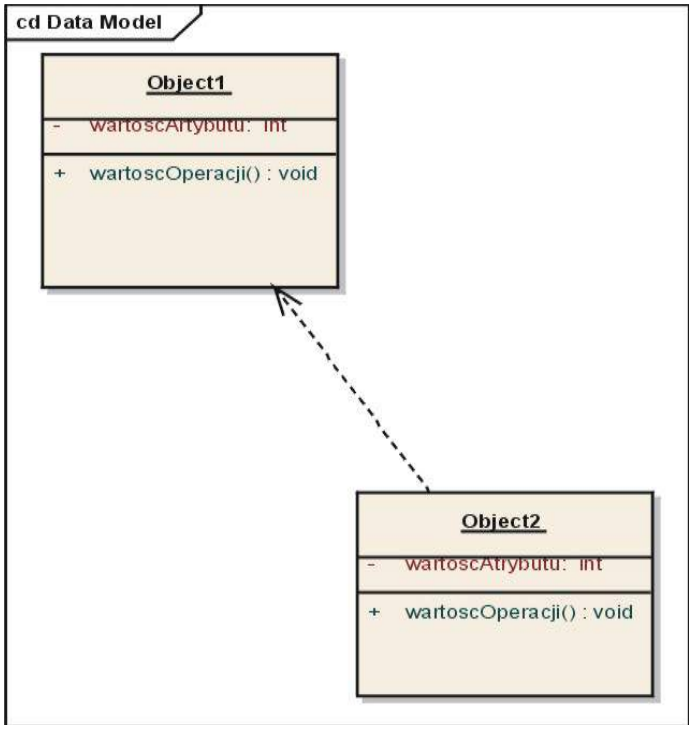
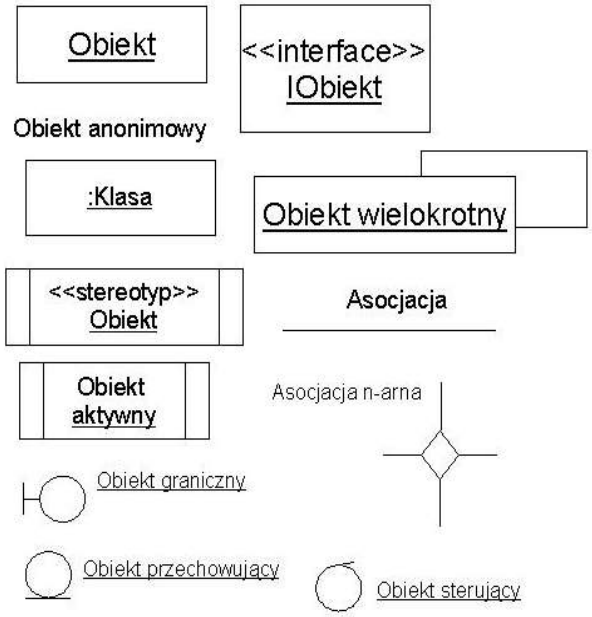


Diagram obiektów Portal www.UML.com.pl



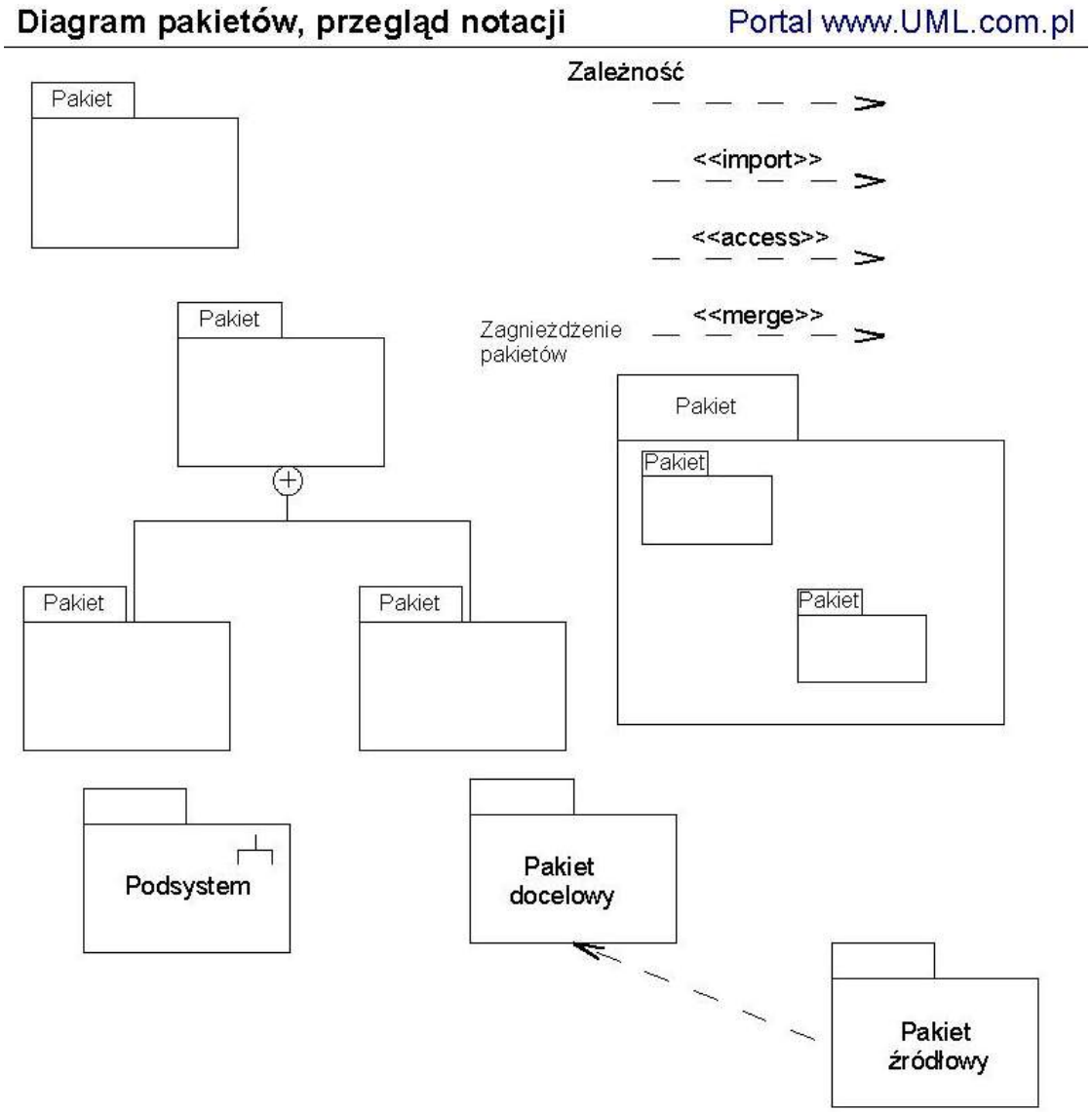
46. Jak jest rola diagramów pakietów UML, jakie elementy i jakie związki najczęściej prezentują?

Diagram pakietów pozwala grupować elementy diagramów

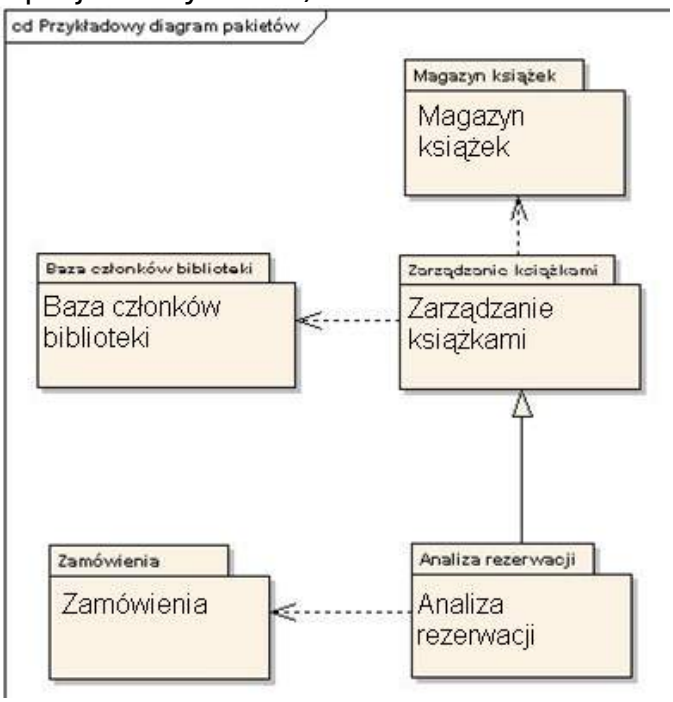
Pakiety mogą mieć przypisane odpowiednie zależności pomiędzy sobą

Bardzo istotne w dużych systemach. Pozwalają na łatwiejszą kontrolę zależności między głównymi elementami systemu

Diagram pakietów służy do tego, by uporządkować strukturę zależności w systemie, który ma bardzo wiele klas, przypadków użycia itp. Przyjmujemy, ze pakiet zawiera w sobie wiele elementów, które opisują jakieś w miarę dobrze określone zadanie. Na diagramie umieszczamy pakiety i wskazujemy na zależności między nimi. Dzięki temu dostajemy na jednym diagramie obraz całości, bądź dużego fragmentu, systemu.



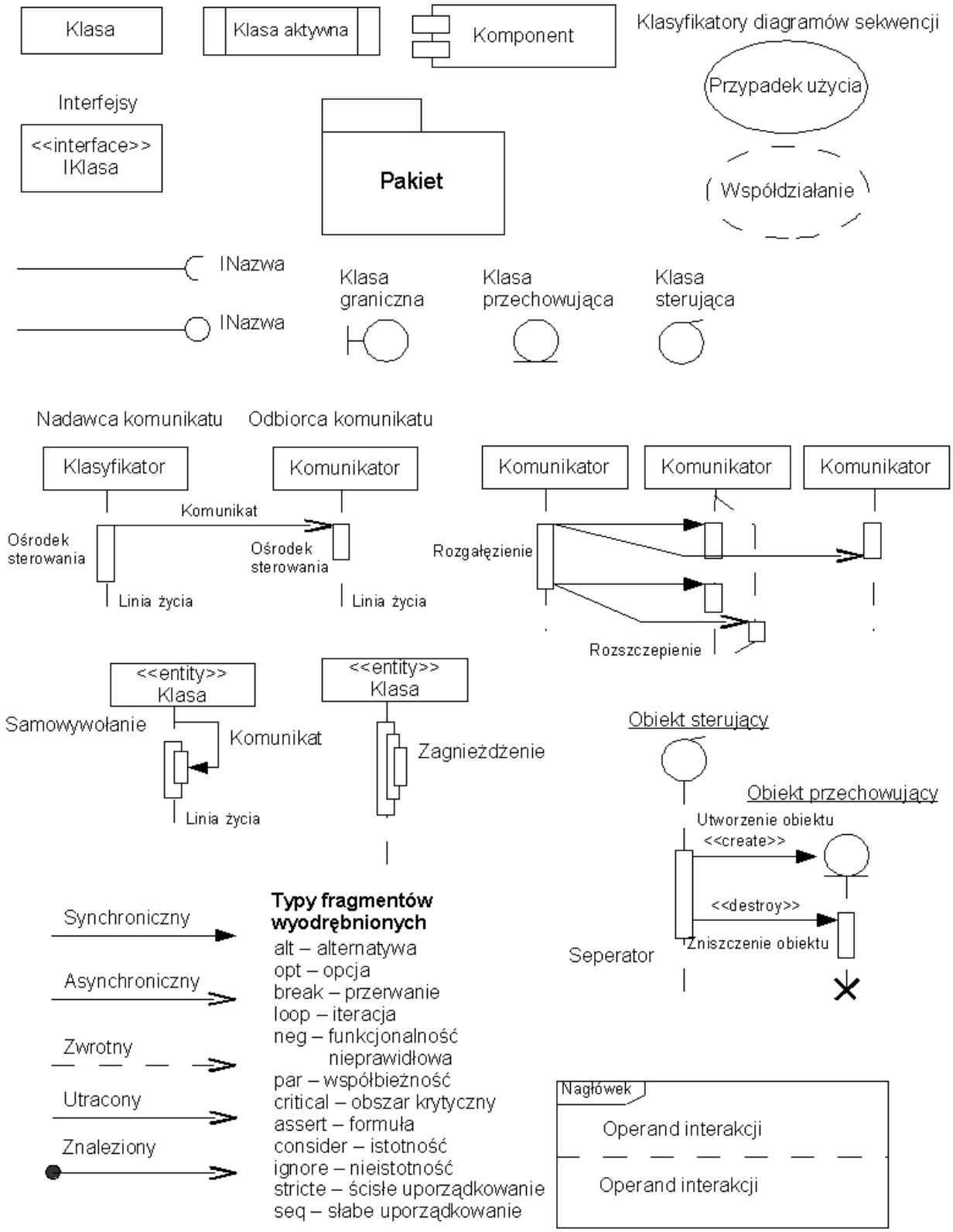
47. Podaj przykład diagramu pakietów UML dla wybranego przez siebie fragmentu systemu,
omów elementy występujące na diagramie; scharakteryzuj także elementy diagramów
pakietów nie występujące na przykładowym diagramie (dla każdego elementu,
występującego lub nie na przykładowym diagramie, przedstaw postać ogólną, możliwe
warianty itp.; ważna jest poprawność przykładowego diagramu i opisu elementów – mniej
ważna sensowność projektu systemu)



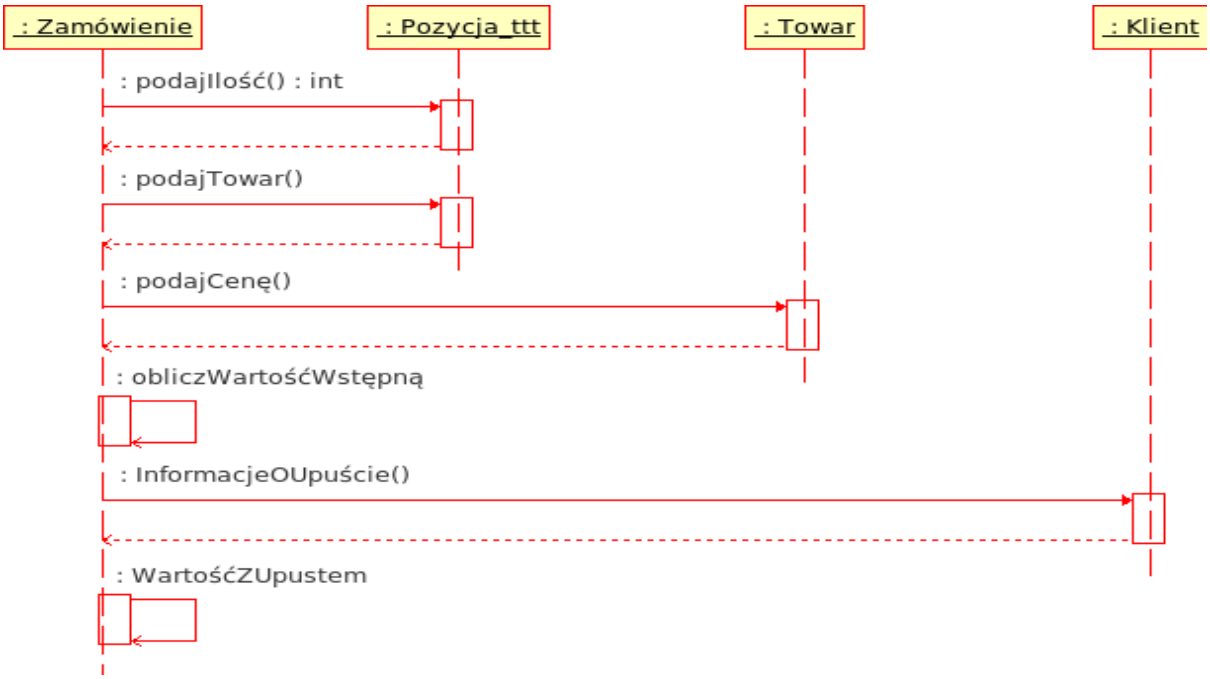
*Nazwy mogą być w nagłówkach (lepiej) lub w środku (opcja)

Diagram sekwencji, przegląd notacji

Portal www.UML.com.pl



48. Podaj przykład diagramu sekwencji (przebiegu) UML dla funkcjonowania wybranego
przez siebie fragmentu systemu, omów elementy występujące na diagramie; scharakteryzuj także elementy diagramów sekwencji nie występujące na przykładowym
diagramie (dla każdego elementu, występującego lub nie na przykładowym diagramie,
przedstaw postać ogólną, możliwe warianty itp.; ważna jest poprawność przykładowego
diagramu i opisu elementów – mniej ważna sensowność projektu systemu)



49. Podaj przykład diagramu komunikacji UML dla funkcjonowania wybranego przez siebie
fragmentu systemu, omów elementy występujące na diagramie; scharakteryzuj także
elementy diagramów komunikacji nie występujące na przykładowym diagramie (dla
każdego elementu, występującego lub nie na przykładowym diagramie, przedstaw postać
ogólną, możliwe warianty itp.; ważna jest poprawność przykładowego diagramu i opisu
elementów – mniej ważna sensowność projektu systemu)

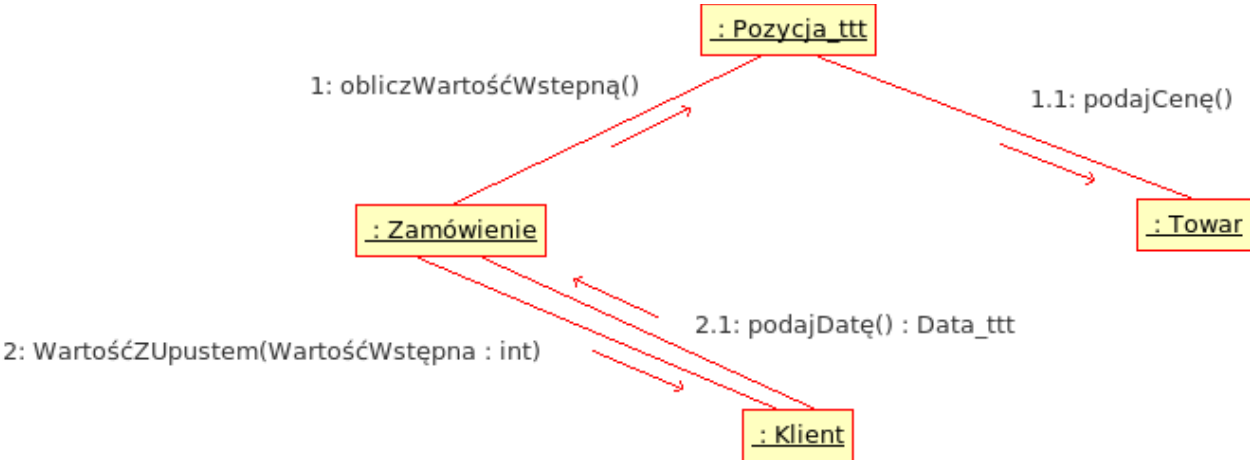
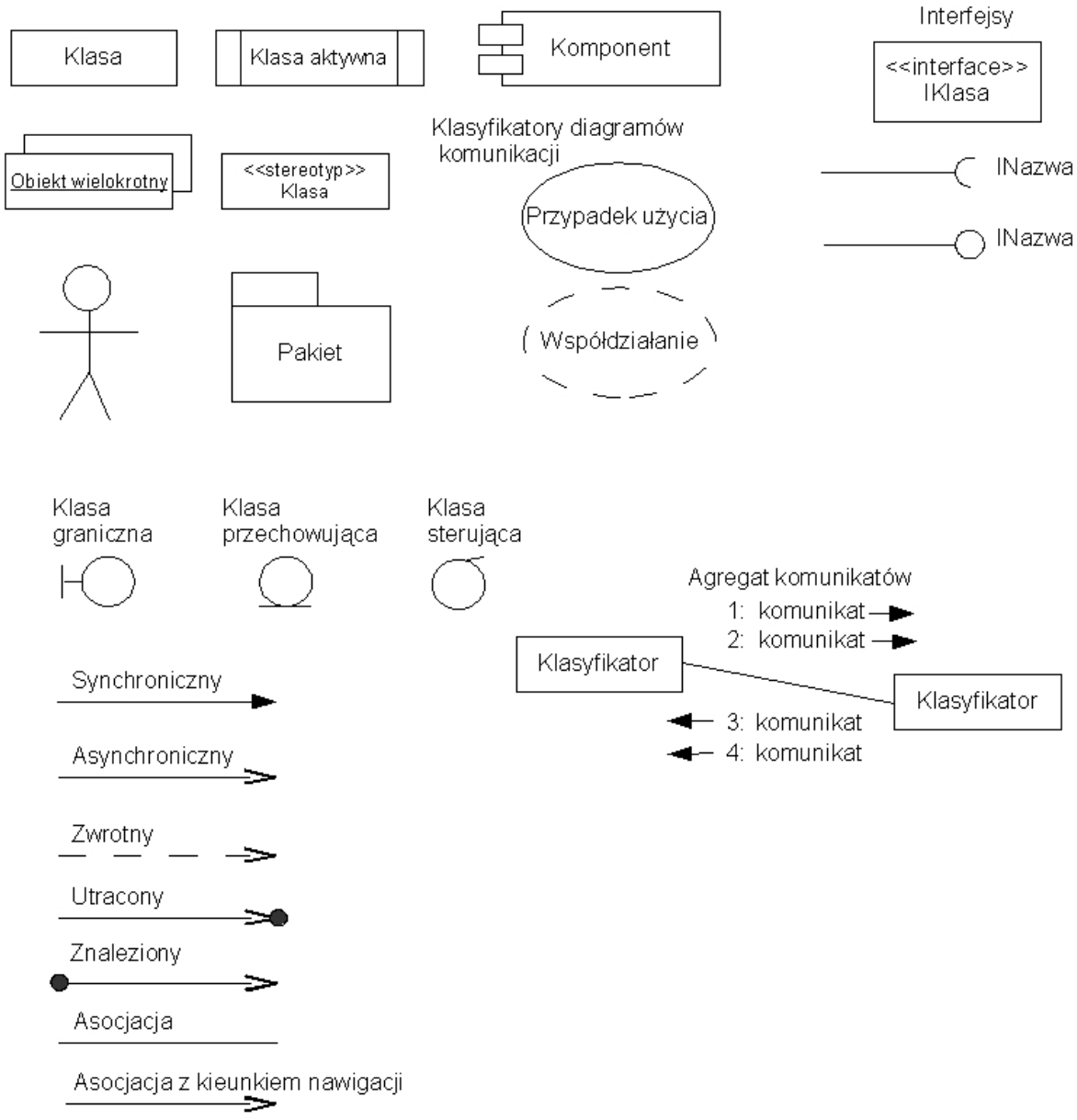

: Pozycja_ttt
1: obliczWartośćWstepną()
1.1: podajCenę()
: Zamówienie
: Towar
2.1: podajDatę() : Data_ttt
2: WartośćZUpustem(WartośćWstępna : int)
: Klient

Diagram komunikacji, przegląd notacji

Portal www.UML.com.pl



50. Wymień i krótko scharakteryzuj sposoby (poziomy) ponownego wykorzystania kodu

Ponowne użycie kodu może być realizowane na różnych poziomach:

→ *copy and paste* – najgorsze i najpopularniejsze rozwiązanie, często gwałci zasadę „raz i tylko raz" („*once and only once"),* stanowi źródło błędów

→ techniki programistyczne – dziedziczenie, programowanie generyczne, stosowanie WebServices, programowanie aspektowe

→ komponenty, biblioteki i zestawy narzędziowe (*toolboxes*) – elementy umieszczane w programach

→ szkielety (*frameworks*) – oprogramowanie wymagające uzupełnienia o pewne elementy dla uzyskania konkretnej, pełnej funkcjonalności

→ całe, gotowe programy – ponowne wykorzystanie ze zmienioną konfiguracją (np. programy ERP – SAP itp.) lub w ramach nowego środowiska (np. linie produktów dla różnych systemów operacyjnych)

51. Podaj wady i zalety tworzenia oprogramowania z ponownym wykorzystaniem kodu

Zalety:

→ redukuje koszt i czas tworzenia oprogramowania (mniej elementów musi zostać wyspecyfikowanych, zaprojektowanych, zaimplementowanych i przetestowanych)

→ zwiększa niezawodność systemów (ponownie używany kod zazwyczaj został już wszechstronnie sprawdzony i przetestowany)

→ zmniejsza ryzyko związane z tworzeniem oprogramowania (w przypadku nowo tworzonego kodu zawsze mogą pojawić się nieprzewidywane problemy)

→ popularne komponenty wielokrotnego użytku tworzą standardy, które ułatwiają tworzenie i stosowanie programów

→ w oparciu o elementy wielokrotnego użycia można tworzyć generatory programów lub istotnych podsystemów (np. interfejsów użytkownika, parserów, kodu na bazie modelu UML)

Wady:

→ nie istnieją jeszcze sposoby łatwego wyszukiwania pasującego istniejącego kodu do zastosowania w tworzonych programach

→ zanim zastosuje się gotowy kod należy go zrozumieć (może to być trudne i kosztowne, zaś brak zrozumienia może prowadzić do rozmaitych problemów)

→ zastosowanie gotowego kodu może wymagać istotnego dostosowania reszty systemu

→ testowanie całości systemu może także być utrudnione, jeśli ponownie użyty kod w wielu miejscach przenika się z kodem wytwarzanym

→ w fazie użytkowania i konserwacji, przy modyfikacjach i rozszerzeniach systemu jest się uzależnionym od kodu, co do którego nie wiadomo czy będzie ewoluował, a jeśli tak to w jakim kierunku

52. Zdefiniuj czym jest komponent, jakie są podstawowe cechy komponentów

Komponent zazwyczaj nie jest czymś co deklaruje lub definiuje się w kodzie (choć istnieją języki posługujące się tym pojęciem)

Komponent pojawia się, gdy myślimy o konsolidacji programu wykonywalnego z uprzednio przygotowanych składników

Aby komponent mógł być w pełni wymienialny (np. w trakcie działania programu) nie powinien posiadać stanu

Komponent zazwyczaj zawiera pewien zbiór klas i jeden lub kilka pakietów

Różnica pomiędzy klasami i pakietami z jednej strony, a komponentami z drugiej, jest w głównej mierze różnicą perspektywy – komponent jest postrzegany z perspektywy klienta, składającego oprogramowanie z przygotowanych modułów, natomiast klasy i pakiety są postrzegane przez programistę tworzącego kod obiektowy. W takim ujęciu komponenty (będące bytami abstrakcyjnymi) naturalnie odwzorowują się na artefakty – fizyczne nośniki oprogramowania instalowane na sprzęcie
komputerowym

53. W jaki sposób wykorzystanie komponentów zmienia proces wytwarzania oprogramowania w porównaniu do wariantu tradycyjnego (bez komponentów)

Tworząc oprogramowanie oparte na komponentach należy zmodyfikować podstawowe czynności:

- → w trakcie ustalania wymagań należy wyszukać komponenty nadające się do wykorzystania w systemie, przeprowadzić ich walidację i w przypadku zaakceptowania dostosować wymagania do zastosowanych komponentów
- → podobnie w fazie projektowania uwzględnia się już wybrane komponenty, a także inne nie wpływające na wymagania, natomiast wykorzystywane w implementacji systemu
- → w fazie implementacji należy uwzględnić odpowiednio zwiększone zasoby przeznaczone na integrację komponentów (dostosowanie do ich interfejsów)
- → w każdej z faz należy uwzględnić ryzyko, że komponent okaże się jednak nieodpowiedni co będzie wymagało zmian – w implementacji, może architekturze, a może także w wymaganiach

54. Jakich zasad należy przestrzegać tworząc komponenty wielokrotnego użycia?

Tworząc komponenty należy starać się aby:

- → realizować w nich funkcjonalność, która będzie opierać się próbie czasu (ponowne użycie oznacza użycie w pewnym przedziale czasu) – funkcjonalność odpowiadającą „stabilnym abstrakcjom z dziedziny zastosowań”

→ uwzględnić w interfejsie wszelkie aspekty użycia komponentu (czyli w praktyce przewidzieć wszelkie możliwe sposoby korzystania z dostarczanej przez komponent funkcjonalności), a jednak nie skomplikować komponentu tak, by jego użycie stało się zbyt trudne
→ uniezależnić stosowanie komponentu od zmienności sprzętu i oprogramowania systemowego
→ umożliwić stosowanie w różnych sytuacjach poprzez dodatkowe konfigurowanie komponentu przed użyciem
→ uczynić komponent maksymalnie samowystarczalnym (np. dołączyć stosowane biblioteki do komponentu)

55. Omów elementy praktycznie stosowanych środowisk komponentowych, podaj przykłady takich środowisk

W praktyce zastosowanie komponentów opiera się dodatkowo na istnieniu:
→ standardów specyfikacji komponentów i wymiany informacji miedzy komponentami
→ środowisk (platform) integracji komponentów (oprogramowania warstwy pośredniej – *middleware*) umożliwiających integrację komponentów i wymianę komunikatów między nimi

Przykładami środowisk (modeli) komponentowych są CORBA, Enterprise Java Beans – EJB, (D)COM(+)

Środowiska integracji komponentów często umożliwiają rozproszone funkcjonowanie systemów komponentowych

Rozwijaną w ostatnich latach alternatywą dla rozproszonych systemów komponentowych jest stosowanie WebServices, usług internetowych

Kompozycja komponentów dokonywana jest zazwyczaj w ramach konkretnego modelu (środowiska) i staje się problemem głównie technicznym (zakładając, że dostarczone komponenty prawidłowo funkcjonują w ramach środowiska)

Środowisko może samo dostarczać dodatkowe funkcje związane z obsługą komponentów (zapewnienie bezpieczeństwa, współbieżności, obsługi transakcji itp.)

Istotnym problemem przy kompozycji jest obsługa błędów i sytuacji wyjątkowych (częściowo wspierana przez model komponentowy) – trzeba pamiętać, że w praktyce w konkretnym systemie wykorzystywana jest tylko część funkcjonalności dostarczanej przez komponent

Dla popularyzacji komponentów istotnym elementem mogłoby stać się certyfikowanie przez cieszące się zaufaniem instytucje, gwarantujące poprawność, efektywność i bezpieczeństwo stosowania komponentów (nie jest to jednak proste do zrealizowania)

Elementy: serwer usług, aplikacja kliencka, middleware

56. Podaj wady i zalety wykorzystania komponentów w tworzeniu oprogramowania

Wady:
→ nie istnieją jeszcze sposoby łatwego wyszukiwania pasującego istniejącego kodu do zastosowania w tworzonych programach

→ zanim zastosuje się gotowy kod należy go zrozumieć (może to być trudne i kosztowne, zaś brak zrozumienia może prowadzić do rozmaitych problemów)
→ zastosowanie gotowego kodu może wymagać istotnego dostosowania reszty systemu
→ testowanie całości systemu może także być utrudnione, jeśli ponownie użyty kod w wielu miejscach przenika się z kodem wytwarzanym
→ w fazie użytkowania i konserwacji, przy modyfikacjach i rozszerzeniach systemu jest się uzależnionym od kodu, co do którego nie wiadomo czy będzie ewoluował, a jeśli tak to w jakim kierunku
→ obsługa błędów i sytuacji wyjątkowych – trzeba pamiętać, że w praktyce w konkretnym systemie wykorzystana jest tylko część funkcjonalności dostarczanej przez komponent
→ zaufanie do komponentów, które specyfikowane przez swoje interfejsy są czarnymi skrzynkami, które mogą kryć w swym wnętrzu nieprzyjemne niespodzianki (niezgodność ze specyfikacją, niespełnianie wymagań niefunkcjonalnych, nie mówiąc o zawieraniu szpiegów i wirusów)

Zalety:

→ redukuje koszt i czas tworzenia oprogramowania (mniej elementów musi zostać wyspecyfikowanych, zaprojektowanych, zaimplementowanych i przetestowanych)
→ zwiększa niezawodność systemów (ponownie używany kod zazwyczaj został już wszechstronnie sprawdzony i przetestowany)
→ zmniejsza ryzyko związane z tworzeniem oprogramowania (w przypadku nowo tworzonego kodu zawsze mogą pojawić się nieprzewidywane problemy)
→ popularne komponenty wielokrotnego użytku tworzą standardy, które ułatwiają tworzenie i stosowanie programów
→ w oparciu o elementy wielokrotnego użycia można tworzyć generatory programów lub istotnych podsystemów (np. interfejsów użytkownika, parserów, kodu na bazie modelu UML)
→ środowisko komponentowe może samo dostarczać dodatkowe funkcje związane z obsługą komponentów (zapewnienie bezpieczeństwa, współbieżności, obsługi transakcji itp.)

57. Scharakteryzuj czym są wzorce projektowe, jaki jest standard prezentacji wzorców?
(wystarczy podać podstawowe elementy standardowego opisu)
58. Podaj przykład wzorca (w postaci opisu zbliżonego do przyjętych standardów oraz
diagramu UML)
59. Jakie zasady programowania (w szczególności obiektowego) propagują wzorce
60. Na czym polega refaktoryzacja i jakie ma ona znaczenie dla wytwarzania kodu
61. Scharakteryzuj programowanie aspektowe – motywacje do jego stosowania, sposoby
realizacji, zalety i wady

62. Podaj elementy typowego środowiska RAD (Rapid Application Development)
63. Scharakteryzuj RAD jako metodologię programowania, podaj jej wady i zalety
64. Podaj i zdefiniuj miary niezawodności systemów informatycznych
65. Omów sposoby gwarantowania niezawodności oprogramowania
66. Omów sposoby uzyskiwania odporności na błędy (*fault tolerance*) – scharakteryzuj
podstawowe zasady ogólne oraz przedstaw typowe właściwości systemów odpornych na
błędy
67. Przedstaw zasady „programowania defensywnego (ostrożnego)” programowania
ukierunkowanego na unikanie błędów
68. Jaka jest różnica pomiędzy walidacją (atestacją) a weryfikacją; scharakteryzuj jedną i
drugą
69. Na czym polega inspekcja kodu, w jaki sposób jest przeprowadzana i kto w niej
uczestniczy?
70. Przedstaw możliwe klasyfikacje testów oraz scharakteryzuj poszczególne rodzaje testów
71. Z czego składa się typowe „wydanie” (*release*) programu?
72. Omów elementy procesu wdrożenia
73. Podaj przykład diagramu wdrożenia UML dla wybranego przez siebie systemu, omów
elementy występujące na diagramie; scharakteryzuj także elementy diagramów wdrożenia
nie występujące na przykładowym diagramie (dla każdego elementu, występującego lub
nie na przykładowym diagramie, przedstaw postać ogólną, możliwe warianty itp.; ważna
jest poprawność przykładowego diagramu i opisu elementów – mniej ważna sensowność
projektu systemu)
74. Scharakteryzuj czynności wchodzące w skład konserwacji oprogramowania; jakie są
podstawowe przyczyny wymuszające modyfikacje oprogramowania?
75. Omów cechy charakterystyczne ewolucji dużych systemów informatycznych
76. Jaka jest specyfika oprogramowania jako produktu w odróżnieniu od produktów z innych
dziedzin inżynierii
77. Przedstaw podstawowe czynności składające się na zarządzanie projektem

78. Wymień podstawowe rodzaje ryzyka związane z realizacją projektów informatycznych
79. Z jakich elementów składa się całkowity koszt realizacji przedsięwzięcia informatycznego
80. Czym można posługiwać się szacując koszt wytwarzania oprogramowania; omów poszczególne możliwości
81. Omów podstawowe zasady oraz wady i zalety algorytmicznego szacowania kosztu realizacji projektu informatycznego (np. na podstawie modelu COCOMO)
82. Wymień zasady zarządzania jakością przy wytwarzaniu oprogramowania
83. Wymień elementy składające się na całościową dokumentację projektu informatycznego
84. Wymień elementy oraz pożądane cechy dokumentacji dostarczanej użytkownikowi oprogramowania
85. Podaj krótką charakterystykę (na czym polega i do czego służy) model CMM (*Capability Maturity Model*)
86. Omów podstawowe cechy „Ujednoliconego Procesu” wytwarzania oprogramowania w jego najważniejszej wersji RUP (*Rational Unified Process*)
87. Scharakteryzuj podstawowe zasady Programowania Ekstremalnego (*Extreme Programming*), jakie są jego wady i zalety
88. Przedstaw zasady manifestu metod zwinnych (*Agile Manifesto*), do realizacji jakich projektów informatycznych szczególnie nadają sie metody zwinne