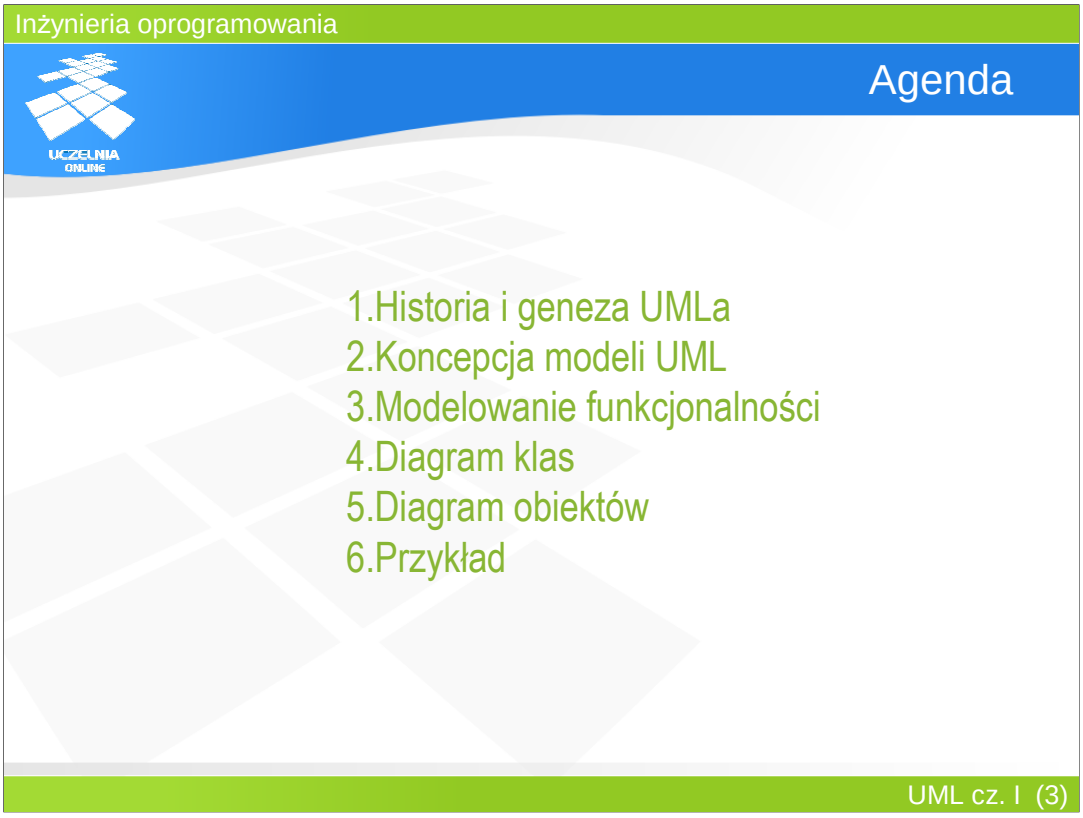
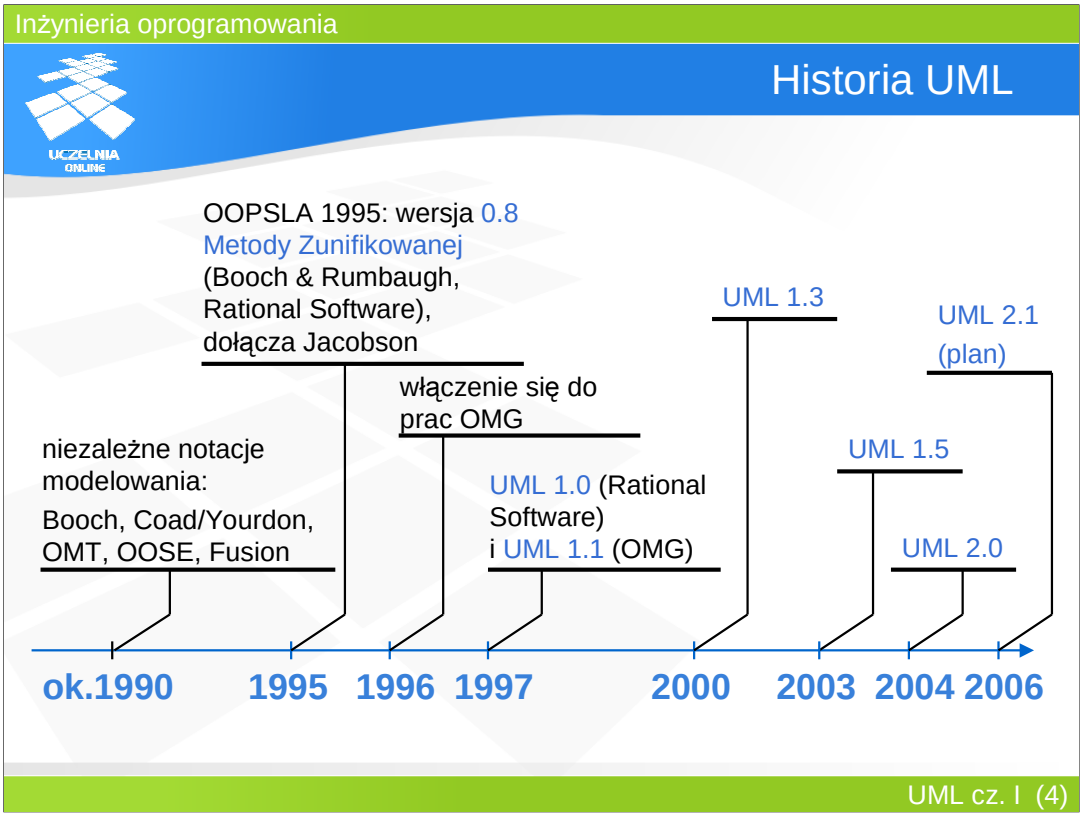

Inżynieria oprogramowania
UML cz. I
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE

Wykład jest pierwszym z dwóch, poświęconych językowi modelowania UML.

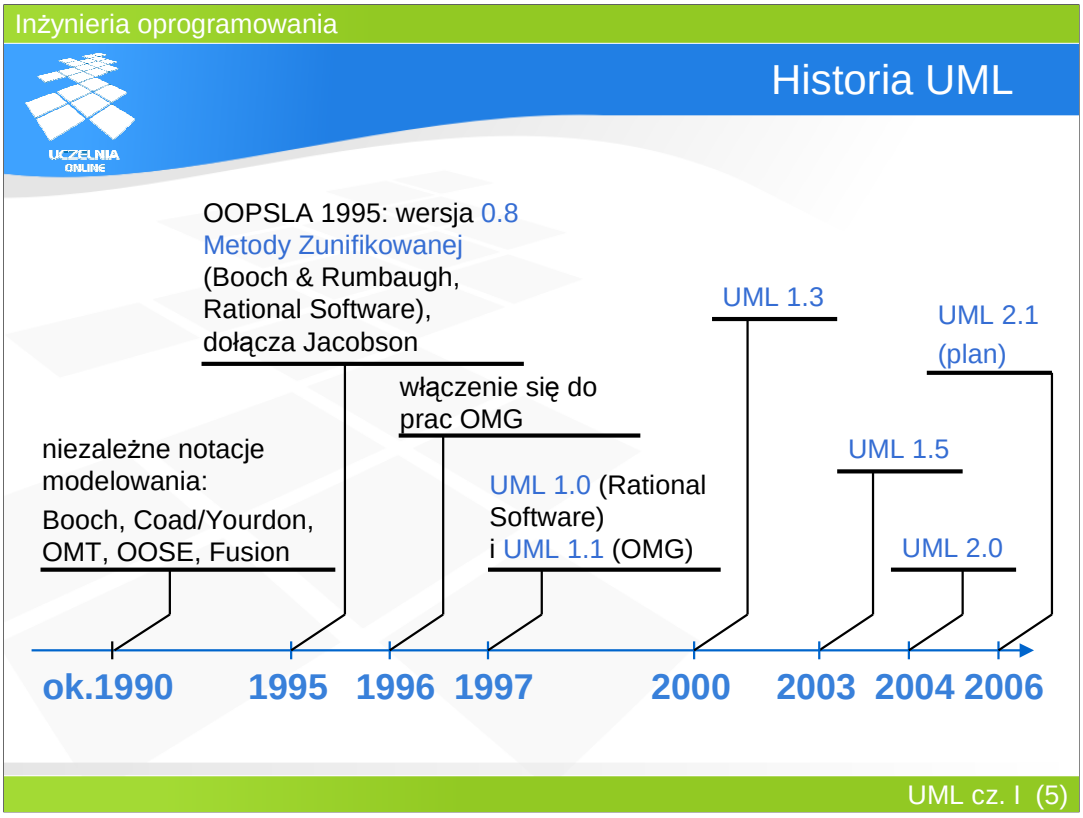


Podczas tego wykładu zostanie przedstawione wprowadzenie do języka UML, jego geneza oraz struktura. Omówione będą także dwie perspektywy UML: przypadków użycia oraz logiczna.



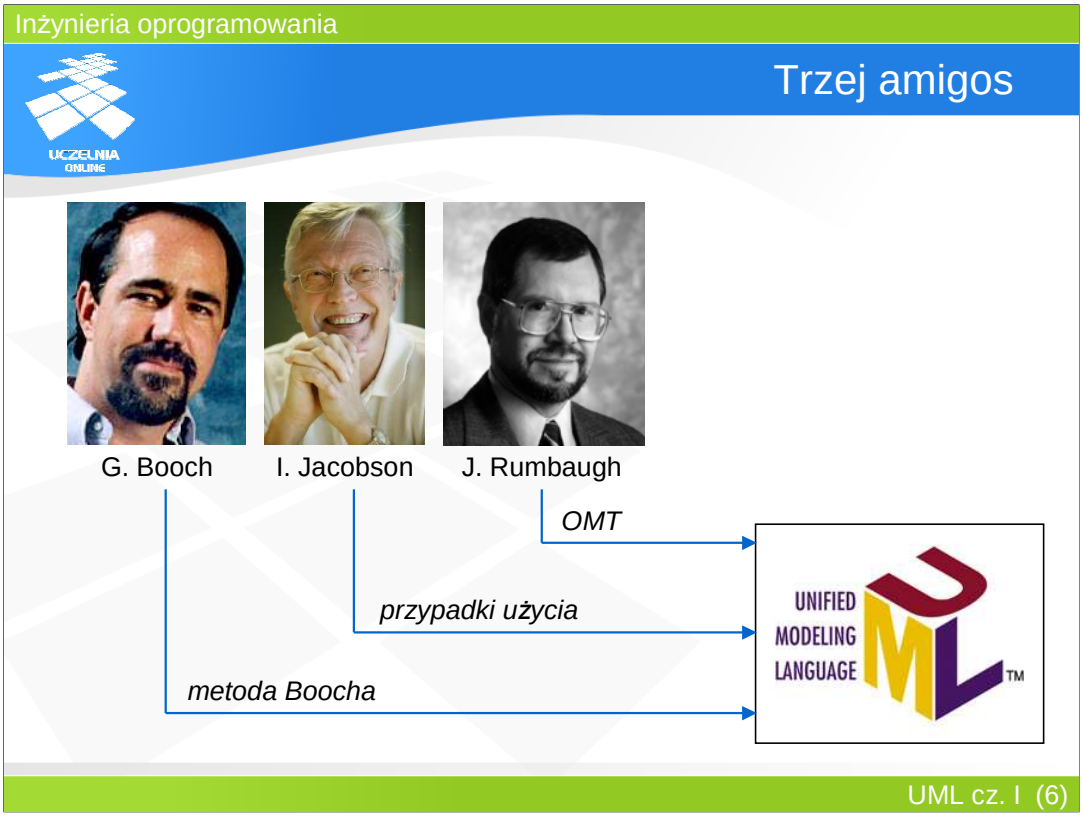
W latach 80-tych i na początku lat 90-tych istniało na rynku wiele notacji i metodyk modelowania, stosujących elementy o zbliżonej semantyce (np. klasy), ale całkowicie różniące się sposobem ich reprezentacji. Część z nich, z uwagi na wcześniejsze doświadczenia ich autorów, zdobyły większą popularność (np. OOAD, OOSE, OMT, Fusion, metoda Shlaera-Mellora czy Coada-Yourdona), jednak nadal brak było jednego standardu, który zaspokajałby wszystkie potrzeby. W większości były to notacje niekompletne, obejmujące część problematyki modelowania, i nie definiujące szczegółowo wielu pojęć.

Dlatego, na początku lat 90-tych G. Booch (twórca metody OOAD, kładącej nacisk na kwestie projektowania i implementacji) i J. Rumaugh (autor metody OMT, skupiającej się na modelowaniu dziedziny przedmiotowej), pracujący dla Rational Software (dzisiaj Rational jest własnością IBMa) dostrzegli możliwość wzajemnego uzupełnienia swoich metod i rozpoczęli prace nad Metodą Zunifikowaną, która miała objąć elementy dotychczas oddzielnych metodyk. Na konferencji OOPSLA w 1995 roku zaprezentowali oni wersję 0.8 Metody Zunifikowanej, a krótko potem dołączył do nich inny metodolog - I. Jacobson (twórca metody OOSE, posiadającej elementy związane z modelowanie funkcjonalności, użytkowników i cyklu życia produktu). W ten sposób powstała "masa krytyczna", która dawała szansę na opanowanie rynku przez nowopowstałą metodę. W roku 1996 do prac włączyła się niezależna organizacja OMG, której udział dawał szansę na wpływ na UML także innym firmom, nie tylko Rational Software.

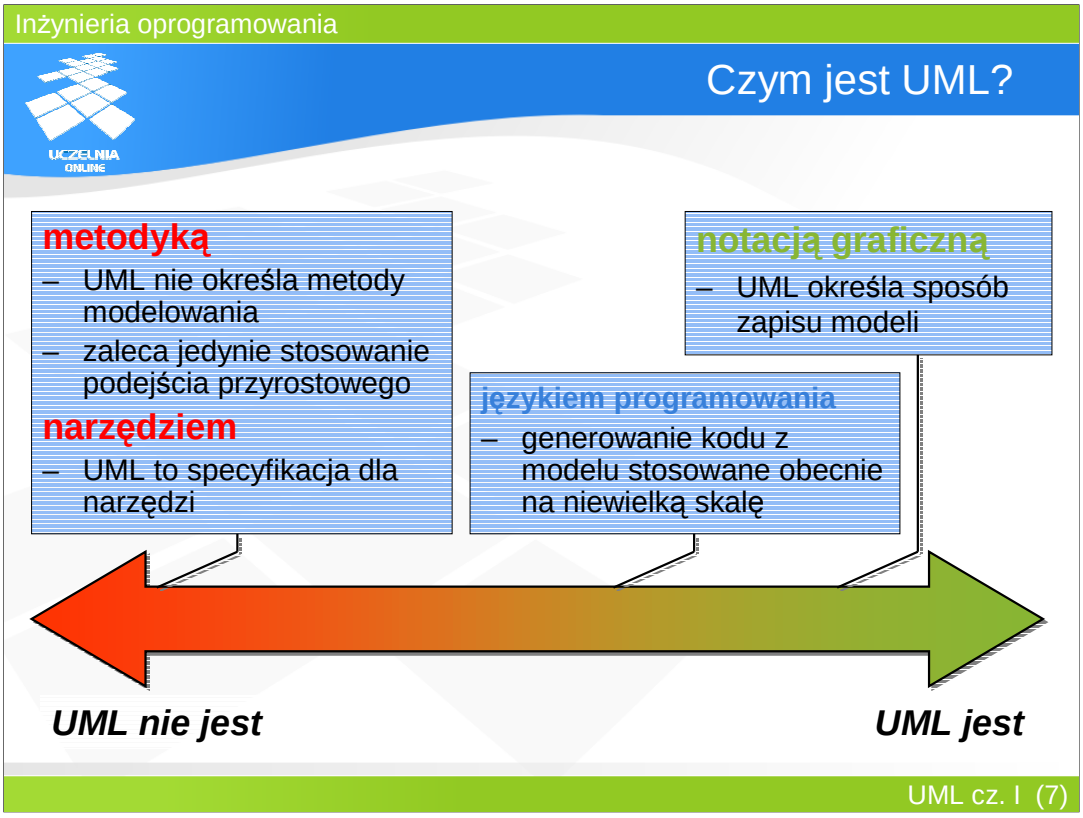


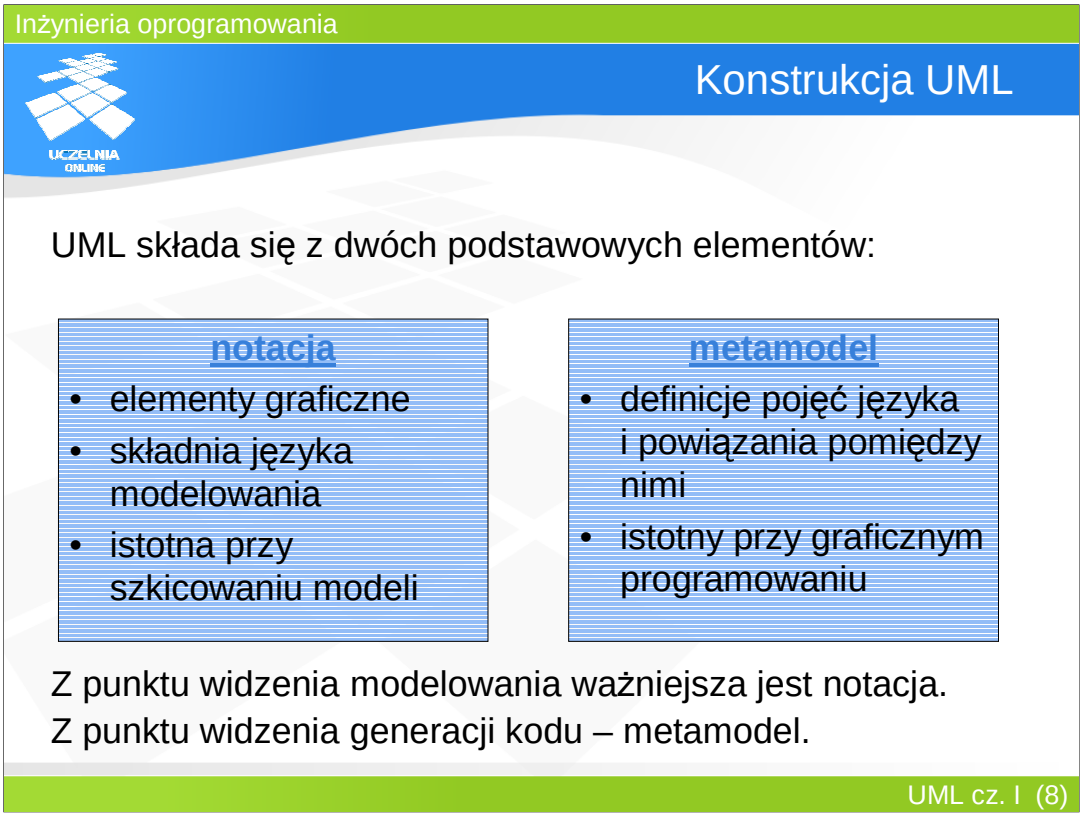
Efektem prac była najpierw wersja 1.0 UML opublikowana przez Rational, a kilka miesięcy później – wersja 1.1, wydana już pod egidą OMT. Kolejne wersje pojawiały się w odstępach kilkuletnich, pozwalając na stosowanie nowych diagramów, uspójniając notację i umożliwiając na modelowanie nowych dziedzin. Najnowsza wersja UML to 2.0.

Wraz z rozwojem UML zdobył dominującą pozycję na rynku – poza nim pozostają jedynie notacje związane z narzędziami 4GL.

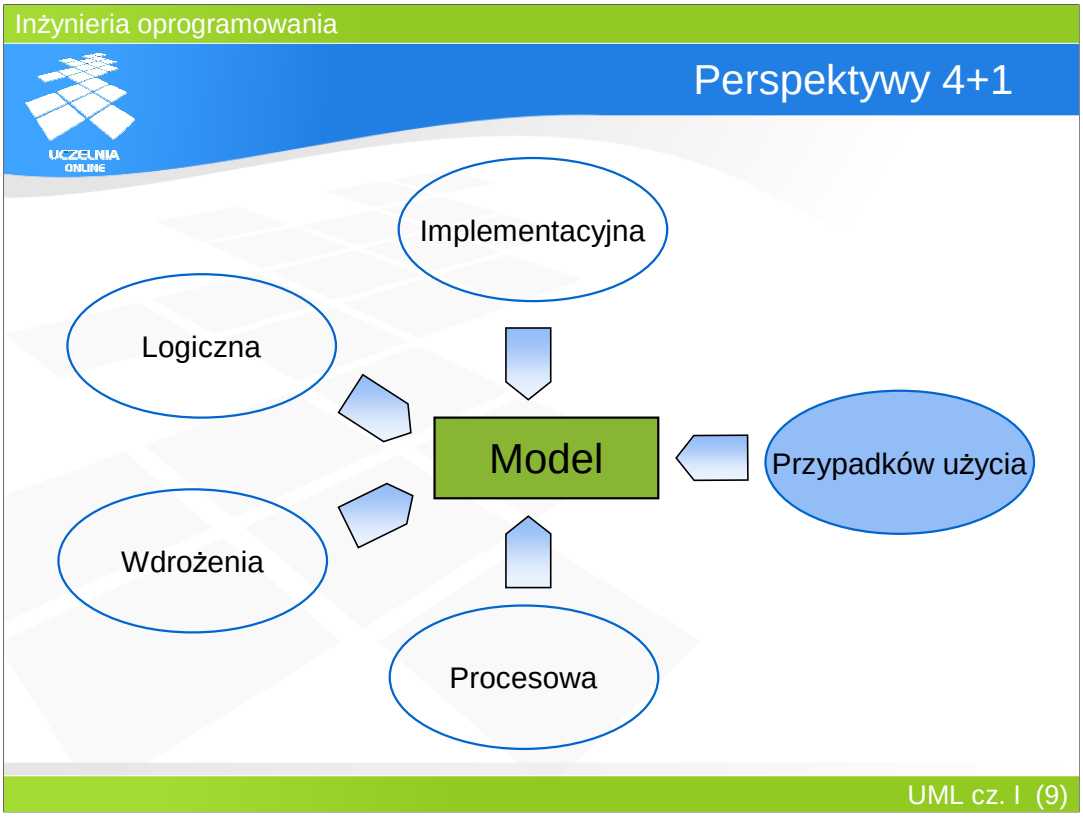


Trzej autorzy UMLa: Booch, Jacobson i Rumbaugh zunifikowali swoje notacje, tworząc jedną metodę. Z poszczególnych notacji przejęto najlepsze rozwiązania.


Inżynieria oprogramowania
Czym jest UML?
UCZELNIA ONLINE
metodyką
– UML nie określa metody modelowania
– zaleca jedynie stosowanie podejścia przyrostowego
narzędziem
– UML to specyfikacja dla narzędzi
notacją graficzną
– UML określa sposób zapisu modeli
językiem programowania
– generowanie kodu z modelu stosowane obecnie na niewielką skalę
UML nie jest
UML jest
UML cz. I (7)



UML definiuje dwie podstawowe składowe: notację poszczególnych elementów używanych na diagramach, a z drugiej strony – ich semantykę, czyli tzw. metamodel. Z punktu widzenia analityka istotniejsze jest czytelne i jednoznaczne opisanie modelu tak, aby inne osoby mogły zrozumieć jego znaczenie. Zatem ważniejsza dla niego jest notacja, zaś metamodel powinien być zrozumiały intuicyjnie. Z kolei przy generowaniu kodu i przejściu do implementacji ważniejsze jest ścisłe rozumienie znaczenia poszczególnych elementów, tak, aby możliwa była automatyczna konwersja modelu do innego formalizmu.



Modelowanie złożonych systemów jest zadaniem trudnym i angażuje wiele osób o różnym sposobie postrzegania systemu. Aby uwzględnić te punktu widzenia, UML jest często określany jako język modelowania z 4+1 perspektywą. Cztery pierwsze opisują wewnętrzną strukturę programu na różnych poziomach abstrakcji i szczegółowości. Ostatnia perspektywa opisuje funkcjonalność systemu widzianą przez jego użytkowników. Każda perspektywa korzysta z własnego zestawu diagramów pozwalających czytelnie przedstawić modelowane zagadnienie. Są to:

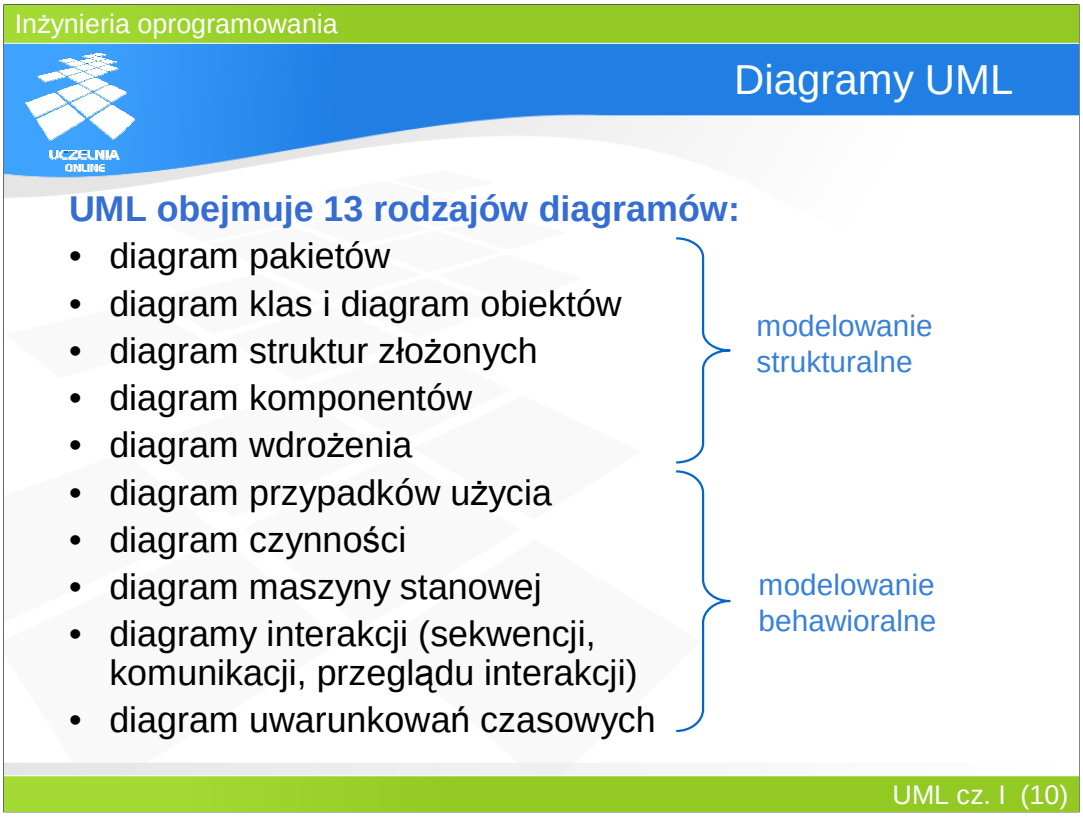
•Perspektywa przypadków użycia – opisuje funkcjonalność, jaką powinien dostarczać system, widzianą przez jego użytkowników.

•Perspektywa logiczna – zawiera sposób realizacji funkcjonalności, strukturę systemu widzianą przez projektanta

•Perspektywa implementacyjna – opisuje poszczególne moduły i ich interfejsy wraz z zależnościami; perspektywa ta jest przeznaczona dla programisty

•Perspektywa procesowa – zawiera podział systemu na procesy (czynności) i procesory (jednostki wykonawcze); opisuje właściwości pozafunkcjonalne systemu i służy przede także programistom i integratorom

•Perspektywa wdrożenia – definiuje fizyczny podział elementów systemu i ich rozmieszczenie w infrastrukturze; perspektywa taka służy integratorom i instalatorom systemu



W UML zdefiniowano 13 rodzajów diagramów podzielonych na dwie główne grupy: opisujących strukturę systemu i opisujących zachowanie. Nie wszystkie są i muszą być używane jednocześnie – zależy to od rodzaju i złożoności modelowanego systemu. Część z nich służy do modelowania tego samego aspektu, jednak ujętego nieco inaczej, dlatego dobór rodzajów diagramów zależy także od preferencji analityka.

Diagram przypadków użycia

Diagram przypadków użycia

- definiuje granice modelowanego systemu
- określa jego kontekst
- wymienia użytkowników systemu i jednostki zewnętrzne
- przedstawia funkcje dostępne dla użytkowników
- określa powiązania i zależności pomiędzy nimi

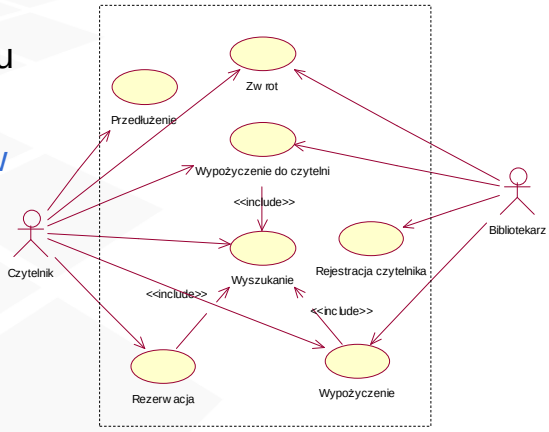
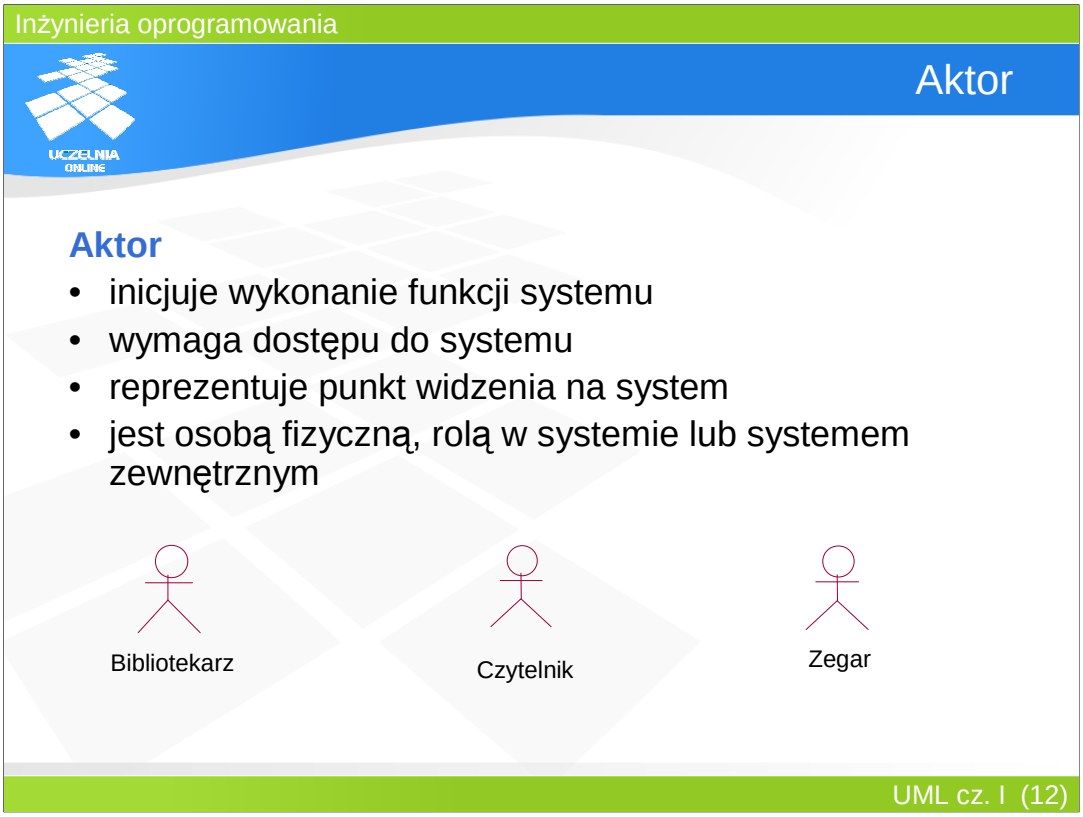


Diagram przypadków użycia (ang. *use-case diagram*) służy do modelowania aktorów (użytkowników systemu, odbiorców efektów pracy systemu, systemów zewnętrznych) i ich potrzeb w stosunku do tworzonego systemu. Przypadki użycia prezentowane na tym diagramie to sekwencje czynności, które prowadzą do spełnienia celu przypadku użycia (zaspokojenia pewnej potrzeby użytkownika).



Aktor jest osobą (lub dowolną inną jednostką), która w jakiś sposób wymienia informacje z systemem, choć pozostaje poza jego zakresem. Jest więc w szerokim znaczeniu użytkownikiem tego systemu, tzn. żąda od systemu wykonania pewnych funkcji i/lub odbiera efekty ich wykonania. Aktor opisuje rolę, a nie konkretną osobę lub jednostkę.

Aktorzy mogą być powiązani ze sobą relacją uogólnienia/uszczegółowienia: w ten sposób zachowanie bardziej ogólnego aktora jest dziedziczone przez aktora bardziej szczegółowego. Ponadto są powiązani z przypadkami użycia, z których korzystają.

Aktor jest reprezentowany na diagramie przypadków użycia w różnoraki sposób: jako sylwetka z nazwą aktora, jako prostokąt ze słowem kluczowym «actor» lub z ikoną.

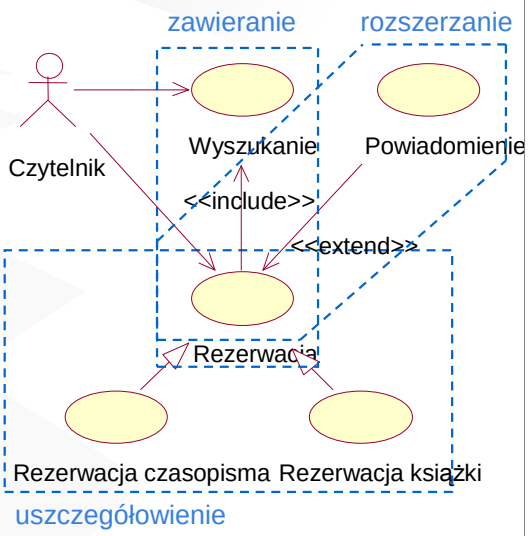
Przykładami aktorów w systemie bibliotecznym mogą być Bibliotekarz i Czytelnik, reprezentujący fizyczne osoby korzystające z tego systemu, ale także Zegar, który cyklicznie wysyła komunikat powodujący wyszukanie książek o przekroczonym terminie zwrotu.

Przypadek użycia

Przypadek użycia reprezentuje kompletną funkcję dostępną dla aktora.

Przypadki użycia mogą być powiązane zależnościami:

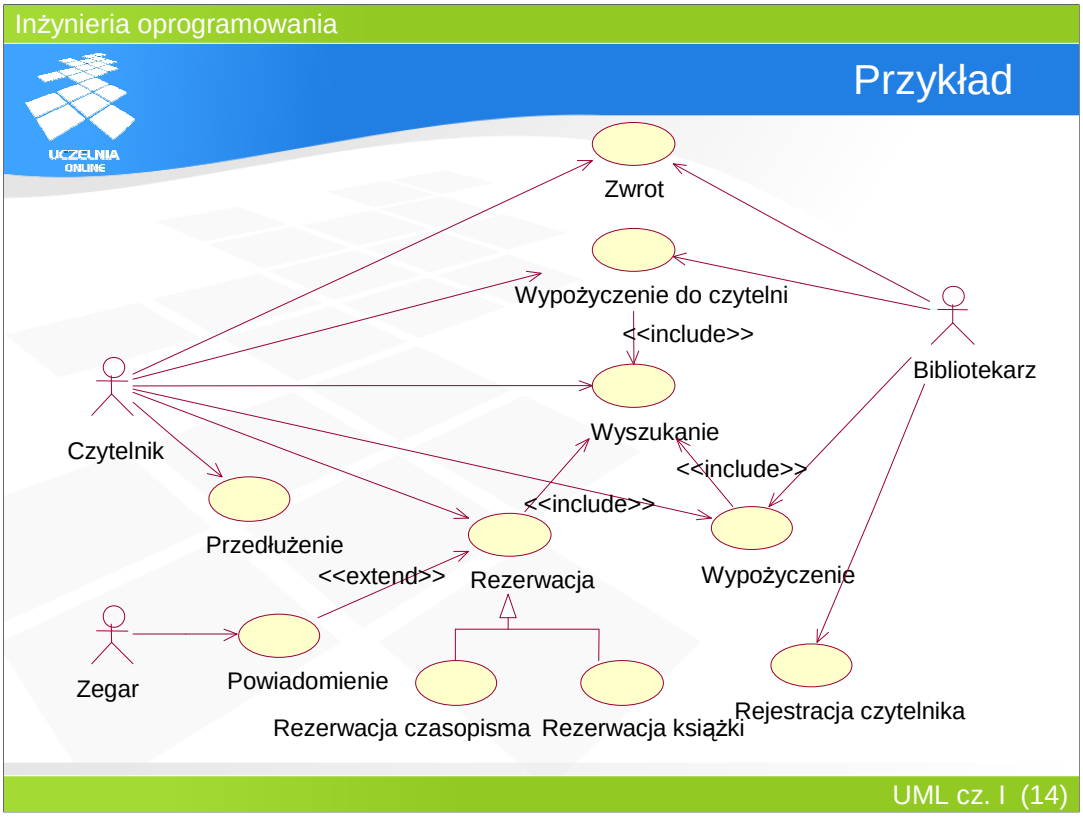
- Uszczegółowienie: specjalizowana wersja przypadku użycia
- Rozszerzanie: dodatkowa funkcjonalność przypadku użycia
- Zawieranie: wykonanie jednego przypadku użycia przez drugi



Przypadek użycia reprezentuje zamkniętą i kompletną funkcjonalność dostępną dla aktora. Zgodnie z definicją, przypadek użycia w UML jest zdefiniowany jako zbiór akcji wykonywanych przez system, które powodują efekt zauważalny dla aktora.

Przypadki użycia zawsze muszą być inicjowane (bezpośrednio lub przez zależności) przez aktora i wykonywane w jego imieniu. Ponadto, przypadek użycia musi dostarczać pewną wartość użytkownikowi oraz musi być kompletny, to znaczy w pełni realizować podaną funkcjonalność oraz dostarczać wyniki aktorowi.

Przypadki użycia komunikują się z aktorami poprzez powiązania, pokazujące, który aktor ma dostęp do podanego przypadku użycia. Ponadto mogą być powiązane pomiędzy sobą: *relacją uogólnienia*, *rozszerzenia* i *zawierania*.



Przykład przedstawia diagram przypadków użycia. Występuje w nim trzech aktorów: Czytelnik, Bibliotekarz i Zegar. Pierwsi dwaj reprezentują role użytkowników systemu, natomiast Zegar służy do generowania cyklicznych Powiadomień.

Czytelnik i Bibliotekarz korzystają z przypadków użycia. Niektóre z nich, np. Zwrot lub Wypożyczenie do czytelni, są przez nich współdzielone, natomiast Rejestracja czytelnika i Przedłużenie są dostępne tylko dla jednego albo drugiego aktora.

Przypadek użycia Wyszukanie jest włączany do kilku innych przypadków użycia: Rezerwację, Wypożyczenie i Wypożyczenie do czytelni. W ten sposób jest on wywoływany w sposób pośredni przez aktora, a bezpośrednio przez inny przypadek użycia.

Przypadek użycia Rezerwacja jest rozszerzany przez Powiadomienie. Oznacza to, że Powiadomienie może uczestniczyć w realizacji funkcji Rezerwacji. Ponadto Rezerwacja posiada dwa szczegółowe przypadki: Rezerwację książki i Rezerwację czasopisma.

Diagram klas

Diagram klas przedstawia klasy występujące w systemie i statyczne relacje pomiędzy nimi wraz z ograniczeniami. Jest podstawowym diagramem struktury logicznej systemu.

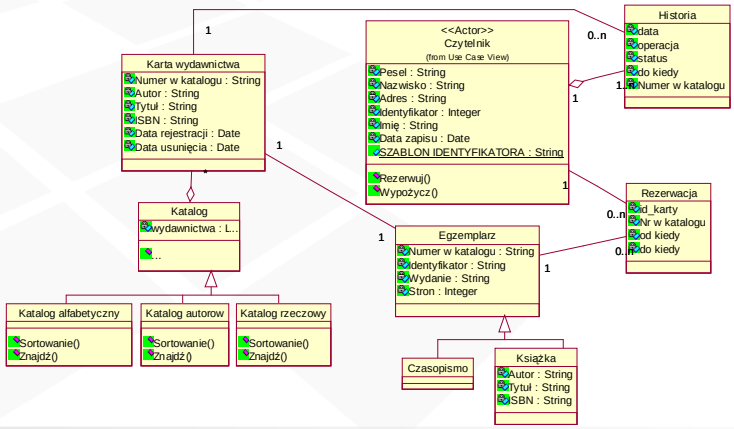
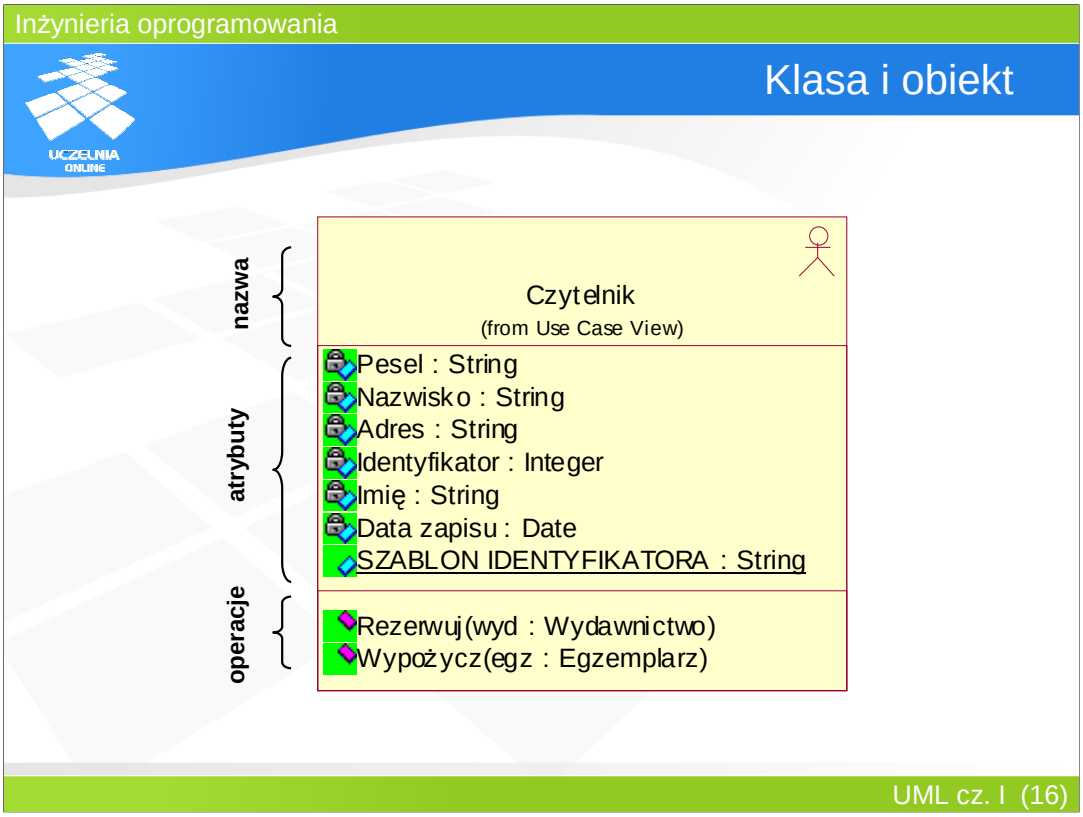


Diagram klas (ang. *class diagram*) jest najczęściej używanym diagramem UML. Z reguły zawiera także największą ilość informacji i stosuje największą liczbę symboli.

Na diagramie są prezentowane klasy, ich atrybuty i operacje, oraz powiązania między klasami. Diagram klas przedstawia więc podział odpowiedzialności pomiędzy klasy systemu i rodzaj wymienianych pomiędzy nimi komunikatów. Z uwagi na rodzaj i ilość zawartych na tym diagramie danych jest on najczęściej stosowany do generowania kodu na podstawie modelu.

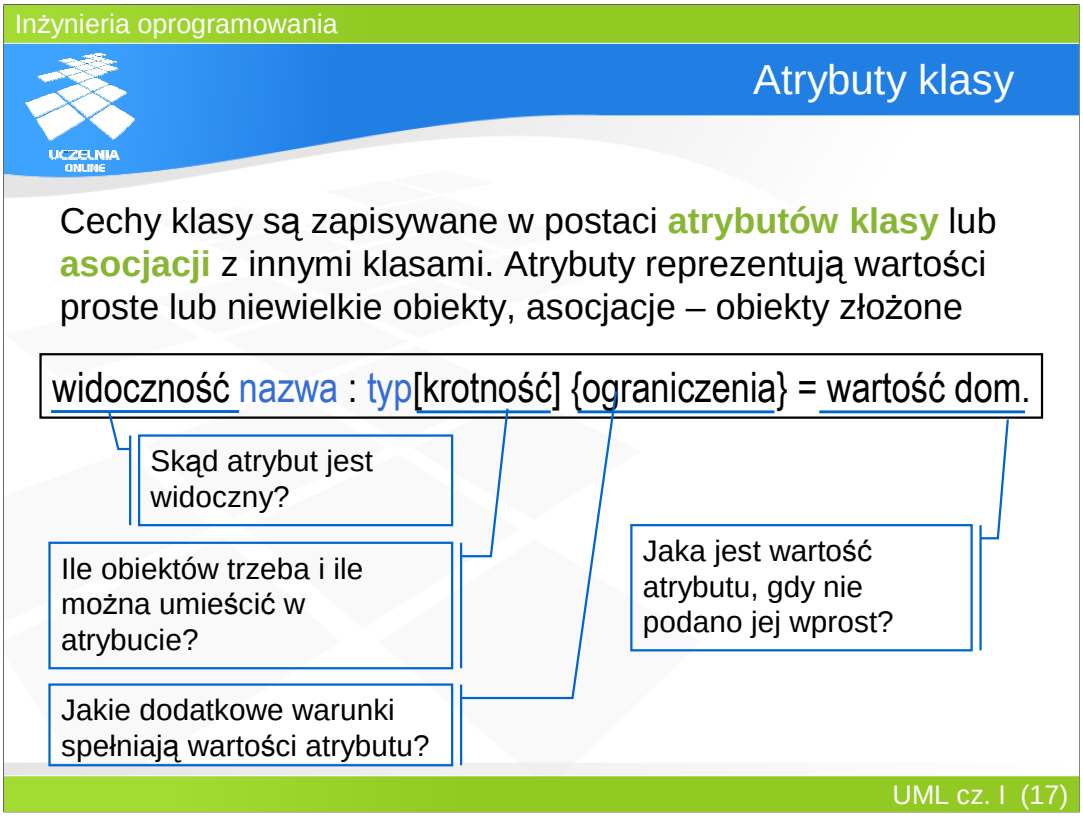


Klasa jest reprezentowana przez prostokąt z wydzielonymi przedziałami: nazwą, atrybutami i operacjami. W celu zwiększenia czytelności, dowolny z nich można ukryć bądź dodać nowy (np. przechowujący zdarzenia lub wyjątki), choć zwykle są to właśnie trzy przedziały. Tradycyjnie nazwa klasy zaczyna się z dużej litery, jest wytłuszczona, a w przypadku klasy abstrakcyjnej – także pochyła.

Obiekt jest instancją klasy, podobnie jak w przypadku programowania obiektowego. Nazwa obiektu jest umieszczana przed nazwą klasy i oddzielana od niej dwukropkiem.

Cechy klasy reprezentują informację, jaką klasa przechowuje. Mogą zostać zapisane w postaci dwóch, w zasadzie równoważnych notacji: jako atrybuty klasy (umieszczane w przedziale atrybutów) lub jako relacje pomiędzy klasami (zapisywane w postaci linii łączącej klasy). Zwykle pierwsza notacja jest stosowana do typów prostych lub obiektów reprezentujących wartości, natomiast druga do typów złożonych.

Operacje reprezentują usługi, jakie klasa oferuje. Ich realizacje – metody – dostarczają implementacji tych usług.



Zwykle atrybut jest opisywany tylko przez dwa elementy: nazwę i typ. Jednak pełna definicja obejmuje także **widoczność atrybutu**, definiującą, z jakich miejsc systemu atrybut jest dostępny, **krotność**, która określa ile obiektów mieści się w atrybucie, dodatkowe **ograniczenia** nałożone na atrybut, i **wartość domyślną**. Elementom, których w definicji atrybutu nie podano wartości, przypisywane są wartości domyślne (widoczność prywatna, krotność 1) lub pomija się je.

Poziomy widoczności

UML definiuje 4 poziomy widoczności cech i metod

- + publiczny – element jest widoczny z każdego miejsca w systemie
- # chroniony – element jest widoczny we własnej klasie i jej podklasach
- – prywatny – element jest widoczny tylko we własnej klasie
- ~ publiczny wewnątrz pakietu – element jest widoczny tylko wewnątrz własnego pakietu

Podobnie jak w wielu wysokopoziomowych językach programowania, UML posiada 4 poziomy widoczności: publiczny, chroniony, prywatny i publiczny wewnątrz pakietu. Poziomy te zwykle służą do opisywania widoczności cech (atrybutów i asocjacji) oraz operacji, jednak dotyczą także np. klas pakietów etc.

Krotność

Krotność pozwala określić minimalną i maksymalną liczbę obiektów, jakie można powiązać z daną cechą:

- dolna granica..górna granica – przedział od-do
- 1 – dokładnie jeden obiekt
- 0..1 – opcjonalnie jeden obiekt
- 1..* - przynajmniej jeden obiekt
- 1, 3, 5 – konkretne liczby obiektów
- * - dowolna liczba obiektów

Krotność cechy wskazuje, ile obiektów można, a ile trzeba w niej zawrzeć. Krotność można określać jako ograniczenie dolne i górne, jednak oczywiste lub powtarzające się wartości graniczne można pomijać, np.. zapis 0..* jest skracany do *, a zapis 1..1 do 1.

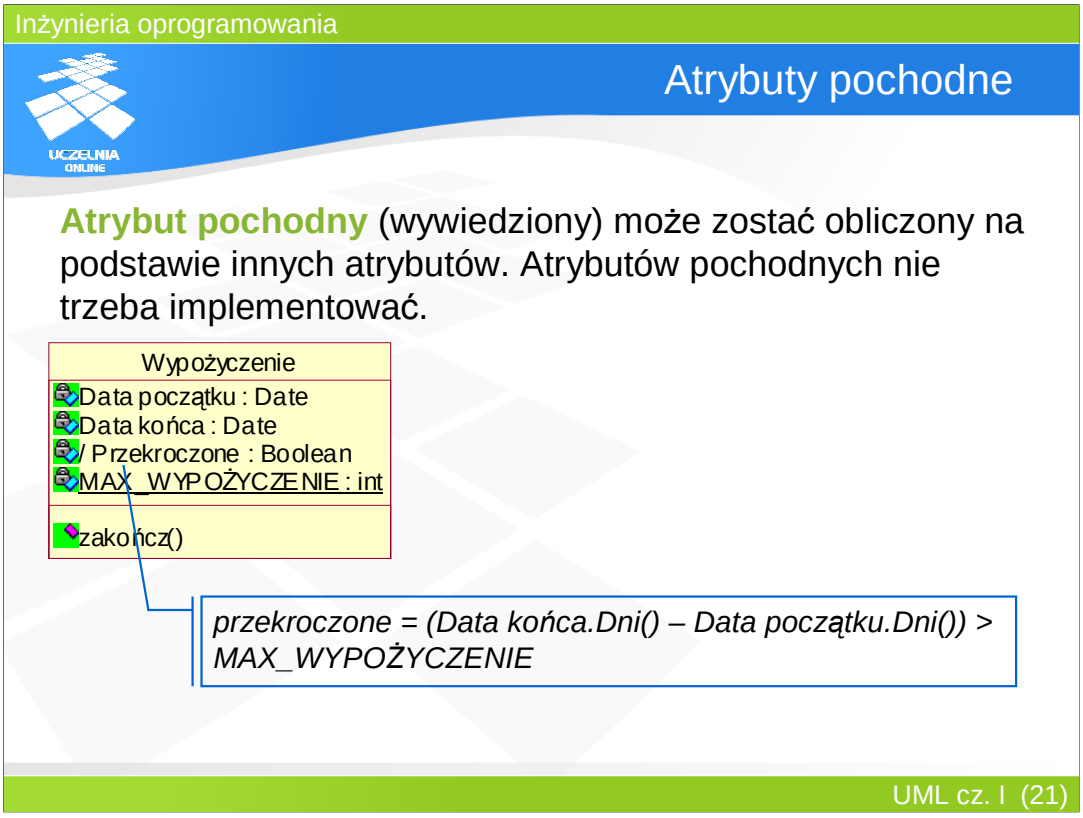
W praktyce programowania istotna jest krotność 0, 1 i dowolna, natomiast wartości dyskretne są mniej ważne, jako szczególne przypadki wymienionych trzech. W UMLu 2.0 dlatego formalnie usunięto możliwość podawania dowolnych liczb będących ograniczeniami, np. 2..4, jednak z uwagi na czytelność tego zapisu użycie go nie stanowi wielkiego błędu.

Właściwości i ograniczenia atrybutów

Z atrybutem mogą być związane dodatkowe ograniczenia, które określają jego właściwości, np.:

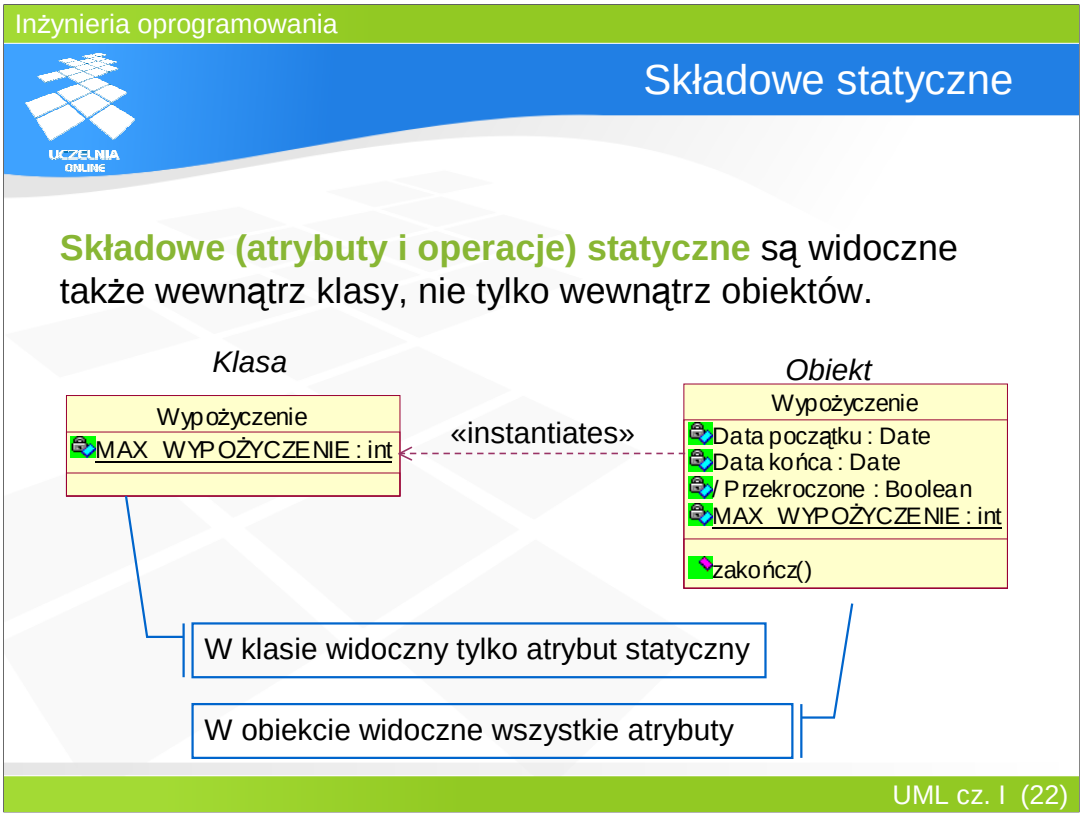
- {ordered} – obiekty wewnątrz cechy są uporządkowane
- {unordered} – obiekty są nieuporządkowane
- {unique} – obiekty wewnątrz cechy nie powtarzają się
- {nonunique} – obiekty wewnątrz cechy mogą się powtarzać
- {readOnly} – wartość atrybutu służy tylko do odczytu
- {frozen} – wartość atrybutu nie może być zmodyfikowana po jej przypisaniu

Niemal każdy element w UML może posiadać dodatkowe właściwości i ograniczenia, które szczegółowo opisują jego zachowanie i przeznaczenie. Są one zapisywane w nawiasach klamrowych. Atrybuty klasy można oznaczyć jako uporządkowane za pomocą ograniczenia {sorted}, co oznacza, że są one w jakiś sposób (zwykle rosnąco, leksykograficznie) posortowane. Ograniczenie {unique} wymaga, aby obiekty pamiętane wewnątrz atrybutu nie powtarzały się. Właściwość {readOnly} oznacza atrybut, którego wartość jest przeznaczona wyłącznie do odczytu, natomiast {frozen} – którego wartość po zdefiniowaniu nie może być zmieniona.



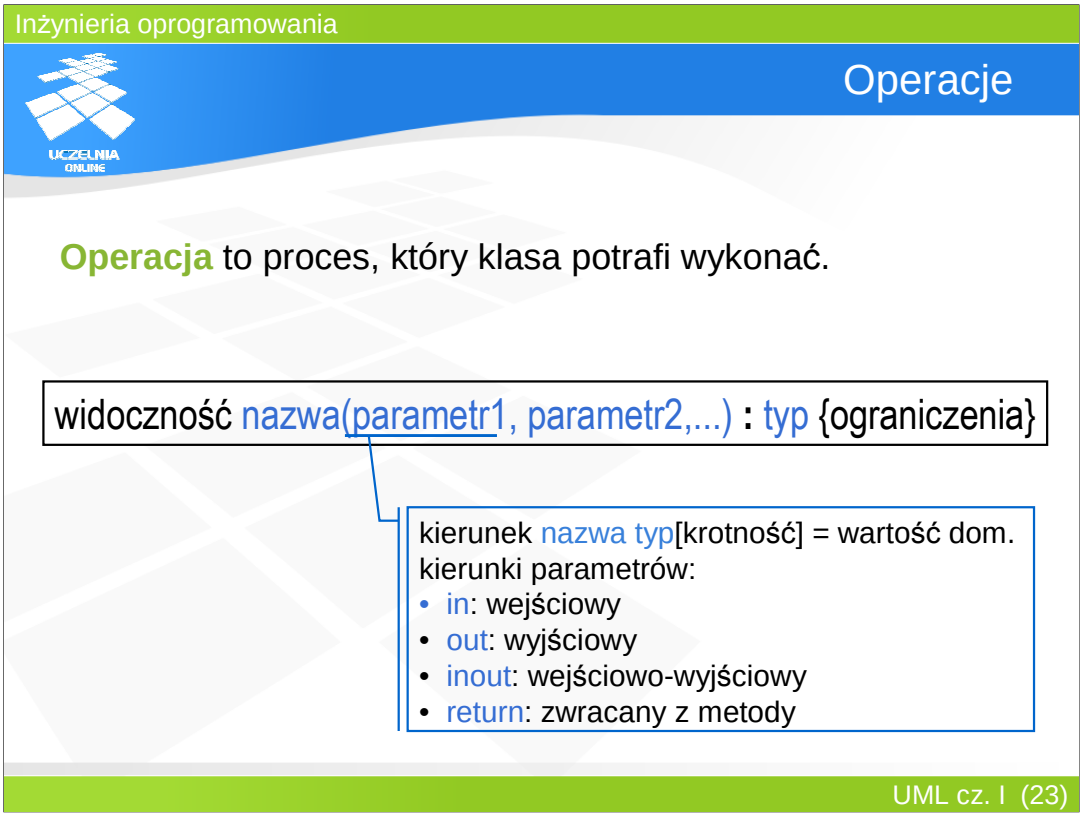
Atrybuty pochodne (ang. *derived*) są zależne od innych atrybutów i ich wartości można obliczyć na podstawie tych atrybutów. Często w fazie implementacji są przekształcane w metody lub ich wartość jest obliczana na bieżąco. Nie ma zatem potrzeby ich zapamiętywania w klasie.

Atrybuty pochodne są oznaczane znakiem '/' umieszczonym przed nazwą atrybutu.

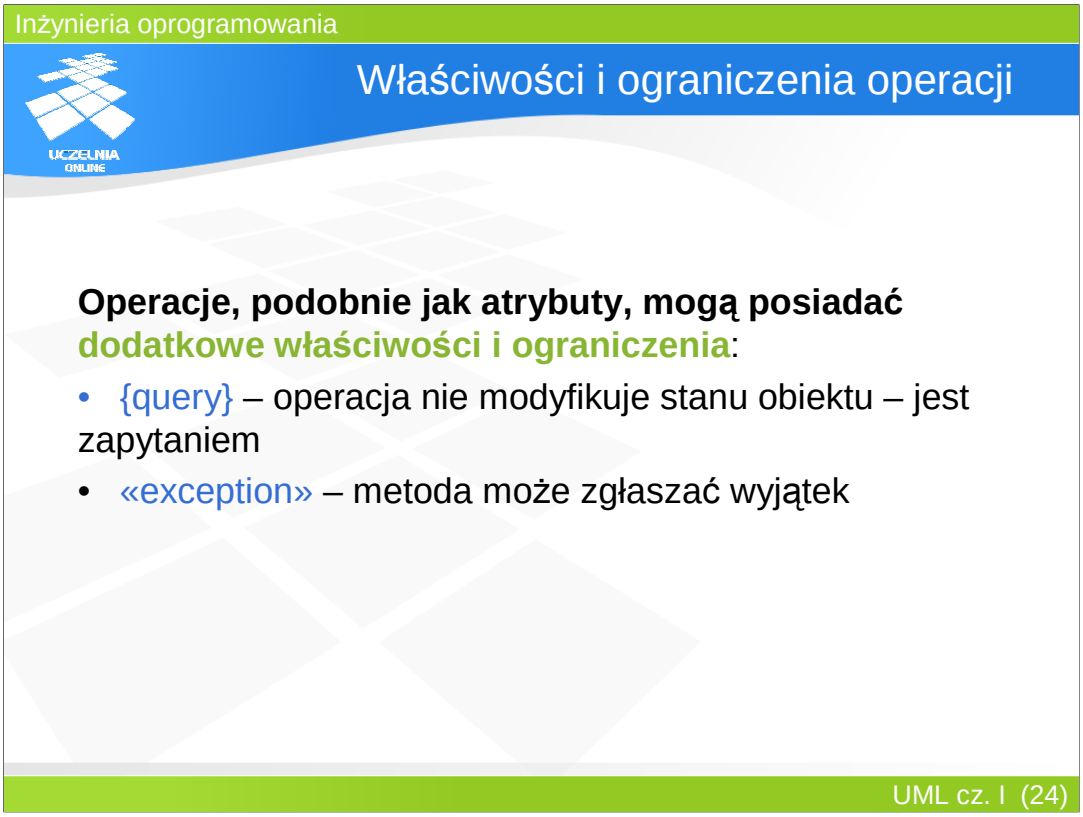


Składowe statyczne w klasie są widoczne zarówno w klasie, jak i w jej instancji. Składowe niestatyczne są widoczne jedynie w obiektach danej klasy, zatem wymagają utworzenia jej instancji.

Składowe statyczne klasy są oznaczane podkreśleniem jej sygnatury.



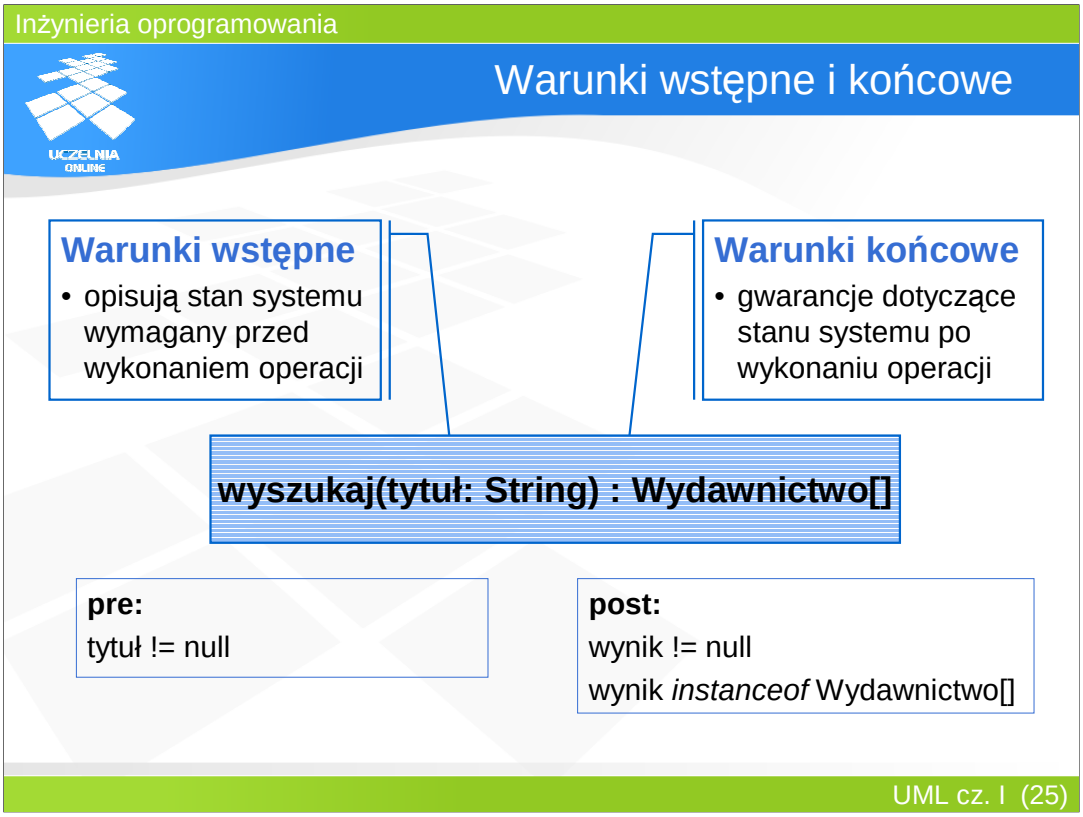
Operacje są opisywane w UMLu podobnie jak atrybuty, oczywiście, z uwzględnieniem listy parametrów i pominięciem wartości domyślnej. Parametry są zapisywane identycznie jak atrybuty klasy, jednak są poprzedzone informacją o kierunku jego przekazania: in, out, inout i return. Domyślnym kierunkiem jest wejściowy.



Definicja operacji wewnątrz klasy przewiduje, podobnie jak w przypadku atrybutów, możliwość umieszczenia dodatkowych informacji i ograniczeń.

Spośród nich największe znaczenie ma słowo kluczowe {query} oznaczające, że metoda jedynie zwraca fragment stanu obiektu, natomiast go nie modyfikuje (czyli nie ma efektu ubocznego). Informacja taka ma bardzo duże znaczenie w fazie implementacji.

Podobnie metoda może zgłaszać wyjątki. Wprawdzie UML nie definiuje sposobu, w jaki powinna być oznaczona taka metoda, jednak powszechnie stosowane jest słowo kluczowe *exception* wraz z nazwą wyjątku jako opis klasy wyjątku, oraz informacja o możliwości zgłoszenia wyjątku skojarzona z metodą.



Operację można także opisywać przez dwa rodzaje warunków: wstępne (ang. *preconditions*) i końcowe (ang. *postconditions*). Opisują one wymagany i oczekiwany stan fragmentu systemu wymagany odpowiednio przed i po wykonaniu operacji. Pozwala to na precyzyjniejsze opisane zadania realizowanego przez metodę, jej wymagań i efektów jej wykonania. Projektant ma możliwość wyrażenia poprzez nie, jakie warunki muszą być spełnione w celu poprawnego wykonania zadania przez operację.

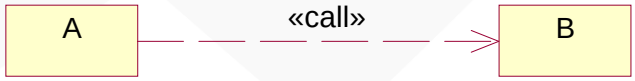
W tym przykładzie warunkiem wstępnym poprawnego wykonania operacji *wyszukaj()* jest przekazanie niepustego parametru reprezentującego tytuł wydawnictwa, a warunkiem końcowym – zwrócenie wartości różnej od null będącej tablicą typu Wydawnictwo. Operacja *wyszukaj()* nie gwarantuje określonego rozmiaru zwracanej tablicy.

Zależność

Zależności są najprostszym i najsłabszym rodzajem relacji łączących klasy. Oznaczają, że zmiana jednej z nich w pewien sposób wpływa na drugą, np.

- «call» - operacje w klasie A wywołują operacje w klasie B
- «create» - klasa A tworzy instancje klasy B

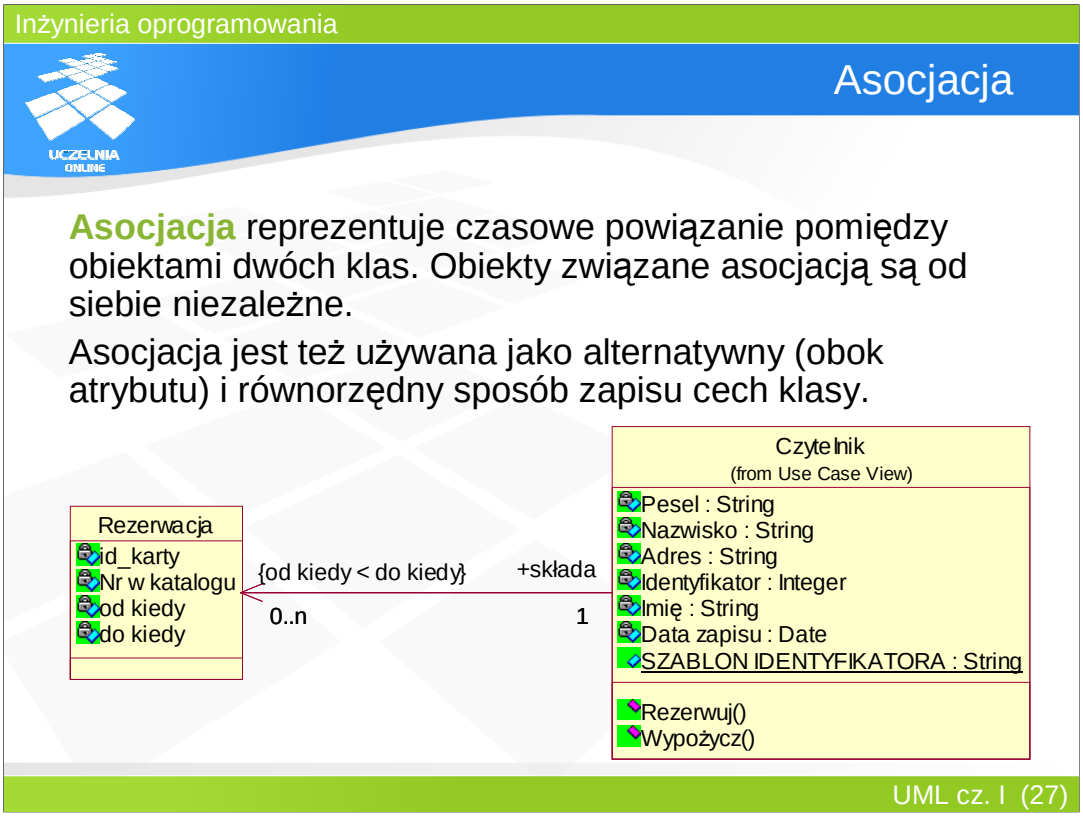
  «instantiate» - obiekt A jest instancją klasy B
- «use» - do zaimplementowania klasy A wymagana jest klasa B



Zależność między klasami jest najsłabszym typem relacji, jaka może między nimi zaistnieć. Ma ona miejsce wówczas, gdy zmiana definicji jednej klasy może spowodować zmianę drugiej. Oznacza to zwykle, że zależność jest jednokierunkowa.

Zależności często opisuje się frazą "...korzysta z...", "...oddziałuje na...", "...ma wpływ na...", "...tworzy...". Z uwagi na różny rodzaj zależności, stosuje się tzw. słowa kluczowe, które doprecyzowują znaczenie zależności. Duża liczba zależności w nawet niewielkim systemie powoduje, że nie warto przedstawiać wszystkich, koncentrując się jedynie na nieoczywistych, znaczących dla zrozumienia diagramu.

Zależności są oznaczane linią przerywaną, a o rodzaju zależności decyduje słowo kluczowe umieszczone nad linią.



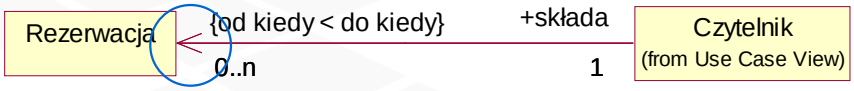
Asocjacje są silniejszymi relacjami niż zależności. Wskazują, że jeden obiekt jest związany z innym przez pewien okres czasu. Jednak czas życia obu obiektów nie jest od siebie zależny: usunięcie jednego nie powoduje usunięcia drugiego.

Relacje asocjacji są zwykle opisywane frazami "...posiada...", "...jest właścicielem...", jednak ich znaczenie często jest mylone z inną relacją – agregacją. W przypadku asocjacji żaden obiekt nie jest właścicielem drugiego: nie tworzy go, nie zarządza nim, a moment usunięcia drugiego obiektu nie jest z nim związany. Z drugiej strony, obiekt powiązany asocjacją z drugim posiada referencję do niego, może się do niego odwołać etc. Asocjacje mogą posiadać nazwy, zwykle w postaci czasownika, który pozwala przeczytać w języku naturalnym jej znaczenie, np. „A posiada B”. Często pomija się jedną z nazw asocjacji dwukierunkowej, jeżeli jest ona jedynie stroną bierną drugiej nazwy, np. „przechowuje” – „jest przechowywany”.

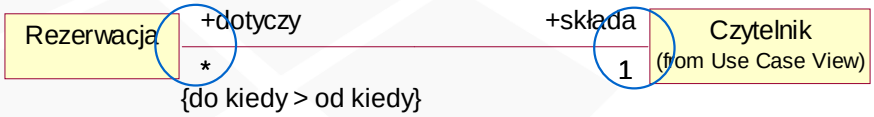
Asocjacja jest równoważna atrybutowi: UML nie rozróżnia obiektu, który jest polem klasy od obiektu i jest z nią związany asocjacją. Warto jednak przyjąć konwencję, w której obiekty reprezentujące wartości (np. daty) oraz typy proste (liczby, napisy, znaki) są modelowane jako atrybuty, natomiast obiekty dostępne poprzez referencje – są przedstawiane poprzez asocjacje.

Nawigowalność asocjacji

Nawigowalność określa wiedzę o sobie nawzajem obiektów uczestniczących w relacji.



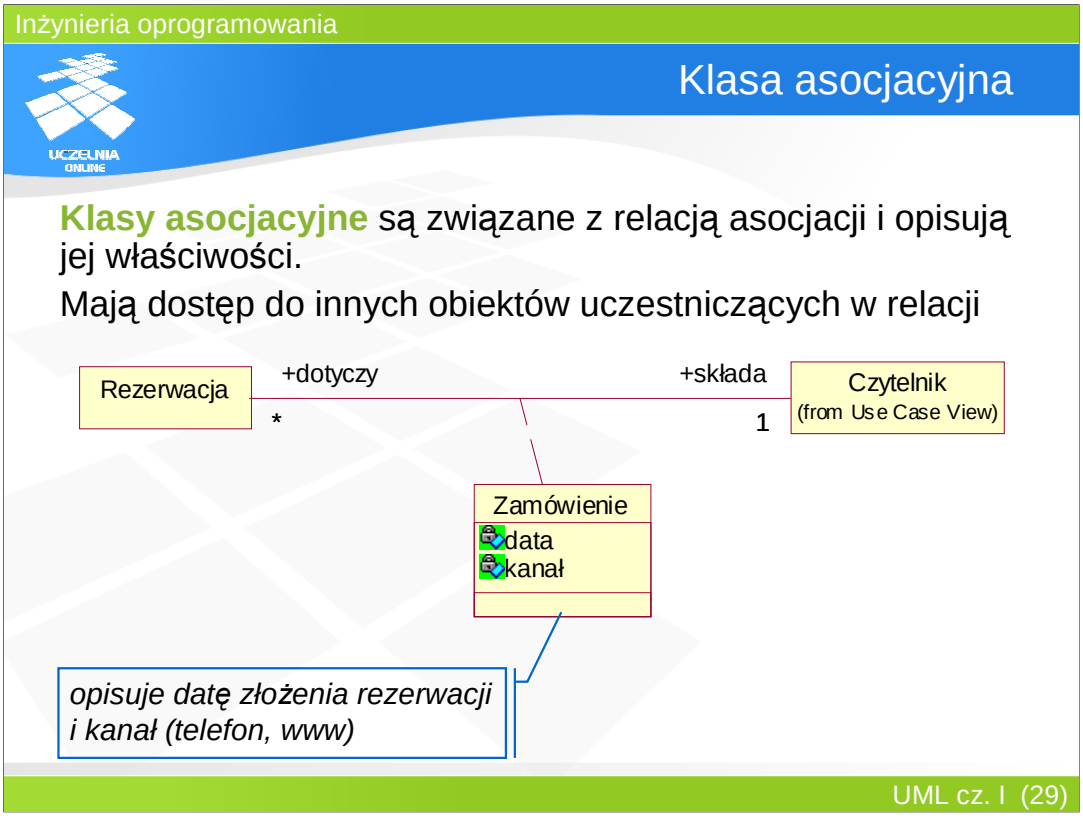
Czytelnik wie o swoich rezerwacjach.
Rezerwacja nie odwołuje się do czytelnika



Oba obiekty pozwalają na nawigację do siebie nawzajem

Asocjacje modelują względną równowagę pomiędzy połączonymi nimi obiektami, jednak nie oznacza to, że ich wiedza o sobie jest taka sama. Informację o kierunku relacji (czyli który obiekt może odwołać się do drugiego) opisuje kierunek asocjacji (czyli jej nawigowalność). Nawigowalność pomiędzy klasą A i klasą B oznacza, że od obiektu klasy A można przejść do obiektu klasy B, ale nie odwrotnie. Nawigowalność dwukierunkowa oznacza, że nawigując od obiektu klasy A do obiektu klasy B, a następnie z powrotem, w zbiorze wyników można znaleźć początkowy obiekt klasy A.

Nawigowalność oznaczana jest na diagramach strzałką. W przypadku nawigowalności dwukierunkowej strzałki pomija się.



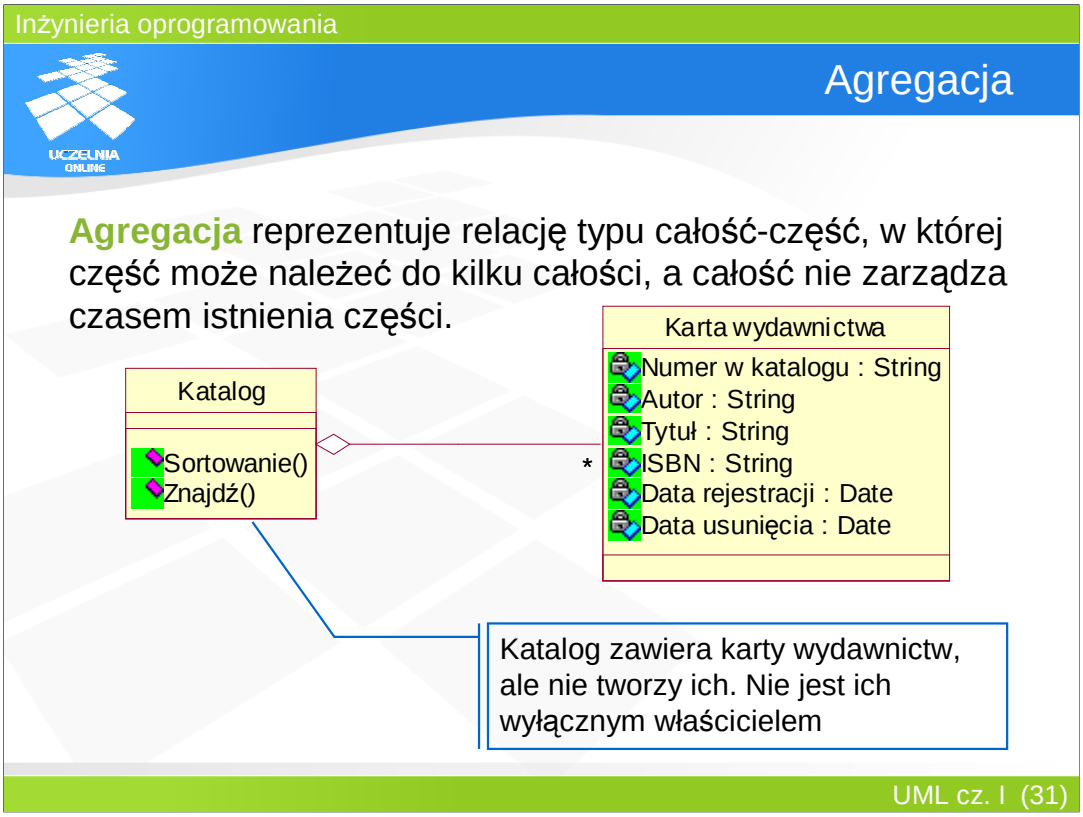
Klasa asocjacyjna umożliwia opisanie za pomocą atrybutów i operacji nie obiektu, ale właśnie samej asocjacji pomiędzy klasami. Informacje przechowywane w klasie asocjacyjnej nie są związane z żadną z klas uczestniczących w asocjacji, dlatego wygodnie jest stworzyć dodatkową klasę i powiązać ją z relacją.

Klasy asocjacyjne są reprezentowane graficznie jako klasy połączone linią przerywaną z relacją asocjacji, której dotyczą.

Asocjacja kwalifikowana

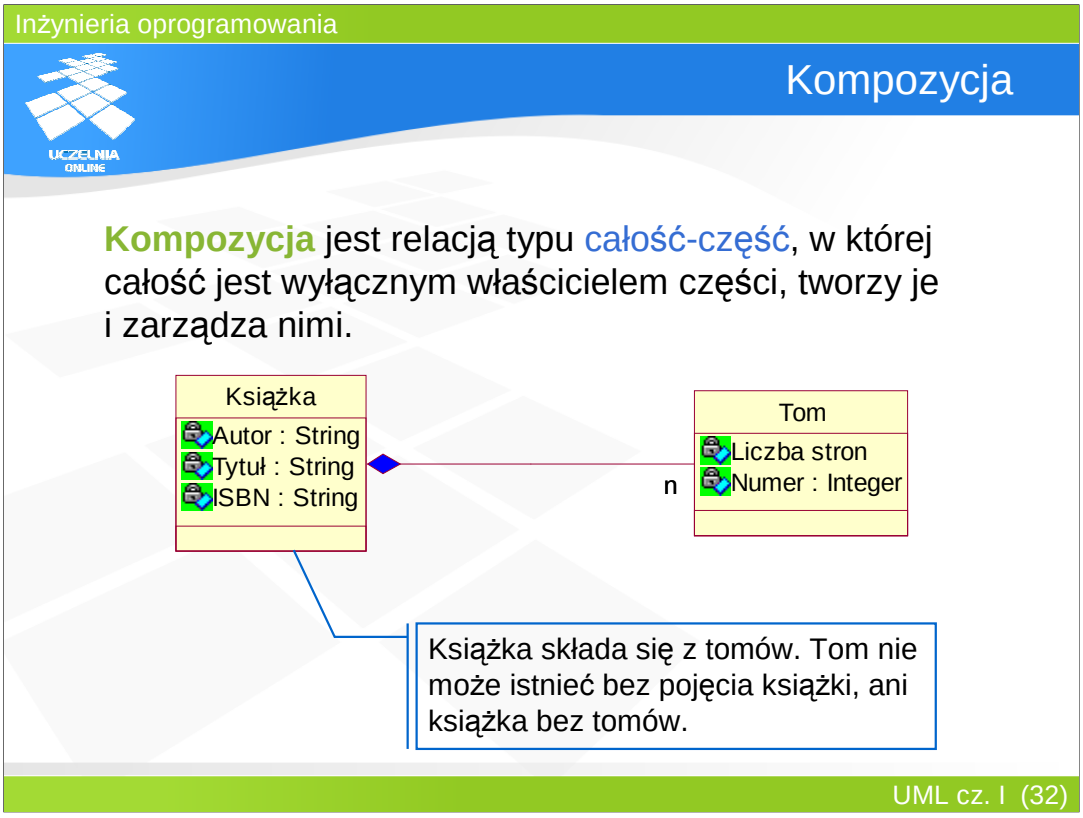
Asocjacja kwalifikowana pozwala wskazać, który atrybut jednej z klas służy do zapewnienia unikatowości związku (jest jego kwalifikatorem).

Asocjacja kwalifikowana jest rozszerzeniem zwykłej asocjacji o możliwość określenia, który z atrybutów jednej z klas decyduje o związku między nimi. Na przykład, składając Rezerwację, Czytelnik podaje listę Wydawnictw, które chciałby pożyczyć. Innymi słowy, między Rezerwacją a Czytelnikiem występuje relacja typu wiele-jeden. Jednak w danym momencie Czytelnik może zarezerwować dane Wydawnictwo tylko jeden raz – i dlatego atrybut id wydawnictwa jest kwalifikatorem tej relacji. W efekcie pomiędzy instancją Czytelnika a instancją Rezerwacji występuje relacja jeden-jeden, ponieważ konkretny Czytelnik rezerwuje konkretne Wydawnictwo w danym momencie tylko raz.



Agregacja jest silniejszą formą asocjacji. W przypadku tej relacji równowaga między powiązanymi klasami jest zaburzona: istnieje właściciel i obiekt podrzędny, które są ze sobą powiązane czasem swojego życia. Właściciel jednak nie jest wyłącznym właścicielem obiektu podrzędnego, zwykle też nie tworzy i nie usuwa go.

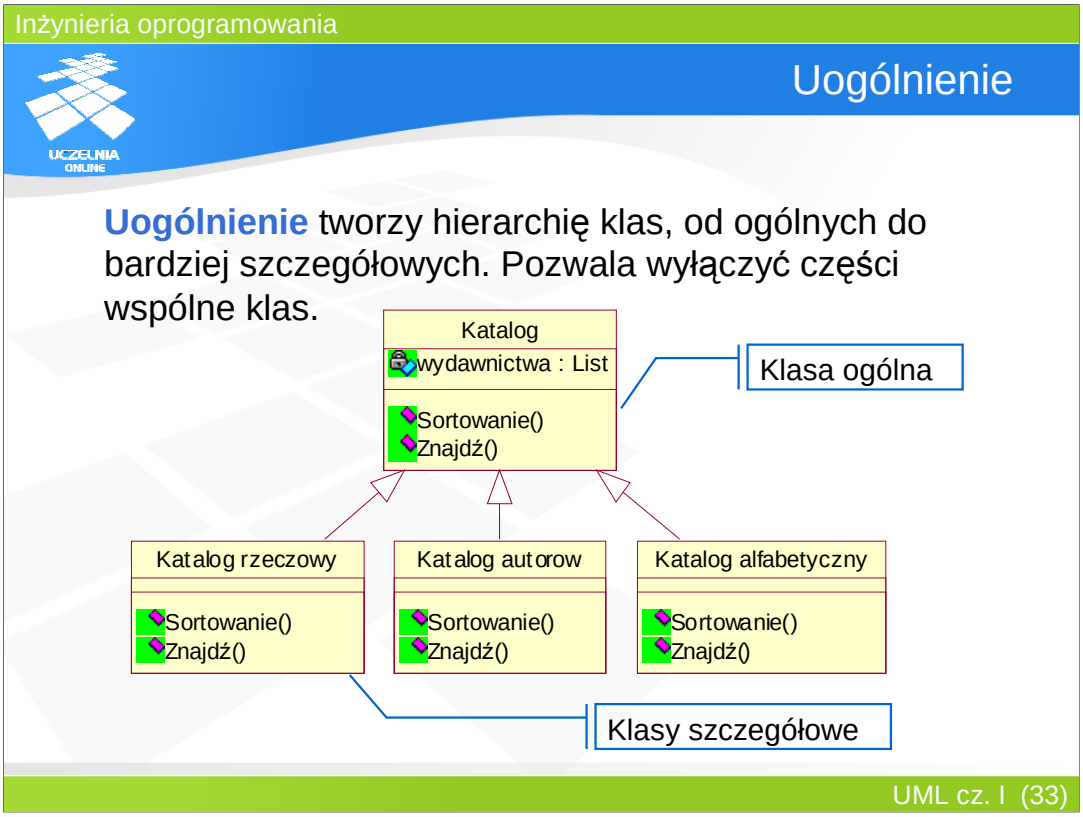
Relacja agregacji jest zaznaczana linią łączącą klasy/obiekty, zakończoną białym rombem po stronie właściciela



Kompozycja jest najsilniejszą relacją łączącą klasy. Reprezentuje relacje całość-część, w których części są tworzone i zarządzane przez obiekt reprezentujący całość. Ani całość, ani części nie mogą istnieć bez siebie, dlatego czasy ich istnienia są bardzo ściśle ze sobą związane i pokrywają się: w momencie usunięcie obiektu całości obiekty części są również usuwane.

Typowa fraza związana z taką relacją to "...jest częścią...".

Kompozycja jest przedstawiana na diagramie podobnie jak agregacja, przy czym romb jest wypełniony.



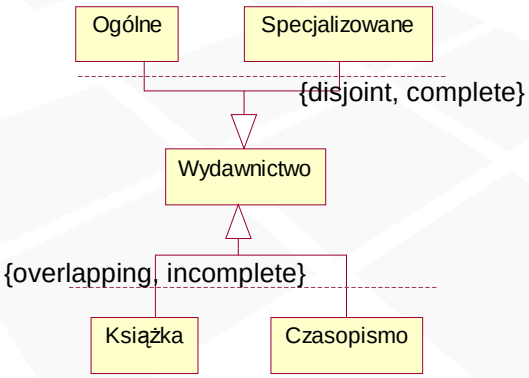
Uogólnienie posiada różne interpretacje. Na przykład, w modelu pojęciowym Katalog jest uogólnieniem Katalogu rzeczowego, jeżeli każda instancja Katalogu rzeczowego jest także instancją Katalogu. Inną interpretacją jest zastosowanie zasady podstawiania Liskov (LSP – Liskov Substitution Principle): w zamian za typ uogólniony można podstawić typ pochodny bez konieczności zmiany reszty programu.

Uogólnienie w przypadku klas często jest traktowane jako synonim dziedziczenia, podczas gdy dziedziczenie jest tylko możliwą techniką uogólniania. Inną jest np. wykorzystanie interfejsów, które pozwalają utworzyć relację uogólnienia/uszczegółowienia pomiędzy typami (dziedziczenie interfejsu) lub klasą i interfejsem (implementacja interfejsu).

Klasyfikacja

Klasyfikacja określa związek między obiektem a jego typem (klasą).

Obiekt może należeć jednocześnie do wielu typów.



{overlapping} – obiekt może należeć do kilku klas

{disjoint} – obiekt może należeć tylko do jednej klasy

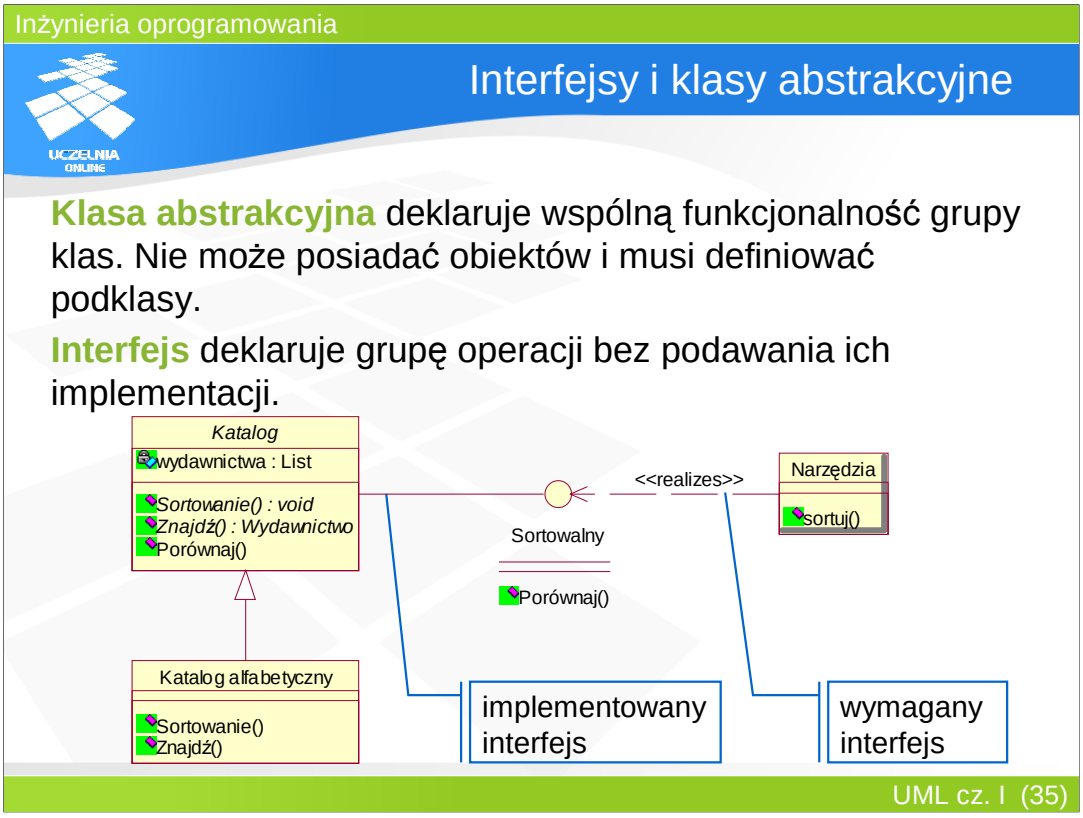
{complete} – nie istnieje więcej podklas

{incomplete} – mogą powstać kolejne podklasy

Klasyfikacja obiektu reprezentuje (w odróżnieniu od relacji uogólnienia/uszczegółowienia) związek pomiędzy obiektami a klasami. Klasyfikacja obiektu określa, z którymi typami (klasami) jest powiązany – poprzez dziedziczenie, interfejsy etc. Ponieważ obiekt może jednocześnie uczestniczyć w wielu niezależnych klasyfikacjach (a zatem posiadać wiele typów, niekoniecznie poprzez dziedziczenie), dlatego do szczegółowego określenia klasyfikacji stosowane są uściślające słowa kluczowe:

•{overlapping} oznacza, że obiekt może jednocześnie należeć do kilku klas/posiadać wiele typów. Na przykład Wydawnictwo, mimo że posiada dwie podklasy: Książka Czasopismo, może wystąpić w postaci łączącej obie te cechy. Przeciwieństwem jest słowo kluczowe {disjoint}, które narzuca rozłączność typów danych

•{complete} oznacza, że wymienione dotychczas podklasy w ramach jednej specjalizacji są wyczerpujące i nie istnieje kategoria, która znalazłaby się poza nimi. W przykładzie Wydawnictwo może być Ogólne lub Specjalizowane, i nie przewiduje się istnienia nowej podklasy. {Incomplete} przewiduje taką możliwość.



Celem tworzenia klas abstrakcyjnych i interfejsów jest identyfikacja wspólnych zachowań różnych klas, które są realizowane w różny od siebie sposób. Użycie tych mechanizmów pozwala na wykorzystanie relacji uogólniania do ukrywania (hermetyzacji) szczegółów implementacji poszczególnych klas.

Klasa abstrakcyjna reprezentuje wirtualny byt grupujący wspólną funkcjonalność kilku klas. Posiada ona sygnatury operacji (czyli deklaracje, że klasy tego typu będą akceptować takie komunikaty), ale nie definiuje ich implementacji.

Podobną rolę pełni interfejs. Różnica polega na tym, że klasa abstrakcyjna może posiadać implementacje niektórych operacji, natomiast interfejs jest czysto abstrakcyjny (choć, oczywiście interfejs i klasa w pełni abstrakcyjna są pojęciowo niemal identyczne).

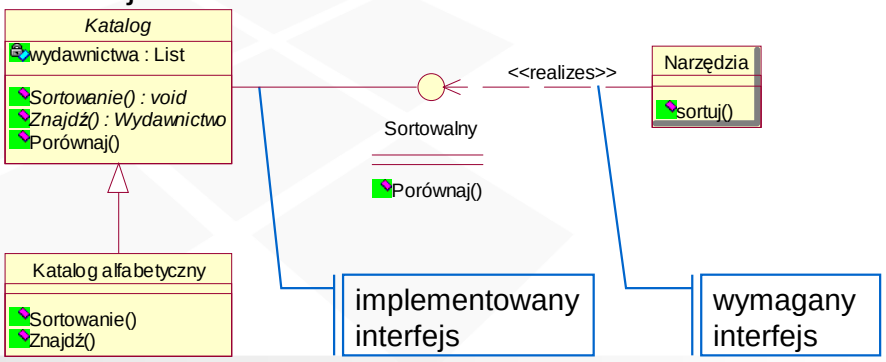
Ponieważ klasy abstrakcyjne nie mogą bezpośrednio tworzyć swoich instancji (podobnie jak interfejsy, które z definicji nie reprezentują klas, a jedynie ich typy) dlatego konieczne jest tworzenie ich podklas, które zaimplementują odziedziczone abstrakcyjne metody. W przypadku interfejsu sytuacja jest identyczna.

Przyjętym sposobem oznaczania klas i metod abstrakcyjnych jest zapisywanie ich pochyłą czcionką lub opatrywanie słowem kluczowym {abstract}.

Interfejsy i klasy abstrakcyjne

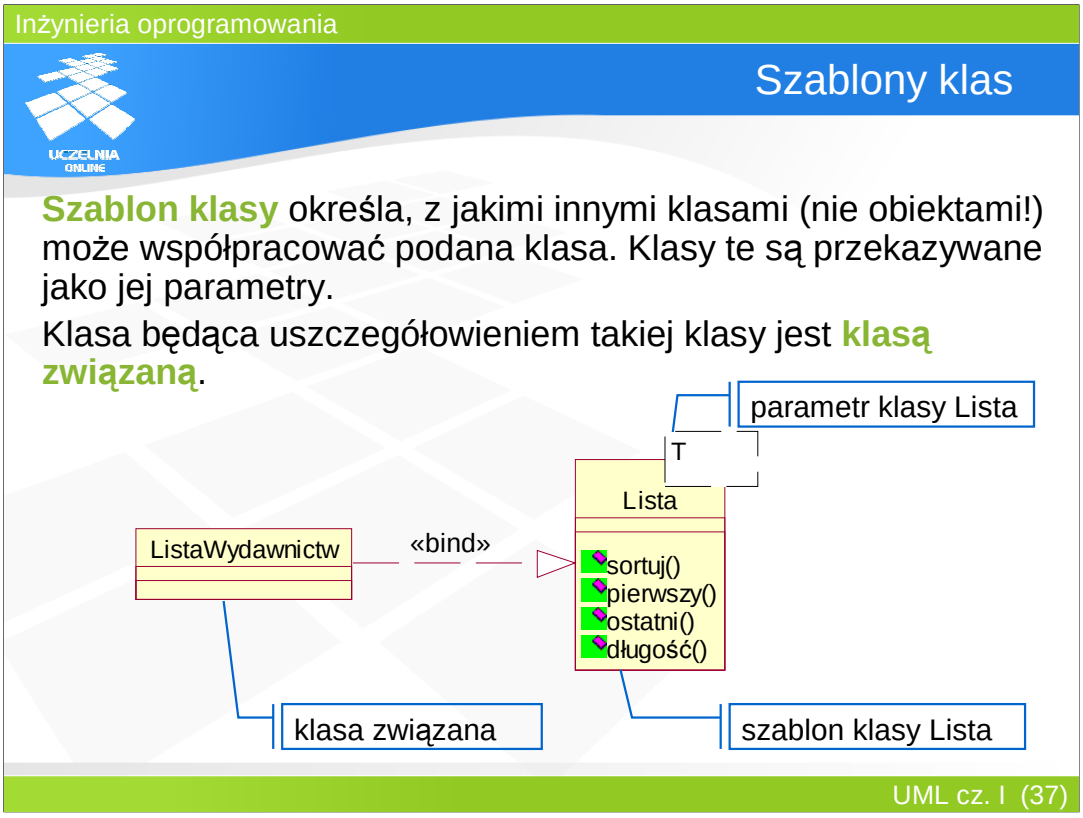
Klasa abstrakcyjna deklaruje wspólną funkcjonalność grupy klas. Nie może posiadać obiektów i musi definiować podklasy.

Interfejs deklaruje grupę operacji bez podawania ich implementacji.



W przykładzie Katalog jest klasą abstrakcyjną posiadającą abstrakcyjne metody *Sortowanie()* i *Znajdź()*. Metody te posiadają implementacje w podklasie Katalog alfabetyczny.

Katalog implementuje interfejs Sortowalny (z metodą *Porównaj()*), który z kolei jest wymagany przez klasę pomocniczą Narzędzia.



Szablony klas to pojęcie wywodzące się z języka C++. Oznaczają one klasy, których definicja wymaga podania argumentów będących innymi klasami. W ten sposób szablon klasy jest swego rodzaju niepełną klasą, która dopiero po ukonkretnieniu może zostać użyta. Na przykład, klasa Lista może przechowywać obiekty pewnego typu. Typ ten może stać się parametrem tej klasy: w ten sposób utworzony zostanie szablon listy dla potencjalnie dowolnego typu. Klasa stanowiąca ukonkretnienie szablonu (ListaWydawnictw) została sparametryzowana (związana) typem Wydawnictwo, dzięki czemu może być już wykorzystana do tworzenia obiektów.

Podobną koncepcję wprowadzono także do innych języków programowania, np. Java, pod nazwą typów generycznych.

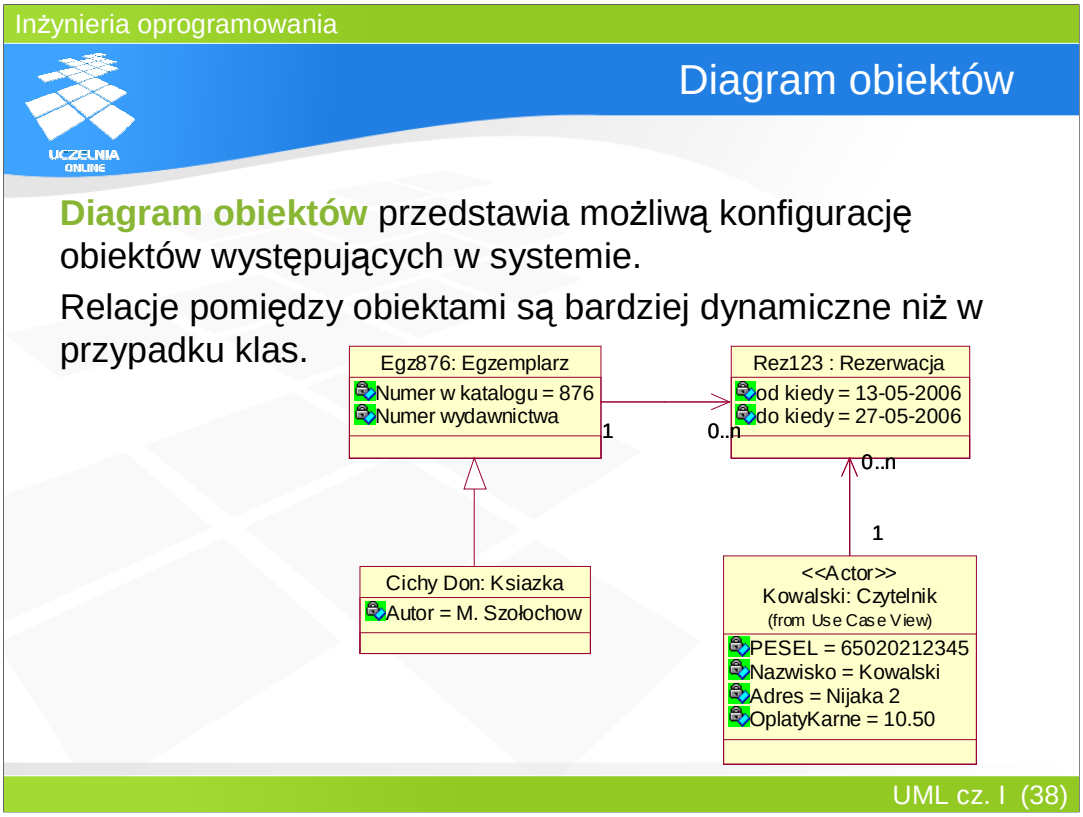


Diagram obiektów (ang. *object diagram*) prezentuje możliwą konfigurację obiektów w określonym momencie, jest pewnego rodzaju instancją diagramu klas, w której zamiast klas przedstawiono ich obiekty.

Diagram ten posługuje się identycznymi symbolami co diagram klas, jednak, dla odróżnienia obiektów od klas, nazwy instancji są podkreślone. Ponadto, nazwa składa się z nazwy obiektu i nazwy klasy, oddzielonych dwukropkiem. Obie części nazwy można pominąć, więc aby uniknąć nieporozumień, jedna część nazwy oznacza nazwę obiektu, a sama nazwa klasy musi być zawsze poprzedzona dwukropkiem.

Diagramy obiektów przydają się w przypadku szczególnie skomplikowanych zależności, których nie można przedstawić na diagramie klas. Wówczas przykładowe konfiguracje obiektów pomagają w zrozumieniu modelu.

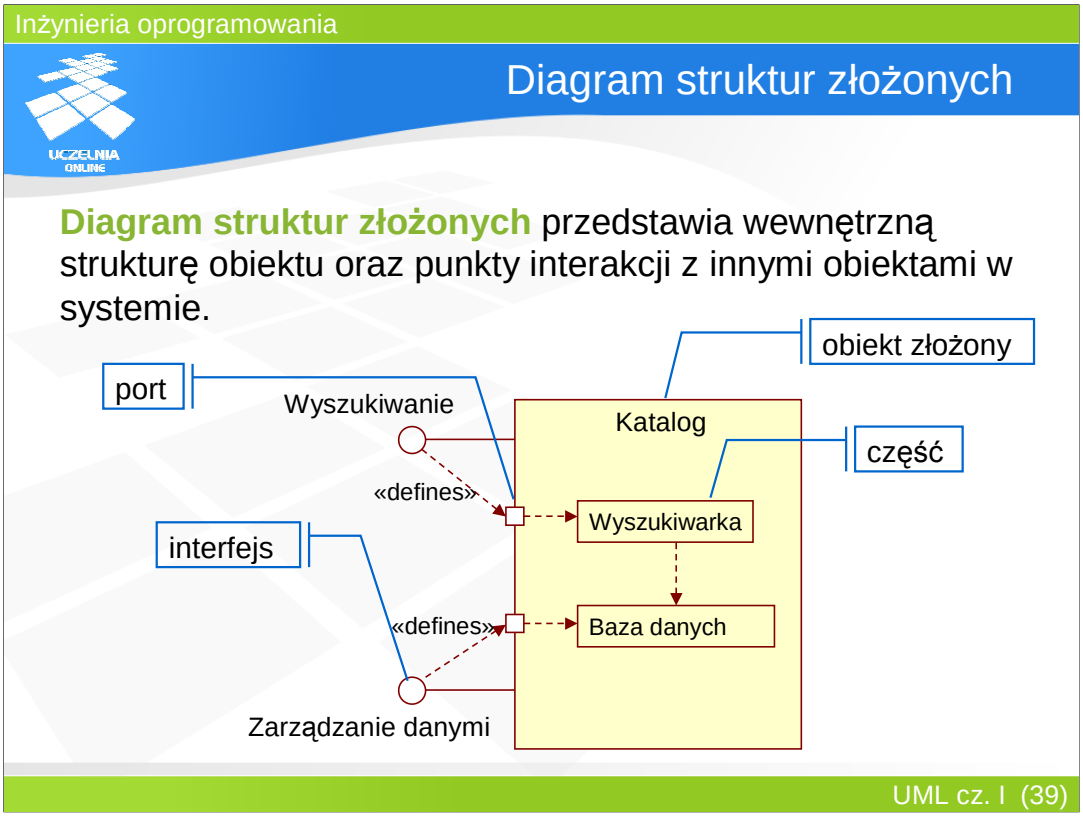


Diagram struktur złożonych (ang. *composite structure diagram*) przedstawia hierarchicznie wewnętrzną strukturę złożonego obiektu z uwzględnieniem punktów interakcji z innymi częściami systemu.

Obiekt składa się z części, które reprezentują poszczególne składowe obiektu realizujące poszczególne funkcje obiektu. Komunikacja pomiędzy obiektem, a jego środowiskiem przebiega poprzez port (oznaczany jako mały prostokąt umieszczony na krawędzi obiektu). Porty są połączone z częściami obiektu, które są odpowiedzialne za realizacje tych funkcji.

Diagramy struktur złożonych mogą także zawierać interfejsy wewnętrzne (równoważne klasom w pełni abstrakcyjnym) i interfejsy udostępnione (widoczne na zewnątrz obiektu; dzielą się na interfejsy wymagane i oferowane).

Podsumowanie

- UML powstał w wyniku połączenia różnych notacji i metodyk modelowania
- UML opisuje system w postaci 4 perspektyw wewnętrznych i 1 zewnętrznej
- Diagram przypadków użycia opisuje funkcje systemu i jego użytkowników
- Diagram klas określa statyczną strukturę logiczną systemu
- Diagram obiektów pokazuje możliwą konfigurację obiektów w systemie

Podczas wykładu przedstawiono genezę języka UML, jego strukturę, oraz trzy typy diagramów: przypadków użycia, klas, obiektów i struktury złożonej.