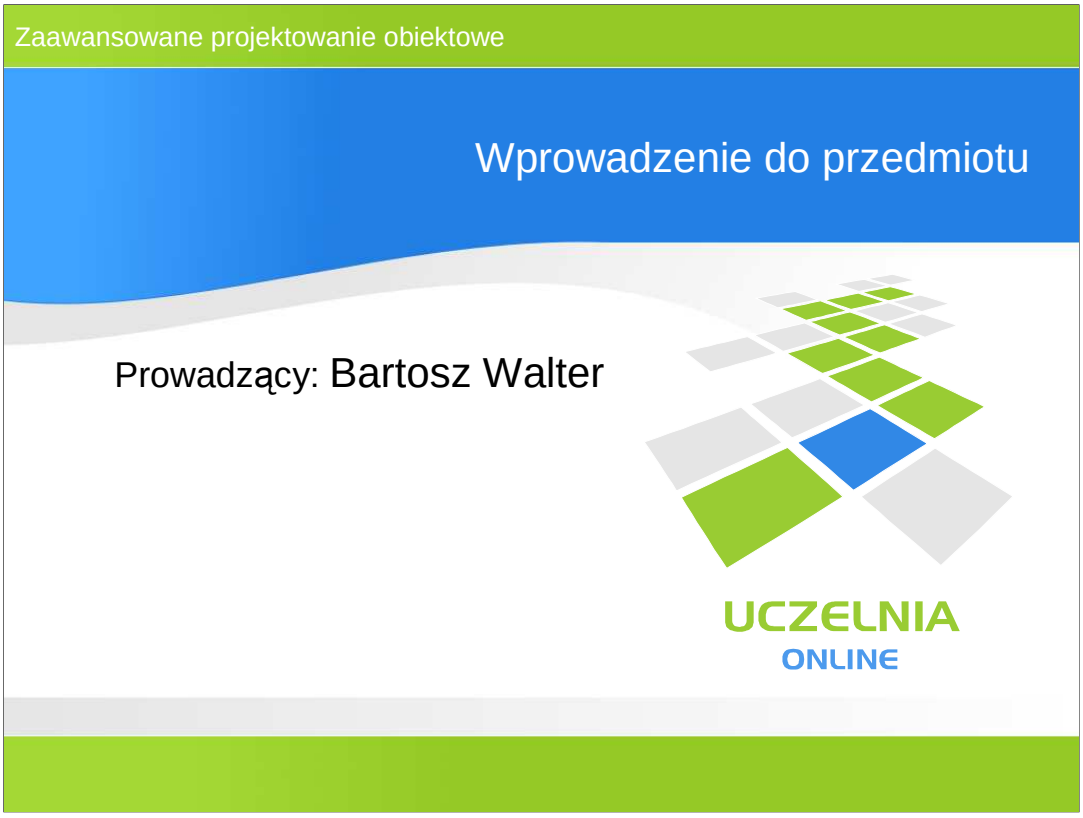
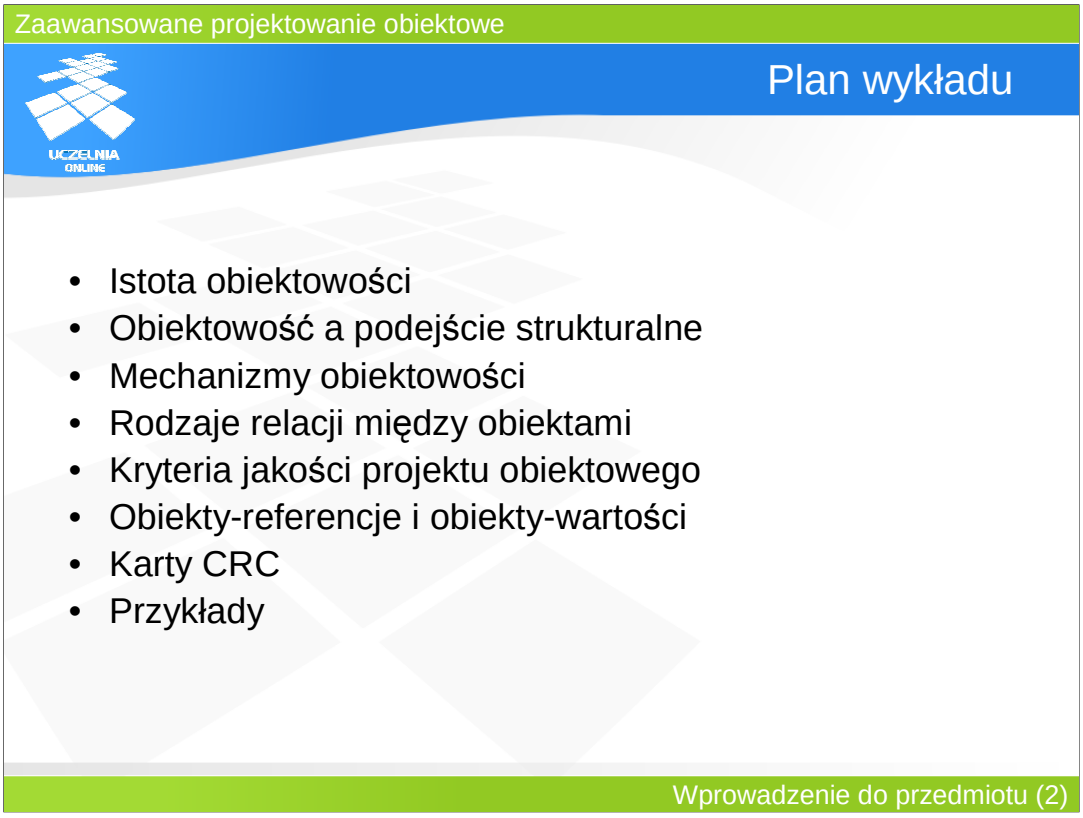
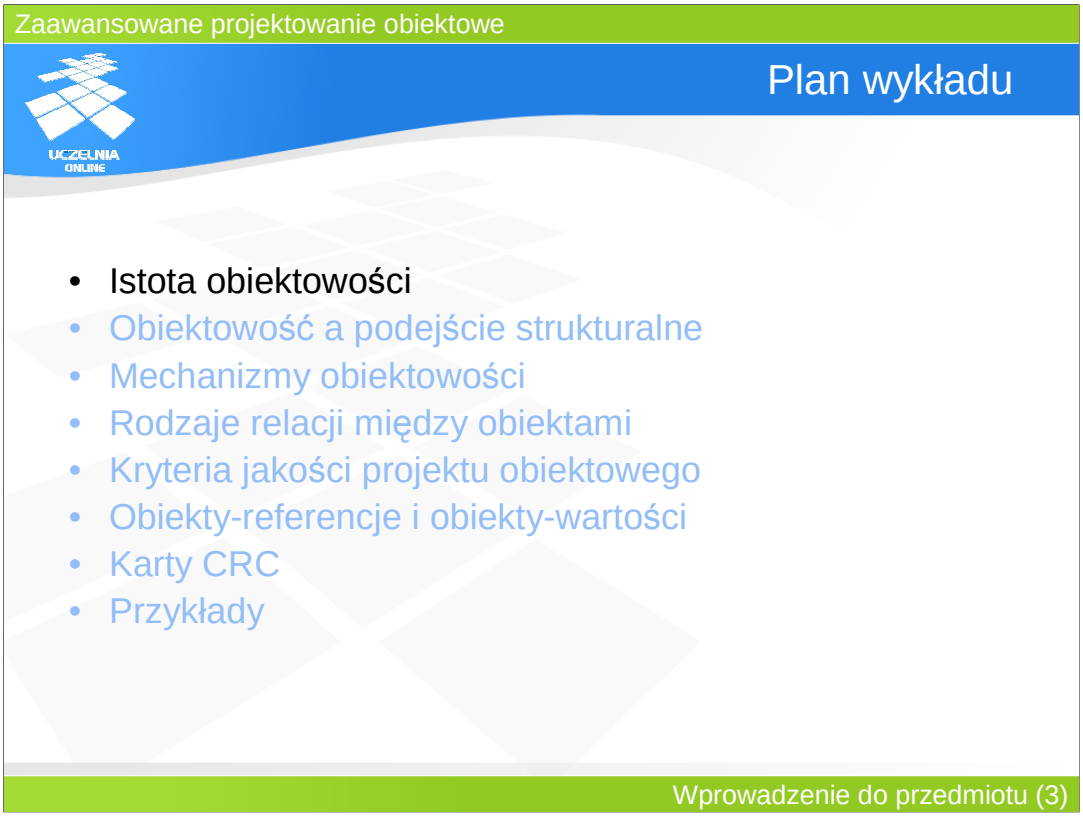

Zaawansowane projektowanie obiektowe
Wprowadzenie do przedmiotu
Prowadzący: Bartosz Walter
UCZELNIA
ONLINE

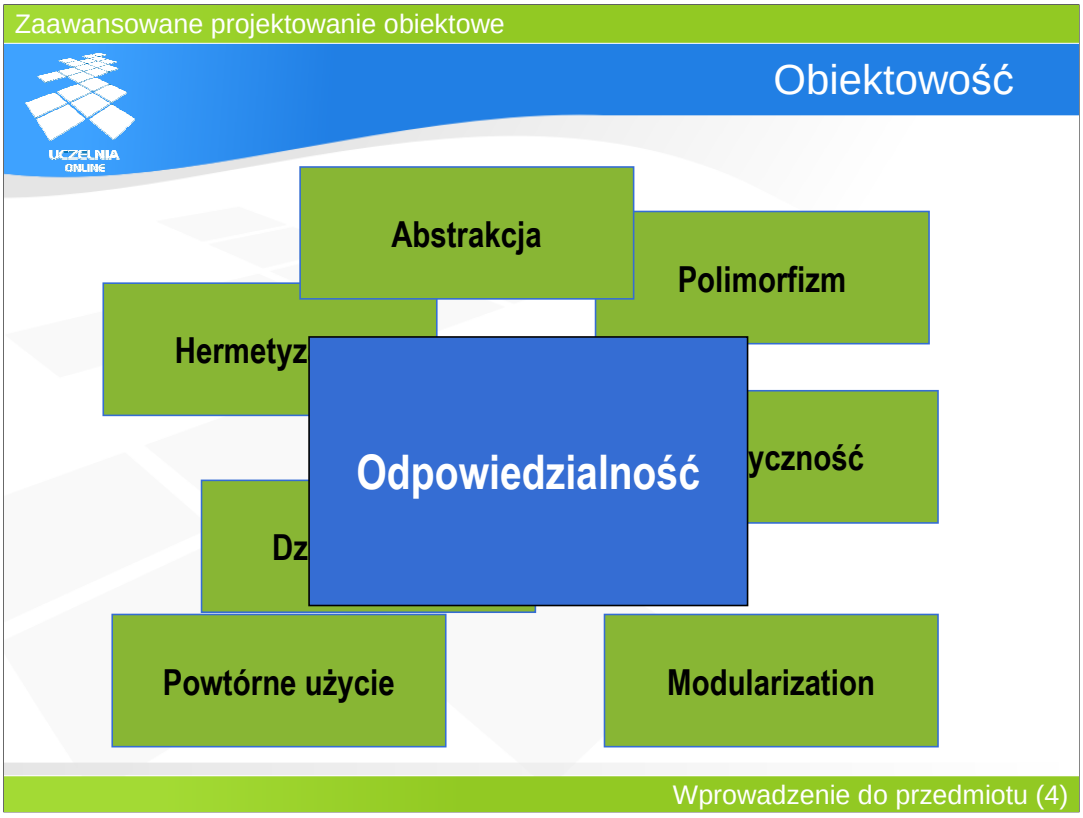


Moduł ten rozpoczyna cykl wykładów poświęconym projektowaniu obiektowemu. Dotyczą one przede wszystkim wzorców obiektowych i refaktoryzacji, ale będzie w nich mowa także m.in. o metrykach obiektowych, testowaniu jednostkowym, programowaniu aspektowym i komponentach.

Podczas bieżącego wykładu przypomniane zostaną podstawowe pojęcia związane z obiektowością i związane z nią mechanizmy. Ponadto będzie mowa o relacjach, jakie mogą łączyć obiekty, o kryteriach oceny jakości projektu obiektowego, a także o obiektach-referencjach i obiektach-wartościach. Ostatnim elementem wykładu będzie krótkie wprowadzenie do metody projektowania opartej na kartach CRC.

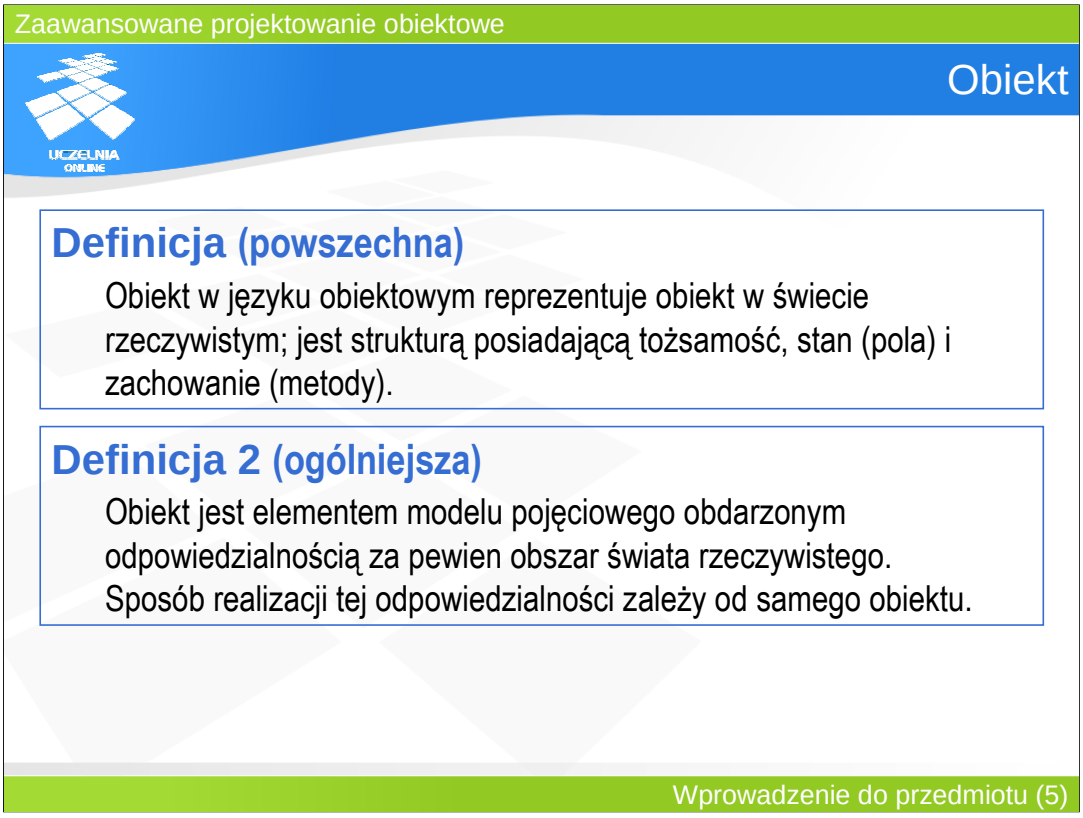


Wykład rozpocznie się od przypomnienia najważniejszej cechy paradygmatu obiektowego i jej znaczenia dla procesu projektowania.



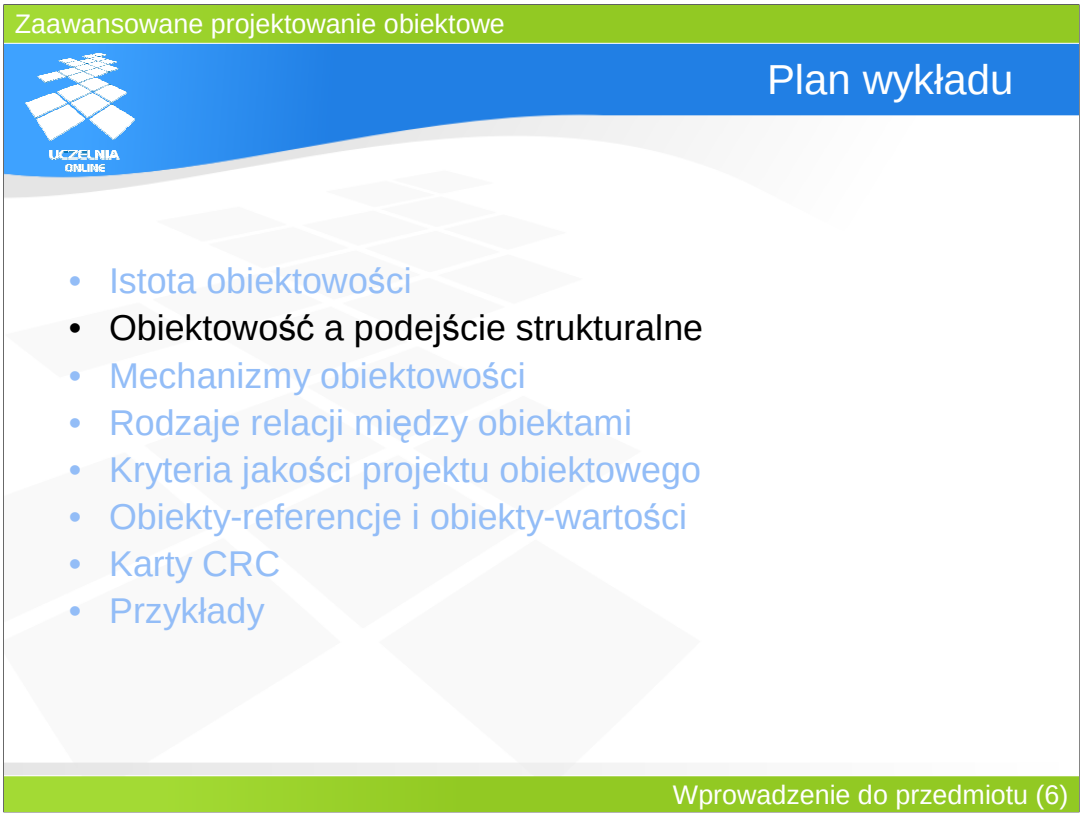
Z pojęciem obiektowości związanych jest wiele terminów, określanych jako najważniejsze cechy świata obiektowego. Wśród nich znajdują się abstrakcja, polimorfizm, hermetyzacja, dziedziczenie i wiele innych. Szczególnie dziedziczenie i hermetyzacja są często traktowane jako swego rodzaju papierek lakmusowy stwierdzający, czy system informatyczny został zbudowany w sposób obiektowy, czy też nie.

Oczywiście, taki test nie daje prawidłowych wyników, przynajmniej nie całkowicie prawidłowych. Istnieją systemy stworzone z wykorzystaniem strukturalnego języka programowania (np. biblioteka graficzna Motif, napisana w języku C), które zostały zaprojektowane w sposób obiektowy. Dlatego cechą, która w rzeczywistości wyróżnia obiektowy sposób myślenia od innych, w szczególności strukturalnego, jest pojęcie odpowiedzialności. Słowo to oznacza, że obiekt przyjmuje na siebie pewne obowiązki wobec innych obiektów, i zapewnia ich wykonanie, ukrywając przed swoimi klientami sposób ich realizacji.

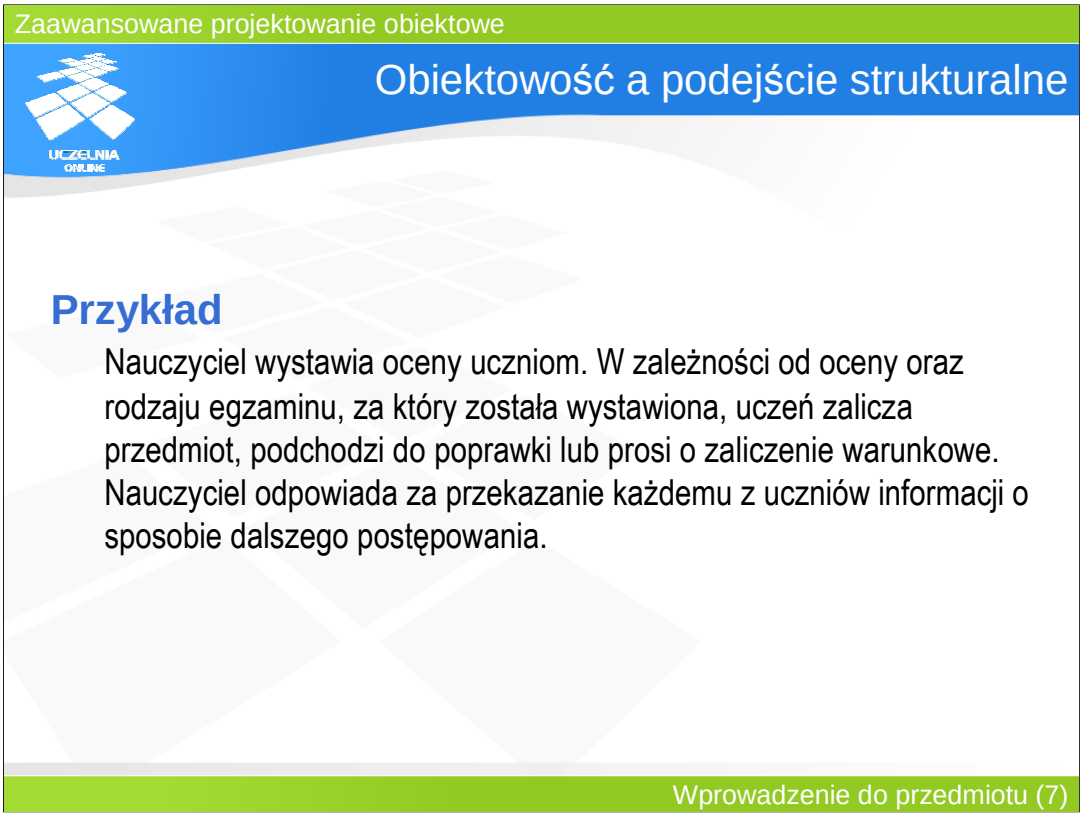


Istnieje wiele definicji obiektu. Zwykle dotyczą one jego fizycznej i logicznej struktury, podkreślając współistnienie w nim pól i metod. Te definicje są prawdziwe, jednak nie obejmują najważniejszego aspektu obiektowości, czyli właśnie odpowiedzialności.

Dlatego z punktu widzenia projektowania (a nie programowania) obiektowego trafniejsza wydaje się definicja, zgodnie z którą obiekt odpowiada za pewien fragment rzeczywistości. Odpowiedzialność oznacza, poza zobowiązaniem do realizacji pewnych zadań, także ukrycie sposobu ich wykonania. Natomiast elementy wymienione w pierwszej definicji są mechanizmami wykorzystywanymi do osiągnięcia celu opisanego w drugiej.



Aby lepiej uzmysłowić sobie znaczenie odpowiedzialności i konsekwencje jej stosowania przy projektowaniu oprogramowania, podczas kolejnej części wykładu porównane zostaną podejście strukturalne do podejścia obiektowego zastosowanych do rozwiązania tego samego problemu.



Podejście obiektowe różni się od strukturalnego nie jedynie środkami wyrazu (dziedziczeniem, polimorfizmem etc.), które są jedynie narzędziami do właściwego przydziału odpowiedzialności do obiektów. Rzeczywista różnica polega właśnie na innym sposobie opisywania odpowiedzialności.

Rozpatrzmy następujący opis rzeczywistości: nauczyciel wystawia oceny. Ocena determinuje sposób dalszego postępowania ucznia, który ją otrzymał:

•w przypadku oceny dostatecznej lub wyższej zalicza on przedmiot,

•w przypadku oceny niedostatecznej z pierwszego egzaminu jest kierowany na egzamin poprawkowy lub może poprosić o zaliczenie warunkowe.

•w przypadku oceny niedostatecznej z kolejnego egzaminu jest skreślany z listy studentów.

Obiektowość a podejście strukturalne

Rozwiązanie 1 (dekompozycja funkcjonalna)

1. Utwórz listę ocen i uczniów, którzy je otrzymali
2. Dla każdego ucznia z listy
   a) określ jego ocenę
   b) określ sposób postępowania
   c) przekaż informacje uczniowi

Stosując dekompozycję funkcjonalną, charakterystyczną dla metod strukturalnych, można rozwiązać problem wykonując następujący algorytm:

1. Nauczyciel tworzy listę uczniów wraz z ocenami
2. Nauczyciel dla każdego ucznia z listy na podstawie jego oceny określa sposób dalszego postępowania, i odpowiednio instruuje ucznia.

Takie rozwiązanie jest, oczywiście, poprawne, jednak zwraca uwagę asymetria zadań stojących przed obiektami istniejącymi w tym systemie. Pełna odpowiedzialność za wykonanie każdego zadania spoczywa wyłącznie na Nauczycielu, natomiast Uczeń jest elementem całkowicie biernym, odbierającym informacje.

Obiektowość a podejście strukturalne

Rozwiązanie 2 (obiektowe)

Nauczyciel

1. Utwórz listę ocen i Uczniów, którzy je otrzymali
2. Utwórz informację o sposobie postępowania w zależności od otrzymanej oceny
3. Udostępnij oceny i informację

Uczeń

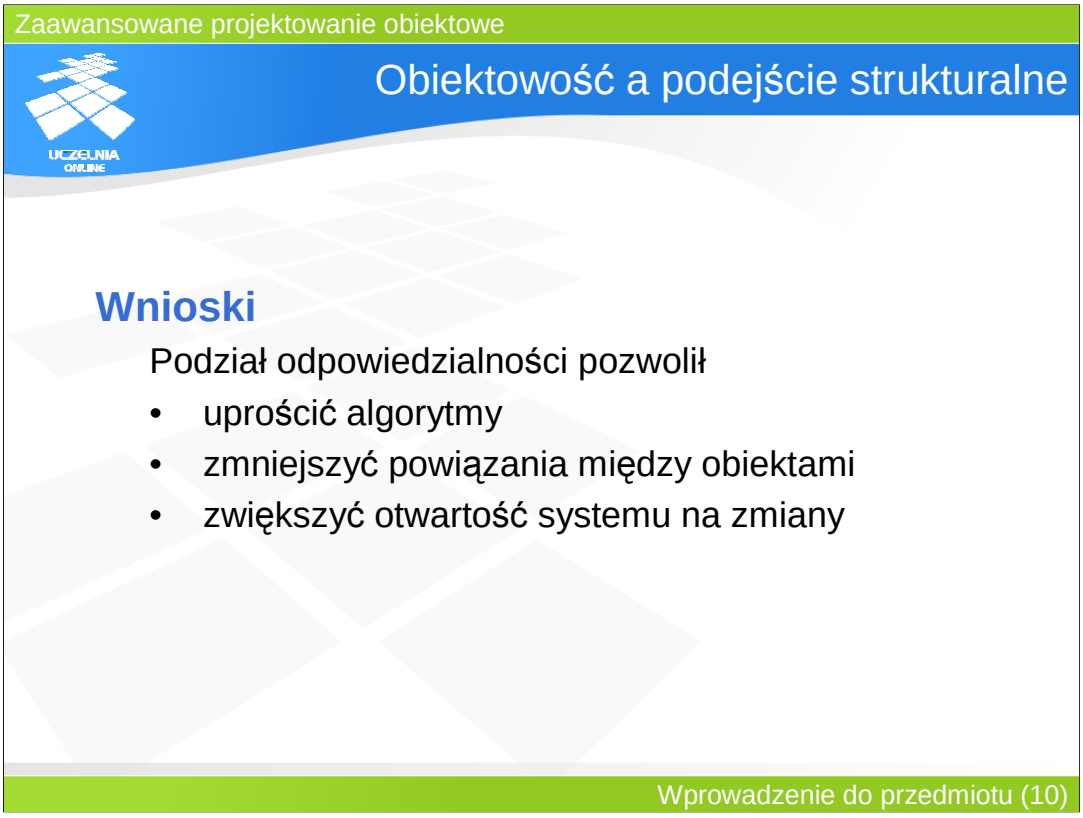
1. Znajdź swoją ocenę na liście udostępnionej przez Nauczyciela
2. Określ sposób postępowania na podstawie oceny
3. Postąp zgodnie z instrukcją

Podejście obiektowe polega na podziale odpowiedzialności za wykonanie określonego zadania pomiędzy obiekty. Linia podziału może przebiegać w różnych miejscach (dlatego nie istnieje jeden projekt obiektowy dla każdego zadania), w zależności od wielu czynników (np. wydajności, sposobu komunikacji między obiektami etc.). Oto przykładowe rozwiązanie:

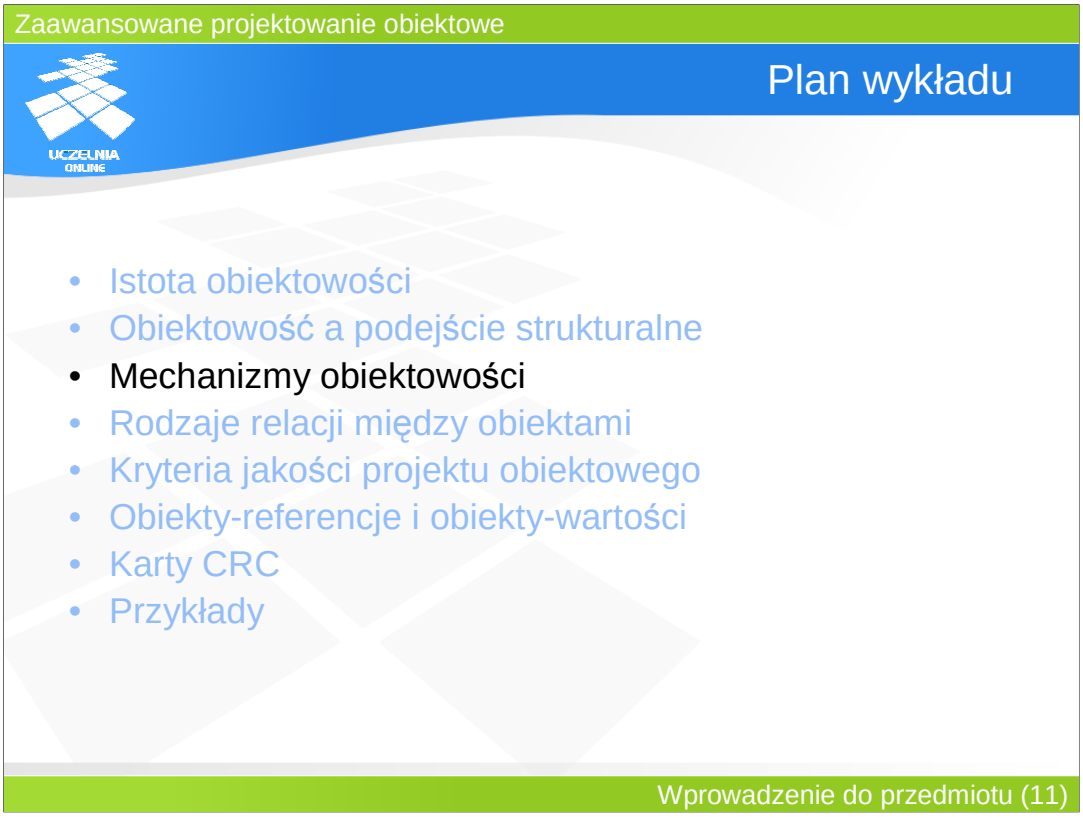
Nauczyciel, podobnie jak w poprzednim przypadku, tworzy listę uczniów i ocen, oraz oddzielną informację na temat postępowania w zależności od otrzymanej oceny. Jednak kolejny krok polega nie na przekazaniu informacji, a na jej udostępnieniu. W ten sposób odpowiedzialność za zdobycie informacji spada na Ucznia.

Uczeń musi zatem znaleźć swoją ocenę, określić sposób swojego dalszego działania, oraz postąpić zgodnie z nim.

Warto zwrócić uwagę, że rozwiązanie to, przenosząc część zadań na Ucznia, jednocześnie zmniejsza stopień powiązań pomiędzy Uczniem i Nauczycielem i wiedzy, jaką muszą posiadać o sobie nawzajem. Nauczyciel nie musi już znać adresu każdego Ucznia, aby przekazać mu ocenę, co więcej – Nauczyciel w ogóle nie musi kontaktować się z Uczniem, ponieważ lista Uczniów służy wyłącznie do określenia ich ocen.

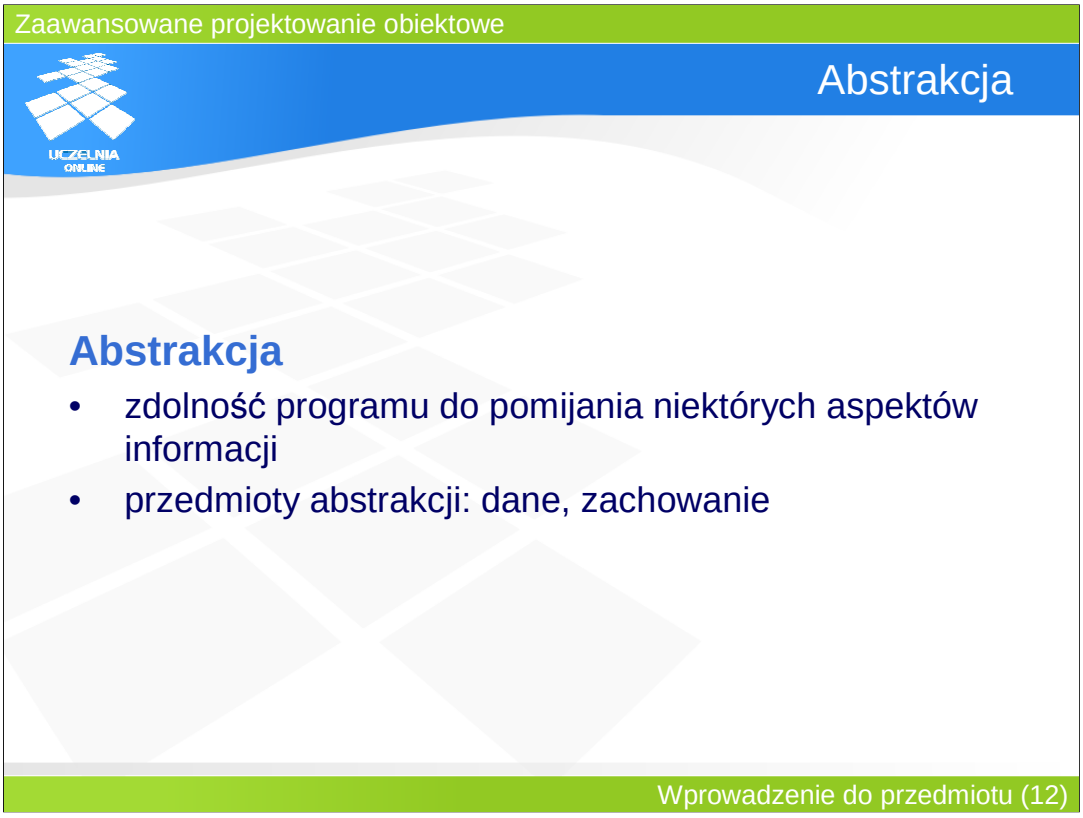


Krótko podsumowując, dokonany podział odpowiedzialności pozwolił uprościć algorytmy stosowane przez wszystkie obiekty w systemie, osłabić powiązania i zależności pomiędzy obiektami (Nauczyciel nie musi już znać każdego Ucznia, żeby przekazać mu informację o ocenie) oraz zwiększyć otwartość systemu na zmiany, jakie mogą w nim zajść w przyszłości (np. pojawienie się nowej kategorii oceny, która spowoduje konieczność wykonania określonych kroków przez Ucznia).



Wspomniane wcześniej mechanizmy obiektowości – dziedziczenie, hermetyzacja, polimorfizm i inne – są często mylone z istotą tego paradygmatu i przedstawiane jako najważniejszy jego aspekt. W praktyce jednak obiektowość bez nich jest ułomna (choć jej realizacja jest możliwa, jak pokazują przykłady obiektowych implementacji wykonanych w językach strukturalnych).

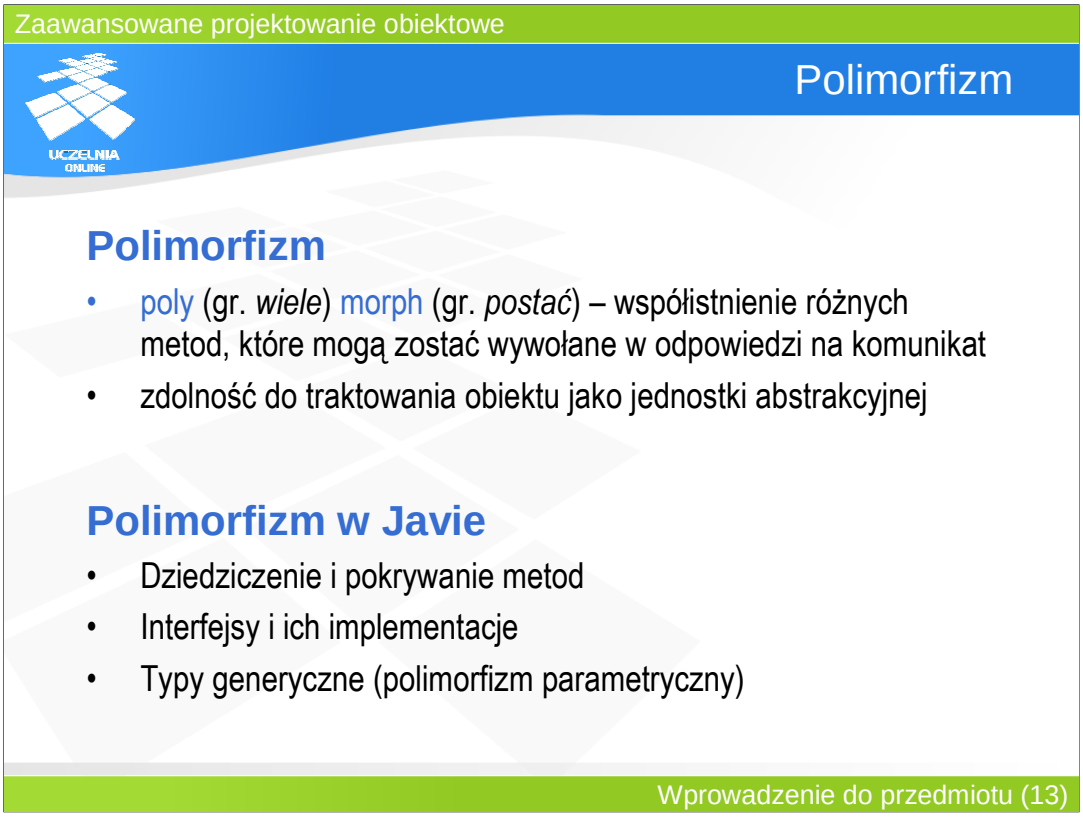
W tej części wykładu omówione zostaną najpopularniejsze mechanizmy obiektowe, ich praktyczne znaczenie i sposób wykorzystania.



Abstrakcję trudno określić mianem mechanizmu: jest własnością klasy lub całego systemu, która wynika z zastosowania innych mechanizmów. Jednak jej rola w obiektowości jest na tyle ważna, że zasługuje ona na krótkie przypomnienie.

Abstrakcja systemu polega na jego zdolności do ignorowania niektórych decyzji projektowych lub możliwości odłożenia ich w czasie. Przejawia się ona w zróżnicowany sposób: poprzez właściwe użycie interfejsów i klas abstrakcyjnych, ograniczanie liczby i rodzaju powiązań, stosowanie warstw pośredniczących w komunikacji między obiektami etc.

Abstrakcja jest podstawowym czynnikiem wpływającym na koszt pielęgnacji oprogramowania. System zbudowany z abstrakcyjnych komponentów może być łatwo rozszerzany, ponieważ zmiany nie są widoczne poza tym komponentem.

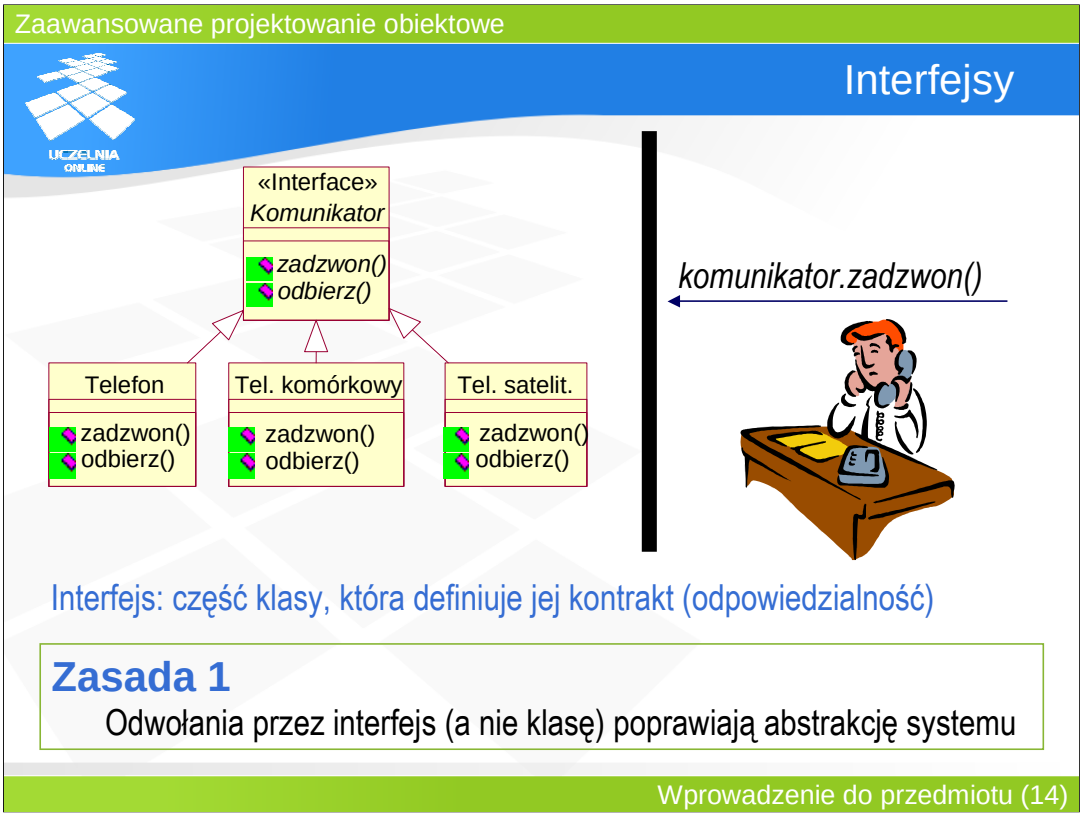


Polimorfizm to mechanizm umożliwiający współistnienie różnych metod, które mogą zostać wykonane w odpowiedzi na komunikat odebrany przez obiekt, przy czym wybór metody dokonywany jest dynamicznie, na podstawie typu obiektu odbiorcy. Polimorfizm pozwala zatem traktować odbiorcę komunikatu w sposób abstrakcyjny, bez znajomości jego typu, ponieważ metoda polimorficzna zostanie wybrana automatycznie, bez wiedzy wywołującego ją obiektu.

W języku Java istnieją trzy mechanizmy polimorficzne. Pierwszym jest dziedziczenie, które umożliwia polimorficzne pokrywanie metod. Mechanizm ten jest obecny we wszystkich współczesnych językach obiektowych. W Javie wszystkie metody są dziedziczone polimorficznie, natomiast w C++ metody takie muszą zostać zadeklarowane jako wirtualne.

Drugim mechanizmem są interfejsy i możliwość ich implementacji w klasach. Interfejsy są w języku Java substytutem obecnego w C++ mechanizmu wielokrotnego dziedziczenia. W porównaniu z nim interfejsy są jednak bezpieczniejszym rozwiązaniem, ponieważ służą jedynie do przekazania zewnętrznego kontraktu klasy (jej typu), natomiast nie umożliwiają współdzielenia implementacji.

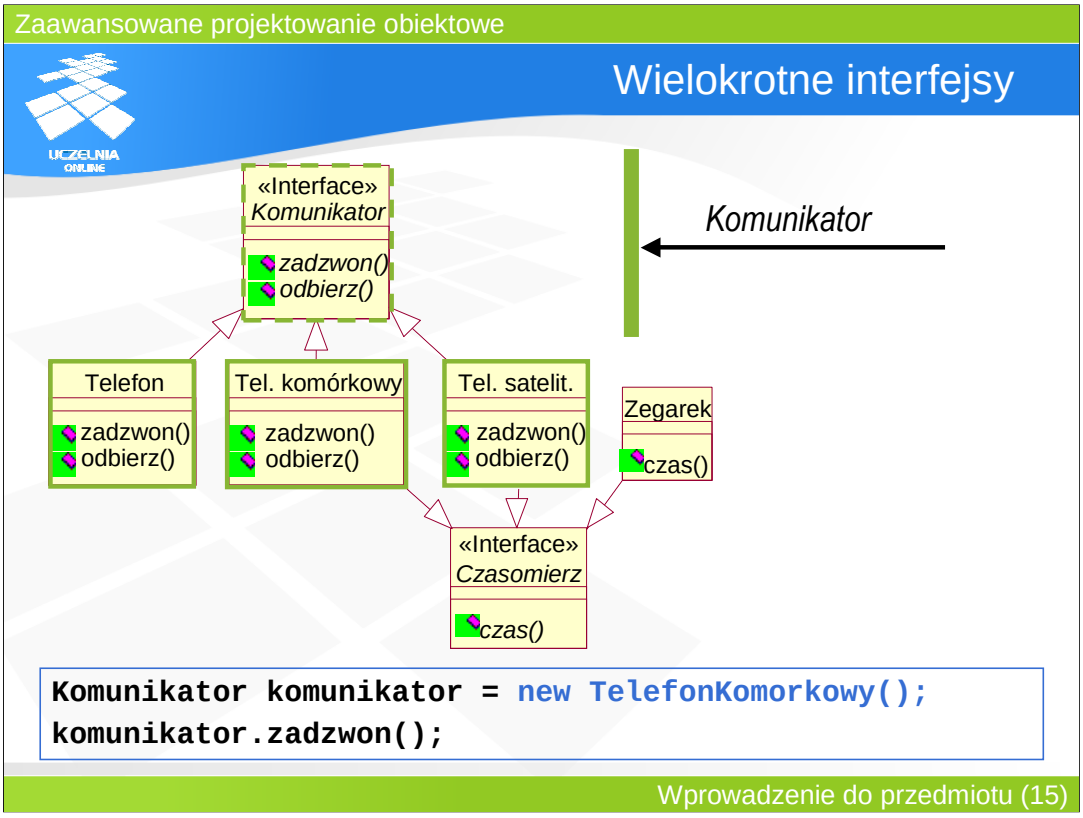
Ostatnim mechanizmem jest polimorfizm parametryczny, obecny w Javie w postaci tzw. typów generycznych. Pozwala on tworzyć szablony klas, w których niektóre kontrakty (typy) są przekazywane jako parametry klasy. Dzięki temu możliwe jest stworzenie nieograniczonej rodziny klas, o podobnej strukturze, ale różnych kontraktach.



Wspomniane wcześniej interfejsy w Javie są mechanizmem pozwalającym poprawiać abstrakcję systemu. Interfejs jest zbiorem kontraktów, które klasa go implementująca musi wypełniać. Ponieważ jednak nie narzuca on ograniczeń na sposób implementacji poszczególnych metod, dlatego jest najprostszym mechanizmem, który umożliwia stosowanie polimorfizmu.

Na slajdzie przedstawiono przykład zastosowania interfejsów: interfejs Komunikator definiuje dwie metody: *zadzwon()* oraz *odbierz().* Metody te są implementowane w trzech klasach: Telefonie, Telefonie komórkowym oraz Telefonie satelitarnym. Z punktu widzenia użytkownika funkcjonalnie nie ma żadnej różnicy pomiędzy tymi telefonami, ponieważ każdy z nich jest w stanie wykonać dwie zdefiniowane w interfejsie Komunikator metody. Zatem użytkownik nie musi znać dokładnego typu telefonu i może komunikować się z nim poprzez interfejs.

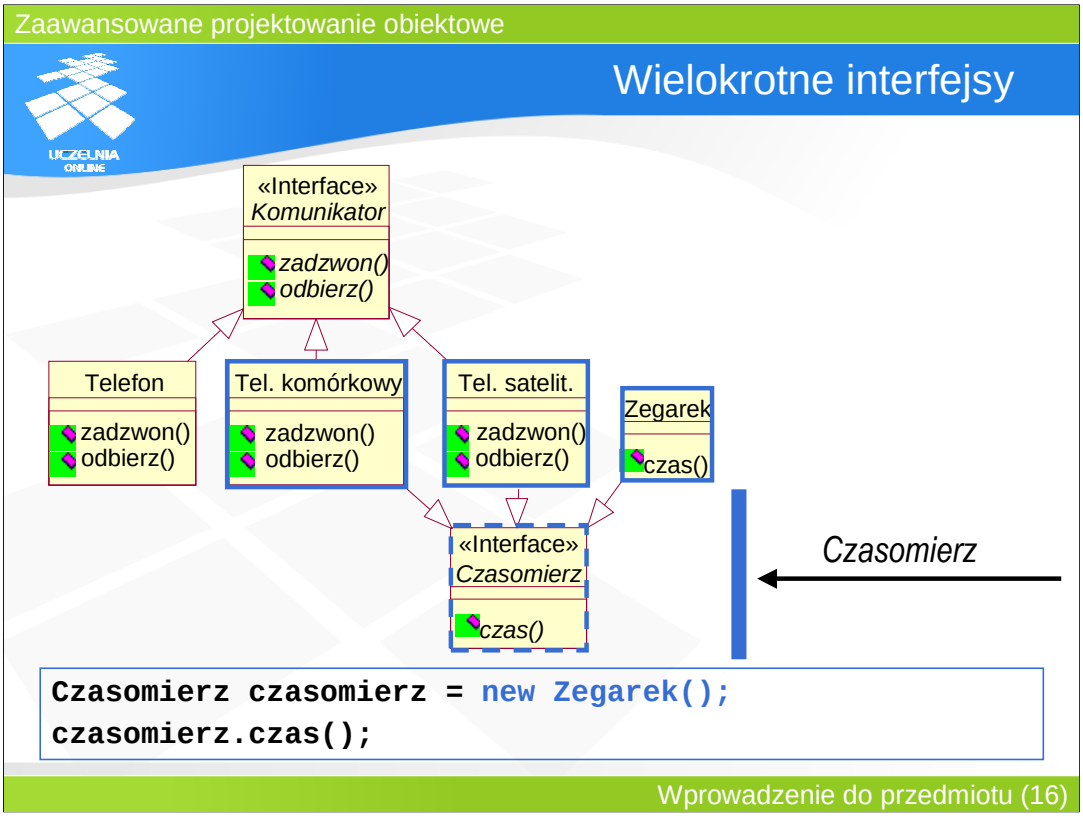
Pierwsza zasada projektowania obiektowego nakazuje zatem preferować odwołania do obiektów poprzez interfejs, ponieważ poprawiają one abstrakcję systemu.



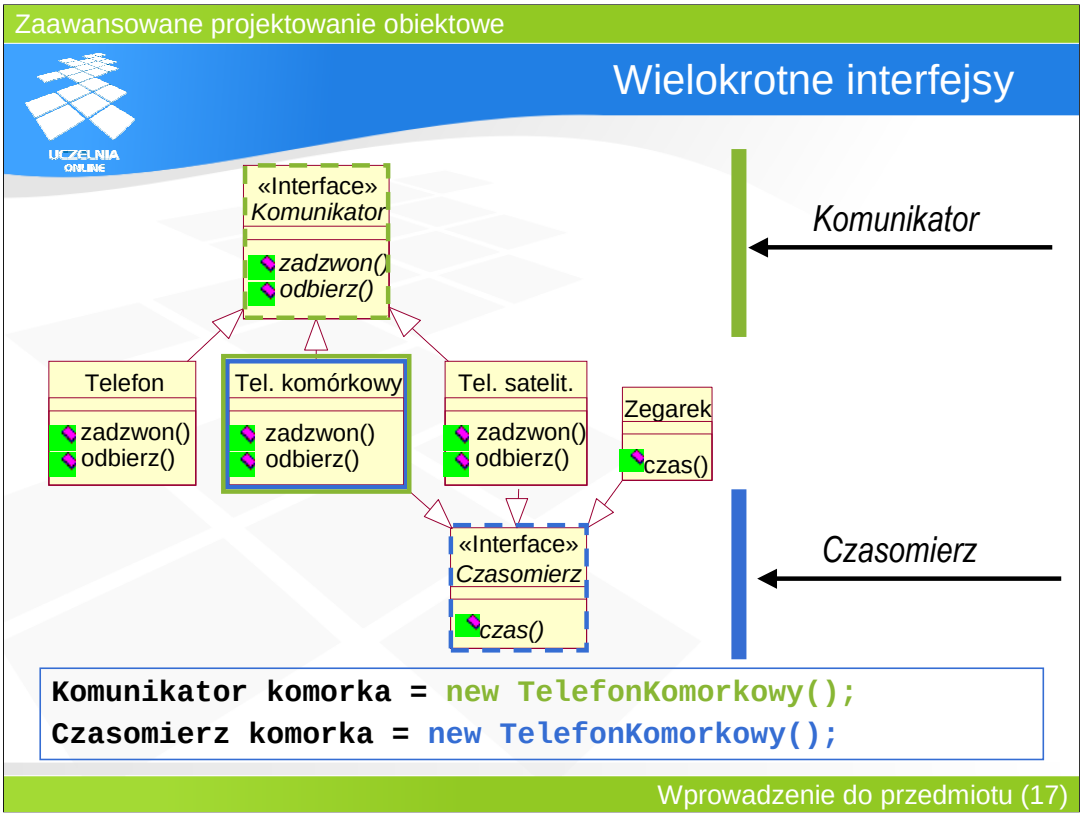
W odróżnieniu od dziedziczenia, które w Javie jest jednobazowe, obiekt w tym języku może implementować wiele interfejsów. Każdy interfejs oznacza jeden kontrakt, a zatem klasa taka przyjmuje na siebie dodatkowe obowiązki. Efektem takiego rozwiązania jest jednak możliwość użycia tej samej klasy w różnych kontekstach.

Na przykład, klasy Telefon komórkowy i Telefon satelitarny implementują jednocześnie interfejsy Komunikator oraz Czasomierz. Zatem w zależności od potrzeb, instancje tych klas mogą być użyte właśnie jako urządzenia komunikacyjne albo jako zegary.

Utworzenie instancji Telefonu komórkowego pozwala traktować go jako rodzaj Komunikatora i wywołać na nim metodę *zadzwon()*.



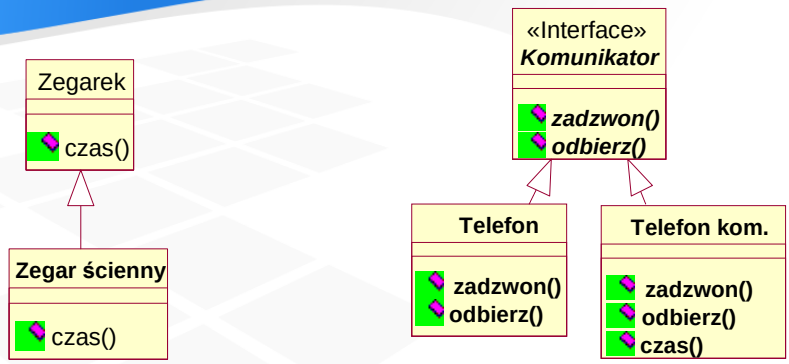
Ta sama instancja, ale potraktowana jako Czasomierz, prawidłowo odpowie na wywołanie metody *czas()* zdefiniowanej w tym interfejsie.



Slajd ten prezentuje oba interfejsy implementowane przez Telefon komórkowy. W zależności od potrzeb, może on pełnić dwie role: Komunikatora i Czasomierza

Warto zastanowić się nad wpływem tego rozwiązania na jakość projektu. Obiekt pełniący dwie role naraz posiada tę zaletę, że może być użyty w wielu kontekstach, jednak z drugiej strony zmiana wprowadzona w jednym z interfejsów może (poprzez tę klasę) spowodować konieczność modyfikacji także w drugim interfejsie. Zatem takie rozwiązanie – choć możliwe – należy uznać za uzasadnione jedynie w szczególnych sytuacjach, ponieważ może zmniejszać abstrakcję tej klasy.

Dziedziczenie a implementacja interfejsów



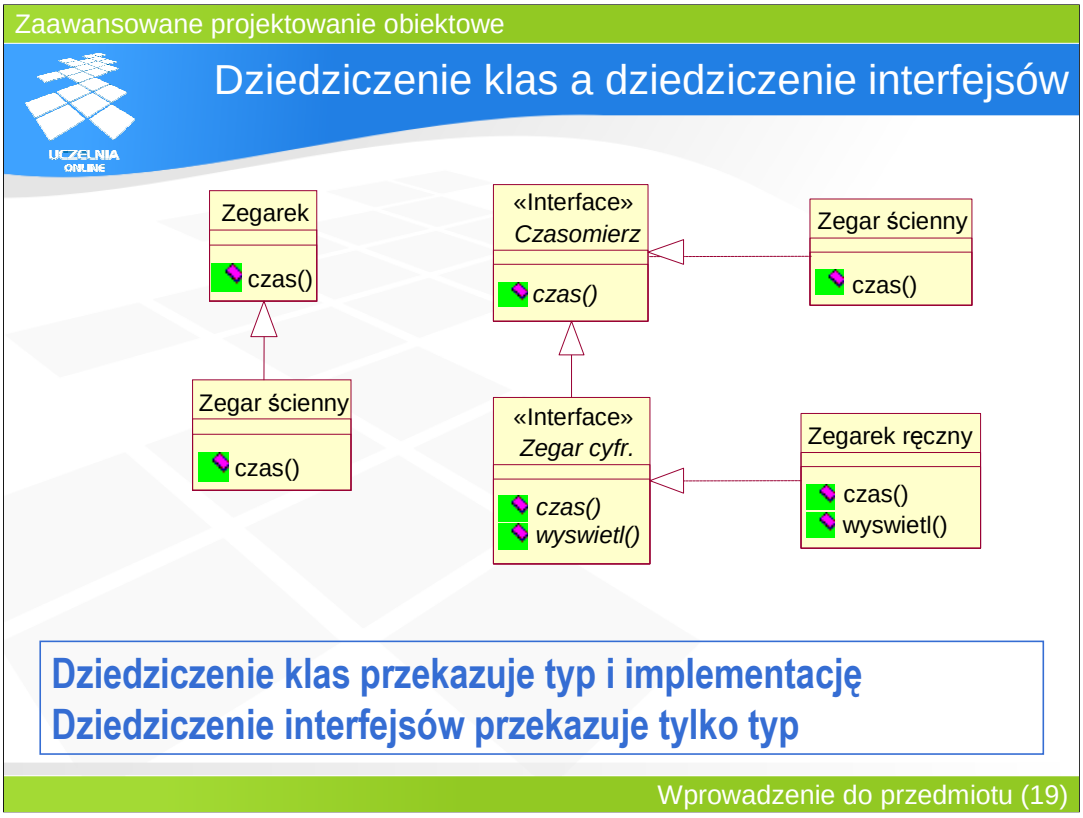
- **Zegar ścienny** jest rodzajem **Zegarka**. Posiada wszystkie jego cechy
- **Zegar ścienny** dziedziczy zarówno typ, jak i implementację
- **Telefon i Telefon kom.** są **Komunikatorami**. Oba obiekty mogą dzwonić i odbierać rozmowy
- **Telefon** i **Telefon kom.** współdzielą tylko typ

Jak wspomniano wcześniej, dziedziczenie i implementacja interfejsów są dwoma sposobami wykorzystania polimorficznego zachowania metod. Warto porównać ich cechy.

Dziedziczenie powoduje przeniesienie z nadklasy do podklasy dwóch elementów: typu oraz implementacji. Pozwala to na ponowne wykorzystanie kodu z nadklasy, ale także ściśle wiąże ze sobą obie klasy

Implementacja interfejsu pełni natomiast tylko jedną z tych ról: przeniesienie typu. Dzięki temu klasy implementujące ten sam interfejs mogą być zróżnicowane w znacznie większym stopniu niż podklasy tej samej nadklasy.

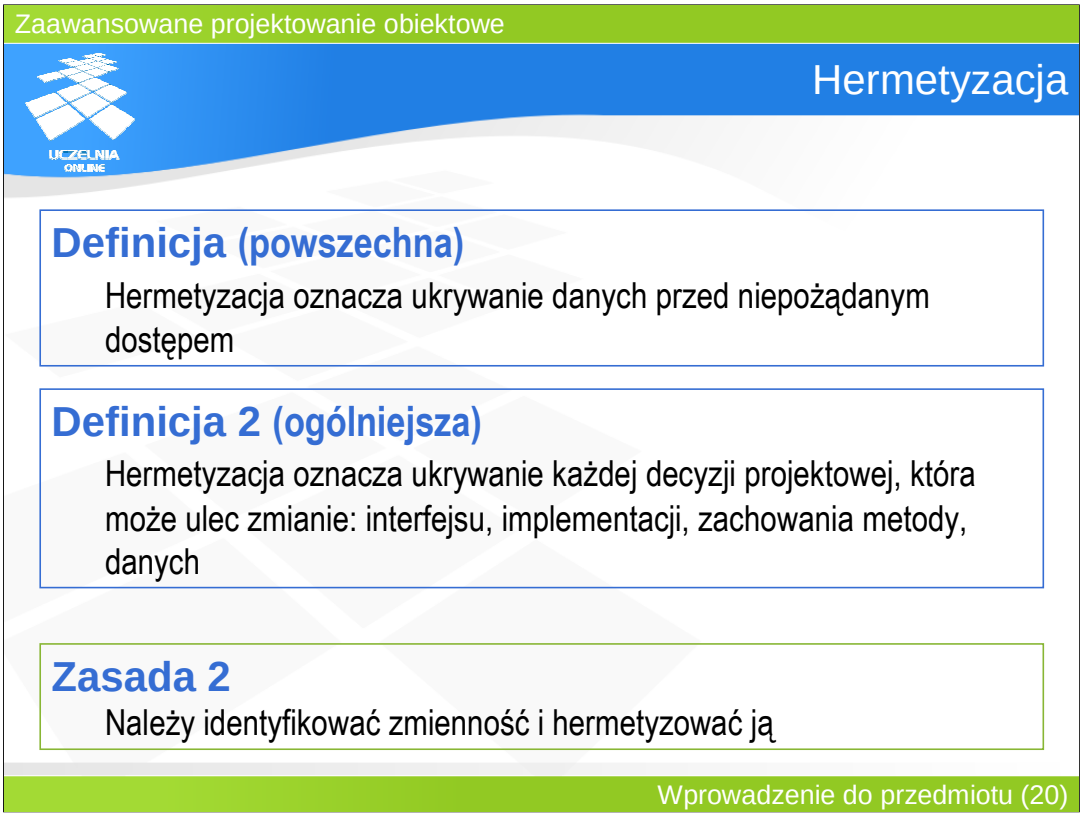
Z zestawienia wynika, że stosowanie interfejsów wyłącznie z punktu widzenia polimorfizmu jest lepszym rozwiązaniem, ponieważ tworzy słabsze powiązanie pomiędzy klasami. Natomiast dziedziczenie pozwala jednocześnie współdzielić kod, co z jednej strony powoduje silne powiązanie pomiędzy klasami, ale w niektórych sytuacjach jest uzasadnione.



Interfejsy, podobnie jak klasy, mogą być dziedziczone. Jak wcześniej powiedziano, dziedziczenie klas tworzy nowy podtyp i jednocześnie przekazuje do podklasy implementację. Natomiast dziedziczenie interfejsu powoduje jedynie utworzenie podtypu, który może być niezależnie zaimplementowany przez nową klasę.

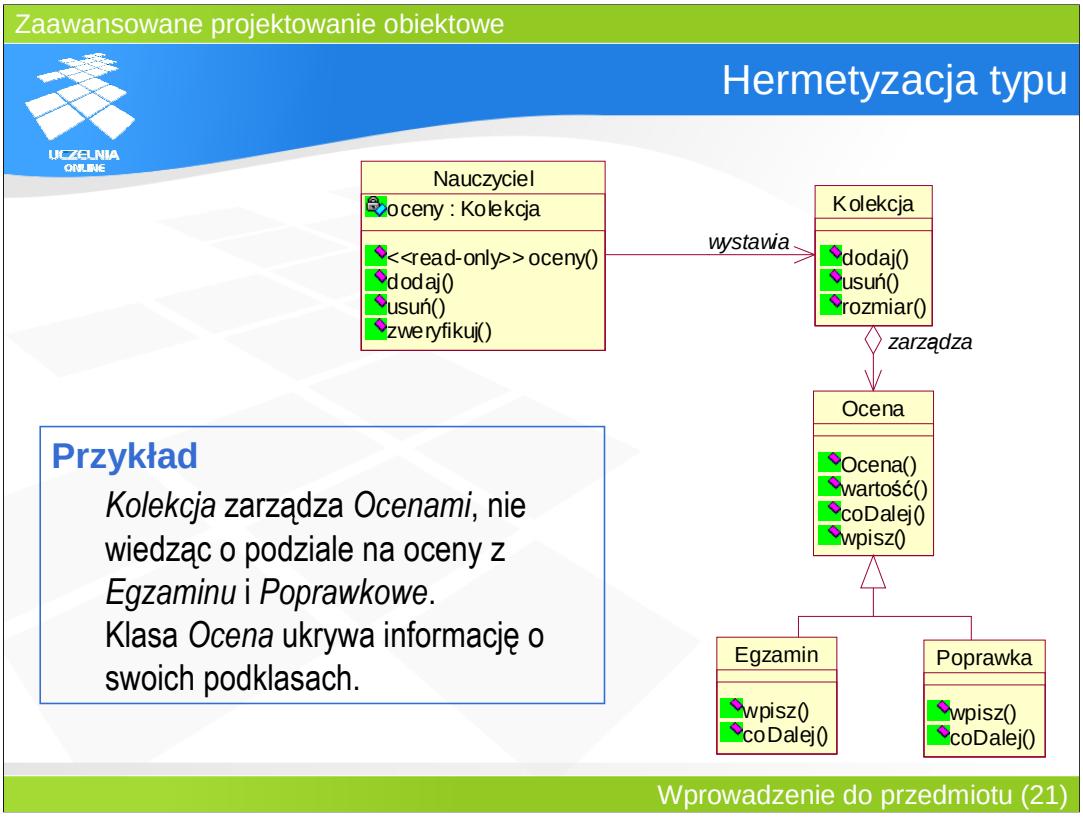
W przedstawionym przykładzie interfejs Zegar cyfrowy dziedziczy po interfejsie Czasomierz. Każdy z nich posiada własną implementację, odpowiednio w postaci klas Zegar ścienny i Zegarek ręczny. W efekcie klasa Zegarek ręczny może być stosowana w zastępstwie Zegara ściennego (ponieważ wypełnia jego kontrakt – potrafi mierzyć czas), co oznacza, że za pomocą interfejsów osiągnięto efekt identyczny jak w przypadku dziedziczenia. Jednak Zegarek ręczny nie dziedziczy implementacji Zegara ściennego, w związku z czym modyfikacja tej klasy nie ma żadnego wpływu na klasę Zegarek ręczny.

Przykład ten wskazuje, że użycie interfejsów powoduje powstanie słabszych zależności pomiędzy klasami niż w przypadku dziedziczenia.

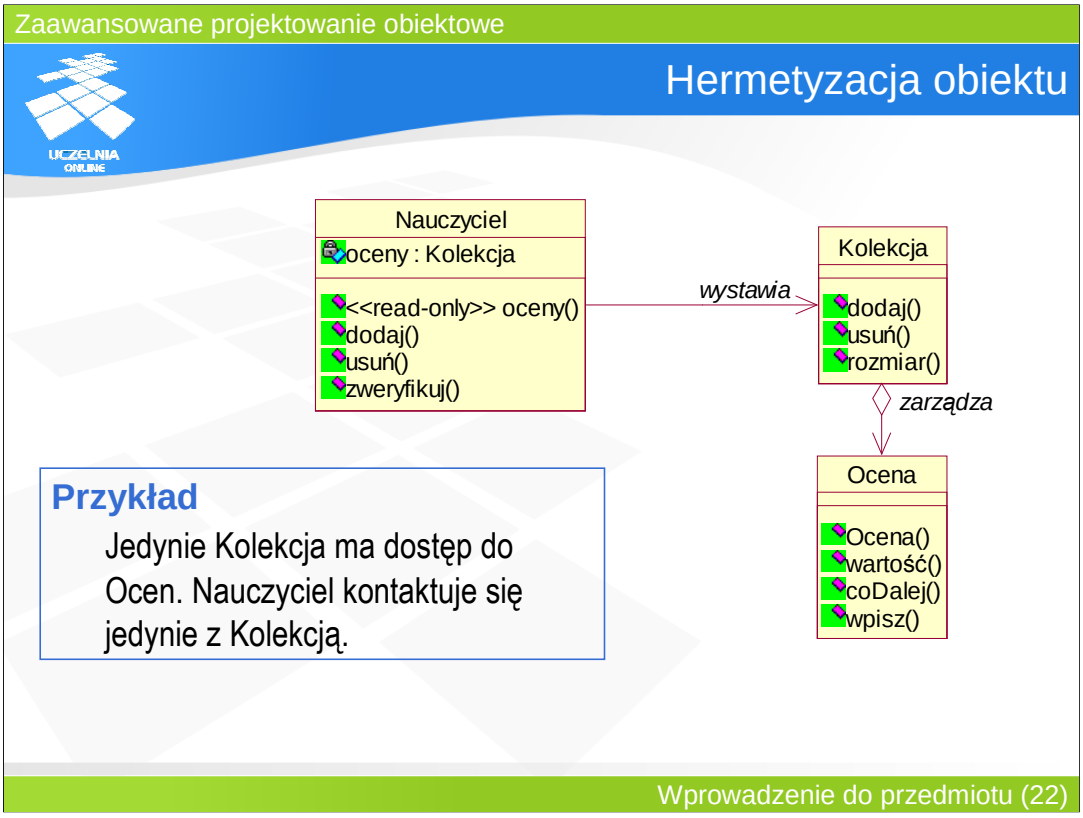


Hermetyzacja jest kolejnym pojęciem stanowiącym w powszechnym odbiorze charakterystyczną cechę obiektowości. Jest to prawda, ponieważ hermetyzacja w znacznym stopniu podnosi stopień abstrakcji. Jednak hermetyzacja często jest rozumiana w wąskim sensie, jako ukrywanie danych przed niepożądanym dostępem. Zgodnie z nią, wystarczy zadeklarować pola jako niepubliczne oraz stworzyć metody dostępu do nich, aby osiągnąć właściwy poziom ukrycia implementacji.

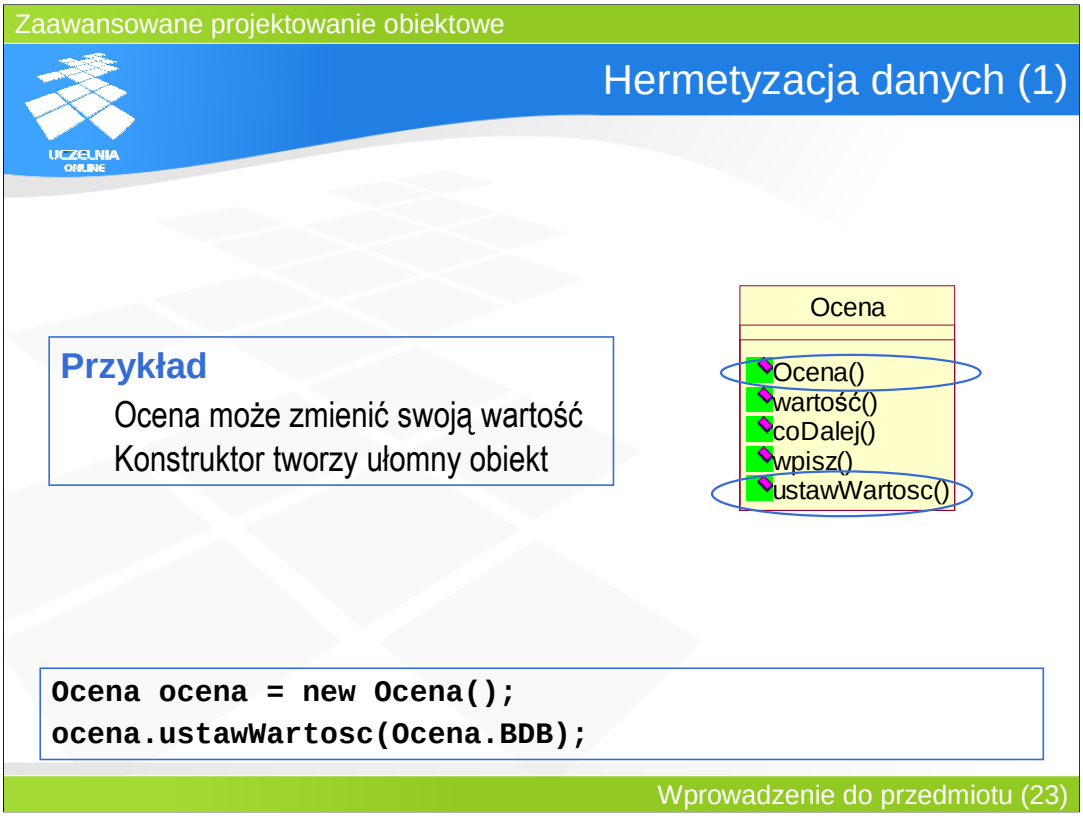
Takie rozwiązanie jest niewystarczające. Hermetyzację należy rozumieć znacznie szerzej: jako ukrywanie każdej decyzji projektowej, która może ulec zmianie. Dlatego projekt powinien identyfikować wszystkie obszary zmienności, dotyczące interfejsu, implementacji, zachowania metod czy danych, i hermetyzować je za pomocą dostępnych środków.



Slajd przedstawia hermetyzację typów osiągniętą za pomocą dziedziczenia. Klasa Kolekcja przechowuje Oceny, które dzielą się na oceny z egzaminu i oceny poprawkowe. Jednak z punktu widzenia Kolekcji ta informacja nie ma znaczenia, ponieważ klasa Ocena ukrywa przed nią swoje typy potomne.

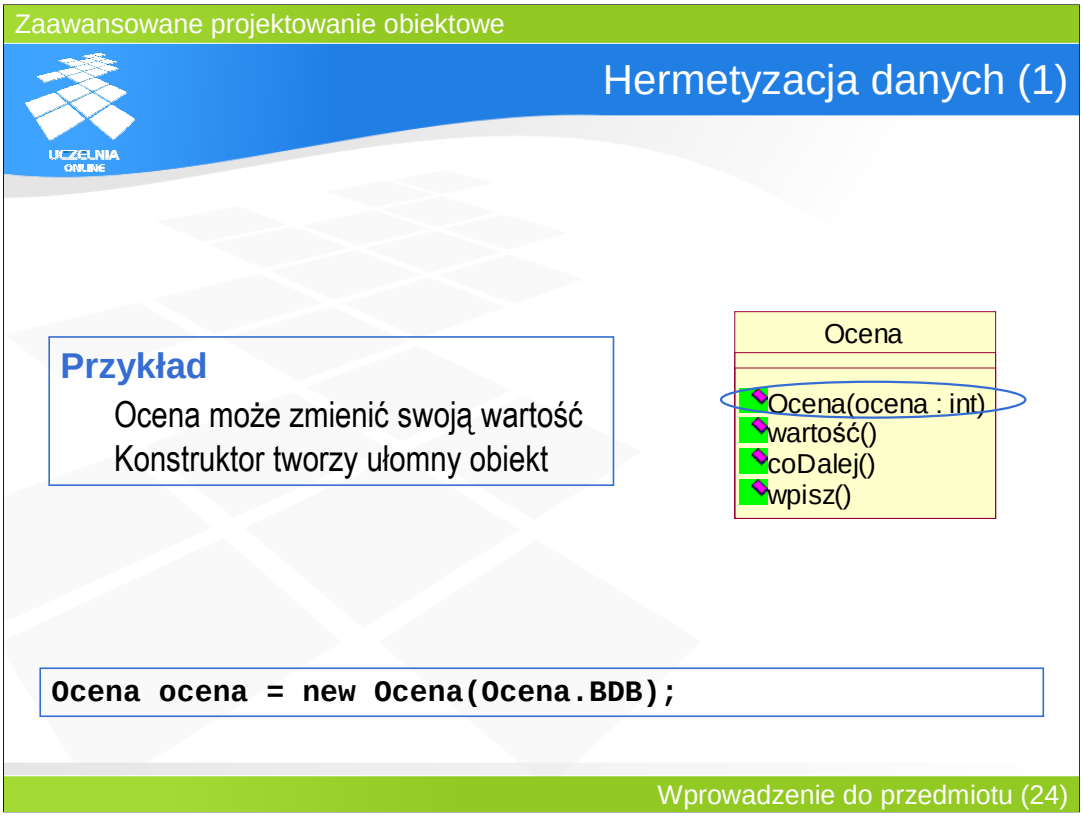


Ten sam przykład posłuży do omówienia hermetyzacji implementacji. Klasa Nauczyciel zarządza kolekcją ocen, jednak nie operuje bezpośrednio na pojedynczych ocenach. Dostępem do nich zarządza Kolekcja, natomiast Nauczyciel jest ich właścicielem w sensie logicznym, a nie fizycznym.

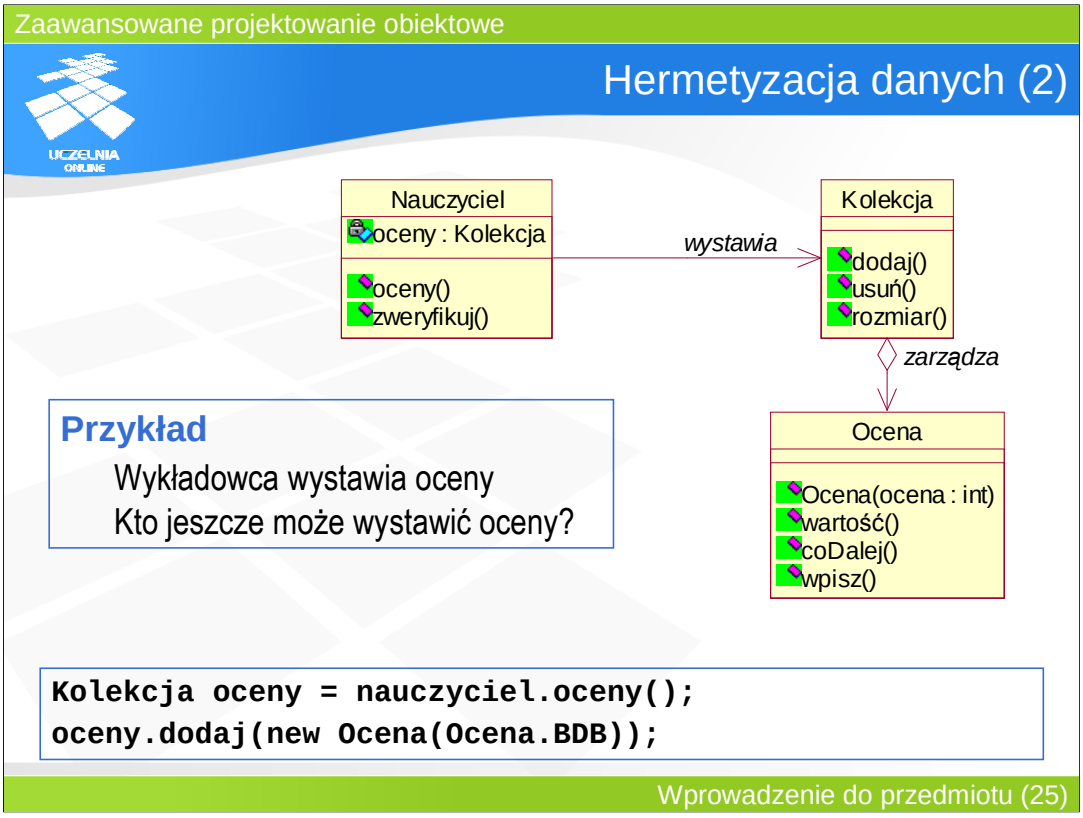


Hermetyzacja danych jest zwykle utożsamiana z hermetyzacją w ogólniejszym znaczeniu, co nie jest błędem, ale sporym uproszczeniem. Umieszczenie w klasie pary metod set/get dla każdego pola nie jest wystarczające, aby uznać ją za prawidłowo zamkniętą przed niepowołanym dostępem z zewnątrz.

W przedstawionym przykładzie klasa Ocena posiada bezparametrowy konstruktor oraz kilka metod, m.in. zmieniającą wartość oceny. Błąd polega właśnie na sposobie przechowywania wartości oceny. Wywołanie konstruktora spowoduje utworzenie obiektu Oceny bez wartości tej oceny, czyli obiektu ułomnego. Bezpośrednio po tym obiekt znajduje się w stanie nieustalonym, ponieważ nie można poznać oceny, jaką reprezentuje. Podobnie, udostępnienie metody zmieniającej wartość oceny może doprowadzić do jej swobodnej modyfikacji, bez wiedzy nauczyciela i/lub ucznia, co może naruszyć spójność systemu.



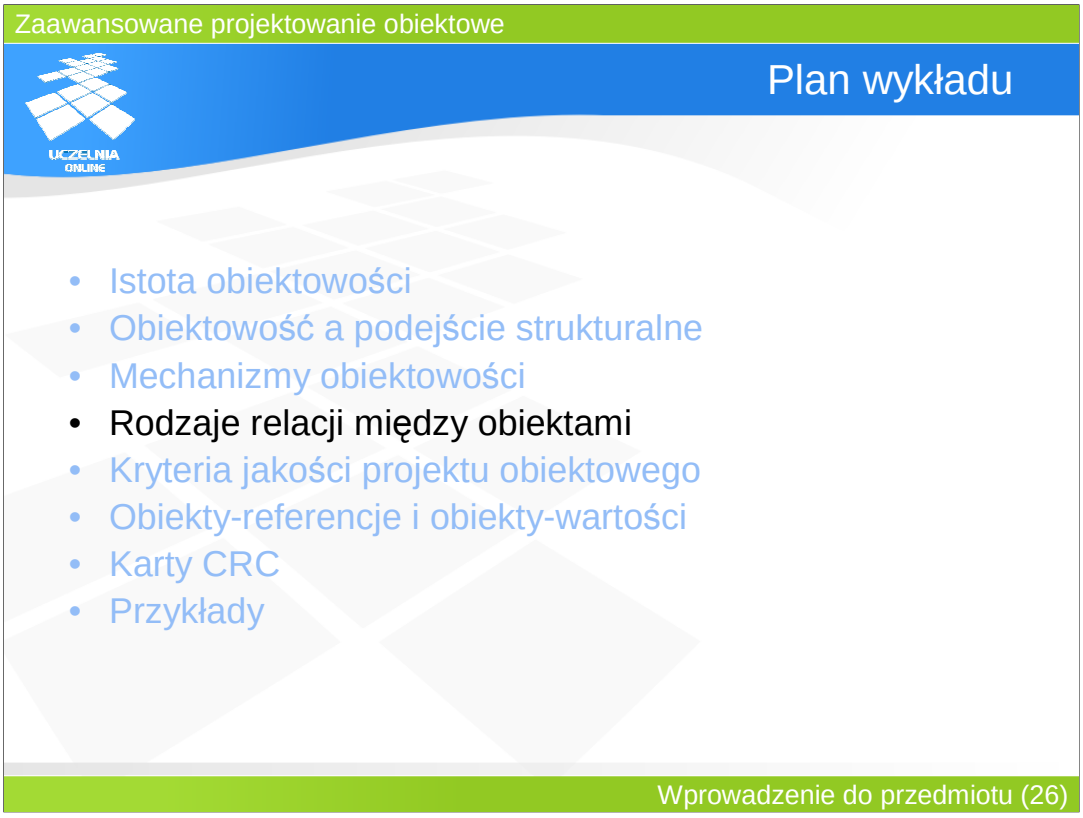
Prawidłowe rozwiązanie polega na stworzeniu sparametryzowanego konstruktora, który będzie przyjmował wartość oceny i tworzył obiekt posiadający sens od początku swojego istnienia.



Ciekawym problemem związanym z hermetyzacją jest dostęp do danych złożonych. Nauczyciel posiada kolekcję ocen, które wystawił. Kolekcja ta jest udostępniona przez niego w postaci metody, która zwraca do niej referencję. Zaimplementowany w ten sposób dostęp do kolekcji pozwala na jej niekontrolowaną modyfikację: dodawanie, usuwanie i zmianę zawartości elementów poza wiedzą właściciela kolekcji, czyli Nauczyciela.

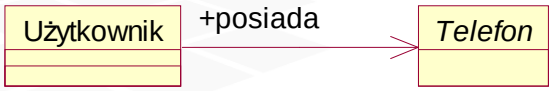
Zatem prawidłowe rozwiązanie powinno przebiegać w następujących krokach:

1. ukrycie metody *oceny(),* zwracającej listę ocen, lub przynajmniej zmiana jej zachowania, tak aby zmiany w liście nie powodowały zmian w obiekcie Nauczyciel.
2. utworzenie metod operujących na kolekcji w klasie jej właściciela, czyli Nauczyciela. W ten sposób wszelkie zmiany będą kontrolowane.



Kolejna część wykładu będzie poświęcona na przypomnienie podstawowych typów relacji, jakie mogą wystąpić pomiędzy obiektami.

Rodzaje relacji: asocjacja



- *Komunikator* należy do *Użytkownika*
- *Komunikator* może zmienić *Użytkownika*, *Użytkownik* może zmienić *Komunikator*
- *Użytkownik* zna swój *Komunikator*, *Komunikator* nie wie, kto jest jego właścicielem
- Istnienie *Użytkownika* nie ma wpływu na istnienie *Komunikatora* i vice wersa

Najczęściej spotykanym rodzajem związku dwóch klas jest asocjacja. Polega ona na umieszczeniu w obiekcie po jednej stronie relacji (lub obu, w przypadku asocjacji dwukierunkowych) referencji do drugiego obiektu, dzięki czemu obiekt ten może odwołać się do niego i jego metod.

Asocjacja reprezentuje związek, w którym oba obiekty istnieją niezależne od siebie, tzn. istnienie jednego nie jest warunkiem istnienia drugiego. Usunięcie związku pomiędzy obiektami nie wpływa na ich sposób funkcjonowania. Ponadto obiekty nie są związane ze sobą na stałe i mogą zostać zmienione na inne (na przykładzie Użytkownik może zmienić Telefon, a Telefon może zmienić właściciela).

Asocjacje jednokierunkowe wprowadzają asymetrię, wyróżniając klasę posiadającą referencję do obiektu podrzędnego. Klasa po drugiej stronie nie posiada żadnej wiedzy o obiekcie nadrzędnym. W przypadku asocjacji dwukierunkowych obiekty pozostają we względnej równowadze, posiadając referencje do siebie nawzajem.

Rodzaje relacji: agregacja

- *Katalog* zawiera *Książki*
- *Książka* może należeć do wielu *Katalogów* jednocześnie
- Istnienie *Katalogu* nie ma wpływu na istnienie *Książki* i vice versa

Silniejszym od asocjacji rodzajem relacji pomiędzy obiektami jest agregacja, nazywana również relacją zawierania. W relacji tej silniejszą stroną jest obiekt pełniący rolę całości, z którą związane są pozostałe obiekty uczestniczące w relacji – jej elementy. Obiekt ten zarządza swoimi elementami, dodając je, usuwając i wyszukując w odpowiedzi na żądania klientów.

Jednak relacja ta nie łączy całości i części w sposób wyłączny: elementy mogą należeć jednocześnie do wielu obiektów-całości (np. Książka może należeć do wielu Kategorii jednocześnie), a ponadto obie strony agregacji mogą istnieć niezależnie od siebie. Ta cecha odróżnia agregację od jej silniejszej postaci – kompozycji.

Rodzaje relacji: kompozycja

- *Książka* składa się z *Rozdziałów*, *Rozdział* jest częścią *Książki*
- *Rozdział* może należeć tylko do jednej *Książki*
- Istnienie *Książki* decyduje o istnieniu *Rozdziału* i vice versa

Kompozycja jest rodzajem agregacji, w której zależność elementów od obiektu-całości jest silniejsza niż w przypadku agregacji i obejmuje niemal każdy aspekt jego istnienia.

W przedstawionym przykładzie Książka składa się z Rozdziałów, które stanowią jej nieodłączną część: nie mogą one istnieć niezależnie od Książki i są od niej zależne. Podobnie Książka pozbawiona Rozdziałów nie jest pełnoprawnym obiektem.

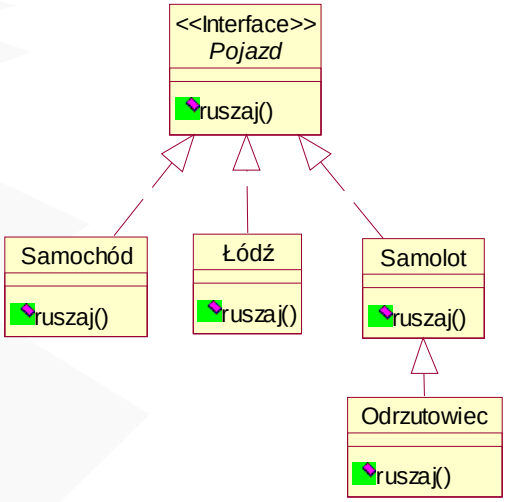
Różnica w stosunku do agregacji polega także na liczbie obiektów-całości, z którymi może być związany element: w przypadku kompozycji elementy należą tylko do jednej całości.

Relacja kompozycji nie ma ściśle określonej krotności: w większości przypadków wynosi ona jeden do wielu, ale zdarza się, że obiektowi-całości odpowiada tylko jeden element składowy. Wówczas relacja ta oznacza, że jest on niezbędny do istnienia całości.

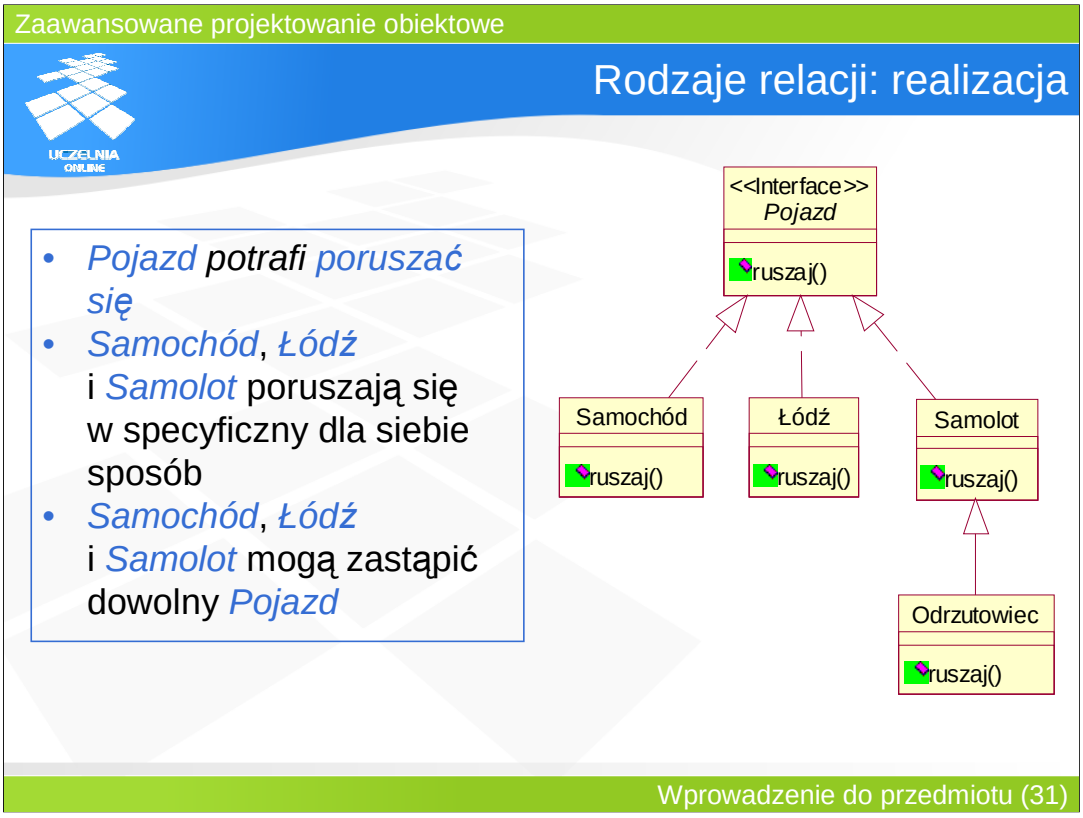
W programowaniu obiektowym relacja kompozycji jest zwykle implementowana w postaci referencji inicjowanej poprzez przekazanie elementów w konstruktorze obiektu całości.

Rodzaje relacji: dziedziczenie

- *Odrzutowiec jest rodzajem Samolotu*
- *Odrzutowiec* może zastąpić *Samolot*
- *Odrzutowiec* posiada niejawną instancję *Samolotu*
- związek między *Odrzutowcem* i *Samolotem* jest nierozerwalny



Dziedziczenie jest relacją wiążącą obiekty w nieco inny sposób i powodującą inne efekty niż relacje przedstawione na poprzednich slajdach. Oznacza ono bowiem nie związanie ze sobą dwóch niezależnych klas, ale utworzenie nowej klasy na bazie już istniejącej. Zatem związek między nadklasą i podklasą jest znacznie silniejszy niż w poprzednich przypadkach. Mimo, że dziedziczenie jest uznawane za typowy mechanizm obiektowy, to jednak w wielu przypadkach tworzy zbyt mocną zależność pomiędzy klasami i dlatego powinno być zastępowane stosowaniem interfejsów



Ostatnim rodzajem relacji jest realizacja (implementacja) interfejsu. Jak wielokrotnie już wspomniano, stanowi ona alternatywę dla dziedziczenia klas, która pozwala na współdzielenie typu bez współdzielenia kodu.

W przykładzie przedstawionym na slajdzie Samochód, Łódź i Samolot są tego samego typu, ponieważ implementują interfejs Pojazd i definiują metodę *ruszaj()*. Każdy z tych obiektów może zastąpić interfejs Pojazd, i w tym sensie są one równoważne. Żadna z nich nie jest jednak jednostką nadrzędną wobec innych i nie narzuca im sposobu wykonania kontraktu wynikającego z typu.

Dziedziczenie a kompozycja

Dziedziczenie	*Kompozycja*
• relacja utrwalona w czasie kompilacji • przekazuje podklasie typ i implementację • silnie wiąże nadklasę i podklasę (hermetyzacja!)	• relacja zmienna w trakcie wykonywania programu • nie zapewnia istnienia obiektu • wiąże obiekty jedynie poprzez typ

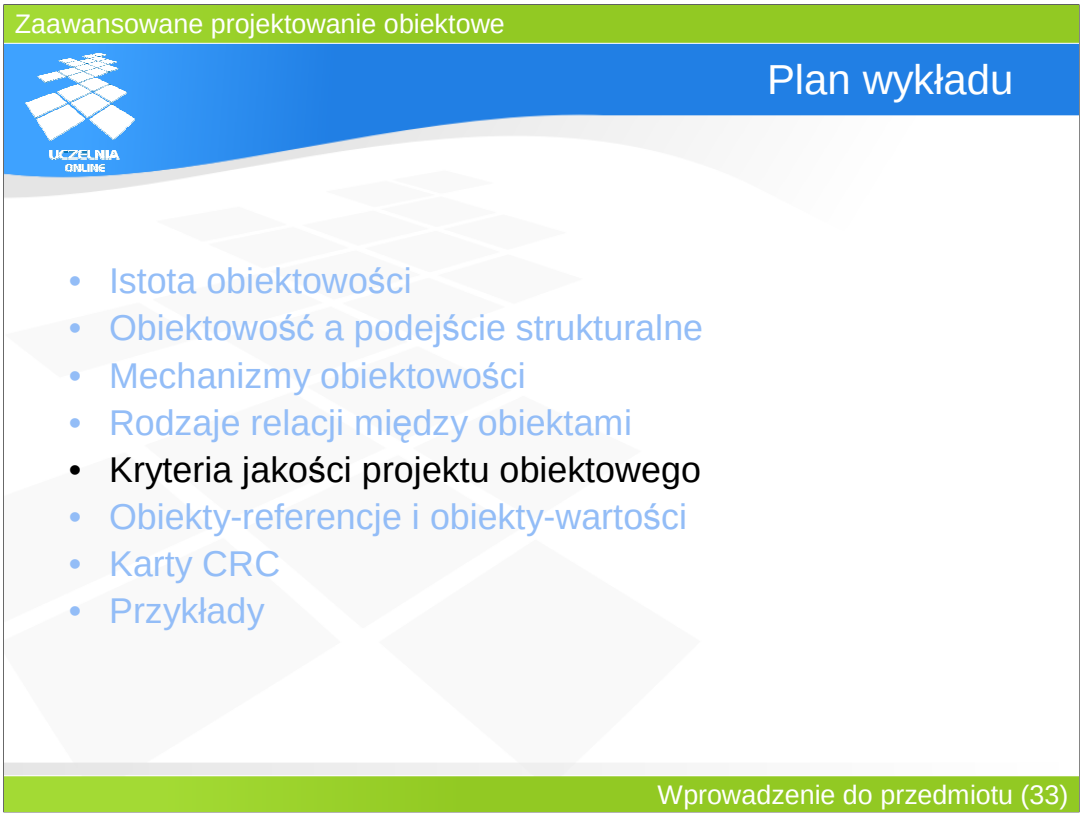
Zasada 3

Należy preferować kompozycję ponad dziedziczenie

Porównanie kompozycji i dziedziczenia – relacji na pierwszy rzut oka nieporównywalnych – staje się uzasadnione, gdy poznamy fizyczny sposób implementacji relacji dziedziczenia. Utworzenie obiektu podklasy zawsze powoduje niejawne utworzenie instancji obiektu nadklasy (widać to m.in. w sposobie wywoływania konstruktorów nadklas) i zapamiętanie referencji do niej w zmiennej dostępnej pod nazwą *super*. Oznacza to, że kompozycja i dziedziczenie – mimo zupełnie innych zastosowań – są w praktyce implementowane w identyczny sposób. Którą z relacji zatem należy preferować?

Dziedziczenie jest relacją ustaloną w momencie kompilacji, która silnie wiąże ze sobą nadklasę i podklasę. Związek ten może nawet naruszać hermetyzację klas, ponieważ podklasa dziedziczy całe zachowanie nadklasy. Z kolei kompozycja może być modyfikowana w trakcie wykonywania programu, ponadto wiąże obiekty znacznie słabiej, poprzez typ.

Zatem zgodnie z ogólną zasadą (od której jednak istnieją wyjątki), należy preferować kompozycję od dziedziczenia.



Dotychczas była mowa o różnych zaleceniach dotyczących stosowania wybranych mechanizmów obiektowych. Argumenty dotyczyły jakości projektu obiektowego, jednak dotąd nie zdefiniowaliśmy żadnego kryterium oceniającego tę wielkość

W tej części wykładu krótko przedstawione zostaną dwa kryteria: spójność i powiązania, które okazały się skorelowane z zewnętrznymi atrybutami jakości programu, takimi jak pielęgnowalność, testowalność, gęstość błędów etc.

Spójność obiektu

Spójność obiektu

- wielkość porządkowa (wysoka – niska)
- opis współpracy elementów obiektu
- stopień powiązania metod i pól obiektu przy wypełnienia nałożonej na niego odpowiedzialności

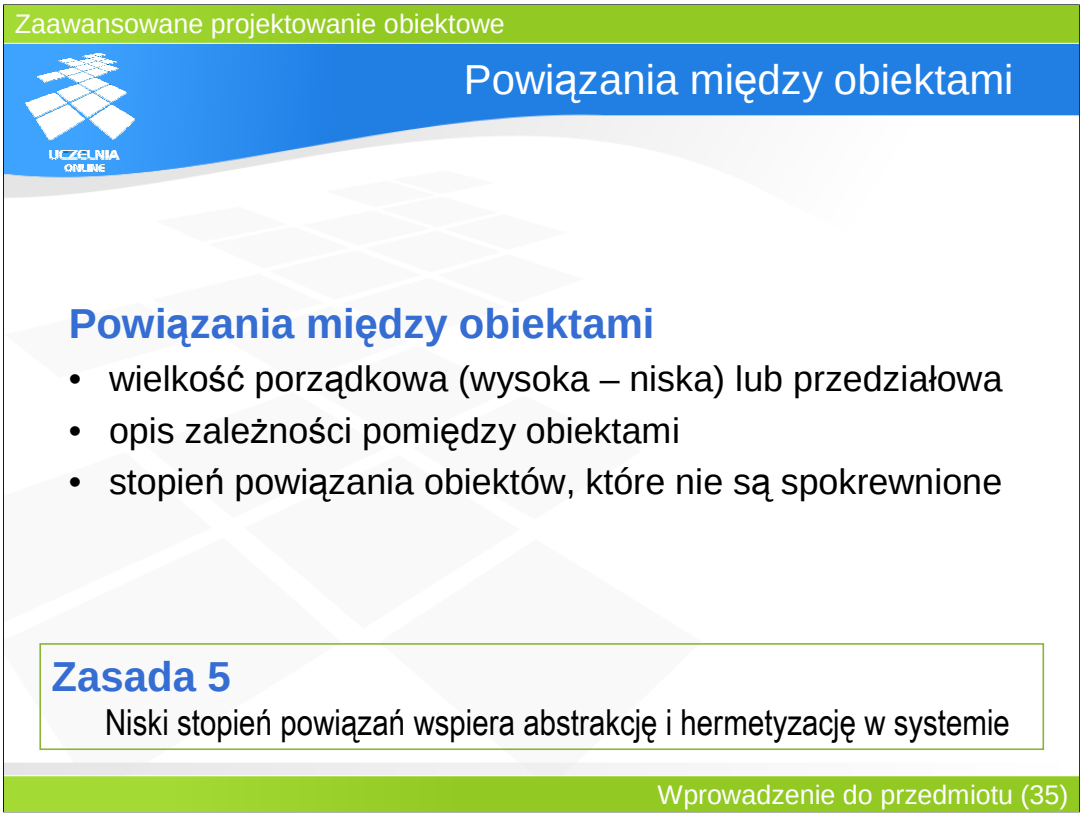
Zasada 4

Prawidłowo zaprojektowana klasa jest spójna

Spójność obiektu jest miarą współpracy metod i pól klasy w celu wypełnienia nałożonej na nią odpowiedzialności. Klasa jest spójna, jeżeli jej metody odwołują się w większości do pól tej samej klasy i innych jej metod. Brak spójności występuje, gdy metody klasy odwołują się do zewnętrznych obiektów lub wykonują niezwiązane ze sobą funkcje.

Wysoka spójność obiektu jest wartością pożądaną, ponieważ jest dodatnio skorelowana z wieloma zewnętrznymi atrybutami jakości: pielęgnowalnością, czytelnością i testowalnością. Niska spójność prowadzi do ograniczenia ponownego użycia klasy oraz jej czytelności, a także zwiększenia pracochłonności testowania.

Dlatego prawidłowo zaprojektowana klasa powinna charakteryzować się wysoką spójnością.



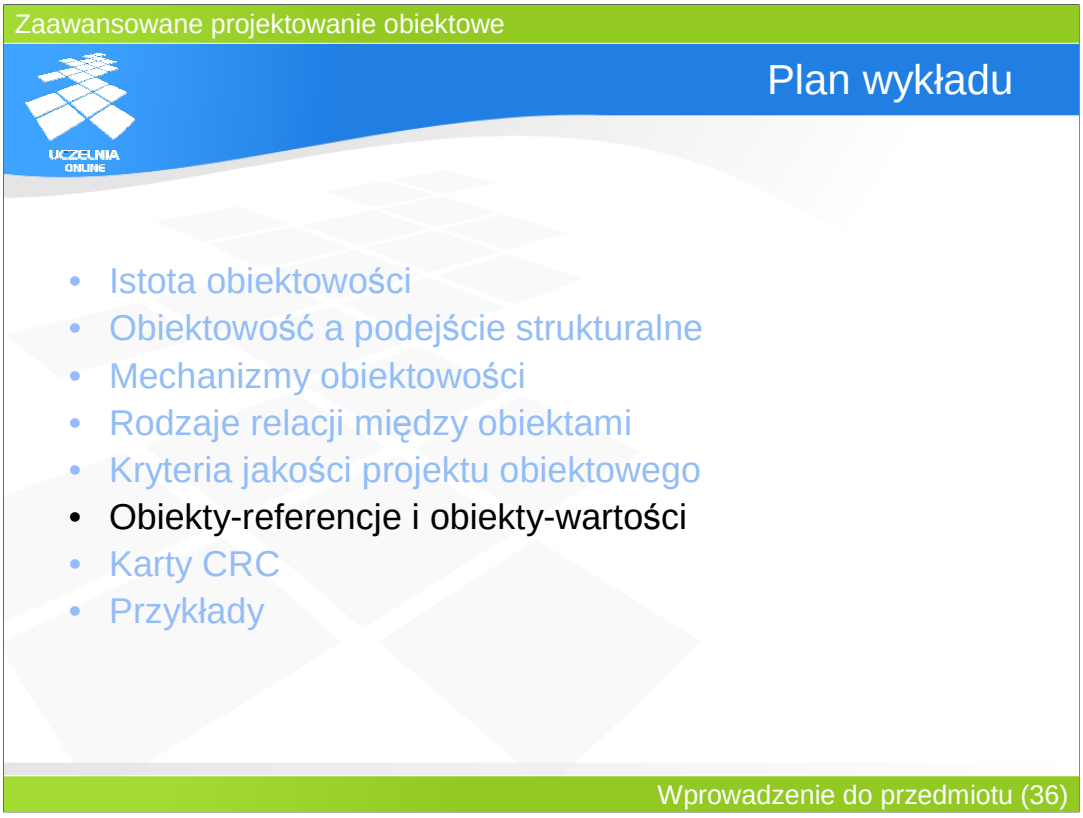
Drugim kryterium jakości projektu obiektowego jest stopień powiązania klas. Powiązanie pomiędzy dwiema niespokrewnionymi klasami jest skierowaną relacją, która odzwierciedla stopień i rodzaj zależności pomiędzy nimi. Klasa A jest powiązana z klasą B, jeżeli musi posiadać o niej jakąś wiedzę: posiadać składową tego typu, implementować typ B, być podklasą B, definiować metodę, która zawiera taką klasę w swojej sygnaturze (jako parametr, wartość zwracana lub deklarowany wyjątek).

Podobnie jak w przypadku spójności, powiązania wyraża się w kategoriach powiązań luźnych (o niskim stopniu) i mocnych (o dużej liczbie zależności).

Słabe powiązanie oznacza, że analizowana klasa w niewielkim stopniu zależy od zmian w innych klasach. Mają na to wpływ dwa czynniki: liczba powiązań oraz ich rodzaj. Siła powiązania jest odwrotnie proporcjonalna do abstrakcyjności klasy zależnej: zależność od stabilnego interfejsu jest mniejsza od zależności od szczegółowej klasy implementacyjnej.

Wysoki stopień powiązań ma negatywny wpływ na wiele zewnętrznych czynników jakości programu, m.in. jego pielęgnowalność, elastyczność i stopień abstrakcji. Dlatego projekt obiektowy powinien minimalizować liczbę powiązań między klasami i interfejsami, i jednocześnie preferować zależności od stabilnych interfejsów.

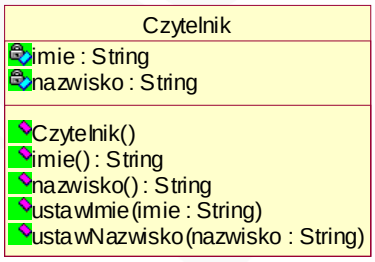
Warto zwrócić uwagę na zależność pomiędzy spójnością a powiązaniami: w większości wypadków wysoka spójność oznacza także niski stopień powiązań, a brak spójności stopień ten zwiększa.



Obiekty, wśród wielu innych klasyfikacji, dzielą się ze względu na sposób odwoływania się do nich na dwie kategorie: obiekty-wartości i obiekty-referencje. W tej części wykładu zostanie krótko przedstawione znaczenie tego podziału, różnice pomiędzy obiema kategoriami oraz zastosowania obu rodzajów obiektów

Obiekty-referencje

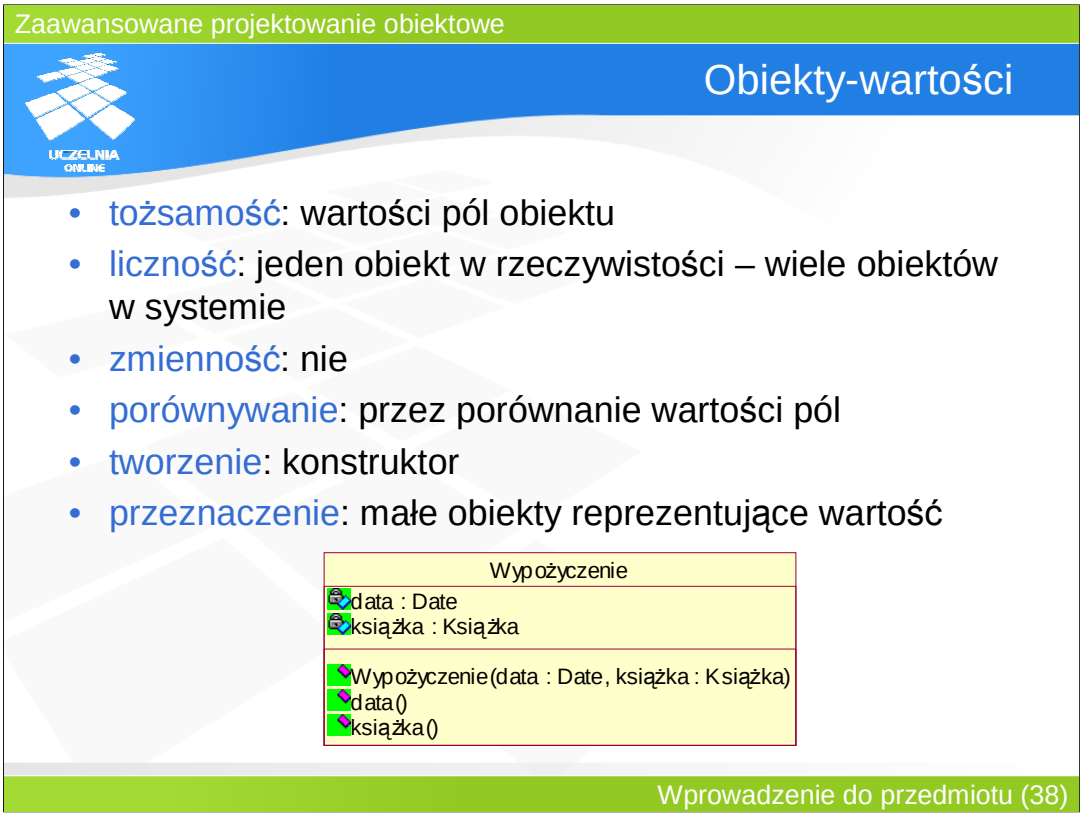
- tożsamość: referencja do obiektu
- liczność: jeden obiekt w rzeczywistości = jeden obiekt w systemie
- zmienność: tak
- porównywanie: przez porównanie referencji
- tworzenie: dedykowana klasa-fabryka
- przeznaczenie: duże złożone obiekty



Obiekty-referencje zwykle reprezentują duże, unikatowe jednostki świata rzeczywistego, które są opisywane za pomocą wielu atrybutów i metod. Obiekty takie trudno utworzyć, a następnie synchronizować z innymi obiektami, dlatego jeden obiekt w rzeczywistości posiada jeden odpowiednik w systemie, do którego wiele obiektów może odwoływać się naraz. Takie rozwiązanie ma na celu uniknięcie problemu synchronizacji wielu kopii tego samego obiektu. Aby zapewnić unikatowość obiektów-referencji, często stosowanym sposobem ich tworzenia jest specjalizowana fabryka, która przechowuje i zarządza referencjami do utworzonych wcześniej obiektów.

Obiekty takie, dzięki swojej unikatowości, mogą być porównywane przez referencję, ponieważ ta sama referencja może wskazywać wyłącznie na ten sam obiekt. Referencja pełni w tym przypadku także rolę identyfikatora każdego obiektu.

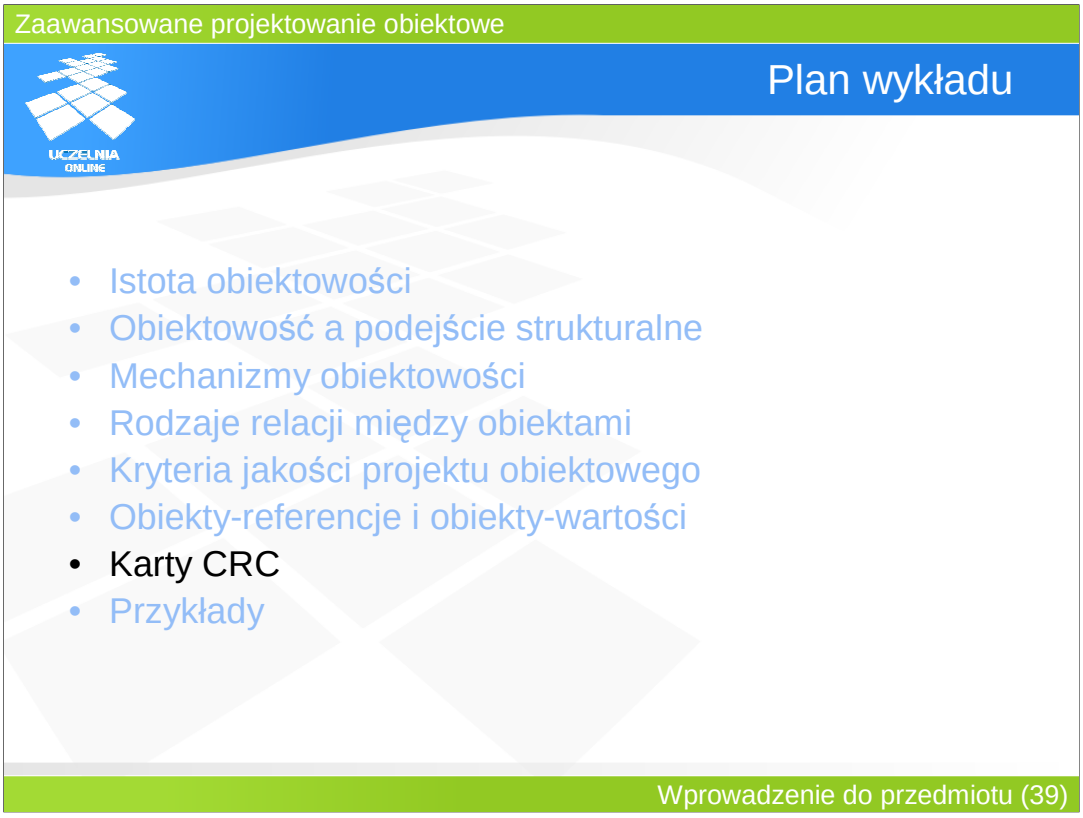
Przykładem obiektu-referencji jest Czytelnik, który przechowuje pola reprezentujące imię i nazwisko, oraz posiada metody zwracające i zmieniające wartości tych pól. Czytelnik o określonym imieniu i nazwisku istnieje w systemie tylko jako jeden obiekt, dlatego jeżeli kilku klientów odwołuje się do tego samego Czytelnika, wówczas posiadane przez nich referencje do tego obiektu są identyczne.



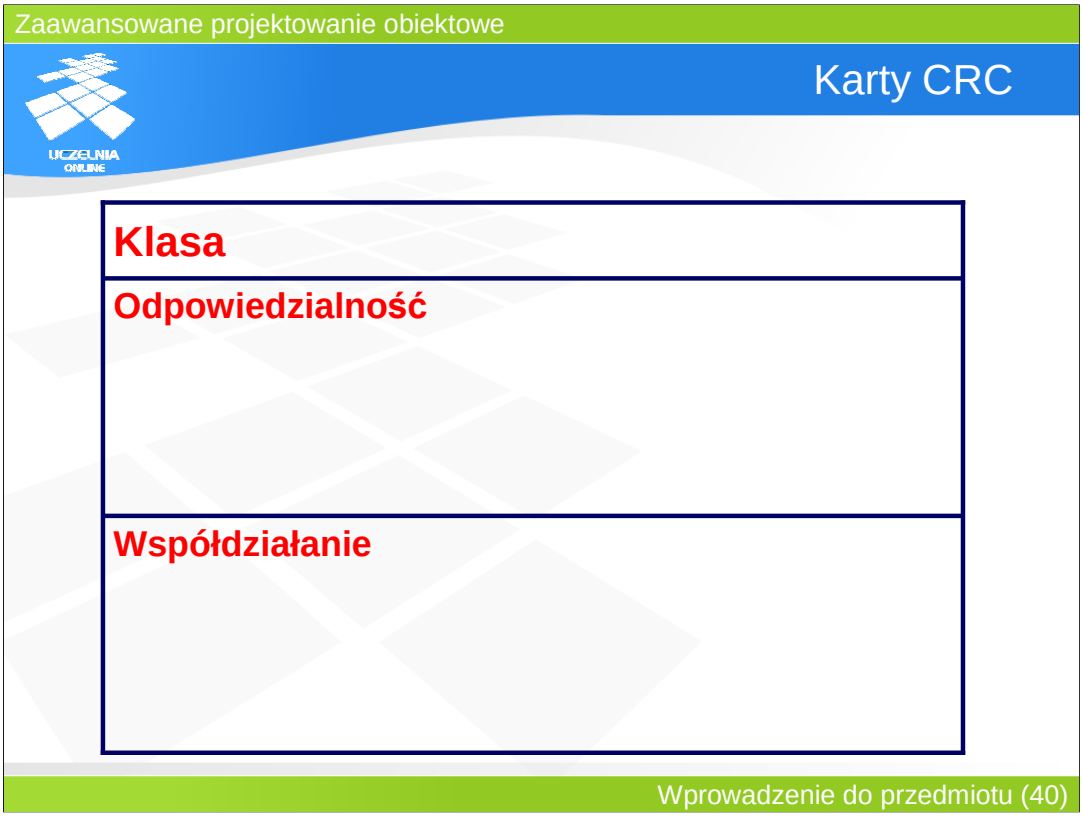
Przeciwieństwem obiektów-referencji są obiekty-wartości. Reprezentują one niewielkie jednostki o małym zakresie odpowiedzialności, zwykle ograniczonym do zapamiętania chwilowego stanu innego obiektu lub określonej wartości. Ich koszt utworzenia i przechowywania jest niewielki, dlatego ten sam obiekt istniejący w rzeczywistości może posiadać dowolną liczbę odpowiedników w systemie obiektowym. Ceną za możliwość istnienia wielu kopii tego samego obiektów jest ich niezmienność (ang. *immutability*) – ich stan jest określany w momencie ich utworzenia i nie ulega zmianie aż do ich usunięcia. Można sobie wyobrazić, jakie konsekwencje miałoby dopuszczenie modyfikacji obiektów przechowujących wartości pieniężne: bez zmiany referencji obiekt zmieniłby swoją wartość z np. 100 PLN do 1000 PLN.

Ponieważ referencja nie jest wystarczającym wyznacznikiem tożsamości obiektu, dlatego porównywanie odbywa się przez porównanie odpowiadających sobie pól obiektu. W języku Java rolę tę pełni metoda *equals()*, która we wszystkich klasach potomnych klasy *java.lang.Object* powinna być implementowana właśnie w oparciu o porównanie wartości poszczególnych pól.

Przykładem obiektu-wartości może być obiekt reprezentujący fakt wypożyczenia pewnej Książki w określonym dniu. Obiekt ten jest tworzony poprzez bezpośrednie wywołanie konstruktora i przekazania wszystkich danych jako argumentów. Metody klasy służą jedynie do odczytu wartości poszczególnych pól. Fakt wypożyczenia może być reprezentowany przez dowolną liczbę obiektów o tych samych wartościach, ponieważ nie mogą one ulec zmianie.



W tej części wykładu krótko omówiona zostanie technika projektowania w oparciu o karty CRC. Metoda ta jest szczególnie przydatna do definiowania klas w oparciu o ich odpowiedzialność (a nie ich atrybuty i metody).

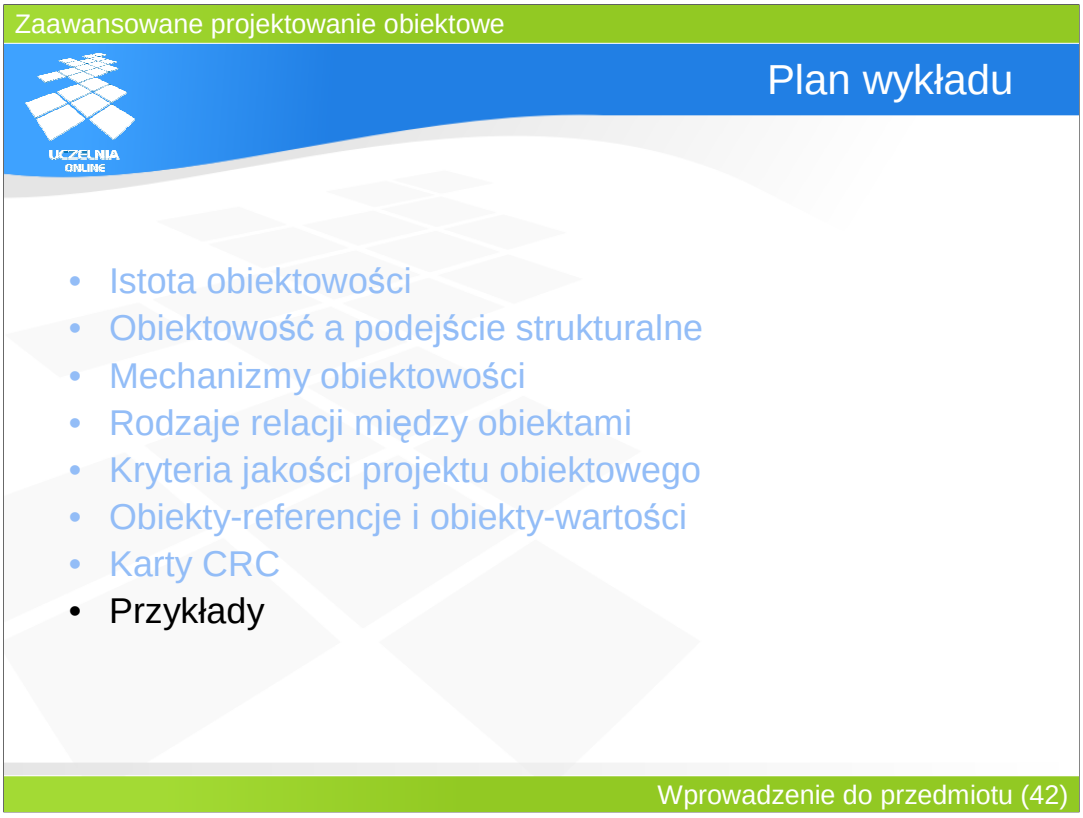


Metoda projektowania z użyciem kart CRC (ang. *class-responsibility-collaboration*) została zaproponowana przez jednego z propagatorów tzw. zwinnych metodyk – Warda Cunninghama. Jest ona szczególnie przydatna na etapie definiowania odpowiedzialności poszczególnych klas i określania sposobu współpracy. Co ważne, pomija ona całkowicie wewnętrzną strukturę klas, skupiając się wyłącznie na ich zachowaniu i odpowiedzialności. Dzięki temu złożoność procesu projektowania schematu klas jest ograniczona do minimum.

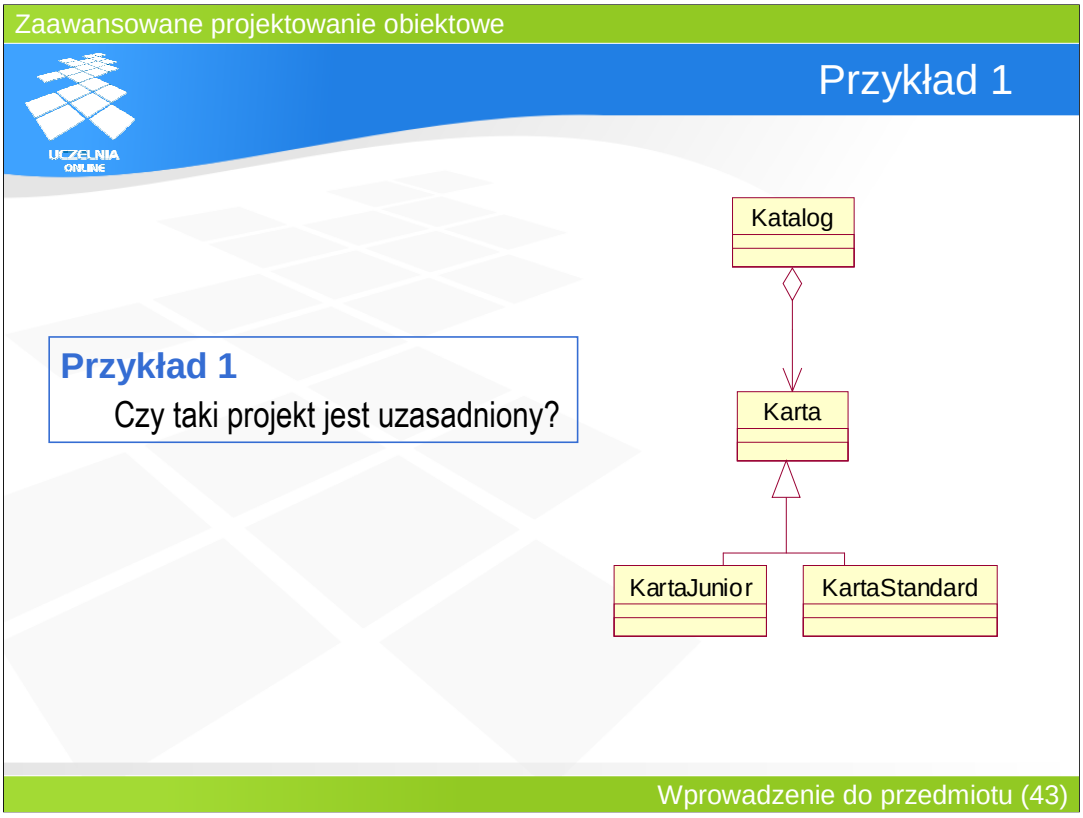
Karty CRC są kartkami papieru podzielonymi na trzy części, opisującymi następujące własności klasy:

- nazwę klasy (ang. *Class*), intuicyjnie opisującą jej odpowiedzialność,
- odpowiedzialność (ang. *Responsibility*), zawierającą dłuższy opis zadań, jakie będą powierzone klasie, oraz
- współdziałanie (ang. *Collaboration*) , przedstawiające interakcje obiektu z innymi klasami.

Na slajdzie przedstawiono przykład karty CRC dla klasy Nauczyciel. Odpowiedzialność tej klasy to wystawianie ocen oraz ich udostępnianie uczniom. Klasa współpracuje z klasą Kolekcja i klasą Uczeń.



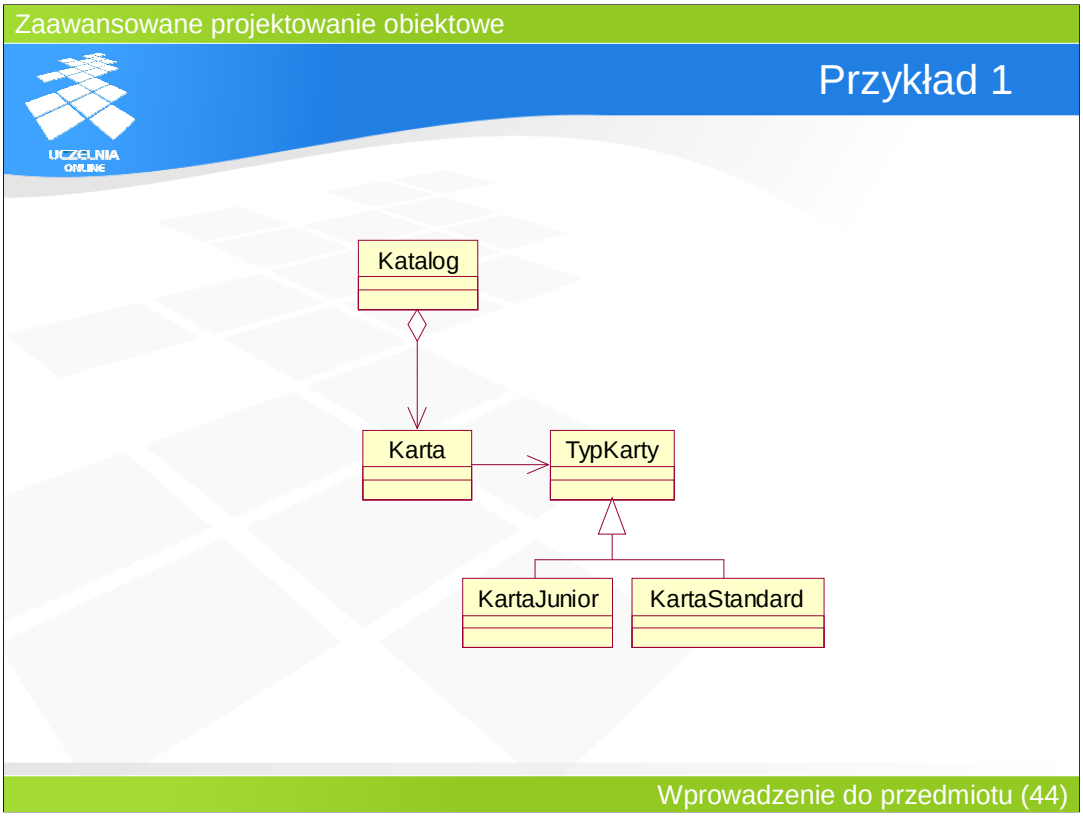
W ramach podsumowania wykładu zostaną omówione przykłady kilku rozwiązań obiektowych.



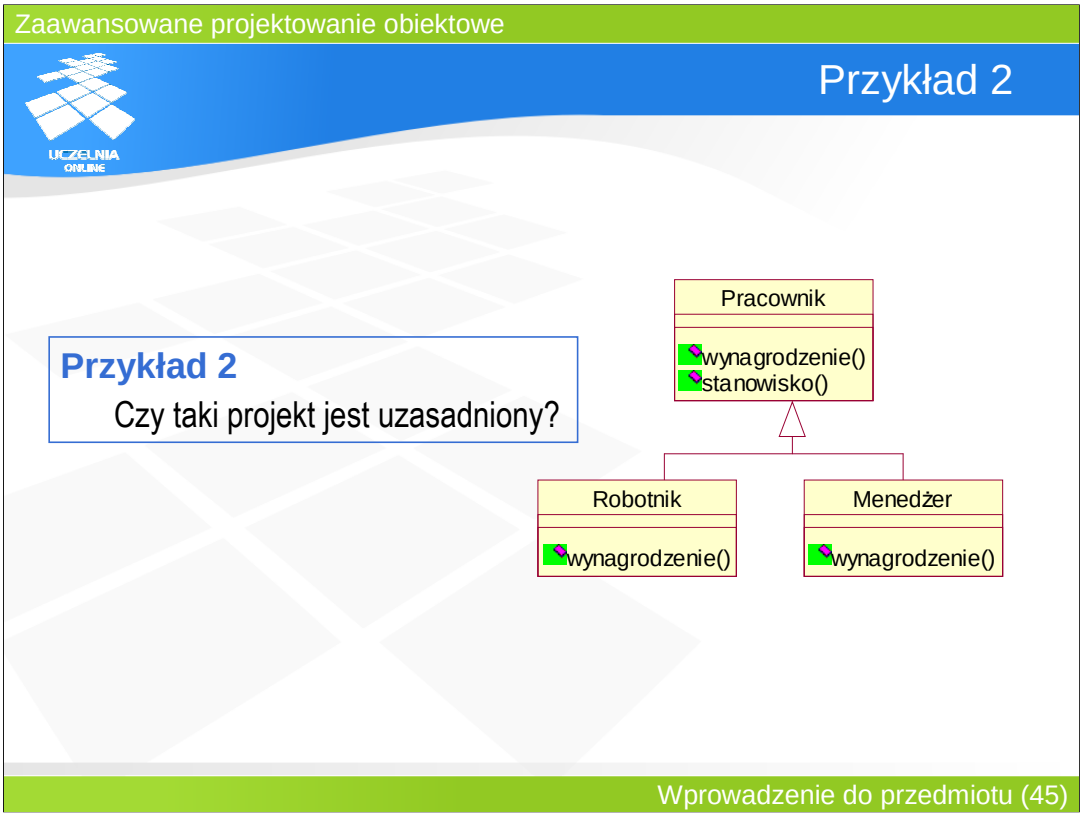
Pierwszy przykład dotyczy struktury fragmentu projektu katalogu, w którym są przechowywane karty czytelników. Katalog składa się z Kart dzielących się na Karty Junior i Karty Standard.

Czy taki schemat relacji między klasami jest uzasadniony? Proszę rozważyć następujące kwestie:

- dodawanie i usuwanie kart z katalogu
- dodanie nowego typu karty
- zmiana typu karty

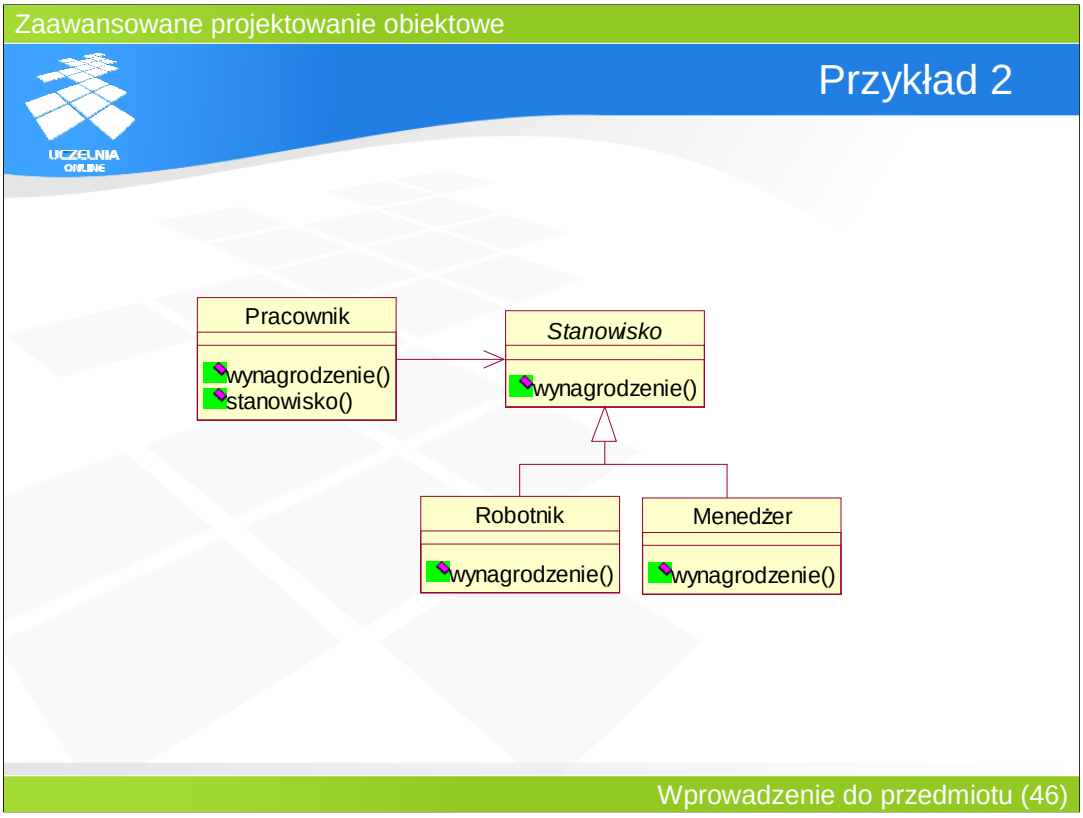


Rozwiązaniem jest podzielenie klasy Karta i wyłączenie z niej części odpowiedzialnej za reprezentację TypuKarty. W ten sposób możliwa jest zmiana typu bez konieczności tworzenia nowego obiektu jednej z podklas klasy Karta.

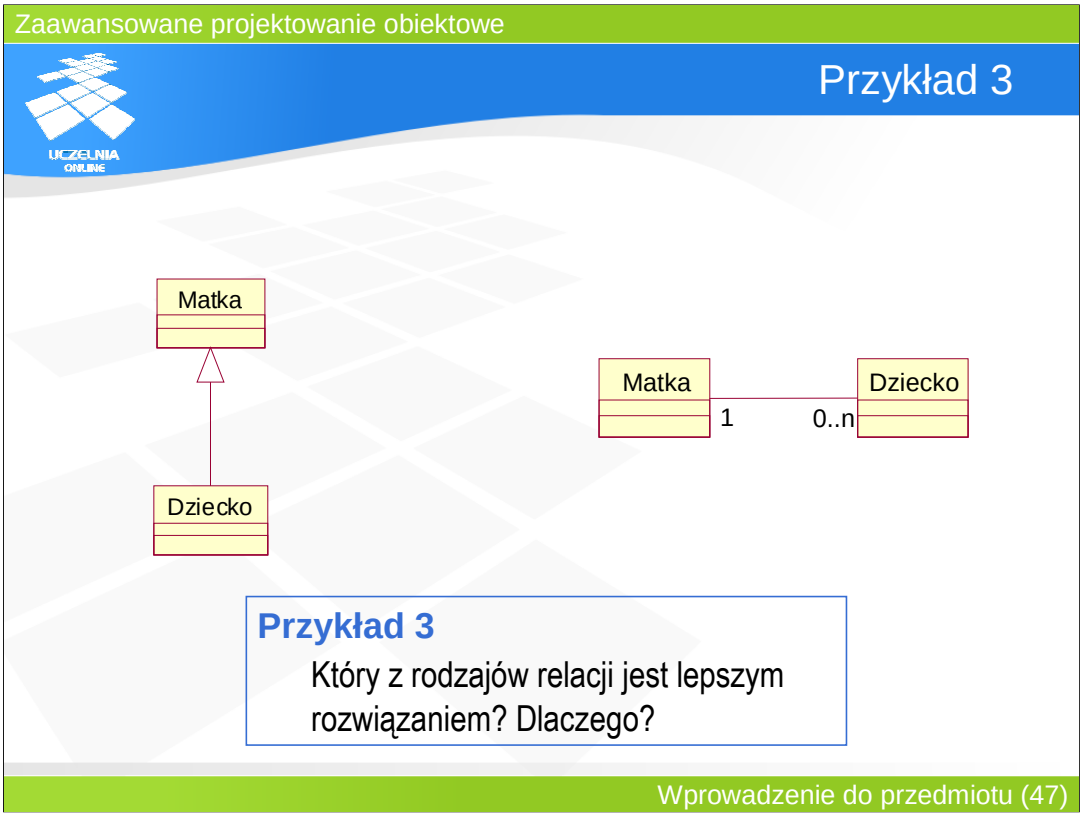


Drugi przykład dotyczy reprezentacji stanowisk, na jakich mogą być zatrudnieni pracownicy. Klasa Pracownik posiada metody *wynagrodzenie(),* zwracające wysokość wynagrodzenia zależnego od stanowiska, oraz *stanowisko(),* zwracające tekstowy opis stanowiska. Metoda *wynagrodzenie()* jest pokryta w podklasach.

Podobnie jak w poprzednim przypadku, należy określić wady i zalety tego rozwiązania, oraz zasugerować ewentualne zmiany.



Podobnie jak w poprzednim przykładzie, z klasy Pracownik została wydzielona klasa abstrakcyjna reprezentująca Stanowisko, która jest pokryta przez klasy Robotnik i Menedżer. Taka zmiana umożliwia pracownikowi zmianę stanowiska, która nie była możliwa w oryginalnym rozwiązaniu.



Trzeci przykład jest zadaniem do samodzielnego rozwiązania. Relacja między matką a dzieckiem może być reprezentowana w różny sposób. Na slajdzie przedstawiono dwie możliwe sposoby reprezentacji. Proszę wymienić zalety i wady każdego z tych rozwiązań oraz określić, które z nich jest lepsze, ewentualnie zaproponować nowe.

Podsumowanie

- Kluczowym elementem obiektowości jest odpowiedzialność
- Mechanizmy obiektowe pomagają realizować właściwy przydział odpowiedzialności
- Obiekty mogą być powiązane za pomocą kilku rodzajów relacji
- Spójność i powiązania określają jakość systemów obiektowych
- Karty CRC są prostą metodą projektowania systemów z uwzględnieniem odpowiedzialności obiektów

Wykład miał charakter przypomnienia wiadomości dotyczących obiektowego sposobu myślenia, znanych z innych przedmiotów. Przedstawiono podczas niego rolę odpowiedzialności jako podstawowego kryterium tworzenia klas i definiowania ich metod. Ponadto, w skrócie przypomniano mechanizmy obiektowe oraz rodzaje relacji, jakie mogą zaistnieć pomiędzy obiektami, wraz z omówieniem ich zastosowań. W dalszej części wykładu określone zostały dwa kryteria jakości projektu: spójność i stopień powiązań, oraz zaprezentowano ich interpretację. Ostatnią częścią wykładu było przedstawienie metody projektowania obiektowego z wykorzystaniem kart CRC, które pozwalają łatwo określać zobowiązania i odpowiedzialność poszczególnych klas.

Literatura

1. J. W. Cooper: *Java. Wzorce projektowe*. Helion 2001
2. B. Eckel: *Thinking in Java*. Wydanie III. Helion 2004
3. J. Shalloway, J. Trott: *Projektowanie zorientowane obiektowo. Wzorce projektowe*. Helion 2005
4. E. Gamma et al.: *Design patterns. Elements of reusable software.* Addison-Wesley 1995
5. M. Fowler: *Refactoring. Improving design of existing software.* Addison-Wesley 1999