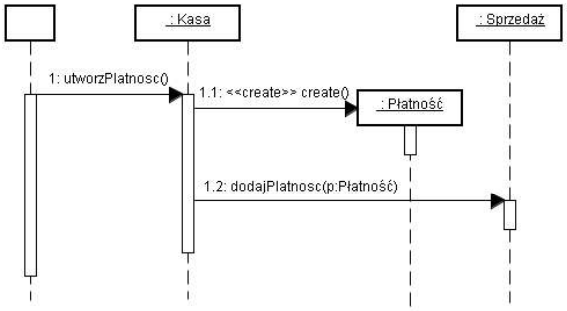
Wzorce projektowe określające zasady przypisywania odpowiedzialności obiektom c.d.

High Cohesion: założenia

- **Problem:** W jaki sposób zarządzać złożonością obiektu?
- **Rozwiązanie:** Odpowiedzialność powinna być przypisywana w taki sposób, aby spójność (cohesion) obiektu pozostawała wysoka.
- Spójność (cohesion) jest miarą na ile odpowiedzialności przypisane danemu obiektowi są ze sobą powiązane:
  - Klasa dla której odpowiedzialności są ściśle ze sobą powiązane oraz która nie ma przypisanych zbyt wielu zadań charakteryzuje się wysoką spójnością (high cohesion),
  - Klasa charakteryzująca się niską spójnością wykonuje wiele niepowiązanych ze sobą czynności, bądź wykonuje zbyt wiele zadań.
- Wady klas o niskiej spójności:
  - Trudne w powtórnym użyciu,
  - Trudne w utrzymaniu,
  - Trudno interpretowalne.
- Klasy o niskiej spójności przejmują często zadania innych klas!

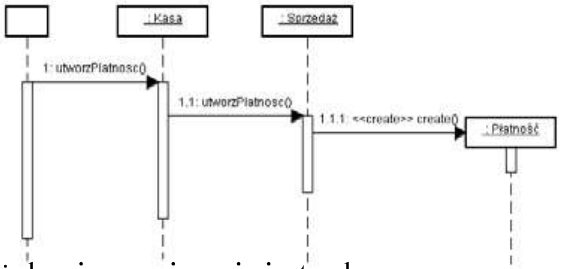
High Cohesion: przykład (1)

- W programie wielozadaniowy POS musimy rozstrzygnąć, kto jest odpowiedzialny za tworzenie instancji klasy *Płatność*?
- W związku z faktem, że w świecie rzeczywistym kasa zapisuje informacje o płatnościach, wzorzec projektowy Creator sugeruje klasę *Kasa* jako kandydata do przypisania odpowiedzialności tworzenia instancji klasy *Płatność*. Następnie utworzona instancja może zostać przesłana (jako parametr) przy pomocy metody *dodajPłatność* do obiektu *Sprzedaż*.



High Cohesion: przykład (2)

- Przyjęte rozwiązanie z punktu widzenia wzorca high cohesion jest niepoprawne w sytuacji gdy klasie *Kasa* trzeba będzie przypisać wiele odpowiedzialności. Uczyni to z tej klasy klasę o niskiej spójności.
- Wzorzec ten zaleca rozwiązanie inne, w którym płatność tworzona jest przez klasę *Sprzedaż*, co pozwala uzyskać wyższą spójność w klasie *Kasa*.



- Ponieważ drugie rozwiązanie jest zalecane przez wzorzec Low Coupling i High Cohesion to właśnie ono powinno zostać wybrane.

High Cohesion: dyskusja

- Podobnie jak Low Coupling wzorzec High Cohesion powinien być każdorazowo rozpatrywany przez projektanta podczas przypisywania odpowiedzialności danej klasie! Jest więc wzorcem oceniającym wyniki uzyskane przy pomocy innych wzorców.
- Klasa z wysoką spójnością posiada stosunkowo mało metod o ściśle powiązanych zadaniach. Klasa nie wykonuje zbyt wiele pracy – współpracuje z innymi obiektami w celu zlecania im wykonywania podzadań związanych z danym zadaniem.
- Wzorzec High Cohesion ma analogię do świata rzeczywistego – nie należy odciążać klasy zbyt wieloma zadaniami, tak samo jak nie należy obciążać danej osoby zbyt dużą ilością pracy. Bardziej korzystne w jednym i drugim przypadku jest delegowanie zadań.

High cohesion: różne poziomy spójności

- Bardzo niski poziom spójności – klasa jest samodzielnie odpowiedzialna za realizowanie zbyt wielu zadań z różnych obszarów funkcjonalnych.
- Średni poziom spójności – klasa jest samodzielnie odpowiedzialna za wykonywanie złożonych zadań z jednego obszaru funkcjonalnego.
- Wysoki poziom spójności – klasa ma przypisaną umiarkowaną ilość odpowiedzialności z jednego obszaru funkcjonalnego i współpracuje z innymi klasami w celu realizacji określonego zadania.
- Umiarkowany poziom spójności – klasa ma przypisaną umiarkowaną ilość odpowiedzialności w różnych obszarach funkcjonalnych, które to obszary są związane z klasą, ale nie są powiązane pomiędzy sobą.

Controller: założenia (1)

- **Problem:** Jaka klasa powinna być odpowiedzialna za obsługę zdarzeń systemowych?
- **Rozwiązanie:** Odpowiedzialność dotycząca obsługi komunikatów systemowych powinna być przypisywana jednej z następujących klas:
  - Reprezentującej cały system, urządzenie lub podsystem (kontroler fasadowy – facade controller)
  - Reprezentującej jeden przypadek użycia w ramach którego obsługiwane są pewne zdarzenia systemowe (kontroler przypadku użycia – use case controller). Kontrolery tego typu przyjmują nazwy: *<NazwaPrzypadkuUżycia>Handler,* *<NazwaPrzypadkuUżycia>Coordinator,* *<NazwaPrzypadkuUżycia>Session.*

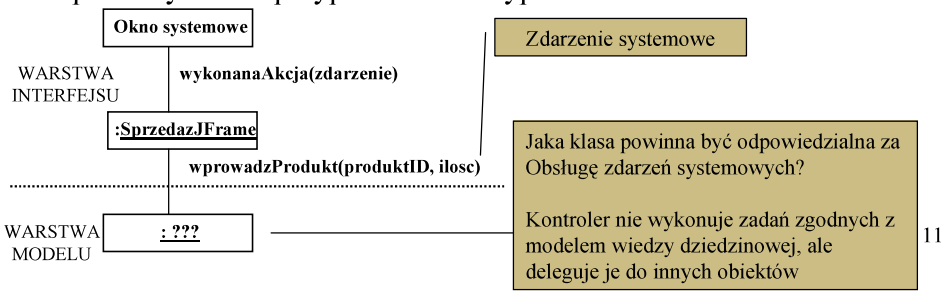
Controller: założenia (2)

- **Zdarzenie systemowe** to zdarzenie generowane przez aktorów zewnętrznych względem rozpatrywanego modelu.
- Zdarzenia takie są powiązane z **operacjami systemowymi**, czyli operacjami systemu wykonywanymi w odpowiedzi na zdarzenia systemowe.
- Kontroler to obiekt nie będący z poziomu interfejsu użytkownika, ale odpowiedzialny za obsługę zdarzeń systemowych. Kontroler definiuje metody umożliwiające realizację operacji systemowych.
- Kontroler to element składowy wzorca MVC (model-view-controller).

Controller: przykład (1)

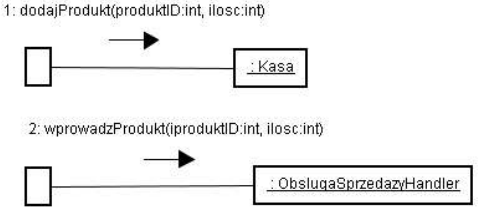
- W aplikacji wielozadaniowy POS można wyróżnić następujące operacje systemowe przypisane roboczo do klasy *System*:

- Powyższe nie oznacza, że klasa *System* zostanie wprowadzona do modelu. Odpowiedzialności związane z operacjami systemowymi powinny zostać przypisane klasie typu kontroler.



Controller: przykład (2)

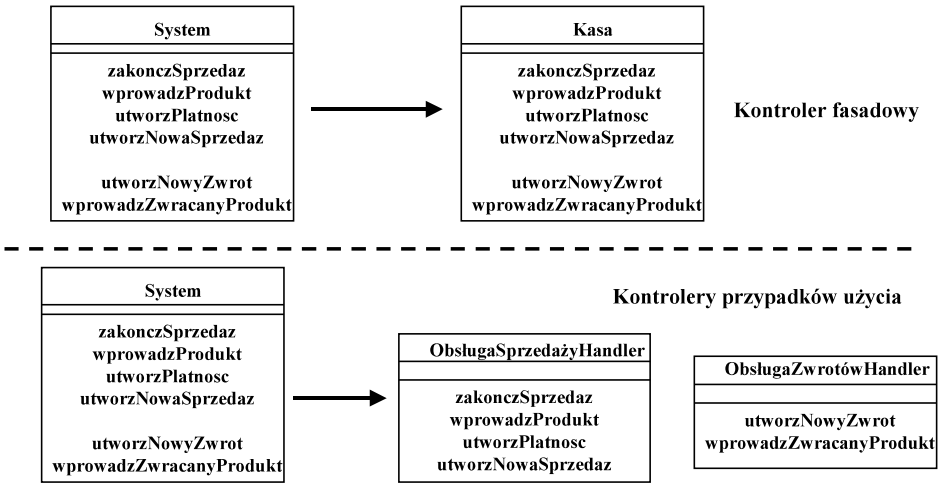
- Wzorzec Controller sugeruje następujących kandydatów na klasę kontrolera:
  - Klasa reprezentująca cały system: *Kasa*, *POSSystem*,
  - Klasa obsługująca zdarzenia systemowe związane z jednym przypadkiem użycia: *ObsługaSprzedażyHandler*.
- Diagramy interakcji dla tych klas są następujące:



- To która z tych dwóch klas zostanie wybrana jako najbardziej odpowiedni kontroler zależy od rozpatrywanego kontekstu.

Controller: przykład (3)

- Podczas fazy projektowej operacje systemowe zostają przypisane do jednego, bądź więcej kontrolerów:



Controller: dyskusja (1)

- Wzorzec Controller dostarcza rozwiązań na obsługę komunikatów różnego rodzaju, w szczególności tych generowanych przez GUI.
- Kontroler powinien delegować zadania do innych obiektów, jego rola powinna polegać na koordynacji i kontroli. Sam nie powinien realizować zbyt wiele.
- Kontrolery fasadowe powinny zostać wprowadzane jeżeli nie ma zbyt wiele komunikatów do obsłużenia lub interfejs użytkownika nie może przekierowywać obsługi komunikatów do wiele kontrolerów.
- Kontrolery przypadków użycia powinny być wprowadzane jeżeli jest wiele komunikatów rozdzielonych pomiędzy różne procesy systemowe. Kontrolery takie mają charakter czysto techniczny, nie znajdujemy dla nich pokrycia w modelu wiedzy dziedzinowej.

Controller: dyskusja (2)

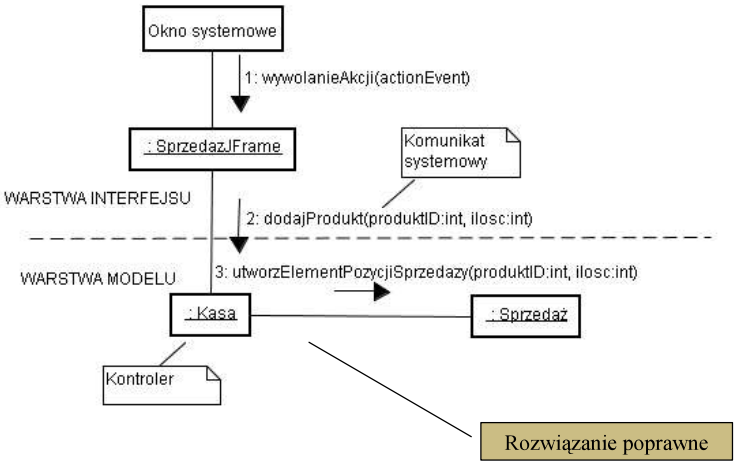
- Wzorzec Controller wyraża koncepcję, która głosi że klasy interfejsu użytkownika (okna, przyciski, itp.) nie powinny posiadać odpowiedzialności związanych z obsługą zdarzeń systemowych. Operacje systemowe powinny być obsługiwane na poziomie modelu dziedzinowego, a nie na poziomie interfejsu!!!
- Kontroler otrzymuje komunikaty z poziomu interfejsu użytkownika i koordynuje wypełnienie zadań w nich wyrażonych poprzez delegowanie zadań do innych obiektów.
- Korzyści:
  - Zwiększenie możliwości powtórnego użycia kodu - łatwość przenoszenia kodu pomiędzy różne narzędzia (biblioteki) do tworzenia interfejsu użytkownika,
  - Możliwość prostej weryfikacji realizacji danego przypadku użycia np. weryfikacja poprawnej kolejności obsługi komunikatów.

Controller: przeładowane kontrolery

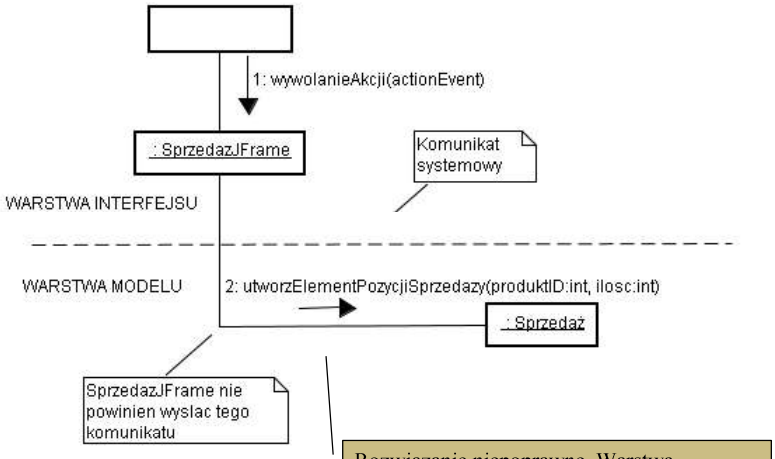
- Przeładowane kontrolery (bloated controller) to źle zaprojektowane kontrolery posiadające niską spójność tzn. posiadające zbyt dużo odpowiedzialności. Sytuacja taka zachodzi gdy:
  - Wprowadzony jest jedynie jeden kontroler otrzymujący wszystkie komunikaty systemowe, których jest wiele,
  - Kontroler wykonuje samodzielnie dużo zadań, aby zrealizować komunikaty systemowe, zamiast delegować zadania do innych obiektów,
  - Kontroler posiada wiele atrybutów i przechowuje istotne dla danego problemu informacje, które powinny być przechowywane w innych obiektach, bądź duplikuje informacje przechowywane w innych obiektach.
- Rozwiązanie problemu:
  - Wprowadzenie licznych kontrolerów,
  - Projektowanie kontrolerów, które poprawnie delegują zadania do innych obiektów

Controller: interfejs użytkownika i kontrolery (1)

- Klasy interfejsu użytkownika nie powinny posiadać odpowiedzialności związanych z obsługą zdarzeń systemowych:



Controller: interfejs użytkownika i kontrolery (2)



Rozwiązanie niepoprawne. Warstwa interfejsu nie powinna decydować jak obsługiwać procesy wiedzy dziedzinowej.

Nie jest pożądane, aby logika biznesowa była zawarta w warstwie prezentacji

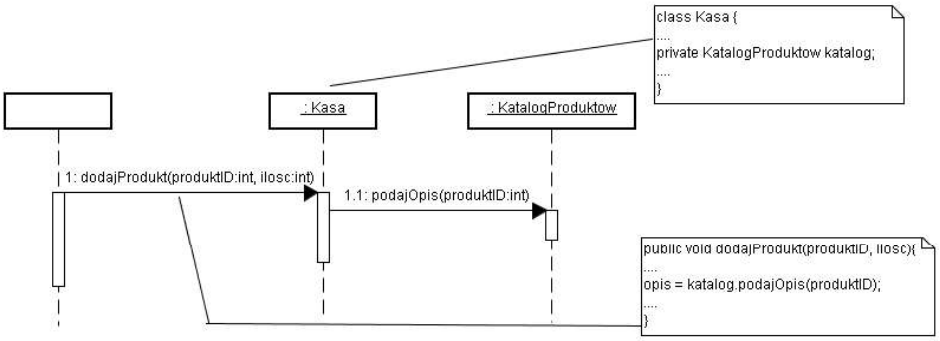
Widoczność

Wprowadzenie

- Widoczność to zdolność jednego obiektu do widzenia, bądź posiadania referencji do drugiego obiektu.
- Aby wysłać komunikat od obiektu nadawcy do obiektu odbiorcy konieczne jest, aby odbiorca był widoczny dla nadawcy tzn. aby nadawca miał referencję, bądź wskaźnik do obiektu odbiorcy.
- Występują następujące rodzaje widoczności:
  - Widoczność atrybutowa (attribute visibility) – B jest atrybutem A,
  - Widoczność parametryczna (parameter visibility) – B jest parametrem metody A,
  - Widoczność lokalna (local visibility) – B jest lokalnym (nie jest parametrem) obiektem w metodzie A,
  - Widoczność globalna (global visibility) – B jest widoczne globalnie.
- Najbardziej typowym rodzajem widoczności jest sytuacja, w której referencja, bądź wskaźnik do jednego obiektu jest atrybutem w innym obiekcie.

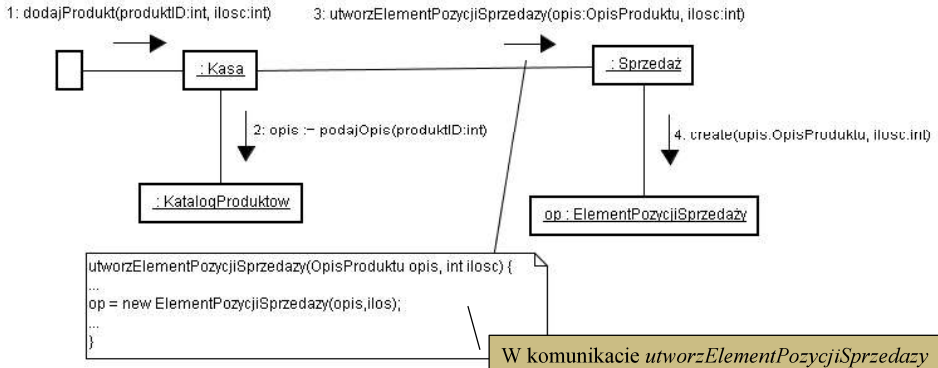
Widoczność atrybutowa

- Widoczność atrybutowa zachodzi gdy obiekt B jest atrybutem obiektu A. Widoczność jest trwała ponieważ istnieje tak długo jak istnieją obiekty A i B .
- Jest to najczęściej spotykany rodzaj widoczności.



Widoczność parametryczna

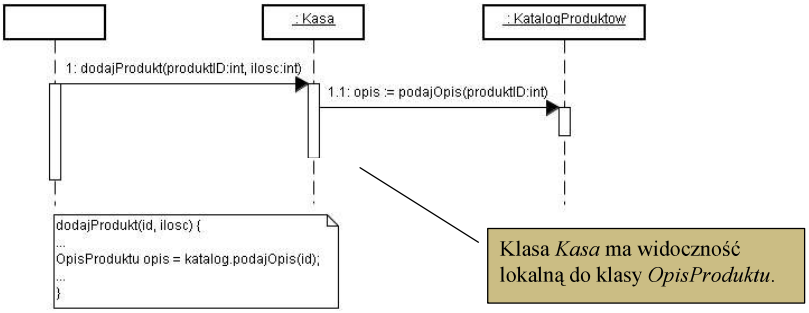
- Widoczność parametryczna zachodzi wtedy, gdy obiekt B jest przekazywany jako parametr do metody A. Widoczność ma charakter czasowy ponieważ trwa tak długo jak wykonywana jest metoda A.



W komunikacie *utworzElementPozycjiSprzedazy* przesyłanym do instancji klasy *Sprzedaż*, instancja klasy *OpisProduktu* jest przekazywana jako parametr.
Klasa *Sprzedaż* ma widoczność parametryczną do klasy *OpisProduktu*.

Widoczność lokalna

- Widoczność lokalna z klasy A do klasy B zachodzi wtedy, gdy B jest deklarowane jako lokalny obiekt w metodzie klasy A. Widoczność ma charakter czasowy ponieważ trwa tak długo jak wykonywana jest metoda A.
- Widoczność ta może zostać uzyskana poprzez:
  - Utworzenie nowego obiektu lokalnego i przypisania go do zmiennej lokalnej,
  - Przypisania zmiennej lokalnej obiektu zwracanego przez wywoływaną metodę.



Widoczność globalna

- Widoczność globalna pomiędzy klasą A, a klasą B istnieje wtedy gdy B jest globalnie dostępne dla klasy A. Widoczność jest trwała ponieważ istnieje tak długo jak istnieją obiekty A i B.
- Podejście to jest najrzadziej spotykane w programowaniu obiektowym!
- Jednym sposobem uzyskania tego typu widoczności jest przypisanie instancji klasy B do zmiennej globalnej (dostępne w języku C++).

Widoczność i UML

- UML pozwala na wyrażenie rodzaju widoczności w diagramach kolaboracji. Notacja ta jest używana opcjonalnie w sytuacjach, w których informacia o rodzaiu widoczności iest istotna.