Tworzenie modelu konceptualnego systemu informatycznego – część 2

1. **Diagramy klas UML**
   http://sparxsystems.com.au/resources/uml2_tutorial/
2. **Diagramy sekwencji UML**
   http://sparxsystems.com.au/resources/uml2_tutorial/
3. **Model konceptualny – model analizy**

Tworzenie modelu konceptualnego systemu informatycznego – część 2

1. **Diagramy klas UML**
   http://sparxsystems.com.au/resources/uml2_tutorial/

Diagramy UML 2 – część druga

Na podstawie

UML 2.0 Tutorial

http://sparxsystems.com.au/resources/uml2_tutorial/

Dwa rodzaje diagramów UML 2

Diagramy UML <u>modelowania strukturalnego</u>

- ***Diagramy pakietów***
- ***Diagramy klas***
- **Diagramy obiektów**
- **Diagramy mieszane**
- **Diagramy komponentów**
- **Diagramy wdrożenia**

Diagramy UML <u>modelowania zachowania</u>

- ***Diagramy przypadków użycia***
- **Diagramy aktywności**
- **Diagramy stanów**
- **Diagramy komunikacji**
- ***Diagramy sekwencji***
- **Diagramy czasu**
- **Diagramy interakcji**

Diagramy klas (Class Diagrams)

- **Diagram klas** reprezentuje statyczny model świata rzeczywistego: jego atrybuty i właściwości, odpowiedzialności oraz powiązania

- **Klasa** reprezentuje model rzeczy konceptualnej i fizycznej i jest powielana w postaci **obiektów**, czyli wystąpień klasy.

- **Atrybuty***:* składowe klasy do przechowywania danych, które posiadają nazwę, typ, zakres wartości oraz określony dostęp.

- **Operacje:** składowe klasy do wykonania operacji na atrybutach, zadeklarowane jako funkcje publiczne lub prywatne posiadające nazwę oraz zdefiniowany sposób wykonania.

Notatacje

Atrybuty: length, width, center**.** Atrybut **center** posiada wartość początkową.

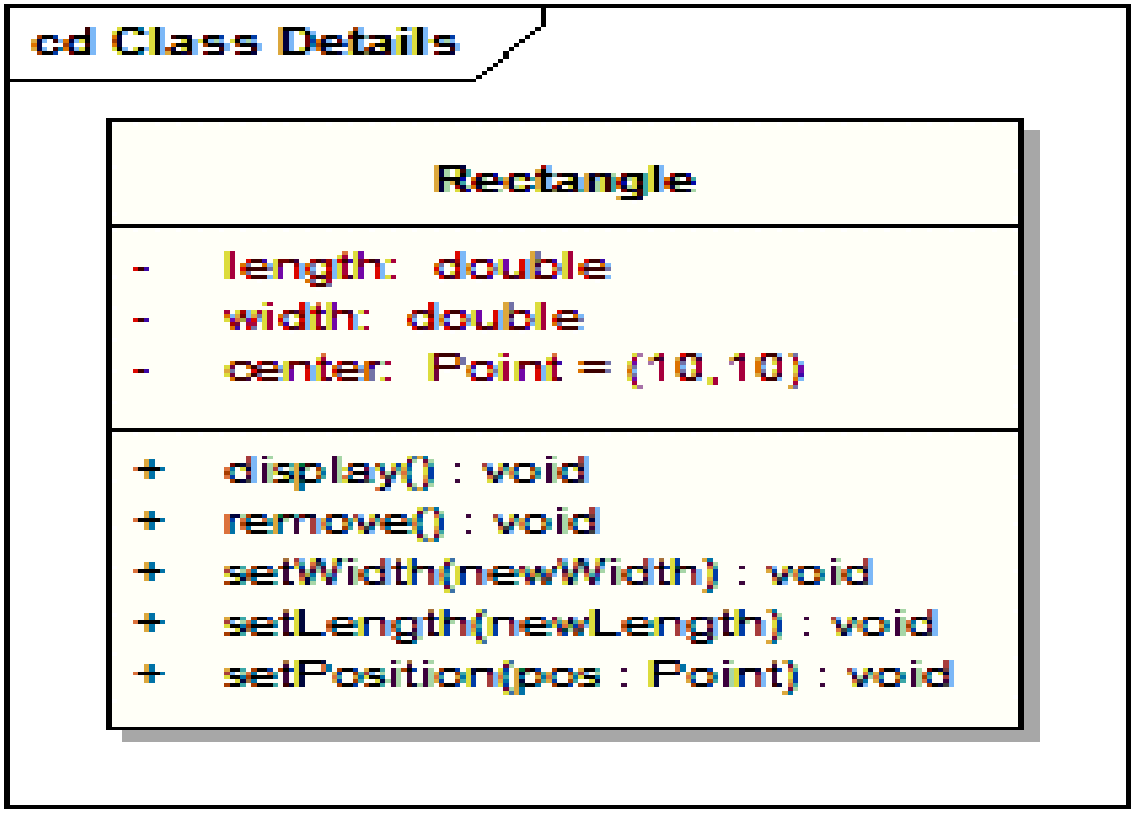
Operacje: setWidth, setLength, setPosition

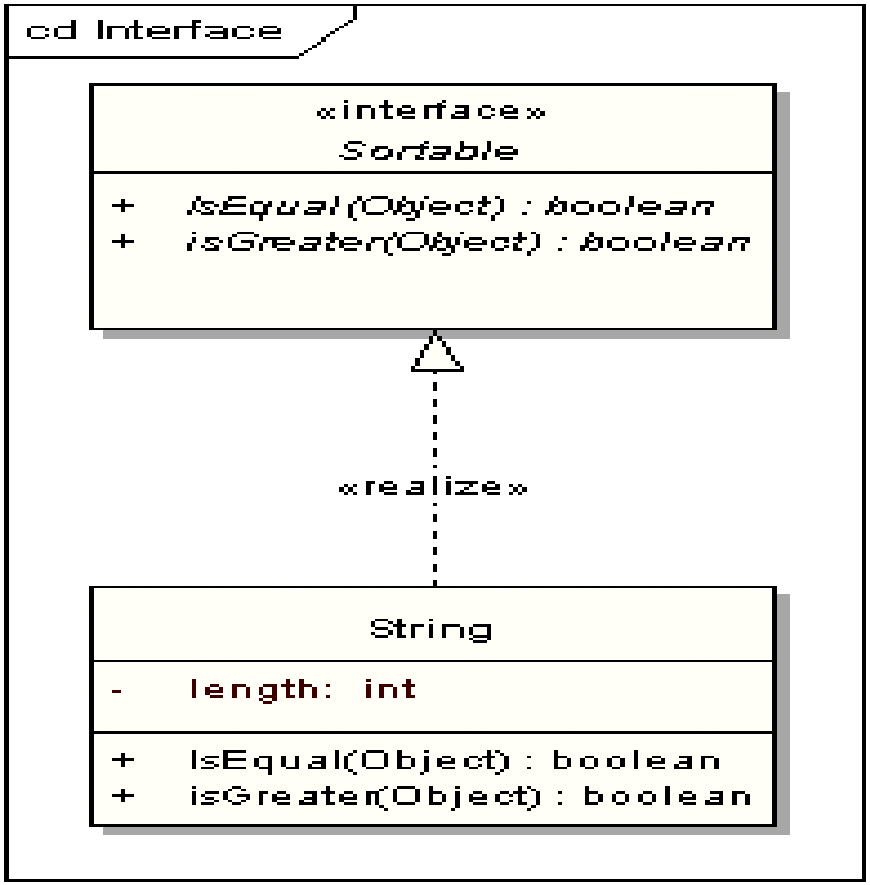
+ składowa publiczna

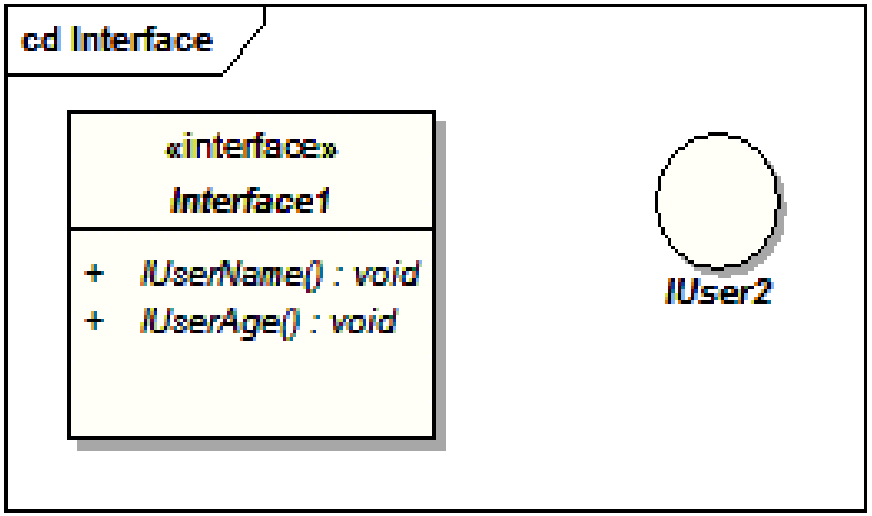
- składowa prywatna

składowa typu protected

~ składowa publiczna w zasięgu pakietu







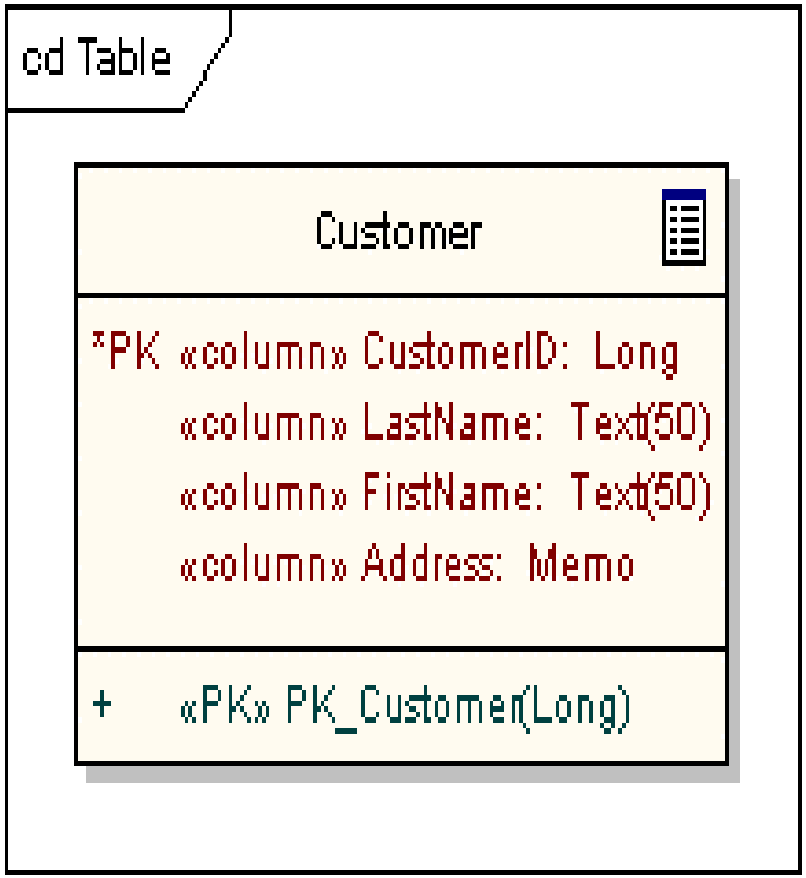
Interfejs<<interface>>

przedstawiany jako klasa zawierająca specyfikację właściwości (operacji czyli metod), które musi zdefiniować implementująca go klasa

Klasa implementująca metody inerfejsu

- **klasa implementująca jest rysowana podobnie jak klasa i jest** połączona relacją typu <<realize>> (strzałka przerywana wychodząca tej z klasy) z klasą typu <<interface>>

- **implementująca klasa reprezentowana jest jako koło** bez wyspecyfikowanych metod i połączenia z interfejsem nie są oznaczane strzałką



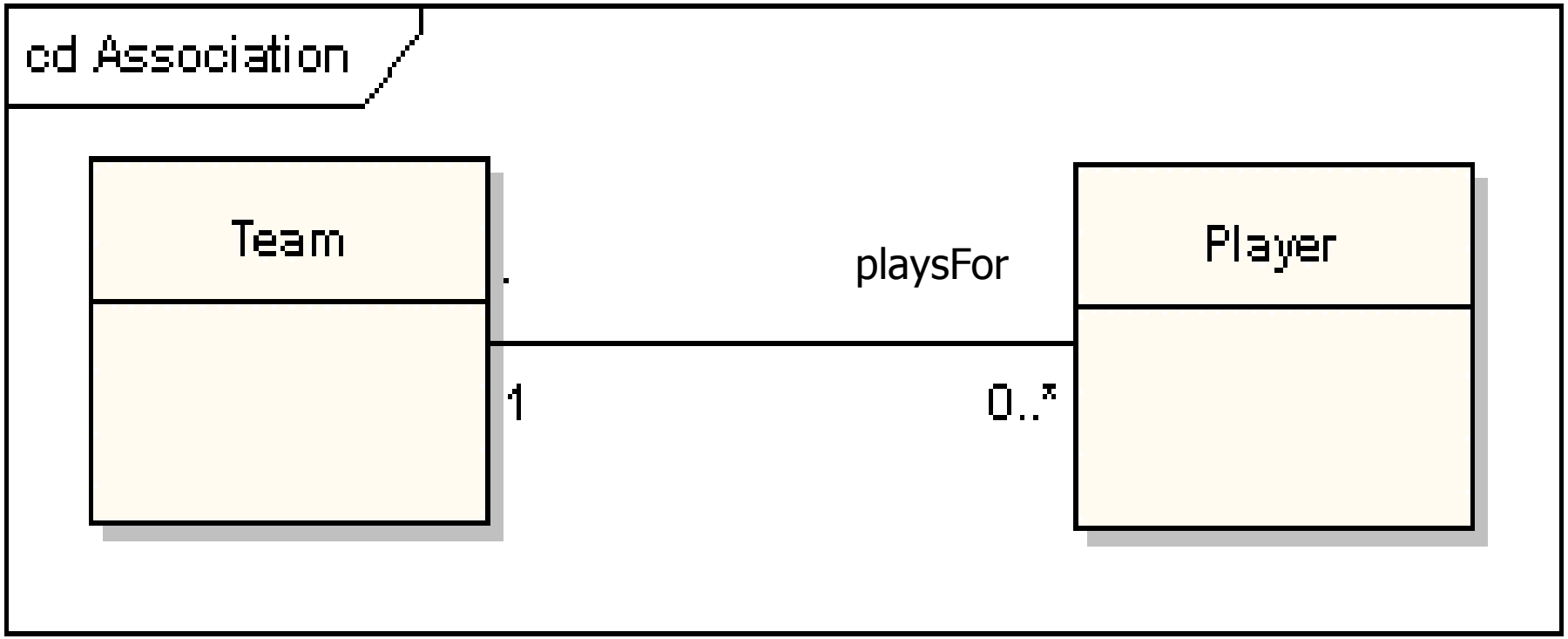
Tabele (table)

- **Klasa stereotypowa**
- **Atrybuty tabeli o stereotypie <<column>>**
- **Posiada klucz główny (<<PK>> – primary key)** obejmujący jedną lub wiele kolumn o unikatowym znaczeniu
- **Może posiadać jeden lub wiele kluczy obcych (<<FK>>- foreign key)** jako kluczy głównych w powiązanych tabelach po stronie „1”

Powiązanie (Association)

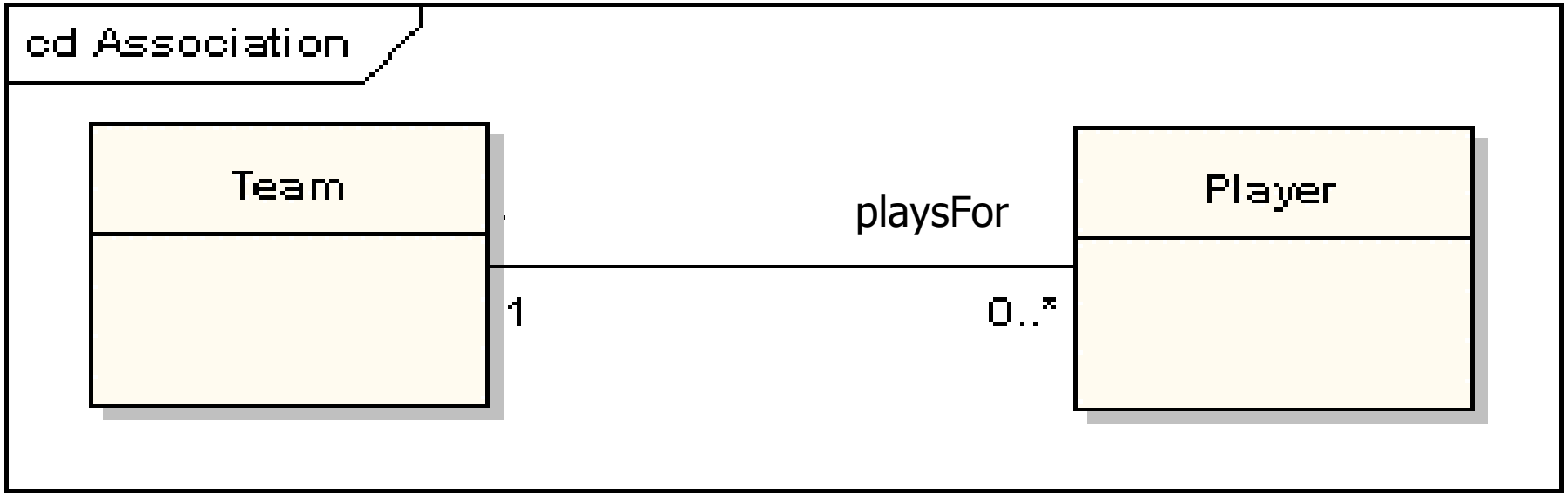
Wiąże dwa elementy modelu w związek strukturalny

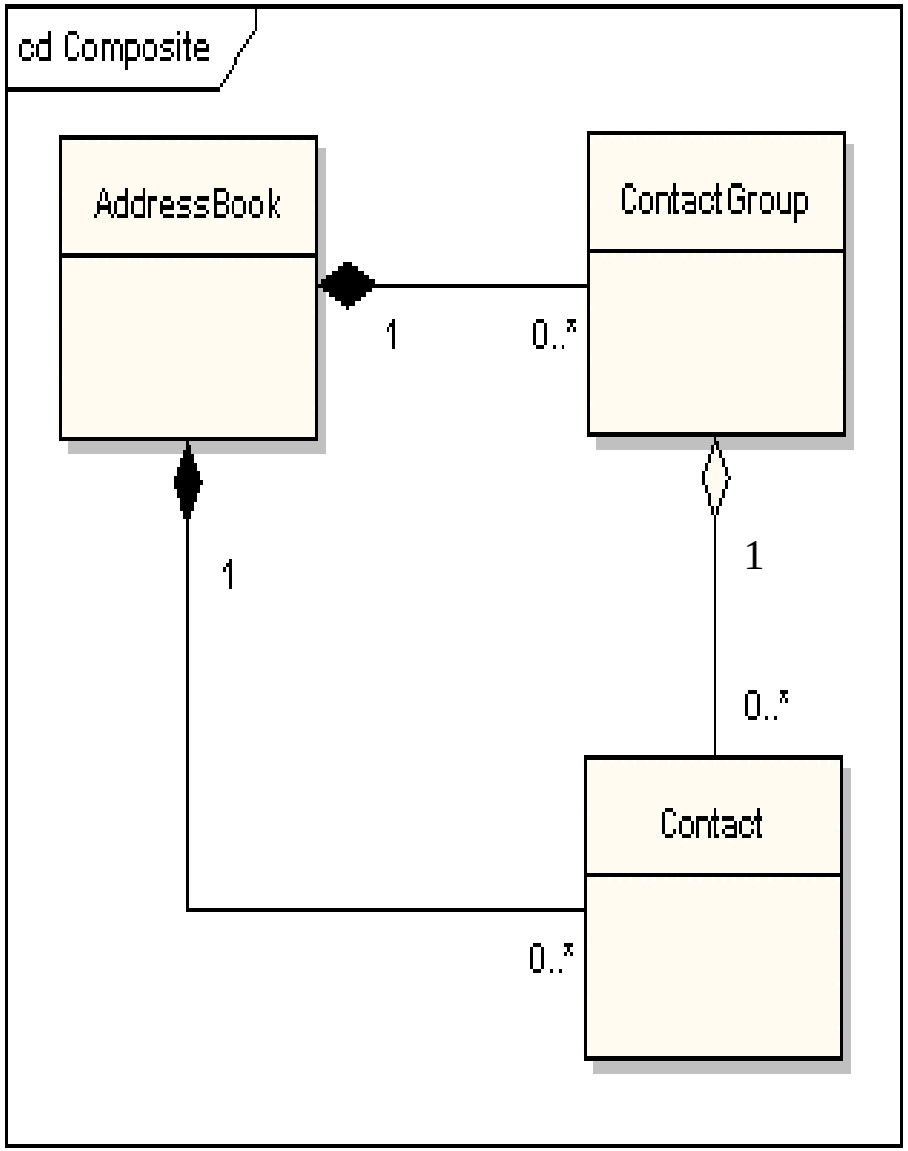
- Połączenie może zawierać nazwy ról na każdym końcu, liczność wystąpień instancji tych elementów, kierunek oraz ograniczenia
- Dla większej liczby powiązanych elementów jest przedstawiana jako romb



- **Jest implementowana następująco:**
  1. **relacje wiele do jeden** lub **jeden do jeden: w obiekcie po stronie wiele lub jeden znajduje się referencja do obiektu z przeciwnej strony relacji (strony jeden)**
  2. **relacje jeden do wiele: kolekcja referencji instancji obiektów po stronie wiele w obiekcie po stronie jeden**

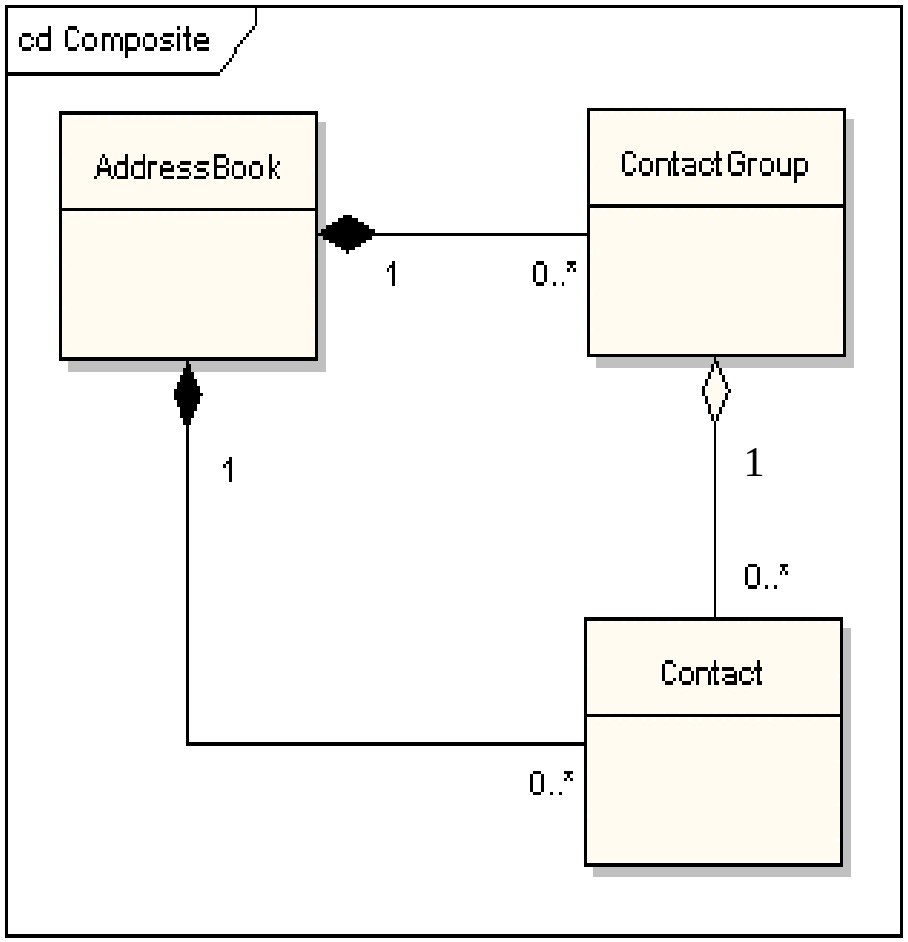
  (np. referencja do obiektu typu Team występuje w obiekcie typu Player oraz kolekcja referencji obiektów typu Player w obiekcie klasy Team)





Agregacja (Aggregation)

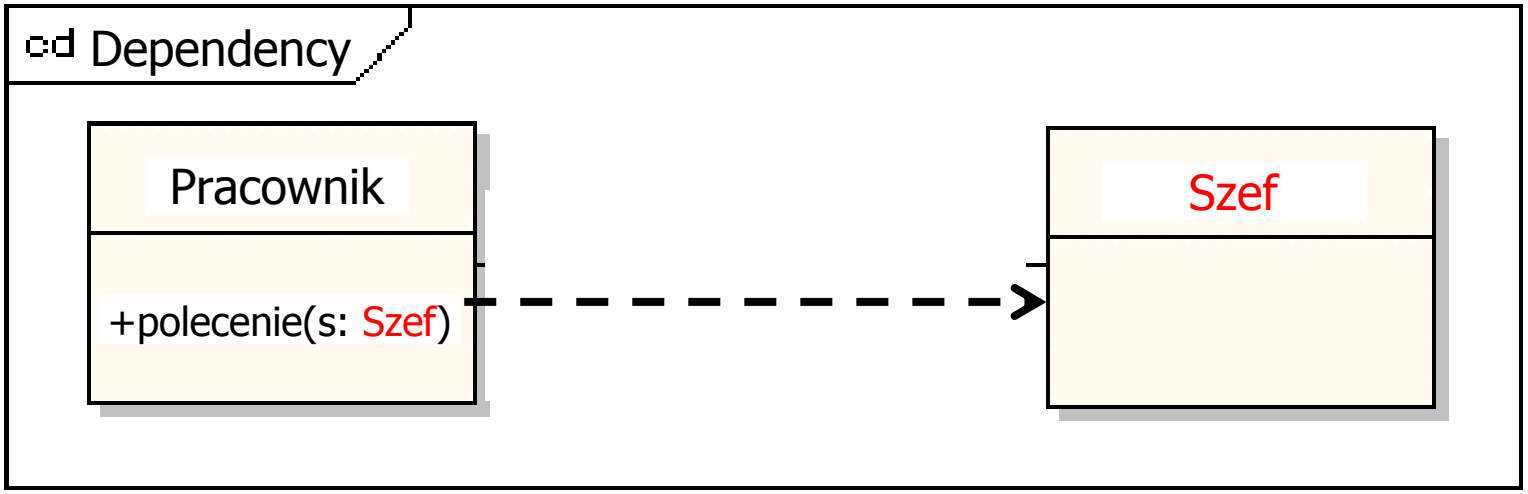
- **Oznacza elementy składające się z innych elementów**
- **Jest tranzytywna, symetryczna, może być rekursywna**
- **Jest wyrażana za pomocą rombów białych i czarnych, umieszczonych przy klasach agregujących**
- **Romby czarne- silna agregacja oznaczająca, że przy usuwaniu obiektu klasy agregującej usuwany jest obiekt klasy agregowanej**
- **Romby białe – słaba agregacja nie pociąga za sobą usuwania z pamięci obiektów agregowanych, gdy usuwany jest obiekt agregujący**



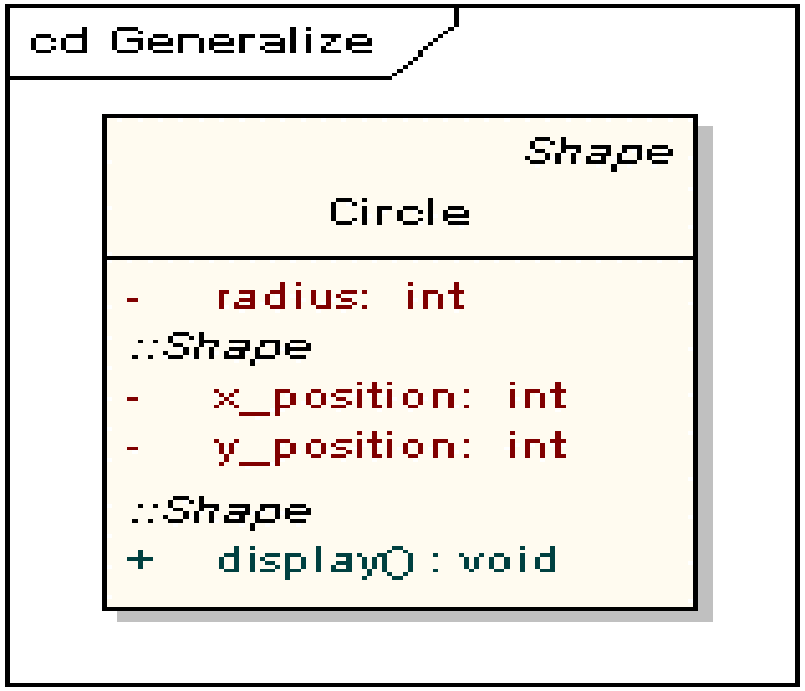
- **Jest implementowana podobnie jak związek typu Association**
  - rerefencje obiektu typu AddressBook oraz typu ContactGroup są atrybutami w obiekcie typu Contact
  - dwie kolekcje referencji obiektów typu Contact oraz referencji obiektów typu ContactGroup są atrybutami w obiekcie klasy AddressBook
  - Obiekt typu ContactGroup zawiera kolekcję referencji do obiektów typu Contact oraz referencję do obiektu typu AddressBook

Przykład: agregacja wielu obiektów klasy *ContactGroup oraz Contact* w księdze adresowej *AddressBook* stanowi silną agregację. Obiekt klasy *ContactGroup* agreguje wiele obiektów klasy *Contact* w sposób słaby. Usunięcie obiektu klasy *AddressBook* pociąga za sobą usuniecie obiektów klasy *Contact* i *ContactGroup*, usunięcie obiektu klasy *Contact Gropup* nie pociąga za sobą usuwania obiektów klasy *Contact*

Zależność (Dependency)



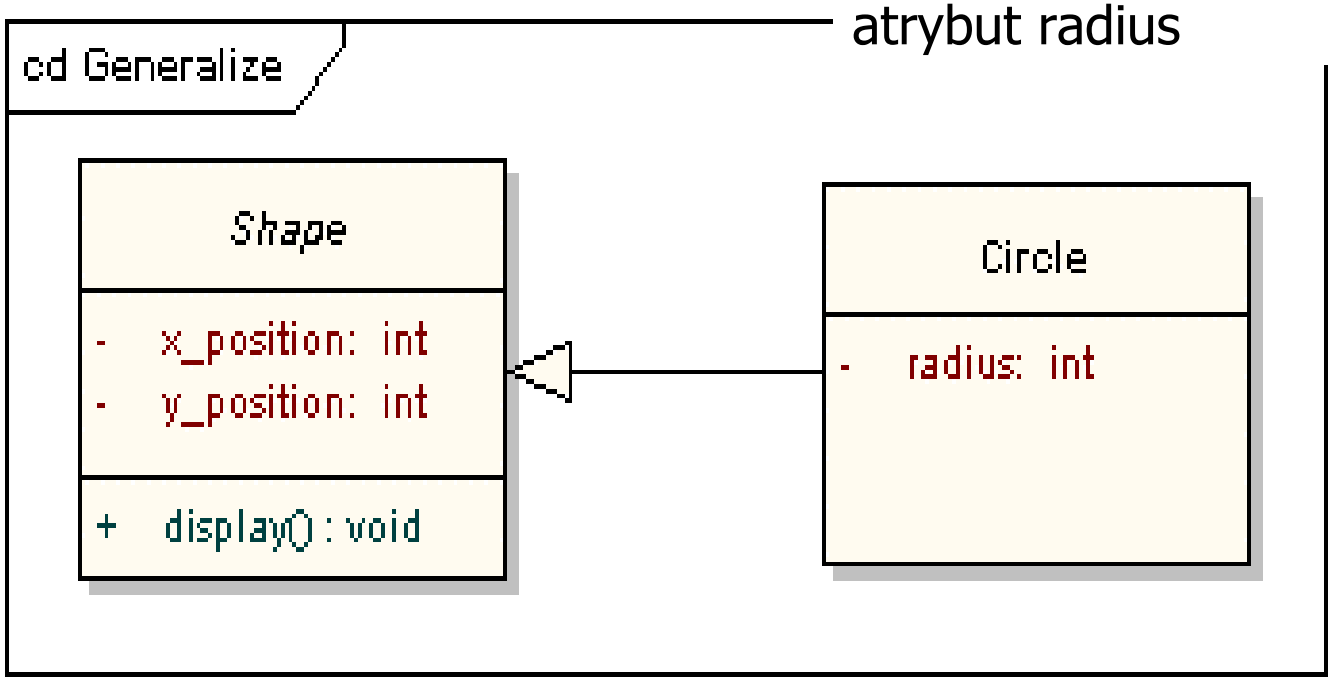
- Zależności są używane do modelowania powiązań między elementami modelu we wczesnej fazie projektowania, jeśli nie można określić precyzyjnie typu powiązania. Stanowią one wtedy związek użycia **(<<usage>>)**.
- Strałka przerywana wskazuje grotem na klasę, od której coś zależy.
- Później są one uzupełniane o stereotypy: «instantiate», «trace», «import» itp. lub zastąpione innym specjalizowanym połączeniem
- **Implementacja zależności: klasa z operacją jest klasą zależną, natomiast parametr tej operacji jest obiektem typu klasy, od której coś zależy**

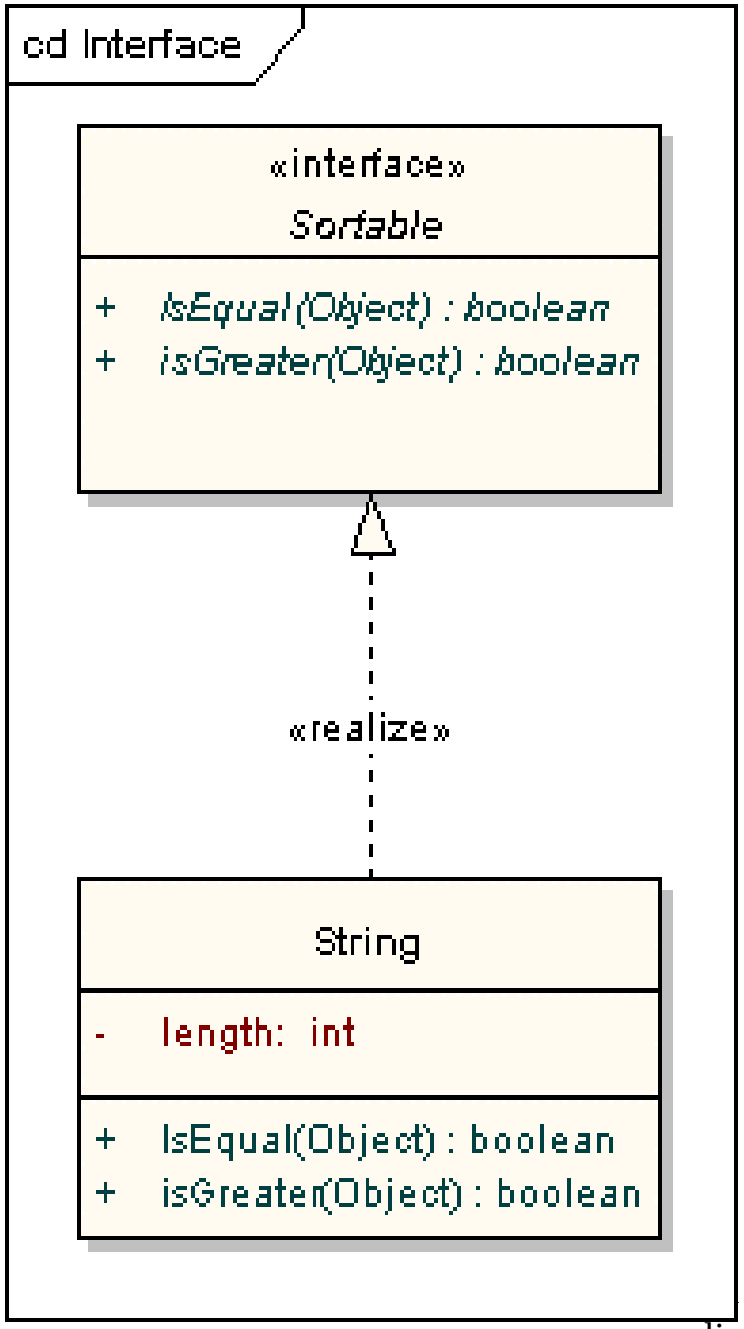


Generalizacja czyli dziedziczenie (Generalization)

Używana do oznaczania dziedziczenia

- **Strzałka wychodzi z klasy** dziedziczącej do klasy, po której dziedziczy
- np. Klasa Circle dziedziczy atrybuty ***x_position, y_position*** **i metodę display() po klasie Shape** oraz dodaje atrybut radius





Realizacja (Realization)

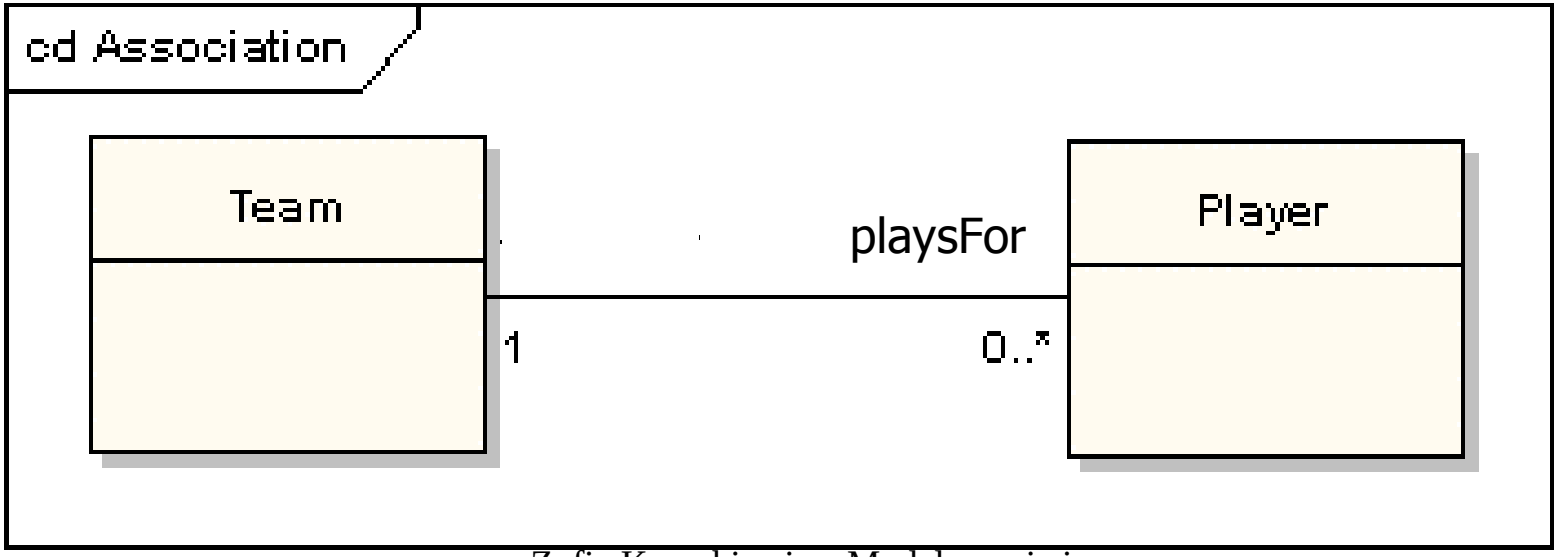
- Oznaczane są przerywaną strzałką ze stereotypem **<<realize>>**
- strzałka wychodzi z klasy implementującej do klasy implementowanej
- implementacja właściwości klasy typu ***interface***

Specjalizacja zależności (Trace)

- Łączy elementy modelu o tym samym przeznaczeniu, wymaganiach lub tym samym momencie zmian
- Ma znaczenie informacyjne

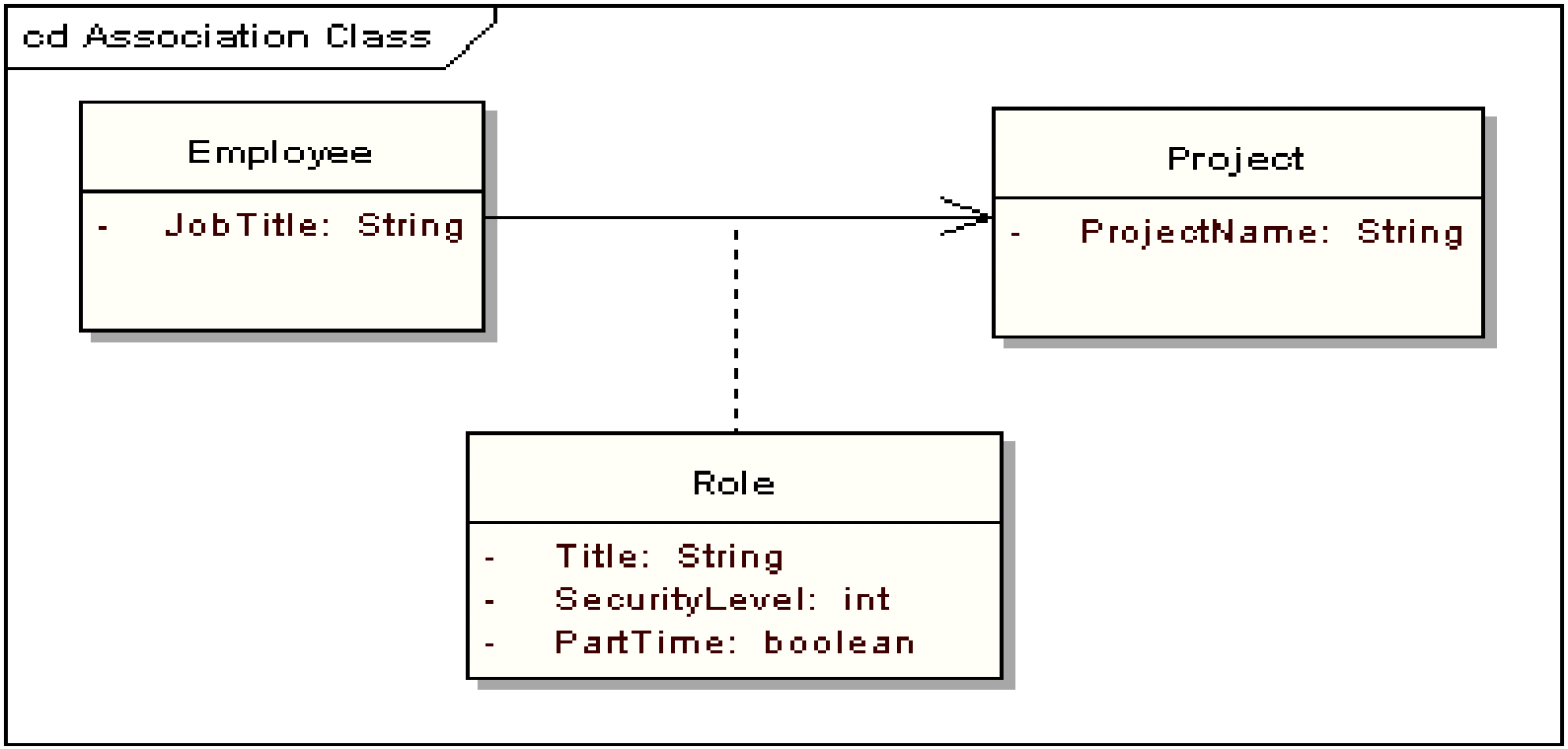
Role

Role to oblicze, jakie prezentuje klasa przy jednym końcu drugiej klasie na drugim końcu



Klasa powiązań (Association Class)

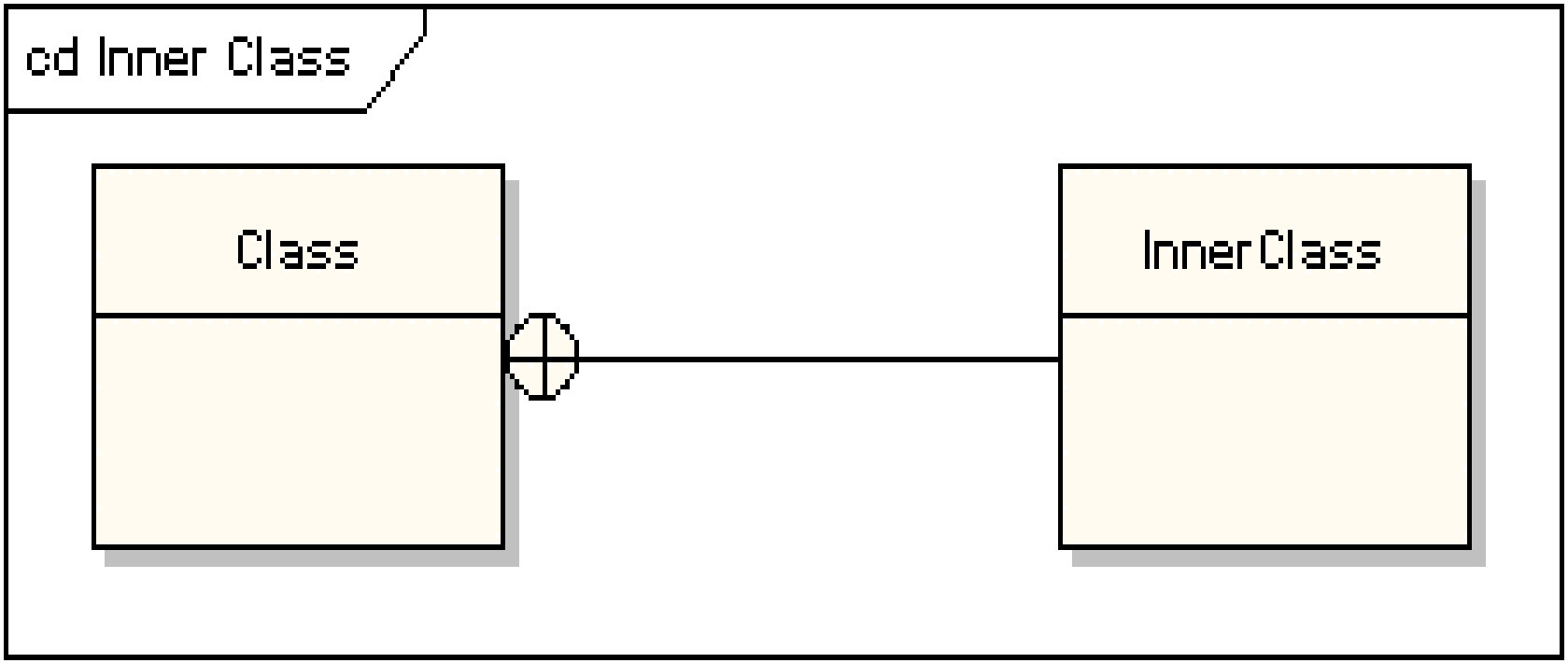
- uzupełnia powiązane obiekty o atrybuty i metody
- np. powiązanie między projektem (obiekt klasy *Project*) a wykonawcą (obiekt klasy *Employee*) dodatkowo jest opisane za pomocą składowych obiektu klasy *Role*. Obiekt klasy *Role* jest przypisany w powiązaniu do jednej pary obiektów klas *Employee* i *Project*, które dodatkowo opisuje jako konkretnego pracownika wykonującego dany projekt

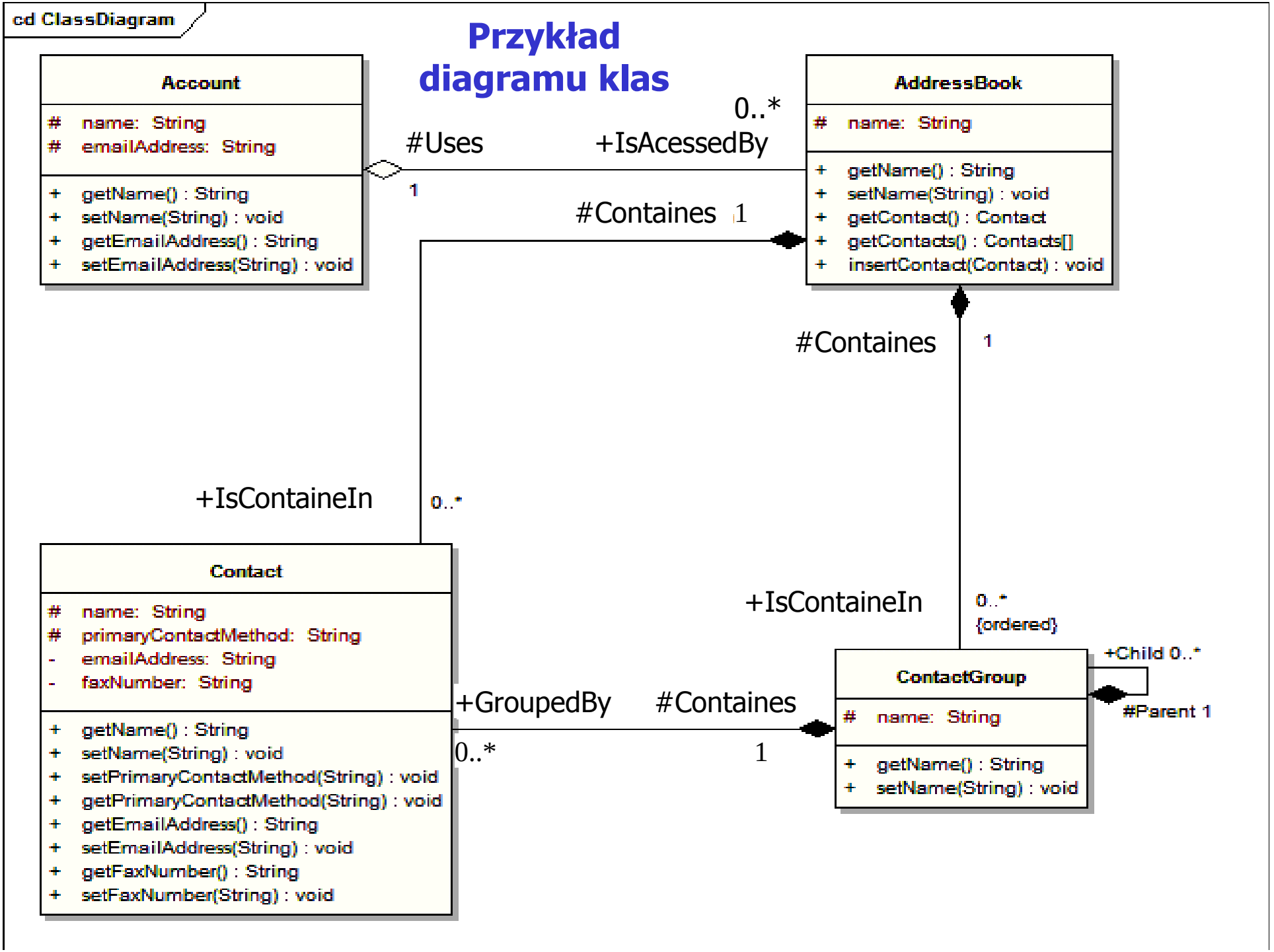


Zagnieżdżenie (Nesting)

• symbol zagnieżdżenia oznacza, że klasa, do której symbol jest dołączony, posiada zagnieżdżoną klasę dołączoną z drugiej strony zagnieżdżenia

• np. Klasa ***Class*** ma zagnieżdżoną klasę ***InnerClass***




cd ClassDiagram
Przykład diagramu klas
Account
name: String
emailAddress: String
+ getName() : String
+ setName(String) : void
+ getEmailAddress() : String
+ setEmailAddress(String) : void
AddressBook
name: String
+ getName() : String
+ setName(String) : void
+ getContact() : Contact
+ getContacts() : Contacts[]
+ insertContact(Contact) : void
#Uses
1
0..*
+IsAcessedBy
#Containes
1
#Containes
1
+IsContaineIn
0..*
Contact
name: String
primaryContactMethod: String
- emailAddress: String
- faxNumber: String
+ getName() : String
+ setName(String) : void
+ setPrimaryContactMethod(String) : void
+ getPrimaryContactMethod(String) : void
+ getEmailAddress() : String
+ setEmailAddress(String) : void
+ getFaxNumber() : String
+ setFaxNumber(String) : void
+IsContaineIn
0..*
{ordered}
+GroupedBy
0..*
#Containes
1
ContactGroup
name: String
+ getName() : String
+ setName(String) : void
+Child 0..*
#Parent 1

Identyfikacja klas

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Zidentyfikuj zbiór klas, które współpracują ze sobą w celu wykonania poszczególnych czynności
- Określ zbiór zobowiązań każdej klasy
- Rozważ zbiór klas jako całość: podziel na mniejsze te klasy, które mają zbyt wiele zobowiązań; scal w większe te klasy, które mają zbyt mało zobowiązań
- Rozpatrz sposoby wzajemnej kooperacji tych klas i porozdzielaj ich zobowiązania tak, aby żadna z nich była ani zbyt złożona ani zbyt prosta
- Elementy nieprogramowe (urządzenia) przedstaw w postaci klasy i odróżnij go za pomocą własnego stereotypu; jeśli ma on oprogramowanie, może być traktowany jako węzeł diagramu klas w celu rozwijania tego oprogramowania
- Zastosuj typy pierwotne (tabele, wyliczenia, typy proste np. boolean itp)

Identyfikacja związków: zależność (Dependency)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

Modelowanie zależności

- Utworzyć zależności między klasą z operacją a klasą użytą jako parametr tej operacji
- Stosuj zależności tylko wtedy, gdy modelowany związek nie jest strukturalny

Identyfikacja związków: generalizacja czyli dziedziczenie (Generalization)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Ustaliwszy zbiór klas poszukaj zobowiązań, atrybutów i operacji wspólnych dla co najmniej dwóch klas
- Przenieś te wspólne zobowiązania, atrybuty i operacje do klasy bardziej ogólnej; jeśli to konieczne, utwórz nową klasę, do której zostaną przypisane te właśnie byty (uważaj z wprowadzaniem zbyt wielu poziomów)
- Zaznacz, że klasy szczegółowe dziedziczą po klasie ogólnej, to znaczy uwzględnij uogólnienia biegnące od każdego potomka do bardziej ogólnego przodka
- Stosuj uogólnienia tylko wtedy, gdy masz do czynienia ze związkiem „jest rodzajem”; dziedziczenie wielobazowe często można zastąpić agregacją
- Wystrzegaj się wprowadzania cyklicznych uogólnień
- Utrzymuj uogólnienia w pewnej równowadze; krata dziedziczenia nie powinna być zbyt głęboka (pięć lub więcej poziomów już budzi wątpliwości) ani zbyt szeroka (lepiej wprowadzić pośrednie klasy abstrakcyjne)

Identyfikacja związków strukturalnych: powiązanie (Association) , agregacja (Aggregation)

(wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Rozważ, czy w wypadku każdej pary klas jest konieczne przechodzenie od obiektów jednej z nich do obiektów drugiej
- Rozważ, czy w wypadku każdej pary klas jest konieczna inna interakcja między obiektami jednej z nich a obiektami drugiej niż tylko przekazywanie ich jako parametrów; jeśli tak, uwzględnij powiązanie między tymi klasami, w przeciwnym wypadku jest to zależność użycia. Ta metoda identyfikacji powiązań jest oparta na zachowaniu
- Dla każdego powiązania określ liczebność (szczególnie wtedy, kiedy nie jest to 1 - wartość domyślna) i nazwy ról (ponieważ ułatwiają zrozumienie modelu)
- Jeśli jedna z powiązanych klas stanowi strukturalną lub organizacyjną całość w porównaniu z klasami z drugiego końca związku, które wyglądają jak części, zaznacz przy niej specjalnym symbolem, że chodzi o agregację.
- Stosuj powiązania głównie wtedy, kiedy między obiektami zachodzą związki strukturalne

Identyfikacja wzorców projektowych

- Dobrze zbudowany system obiektowy jest pełen wzorców obiektowych
- Wzorzec to zwyczajowo przyjęte rozwiązanie typowego problemu w danym kontekście
- Strukturę wzorca przedstawia się w postaci diagramu klas
- Zachowanie się wzorca przedstawia się za pomocą diagramu sekwencji
- Wzorce projektowe: Wzorzec reprezentuje powiązanie problemu z rozwiązaniem
  (wg Booch G., Rumbaugh J., Jacobson I., UML przewodnik użytkownika)

- Każdy wzorzec składa się z trzech części, które wyrażają związek między konkretnym kontekstem, problemem i rozwiązaniem **(Christopher Aleksander)**
- Każdy wzorzec to trzyczęściowa reguła, która wyraża związek między konkretnym kontekstem, rozkładem sił powtarzającym się w tym kontekście i konfiguracją oprogramowania pozwalająca na wzajemne zrównoważenie się tych sił w celu rozwiązania zadania. **(Richar Gabriel)**
- Wzorzec to pomysł, który okazał się użyteczny w jednym rzeczywistym kontekście i prawdopodobnie będzie użyteczny w innym. **(Martin Fowler)**

Tworzenie modelu konceptualnego systemu informatycznego – część 2

1. **Diagramy klas UML**

   http://sparxsystems.com.au/resources/uml2_tutorial/

2. **Diagramy sekwencji UML**

   http://sparxsystems.com.au/resources/uml2_tutorial/

Diagramy sekwencji (Sequence Diagrams)

- wyrażają **interakcje w czasie** (wiadomości wymieniane między obiektami jako poziome strzałki wychodzące od linii życia jednego obiektu i wchodzące do linii życia drugiego obiektu)

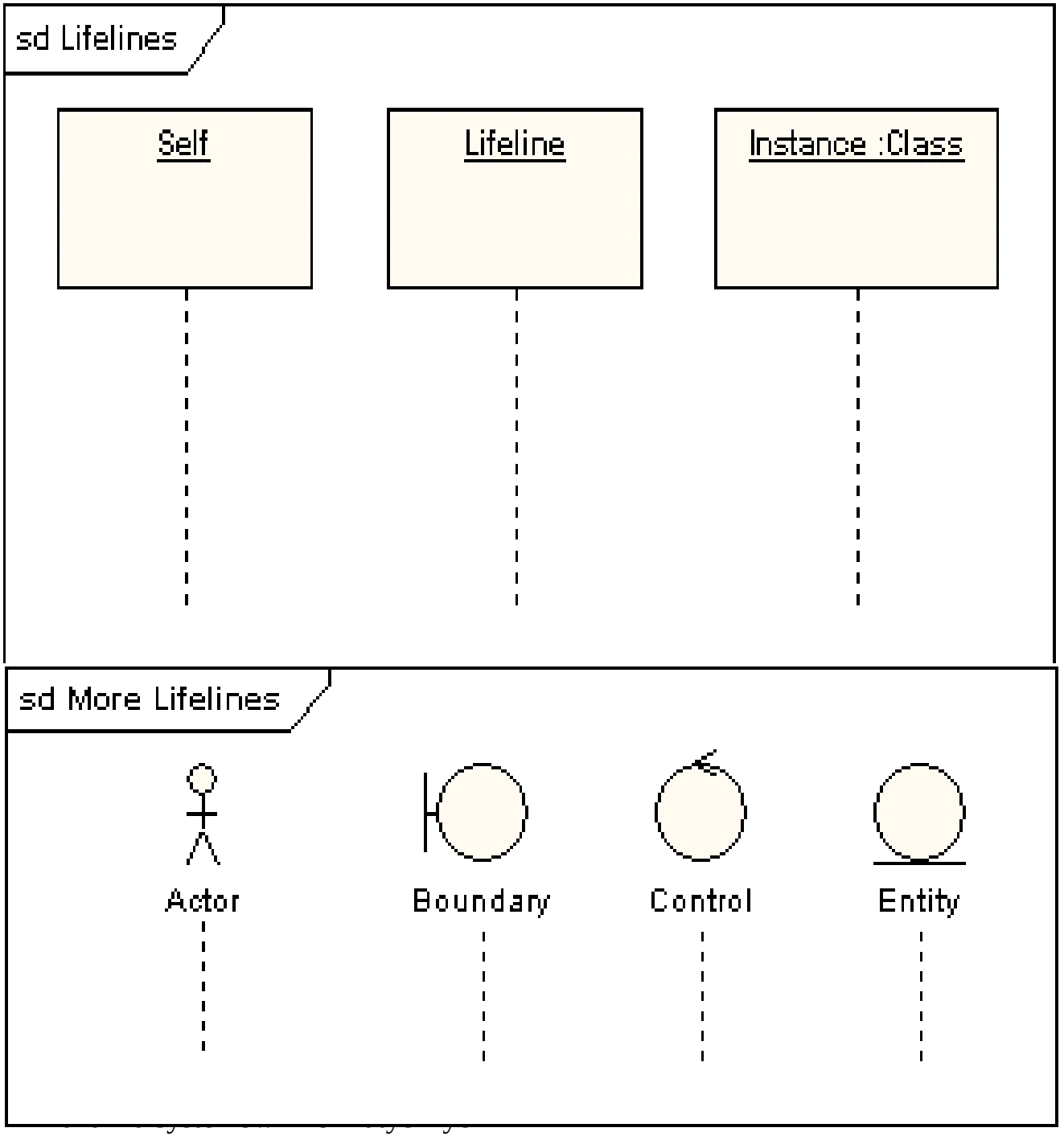
- wyrażają dobrze **komunikację** między obiektami i zarządzanie przesyłaniem wiadomości

- **nie są używane do wyrażania złożonej logiki proceduralnej**

- **są używane do modelowanie scenariusza przypadku użycia**

Linie życia (Lifelines)
Linie życia reprezentują indywidualne uczestniczenie obiektu w diagramie. Posiadają one często prostokąty zawierające nazwę i typ obiektu.

Czasem diagram sekwencji zawiera **linię życia aktora**. Oznacza to, że właścicielem diagramu sekwencji jest **przypadek użycia**. Elementy oznaczające ***obiekty typu „boundary", „control"*** **,** ***„Entity"*** mają również swoje linie życia.

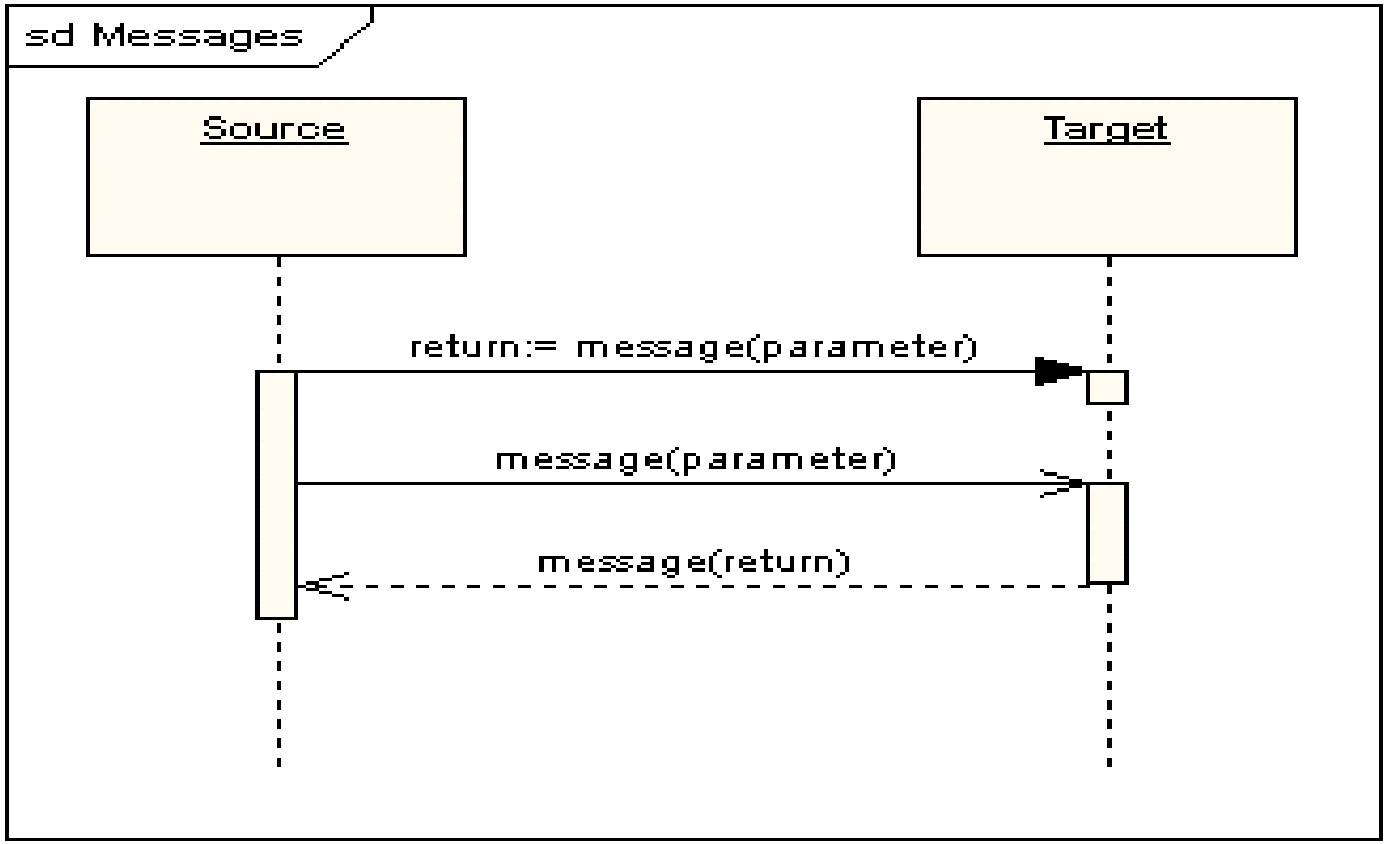


Wiadomości (Messages)

- są wyświetlane jako strzałki.
- mogą być ***kompletne, zgubione i znalezione;***
- mogą być ***synchroniczne i asychroniczne***
- Mogą być typu wywołanie operacji **(*call)*** lub sygnał (***signal***)
- **dla wywołań operacji (call) wyjście strzałki z linii życia oznacza, że obiekt ten wywołuje metodę obiektu, do którego strzałka dochodzi**

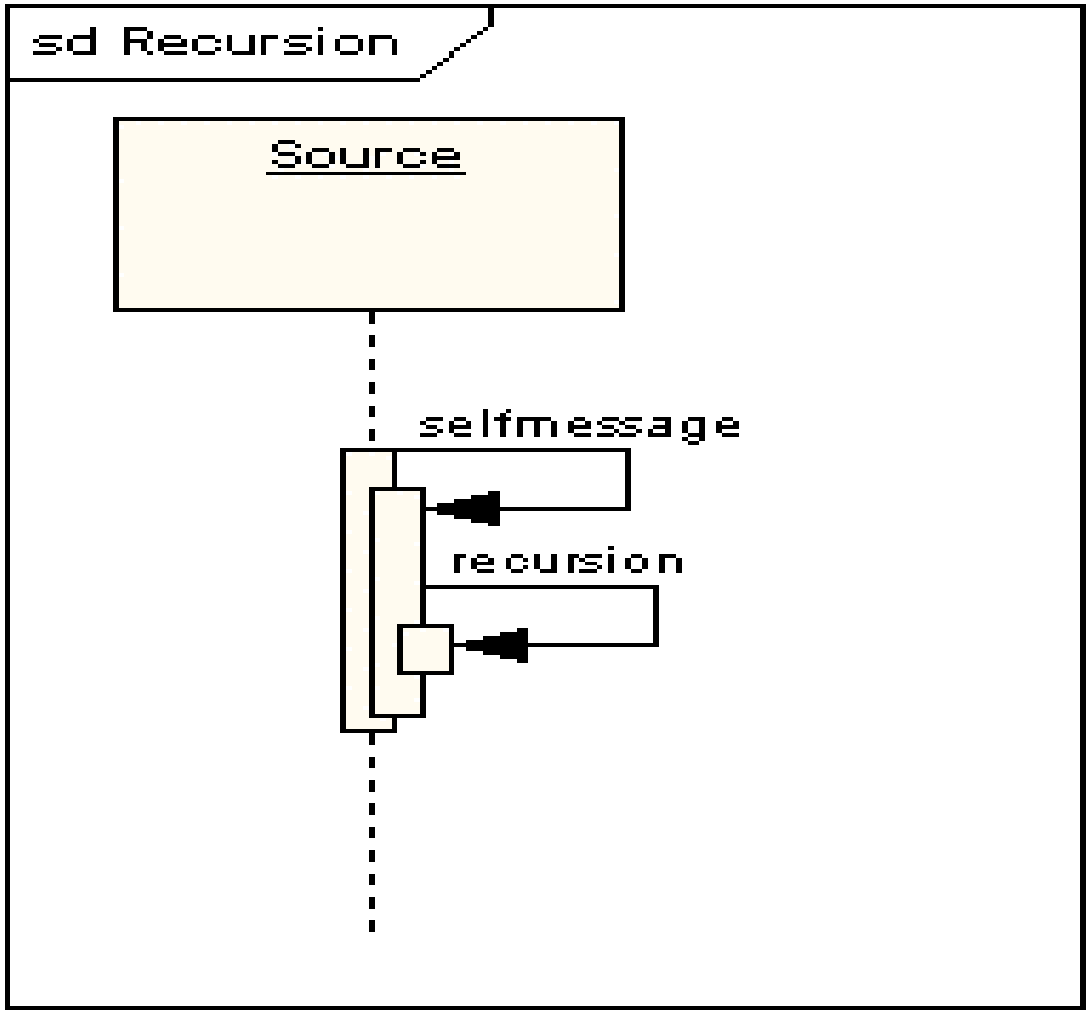
Wykonywanie interakcji (Execution Occurrence)

1. pierwsza wiadomość jest synchroniczna, kompletna i posiada return (wywołanie metody obiektu Target przez obiekt przez Source),
2. druga wiadomość jest asynchroniczna (wywołanie metody obiektu Target przez obiekt przez Source),
3. trzecia wiadomość jest asynchroniczną wiadomością typu return (przerywana linia – return metody asynchronicznej obiektu Target).



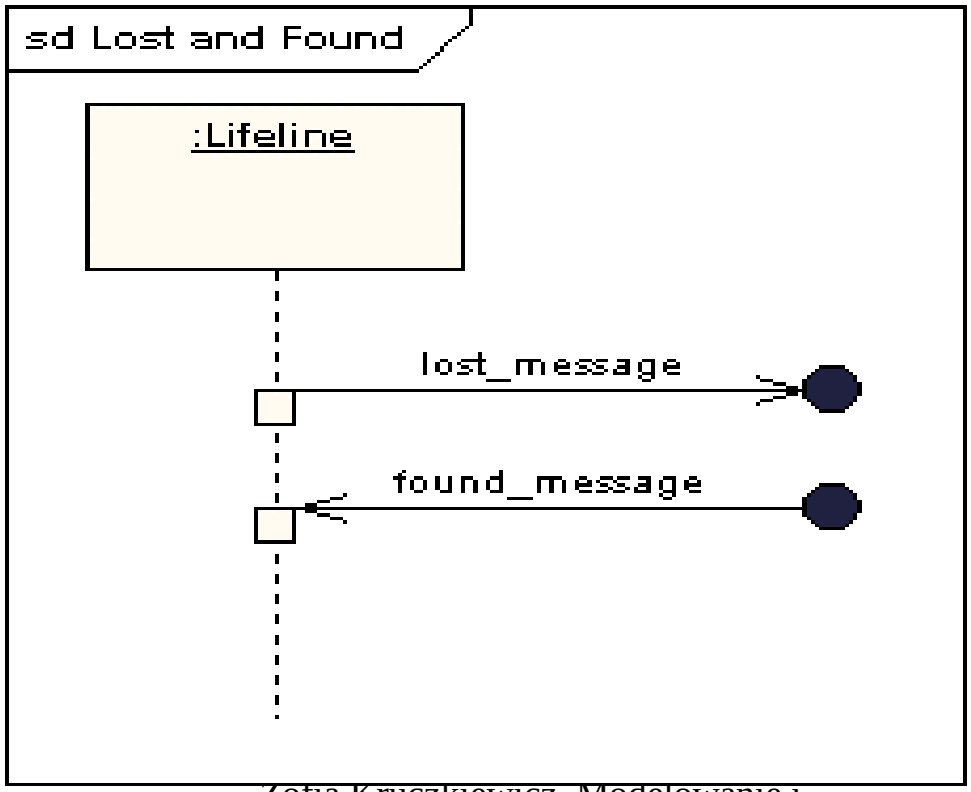
Własne wiadomości (Self Message)

Własne wiadomości reprezentują rekursywne wywoływanie operacji albo jedna operacja wywołuje inną operację należącą do tego samego obiektu.



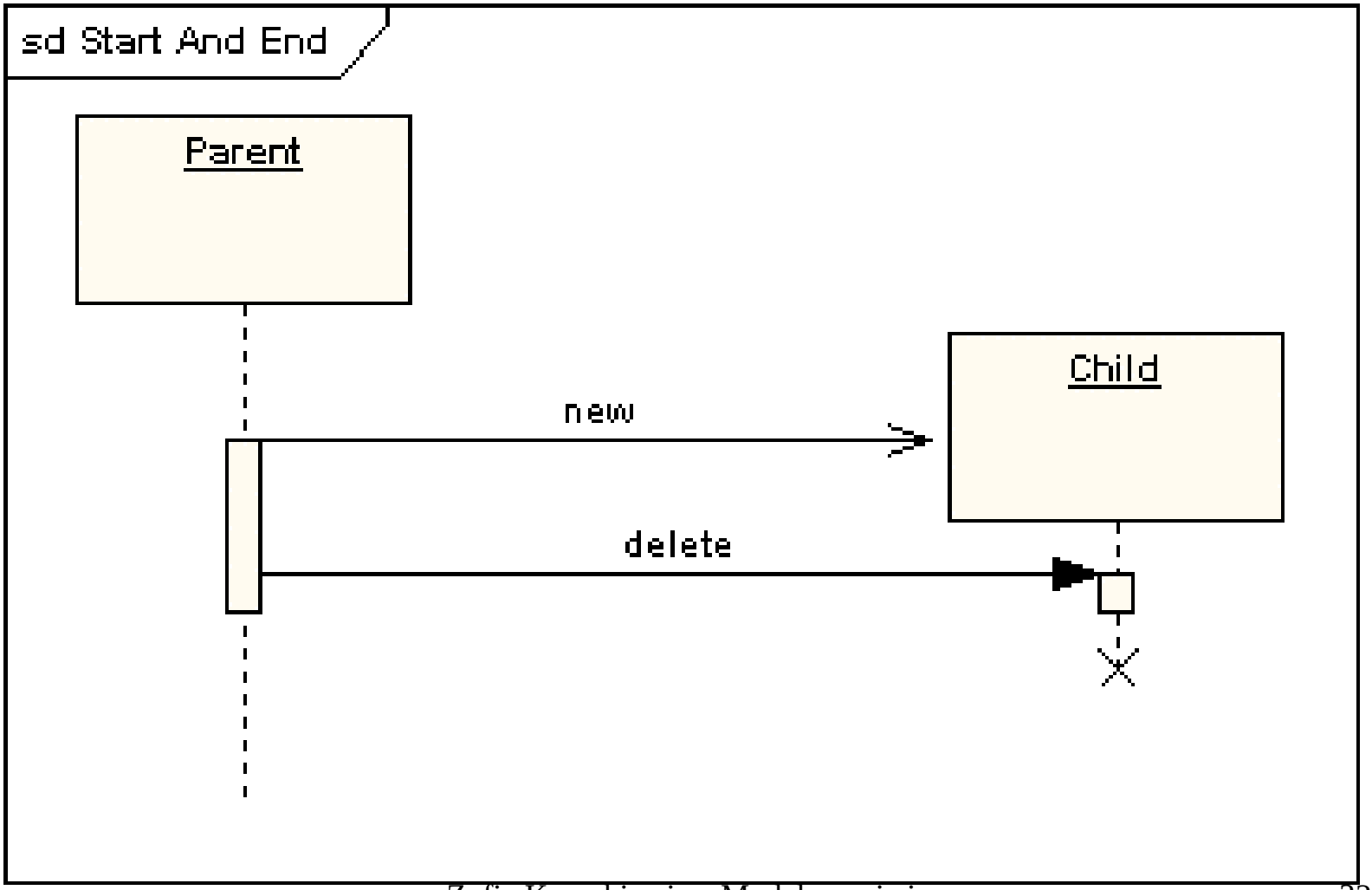
Zgubione i znalezione wiadomości
(Lost and Found Messages)

- **Zgubione wiadomości** są wysłane i nie docierają do obiektu docelowego lub nie są pokazane na bieżącym diagramie.
- **Znalezione wiadomości** docierają od nieznanego nadawcy albo od nadawcy, który nie jest pokazany na bieżącym diagramie.



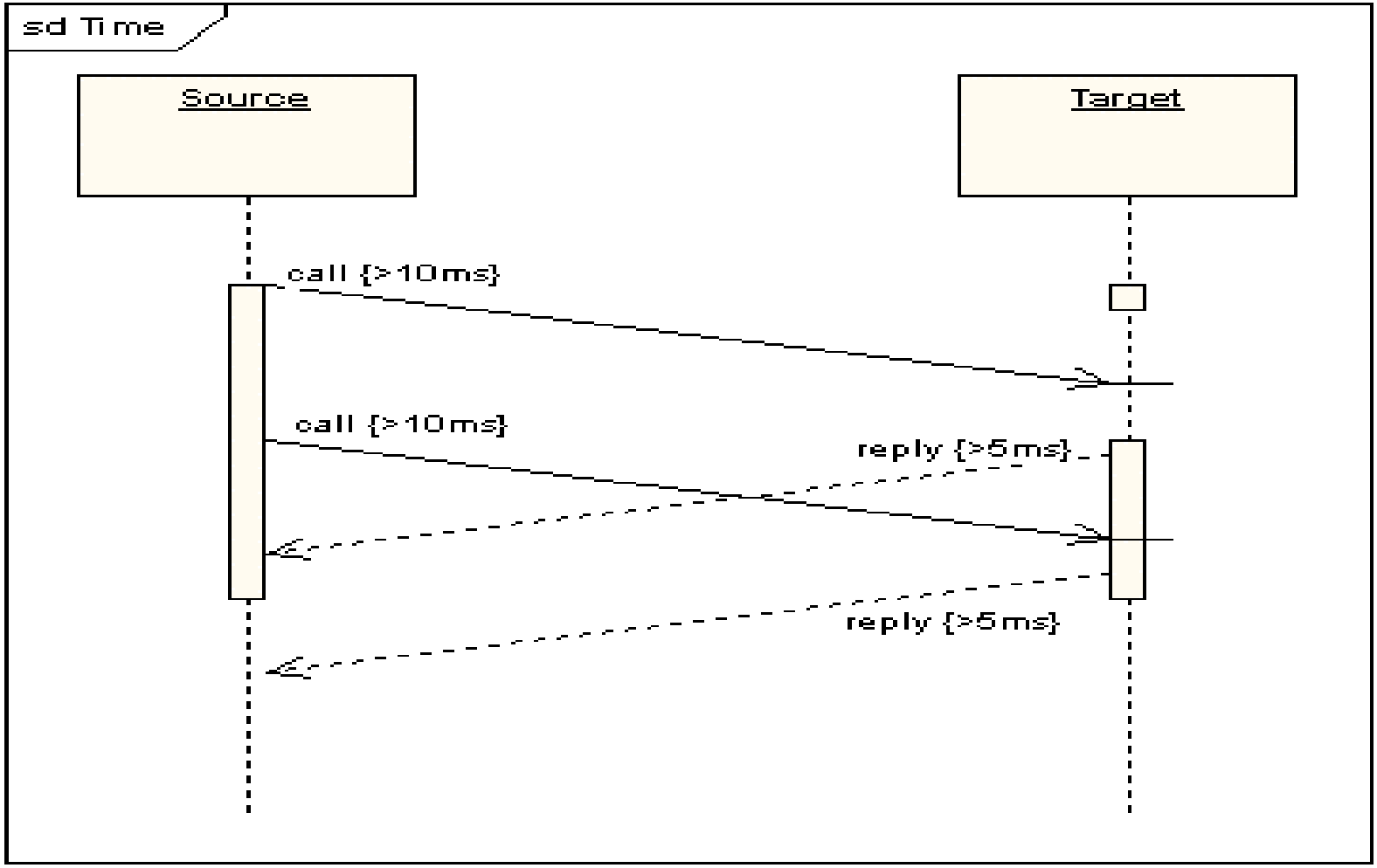
Start linii życia i jej koniec (Lifeline Start and End)

Oznacza to tworzenie i usuwanie obiektu (typu Create Message)



Ograniczenia czasowe (Duration and Time Constraints)

Domyślnie, wiadomość jest poziomą linią. W przypadku, gdy należy ukazać opóźnienia czasu wynikające z czasu podjętych akcji przez obiekt po otrzymaniu wiadomości, wprowadza się ukośne linie wiadomości.

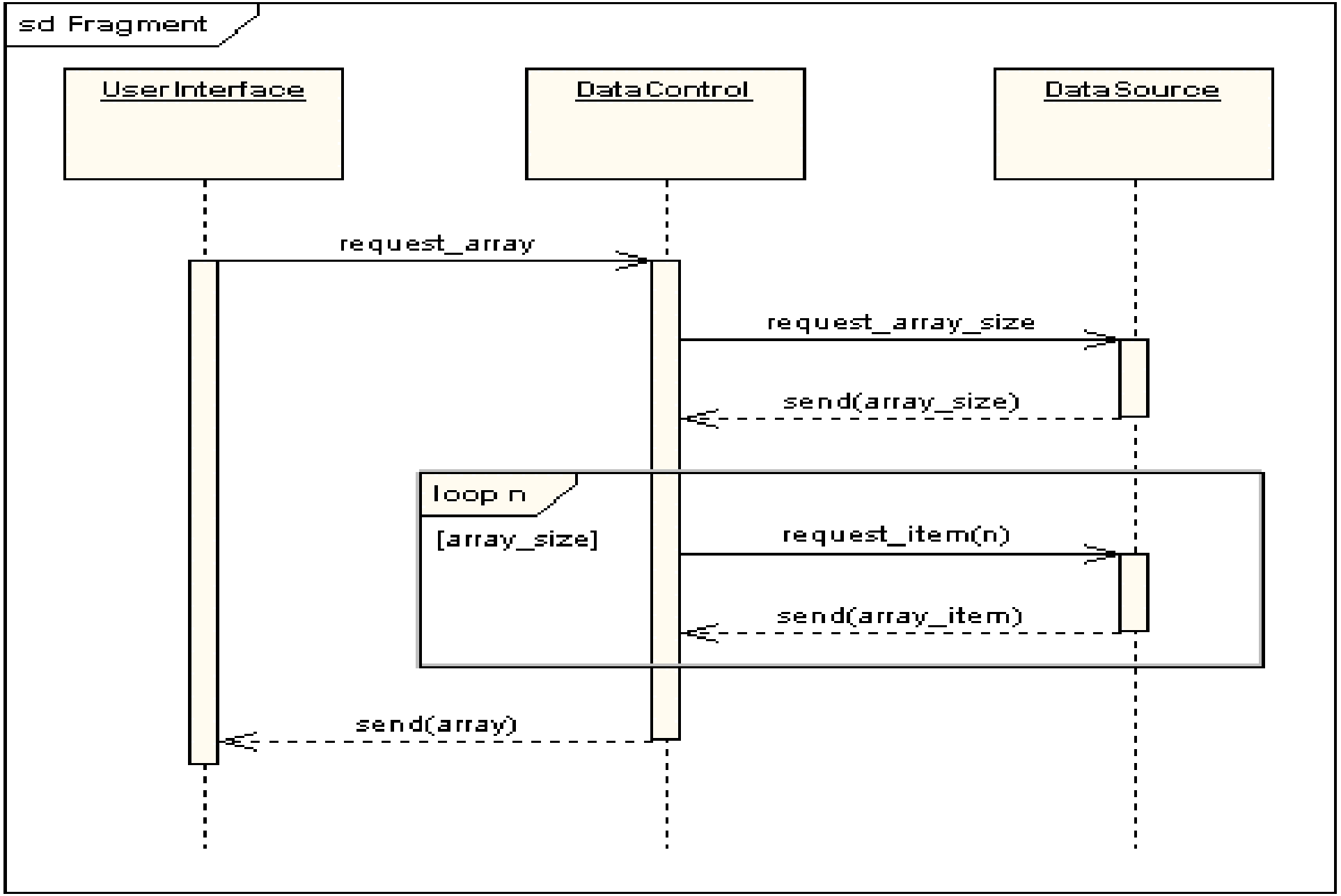


Złożone modelowanie sekwencji wiadomości

Fragmenty ujęte w ramki umożliwiają:

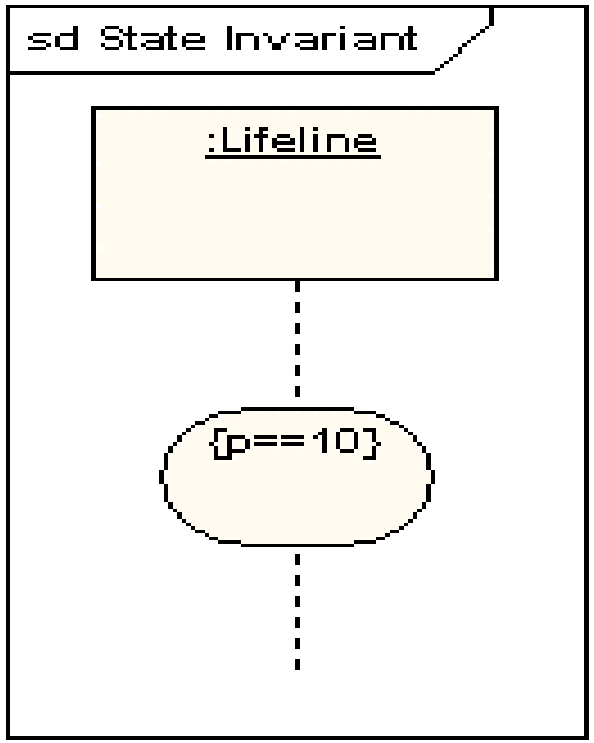
1. **fragmenty alternatywne** (oznaczone "alt") modelują konstrukcje **if…then…else**
2. **fragmenty opcjonalne** (oznaczone "opt") modelują konstrukcje **switch**.
3. **fragment Break** modeluje alternatywną sekwencję zdarzeń dla pozostałej części diagramu.
4. **fragment równoległy** (oznaczony "par") modeluje proces równoległy.
5. **słaba sekwencja** (oznaczona "seq") zamyka pewna liczbę sekwencji, w której wszystkie wiadomości muszą być wykonane przed rozpoczęciem innych wiadomości z innych fragmentów, z wyjątkiem tych wiadomości, które nie dzielą linii życia oznaczonego fragmentu.
6. **dokładna sekwencja** (oznaczona jako "strict") zamyka wiadomości, które muszą być wykonane w określonej kolejności
7. **fragment negatywny** (oznaczony "neg") zamyka pewną liczbę niewłaściwych wiadomości
8. **fragment krytyczny** (oznaczony jako „"critical") zamyka sekcję krytyczną.
9. **fragment ignorowany** (oznaczony jako "ignored") deklaruje wiadomość/ci nieistotne; określa **fragment**, w którym wszystkie wiadomości powinny być ignorowane.
10. **fragment asercji** (oznaczony "assert") eliminuje wszystkie sekwencje wiadomości, które są objęte danym operatorem, jeśli jego wynik jest fałszywy
11. **pętla** (oznaczony "loop") oznacza powtarzanie interakcji w wybranym fragmencie.

Pętla Wykonanie w pętli fragmentu diagramu sekwencji



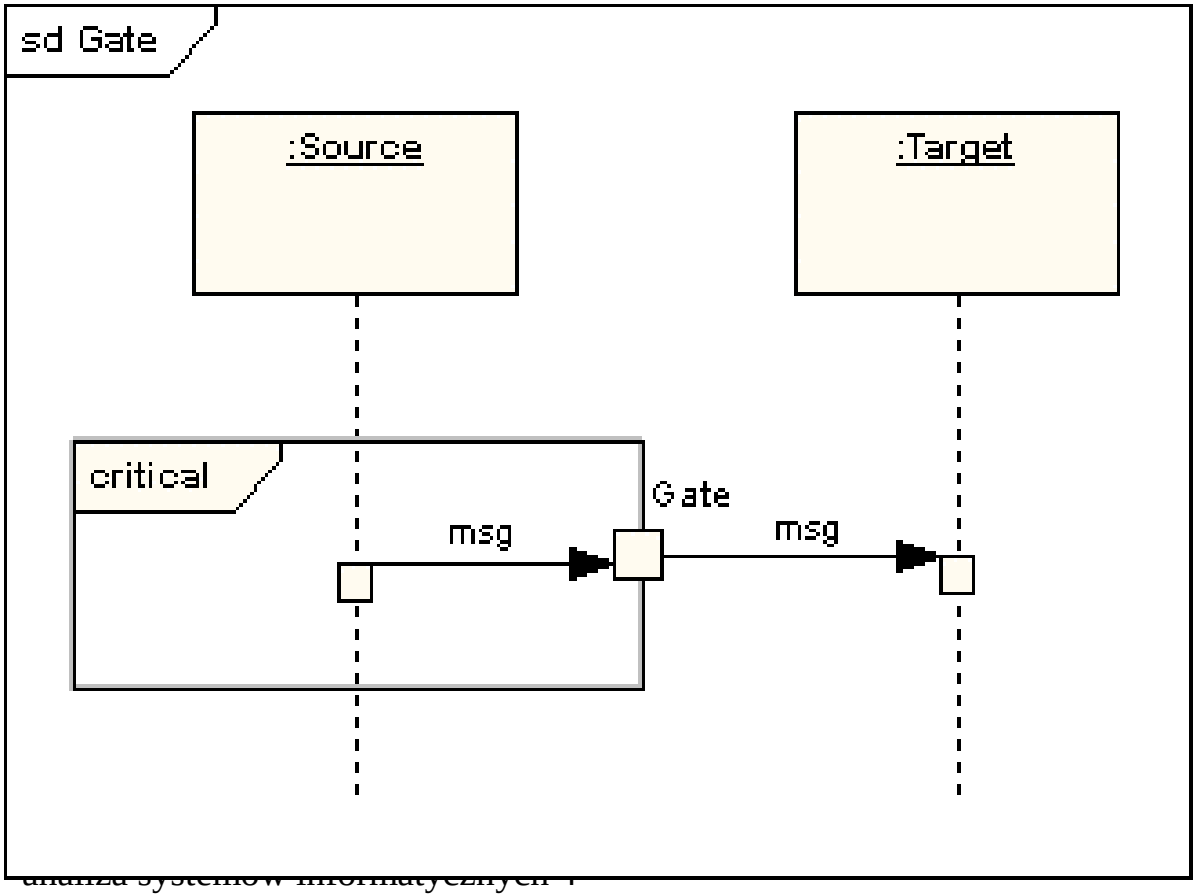
Stan niezmienny lub ciągły (State Invariant /Continuations)

- **Stan niezmienny** jest oznaczany symbolem prostokąta z zaokrąglonymi wierzchołkami.
- **Stany ciągłe** są oznaczone takim samym symbolem, obejmującym kilka linii życia



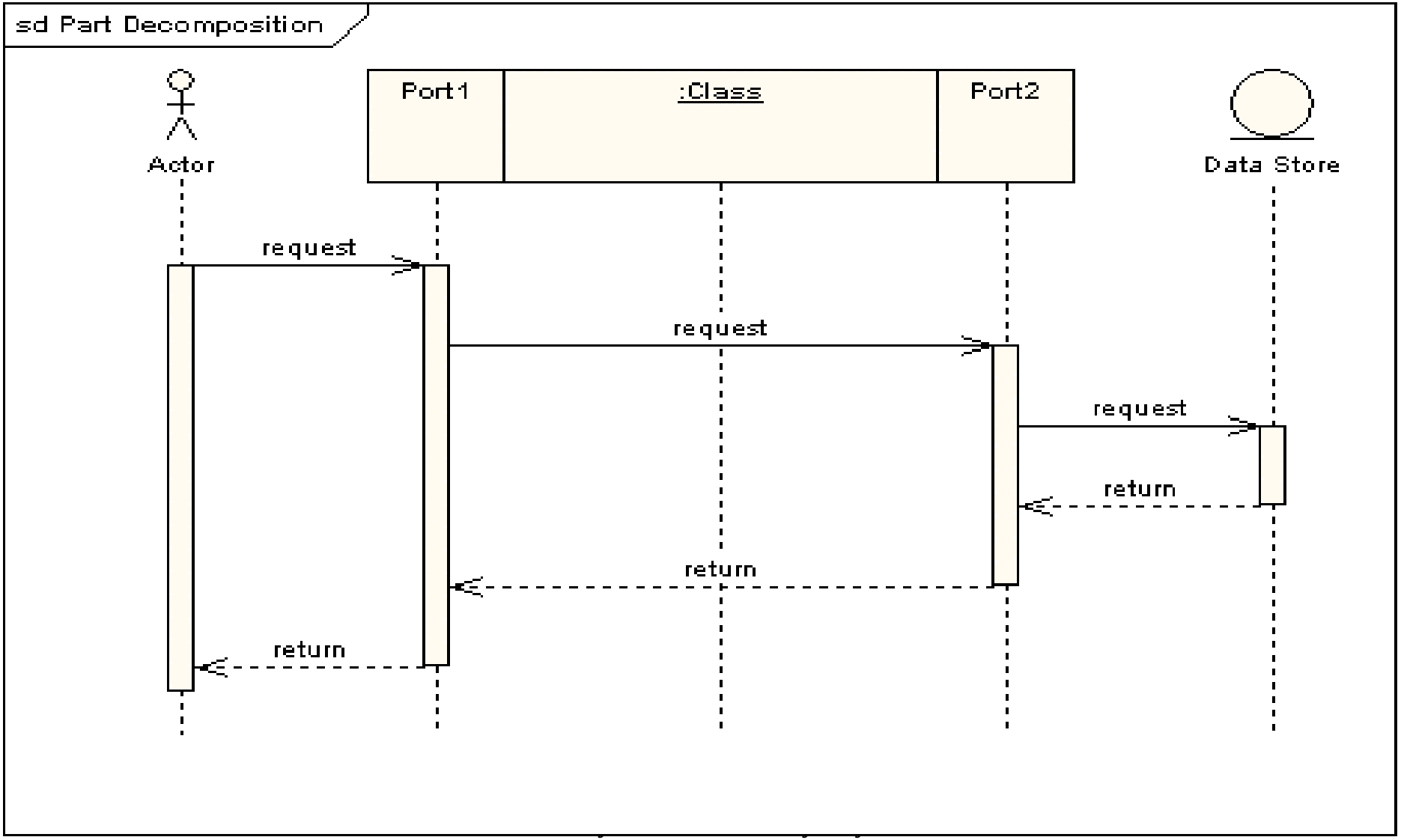
Brama (Gate)

Oznacza przekazywanie wiadomości na zewnątrz między fragmentem i pozostałą częścią diagramu (linie życia, inne fragmenty)



Dekompozycja (Part Decomposition)

Obiekt ma więcej niż jedną linę życia (np. typu Class). Pozwala to pokazać zagnieżdżone protokoły przekazywanych wiadomości np. wewnątrz obiektu i na zewnętrz (w przykładzie typu Class)



Tworzenie modelu konceptualnego systemu informatycznego – część 2

1. **Diagramy klas UML**
   http://sparxsystems.com.au/resources/uml2_tutorial/
2. **Diagramy sekwencji UML**
   http://sparxsystems.com.au/resources/uml2_tutorial/
3. **Model konceptualny – model analizy**

Porównanie modelu przypadków użycia z modelem analizy – ogólne właściwości

Model przypadków użycia	***Model analizy (Analysis Model)***
Opis w języku klienta	Opis w języku wykonawcy
Zewnętrzna postać systemu	Wewnętrzna postać systemu
Strukturyzacja za pomocą przypadku użycia czyli struktura postaci zewnętrznej systemu	Strukturyzacja wewnętrznej postaci systemu za pomocą stereotypowych klas i pakietów
Używany głównie jako kontrakt między klientem i wykonawcami, określający co system powinien robić i czego nie powinien robić	Używany przez wykonawców do zrozumienia jak system powinien wykonany podczas projektowania i implementacji
Może zawierać redundancję, sprzeczności	Nie może zawierać redundancji i sprzeczności
Przedstawia funkcjonalność systemu, dołączając architekturę ważnej funkcjonalności	Określa, jak realizować funkcjonalność dołączając architekturę ważnej funkcjonalności; stanowi punkt wejścia do projektowania
Definiuje przypadki użycia analizowane w modelu analizy	Definiuje realizacje przypadków użycia – realizacje reprezentują analizę przypadków użycia z modelu przypadków użycia

Model konceptualny systemu – model analizy

Klasy analizy (*analysis class*) - klasy reprezentują abstrakcyjne atrybuty realizowane jako klasy podczas projektowania i implementacji, powiązania (*relationships*) reprezentują funkcjonalne wymagania odkładając podczas analizy realizację niefunkcjonalnych wymagań przez specjalne oznaczenie tych wymagań dla danej klasy

Produkt	Opis produktu (reprezentowanego w języku UML)
a) klasy typu „Control” **Warstwy: prezentacji biznesowa, integracji**	• **reprezentują koordynację (*coordination*), sekwencje (*sequencing*), transakcje (*transactions*), sterowanie (*control*) innych obiektów** • **często są używane do hermetyzacji sterowania odniesionego do przypadku użycia dla każdej warstwy tzn hermetyzują warstwę biznesową dla warstwy prezentacji oraz wartwę integracji dla warstwy biznesowej;** • **klasy te modelują dynamikę systemu czyli główne akcje (*actions*) i przepływ sterowania (*control flows*) i przekazują działania do klas warstwy prezentacji, biznesowej oraz integracji ;**

b) klasy typu „Entity” - formalnie obiekty realizowane przez system, często przedstawiane jako logiczne struktury danych (*logical data structure*) **Warstwa biznesowa**	• **używane do modelowania informacji o długim okresie istnienia i często niezmiennej (*persistent*);** • **klasy realizowana jako obiekty typu „real-life” lub zdarzenia typu „real-life”;** • **są wyprowadzane z obiektowego modelu biznesowego lub klas modelu dziedziny *** • **mogą zawierać specyfikację złożonego zachowania reprezentowanej informacji**
c) klasy typu „Boundary” **Warstwa klienta**	• klasy te reprezentują abstrakcje: okien, formularzy, interfejsów komunikacyjnych, interfejsów drukarek, sensorów, terminali i API (również nieobiektowych); • jedna klasa odpowiada jednemu użytkownikowi typu aktor • używane do modelowania interakcji między systemem i aktorami czyli użytkownikami (*users*) lub zewnętrznymi systemami;

Analiza realizacji przypadków użycia

1) Diagramy klas (*class diagrams*)	• diagramy klas analizy i ich obiekty przedstawiane jako modele poszczególnych przypadków użycia; • realizacje jednego lub kilku przypadków użycia za pomocą obiektów typu „Boundary” , „Entity” oraz „Control”
2) Diagramy interakcji (*interaction diagrams: sequence diagram, activity diagram, collaboration diagrams*)	• reprezentują sekwencje akcji danego przypadku użycia, kiedy aktor (czyli obiekt typu „boundary”) wyśle wiadomość (*message*) za pośrednictwem systemu; • sterowanie obiektów za pośrednictwem wiadomości modelowane jest za pomocą diagramów współpracy (*collaboration diagrams*) lub diagramów sekwencji (*sequence diagrams*)

3) Wyjaśnienia (*explains*) diagramu współpracy (*collaboration diagram*)	opis tekstowy wyjaśniający działanie diagramu współpracy (*collaboration diagram*)
4) Specjalne wymagania (*special requirements*)	specjalne wymagania są tekstowym opisem, stanowiącym kolekcję wszystkich niefunkcjonalnych wymagań (*nonfunctional requirements*) dotyczącej realizacji danego przypadku użycia; mogą być wyprowadzone z modelu wymagań lub uzupełniane nowymi wymaganiami

5) Pakiety analizy (*analysis package*) – organizacja produktów analizy	• reprezentują separację produktów analizy; • bazują na funkcjonalnych wymaganiach DP (aplikacji lub biznesu); • odpowiadają podsystemom realizowanym na etapie projektowania
6) Pakiety usług (*service packages*)	• usługi realizowane przez akcje poszczególnych przypadków użycia na żądanie klienta systemu (*actor*); • mogą być wieloużywalne (*reusable*); • realizacje tych samych przypadków użycia mogą wystąpić w różnych pakietach usług
7) Opis architektury (*architecture description*)	• opis architektury systemu z punktu widzenia modelu analizy; • opis dekompozycji modelu w pakiety analizy (*analysis packages*) i ich zależności; • opis ważnych i krytycznych realizacji przypadków użycia;

Porównanie modelu analizy z modelem projektowym – ogólne właściwości

Model analizy (*Analysis model*)	**Model projektowania (*Design model*)**
Model konceptualny, ponieważ jest abstrakcją systemu i jest niezależny od implementacji	Model fizyczny, ponieważ jest zorientowany na implementację systemu
Można go zastosować w różnych projektach	Specyficzny dla konkretnej implementacji
Trzy stereotypy w postaci klas typu: **„Control", „Entity", „Boundary"**	Pewna liczba fizycznych stereotypów klas zależnych od języka implementacji
Mniej formalna postać modelu	Bardziej formalna postać modelu
Mniejszy koszt tworzenia modelu (1:5)	Większy koszt tworzenia modelu (5:1)
Kilka poziomów (*layers*) modelowania	Wiele poziomów modelowania
Dynamiczny, lecz niewystarczająco skupiony na sekwencjach akcji (*sequence*)	Dynamiczny i bardziej skupiony na sekwencjach akcji
Dane wejściowe do tworzenia modelu projektowego	Reprezentuje projekt systemu i jego architektury
Głównie realizowany za pomocą „worhshops"	Głównie realizowany za pomocą „wizualnego" programowania w systemach wspomagania tworzenia oprogramowania
Jest realizowany tylko przez pewien okres podczas cyklu tworzenia oprogramowania (*software life cycle*)	Realizowany przez cały cykl tworzenia oprogramowania
Definiuje strukturę, która stanowi podstawową formę do tworzenia systemu (shaping system), a szczególnie modelu projektowego	Realizuje system w oparciu o model analizy tak dokładnie, jak tylko to możliwe