

# Inżynieria Oprogramowania wprowadzenie

Dr hab. inż. Ilona Bluemke

# Plan wykładu

---

- Powstanie inżynierii oprogramowania
- Cele i zadania inżynierii oprogramowania
- Czym zajmuje się inżynieria oprogramowania
- Oprogramowanie wysokiej jakości

# Powstanie inżynierii oprogramowania - 1

- Koniec lat sześćdziesiątych - tzw. kryzys oprogramowania.
- Wiele realizowanych wówczas projektów kończyło się fiaskiem, a ceny realizowanego wówczas oprogramowania rosły szybko (około 12% na rok) przy zmniejszających się cenach sprzętu.

# Powstanie inżynierii oprogramowania - 2

Przyczyny upadku wielu projektów to:

- duża złożoność systemów,
- nowe dziedziny zastosowań, niepowtarzalność przedsięwzięć,
- niesystematyczny proces budowy oprogramowania,
- trudności w ocenie stopnia zaawansowania prac programistycznych,
- pozorna łatwość wytwarzania i dokonywania poprawek (np. 100 linii w 1 dzień, 1000 linii w 10 dni ?).

# Powstanie inżynierii oprogramowania - 3

- Ulepszenia w procesie produkcji oprogramowania mogą przynieść duże korzyści ekonomiczne.
- Pomysłów na poprawę procesu produkcji szukano w innych naukach inżynierijnych np. u inżynierów mechaników (ang. mechanical engineering) czy inżynierów budowy dróg i mostów (ang. civil engineering).
- Powstająca dziedzina, poprzez analogię została nazwana ang. **software engineering, inżynieria oprogramowania**.

# Cel inżynierii oprogramowania

---

- poszukiwanie i wdrażanie metod oraz technik produkcji programów o wysokiej jakości,
- produkcja w sposób najbardziej efektywny.

# Oprogramowanie wysokiej jakości

- działa zgodnie z wymaganiami określonymi przez specyfikację,
- jest tak szybkie, wydajne i funkcjonalne jak oczekuje użytkownik,
- daje się łatwo pielęgnować (korekcja i modyfikacja),
- posiada pełną dokumentację użytkową i projektową, która umożliwia spełnienie poprzednich postulatów.

# Inżynieria oprogramowania

- dotyczy oprogramowania tworzonego przez zespoły,
- jej zasady są wykorzystywane w rozwoju systemu,
- zawiera aspekty techniczne i nietechniczne,
- występują w niej podejścia formalne i praktyczne.

# Co oferuje inżynieria oprogramowania

- techniki i narzędzia ułatwiające pracę nad złożonymi systemami,
- systematyzację procesu produkcji oprogramowania, tak by ułatwić jego monitorowanie i planowanie,
- metody wspomagające analizę nieznanych problemów i ułatwiające wykorzystywanie wcześniejszych doświadczeń.

# Inżynieria oprogramowania

## zajmuje się :

- sposobami prowadzenia przedsięwzięć informatycznych,
- technikami szacowania kosztów, harmonogramowania,
- metodami analizy i projektowania systemów,
- technikami zwiększania niezawodności oprogramowania,

# Inżynieria oprogramowania zajmuje się – 2:

- sposobami testowania systemów, szacowania niezawodności,
- sposobami przygotowywania dokumentacji technicznej i użytkowej,
- procedurami kontroli jakości,
- technikami pracy zespołowej.

# Jakość oprogramowania

Ocena jakości oprogramowania jest sprawą subiektywną.

Model Mc Call'a dzieli kryteria oceny jakości na grupy związane:

- ze sposobem działania
- z możliwością zmian i poprawek
- z mobilnością oprogramowania.

# Kryteria związane ze sposobem działania

- przyjazność - dotyczy projektu interfejsu
- bezpieczeństwo - kontrola uprawnień dostępu,
- wydajność,
- poprawność - stopień realizacji wymagań,
- kompletność i logiczność implementacji, zgodność działania ze specyfikacją,
- niezawodność - odporność na błędy.

# możliwość wprowadzenia zmian i poprawek

- pielęgnowalność - stopień przystosowania do poprawienia, modyfikacji, rozszerzania, adaptowania,
- elastyczność - możliwości rozbudowywania oprogramowania o nowe funkcje oraz uniwersalność zaimplementowanych rozwiązań,
- testowalność

# mobilność oprogramowania

- przenośność - zdolność do łatwego uruchamiania na innych systemach,
- uniwersalność - odnosi się do możliwości wykorzystania istniejącego oprogramowania lub jego fragmentów do konstrukcji innych systemów,
- otwartość - stopień przystosowania programu do współpracy lub wymiany informacji z innymi systemami komputerowymi.

# problem

- Problem - osiągnięcia optimum
- **Co optymalizować** powinno być ustalone z klientem - np. lepszy interfejs z użytkownikiem to spadek efektywności.

# Jakość

- produktu
- procesu wytwarzania

**Technologie**

**Jakość  
procesu**

**Jakość  
produktu**

**Ludzie**

**Koszt, czas  
harmonogram**

# niezawodność a efektywność oprogramowania

- Szybszy, tańszy sprzęt, ważniejsza wygoda użytkownika
- Zawodne oprogramowanie będzie unikane
- Są zastosowania gdzie koszt błędu systemu może znacznie przekraczać koszt samego systemu. Koszt ludzkie - nie do zaakceptowania (safety -critical systems)
- System zawodny trudno jest ulepszyć, poprawić.

# niezawodność a efektywność oprogramowania -2

- System niezawodny można stroić, lokalizować przyczyny opóźnień
- Nieefektywność - program wykonuje się dłużej, skutki można przewidzieć
- Zawodność - skutki mogą być trudne do przewidzenia, błędy w projekcie mogą prowadzić do katastrofy
- Zawodne systemy mogą powodować utratę danych

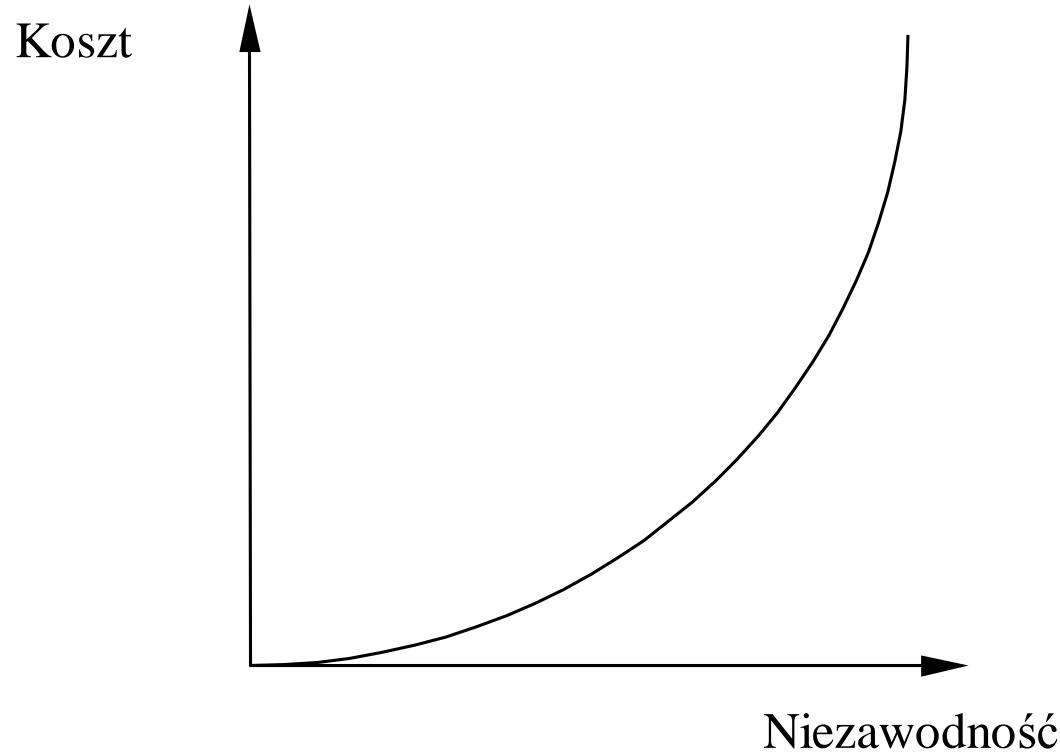
# Niezawodność zależy od:

- poprawności projektu
- poprawności odwzorowania projektu w implementację
- poprawności elementów i ich złożenia

Miary niezawodności oprogramowania - inne niż sprzętu

Kluczem do niezawodności oprogramowania jest specyfikacja

# Koszt a niezawodność



Rys. 1.1 Koszt oprogramowania a niezawodność

# Przykładowe pytania

- Czym zajmuje się inżynieria oprogramowania ?
- Jakie są cechy oprogramowania wysokiej jakości ?
- Jakie można stosować kryteria oceny oprogramowania ?
- Jakie cechy oprogramowania są związane z jego działaniem ?
- Co oferuje inżynieria oprogramowania ?
- Dlaczego ważniejsza jest niezawodność oprogramowania, niż efektywność ?

# Analiza obiektowa wstęp do UML

Dr hab. inż. Ilona Bluemke

# Plan wykładu

---

- Podstawy analizy obiektowej
- Historia powstania UML
- Perspektywy widzenia systemu przy projektowaniu w UML

# Analiza systemu

---

Koncentruje się na zrozumieniu systemu oraz  
znalezieniu odpowiednich rozwiązań.

- V. Weinberg: "Analiza systemu jest badaniem problemów, celów, wymagań, priorytetów i ograniczeń wynikających ze środowiska wraz z szacowaniem kosztów, zysków i wymagań czasowych, w celu wypracowania pierwszych propozycji rozwiązań"

# Analitycy biorą udział w:

---

- analizie wymagań,
- weryfikacji rozwiązań,
- ocenie technicznej projektu wstępnego i szczegółowego,
- walidacji i weryfikacji oprogramowania.
- Określają także wytyczne co do zasad pielęgnowania systemu.

# Analiza obejmuje sfery:

---

- problemu (co ma być wykonane),
- wykonalności (czy realizacja jest możliwa),
- ekonomiczną ( jakie będą koszty i zyski realizacji).

# Zasady, sposoby przy podejściu do złożonego problemu:

---

- **Abstrakcja** - ignorowanie nieistotnych aspektów, skoncentrowanie się na istotnych.
- **Abstrakcja proceduralna** - dowolna operacje dająca określony efekt może być traktowana elementarnie, w rzeczywistości może być realizowana przez operacje niższych poziomów.
- **Abstrakcja danych** - definiowanie typu danych w sensie operacji dotyczących obiektów tego typu, z ograniczeniem takim, że wartości tych obiektów mogą być modyfikowane i odczytywane tylko za pomocą tych operacji.

# Zasady, sposoby przy podejściu do złożonego problemu -2:

---

- **Ukrywanie informacji** - każdy składnik programu powinien ukrywać pojedyncze decyzje projektowe. Interfejs modułu jest tak projektowany aby jak najmniej odsłaniać sposób jego wewnętrznej pracy.
- **Dziedziczenie** - mechanizm wyrażania podobieństwa między klasami, ułatwiający definiowanie klas podobnych do już zdefiniowanych. Opisuje podział na cechy ogólne i szczegółowe, wyrażając atrybuty i usługi w hierarchii klas.
- **Skojarzenia** - łączenie idei.

# Cechy modelu obiektowego

---

- abstrakcja,
- enkapsulacja,
- modularność,
- hierarchia.

# Obiektowo zorientowane projektowanie

---

- Maksymalizuje ukrywanie informacji, może prowadzić do systemów bardzo spójnych, o mniejszym stopniu zależności elementów niż podejście funkcjonalne.
- System widziany jako **zbiór współdziałających obiektów.**

# Zalety metody obiektowej

---

- wyeliminowane wspólne obszary danych (komunikacja poprzez przekazywanie komunikatów),
- obiekty są łatwo modyfikowalne, reprezentacja informacji jest wewnątrz nich, zmiany są lokalne i nie wpływają na inne obiekty,
- decyzje o implementacji sekwencyjnej lub równoległej nie muszą zapadać we wczesnej fazie projektowania.

# Obiekt

---

- ma **indywidualność**
- ma **stan** - zawiera wszystkie statyczne cechy obiektu i dynamicznie się zmieniające wartości tych cech, skumulowane zachowanie obiektu,
- ma **zachowanie** ( zmiany stanów, przekazywanie komunikatów).

# Rola jaką obiekt może pełnić

---

- **Aktor** - aktywny, pracuje, steruje innymi, sam nigdy nie jest sterowany
- **Serwer** - sam innymi nie steruje, dostarcza usługi
- **Agent** - obie role, pracuje na innych, inne na nim

# Analiza zorientowana obiektowo

---

Analiza obiektowa przebiega zazwyczaj w następujących krokach:

- Znajdowanie - identyfikacja obiektów,
- Organizowanie obiektów,
- Opis interakcji,
- Definicja operacji obiektu,
- Definicja wnętrza obiektu.

# Identyfikacja obiektów

---

Ważny **podmiot (rzeczownik)** z dziedziny problemu jest kandydatem na obiekt.

Typy obiektów:

- aktywne/pasywne
- fizyczne/konceptualne
- chwilowe/stałe
- prywatne/publiczne
- część/całość
- ogólne/specyficzne

# Organizowanie obiektów

---

Kryteria organizowania obiektów:

- Jakie są cechy wspólne klas/obiektów - tworzona hierarchia dziedziczenia,
- Które obiekty współpracują ze sobą,
- Które obiekty są częścią innych obiektów,
- Jakie obiekty są zależne od siebie.

# Organizowanie obiektów

---

Opisuje się różne **scenariusze** - przykłady  
użycia systemu (use case).

Z nich wynika które obiekty, i w jaki sposób  
mają się komunikować, czego obiekty  
oczekują po sobie.

Na tej podstawie można określić, **interfejsy**  
obiektów oraz które obiekty są częścią  
innych.

# Definicja operacji i wnętrza obiektu

---

## Definicja operacji obiektu

- Określenie **Co** obiekt ma wykonywać.
- Jeśli operacje są złożone to można identyfikować nowe obiekty.

## Definicja wnętrza obiektu - implementacja

---

# METODY ZORIENTOWANE OBIEKTOWO

---

- OOD Booch 1991
- Object Oriented System Analysis (OOSA)
- Shaer Mellor 1988
- OMT Object Modelling Technique (Rumbaugh et al 1991)
- OOA/OOD Coad & Yourdon 1991
- OOSE (I. Jacobson)
- HOOD 1989 (Hierarchical O-O Design)
- Responsibility Driven Design (Wirfs-Brock et al 1990)
- OORASS (O-O Role Analysis, Synthesis & Structuring)
- Reenskang et al 1990
- OOSD (O-O Structured Design)
- Wasserman et al 1989, 1990
- OSA O-O System Analysis (Embley et al 1992)
- OBA Object Behavior Analysis (Gibson 1990)
- Synthesis Page-Jones & Weiss 1989
- Fussion (HP)
- Merise

# Przyczyny standaryzacji

---

UML - ang. Unified Modelling Language  
Unifikacja metod modelowania została zapoczątkowana przez Booch'a i Rumbaugh.

- 1989 – 10 obiektowo zorientowanych metod projektowania i analizy
- 1994 – 50 metod (najbardziej znane to Booch, Jacobson's OOSE – Object Oriented Software Engineering, Rumbaugh's OMT – Object Modelling Technique, Fussion, Shlaer-Mellor, Coad-Yourdon,

# Historia standaryzacji

---

- X 1994 Rumbaugh, Booch w firmie Rational
- X 1995 Unified Method version 0.8
- VI 1996 Unified Method version 0.9 (Jacobson dołączył do Rational)
- I 1997 UML 1.0 przedłożony do standaryzacji do OMG (Object Management Group)
- IX 1997 UML 1.1 zaakceptowany przez OMG
- VI 1998 UML 1.2
- 1998 UML 1.3
- 2003 UML 1.5
- 2006 UML 2.0
- 2007 UML 2.1

# Unifikacja metod projektowania

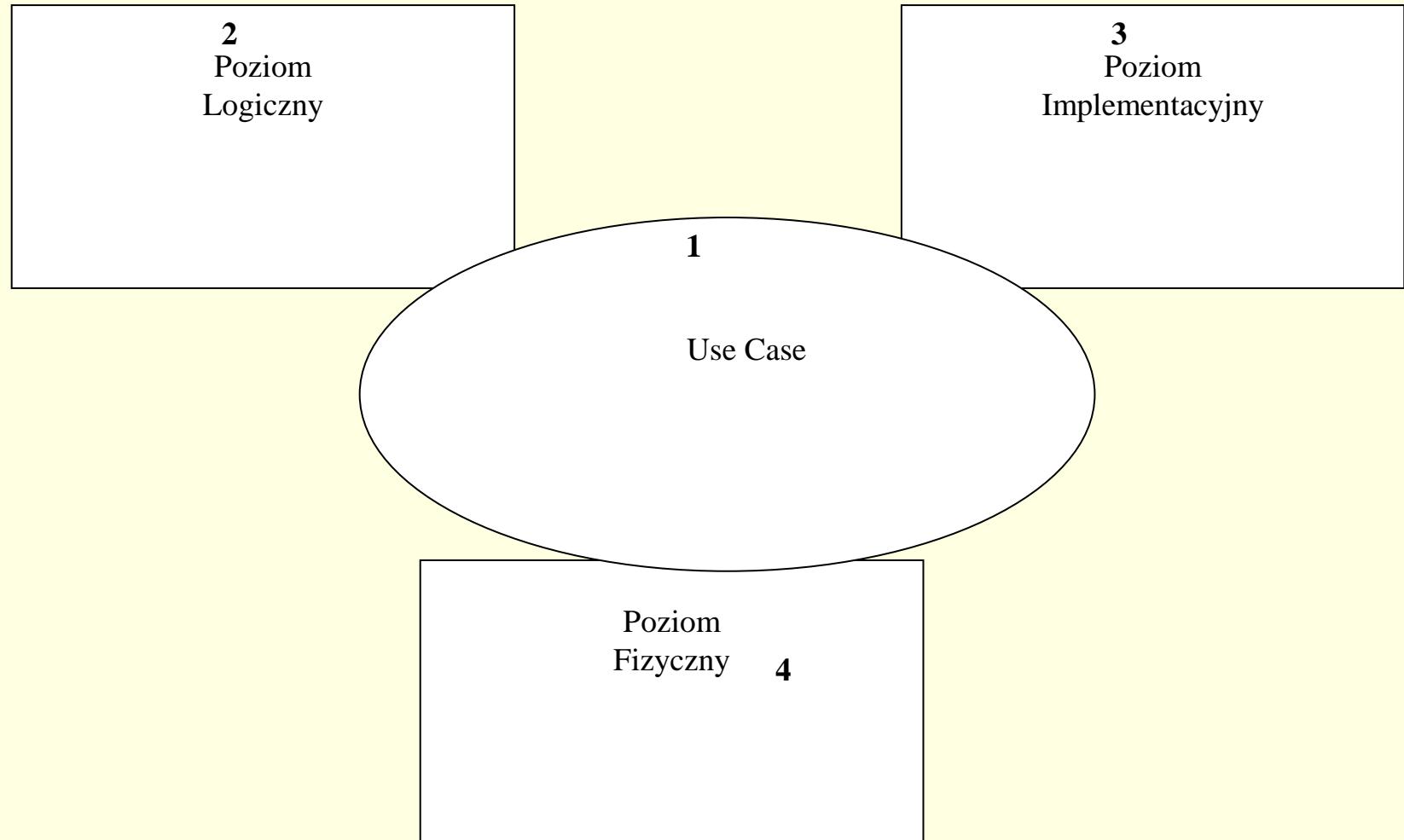
---

Unifikacja metod modelowania została zapoczątkowana przez Jacobsona, Boocha i Rumbaugh.

Podstawowe składniki UML:

- pojęcie podsystemu, kategorie podsystemów - Booch
- przykłady użycia - Use Case - Jacobson
- pojęcie asocjacji - Rumbaugh
- pojedyncze klasy, obiekty – kompozycje - Embley
- opisy operacji, numerowanie komunikatów - Fusion
- diagramy stanów - Harel
- warunki przed i po operacji - Meyer
- dynamiczna klasyfikacja, nacisk na znaczenie zdarzeń - Odell
- cykle życia obiektów – Shlaer/Mellor

# PERSPEKTYWY



# Model Use Case

---

- Przedstawia system z punktu widzenia użytkownika (różnych klas użytkowników systemu).
- Modeluje zachowanie systemu w odpowiedzi na polecenia użytkownika.

Na tym etapie tworzone są diagramy „Use Case”  
Posłużą one do następnych etapów projektowania  
oraz do końcowego testowania systemu pod kątem  
spełniania wymogów użytkowników.

Określa **CO** system robi

# Model logiczny

---

Przedstawia system w postaci klas, powiązań i interakcji między nimi, zachowań obiektów należących do tych klas oraz sekwencji działań systemu.

Na tym etapie tworzy się następujące diagramy:

- klas, obiektów
- sekwencji (interakcji)
- współpracy
- przejść stanów

Określa **CO jest** w systemie, **JAK** system działa

# Model implementacyjny i wdrożeniowy

---

## Model implementacyjny

- Przedstawia system jako moduły, podsystemy, zadania. Na tym etapie powstaje diagram komponentów.

## Model wdrożeniowy (Deployment )

- Modeluje fizyczne rozmieszczenie modułów systemu na komputerach. Uwzględnia wymagania sprzętowe, obszary krytyczne.
- Na tym etapie tworzymy diagramy rozmieszczenia (deployment).

# Inżynieria Oprogramowania modele procesu produkcji

Dr hab. inż. Ilona Bluemke

# Plan wykładu

---

- Model wodospadowy
- Model ewolucyjny
- Prototypowanie
- Formalne transformacje
- Metoda iteracyjna
- Metoda reuse
- Model spiralny
- Czynniki nie-techniczne w inżynierii oprogramowania

# Model wodospadowy (waterfall)

## (Royce 1970)

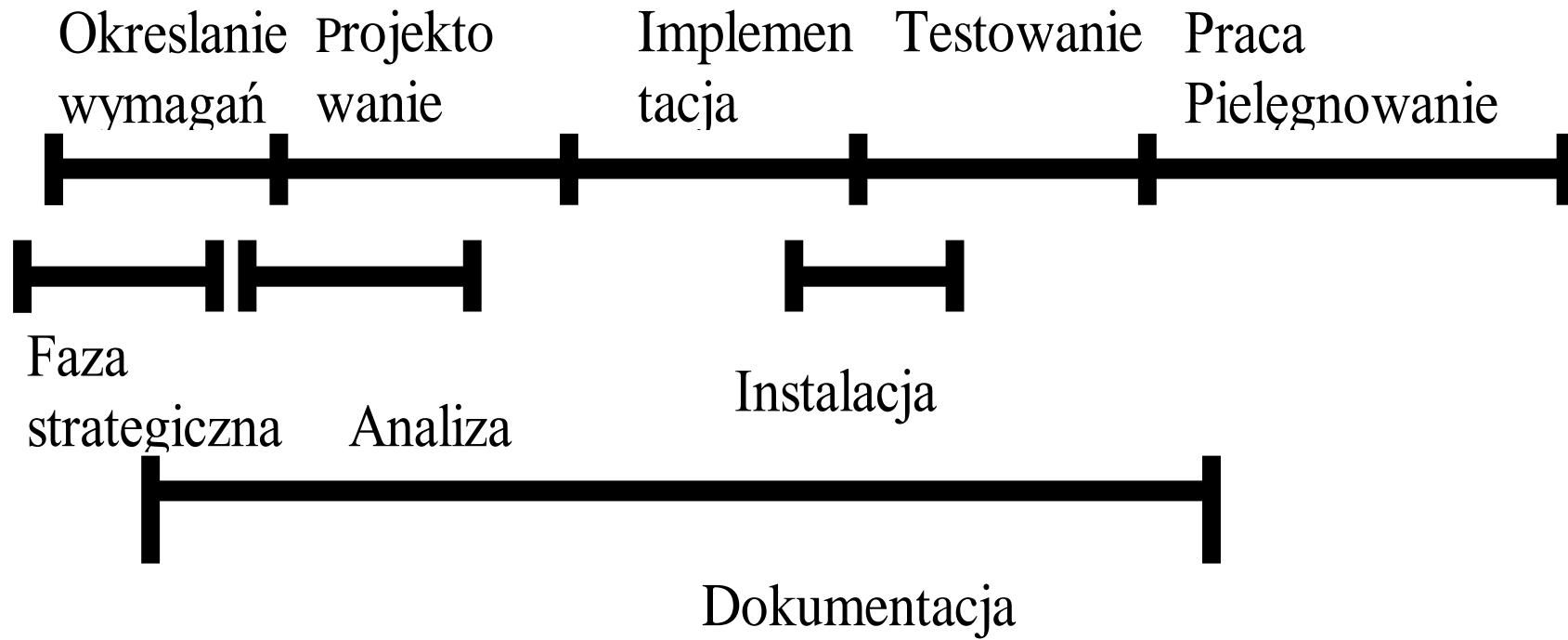
Inaczej kaskadowy, liniowy

Analogia do prowadzenia prac inżynierycznych.

Wprowadzono fazy:

- **specyfikacji wymagań**
- **projektowania oprogramowania (design)**
- **implementacji (implementation, coding)**
- **testowania**
- **użytkowania i pielęgnowania**

# Nakładanie się faz



# Estymacja kosztów

	Wymagania/ projekt	Implement.	testy
s. sterow.	46	20	34
s. operac.	33	17	50
s. nauk.	44	26	30
s. biznes.	44	28	28

# Zalety i wady modelu wodospadowego

## Zalety:

- łatwość zarządzania przedsięwzięciem (planowanie, harmonogramowanie, monitorowanie)
- narzucenie kolejności wykonywania prac

## Wady modelu:

- narzucenie kolejności wykonywania prac
- wysoki koszt błędów popełnionych we wczesnych fazach
- długa przerwa w kontaktach z klientem

# Rezultaty faz modelu wodospadowego - 1

- Analiza wymagań - studium wykonalności, „zgrubne” wymagania
- Definicja wymagań - dokument opisujący wymagania
- Specyfikacja systemu - funkcjonalna specyfikacja systemu, plan testów akceptacyjnych, szkic podręcznika użytkownika
- Projektowanie architektury - specyfikacja architektury, testy systemowe
- Projektowanie interfejsów - specyfikacja interfejsów, testy integracyjne

# Rezultaty faz modelu wodospadowego - 2

- Projektowanie jednostek - projekt szczegółowy, testy jednostkowe
- Kodowanie - kod programu
- Testowanie jednostek - raport testowania
- Testowanie modułów - raport testowania
- Testowanie integracyjne - raport testowania integracyjnego, podręcznik użytkownika
- Testowanie systemowe - raport testowania
- Testowanie akceptacyjne - system i dokumentacja

## **Weryfikacja**

- Czy właściwie budujemy produkt ?
- Czy spełnia wymagania ?

## **Walidacja**

- Czy budujemy właściwy produkt ?
- Czy funkcje produktu są takie jak klient naprawdę oczekiwał ?

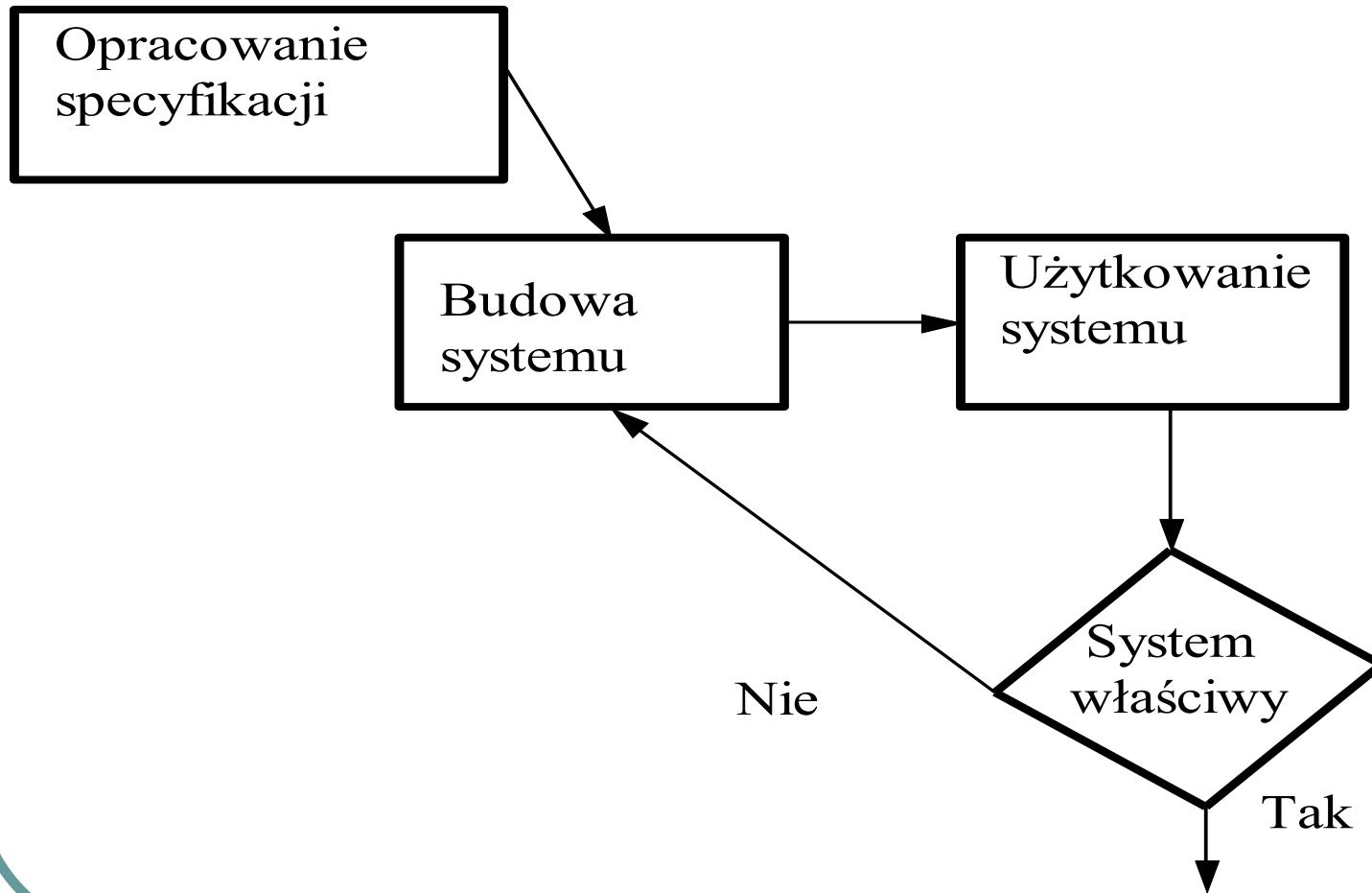
# Model ewolucyjny

---

(exploratory programming), odkrywczy

Stosowany w przypadkach, gdy określenie dokładnych wymagań klienta nie jest możliwe (np. systemy sztucznej inteligencji).

# Model ewolucyjny - 2



# Zalety i wady m. ewolucyjnego

## Zalety:

- możliwość stosowania nawet w przypadkach kłopotów z określeniem wymagań klienta.

## Wady:

- struktura systemu b. zagmatwana
- konieczność bardzo szybkiej produkcji stąd użycie języków takich jak Lisp, Prolog, środowisk produkcji
- nie ma weryfikacji (sprawdzenia czy spełnia wymogi)
- nie gwarantuje możliwości pielęgnowania systemu

# Prototypowanie

---

Fazy:

- ogólne określenie wymagań
- opracowanie szybko działającego prototypu
- walidacja prototypu przez klienta
- określenie szczegółowych wymagań
- opracowanie pełnego systemu

# Prototyp – cele, możliwości

Celem prototypu jest wykrycie:

- trudnych usług
- braków w specyfikacji
- nieporozumień między klientem a projektantami

Prototyp pozwala na:

- demonstrację pracującego systemu
- daje możliwości szkolenia zanim zostanie zbudowany pełen system

# Budowa prototypu

---

- programowanie ewolucyjne
- wykorzystanie gotowych komponentów
- niepełna realizacja
- języki wysokiego poziomu
- generatory np. interfejsów użytkownika

# Formalne transformacje

Wymagania wobec systemu są zapisywane w języku formalnym. Podlegają one automatycznym przekształceniom do programu.

## Zaleta:

- wysoka niezawodność (brak błędów przy transformacjach)

## Wady:

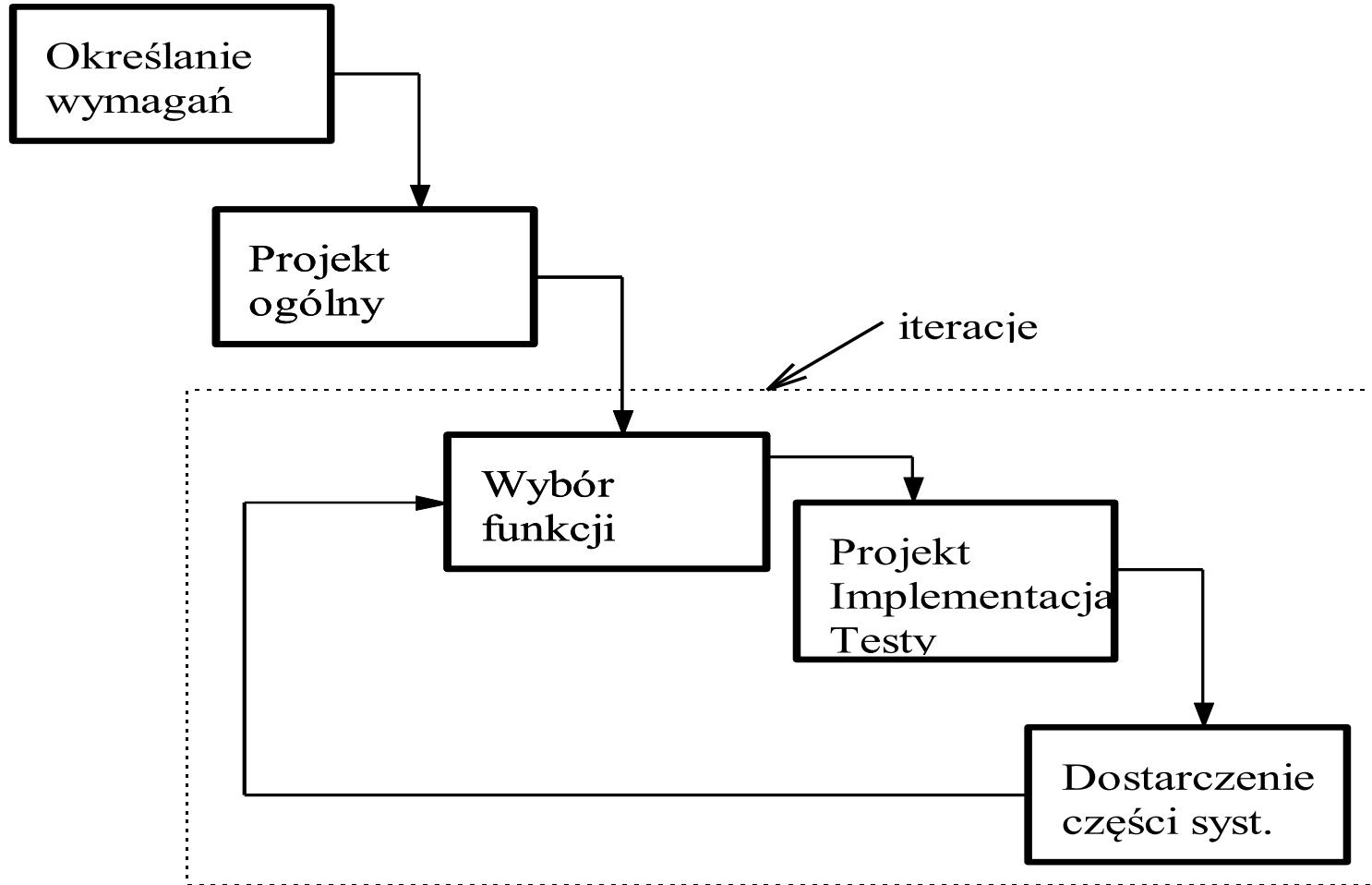
- trudności formalnego specyfikowania
- mała efektywność kodu

# Formalne specyfikacje

---

- Stosuje się ją do produkcji systemów wymagających dużej niezawodności, bezpieczeństwa.
- Formalne specyfikacje wymagają dużych umiejętności zespołu.
- Nie wszystkie wymagania można formalnie wyspecyfikować np. nie można formalnie wyspecyfikować interfejsu użytkownika.

# Realizacja przyrostowa (incremental development)



# Kryteria wyboru funkcji do realizacji

---

- Priorytet dla klienta
- Łatwość realizacji
- Przydatność dla dalszych iteracji

# Wady i zalety metody iteracyjnej

---

## Zalety:

- częsty kontakt z klientem
- możliwość wczesnego wykorzystywania części systemu

## Wady:

- - dodatkowy koszt związany z realizacją fragmentów systemu (pisanie szkieletów modułów)

# Montaż z gotowych elementów (off-shell programming, reuse )

W montażu z gotowych elementów (ang. reuse, off-shell programming) korzysta się z gotowych, dostępnych komponentów i tworzy się kod integrujący je.

COTS ang. Commercial Off The Shelf

Stosowanie :

- bibliotek
- języków czwartej generacji
- pełnych aplikacji

# Reuse - Proces produkcji - 1

- Specyfikacja wymagań.
- Analiza komponentów - poszukiwanie komponentów spełniających funkcje systemu, zwykle komponenty spełniają tylko część wymagań.
- Modyfikacja wymagań – dostosowanie wymagań do znalezionych komponentów, w przypadku wymagań bardzo istotnych dla systemu, dla których nie znaleziono komponentów, poszukiwanie rozwiązań alternatywnych.

# Reuse – proces produkcji - 2

---

- Projektowanie systemu z komponentami – zaprojektowanie „połączeń” między komponentami, ewentualne zaprojektowanie kodu realizującego wymagania, dla których komponenty nie były dostępne.
- Realizacja systemu i integracja – implementacja i testowanie zaprojektowanego kodu i integracja komponentów.

# Wady i zalety metody reuse

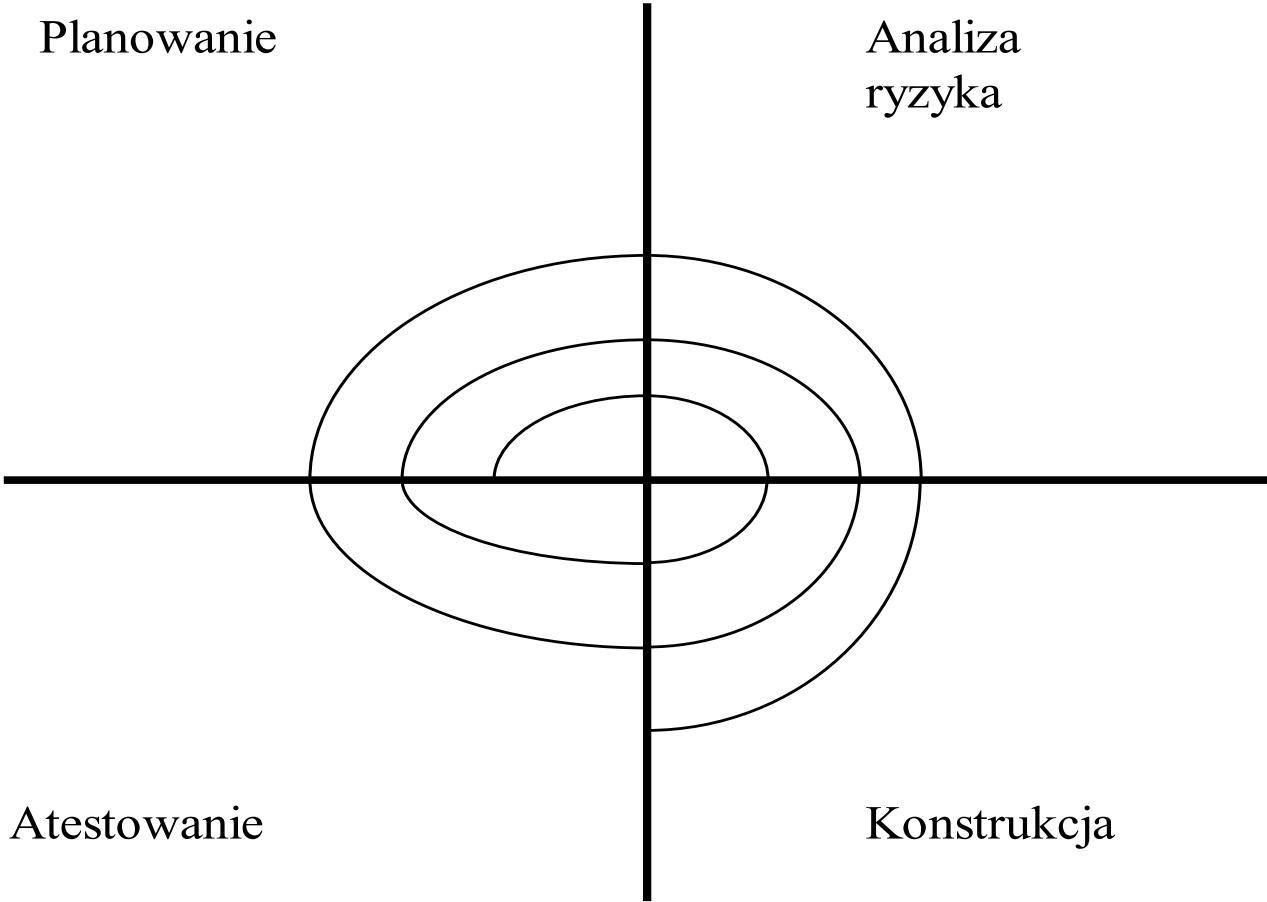
## Zalety:

- wysoka niezawodność
- narzucenie standardów
- małe koszty, szybko

## Wady:

- dodatkowy koszt przygotowania elementów do ponownego użycia
- ryzyko uzależnienia od dostawcy komponentu
- wysokie koszty pielęgnowania
- nie w pełni realizowane wymagania klienta.

# Model spiralny (Boehm)



# Sektorы спиралі

Каждая спираль состоит из четырех секторов:

1. Определение целей, идентификация ограничений, поиск альтернативных решений.
2. Анализ риска, связанного с предлагаемыми решениями, снижение риска например в случае риска, связанного с требованиями, построение прототипа.
3. По определению риска выбирается лучшее решение (о минимальном риске).
4. Планирование следующей спирали, принятие решений о продолжении.

# Model spiralny

---

W modelu spiralnym można włączać inne modele. Np. prototypowanie można zastosować do wykrycia braków w wymaganiach, formalne specyfikacje do budowy fragmentów systemu, dla których są wymagane bardzo wysokie parametry niezawodnościowe, a model wodospadowy dla podsystemów o dobrze określonych wymaganiach.

# Możliwość obserwowania postępu wykonywanych prac

Modele o **dobrej obserwowalności** :

- wodospadowy ,
- formalnych transformacji (każda transformacja kończy się dokumentem, raportem),
- spiralny (w każdym segmencie spirali powstaje dokument).

Modele o **słabej obserwowalności**

- Montaż z gotowych komponentów
- Najmniejsza możliwość obserwowania procesu produkcji jest w modelu ewolucyjnym

# Przykładowe pytania

---

- Wady i zalety modelu wodospadowego.
- Jakie są „deliverable” w modelu wodospadowym ?
- Kiedy można stosować model wodospadowy ?
- Kiedy stosuje się prototypowanie ?
- Dla jakiego typu oprogramowania stosuje się formalne specyfikacje ?
- W jakich modelach procesu produkcji oprogramowania zajmujemy się analizą ryzyka ?
- Na czym polega walidacja ? Na czym polega weryfikacja ?
- Jakie kryteria wyboru funkcji do realizacji możemy stosować w modelu iteracyjnym ?

# Czynniki nie-techniczne w inżynierii oprogramowania

- Zarządzający projektem powinien rozumieć zespół projektowy, osoby indywidualne, ich interakcje, lepsze zrozumienie pozwala na stawianie realnych celów
- Systemy są użytkowane przez ludzi i ich możliwości i ograniczenia powinny być brane pod uwagę podczas projektowania systemu.
- Znajomość czynników społecznych pozwala zwiększyć produktywność pracowników

# Proporcje czasu

---

- praca własna
  - czynności nieprodukcyjne
  - interakcje
- ( 30% , 20% , 50%)

Zespoły których członkowie znają się (wady, zalety) mają większą wydajność niż takie, w których członkowie mają niewielki kontakt ze sobą.

# Osobowości w grupie

zorientowany na:

- **na zadania** - motywacją jest sama praca
- **na siebie** - motywacją jest dążenie do sukcesu, praca jest środkiem
- **na interakcje** - motywacją jest obecność w grupie

Najlepsze efekty - zespoły zróżnicowane, przywódcą osoba zorientowana na pracę.

# ego -less programming

---

- Program uważany jako "własny", trudno przyjmowana krytyka, traktowany jako bezbłędny
- Błędy - normalne, muszą wystąpić
- Program traktowany jako "wspólny" - **ego -less**, przyjmowana krytyka

# Przywódca grup

---

- **wyznaczony** może pełnić funkcje administracyjne
- **wyłaniany** mający największy wpływ na zespół, osobowość dominująca, często na każdym etapie inny

# Integracja zespołu

---

## **Lojalność w grupie, integracja**

- + współpraca, pomoc
- - utrata krytyczyzmu

# Interakcje w grupie

Czynniki wpływające na efektywność interakcji:

- wielkość grupy
- struktura
- status i osobowość członków
- środowisko pracy

$n(n - 1)$  liczba potencjalnych dróg  
komunikacyjnych  $n$  członków grupy

# Inżynieria Oprogramowania inżynieria wymagań

---

Dr hab. inż. Ilona Bluemke

# Plan wykładu

---

- Faza strategiczna
- Niepowodzenia projektów
- Koszty usuwania błędów
- Wymagania

# Faza strategiczna

Wykonywana zanim zostanie podjęta decyzja o realizacji dalszych etapów przedsięwzięcia

- oprogramowanie zamawiane - negocjacje i/lub przetarg
- oprogramowanie rynkowe - rozważana i planowana produkcja nowego programu czy nowej wersji

# Studium wykonalności (feasibility study) -1

czynności:

- rozmowy, wywiady z przedstawicielami klienta
- określenie celów przedsięwzięcia z punktu widzenia klienta
- określenie zakresu oraz kontekstu przedsięwzięcia
- określenie wymagań - ogólne, zgrubna analiza i projekt systemu
- propozycja kilku możliwych sposobów realizacji

# Studium wykonalności (feasibility study) - 2

- oszacowanie kosztów
- analiza rozwiązań
- prezentacja wyników, korekcja
- określenie wstępnego harmonogramu oraz przedstawienie struktury zespołu realizującego
- określenie standardów zgodnie z którymi będzie realizacja

# Decyzje strategiczne

- wybór modelu, zgodnie z którym będzie realizowane przedsięwzięcie
- wybór technik stosowanych w analizie
- wybór środowiska implementacji
- wybór narzędzia CASE
- określenie stopnia wykorzystania gotowych komponentów
- podjęcie decyzji o współpracy z innymi producentami i/lub zatrudnieniu ekspertów zewnętrznych

# faza strategiczna

---

Rozważa się kilka możliwych rozwiązań.

Propozycje rozwiązań powinny być poprzedzone określeniem ograniczeń, przy których przedsięwzięcie będzie realizowane

- maksymalne nakłady
- dostępny personel
- dostępne narzędzia
- ograniczenia czasowe

miesiące

Wstępne zbieranie wymagań

Budowa prototypu

Ocena prototypu

Opracowanie wymagań

Analiza

Projekt dziedziny problemu

Projekt interfejsu użytkownika

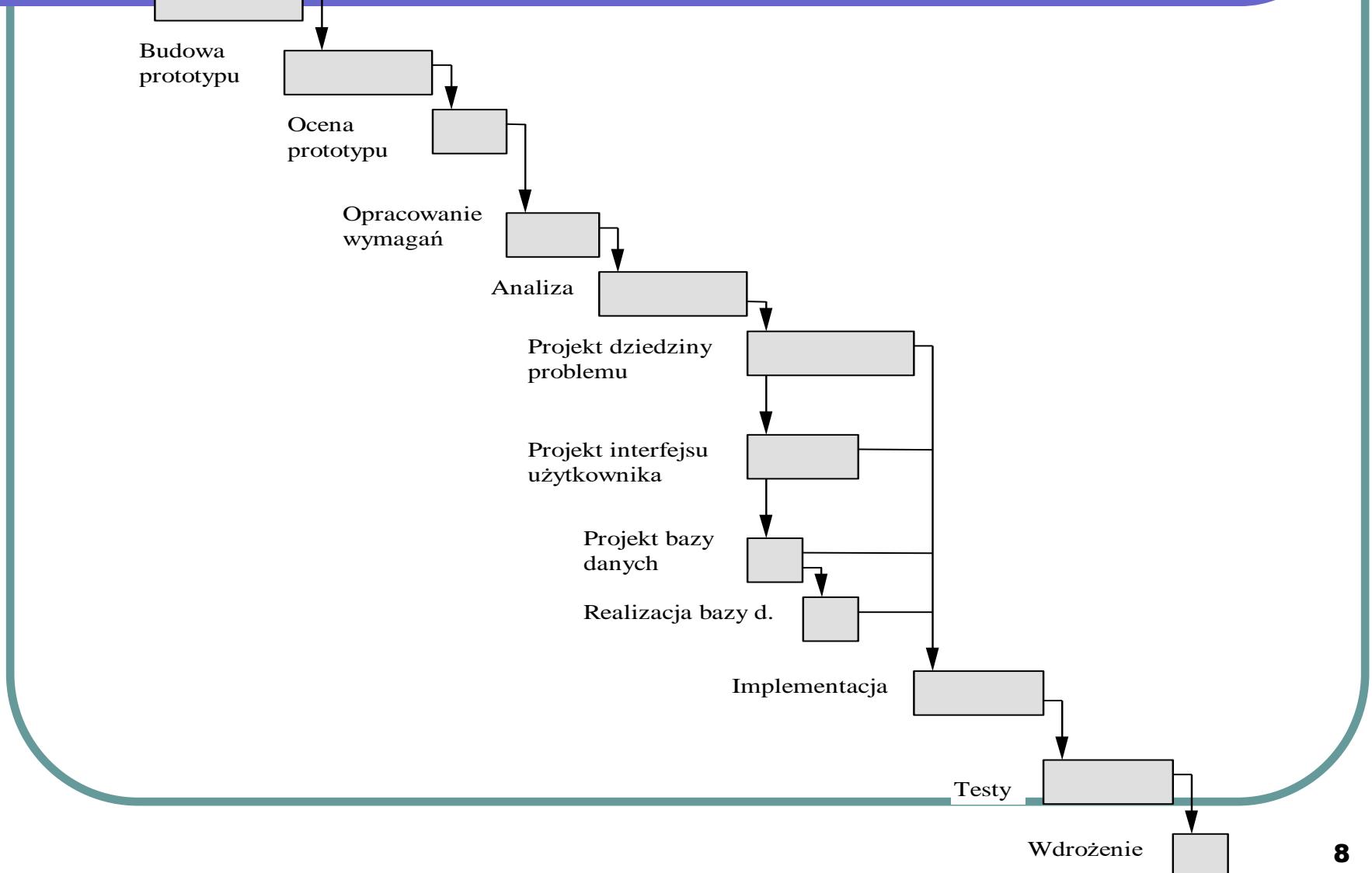
Projekt bazy danych

Realizacja bazy d.

Implementacja

Testy

Wdrożenie



# Normy jakości oprogramowania

---

**ISO/IEC TR 9126** software engineering product quality standards:

- part 1- Quality model : 2001
- part 2 – External metrics : 2003
- part 3 - Internal metrics : 2002

# Syndrom LOOP

**Late**

- późno

**Overbudget**

- przekroczony budżet

**Overtime**

- nadgodziny

**Poor quality**

- kiepska jakość

# Przyczyny niepowodzenia projektów

(wg Standish Group 1994)

- 13% - brak danych wejściowych
- 12% - niepełne wymagania i specyfikacje
- 12% - zmiany wymagań i specyfikacji
- 4% - nierealny plan, harmonogram
- 6% - nieodpowiedni personel
- 7% - nieznajomość technologii

1/3 projektów stwarza problemy z gromadzeniem, dokumentowaniem i zarządzaniem wymaganiami.

# Udane projekty

projekty dostarczone na czas i w ramach budżetu:

- duże firmy - 9%
- małe firmy – 16%

**Przyczyny sukcesu:** (wg Standish Group 1994)

- 16% - zaangażowanie użytkownika
- 14% - wsparcie kierownictwa
- 12% - jasne określenie wymagań

# Podsumowanie wad wg Capersa Jonesa

wada	Potencjalna wada	Skuteczność usuwania	Dostarczone wady
wymagania	1.00	77%	<b>0.23</b>
projekt	1.25	85%	<b>0.19</b>
kod	1.75	95%	<b>0.09</b>
dokumentacja	0.60	80%	<b>0.12</b>
niewłaściwe poprawki	0.40	70%	<b>0.12</b>
suma	<b>5.00</b>	<b>85%</b>	<b>0.75</b>

# Względne koszty usuwania błędów na różnych etapach (Davies 1993):

Względne koszty usuwania błędów oprogramowania na różnych etapach (Davies 1993):

- **0.1-0.2** Określanie wymagań
- **0.5** Projektowanie
- **1** **Kodowanie**
- **2** Testowanie jednostek
- **5** Testowanie akceptacyjne
- **20** Pielęgnacja

# Kategorie błędów projektowych:

---

- projekt budowany z poprawnego zbioru wymagań
- projekt budowany na błędnych wymaganiach (**kosztowne**)
  - projekt będzie przerobiony lub odrzucony, marnotrawstwo czasu, wysiłku
  - błędy ukryte, wykrywane jako błędy wymagań po długim czasie

# Przeciekanie błędów

74 % błędów wymagań wykrywanych na etapie analizy wymagań

„przeciekanie błędów:

- (Hughes Aircraft 15 lat)
- 4% - projekt wstępny, zaawansowany
- 7% - projekt szczegółowy
- 4% - pielęgnowanie

Błędy wymagań pochłaniają 25-40% sumy budżetu

# Naprawa błędu może pociągnąć koszty w obszarach:

- Ponowna specyfikacja
- Ponowne projektowanie
- Ponowne kodowanie
- Ponowne testowanie
- Dokumentowanie
- Działania korygujące – likwidacja uszkodzeń
- Anulowanie np. kodu, projektu bazującego na błędnych wymaganiach
- Wycofanie gotowych wersji oprogramowania
- Koszty gwarancji
- Koszty serwisu (np. instalacja nowej wersji)
- Odpowiedzialność karna związana z produktem

# Wymaganie (def Thyler )

- Możliwość rozwiązania problemu i osiągnięcia celu wymagana przez użytkownika
- Możliwość spełnienia umowy, normy, specyfikacji lub innej dokumentacji, którą musi mieć system

# Zarządzanie wymaganiami

- Systematyczne podejście do **uzyskiwania, organizowania i dokumentowania** wymagań systemu oraz proces, który ustala i zachowuje umowę między klientem a zespołem realizującym przedsięwzięcie w zależności od zmieniających się wymagań systemu.
- Zbiór zorganizowanych, uniwersalnych i usystematyzowanych procesów i technik zajmowania się wymaganiami stawianymi złożonemu dużemu przedsięwzięciu.

# Analiza problemu

Proces rozumienia rzeczywistych problemów i potrzeb użytkownika oraz proponowanie rozwiązań spełniających te potrzeby.

- Uzgodnienie definicji problemu
- Zrozumienie podstawowych przyczyn problemu kryjącego się za innym problemem
- Zidentyfikowanie udziałowców i użytkowników tworzonego systemu
- Zidentyfikowanie granicy systemu
- Zidentyfikowanie ograniczeń nałożonych na rozwiązanie

# Uzgodnienie definicji problemu

Rozpisanie i sprawdzenie, czy zgadza się z tym każdy uczestnik przedsięwzięcia.

Format:

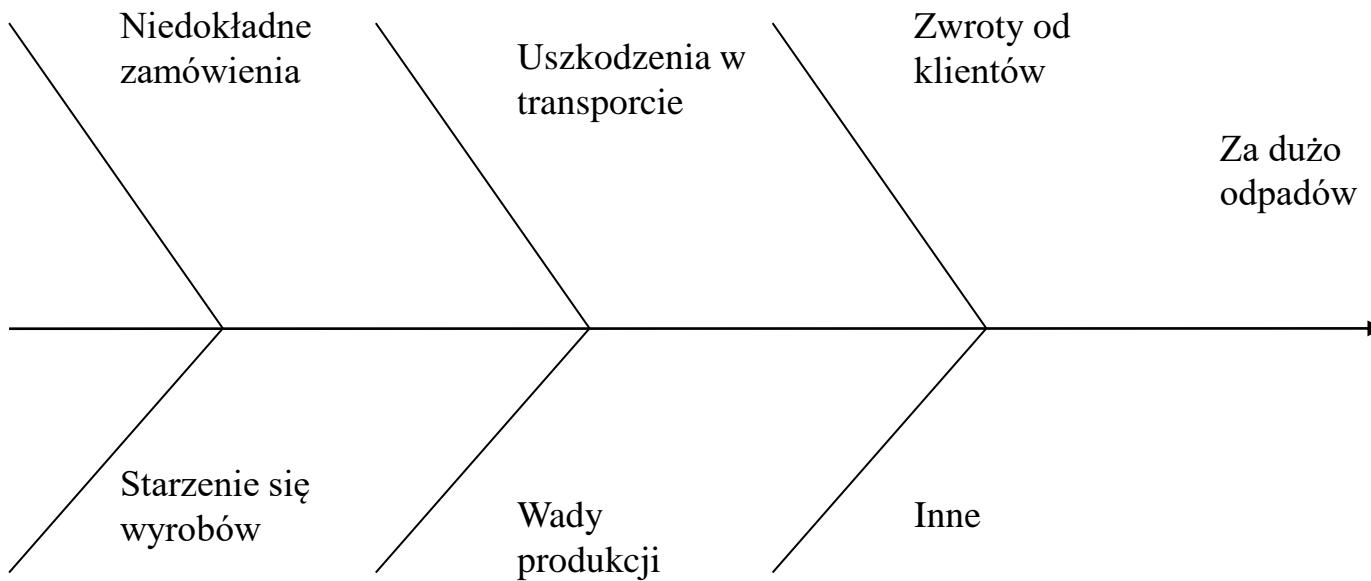
**Problem polega na** - opisz

**Problem dotyczy** - wskaż udziałowców

**Rezultatem problemu jest** - opisz wpływ na udziałowców i przedsiębiorstwo

**Korzyści z rozwiązania problemu** - wskaż proponowane rozwiązanie i wymień podstawowe korzyści

# Zrozumienie podstawowych przyczyn problemu kryjącego się za innym



# Zidentyfikowanie udziałowców i użytkowników

- **Udziałowiec** – każdy na kogo implementacja systemu ma zasadniczy wpływ
- Potrzeby udziałowców nie będących użytkownikami muszą być również określone i uwzględnione

# Pomocne pytania

---

- Kim są użytkownicy
- Jaka jest rola klienta systemu
- Na kogo jeszcze będą miały wpływ wyniki działania systemu
- Kto będzie oceniał i zatwierdzał system po jego dostarczeniu
- Czy są inni zewnętrzni i wewnętrzni użytkownicy systemu, których potrzeby muszą być uwzględnione
- Kto będzie pielęgnował system

# Identyfikacja aktorów

---

klucz w analizie problemu

- Kto dostarcza, używa lub usuwa informacje
- Kto obsługuje
- Kto utrzymuje, pielęgnuje, konserwuje
- Gdzie system będzie stosowany
- Gdzie zostaną wprowadzone dane
- Jakie inne systemy zewnętrzne będą z nim oddziaływać

# Zidentyfikowanie ograniczeń nałożonych na rozwiązanie

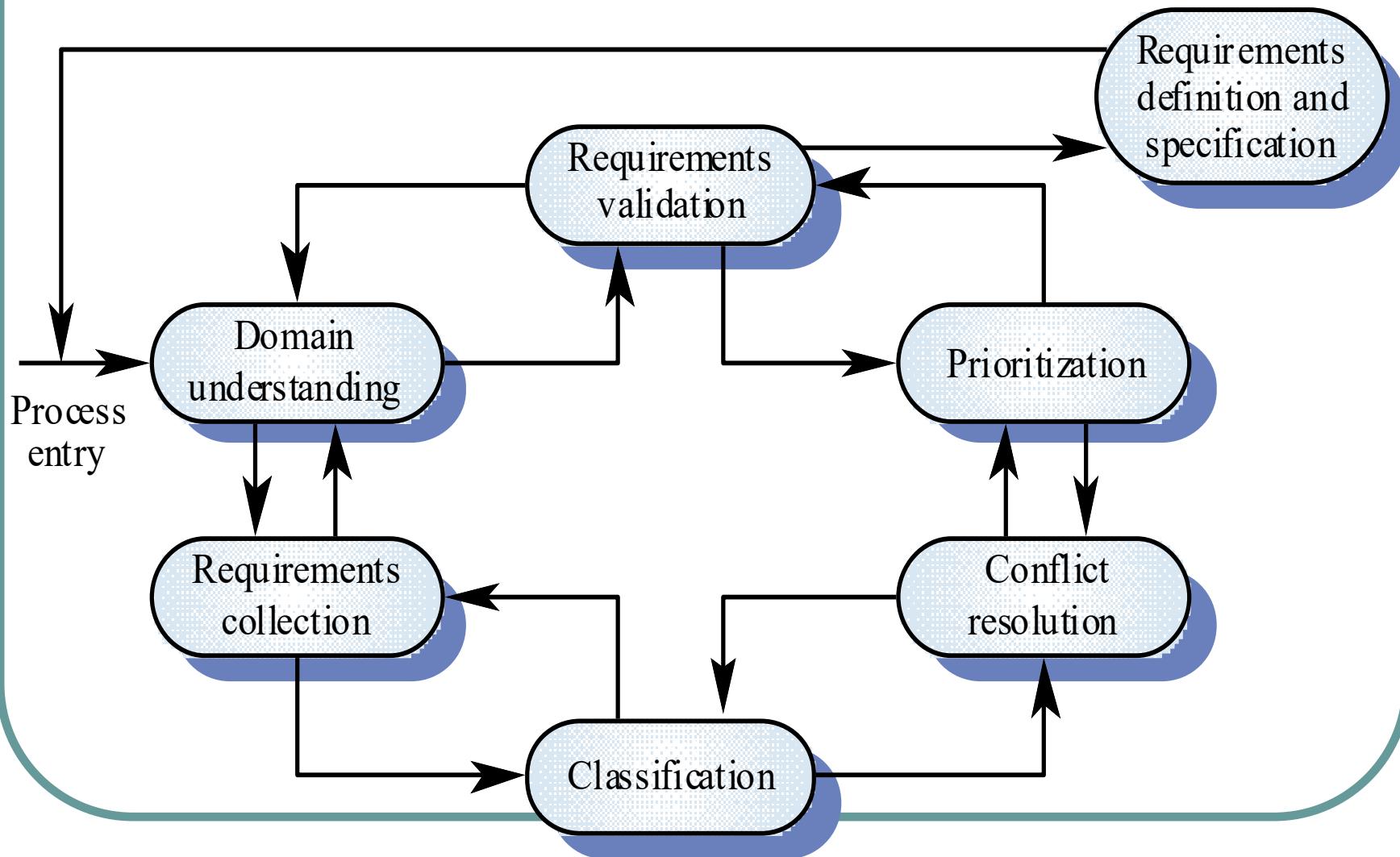
- **Ekonomiczne (ograniczenia finansowe, budżetowe, licencyjne)**
- **Polityczne**
- **Techniczne (ograniczenia w wyborze technologii, platformy, zakupione pakiety)**
- **Systemowe (rozwiązanie wbudowane w istniejący system, kompatybilność z innymi rozwiązaniami, systemy operacyjne)**
- **Środowiskowe (bezpieczeństwo, prawo, normy)**
- **Planowanie i zasoby (ograniczenia zasobów, ich zwiększenie stałe, chwilowe, zatrudnienie osób spoza firmy)**

# Uzyskiwanie wymagań

---

- Wywiad, ankieta
- Warsztaty wymagań
- Burza mózgów
- Przypadki użycia
- Odgrywanie ról
- Prototypy

# Proces analizy wymagań



# Rodzaje wymagań

- **Wymagania funkcjonalne** są to usługi oczekiwane przez użytkownika (bez uwag implementacyjnych).
- **Wymagania niefunkcjonalne** są to ograniczenia w jakich system ma pracować, standardy jakie spełnia np.
  - ✓ czas odpowiedzi,
  - ✓ zajętość pamięci,
  - ✓ niezawodność.

# Wymagania funkcjonalne

**Wymagania funkcjonalne** powinny być:

- pełne, czyli zawierać wszystkie wymagania użytkownika,
- spójne, nie sprzeczne.

Istnieje wiele metod opisu wymagań np. w postaci formularzy, diagramów przypadków użycia, metod formalnych.

# Przykłady

- **Wymaganie produktu**

Komunikacja pomiędzy systemem a użytkownikiem powinna być wyrażona w Unicode.

- **Wymaganie organizacyjne**

Proces rozwoju systemu i dostarczane dokumenty muszą być zgodne z normą (ISO 9000, CMMI).

- **Wymaganie zewnętrzne**

System nie powinien ujawniać operatorom żadnych danych osobowych klientów oprócz nazwisk i numerów identyfikacyjnych.

# Wymagania niefunkcjonalne

- **Szybkość** – czas odpowiedzi użytkownika,
  - liczba przetworzonych transakcji w sekundzie
- **Rozmiar** – kilobajty
- **Łatwości użycia** - czas szkolenia,
  - liczba ekranów pomocy,
- **Niezawodność** – dostępność systemu,
  - średni czas pomiędzy błędami,
  - częstotliwość pojawiania się błędów,
  - prawdopodobieństwo błędu żądanej usługi,

# Wymagania niefunkcjonalne-2

- **Solidność (robustness)** – czas uruchomienia po awarii,
  - procent zdarzeń powodujących awarie,
  - prawdopodobieństwo zniszczenia danych po awarii
- **Przenośność** – liczba systemów na których działa programowanie,
  - procent zależnych od systemu instrukcji.

# Specyfikacje wymagań

- **Strukturalny język naturalny**  
(formularze, szablony)
- **Pseudokod**  
(**if, else, while, ...**, operatory logiczne, wcięcia, tekst)
- **Języki opisu projektu, opisu wymagań**  
(PDL, PSL/PSA)
- **Notacje graficzne ze strukturalnymi opisami**  
(SADT, przypadki użycia – use case)

# Specyfikacje techniczne

- **Maszyny skończenie stanowe (FSM)**  
(automaty, diagramy stanów)
- **Tabele decyzyjne, Drzewa decyzyjne**
- **Diagramy czynności (schematy blokowe)**
- **Modele związków encji**
- **Modele przepływu danych (DFD)**
- **Modele obiektowe dziedziny problemu**
- **Modele sieciowe (sieci Petri)**
- **Specyfikacje formalne**

# Wymaganie użytkownika

## 3.5.1 Dodawanie węzłów do projektu

3.5.1.1 Edytor będzie udostępniał użytkownikom udogodnienia do dodawania do swoich projektów węzłów określonego typu

3.5.1.2 Sekwencja czynności:

1. Użytkownik powinien wybrać typ węzła, jaki należy dodać
2. Użytkownik powinien przesunąć wskaźnik w okolice miejsca nowego węzła i zlecić jego dodanie
3. Użytkownik powinien przeciągnąć węzeł do jego ostatecznego położenia

# Formularz specyfikacji wymagania

---

- Funkcja
- Opis
- Dane wejściowe
- Źródło danych wejściowych
- Dane wyjściowe, wynik
- Przeznaczenie
- Ograniczenia
- Warunek wstępny
- Warunek końcowy
- Efekty uboczne
- Uzasadnienie

# Przykład specyfikacji wymagania

**Funkcja** *Dodaj węzeł*

**Opis** *Dodaje węzeł do istniejącego projektu. Użytkownik wybiera typ i położenie węzła (przesuwa wskaźnik na właściwy obszar). Po dodaniu węzeł jest zaznaczony.*

**Dane wejściowe** *Typ, położenie węzła, Identyfikator projektu*

**Źródło danych wejściowych** *Użytkownik, Baza danych*

**Dane wyjściowe, wynik** *Identyfikator projektu*

**Przeznaczenie** *Baza danych projektów*

**Wymagania** *Identyfikator określa korzeń grafu projektu*

**Warunek wstępny** *Projekt jest otwarty i wyświetlony*

**Warunek końcowy** *Pozostałe elementy projektu nie ulegają zmianie*

**Efekty uboczne** *Brak*

# Klasy stałości wymagań

- **Wymagania stałe** – stabilne wymagania wynikające z podstawowej działalności firmy. Można je wywnioskować z modeli dziedziny zastosowania.  
Np. szpitalu zawsze są wymagania dotyczące pacjentów, lekarzy, pielęgniarek itp.
- **Wymagania niestałe** – prawdopodobnie ulegną zmianie w trakcie tworzenia systemu lub po przekazaniu go użytkownikowi.  
Np. zasady finansowania szpitala wg aktualnej ustawy o ochronie zdrowia

# Poziomy identyfikacji wymagań

## 1) Potrzeby udziałowców

### Cele systemu (goals)

- często niejasne i niejednoznaczne

## 2) Cechy systemu – usługi, których dostarcza system w celu spełnienia jednej lub więcej potrzeb udziałowca

## 3) Wymagania stawiane oprogramowaniu

# Atrybuty cech produktu (1)

- **status**

śledzi postęp np. zaproponowano, zatwierdzono, wprowadzono

- **priorytet/korzyść**

stosuje się w zarządzaniu zakresem  
np. konieczne, ważne, użyteczne

- **wysiłek**

oszacowanie liczby osób, tygodni, linii kodu np.  
poziom wysiłku niski, średni, wysoki

- **ryzyko**

wielkość prawdopodobieństwa, że cecha  
spowoduje niepożądane zdarzenie np. poziom  
ryzyka niski, średni, wysoki

# Atrybuty cech produktu (2)

- **stabilność**  
wielkość prawdopodobieństwa, że cecha zmieni się
- **wersja docelowa**  
w której cecha pojawi się po raz pierwszy
- **przydzielenie do**  
cechy musza być przydzielone do przyszłych zespołów odpowiedzialnych za definiowanie, ew. realizację
- **powód**  
do śledzenia źródła pożądanej cechy, np. odwołanie do strony i wiersza specyfikacji

# Macierz atrybutów

Identyfikator

Krótki tekst

Atrybut

Atrybut

AT: Attribute Ma		Priority	Difficulty
Requirements:		Medium	High
FEAT1: The QBS system shall, upon user request,...		Low	Medium
FEAT2: The QBS system shall provide a loan officer...		Medium	Medium
FEAT3: The QBS System shall calculate the blue...		High	Medium
FEAT4: The QBS system shall allow only...		High	Medium
FEAT5: The QBS system shall allow only...			

FEAT1: The QBS system shall, upon user request, display detailed customer information including:  Name,  Address,  Phone Number  Account Numbers  L

Pełny tekst

# Miary jakości

**Miary jakości** do oceny specyfikacji wymagań stawianych oprogramowaniu (IEEE 830)

- poprawny
- jednoznaczny
- kompletny
- spójny
- uporządkowany wg ważności i stabilności
- sprawdzalny
- modyfikowalny
- możliwy do śledzenia oraz możliwy do zrozumienia

# Zatwierdzanie wymagań

- **Przeglądy wymagań**
  - Systematyczna analiza przez zespół recenzentów
- **Prototypowanie**
  - Wykonywalny model systemu (poziomy prot.)
- **Generowanie przypadków testowych**
  - Zaprojektowanie testów akceptacyjnych dla wymagań funkcjonalnych i niefunkcjonalnych
- **Automatyczna weryfikacja niesprzeczności**
  - Wykrywanie niezgodności w bazie wymagań zgodnie z regułami (CASE, modele formalne)

# Zarządzanie zmianami wymagań

- **Planowanie zmian**  
(atrybut stabilności, rozróżnianie starych, znanych, nowych, modyfikowanych wymagań)
- **Tolerancja linii bazowej na zmiany**
- **Kanał kontroli zmian**  
(gromadzenie żądań zmian, ocena legalności, wpływu na system, podejmowanie decyzji, rozpowszechnianie informacji o zmianach)
- **Dokumentacja historii zmian**
- **Zarządzanie konfiguracją wymagań**  
(wersjonowanie)

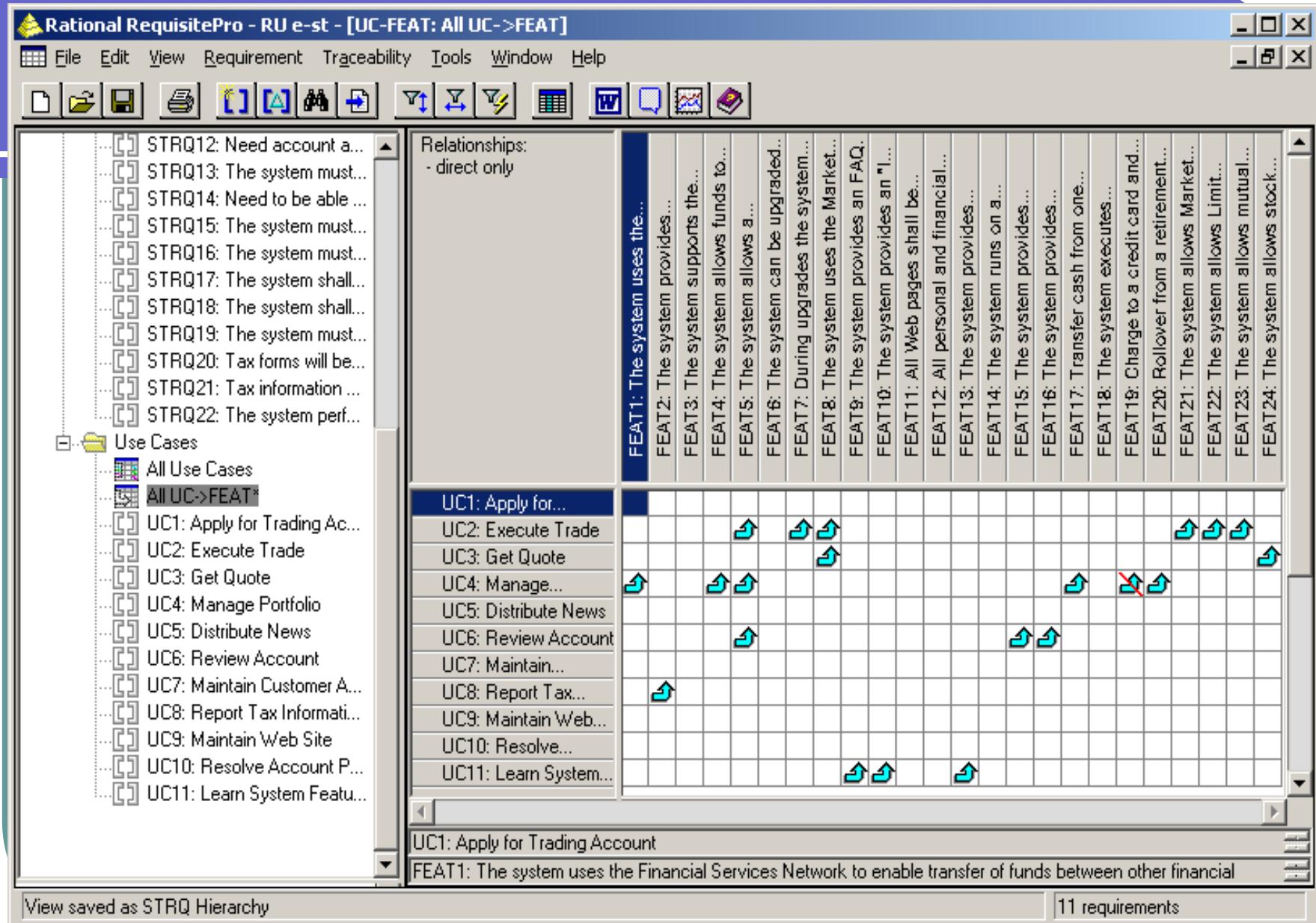
# Śledzenie zależności (Traceability)

Określanie i pielęgnowanie relacji zależności pomiędzy różnymi artefaktami tworzonymi w cyklu rozwoju oprogramowania.

- **Pochodzenie**
  - Wskazanie na udziałowców, którzy proponowali to wymaganie
- **Uzależnienie wymagań**
  - Powiązania pomiędzy wzajemnie zależnymi wymaganiami
- **Związki z projektem**
  - Powiązania z modułami projektu implementującymi wymagania, przypadkami testującymi,...

# Macierz zależności (traceability matrix)

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			D		D
1.3	R			R				
2.1		R			D			D
2.2							D	
2.3		R		D				
3.1							R	
3.2							R	



# Narzędzia wspomagające CASE

- **Przechowywanie wymagań**
  - Gromadzenie w sposób bezpieczny i zorganizowany wymagań i ich atrybutów
- **Zarządzanie zmianami**
  - Dokumentowanie procesu zmian
  - Zachowanie spójności zbioru wymagań
  - Zarządzanie konfiguracjami wymagań
- **Wspomaganie śledzenia zależności**
  - Automatyczne identyfikowanie zależności
  - Przechowywanie, aktualizacja zależności.

# Specyfikacje formalne

Dr hab. inż. Ilona Bluemke

# Typy specyfikacji formalnych

---

Specyfikacje formalne można podzielić na dwa typy:

- Algebraiczne
- Bazujące na modelu.

# Specyfikacje algebraiczne

Obiekt jest specyfikowany jako zbiór relacji między operacjami na tym obiekcie. Po raz pierwszy specyfikacja algebraiczna wprowadzona była przez Guttag'a w 1977 do specyfikacji abstrakcyjnych typów danych.

W specyfikacjach algebraicznych są notacje **sekwencyjne**:

- **OBJ** (Futatsugi.. 1985),
- **Larch** (Guttag.. 1985),

oraz notacje **współbieżne**: **Lotos** (Bolognesi , Brinksma 1987)

# Specyfikacje algebraiczne-2

Specyfikacja algebraiczna określa operacje na obiekcie i specyfikuje je podając relacje między nimi. Składa się z dwóch części:

- **sygnaturowej** - zawierającej operacje i ich parametry,
- **aksjomatycznej** - zawierającej definicje operacji.

Można tworzyć nowe typy dziedziczące operacje i aksjomaty istniejących już typów i dodając nowe operacje i aksjomaty.

# Współrzędne kartezjańskie

```
sort Coord  
imports Integer, Boolean;
```

---

Specyfikacja def. Coord - reprezentujący współrzędne kartezjańskie.  
Operacje zdefiniowane dla Coord to X, Y, które obliczają atrybuty  
oraz Eq porównująca 2 obiekty typu Coord.

---

```
Create (Integer, Integer) -> Coord;  
X (Coord) -> Integer;  
Y (Coord) -> Integer;  
Eq (Coord,Coord) -> Boolean;
```

---

```
X (Create(x,y)) = x;  
Y (Create(x,y)) = y;  
Eq(Create(x1,y1),Create(x2,y2)) = ((x1=x2) and (y1=y2))
```

---

# Tworzenie specyfikacji algebraicznych

- Strukturalizacja specyfikacji (wybór typów abstrakcyjnych).
- Nazwanie specyfikacji, określenie, czy ma ona parametry generyczne.
- Wybór operacji np. tworzenia, modyfikacji, inspekcji.
- Nieformalna specyfikacja operacji.
- Definicja składni i parametrów.
- Definicja aksjomatów, określenie semantyki operacji poprzez określenie warunków, które są zawsze spełnione dla różnych kombinacji operacji.

# Specyfikacji algebraiczna dla listy

```
sort Lista  
imports Integer
```

---

Specyfikacja definiuje listę, której elementy są dodawane na końcu, a usuwane z początku. Operacja Create tworzy pustą listę, a Cons dodaje element do listy. Operacja Length podaje liczbę elementów listy, Head podaje początkowy element listy a Tail usuwa pierwszy element z listy.

---

```
Create -> List  
Cons (List,element) -> List  
Head (List) -> element  
Length(List) -> Integer  
Tail (List) -> List
```

---

Head (Create) = undefined **exception** (empty list)  
Head (Cons(L,v)) = **if** L = Create **then** v **else** Head (L)  
Length (Create) = 0  
Length (Cons(L,v)) = Length (L) + 1  
Tail (Create) = Create  
Tail (Cons (L,v)) = **if** L = Create **then** Create **else** Cons (Tail (L),v)

---

# Rekursja w specyfikacji algebraicznej

Operacja Tail jest zdefiniowana na pustej liście, a potem rekursywnie na liście niepustej, rekursja kończy się, gdy lista jest pusta.

Przy specyfikacji rekursywnej warto jest sprawdzić ją na prostym przykładzie.

$\text{Tail} ([3,4,5]) = \text{Tail} (\text{Cons} ([3,4], 5) ) =$

$\text{Cons} (\text{Tail} ([3,4], 5)) = \text{Cons} (\text{Tail} (\text{Cons} ([3], 4 )), 5 ) =$

$\text{Cons} (\text{Cons} (\text{Tail} ([3]), 4 ), 5) =$

$\text{Cons} (\text{Cons} (\text{Tail} (\text{Cons} ([]), 3)), 4), 5) =$

$\text{Cons} (\text{Cons} ([\text{Create}], 4), 5) = \text{Cons} ([4],5) = [4,5]$

# Specyfikacja bazująca na modelu

może być stosowana, gdy system modelowany jest takimi obiektami jak zbiory, funkcje.

Istnieją techniki **sekwencyjne** np. :

- **VDM** ( Vienna Definition Method - Jones 1980, 1986),
- **Z** (Zed) (Abrial 1980, rozwijany w Oxford Hayes 1986, Spivey 1990)

oraz współbieżne.

# Z-schematy

Dla Z-schematów są dostępne programy do tworzenia i sprawdzania specyfikacji.

Specyfikacja w Z jest pewną liczbą schematów.

Każdy schemat wprowadza określonego typu nazwę i definiuje predykaty dla tej nazwy.

Schematy mogą być prezentowane w grafice. Są "bloczkami" do budowy innych schematów.

# Przykład Z-schematu

**pojemnik**

---

*nazwa*

**zawartość : N**

*sygnatury*

**pojemność : N**

---

**zawartość <= pojemność  
schematu**

*predykat*

# Z-schematy

- Można "składać" schematy tak by dziedziczyły sygnatury, predykaty.
- Można określać
  - operacje **wejścia** nazwa?: N
  - operacje **wyjścia** nazwa!: N
- Jeśli nazwa schematu jest poprzedzona grecką literą  $\Delta$  (delta schemat) oznacza to, że jedna lub więcej wartości zmiennych stanu **zostanie zmienionych** przez operacje. Dla wszystkich zmiennych wprowadzonych w schemacie można wprowadzić także zmienną<sup>1</sup> określającą zmienioną wartość zmiennej.

# Z-schematy-2

- Jeśli nazwa schematu jest poprzedzona grecką literą **Ξ** (Xi schemat) oznacza to, że wartości zmiennych stanu **nie zostaną zmienione** przez operacje (zmienna' nie zmieniona).
- Można korzystać z funkcji, także takich, które nie dla wszystkich dozwolonych wejść (dziedzina) mają określone wartości wyjścia (**partial**). Są operatory pozwalające na działania na dziedzinie.
- Jeśli istotna jest kolejność to można użyć sekwencji (**sequence**).

# Schemat wskaźnik

## **wskaźnik**

---

lampka : {on, off}

odczyt : N

niebezpieczny\_poziom : N

---

lampka = on  $\Leftrightarrow$  odczyt  $\leq$  niebezpieczny\_poziom

# Schemat zbiornik

Zmienne stanu schematów (odczyt , zawartość)  
użyte są do połączenia obu schematów.

## **zbiornik**

---

pojemnik

wskaźnik

---

odczyt = zawartość

pojemność = 4000

niebezpieczny\_poziom = 40

# Schemat napełnianie\_OK

**napełnianie\_OK**

---

Δ zbiornik

ilość? : N

---

zawartość + ilość? ≤ pojemność

zawartość' = zawartość + ilość

# Schemat przepelnienie

- ☒ oznacza, że zmienne stanu tego schematu **nie mogą zostać zmienione**, jest użyty w schemacie **przepelnienie**
- 

☒ zbiornik

ilość? : N

t! : seq char

---

pojemność < zawartość + ilość?

t! = „niewystarczająca pojemność zbiornika – napełnianie skasowane”

# Schemat napełnianie

Następnie schematy **przepelnienie** oraz **napelnianie\_OK** są połączone w definicji schematu **napelnianie**.

**napelnianie**

---

**napelnianie\_OK**  $\cup$  **przepelnienie**

# **ANALIZA I PROJEKTOWANIE STRUKTURALNE**

---

**Dr hab. inż. Ilona Bluemke**

# Metody strukturalne (structured methods)

(koniec lat 60) opierają się na wyróżnieniu w analizowanym systemie składowych :

- **pasywnych** - odzwierciedlających fakt przechowywania w systemie pewnych danych
- **aktywnych** - odzwierciedlających fakt wykonywania w systemie pewnych operacji

# Analiza strukturalna

budowa dwóch modeli systemu:

- **danych** - pasywna część
- **funkcji** - aktywna część

Modele te są integrowane w **model przepływu danych (data flow)**

Strukturę oprogramowania pokazują **drzewa struktur (structure charts)**

# Metody strukturalne przydatne:

gdy:

- system realizuje złożone funkcje na prostych danych
- system służy do przechowywania i wyszukiwania złożonych danych

Wirth 1971, 1976

Myers 1975

Constantine, Yourdon 1979, 1986

# Narzędzia modelowania systemu

- **diagram przepływu danych (DFD)** - ilustruje funkcje które musi realizować system
- **diagram związków encji (ERD)** - uwypukla związki między danymi
- **diagram sieci przejść (STD)** - koncentruje się na charakterystyce czasowej
- **diagram struktury (structure charts)** - opisuje strukturę oprogramowania

# Diagram przepływu danych (DFD)

Pokazuje jak dane przepływają z jednej jednostki przetwarzającej do następnej. Nie zawierają informacji sterujących. Autorzy stosują różne notacje graficzne.

- **Procesy (centra transformacji danych)**

bąble, elipsy, kółka

Oznacza się je mnemonikami opisującymi rodzaj transformacji. Reprezentują realizowane funkcje

- **Przepływy przedstawiane strzałkami.**

Określają kierunek przepływu danych, oznaczenia opisują przepływające dane.

Tworzą związek między procesami

# Diagram przepływu danych (DFD)-2

- **Magazyny danych (data store)**

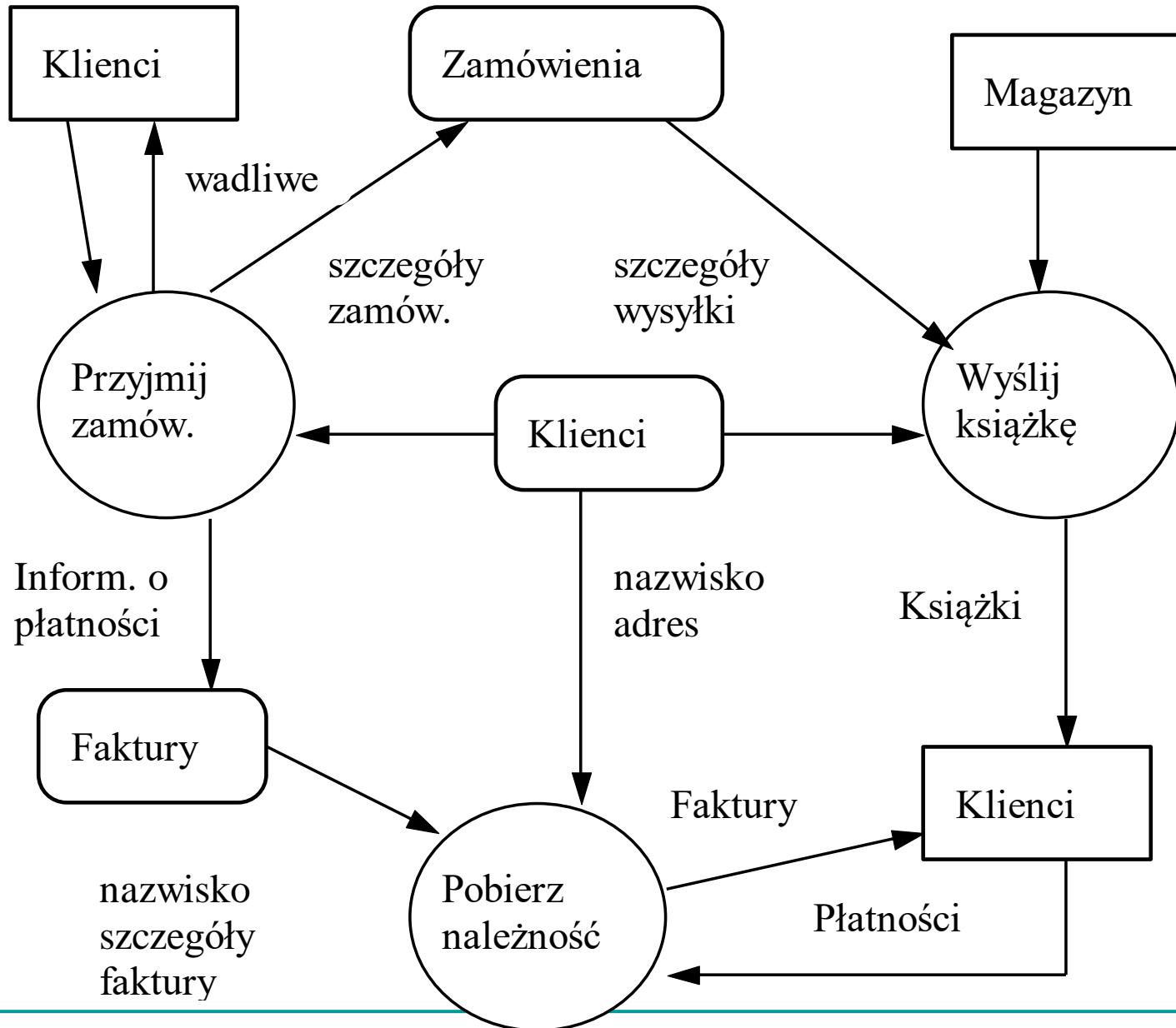
linie równoległe, prostokąty

Pokazują zbiory danych, które system przechowuje.

- **Terminatory, interakcje**

kółka

Przedstawiają zewnętrzne obiekty z którymi komunikuje się system (osoby, komputery)



# DFD

- Diagramy przepływu danych pokazują transformacje bez robienia założeń jak są one implementowane.
- Dla złożonych systemów **DFD są hierarchiczne**.
- Tekstowe narzędzia modelowania pokazują jak informacja jest przekształcana
  - **słownik danych**
  - **specyfikacja procesu**

# Słownik danych

np.

nazwisko = tytuł + imię + (inicjał) + nazwisko

tytuł = [Pan | Pani | Dr | Prof. | Mgr ]

imię = {dozwolony znak }

nazwisko = {dozwolony znak }

dozwolony znak = [ A-Z | a-z | ' | - ]

# Specyfikacja procesu

Opis co dzieje się wewnątrz każdego procesu na najniższym poziomie diagramu przepływu danych (minispec) np. Demarco 1978, Gene Sarson 1977, Weinberg 1978

## Narzędzia specyfikacji procesu:

- język naturalny
- warunki początkowe i końcowe
- tablice decyzyjne
- diagramy przepływu sterowania
- diagramy Nassi- Shneidermana
- inne

# Język naturalny

Proces 1.1.4: Wprowadź zamówienie

**BEGIN**

**DO WHILE** istnieje więcej pozycji-zamówienia w poprawnych-szczegółach-zamówienia

**TWÓRZ** rekord pozycji-zamówienia z następnej pozycji-zamówienia w poprawnych-szczegółach zamówienia

**DOŁĄCZ** rekord pozycji-zamówienia do POZYCJI-ZAMÓWIENIA

**ENDDO**

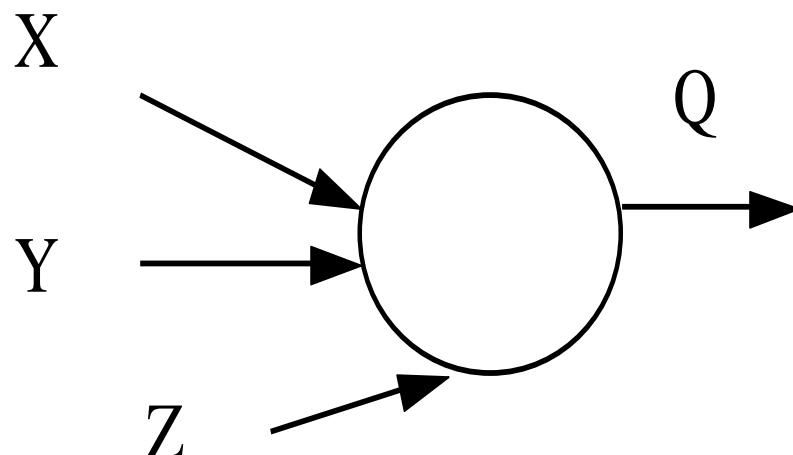
...

**END**

# Warunki początkowe i końcowe

Wybór algorytmu pozostawiony programiście

**Warunki początkowe** dają odpowiedź na pytania: jakie dane wejściowe muszą być gotowe np.



X - niezbędna  
Y, Z - nie są konieczne

# Warunki początkowe

- Jakie związki między wejściami lub wewnątrz wejść muszą być spełnione np. 2 wejścia nadchodzą
- Jakie związki między wejściami a magazynami danych muszą być spełnione np. istnieje zamówienie klienta z numerem klienta pasującym do numeru klienta w magazynie klienci
- Jakie związki muszą istnieć między różnymi magazynami lub wewnątrz tego samego magazynu np. istnieje zamówienie w magazynie zamówienia, którego numer konta klienta pasuje do numeru konta klienta w magazynie klientów.

# Warunki końcowe

Określają co musi być spełnione, gdy proces zakończy działanie. Opisuję:

- wyniki wyprodukowane przez proces
- związki, jakie będą istnieć między wartościami wynikowymi a oryginalnymi wartościami wejściowymi np. suma faktury będzie sumą cen pojedynczych pozycji i kosztów przesyłki
- związki między wartościami wynikowymi a wartościami w jednym lub więcej magazynie np. saldo na stanie w magazynie Magazyn zostanie zwiększone o otrzymaną kwotę
- zmiany jakie będą dokonane w magazynach (dodanie nowych pozycji, modyfikacja istniejących lub usunięcie)

# Specyfikacje za pomocą warunków

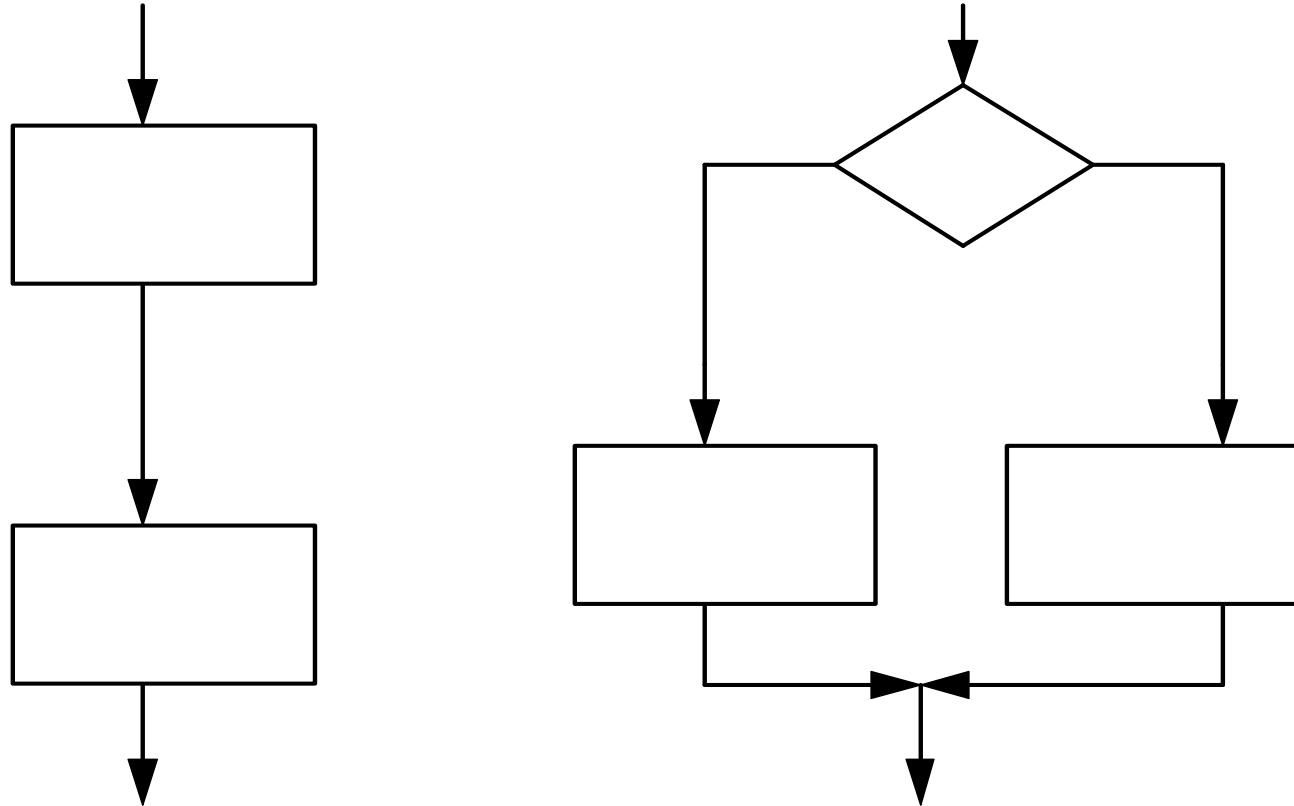
- Budując specyfikację warunków początkowych i końcowych należy zacząć od warunków normalnych, potem dołączyć warunki dla sytuacji błędnych.

# Tablice decyzyjne

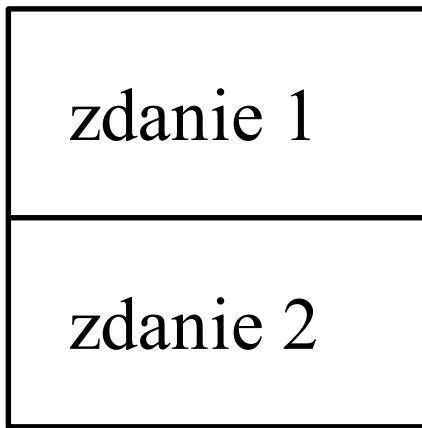
Proces może podejmować akcje zależne od wielu zmiennych a zmienne te mogą przyjmować wiele różnych wartości

zmienne	1	2	3	4	5
wiek > 21	T	T	T	N	N
płeć	M	K	K	M	M
waga >70	T	N			
lek 1	X				
lek 2		X			
lek 3			X		
lek 4				X	
bez leku					X

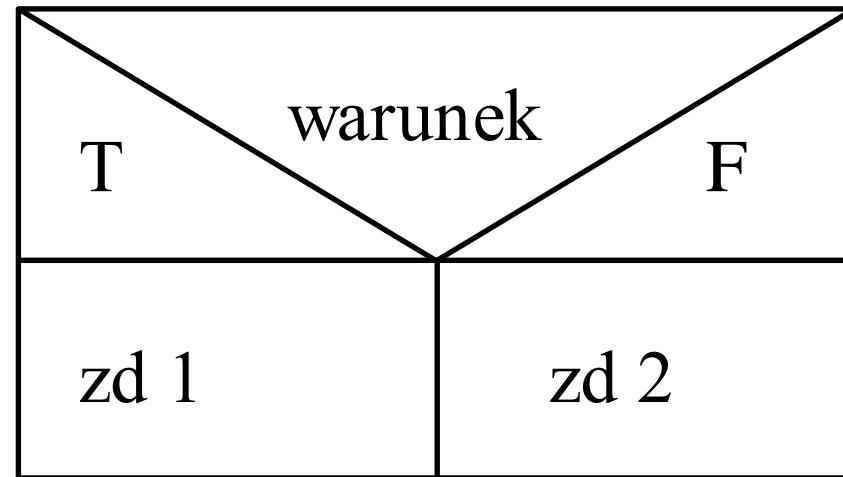
# Diagramy przepływu sterowania (symbole Bohma-Jacopiniego)



# Diagramy Nassi- Shneidermana

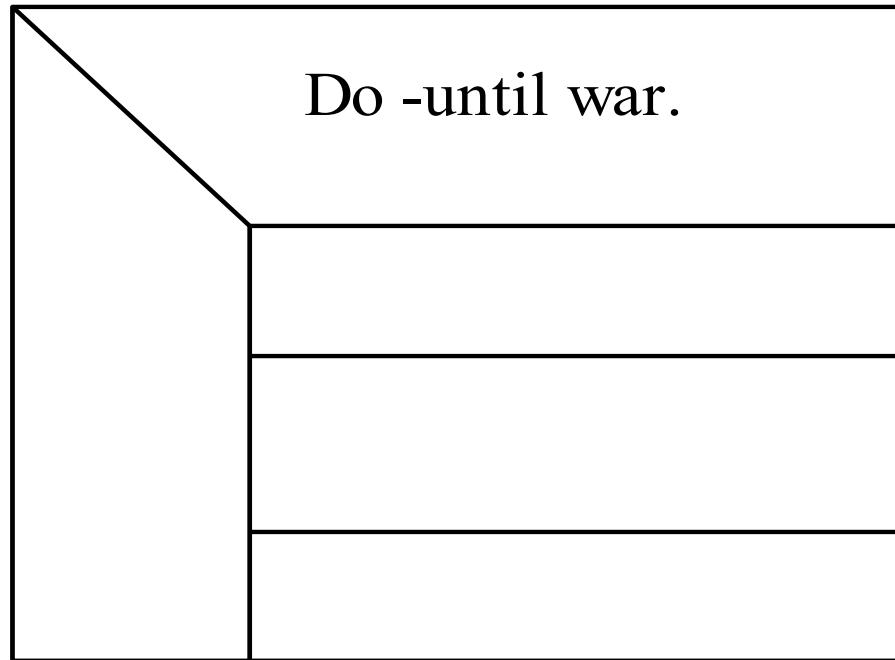


Reprezentacja  
zdań  
sekwencyjnych



Reprezentacja  
konstrukcji  
if-then-else

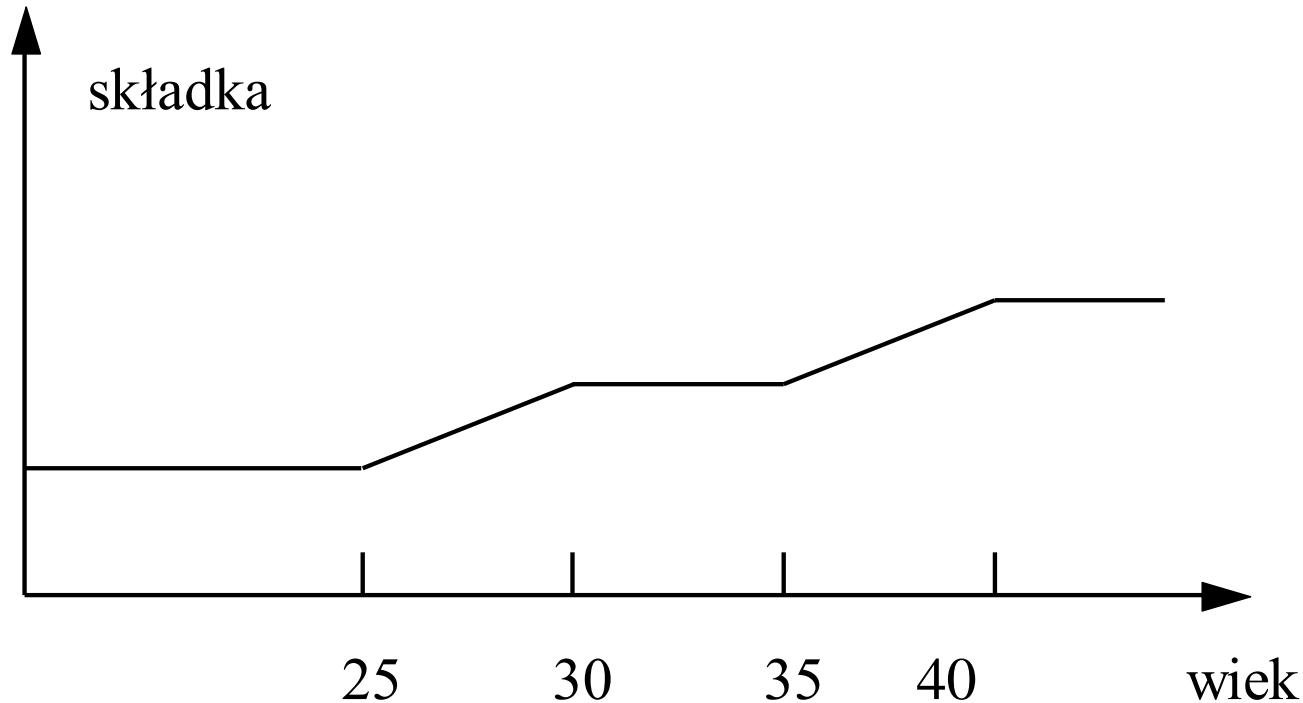
# Reprezentacja konstrukcji cyklu



Reprezentacja  
konstrukcji  
DO- WHILE

# Inne narzędzia specyfikacji procesów

## Grafy, wykresy



# Diagram związków encji (ERD)

Modelowanie gromadzonych danych

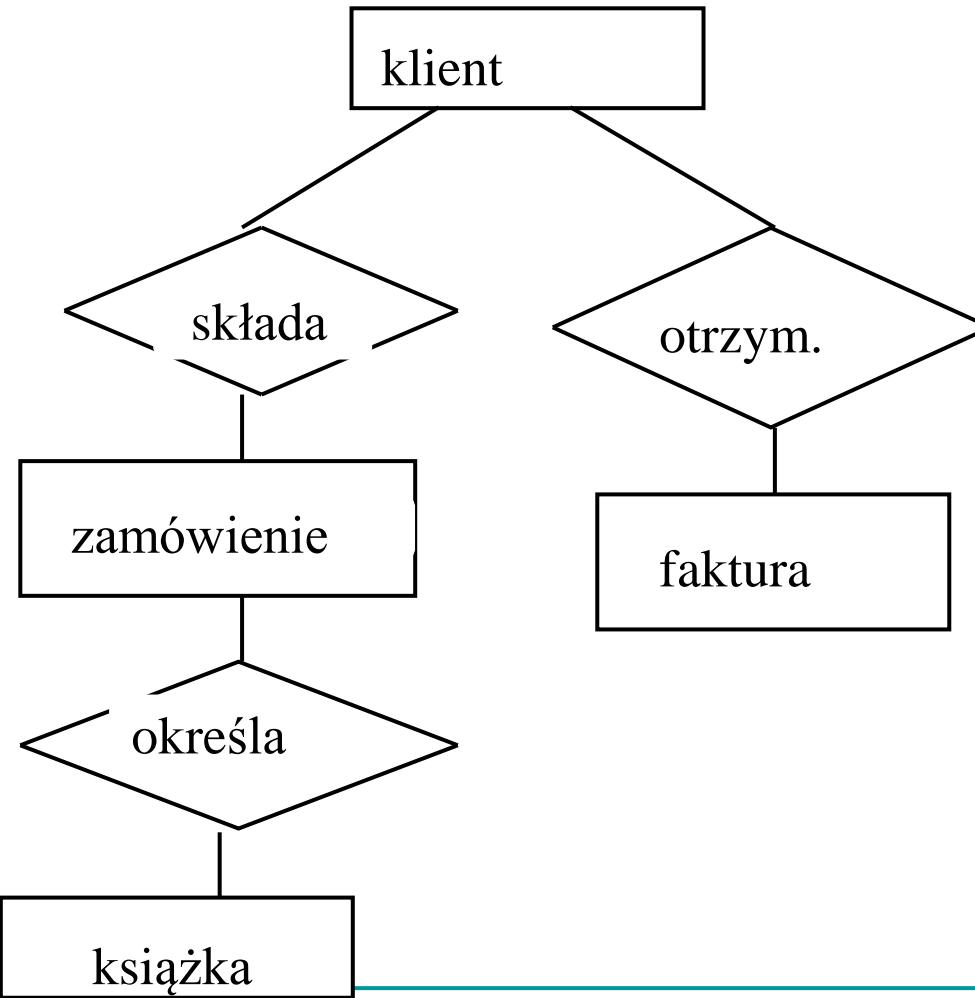
## ■ Typy obiektów

- Typ obiektu reprezentuje zbiór lub kolekcję obiektów ze świata rzeczywistego, które mają znaczenie dla budowanego systemu, mogą być jednoznacznie zidentyfikowane i opisane przez atrybuty, zazwyczaj oznaczane prostokątami.

## ■ Związki

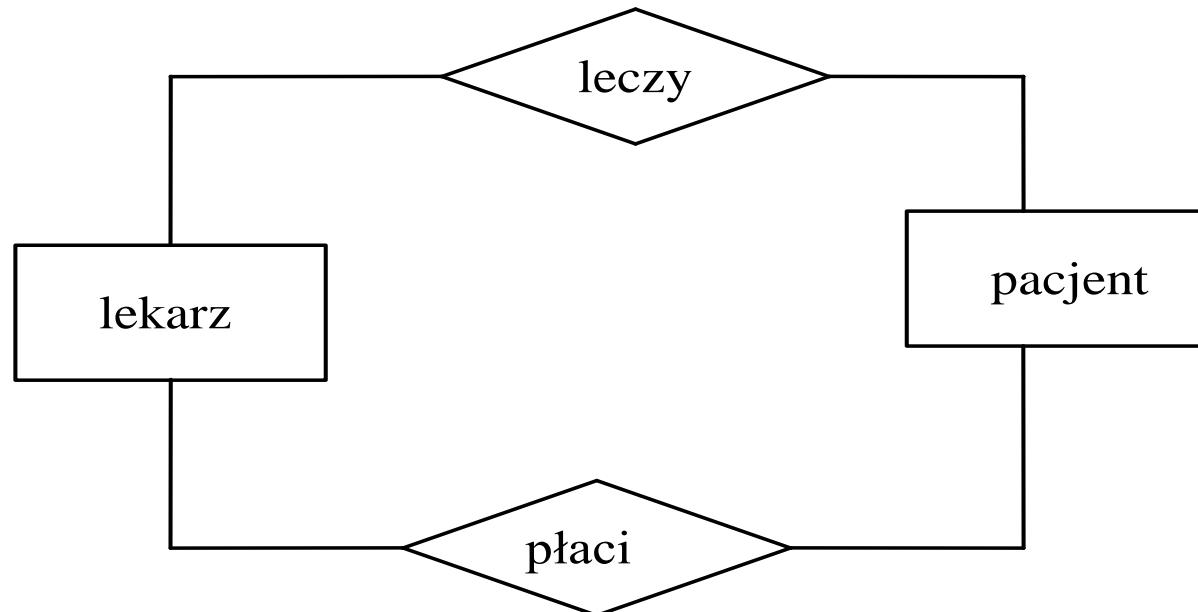
- Reprezentują zbiór powiązań między typami obiektów. Przedstawiane za pomocą rombów.

# Diagram związków encji - przykład

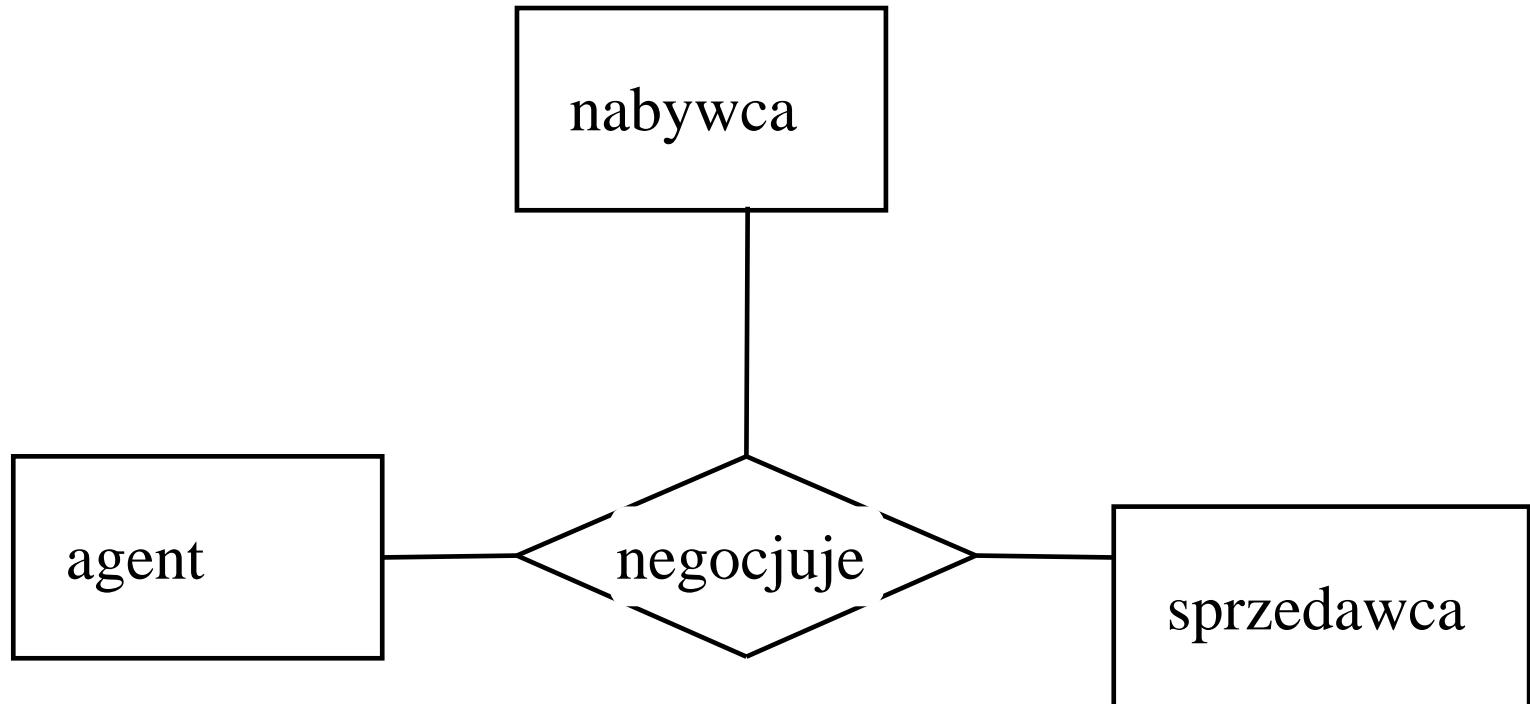


# ERD - przykład 2

Między dwoma obiektami może istnieć więcej niż jeden związek



# ERD - przykład 3



Związek między trzema encjami

# ERD - przykład 4

Możliwe jest także określenie liczności i kierunku



Związek jeden-do-wielu

# Wskaźniki asocjowanych typów

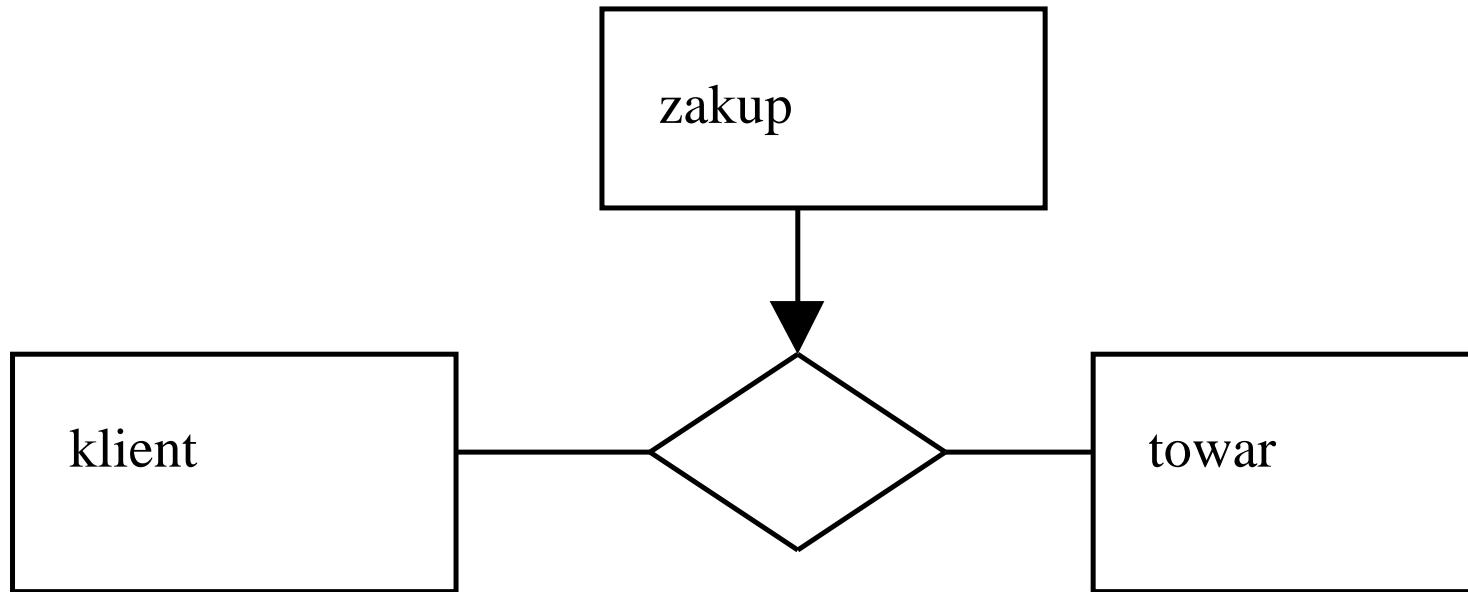
Reprezentacją związku o którym chcemy zachować pewną informację (np. czas zakupu) są

**wskaźniki asocjowanych typów obiektów.**

W obiekcie zakup będą przechowywane dane np. czas zakupu.

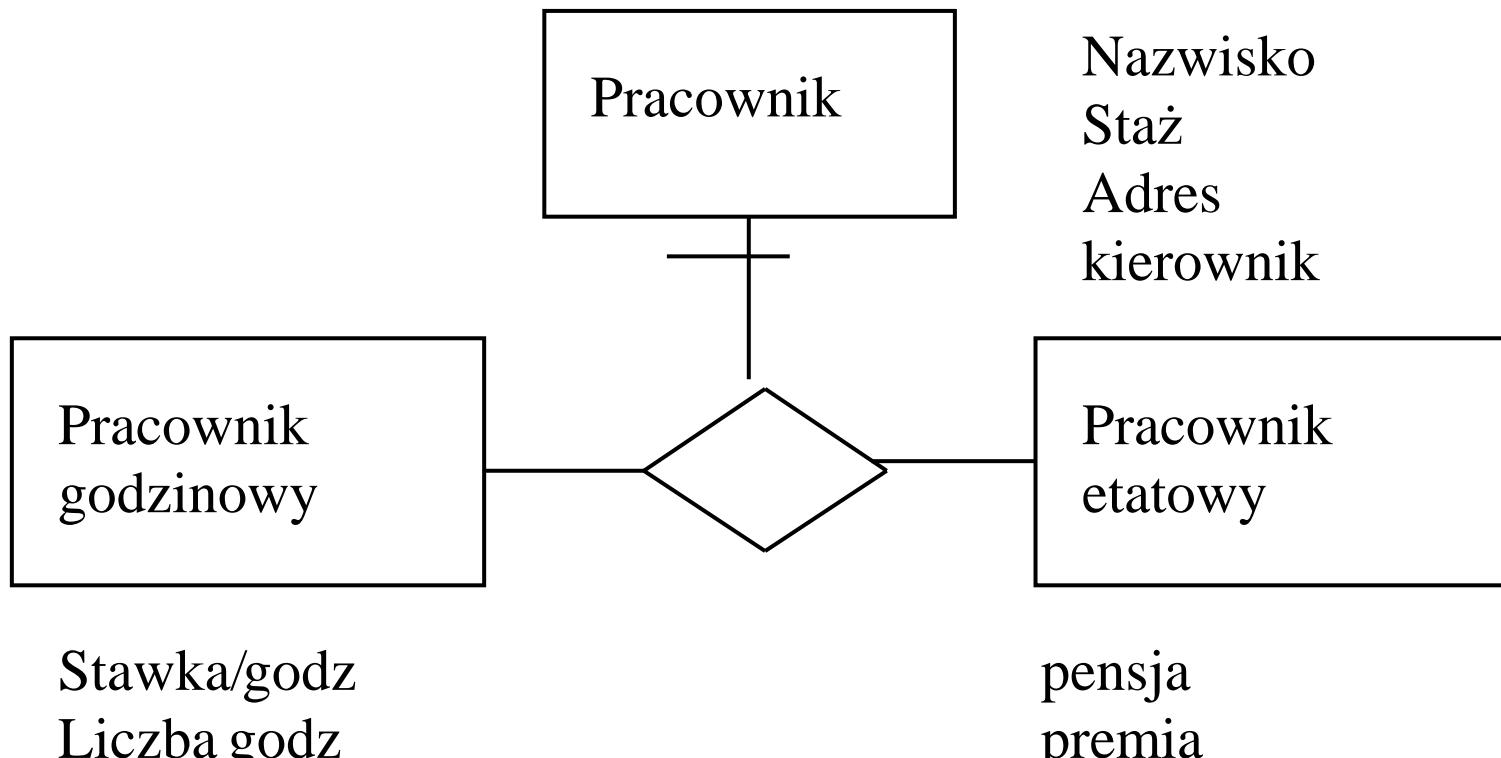
Asocjowany typ obiektu jest jednocześnie nazwą związku.

# Wskaźniki asocjowanych typów-przykład



Wskaźnik asocjowanego typu obiektu

# Wskaźniki nadtypu/podtypu



Wskaźniki nadtypu/podtypu

# Diagram przejść stanów

Diagram przejść stanów (ang. State Transition Diagram) służy do modelowania zachowania w czasie.

Dla procesów (na najniższym poziomie w hierarchii), które mają zmieniające się w czasie zachowanie można model zachowania procesu pokazać na diagramie przejść stanów.

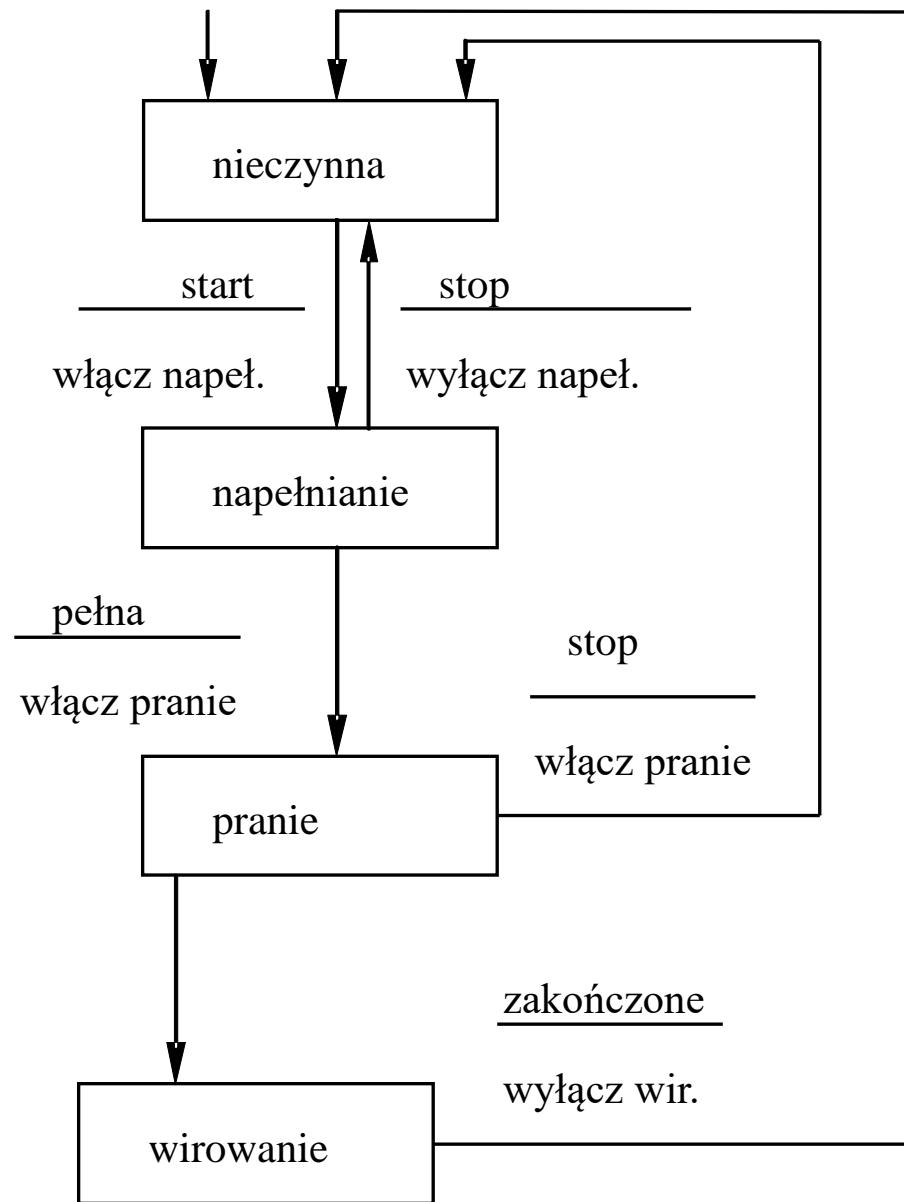


Diagram przejść stanów

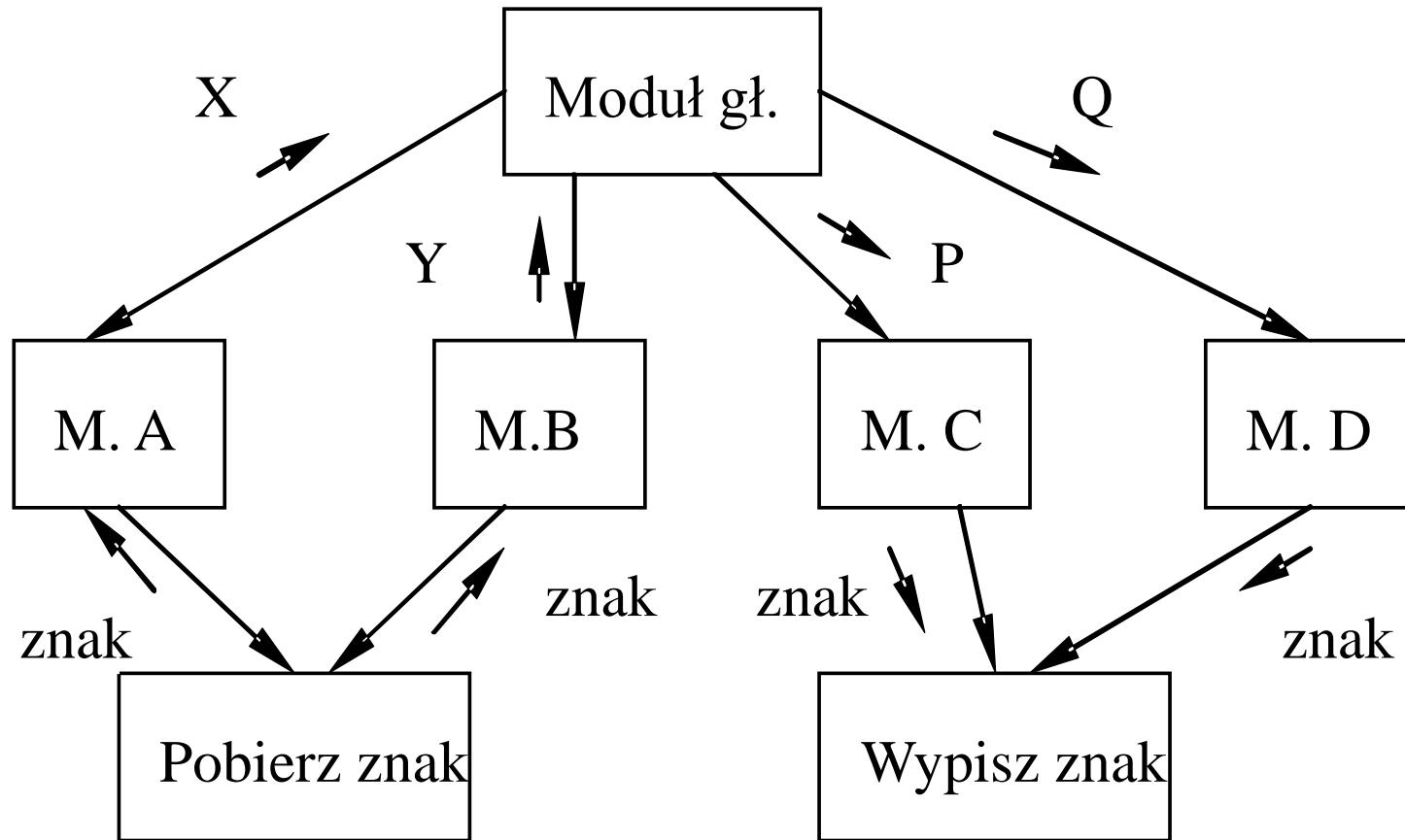
# Diagramy struktury (ang. structure charts)

Diagramy struktury (ang. structure charts) są używane do reprezentowania i modelowania struktury realizowanego oprogramowania, czyli jego architektury.

Diagram struktury pokazuje dekompozycję funkcjonalnych elementów oprogramowania.

Diagram przedstawia w postaci drzewa, moduły, podprogramy, procedury, i wywoływane przez nie elementy niższego poziomu. Można na tym diagramie pokazać przepływ sterowania lub ważnych danych.

# Diagram struktury



# Model use case

---

■ Dr hab. inż. Ilona Bluemke

# MODEL USE CASE

---

Bibliografia:

A. Cockburn - " Jak pisać efektywne przypadki użycia", WNT 2004

<http://www.usecases.org>

# Model Use Case

---

- Przedstawia system z punktu widzenia użytkownika (różnych klas użytkowników systemu).
- Modeluje zachowanie systemu w odpowiedzi na polecenia użytkownika.

Na tym etapie tworzone są diagramy „Use Case”  
Posłużą one do następnych etapów projektowania  
oraz do końcowego testowania systemu pod kątem  
spełniania wymogów użytkowników.

Określa **CO** system robi

# Model logiczny

---

Przedstawia system w postaci klas, powiązań i interakcji między nimi, zachowań obiektów należących do tych klas oraz sekwencji działań systemu.

Na tym etapie tworzy się następujące diagramy:

- klas, obiektów
- sekwencji (interakcji)
- współpracy
- przejść stanów

Określa **CO jest** w systemie, **JAK** system działa

# Model implementacyjny i wdrożeniowy

---

## Model implementacyjny

- Przedstawia system jako moduły, podsystemy, zadania. Na tym etapie powstaje diagram komponentów.

## Model wdrożeniowy (Deployment )

- Modeluje fizyczne rozmieszczenie modułów systemu na komputerach. Uwzględnia wymagania sprzętowe, obszary krytyczne.
- Na tym etapie tworzymy diagramy rozmieszczenia (deployment).

# Use Case – przykład użycia

---

Opisuje pewne zachowanie systemu, interakcje systemu ze środowiskiem zewnętrznym (człowiek, inny system – aktor)

**Ciąg wykonywanych przez system akcji, które na żądanie aktora realizują jego cele i dostarczają mierzalne wyniki.**

Reprezentuje wymagania funkcjonalne

use case – **CO** system robi a **NIE** jak to robi

# Elementy diagramów use case

---

- aktor
- przypadek użycia
- relacje (asocjacje, zależności)

## **Aktor (osoba, system zewnętrzny):**

- korzysta z systemu
- dostarcza/odbiera dane do/z systemu
- administruje systemem

# Aktor

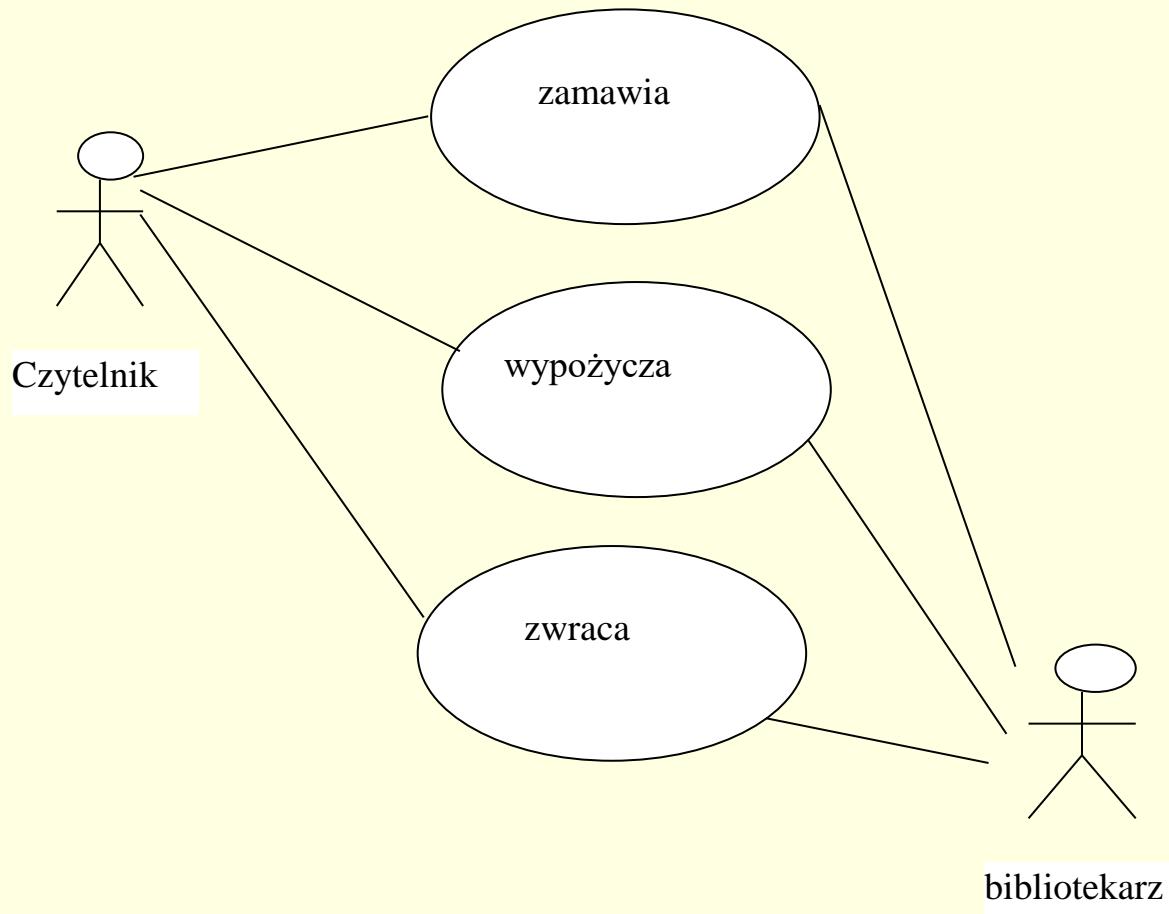
---

**Aktor:** przyczyna napędzająca przypadki użycia, sprawca zdarzeń powodujących uruchomienie przypadku użycia, odbiorca danych wyprodukowanych przez przypadki użycia.

**Aktor** – osoba, organizacja, inny system komputerowy.

**Grupa osób pełniących pewną rolę, a nie konkretna osoba**

# Przykład diagramu use case



# Działania wstępne

---

rozpoznanie dziedziny problemu

- wywiady z udziałowcami
- określenie użytkowników i ich punktów widzenia
- modelowanie procesów biznesowych
- źródła danych
- wizja systemu

# Strukturalizacja use case'ów

---

- przykład użycia może być specjalizacją innego (**generalizacja**)
- przykład użycia może być włączany jako część innego (**<<include>>**)
- przykład użycia może rozszerzać zachowanie innego ( **<<extend>>** )

# Relacje

---

- **Zależność** (dependency) między elementami, zmiana w jednym może wpływać na drugi, skierowanie pokazuje kierunek zależności .      - - - - >
- **Generalizacja** (dziedziczenie)      →
- **Asocjacja** (association) powiązanie.
  - Dwukierunkowa :      \_\_\_\_\_
  - Jednokierunkowa:      →

# Scenariusz przypadku użycia

---

Opisuje się sekwencję zdarzeń:

- jak się rozpoczyna,
- przepływ zdarzeń,
- jak się kończy,
- jakie są interakcje z aktorem.

---

Tekst nieformalny, tekst strukturalny (z warunkami początkowymi i końcowymi), pseudokod

# Format opisu przypadku użycia (1)

---

**Nazwa:** <W postaci wyrażenia czasownikowego>

**Kontekst użycia:** <Cel, normalne warunki wystąpienia>

**Zakres i poziom:** <Czy przypadek użycia dotyczy całego przedsiębiorstwa, wybranego systemu czy fragmentu oprogramowania? Na jakim poziomie szczegółowości jest opisany?>

**Aktor główny:** <Nazwa głównego aktora, opis jego roli>

**Pozostali aktorzy i udziałowcy:** <Nazwy aktorów, ich interesy>

**Wyzwalacze / Inicjacja:** <Zdarzenie powodujące rozpoczęcie przypadku użycia>

**Warunki początkowe:** <Co system zapewnia przed zezwoleniem na rozpoczęcie przypadku użycia?>

# Format opisu przypadku użycia (2)

---

## Warunki końcowe:

- **Gwarancje powodzenia:** <Warunki spełnione po pomyślnym wykonaniu głównego scenariusza przypadku użycia>
- **Minimalne gwarancje:** <Minimalne wymagania prawdziwe na końcu każdego przebiegu przypadku użycia (również niepoprawnego)>

## Główny scenariusz powodzenia / Przepływ podstawowy:

<Numer kroku> <Opis akcji>  
<Numer kroku> <Opis akcji>

## Przepływy alternatywne:

<Numer zmienionego kroku> <Opis akcji>

**Punkty rozszerzenia:** <Miejsca i warunki rozszerzeń>

**Specjalne wymagania (np. niefunkcjonalne):**

# Scenariusz główny – przykład *zamawia książkę*

---

1. System prezentuje ekran wyszukiwania.
2. **Czytelnik** wprowadza dane bibliograficzne.
3. System przeszukuje katalog i wyświetla listę tytułów.
4. **Czytelnik** przegląda listę i wybiera tytuł.
5. System wyświetla listę książek.
6. **Czytelnik** zamawia wolną książkę.
7. System wyświetla okno logowania.
8. **Czytelnik** wprowadza nazwę i hasło.
9. System autoryzuje czytelnika.
10. System potwierdza przyjęcie zamówienia.

# Scenariusze alternatywne

---

## Scenariusz **alternatywny 1 (odmowa autoryzacji)**

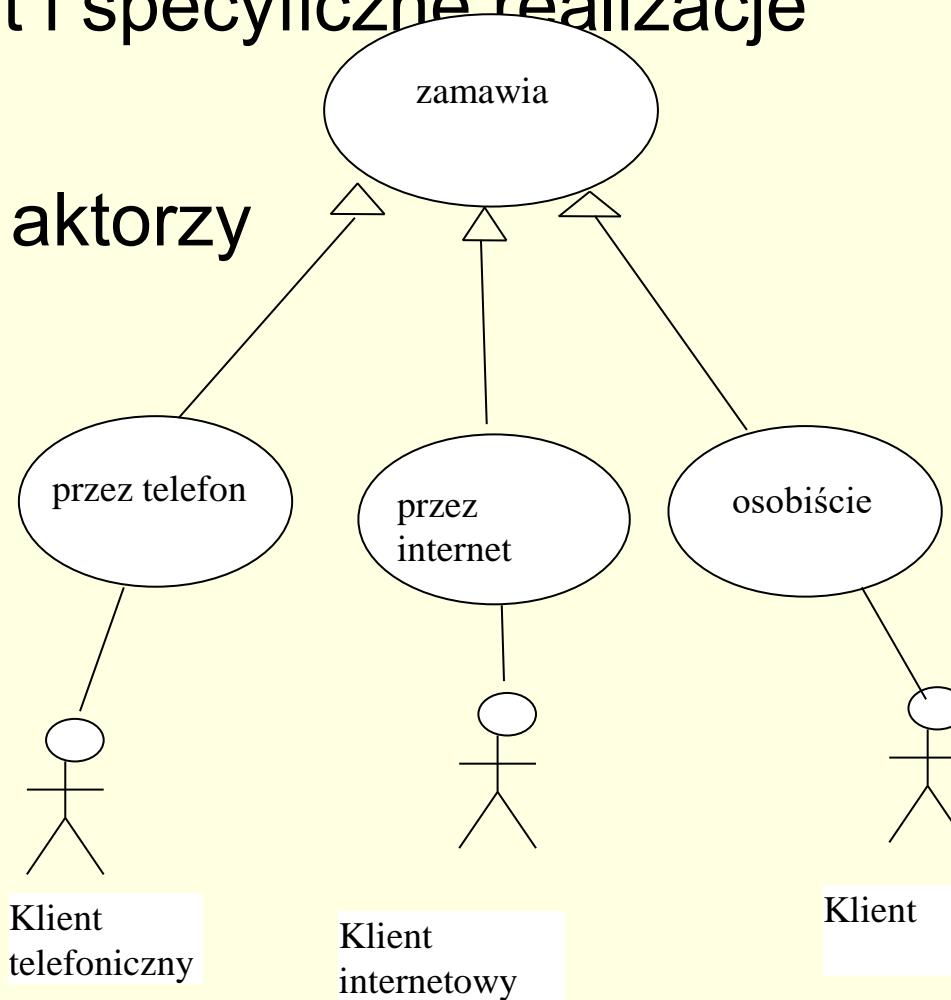
- 1 – 8 Jak w scenariuszu głównym.
- 9 System odmawia autoryzacji: powrót do kroku 7.

## Scenariusz **alternatywny 2 (brak wolnej książki)**

- 1 – 5 Jak w scenariuszu głównym.
- 6. Brak wolnej książki: czytelnik zapisuje się do kolejki.
- 7. System wyświetla okno logowania.
- 8. *Czytelnik* wprowadza nazwę i hasło.
- 9. System autoryzuje czytelnika.
- 10. System potwierdza zapisanie do kolejki.

# Generalizacja use case

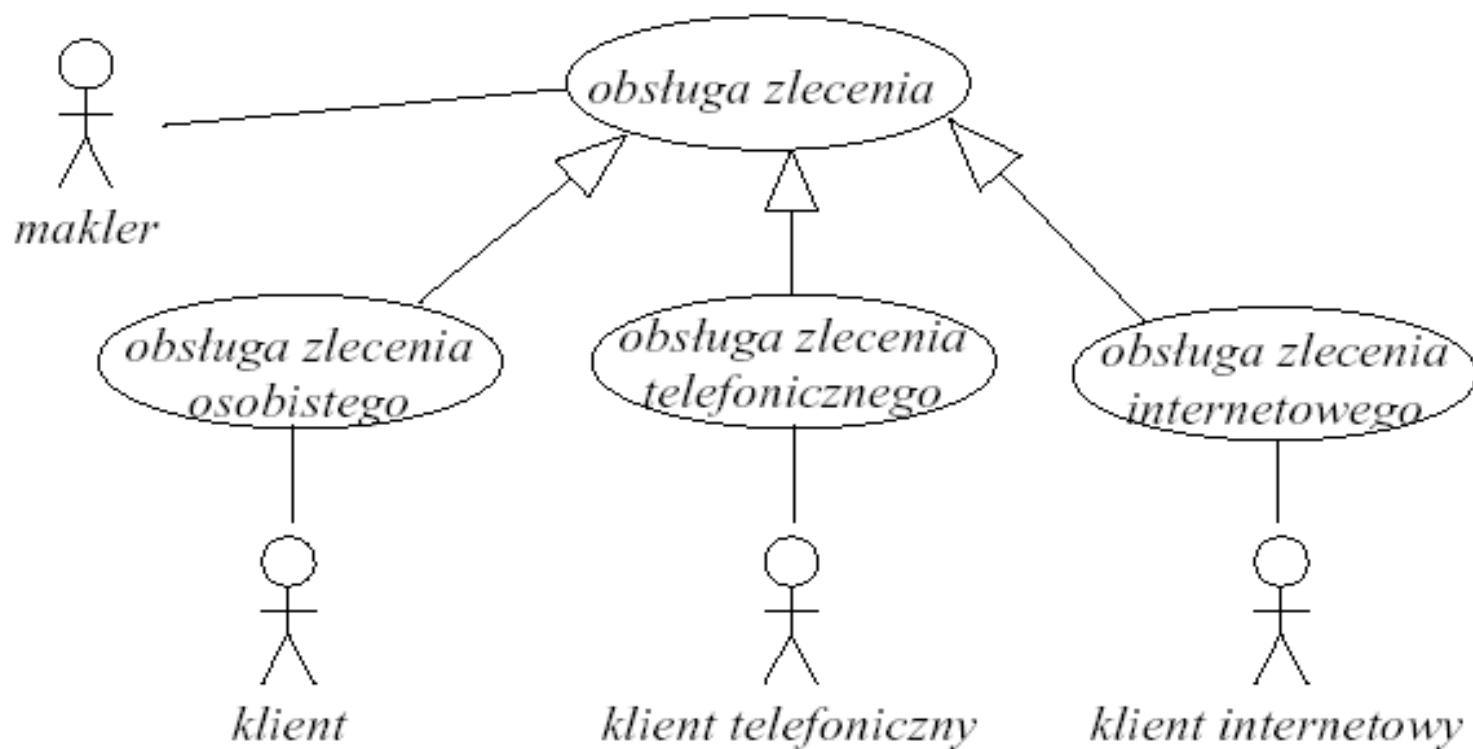
- Ogólny schemat i specyficzne realizacje
- Ten sam cel
- Mogą być różni aktorzy



## Generalizacja

- Ogólny schemat i specyficzne realizacje
- Ten sam cel
- Różni aktorzy

Przykład: biuro maklerskie.



## obsługa zlecenia

1. Makler przyjmuje zlecenie i wprowadza do systemu BM.
2. System BM blokuje środki na rachunku klienta.
3. System BM przekazuje zlecenie do systemu Warset.

## obsługa zlecenia telefonicznego

- 1.1. Makler odbiera i wprowadza do systemu BM dane klient .
  - 1.2. Makler odbiera i wprowadza do systemu hasło klienta.
  - 1.3. System potwierdza autoryzację klienta.
  - 1.4. Makler odbiera i wprowadza zlecenie do systemu BM.
- 2 – 3. Jak w przypadku generalizującym.

## obsługa zlecenia internetowego

.....

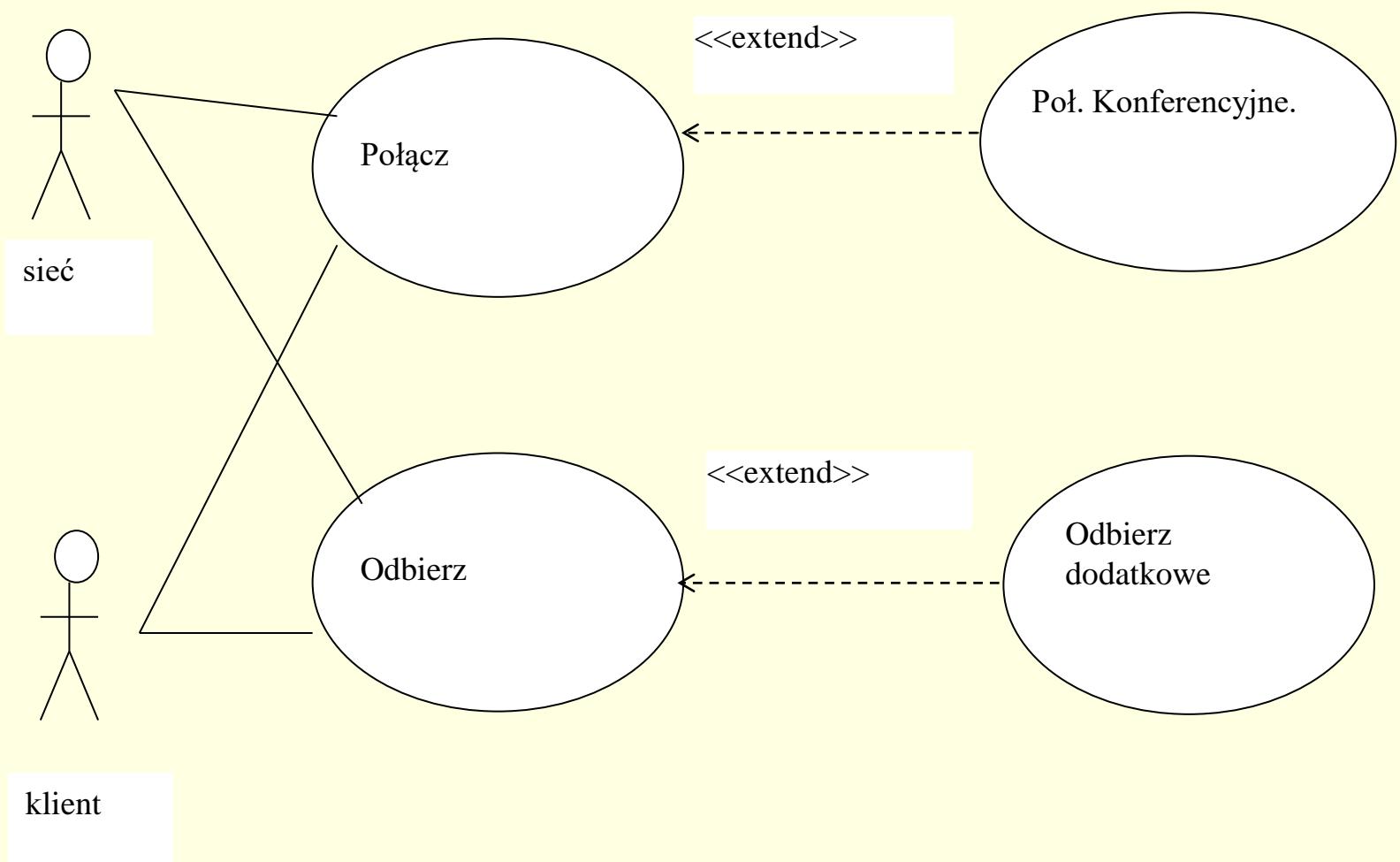
## obsługa zlecenia osobistego

# Rozszerzenie <<extend>>

---

- Typowy schemat i dodatkowe czynności
- Określone punkty rozszerzenia
- Brak odrębnych aktorów
- Przykład użycia podstawowy może wystąpić sam, ale w pewnych warunkach jego zachowanie może być rozszerzone – w pewnych punktach ( extension point), przez inny przypadek), pokazane opcjonalne zachowanie systemu
- Wariant, opcja

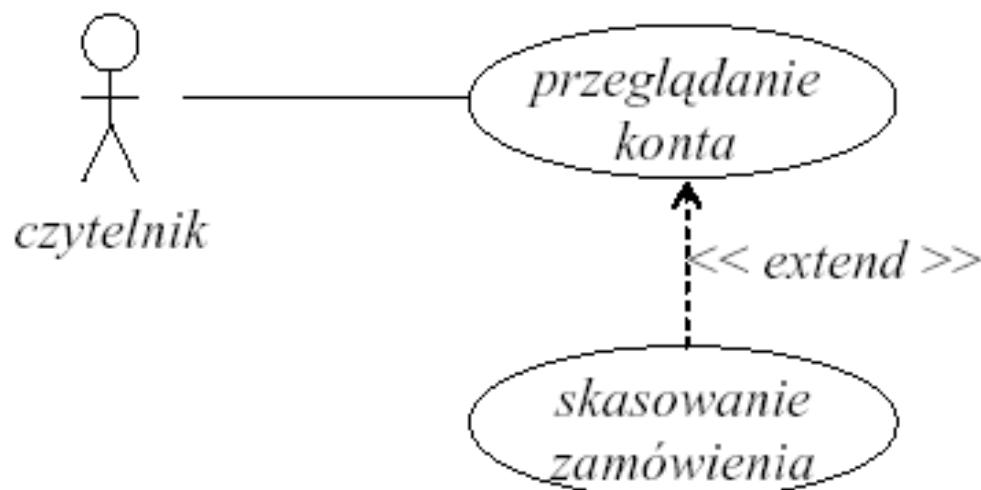
# Przykład <<extend>>



## Rozszerzenie

- Typowy schemat i dodatkowe czynności
- Określone punkty rozszerzenia
- Brak odrębnych aktorów

Przykład: system biblioteczny.



## przeglądanie konta

1. *Czytelnik* wybiera opcję przeglądania konta.
2. System wyświetla okno logowania.
3. *Czytelnik* wprowadza nazwę i hasło.
4. System autoryzuje czytelnika.
5. System wyświetla stan konta (***kasowanie***).
6. *Czytelnik* wybiera następną opcję.

## skasowanie zamówienia

wstawić w punkcie rozszerzenia ***kasowanie***:

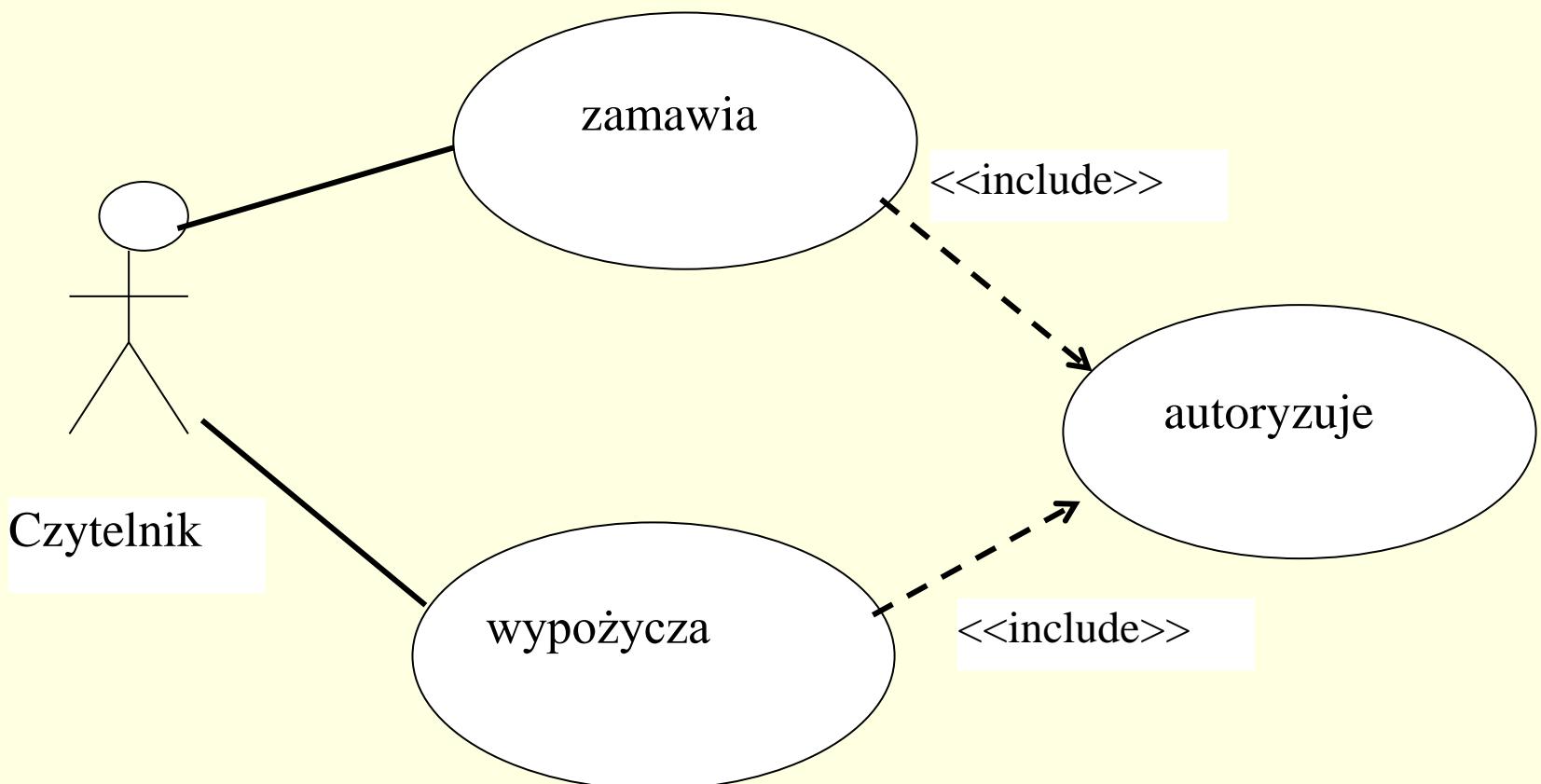
1. *Czytelnik* kasuje zamówienie.
2. System potwierdza skasowanie zamówienia.

# Zawieranie <<include>>

---

- Przykład użycia włącza zachowanie innego, przykład „included” nie może być samodzielny
- Wyodrębnienie fragmentu przypadku użycia
- Możliwe fragmenty wspólne
- Brak odrębnych aktorów

# Przykład <<include>>



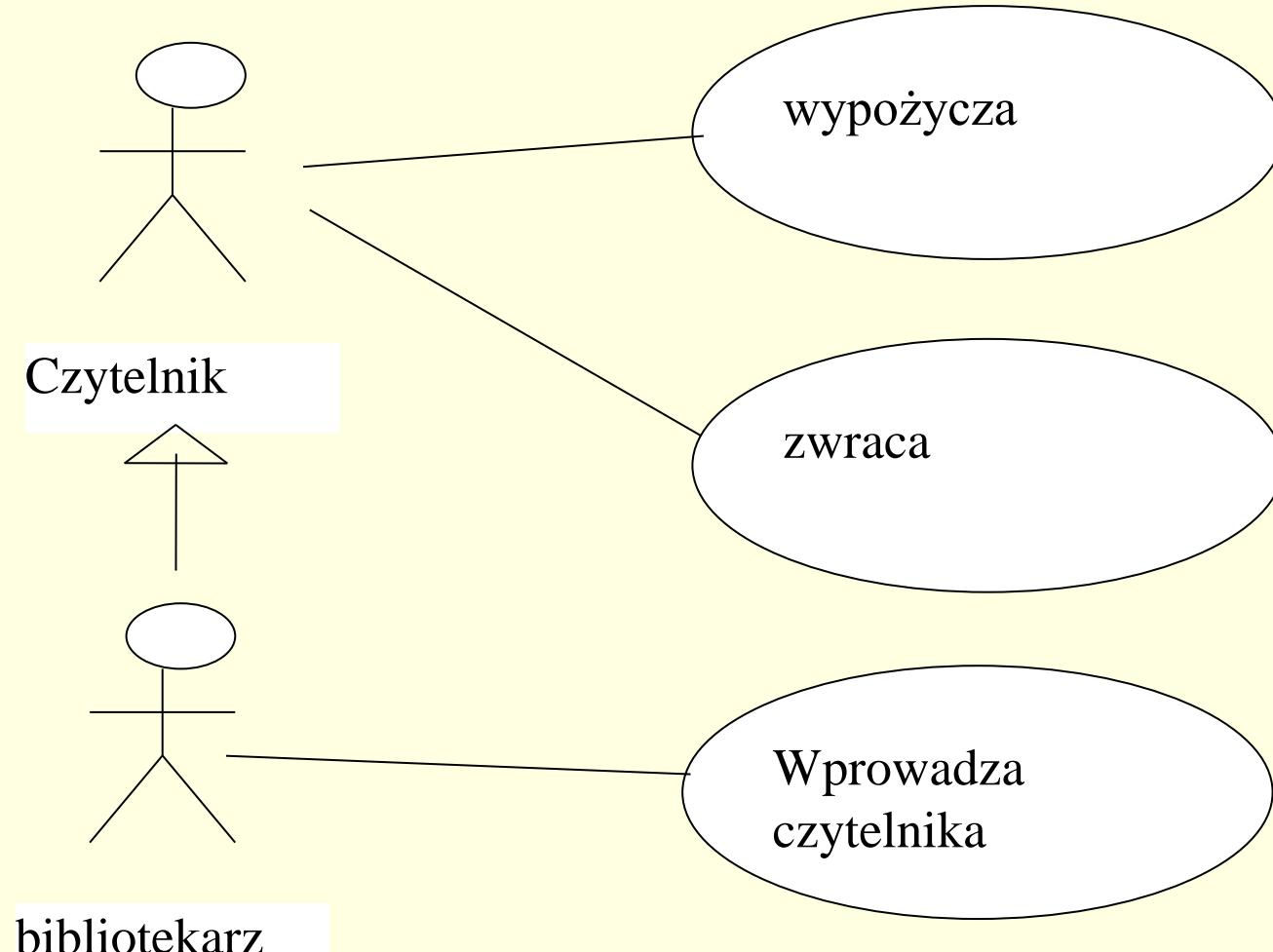
## zamówienie książki

1. System prezentuje ekran wyszukiwania.
2. *Czytelnik* wprowadza dane bibliograficzne.
3. System przeszukuje katalog i wyświetla listę pozycji (tytułów).
4. *Czytelnik* przegląda listę i wybiera pozycję.
5. System wyświetla listę książek.
6. *Czytelnik* zamawia wolną książkę.
7. **Wykonaj autoryzację czytelnika.**
8. System potwierdza przyjęcie zamówienia.

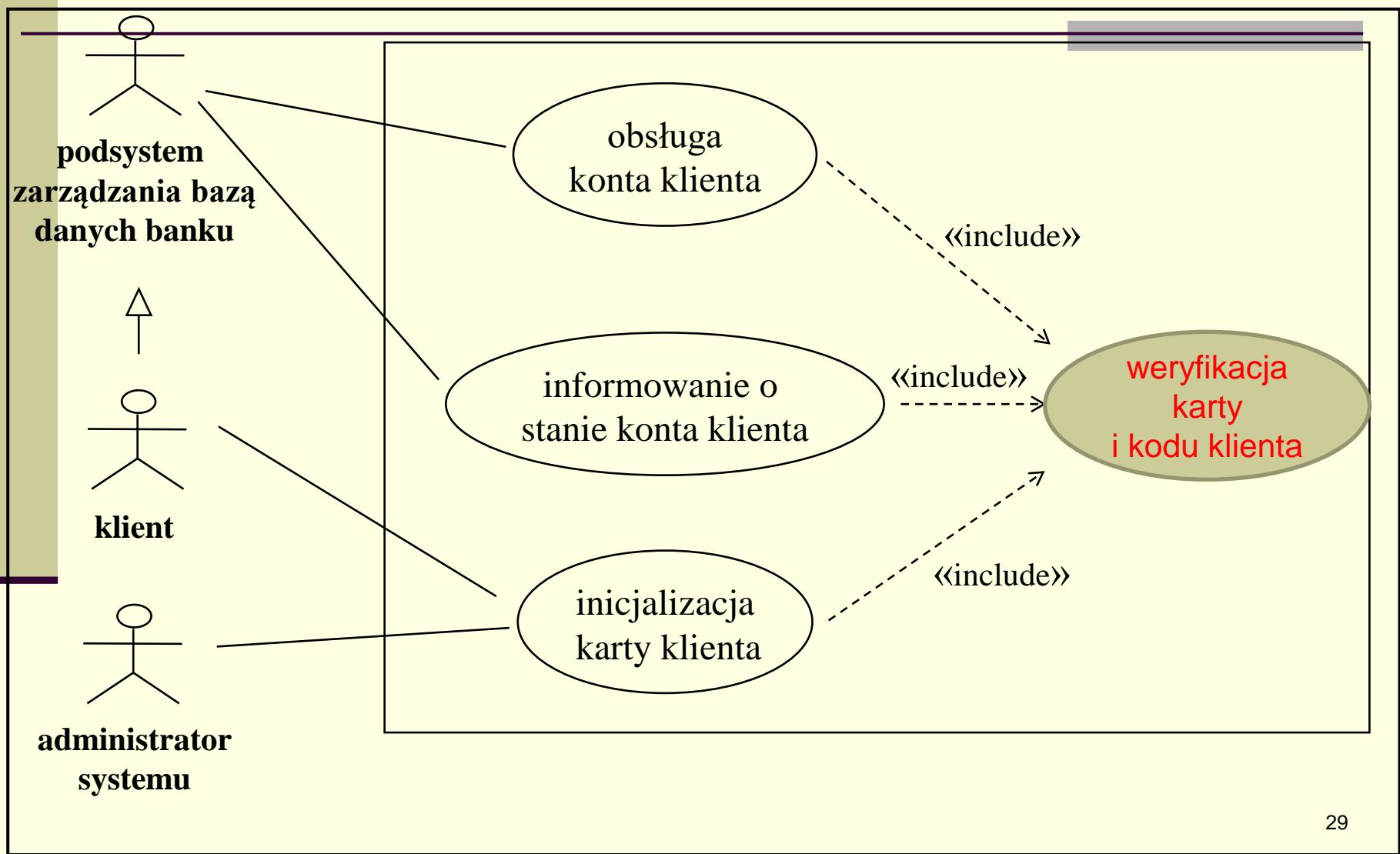
## autoryzacja czytelnika

1. System wyświetla okno logowania.
2. *Czytelnik* wprowadza nazwę i hasło.
3. System autoryzuje czytelnika.

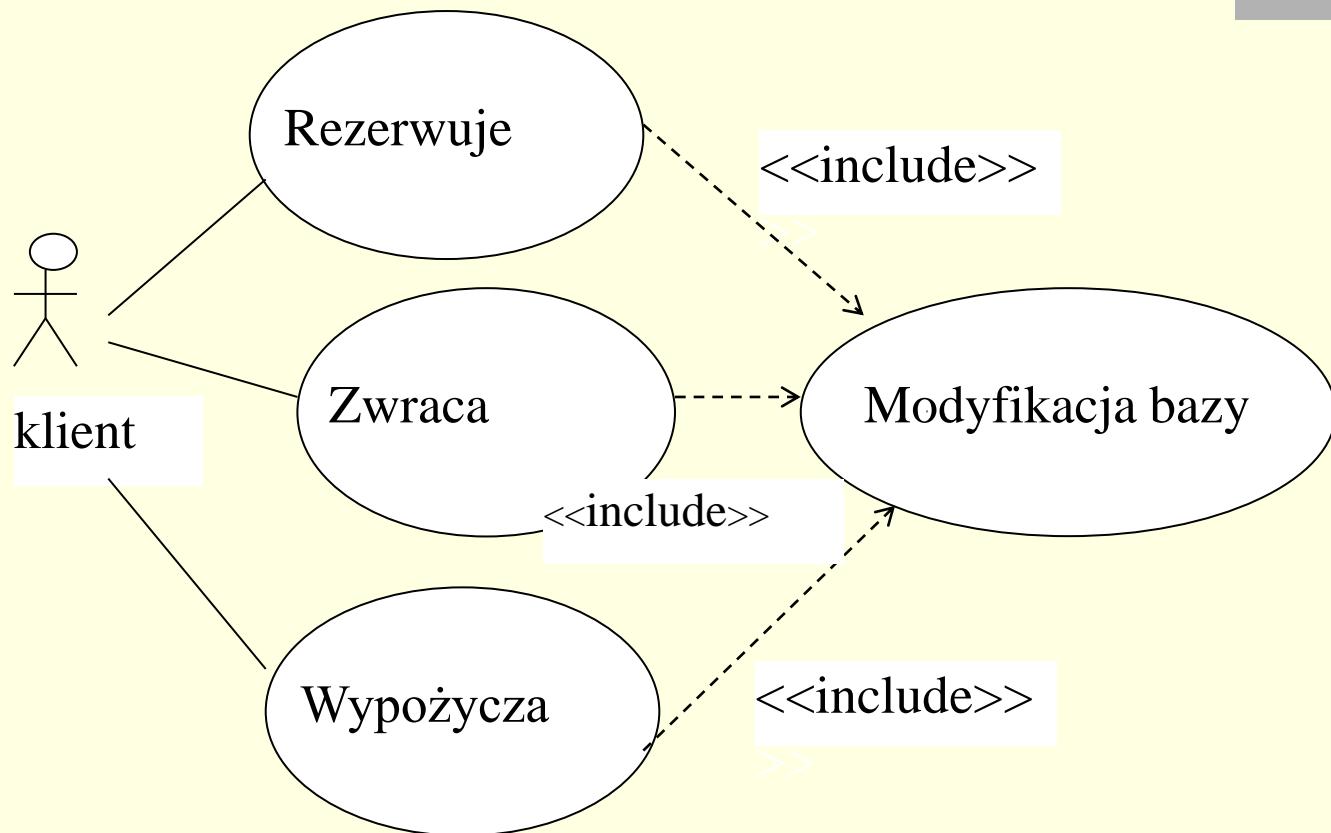
# Generalizacja aktorów



# Użycie generalizacji aktorów



# Diagram przypadków użycia z <<include>>



# Heurystyka tworzenia przypadków użycia

---

## 4 kroki wg Cockburna

1. Zidentyfikuj aktorów i ich cele.
2. Napisz główny scenariusz powodzenia
3. Zidentyfikuj i wylistuj możliwe rozszerzenia  
(zwłaszcza możliwe błędy)
4. Opisz jak system obsługuje każde  
rozszerzenie (w tym każdy błąd)

# Zalety i wady przypadków użycia

---

+

- przedstawiają wymagania funkcjonalne w prostym do czytania formacie tekstowym

-

- pokazują tylko wymagania funkcjonalne – bez niefunkcjonalnych
- projekt nie jest tylko tworzony w kategoriach przypadków użycia

# Diagram przypadków użycia – podsumowanie

---

- Przedstawia wymagane zachowanie systemu
- Modeluje kontekst systemu

Czynności:

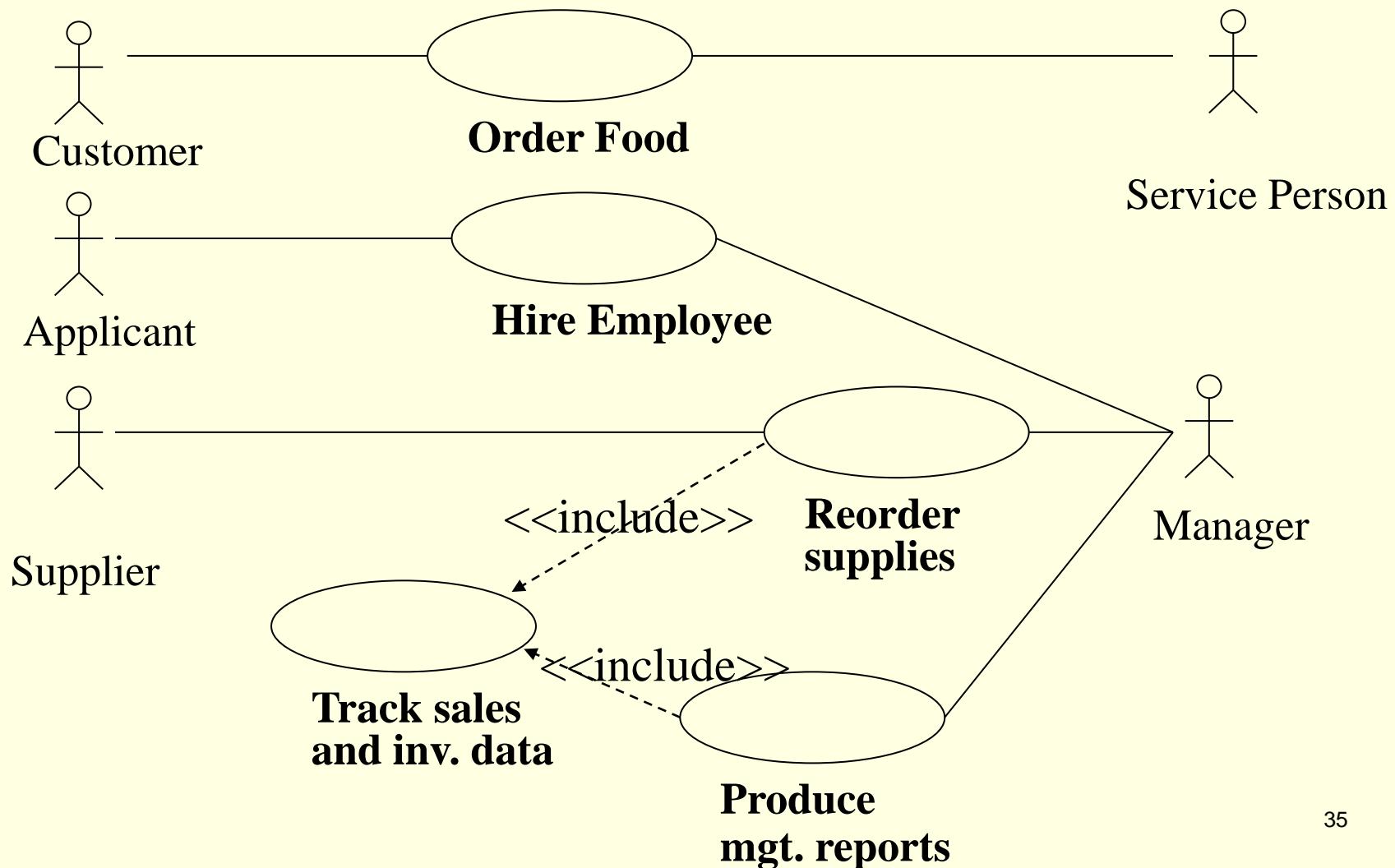
1. Identyfikacja aktorów
2. Określenie przypadków użycia
  - zadania aktora (np. wprowadzanie danych, prezentacja danych, utrzymanie systemu)
3. Uporządkowanie modelu (generalizacja, rozszerzenie, zawieranie)
4. Dokumentacja modelu

# Diagram przypadków użycia – podsumowanie -2

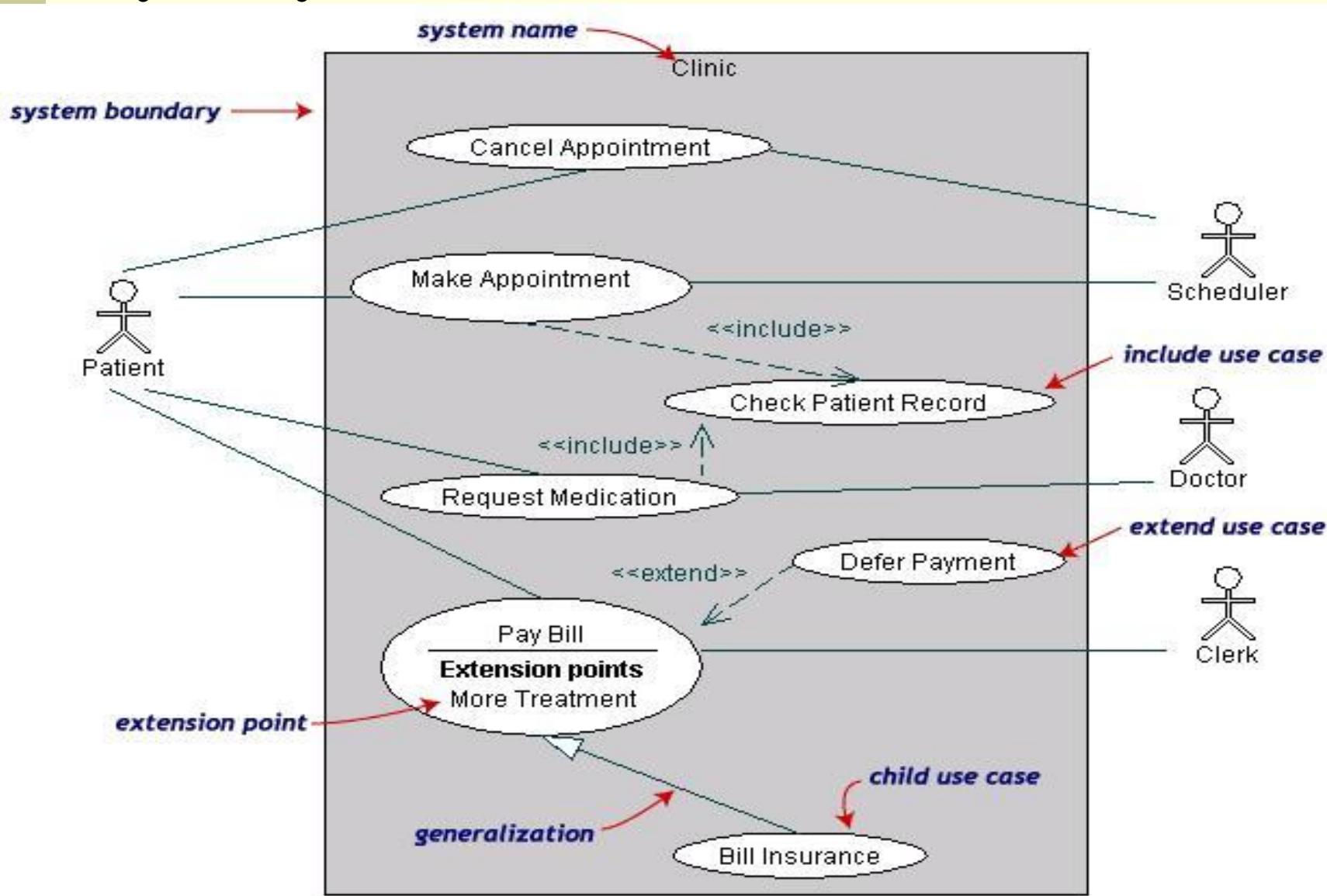
---

- Przypadki użycia **nie dotykają problemów projektowych**:
  - struktury danych
  - współbieżności operacji
  - struktury programu
- Diagram przypadków użycia **nie pokazuje kolejności**, w jakiej przypadki mogą być wykonywane. Właściwą kolejność określają diagramy realizowane w modelu logicznym.

# Przykłady przypadeków użycia

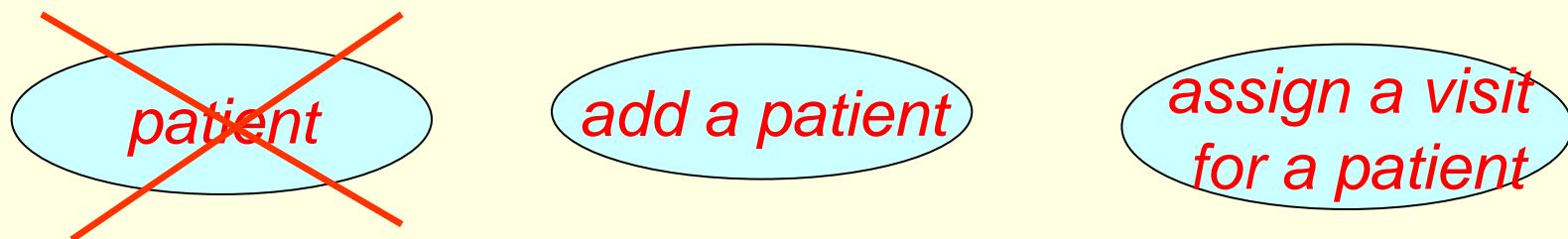


# Przykłady

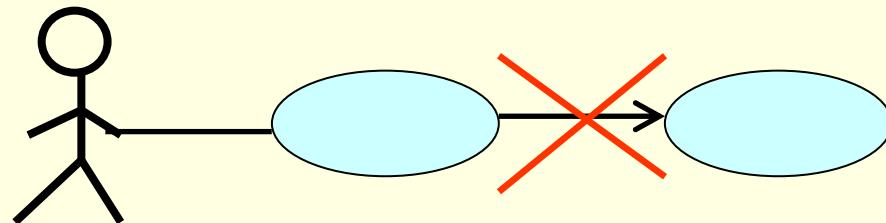


# Use Case model – typowe błędy

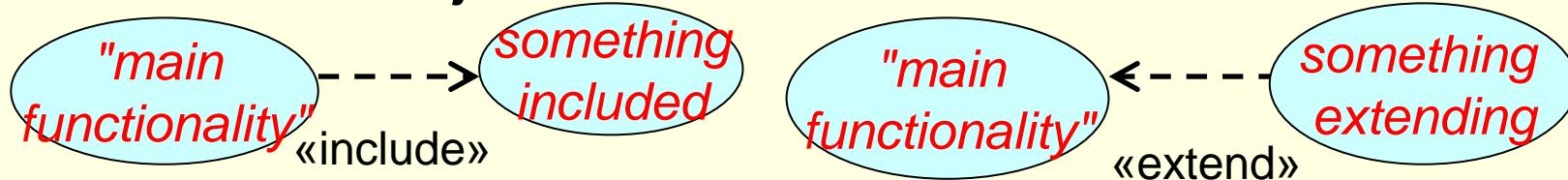
Złe nazwy (rzeczowniki zamiast czasowników):



Niewłaściwe użycie  
relacji



Kierunki relacji



# Use cases vs. internal features

---

- consider software to run a cell phone:



## Use Cases

- call someone
- receive a call
- send a message
- memorize a number

## Internal Functions

- transmit / receive data
- energy (battery)
- user I/O (display, keys, ...)
- phone-book mgmt.

*Point of view: user*

*Point of view: developer / designer*

# Diagramy klas

---

Dr hab. inż. Ilona Bluemke

# Plan wykładu

- Klasa
- Relacje między klasami (asocjacje, generalizacje, agregacje)
- Ograniczenia relacji
- Klasy interfejsowe

# Diagramy klas

- Przedstawiają typy obiektów występujących w aplikacji i powiązania między nimi.
- Jest to model statyczny dziedziny problemu.
- Określają **CO JEST** w systemie potrzebne do realizacji wymaganych funkcji (model use case).

# Klasa

Osoba

nazwa

Wiek

atrybuty

Adres

Data\_urodzenia

Podaj\_wiek

operacje

Daj\_adres

Ustaw\_adres

# Graficzna reprezentacja klasy

Osoba

Osoba

Podaj\_wiek  
Podaj\_adres  
Ustaw\_adres

Osoba

wiek  
adres  
data\_urodzenia

# Klasa

- Opisuje grupę obiektów o podobnych właściwościach (atrybutach), zachowaniach (operacjach), wspólnych relacjach z innymi obiektami, identycznym znaczeniu.
- Jest uogólnieniem zbioru obiektów. Obiekt jest instancją klasy.
- Klasy pomagają w abstrakcji, generalizacji problemu.

# Atrybuty

- wartości danych przechowywane w obiekcie określonej klasy, są unikalne w klasie, różne klasy mogą mieć takie same nazwy atrybutów.
- są określonego typu
- mogą mieć wartość domniemaną.
- może mieć określony **zakres dostępności** :
  - prywatny (private) -name : tylko klasa
  - publiczny (public) +name : wszystkie klasy
  - chroniony (protected) #name : klasa, podklasy

## Atrybut -2

może być:

- implementacyjny (implementation) ?name
  - wyprowadzalny (derived) /name  
daje się wyprowadzić z innych atrybutów
  - kluczowy (key) \*name  
jednoznacznie identyfikuje instancję klasy lub połączenie

**Właściwości atrybutu** definiują dodatkowe cechy

- *changeable* modyfikowalny bez ograniczeń
  - *addOnly* brak możliwości usuwania
  - *frozen* stały

# Operacja

Operacja jest funkcją lub transformacją, którą można zastosować **do** lub może być stosowana **przez** obiekty tej klasy.

Wszystkie obiekty danej klasy posiadają te same operacje.

**Metoda** - implementacja operacji dla klasy.

# Operacja -2

może mieć **argumenty parametryzujące** (ale nie wpływają one na wybór metody).

- może **zwracać wynik** (określonego typu).
- może mieć określony **zakres widoczności**, określający dostępność operacji, jakie inne klasy mogą z niej korzystać.
  - prywatna (private)                   **-name :**               tylko klasa
  - publiczna (public)                  **+name :**               wszystkie klasy
  - chronione (protected)              **#name :**               klasa, podklasy
  - implementacyjny (implementation)   **?name**  
zasięg będzie określony w implementacji

**Specyfikacja operacji klasy w języku UML:**

**widoczność nazwa ( lista-argumentów ) : typ { właściwości }**

# lista-argumentów

określa argumenty operacji:

**sposób-przekazywania nazwa : typ = wartość-domyślna**

gdzie:

**sposób-przekazywania**

- in                        przekazanie przez wartość
- out                      przekazanie przez referencję
- inout                    przekazanie przez referencję

**właściwości** definiują dodatkowe cechy operacji:

- *leaf*                    operacja nie jest polimorficzna
- *isQuery*                operacja nie zmienia atrybutów
- *sequential*             wymaga sekwencyjnego działania obiektu
- *guarded*                wykonywana rozłącznie z innymi
- *concurrent*            wykonywana współbieżnie z innymi

# Relacje

- Połączenie - wiąże ze sobą obiekty.
- Fizyczne lub koncepcyjne połączenie między obiektami.
- Obiekt współpracuje z innymi obiektami z którymi jest połączony.
- Poprzez link obiekt klient prosi o usługę inny obiekt lub poprzez link może sterować innym obiektem, wywoływać operacje, otrzymywać rezultaty operacji.

# Powiązanie - (association)

- Modeluje relacje takie jak: dotyczy, komunikuje się z, obsługuje
- Asocjacja dwukierunkowa:

nazwa

---

- Asocjacja jednokierunkowa

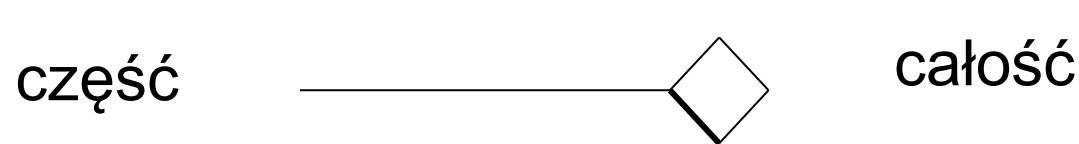
nazwa



# Agregacja

modeluje relacje takie jak:

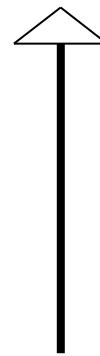
- składa się z,
- jest zbudowany z,
- zawiera



# Generalizacja -dziedziczenie

modeluje dziedziczenie klas

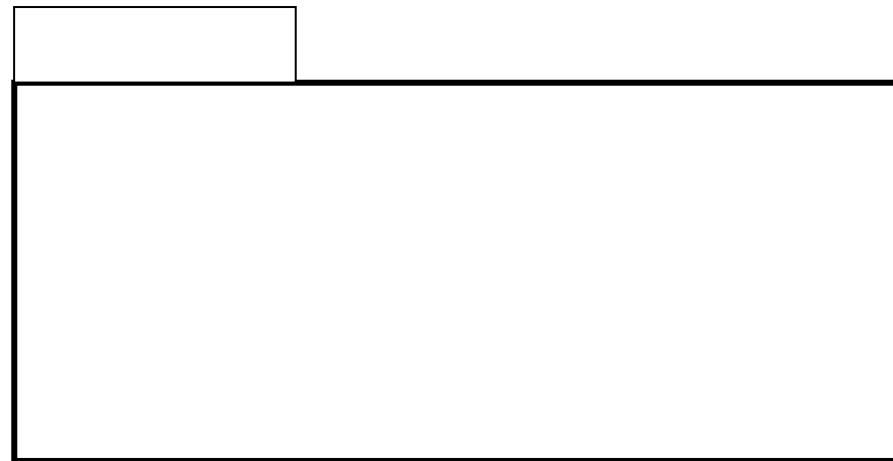
Klasa ogólna



Klasa specjalizowana

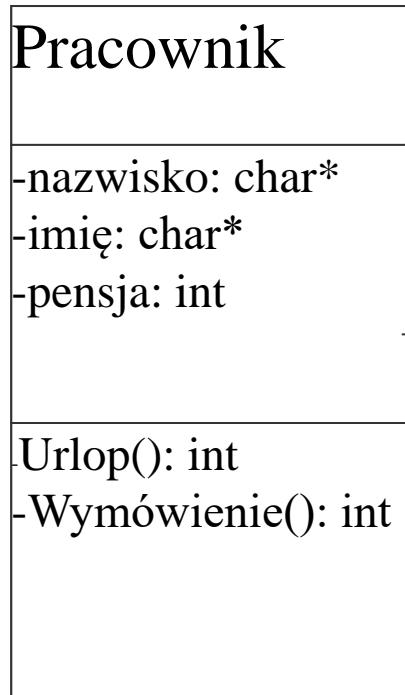
# Pakiet

- Grupuje części diagramu klas
- Służy do porządkowania i hierarchizacji diagramów

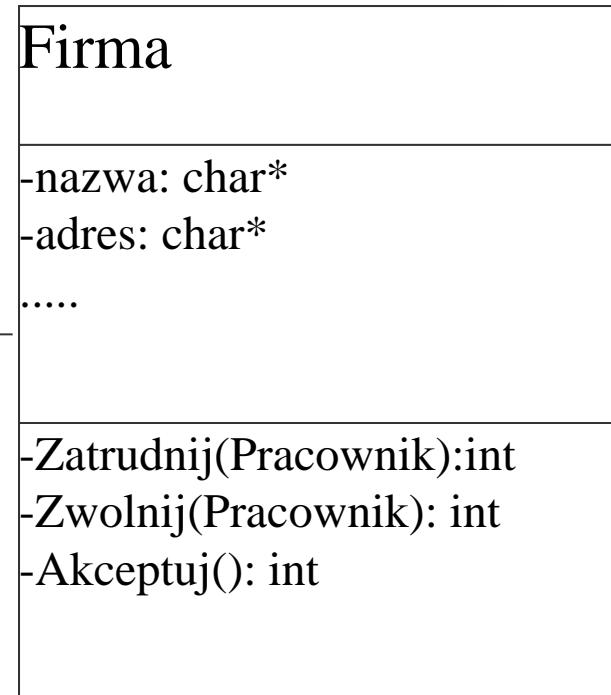


# przykład

pracownik „pracuje” dla firmy



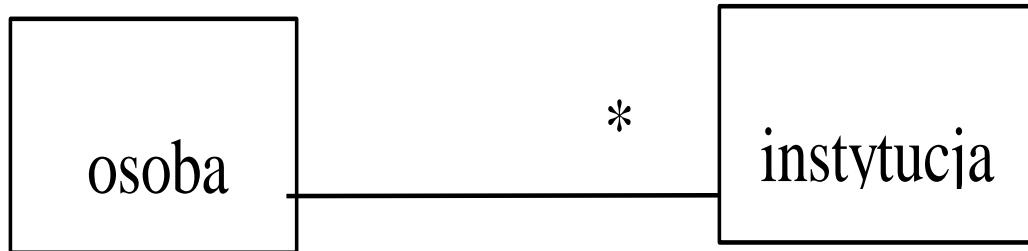
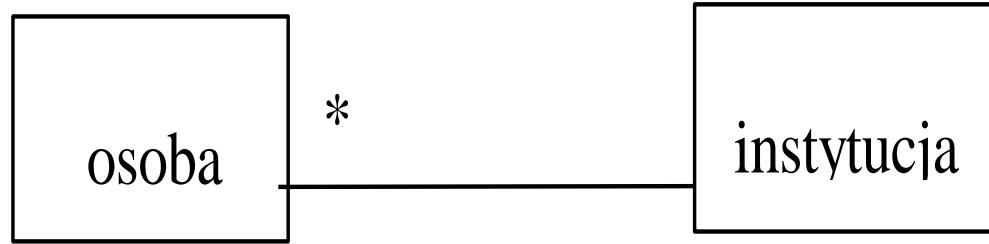
\*                    pracuje



# powiązania

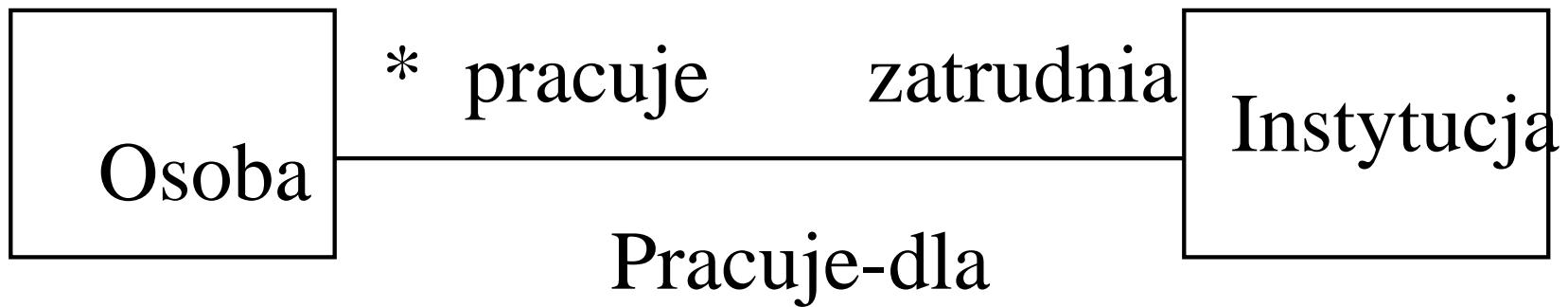
- czasowniki w opisie problemu.
- Nazwa dwukierunkowego powiązania jest zwykle czytana w określonym kierunku np. pracownik pracuje dla firmy, firma zatrudnia pracownika.
- Powiązania mogą być **wielowartościowe**, zależnie od założeń modelu. **Krótność** określa ile obiektów danej klasy może być w relacji z obiektem innej klasy np.:
  - 1      dokładnie jeden
  - 0 ..1    zero lub jeden
  - M ..N    od M do N (liczby naturalne)
  - \*        od zera do dowolnej liczby całkowitej
  - 1 .. \*    od jednego do dowolnej liczby całkowitej

# Przykład – różnice znaczeniowe

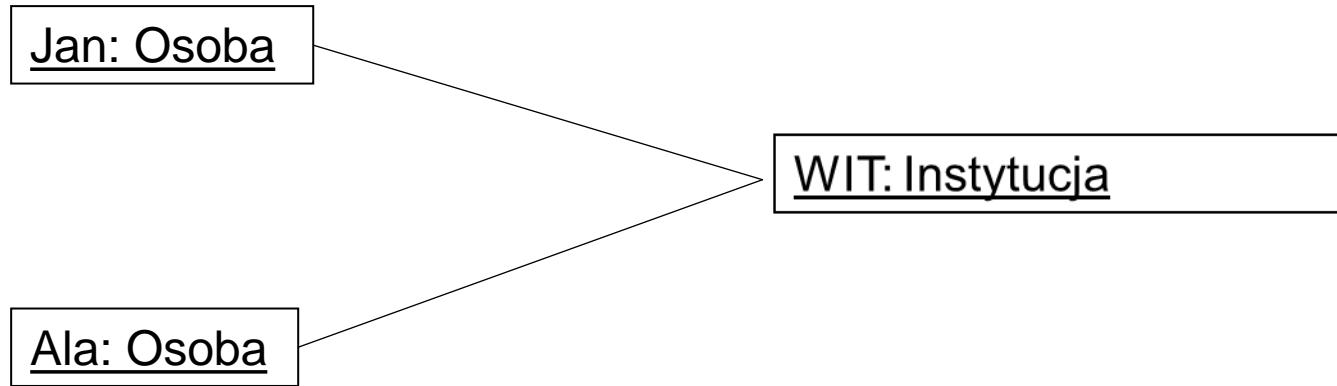


# role

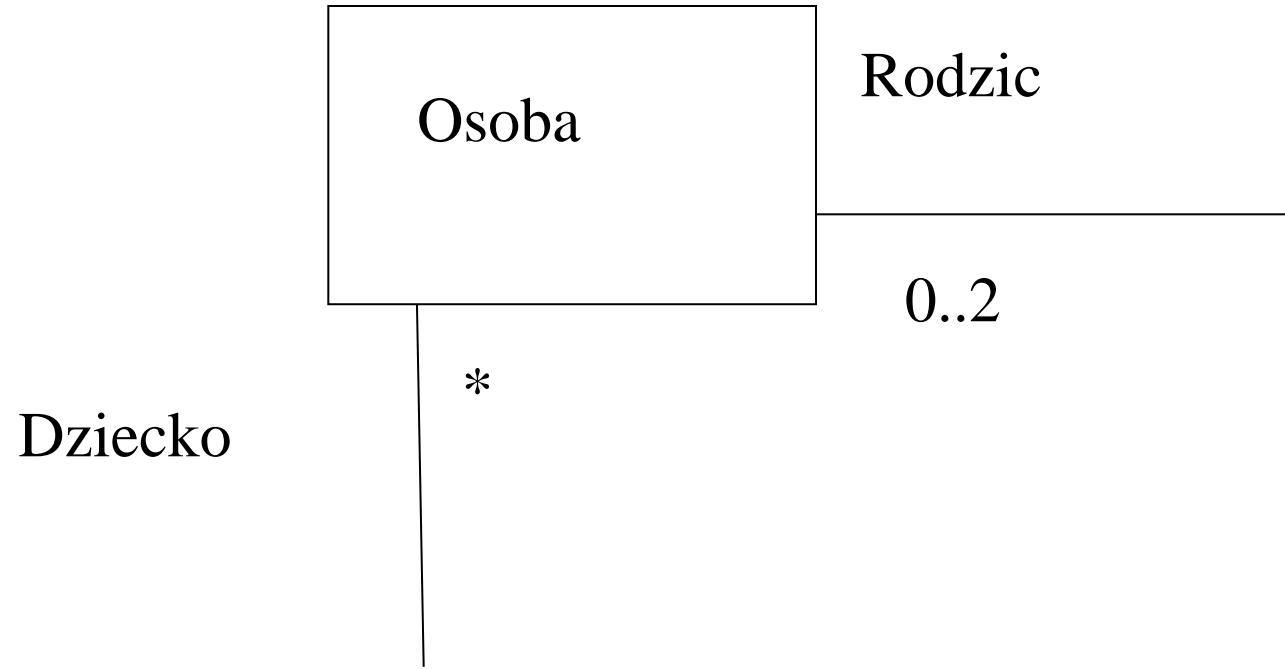
- Można określić role obiektów pełnione w powiązaniu np.



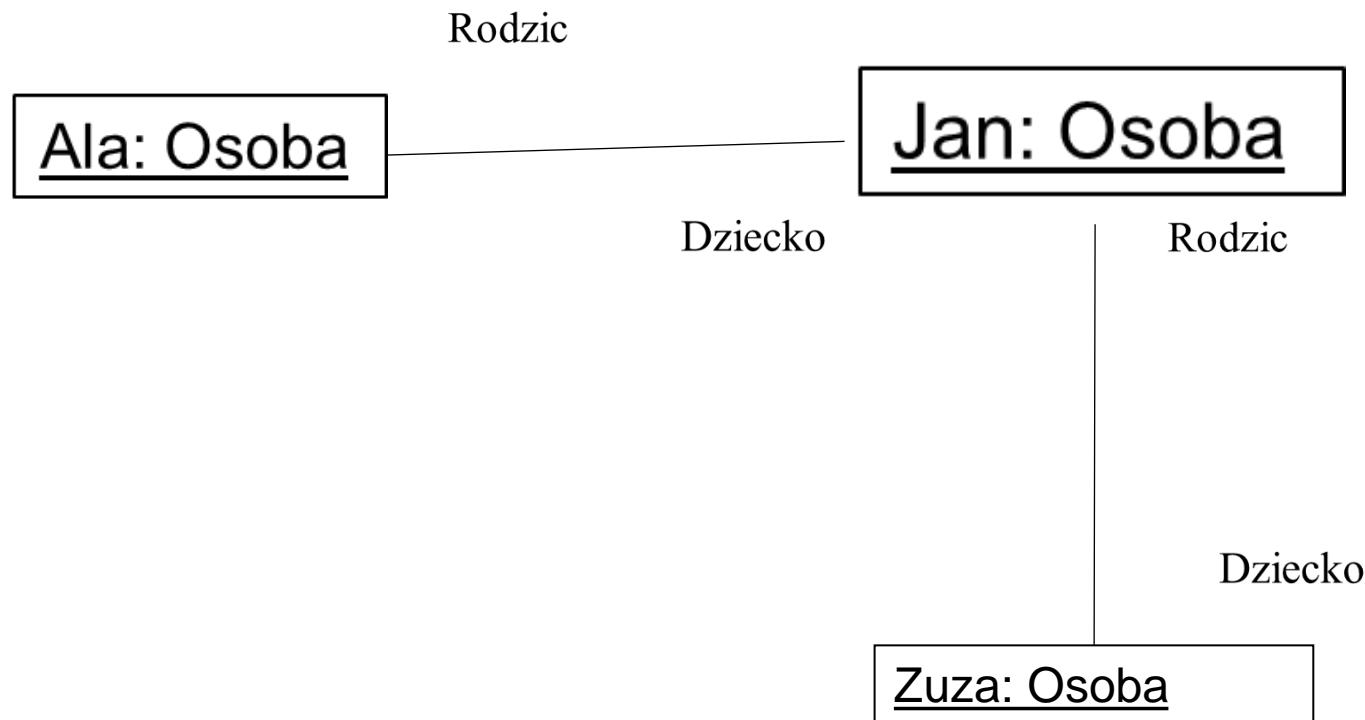
# Diagram obiektów



# Asocjacja zwrotna

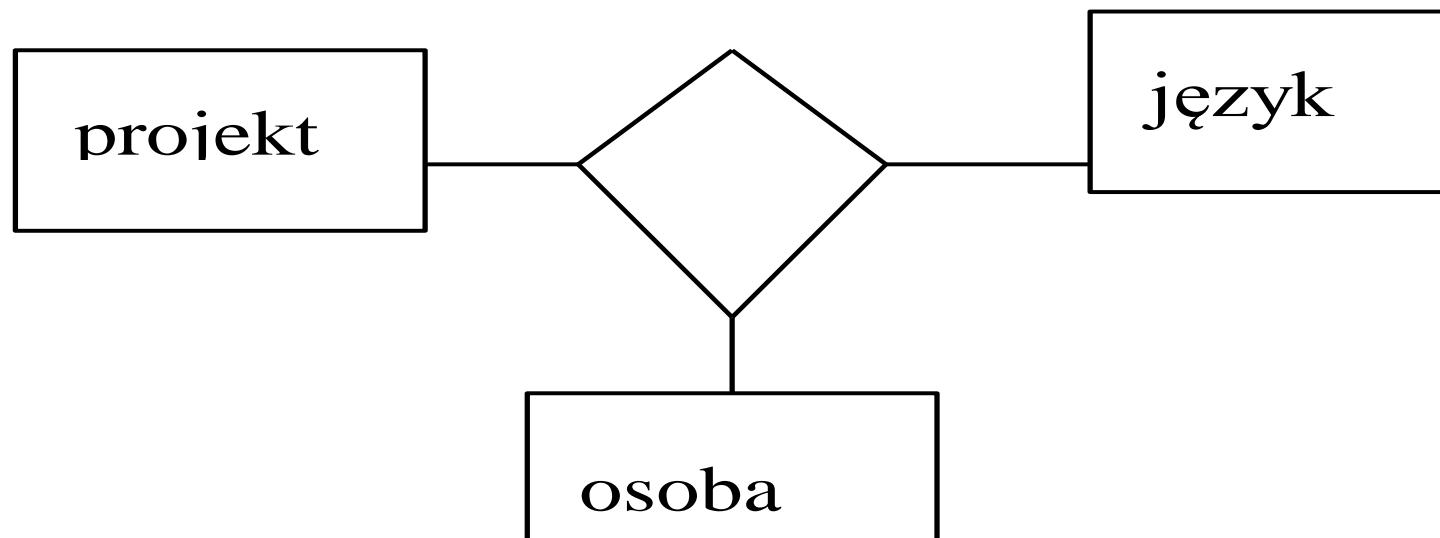


# Diagram obiektów - Asocjacja zwrotna

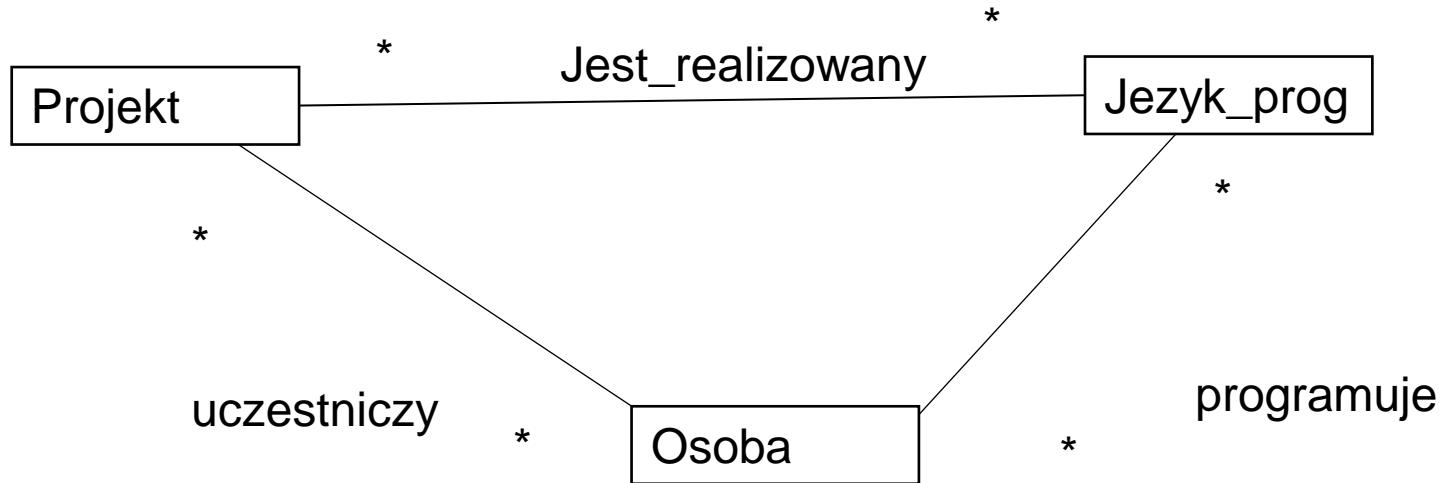


# Powiązania ternarne (ang. ternary association)

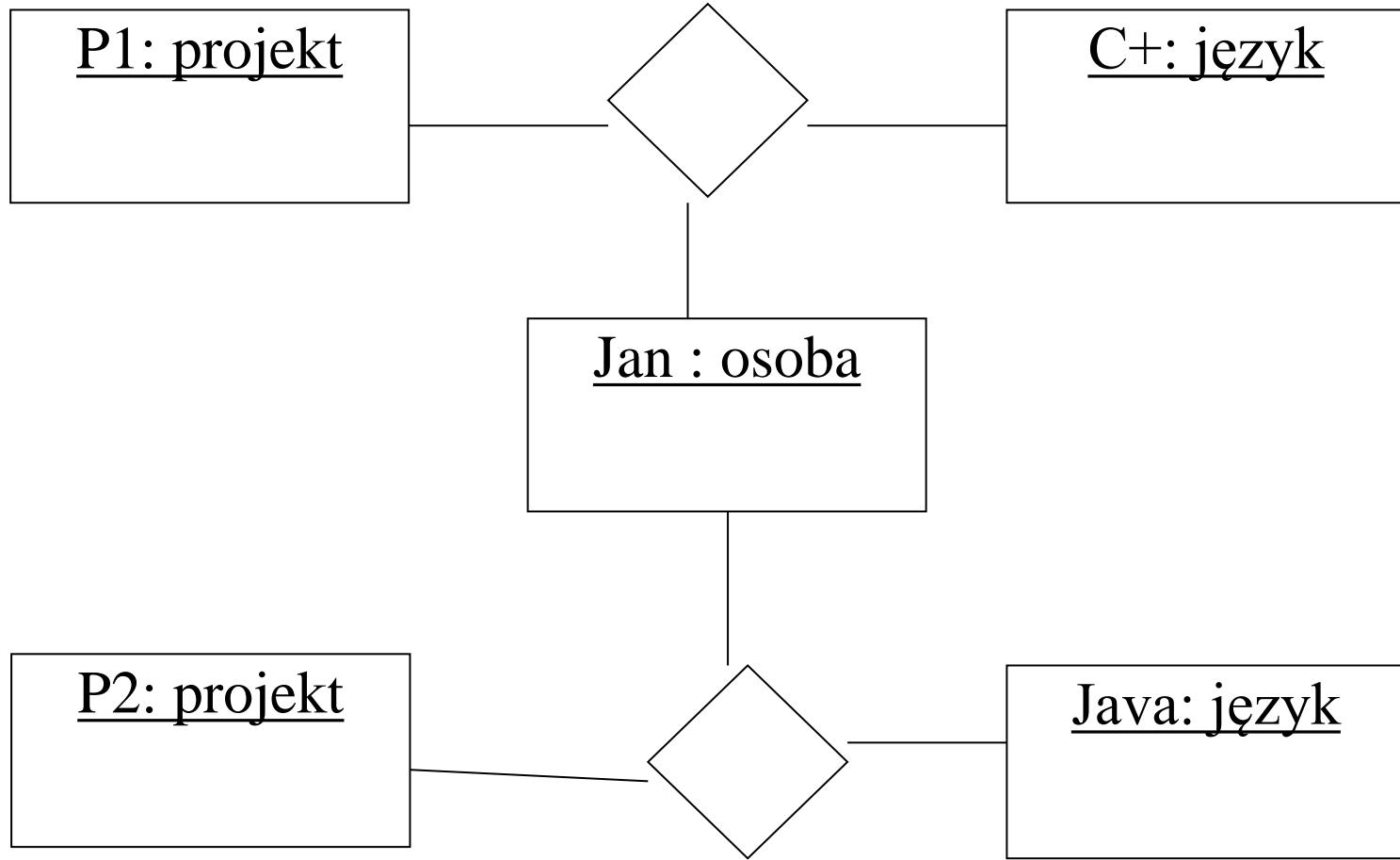
Pomiędzy 3 lub więcej klasami, nie może być podzielone bez utraty informacji.



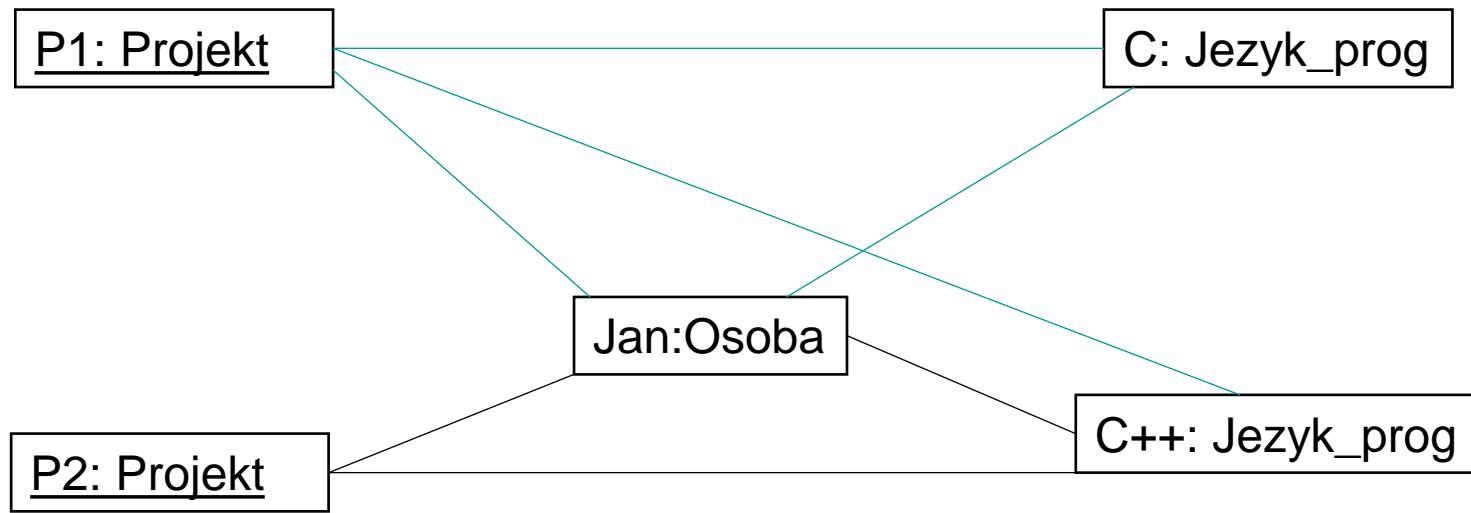
# Bez relacji ternarnej



# Diagram obiektów



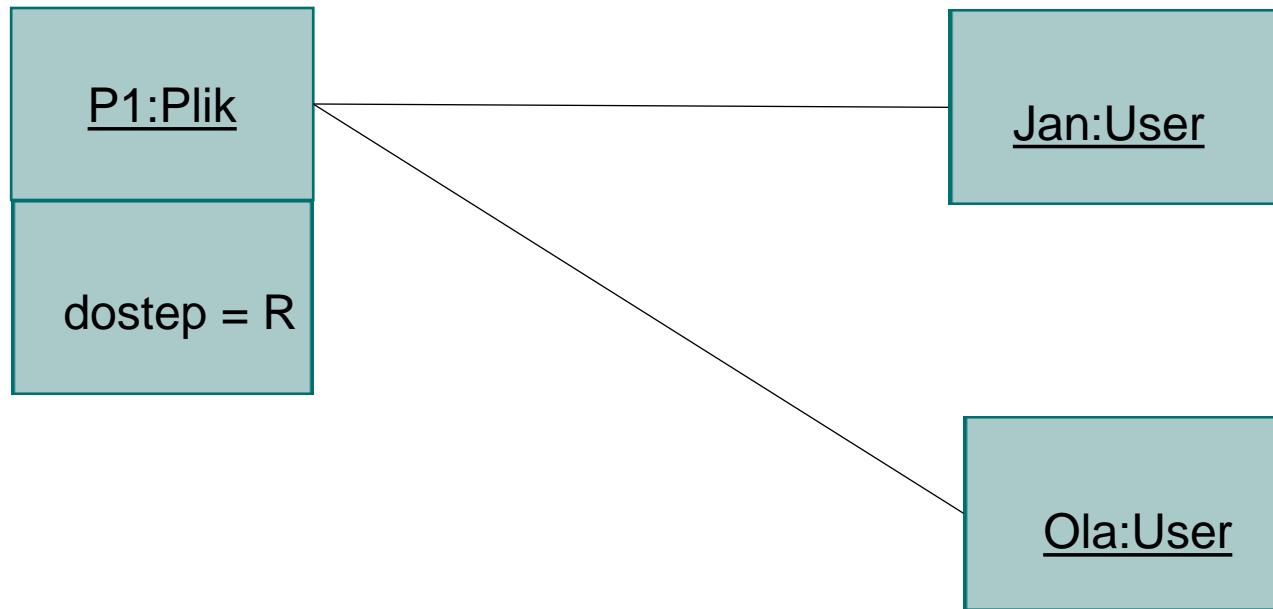
# Diagram obiektów



# Klasy Plik- User



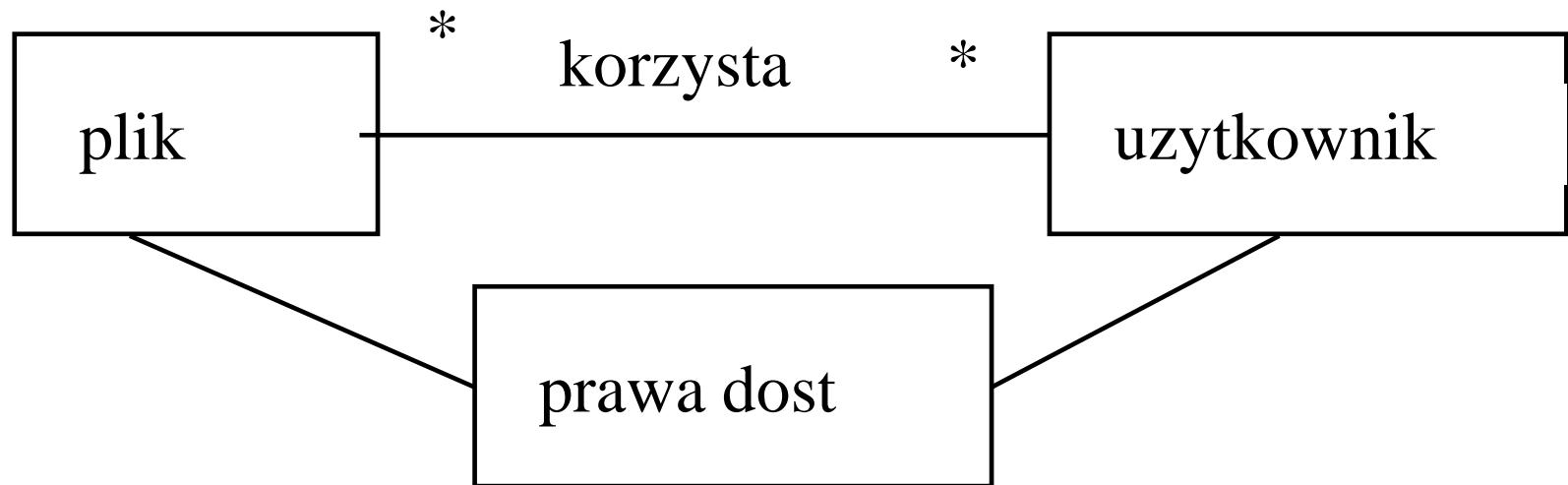
# Diagram obiektów



Co jeśli prawa dostępu „włożymy” do klasy User ?

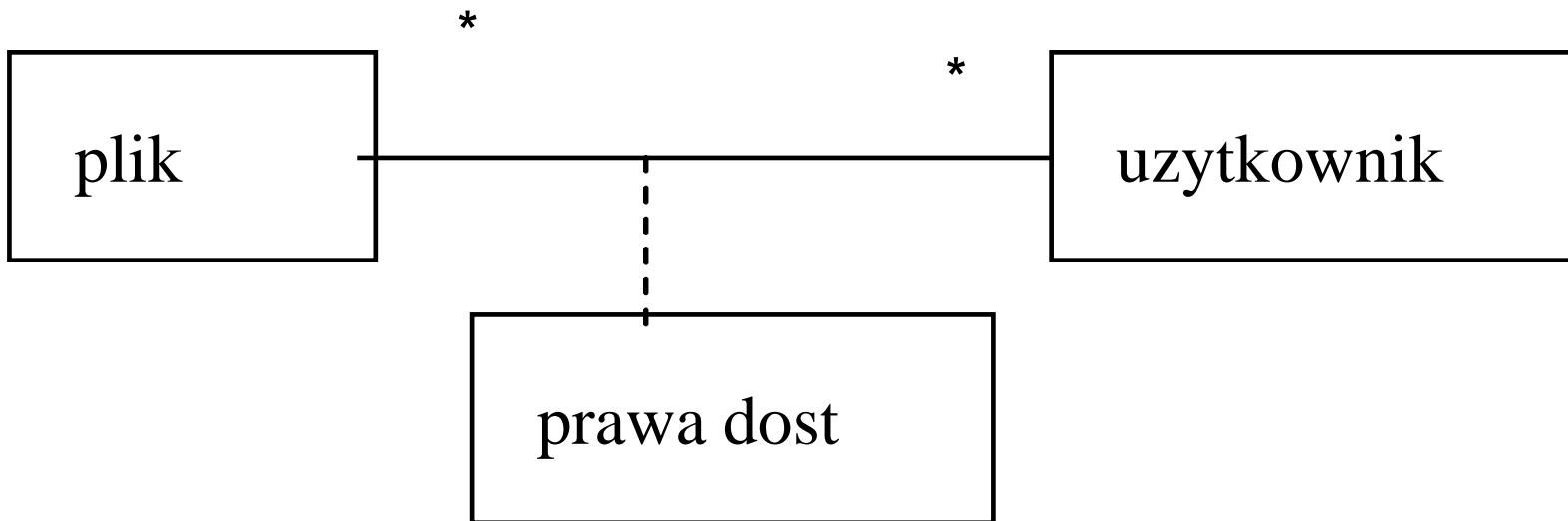
# Wariant ze zwykłą klasą

Przy wartościowściach wiele - wiele atrybut ten nie może być dołączony do obiektu np.

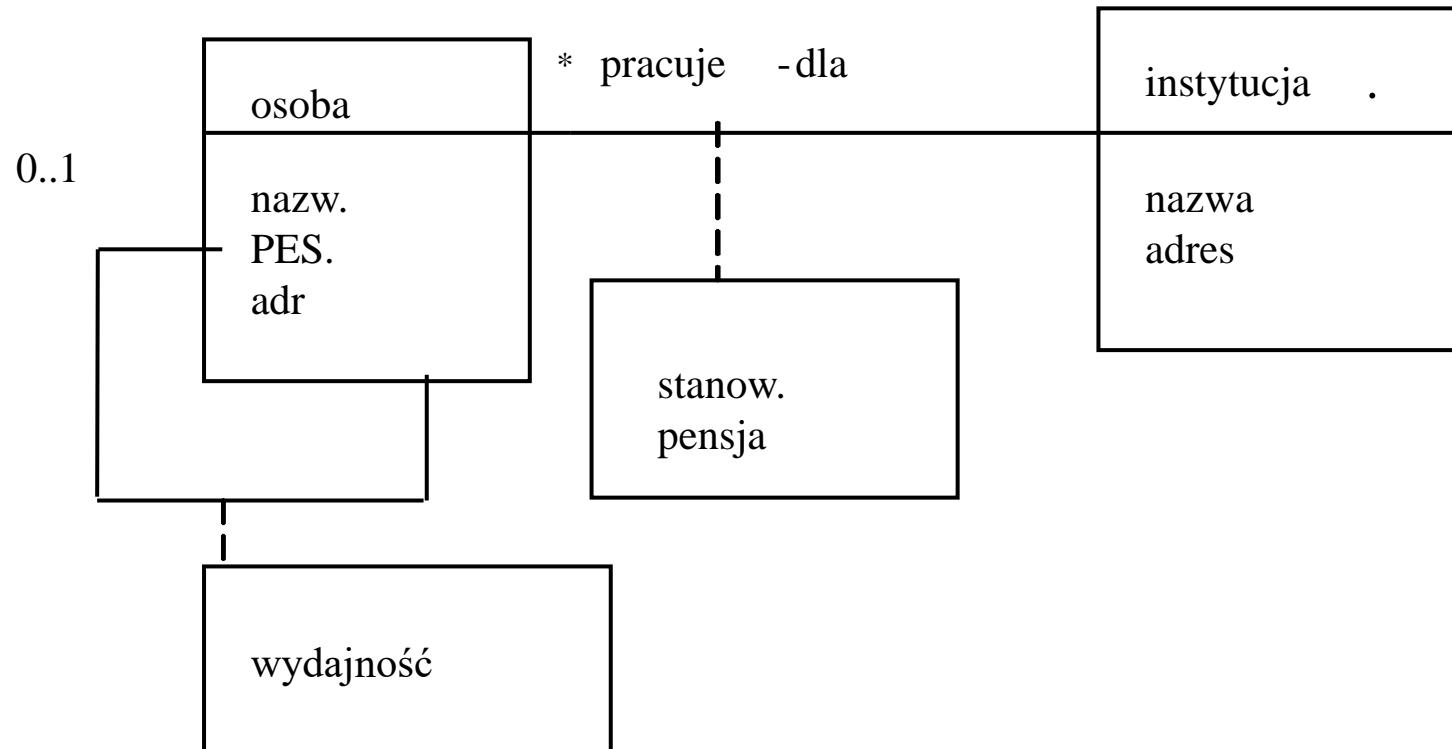


# Atrybuty powiązań

Określają pewną własność powiązania  
**klasa asocjacyjna**

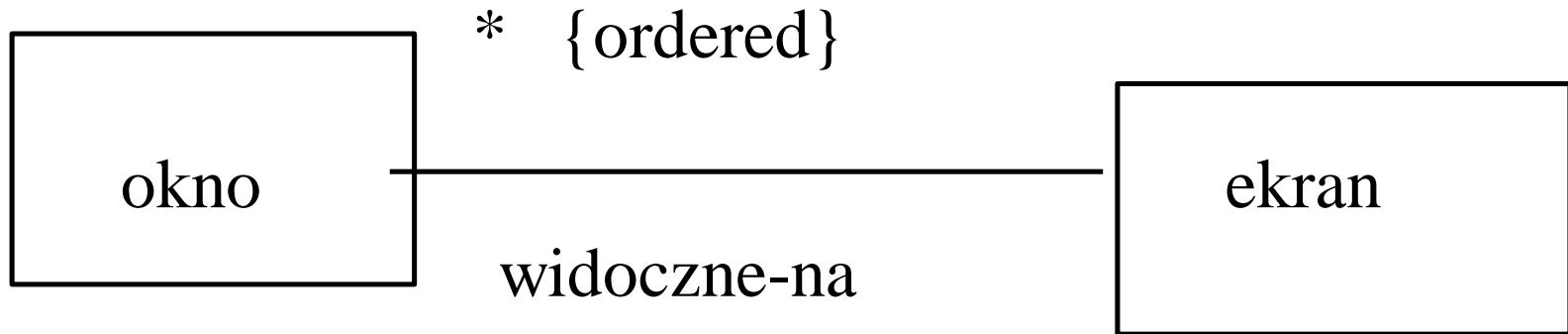


# Przykład z klasami asocjacyjnymi



# Ograniczenie {ordered}

**Powiązania** mogą także posiadać własność  
**"uporządkowania"**



# Różne asocjacje pomiędzy tymi samymi klasami

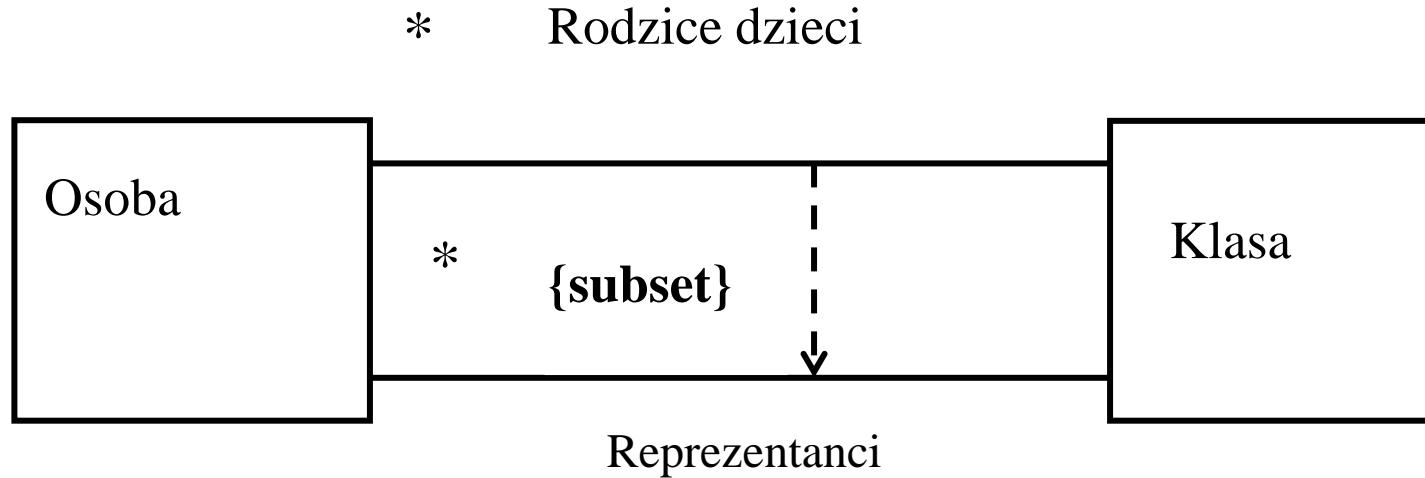
Obiekt Osoba może być w relacji „Reprezentanci” , „Rodzice dzieci” z różnymi obiektami Klasa.



# Ograniczenie {subset}

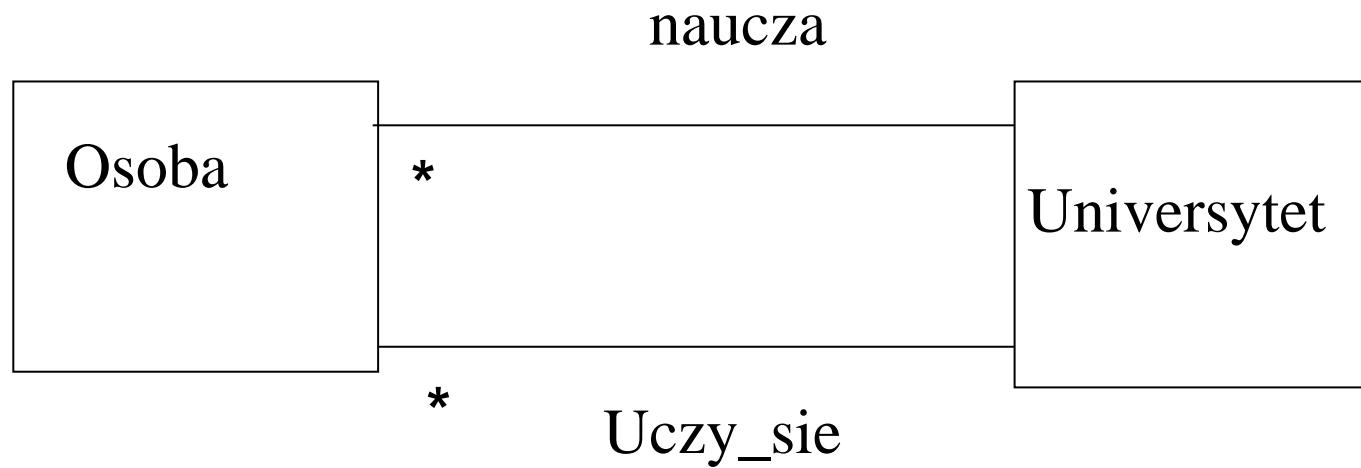
wskazuje, że pewien zbiór (kolekcja) jest włączony w inny zbiór np.

Reprezentanci rodziców dzieci są także rodzicami dzieci.



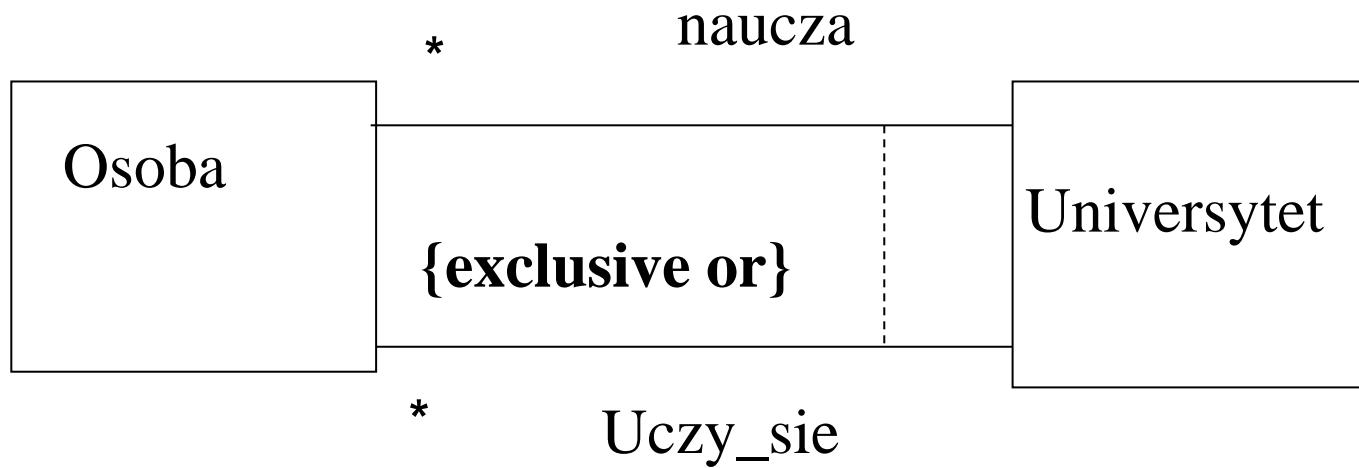
# Różne asocjacje pomiędzy tymi samymi klasami

Dla danego obiektu obie relacje asocjacji są możliwe.



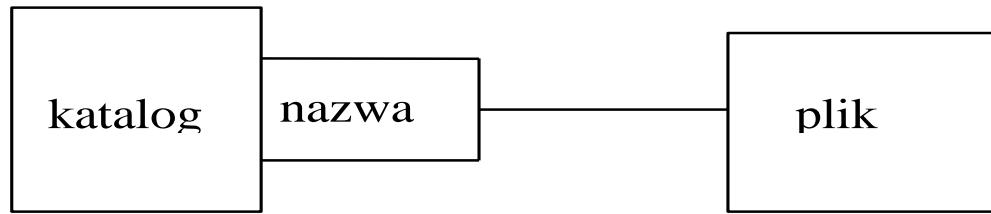
# Ograniczenie {exclusive or}

wskazuje, że dla danego obiektu jedynie jedna relacja asocjacji spośród grupy relacji jest właściwa.

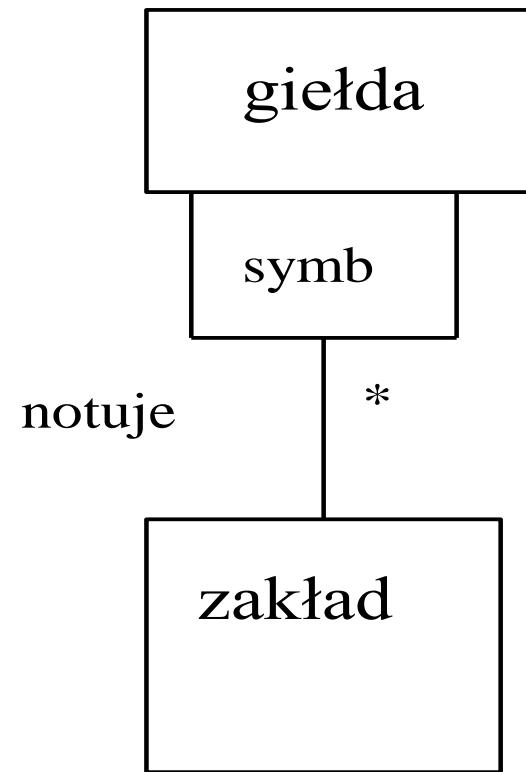
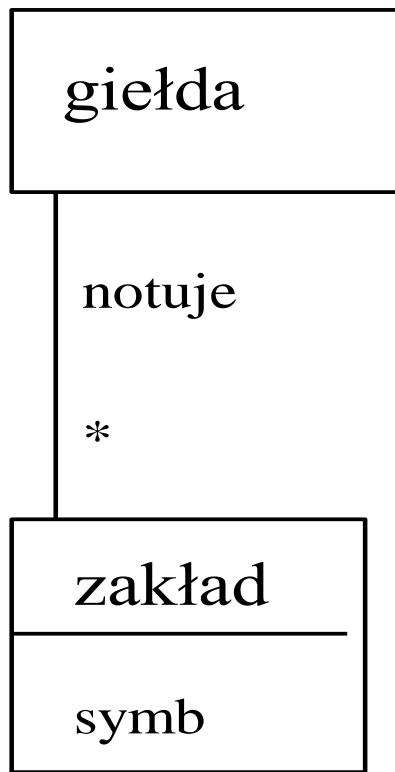


# Powiązania "kwalifikowane"

Kwalifikator wyróżnia między wieloma obiektami, sprowadza punkt wiele do 1.



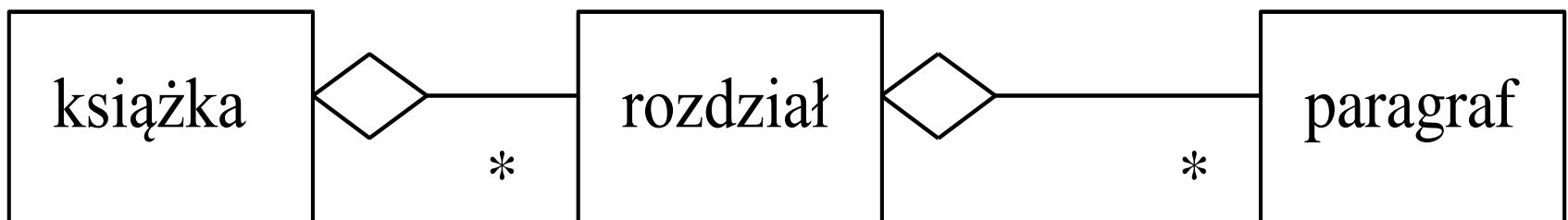
# Przykład z kwalifikatorami powiązań



# Agregacja -kompozycja

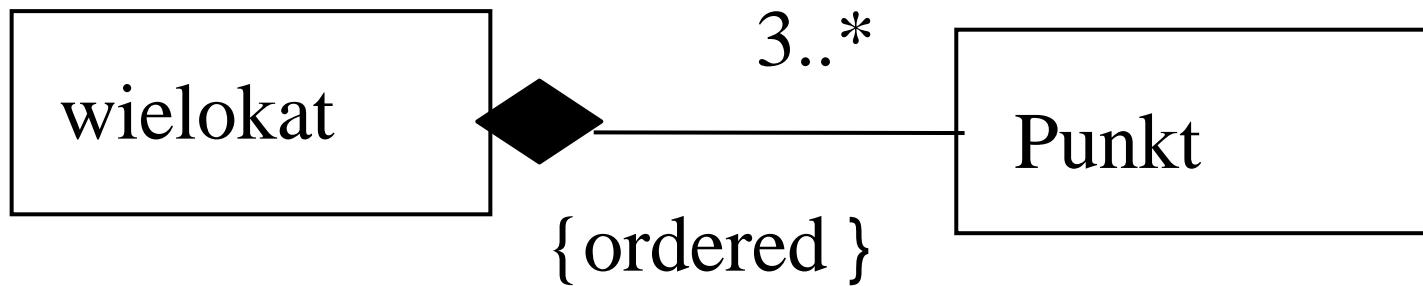
Określa relację :

- "składa się z"
- "jest zbudowany z"
- "jest częścią"



# Kompozycja – agregacja całościowa

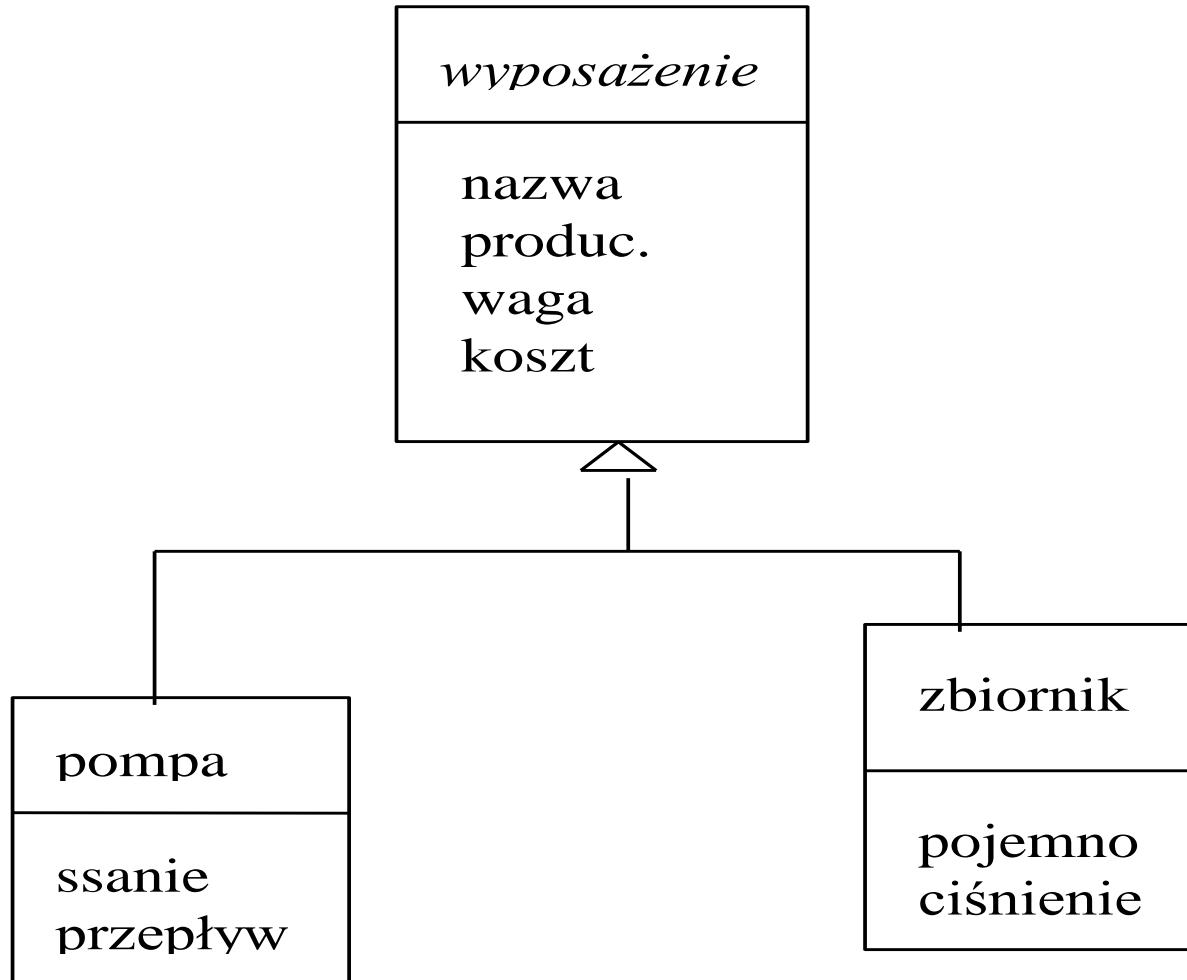
oznacza „fizyczną” agregację np. wielokąt składa się z uporządkowanych punktów.



# Dziedziczenie i generalizacja

- **Generalizacja** jest związkiem między klasą a jej "ulepszeniami" -podklasami.
- Atrybuty i operacje wspólne dla grupy podklas są umieszczane w superklasie.
- "podklaśa" dziedziczy atrybuty, operacje.
- Dziedziczenie jest przechodnie.

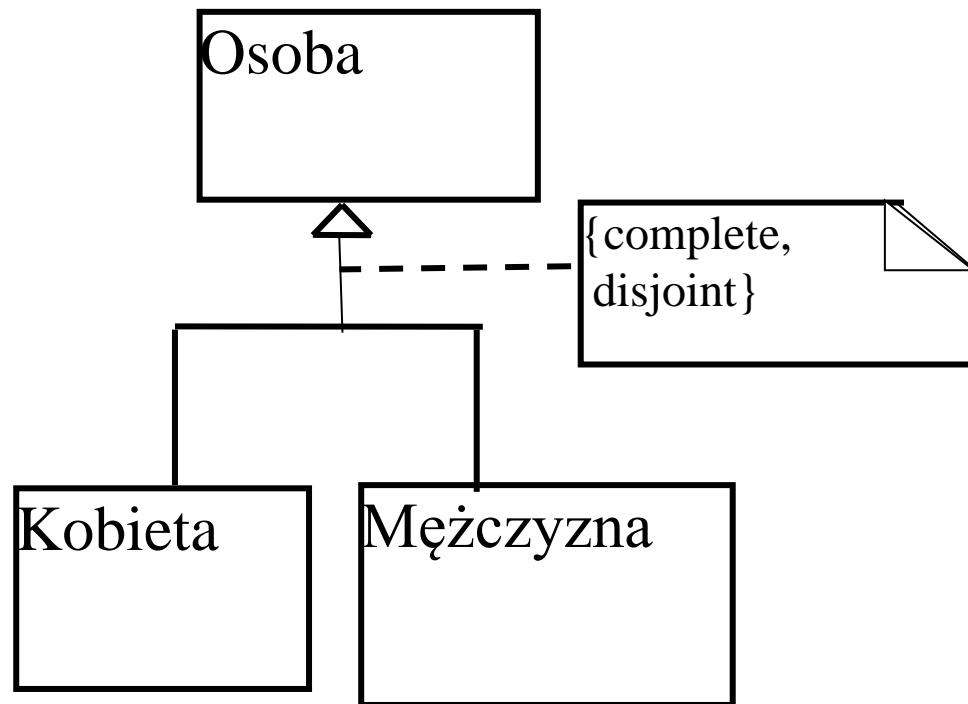
# Przykład dziedziczenia



# Ograniczenia relacji dziedziczenia

- **{exclusive}** domniemana, obiekt jest instancja tylko jednej podklasy
- **{disjoint}** rozłączna: klasa pochodna od A jest podkласą tylko jednej podklasy klasy A
- **{overlapping}** nakładająca się : klasa pochodna od A należy do produktu kartezjańskiego podklas klasy A np. (urządzenia, urządzenia do rejestracji obrazu, urządzenia do rejestracji dźwięku, magnetowid)
- **{complete}**
- **{incomplete}**

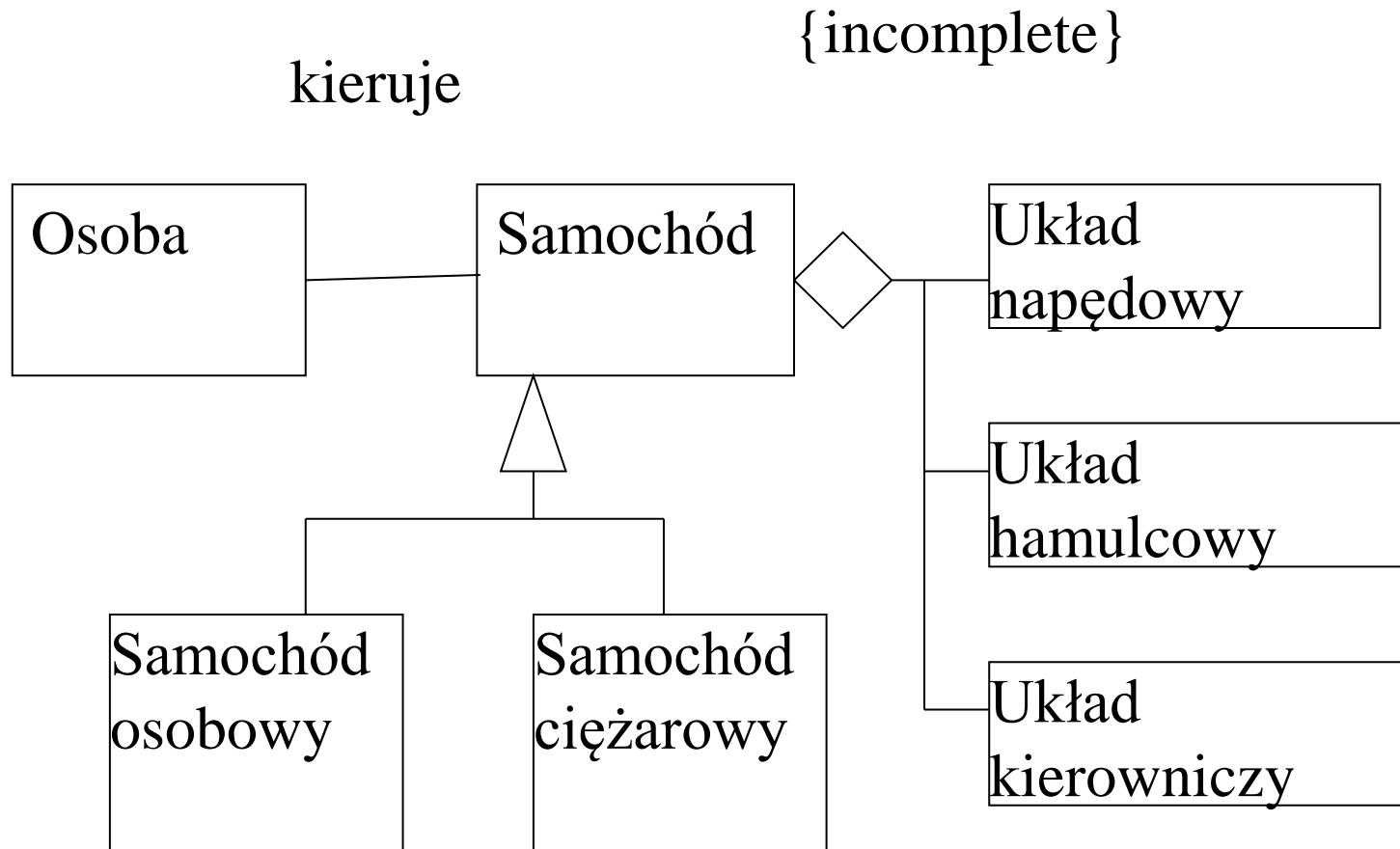
# Przykład ograniczeń generalizacji



# Podsumowanie

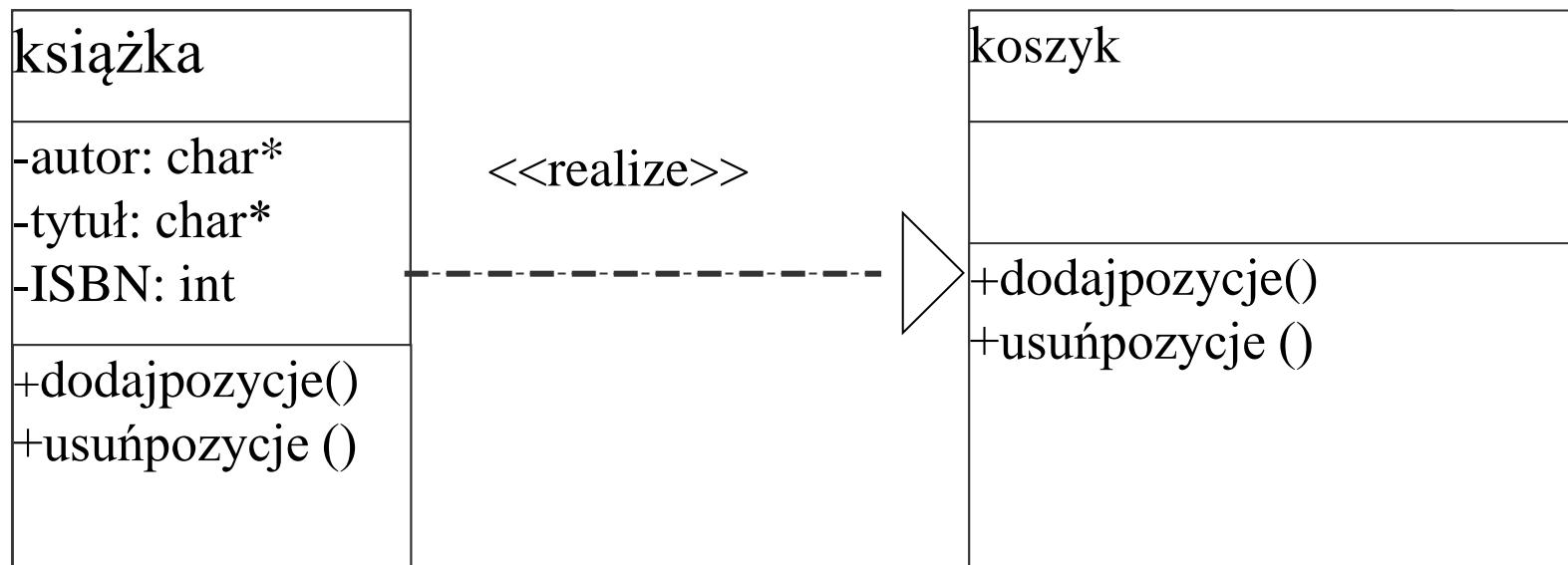
- ze specyfikacji wyodrębnij obiekty (ważne rzeczowniki z dziedziny problemu ale nie dotyczące implementacji),
- utwórz powiązania,
- zastosuj generalizację, kompozycję,
- dodaj atrybuty, operacje,
- podziel klasy na moduły spójne pod względem świadczonych usług, umieść je w pakietach.

# Przykład – różne typy relacji



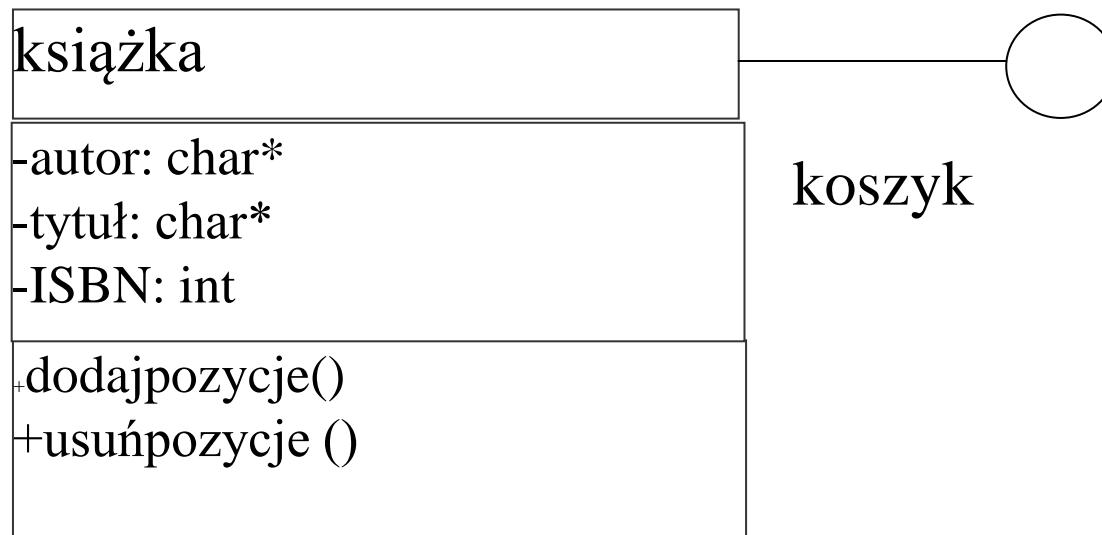
# Klasa interfejsowa

Klasa definiuje operacje udostępniane innym obiektom



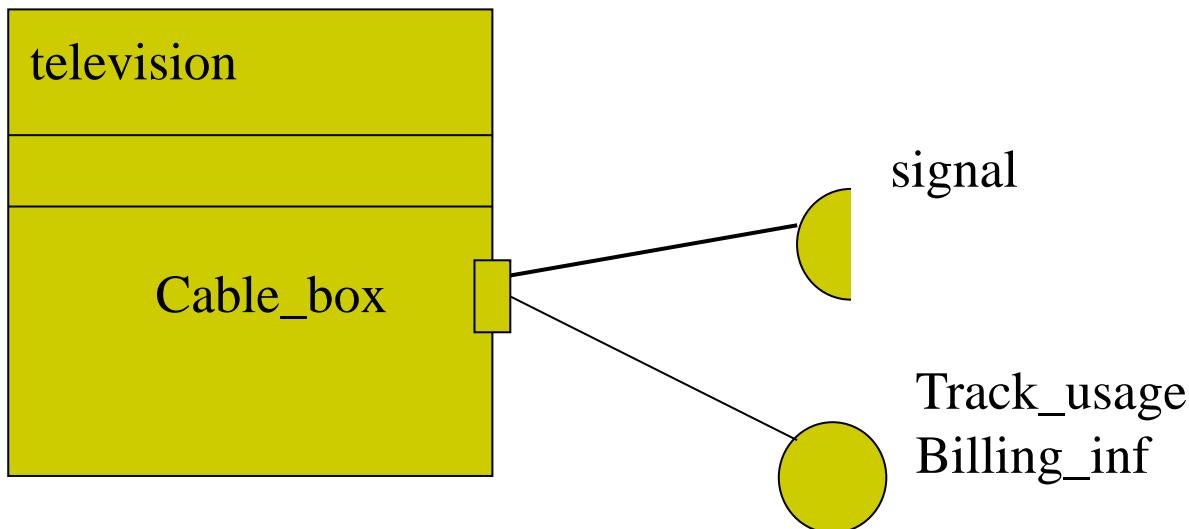
# Klasa interfejsowa -2

## Inna notacja



# Bramy (ports)

Klasa *television* wymaga sygnału z zewnętrznego źródła, firma telewizja kablowa chce mieć możliwość śledzenia używania telewizji. Bramy *Cable\_box* dostarcza tego interfejsu (*Track\_usage*), także dostarczony jest interfejs umożliwiający obciążanie klienta (*Billing\_inf*).



# Zadanie 1

Określ i narysuj w notacji UML typy relacji pomiędzy obiektami w poniższych zdaniach. Odpowiedź należy uzasadnić

1. Klient ma miejscowościę na określony pociąg
2. Listonosz dostarcza przesyłki
3. Bileter sprzedaje bilety
4. W plecaku znajdują się książki, zeszyty
5. Magnetofon, magnetowid są urządzeniami do rejestracji dźwięku

## Zadanie 2

Na podstawie podanego poniżej zbioru słów opracuj spójny diagram klas w UML, pokazujący relacje między obiektami klas.

Należy podać typ relacji (powinny występować wszystkie typy relacji), jej nazwę ewentualnie krotność. **Odpowiedź należy uzasadnić.**

{**Zamek, most zwodzony, wieża, schody, korytarz, pokój, okno, podłoga, duch, strażnik, kucharz, hrabia, hrabina, lokaj**}

# Diagramy maszyny stanowej (state machines)

---

Dr hab. inż. Ilona Bluemke

# Diagram maszyny stanowej

- Opisuje zmiany stanów obiektu na skutek zdarzeń. Do notacji używa się diagramów Davida Harel'a (1987).
- **Węzły - stany** - abstrakcja zbioru wartości atrybutów i połączeń obiektu
- **Skierowane krawędzie** - zmiany stanów etykietowane nazwami zdarzeń
- **diagram deterministyczny**

# stany

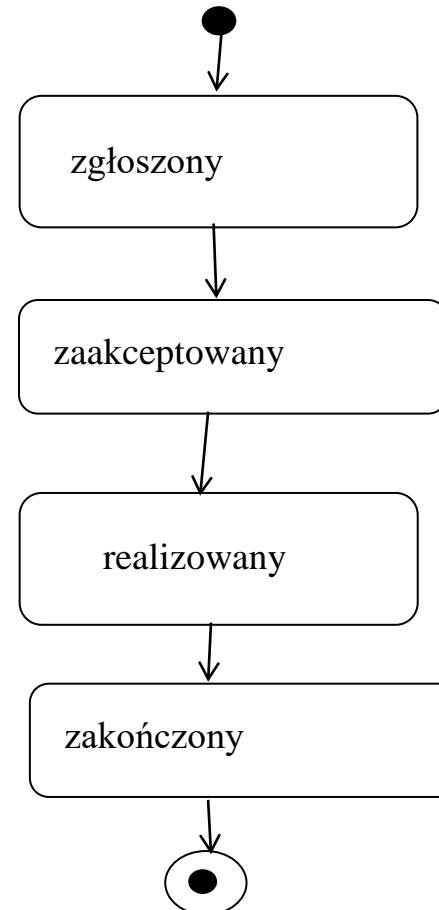
- Stan określa reakcję obiektu na zdarzenie. Reakcja może być różna dla różnych stanów. Odpowiedzią na zdarzenie może być akcja lub zmiana stanu obiektu.
- zdarzenia reprezentują chwile czasu, stany - interwały czasu.
- Stan obiektu zależy od sekwencji zdarzeń jakie obiekt otrzymał w przeszłości. Definiując stany ignorujemy atrybuty, które nie wpływają na zachowanie obiektu. Łączymy w jednym stanie wartości atrybutów, połączeń, które dają taką samą odpowiedź na zdarzenie.

# Stany charakteryzują

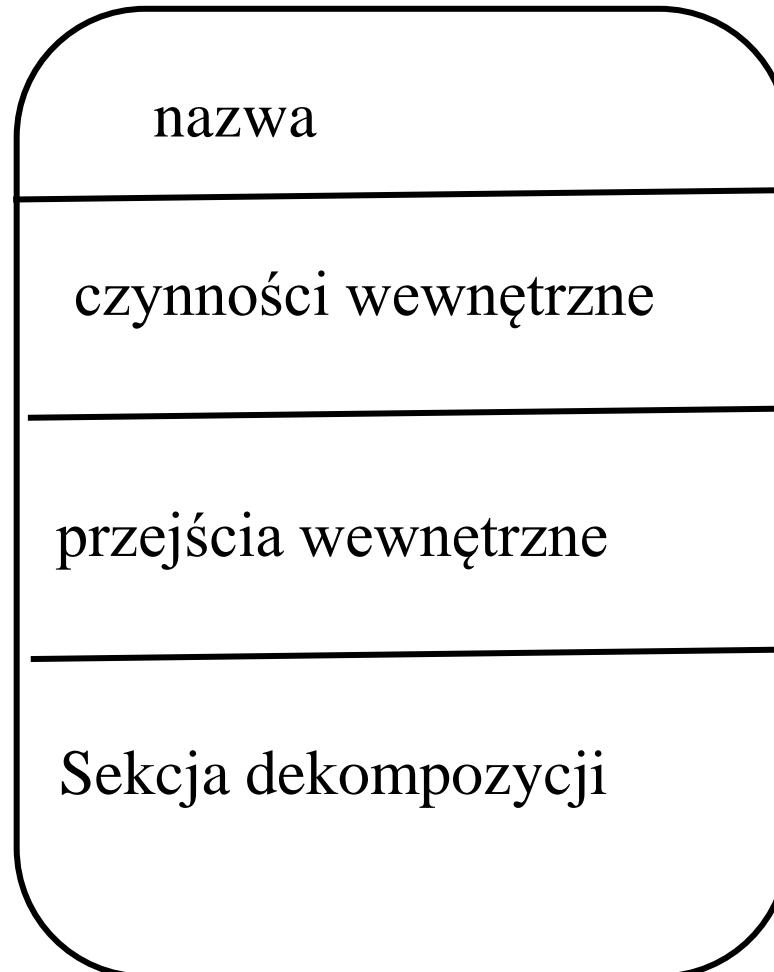
- nazwa stanu - **PRZYMIOTNIK**
- sekwencja zdarzeń powodujących wejście do tego stanu
- warunki charakteryzujące stan
- oczekiwane zdarzenia
- akcje - reakcje na zdarzenia
- stany następne
- stan początkowy                      •
- stan końcowy                      ○

# Przykład diagramu maszyny stanowej

- Np. kurs na uczelni



# Sekcje symbolu graficznego stanu



# Diagram maszyny stanowej

- Opisuje zachowanie obiektów jednej klasy. Wszystkie instancje klasy mają takie same zachowanie - "dzielą" diagram stanów. Każdy obiekt jest w swoim stanie ale jest niezależny od innych obiektów.
- Ze zdarzeniami mogą być związane akcje, zapisywane na krawędziach diagramu stanu po / , mogą także reprezentować wewnętrzne operacje sterujące np. ustawienie atrybutów, generowanie zdarzeń.

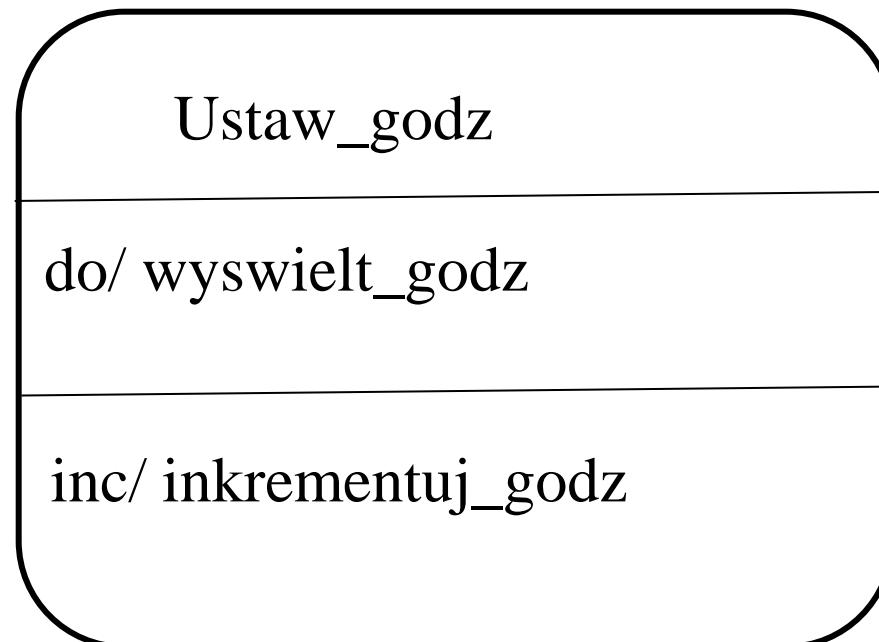
**zdarzenie/ akcja**

# Słowa kluczowe opisujące zdarzenia

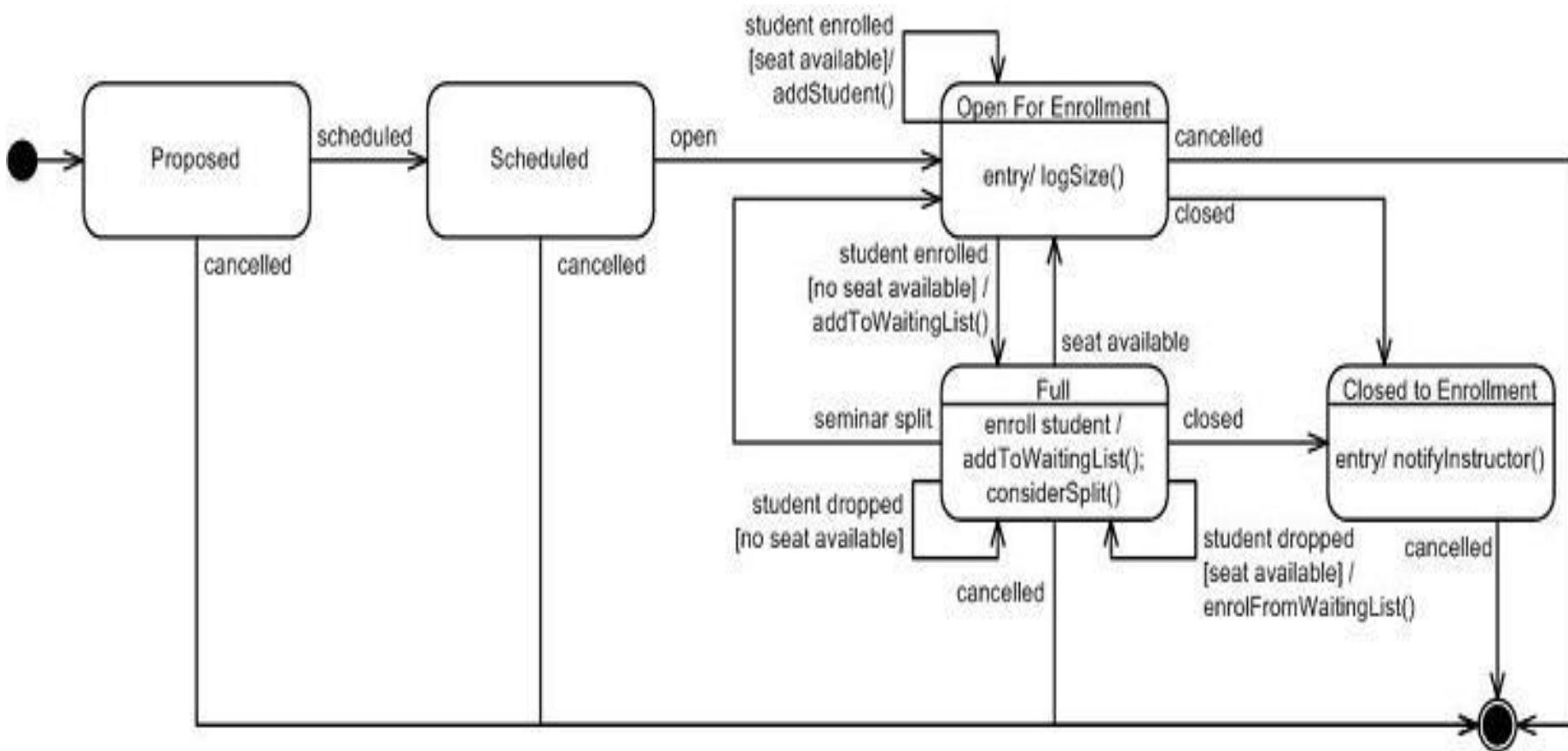
- **entry** identyfikuje czynność wykonywaną przy wejściu obiektu do stanu (jedną)
- **do** identyfikuje czynność wykonywana w sposób ciągły na obiekcie znajdującym się w danym stanie, można określić kilka takich czynności (niezależnie wykonywanych), czynności te są wykonywane po czynności *entry*
- **exit** identyfikuje czynność wykonywaną przy wyjściu ze stanu (jedną)

# przejścia wewnętrzne

**zdarzenie/operacja** określają czynności wykonywane przez obiekt będący w stanie pod wpływem **zdarzenia**



# Zachowanie klasy Seminarium podczas rejestracji

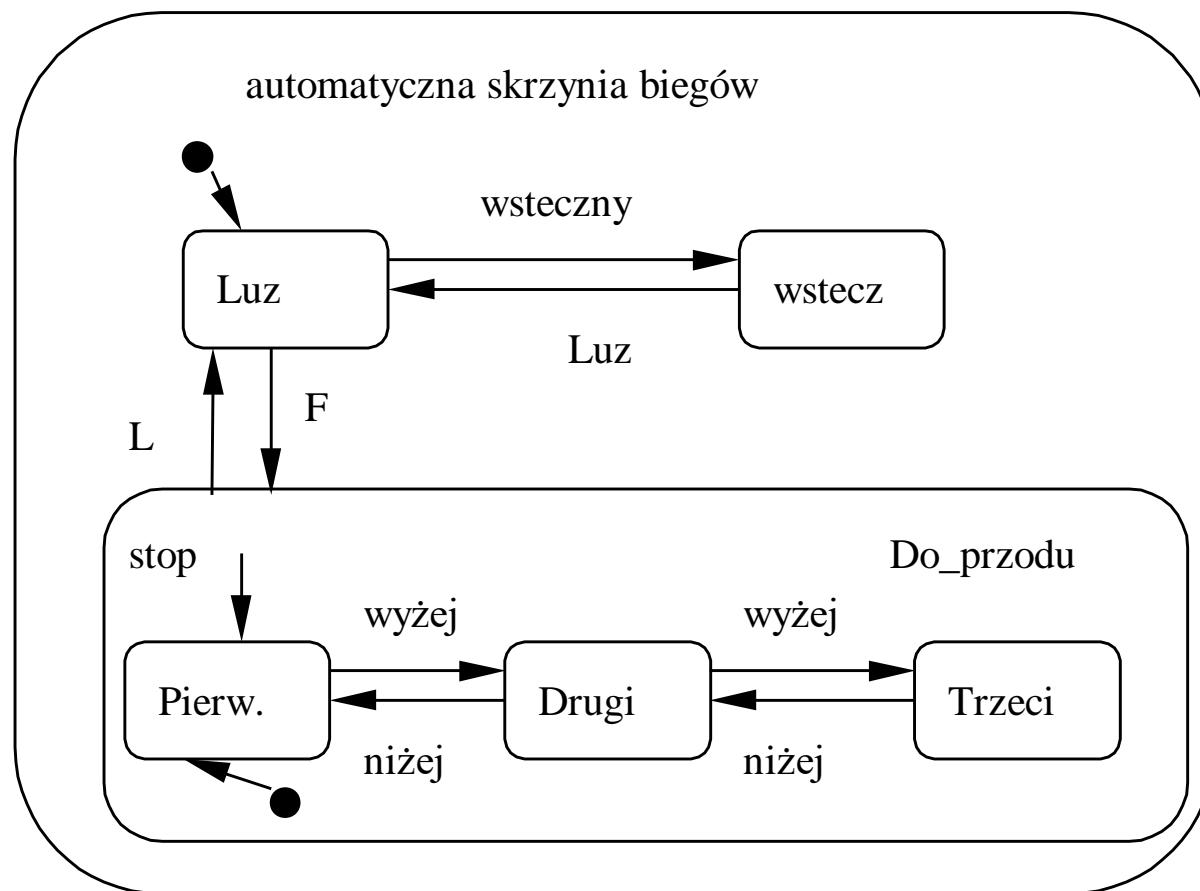


# Klasyfikacja stanów

- proste
- złożone
  - stan **złożony** zawiera maszynę stanową
  - lub **jest podzielony na obszary współbieżne**  
(zawiera podstany)

# Generalizacja stanów (relacja or )

Stan zawiera maszynę stanową



# Przykład zmian stanów

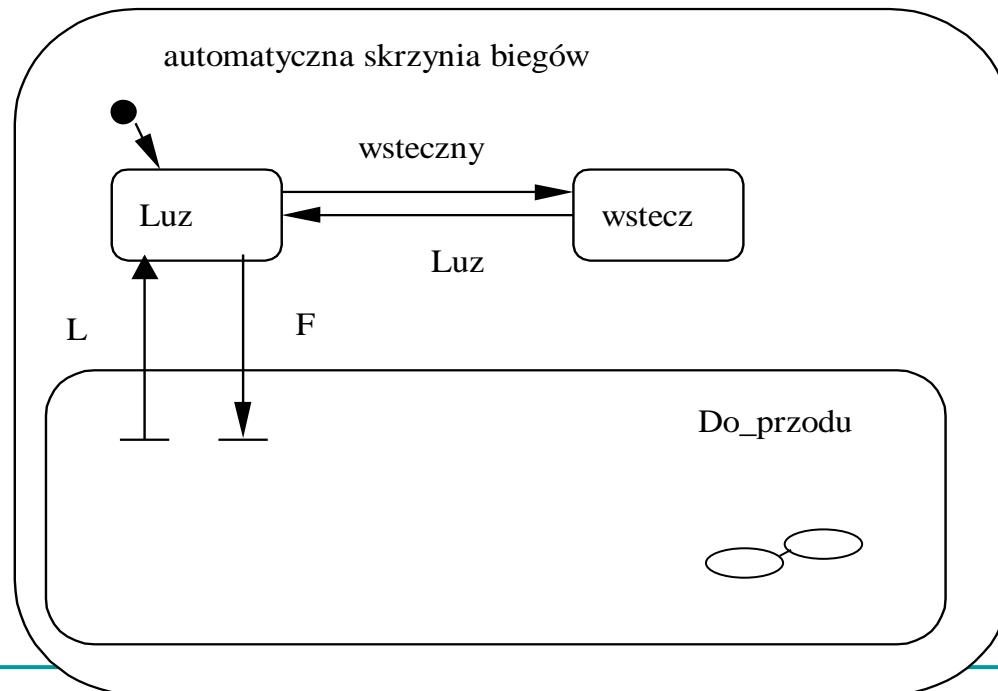
- $Luz \rightarrow F$  Pierw  $\rightarrow$  wyżej Drugi  $\rightarrow L$  Luz
- $Luz \rightarrow F$  Pierw  $\rightarrow$  wyżej Drugi  $\rightarrow$  wyżej Trzeci  $\rightarrow L$  Luz

# Generalizacja stanów

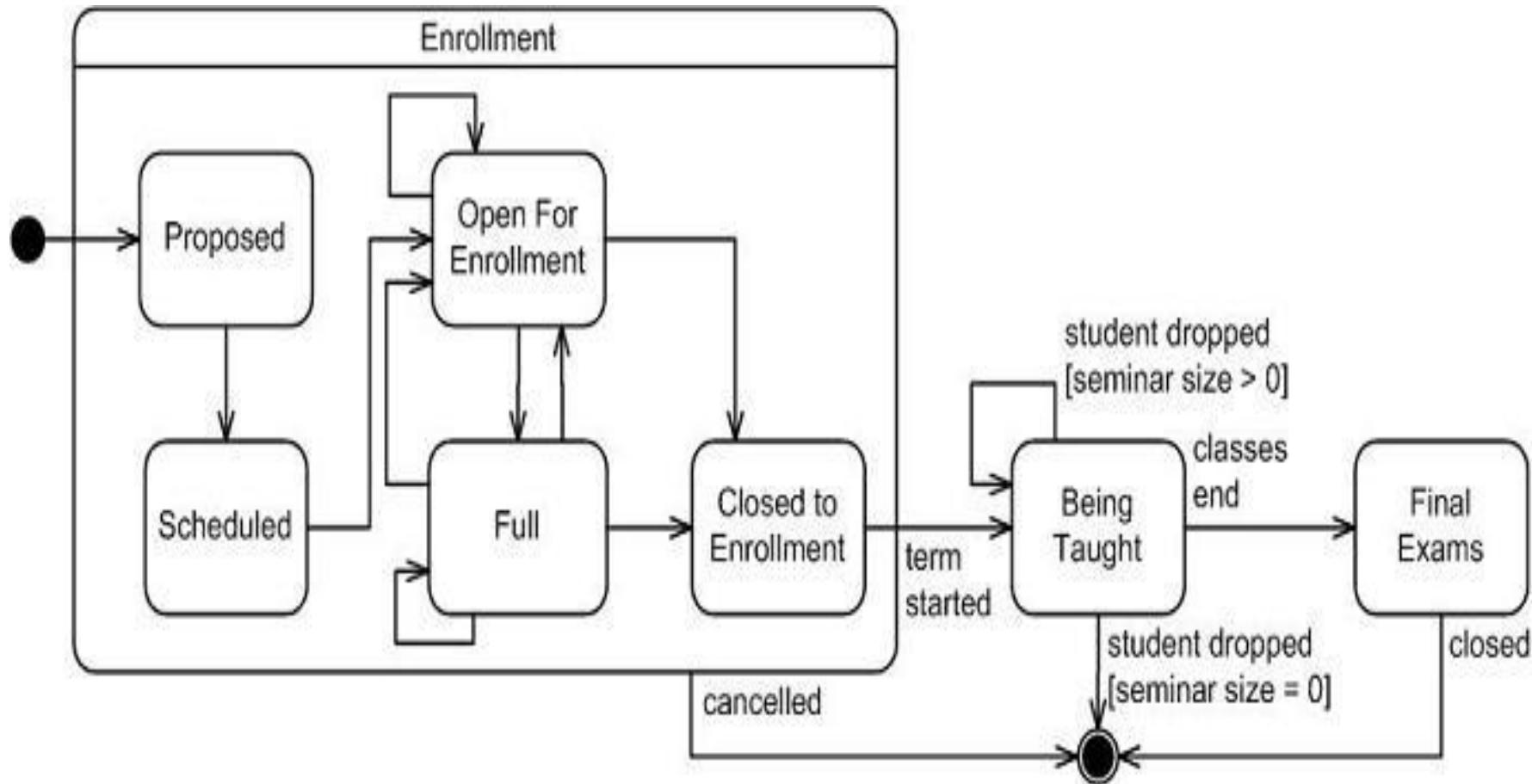
- Pozwala na opis na wysokim poziomie a następnie na uszczegóławianie na coraz niższych poziomach.
- Tworzona jest struktura hierarchiczna z dziedziczeniem wspólnego zachowania i struktury.
- Stan może mieć diagram maszyny stanowej, który dziedziczy przejścia superstanu. Przejście, akcja superstanu dotyczy wszystkich jego podstanów (chyba, że zostanie przysłonięte przez przejście w podstanie).
- Wybór "L" w dowolnym podstanie "**do przodu**" powoduje przejście do stanu "**luz**". Wybór "F" powoduje przejście do stanu "**pierwszy**". Zdarzenie "**stop**" w dowolnym podstanie "**do przodu**" powoduje przejście do stanu "**pierwszy**".

# Notacja graficzna

W złożonych diagramach stanów diagramy przejść podstanów można rysować na oddzielnych diagramach.



# Pełny diagram stanów dla klasy Seminarium

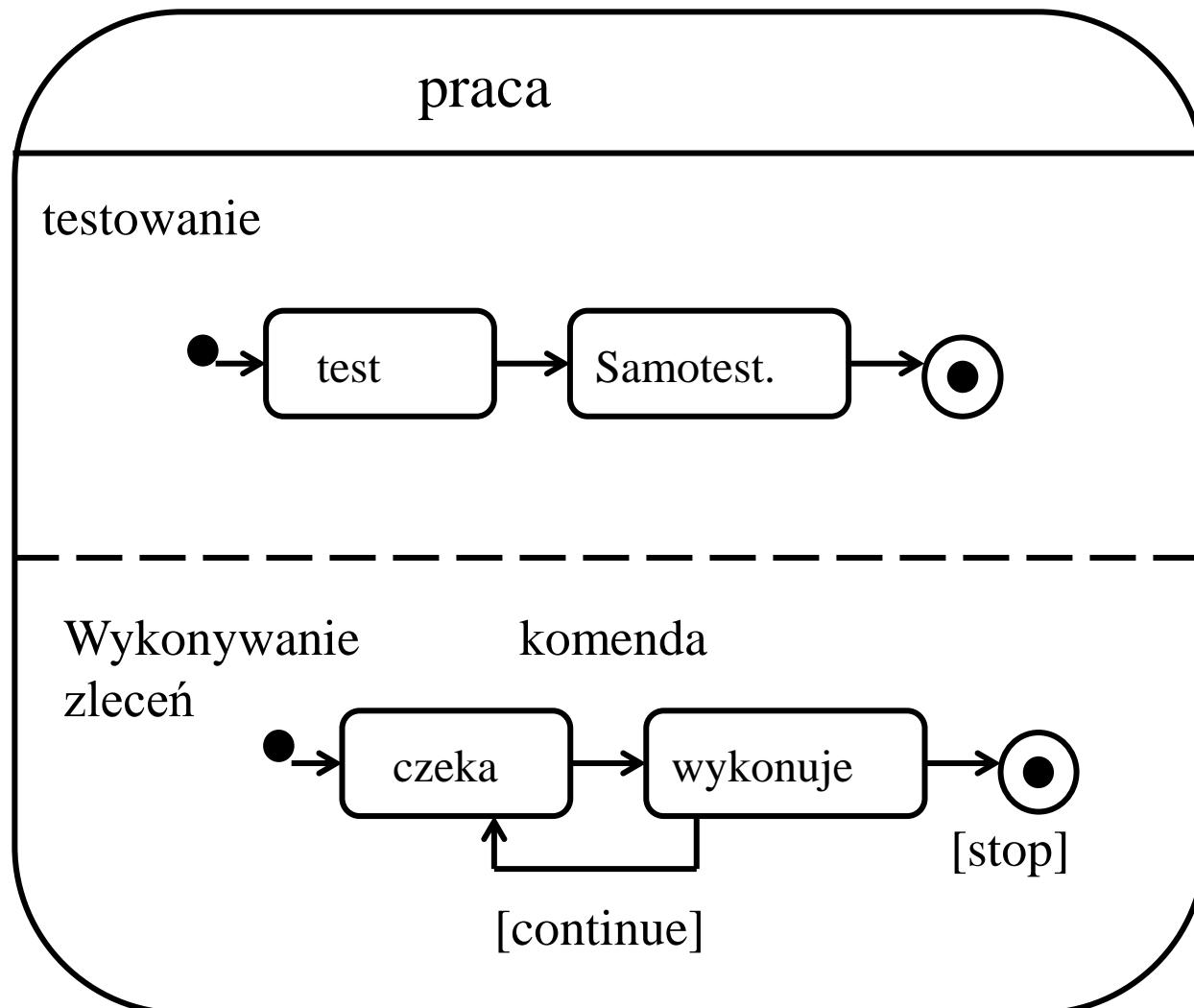


# Kompozycja – agregacja stanów

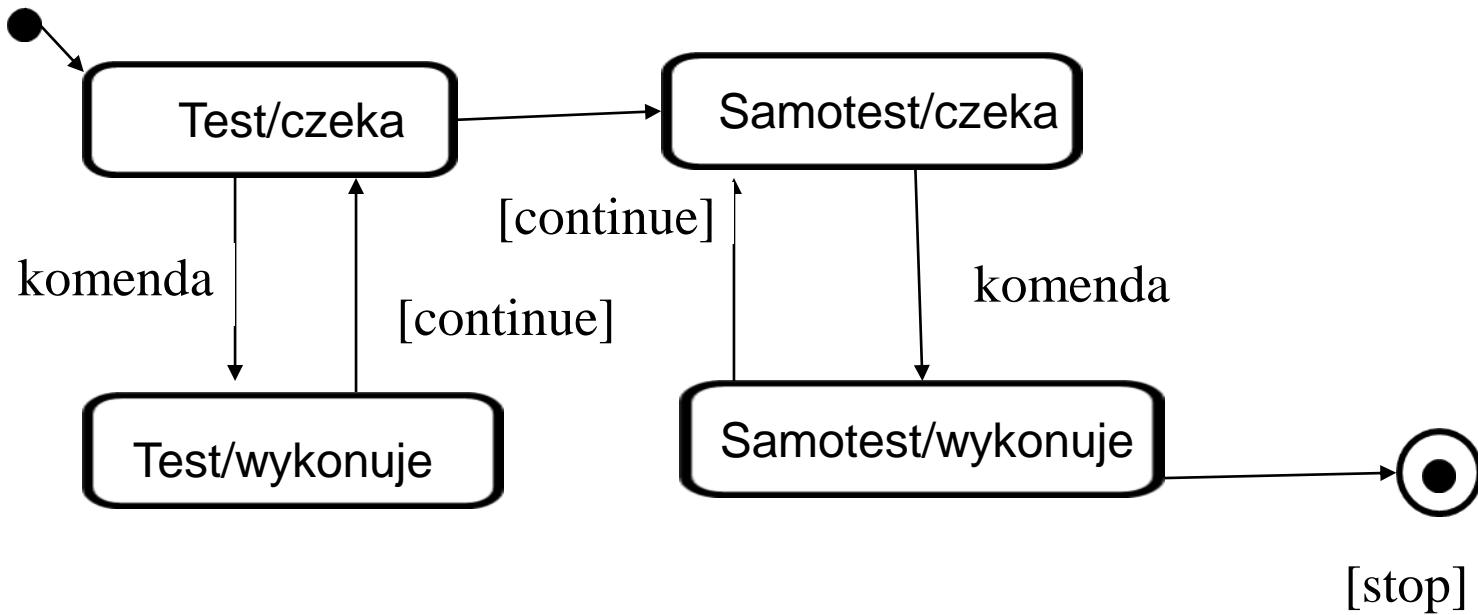
Pozwala na podział na części składowe z ograniczoną interakcją między nimi.

- Jest równoważne **współbieżności stanów**.
- Każdy komponent wykonuje przejścia równolegle z pozostałymi (równolegle działając „testowanie” i „wykonywanie zleceń” na slajdzie następnym).

# Przykład agregacji stanów



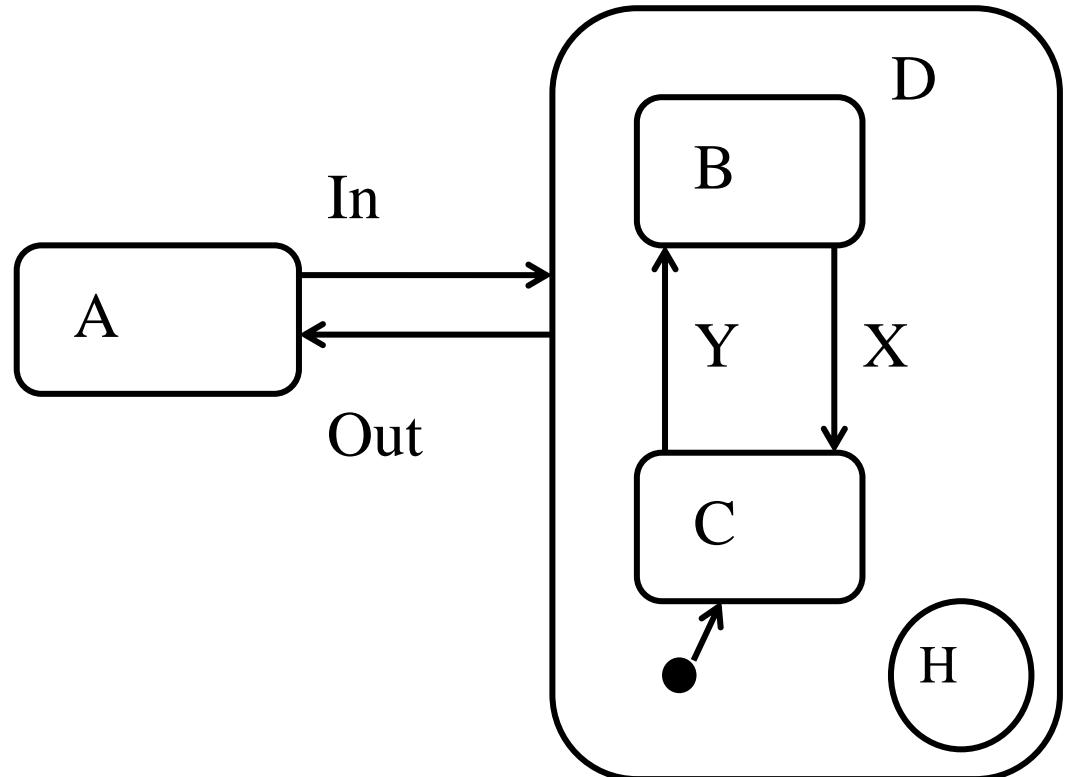
# Zadanie -1



„płaski” model - bez  
agregacji stanów

# Mechanizm historii

Pozwala na pamiętanie stanu ostatnio odwiedzonego w podstanie i wejście do niego przy kolejnym „wejściu” do stanu złożonego.

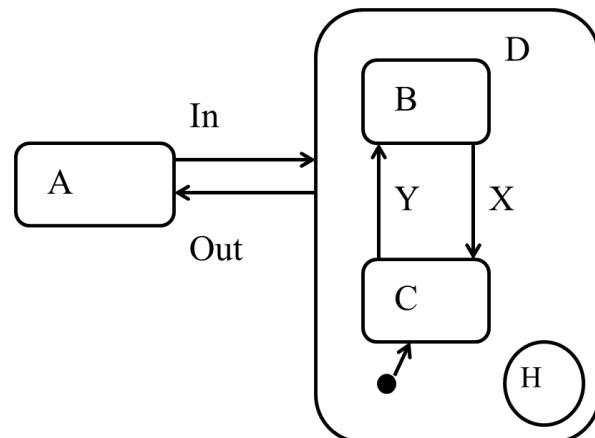


## ■ Bez mechanizmu historii

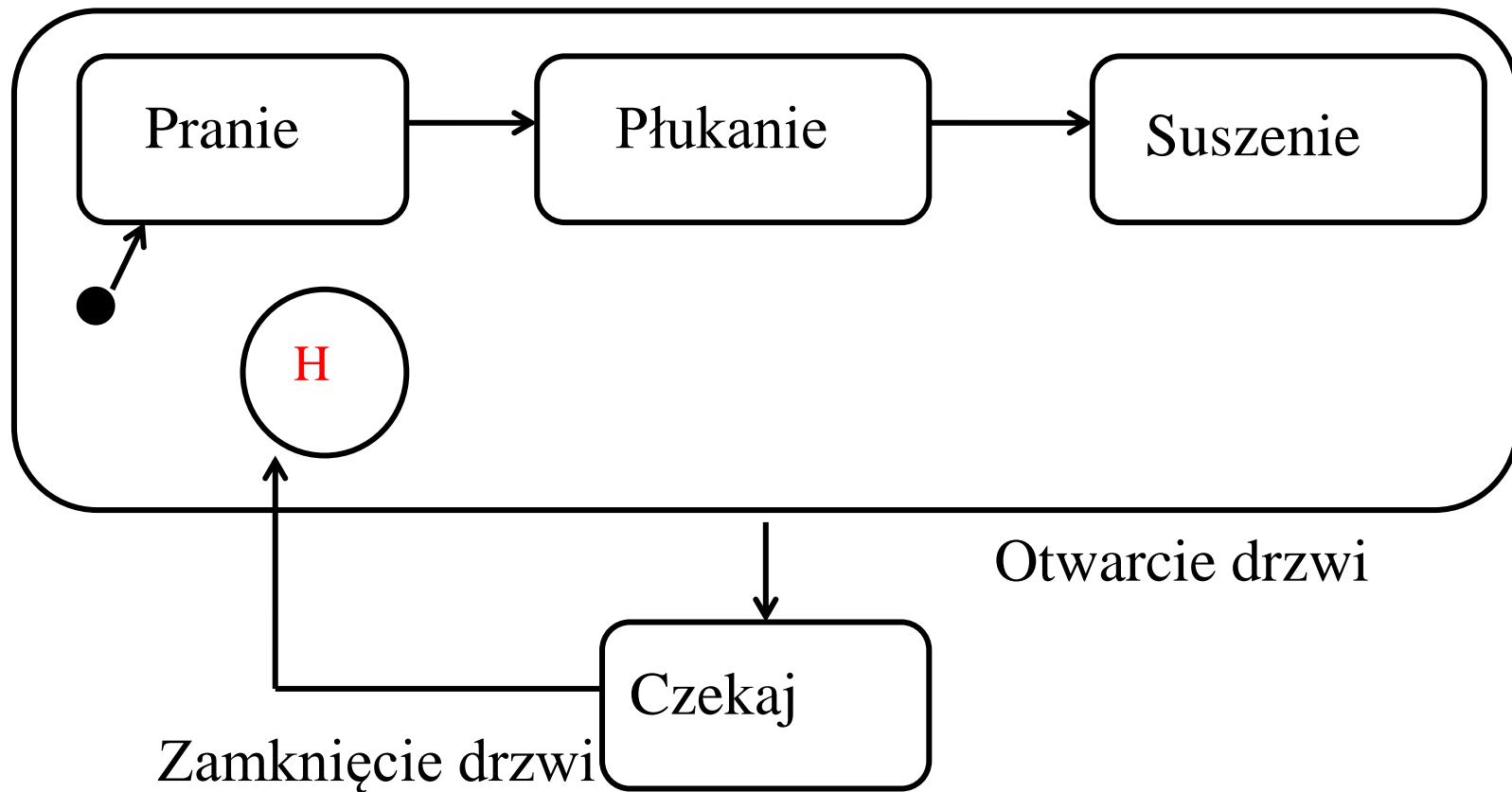
- A -><sub>(In)</sub> C -><sub>(Y)</sub> B -><sub>(Out)</sub> A -> <sub>(In)</sub> C

## ■ Z mechanizmem historii

- A -><sub>(In)</sub> C -><sub>(Y)</sub> B -><sub>(Out)</sub> A -> <sub>(In)</sub> B



# Przykład maszyny stanowej z historią



# Zmiany stanów

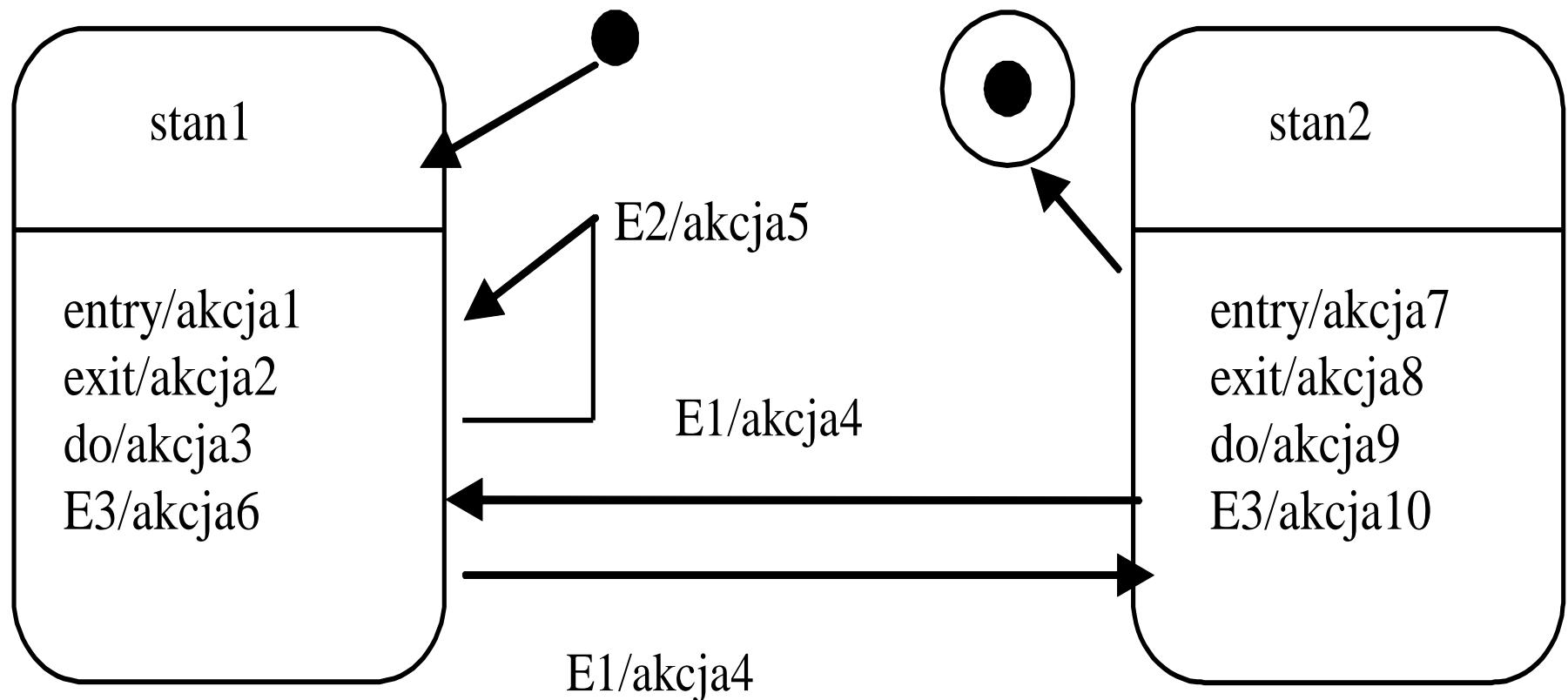
- Bez mechanizmu historii
  - Pranie -> Płukanie -><sub>otwarcie drzwi</sub> -> Pranie
- Z mechanizmem historii
  - Pranie -> Płukanie -><sub>otwarcie drzwi</sub> -> Płukanie

## Zadanie -2

Poniżej podano diagram zmian stanów dla pewnej klasy. Podaj jakie czynności będą kolejno wykonane przez obiekt tej klasy dla następującej sekwencji zdarzeń:

- **utworzenie obiektu,**
- E1,
- E3,
- E2,
- E1.

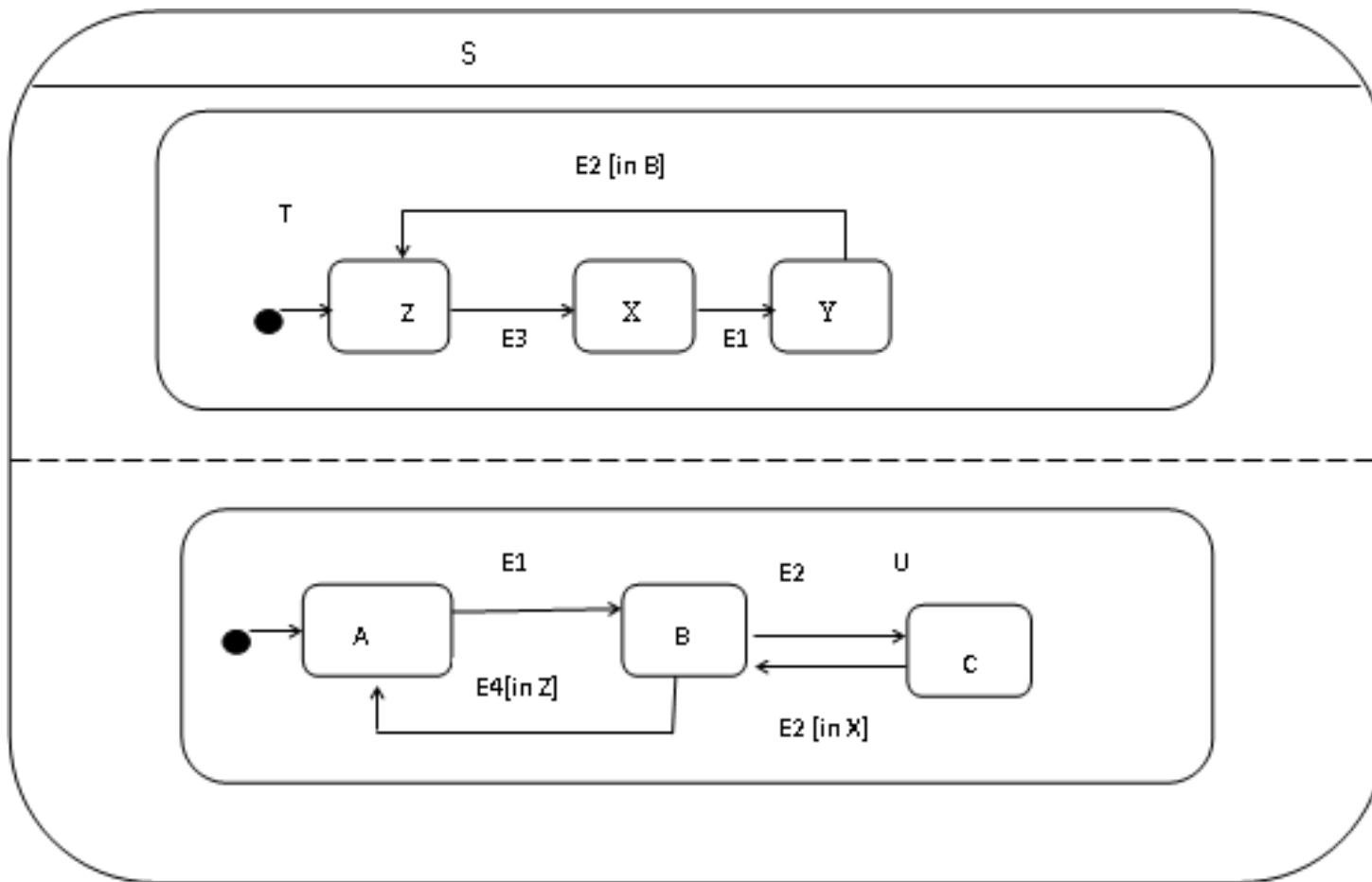
Uzasadnij swoje rozwiązanie.



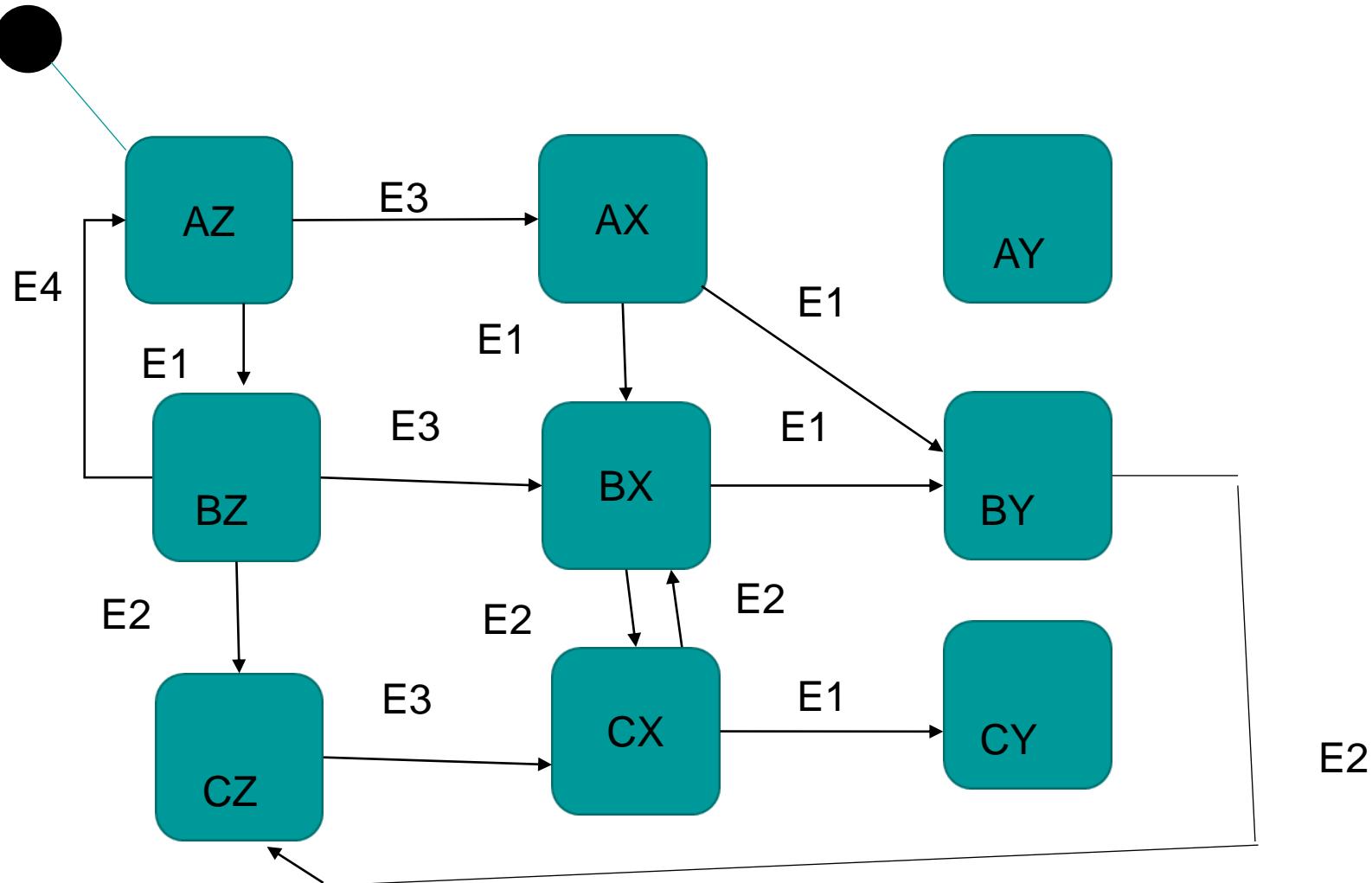
# Zadanie -1- rozwiązanie początek

<b>zdarzenie</b>	<b>Stan osiągnięty</b>	<b>Czynności</b>
Utworzenie obiektu	stan1	akcja1/entry; akcja3/do
E1	stan2	akcja2/exit; akcja4/E1; akcja7/entry; akcja9/do
E3	stan2	akcja10/E3; akcja9/do
E2	stan2	akcja9/do

# zadanie



# rozwiązanie



# Diagramy implementacyjne:

## Diagram komponentów

## Diagram rozmieszczenia

---

Dr hab. inż. Ilona Bluemke

# Diagram komponentów

Model zależności komponentów oprogramowania

Elementy modelu:

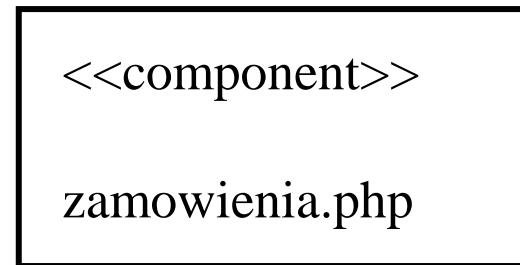
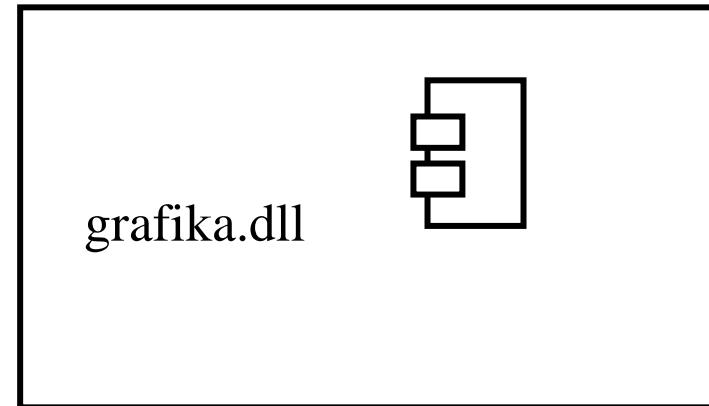
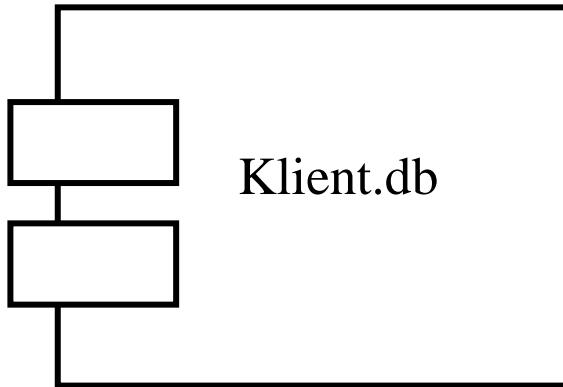
- komponenty
- zależności — wskazują potrzebę dostosowania komponentów

# Komponent – element oprogramowania

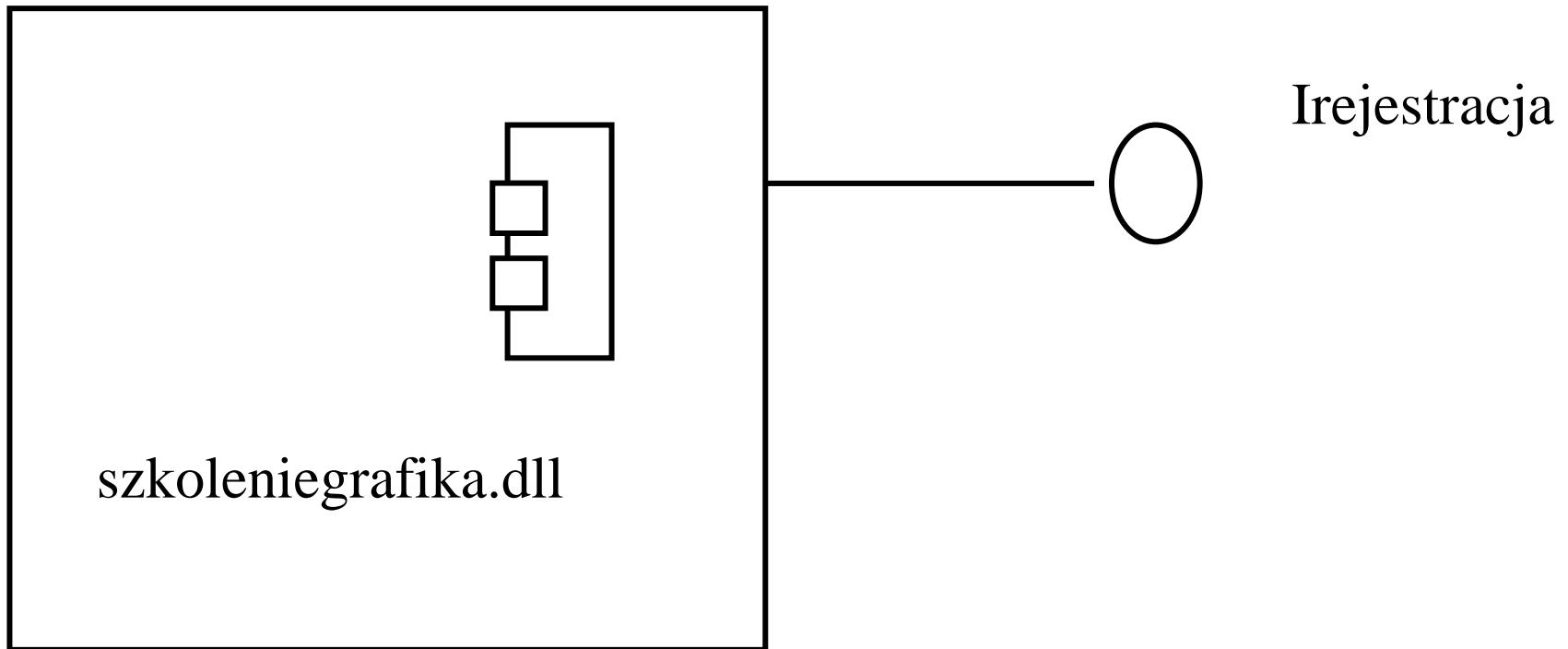
Może to być:

- program wykonywalny - <<executable>>,
- biblioteka - <<library>>
- fizyczne bazy danych, tabele baz danych - <<table>>
- podsystemy - << subsystem>>
- komponenty przetwarzające - << service>>

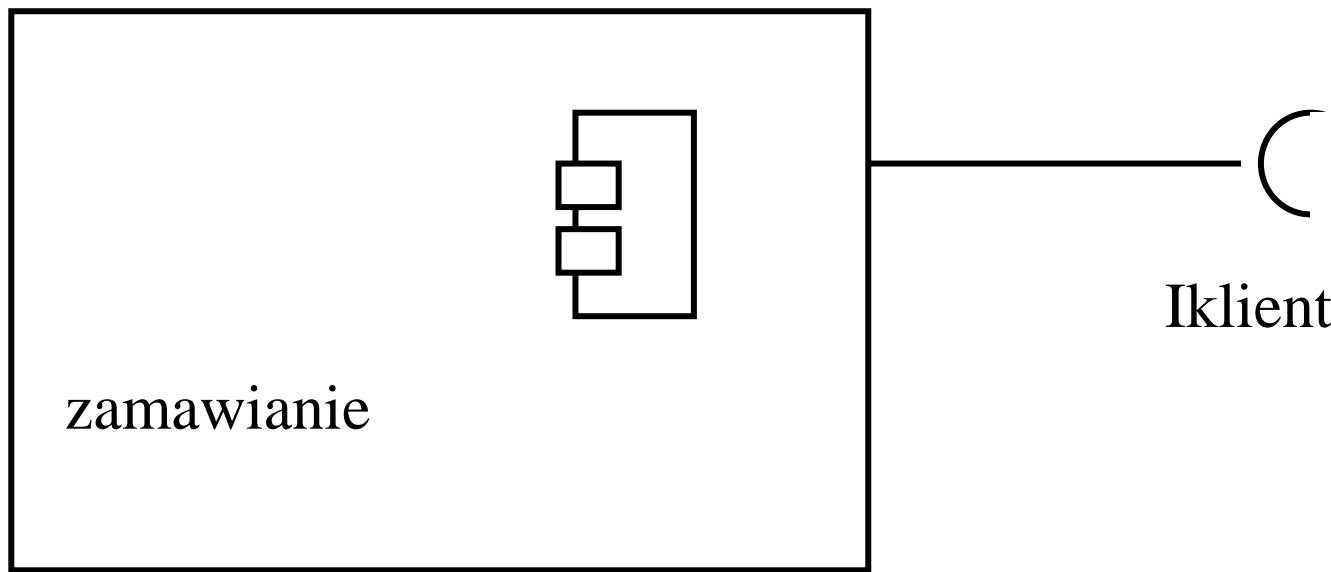
# symbole graficzne komponentu



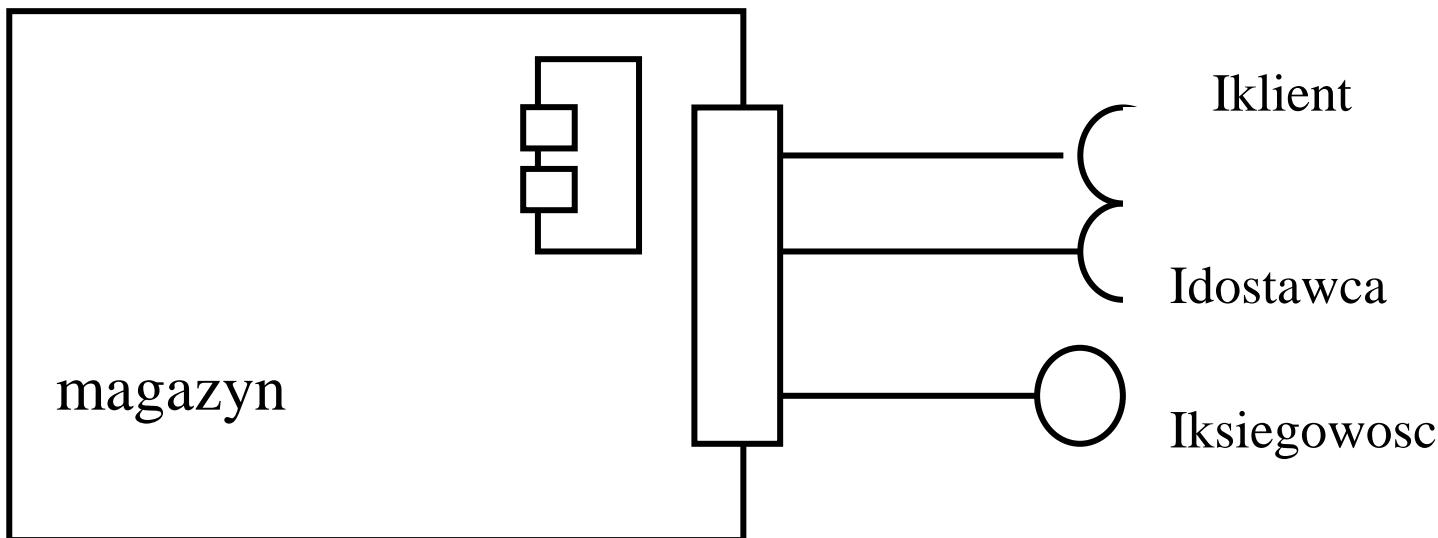
# Interfejs udostępniany



# Interfejs wymagany (pozyskujący)

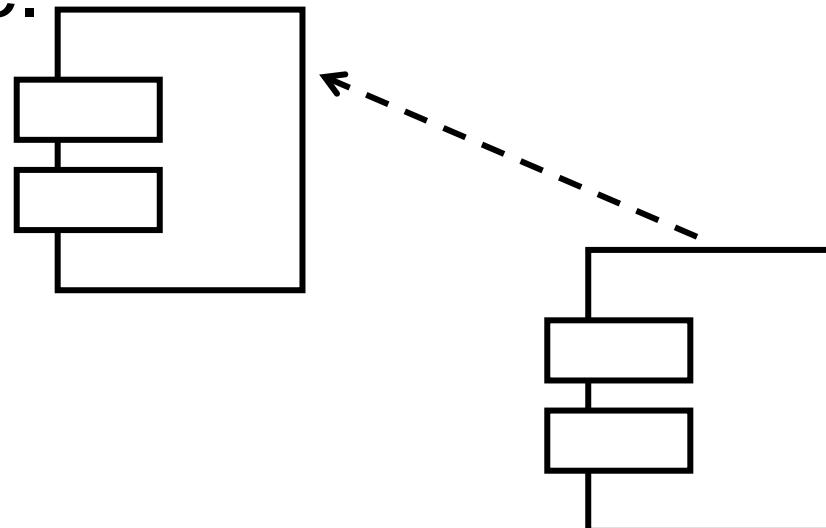


# Porty (bramy)



# Zależności

- Pokazują, że komponent korzysta z usług oferowanych przez inny komponent.
- Oznaczenie – linia przerywana skierowana w kierunku dostarczyciela (wskazują na interfejsy wymagane).



# Realizacja i konektory

- Realizacja - wskazuje na interfejsy udostępnione



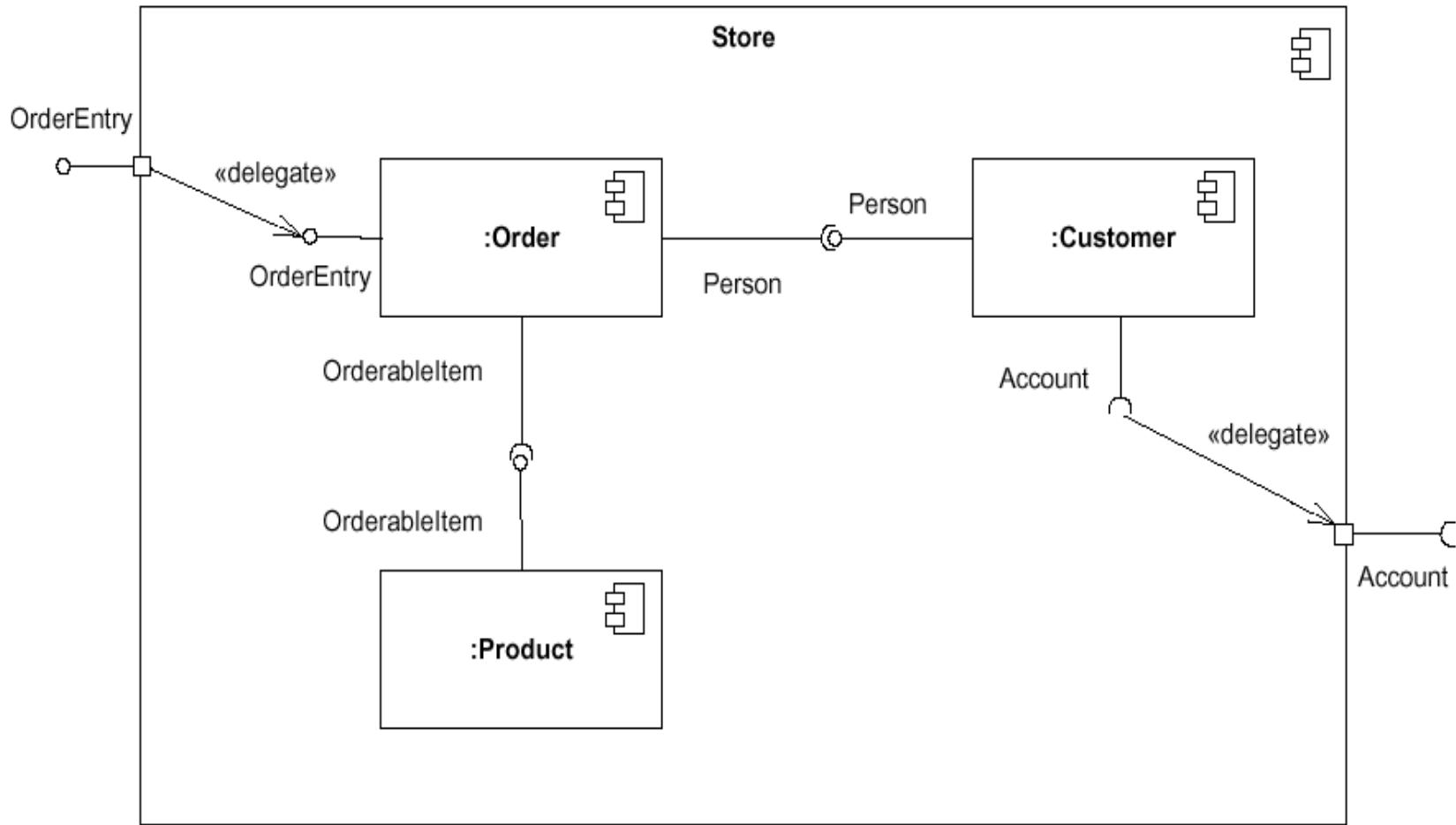
- Konektor delegowany



- Konektor składany

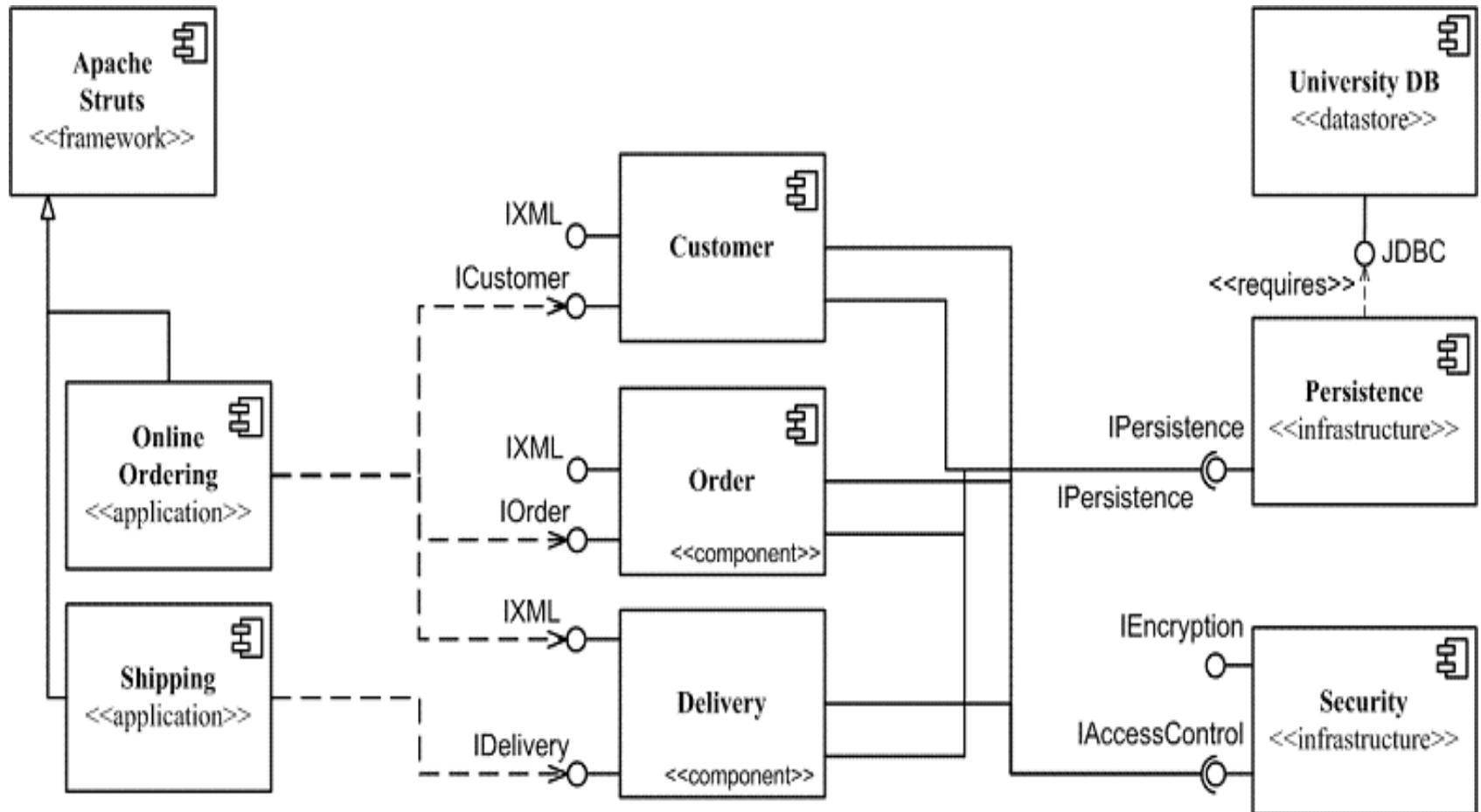


# przykład



# Omówienie przykładu

- Przykład komponentu w konwencji białej skrzynki
- Pokazano – połączenia interfejsów
- konektory typu <<delegate>> strzałka wskazuje kierunek delegacji
- Port OrderEntry deleguje komunikację do interfejsu wewnętrznego komponentu



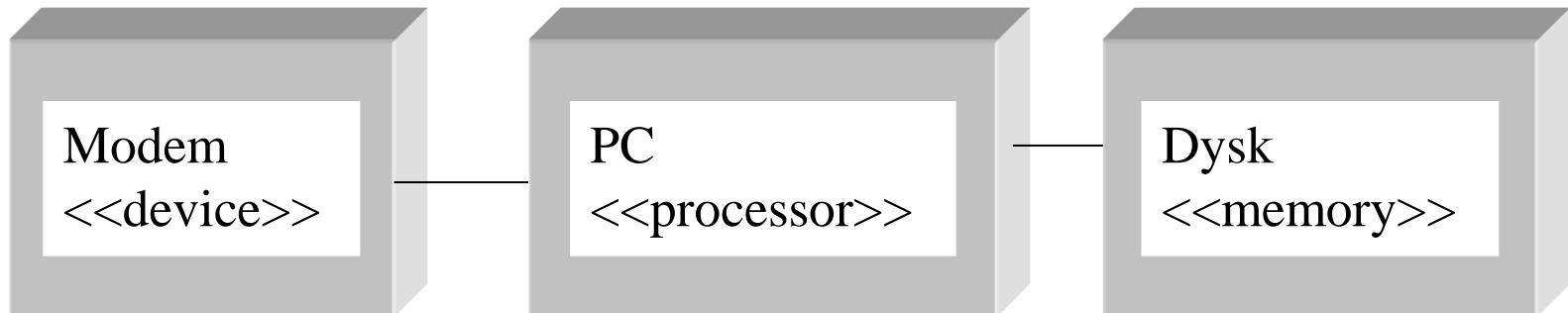
# Diagram rozmieszczenia - deployment diagram

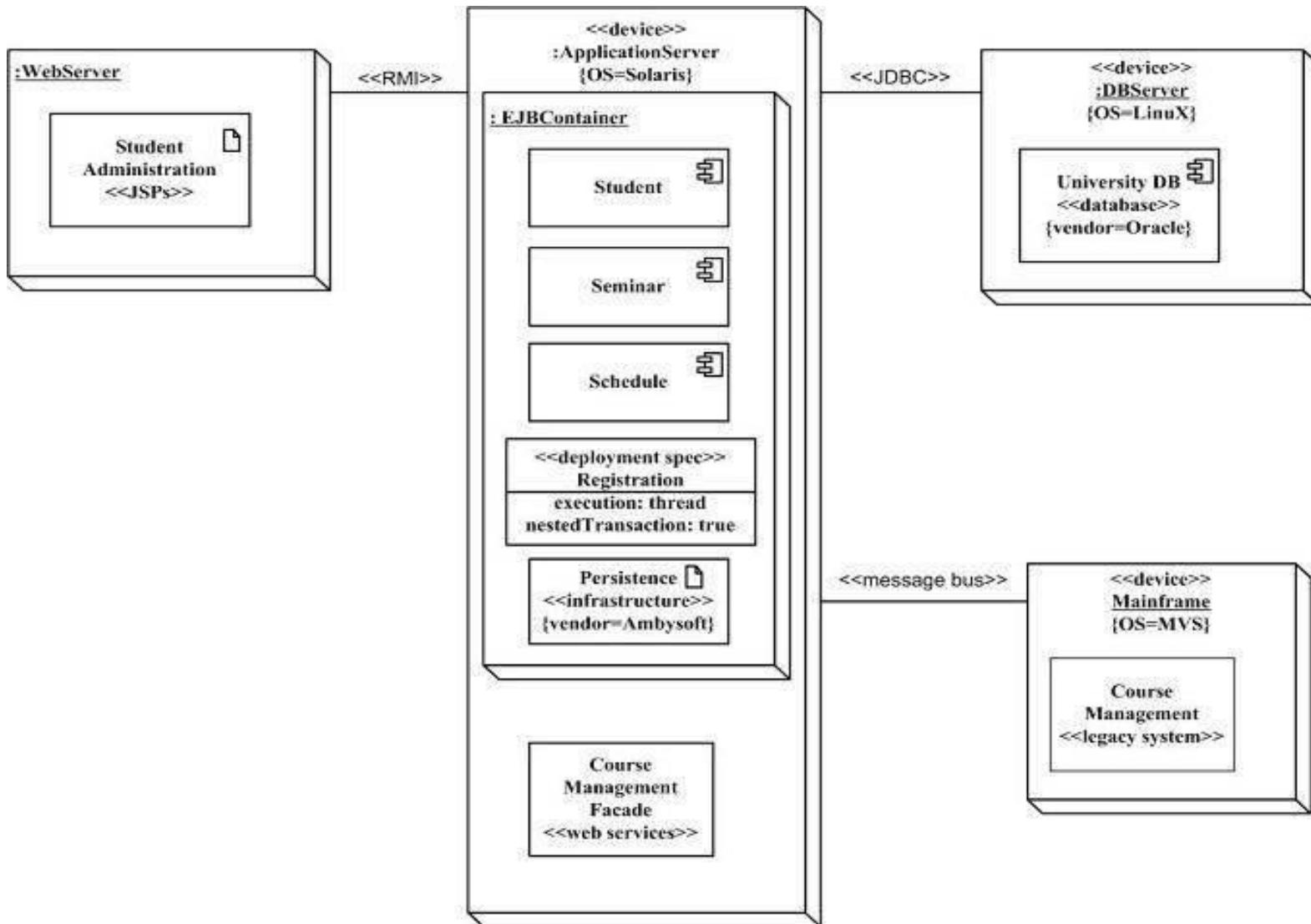
- Pokazuje różne elementy sprzętu wchodzącego w skład systemu i rozmieszczenie oprogramowania na sprzętie

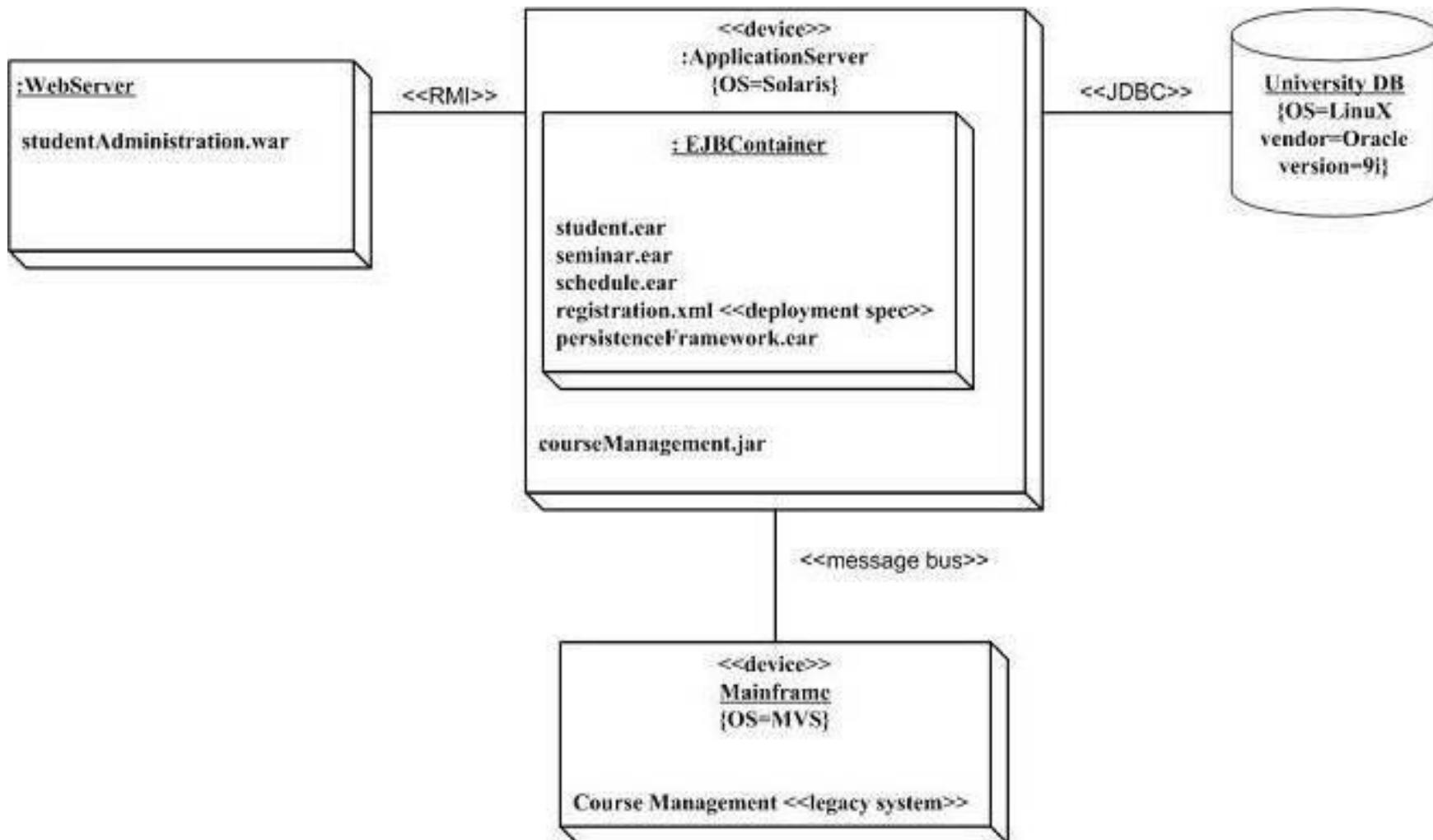


# Przykład diagramu rozmieszczenia

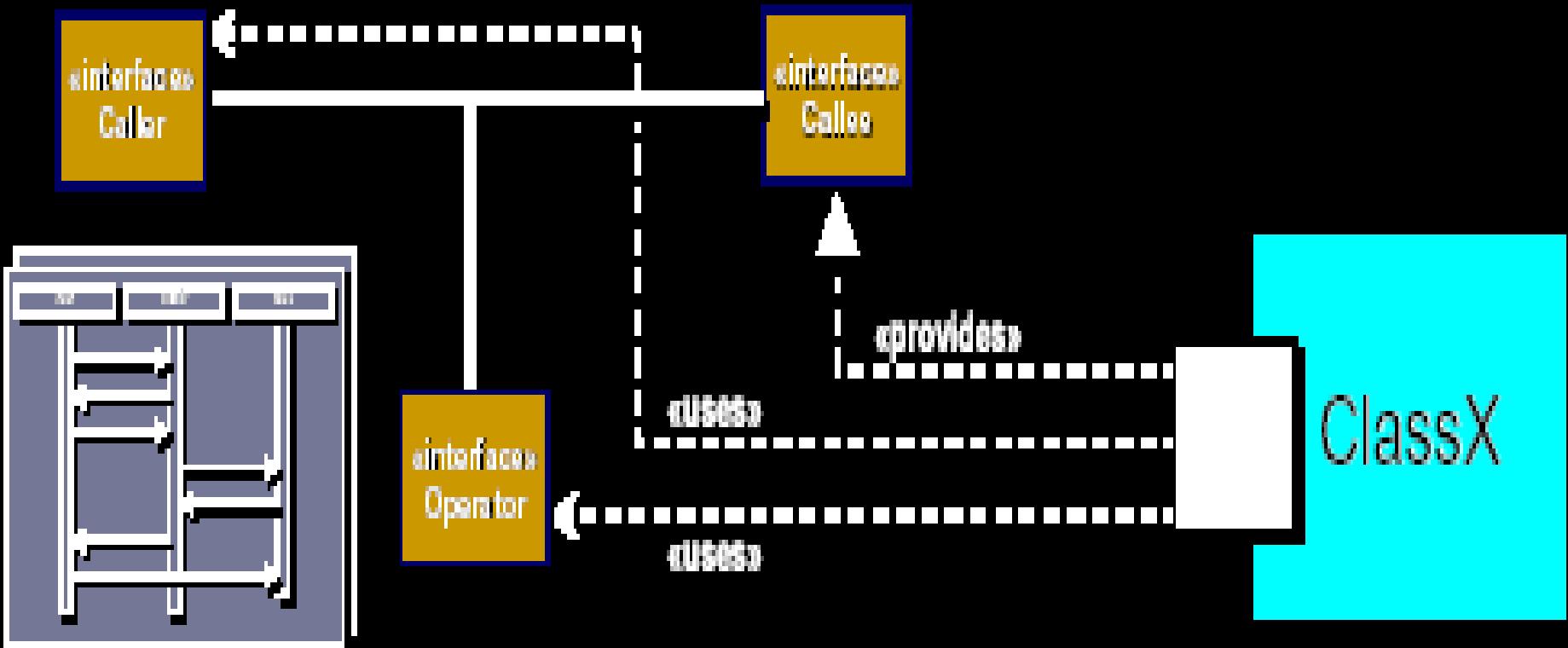
- Można określić typ urządzenia
- pokazywać połączenia – dwukierunkowe , określić krotności połączeń







## Operator Assisted Call



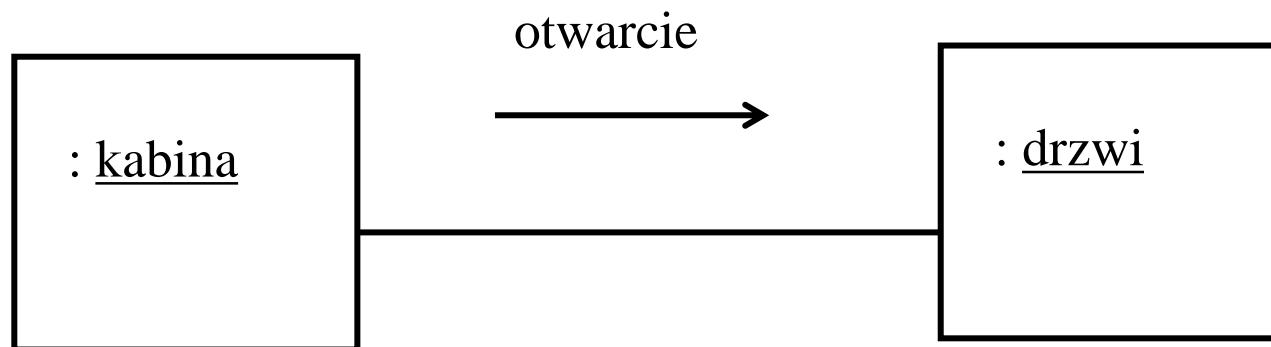
# Diagramy komunikacji

---

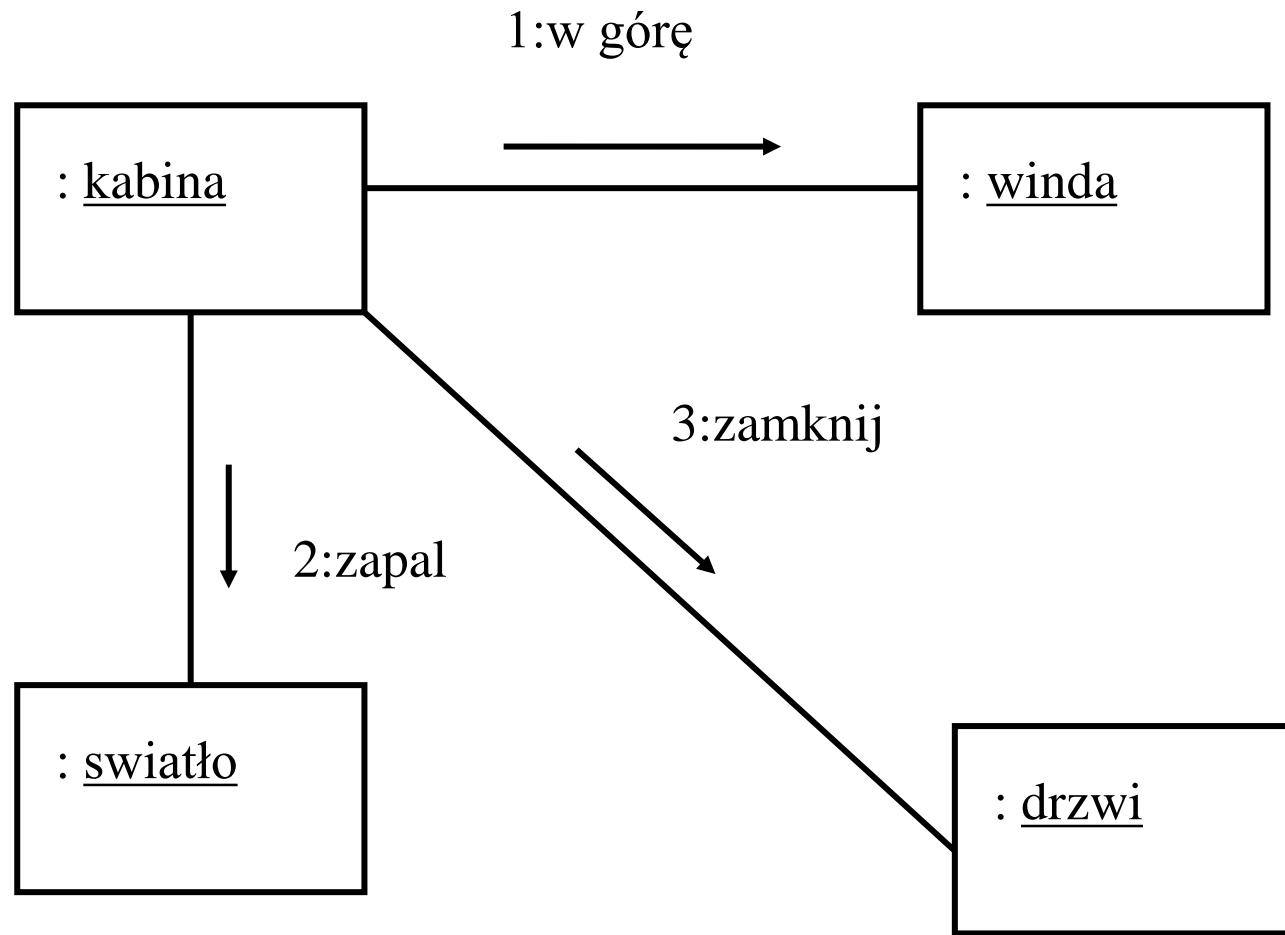
Dr hab. inż. Ilona Bluemke

# Diagram komunikacji

- Przedstawia wzajemną komunikację między obiektami. W przeciwieństwie do diagramów sekwencji nie pokazuje przepływu komunikatów rozłożonego w czasie, tylko obiekty źródłowe i docelowe tych przepływów.



# Numerowanie komunikatów

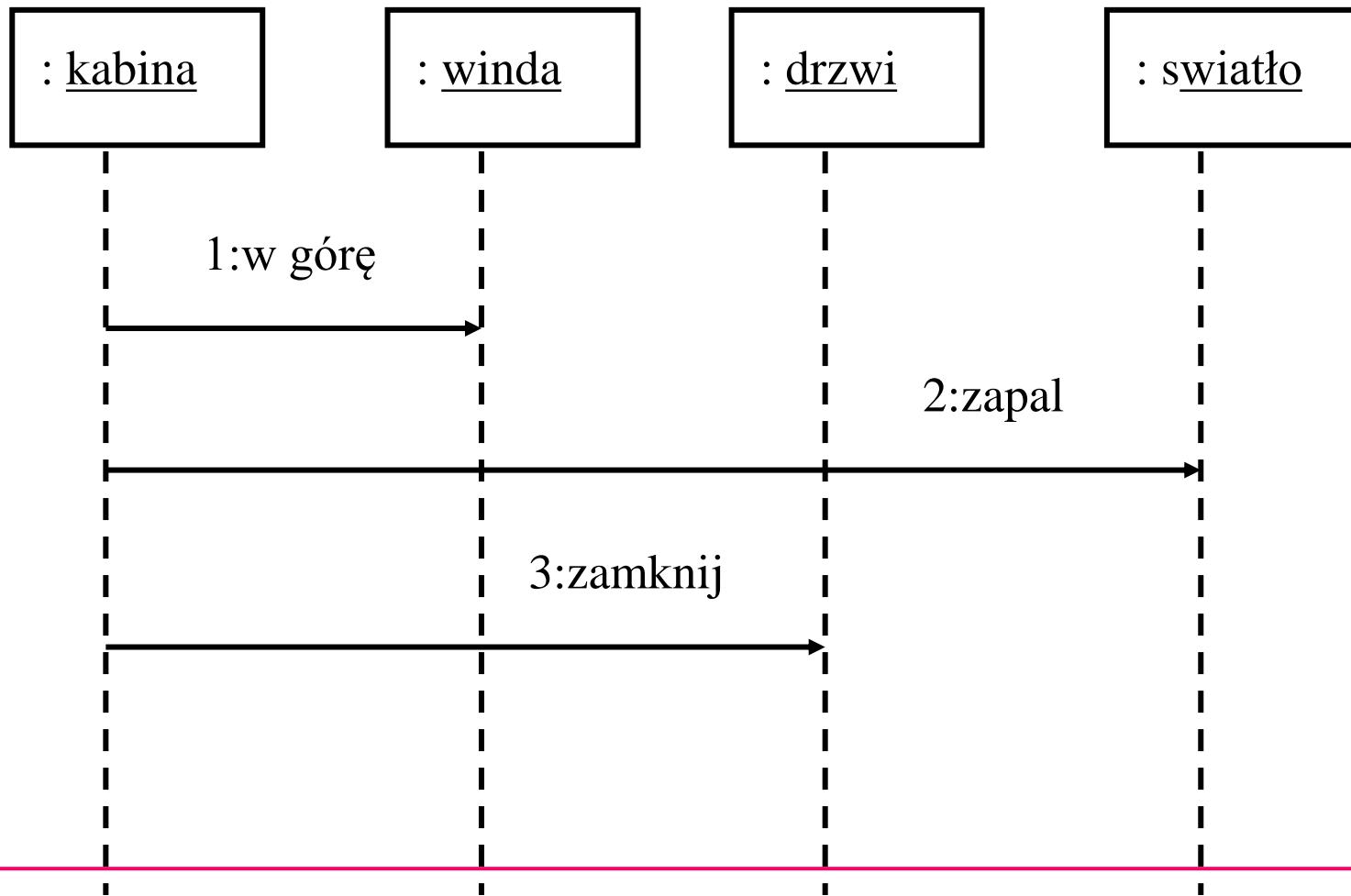


# Rodzaje komunikatów

(jak na diagramach interakcji):

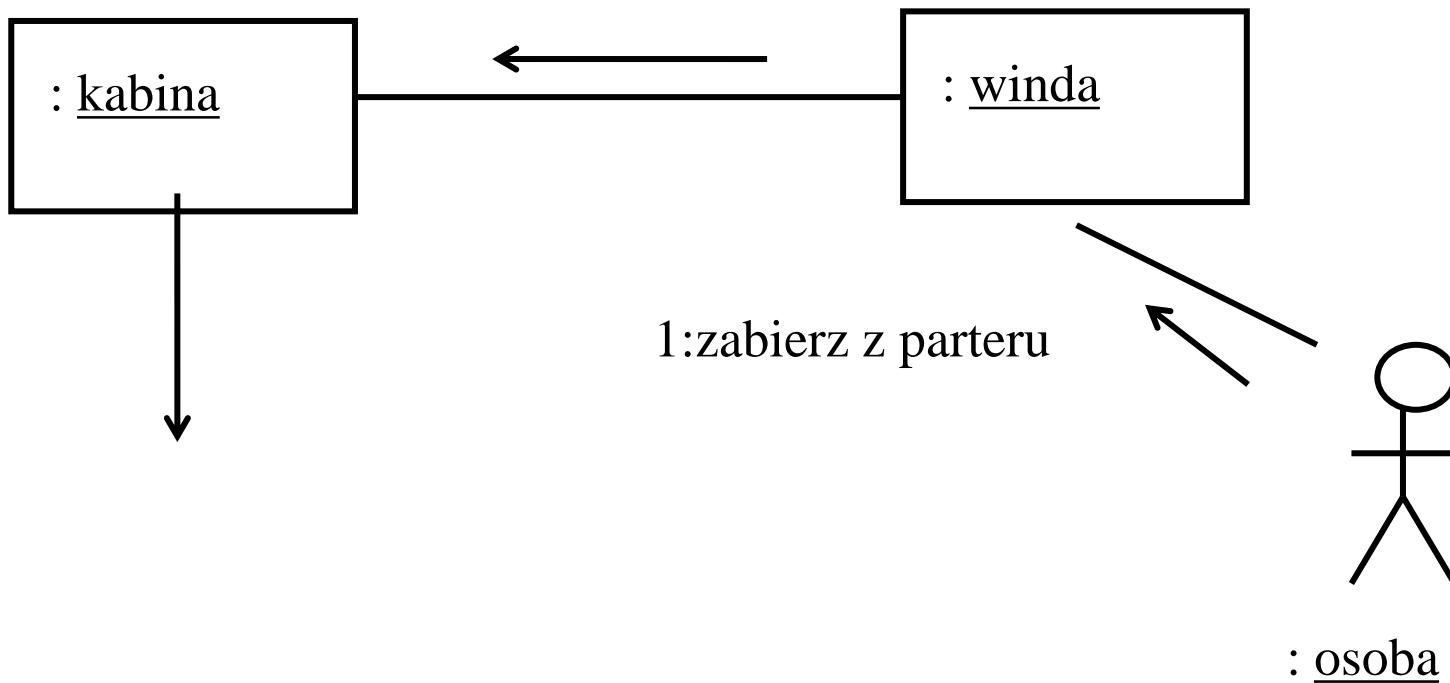
- asynchroniczny 
- synchroniczny 
- zwrotny 

# Izomorfizm diagramów sekwencji i komunikacji



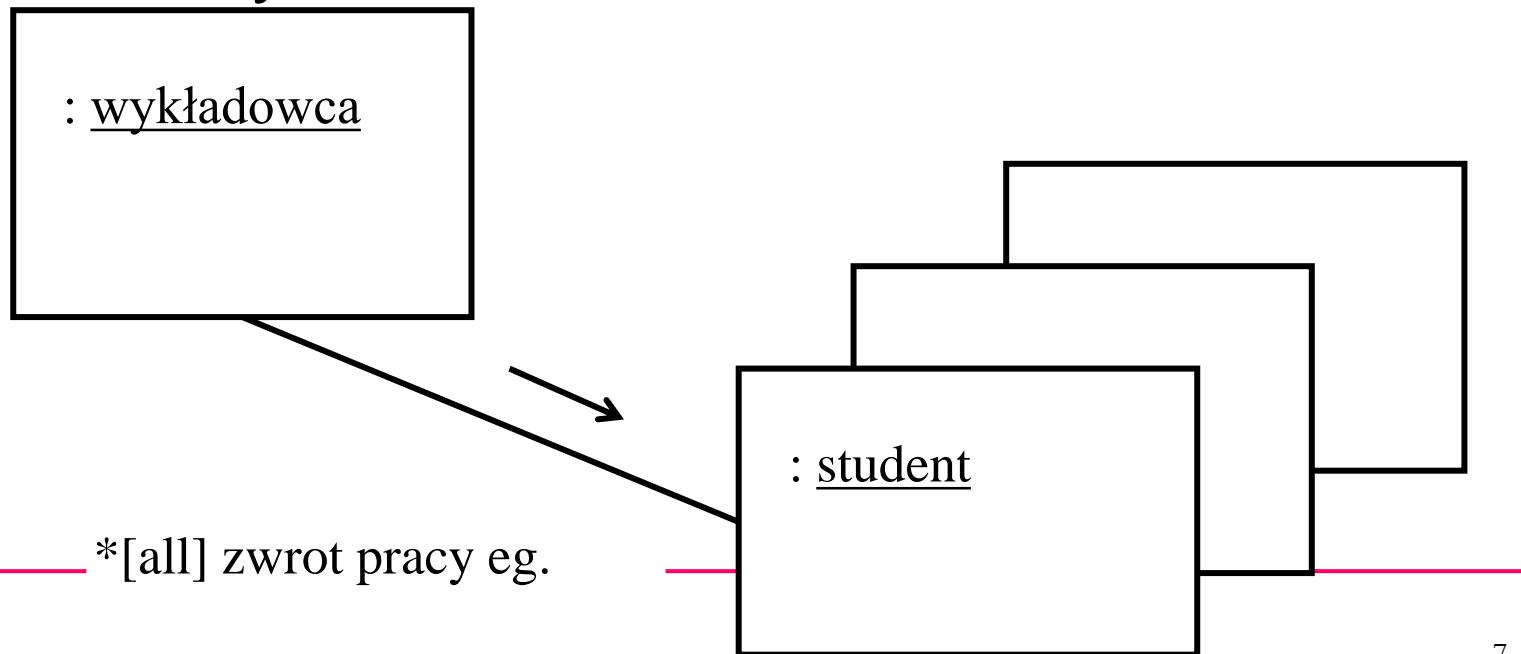
# Aktor na diagramach komunikacji

2: dodaj parter do celów



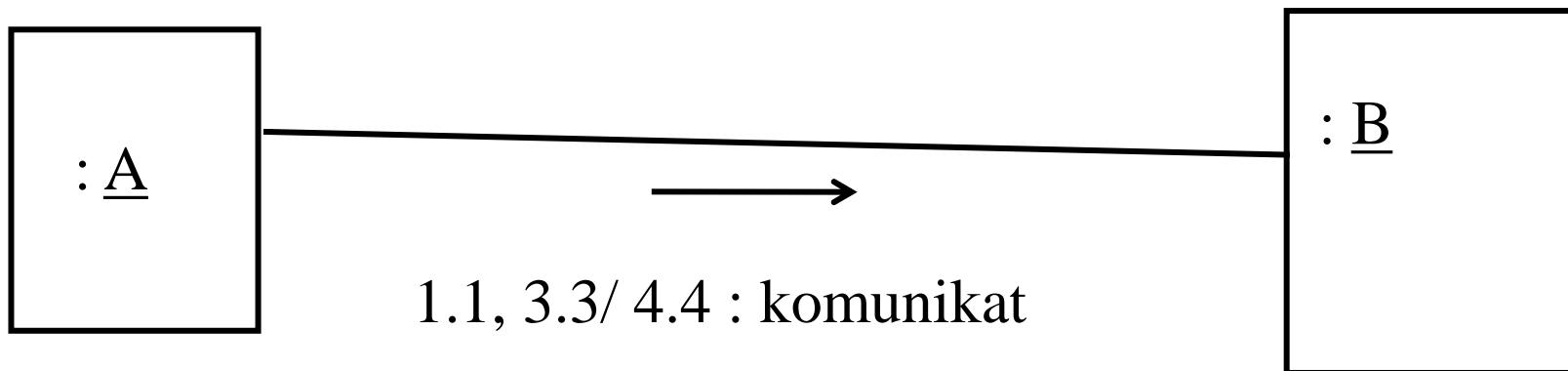
# grupa obiektów otrzymuje ten sam komunikat

- Komunikat jest wysyłany do wszystkich obiektów student, znacznik \* jest nadmiarowy w połączeniu z notacją obiektów wielokrotnych.



# Poprzedniki komunikatów

- Można podać listę komunikatów poprzedzających wysłanie komunikatu aktualnego. Np. „komunikat będzie wysłany gdy były wysłane komunikaty 1.1 i 3.3



# Zagnieżdżanie komunikatów

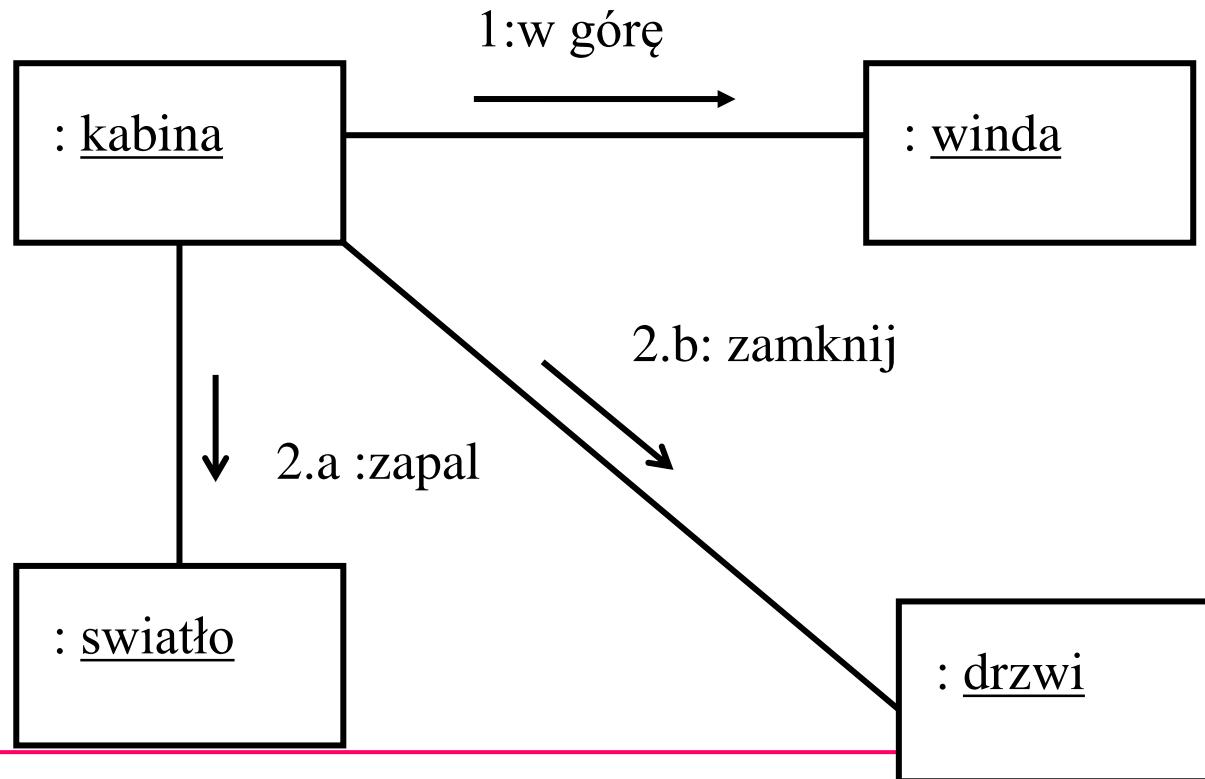
- Kolejność wywołania komunikatów w zagnieżdżeniu jest określona kolejnymi liczbami naturalnymi i pozycjami dziesiętnymi (numeracja kwalifikowana).
- Głębokość zagnieżdżenia nie jest ograniczona.
- Zagnieżdżenie jest równoznaczne z pojęciem wątku (pojedyncza ścieżka przepływu sterowania wykonana przez program).

Np.

```
1 -> 1.1    -> 1.1.1  
2 -> 2.1  
          2.2
```

# Współbieżność komunikatów

- litera oznacza równoległy przepływ np. 2.a i 2.b mogą być wysyłane w tym samym czasie.

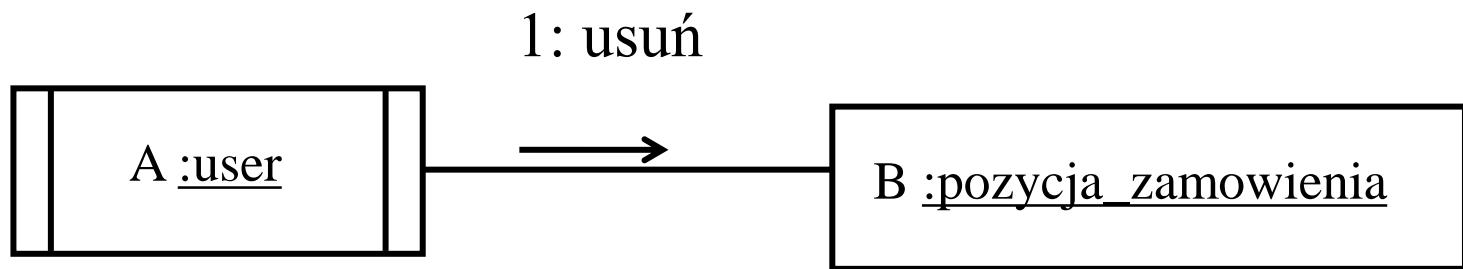


# Obiekty aktywne

Mogą inicjować operacje własne, innych obiektów, wątki. np.

- inicjują wysłanie pierwszego komunikatu w serii komunikatów zagnieżdżonych,
- wykonują obliczenia na podstawie danych przechowywanych w innych obiektach
- samoistnie wysyłają komunikaty w określonym czasie

# Przykład obiektu aktywnego



# Diagramy sekwencji

## UML 2.x

---

Dr hab. inż. Ilona Bluemke

# plan

- Zastosowanie diagramów sekwencji
- Typy komunikatów
- Precyzyjne modelowanie interakcji

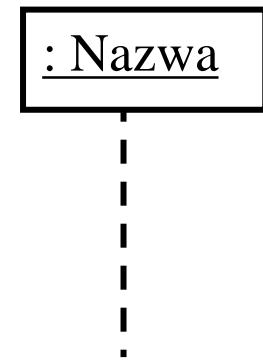
# Diagramy sekwencji (sequence diagrams)

- Modelują dynamiczne cechy systemu.
- Stanowią pomoc do tworzenia diagramów stanów i do testowania końcowego programu.
- Każdy pojedynczy diagram dotyczy jednej ścieżki wywołania gotowego programu będącego końcowym wynikiem projektu.
- Diagramy przedstawiają sekwencję odwołań obiektów rozłożoną w czasie. Czas rośnie w dół diagramu.

# Elementy diagramu sekwencji

- obiekty
- linie życia
- komunikaty

Obiekt uczestniczący w interakcjach:

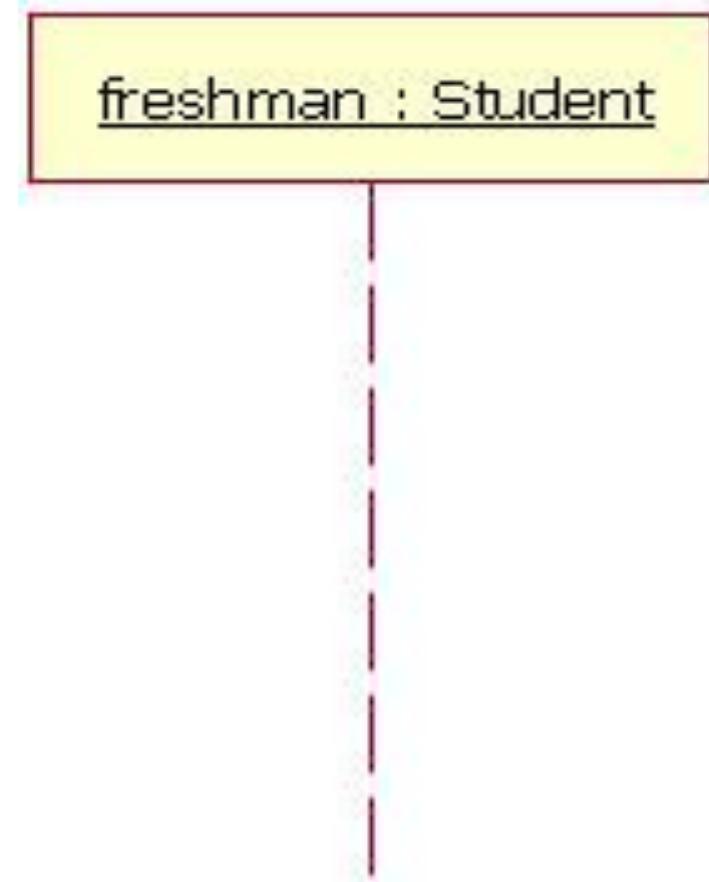


# Linie życia

## Obiekty lub role

- freshman – nazwa obiektu - instancji klasy
- Student – nazwa klasy

Linia życia – upływ czasu z góry do dołu



# Komunikat

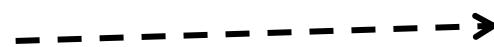
- **asynchroniczny**



- **synchroniczny**

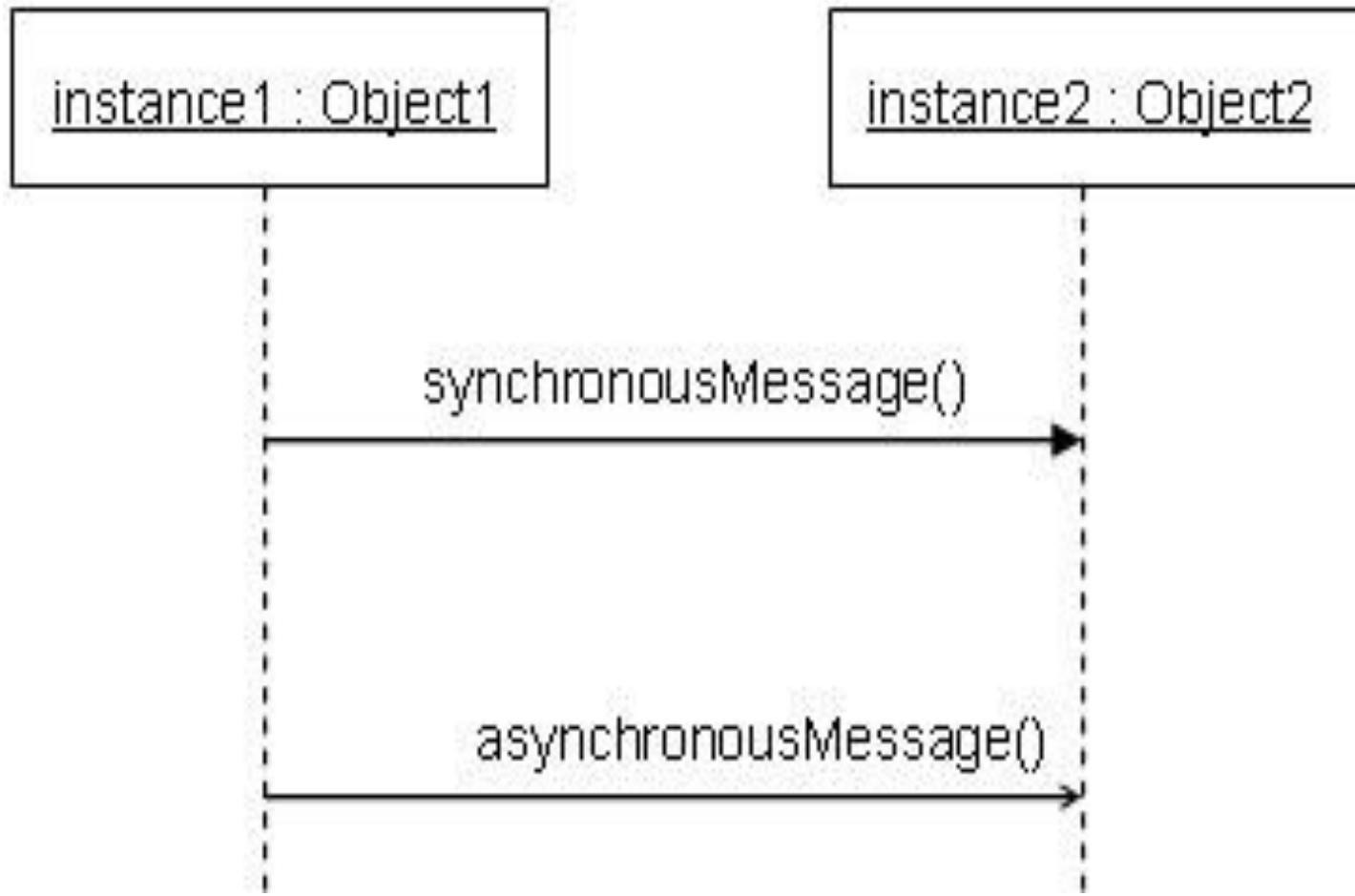


- **zwrotny (powrót sterowania)**

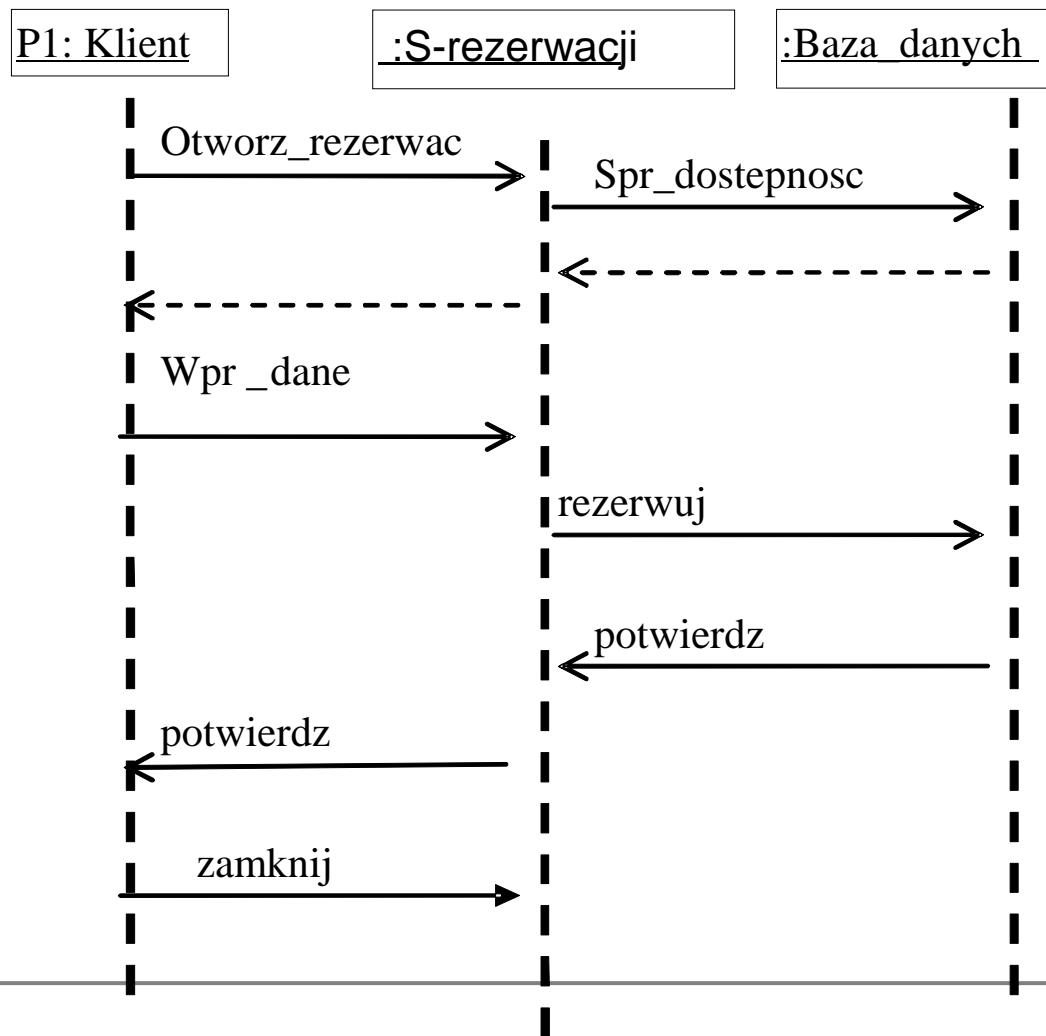


**komunikat – metoda w klasie odbierającej.**

# Komunikaty synchroniczne i asynchroniczne

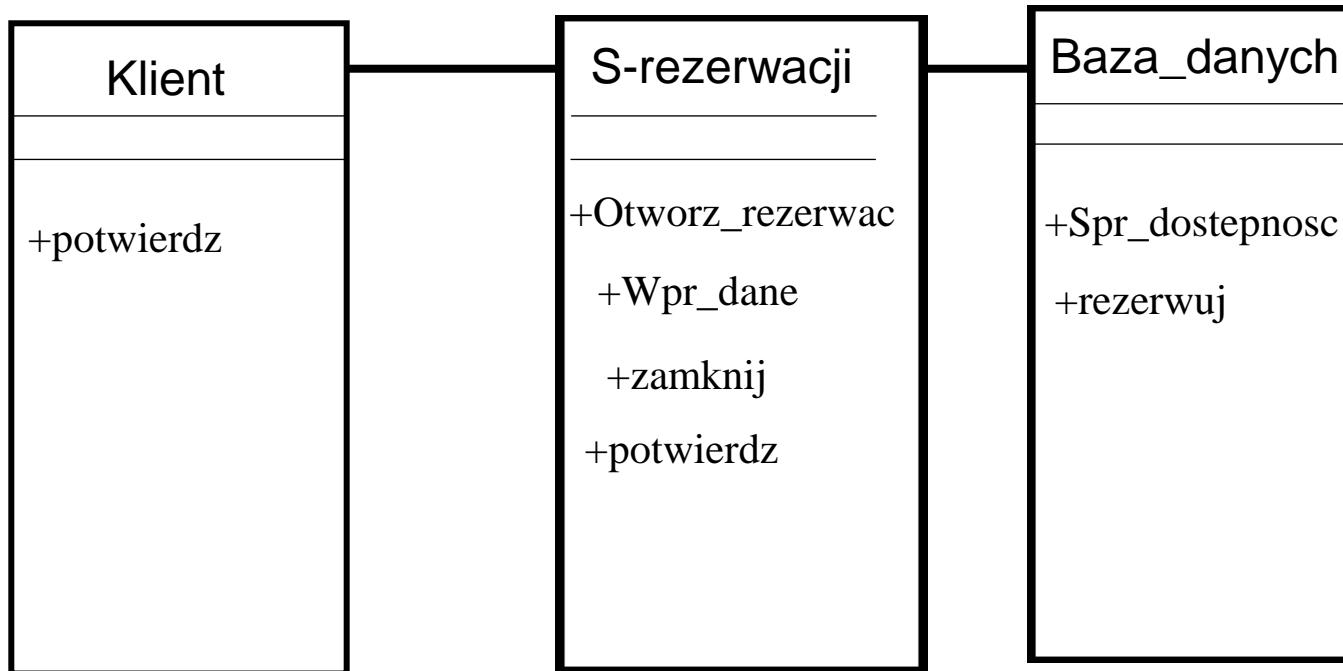


# przykład



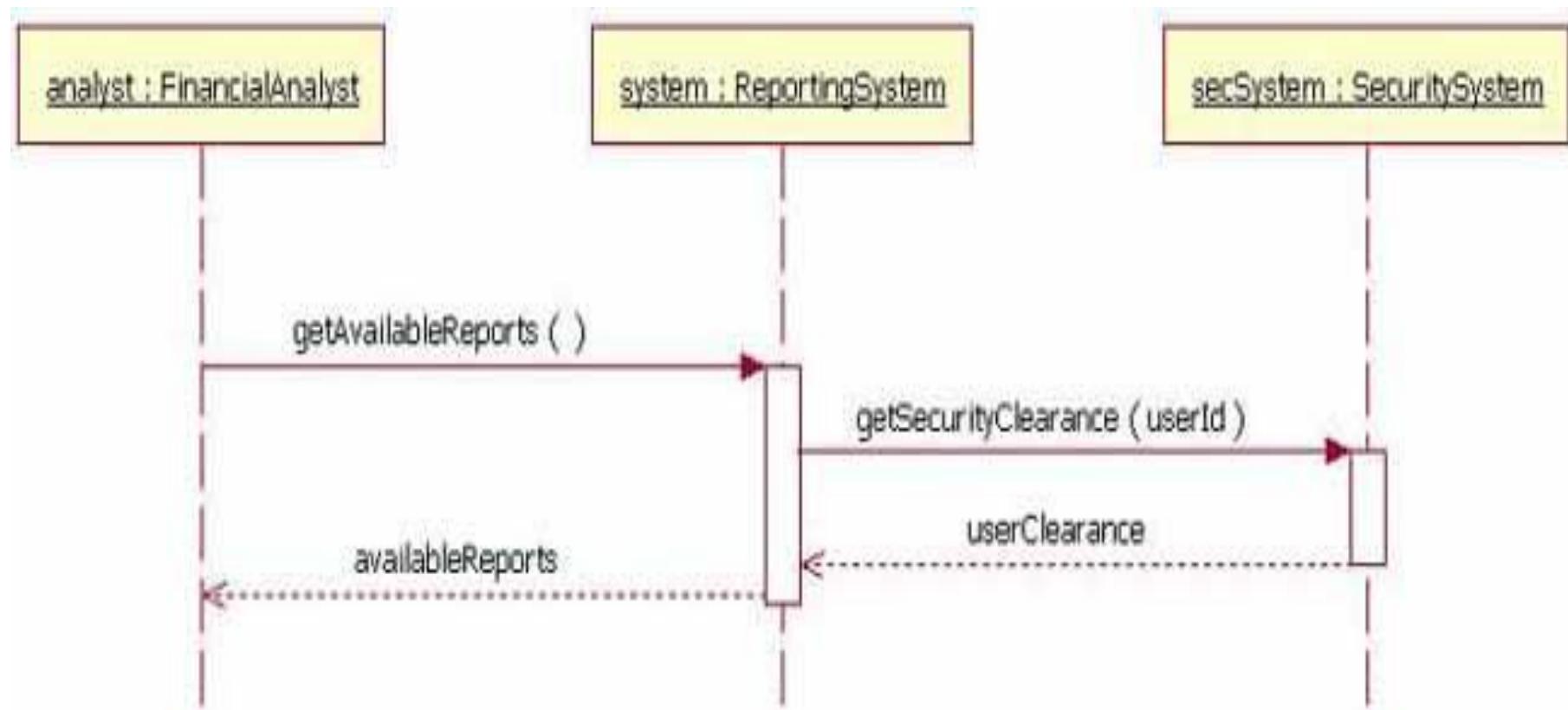
# Diagram klas- spójny z diagramem sekwencji

Operacje w klasie odbierajacej



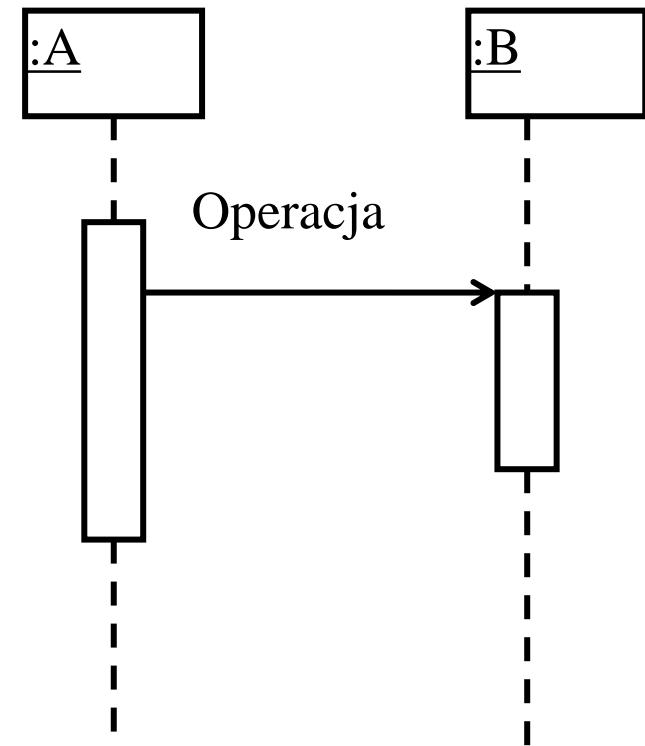
Asocjacje dwukierunkowe bo dwukierunkowa komunikacja miedzy obiektami

# Wiadomości zwrotne (return messages)

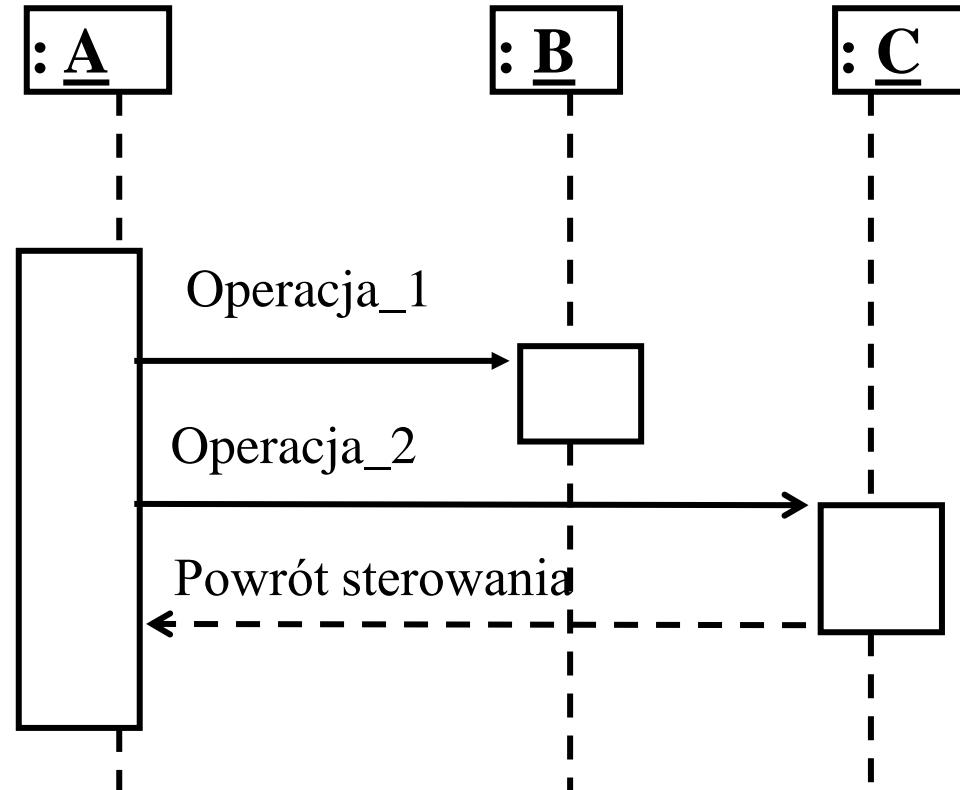


# Czas aktywacji obiektu

- obiekt A aktywuje obiekt B – komunikat asynchroniczny, może dalej się wykonywać A i B.
- W przypadku gdy komunikat jest synchroniczny, A zostaje zablokowany do czasu, aż do niego wróci sterowanie (zakończenie wykonania metody)

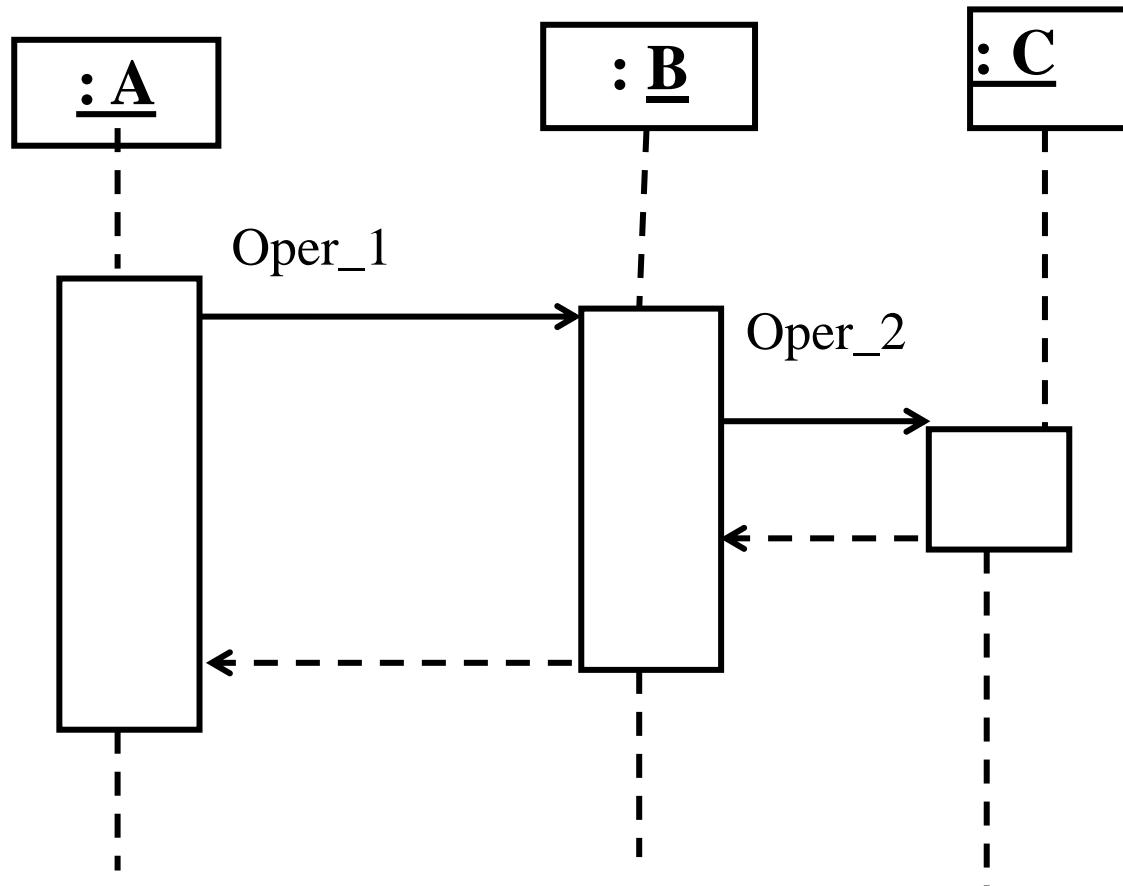


# Struktura sterowania



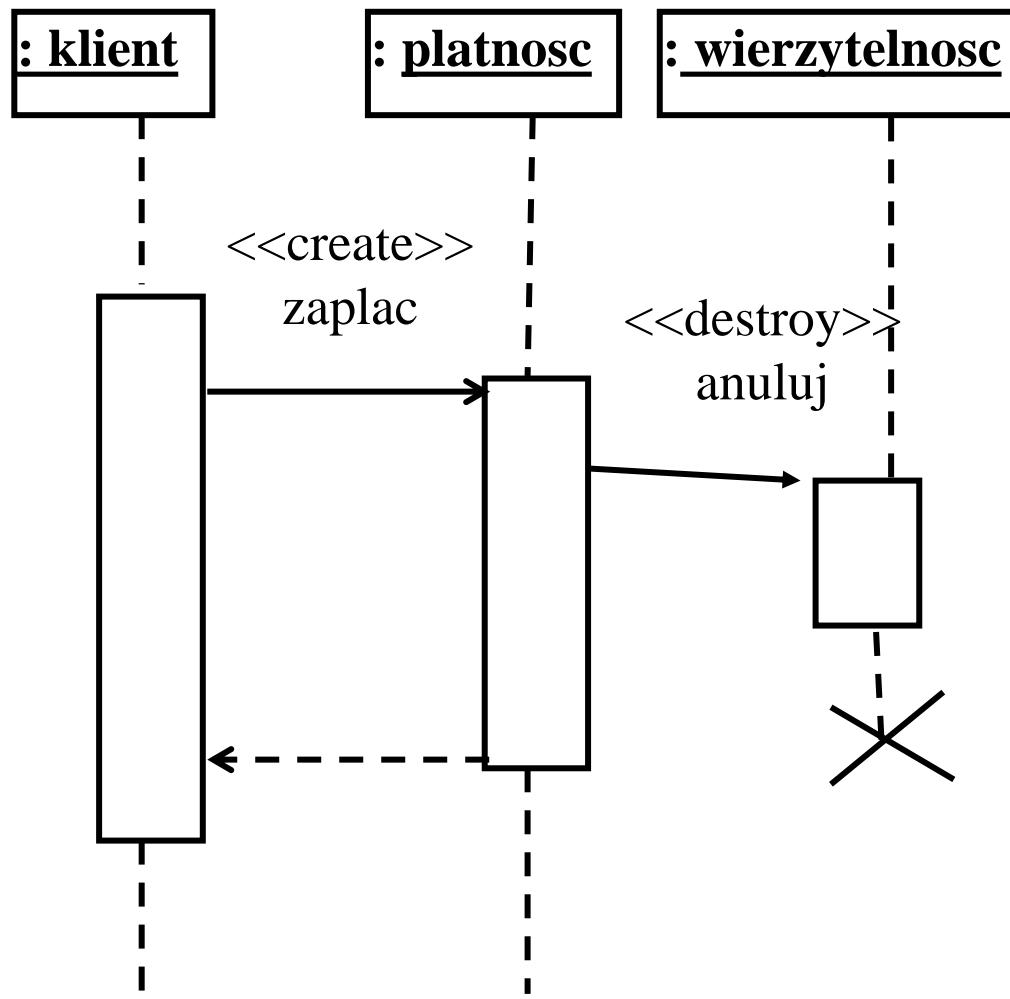
Struktura scentralizowana

# Sterowanie zdecentralizowane

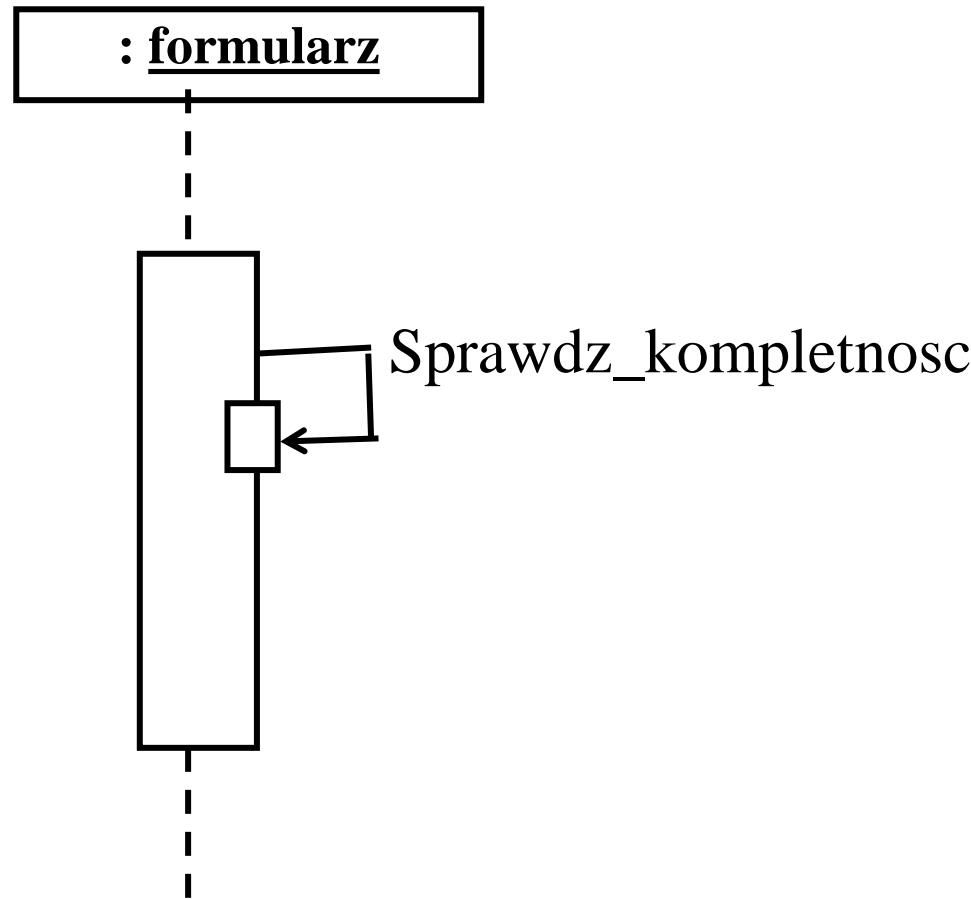


**Struktura zdecentralizowana**

# Tworzenie i niszczenie obiektów



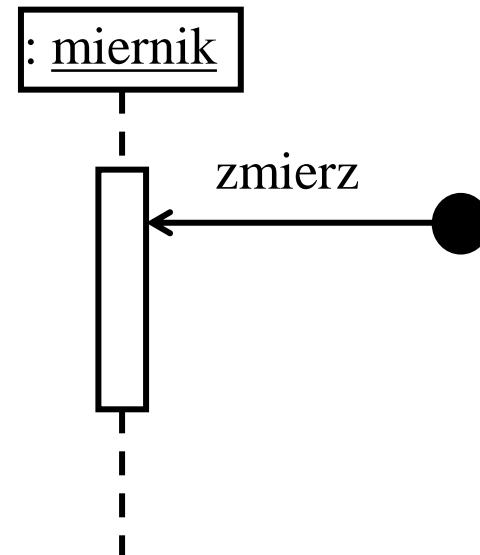
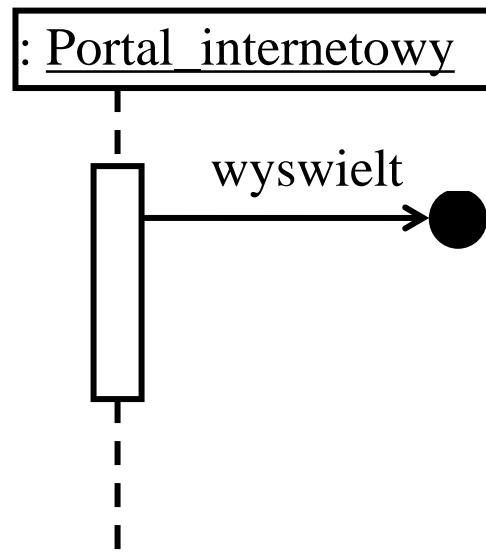
# Samowywołanie (message to self)



# Komunikat utracony i znaleziony

**utracony**  
(nieznany odbiorca)

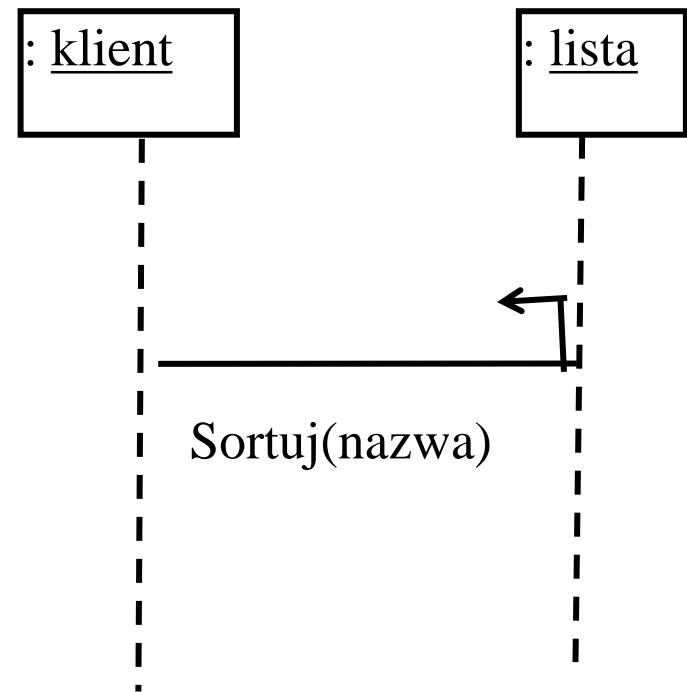
**znaleziony**  
(nieznany nadawca)



# Komunikat opcjonalny (balking message)

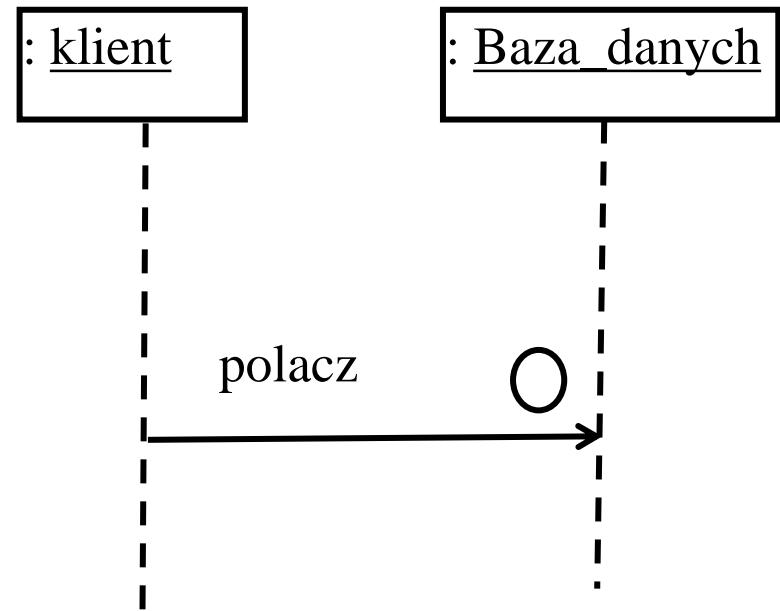
Nadawca wysyła komunikat oczekując, że odbiorca jest gotowy do jego natychmiastowej obsługi.

Jeżeli komunikat nie może zostać przyjęty nadawca nie podejmuje kolejnych prób jego wysłania (może nie być obsłużony).

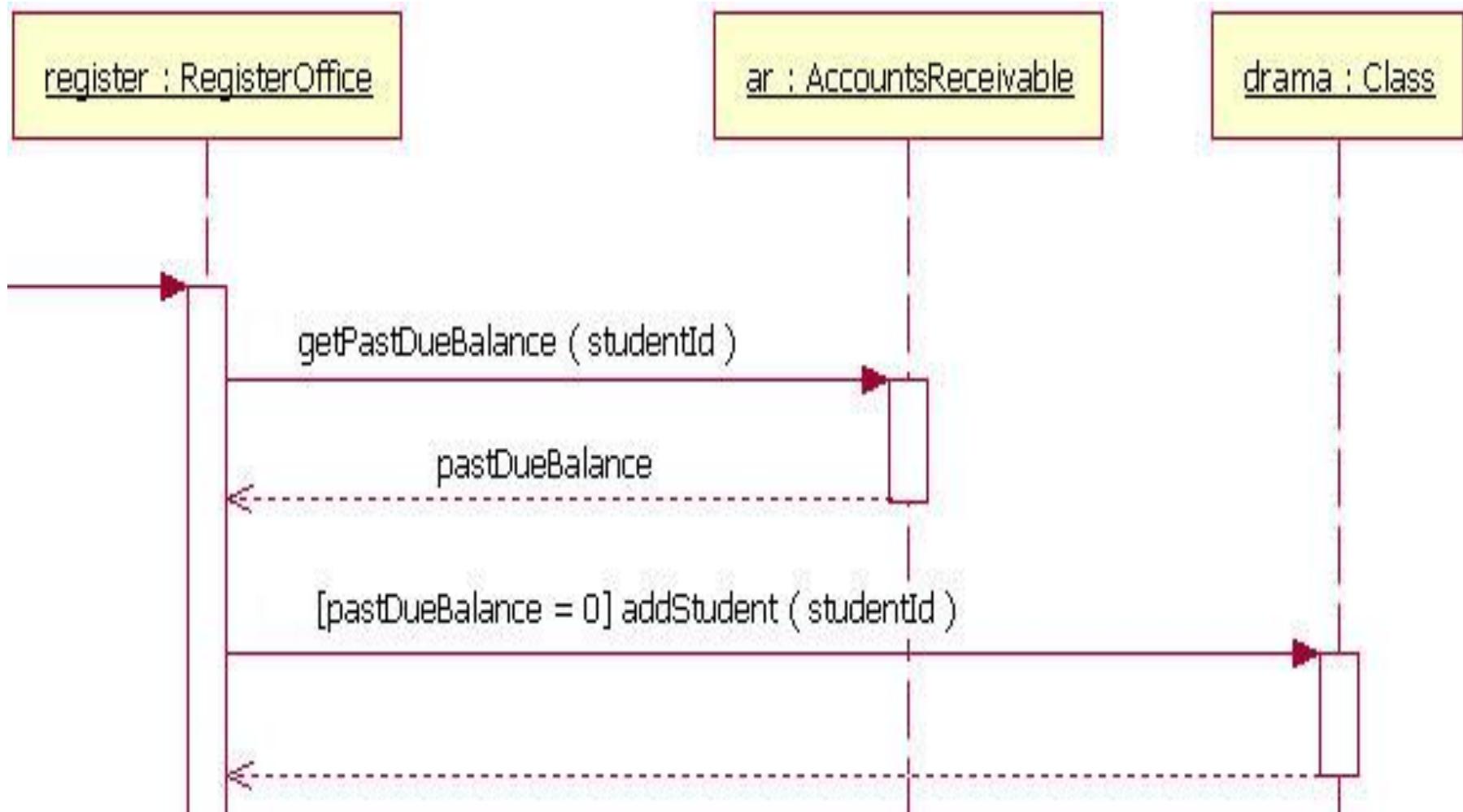


# Komunikat oczekujący (timeout message)

Nadawca wysyła komunikat oczekuje, że odbiorca obsłuży go w ciągu określonego okresu czasu. Jeżeli komunikat nie może zostać obsłużony w tym czasie to nadawca rezygnuje z danej interakcji.

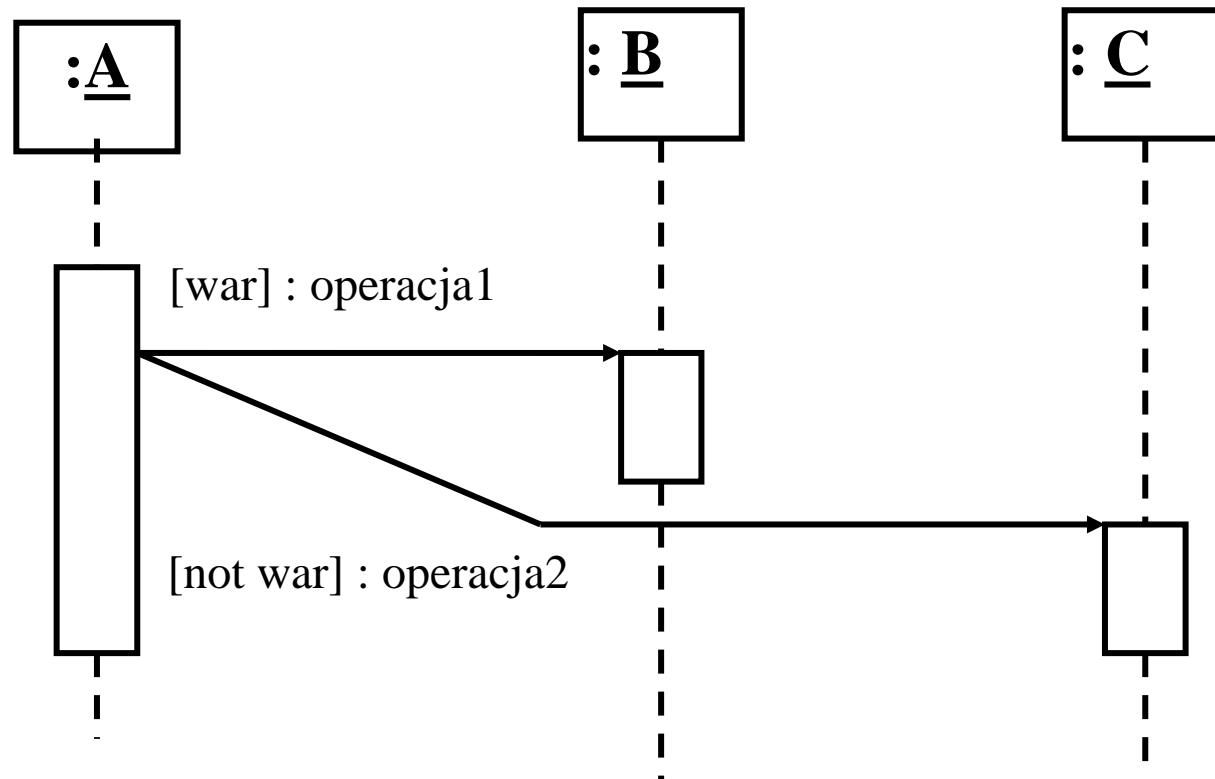


# Warunki – dozory (guards)

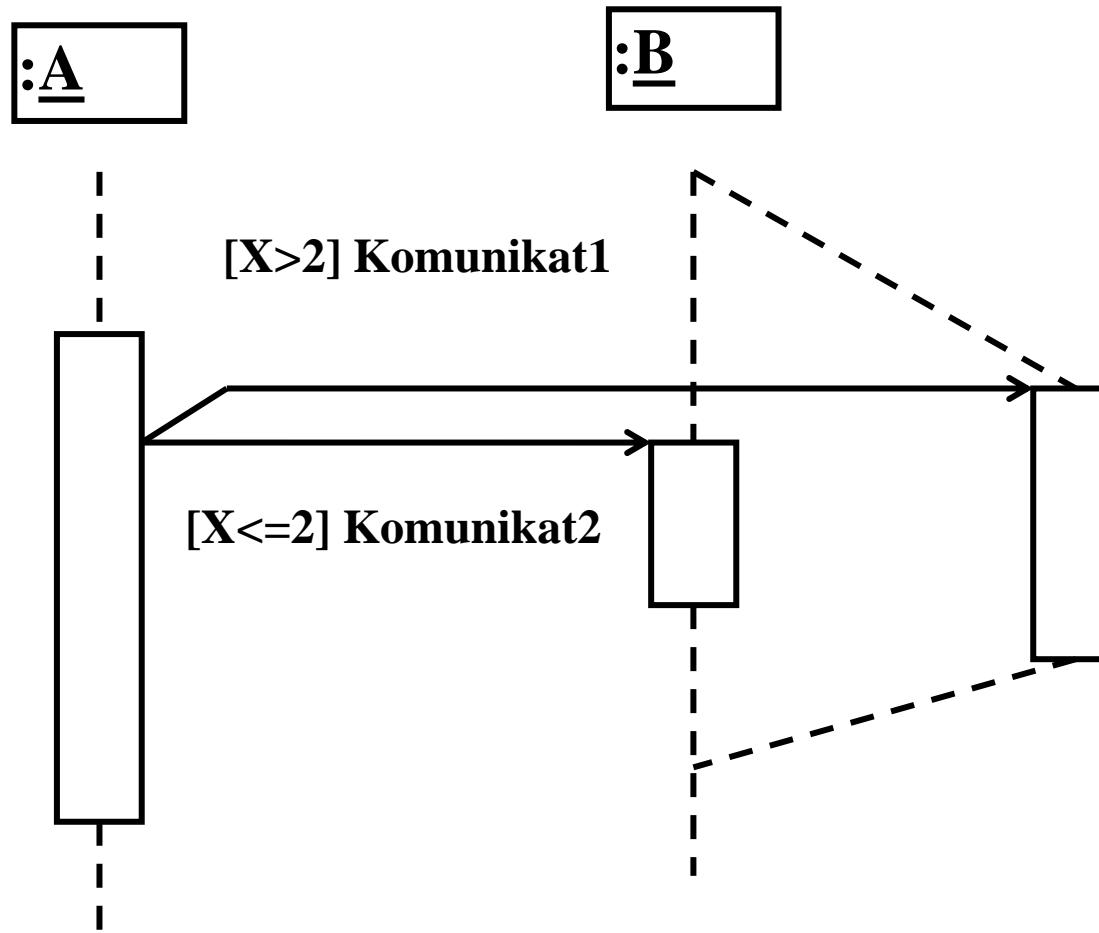


# Warunkowe wysłanie komunikatu - rozgałęzienie

- Warunki muszą być wzajemnie rozłączne



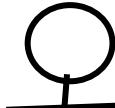
# Rozgałęzienie u odbiorcy



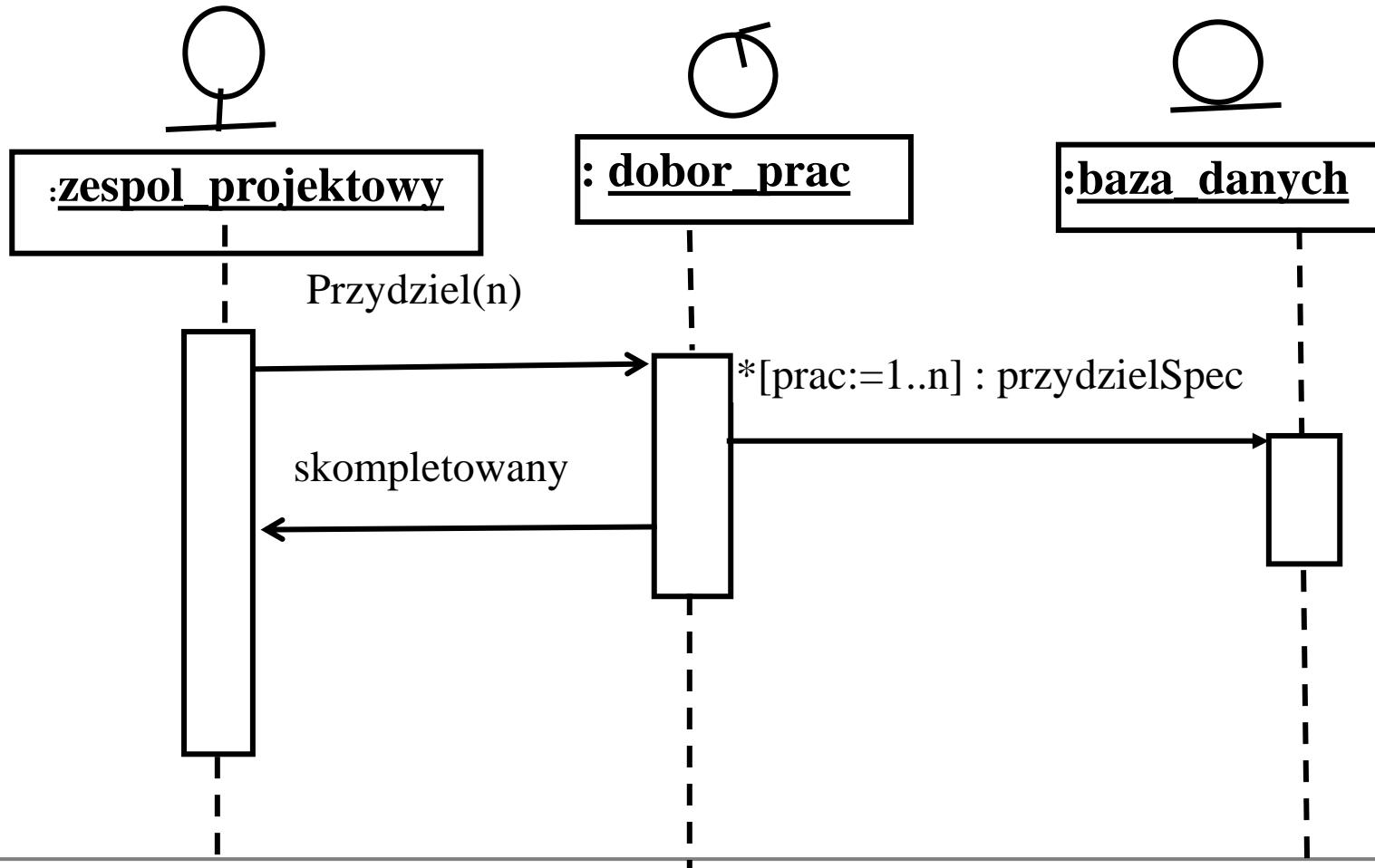
# Iteracja i oznaczenia obiektów

Ten sam komunikat wykonywany wielokrotnie

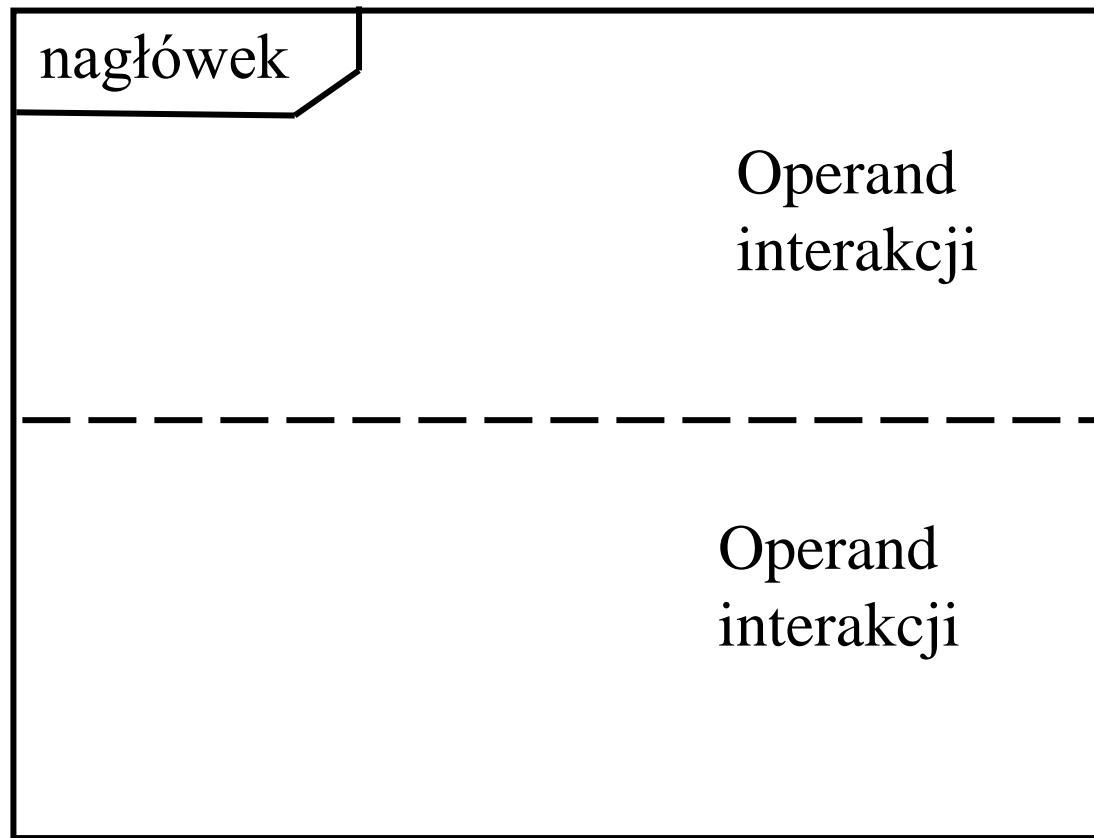
[ **<specyfikacja-iteracji>** ] operacja

- Klasa (obiekt ) sterująca 
- Klasa (obiekt ) przechowująca 
- Klasa (obiekt ) graniczna 

# przykład



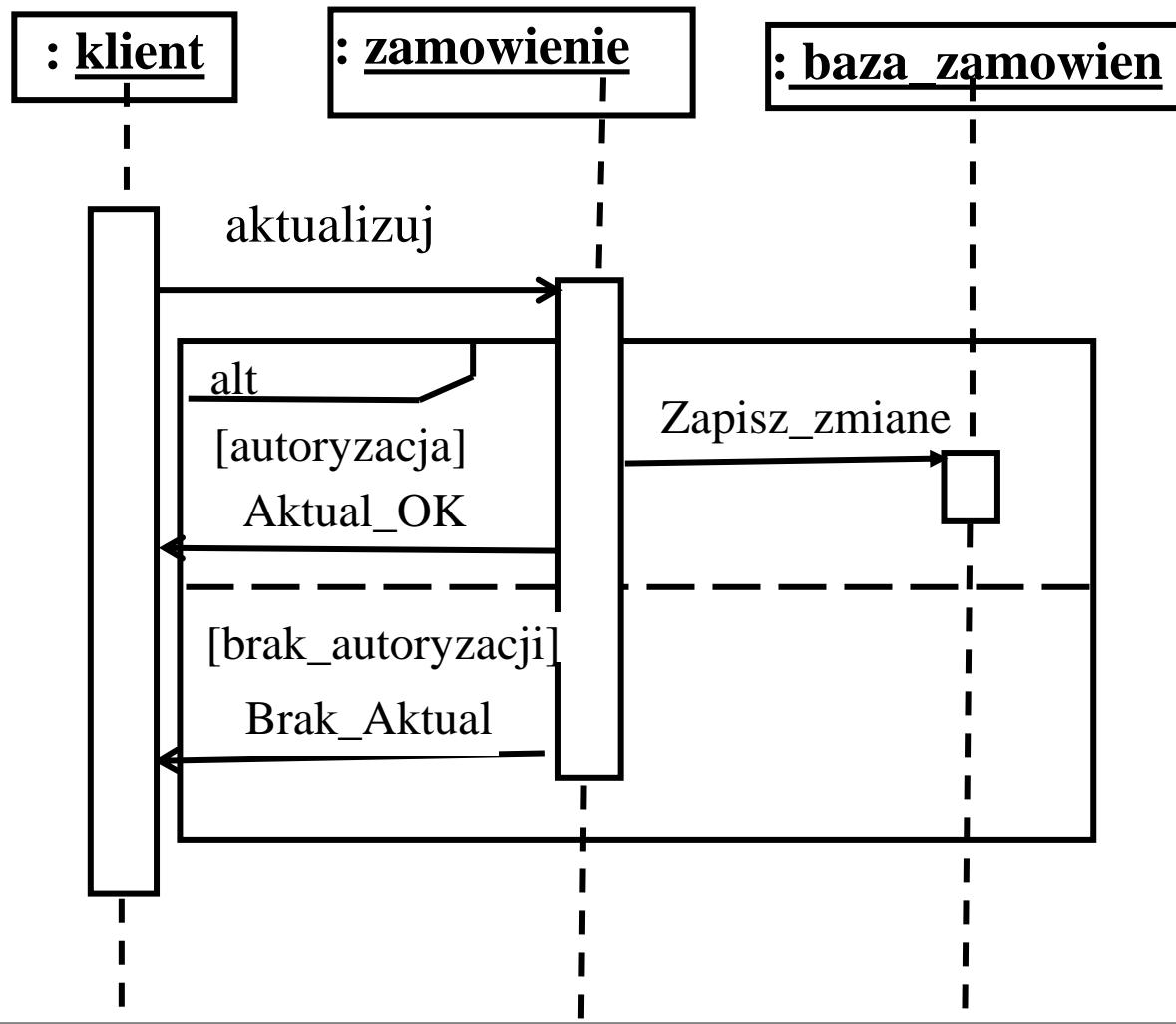
# Fragmenty wyodrębnione (combined fragments)



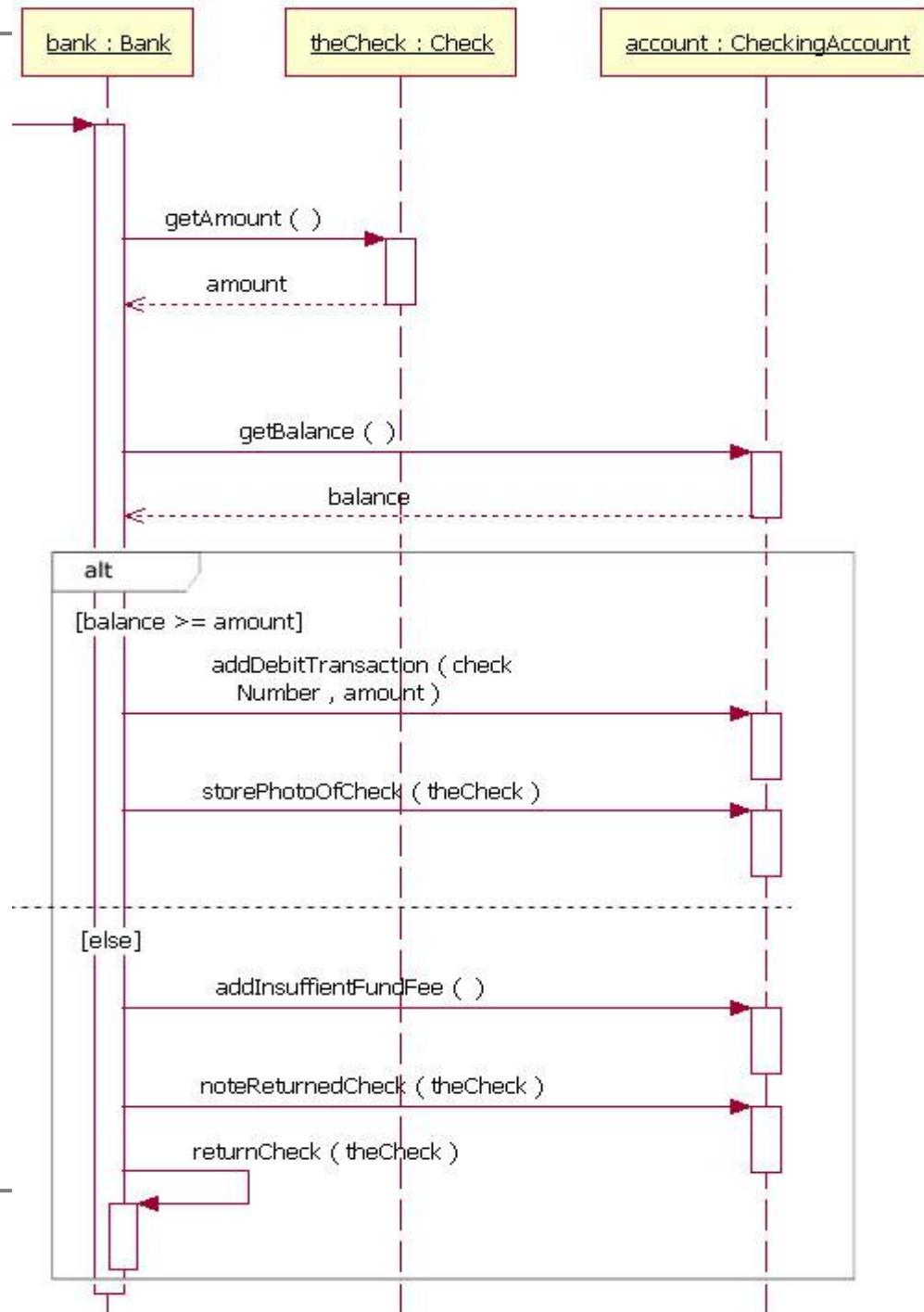
# Operatory interakcji

- **alt** – alternatywa
- **opt** – opcja
- **break** – przerwanie
- **loop** – iteracja
- **par** – współbieżność
- **neg** - funkcjonalność nieprawidłowa
- **strict** – ścisłe uporządkowanie
- **seq** – słabe uporządkowanie
- **ignore** - nieistotne
- **consider** – istotne
- **assert** – formuła
- **critical** – obszar krytyczny

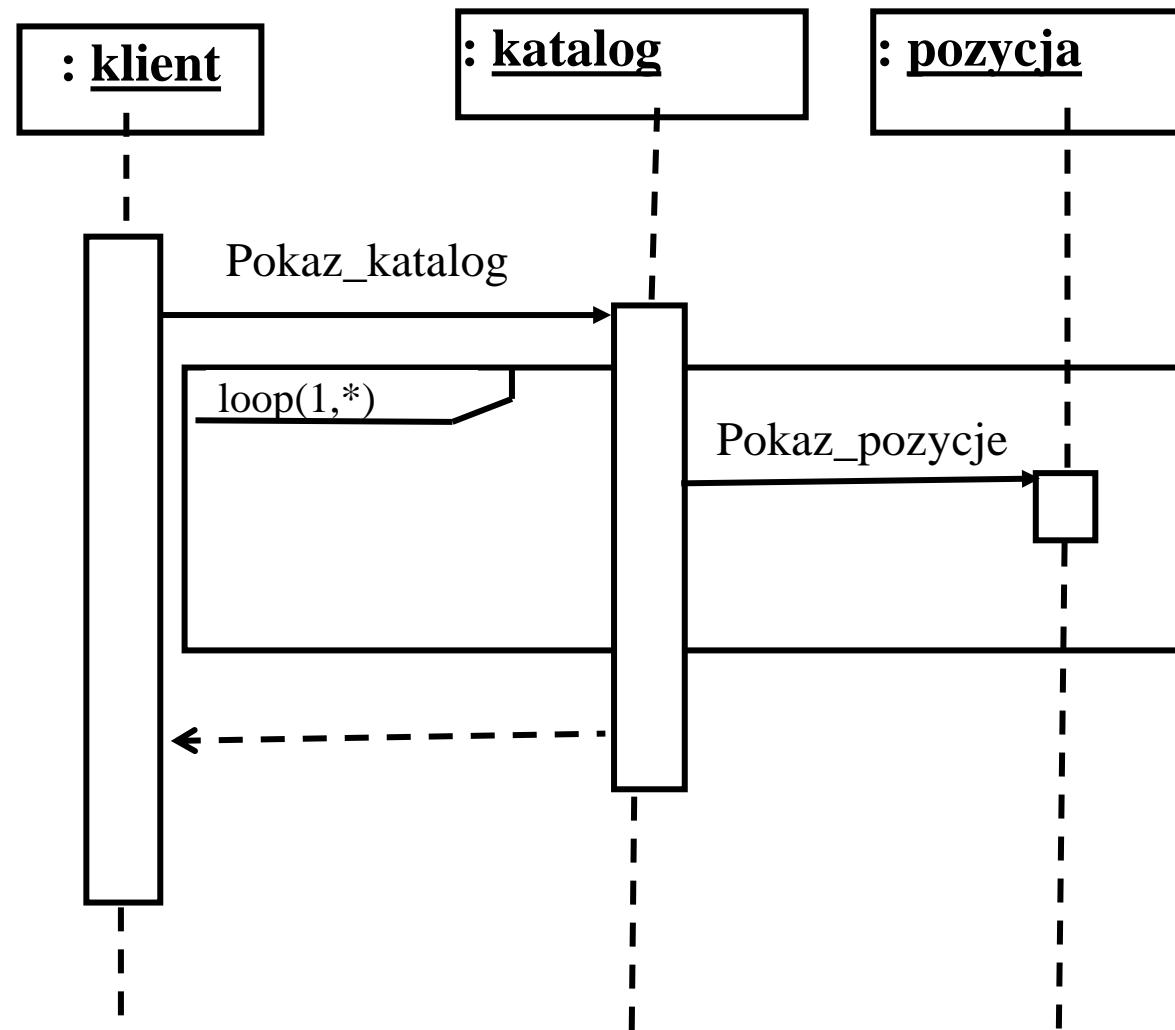
# alternatywa - alt



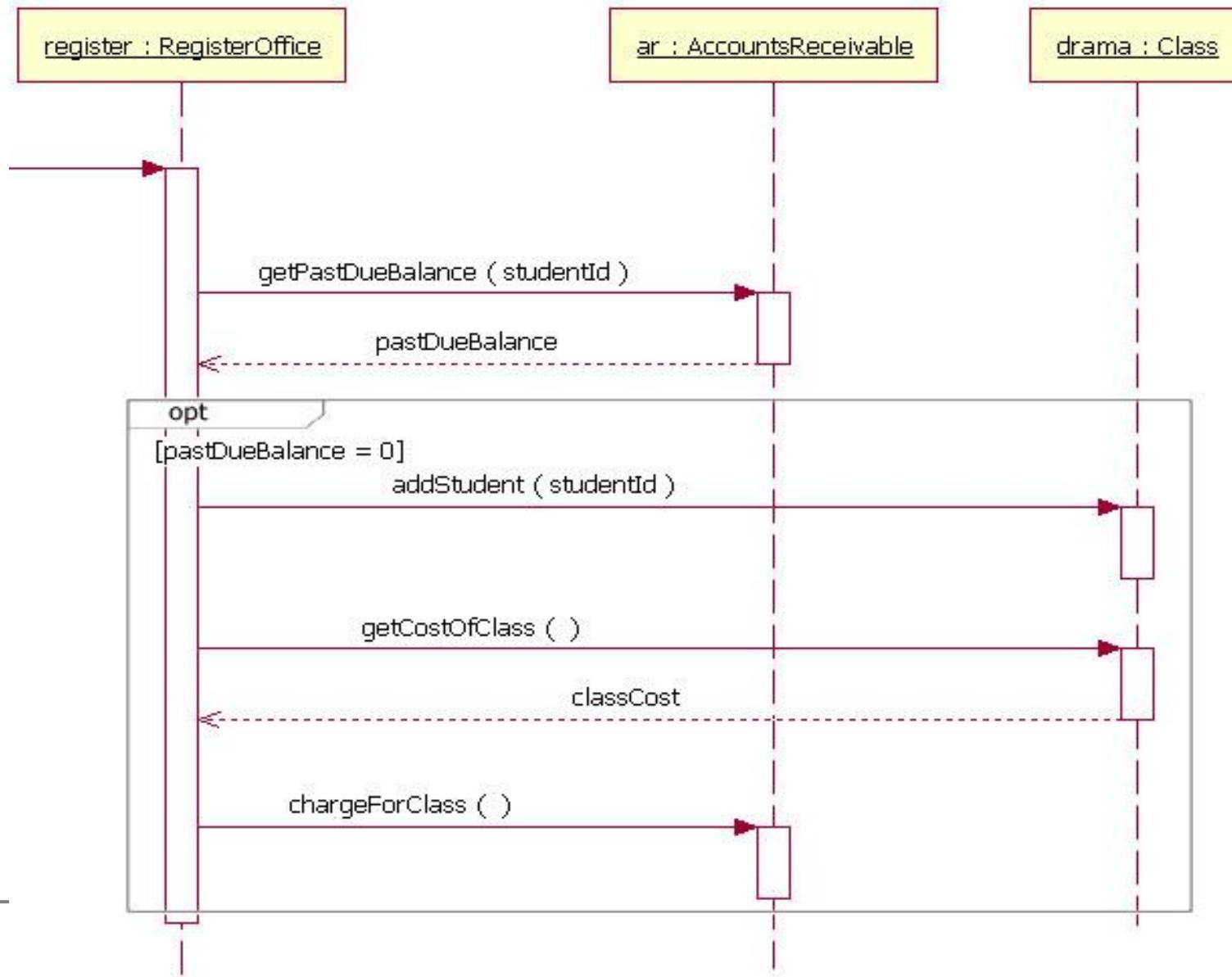
# Alternatywa



# Iteracja – loop



# Opcja

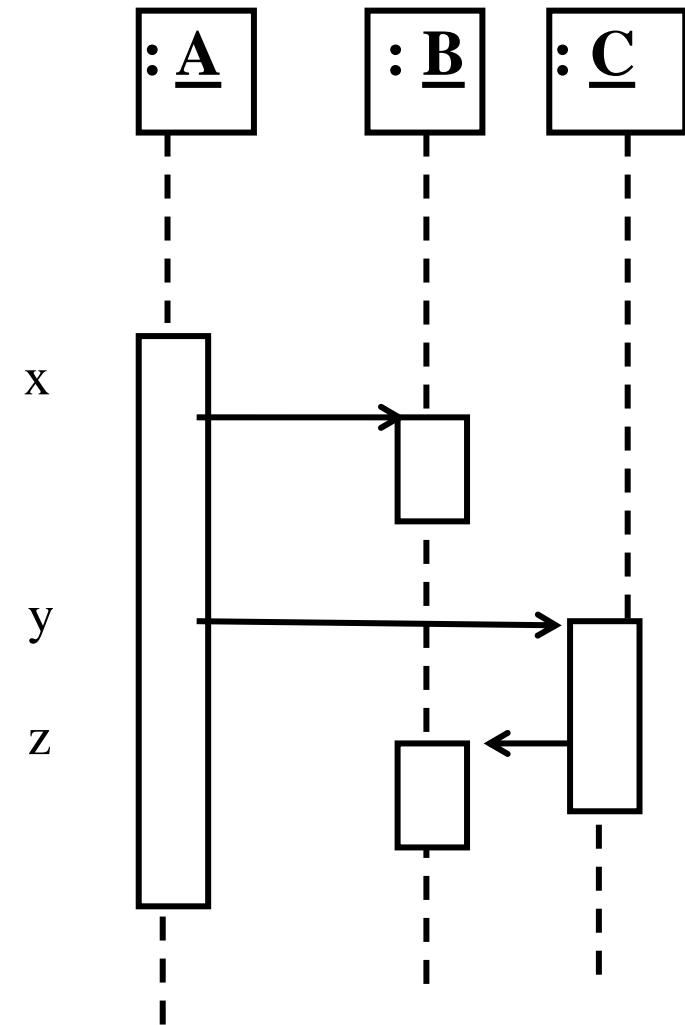


# ograniczenia czasowe

- Diagramy sekwencji mogą być uzupełnione informacjami tekstowymi (ograniczeniami czasowymi), w postaci tekstu swobodnego lub pseudokodu, umieszczonego w pobliżu punktu startowego komunikatu.

$\{y-x < 3s\}$

$\{z-y < 1s\}$



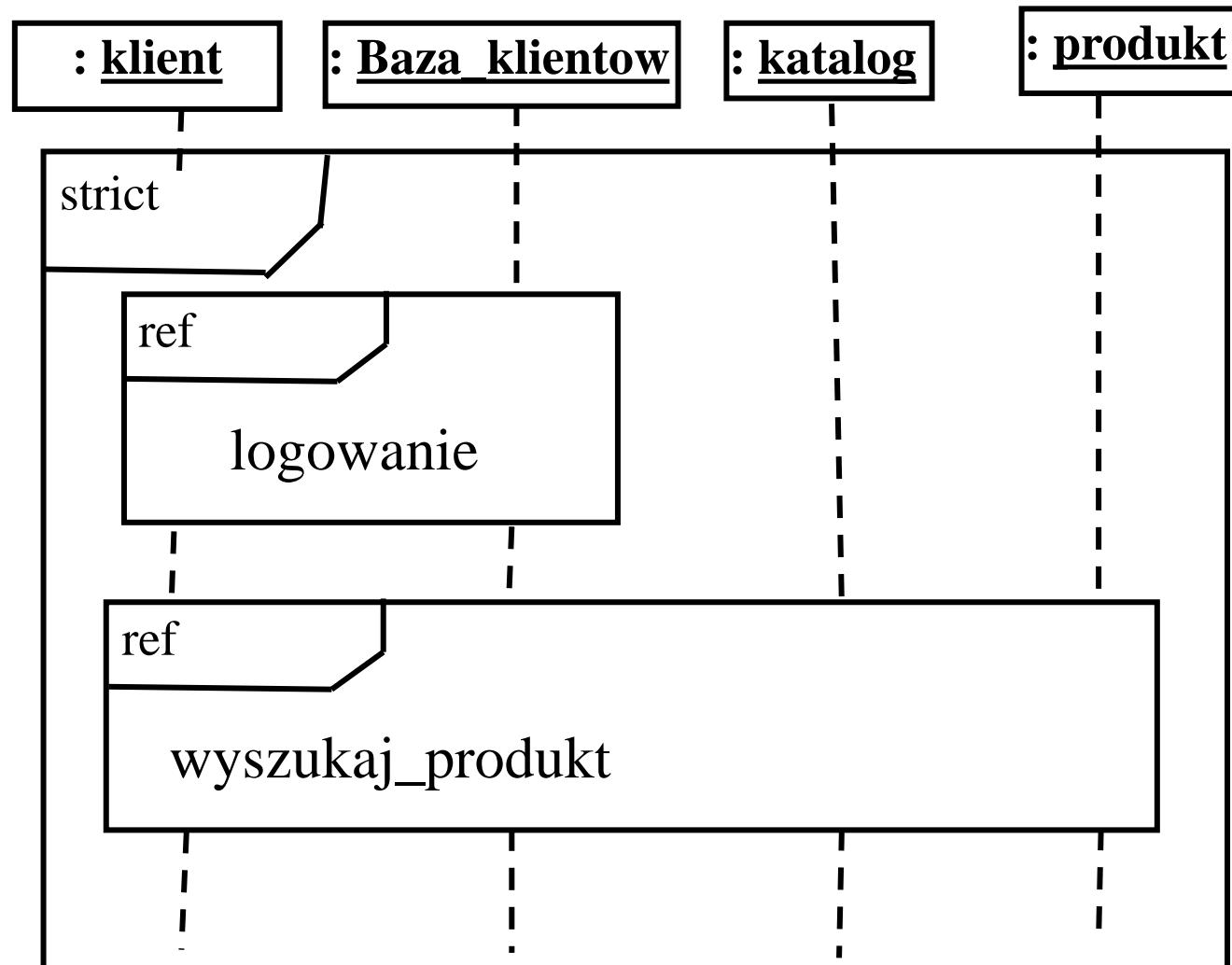
# Przywoływanie wystąpienie interakcji – **ref**

(interaction occurrences) - odwołanie na diagramie bazowym do innego diagramu sekwencji – operator **ref**.

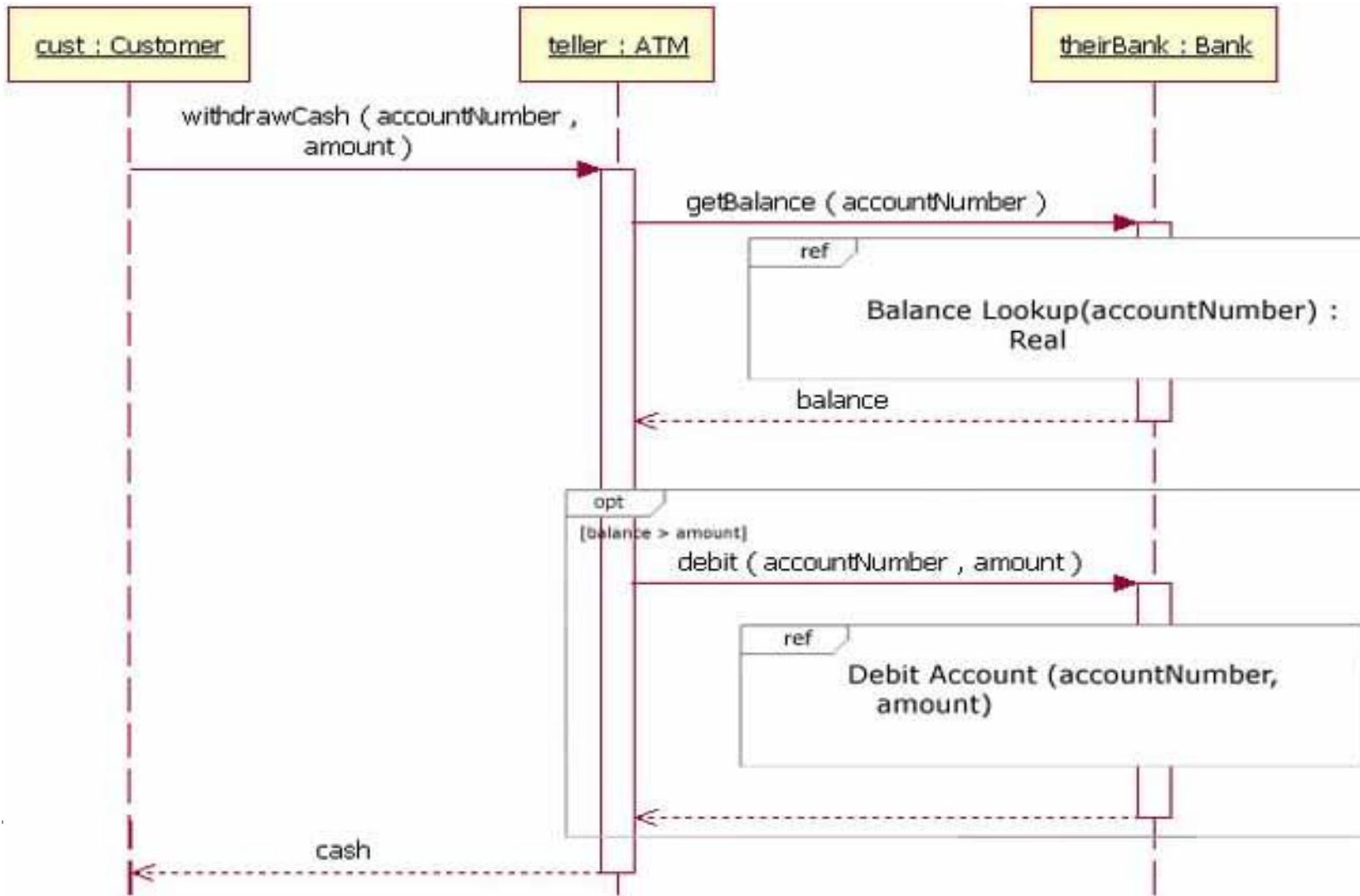
Zainicjowanie wystąpienia interakcji poprzez:

- komunikat
- czynnik czasu

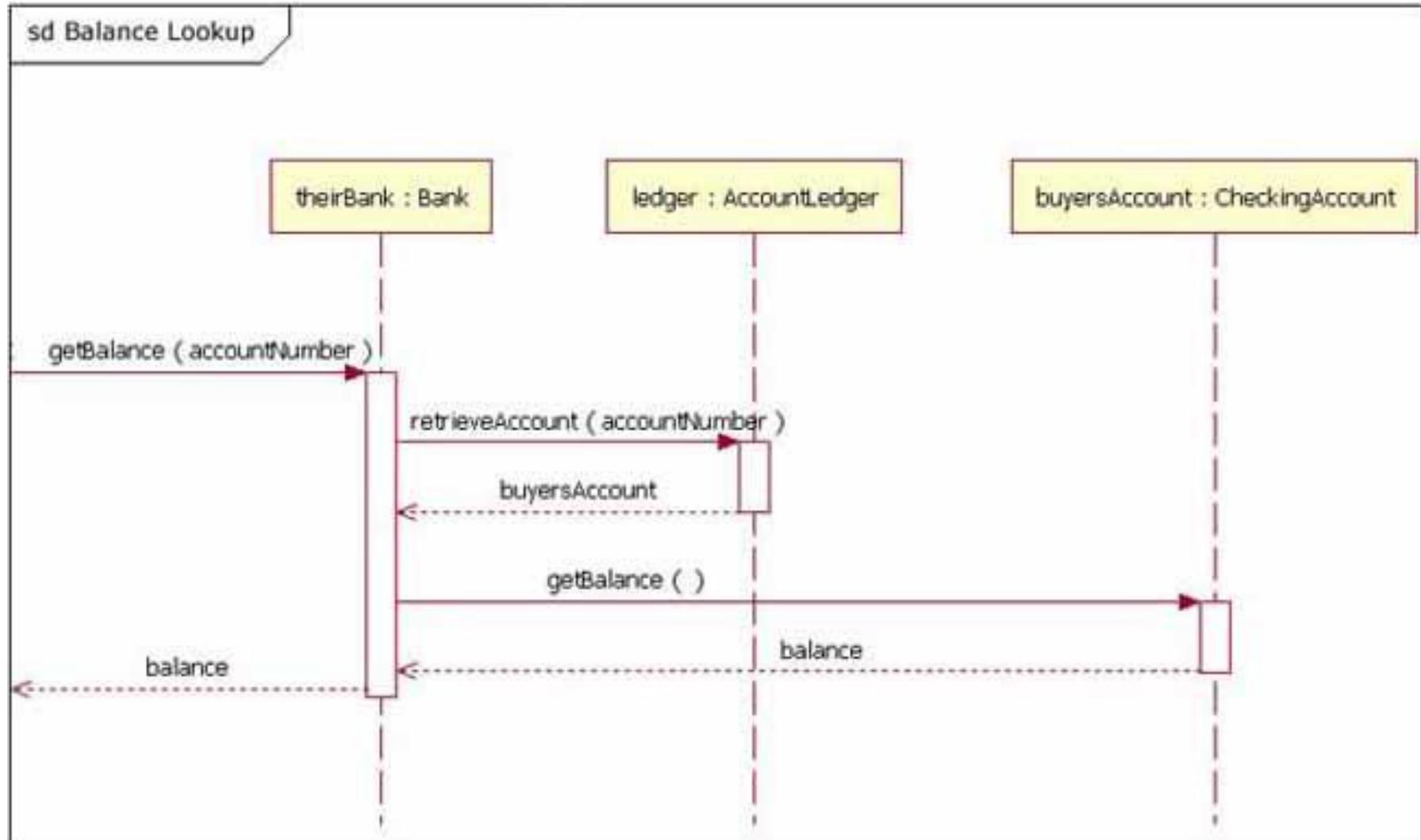
# Przykład 1 (czynnik czasu)



# Wywołanie diagramu - referencja

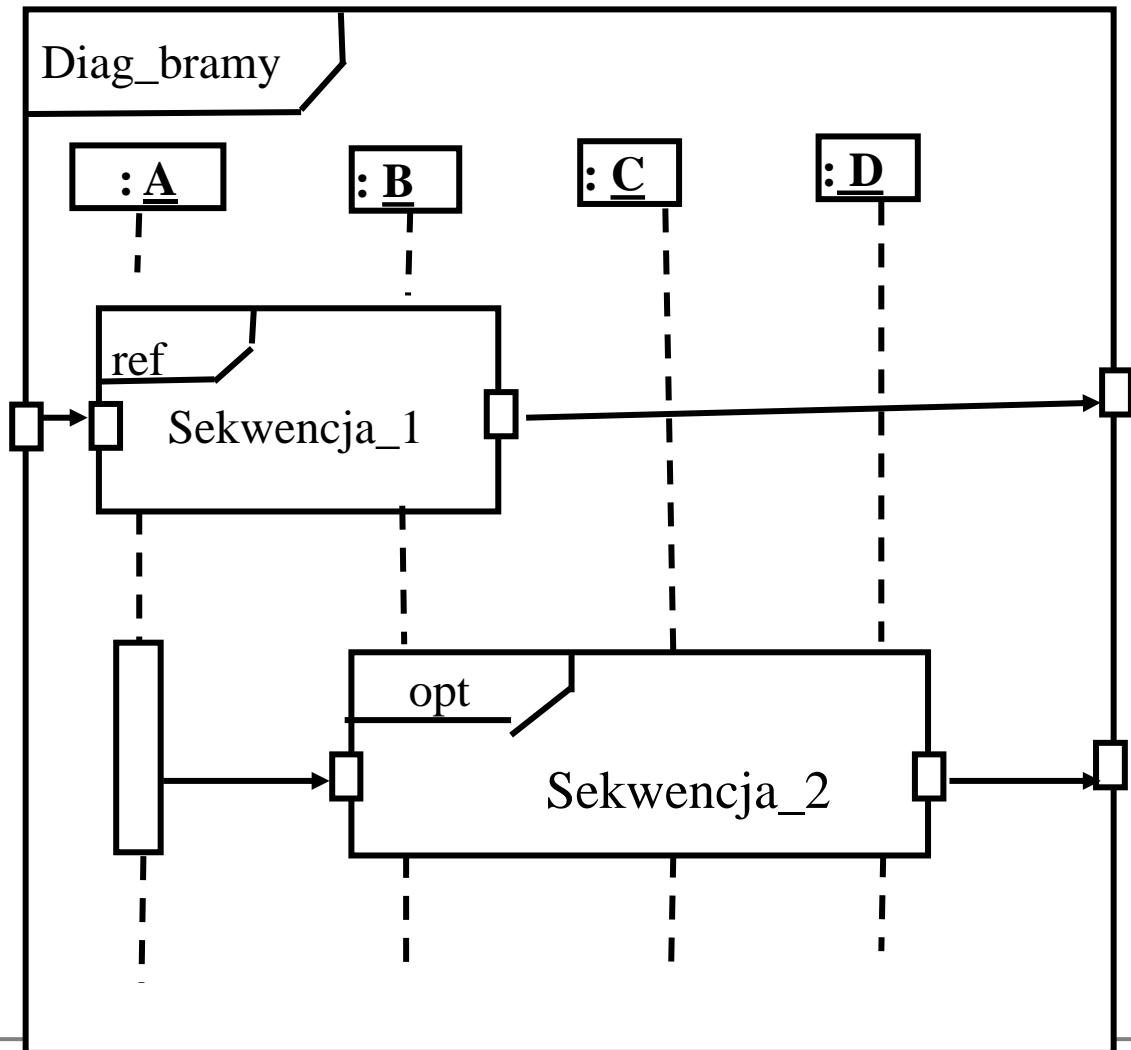


# Bramki (gates) wejściowe i wyjściowe



# Bramy (gates)

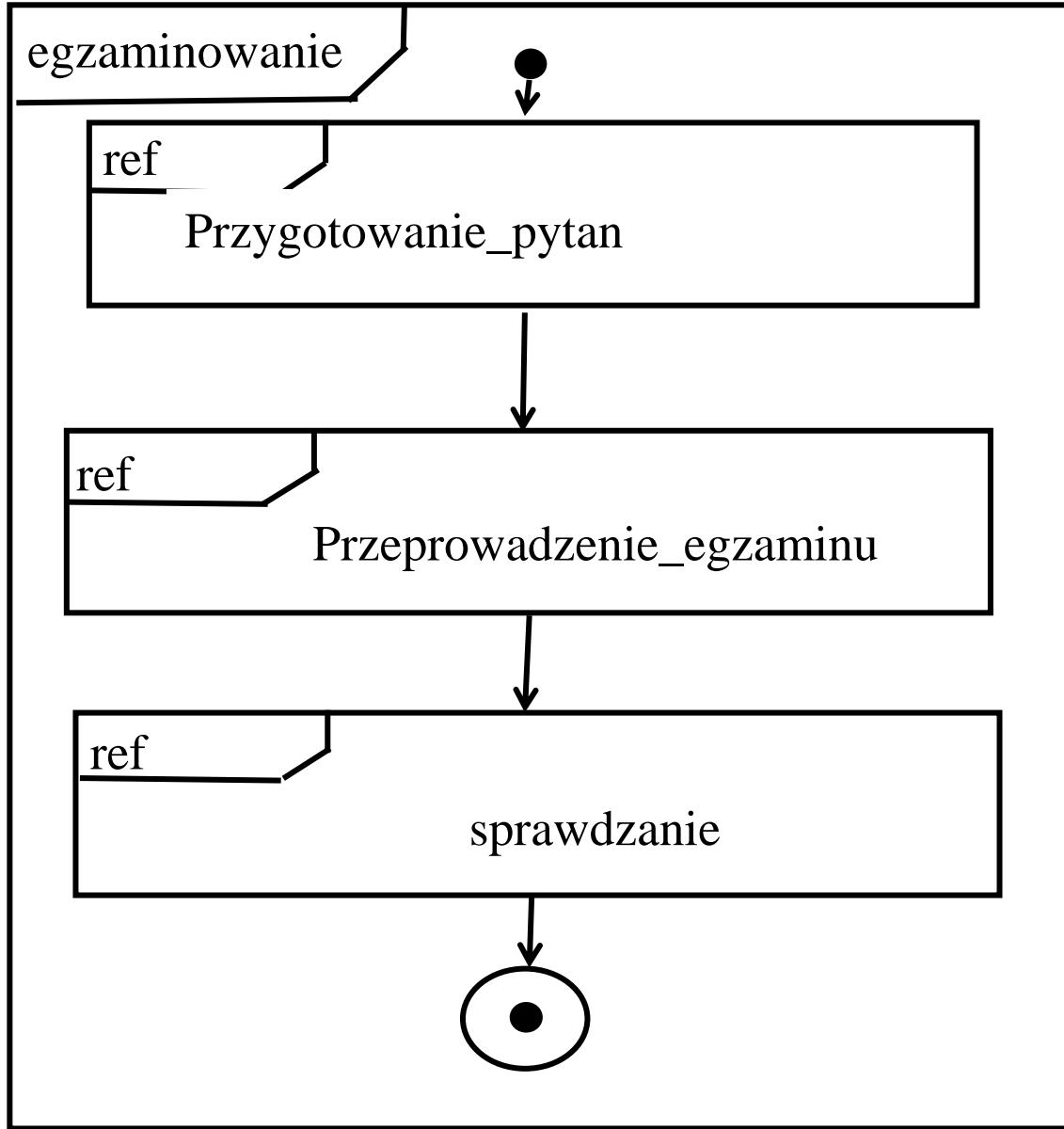
- punkty przejścia komunikatów z/do diagramu sekwencji, przywoływanych interakcji, sekwencji wyodrębnionych



# Diagramy sterowania interakcją

(interaction overviews)

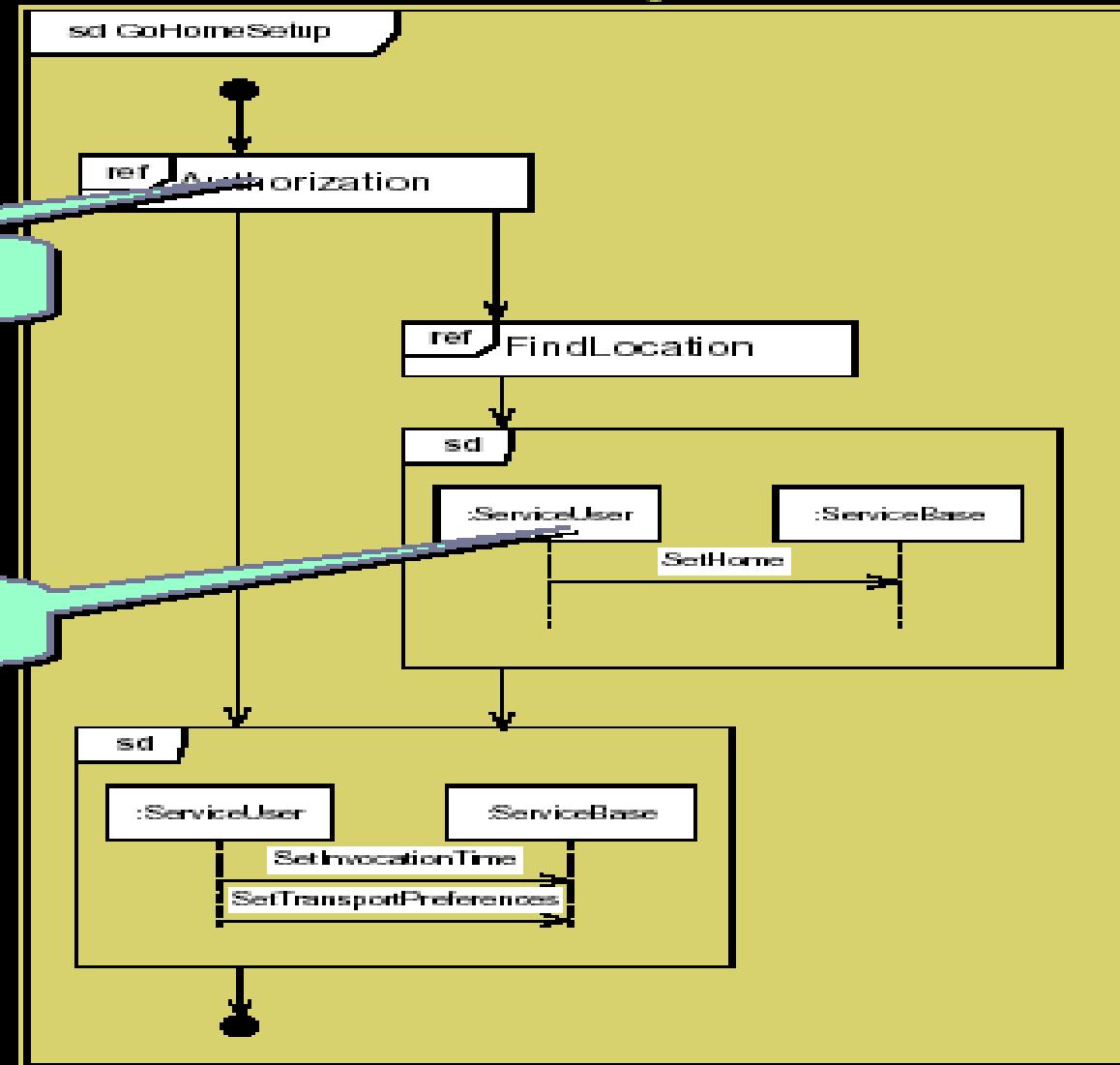
- Dokumentują przepływ sterowania pomiędzy logicznie powiązanymi diagramami sekwencji, fragmentami interakcji wykorzystując operatory modelowania z diagramów czynności.



# Diagram widoku interakcji

Interaction Occurrence

Expanded sequence diagram



Przykład\_decyzj

i



ref

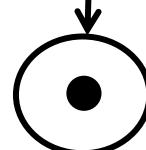
Wyszukaj produkt

[nie znaleziono]

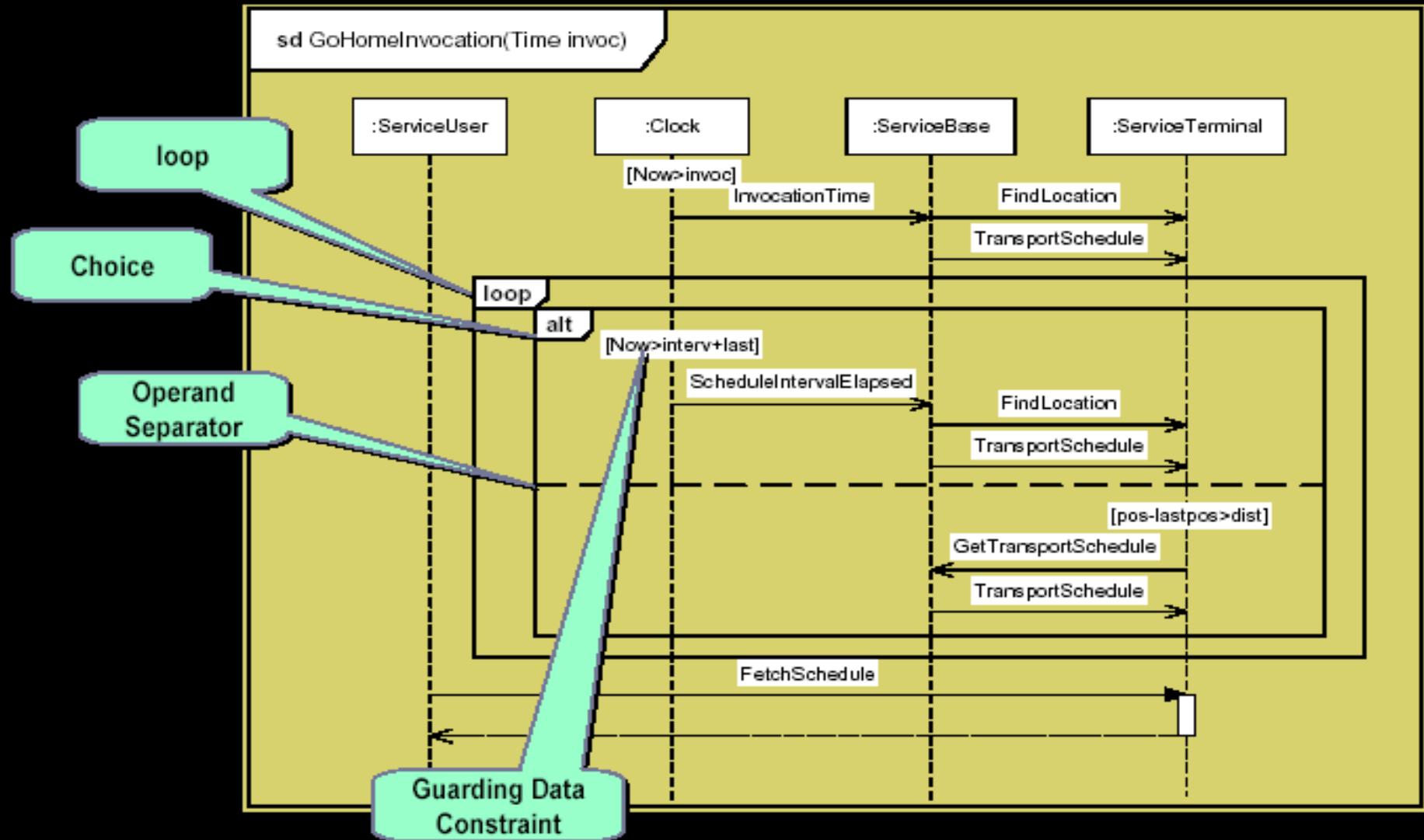
[znaleziono]

ref

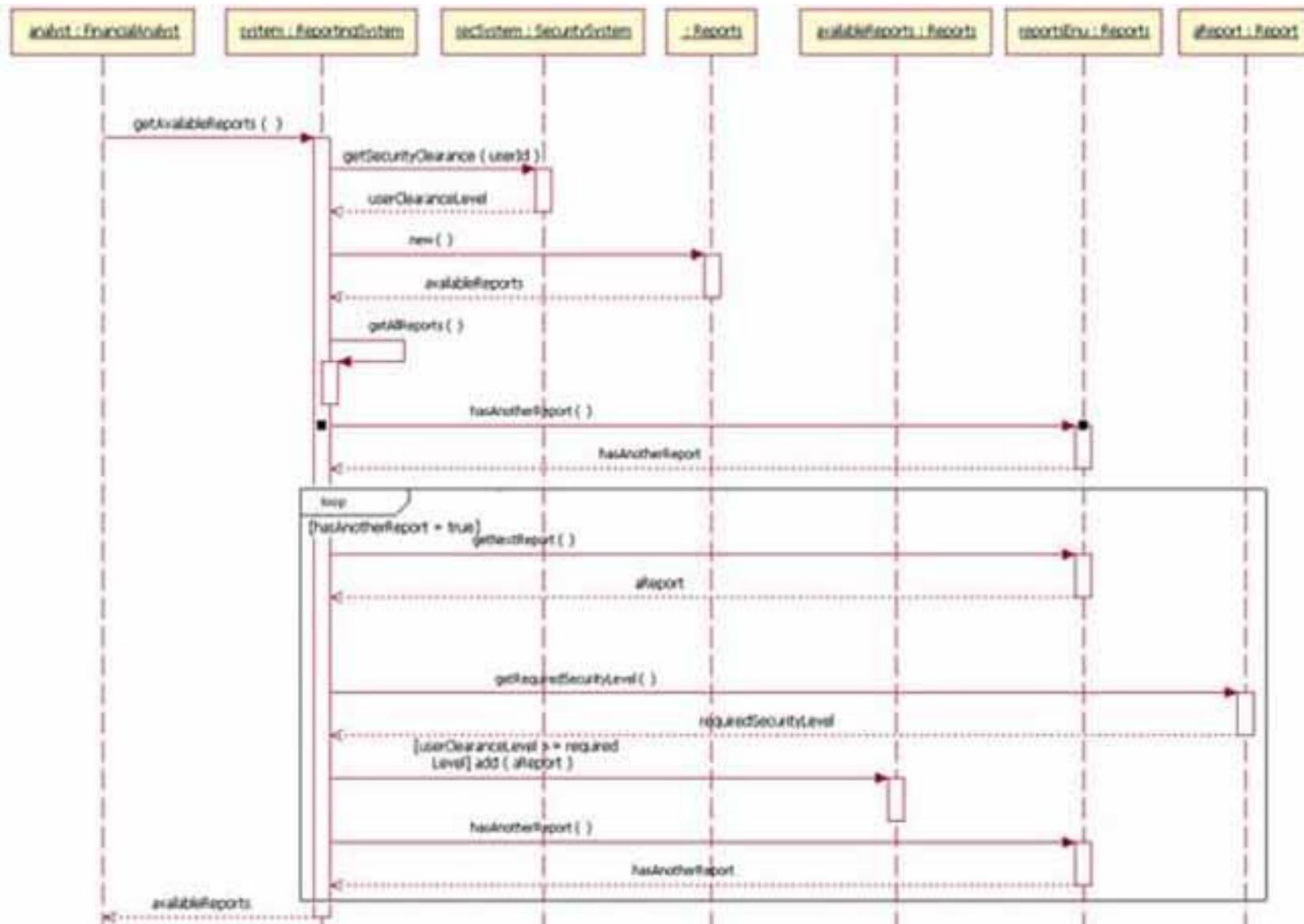
rezerwuj



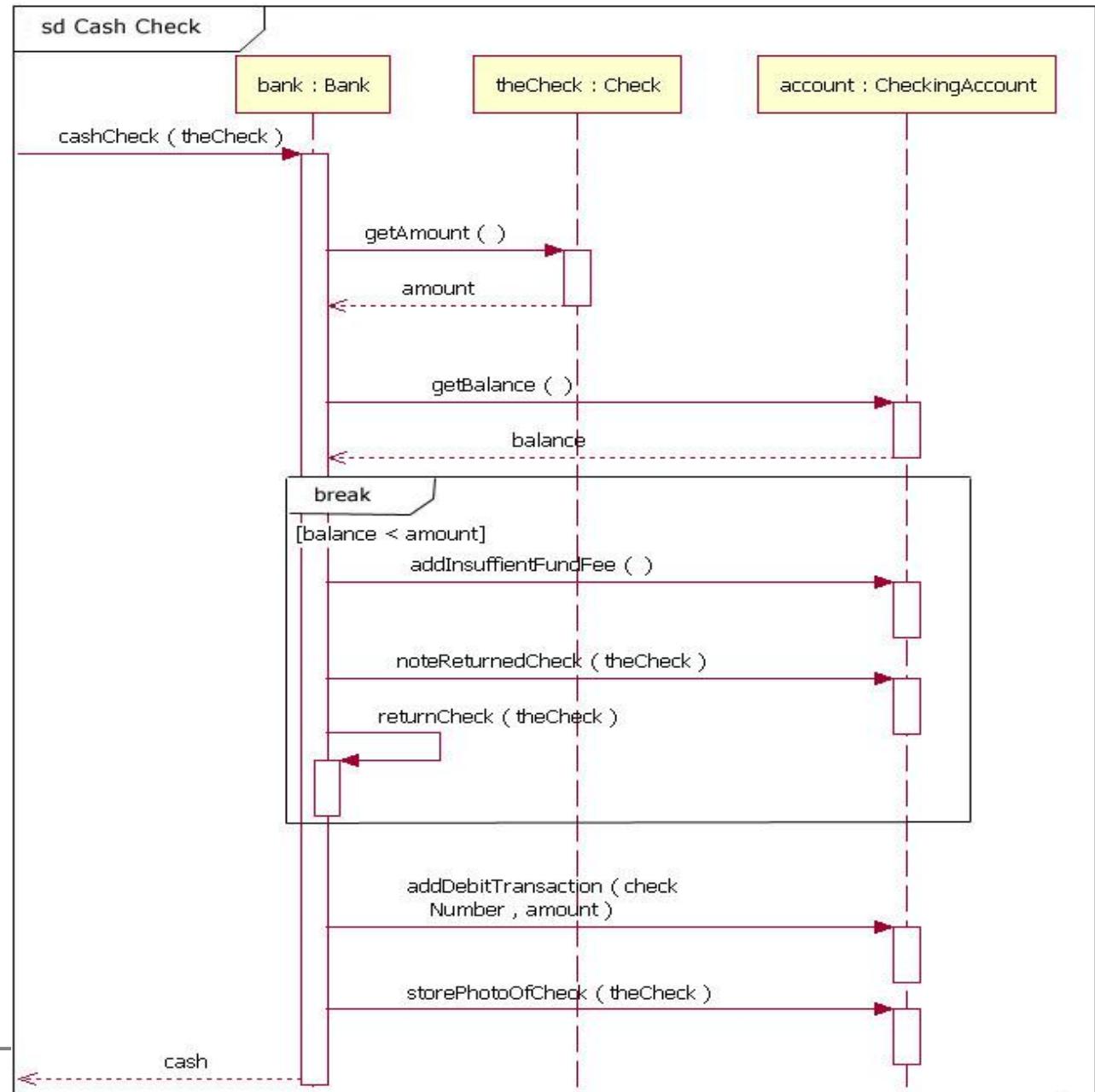
# Pętla i alternatywa



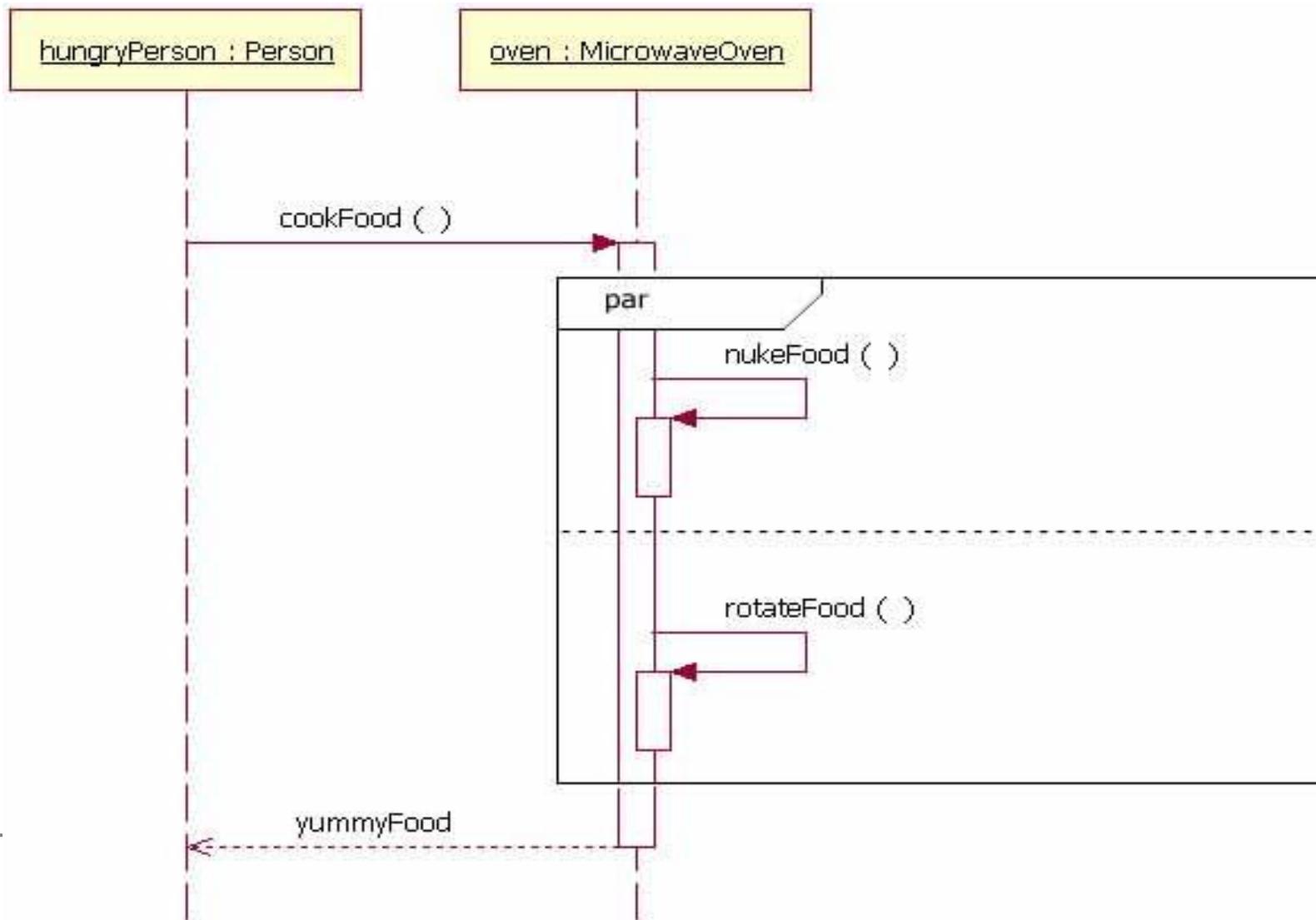
# Pętla (loop)



# Break



# Równoległe wykonanie



# Diagramy czynności (activity diagrams)

---

Dr hab. inż. Ilona Bluemke

# Diagramy czynności

Opisują dynamikę systemu. Stosowane są w modelowaniu:

- Procesów biznesowych
- Scenariuszy przypadków użycia
- Systemów, podsystemów
- Procesów systemowych z dużą liczbą równoległych czynności i decyzji
- Operacji
- Algorytmów

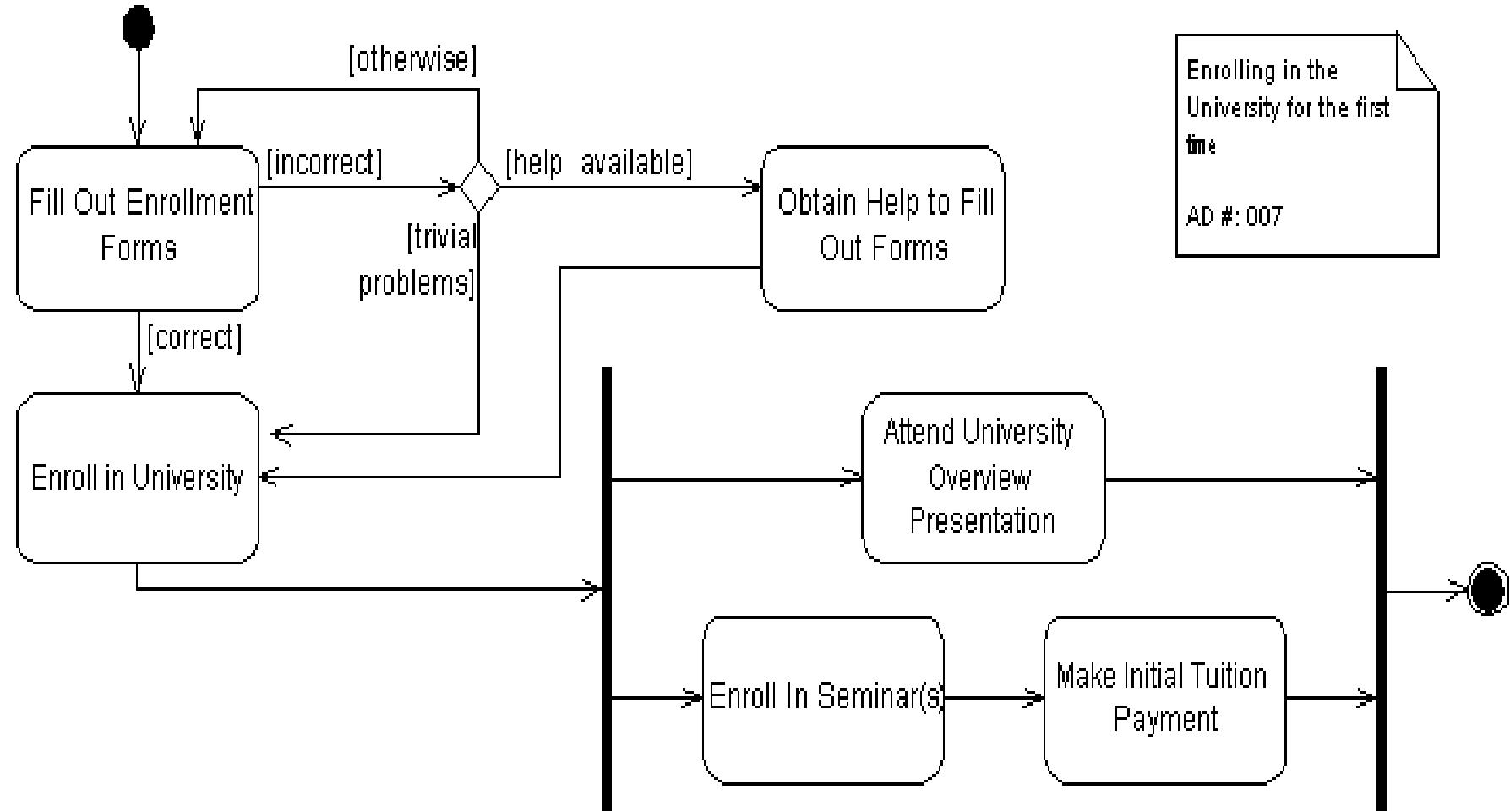
# czynność

- Czynność może być prostą operacją, ale także złożoną funkcjonalnością wymagającą dekompozycji za pomocą odrębnego diagramu czynności.

Generuj\_raport

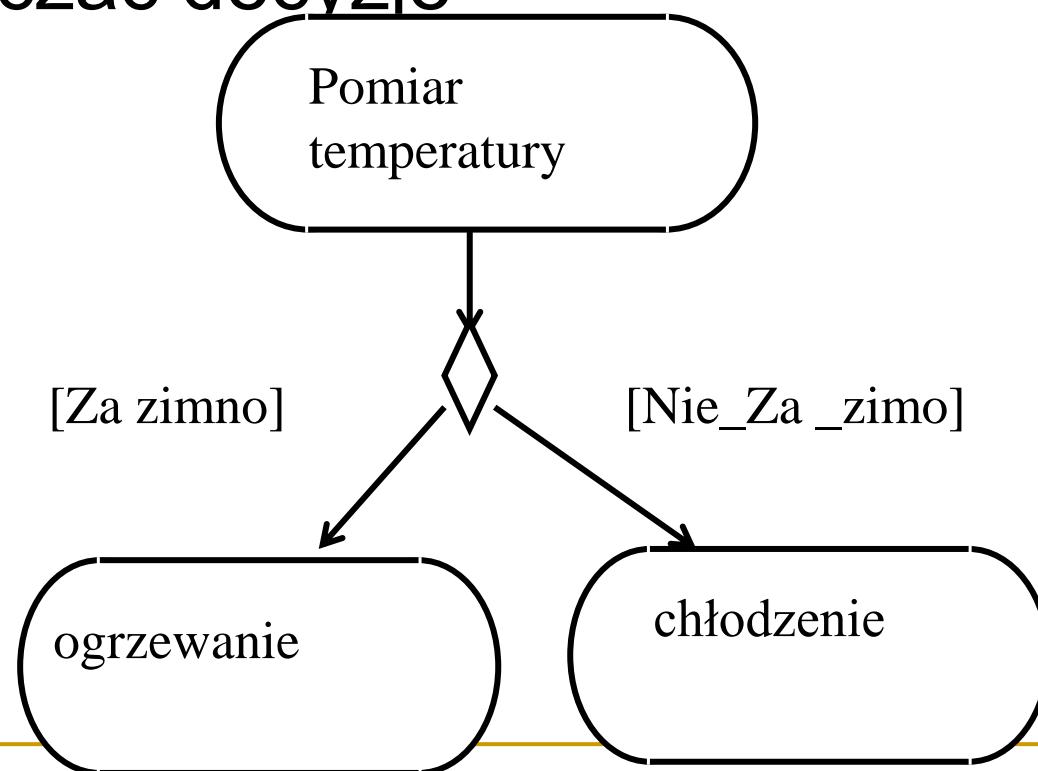


# Zapis na uniwersytet

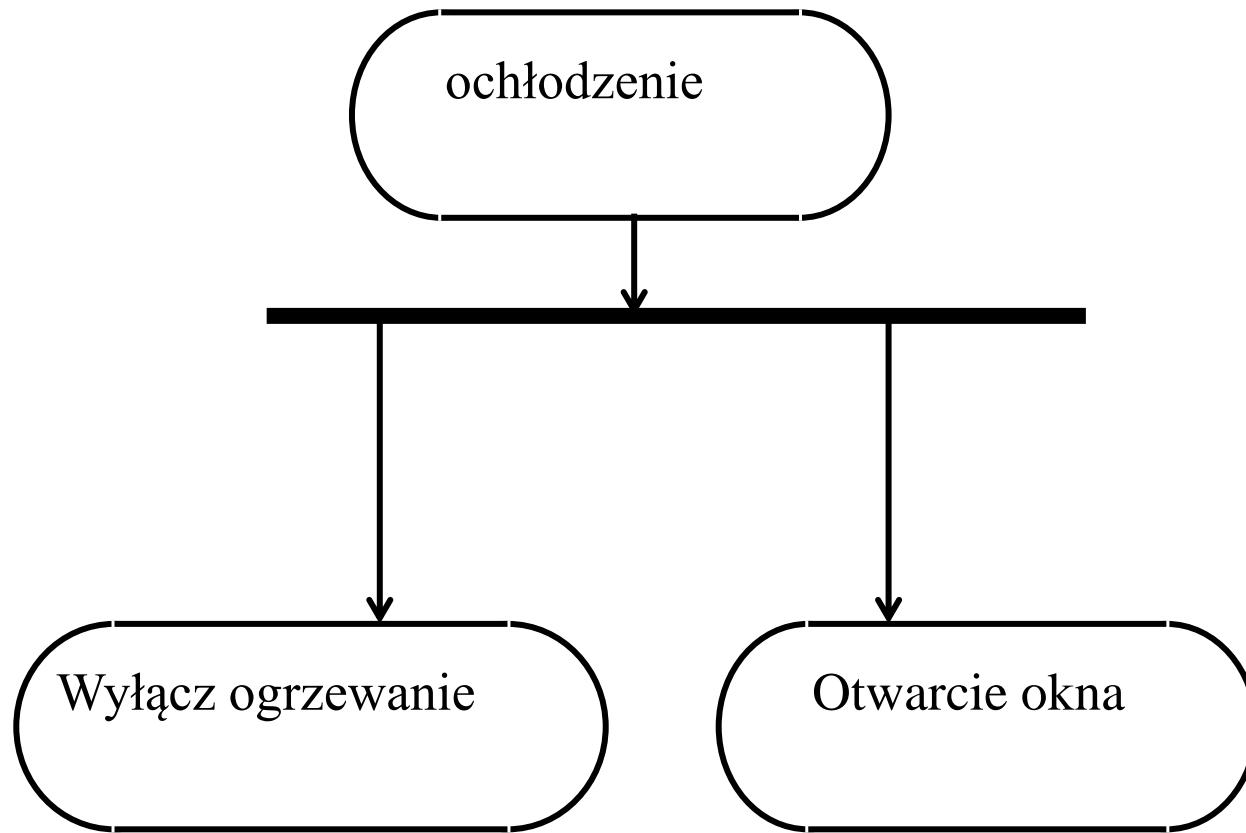


# Rozejścia warunkowe

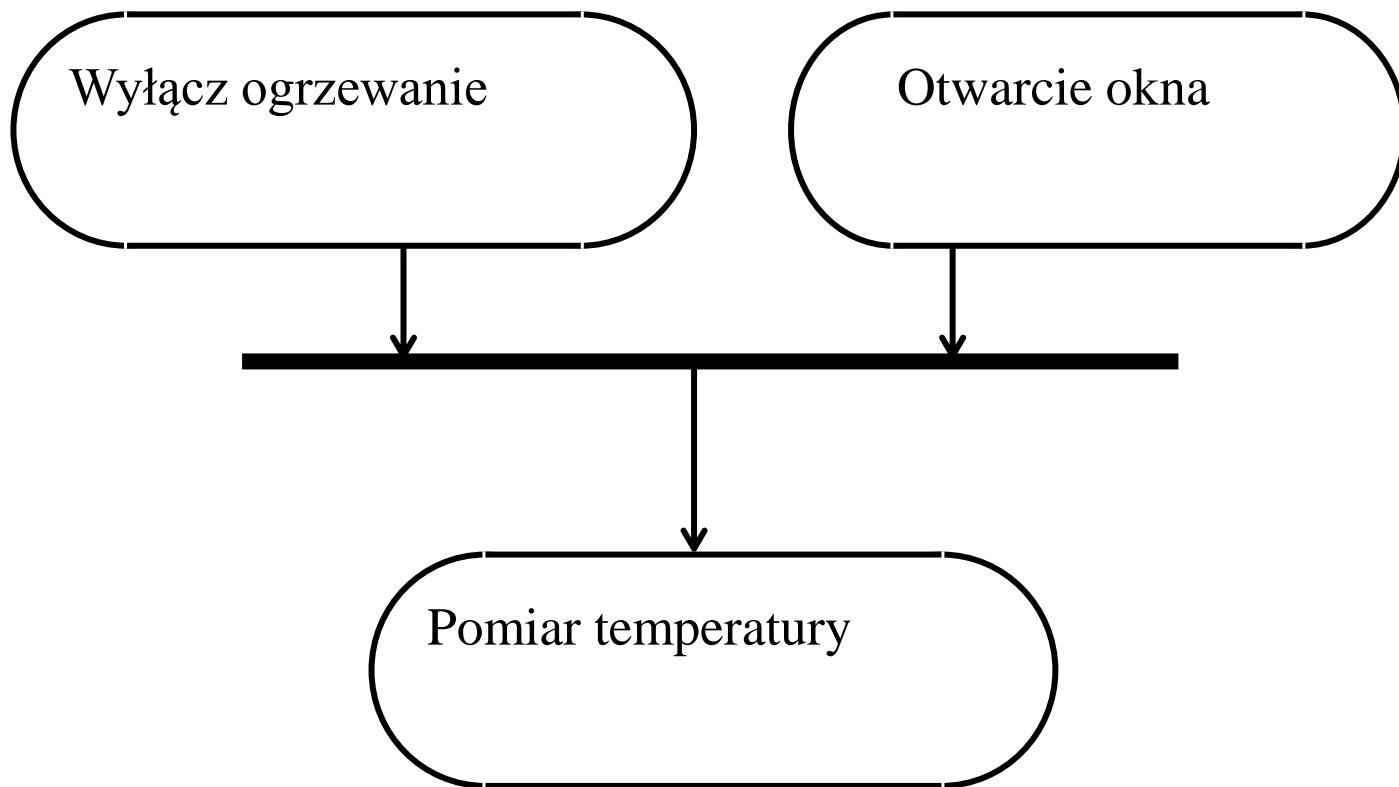
- Przejścia pomiędzy czynnościami mogą być warunkowe. Na diagramach aktywności można zaznaczać decyzje

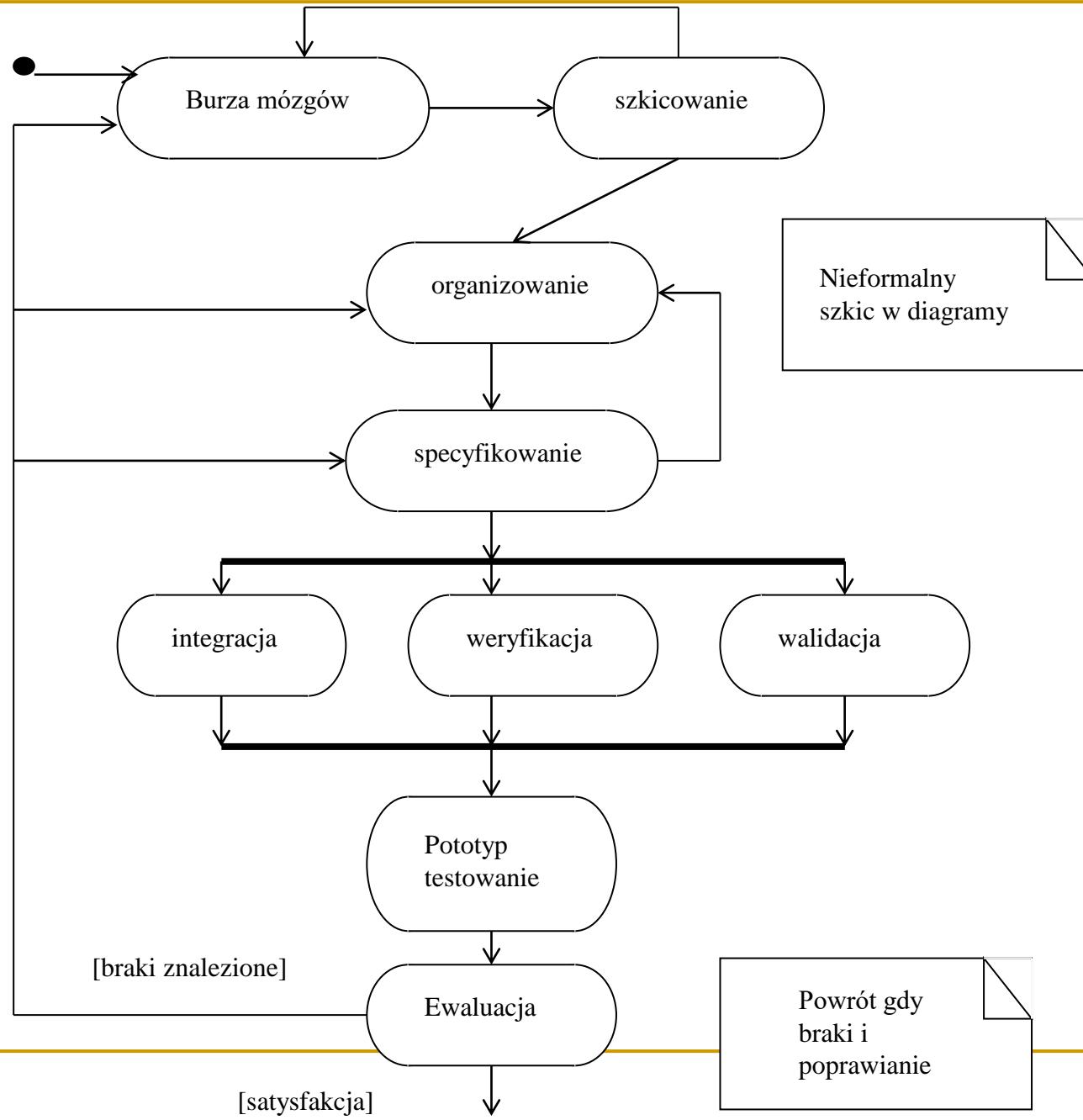


# rozpoczynanie równoległych czynności



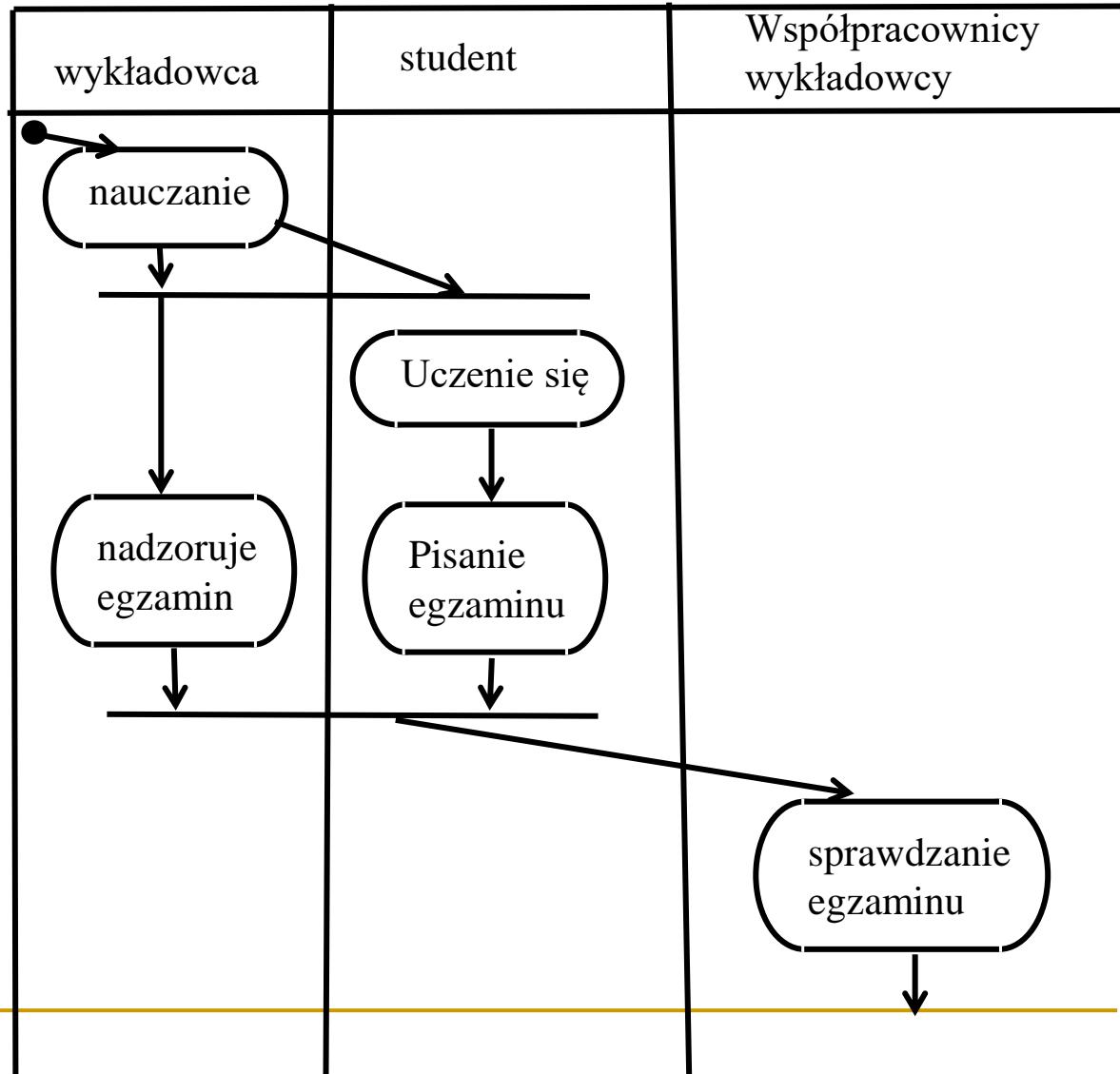
# Kończenie czynności równoległych



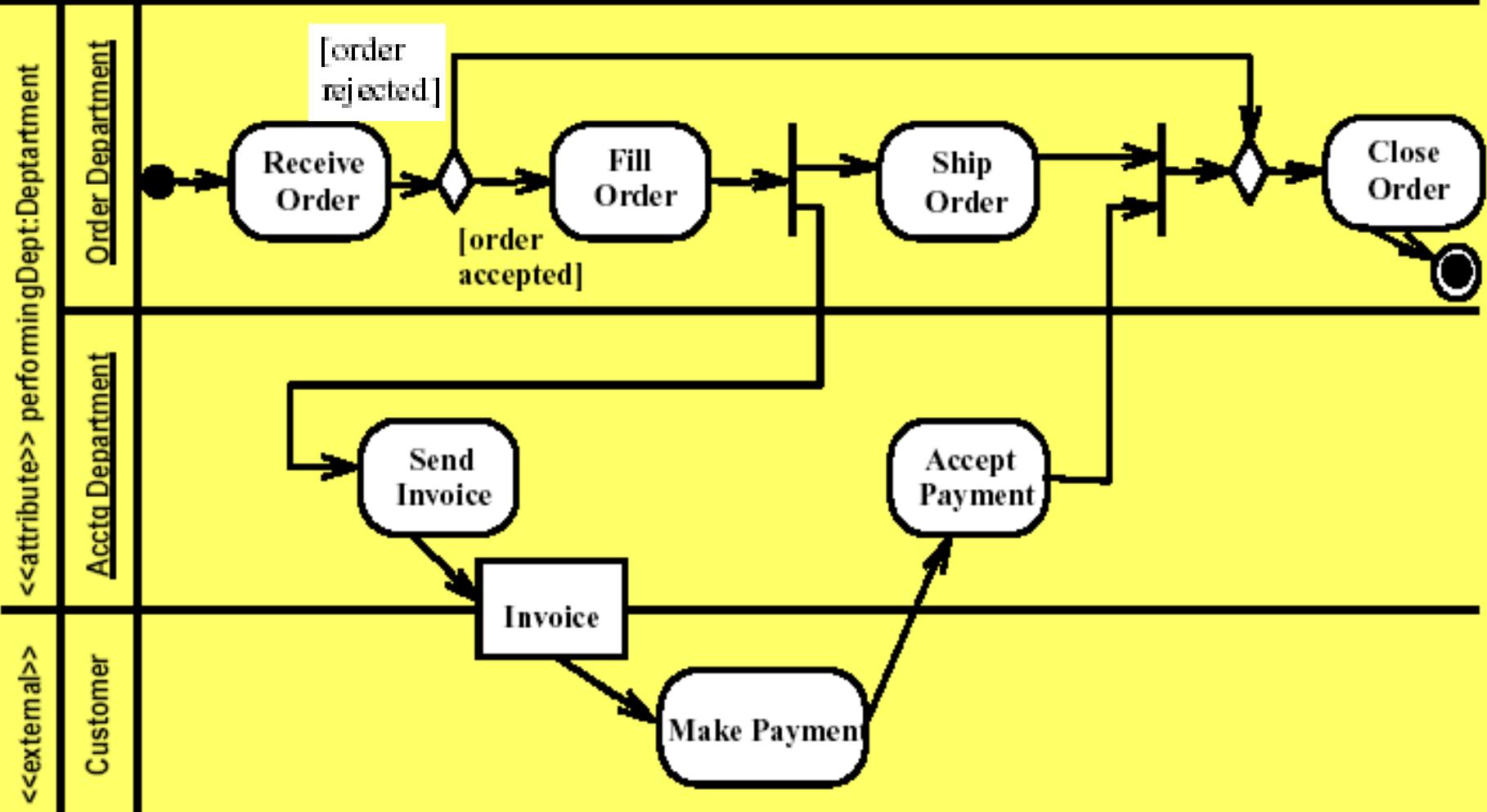


# Partycje diagramów czynności

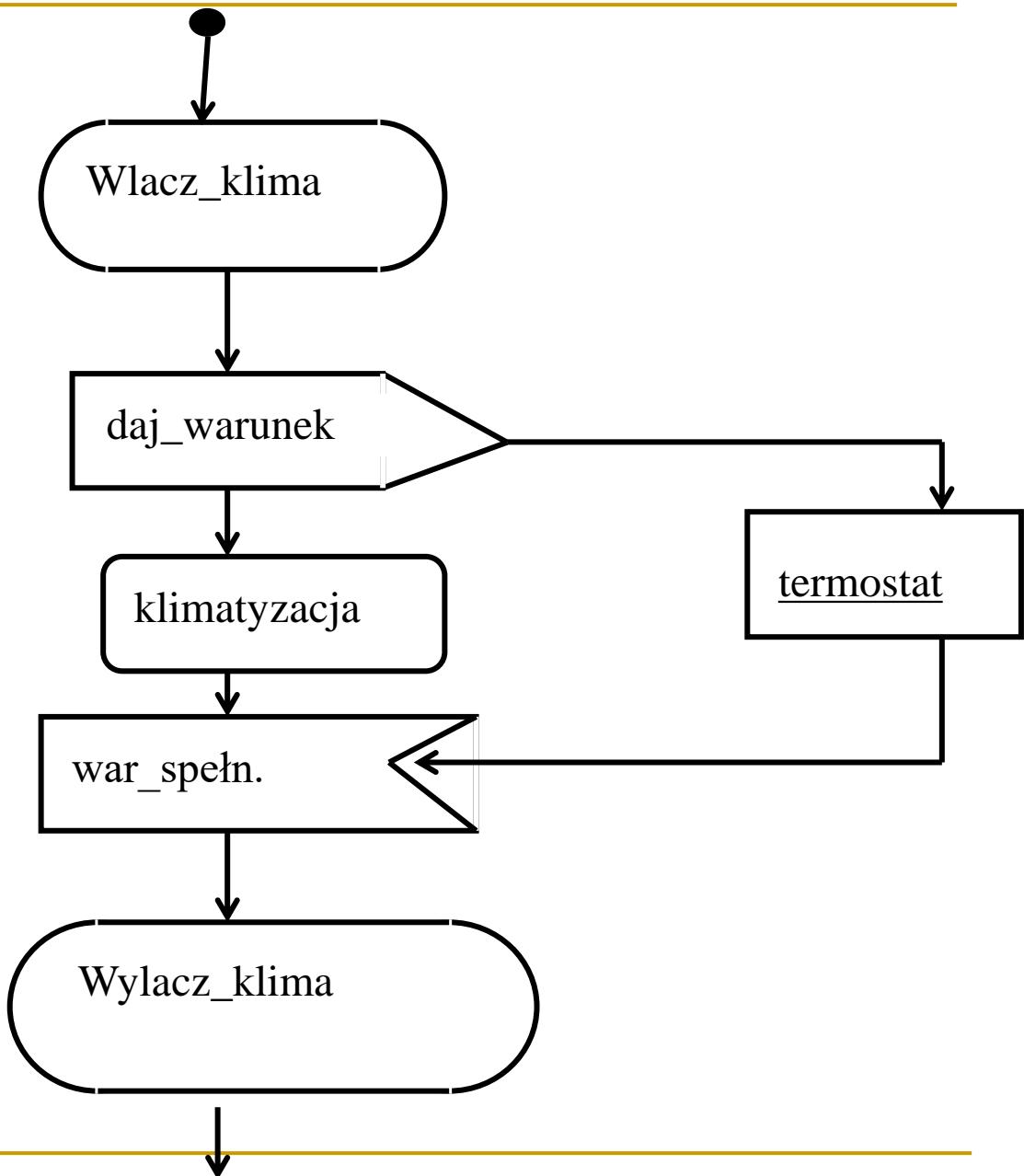
- Partycja pokazuje klasę odpowiedzialną za wykonanie poszczególnych czynności.



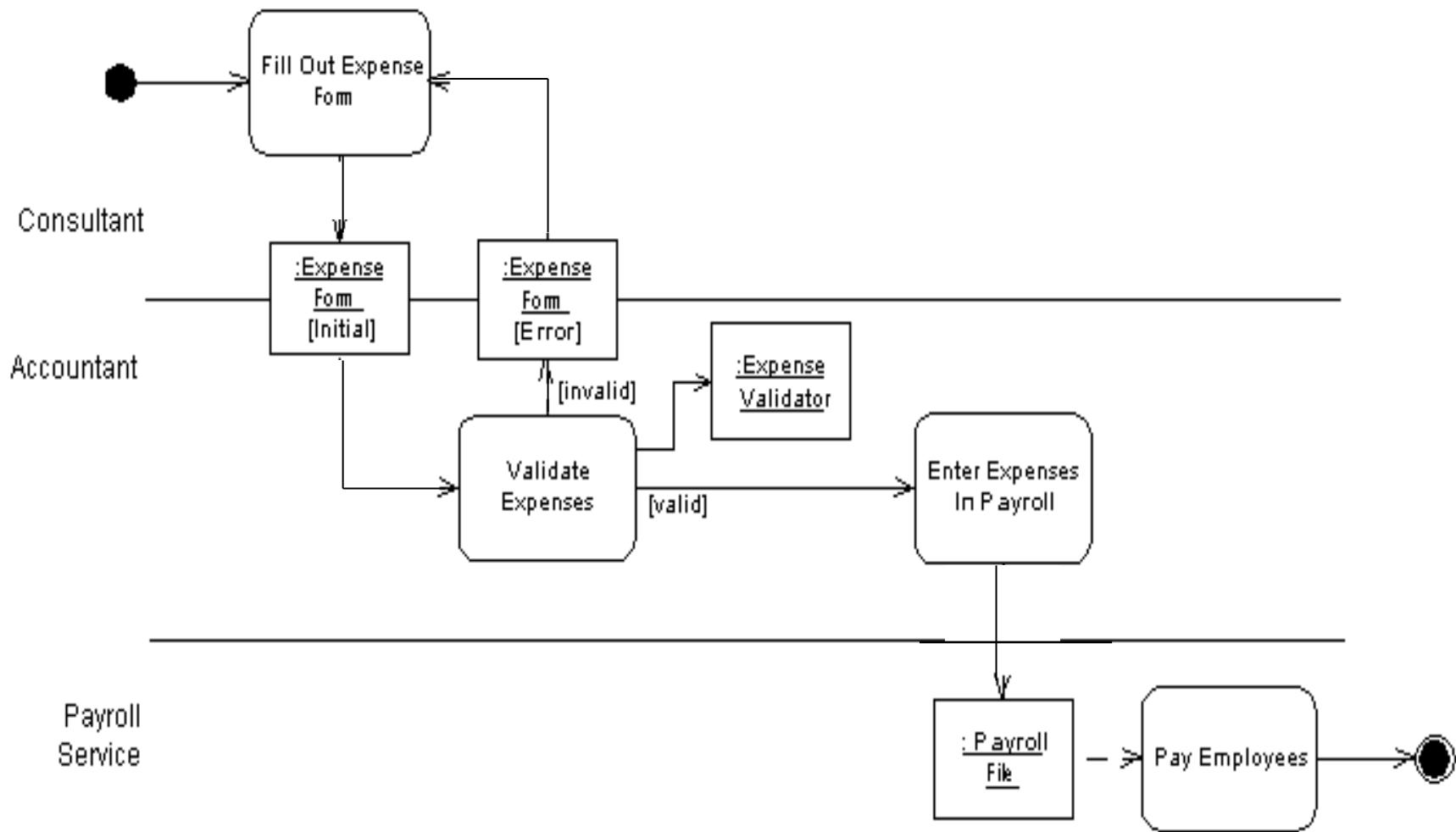
# Partycje (swimlanes)



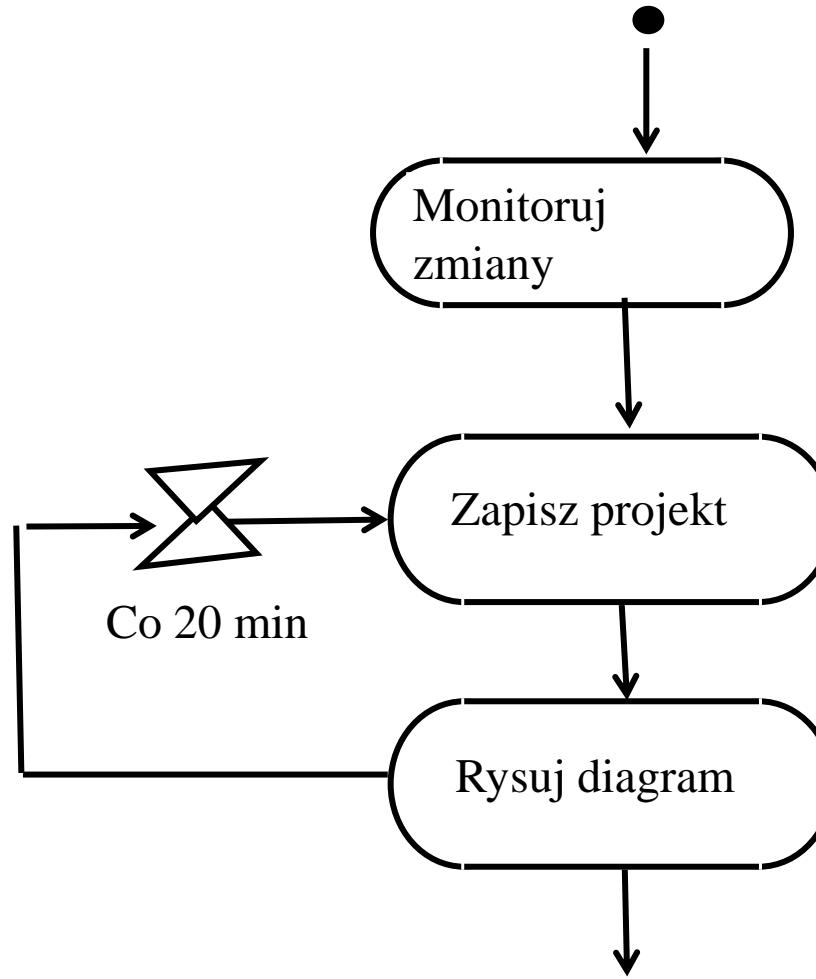
przepływ  
sygnałów  
(bodźców  
inicjujących  
czynność,  
akcję).



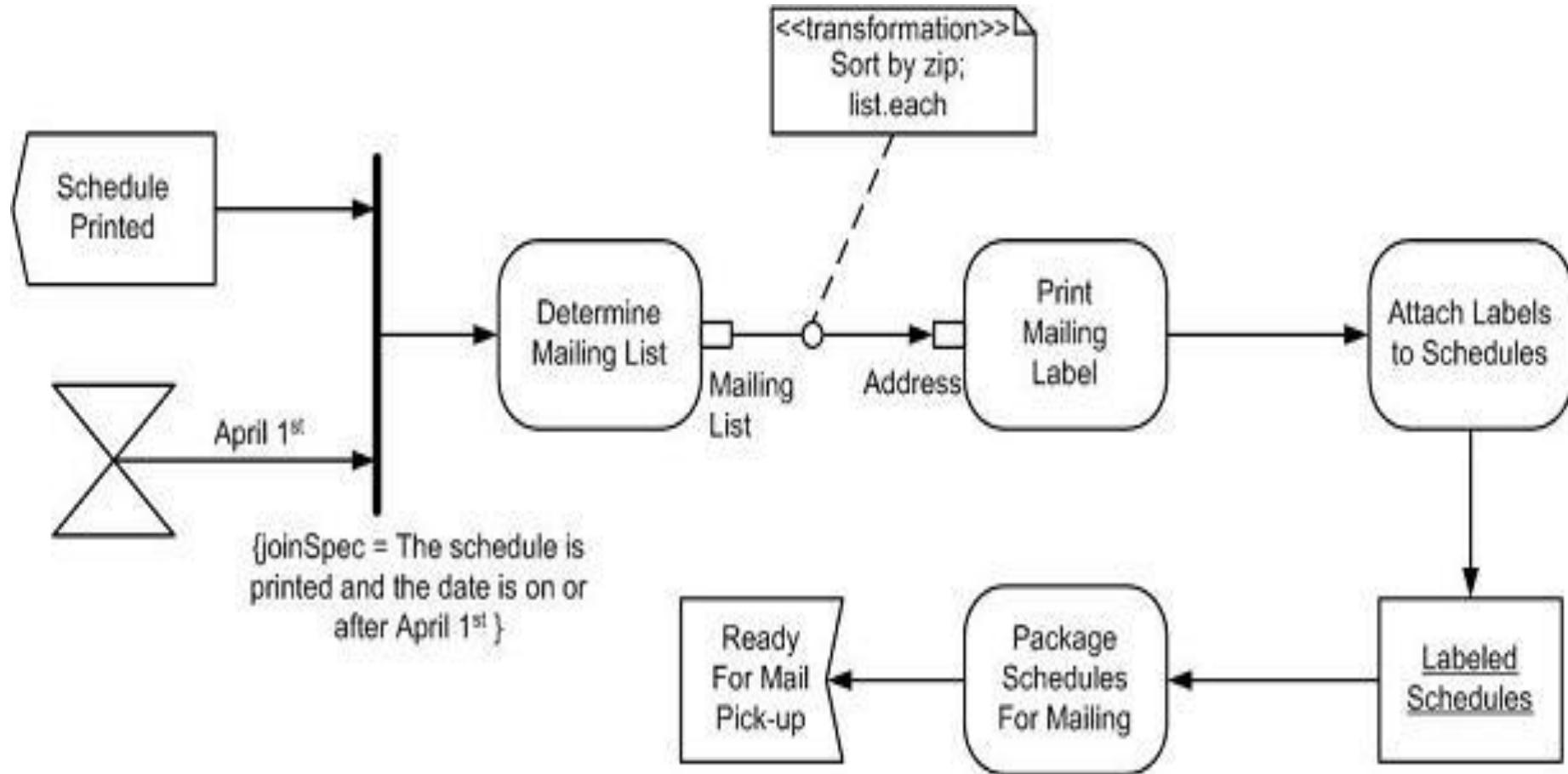
# Obiekty na diagramie czynności



# Diagram czynności z „czasem”

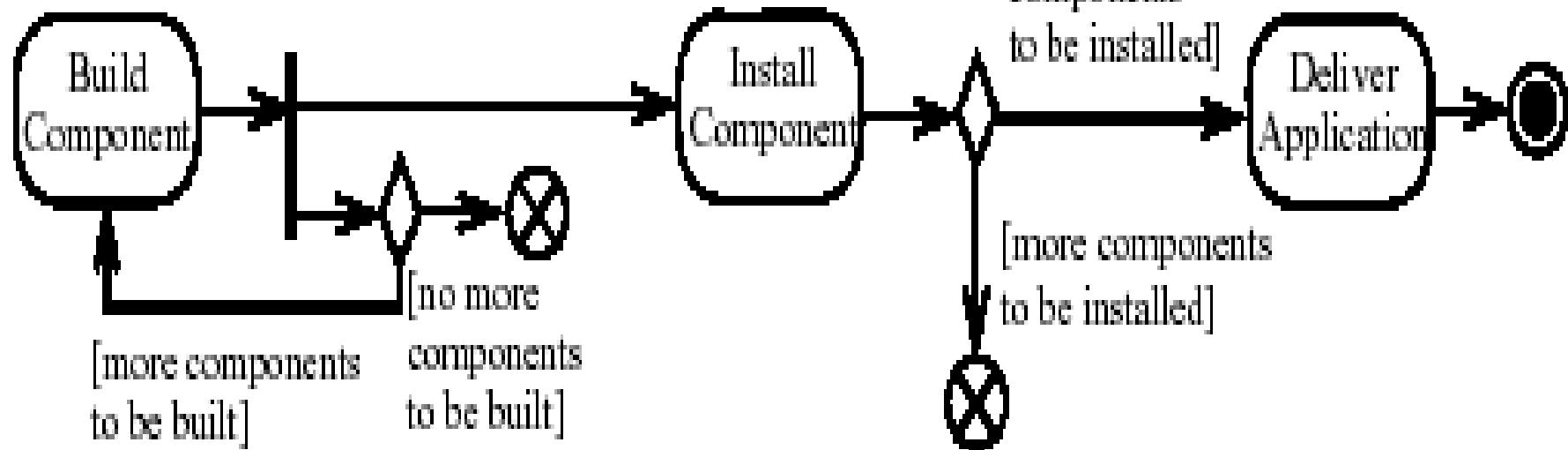


# Dystrybucja poczty



# Węzeł końcowy i zakończenie przepływu

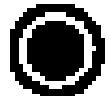
Wiele komponentów jest tworzonych i instalowanych równolegle



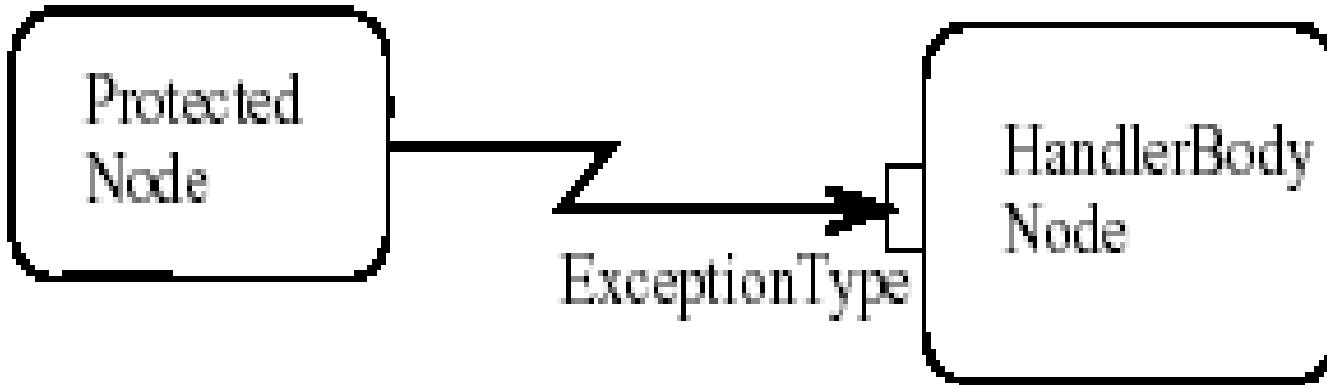
Flow final - zakończenie danego przepływu (wątku)



Węzeł końcowy (Activity final) – zatrzymane są wszystkie przepływy (np. równoległe wątki) danej czynności

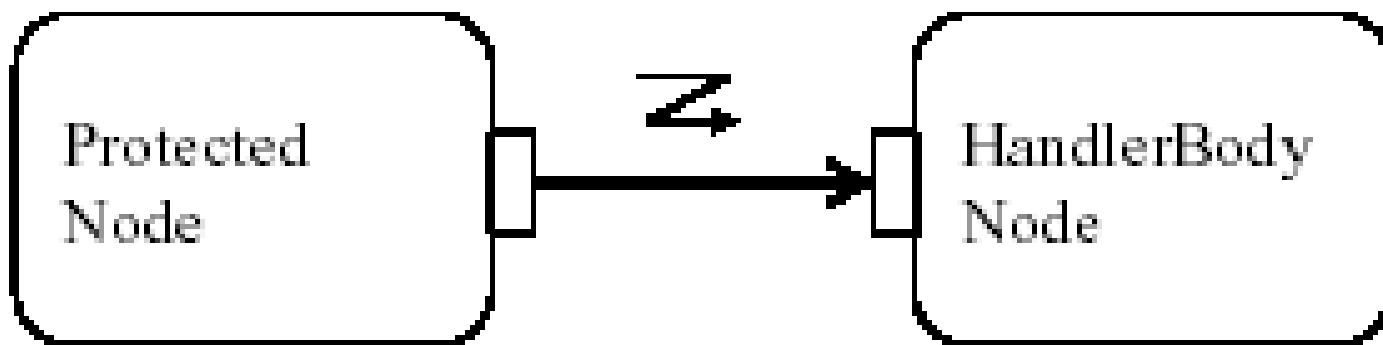


# Obsługa wyjątków (exception handler)



czynność chroniona  
– podczas jej wykonania  
może wystąpić wyjątek

czynność wykonywana  
po wystąpieniu  
podczas czynności chronionej  
wyjątku danego typu



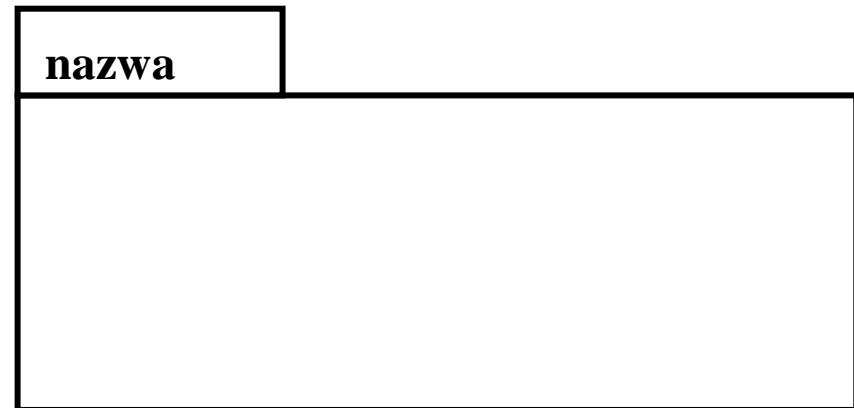
# Pakiety w UML 2.x

---

Dr hab. inż. Ilona Bluemke

# Pakiet

- Pojemnik zawierający byty dowolnego rodzaju (np. klasy, interfejsy, przypadki użycia, inne pakiety itd.)
- Symbol graficzny



## Pakiet - 2

- jest grupą innych bytów: bez tożsamości i egzemplarzy
  - określa przestrzeń nazw :
- pakiet\_zewnętrzny::pakiet\_wewnętrzny::element***
- ogranicza widoczność: publiczne, prywatne, chronione

# Zastosowanie pakietów

## 1. Zarządzanie projektem

- podział systemu na heterogeniczne pakiety, grupujące np. klasy.
- Pakiet zawiera artefakty związane z budową części systemu

## 2. Dekompozycja systemu

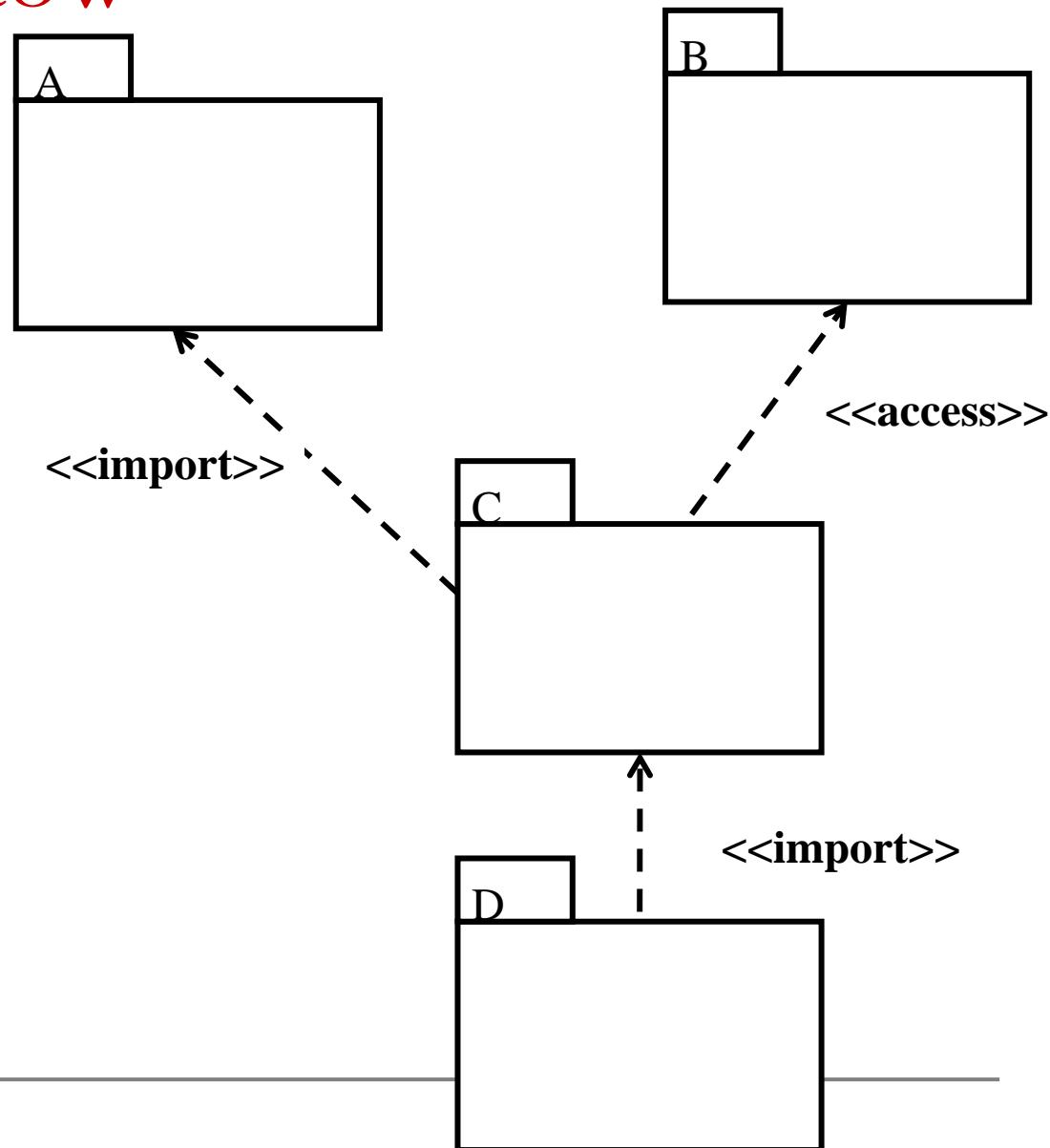
- podział systemu na jednorodne pakiety (klas) opisuje strukturę systemu na wyższym poziomie abstrakcji. Podział na pakiety może poprawić modyfikowalność.

# Diagram pakietów

Elementy zdefiniowane w pakiecie A są publicznie importowane do pakietu C.

Elementy w pakiecie B są importowane prywatnie do pakietu C.

Elementy pakietu A i C są dostępne dla pakietu D. Import pakietu nie jest przechodni.

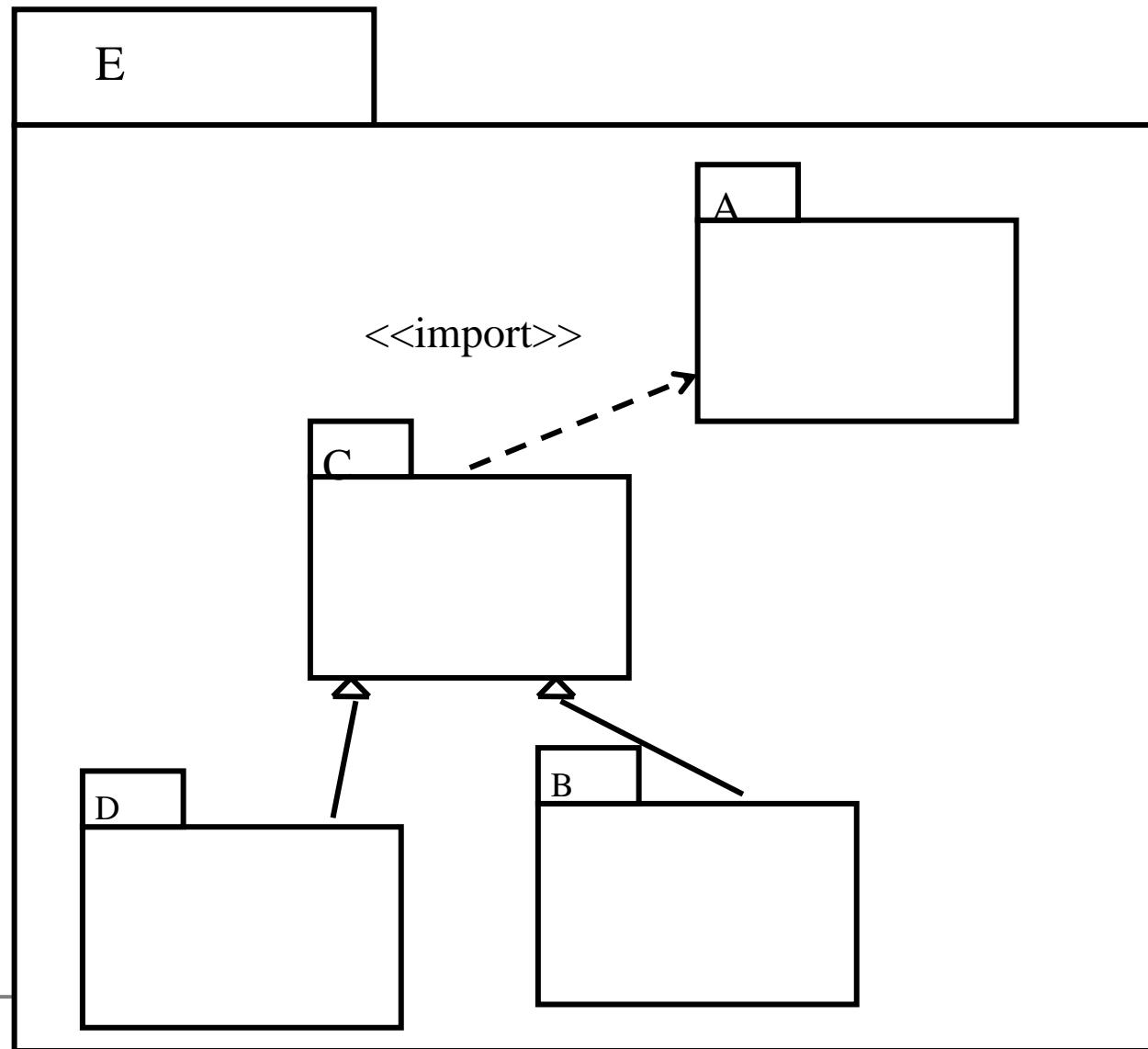


# Przykład pakietów

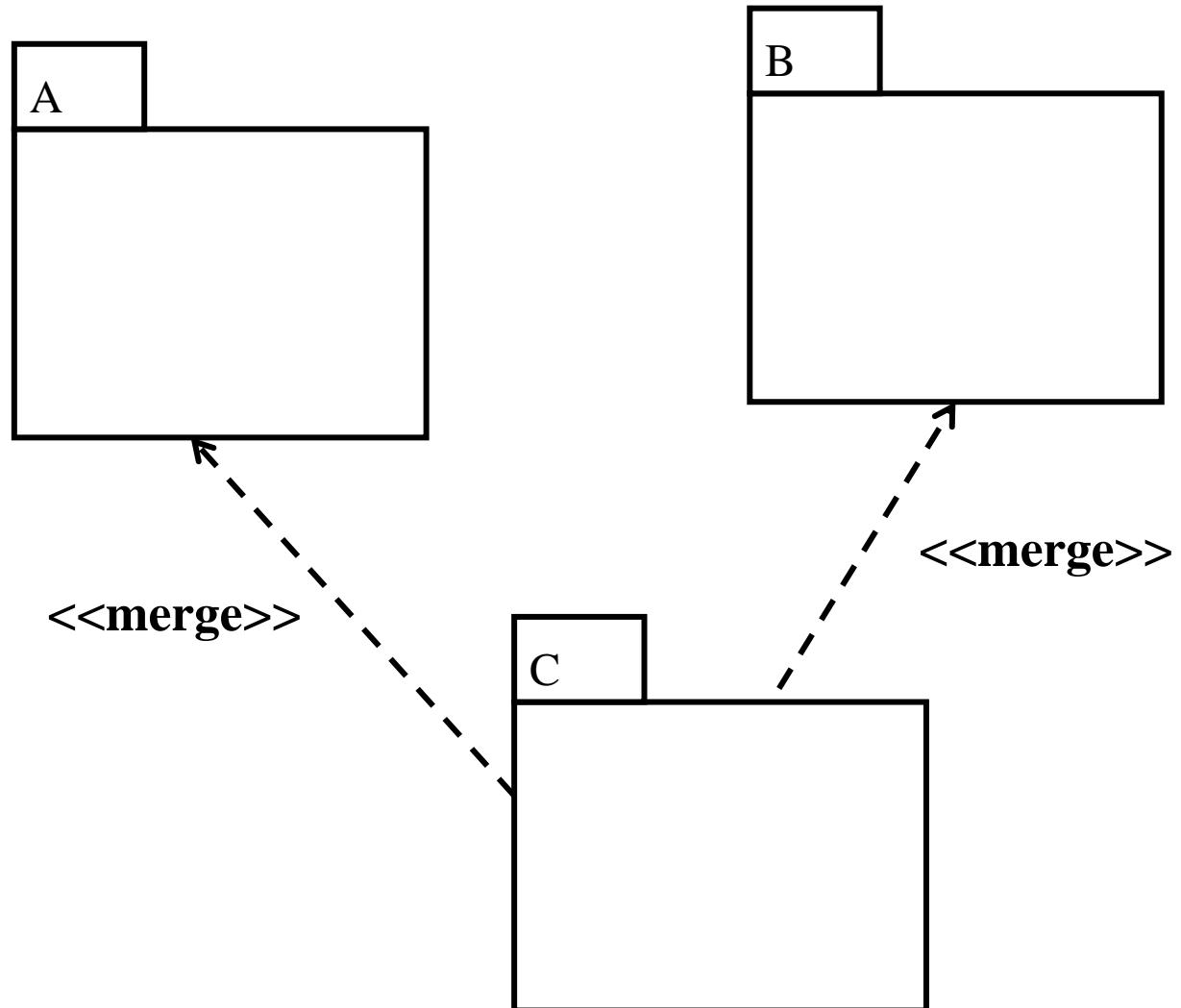
Pakiety **D** i **B**  
są  
specjalizacj  
ami pakietu  
**C**.

Pakiet **C**  
zależy od  
pakietu **A**.

Pakiety **A**, **B**,  
**C**, **D**  
znajdują się  
w pakiecie  
**E**.



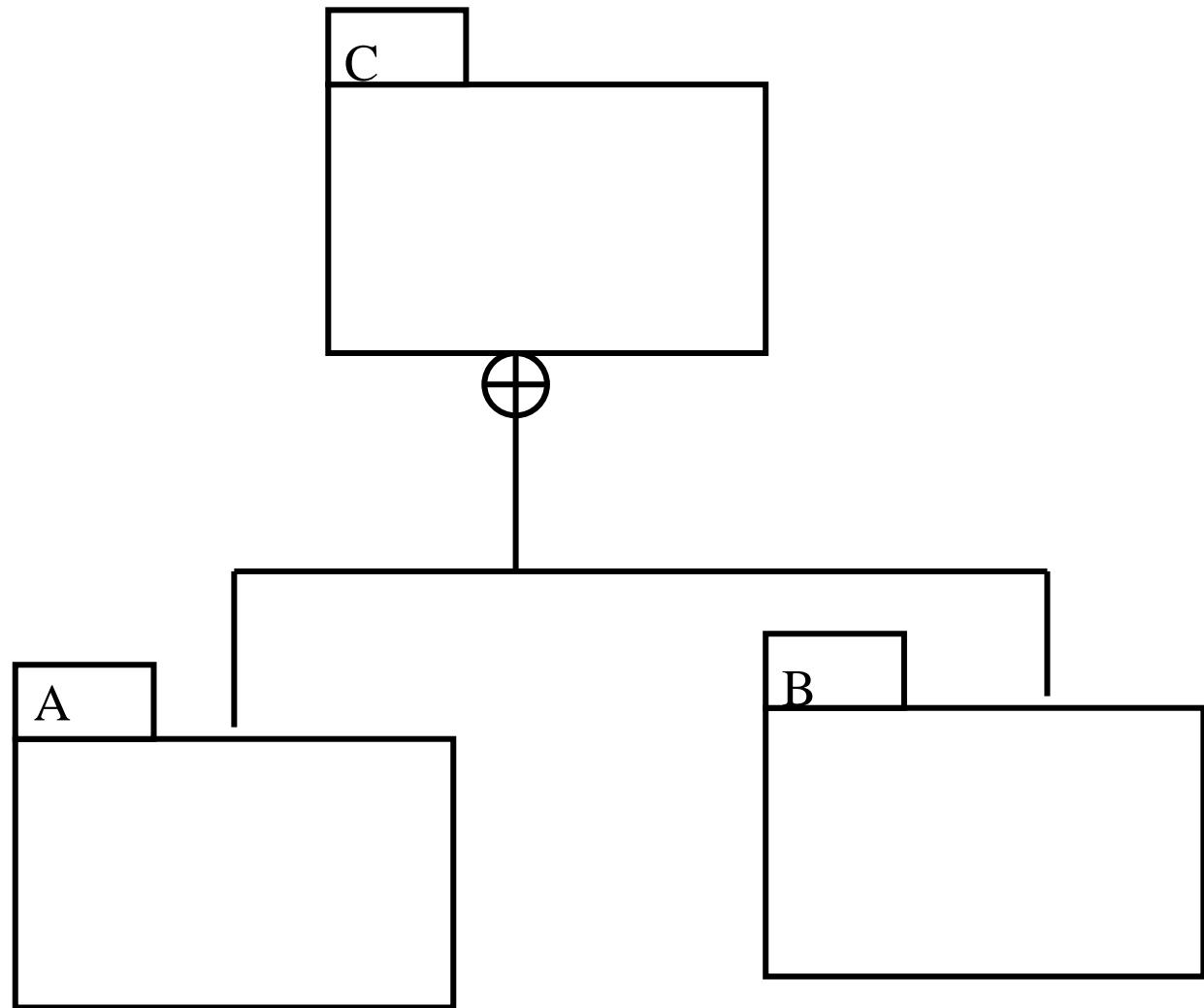
# Scalanie pakietów



# stereotypy w diagramie pakietów

- **<<system>>** domyślny pakiet obejmujący cały system
- **<<subsystem>>** niezależna część systemu
- **<<facade>>** pakiet opisujący interfejs innego pakietu
- **<<stub>>** reprezentant (symulator) innego pakietu
- **<<framework>>** zrąb - pakiet parametryzowanych wzorców architektonicznych
- **<< modelLibrary>>** pakiet grupujący elementy, które mają być użytkowane przez inne pakiety

# Alternatywna notacja zawierania pakietów



# Proces tworzenia diagramu pakietów

- identyfikacja i nazwanie pakietów
- zakwalifikowanie pakietów (*subsystem, framework, model*)
- określenie zagnieżdżeń
- określenie zależności
- wyspecyfikowanie zależności (*import, access, merge*)

# ***WZORCE PROJEKTOWE***

Dr hab. inż. Ilona Bluemke

# **Wzorce projektowe - architektura**

---

C. Alexander, S. Ishikawa, M. Silverstein: A Pattern Language, Nowy York, Oxford University Press, 1977

- „**Każdy wzorzec opisuje problem, który ciągle pojawia się w naszej dziedzinie, a następnie określa zasadniczą część jego rozwiązania w taki sposób, by można było zastosować je nawet milion razy za każdym razem w nieco inny sposób**”

**E.Gamma, R. Helm, R. Johnson,  
J.Vlissides**

---

Design Patterns: Elements of Reusable  
Software, Reading, Mass., Addison-Wesley,  
1995  
(banda Czworga)

# **Wzorce w inżynierii oprogramowania**

---

- tworząc programy spotykamy się z problemami, które rozwiązać można w podobny sposób
- w projektowaniu oprogramowania można zastosować wzorce, tak by pomagały w tworzeniu rozwiązań
- „banda czworga” - zaproponowali sposób katalogowania i opisu wzorców
- skatalogowali 23 wzorce
- postulowali wprowadzenie do projektowania obiektowego zasad i strategii opartych na wzorcach projektowych.

# **Powody poznawania wzorców projektowych**

---

- wykorzystanie istniejących, wcześniej sprawdzonych wzorców przyspiesza pracę nad projektem i pozwala uniknąć błędów (nie wymyślamy rozwiązań typowych problemów)
- wzorce zapewniają wspólny punkt odniesienia, ułatwiają pracę i komunikację w zespole
- dają ogólną perspektywę widzenia, uwalniają projektanta od konieczności zbyt wcześniego zgłębiania szczegółów.

# Kategorie wzorców projektowych

---

## Strukturalne :

Do powiązania istniejących obiektów. Np.:

- *fasada, adapter* - do obsługi interfejsów,
- *most, dekorator* - do powiązania implementacji i abstrakcji.

## Czynnościowe:

Do manifestacji zmiennego zachowania np.

- *Strategia* - do zawierania zmienności.

## Kreacyjne:

Do utworzenia obiektów. Np.:

- *fabryka abstrakcyjna, singleton, metoda produkcyjna* – tworzenie instancji.

# Metoda szablonu

---

## Opis wzorca:

- **Intencja:** Zdefiniowanie szkieletu algorytmu i pozostawienie implementacji niektórych jego operacji klasom pochodnym. Możliwość ponownego zdefiniowania operacji bez konieczności zmieniania struktury algorytmu.
- **Problem:** Istnieje stała procedura, której poszczególne kroki mogą różnić się szczegółami.
- **Rozwiążanie:** Pozwala na zdefiniowanie zmieniających się operacji przy zachowaniu ogólnej procedury.

# Metoda szablonu-2

---

- **Uczestnicy i współpracownicy:** *Klasa\_abstrakcyjna* definiuje podstawowe, wspólne zachowanie klas pochodnych w postaci metody *Metoda\_szablonu* oraz szczegółowych operacji *Operacja\_szczeg1*, których implementacji muszą dostarczyć klasy potomne.
- **Konsekwencje:** Szablony stanowią rozwiązanie służące powtórнемu wykorzystaniu kodu. Pomagają także zapewnić implementację określonych operacji. Wiążą one poszczególne, zmieniające się operacje wewnątrz klasy *Klasa\_konk*.

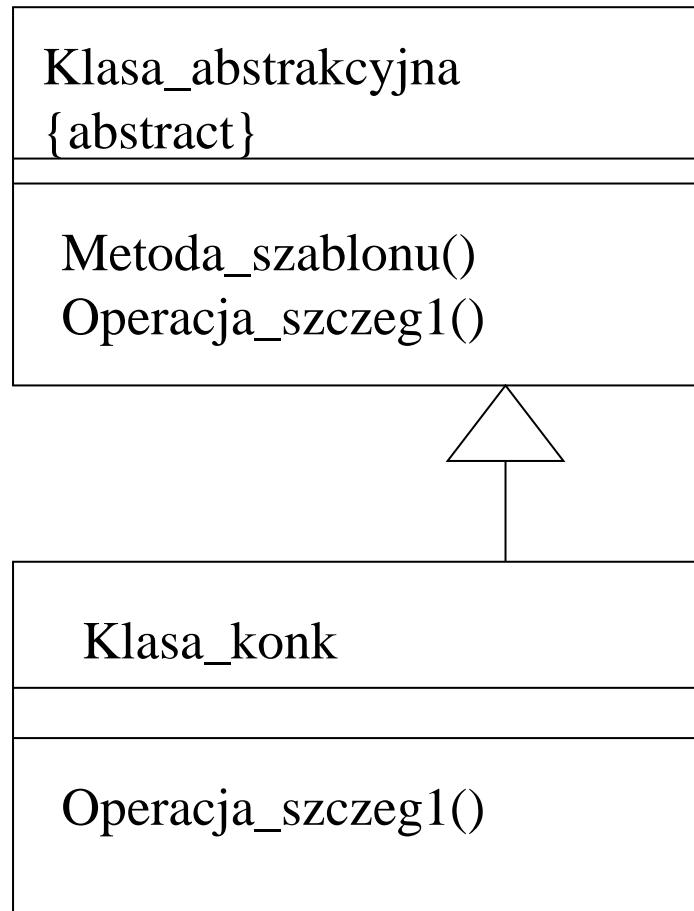
# **Metoda szablonu-3**

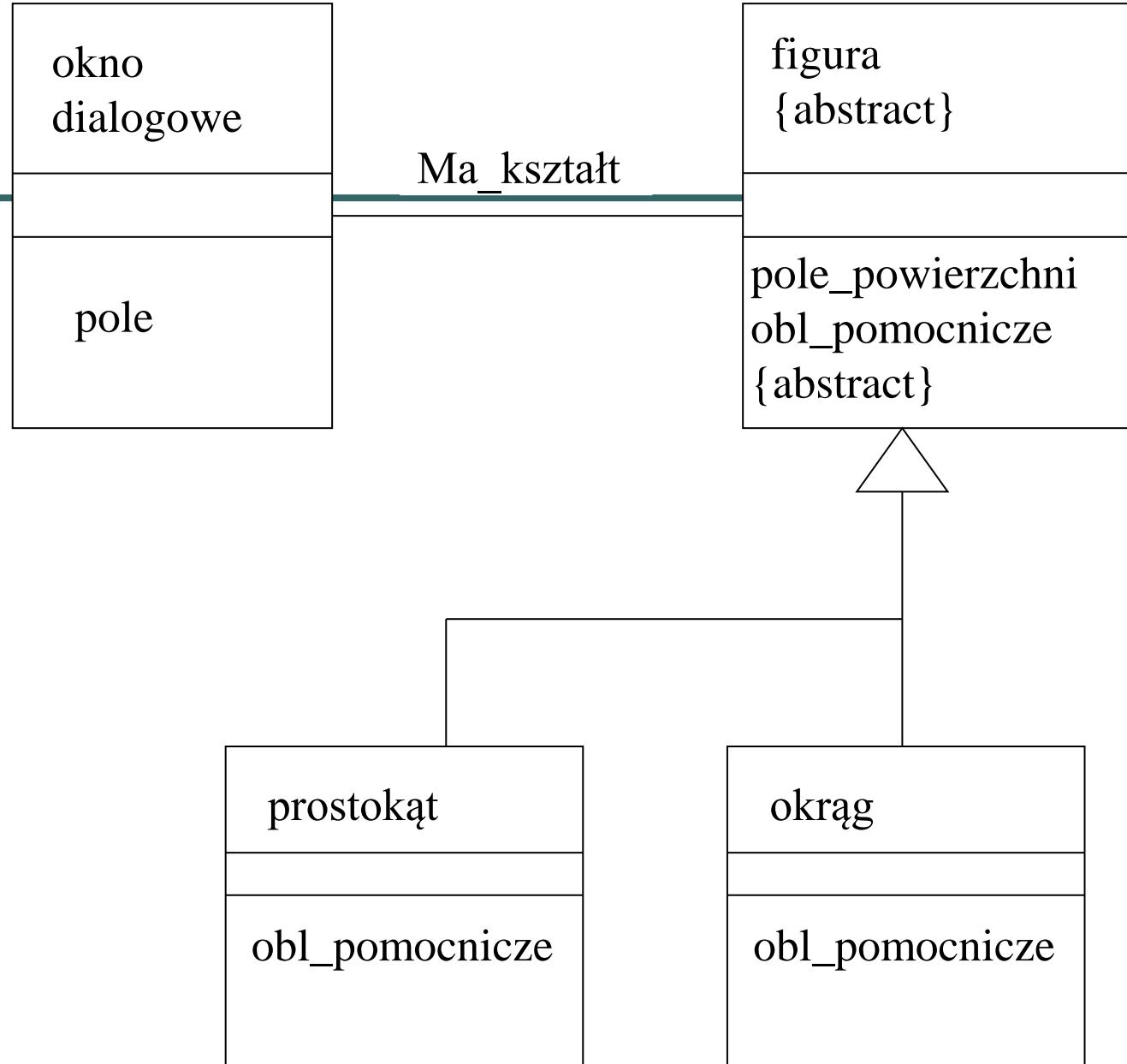
---

- **Implementacja:** Utworzenie klasy abstrakcyjnej implementującej ogólną procedurę za pomocą metod abstrakcyjnych. Implementacja tych metod musi zostać dostarczona w klasach pochodnych.
- **Referencje:** A.Shalloway, J.R. Trott „Projektowanie zorientowane obiektowo – wzorce projektowe”, Helion 2001

# Metoda szablonu-4

---





# **Metoda produkcyjna- wirtualny konstruktor**

---

Uniezależnienie algorytmu tworzenia nowych produktów od samych produktów

## **Opis wzorca:**

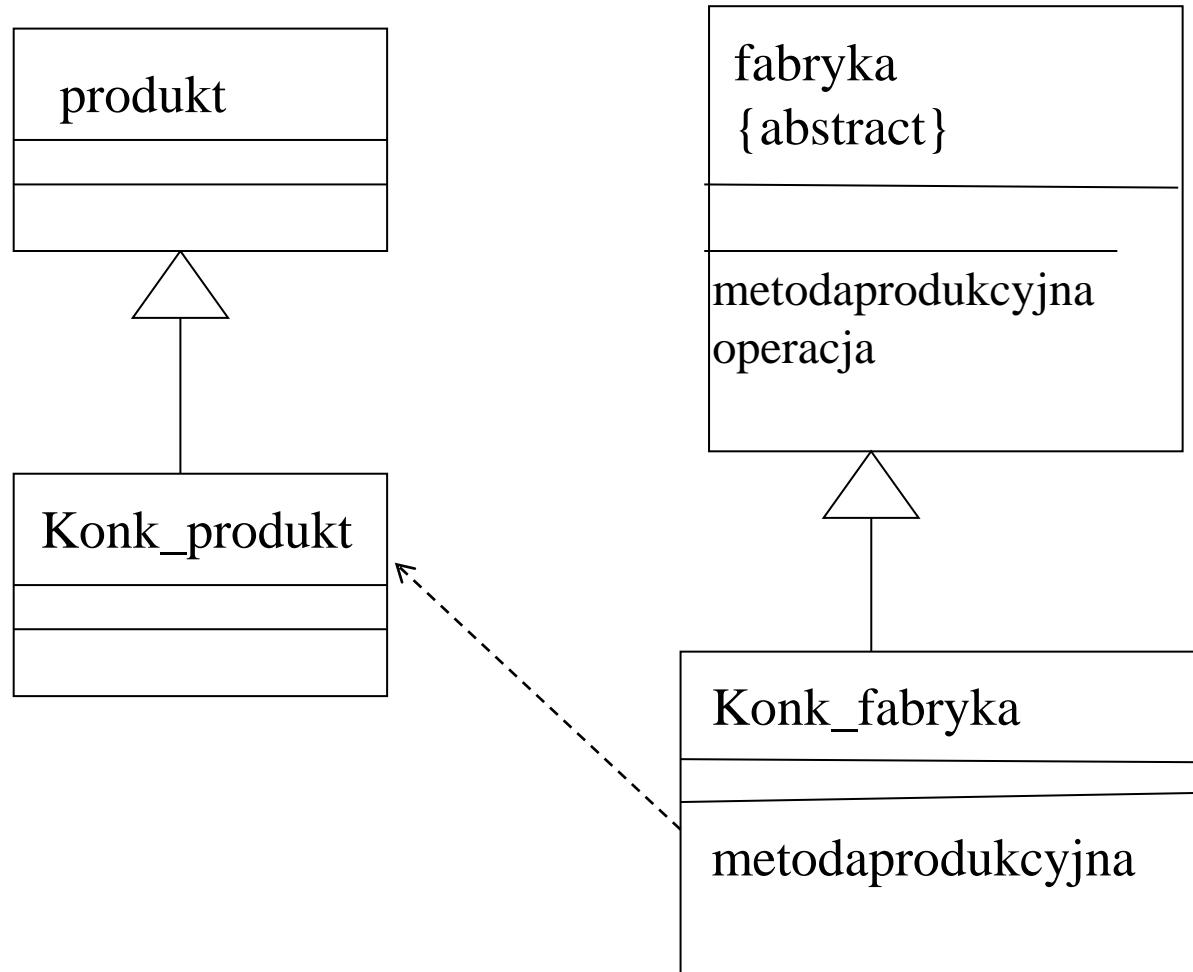
- **Intencja:** Zdefiniowanie interfejsu służącego do stworzenia obiektu, ale pozostawienie decyzji o klasie tworzonego obiektu klasom pochodnym.
- **Problem:** Klasa musi utworzyć obiekt jednej z klas pochodnych innej klasy lecz nie wie dokładnie której. Metoda produkcyjna pozwala podjąć tę decyzję klasie pochodnej.
- **Rozwiązanie:** Klasa pochodna podejmuje decyzję o klasie i sposobie utworzenia obiektu.

# Metoda produkcyjna- 2

---

- **Uczestnicy i współpracownicy:** *produkt* określa interfejs obiektu tworzonego przez metodę *fabryki*. *fabryka* definiuje interfejs zawierający metodę produkcyjną.
- **Konsekwencje:** Aby utworzyć obiekt klasy *Konk\_produkt*, należy utworzyć klasę pochodną klasy *fabryka*.
- **Implementacja:** Wykorzystuje abstrakcyjną metodę klasy abstrakcyjnej (metodę wirtualną w C++). Klasa abstrakcyjna używa tej metody, kiedy musi utworzyć odpowiedni obiekt, ale nie wie jaki obiekt będzie w rzeczywistości utworzony.

# Metoda produkcyjna-3



# Metoda produkcyjna-4

---

- Nowy produkt definiowany jako podklasa klasy *produkt*, tworzona jest konkretna fabryka odpowiedzialna za jego tworzenie.
- Algorytm produkcji w metodzie *fabryka.operacja* (*produkt:=metodprodukcyjna*).
- *Konk\_fabryka.metodprodukcyjna* zwraca nowy *Konk\_produkt*. Wadą jest tworzenie nowej podklasy dla każdego rodzaju produktu

# Fabryka abstrakcyjna

---

Wzorzec fabryka abstrakcyjna jest wykorzystywany w sytuacjach, gdy wybór określonej konfiguracji powinien powodować tworzenie grupy określonych produktów.

Dodanie nowej konfiguracji to zdefiniowanie nowej konkretnej fabryki oraz grupy nowych produktów. W trakcie działania systemu wykorzystywana jest tylko jedna konkretna fabryka. Wybór tej klasy nie ma wpływu na żądania kierowane przez klienta, poprzez dziedziczenie interfejsu.

# Fabryka abstrakcyjna - 2

---

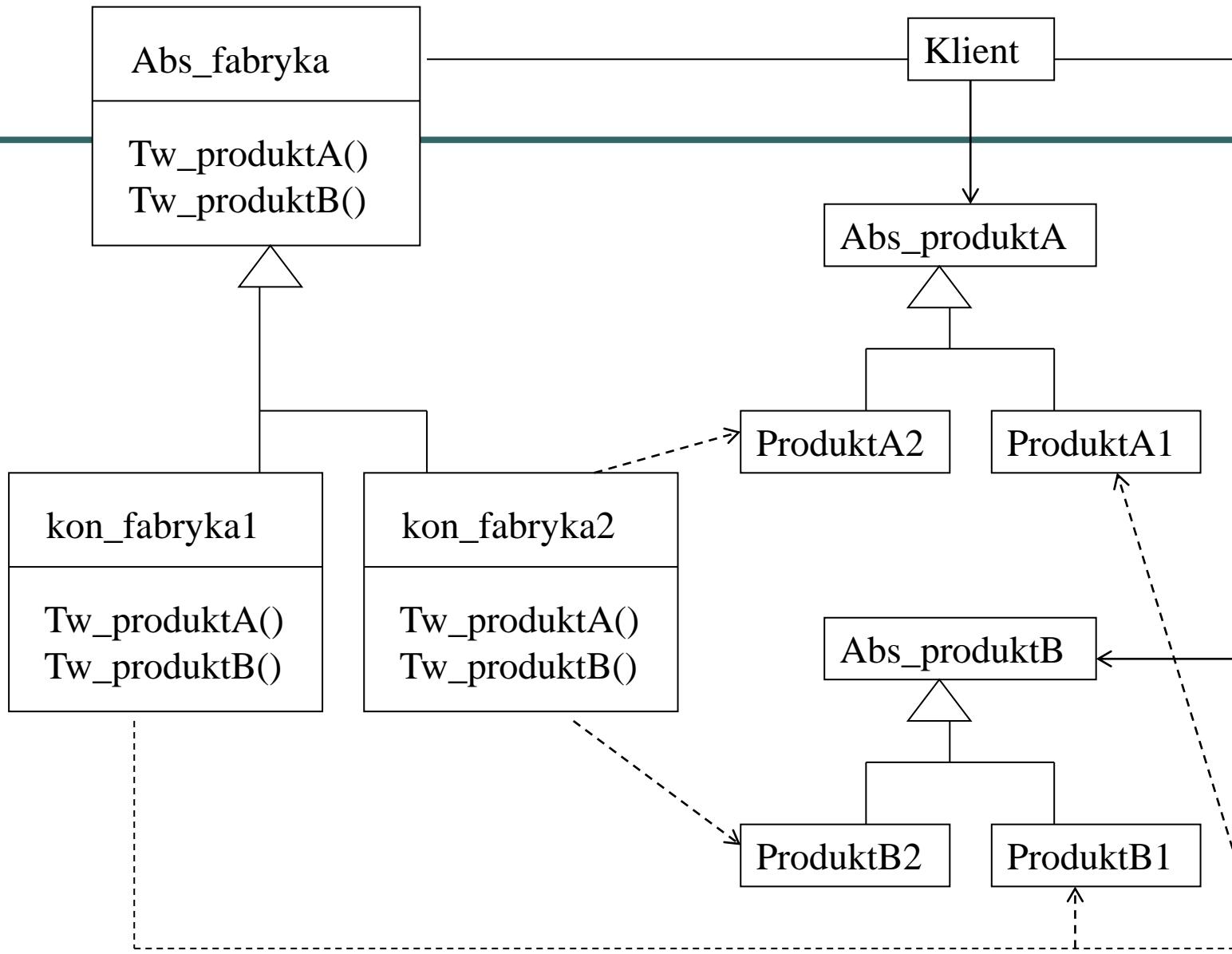
## Opis wzorca:

- **Intencja:** Uzyskanie rodzin obiektów właściwych w określonym przypadku
- **Problem:** Utworzenie odpowiednich rodzin obiektów
- **Rozwiążanie:** Koordynuje utworzenie rodzin obiektów. Podsuwa sposób pozwalający wydzielić z obiektów użytkownika reguły tworzenia obiektów, które są przez nie używane.
- **Uczestnicy i współpracownicy:** *Abs\_Fabryka* definiuje interfejs określający sposób utworzenia każdego z obiektów danej rodziny. Typowo każda z rodzin obiektów posiada własną klasę *kon\_fabryka*.

# Fabryka abstrakcyjna - 3

---

- **Konsekwencje:** Wzorzec izoluje reguły opisujące sposób wykorzystania obiektów od reguł decydujących o utworzeniu tych obiektów.
- **Implementacja:** Definiuje klasę abstrakcyjną specyfikującą tworzone obiekty. Dla każdej z rodzin obiektów implementuje się klasę konkretną. W celu dokonania wyboru tworzonych obiektów mogą być zastosowane pliki konfiguracyjne lub tabela bazy danych.
- **Referencje:** A.Shalloway, J.R. Trott „Projektowanie zorientowane obiektowo – wzorce projektowe”, Helion 2001



# Adapter

---

## Opis wzorca:

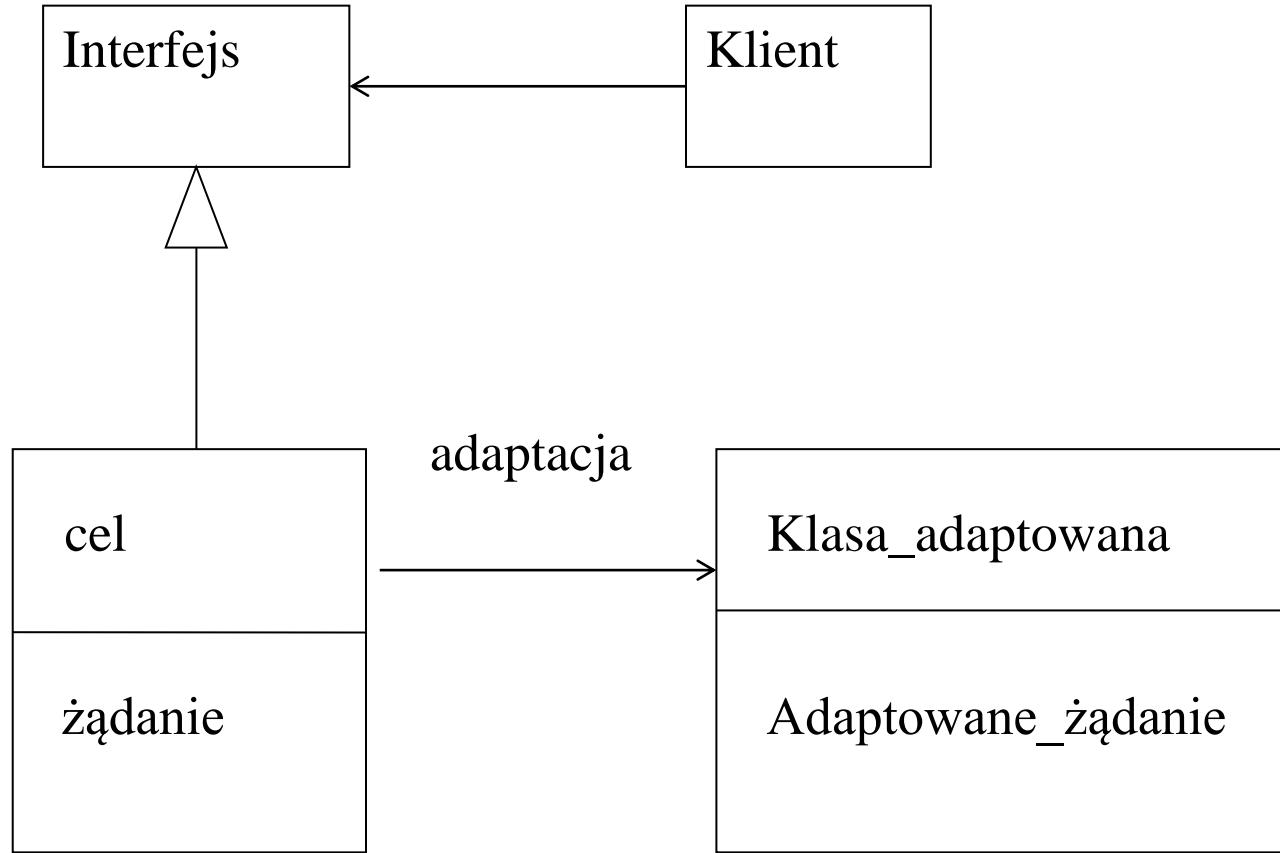
- **Intencja:** dopasowanie istniejącego obiektu (nad którym programista nie ma kontroli) do określonego sposobu wywołania.
- **Problem:** Obiekt przechowuje potrzebne dane oraz zachowuje się w pożądany sposób, jednak posiada nieodpowiedni interfejs. Często włączamy go do hierarchii dziedziczenia pewnej klasy abstrakcyjnej posiadanej lub definiowanej.
- **Rozwiązanie:** Adapter obudowuje obiekt pożądanym interfejsem

# Adapter - 2

---

- **Uczestnicy i współpracownicy:** Adapter dostosowuje interfejs klasy *klasa-adaptowana* tak, by był zgodny z interfejsem klasy bazowej *cel*. Umożliwia to użytkownikowi wykorzystanie obiektu klasy *klasa-adaptowana* oraz instancji klasy *cel*.
- **Konsekwencje:** Zastosowanie wzorca adapter pozwala dopasować istniejące obiekty do tworzonych struktur klas i uniknąć ograniczeń związanych z ich interfejsem.
- **Implementacja:** Zawiera istniejącą klasę w nowej klasie, która posiada wymagany interfejs i wywołuje odpowiednie metody zawieranej klasy

# Adapter - 3



## Adapter - 4

---

- Interfejs jest definiowany w abstrakcyjnej klasie *Interfejs* a jego funkcjonalność w podklasach. Metody klasy *cel* mogą dokonywać konwersji typów lub tworzyć nowe algorytmy określające funkcjonalność nie przewidzianą w adaptowanych klasach.
- Wywołanie metody *klasy-adaptowanej* jest dokonywane przez traversowanie związku.

# Strategia

---

Wzorzec jest stosowany w celu ukrycia szczegółów algorytmu przed klientem, konfigurowania różnych zachowań klas.

W metodzie klasy abstrakcyjnej można umieścić wspólne części algorytmu, w podklasach ich specjalizację.

## Opis wzorca:

- **Intencja:** Umożliwia użycie różnych wersji algorytmu czy reguł biznesowych w zależności od kontekstu

## Strategia -2

---

- **Problem:** Wybór algorytmu zależy od używającego go obiektu lub danych na których operuje. Jeśli algorytm nie zmienia się wzorzec nie jest potrzebny.
- **Rozwiązanie:** Separuje wybór wersji algorytmu od jego implementacji. Umożliwia wybór algorytmu na podstawie kontekstu.

## Strategia -3

---

- **Uczestnicy i współpracownicy:**

*strategia* specyfikuje sposób użycia różnych algorytmów.

*strategiaA*, *strategiaB* implementują konkretny algorytm.

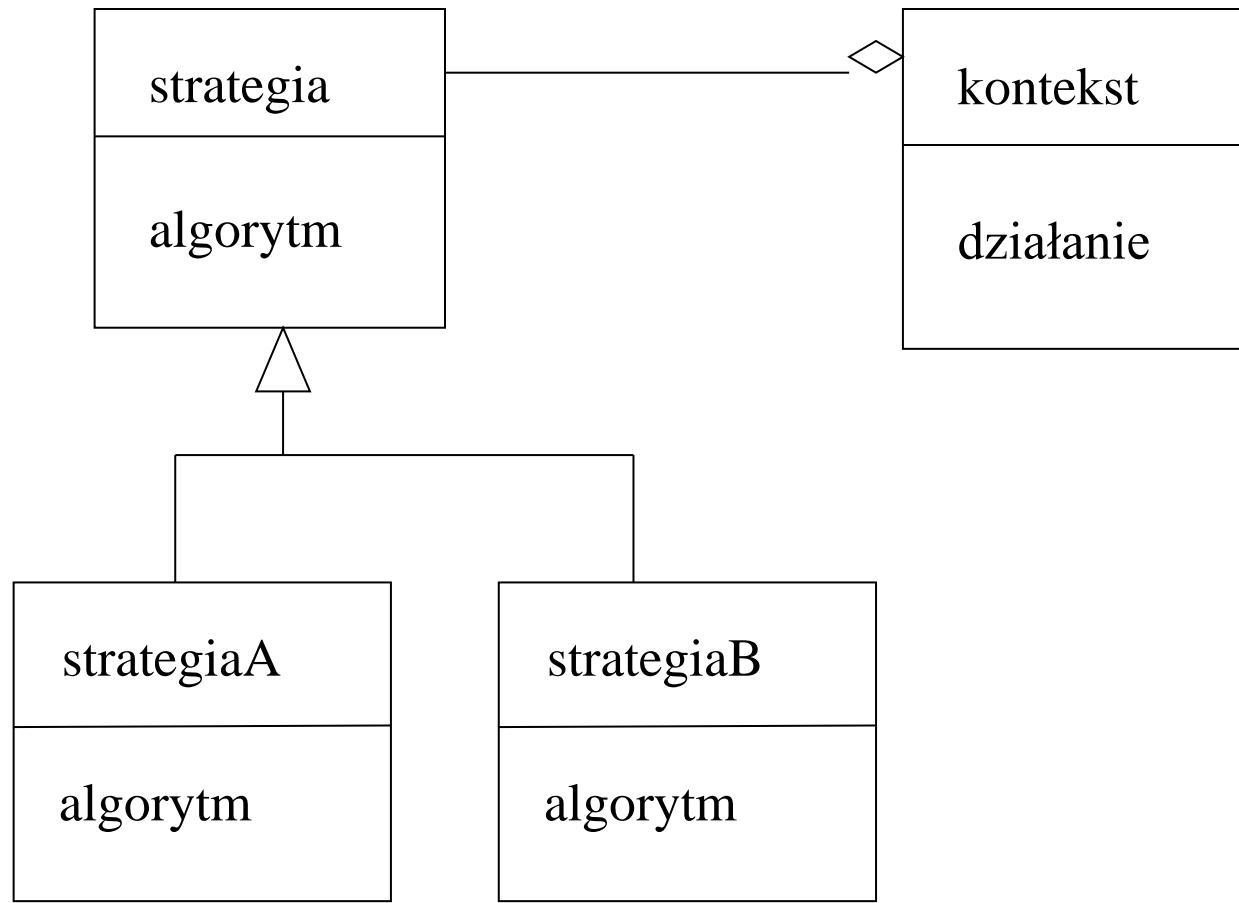
*kontekst* korzysta z obiektu klasy *strategiaA* czy *strategiaB* odwołując się do niego jak do obiektu klasy *strategia*. *strategia* i *kontekst* współdziałały w wyborze algorytmu, czasami *strategia* kieruje w tym celu żądanie do klasy *kontekst*. *kontekst* przekazuje żądania korzystających z niego obiektów klasie *strategia*.

# Strategia - 4

---

- **Konsekwencje:** Strategia definiuje rodzinę algorytmów. Instrukcje wyboru mogą zostać wyeliminowane. Wszystkie algorytmy muszą być wywoływanie w ten sam sposób.
- **Implementacja:** Klasa, która używa algorytmu *kontekst* zawiera klasę abstrakcyjną *strategia*, która posiada metodę abstrakcyjną określającą sposób wywołania algorytmu. Każda z jej klas pochodnych implementuje algorytm we własnym zakresie. W przypadku, gdy metody mają wspólną część, metoda ta nie musi być abstrakcyjna.
- **Referencje:** A.Shalloway, J.R. Trott „Projektowanie zorientowane obiektowo – wzorce projektowe”, Helion 2001

# Strategia - 5



# Obserwator

---

Wzorzec stosowany przy projektowaniu środowisk prezentacji danych.

- **Opis wzorca:**
- **Intencja:** Definiuje „jeden-do-wielu” zależności między obiektami. Zmiana stanu jednego obiektu automatycznie propaguje na związane z nim obiekty.
- **Problem:** Należy powiadomić zmieniającą się listę obiektów o pewnym zdarzeniu.
- **Rozwiązanie:** Obiekty klasy *obserwator* przekazują odpowiedzialność za monitorowane zdarzenia centralnemu obiekowi *dana*.

## Obserwator -2

---

- **Uczestnicy i współpracownicy:** *dana* wie, które obiekty klasy *obserwator* należy zawiadomić, ponieważ rejestrują się u niego. *dana* zawiadamia obiekty klasy *obserwator* o wystąpieniu zdarzenia. Obiekty klasy *obserwator* są odpowiedzialne za zarejestrowanie się u obiektu *dana* i uzyskanie od niego potrzebnej informacji, gdy zostaną powiadomione o zdarzeniu.
- **Konsekwencje:** *dana* może niepotrzebnie powiadamiać o zdarzeniu różne rodzaje obserwatorów nawet, wtedy gdy są zainteresowane jego wystąpieniem tylko w pewnych przypadkach.

## Obserwator -3

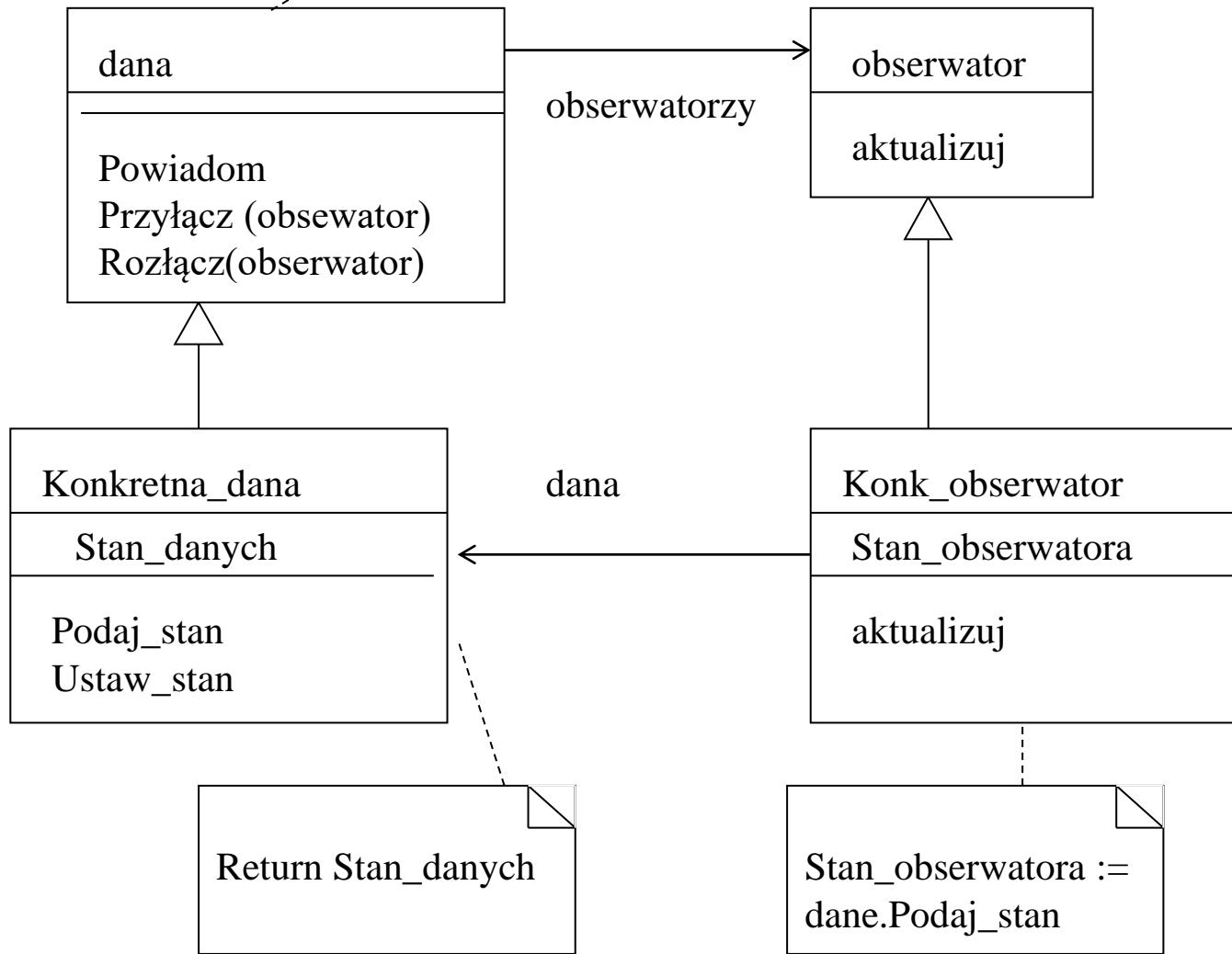
---

- **Implementacja:** Tworzy obiekty klasy ***obserwator***, które mają być informowane o wystąpieniu pewnego zdarzenia i w tym celu rejestrują się u obiektu klasy ***dana***.

Kiedy zachodzi zdarzenie, ***dana*** powiadamia zarejestrowane obiekty klasy ***obserwator***.

- **Referencje:** A.Shalloway, J.R. Trott „Projektowanie zorientowane obiektowo – wzorce projektowe”, Helion 2001

Dla wszystkich obserwatorów o wykonaj o->aktualizuj



# Obserwator -5

---

- Klasa **dane** – zna swoich obserwatorów, dowolna liczba obserwatorów może obserwować dane. Obserwatorzy są rejestrowani – **przyłącz, rozłącz** (wskazanie na klasę). Metoda *powiadom* informuje wszystkie zarejestrowane obiekty o zmianach wewnątrz niej.
- Klasa **obserwator** – definiuje interfejs dla obiektów, które mają być powiadamiane o zmianach w danych. Obserwator po otrzymaniu informacji o konieczności aktualizacji wysyła żądanu do klasy *dane* z prośbą o podania aktualnego stanu. Powiązanie na poziomie klas konkretnych daje możliwość komunikacji od klas obserwujących do danych.
- Klasa **Konk\_obserwator** – zawiera wskazanie na konkretne dane oraz stan, który powinien być spójny z danymi.
- Klasa **Konkretne\_dane** – powiadamia obserwatorów o zmianie stanu, przechowuje stan

## **Stan**

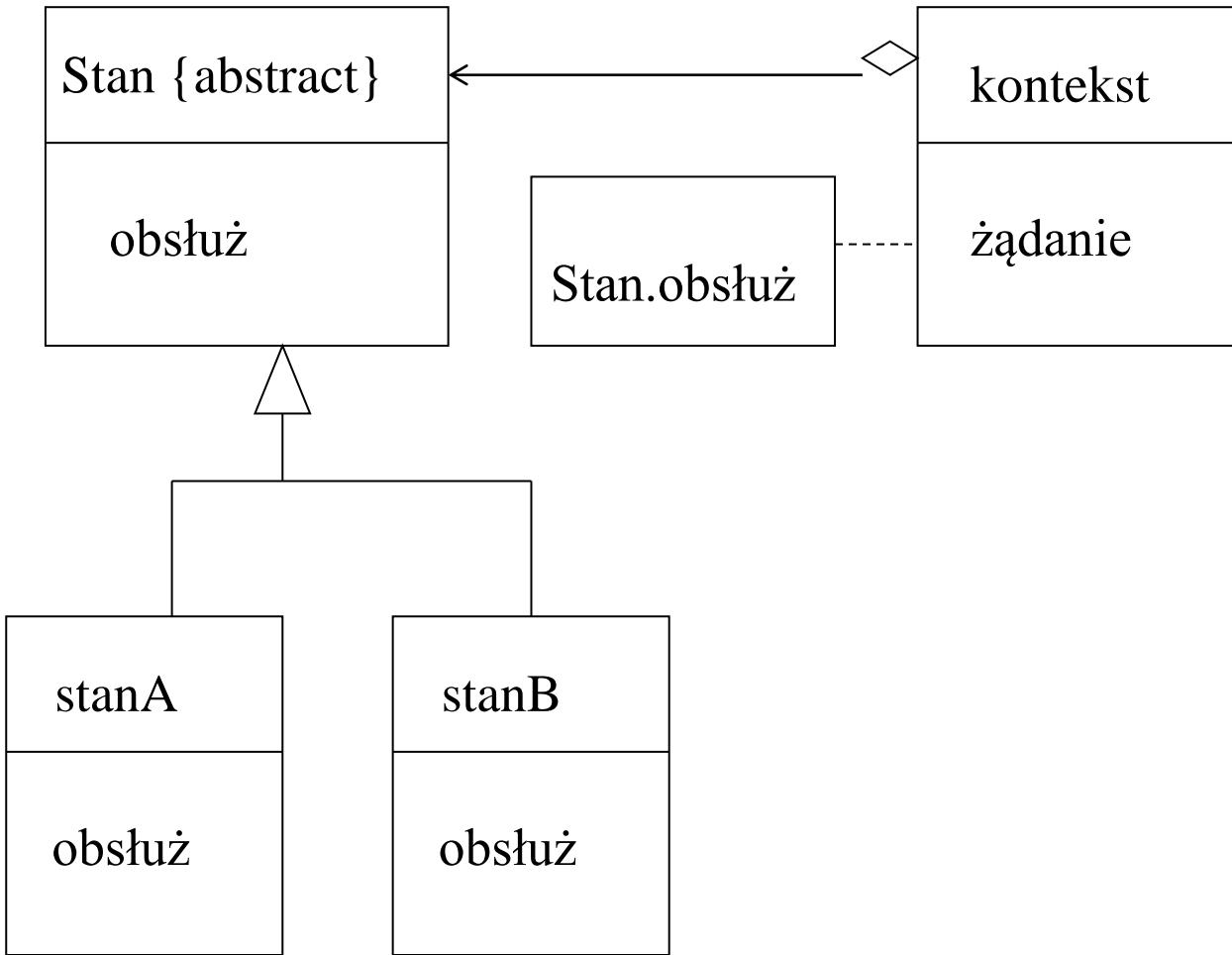
---

Wzorzec stan umożliwia wygodną implementację informacji zawartych w diagramie stanów klasy.

Ułatwia wprowadzanie zmian zachowania, są to zmiany prostych metod klas implementujących stany.

Wadą jest tworzenie rozbudowanej hierarchii klas.

# Stan - 2



## **Stan - 3**

---

- Abstrakcyjna klasa **Stan** definiuje interfejs dla wszystkich klas pochodnych definiujących zachowanie klasy w określonym stanie.
- Klasa **kontekst** udostępnia użytkownikowi usługi. Wykonanie usługi uzależnionej od stanu jest w odpowiedniej klasie.
- Np. TCPConnection, TCPState, TCPEstablished, TCPIlisten, TCPclosed

# Ambasador

---

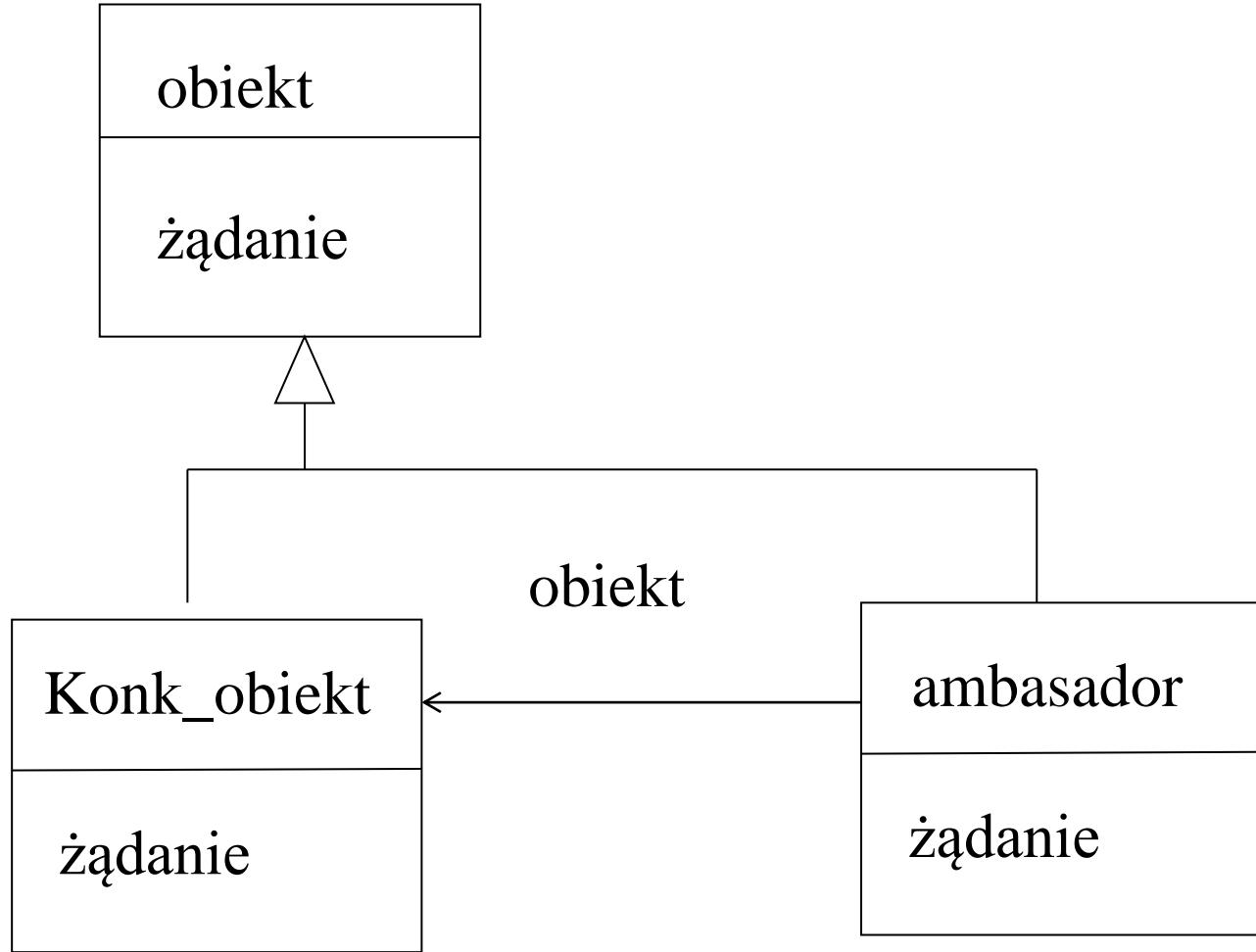
Wykorzystywany w sytuacjach, gdy nie jest konieczne stałe utrzymywanie zainicjowanego konkretnego obiektu w systemie. Rzeczywiste tworzenie obiektu jest opóźnione.

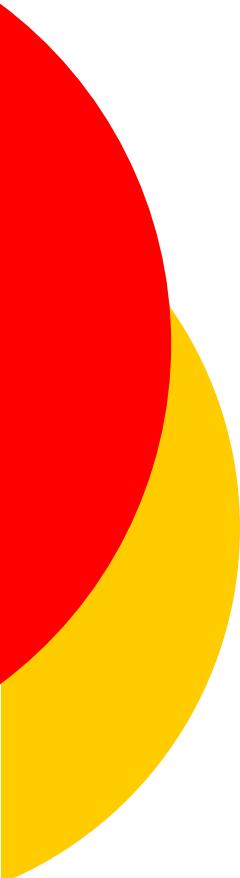
Ambasador ma identyczny interfejs jak reprezentowany przez niego obiekt.

Ambasador powołuje do życia konkretny obiekt.

Wzorzec ambasador stosowany w sytuacjach, gdy utrzymywanie zainicjowanego obiektu powoduje duże zużycie zasobów wpływające na efektywność pracy aplikacji..

# Ambasador -2





# **ESTYMACJA KOSZTÓW OPROGRAMOWANIA**

---

Dr hab. inż. Ilona Bluemke

# szacowania

---

Zarządzający projektem musi znaleźć odpowiedź na pytania:

- ile wysiłku potrzeba na zakończenie aktywności
- ile czasu potrzeba na zakończenie aktywności
- jaki jest koszt czynności.

Pewne szacowania kosztów należy przeprowadzić już we wczesnym stadium (faza strategiczna) np. by określić cenę dla klienta.



## Parametry wpływające na koszty:

---

- sprzętu i oprogramowania
- szkolenia, podróże
- „wysiłku” np. inżynierów i koszty utrzymania firmy (są one zwykle dominujące, np. płace \*2 – koszty utrzymania firmy)

# Współczynniki brane pod uwagę przy określaniu ceny oprogramowania

---

- **Rynek** - np. niższa cena by wejść na rynek
- **Niepewność kosztów** – jeśli koszty trudno jest dokładnie oszacować to do zysku trzeba dodać „coś” na niepewność szacowania
- **Kontraktowe** – np. klient może wyrazić zgodę na własność części kodu – do ewentualnego dalszego wykorzystania przez firmę
- **Zmienne wymagania** - Jeśli prawdopodobne, że wymagania się zmienią to można podać niską cenę (by wgrać przetarg) a określić wysoką cenę zmiany wymagań.
- **Kondycji finansowej firmy** – firmy w złej kondycji mogą obniżać zysk by utrzymać się w branży

# PRODUKTYWNOŚĆ

---

Miary:

- związane z wielkością kodu np. liczba linii kodu/miesiąc (różna dla różnych języków programowania)
- związane z funkcjami (**function point**) (niezależne od języków programowania)  
Albrecht 1979, Albrecht&Gaffnej 1983

# Punkty funkcyjne (function point)

---

Punkty funkcyjne określa się mierząc:

1. wejścia zewnętrzne
2. wyjścia zewnętrzne
3. interakcje użytkownika
4. interfejsy zewnętrzne
5. pliki używane przez system

# **UFC (unadjusted function point)**

---

Każdy z punktów funkcyjnych ma przydzielony współczynnik (3- proste wejście zewnętrzne, 15 złożone pliki wewnętrzne)

**UFC** (unadjusted function point)

**UFC =  $\Sigma$  liczby elem.typu \* współczynnik**

Otrzymana liczba jest modyfikowana współczynnikami określającymi złożoność całego projektu, wydajność, liczbę ponownie użytych elementów.

## AFC ( average number of lines of code)

---

Średnia liczba linii kodu na punkt – jest różna dla różnych języków programowania np.

- 2-300 LOC/FC asembler
- 2-40 LOC/FP języki 4GL

UFC i AFC pozwala na oszacowanie wielkości implementacji

# TECHNIKI ESTYMACJI

---

- Algorytmiczne modelowanie kosztów (na podstawie inf. historycznych i przewidywanej wielkości projektu)
- Osąd ekspertów
- Estymacja poprzez analogię (technika możliwa jeśli inne projekty z tej dziedziny zostały zakończone)
- Prawo Parkinsona – praca rozrasta się aż do końca dostępnego czasu. Koszt określają dostępne zasoby i czas
- Cena do wygrania – koszt określa co klient może wydać na projekt

# Algorytmiczne modelowanie kosztów

---

**Wysiłek = C \* MP<sup>s</sup> \* M**

C – współczynnik złożoności projektu

MP – metryka produktu np. liczba linii kodu

s - bliskie 1 ale odzwierciedla wzrost wysiłku  
przy bardzo dużych projektach

M – współczynnik określający atrybuty  
produkту, procesu rozwoju

Powinno się wykonać estymację na  
**najgorszy, oczekiwany i najlepszy**  
przypadek

# Model COCOMO (Boehm 1981)

---

**Prosty** projekt (aplikacje tworzone przez małe zespoły, aplikacje dobrze rozumiane):

$$\mathbf{PM = 2.4 (KDSI)^{1.05} * M}$$

**Średniej złożoności** projekt (bardziej złożone projekty, zespół ma niewielkie doświadczenie w dziedzinie problemu)

$$\mathbf{PM = 3.0 (KDSI)^{1.12} * M}$$

**Złożone** projekty:

$$\mathbf{PM = 3.6 (KDSI)^{1.20} * M}$$

# modelu podstawowym COCOMO

---

**M=1** korzysta się tylko z szacowanej wielkości kodu

**KDSI** (kilo delivered instructions) liczba tysięcy dostarczonych instrukcji źródłowych (bez liczenia komentarzy)

Model podstawowy daje punkt startowy estymacji kosztów.

Istnieją **modele średni i szczegółowy**.

# Model średni COCOMO

---

Dodawane są współczynniki określające:

- niezawodność oprogramowania
- wielkość bazy danych
- ograniczenia pamięci, wykonania
- atrybuty personelu
- użycia narzędzi

**0.7 (zmniejszony wysiłek) - 1.66  
(zwiększyony wysiłek)**



# Atrybuty sugerowane w modelu COCOMO

---

- produktu – niezawodność, wielkość bazy danych, złożoność produktu
- komputera – ograniczenia czasu, pamięci, stabilność platform sprzętowych na których oprogramowanie pracuje
- personelu – doświadczenie ludzi pracujących nad projektem
- projektu – związane z użyciem narzędzi , nowoczesnych technik, terminarz projektu

## Przykład

---

Np. złożony system 128 000 (DSI)

Model podstawowy 1216 osobo/miesiąc

Niezawodność 1.4(wysoka)

Złożoność 1.3

Org.pamięci 1.2

Narzędzia 1.1(mały)

harmonogram 1.23 (przyśp)

3593(osob/mies)

## Przykład 2

---

Niezawodność	0.75(niska)
Złożoność	0.7
Ograniczenia pamięci	1
Narzędzia	0.9(wysoki)
harmonogram	1(normalny)
	575(osob/mies.)

# Kalibrowanie modelu

---

Porównywanie kosztów  
oszacowanych z aktualnymi  
daje współczynnik skalowania  
określający wpływ warunków  
lokalnych.

# Model COCOMO określający czas trwania projektu

---

**Proste** projekty      **TDEV = 2.5 (PM) <sup>0.38</sup>**

**Średnie** projekty      **TDEV = 2.5 (PM) <sup>0.35</sup>**

**Złożone** projekty      **TDEV = 2.5 (PM) <sup>0.32</sup>**

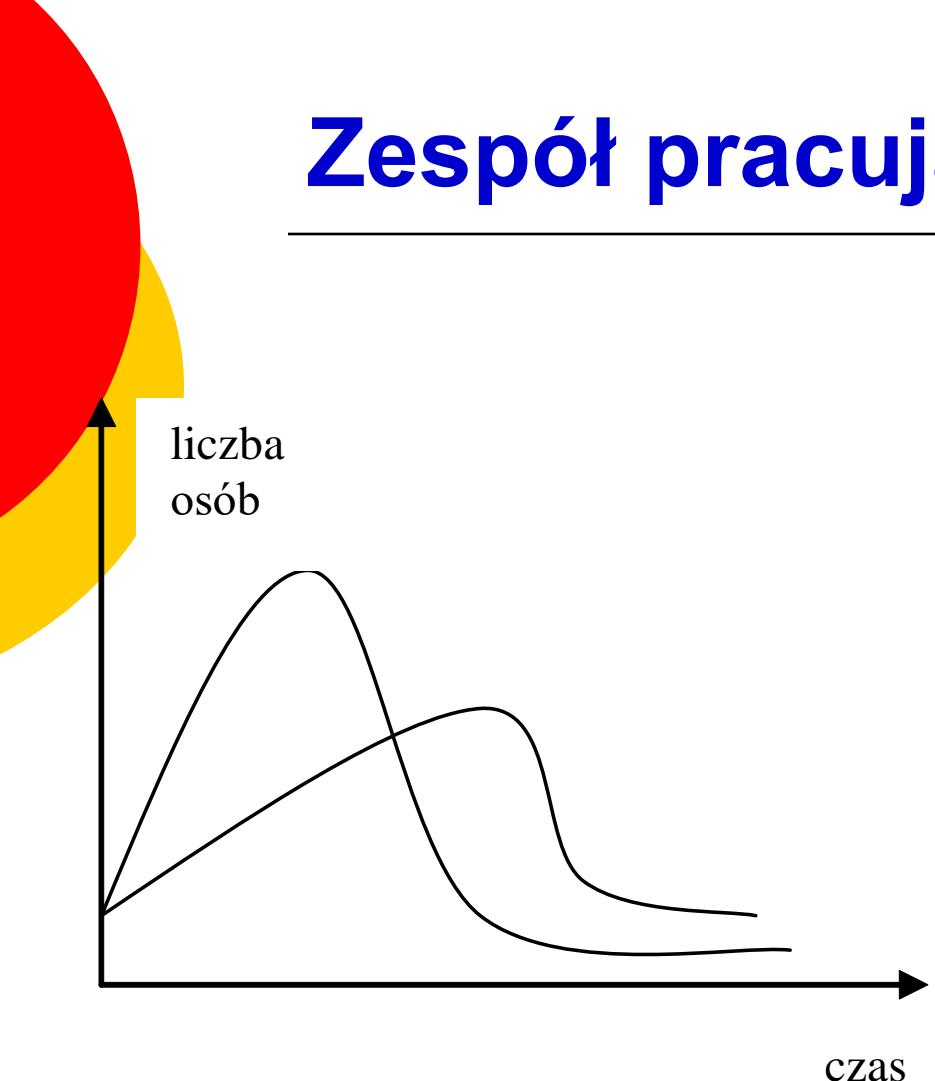
Np. 32000 przewidziano DSI

$$PM = 2.4 (32)^{1.05} = 91 \text{ p.m}$$

$$TDEV = 2.5 (91)^{0.38} = 14 \text{ miesięcy}$$

# Zespół pracujący nad projektem

---



Krzywa Rayleigh'a  
prawdopodobieństwa  
Mała liczba osób zatrudniona  
na początku projektu –  
specyfikacja, planowanie.  
Po implementacji i  
testowaniu jednostek  
spada do 2-3 osób  
dostarczających produkt.

# Model COCOMO 2

---

W modelu COCOMO 2 wzięto pod uwagę różne podejścia do produkcji oprogramowania. Poziomy modelu są związane z czynnościami procesu produkcji oprogramowania. Model COCOMO 2 proponuje trzy poziomy:

1. wczesnego prototypowania,
2. wczesnego projektowania,
3. post-architektoniczny.

# punkty obiektowe (OP)

---

**punkty obiektowe (OP)** - Banker i inni w 1992 jako alternatywa dla punktów funkcyjnych.

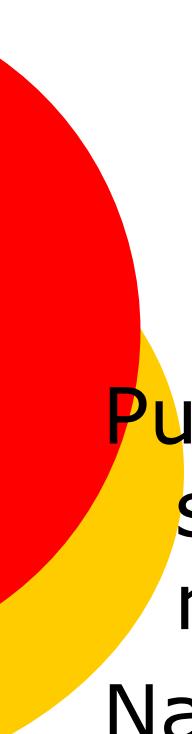
Punkty obiektowe nie są klasami z projektu

Na liczbę punktów obiektowych składa się:

- **Liczba wyświetlanych ekranów**, proste ekrany - 1, bardziej złożone - 2, 3 .
- **Liczba tworzonych raportów**, raporty proste - 2, dla średnie -5, trudne - do 8 dla raportów,
- **Liczba modułów 3GL**, które należy opracować, by uzupełnić kod 4GL. Każdy moduł -10.

# Wysiłek

---



Punkty obiektowe są podstawą do szacowania wysiłku potrzebnego do realizacji systemu.

Należy uwzględnić procentowe użycie gotowych komponentów (%reuse).

$$\mathbf{PM = (NOP * (1 - %reuse / 100)) / PROD}$$

*NOP* jest liczba punktów obiektowych,  
*PROD* jest produktywnością programisty

# Produktywność w punktach obiektowych

---

Doświadczenia i umiejętności	Bardzo małe	Małe	Przeciętne	Duże	Bardzo duże
Możliwości CASE	Bardzo małe	Małe	Przeciętne	Duże	Bardzo duże
<i>PROD</i> (NOP/miesiąc)	4	7	13	25	50

# Poziom wczesnego projektowania

---

$$\textbf{Wysiłek} = 2,5 * W^s * M$$

**W** – wielkość kodu źródłowego podawana jest w tysiącach linii **KDSI** (szacowanie liczby punktów funkcyjnych i przeliczeniu na linie kodu). Takie szacowania dotyczą kodu pisanej „ręcznie”.

Wykładnik **s** może mieć wartość z przedziału **1,1 do 1,24** zależnie od charakteru projektu, jego nowatorstwa, zespołu tworzącego oprogramowanie, firmy itp.

# **M= RCPX\*RUSE\*PDIF\***

---

# **PERS\*PREX\*SCED\*FCIL**

Współczynniki atrybutów produktu i przedsięwzięcia:

- RCPX - niezawodność i złożoność produktu
- RUSE – użycie wielokrotne
- PDIF – trudności platformy
- PERS – możliwości personelu
- PREX – doświadczenie personelu
- SCED – harmonogramu
- FCIL – udogodnienia pomocnicze.

# Wysiłek związany z produkcją

---

$$PM = 2,5 * W^s * M + PMm$$

Gdzie:

$$PMm = (AKDSI * (AT/100)) / ATPROD$$

a :

*AKDSI* – liczba automatycznie wygenerowanych linii kodu źródłowego,

*AT* – wyrażony w procentach odsetek kodu całkowitego, który został wygenerowany automatycznie,

*ATPROD* – poziom produktywności dla tworzenia kodu.

## poziom postarchitektoniczny

---

Używa się tych samych formuł, co poziom wczesnego projektowania.

Szacowania powinny być dokładniejsze toteż używa się aż **17 atrybutów**.

W tej fazie szacuje się liczbę wierszy kodu, które muszą być zmodyfikowane, aby dostosować je do zmian wymagań systemowych.

Bierze się pod uwagę także możliwość wielokrotnego użycia kodu. Wielokrotne użycie kodu wiąże się z pracą związaną ze znalezieniem komponentów, zrozumieniem ich interfejsów oraz z modyfikacjami kodu (by dostosować go do komponentów).

# Wpływ użycia wielokrotnego na wielkość kodu

---

$$\text{ESLOC} = \text{ASLOC} * (\text{AA} + \text{SU} + 0.4\text{DM} + 0.3\text{CM} + 0.3 \text{IM}) / 100$$

Gdzie:

*ESLOC* – równoważna liczba wierszy nowego kodu

*ASLOC* - liczba wierszy kodu użycia wielokrotnego

*DM* – procenty modyfikowanego projektu

*CM* - procenty modyfikowanego kodu

*IM* - procenty pierwotnej pracy integracyjnej wymaganej przy integracji kodu wielokrotnego

*SU* – określa koszt zrozumienia oprogramowania, przyjmuje wartości z zakresu 10 – dobrze napisany kod obiektowy do 50 – dla złożonego kodu.

*AA* – odzwierciedla początkowy koszt ustalenia, czy może być użyte oprogramowanie wielokrotne, przyjmuje wartości z zakresu 0 do 8.

# Określenie wykładownika

---



**wykładownik** określa się na podstawie pięciu czynników skali (wartość z zakresu 0 – „bardzo duży” do 5 – „bardzo mały”):

1. doświadczenie firmy z tego typu systemami,
2. elastyczność tworzenia, czyli udział klienta,
3. analiza ryzyka, duża wartość oznacza pełną analizę,
4. zespół zintegrowany, dobrze komunikujący się,
5. dojrzałość procesu w firmie, określa się przez odjęcie poziomu CMM firmy od 5

## Określenie wykładnika -2

---

W celu otrzymania wykładnika wartości powyższych czynników należy

1. zsumować,
2. podzielić przez 100 i
3. dodać do 1.01.

# Przykład –obliczanie wykładnika

---

- po raz pierwszy realizuje tego typu system - wartość 5 (brak doświadczeń),
- brak jest udziału strony klienta – wartość 1 (bardzo duża elastyczność),
- nowy zespół – wartość 3 (przeciętna),
- nie prowadzi się analizy ryzyka (wartość 5 – bardzo mała),
- firma ma poziom 2 CMM, czyli wartość 3.

Suma (5+1+3+5+3) wynosi 17,  
a wykładnik będzie miał wartość 1.18.

# Atrybuty produktu

---

1. RELY – wymagana niezawodność systemu,
2. CPLX – złożoność modułów systemowych,
3. DOCU – zakres wymaganej dokumentacji,
4. DATA – rozmiar użytej bazy danych,
5. RUSE – wymagany odsetek komponentów użycia wielokrotnego

# Atrybuty komputera

---

- 6. TIME – ograniczenia na czas działania
- 7. PVOL – płynność platformy tworzenia
- 8. STOR – ograniczenia pamięciowe



## Atrybuty personelu

---

9. ACAP – możliwości analityków systemowych
10. PCON – ciągłość zatrudnienia personelu
11. PEXP – doświadczenia programistów w dziedzinie przedsięwzięcia
12. PCAP – możliwości programistów
13. AEXP – doświadczenia analityków w dziedzinie przedsięwzięcia
14. LTEX- doświadczenie w stosowaniu języków i narzędzi



## Atrybuty przedsięwzięcia

---

15. TOOL – użycie narzędzi programowych
16. SCED – kompresja harmonogramu tworzenia
17. SITE – stopień rozproszenia pracy po różnych ośrodkach i jakość komunikacji między ośrodkami

# Szacowanie czasu trwania projektu

---

$$\mathbf{TDEV = 3 * (PM) (0.33 + 0.2 * (B-1.01))}$$

$PM$  to wysiłek potrzebny do realizacji pracy,  
 $B$  wykładnik, a w modelu wczesnego  
prototypowania  $B=1$ .

można uwzględnić skrócenie lub wydłużenie  
harmonogramu (SCEDPercentage).

$$\mathbf{TDEV = 3 * (PM) (0.33 + 0.2 * (B-1.01)) *}\\ \mathbf{SCEDPercentage/100}$$

Np.

$PM = 60$  osobomiesiący,  $B = 1.17$

$$TDEV = 3 (60)^{0.36} = 13 \text{ miesięcy}$$

# Miary niezawodności oprogramowania

Dr hab. inż. Ilona Bluemke

# Miary niezawodności oprogramowania

- Ewoluowały z miar sprzętowych.
- Błędy **sprzętowe** są często **stałe** – aż do naprawy,
- błędy **oprogramowania** są **przemijające** – pokazują się dla pewnych wejść, system często może pracować nadal po ujawnieniu się błędu.

# [POFOD (probability of failure on demand)]

Prawdopodobieństwo błędu żądanej usługi

- systemy sterowania, safety-critical

np.:

0.002

2 na 1000 żądanych usług może dać błąd

# [ROCOF (rate of failure occurrence)]

Współczynnik pojawienia się błędu,  
częstotliwość nieoczekiwanych  
zachowań systemu

- systemy operacyjne, transakcyjne  
(duże koszty startu systemu)

np.

0.01 - 1 błąd prawdopodobnie pojawi  
się w 100 jednostkach czasu działania  
systemu

# [MTTF (mean time to failure)]

Średni czas między obserwowalnymi błędami

np.

1 błąd na każde 500 jednostek czasu  
 $MTTF=500$  (odwrotność ROCOF)

$MTTF >$  czas trwania transakcji

- systemy o długim czasie transakcji ,  
CAD

# [ AVAIL (availability) ]

## Dostępność

- systemy ciągle pracujące,  
telekomunikacyjne

Miara prawdopodobieństwa dostępności dla  
użycia systemu

np.

0.997 oznacza, że na 1000 jednostek  
czasu system jest dostępny w ciągu 997

# [Jednostki czasu ]

- zegar
- czas procesora (cykl)
- liczba transakcji np. systemy o zmiennym obciążeniu - systemy rezerwacji (noc/dzień)

# Wymagania niezawodnościowe

- Wymagania niezawodnościowe są określone nieformalnie.
- Do pomiarów używa się **testowania statystycznego**.

# TESTOWANIE STATYSTYCZNE

Służy do zmierzenia miar niezawodności (do wykrycia błędów oprogramowania służy defect testing).

Kroki:

1. Określenie **profilu działania systemu** (wzór użycia), można osiągnąć analizując historyczne dane wejściowe, określając ich prawdopodobieństwo występowania. Profil określa jak system jest używany w praktyce.
2. Wybór lub generacja danych wg określonego profilu.
3. Wykonanie testu, zbieranie czasu pracy między obserwowalnymi błędami w odpowiednich jednostkach czasu.
4. Po obserwacji wielu błędów obliczenia miar niezawodności

[

# Testowanie statystyczne

]

Jest trudne do wykonania:

- niepewny profil operacyjny
- wysokie koszty generacji profilu
- statystyczna niepewność (przy wysokiej niezawodności)

# Modele wzrostu niezawodności

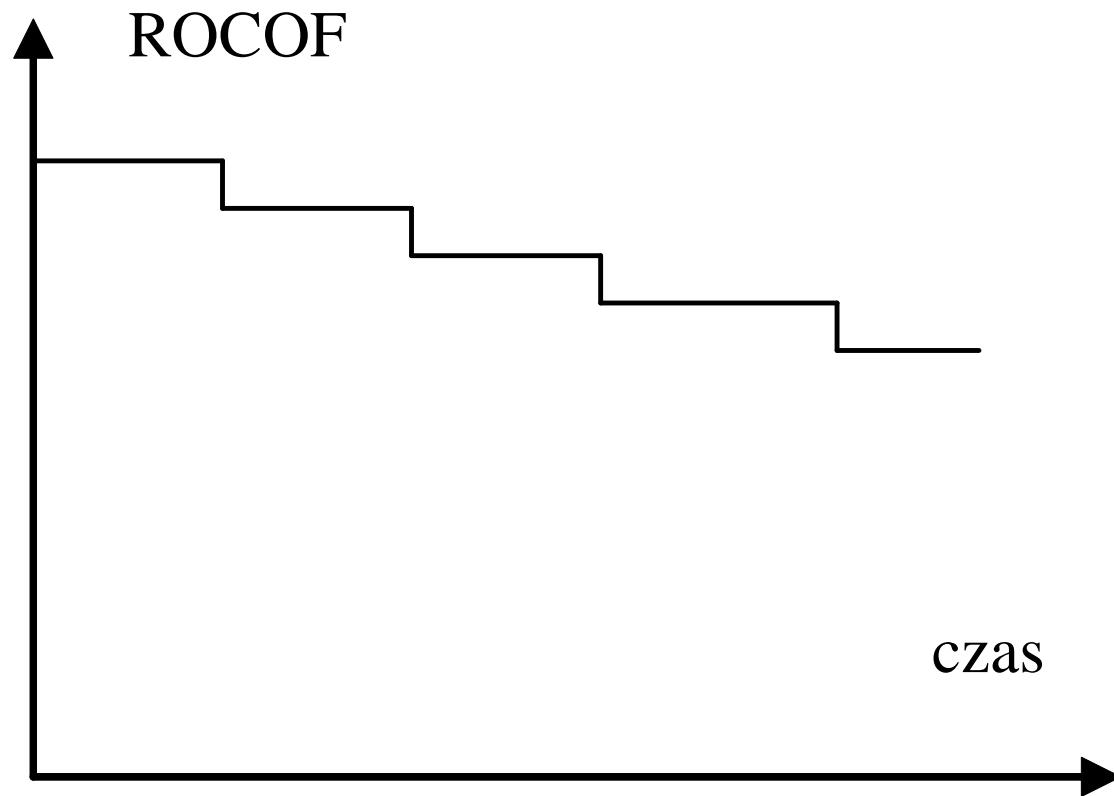
Testowanie powinno być prowadzone aż do osiągnięcia wymaganego poziomu niezawodności. Ponieważ testowanie jest kosztowne powinno być zaprzestane jak tylko stanie się to możliwe

- **Funkcja jednakowego kroku**
- **Funkcja losowego kroku**
- **Modele ciągłe**

# Funkcja jednakowego kroku (Jelinski & Moranda 1972)

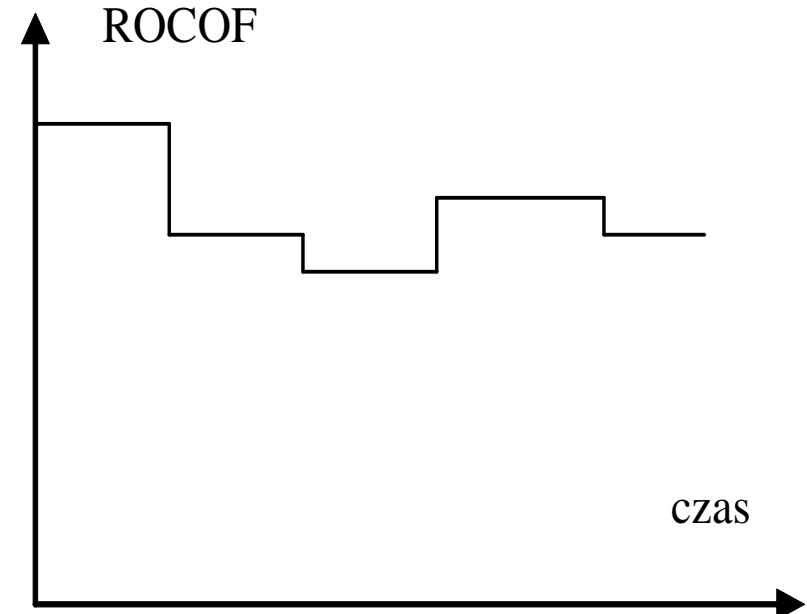
- Niezawodność wzrasta o stałą wartość po wykryciu i poprawieniu każdego błędu.
- Zakłada, że **naprawa błędu jest zawsze poprawna** i nigdy nie powoduje wzrostu liczby błędów w systemie.
- Poprawienie błędu nie zawsze powoduje wzrost niezawodności, mogą być wprowadzone nowe błędy.
- Model zakłada także, że każdy błąd powoduje jednakowy wzrost niezawodności.

# [ Funkcja jednakowego kroku ]



# Funkcja losowego kroku (Littlewood & Verrall 1973)

Modeluje fakt, że wraz z naprawianiem błędów średnie polepszenie niezawodności na naprawę zmniejsza się (może być także negatywny wzrost niezawodności).



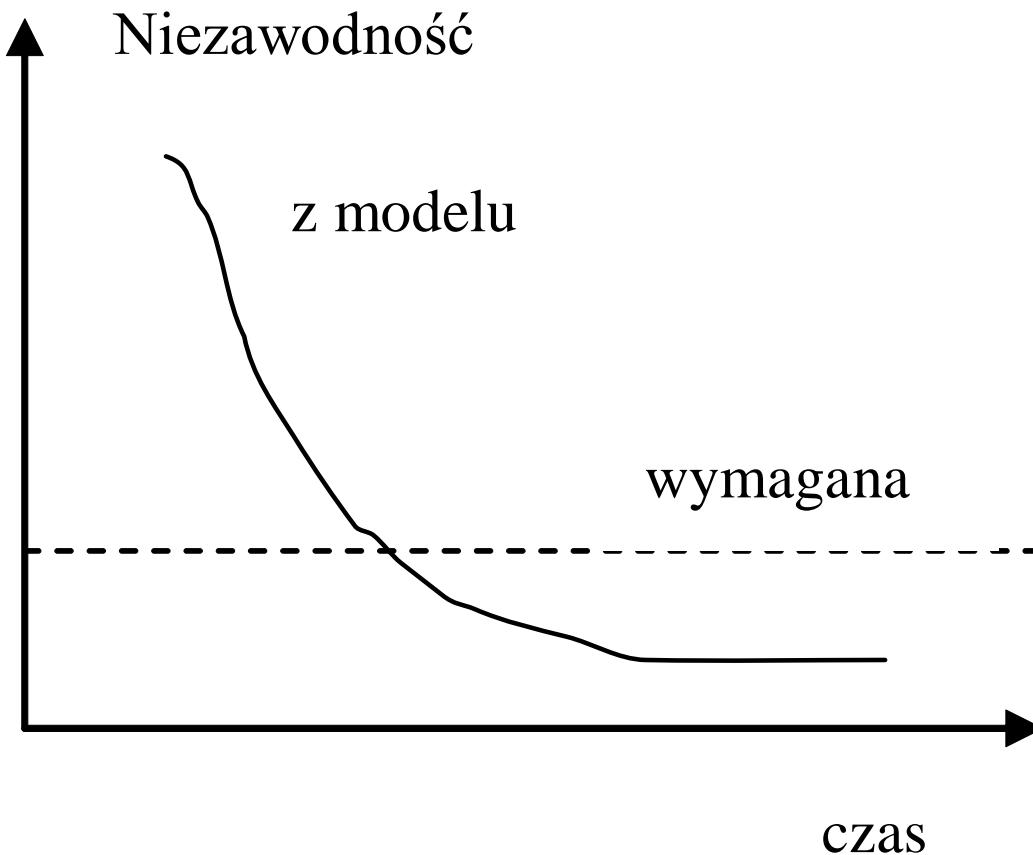
# [ Modele ciągłe ]

- Musa et al. 1987, Abdel – Ghaly 1986,  
...  
■ Modele używane do określenia jak szybko oprogramowanie „poprawia się” w czasie.  
■ Oprogramowanie jest testowane statystycznie, mierzona jest niezawodność, błędy poprawiane, testowanie itp.

[

# Model ciągły

]



# Techniki programowania dla systemów o dużej niezawodności

W systemach o wysokich parametrach niezawodnościowych stosuje się następujące strategie:

- **unikanie błędów** (ang. fault avoidance) technika dostosowana do wszystkich typów systemów, polega na organizacji procesu projektowania i implementacji ukierunkowanej na system „bez błędów”
- **tolerowanie błędów** (ang. fault tolerance) pewne błędy pozostają w systemie, są rozwiązywane tak, by system działał nadal mimo błędu.
- **detekcja błędów** (ang. fault detection) – detekcja błędów przed dostarczeniem systemu. Proces walidacji systemu korzysta z metod statycznych (przeglądy) i dynamicznych (testowanie) wykrywania błędów.

# [ unikanie błędów i detekcja ]

- Zazwyczaj unikanie błędów i detekcja błędów są wystarczające do uzyskania żądanego poziomu niezawodności.
- Proces produkcji oprogramowania powinien być ukierunkowany na **unikanie błędów**, a nie na ich detekcję.
- Oprogramowanie bez błędów „fault-free” oznacza oprogramowanie odpowiadające specyfikacji. Mogą być błędy w specyfikacji, powodujące, że oprogramowanie nie będzie zachowywało się tak, jak by chciał użytkownik.

# Czynniki sprzyjające bezusterkowemu oprogramowaniu

- Precyzyjna specyfikacja (ew. formalna), która jest niesprzecznym opisem tego co ma być wykonane.
- Podejście do projektowania i implementacji bazujące na ukrywaniu informacji, enkapsulacji.
- Właściwa organizacja procesu produkcji – programiści piszą oprogramowanie bez błędów.
- Korzystanie z języków programowania ze sprawdzaniem typów.
- Ograniczenia konstrukcji programowych będących źródłem wielu błędów np. liczby zmiennoprzecinkowe często nieprecyzyjne, wskaźniki, dynamiczny przydział pamięci, współbieżność, rekursja, przerwania są często źródłem błędów.

# System tolerujący uszkodzenia

Kontynuuje działanie mimo pojawienia się błędu. Tolerowanie uszkodzeń jest konieczne w pewnych typach systemów np. kontrola lotów, w systemach, gdzie niesprawność systemu może powodować duże straty ekonomiczne lub ludzkie.

# Aspekty tolerowania uszkodzeń

- **Detekcja uszkodzenia** – system musi wykrywać, że pewna kombinacja dała, lub może dać błąd.
- **Rozmiar zniszczeń** (ang. damage assesment) – wykrycie części systemu, na które błąd miał wpływ.
- **Powrót z błędu** (ang. fault recovery) – przejście systemu do stanu „bezpiecznego”. Możliwe jest:
  - **poprawienie stanu błędного** (ang. forward error recovery) – jest bardzo trudne, wymaga przewidywania stanu systemu
  - **odtworzenie stanu** systemu (ang. backward error recovery).

# Aspekty tolerowania uszkodzeń -2

- **Naprawienie błędu** (ang. fault repair) – modyfikacja systemu.
- Błędy oprogramowania są często przemijające i w wielu sytuacjach naprawa nie jest konieczna. Jeżeli błąd nie jest przemijający to powinna być zainstalowana nowa wersja systemu. Dla systemów ciągle pracujących powinno to być wykonywane dynamicznie.

# [podejście redundancyjne]

## N-wersji oprogramowania.

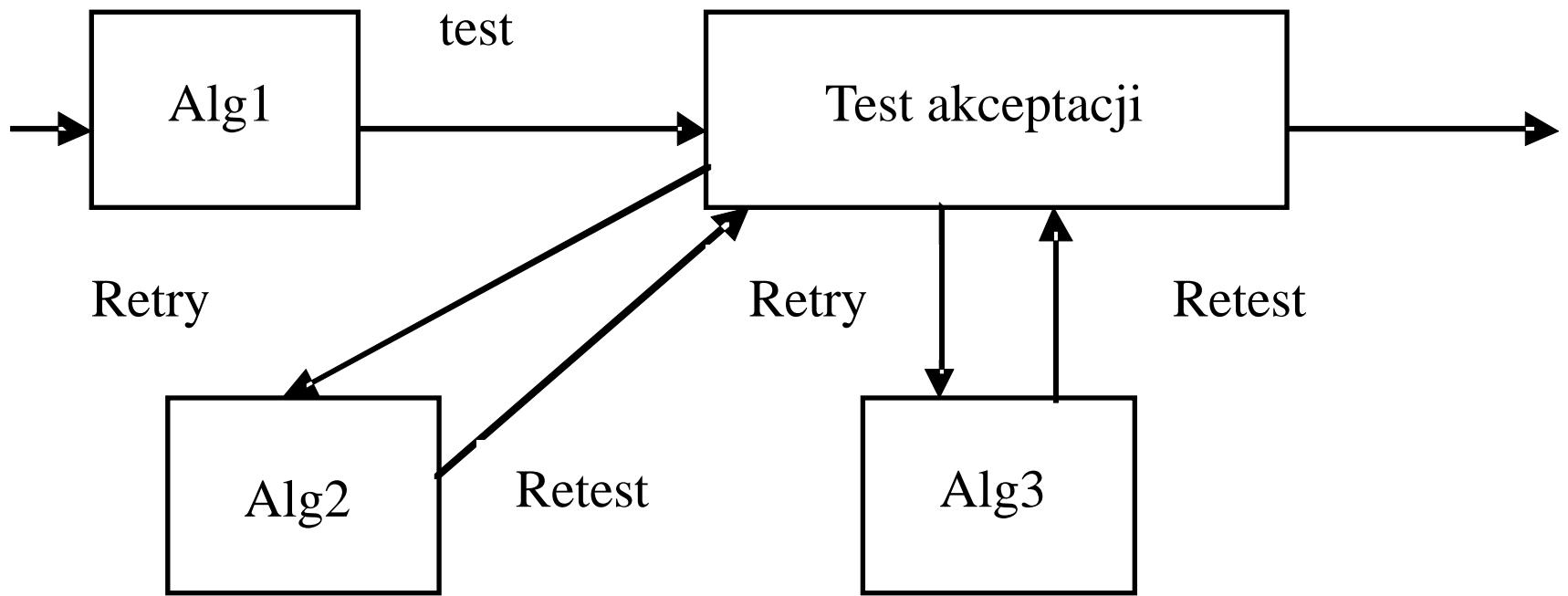
- Dla tej samej specyfikacji różne zespoły produkują oprogramowanie.
- Wersje są wykonywane równolegle.
- Wyjścia są porównywane w systemach głosujących (Avizienis 1985, 1995).
- Wyjście niespójne jest odrzucone.

W podejściu tym zakłada się, że ludzie nie popełnią tego samego błędu projektowego lub implementacyjnego, a to okazało się nieprawdą np. niejasności w specyfikacji mogą być zinterpretowane w ten sam sposób przez różne grupy ludzi.

# [bloki rezerwowe (ang. recovery blocks).]

- Każdy komponent programu zawiera test sprawdzający, czy komponent pracował poprawnie.
- Zawiera także kod pozwalający systemowi na odtworzenie i powtórzenie innego bloku kodu jeśli test wykrył błąd.
- Wykonanie bloków rezerwowych jest sekwencyjne.
- Bloki rezerwowe często są napisane w innych językach programowania, korzystają z różnych algorytmów. Autorami tego rozwiązania byli Randell(1975) Randell & Xu (1995).

# [ Recovery blocks ]



# [ sytuacje wyjątkowe ]

- W niektórych językach programowania np. C++ mamy możliwość **obsługi sytuacji wyjątkowych** (ang. exception handling), czyli określenia kodu obsługi sytuacji wyjątkowej.
- Sytuacje wyjątkowe pozwalają na wykrywanie pewnych błędów wykonania (badanie czy wartości nie przekraczają dozwolonych zakresów, czy zachowane są relacje między zmiennymi).

# programowanie defensywne [ ang. defence programming ]

- Programista zakłada, że w programie mogą wystąpić błędy i niespójności.
- Włącza kod redundancyjny do sprawdzania stanu systemu i powrotu do stanu właściwego.

Np. Programista opracowuje procedury współpracy ze stosem, *push* – włożenie elementu na stos i *pop* – zdjęcie elementu ze stosu. Programista w procedurze *pop* włącza kod sprawdzający, czy stos nie jest pusty.

# [prewencja błędów (failure prevention)].

- Pewne typy błędów są wykrywane przez kompilatory, statycznie.
- W kod programu mogą być włączane asercje, dynamicznie sprawdzające stan zmiennych systemowych.
- Asercje zwalniają wykonanie programu, zajmują dodatkową pamięć ale pozwalają uchronić system przed poważnymi błędami.

# [ocena zniszczeń]

Stosowane techniki umożliwiające ocenę zniszczeń to:

- użycie sum kontrolnych,
- użycie linków redundancyjnych,
- w systemach współbieżnych użycie zegarów kontrolnych (ang. watch dog), resetowanych po zakończeniu wykonania przez proces, jeśli proces nie zakończy się, zegar nie zostanie zresetowany i kontroler zauważy tę sytuację.

# [ Powrót z błędu - fault recovery ]

Przeprowadzenie systemu do stanu „bezpiecznego”, w którym rezultaty błędu będą zminimalizowane, a system może kontynuować pracę, być może w formie zdegradowanej.

- **Poprawienie stanu błędego** (forward error recovery) jest trudne, wymaga przewidywania stanu systemu, stosowania odpowiednich mechanizmów programowania np. kody korekcyjne, zwielokrotnione linki.
- **Odtworzenie stanu systemu** (backward error recovery) jest prostsze, system powraca do zachowanego wcześniej stanu (np. zapisanego na nośniku).

# Podejścia do testowania

Dr hab. inż. Ilona Bluemke

# [ Podejścia do testowania ]

Wybór zależy od typu systemu, różne strategie mogą być stosowane do różnych części systemu

- T-D (top-down) (testowanie zstępujące)
- B-U (bottom-up) (testowanie wstępujące)

# [ Podejście inkrementalne ]

Niezależnie od stosowanej strategii korzystne jest podejście **inkrementalne** do testowania.

A      T1

T2

B      T3

A      T1

T2

B      T3

C      T4

# Testowanie zstępujące T-D [top-down)

- Rozpoczyna się od komponentu najbardziej abstrakcyjnego i posuwa się w głąb.
- Pod-komponenty są reprezentowane jako „**stubs**” – „**namiastki**”, mają ten sam interfejs jak komponent ale ograniczoną funkcjonalność.

# [Wady i zalety testowania T-D ]

## Zalety:

- Błędy projektowe będą szybko wykryte, testowanie przebiega równolegle z rozwojem programu.
- Pracujący system dostępny we wczesnej fazie rozwoju.

## Wady:

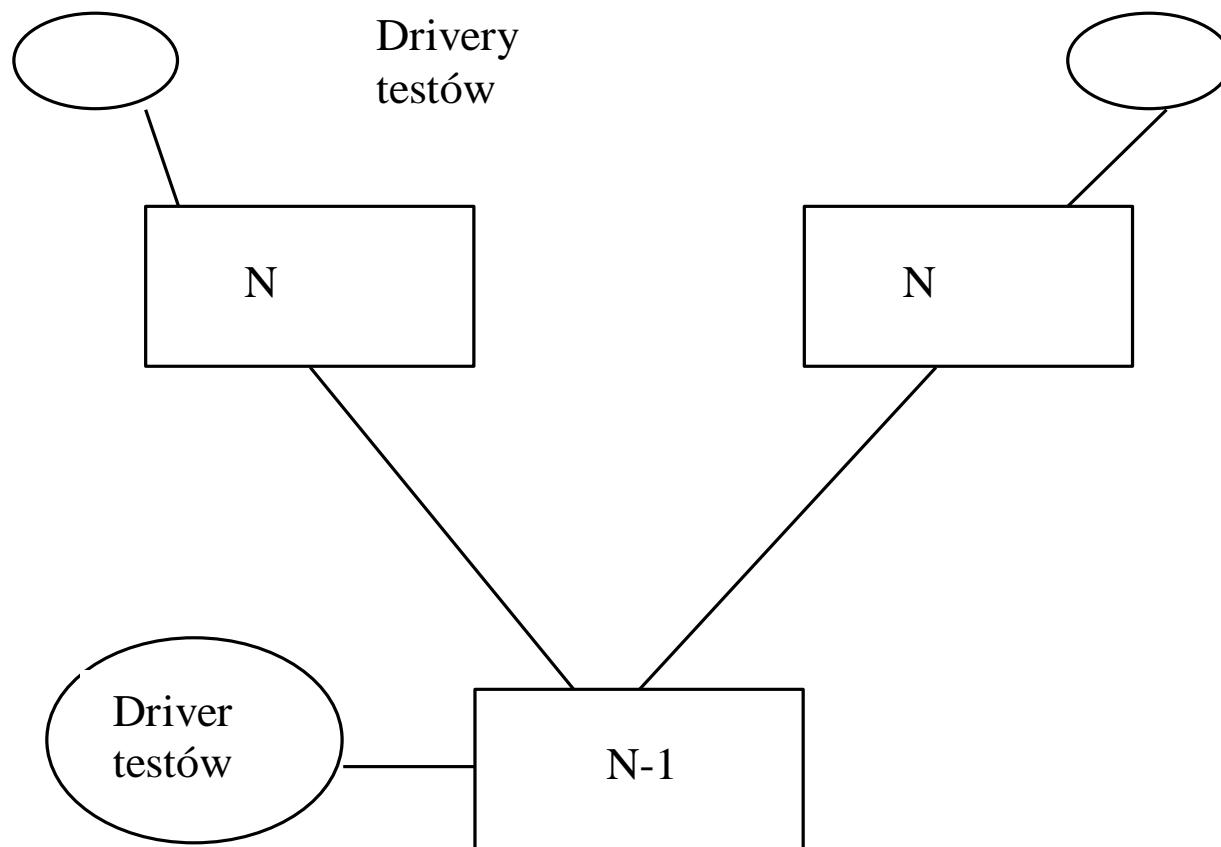
- Symulacje niższych warstw są często trudne do realizacji

# Testowanie wstępujące B-U (bottom-up)

Rozpoczyna się od testowania komponentów (modułów) fundamentalnych i posuwa w górę aż do finalnego.

Konieczne są **drivery testów** symulujące otoczenie komponentu przy wykonywaniu testów.

# [Testowanie B-U



# [ Wady i zalety testowania B-U ]

## Wady:

- Wykrycie błędów w architekturze nastąpi późno, koszt ich usunięcia będzie wysoki.

## Zalety:

- Podejście krytykowane ale używane do testowania komponentów niskiego poziomu. Może być użyte w systemach obiektowych. Indywidualne obiekty mogą być testowane z własnymi driverami testów, potem integrowane, testowany obiekt zbiorczy itp.

# Testowanie stresowe (stress testing)

- Pewne typy systemów są zaprojektowane do „radzenia sobie” z określonymi obciążeniami np. 100 transakcji/sek.
- Specjalne typy testów powinny być zaprojektowane by sprawdzić, jak system „radzi sobie” ze zwiększanym obciążeniem.
- Testowanie stresowe jest kontynuowane po przekroczeniu planowanego obciążenia aż do upadku systemu.

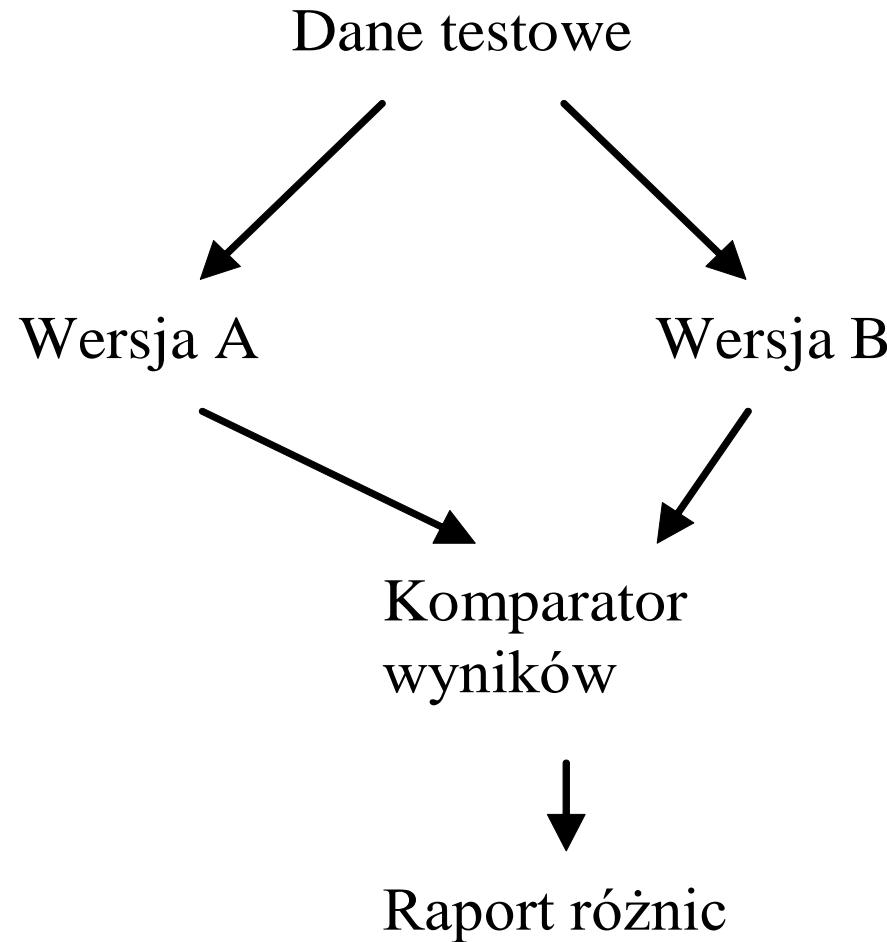
# Funkcje testowania stresującego

- Badanie jak nastąpi upadek systemu, czy dane nie zostały zagubione, „łagodny upadek” po przekroczeniu planowanego obciążenia.
- W systemie „stresowanym” mogą wystąpić defekty, które przy normalnej pracy nie ujawniły się.
- Stosowane w systemach rozproszonych

# Testowanie porównawcze (back-to-back )

- Może być stosowane gdy dostępna jest **więcej niż jedna wersja** systemu (np. różne wersje systemu na różnych komputerach, po modyfikacji systemu do sprawdzenia czy zachowano funkcjonalność).

# Testowanie porównawcze

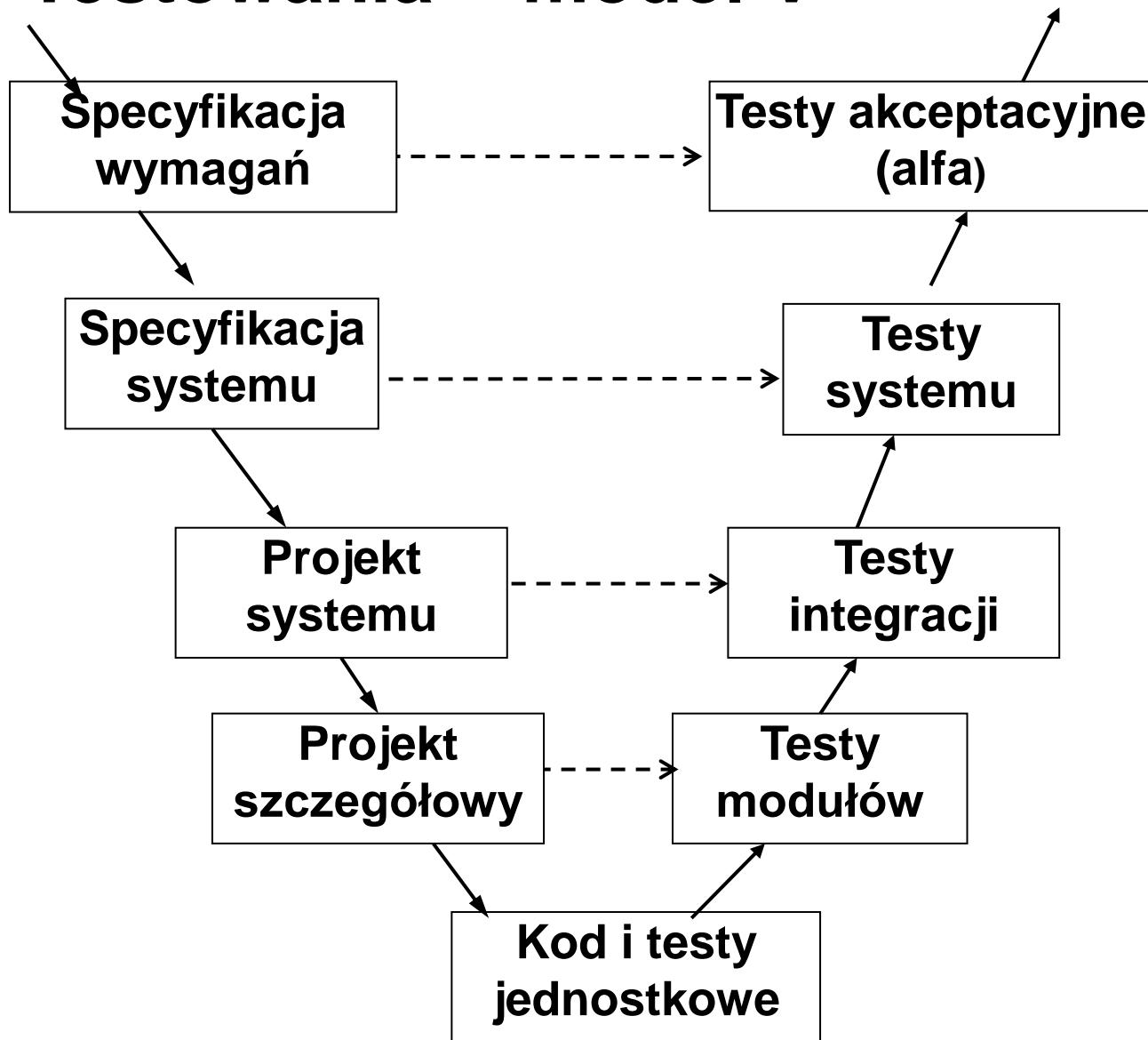




# **TESTOWANIE UKIERUNKOWANE NA WYSZUKIWANIE DEFEKTÓW W PROGRAMIE (DEFECT TESTING)**

**Dr hab. inż. Ilona Bluemke**

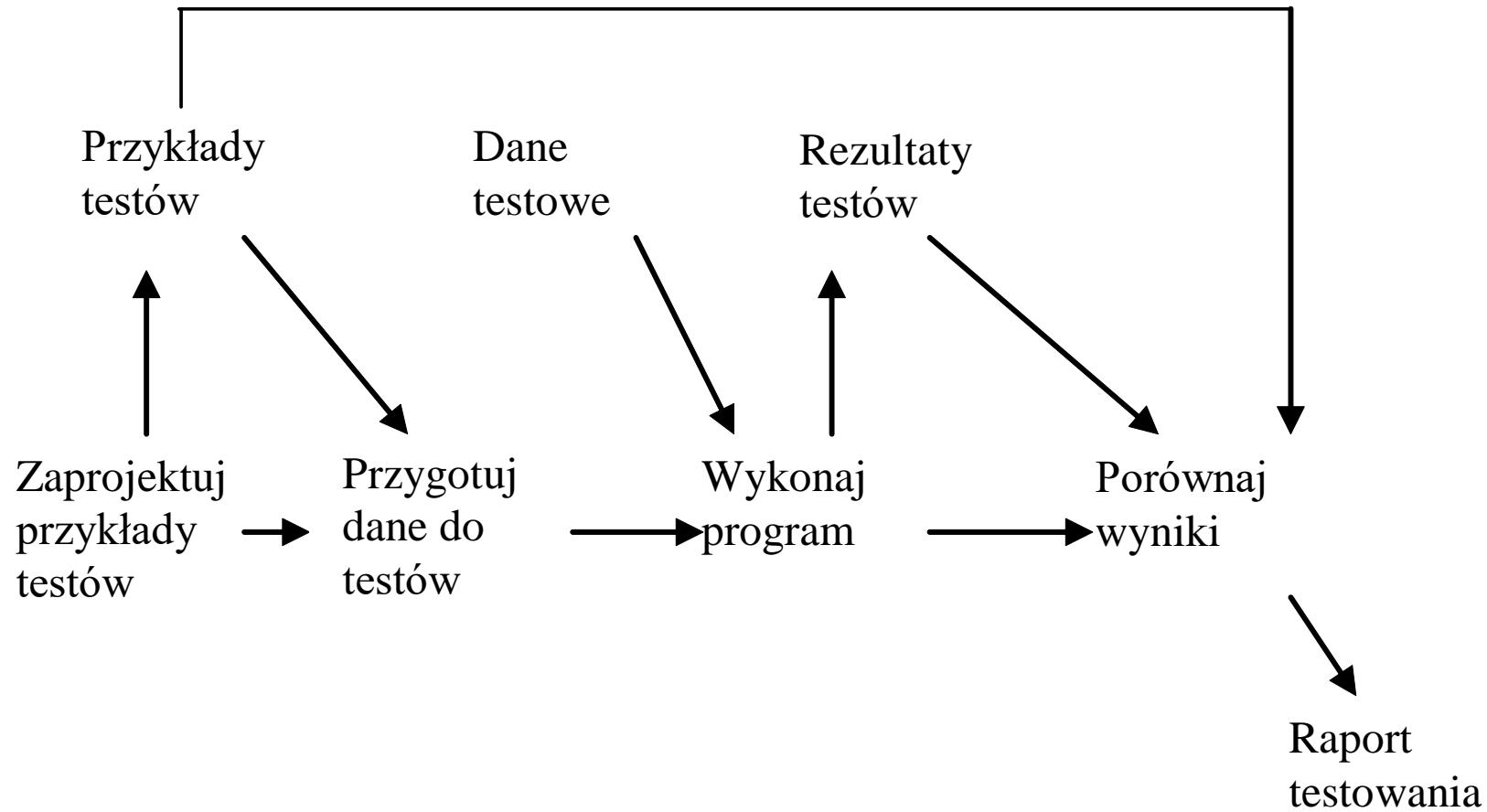
# Fazy Testowania – model V



# Cel testowania

- Celem „**defect testing**” jest ujawnienie defektów systemu.
- Celem **testowania walidacyjnego** jest pokazanie, że system spełnia specyfikację (testy akceptacyjne).
- **Dobry test** (defect test) to taki, który wykrywa błąd oprogramowania. Jeśli testy nie wykryły błędów nie znaczy to, że program jest poprawny, lecz że wykonano testy, które nie wykryły defektów.

# Model testowania



# Test case – przypadek testowy

- określenie funkcji testu,
- specyfikacja WE
- specyfikacja WY

# Test case'y należy zaprojektować

- Testowanie **wyczerpujące** (exhaustive)

Wykonanie każdej instrukcji w programie,  
przejście przez każdą możliwą ścieżkę –  
**praktycznie niemożliwe.**

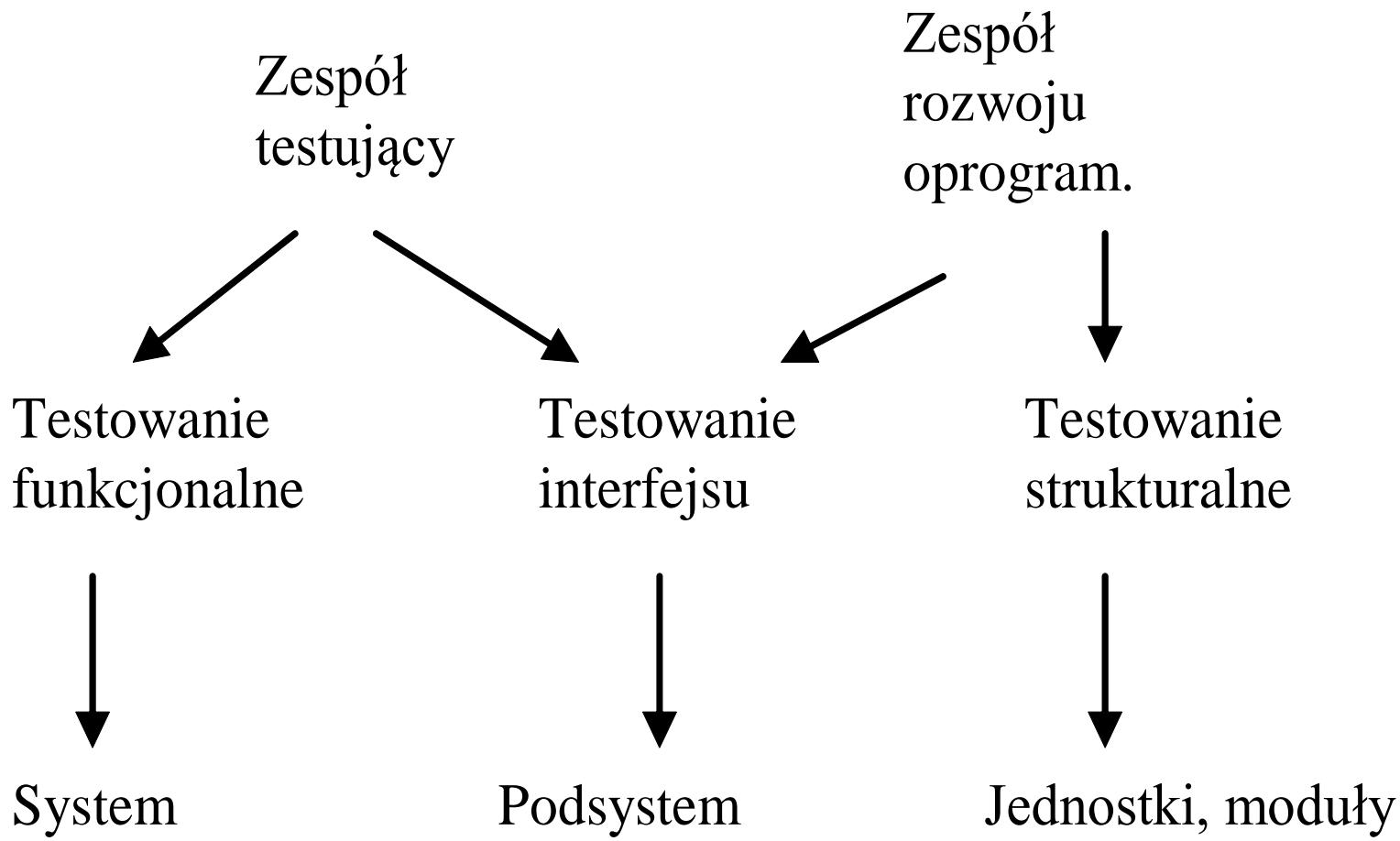
- Testowanie powinno się opierać o  
podzbiór możliwych przypadków.

# Wskazówki – Petchenik

- Testowanie **możliwości systemu** jest ważniejsze od testowania jego komponentów. Przypadki powinny być tak dobrane by zidentyfikować błędy wstrzymujące, uniemożliwiające pracę użytkownika np. utraty danych
- Testowanie **starych możliwości** jest ważniejsze niż testowanie nowych, użytkownicy pracują wg starych nawyków.
- Testowanie **typowych sytuacji** jest ważniejsze niż sytuacji brzegowych

# Podejścia do „defect testing”

- **Funkcjonalne (black box)** – testy wyprowadzone ze specyfikacji
- **Strukturalne (white box)** – testy wyprowadzone na podstawie znajomości struktury programu
- **Testowanie interfejsów** - testy wyprowadzone ze specyfikacji i na podstawie znajomości wewnętrznych interfejsów



# Efektywność testowania

*liczba wykrytych błędów / jedn. czasu*

Basili & Selby (1987) eksperyment porównujący efektywność testowania „czarnych skrzynek” i testowania strukturalnego.

- Testowanie „czarnych skrzynek” okazało się efektywniejsze.

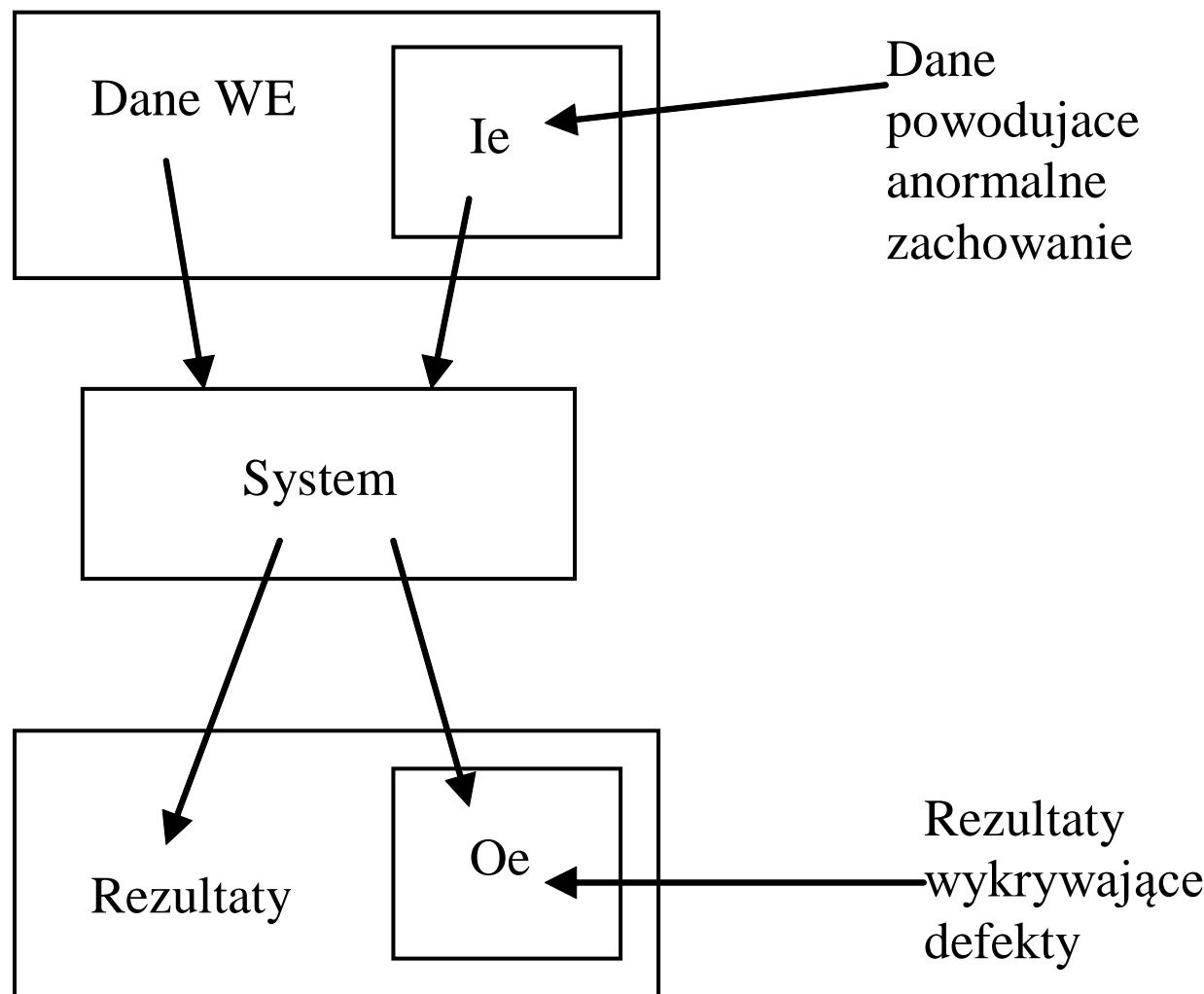
Badano także efektywność inspekcji kodu i testowania

- Statyczne inspekcja kodu okazała się tańsza i b. efektywna. Potwierdziły to eksperymenty Gilb'a & Graham'a 1993 i innych (1996)

# Testowanie funkcjonalne

- Polega na wyprowadzeniu testów na podstawie specyfikacji systemu.
- System traktowany jest jako „**czarna skrzynka**”, której zachowanie może być określone na podstawie wejść i odpowiadających im wyjść.

# Testowanie funkcjonalne



# **Podział na klasy równoważności**

## **Equivalence partitioning (Bezier 1990)**

Podział danych wejściowych na klasy, grupy o wspólnej charakterystyce. Program zachowuje się podobnie dla wszystkich elementów grupy.

Rezultaty programu też można podzielić na pewne grupy (podziały mogą się nakładać).

Cel - znalezienie takich podziałów

Wskazania:

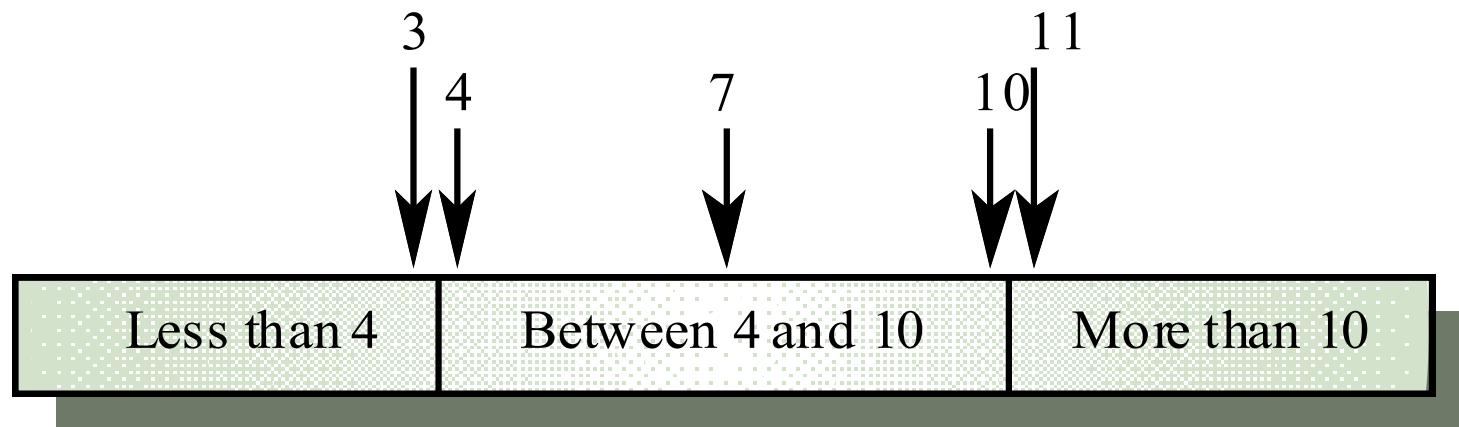
wybierać przykłady testów ze środka (typowe)  
i z brzegów (nietypowe) grupy.

# Testowanie warunków granicznych dziedziny danych

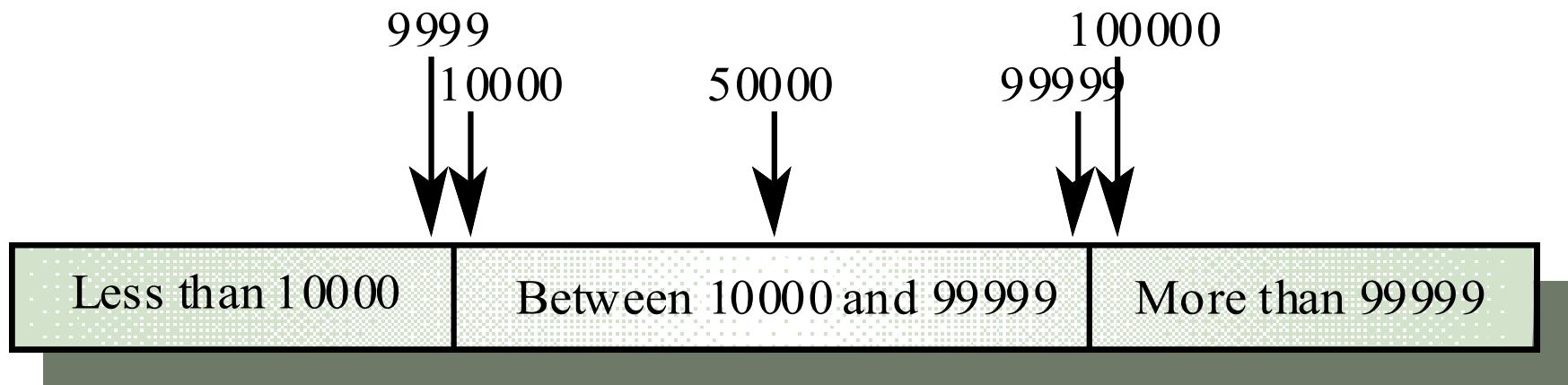
- element pierwszy, środkowy i ostatni,
- zbiór pusty, jedno-elementowy, wielo-elementowy, maksymalny
- element najbliższy i najdalszy,

Wartość domyślna, wartość pusta, spacja, zero, brak danych

# Klasy równoważności - przykład



Number of input values



Input values

# Przykład specyfikacji

```
procedure Search (Key : ELEM ; T: ELEM_ARRAY;  
Found : in out BOOLEAN; L: in out LEM_INDEX) ;
```

## Pre-condition

- the array has at least one element

- T'FIRST <= T'LAST

## Post-condition

- the element is found and is referenced by L

- ( Found and T (L) = Key)

or

- the element is not in the array

- ( **not** Found **and**

- not** (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key ))

# **Podziały na bazie specyfikacji funkcji**

**Grupy podziału ze względu na tablicę  $T$ :**

- tablica pusta (nie zawsze możliwe)
- tablica 1-elementowa,
- tablica wielo-elementowa

**Grupy podziału ze względu na klucz  $Key$ :**

- nie ma klucza w tablicy
- jest klucz w tablicy (w tym podgrupy):
  - jest pierwszym elementem tablicy
  - jest ostatnim elementem tablicy
  - jest wewnętrznym elementem tablicy

# Testy na bazie specyfikacji funkcji

Przykładowe przypadki testowe wg kombinacji podziałów na grupy

Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

# Testowanie strukturalne

- Osoba testująca może analizować kod, korzystać ze struktury komponentu do opracowania testu. Znalezienie algorytmu pozwala na znalezienie dalszych podziałów.
- Metody:
  - Pokrycia kodu
  - Pokrycia danych

# Pokrycie przepływu sterowania

## Pokrycie:

- instrukcji, linii kodu (line coverage)  
co najmniej jednokrotne wykonanie każdej instrukcji dla danego testu (zestawu testów)
- warunków (rozejść decyzyjnych)  
każdy elementarny warunek ma zostać co najmniej raz spełniony i co najmniej raz nie spełniony
- bloków, funkcji
- ścieżek

# Wyznaczanie pokrycia

Narzędzia - **analizatory pokrycia kodu**:

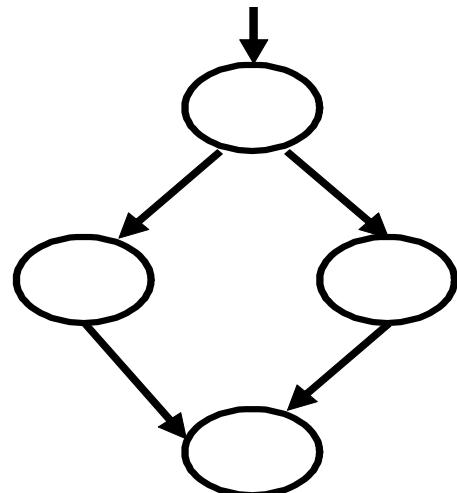
- określają pokrycie dla testu
- sumaryczne pokrycie dla zbioru testów
- wskazują niepokryty kod.

Testowanie **wyczerpujące** (exaustive)  
przejście przez każdą możliwą ścieżkę wykonania  
programu – praktycznie niemożliwe.

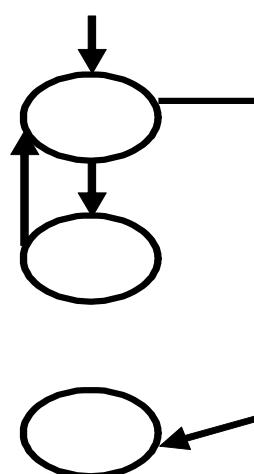
# Graf przepływu sterowania

Węzły - decyzje, krawędzie - przepływ sterowania.  
Można pominąć ciągi instrukcji sekwencyjnych.

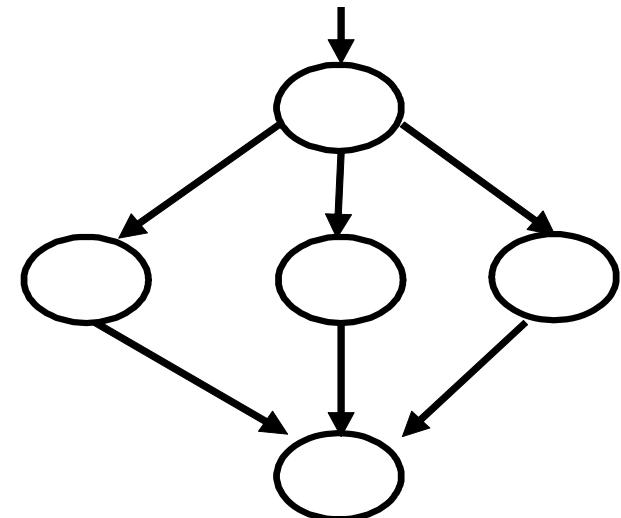
**if – then  
– else**



**loop – while**



**case - of**



# Testowanie ścieżek

**Bazowy zbiór niezależnych ścieżek – minimalny zbiór ścieżek, których liniowa kombinacja generuje każdą możliwą ścieżkę w grafie.**

Niezależna ścieżka przechodzi przez co najmniej jedną nową krawędź grafu przepływu sterowania.

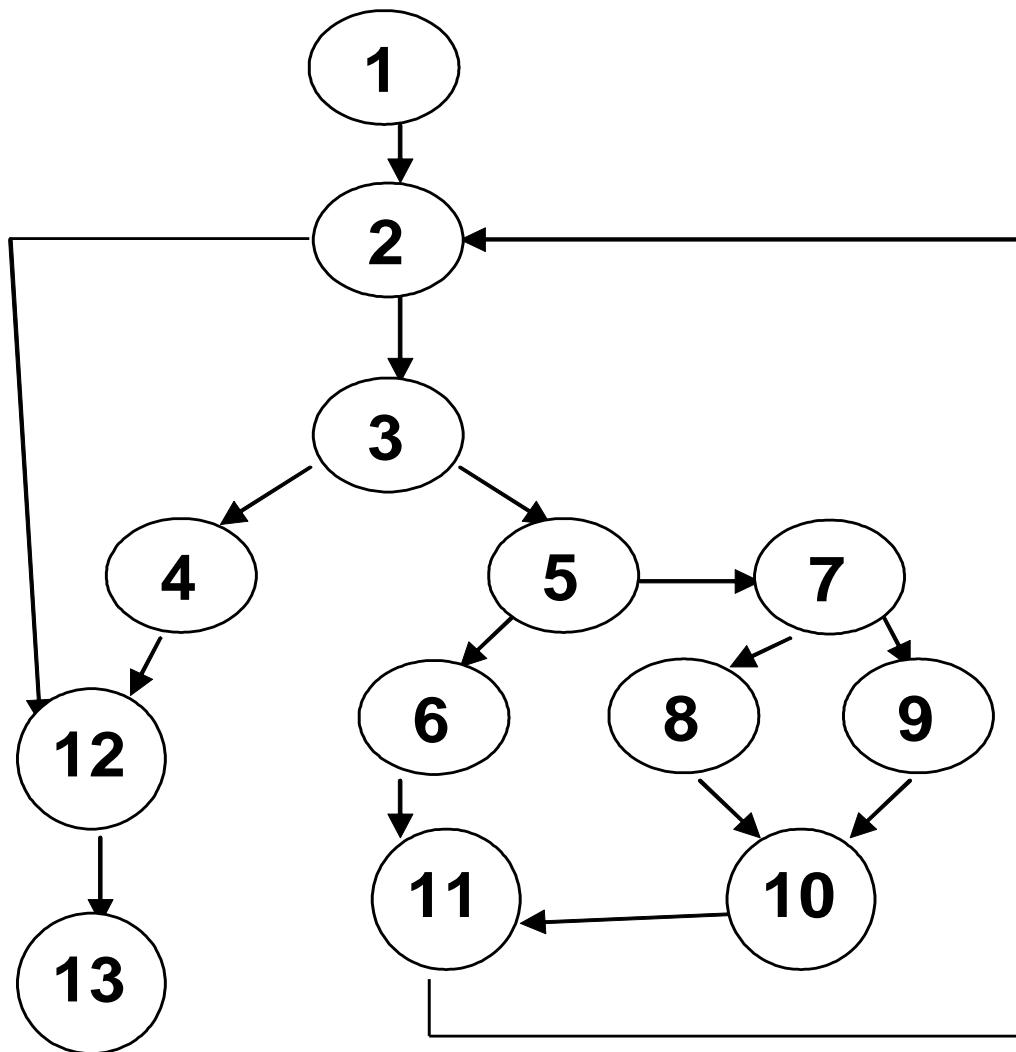
**Liczba niezależnych ścieżek - złożoności cyklicznej (cyklotomatycznej) Mc Cabe z grafu przepływu sterowania.**

**CC(G) = liczba krawędzi – liczba węzłów + 2**

Liczba prostych warunków w programie **+1** (bez goto)

Zapewnia **pokrycie** instrukcji i warunków

# Testowanie ścieżek - przykład



Np.

- 1) 1, 2, 12, 13
- 2) 1, 2, 3, 4, 12, 13
- 3) 1, 2, 3, 5, 6, 11,  
2, 12, 13
- 4) 1, 2, 3, 5, 7, 8,  
10, 11, 2, 12, 13
- 5) 1, 2, 3, 5, 7, 9,  
10, 11, 2, 12, 13

# Wykonanie wszystkich niezależnych ścieżek gwarantuje

- wykonanie co najmniej 1 każdej instrukcji
- wykonanie skoku warunkowego dla prawdy i fałszu

# Liczba niezależnych ścieżek programu

- może być określona po obliczeniu **złożoności cyklotomatyycznej Mc Cabe** (1976) z grafu przepływu sterowania.

$$CC(G) = I.krawędzi - I.węzłów + 2$$

# Testowanie niezależnych ścieżek

- Po obliczeniu liczby niezależnych ścieżek następnym krokiem jest **opracowanie przykładu testu wykonania każdej ścieżki.**
- Testowanie niezależnych ścieżek nie testuje wszystkich możliwych kombinacji przejść w programie. Defekty mogą się pojawić przy określonej kombinacji ścieżek.

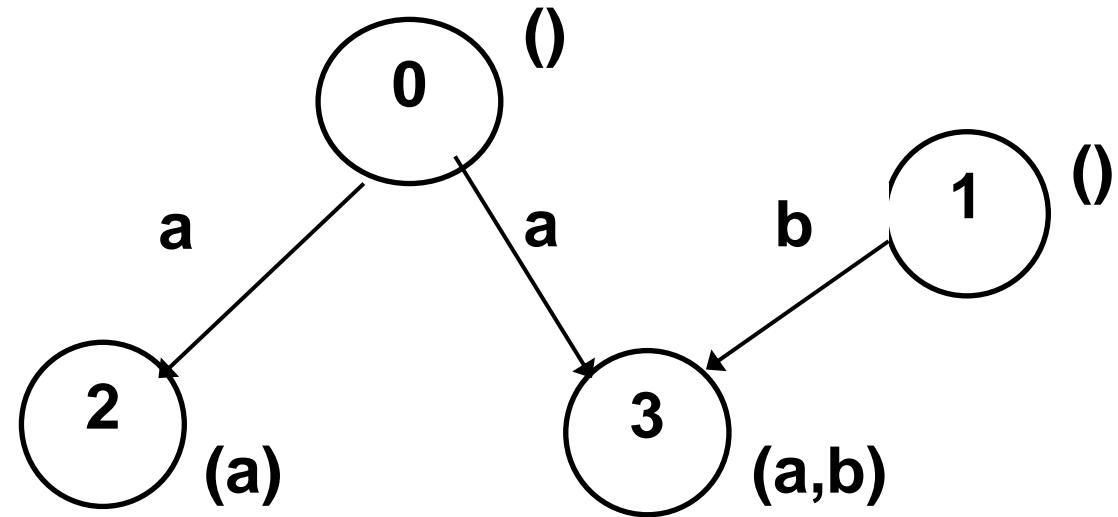
# Pokrycie przepływu danych

Oparte na modelu przepływu danych:  
sekwencja par węzłów pozostających w relacji  
**<definicja(X)i , użycie(X)j>**

Możliwe jest takie wykonanie programu,  
że pomiędzy **podstawieniem** wartości  
zmiennej X w i-tym węźle,  
a jej **użyciem** w j-tym węźle  
nie występuje żadne inne podstawienie zmiennej X.

# Pokrycie przepływu danych - przykład

```
0)    int fun (int a)
     { int b;
1)          cin >> b;
2)          while (a) {
3)              b = b - a;
```



$<0,2>, <1,3>, <0,3>$

# Kryteria testowania pokryć danych

- wykonanie wszystkich par *<definicja-użycie>* dla wszystkich zmiennych  
(all def uses paths coverage)

Np. **<0,2>, <1,3>, <0,3>**

- dla każdej definicji (przypisania) wykonanie co najmniej jednej pary  
Np. **<0,3>, <1,3>**

- wykonanie wszystkich par typu:  
*<definicja, użycie zmiennej w wyrażeniu warunkowym>*

(all-c-uses coverage)

Np. **<0,2>**

# Testowanie z pokryciem przepływu danych

- Po określaniu zbioru par wybrać najkrótsze ścieżki do ich pokrycia.
- Podać testy dla tych ścieżek, jeśli możliwe do wykonania
- Ewentualnie szukać dłuższych ścieżek
- Lub automatycznie sprawdzać pokrycie par dla różnych testów bez wyznaczania ścieżek

# Testowanie obiektowe

## Poziomy

- testowanie operacji (funkcjonalne i strukturalne)
- testowanie obiektów
- testowanie zbiorów (gron) obiektów
- testowanie systemu obiektowego V&V – względem wymagań funkcjonalnych i niefunkcjonalnych

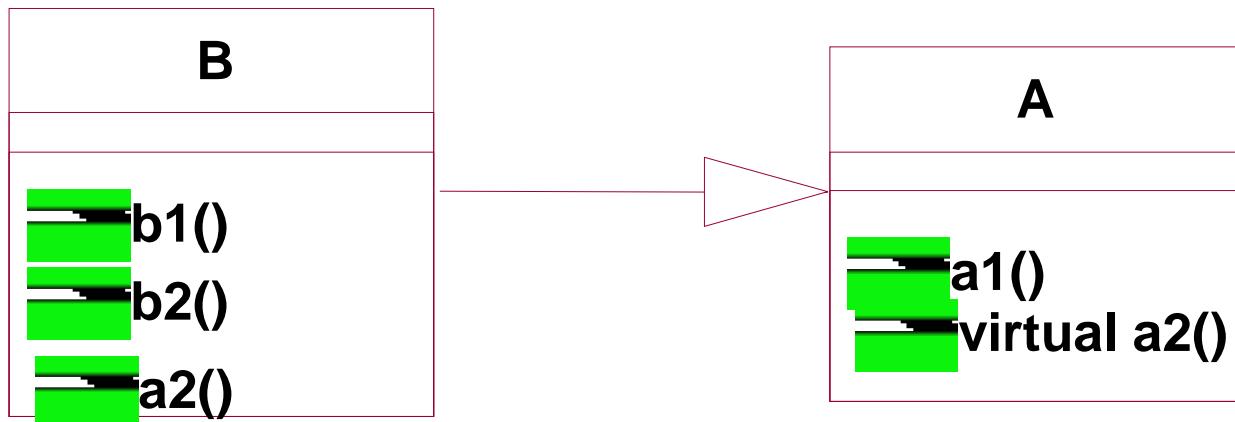
# Testowanie obiektów

- Testowanie w izolacji wszystkich operacji
- Testowanie ciągów wykonań operacji danej klasy
- Ustawienie i użycie wszystkich atrybutów klasy
- Klasy równoważności operacji  
np. inicjalizacja atrybutów, dostęp, modyfikacja
- Użycie obiektu we wszystkich możliwych stanach (ciągi zmian stanów)

# Zbiory obiektów zależnych

- **Testowanie hierarchii klas**
  - operacje odziedziczone,
  - funkcje wirtualne dla obiektów klasy bazowej i potomnych, polimorfizm,
  - dziedziczenie wielopoziomowe,
  - operacje przy dziedziczeniu wielobazowym
- **Testowanie klas zaprzyjaźnionych**

# Testowanie klasy B - przykład



- Testowanie użycia atrybutów z klasy B i dostępnych atrybutów klasy A
- Testowanie strukturalne i funkcjonalne metod.
- Testowanie odziedziczonej metody `a1()` w kontekście obiektów klasy B.
- Testowanie metody `a2()` w kontekście obiektów klasy A i klasy B
- Testowanie użycia obiektów klasy B w możliwych stanach.

# Integracja obiektów

**Integracja obiektów** - testowanie wstępujące lub zstępujące zwykle nieadekwatne

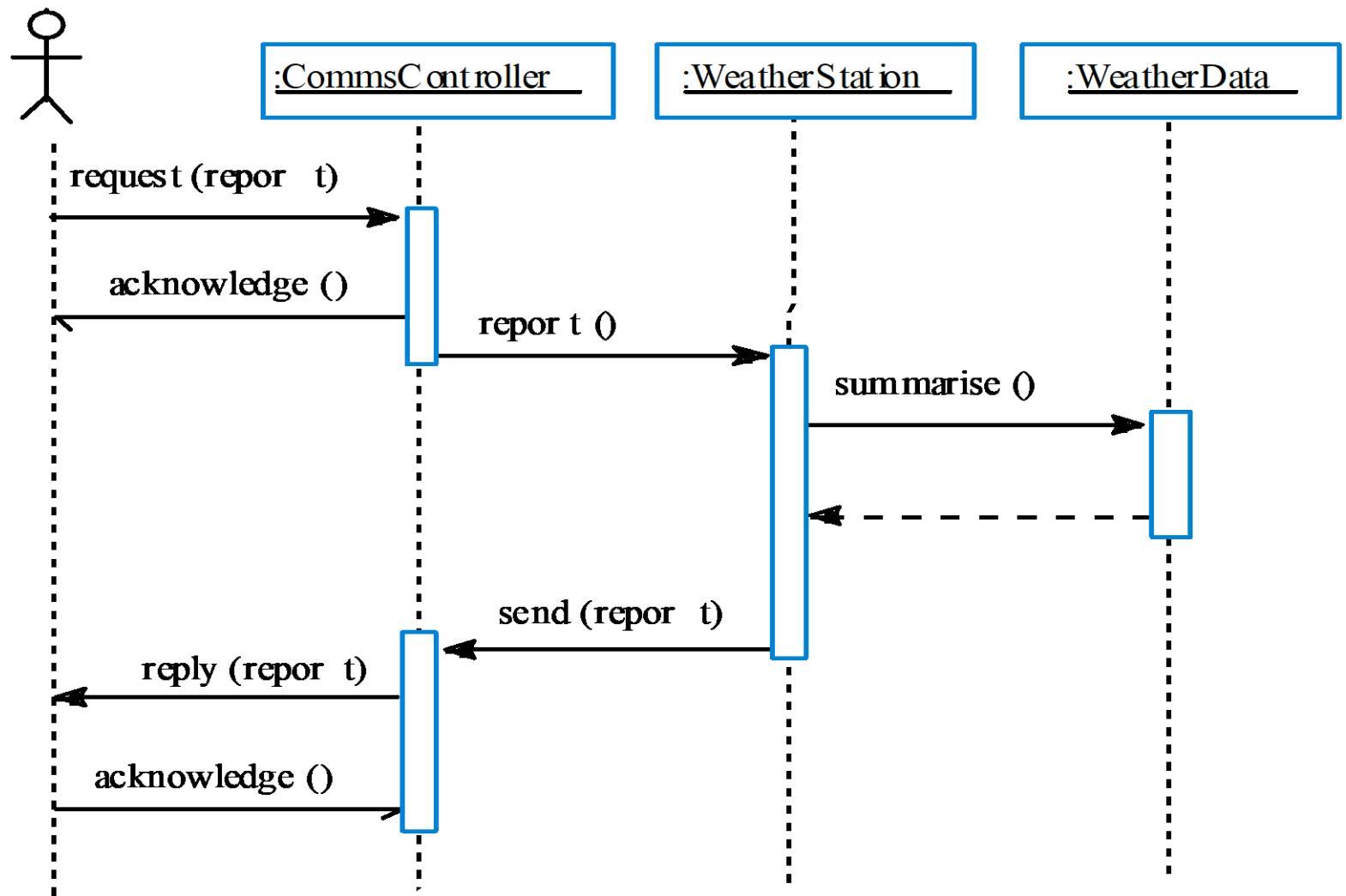
- **Testowanie w oparciu o opis użycia systemu**

przypadki użycia (use case) i definiujące je diagramy sekwencji lub współpracy

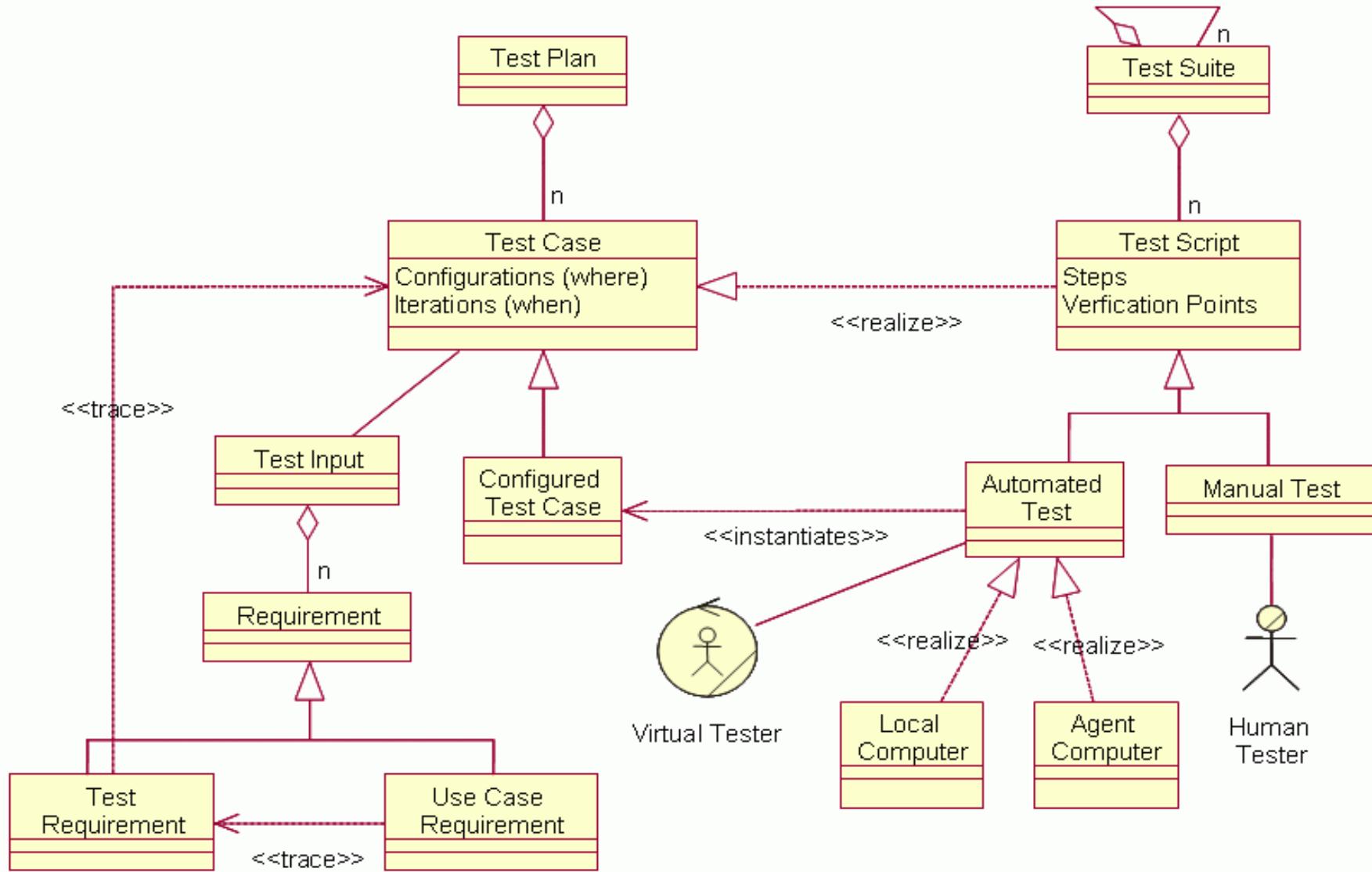
- **Testowanie przepływów (wątków)**  
reakcje na zbiory zdarzeń wejściowych

Podstawą testów zbiorów klas i systemu są modele analityczne i projektowe

# Generacja testów z diagramów sekwencji



# Realizacja testów



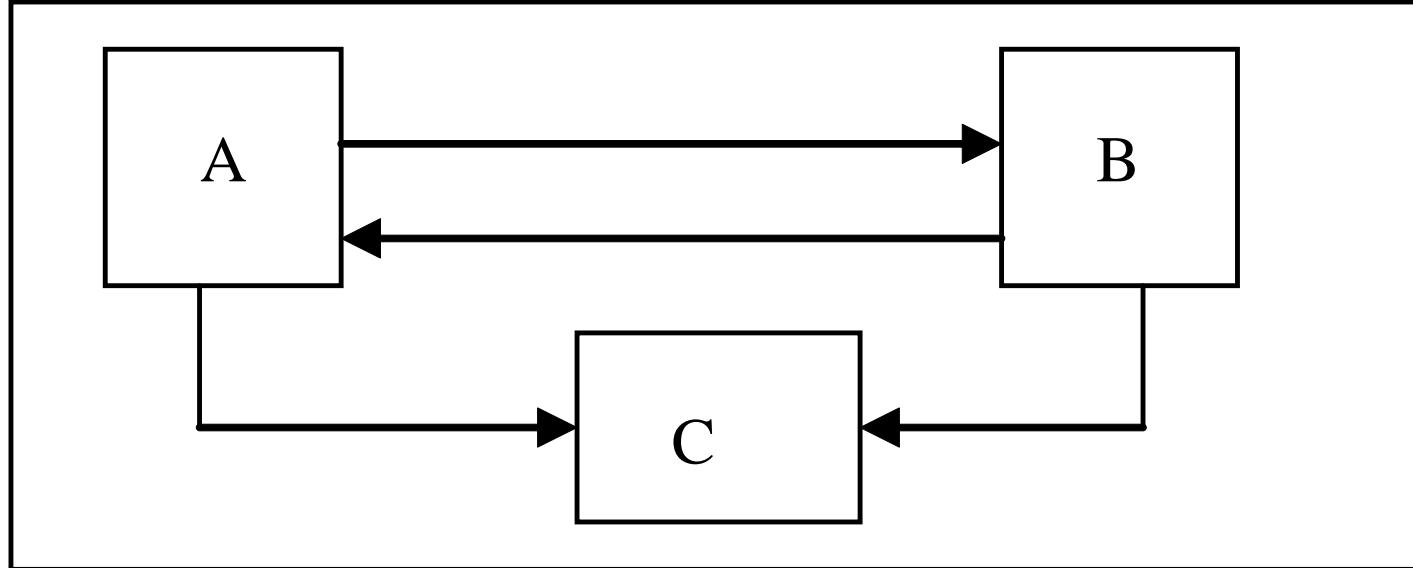
# Narzędzia

- Do testowania jednostkowego np. JUnit, NUnit  
(w TDD test driven development)
- Do generacji skryptów testujących np. z modeli
- Narzędzia przechwytująco/odtwarzające  
(testowanie GUI)
- Do organizacji procesu testowania  
(bazy skryptów testujących, danych wejściowych i wyroczni)
- Do symulacji obciążenia

# Testowanie interfejsów

- Stosowane przy integracji modułów, podsystemów.
- Testowanie ma wykryć błędy w interfejsie lub w założeniach dotyczących interfejsu.
- Ważne dla systemów obiektowych, szczególnie, gdy klasy są „reuse”.

## Przykłady testów



# Typy interfejsów

- parametryczne (przekazywane wskazania na dane, funkcje)
- wspólna pamięć (shared memory)
- proceduralne (podsystem zawiera zbiór procedur i ich wywołania mogą mieć miejsce z innych podsystemów)
- przekazywanie komunikatów (message passing) (podsystem wysyła komunikat żądający usługi innego podsystemu, zwrotny komunikat zawiera rezultat usługi – systemy klient-serwer)

# Klasy błędów interfejsów

- **Użycia** – szczególnie częsty w interfejsach parametrycznych (błędny typ parametru, kolejność, liczba parametrów)
- **Błędnego zrozumienia** – komponent wywołujący błędnie zakłada, jakie ma być zachowanie komponentu wywoływanego (np. że wektor ma być uporządkowany a nie jest)
- **Błędy czasowe** – systemy czasu rzeczywistego, komunikacja poprzez wspólną pamięć lub przekazywanie komunikatów np. Producent i konsument danych pracują z różnymi prędkościami. Konsument może dostać „stare” dane. Dobrze zaprojektowany interfejs powinien uniemożliwić takie sytuacje. Testowanie jest trudne, błędy ujawniają się w pewnych, często nieoczekiwanych sytuacjach.

# Wskazania:

- Zaprojektuj test, gdzie **wartości parametrów są ekstremalne** w swoich zakresach
- Sprawdź interfejs ze **wskazaniami null**
- Interfejs proceduralny – zaprojektuj test, który spowoduje błąd komponentu
- Systemy z przekazywaniem komunikatów – **testowanie stresujące**. Generowanie większej liczby komunikatów niż zakładano może ujawnić problemy czasowe.
- Komunikacja poprzez wspólną pamięć – testy powinny zawierać **różną kolejność dostępu** do pamięci (mogą ujawnić błędy spowodowane założeniem określonej kolejności)

# Zwinne metodyki AGILE

Dr hab. inż. Ilona Bluemke  
Oraz slajdy Ian Sommerville  
„Software Engineering” 10 ed

# LOOP

**Late (późno)**

**Over budget (przekroczony budżet)**

**Overtime (nadgodziny)**

**Poor quality (kiepska jakość)**

# Manifest zwinności (*Agile*)

Luty 2001

**Kent Beck** (karty CRC, xUnit, XP)

**Alistair Cockburn** (przypadki użycia)

**Marin Fowler** (refaktoryzacja, UML Distilled)

**Jim Highsmith** (Adaptive Software Development)

**Ważniejsze jednostki i interakcje niż procesy i narzędzia**

# ISO 9001:2000

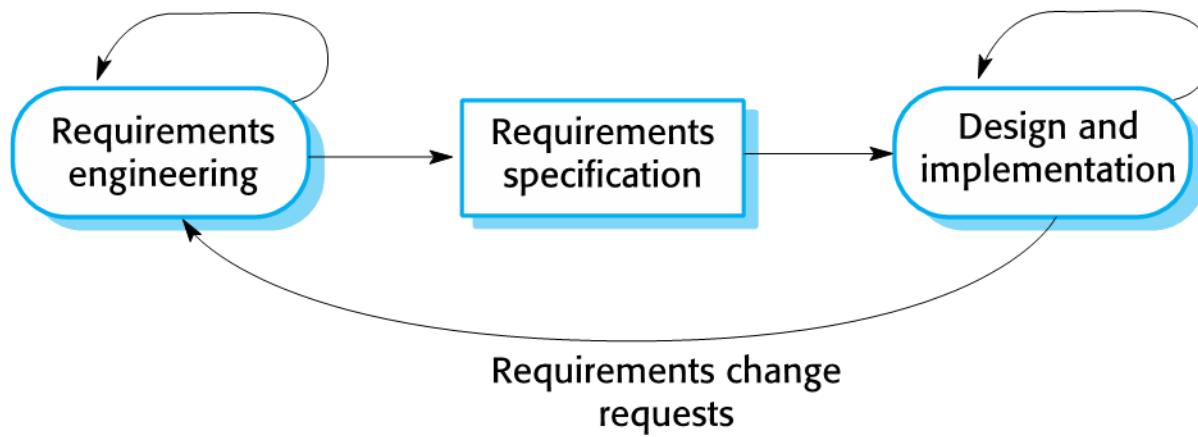
## główe części:

- System zarządzania jakością (dokumentacja)
- Odpowiedzialność kierownictwa
- Zarządzanie zasobami
- Realizacja wyrobu
- Pomiary, analiza i doskonalenie

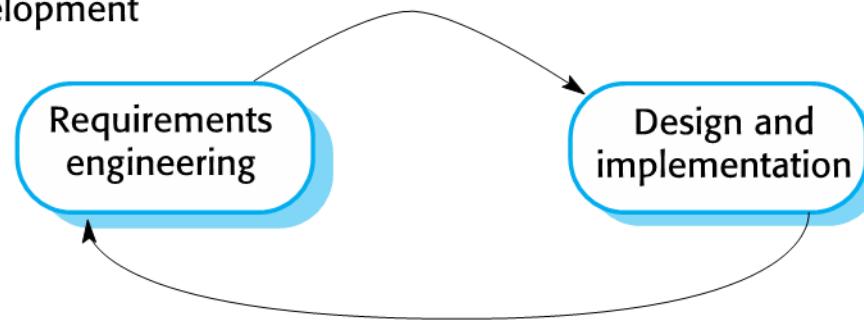
Problem – zbyt dużo dokumentacji, brak elastyczności

# Źródło - Ian Sommerville

Plan-based development



Agile development



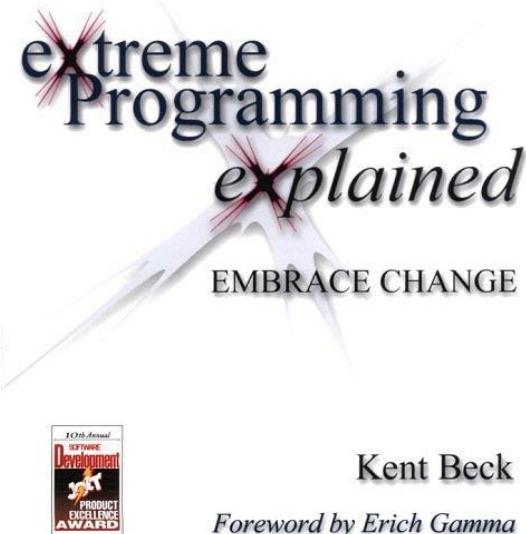
# Manifest zwinności

## Ważniejsze:

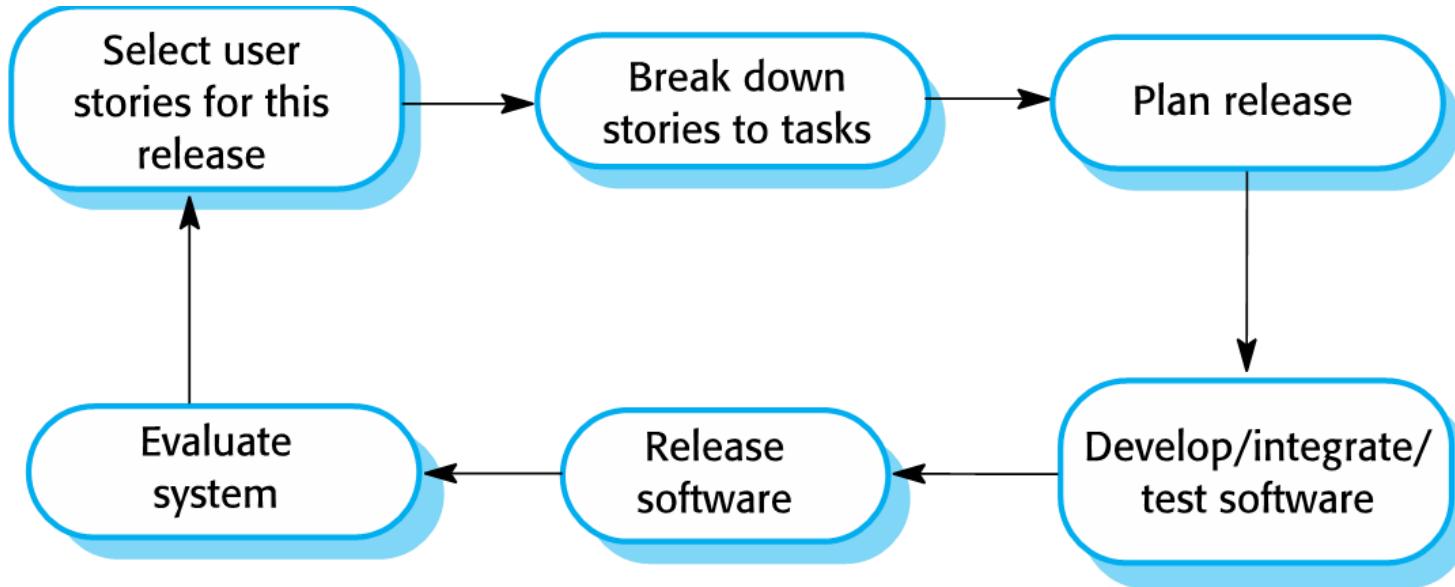
- **Jednostki i interakcje niż procesy i narzędzia**
- **Działające oprogramowanie niż obszerna dokumentacja**
- **Współpraca klienta niż negocjacja kontraktu**
- **Nadążanie za zmianami niż trzymanie się planu**

# Programowanie Ekstremalne (XP)

- lekka (zwinna)
- metodyka tworzenia oprogramowania



# Źródło - Ian Sommerville



# **Wybrane praktyki XP**

- **Klient na miejscu**
- **Krótkie przyrosty i wydania**
- **Najpierw przypadki testowe potem kod**
- **Automatyzacja wykonywania testów**
- **Dokumentacja = Przypadki testowe + kod**
- **Programowanie parami**
- **Małe zespoły**

# Źródło - Ian Sommerville

## Extreme programming practices (a)



Principle or practice	Description
Incremental planning	Requirements are recorded on story cards and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development 'Tasks'. See Figures 3.5 and 3.6.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple design	Enough design is carried out to meet the current requirements and no more.
Test-first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.

# Źródło - Ian Sommerville

## Extreme programming practices (b)



Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of overtime are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site customer	A representative of the end-user of the system (the customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

# Programowanie parami

- 2 osoby przy jednym komputerze
- Pary się zmieniają
- Wymiana wiedzy, informacji, mniejsze ryzyko
- efektywne

# Raport Sackmana, Eriksona i Granta

Różnice w wydajności programowania jak  
**10:1**

Różnice w rozmiarze programu jak  
**5:1**

## Scrum



- ✧ Scrum is an agile method that focuses on managing iterative development rather than specific agile practices.
- ✧ There are three phases in Scrum.
  - The **initial** phase is an outline planning phase where you establish the general objectives for the project and design the software architecture.
  - This is followed by a series of **sprint cycles**, where each cycle develops an increment of the system.
  - The project **closure** phase wraps up the project, completes required documentation such as system help frames and user manuals and assesses the lessons learned from the project.



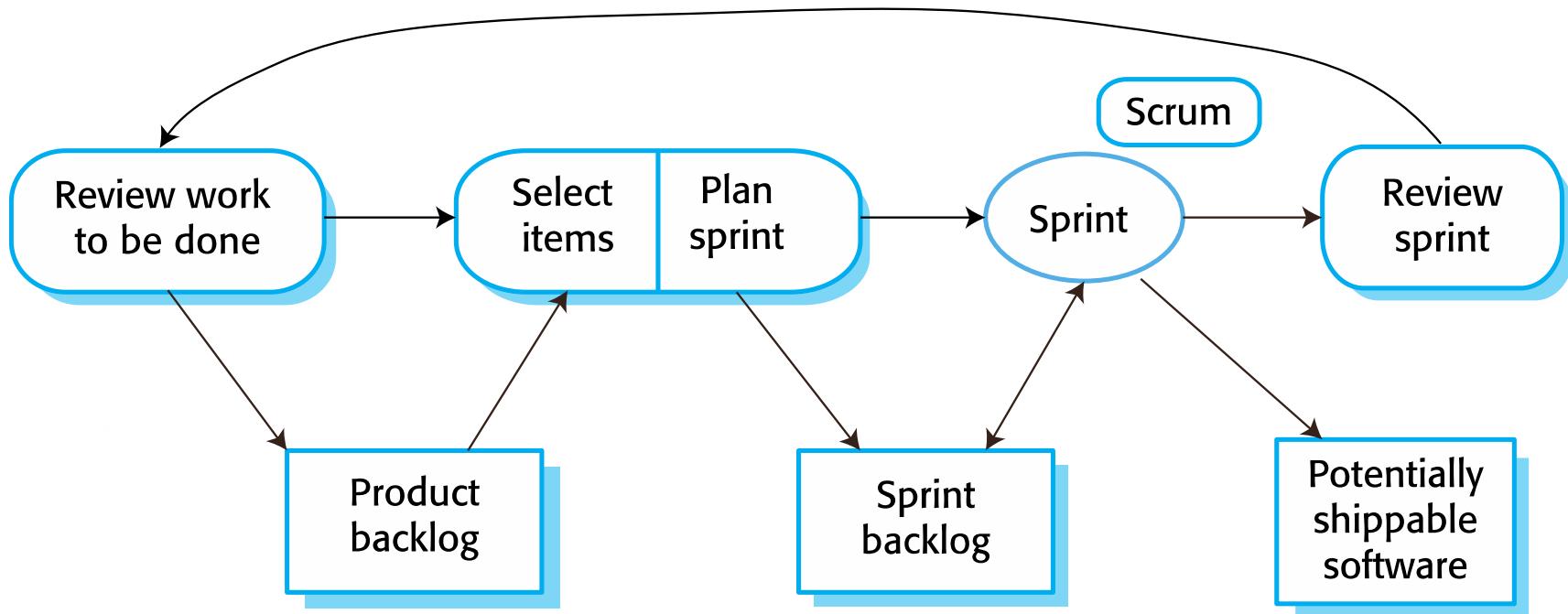
# Scrum terminology (a)

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than 7 people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be 'potentially shippable' which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of 'to do' items which the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.

# Scrum terminology (b)

Scrum term	Definition
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2-4 weeks long.
Velocity	An estimate of how much product backlog effort that a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

# Scrum sprint cycle



# The Scrum sprint cycle

- Sprints are fixed length, normally 2–4 weeks.
- The starting point for planning is the product backlog, which is the list of work to be done on the project.
- The selection phase involves all of the project team who work with the customer to select the features and functionality from the product backlog to be developed during the sprint.

# Słabe strony XP

- **Brak dokumentacji**
- **Jeden klient (na miejscu)**
- **Zbyt krótka perspektywa planu**

# Problemy metod zwinnych

---

- Trudność skalowania

# ***Zadania inżynieria oprogramowania***

Dr inż. Ilona Bluemke

## **Punkty funkcyjne -1**

---

Ze specyfikacji prostego projektu wynika, że

- będzie on korzystał z 10 wejść zmieniających dane systemu,
- generuje 12 raportów dla użytkownika,
- ma 20 interakcji z użytkownikiem nie zmieniających danych systemu,
- korzysta z 6 wewnętrznych oraz
- 4 zewnętrznych zbiorów danych. Jaki jest rozmiar tego projektu wyrażony w punktach funkcyjnych.

## Punkty funkcyjne -2

Funkcja	Prosta	Średnia	Złożona
EIF-pliki zew.	7	10	15
ILF-pliki wew.	5	7	10
EI-wejścia zew.	3	4	6
EO- wyjścia zew.	4	5	7
EQ- interakcje	3	4	6

## **Punkty funkcyjne -3**

---

- Współczynnik AFC (średnia liczba linii kodu na punkt) wynosi 50 LOC/FP.
- Przyjmując podstawowy model COCOMO dla prostego projektu, podać ile wynosi wysiłek potrzebny na wykonanie tego projektu [w osobo/miesiącach] oraz
- ile czasu zajmie wykonanie tego projektu [w miesiącach] (wystarczy podać wyrażenia z podstawionymi liczbami).

## **rozwiązanie**

---

1) Liczymy punkty funkcyjne

$$\text{UFC} = 10 * 3 \text{ (EI)} + 12 * 4 \text{ (EO)} + 20 * 3 \text{ (EQ)} + \\ 6 * 5 \text{ (ILF)} + 4 * 7 \text{ (EIF)} =$$

$$30 + 48 + 60 + 30 + 28 = 196 \text{ [punktów funkcyjnych]}$$

2) Liczymy liczbę linii kodu

$$\text{LOC} = \text{AVC} * \text{liczba punktów funkcyjnych}$$

$$\text{LOC} = 50 * 196 = 9800 \text{ [linii kodu]}$$

## Rozwiążanie cd

---

3) Liczymy wysiłek (model podstawowy dla tego  $M=1$ )

$$PM = 2.4 \text{ (KDSI)}^{1.05} * M = 2.4 (9.8)^{1.05} * 1 = 2.4 * 10.98 = 26.36$$

4) Liczymy czas

$$TDEV = 2.5 (PM)^{0.38} = 2.5 (26.36)^{0.38} = 2.5 * 3.46 = 8.66 \text{ [miesiący]}$$

## COCOMO 2

**Ile wyniosą koszt i czas wykonania projektu A oszacowane według COCOMO 2 po zakończeniu projektowania, jeśli :**

- projekt A przewiduje 100 klas,
- średnia liczba linii kodu dla klasy wynosi 80,
- wykładnik dla typowego projektu w danej firmie wynosi 1.14, ale w projekcie A występuje większe ryzyko (o 1 poziom większe niż w typowym projekcie),
- przewidywane jest implementacja mechanizmów tolerowania błędów (1.5).

**Wyniki uzasadnić.**

- **Jak zwolnienie jednej osoby w firmie wpłynie na obliczony czas wykonania projektu A?**

## **rozwiążanie**

---

- $PM = 2.94 * (100*80/1000)^{(1/100+1.14)} * 1.5 =$   
 $2.94 * 8^{1.15} * 1.5 = 2.94 * 10.93 * 1.5 =$   
48.2 osobo\_miesiące
- $TDEV = 3 * PM (0.33 + 0.2(1..15-1..01)) =$   
 $3*( 48.2)^{0.358} = 12 \text{ miesięcy}$

## cocomo 2

---

1. Na podstawie specyfikacji projektu wyznaczono, że jego realizacja wymaga 2000 punktów obiektowych. Średnia produktywność pracowników wynosi 20 op/miesiąc. Jaki jest przewidywany **koszt tego projektu** w modelu COCOMO 2 ?

## COCOMO 2

---

2. Ile w modelu COCOMO 2 wyniesie **koszt oszacowany po zakończeniu projektowania** oraz jaki jest przewidywany nominalny **czas wykonania** jeśli:
- projekt przewiduje 19000 linii kodu,
  - na podstawie doświadczeń wykładowik w danej firmie wynosi 1.16, ale w obecnym projekcie występuje większe ryzyko (o 1 poziomy większe niż poprzednio),
  - przewidywane są maksymalne wymagania niezawodnościowe (1,66).

Rozwiązanie uzasadnić.

# COCOMO

---

Wysiłek potrzebny na wykonanie pewnego projektu policzony wg podstawowego modelu COCOMO wynosi 1200 osobo/miesiąc. Podaj ile osobo/miesiący wyniesie on przy założonych współczynnikach:

1. niezawodność 1.3, narzędzia 1.1, normalny harmonogram 1.1
2. niezawodność 0.8, narzędzia 0.9, przyspieszony harmonogram 1.2

# **UML**

---

Opracuj diagram klas w UML, pokazujący co najmniej 10 relacji między obiektami klas .

Należy podać typ relacji, jej nazwę ewentualnie krotność. Odpowiedź należy uzasadnić.

Nazwy klas:

- **Szkoła, boisko, dyrektor, klasa, uczeń, nauczyciel, grono nauczycielskie, stołówka, sala, stół, krzesło, tablica, książka, biblioteka**

## **UML**

---

Listy kandydatów do parlamentu są zgłaszane przez partie. W każdym okręgu są zarejestrowane listy tylko tych partii, które uzyskały wymagany procent głosów. Dla każdego kandydata udostępnione są jego dane personalne. Kandydat może modyfikować swoje dane. Dane kandydatów danej partii może zmieniać również przewodniczący tej partii.

- Narysować przykładowy **diagram klas**.

## **UML**

---

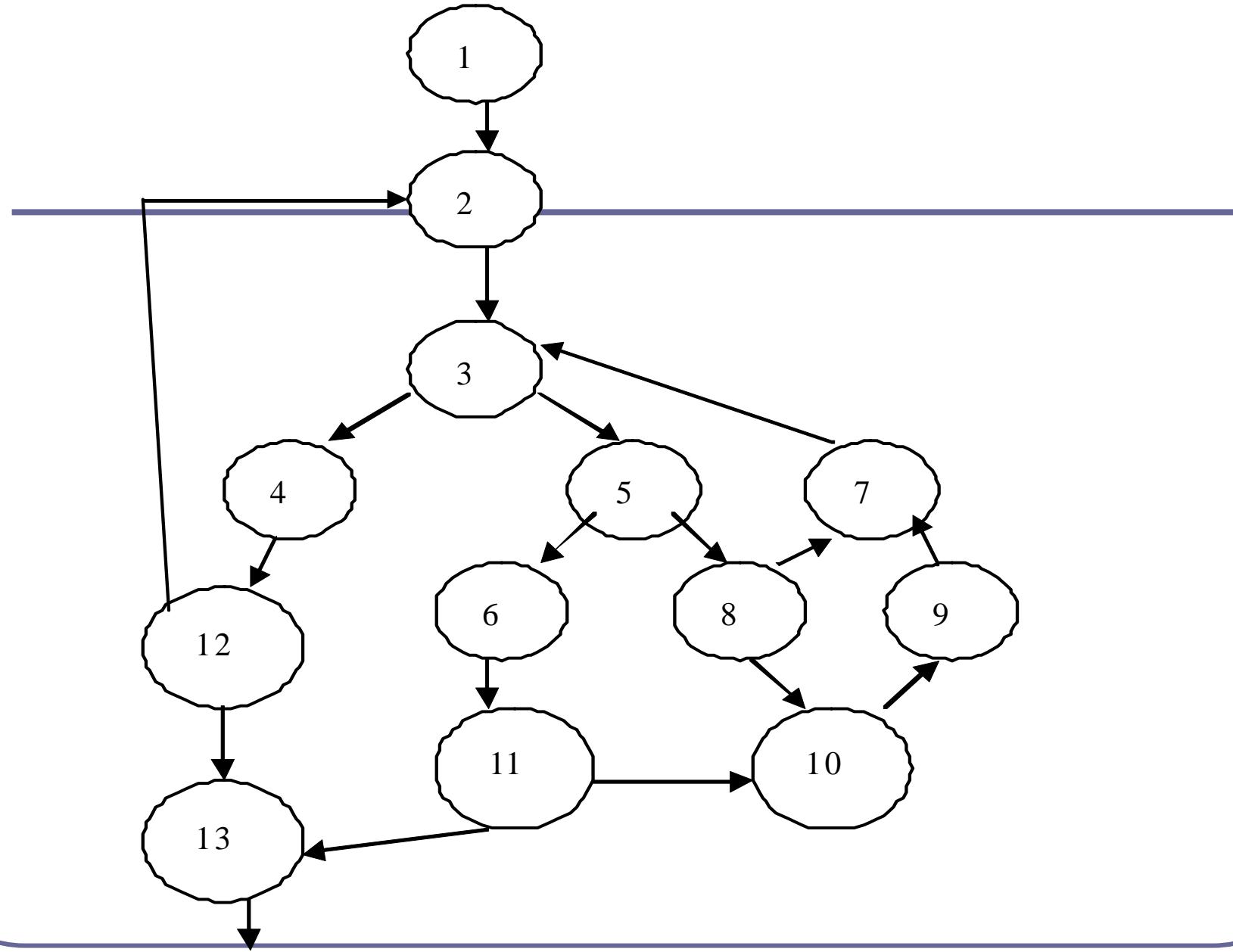
Dziennik zajęć przedmiotu zawiera zbiór danych dla poszczególnych studentów (w tym listę obecności oraz oceny). Dziennik może być modyfikowany przez prowadzącego zajęcia lub po uzgodnieniu z prowadzącym przez dziekana wydziału.

- Narysować przykładowy **diagram klas**.

## testowanie

---

- Dla poniższego grafu przepływu sterowania oblicz złożoność cyklotomatyczną Mc Cabe'a i podaj **wszystkie ścieżki niezależne**.
- Węzeł 1 jest węzłem początkowym a węzeł 13 jest węzłem końcowym.



## UML

---

Określ i narysuj w notacji UML typy relacji pomiędzy obiektami w poniższych zdaniach.  
Odpowiedź należy uzasadnić

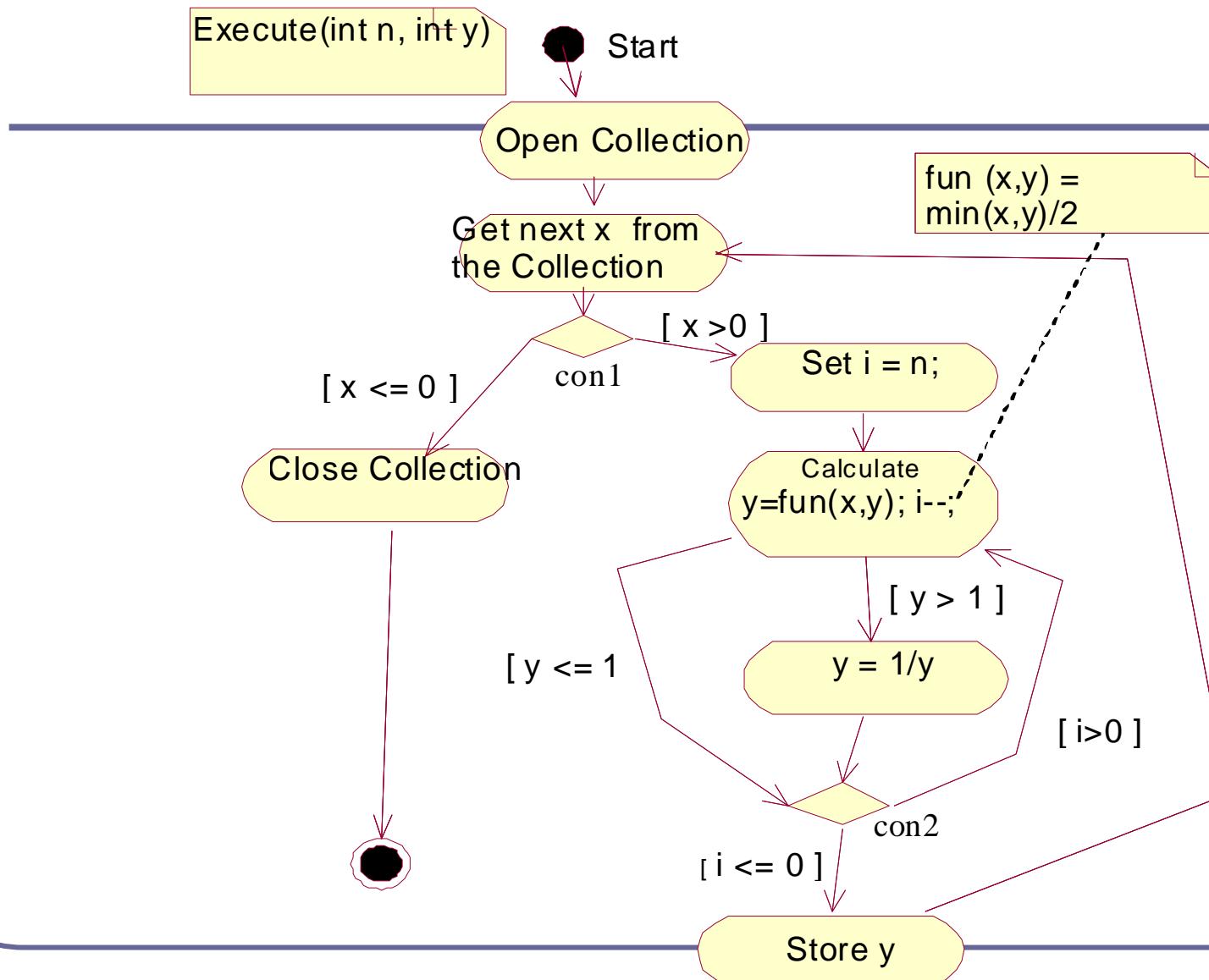
1. Miejscówka określa miejsce w pociągu
2. Poligon składa się z uporządkowanego zbioru
3. Aparat fotograficzny, kamera są urządzeniami rejestrującymi obraz
4. W plecaku znajdują się zeszyty, książki

## **Testowanie-1**

---

- Operacja Execute(int n, int y) posiada specyfikację w postaci diagramu czynności. Zaproponować taki minimalny zestaw danych testowych, żeby pokryć wszystkie liniowo niezależne ścieżki dla danej operacji. Określić te ścieżki.

## Testowanie-2



## Miary niezawodności

---

- Ile wynosi **miara dostępności** systemu pracującego 10 godzin na dobę,
- jeśli wiadomo, że średni czas naprawy błędu wynosi 15 minut,
- żądana jest 100 usług dziennie w godzinach pracy, a POFOD=0.001

## **Miary niezawodności - rozwiązańia**

---

$$(1-(100*0.001 *15)/ (10* 60))=$$

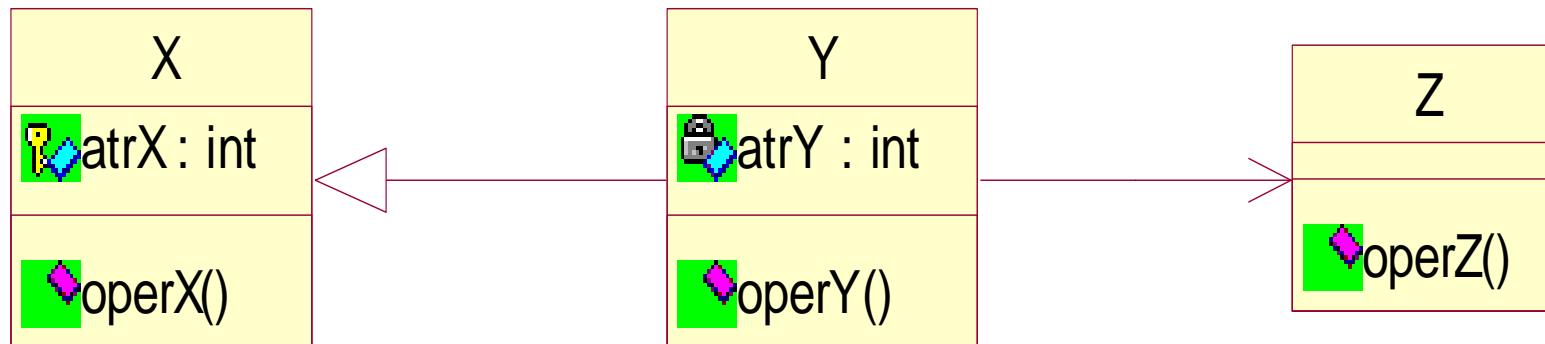
$$1-1/400=$$

$$1-0.0025= 0.9975$$

Odp 0.9975

# Testowanie obiektowe

- Dla podanego modelu określić jak powinniśmy testować klasę Y.



## **Testowanie obiektowe - rozwiążanie**

---

- funkcjonalnie i strukturalnie operacje,
- użycie atrybutu atrY;
- użycie odziedziczonego atrybutu atrX;
- metodę operY;
- odziedziczoną metodę operX na rzecz obiektów klasy Y;
- odwołanie w klasie Y do obiektów klasy Z
- stany i możliwe przejścia klasy Y, itp

## **Dla podanego programu przedstawić diagram Nassi-Shneidermana:**

```
unsigned P(int i)
/* Przekazuje i-tą liczbę pierwszą */
{ unsigned int n,pp,k;
n=tp[mtp];
while (mtp<i)
{ n+=2; k=2; pp=3; /* pp=tp[k] */
  while(pp*pp<=n)
    if (n%pp) pp=tp[++k]; else {n+=2; k=2; pp=3;}
    tp[++mtp]=n;
} return tp[i];
}
```

# ***Zadania inżynieria oprogramowania***

Dr inż. Ilona Bluemke

## **Punkty funkcyjne -1**

---

Ze specyfikacji prostego projektu wynika, że

- będzie on korzystał z 10 wejść zmieniających dane systemu,
- generuje 12 raportów dla użytkownika,
- ma 20 interakcji z użytkownikiem nie zmieniających danych systemu,
- korzysta z 6 wewnętrznych oraz
- 4 zewnętrznych zbiorów danych. Jaki jest rozmiar tego projektu wyrażony w punktach funkcyjnych.

## Punkty funkcyjne -2

Funkcja	Prosta	Średnia	Złożona
EIF-pliki zew.	7	10	15
ILF-pliki wew.	5	7	10
EI-wejścia zew.	3	4	6
EO- wyjścia zew.	4	5	7
EQ- interakcje	3	4	6

## **Punkty funkcyjne -3**

---

- Współczynnik AFC (średnia liczba linii kodu na punkt) wynosi 50 LOC/FP.
- Przyjmując podstawowy model COCOMO dla prostego projektu, podać ile wynosi wysiłek potrzebny na wykonanie tego projektu [w osobo/miesiącach] oraz
- ile czasu zajmie wykonanie tego projektu [w miesiącach] (wystarczy podać wyrażenia z podstawionymi liczbami).

## **rozwiązanie**

---

1) Liczymy punkty funkcyjne

$$\text{UFC} = 10 * 3 \text{ (EI)} + 12 * 4 \text{ (EO)} + 20 * 3 \text{ (EQ)} + \\ 6 * 5 \text{ (ILF)} + 4 * 7 \text{ (EIF)} =$$

$$30 + 48 + 60 + 30 + 28 = 196 \text{ [punktów funkcyjnych]}$$

2) Liczymy liczbę linii kodu

$$\text{LOC} = \text{AVC} * \text{liczba punktów funkcyjnych}$$

$$\text{LOC} = 50 * 196 = 9800 \text{ [linii kodu]}$$

## Rozwiążanie cd

---

3) Liczymy wysiłek (model podstawowy dla tego  $M=1$ )

$$PM = 2.4 \text{ (KDSI)}^{1.05} * M = 2.4 (9.8)^{1.05} * 1 = 2.4 * 10.98 = 26.36$$

4) Liczymy czas

$$TDEV = 2.5 (PM)^{0.38} = 2.5 (26.36)^{0.38} = 2.5 * 3.46 = 8.66 \text{ [miesiący]}$$

## COCOMO 2

**Ile wyniosą koszt i czas wykonania projektu A oszacowane według COCOMO 2 po zakończeniu projektowania, jeśli :**

- projekt A przewiduje 100 klas,
- średnia liczba linii kodu dla klasy wynosi 80,
- wykładnik dla typowego projektu w danej firmie wynosi 1.14, ale w projekcie A występuje większe ryzyko (o 1 poziom większe niż w typowym projekcie),
- przewidywane jest implementacja mechanizmów tolerowania błędów (1.5).

**Wyniki uzasadnić.**

- **Jak zwolnienie jednej osoby w firmie wpłynie na obliczony czas wykonania projektu A?**

## **rozwiążanie**

---

- $PM = 2.94 * (100*80/1000)^{(1/100+1.14)} * 1.5 =$   
 $2.94 * 8^{1.15} * 1.5 = 2.94 * 10.93 * 1.5 =$   
48.2 osobo\_miesiące
- $TDEV = 3 * PM (0.33 + 0.2(1..15-1..01)) =$   
 $3*( 48.2)^{0.358} = 12 \text{ miesięcy}$

## cocomo 2

---

1. Na podstawie specyfikacji projektu wyznaczono, że jego realizacja wymaga 2000 punktów obiektowych. Średnia produktywność pracowników wynosi 20 op/miesiąc. Jaki jest przewidywany **koszt tego projektu** w modelu COCOMO 2 ?

## COCOMO 2

---

2. Ile w modelu COCOMO 2 wyniesie **koszt oszacowany po zakończeniu projektowania** oraz jaki jest przewidywany nominalny **czas wykonania** jeśli:
- projekt przewiduje 19000 linii kodu,
  - na podstawie doświadczeń wykładowca w danej firmie wynosi 1.16, ale w obecnym projekcie występuje większe ryzyko (o 1 poziomie większe niż poprzednio),
  - przewidywane są maksymalne wymagania niezawodnościowe (1,66).

Rozwiązanie uzasadnić.

# COCOMO

---

Wysiłek potrzebny na wykonanie pewnego projektu policzony wg podstawowego modelu COCOMO wynosi 1200 osobo/miesiąc. Podaj ile osobo/miesiący wyniesie on przy założonych współczynnikach:

1. niezawodność 1.3, narzędzia 1.1, normalny harmonogram 1.1
2. niezawodność 0.8, narzędzia 0.9, przyspieszony harmonogram 1.2

# **UML**

---

Opracuj diagram klas w UML, pokazujący co najmniej 10 relacji między obiektami klas .

Należy podać typ relacji, jej nazwę ewentualnie krotność. Odpowiedź należy uzasadnić.

Nazwy klas:

- **Szkoła, boisko, dyrektor, klasa, uczeń, nauczyciel, grono nauczycielskie, stołówka, sala, stół, krzesło, tablica, książka, biblioteka**

## **UML**

---

Listy kandydatów do parlamentu są zgłaszane przez partie. W każdym okręgu są zarejestrowane listy tylko tych partii, które uzyskały wymagany procent głosów. Dla każdego kandydata udostępnione są jego dane personalne. Kandydat może modyfikować swoje dane. Dane kandydatów danej partii może zmieniać również przewodniczący tej partii.

- Narysować przykładowy **diagram klas**.

## **UML**

---

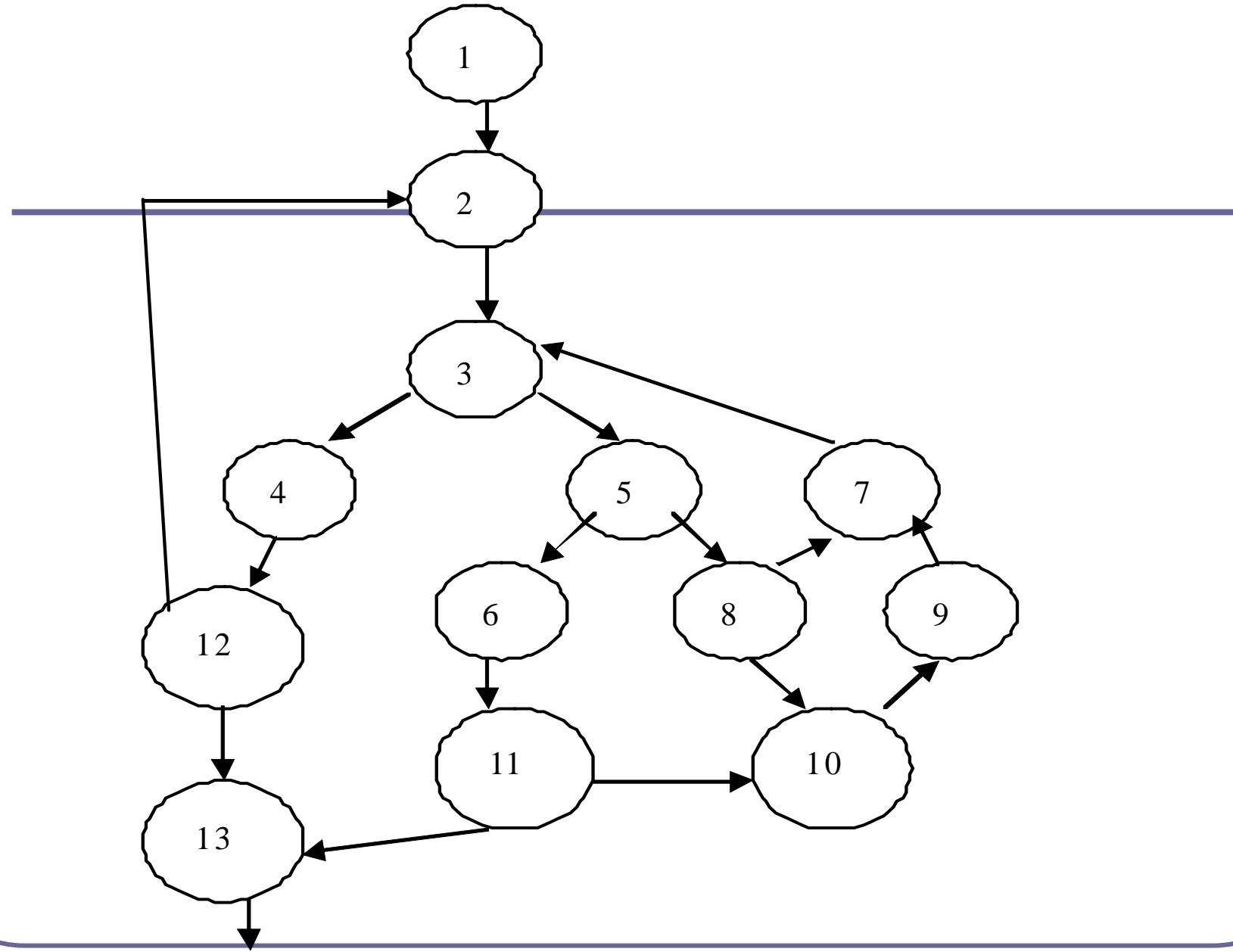
Dziennik zajęć przedmiotu zawiera zbiór danych dla poszczególnych studentów (w tym listę obecności oraz oceny). Dziennik może być modyfikowany przez prowadzącego zajęcia lub po uzgodnieniu z prowadzącym przez dziekana wydziału.

- Narysować przykładowy **diagram klas**.

## **testowanie**

---

- Dla poniższego grafu przepływu sterowania oblicz złożoność cyklotomatyczną Mc Cabe'a i podaj **wszystkie ścieżki niezależne**.
- Węzeł 1 jest węzłem początkowym a węzeł 13 jest węzłem końcowym.



## UML

---

Określ i narysuj w notacji UML typy relacji pomiędzy obiektami w poniższych zdaniach.  
Odpowiedź należy uzasadnić

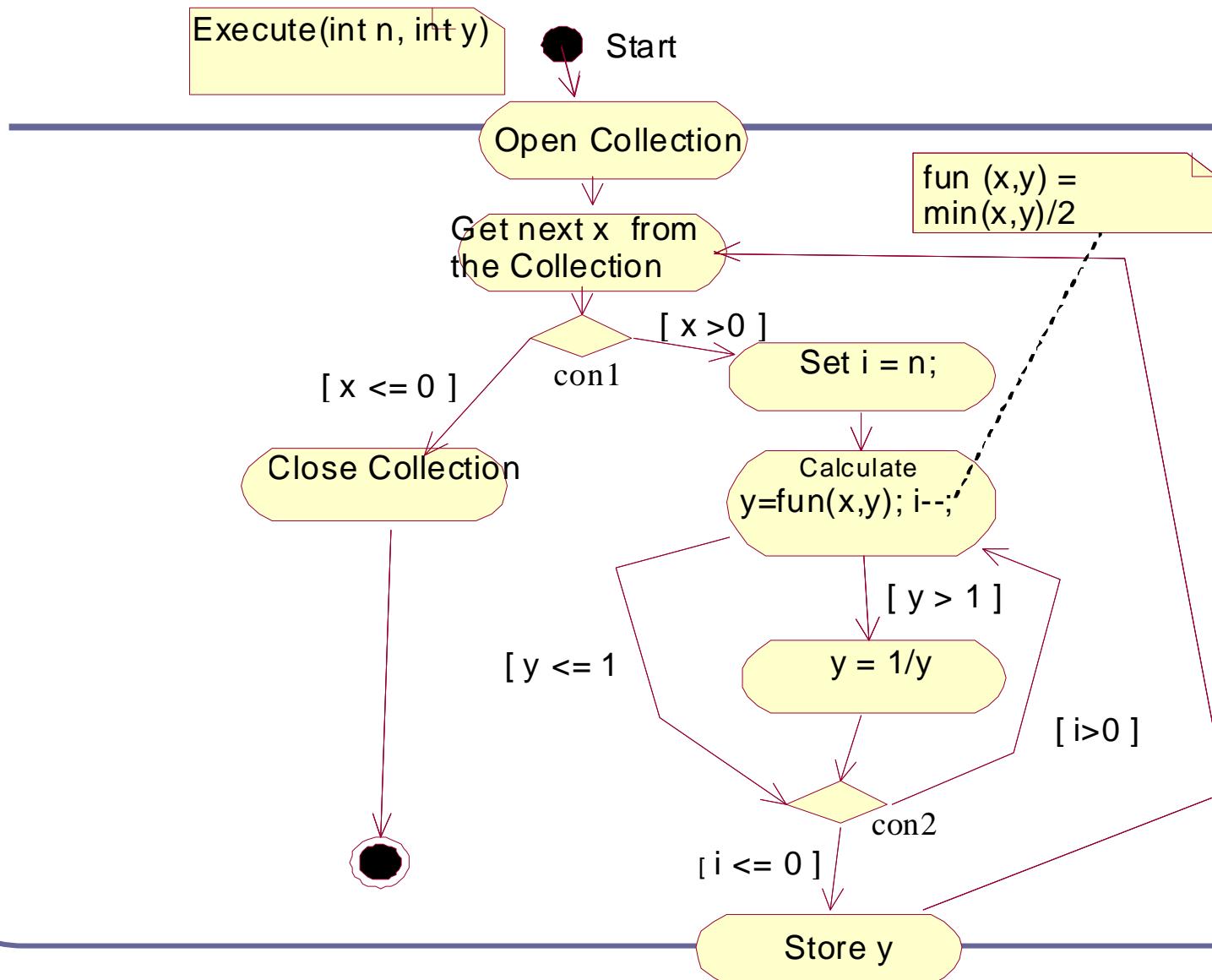
1. Miejscówka określa miejsce w pociągu
2. Poligon składa się z uporządkowanego zbioru
3. Aparat fotograficzny, kamera są urządzeniami rejestrującymi obraz
4. W plecaku znajdują się zeszyty, książki

## **Testowanie-1**

---

- Operacja Execute(int n, int y) posiada specyfikację w postaci diagramu czynności. Zaproponować taki minimalny zestaw danych testowych, żeby pokryć wszystkie liniowo niezależne ścieżki dla danej operacji. Określić te ścieżki.

## Testowanie-2



## Miary niezawodności

---

- Ile wynosi **miara dostępności** systemu pracującego 10 godzin na dobę,
- jeśli wiadomo, że średni czas naprawy błędu wynosi 15 minut,
- żądana jest 100 usług dziennie w godzinach pracy, a POFOD=0.001

## **Miary niezawodności - rozwiązańia**

---

$$(1-(100*0.001 *15)/ (10* 60))=$$

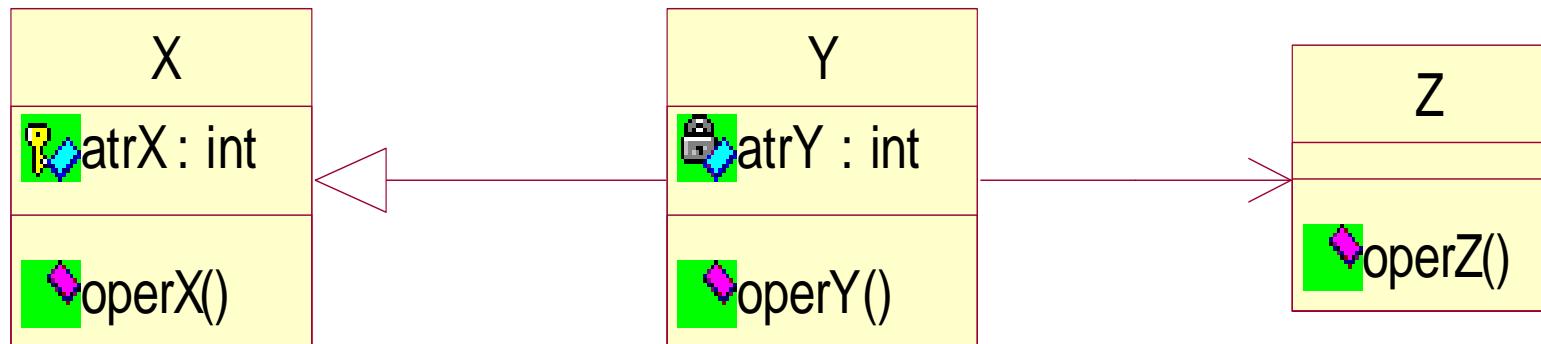
$$1-1/400=$$

$$1-0.0025= 0.9975$$

Odp 0.9975

# Testowanie obiektowe

- Dla podanego modelu określić jak powinniśmy testować klasę Y.



## **Testowanie obiektowe - rozwiążanie**

---

- funkcjonalnie i strukturalnie operacje,
- użycie atrybutu atrY;
- użycie odziedziczonego atrybutu atrX;
- metodę operY;
- odziedziczoną metodę operX na rzecz obiektów klasy Y;
- odwołanie w klasie Y do obiektów klasy Z
- stany i możliwe przejścia klasy Y, itp

## **Dla podanego programu przedstawić diagram Nassi-Shneidermana:**

```
unsigned P(int i)
/* Przekazuje i-tą liczbę pierwszą */
{ unsigned int n,pp,k;
n=tp[mtp];
while (mtp<i)
{ n+=2; k=2; pp=3; /* pp=tp[k] */
  while(pp*pp<=n)
    if (n%pp) pp=tp[++k]; else {n+=2; k=2; pp=3;}
    tp[++mtp]=n;
} return tp[i];
}
```

Narysuj fragment diagramu klas (klasy, relacje, operacje), wynikający z podanego poniżej diagramu sekwencji.  
Uzasadnij swoje rozwiązanie.

