

ZALICZENIE:

Egzamin - 0-100 pkt, zdobycie 51 pkt

Extra punkty na wykładech za aktywność.

MATERIAŁ:

- UML - projektowanie systemów (wersja 2.0)

- Inżynieria oprogramowania

* Inżynieria oprogramowania (ang. software engineering) - wprowadzenie

- Początki pod koniec lat 60-tych ze względu na kryzys oprogramowania (wiele niepowodnych projektów nie zostało ukończonych, koszt produkcji oprogramowania rosły liniowo - bardzo szybko)

- Jej celem jest produkcja oprogramowania wydajnej jakości w sposób jak najbardziej efektywny
Oprogramowanie wydajnej jakości - daje zgodnie z wymaganiami określonymi przez specyfikację, jest szybkie, wydajne, funkcjonalne dla użytkownika, bezpieczne w miarę tego modyfikowalne i poprawiać (pielęgnacja oprogramowania), posiada pełną dokumentację projektową i użytkową.

* Modele procesu produkcji:

- Model wodospadowy (ang. waterfall) - kaskadowy, liniowy, nieprzypadkowy z prec inżynierijnych.

Fazy:

Wsp. kosztów bieżących

0,2-0,3

I 1) Specyfikacja wymagań - jakie funkcje powinno zawierać oprogramowanie, jakie ograniczenia należy brać pod uwagę (np. ile pamięci moindre wykorzystać).

0,5

II 2) Projektowanie (ang. design)

1

III 3) Implementacja (ang. implementation, coding)

5-15

IV 4) Testowanie

20-100

V 5) Urzatkowanie i pielęgnacja

Zalety:

- łatwe zarządzanie projektem

- Narzuca kolejność wykonywanie prec (kiedy zmianielegienia)

etap kończy się fizyczną realizacją)

Wady:

- Narzuca kolejność wykonywanie prec (kiedy zmianielegienia)

- Wykonuje się bieżących poprzednich w pojętkowych fazach

- Długi przerwy kontaktach z klientem

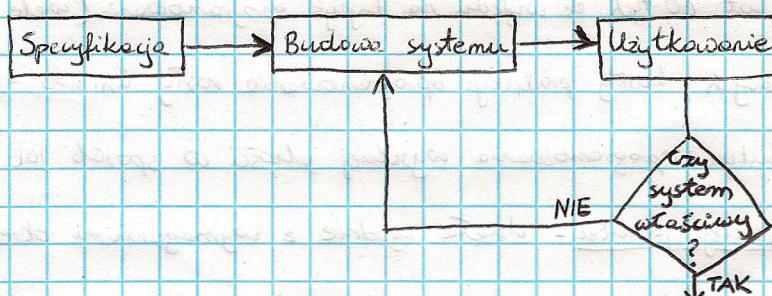
Weryfikacja - sprawdzenie czy właściwie budujemy produkt i czy spełnia wymagania, wykonanie testów akceptacyjnych można rozwać weryfikacją, na podstawie specyfikacji.

Weryfikacja - sprawdzenie czy budujemy właściwy produkt i my funkcje są takie, jakich klient oczekuje, musi być wykonane z uziemiem zamawiającego.

Oprogramowanie przejście weryfikacji i weryfikacji, kiedy specyfikacje w pełni odpowiadają oczekiwaniom klienta.

- Model ewolucyjny (ang. exploratory programming)

Fazy:



Zalety:

- Może stosować się, kiedy są kłopoty ze stworzeniem dokładnej specyfikacji

- Struktura systemu jest bardzo rozwinięta
- Konieczność bardzo szybkiej produkcji
- Nie ma weryfikacji
- Nie gwarantuje możliwości pięgnowania systemu

- Prototypowanie

Fazy:

- 1) Ogólne określenie wymagań
- 2) Opracowanie syntetycznego prototypu
- 3) Weryfikacja prototypu przez klienta
- 4) Określenie szczegółowych wymagań

Cele prototypu:

- może wyjaśnić trudne usługi
- może wyjaśnić błęki w specyfikacji
- może wyjaśnić ewentualne nieporozumienia między klientem a projektantem
- pozwala na demonstrację przyszłego systemu

- daje możliwość skutecznego zarządzania zmianami w systemie.

04.10.2012
INO-W

Budowanie prototypów:

- programowanie ewolucyjne
- wykorzystanie gotowych komponentów
- niepełna realizacja
- język o wysokim poziomie
- generatory np. interfejsów użytkownika

Formalne transformacje:

11.10.2012
INO-W

- wymaganie wobec systemu są represowane w języku formalnym, które podlegają automatycznym przekształceniom

Zalety:

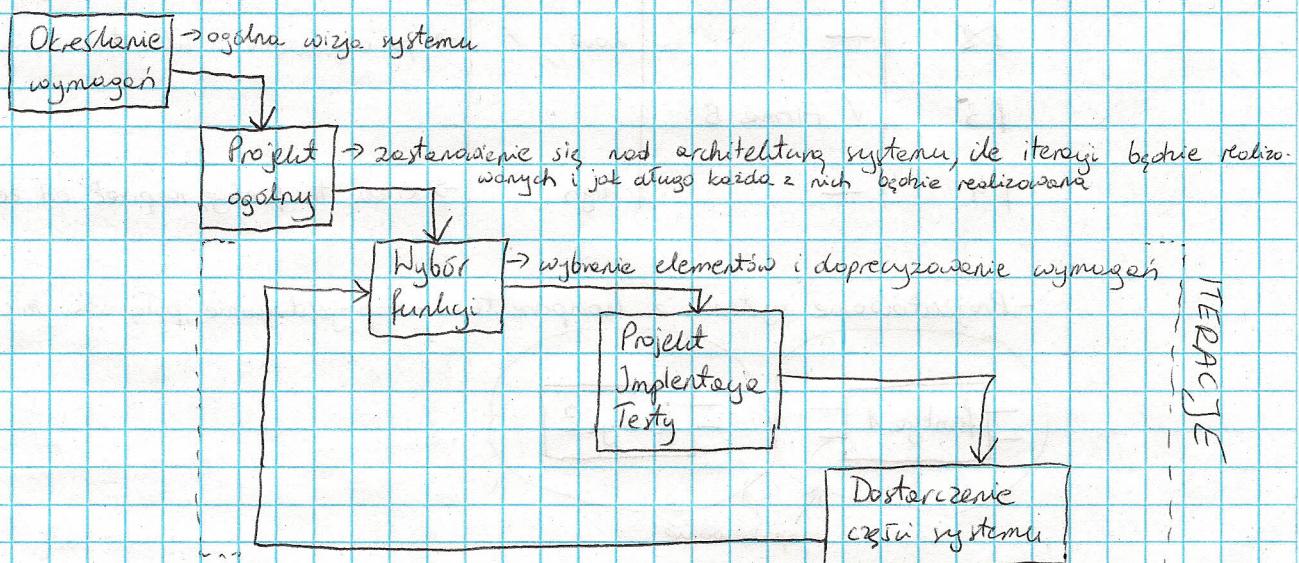
- wydajna niezawodność (brak błędów w transformacjach)

Wady:

- trudność formalnego specyfikowania
- mała efektywność kodu (nie wszystko da się w ten sposób opisać)

- stosuje się do produkcji systemów wymagających dużej niezawodności, bezpieczeństwo, wymaga dużych umiejętności zespołu.

Realizacja przyrostowa (ang. incremental development)



Kryteria wyboru funkcji:

- 1) potrzebno do realizacji następnych
- 2) priorytet dla użytkownika
- 3) łatwość realizacji

Zalety:

- prosty kontekst z klientem
- możliwość wcześniego wykorzystania części systemu
- długie testowanie najnowocześniejszych części systemu przez użytkownika

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

- korzysta się z gotowych, dostępnych komponentów i tworzy nisz kod integrujący je.

COTS (ang. Commercial Off The Shelf)

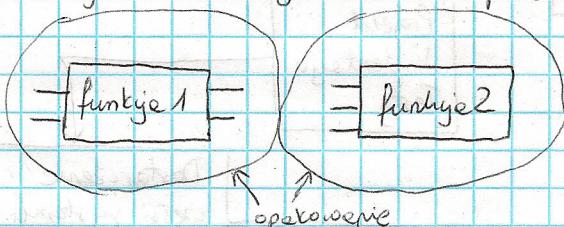
Stosowanie: bibliotek, języków czwartej generacji, pełnych aplikacji.

Proces produkcji:

- Specyfikuje wymagania
- Analiza komponentów - poszukiwanie komponentów spełniających oczekiwania lub częściowo wymaganie funkcji systemu.
- Modyfikuje wymagania - dostosowanie wymagań do znalezionych komponentów

wymaganie	gotowe komponenty	priorytet	
f.1	✓ Firma A		
f.2	—	low	→ po użyczeniu z klientem można pominić
f.3	✓ Firma B		
f.4	—	High	→ trzeba tę funkcję napisać od zera.

- Projektowanie systemu z komponentami - zaprojektowanie „potoku” między komponentami



- Realizuje systemu i integruje

Zalety

- wysoka niezawodność (mniej błędów)
- unowocześnienie standardów
- małe koszty, duża szybkość

Wady:

- dodatkowy koszt związany z realizacją fragmentów systemu (pinowanie subelementów modułów)

Wady

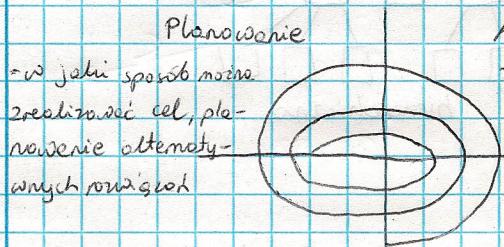
- dodatkowy koszt przygotowania elementów do ponownego użycia
- ryzyko uszkodzenia od dostarczonych komponentów

- wysokie kwestie pielęgnowania

INO-W

- nie w pełni realizowane wymagania klienta

Model spiralny (Boehm)



Analiza ryzyka

- przemyślenie, jakie ryzyko wiele się z innymi związane, wybranie rozwiązania o najmniejszym ryzyku

Konstrukcja

- doprecyfrowanie rozwiązania

W modelu spiralnym można wykorzystać inne modele np. prototypowanie

- Modele o dobrej obszerności:

1) wodoszczelny

2) formalnych transformacji (kiedy transformacje konwertują się dokumentem)

3) spiralny (w każdej segmentacji spirali powstaje dokument)

- Modele o dużej obszerności

1) Montaż z gotowych komponentów

2) Model ewolucyjny - największa możliwość obserwowania procesu produkcji

Poniekad... - Jakią rola "deliverable" w modelu wodoszczelnym? (Co dostaje klient?)
we pytanie

Specyfikacja wymagań, dokumentacje techniczna i użytkownika, kod, report.

Czynniki nie-techniczne w inżynierii oprogramowania:

- zarządzający projektorem powinien rozumieć język projektowy

- brać pod uwagę możliwości i ograniczenia użytkowników systemu

- znajomość czynników społecznych

Dane z IBM Proporcje czasu:

30% - praca własna

20% - czynności nieprodukcyjne

50% - interakcje z innymi członkami zespołu

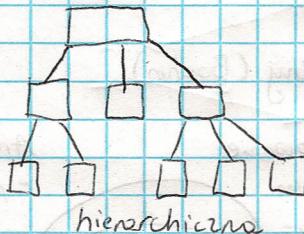
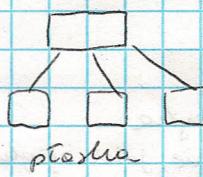
Zespoły, których członkowie znają się lepiej, efektywniej pracują.

Interakcje w grupie.

Czynniki wpływające na efektywność interakcji:

- wielkość grupy

- struktura:



- status i osobowość członków

- środowisko pracy

Inżynieria wymagań:

- Faza strategiczna - zaczyna się przed podjęciem decyzji o realizacji dalszych etapów przedsięwzięcia.

• Studium wykonalności (ang. feasibility study) - czy w ogóle jesteśmy w stanie to oprogramowanie wykonać, oszacowanie kosztów (czas i pieniądze), określenie wstępnego harmonogramu.

• Decyzje strategiczne - wybór modelu, technik, środowiska, narzędzi, rozważanie wykonywanie gotowych komponentów (ang. w jednym stopniu)

• Rozwijać kilka możliwych rozwiązań pod kątem ryzyka, przy uwzględnieniu ograniczeń.

• Normy jakości oprogramowania.

• Syndrom LOOP (ang. Late Overbudget Overtime Poor quantity)
poziom, przekroczone budżet, nadgodziny, niepełna jakość.

• Przyczyny niepowodzenia projektów

Aktor - użytkownik lub system, który komunikuje się z naszym systemem.

Ukierunkowany - osoby, na które system ma wpływ (nie tylko użytkownicy)

• Ograniczenie motywacji na rozwijanie

Uzyskiwanie wymagań: wysiad/ankiety (wymowy z użytkownikami), warsztaty

wymagań (obiera się grupy użytkowników), burza mózgów, przypadkii użycia, odgrywanie ról, prototypy.

Rodzaje wymagań -

- * Wymaganie funkcyjne - usługi określone przez użytkownika (co system robi)

- * Wymaganie niefunkcyjne - ograniczenie w jakich system ma pracować. (ograniczenie)

Wymaganie funkcyjne powinno być pełne (zwierzęć wszystkie wymagania użytkownika) i spójne (nie sprzeczne).

Przykłady: wymagania produktu (wym. niefunkcyjne), wymaganie organizacyjne (wym. niefunkcyjne), wymaganie reaktywne (wym. niefunkcyjne).

Specyfikacje wymagań:

- strukturalny język naturalny (formularze, schematy)

- pseudokod (if, else, while...)

- język opisu projektu, wymagań (PDL, PSL/PSA)

- notatki graficzne ze strukturalnymi opisami (SADT, przypadki użycia - use case).

Klasy stałości wymagań:

- wymaganie stałe - stabilne, wynikające z podstawowej działalności firmy.

- wymaganie niestałe - prawdopodobnie nieogniżemie zmianie w toku tworzenia systemu.

Poziomy identyfikacji wymagań

1) Potrzeby udzielańców

Cele systemu - często niesprecyficzne i niejednoznaczne

2) Cechy systemu - usługi, które system będzie realizować w celu spełnienia potrzeb udzielańców

3) Wymagania stwierdzone w projekcie

Wprowadzenie do projektowania:

Projektowanie - proces tworzenia, w następujących krokach:

- Studiowanie i zrozumienie problemu, badanie problemu z różnych punktów widzenia,

- Sztućczenie wielu rozwiązań i ich ocena pod kątem wad i zalet.

- Opis każdej abstrakcji w uzyciu rozwiązań.

Metody projektowania (zbior dobrych nad i wstężeńek):

- podejście funkcyjne (strukturalne) - ~~zakresu zarządzania~~ transformacji danych

- podejście obiektowe - system jako zbiór obiektów, powiązanych relacjami

Jakość projektu - nie ma obiektywnej metody stwierdzającej, że projekt jest dobry.

"Dobry" projekt:

- spełnia specyfikę
- powstaje na produkcyjnego kodu
- posiada pożądane właściwości

System daje się łatwo pielęgnować, gdy jest:

- spójny,
- niski stopień zależności między jednostkami systemu,
- adekwatny, łatwy do zrozumienia.

* Modelowanie z różnych perspektyw

* Modele architektury

* Organizacje systemu:

- Dzielone reprezentatorium - podsystemy muszą wymieniać dane (1) centralne bazy danych, 2) dane w podsystemach są przechowywane)
- Dzielone serwisy - rozproszony model
 - Warstwowa - system jako zbiór warstw (maszyn obliczeniowych)

* Style sterowania

- skonsolidowane sterowanie - 1 podsystemem odpowiedzialnym za sterowanie (model call-return/manager)
- sterowanie zdarzeniowe - sterowanie generowane zdarzeniami (modele systemów zdarzeniowych)

22.10.2012
INO - W

- Analiza systemu

Analitycy biorą udział w:

- weryfikacji wymagań
- analizie wymagań
- ocenie technicznej projektu wstępnie i szczegółowej
- wdrożeni i weryfikacji oprogramowania

Określenie wytyczne co do pielęgnowania systemu.

Analiza obejmuje stery:

- problemu (co ma być wykonane)
- wykonalności (czy realizacja jest możliwa)
- ekonomicznej (jakią będą koszty i ryzyko realizacji)

Zasady (sposoby) przy podejściu do stojącego problemu (podejście hierarchiczne) :

- Abstrakcja - skoncentrowanie się na istotnych aspektach.

- Abstrakcja proceduralna

- Abstrakcja danych.

W projektowaniu zorientowanym obiektowo system jest widziany jako

zbior współpracujących ze sobą obiektów.

Obiekt ma pewną indywidualność, pełni różne role:

- aktor - aktynowy, prezyde

- sensor - dostarcza informacji na życzenie

- agent - obiekt powyższej role.

* Analiza zorientowana obiektowo (proces iteracyjny)

1) Znajdowanie - identyfikacja obiektów

2) Organizowanie obiektów - ustalenie hierarchii, zależności (podobieństwo, dziedziczenie)

3) Opis interakcji - które obiekty mają ze sobą współpracować i w jakim sposób.

4) Definicja operacji obiektu - jakie metody obiekt powinien posiadać i parametry.

5) Definicja wnętrza obiektu - typy danych, pole, algorytmy w metodach.

Ad. 1) Obiekt to ważny podmiot (reprezentant) z dziedziny problemu.

Typy obiektów:

- aktywne / pasywne - obiekty które są aktywne od systemu / obiekty przechowujące dane

- fizyczne / konceptualne - obiekty istniejące / obiekty wymodelowane potrzeby projektu

- chwilowe / stałe - obiekty przechowujące dane na jakiś czas / na zawsze

- prywatne / publiczne - dostępność obiektów

- częśc / całość - obiekty skonkretizowane / grupy obiektów skojarzonych

Na tym etapie nie bierzemy pod uwagę ograniczeń / problemów implementacyjnych.

- ogólne / specyficzne - gromadzenie cech wspólnych wielu obiektów / specyficzne cechy

Ad. 2) Jakią są cechy wspólne klas / obiektów - tworzenie hierarchii dziedziczenia

Które obiekty współpracują ze sobą?

Które obiekty są częściami innych obiektów?

Jakie obiekty są od siebie zależne?

Ad. 3) Opisywanie scenariuszy użycia systemu (use case)

Ad. 4) Definicja operacji obiektów na podstawie scenariusza

Jeśli operacje są zlozone to można identyfikować nowe obiekty.

Ad 5) Implementacja, zastanowienie się nad algorytmami i strukturami danych, które powinny być użyte.

* UML.

- Perspektywy

2
Poziom logiczny
Oznaczenie obiektów
ich powiązań itd

1
Use Case
przykładu użycia.
jako to ma działać,
co system ma zrobić

Use Case - zbiór wykonywanych przez system akcji.

Elementy diagramów Use Case:

- aktor (osoba, system zewnętrzny), przyczyna bezpośredniego przyjęcia użycia.

- przyjęcie użycie  nowe - nowy

- relacje  asocjacja (powiązanie)
dwukierunkowa  asocjacja
jednokierunkowa

Powiązanie aktora jello osoby z use case jest ZAWSZE dwukierunkowe !

Jeli aktor jest uogólnieniem /systemem to powiązanie się jednokierunkowe

Działanie wstępne:

- rozpoznanie przedstawionego problemu - wyjaśnij, określanie użytkownika, modelowanie itp.

Relacje:

 zależność jednokierunkowa
 zależność dwukierunkowa
spejalizuje ogólne
generaliżuje (dzieńczenie)

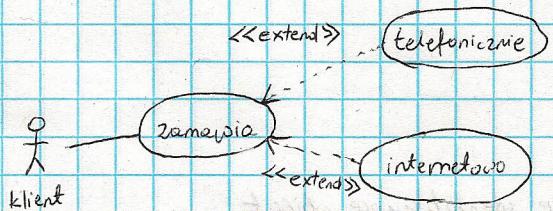
Strukturyzacja use case:

- use case może być spejalizacją innego (generalizacja)

- use case może być włączony jako część innego (<<include>>) : 

- use case może rozszerzać zachowanie innego (<<extend>>) : 

Pomiędzy use case'ami musi być linie zależności (przymiotne !)



Zazwyczaj nie stosujemy generalizacji use case - do, ale stosujemy generalizację aktorów.

• Konsystentne tworzenie przypadków użycia (wg. Cockburna)

1) identyfikuj aktorów i ich cele

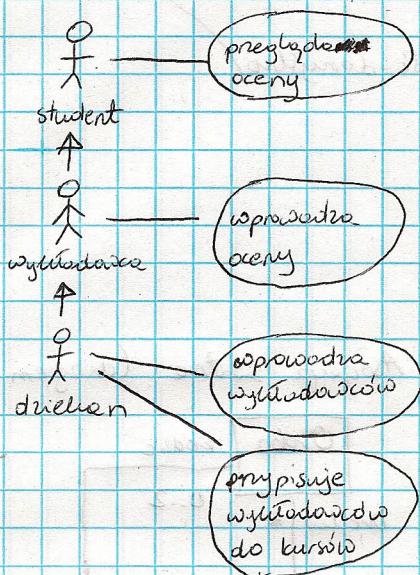
2)

3)

4)

ćWICZENIE: Narysuj diagram use case

Student przegląda oceny, wykładowca przegląda oceny i wprowadza oceny, druhów wprowadza i przegląda oceny i wprowadza wykładowców i przypisuje wykładowców do kursu.



Diagramy klas

25.10.2012

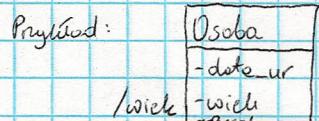
INO-W

- Klasa

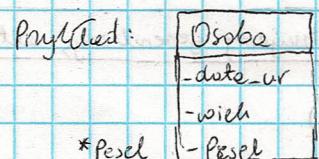
nowe	klasy
atrybuty	- name - zakres dostępnego - prywatny - widoczny tylko dla obiektów tej klasy + name - zakres dostępnego - publiczny - widoczny dla wszystkich klas # name - zakres dostępnego - chroniony - widoczny dla obiektów tej klasy i klas, które po niej dziedziczą
operacje (metody)	

Atrybuty mogą być:

- /name - wyodrębnienie z innych atrybutów



- *name - kluczowe, myli jednoznacznie identyfikujące obiekt



- ?name - implementując - decyduje co do widoczności lub wstępów atrybutów będzie ustalone na etapie implementacji.

Operacje: funkcje lub transformacje.

- może mieć argumenty parametryzujące

- może zwracać wynik

- może mieć określony zakres widoczności

Specyfikacja operacji klasy w języku UML: widoczność nazwa (list_argumentow): typ (wartosciawslis)

* list_argumentow

spśób - precyzowania nazwy : typ = wartość - domyślna
 ↓
 in, out, inout

Przykład:

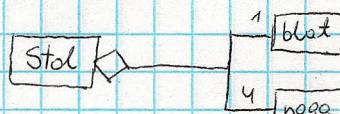


Diagram obiektów

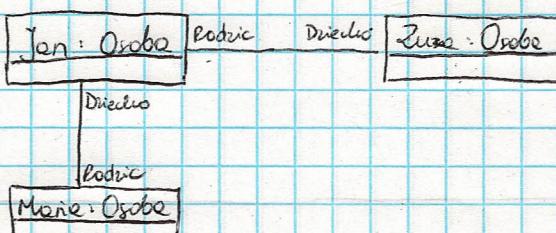
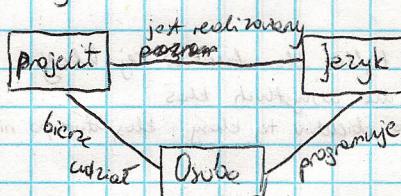


Diagram klas



Asocje zmienné: (diagram UML)



Diagram obiektów:

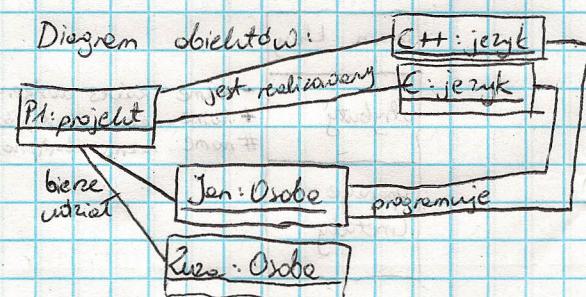
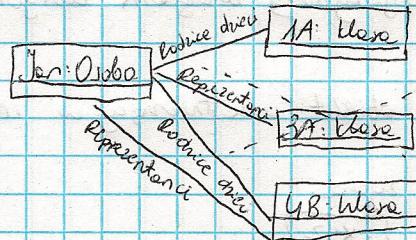


Diagram klas:



ogreniczenie - reprezentanci
podzbiorzem rodziców dzieci.

Diagram obiektów

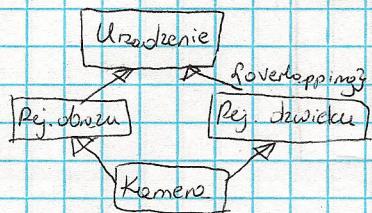


niedopuszczalne by Jan
był reprezentantem w
klasie, w której nie
ma dzieci.

○ — agreguje (ang. aggregation) - mogą istnieć egz. bez wartości

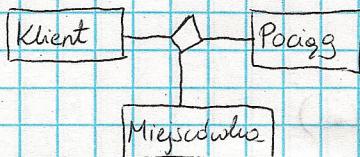
◆ — kompozycje (ang. composition) - nie mogą istnieć egz. bez wartości

* Dziedziczenie:



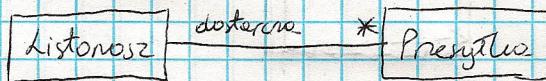
ZADANIE EGZAMINACYJNE:

1)



asocjacja tematyczna, dlatego,
że pomiędzy tymi plikami
jest bardzo silny związek.

2)

3)

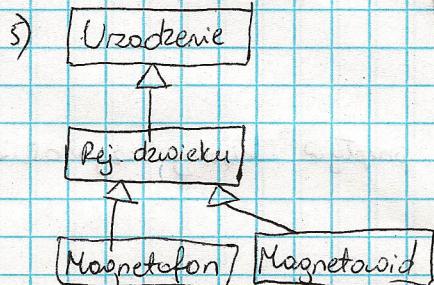
```

classDiagram
    class Bileter
    class Bilet
    Bileter --> Bilet : sprzedaje
  
```

4)

```

classDiagram
    class Plecak
    class Książka
    class Zeszyt
    Plecak "3" --> Książka
    Plecak "3" --> Zeszyt
  
```



II ZADANIE 2 z prezentacji na następny wykład.

Diagramy sekwencji:

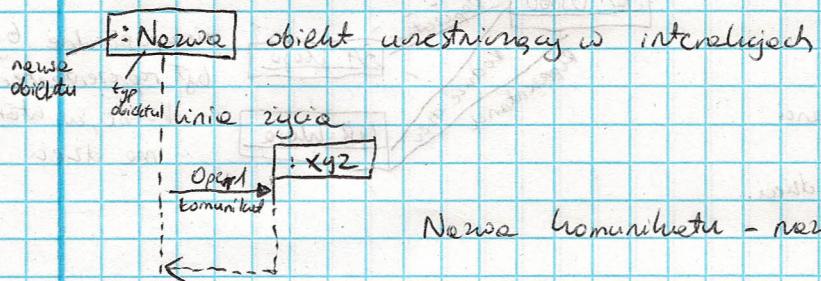
- Jak należy realizować diagramy use-case?

Use case \leftrightarrow diagramy sekwencji

Diagramy sekwencji (interakcji)

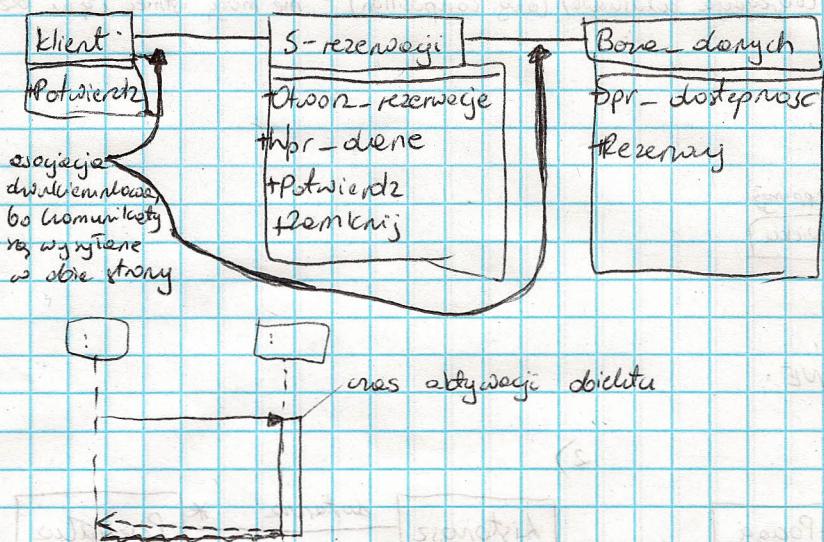
- modelują dynamiczne cechy systemu

- pomoc do tworzenia diagramów stanów i do testowania końcowego programu
- każdy pojedynczy diagram dotyczy jednej ścieżki wywrotienia programu

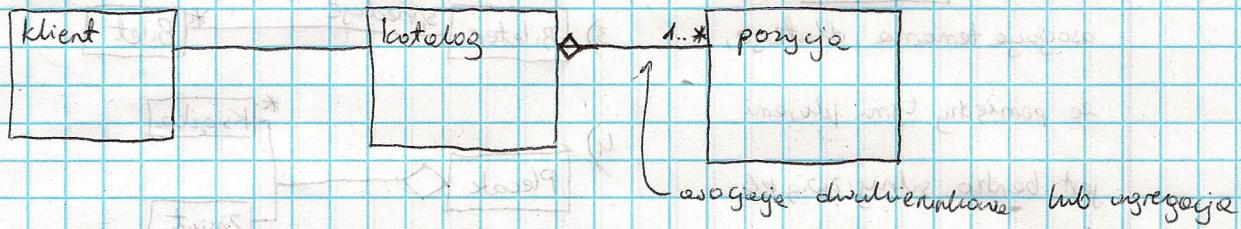


Nerwa komunikatu - nerwa metody w klasie odbierającej

Słajd 8 - przykład (wpisanie operacji)

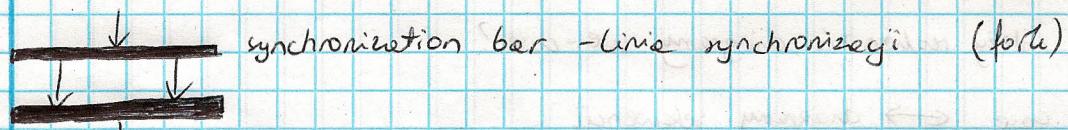
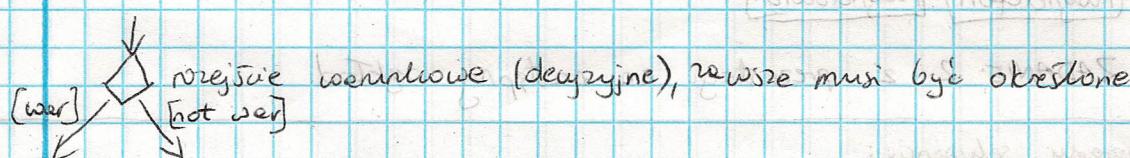
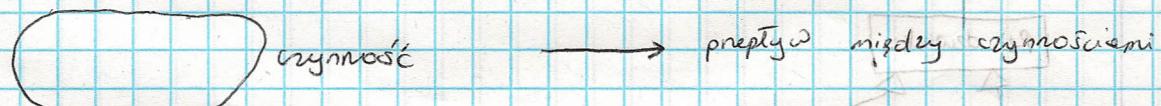


Słajd 28 - przykład (wpisanie diagramu klas)



Diagramy czynności (ang. activity diagrams)

- opisują dynamicę systemu



Petycie diagramów czynności.

Diagramy maszyny stanowej:

Model zachowania klasę

diagram maszyny stanowej \leftrightarrow klasa

Graf:

Węzły - (stany) abstrakcyjne zbioru wartości atrybutów i położenia obiektu

Skierowane krawędzie +

Nowo stanu - przyniötnik

[wewnętrzny]

ZADANIE: (stan z kolejki 12)

Schwengiż edarenie: 1) utworzenie obiektu, 2) E1, 3) E3, 4) E1, 5) E2

edarenie	stan oryginalny	schwengiż czynności
wt. obiektu	stan1	(entry) aktyw1; aktyw3
E1	stan2	(exit) aktyw2; aktyw4; aktyw7; aktyw9
E3	stan2	(włączenie) aktyw10; aktyw9
E1	stan 1	(exit) aktyw8; aktyw4; aktyw1; aktyw3
E2	stan 1	(exit) aktyw2; aktyw5; aktyw1; aktyw3

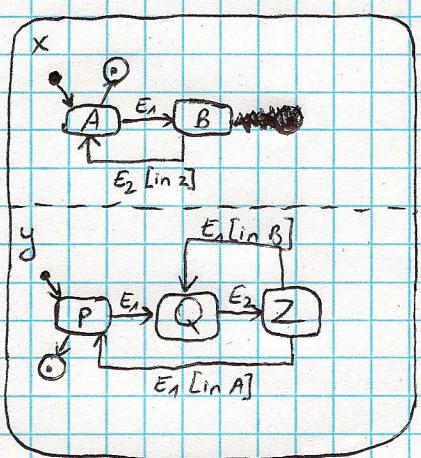
Jeśli pojawi się edarenie niewieliwane w danym stanie, to:

edarenie nie w stanie ten sam aktyw dla do
np. E10 stan1 aktyw3

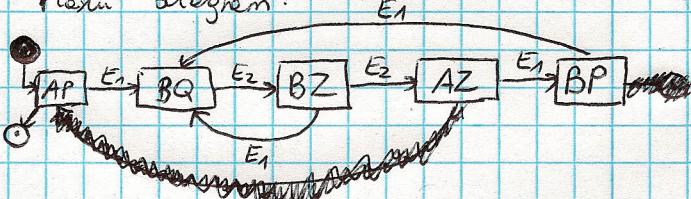
Generowijo stanów (relacja OR)

Agregacja stanów: (relacja AND)

Zadanie:



"Płaski" diagram:



Mechanizm historii:

Słajd 20: Normalnie: $A \xrightarrow{\text{In}} C \xrightarrow{\text{Out}} B \xrightarrow{\text{In}} A \xrightarrow{\text{In}} C$

Z (1): $A \xrightarrow{\text{In}} C \xrightarrow{\text{Out}} B \xrightarrow{\text{In}} A \xrightarrow{\text{In}} C$

Diagram komunikacji:

Ponosiże wzajemną komunikację między obiektami.

Diagramy implementacyjne:

Testowanie pełnotow:

- Przedzienie projektantem

- Dekompozycja systemu.

Diagramy pełnotow.

Diagram rozbiorczości (instalacyjny, implementacyjny)

06.12.2012
IND-W

* Podjście inkrementalne: (do testowania)

* Testowanie następujące T-D (top-down):

- rozpoczęty od komponentów najbardziej abstrakcyjnych

- postępujący się w góre

ZALETY:

- Szybkie wykrycie błędów projektowych

- Prejący system dostępny we wczesnej fazie rozwoju

WADY:

- Trudne do realizacji symulacji niższych warstw.

* Testowanie następujące B-U (bottom-up)

- rozpoczęty od komponentów fundamentalnych

- postępujący się w góre

- potrzebne są driversy testów

WADY:

- budżeting błędów architektonicznych następuje późno, co powiązuje ze sobą duże koszty

ZALETY:

- Indywidualne obiekty mogą być testowane z wewnętrznymi driverami testów.

* Testowanie stresowe (stress testing)

- Firmaje testowania stresującego - badanie jeli następstw przekroczenia warunków programu nie zostały zarejestrowane, sprawdzenie czy nie ma defektów.

* Testowanie porównawcze (bech-to-bech)

- stosowane, gdy dostępne więcej niż 1 wersje systemu.

* Testowanie uzupełnianie po poszukiowaniu defektów w programie.

TEST	- cel testu (scenariusz)
	- dane wyjściowe
	- dane wyjściowe (wyniki)

Po wykryciu błędu:

- lokalizacja błędu
- reprojektować poprawkę
- implementować poprawki
- wykonać ponownie cały zbiór testów. ☐

Testowanie wykorzystujące - nie jest możliwe, by przebadać każdą instrukcję.

- Podaje się do „defekt testing”:

- Funkcjonalne - (black box) testy wykonywane ze specyfikacji. $\xrightarrow{\text{WE}} \xrightarrow{\text{WY}}$ (bez egzaminu)
- Strukturalne - (white box) testy wykonywane na podstawie znajomości struktury programu $\xrightarrow{\text{WE}} \xrightarrow{\text{WY}}$ (ogreniczenie - nie za duża ilość kodu.)

$$\text{Efektywność testowania} = \frac{\text{liczba wykrytych defektów}}{\text{jumlah testów}} \quad (\text{wysoka efektywność nie oznacza testowania funkcjonalnego})$$

Analiza staticzna (code review) - przeglądanie kodu, recenzowanie kodu

a) review (staticzne)



b) testowanie (dynamiczne)

Testowanie interfejsów

Podział na klasy równoważności (equivalence partitioning)

Nazywanie poliglota (anglistyczny)

Testowanie ścieżek - musi się znać co najmniej jednej ścieżki.

Kryteria testowania poliglota danych

Miary niezawodności oprogramowania:

MTTF i MTBF (mean time to/between failure)

POFOD - prawdopodobieństwo błędu zadanego dostępu

ROCOF - współczynnik pojawiania się błędu (częstotliwość nieoczekiwanych zachorzeń systemu)

AVAIL (availability) - dostępność

Wymagania niezawodnościowe są określone nietorfemnicie, a do pomiarów używają się testów statystycznego.

Modele wzrostu niezawodności:

- Funkcja jednolatowego kroku

- Funkcja losowego kroku

- Modele ciągłe

Strategie:

unielenie błędów, tolerowanie błędów, detekcja błędów,

Blokis rezerwowe

Estymacja kosztów oprogramowania:

80% projektów preliminary planowane koszty o 100%

VFC

AFC

Model COCOMO (Constructive Cost Model)

$$KDSI < 30 \quad \text{Prosty: } PM = 2.4 \cdot (KDSI)^{1.05} * M$$

KDSI - Kilo Delivered Source Instructions

$$KDSI < 100 \quad \text{Średni: } PM = 3.0 \cdot (KDSI)^{1.12} * M$$

$M = 1$ (podstawowo)

$$KDSI > 100 \quad \text{Złożony: } PM = 3.6 \cdot (KDSI)^{1.20} * M$$

Zadanie (slajd 2):

Wejśc: 10, repository: 12, interakcje: 20, czas: 6 godz. 4 min.

~~Wielozmiennik~~
Projekt: (slajd 15)

$KDSI = 128$, system skomplikowany

$$\text{Podst: } PM = 3.6 \cdot 128^{1.20} \cdot 1 = 1216 \text{ obiektów/miesiąc}$$

$$\text{Param: } M = 1.4 * 1.3 * 1.2 * 1.1 \approx 1.23$$

$$PM = 3.6 \cdot 128^{1.20} \cdot 1.23 = 3593 \text{ obiektów/miesiąc}$$

Model COCOMO 2 (mniej nie będzie na egzaminie): (może być np. liczenie wykładek)

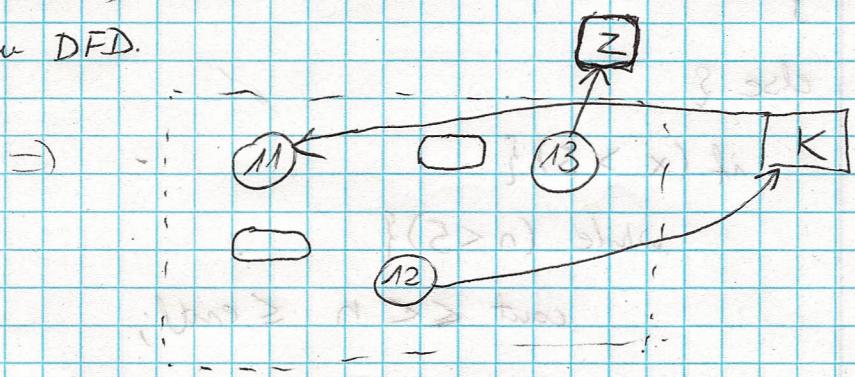
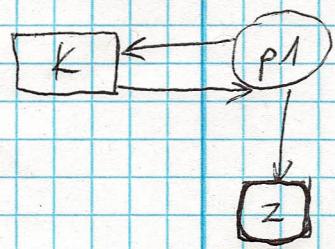
Metody strukturalne opierają się na wyznaczeniu w analizowanym systemie strukturalnych obiektów i perywów.

Model przepływu danych (data flow) - DFD - data flow diagram.

Nengdzie modelowanie systemu:

- diagramem przepływu danych (DFD)
- diagramem zwierciadła encji (ERD)
- diagramem sieci przepięć (STD).

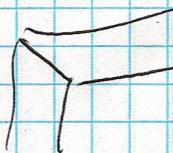
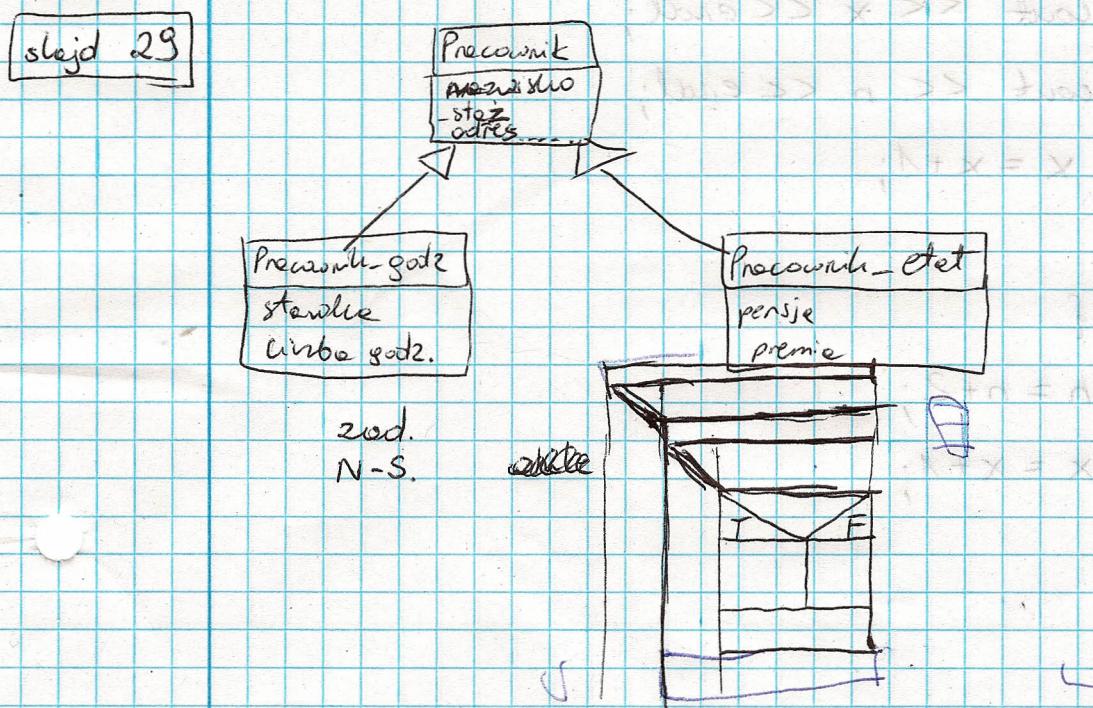
DFD nie pokazuje kolejności wykonywania procedur. Tworzymy hierarchię DFD - rysujemy osobne diagramy (berając szczegółowe) dla każdej funkcji z podstawowego diagramu DFD.



poziom ogólny

poziom szczegółowy.

Diagram zwierciadła encji można zmodyfikować w UML'u ze pomocą diagramu klas.



+ tutaj

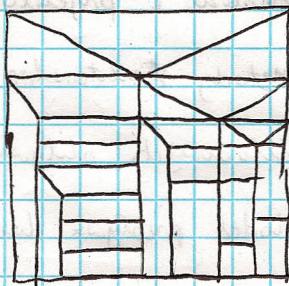
$HPC = \pi$

```
int x=1 n=0
```

```
if (x == 1) { ... }
```

```
while (n < 10) {
```

E



```
    cout << n << endl;
```

```
    cout << x << endl;
```

```
    while (n < 10) {
```

```
        cout << n << endl;
```

```
        cout << x << endl;
```

```
        n++;
```

```
}
```

```
}
```

```
else {
```

```
    if (x > 5) {
```

```
        while (n < 5) {
```

```
            cout << n << endl;
```

```
            n++;
```

```
}
```

```
else {
```

```
    if (x > 2) {
```

```
        cout << x << endl;
```

```
        cout << n << endl;
```

```
        x = x + 1;
```

```
}
```

```
else {
```

```
    n = n + 2;
```

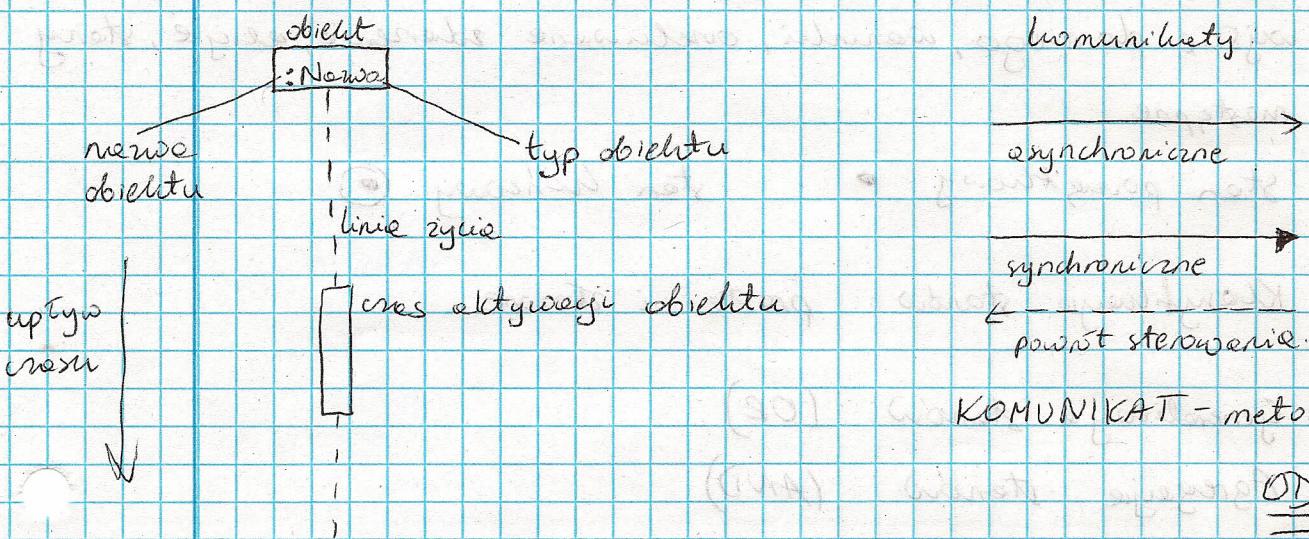
```
    x = x + 1;
```

```
}
```

```
}
```

Diagramy sekwencji:

- modelują dynamiczne cechy systemu
- pomoc do tworzenia diagramów stanów
- pojedynczy diagram dotyczy 1 sekwencji wywołania programu



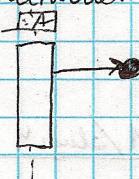
KOMUNIKAT - metoda o kierunku

ODBIERAJĄCEJ!

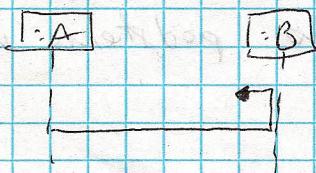
Semowy zadek:



Komunikat utwórzony (niewymy obiekta):



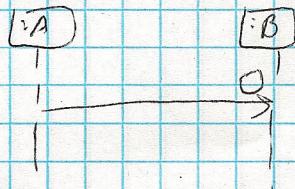
Komunikat opcjonalny
(do natychmiastowej obtruzji):



Komunikat zadekowany (niewymy nadawca):



Komunikat określony (z określonym makiem i wewnątrz)



Oznaczenie:

Ⓐ lincze stojące,

Ⓑ lincze przechowujące

♀ lincze gniazna

Operatory interakcji: alt (alternatywa), opt (copyje), break, loop, ref

Diagram maszyny stanowej:

- opisuje zmiany stanów obiektu na skutek zdarzeń

Stan: reprezentuje obiektu na zdarzenie

charakteryzowany przez nowe, skutki zdarzeń powodujących wyjście do niego, wejścia, określone zdarzenie, elice, stany mestypne

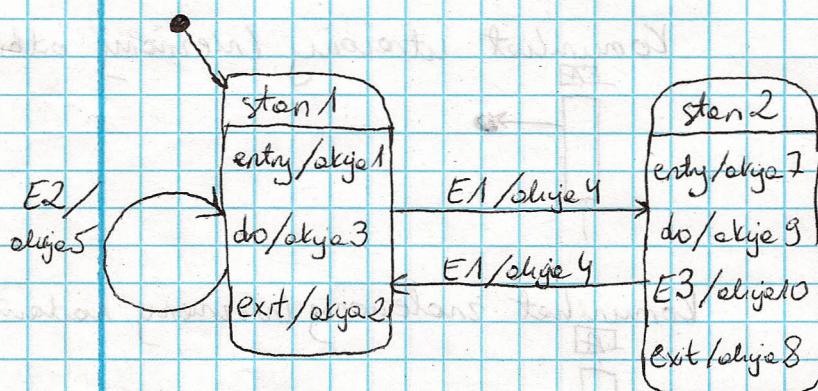
Stan początkowy •

stan końcowy ①

Klasyfikacja stanów: proste i złożone.

Generalizuje stanów (OR)

Agreguje stanów (AND)



utworzenie obiektu: entry + do

zdarzenie wej.: zdarzenie + do

zdarzenie wyj.: exit + no przejścia
+ entry + do

zdarzenie nieaktywane: do

Mechanizm historii: zapamiętuje ostatnio odwiedzony stan w podstanie i przy ponownym wejściu do podstania wróci do tego samego stanu.