

Inżynieria oprogramowania

Wykład 1: Wprowadzenie

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Agenda

- Czym jest inżynieria oprogramowania?
- Proces tworzenia oprogramowania
- SWEBoK

Znaczenie

- Mały projekt = mały problem
 - mały – wystarcza programowanie
 - większy – modelowanie, opanowanie złożoności, zarządzanie

Projekty wykorzystujące metody inż. opr.

- zaangażowanie zespołu ze zmieniającym się składem (fluktuacja)
- wymagania trudne do uchwycenia i wyrażenia, zmieniające się
- eksploatacja oprogramowania trwa wiele lat
 - ewolucja
- złożone przetwarzanie
 - skutki awarii rozległe i dotkliwe

Przykład – system dla banku

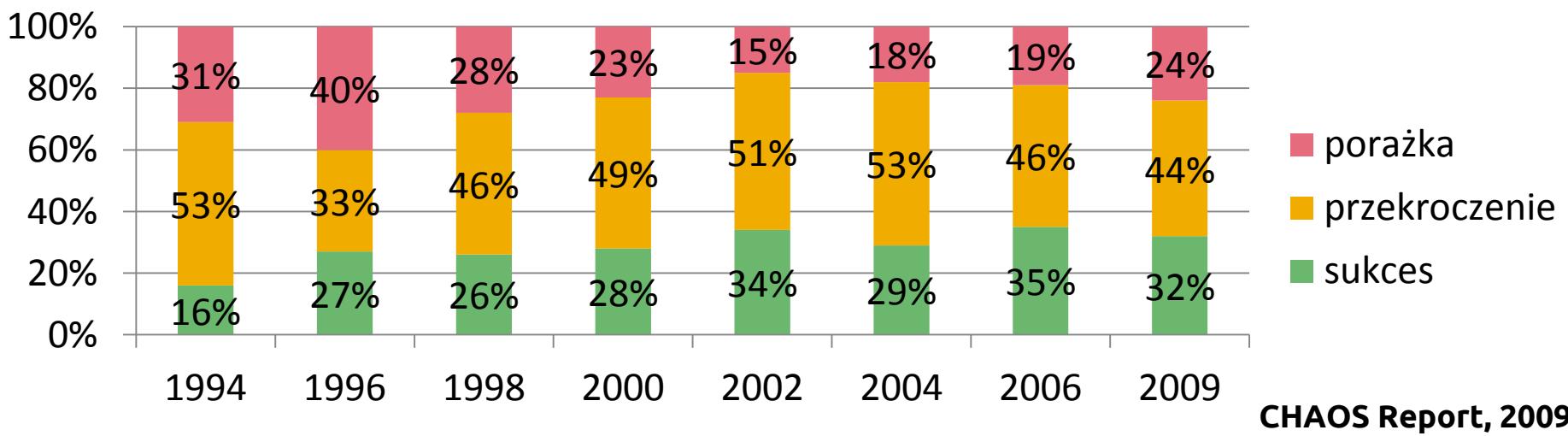
- rachunki setek tysięcy klientów
- proces tworzenia 2-3 lata
- zaangażowanie ok. 100 osób
- wiele danych i informacji – nie sposób ich zapamiętać
 - potrzebna dekompozycja → modele
- zmieniające się wymagania
 - nowe i zaktualizowane produkty
 - przepisy prawne
- koszt ponad 100 mln zł – zwrot po kilkunastu/kilkudziesięciu latach
 - w tym czasie zmiany → inni ludzie
- awaria → wiele oddziałów, klientów
 - brak dostępu do pieniędzy
 - w efekcie nawet bankructwo banku

Inżynieria oprogramowania

- Cel: wypracowanie skutecznych sposobów **budowania i wdrażania wielkich** systemów informatycznych w przewidzianym **terminie** i z rozsądnym **kosztem**
- Czy cel zrealizowany?
 - wiele zbudowanych i wdrożonych systemów
 - oprogramowanie niemal w każdej dziedzinie życia
- Ale duży odsetek projektów
 - anulowanych
 - przekraczających czas i/lub budżet

Rezultaty projektów

- sukces
 - ukończone w terminie i w ramach budżetu
- przekroczenie
 - znaczne przekroczenie budżetu lub czasu wykonania albo ograniczenie funkcjonalności
- porażka
 - porzucone bez zakończenia



Oprogramowanie

- twór niematerialny
 - elementów nie można obejrzeć i ocenić za pomocą zmysłów
- wymagania często złożone i różnorodne
 - trudno je określić
 - ocenić stopień ich wykonania
- podatne na zmiany
 - niespotykana w innych dziedzinach zmienność wymagań
- koszt i czas skupiają się w procesie projektowania
 - masowa produkcja niemal nie zwiększa czasu ani kosztu
- projekty w dużym stopniu niepowtarzalne, a techniki wytwarzania nowe
 - trudne oszacowanie czasu i nakładów
- wiedza dziedzinowa odległa od informatycznej
 - trudności w komunikacji zamawiający-wykonawca

Inżynieria oprogramowania – definicja

praktyczne zastosowanie wiedzy naukowej

do projektowania i tworzenia

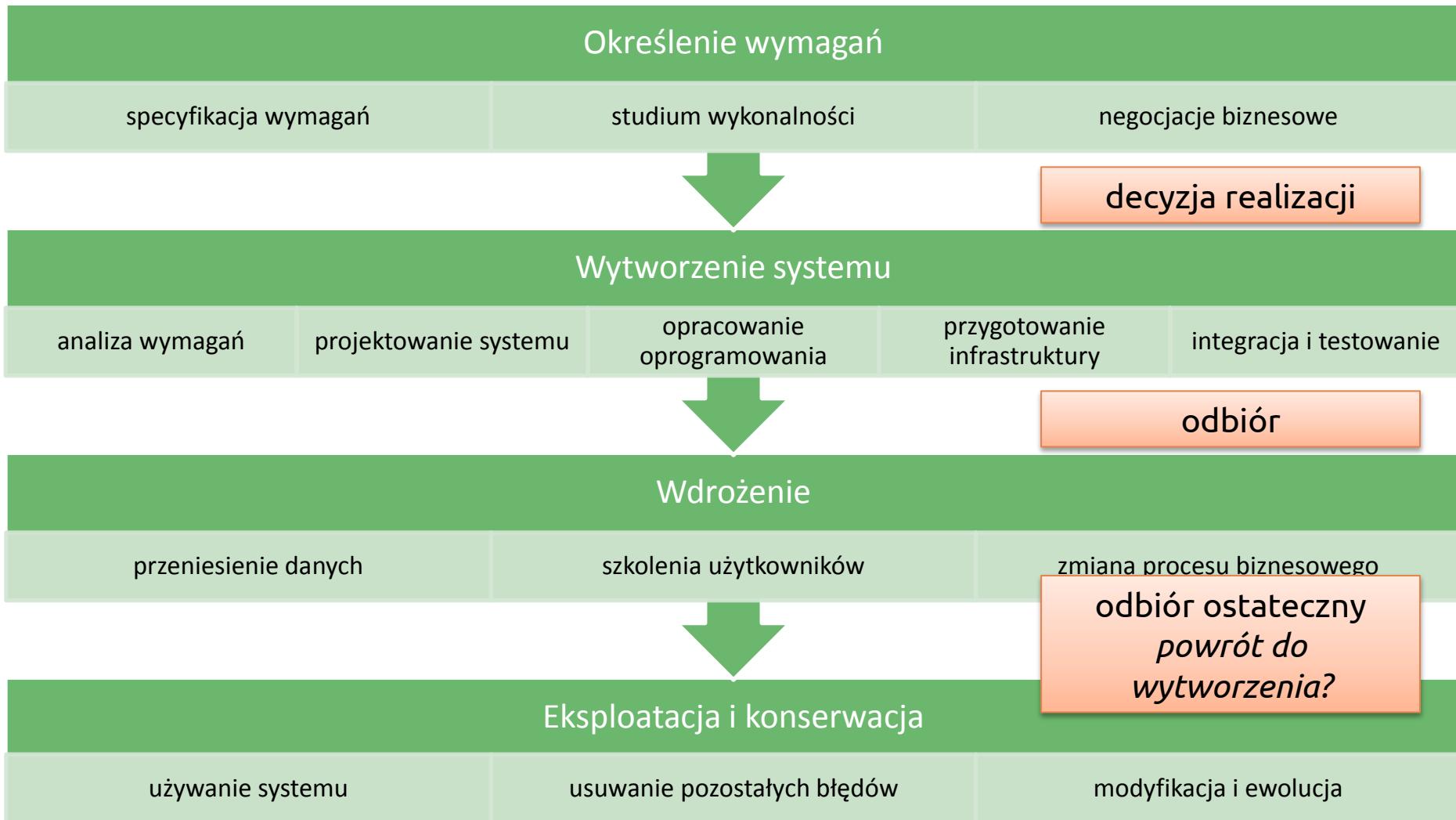
systemów informacyjnych i
informatycznych

oraz

dokumentacji wymaganej do ich
opracowania, uruchomienia i pielęgnacji

Proces tworzenia oprogramowania

Proces rozwoju systemu informatycznego



Dwa rodzaje projektów/programowania

na zamówienie konkretnego odbiorcy

- wymagania od **zleceniodawcy**
- decyzja o realizacji – często w drodze **przetargu i/albo negocjacji**
- wytwarzanie – implementacja **wymaganych** funkcji i budowa **infrastruktury** jak najniższym kosztem
- **zleceniodawca** dokonuje odbioru – testowanie akceptacyjne
- **wdrożenie** w docelowym środowisku
- późniejsze zmiany inicjowane przez **użytkowników**
- rzadka możliwość wdrożenia w innym środowisku

dla masowego odbiorcy

- wymagania określa **producent**
 - działanie w typowych konfiguracjach
 - brak potrzeby budowania specjalnej infrastruktury
- brak niezależnego odbioru
 - testowanie **alfa** przez twórcę
 - testowanie **beta** u wybranych odbiorów
 - wydanie **pierwszej** wersji
- wdrożenie – podobnie jak zamawiane
- konserwacja – **brak konieczności reakcji** na indywidualne potrzeby użytkowników
- wytwarzane i oferowane wielokrotnie

Określenie wymagań

- kluczowe dla przedsięwzięcia
- specyfikacja mówi projektantom
 - co ma być zrobione
 - jaki system/oprogramowanie ma powstać
- odbiór następuje przez sprawdzenie zgodności działania ze specyfikacją
 - jeśli wymagania nie odzwierciedlają rzeczywistych potrzeb → system może być niepotrzebny

Określenie wymagań, przykład - bank

- Czy jedynie konta klientów, którzy lokują swoje środki? Czy również obsługa klientów bez kont?
- Jeśli kredyty, to:
 - czy tylko kontrola i księgowanie rat?
 - czy także ocena ryzyka kredytowego?
 - czy także automatyzacja obsługi?
- Jak ma przebiegać proces przyznania kredytu?
 - na podstawie jakich danych?
 - kto decyduje o przyznaniu – system czy pracownik?
- Jak naliczane oprocentowanie kont i kredytów – tygodniowo, miesięcznie, itd.?
 - czy obliczana efektywna stopa procentowa?
- Czy konta przechowywane w centralnej bazie czy w lokalnych bazach oddziałów?
 - co w przypadku przerwania łączności oddziału z centralą?
- Co może / nie może zdarzyć się w przypadku awarii?
 - na pewno brak utraty danych
 - jak szybko wznowienie działania – natychmiast, kilka godzin, dni...?

Określenie wymagań

- Kto ma udzielić odpowiedzi na takie pytania?
 - projektant
 - użytkownik
 - możliwe duże koszty realizacji wymagań, które nie są niezbędne
- Uzgodnione wymagania są częścią decyzji realizacji oprogramowania → umowa
 - często również metody sprawdzenia stopnia spełnienia

Określenie wymagań

- Określenie i zatwierdzenie – **nie jest jednorazowe!**
 - zmiany w biznesie, prawie, technologiach
 - modyfikacje → nowe błędy i konieczność weryfikacji zmienionych elementów
- Konieczne **zarządzenie zmianami**
 - sprawdzanie celowości nowych wymagań
 - sprawdzenie sprzeczności z innymi wymaganiami
 - grupowanie zmian → łączna realizacja obniża koszty
- Kto zarządza zmianami?
 - ocena celowości i ryzyka biznesowego – użytkownik
 - ocena kosztu i konsekwencji – wykonawca

Główna faza: Wytwarzanie oprogramowania

Wytwarzanie oprogramowania

4 rodzaje działań

Analiza

analysis

Projektowanie

design

Implementacja

implementation

Weryfikacja i
zatwierdzanie

verification and validation

Wytwarzanie oprogramowania

4 rodzaje działań

Analiza

- poznanie i opisanie problemu określonego w wymaganiach, **wskazanie elementów i powiązań**
- zdefiniowanie tego, **co oprogramowanie ma robić**, a nie jak ma być zbudowane
- obejmuje badanie dziedziny zastosowania, zrozumienie potrzeb, wypracowanie koncepcji rozwiązania, budowa modelu opisującego sposób spełnienia wymagań
- model określa wszystkie funkcje i działania, dane, algorytmy i ograniczenia wykonania funkcji
- model nie pokazuje budowy ani technologii wykonania funkcji
- sposób wykonania i udokumentowania zależą od metody tworzenia oprogramowania

Wytwarzanie oprogramowania

4 rodzaje działań

Projektowanie

- **przekształcenie niezależnego od technologii opisu działania (modelu) w schemat budowy**
- obejmuje wyznaczenie podstawowych elementów, przypisanie im funkcji określonych podczas analizy, wybranie odpowiedniej technologii informatycznej, stworzenie modelu opisującego szczegóły budowy
- model powinien określać strukturę programów, strukturę danych, sposób współdziałania elementów (w danej technologii)
- model projektowy nie zawiera szczegółów określanych dopiero w kodzie
- sposób wykonania i natura elementów zależą od metody projektowania i technologii implementacyjnej

Wytwarzanie oprogramowania

4 rodzaje działań

Implementacja

- **przekształcenie schematu budowy oprogramowania (modelu projektowego) w działający kod programu**
- obejmuje napisanie i uruchomienie wszystkich elementów, połączenie w działający system, przygotowanie testów sprawdzających poprawność działania
- często również: dokumentacja użytkowa i techniczna

Wytwarzanie oprogramowania

4 rodzaje działań

Weryfikacja i zatwierdzanie

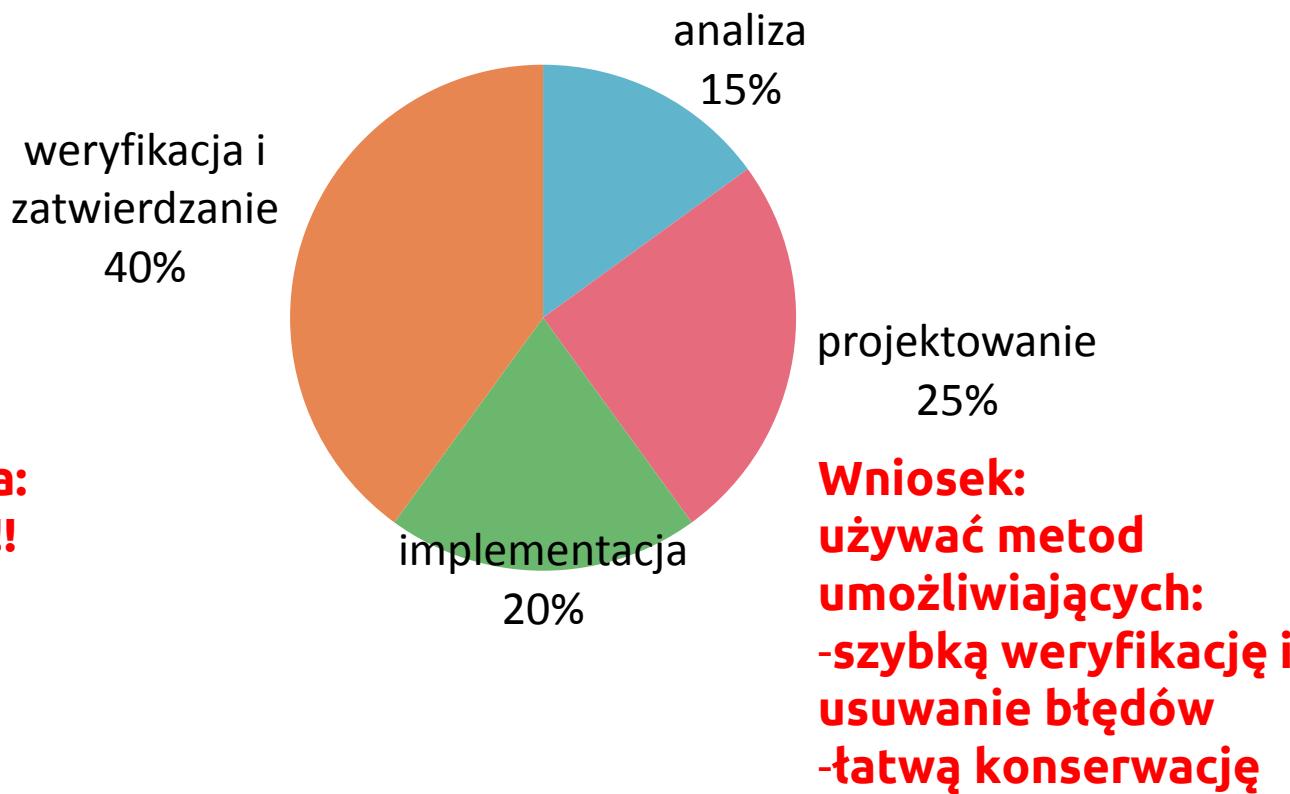
- **cel weryfikacji:** kontrola prawidłowości wykonania działań i poprawności wytwarzanych wyników
- **cel zatwierdzania:** sprawdzenie zgodności produktu z potrzebami użytkowników
- sposób wykonania zależy od postaci produktu
 - dokumenty analityczne i projektowe → weryfikacja na podstawie przeglądu ich treści
 - program → sprawdzenie eksperymentalne – testowanie w działaniu (funkcje, moduły, komponenty, cały system)
 - ostateczna ocena i zatwierdzenie – podstawa do odbioru produktu

Wytwarzanie oprogramowania dodatkowa czynność – **konserwacja**

- obejmuje wszelkie zmiany po rozpoczęciu eksploatacji
 - usuwanie późno wykrytych błędów
 - wprowadzanie poprawek i modyfikacji
 - zmiany w świecie biznesowym
- obejmuje czynności analizy, projektowania, implementacji, weryfikacji i zatwierdzania
- konserwacja a oryginalne opracowanie
 - konieczność ingerencji w działające oprogramowanie
 - identyfikacja budowy i działania oprogramowania
 - stała analiza potencjalnego wpływu zmian na inne części

Wytwarzanie oprogramowania

- Jaka waga i udział poszczególnych działań?
 - potrzebne do planowania
 - odpowiednie rozłożenie zasobów

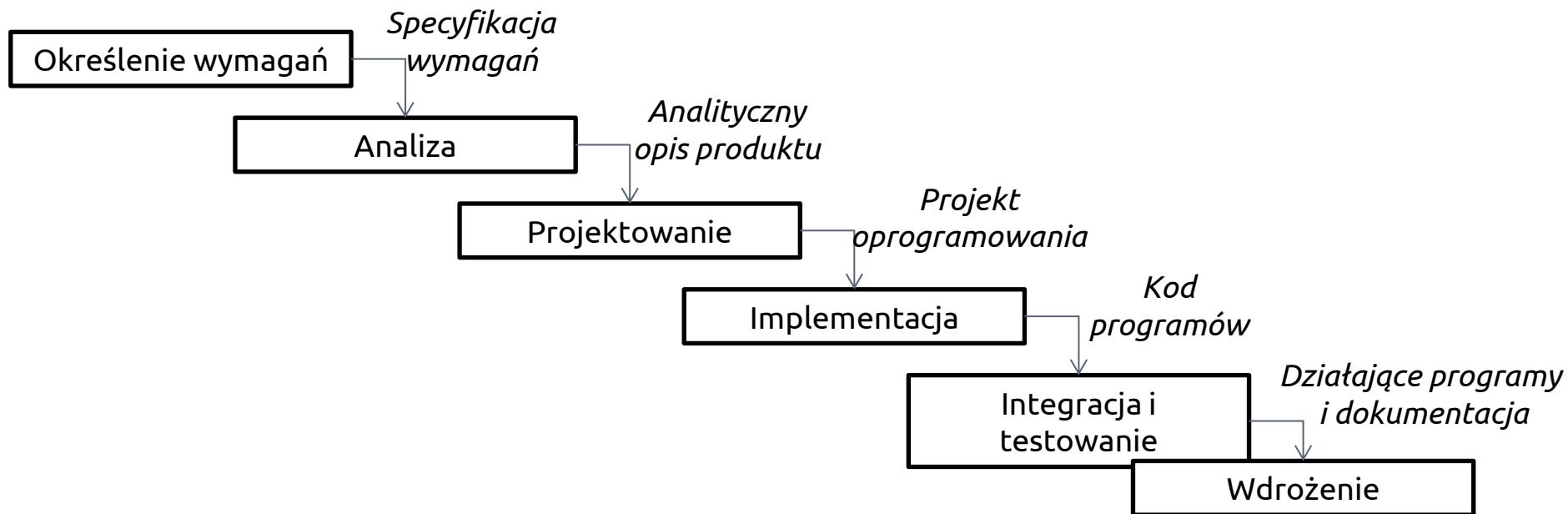


Wytwarzanie oprogramowania procesy

- **Proces wytwarzania oprogramowania**
 - ang. *software development process*
- **układ faz**, działań widocznych z poziomu zarządzania projektem
 - następujących po sobie
 - nakładają się na siebie
 - powtarzają się
- Nie ma jednego **uniwersalnego** procesu!
- Różne modele:
 - proces kaskadowy
 - proces iteracyjny
 - proces zwinny
 - proces komponentowy

Wytwarzanie oprogramowania proces kaskadowy

- zasadnicze działania uporządkowane w sposób **szeregowy**
 - tworzą ciąg następujących po sobie faz
 - brak powracania do wcześniejszych faz

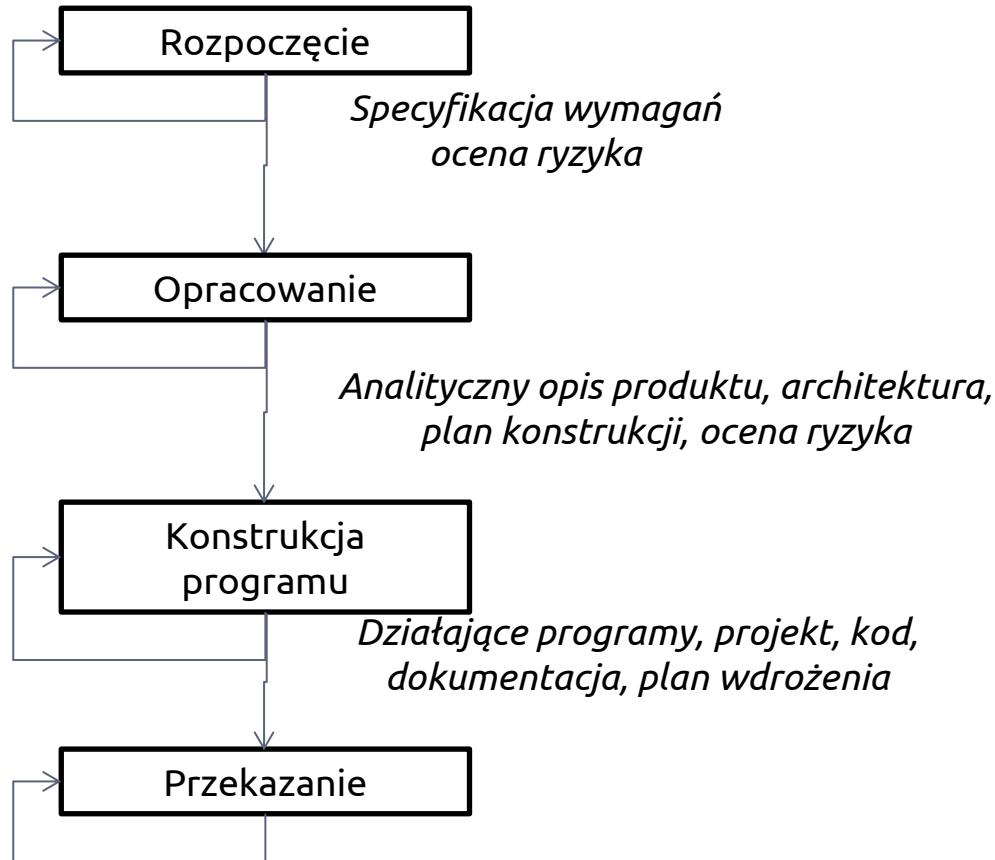


Wytwarzanie oprogramowania

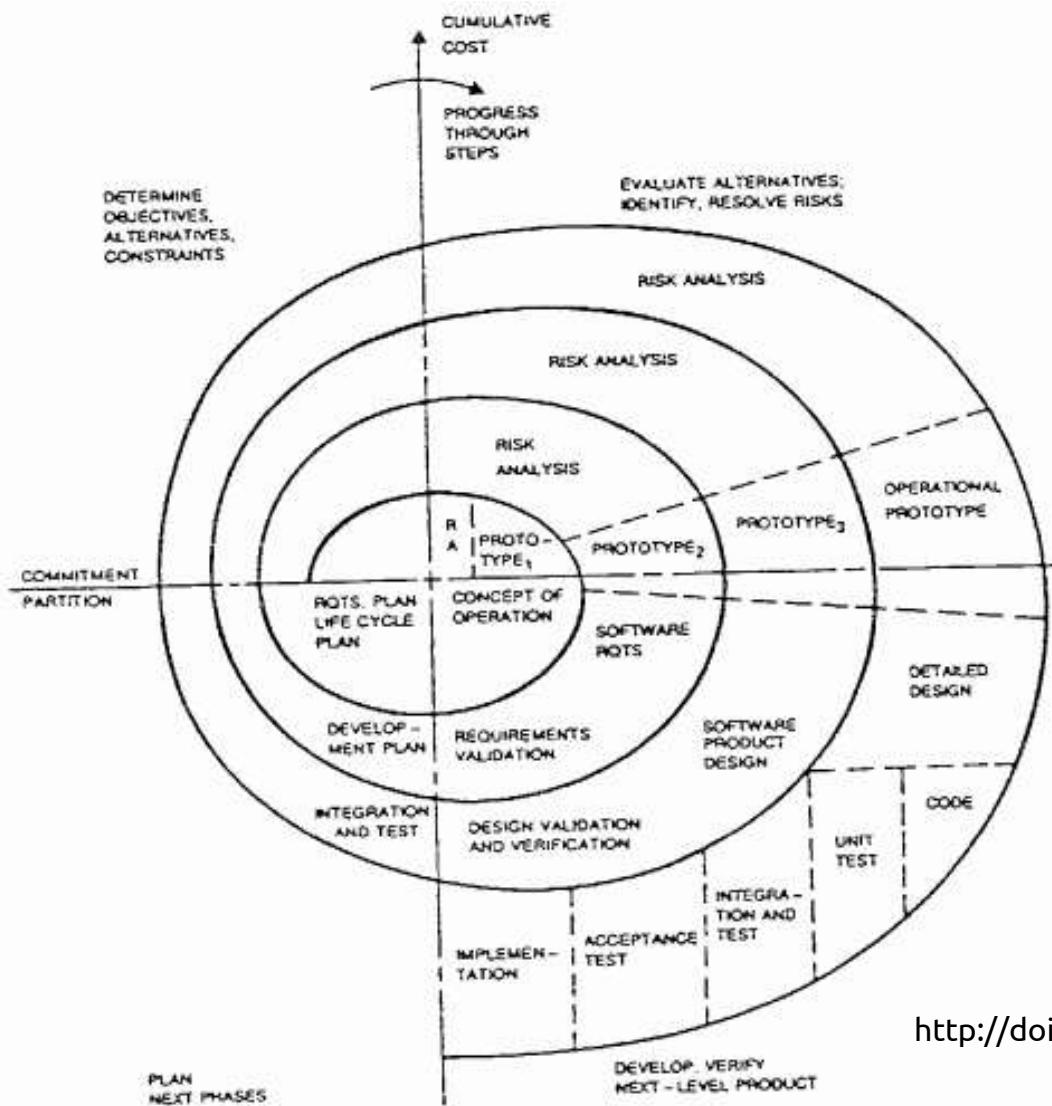
proces iteracyjny

- brak rozmieszczenia poszczególnych działań w konkretnych fazach
 - iteracyjne powtarzanie wszystkich rodzajów działań
 - lub przynajmniej ich elementów
- wykonanie każdej fazy może wymagać kilku iteracji
- każda iteracja zwiększa wiedzę o wszystkich aspektach oprogramowania
 - przyczynia się do lepszej oceny ryzyka zakłóceń
- podział na fazy odzwierciedla kolejność podejmowania decyzji o angażowaniu coraz większych środków

Wytwarzanie oprogramowania proces iteracyjny

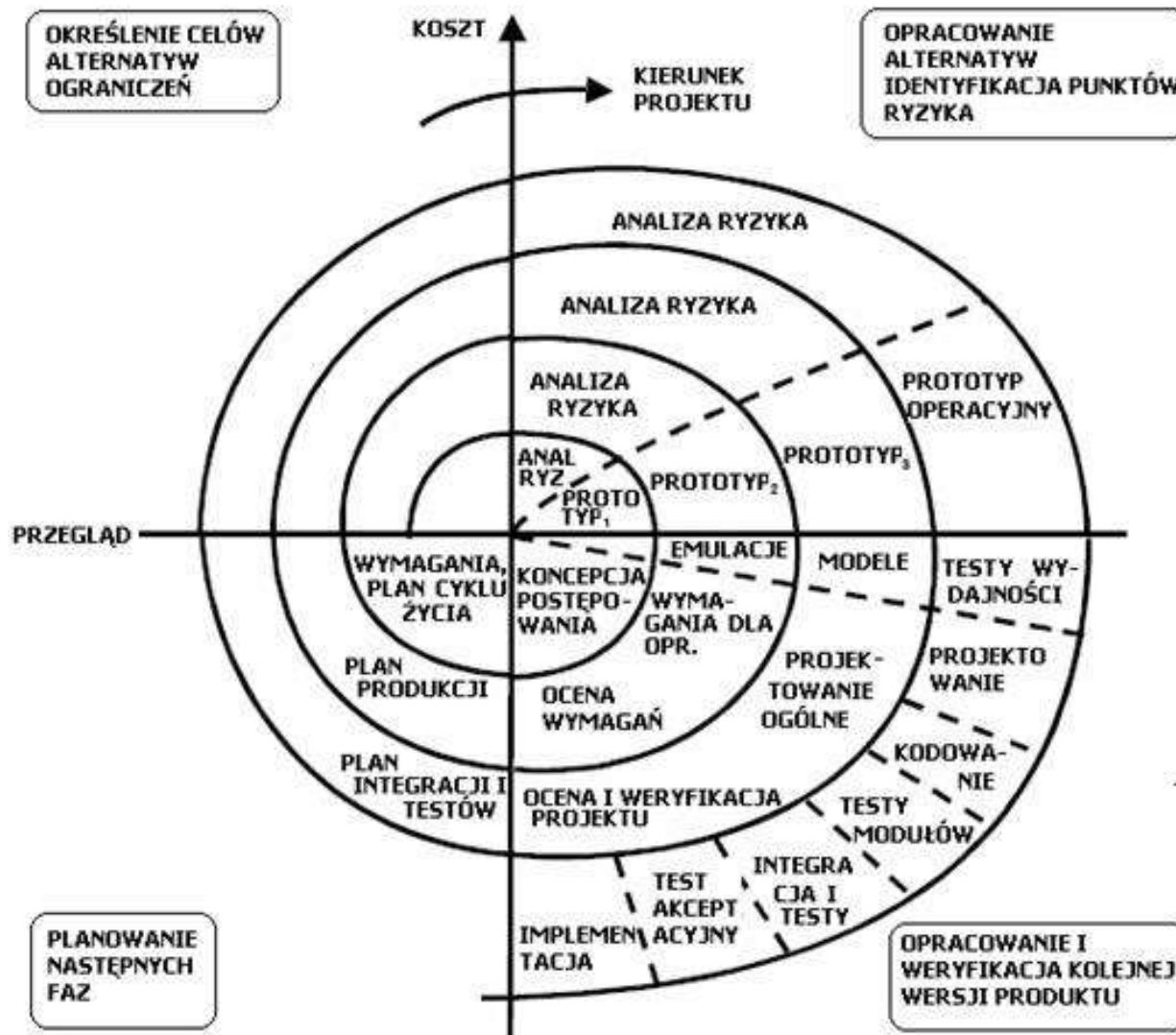


Wytwarzanie oprogramowania szczególny proces iteracyjny – spiralny



<http://doi.acm.org/10.1145/12944.12948>

Wytwarzanie oprogramowania szczególny proces iteracyjny – spiralny



Wytwarzanie oprogramowania proces kaskadowy i iteracyjny

- obejmują działania w kontekście **całości systemu**
 - całość wymagań
 - projekt architektury całego systemu
 - dopiero wtedy pisanie kodu
- **wady**
 - monstralna dokumentacja
 - trudna i kosztowna realizacja systemu
 - opracowanie i utrzymanie aktualnej dokumentacji
 - zmiana wymaga modyfikacji wielu dokumentów, weryfikacji, zatwierdzania

Wytwarzanie oprogramowania proces zwinny (*agile*)

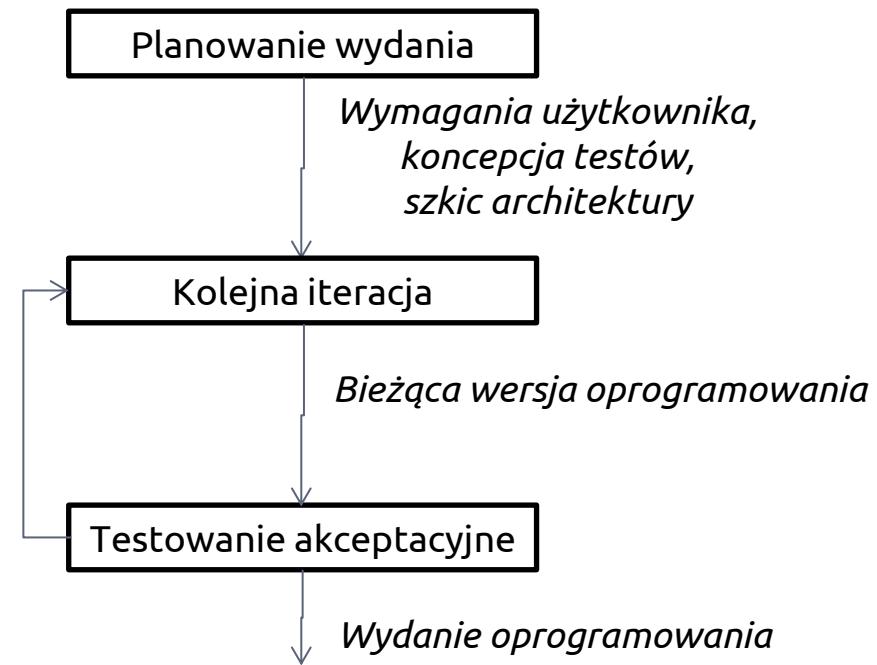
- **porzucenie całościowego sposobu postrzegania i modelowania rozwiązań**
- szybka implementacja **bieżących wymagań**
- **lokalna** optymalizacja w odpowiedzi na zmiany
- motywacja: skoro nie da się przewidzieć całości wymagań i rozwiązań, **nie warto tworzyć i dokumentować modeli**
- zamiast tego:
 - szybkie tworzenie produktu realizującego część potrzeb
 - później rozwijać dalej zgodnie z nowymi wymaganiami/potrzebami

Wytwarzanie oprogramowania proces zwinny (*agile*)

- horyzont czasowy obejmuje co najwyżej **kilka miesięcy**
 - czas na opracowanie kompletnego **wydania** (*release*)
- większy produkt powstaje w ciągu **kilku wydań**
 - bez określenia ilu, bo zawsze planowane jest tylko jedno – bieżące
- zakres wydania określany przez użytkownika w czasie negocjacji – planowania wydania (*release planning*)
 - zwięzły opis wymagań
 - opis testów akceptacyjnych
 - szkic architektury programów
- później
 - rozmowa z użytkownikiem (stale obecnym w zespole)
 - opis testów

Wytwarzanie oprogramowania proces zwinny (*agile*)

- proces przebiega iteracyjnie
 - z góry określona liczba iteracji
- stały czas trwania iteracji
 - zwykle 2-3 tygodnie
- zakres działań w iteracji
 - ustalany w chwili jej rozpoczęcia
 - brane pod uwagę preferencje użytkownika i możliwości deweloperów
 - ludzie nie są wymienni
 - mają różne predyspozycje do wykonywania określonych działań
- podczas iteracji
 - programy realizujące określone funkcje
 - testy akceptacyjne



Wytwarzanie oprogramowania proces zwinny (*agile*)

- **jedyny miernik postępu prac → przyrost działającego kodu**
- proces:
 - najpierw implementacja **testów**
 - potem sam **program**
- jeśli jednostka programowa nie przejdzie testów, to praca nie jest zakończona
- jeśli testy pomyślne → jednostka integrowana z resztą kodu
- dzięki temu zawsze istnieje działająca wersja

Wytwarzanie oprogramowania proces zwinny (*agile*)

- każda iteracja kończy się **po wyczerpaniu czasu**
- funkcje, które przechodzą testy akceptacyjne
 - zaliczane do danej iteracji
- funkcje, które nie przechodzą testów lub w ogóle nie zostały zaimplementowane
 - przesuwane do kolejnej iteracji
- stały rytm czasowy umożliwia precyzyjną **ocenę postępów prac**
 - w razie potrzeby – podejmowanie działań korygujących

Wytwarzanie oprogramowania proces zwinny (*agile*)

- pozwala na:
 - istotne zmniejszenie kosztów
 - zapewnienie wysokiego poziomu satysfakcji użytkowników
- szybkie tworzenie kodu, powiązane od razu z testowaniem → szybkie wykrywanie i naprawianie defektów
- utrzymanie działającej wersji → stała kontrola i ocena rezultatów przez użytkowników

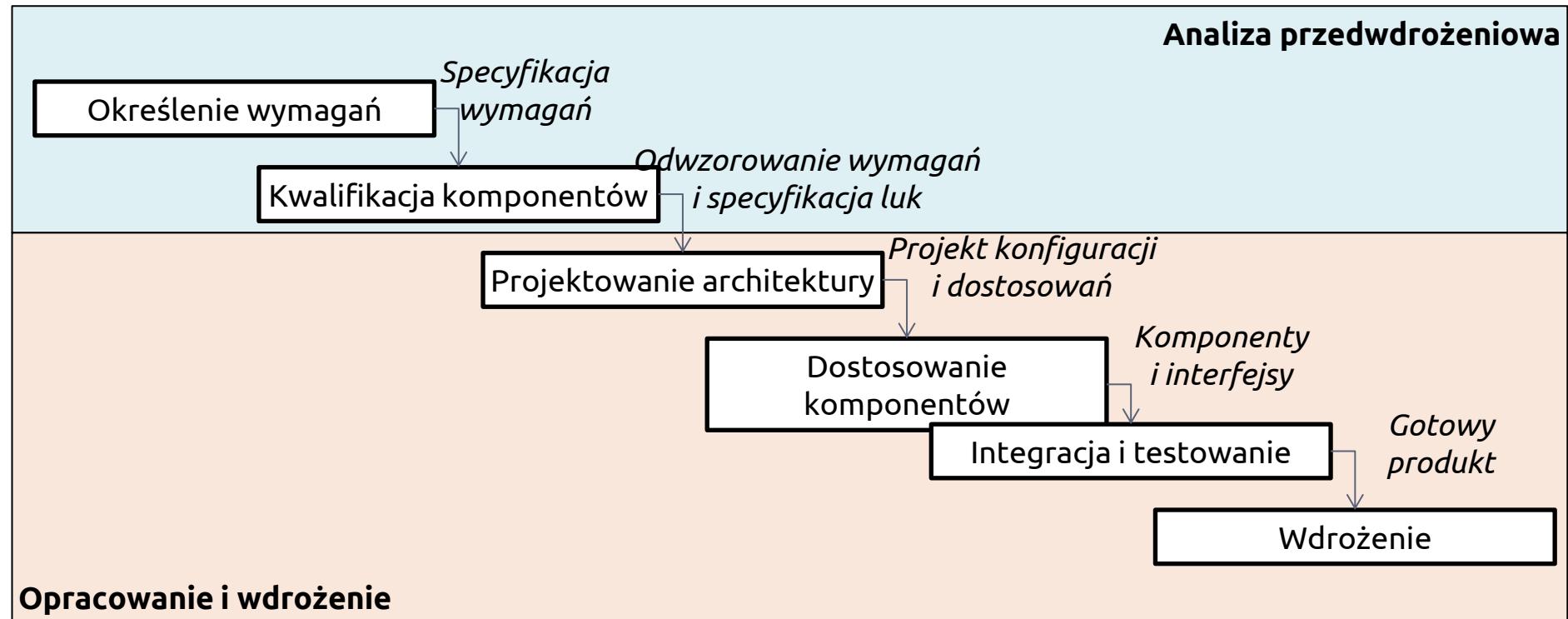
Wytwarzanie oprogramowania proces zwinny (*agile*)

- wady:
 - krótki horyzont planowania
 - potrzebne zaufanie do wykonawcy
 - akceptacja obietnicy wykonania systemu w ciągu kilku wydań (**ilu?**)
 - praca z niepełną dokumentacją
 - zastępowaną przez bezpośrednią komunikację deweloperów i z użytkownikami – możliwe tylko w niewielkim zespole

Wytwarzanie oprogramowania proces komponentowy

- ang. *component-based process*
- nie zawsze oprogramowanie opracowywane jest w całości od początku
- wykorzystanie gotowych komponentów
 - opracowane wcześniej
 - stworzenie z myślą o wielokrotnym użyciu
- proces tworzenia zwykle zgodny z modelem kaskadowym

Wytwarzanie oprogramowania proces komponentowy



Metody wytworzania oprogramowania

Wytwarzanie oprogramowania metody

- definicja procesu **nie określa metod** wykonywania działań
- metody określają:
 - rodzaj modeli budowanych w wyniku różnych działań
 - schematy ułatwiające tworzenie modeli
 - wskazówki sugerujące prawidłową kolejność tworzenia modeli
- 2 podstawowe grupy metod:
 - strukturalne – zwykle w procesie kaskadowym
 - obiektowe – zwykle w procesach iteracyjnych i zwinnych

Wytwarzanie oprogramowania metody strukturalne

- wykorzystują **model przetwarzania danych** zawierające dwie warstwy:
 - pasywne dane – opisują stan dziedziny zastosowania
 - aktywne funkcje – przetwarzają dane zgodnie z określonym algorytmem

Wytwarzanie oprogramowania metody strukturalne

- działania analityczne obejmują
 - określenie funkcji (jako diagram przepływu danych)
 - dekomponowanych na funkcje prostsze
 - określenie struktur danych (jako diagram związków encji)
 - i ich wzajemnych powiązań
- diagramy opisują jakie dane są przetwarzane i w jaki sposób
 - ale nie określają budowy programu

Wytwarzanie oprogramowania

metody strukturalne

- działania projektowe obejmują
 - wyznaczenie podstawowych elementów programu
 - do nich zostaną przypisane funkcje zdefiniowane podczas analizy
 - określenie sposobów wywoływania i komunikowania się podprogramów
 - opracowanie struktury danych
- wynikiem jest:
 - model budowy programu (jako diagram struktury)
 - opisuje hierarchię wywołujących się podprogramów
 - definiuje sposób komunikowania się podprogramów
 - model tabel i indeksów bazy danych
- oba modele powstają przez przekształcenie wcześniejszych modeli analitycznych

Wytwarzanie oprogramowania metody strukturalne

- wady
 - konieczność zmiany rodzaju modelu podczas przejścia z analizy do projektowania
 - mała modyfikowalność implementacji
- zmiana lub dodanie nowego zachowania wymaga
 - zmiany wszystkich modeli
 - modyfikacji istniejących programów
 - powtórzenia procesu testowania

Wytwarzanie oprogramowania metody obiektowe

- opisują proces przetwarzania jako wynik **interakcji wielu autonomicznych obiektów**
- każdy obiekt reprezentuje **fragment dziedziny** i zawiera
 - porcję danych
 - funkcje przetwarzające te dane

Wytwarzanie oprogramowania metody obiektowe

- działania analityczne obejmują
 - określenie zachowań oprogramowania z punktu widzenia zewnętrznych użytkowników
 - jako diagram przypadków użycia
 - wyodrębnienie i sklasyfikowanie elementów dziedziny, których te zachowania dotyczą
 - jako diagram klas

Wytwarzanie oprogramowania metody obiektowe

- działania projektowe obejmują
 - odwzorowanie klas trwałych w tabele bazy danych
 - zdefiniowanie sposobu wykonywania przypadków użycia przez obiekty różnych klas
 - potraktowanie diagramu klas jako hierarchii dziedziczenia języka obiektowego

Wytwarzanie oprogramowania metody obiektowe

- zalety
 - elastyczność i łatwość wprowadzania zmian
 - brak radykalnej zmiany modelu przy przejściu analiza-projektowanie
 - dostępność dziedziczenia
 - dodawanie nowych zachowań bez zmieniania pozostazej części modelu lub programu
- wady
 - modele mniej intuicyjne od strukturalnych
 - duża liczba modeli pomocniczych zwiększa złożoność projektu
 - iteracyjny styl pracy → konieczność modyfikowania wcześniejszej opracowanego kodu podczas integrowania z wytworzonym w kolejnej iteracji

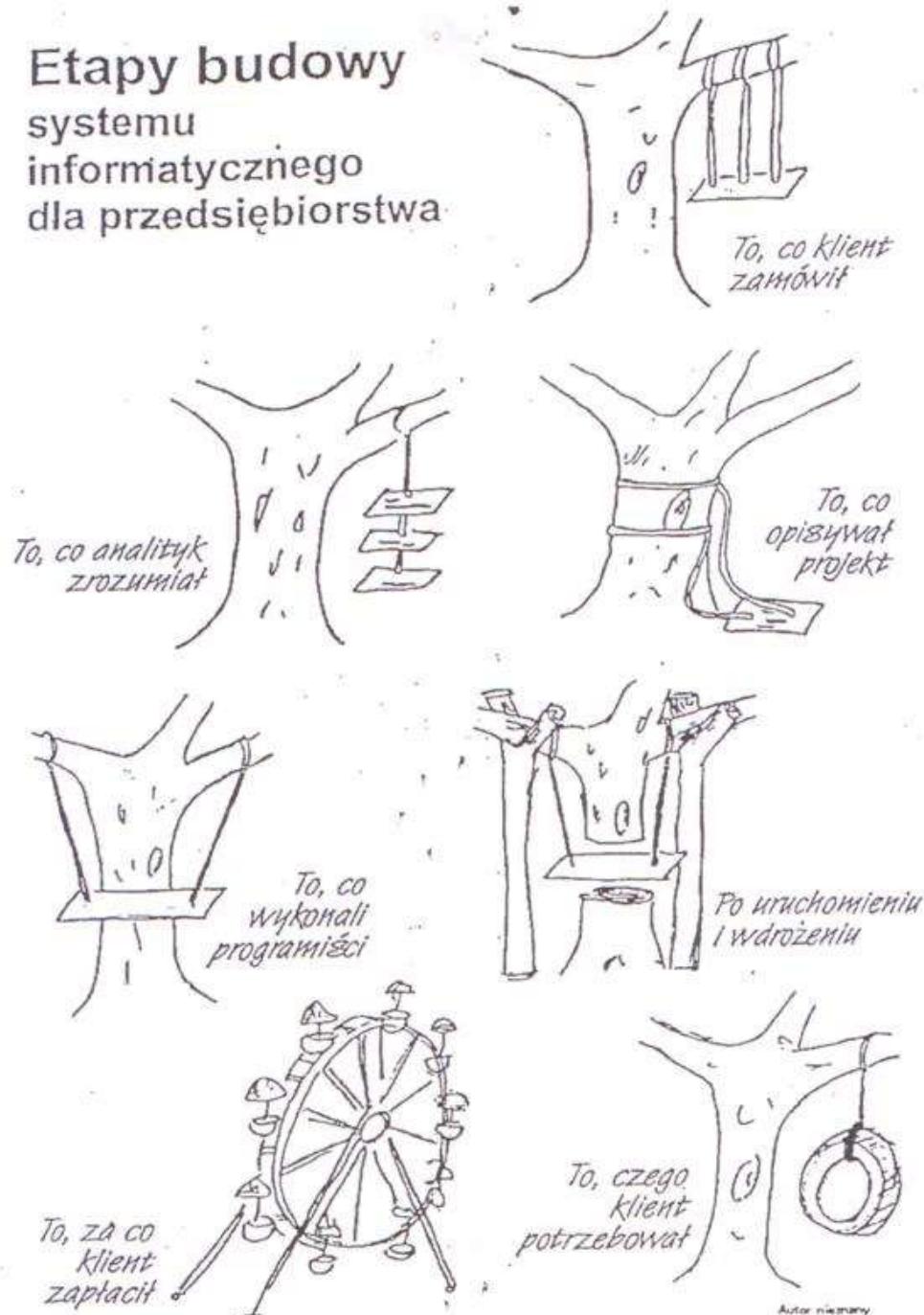
Weryfikacja i zatwierdzanie

Wytwarzanie oprogramowania

V&V

- **V&V – verification and validation**
- **Weryfikacja**
 - na każdym etapie ludzie mogą popełniać błędy
 - konieczne jest sprawdzanie poprawności względem poprzedniego etapu
 - czy projekt zapewnia wymagania ze specyfikacji?
 - czy implementacja zgodna z projektem?
- **Zatwierdzanie**
 - nawet poprawny program może nie zaspokoić potrzeb
 - potrzeby źle rozpoznane
 - wymagania nie zrealizowane w całości
 - po analizie wymagań
 - czy specyfikacja określa funkcje oczekiwane przez użytkownika?
 - po implementacji
 - czy gotowe oprogramowanie działa zgodnie z oczekiwaniami i potrzebami?

Etapy budowy systemu informacyjnego dla przedsiębiorstwa



Autor nieznany

Wytwarzanie oprogramowania

V&V

- konkretne działania zależą od charakteru zastosowania
 - tym wyższe wymagania, im większe możliwe straty
 - najwyższe wymagania w dziedzinach, w których błędne działanie może zagrozić życiu lub zdrowiu ludzi
 - sterowanie przemysłowe, transport, energetyka, medycyna
 - niedopuszczalne nawet pojedyncze błędy
 - najniższe wymagania – gdzie błędy nie powodują poważnych następstw
 - błąd edytora tekstu lub gry komputerowej
 - drobne usterki nie dyskwalifikują produktu

Wytwarzanie oprogramowania

V&V

- oprócz poprawności ważne również inne aspekty jakości:
 - łatwość użycia
 - wydajność
 - niezawodność
 - łatwość konserwacji

Wytwarzanie oprogramowania

V&V

- V&V mają trochę inny cel i nastawienie
- ale często używają podobnych metod oceny
- 3 główne rodzaje metod V&V
 - testowanie
 - przeglądy i inspekcje
 - dowodzenie poprawności

Wytwarzanie oprogramowania

V&V: testowanie

- wykonywanie programu z zadanym zestawem danych wejściowych, rejestrowanie i ocena wyników
 - na przykład: poprawności wykonania funkcji czy wydajności
- celem jest zawsze **wykrycie i usunięcie defektów**
- ograniczone jedynie do **produktów wykonywalnych**
 - programów lub prototypów
- może wykazać istnienie defektów
 - ale nie może wykazać braku defektów
- jest procesem spóźnionym
 - defekty mogły powstać na początku projektu
- pomimo wad jest najważniejszą i najbardziej wiarygodną metodą weryfikacji i zatwierdzania

Wytwarzanie oprogramowania

V&V: przeglądy i inspekcje

- ang. *reviews & inspections*
- forma oceny jakości prac i produktów – kontrola treści **dokumentów analitycznych i projektowych**
 - specyfikacja wymagań, projekt, program, plan testowania
- przygotowanie przeglądu obejmuje często opracowanie recenzji oceniających badany dokument lub stan prac
- najważniejszy element przeglądu → **spotkanie z innymi członkami zespołu**
 - również kierownictwo i klient
- **pozytywna ocena** → możliwe zatwierdzenie produktu
- **negatywna ocena** → ocena stanu zaawansowania i zbiór uwag
- główna zaleta → możliwa ocena przez ekspertów **wszystkich produktów i prac**

Wytwarzanie oprogramowania

V&V: dowodzenie poprawności

- ang. *correctness proving*
- formalne (matematyczne) wykazanie, że program lub inny produkt ma pożądane właściwości
- teoretycznie możliwe bo
 - każdy program jest formalnym zapisem przekształceń liczbowych
- ale bardzo **trudne**
 - przeprowadzenie dowodu trudniejsze niż napisanie programu
 - dowód dłuższy niż program
 - dowód może zawierać błędy
 - jak zapisać wymagania w sposób formalny i w takiej postaci zatwierdzić
 - eksperci dziedzinowi nie są w stanie tego zrobić

Software Engineering Body of Knowledge (SWEBok)

SWEBoK

- "*zasób wiedzy w zakresie inżynierii oprogramowania*"
- sponsor: IEEE Computer Society
- opracowanie: Professional Practices Committee
- wersja z 2004 roku
 - v3.0 z 2014 roku
- przewodnik musi się rozwijać i ewoluować
 - zmiany w technologii

SWEBoK

- Cel:
 - dostarczenie uzgodnionego i zatwierdzonego opisu zakresu dziedziny inżynierii oprogramowania oraz
 - udostępnienie zasobów tematycznych wspomagających tę dziedzinę
- podział inżynierii oprogramowania na 12 obszarów wiedzy
 - plus 3 dodatkowe rozdziały omawiające obszary wiedzy w pokrewnych dziedzinach
- nie zawiera treści informatyki teoretycznej (*computer science*)
 - nacisk na konstruowanie przydatnych produktów programowych

SWEBoK

- przewodnik **nie jest kompletny**
- obejmuje wiedzę, która jest konieczna ale **nie jest wystarczająca** dla inżyniera oprogramowania
 - dodatkowa wiedza: zarządzanie projektami, informatyka, inżynieria systemów

SWEBoK

- Czym jest inżynieria oprogramowania?
 1. zastosowanie systematycznego, zdyscyplinowanego i policzalnego podejścia do rozwoju, użycia i konserwacji oprogramowania;
tj. zastosowanie inżynierii do oprogramowania
 2. nauka o podejściach wymienionych wyżej

SWEBoK

- komponenty zawodu inżyniera
 - wstępna edukacja zawodowa w organizacji akredytowanej przez stowarzyszenie
 - dobrowolna certyfikacja lub obowiązkowe licencje
 - wyspecjalizowany rozwój umiejętności i stała edukacja zawodowa
 - wspieranie społeczne przez zawodowe stowarzyszenie
 - postępowanie zgodnie z kodeksem etycznym
- SWEBoK odnosi się do pierwszych trzech

SWEBoK

- 5 celów szczegółowych:
 1. promocja wspólnej wizji inżynierii oprogramowania na całym świecie
 2. objяснienie miejsca i zakresu inżynierii oprogramowania w odniesieniu do innych dziedzin, takich jak informatyka, zarządzanie projektami, inżynieria komputerów, matematyka
 3. opisanie przedmiotu dziedziny inżynierii oprogramowania
 4. umożliwienie tematycznego dostępu do SWEBoK
 5. dostarczenie podstawy rozwoju kariery i materiału certyfikacji indywidualnej i licencjonowania

SWEBoK – 12 głównych obszarów wiedzy

wymagania wobec oprogramowania	software requirements
projekt oprogramowania	software design
konstrukcja oprogramowania	software construction
testowanie oprogramowania	software testing
konserwacja oprogramowania	software maintenance
zarządzanie konfiguracją oprogramowania	software configuration management
zarządzanie inżynierią oprogramowania	software engineering management
proces inżynierii oprogramowania	software engineering process
narzędzia i metody inżynierii oprogramowania	software engineering tools and methods
jakość oprogramowania	software quality
praktyka zawodowa inżynierii oprogramowania	software engineering professional practice
ekonomia inżynierii oprogramowania	software engineering economics

SWEBoK

1. wymagania wobec oprogramowania

- podstawy wymagań wobec oprogramowania
- proces wymagań
- pozyskiwanie wymagań
- analiza wymagań
- specyfikacja wymagań
- zatwierdzenie wymagań
- rozważania praktyczne

2. projekt oprogramowania

- podstawy projektowania oprogramowania
- kluczowe zagadnienia w projektowaniu oprogramowania
- struktura i architektura oprogramowania
- analiza i ocena jakości projektu oprogramowania
- notacje projektu oprogramowania
- strategie i metody projektowania oprogramowania

3. konstrukcja oprogramowania

- podstawy konstrukcji oprogramowania
- zarządzanie konstrukcją
- rozważania praktyczne

4. testowanie oprogramowania

- podstawy testowania oprogramowania
- poziomy testowania
- techniki testowania
- miary testowania
- proces testowania

5. konserwacja oprogramowania

- podstawy konserwacji oprogramowania
- kluczowe zagadnienie konserwacji oprogramowania
- proces konserwacji
- techniki konserwacji

6. zarządzanie konfiguracją oprogramowania

- zarządzanie procesem konfiguracji
- identyfikacja konfiguracji oprogramowania
- kontrola konfiguracji oprogramowania
- raportowanie stanu konfiguracji oprogramowania
- audyt konfiguracji oprogramowania
- zarządzanie wydaniami oprogramowania i dostawami

7. zarządzanie inżynierią oprogramowania

- inicjacja i definicja zakresu
- planowanie projektu informatycznego (~programowego)
- przeprowadzenie projektu informatycznego
- przegląd i ocena
- zamknięcie
- pomiar inżynierii oprogramowania

8. proces inżynierii oprogramowania

- implementacja procesu i zmiany
- definicja procesu
- ocena procesu
- pomiar procesu i produktu

9. narzędzia i metody inżynierii oprogramowania

- narzędzia inżynierii oprogramowania
- metody inżynierii oprogramowania

10. jakość oprogramowania

- podstawy jakości oprogramowania
- procesy zarządzania jakością oprogramowania
- rozważania praktyczne

11. praktyka zawodowa inżynierii oprogramowania

- profesjonalizm
- dynamika i psychologia grup
- umiejętności komunikacyjne

12. ekonomia inżynierii oprogramowania

- podstawy ekonomii inżynierii oprogramowania
- ekonomia cyklu życia
- ryzyko i niepewność
- metody analizy ekonomicznej
- aspekty praktyczne

SWEBoK

pokrewne dziedziny – wersja z 2004 roku

- inżynieria komputerów
- informatyka
- zarządzanie
- matematyka
- zarządzanie projektami
- zarządzanie jakością
- ergonomika oprogramowania
- inżynieria systemów

SWEBoK

- pokrewne dziedziny – v3.0 – trzy większe rozdziały
 - podstawy przetwarzania komputerowego
 - podstawy matematyki
 - podstawy inżynierii

Problem

- Otrzymujesz wstępную информацию, że pewna szkoła podstawowa zamierza zlecić Twojej firmie opracowanie oprogramowania wspomagającego prowadzenie bieżącej działalności:
 - ewidencja uczniów, nauczycieli, obecności uczniów, nauczycieli, dzienniczek, plan zajęć, zajęcia dodatkowe, konkursy przedmiotowe
- W jaki sposób zamierzałbyś/zamierzałabyś realizować projekt?
 - proces: kaskadowy, iteracyjny, zwinny, inny?
 - metody strukturalne czy obiektowe?
 - ile osób byłoby zaangażowanych w projekt po stronie wykonawcy?
 - jak długo trwałby projekt do czasu ostatecznego przekazania do użytkowania?

Podsumowanie

podstawowe koncepcje
inżynierii oprogramowania

Pytania



Źródła

- Sacha K., Inżynieria oprogramowania, PWN 2010
- Guide to the Software Engineering Body of Knowledge, IEEE Computer Society, 2004
- Guide to the Software Engineering Body of Knowledge, IEEE Computer Society, 2014
<http://www.computer.org/portal/web/swebok/swebokv3>

Następny wykład

Metodyki wytwarzania oprogramowania

Inżynieria oprogramowania

Wykład 2: Metodyki wytwarzania oprogramowania

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Agenda

- RUP
- Metodyka zwinna
 - Extreme Programming
 - Scrum

Rational Unified Process (RUP)

RUP

- opracowany przez Rational Software
 - przejęta przez IBM
- jest procesem inżynierii oprogramowania
 - a raczej **szablonem procesu**
 - z szablonu można wybrać elementy odnoszące się do danej organizacji / zespołu / projektu
- dostarcza zdyscyplinowanego podejścia w przydzielaniu zadań i obowiązków w organizacji
- celem jest zapewnienie wytworzenia oprogramowania o wysokiej jakości
 - które spełnia **potrzeby** użytkowników końcowych
 - zgodnie z przewidywalnym **harmonogramem i budżetem**

RUP – założenia

- zwiększa **produktywność** zespołu
 - umożliwia członkom dostęp do bazy wiedzy
 - przewodniki, szablony, narzędzia we wszystkich kluczowych działaniach
 - wspólna baza wiedzy
 - używanie wspólnego języka, procesu i koncepcji tworzenia oprogramowania
- w trakcie działań tworzone są **modele**
 - zamiast papierowych dokumentów
 - modele – bogate semantycznie reprezentacje rozwijanego systemu

RUP

- wskazuje jak skutecznie wykorzystywać UML
- jest wspomagany narzędziami
 - automatyzują znaczne części procesu
 - tworzenie i aktualizacja różnych artefaktów
 - w szczególności modeli
 - wspieranie zarządzania zmianami
 - wspieranie zarządzania konfiguracją
- jest procesem konfigurowalnym
 - żaden proces nie jest idealny w każdej sytuacji
 - RUP w małych i dużych zespołach / projektach

RUP – bazuje na najlepszych praktykach

iteracyjne wytwarzanie oprogramowania

zarządzanie wymaganiami

architektura bazująca na komponentach

wizualne modelowanie oprogramowania

weryfikacja jakości oprogramowania

kontrolowanie zmian w oprogramowaniu

RUP – najlepsze praktyki

iteracyjne wytwarzanie oprogramowania

- ze względu na złożoność i zmiany nie da się prowadzić procesu sekwencyjnie (kaskadowo)
- potrzebne podejście iteracyjne
 - rosnący poziom zrozumienia problemu
- skupia się na obszarach najbardziej ryzykownych na każdym etapie projektu
 - znaczne ograniczenie ryzyka niepowodzenia
 - widoczny postęp prac
 - częste działające wydania
 - możliwość oceny przez użytkowników

RUP – najlepsze praktyki zarządzanie wymaganiami

- jak zarządzać wymaganiami
 - pozyskiwanie
 - organizowanie
 - dokumentowanie
- śledzenie i dokumentowanie kompromisów i decyzji
- pozyskiwanie i uzgadnianie wymagań biznesowych
- techniczna realizacja za pomocą
 - przypadków użycia
 - scenariuszy

RUP – najlepsze praktyki

zarządzanie wymaganiami - czynności

analiza problemu

- uzgodnienie problemu i miar jego istotności

zrozumienie
potrzeb
udziałowców

- konsultacje, zidentyfikowanie rzeczywistych potrzeb

definicja systemu

- projekt funkcjonalności
- identyfikacja przypadków użycia

zarządzanie
zakresem systemu

- modyfikacje zakresu prac
- wybór kolejności implementacji przypadków użycia

zawężanie definicji
systemu

- uszczegóławiania przypadków użycia → dokładna specyfikacja wymagań
- na podstawie specyfikacji powstaje później projekt i scenariusze testów

zarządzanie
zmianami wymagań

- zmienione lub nowe wymagania

RUP – najlepsze praktyki architektura bazująca na komponentach

- wczesne opracowanie solidnej uruchamialnej architektury
 - przed przydzieleniem zasobów do pełnego projektu
- RUP opisuje jak projektować elastyczne architektury
 - łatwiej do adaptacji
 - intuicyjnie zrozumiałej
 - promującej skuteczne ponowne wykorzystanie oprogramowania

RUP – najlepsze praktyki

architektura bazująca na komponentach

- RUP wspiera tworzenie systemu bazującego na komponentach (*component-based*)
- komponent
 - nietrywialny moduł lub podsystem, który realizuje określoną funkcję
- dzięki iteracyjnemu wytworzaniu
 - możliwość stopniowej identyfikacji komponentów
 - zakupione
 - zbudowane
 - ponownie użyte

RUP – najlepsze praktyki

wizualne modelowanie oprogramowania

- RUP opisuje sposób wizualnego modelowania oprogramowania
 - pozyskanie struktury i zachowania architektur i komponentów
- pozwala na ukrycie szczegółów i pisanie kodu za pomocą "graficznych klocków"
- wizualna abstrakcja
 - pomaga w komunikowaniu różnych aspektów oprogramowania
 - ocenę wzajemnego dopasowania elementów
 - zapewnienie zgodności klocków z kodem
 - utrzymanie spójności między projektem a implementacją
 - promuje jednoznaczna komunikację
- podstawą są modele UML

RUP – najlepsze praktyki

weryfikacja jakości oprogramowania

- główne czynniki obniżające akceptowalność
 - niska wydajność
 - niska niezawodność
- zapewnienie jakości na podstawie wymagań
 - niezawodności
 - funkcjonalności
 - wydajności aplikacji
 - wydajności systemu
- RUP asystuje w tych typach testów
 - planowanie, projektowanie, implementacja, wykonanie i ocena
- ocena jakości jest wbudowana w proces
 - wszystkie działania
 - wszyscy uczestnicy
 - użycie obiektywnych miar i kryteriów
 - **nie traktowana jako osobne działanie** (inna grupa)

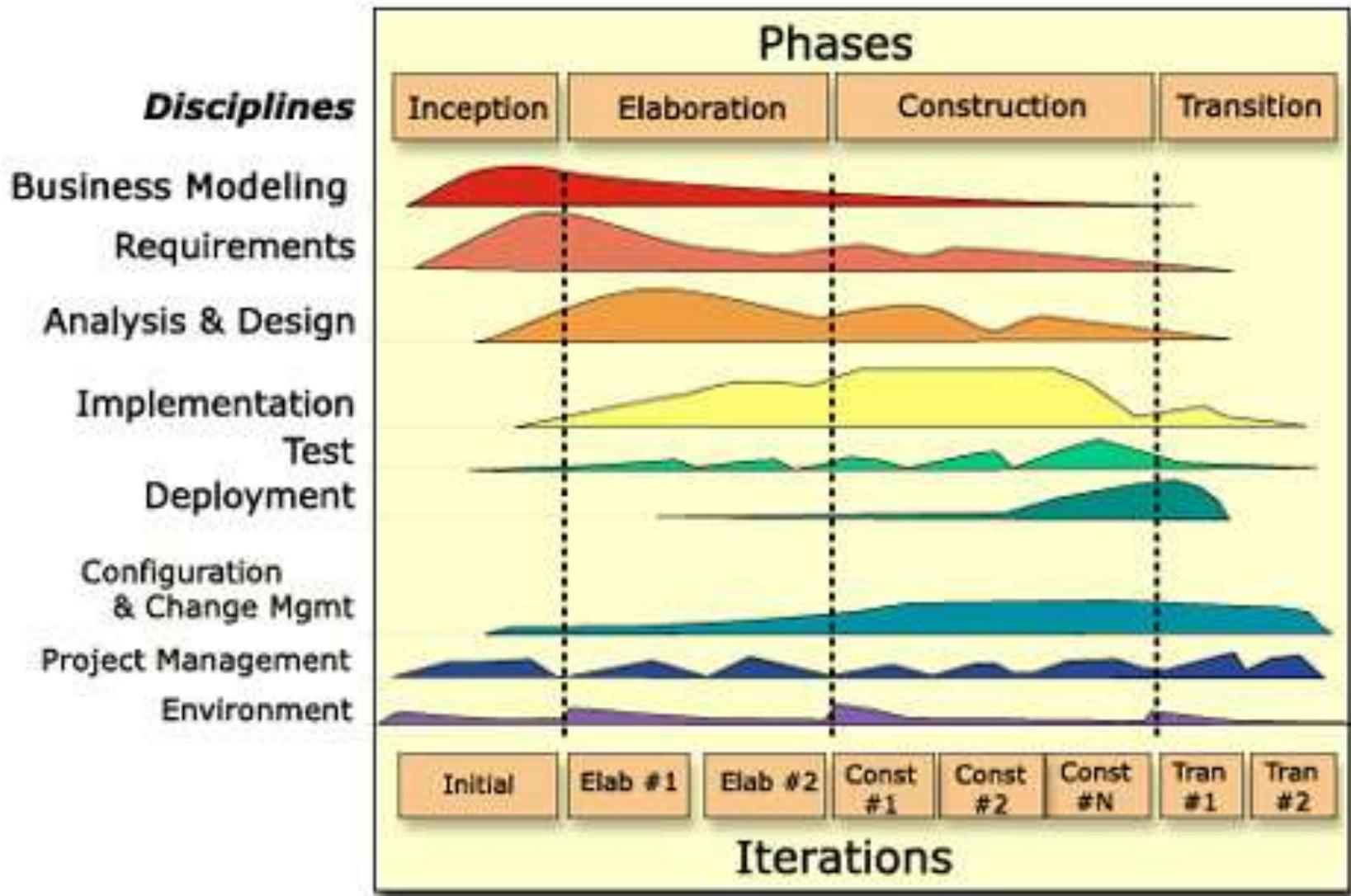
RUP – najlepsze praktyki

kontrolowanie zmian w oprogramowaniu

- zarządzanie zmianami to
 - zapewnienie, że każda zmiana jest akceptowalna
- możliwość śledzenia zmian
 - kluczowe w zmiennym środowisku
- RUP opisuje jak
 - kontrolować
 - śledzić
 - monitorować zmiany
- jak zapewnić przestrzeń roboczą (workspace) dla programisty
 - izolowanie zmian z innych przestrzeni roboczych
 - kontrolowanie zmian wszystkich artefaktów

Proces RUP

Proces RUP



4 fazy RUP

Nr	Faza	Nakłady	Czas
1	rozpoczęcie (inception)	5%	10%
2	opracowywanie (elaboration)	20%	30%
3	konstrukcja (construction)	65%	50%
4	przekazanie systemu (transition)	10%	10%

4 fazy RUP

1. rozpoczęcie (inception)

- sformułowanie problemu – "zagadnienie biznesowe"
 - kontekst
 - czynniki sukcesu
- dodatkowe elementy
 - wstępny model przypadków użycia (ukończony w 10%-20%)
 - plan projektu
 - wstępna analiza ryzyka
 - opis projektu
 - główne wymagania, ograniczenia, funkcjonalność
 - jeden lub więcej prototypów
- koniec fazy – Lifecycle Objective Milestone
 - przy braku osiągnięcia kryteriów
 - zakończenie projektu
 - powtórzenie fazy

4 fazy RUP

2. opracowywanie (elaboration)

- cele:
 - analiza dziedziny
 - budowa podstawowej architektury
- rezultaty
 - model przypadków użycia (ukończony w $\geq 80\%$)
 - dodatkowe wymagania
 - niefunkcjonalne
 - nie powiązane z przypadkami użycia
 - opis architektury oprogramowania
 - działający prototyp architektury
 - poprawione lista ryzyk i zagadnienie biznesowe
 - plan rozwoju całego projektu
 - opcjonalniestępny podręcznik użytkownika
- koniec fazy – Lifecycle Architecture Milestone
 - zaniechanie projektu
 - powtórzenie fazy
 - przejście do następnej fazy

4 fazy RUP

3. konstrukcja (construction)

- cele:
 - budowa komponentów i innych artefaktów
- w tej fazie większość prac programistycznych
- rezultaty – do przekazania dla użytkowników
 - produkt programowy
 - podręczniki użytkownika
 - opis bieżącego wydania
- koniec fazy - Initial Operational Capability Milestone

4 fazy RUP

4. przekazanie systemu (*transition*)

- cele:
 - przekazanie od zespołu do użytkowników końcowych
- czynności
 - szkolenie użytkowników końcowych i administratorów
 - testy akceptacyjne (beta)
 - zgodność z miarami określonymi w pierwszej fazie
 - równoległe działanie wraz ze starym systemem
 - konwersja działających baz danych
- koniec fazy - Product Release Milestone
 - zakończenie cyklu wytwarzania

4 fazy RUP - podsumowanie

Faza	Kluczowe pytanie	Nacisk
Rozpoczęcie	Czy powinniśmy budować system?	Zakres
Opracowanie	Czy możemy zbudować system?	Ryzyko
Konstrukcja	Czy zbudowaliśmy system?	Funkcjonalność
Przekazanie	Czy dostarczyliśmy system?	Dostarczenie

Klocki

Klocki

- oś pionowa z rysunku
- opisuję:
 - co ma zostać stworzone
 - jakie są wymagane umiejętności
 - jak powinien wyglądać proces krok po kroku

Klocki – rodzaje

- **rola (role)** – kto?
 - zachowanie i obowiązki
 - osoba lub grupa (zespół)
 - przykłady: projektant, autor przypadków użycia, architekt
- **zadanie (task)** – jak?
 - jednostka pracy przydzielona do wykonania określonej roli
 - przykłady: planowanie iteracji (dla kierownika projektu), identyfikacja przypadków użycia i aktorów (dla analityka)
- **produkt (product, artifact)** – co?
 - namalany produkt projektu
 - przykłady: model, element modelu, dokument, kod
- **dyscyplina (workflow)** – kiedy?
 - sekwencja działań wytwarzająca określony wynik

9 rodzajów dyscyplin (workflows)

- podstawowe
 - modelowanie biznesowe
 - wymagania
 - analiza i projektowanie
 - implementacja
 - testowanie
 - wdrożenie
- pomocnicze
 - zarządzanie konfiguracją i zmianami
 - zarządzanie projektem
 - środowisko

9 rodzajów dyscyplin (workflows)

modelowanie biznesowe

- brak komunikacji między
 - światem inżynierii oprogramowania
 - światem inżynierii biznesowej
- rezultat inżynierii biznesowej niewykorzystywany odpowiednio w inżynierii oprogramowania
 - i odwrotnie
- RUP:
 - wspólny język i procesy obu społeczności
 - jak tworzyć i śledzić bezpośrednie zależności między modelami biznesowymi i oprogramowania
- dokumentowanie procesów biznesowych przy pomocy biznesowych przypadków użycia
 - w wielu projektach nie jest realizowane

9 rodzajów dyscyplin (workflows) wymagania

- cel
 - opisanie tego, co system powinien robić
 - pozwala na osiągnięcie porozumienia przez deweloperów i klienta
- działania
 - pozyskiwanie, organizowanie i dokumentowanie wymaganej funkcjonalności i ograniczeń
 - śledzenie i dokumentowanie kompromisów i decyzji
- jak
 - model przypadków użycia
 - z opisami poszczególnych przypadków użycia
 - wymagania niefunkcjonalne w dodatkowej specyfikacji

9 rodzajów dyscyplin (workflows)

analiza i projektowanie

- cel
 - pokazanie jak system będzie realizowany w fazie implementacji
- rezultat
 - model projektowy
 - jest abstrakcją kodu źródłowego – struktura kodu
 - klasy, pakiety, interfejsy, które staną się komponentami w fazie implementacji
 - jak obiekty tych klas współpracują przy realizacji przypadków użycia
 - model analityczny (opcjonalnie)
- działania skupione na architekturze

9 rodzajów dyscyplin (workflows) implementacja

- cele
 - zdefiniowanie sposobu organizacji kodu w podsystemach
 - implementacja klas i obiektów w ramach komponentów
 - przetestowanie komponentów
 - integracja rezultatów poszczególnych programistów
 - w postaci działającego systemu
- RUP opisuje jak
 - ponownie używać istniejące komponenty
 - implementować nowe komponenty

9 rodzajów dyscyplin (workflows)

testowanie

- cele
 - zweryfikowanie interakcji między obiektami
 - zweryfikowanie poprawności integracji komponentów
 - zweryfikowanie poprawności implementacji wymagań
 - identyfikacja i naprawa defektów
 - przed wdrożeniem
- RUP jest iteracyjny → testowanie w ciągu całego projektu
- testowanie trzech wymiarów jakości
 - niezawodność
 - funkcjonalność
 - wydajność aplikacji i systemu

9 rodzajów dyscyplin (workflows) wdrożenie

- **cel**
 - pomyślne wytwarzanie wydań produktów
 - przekazanie oprogramowania użytkownikom
- **działania**
 - wytwarzanie zewnętrznych wydań oprogramowania
 - pakowanie oprogramowania
 - rozpowszechnianie oprogramowania
 - instalowanie oprogramowania
 - pomoc użytkownikom
- **dodatkowe działania**
 - planowanie i przeprowadzanie testów beta
 - migracja istniejącego oprogramowania i/lub danych
 - formalne zatwierdzanie

9 rodzajów dyscyplin (workflows)

zarządzanie konfiguracją i zmianami

- jak kontrolować różne wytwarzane artefakty
- kontrola
 - ogranicza koszty związane z dezorganizacją
 - zapewnia brak konfliktu między artefaktami w sytuacjach
 - jednoczesnej aktualizacji
 - ograniczonego powiadamiania
 - wielości wersji
- RUP dostarcza
 - wskazówek dotyczących zarządzania wariantami ewoluującego systemu
 - śledzenia wykorzystania wersji w poszczególnych buildach
 - tworzenie buildów poszczególnych programów lub wydań
- opisuje
 - zarządzanie wytwarzaniem równoległym
 - wytwarzaniem w wielu lokalizacjach
 - automatyzację procesu budowania oprogramowania
 - sposób śledzenia zmian w poszczególnych artefaktach
 - zarządzanie zmianami
 - sposób raportowania defektów, zarządzanie defektami w cyklu, wykorzystanie danych o defektach do śledzenia postępów i trendów

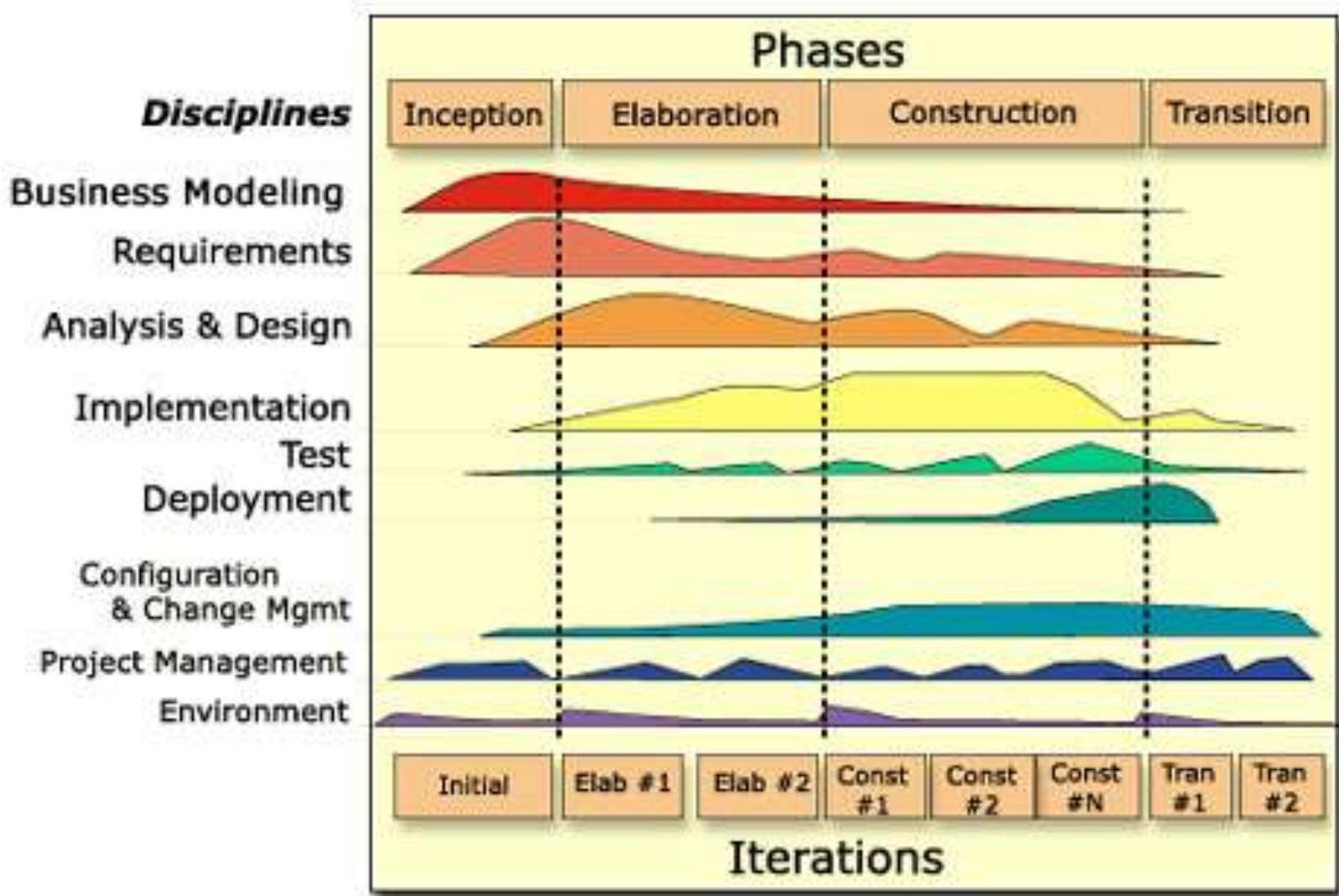
9 rodzajów dyscyplin (workflows) zarządzanie projektem

- jest kombinacją
 - sztuki balansowania rywalizującymi celami
 - zarządzania ryzykiem
 - pokonywania ograniczeń
- w celu dostarczenia produktu spełniającego wymagania klienta i użytkownika
- wielość niepowodzeń wskazuje na trudność tego zadania
- RUP dostarcza
 - szkielet zarządzania projektami informatycznymi
 - praktyczne wskazówki planowania, obsadzania personelem, realizacji i monitorowania projektów
 - szkielet zarządzania ryzykiem
- **nie stanowi recepty na sukces**
 - ale zwiększa szanse sukcesu

9 rodzajów dyscyplin (workflows) środowisko

- cel
 - dostarczenie środowiska wytwarzania oprogramowania
 - procesów i narzędzi
- działania
 - konfiguracja procesu w kontekście projektu
 - opracowanie wskazówek wspomagających projekt
- zawiera *development kit*
 - wskazówki, szablony i narzędzia

Proces RUP i dyscypliny



Metodyka zwinna

Sformalizowane procesy wytwarzcze

- do dużych projektów OK
- w małych zespołach i przy szybko zmieniających się wymaganiach – szereg wad
 - kosztowne opracowanie i utrzymanie dokumentacji
 - niska czytelność dokumentacji
 - trudna ocena przez użytkowników
 - budowa systemu niespełniającego oczekiwania

Metodyka zwinna (*agile*)

- jako odpowiedź na mankamenty sformalizowanych procesów
- nazywana lekką
- główne cechy
 - odejście od długofalowego planowania, zarządzania i dokumentowania
 - obserwowanie sytuacji i reagowanie na zmiany
 - szybkie tworzenie kodu
- zwykle nie obejmuje wielkich projektów

Metodyka zwinna

- jednostka czasu
 - kilka miesięcy
 - na zbudowanie i przekazanie **wydania (release)**
- rozbudowa systemu
 - w kolejnych wydaniach
 - jeśli takie są potrzeby użytkowników

Metodyka zwinna

- sposób użycia zasobów
 - planowany dla **iteracji**
 - trwających pojedyncze tygodnie
- wynikiem **iteracji** jest **działająca wersja**
 - przekazywana natychmiast użytkownikom
 - brak potrzeby oceny dokumentacji
 - oceniany od razu sam program
- krótki horyzont czasowy
 - brak potrzeby rygorystycznego planowania i zarządzania
 - samoorganizacja zespołu projektowego
- organizacja pracy nie jest związana z nowościami technicznymi
 - w razie potrzeby znane modele obiektowe
 - stosowane zwykle oszczędniej

Planowanie wydania

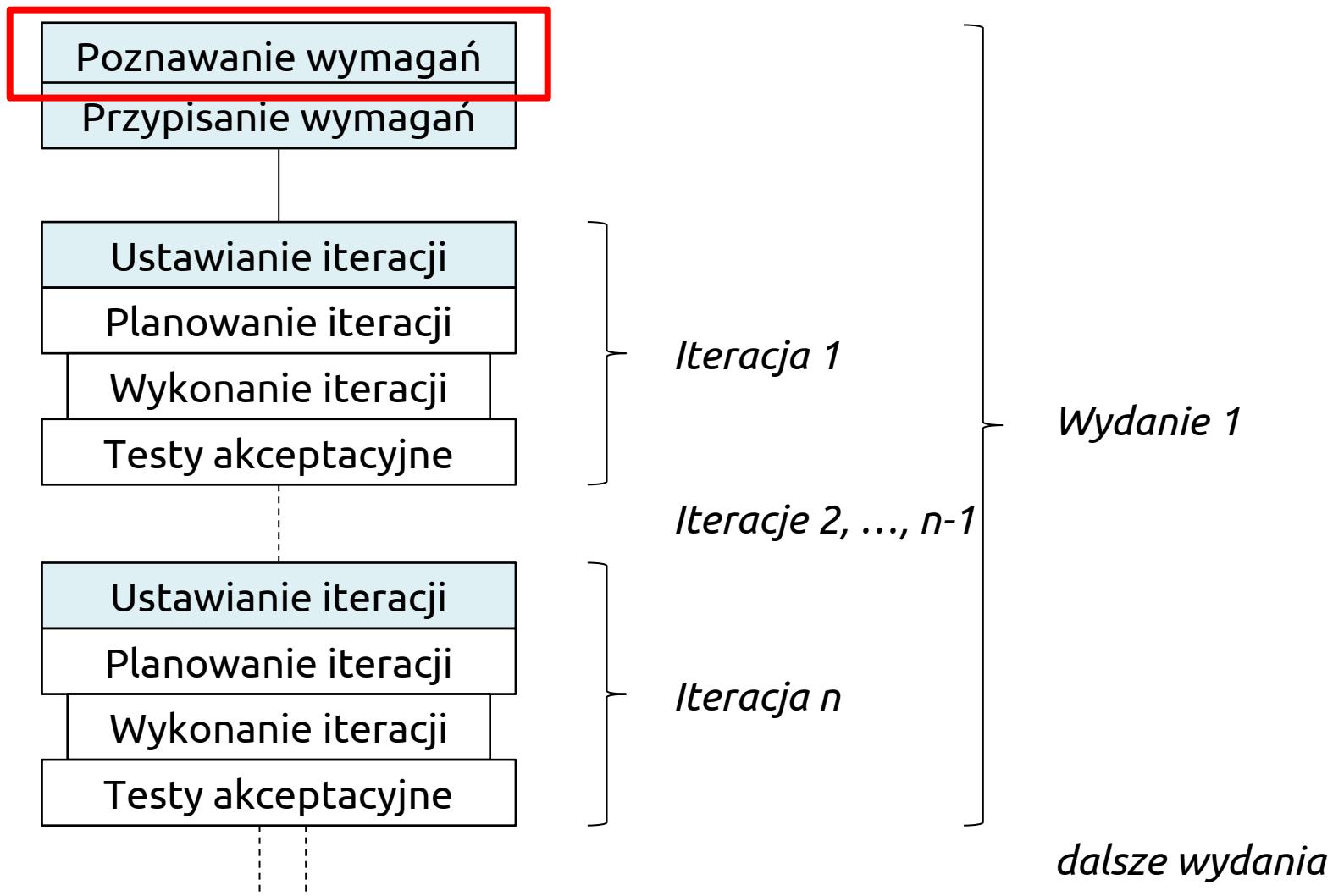
- gospodarzem projektu jest zespół
 - sam organizuje pracę i odpowiada za wynik
- członkowie zespołu
 - programiści
 - projektują i programują
 - przedstawiciel użytkowników (klient)
 - stawia i wyjaśnia wymagania
 - akceptuje wyniki
- **zespół decyduje o zakresie i terminie wydania**
- nikt z zewnątrz **nie narzuca decyzji**

Planowanie wydania

- wykonanie wydania
 - kilka miesięcy, nawet pół roku
- w tym czasie możliwe zmiany w projekcie
 - pomimo krótkiego horyzontu
 - układanie szczegółowego planu obarczone dużym ryzykiem
- plan wydania określa jedynie zakres i termin
 - bez struktury podziału pracy
 - bez harmonogramu
 - ani zakres, ani termin nie są niezmienne
 - na początku każdej iteracji klient może dokonać zmian wymagań
 - nowe ustalenia automatycznie zastępują poprzedni plan

Proces wg extreme programming

Planowanie i wykonanie wydania fazy wg *extreme programming*



Planowanie wydania

poznawanie wymagań

- do opisywania wymagań
 - historie użytkownika (*user stories*)
 - krótkie kilkuzdaniowe opisy potrzeb – sposobów działania systemu
 - przypominają opisy przypadków użycia
 - bez ułożonych scenariuszy działania
 - nie tworzą sformalizowanego dokumentu
 - często na luźnych kartkach
 - łatwe dodawanie, usuwanie, wymienianie
 - ile historii? – aby opisać wszystkie aspekty działania systemu

Planowanie wydania poznawanie wymagań

As a Game Player,

I want my Rocket to move back
and forth when I press left and
right arrows

so that I can avoid asteroids

<http://it-consulting.pl/autoinstalator/wordpress/2013/09/11/user-story-czyli-nic-nie-poprawic-a-moze-bardziej-skomplikowac/user-story-asteroids/>

AS A NEW CUSTOMER

I WANT TO REGISTER ON
THE SITE

SO I CAN BUY A PRODUCT

<http://www.cuanmulligan.com/2010/08/31/how-to-write-a-user-story/>

As a librarian, I
want to be able
to search for books
by publication year.

<http://code.google.com/p/econference-planning-poker-plugin/wiki/PlanningPoker>

As who, I want
what so that why.

<http://www.christiaanverwijs.nl/post/2012/01/03/Writing-application-requirements-as-User-Stories-an-elegant-simple-user-and-value-focussed-approach.aspx>

Planowanie wydania poznawanie wymagań



http://www.goshen.edu/physix/385/pub/user_stories.php

<http://northshoreagilegroup.pbworks.com/w/page/13234694/North%20Shore%20Agile%20Group%20Backlog>

Planowanie wydania

poznawanie wymagań

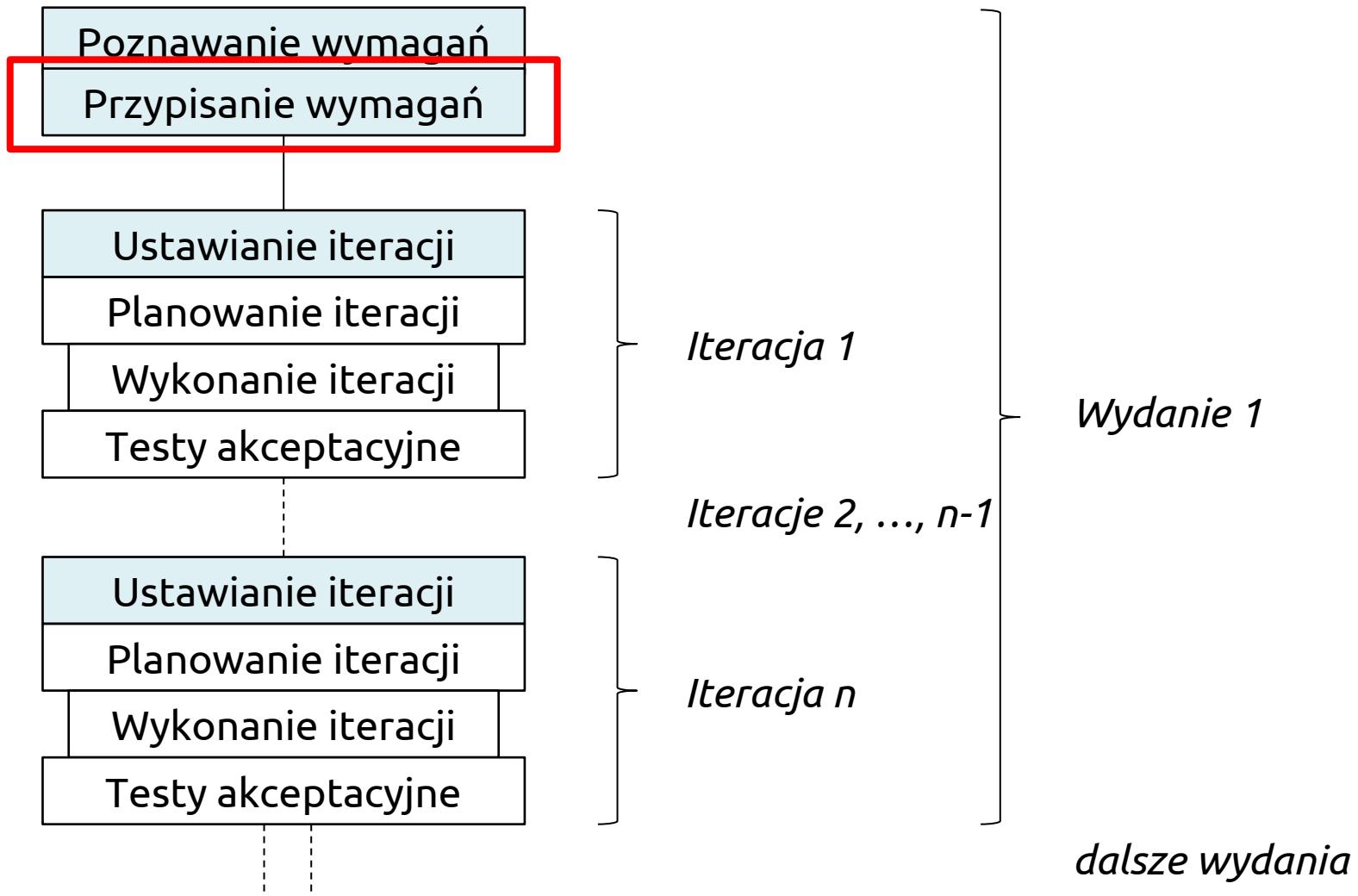
- następnie zespół ocenia **pracochłonność** implementacji każdej historii
 - ile tygodni pracy programisty
 - zakładając poświęcenie całego czasu
- dobrze opisana historia
 - do wykonania 1-3 tygodni
- historie z dłuższym czasem
 - zbyt mało szczegółowy podział funkcji systemu
 - klient powinien podzielić
- historie z krótszym czasem
 - zbyt proste
 - klient powinien połączyć kilka
- oszacowanie czasu musi być wiarygodne
 - to ten sam zespół będzie później implementował

Planowanie wydania

poznawanie wymagań

- możliwa szybka budowa **prototypowych** rozwiązań
 - możliwe eksperymenty z architekturą i technologią
 - nabycie doświadczeń ułatwiających późniejsze szacowanie pracochłonności
- powstaje **szkic architektury** na poziomie **metafory** (*metaphor*)
 - obrazowy opis porównujący system do ogólnie znanej konstrukcji
 - nie jest to dokładny projekt
 - tylko wizja pomagająca programistom utrzymanie kierunku
 - podczas ewolucyjnego tworzenia architektury w kolejnych iteracjach

Planowanie i wykonanie wydania fazy wg *extreme programming*



Planowanie wydania przypisanie wymagań

- kilkadziesiąt historii
 - z nich można wybrać te do realizacji w wydaniu
- klient sortuje historie względem **ważności**
- zespół może posortować według ryzyka
- **zakres proponuje klient**
 - które historie do realizacji
 - zwykle te najważniejsze
- zespół wpływa na wybór
 - implementacja których historii wiąże się z największym ryzykiem

Planowanie wydania przypisanie wymagań

- **pracochłonność wydania**
 - suma czasów wykonania poszczególnych historii
- **pracochłonność nie równoznaczna z czasem kalendarzowym**
 - deweloper poświęca czas na konsultacje z klientem, pomoc innym deweloperom
- **sztywność projektu (*velocity*)**
 - łączna pracochłonność historii w poprzedniej iteracji
- **wszystkie iteracje trwają tyle samo czasu**
 - dlatego szybkość \approx wydajność zespołu na iterację
- **ustalenie szybkości**
 - w pierwszej iteracji \rightarrow arbitralnie na podstawie poprzednich projektów
 - w kolejnych iteracjach \rightarrow doświadczalnie z bieżącego wydania

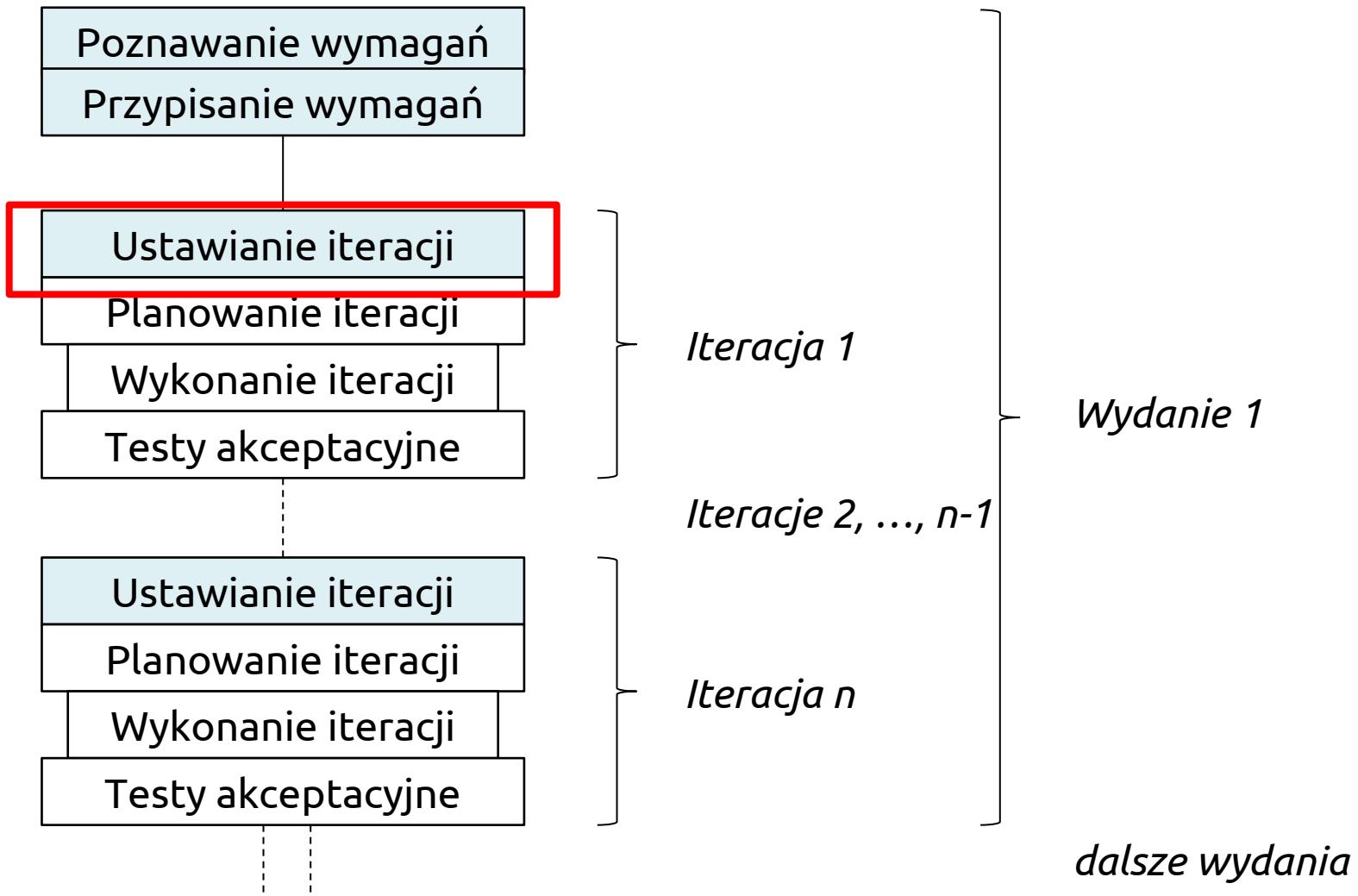
Planowanie wydania przypisanie wymagań

- znając długość iteracji i szybkość projektu
 - klient ustala **zakres** i **termin**
- **planowanie czasu**
 - ustalenie daty zakończenia
 - obliczenie liczby iteracji możliwych do wykonania
 - pracochłonność historii możliwych do wykonania
 - liczba iteracji x szybkość projektu
 - w razie potrzeby można dodać lub usunąć historie
- **planowanie zakresu**
 - zbiór historii do wykonania / szybkość projektu
 - w wyniku liczba niezbędnych iteracji
 - wyznacza czas trwania i datę zakończenia

Planowanie wydania przypisanie wymagań

- zbiór wybranych historii użytkownika
 - pełni rolę **specyfikacji wymagań**
 - określa rzeczywiste potrzeby
 - **nie narzuca** technicznej realizacji
 - technologii, formatu bazy danych, opisu algorytmów, ...
- przypisanie wymagań nie jest proste
 - to nie tylko wybór historii i obliczenie pracochłonności
 - negocjacje – zgoda udziałowców
 - zakres, koszt i czas – "wybierz dwie z nich"
 - niemożność dowolnego manipulowania
 - wielkość sprawnie działającego zespołu

Planowanie i wykonanie wydania fazy wg *extreme programming*



Planowanie wydania

ustawianie iteracji

- na początku każdej iteracji
 - klient wybiera historie do realizacji w danej iteracji
 - sumaryczna pracochłonność nie może przekroczyć ustalonej szybkości
 - w pierwszej iteracji historie muszą składać się na spójne działanie
- możliwa modyfikacja planu działania
 - zmiana potrzeb
 - zmiana treści historii
 - nowe historie
 - usuwane z planu historie o takiej samej pracochłonności
 - realna szybkość niższa od zakładanej
 - zespół prosi klienta o zredukowanie planu

Iteracja projektu

- jest podstawową jednostką
 - planowania
 - wykonania
 - kontroli postępów
- iteracje wykonywane **sekwen cyjne**
- czas iteracji
 - XP: 1-3 tygodnie
 - Scrum: 1-4 tygodnie (*Sprint*)
- wynik iteracji
 - kolejna działające wersja oprogramowania

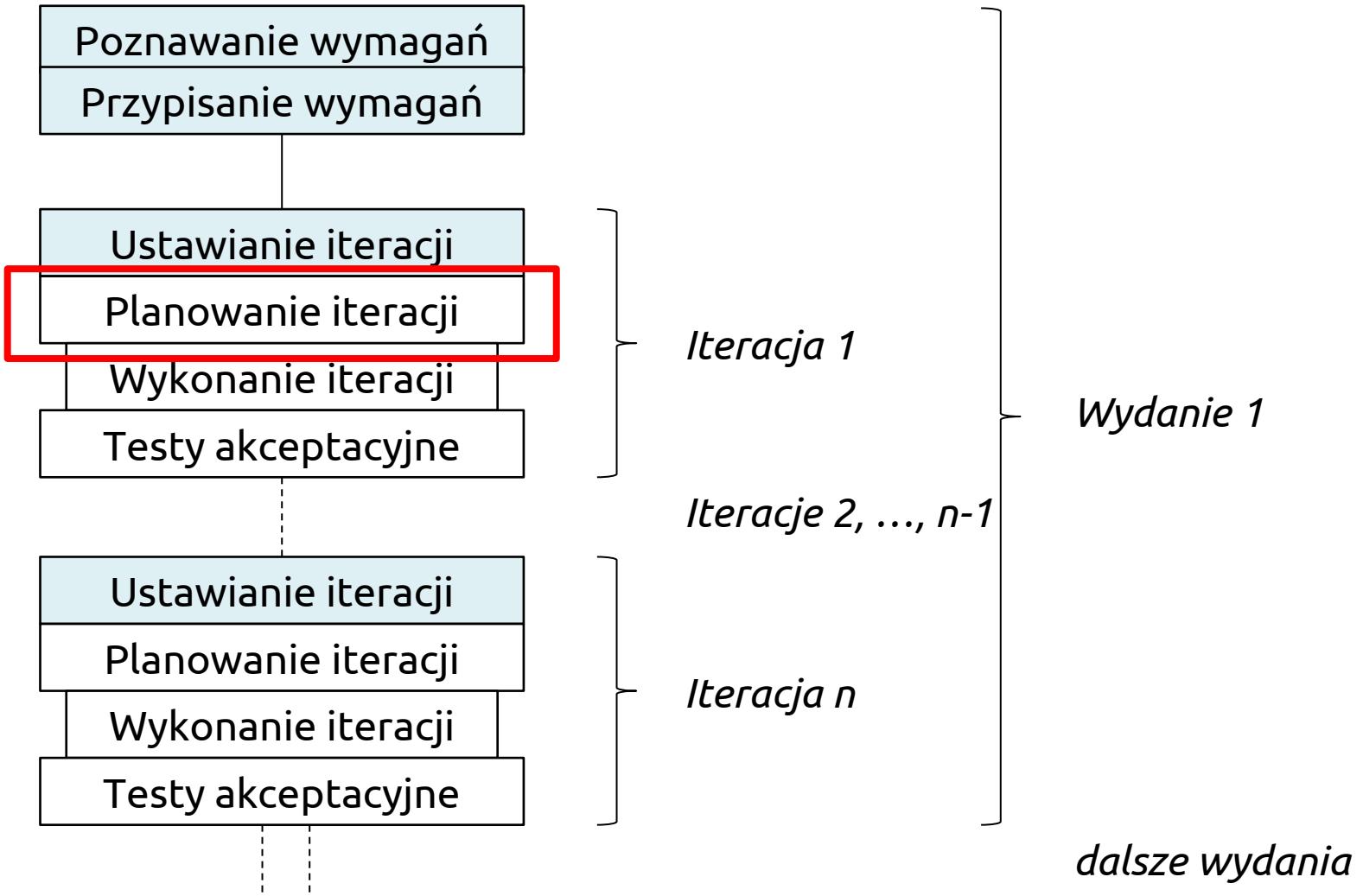
Iteracja projektu

- **cel iteracji**
 - implementacja historii wybranych do wykonania w fazie ustawiania iteracji
- co z opóźnieniami?
 - czas iteracji się nie zmienia
 - nieukończone historie przechodzą do następnej iteracji
 - zmniejsza się przyrost kodu → **spada szybkość projektu**
- jeśli jest trochę czasu?
 - klient przypisuje dodatkowe historie
 - zwiększa się przyrost kodu → **rośnie szybkość projektu**

Iteracja projektu

- stały rytm iteracji
- dlatego jedyny miernik postępu to **przyrost kodu**
 - bieżąca szybkość projektu
- prostsze i pewniejsze od oceniania dokumentów
 - nie wiadomo kiedy powstanie kod

Planowanie i wykonanie wydania fazy wg *extreme programming*



Iteracja projektu

planowanie iteracji

- 2 fazy
 1. poznawanie zadań
 2. przypisanie zadań

Iteracja projektu

planowanie iteracji – poznawanie zadań

- oprogramowanie **nie jest sumą modułów** (historii)
- program składa się z obiektów
 - jedna historia – wiele obiektów
 - jeden obiekt – wiele historii
- kluczowa jest zamiana historii użytkownika na zadania programistyczne
 - mogą być potrzebne dodatkowe zadania nie powiązane bezpośrednio z żadną historią
 - najczęściej opracowanie wspomagające proces tworzenia oprogramowania

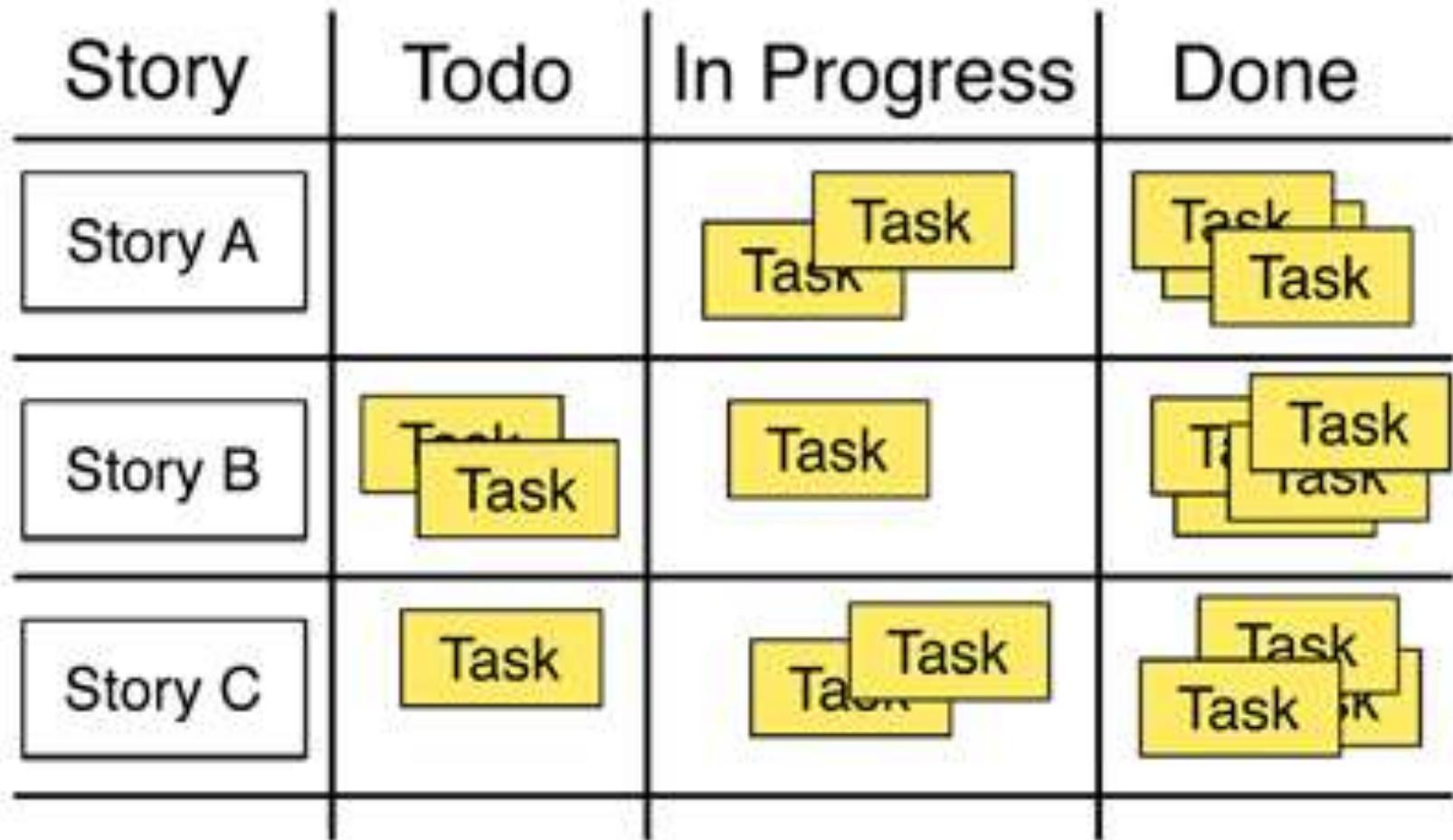
Iteracja projektu

planowanie iteracji – poznawanie zadań

- zadania mają charakter techniczny
 - konkretne problemy dla programistów
- realizacja zadania
 - prostsza niż implementacja historii
 - dobrze określone → w ciągu kilku dni
 - zbyt ogólne – zbyt szczegółowe
- w tej fazie rozmiar zadania szacuje się bardzo zgrubnie
 - później bardziej szczegółowa ocena pracochłonności

Iteracja projektu

planowanie iteracji – poznawanie zadań



Iteracja projektu

planowanie iteracji – przypisanie zadań

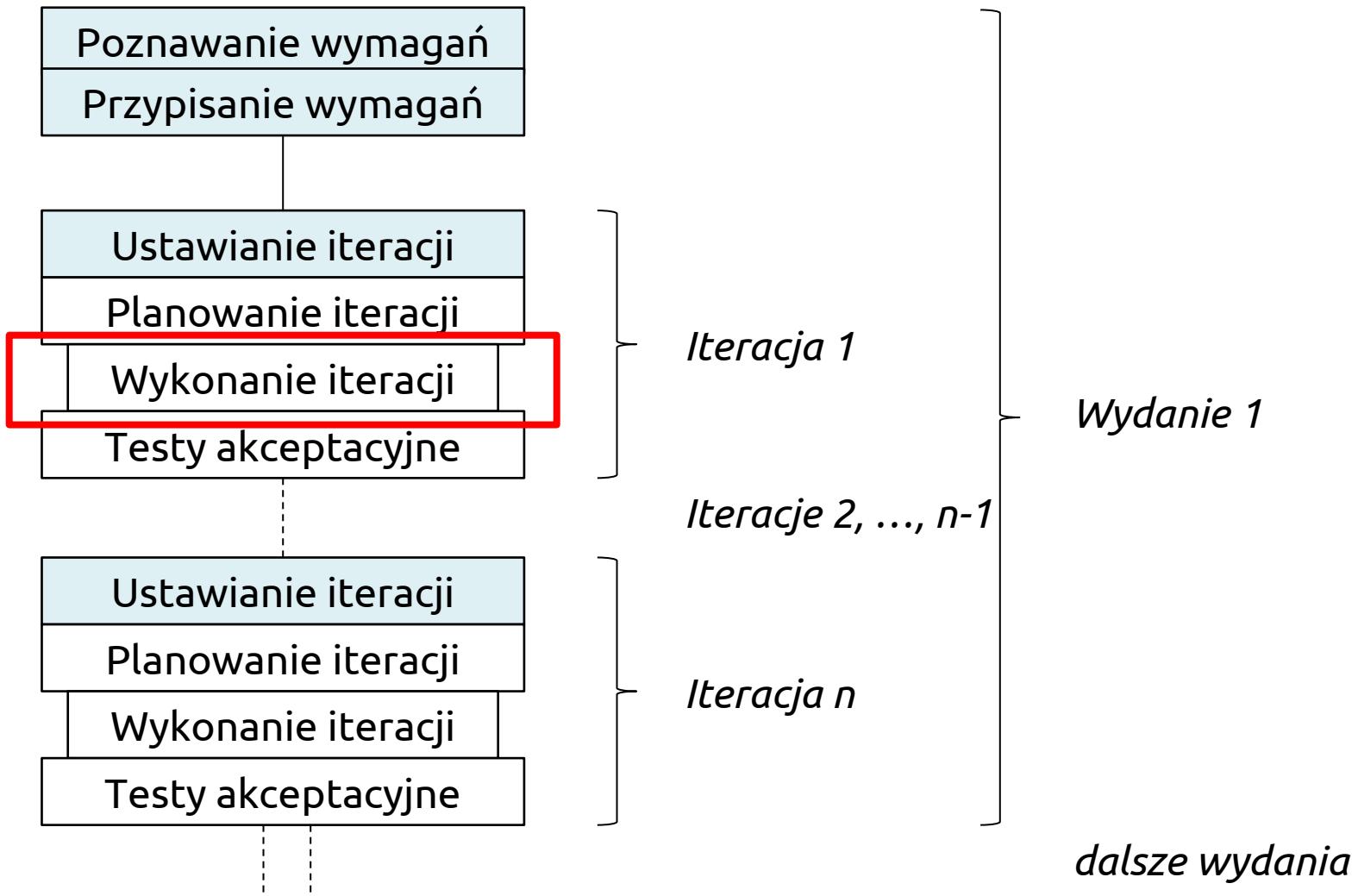
- pula zadań do wykonania
 - należące do nowych historii
 - niewykonane w poprzedniej iteracji
- **programiści sami wybierają zadania**
 - po dokonaniu wyboru **sami określają pracochłonność**
 - w dniach pracy
 - akceptowalne: 1-3 dni
 - jeśli inna → należy zmienić definicję zadania
 - pracochłonność zadań nie może przekroczyć indywidualnej szybkości
 - na podstawie poprzedniej iteracji

Iteracja projektu

planowanie iteracji – przypisanie zadań

- **istotne szacowanie pracochłonności przez osobę realizującą**
 - ludzie nie są zamienni
 - ile czasu konkretna osoba
- pracochłonność zadań (programiści) ≠ pracochłonności historii użytkownika
 - miarodajna: pracochłonność zadań
 - dotyczy mniejszych i lepiej określonych jednostek pracy

Planowanie i wykonanie wydania fazy wg *extreme programming*



Iteracja projektu

wykonanie iteracji

- po rozdzieleniu zadań programiści przystępują do pracy
- **bezwzględna reguła:**
 - **najpierw testy jednostkowe**
 - **potem program realizujący zadanie**
- program jest
 - testowany za pomocą testów jednostkowych
 - integrowany z całością oprogramowania
- testy
 - dołączane do zestawu testów
 - zintegrowana całość – ponownie testowana
 - czy nowe jednostki nie zaburzyły istniejących części

Iteracja projektu

wykonanie iteracji - TDD

- programowanie sterowane testami
 - test driven development (TDD)
- cel programisty
 - napisanie takiego programu, który przejdzie testy
- testy pełnią rolę dokumentacji zachowania

Iteracja projektu

wykonanie iteracji - TDD

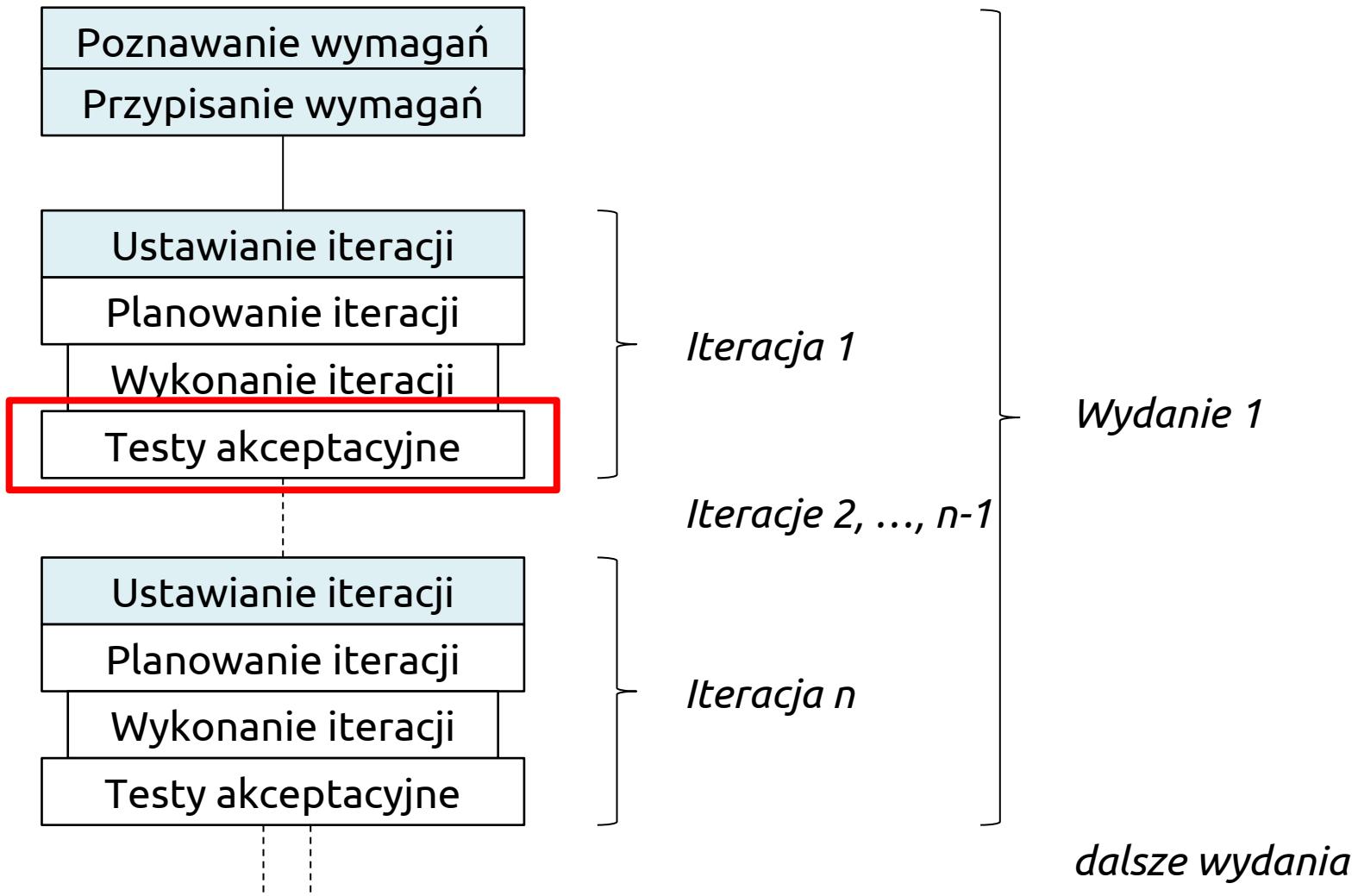
- wysoka ranga testów jednostkowych
 - testy nie mogą być dopasowane do programów
 - programów jeszcze nie ma
 - testy dopasowywane do opisów zadań
- opracowanie testów wymaga wcześniejszego określenia klas programu
 - potrzebna struktura programu
- najpierw ustalenie lokalnej architektury
 - co najmniej klasy i podstawowe metody
 - wykorzystanie modeli i wzorców projektowych dla zadania

Iteracja projektu

wykonanie iteracji – TDD

- opracowanie zadań
 - równolegle w kilkuosobowym zespole
- każda osoba
 - swoje programy, testy, integracja
- problem – całość może być tylko jedna
 - integracje szeregowo jedna po drugiej
- rozwiązanie
 - pojedyncze stanowisko do integracji
 - kolejka programistów do integracji
 - testy niepomyślne
 - wycofanie zmian i przywrócenie poprawnego stanu
 - integracje często
 - kilka razy dziennie
- potrzebna automatyzacja testów i integracji
 - na przykład JUnit

Planowanie i wykonanie wydania fazy wg extreme programming



Iteracja projektu

wykonanie iteracji – testy akceptacyjne

- istnieją 2 różne poziomy opisu pracy
 - biznesowy – historie użytkownika
 - techniczny – zadania programistyczne
- potrzebne 2 różne poziomy testowania
 - testy jednostkowe – dla zadań
 - opracowują i wykonują programiści
 - testy akceptacyjne (*functional tests*) – dla historii użytkownika
 - odpowiada klient

Iteracja projektu

wykonanie iteracji – testy akceptacyjne

- przygotowanie testów akceptacyjnych
 - rozpoczyna się po ustaleniu iteracji
 - wraz z rozpoczęciem jej wykonania
- klient
 - opracowuje przypadki testowe sprawdzające poprawność realizacji wszystkich historii
- przypadek testowy zawiera
 - dane wejściowe
 - oczekiwane poprawne wyniki
- pomoc dla klienta
 - tester
 - zespół – dodatkowe programy testowe

Iteracja projektu

wykonanie iteracji – testy akceptacyjne

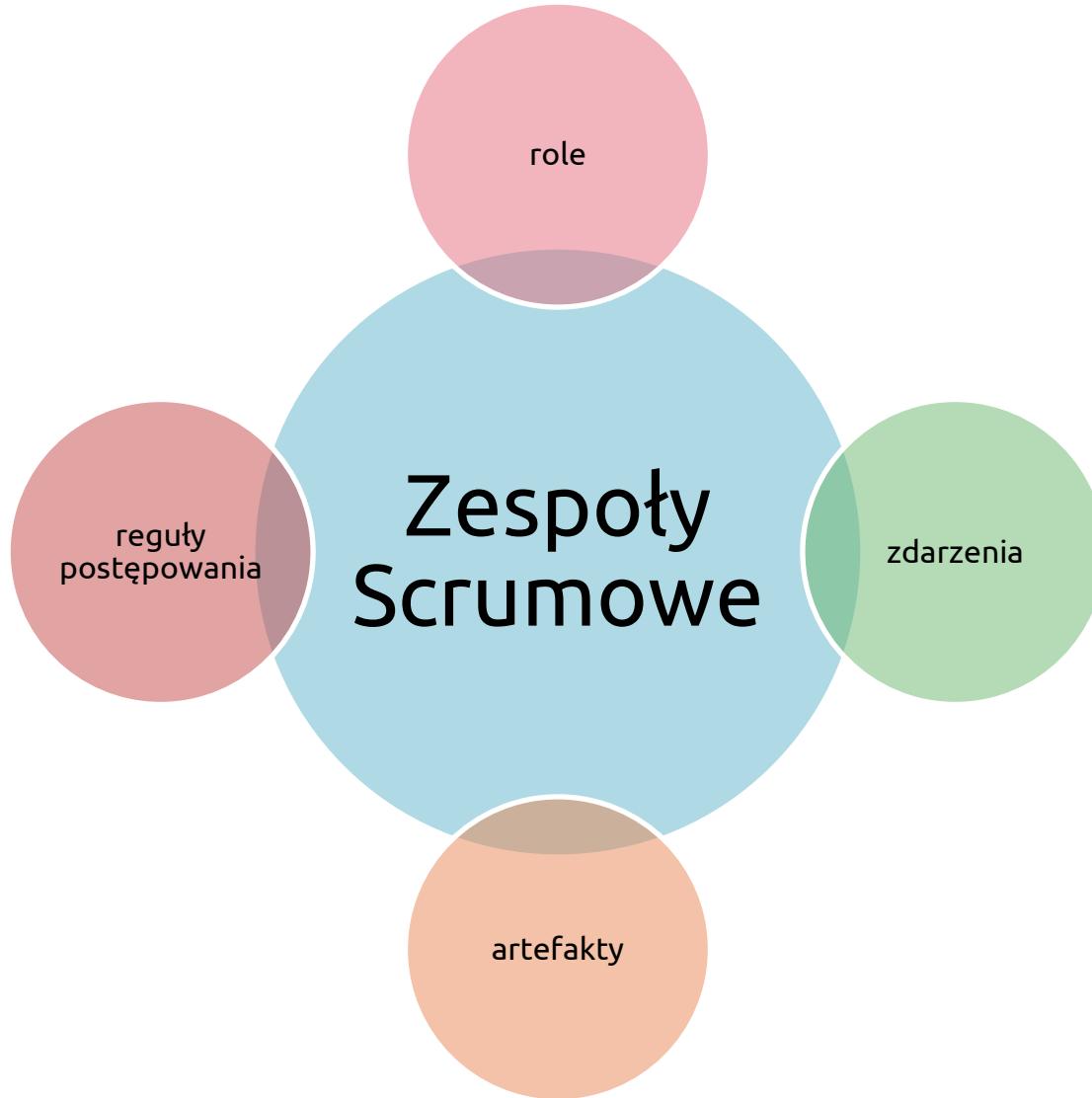
- kiedy?
 - programiści mogą wykonać TA od razu po zakończeniu integracji
 - o ile TA są gotowe
- historia użytkownika poprawnie zrealizowana
 - jeśli przechodzi wszystkie testy
- wielokrotne powtarzanie testów
 - narzędzie automatyzujące
 - *record and playback*
- ukończenie wydania
 - przeszły wszystkie testy
 - zatwierdzenie wydania przez użytkownika

Scrum

Czym jest Scrum?

- **metoda**, przy użyciu której ludzie mogą z powodzeniem **rozwiązywać złożone problemy adaptacyjne**, by w sposób produktywny i kreatywny **wytwarzać produkty o najwyższej możliwej wartości**
- Scrum jest
 - Lekki
 - Łatwy do zrozumienia
 - Bardzo trudny do opanowania

Struktura Scruma



Zespół Scrumowy

- zespoły
 - samoorganizujące
 - samodzielnie decydują w jaki sposób wykonywać pracę
 - nie są kierowane przez osoby spoza zespołu
 - posiadają wszelkie kompetencje do ukończenia pracy
 - dostarczają produkty iteracyjnie i przyrostowo
- skład Zespołu Scrumowego
 - Właściciel Produktu (*Product Owner*)
 - Zespół Deweloperski (*Development Team*)
 - Mistrz Scruma (*Scrum Master*)

Właściciel Produktu

- **odpowiedzialny za maksymalizację wartości produktu i pracy Zespołu Deweloperskiego**
 - osoba, **nie komitet!**
 - zadania **może zlecać** Zespółowi Deweloperskiemu, ale wciąż jest odpowiedzialny
- **sposób realizacji tej strategii może się różnić** między organizacjami, Zespołami Scrumowymi, czy osobami pełniącymi rolę Właściciela Produktu

Właściciel Produktu

- **jest jedyną osobą zarządzającą Rejestrem Produktu (Product Backlog):**
 - jasne artykułowanie elementów Rejestru Produktu
 - określanie kolejności elementów Rejestru Produktu w sposób zapewniający osiąganie założonych celów i misji
 - zapewnianie wartości pracy wykonywanej przez Zespół Deweloperski
 - zapewnianie, że Rejestr Produktu jest dostępny, przejrzysty oraz jasny dla wszystkich, a także, że dobrze opisuje to, czym Zespół Scrumowy będzie się zajmował w dalszej kolejności
 - zapewnianie, że Zespół Deweloperski rozumie elementy Rejestru Produktu w wymaganym stopniu

Zespół Deweloperski

- **profesjonalisi, których zadaniem jest dostarczanie, na koniec każdego Sprintu, gotowego do wydania Przyrostu (*Increment*) produktu**
 - tylko oni wytwarzają Przyrost

Zespół Deweloperski – cechy

- samoorganizujące się
 - nikt (nawet Scrum Master) nie może mówić Zespołowi Deweloperskiemu, jak należy przekształcać elementy Rejestru Produktu w Przyrosty gotowej do wydania funkcjonalności
- wielofunkcyjne
 - w swoim składzie posiadają wszystkie umiejętności niezbędne dotworzenia Przyrostu
- **brak tytułów innych niż „Deweloper” dla członków Zespołu Deweloperskiego**
 - bez względu na charakter pracy wykonywanej przez daną osobę i nie ma od niej wyjątków
- **odpowiedzialność za wykonywaną pracę ponosi cały Zespół Deweloperski**
 - mimo, iż pojedynczy członkowie Zespołu Deweloperskiego mogą posiadać wyspecjalizowane umiejętności oraz mogą skupiać się na konkretnych dziedzinach
- **nie składa się z podzespołów przeznaczonych do wykonywania konkretnych rodzajów zadań, jak na przykład testowanie czy analiza biznesowa**

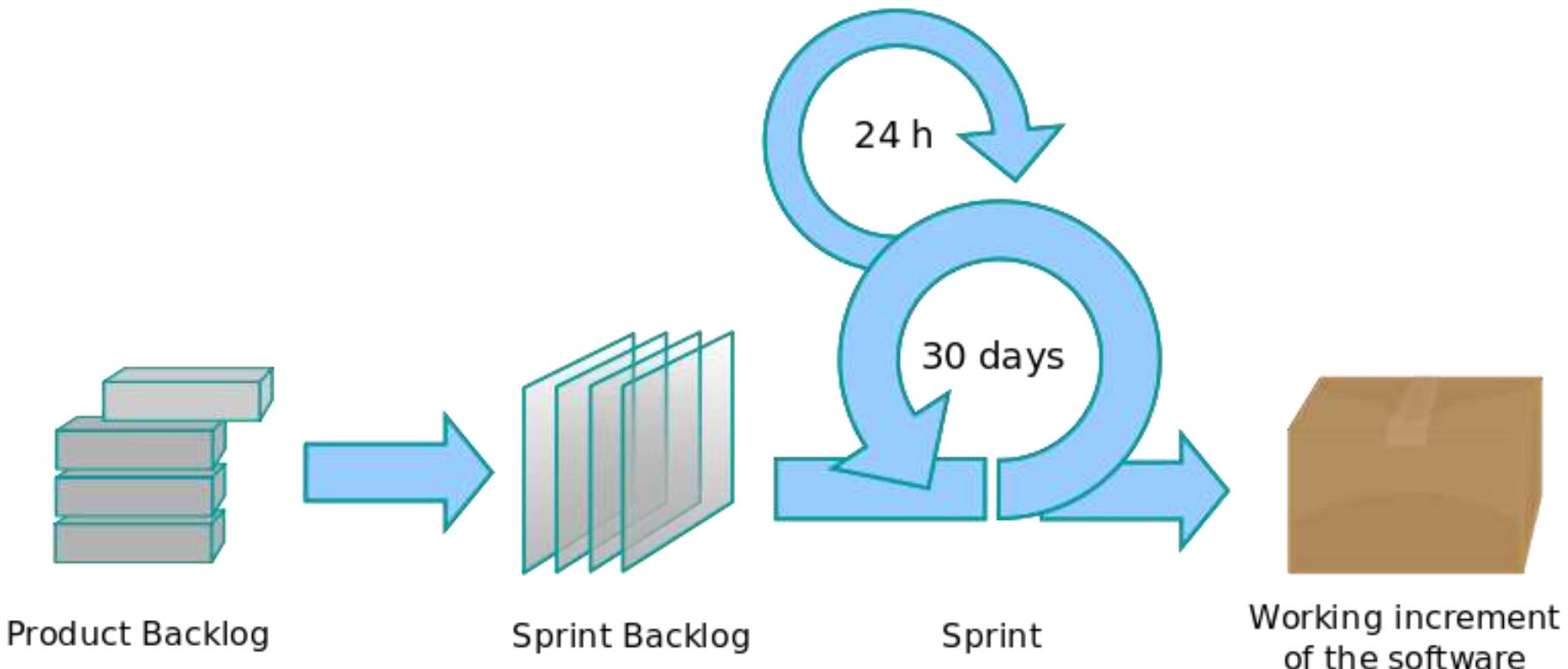
Zespół Deweloperski

- **Jaka wielkość zespołu?**
 - wystarczająco mała, by Zespół pozostał zwinnym
 - wystarczająco duża, by Zespół mógł wykonać znaczącą pracę
- Czyli od 3 do 9, bo:
 - mniej niż 3 członków to mniejszy stopień interakcji i mniejszy niż oczekiwany wzrost produktywności.
 - mniejsze Zespoły Deweloperskie mogą napotykać braki kompetencji uniemożliwiające im dostarczanie potencjalnie zbywalnego Przyrostu co Sprint
 - więcej niż 9 członków wymaga zbyt dużych nakładów na koordynację.
 - duże Zespoły Deweloperskie wprowadzają zbyt dużą złożoność, aby możliwe było zarządzanie nimi przy wykorzystaniu procesu empirycznego.
- Osoby Właściciela Produktu i Scrum Mastera **nie są wliczane** w te wartości
 - chyba że wykonują one jednocześnie pracę wynikającą z Rejestru Sprintu (*Sprint Backlog*)

Mistrz Scruma

- **odpowiedzialny za to, by Scrum był rozumiany i stosowany**
 - poprzez upewnianie się, że Zespół Scrumowy stosuje się do założeń teorii Scruma, jego praktyk i reguł postępowania
- można określić mianem **przywódcy służebnego** w stosunku do Zespołu Scrumowego
- pomaga osobom spoza Zespołu Scrumowego zrozumieć, które z ich interakcji z Zespołem Scrumowym są pomocne, a które nie
 - pomaga zmieniać te zachowania celem zmaksymalizowania wartości wytwarzanej przez Zespół Scrumowy

Proces Scruma



XP a Scrum

XP	Scrum
Iteracja 1-3 tygodnie	Sprint 1-4 tygodnie
niższy poziom projektu – określa praktyki (np. programowanie w parach, TDD, automatyzacja testów, refaktoryzacja)	wyższy poziom projektu – bardziej dotyczy zarządzania projektem; członkowie mogą wykorzystywać te szczegółowe praktyki, ale nie muszą
klient może zmieniać wymagania w dowolnym czasie	po starcie Sprintu klient nie może zmienić wymagań (<i>Sprint Backlog</i>), tj. czeka na zakończenie Sprint
wymagania opracowywane w ścisłe określonej kolejności (wg priorytetów)	zespół decyduje o kolejności realizacji wymagań
rola moderatora (<i>coach</i>) jest mniej formalna i może przechodzić między członkami zespołu	Mistrz Scruma powinien posiadać stosowny certyfikat do tej roli

Metodyka zwinna – zasady

Zasady

- **podstawą jest komunikacja!**
 - bo mocno ograniczona dokumentacja
- **otwarta przestrzeń dla zespołu**
- **działania wspierające**
 - tester wspomaga klienta (testy akceptacyjne)
 - obserwator (*tracker*) śledzi postępy projektu i ocenia trafność szacunków
 - moderator (*coach*) mobilizuje zespół, wskazuje zaniedbania, ogranicza zbędne dyskusje,
 - konsultant pomaga rozwiązać problemy przekraczające możliwości zespołu
- **dyrekcja podpisuje umowę, przydziela zasoby – liczebność i skład zespołu**

Zasady – reguły tworzenia kodu

- rozwiązywanie problemów **bieżących** a nie przyszłych
 - zaprzecza tradycji projektowania ograniczającej koszty zmian
- koszty zmian nie muszą rosnąć szybko w małym projekcie
 - po co płacić teraz za (możliwe) przyszłe zmiany
 - przyrostowy charakter
 - praca tylko nad bieżącymi historiami i zadaniami
 - architektura rozwiązania
 - do bieżących potrzeb

Zasady – reguły tworzenia kodu

- architektura **nie powstaje jednorazowo**
 - tylko etapami (poszczególne zadania)
- brak optymalności
 - zmiany mogą pogarszać strukturę i utrudniać kolejne zmiany
 - po pewnym czasie zagmatwana struktura

Zasady – reguły tworzenia kodu

- rozwiązanie
 - **refaktoryzacja kodu**
 - okresowa modyfikacja i optymalizacja kodu bez zmiany funkcji
- kto refaktoryzuje?
 - programista
 - możliwość ulepszeń
 - stary kod wymaga zmiany w celu dołączenia nowego
 - zespół
 - gdy spadnie szybkość projektu
 - zatrzymanie prac i duża refaktoryzacja
- po refaktoryzacji konieczna weryfikacja nowego rozwiązania
 - wykonanie wszystkich testów

Zasady – reguły tworzenia kodu

- refaktoryzacja wymaga uzgodnienia i stworzenia nowej architektury rozwiązania
 - techniki projektowania obiektowego
 - techniki mniej sformalizowane
 - nawet bez wspomagania komputerowego

Zasady – reguły tworzenia kodu

- zwykle jednostka jest własnością tego, kto napisał
- jedynie autor modyfikuje
 - ponosi odpowiedzialność
- ale częsta refaktoryzacja
 - w metodyce zwinnej brak indywidualnej odpowiedzialności
 - kod jest własnością zespołu
 - na równych prawach
 - testy są zabezpieczeniem
- współwłasność podnosi rangę standardów kodowania
 - napisany w standardowy sposób → zrozumiałы
 - wszyscy przestrzegają standardów kodowania

Zasady – metody pracy

- programowanie w parach
 - tryb 1
 - jeden pisze
 - drugi ocenia i wyłapuje pomyłki
 - dyskusje o sposobie rozwiązania
 - nawet na kartkach
 - tryb 2 – podział pracy
 - jeden pisze
 - drugi układa testy
 - ale obie osoby są autorami i znają wszystkie szczegóły

Zasady – metody pracy



http://en.wikipedia.org/wiki/File:Pair_programming_1.jpg



http://www.grokpodcast.com/images/2012/02/pair_programming.jpg

Zasady – metody pracy

- pary nie są stałe
 - do rozwiązania doraźnych problemów
- rotacja sprzyja wymianie idei
- gdy konieczna zmiana nieznanego fragmentu
 - możliwe znalezienie osoby, która zna dany fragment
- koszty
 - dwie osoby wykonują pracę jednej osoby

Zasady – metody pracy

- zalety programowania w parach
 - jest narzędziem weryfikacji
 - zastępuje przeglądy i inspekcje kodu
 - jest środkiem komunikacji
 - zastępuje dokumentację
 - jest sposobem na udzielanie pomocy
 - różne zdolności ludzi i różna znajomość fragmentów programu

Zasady – metody pracy

- programowanie w parach wymaga treningu
- nie jest obowiązkowe w każdym projekcie
 - zależnie od korzyści/kosztów
- poprawia się jakość programu
 - obniża koszt poprawek
 - zwiększa zadowolenie klienta
- przemęczeni ludzie nie pracują dobrze
 - metodyka zwinna nie uznaje nadgodzin!
 - takie zaplanowanie, żeby można wykonać w danym czasie
 - zakres iteracji
 - rozdział pracy przy planowaniu iteracji

Zasady – manifest zwinności

- istnieje szereg metod zwinnych
- najbardziej popularne
 - Extreme Programming (XP), Scrum
- inne
 - Crystal, Feature Driven Development
- różne szczegóły, ale wspólne założenia
 - manifest

Zasady – manifest zwinności

- *Agile Manifesto*
- **największą wartość w projekcie przedstawiają**
 - ludzie i ich komunikacja
 - a nie procesy i narzędzia
 - działające oprogramowanie
 - a nie olbrzymia dokumentacja
 - stała współpraca z klientem
 - a nie wynegocjowane umowy
 - reagowanie na zmiany
 - a nie trzymanie się planu

Praktyki zwinne

- Ograniczenia
 - wielkość zespołu
 - max 10 osób
 - elastyczność metody i szybkość reakcji na zmiany
 - nie pasuje do wielkich projektów ze sztywnymi wymaganiami
 - ograniczenia prawne
 - umowy zawierane przez przedsiębiorstwa i administrację publiczną muszą zawierać – zakres, termin, koszt
 - nie do pogodzenia z wymaganiem umieszczenia tylko dwóch z nich

Praktyki zwinne

- niezbyt duża liczba projektów zwinnych
- ale zasady wdrażane w tradycyjnie realizowanych projektach
 - ranga komunikacji i praca w otwartej przestrzeni
 - stosowana w wielu firmach programistycznych
 - częsta dostawa kolejnych wersji
 - wdrażana we wszystkich procesach iteracyjnych, np. RUP
 - ograniczenie zakresu modelowania
 - modele mają być do tworzenia kodu a nie do dokumentowania pracy
 - wiele firm nie przywiązuje wielkiej wagi do formalnej poprawności modeli
 - zmiana roli dokumentacji
 - opisywanie realnego projektu a nie zamiarów
 - kluczowa rola testowania
 - programowanie sterowane testami
 - testy regresywne
 - automatyzacja testów
- brak uznania programowania w parach – potrzebny trening
- brak potwierdzenia użyteczności definiowania architektury przez metaforę [CMU]

Podsumowanie

metodyki wytwarzania
oprogramowania:

RUP

i

zwinna

Pytania



Źródła

- Sacha K., Inżynieria oprogramowania, PWN 2010
- Rational Unified Process. Best Practices for Software Development Teams, Rational Software White Paper, 2001
- Rational Unified Process
http://pl.wikipedia.org/wiki/Rational_Unified_Process
- Sutherland J., Schwaber K., The Scrum Guide. Przewodnik po Scrumie: Reguły gry, 2011
<http://www.scrum.org/Scrum-Guides>

Następny wykład

Inżynieria wymagań

Inżynieria oprogramowania

Wykład 3: Inżynieria wymagań

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Agenda

- Klasyfikacja wymagań
- Proces określania wymagań
- Pozyskiwanie i dokumentowanie wymagań
- Prototypowanie
- Zarządzanie wymaganiami
- Problemy produkcji specyfikacji
- Jakość wymagań

Określenie wymagań

- jest miejscem styku interesów udziałowców
 - klient
 - użytkownik
 - kierownik projektu
 - deweloper
 - tester
 - sprzedawca
- przez / dla różnych udziałowców różne
 - przeznaczenie
 - język
 - poziom szczegółowości

Określenie wymagań

- wymagania zmieniają się w czasie
 - potrzeby
 - technologie
 - uwarunkowania prawne
- proces określania wymagań może być długotrwały
 - możliwość błędów
 - system nie spełnia oczekiwania
 - poprawienie wymaga istotnej przebudowy
 - koszty
- potrzebne staranne
 - dokumentowanie
 - uzgadnianie
 - zatwierdzanie



Klasyfikacja wymagań

Wymaganie

- **czym jest wymaganie?**
 - każda właściwość oprogramowania potrzebna użytkownikowi
 - do zaspokojenia potrzeb
 - do zatwierdzenia gotowego produktu
 - zapis dokumentujący właściwości
- **zatem**
 1. istnieje wiele postaci wymagań
 2. wymagania muszą określić wszystkie rodzaje właściwości potrzebnych użytkownikowi
 3. wymagania muszą być udokumentowane

Wymaganie

czego potrzebuje użytkownik?

co system musi zrobić, żeby zaspokoić potrzeby użytkownika?

co system musi zrobić, żeby zaspokoić potrzeby biznesowe?

co każdy podsystem musi zrobić?

jak podsystemy współpracują ze sobą?

co każdy komponent musi zrobić?

jak komponenty współpracują ze sobą?

Wymagania są listą TO-DO dla zespołu projektowego

Określenie wymagań

- **to proces**
 - budowania
 - dokumentowania
- **kolejnych postaci wymagań**
- rodzaje wymagań
 - względem zakresu
 - względem języka i poziomu szczegółowości

Zakres wymagań

- **funkcjonalne** (*functional*)
 - określają zestaw funkcji realizowanych przez oprogramowanie
- **niefunkcjonalne** (*non-functional*)
 - określają jakość i granice realizowania tych funkcji
- **wymagania zgodności** (*compliance*)
 - trudne do sklasyfikowania jako funkcyjonalne lub niefunkcyjonalne
 - np. zgodność działania z obowiązującym prawem
 - czy wymaga to dodania funkcji?
 - czy wymaga to dostosowania innych właściwości?

Wymagania funkcjonalne

- przykład
 - system powinien prowadzić indywidualne konta klientów banku
 - system powinien księgować wszystkie wpłaty i wypłaty na koncie klienta
 - system powinien umożliwiać posiadaczowi konta wykonanie przelewu na dowolny inny rachunek bankowy
 - system powinien przechowywać historię wszystkich operacji wykonanych na koncie
- **każdy punkt → jedna funkcja**
- opis jest zgrubny i nie zawiera szczegółów
 - czy klientami będą osoby, firmy czy obie?
 - jakie dane identyfikacyjne?
 - czy przy zmianie nazwiska/nazwy zamykane konto i tworzone nowe?
 - w jakich walutach mają/mogą być konta?

Wymagania funkcjonalne

- w różnych fazach opracowania oprogramowania
 - różne poziomy szczegółowości
 - poziom szczegółowości rośnie z czasem
 - realizacją kolejnych etapów
- **nie narzucać przedwcześnie sposobu implementacji**
 - przykład
 - „*system powinien chronologicznie pamiętać wszystkie transakcje i wyświetlać je na żądanie użytkownika*
 - narzucenie sposobu organizacji
 - prawdziwe wymagania – chronologiczne wyświetlanie

Wymagania niefunkcjonalne

- funkcje nie określają wszystkich aspektów z punktu widzenia użytkownika
 - czas przetwarzania: szybciej – wolniej
 - liczba kont: mała – duża
 - czas wznowienia po awarii: szybki – wolny
- przydatność zależy od
 - wydajności
 - niezawodności
 - bezpieczeństwa
 - wygody użytkowania
 - przenośności
 - łatwości konserwacji
 - ...
- różna waga cech w zależności od zastosowania

Wymagania niefunkcjonalne

- zdefiniowanie nie jest łatwe
 - „*ma być wysoka wydajność*”
 - subiektywne → spory
- **określenie w postaci ilościowej**
 - system powinien obsługiwać nie mniej niż 300 000 kont
 - średni czas wykonania transakcji na koncie nie powinien przekroczyć 2 ms; żadna transakcja nie powinna trwać dłużej niż 10 ms
 - maksymalny czas niesprawności systemu po awarii \leq 8 godzin
 - system powinien gwarantować prawidłowe zachowanie stanu konta pomimo awarii bazy danych
- część wymagań odnosi się tylko do oprogramowania
 - ale też do sprzętu
 - a zatem do całości systemu informatycznego

Wymagania zgodności

- oprogramowanie musi / powinno funkcjonować zgodnie z
 - regulacjami prawnymi
 - standardami
 - zwyczajami
- na przykład
 - ustanowiona ochronie danych osobowych
 - branżowe standardy
 - w systemach przemysłowych

Wymagania niefunkcjonalne i zgodności

Minimalna liczba pacjentów, których jednostka ratunkowa powinna móc obsłużyć jednocześnie to 100.

Czas reakcji

Ograniczenia

Po otrzymaniu ostrzeżenia o ataku statek powinien móc wystartować helikoptery w ciągu 3 minut.

Kierowca karetki powinien stosować się do wszystkich zasad ruchu drogowego.

Pojemność

Liczbowe ujęcie wymagań

Czyni wymagania testowalnymi

Pravo

Wydajność

Obejmuje dostępność, pokrycie, czytelność, ...

Definiuje przestrzeń kompromisów

Poziomy opisu wymagań

- zidentyfikowanie i opisanie wymagań
 - dużej wiedzy na temat potrzeb
 - technologicznych możliwości
 - kosztów i ryzyka różnych wariantów
- na początku tej wiedzy nie ma
 - dlatego określenie wymagań **nie jest czynnością**
 - ale procesem → kolejne poziomy szczegółowości
 - rośnie wiedza → świadomość podejmowania decyzji

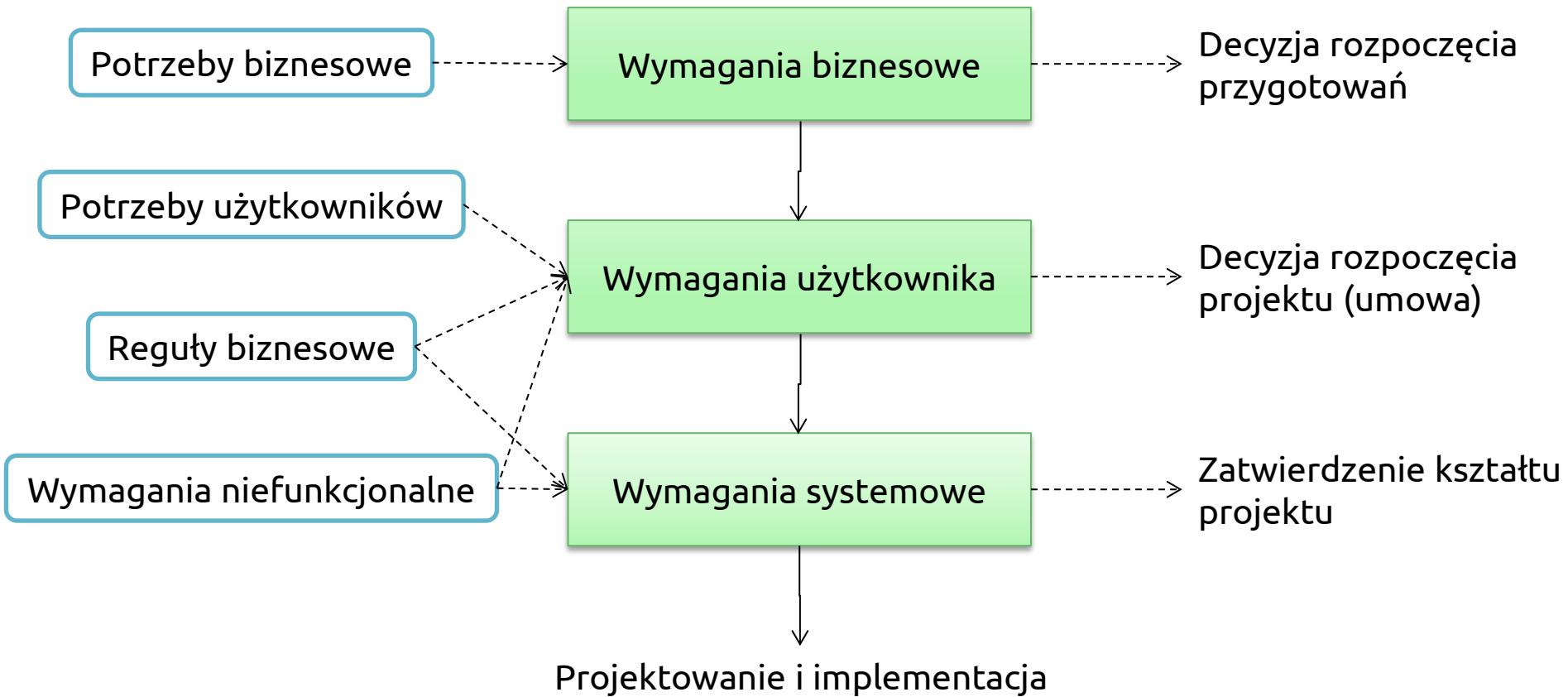
Poziomy opisu wymagań

- **Problem**
 - opis problemu i jego kontekstu
 - czego udziałowcy oczekują od systemu
 - nie opisuje rozwiązania oprócz środowiska
 - jakość efektów
 - właścicielem jest udziałowiec lub przedstawiciel
- **Rozwiążanie**
 - abstrakcyjna reprezentacja rozwiązania
 - co robi system
 - nie opisuje projektu
 - jak dobre jest rozwiązanie
 - właścicielem jest inżynier systemowy

Użytkownik może zrobić ...

System powinien zrobić ...

Poziomy opisu wymagań i podejmowane decyzje



Unikanie przedwczesnych szczegółów

Co jest celem?

Skupienie na problemie nie rozwiązaniu

Co jest zasadniczym celem?



Wyrażenie wymagań bez proponowania rozwiązań

Ułatwia prawidłowe wyrażenie wymagania

Ułatwia liczbowe wyrażenie wymagania

Czy celem jest maksymalizacja, minimalizacja czy optymalizacja?



Czy jest ukryte rozwiązanie?

Zrozumienie dlaczego określone rozwiązanie może być potrzebne

Zebranie ograniczeń prowadzących do określonego rozwiązania

Skąd wiadomo, że potrzeba jest spełniona?

Czyni wymagania ujętymi liczbowo i testowalnymi

Proces określenia wymagań

Źródła wymagań

biznesowe potrzeby klientów

produkty ogólnego przeznaczenia (masowe)

- dział marketingu

produkty na zamówienie

- zarządy odbiorców

proces ustalania potrzeb i planowania sposobu zaspokajania

- planowanie strategiczne

Proces planowania strategicznego

określenie potrzeb informacyjnych firmy

ocena aktualnego stanu informatyzacji i ograniczeń/uwarunkowań

budowa docelowego modelu systemu

opracowanie wstępniego planu realizacji

Proces planowania strategicznego

1. określenie potrzeb informacyjnych firmy

- cel: **znalezienie obszarów działania, które można usprawnić**
- metoda:
 - analiza struktury organizacyjnej
 - analiza obszarów działania
 - rozpoznanie problemów w dotychczasowym środowisku
 - wskazanie celów biznesowych

Proces planowania strategicznego

2. ocena aktualnego stanu informatyzacji i ograniczeń/uwarunkowań

– cele:

- przegląd posiadanych systemów IT
- analiza możliwości ich wykorzystania do zaspokojenia potrzeb organizacji
- określenie czynników ograniczających swobodę podejmowania decyzji

– metoda:

- inwentaryzacja posiadanych systemów
- ocena wydajności systemów
- ocena możliwości rozbudowy
- przegląd kadry
- analiza aktów prawnych i norm
- identyfikacja współpracujących systemów zewnętrznych
- analiza źródeł finansowania rozwoju

Proces planowania strategicznego

3. budowa docelowego modelu systemu

- cel: **określenie kształtu przyszłych rozwiązań**
 - uwzględnienie uwarunkowań
 - minimalizacja nakładów
 - zachowanie elementów istniejących systemów
- metoda:
 - zestawienie potrzeb z możliwościami istniejących systemów
 - wskazanie i uporządkowanie luk
 - zdefiniowanie nowych systemów
 - zdefiniowanie działań osiągnięcia docelowego stanu

Proces planowania strategicznego

4. opracowanie wstępnego planu realizacji

- cel:
 - ocena możliwości i kosztu realizacji systemów
 - określenie odpowiedzialności za przygotowanie projektów
 - ramowy harmonogram realizacji
 - poziom nakładów i sposobu finansowania
- metoda:
 - przeprowadzenie studium wykonalności
 - przegląd kadry zarządczych
 - szacowanie kosztów
 - analiza finansowych możliwości organizacji

Proces planowania strategicznego

- podstawowy problem na początku planowania strategicznego
 - uporządkowanie informacji i wybór kierunku działania
 - analiza SWOT
- decyzja o realizacji wybranych projektów wymaga
 - dokładnego określenia wymagań, zbadania możliwości i kosztów
 - studium wykonalności → szybka symulacja wykonania projektu
- do negocjacji i podpisania umowy
 - wymagania użytkownika
 - metody zależne od metodyki analizy i projektowania

Proces planowania strategicznego

SWOT – 1. ocena sytuacji

- silne i słabe strony
- szanse i zagrożenia w otoczeniu

Silne strony		Słabe strony	
rozbudowana sieć placówek	10	brak centralnego systemu bankowego	10
gotowy model procesów biznesowych	6	duże zaangażowanie kadry w opóźniony projekt	9
dobre stosunki z producentem obecnego systemu	6	brak środków do zakończenia inwestycji	7
dobra grupa analityczno-projektowa	4	duże środki zaangażowane w projekt	7
dodatni bilans działalności operacyjnej	3	brak dostatecznej kadry	6
określona strategia prowadzenia projektu	2	mała skala operacji banku	3
		brak dobrego menedżera banku	2

Proces planowania strategicznego

SWOT – 2. zestawienie w tabeli SWOT

Silne strony	[%]	Ślabe strony	[%]
rozbudowana sieć placówek	14,7	brak centralnego systemu bankowego	14,7
gotowy model procesów biznesowych	8,8	duże zaangażowanie kadry w opóźniony projekt	13,3
dobre stosunki z producentem obecnego systemu	8,8	brak środków do zakończenia inwestycji	10,3
dobra grupa analityczno-projektowa	5,9	duże środki zaangażowane w projekt	10,3
dodatni bilans działalności operacyjnej	4,4	brak dostatecznej kadry	8,8
suma	42,6	suma	57,4
Szanse	[%]	Zagrożenia	[%]
możliwość rozwoju działalności detalicznej	12,2	dalszy odpływ klientów	13,5
możliwość obsługi gmin i powiatów	10,8	potrzeba dużych środków na dokończenie inwestycji	9,5
możliwość obsługi lokalnych przedsiębiorców	10,8	groźba kar umownych za zerwanie umowy	8,5
niski koszt modernizacji obecnego systemu	9,5	niejasne perspektywy serwisowania realizowanego systemu	8,1
możliwość modernizacji przez wytwórcę	9,5	groźba sporu sądowego o zerwanie umowy	6,8
suma	52,7	suma	47,3

dominują słabe strony i szanse → strategia naprawcza (usunięcie słabości przeszkadzających szansom)

Proces planowania strategicznego

SWOT – 3. tabela korelacji czynników domin.

Szanse Słabe strony	działalność detaliczna	obsługa gmin i powiatów	obsługa firm lokalnych	niski koszt modernizacji	wytwórcza modernizuje	Σ
brak centralnego systemu bankowego	1	0	1	0	0	2
duże zaangażowanie kadry w opóźniony projekt	1	1	1	0	1	4
brak środków do zakończenia inwestycji	0	0	0	1	0	1
duże środki zaangażowane w projekt	0	0	0	0	0	0
brak dostatecznej kadry	0	0	0	1	1	2
Σ	2	1	2	2	2	

Proces planowania strategicznego studium wykonalności

- *feasibility study*
- obejmuje
 - dokładniejsze wymagania
 - analiza wariantów rozwiązań
 - wiarygodne określenie czasu i kosztu wykonania
- wynik → raport wykonalności
 - do podjęcia decyzji o rozpoczęciu projektu
- kto przeprowadza
 - inwestor we własnym zakresie
 - firma konsultingowa
- powinno trwać krótko
 - celem ocena całości, a nie szczegółów

Proces planowania strategicznego studium wykonalności

czynniki
techniczne

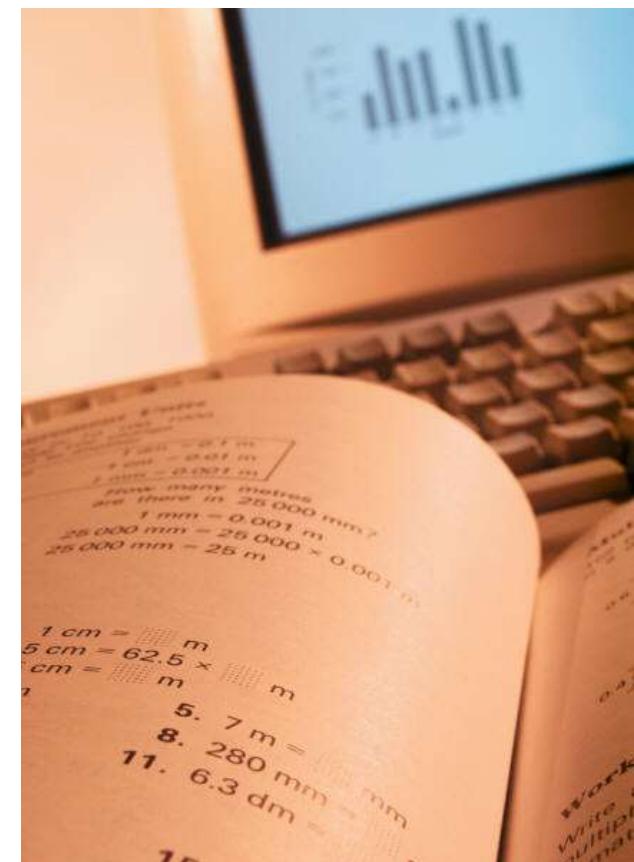
czynniki ludzkie
i organizacyjne

opłacalność
ekonomiczna

czynniki
prawne

Proces planowania strategicznego stud. wykonalności – czynniki techniczne

- **cel**
 - znalezienie rozwiązania ze stanu aktualnego do docelowego
- **problem**
 - identyfikacja i analiza krytycznych obszarów projektu
 - ze skumulowanym ryzykiem
 - zmiany
 - nowe konstrukcje
 - nowe metody i technologie
 - pozostałe obszary badane w sposób zgrubny
- **wynik**
 - określenie wymaganych funkcji
 - określenie zbiorów danych
 - wskazanie wariantów ich realizacji
 - architektura
 - rozmiar
 - strategia budowy
 - następnie → pracochłonność, czas i koszt



Proces planowania strategicznego stud. wykonalności – czynniki techniczne

- kategorie problemów
 - konieczność
 - zreorganizowania struktury organizacji
 - wprowadzenia nowych procesów biznesowych
 - przejęcie części ręcznych operacji przez system
 - eliminacja stanowisk
 - potrzeba wykonywania innych działań
 - zmiana charakteru pracy ludzi
 - reorganizacja zatrudnienia
 - szkolenia
 - kontakty z otoczeniem
 - możliwości adaptacji otoczenia

Proces planowania strategicznego stud. wykonalności – czynniki ludzkie i org.

- cel
 - zbadanie zakresu potrzebnego dostosowania organizacji i ludzi do osiągnięcia celu projektu
- dostosowania
 - mogą generować koszty
 - zakres może utrudniać wykonanie i wdrożenie rezultatów
- opór ludzi
 - nawet klęska całego przedsięwzięcia



Proces planowania strategicznego stud. wykonalności – opłacalność ekonom.

- tradycyjna miara opłacalności
 - zwrot z inwestycji (*return of investment – ROI*)
 - zysk oczekiwany (lub osiągnięty) / poniesione nakłady
- jeśli zysk realizowany w ciągu kilku lat
 - wewnętrzna stopa zwrotu (*internal return ratio – IRR*)
 - stopa oprocentowania kapitału przy której zysk z inwestycji w ciągu n lat równoważy wartość nakładów oprocentowanych na r procent rocznie
 - znalezienie odpowiedniej wartości r

$$\sum_{t=1}^n \frac{zysk_t}{(1+r)^n} = naklad$$



Proces planowania strategicznego stud. wykonalności – opłacalność ekonom.

- stosowanie miar łatwe dla produktów masowych
 - zysk zależny od wielkości sprzedaży
- produkty zamawiane
 - trudniejsze oszacowanie zysku
- źródła zysku
 - spodziewana redukcja zatrudnienia – **łatwe oszacowanie**
 - **trudne**
 - poprawa działania przedsiębiorstwa
 - możliwość podjęcia nowych działań
 - zdobycie przewagi konkurencyjnej

Proces planowania strategicznego stud. wykonalności – opłacalność ekonom.

- obliczenie nakładów zwykle prostsze
- koszty ponoszone od razu
 - opracowanie oprogramowania
 - w tym narzędzi pomocniczych
 - zakupu i instalacji infrastruktury
 - szkolenia personelu
 - i spadku wydajności w okresie wdrożenia
 - pomieszczeń i energii
- koszty ponoszone w trakcie eksploatacji
 - eksploatacja sprzętu, sieci, licencje oprogramowania, serwis
 - konserwacja i ewolucja
 - amortyzacja sprzętu
- wycena ma ułatwić porównanie i selekcję wariantów

Proces planowania strategicznego stud. wykonalności – czynniki prawne

- mogą być źródłem wymagań zgodności
- stwarzają ograniczenia
 - na przykład gromadzenia określonych danych
- stwarzają obowiązki
 - na przykład gromadzenia i zabezpieczenia określonych danych
- przestrzeganie praw autorskich
- wyłanianie wykonawcy w przetargu
 - np. w administracji publicznej



Przygotowanie wykonania projektu

- jeśli projekt własnymi siłami
 - rozpoczęcie zależne
 - od wewnętrznej decyzji zarządu
 - przydzielenia zasobów
 - ludzi
 - pomieszczeń
 - infrastruktury technicznej
- jeśli projekt zlecony zewnętrznemu wykonawcy
 - wybór wykonawcy i zawarcie umowy
 - najczęściej w formie przetargu

Przygotowanie wykonania projektu

1. rozpisanie przetargu

- zamawiający publikuje specyfikację istotnych warunków zamówienia (SIWZ)
 - podstawowe informacje o zamawiającym
 - zadania, struktura i ramy prawne przedsiębiorstwa
 - charakterystyka posiadanych systemów IT
 - wskazanie problemów i mankamentów istniejących rozwiązań
 - opis przedmiotu zamówienia
 - cel i zakres projektu
 - wymagania funkcjonalne i niefunkcjonalne
 - ograniczenia realizacyjne
 - oczekiwany termin i harmonogram wdrożenia
 - wymagania dodatkowe
 - projekt umowy realizacyjnej
 - projekt organizacji zarządzania przedsięwzięciem
 - wymaganie przedstawienia referencji – wiarygodność oferenta
- zamawiający powołuje komisję przetargową
 - ocena zgodności złożonych ofert z kryteriami
 - wybór wykonawcy

Przygotowanie wykonania projektu

2. opracowanie oferty

- opublikowanie SIWZ → dla wykonawcy punkt startowy procesu wytwórczego
- otrzymuje
 - kontekst przedsięwzięcia
 - wymagania użytkownika
 - wymagania dodatkowe – ograniczenia
- konieczna analiza tych danych (strategiczna)
 - aby rozwinąć wymagania
 - stworzenie koncepcji rozwiązania
 - określenie ogólnej architektury
 - harmonogram wykonania
 - pracochłonność i koszt

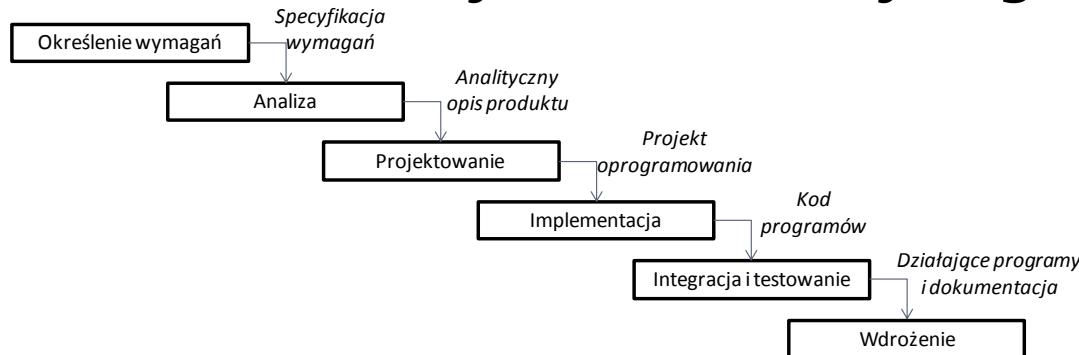
Przygotowanie wykonania projektu

2. opracowanie oferty

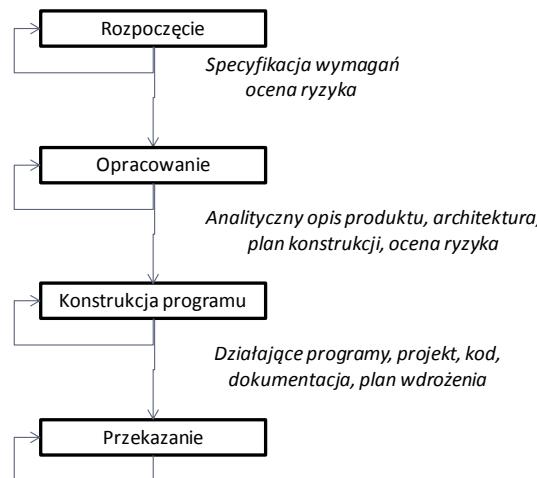
- typowe elementy oferty
 - charakterystyka proponowanego rozwiązania
 - projekt koncepcyjny
 - opis architektury
 - opis interfejsu
 - technologia realizacji
 - dokładny opis rozwiązań wybranej części systemu
 - zgodnie z wymaganiami SIWZ
 - organizacja procesu wytwórczego
 - metoda realizacji
 - charakterystyka zespołów roboczych
 - kosztorys i harmonogram
 - inne informacje wymienione w SIWZ
 - pełnomocnictwa reprezentantów firmy
 - referencje z poprzednich projektów

Przygotowanie wykonania projektu

- wybór oferty i zawarcie umowy oznacza rozpoczęcie realizacji przedsięwzięcia
 - zakończenie fazy określenia wymagań (kaskadowy)



- zakończenie fazy rozpoczęcia (iteracyjny)



Przygotowanie wykonania projektu

- opracowanie SIWZ i oferty (przez różne podmioty) obejmują prace analityczne
 - jakość tych prac → powodzenie całego przedsięwzięcia
 - wszystkie prace przed podpisaniem umowy
 - bez gwarancji zysku
 - dlatego ważne ustalenie
 - czasu trwania prac
 - stopnia szczegółowości opracowań
 - nie za długo
 - koszty
 - celem nie jest dokładna analiza szczegółów, ale
 - ustalenie i uzgodnienie wymagań
 - wiarygodne koszty
 - podjęcie decyzji o rozpoczęciu/zaniechaniu projektu

Przygotowanie wykonania projektu

- w projektach dla masowego odbiorcy
 - proces ulega zmianie
- faza planowania strategicznego → faza badania rynku
 - odkrywanie potrzeb potencjalnych odbiorców
 - przez dział marketingu
 - weryfikacja propozycji marketingowych
 - w czasie studium wykonalności
- postępowanie przetargowe → analiza strategiczna
 - przez własne działy rozwojowe
 - wyniki analizy do podjęcia decyzji o realizacji / odrzuceniu projektu

Przekształcanie wymagań – dekompozycja

Submarine shall travel at 20 knots when submerged

Hull length : width ratio shall be [TBD] +/- 5%

Power transferred to water shall be at least [TBD] KW

Coefficient of drag shall be less than [TBD]

Przekształcanie wymagań – dekompozycja

SHR 21: The driver shall be able to deploy the vehicle over terrain type 4A.

Note:
Terrain type 4A specifies soft wet mud, requiring constraints on weight, Clearance, and power delivery.

SR 15: The vehicle shall transmit power to all wheels.

SR 32: The vehicle shall have ground clearance of not less than 12 inches.

SR 53: The vehicle shall not weigh more than 1.5 tons.

Przekształcanie wymagań – alokacja

$$30 = 15 + 10 + 5$$

Total system power consumption shall not exceed 30 W

Subsystem 1 power consumption shall not exceed 15W

Subsystem 2 power consumption shall not exceed 10W

Subsystem 3 power consumption shall not exceed 5W

Przekształcanie wymagań – bezpośrednie zastosowanie

bezpośrednie zastosowanie

All external surfaces shall be finished to int-std-34453



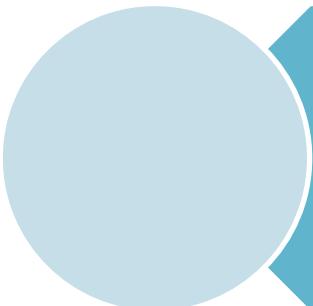
All external surfaces shall be finished to int-std-34453

Pozyskiwanie i dokumentowanie wymagań

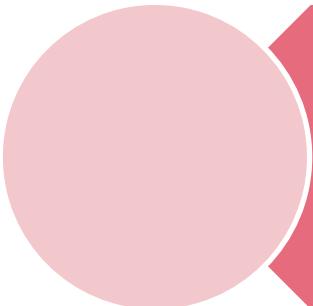
Pozyskiwanie i dokumentowanie wymagań

- tylko w nielicznych przypadkach organizacja sama
 - wykonuje prace analityczne
 - opracowuje specyfikację wymagań
- zakres tych prac jest inny niż codzienna praca
 - brak odpowiedniej kadry
- rozwiązanie
 - zlecenie firmie doradczej
 - zlecenie firmie informatycznej
 - może być późniejszym wykonawcą
- ale wtedy problem
 - brak znajomości dziedziny i celów biznesowych przez zewnętrznych analityków
- dlatego kluczowy czynnik sukcesu
 - zrozumienie użytkowników
 - pozyskanie od nich wymagań
 - udokumentowanie wymagań w zrozumiały sposób

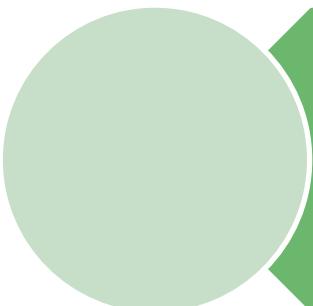
Metody pozyskiwania wymagań



studowanie dostępnej dokumentacji



wywiady z przedstawicielami kierownictwa



obserwacja i analiza obiegu dokumentów

Metody pozyskiwania wymagań

- studiowanie dostępnej dokumentacji
 - akty prawne
 - schemat struktury organizacyjnej
 - regulaminy, procedury i instrukcje stanowiskowe
 - strategia rozwoju
 - opisy funkcjonujących systemów IT
 - opisy produktów konkurencyjnych

Metody pozyskiwania wymagań

- wywiady z przedstawicielami kierownictwa
 - zadania i problemy organizacji
 - podstawowe procesy biznesowe
 - zakresy odpowiedzialności pracowników
 - plany na przyszłość
 - kluczowe aspekty
 - przygotowanie odpowiedniego zestawu pytań
 - dobór respondentów
 - cel wywiadu jasny dla obu stron
 - prowadzenie notatek
 - autoryzacja ostatecznej wersji od respondenta
 - po uporządkowaniu notatek

Metody pozyskiwania wymagań

- obserwacja i analiza obiegu dokumentów
 - szczególnie: sposób tworzenia dokumentów
 - kto
 - w jakim celu
 - na podstawie jakich danych
 - treść
 - rozmiar
 - przeznaczenie
 - jak przechowywane

Metody pozyskiwania wymagań

- dzięki wiedzy z tych źródeł
 - stworzenie modelu działania organizacji
 - struktura
 - podstawowe procesy biznesowe
 - dane
 - budowa modelu
 - rozpoczęcie równolegle ze zbieraniem informacji
 - modyfikacja modelu wraz z nowymi danymi
 - dzięki temu
 - łatwe łączenie informacji z wielu źródeł
 - wykrywanie i usuwanie sprzeczności
 - modele można pokazywać użytkownikom w czasie wywiadów
 - weryfikacja i zatwierdzanie

Metody pozyskiwania wymagań

- jak jest model organizacji
 - to można wskazać obszary podlegające zmianom
- nowy system
 - wspieranie istniejących procesów
 - zmiana przebiegu istniejących procesów
 - nowe działania
- nie ma jednej właściwej postaci modelu
 - zależą od użytych metod
 - strukturalne
 - obiektowe

Specyfikacja wymagań

- jest podstawowym dokumentem przy rozpoczęciu prac nad budową oprogramowania
- **zawartość specyfikacji**
 - **co oprogramowanie ma robić**
 - **nie powinna narzucać – jak ma być zbudowane**
 - narzucenie przedwczesne i niepotrzebnie ograniczające projektanta
- opracowanie wymaga ścisłej współpracy
 - klienta i użytkowników
 - znają dziedzinę zastosowania i potrzeby
 - wykonawcy
 - zna technologie i ich ograniczenia
- żadna ze stron samodzielnie nie ma wystarczających kompetencji



Specyfikacja wymagań

- odmienny zakres działania udziałowców
 - ogranicza język komunikacji
 - utrudnia wykorzystanie specjalistycznych modeli i notacji
- rozwiązanie (?)
 - najczęściej język naturalny
 - wzbogacony formalizmami (np. formułami matematycznymi)
- modele analizy strukturalnej i obiektowej
 - na tym etapie w ograniczonym zakresie

Specyfikacja wymagań

- Badania na Uniwersytecie w Trento (2002) – 151 firm IT
 - 79% w języku naturalnym
 - 16% strukturalizowany język naturalny
 - formularze
 - wzorce pseudokodu
 - 5% specjalizowane języki specyfikacyjne
- **wada języka naturalnego**
 - wieloznaczność zapisu
- **wada języków specjalizowanych**
 - mniejsza czytelność
 - potencjalna możliwość przedwczesnego narzucenia rozwiązań

Specyfikacja wymagań

podstawowa rolą specyfikacji



część umowy między zleceniodawcą a wykonawcą

- efekt
 - usuwanie skutków błędów w specyfikacji trudne i kosztowne
 - defekty zwykle nie są widoczne podczas projektowania i implementacji
 - defekty ujawniają się podczas oceny i prób eksploatacyjnych
 - usunięcie defektów wymaga przeprojektowania, ponownego zaprogramowania, powtórzenia testowania

Specyfikacja wymagań standard ANSI/IEEE 830

- formułuje zalecenia, które powinna spełniać dobrze napisana specyfikacja wymagań
- treść
 - powinna opisywać wszystkie pożądane cechy oprogramowania
 - wymagania funkcjonalne
 - wymagania niefunkcjonalne
 - sprzęgi zewnętrzne
 - narzucone ograniczenia
 - nie powinna zawierać
 - wymagań dotyczących procesu wytwórczego
 - ważne ale nie w tym miejscu → umowa

Specyfikacja wymagań standard ANSI/IEEE 830

- cechy dobrze opracowanej specyfikacji
 - poprawność
 - jednoznaczność
 - kompletność
 - spójność
 - uporządkowanie
 - weryfikowalność
 - modyfikowalność
 - powiązanie (*traceability*)

Specyfikacja wymagań standard ANSI/IEEE 830

- **poprawność**
 - opisanie tylko tych wymagań, które są potrzebne użytkownikom
 - jeśli oprogramowanie jest częścią większego systemu
 - specyfikacja wynika z projektu systemu
 - nie może być sprzeczna z innym dokumentem projektowym
 - w innych przypadkach
 - weryfikacja specyfikacji – subiektywna ocena użytkownika
- **jednoznaczność**
 - zapis każdego wymagania ma tylko jedną interpretację
 - prawie niemożliwe w dokumencie w języku naturalnym
 - unikanie synonimów dla tego samego pojęcia
 - w razie potrzeby ich użycia → umieścić je w słowniku

Specyfikacja wymagań standard ANSI/IEEE 830

- kompletność
 - wymienia wszystkie wymagania funkcjonalne i niefunkcjonalne
 - zdefiniowanie odpowiedzi systemu na możliwe wartości danych wejściowych
 - poprawnych i niepoprawnych
 - we wszystkich stanach programu
 - podpisy i odnośniki w tekście do tabel i rysunków
- spójność
 - opisy nie mogą zawierać sprzeczności
 - na przykład odpowiednia organizacja tekstu
 - każde wymaganie tylko w jednym miejscu
 - użycie jednolitej terminologii

Specyfikacja wymagań standard ANSI/IEEE 830

- **uporządkowanie**
 - jeśli wymagania nie posiadają jednakowej ważności
 - to klasyfikacja pod względem ważności
 - niezbędne
 - pożądane
 - mile widziane
 - poprawia to czytelność dokumentu
 - umożliwia właściwe rozłożenie nakładów
- **weryfikowalność**
 - sformułowanie w sposób umożliwiający ocenę spełnienia przez finalny produkt
 - w szczególności dotyczy wymagań niefunkcjonalnych

Specyfikacja wymagań standard ANSI/IEEE 830

- modyfikowalność
 - struktura i styl umożliwiają wprowadzanie zmian
 - bez naruszania spójności dokumentu
 - rozwiązań
 - podział na rozdziały i punkty
 - każde wymaganie w jednym punkcie
 - związki między pokrewnymi punktami za pomocą odsyłaczy
- powiązanie (*traceability*)
 - wskazanie źródła pochodzenia każdego wymagania
 - np. numer punktu wcześniejszego dokumentu, aktu prawnego, normy branżowej
 - wszystkie punkty powinny być numerowane
 - umożliwienie powiązania z dokumentami projektowymi

Specyfikacja wymagań standard ANSI/IEEE 830 – wzorzec treści

- wstęp
- opis ogólny
- wymagania szczegółowe
- indeks i dodatki

Specyfikacja wymagań standard ANSI/IEEE 830 – wzorzec treści

- **wstęp**
 - cel dokumentu i krąg czytelników
 - zakres specyfikacji – nazwa produktu, obszar zastosowania
 - słownik pojęć i skrótów
 - lista dokumentów powiązanych
 - wraz ze sposobem dostępu do nich
 - streszczenie – zawartość i struktura pozostałej części
- **opis ogólny**
 - opis otoczenia – systemy, interfejsy komunikacyjne, ludzie
 - podstawowe funkcje – porządek zgodny z biegiem procesów biznesowych
 - opis ograniczeń – regulacje prawne, wymogi bezpieczeństwa, wymagania zgodności, itp.
 - założenia, sprawy nierostrzygnięte, wymagania pominięte w tej wersji

Specyfikacja wymagań

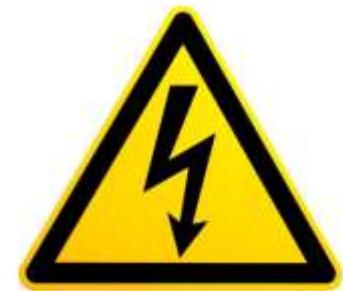
standard ANSI/IEEE 830 – wzorzec treści

- **wymagania szczegółowe**
 - **opis danych** – rodzaje danych wprowadzanych i wytwarzanych
 - dokumenty, raporty, ekranы, komunikaty
 - **opis funkcji**
 - dla każdej funkcji dane wejściowe, algorytm przetwarzania danych prawidłowych i nieprawidłowych, wyniki, warianty działania
 - **opis bazy danych**
 - kategorie danych, powiązania, sposób wykorzystania przez funkcje, więzy integralności, okresy przechowywania
 - **wymagania niefunkcjonalne**
 - wydajność, niezawodność, bezpieczeństwo, zachowanie w razie awarii, przenośność, itp.
 - ilościowo i z odniesieniem do funkcji
 - ograniczenia projektowe
- **indeks i dodatki**
 - indeks – lista używanych pojęć
 - dodatki – przykładowe formaty dokumentów, modele analityczne, wyniki analizy kosztów

Prototypowanie

Prototypowanie

- poleganie na specyfikacji
 - wymaga wyobrażania sobie przyszłego systemu
 - ryzykowne
- ograniczenie tego ryzyka
 - podstawa techniki prototypowania
 - budowa prototypu przed budową rzeczywistego produktu



Prototypowanie

- prototyp
 - **provizoryczna implementacja**
 - tylko wybrane cechy produktu końcowego
- przykład
 - tylko interfejs – menu i nawigacja między oknami
 - brak logiki realizującej funkcje
 - może przyjmować dane wejściowe
 - wyświetla wyniki
 - stałe
 - losowo generowane
- użytkownik może ocenić
 - kompletność danych wejściowych
 - kompletność funkcji
 - dostępność raportów
- w razie potrzeby → szybka zmiana

Prototypowanie

- **prototyp poziomy (*horizontal prototype*)**
 - przedstawia obraz działania programu
 - szeroki
 - ale powierzchowny
 - wytworzenie prototypu nie musi znacznie zmieniać procesu
 - powstaje podczas analizy wymagań
 - wykorzystywany do oceny i zatwierdzania specyfikacji
 - może stać się częścią specyfikacji
 - korzyści
 - lepsze rozpoznanie potrzeb użytkowników
 - zebranie przez programistów doświadczeń

Prototypowanie

- **prototyp pionowy (*vertical prototype*)**
 - implementuje wybrany fragment funkcjonalności
 - zgodnie z docelowym rozwiązaniem
 - przykładowe cele
 - sprawdzenie wydajności wybranej architektury
 - wykazanie wykonalności rozwiązania w wybranej technologii
 - eksperymentalne zbadanie nowatorskiego algorytmu
 - kiedy
 - we wczesnych fazach procesu wytwarzania – studium wykonalności
 - później – projektowanie i ocenianie różnych wariantów projektu
 - wiarygodność oceny rośnie wraz ze stopniem wykorzystania docelowej technologii produkcyjnej
 - korzyść
 - ograniczenie ryzyka wyboru niewłaściwych metod lub technologii

Prototypowanie

sposoby wykorzystania prototypu

1. budowa jednorazowych prototypów

- do badania wymagań lub wariantów
- po zakończeniu etapu
 - **prototyp jest odrzucany**
 - projekt realizowany dalej zgodnie z modelem procesu
- zysk – ułatwia zrozumienie problemu
- cena – dodatkowy koszt budowy prototypu
- **powinien być budowany szybko i tanio**

2. stopniowa rozbudowa możliwości prototypu

- aż do spełnienia wymagań użytkownika
- prototyp przekształca się w implementację oprogramowania
- częściowo realizowane przez procesy iteracyjne, zwłaszcza zwinne
- wyniki iteracji – kolejne prototypy pionowe
- **powinien być budowany wg standardów jakości narzędziami docelowego środowiska**
 - unikanie prowizorycznych rozwiązań

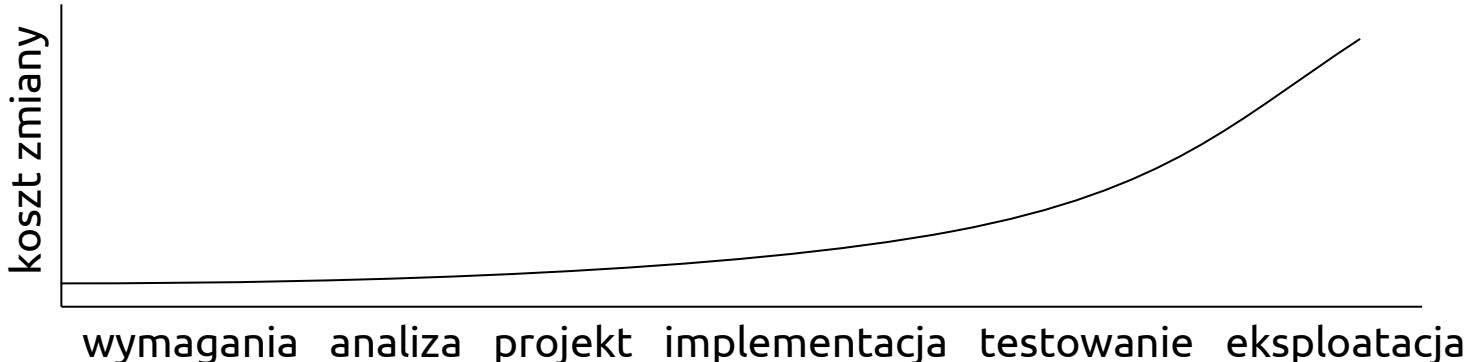
Zarządzanie wymaganiami

Zarządzanie wymaganiami

- określenie wymagań **nie jest jednorazową czynnością**
- proces określania wymagań
 - początek wraz z powstaniem potrzeb
 - obejmuje okres realizacji projektu
 - trwa w czasie eksploatacji
 - nowe wersje

Zarządzanie wymaganiami

- każda zmiana zaburza przebieg procesu
- koszt zmiany rośnie wraz z upływem czasu
 - zaawansowaniem prac



- cel zarządzania wymaganiami
 - stworzenie i utrzymanie spójnego zbioru wymagań
 - zapanowanie nad zmianami
 - ograniczenie kosztów zmian

Zarządzanie wymaganiami

- na początku główny problem
 - odkrywanie, gromadzenie i dokumentowanie wymagań
 - usuwanie sprzeczności między wymaganiami
- wymagania zgłasiane przez różnych użytkowników
 - mogą różnić się
 - konieczne
 - porównywanie wymagań
 - rozstrzyganie konfliktów
- **analityk**
 - wykrywanie konfliktów
 - utrzymywanie spójnego zbioru wymagań
- **klient** – podejmuje decyzje rozstrzygające

Zarządzanie wymaganiami

ocena wymagań

- **cel oceny**
 - czy przyjęty zbiór wymagań poprawnie i kompletnie określa potrzeby i oczekiwania klienta?
- popularne metody oceny
 - przegląd
 - ocena prototypu
 - automatyczne sprawdzanie poprawności
 - tworzenie testów

Zarządzanie wymaganiami

ocena wymagań

- **przegląd**
 - eksperci oceniają jakość specyfikacji
 - przegląd i analiza treści
 - wada
 - poleganie jedynie na wiedzy i wyobraźni ekspertów
- **ocena prototypu**
 - eksperci i użytkownicy oceniają prototyp
 - należy skupić się na **kompletności i użyteczności** funkcji
 - nie na wyglądzie!!!

Zarządzanie wymaganiami

ocena wymagań

- **automatyczne sprawdzanie poprawności**
 - wymagania zapisane są w postaci modelu o matematycznie określonych regułach poprawności
 - narzędzie automatycznie sprawdza te reguły
 - zaleta
 - dowód poprawności modelu
 - wada
 - brak możliwości automatycznego sprawdzenia modelu z potrzebami klienta
- **tworzenie testów**
 - testy akceptacyjne są końcowym sprawdzianem zgodności z wymaganiami
 - treść testów musi obejmować istotne elementy wymagań
 - opracowanie tych testów pomaga w ocenie znaczenia, spójności i wykonalności wymagań

Zarządzanie wymaganiami

- zatwierdzenie specyfikacji **nie gwarantuje niezmienności wymagań**
 - wymagania mogą być usuwane
 - potem ponownie włączane
 - potrzebna historia zmian
 - ponowne użycie wymagań wcześniej zgłoszonych i usuniętych
 - nie powtarzanie przeprowadzonych wcześniej analiz

Nothing is permanent except change

Zarządzanie wymaganiami

- potrzebne narzędzia wspomagające analityków
 - CASE (*Computer-Aided Software Engineering*)
- funkcje
 - wprowadzanie i przechowywanie wymagań
 - w języku naturalnym i/lub za pomocą modeli
 - nadawanie wymaganiom unikalnych oznaczeń, przechowywanie treści wraz z
 - pochodzeniem, powiązaniami, historią zmian
 - przeglądanie, zmienianie, usuwanie wymagań
 - automatyczne śledzenie i aktualizacja powiązań
 - sortowanie i filtrowanie pod względem znaczenia, drukowanie w wymaganej formie
 - zamrażanie obowiązującej wersji
 - porównywanie wersji, automatyczne śledzenie historii zmian
- przykłady
 - Rational DOORS, Rational Requisite Pro, Borland CaliberRM

Problemy produkcji specyfikacji

Problemy produkcji specyfikacji

1. wymagania powstają bez dostatecznego udziału i wsparcia analityków systemowych

– klient i użytkownik – najczęściej

- nie – co system powinien robić
- ale – czym system ma być
- efekty
 - tracą kontrolę, nie panują nad tym, co chcą i potrafią wyrazić
 - opuszczają istotne wymagania
 - » nie potrafią odwzorować
 - » nie widzą sposobu implementacji
 - » implementacja może być zbyt kosztowna
- rezultat
 - przedstawiają nie specyfikację, ale częściowy projekt

Problemy produkcji specyfikacji

2. klienci i użytkownicy nie rozumieją konieczności ujęcia informacji o docelowym środowisku funkcjonowania

- interfejsy z innymi systemami
- ograniczenia operacyjne
- fizyczne umiejscowienie systemu

wtedy

- stopień dopasowania systemu do środowiska zależy od
 - poprawności interpretacji
 - dodatkowych badań
 - stopnia zbadania wymagań przez wykonawcę

Problemy produkcji specyfikacji

3. napisanie przez osobę o nieodpowiednich kwalifikacjach

- rezultat – źle napisany dokument
- niepewność interpretacji
- brak uświadomienia sobie potrzeby uściślenia interpretacji przez wykonawcę
 - efekt
 - błędy projektowe, później implementacyjne
 - wzajemne oskarżenia użytkownik-wykonawca

Problemy produkcji specyfikacji

4. niepoważne traktowanie zadania produkcji specyfikacji przez kierowników

- brak odpowiednich zasobów
- często jedynie jako powinność
 - żeby zachować pozory
- kierownicy wywierają presję na personel
 - jak najszybciej zakończyć etap
 - bez dbania o jakość
- efekt
 - źle przygotowana specyfikacja
 - problemy i koszty w późniejszych fazach i eksploatacji

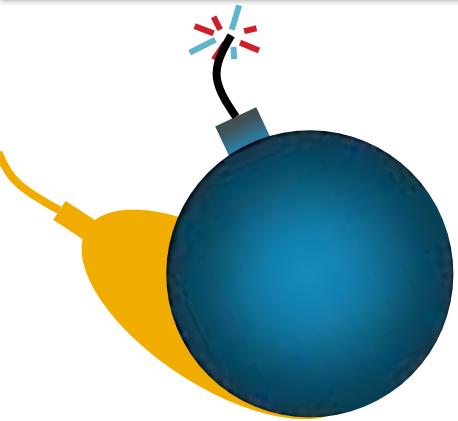
Jakość wymagań

Anatomia dobrego wymagania użytkownika



Wyzwanie polega na ujęciu rodzaju użytkownika, rezultatu i miary sukcesu w każdym wymaganiu.

Unikanie pułapek – wieloznaczność

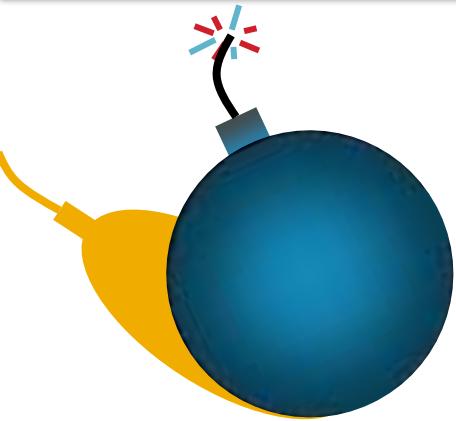


- ✖ ... lub ...
- ✖ ... i tak dalej
- ✖ ... włączając, ale nie ograniczając do ...

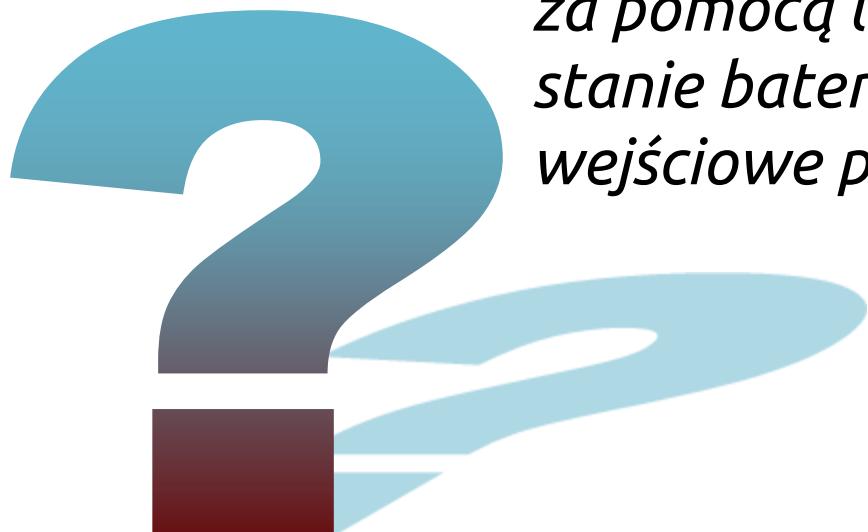


"Pilot lub drugi pilot powinien usłyszeć lub zobaczyć wizualny lub dźwiękowy sygnał ostrzegający w przypadku zagrożenia, nagłego wypadku, itp."

Unikanie pułapek – wielokrotność

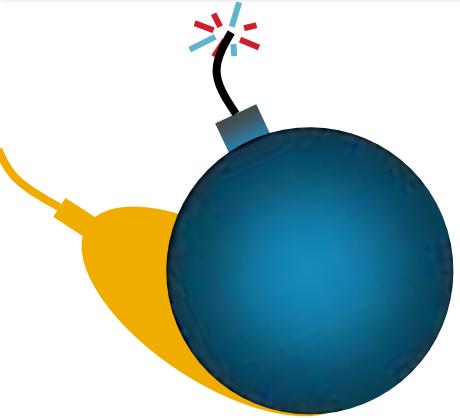


- ✓ Każde wymaganie jako osobne zdanie
- ✗ Koniunkcje
- ✗ ...i..., ...lub..., ...z..., ...również...



“Kiedy napięcie spadnie poniżej 3.6 V, użytkownik powinien zostać powiadomiony za pomocą lampki ostrzegającej o niskim stanie baterii oraz środowisko pracy i dane wejściowe powinny zostać zapisane.”

Unikanie pułapek – warunki

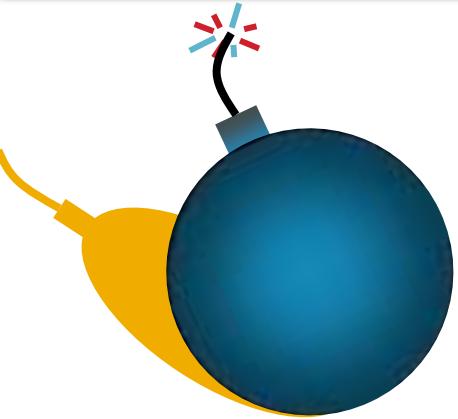


- ✖ klauzule warunkowe
- ✖ ...jeśli... , ...ale... , ...kiedy...
- ✖ ...z wyjątkiem..., ...chyba że... , ...pomimo...

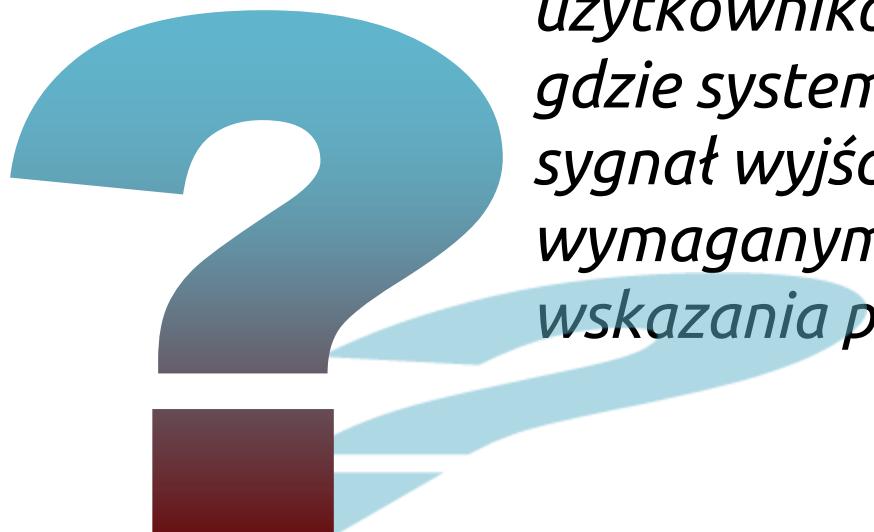
“Właściciel domu powinien zawsze usłyszeć dźwięk czujnika dymu, kiedy dym zostanie wykryty, chyba że czujnik jest testowany lub nieaktywny.”



Unikanie pułapek – chaotyczność

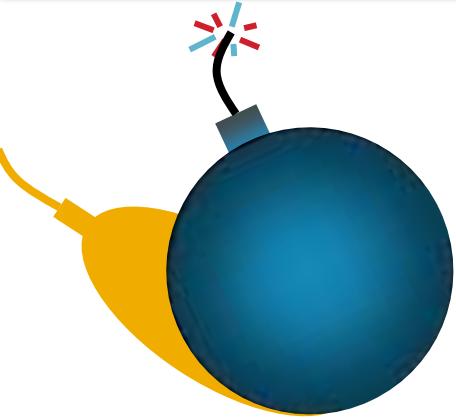


- ✖ długie zdania
- ✖ archaiczny język
- ✖ odnośniki do nieosiągalnych źródeł

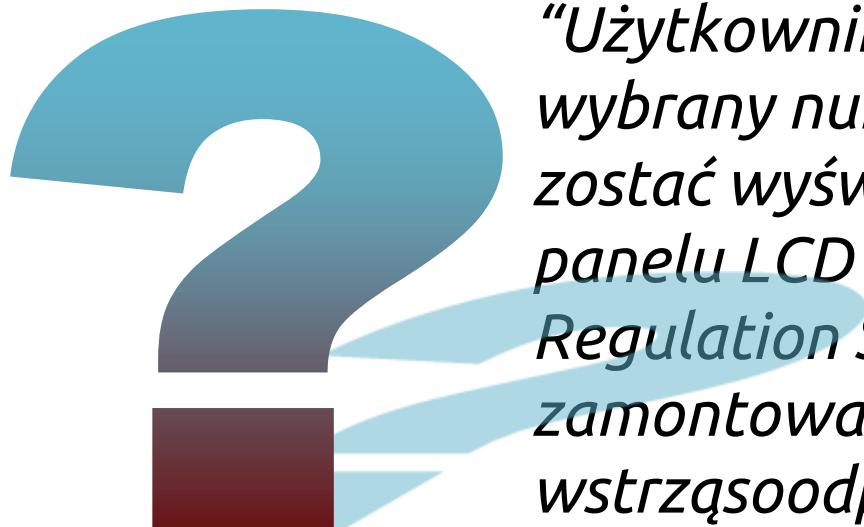


“Zakładając, że wskazane sygnały wejściowe z określonych urządzeń zostały przekazane użytkownikowi w poprawnej kolejności, gdzie system potrafi rozróżnić desygnatora, sygnał wyjściowy powinien być zgodny z wymaganym schematem z części 3.1.5 w celu wskazania pożądanego stanu wejściowego.”

Unikanie pułapek – pomieszanie

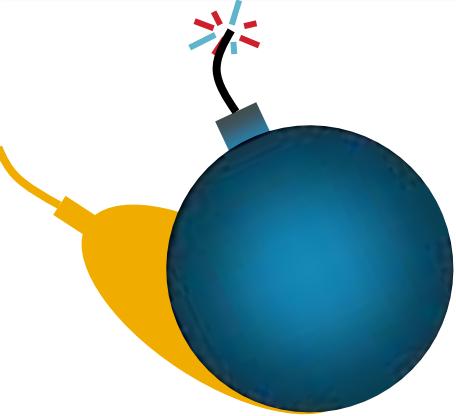


- ✖ mieszanie użytkownika, systemu, projektu, testu i instalacji
- ✖ wysoki poziom pomieszany z projektem bazy danych, terminologią oprogramowania, szczególnie technicznymi

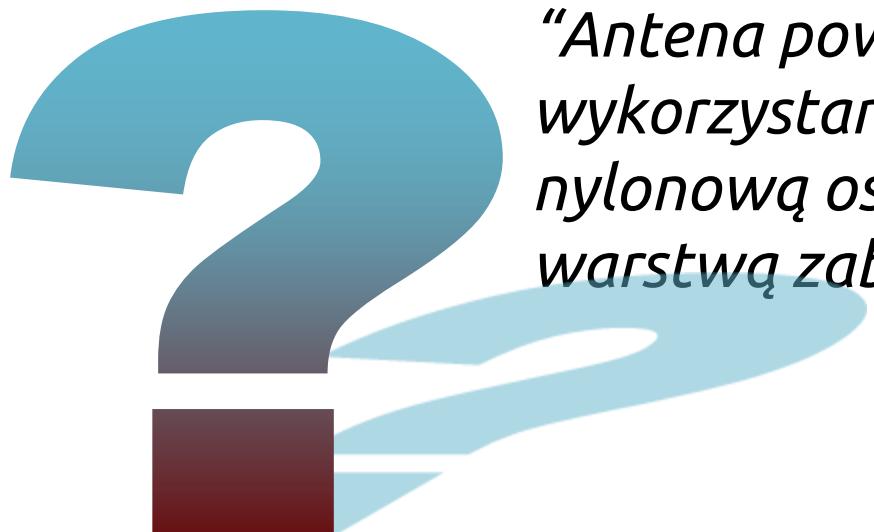


“Użytkownik może zobaczyć aktualnie wybrany numer kanału, który powinien zostać wyświetlony czcionką 14pt Swiss na panelu LCD zgodnym ze standardem Federal Regulation Standard 567-89 i zamontowanym z użyciem wstrząsoodpornych gumowych uszczelek.”

Unikanie pułapek – projektowanie

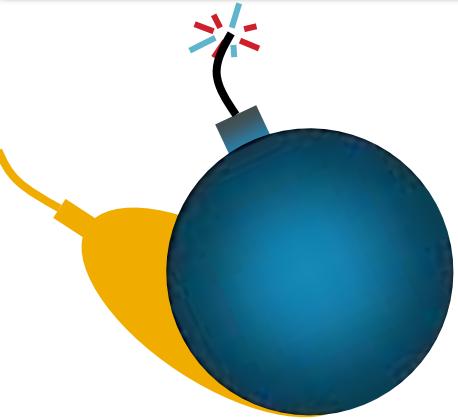


- ✓ specyfikacja ram projektowych dla odpowiedniego poziomu
- ✗ nazywanie komponentów, materiałów, obiektów programowych, pól, rekordów w wymaganiach użytkownika czy systemowych



“Antena powinna pobierać sygnał FM przy wykorzystaniu miedzianego rdzenia z nylonową osłoną i wodoszczelną gumową warstwą zabezpieczającą”

Unikanie pułapek – spekulowanie

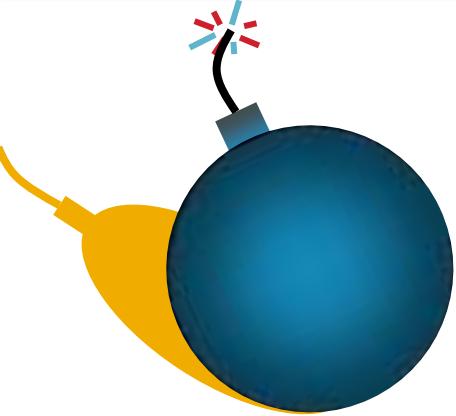


- ✖ listy życzeń
- ✖ niejednoznaczne określenie udziałowca, którego dotyczą wymagania
- ✖ ...zwykle... , ...w zasadzie... , ...często... ,
...normalnie... , ...ogólnie...



“System alarmowy powinien najprawdopodobniej działać z wykorzystaniem linii telefonicznej.”

Unikanie pułapek – brak precyzji

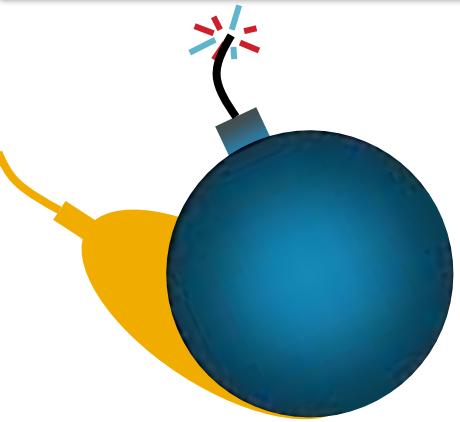


- ✗ terminy jakościowe
- ✗ przyjazny dla użytkownika, wysoce wszechstronny, elastyczny
- ✗ na maksymalnym poziomie, około, tyle ile to możliwe, z minimalnym wpływem...



“Użytkownik może używać przyjaznego interfejsu.”

Unikanie pułapek – sugerowanie

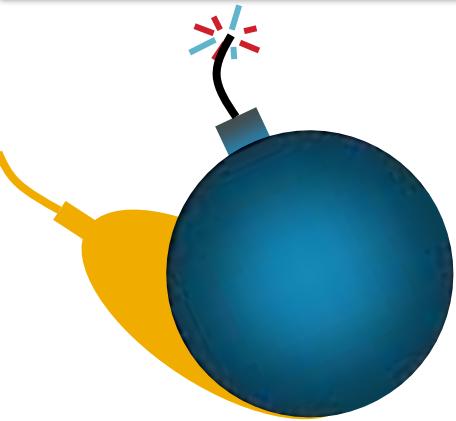


- ✓ sugestie będą zignorowane przez deweloperów
- ✗ może, można, powinno, być może, zapewne, prawdopodobnie

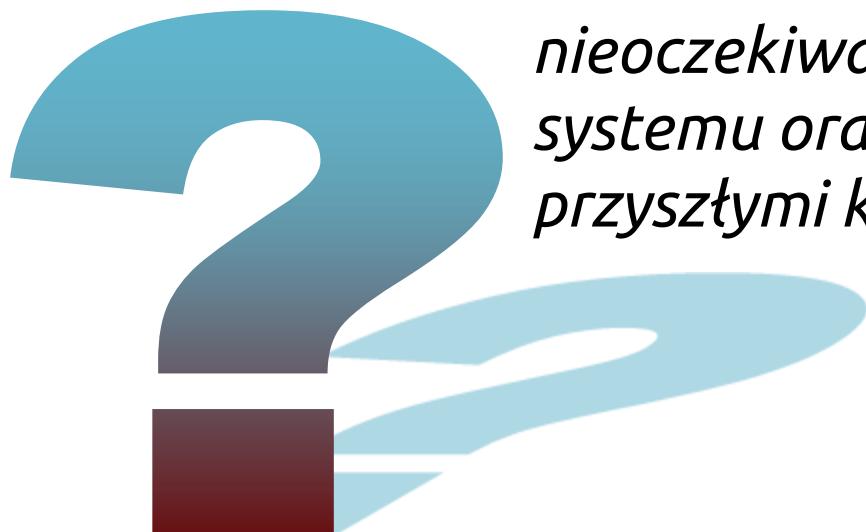
“Kierownik magazynu może wyświetlić listę towarów z możliwym niskim stanem magazynowym i powinien niezwłocznie dokonać zamówienia tych towarów.”



Unikanie pułapek – życzenia



- ✗ prośba o niemożliwe
- ✗ 100% niezawodność, bezpieczeństwo, obsługa wszystkich awarii, w pełni aktualizowalny, działający na wszystkich platformach



“Menedżer sieci powinien obsłużyć wszystkie nieoczekiwane błędy bez zawieszania systemu oraz być w stanie w pełni zarządzać przyszłymi konfiguracjami sieci.”

Podsumowanie

podstawowe koncepcje
inżynierii wymagań

*etap specyfikacji jest najważniejszy
w całym projekcie!*

Pytania



Źródła

- Sacha K., Inżynieria oprogramowania, PWN 2010
- Górska J. (red.), Inżynieria oprogramowania w projekcie informatycznym, Mikom 2000
- Requirement Writing Techniques, IBM course QN220, Version 2.0, 2010

Następny wykład

Projektowanie architektury systemu (Modelowanie oprogramowania)

Inżynieria oprogramowania

Wykład 4: Projektowanie architektury systemu.

Modelowanie oprogramowania

Łukasz Radliński

Zachodniopomorski Uniwersytet
Technologiczny

lukasz.radlinski@zut.edu.pl

Agenda

- metody strukturalne
 - hierarchia funkcji
 - diagramy przepływu danych
 - diagram związków encji
- metody obiektowe – UML
 - podstawowe
 - diagram przypadków użycia
 - diagram klas
 - procesy biznesowe
 - diagram czynności
 - dziedzina
 - diagram maszyny stanowej
 - struktura
 - diagram pakietów
 - diagram komponentów
 - diagram rozmieszczenia
 - współdziałanie
 - diagram sekwencji
 - diagram komunikacji
- SysML

Metody strukturalne

Hierarchia funkcji

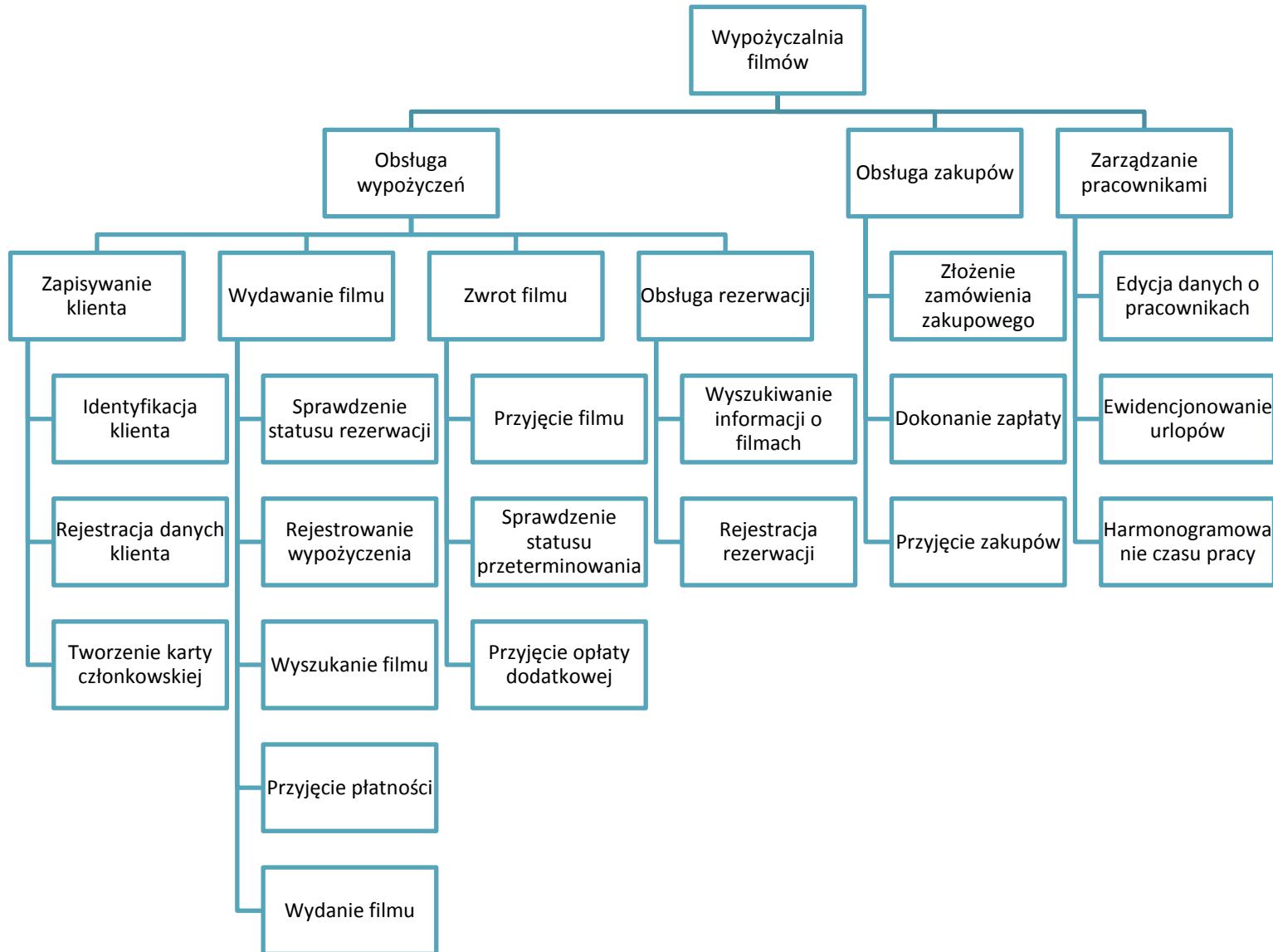


Diagram przepływu danych

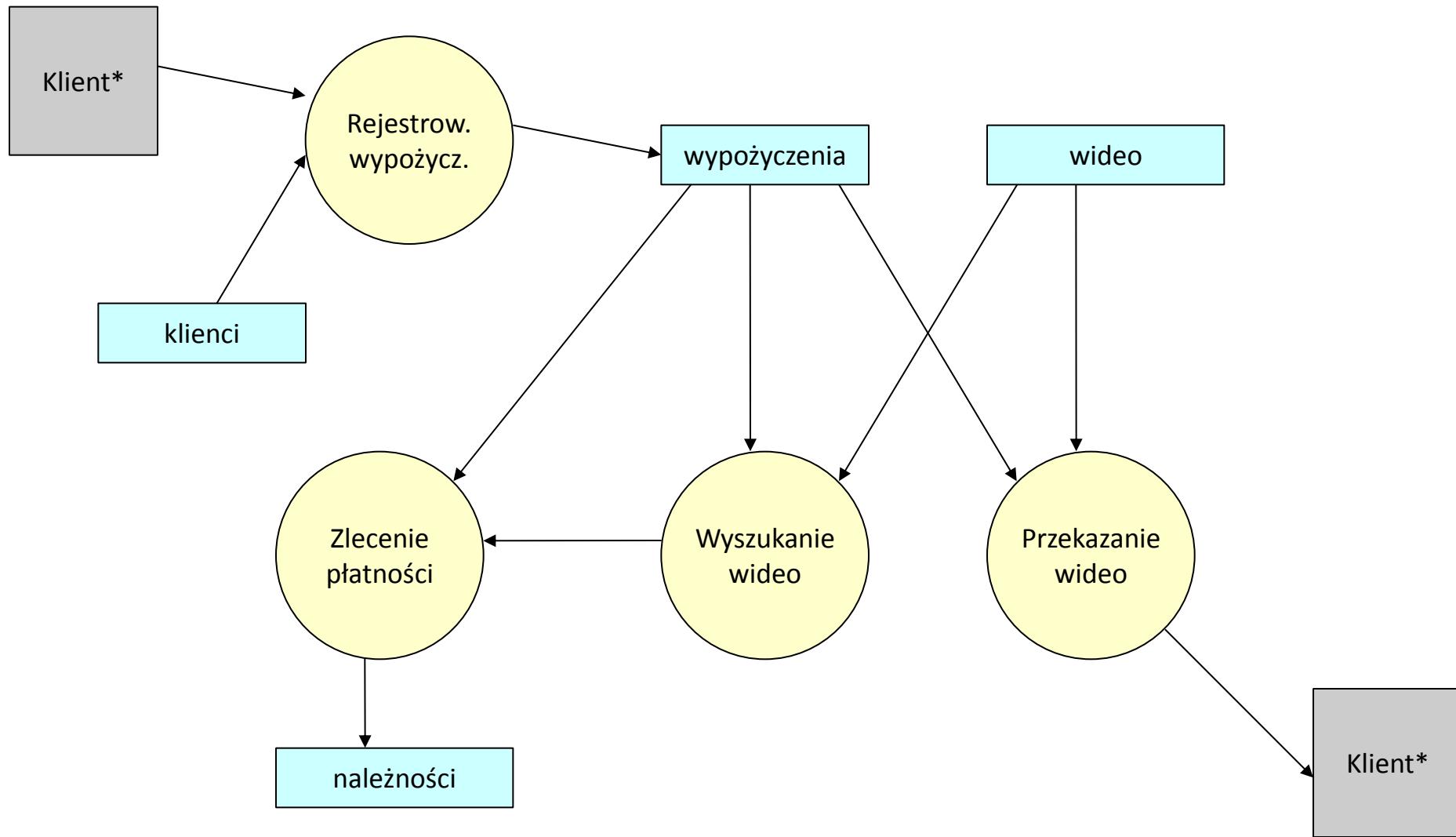


Diagram związków encji

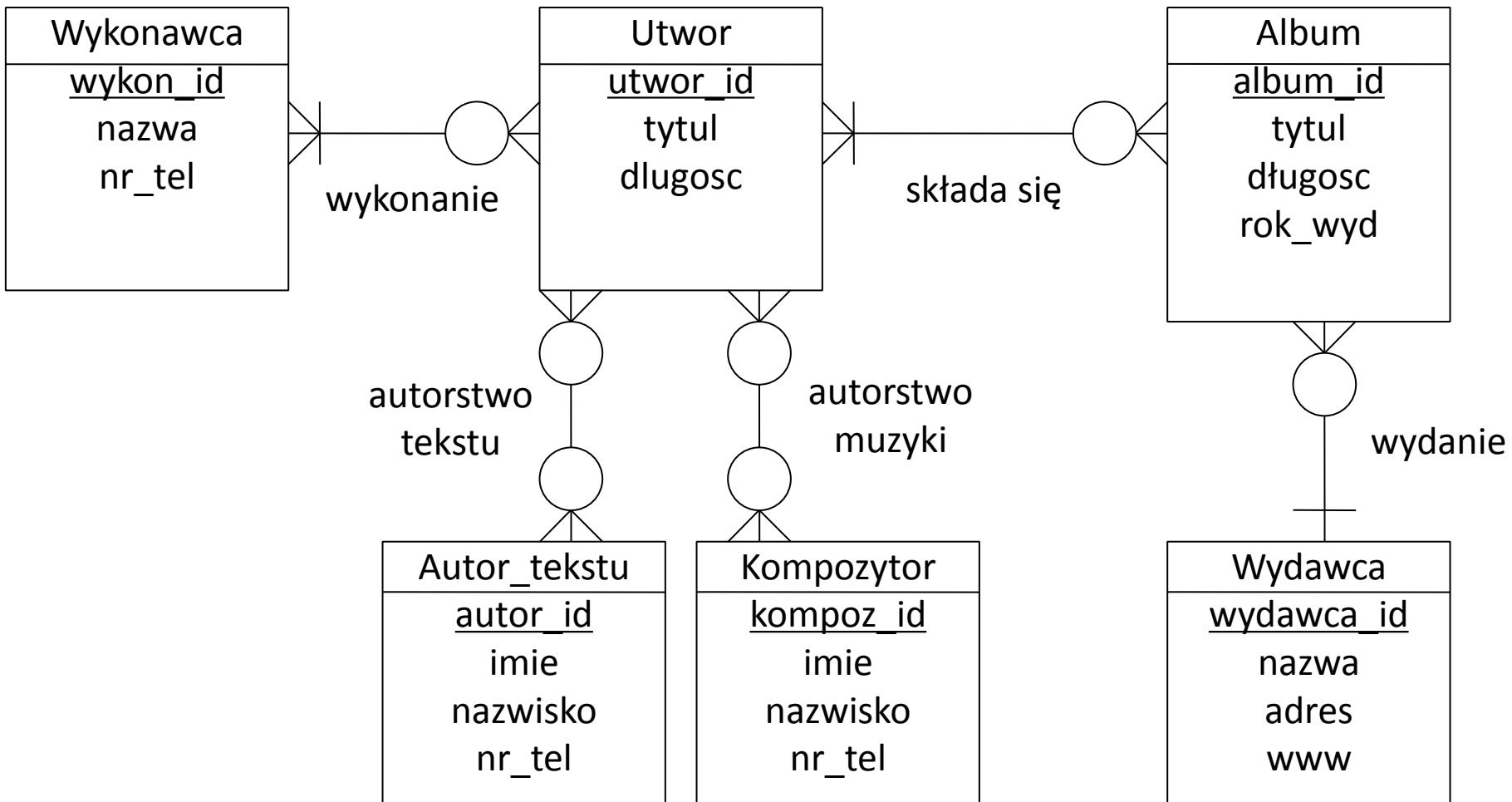
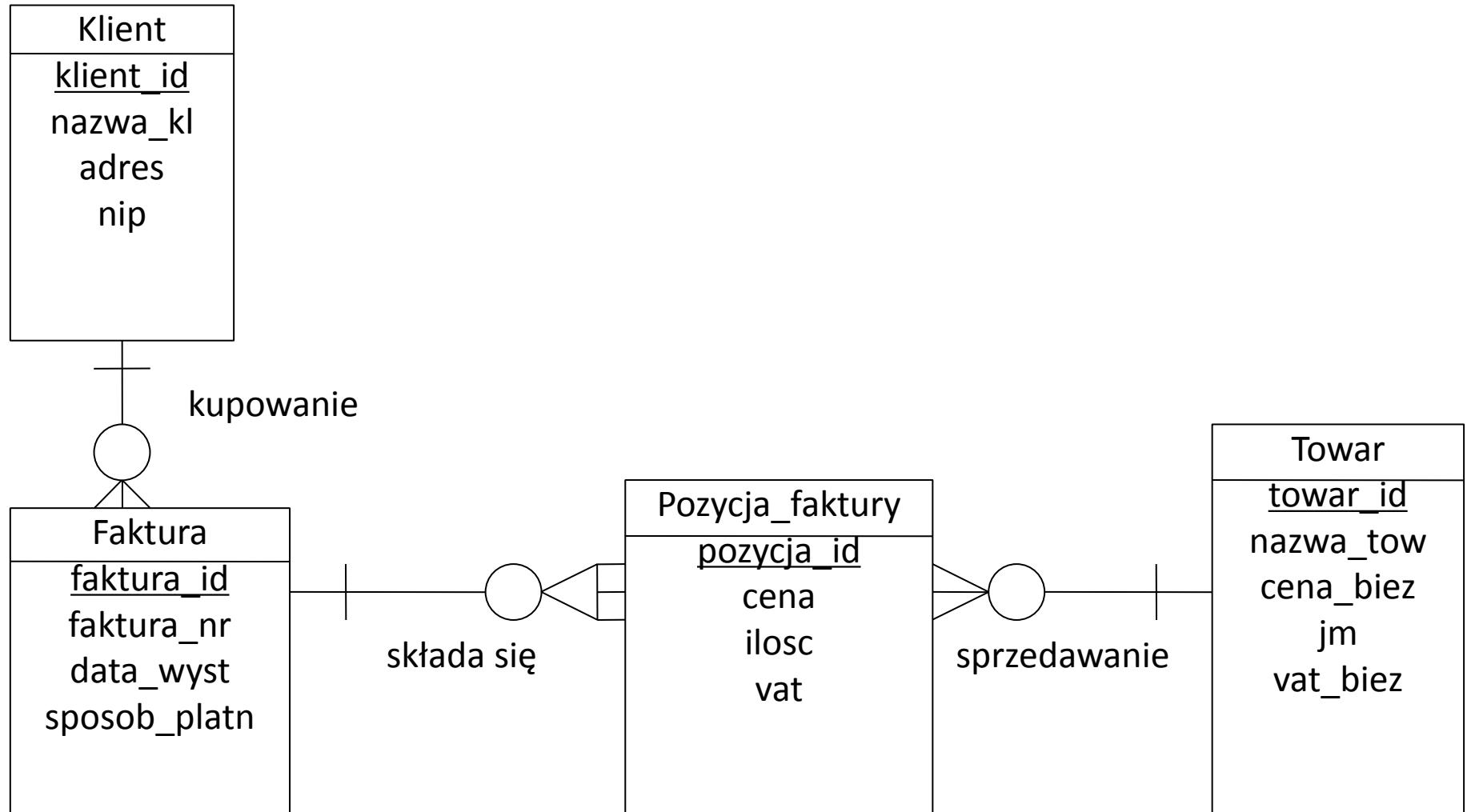


Diagram związków encji



Metody obiektowe – UML

Modele obiektowe

Diagram	Ang.	Przeznaczenie
przypadków użycia	use case	kategorie użytkowników, sposoby używania systemu
klas	class	klasy obiektów i interakcje
czynności (aktywności)	activity	procesy biznesowe, scenariusze przypadków użycia, algorytmy
maszyny stanowej	state machine	historia życia obiektu – stany i przejścia między nimi
komponentów	component	fizyczne składniki oprogramowania, zależności i interfejsy
pakietów	package	grupowanie składników w pakiety, zależności między pakietami
rozmieszczenie (wdrożenia)	deployment	konfiguracja sprzętowa i programowa komponentów systemu
sekwencji (przebiegu)	sequence	czasowa sekwencja wymiany komunikatów między obiekta mi, pakietami, komponentami
komunikacji	communication	przepływ komunikatów między obiekta mi, pakietami, komponentami
struktury złożonej	composite structure	wewnętrzna struktura złożonej klasy, komponentu, przypadku użycia
przeglądu interakcji	interaction overview	przepływ sterowania w procesie biznesowym lub systemie
obiektów	object	chwilowa konfiguracja obiektów oprogramowania
czasowy	timing	uzależnienia czasowe
profili	profile	poziom metamodelu – profile stereotypów, metaklasy

diagram przypadków użycia
diagram klas

Podstawowe diagramy

Diagram przypadków użycia

- reprezentacja systemu lub podsystemu, pokazująca funkcje realizowane przez system z punktu widzenia użytkownika
- ang.: *Use Case Diagram*
- ~odpowiednik diagramu przepływu danych w modelowaniu strukturalnym
- nie pokazuje technologii przetwarzania danych

Diagram przypadków użycia

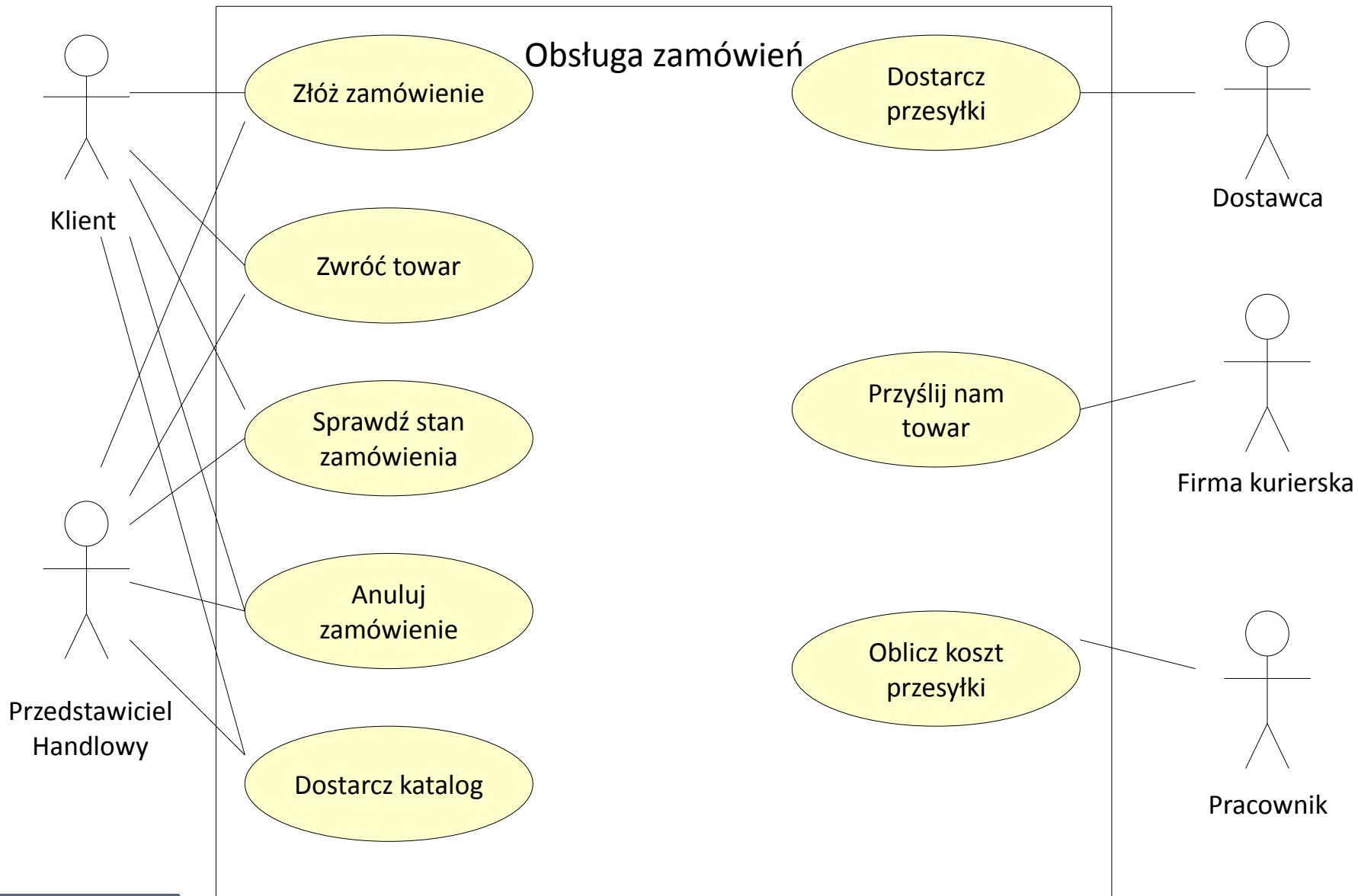
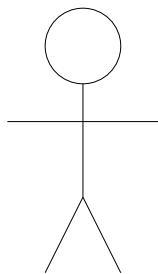
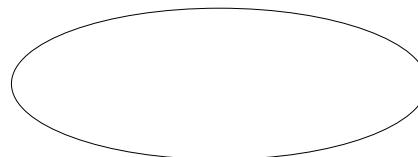


Diagram przypadków użycia

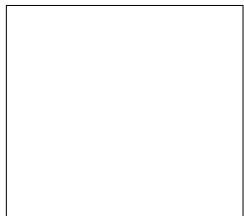
Aktor



Przypadek użycia



System



Powiązanie

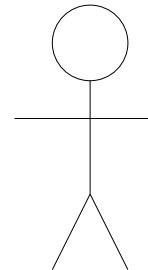


Diagram przypadków użycia aktor

- dowolny byt będący w interakcji z systemem
 - np. człowiek, inny program, sprzęt elektroniczny, składnica danych, sieć komputerowa, firma, jednostka organizacyjna
- leży **poza systemem**
- aktor pełni określoną rolę w systemie:
 - osoba może być reprezentowana przez wielu aktorów (wiele ról)
 - wiele osób może być reprezentowanych przez jednego aktora (jedna rola)
- ~ odpowiednik terminatora (encji zewnętrznej) na diagramie przepływu danych

Diagram przypadków użycia aktor

- symbole graficzne:



- konwencje nazewnicze:
– rzeczownik (może być z innymi wyrazami)
 - np.: Klient, Kierownik, Firma Kurierska, Bank, Dział Sprzedaży, System Magazynowy

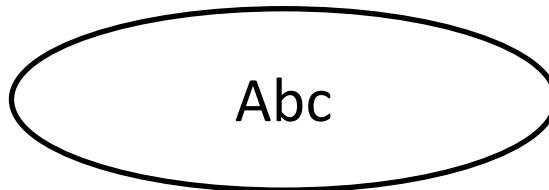
Diagram przypadków użycia

przypadek użycia

- **zespoł czynności** realizowanych przez system w celu wytworzenia określonego wyniku, mającego znaczenie dla aktora
- jest **całością** zadania widzianego z punktu widzenia aktora
- zwykle obejmuje działania niezbyt odległe w czasie
- zwykle obejmują działania obsługiwane przez jednego aktora
- zwykle jest inicjowany przez aktora
- ~ Odpowiednik procesu na diagramie przepływu danych

Diagram przypadków użycia przypadek użycia

- symbole graficzne:



- konwencje nazewnicze:
 - czasownik + dopełnienie:
 - Złoż zamówienie, Odbierz towar, Oblicz wartość podatku, Wyślij zawiadomienie

Diagram przypadków użycia

przypadek użycia

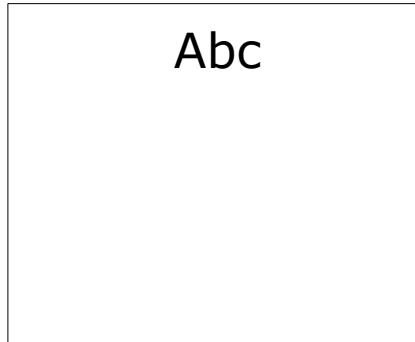
- jakich funkcji oczekuje aktor od systemu?
- którzy aktorzy tworzą nowe dane, modyfikują lub usuwają istniejące?
- którzy aktorzy czytają zapamiętane dane?
- jak system zawiadamia aktora o zmianach w swoim wewnętrznym stanie?
- jak system będzie powiadamiany o zewnętrznych zdarzeniach?

Diagram przypadków użycia system

- pokazuje granice systemu (lub podsystemu)
 - obejmując przypadki użycia
 - pozostawiając aktorów na zewnątrz
- jeden diagram – jeden system

Diagram przypadków użycia system

- symbole graficzne



- konwencje nazewnicze
 - rzeczownik odczasownikowy + dopełnienie:
 - Przyjmowanie zamówienia, Realizacja przewozu, Zarządzanie zadaniami, Obsługa płatności
 - rzeczownik
 - Zamówienia, Płatności

Diagram przypadków użycia powiązanie

- łączy aktora z przypadkiem użycia
- pokazuje, jaki aktor bierze udział przy realizacji danego przypadku użycia
- nie ma nazwy (podpisu)
- symbole graficzne



Diagram przypadków użycia

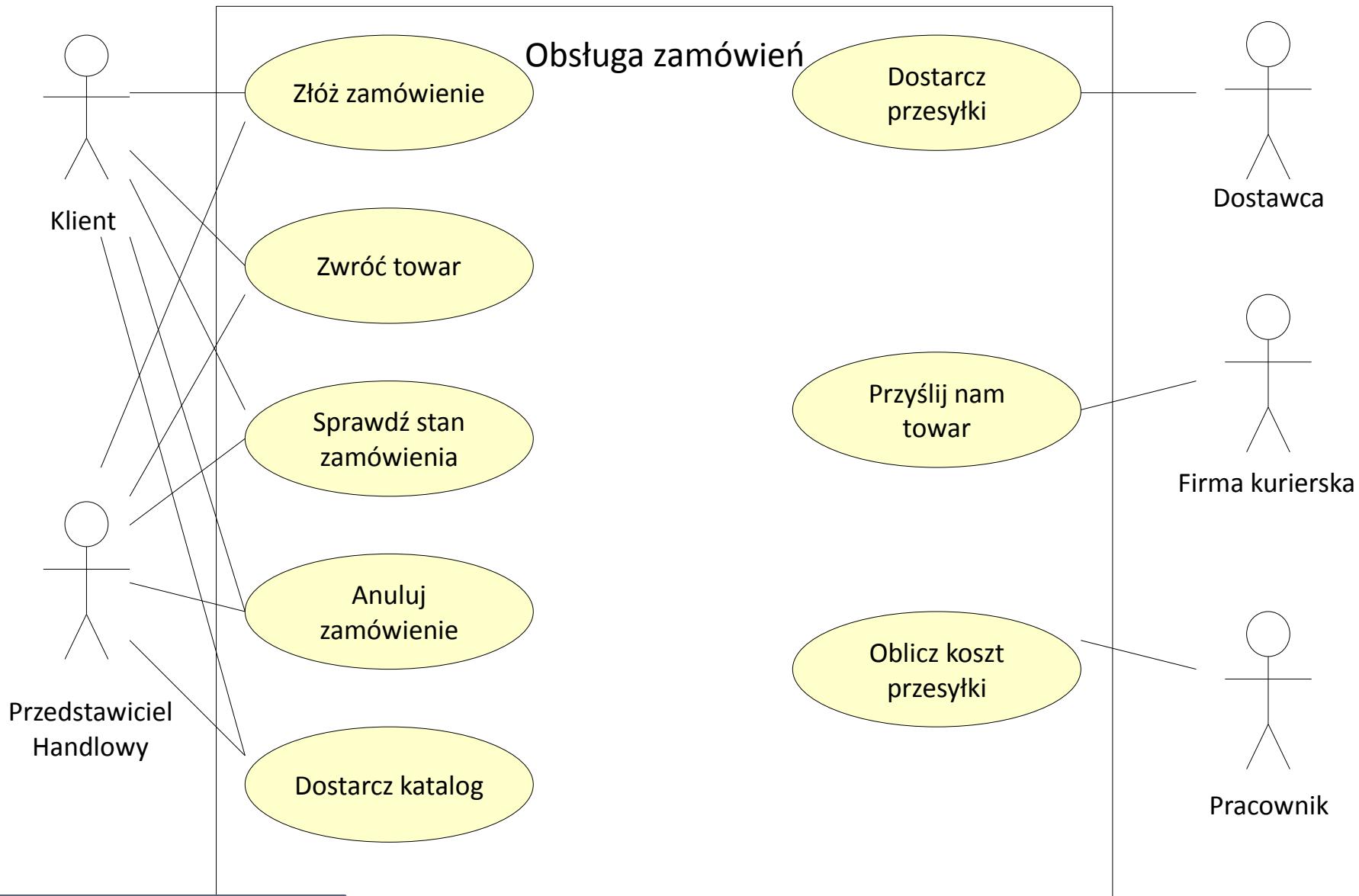


Diagram przypadków użycia

dokumentowanie przypadków użycia

- warunki wstępne
 - jakie warunki muszą być spełnione, aby przypadek użycia mógł być zrealizowany
- przebieg zdarzeń
 - lista działań charakteryzujących przypadek użycia
- warunki końcowe
 - co jest efektem realizacji przypadku użycia
- ~ odpowiednik specyfikacji procesu z diagramu przepływu danych

Diagram przypadków użycia

dokumentowanie przypadków użycia

przyp. użycia nr XX: Utworzenie sprawy likwidacji szkody

scenariusz główny

1. System wyświetla formularz utworzenia sprawy
2. Przyjmujący wprowadza opis sprawy (numer polisy, dane zgłaszającego, datę wypadku i datę zgłoszenia)
3. Przyjmujący zatwierdza opis sprawy
4. System sprawdza poprawność zgłoszenia
5. System tworzy sprawę i nadaje jej unikalny numer

scenariusz alternatywny 1 – duplikat zgłoszenia

- 1-4. Jak w scenariuszu głównym
5. System powiadamia o istniejącym zgłoszeniu i odmawia utworzenia sprawy

scenariusz alternatywny 2 – nieważna polisa

- 1-4. Jak w scenariuszu głównym
5. Likwidator powiadamia klienta o odmowie odszkodowania i zamyka sprawę

Diagram przypadków użycia bardzo rozbudowany

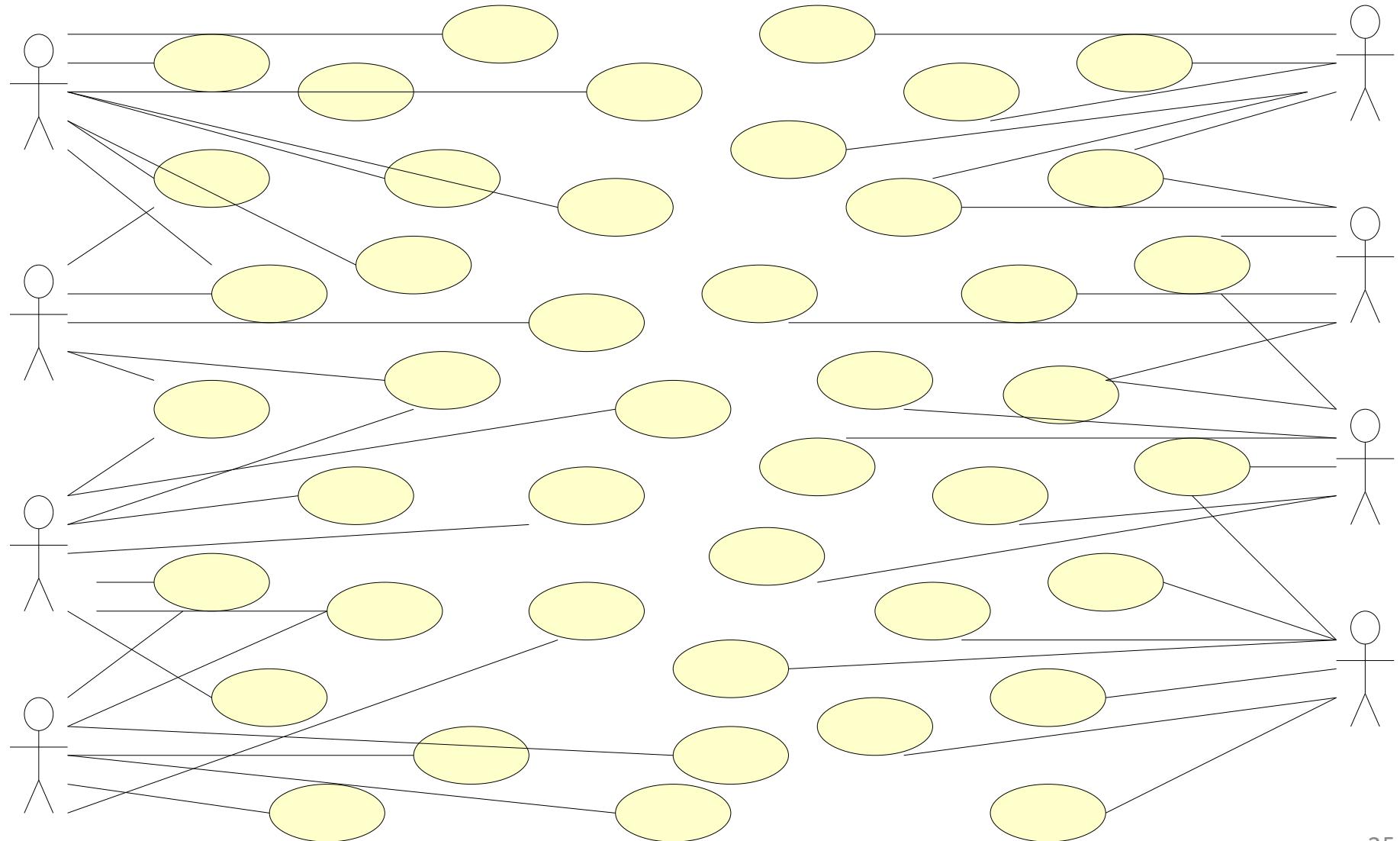


Diagram przypadków użycia

- rozwiązanie
 - ograniczenie liczby przypadków użycia na jednym diagramie
 - przypadki użycia ujęte w pakiety (podsystemy)
 - model wielopoziomowy – hierarchiczny
 - podobnie do diagramów przepływu danych

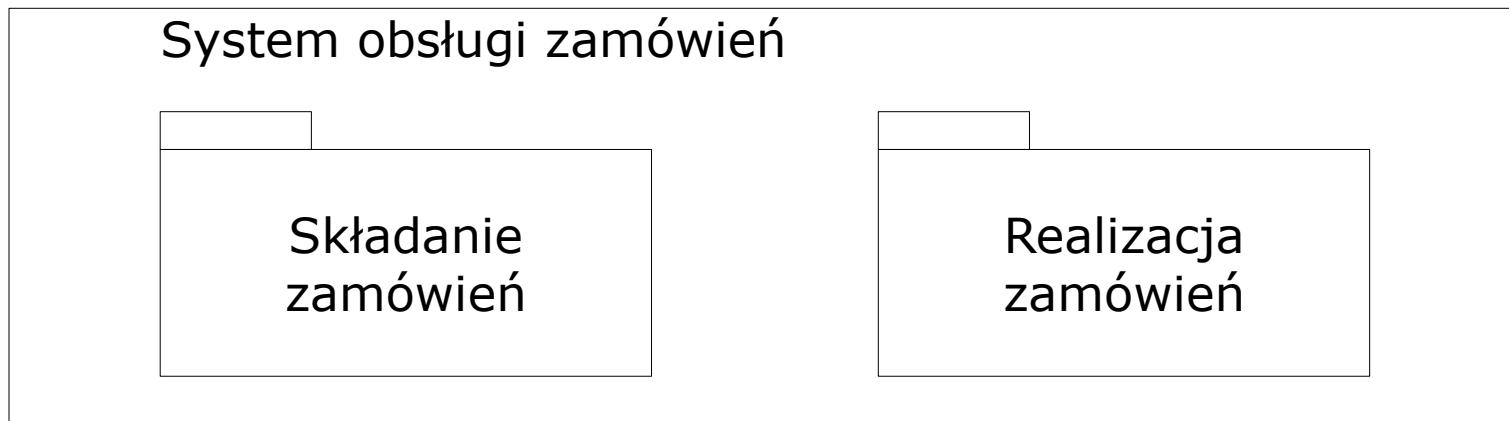


Diagram przypadków użycia

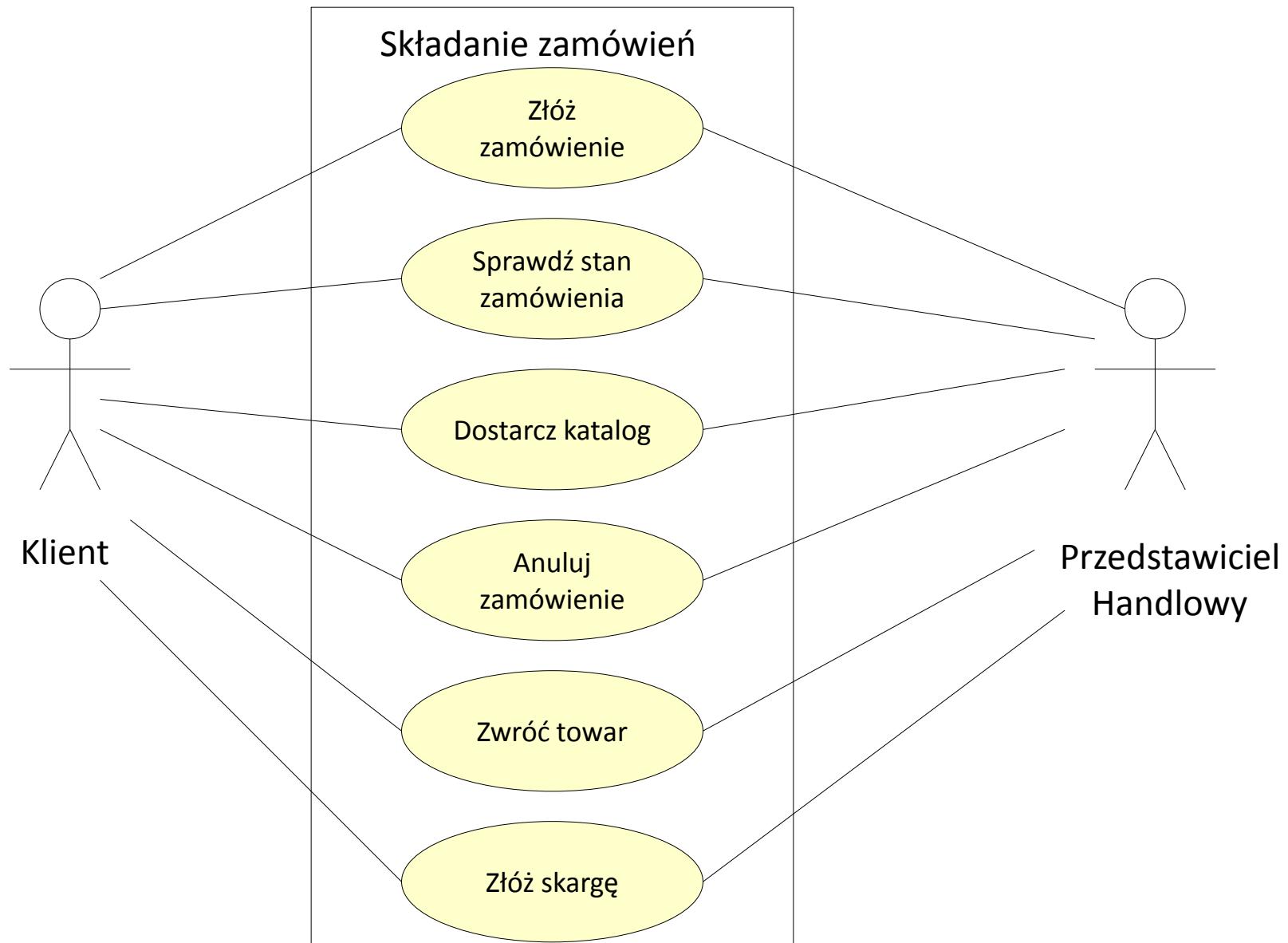


Diagram przypadków użycia

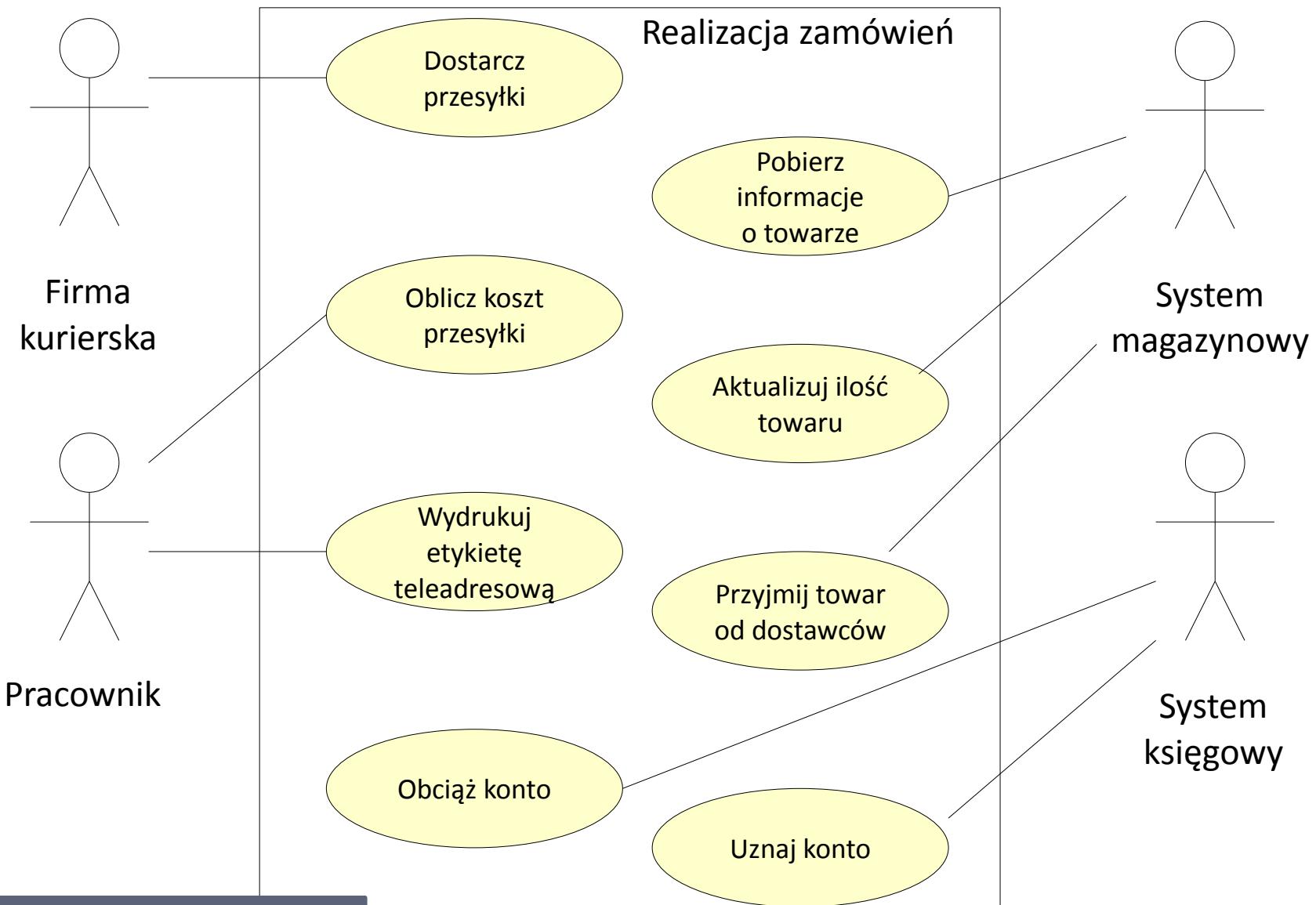
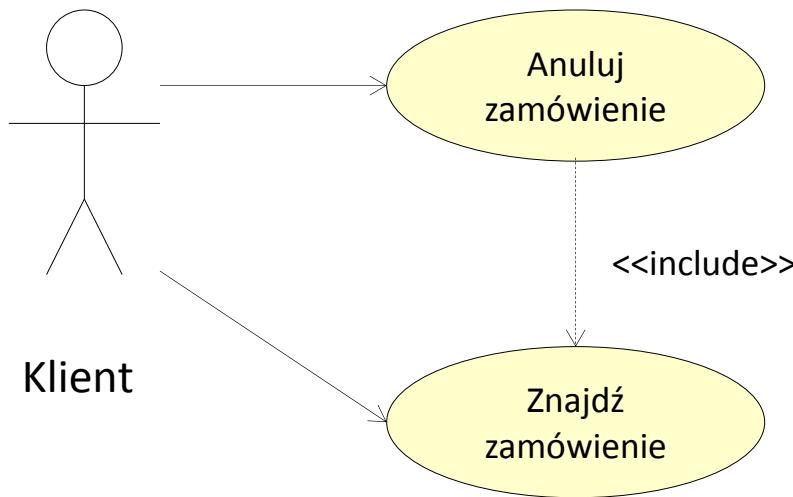


Diagram przypadków użycia dodatkowe symbole

- związek zawierania
- związek rozszerzania
- dziedziczenie

Diagram przypadków użycia związek zawierania

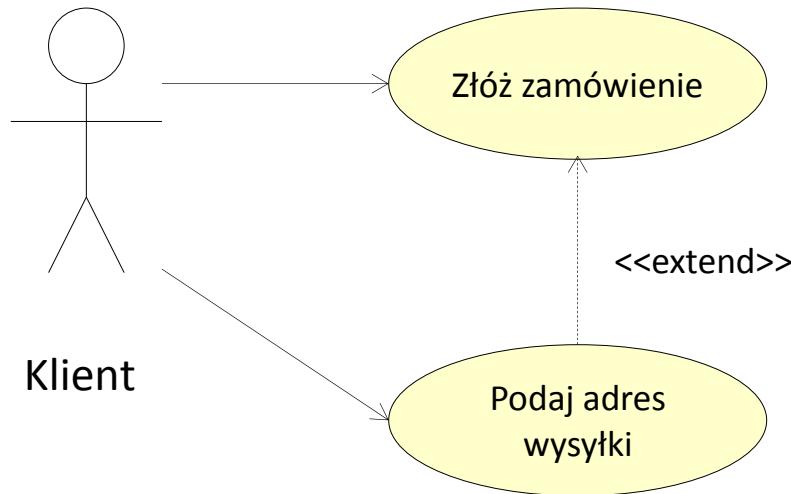
- pokazuje, że jeden przypadek użycia zawiera drugi



- “Anuluj zamówienie” zawiera “Znajdź zamówienie”
- “Znajdź zamówienie” jest częścią “Anuluj zamówienie”

Diagram przypadków użycia związek rozszerzania

- pokazuje, że jeden przypadek użycia w niektórych sytuacjach może być rozszerzony przez inny



- “Podaj adres wys.” rozszerza “Złóż zamówienie” –
 - czasami “Złóż zamówienie” wymaga również realizacji “Podaj adres wysyłki”

Diagram przypadków użycia związek rozszerzania

- konieczne jest
 - zdefiniowanie warunku
 - którego spełnienie decyduje o rozszerzeniu przypadku użycia
 - miejsca rozszerzania

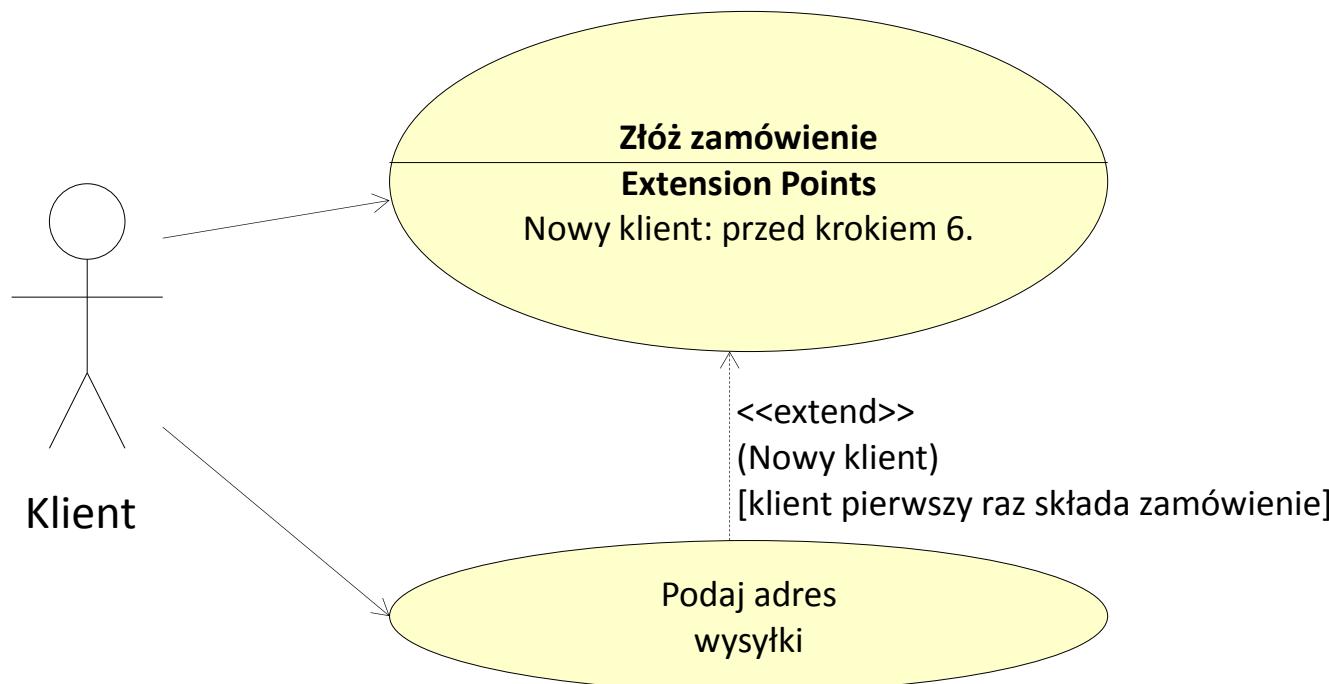


Diagram przypadków użycia

dziedziczenie

- to związek “jest rodzaju”
 - jeden element jest rodzajem drugiego elementu
- pomiędzy aktorami
 - jeden aktor odgrywa wszystkie role drugiego aktora, może dodatkowo odgrywać inne role
- pomiędzy przypadkami użycia
 - jeden przypadek użycia jest uszczegółowioną wersją drugiego
 - uszczegółowiony dziedziczy zachowanie ogólnego i może dodawać dodatkowe zachowania

Diagram przypadków użycia dziedziczenie

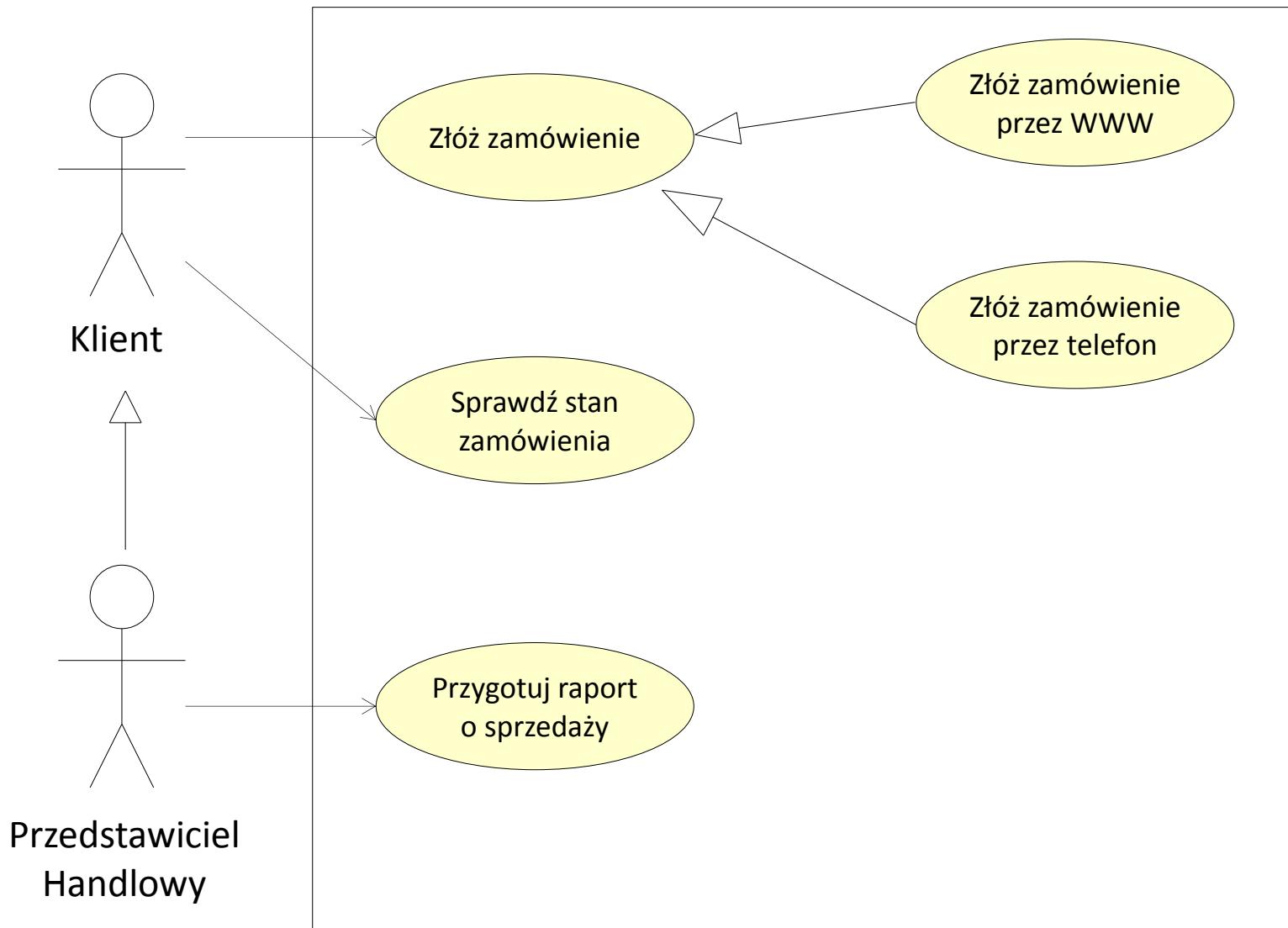


Diagram przypadków użycia przykład z dodatkowymi symbolami

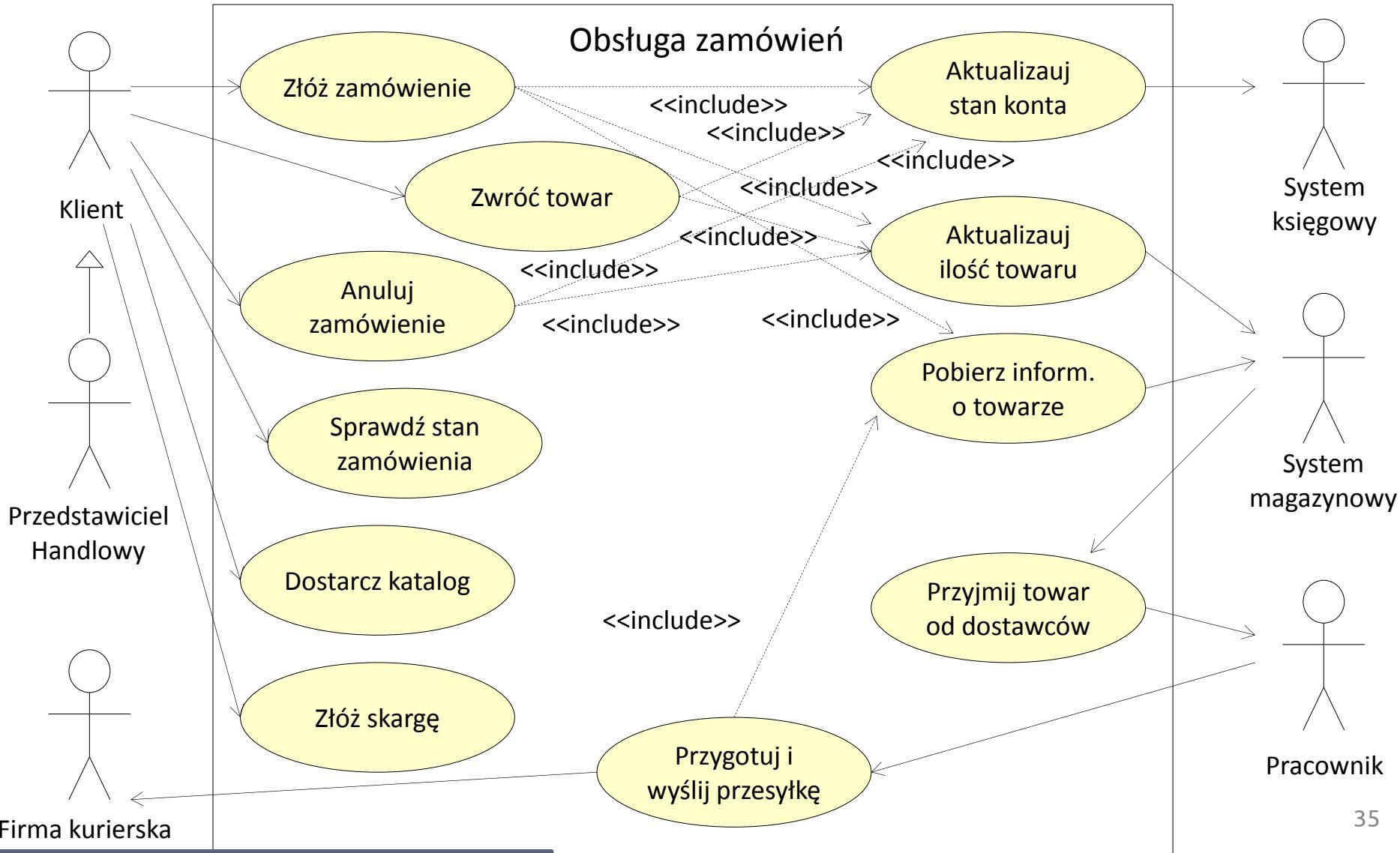


Diagram klas

- odwzorowuje świat rzeczywisty obejmując obiekty świata rzeczywistego i powiązania między nimi
- ang.: *Class Diagram*
- jest rozszerzoną formą diagramu związków encji z modelowania strukturalnego
- główne elementy diagramu
 - klasa (*class*)
 - związek (*association*)

Diagram klas

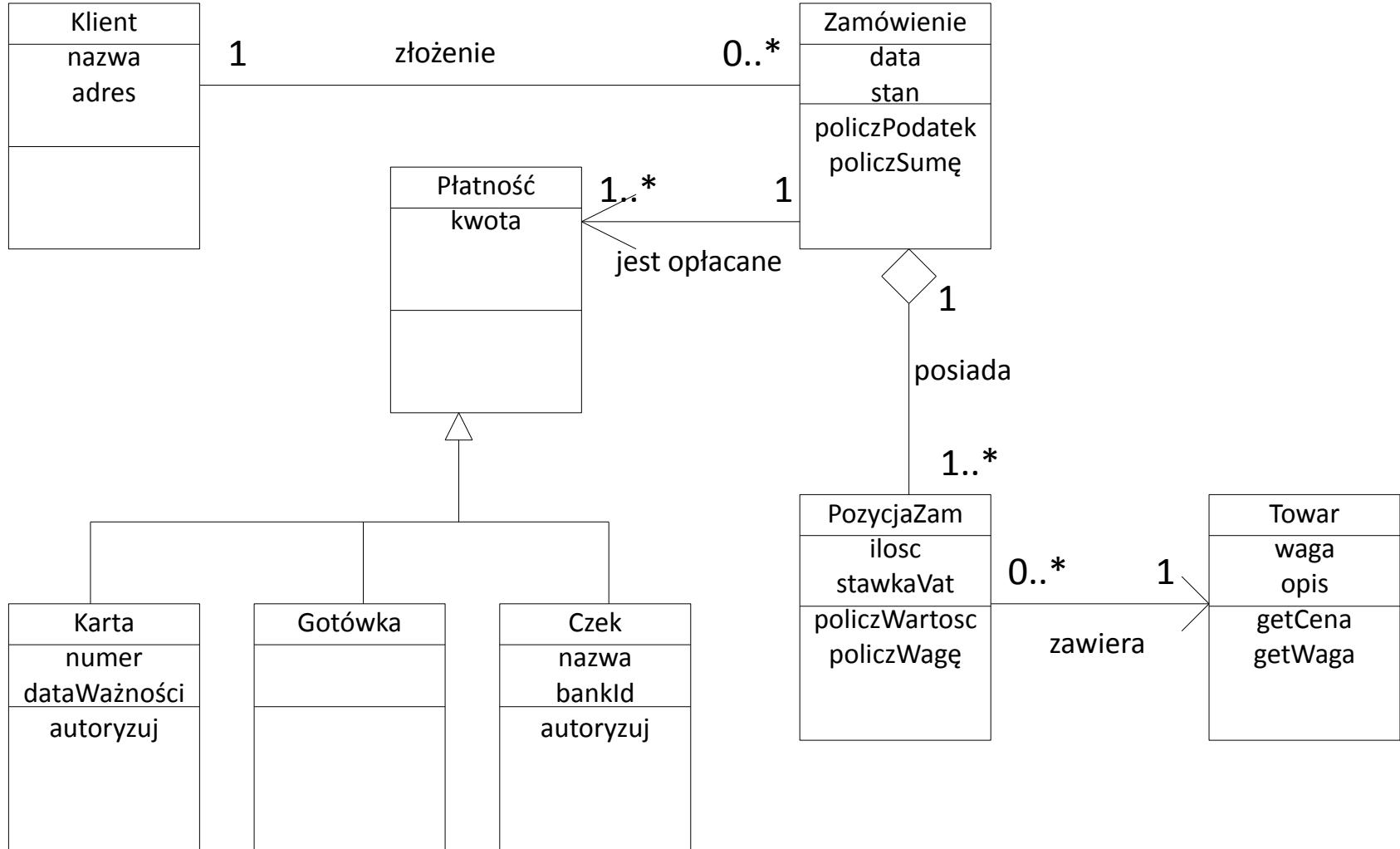


Diagram klas symbole

- klasa



- związek



Diagram klas

klasa

- jest reprezentacją obiektu świata rzeczywistego
- cechy klasy:
 - nazwa (unikalna)
 - zestaw atrybutów
 - zestaw metod
- może implementować działanie elementów innych diagramów:
 - aktora:
 - Samochód – Silnik, Karoseria, Hamulec
 - przypadku użycia:
 - Wyślij towar – Wysyłka, Towar, Opłata

Diagram klas klasa

- formy zapisu

Klient

Klient

+ klient_id
+ nazwa

+ zlozZamowienie
podajAdres

Klient

+ klient_id
+ nazwa

+ zlozZamowienie
podajAdres

Klasa implementuje
osoby składające
zamówienia

Diagram klas

klasa

- atrybuty – właściwości klasy
 - nazwa
 - typ danych
 - widoczność
 - + publiczny (*public*)
 - wszystkie obiekty w systemie
 - # chroniony (*protected*)
 - obiekty z danej klasy i klas potomnych (dziedziczących)
 - prywatny (*private*)
 - obiekty z danej klasy
 - ~ pakietowy (*package*)
 - instancje klasy z tego samego pakietu
 - zasięg
 - egzemplarzowy (*instance*) – odrębne wartości dla poszczególnych obiektów klasy
 - klasyfikatorowy (*classifier*) – jedna wartość wspólna dla wszystkich obiektów klasy

Diagram klas

klasa

- metody – operacje
 - które klasa może wykonać
 - które na danej klasie może wykonać inna klasa
- nazwa
 - unikalna w obrębie klasy i klas nadzędnych
- mogą używać parametrów (wej., wyj.)
- mogą zwracać wartość określonego typu
- widoczność
 - jak w przypadku atrybutów
- zasięg
 - jak w przypadku atrybutów

Diagram klas związek

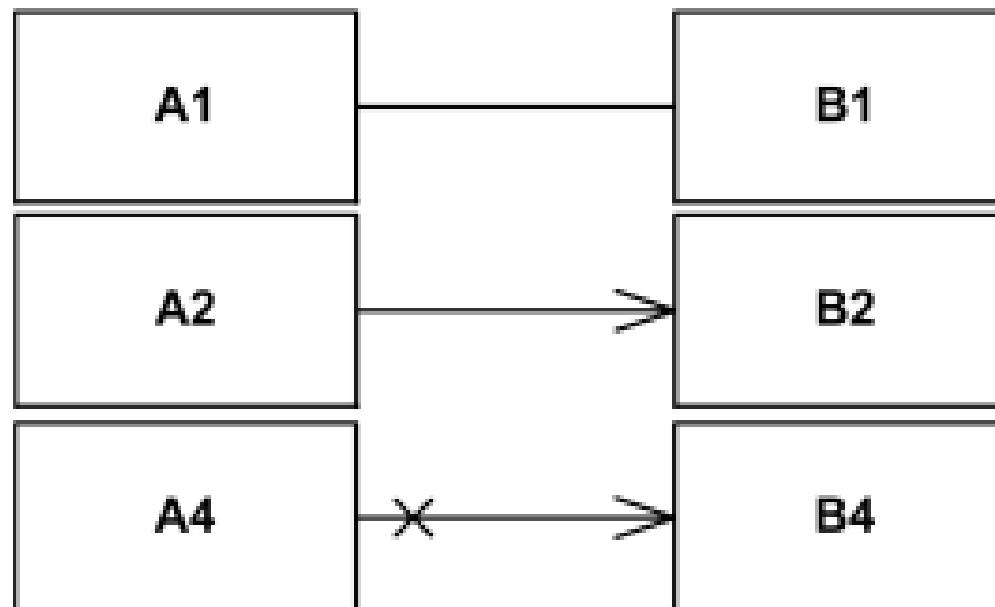
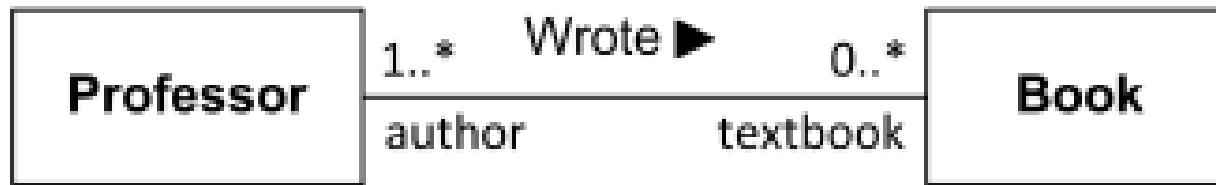
- ilustruje związek znaczeniowy między klasami
- cechy związku:
 - nazwa (unikalna?) - nieobowiązkowa
 - kierunek
 - liczebność/udział
 - brak atrybutów, ale możliwe klasy powiązane

Diagram klas związek

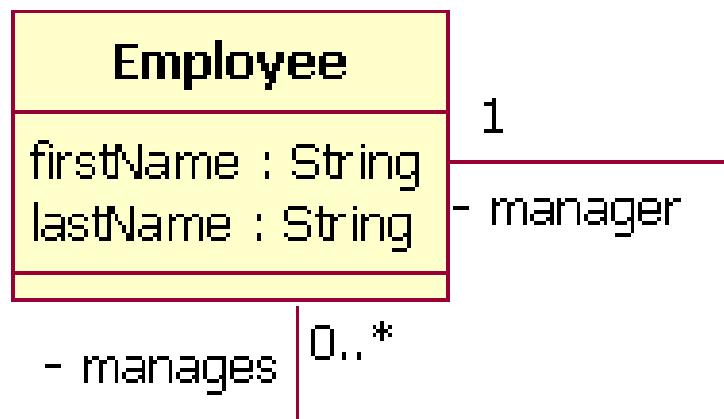
- liczebność

- 0..1 – zero lub jeden
- 1 – dokładnie jeden
- 0..* – zero lub więcej
- 1..* – jeden lub więcej
- n – dokładnie n ($n > 1$)
- * – wiele (niezalecane)
- 0..n – od zera do n ($n > 1$)
- 1..n – od jeden do n ($n > 1$)
- n..m – od n do m ($n > 1, m > 1$)
- n..* – n lub więcej ($n > 1$)

Kierunek nazwy związku a nawigowalność



Asocjacje zwrotne (rekurencyjne)



<http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/>

Diagram klas związek i klasa powiązana



Diagram klas dodatkowe symbole

- związek
- agregacja
- kompozycja
- dziedziczenie
- zależność
- realizacja

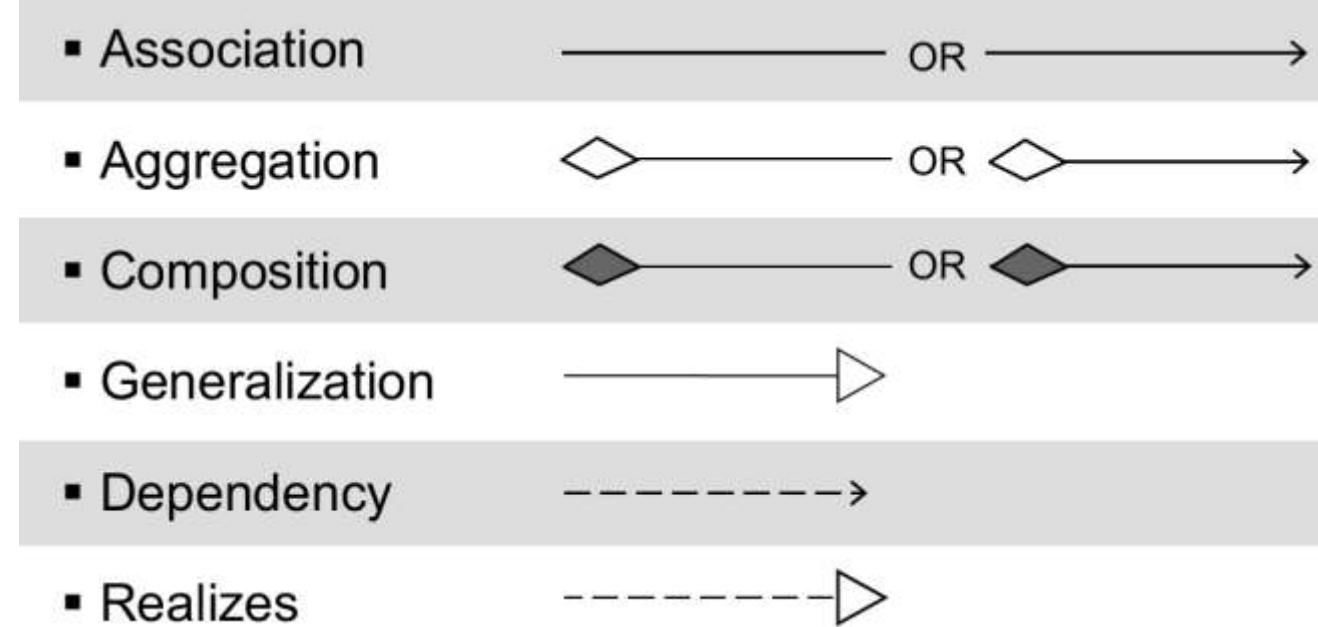


Diagram klas agregacja

- pokazuje, że dana klasa składa się z innych
 - obiekty jednej klasy zawierają obiekty innej klasy, np.:
 - drużyna – zawodnik
 - grupa – student



Diagram klas kompozycja

- silniejsza odmiana agregacji
 - jeden komponent może należeć tylko do jednej całości
 - czas życia komponentu jest zdeterminowany przez czas życia całości:
 - tylko całość może utworzyć nową część
 - jeśli całość zostanie zniszczona, niszczone są wszystkie jej części
- przykłady
 - biurko – blat
 - klawiatura – przycisk
 - projekt – etap – zadanie



Diagram klas agregacja i kompozycja

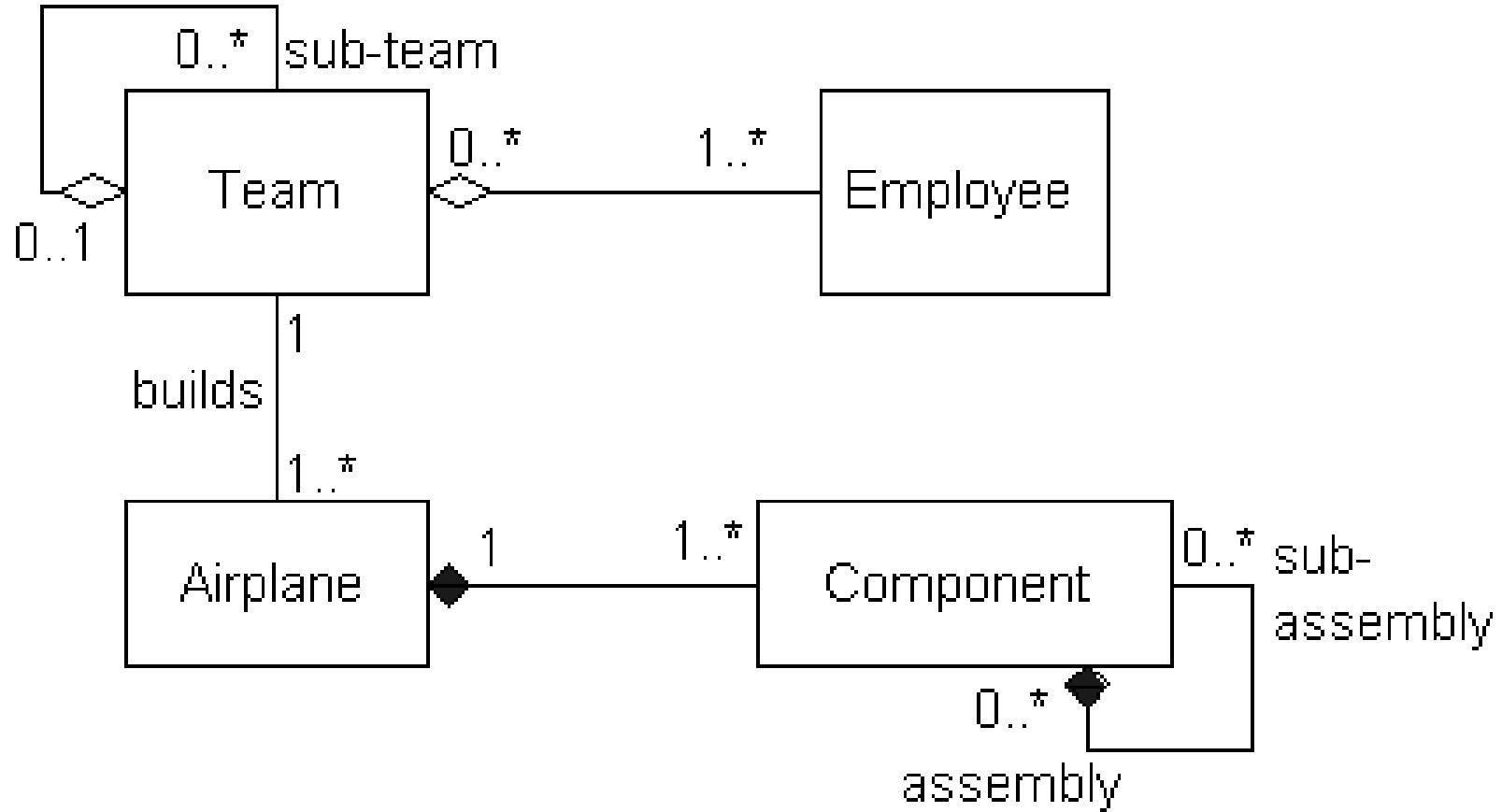


Diagram klas zależność

- pokazuje, że zachowanie jednej klasy wpływa na zachowanie innej
 - komunikat wysyłany z jednej klas do drugiej
 - jedna klasa jest częścią danych innej klasy
 - jedna klasa wykorzystuje inną jako parametr operacji
- nie pokazuje powiązania między danymi
 - na zasadzie relacji w bazie danych
- Symbol

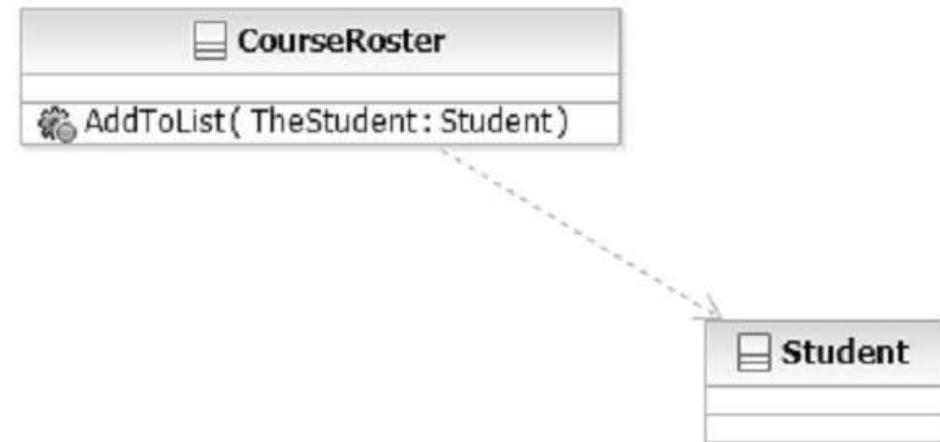


Diagram klas dziedziczenie

- klasa dziedzicząca zachowuje się podobnie do bazowej
 - przejmuje cechy klasy bazowej
 - posiada dodatkowe cechy (atrybuty i metody)
 - może łączyć się z innymi klasami
- przykłady
 - zwierzę – ssak – kot
 - osoba – student
 - dokument – faktura – faktura korygująca

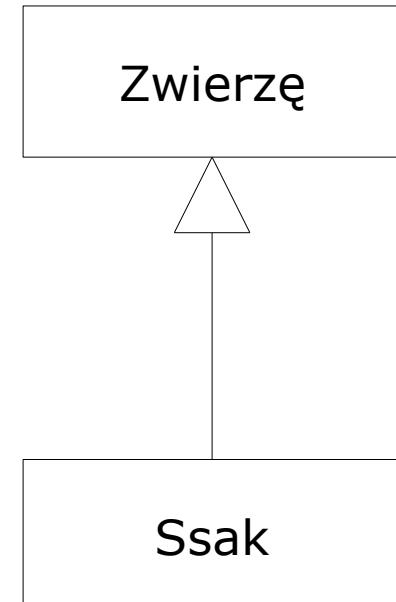


Diagram klas realizacja

- rodzaj powiązania, w którym klient implementuje dostarczoną specyfikację
 - implementacja interfejsu

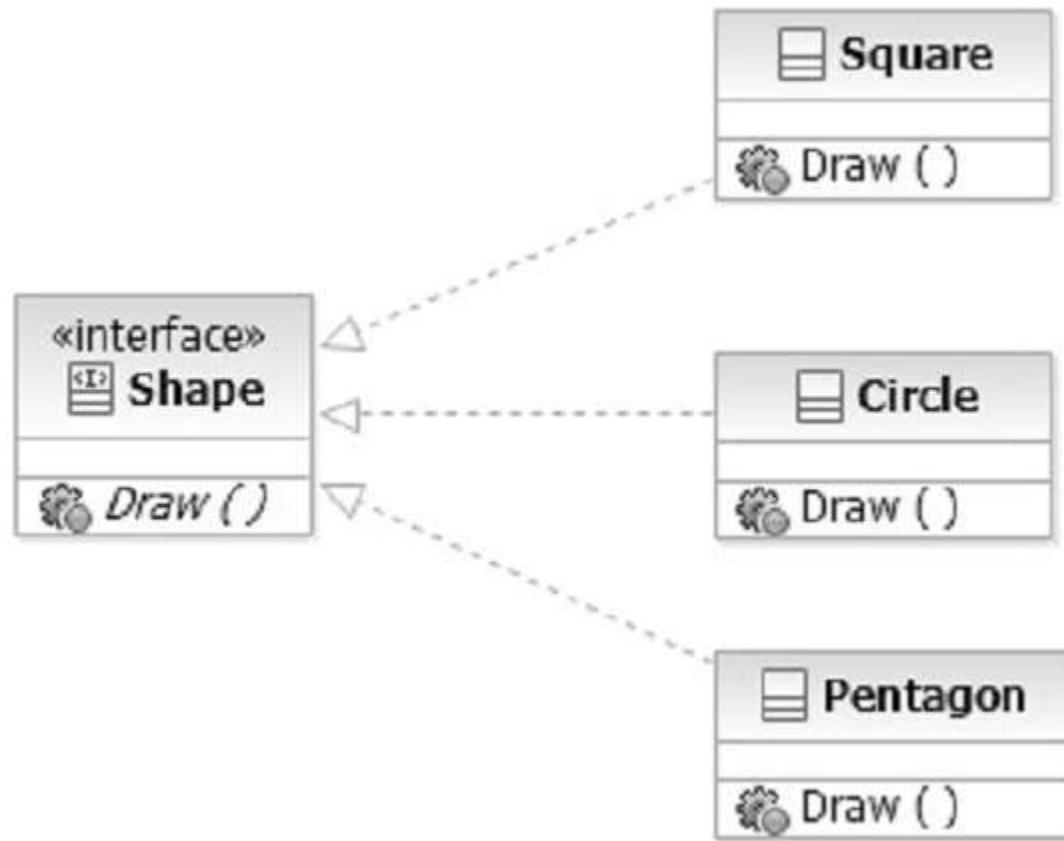


Diagram klas

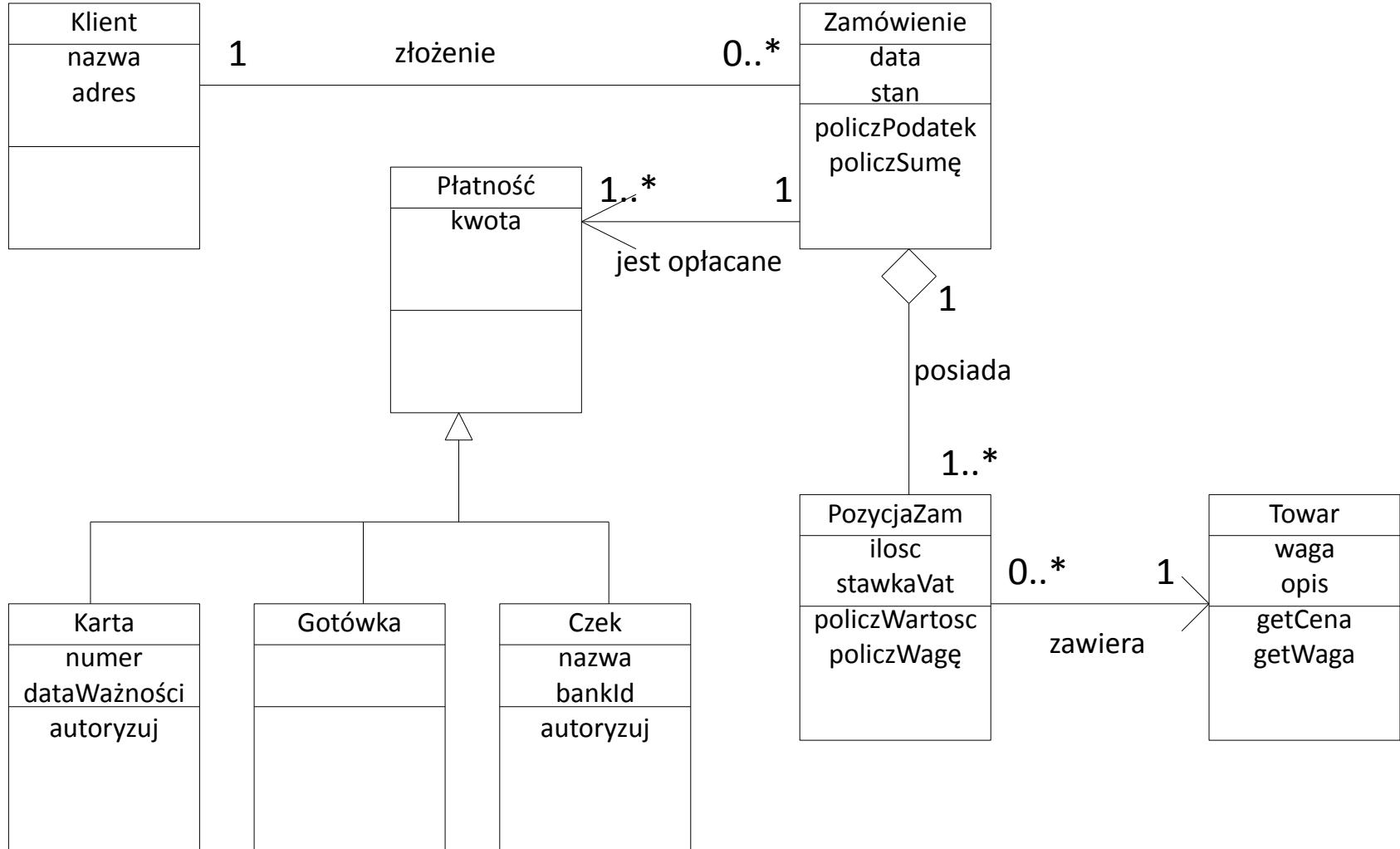


Diagram klas

- co modelujemy?
 - strukturę oprogramowania
 - strukturę danych
 - bazy danych

Diagram czynności

Modelowanie procesów biznesowych

Diagram czynności

- działania wykonywane podczas realizacji algorytmów
- ang.: *activity diagram*
- główne elementy
 - czynności
 - przejścia między czynnościami
 - punkty decyzyjne
 - rozwidlenia procesu – działania równoległe

Diagram czynności

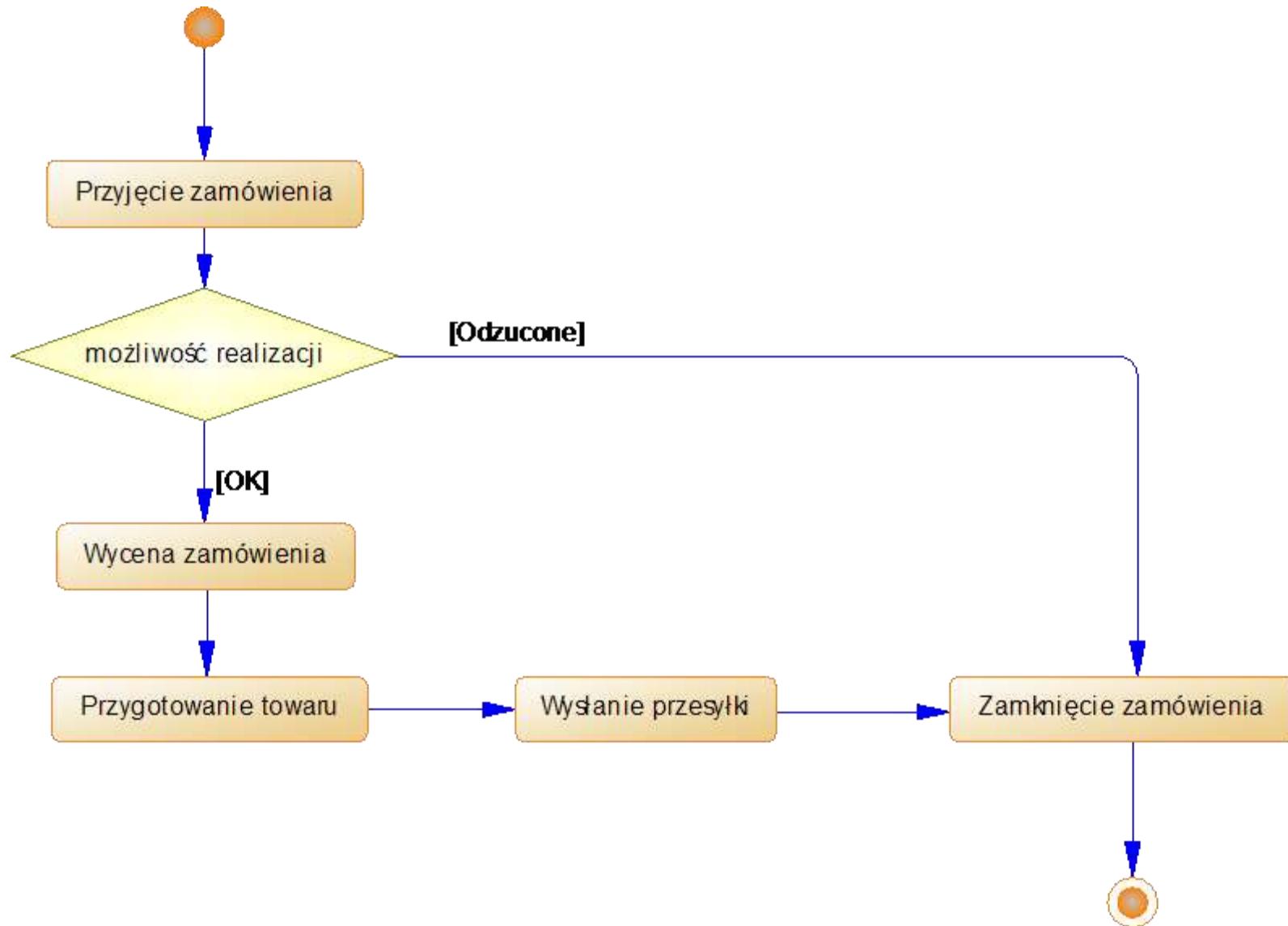


Diagram czynności

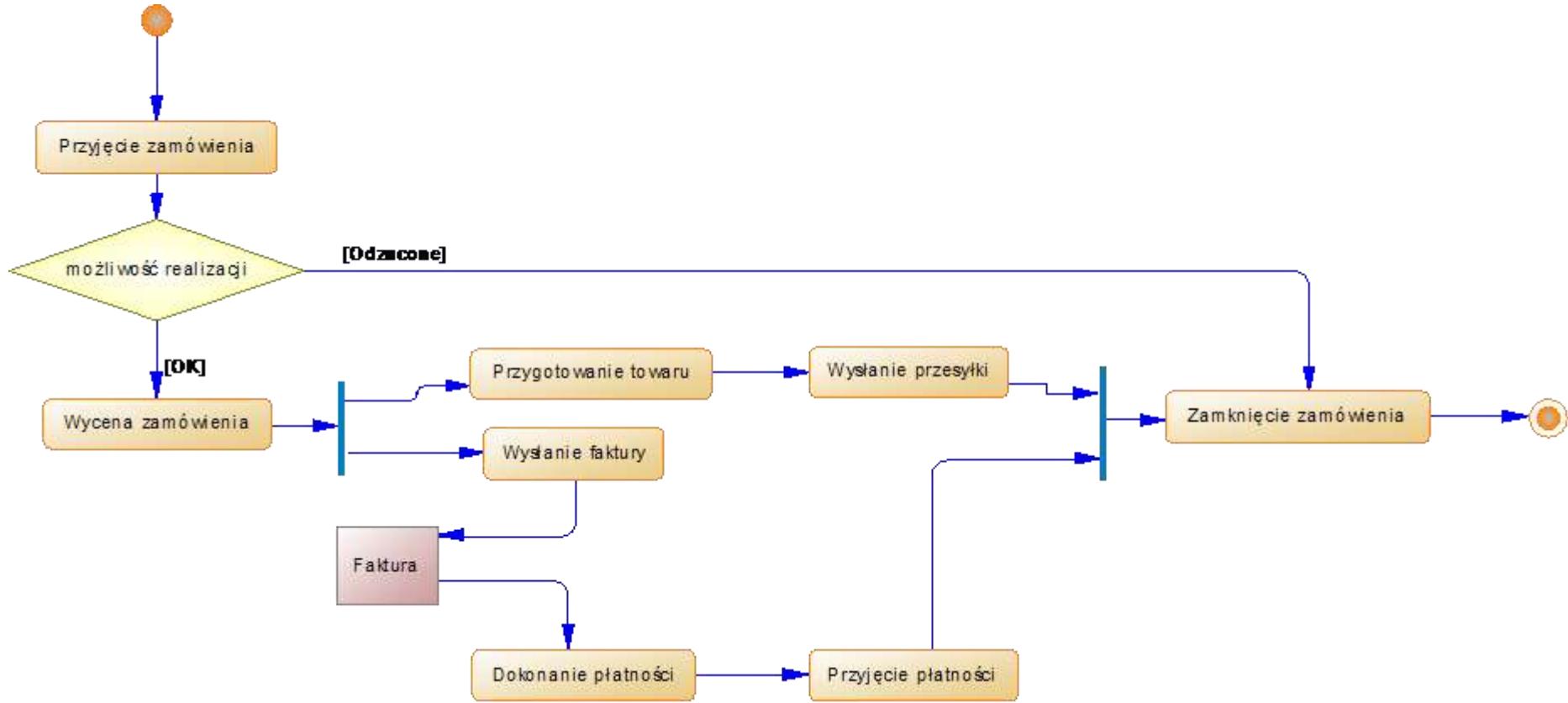


Diagram czynności

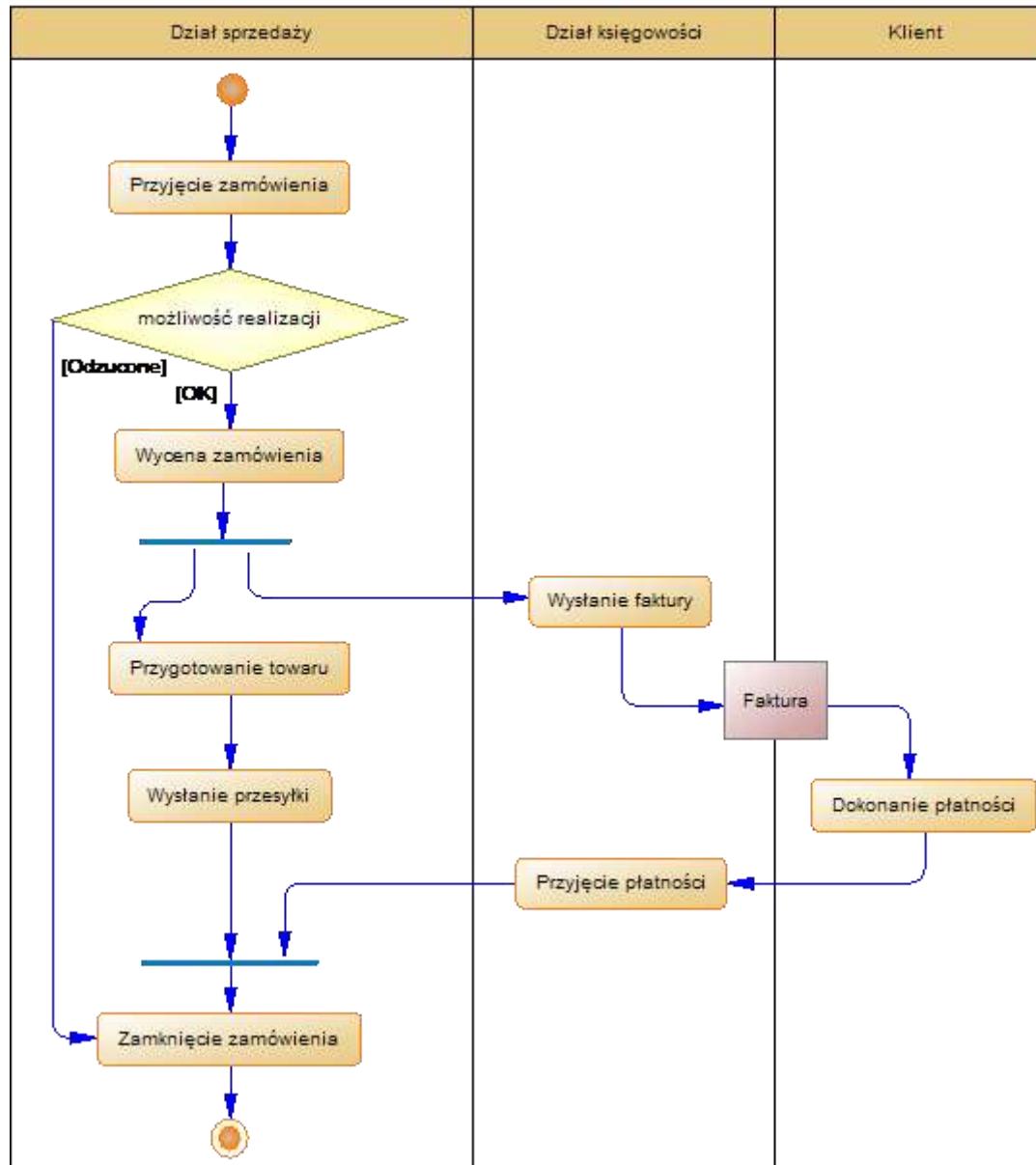


Diagram maszyny stanowej

Modelowanie dziedziny

Diagram maszyny stanowej

- stany, w jakich może znajdować się obiekt
 - dopuszczalne przejścia między stanami
-
- ang.: *state machine diagram*
 - pokazane stany pojedynczego obiektu

Diagram maszyny stanowej cykl życia woluminu

[after (termin zwrotu)]/ wysłanie powiadomienia

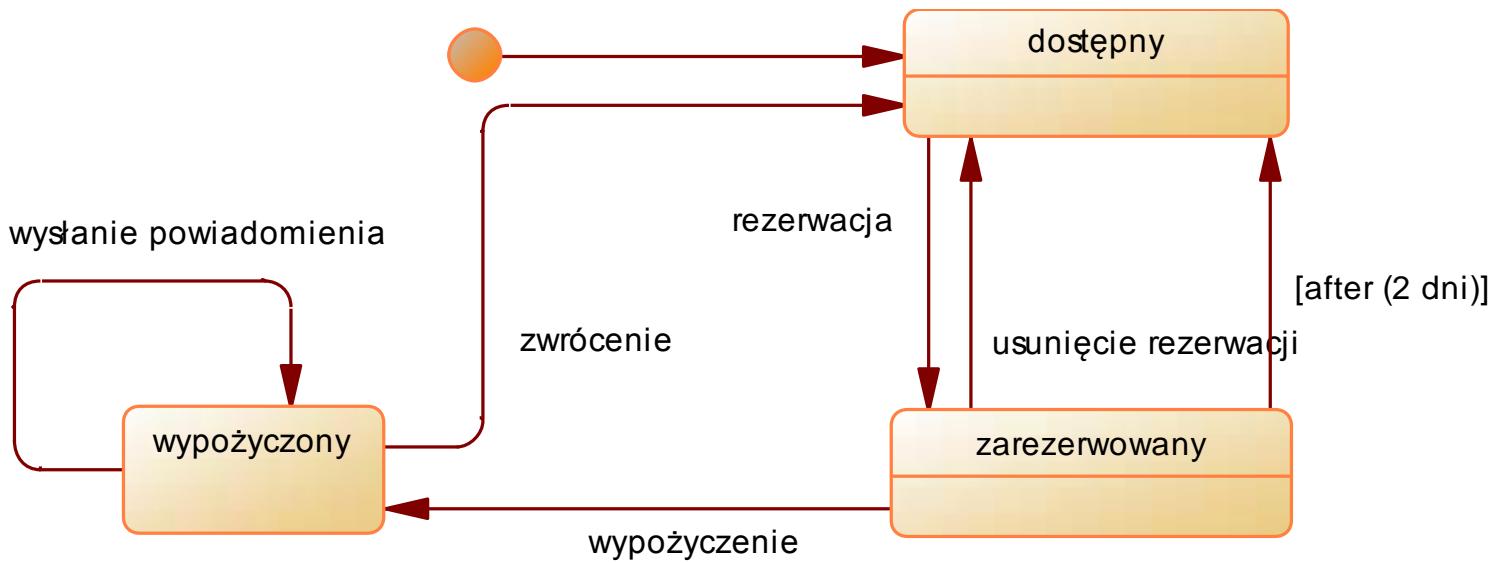


diagram komponentów
diagram pakietów
diagram rozmieszczenia

Modelowanie struktury

Diagram komponentów

- struktura aplikacji na poziomie jednostek, które mogą być projektowane i wytwarzane niezależnie
- ang.: *component diagram*
- główne elementy
 - komponenty
 - zależności
 - interfejsy

Diagram komponentów

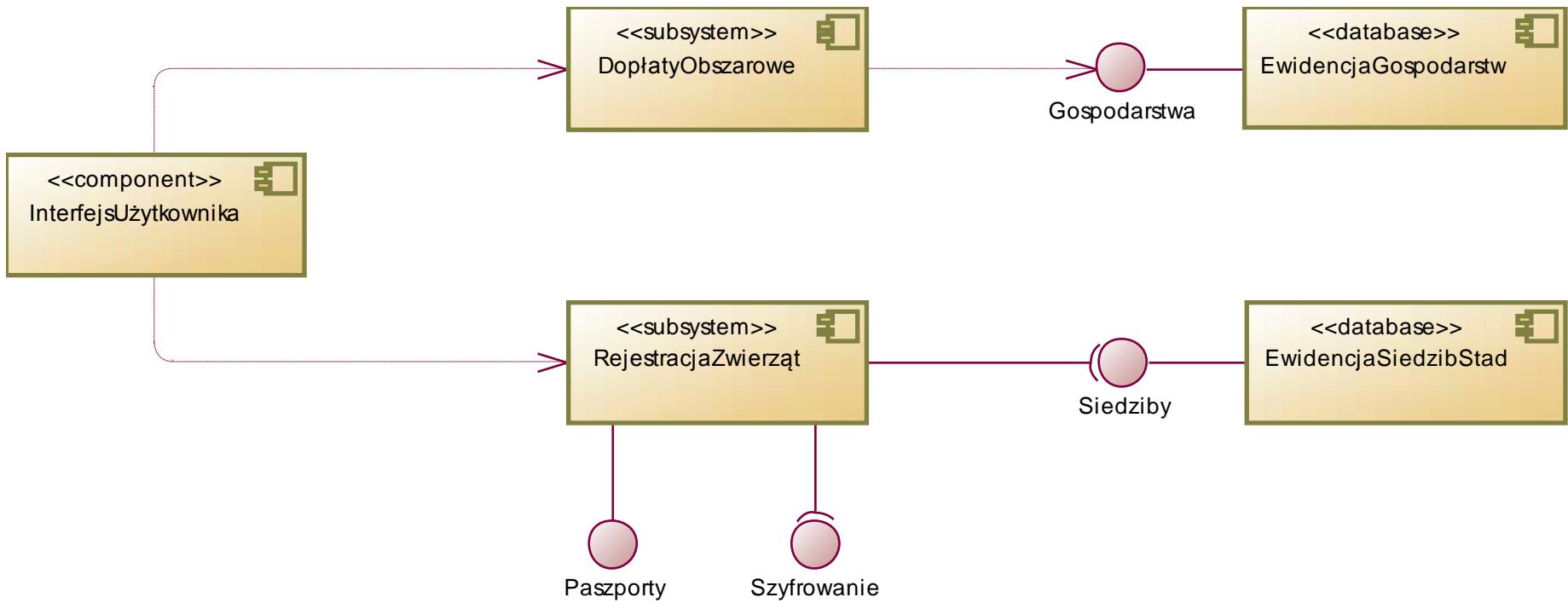


Diagram pakietów

- zależności między pakietami
- ang.: *package diagram*
- pakiet
 - pojemnik przechowujący wybrane elementy modelu
 - głównie klasy
- należy minimalizować zależności między pakietami

Diagram pakietów

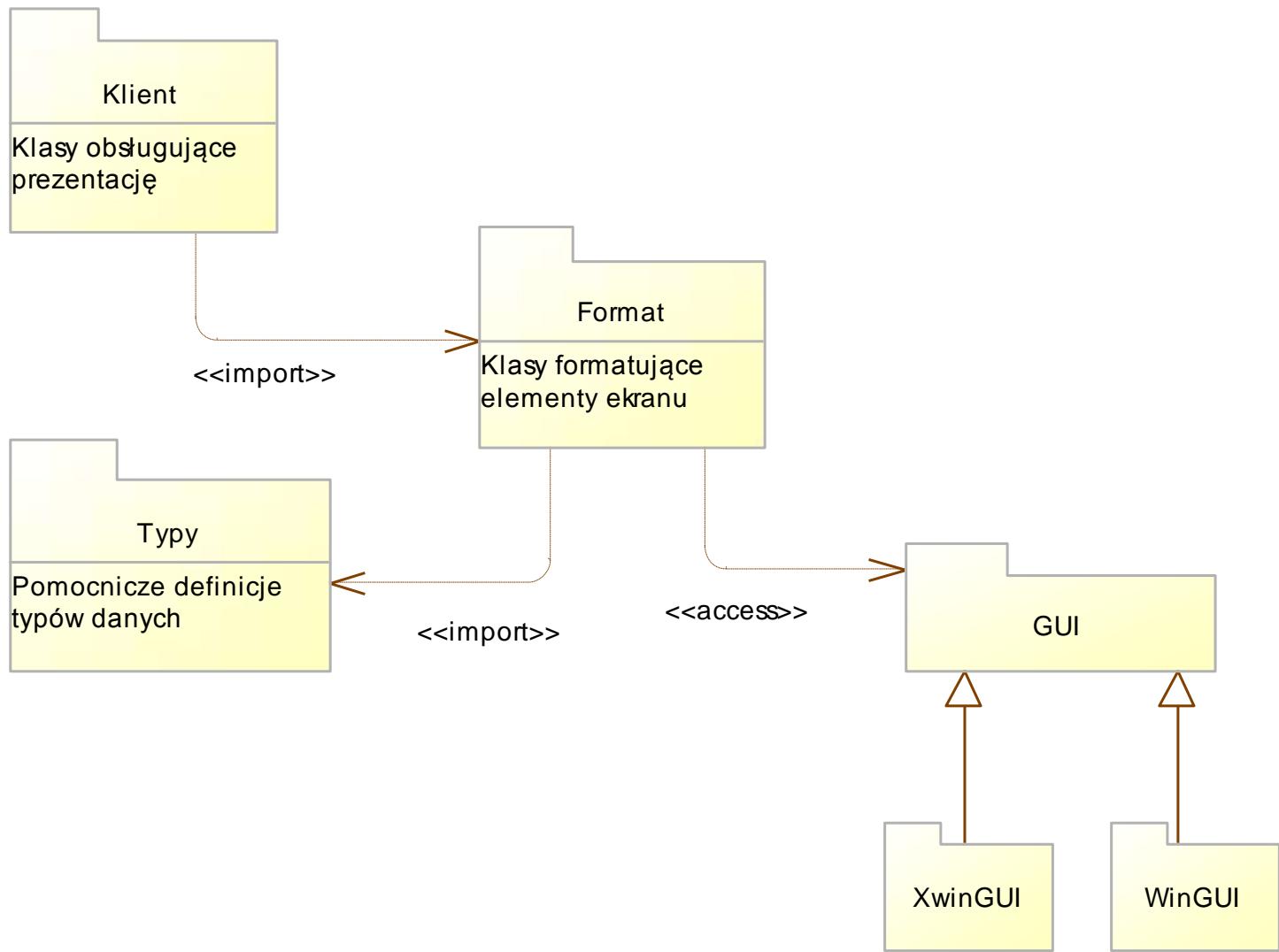


Diagram rozmieszczenia

- struktura budowanego systemu
i
- sposób rozmieszczenia komponentów programowych w węzłach systemu
- ang.: *deployment diagram*
- węzły to środowiska wykonawcze
 - komputery, serwery aplikacyjne, kontenery
- jako komponenty mogą również występować artefakty projektowe implementujące te komponenty
 - pliki wykonywalne – exe, jar, dll
 - pliki konfiguracyjne – xml, ini
 - skrypty generowania tabel - ddl

Diagram rozmieszczenia

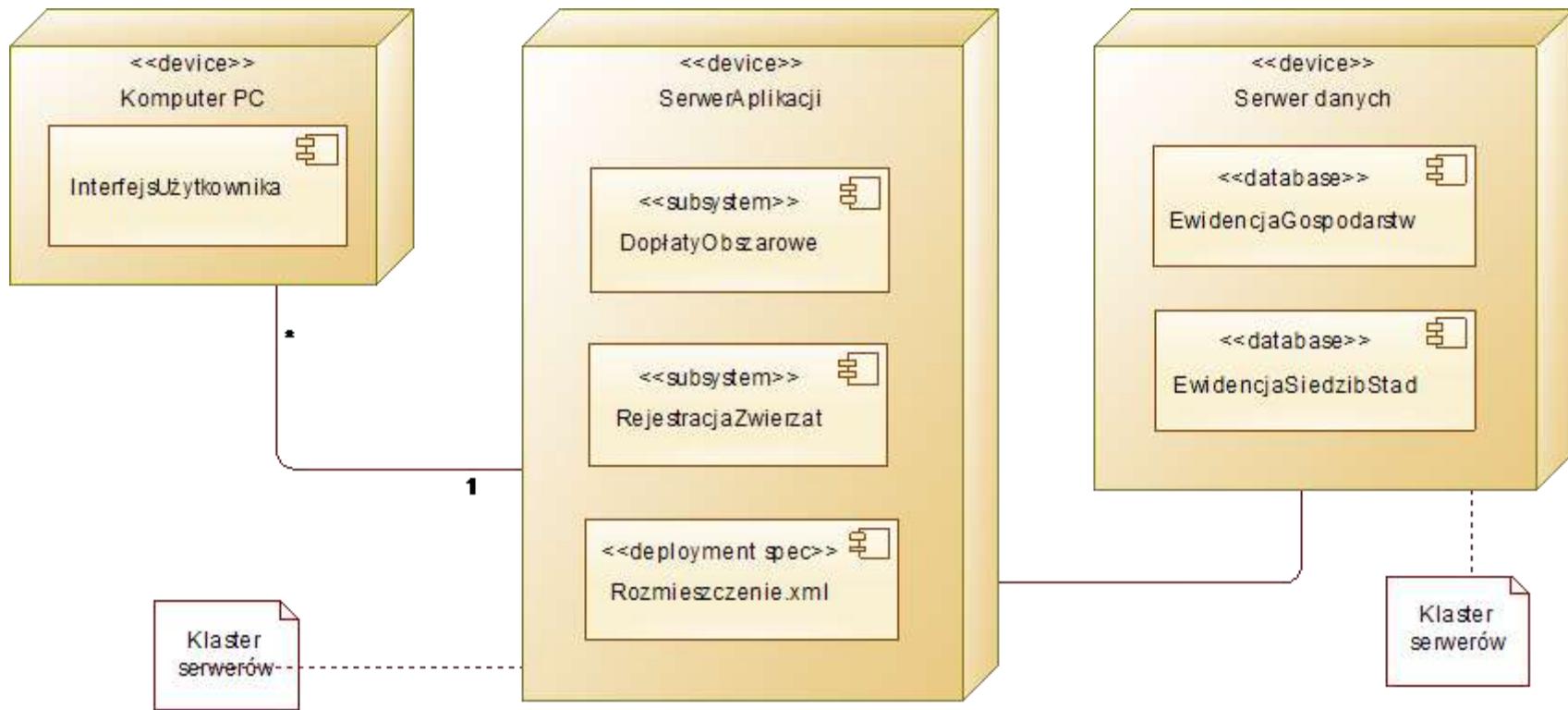


diagram sekwencji
diagram komunikacji

Modelowanie współdziałania

Diagram sekwencji

- pokazanie jednostek współpracujących przy realizacji zadania oraz komunikatów wymienianych między nimi
- inna nazwa: diagram przebiegu
- ang.: *Sequence Diagram*
- stanowi formę specyfikacji **pojedynczego** (zazwyczaj) przypadku użycia
- zawiera wymiar czasu (osi pionowa)
 - czynności realizowane później umieszczane są niżej

Diagram sekwencji

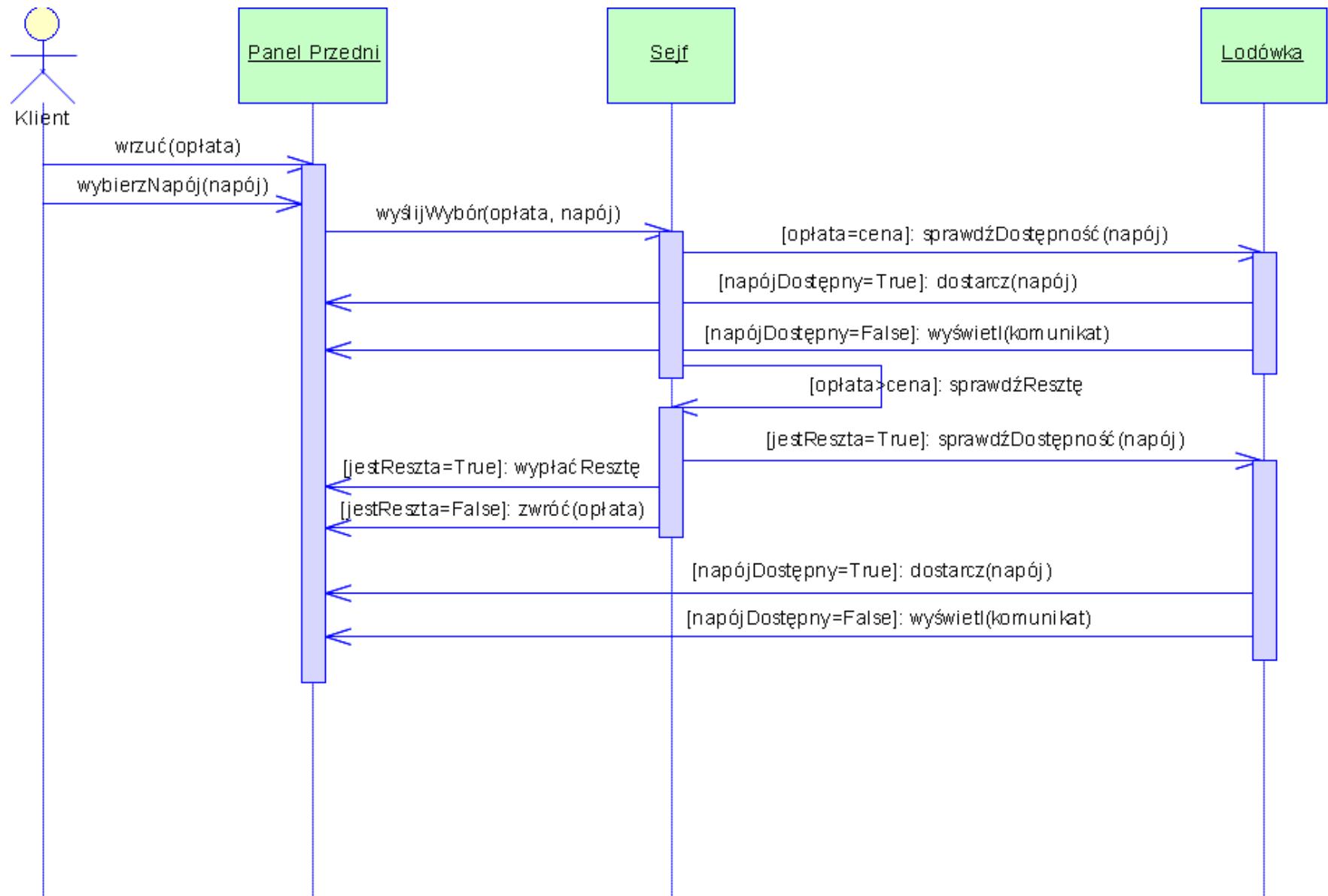


Diagram sekwencji

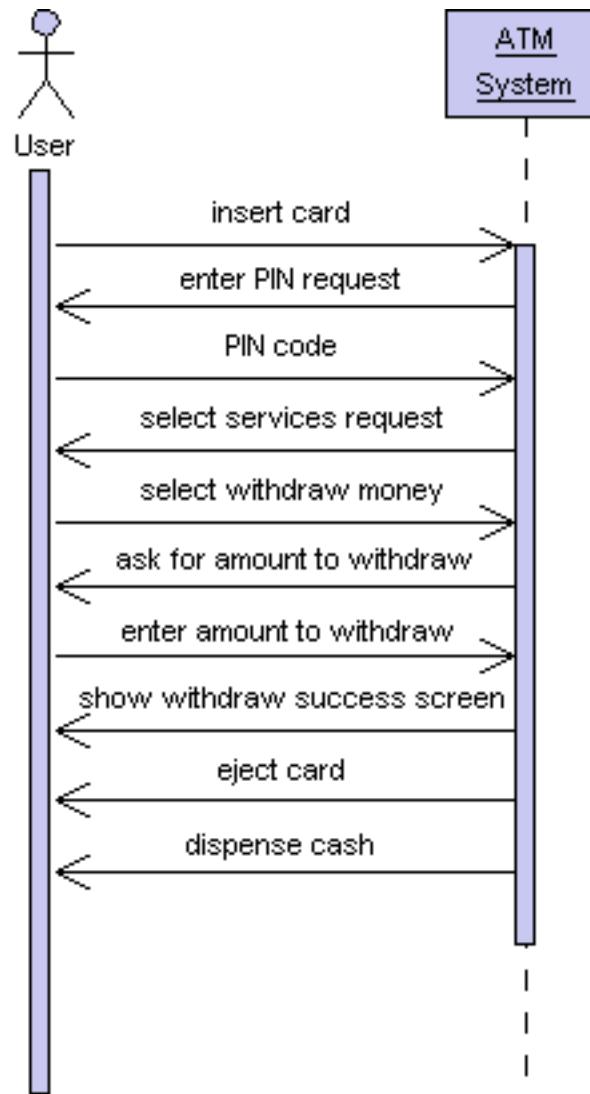
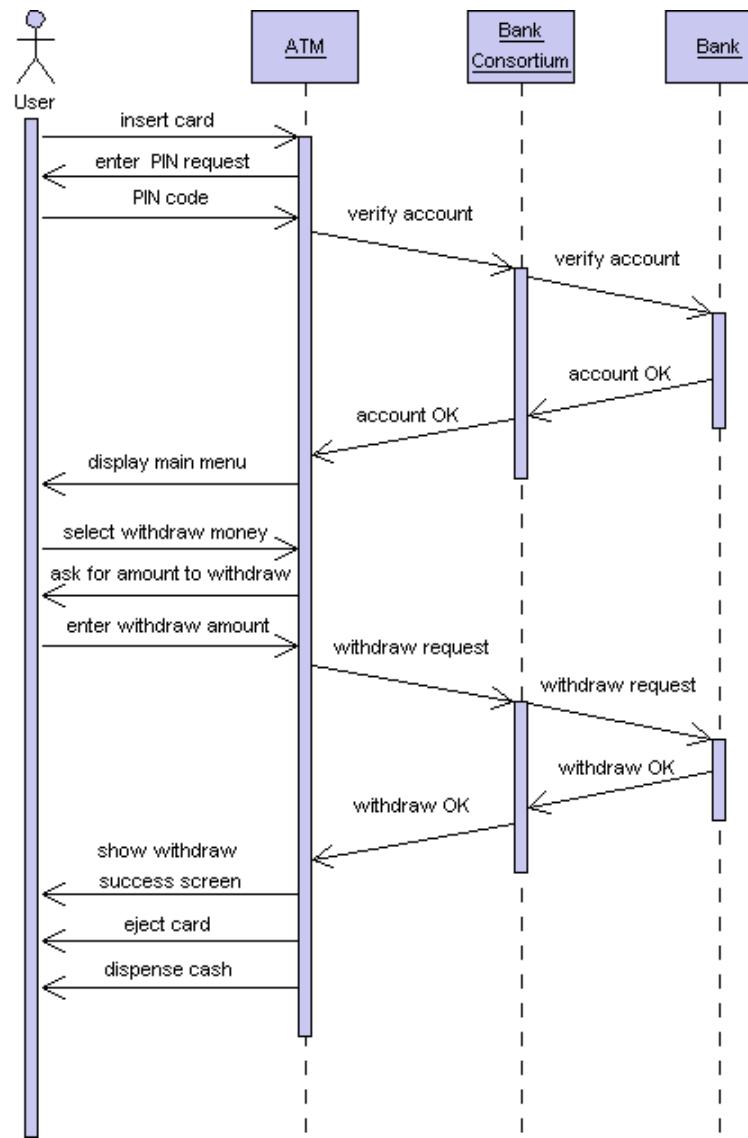


Diagram sekwencji



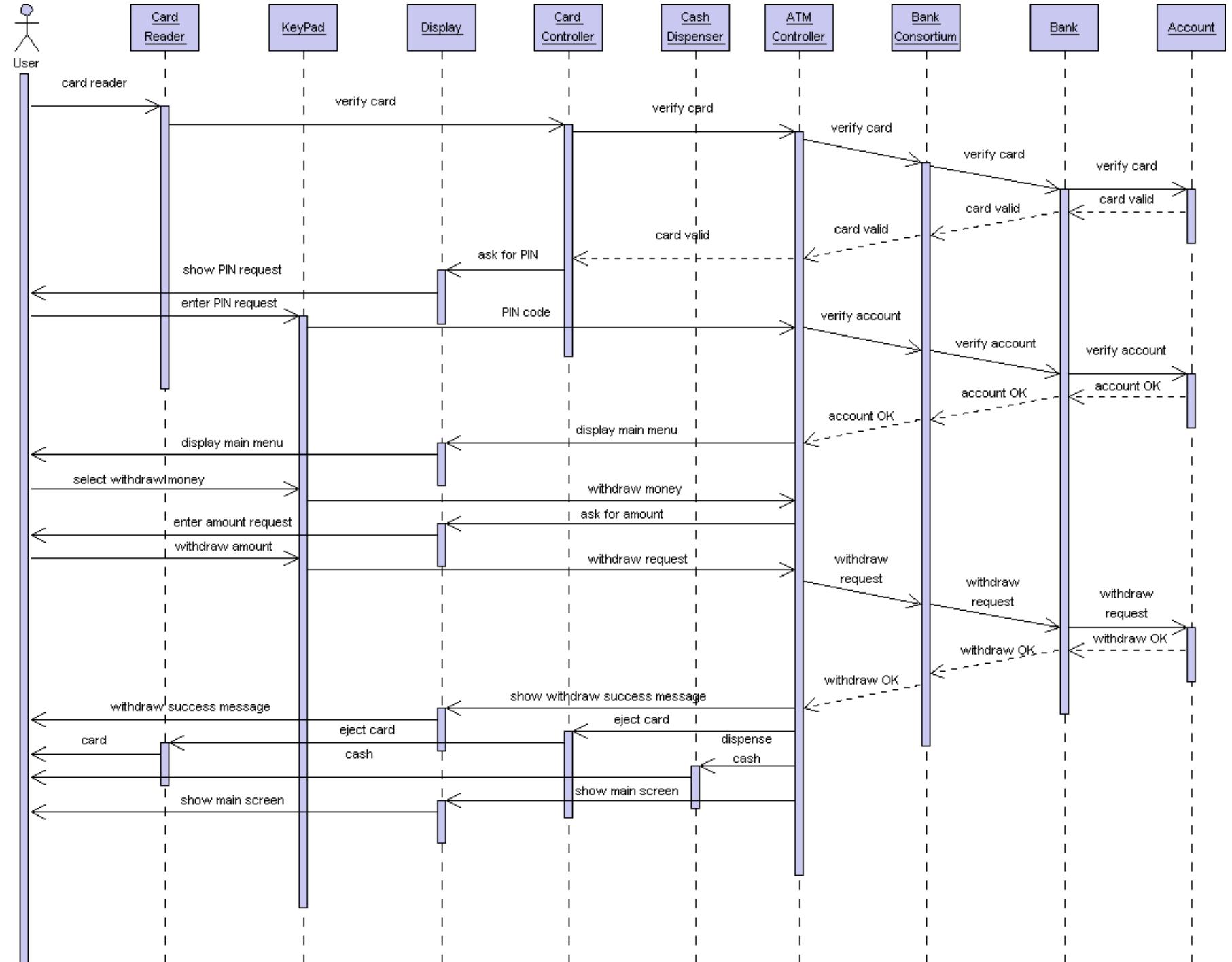
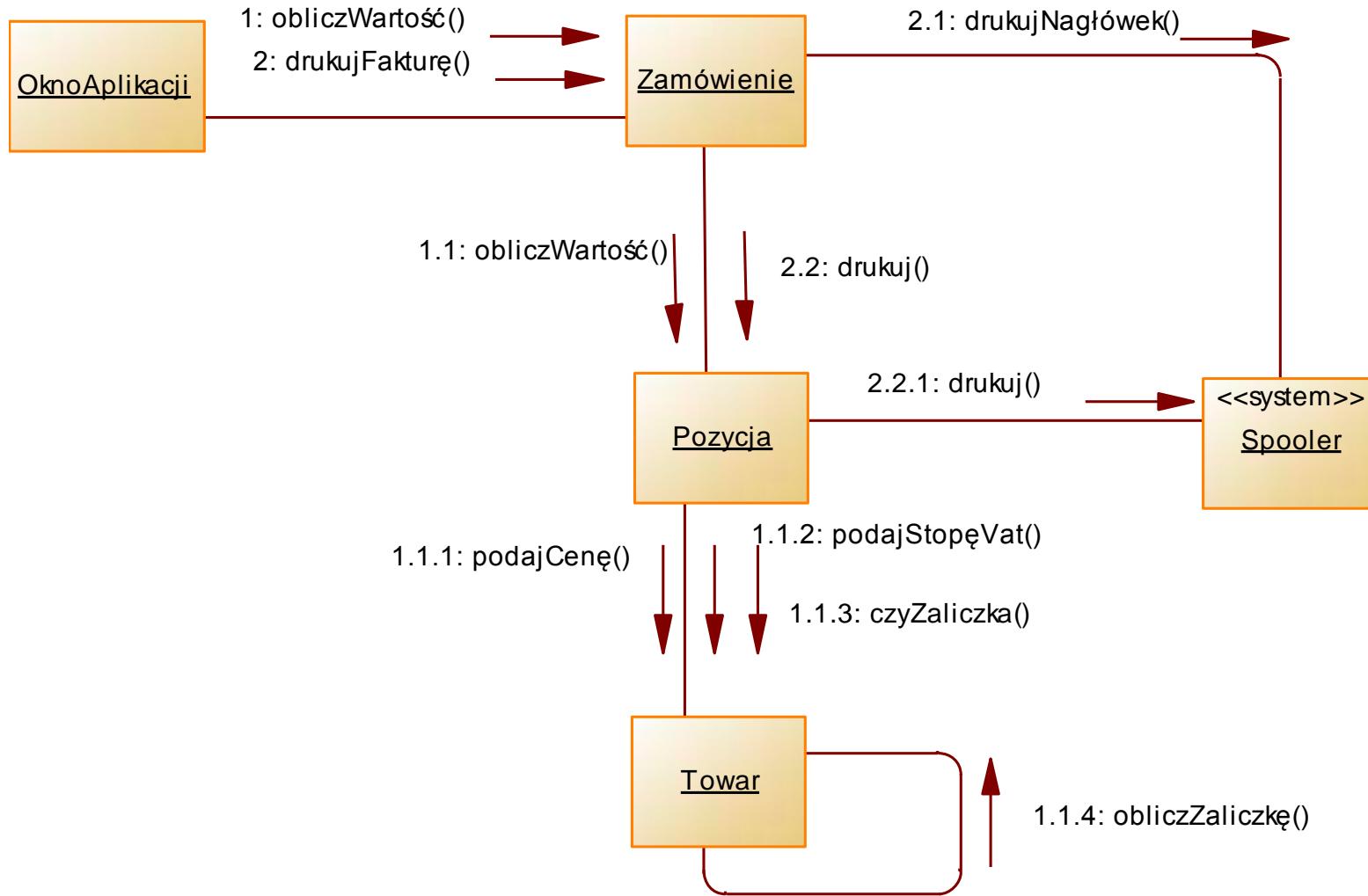


Diagram komunikacji

- przedstawienie komunikacji i sposobu współpracy obiektów podczas wspólnej realizacji zadań
- ang.: *communication diagram*
- główne elementy
 - obiekty
 - komunikaty między obiektami
- elementy i treść – jak na diagramie sekwencji!
 - inna forma graficzna
 - nacisk na strukturę współpracy

Diagram komunikacji



SysML

SysML

- język modelowania do inżynierii systemów
- wspomaga
 - specyfikację
 - analizę
 - projektowanie
 - weryfikację i zatwierdzanie
- w "systemach systemów" (*systems of systems*)
 - systemy będące częściami większych systemów
- jest rozszerzeniem podzbioru UML
- UML → dla przedsięwzięć software'owych
- SysML → szerszy zakres
 - sprzęt, oprogramowanie, informacje, procesy, personel, urządzenia

SysML – cechy

- **semantyka bardziej elastyczna i ekspresyjna**
 - nowe typy diagramów
 - wymagań – zależności i związki z innymi elementami
 - do inżynierii wymagań i V&V
 - parametryczny – parametryczne zależności między elementami struktury
 - analiza wydajności i analizy ilościowe
- **jest mniejszy niż UML**
 - usunięcie konstrukcji dedykowanych dla oprogramowania
 - mniej diagramów
 - mniej konstrukcji
- **wspieranie różnych typów rodzajów alokacji**
 - UML – alokacje tabelaryczne
 - elastyczne tabele alokacji – alokacje wymagań, funkcjonalna, strukturalna
 - wspomaganie V&V i analizy odchyleń
- **konstrukcje zarządzania modelem wspierają modele, widoki, punkty widzenia**
 - rozszerzają możliwości UML
 - architektonicznie zgodne ze standardem IEEE-Std-1471-2000

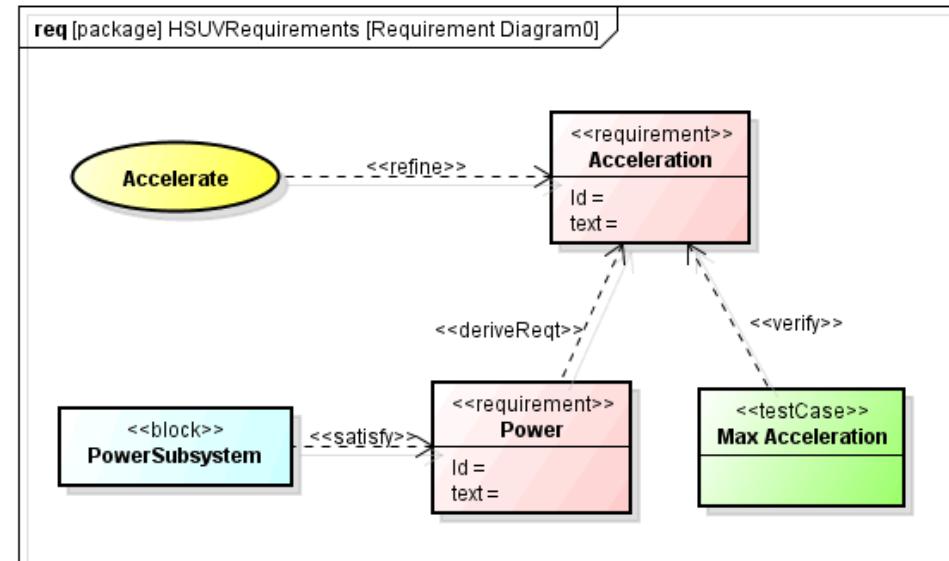
SysML – diagramy przejęte z UML

- czynności
- klas
- struktury złożonej
- sekwencji
- maszyny stanowej
- przypadków użycia
- pakietów
- dodatkowe
 - wymagań
 - parametryczny

SysML

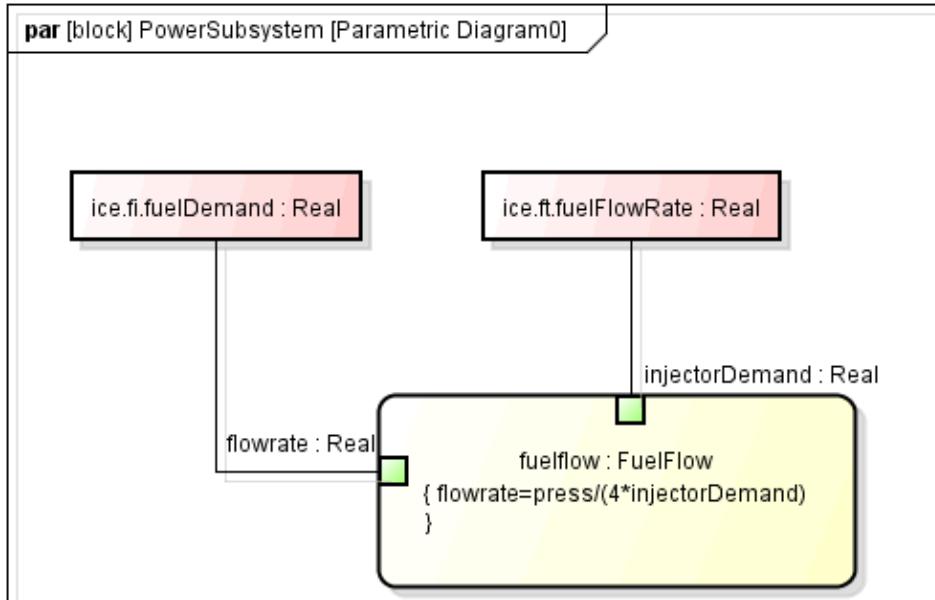
- diagram wymagań

<http://astah.net/tutorials/sysml/requirement-diagram>



- parametryczny

<http://astah.net/tutorials/sysml/parametric>



Podsumowanie

podstawowe koncepcje
obiektowego
modelowania oprogramowania

Pytania



Literatura

- Sacha K., Inżynieria oprogramowania, PWN 2010
- Górska J. (red.), Inżynieria oprogramowania w projekcie informatycznym, Mikom 2000
- Eeles P., Cripps P., The process of software architecting, Addison-Wesley 2010
- Schmoller J., UML dla każdego, Helion 2003
- <http://en.wikipedia.org/wiki/SysML>
- <http://www.sysmlforum.com/>

Następny wykład

Projektowanie architektury systemu c.d. i wzorce architektoniczne

Inżynieria oprogramowania

Wykład 5: Projektowanie architektury systemu – c.d.

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

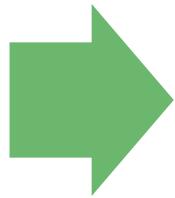
Agenda

- Podstawy projektowania oprogramowania
- Wzorce architektoniczne
- Czynności projektowania systemu

Podstawy projektowania oprogramowania

Czym jest projektowanie?

model
analityczny



model
projektowy

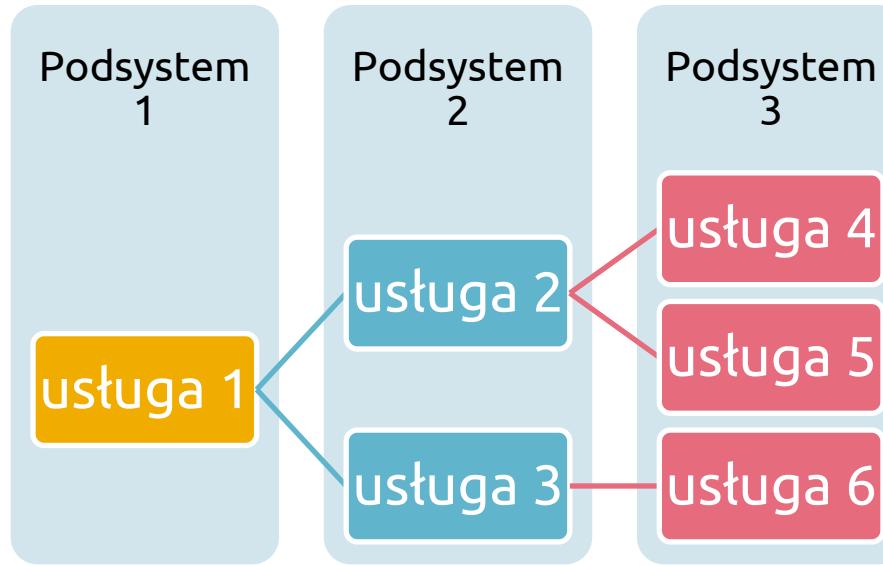
Składniki modelu analitycznego

- model przypadków użycia
- model obiektowy/klas
- diagramy sekwencji
 - dla każdego przypadku użycia
- zbiór wymagań niefunkcjonalnych i ograniczeń

Produkty projektowania

- cele projektowe
 - na podstawie wymagań niefunkcjonalnych
- architektura programowa
- graniczne przypadki użycia

Ogólny schemat architektury oprogramowania

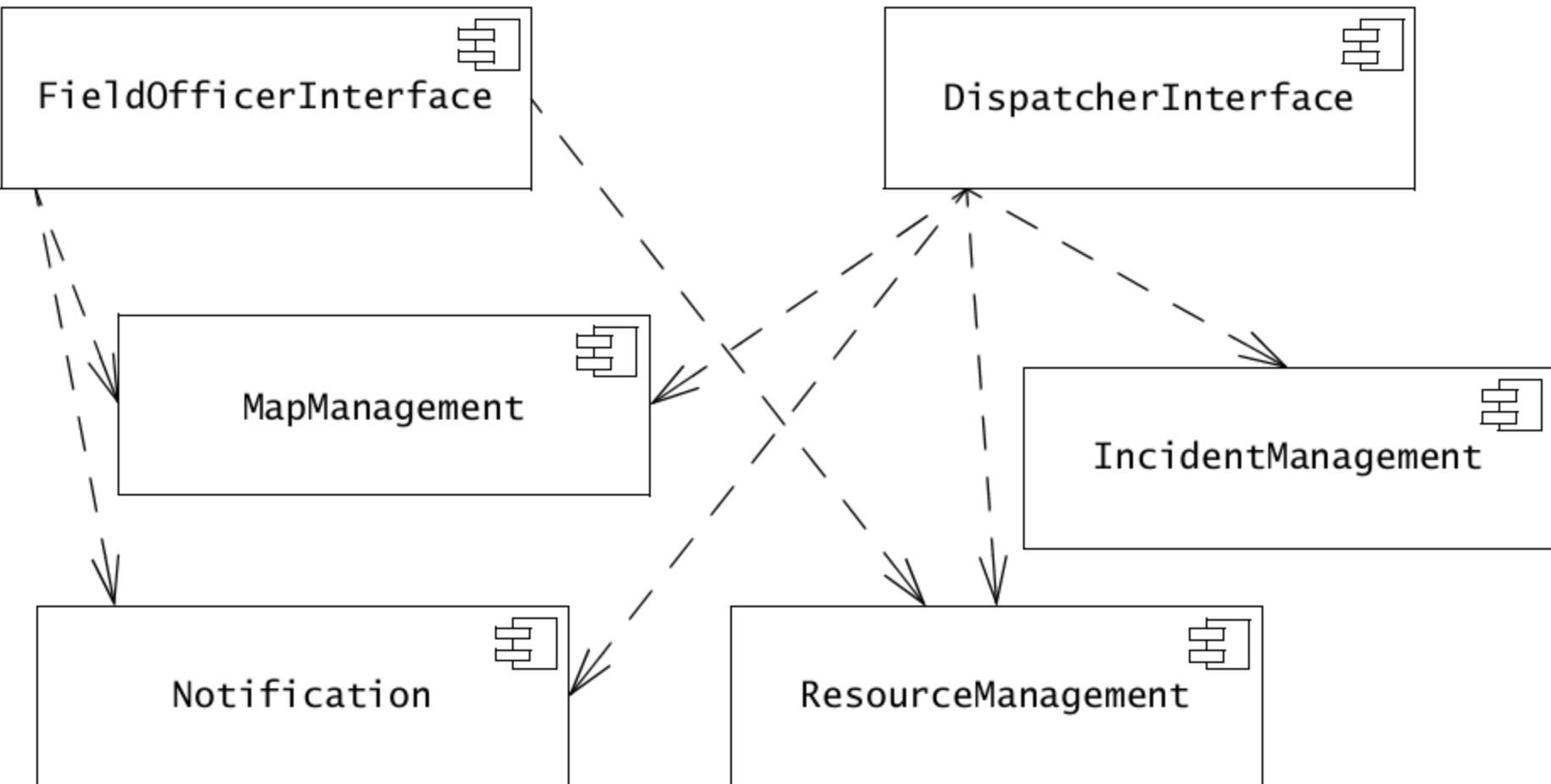


sprzężenie między podsystemami → **niskie**
spoistość podsystemu → **wysoka**

Podsystem

- **wymienna część systemu**
 - posiada dobrze zdefiniowane interfejsy
 - hermetyzuje stan i zachowanie składowych klas
- pracochłonność budowy podsystemu → zwykle pojedynczy zespół
- podsystemy mają być niezależne → niezależna praca zespołów
 - niewielki narzut na komunikację

Dekompozycja – diagram komponentów



Komponenty a języki/technologie

- pakiety – Java
- moduły – Modula, Pascal i pochodne
- przestrzenie nazw – C#, C++
- pliki źródłowe w katalogach i podkatalogach
- biblioteki – DLL

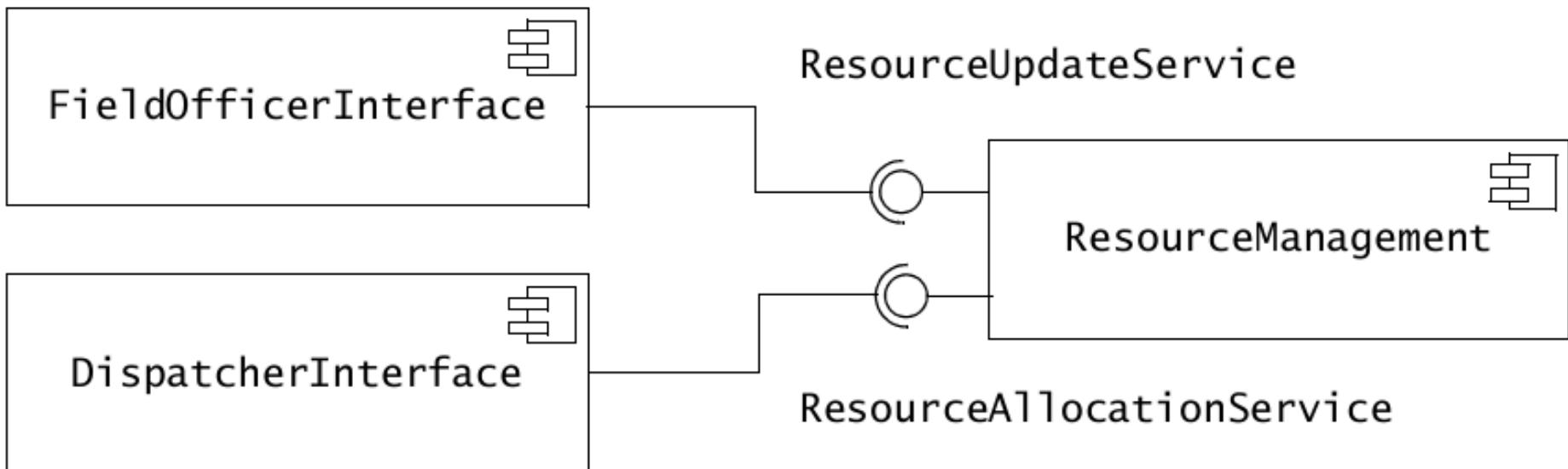
Usługa

- **zbiór powiązanych operacji podporządkowanych realizacji wspólnego celu**
- np. usługa powiadamiania – operacje:
 - wysyłanie powiadomień
 - nasłuchiwanie w kanałach komunikacyjnych
 - subskrybowanie do kanałów
 - anulowanie subskrypcji

Interfejs

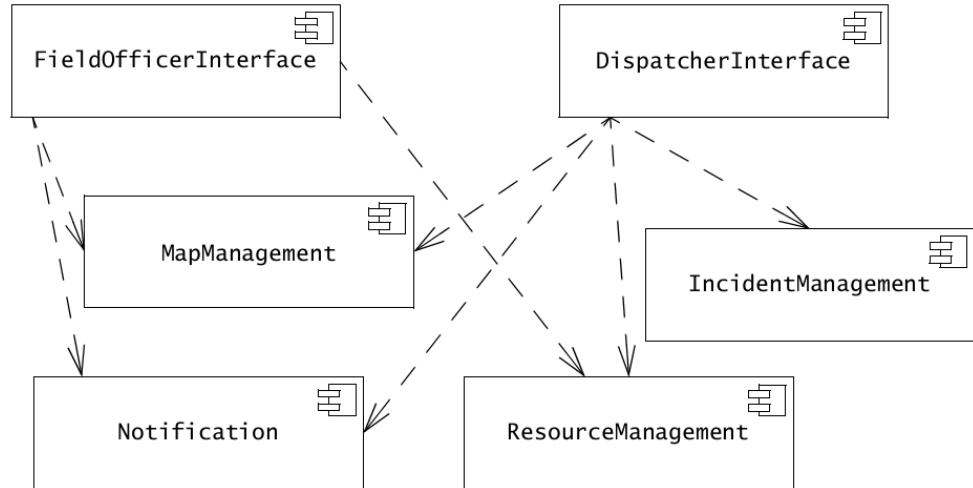
- zbiór operacji, jakie podsystem udostępnia innym podsystemom
- każda operacja:
 - nazwa
 - zestaw parametrów
 - typ zwracanego wyniku
- doskonalenie interfejsów podsystemów → API

Udostępnianie usług

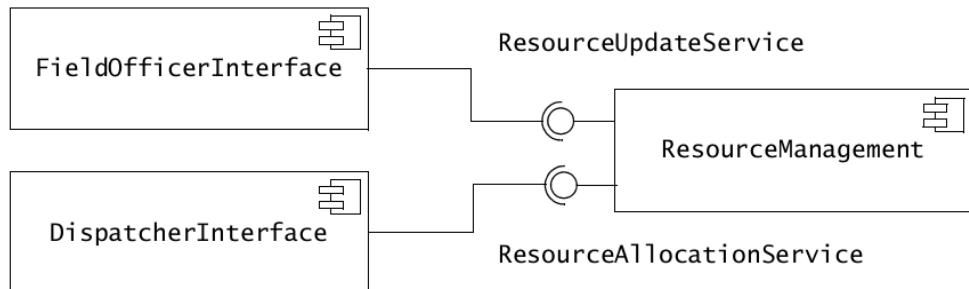


Dwie notacje na diagramach komponentów

- na początku:
 - zależności



- po określaniu podsystemów i ich usług
 - kółko-gniazdo

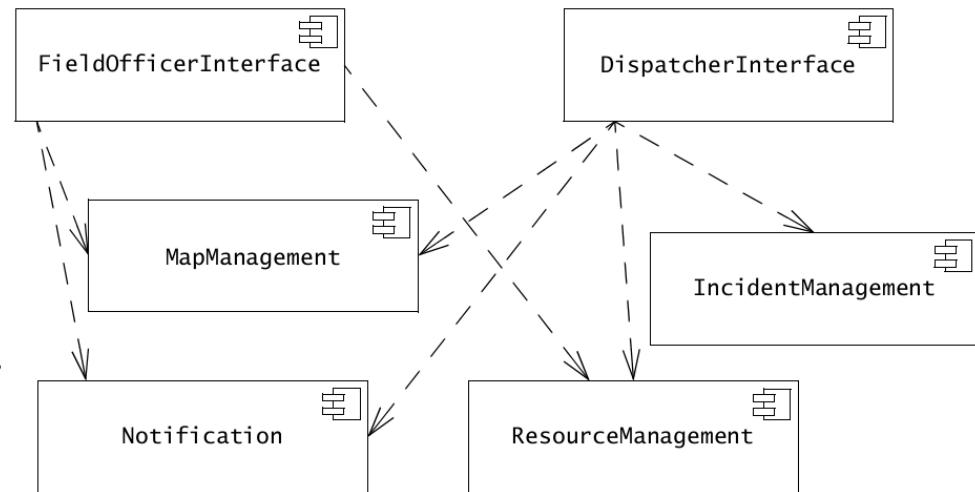


Sprzężenie i spoistość c.d.

- **sprzężenie** – liczba powiązań między podsystemami
- małe (luźne)
 - podsystemy są niezależne, niewielki wpływ na pozostałe
- wysokie (ścisłe)
 - podsystemy wrażliwe na zmiany

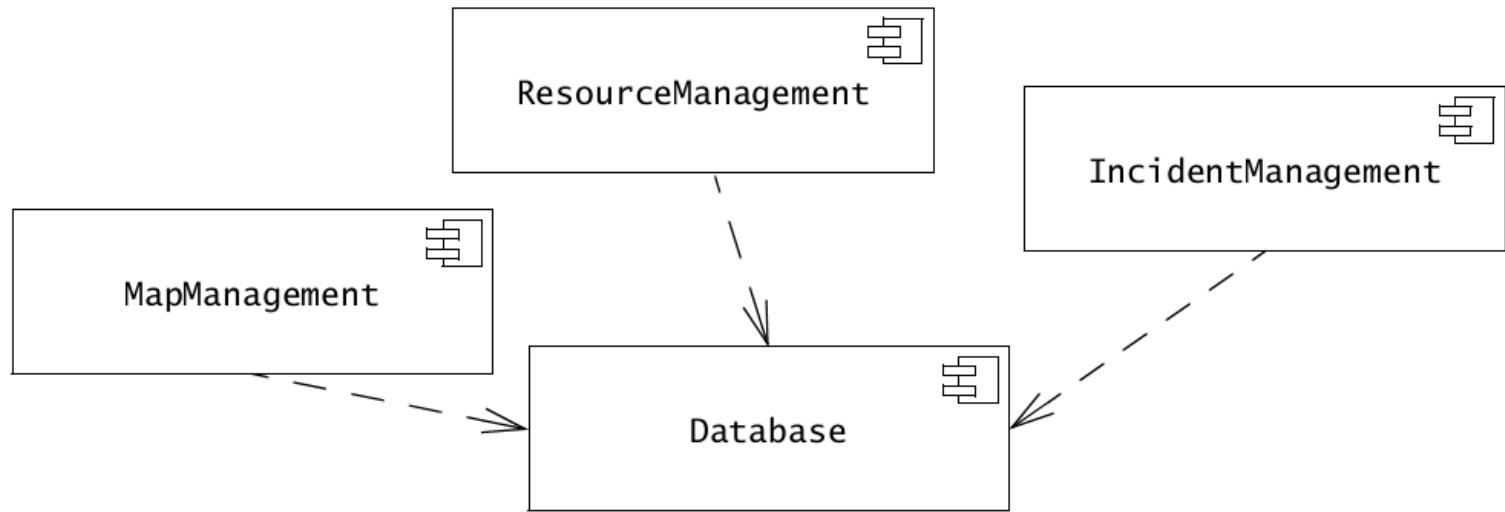
przykład:

podjęto decyzję:
dane będą w relacyjnej b.d.



Sprzężenie i spoistość c.d.

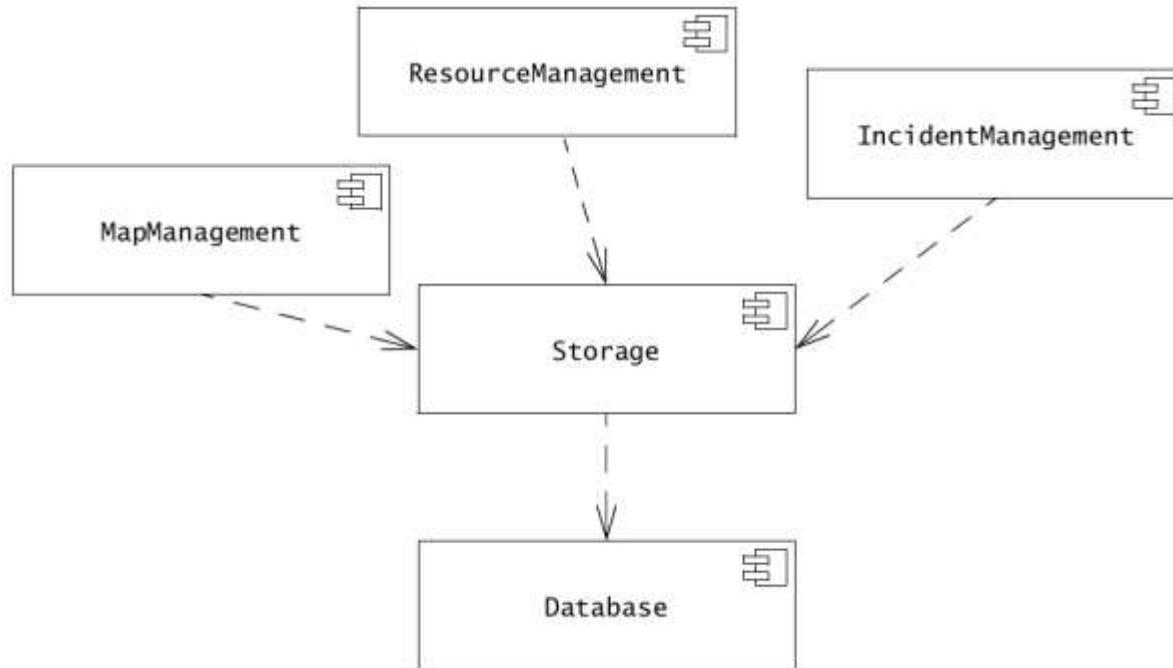
- efekt:



- problem
 - każdy komponent korzysta z natywnego SQL
 - zatem silne sprzężenie każdego z Database
 - co przy zmianie bazy na inną (dialekt SQL)?
 - konieczność modyfikacji sposobu komunikowania z Database

Sprzężenie i spoistość c.d.

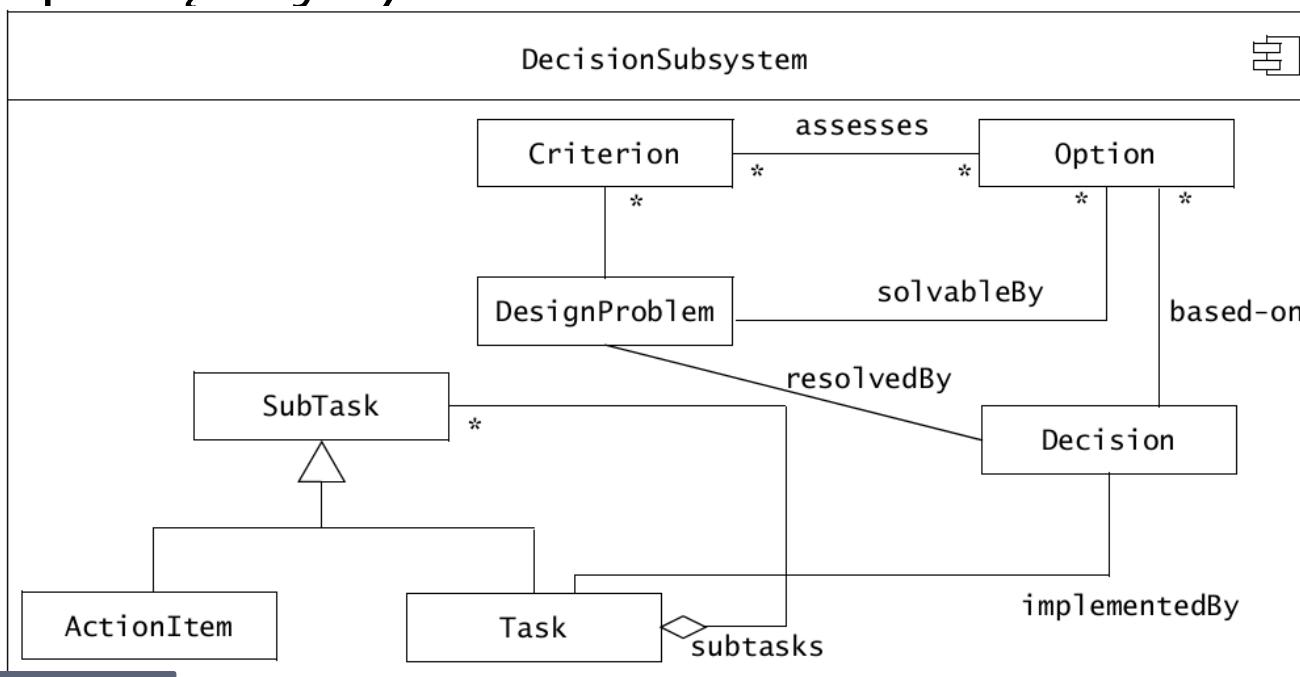
- lepiej:
 - Storage



- interfejs Storage **niewrażliwy na implementację** Database
- w przypadku zmiany Database → zmiana jedynie Storage

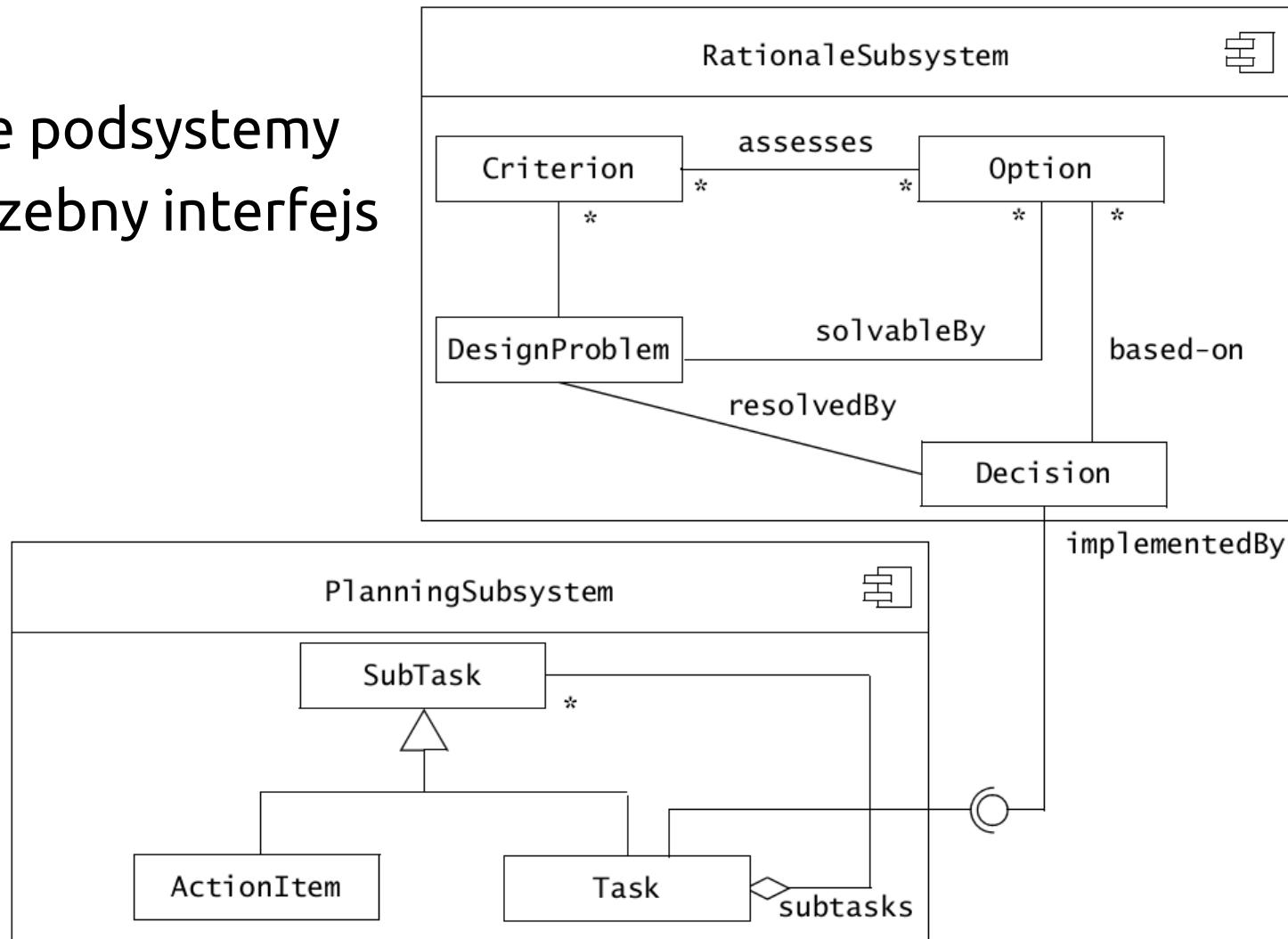
Sprzężenie i spoistość c.d.

- **spoistość** – powiązania własnych klas komponentu
 - wysoka
 - w podsystemie wiele powiązanych obiektów i realizujących podobne zadania
 - niska
 - w podsystemie niewiele obiektów i luźno powiązanych (lub w ogóle niepowiązanych)
- przykład



Sprzężenie i spoistość c.d.

- lepiej
 - prostsze podsystemy
 - ale potrzebny interfejs



Sprzężenie i spoistość c.d.

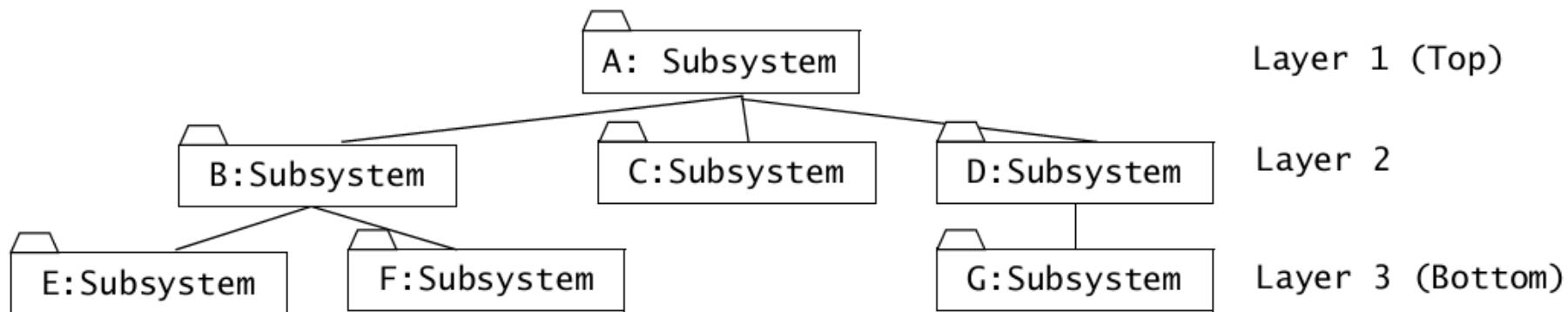
maksymalna spoistość
czy
minimalne sprzężenie



7 ± 2 elementy na danym szczeblu

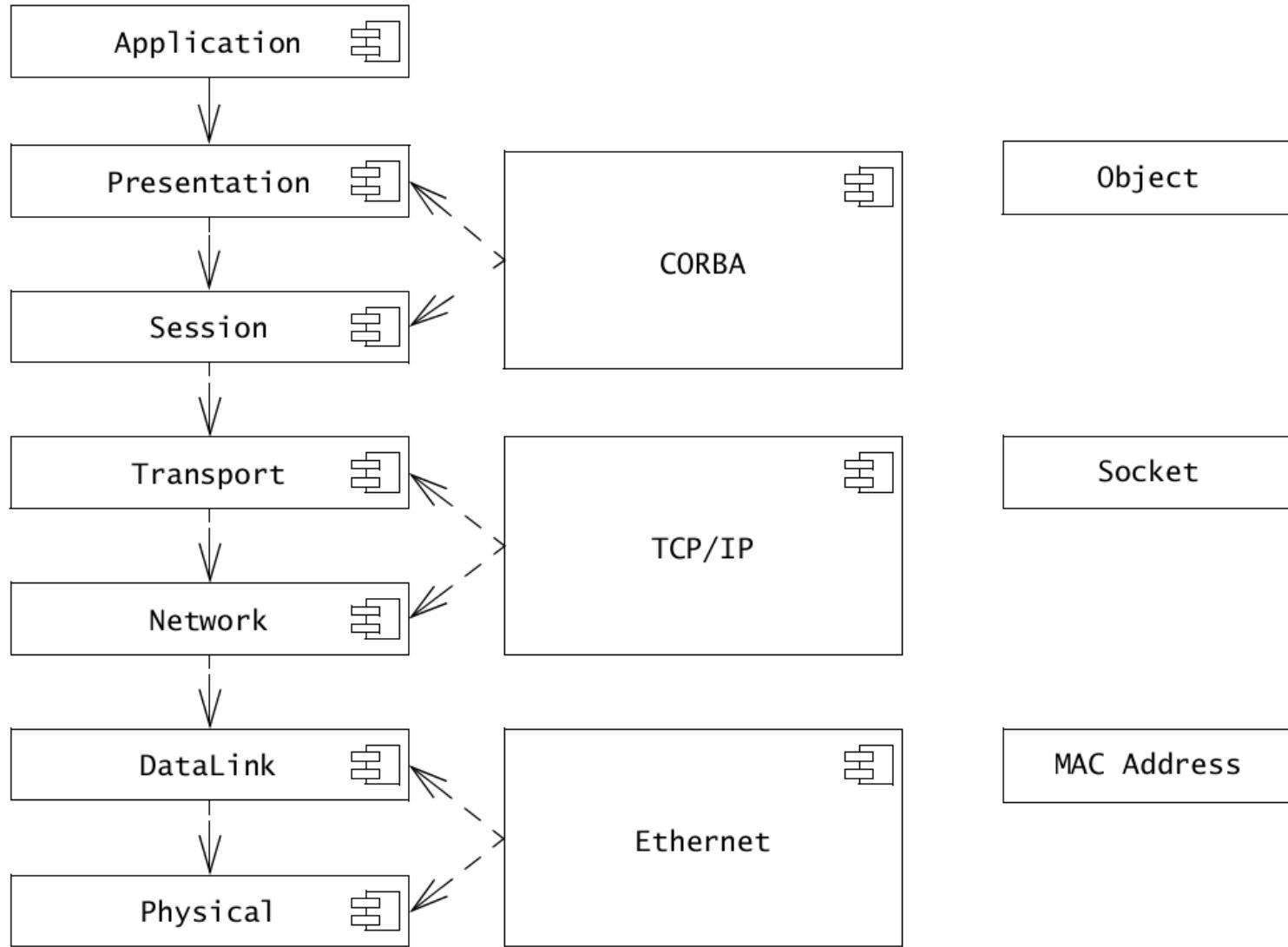
Warstwy i partycje

- **dekompozycja hierarchiczna → zbiór warstw**
- **warstwa**
 - zgrupowanie podsystemów oferujących powiązane usługi
 - zwykle korzystających z usług w nizszych warstwach



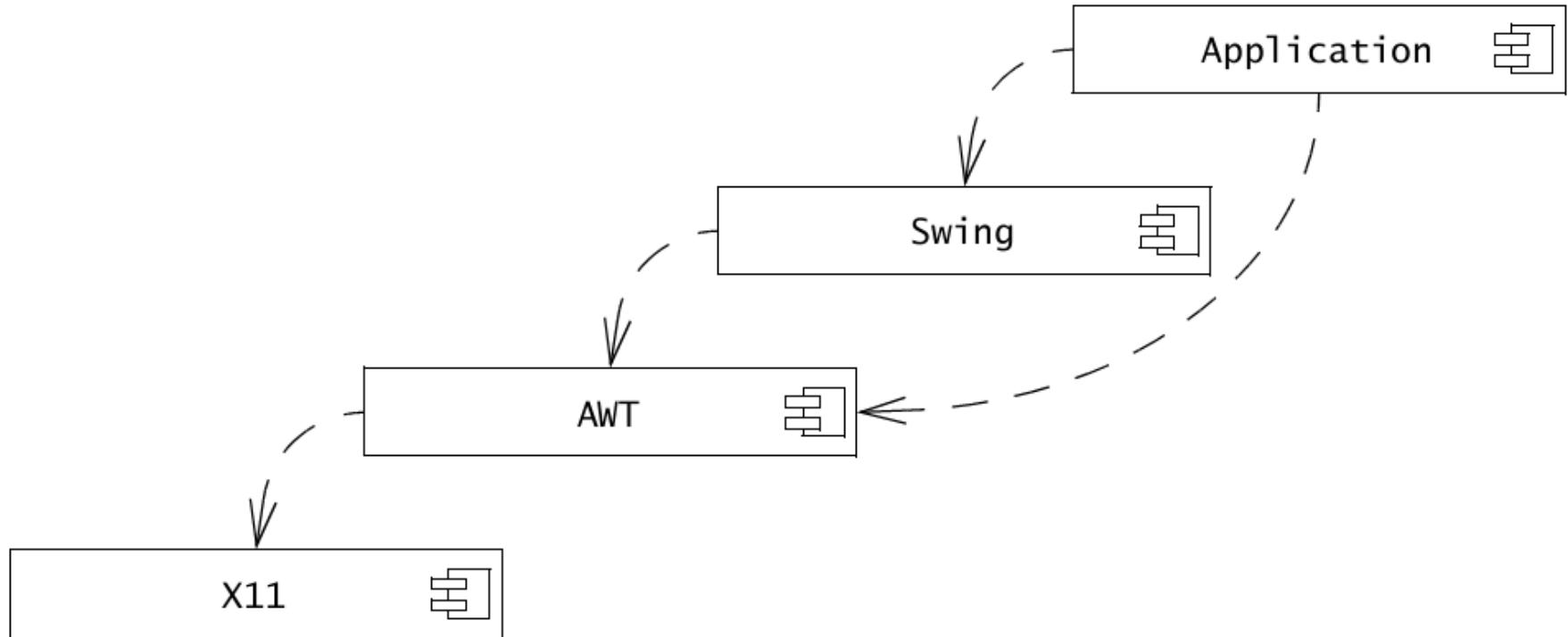
Warstwy i partycje

warstwy – architektura zamknięta



Warstwy i partycje

warstwy – architektura otwarta



po co używać otwartej?
niwelowanie wydajnościowych wąskich gardeł

Warstwy i partycje

- partycjonowanie – podział na równorzędne systemy
 - każdy odpowiedzialny za inną klasę usług
- np. samolot
 - nawigacja i trasa
 - personalizacja środowiska (pasażera)
 - zużycie paliwa
 - harmonogram przeglądów
- luźne powiązanie z pozostałymi
- ale możliwe funkcjonowanie bez innych

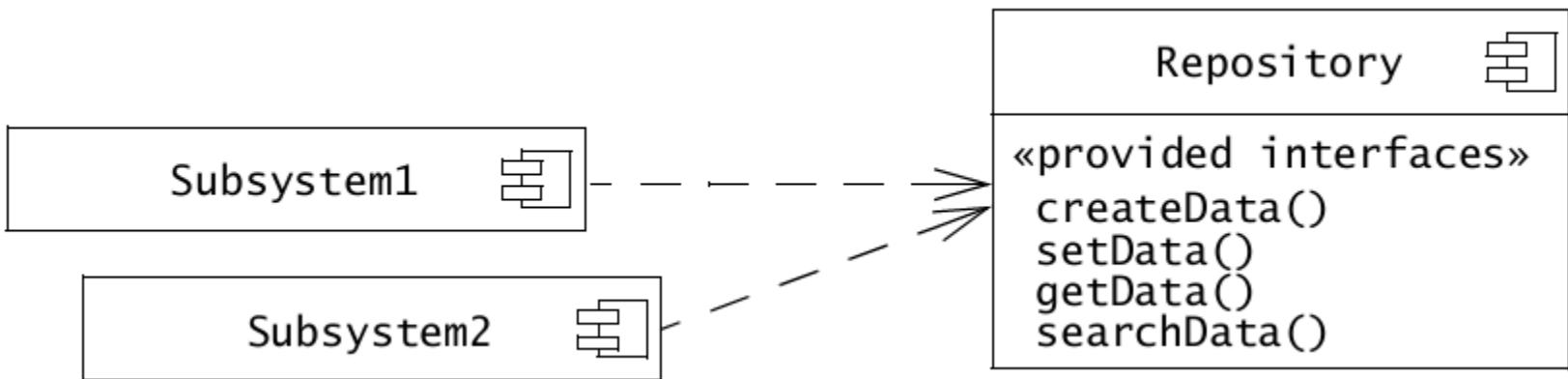
Wzorce architektoniczne

Wybrane wzorce architektoniczne

- repozytorium
- klient-serwer
- P2P – peer-to-peer
- MVC – model-widok-kontroler
- architektura trójwarstwowa
- architektura czterowarstwowa
- filtry i potoki

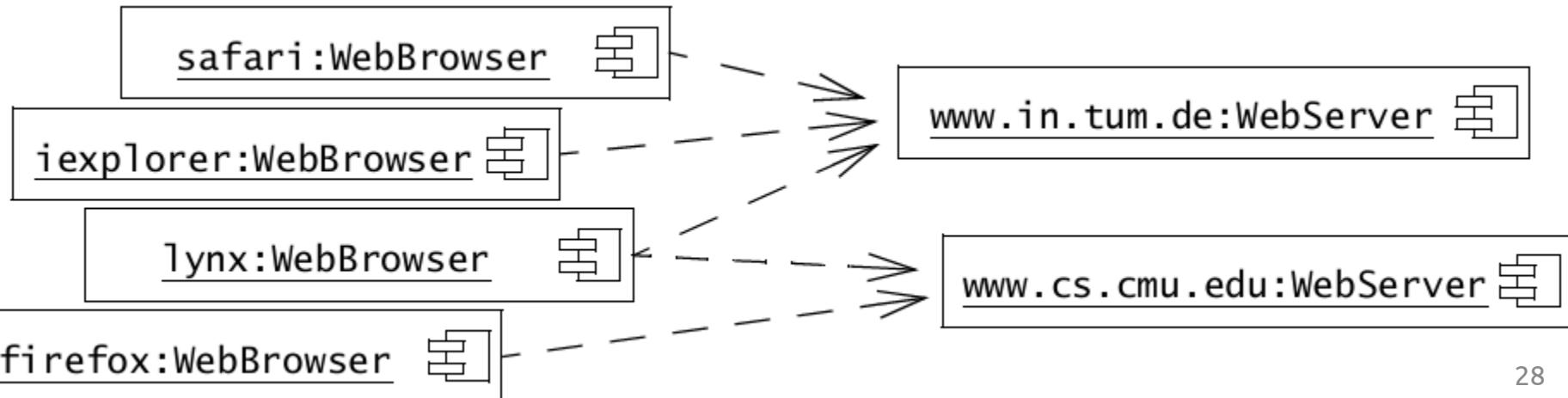
Wzorce architektoniczne – repozytorium

- podsystemy operują na pojedynczej strukturze danych (repozytorium)
- repozytorium – **jedynie** medium komunikacji między systemami
- sterowanie – podsystemy lub repozytorium
- zastosowanie – systemy skupione na obsłudze b.d.
- wada – repozytorium wąskim gardłem



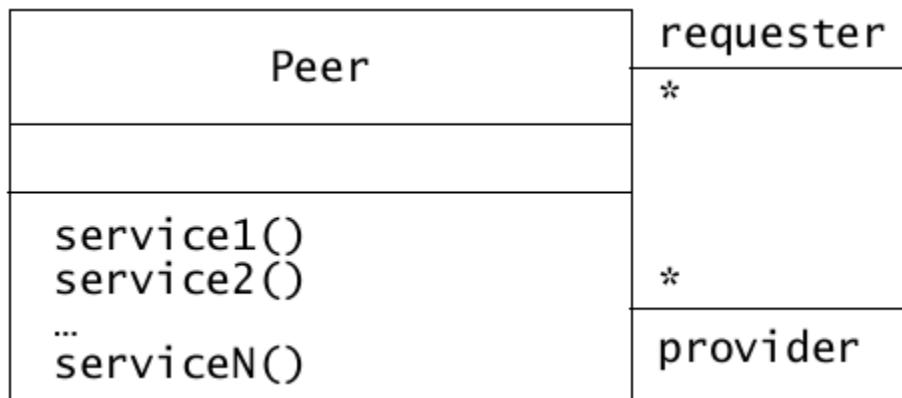
Wzorce architektoniczne – klient-serwer

- **serwer** – dostawca usług
- **klient** – odbiorca usług i interakcja z użytkownikiem
- żądanie za pomocą
 - Remote Procedure Call
 - obiektów brokerów
- *szczególny przypadek repozytorium*



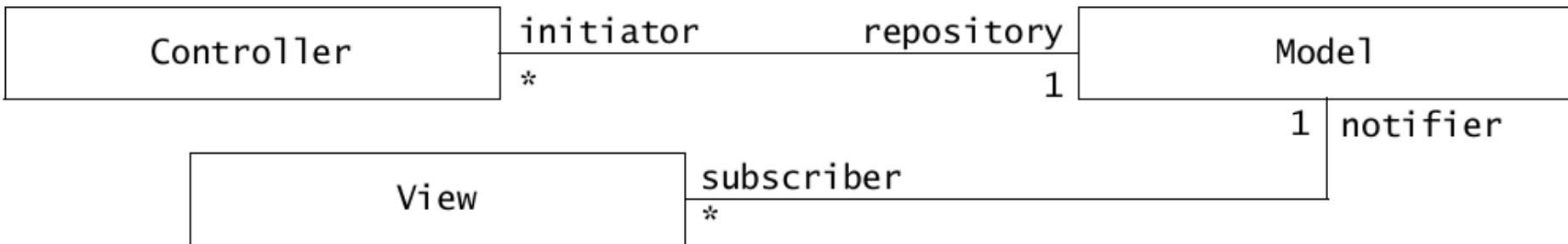
Wzorce architektoniczne – peer-to-peer

- funkcjonowanie podsystemów niezależne
- z wyjątkiem synchronizacji dla obsługi żądań
- *uogólnienie klient-serwer*



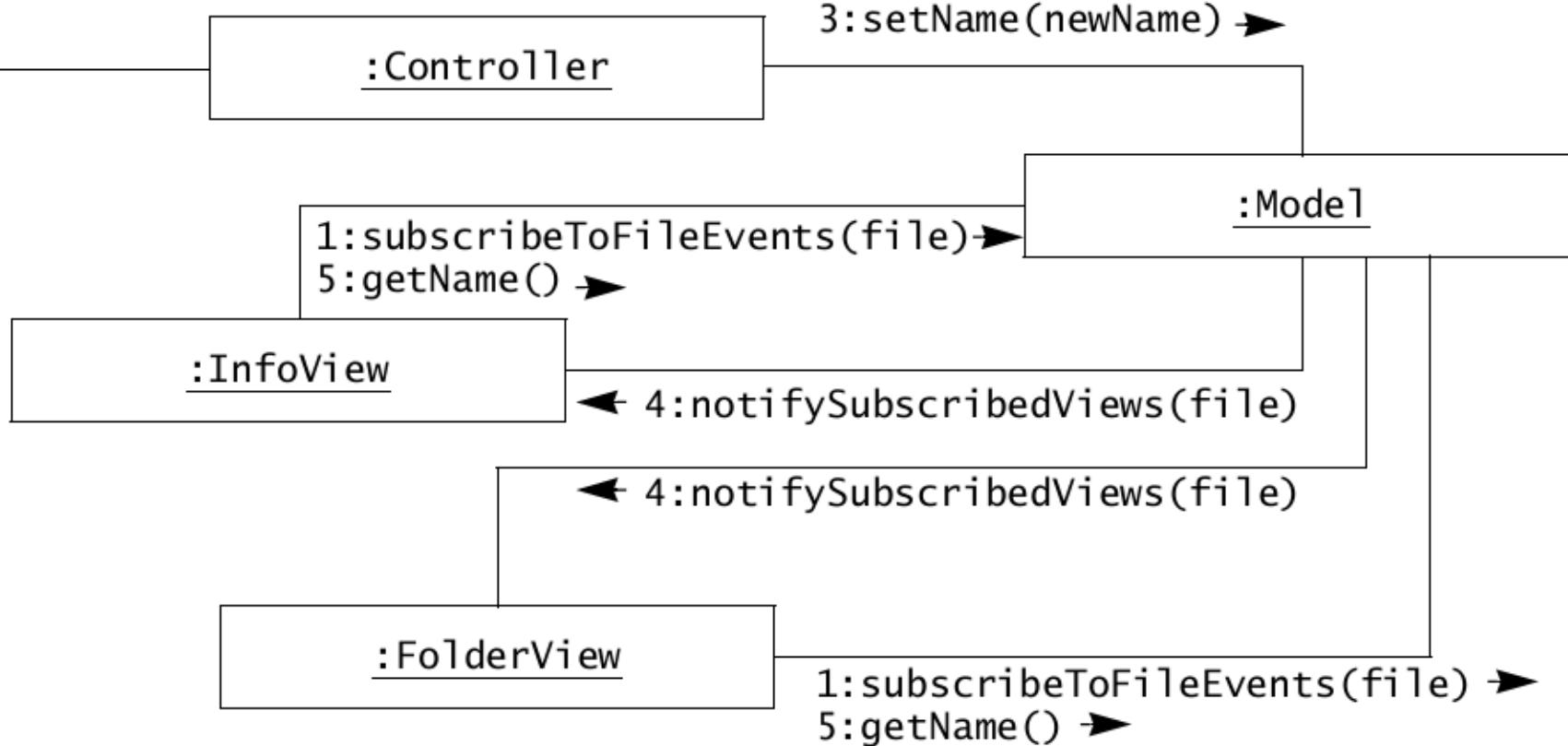
Wzorce architektoniczne – MVC

- **model** – wiedza dziedziny aplikacyjnej
 - **widok** – prezentacja informacji użytkownikowi
 - **kontroler** – zarządzanie interakcją z użytkownikiem
-
- model jest niezależny od pozostałych
 - widok i kontroler bez powiązania?
 - model – nie tylko model danych



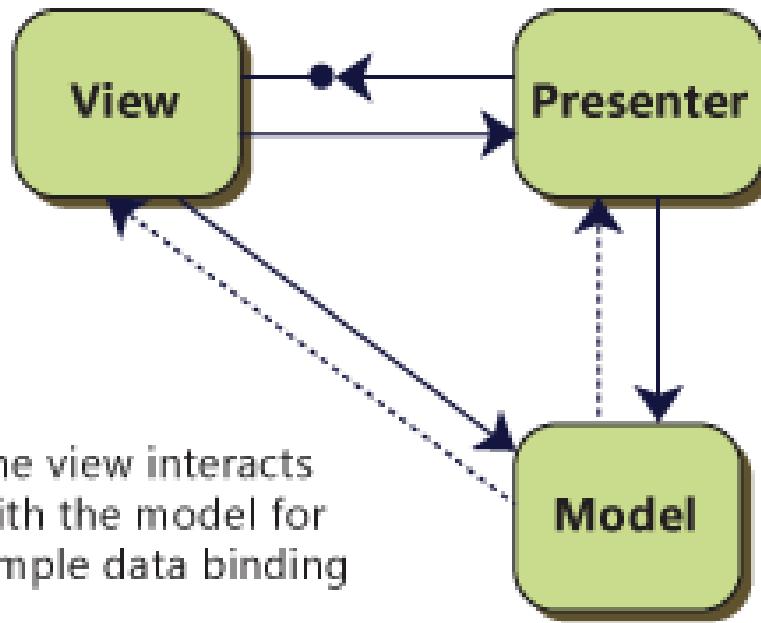
Wzorce architektoniczne – MVC – przykład

2:enterNewFileName(file, newName) ➔



Wzorce architektoniczne – MVP

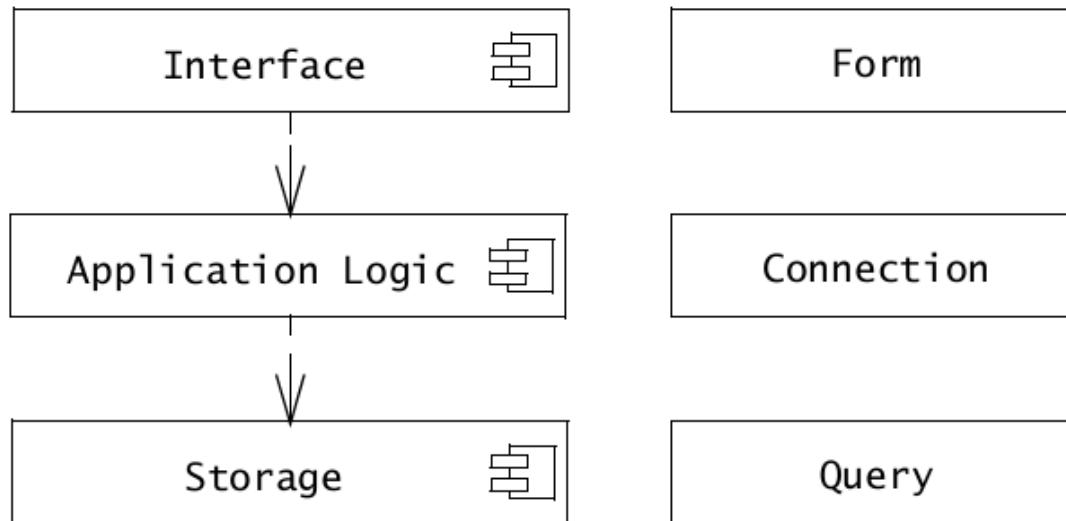
Model-View-Presenter



- The view interacts with the model for simple data binding
- The view is updated by the presenter and through data binding

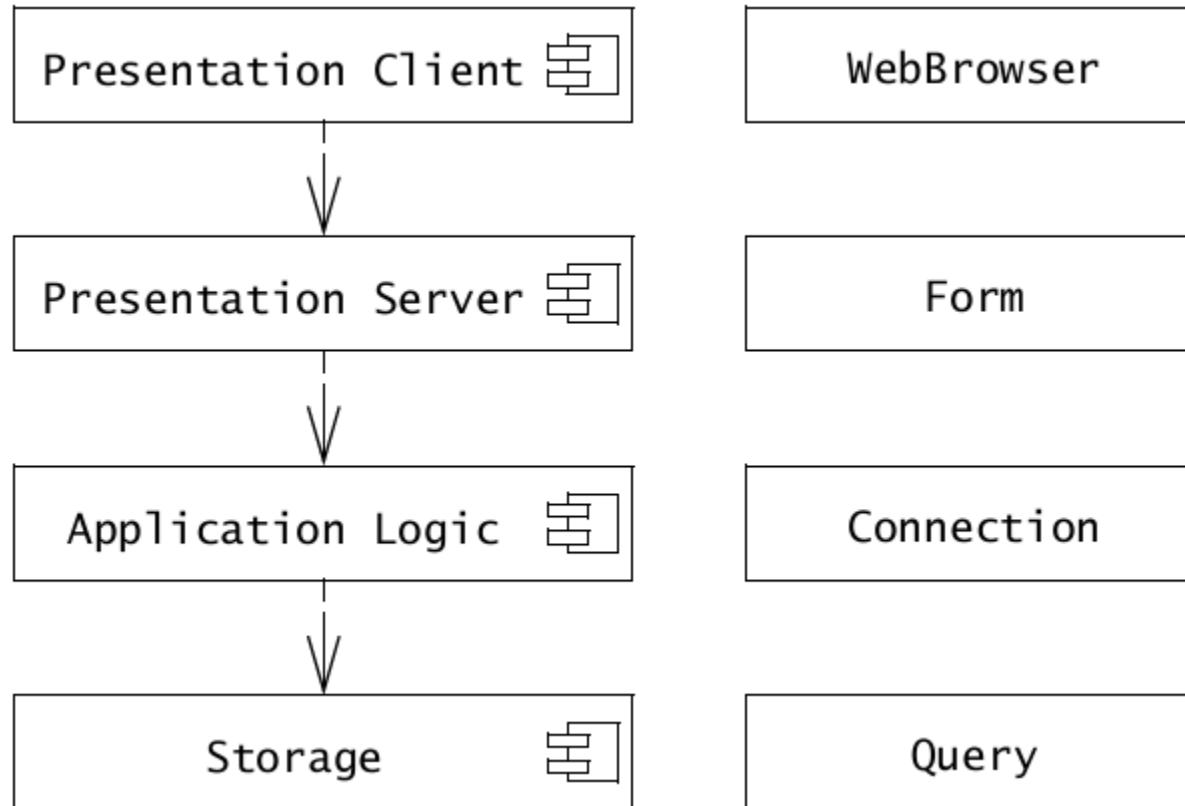
Wzorce architektoniczne – arch. 3-warstwowa

- **interfejs** – wszystkie obiekty interakcji z użytkownikiem
- **logika aplikacji** – wszystkie obiekty przetwarzania danych, sprawdzania reguł, obsługi zdarzeń/powiadomień
- **magazynowanie danych** – przechowywanie, odczyt i modyfikacja obiektów trwałych



Wzorce architektoniczne – arch. 4-warstwowa

- rozdzielona warstwa prezentacji

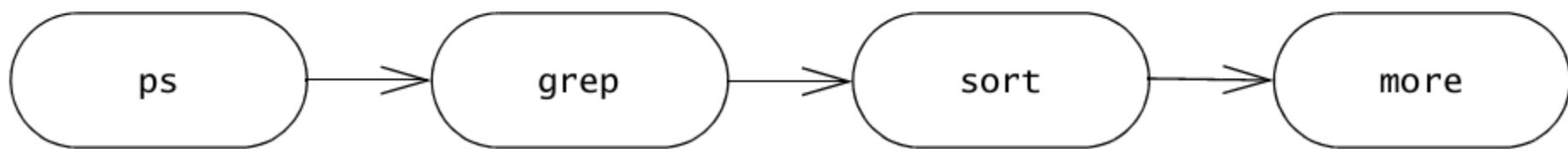


Wzorce architektoniczne – filtry i potoki

- **filtr** – przetwarza otrzymane dane i wysyła na wyjście wyniki
- **potok** – połączenie między podsystemami
- filtr zna tylko dane z jego potoku wejściowego
– nie zna źródła tych danych

```
% ps auxwww | grep dutoit | sort | more
```

dutoit	19737	0.2	1.6	1908	1500	pts/6	0	15:24:36	0:00	-tcsh
dutoit	19858	0.2	0.7	816	580	pts/6	S	15:38:46	0:00	grep dutoit
dutoit	19859	0.2	0.6	812	540	pts/6	0	15:38:47	0:00	sort



Czynności projektowania systemu

Przykład

- System MyTrip do planowania podróży dla kierowców
- w skrócie
 - kierowca planuje podróż na komputerze
 - pokładowy asystent kierowcy informuje o przebiegu trasy
- model analityczny – *wejście do projektowania*
- cele projektowe
- dekompozycja systemu

Przykład

Use case name: PlanTrip

Flow of events

1. The Driver activates her computer and logs into the trip-planning Web service.
2. The Driver enters constraints for a trip as a sequence of destinations.
3. Based on a database of maps, the planning service computes the shortest way of visiting the destinations in the order specified. The result is a sequence of segments binding a series of crossings and a list of directions.
4. The Driver can revise the trip by adding or removing destinations.
5. The Driver saves the planned trip by name in the planning service database for later retrieval.

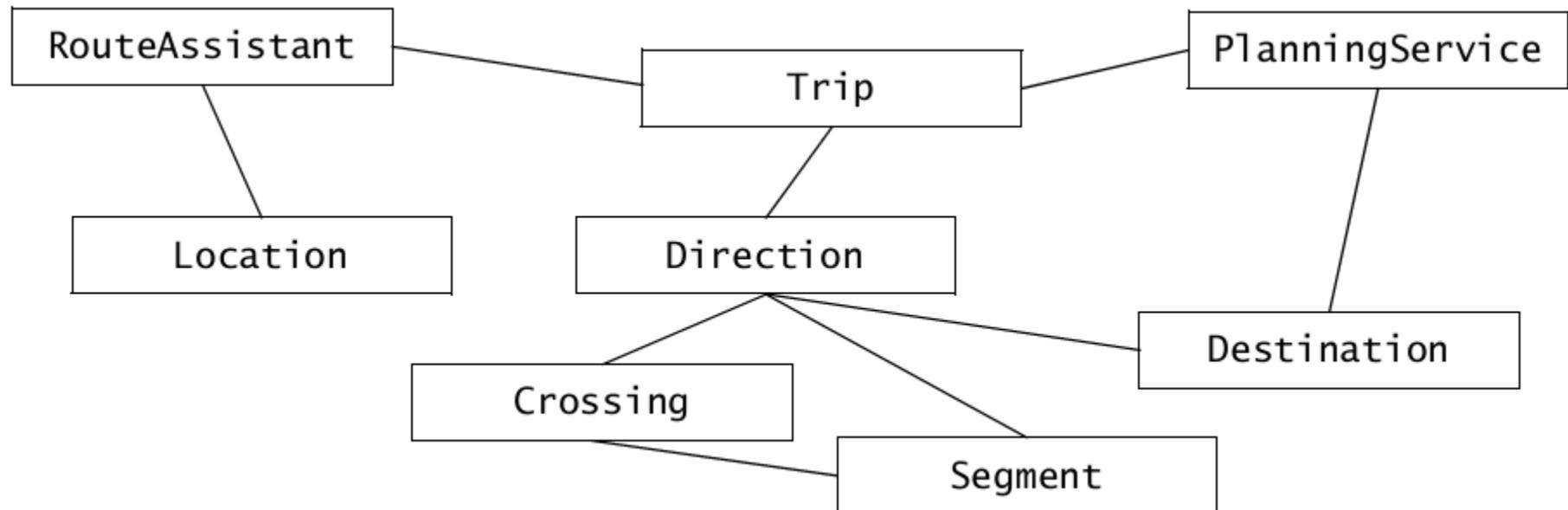
Przykład

Use case name: Execute Trip

Flow of events

1. The Driver starts her car and logs into the onboard route assistant.
2. Upon successful login, the Driver specifies the planning service and the name of the trip to be executed.
3. The onboard route assistant obtains the list of destinations, directions, segments, and crossings from the planning service.
4. Given the current position, the route assistant provides the driver with the next set of directions.
5. The Driver arrives to destination and shuts down the route assistant.

Przykład – model analityczny



Przykład – model analityczny

Crossing	A Crossing is a geographical point where several Segments meet.
Destination	A Destination represents a location where the driver wishes to go.
Direction	Given a Crossing and an adjacent Segment, a Direction describes in natural language how to steer the car onto the given Segment.
Location	A Location is the position of the car as known by the onboard GPS system or the number of turns of the wheels.
PlanningService	A PlanningService is a Web server that can supply a trip, linking a number of destinations in the form of a sequence of Crossings and Segments.
RouteAssistant	A RouteAssistant gives Directions to the driver, given the current Location and upcoming Crossing.
Segment	A Segment represents the road between two Crossings.
Trip	A Trip is a sequence of Directions between two Destinations.

Przykład – wymagania niefunkcjonalne

- Kontakt z serwerem (*Planning Service*) powinien odbywać się przez modem bezprzewodowy. Zakładamy, że jakość łączności jest zadowalająca w początkowej lokalizacji podróży.
- Gdy podróż zostanie rozpoczęta, MyTrip powinien wskazywać właściwe kierunki nawet w przypadku braku połączenia z planistą (*PlanningService*).
- Łączny czas połączenia z planistą powinien być jak najmniejszy, ze względu na koszty.
- Zmiana planu podróży powinna być możliwa w warunkach połączenia z planistą.
- Planista powinien być zdolny do równoczesnej obsługi przynajmniej 50 klientów i 1000 tras.

Przykład – identyfikacja celów projektowych

- **Niezawodność** – MyTrip powinien funkcjonować niezawodnie
- **Odporność na awarie** – MyTrip powinien tolerować utratę połączenia z serwerem
- **Zabezpieczenia** – MyTrip powinien być bezpieczny, tj. nie powinien udostępniać danych użytkownika innym użytkownikom, nie powinien też zezwalać na nieautoryzowany dostęp
- **Modyfikowalność** – MyTrip powinien być modyfikowalny, czyli musi umożliwiać wybór między różnymi usługami planowania podróży

Identyfikacja podsystemów – heurystyki

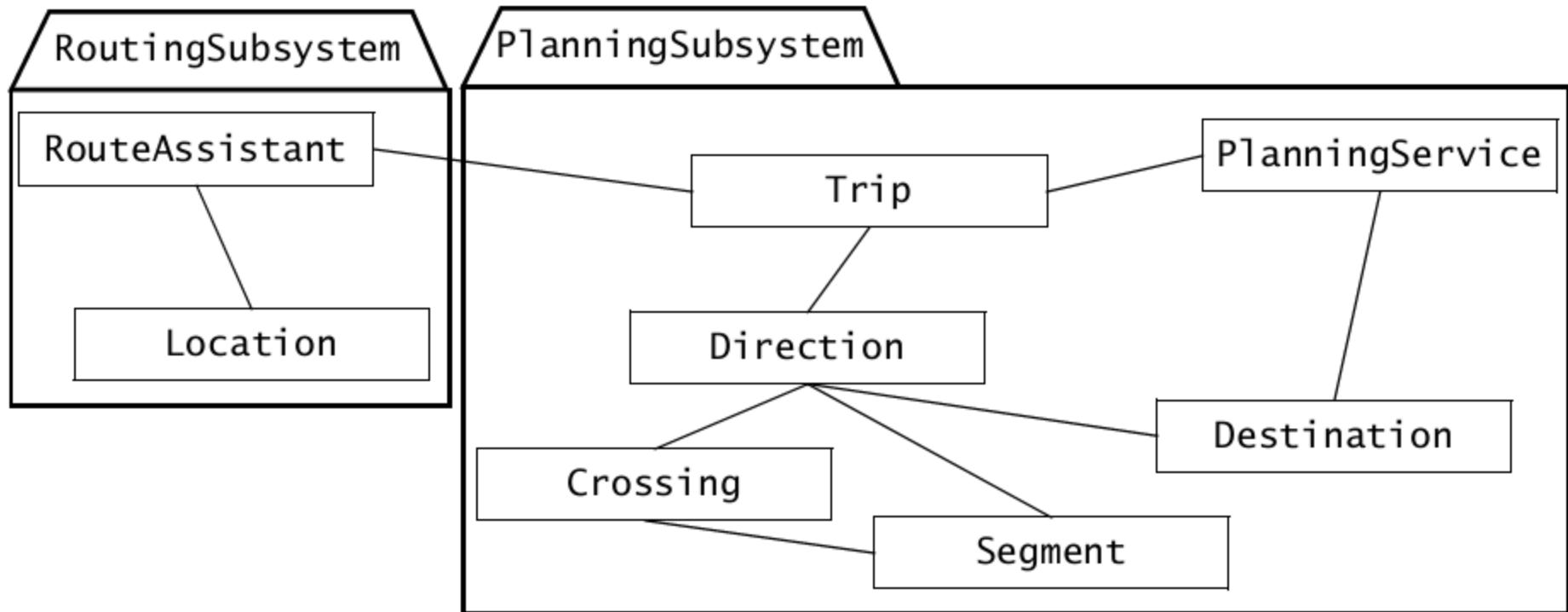
obiekty uczestniczące w tym samym przypadku
użycia → jeden podsystem

obiekty wykorzystywane do przesyłania danych
między podsystemami → oddzielne podsystemy

minimalizacja liczby powiązań między
podsystemami

obiekty podsystemu powinny być funkcjonalnie
powiązane

Przykład – identyfikacja podsystemów



Szczegółowe czynności projektowania

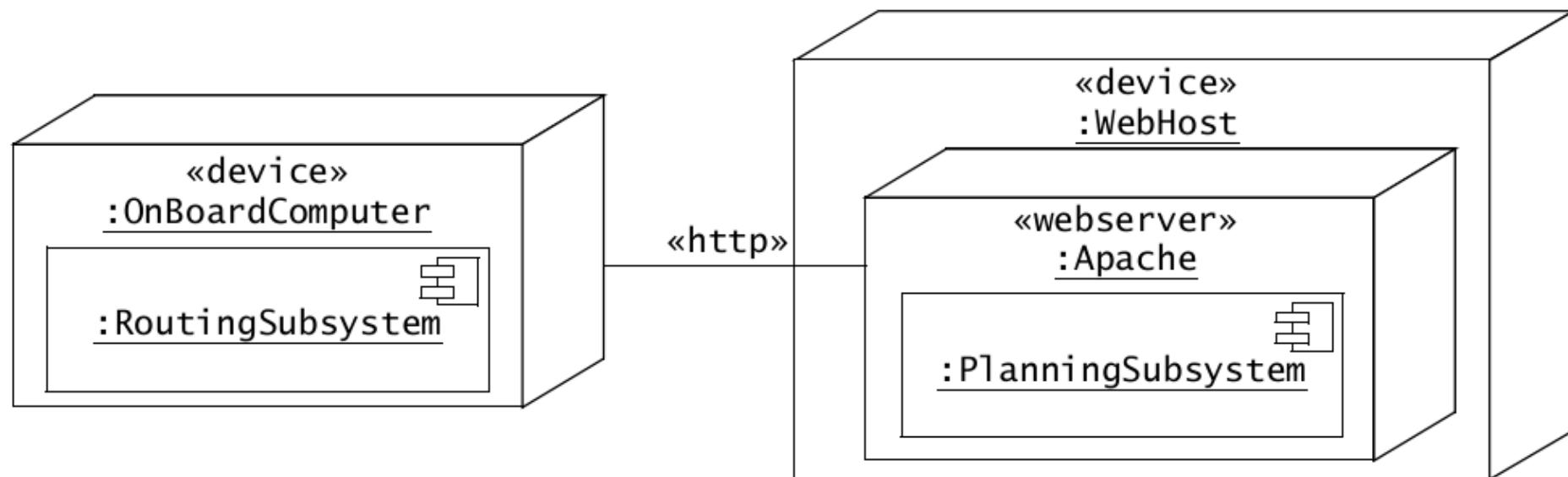
Definiowanie celów projektowych

Definiowanie podsystemów

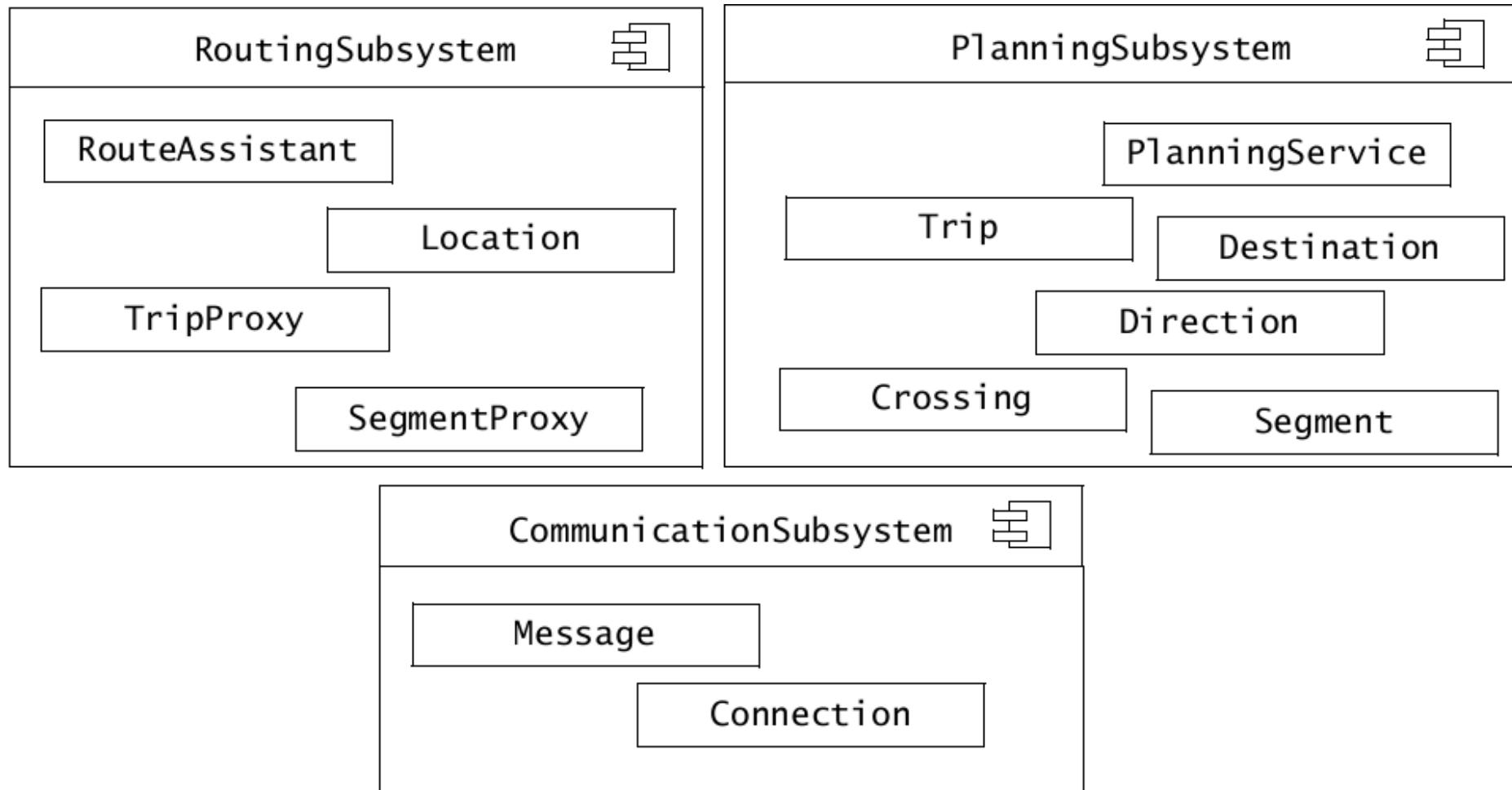
Implementacja podsystemów

- odwzorowanie podsystemów w platformę sprzętową i programową
- zarządzanie trwałymi danymi
- definiowanie założeń kontroli dostępu
- wybór globalnej struktury przepływu sterowania
- identyfikacja usług
- opisywanie warunków granicznych

Przykład – przypisanie systemów do urządzeń

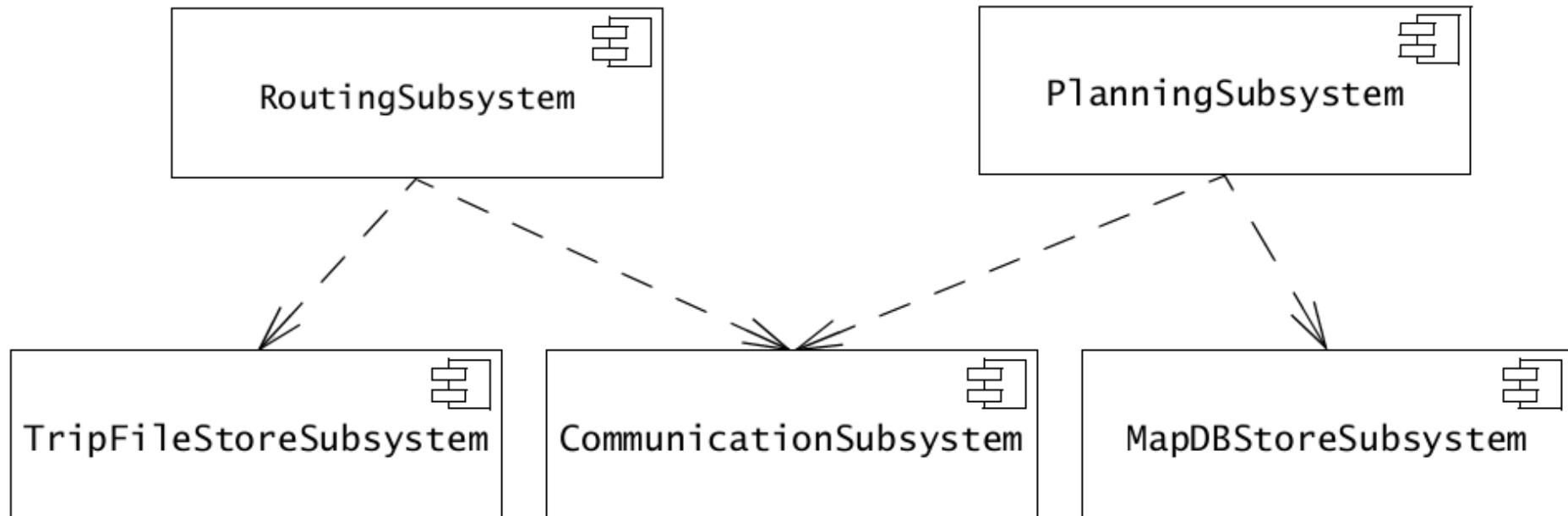


Przykład – przydział obiektów do węzłów



Przykład – obsługa trwałych danych

Czy przechowywać dane w jednym miejscu?



Przykład – obiekty trwałe i strategia

**które obiekty muszą przetrwać
zakończenie pracy systemu?**

- Trip – Crossing, Destination, PlanningService, Segment
- nie: Direction, Location
 - wyliczane automatycznie wraz z przemieszczaniem się samochodu

jaka strategia przechowywania danych?

- niezależne pliki
- relacyjna baza danych
- obiektowa baza danych
- inne bazy danych – NoSQL: <http://en.wikipedia.org/wiki/NoSQL>
- ...

Przykład – definiowanie założeń dostępu

Communication Subsystem

The CommunicationSubsystem is responsible for transporting Trips from the PlanningSubsystem to the RoutingSubsystem. The CommunicationSubsystem uses the Driver associated with the Trip being transported for selecting a key and encrypting the communication traffic.

Planning Subsystem

The PlanningSubsystem is responsible for constructing a Trip connecting a sequence of Destinations. The PlanningSubsystem is also responsible for responding to replan requests from RoutingSubsystem. Prior to processing any requests, the PlanningSubsystem authenticates the Driver from the RoutingSubsystem. The authenticated Driver is used to determine which Trips can be sent to the corresponding RoutingSubsystem.

Driver

A Driver represents an authenticated user. It is used by the CommunicationSubsystem to remember keys associated with a user and by the PlanningSubsystem to associate Trips with users.

Inny przykład – bank

macierz kontroli dostępu

Objects Actors	Corporation	LocalBranch	Account
Teller			postSmallDebit() postSmallCredit() examineBalance()
Manager		examineBranchStats()	postSmallDebit() postSmallCredit() postLargeDebit() postLargeCredit() examineBalance() examineHistory()
Analyst	examineGlobalStats()	examineBranchStats()	

implementacje

- globalna tabela dostępu: (aktor, klasa, operacja)
- lista kontroli dostępu: dla klasy (aktor, operacja)
- lista uprawnień: dla aktora (klasa, operacja)

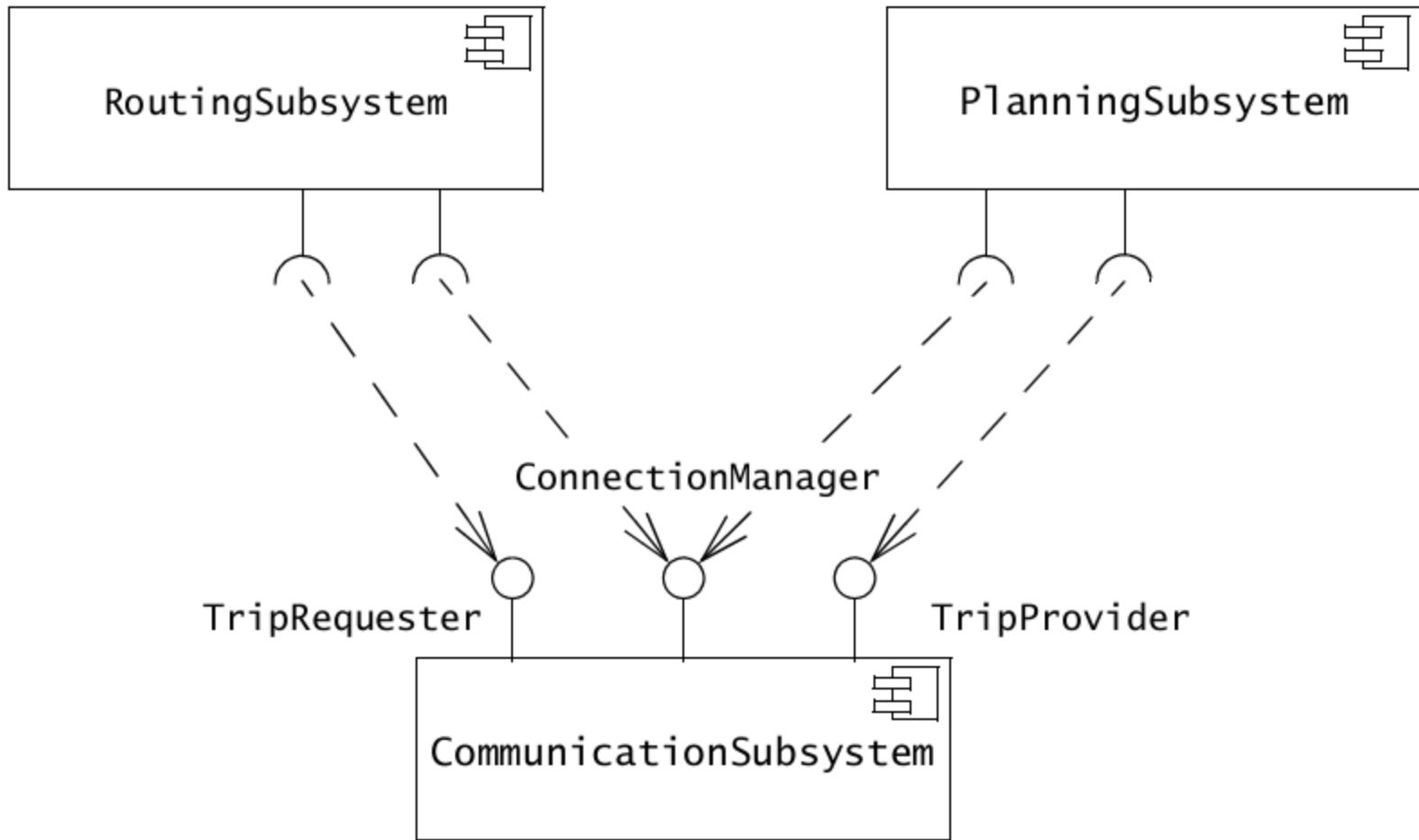
Projekt globalnego przepływu sterowania

sterowanie proceduralne

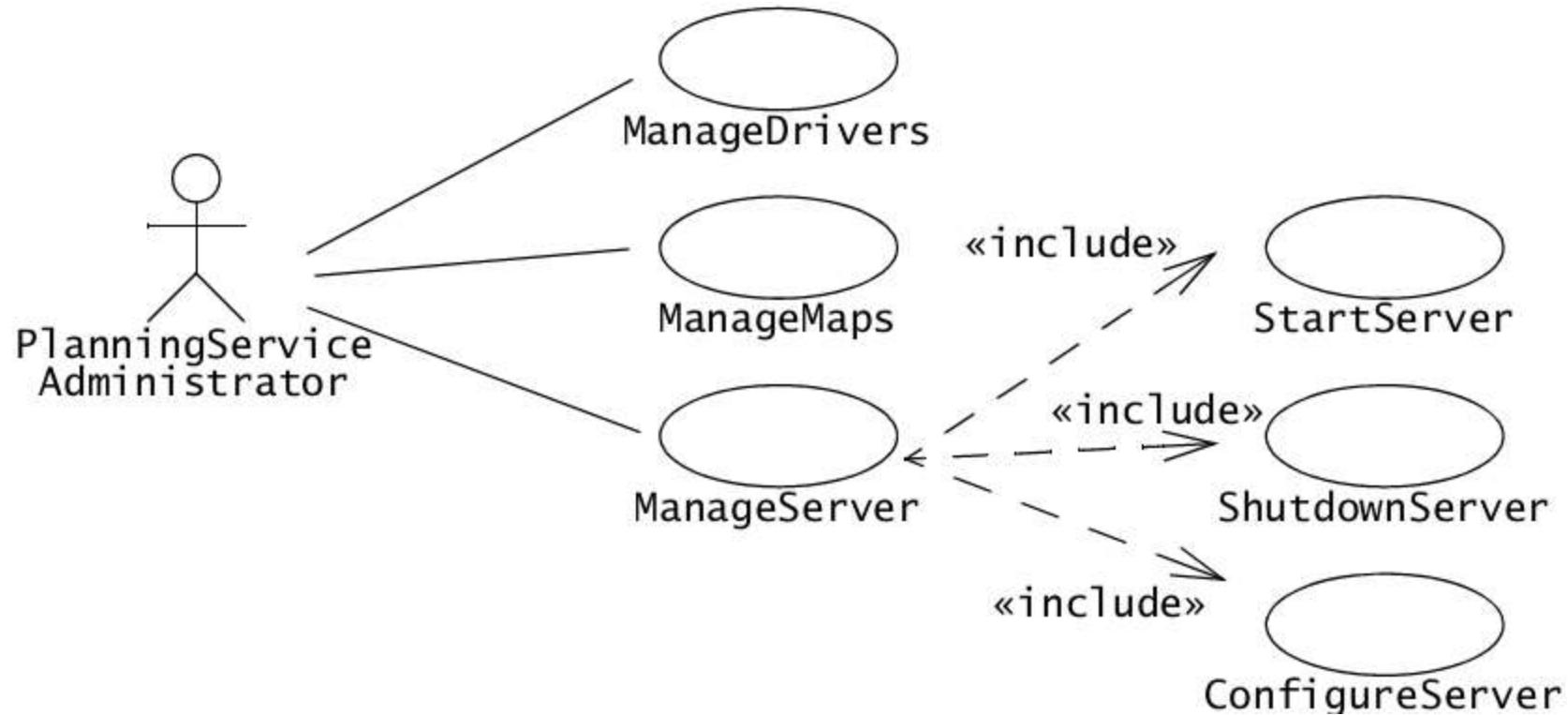
sterowanie zdarzeniowe

sterowanie wielowątkowe

Przykład – identyfikacja usług



Przykład – identyfikacja warunków granicznych



Podsumowanie

podstawowe koncepcje
projektowania oprogramowania

Pytania



Literatura

- **Bruegge B., Dutoit A. H, Inżynieria oprogramowania w ujęciu obiektowym. UML, wzorce projektowe i Java, Helion 2011**
- Sacha K., Inżynieria oprogramowania, PWN 2010
- Górska J. (red.), Inżynieria oprogramowania w projekcie informatycznym, Mikom 2000

Następny wykład

Projektowanie architektury systemu wzorce projektowe

Inżynieria oprogramowania

Wykład 6: Projektowanie architektury systemu. Wzorce projektowe

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Agenda

- UML → kod źródłowy
- Wzorce projektowe
 - motywacja
 - wprowadzenie
 - przegląd
 - korzyści ze stosowania wzorców
 - proces stosowania wzorców

UML → kod źródłowy

[DEV475]

Notacja

Course



```
public class Course
{
    Course() {}
    protected void finalize()
        throws Throwable {
            super.finalize();
        }
};
```

Course



```
class Course
{
public:
    Course();
    ~Course();
};
```

Widoczność atrybutów i operacji

Student

- name : String

+ addSchedule (theSchedule: Schedule, forSemester: Semester)
+ hasPrerequisites(forCourseOffering: CourseOffering) : boolean
passed(theCourseOffering: CourseOffering) : boolean

```
public class Student
{
    private String name;

    public void addSchedule (Schedule theSchedule; Semester forSemester) {
    }

    public boolean hasPrerequisites(CourseOffering forCourseOffering) {
    }
    protected boolean passed(CourseOffering theCourseOffering) {
    }
}
```

Widoczność atrybutów i operacji

Student

- name : char*

+ addSchedule (theSchedule: Schedule, forSemester: Semester)
+ hasPrerequisites(forCourseOffering: CourseOffering) : int
passed(theCourseOffering: CourseOffering) : int

```
class Student
{
```

```
public:
    void addSchedule (theSchedule: Schedule, forSemester: Semester);
    int hasPrerequisites(forCourseOffering: CourseOffering);
```

```
protected:
    int passed(theCourseOffering: CourseOffering);
```

```
private:
    char* name;
};
```

Atrybuty i operacje klasowe (statyczne)

Student

- nextAvailID : int = 1

+ getNextAvailID() : int

```
class Student
{
    private static int nextAvailID = 1;

    public static int getNextAvailID() {
    }
}
```

Student

- nextAvailID : int = 1

+ getNextAvailID() : int

```
class Student
{
    public:
        static int getNextAvailID();

    private:
        static int nextAvailID = 1;
};
```

Klasy narzędziowe – statyczne

<<utility>>
MathPack

-randomSeed : long = 0
-pi : double = 3.14159265358979

+sin (angle : double) : double
+cos (angle : double) : double
+random() : double

```
void somefunction() {  
    ...  
    myCos = MathPack.cos(90.0);  
    ...  
}
```

```
import java.lang.Math;  
import java.util.Random;  
class MathPack  
{  
    private static randomSeed long = 0;  
    private final static double pi =  
        3.14159265358979;  
    public static double sin(double angle) {  
        return Math.sin(angle);  
    }  
    static double cos(double angle) {  
        return Math.cos(angle);  
    }  
    static double random() {  
        return new  
            Random(seed).nextDouble();  
    }  
}
```

Klasy narzędziowe – statyczne

<<utility>>
MathPack

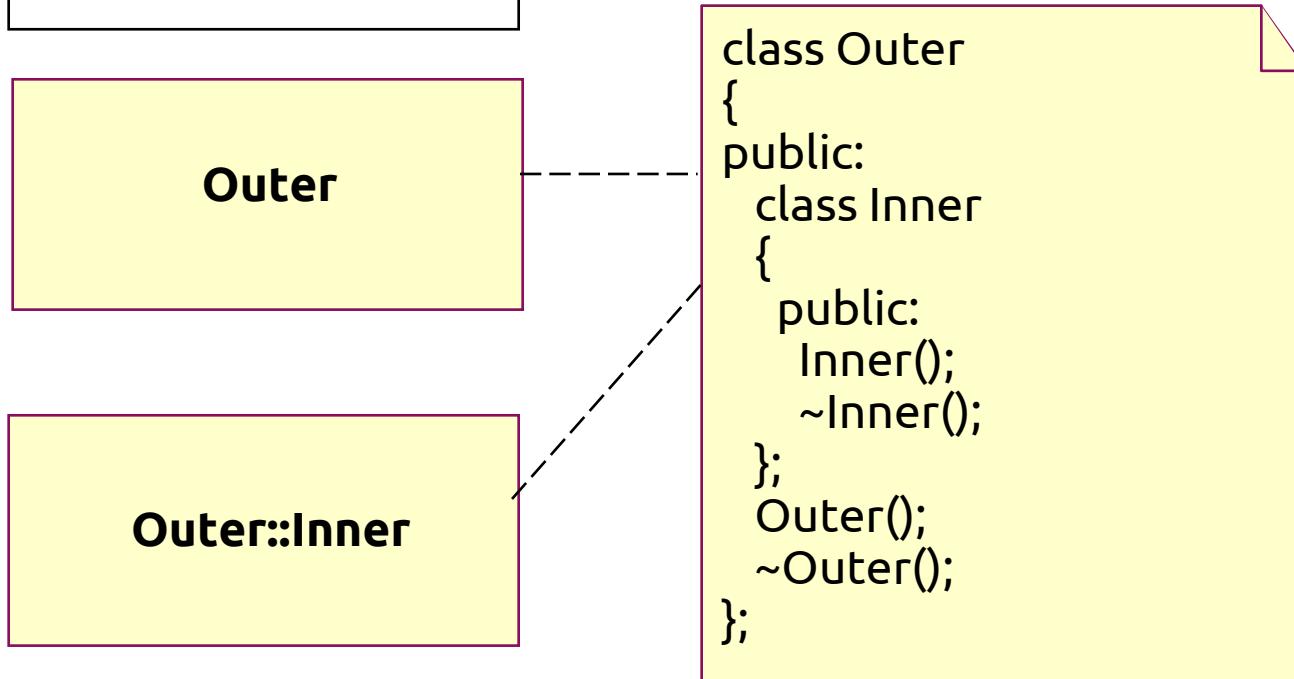
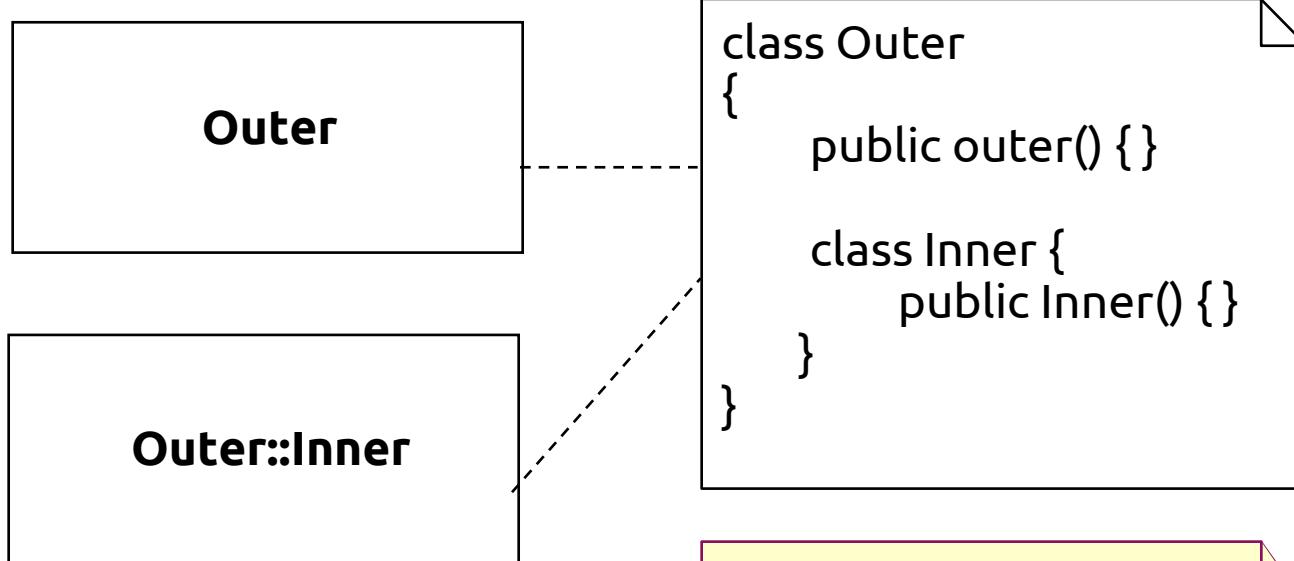
-randomSeed : long = 0
-pi : double = 3.14159265358979

+sin (angle : double) : double
+cos (angle : double) : double
+random() : double

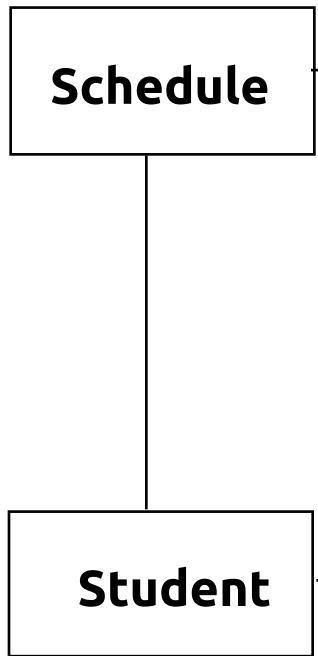
class MathPack
{
public:
 static double sin(double angle);
 static double cos(double angle);
 static double random();
private:
 static long randomSeed = 0;
 static double pi = 3.14159265358979;
};

void main(void)
...
 myCos = MathPack::cos(90.0);
...

Klasy zagnieżdżone



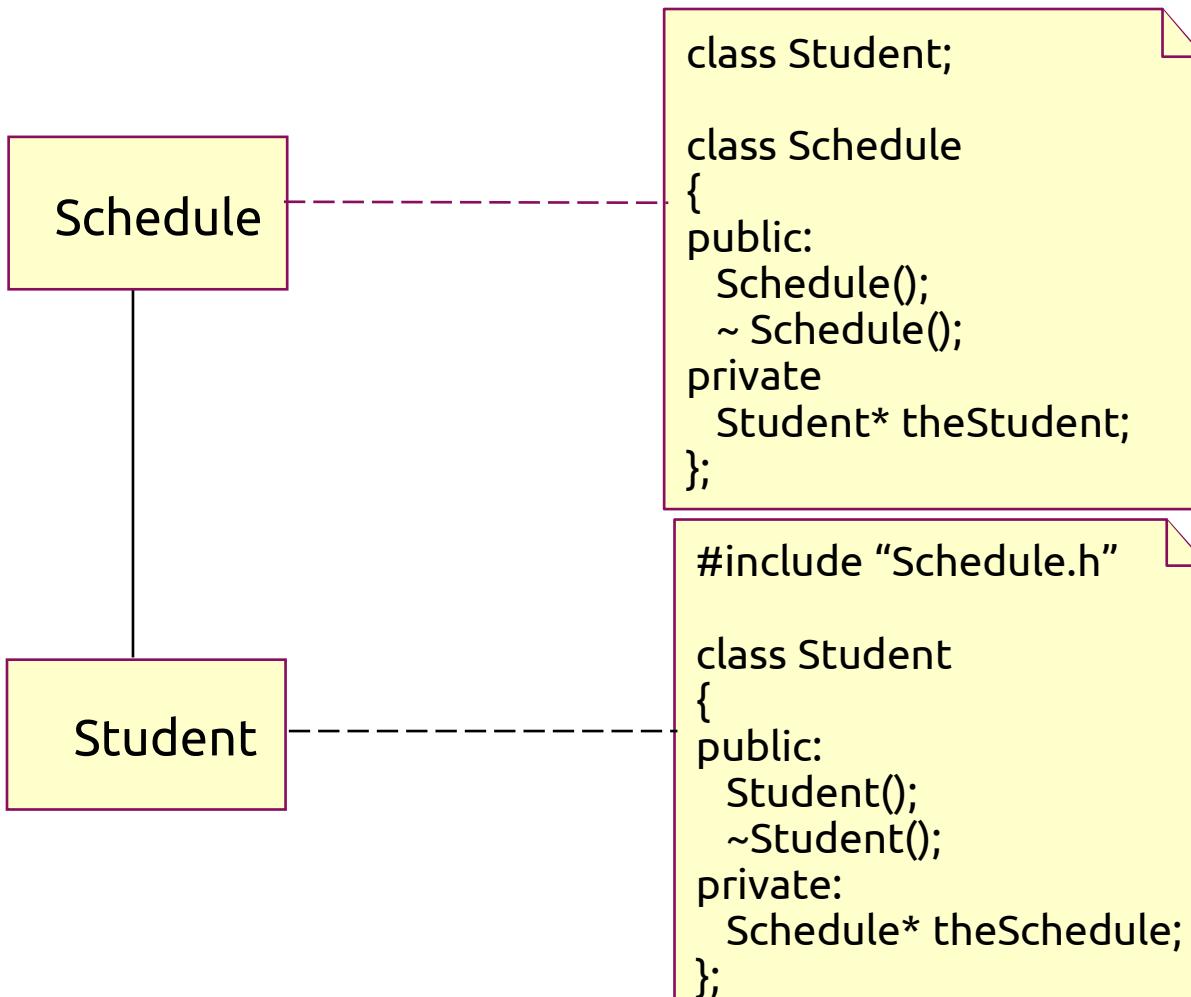
Asocjacje



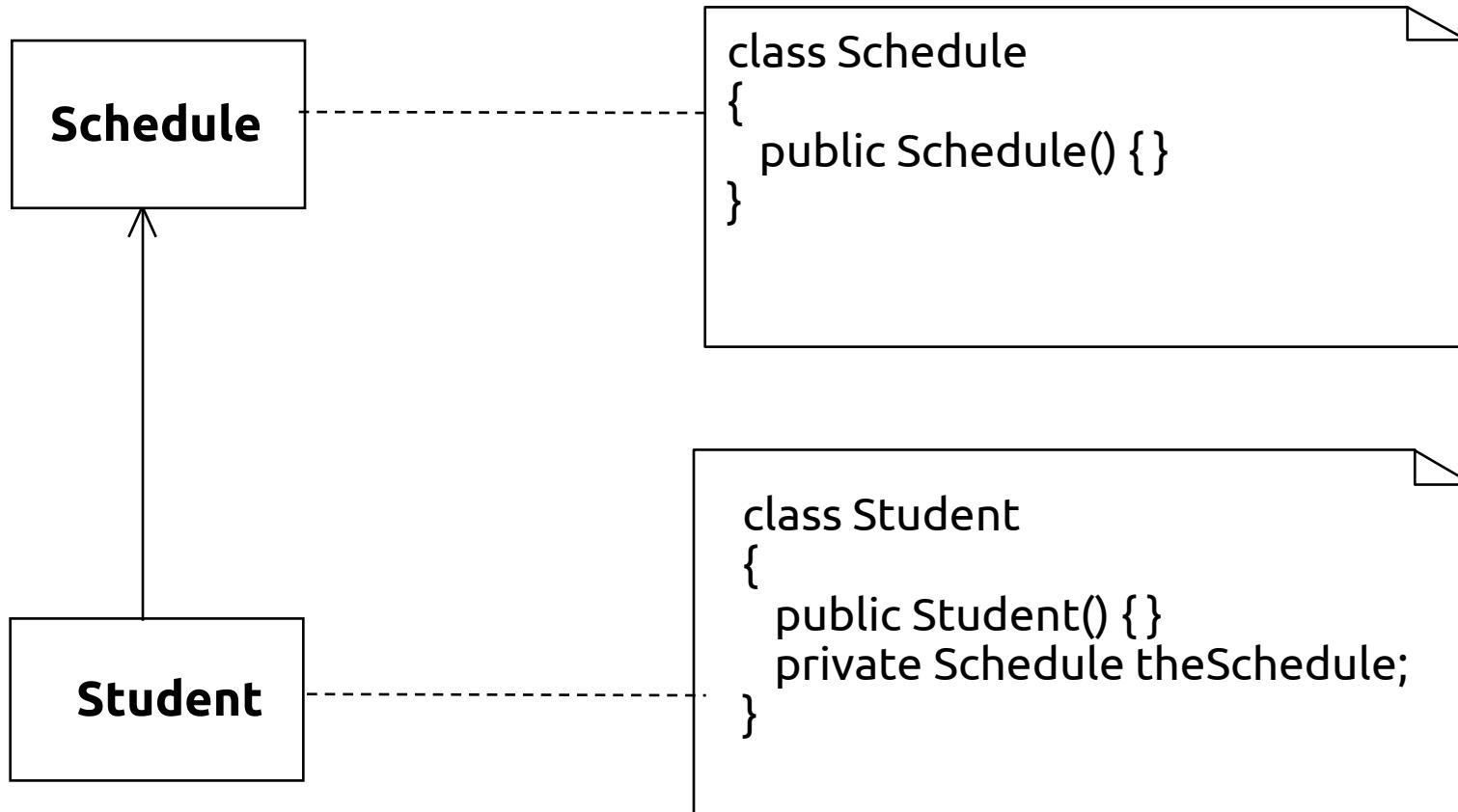
```
// niepotrzebne importowanie,  
// jeśli ten sam pakiet  
  
class Schedule  
{  
    public Schedule() {} //constructor  
    private Student theStudent;  
}
```

```
class Student  
{  
    public Student() {}  
    private Schedule theSchedule;  
}
```

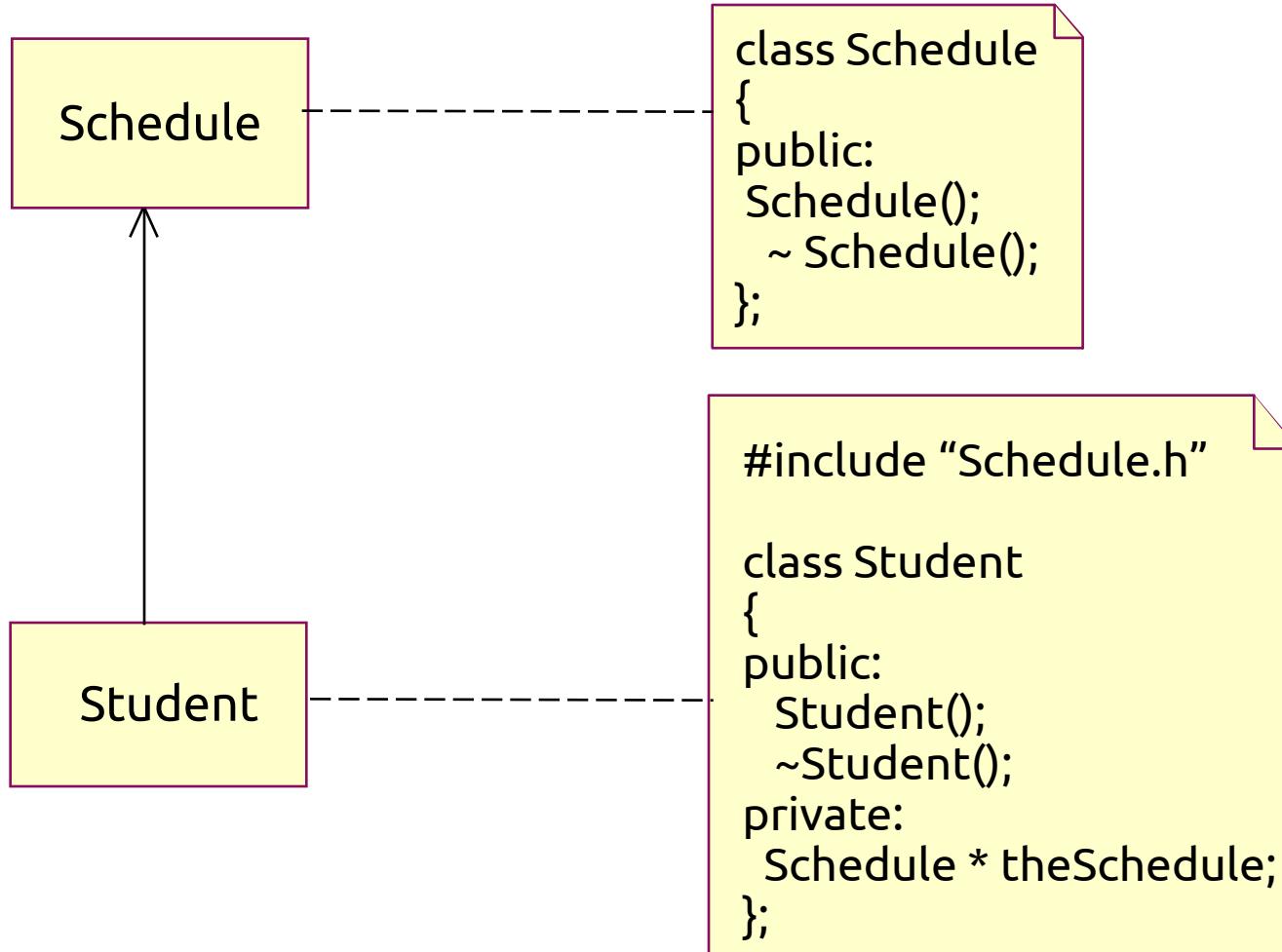
Asocjacje



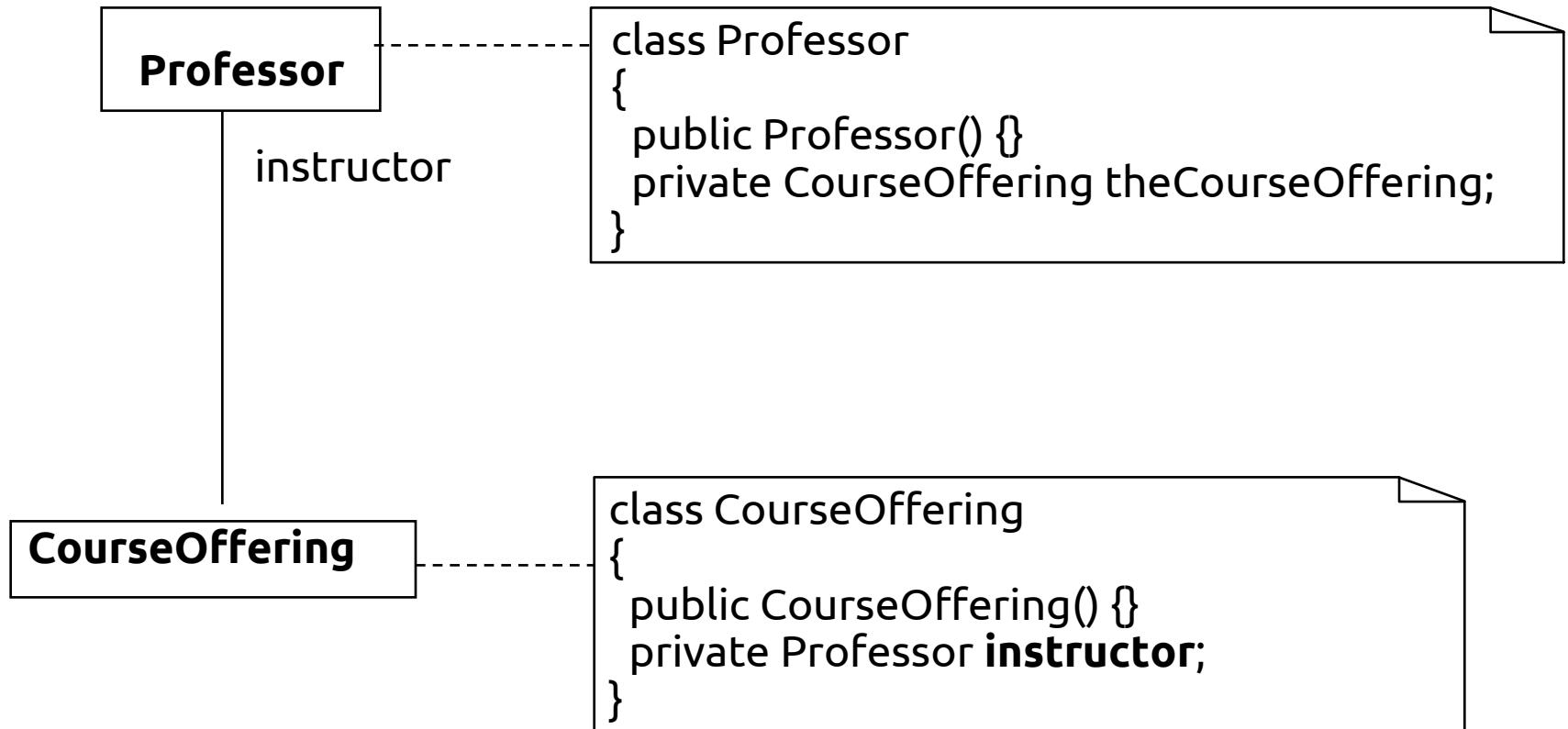
Nawigowalność asocjacji



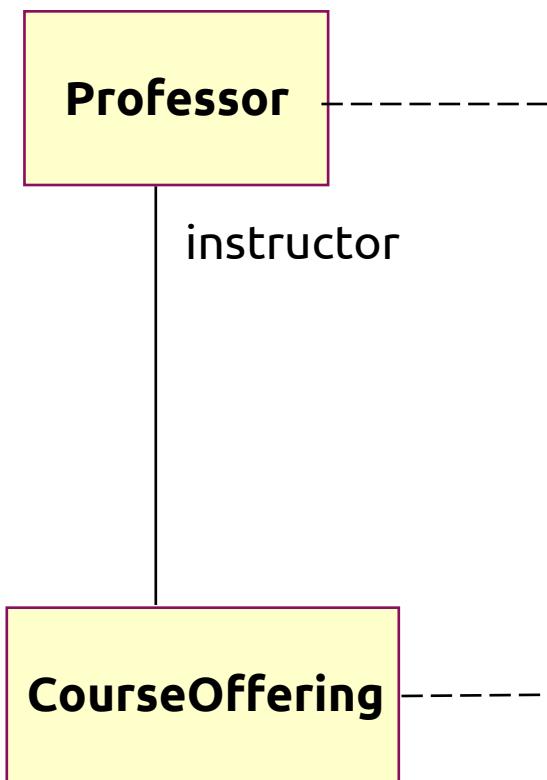
Nawigowalność asocjacji



Role w asocjacji



Role w asocjacji



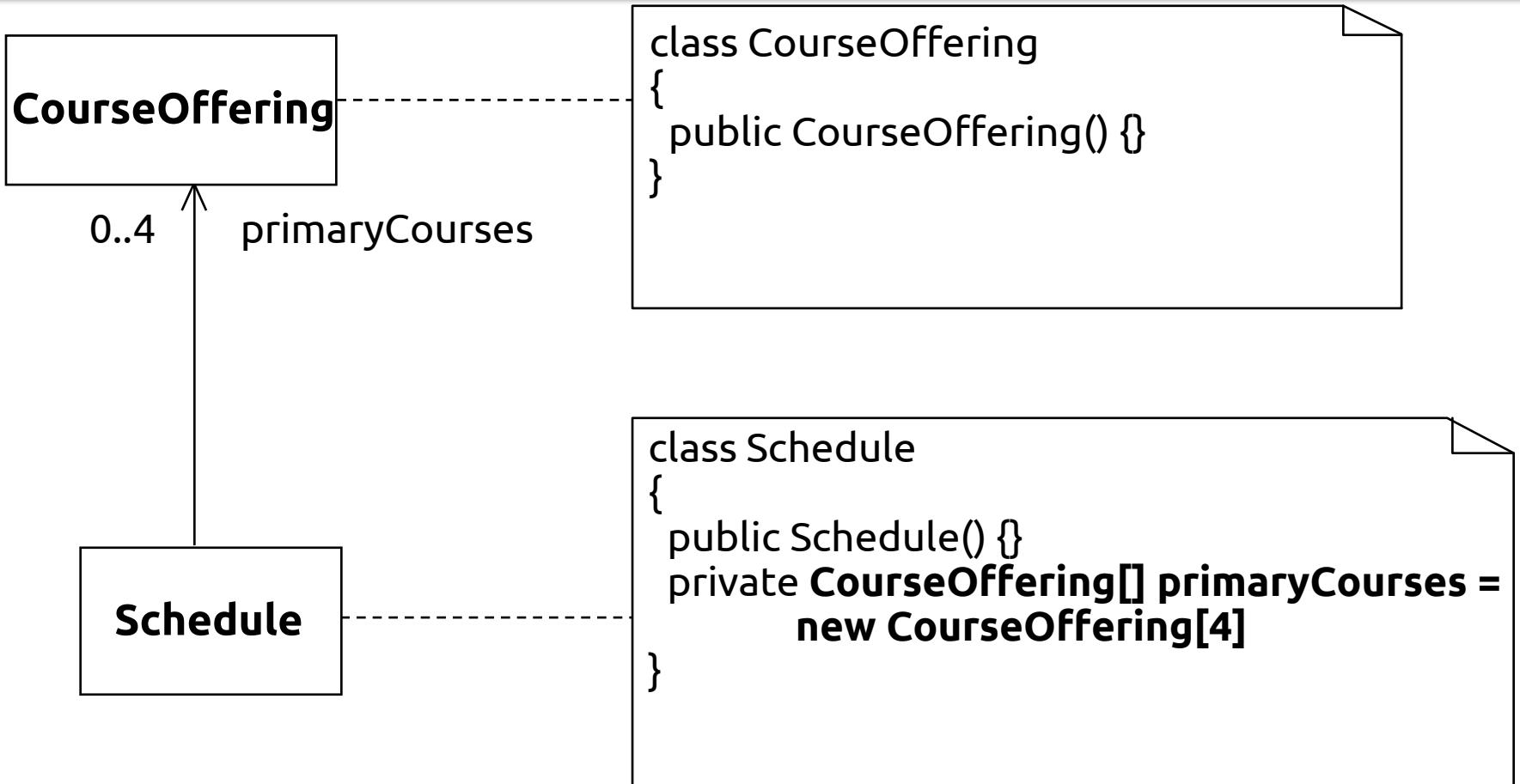
```
#include "Company.h"

class Professor
{
public:
    Professor();
    ~Professor();
private:
    CourseOffering* theCourseOffering;
};
```

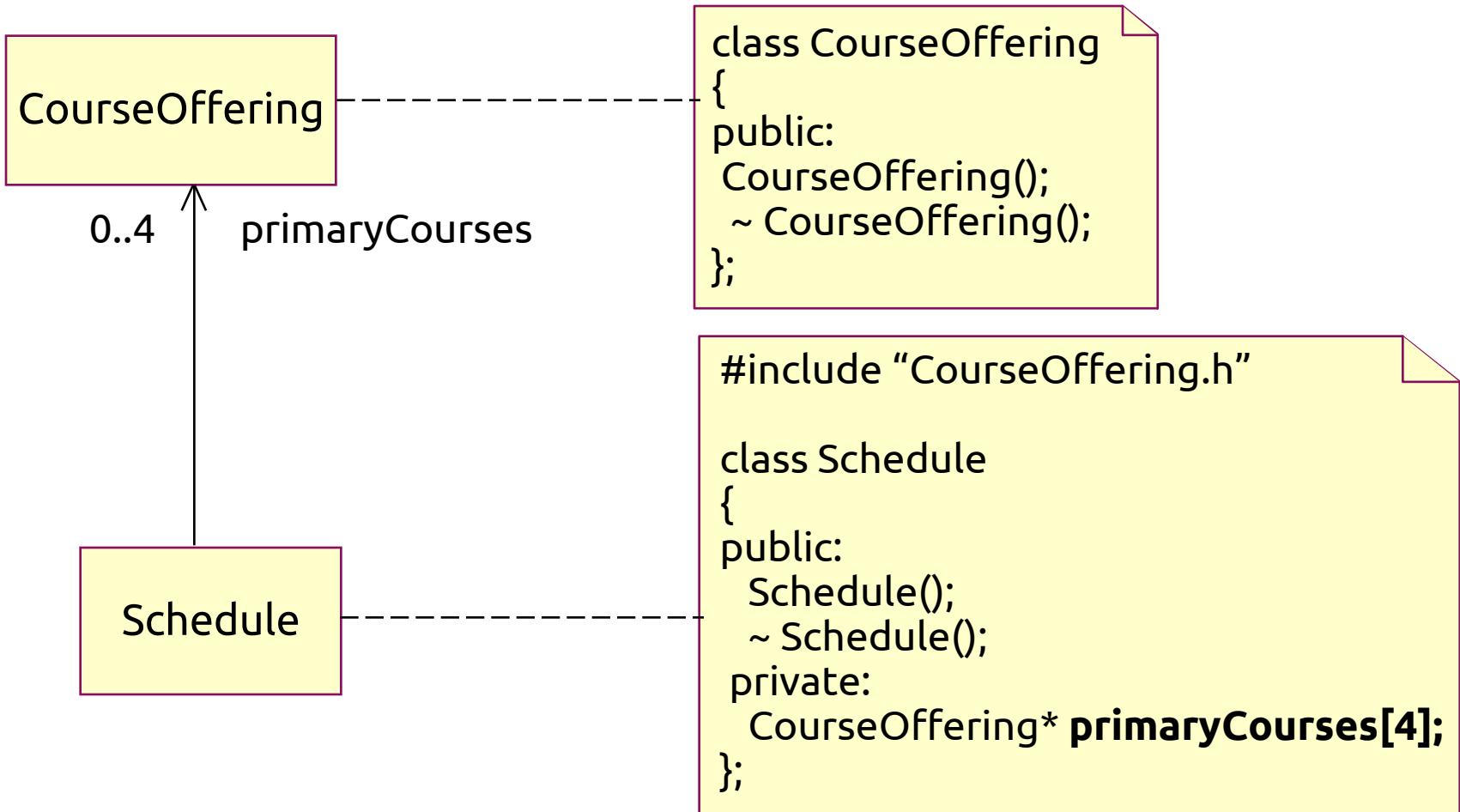
```
class Professor;

class CourseOffering
{
public:
    CourseOffering();
    ~CourseOffering();
private:
    Professor* instructor;
};
```

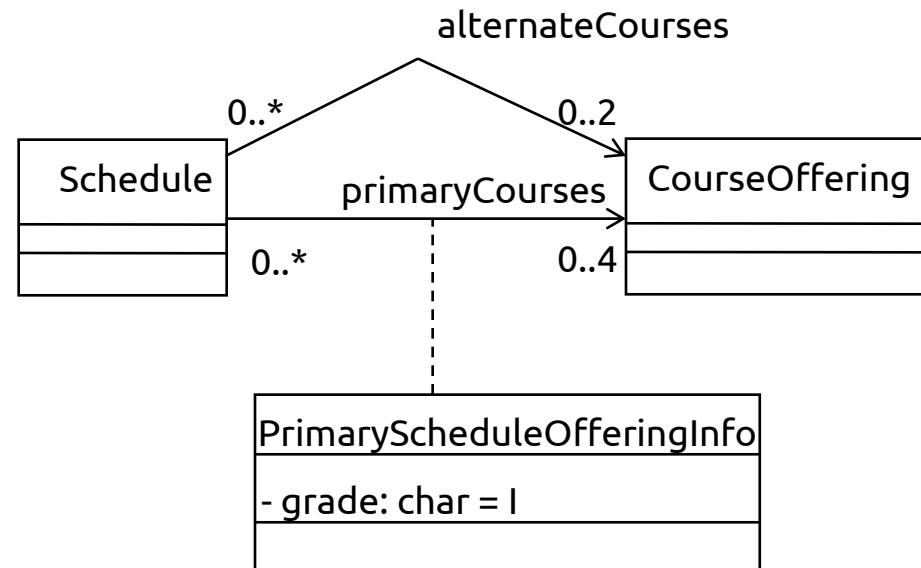
Liczliwość asocjacji



Liczliwość asocjacji



Klasa asocjacyjna

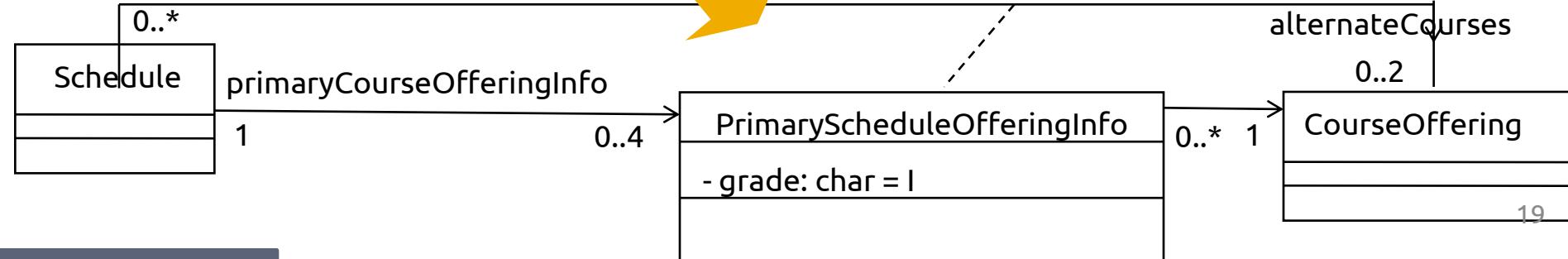


```
class PrimaryScheduleOfferingInfo
{
    public PrimaryScheduleOfferingInfo() {}

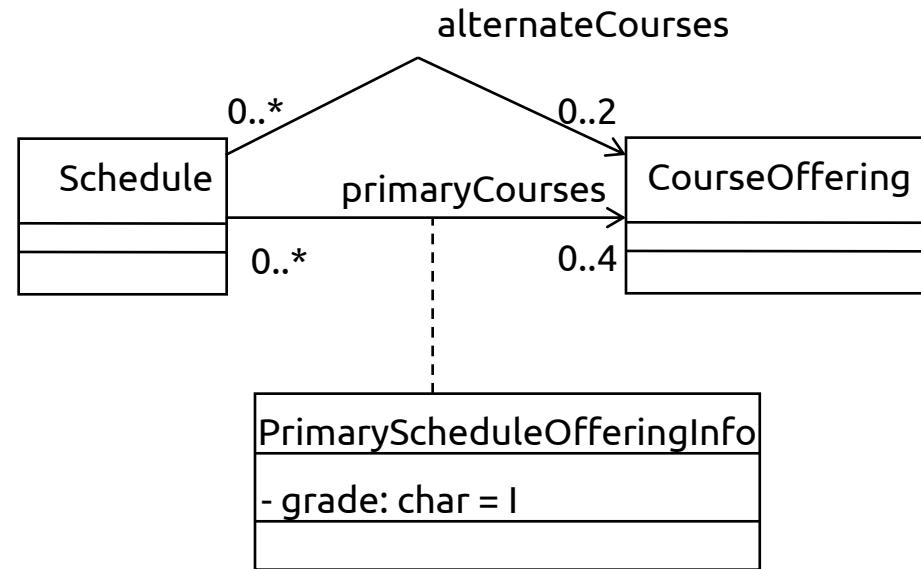
    public CourseOffering get_theCourseOffering(){
        return theCourseOffering;
    }

    public void set_theCourseOffering(CourseOffering toValue){
        theCourseOffering = toValue;
    }

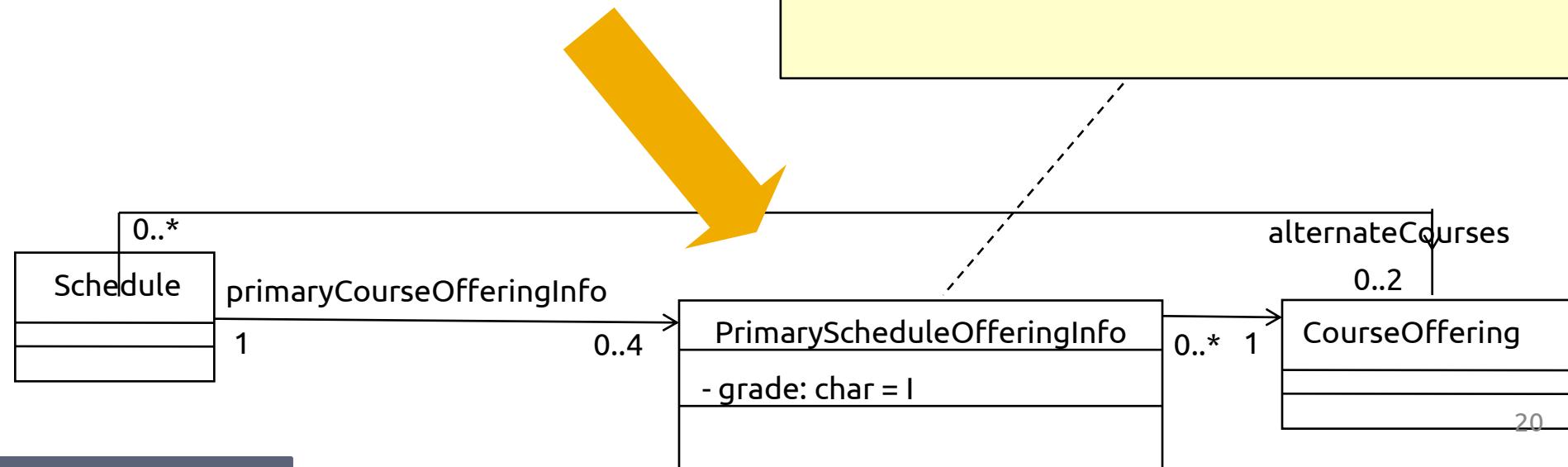
    private char get_Grade (){ return grade; }
    private void set_Grade(char toValue) { grade = toValue; }
    private char grade = 'I';
private CourseOffering theCourseOffering;
}
```



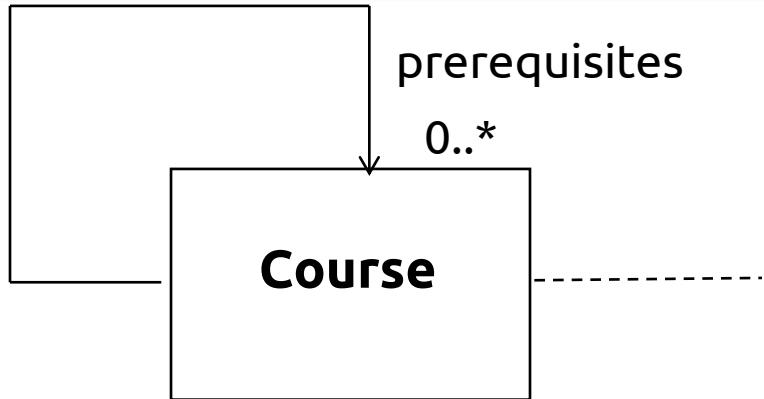
Klasa asocjacyjna



```
class CourseOffering;  
  
class PrimaryScheduleOfferingInfo  
{  
public:  
    PrimaryScheduleOfferingInfo();  
    ~ PrimaryScheduleOfferingInfo ();  
    const CourseOffering * get_the_CourseOffering() const;  
    void set_the_CourseOffering(CourseOffering *const toValue);  
private:  
    const char get_Grade() const;  
    void set_Grade(const char toValue);  
    char value = 'I';  
    CourseOffering *theCourseOffering;  
};
```

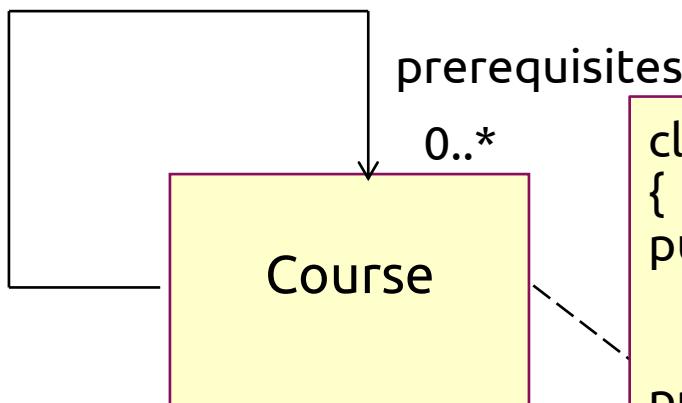


Asocjacje refleksywne (rekurencyjne)



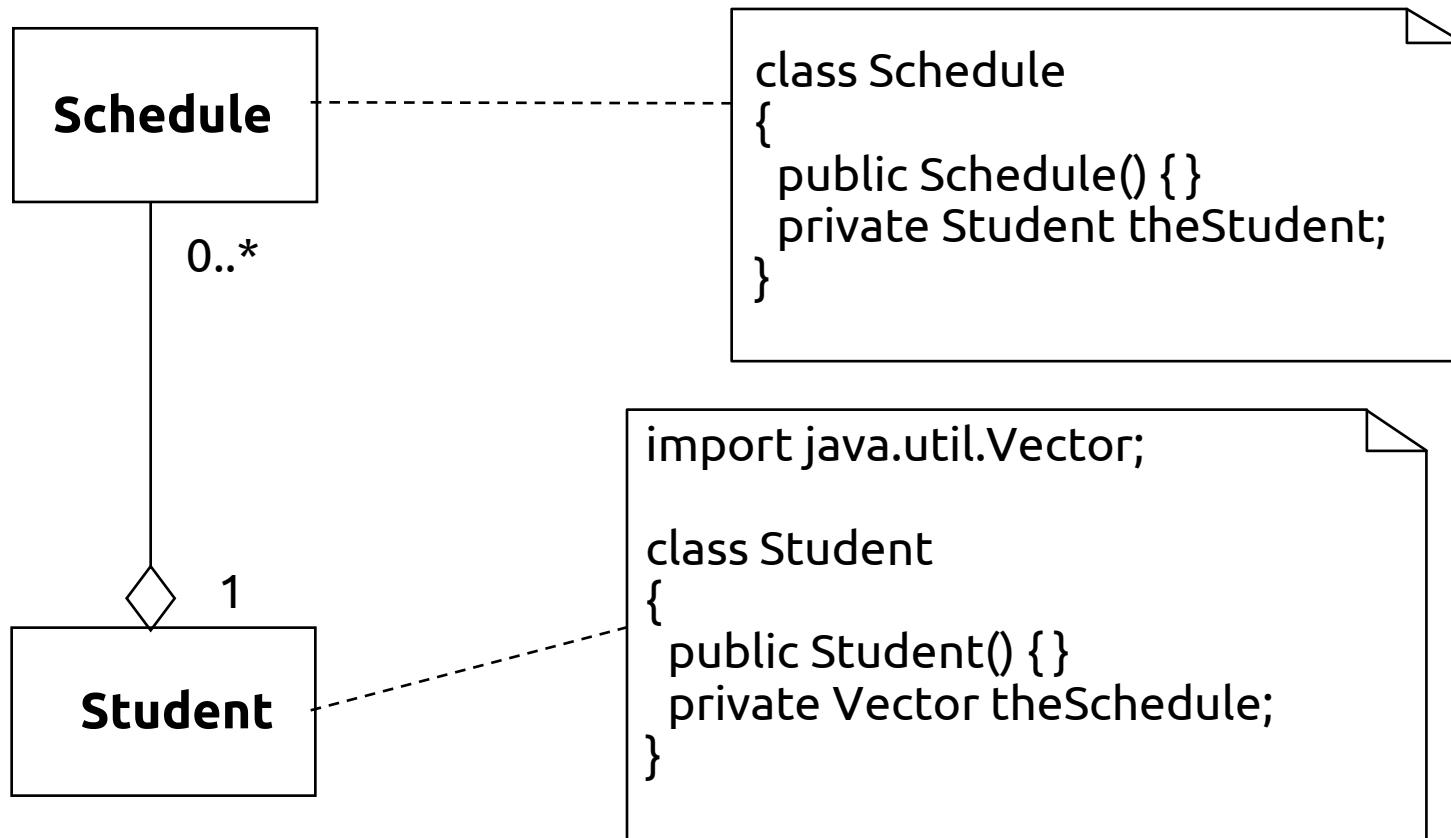
```
import java.util.Vector;
```

```
class Course  
{  
    public Course() {}  
    private Vector prerequisites;  
}
```

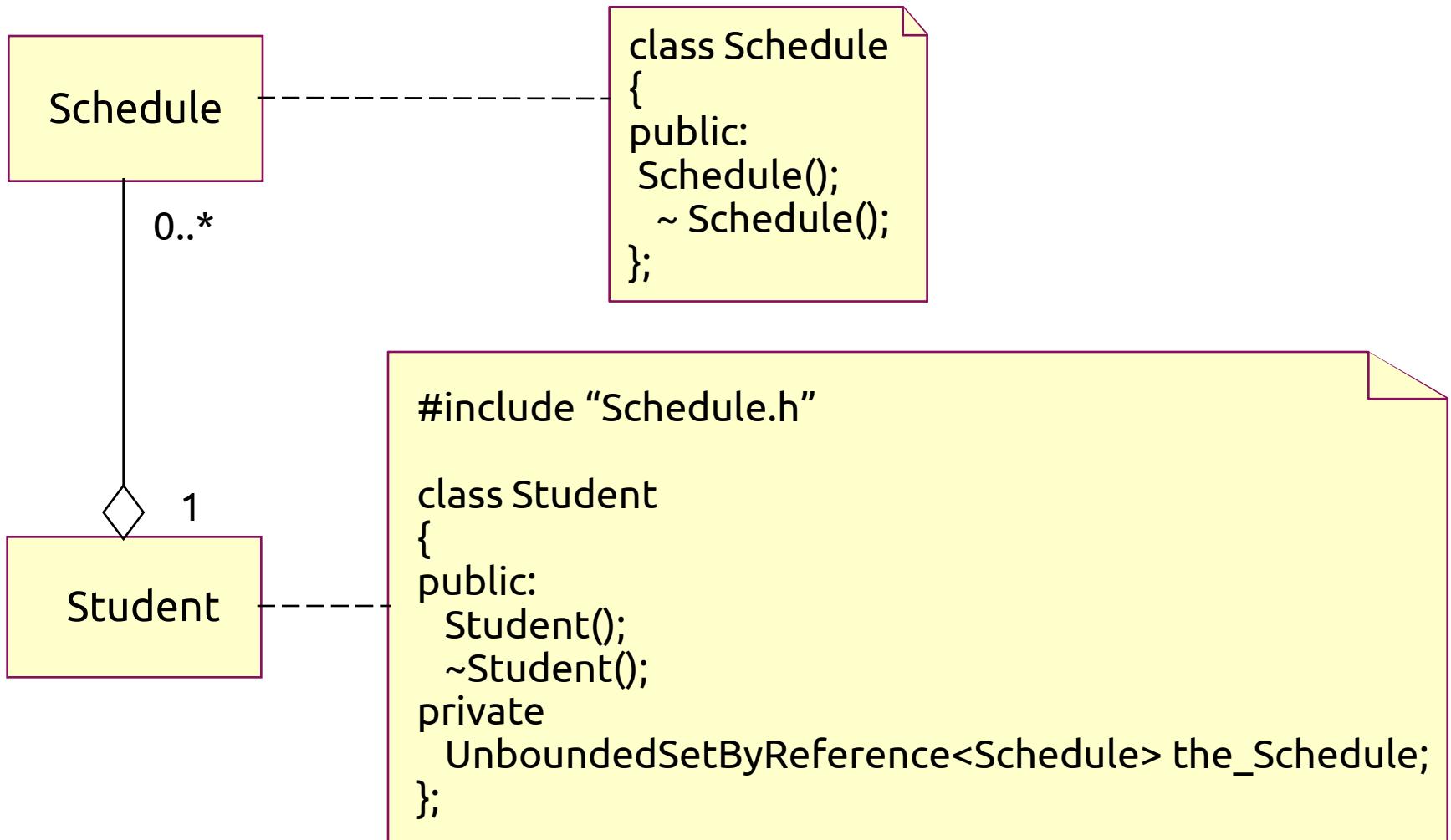


```
class Course  
{  
public:  
    Course();  
    ~Course();  
private:  
    UnboundedSetByReference<Course> prerequisites;  
};
```

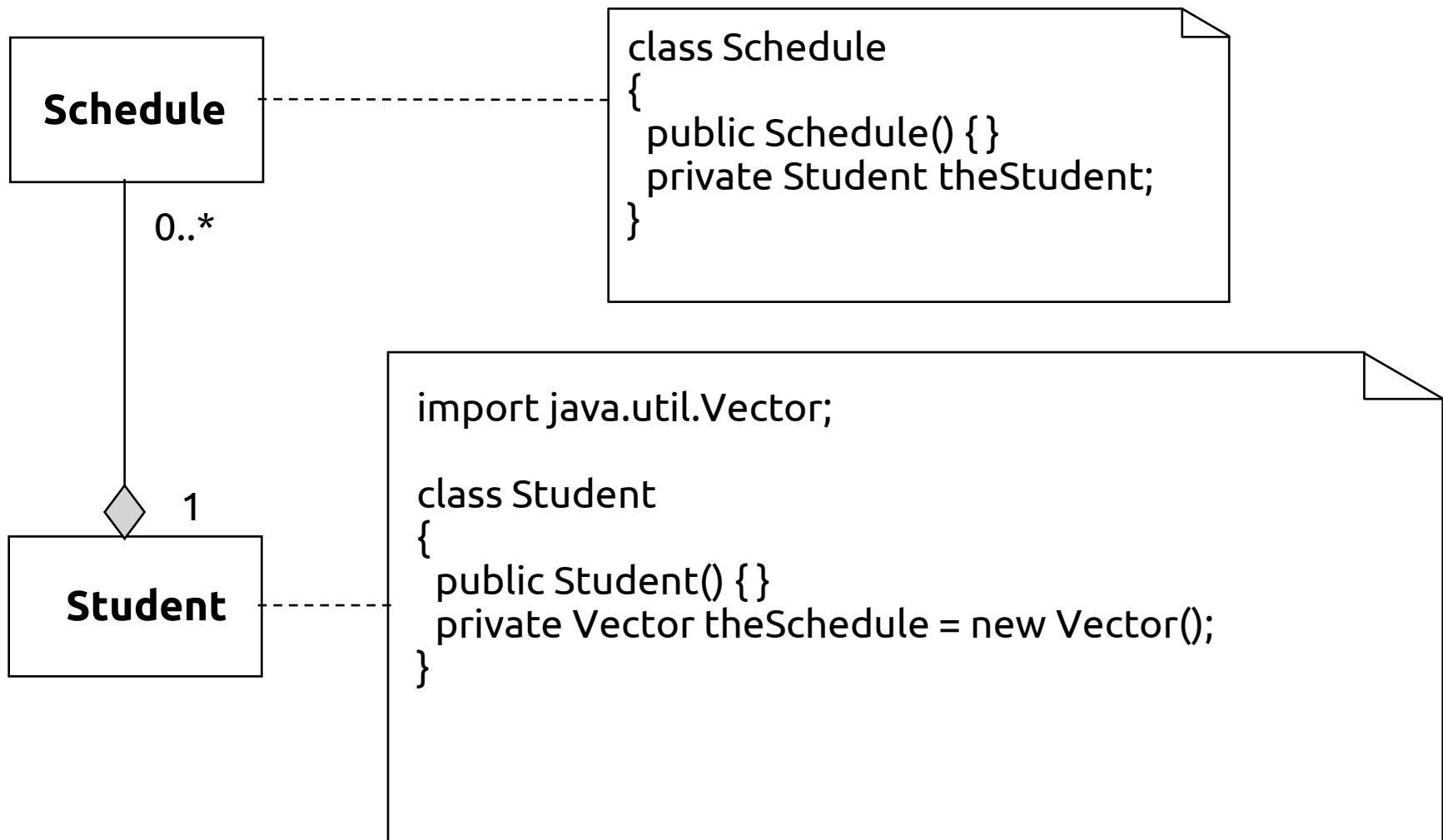
Agregacja



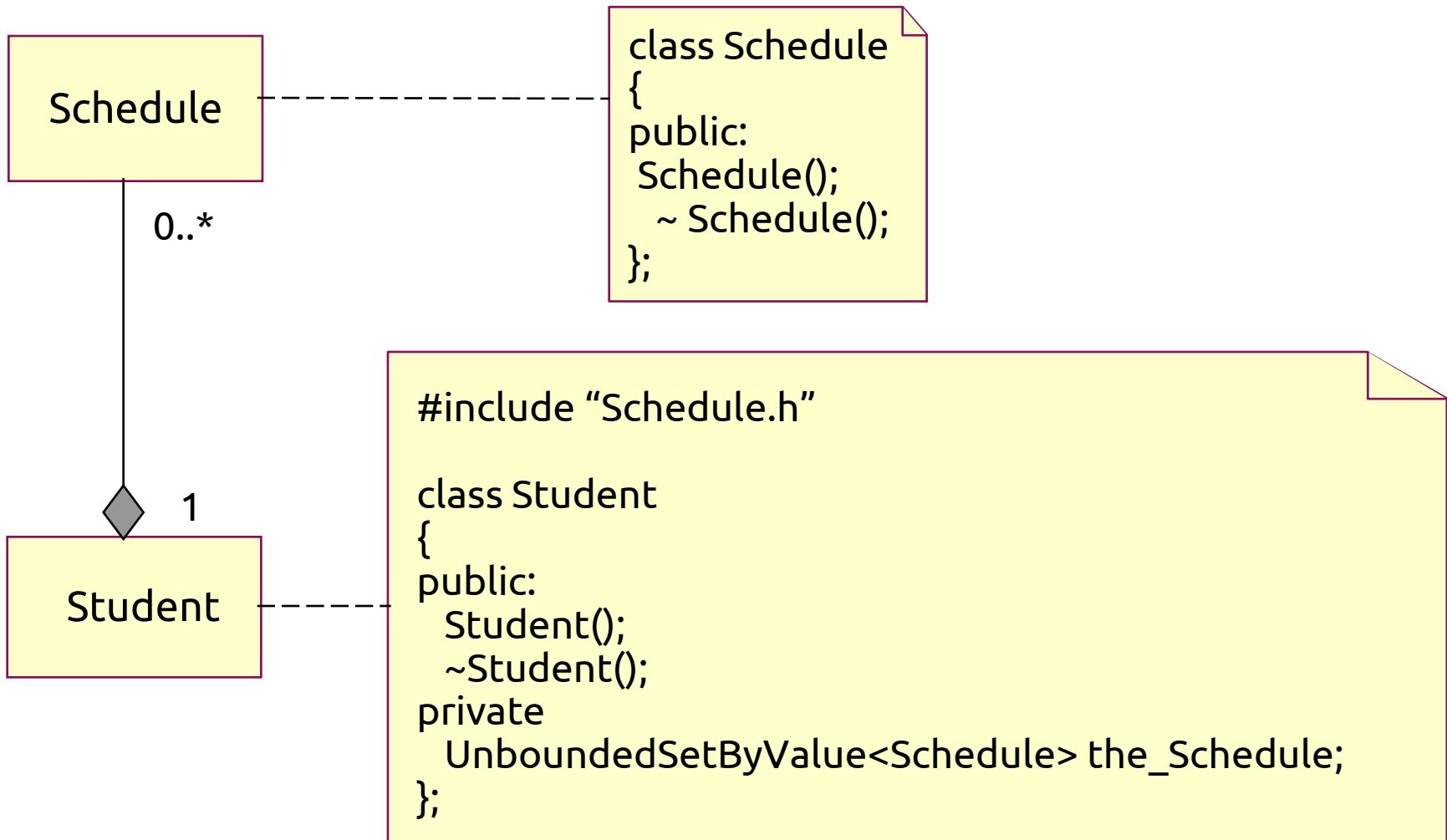
Agregacja



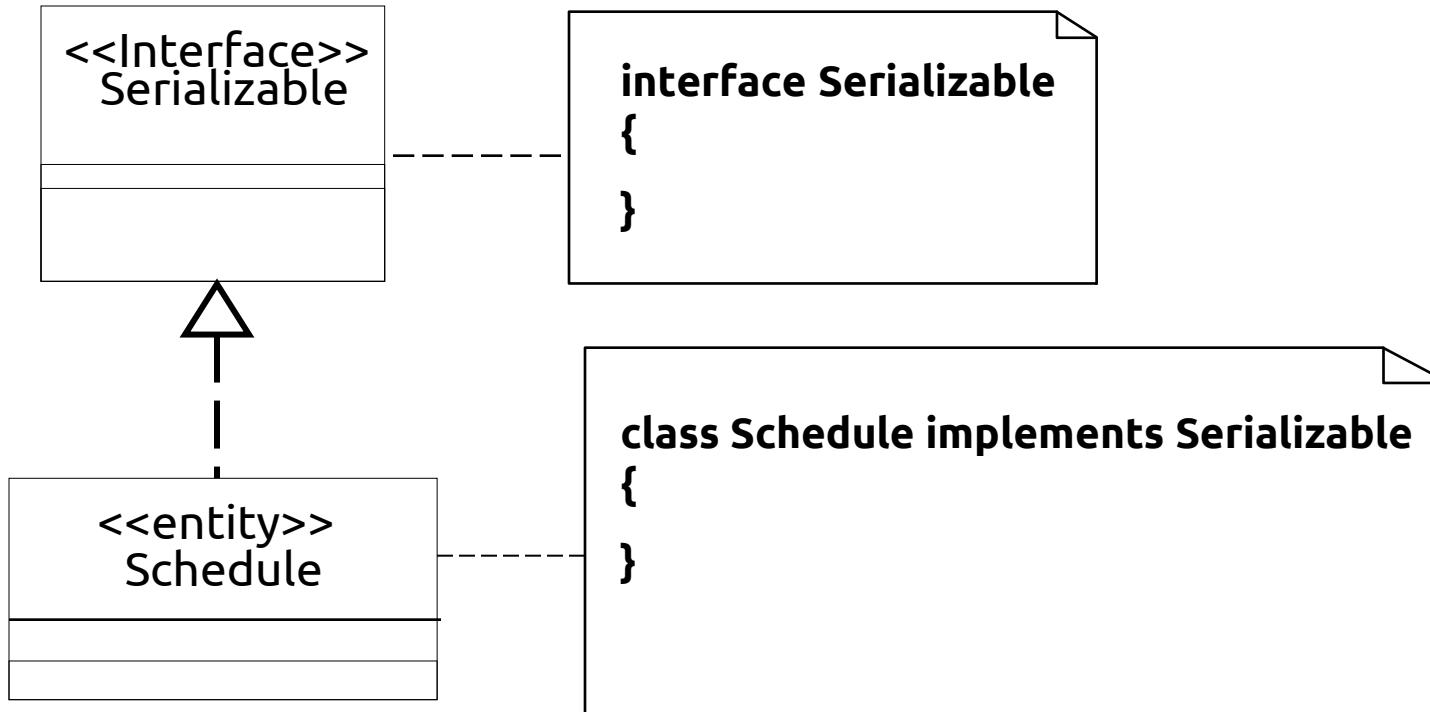
Kompozycja



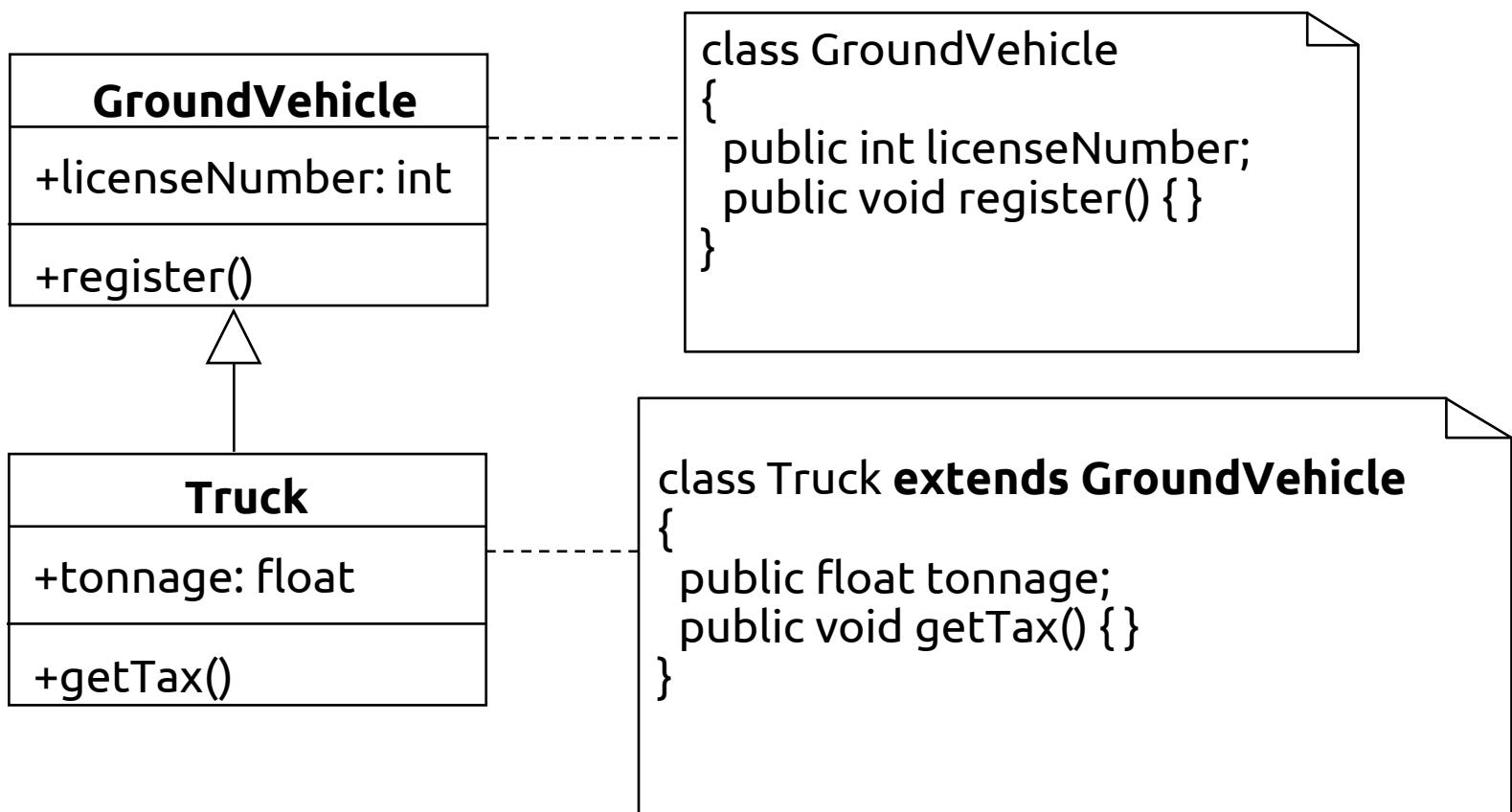
Kompozycja



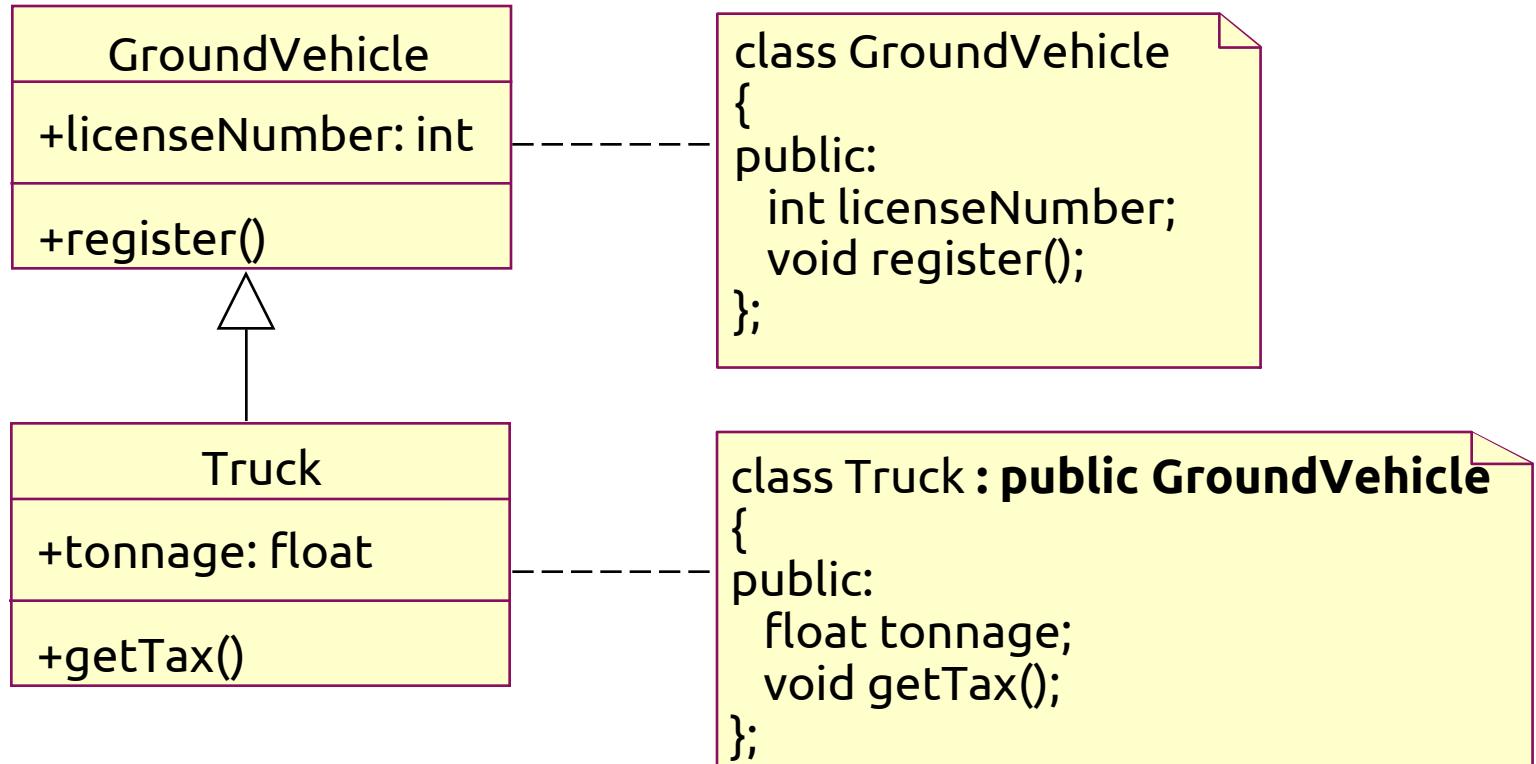
Realizacja interfejsów



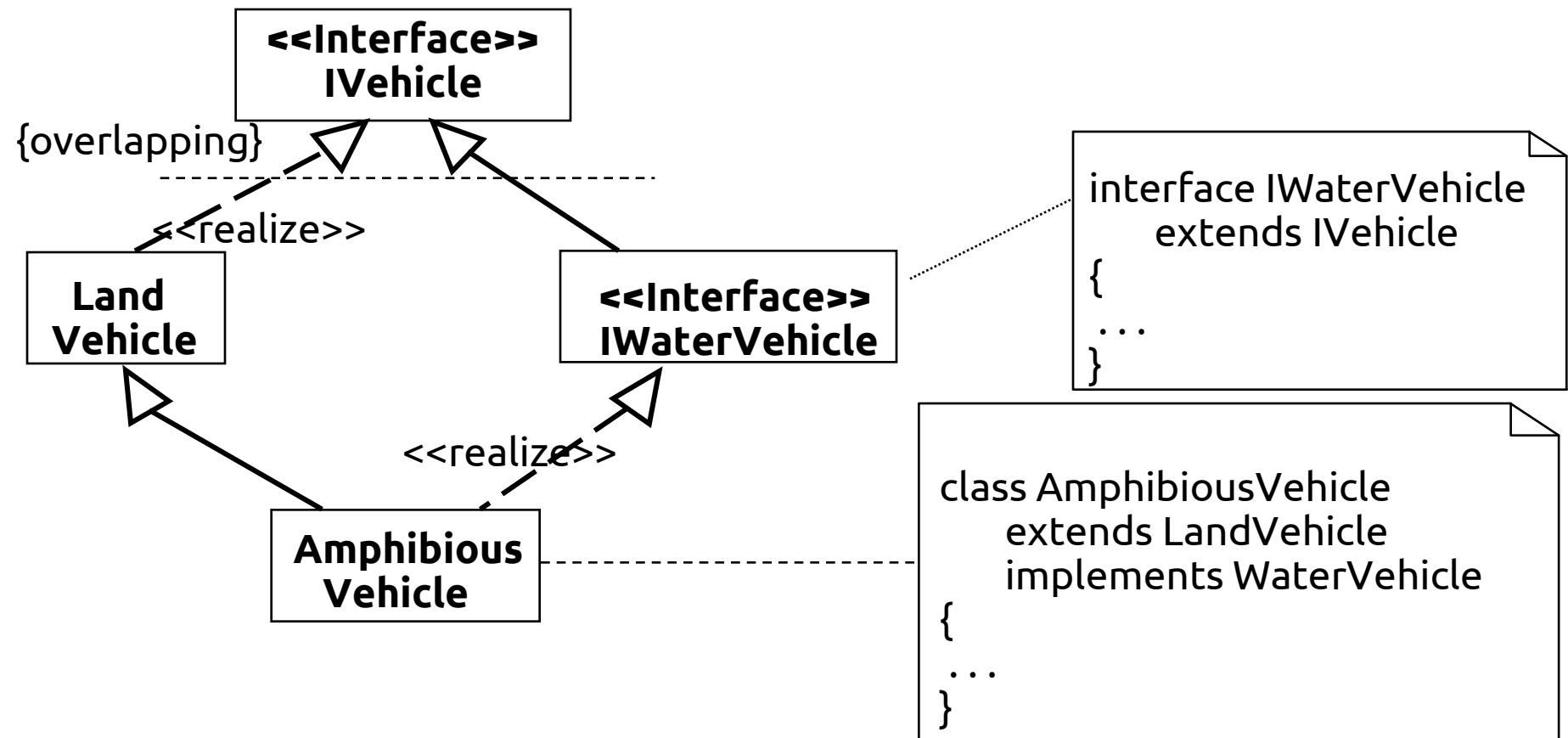
Dziedziczenie – generalizacja



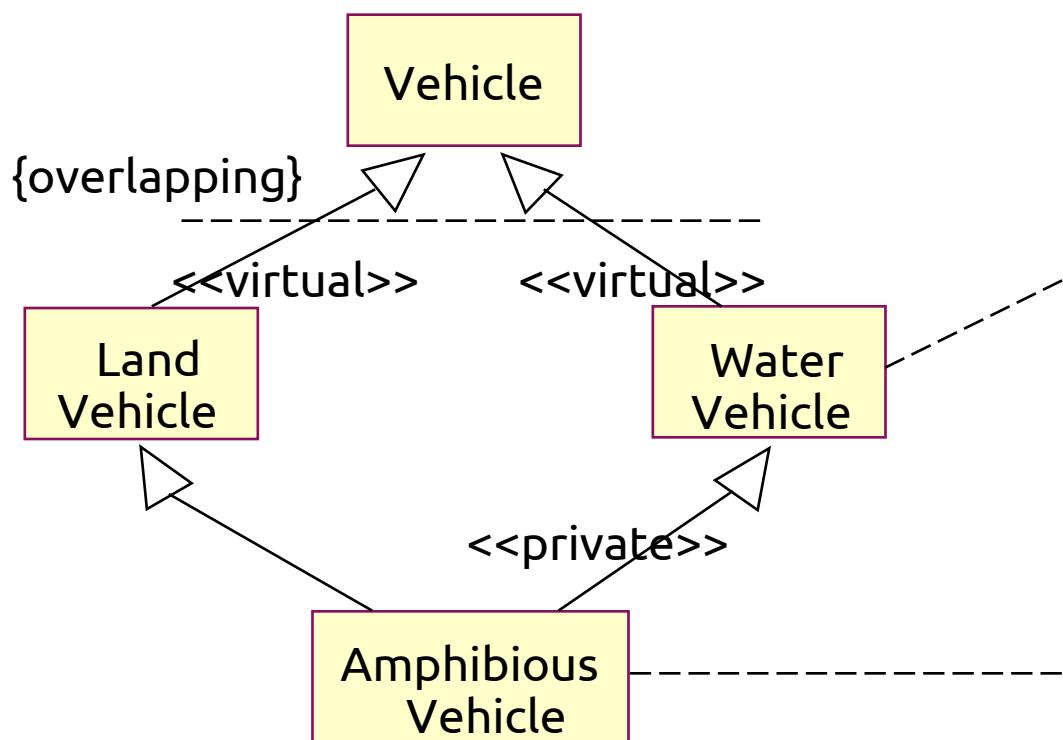
Dziedziczenie – generalizacja



Dziedziczenie wielokrotne



Dziedziczenie wielokrotne



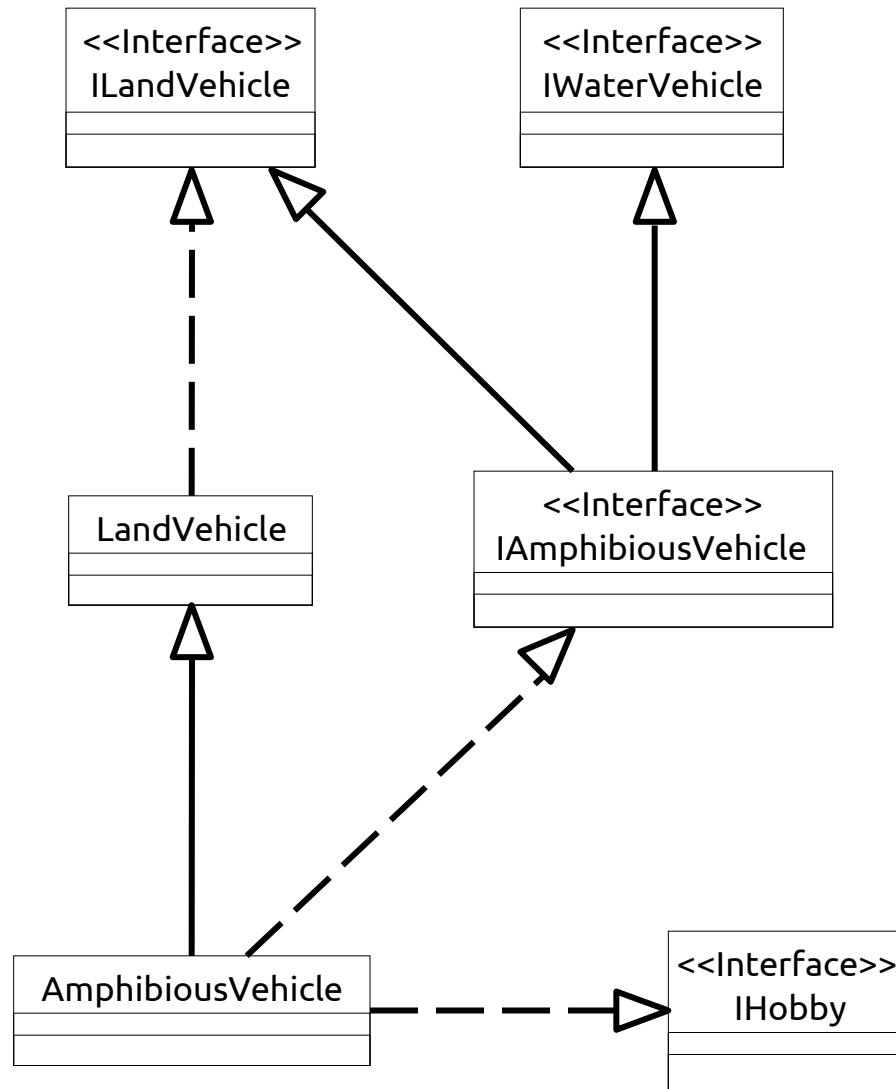
```
#include "Vehicle.h"

class WaterVehicle :
    virtual public Vehicle
{
    ...
};
```

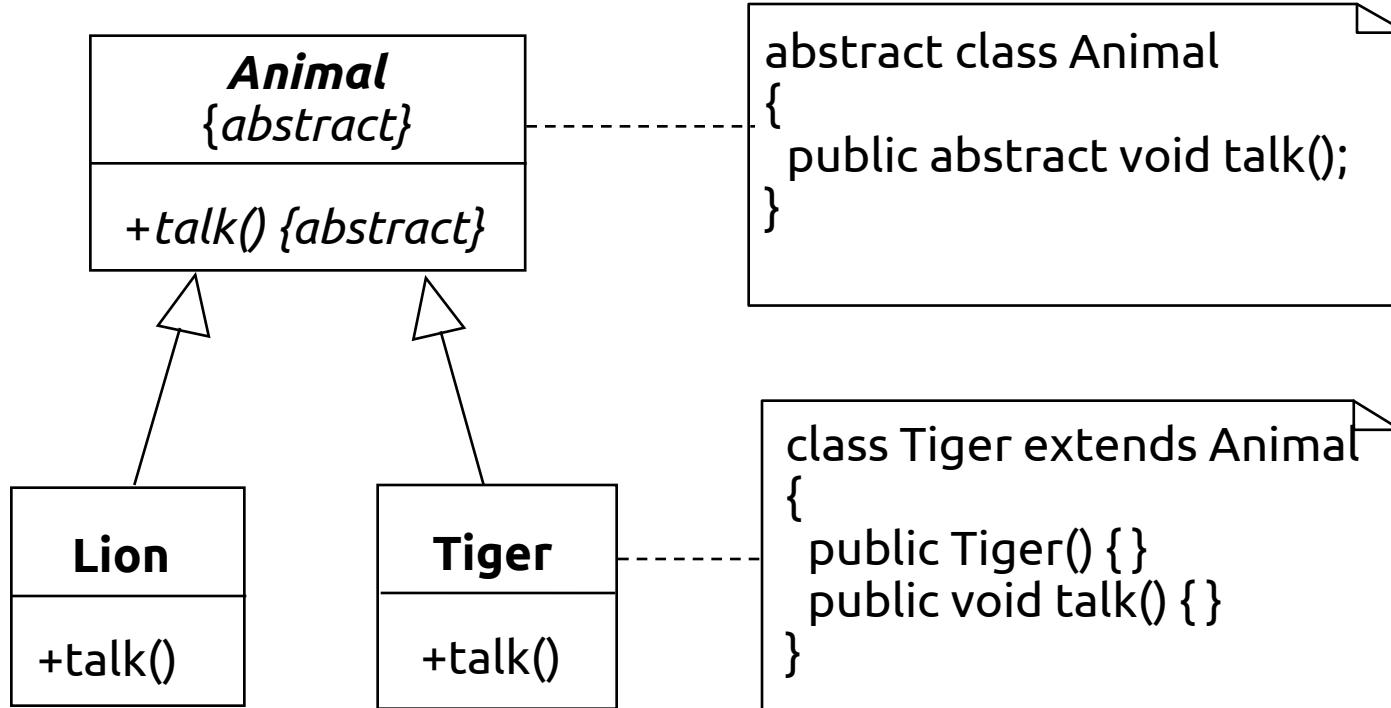
```
#include "LandVehicle.h"
#include "WaterVehicle.h"

class AmphibiousVehicle :
    public LandVehicle,
    private WaterVehicle
{
    ...
};
```

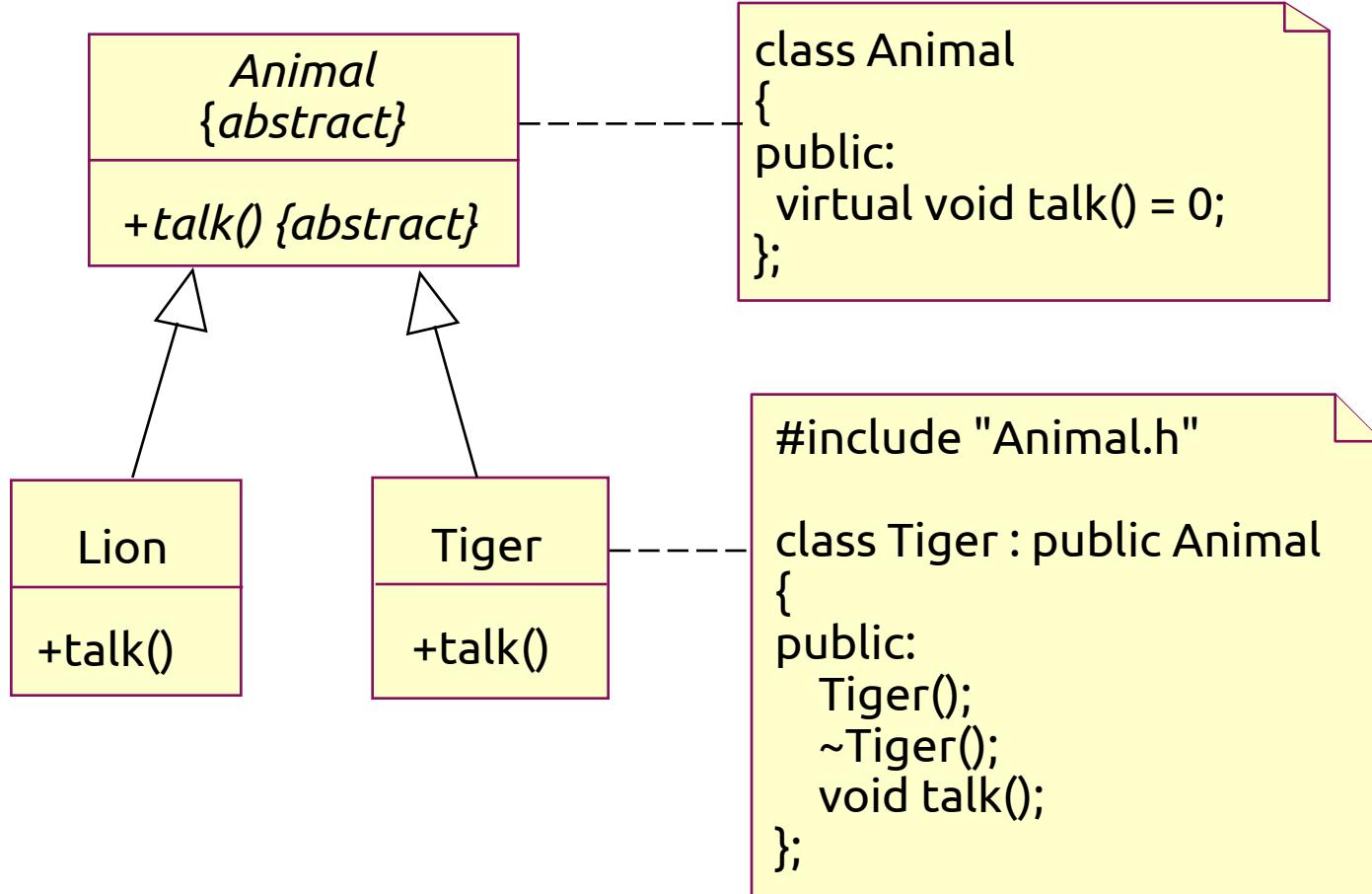
Dziedziczenie wielokrotne interfejsów



Klasa abstrakcyjna



Klasa abstrakcyjna



Wzorce projektowe – motywacja

Zostać mistrzem



eksperci
zachowują się
inaczej niż
początkujący



Zostać wspaniałym deweloperem

znajomość programowania / języków jest kluczowa
ale nie jest wystarczająca

metody inżynierii oprogramowania
kładą nacisk na notacje projektowe

dobry projekt to coś więcej niż rysowanie diagramów

zatem:
najlepsi deweloperzy bazują na doświadczeniu w projektowaniu

Gdzie widać doświadczenie projektowe?

dobrze zaprojektowane
oprogramowanie



powtarzalne struktury i zachowanie

abstrakcja

elastyczność

ponowne
użycie

jakość

modularność

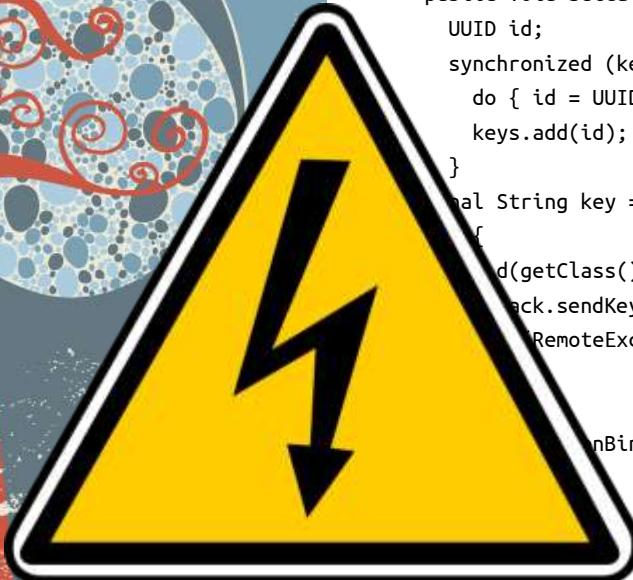
Gdzie widać doświadczenie projektowe?

głowy ekspertów



kod źródłowy

```
public class KeyGeneratorImpl extends Service {  
    private Set<UUID> keys = new HashSet<UUID>();  
    private final KeyGenerator.Stub binder = new KeyGenerator.Stub()  
{  
        public void setCallback (final KeyGeneratorCallback callback) {  
            UUID id;  
            synchronized (keys) {  
                do { id = UUID.randomUUID(); } while (keys.contains(id));  
                keys.add(id);  
            }  
            final String key = id.toString();  
            callback.sendKey(key);  
            if (e instanceof RemoteException) { e.printStackTrace(); }  
        }  
        onBind(Intent intent) { return this.binder; }  
    };  
}
```



Rozwiążanie



Czym jest wzorzec projektowy?

**opis rozwiązania
typowego problemu
powstającego w określonym kontekście**

Znajomość wzorców kluczem do mistrzostwa



Czym jest wzorzec oprogramowania?

**opis rozwiązania
typowego problemu
powstającego w określonym kontekście**

- nazewnictwo w powtarzalnej strukturze projektowej
- jawne wyspecyfikowanie struktury → identyfikacja kluczowych klas/obiektów
 - role i powiązania, zależności, interakcje, konwencje
- abstrakcja od konkretnych elementów projektowych
- wydobyta i skodyfikowana wiedza ekspertów

Wprowadzenie do wzorców projektowych

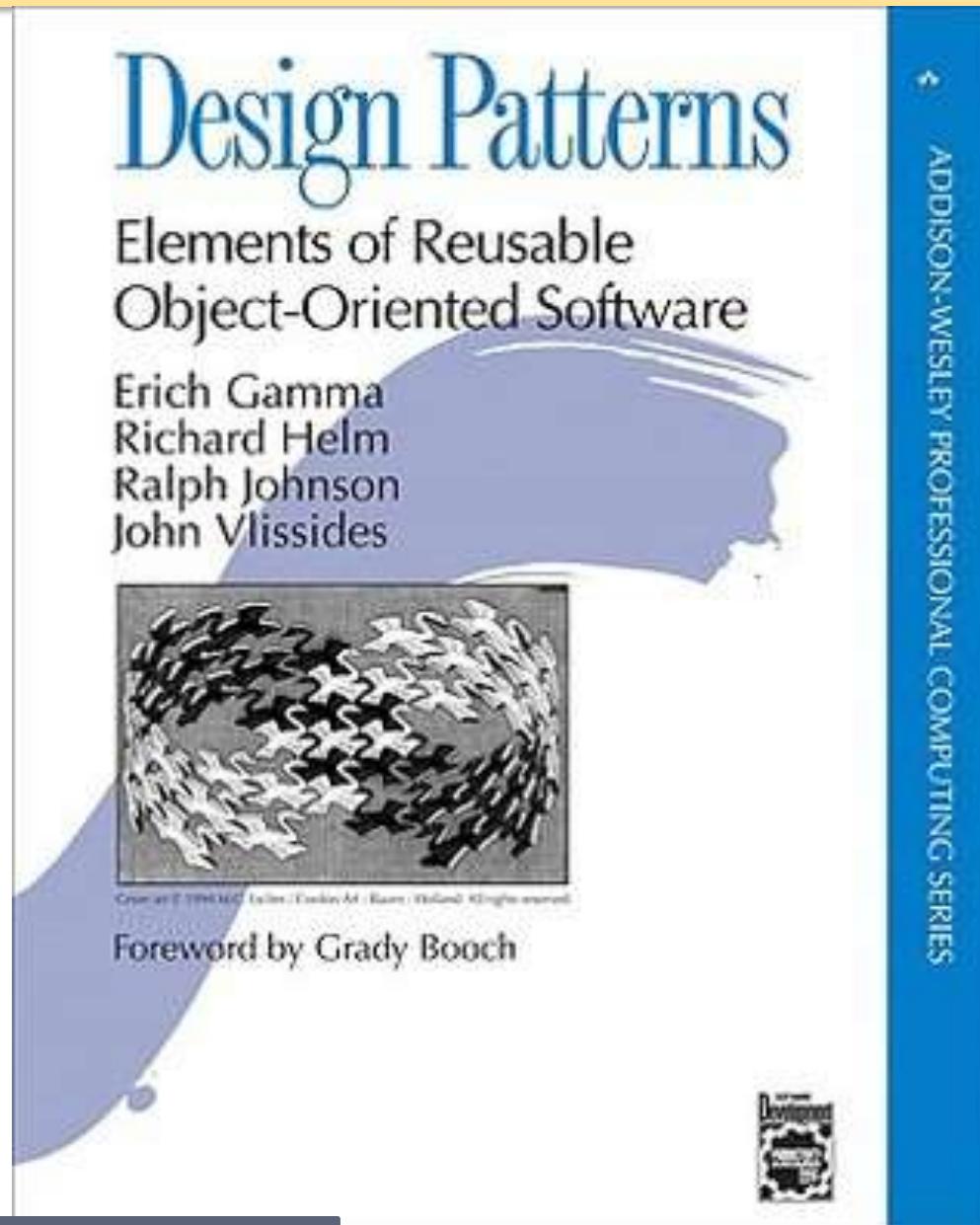
Typowe cechy wzorca

- opisuje *rzecz i proces*
 - rzecz – szkic konkretnego rozwiązania projektowego i/lub opis fragmentu kodu
 - proces – kroki potrzebne do zbudowania rzeczy
- może być niezależny od języka programowania / techniki implementacji
- określa „mikro-architekturę”
- dostosowanie do konkretnej technologii / języka
- nie są metodykami wytwórczymi

Opisywanie wzorców projektowych

- nazwa
- zamiar
- rozwiązywany problem
- rozwiązanie
- przykłady i pomoc w implementacji
- konsekwencje
- znane zastosowania
- powiązane wzorce

Banda Czterech



Klasyfikacja wzorców projektowych

		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	Metoda wytwórcza	Adapter	Interpreter Metoda szablonowa
	Obiekt	Fabryka abstrakcyjna Budowniczy Prototyp Singleton	Adapter Kompozyt Dekorator Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań Polecenie Mediator Pamiątka Obserwator Stan Strategia Odwiedzający

Przegląd wzorców projektowych

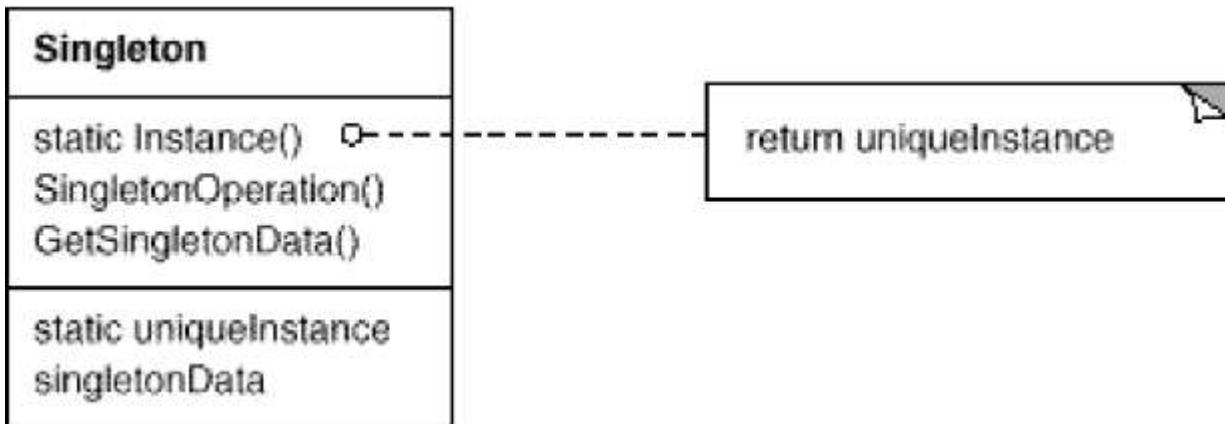
Singleton

		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	Metoda wytwórcza	Adapter	Interpreter Metoda szablonowa
	Obiekt	Fabryka abstrakcyjna Budowniczy Prototyp <u>Singleton</u>	Adapter Kompozyt Dekorator Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań Polecenie Mediator Pamiątka Obserwator Stan Strategia Odwiedzający

Singleton

- przeznaczenie
 - zagwarantowanie możliwości **utworzenia tylko jednego obiektu danej klasy** i zapewnienie globalnego dostępu do niego
- uzasadnienie
 - wiele obiektów wykorzystuje dany obiekt, który powinien być jedną instancją danej klasy

Singleton – struktura



Singleton – przykłady implementacji

- schemat prosty
 - headfirst.singleton.stat – tworzony od razu
 - headfirst.singleton.classic – tworzony w razie potrzeby
- konkretny
 - headfirst.singleton.chocolate
 - jaki problem?
- bezpieczny (wielowątkowość), ale kosztowny
 - headfirst.singleton.threadsafe
- bezpieczny, mniej kosztowny
 - headfirst.singleton.dcl

Dekorator

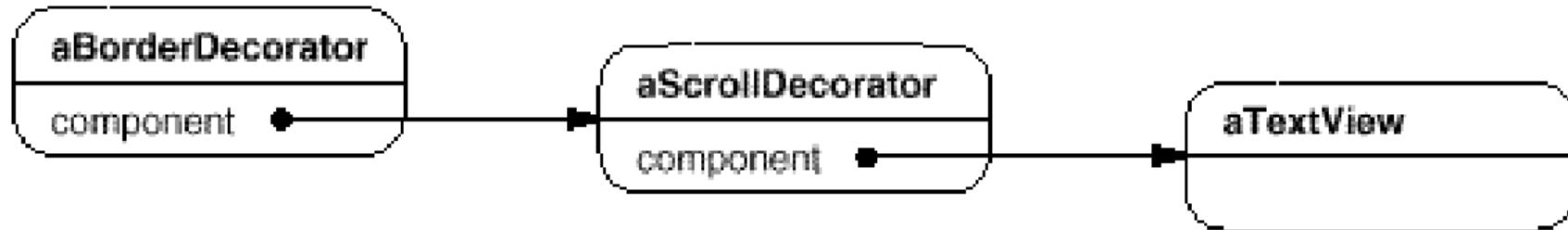
		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	Metoda wytwórcza	Adapter	Interpreter Metoda szablonowa
	Obiekt	Fabryka abstrakcyjna Budowniczy Prototyp Singleton	Adapter Kompozyt Dekorator Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań Polecenie Mediator Pamiątka Obserwator Stan Strategia Odwiedzający

Dekorator (nakładka – wrapper)

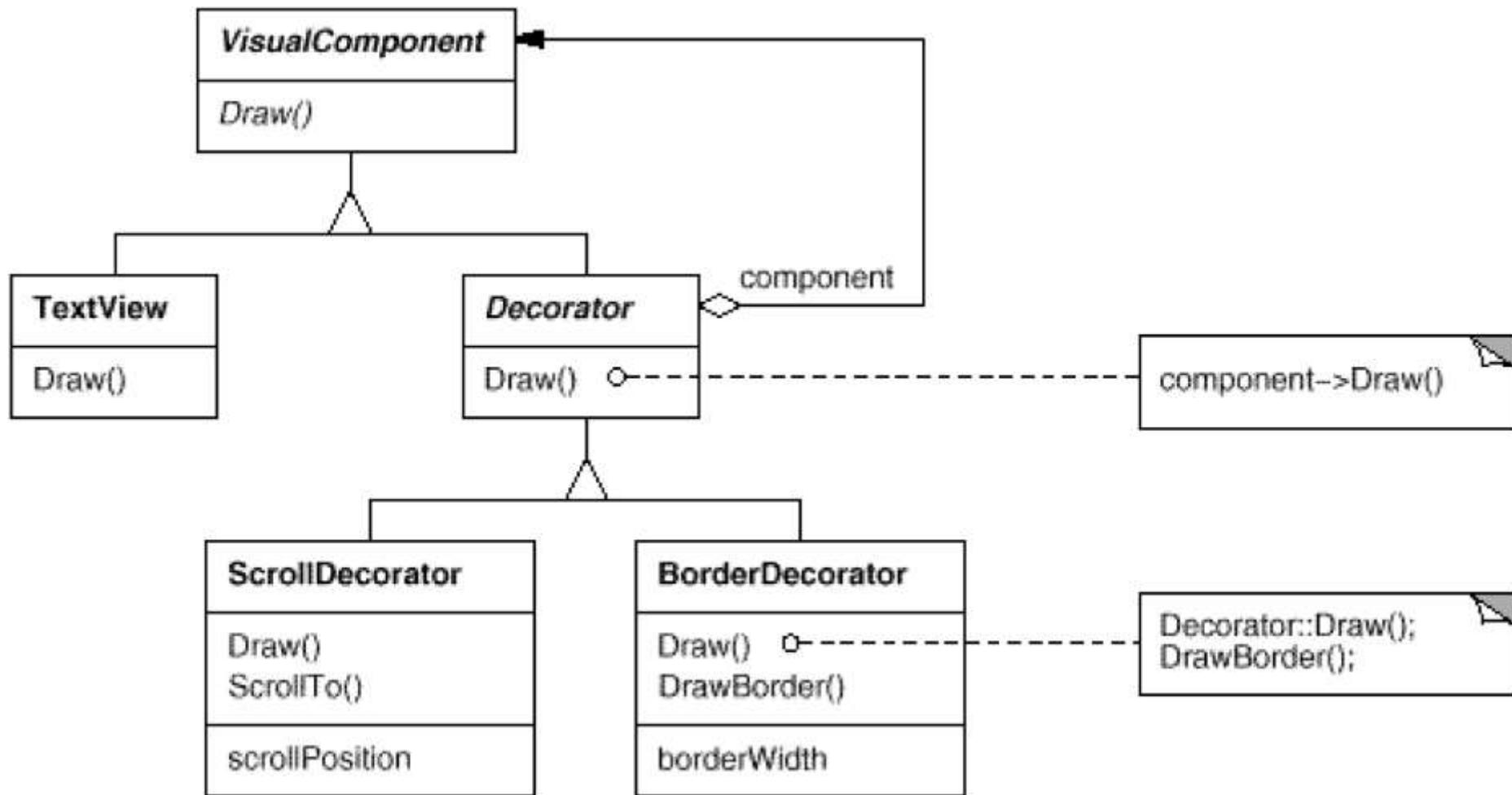
- przeznaczenie
 - umożliwia dynamiczne dołączanie nowych funkcjonalności obiektu → alternatywny elastyczny sposób tworzenia podklas o wzbogaconych funkcjach
- uzasadnienie
 - potrzeba dodania zadań do pojedynczych obiektów, nie całej klasy
 - tj. do obiektów różnych klas
 - np. dodawanie atrybutów (np. ramka) i operacji (przewijanie) do dowolnego komponentu GUI

Dekorator

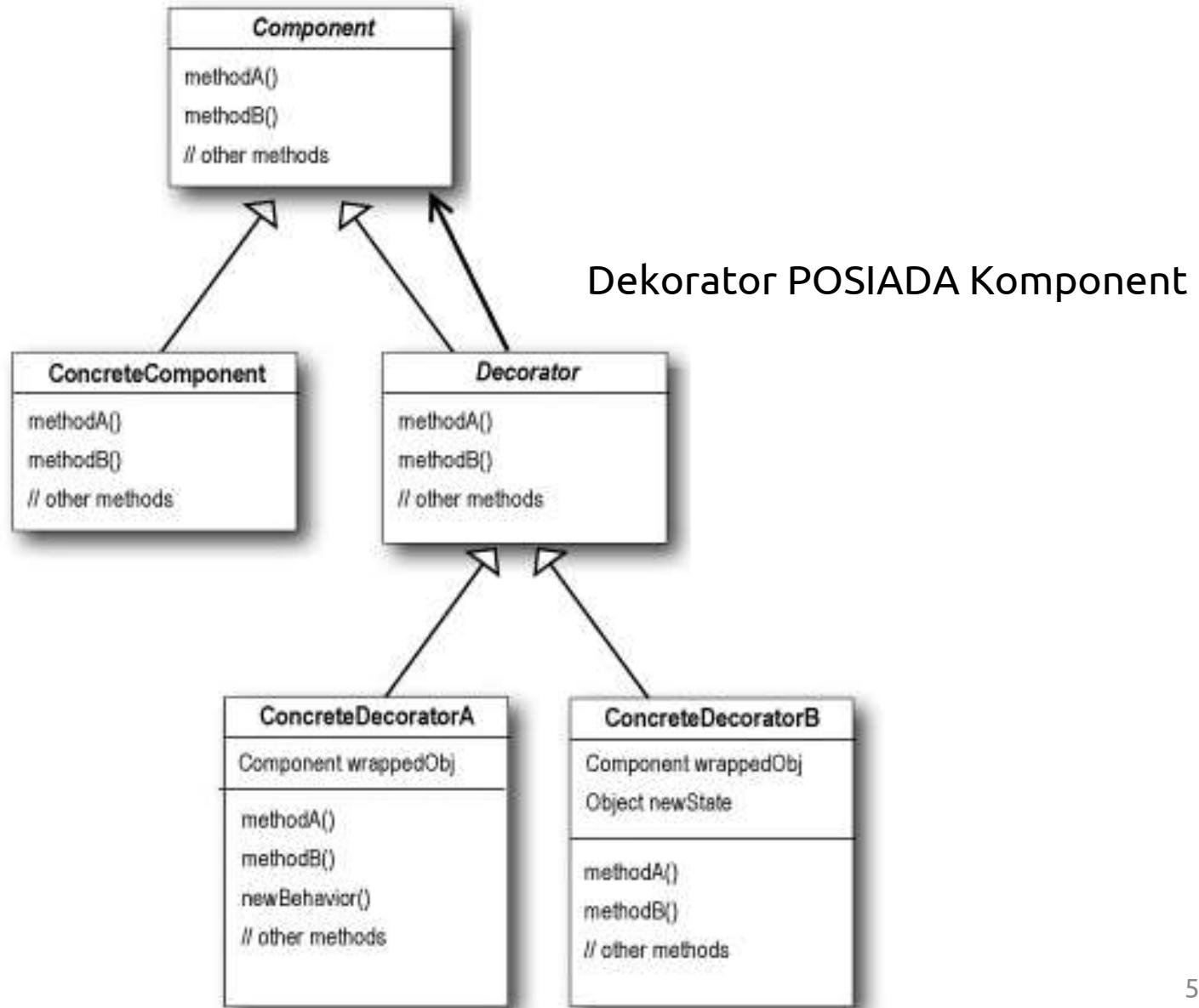
- rozwiązanie: umieszczenie obiektu w innym obiekcie (dekoratorze)
- dekorator zgodny z interfejsem ozdabianego elementu
 - efekt: jego obecność niezauważalna dla klientów ozdabianego komponentu
- dekorator przekazuje żądania do komponentu
 - może wykonywać dodatkowe działania
- efekt: rekurencyjne zagnieżdżanie dekoratorów



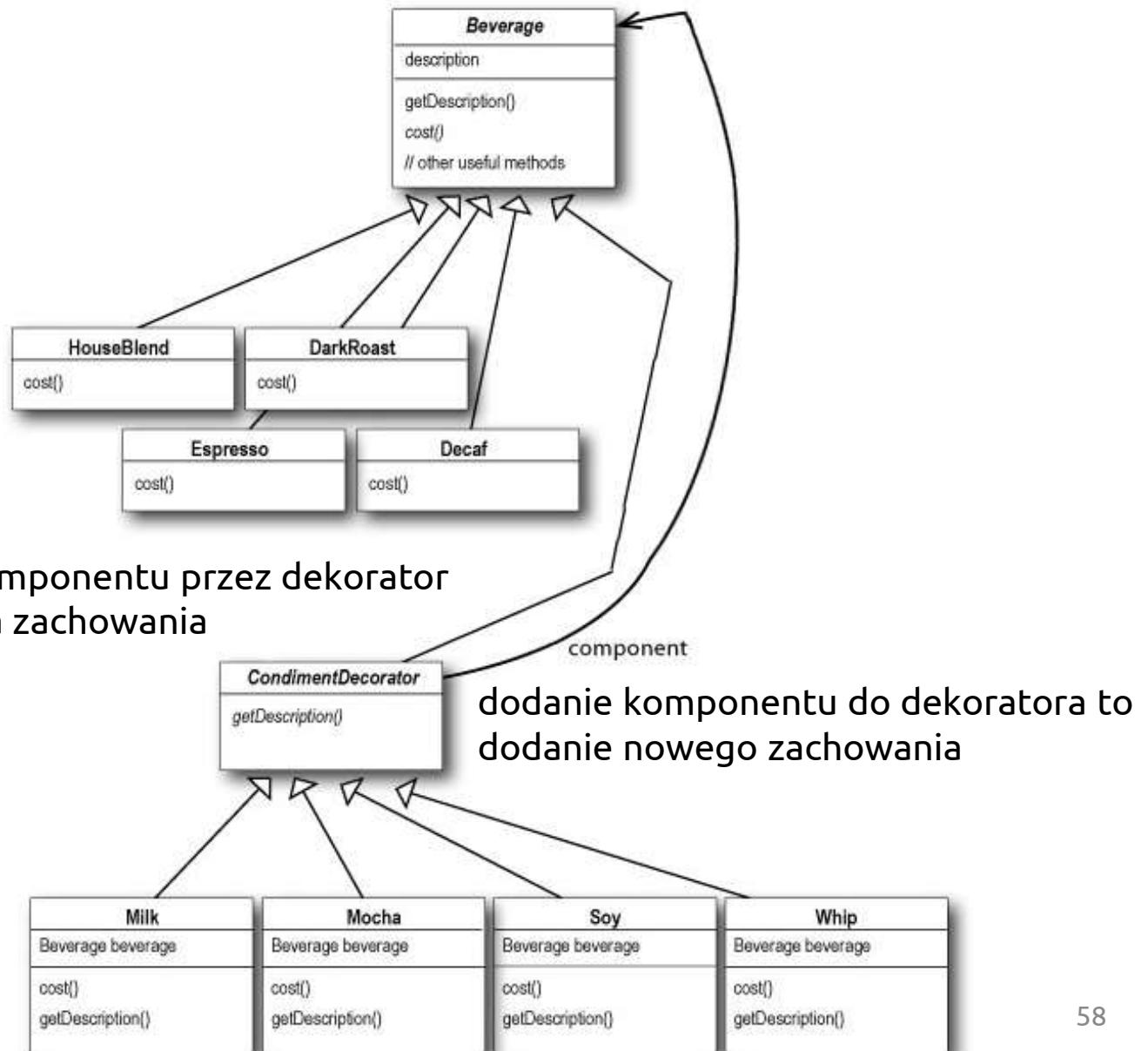
Dekorator



Dekorator – struktura



Dekorator – przykład



Dekorator – przykład



Dekorator – przykład – kod

- `headfirst.decorator.starbuzz`
 - Beverage
 - CondimentDecorator
 - Espresso, HouseBlend
 - Mocha, Milk
 - StarbuzzCoffee

Dekorator – wnioski

- dziedziczenie jest jedną z form rozszerzania, nie zawsze najlepszą
- powinniśmy pozwalać na rozszerzanie zachowania bez modyfikacji istniejącego kodu
- kompozycja i delegacja używane do dodawania zachowania w czasie działania
- wzorzec Dekorator jest alternatywą wobec dziedziczenia
- wzorzec Dekorator obejmuje zbiór klas dekoratorów obejmujących konkretne komponenty
- klasa Dekoratora naśladuje zachowanie komponentu, który dekoruje

Adapter

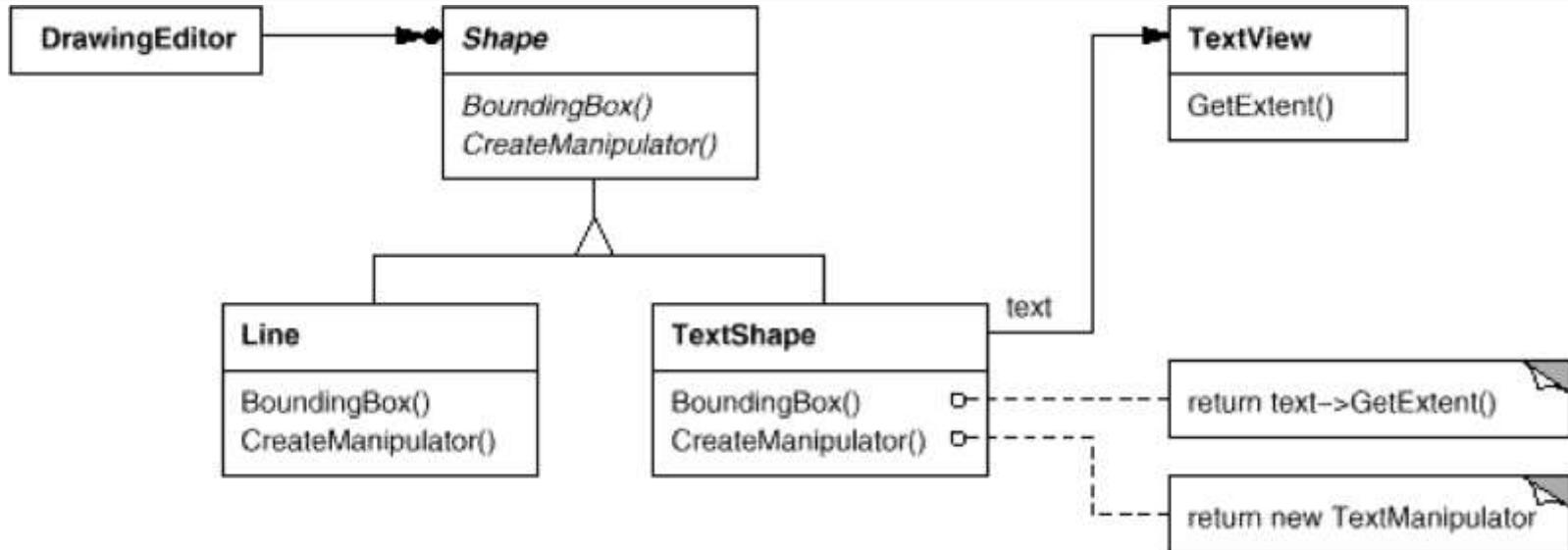
		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	Metoda wytwórcza	<u>Adapter</u>	Interpreter Metoda szablonowa
	Obiekt	Fabryka abstrakcyjna Budowniczy Prototyp Singleton	<u>Adapter</u> Kompozyt Dekorator Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań Polecenie Mediator Pamiątka Obserwator Stan Strategia Odwiedzający

Adapter

- przeznaczenie
 - przekształca interfejs klasy na inny, oczekiwany przez klienta
 - umożliwia współdziałanie klasom, które nie mogą bezpośrednio współpracować ze względu na niezgodne interfejsy

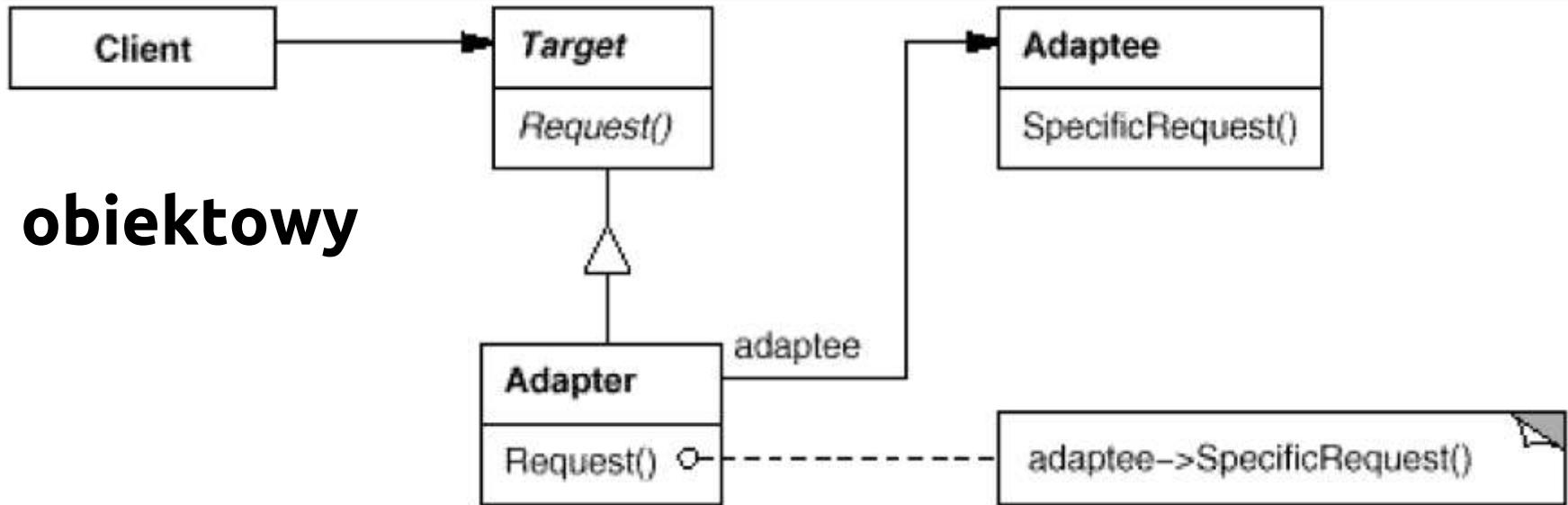


Adapter – przykład



- dostosowanie wykorzystania **TextView** przez **DrawingEditor**
- **DrawingEditor** potrafi obsługiwać **Shape** (i pochodne)
- nie możemy bezpośrednio dostosować **TextView**
 - np. brak dostępu do kodu źródłowego

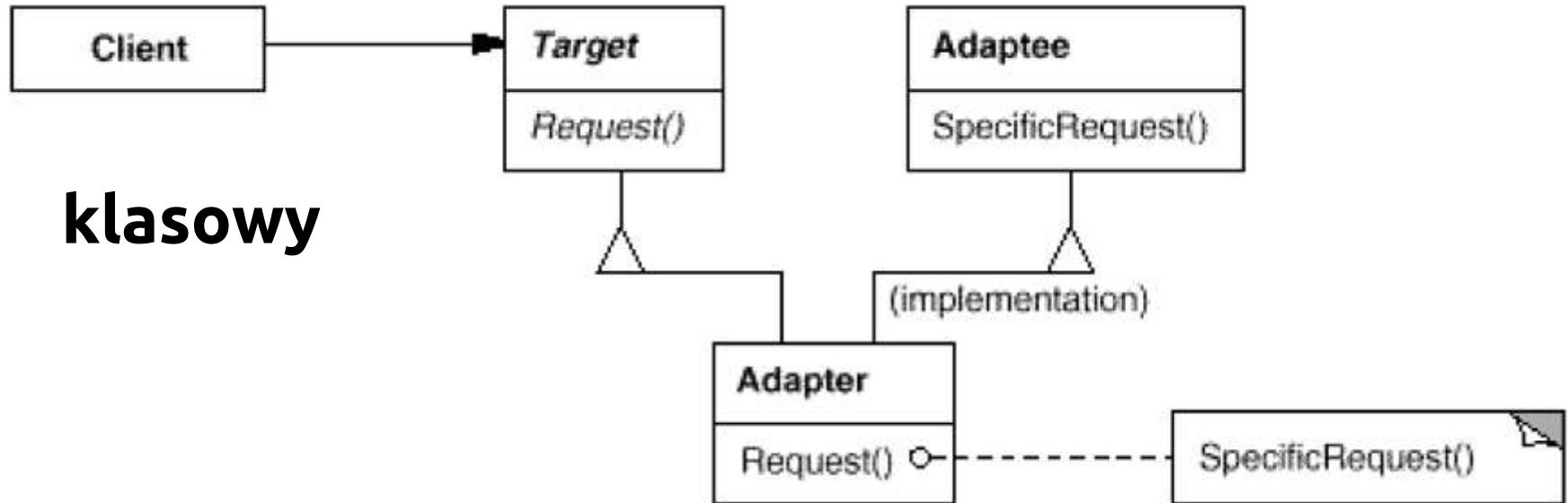
Adapter – struktura



obiektowy

- **Target** definiuje interfejs używany przez klienta
- **Klient** współdziała z obiektami zgodnymi z interfejsem **Target**
- **Adaptee** (adaptowany) definiuje interfejs, który trzeba dostosować
- **Adapter** dostosowuje interfejs Adaptee do Target

Adapter – struktura



- **Target** definiuje interfejs używany przez klienta
- **Klient** współdziała z obiektami zgodnymi z interfejsem **Target**
- **Adaptee** (adaptowany) definiuje interfejs, który trzeba dostosować
- **Adapter** dostosowuje interfejs Adaptee do Target

Adapter – przykład – kod

- `headfirst.adapter.ducks`
 - Duck – interfejs
 - MallardDuck – implementacja
 - Turkey – interfejs
 - WildTurkey – implementacja
 - mamy mało obiektów Duck, spróbujmy dostosować coś podobnego (Turkey) → TurkeyAdapter (obiektowy)
 - implements Duck – „udaje” Duck
 - ale wykorzystuje Turkey
 - `duck.fly()` odpowiada za 5x `turkey.fly()`
 - `DuckTestDrive.testDuck(Duck duck)`

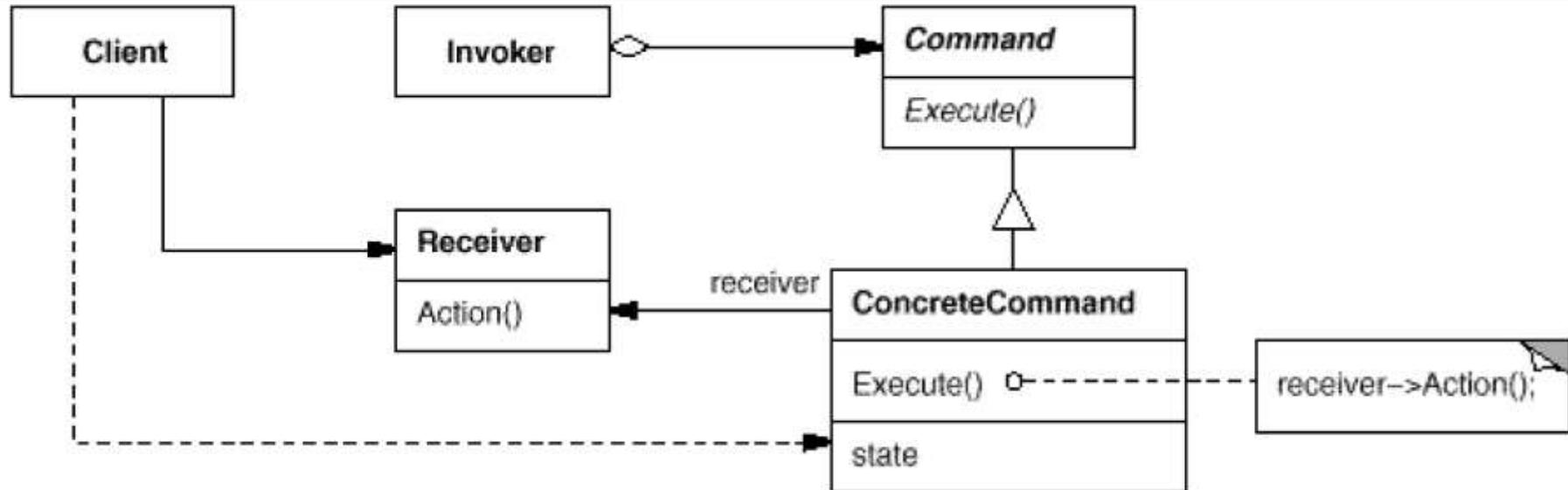
Polecenie

		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	Metoda wytwórcza	Adapter	Interpreter Metoda szablonowa
	Obiekt	Fabryka abstrakcyjna Budowniczy Prototyp Singleton	Adapter Kompozyt Dekorator Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań Polecenie Mediator Pamiątka Obserwator Stan Strategia Odwiedzający

Polecenie

- przeznaczenie
 - hermetyzuje żądanie w formie obiektu, a w efekcie
 - umożliwia parametryzację klienta przy użyciu różnych żądań
 - umieszczenie żądań w kolejkach i dziennikach
 - zapewnia obsługę cofania operacji

Polecenie



- **Command** – interfejs przeznaczony do wykonywania operacji
- **ConcreteCommand** definiuje powiązanie z konkretnym działaniem **Receivera**
- **Client** tworzy obiekt **ConcreteCommand** i wiąże go z odpowiednim odbiorcą
- **Invoker** (nadawca) żąda obsłużenia żądania przez **Command**
- **Receiver** (odbiorca) potrafi wykonać operacje potrzebne do obsłużenia żądania; może być dowolną klasą!

Polecenie – przykład

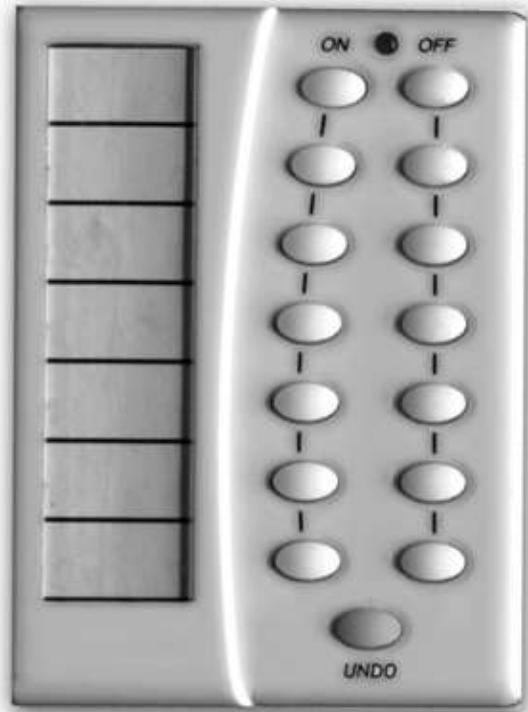


pilot sterujący różnymi urządzeniami
jedno urządzenie
włączanie

Polecenie – przykład – kod

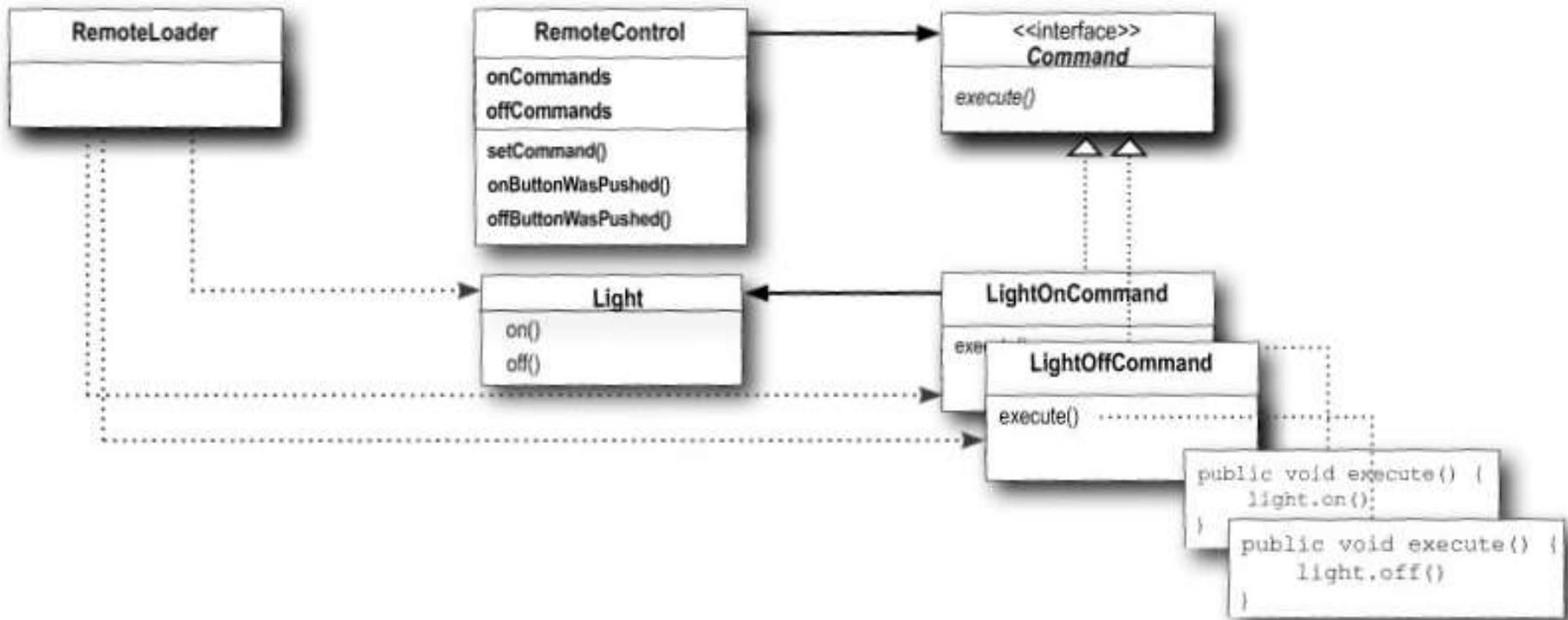
- `headfirst.command.simpleremote`
 - `Command`
 - `SimpleRemoteControl` – *invoker*
 - `GarageDoor`, `LightOnCommand` – *receiver*
 - `GarageDoorOpenCommand`, `LightOnCommand` – *ConcreteCommand*
 - `RemoteControlTest` – *client*

Polecenie – przykład



pilot sterujący różnymi urządzeniami
max 7 urządzeń
włączanie / wyłączanie

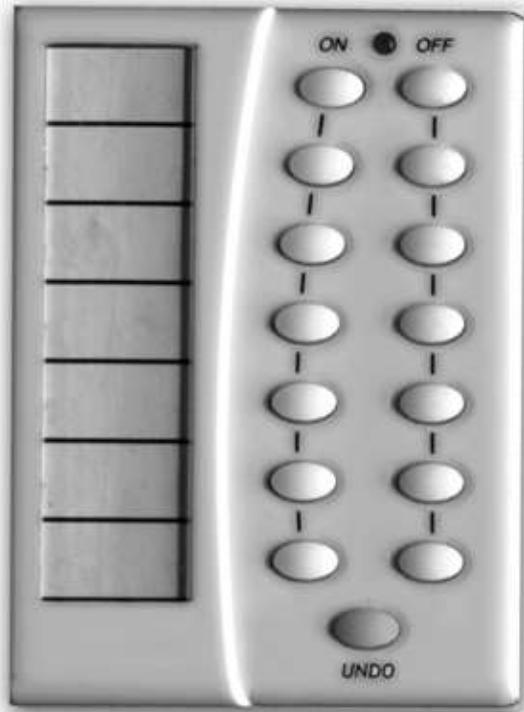
Polecenie – przykład



Polecenie – przykład – kod

- `headfirst.command.remote`
 - Command
 - Light, LightOnCommand, LightOffCommand
 - CeilingFan, CeilingFanOnCommand,
CeilingFanOffCommand
 - Stereo, StereoOnWithCDCommand, StereoOffCommand
 - NoCommand – niezaprogramowane przyciski
 - RemoteControl
 - RemoteLoader

Polecenie – przykład



pilot sterujący różnymi urządzeniami
max 7 urządzeń
włączanie / wyłączanie
cofanie operacji

Polecenie – przykład – kod

- headfirst.command.undo
 - Command – undo()
 - Light
 - LightOnCommand, LightOffCommand – int level;
 - CeilingFan
 - CeilingFanXXXCommand – int prevSpeed;
 - Stereo, StereoOnWithCDCommand, StereoOffCommand
 - NoCommand – puste undo()
 - RemoteControlWithUndo
 - RemoteLoader

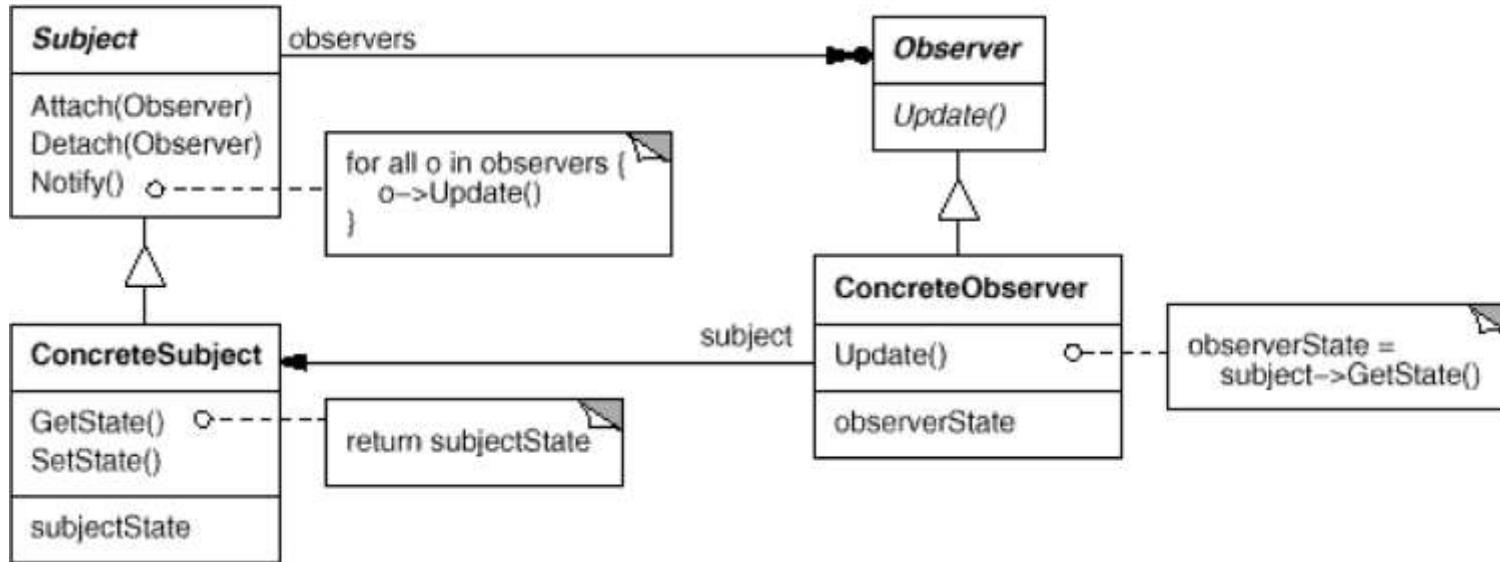
Obserwator

		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	Metoda wytwórcza	Adapter	Interpreter Metoda szablonowa
	Obiekt	Fabryka abstrakcyjna Budowniczy Prototyp Singleton	Adapter Kompozyt Dekorator Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań Polecenie Mediator Pamiątka Obserwator Stan Strategia Odwiedzający

Obserwator

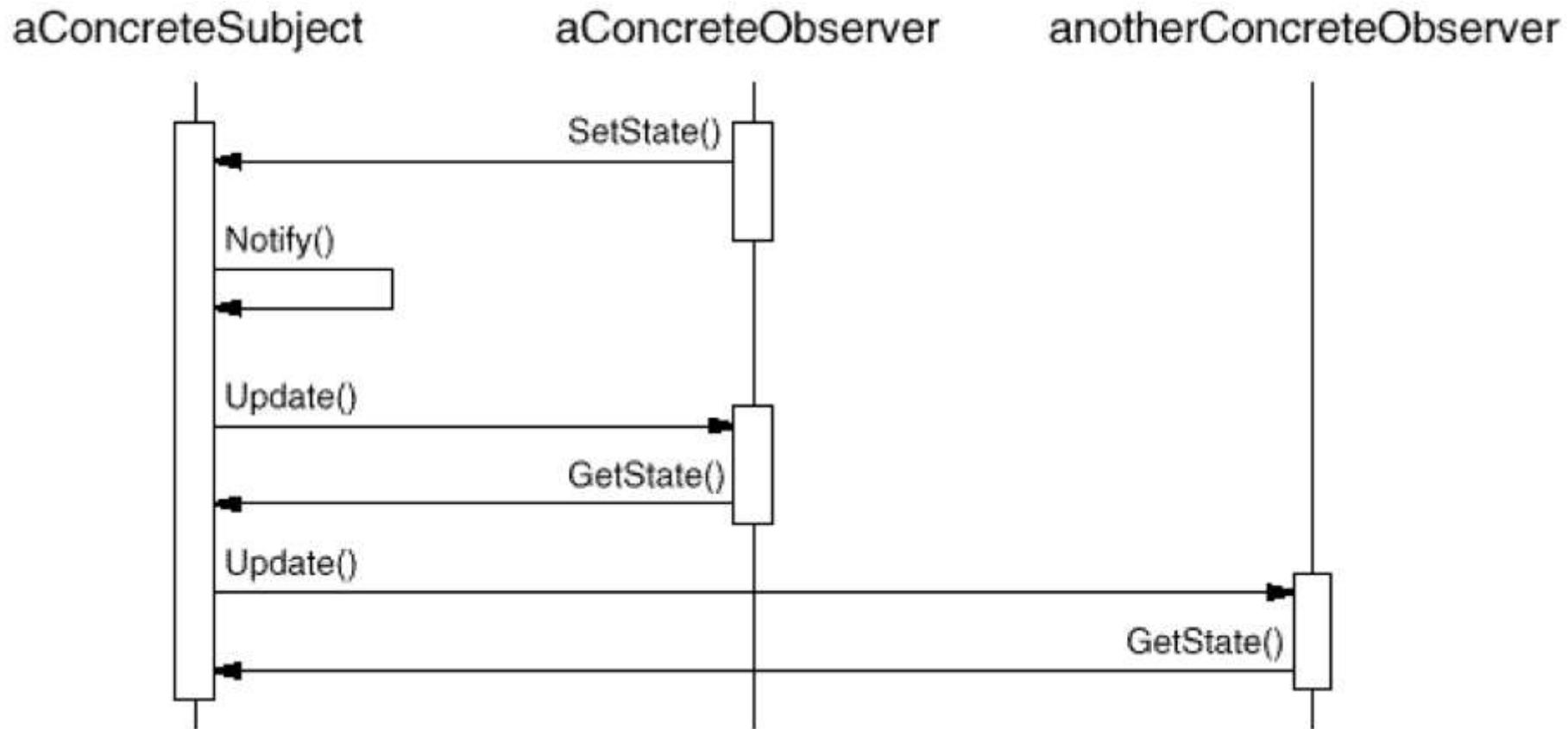
- przeznaczenie
 - określa zależność jeden do wielu między obiektami
 - kiedy zmieni się stan jednego obiektu, wszystkie zależne od niego są automatycznie powiadamiane i aktualizowane
- uzasadnienie
 - niepożądane osiąganie spójności między powiązanymi obiektami przez tworzenie ścisłe powiązanych klas
 - np. dane prezentowane w arkuszu i na wykresie
 - niepożądane powiązanie arkusza i wykresu – trudne ponowne użycie
 - lepiej powiązać każdy sposób prezentacji (Obserwator) z obiektem danych (Podmiot)

Obserwator – struktura

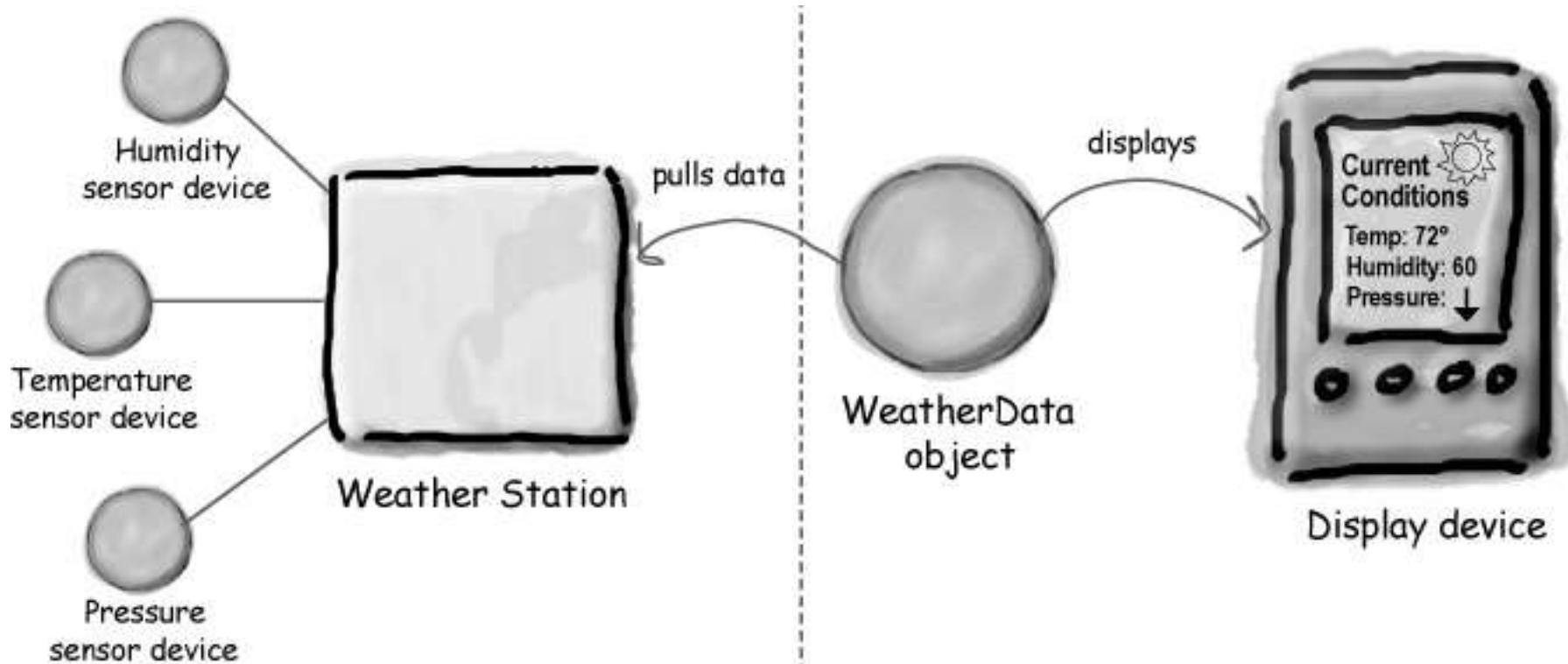


- **Podmiot** zna **Obserwatory**, udostępnia dołączanie i odłączanie
- **Obserwator** definiuje interfejs aktualizacji obiektów
- **Podmiot Konkretny** przechowuje stan i wysyła powiadomienie o jego zmianie
- **Obserwator Konkretny** przechowuje referencję do **Podmiotu**, stan podmiotu, implementuje aktualizację stanu (spójną ze stanem **Podmiotu Konkretnego**)

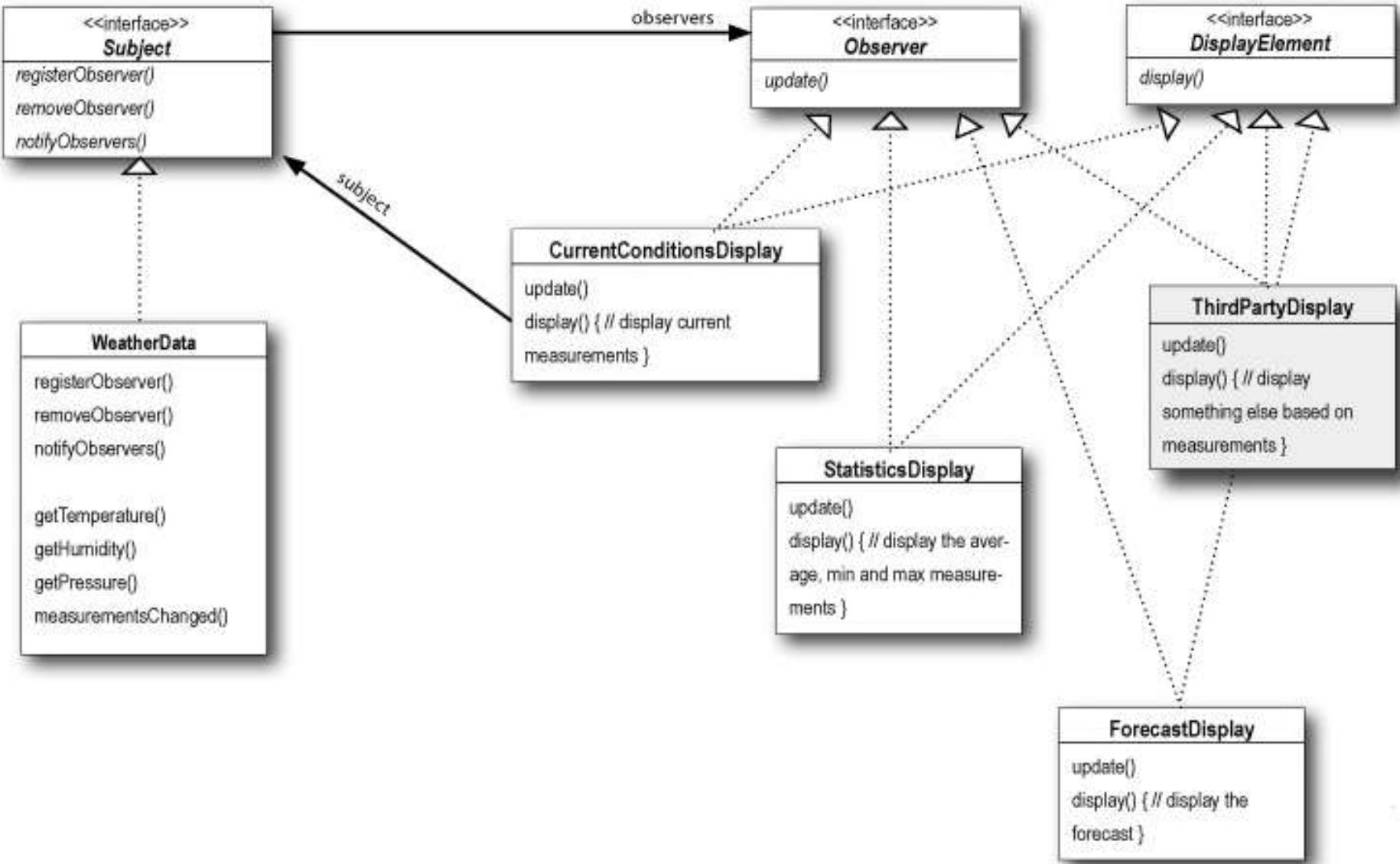
Obserwator – współdziałanie



Obserwator – przykład



Obserwator – przykład



Obserwator – przykład

- `headfirst.observer.weather`
 - Podmiot – Subject
 - Obserwator – Observer
 - DisplayElement
 - Konkretny Podmiot – WeatherData
 - Konkretny Obserwator – CurrentConditionsDisplay, StatisticsDisplay, ForecastDisplay
 - użycie: WeatherStation

rozbudowa – nowy sposób prezentacji danych:

- Konkretny Obserwator: HeatIndexDisplay
- użycie: WeatherStationHeatIndex

Obserwator – wnioski

- wzorzec definiuje relację jeden-do-wielu
- Podmioty aktualizują Obserwatorów za pomocą wspólnego interfejsu
- Obserwatory są luźno powiązane
 - brak powiązań z innymi Obserwatorami
 - Podmiot wie tylko, że Obserwator implementuje wspólny interfejs
- nie powinniśmy polegać na określonej kolejności powiadamiania

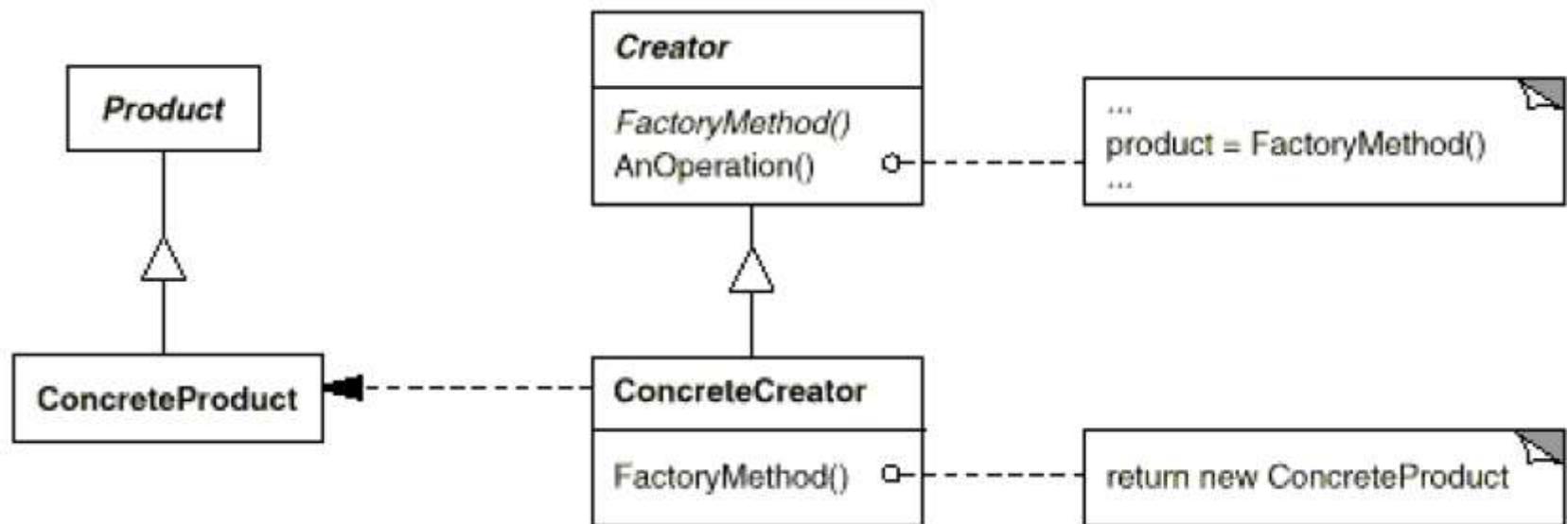
Metoda wytwórcza

		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	<u>Metoda wytwórcza</u>	Adapter	Interpreter Metoda szablonowa
	Obiekt	Fabryka abstrakcyjna Budowniczy Prototyp Singleton	Adapter Kompozyt Dekorator Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań Polecenie Mediator Pamiątka Obserwator Stan Strategia Odwiedzający

Metoda wytwórcza (konstruktor wirtualny)

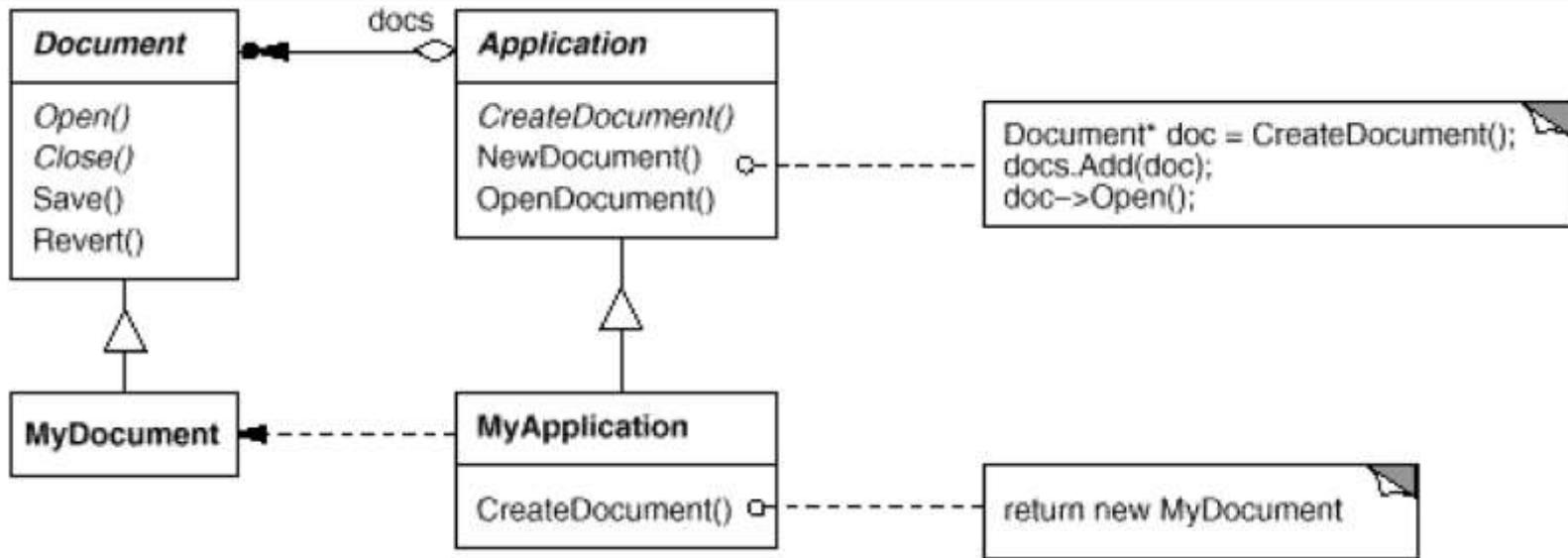
- przeznaczenie
 - określa interfejs do tworzenia obiektów umożliwiając podklasom wyznaczenie klasy danego obiektu
 - klasa przekazanie procesu tworzenia egzemplarzy podklasom
- uzasadnienie
 - klasa A ma utworzyć obiekt jednej z klas pochodnych klasy B, ale nie wie, której dokładnie
 - klasa pochodna klasy A podejmuje tę decyzję

Metoda wytwórcza – schemat



- **Product** definiuje interfejs obiektów generowanych przez metodę wytwórczą
- **ConcreteProduct** implementuje interfejs **Product**
- **Creator** zawiera deklarację metody wytwórczej zwracającej obiekty klasy **Product**
 - może zawierać implementację tej metody zwracającej domyślny obiekt **ConcreteProduct**
- **ConcreteCreator** przesyłania metodę wytwórczą, żeby zwracała obiekt klasy **ConcreteProduct**

Metoda wytwórcza – przykład

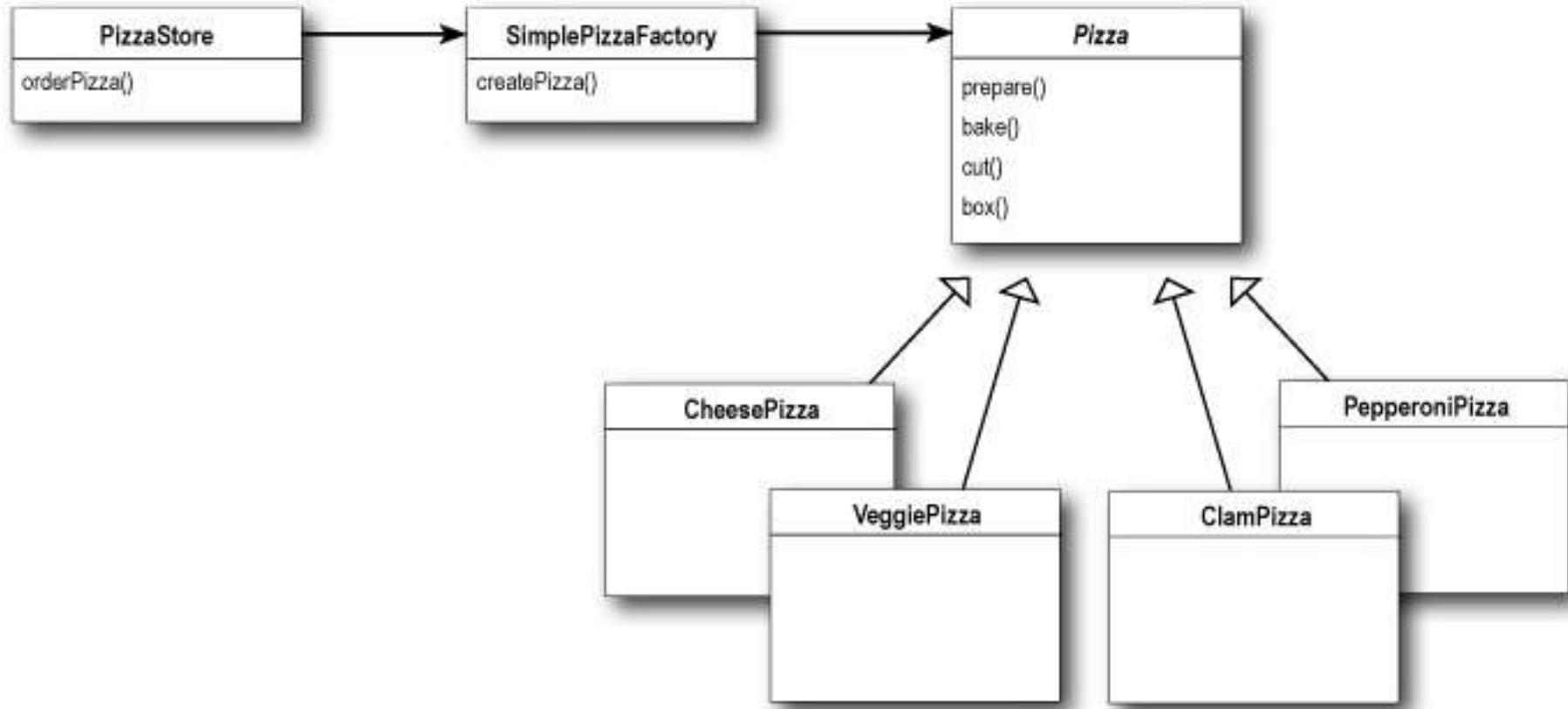


- **Application** tworzy **Document**, ale nie wie jaki konkretnie
- **MyApplication** tworzy konkretny **MyDocument**
 - np. DrawingApplication tworzy DrawingDocument

Metoda wytwórcza – konsekwencje

- metoda wytwórcza eliminuje potrzebę wiązania klas konkretnych dla aplikacji z jej kodem
 - jej kod odwołuje się jedynie do interfejsu Produkt → może działać z dowolną implementacją ConcreteProduct
- **wada:** czasem klient tworzy podklasę Creator tylko w celu wygenerowania określonego obiektu ConcreteProduct

Metoda wytwórcza – przykład



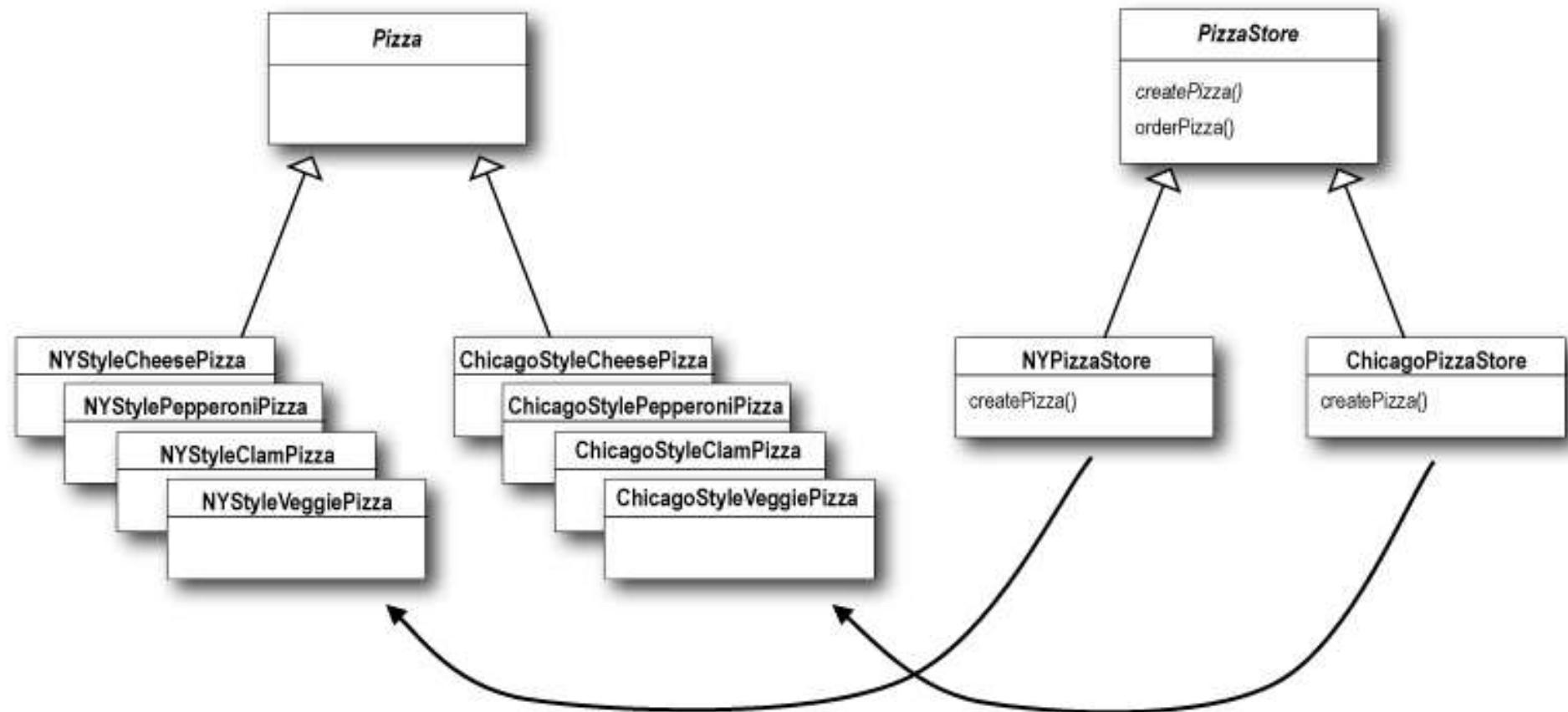
**Prosta Fabryka nie jest wzorcem projektowym,
ale często wykorzystywanym idiomem programowania**

Metoda wytwórcza – przykład – kod

- headfirst.factory.pizzas
 - Pizza – produkt
 - CheesePizza, PepperoniPizza, ... - konkretny produkt
 - PizzaStore – klient fabryki
 - SimplePizzaFactory – konkretna fabryka
 - PizzaTestDrive – main()

Metoda wytwórcza – przykład

konkretni twórcy (fabryki)



Metoda wytwórcza – przykład – kod

- `headfirst.factory.pizzafm`
 - `NYStyleXXXPizza`
 - `PizzaStore`
 - abstract class
 - abstract `createPizza()` – metoda wytwórcza
 - `NYPizzaStore`

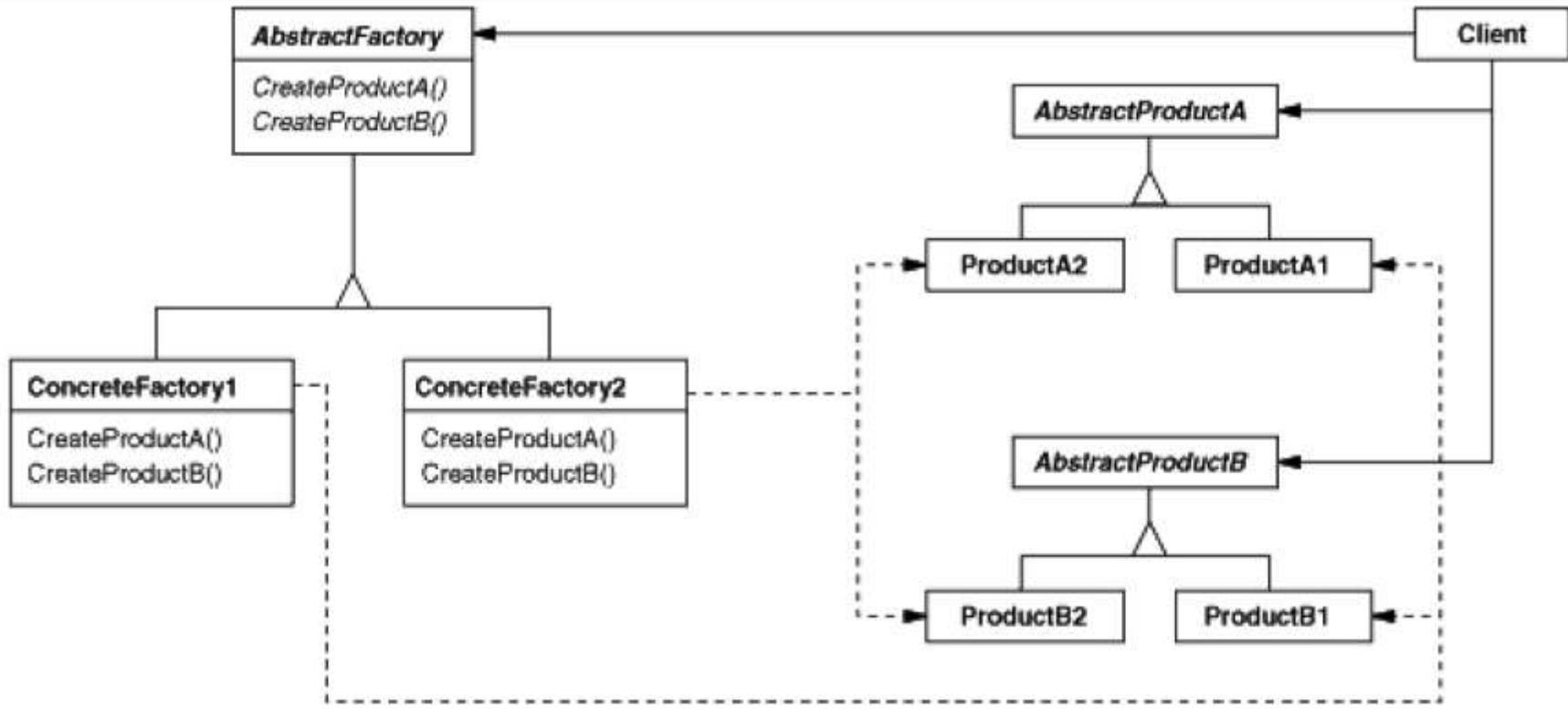
Fabryka abstrakcyjna

		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	Metoda wytwórcza	Adapter	Interpreter Metoda szablonowa
	Obiekt	<u>Fabryka abstrakcyjna</u> Budowniczy Prototyp Singleton	Adapter Kompozyt Dekorator Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań Polecenie Mediator Pamiątka Obserwator Stan Strategia Odwiedzający

Fabryka abstrakcyjna

- przeznaczenie
 - udostępnia interfejs do tworzenia rodzin powiązanych ze sobą lub zależnych od siebie obiektów bez określania ich klas konkretnych
- warunki stosowania
 - oddzielenie systemu od sposobu tworzenia, składania i reprezentowania jego Produktów
 - system ma być skonfigurowany za pomocą jednej z wielu rodzin produktów
 - powiązane obiekty (produkty) z jednej rodziny są zaprojektowane do wspólnego użytku i jednocześnie wykorzystywane

Fabryka abstrakcyjna – schemat



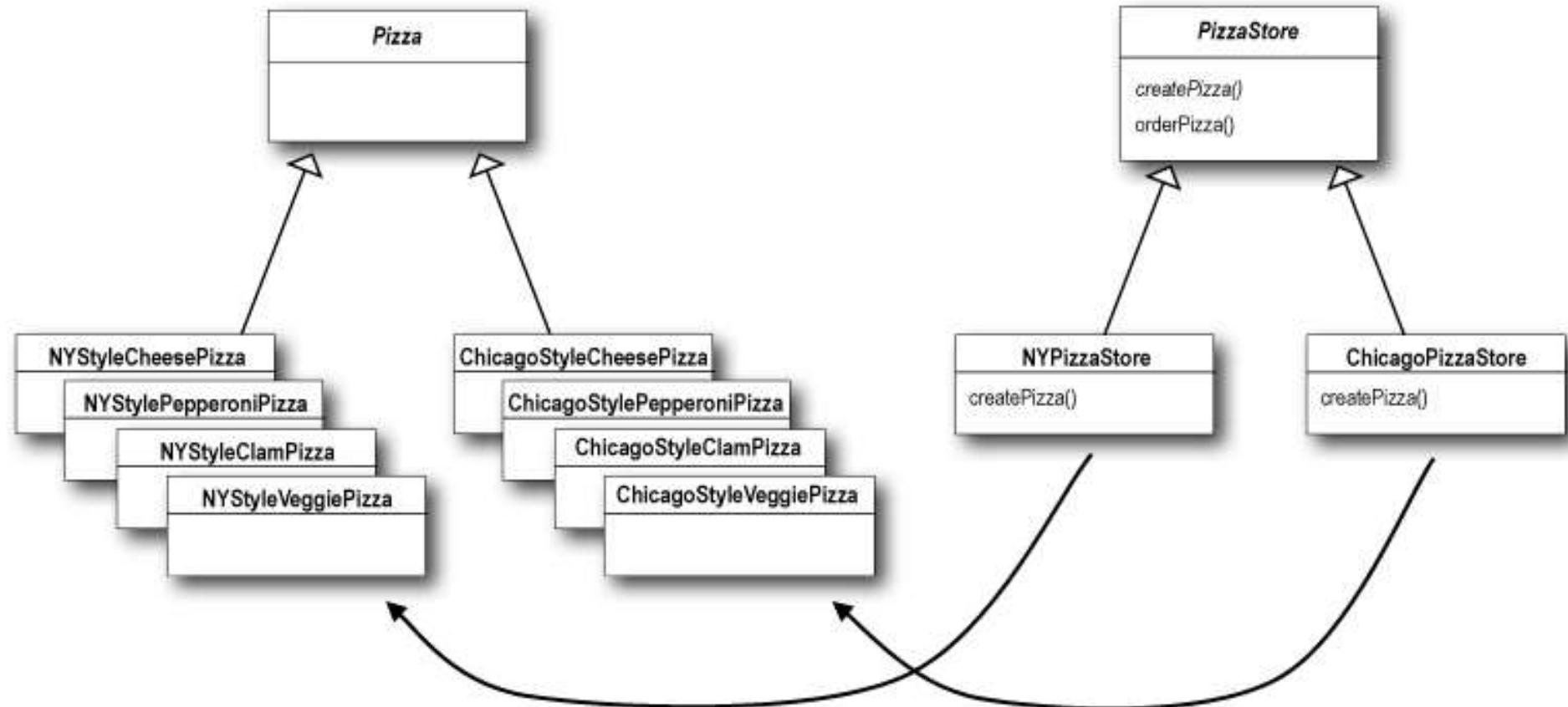
- **AbstractFactory** – interfejs z operacjami tworzącymi Produkty abstrakcyjne
- **ConcreteFactory** – implementacja operacji tworzących konkretnie produkty
- **AbstractProduct** – interfejs dla produktu abstrakcyjnego
- **Product** – obiekt tworzony przez konkretną fabrykę
- **Client** – korzysta jedynie z interfejsów!

Fabryka abstrakcyjna – przykład cz. 1/2

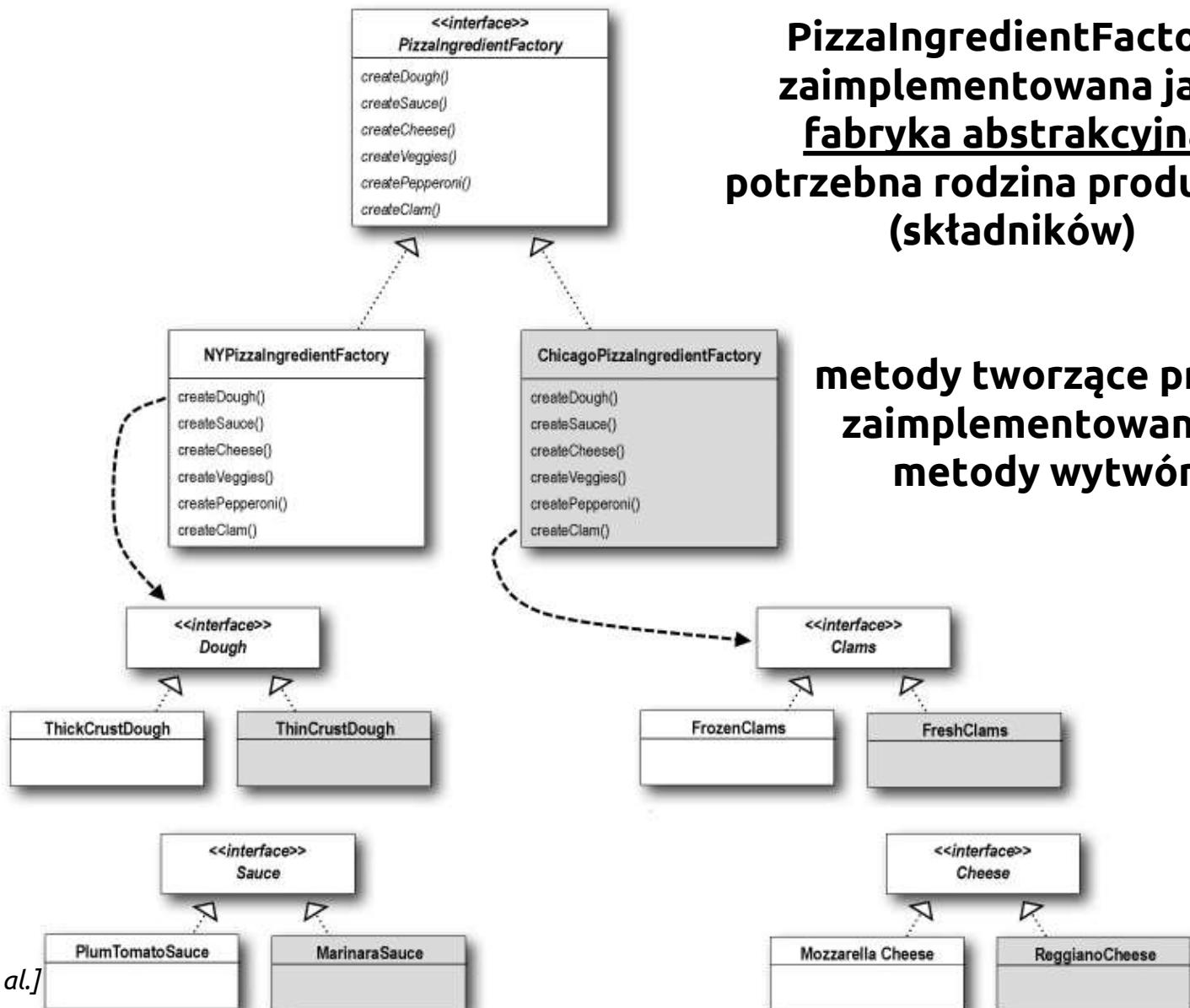
Rodzaj pizzy (pojedynczy produkt) w zależności od pizzerii → **metoda wytwórcza**

Różne składniki (wiele produktów) w zależności od rodzaju fabryki → **fabryka abstrakcyjna**

PizzaStore zaimplementowana jako metoda wytwórcza: `createPizza()`



Fabryka abstrakcyjna – przykład – cz. 2/2



PizzaIngredientFactory
zaimplementowana jako
fabryka abstrakcyjna:
potrzebna rodzina produktów
(składników)

metody tworzące produkty
zaimplementowane jako
metody wytwórcze

Fabryka abstrakcyjna – przykład – kod

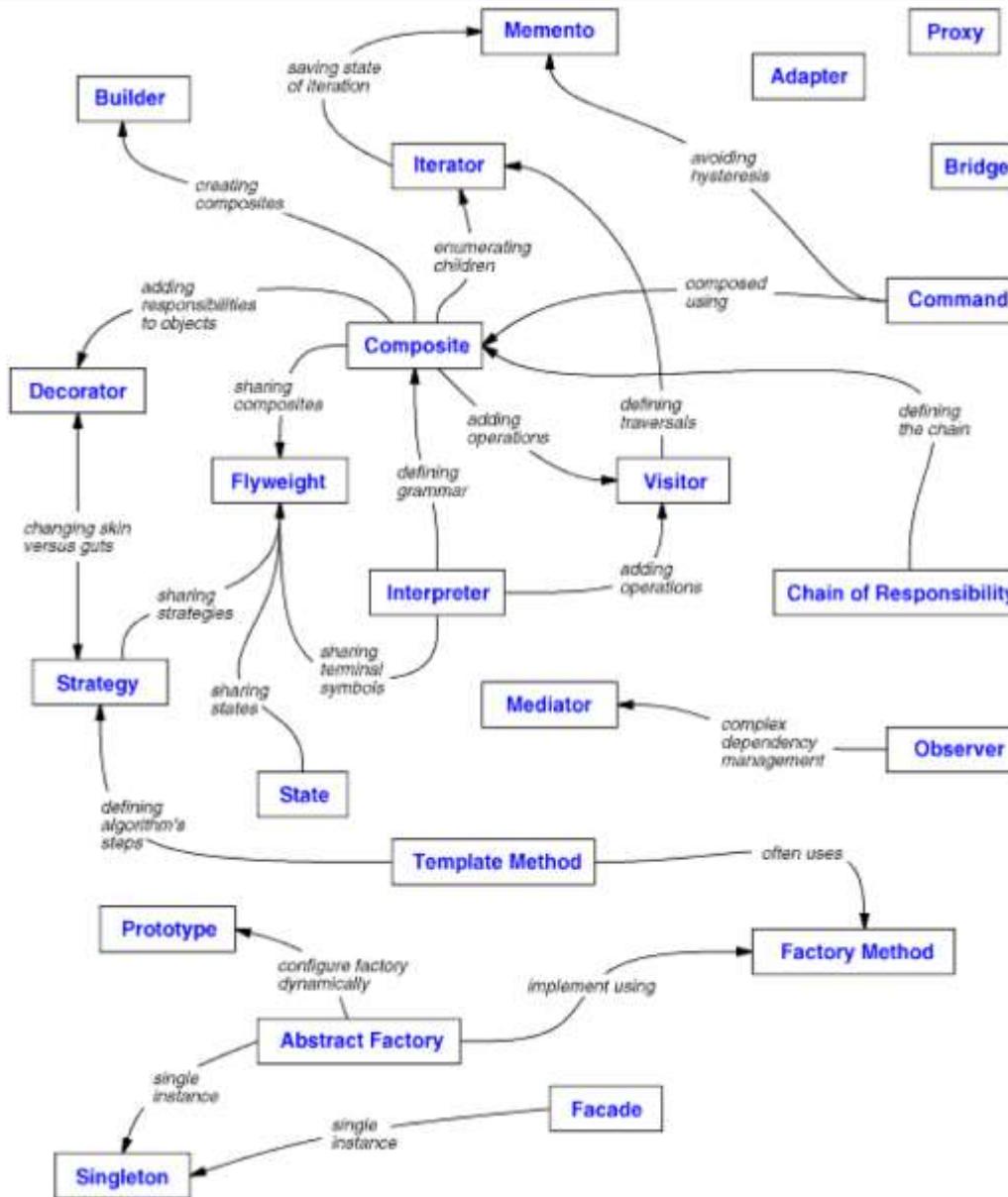
- `headfirst.factory.pizzaaf`
 - składniki
 - Dough, ThickCrustDough, ThinCrustDough
 - Sauce, MarinaraSauce, PlumTomatoSauce
 - Cheese, MozzarellaCheese, ParmesanCheese
 - ...
 - PizzaIngredientFactory
 - NYPizzaIngredientFactory, ChicagoPizzaIngredientFactory
 - Pizza – składniki, abstract prepare()
 - CheesePizza, PepperoniPizza – składniki, prepare()
 - PizzaStore – bez zmian
 - NYPizzaStore, ChicagoPizzaStore
 - PizzaTestDrive

Porównanie fabryk

- wszystkie hermetyzują tworzenie obiektów
- prosta fabryka
 - oddziela klienta od tworzenia konkretnych klas
- metoda wytwórcza
 - wybór i tworzenie obiektów konkretnego Produktu
- fabryka abstrakcyjna
 - wybór i utworzenie Fabryk (Twórców), z których każda potrafi wytwarzać określone Produkty

Zakończenie

Relacje między wzorcami projektowymi



Korzyści ze stosowania wzorców projektowych

sponsor projektu	kierownik projektu	architekt/inżynier systemu
<ul style="list-style-type: none">• promowanie ograniczenie czasu wytwarzania → krótszy harmonogram i wyższy ROI w porównaniu do projektów niekorzystających z wzorców	<ul style="list-style-type: none">• ograniczenie czasu projektowania• komponenty systemu o zweryfikowanej i potwierzonej jakości	<ul style="list-style-type: none">• ograniczenie czasu projektowania trzonu systemu, ale wciąż potrzebny czas na dostosowanie wzorców do wymagań projektowych
programista	tester	użytkownik
<ul style="list-style-type: none">• ponowne użycie kodu wygenerowanego na podstawie wzorca, ale potrzebna integracja zmian wskazanych przez architektów/inżynierów systemowych	<ul style="list-style-type: none">• możliwość dostosowania istniejących testów zaimplementowanych dla konkretnych wzorców do zmian wskazanych przez architektów/inżynierów systemowych	<ul style="list-style-type: none">• zwykłe szybsze dostarczenie oprogramowania• spodziewana zgodność z projektem (wymaganiami?) dzięki bazowaniu na sprawdzonej architekturze

Jak stosować wzorce projektowe?

Przeczytaj raz opis wzorca w ramach jego przeglądu

- szczególnie: Zastosowania i Konsekwencje

Cofnij się i przeanalizuj punkty Struktura, Elementy i Współdziałanie

- zrozumienie klas, obiektów i ich współdziałania

Zajrzyj do punktu Przykładowy kod, aby zapoznać się z przykładem zastosowania wzorca w kodzie

- jak zaimplementowany jest wzorzec

Wybierz dla elementów wzorca nazwy mające odpowiednie znaczenie w kontekście aplikacji

- ConcreteObserver → ForecastDisplay

Zdefiniuj klasy

- interfejsy, dziedziczenie, zmienne, wpływ na istniejące klasy

Ustal specyficzne dla aplikacji nazwy operacji używanych we wzorcu

- np. przedrostek Create dla metod wytwórczych

Zaimplementuj operacje, aby zrealizować zadania i zapewnić współdziałanie opisane we wzorcu

- pomoc: Implementacja i Przykładowy kod

Podsumowanie

		Rodzaj		
		Konstrukcyjne	Strukturalne	Operacyjne
Zasięg	Klasa	<u>Metoda wytwórcza</u>	<u>Adapter</u>	Interpreter Metoda szablonowa
	Obiekt	<u>Fabryka abstrakcyjna</u> Budowniczy Prototyp <u>Singleton</u>	<u>Adapter</u> Kompozyt <u>Dekorator</u> Fasada Pyłek Pełnomocnik	Łańcuch zobowiązań <u>Polecenie</u> Mediator Pamiątka <u>Obserwator</u> Stan Strategia Odwiedzający

Pytania



Literatura

- Freeman E., Robson E., Bates B., Sierra K., Head First Design Patterns, O'Reilly Media, Inc. 2004
- Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional 2004
- Schmidt D., Otte W., CS 251: Intermediate Software Design, 2014, <http://www.dre.vanderbilt.edu/~schmidt/cs251/>
- DEV475 Mastering OOAD with UML 2.0, IBM, 2004
- Schmidt D., LiveLessons: Design Patterns in Java, 2014, <http://www.dre.vanderbilt.edu/~schmidt/LiveLessons/>
- Bruegge B., Dutoit A. H, Inżynieria oprogramowania w ujęciu obiektowym. UML, wzorce projektowe i Java, Helion 2011

Następny wykład

Zarządzanie konfiguracją

Inżynieria oprogramowania

Wykład 7:

Zarządzanie konfiguracją

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Agenda

- Wprowadzenie do systemów kontroli wersji
- Wprowadzenie do Git
- Prezentacja na żywo

Po co nam kontrola wersji?

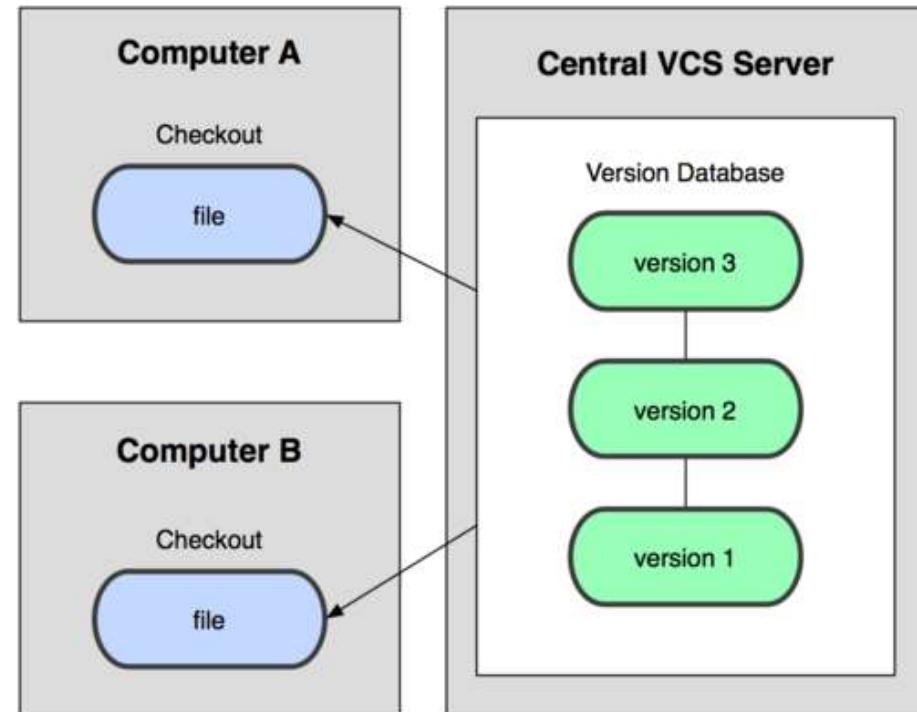
- można przechowywać pliki w systemie plików
 - np. z podfolderami z nazwą wersji / datą modyfikacji
- ale lepiej skorzystać z systemu kontroli wersji
- Dlaczego?
 - przywrócenie do wcześniejszej wersji
 - odtworzenie stanu całego projektu
 - zbadanie kto i kiedy wprowadził modyfikację
 - odzyskanie danych

Scentralizowane
Rozproszone

Rodzaje systemów kontroli wersji

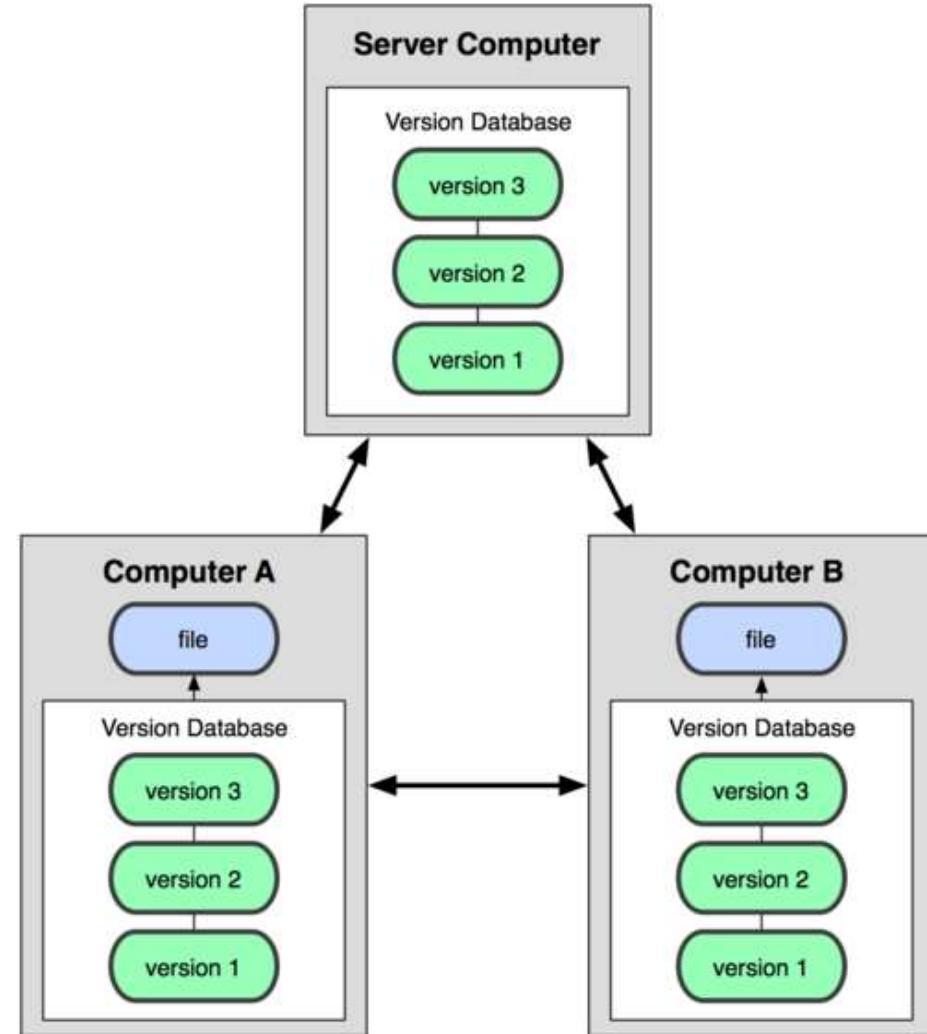
Scentralizowane systemy kontroli wersji

- np. CVS, Subversion
- Zalety:
 - śledzenie prac poszczególnych osób
 - uprawnienia dla użytkowników
- Wady:
 - awaria serwera
 - brak dostępu do repozytorium
- rozwiążanie → systemy rozproszone



Rozproszone systemy kontroli wersji

- np. Git, Mercurial
- klienci posiadają kopie całego repozytorium
- w przypadku awarii serwera
 - repozytorium klienta można skopiować na serwer
- możliwość wielu zdalnych repozytoriów
 - większe możliwości dostosowania schematu współpracy



Git

Historia Git

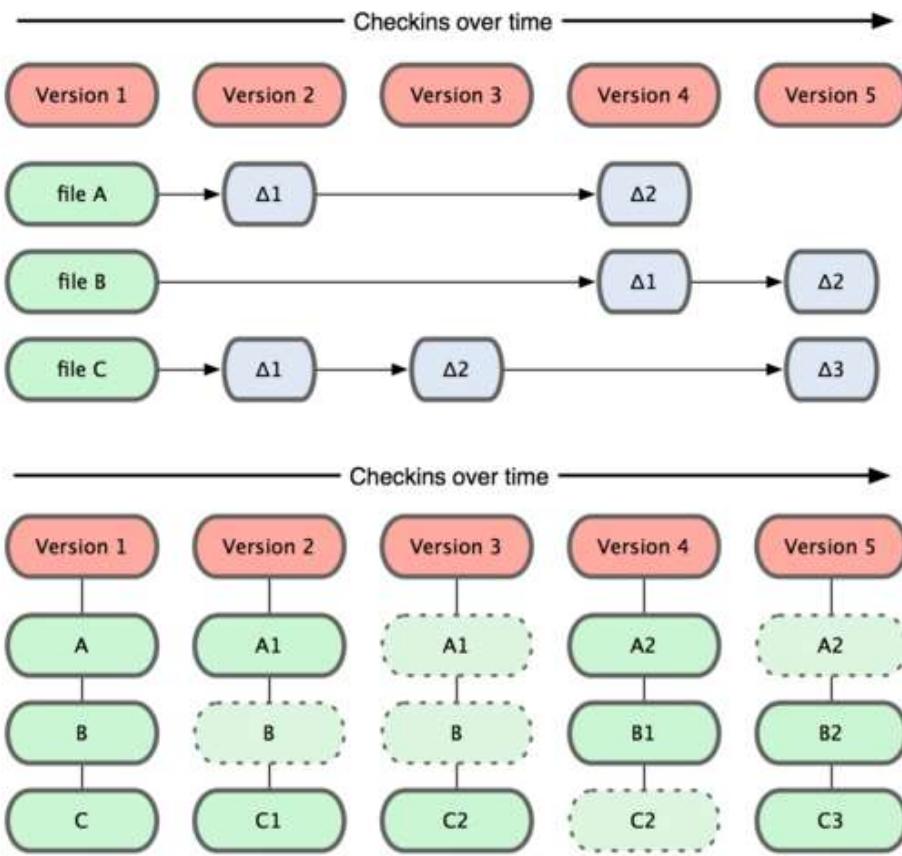
- prace nad jądrem Linuksa
 - 1991-2002 – zmiany w źródle przekazywane jako łaty (patch) i zarchiwizowane pliki
 - w 2002 – rozpoczęto korzystanie z BiTKeeper rozproszonego VCS
 - w 2005 wycofano pozwolenie na bezpłatne korzystanie z BitKeepera
 - programiści Linuksa (w szczególności Linus Torvalds) podjęli decyzję o budowie własnego systemu
 - cele nowego systemu:
 - szybkość
 - prosta konstrukcja
 - silne wsparcie dla nieliniowego rozwoju (tysiący równoległych gałęzi)
 - pełne rozproszenie
 - wydajna obsługa dużych projektów (jądro Linuksa) – szybkość i rozmiar danych

Cechy Git

- Git przechowuje dane inaczej niż inne wcześniejsze systemy!
- Cechy:
 - migawki zamiast różnic
 - operacje są lokalne
 - mechanizm spójności danych
 - dodawanie nowych danych
 - trzy stany

Cechy Git – migawki zamiast różnic

- większość systemów przechowuje dane jako zbiór plików i zmian na nich dokonanych
- Git przechowuje obrazy (migawki – snapshot) plików z danej chwili
 - w celu wydajności Git nie zapisuje plików niezmienionych, ale referencje do poprzednich wersji



Cechy Git – operacje są lokalne

- większość operacji wymaga jedynie dostępu do lokalnych plików
 - nie są potrzebne żadne dane przechowywane w sieci
- to istotny aspekt zwiększający szybkość
 - kompletne repozytorium przechowywane lokalnie
 - więc brak narzutu połączeń sieciowych
- działa również przy braku dostępu do sieci
 - w innych systemach wiele operacji nie jest możliwych
 - np. zapisywanie zmian do repozytorium

Cechy Git – mechanizm spójności danych

- przed zapisem Git wylicza sumę kontrolną dla każdego obiektu
 - na podstawie tej sumy można odwoływać się do obiektu
 - nie można zmienić zawartości pliku / katalogu bez reakcji Git
- efekt:
 - brak możliwości utraty informacji lub uszkodzenie zawartości pliku podczas przesyłania lub pobierania danych
- jak?
 - Git wykorzystuje skrót SHA-1
 - 40 znaków (a-z, 0-9) na podstawie zawartości pliku/katalogu, np.
24b9da6552252987aa493b52f8696cd6d3b00373
 - baza Git → klucz-wartość
 - klucz – skrót SHA-1
 - wartość – zawartość pliku / struktura katalogu

Cechy Git – dodawanie nowych danych

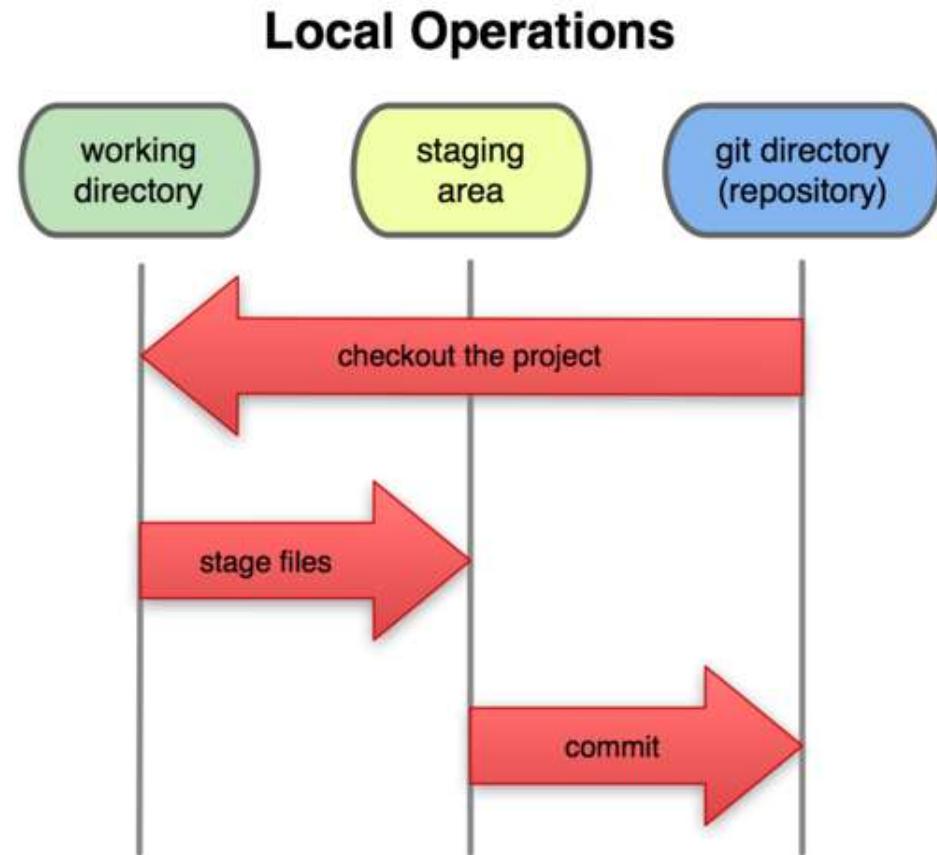
- standardowo Git wyłącznie dodaje nowe dane
- trudno jest zmusić Git do wykonania operacji, z której nie można się wycofać, tzn.
 - można stracić / nadpisać zmiany nie zatwierdzone
 - ale: trudno stracić zmiany
 - zwłaszcza przy regularnym przesyłaniu do zdalnego repozytorium
- efekt:
 - większe bezpieczeństwo i poczucie bezpieczeństwa
 - eksperymentowanie bez ryzyka zepsucia czegokolwiek

Cechy Git – trzy stany

- zatwierdzony
 - dane zostały zachowane w lokalnym repozytorium
- zmodyfikowany
 - dane zmienione, ale nie zachowane w repozytorium
- śledzony
 - plik przeznaczony do zatwierdzenia w bieżącej postaci przy następnej operacji zatwierdzania
- z tego wynikają trzy sekcje projektu
 - katalog Git
 - katalog roboczy
 - przechowalnia

Cechy Git – trzy stany

- katalog Git (*Git repository*)
 - metadane i baza projektu
 - kopiowany przy klonowaniu projektu z innego komputera
- katalog roboczy (*working directory*)
 - obraz jednej wersji projektu
 - pliki rozpakowywane umieszczane na dysku do odczytu/zapisu
- przechowalnia (*staging area*)
 - zawiera dane do zatwierdzenia w następnej operacji commit



Wstępna konfiguracja Git

- za pomocą narzędzia git config z odpowiednimi argumentami
- pierwsza rzecz po instalacji
 - konfiguracja nazwy użytkownika i adresu email

```
$ git config --global user.name "Jan Nowak"
```

```
$ git config --global user.email jannowak@example.com
```
 - global informuje, żeby Git od tej pory korzystał z tych danych
 - w razie potrzeby modyfikacji – bez opcji global

Wstępna konfiguracja Git

- edytor
 \$ git config --global core.editor emacs
- narzędzie obsługi różnic
 \$ git config --global merge.tool vimdiff
- inne narzędzia obsługi różnic
 - kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, opendiff
- sprawdzanie ustawień
 \$ git config -list
 user.name=Scott Chacon
 user.email=schacon@gmail.com
 color.status=auto
 color.branch=auto
 color.interactive=auto
 color.diff=auto

 ...

Uzyskiwanie pomocy

- trzy sposoby wyświetlenia strony podręcznika

```
$ git help <polecenie>
```

```
$ git <polecenie> --help
```

```
$ man git-<polecenie>
```

- na przykład

```
$ git help config
```

Podstawy korzystania z Gita

Tworzenie repozytorium

- Dwa sposoby:

- inicjalizacja Gita w istniejącym katalogu

- ```
$ git init
```

- tworzy nowy podkatalog o nazwie .git – k szkielet repozytorium Gita

- na razie żadna część projektu nie jest jeszcze śledzona

- rozpoczęcie kontroli wersji, na przykład

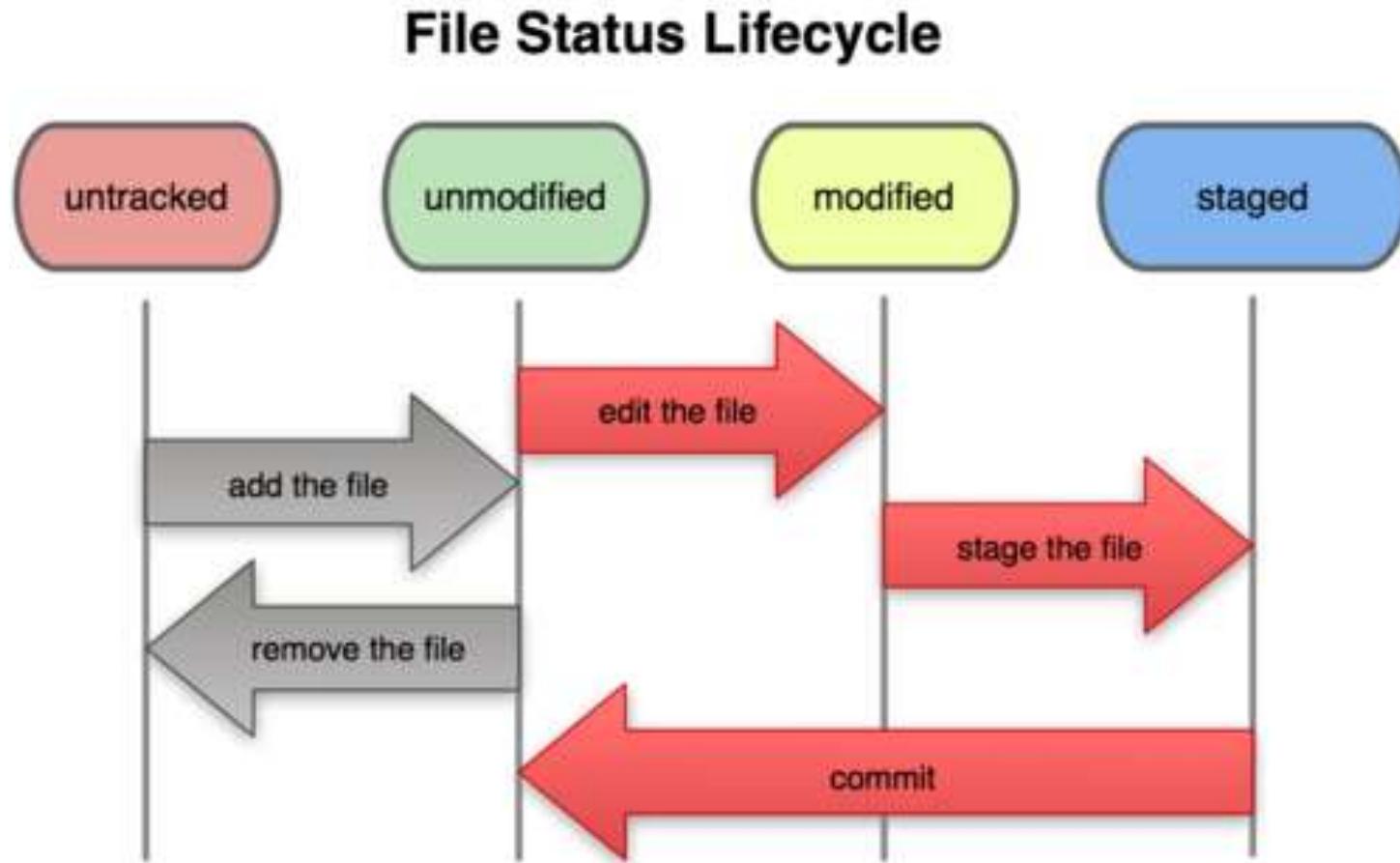
- ```
$ git add *.c
```

- ```
$ git add README
```

- ```
$ git commit -m 'initial project version'
```

- klonowanie istniejącego repozytorium

Rejestrowanie zmian w repozytorium



Prezentacja na żywo

Pytania



Literatura

- Chacon S., **Pro Git**, 2009
<http://git-scm.com/book>
<http://git-scm.com/book/pl/>
- Lanubile F., Ebert C., Prikladnicki R., Vizcaíno A.,
Collaboration Tools for Global Software Engineering,
IEEE Software, 2010, 27: 52–55, doi: 10.1109/MS.2010.39
- Spinellis D., **Git**. IEEE Software, 2012, 29: 100–101,
doi:10.1109/MS.2012.61
- Dudler R., **Git – the simple guide**
<http://rogerdudler.github.com/git-guide/>
- **Git Immersion**
<http://gitimmersion.com/>
- **Code School, Try Git**
<http://www.codeschool.com/courses/try-git>
- **Git User Guide**
<http://netbeans.org/kb/docs/ide/git.html>
- Vogel L., **Git Tutorial**
<http://www.vogella.com/articles/Git/article.html>

Następny wykład

Zapewnienie jakości oprogramowania

Inżynieria oprogramowania

Wykład 8: Zapewnienie jakości oprogramowania

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Agenda

Po co zapewniać jakość oprogramowania?

Jakość produktu

Miary jakości produktu

Projektowanie miar jakości

Przeglądy i inspekcje

Jakość procesu

Plan zapewnienia jakości

Po co zapewniać jakość oprogramowania?



- oprogramowanie spełnia potrzeby informacyjne użytkownika/klienta
- oprogramowanie jest odpowiednio wydajne
- oprogramowanie można łatwo obsługiwać
- ... ale również aspekty bardziej poważne ...

Po co zapewniać jakość oprogramowania?

Ariane 5, lot 501 – 4/6/1996



- niedostateczne zabezpieczenie przed przekroczeniem zakresu liczb całkowitych
 - konwersja z 64-bitowego float do 16-bitowego signed integer
- brak ofiar
- ale opóźnienie kolejnych lotów
 - straty > \$370 mln

Po co zapewniać jakość oprogramowania?

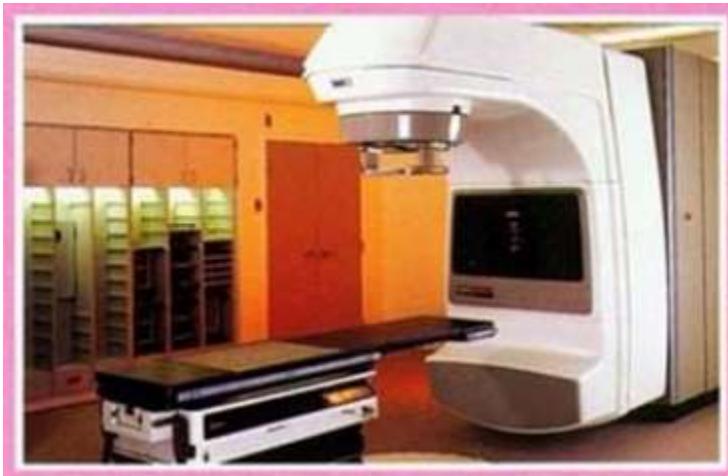
Airbus A320 - Air France 296Q - 26/6/1988



- pokazy lotnicze na lotnisku Miluza – Habsheim
 - lot pokazowy na małej wysokości
- na pokładzie zaproszeni goście i dziennikarze (130 pasażerów + 6 osób załogi)
- nie do końca wyjaśnione przyczyny
 - oficjalnie: "za nisko, za wolno, za późno" → błąd pilota
 - ale organizatorzy uznani później za współwinnych
- **automatyczne systemy lotu uniemożliwiły pilotowi przejęcie kontroli nad sterami**
- 3 ofiary śmiertelne, ok. 50 osób rannych

Po co zapewniać jakość oprogramowania?

Therac 25 – 1985-1987



<http://hci.cs.siu.edu/NSF/Files/Semester/Week13-2/PPT-Text/Slide13.html>

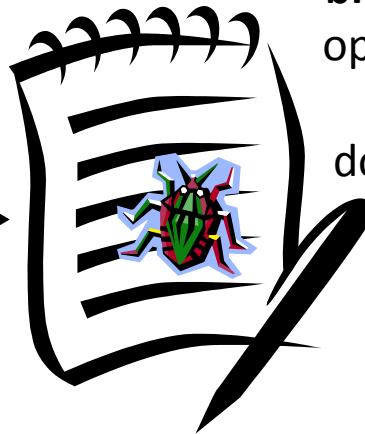
- urządzenie do terapii radiacyjnej
 - skoncentrowana wiązka
 - niszczenie komórek rakowych
- wymagany przeszkolony personel
- 6 pacjentów narażonych na nadmierne promieniowanie
 - część zmarła
- problem pojawiał się przy jednoczesnym wystąpieniu określonych warunków
 - w szczególności szybkiej obsługi urządzenia
- obsługa za pomocą specjalnej klawiatury
 - w czasie testów zapoznawanie się z urządzeniem → powolna obsługa
 - po pewnym czasie eksploatacji → szybsza obsługa ujawniała defekt

Błąd – defekt - awaria



deweloper
(uczestnik projektu)

błąd człowieka
może
doprowadzić do



defekt
(usterka)

błędne działanie
oprogramowania
może
doprowadzić do



awaria

Pierwszy defekt w historii

- **Harvard Mark II** – komputer elektromechaniczny zbudowany na Harvardzie w 1947 na potrzeby marynarki USA
- **9/9/1947**
 - ćma utknęła w przekaźniku, zwęgiła się, spowodowała zwarcie
 - w efekcie – awaria urządzenia
- technicy wyjęli ćmę z przekaźnika
 - pierwszy robak (**bug**) znaleziony w komputerze
 - obecnie wystawiony Muzeum Instytutu Smithsona w Waszyngtonie



9/9

0800 Antran started
1000 " stopped - antran ✓
13° UC (032) MP - MC { 1.2700 9.037 847 025
033 PRO 2 2.130476415 9.037 846 995 const
const 2.130676415
Relays 6-2 in 033 failed special speed test
in relay 10.000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
1630 First actual case of bug being found.
1700 closed down.

Agenda

Po co zapewniać jakość oprogramowania?

Jakość produktu

Miary jakości produktu

Projektowanie miar jakości

Przeglądy i inspekcje

Jakość procesu

Plan zapewnienia jakości

Czym jest jakość oprogramowania?

- subiektywne zadowolenie klienta
 - jeśli proces wytwórczy jest dobry, to klient będzie zadowolony
- obiektywnie określona jakość
 - zespół cech, które można określić ilościowo i mierzyć za pomocą miar

Norma ISO 9126

- perspektywa wewnętrzna (*internal quality*)
 - cechy dające się ocenić na podstawie produktów pośrednich powstających w procesie wytwórczym
- perspektywa zewnętrzna (*external quality*)
 - cechy postrzegane podczas oceny gotowego produktu
- perspektywa użytkowa (*quality in use*)
 - cechy określające stopień spełnienia biznesowych potrzeb użytkownika

Norma ISO 9126 i ISO 25000

- 3 stopnie modelu jakości
 - cechy (*characteristics*)
 - właściwości (*subcharacteristics*)
 - miary/metryki (*metrics*)
- norma nie mówi jak osiągnąć jakość

Model jakości ISO 25000:2005

- stopień, w jakim oprogramowanie zaspokaja przedstawione i implikowane potrzeby, gdy jest wykorzystywane w określonych warunkach

Model jakości ISO 25000:2005

- dopasowanie funkcjonalne
- niezawodność
- łatwość użycia
- wydajność
- zabezpieczenie
- kompatybilność
- łatwość konserwacji
- przenośność
- **użyteczność**
- **bezpieczeństwo**
- **elastyczność**

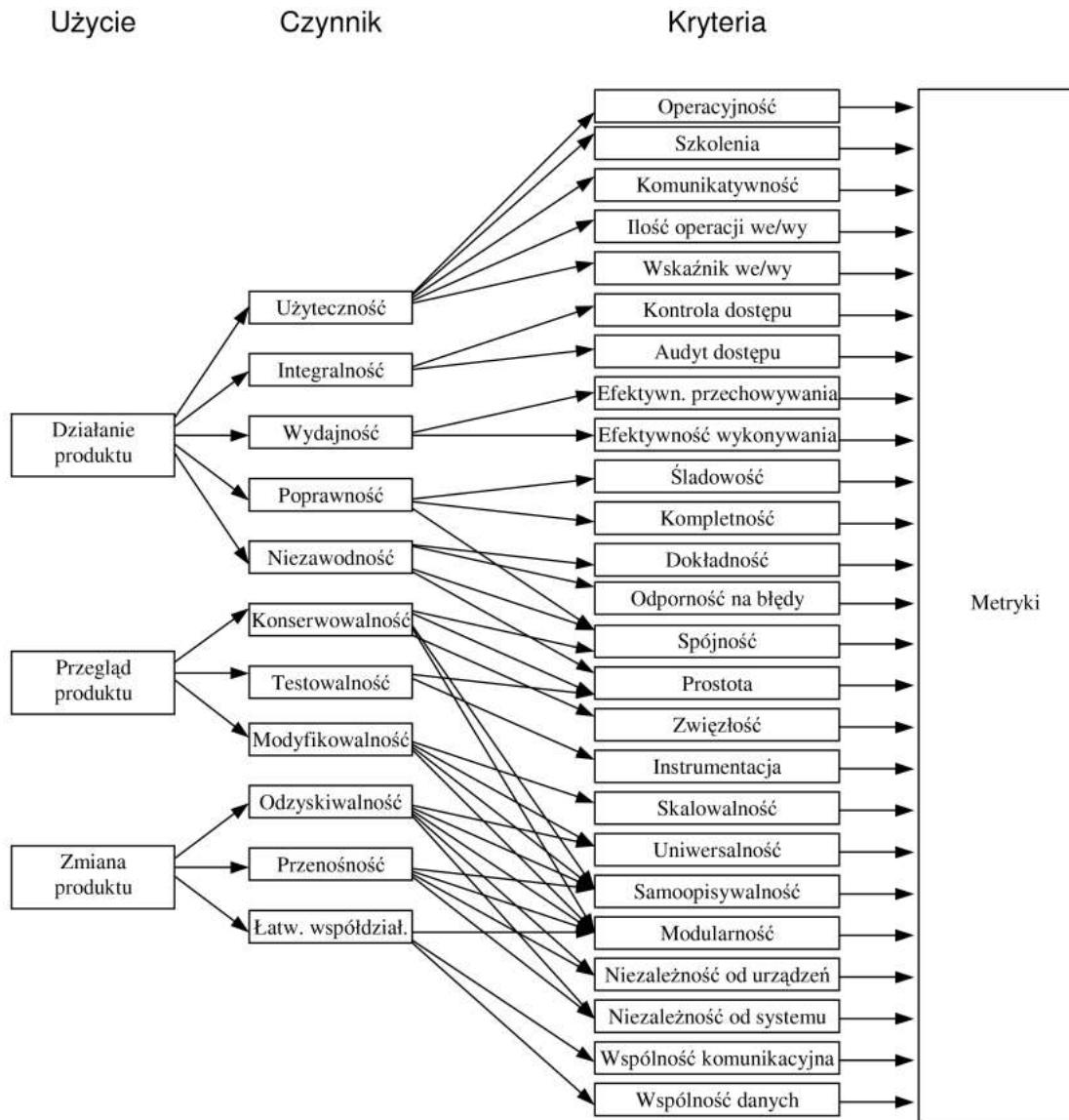
Model jakości ISO 25000:2005

functional suitability <ul style="list-style-type: none">• functional appropriateness• accuracy• suitability compliance	reliability <ul style="list-style-type: none">• maturity• availability• fault tolerance• recoverability• reliability compliance	performance efficiency <ul style="list-style-type: none">• time behavior• resource utilization• performance efficiency compliance	operability <ul style="list-style-type: none">• appropriateness recognisability• learnability• ease of use• attractiveness• technical accessibility• operability compliance	security <ul style="list-style-type: none">• confidentiality• integrity• non-repudiation• accountability• security compliance	compatibility <ul style="list-style-type: none">• co-existence• interoperability• compatibility compliance
maintainability <ul style="list-style-type: none">• modularity• reusability• analyzability• changeability• modification stability• testability• maintainability compliance	portability <ul style="list-style-type: none">• adaptability• installability• replaceability• portability compliance	usability <ul style="list-style-type: none">• effectiveness• efficiency• satisfaction• usability compliance	flexibility <ul style="list-style-type: none">• context conformity• context extendibility• accessibility• flexibility compliance	safety <ul style="list-style-type: none">• operator health and safety• commercial damage• public health and safety• environmental harm• safety compliance	

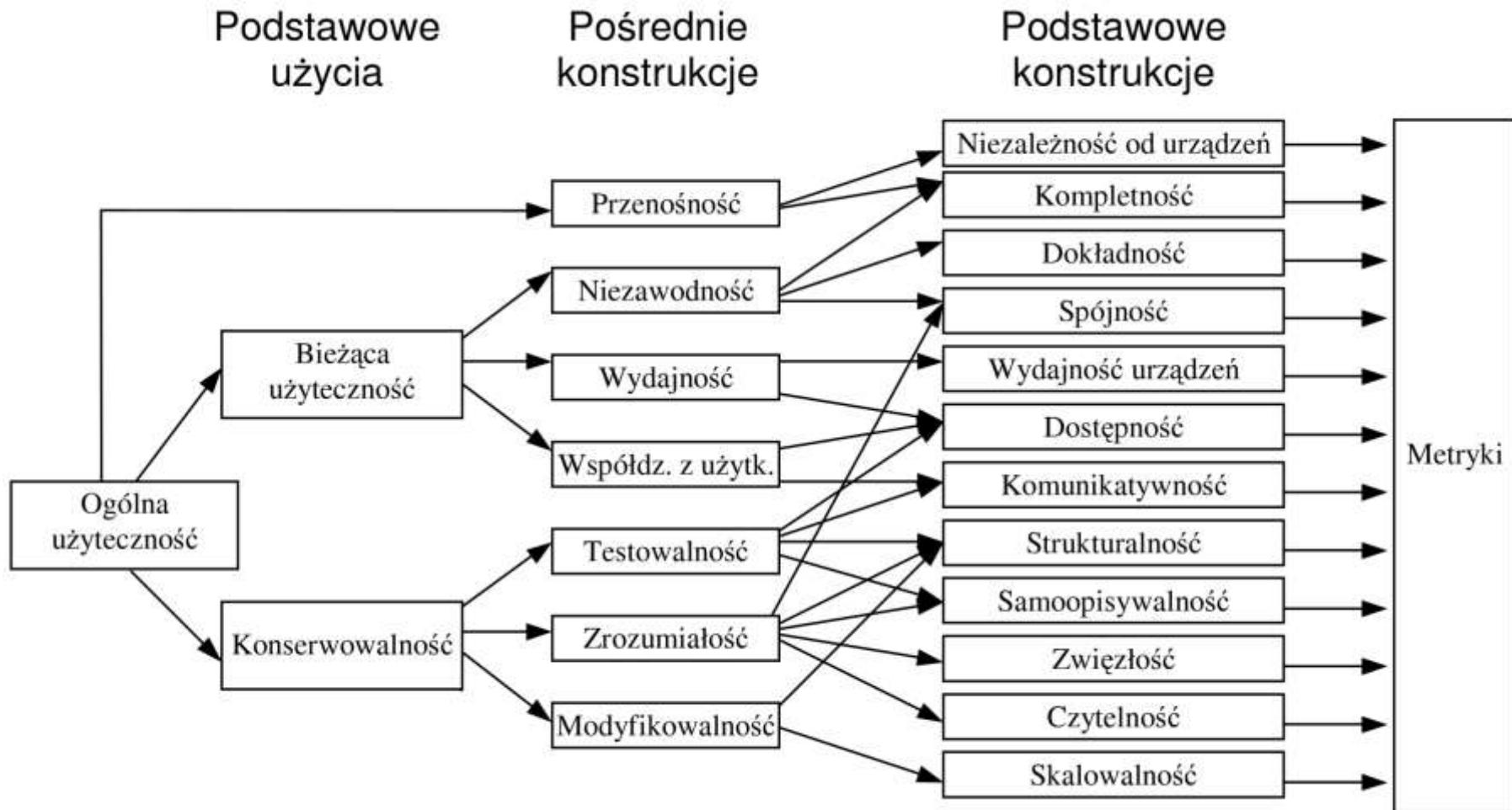
Model jakości ISO/IEC 25010:2011

Functional suitability <ul style="list-style-type: none">• Functional completeness• Functional correctness• Functional appropriateness	Performance efficiency <ul style="list-style-type: none">• Time behaviour• Resource utilisation• Capacity	Compatibility <ul style="list-style-type: none">• Co-existence• Interoperability	Transferability <ul style="list-style-type: none">• Adaptability• Installability• Replaceability
Usability <ul style="list-style-type: none">• Appropriateness recognizability• Learnability• Operability• User error protection• User interface aesthetics• Accessibility	Reliability <ul style="list-style-type: none">• Maturity• Availability• Fault tolerance• Recoverability	Security <ul style="list-style-type: none">• Confidentiality• Integrity• Non-repudiation• Accountability• Authenticity	Maintainability <ul style="list-style-type: none">• Modularity• Reusability• Analyzability• Modifiability• Testability
Effectiveness <ul style="list-style-type: none">• Effectiveness	Efficiency <ul style="list-style-type: none">• Efficiency	Satisfaction <ul style="list-style-type: none">• Usefulness• Trust• Pleasure• Comfort	Freedom from risk <ul style="list-style-type: none">• Economic risk mitigation• Health and safety risk mitigation• Environmental risk mitigation
			Context coverage <ul style="list-style-type: none">• Context completeness• Flexibility

Inne modele jakości McCalla (1977)



Inne modele jakości Boehma (1978)

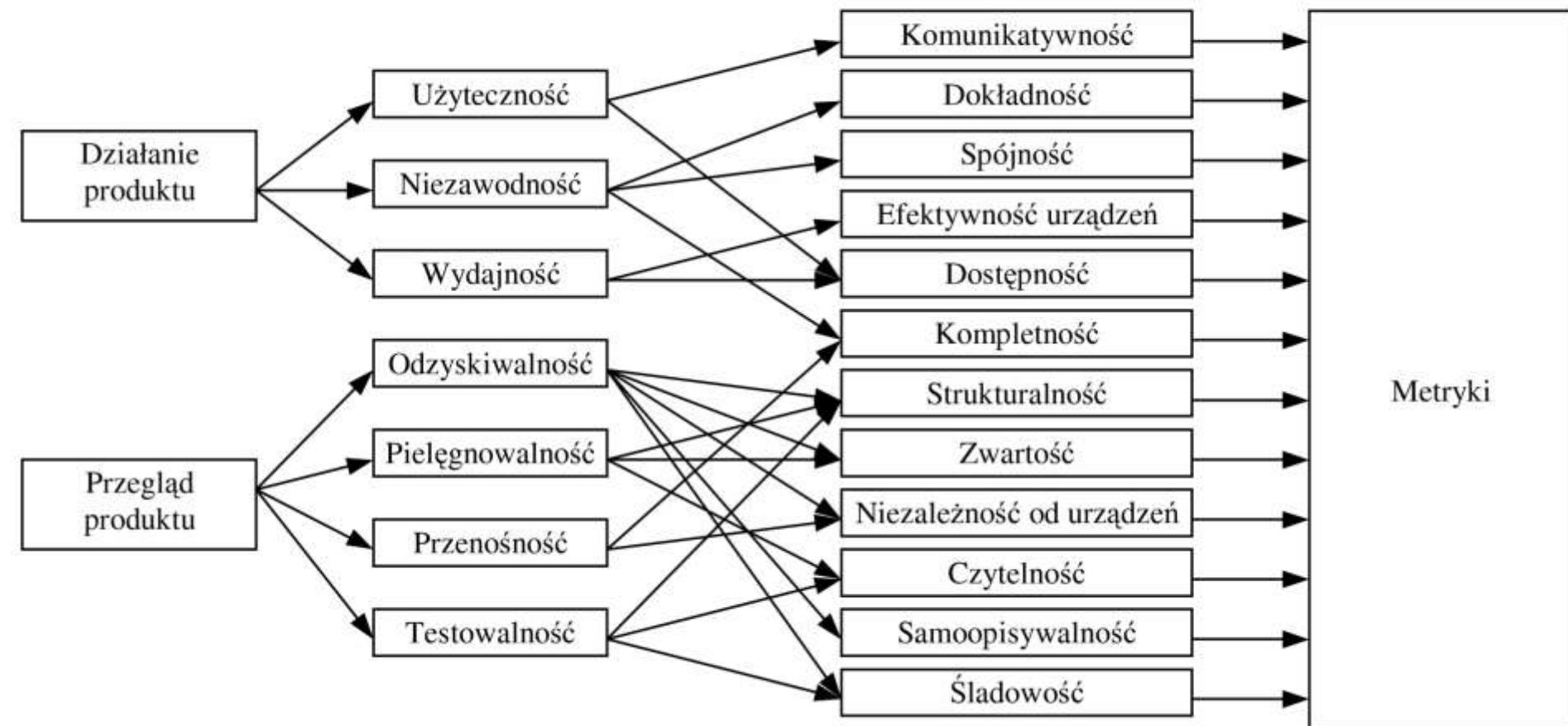


Inne modele jakości Fentona (1997)

Użycie

Czynnik

Kryteria



Agenda

Po co zapewniać jakość oprogramowania?

Jakość produktu

Miary jakości produktu

Projektowanie miar jakości

Przeglądy i inspekcje

Jakość procesu

Plan zapewnienia jakości

Miary pokrycia kodu

- analiza pokrycia kodu jest elementem testowania białej skrzynki
 - wykorzystywana znajomość wewnętrznej struktury programu
- **wada** – brak jednolitej definicji pokrycia
 - różne miary pokrycia
- **zaleta** – łatwość interpretacji
 - wartości zmieniają się w przedziale 0% – 100%
- rodzaje pokrycia
 - pokrycie bloków instrukcji
 - pokrycie decyzji
 - pokrycie ścieżek

Pokrycie bloków instrukcji

- czym jest blok instrukcji?
 - ciąg instrukcji, który nie zawiera instrukcji rozgałęzień (if, switch, for)

pokrycie bloków instrukcji = $\frac{\text{liczba bloków przetestowanych}}{\text{liczba wszystkich bloków programu}}$

Pokrycie bloków instrukcji

- przykład bloków

```
if (a != 0) {  
    x = -b / a;  
    System.out.println("x = " + x);  
} else {  
    System.out.println("Niepoprawne dane!");  
}
```

- w powyższym kodzie są dwa bloki
 - pełne pokrycie bloków wymaga co najmniej dwóch przypadków testowych
 - każdy do wykonania jednej z gałęzi if

Pokrycie bloków instrukcji

- odmianą miary pokrycia bloków jest **miara pokrycia instrukcji**

$$\text{pokrycie instrukcji} = \frac{\text{liczba przetestowanych instrukcji}}{\text{liczba wszystkich instrukcji programu}}$$

Pokrycie bloków instrukcji

- **zalety**
 - intuicyjna zrozumiałość
 - prostota pomiaru
 - możliwość bezpośredniego zastosowania do kodu wynikowego
 - a nie tylko do źródłowego
- **wady**
 - słabe odzwierciedlenie jakości sprawdzania warunków – przykład:

```
float a = 0;  
float b = 0;  
if (b == 0) {  
    a = 1;  
}  
x = b / a;
```

problem:
 - przypadek testowy z $b = 0$ zapewni 100% pokrycie kodu
 - ale tylko przypadek testowy z $b \neq 0$ pozwoli wykryć błąd przy dzieleniu przez 0
 - nie odzwierciedla jakości sprawdzania różnych wariantów obliczania złożonych warunków if ani warunków zakończenia pętli

Pokrycie decyzji

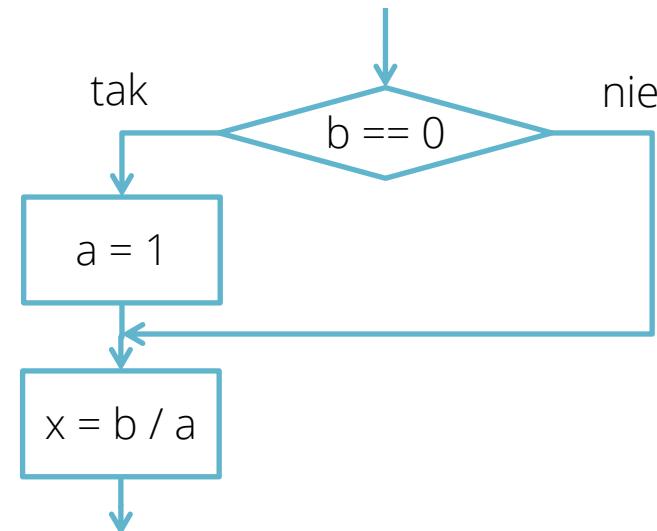
- czym jest decyzja?
 - każda możliwa wartość warunku w instrukcji powodującej rozgałęzienie programu

$$pokrycie\ decyzji = \frac{\text{liczba przetestowanych wyjść z instrukcji rozgałęziających}}{\text{liczba wszystkich wyjść z instrukcji rozgałęziających}}$$

Pokrycie decyzji

- przykład 1

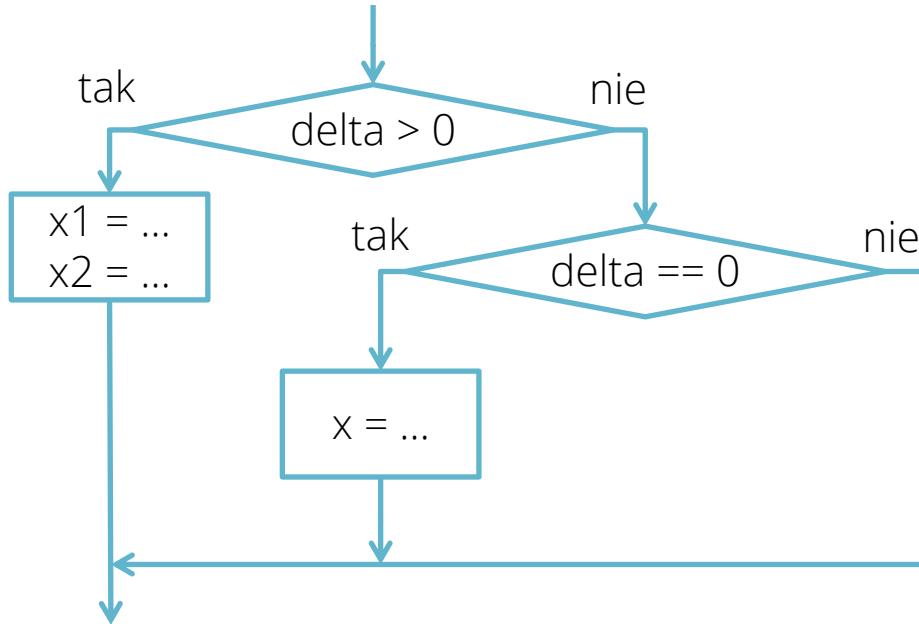
```
float a = 0;  
float b = 0;  
if (b == 0) {  
    a = 1;  
}  
x = b / a;
```



- zawiera jedną instrukcję if – warunek może przyjąć jedną z dwóch wartości: true lub false
- pełne pokrycie decyzji wymaga dwóch przypadków testowych
 - gdzie każdy spowoduje wyliczenie innej wartości warunku

Pokrycie decyzji

- przykład 2



- dwie instrukcje warunkowe
- pełne pokrycie decyzji wymaga trzech przypadków testowych
 - gdzie każdy spowoduje wyliczenie innej wartości warunku

Pokrycie decyzji

- miara pokrycia decyzji zwykle wymaga większej liczby przypadków testowych niż miara pokrycia bloków instrukcji
 - **zalety**
 - zapewnia nie gorsze sprawdzenie bloków
 - lepsze sprawdzenie sposobu testowania warunków
 - **wady**
 - brak wrażliwości na sposób testowania warunków złożonych, które nie w każdym przebiegu są obliczane do końca – przykład
 - if (a > 0 || b > 0 || obliczC() > 0) { x = 1; }
 - jeśli a > 0 → pozostałe warunki nie są sprawdzane
 - if (a > 0 && b > 0 && obliczC() > 0) { x = 1; }
 - jeśli a <= 0 → pozostałe warunki nie są sprawdzane
- w szczególności → nie jest wywoływana funkcja obliczC()

Pokrycie ścieżek

- czym jest ścieżka?
 - każda sekwencja instrukcji prowadząca od początku do końca programu

$$pokrycie\ ścieżek = \frac{\text{liczba przetestowanych ścieżek}}{\text{liczba wszystkich ścieżek}}$$

Pokrycie ścieżek

- niekompletne obliczanie warunków złożonych w niektórych językach tworzy nowe ścieżki wykonania
 - które nie są jawnie pokazane w kodzie programu!!!
- liczba wszystkich ścieżek wykonania warunków złożonych jest równa liczbie wszystkich wykonalnych kombinacji wartości warunków – przykład

```
if (a > 0 || b > 0 || obliczC() > 0) { x = 1; }
```
- Quiz: ile możliwych ścieżek?
 - 2
 - 3
 - 4
 - 8

Pokrycie ścieżek

- pełne pokrycie ścieżek wymaga takich przypadków testowych, gdzie
 - każdy przechodzi inną ścieżką
 - każda ścieżka jest testowana przez przypadek testowy
- miara pokrycia ścieżek wymaga większej liczby testów do uzyskania pełnego pokrycia
 - zapewnia to lepszą jakość testowania
- problemem są pętle
 - bardzo duża liczba ścieżek → czasem nieograniczona
 - bo wykonanie programu z dwoma obiegami pętli przebiega po innej ścieżce niż wykonanie z trzema obiegami pętli, itd...

Inne miary pokrycia kodu

- pokrycie wywołań funkcji
- dróg od przypisania wartości zmiennej do odczytania tej wartości
- odwołań do komórek i tablic
- linii kodu
 - w wielu językach
 - jedna linia kodu może zawierać wiele instrukcji
 - jedna instrukcja może być umieszczona w wielu liniach
- ...

Miary pokrycia wymagań

- jest elementem testowania czarnej skrzynki
 - nie korzysta się z wiedzy o wewnętrznej strukturze programu
- projekt testów i ocena jakości testowania odwołują się do specyfikacji wymagań
 - jako wzorca poprawnego zachowania programu
- **miarą jakości testów jest stopień pokrycia wymagań przypadkami testowymi**
- **zaleta** – bezpośrednie powiązanie jakości testów z oceną stopnia spełnienia wymagań
 - co może prowadzić do wysokiej akceptowalności wyników oceny przez użytkowników
- **wady**
 - brak jednolitej definicji pokrycia wymagań
 - brak standaryzacji miar i ich nazewnictwa

Miary pokrycia wymagań

$$pokrycie\ wymagań = \frac{\text{liczba przetestowanych wymagań}}{\text{liczba wszystkich wymagań w specyfikacji}}$$

- liczba przypadków testowych do przetestowania pojedynczego wymagania zależy od jego charakteru i złożoności

Miary pokrycia wymagań

ale czym są wymagania?

to zależy od metody przyjętej do określenia wymagań

- jeśli **tekstowo w postaci listy** – wymaganiem jest każdy punkt listy
 - zaleta – możliwość uwzględnienia w wartościach miary wymagań funkcjonalnych i niefunkcjonalnych
 - wady – niska precyzja i potencjalny brak rozłączności punktów listy
- jeśli jako **hierarchia funkcji** – wymaganiem jest każda funkcja na najniższym poziomie hierarchii
- jeśli jako **diagram przypadków użycia** – wymaganiem jest każdy zdefiniowany scenariusz przypadku użycia

Miary pokrycia wymagań

- koncentrują się na kontroli poprawnego zachowania programu
- ważne jest również zachowanie w przypadku błędu
 - pomyłki operatora
 - awarii urządzenia periferyjnego
 - błędnych danych wejściowych
 - wewnętrznego błędu programu
- zazwyczaj takie sytuacje wykrywane i sygnalizowane komunikatem o błędzie
- lista wykrywanych błędów powinna być jawnie podana w dokumentacji programów

$$pokrycie\ błędów = \frac{\text{liczba błędów zademonstrowanych w czasie testów}}{\text{liczba wszystkich błędów wykrywanych zgodnie z dokumentacją}}$$

Miary pokrycia wymagań

- specyfikacja w postaci modelu przypadków użycia jest bardzo przydatna w procesie projektowania testów akceptacyjnych
 - **biznesowe przypadki użycia** przekładają się na **zestawy testów**
 - **systemowe przypadki testowe** na **przypadki testowe**
- jakie do tego miary?
 - pokrycia biznesowych przypadków użycia zestawami testów
 - pokrycia systemowych przypadków użycia scenariuszami przypadków testowych
 - pokrycia scenariuszy systemowych przypadków użycia przez przypadki testowe

Miary pokrycia wymagań

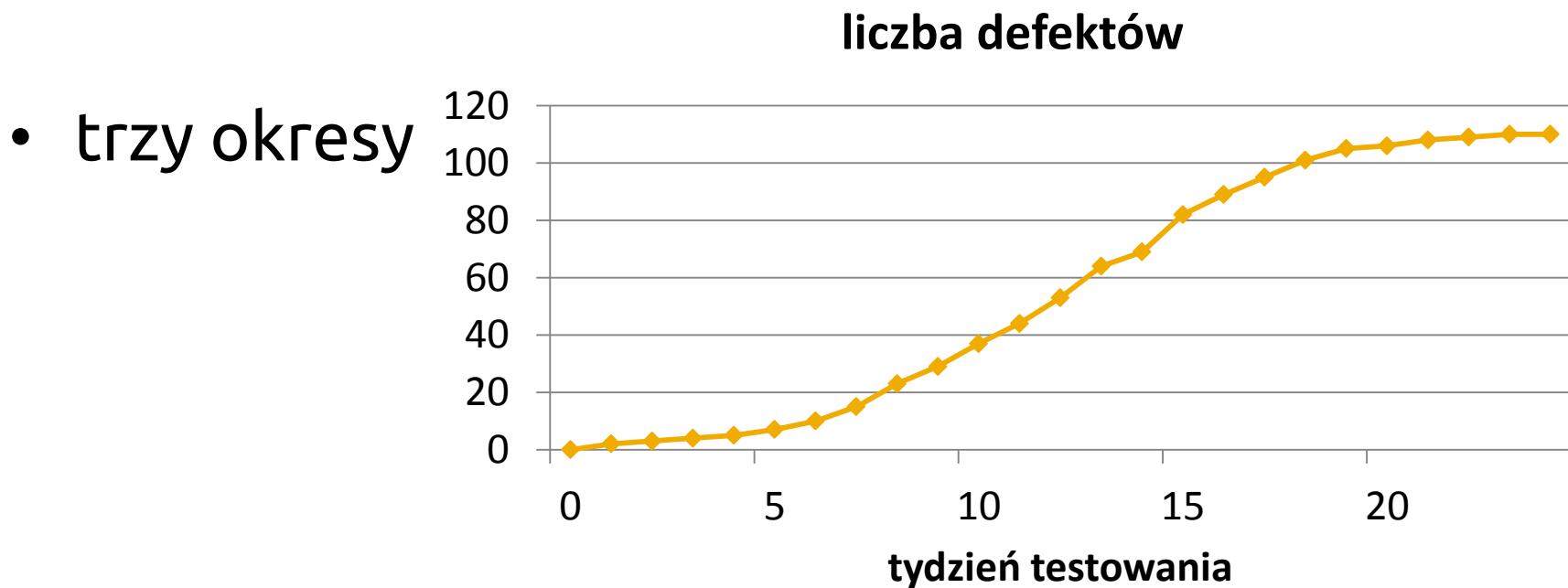
- takie miary można obliczać zbiorczo
 - liczba wszystkich scenariuszy przypadków testowych do liczby wszystkich przypadków testowych
- lub odrębie dla każdego przypadku użycia
- niskie wartości takich miar świadczą o niskiej jakości testowania
- wartością graniczną przy ocenie liczby przypadków testowych dla danego przypadku użycia jest liczba jego scenariuszy
 - jeśli liczba przypadków testowych < liczba scenariuszy przypadku użycia
 - to znaczy, że pewne scenariusze zostały pominięte przy testowaniu

Miary bazujące na defektach

- od czego zależy liczba wykrytych defektów?
 - jakość testów
 - jakość wytworzzonego kodu
 - wielkość i złożoność wytworzonego kodu
- potrzebne miary dotyczące bezpośrednio jakości kodu
 - liczba wykrytych defektów
 - liczba defektów komponentu
 - częstość defektów
 - procent defektów poprawionych

Liczba wykrytych defektów

- rejestrowana regularnie podczas testów i w różnych projektach pozwala ocenić:
 - początkową jakość wytworzzonego kodu
 - przyrost jakości w efekcie naprawiania defektów
 - efektywność różnych metod i etapów testowania



Liczba defektów komponentu

- wykrytych w czasie testowania danego komponentu

jak interpretować?

- duża liczba może świadczyć o

- wysokiej złożoności
 - niskiej jakości kodu

porównanie komponentów może pomóc w ustaleniu komponentów ryzykownych, tj. podatnych na defekty

- duża liczba może świadczyć o

- niskiej jakości pracy
 - przeciążeniu deweloperów

pomoc dla kierownika przy przydzielaniu/przemieszczeniu zasobów

Liczba defektów komponentu

- wariantem miary może być zliczanie defektów w rozbiciu na testy lub zestawy testów
- dzięki temu można wybrać testy wykrywające najwięcej defektów
- a dalej jako pomoc w konstruowaniu
 - zbioru testów regresji
 - zbioru testów zaufania (*smoke tests*)
 - do szybkiego sprawdzenia poprawności programu podczas konserwacji
 - czy program da się uruchomić, interfejsy są dostępne i reagują na działania

Częstość defektów

- jedna z najpopularniejszych miar kodu w projektach komercyjnych
- liczba defektów rośnie wraz z rozmiarem kodu
- potrzeba porównywalności wyników niezależnie od wielkości

$$\text{częstość defektów} = \frac{\text{liczba defektów}}{\text{liczba linii kodu}}$$

- można stosować zarówno dla komponentów jak i całości

Częstość defektów

problem!!!
czym jest linia kodu programu?

- liczba instrukcji kodu wynikowego (np. w asemblerze)
- liczba linii kodu źródłowego (np. w Javie)
- jeśli kod źródłowy
 - wszystkie linie wraz z komentarzami i nagłówkami
 - tylko instrukcje wykonalne i deklaracje danych
- jeśli instrukcja w kilku liniach
 - liczyć jako jedną instrukcję
 - liczyć jako kilka instrukcji
- **firma może wprowadzić własny sposób liczenia wielkości kodu**

Procent defektów poprawionych

procent defektów poprawionych = $\frac{\text{liczba defektów poprawionych}}{\text{liczba defektów znalezionych}}$

- pozwala ocenić postęp prac nad usuwaniem defektów programu
- może być wskazówką
 - czy przejść do kolejnego etapu testów?
 - czy przejść do próbnej eksploatacji?

Agenda

Po co zapewniać jakość oprogramowania?

Jakość produktu

Miary jakości produktu

Projektowanie miar jakości

Przeglądy i inspekcje

Jakość procesu

Plan zapewnienia jakości

Projektowanie miar jakości

- istnieją różne modele jakości
 - ISO 9126 / ISO 25010
 - starsze McCalla, Boehma
 - inne: FURPS, Fentona, ...
- wyznaczają zasób środków do
 - pomiaru jakości poszczególnych artefaktów
 - oceny przewidywanej jakości całości oprogramowania
- jaki problem?
- **jak wybrać lub zaprojektować miary w celu oceny i poprawy aspektów najistotniejszych w danym projekcie?**

Metoda GQM

- Goal – Question – Metric
- określa sposób odwzorowania **biznesowych celów przedsiębiorstwa** w zestaw **mierzalnych cech oprogramowania** lub procesu wytwórczego
- zakłada z następującą analizę problemu, obejmującą 3 kroki
 1. zdefiniowanie celu biznesowego jako zadania oceny lub poprawy określonego aspektu produktu lub procesu (*Goal*)
 2. określenie zestawu pytań opisujących stopień osiągnięcia celu (*Question*)
 3. dla każdego pytania – określenie zestawu miar potrzebnych do udzielenia odpowiedzi (*Metric*)

Metoda GQM – przykład

- cel biznesowy powinien wskazywać
 - intencje działania
 - perspektywę oceny
 - przedmiot poddany działaniu
 - cechę przedmiotu podlegającą ocenie
- przykład – sytuacja:
 - podczas nieudanych testów akceptacyjnych odnotowano dużą liczbę błędnych wyników obliczenia – można podejrzewać niską jakość wcześniejszej fazy testów jednostkowych
- aby poprawić sytuację można sformułować następujący cel:
 - ***celem jest zwiększenie wiarygodności testów jednostkowych***
- jakie pytania do takiego celu?
 - *jaka jest wiarygodność obecnego planu testów?*
 - *czy wiarygodność testów uległa poprawie po wprowadzonych zmianach?*
- jakie miary do tych pytań?
 - na przykład miary pokrycia kodu przed i po zmianach sposobu testowania

Agenda

Po co zapewniać jakość oprogramowania?

Jakość produktu

Miary jakości produktu

Projektowanie miar jakości

Przeglądy i inspekcje

Jakość procesu

Plan zapewnienia jakości

Przeglądy i inspekcje

- w czasie projektu powstają
 - dokumenty
 - mogą zawierać modele analityczne i projektowe
 - kod programu
- podstawowa metoda oceny → przegląd formy i treści dokumentu
 - przez zespół oceniający → w trakcie spotkania
- przegląd kończy się zestawieniem błędów i niezgodności
 - autor może wykorzystać do poprawienia jakości dokumentu

Przeglądy i inspekcje

- trzy rodzaje
 1. przeglądy jakości – sprawdzenie zgodności artefaktów z przyjętymi standardami
 2. przeglądy postępów – zatwierdzenie ocenianych produktów
 - najczęściej w punkcie kontrolnym produktu
 3. inspekcje – usunięcie błędów w ramach procesu weryfikacji

Przegląd

- ang. *review, walkthrough*
- **technika oceny** produktu, grupy produktu lub stanu prac na jakimś etapie wytwarzania **względem określonego zbioru kryteriów**
- najważniejszy element to spotkanie
 - przewodniczący
 - recenzenci oceniający produkt
 - osoba reprezentująca autorów

Przegląd

- **przewodniczący**
 - wybiera recenzentów
 - dostarcza materiały do oceny
 - przeprowadza spotkanie
- **recenzent**
 - sprawdza oceniany produkt względem ustalonych kryteriów
- **osoba reprezentująca autorów**
 - udziela wyjaśnień w czasie spotkania
 - usuwa błędy w przyszłości

**ocena autorów i sposoby usunięcia błędów
są poza zakresem spotkania**

Przegląd – fazy

1. przygotowanie

- wyznaczenie terminu spotkania
- wyznaczenie recenzentów
- dostarczenie kopii produktów do oceny
- recenzenci oceniają produkt(-y)
 - z różnych punktów widzenia – projektanta, testera, użytkownika
 - przygotowują listy błędów

2. spotkanie

- przedstawienie i dyskusja błędów i usterek
- listy błędów mogą być skorygowane w czasie dyskusji
- przewodniczący przekazuje listy błędów autorom z zaleceniami wykonania uzgodnionych poprawek

3. zamknięcie – krótki okres (np. 1 tydzień)

- naprawa przez autorów
- przedstawienie poprawionego produktu przewodniczącemu
 - i być może ponownie recenzentom
- zatwierdzony produkt przyjmowany do repozytorium projektu

Przegląd

- mogą się różnić stopniem sformalizowania procedury i składem uczestników
 - przeglądy jakości zwykle przeprowadzane przez kontrolerów jakości lub auditorów
 - koncentrują się często na badaniu formalnej zgodności artefaktów z przyjętymi standardami produktowymi
 - przeglądy postępów
 - zwykle mają na celu ustalenie stanu prac i zatwierdzenie merytorycznych treści kluczowych artefaktów
- podstawowa lista przeglądów
 - przegląd specyfikacji wymagań
 - przegląd projektu (wstępnego i szczegółowego)
 - przegląd kodu programu
 - przegląd gotowości do testowania
 - przegląd zatwierdzenia produktu

Przegląd

- zaletą przeglądu jest brak ograniczeń co do zakresu stosowania
 - oceniane mogą być wszystkie produkty i działania
- **systematyczne przeglądy są skuteczną metodą oceny postępów prac, zatwierdzania wyników i podnoszenia jakości**
- mogą też służyć **weryfikacji** produktów

Inspekcja

- ang. *inspection*
- metoda statycznej analizy **kodu lub projektu programu**
- polega na systematycznej analizie treści badanego dokumentu
 - przez niewielki zespół specjalistów
- członkowie pełnią ścisłe przypisane role
 - projektant badanego modułu
 - programista
 - tester
 - przewodniczący

Inspekcja

- uczestnicy (poza przewodniczącym) analizują badany produkt
 - każdy ze swojego punktu widzenia
 - zgłaszają zastrzeżenia
 - wskazują błędy
- po zakończeniu spotkania autor wprowadza poprawki i usuwa błędy
- przewodniczący
 - przygotowuje ten proces
 - kieruje spotkaniem
 - sporządza raport podsumowujący wyniki

Inspekcja

- zalety
 - możliwość wczesnej eliminacji błędów
 - przed i w czasie pisania kodu
 - wysoka skuteczność
 - do 80% błędów – wg Fagana (autora)
- odbywa się zawsze w kontekście specyfikacji badanego produktu
 - inspekcja kodu obejmuje – względem specyfikacji projektu
 - inspekcja projektu – względem specyfikacji wymagań

**inspekcje kodu i projektu są metodą weryfikacji
nie nadają się do zatwierdzania**

Agenda

Po co zapewniać jakość oprogramowania?

Jakość produktu

Miary jakości produktu

Projektowanie miar jakości

Przeglądy i inspekcje

Jakość procesu

Plan zapewnienia jakości

Jakość procesu

- jak zapewnić jakość procesu aby osiągnąć jakość produktu?
- ISO 9001
- CMMI

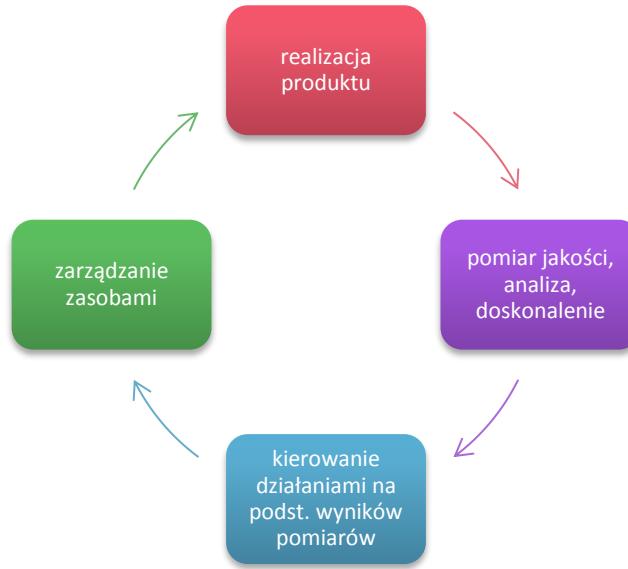
ISO 9001

- określa model i dokumentację działań służących zapewnieniu jakości podczas
 - projektowania
 - wytworzania
 - instalowania
 - obsługiwania produktów
- model opiera się na następujących zasadach
 - orientacja na klienta i jego potrzeby
 - przywódcza rola kierownictwa i zaangażowanie ludzi
 - procesowe podejście do zarządzania
 - podejmowanie decyzji na podstawie faktów
 - ciągłe doskonalenie systemu

ISO 9001

- możliwość uzyskania certyfikatu jakości
 - duża popularność normy
 - umocnienie firmy na rynku
 - dowód dla klienta
 - proces wytwórczy przebiega w warunkach stabilnych, udokumentowanych i nadzorowanych

system zarządzania jakością: procesy realizowane w cyklu



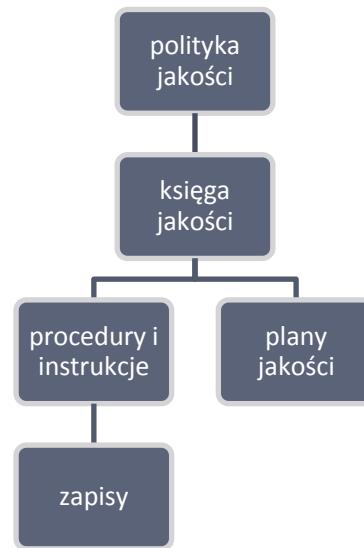
- cel realizacji produktu
 - spełnienie wymagań klienta
- cel pomiaru
 - ocena rzeczywistego zadowolenia klienta
 - na jej podstawie kierownictwo może ulepszyć procesy i przydzielić zasoby umożliwiające wdrożenie poprawek

ISO 9001

- sposób zdefiniowania procesów jest bardzo ogólny
 - norma może być stosowana w dowolnych firmach produkcyjnych lub usługowych
- norma nie narzuca konkretnej organizacji systemu zarządzania jakością
- ale stawia trzy wymagania
 1. **zdefiniowanie i udokumentowanie procesów** objętych działaniem systemu zarządzania jakością i określenie sposobu kontroli ich skuteczności
 2. **codzienne prowadzenie procesów i działań kontrolnych** zgodnie z tą definicją oraz **dokumentowanie ich przebiegu** za pomocą odpowiednich zapisów
 3. **wdrożenie działań** zmierzających do **ciągłego doskonalenia procesów**

ISO 9001

- duże znaczenie ma dokumentacja
 - opracowywana i prowadzona przez przedsiębiorstwo
 - oceniana przez zewnętrznych audytorów
- w postaci hierarchicznie powiązanej struktury



ISO 9001 – polityka jakości

- *quality policy*
- dokument strategiczny
- określa cele, zadania i podejście do doskonalenia systemu jakości
- często ma postać deklaracji kierownictwa wyrażającej
 - zrozumienie potrzeby spełnienia oczekiwania klientów
 - wolę stosowania norm
 - wprowadzenia i stałego doskonalenia systemu jakości
 - zapewnienia do tego niezbędnych zasobów
- z tej deklaracji, z misji przedsiębiorstwa i z oczekiwania udziałowców wynikają cele i podejście do zadań systemu jakości
- jest dokumentem jawnym, zwykle krótkim

ISO 9001 – księga jakości

- *quality manual*
- najważniejszy dokument systemu opisujący
 - przedsiębiorstwo
 - procesy podlegające działaniu systemu jakości
 - dostrzegane zagrożenia przebiegu tych procesów
 - przewidziane środki zaradcze
- może również zawierać
 - zapis polityki jakości
 - opisy procedur
 - w małym przedsiębiorstwie
- najważniejsza część → opis procesów
 - przebieg – np. jako sieć działań
 - wskazanie elementów ryzyka
 - lista procedur niwelujących ryzyko i zapewniających utrzymanie wysokiej jakości

ISO 9001 – procedury i instrukcje

- **procedury**

- opisują szczegółowo kolejność działań
 - wchodzących w skład procesu
 - lub jego czynności składowych
- wskazują osoby odpowiedzialne za ich wykonanie
- przykłady
 - procedura przegląd wymagań
 - procedura testowania po zmianie wersji
- 6 procedur, które powinny być wdrożone
 - nadzór nad obiegiem dokumentów systemu jakości
 - nadzór nad zapisami
 - nadzór nad wyrobami wybrakowanymi
 - audyty wewnętrzne
 - działania korygujące
 - działania zapobiegawcze

- **instrukcje**

- mają węższy zakres (o ile są udokumentowane)
- opisują sposób działania osób na poszczególnych stanowiskach

ISO 9001 – zapisy

- dokumentują wykonanie procesów i procedur
 - opisanych w dokumentacji systemu jakości
- w efekcie
 - dowodzą działania systemu w praktyce
- rodzaje zapisów – zwykle jako formularze
 - zgłoszenie zmian
 - protokół przeglądu dokumentów projektowych
 - protokół testowania
 - projekt zarządzenia
- lista wymaganych zapisów
 - jest częścią definicji procedur
- wymaganie normy – zarządzanie zapisami
 - zdefiniowane w postaci procedury
 - nadzorowane

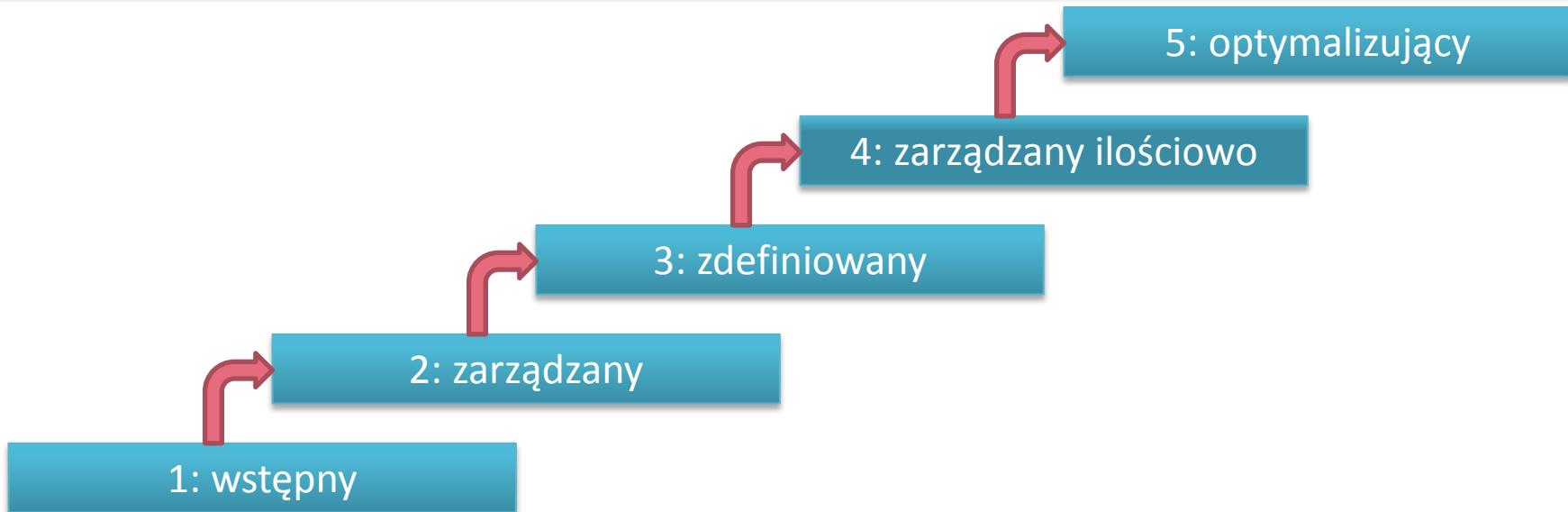
ISO 9001 – plany jakości

- są dokumentami zarządczymi
- zadanie
 - przygotowanie zmian doskonalących system zarządzania jakością
 - przeprowadzenie zmian tak, aby
 - nie zakłócić działania systemu
 - nie stworzyć zagrożeń dla jakości produktów

CMMI

- kompleksowy model dojrzałości organizacyjnej
Capability Maturity Model Integration
- umożliwia
 - ocenę dojrzałości procesów zarządzania projektem u wytwórcy oprogramowania
 - dokonywanie samooceny i doskonalenia procesu zarządzania jakością
- ocena dokonywana w 5-stopniowej skali

CMMI



- do każdego poziomu przypisane procesy
 - które są wymagane do zrealizowania
- zakres każdego procesu określają
 - cele
 - praktyki umożliwiające spełnienie tych celów
- przejście na wyższy poziom wymaga
 - wdrożenia wszystkich procesów przypisanych do tego poziomu
 - osiągnięcia wszystkich celów

Agenda

Po co zapewniać jakość oprogramowania?

Jakość produktu

Miary jakości produktu

Projektowanie miar jakości

Przeglądy i inspekcje

Jakość procesu

Plan zapewnienia jakości

Plan zapewnienia jakości

- dla całego projektu opracowywany jest **plan projektu**
 - zadania do osiągnięcia założonych celów projektu (np. wytworzenie oprogramowania)
 - każde zadanie ma termin wykonania, zasoby, dane do rozpoczęcia zadania, artefakty do wytworzenia
- **plan zapewnienia jakości**
- definiuje przeglądy i inne metody oceny do wykonania w obrębie oceny zadań lub pomiędzy nimi
 - po co?
 - aby zapewnić odpowiednią jakość **wszystkich produktów** tworzonych w czasie projektu

Plan zapewnienia jakości – jak?

- jak dokumentować plan?
- zależnie od rangi jakości w hierarchii celów projektu
 - a) w systemach krytycznych dla prowadzenia biznesu lub mogących powodować zagrożenie dla życia ludzi → jakość na 1. miejscu
 - plan zapewnienia jakości jest odrębnym dokumentem, powiązanym z planem zarządzania projektem
 - b) w projektach komercyjnych → jakość na dalszym miejscu
 - skromniejsza dokumentacja, np. jako rozdział w planie zarządzania projektem
- plan zapewnienia jakości powinien być zatwierdzony przez szefa każdego działu, który odpowiada za realizację wymienionych zadań!

Plan zapewnienia jakości – treść

- treść planu powinna
 - definiować standardy, konwencje i miary używane w projekcie
 - określać harmonogram przeglądów
 - definiować procedury nadzoru nad różnymi elementami procesu wytwórczego
- firmy informatyczne mają często korporacyjne plany jakości
 - adaptowane do konkretnych projektów

IEEE 730-2002

- IEEE Standard for Software Quality Assurance Plans
- może być wzorem przy opracowywaniu swojego planu
- proponuje konkretny układ i zakres treści planu zapewnienia jakości w projektach systemów krytycznych
 - w projektach komercyjnych można wykorzystać podzbiór standardu
- **główe elementy zawartości**
 - cel
 - powiązane dokumenty
 - zarządzanie
 - dokumentacja
 - **metody – kilka rozdziałów**
 - kontrola dostawców
 - uzupełnienia
 - zakończenie

IEEE 730-2002

1. cel

- cel i zakres konkretnego planu zapewnienia jakości
- lista artefaktów oprogramowania objętych planem
- planowany sposób użycia oprogramowania

2. dokumenty powiązane

- odnośniki do innych dokumentów wymienionych w tekście planu

3. zarządzanie

- struktura organizacyjna projektu
 - powiązanie zadań z rolami i odpowiedzialnością
 - poziom swobody organizacyjnej i obiektywizmu w ocenie i monitorowaniu jakości oprogramowania oraz w weryfikowaniu rozwiązań problemów
- zadania do wykonania
 - kryteria rozpoczęcia i zakończenia każdego zadania
 - powiązania między zadaniami i planowanymi głównymi punktami kontrolnymi
 - kolejność zadań
- role i odpowiedzialność za wykonanie zadań
- planowane zasoby

IEEE 730-2002

4. dokumentacja

- określenie dokumentacji wytwarzania, weryfikacji i zatwierdzania, użytkowania i konserwacji oprogramowania
- lista dokumentów, które powinny być poddane przeglądowi lub audytom
 - dla każdego dokumentu – które przeglądy i audyty należy przeprowadzić wraz z kryteriami oceny ich poprawności
- **minimalny zakres tych dokumentów**
 - specyfikacja wymagań
 - projekt oprogramowania
 - plany weryfikacji i zatwierdzania
 - działania takie jak: analiza, ocena, przegląd, inspekcja, testowanie produktów i procesów;
 - określa zadania, dane wejściowe, oczekiwane wyniki
 - raport z weryfikacji i raport z zatwierdzania
 - dokumentacja użytkownika
 - plan zarządzania konfiguracją

IEEE 730-2002

- **metody – kilka rozdziałów**
- 5. standardy, praktyki, konwencje i metryki
- 6. przeglądy oprogramowania
 - minimalny zakres przeglądów
 - przegląd specyfikacji oprogramowania
 - przegląd projektu architektury
 - przegląd projektu szczegółowego
 - przegląd planu weryfikacji i zatwierdzania
 - audyt funkcjonalny
 - audyt fizyczny
 - audyty wewnętrzprocesowe
 - przeglądy zarządcze
 - przegląd planu zarządzania konfiguracją
 - przegląd poimplementacyjny
- 7. testy
- 8. raportowanie problemów i działania korygujące
- 9. narzędzia, techniki i metodyki

10.kontrola mediów

- forma w jakiej dane artefakty będą wytworzone (wraz z dokumentacją) oraz proces tworzenia kopii zapasowych i odtwarzania
- zabezpieczenie mediów z programem komputerowym przed nieautoryzowanym dostępem, zniszczeniem itp. na każdym etapie projektu
 - może być jako część planu zarządzania konfiguracją

11.kontrola dostawców

- szczególnie ważna w projektach integrujących komponenty z różnych źródeł (*component-based software*)
- zapewnienie jakości produktów powstających na zewnątrz
 - procedury kontroli produktów i procesów na różnych etapach

IEEE 730-2002

12.zbieranie danych, konserwacja i retencja

- dane do dokumentacji i sama dokumentacja dotycząca kontroli
 - co i jak zbierać, jak i jak długo przechowywać, itp.

13.szkolenia

14.zarządzanie ryzykiem

15.słownik pojęć

16.procedury zmian planu zapewnienia jakości i historia zmian

Podsumowanie

podstawy zapewnienia jakości
oprogramowania

duży nacisk na dokumentację

Pytania



Źródła

- Sacha K., Inżynieria oprogramowania, PWN 2010
- Górski J. (red.), Inżynieria oprogramowania w projekcie informatycznym, Mikom 2000
- Patton R., Testowanie oprogramowania, Mikom 2002
- Radliński Ł., Analiza porównawcza modeli jakości oprogramowania. Zeszyty Naukowe Uniwersytetu Szczecińskiego, Seria: Studia Informatica. 19, 131-150 (2006)
- Shafer L., Software Testing, IEEE eLearning Library, 2011
- IEEE Standard for Software Quality Assurance Plans, IEEE Computer Society, 2002

Następny wykład

Testowanie oprogramowania

Inżynieria oprogramowania

Wykład 9:

Testowanie oprogramowania

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Agenda

Poziomy testowania

Proces testowania

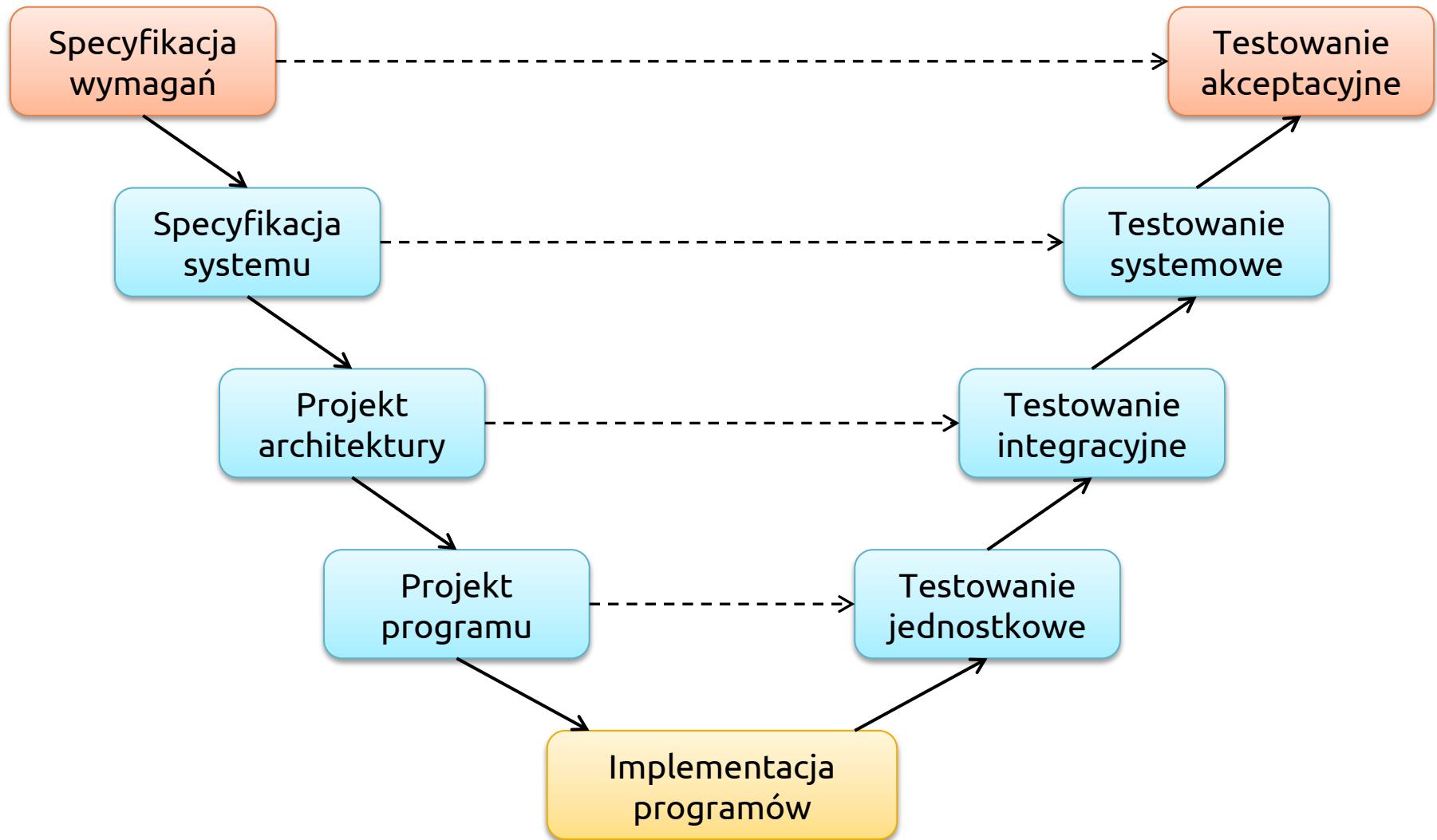
Metody testowania

Automatyzacja testowania

Debugowanie

Testowanie – demo

Model V procesu testowania



Testowanie jednostkowe

- *unit testing, module testing*
- testowane są **podstawowe jednostki programu**
 - opisane w projekcie szczegółowym
- czym są te jednostki?
 - zależnie od konkretnej implementacji
 - procedury, funkcje
 - skrypty SQL
 - metody, klasy
- cel
 - sprawdzenie zgodności działania jednostek programu ze specyfikacją
 - + wykrycie i usunięcie defektów

Testowanie jednostkowe

- sposób realizacji testowania jednostkowego
 - zależy od wymaganej niezawodności
 - wysoka → osobny etap, niezależny zespół
 - "niska" → zazwyczaj deweloperzy, w czasie implementacji

Testowanie jednostkowe

- problem
 - zależność testowanej jednostki od innych jednostek
- rozwiązanie
 - użycie specjalnego oprogramowania wspomagającego
 - opracowanie sterowników testowych
 - wywoływanie badanych jednostek z odpowiednimi argumentami
 - namiastki symulujące działanie brakujących jednostek
- takie środowisko zachowane i wykorzystane przy testach regresji
 - po poprawkach – usuwaniu błędów
 - po zmianach w kolejnych wersjach programu

Testowanie integracyjne

- *interface testing, integration testing*
- połączenie dwóch jednostek tworzy nową jednostkę
 - mogą ujawniać się błędy
 - niedopasowanie mechanizmów współpracy
- przedmiot działań testowania integracyjnego
 - łączenie jednostek w coraz większe komponenty
 - sprawdzanie zgodności ich działania ze specyfikacją
 - projektem architektury oprogramowania

Testowanie integracyjne

- cel
 - sprawdzenie **funkcjonowania interfejsów**
 - sposobu wywoływania "usług" i przekazywania argumentów
 - postaci zwracanych rezultatów
 - zgodności jednostek miary
 - zgodności wyjątków
 - zgłaszanych w jednej jednostce
 - obsługiwanych w drugiej jednostce
 - wykrycie i usunięcie defektów
 - aż do zintegrowania i przetestowania działania całości architektury

Testowanie integracyjne

- kiedy plan testowania integracyjnego?
 - wraz z opracowaniem architektury
- jak przeprowadzać testowanie integracyjne
 - najpierw złożyć całość
 - później testować od razu całość
 - składać jednostki etapami
 - i etapami je testować

Testowanie systemowe

- *system testing*
- testowana jest **całość oprogramowania**
 - zintegrowana i zainstalowana w odpowiednim środowisku wykonawczym
- cel
 - sprawdzenie sprawdzenie zgodności sposobu działania wszystkich funkcji ze specyfikacją
 - weryfikacja innych cech systemu
 - określonych jako wymagania niefunkcjonalne
- środowisko testowe powinno być jak najbardziej zbliżone do przewidywanego środowiska użytkowania
- konstrukcja testów traktuje system jako całość
 - nie jest rozpatrywana szczegółowa budowa oprogramowania

Testowanie systemowe

- kiedy plan testowania systemowego?
 - wraz z powstaniem specyfikacji wymagań systemu
 - np. w formie systemowego modelu przypadków użycia
 - przygotowanie testów nie wymaga znajomości szczegółów konstrukcji oprogramowania
 - ale wymaga znajomości sposobów sprawdzania wymagań niefunkcjonalnych
 - dlatego testowanie systemowe zazwyczaj przeprowadzane przez wyspecjalizowany zespół testujący
- kiedy przeprowadzane testowanie systemowe?
 - po zakończeniu integracji oprogramowania

Testowanie systemowe

- zwykle złożone z szeregu odrębnych kroków
 - sprawdzane różne aspekty działania systemu
 - zależnych od dziedziny zastosowania i specyfikacji wymagań
- przykłady testów systemu
 - **testy funkcjonalne** (*functional testing*)
 - poprawność wykonania wszystkich funkcji systemu
 - np. systemowych przypadków użycia
 - **testy wydajnościowe** (*performance testing*)
 - działanie systemu przy nominalnym obciążeniu
 - np. czas przetwarzania
 - **testy przeciążeniowe** (*stress testing*)
 - zachowanie systemu przy przekroczeniu założonego obciążenia

Testowanie systemowe

- przykłady testów systemu – c.d.
 - **testy zabezpieczeń** (*security testing*)
 - skuteczność mechanizmów ochrony systemu
 - przed nieuprawnionym użyciem
 - **testy odporności** (*recovery testing*)
 - działanie oprogramowania w warunkach awarii
 - np. nagłe wyłączenie, restart komputera, odłączenie sieci
 - **testy zgodności** (*compatibility testing*)
 - możliwość pracy na różnych platformach
 - np. systemach operacyjnych, systemach baz danych
 - **testy dokumentacji** (*documentation testing*)
 - zgodność działania z opisem zawartym w dokumentacji

Testowanie akceptacyjne

- *acceptance testing*
- oprogramowanie dostarczane do użytkownika zainstalowane
 - w docelowym środowisku
 - w środowisku imitującym docelowe
- cel
 - sprawdzenie zgodności działania z wymaganiami i potrzebami użytkownika

Testowanie akceptacyjne

- **forma testowania**
 - może być podobna do testów systemowych, ale
 - brak zorientowania na znajdowanie i usuwanie defektów
 - zademonstrowanie i zatwierdzenie produktu przez użytkownika
 - ewentualne dostrojenie do jego potrzeb
- **powinien przeprowadzać użytkownik**
 - w praktyce często przeprowadza wykonawca
 - pod nadzorem użytkownika
 - pod nadzorem firmy działającej na zlecenie użytkownika
 - wtedy
 - użytkownik najpierw zatwierdza scenariusze testowe
 - później kontroluje wiarygodność przeprowadzania testów
- **w razie wykrycia defektów**
 - opisanie ich w raporcie problemów
 - usunięcie w terminie określonym w umowie

Testowanie akceptacyjne

- w przypadku systemów ogólnodostępnych
 - może być w formie kontrolowanej eksploatacji w siedzibie wytwórcy
 - **testy alfa**
 - u wytypowanych odbiorców lub dystrybutorów
 - **testy beta**

Testowanie akceptacyjne

- typowe kroki
 - **testy funkcjonalne**
 - dopasowanie funkcji systemu do potrzeb procesów biznesowych
 - **testy operacyjne**
 - działanie funkcji wykorzystywanych przez administratorów systemu
 - **testy niefunkcjonalne**
 - podobne lub identyczne z testami systemowymi

Agenda

Poziomy testowania

Proces testowania

Metody testowania

Automatyzacja testowania

Debugowanie

Testowanie – demo

Proces testowania

- nie ma jednej standardowej organizacji procesu
- typowe działania
 1. planowanie testów
 2. przygotowanie testów
 3. testowanie i usuwanie defektów

Proces testowania – planowanie testów

- podstawowy dokument planistyczny
 - plan testów
- może mieć różną postać
 - pojedynczy dokument opisuje testowanie na wszystkich poziomach
 - plany testowania na różnych poziomach w osobnych dokumentach
 - jeśli testy jednostkowe i integracyjne wykonują deweloperzy
 - często nie tworzy się formalnych planów
 - jedynie zarezerwowany dodatkowy czas i zasoby w planie projektu
 - plan testów obejmuje jedynie testowanie systemowe i akceptacyjne

Proces testowania – planowanie testów

- zawartość planu testów
 - zakres
 - strategia testowania i raportowania błędów
 - zasoby
 - harmonogram działań

Proces testowania – planowanie testów

- **zakres**
 - identyfikacja testowanych produktów
 - wymagania, które będą testowane
 - wymagania, które nie będą testowane
- **strategia testowania i raportowania defektów**
 - środki użyte do zapewnienia jakości różnych grup wymagań
 - działania i metody testowania
 - kryteria zakończenia testów
 - np. za pomocą miar ujętych liczbowo
 - kryteria oceny pozytywnego lub negatywnego wyniku testu oprogramowania lub jego komponentów
 - klasyfikacja defektów pod względem szkodliwości → określenie dopuszczalnych limitów
 - opis na tyle dokładny, żeby umożliwić
 - identyfikację głównych zadań
 - ocenę zasobów i czasu potrzebnych do wykonania zadań

Proces testowania – planowanie testów

- **zasoby**
 - środowisko testowe
 - system komputerowy wraz z oprogramowaniem do prowadzenia testów
 - zakupy
 - prace instalacyjne i konfiguracyjne
 - wytworzenie sterowników i namiastek testowych
 - pozyskanie i przeniesienie danych
 - zespół lub zespoły wykonawców
 - różne kwalifikacje
 - małe projekty → wskazanie osób przydzielonych do testowania
 - większe projekty → zdefiniowanie struktury organizacyjnej
 - podporządkowanej kierownikowi projektu
 - obejmującej projektantów i wykonawców testów
 - określenie odpowiedzialności za
 - dostarczenie testowanych programów
 - przygotowanie środowiska
 - projektowanie i wykonanie testów
 - usunięcie defektów
 - przestrzeń biurowa

Proces testowania – planowanie testów

- **harmonogram działań**
 - czas testowania zależy od
 - wielkości oprogramowania
 - stopnia automatyzacji testów
 - liczby jednocześnie pracujących testerów
 - harmonogram testowania jest ścisłe powiązany z
 - harmonogramem opracowania oprogramowania
 - wielkością przydzielonych do projektu zasobów
 - najtrudniejsze jest oszacowanie pracochłonności testowania

Proces testowania – przygotowanie testów

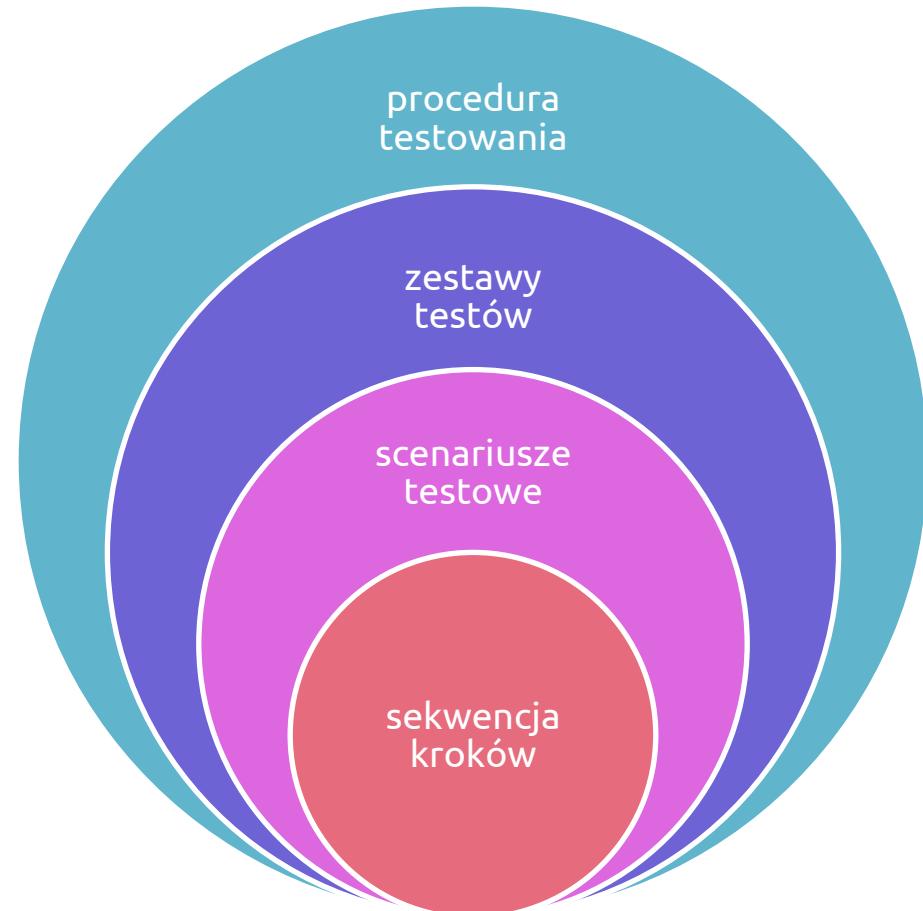
- **zaprojektowanie zestawu testów**
 - który zapewni kompletne i wiarygodne sprawdzenie produktu
 - zgodnie ze strategią określoną w planie testowania
- **najważniejszy dokument projektowy**
 - specyfikacja testów
 - opisuje planowane testy oprogramowania
- **treść specyfikacji testów**
 - lista przypadków testowych
 - wymagania lub ograniczenia ich wykonania
 - na przykład obecność określonych danych w bazie danych
- **scenariusz przypadku testowego (*test case scenario*)**
 - sekwencja czynności, które mają być wykonane przez testera
 - np.: wypełnienie formularza, zatwierdzenie przyciskiem OK, zaznaczenie opcji w kolejnym oknie, zatwierdzenie tych opcji przyciskiem Zakończ

Proces testowania – przygotowanie testów

- struktura i postać specyfikacji testów
 - zależą od wielkości projektu
 - małe projekty → kilka kartek z opisem wszystkich przypadków testowych
 - większe projekty → kilka odrębnych dokumentów
 - poświęconych testowaniu różnych obszarów funkcjonalnych i cech niefunkcjonalnych
- w większych projektach
 - trudne jest przejście z planów testów do specyfikacji testów
 - rozwiązaniem jest opracowanie dodatkowy projekt testów
 - powiązanie strategii z planu testów z dokładnym opisem przypadków testowych zapisanym w specyfikacji testów

Proces testowania – przygotowanie testów

- **procedura testowania**
 - dodatkowy dokument
 - określa kolejność i sposób przeprowadzania testów
- **określa**
 - początkowy stan systemu w chwili rozpoczęcia testów
 - listę zestawów testów do wykonania
- **zestaw testów** zawiera
 - listę scenariuszy przypadków testowych
 - wykonywanych w określonej kolejności
 - listę zbiorów danych testowych używanych w tych scenariuszach
- **scenariusz testowy**
 - składa się z sekwencji kroków – pojedynczych czynności



Proces testowania – przygotowanie testów

- pojedynczy przypadek testowy odpowiada wykonaniu
 - jednego scenariusza przypadku testowego
 - z użyciem konkretnych danych testowych
- w procedurze testowania przypadki testowe nie są jawnie wymieniane
 - bo jest ich zbyt dużo
- zamiast tego
 - zdefiniowanie scenariuszy przypadków testowych
 - zbiory danych testowych

Proces testowania

testowanie i usuwanie błędów

- dwie różne czynności – mogą być realizowane w innym czasie
 - testowanie
 - wykrywanie defektów
 - usuwanie defektów
 - poprawianie programu

Proces testowania

testowanie i usuwanie błędów

- cykl testowania
 - seria testów
 - poprawki
 - testy regresji
 - czy wszystkie dotychczas działające funkcje nadal działają
 - retesty
 - czy wszystkie wykryte defekty zostały rzeczywiście usunięte
- kolejny cykl testowania...



Agenda

Poziomy testowania

Proces testowania

Metody testowania

Automatyzacja testowania

Debugowanie

Testowanie – demo

Metody testowania

metoda czarnej skrzynki (*black box testing*)

- bez wykorzystania wewnętrznej struktury
- przez użytkownika

metoda białej skrzynki (*white box testing*)

- poszczególne składniki oprogramowania
- testy dopasowane do struktury oprogramowania

Testowanie czarnej skrzynki

- sprawdzenie poprawności działania we wszystkich sytuacjach
 - które można wyróżnić na podstawie specyfikacji
- projektowanie przypadków testowych polega na
 - wybraniu zbioru poprawnych i niepoprawnych danych wejściowych
 - zapewniających pokrycie całej przestrzeni zachowań programu
- metoda **nie gwarantuje** wyczerpującego sprawdzenia wszystkich wewnętrznych elementów programu

Testowanie czarnej skrzynki

- deterministyczne projektowanie przypadków testowych
 - polega na poszukiwaniu jak najmniejszego zbioru wartości danych wejściowych
 - pozwalającego zademonstrować wszystkie rodzaje zachowań programu
- w tym celu konieczne jest:
 1. podzielenie wszystkich możliwych wartości danych wejściowych na klasy wartości przetwarzanych podobnie
 2. wybranie do testów pewnej liczby wartości z każdej z tych klas

Testowanie czarnej skrzynki

typowe metody

- analiza klas równoważności
- analiza wartości brzegowych
- testowanie sposobów użycia
- testowanie losowe

Testowanie czarnej skrzynki

typowe metody

- **analiza klas równoważności**
 - podział wszystkich możliwych wartości danych wejściowych na zbiory (klasy) przetwarzane podobnie
 - wybranie pewnej liczby wartości z każdej klasy
 - należy uwzględnić zarówno wartości danych poprawnych jak i błędnych
 - np. dane wejściowe zawierają określenie czasu →
 - godzina $\in <0, 23>$, minuta $\in <0, 59>$ →
 - dziewięć klas równoważności – możliwych poprawnych kombinacji poprawnych/błędnych wartości godziny i minuty
- **analiza wartości brzegowych**
 - podobnie jak w analizie klas równoważności
 - ale tu analizowane są wartości szczególne
 - leżące na granicach klas
 - dodatkowo wartości o 1 większe i o 1 mniejsze od wartości granicznych

Testowanie czarnej skrzynki

typowe metody

- **testowanie sposobów użycia**
 - *use case testing*
 - **znalezienie wszystkich dostępnych dla użytkownika sposobów wykorzystania programu**
 - szczegóły metody zależą od postaci specyfikacji programu
 - jeśli stosowany model przypadków użycia
 - to sposobem wykorzystania jest **każdy przypadek użycia**
 - metoda często wykorzystywana do testów akceptacyjnych
 - wiarygodność metody
 - większa im większa liczba przypadków testowych sprawdzających poszczególne scenariusze

Testowanie czarnej skrzynki

typowe metody

- **testowanie losowe**
 - losowe projektowanie przypadków testowych (Monte Carlo)
 - założenie:
 - przypadkowo generowane dane testowe rozłożą się w całym zakresie możliwych wartości
 - po dostatecznie dużej liczbie prób pokryją cały zakres przetwarzania testowanego programu
 - zalety
 - uproszczenie etapu projektowania przypadków testowych
 - łatwość automatyzacji
 - wady
 - skomplikowany proces analizy zarejestrowanych wyników
 - brak gwarancji osiągnięcia zadowalającego pokrycia w założonym czasie

Testowanie białej skrzynki

- sprawdzenie poprawności elementów
 - z których zbudowany jest program
- projektowanie przypadków użycia polega na
 - wybraniu takiego zbioru danych wejściowych
 - który zapewni pełne pokrycie kodu programu
- wada
 - zależność od struktury programu
 - zmiana struktury może wymagać modyfikacji przypadków testowych
- może być stosowane na wszystkich etapach testowania
 - z wyjątkiem testowania akceptacyjnego
 - w praktyce – najczęściej w testowaniu jednostkowym

Testowanie białej skrzynki

- metody
 - testowanie ścieżek
 - testowanie mutacyjne

Testowanie białej skrzynki testowanie ścieżek

- *path testing*
- cel:
 - sprawdzenie poprawności działania wszystkich instrukcji wchodzących w skład programu
- założenie:
 - źródłem błędów są defekty tkwiące w instrukcjach programu
- kluczowe znaczenie przy projektowaniu przypadków testowych
 - rozmieszczenie instrukcji warunkowych decydujących o wyborze ścieżki obliczeń

Testowanie białej skrzynki

testowanie ścieżek

- przygotowanie testów wymaga
 - znalezienia wszystkich rozgałęzień programu
 - umieszczenia w punktach wyjścia pułapek
 - przekazujących sterowanie do programu uruchomieniowego
- tak przygotowany program
 - może być wielokrotnie wykonywany w środowisku testowym
- każde działanie pułapki jest rejestrowane
 - potem działanie programu jest wznowiane
 - i przebiega dalej do osiągnięcia końca

Testowanie białej skrzynki

testowanie ścieżek

- wartości danych testowych mogą być
 - wybierane przez testera
 - generowane losowo
- koniec testowania
 - wyznacza osiągnięcie pełnego pokrycia programu
- wyniki zarejestrowane dla wszystkich wykonanych przypadków testowych
 - porównywane z wartościami obliczonymi na podstawie specyfikacji
 - jeśli zgodne → testowany program uznany za poprawny
- wada metody
 - duża liczba przypadków testowych
 - tym samym duży rozmiar danych do analizy po zakończeniu testów

Testowanie białej skrzynki

testowanie mutacyjne

- *mutation testing*
- cel
 - identyfikacja skutecznych przypadków testowych
 - ograniczenie rozmiaru danych
 - rejestrowanych podczas testów
 - analizowanych po zakończeniu testów
- zasada działania:
 - odrzucenie wyników przypadków testowych uznanych za nieskuteczne

Testowanie białej skrzynki

testowanie mutacyjne

- rozróżnienie skutecznych i nieskutecznych przypadków testowych
 - na podstawie wyniku wykonania tego samego testu dla dwóch wersji programu
 - oryginalnej
 - zmutowanej przez celowe wprowadzenie pewnej liczby defektów
- uzasadnienie metody
 - założenie, że defekty "naturalne" są wykrywane równie skutecznie co "sztuczne"
- defekt sztuczny
 - gdy wynik jest inny w programie oryginalnym niż w zmutowanym
- przypadek testowy wykrywający defekt sztuczny
 - jest uznawany za skuteczny
- przypadek testowy, który nie wykrywa defektu i nie rozróżnia mutanta
 - uznawany za nieskuteczny

Testowanie białej skrzynki

testowanie mutacyjne

- przygotowanie testów mutacyjnych nie wymaga statycznej analizy programu
 - mutacje mogą być generowane losowo
 - np. losowa zmiana znaków w kodzie programu z uwzględnieniem poprawności składni danego języka programowania
- zwykle generuje się dużą liczbę mutantów
- wszystkie mutanty są komplilowane i wykonywane równolegle z oryginalnym programem
 - na tych samych danych testowych
- jeśli wyniki mutantów i oryginalnego programu są takie same
 - przypadek testowy jest uznawany za nieskuteczny
 - jego wyniki są ignorowane
- jeśli wynik testowanego programu różni się od co najmniej jednego mutanta
 - przypadek testowy uznaje się za skuteczny
 - wynik obliczony przez testowany program jest rejestrowany do późniejszej analizy
- wyniki uzyskane przez mutantów
 - nie mają znaczenia
 - nie podlegają rejestrowaniu

Testowanie białej skrzynki

testowanie mutacyjne

- zalety
 - łatwość automatyzacji
- wady
 - ograniczenie mutacji sterujących wyborem rejestrówanych przypadków testowych
 - do defektów, które mogą być wynikiem drobnych pomyłek programisty
 - konieczność posiadania
 - automatycznych narzędzi wspomagających
 - wydajnego środowiska testowego
 - zdolnego do równoległego wykonywania programu i mutantów

Agenda

Poziomy testowania

Proces testowania

Metody testowania

Automatyzacja testowania

Debugowanie

Testowanie – demo

Automatyzacja testowania

- sterowniki i namiastki
- automatyzacja testów jednostkowych
- narzędzia pokrycia kodu
- odtwarzanie działań użytkownika
- testowanie wydajności

Automatyzacja testowania

sterowniki i namiastki

sterownik (*test driver*)

- program główny, który wywołuje funkcję/metodę poddawaną testom – z odpowiednimi argumentami

namiastka (*test stub*)

- program symulujący podczas testów zachowanie nieistniejącego jeszcze komponentu

Automatyzacja testowania automatyzacja testów jednostkowych

- testowanie jest procesem
 - masowym
 - wielokrotnie powtarzanym
- JUnit
 - wzorzec wyznaczający kierunek rozwoju automatyzacji testów jednostkowych
 - napisany dla programów w Javie
 - ale są odpowiedniki dla innych języków

Automatyzacja testowania automatyzacja testów jednostkowych

- typowy proces przeprowadzenia testu
 1. utworzenie obiektu testowanej klasy
 2. wywołanie metody tego obiektu
 3. porównania wyników z oczekiwaniami
- testy są zaprogramowane jako kod w danym języku programowania
 - jako klasy dołączane do testowanego programu
 - komplilowane i wykonywane za pomocą dołączonego do biblioteki sterownika *JUnit test runner* na konsoli tekstowej lub w środowisku graficznym

Automatyzacja testowania narzędzia pokrycia kodu

- ręczne obliczanie miar kodu jest uciążliwe
- potrzebne
 - wspomaganie przez narzędzia pokrycia kodu
code coverage tool
- typowe działanie narzędzi
 1. statyczna analiza kodu i znalezienie wszystkich rozgałęzień
 2. instrumentacja kodu
 - dołączenie funkcji rejestrujących przejście programu przez każde rozgałzienie
 3. rejestracja śladu wykonania programu podczas testów i obliczenie wartości miar

Automatyzacja testowania

odtwarzanie działań użytkownika

- automatyzacja (symulowanie) wykorzystania graficznego interfejsu użytkownika
 - przechwycenie i zapamiętanie działań operatora
 - kliknięcie przycisków, ruchy myszy, wypełnianie pól formularza,
 - ...
 - pierwszy przebieg testów
 - obsługiwany interaktywnie przez użytkownika
 - później można odtwarzać wszystkie zapamiętane działania
 - i porównywać zgodność wyników
- wspomaganie narzędziem *record and playback*

Automatyzacja testowania

testowanie wydajności

- symulowanie jednoczesnego działania wielu użytkowników
 - za pomocą robota testowego
- robot testowy zwykle nie wykorzystuje interfejsu użytkownika
 - tylko przesyła dane najłatwiej dostępnym kanałem
 - na przykład za pomocą protokołów HTTP, HTTPS, JDBC, ...
- potrzebne rejestrowanie wyników, np.
 - strumieni danych wejściowych i wyjściowych
 - analiza dziennika systemu
 - użycie procesora, zajętości pamięci

Agenda

Poziomy testowania

Proces testowania

Metody testowania

Automatyzacja testowania

Debugowanie

Testowanie – demo

Czym jest debugowanie?

- **znanie defektu i naprawienie go**
- czym się różni od testowania?
- testowanie nastawione jest na **wykrycie** defektu
 - jeszcze nie wiemy, czy na pewno tam jest
- w debugowaniu **wiemy, że defekt jest**
 - chociaż na początku może niezbyt dokładnie gdzie
- w czasie debugowania znalezienie defektu i zrozumienie go zwykle stanowi zwykle ok. 90% czasu pracy
 - inne źródła: 75%
 - co najmniej 50%

Koszty

- koszty defektów – ok. 60 mld dolarów
 - dla całej gospodarki USA, rocznie
- usprawnienie debugowania i testowania może ograniczyć te koszty o $\frac{1}{3}$
- największym problemem debugowania **nie jest czas i koszty**
- ale, że **czas jest nieprzewidywalny**
 - czasem kilka minut, godzin
 - a czasem dni, tygodnie

Historia debugowania

Epoka kamienia
łupanego

- urządzenia fizyczne

Epoka brązu

- print

Średniowiecze

- runtime

Obecnie

- automatyczne debugery

Najbliższa
przyszłość

- integracja debugowania z komplikacją?

5 kroków debugowania

zdefiniuj i ustabilizuj

zbadaj przyczynę

napraw defekt

przeprowadź testy regresji

zastosuj wyciągnięte wnioski

1. zdefiniuj i ustabilizuj

- zwykle na początku należy skategoryzować defekt i dokładnie zdefiniować problem
- defekty często ujawniają się w trakcie użytkowania
 - po stronie użytkowników
- dlatego często należy zacząć od
 - zweryfikowania parametrów konfiguracyjnych
 - sprawdzenia konfiguracji instalacyjnej
 - sprawdzenia procedur działania
- następnie należy sprawdzić możliwość powtórzenia procedury w normalnych warunkach działania
 - tzn. odtworzenie warunków ujawniania się defektu
- niektóre defekty wydają się ujawniać losowo
 - sporadycznie i w nieznanych warunkach
- **należy koniecznie prześledzić defekt i zrozumieć warunki w których powstał**

2. zbadaj przyczynę

- należy zbudować hipotezę dotyczącą defektu
- przydatny może być wstępny przegląd elementów
 - w celu ograniczenia poszukiwania defektu do mniejszego obszaru
- następnie można przeprowadzić dynamiczny przegląd działającego kodu
- bazując na postawionej hipotezie można budować **pozytywne** i **negatywne** testy – odpowiednio w celu **odtworzenia** lub **braku odtworzenia** defektu
- należy tak postępować aż do wyśledzenia miejsca w kodzie, które prawdopodobnie powoduje problemy
 - na przykład za pomocą poszukiwania binarnego lub zaawansowanych technik debugowania

3. napraw defekt

- defekt należy naprawić po upewnieniu się, że problem jest już całkiem zrozumiały
- w razie potrzeby należy korzystać z narzędzi kontroli wersji
 - w celu wycofania wprowadzonych zmian
- istotne jest, żeby poprawka nie wprowadzała nowego defektu
 - należy zastosować jak najmniejszą poprawkę eliminującą defekt
 - należy ograniczyć efekty uboczne poprawek
 - na przykład – nie należy próbować jednocześnie naprawić innych defektów
 - należy wprowadzać jedną zmianę w danym czasie i udokumentować lokalnie naprawienie defektu

4. przeprowadź testy regresji

- należy zweryfikować postawioną hipotezę o poprawności źródła defektu
 - uruchamiając pozytywne i negatywne testy dotyczące tego defektu
- pozytywny test wykrywa defekt odtwarzając warunki, w których się ujawnił
 - taki test tylko potwierdza istnienie defektu
- po wyeliminowaniu defektu nie powinien się więcej pojawiać
 - należy utworzyć negatywny test do uniknięcia defektu
 - aby sprawdzić czy istnieją inne problemy
- należy wykonać testy w zestawie testów regresji opracowanych dla tego modułu
 - aby sprawdzić czy nie zostały wprowadzone inne defekty
- na koniec należy dodać pozytywne i negatywne testy do zestawu testów regresji
 - aby upewnić się, że ten defekt nie pojawi się ponownie

5. zastosuj wyciągnięte wnioski

- należy zastosować wnioski wyciągnięte w czasie naprawiania defektu
- defekty zwykle ujawniają się w powiązanych zbiorach
 - jeśli zdiagnozujemy i naprawimy jeden defekt
 - powinniśmy sprawdzić, czy w ten sposób da się również wykryć inne defekty
- należy w tym kroku przeanalizować przyczynę tego defektu, np.
 - założone niepuste pole wejściowe
 - założone dwie wartości w określonej kolejności
 - brak inicjalizacji zmiennej
 - wyjście poza granicę tablicy
- czy takie kategorie defektów pojawiają się w innych miejscach kodu?

Techniki debugowania

- upraszczanie awarii
- debugowanie delta
- plasterkowanie (*slicing*)

Upraszczanie awarii

- wyobraźmy sobie sytuację:
 - startujący samolot ulega awarii – zapala się
 - jaka może być przyczyna?
-
- awaria wystąpiła w następujących warunkach
 - pasażerowie wewnątrz samolotu
 - bagaż w luku bagażowym
 - system rozrywki uruchomiony
 - klimatyzacja włączona
 - ...
 - czy wszystkie te czynniki mają wpływ?
 - wyobraźmy sobie że możemy odtworzyć tę sytuację
 - lub przeprowadzić na simulatorze

Awaria samolotu

- czynniki:
 - pasażerowie wewnątrz samolotu
 - bagaż w luku bagażowym
 - system rozrywki uruchomiony
 - klimatyzacja włączona
- problem nadal występuje
- czynniki:
 - ~~– pasażerowie wewnątrz samolotu~~
 - bagaż w luku bagażowym
 - system rozrywki uruchomiony
 - klimatyzacja włączona
- problem nadal występuje

Awaria samolotu

- czynniki:
 - ~~pasażerowie wewnątrz samolotu~~
 - ~~bagaże w luku bagażowym~~
 - system rozrywki uruchomiony
 - klimatyzacja włączona
- problem nadal występuje
- czynniki:
 - ~~pasażerowie wewnątrz samolotu~~
 - ~~bagaże w luku bagażowym~~
 - ~~system rozrywki uruchomiony~~
 - klimatyzacja włączona
- problem nadal występuje

Awaria samolotu

- czynniki:
 - ~~pasażerowie wewnątrz samolotu~~
 - ~~bagaże w luku bagażowym~~
 - ~~system rozrywki uruchomiony~~
 - ~~klimatyzacja włączona~~
- problem nadal występuje
- dostrzegamy metalową blachę na pasie startowym
- po jej usunięciu problem nie występuje
- zatem mamy ograniczony zestaw czynników powodujących awarię
 - prostszy w analizie i naprawie

Błąd Mozilli #24735

Summon comment box

anantk 2000-01-21 16:57:51 PST	Description
Ok the following operations cause mozilla to crash consistently on my machine -> Start mozilla -> Go to bugzilla.mozilla.org -> Select search for bug -> Print to file setting the bottom and right margins to .50 (I use the file /var/tmp/netscape.ps) -> Once it's done printing do the exact same thing again on the same file (/var/tmp/netscape.ps) -> This causes the browser to crash with a segfault	

- Mozilla Bugathon
 - ochotnicy mogą upraszczać przypadki testowe dla projektu Mozilla
- strona HTML bugzilla.mozilla.org zawiera wiele elementów
 - usuwanie kolejnych fragmentów HTMLa
 - aż zostanie niewielki fragment strony który wciąż powoduje problem
 - ale jeszcze mniejszy fragment nie powoduje problemu
- uproszczona zawartość strony powodująca awarię
<SELECT>

Po co upraszczać?

- łatwiejsze zrozumienie problemu
- prostsze wyjaśnienie
- mniej danych potrzebnych do odtworzenia defektu
 - jak nie mamy uproszczonej przyczyny, czy załączać zawartość całego dysku w zgłoszeniu defektu?
- szybsze testowanie
 - prostsze środowisko testowe

Albert Einstein:

*Wszystko powinno być tak proste, jak to tylko możliwe,
ale nie prostsze.*

Jak szybko uprościć dane wejściowe?

- nie da się ręcznie
 - ale można/trzeba napisać program upraszczający
 - jak zaimplementować?
 - rozwiązanie: przeszukiwane binarne
-
1. usuń połowę danych i sprawdź, czy wynik jest nadal błędny
 2. jeśli nie – wróć do poprzedniego stanu i usuń drugą połowę danych
-
- jak to zaimplementować w kodzie?

Implementacja algorytmu

```
def simplify(s):
    assert test(s) == "FAIL"

    split = len(s) / 2
    s1 = s[:split]
    s2 = s[split:]

    if test(s1) == "FAIL":
        return simplify(s1)
    if test(s2) == "FAIL":
        return simplify(s2)

    return s
```

Wada poszukiwania binarnego

- może nie uprościć wystarczająco danych wejściowych, np.
 - błędne dane znajdują się po środku – w obu połowach

```
<select>abc</select>
```

```
<select>ab
```

```
<sele  
ct>ab
```

- więc zwracany ciąg: **<select>ab**
- potrzebne lepsze rozwiążanie
 - dzielenie nie tylko na pół, ale również na mniejsze części
 - czwartki, ..., pojedyncze znaki

Debugowanie delta

1. podziel dane na n podzbiorów
 - początkowo n = 2
 2. jeśli usunięcie jednej z tych części wciąż daje błędny wynik
 - wybieramy część z błędny wynikiem
 3. w przeciwnym wypadku
 - zwiększamy liczbę podzbiorów: n = 2n
 - przechodzimy ponownie do kroku 1
- przykład – po kroku 3: n = 4
 - ciąg znaków ma długość 10, więc usuwane podciągi będą miały 2 lub 3 znaki

<select>abc</select>

<select>ab

<sele

ct>ab

elect>ab

<s ct>ab

<sele ab

<select>

co robić, jak znajdziemy podzbiór generujący błędny wynik?

w kroku 2 – zmniejszamy n o 1

ale musimy mieć co najmniej 2 podzbiory,
więc: n = max(n-1, 2)

Debugowanie delta

1. podziel dane na n podzbiorów
 - początkowo n = 2
 2. jeśli usunięcie jednej z tych części wciąż daje błędny wynik
 - wybieramy część z błędym wynikiem i $n = \max(n-1, 2)$
 3. w przeciwnym wypadku
 - zwiększamy liczbę podzbiorów: $n = 2n$
 - przechodzimy ponownie do kroku 1
- przykład – teraz $n = 3$
 - więc trzy podzbiory – pozostało 8 znaków, więc podział np. `<s | ele | ct>`

```
...
    elect>ab
<s  ct>ab
<sele ab
<select>
    elect>
<s  ct>
<sele
```

podział nie przynosi efektu – dla uproszczonych danych nie występuje awaria

więc w kroku 3 zwiększamy $n = 2n = 2*3 = 6$
mamy 6 podzbiorów – po 1 lub 2 znaki

podział nie przynosi efektu – dla uproszczonych danych nie występuje awaria

więc w kroku 3 zwiększamy $n = 2n = 2*6 = 12$
ale mamy tylko 8 znaków

więc w kroku 3 możemy n zwiększyć tylko do długości ciągu
 $n = \min(2n, \text{len}(s))$

Debugowanie delta

1. podziel dane na n podzbiorów
 - początkowo n = 2
 2. jeśli usunięcie jednej z tych części wciąż daje błędny wynik
 - wybieramy część z błędym wynikiem i $n = \max(n-1, 2)$
 3. w przeciwnym wypadku
 - zwiększamy liczbę podzbiorów: $n = \min(2n, \text{len}(s))$
 - przechodzimy ponownie do kroku 1
- przykład – teraz n = 8
 - usuwamy po jednej literze

...

```
select>
< elect>
<s lect>
<se ect>
<sel ct>
<sele t>
<selec >
<select
```

podział nie przynosi efektu – dla uproszczonych danych nie występuje awaria

a już usuwaliśmy po jednej literze wniosek – nie da się bardziej uprościć danych!

Debugowanie delta – inne zastosowania

- nie tylko do danych wejściowych jako ciągów znaków
- na przykład w zmianach/aktualizacjach oprogramowania
 - zmiana związana z aktualizacją pewnych bibliotek
 - ta zmiana obejmuje 10.000 linii kodu
 - po aktualizacji program nie działa
 - zastosowanie debugowania delta
 - podział zmiany na mniejsze części
 - być może problem powoduje konkretna pojedyncza linia kodu

Plasterkowanie

- ang. *slicing*
- polega na okrojeniu analizowanego programu, aby w określonym miejscu programu oryginalnego i okrojonego wybrane zmienne miały te same wartości
- kryterium plasterkowania
 - określony punkt programu
 - zbiór zmiennych z danego punktu programu
- rodzaje:
 - wycinek wstecz – *backward slice* (statyczny)
 - wycinek wprzód – *forward slice* (statyczny)
 - kotlet – *chop* (statyczny)
 - wycinek dynamiczny

Wycinek wstecz

- podzbiór programu oryginalnego, który wpływa na wybrany jego punkt

```
int suma = 0;  
int iloczyn = 1;  
for (int i = 1; i < max_zakres; i++) {  
    suma = suma + i;  
    iloczyn = iloczyn * i;  
}  
print(suma);  
print(iloczyn);
```

jako kryterium ustawiamy instrukcję `print(suma)`;
i zmienną `suma`

instrukcje, które nie są częścią wycinka nie wpływają na wybraną zmienną
tych instrukcji mogłyby nawet nie być w kodzie

Wycinek wprzód

- podzbiór programu oryginalnego, na który wpływa wybrany jego punkt

```
int suma = 0;
int iloczyn = 1;      jako kryterium ustawiamy instrukcję
for (int i = 1; i < max_zakres; i++) {
    suma = suma + i;
    iloczyn = iloczyn * i;
}
print(suma);
print(iloczyn);
```

Kotlet – chop

- pomiędzy punktami a i b → podzbiór programu oryginalnego
 - zależny od a
 - który wpływa na b

```
int suma = 0;  
int iloczyn = 1;  
for (int i = 1; i < max_zakres; i++) {  
    suma = suma + i;  
    iloczyn = iloczyn * i;  
}  
print(suma);  
print(iloczyn);
```

pomiędzy int iloczyn = 1;
a print(suma);

brak takich instrukcji

Wycinek dynamiczny

- wykorzystuje informacje o **wykonaniu** programu
- wszystkie instrukcje, które **rzeczywiście wpłynęły** na zmienną w danym miejscu programu
 - nie instrukcje, które **mogłyby mieć potencjalny wpływ** w ogólnie określonym przebiegu programu
- szczególnie przydatne w błędnym przebiegu
 - nie tylko – co mogło się zdarzyć
 - ale – **co się rzeczywiście zdarzyło w tym konkretnym przebiegu**

Zastosowania plasterkowania

- lepsze zrozumienie programu
- specjalizacja
 - wycinek to wyspecjalizowany program
- porównywanie programów
- testowanie
 - które fragmenty nie wymagają ponownego testowania
- wykrywanie niebezpiecznych fragmentów programu

Agenda

Poziomy testowania

Proces testowania

Metody testowania

Automatyzacja testowania

Debugowanie

Testowanie – demo

Podsumowanie

**testowanie głównym sposobem
zapewnienia jakości**

Pytania



Źródła

- Sacha K., Inżynieria oprogramowania, PWN 2010
- Górski J. (red.), Inżynieria oprogramowania w projekcie informatycznym, Mikom 2000
- Patton R., Testowanie oprogramowania, Mikom 2002
- Best Practices for Software Programming. Maintaining Quality Code, Skillsoft, materiały szkoleniowe
- Zeller A., Software Debugging, Udacity, materiały szkoleniowe, 2012
<https://www.udacity.com/course/cs259>
- Dependence Graphs and Program Slicing, CodeSurfer Technology Overview, GrammaTech, Inc.
<http://www.grammotech.com/research/publications/dependence-graphs-and-program-slicing>
- Kolawa A., The Evolution of Software Debugging
<http://www.parasoft.com/jsp/products/article.jsp?articleId=490>

Następny wykład

Ewolucja oprogramowania i/lub Ryzyko w projektach informatycznych

Inżynieria oprogramowania

Wykład 10:

Testowanie c.d.

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Wprowadzenie do testowania jednostkowego
Proste testy jednostkowe
Implementowanie testów jednostkowych
Cechy poprawnych testów jednostkowych
Projekt i testowanie

Plan

Poziomy testowania

1. Testowanie jednostkowe

„sprawdza działanie poszczególnych części oprogramowania, które mogą być poddane testom, bez połączenia z innymi częściami”

2. Testowanie integracyjne

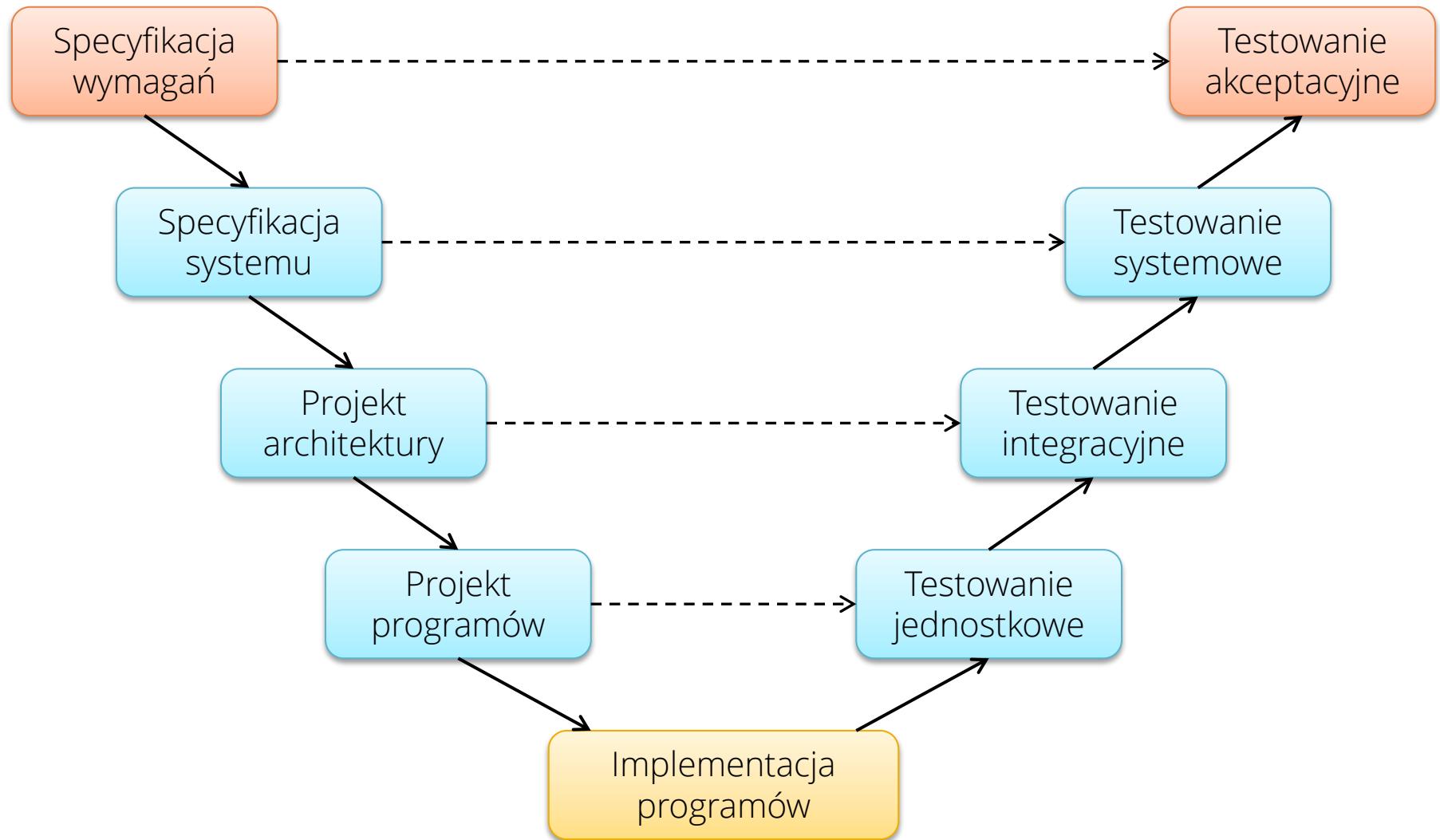
- „*proces weryfikacji interakcji między komponentami oprogramowania*”
- ciągła działalność w której inżynierowie oprogramowania muszą koncentrować się na perspektywie poziomu, który testują, w oderwaniu od niższego poziomu (poszczególnych komponentów)

3. Testowanie systemowe

- „*dotyczy zachowania całego systemu*”
 - większość błędów funkcjonalnych została zidentyfikowana wcześniej
 - skupienie się na wymaganiach niefunkcjonalnych i zewnętrznych interfejsach do innych aplikacji, narzędzi, urządzeń, itp.

[SWEBoK, 2004]

Model V



Wprowadzenie do testowania jednostkowego

Proste testy jednostkowe

Implementowanie testów jednostkowych

Cechy poprawnych testów jednostkowych

Projekt i testowanie

Plan

Czym jest test jednostkowy?

- Fragment kodu, który sprawdza działanie małej, dokładnie określonej funkcjonalności
- Zwykle obejmuje wywołanie konkretnej metody w odpowiednim kontekście
 - np. umieszczenie dużej liczby na posortowanej liście i sprawdzenie, czy rzeczywiście ta liczba jest na końcu
 - np. usunięcie z ciągu znaków wybranych znaków odpowiadających określzonemu wzorcowi i sprawdzenie, czy ten ciąg rzeczywiście nie zawiera tych znaków
- **Cel**
 - **wykażanie, że kod działa zgodnie z założeniami programisty**

Po co testy jednostkowe?

Aby ułatwić pracę programistom

umożliwienie poprawienia projektu kodu

umożliwienie skrócenia czasu ręcznego poszukiwania defektu

dzięki uruchamianiu oprogramowania „na żywo”

Co chcemy osiągnąć?

- Testowanie jednostkowe może być absorbujące
 - może odciągać programistów od pisania kodu produkcyjnego
- Dlatego konieczne jest ustalenia celów testowania jednostkowego
 - Czy kod działa zgodnie z naszymi **celami**?
 - zakładając poprawność tych celów
 - Czy kod **zawsze** działa zgodnie z naszymi celami?
 - wystarczająca liczba scenariuszy testowych
 - Czy jesteśmy pewni **poprawnego** działania kodu?
 - jeśli nie → kod jest bezużyteczny
 - poznanie ograniczeń kodu
 - np. kompatybilność z wersjami bibliotek
 - np. przekazanie pustej wartości argumentu do metody → akceptowalne przez specyfikację?
 - Czy testy jednostkowe **komunikują** nasze zamiary?
 - efekt uboczny → sposób użycia kodu → forma żywej dokumentacji

Jak testować?

1. Definicja sposobu testowania metody

- jeśli mamy ogólne pojęcie o testowaniu → rozpoczęnamy implementację testów
- przed implementacją kodu produkcyjnego, lub
- równolegle z implementacją kodu produkcyjnego

2. Uruchamianie testu

- prawdopodobnie również innych testów dla tej części systemu
- wszystkie testy muszą zakończyć się sukcesem
- każdy test powinien określić swój wynik (przejście lub nie)
 - ręczna analiza wyniku testu jest bezużyteczna

Wymówki od testowania

Pisanie testów zabiera za dużo czasu

- przy pisaniu testów na bieżąco → efektywność programisty jest raczej stała
- przy testach pisanych jedynie na końcu → efektywność programisty drastycznie spada → możliwa katastrofa całego projektu

Uruchamianie testów zajmuje za dużo czasu

- zwykle nie jest to prawda
- jeśli tak się dzieje → oddzielenie czasochłonnych testów i uruchamianie ich rzadziej (np. raz dziennie lub raz na kilka dni)

Testowanie nie należy do moich obowiązków

- ale do obowiązków należy dostarczenie działającego oprogramowania
- jeśli dostarczasz kod, którego działania nie jesteś pewien, to nie wypełniasz swoich obowiązków

Nie wiem, co kod powinien robić, więc nie wiem jak go przetestować

- zatem prawdopodobnie nie powinieneś również pisać kodu produkcyjnego
- prototyp może być dobrym rozwiązaniem w celu uściślenia wymagań

Wymówki od testowania

Ale program kompliluje się poprawnie!

- nikt nie podaje takiej wymówki
- komplilacja nie jest wystarczająca

Płacą mi za pisanie kodu a nie za testowanie

- ale za pisanie działającego kodu

Nie chcę odbierać pracy testerom i zespołowi QA

- nie musisz się nimi martwić
- testowanie jednostkowe jest dla programistów
- testerzy i zespół QA zajmuje się innymi testami

Firma nie pozwoli mi na uruchamianie testów na działającym systemie

- ale testy jednostkowe nie wymagają działającego systemu

Testowanie jednostkowe w Javie

- JUnit – <http://www.junit.org/>
- inne języki
 - SUnit (Smalltalk)
 - NUnit (C#)
 - PyUnit (Python)
 - CPPUNIT (C++)
 - fUnit (Fortran)
 - JSUnit (JavaScript)

Wprowadzenie do testowania jednostkowego

Proste testy jednostkowe

Implementowanie testów jednostkowych

Cechy poprawnych testów jednostkowych

Projekt i testowanie

Plan

Przykład

napisanie testu dla następującej metody

```
public class Largest{  
    public static int largest(int[] list) {  
        //ta metoda ma zwrócić najwyższą wartość z podanej tablicy  
    }  
}
```

Jakie testy przygotować?

Planowanie testów

- jeśli przekażemy tablicę {7, 8, 9}
 - powinniśmy otrzymać wartość 9
- kolejność elementów nie powinna mieć znaczenia
 - $\{7, 8, 9\} \rightarrow 9$
 - $\{8, 9, 7\} \rightarrow 9$
 - $\{9, 7, 8\} \rightarrow 9$
- najwyższa wartość może pojawić się wielokrotnie
 - $\{7, 9, 8, 9\} \rightarrow 9$
 - nie ma znaczenia, która wartość 9 zostanie zwrócona
- tablica może mieć jeden element
 - $\{1\} \rightarrow 1$
- wartości mogą być ujemne
 - $\{-8, -7, -9\} \rightarrow -7$
 - czasami kłopotliwe w analizie kodu produkcyjnego

Przykład – pierwsza implementacja

```
package unittestingtutorial;  
public class Largest{  
    public static int largest( int[] list ){  
        int i;  
        int max=Integer.MAX_VALUE;  
        for( i=0; i<list.length-1; i++ ){  
            if( list[i]>max ) {  
                max = list[i];  
            }  
        }  
        return max;  
    }  
}
```

Czy to jest poprawne?

Testowanie prostej metody

```
package unittestingtutorial;
import junit.framework.*;
import static org.junit.Assert.*;

public class LargestTest{
    @Test
    public void testOrder() {
        assertEquals(9, Largest.largest(new int[] {8,9,7}));
    }
}
```

```
Testcase: testOrder(unittestingtutorial.LargestTest): FAILED
expected:<9> but was:<2147483647>
junit.framework.AssertionFailedError: expected:<9> but was:<2147483647>
at unittestingtutorial.LargestTest.testLargest(LargestTest.java:46)
```

Przykład – wróćmy do naszej metody

```
package unittestingtutorial;  
public class Largest{  
    public static int largest( int[] list ){  
        int i;  
        int max=Integer.MAX_VALUE;  
        for( i=0; i<list.length-1; i++ ){  
            if( list[i]>max ) {  
                max = list[i];  
            }  
        }  
        return max;  
    }  
}
```

Przykład – po poprawieniu

```
package unittestingtutorial;  
public class Largest{  
    public static int largest( int[] list ){  
        int i;  
        int max=0;  
        for( i=0; i<list.length-1; i++ ){  
            if( list[i]>max ) {  
                max = list[i];  
            }  
        }  
        return max;  
    }  
}
```

Czy teraz jest poprawnie?

Test przechodzi.

Testowanie prostej metody rozbudowa testów

```
package unittestingtutorial;
import junit.framework.*;
import static org.junit.Assert.*;

public class LargestTest{
    @Test
    public void testOrder() {
        assertEquals(9, Largest.largest(new int[] {9,8,7}));
        assertEquals(9, Largest.largest(new int[] {8,9,7}));
        assertEquals(9, Largest.largest(new int[] {7,8,9}));
    }
}
```

```
Testcase: testOrder(unittestingtutorial.LargestTest): FAILED
expected:<9> but was:<8>
junit.framework.AssertionFailedError: expected:<9> but was:<8>
at unittestingtutorial.LargestTest.testOrder(LargestTest.java:51)
```

Przykład – wróćmy do kodu

```
package unittestingtutorial;  
public class Largest{  
    public static int largest( int[] list ){  
        int i;  
        int max=0;  
        for( i=0; i<list.length-1; i++ ){  
            if( list[i]>max ) {  
                max = list[i];  
            }  
        }  
        return max;  
    }  
}
```

Przykład – po poprawieniu

```
package unittestingtutorial;  
public class Largest{  
    public static int largest( int[] list ){  
        int i;  
        int max=0;  
        for( i=0; i<list.length; i++ ){  
            if( list[i]>max ) {  
                max = list[i];  
            }  
        }  
        return max;  
    }  
}
```

Czy teraz jest poprawnie?

Test przechodzi.

Testowanie prostej metody dalsza rozbudowa testów

```
Testcase: testNegative(unittestingtutorial.LargestTest): FAILED
expected:<-7> but was:<0>
junit.framework.AssertionFailedError: expected:<-7> but was:<0>
at unittestingtutorial.LargestTest.testNegative(LargestTest.java:66)
assertEquals(9, Largest.largest(new int[] {7,8,9}));

}

@Test
public void testDuplicates() {
    assertEquals(9, Largest.largest(new int[] {9, 7, 9, 8}));
}

@Test
public void testOne() {
    assertEquals(1, Largest.largest(new int[] {1}));
}

@Test
public void testNegative() {
    assertEquals(-7, Largest.largest(new int[]{-9, -8, -7}));
}
```

Przykład – wróćmy do kodu

```
package unittestingtutorial;  
public class Largest{  
    public static int largest( int[] list ){  
        int i;  
        int max=0;  
        for( i=0; i<list.length-1; i++ ){  
            if( list[i]>max ) {  
                max = list[i];  
            }  
        }  
        return max;  
    }  
}
```

Przykład – po poprawieniu

```
package unittestingtutorial;  
public class Largest{  
    public static int largest( int[] list ){  
        int i;  
        int max = Integer.MIN_VALUE;  
        for( i=0; i<list.length; i++ ){  
            if( list[i]>max ) {  
                max = list[i];  
            }  
        }  
        return max;  
    }  
}
```

Czy teraz jest poprawnie?

Test przechodzi.

Przykład – wszystkie testy przechodzą

- Czy to oznacza, że
 - mamy wystarczającą liczbę testów?
 - nasza metoda jest prawidłowa?
- Co przy przekazaniu pustej tablicy?
 - specyfikacja nie wspomina o takiej sytuacji
 - założmy, że jest to błąd danych wejściowych
 - powinien być wyrzucony wyjątek RuntimeException

Testowanie prostej metody dalsza rozbudowa testów

```
...
public class LargestTest{
    ...
    @Test(expected = RuntimeException.class)
    public void testEmpty() {
        Largest.largest(new int[]{}));
    }
}
```

```
Testcase: testEmpty(unittestingtutorial.LargestTest): FAILED
Expected exception: java.lang.RuntimeException
junit.framework.AssertionFailedError: Expected exception:
java.lang.RuntimeException
```

Przykład – po poprawieniu

```
package unittestingtutorial;  
public class Largest{  
    public static int largest( int[] list ){  
        if (list.length == 0)  
            throw new RuntimeException("The list is empty.");  
        int i;  
        int max = Integer.MIN_VALUE;  
        for( i=0; i<list.length; i++ ){  
            if( list[i]>max ) {  
                max = list[i];  
            }  
        }  
        return max;  
    }  
}
```

Czy teraz jest poprawnie?

Test przechodzi.

Przykład - podsumowanie

- To nie oznacza, że nasza metoda jest prawidłowa na 100%
 - pomyślmy o innych testach

Kod testowy może być większy niż kod produkcyjny

NIE JEST TO SYTUACJA NADZWYCZAJNA!!!

Wprowadzenie do testowania jednostkowego
Proste testy jednostkowe
Implementowanie testów jednostkowych
Cechy poprawnych testów jednostkowych
Projekt i testowanie

Plan

Struktura testów jednostkowych

- Możliwych wiele metod testujących jedną metodą produkcyjną
 - jak we wcześniejszym przykładzie
- Kod testowy jest dla programistów
 - nie będzie dostarczony użytkownikom
 - kod produkcyjny nie może wywoływać kodu testującego!!!
- Co powinien robić kod testujący?
 1. przygotowanie niezbędnej konfiguracji do wykonywania testów
 - utworzenie obiektów, przydzielanie zasobów, ...
 2. wywołanie testowanej metody
 3. sprawdzenie, czy metoda działa, jak powinna
 4. rozmontowanie konfiguracji z kroku 1

→ arrange

→ act

→ assert

Asercje JUnit

Typ	Opis
<code>assertEquals</code>	Sprawdza, czy dwie wartości/zmienne są równe
<code>assertArrayEquals</code>	Sprawdza, czy dwie tablice są równe
<code>assertTrue</code>	Sprawdza, czy warunek jest spełniony
<code>assertNull</code>	Sprawdza, czy obiekt jest pusty
<code>assertSame</code>	Sprawdza, czy dwa obiekty wskazują na ten sam obiekt
<code>assertThat</code>	Sprawdza zgodność z podanym wzorcem
<code>fail</code>	Test zawodzi – z ew. dodatkowym komunikatem

Asercje JUnit – assertEquals

- Sprawdza, czy dwie wartości/zmienne są równe
- assertEquals([String message,] expected, actual)
 - expected i actual – double, long, Object, Object[]
 - dla Object[] – sprawdzany jest wskaźnik do obiektu, nie zawartość
 - nie zalecane, zamiast tego: assertArrayEquals(...)
 - dla wartości double – używa się
 - assertEquals(double expected, double actual, double epsilon)

```
assertEquals(9, Largest.largest(new int[] {8,9,7}));
```

```
assertEquals(20, MyUtils.customMultiply(2, 10));
```

```
assertEquals("Wynik powinien wynosić 3 1/3.", 3.33, 10.0/3.0, 0.01);
```

Asercje JUnit – assertEquals

- Sprawdza, czy dwie tablice są równe
 - assertEquals([String message,] int[] expecteds, int[] actuals)

```
assertEquals(new int[] {1, 2, 3}, mojaTablica);
```

```
assertEquals(  
    posortowanaTablica, MyUtils.sort(nieposortowanaTablica));
```

Asercje JUnit – assertTrue

- Sprawdza, czy warunek jest spełniony
 - assertTrue([String message,] boolean condition)
- lub nie jest spełniony
 - assertFalse([String message,] boolean condition)

```
assertTrue(user.loggedIn());  
assertTrue(expected.equals(result));  
assertTrue(order.getValue()>=0);  
assertFalse(order.getValue()<0);  
assertTrue(true);
```

można wykorzystać do predefiniowania wyniku testu, jeśli test nie jest jeszcze zaimplementowany

Asercje JUnit – assertNull

- Sprawdza, czy obiekt jest pusty
 - assertNull([String message,] Object object)
- lub nie jest pusty
 - assertNotNull([String message,] Object object)

```
assertNull(product.getCategory());
```

```
assertNotNull(order.getCustomer());
```

Asercje JUnit – assertEquals

- Sprawdza, czy dwa obiekty wskazują na ten sam obiekt
 - assertEquals([String message,] Object expected, Object actual)
- lub różne obiekty
 - assertNotSame([String message,] Object expected, Object actual)

assertEquals()

używa .equals() dla obiektów i == dla typów podstawowych

assertSame()

używa == żeby sprawdzić, czy dwa obiekty wskazują na ten sam obiekt

Asercje JUnit – assertThat

- Sprawdza zgodność z podanym wzorcem
 - assertThat([String reason,] T actual,
org.hamcrest.Matcher<T> matcher)

```
assertThat(emptyList.size(), equalTo(0));
```

```
assertThat(categoriesList, hasItem("cars"));
```

```
assertThat(value, allOf(greaterThan(1), lessThan(3)));
```

Asercje JUnit – fail

- Odrzuca test z podaniem komunikatu
 - fail([String message])
- używany do oznaczenia kodu, który nie powinien zostać osiągnięty

```
throw new IndexOutOfBoundsException(10);  
fail("Ten kod nie powinien się wykonać");
```

Używanie asercji

- Metoda testowa zawiera zwykle wiele asercji
 - każda asercja sprawdza inny aspekt testowanej funkcjonalności
- Gdy asercja zawiedzie
 - testowana metoda jest wstrzymywana
 - pozostałe asercje w danej metodzie nie są wykonywane!
 - ale nie stanowi to problemu → kod i tak musi być poprawiony
- Nie rozbudowujemy kodu jeśli jakakolwiek asercja zawiedzie
 - najpierw naprawić defekt
 - następnie sprawdzić wszystkie testy
 - dopiero wtedy rozbudowa kodu

Konfiguracja testów

Anotacja	Opis
<code>@Test public void <i>method</i>()</code>	metoda testowa
<code>@Before public void <i>setUp</i>()</code>	przed każdym testem przygotowanie środowiska, np. wczytanie danych, inicjacja obiektu
<code>@After public void <i>tearDown</i>()</code>	po każdym teście posprzątanie środowiska, np. usunięcie danych tymczasowych, przywrócenie wartości domyślnych
<code>@BeforeClass public void <i>setUpClass</i>()</code>	jeden raz, przed rozpoczęciem wszystkich testów wykonanie czasochłonnych operacji, np. połączenie do bazy danych
<code>@AfterClass public void <i>tearDownClass</i>()</code>	jeden raz, po zakończeniu wszystkich testów wykonanie sprzątania, np. rozłączenie z bazą danych
<code>@Ignore</code>	metoda testowa ma być zignorowana kiedy kod został zmieniony, a test nie został dostosowany albo kiedy czas działania testu jest za długi w danej chwili
<code>@Test (expected = Exception.class)</code>	zawodzi, gdy metoda nie wyrzuca danego rodzaju wyjątku
<code>@Test(timeout=100)</code>	zawodzi, gdy metoda wykonuje się dłużej niż 100 milisekund

Zestawy testów

- Umożliwiają łączenie przypadków testowych
- w IDE – można wybrać, który zestaw ma być wykonany

```
@RunWith(Suite.class)
@Suite.SuiteClasses({unittestingtutorial.LargestTest.class})
@Suite.SuiteClasses({unittestingtutorial.OtherTest.class})
public class AllTestsSuite {
    @BeforeClass
    public static void setUpClass() throws Exception { }
    @AfterClass
    public static void tearDownClass() throws Exception { }
    @Before
    public void setUp() throws Exception { }
    @After
    public void tearDown() throws Exception { }
}
```

Co powinniśmy testować?

poprawność wyników

- względem specyfikacji wymagań

warunki brzegowe

- dopasowanie do wzorca, wprowadzanie wartości pustych, bardzo wysokie wartości, niepotrzebne duplikaty

operacje/relacje w odwrotnej kolejności

- np. dla sqrt → podnieść wynik do kwadratu i porównać z pierwotną wartością

wyniki uzyskiwane różnymi metodami

- np. obliczenie liczby książek == książki dostępne + książki wypożyczone

wymuszanie warunków błędów

- np. brak miejsca na dysku, brak połączenia sieciowego, zbyt mała rozdzielcość

charakterystyka wydajnościowa

- nie tylko wydajność kodu, ale również np. wydajność przy zwiększym obciążeniu

Wprowadzenie do testowania jednostkowego
Proste testy jednostkowe
Implementowanie testów jednostkowych
Cechy poprawnych testów jednostkowych
Projekt i testowanie

Plan

Cechy poprawnych testów jednostkowych

Automatyzacja

Kompletność

Powtarzalność

Niezależność

Profesjonalizm

Automatyzacja

- Testy jednostkowe muszą być uruchamiane w zautomatyzowany sposób
 - uruchamianie testów
 - musi być łatwe, bo często powtarzane
 - 1-click, pojedyncza komenda, nawet automatycznie w tle przez IDE
 - ważna pielęgnacja takiego środowiska
 - niedopuszczanie do wprowadzania testów, które naruszają model zautomatyzowanego uruchamiania
 - również zautomatyzowanie dostępu do środowiska uruchamiania testów
 - baza danych, sieć, itp.
 - wykorzystanie obiektów imitacji umożliwia niezależność testów od zmian w środowisku wykonania kodu produkcyjnego
 - w zespole jest wielu deweloperów
 - uruchamianie testów dla całego systemu w jednym miejscu
 - sprawdzanie wyników
 - ręczne sprawdzanie przez człowieka nie jest wydajne
 - komputery wykonują powtarzalne zadania lepiej niż ludzie

Kompletność

- Trzeba przetestować wszystko, co może zanieść
- Co to oznacza w praktyce?
 1. testowanie **każdej linii** kodu i **każdej gałęzi** przepływu sterowania?
 2. testowanie **tylko wybranych fragmentów** najbardziej podatnych na defekty?
- Taki wybór zależy od specyfiki projektu
- Defekty nie są rozłożone równomiernie w kodzie
 - są części kodu bardziej narażone na defekty
 - czasem „nie poprawiaj, napisz od nowa”
 - zwłaszcza, że czasem możemy już mieć testy jednostkowe

Powtarzalność

- Powtarzanie wykonania testów w różnej kolejności zawsze tworzy te same wyniki
- Testy nie mogą zależeć od elementów środowiska poza kontrolą dewelopera
- Rozwiązanie → obiekty imitacji
 - umożliwiają wyizolowanie testowanych metod od zmian środowiska
 - więcej – na następnym wykładzie

Niezależność

- **Testy powinny**
 - być skupione na testowanej metodzie
 - być niezależne od środowiska i innych testów
- **Jeden test → jedna mała funkcjonalność**
 - może zawierać wiele asercji
 - może testować jedną metodę
 - może testować mały zestaw metod, które razem dostarczają określoną funkcjonalność
- **Czasem metoda testowa może testować jeden aspekt testowanej metody**
 - wtedy potrzebne inne metody testowe do przetestowania innych aspektów
- **Wykonanie testów nie zależy od innych testów**
 - wykonanie danego testu nie może wymuszać uprzedniego wykonania innego testu

Profesjonalizm

- Kod testowy jest co najmniej tak samo ważny jak kod produkcyjny
 - musi spełniać te same wymagania względem jakości
 - enkapsulacja, zasada DRY, niski poziom powiązań między klasami itp.
- Łatwo wpaść w pułapkę liniowego tworzenia kodu testowego
 - kodu, który wykonuje to samo zadanie, korzysta z tych samych instrukcji, zmieniając jedynie testowaną metodę lub obiekt
 - kod testowy powinien być rozwijany tak jak kod produkcyjny
 - np. wyizolowanie powtarzającego się kodu do osobnych metod
 - wywoływanych przez różne testy
 - może być przydatna enkapsulacja metody testowej do osobnej klasy
 - nie tracimy czasu na testowanie mało ważnych aspektów
 - np. bardzo rzadko występujących
 - np. proste settery / gettery

Testowanie testów

- W celu znalezienia defektów w kodzie produkcyjnym piszemy testy
- A co jeśli kod testowy zawiera defekty?
 - Czy powinniśmy pisać testy testów?
- Dwa główne rozwiązania
 1. ulepszanie testów w czasie poprawiania defektów
 - np. kiedy defekty zostały odkryte w inny sposób niż dzięki tym testom
 - sprawdzenie istniejących testów – dlaczego nie wykazały defektów?
 - dodanie nowych testów
 2. sprawdzanie testów poprzez wstrzyknięcie defektów do kodu produkcyjnego
 - np. poprzez zakomentowanie części kodu produkcyjnego

Wprowadzenie do testowania jednostkowego
Proste testy jednostkowe
Implementowanie testów jednostkowych
Cechy poprawnych testów jednostkowych
Projekt i testowanie

Plan

Umieszczenie kodu testującego

w tym samym katalogu

w podkatalogu

w równoległych gałęziach

Ten sam katalog

```
unittestingtutorial/  
 └ Largest.java  
 └ LargestTest.java
```

- Zaleta
 - LargestTest ma dostęp do składowych protected klasy Largest
- Wada
 - kod testowy pomieszany z kodem produkcyjnym
 - możliwe wspomaganie zarządzanie kodem przez IDE / narzędzie do budowania
 - najczęściej co najmniej jest to irytujące

Podkatalog

unittestingtutorial/

 └ Largest.java

 └ test/

 └ LargestTest.java

- Zaleta
 - kod testowy oddzielnie w podkatalogu
- Wada
 - kod testowy w innym pakiecie niż testowana klasa
 - LargestTest nie ma dostępu do składowych protected klasy Largest
 - chyba, że zostanie utworzona klasa pośrednia, która będzie dawała publiczny dostęp do składowych pierwotnie z mniejszym zasięgiem

Równoległe gałęzie

prod/

- └ unittestingtutorial/
 - └ Largest.java

test/

- └ unittestingtutorial/
 - └ LargestTest.java

- Zalety
 - kod testujący całkowicie odseparowany od produkcyjnego
 - dostęp do składowych `protected` testowanej klasy
 - ponieważ w tym samym pakiecie

Testowanie w projektach zespołowych

- Główna różnica między testowaniem samodzielnym i w zespole
 - potrzeba **synchronizowania** testów i kodu
- Przy wprowadzaniu nowego kodu do repozytorium
 - upewnij się, że wszystkie testy dla nowego kodu zakończyły się pomyślnie
 - każdy test w całym systemie powinien zakończyć się pomyślnie
- Zatem ogólna zasada:
 - **wszystkie testy w systemie powinny zawsze kończyć się pomyślnie**

Testowanie w projektach zespołowych

- Jak to osiągnąć?
- Wiele zespołów postępuje zgodnie z określonymi zasadami
 - jaki kod nie powinien być wprowadzony do repozytorium?
 - niekompletny
 - niekomplilowalny
 - komplilowalny, ale uniemożliwiający komplikację istniejącego kodu
 - dla którego nie zostały opracowane testy jednostkowe
 - który nie przeszedł testów jednostkowych
 - który przeszedł testy jednostkowe, ale który powoduje, że nie przechodzą inne testy jednostkowe
- Czasami zachodzi potrzeba złamania tych zasad
 - ale należy zapewnić, żeby wszyscy członkowie zespołu byli tego świadomi

Częstotliwość testowania

Po napisaniu nowej metody

- skompiluj ją i uruchom lokalne testy

Po naprawieniu defektu

- uruchom ponownie testy jednostkowe, aby sprawdzić, czy defekt rzeczywiście został usunięty

Po każdej pomyślnej komplikacji

- uruchom lokalne testy jednostkowe

Po dołączeniu nowego kodu do systemu kontroli wersji

- uruchom testy jednostkowe dla wszystkich jednostek / całego systemu

Przez cały czas

- na dedykowanej maszynie powinien być uruchomiony proces budujący kod projektu i uruchamiający wszystkie testy jednostkowe

Demo

- testy integracyjne – Arquillian
 - <http://arquillian.org/>
 - <https://github.com/paulbakker/ducttape>
 - <https://github.com/paulbakker/ducttape/blob/master/src/test/java/ducttape/managers/ProductManagerTest.java>
- testy funkcjonalne / akceptacyjne – Selenium
 - <http://docs.seleniumhq.org/>

Podsumowanie

podstawy testowania
jednostkowego, integracyjnego,
funkcjonalnego

Pytania

?



Literatura

- Hunt A., Thomas D., Pragmatic Unit Testing in Java with JUnit, The Pragmatic Programmers, 2004
- Guide to the Software Engineering Body of Knowledge, IEEE Computer Society, 2004
- Vogel L., JUnit – Tutorial,
<http://www.vogella.com/articles/JUnit/article.html>
- JUnit.org, <http://www.junit.org/>

Następny wykład

Ewolucja oprogramowania

i/lub

Ryzyko w projektach
informatycznych

Inżynieria oprogramowania

Wykład 11: Ryzyko w projektach informatycznych

Łukasz Radliński
Zachodniopomorski Uniwersytet
Technologiczny
lukasz.radlinski@zut.edu.pl

Plan

Podstawy

Ryzyko a przedsięwzięcia informatyczne

Identyfikacja ryzyka

Analiza i ocena ryzyka

Planowanie reakcji na ryzyko

Monitorowanie i kontrola ryzyka

Znaczenie

- Mały projekt = mały problem
 - mały – wystarcza programowanie
 - większy – modelowanie, opanowanie złożoności, zarządzanie

Dobre
zarządzanie projektem



Efektywne
zarządzanie niepewnością

Czym jest ryzyko?

PMI (2000)

- niepewne zdarzenie lub warunek, które, jeśli ma miejsce, ma pozytywny lub negatywny wpływ na cel projektu

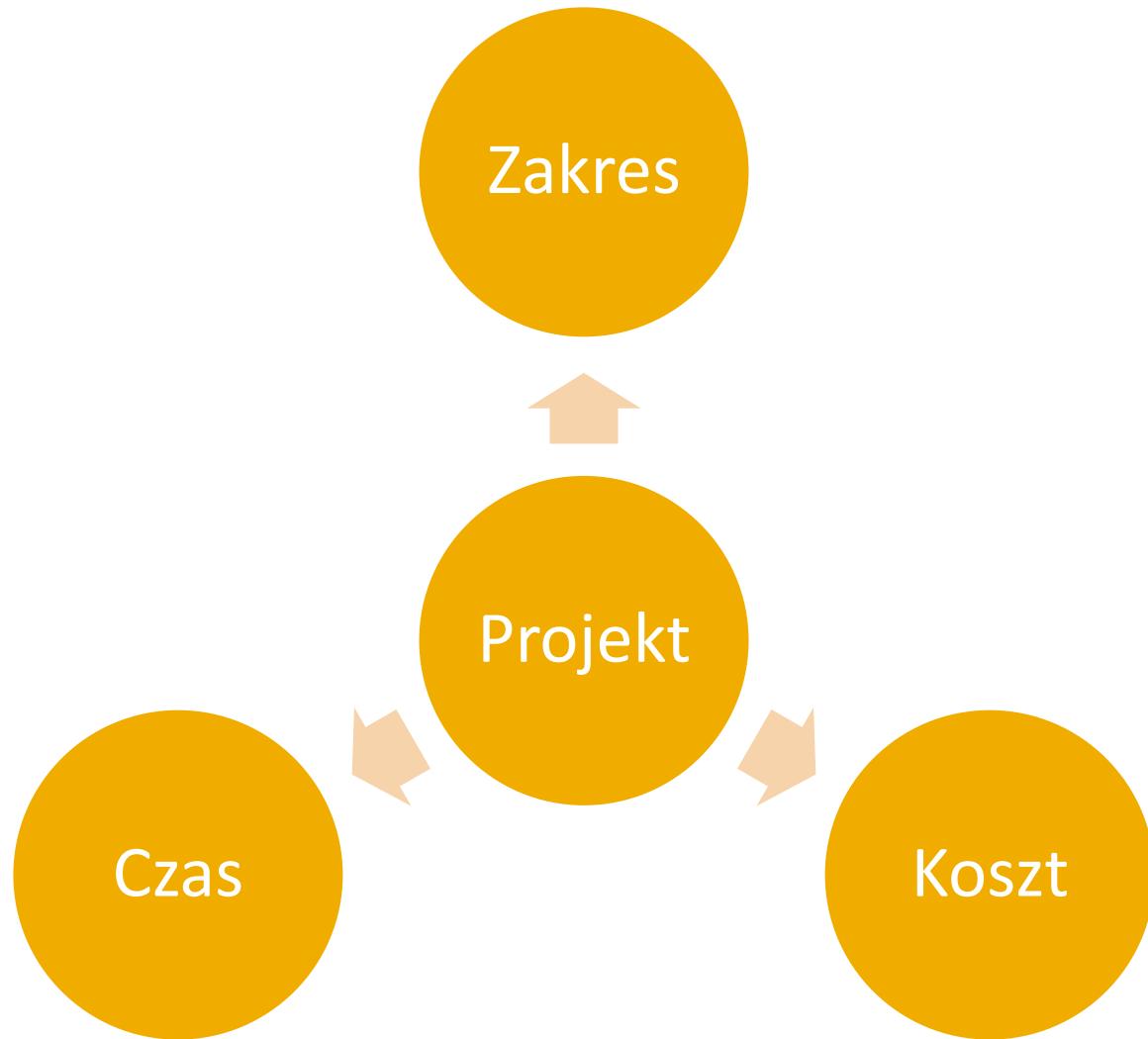
APM (1997)

- niepewne zdarzenie lub zbiór okoliczności, które, jeśli ma miejsce, będzie miało wpływ na osiągnięcie celu projektu

Metodyka PMI – PMBoK



Trzy główne wymiary projektu

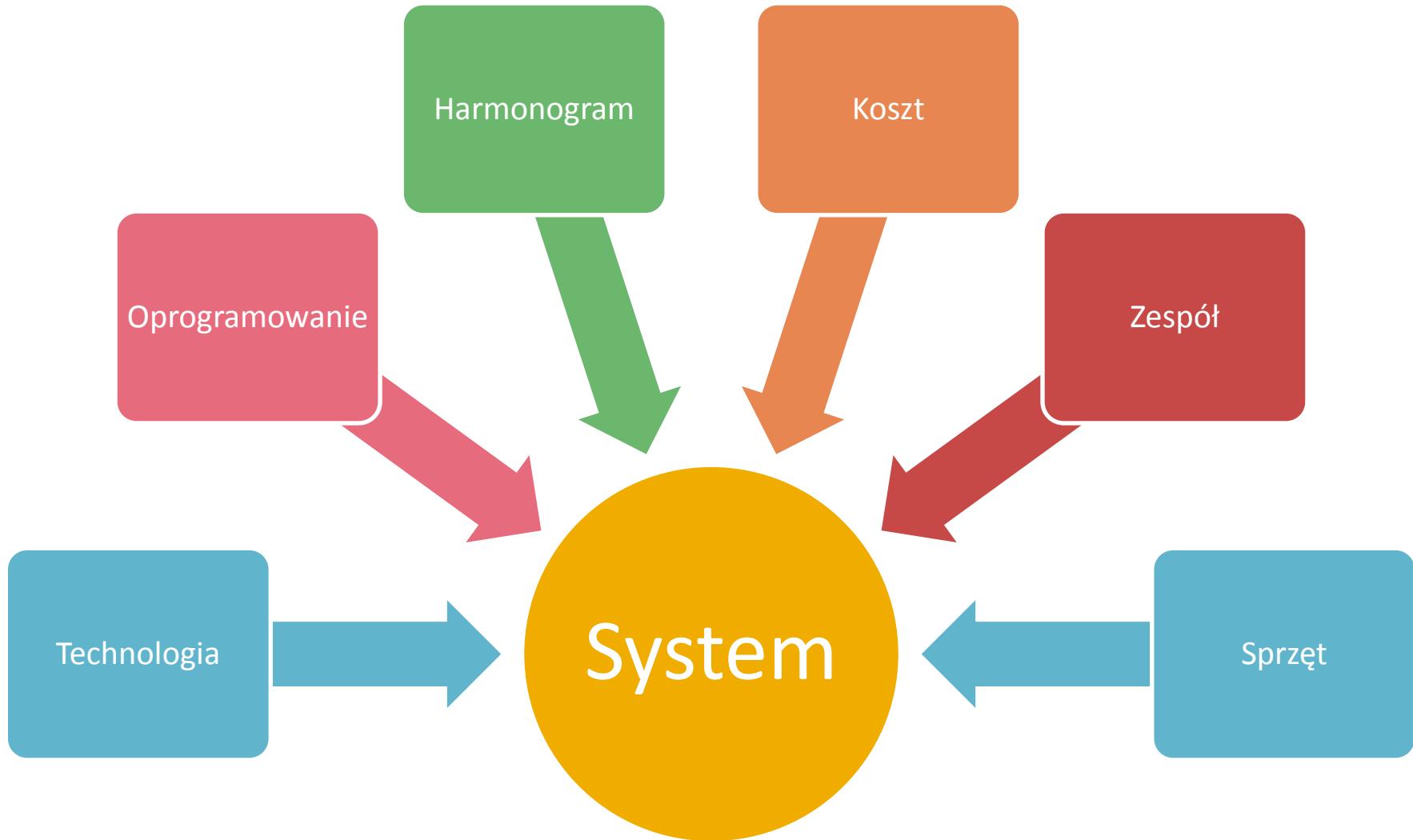


Źródła i czynniki ryzyka

Potrzeba zarządzania ryzykiem

Ryzyko a przedsięwzięcia informatyczne

Czynniki ryzyka w kontekście IT



Źródła ryzyka – I³

Intrinsic
wewnętrzne

- związane z naturą projektu → projekt to zmiana!!!

Imposed
narzucone

- obowiązujące warunki
 - np. czas ograniczający możliwość osiągnięcia celów

Induced
wprowadzone

- wynik niedostatecznego planowania lub zarządzania

Źródła ryzyka: wewnętrzne (intrinsic)

- zależność projektu od zdarzeń zewnętrznych
- nowa technologia
- priorytety technologiczne
- rozproszenie lokalizacji
- zaangażowanie trzecich osób
- systemy heterogeniczne
- brak kultury zarządzania wśród partnerów

Źródła ryzyka: narzucone (*imposed*)

- ustalony termin ukończenia
- niezmienна cena
- narzucone kryteria wydajności, bezpieczeństwa, niezawodności, itp.
- kary umowne

Źródła ryzyka: wprowadzone (*induced*)

- brak definicji celów
- brak uzgodnienia potrzeb / wymagań
- brak odpowiedzialności kierownictwa
- nietrafne planowanie / szacowanie
- brak formalnych umów z dostawcami
- brak formalnych ustaleń dotyczących zasobów
- brak jednoznacznych zobowiązań dotyczących dostaw
- brak określenia systemu zarządzania
- brak określenie systemu zapewnienia jakości

Czynniki ryzyka

- zależność od wydarzeń zewnętrznych
- założenia początkowe
- nowe technologie
- brak ekspertów
- niedoświadczony zespół
- brak doświadczenia kierownika
- geograficzne rozproszenie
- napięty harmonogram
- bardzo długie harmonogramy
- osoby trzecie (dostawcy, podwykonawcy)
- spiętrzenie prac
- szybki rozrost zespołu
- fluktuacja zespołu

Czynniki ryzyka c.d.

- wysokie wymagania odnośnie wydajności, niezawodności, zabezpieczeń, bezpieczeństwa ...
- czynniki obiektywne
- duża zmienność (wymagania, organizacja, plany ...)
- systemy heterogeniczne
- brak kultury zarządzania projektem w organizacji

Czynniki ryzyka – etap wstępny

- nieokreślony zakres projektu
- płynne terminy i plany bazowe
- źle zrozumiane wymagania użytkownika
- brakujące wymagania
- niedoszacowanie
- niejednoznaczność
- niespójność
- brak analizy wymagań pochodnych
- brak formalizacji wymagań niefunkcjonalnych
- brak kryteriów akceptacji

Czynniki ryzyka – etap realizacji

- nieudokumentowany lub niezrozumiały system realizacji projektu
- nieformalny system zarządzania zmianami
- dopuszczenie problemów nieudokumentowanych lub nieprzypisanych
- brak przeglądów prac lub ich nieefektywność
- niejasny sposób rozliczania z realizacji zadań
- brak lub źle wytyczone obszary odpowiedzialności
- niedostateczne zobowiązania kontraktowe

Czynniki ryzyka – spoza projektu

- niewypełnianie zobowiązań przez klienta
- opóźnienie dostaw
- zła jakość dostaw
- nieformalność zobowiązań
- zła jakość prac podwykonawców
- siła wyższa

Źródła i czynniki ryzyka
Potrzeba zarządzania ryzykiem

Ryzyko a przedsięwzięcia informatyczne

Cele zarządzania ryzykiem

spojrzenie w przyszłość

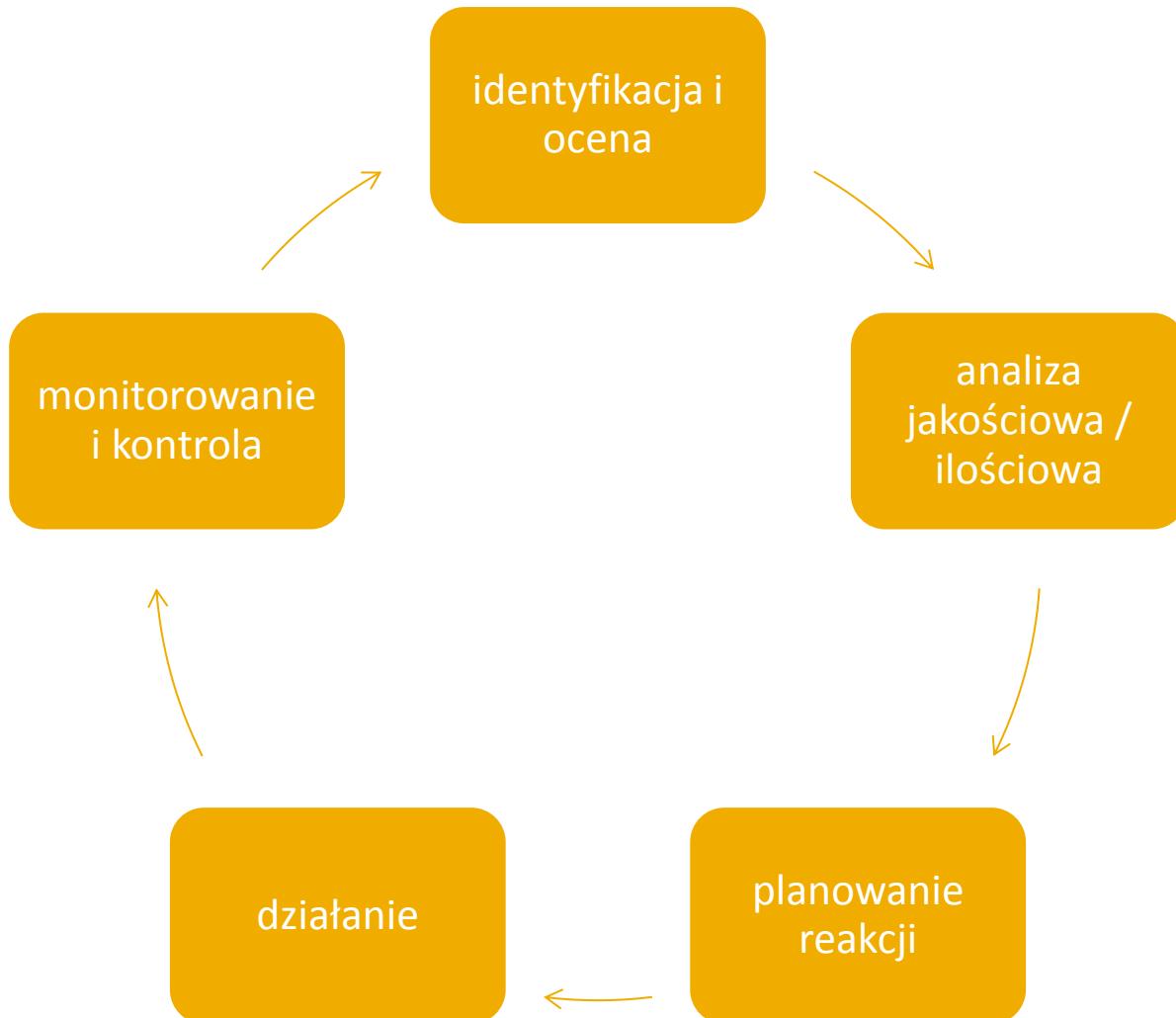
dostrzeżenie czynników, które mogą przeszkodzić w realizacji celów

realizacja działań minimalizujących ryzyko realizacji projektu

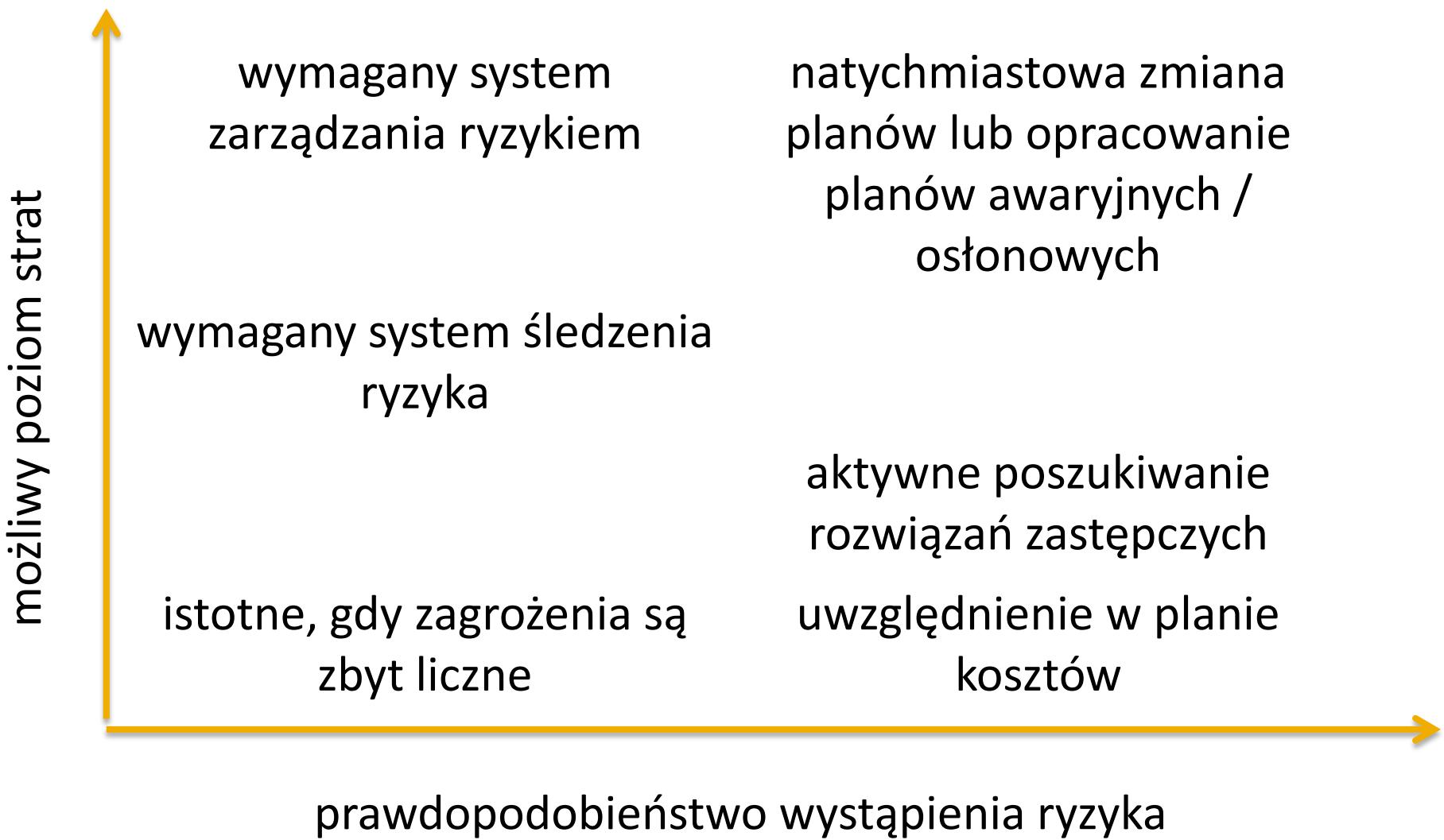
określenie i akceptacja kosztów związanych z zarządzaniem ryzykiem

Wzrost prawdopodobieństwa sukcesu

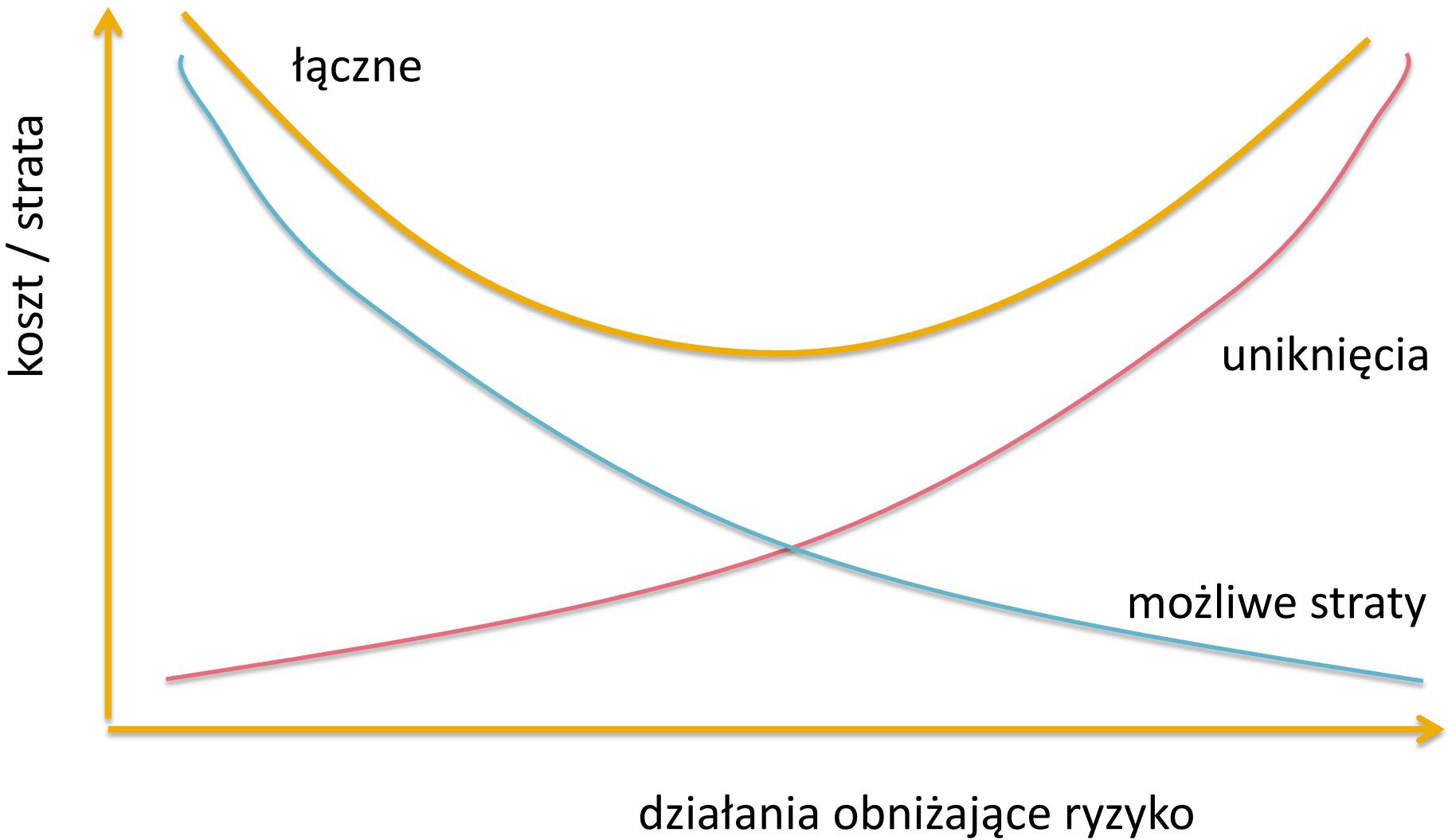
Działania



Potrzeba działań



Koszty zarządzania ryzykiem



Kiedy zarządzać ryzykiem?

- najlepiej na początku projektu?
- tylko na początku projektu?
- na etapie realizacji (implementacji)?
- na etapie testowania?
- na etapie wdrażania?
- na etapie wystawiania faktury?



przez cały czas trwania projektu!!!

Techniki i narzędzia identyfikacji ryzyka
Zakończenie

Identyfikacja ryzyka

Techniki i narzędzia identyfikacji ryzyka

- burze mózgów
- technika delficka
- wywiady
- identyfikacja przyczyn bazowych
- SWOT
- listy kontrolne / listy zagrożeń

Burza mózgów

- cel: uzyskanie obszernej listy czynników ryzyka projektu
- uczestnicy
 - członkowie zespołu
 - eksperci dziedzinowi spoza zespołu
- moderowanie dyskusji
- można wykorzystać podział kategorii ryzyka
- ryzyka
 - identyfikowane
 - kategoryzowane wg typu
 - uściślanie definicji

Technika delficka

- sposób osiągnięcia porozumienia przez ekspertów
 - anonimowy udział ekspertów
1. prowadzący przekazuje ekspertom kwestionariusze na temat istotnych czynników ryzyka
 2. odpowiedzi ekspertów są agregowane i ponownie rozsyłane po dodatkowe komentarze
- porozumienie może być osiągnięte po kilku rundach
 - ogranicza stronniczość i nadmierny wpływ pojedynczej osoby

Wywiady

- z uczestnikami projektu, interesariuszami, ekspertami
- jedno z głównych źródeł identyfikacji ryzyka

Identyfikacja przyczyn bazowych

- dogłębne badanie źródłowych przyczyn czynników ryzyka
- zalety:
 - uściśla definicję ryzyka
 - pozwala na grupowanie ryzyk wg źródeł
- skuteczna reakcja na ryzyko możliwa jeśli poprawnie zidentyfikowano źródło ryzyka

SWOT

Silne strony	[%]	Słabe strony	[%]
rozbudowana sieć placówek	14,7	brak centralnego systemu bankowego	14,7
gotowy model procesów biznesowych	8,8	duże zaangażowanie kadry w opóźniony projekt	13,3
dobre stosunki z producentem obecnego systemu	8,8	brak środków do zakończenia inwestycji	10,3
dobra grupa analityczno-projektowa	5,9	duże środki zaangażowane w projekt	10,3
dodatni bilans działalności operacyjnej	4,4	brak dostatecznej kadry	8,8
suma	42,6	suma	57,4
Szanse	[%]	Zagrożenia	[%]
możliwość rozwoju działalności detalicznej	12,2	dalszy odpływ klientów	13,5
możliwość obsługi gmin i powiatów	10,8	potrzeba dużych środków na dokończenie inwestycji	9,5
możliwość obsługi lokalnych przedsiębiorców	10,8	groźba kar umownych za zerwanie umowy	8,5
niski koszt modernizacji obecnego systemu	9,5	niejasne perspektywy serwisowania realizowanego systemu	8,1
możliwość modernizacji przez wytwórcę	9,5	groźba sporu sądowego o zerwanie umowy	6,8
suma	52,7	suma	47,3

dominują słabe strony i szanse → strategia naprawcza (usunięcie słabości przeszkadzające szansom)

Lista kontrolna

					Rating (check one)							
Fact or ID	Risk Factors	Low Risk Cues	Medium Risk Cues	High Risk Cues	L	M	H	NA	NI	TBD	Notes	
Mission and Goals												
1	Project Fit to Customer Organization	directly supports customer organization mission and/or goals	indirectly impacts one or more goals of customer	does not support or relate to customer organization mission or goals								
2	Project Fit to Provider Organization	directly supports provider organization mission and/or goals	indirectly impacts one or more goals of provider	does not support or relate to provider organization mission or goals								
3	Customer Perception	customer expects this organization to provide this product	organization is working on project in area not expected by customer	project is mismatch with prior products or services of this organization								

Lista kontrolna

- jest dobrym punktem startowym
 - opiera się na doświadczeniu
 - nie wszystkie pozycje są trafne w odniesieniu do każdego projektu
 - doskonały wstępny zestaw

nigdy nie zawiera wszystkich zagrożeń!

Proces identyfikacji ryzyka

1. sformułowanie celów
2. określenie obszarów ryzyka
3. wskazanie czynników ryzyka
4. identyfikacja ryzykownych zdarzeń
5. opisanie i udokumentowanie tych zdarzeń
6. ustalenie wstępnych priorytetów

wprowadzenie do analizy i oceny ryzyka
techniki analizy i oceny jakościowej
po oszacowaniu

Analiza i ocena ryzyka

Na czym polega analiza i ocena?

- **proces określania dwóch cech**
 - prawdopodobieństwa wystąpienia zdarzenia
 - skutków wystąpienia zdarzenia
- hierarchizacja czynników pod względem potencjalnego wpływu
- analiza łącznego ryzyka projektu
- dokumentowanie oceny ryzyka

Interakcja zdarzeń i skutków

- interakcja czynników ryzyka
- interakcja skutków ryzyka
- problemy
 - różny wpływ
 - różne oddziaływanie
 - efekt synergii, wzmacnianie, kumulowanie

wprowadzenie do analizy i oceny ryzyka
techniki analizy i oceny jakościowej
po oszacowaniu

Analiza i ocena ryzyka

Określenie prawdopodobieństwa ryzyka



Określenie wpływu

katastroficzne

- znaczące straty
- koszty poza kontrolą
- porzucenie projektu

krytyczne

- poważne straty
- przekroczenie kosztów
- przesunięcie startu
- projekt pod znakiem zapytania

marginalne

- niewielka wartość rezultatów projektu
- przesunięcie daty zakończenia możliwe do nadrobienia

pomijalne

- niepewność odnośnie rezultatów projektu

Określenie wpływu ryzyka

Defined Conditions for Impact Scales of a Risk on Major Project Objectives

(Examples are shown for negative impacts only)

Project Objective	Relative or numerical scales are shown				
	Very low /.05	Low /.10	Moderate /.20	High /.40	Very high /.80
Cost	Insignificant cost increase	<10% cost increase	10-20% cost increase	20-40% cost increase	>40% cost increase
Time	Insignificant time increase	<5% time increase	5-10% time increase	10-20% time increase	>20% time increase
Scope	Scope decrease barely noticeable	Minor areas of scope affected	Major areas of scope affected	Scope reduction unacceptable to sponsor	Project end item is effectively useless
Quality	Quality degradation barely noticeable	Only very demanding applications are affected	Quality reduction requires sponsor approval	Quality reduction unacceptable to sponsor	Project end item is effectively useless

This table presents examples of risk impact definitions for four different project objectives. They should be tailored in the Risk Management Planning process to the individual project and to the organization's risk thresholds. Impact definitions can be developed for opportunities in a similar way.

Macierz poziomu ryzyka

Probability and Impact Matrix

Probability	Threats					Opportunities				
	0.90	0.05	0.09	0.18	0.36	0.72	0.72	0.36	0.18	0.09
0.70	0.04	0.07	0.14	0.28	0.56	0.56	0.28	0.14	0.07	0.04
0.50	0.03	0.05	0.10	0.20	0.40	0.40	0.20	0.10	0.05	0.03
0.30	0.02	0.03	0.06	0.12	0.24	0.24	0.12	0.06	0.03	0.02
0.10	0.01	0.01	0.02	0.04	0.08	0.08	0.04	0.02	0.01	0.01
	0.05	0.10	0.20	0.40	0.80	0.80	0.40	0.20	0.10	0.05

Impact (ratio scale) on an objective (e.g., cost, time, scope or quality)

Each risk is rated on its probability of occurring and impact on an objective if it does occur. The organization's thresholds for low, moderate or high risks are shown in the matrix and determine whether the risk is scored as high, moderate or low for that objective.

Model kosztowy

- podatność na ryzyko = $p * \text{strata}$
 - podatność na ryzyko to oczekiwana wartość ryzyka
 - p to prawdopodobieństwo wystąpienia ryzyka
 - strata to wielkość możliwej straty
- przykład 1
 - brak dostępności zasobów: $25\% * 4 \text{ tygodnie} = 1 \text{ tydzień}$
- przykład 2
 - odejście kierownika projektu: $10\% * 200\text{K PLN} = 20\text{K PLN}$

Szacowanie ryzyka

Czynnik / zagrożenie	Kategoria	Prawdopodobieństwo	Wpływ
Większa liczba użytkowników	Produkt	30%	Marginalny
Opór użytkowników przed systemem	Działalność przedsiębiorstwa	40%	Marginalny
Utrata funduszy	Klient	20%	Katastroficzny
Zmiana wymagań przez klienta	Produkt	99%	Krytyczny
Niedoświadczani programiści	Zespół	50%	Krytyczny
Fluktuacja personelu	Zespół	75%	Krytyczny
Realizacja prac niezgodnie z procesem	Proces	50%	Krytyczny

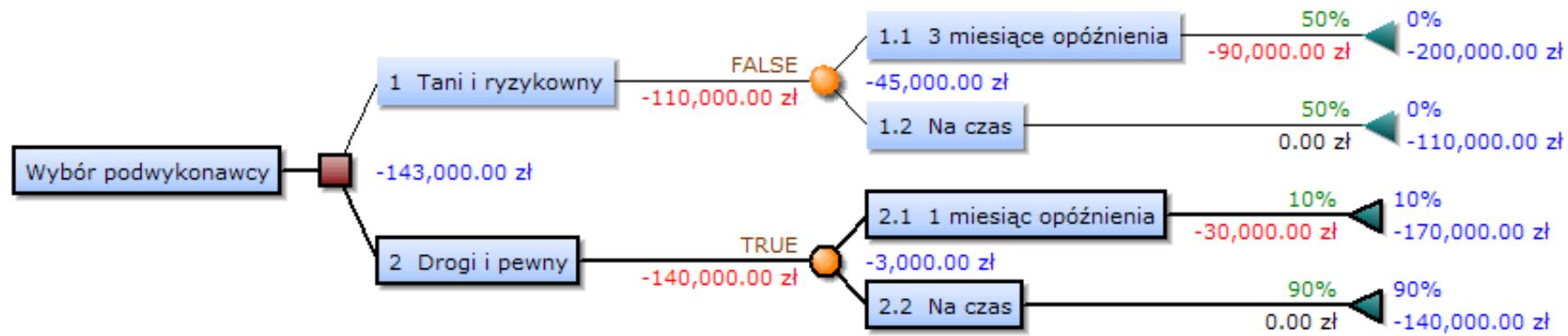
+ ew. pobieżnie określone działanie dla każdego czynnika

Szacowanie ryzyka – wersja złożona

Czynnik ryzyka /zagrożenie	Poziom	Prawdo-podobieństwo	Wpływ (koszt w zł)	Podatność na ryzyko (prawdo-podobieństwo * wpływ)
	N			
	Ś			
	W			
	N			
	Ś			
	W			

+ dodatkowa kolumna SUMA dla każdego czynnika
+ SUMA podatności dla wszystkich czynników łącznie

Ocena ryzyka – drzewo decyzyjne



wprowadzenie do analizy i oceny ryzyka
techniki analizy i oceny jakościowej
po oszacowaniu

Analiza i ocena ryzyka

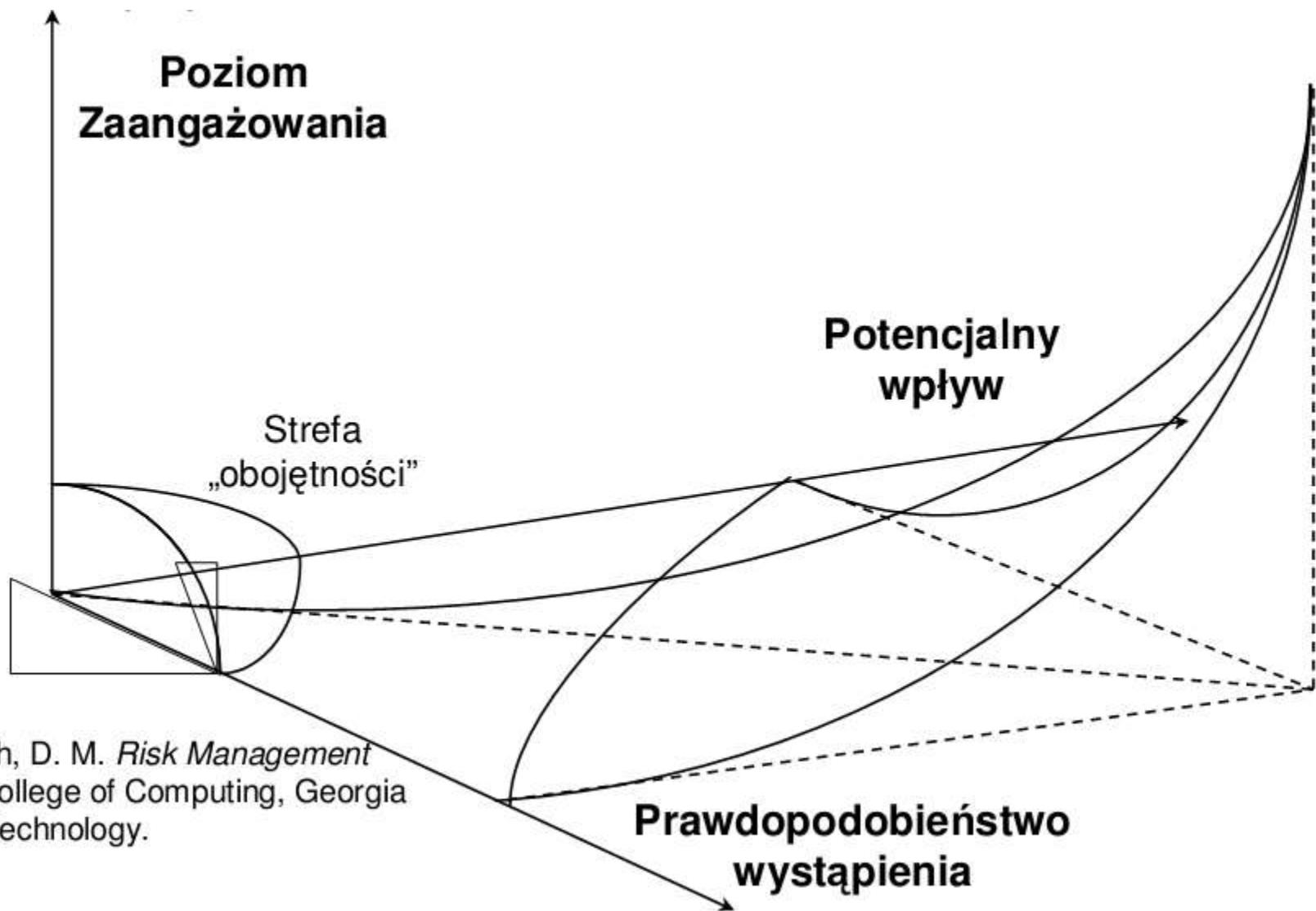
Hierarchizacja ryzyka

- wyższy priorytet
 - wysoka podatność na ryzyko
 - możliwe większe straty
 - bardziej prawdopodobne
- grupowanie czynników ryzyka
 - czynniki ludzkie
 - zakres projektu
 - technologia
 - klient ...
- identyfikacja czynników, które można
 - zignorować
 - tylko monitorować

Hierarchizacja ryzyka

- cel:
 - skupienie uwagi i działań
 - zwrócenie uwagi kierownictwa
- uporządkowanie może ulec zmianie wraz z upływem czasu

Wymagany poziom zaangażowania



Źródło: Smith, D. M. *Risk Management & Metrics*. College of Computing, Georgia Institute of Technology.

Wprowadzenie do planowania reakcji na ryzyko
Strategie postępowania

Planowanie reakcji na ryzyko

Czym jest planowanie reakcji?

Proces opracowywania wariantów postępowania dotyczących czynności zmniejszających zagrożenia i zwiększających potencjalne korzyści dla sformułowanych celów projektowych

... czyli w skrócie

co robić?
(w określonych sytuacjach)

Istotność planowania reakcji

- Kluczowy etap procesu zarządzania ryzykiem
 - opracowanie sposobów reagowania na
 - zdarzenia korzystne
 - zdarzenia niekorzystne
- Skuteczność planowania reakcji na ryzyko ma bezpośredni wpływ na wzrost lub spadek ryzyka realizacji całego projektu
- Planowane działania
 - adekwatne do skutków wystąpienia niekorzystnych zjawisk
 - likwidowanie lub niwelowanie wpływu zagrożenia w sposób efektywny kosztowo
 - realizowane terminowo

Materiały wejściowe

- plan zarządzania ryzykiem
- lista hierarchii ryzyk
- ranking projektu pod względem ryzyka
- lista zmierzonych ilościowo ryzyk uszeregowanych według priorytetów
- analiza probabilistyczna projektu
- prawdopodobieństwo osiągnięcia celów kosztowych i czasowych
- lista potencjalnych metod reakcji
- progi ryzyka
 - akceptowalne wielkości ryzyka przyjęte przez organizację
- lista dysponentów ryzyka
 - zestawienie interesariuszy, którzy mają możliwości reakcji na dane ryzyko
- lista wspólnych ryzyk
 - ewidencja zjawisk niekorzystnych, które mogą wywierać wielorakie skutki na proces realizacji projektu
- trendy będące wynikami wielokrotnie przeprowadzonej jakościowej i ilościowej analizy ryzyka

Etapy procesu planowania reakcji

1. zdefiniowanie indywidualnych planów obejmujących zamierzenia związane z działaniami względem poszczególnych zagrożeń

- cel
- działania i terminy
- odpowiedzialność
- sposób realizacji
- zasoby

2. integracja ... →

Etapy procesu planowania reakcji

- 1. zdefiniowanie indywidualnych planów obejmujących zamierzenia związane z działaniami względem poszczególnych zagrożeń**
- 2. integracja**
 - a) poszczególnych elementów we wspólnym planie zarządzania ryzykiem
 - b) tego planu w ramach planu zarządzania projektem

Rezultaty procesu planowania reakcji

- **plan reakcji na ryzyka**
- ewidencja ryzyk rezydualnych
 - lista ryzyk, które pozostają w projekcie po wdrożeniu strategii unikania, przenoszenia i łagodzenia ryzyka
- ewidencja ryzyk wtórnych
 - powstają w wyniku wdrożenia strategii reagowania na ryzyko
- postanowienia kontraktowe
 - umowy wraz zakresami odpowiedzialności, jakie przejmują na siebie inne podmioty współuczestniczące w realizacji projektu
- wielkości niezbędnych kwot rezerw projektowych
 - tzw. bufory finansowo-zasobowe zarezerwowane przez menedżerów na wypadek wystąpienia sytuacji niekorzystnych
- materiały wejściowe do innych procesów
- materiały wejściowe do weryfikacji całości planu projektu
 - rezultaty planowania reakcji na ryzyko powinny być uwzględnione w ostatecznym planie projektu

Wprowadzenie do planowania reakcji na ryzyko
Strategie postępowania

Planowanie reakcji na ryzyko

Strategie postępowania

zapobieganie ryzyku

łagodzenie ryzyka

usunięcie skutków ryzyka

rozproszenie ryzyka

akceptacja ryzyka

Zapobieganie ryzyku

- **zlikwidowanie lub zmniejszenie ryzyka wystąpienia niepożądanej sytuacji**
- inaczej – *unikanie ryzyka*
- Przykład 1:
 - Problem:
 - przewidywane są problemy z opracowaniem specjalistycznego komponentu oprogramowania
 - Rozwiązanie:
 - zakup gotowego komponentu (jeśli dostępny na rynku)
- Przykład 2:
 - Problem:
 - Brak doświadczenia w stosowaniu najnowszej wersji narzędzia
 - Rozwiązanie:
 - Użycie starszej wersji, lepiej znanej zespołowi

Łagodzenie ryzyka

- **łagodzenie skutków lub zmniejszanie prawdopodobieństwa jego wystąpienia**
- inaczej – *obniżenie ryzyka*
- Przykład 1:
 - Problem:
 - Młody i niedoświadczony zespół
 - Rozwiązanie:
 - Zatrudnienie doświadczonego lidera
- Przykład 2:
 - Problem:
 - Brak doświadczenia w stosowaniu najnowszej wersji narzędzia
 - Rozwiązanie:
 - Dodatkowe szkolenia

Jest odmianą strategii zapobiegania → czasami trudno je rozróżnić!

Usunięcie skutków ryzyka

- **opracowanie planu awaryjnego, który ma być wdrożony, kiedy wystąpi ryzyko**
- Przykład 1:
 - Problem:
 - przewidywane są problemy z opracowaniem specjalistycznego komponentu oprogramowania
 - Rozwiązanie:
 - próba zbudowania pomimo ryzyka niepowodzenia
 - w przypadku niepowodzenia – zakup gotowego komponentu

Rozproszenie ryzyka

- **ograniczenie zagrożeń za pomocą instrumentów finansowych**
 - kary umowne za niedotrzymanie terminów
 - ubezpieczanie od następstw określonych zdarzeń
 - kontrakt *fixed-price* z dostawcą/podwykonawcą
- inaczej – *transfer ryzyka*
- najczęściej stosowana w odniesieniu do ryzyka zewnętrznego i biznesowego
 - na ogół nie do ryzyka technicznego

Akceptacja ryzyka

- **przyjęcie ryzyka i niepodejmowanie żadnych działań licząc się z konsekwencjami**
 - kiedy małe prawdopodobieństwo
 - kiedy nie uda się znaleźć skutecznego przeciwdziałania
- inaczej – *zaniechanie działań*
- może być racjonalna, ale powinna wynikać ze świadomej decyzji kierownictwa
- czasami może łączyć się z próbą pewnych działań nadzwyczajnych łagodzących skutki ryzyka
 - przesunięcie niektórych celów na później
 - odzyskanie części kosztów przez wykorzystanie wyników w innym projekcie

Akceptacja ryzyka – rodzaje

- akceptacja aktywna
 - opracowanie planów rezerwowych oraz rezerw „na wszelki wypadek”
- akceptacja pasywna
 - podejmowanie działań „na bieżąco”, kiedy dany czynnik się uaktywni
- plan awaryjny
 - na wypadek wystąpienia ryzyka o wysokim wpływie
 - gdy opracowana strategia nie jest skuteczna

Wprowadzenie do monitorowania i kontrolowania
Nadzór nad ryzykiem

Monitorowanie i kontrolowanie ryzyka

Czym jest monitorowanie i kontrola?

proces

- wdrożenia planu zarządzania ryzykiem
- nieustannej obserwacji i nadzorowania zidentyfikowanych ryzyk
- identyfikacji nowo powstałych zagrożeń
- systematycznego oceniania skuteczności podejmowanych działań prewencyjnych

Cele monitorowania ryzyka

czy strategie reakcji na ryzyka wdrożono zgodnie z planem?

czy działanie podejmowane w ramach realizacji planów reakcji na ryzyko skutkują oczekiwanyimi rezultatami?

czy przyjęte założenia projektu są aktualne?

czy podczas realizacji projektu nie doszło do zmian w szczególnym i ogólnym poziomie ryzyka?

np. zgodnie z analizą trendów

czy wystąpiły czynniki wyzwalające zidentyfikowane ryzyka?

czy wystąpiły nowe ryzyka nierozniane uprzednio?

Materiały wejściowe

- **plan zarządzania ryzykiem**
- **plan reakcji na ryzyko**
- raporty będące wynikiem komunikacji podczas planowania i realizacji projektu
- zestawienia dodatkowych wyników identyfikacji i analiz ryzyka
 - wykrycie nowych źródeł ryzyka
- ewidencje proponowanych lub już wdrożonych zmian zakresów projektu

Metody i techniki

- audyty reakcji na ryzyko
- okresowe przeglądy ryzyka w projekcie
- analizy wartości wypracowanej
- zestawienia pomiarów technicznych
 - ewidencja planowanych i osiągniętych wyników podczas przebiegu realizacji projektu
- opisy dodatkowo planowanych reakcji na ryzyko

Rezultaty monitorowania i kontroli

- plany improwizacyjne
 - ewidencja reakcji na zagrożenia wcześniej nie zidentyfikowane
- listy działań korygujących
- aktualizacje planów reakcji na ryzyko
- baza danych o ryzykach w danym projekcie
 - bardzo cenne repozytorium wiedzy, umożliwiające prawidłowe planowanie zarządzania ryzykiem w przyszłych projektach
- uaktualnione listy kontrolne zidentyfikowanych ryzyk

Wprowadzenie do monitorowania i kontrolowania
Nadzór nad ryzykiem

Monitorowanie i kontrolowanie ryzyka

Nadzór

- Do adekwatnej oceny ryzyka potrzebne jest gromadzenie odpowiednich miar
 - obiektywność
 - dokładność
 - właściwy czas
- Nie zbieramy miar? → To ocena ryzyka będzie
 - bardziej subiektywna
 - mniej wiarygodna

Obszary rozpatrywania skutków

- wpływ na harmonogram
- wpływ na koszty
 - szacowane i rzeczywiste
- wpływ na jakość
 - funkcjonalność i inne miary jakości
- wpływ na zasoby
 - również ludzkie

Nadzór nad ryzykiem

- **śledzenie głównych czynników ryzyka**
 - gromadzenie informacji o głównych czynnikach wpływających na dane zagrożenie
 - zakres i czas zbierania tych informacji oraz przypisanie odpowiedzialności powinny być
 - jawnie zdefiniowane i z odpowiednim wyprzedzeniem
- **ponawianie oceny ryzyka**
 - w punktach kontrolnych określonych w planie zarządzania ryzykiem
 - w celu ustalenia nowych priorytetów i/lub ew. zmiany planów
- **uruchamianie akcji naprawczych**
 - ocena ryzyka daje podstawę do podejmowania decyzji o uruchamianiu przewidzianych w planach akcji naprawczych
 - może być na podstawie wartości progowych zmienionych w wyniku ponownej oceny ryzyka

Uruchomienie planu awaryjnego

- **uruchamiany, gdy wartość ryzyka przekroczy zdefiniowany wcześniej próg**
- czasami trudne jest przekonanie zainteresowanych stron, że rzeczywiście wystąpił poważny problem – szczególnie na wczesnych etapach projektu
- częstą reakcją jest przekonanie, że „wszystko się ułoży” w kolejnym etapie projektu
- jednak zwykle problem sam nie znika
 - bez podjęcia odpowiednich działań skutki mogą być bardziej dotkliwe
- konieczne jest przygotowanie harmonogramu realizacji planu awaryjnego
 - uniknięcie rozwleczenia go w czasie
- jeśli problem nie rozwiążany w danym czasie → realizacja planu zarządzania kryzysowego

Pytania



Źródła

- Górski J., Zarządzanie ryzykiem, [w:] Inżynieria oprogramowania w projekcie informatycznym (red. J. Górski), Mikom 2000
- Nawrocki J., Zarządzanie ryzykiem, Zaawansowana inżynieria oprogramowania – wykłady, 2006,
<http://wazniak.mimuw.edu.pl>
- Sacha K., Inżynieria oprogramowania, PWN 2010
- Szyjewski Z, Zarządzanie ryzykiem – wykłady, 2009
- A Guide to Project Management Body of Knowledge, Project Management Institute, 2004
- Wikipedia, Zarządzanie ryzykiem projektowym,
http://pl.wikipedia.org/wiki/Zarządzanie_ryzykiem_projektowym
- Generic Software Project Risk Factors, Department of Information Resources, State of Texas, 2008,
<http://www2.dir.state.tx.us/management/projectdelivery/projectframework/planning/Pages/SupplementalTools.aspx>

Następny wykład

