

# Inżynieria oprogramowania

## Wykład wprowadzający

*Aleksander Jarzębowicz*

*Katedra Inżynierii Oprogramowania  
Wydział ETI, Politechnika Gdańsk*

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich  
rozprowadzanie jest zabronione.

# Informacje kontaktowe

**dr inż. Aleksander Jarzębowicz**  
**Katedra Inżynierii Oprogramowania**

pokój 648, budynek WETI A

e-mail: [olek@eti.pg.edu.pl](mailto:olek@eti.pg.edu.pl)

tel. 1464

**konsultacje:**

**poniedziałki 11:15-12:00, czwartki 12:15-13:00**

**(fizycznie w EA 648, zdalnie na <https://discord.gg/KB6d58V>)**

# Zespół przedmiotu

## Katedra Inżynierii Oprogramowania WETI

- wykład: dr inż. Aleksander Jarzębowicz
- zajęcia laboratoryjne: dr inż. Aleksander Jarzębowicz  
dr inż. Magdalena Mazur-Milecka  
mgr inż. Małgorzata Pykała  
dr inż. Elżbieta Zamiar

# Zaliczenie przedmiotu

- **Jedna ocena końcowa z przedmiotu**
  - 50 % punkty z egzaminu i 50 % punkty z laboratorium
  - dopuszczenie do egzaminu pod warunkiem zaliczenia laboratorium
- **Egzamin**
  - Część teoretyczna (materiał z wykładów)
  - Część zadaniowa (modelowanie w UML, zakres pokrywający się z laboratorium)
  - 2 terminy (sesja podst. i popr.), bez dodatkowych
- **Laboratorium**
  - Zadania realizowane zespołowo
  - Indywidualne punkty za sprawdziany
  - Wymagane jest min. 50% punktów, w tym pozytywnej ocena każdego z zadań
- **Przypadki szczególne**
  - Powtarzanie przedmiotu – przepisywane zaliczenie laboratorium (sprawdzić w MojaPG), przepisywany wynik zdanej części egzaminu
  - Awanse (IOS, IZP, ...) – nie są dopuszczane
  - Osoby spoza list – po publikacji list czas na wyjaśnienie sytuacji

# Skala ocen

Punkty - progi	Ocena
40	3
48	3,5
56	4
64	4,5
72	5

**Max. 80 pkt**

# Organizacja - wykład

- **Materiały w postaci slajdów wykładowych dostępne, przekazywane przed wykładem**
  - serwis [enauczanie.pg.edu.pl/moodle](http://enauczanie.pg.edu.pl/moodle);
  - kurs <https://enauczanie.pg.edu.pl/moodle/course/view.php?id=30918>  
Dostęp wymaga podania kodu, kody są przydzielane na pierwszych zajęciach laboratoryjnych
- **Cały zakres wiadomości wymaganych na egzaminie będzie prezentowany na wykładach...**
- **... co nie znaczy, że każda istotna informacja musi być zapisana na slajdach wykładowych**

# Organizacja - laboratoria (1)

- **Obecności obowiązkowe**
- **Ramowy harmonogram zajęć laboratorium, wraz z terminami oddania poszczególnych zadań i punktacją jest dostępny w instrukcji laboratoryjnej (eNauczanie)**
- **Konkretnie terminy dla poszczególnych terminów lab. będą komunikowane przez prowadzących zajęcia lab.**
- **Tryb pracy w postaci cykli:**
  - informacje na wykładzie
  - sprawdzian
  - omówienie przykładu
  - samodzielne zadanie
- **Oddawanie zadań po terminie powoduje obniżenie oceny (punkty ujemne)**

# Organizacja - laboratoria (2)

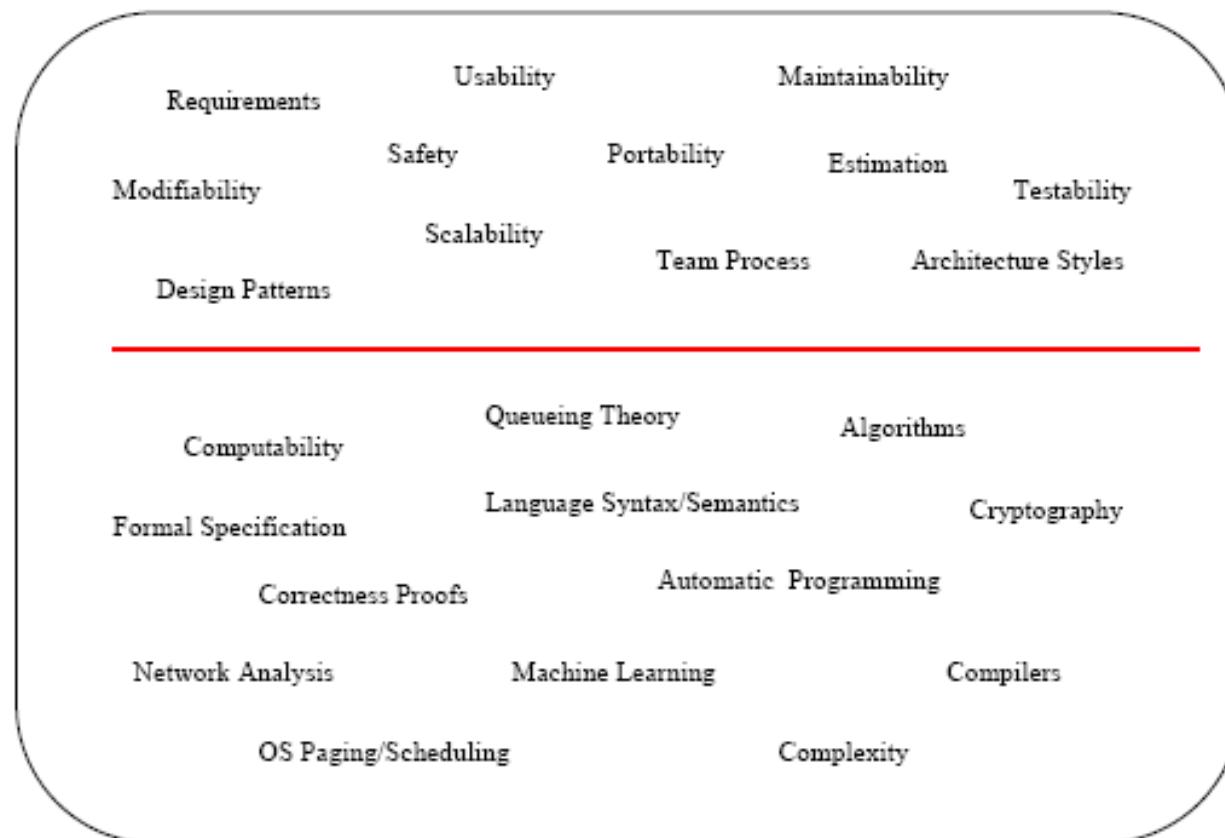
- **Terminy laboratoriów:**
  - Informatyka i Inżynieria Biomedyczna osobno
  - Limit na termin: 18 studentów
  - Każda grupa dziekańska ma 2 dostępne terminy
  - Studenci spoza grupy mogą uczestniczyć w przypadku dostępności miejsc i uzyskania zgody prowadzącego zajęcia laboratoryjne
  - Ewentualnie zamiana 1:1
- **Więcej informacji przekażą prowadzący poszczególne terminy laboratoryjne**
- **To tyle w kwestiach organizacyjnych**
- **Pytania?**

# Inżynieria oprogramowania

- **Inżynieria (def. prakt.)** - oznacza kreatywne działania, ukierunkowane na praktyczny cel, podlegające planowaniu i bazujące na powtarzalnych technikach

**Software  
Engineering:**

**Computer  
Science:**



Źródło: Ch. Connell, Software Engineering ≠ Computer Science, Dr.Dobb's, 2009

# Zderzenie perspektyw

## Student II-III roku

- Brak doświadczeń zawodowych lub w niewielkim zakresie
- Niewielki rozmiar tworzonego oprogramowania
- Praca indywidualna, ewentualnie z kolegą
- Brak rzeczywistego klienta (praca dla siebie albo dla wykładowcy)
- Problemy skoncentrowane wokół technologii lub zagadnień akademickich
- Brak konieczności utrzymania produktu

## Praktyka przemysłowa

- Profesjonalna działalność w obszarze wytwarzania oprogramowania
- Złożone systemy (mikro: do 10PM, małe: 10-100PM, duże:>1000PM)
- Zespoły kilku (-nasto, -dziesięcio, -set) osobowe
- Rzeczywisty klient mający określone potrzeby i wymagający!
- Problemy skoncentrowane wokół zastosowań, potrzeb biznesu, technologia jako środek do celu
- Produkt utrzymywany i rozwijany, nierzadko przez wiele lat

# Przedsięwzięcia inżynierskie - analogie

- **Analogia projektu informatycznego do przedsięwzięcia budowlanego**
- **Problem skali – budowa altanki na działce, a budowa galerii handlowej**
- **Nie tylko bezpośrednie prace konstrukcyjne, inne obszary do uwzględnienia to np.:**
  - Negocjacje z klientem, komunikacja
  - Planowanie przedsięwzięcia, harmonogram, przydział zasobów
  - Pozyskanie wymagań
  - Zaprojektowanie budowli (plany, modele, obliczenia)
  - Zarządzanie przedsięwzięciem (bieżący nadzór, reagowanie na problemy)
  - Zapewnienie jakości (niezawodność, spełnienie norm)
  - Kwalifikacje pracowników, uprawnienia, szkolenia
  - Narzędzia i technologie

# Dodatkowy czynnik - specyfika

- Kwestia skali i złożoności każdego przedsięwzięcia konstrukcyjnego wymuszającego podejście inżynierskie w różnych dziedzinach
  - budownictwo
  - mechanika
  - elektronika
  - przemysł chemiczny
  - ...
- Kwestia dodatkowa – specyfika oprogramowania ...

# Na czym polega specyfika oprogramowania? (1)

- **niematerialny produkt**
  - oprogramowanie to „niematerialny wytwór intelektu”; zobaczyć można dopiero (częściowo) działający produkt, wcześniej dostępne jedynie pewne reprezentacje (np. specyfikacja wymagań, projekt GUI, algorytm), z których każda jest mocno zawężona/ograniczona
- **zdominowane przez projektowanie**
  - pomijalne nakłady na produkcję (kopiowanie), większość wysiłku skierowana na projektowanie; każdy produkt jest nieco/mocno inny
- **nie zużywa się**
  - nie zużywa się; wadliwe działanie nie wynika ze „zmęczenia materiału”, ale z błędów projektowych (często ujawniających się bardzo późno) lub z niedopasowania do środowiska
- **duża złożoność**
  - większa niż w przypadku budynku czy urządzenia sprzętowego, trudno prześledzić wszystkie możliwości i wypuścić produkt pozbawiony błędów

# Specyfika oprogramowania (2)

- **dowolność struktury**
  - brak ograniczeń typowych dla świata fizycznego (np. prawa fizyki, konieczność zmieszczenia produktu w obudowie); duża dowolność projektantów w kształtowaniu systemu; w efekcie jednak ryzyko trudności w modyfikacji i utrzymaniu systemu
- **zależności pomiędzy elementami**
  - niejawne zależności pomiędzy elementami (współdzielone dane, pamięć, przestrzeń nazw itp.), co prowadzi do trudności konstrukcyjnych, błędów (często trudnych do zdiagnozowania), problemów z wprowadzaniem modyfikacji
- **działanie cyfrowe, nie analogowe**
  - inaczej niż dla systemów analogowych (dominujących w świecie fizycznym), gdzie mała zmiana na wejściu na ogólny oznacza małą zmianę wyniku, tutaj nieznaczna zmiana (nawet pojedynczego bitu) może spowodować kompletnie inny rezultat
- **łatwość zmian**
  - cecha bardzo pożądana, jednak może prowadzić do wprowadzania zmian w sposób nieprzemyślany, błędów i bylejakości

## Specyfika oprogramowania (3)

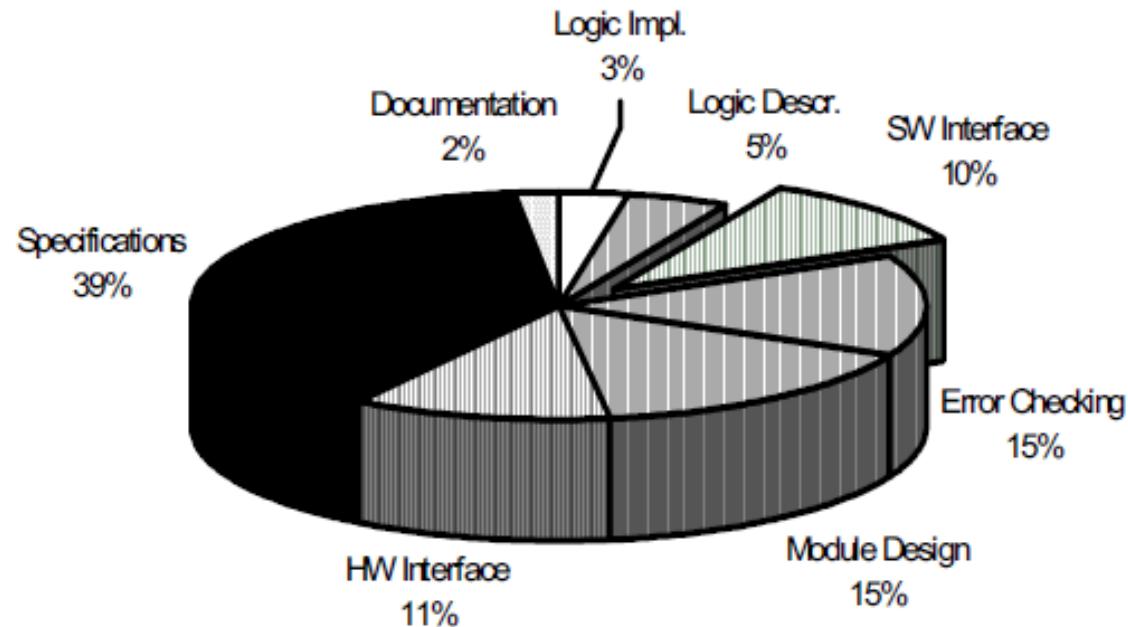
- **Pytanie** - Co zrobić kiedy przedsięwzięcie jest opóźnione i widać, że nie „wyrabiamy się” na ustalony termin?
- **Odpowiedź („zdroworozsądkowa”)** – Zaangażować większą liczbę ludzi!
- **Brutalna rzeczywistość** – w projekcie informatycznym prawdopodobnie jeszcze pogorszy to sytuację ☹

# Specyfika oprogramowania (4)

- Pytanie – Jak oszacować postęp prac?
- Odpowiedź („zdroworozsądkowa”) – Zapytać wykonawcę o stopień zaawansowania!
- Brutalna rzeczywistość – typowy projekt informatyczny szacowany w ten sposób jest ukończony w 90% przez 90% czasu jego trwania

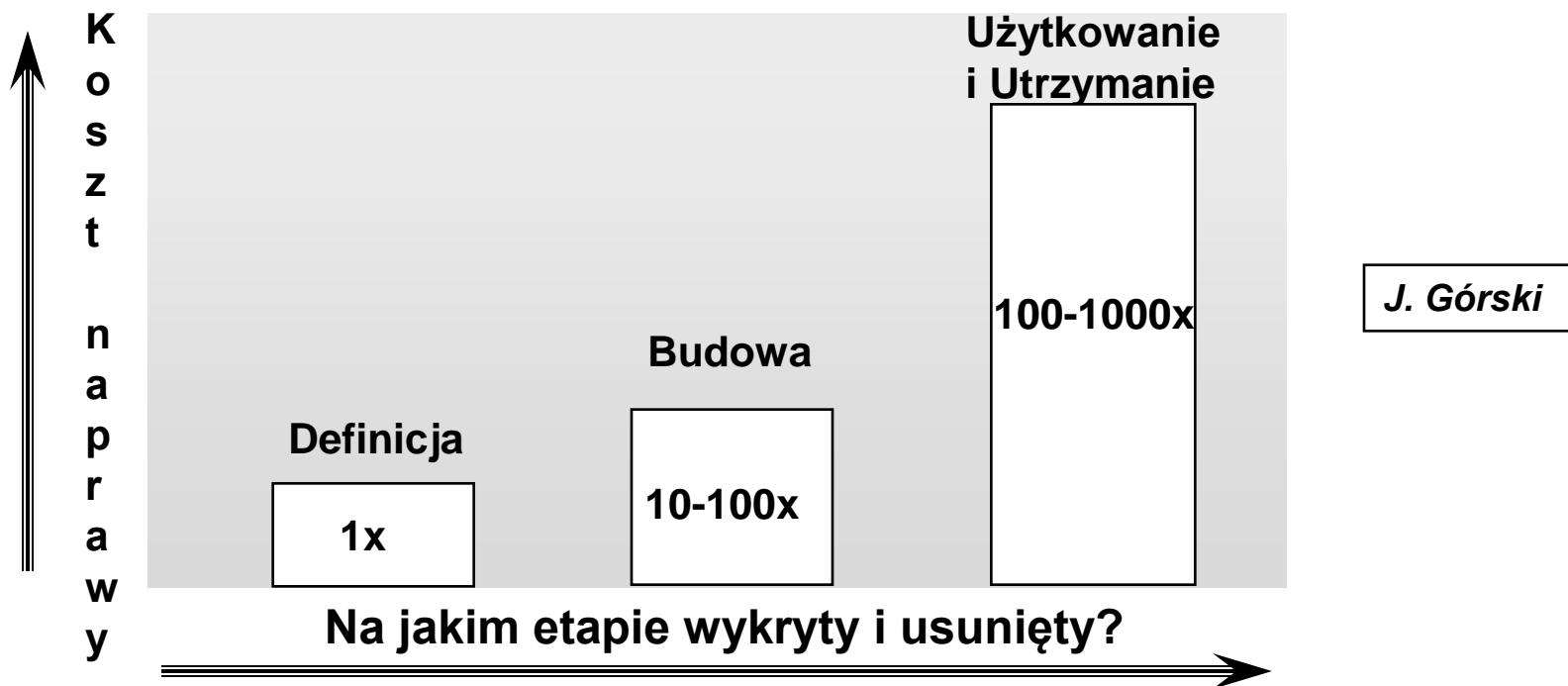
# Specyfika – gdzie popełniane są błędy?

Błędy popełniane podczas implementacji (kodowania)  
wcale nie muszą być największym problemem!



van Moll J., Jacobs J., Freimut B., Trienekens J., *The importance of life cycle modeling to defect detection and prevention, Proceedings of the 10th International Workshop on Software Technology and Engineering Practice*, 2002, pp. 144-156.

# Koszt błędów w czasie (reguła 1:10)

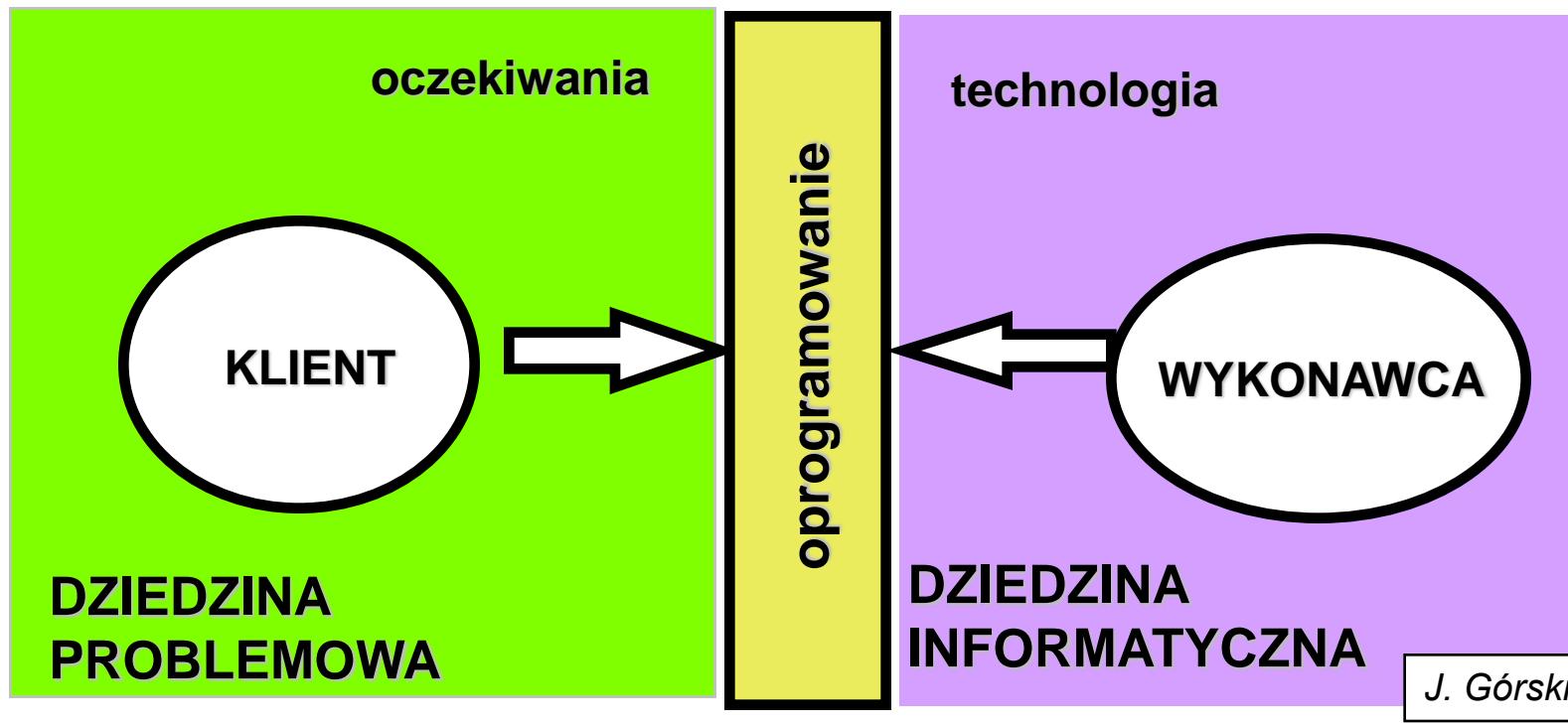


Analogia do innych dziedzin inżynierskich...

**WNIOSEK:** lekceważenie wczesnych etapów pracy to ryzyko dodatkowych, nadmiarowych nakładów!

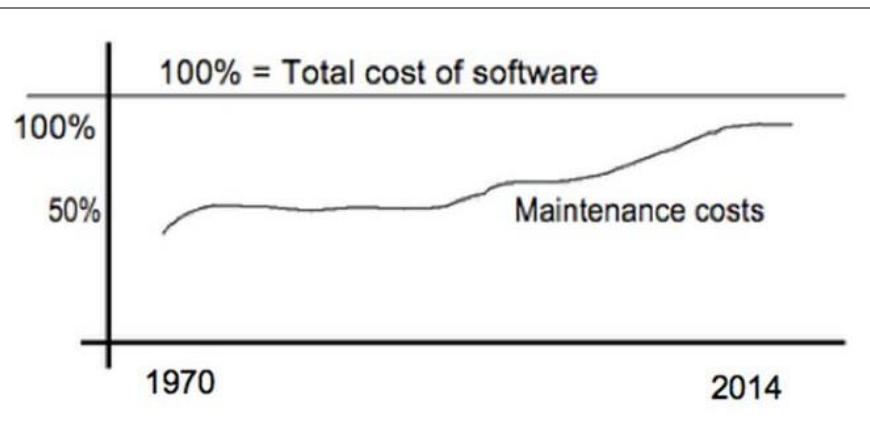
# Oprogramowanie wartościowe dla klienta

*Zdecydowana większość faktycznie przydatnego oprogramowania powstaje w układzie klient-wykonawca*



J. Górski

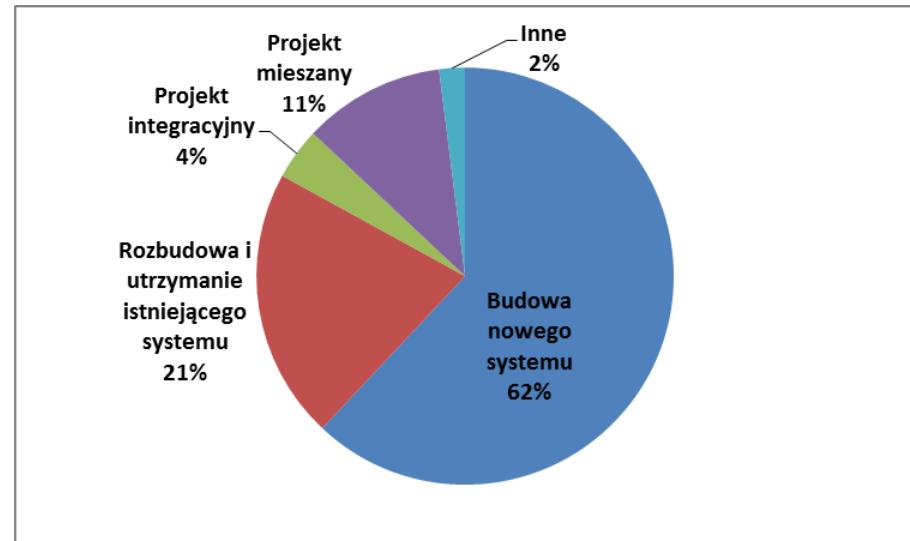
# Wytwarzanie a utrzymanie systemów



C. Burki, H. Vogt, How to save on software maintenance costs, 2014

J. Koskinen, Software Maintenance Costs, 2015

K. Frączkowski,  
Krytyczne czynniki sukcesu  
i porażki projektów IT w Polsce.  
Raport z badań 2016–2021



# Realia ?

**Możliwa (i niestety dość prawdopodobna)  
porażka przedsięwzięcia informatycznego...**

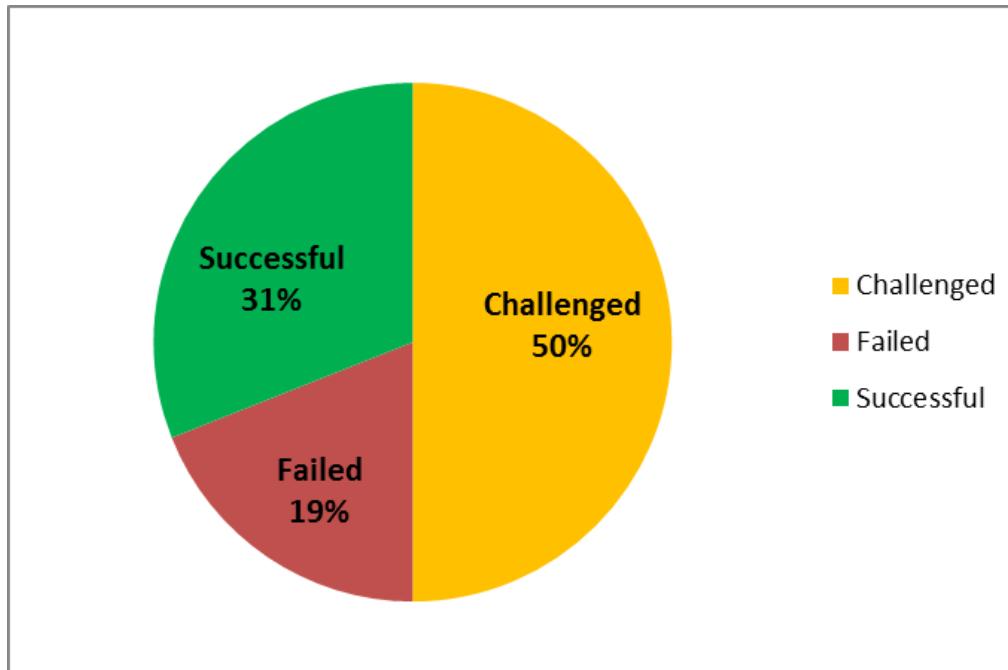
**... lub problemy w przynajmniej niektórych  
poniższych kwestiach:**

- funkcjonalność – oprogramowanie nie pozwala klientowi na realizację niezbędnych/pożądanych usług
- przekroczone koszty
- przekroczony harmonogram
- problemy z jakością (niezawodność, łatwość użycia, wydajność, ...)



# Wyniki przedsięwzięć informatycznych (1)

„Chaos Report” grupy Standish, rok 2020



1994: 16S 53C 31F

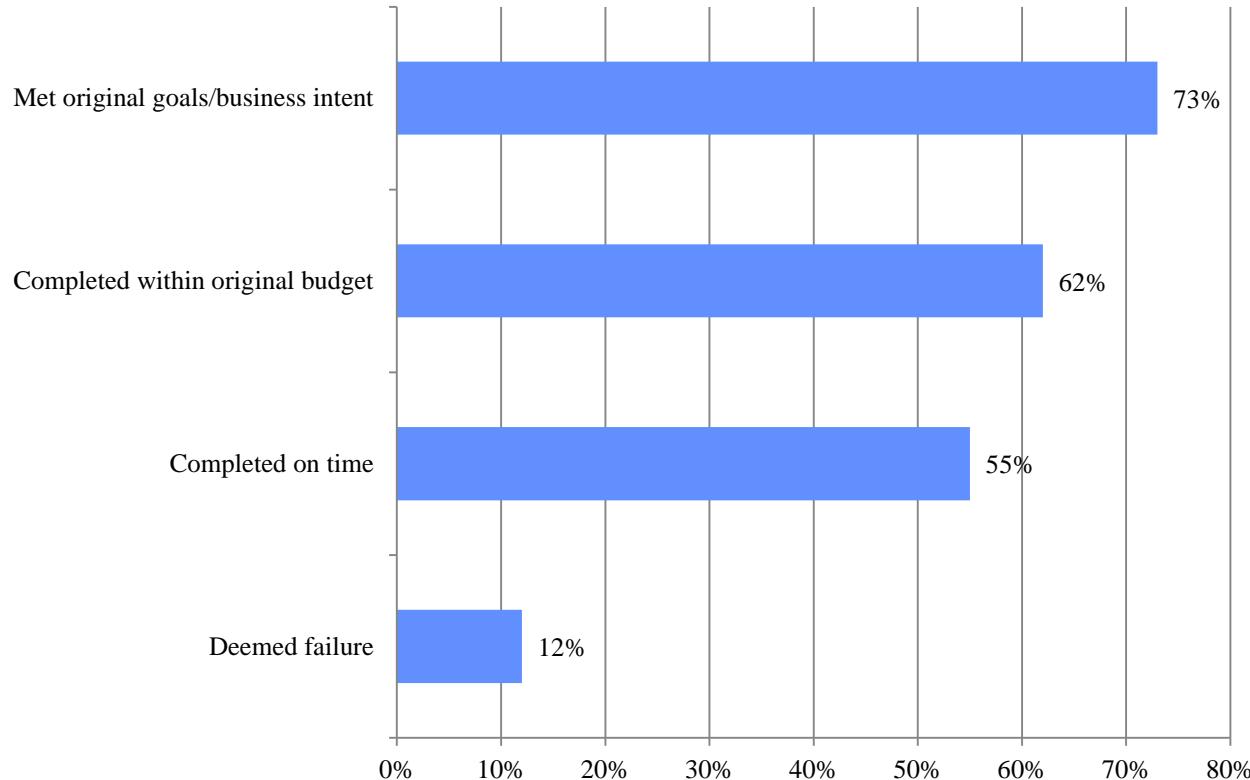
2006: 35S 45C 19F

2009: 32S 44C 24F

2012: 39S 43C 18F

2015: 29S 52C 19F

## Wyniki przedsięwzięć informatycznych (2)



**Project Management Institute – Pulse of the Profession 2021**

# Problemy przedsięwzięć informatycznych

## Przyczyny porażek projektów:

- Cele nierealistyczne lub w ogóle nie sformułowane
- Błędne oszacowanie potrzebnych zasobów
- Źle zdefiniowane wymagania
- Brak raportowania stanu projektu
- Brak zarządzania ryzykiem
- Słaba komunikacja
- Wykorzystanie niedojrzalej technologii
- Problemy z opanowaniem złożoności projektu
- Naganne praktyki deweloperskie
- Słabe zarządzanie projektem

Źródło: Charette R., *Why software fails*,  
IEEE Spectrum, Sept 2005

USA

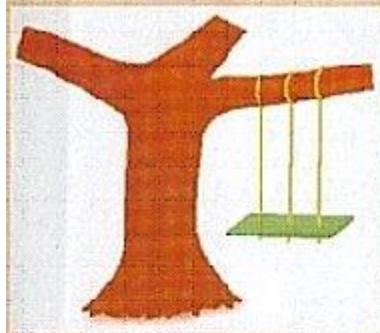
Reason for Failure	All
Lack of executive support	17%
Incomplete requirements	13%
Expectations not set / managed	12%
Scope creep	12%
Project / Organisation not aligned	10%
Lack of resources	7%
Technology Issues	7%
Other	6%
Lack of user involvement	6%
Poor Planning	5%
Inexperienced PPM People	4%

Źródło: Arras People Group,  
2010 Project Management  
Benchmark Report

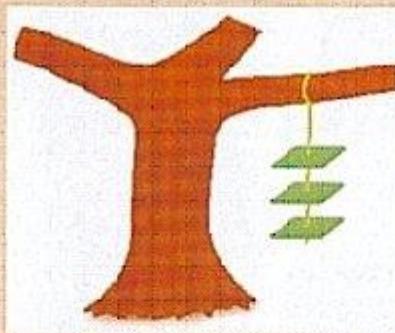
UK

# Przykładowy projekt...

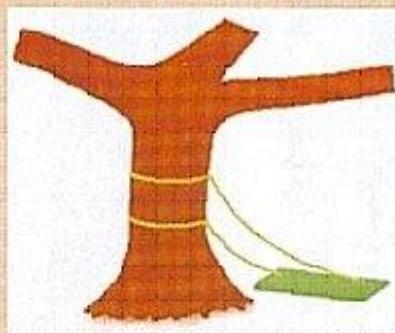
## **Etapy budowy systemu informatycznego dla przedsiębiorstwa**



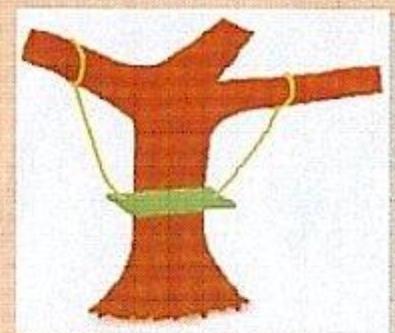
**1** To, co klient zamówił.



**2** To, co analityk zrozumiał.



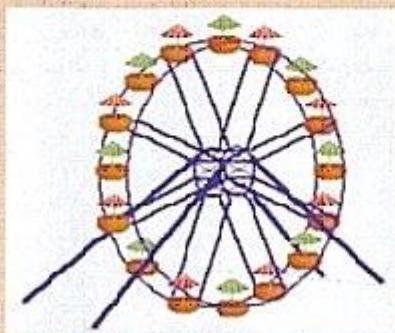
**3** To, co opisywał projekt.



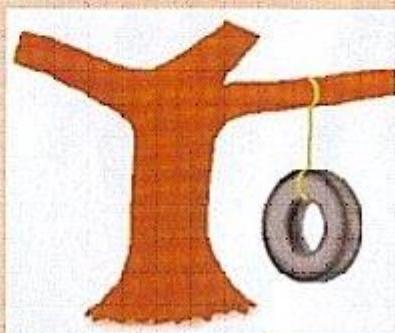
**4** To, co wykonali programiści.



**5** Projekt po uruchomieniu i wdrożeniu.



**6** To, za co klient zapłacił.



**7** A to, czego klient potrzebował.



**8** Praktyczne zastosowanie projektu.

# Czynniki sukcesu

## Wpływ wybranych czynników na sukces projektu

jedna odpowiedź	Zdecydowanie NIE	Raczej NIE	Nie mam zdania	Raczej TAK	Zdecydowanie TAK	Odpowiedzi
wsparcie i zaangażowanie sponsora – klienta	18 (13%) -	15 (11%) -	29 (21%) -	42 30%	34 (25%) -	138
dojrzałość emocjonalna, właściwa współpraca	8 (6%) -	15 (11%) -	26 (19%) -	51 37%	38 (28%) -	138
zaangażowanie użytkownika	10 (7%) -	21 (15%) -	18 (13%) -	48 35%	41 (30%) -	138
zoptymalizowany zakres zarządzania projektem	6 (4%) -	27 (20%) -	41 (30%) -	45 33%	19 (14%) -	138
wykwalifikowani pracownicy – rozumienie biznesu i technologii	3 (2%) -	12 (9%) -	18 (13%) -	62 45%	43 (31%) -	138
stosowanie standardów architektury	10 (7%) -	21 (15%) -	25 (18%) -	56 (41%) -	26 (19%) -	138
wykorzystanie procesów Agile	38 (28%) -	15 (11%) -	39 (28%) -	25 (18%) -	21 (15%) -	138
umiejętnne wykorzystanie narzędzi zarządzania projektami	13 (9%) -	20 (14%) -	34 (25%) -	51 (37%) -	20 (14%) -	138
wykorzystanie wiedzy opartej na metodykach zarządzania projektami	10 (7%) -	17 (12%) -	35 (25%) -	56 (41%) -	20 (14%) -	138
jasne zdefiniowanie i rozumienie celów biznesowych	8 (6%) -	8 (6%) -	20 (14%) -	55 (40%) -	47 (34%) -	138

# Geneza problemu

- **Zamierzchłe czasy – II połowa lat 60**
- **Rozwój sprzętu i języków programowania**
- **Oprogramowanie użytkowe, interaktywne, wspomagające procesy (zamiast czystych obliczeń, przetwarzania wsadowego)**
- **Wzrost rozmiaru i złożoności tworzonego oprogramowania**
- **Obserwacja:** Rozwój metod tworzenia oprogramowania nie nadąża za rozwojem technologicznym
- **Początki dziedziny Software Engineering**



# A kilkadziesiąt lat później...

- Rewolucja informacyjna - zastosowania informatyki
  - transport, produkcja, obsługa pieniądza, ochrona zdrowia, administracja, rozrywka, telekomunikacja, ...
- Oczekiwania (głównie jakościowe)
  - wysoce niezawodne, bezpieczne
  - użyteczne, łatwe w użyciu, trudne do użycia w sposób niewłaściwy
  - tanie, dostępne, ...
- Ciągły dynamiczny rozwój technologii
- Dynamika zmian (biznes, usługi, oferta) -> odzwierciedlenie w systemach inf.



# Wnioski

- Analogia pokazuje, że w przypadku **przedsięwzięć inżynierskich** większej skali potrzebny jest **szereg metodycznych działań** umożliwiających ich skutecną realizację
- Istnieje jednak dodatkowa **specyfika oprogramowania** i co za tym idzie projektów informatycznych, która generuje **dodatkowe problemy** i wymaga znacznie bardziej (w stosunku do innych obszarów inżynierskich) **zaawansowanych metod** radzenia sobie z tymi problemami.

# Potrzeby – wymagania dot. kształcenia

- Postulowane programy nauczania (curricula) – ACM, IEEE



- Programy nauczania IO:
  - SWEBOK (Software Engineering Body of Knowledge) v.3 (2013)
  - GSERC / GSE2009 (Graduate Software Engineering)
  - Polskie normy MEiN



# Perspektywy kariery



- **Niekoniecznie tylko ścieżka technologiczna typu junior/regular/senior developer**
- **Różne inne role/stanowiska wymagające również pewnych innych umiejętności czy kompetencji (w tym miękkich)**
  - Project Manager
  - Business/System/IT Analyst
  - Enterprise/System Architect
  - UX Designer
  - Product Manager
  - Scrum Master
  - Database Administrator
  - Business Intelligence Developer
  - Security Specialist
  - ...

# Wykład

## Cele wykładu:

- **Przekazanie wiedzy nt. budowy rzeczywistych systemów informatycznych i metodycznego wytwarzania oprogramowania**
- **Przybliżenie obszarów praktyki wytwarzania oprogramowania innych niż implementacja (kodowanie)**
- **Nauka tworzenia abstrakcyjnych modeli systemów, w szczególności modeli tworzonych za pomocą obiektowej notacji UML (*Unified Modeling Language*)**

# Laboratorium

## Cele laboratorium:

- **Nabycie umiejętności analizy postawionego problemu i udokumentowania rezultatu**
- **Praktyczna nauka posługiwania się językiem modelowania UML**
- **Zapoznanie z narzędziami CASE**

## Zadania:

- **Przygotowanie założeń wstępnych budowy systemu**
- **Modelowanie za pomocą przypadków użycia (use-cases)**
- **Budowa modelu klas**
- **Modelowanie zachowania (dynamiki systemu)**

# Literatura

- Sommerville I., [Software Engineering](#), 10th edition, 2015 (wydanie polskie WNT 2020)
  - Pressman R., Maxim B., [Software Engineering: a Practitioner's Approach](#), McGraw-Hill, 9th edition, 2020 (wydanie polskie WNT 2005)
  - Sacha K., [Inżynieria Oprogramowania](#), PWN, 2010
- 
- Booch G., Rumbaugh J., Jacobsen I.: [UML przewodnik użytkownika](#), WNT, 2012
  - Fowler M., Scott K.: [UML Distilled 3.0](#), Addison-Wesley 2003
  - Maciaszek L.: [Requirements analysis and system design](#), 3rd edition, Addison-Wesley, 2007
  - McLaughlin B., Pollice G., West D., [Head First: Object-Oriented Analysis and Design](#), Edycja polska (Rusz głową!), Helion, 2008

# Dokument „Wizji Systemu”

*Aleksander Jarzębowicz*

*Katedra Inżynierii Oprogramowania  
Wydział ETI, Politechnika Gdańsk*

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich  
rozprowadzanie jest zabronione.

# Analiza biznesowa i systemowa

- Zagadnienia tu omawiane generalnie są pewnym (zawężonym) wycinkiem działań z obszaru **analizy biznesowej i systemowej**
- **Analiza taka ma na celu:**
  - zrozumienie tzw. obszaru problemowego czyli organizacji, gdzie będzie wprowadzany system, jej działalności i procesów
  - identyfikację i wyspecyfikowanie wymagań względem tego systemu
  - często również opisanie obszaru problemowego i wymagań na system za pomocą różnorodnych modeli
- Szersze omówienie na kolejnych wykładach, tutaj tylko to, co niezbędne do zadań na laboratorium

# Dokument założeń wstępnych

- „Wizja Systemu”
- Pierwsza, jeszcze ogólnikowa, próba określenia obszaru problemowego podlegającego informatyzacji oraz zakresu systemu
- Ujęcie częściowo sformalizowane (struktura dokumentu), choć nie tak sformalizowane i kompletne jak np.:
  - Studium Wykonalności (ang. *Feasibility Study*)
  - Specyfikacja Wymagań Systemowych (ang. *System Requirements Specification*)
  - Specyfikacja Istotnych Warunków Zamówienia (SIWZ), przy zamówieniach publicznych i przetargach

# Struktura Wizji Systemu

- 1. Opis organizacji**
- 2. Interesariusze systemu**
- 3. Kontekst systemu – użytkownicy i systemy zewn.**
- 4. Wymagania funkcjonalne**
- 5. Wymagania jakościowe**
- 6. Ograniczenia**

**W poszczególnych punktach również jest narzucona struktura  
(tabele, listy, wypunktowania)**

**To dokument techniczny, nie esej, wpis na blogu czy poemat dygresyjny!**

## Opis organizacji (opis dziedziny problemowej)

- **Opis organizacji, dla którego ma być tworzony system**
  - może to także być wyróżniona część organizacji (dział, pion) lub organizacja plus elementy jej otoczenia (np. klienci, dostawcy)
- **Należy założyć, że chodzi o jedną, konkretną organizację, nie wszystkie zajmujące się daną działalnością**
  - czyli np. system dla Politechniki Gdańskiej, nie system dla wszystkich możliwych uczelni na świecie
- **Ważny jest taki dobór tematu, żeby możliwe było sporządzenie takiego opisu np.**
  - bazując na jakiejś rzeczywistej organizacji
  - mając dostęp do wiedzy dziedzinowej

## Opis organizacji (opis dziedziny problemowej) c.d.

- **Opis zawierający konkretne dane, również liczbowe (np. ilu pracowników), profil działalności, główne procesy (np. sprzedaż, produkcja, pozyskiwanie surowców), powiązania z innymi organizacjami**
- **Perspektywy dalszej działalności, kierunki rozwoju**
- **Struktura organizacyjna, podział odpowiedzialności**
- **Identyfikacja problemów w aktualnym funkcjonowaniu, które można wyeliminować lub ograniczyć dzięki systemowi IT**
- **Ogólna koncepcja planowanego systemu IT**

# Interesariusze i ich punkty widzenia (1)

- **Interesariusz (udziałowiec, ang. stakeholder)**

- **każdy podmiot (niekoniecznie ożywiony), który ma uzasadnione prawo wywarcia wpływu na powstający system**
- **a więc: potencjalne źródło wymagań dotyczących systemu**

## Do interesariuszy zaliczają się:

- Przyszli użytkownicy
- Inni pracownicy organizacji, w której wdrażany jest system (np. kierownicy, którzy nie będą bezpośrednio używać systemu)
- Zewnętrzni partnerzy tej organizacji (np. kontrahenci)
- Klienci tej organizacji
- Zewnętrzne organizacje
- Systemy informatyczne współpracujące z tym powstającym
- Prawo, przepisy branżowe
- Reprezentanci dostawcy systemu (zespół projektowy, kierownictwo projektu)
- Grupy nacisku np. związki zawodowe

# Interesariusze i ich punkty widzenia (2)

## Przykłady:

- Klient banku oczekuje, że wprowadzenie systemu zmniejszy czas obsługi w okienkach oraz umożliwi mu zdalne wykonywanie operacji dotyczących przelewów. Jest silnie zainteresowany kwestiami ochrony w związku z możliwością nieuprawnionego dostępu do jego konta oraz ujawnienia informacji o jego operacjach finansowych.
- Kasjer banku oczekuje, że system ułatwi jego pracę eliminując błędy, za które mógłby być pociągnięty do odpowiedzialności. Obawia się zwolnień, które mogą mieć miejsce jeśli system usprawni obsługę klientów oraz zwiększonej kontroli ze strony kierownika.

## Punkty widzenia są związane:

- **z różnymi sposobami użytkowania systemu**
- **z organizacją (własną i powiązanych podmiotów)**
- **ze współpracującymi systemami**
- **z wiedzą dziedzinową**
- **z regulacjami (prawnymi, normami i standardami itp.)**
- ...

# Kontekst systemu

- **Użytkownicy i ich specyfika**
  - kto będzie korzystał z systemu?
  - w jakich warunkach (niekoniecznie biuro – samochód, hala fabryczna, sklep, ...)?
  - co jest kluczowe w jego/jej pracy z systemem np. szybkość wykonania zadania?
  - jakie ma umiejętności obsługi komputera i systemów określonej klasy?
  - czy będą potrzebne szkolenia, a jeśli tak, to w jakim zakresie?
  - jaka pomoc powinna być zapewniona?
  - czy użytkownicy nie mają jakichś przyzwyczajeń, które warto przenieść na interfejs systemu (np. określone formularze dokumentów)?
  - czy występuje jakaś dodatkowa specyfika wymagająca dostosowania interfejsu np. cudzoziemcy, ludzie starsi, osoby niepełnosprawne?
- **Inne systemy współpracujące**
  - zakres współpracy
  - funkcje udostępniane lub wywoływanie (API)
  - interfejsy między systemami
  - format wymienianych danych
  - współdzielone dane

# Wymagania względem systemu

- **Funkcjonalne**
  - Co system ma robić? Jakie usługi/funkcje będzie udostępniał poszczególnym użytkownikom?
- **Jakościowe (pozafunkcjonalne) – jakie daje gwarancje na jakość tych usług**
  - **wydajność** – jakie są wymagania odnośnie liczby użytkowników, wolumenów danych, czasu reakcji oraz jak te parametry mogą się zmieniać w dłuższym horyzoncie czasowym
  - **niezawodność** – jak ważne jest, aby w działaniu systemu nie wystąpiły błędy ((przez określony przedział czasu lub dla określonej liczby wywołań usług)
  - **dostępność** – ile czasu i ew. w jakich porach system może nie działać/ być wyłączony
  - **ochrona (security)** – na ile system musi być odporny na ataki z zewnątrz
  - **bezpieczeństwo (safety)** – na ile system musi dawać gwarancje, że nie zagrozi swojemu otoczeniu (ważne zwłaszcza dla systemów krytycznych)
  - **przenośność** – na ile wymagane jest aby system działał na różnych urządzeniach, platformach, OSach, przeglądarkach itp.
  - **elastyczność** – na ile system powinien być możliwy do łatwej dalszej rozbudowy
  - **konfigurowalność** – czy dla systemu można zidentyfikować kluczowe parametry, które mogą się zmieniać i trzeba umożliwić te zmianę zamiast ich „twardego wkodowania” (np. stawki podatków)
- **Ograniczenia – co jest narzucane wykonawcy**
  - np. czas, budżet, określone technologie, wymagana dokumentacja, sposób wdrożenia

# Formułowanie wymagań

- **Funkcjonalne** – wystarczy ogólnie wymienić poszczególne funkcje dostępne dla poszczególnych użytkowników np.

**Lekarz:**

- Odczyt pełnej zawartości karty pacjenta
- Przesłanie zlecenia badania do laboratorium
- Potwierdzenie przyjęcia pacjenta na oddział
- Zapis zlecenia podania leków
- ...

- **Jakościowe (pozafunkcjonalne)** – wymagania formułowane w sposób konkretny, mierzalny, ale bez narzucania określonych rozwiązań np.

**Wydajność** – System ma pozwalać na jednoczesną pracę 2000 użytkowników bezauważalnych opóźnień (czas reakcji nie przekraczający 1 sek.). Ma być przygotowany na obsługę wolumenów danych rzędu 10 mln rekordów, z przewidywanym przyrostem o 1 mln rekordów rocznie.

- **Ograniczenia**

Dot. projektu i zasobów np.

- Pełny system musi być dostarczony i wdrożony do dnia 31.10.2023.

Dot. produktu np.

- W celu spójności z polityką technologiczną klienta, warstwa danych systemu musi być oparta o rozwiązania MS SQL (**ograniczenie, nie decyzja o rozwiązaniu projektowym!**)

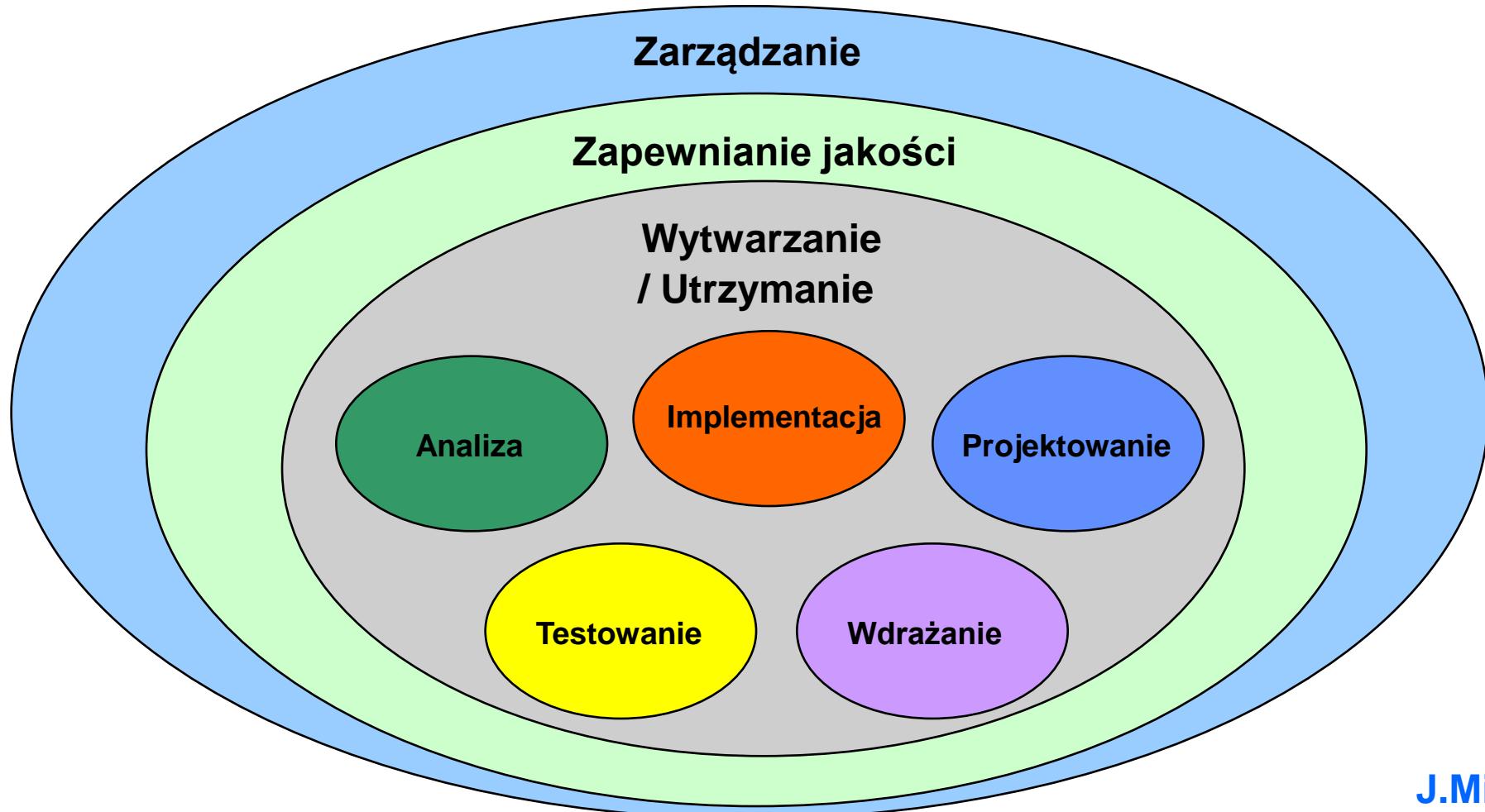
# Obszary działań w inżynierii oprogramowania

*Aleksander Jarzębowicz*

Katedra Inżynierii Oprogramowania  
Wydział ETI, Politechnika Gdańskia

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich  
rozprowadzanie jest zabronione.

# Główne obszary IO



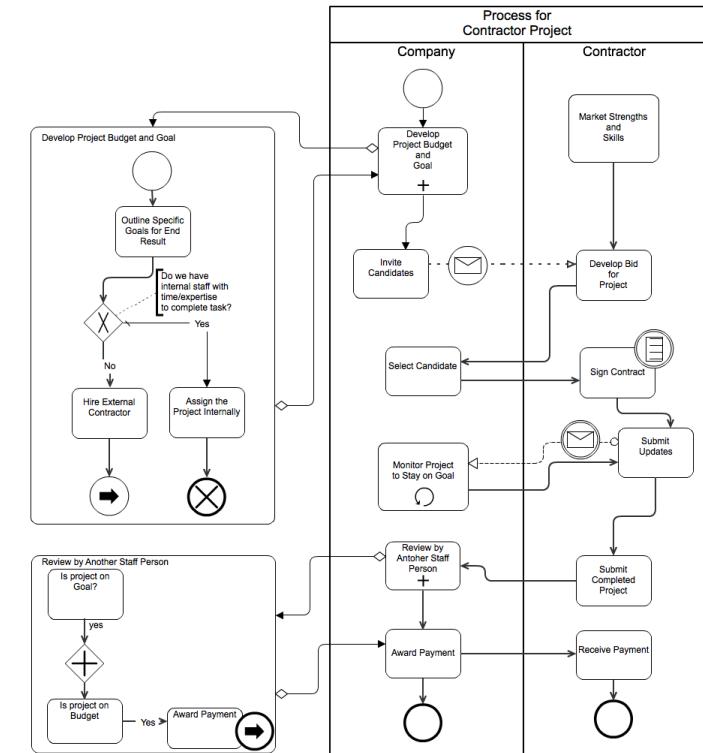
# Analiza (*odp. na pytanie: co?*)

- **Analiza biznesowa**

- **Identyfikacja i zrozumienie obszaru problemowego (najczęściej w ramach informatyzowanej organizacji)**
- **Identyfikacja celów biznesowych i związanych z nimi strategii**
- **Identyfikacja i ewentualne przekształcenia procesów biznesowych**

- **Analiza systemowa**

- **Pozyskanie i określenie wymagań względem systemu (inżynieria wymagań)**
- **Reprezentacja wymagań np. w postaci abstrakcyjnych modeli**
- **Zarządzanie tymi wymaganiami, uwzględnianie zmian i wpływu na dalsze prace**



**Granice 2 powyższych dziedzin przenikają się!**

## Analiza biznesowa przed projektem

- Część analizy biznesowej może nie tyle być częścią projektu, co poprzedzać go, a wręcz warunkować jego rozpoczęcie
- Analiza biznesowa może też być prowadzona na poziomie strategii rozwoju całej organizacji i „uruchamiać” wiele projektów
- Rezultaty takich działań „przedprojektowych” mogą być różne np.
  - uruchomienie jednego projektu,
  - uruchomienie wielu projektów (program/portfolio),
  - wybór „najlepszych” projektów do realizacji spośród większej liczby zgłoszonych propozycji,
  - odmowa realizacji projektu (bo np. „nie rokuje”)
- W ogólności analiza biznesowa może skutkować podjęciem innych działań niż realizacja projektu IT
  - zamiast tego np. same zmiany organizacyjne, szkolenia, rebranding, ...

# Analiza systemowa

*(odpowiedź na pytanie: co?)*

- **Identyfikacja i specyfikacja wymagań względem systemu**
- **Specyfikacja wymagań dotyczących oprogramowania np. konstrukcja modelu logicznego**

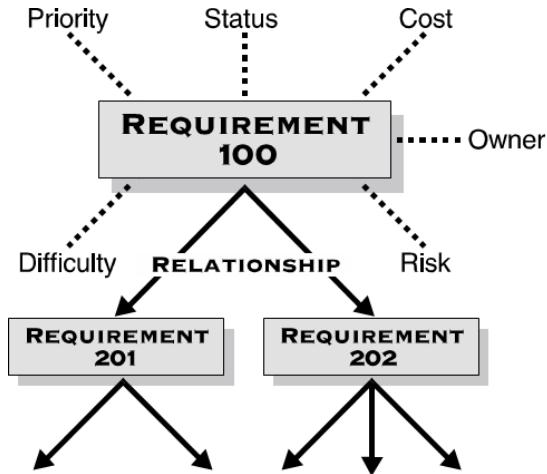
**Wymaganie to:**

1. **Warunek lub zdolność jaką musi spełniać system aby zostać zaakceptowanym przez klienta (na podstawie kontraktu, specyfikacji, umowy itp.)**
2. **Warunek lub zdolność systemu potrzebna interesariuszowi do rozwiązania problemu lub osiągnięcia celu.**
3. **Udokumentowana reprezentacja warunku/zdolności z (1) lub (2)**

**Przypomnienie:** **interesariusz** (inaczej **udziałowiec**, ang. *stakeholder*) to każdy podmiot (niekoniecznie ożywiony), który ma uzasadnione prawo wywarcia wpływu na powstający system, a więc: potencjalne źródło wymagań dotyczących systemu

# Analiza systemowa - inżynieria wymagań

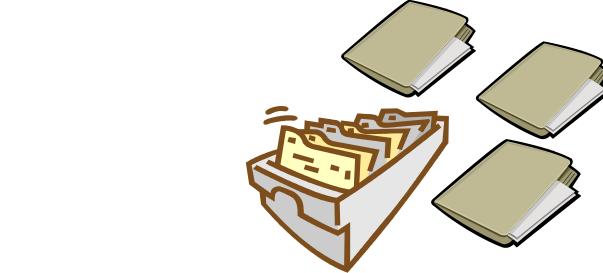
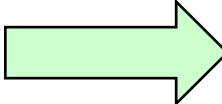
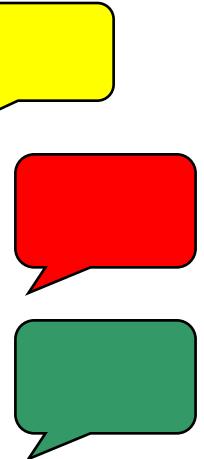
- **Główna część analizy systemowej**
- **Podstawowe obszary działania:**
  - wydobywanie wymagań
  - analizowanie wymagań
  - specyfikowanie wymagań
  - walidacja wymagań
  - zarządzanie wymaganiami
- Termin „inżynieria” sugeruje, że działania te podlegają planowaniu oraz bazują na technikach systematycznych i powtarzalnych



# Analiza systemowa - modelowanie



Wymagania, potrzeby – np. informacje ustne od reprezentantów klienta i innych interesariuszy



Tekstowy zapis informacji o potrzebach i wymaganiach



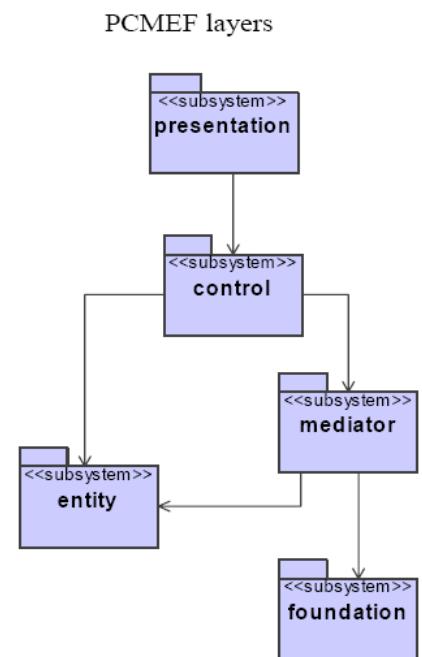
```
private static int insert_table(Connection con, String eName, String eDescr, int eType, String cName1,  
try {  
    CallableStatement cs = con.prepareCall("{ call sp_HAZOPTableNew (?, ?, ?, ?, ?, ?, ?, ?)}");  
    cs.registerOutParameter(7, Types.INTEGER);  
    cs.setString(1,eName);  
    cs.setString(2,eDescr);  
    cs.setInt(3,eType);  
    cs.setString(4,cName1);  
    cs.setString(5,cName2);  
    cs.setInt(6,diagramID);  
  
    cs.executeUpdate();  
    int tableID = cs.getInt(7);  
    return(tableID);  
} catch (Exception e) {  
    System.err.println("A problem with storing table data has occurred");  
    System.err.println(e.getMessage());  
    return (-1);  
}
```

Nie jest to jedyny sposób specyfikowania wymagań (więcej na wykładach poświęconych analizie)

# Projektowanie

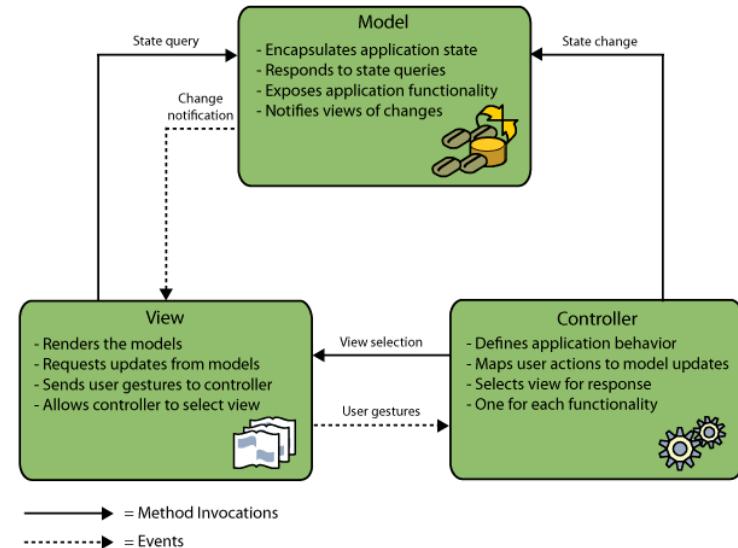
*(odpowiedź na pytanie: jak?)*

- Projekt – ang. design
- Projekt systemu (inne nazwy: p. architektury, p. wysokiego poziomu)
  - Opisuje system w kategoriach jego części składowych (modułów)
- Projekt klas (inne nazwy: p. szczegółowy, p. niskiego poziomu)
  - Opisuje wewnętrzne zależności i funkcjonowanie poszczególnych części składowych (modułów)

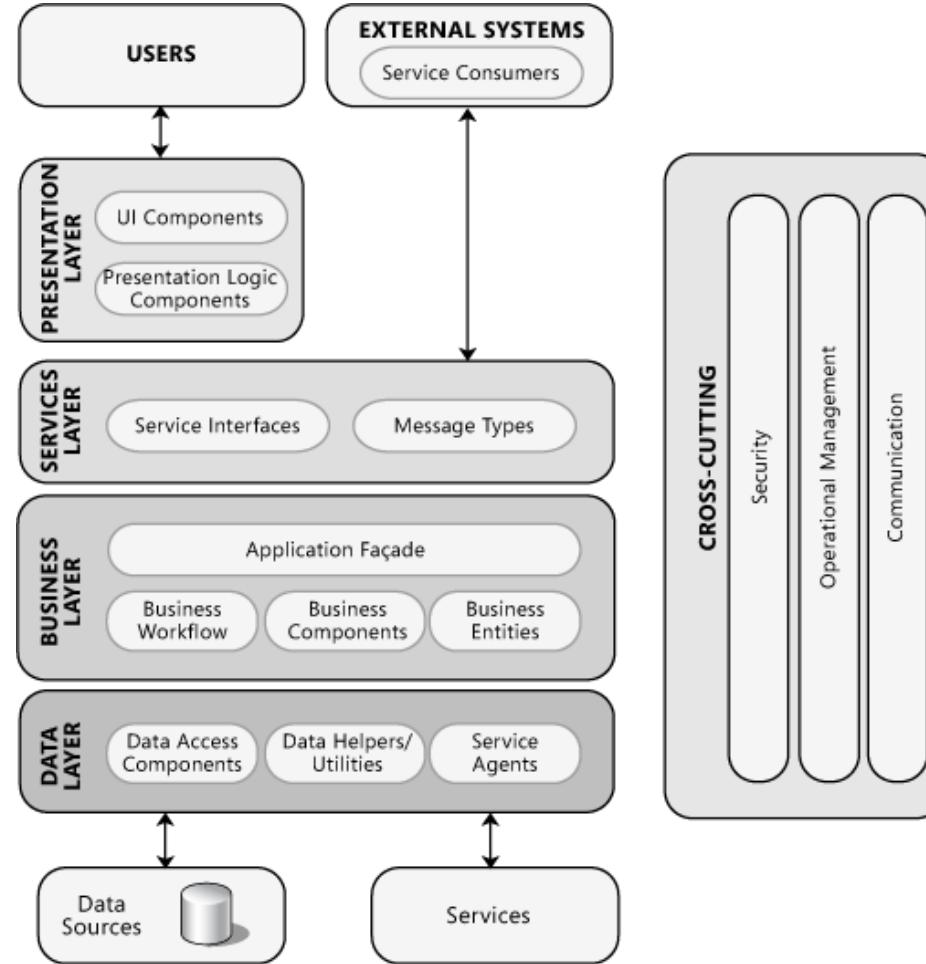


# Projektowanie – wybrane zagadnienia

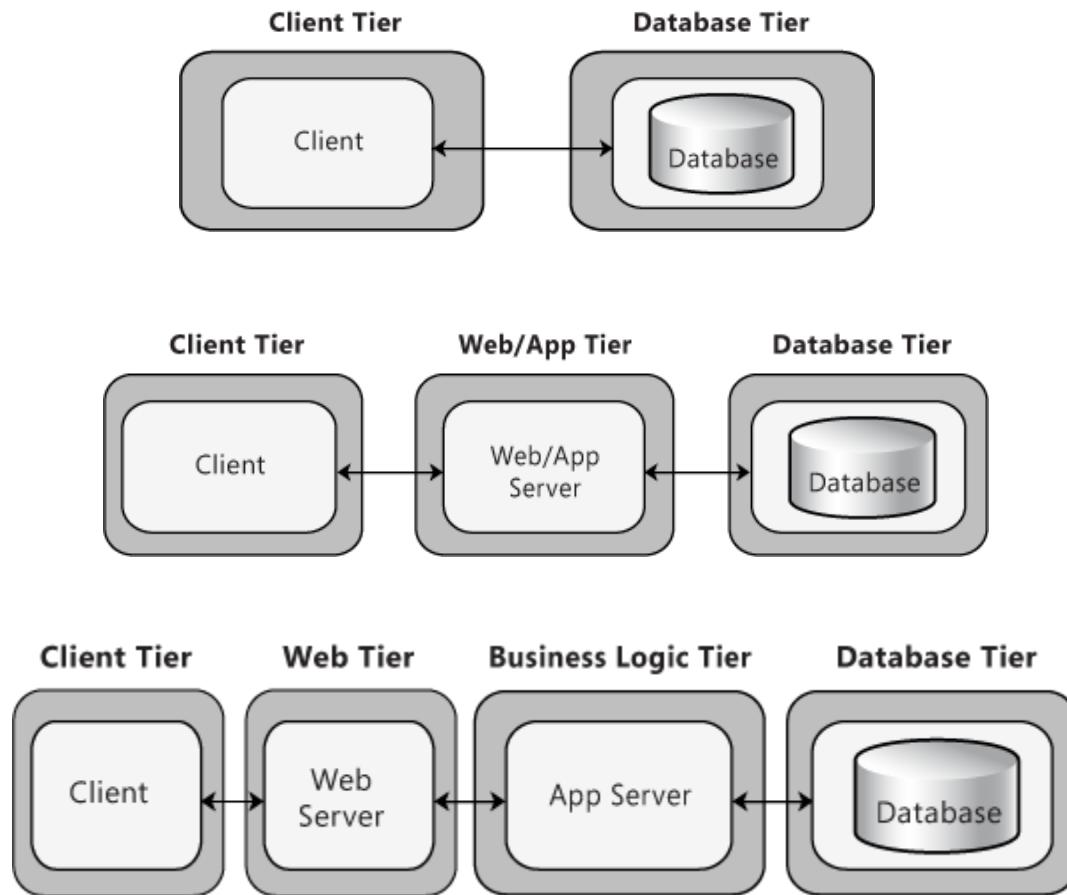
- **Definicja architektury**
  - architektura logiczna - podział na warstwy i partie
  - architektura fizyczna
  - interfejsy komunikacyjne
  - reuse (frameworki, biblioteki, komponenty)
- **Wzorce projektowe**
  - ang. *design patterns*
  - powtarzalne „dobre rozwiązania”
- **Dobre praktyki projektowania**
  - np. SOLID principles, podstawy obiektowości
  - metryki np. high cohesion, low coupling
- **Decyzje co do priorytetów**
  - wynikające z wymagań
  - np. wydajność vs. przenośność, prostota kodu vs. optymalizacja algorytmów



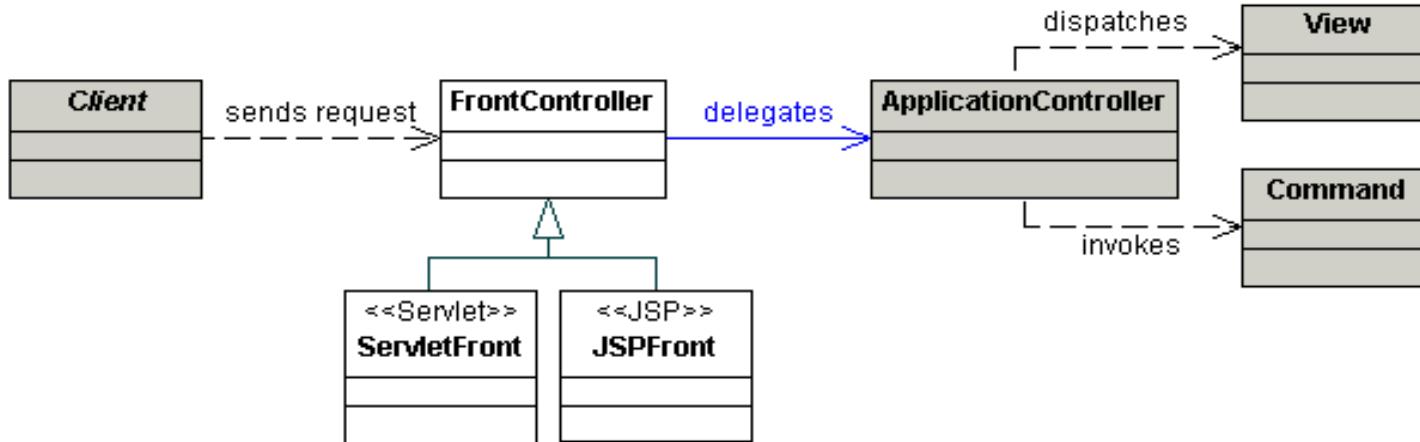
# Architektura logiczna



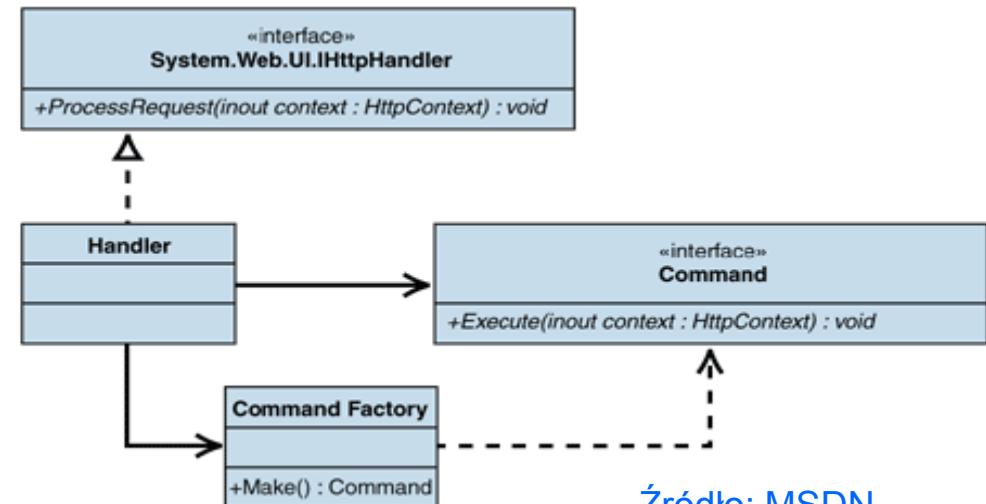
# Architektura fizyczna



# Wzorce projektowe

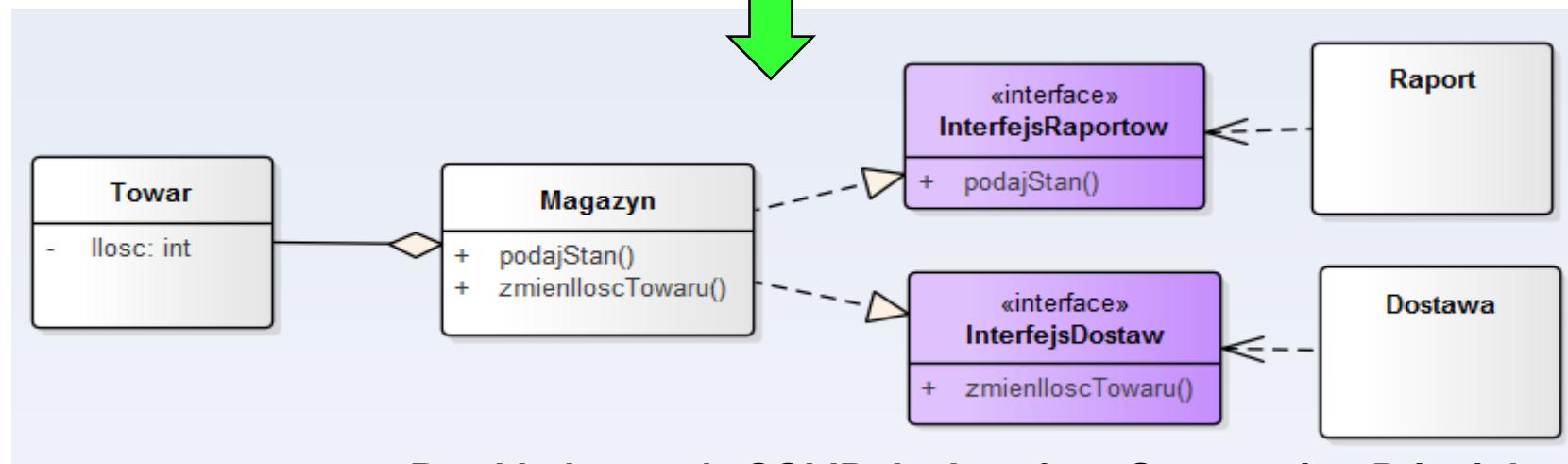
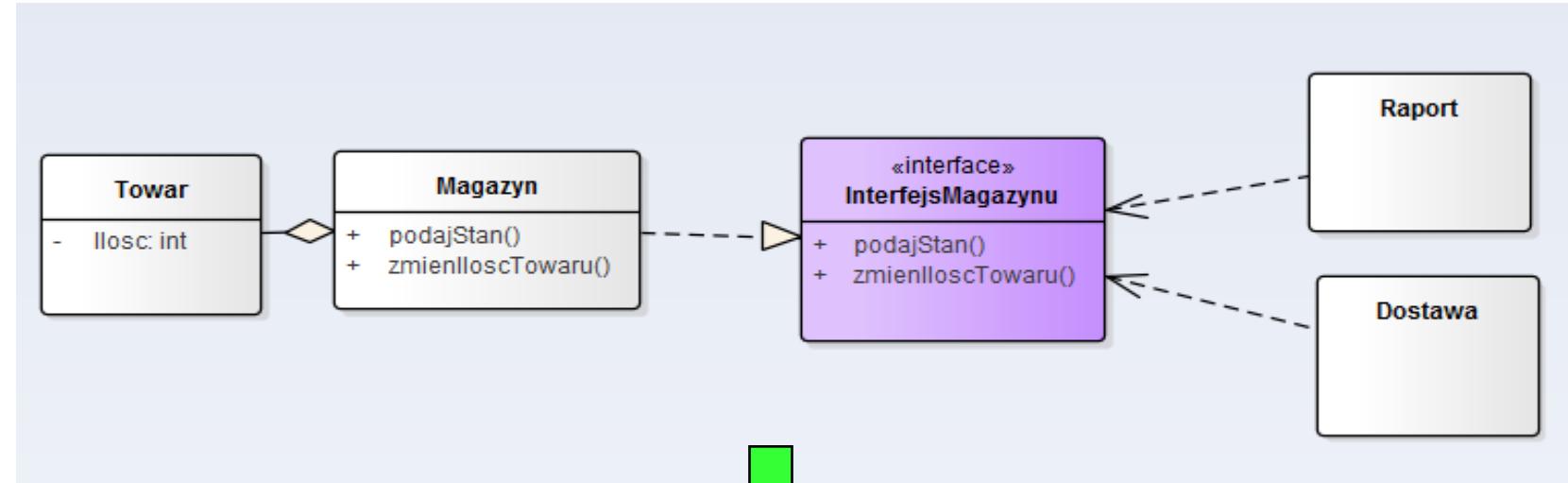


Źródło: Core J2EE Patterns Catalogue



Źródło: MSDN

# Dobre praktyki projektowania



Przykład: zasady SOLID, I – Interface Segregation Principle

# Implementacja

- Wykorzystanie technologii
- Programowanie (kodowanie)
- Często również od razu testy jednostkowe (klas, funkcji)
- Dobre praktyki



# Testowanie

***Testowanie – analiza zachowania oprogramowania w celu oceny jego jakości***

- Czyli - testowanie to eksperymentowanie z działającym kodem
  - Bo możliwe są techniki niewymagające uruchamiania kodu (analiza statyczna, przeglądy i inspekcje)
- Podstawowym celem jest wykrycie defektów



## Pojęcia:

- przypadek testowy
- scenariusz testowy
- skrypt testowy

## Strategie testowania:

- black box
- white box
- mixed

## Poziomy testowania:

- jednostkowe
- integracyjne
- systemowe
- akceptacyjne

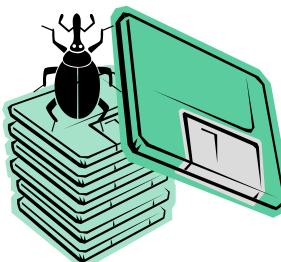
# Kluczowe pojęcia i zależności



pomyłka  
człowieka

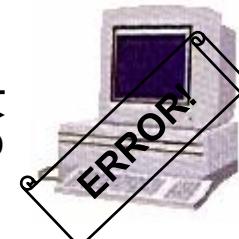


może pro-  
wadzić do



defekt (bug)

może pro-  
wadzić do



błąd

może pro-  
wadzić do



awaria

usterka (defekt) – niepoprawny krok, proces lub definicja danych w oprogramowaniu

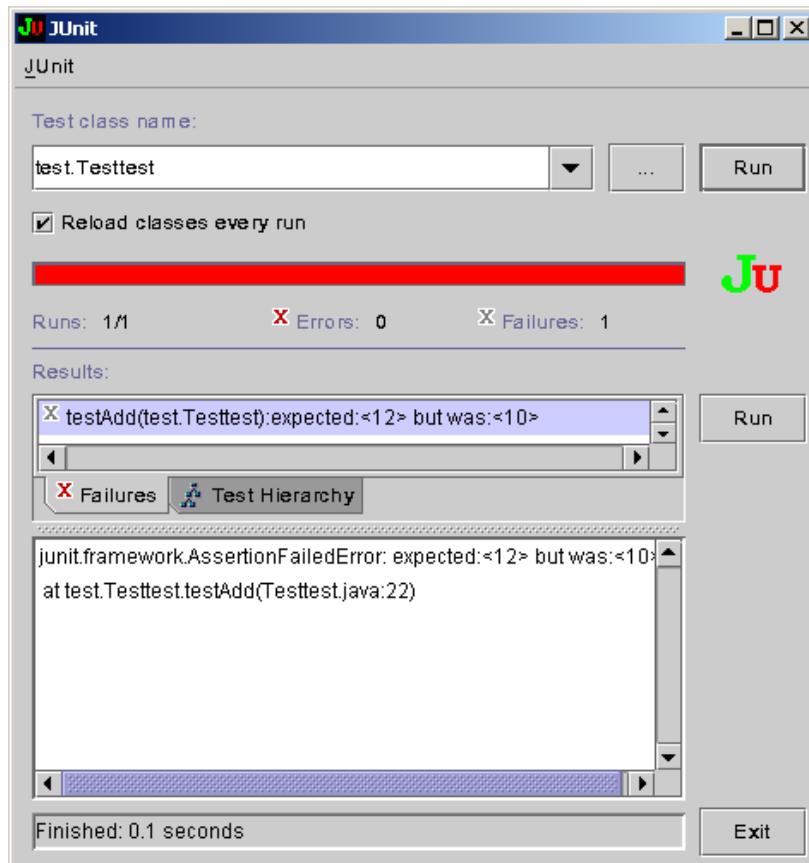
błąd – niepoprawny stan lub zachowanie oprogramowania (ale mogą nie być widoczne)

awaria – niemożność skorzystania z funkcji oprogramowania (widoczna dla użytkownika)

*Jak za pomocą testowania udowodnić, że system jest bezbłędny, nie zawiera żadnych defektów?*

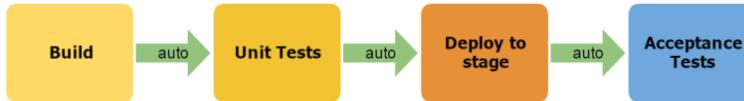
*Testowanie nie jest w stanie wykazać poprawności działania systemu - może tylko pokazać jego błędy, a więc obecność w nim defektów ☹*

# Automatyzacja testów



# CI/CD, DevOps

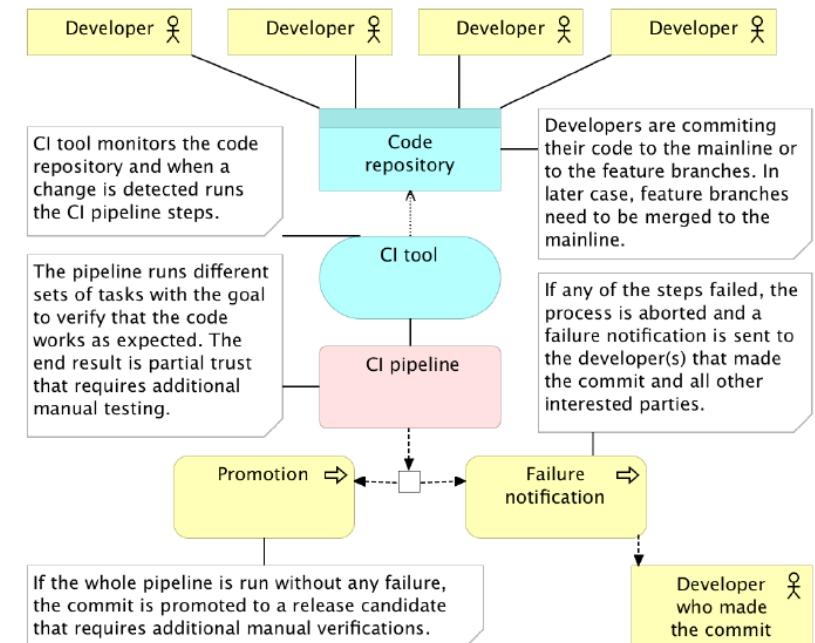
## Continuous Integration



## Continuous Delivery



## Continuous Deployment



# Wdrażanie i utrzymanie

**Wdrożenie – przekazanie systemu i doprowadzenie go do użytkowania w ramach realizacji celu biznesowego**



**Utrzymanie - wprowadzanie zmian w systemie po jego dostarczeniu i wdrożeniu**



# Wdrażanie



- **Wytwarzanie** – prace projektowe nad budową systemu
- **Odbiór** – akceptacja systemu przez klienta
- **Wdrażanie** – działania w celu umożliwienia użytkowania
- **Start produktywny** – moment, w którym można przejść do użytkowania
- **Użycikowanie** – wykorzystywanie systemu w procesach biznesowych klienta

# Wdrażanie - elementy

- Zaplanowanie / przygotowanie wdrożenia
- Instalacja systemu
  - być może również sprzętu / elementów infrastruktury
- Konfiguracja
- Kustomizacja (dostosowanie ogólniejszego produktu do konkretniejszych potrzeb organizacji)
- Szkolenia
  - użytkowników
  - administratorów
  - możliwe, że na początku samych wdrożeniowców
- Migracja / wprowadzenie danych
- Zmiany w organizacji klienta (!!?)
- Audyt / certyfikacja / formalne zatwierdzenie
- Gotowość do startu produktywnego
  - być może również wygaszenie starego systemu



Poprawki defektów  
Realizacja wymagań  
Dopasowania do środowiska  
Optymalizacje  
Testy!

# Utrzymanie oprogramowania

**Utrzymanie - wprowadzanie zmian w oprogramowaniu po jego dostarczeniu i wdrożeniu**

- Poprawianie błędów [*corrective maintenance*]
- Nowa funkcjonalność, rozbudowa, udoskonalanie [*perfective maintenance*]
- Adaptacja do zmian środowiska (serwer aplikacji, DBMS, przeglądarki etc.) [*adaptive maintenance*]
- Utrzymanie zapobiegawcze ukierunkowane na dalszy rozwój [*preventive maintenance*]

# Utrzymanie - uaktualnianie oprogramowania

- **Rodzaje uaktualnień oprogramowania**
  - Wersja (*version*) – nowy produkt w znacznym stopniu zmieniony
  - Wydania (*revision*) – nowa wersja oprogramowania zawierająca nowe funkcjonalności likwidującą szereg niedogodności i błędów
  - Poprawki (*hotfixes, patches*) – uaktualnienia poprawiające konkretne błędy
  - Uaktualnienia zbiorcze (*service packs*) – uaktualnienia zawierające większą liczbę poprawek
- **Problemy związane z uaktualnianiem oprogramowania**
  - Zgodność z danymi przechowywanymi w systemie, ewentualnie konwersja danych
  - Zgodność uaktualnienia z istniejącymi wersjami oprogramowania
  - Zapewnienie ciągłości pracy systemu
  - Dystrybucja oprogramowania

## Utrzymanie - konsekwencje

- Zabiegi związane z utrzymaniem zazwyczaj psują strukturę kodu, kod staje się coraz trudniejszy do zrozumienia i dalszych modyfikacji
- Programy mogą być automatycznie modyfikowane aby usunąć np. nieużywane fragmenty kodu
- Warunki mogą być upraszczane aby poprawić ich czytelność

### Refaktoryzacja:

- zmiana wewnętrznej struktury kodu która czyni go czytelniejszym i łatwiejszym w modyfikacji jednak nie zmienia jego obserwowalnego zachowania
- eliminacja tzw. „brzydkich zapachów” w kodzie (ang. *bad smells*)

### Restrukturyzacja:

- zmiany na wyższym poziomie abstrakcji – architektura, sposób komunikacji między modułami, interfejsy

*Nowe błędy, ponowne testy, ...*

# Zapewnianie jakości

*Zasady dobrego stylu projektowania i programowania*

*Wzorce projektowe*

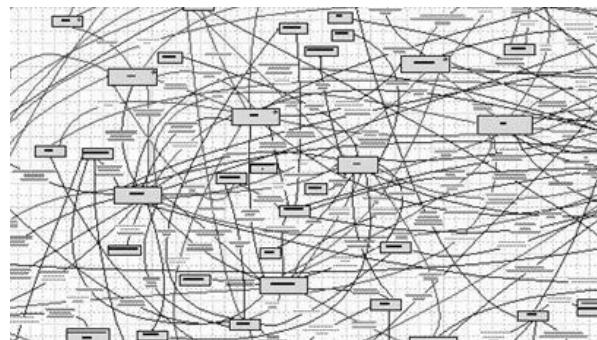
*Wykrywanie defektów i zapobieganie defektom*

*Metryki (np. jakość kodu, liczba defektów, pokrycie testami)*

*Standardy*

*Zarządzanie procesami*

*(...)*



# Zapewnianie jakości - przedmioty ‘troski’

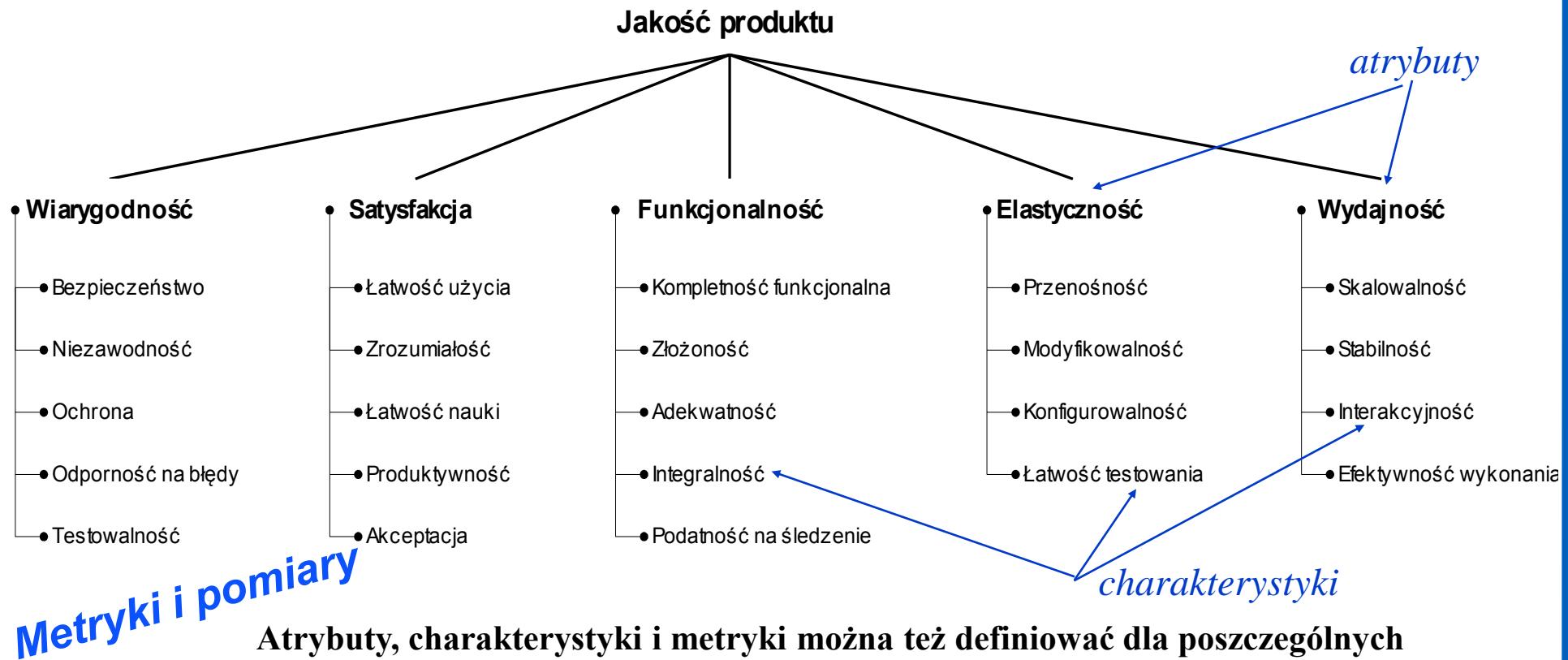
## Jakość :

- (1) Stopień w jakim system, komponent lub proces spełnia wyspecyfikowane wymagania
- (2) Stopień w jakim system, komponent lub proces spełnia potrzeby lub oczekiwania klienta lub użytkownika

- **Procesy** - czynności realizowane w trakcie wytwarzania oprogramowania
- **Produkty** - tworzone w procesie artefakty: kod, dokumenty, ...
- **Zasoby** - wejścia procesów; wszystkie elementy, które są konieczne dotworzenia oprogramowania

# Jak ocenić/zmierzyć jakość oprogramowania?

- Modele hierarchiczne np. norma ISO 25010, wcześniej ISO 9126



Atrybuty, charakterystyki i metryki można też definiować dla poszczególnych produktów: kodu, specyfikacji wymagań, projektu, dokumentacji,...  
Analogicznie, również dla procesów

# Zarządzanie



# Zarządzanie projektem

- **Co to znaczy, że projekt osiągnął sukces?**
- **Wiele wymiarów:**
  - Cele (<- zakres, <- wymagania)
  - Budżet / koszty
  - Czas / harmonogram
  - Jakość (czasem ujmowana w ramach pierwszego wymiaru, czasem wyróżniana jako osobny wymiar)
- **Nie tylko aspekty bezpośrednio związane z wytwarzaniem!**
- **Im większy projekt, tym więcej wysiłku wymaga planowanie, koordynacja prac, komunikacja, ...**

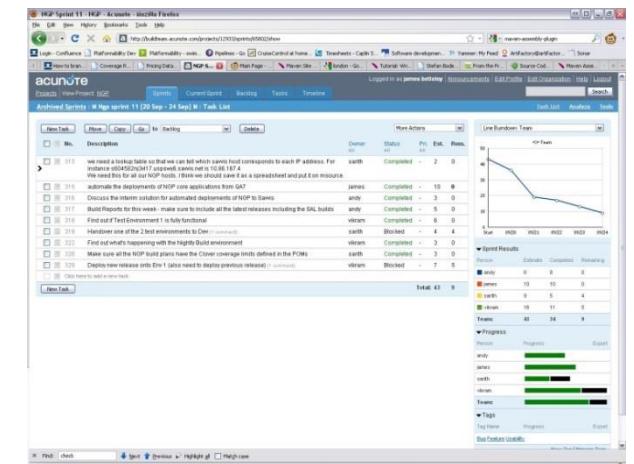
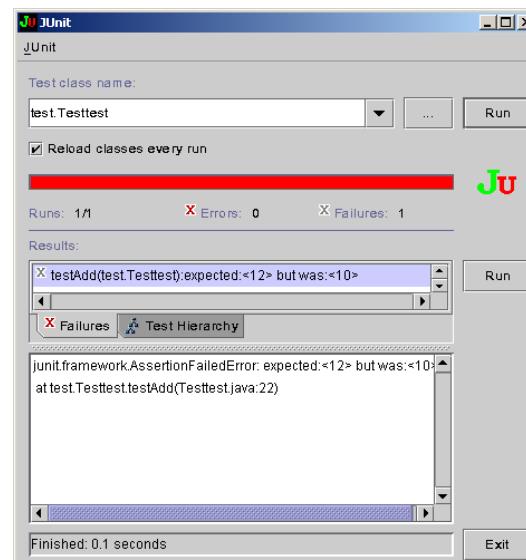
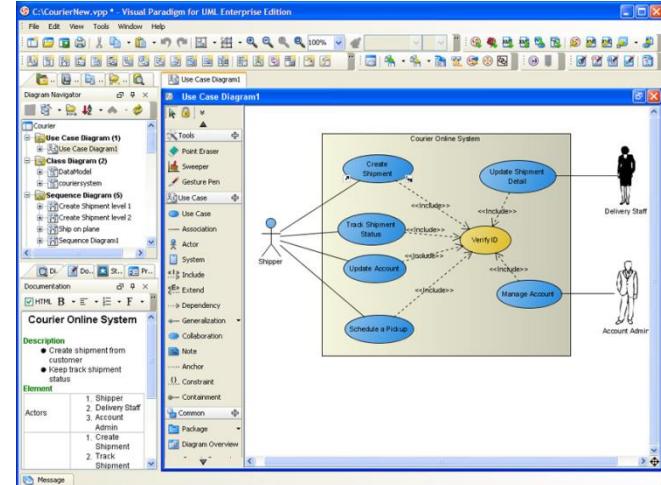
# Przykładowe obszary zarządzania

- Zarządzanie zakresem / interesariuszami
- Zarządzanie czasem
- Zarządzanie kosztami
- Zarządzanie ludźmi / zespołem
- Zarządzanie komunikacją
- Zarządzanie konfiguracją
- Zarządzanie zmianą
- Zarządzanie jakością
- Zarządzanie ryzykiem
- .....



# Wspomaganie narzędziowe IO

- IDE
- CASE
- Requirements management
- Configuration management
- Change management
- CAST
- Refaktoryzacja
- CI/CD
- ...



# Główne obszary IO (przypomnienie)



J.Miler

# Procesy wytwarzania oprogramowania (1)

- Jak powiązać ze sobą wymienione obszary wytwarzania oprogramowania (analizę, projektowanie, implementację, testowanie, wdrożenie, do tego zapewnianie jakości i zarządzanie) w spójny proces?
- Proces wytwarzania - Struktura obrazująca podstawowe obszary działań w rozwoju i eksploatacji systemu wraz z ich kontekstem, produktami, wzajemnymi relacjami pomiędzy tymi obszarami oraz zależnościami w czasie



# Procesy wytwarzania oprogramowania (2)

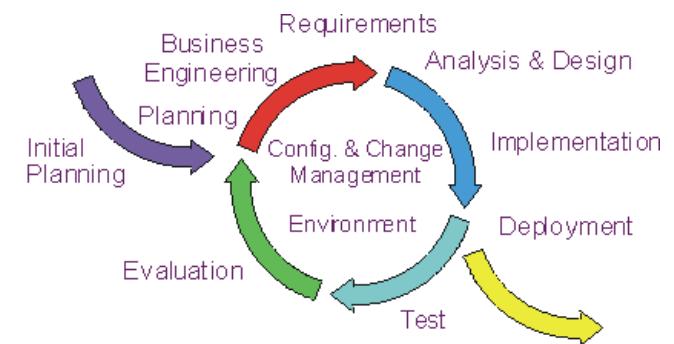
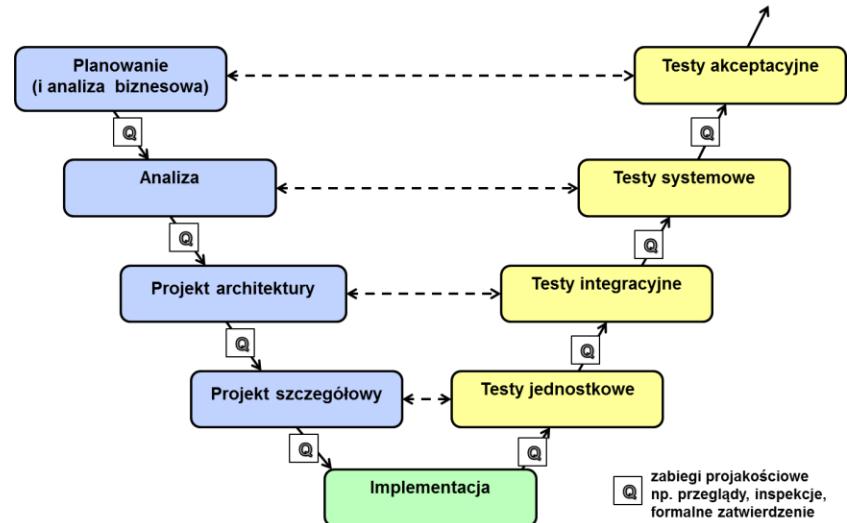
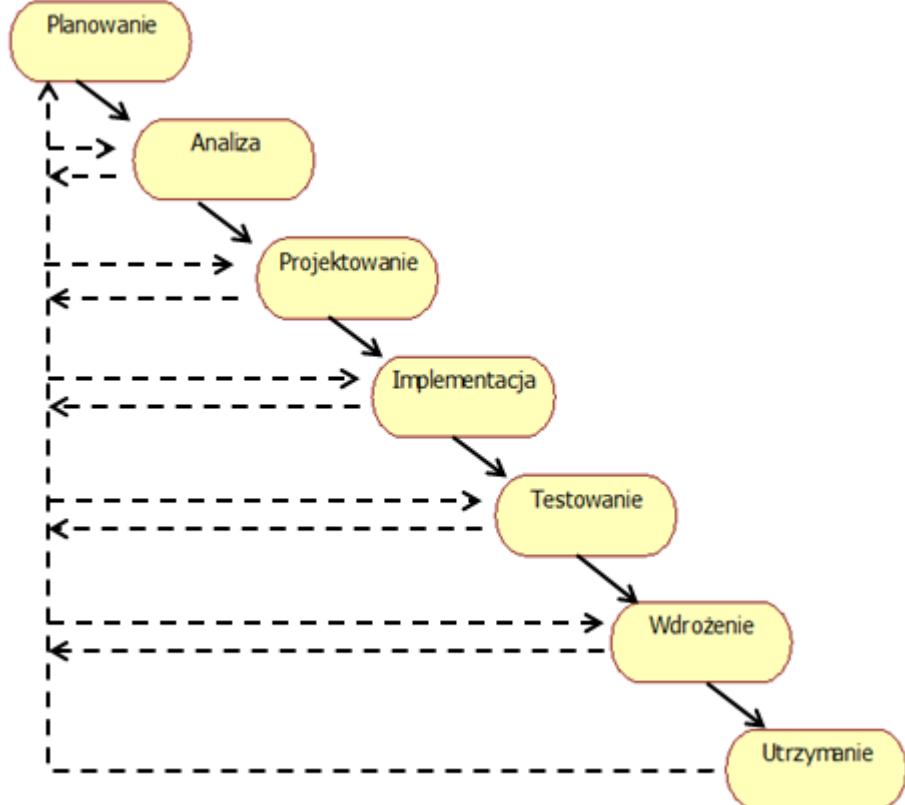
**Zależność logiczna (przyczynowo-skutkowa):**



**Zależność czasowa?**

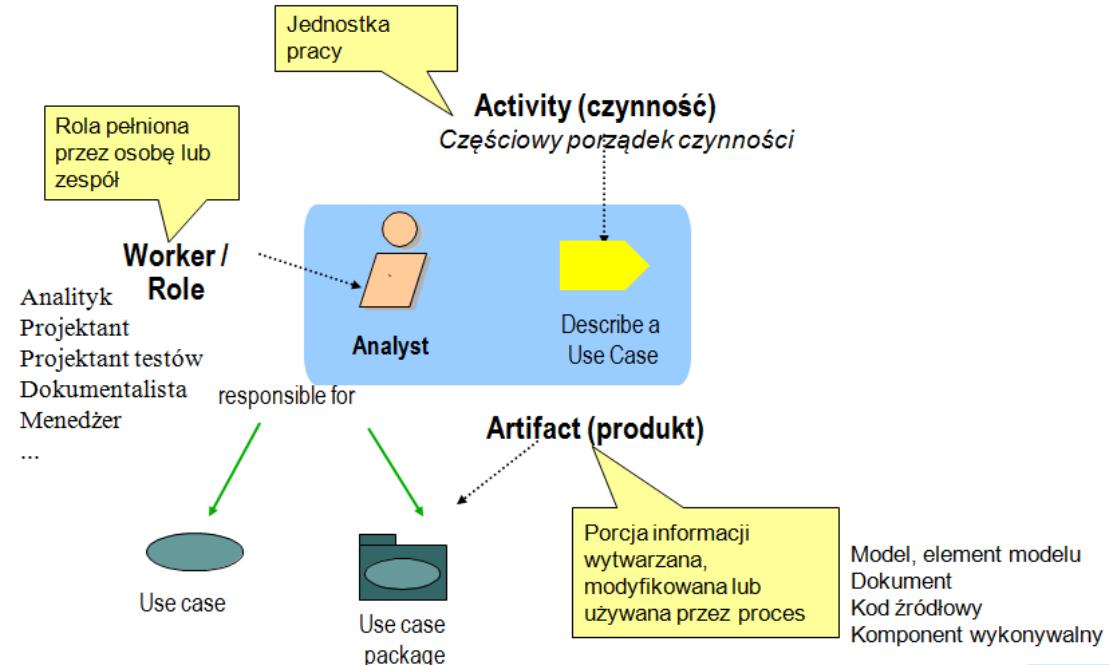
**Niekoniecznie musi to być sekwencja!**

# Procesy wytwarzania oprogramowania (3)



# Metodyki

- Metodyka to nie tylko ogólny pomysł procesu wytwarzania oprogramowania
- Metodyki na ogólnie dokładnie precyzują zadania/czynności, praktyki, produkty, role ludzi



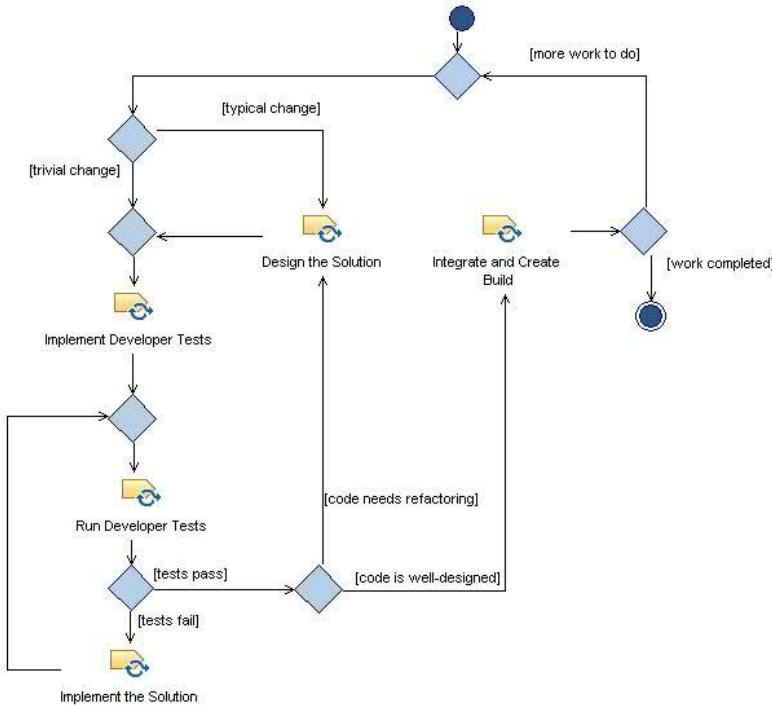
## Kategorie metodyk

- Wytwórcze (*Software Development Methodology*)
- Zarządzania (*Project Management Methodology*)
  
- Zdyscyplinowane (*Plan-Driven*)
- Zwinne (*Agile*)



# Metodyki wytwarzania

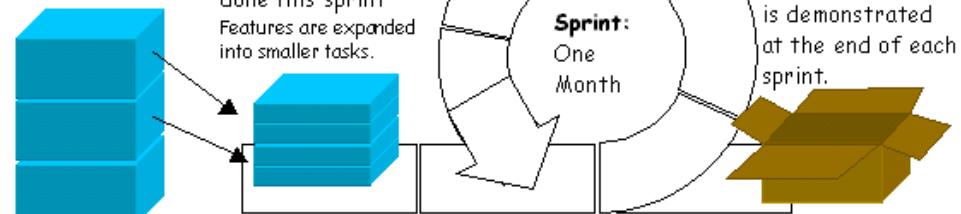
- „Przepis” na organizację procesu wytwarzania



## SCRUM Sprint Cycle

**Product Backlog:**  
Prioritized list of features required by the customer

**Sprint Backlog:**  
Features to be done this sprint  
Features are expanded into smaller tasks.



**Every Day**, a 15-minute meeting is held, and the SCRUM Master asks the 3 questions:

- 1) What have you accomplished since the last meeting?
- 2) Are there any obstacles in the way of meeting your goal?
- 3) What will you accomplish before the next meeting?

**New Functionality** is demonstrated at the end of each sprint.

# Analiza biznesowa i systemowa

*Aleksander Jarzębowicz*

*Katedra Inżynierii Oprogramowania  
Wydział ETI, Politechnika Gdańsk*

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich  
rozprowadzanie jest zabronione.

# Literatura

## Książki:

- Wiegers K., Beatty J., *Specyfikacja oprogramowania. Inżynieria wymagań*, 3rd Edition, Helion, 2014
- Chrabski B., Zmitrowicz K., *Inżynieria wymagań w praktyce*, PWN, 2015
- Zmitrowicz K., Roman A., *Inżynieria wymagań. Studium przypadków*, PWN, 2018

## Standardy:

- ISO/IEC/IEEE Std 29148-2018, *Systems and software engineering — Life cycle processes — Requirements engineering*

## Standardy przemysłowe / Materiały certyfikacji:

- International Institute of Business Analysis, *A Guide to the Business Analysis Body of Knowledge (BABOK)*, ver. 3, 2015
- International Institute of Business Analysis, *Agile Extension to the BABOK Guide*, 2017
- Project Management Institute, *Business Analysis for Practitioners: A Practice Guide*, PMI, 2015
- International Requirements Engineering Board, *IREB Certified Professional for Requirements Engineering*, ver. 3, 2020
- Project Management Institute, *Requirements Management: A Practice Guide*, 2016

# Motywacja – statystyki, przyczyny porażek

Project Impaired Factors	% of Responses
1. Incomplete Requirements	13.1%
2. Lack of User Involvement	12.4%
3. Lack of Resources	10.6%
4. Unrealistic Expectations	9.9%
5. Lack of Executive Support	9.3%
6. Changing Requirements & Specifications	8.7%
7. Lack of Planning	8.1%
8. Didn't Need It Any Longer	7.5%
9. Lack of IT Management	6.2%
10. Technology Illiteracy	4.3%
Other	9.9%

Źródło: Standish Group,  
Chaos Report 2015

## Przyczyny porażek projektów:

- Cele nierealistyczne lub w ogóle nie sformułowane
- Błędne oszacowanie potrzebnych zasobów
- Źle zdefiniowane wymagania
- Brak raportowania stanu projektu
- Brak zarządzania ryzykiem
- Słaba komunikacja
- Wykorzystanie niedojrzalej technologii
- Problemy z opanowaniem złożoności projektu
- Naganne praktyki deweloperskie
- Słabe zarządzanie projektem

Źródło: Charette R., *Why software fails*,  
IEEE Spectrum, Sept 2005

Reason for Failure	All
Lack of executive support	17%
Incomplete requirements	13%
Expectations not set / managed	12%
Scope creep	12%
Project / Organisation not aligned	10%
Lack of resources	7%
Technology Issues	7%
Other	6%
Lack of user involvement	6%
Poor Planning	5%
Inexperienced PPM People	4%

Źródło: Arras People Group,  
2010 Project Management  
Benchmark Report

# Motywacja – statystyki, przyczyny porażek

Project Impaired Factors	% of Responses
1. Incomplete Requirements	13.1%
2. Lack of User Involvement	12.4%
3. Lack of Resources	10.6%
4. Unrealistic Expectations	9.9%
5. Lack of Executive Support	9.3%
6. Changing Requirements & Specifications	8.7%
7. Lack of Planning	8.1%
8. Didn't Need It Any Longer	7.5%
9. Lack of IT Management	6.2%
10. Technology Illiteracy	4.3%
Other	9.9%

## Przyczyny porażek projektów:

- Cele nierealistyczne lub w ogóle nie sformułowane
- Błędne oszacowanie potrzebnych zasobów
- Źle zdefiniowane wymagania
- Brak raportowania stanu projektu
- Brak zarządzania ryzykiem
- Słaba komunikacja
- Wykorzystanie niedojrzalej technologii
- Problemy z opanowaniem złożoności projektu
- Naganne praktyki deweloperskie
- Słabe zarządzanie projektem

Źródło: Charette R., *Why software fails*, IEEE Spectrum, Sept 2005

Reason for Failure	All
Lack of executive support	17%
Incomplete requirements	13%
Expectations not set / managed	12%
Scope creep	12%
Project / Organisation not aligned	10%
Lack of resources	7%
Technology Issues	7%
Other	6%
Lack of user involvement	6%
Poor Planning	5%
Inexperienced PPM People	4%

Źródło: Arras People Group,  
2010 Project Management  
Benchmark Report

## Motywacja c.d.

- **Zagadnienia związane z wymaganiami i komunikacją z interesariuszami należą do głównych czynników ryzyka i przyczyn porażek projektów**
- **A ponadto, tak na „chłopski rozum”:**
  - Konieczność zrozumienia dziedziny problemowej (pojęć, języka, potrzeb, punktów widzenia)
  - Konieczność zrozumienia, co ma robić budowany przez nas system i z jakimi gwarancjami/ograniczeniami
  - Konieczność wywiązania się ze zobowiązań wobec klienta (to jest transakcja – klient płaci, my mamy za to dostarczyć system odpowiadający jego potrzebom!)

# Pojęcia

- **Inżynieria wymagań** – dziedzina zajmująca się identyfikowaniem, dokumentowaniem, analizowaniem, walidacją oraz utrzymaniem zbioru wymagań dla systemu IT
- **Analiza systemowa** – działania skoncentrowane na zrozumieniu wymagań względem tworzonego systemu (inżynieria wymagań, plus ewentualnie studia wykonalności lub/i rozszerzone modelowanie analityczne)
- **Analiza biznesowa** – działania związane ze zrozumieniem obszaru problemowego (np. organizacji klienta), sformułowaniem celów, definicją procesów biznesowych i aspektami ekonomicznymi przedsięwzięcia
- **Ale – uwaga 1:** trudno jest zdecydowanie rozdzielić analizę biznesową i systemową, częściowo się przenikają
- **Ale – uwaga 2:** często spotyka się też używanie terminu „analiza biznesowa” jako pojęcia pokrywającego wszystkie 3 powyższe obszary

# **Analiza biznesowa**

(rozumiana w tym węższym znaczeniu):

- **Identyfikacja obszaru problemowego i jego granic (najczęściej informatyzowanej organizacji lub jej części)**
- **Identyfikacja celów biznesowych i związanych z nimi strategii**
- **Identyfikacja zmian przynoszących organizacji korzyści**
- **Identyfikacja stanu obecnego i docelowego organizacji, przekształcenia jej procesów biznesowych**
- **Decyzja o uruchomieniu projektu/projektów IT (lub nie)**
- **Aspekty ekonomiczne projektu IT i jego spodziewanych rezultatów**

# Inżynieria wymagań

- Podstawowa część analizy systemowej
- Termin „inżynieria” sugeruje, że działania te podlegają planowaniu oraz bazują na technikach systematycznych i powtarzalnych
- Termin „wymagania”?

# Pojęcie wymagania

## Requirement:

**statement which translates or expresses a need and its associated constraints and conditions**

- **Note 1 to entry:** Requirements exist at different levels in the system structure.
- **Note 2 to entry:** A requirement is an expression of one or more particular needs in a very specific, precise and unambiguous manner.
- **Note 3 to entry:** A requirement always relates to a system, software or service, or other item of interest.

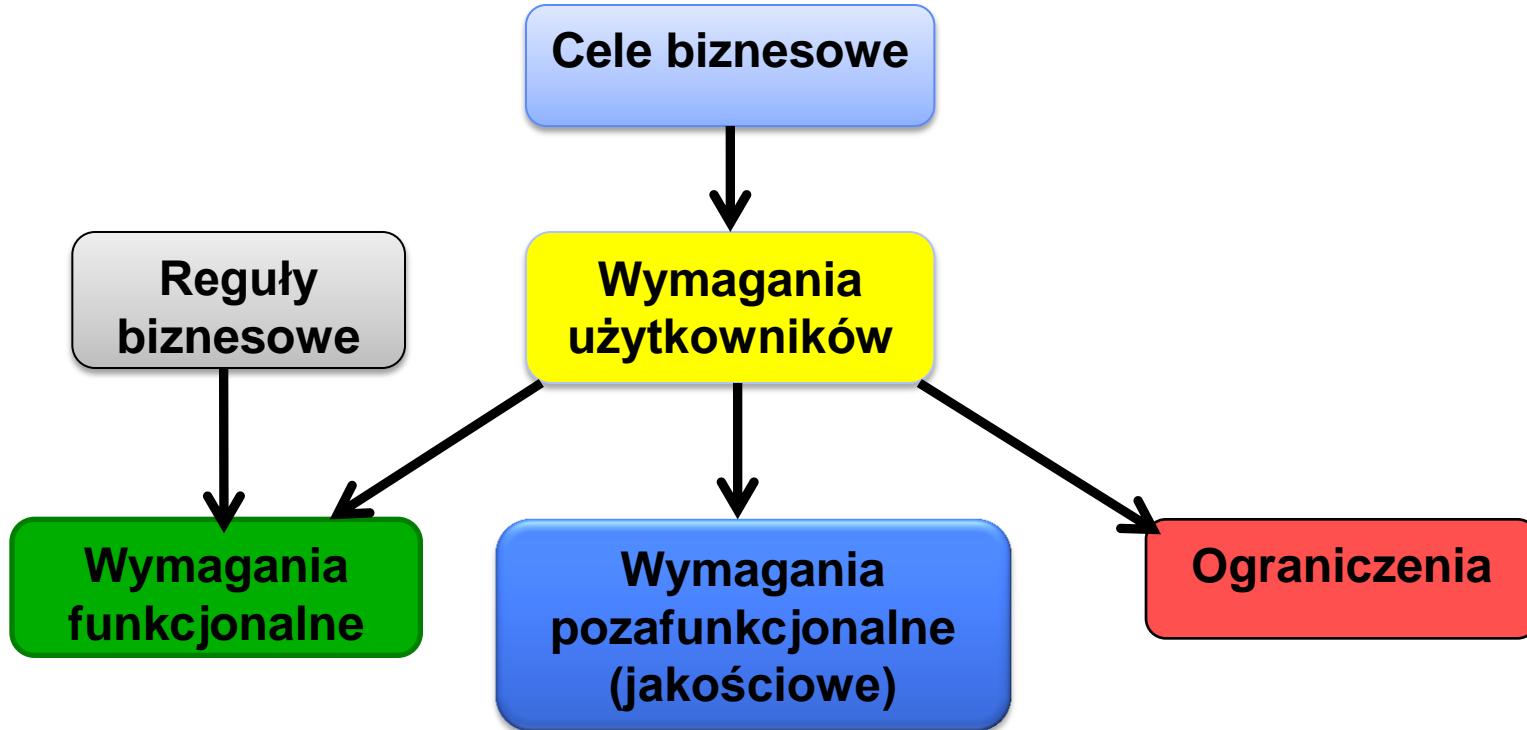
**International Standard ISO/IEC/IEEE 29148:2018**

# Wymagania względem systemu

- Wymagania mogą odnosić się do różnych cech systemu oraz procesu jego wytwarzania lub/i utrzymania (zróżnicowany poziom ogólności, proces/produkt, funkcjonalne/pozafunkcjonalne, ograniczenia, założenia, itp.)
- W typowych sytuacjach wymagania stanowią zbiór informacji dotyczących dziedziny problemowej, własności i zachowań systemu, oraz różnego rodzaju ograniczeń (projektowych, wytwórczych, środowiskowych, prawnych itp.).
- Wymagania (na najwyższym poziomie abstrakcji) przekazują uzasadnienie dla potrzebytworzenia systemu - określają jego cele i wyjaśniają, na ile i dlaczego są one ważne
- Wymagania powinny odzwierciedlać to czego Klient/Odbiorca/Użytkownicy *potrzebują* a nie to czego *chcą*

**DLACZEGO? – ważne pytanie analityczne**

# Kategorie wymagań



- Uwaga – celów biznesowych, reguł biznesowych i wymagań użytkowników nie uwzględnialiśmy na laboratorium w Wizji Systemu

## Cele biznesowe

- **Jaki jest powód i uzasadnienie całego projektu i jego kosztów?**
- **Jakich korzyści oczekuje odbiorca systemu po jego wdrożeniu?**
  - Rozwiążanie istniejących problemów np. opóźnień, popełnianych błędów
  - Rozszerzenie udziału w rynku
  - Zwiększenie zysku
  - Oferowanie konkurencyjnych produktów/usług
  - .....
- **Cele powinny być odpowiednio formułowane np. według podejścia SMART postulującego, aby cel był:**
  - Skonkretyzowany (ang. **Specific**) – konkretny i jednoznacznie zrozumiały,
  - Mierzalny (ang. **Measurable**) – sformułowany liczbowo lub w inny sposób pozwalający jednoznacznie sprawdzić stopień jego realizacji,
  - Osiągalny (ang. **Achievable**) – po prostu możliwy do realizacji,
  - Istotny (ang. **Relevant**) – mający rzeczywiste znaczenie i wartość dla organizacji
  - Określony w czasie (ang. **Time-bound**) – z określonym horyzontem czasowym, w jakim ma być osiągnięty

# S.M.A.R.T.

Cel biznesowy systemu:

## 1. Poprawa jakości obsługi klienta

- ale co to konkretnie znaczy? poprawa, znaczy się o ile?
- not SMART

## 2. Skrócenie średniego czasu realizacji zamówienia z 5 do 2 dni w ciągu 3 miesięcy po wdrożeniu systemu

- Specific: średni czas realizacji zamówienia
- Measurable: 5 dni -> 2 dni
- Achievable: niekoniecznie samo sformułowanie, może być kontekst - upewnienie się czy fizycznie możliwa jest realizacja w 2 dni
- Relevant: niekoniecznie samo sformułowanie, może być kontekst – czy to coś da firmie? bo może klienci bez problemu akceptują te 5 dni?
- Time-bound: w ciągu 3 miesięcy po wdrożeniu

# Wymagania użytkowników

- **Opisują (na wysokim poziomie abstrakcji) co użytkownicy będą mogli robić za pomocą systemu np.**
  - dokonywanie zakupów w sklepie internetowym
  - zarządzanie dokumentacją medyczną pacjentów
  - obsługa reklamacji klientów
  - bezpieczne transakcje płatności
- **Jest to dość naturalny sposób wyrażania wymagań przez użytkowników, natomiast wymaga doprecyzowania w postaci konkretniejszych funkcji, wymagań jakościowych itp.**

# Reguły biznesowe

- **Reguły wynikające z przepisów prawnych, polityki/regulaminów organizacji, wiedzy dziedzinowej np.**
  - Kwota wyплатy z bankomatu nie może przekraczać sumy środków dostępnych na koncie oraz debetu ustalonego dla danego typu karty
  - Pracownik musi wykorzystać cały przysługujący mu urlop do końca danego roku kalendarzowego, przy czym co najmniej jedna część urlopu musi trwać nie mniej niż 10 kolejnych dni kalendarzowych
  - Do zamówień o wartości poniżej 200 PLN sklep dolicza koszty przesyłki, dla 200 PLN i większych kwot przesyłka jest bezpłatna
- **Reguły biznesowe będą docelowo implementowane jako różnego rodzaju warunki sprawdzane i egzekwowane przez system**



# Wymagania funkcjonalne

**Przede wszystkim odpowiedź na pytania:**

- co ten system ma robić?
- jakie funkcje/usługi udostępniać użytkownikom?

**Dodatkowo:**

- **Dane**
  - napływające, przechowywane, wychodzące,
  - zakres danych
  - formaty
  - udostępnianie
- **Inne systemy**
  - zakres współpracy
  - funkcje udostępniane lub wywoływane
  - interfejsy między systemami
  - format wymienianych danych
  - współdzielone dane

# Wymagania pozafunkcjonalne (jakościowe)

- Mniej oczywiste niż funkcjonalne !
- Dotyczą tego jakimi gwarancjami jakościowymi powinna być opatrzona dostarczana funkcjonalność
  - OK, system ma umożliwiać klientowi korzystanie z funkcji bankowości internetowej (*/specyfikacja funkcji/*), ale jak wydajnie musi działać ten system? jaki powinien być interfejs użytkownika? jakie gwarancje z zakresu *security*?
- Podstawowe problemy:
  - wymagania pozafunkcjonalne nie są specyfikowane
  - są specyfikowane w sposób niemierzalny / nieweryfikowalny, nie przekładający się na żadne konkretы („**System powinien działać szybko i mieć ładny interfejs**”)
  - są specyfikowane od razu w kategoriach rozwiązań projektowych – może to być robione przez interesariuszy, ale również deweloperów, którzy z miejsca przechodzą do nasuwających się im rozwiązań zawężając możliwość wyboru („**Powiedzieli, że system ma być wydajny? Pisz, że ma chodzić na silniku bazowanym X.**”)

## Wymagania pozafunkcjonalne (jakościowe) c.d.

- Przypomnienie głównych kategorii:

- wydajność – jakie są wymagania odnośnie liczby użytkowników, wolumenów danych, czasu reakcji oraz jak te parametry mogą się zmieniać w dłuższym horyzoncie czasowym
- niezawodność – jak ważne jest aby na pewno nie pojawiły się w działaniu systemu żadne błędy
- dostępność – ile czasu i ew. w jakich porach system może nie działać/ być wyłączony
- ochrona (security) – na ile system musi być odporny na ataki z zewnątrz
- bezpieczeństwo (safety) – na ile system musi dawać gwarancje, że nie zagrozi swojemu otoczeniu (dla systemów krytycznych)
- przenośność – na ile wymagane jest aby system działał na różnych urządzeniach, platformach, OSach, przeglądarkach itp.
- elastyczność – na ile system powinien być możliwy do łatwej dalszej rozbudowy
- konfigurowalność – czy dla systemu można zidentyfikować kluczowe parametry, które mogą się zmieniać i trzeba umożliwić zmianę zamiast ich „twardego wkodowania” (np. stawki VAT)
- interfejs użytkownika – szybkość pracy z systemem, łatwość nauki, estetyka interfejsu

# Ograniczenia związane z produktem (systemem)

- **Konieczność działania w specyficznych warunkach**
  - środowisko mobilne, wysokie/niskie temperatury, kurz, wibracje, uszkodzenia
- **Określony sprzęt**
  - klient określa konkretny sprzęt lub klasę sprzętu, na którym ma działać system
- **Narzucona technologia wykonania**
  - wybór języka programowania, frameworku, DBMS itp. określony przez klienta
- **Wymagana dokumentacja**
  - np. instrukcja użytkownika, instrukcja dla administratora, dokumentacja techniczna (wymagania, projekt architektury, projekt BD), poradnik maintenance (dalszego rozwoju)
- **Wymagane prowadzenie szkoleń**
  - dla użytkowników przez reprezentantów wytwórcy
- **Sposób wdrożenia**
  - naraz czy etapami, próbne wdrożenie na niektórych stanowiskach, wdrożenie niezakłócające funkcjonowania organizacji (np. w weekend)
- **Zgodność produktu/procesu wytwarzania ze standardami**
- **Inne specyficzne wymagania użytkownika**

# Ograniczenia związane z procesem

- **Czasowe**
  - do kiedy ma powstać system, czy będą jakieś etapy pośrednie (tzw. kamienie milowe)
- **Budżetowe**
  - jaki budżet jest do dyspozycji, kiedy te środki będą dostępne
- **Organizacyjne**
  - sposób organizacji projektu np. w projektach dla administracji państwowej w pewnym okresie wymagano zgodności z metodyką zarządzania PRINCE2
- **Ludzkie**
  - zasoby ludzkie wytwórcy – liczba dostępnych osób, ich dyspozycyjność, kompetencje
- **Sprzętowe**
  - jaki sprzęt ma do dyspozycji wytwórcza (stanowiska deweloperów, serwer produkcyjny, serwer testowy, dedykowane urządzenia)
- **Oprogramowanie**
  - jakie oprogramowanie ma do dyspozycji wytwórcza (IDE, narzędzia do zarządzania, testowania etc., kwestia licencji...)
- **Inne**

# Obszary działania IW

- **Wydobywanie wymagań** – identyfikowanie wymagań i pozyskiwanie ich od interesariuszy
- **Analizowanie wymagań** – przetworzenie wydobytych wymagań w spójną i kompletną całość
- **Specyfikowanie wymagań** – zapis wymagań w jednoznacznej i zrozumiałej postaci
- **Walidacja wymagań** – potwierdzenie u interesariuszy poprawności i kompletności wymagań
- **Zarządzanie wymaganiami** – utrzymywanie wymagań w trakcie całego projektu i wprowadzanie do nich zmian

# Procesy Inżynierii Wymagań

Źródło: Wiegers, Beatty, Software Requirements, 3rd Ed.

## Zarządzanie wymaganiami

Wydobywanie  
wymagań

Analizowanie  
wymagań

Specyfikowanie  
wymagań

Walidacja  
wymagań

Wyjaśnianie

Uzupełnianie luk

Przepisywanie

Ponowne ocenianie

Potwierdzanie i korekty

- Nie sekwencja, tylko proces iteracyjny, konieczność cofania się itp.

# Wydobywanie wymagań

- **Identyfikacja źródeł wymagań (interesariuszy)**
- **Określenie przedstawicieli interesariuszy, od których będą wydobywane wymagania**
  - reprezentatywność!
- **Dotarcie do tych przedstawicieli**
- **Nawiązanie współpracy**
  - Może wymagać zgody przełożonych, wydzielenia części czasu pracy przedstawiciela na obowiązki związane z udziałem w wydobywaniu wymagań
- **Dopiero wtedy samo pozyskiwanie wymagań...**

# Problemy przy wydobywaniu wymagań (1)

- **Identyfikacja interesariuszy i ich reprezentatywność**
  - można przeoczyć istotnego interesariusza, który ujawni się później ☹
  - dana osoba/źródło niekoniecznie muszą być reprezentatywne
- **Brak zaangażowania**
  - tłumaczony np. brakiem czasu
- **Komunikacja**
  - klient i twórca często mówią innymi językami (w sensie: twórca technicznym, klient dziedzinowym z branży np. medycznej, handlowej, prawniczej, technicznej ale i innej np. subkultur)
- **Wymagania nie są dostępne w gotowej postaci**
  - „pójdę, zapytam, a on mi wszystko powie...”
  - raczej nie powie „z głowy”, trzeba zadać konkretniejsze pytania, wychwycić „luki” i sprzeczności etc.
  - wymagana intensywna komunikacja

# Problemy przy wydobywaniu wymagań (2)

- **Interesariusz może nie wiedzieć czego chce**
  - może uświadomić sobie swoje potrzeby dopiero gdy zobaczy coś bardziej „namacalnego” (sam system, prototyp interfejsu etc.)
  - nie wie co jest możliwe do zrealizowania w danej technologii
- **Nierealistyczne oczekiwania**
  - niemożliwe do zrealizowania (w ogóle albo w ramach ograniczeń projektu)
- **Wiedza dziedzinowa**
  - przyswojenie aparatu pojęciowego, realiów
  - rzeczy „oczywiste”, które dla osoby z zewnątrz wcale oczywiste nie są
- **Problemy natury ludzkiej**
  - różne typy osobowości (np. gaduła, osoba zamknięta w sobie, choleryk)
  - problematyczne zachowania (np. nie zna odpowiedzi, ale nie przyzna się, tylko zgaduje)
- (...)

# Techniki wydobywania wymagań

- Wywiady
- Warsztaty
- Burze mózgów
- Kwestionariusze
- Prototypowanie
- Analiza dokumentów
- Analiza istniejącego systemu
- Obserwacje
- Terminowanie
  
- Zwykle w danym projekcie wykorzystuje się łącznie kilka technik!

# Technika: Wywiady

- **Cele:**
  - Pozyskanie informacji o obszarze problemowym (zadania, procedury itp.)
  - Pozyskanie informacji na temat oczekiwania względem nowego systemu
  - Zweryfikowanie zrozumienia przez analityka wymagań pozyskanych z różnych źródeł
- **Typy wywiadów:**
  - Strukturalne (wcześniej przygotowana lista pytań)
  - Niestrukturalne (rozmowa otwarta, bez listy pytań)
- **Problemy:**
  - zwykle wymaga istotnego nakładu na zorganizowanie i przeprowadzenie
  - brak odpowiedniej wiedzy respondenta, zgadywanie odpowiedzi, niechęć
  - problemy komunikacyjne
  - wiedza uznawana za oczywistą

## Technika: Warsztaty

- Spotkanie grupy interesariuszy w celu wspólnej dyskusji na temat wymagań widzianych z ich punktów widzenia i uzgodnienia wspólnego stanowiska
- Warsztaty często bazują na różnych scenariuszach, uczestnicy poprzez wykonanie pewnych zadań mają lepiej zrozumieć i wyrazić wymagania
- Problemy:
  - Dojście do wspólnego stanowiska może być czasochłonne, trudne, a czasem wręcz niemożliwe
  - Możliwość odejścia od dyskusji merytorycznej (kłótnie, rozgrywki interpersonalne, lokalna „polityka”)
  - Kwestie osobowościowe np. osoby dominujące i nieśmiałe
  - Konieczny odpowiedni moderator

## Technika: Burze mózgów

- Spotkanie grupy interesariuszy wg określonego scenariusza ukierunkowane na generowanie nowych, nieszablonowych pomysłów (np. na rozwiązanie danego problemu, na innowacyjne usługi systemu) z odroczoną krytyką/oceną tych propozycji
- Istnieją konkretniejsze wersje tej techniki np. spojrzenie z innej perspektywy czy podejście „jak mógłbym spowodować taki problem?”
- Problemy:
  - Analogicznie jak dla warsztatów – problemy natury ludzkiej lub lokalnej polityki
  - Zależy od inicjatywy i kreatywności uczestników – w przypadku braku chęci, zmęczenia czy innych tego typu czynników nic z tego nie wyjdzie
  - Potencjalne trudności z przestrzeganiem ustalonego scenariusza i np. dążenie do natychmiastowej oceny rozwiązań, co przeszkadza w kreatywności

## Technika: Kwestionariusze

- **Możliwe do zastosowania wtedy gdy inne metody są niemożliwe lub zbyt kosztowne (zbyt wielu respondentów, niemożność dotarcia do niektórych)**
- **Pytania „zamknięte” i „otwarте”**
- **Istnieją zasady i dobre praktyki formułowania pytań i konstrukcji całych kwestionariuszy**
- **Problemy:**
  - często nikły odsetek odpowiedzi
  - kwestia reprezentatywności respondentów
  - niejednoznaczność/niezrozumiałość pytań
  - pytania „zamknięte” często ograniczają, pytania „otwarте” trudne do przetworzenia i analizy
  - zwykle brak możliwości „dopytania”

# Technika: Prototypowanie (1)

- **Zamiast oczekiwania od interesariusza (np. przyszłego użytkownika), że wyobrazi sobie system i jego możliwości, dostarcza mu się prototyp systemu**
- **Prototyp stanowi ograniczoną reprezentację przyszłego systemu np. samego interfejsu użytkownika**
- **Użytkownik oglądając prototyp i eksperymentując z nim nabiera pojęcia o docelowym systemie, wyraża swoje opinie o spełnieniu jego wymagań i formuluje nowe wymagania**
- **Zróżnicowane rodzaje prototypów**
  - papierowy
  - symulowany (**Wizard of Oz**)
  - software'owy „do wyrzucenia”
  - software'owy - rozwojowy

# Technika: Prototypowanie (2)

- **Prototypowanie warto zastosować w sytuacjach gdy:**
  - budujemy nowy produkt, który dotąd nie istniał i jest trudny do wyobrażenia
  - interesariusze nie mają wcześniejszego doświadczenia z podobnym produktem ani z technologią proponowaną do jego wytworzenia
  - interesariusze są tak przyzwyczajeni do dotychczasowego sposobu pracy, że trudno im sobie wyobrazić, że można to robić inaczej
  - interesariusze mają problemy w artykułowaniu swoich wymagań
  - analitycy mają problemy w zrozumieniu co jest oczekiwane przez interesariuszy
  - powstały wątpliwości, czy dane wymaganie jest wykonalne
- **Problemy:**
  - czas/praca na przygotowanie i dostosowanie prototypów
  - wyjaśnienie interesariuszom, że prototyp to nie gotowy produkt ☺

## Technika: Analiza dokumentów

- Duża ilość informacji o danej dziedzinie, organizacji, dla której wytwarzany jest system oraz dotychczasowych systemów można znaleźć w różnego rodzaju źródłach pisanych
- W przypadku niektórych interesariuszy nieożywionych (systemy, przepisy) może to być główne źródło pozwalające ustalić taki punkt widzenia
- Potencjalne źródła:
  - Ogólna wiedza dziedzinowa
  - Regulacje i przepisy
  - Wewnętrzne regulaminy
  - Schematy organizacyjne
  - Plany strategiczne
  - Opisy procedur administracyjnych
  - Opisy stanowisk pracy
  - Materiały szkoleniowe
  - Opisy istniejących lub planowanych systemów i ich interfejsów
  - (...)

# Technika: Analiza istniejącego systemu

- Chodzi o sytuację gdy nowy system ma (przynajmniej w jakimś stopniu) zastąpić stary, już użytkowany
- Istnieje wówczas możliwość, żeby pozyskać wymagania traktując „stary” system jako pewien punkt odniesienia np.
  - Na czym polegają obecne problemy/ograniczenia?
  - Dlaczego istniejący system jest zamieniany na nowy?
  - Jakie funkcje/cechy mają zostać przejęte ze starego systemu, a jakie należy pominąć i dlaczego?
  - Jakie są nowe funkcje/cechy przewidziane dla systemu?
  - Czy kontekst organizacyjny systemu pozostanie bez zmian?
- Jest to sytuacja w pewnym sensie podobna do prototypowania – interesariusz ma coś konkretnego, do czego może się odnosić
- Technika ta obejmuje też wydobycie kluczowych informacji o funkcjonowaniu istniejącego systemu (np. reguły biznesowe) z jego dokumentacji lub wręcz kodu (tzw. archeologia oprogramowania)

## Technika: Obserwacje

- **Analityk spędza dłuższy okres czasu obserwując przyszłych użytkowników w ich normalnych miejscach pracy**
- Występuje w 2 wersjach: „pasywnej” (analityk tylko patrzy, nie może ingerować) i „aktywnej” (w trakcie wydarzeń analityk może dopytywać o to, co się dzieje)
- Potencjalnie głęboki wgląd w dziedzinę i środowisko użytkownika
- **Identyfikacja niewypowiedzianych założeń**
- **Problemy:**
  - metoda czasochłonna
  - „efekt Hawthorne”
  - zdarzenia rzadkie mogą umknąć obserwacji
  - można obserwować tylko istniejące aktualne procesy, docelowe mogą być inne

## Technika: Terminowanie

- Przy terminowaniu (*ang. apprenticing*) podobnie w technice obserwacji analityk spędza dłuższy czas w danym środowisku
- Twarzyszy ekspertowi dziedzinowemu, od którego uczy się pracy czy obowiązków w tej dziedzinie (*analogicznie jak uczeń u rzemieślnika, stąd nazwa*)
- Dzięki temu analityk jest w stanie dogłębniej zrozumieć wymagania trudne do pozyskania innymi technikami
- Problemy:
  - metoda bardzo czasochłonna i wymagająca zaangażowania (zarówno analityka, jak i eksperta)
  - nie zawsze możliwa np. w warunkach niebezpiecznych

# Analizowanie wymagań

- **Przetworzenie uprzednio wydobytych wymagań**
  - czy to, co zebraliśmy od interesariusza, kwalifikuje się jako wymaganie?
- **Uporządkowanie**
  - przypisanie do kategorii (cele, wymagania użytkowników, wymagania funkcjonalne, ograniczenia itd.)
  - określenie zależności między wymaganiami
- **Analiza poszczególnych wymagań**
  - m.in. zrozumiałość, jednoznaczność, kompletność, wykonalność, testowalność, brak niepotrzebnych decyzji projektowych, ...
- **Analiza zbioru wymagań**
  - pod kątem luk, nadmiarowości, sprzeczności
- **Priorytetyzacja wymagań**

## Jednoznaczność wymagań

- Język naturalny jest podstawową formą komunikacji, ale bywa niejednoznaczny...
- „Zamówienie ma zawierać dane klienta, adres wysyłki itp.”  
*itp. czyli co?*
- „Klient ma zwykły status do złożonych 5 zamówień, potem zyskuje status VIP” *co dla 5?*
- „System ma uruchomić alarm, gdy wykryto ruch wewnętrz domu lub otwarte zostały drzwi i w ciągu 2 min nie został wpisany prawidłowy kod”
  - *(A or B) and C?*
  - *A or (B and C)?*
- „Aplikacja ma być dostępna za pośrednictwem wszystkich popularnych przeglądarek” *Opera? UC Browser? Konqueror?*

# Analizowanie wymagań - problemy

- **Spójność wymagań**
  - nawet pojedynczy interesariusz może mówić rzeczy sprzeczne, nie zdawać sobie sprawy z konsekwencji swoich wyborów
  - przy wielu źródłach wymagań, wymagania mogą być sprzeczne również z przyczyn obiektywnych (różne potrzeby)
- **Kompletność wymagań**
  - nie tyle rozumiana w kategoriach absolutnych, co na zasadzie niepominięcia czegoś ważnego dla kolejnego etapu/iteracji prac
  - łatwiej jest zauważać błąd czy sprzeczność, niż to czego brakuje!
- **Ustalenie priorytetów**
  - różni interesariusze mogą mieć odmienne potrzeby/zdanie
  - dochodzą dodatkowe czynniki np. ryzyko technologiczne, wzajemne zależności między funkcjonalnościami
- **Zapewne ujawni się potrzeba ponownego kontaktu z interesariuszami i doprecyzowania wymagań (powrót do obszaru wydobywania wymagań)**

# Dobór technik analizowania wymagań względem celów

- **Spójność** – przeglądy, modelowanie, macierze śladowości
- **Kompletność** – przeglądy, modelowanie, listy kontrolne, macierze śladowości, tabele CRUD
- **Priorytetyzacja** – określenie skali priorytetów np. MoSCoW, Timeboxing/budgeting, negocjacje interesariuszy

# Techniki analizowania wymagań - przykłady

- **Przeglądy** (spójność, kompletność)
  - połączenie wymagań od różnych interesariuszy
  - przegląd pod kątem nadmiarowości, sprzeczności i konfliktów, względnej ważności wymagań, kompletności (brakujących wymagań)
- **Listy kontrolne** (kompletność) zawierające odpowiednie pytania np.
  - Czy zebrano wymagania od wszystkich interesariuszy? Czy cały, wcześniej określony zakres projektu jest ‘pokryty’ wymaganiami? Czy wyłączono wymagania leżące poza zakresem?
  - Czy każde wymaganie jest powiązane z którymś z celów biznesowych systemu (ma uzasadnienie)?
  - Czy dla wymagań zdefiniowano mierzalne kryteria akceptowalności?
- **Modelowanie** (spójność, kompletność)
  - przy przekładaniu wymagań na reprezentację w postaci modelu ujawniają się problemy ze spójnością (nie wiadomo jaką drogę wybrać) czy kompletnością (brakuje jakieś informacji potrzebnej do stworzenia modelu)

# Technika – macierze śladowości

	Feature 1	Feature 2	Feature 3
Use Case 1	↑		
Use Case 2		↑	!
Use Case 3	↑		
Use Case 4		!	↑

- **Zestawienie dla wymagań tych samych lub różnych kategorii**
- **Pokazuje dekompozycję / wynikanie wymagań (tutaj to jak przypadki użycia realizują wysokopoziomowe wymagania użytkownika), mogą też pokazywać inne zależności (np. operowanie na tych samych danych, powiązane funkcjonalności)**
- **Możliwe jest zaznaczanie wymagań sprzecznych i niespójnych**

## Technika – tabele CRUD

	Create	Read	Update	Delete
Zamówienie	UC-3	UC-12, UC-16	???	UC-13
Klient	UC-1	UC-24, UC-28	UC-25	UC-31
Produkt	UC-31	UC-5, UC-12, UC-18	UC-33, UC-35	--- (potw.)
Kategoria	???	UC-12, UC-18 UC-22	UC-37	???

- **Każdy obiekt informacyjny w systemie zapewne będzie tworzony, odczytywany, zmieniany i usuwany (choć mogą istnieć wyjątki!), a więc powinien mieć funkcjonalność, która za te wszystkie rzeczy odpowiada**
- **CRUD pozwala zidentyfikować luki**

# Techniki priorytetyzacji

- **MoSCoW**
  - przypisanie priorytetów Must, Should, Could, Won't
- **Inna zdefiniowana skala (0-5, 1-10 etc.)**
- **Timeboxing/budgeting**
  - oszacowanie czasochłonności/kosztu poszczególnych wymagań i dobór takiego podzbioru żeby dał się zrealizować np. w następnej iteracji
- **Negocjacje interesariuszy**
  - kompromis, głosowanie, odwołanie do osoby decyzyjnej
  - różnego rodzaju „gry planistyczne” np. „kupowanie” nowych funkcji/cech tworzonego oprogramowania za określoną „kwotę”

# Specyfikowanie wymagań

- **Udokumentowanie wymagań w sposób spójny i zrozumiały dla wszystkich zainteresowanych stron**
- **Odpowiednie formy wyrazu dla różnych wymagań (funkcjonalne, jakościowe, reguły biznesowe itd.)**
- **Opisanie wymagań określonymi atrybutami np. ID, źródło, status**
- **Dokument wymagań (Specyfikacja Wymagań Systemowych – SWS, ang. System Requirements Specification – SRS)**

# Atrybuty opisu wymagania

- **Poza samym wyrażeniem treści wymagania, istotne są różne informacje, które dodatkowo je opisują np.**
  - Jednoznaczny identyfikator
  - Źródło wymagania (kto je zgłosił)
  - Odpowiedzialny (do którego analityka przypisane jest wymaganie)
  - Priorytet (wg przyjętej skali)
  - Złożoność (trudność, koszt realizacji lub inne oszacowanie)
  - Ryzyka (związane z implementacją tego wymagania)
  - Status (np. nowe, sprawdzone, zatwierdzone, zaimplementowane, ...)
  - Wersja
  - Wymagania powiązane
  - ...

## Przykład wymagania opisanego atrybutami

Identyfikator:	F1	Priorytet:	1	Status:	Zaakceptowany
Tytuł:	<b>System powinien zapewnić możliwość wspomagania procesu zarządzania problemami</b>				
Opis:	<p>Wymaganie wywodzi się bezpośrednio z celów biznesowych systemu. Wytworzenie i wdrożenie systemu ma na celu wspomaganie procesu zarządzania problemami przez gromadzenie danych na temat projektów i napotkanych w ich trakcie problemów. System musi udostępniać funkcjonalność w zakresie dodawania, edycji oraz usuwania danych istotnych z punktu widzenia tego procesu.</p> <p><u>Kryterium akceptacji</u> Pomyślne przejście testów walidacyjnych odnoszących się do kompletności funkcjonalnej systemu w myśl wymagań specyfikujących zakres funkcjonalności.</p> <p><u>Powiązania z celami biznesowymi</u> Wymaganie to jest kluczowe do wypełnienia wszystkich celów biznesowych systemu.</p> <p><u>Powiązanie z przypadkami użycia systemu</u> Wymaganie powiązane jest z przypadkiem użycia „Przeglądanie listy projektów”</p>				
Źródło:	Klient				
Powiązane wymagania:	Wymaganie jest nadrzędne w stosunku do F2 – F9, które szczegółowo definiują zakres funkcjonalności systemu w dziedzinie wspomagania procesu zarządzania problemami.				

# Techniki specyfikowania

- Uproszczona reprezentacja (np. lista cech)
- Tekst naturalny
- Tekst ustrukturalizowany
- Notacje modelujące
- Specyfikacje formalne
- Inne techniki np. prototypy, testy

# Techniki specyfikowania – uproszczona reprezentacja wymagań

- **User stories** - „jako *[rola]* chciałbym, żeby *[usługa systemu]* w celu *[powód]*”
  - jako zarejestrowany użytkownik chcę móc się zalogować, żeby korzystać z zastrzeżonych funkcji
  - jako sprzedawca chcę móc przeglądać złożone zamówienia, żeby móc je weryfikować i zatwierdzać
- **Feature list** – krótkie określenia funkcjonalnych i pozafunkcjonalnych cech systemu
  - przeglądanie listy zamówień
  - odporność na błędy przy składaniu zamówienia
- **Use Case briefs** – diagram przypadków użycia plus krótkie jednozdaniowe opisy poszczególnych przypadków
- **To tylko „zarys”** – dalej trzeba albo udokumentować dokładniej innymi środkami albo polegać na intensywnej komunikacji!

## Opis za pomocą tekstu

- **Tekst naturalny** – po prostu opis słowny, może to nawet być dokładny zapis wypowiedzi interesariusza
- **Tekst ustrukturalizowany** – wykorzystanie pewnej predefiniowanej struktury pozwalającej na wyróżnienie określonych zagadnień np.

Wymaganie wywodzi się bezpośrednio z celów biznesowych systemu. Wytworzenie i wdrożenie systemu ma na celu wspomaganie procesu zarządzania problemami przez gromadzenie danych na temat projektów i napotkanych w ich trakcie problemów. System musi udostępniać funkcjonalność w zakresie dodawania, edycji oraz usuwania danych istotnych z punktu widzenia tego procesu.

Kryterium akceptacji

Pomyślne przejście testów walidacyjnych odnoszących się do kompletności funkcjonalnej systemu w myśl wymagań specyfikujących zakres funkcjonalności.

Powiązania z celami biznesowymi

Wymaganie to jest kluczowe do wypełnienia wszystkich celów biznesowych systemu.

Powiązanie z przypadkami użycia systemu

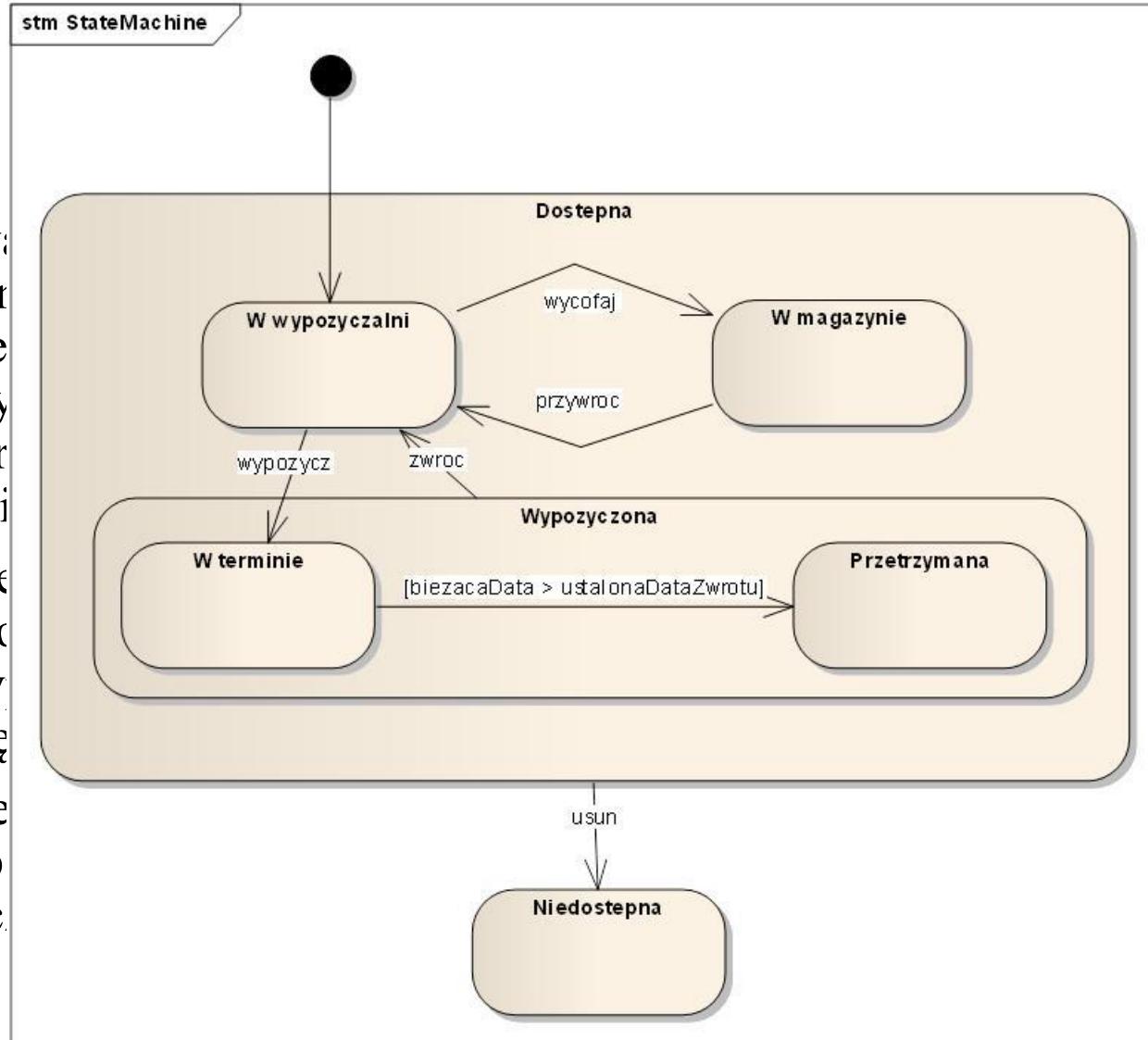
Wymaganie powiązane jest z przypadkiem użycia „Przeglądanie listy projektów”

# Model czy opis?

- Każda nowa kopia filmu włączana do zasobów „Video++” i rejestrowana w systemie inicjalnie znajduje się w wypożyczalni. Kopia może zostać wypożyczona przez klienta. Kiedy kopia jest wypożyczona, można wyróżnić jej dwa stany: początkowo jest wypożyczona zgodnie z umową, bez przekroczenia terminu zwrotu, jeśli jednak ustalony termin zwrotu minie, kopia jest uznawana za przetrzymaną. Gdy klient zwróci kopię, trafia ona znowu do wypożyczalni.
- Kopia może zostać wycofana z wypożyczalni do magazynu (np. w celu sprawdzenia informacji o jej złym stanie technicznym albo po prostu żeby zwolnić miejsce na półce w wypożyczalni). Z magazynu kopia może wrócić do wypożyczalni. Gdy kopia znajduje się w magazynie nie można jej bezpośrednio wypożyczać.
- Niezależnie od stanu, w którym aktualnie znajduje się kopia, może ona zostać usunięta np. w przypadku stwierdzenia zniszczenia (co może wydarzyć się zarówno w wypożyczalni, magazynie i u klienta).

Przykład z wypożyczalni filmów

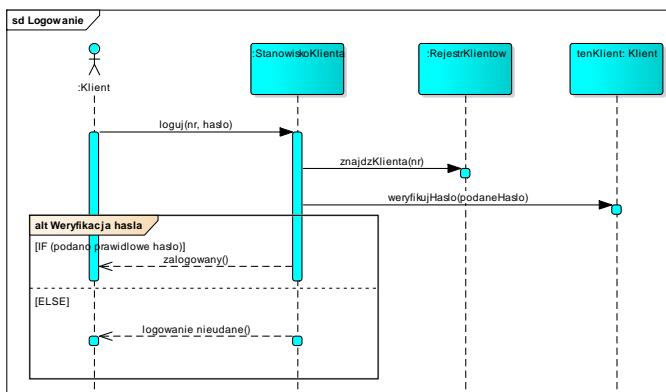
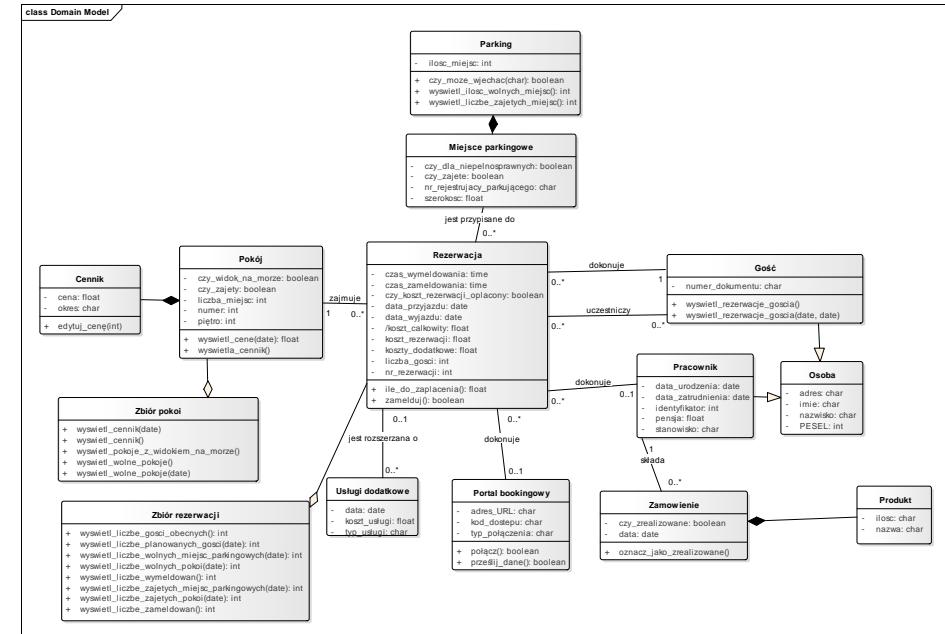
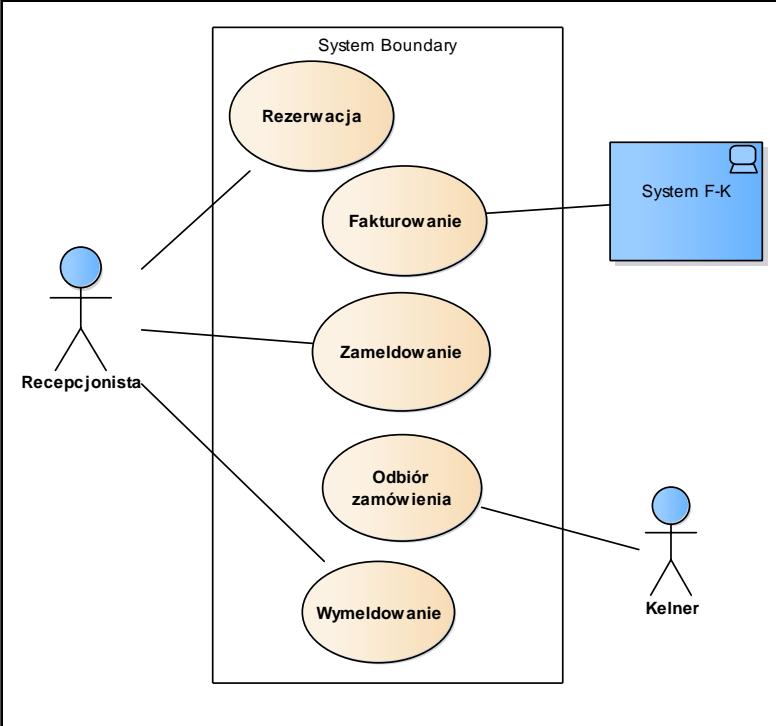
- Każda nowa kopię inicjalnie znajduje się w magazynie klienta. Kiedy klient ją wypożyczy, ustalony termin zwrotu i kopia zwróci kopię do magazynu.
- Kopia może być informacją na półce wypożyczalni, jeśli znajduje się tam.
- Niezależnie od tego, whether usunięta np. z półki, kopia wypożyczalni jest dostępna.



Przykład z wypożyczalni filmów

ana w systemie  
ona przez  
: początkowo  
u, jeśli jednak  
jdy klient  
  
u sprawdzenia  
miejsce na  
alni. Gdy kopia  
  
a zostać  
ć się zarówno

# Techniki specyfikowania - UML



Inne notacje, np. ERD, SysML, BPMN, ...

# Techniki specyfikowania – specyfikacje formalne

- **Tzw. metody formalne w inżynierii oprogramowania**
  - Silnie oparte o pewien aparat matematyczny i „theoretical computer science fundamentals”
  - Opis oprogramowania/systemu na bazie tego aparatu i dalej możliwość analizy tego opisu względem pewnych (również formalnie zdefiniowanych) własności)

*BoxOffice*

*seating :*  $\mathbb{P} \text{Seat}$ ; *sold :*  $\text{Seat} \leftrightarrow \text{Customer}$   
*dom sold*  $\subseteq$  *seating*

*Purchase*<sub>0</sub>

*BoxOffice*  
*BoxOffice'*  
*s? : Seat*  
*c? : Customer*

*s?*  $\in$  *seating \ dom sold*  
*sold' = sold*  $\cup$  {*s?*  $\mapsto$  *c?*}  
*seating' = seating*

*AnonymousReturn*<sub>0</sub>

*ΔBoxOffice*  
*s? : Seat*  
*r! : Response*

*s?*  $\in$  *dom sold*  
*sold' = {s?}*  $\lhd$  *sold*  
*seating' = seating*

Przykład: Notacja Z (oparta na zbiorach i relacjach)

# Techniki specyfikowania - prototypy

The screenshot shows the homepage of the 'Inteligentna Księgą Kucharska' website. At the top, there's a navigation bar with links for 'Plik', 'Edytuj', 'Widok', 'Pomoc', 'Twoja waluta: PL', 'Zaloguj', 'Zarejestruj', 'EN', and 'PL'. The main content area features a search bar with the placeholder 'wyszukaj' and a dropdown menu showing filters for 'Mleko', 'Mąka', and 'Cukier'. Below this is a list of recently selected ingredients: 'jajka wołowina mleko mąka cukier chleb'. To the right, a large table displays search results for 'Italian Buttercream', including details like price (37 zł), dish type (Breakfast), availability of ingredients (checked for all), and a rating (4.7 / 5). Other results listed include Marco Botton, Tuttofare, Mariah MacLachlan, Better Half, Valerie Liberty, Head Chef, and Guido Jack. A sidebar on the left offers account creation ('Moje konto') and a note about available features ('Nie masz jeszcze konta? Sprawdź jakie możliwości daje Ci Inteligentna Księgą Kucharska'). At the bottom, social media links ('Znajdz nas') and a footer note ('Powered by @ Team') are visible.

Źródło: Praca inż . K.Majcher, K.Mossakowska, M.Tomczewski, M.Zych (2013)

## Techniki specyfikowania - testy akceptacyjne

<b>ID</b>	T0014
<b>Description</b>	Checking accounts have an overdraft limit of \$500. As long as there are sufficient funds (e.g. -\$500 or greater) within a checking account after a withdrawal has been made the withdrawal will be allowed.
<b>Setup</b>	1.Create account 12345 with an initial balance of \$50 2.Create account 67890 with an initial balance of \$0
<b>Instructions</b>	1.Withdraw \$200 from account #12345 2.Withdraw \$350 from account #67890 3.Deposit \$100 into account #12345 4.Withdraw \$200 from account #67890 5.Withdraw \$150 from account #67890 6.Withdraw \$200 from account #12345 7.Deposit \$50 into account #67890 8.Withdraw \$100 from account #67890
<b>Expected Results</b>	Account #12345: <ul style="list-style-type: none"> <li>• Ending balance = -\$250</li> <li>• \$200 Withdrawal transaction posted against it</li> <li>• \$100 Deposit transaction posted against it</li> <li>• \$200 Withdrawal transaction posted against it</li> </ul> Account #67890: <ul style="list-style-type: none"> <li>• Ending balance = -\$500</li> <li>• \$350 Withdrawal transaction posted against it</li> <li>• \$150 Withdrawal transaction posted against it</li> <li>• \$50 Deposit transaction posted against it</li> </ul> Errors logged: <ul style="list-style-type: none"> <li>• Insufficient funds in Account #67890 (balance -\$350) for \$200 Withdrawal</li> <li>• Insufficient funds in Account #67890 (balance -\$450) for \$100 Withdrawal</li> </ul>

- **Wymagania można również wyrażać za pomocą przypadków i scenariuszy testowych**
- **Poza samą specyfikacją wymagań mogą również zostać wykorzystane w fazie testowania (zwłaszcza po automatyzacji)**

# Dokument SWS i jego zawartość

- **SWS – ustrukturalizowany zbiór informacji obejmujących całokształt wymagań względem systemu**
- **Wymaganie te mogą mieć różne formy (tekst, diagramy itp.)**
- **Niektóre standardy czy podejścia sugerują osobne dokumenty wymagań dot.:**
  - Systemu jako całości
  - Oprogramowania w ramach systemu
  - Sprzętu i innych elementów poza oprogramowaniem

Przykład: Zawartość SWS wg standardu IEEE 29148:

<b>1. Introduction</b>
1.1 System purpose
1.2 System scope
1.3 System overview
1.3.1 System context
1.3.2 System functions
1.3.3 User characteristics
1.4 Definitions
<b>2. References</b>
<b>3. System requirements</b>
3.1 Functional requirements
3.2 Usability requirements
3.3 Performance requirements
3.4 System interface
3.5 System operations
3.6 System modes and states
3.7 Physical characteristics
3.8 Environmental conditions
3.9 System security
3.10 Information management
3.11 Policies and regulations
3.12 System life cycle sustainment
3.13 Packaging, handling, shipping and transportation
<b>4. Verification</b>
(parallel to subsections in Section 3)
<b>5. Appendices</b>
Assumptions and dependencies
Acronyms and abbreviations

# Walidacja wymagań

- Pojęcia weryfikacji i walidacji
  - **Weryfikacja:** czy robimy system dobrze względem jakichś kryteriów lub punktu odniesienia (np. dokument czy model będący reprezentacją systemu)
  - **Walidacja:** czy robimy dobry system czyli taki, który odpowiada potrzebom interesariuszy
- Przedstawienie wymagań interesariuszom w celu ich potwierdzenia i upewnienia się, że dalej na podstawie tych wymagań można przeprowadzić następny krok (etap projektu, iterację projektu, itp.)

# Podstawowa walidacja wymagań

- **Polega na przedstawieniu udokumentowanych wymagań interesariuszowi, który je zgłosił, w celu ich sprawdzenia**
  - *Czy to jest poprawny zapis tego co powiedziałeś?*
  - *Czy to odzwierciedla to co miałoś na myśli?*
  - *Czy moja interpretacja jest właściwa?*
- **Daje to możliwość dokonania korekty - tak, aby usunąć wątpliwości czy wymagania zostały jasno zrozumiane**
- **Warto też w takim przypadku skorzystać z pomocy innego reprezentanta tego samego interesariusza (np. innego kasjera, innego kierownika zmiany), co pozwoli ograniczyć „skrywienie” wynikające z subiektywnego punktu widzenia pojedynczego reprezentanta**
- **W dalszej kolejności warto pokazać innym interesariuszom**

# Techniki walidacji

- **Przeglądy i inspekcje**
  - Reprezentacje wymagań (tekstowe, diagramy) czytane przez ludzi pod kątem znajdowania w nich błędów lub ogólniej rzeczy nieodpowiadających potrzebom interesariuszy
- **Praca z prototypami**
  - Prototypowanie to bardzo uniwersalna technika – służy zarówno do wydobywania wymagań, ich specyfikowania, jak i walidacji. Interesariusz zapoznaje się z prototypem i na tej podstawie potwierdza, że system będzie odpowiadał jego potrzebom (lub nie ☺)
- **Uzgadnianie kryteriów akceptacji**
  - Określenie, w jaki sposób będzie można sprawdzić spełnienie wymagań

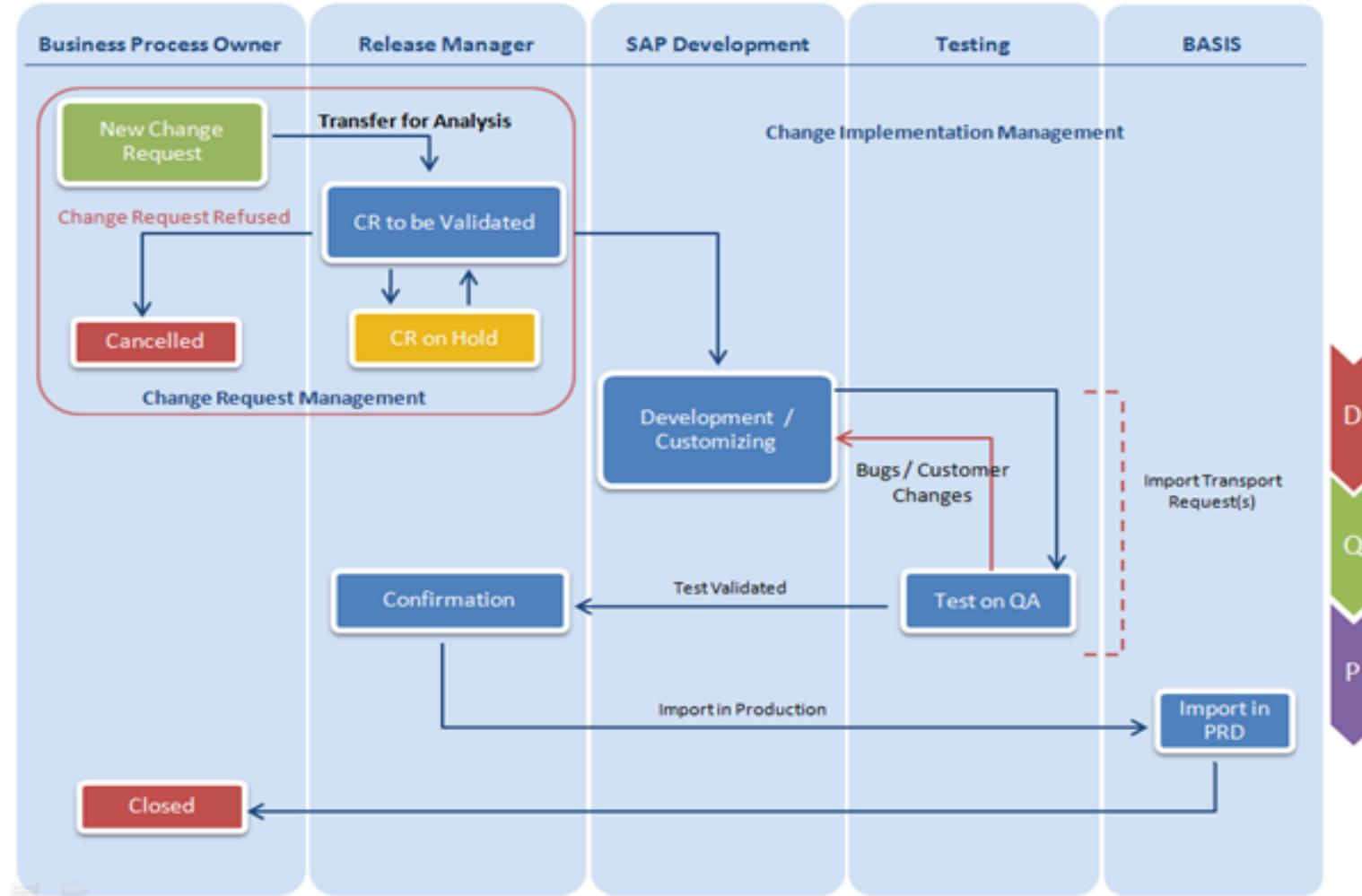
# Zarządzanie wymaganiami

- Wymagania nie istnieją każde osobno, lecz występują między nimi zależności (pole: **wymagania powiązane**)
- Wymagania są pewną wczesną reprezentacją systemu, elementy kolejnych reprezentacji (projektu, kodu, testów) będą wynikały z wymagań (istotna jest więc tzw. **śladowość wymagań** – ang. *traceability*)
- Dobra inżynieria wymagań ograniczy liczbę **błędów** w wymaganiach, ale raczej nie zmniejszy jej do zera – trzeba się liczyć z koniecznością poprawek
- Wymagania mogą podlegać **zmianom** w dalszych fazach trwania przedsięwzięcia np. z uwagi na zmianę sytuacji rynkowej, oferty, procedur itp.

# Zakres zarządzania wymaganiami

- Zarządzanie procesami (wydobywania, analizy, specyfikowania, walidacji)
- Zarządzanie zmianami
- Zarządzanie śladami

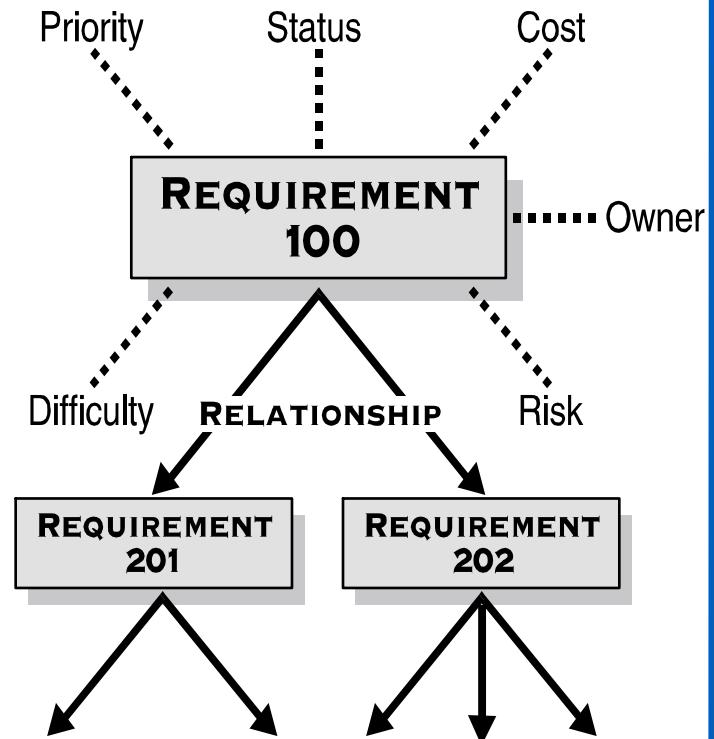
# Zarządzanie zmianą



SAP change request lifecycle

# Ślady wymagań

- Wskazują uzasadnienia wymagań
- Wyróżniają grupy wymagań wzajemnie powiązanych
- Wskazują, które elementy systemu są odpowiedzialne za wypełnienie danego wymagania
- Wspomagają zarządzanie projektem
  - Pozwalają oszacować zaawansowanie prac (np. sprawdzić, które wymagania są już odwzorowane w aktualnym stanie implementacji systemu, a które jeszcze nie)



# Analityk

- **Różne tytuły/nazwy stanowisk, niekoniecznie dokładnie precyzujące zakres obowiązków:**
  - analityk biznesowy, analityk systemowy, analityk IT, inżynier wymagań, analityk wymagań, kierownik produktu, architekt wymagań, ...
- **Pożądane kompetencje:**
  - Myślenie analityczne
  - Ukierunkowanie na rozwiązywanie problemów
  - Decyzyjność
  - Odpowiedzialność i zorganizowanie
  - Umiejętność nauki i adaptacji do zmian
  - Znajomość aspektów biznesowych
  - Umiejętności komunikacyjne (komunikacja werbalna, pozawerbalna, pisemna)
  - Umiejętność pracy z ludźmi
  - Znajomość technologii (niekoniecznie szczegółowa, ale orientacja co do możliwości)

# Certyfikacja

- **International Institute of Business Analysis (IIBA)**
  - Entry Certificate in Business Analysis (ECBA)
  - Certification of Competency in Business Analysis (CCBA)
  - Certified Business Analysis Professional (CBAP)
  - Specjalizowane np. Agile Analysis Certification
- **International Requirements Engineering Board (IREB)**
  - Certified Professional for Requirements Engineering (Foundation level, Advanced level, Expert level)
- **Project Management Institute (PMI)**
  - PMI Professional in Business Analysis

# Zmienna w dziedzinie analizy

- **Na pewno jest to dziedzina mniej zmienna, niż obszar technologii, wiedza/umiejętności nie dezaktualizują się tak szybko**
- **Co się zmieniło/zmienia?**
  - Wzrost znaczenia aspektów biznesowych w stosunku do systemowych
  - Rozpowszechnienie zwinnych (agile) metodyk wytwarzania oprogramowania i pojawienie się specyficznych technik inżynierii wymagań z nimi związanych
  - Rozwój narzędzi wspomagających zarządzanie wymaganiami, prototypowanie, modelowanie, ...
- **Co się nie zmienia?**
  - Trudność kompleksowego zebrania i wyrażenia wymagań (złożoność problemu)
  - Problematyczność komunikacji międzyludzkiej
  - To, że wymagania zmieniają się w trakcie projektu

# Projektowanie

*Aleksander Jarzębowicz*

*Katedra Inżynierii Oprogramowania  
Wydział ETI, Politechnika Gdańsk*

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich  
rozprowadzanie jest zabronione.

# Co to jest projekt?

- **Ang. - Design**
- **Proces definiowania architektury systemu oraz jej elementów (m.in. moduły, komponenty, interfejsy, dane) w taki sposób, aby spełnione zostały cele i wymagania postawione przed tym systemem**
- **Często projektem nazywa się również rezultat tego procesu wyrażony np. w postaci modeli UML czy AADL dokumentujących przyjęte rozwiązania**

# Analiza a projekt (1)

- Relację między analizą a projektowaniem najogólniej można podsumować następująco:
  - Analiza ma odpowiedzieć na pytanie: **co?**
  - Projektowanie ma odpowiedzieć na pytanie: **jak?**
- W ramach analizy mają miejsce:
  - działania ukierunkowane na maksymalizację korzyści biznesowych klienta
  - identyfikacja i dokumentacja wymagań względem systemu
  - jeśli opieramy się na modelowaniu - konstrukcja modelu logicznego specyfikującego wymagania dot. oprogramowania
- W ramach projektowania mają miejsce:
  - decyzje co do docelowej architektury systemu
  - uwzględnienie rozwiązań technicznych dot. software'u i hardware'u
  - uszczegółowienie modelu logicznego z etapu analizy, przekształcenie go w kierunku docelowej implementacji

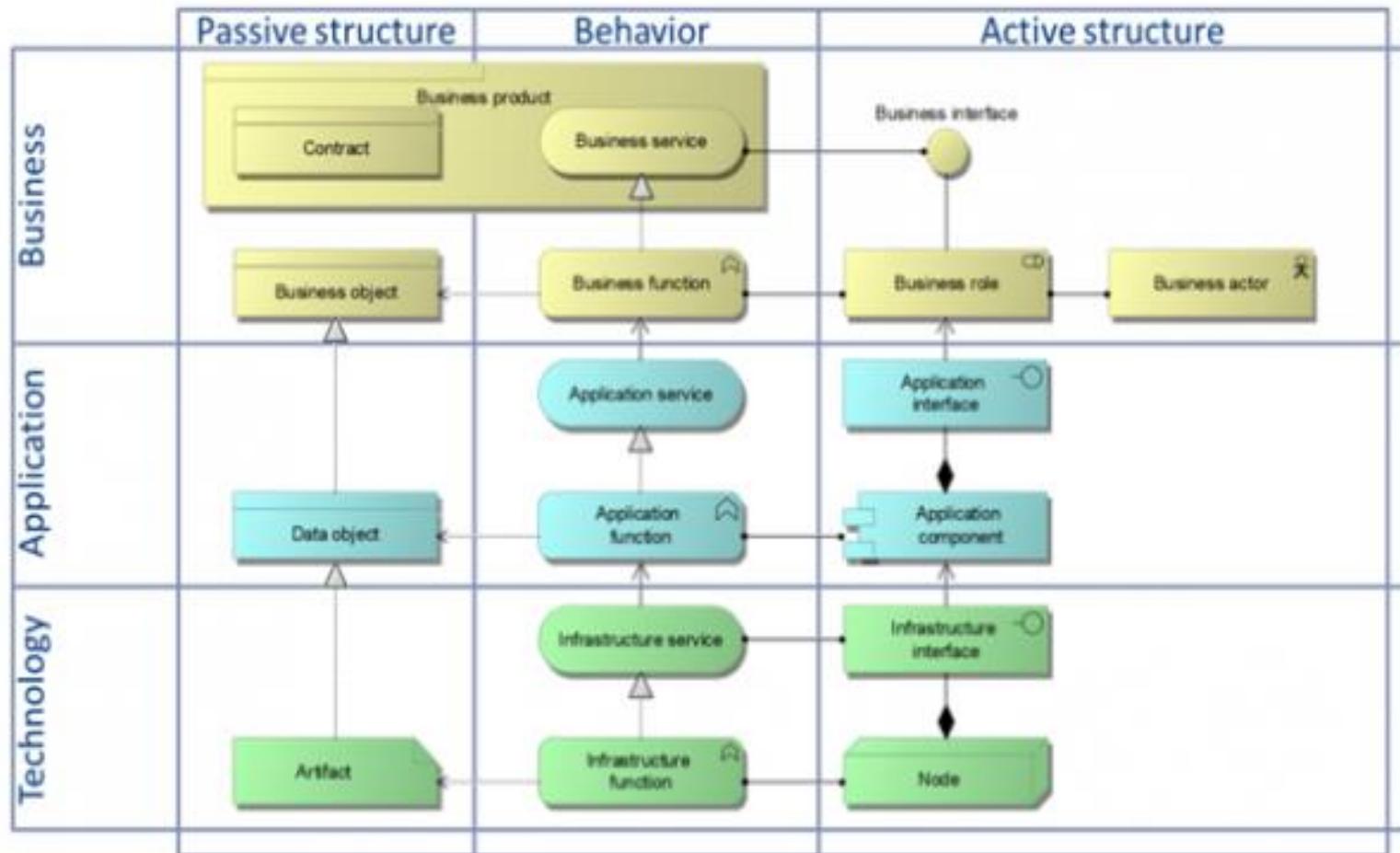
## Analiza a projekt (2)

- Trudno wyodrębnić ścisłą granicę między tymi etapami
- Przykładowe kwestie:
  - Wymagania co do systemu mogą już w pewnym stopniu decydować o architekturze i możliwych technologiach np. jeśli z systemu mają korzystać klienci firmy, to implikuje to system dostępny poprzez Internet, zapewne przez przeglądarkę lub aplikację mobilną
  - Pozyskane od interesariuszy wymagania mogą wręcz wymieniać określone rozwiązania sprzętowe czy software'owe, które mają być wykorzystane w tworzonym systemie np. ograniczenie dotyczące tego, jaki DBMS powinien być wykorzystany
- Idea jest jednak taka, żeby w ramach analizy jak najlepiej zrozumieć rozważany problem i związane z nim ograniczenia, aby przy projektowaniu móc podejmować decyzje co do alternatywnych sposobów budowy systemu

# Architektura Korporacyjna (1)

- **Podejście, które integruje działania z zakresu analizy (biznesowej i systemowej) oraz projektowania systemów**
- **Architektura Korporacyjna (ang. *Enterprise Architecture*) – kompleksowe podejście do zarządzania rozwojem IT w organizacji (korporacji)**
- **Wiele różnych poziomów – od procesów biznesowych, poprzez usługi świadczone przez systemy, aż po rozwiązania techniczne i infrastrukturę**
- **Ścisłe odwzorowania między tymi poziomami**
- **Różne aspekty – procesy, informacje, struktura**

# Architektura Korporacyjna (2)



W dalszej części wykładu skupimy się jednak na projektowaniu czyli tych „niższych” poziomach Architektury Korporacyjnej

# Projekt - poziomy

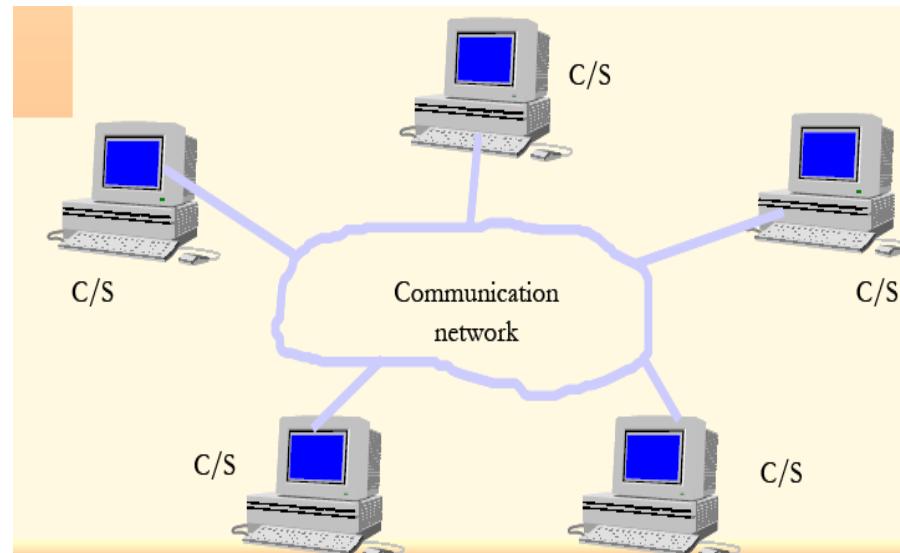
- **Projekt systemu (inne nazwy: p. architektury, p. wysokiego poziomu)**
  - Opisuje system w kategoriach jego części składowych (modułów)
- **Projekt klas (inne nazwy: p. szczegółowy, p. niskiego poziomu)**
  - Opisuje wewnętrzne zależności i funkcjonowanie poszczególnych części składowych (modułów)

# Projekt systemu – zakres czynności

## Czynności do wykonania:

- **Określenie architektury - wewnętrznej organizacji systemu na bazie jego składowych**
- **Określenie podsystemów i ich interfejsów**
- **Decyzje o fizycznym rozmieszczeniu podsystemów (topologia systemu)**
- **Identyfikacja i obsługa dostępu do zasobów globalnych**
- **Obsługa stanów przejściowych i awaryjnych**
- **Wybranie podejścia dla zarządzania pojemnikami danych**
- **Decyzje co do stylu projektowania i wypracowanie rozwiązań kompromisowych**

# Architektura



Źródło: Maciaszek, RA&SD 2ed.

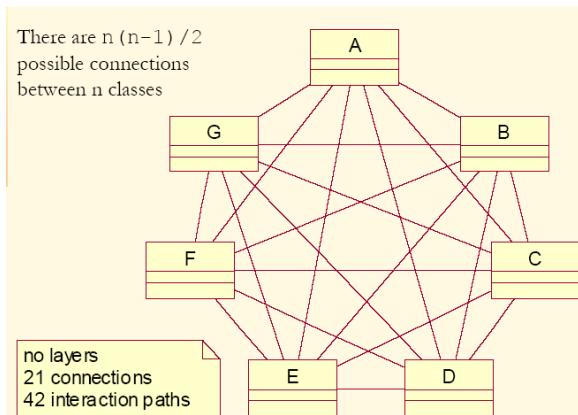
- Wyodrębnienie warstw systemu
- Podział na podsystemy
- Określenie interfejsów pomiędzy warstwami i pomiędzy podsystemami

# Analiza systemowa a projekt architektury

- Decyzje co do architektury powinny wynikać z wymagań funkcjonalnych i pozafunkcjonalnych
- Funkcjonalne np.
  - „System ma umożliwiać klientom przeglądanie produktów i składanie zamówień” – sugeruje aplikację webową dostępną przez przeglądarkę
- Pozafunkcjonalne np.
  - **Dostępność** – sugeruje użycie redundancji, nadmiarowych komponentów sprzętowych/software'owych
  - **Wydajność** – przesłanka do zaimplementowania krytycznych operacji w małej liczbie komponentów z jak najbardziej ograniczoną komunikacją między nimi (mniejsza liczba większych komponentów)
  - **Elastyczność** – sugeruje użycie niewielkich, dobrze wyodrębnionych komponentów, które będzie można niezależnie wymieniać/modyfikować
  - **Często to wybór między sprzecznymi opcjami!**
- Dodatkowe elementy wbudowywane w architekturę
  - Np. dla poprawy wydajności można m.in. utrzymywać kilka kopii danych lub obliczeń; zmniejszać narzut obliczeniowego (poprzez usunięcie obiektów pośredniczących); ...

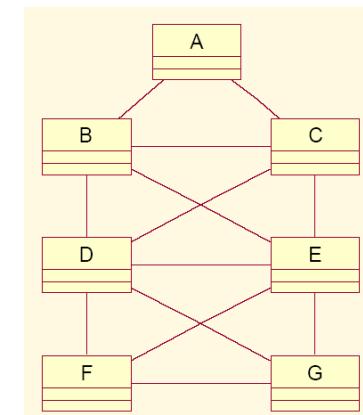
# Warstwy systemu

- Obiekty systemu nie powinny komunikować się w sposób dowolny (każdy z każdym) – utrudnia to utrzymanie systemu i przyczynia się do błędów
- Współczesne systemy zorganizowane są w struktury hierarchiczne składające się z warstw (ang. *layers/tiers*)
- Komunikacja dozwolona jest tylko pomiędzy obiektami tej samej warstwy lub sąsiednich warstw
- Dodatkową kontrolę stanowi fakt że, komunikacja odbywa się poprzez zdefiniowane interfejsy



Dowolna komunikacja

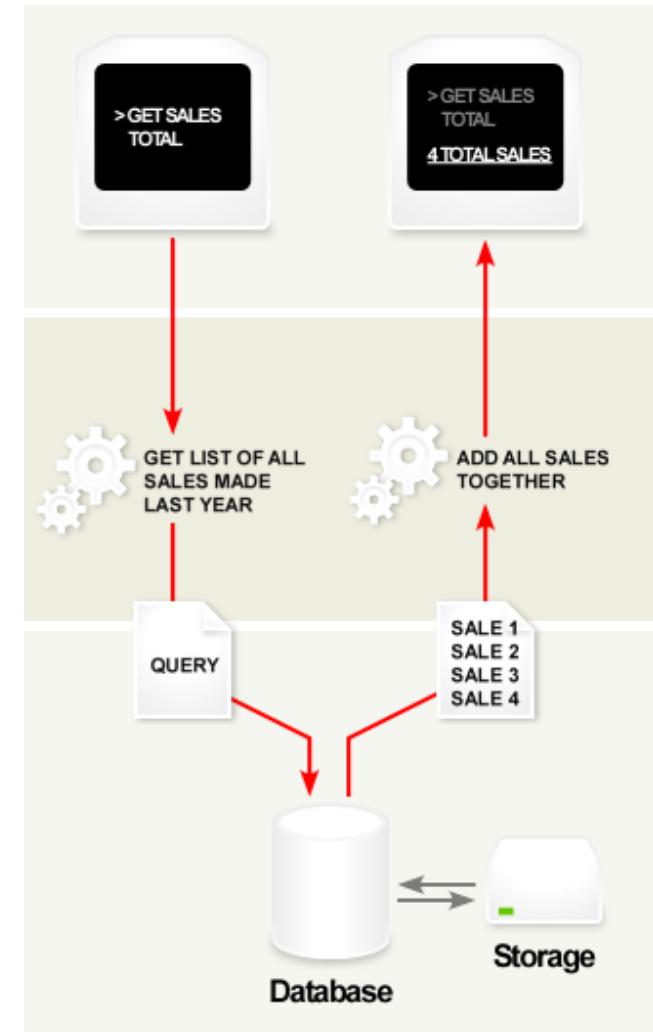
Wyodrębnienie warstw



Źródło: Maciaszek, RA&SD 2ed.

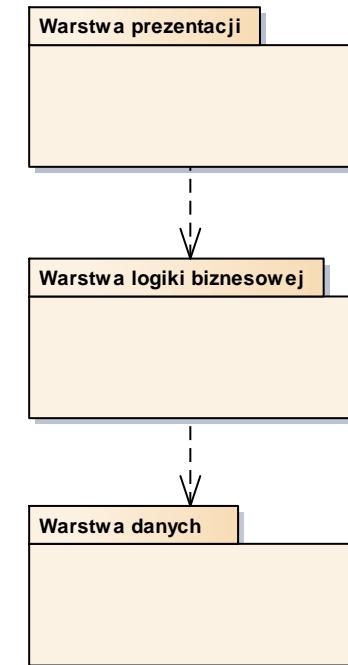
# Model trójwarstwowy

- Przy podziale na warstwy obiekty grupuje się ze względu na rodzaj wykonywanych zadań np. prezentacja informacji użytkownikowi, odczyt informacji z pojemników danych itp.
- Warstwy traktowane są wówczas jako odrębne moduły, często wytwarzane w zróżnicowanych technologiach
- Chyba najbardziej znanym ogólnym modelem podziału jest model trójwarstwowy wyróżniający:
  - Warstwę prezentacji
  - Warstwę logiki biznesowej
  - Warstwę danych



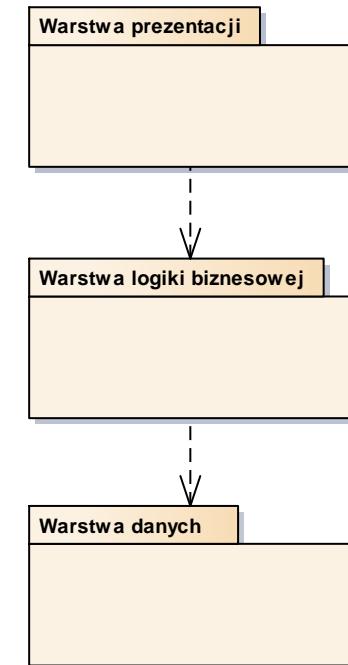
# Warstwa prezentacji

- **Najwyższa (najbliższa użytkownikowi) warstwa systemu**
- **Odpowiada za interakcję użytkownika z systemem; odbiera polecenia użytkownika i przekazuje mu wyniki działań oraz wszelkie inne informacje**
- **Korzysta z usług warstwy logiki biznesowej, jest więc od niej zależna, natomiast warstwa logiki biznesowej nie powinna być zależna od niej**
- **Zależność (*dependency*) jest rozumiana jako korzystanie z usług i co za tym idzie fakt, że usługobiorca musi coś wiedzieć o usługodawcy**



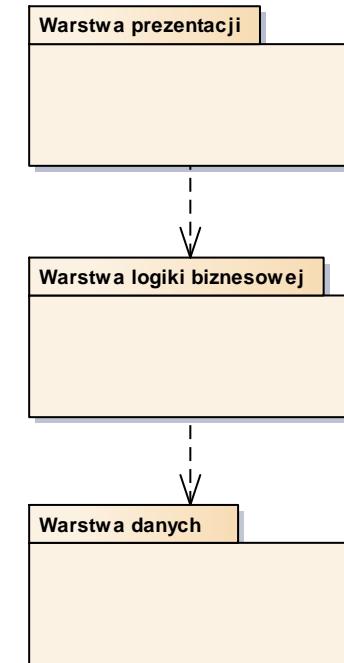
# Warstwa logiki biznesowej

- **Jest to warstwa, w której odbywa się główne przetwarzanie informacji oraz operacje typu: porównywania danych, obliczenia, decyzje**
- **Warstwa logiki biznesowej dostarcza na żądanie usługi dla warstwy prezentacji**
- **Warstwa logiki biznesowej korzysta z usług warstwy danych w zakresie pobierania potrzebnych danych i zapisywania wyników swoich działań**
- **Zależność również tutaj powinna być jednostronna**



# Warstwa danych

- **Odpowiada za przechowywanie i udostępnianie danych**
- **Zarządza pojemnikami danych np. bazą danych, plikami**
- **Na żądanie udostępnia dane wyższej warstwie (logiki biznesowej) oraz zapisuje dane przekazane od wyższej warstwy**

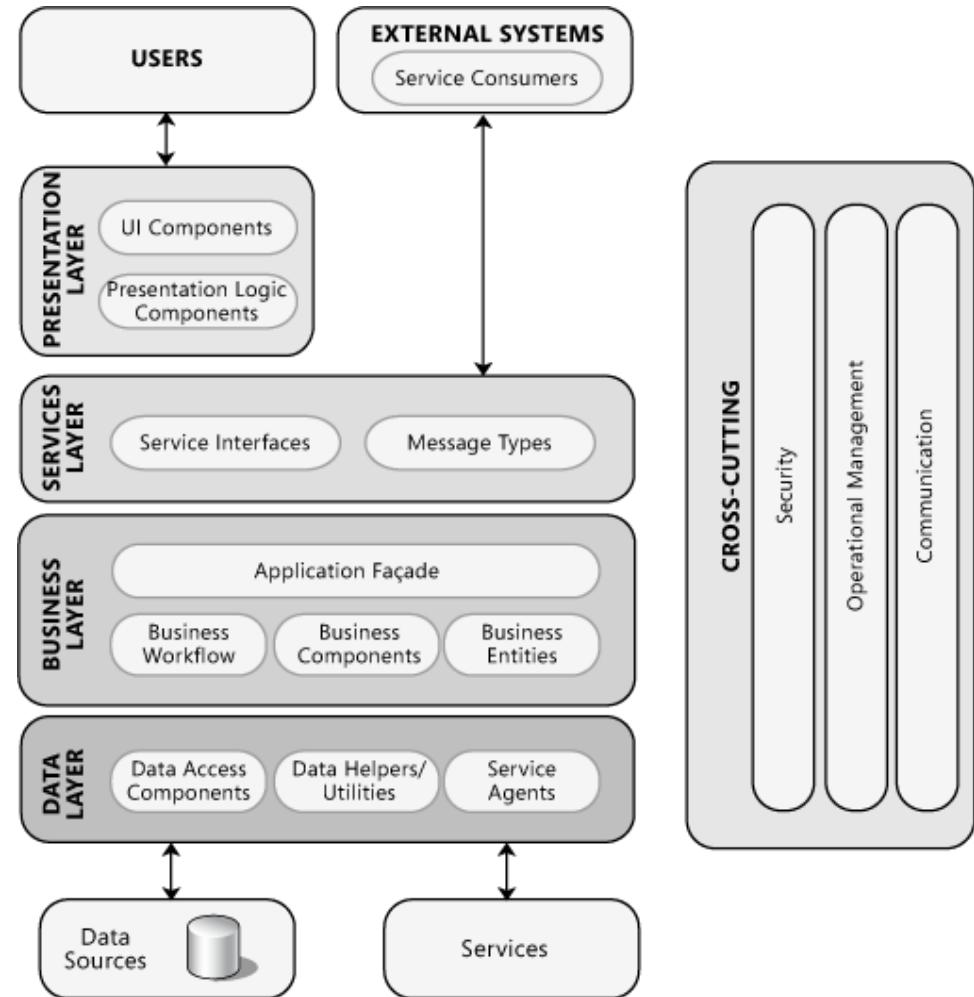


# Przełożenie klas modelu analitycznego na warstwy

- W ramach modelu analitycznego wyrażanego w UML powstał diagram klas. Zawiera on abstrakcje bytów ze świata rzeczywistego, które mają być reprezentowane w systemie
- Gdzie znajdują się te klasy przy dzieleniu systemu na warstwy?
- Nie ma prostej odpowiedzi:
  - To, co musi być trwale przechowywane w systemie znajdzie się w **warstwie danych**
  - To, co wiąże się z decyzjami, obliczeniami, przetwarzaniem (operacje zdefiniowane dla klas!) znajdzie się w **warstwie logiki biznesowej**
  - To, co będzie bezpośrednio wizualizowane użytkownikowi znajdzie się w **warstwie prezentacji**
  - W ten sposób pojedyncza klasa z modelu analitycznego może być reprezentowana jako np. 2 albo 3 klasy w modelu projektowym

# Przykłady modeli warstwowych (1)

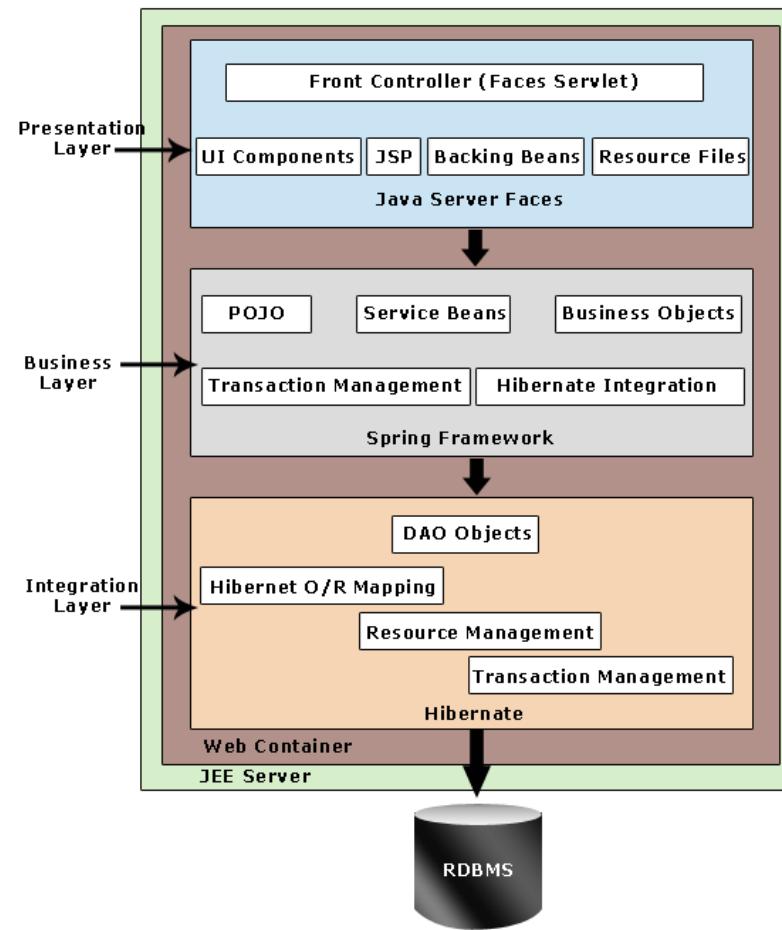
- Możliwe jest zdefiniowanie bardziej zaawansowanych modeli architektonicznych np. poprzez wyróżnienie dodatkowych grup obiektów jako osobnych warstw.
- Model trójwarstwowy plus dodatkowa warstwa „Services” między „Presentation” a „Business Logic”
- „Services” pośredniczy między dwoma ww. warstwami, ale przede wszystkim potrzebna jest do dostarczania usług zewn. systemom w architekturze zorientowanej na usługi



Źródło: Microsoft Application Architecture Guide, MSDN

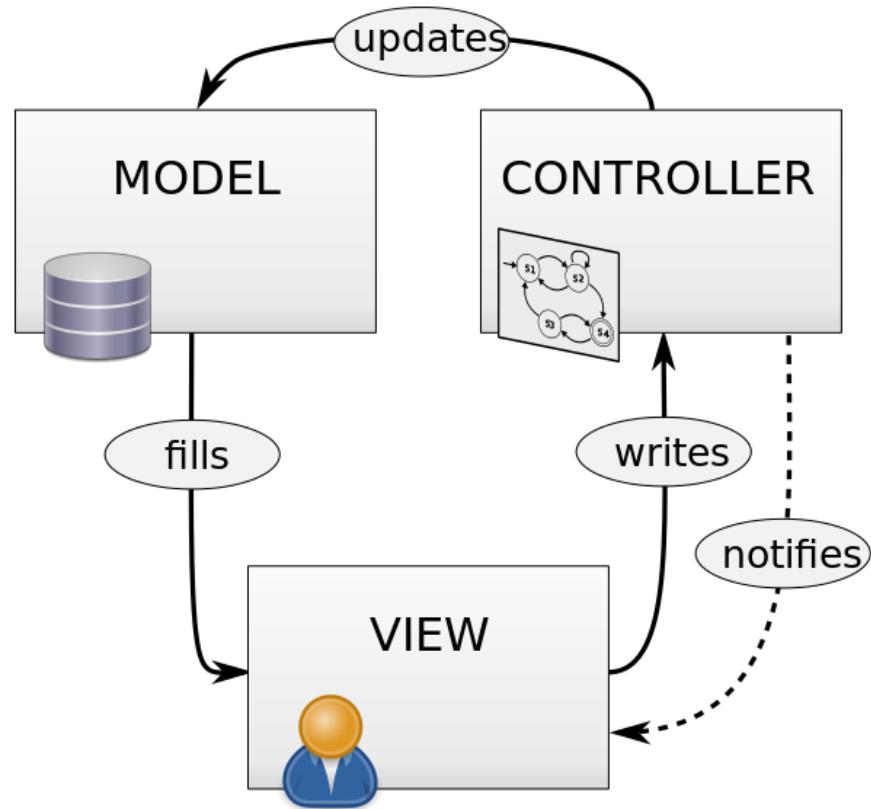
## Przykłady modeli warstwowych (2)

- Współczesne technologie bardzo często bazują na modelach warstwowych, a wręcz wymuszają ich stosowanie (zwł. frameworki)
- Podział systemu na warstwy jest wówczas w znacznie mniejszym stopniu odpowiedzialnością twórców systemu



# Model warstwowy a MVC

- **Model-View-Controller (MVC) – popularny model architektury systemów interaktywnych**
- **Nie jest to jednak architektura warstwowa (często błędnie tak się interpretuje)**
- **Nie jest zachowana zasada komunikacji jedynie z sąsiednią warstwą, tutaj komunikacja następuje „w trójkącie”**



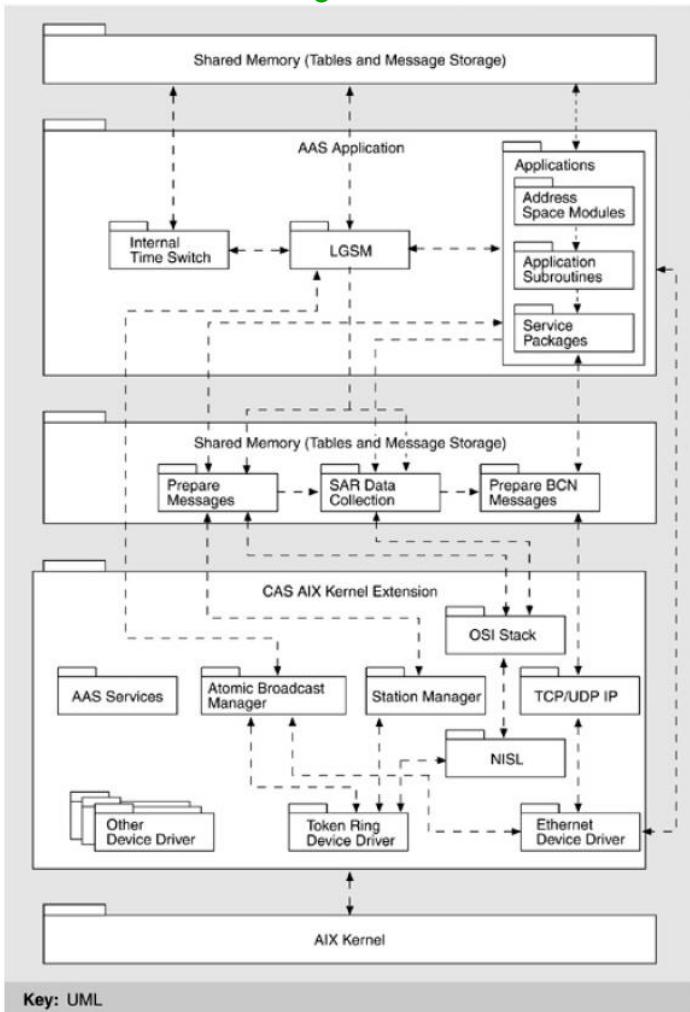
# Podział na podsystemy

- Oprócz podziału na **warstwy**, możliwy jest również podział na **podsystemy** realizujące wspólne zestawy usług funkcjonalnych np. dla sklepu mogą to być **podsystemy: zarządzania magazynem, obsługi zamówień, zarządzania pracą personelu itp.**
- Podsumowując, przy podziale na podsystemy stosowane są kryteria:
  - wspólny zestaw oferowanych usług
  - wspólne umiejscowienie
  - podobny sprzęt (platforma)

**UML – diagramy pakietów**

# Oczywistość architektury?

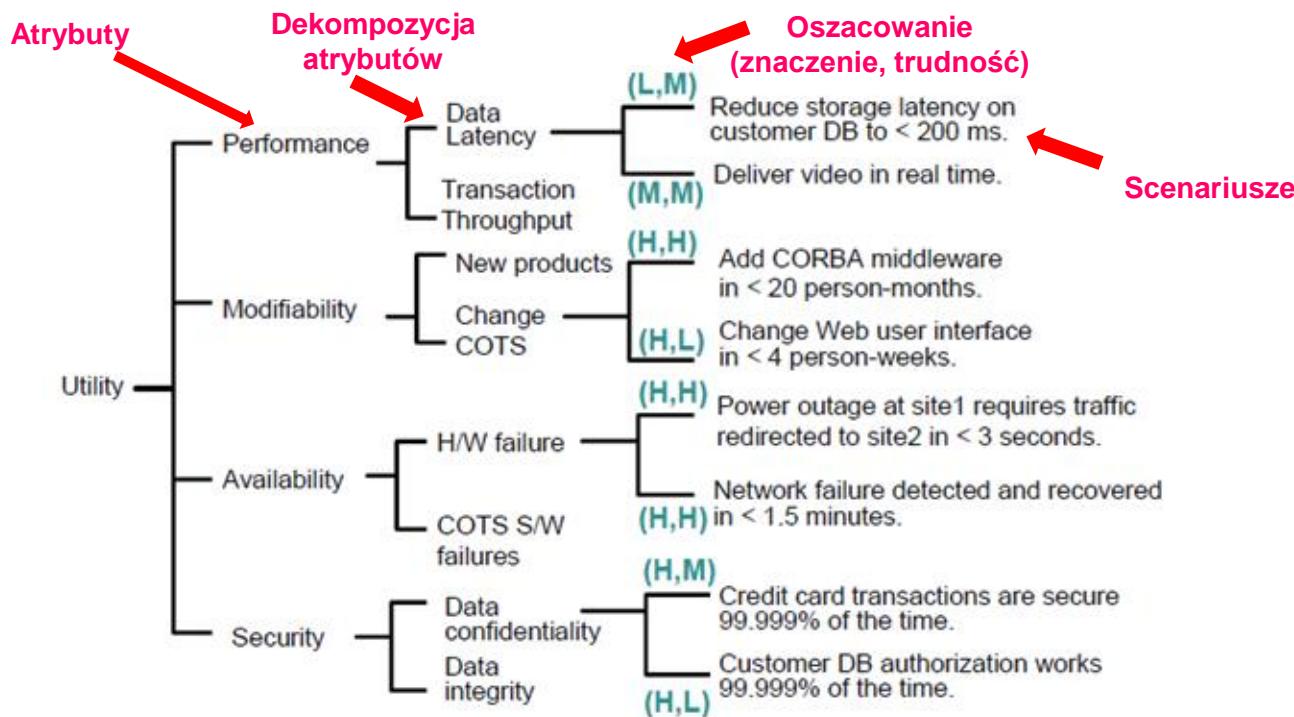
- Czasem może być „oczywista”
  - np. tworzę aplikację webową wykorzystując określony framework
- Czasem zdecydowanie nie, zwłaszcza przy:
  - systemach, gdzie oprogramowanie jest tylko częścią (obsługa dedykowanych urządzeń, komunikacja ze sprzętem)
  - wykorzystaniu zróżnicowanych technologii w ramach jednego systemu
  - systemach współpracujących z wieloma systemami zewn. lub integrujących istniejące systemy w jedną całość
  - nietypowych/rygorystycznych wymaganiach jakościowych (pozafunkcjonalnych)



ISSS System (Air Traffic Control)  
 Źródło: L. Bass, P. Clements, R. Kazman:  
 Software Architecture in Practice

# Ocena możliwych architektur

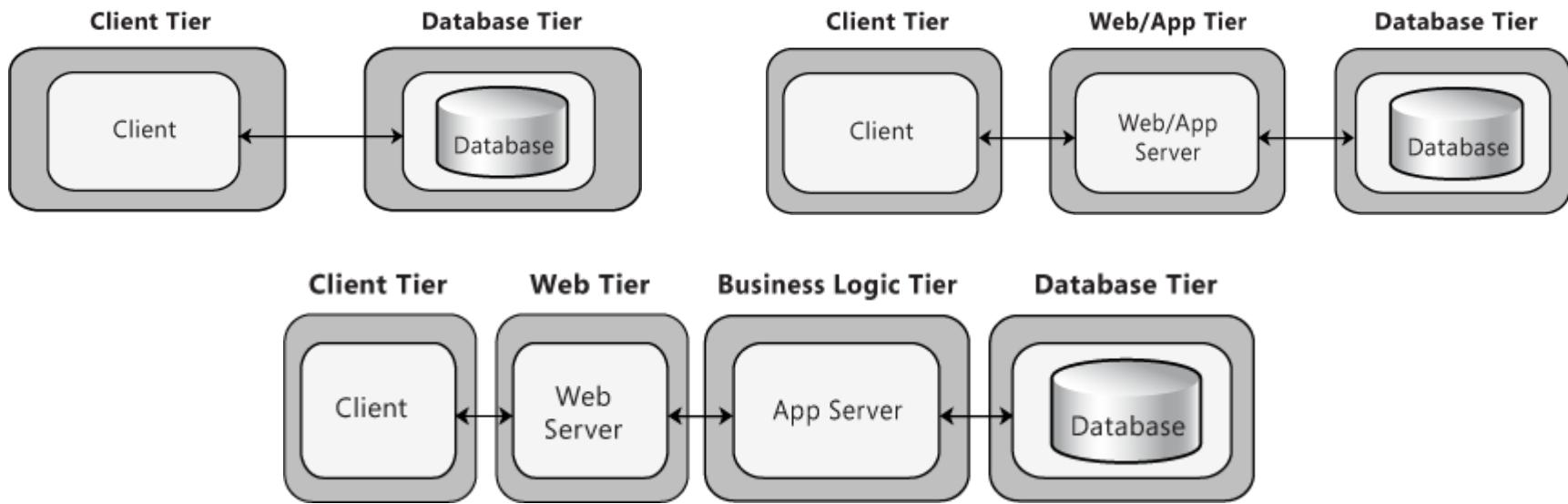
- Przykład: ATAM (Architecture Tradeoff Analysis Method)
- Metoda pracy zespołowej (architekci, interesariusze, osoby decyzyjne w projekcie)
- Wybory na podstawie identyfikacji kluczowych atrybutów jakościowych i powiązanych z nimi scenariuszy



Źródło: Architecture Tradeoff Analysis Method (CMU/SEI-2000-TR-004)

# Fizyczne rozmieszczenie podsystemów

- Fizyczne czyli na poszczególnych maszynach/procesorach
- Przykładowe warianty dla warstw:

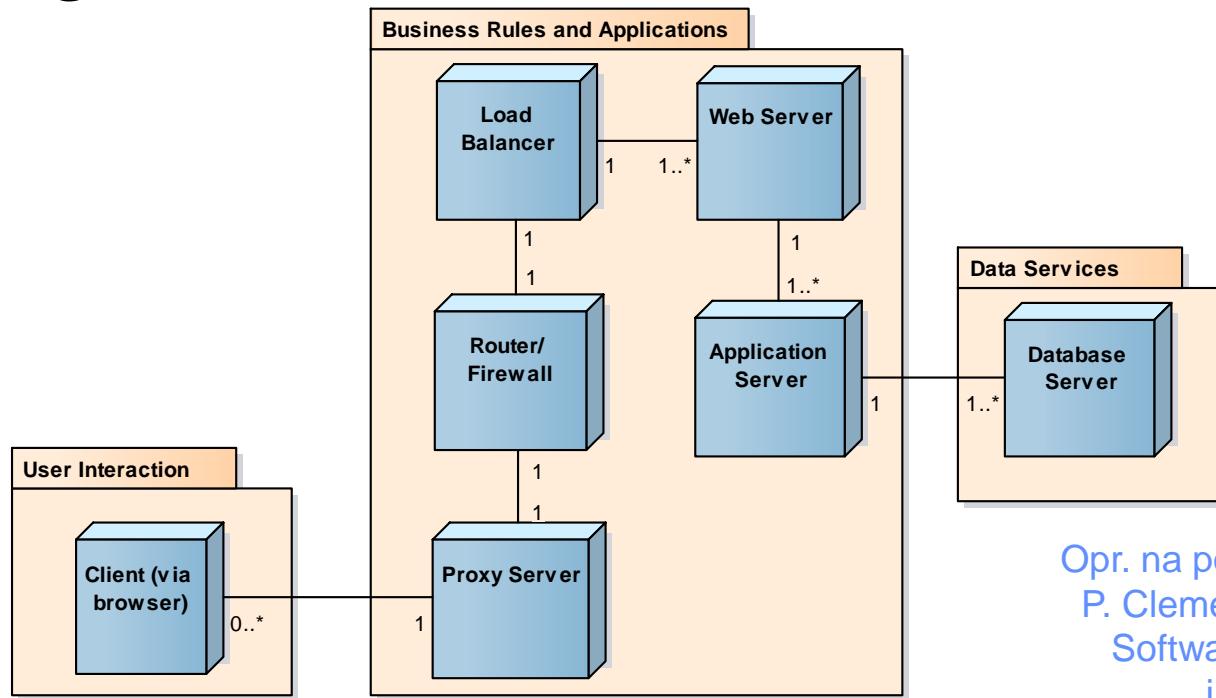


Źródło: Microsoft Application Architecture Guide, MSDN

- Analogicznie wygląda to dla podsystemów

# Architektura fizyczna

- Dodatkowo możliwe jest uwzględnienie innych niezbędnych elementów np. firewall
- Wykorzystanie diagramów wdrożenia (UML deployment diagrams)



Opr. na podstawie: L. Bass,  
P. Clements, R. Kazman:  
Software Architecture  
in Practice

# Obsługa zasobów globalnych

- **Identyfikacja zasobów globalnych:**

- jednostki fizyczne (procesory, napędy pamięci zewnętrznej, urządzenia komunikacyjne)
- dane (pliki, zawartość bazy danych, dane w pamięci)
- “przestrzeń działania” (miejsce na dysku, ekran stacji roboczej)
- nazwy logiczne (identyfikatory, nazwy plików, nazwy obiektów)
- ...

- **Metody synchronizacji dostępu do zasobów współużytkowanych:**

- Zastosowanie obiektu synchronizującego dostęp (guardian object) - obiekt taki działa we własnym wątku, a wszystkie operacje dostępu do zasobu są realizowane za jego pośrednictwem. Metoda bezpieczna, lecz nieefektywna czasowo.
- Wykorzystanie mechanizmów synchronizacyjnych systemu operacyjnego np. semaforów (locks) czy sygnałów. Metoda niebezpieczna (deadlock!), lecz efektywna czasowo.
- Wykorzystanie mechanizmów systemu zarządzania bazami danych. Metoda właściwa dla synchronizacji wielodostępu do danych.

# Obsługa warunków granicznych

- **INICJALIZACJA**

**Przejście od stanu spoczynku do ustabilizowanego stanu pracy**

- inicjowanie danych stałych, parametrów, zadań, kanałów komunikacyjnych,...

- **ZAKOŃCZENIE**

**Przejście od stanu pracy do stanu spoczynku**

- zwalnianie zasobów
- likwidacja obiektów nietrwałych
- systematyczne kończenie zadań zależnych od siebie

- **UPADEK (AWARIA)**

**Nieplanowane zakończenie pracy systemu**

- zarejestrowanie przyczyn i stanu systemu (jeśli możliwe)

# Pojemniki danych (1)

**Pojemniki danych mogą służyć jako interfejs pomiędzy podsystemami lub jako zasoby umożliwiające współużytkowanie informacji**

- **Różne sposoby realizacji pojemników danych - pliki zwykłe, pliki ustrukturalizowane, bazy danych (relacyjne, obiektowe, hybrydowe, no sql)**
- **Pojemniki danych stanowią wygodny środek pomagający w określaniu podziału na podsystemy**
- **Wybór rozwiązania musi uwzględniać: koszty, czas dostępu, pojemność, niezawodność, ...**

## Pojemniki danych (2)

PLIKI ZWYKŁE	RELACYJNE BAZY DANYCH
<ul style="list-style-type: none"><li>(+) niski koszt</li><li>(+) lepsza wydajność</li><li>(+) łatwość dostępu (odczyt/zapis)</li><li>(+) łatwość migracji (np. do chmury)</li></ul>	<ul style="list-style-type: none"><li>(+) dbałość o spójność danych</li><li>(+) strukturyzacja danych</li><li>(+) niezawodność (backupy)</li><li>(+) większa ochrona (security)</li></ul>
<ul style="list-style-type: none"><li>(-) brak struktury informacyjnej</li><li>(-) słaba ochrona (security)</li></ul>	<ul style="list-style-type: none"><li>(-) wysoki koszt</li><li>(-) narzuty na wydajność</li><li>(-) problemy z zapisem niektórych danych (jako BLOB)</li></ul>
<ul style="list-style-type: none"><li>• Obszerne wolumeny danych, zwłaszcza pozbawione wewnętrznej struktury</li><li>• Dane archiwalne, kopie, logi</li><li>• Dane tymczasowe</li><li>• Masowe dane będące źródłem informacji dla bazy danych</li></ul>	<ul style="list-style-type: none"><li>• Dane dostępne dla wielu użytkowników, z różnych perspektyw</li><li>• Dane ustrukturalizowane, podlegające wyszukiwaniu i przetwarzaniu na różne sposoby</li><li>• Dane specjalnie chronione</li></ul>

## Pojemniki danych (3)

- **Poza plikami zwykłymi (bez struktury) i relacyjnymi bazami danych istnieją również inne rozwiązania np.:**
  - Pliki przechowywania/transferu danych ustrukturalizowanych (np. XML, JSON)
  - Bazy danych obiektowe i obiektowo-relacyjne
  - Możliwość przechowywania obszernych danych w jednym polu BD (np. Binary Large Object – BLOB, Character Large Object – CLOB)
  - Bazy danych bez ścisłe narzuconej struktury (NOSQL)

# Wybór priorytetów

- Określają przyjęty styl projektowania
- Obowiązują dla dalszych etapów projektu
- Często cele są trudne do zmierzenia i nie dają się wzajemnie porównać
- Bierze się pod uwagę nie tylko docelowy system, ale i proces jego budowy (np. termin instalacji ważniejszy niż wydajność)
- Przykładowe kryteria:
  - zapotrzebowanie na zasoby sprzętowe
  - wydajność czasowa
  - łatwość utrzymywania
  - przenośność
  - odporność na awarie
  - jasność i przejrzystość rozwiązania
  - modyfikowalność

**Wypracowanie rozwiązań kompromisowych!**

# Projektowanie klas

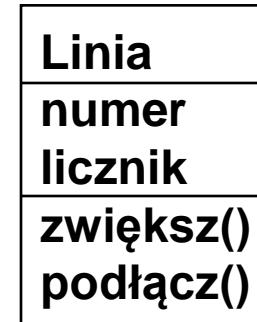
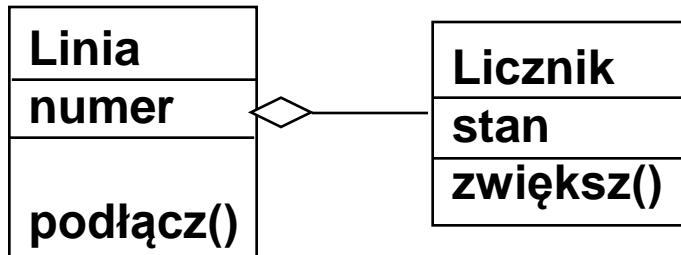
- Czyli projekt szczegółowy / niskiego poziomu
- Praca nad wewnętrzną budową poszczególnych podsystemów czyli określenie struktury kodu dla nich
- Przesunięcie nacisku z pojęć dziedziny aplikacyjnej w kierunku pojęć informatycznych, uwzględnienie technologii
- Optymalizacja klas pod kątem projektu i implementacji (elastyczny kod), nie pod kątem analizy (reprezentacja dziedziny problemowej)
- Dodanie lub/i rozwinięcie elementów modelu analitycznego (jeśli został przygotowany)
- Projektowanie klas jest już bardzo blisko **związane z implementacją** (kodowaniem) i niekoniecznie musi być wyróżniane jako osobny podobszar – często te działania są po prostu wykonywane w ramach pracy nad kodem

# Projekt klas - zakres

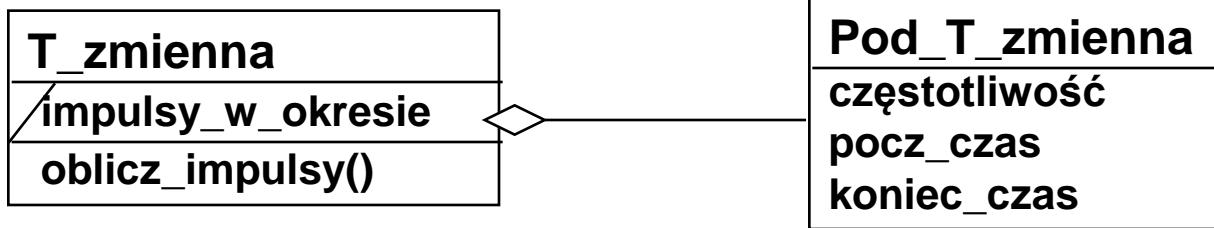
- **Wprowadzenie dodatkowych klas ukierunkowanych na implementację (np. kontrolery, obsługa urządzeń we/wy, dostęp do danych)**
- **Optymalizacja struktury (scalanie klas w większe lub rozbijanie na osobne, dodatkowe związki, atrybuty wywiedzione)**
- **Integracja modeli: klas i dynamicznego**
- **Projekt algorytmów (oraz struktur danych do nich)**
- **Projekt realizacji związków**
- **Projekt danych**
- **Uwzględnienie dobrych praktyk projektowania np. SOLID**

# Optymalizacja struktury

- Decyzje co do docelowych klas (grupowanie/rozbijanie tych z modelu analitycznego)



- Przechowywanie atrybutów nadmiarowych



- Dodatkowe związki (np. żeby klasy, które często korzystają nawzajem ze swoich metod, nie musiały robić tego za pośrednictwem innych)
- Zastosowanie wzorców projektowych (następny wykład)

# Optymalizacja – docelowa postać

- **Spójność semantyczna:** klasa dotyczy jednego aspektu projektowanego systemu
- **Klasa realizuje tylko kilka zadań**
  - „rule of thumb”: nie więcej niż 20 atrybutów, 10 asocjacji i 20 operacji
  - W razie potrzeby podział klasy: generalizacja, agregacja
- **Metoda realizuje tylko jedno zadanie**
- **Techniki polepszające modularność dotyczące atrybutów**
  - atrybuty prywatne (hermetyzacja)
  - dostęp do atrybutów przez operacje

# Optymalizacja - metryki obiektowe

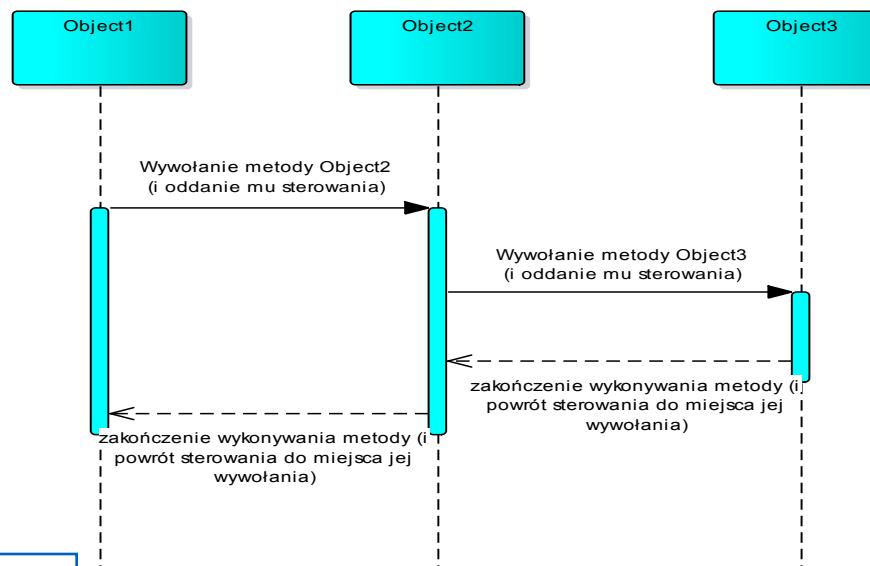
- W analizie struktura klas ma odzwierciedlać świat rzeczywisty, dziedzinę problemową
- W projekcie (i potem implementacji) struktura klas ma być odpowiednio elastyczna i modyfikowalna, co można mierzyć za pomocą pewnych metryk np.:
  - (średnia) liczba metod w klasie;
  - (średnia) długość metody;
  - Cohesion (spójność) – na ile metod danej klasy korzystają z siebie wzajemnie oraz z atrybutów klasy;
  - Coupling (zależność) – na ile dana klasa korzysta z zewnętrznych klas i ich metod;



# Integracja modelu klas z modelem dynamicznym

- Akcje i czynności będą implementowane jako **metody**
- Tranzycje (zmiany stanów) też będą realizowane przez metody
- Obsługa zdarzeń zależy od zdarzenia i aktualnego stanu
- Z odbieranym zdarzeniem można związać metodę
- W programie sekwencyjnym wysyłanie zdarzenia do innego obiektu powoduje oddanie sterowania

Hasło
hasło
liczba_prób
max_liczba_prób
wczytaj()
sprawdź( )



# Projekt algorytmów

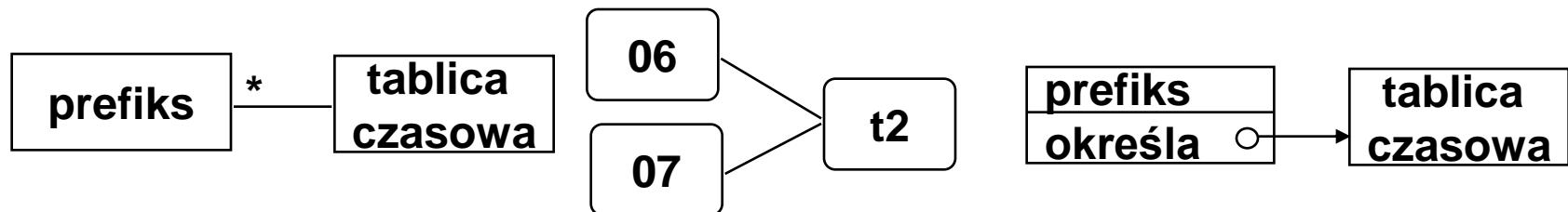
- Analiza: punkt widzenia usługi (CO?) ← operacje  
Projekt: punkt widzenia realizacji (JAK?) ← metody
- Projekt algorytmów obejmuje
  - wybór algorytmów
  - wybór struktur danych dla algorytmów
  - definicje klas wewnętrznych (pomocniczych)
  - zdefiniowanie skomplikowanych operacji za pomocą prostszych docelowych metod
  - przypisanie metod do klas
- Wybór algorytmów – możliwe kryteria wyboru
  - złożoność obliczeniowa/zajętość pamięci
  - łatwość implementacji i przejrzystość
  - modyfikowalność
  - metryki kodu

# Projekt związków (1)

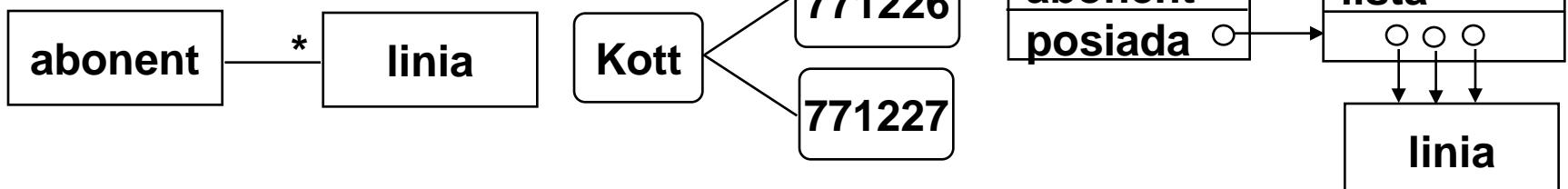
Związek jednokierunkowy, w zależności od jego liczności i kierunku zapytań, jest realizowany jako wskaźnik lub zbiór (lista) wskaźników.

*kierunek zapytań*

Znajdź tablicę czasową odpowiadającą danemu prefiksowi.



**znajdź linie danego abonenta**

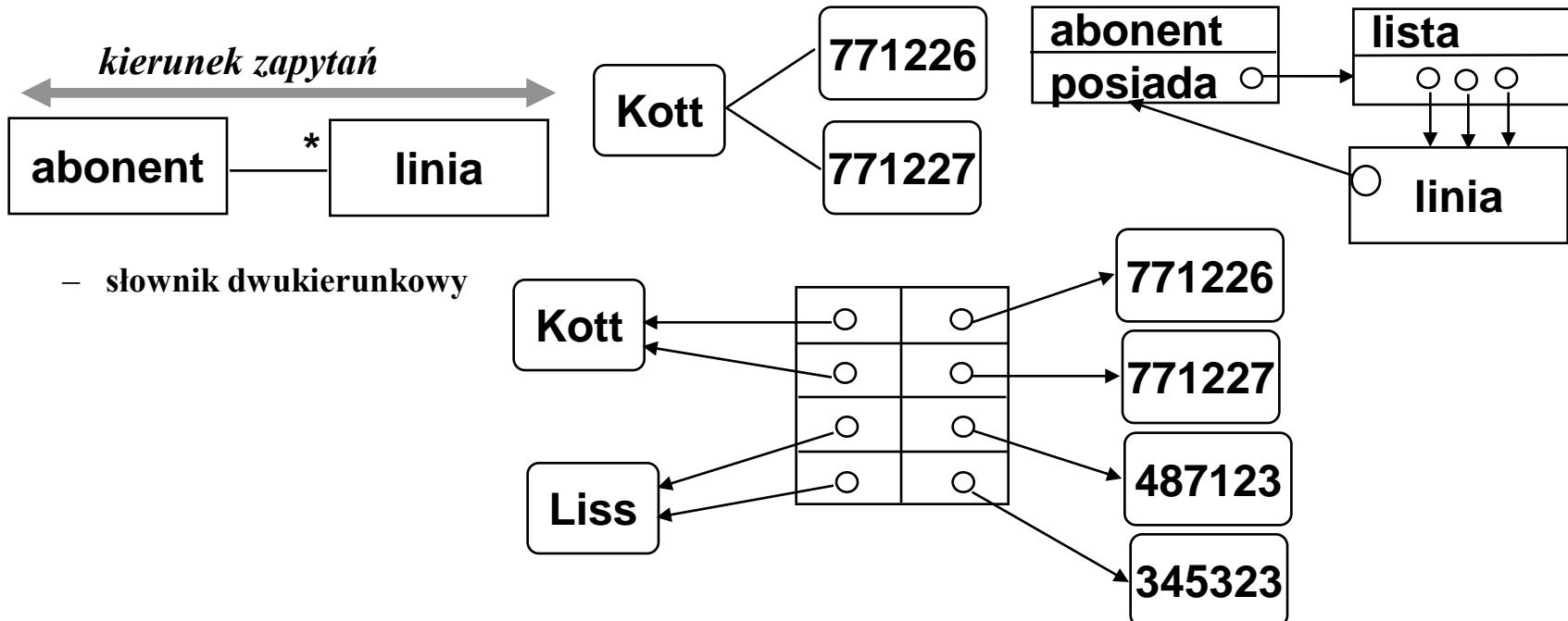


# Projekt związków (2)

Realizacja związku dwukierunkowego:

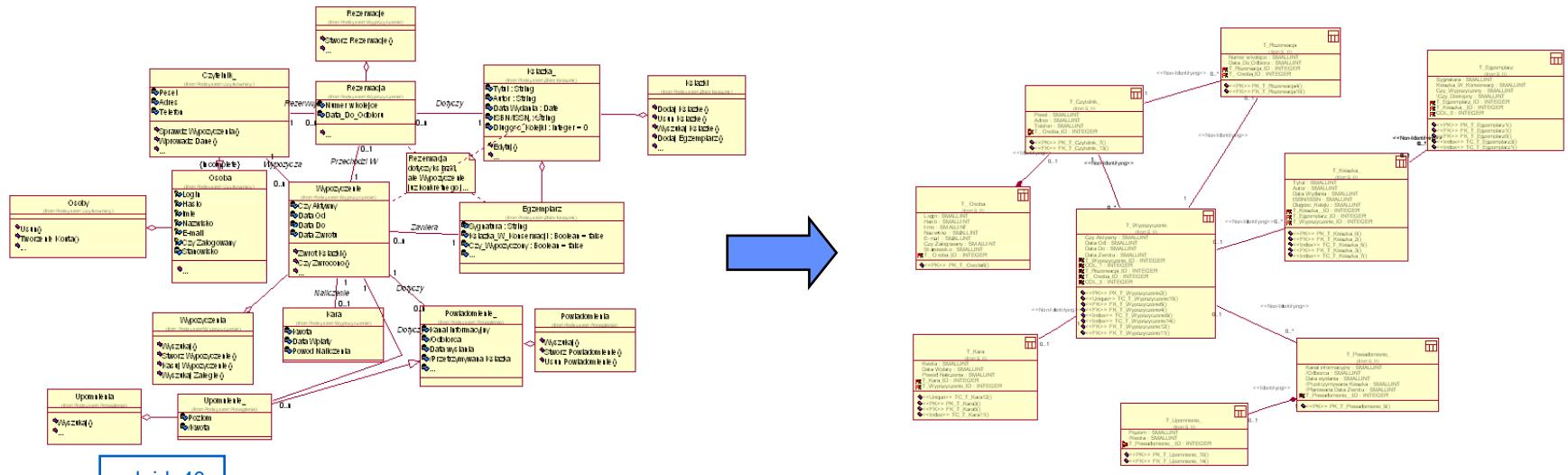
- jak jednokierunkowe i przeszukiwanie (kosztowne)
- podwójne jednokierunkowe (trudna aktualizacja)

Znajdź wszystkie linie tego abonenta, do którego należy dana linia.



# Projektowanie danych

- Jeśli w skład systemu ma wchodzić baza danych, należy zdecydować, które klasy mają być trwałe (ang. *persistent*)
  - Narzędzia CASE umożliwiają automatyczne wygenerowanie modelu BD (i wyrażeń SQL tworzących bazę dla określonych DBMS) w oparciu o model obiektowy
  - Pojawiają się wówczas zagadnienia dobrego projektowania baz danych (-> przedmioty „Bazy Danych” i „Struktury BD”)



# Niektóre problemy projektowania danych

- Różnice w paradymatach: obiektowym i relacyjnym
- Tożsamość obiektów
- Złożone atrybuty (kolekcje, inne obiekty)
- Dziedziczenie (w tym redefiniowanie własności)
- Kontenery
- Zachowanie (operacje/metody)
- Hermetyzacja
- Ukrywanie/przesłanianie informacji

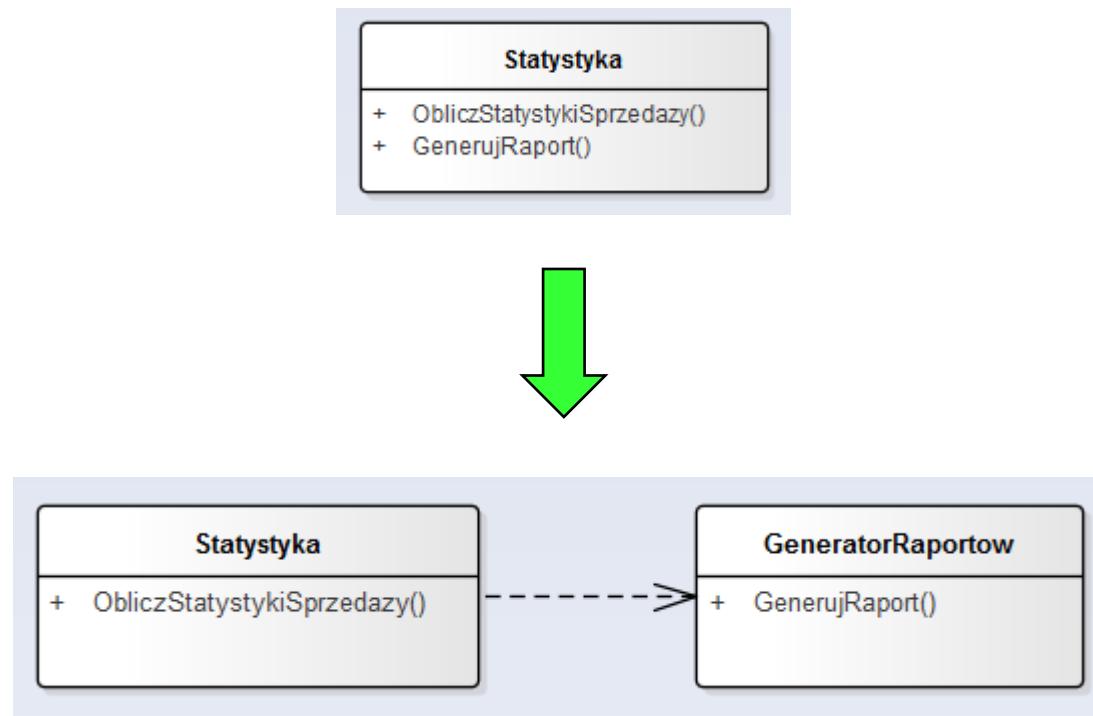
# Dobre praktyki projektowania obiektowego

- **Zestawy dobrych praktyk / zasad projektowania (ang. *principles*)**
- **Przykład: SOLID principles Roberta C. Martina**
- **Akronim SOLID:**
  - **S – Single-responsiblity principle**
  - **O – Open-closed principle**
  - **L – Liskov substitution principle**
  - **I – Interface segregation principle**
  - **D – Dependency Inversion Principle**

## SOLID: S

- **S: Single-responsibility principle (zasada pojedynczej odpowiedzialności)**
- „Nigdy nie powinno być więcej niż jednego powodu do modyfikacji klasy”
- Czyli klasa ma pojedynczą odpowiedzialność, powód istnienia
- Jeżeli w danej klasie „jest zbyt dużo”, odpowiada ona za 2 lub więcej różnych spraw – należy zdefiniować 2 lub więcej klas komunikujących się ze sobą przez zdefiniowane interfejsy

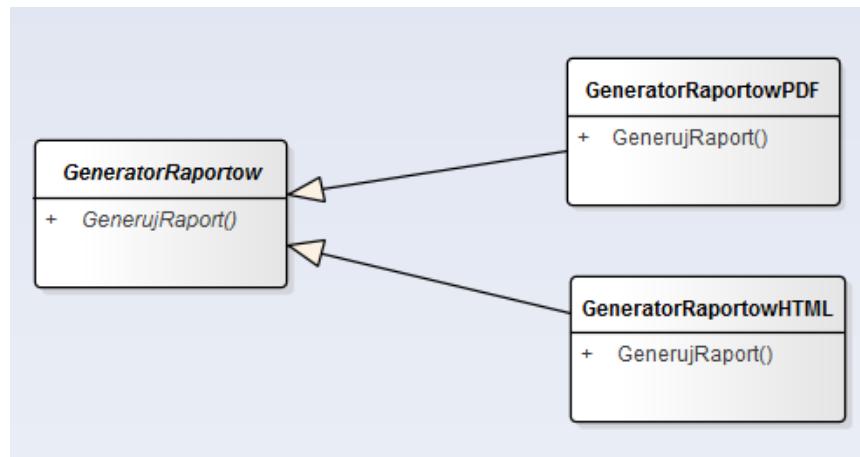
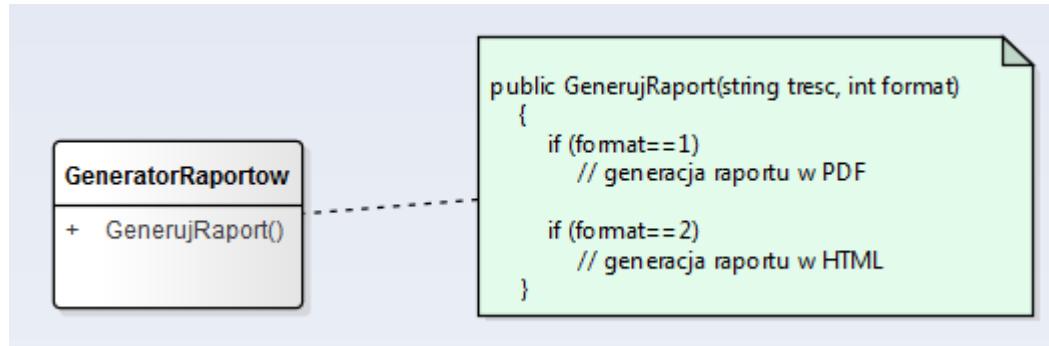
# SOLID: S - przykład



## SOLID: O

- **O: Open-Close principle ([zasada otwarte-zamknięte](#))**
- „**Klasa powinna być otwarta na rozszerzenia, ale zamknięta na modyfikacje**”
- **Czyli jeżeli chcemy zaimplementować dodatkowe wymagania, najlepiej żeby nie zmieniać już istniejących właściwości klasy (atrybutów, metod), lecz dodać nowe**
- **Wykorzystanie możliwości paradygmatu obiektowego np. rozdziału interfejs/implementacja, metod abstrakcyjnych, polimorfizmu**

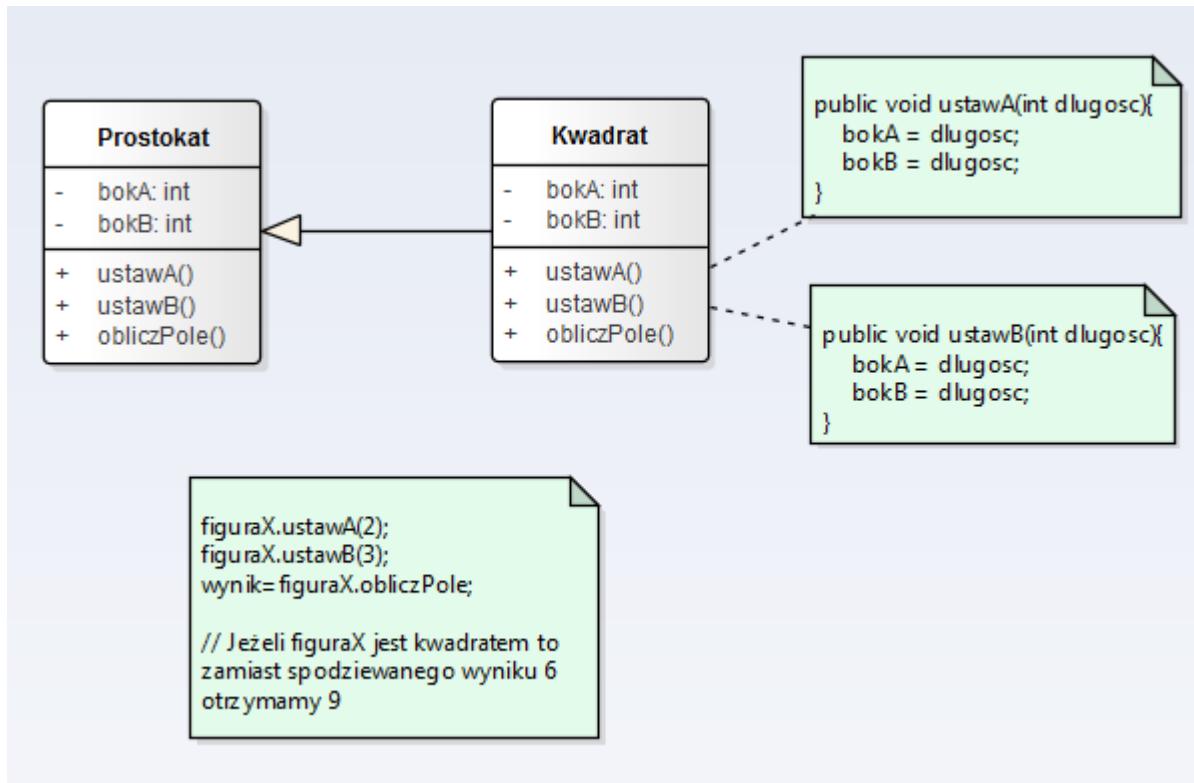
# SOLID: O - przykład



## SOLID: L

- **L: Liskov substitution principle (zasada podstawienia Liskov)**
- „Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów”
- Czyli jeżeli w jakimś miejscu używamy klasy bazowej, powinniśmy również móc użyć klasy dziedziczącej po niej
- **Polimorfizm!**

# SOLID: L - przykład

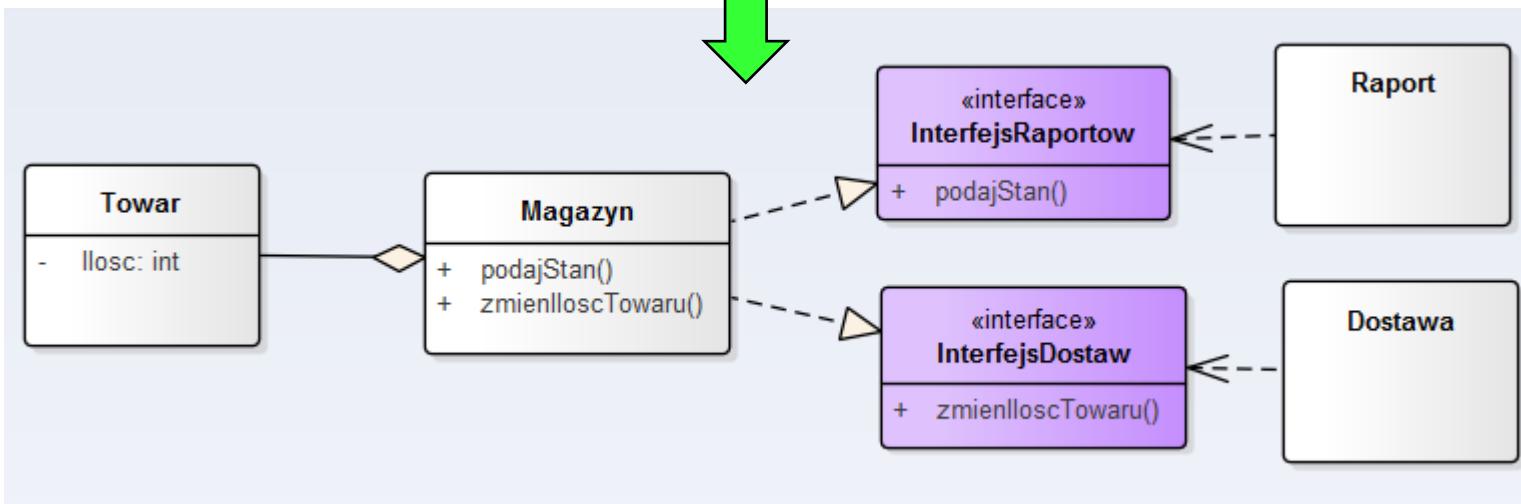
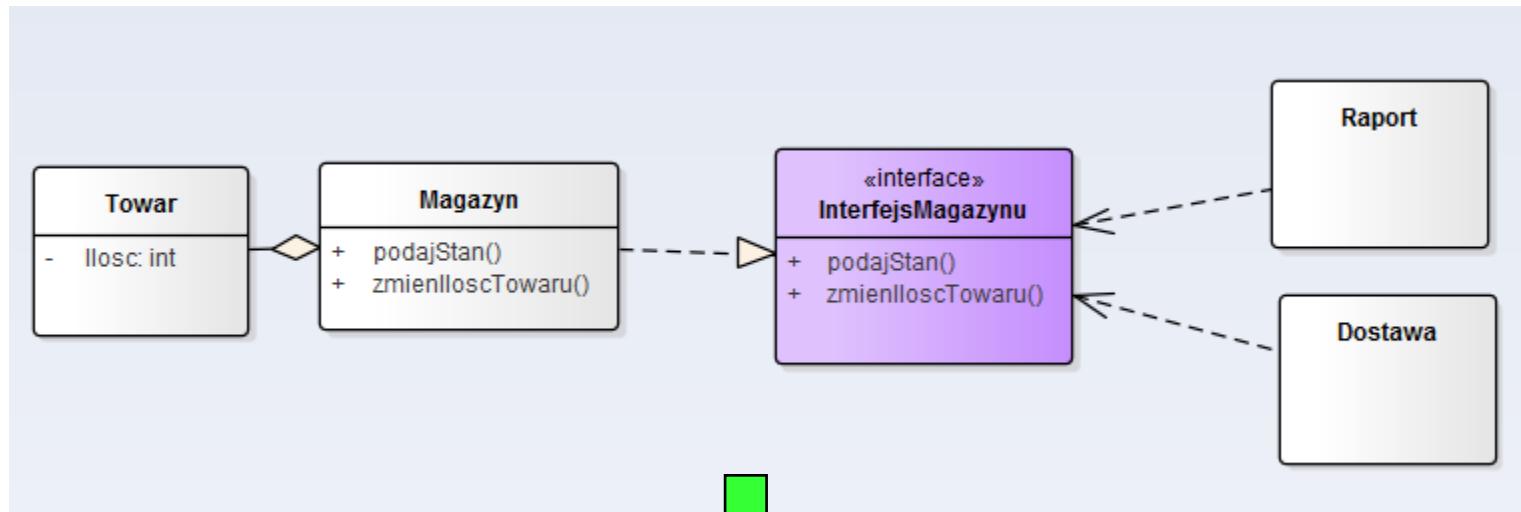


W tym przypadku możliwym rozwiązaniem jest wprowadzenie poprzez generalizację ogólniejszej klasy np. Figura, a w niej uniwersalnej metody definiowania jej boków np. poprzez podawanie współrzędnych wierzchołków

## SOLID: I

- **I: Interface segregation principle (zasada segregacji interfejsów)**
- „Klasy nie powinny być zmuszane do zależności od metod, których nie używają”
- Czyli jeżeli definiujemy interfejsy, nie powinny one zawierać kombinacji dużej liczby metod, z których tylko niektóre będą potrzebne klasom klienckim korzystającym z tych interfejsów
- Lepiej jest stosować wiele niewielkich interfejsów niż jeden rozbudowany

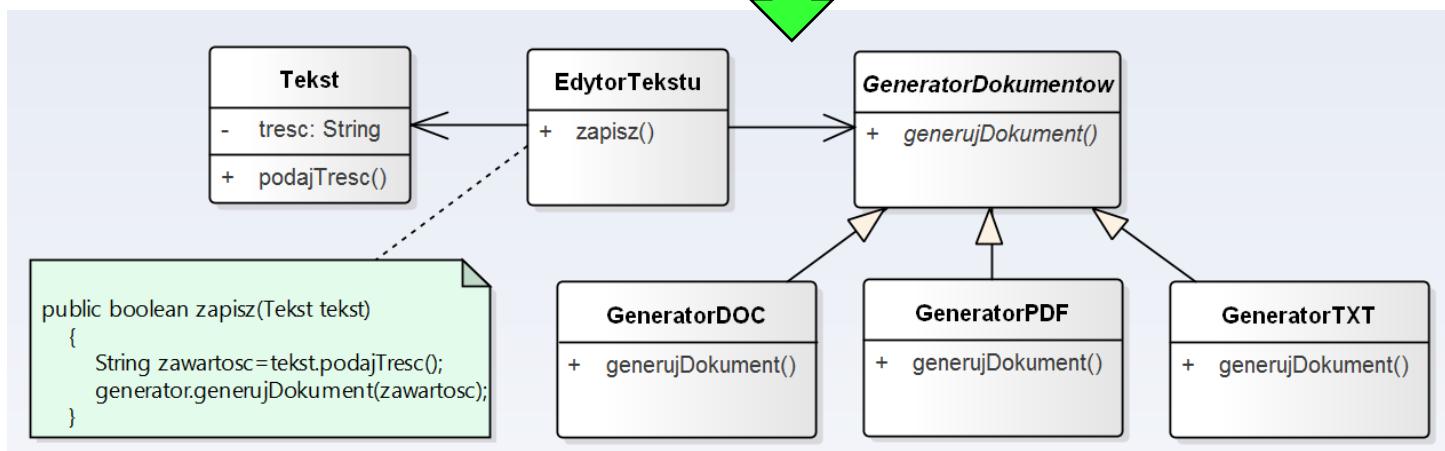
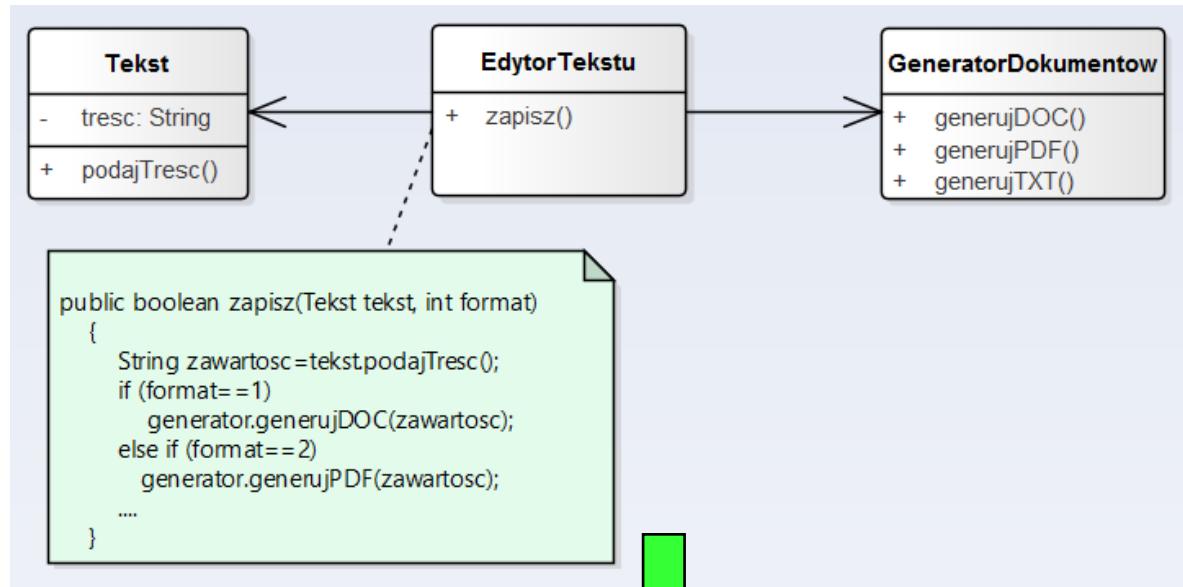
# SOLID: I - przykład



## SOLID: D

- **D: Dependency inversion principle (zasada odwrócenia zależności)**
- „Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych – zależności między nimi powinny wynikać z abstrakcji”
- Czyli główne elementy naszego kodu obiektowego nie powinny zależeć od swoich składowych np. na zasadzie wywoływania konkretnych metod
- Wykorzystanie interfejsów, z których korzystają moduły wysokopoziomowe, a które są implementowane przez moduły niskopoziomowe

# SOLID: D - przykład



# Literatura

- **Len Bass, Paul Clements, Rick Kazman:** *Software Architecture in Practice*, Third Edition. Addison Wesley, 2012
- **Robert C. Martin:** *Czysta Architektura*, Helion, 2018
- **Ian Gordon:** *Essential Software Architecture*, Second Edition, Springer, 2011

# Software Reuse

*Aleksander Jarzębowicz*

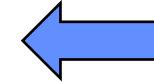
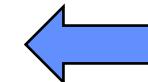
*Katedra Inżynierii Oprogramowania  
Wydział ETI, Politechnika Gdańsk*

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich  
rozprowadzanie jest zabronione.

# Idea wielokrotnego użycia (*reuse*)

- Istotą wielokrotnego (ponownego) użycia jest wykorzystanie efektów wcześniejszej działalności człowieka np.:
  - pomysłu,
  - wiedzy,
  - doświadczenia,
  - gotowego produktujako elementu wspierającego proces nabywania nowej wiedzy, doświadczenia, czy też wytwarzania nowego produktu
- Definicja szeroka, dotyczy nie tylko oprogramowania

# Software reuse

- **Software reuse** to proces budowy lub rozwoju systemów informatycznych przez wykorzystanie już istniejących zasobów ponownego użycia (*reuse assets*)
  - Ogólnie polega na tworzeniu, udostępnianiu i wykorzystywaniu tych składników przedsięwzięć informatycznych, które nadają się do wielokrotnego zastosowania w tym samym lub w innych przedsięwzięciach np.
    - kodu
    - dokumentacji
    - rozwiązań projektowych
  - Tworzenie składników – bardzo istotne i ciekawe zagadnienie, ale w ramach tego wykładu nacisk położony jest przede wszystkim na wykorzystanie
-  *Develop with reuse*
-  *Develop for reuse*

# Kategorie software reuse

- Generalnie zasobem ponownego wykorzystania może być praktycznie każdy artefakt czy know how związany z procesem wytwarzania oprogramowania
  - wiedza dziedzinowa
  - wymagania
  - modele biznesowe/analityczne
  - architektury systemów (w tym wzorce architektoniczne)
  - rozwiązania projektowe (w tym wzorce projektowe)
  - kod źródłowy
  - kod wykonywalny
  - scenariusze/przypadki/skrypty testowe
  - dokumentacja
  - dane
  - narzędzia
  - ...

**Wyróżniamy jednak pewne podstawowe kategorie:**

- **Code reuse**
- **Component reuse**
- **Framework reuse**
- **Software Product Lines**
- **Pattern reuse**

## Code reuse (1)

- Od początku programowania formą *reuse* było wyodrębnianie fragmentów kodu możliwego do wykorzystanie bezpośrednio lub po parametryzacji w wielu systemach
- Wraz z rozwojem języków programowania i środowisk wytwarzania, coraz więcej dostępnych bibliotek zawierających użyteczne funkcje/struktury np.
  - kolekcje danych,
  - algorytmy,
  - operacje wejścia-wyjścia,
  - elementy GUI,
  - security,
  - zarządzanie procesami,
  - ...

## Code reuse (2)

- Dla poszczególnych języków programowania istnieją stowarzyszone z nimi *biblioteki standardowe*, ponadto jednak dostępna jest cała masa *innych bibliotek udostępnianych przez różne podmioty/autorów*
  - choć granica tutaj trochę się zaciera – dołączanie bibliotek w kolejnych wersjach
- **Biblioteki standardowe - przykłady**
  - Java: `java.lang`, `java.text`, `java.security`, `java.sql`
  - .NET: `System.Collections`, `System.IO`, `System.Threading`
- **Inne biblioteki – praktycznie nieograniczone zakres, wszystko, co ktoś uzna za warte opracowania i udostępnienia (np. na GitHub, Sourceforge)**
  - elementy GUI, testy jednostkowe (JUnit), logowanie zdarzeń (Log4j), integracja z popularnymi aplikacjami (Twitter4J), zoptymalizowana obsługa np. typów prostych czy dat w por. do bibliotek standardowych (`lang3`, `joda-time`), uczenie maszynowe (TensorFlow), ...

# Component reuse

- Component reuse polega na wykorzystaniu gotowych komponentów udostępniających określoną funkcjonalność
- Przykłady: sterownik, komponent przetwarzający dane/sygnały, komponent dostępu do określonego źródła danych, parser
- Może być dostępny w gotowej postaci, bez ujawniania kodu źródłowego
- Kluczowa jest specyfikacja interfejsów zewnętrznych
- Komponenty jako realizacja usług w podejściach opartych o *web services* i *Service-Oriented Architecture*

**Komponent** – niezależnie wytworzony element programowy, udostępniający swoją funkcjonalność za pomocą jednoznacznie zdefiniowanego interfejsu, przy niekoniecznie jawnych szczegółach implementacyjnych, zdolny do współdziałania z większą całością oraz innymi komponentami.

## Framework reuse (1)

- Wykorzystanie szkieletu (in. zrębu, ramy, platformy) aplikacji, wypełnianego i dostosowywanego do rozwiązania konkretnego problemu
- Chyba nie ma dobrego tłumaczenia, więc i po polsku mówi się o frameworkach
- Framework zapewnia podstawowe, wspólne mechanizmy oraz określa ogólny przepływ sterowania
  - Inversion of Control – to framework kontroluje przepływ sterowania, w przeciwieństwie do np. bibliotek, z których po prostu można coś wywołać
- Programista rozszerza istniejący szkielet o konkretną specyficzną funkcjonalność

## Framework reuse (2)

- **Framework dla określonych zastosowań dziedzinowych:**
  - systemy Enterprise Resource Planning (ERP),
  - przetwarzanie multimediiw,
  - budowa parserów i kompilatorów
- **Framework dla wytwarzania określonego rodzaju oprogramowania np. *web application framework* (WAF):**
  - ASP.NET,
  - Django,
  - Node.js
  - Apache Wicket,
  - Ruby on Rails,
  - Symfony,
  - ...

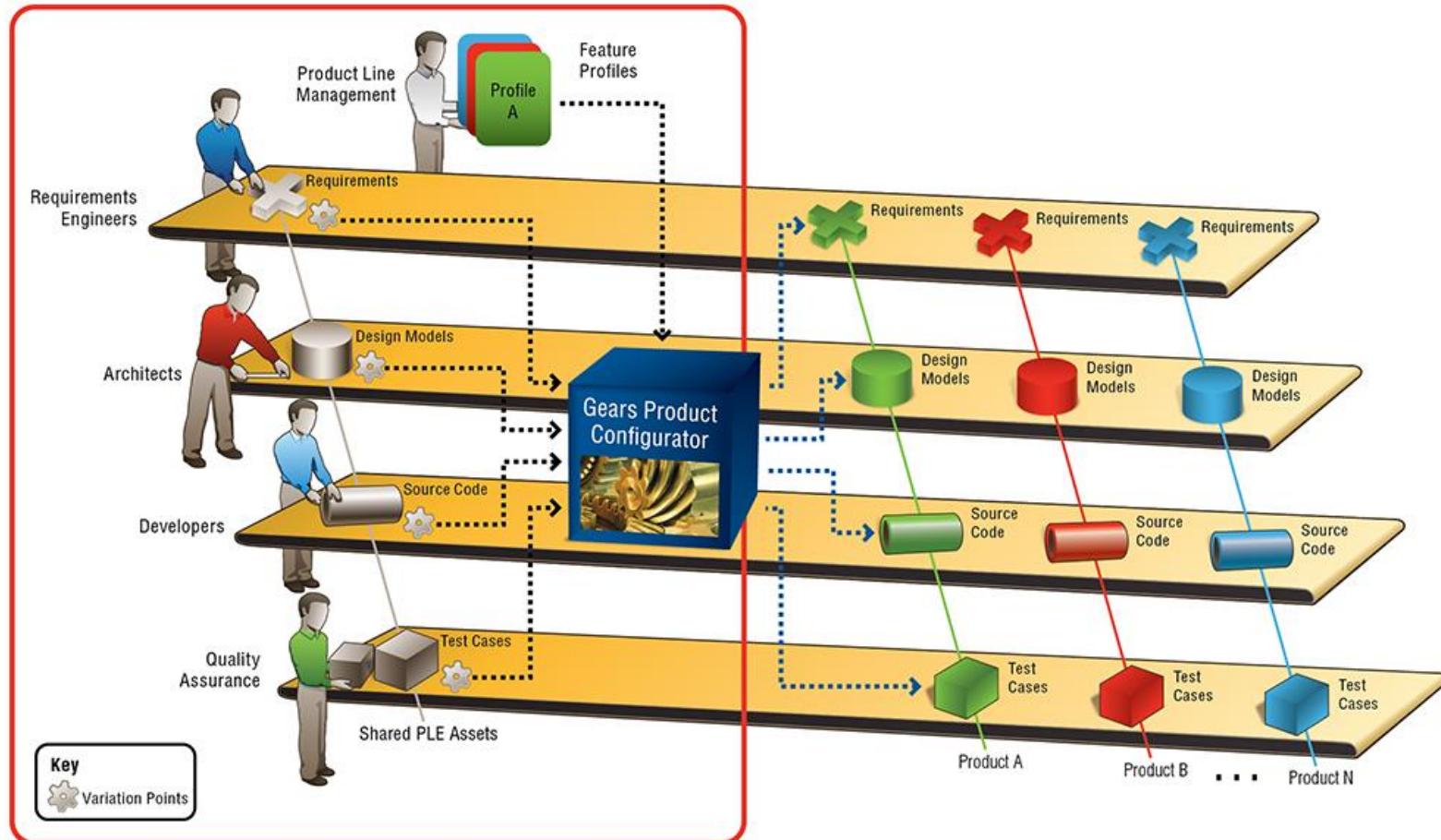
# Problemy z kategoriami reuse

- **Wątpliwości np. gdzie „kończy się” biblioteka a „zaczyna” komponent?**
  - Komponent oferuje raczej zestaw funkcjonalności niż zbiór funkcji, ponadto nie musi mieć jawnych źródeł
- **Podobnie, jaka jest różnica między biblioteką a frameworkiem?**
  - Generalnie biblioteka oferuje pewne funkcje, które są wywoływanie z kodu tam, gdzie określi to programista
  - Framework natomiast narzuca przepływ sterowania; upraszczając można powiedzieć, że programista tworzy specyficzny kod, który jest wywoływany przez framework
  - Framework może również zawierać zestaw bibliotek do wykorzystania przez programistę
- **Niekoniecznie ta klasyfikacja jest przestrzegana w nazwach własnych, nadawanych przez twórców swoim dziełom (framework, toolkit, platform, library, etc.)**

# Software Product Lines (1)

- **Reuse w sytuacji gdy mamy do czynienia ze zbiorem „podobnych” produktów wytwarzanych i utrzymywanych przez danego producenta**
- **Pojęcie rodzin produktów (*Product Families*) np.**
  - systemy finansowo-księgowe dla różnych rodzajów firm i dodatkowo z różnych krajów (odmienne przepisy)
  - oprogramowanie systemowe dla różnych urządzeń (telefony komórkowe, telewizory)
  - systemy wspomagające pilota dla różnych rodzajów samolotów (Boeing)
  - różne wersje danej aplikacji (wersje Windows/Linux, wersje Standard/Professional/Enterprise)
- **Produkty takie bazują na pewnych wspólnych elementach (projekt, część kodu) oraz na ujednoliconym procesie produkcji**

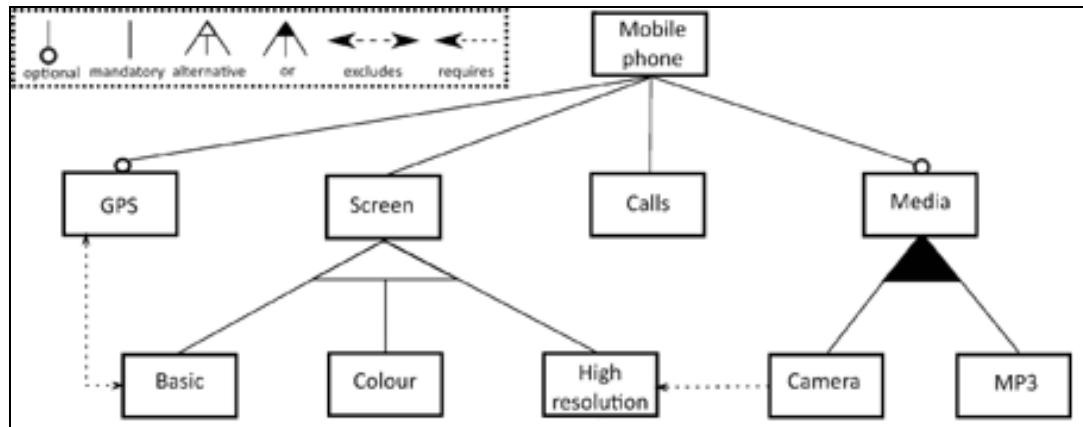
## Software Product Lines (2)



Źródło: <http://www.productlineengineering.com>

## Software Product Lines (3)

- Gdy produkty różnią się pewnymi cechami (*features*), można to zobrazować za pomocą tzw. *feature diagrams*



Źródło:  
ERCIM News  
No. 93 (2013)

- Ujednolicony proces z wykorzystaniem wspólnych komponentów składowych
- „Konfigurowalność” tych komponentów

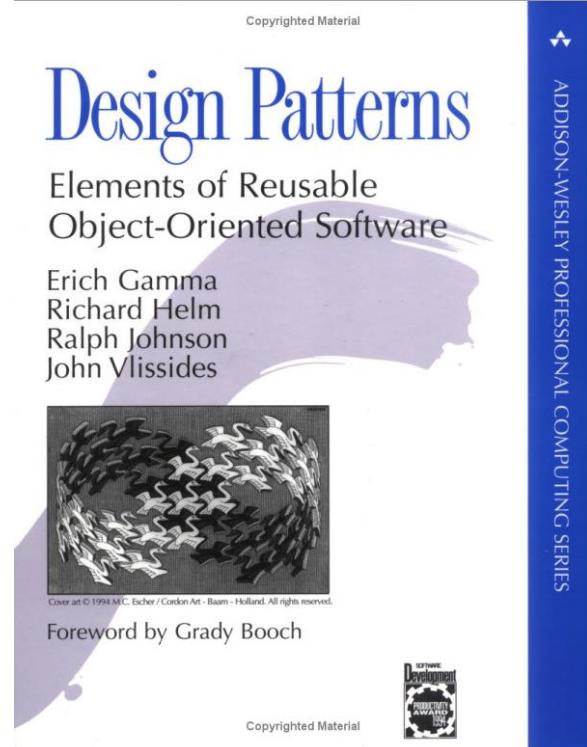
## Pattern reuse (1)

- Sytuacja – pierwsza połowa lat 90, rozwój i popularyzacja technologii obiektowych
- Problem umiejętności wykorzystania środków wynikających z paradygmatu obiektowego i mechanizmy oferowane przez obiektowe języki programowania
- Dobry projekt systemu obiektowego – kwestie elastyczności, łatwości utrzymania i modyfikacji, ponownego wykorzystania
- Obserwacja, że doświadczeni projektanci najczęściej tworzą dobre projekty, bazując (nawet nie do końca świadomie) na sprawdzonych wcześniej rozwiązaniach

## Pattern reuse (2)

- **Obserwacja** – istnieje szereg problemów spotykanych przy projektowaniu, które są (w znacznym stopniu) niezależne od kontekstu, tematyki budowanego systemu, języka programowania
- Dla problemów tych istnieją „eleganckie” rozwiązania, które jak pokazuje praktyka często spotyka się w dobrze zaprojektowanych systemach
- **Pomysł** – opisać problemy i rozwiązania w takiej formie aby można było je w sposób świadomy wielokrotnie wykorzystywać, w postaci wzorców (ang. *patterns*)
- **Pattern reuse** – wykorzystanie nie konkretnego kawałka kodu, ale pomysłu rozwiązania

# Gang of Four i „Design Patterns”



- **Książka „*Design Patterns. Elements of Reusable Object-Oriented Software*”, Addison-Wesley, 1995**
- **Autorzy: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (*Gang of Four - GoF*)**
- **Upowszechnienie idei wzorców projektowych**
- **Opis ponad 20 najbardziej interesujących wzorców**

# Czym jest wzorzec projektowy?

**Wzorzec** w systematyczny sposób nazywa, uzasadnia i wyjaśnia pewne ogólne rozwiązanie problemu powtarzającego się w kodzie różnych systemów.

Wzorzec opisuje **problem**, jego **rozwiązanie**, wskazówki **kiedy** zastosować rozwiązanie (problem i kontekst) i jakie będą tego **konsekwencje**. Rozwiązaniem jest ogólny model elementów, klas i obiektów, ich związki, odpowiedzialności i współpracę.

Rozwiązanie to powinno następnie zostać przystosowane i zaimplementowane tak, aby rozwiązywało problem w jego szczególnym kontekście (klasy/obiekty we wzorcu będą odpowiadać klasom/obiektom danego systemu).

- **Wzorzec jest rozwiązaniem:**
  - uniwersalnym
  - zweryfikowanym w praktyce
  - dla powtarzalnych, nie jednostkowych problemów
- **Wzorzec nie jest:**
  - algorytmem
  - komponentem
  - biblioteką klas, funkcji czy innych elementów kodu

# Elementy wzorca projektowego

- **Nazwa wzorca** - adekwatna i łatwa do użycia :-)
- **Problem** – kiedy zastosować dany wzorzec
  - jaki problem rozwiązuje
  - jak rozpoznać symptomy tego problemu np. w strukturze obiektów
  - jaki jest wymagany kontekst i warunki, które trzeba spełnić aby zastosować dany wzorzec
- **Rozwiązanie** – jak powinien wyglądać projekt
  - struktura i jej elementy: klasy i związki, obiekty i powiązania
  - dynamiczna współpraca elementów
  - wskazówki implementacyjne dla różnych technologii
- **Konsekwencje** – świadome przedstawienie za i przeciw
  - jakie są potencjalne braki wzorca
  - *trade offs*
  - utrzymanie i rozszerzanie systemu z budowanego z użyciem wzorca

# Przykładowa struktura opisu wzorca

- Name
- Also known as
- Intent
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Struktura wg GoF „Design Patterns”

# Kategorie wzorców

- Wg propozycji GoF wyróżniono 3 kategorie (rodziny) wzorców:
- Konstrukcyjne (ang. *creational*) – dotyczą procesu tworzenia nowych obiektów
- Strukturalne (ang. *structural*) – opisują określone kompozycje, struktury powiązanych ze sobą obiektów
- Zachowań (ang. *behavioral*) – opisują dynamiczne zachowanie i odpowiedzialność współpracujących ze sobą obiektów

# Przykłady wzorców projektowych

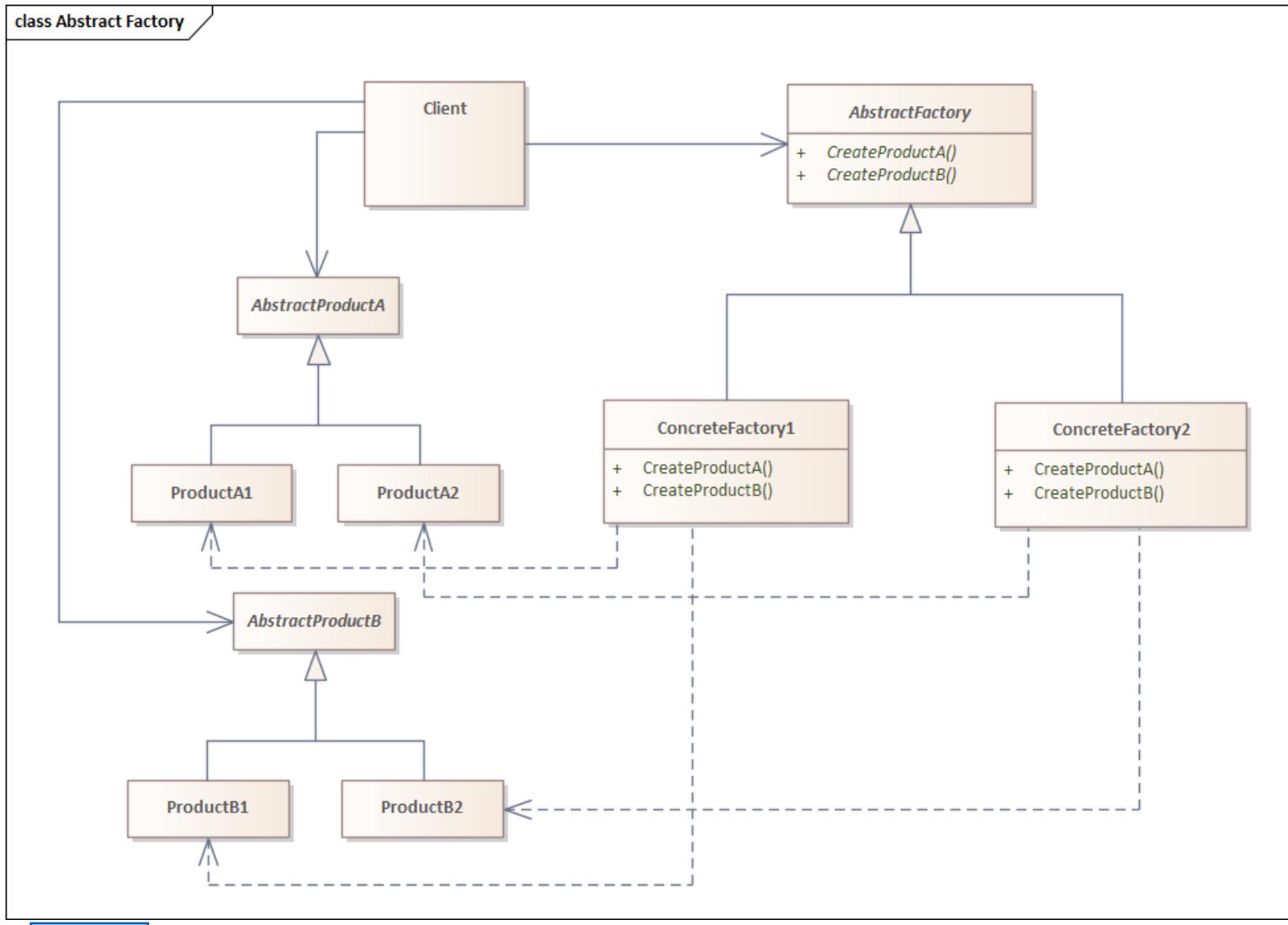
- **Creational patterns:**
  - Abstract Factory
  - Builder
- **Structural patterns:**
- **Behavioral patterns:**
  - Observer

Źródło: GOF „Design patterns”

# Abstract Factory (1)

- Cel: dostarczyć interfejs do tworzenia grup („rodzin”) powiązanych ze sobą obiektów, ale bez specyfikowania ich konkretnych klas
- Przykładowy problem:
  - chcemy aby system oferował różne standardowe typy interfejsów użytkownika (MS Win, OSX, Mac)
  - każdy typ interfejsu będzie oferował podobny zestaw elementów (okienka, przyciski, ...), które jednak w każdym mają inne właściwości i inaczej wyglądają
  - nie chcemy wkodować informacji o wszystkich konkretnych oknach itp. z uwagi na małą elastyczność i problemy z dodaniem nowych typów interfejsów
  - klient (obiekt, aplikacja) korzystający z elementów nie musi wiedzieć jakiej do jakiego konkretnego typu należy element, wystarczy, że wie jak korzystać z operacji np. narysuj()

# Abstract Factory (2)



# Abstract Factory (3)

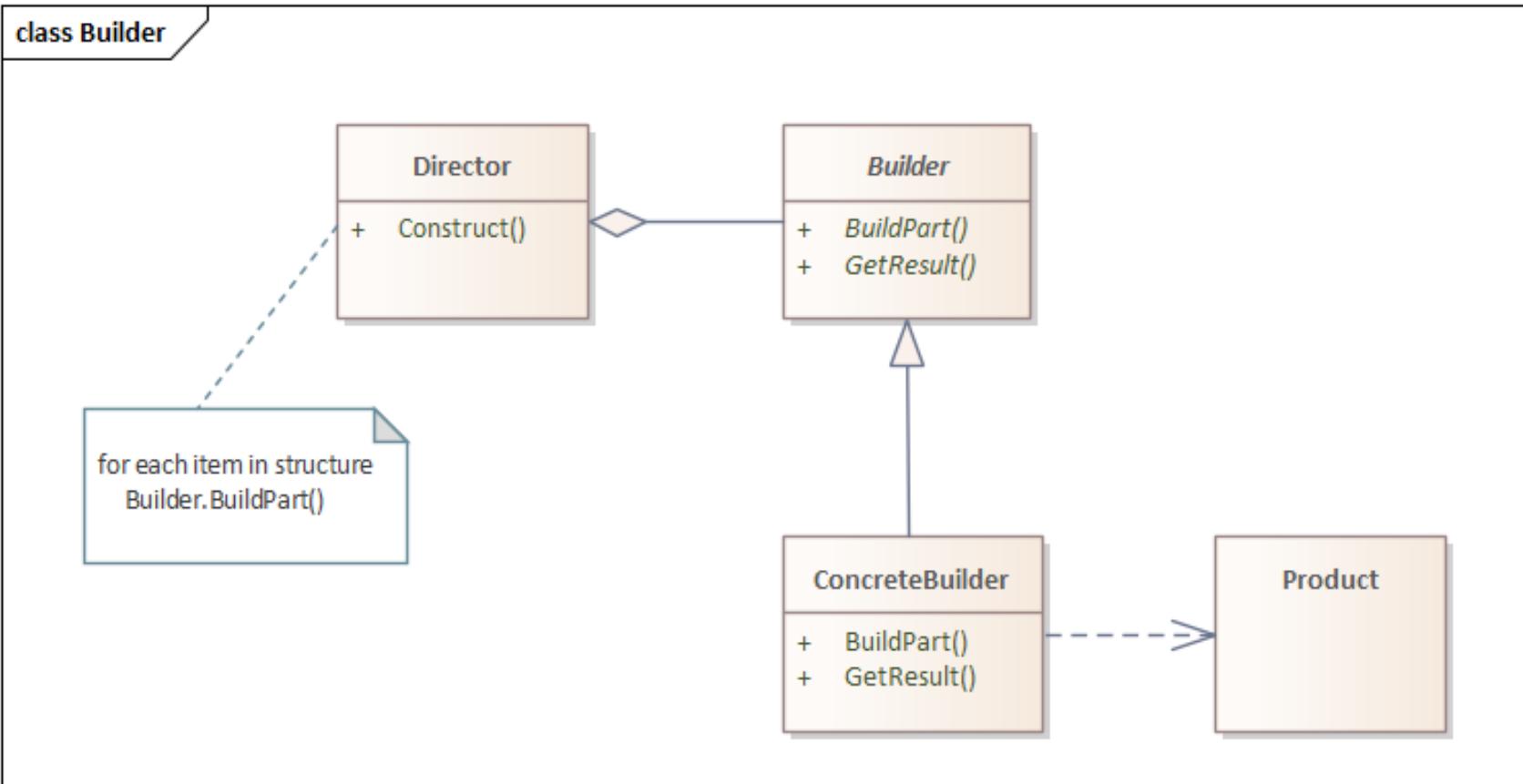
- **Konsekwencje:**

- Odizolowanie klienta od problemu do jakiej konkretnej klasy należy obiekt. Klient posługuje się nim jedynie przez abstrakcyjny interfejs. Nazwy konkretnych klas nie pojawiają się w kodzie klienta
- Możliwość łatwej podmiany rodzin obiektów (można dodać kolejną)
- Spójność – wszystkie obiekty używane przez klienta będą należały do jednej rodziny
- Problemy z rozszerzaniem rodziny o nowe obiekty – trzeba zmodyfikować i klasę AbstractFactory i wszystkie jej konkretne podklasy (**przykład konsekwencji negatywnej!**)

# Builder (1)

- Cel: Odseparować proces tworzenia złożonej struktury od jej konkretnej reprezentacji, w taki sposób, że ten sam proces tworzenia pozwala tworzyć różne reprezentacje
- Przykładowy problem:
  - tworzymy edytor, który ma zapisywać dokumenty w różnych formatach (ASCII, RTF, HTML, PDF, ...)
  - proces zapisu wygląda zawsze podobnie: edytor parsuje element po elemencie dokument wyrażony w swojej wewnętrznej reprezentacji i zapisuje ten element (tekst, znacznik) w odpowiednim formacie
  - różnica dotyczy konwersji na określony format
  - nie chcemy ograniczać liczby i rodzaju formatów, w przyszłości może trzeba będzie dodać nowe

## Builder (2)



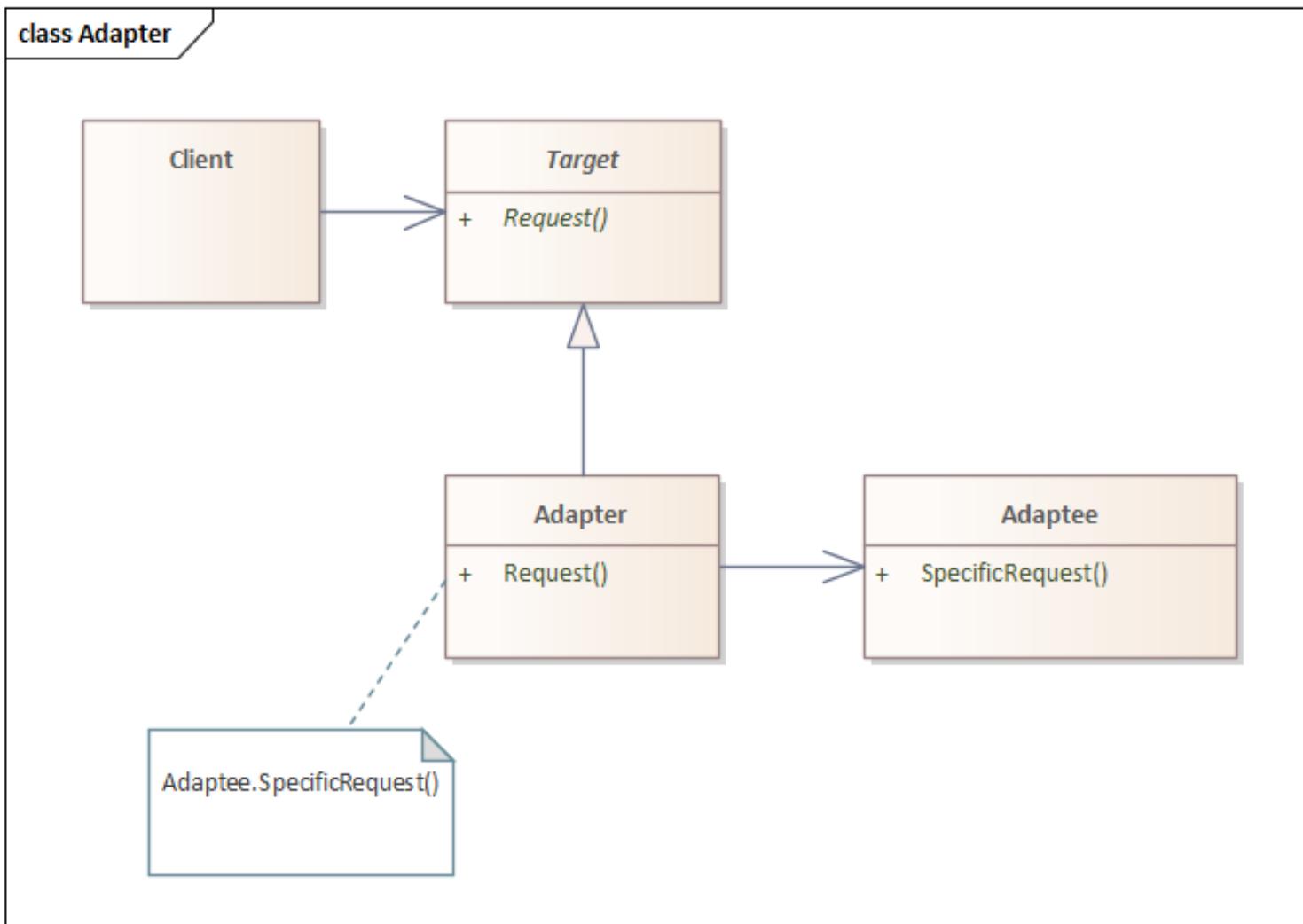
## Builder (3)

- Konsekwencje:
  - Możliwość zróżnicowania wewnętrznej reprezentacji jakiejś złożonej struktury
  - Możliwość odseparowania kodu dotyczącego algorytmu tworzenia i kodu dotyczącego wewnętrznej reprezentacji struktury (lepsza modularyzacja)
  - Zachowanie kontroli nad procesem tworzenia po stronie klienta (obiektu Director) – struktura jest tworzona krok po kroku, a nie w rezultacie wywołania jednej operacji

# Adapter (1)

- Cel: przekształcić interfejs danej klasy w inny interfejs (taki jakiego oczekuje klient)
- Wzorzec pomocny w sytuacji gdy mamy jakieś klasy, których chcemy użyć, ale nie możemy z uwagi na to, że ich interfejsy są niekompatybilne
- Przykładowy problem:
  - Tworzymy edytor, w którym użytkownik ma mieć do dyspozycji elementy takie jak: linie, kształty geometryczne, pola tekstowe itp.
  - Zdefiniowaliśmy sobie, że wszystkie te elementy mogą być traktowane jednakowo (przesuwane, skalowane), dzięki temu, że mają taki sam interfejs (wszystkie dziedziczą od abstrakcyjnej klasy *Shape*)
  - Chcielibyśmy wykorzystać gotowy element – klasę *TextView* do obsługi pól tekstowych, który jednak ma inny interfejs, niepasujący do *Shape*

# Adapter (2)



# Adapter (3)

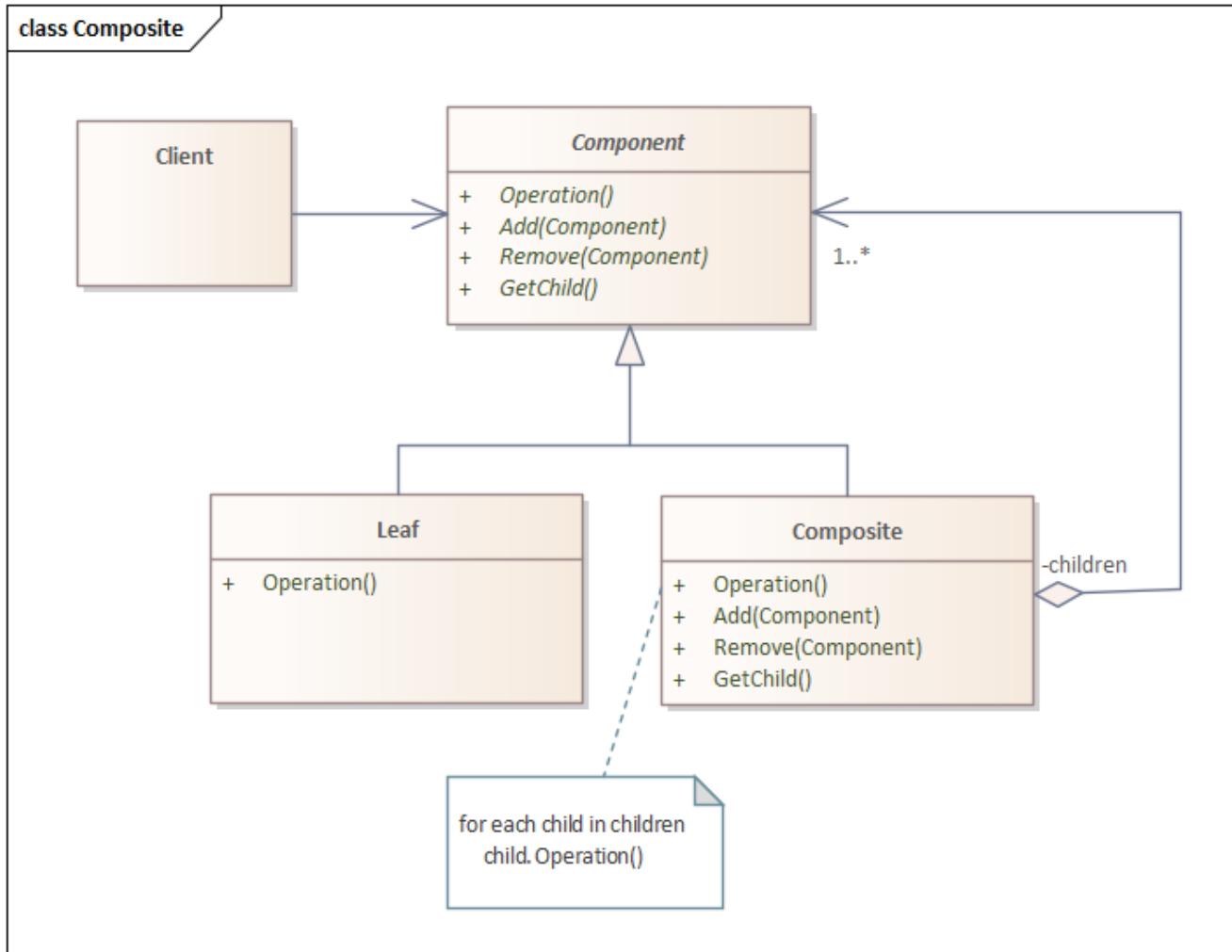
- **Konsekwencje:**

- Adapter musi wykonać pewną pracę, żeby zrealizować interfejs Target-u za pomocą operacji udostępnianych przez Adaptee. W poszczególnych przypadkach może to być albo proste wywoływanie analogicznych operacji, albo też wykonywanie niemal wszystkich operacji samemu.
- Jeśli zbyt wiele pracy będzie wykonywane przez sam Adapter, może się okazać, że nie warto korzystać z Adaptee...
- Możliwość pracy Adaptera z wieloma Adaptees

# Composite (1)

- **Cel: Pogrupowanie obiektów w struktury drzewiaste obrazujące zależność część-całość i umożliwienie klientowi traktowania zarówno pojedynczych obiektów, jak i całych struktur w ten sam sposób**
- **Przykładowy problem:**
  - Piszemy edytor graficzny, w którym użytkownik powinien mieć możliwość grupowania elementów
  - Użytkownik powinien mieć możliwość wykonywania różnych manipulacji (np. przesunięcie, skasowanie) w taki sam sposób na pojedynczych liniach czy figurach, jak i na zgrupowanych strukturach elementów
  - (Inny przykład: edytor tekstowy i działania typu: zmiana typu czy koloru czcionki możliwe do wykonania na pojedynczym znaku, akapicie, całym dokumencie)

# Composite (2)



# Composite (3)

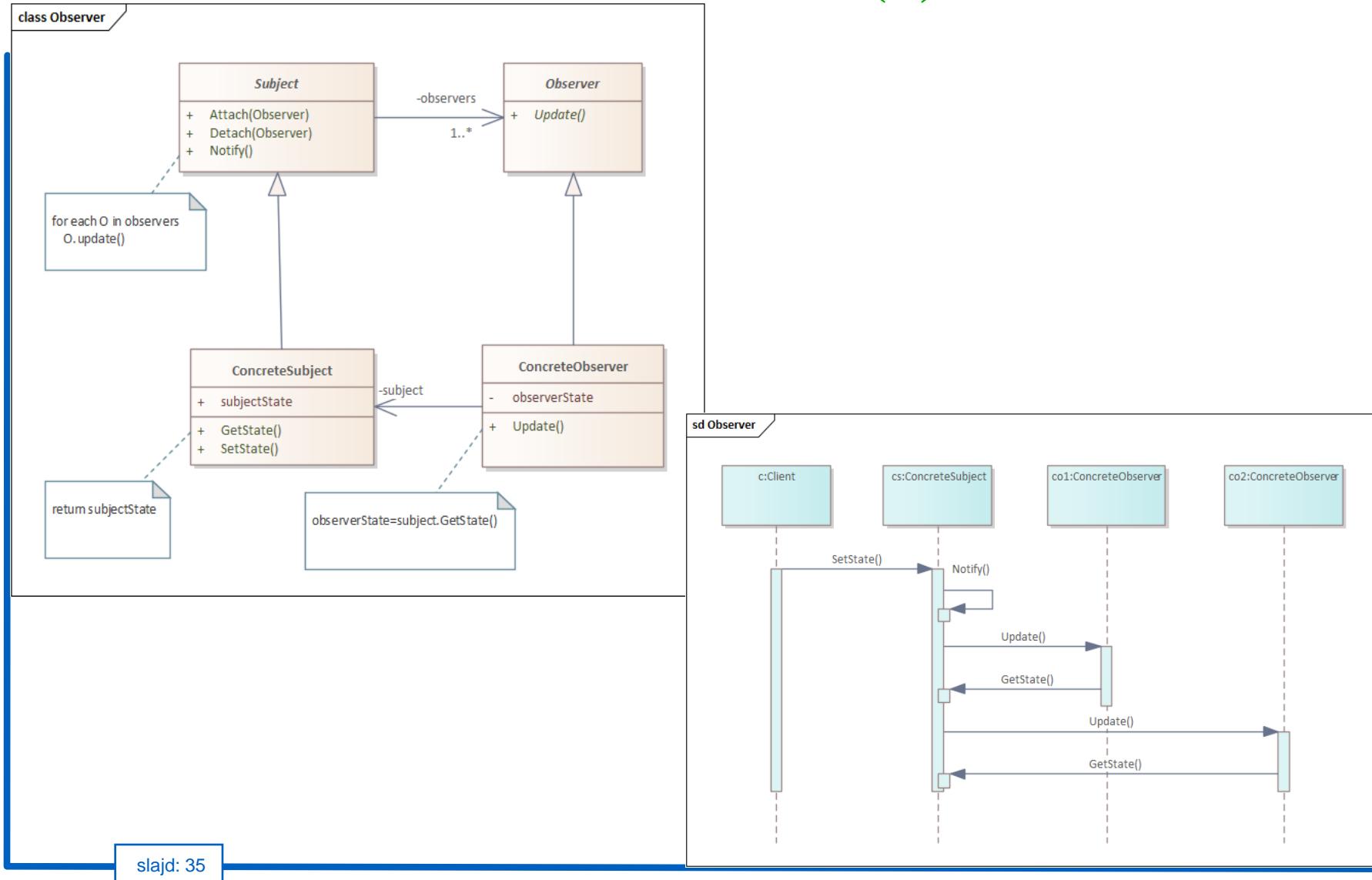
- **Konsekwencje:**

- Umożliwia tworzenie dowolnie zagnieżdżonych struktur
- Prostota kodu Klienta – Klient nie musi się przejmować tym czy ma do czynienia z pojedynczym obiektem (Leaf) czy strukturą (Composite), traktuje je w jednakowy sposób (nie trzeba więc rozpisywać warunków dla różnych klas wchodzących w skład struktury)
- Możliwość dodania nowych Komponentów (zarówno prostych, jak i struktur) bez żadnej modyfikacji Klienta
- Łatwość dodawania nowych komponentów pociąga za sobą trudność kontrolowania czy w skład struktury wchodzą tylko właściwe rodzaje Komponentów (konsekwencja negatywna)

# Observer (1)

- Cel: ustanowienie między obiektami zależności jeden-do-wiele, w której kiedy jeden kluczowy obiekt zmieni swój stan, wszystkie zależne od niego są automatycznie powiadamiane i aktualizowane
- Przykładowy problem:
  - Mamy w systemie przechowywane dane, które są jednocześnie wizualizowane przez wiele rodzajów interfejsu użytkownika
  - Np. dane mogą dotyczyć ilości towarów w magazynie i są one jednocześnie przedstawiane w formularzu używanym przez magazyniera, w witrynie sklepu internetowego przeglądanego przez klienta i na wykresach oglądanych przez menedżera sklepu
  - Chcemy żeby zmiany były natychmiast odzwierciedlane na wszystkich interfejsach tzn. jeśli magazynier zmieni ilość towarów po wysyłce, to żeby klient i menedżer korzystający z systemu od razu też widzieli zmianę

# Observer (2)



# Observer (3)

- Konsekwencje:
  - Zależność między Subject a Observer jest dość luźna, Subject wie tylko tyle, że ma listę „jakichś” Obserwatorów. Można we wzorcu stosować obiekty z różnych warstw (np. warstwy interfejsu użytkownika i warstwy logiki biznesowej)
  - Możliwość komunikacji typu „broadcast”
  - Obserwator (nie wiedząc o innych) może nie zdawać sobie sprawy z kosztu wprowadzenia zmian w obiekcie Subject, może to prowadzić do kaskady żądań i powiadomień

# Rozwój wzorców projektowych

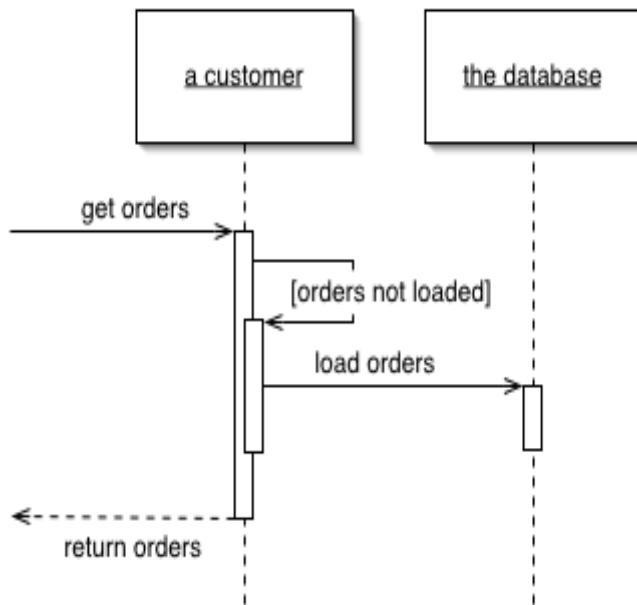
- W początkowym okresie opracowywane wzorce projektowe dotyczyły projektów wykonywanych w „klasycznych” technologiach; C++, Smalltalk, Java i aplikacji „desktopowych”.
- Rozwój technologii wspierających tworzenie aplikacji internetowych (.NET, J2EE) czy pojawienie się *Rich Internet Applications* spowodowało powstawanie nowych wzorców ukierunkowanych na nowe problemy i ich rozwiązania.
- Uwaga – nie oznacza to, że klasyczne wzorce nie mają już zastosowania, takie konstrukcje projektowe wciąż są przydatne.

## Nowe wzorce projektowe - przykłady

- **Wzorce logiki dziedziny** – tworzenie i utrzymywanie rozbudowanych aplikacji obiektowych (np. Domain Model, Service Layer)
- **Wzorce odwzorowania O-R** – utrzymywanie spójności, transfer danych między relacyjną bazą danych a obiektowymi warstwami realizującymi logikę biznesową (np. Data Mapper, Lazy Load, Unit of Work, Identity Map, Serialized Lob)
- **Wzorce prezentacji internetowych** – wizualizacja na interfejsie użytkownika w internecie (np. Two-Step View, Page Controller)
- **Wzorce dystrybucji** – wywołania i transfer danych między obiektemi rozproszonymi (np. Remote Facade, Data Transfer Object)
- **Wzorce współbieżności** – dostęp współbieżny, blokady, konieczność utrzymywania spójności (np. Optimistic Offline Lock, Implicit Lock)
- **Wzorce stanu sesji** – sposoby zapamiętywania stanu w zależności od rozmiaru i złożoności danych (np. Client Session State, Server Session State)

źródło: P. Bejger - Praca mgr WETI PG

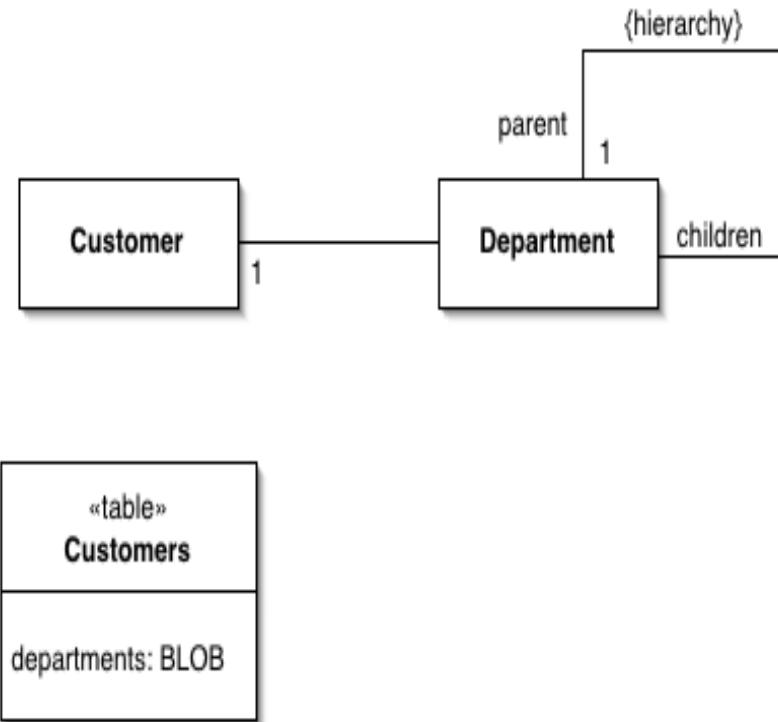
# Lazy Load



- Wzorzec odwzorowania O-R
- Przy wczytywaniu danych z BD do pamięci programu reprezentowane są one tam w postaci obiektów
- Wczytując dane np. odpowiedni wiersz czy wiersze tabeli, na ogólnie trzeba też pobrać powiązane dane (patrz przykład – zamówienia dla klienta), ale niekoniecznie wszystkie
- Lazy Load stosuje znaczniki pokazujące, które dane zostały już wczytane, a które nie i opóźnia moment pobrania danych do momentu gdy są one potrzebne

źródło: [www.martinfowler.com](http://www.martinfowler.com)

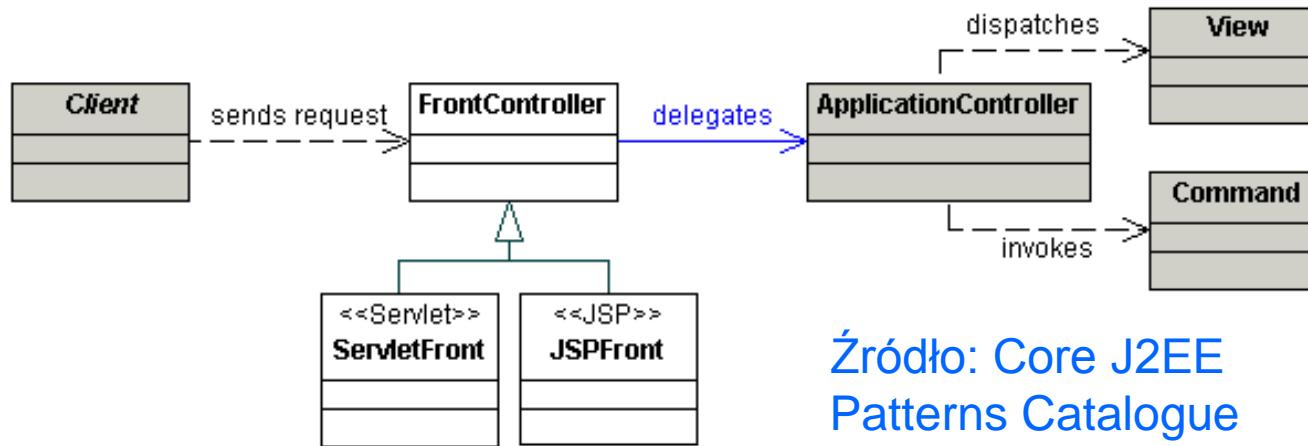
# Serialized LOB



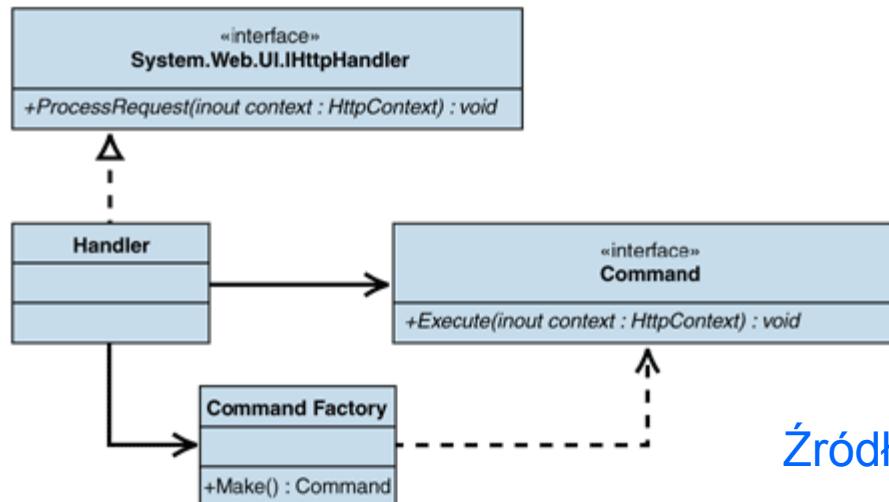
- **Wzorzec odwzorowania O-R**
- **Modele obiektowe często zawierają skomplikowane struktury hierarchiczne (czy ogólniej grafowe)**
- **Informacja, którą niosą zawarta jest nie tyle w samych obiektach, co w powiązaniach między nimi, jest to trudne do odwzorowania w schemacie relacyjnej bazy danych**
- **Jeśli jest to odizolowane od reszty aplikacji, to dla celów optymalizacyjnych, można zdecydować, by przechowywać całą tę strukturę jako pojedynczy Large Object (LOB) np. wyrażony w XML**

źródło: [www.martinfowler.com](http://www.martinfowler.com)

# Wzorce/implementacje wzorców dla konkretnych technologii



Źródło: Core J2EE  
Patterns Catalogue

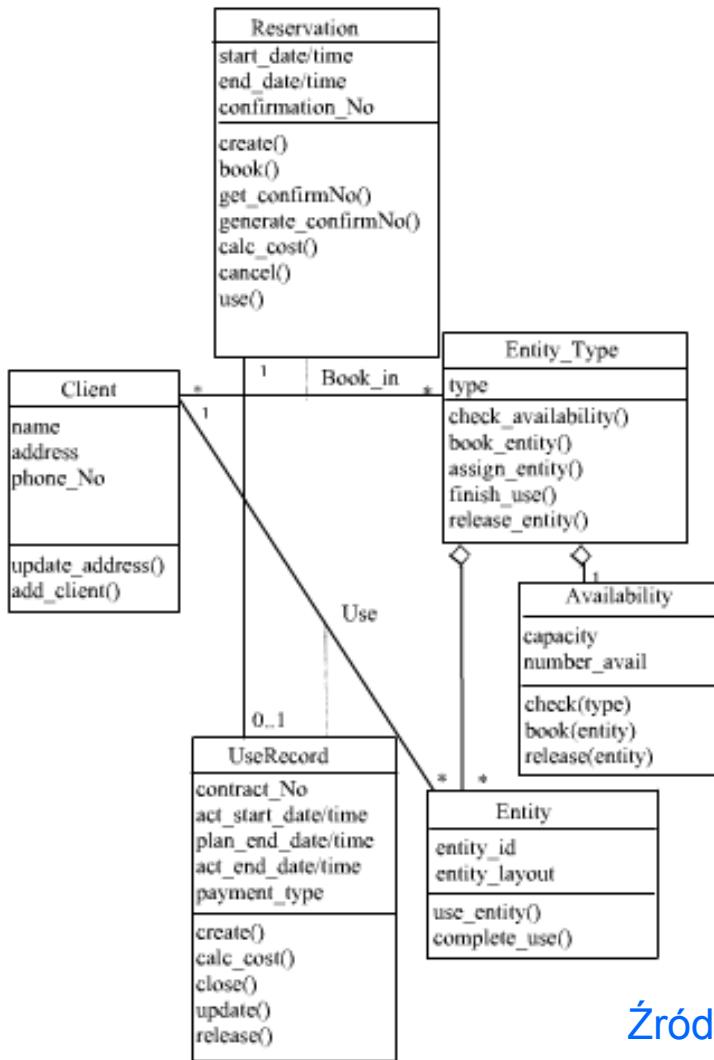


Źródło: MSDN

## Inne kategorie wzorców

- Wzorce projektowe były pierwszym tego rodzaju rozwiązaniem i stworzyły podstawy tego podejścia w inżynierii oprogramowania.
- Potem pojawiły się również inne propozycje:
  - wzorce analityczne
  - wzorce architektoniczne
  - antywzorce (programowania, architektury, zarządzania)
  - ...

# Wzorce analityczne

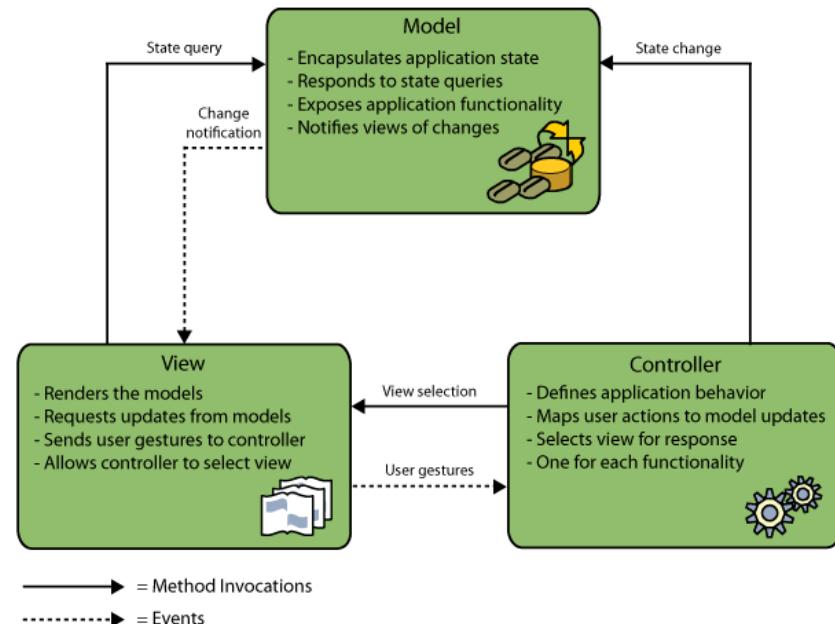


- Odzwierciedlają strukturę konceptualną procesów biznesowych i bytów świata rzeczywistego (w przeciwieństwie do faktycznej implementacji)
- W procesie analizy daje się czasem wychwycić pewne ogólniejsze reguły, możliwe do zastosowania dla wielu systemów
- Przykład: problem rezerwacji (pokojów, kaset video, biletów lotniczych, biletów do opery, nart, jachtów itp.)

Źródło: Fernandez & Yuan „An analysis pattern for reservation and use of reusable entities”

# Wzorce architektoniczne

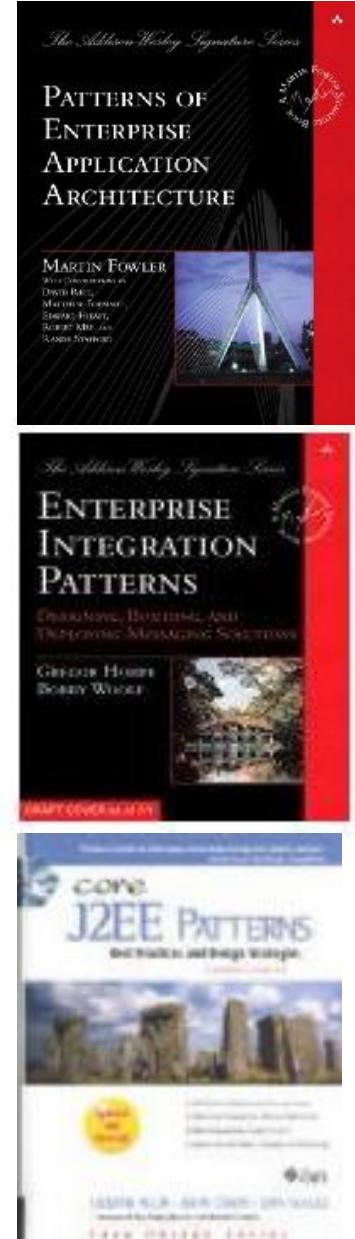
- Wzorzec architektury opisuje wysokopoziomowy podział systemu na główne podsystemy i zależności między tymi podsystemami
- Przykład: wzorzec MVC (Model-View-Controller)
- Inne przykłady:
  - Broker (komunikacja w systemach rozproszonych)
  - Pipes & Filters (krokowe przetwarzanie strumienia danych)



Źródło: [www.netbeans.org](http://www.netbeans.org)

# Źródła

- M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley
- D. Alur, J. Crupi, D. Malks, *Core J2EE Patterns*, Prentice Hall/Sun Microsystems Press
- E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns. Elements of reusable object-oriented software*, Addison-Wesley
- G. Hohpe, B. Woolf, *Enterprise Integration Patterns*, Addison-Wesley
- [www.martinfowler.com](http://www.martinfowler.com)
- [www.corej2eepatterns.com/](http://www.corej2eepatterns.com/)
- [www.enterpriseintegrationpatterns.com/](http://www.enterpriseintegrationpatterns.com/)
- [www.dofactory.com/](http://www.dofactory.com/)



## Podsumowanie

- **Wiele katalogów wzorców i innych materiałów dostępnych w Internecie. Szukając określonego wzorca można łatwo znaleźć jego opis.**
- **Stosowanie wzorców we własnych projektach wymaga pewnego doświadczenia, często dopiero wraz z kolejnymi projektami deweloper zaczyna świadomie wprowadzać wzorce albo identyfikuje w swoich dziełach potencjalne wzorce.**
- **Ogólna idea polega na tym, żeby „nie wymyślać koła na nowo” ani „nie wyważać otwartych drzwi”.**

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich  
rozprowadzanie jest zabronione.

# Testowanie oprogramowania

*Aleksander Jarzębowicz*

*Katedra Inżynierii Oprogramowania  
Politechnika Gdańskia*

# Testowanie

- **Testowanie** – analiza zachowania oprogramowania w celu pomiaru (oceny) jego jakości
- Motywacja – dość oczywista; praktycznie nie sposób napisać dłuższego kodu bez wprowadzenia do niego defektów

Uwaga – przyjmujemy dalej definicję:

*Testowanie w dziedzinie IO polega na eksperymentowaniu z wykonywalnym kodem*



## Kluczowe pojęcia i „zasady gry”



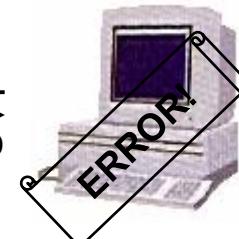
pomyłka  
człowieka

może pro-  
wadzić do



usterka (defekt)

może pro-  
wadzić do



błąd

może pro-  
wadzić do



awaria

**usterka (defekt)** – niepoprawny krok, proces lub definicja danych w oprogramowaniu

**błąd** – niepoprawny stan lub zachowanie oprogramowania (ale mogą nie być widoczne)

**awaria** – niemożność skorzystania z funkcji oprogramowania (widoczna dla użytkownika)

**Zasadniczym celem testowania jest wykrycie awarii i błędów, żeby w następnej kolejności możliwe było znalezienie i poprawienie defektów**

# Co można nazwać testowaniem?

- Środowisko powinno umożliwiać wykonanie kodu w sposób:
  1. nadzorowany:
    - rejestrowanie warunków i przebiegu testu,
    - wykonywanie “kroków testowych” w określonej, zaplanowanej kolejności
  2. kontrolowany:
    - możliwość inicjowania i wymuszania zdarzeń
    - dane wejściowe reprezentujące wszystkie istotne dla działania systemu zakresy wartości
- Dla każdego wybranego zestawu danych wejściowych powinny (przed wykonaniem testu) zostać określone oczekiwane wyniki
- Na koniec testu przeprowadzana jest analiza otrzymanych wyników pod kątem zgodności z wynikami oczekiwanyimi

U mnie działa...

# Testowanie a debugowanie

- **Testowanie**

- ma na celu sprawdzenie jakości produktu software'owego
- czasem wręcz można postawić sprawę następująco: **testowanie ma wykryć jak najwięcej błędów**
- często wykonywane przez testerów (osobny zespół, dział jakości), a przynajmniej przez kogoś niezaangażowanego w działania deweloperskie nad daną częścią systemu

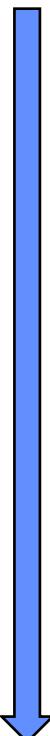
- **Debugowanie**

- ma na celu znalezienie w kodzie defektów odpowiedzialnych za błędy i awarie oraz ich usunięcie
- w skrócie: **debugowanie ma usunąć defekty**
- najczęściej wykonywane przez deweloperów – autorów kodu danej części systemu

Ale – w ramach debugowania też zwykle wykonuje się testy, żeby odtworzyć błąd, ustalić przyczynę, a po wprowadzeniu poprawek sprawdzić, czy już nie ma błędów

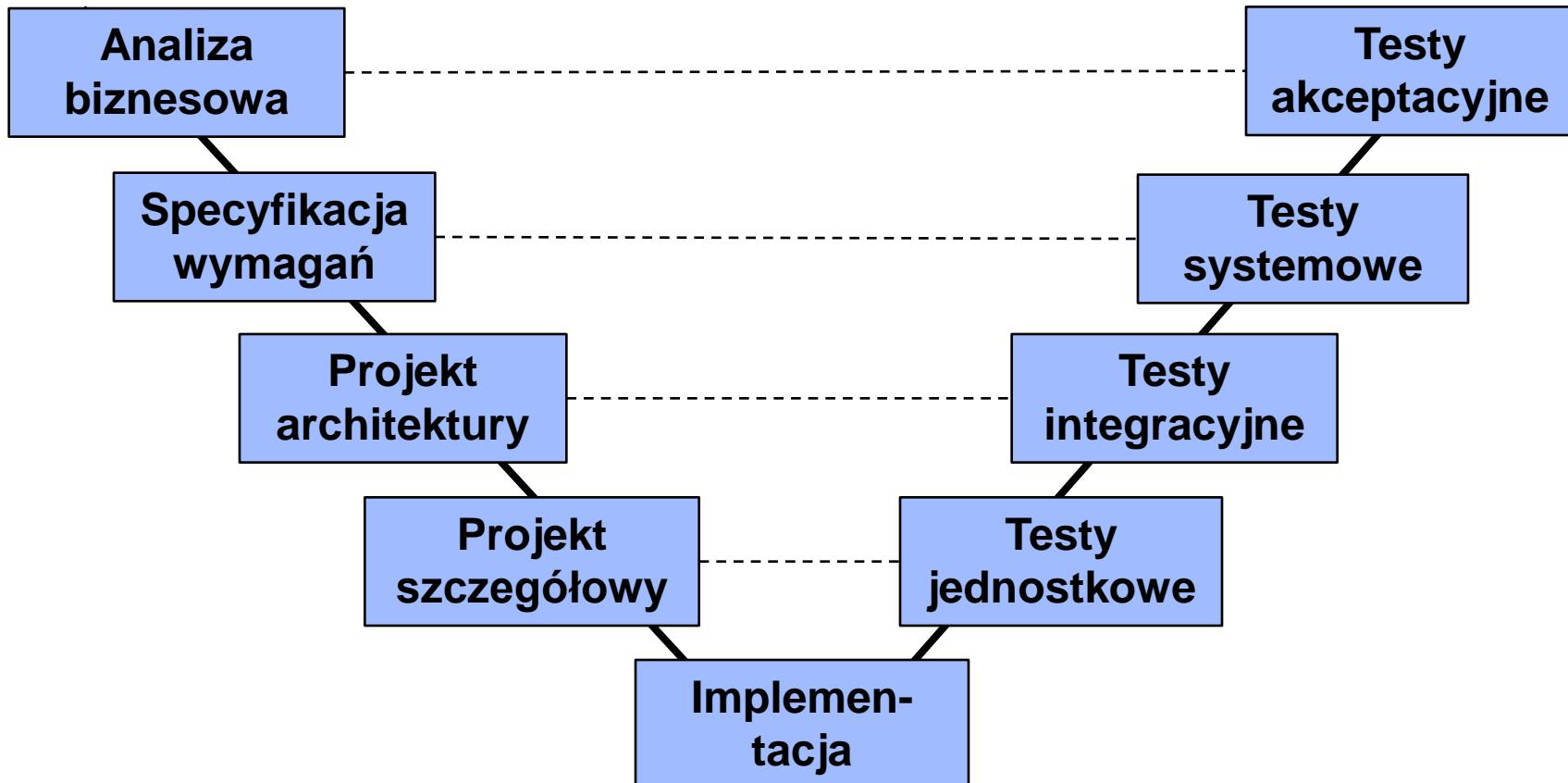
# Poziomy testowania

Poziom abstrakcji



- **Testy jednostkowe**
  - Głównym celem jest lokalizacja i usunięcie defektów w poszczególnych metodach/funkcjach/klasach
- **Testy integracyjne**
  - Celem jest sprawdzenie przygotowania poszczególnych modułów systemu do współpracy oraz sprawdzenie wspólnego działania scalonych modułów
- **Testy systemowe**
  - Celem jest sprawdzenie czy system wykonuje swoje funkcje, jest kompletny i może być wykorzystany w zakładanym środowisku
- **Testy akceptacyjne**
  - Celem jest zademonstrowanie zgodnego z potrzebami interesariuszy działania systemu

# Poziomy testowania – odpowiadanie etapom/obszarom wytwarzania



# Testowanie jednostkowe

(ang. *Unit testing*)

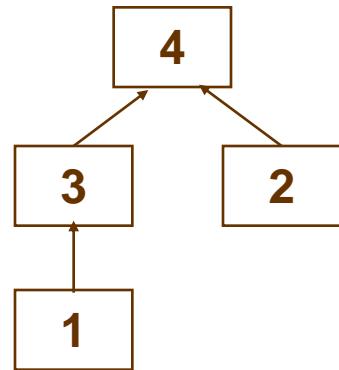
- **Cel**
  - sprawdzenie poprawności działania niewielkiego, wydzielonego „kawałka kodu”  
np. metody, klasy, kilku powiązanych klas (tzw. klaster)
  - zapewnienie poprawności działania niezależnie rozwijanych małych części kodu w oderwaniu od reszty systemu
- **Wybrane zagadnienia**
  - często o charakterze nieformalnym, bez zgłoszania/dokumentowania, po prostu naprawa od razu po wykryciu obecności defektu
  - zwykle przeprowadzane przez programistów
  - częste braki w praktyce testów jednostkowych
    - brak sformalizowania, niska waga w procesie wytwarzania
    - niedoczas i „konflikt interesów” pomiędzy pisaniem kodu a testowaniem
    - zbyt małe pokrycie (kończenie testów po sprawdzeniu kilku przypadków)
  - testy automatyczne vs testy manualne

# Testowanie integracyjne

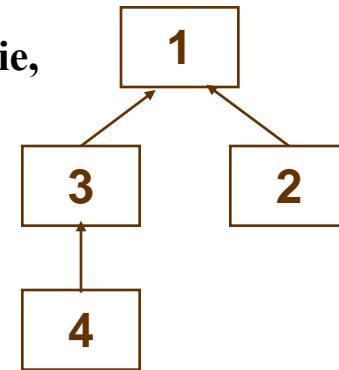
(ang. *Integration testing*)

- Testy interfejsów i scalenia modułów systemu
- Dekompozycja może być wielopoziomowa
  - System składa się z podsystemów
  - Podsystemy mogą mieć własne składowe
  - Zbiór powiązanych klas (po testach jednostkowych) może być łączony z innym zbiorem
- Strategie integracji
  - Strategia skokowa (bezprzyrostowa, ang. *big-bang integration*)
    - jednoczesne scalanie wszystkich modułów (wysokie ryzyko niepowodzenia)
  - Integracja przyrostowa
    - wступująca (oddolnie, ang. *bottom-up*)
      - wywoływanie funkcji modułów, test drivers
    - zstępująca (odgórnie, ang. *top-down*)
      - symulacja, użycie atrap

Strzałki określają udostępnianie usług/funkcji



- zstępująca (odgórnie,  
ang. *top-down*)  
- symulacja,  
użycie atrap



# Testowanie systemowe

(ang. *System testing*)

- **Weryfikacja całego, uprzednio zintegrowanego, systemu w celu sprawdzenia, czy spełnia on wyspecyfikowane wymagania**
- **Może obejmować szereg zagadnień:**
  - „testy na dym” (ang. *smoke test*) sprawdzające najważniejszy podzbiór funkcji
  - testy funkcjonalności
  - testy wymaganych cech jakościowych
    - użyteczności, ochrony (penetracyjne), wydajności (obciążeniowe i przeciążeniowe), ...
  - testy współpracy z innymi systemami
  - testy dostosowania do środowiska, ewentualnie – przenośności
- **Środowisko testowania – raczej nie docelowe, ale jak najbardziej zbliżone**
- **Zwykle prowadzone przez niezależnych testerów (nie samych programistów)**

# Testowanie akceptacyjne

(ang. *Acceptance testing*)

- Ostateczna odpowiedź, czy zbudowano właściwy system, czy jest on gotów do pełnienia funkcji biznesowych zamawiającego (wprowadzenia na rynek)
- Test formalny, wykonywany w celu ustalenia, czy system spełnia kryteria odbioru i może być przyjęty przez klienta
- Zademonstrowanie spełnienia kryteriów akceptacji
  - kryteria te powinien określić dokument specyfikacji wymagań (SWS), choć nie można wykluczyć, że SWS nie oddaje w 100% rzeczywistych potrzeb
- Wykonywane przez reprezentantów klienta i twórcy, możliwy też udział niezależnej trzeciej strony
- Środowisko testowania – produkcyjne, docelowe
- Wynik
  - przyjęcie
  - odrzucenie
  - akceptacja warunkowa

## Dodatkowe pojęcia

- **Retesty** – testy sprawdzające, czy naprawa defektu dała pomyślny rezultat
  - powtórzenie testów, które poprzednio zakończyły się porażką i doprowadziły do wykrycia błędu
- **Testy regresji (lub regresywne)** – sprawdzenie, czy przy okazji wprowadzenia zmiany nie popsuto czegoś innego
  - ponowne uruchomienie testów wykonanych poprzednio, również tych, które zakończyły się sukcesem
  - „wprowadzenie zmiany” może oznaczać naprawę defektu, ale również inne rzeczy np. implementację nowej lub zmodyfikowanej funkcji, refaktoryzację etc.
- **Retesty i testy regresji mogą być prowadzone na różnych poziomach!**

# Proces testowania

- Określenie celu i zakresu testowania
- Przygotowanie *planu testów*
  - organizacja i strategia testowania, konfiguracja oprogramowania, harmonogram
  - określenie poziomu/poziomów testowania: jednostkowe, integracyjne, systemowe, akceptacyjne
- Przygotowanie specyfikacji testów
- Projektowanie scenariuszy i przypadków testowych
  - dla poszczególnych przypadków dobierane są odpowiednie dane testowe, podawane oczekiwane wyniki, określone kryteria zaliczenia testu, ...
- Wykonanie testów i zebranie danych
  - raporty z przeprowadzonych testów
- Ocena wyników testów

# Specyfikacja testu

- **Przedmiot testowania**
  - funkcja, komponent, podsystem, system?
  - wersja / wydanie
- **Środowisko**
  - sprzęt, urządzenia
  - system operacyjny, platforma, oprogramowanie współpracujące
  - inne elementy systemu (i ich wersje), niepodlegające testowaniu
  - instrumentacja: test drivers, test stubs
- **Zakres testów**
  - co (funkcje, interfejsy, cechy itp.) będzie podlegało testowaniu, a co nie
  - podejście (czarna skrzynka / biała skrzynka / mieszane)
  - dobór scenariuszy testowych

# Scenariusz testowy

**Scenariusz testowy – systematyczna obserwacja oczekiwanej  
działalności oprogramowania;  
najczęściej - sekwencja przypadków testowych**

## Przykład

Testy (na poziomie systemowym) dokonania zakupu w systemie sklepu internetowego:

- TC.05 - Wejście do katalogu produktów
- TC.23 - Wyszukanie produktu po jego nazwie
- TC.10 - Przeglądanie szczegółów dostępnego produktu
- TC.16 - Dodanie pojedynczego produktu (1 sztuka) do koszyka
- TC.31 - Przejście do kaszy
- TC.02 - Zalogowanie się (poprawne) na istniejące konto
- TC.32 - Potwierdzenie zamówienia
- TC.38 - Wprowadzenie innego adresu dostawy dla zamówienia
- TC.64 - Płatność (pomyślana) kartą kredytową

Można i warto utworzyć inne scenariusze poprzez dobór innego zestawu przypadków np. zamiast wyszukania po nazwie – zwykłe znalezienie w katalogu, niepomyślne logowanie po przejściu do kaszy

# Przypadek testowy (1)

**Przypadek testowy** – obserwacja działania oprogramowania, związana z interpretacją interesującego nas zdarzenia, danymi, funkcjami

- Powinien zawierać wystarczające dane do przeprowadzenia testu:
  - warunki wstępne (stan przed przeprowadzeniem testu)
  - kolejne kroki realizowanych czynności
  - oczekiwane rezultaty (odpowiedzi systemu i jego stan)
- Możliwe różne poziomy szczegółowości jeśli chodzi np. o dane testowe:
  - a) Wpisz poprawną nazwę użytkownika
  - b) Wpisz „jkowalski”

# Przypadek testowy (2)

ID przypadku:	TC.02			
Nazwa przypadku:	Zalogowanie się (poprawne) na istniejące konto			
Warunki początkowe:	<ol style="list-style-type: none"><li>Użytkownik nie jest zalogowany</li><li>Konto użytkownika zostało uprzednio założone w systemie</li><li>Konto nie zostało uprzednio zablokowane</li></ol>			
Krok:	Czynność:	Oczekiwany rezultat:	Rezultat (Pass/Fail)	Komentarz:
1	Wybierz opcję zalogowania	System prezentuje okno logowania		
2	Wpisz nazwę użytkownika	Wpisana nazwa widoczna w polu „Login”		
3	Wpisz hasło	Wpisane hasło widoczne w postaci zamaskowanej (*) w polu „Hasło”		
4	Potwierdź wybierając przycisk „Zaloguj”	Użytkownik zalogowany do systemu. Dostęp do profilu użytkownika, koszyka i historii transakcji		

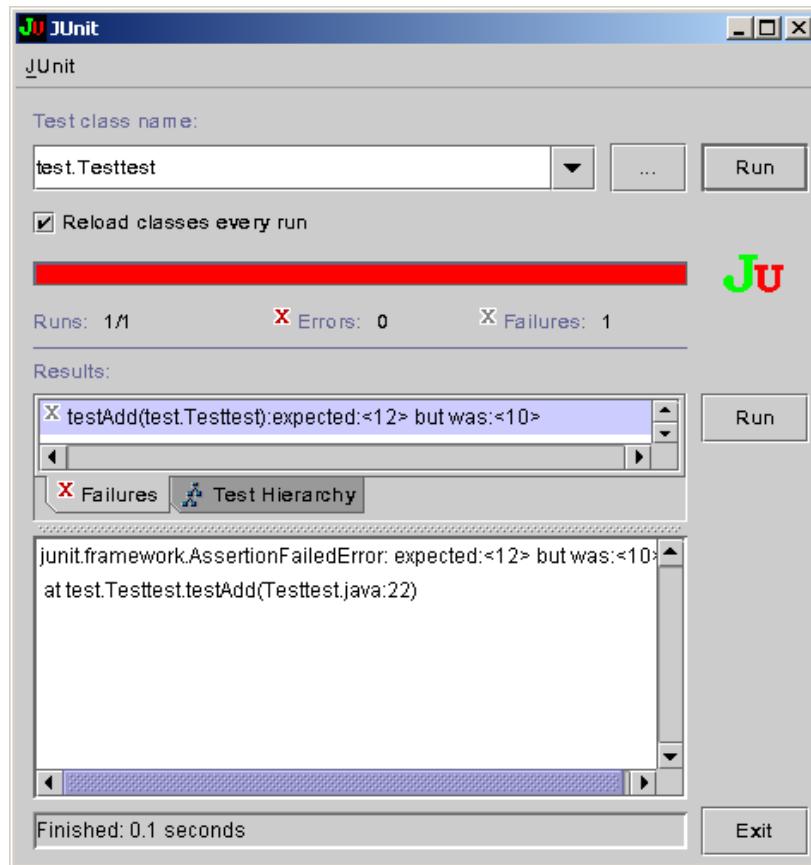
# Przypadki testowe - przykłady

<b>Test Case ID</b>	LC.18
<b>Feature being tested</b>	Editing nodes
<b>Procedure</b>	<p>The test should be performed using editor and admin account.</p> <ul style="list-style-type: none"> <li>• Create a node of every possible type.</li> <li>• Open their descriptions and press apply on them.</li> </ul>
<b>Intended result</b>	Every node should be updated.

<b>Test Case ID</b>	LC.20
<b>Feature being tested</b>	Editing nodes – scripts
<b>Procedure</b>	<p>The test should be performed using editor and admin account.</p> <ul style="list-style-type: none"> <li>• Create a node.</li> <li>• Change every of its fields to:           <ul style="list-style-type: none"> <li>◦ &lt;script&gt;alert('success')&lt;/script&gt;</li> </ul> </li> <li>• Press apply on them.</li> <li>• Logout and login again.</li> <li>• Select the modified node.</li> </ul>
<b>Intended result</b>	<p>No message boxes should be visible.</p> <p>The node label in the tree should have appropriate title.</p>

<b>Test Case ID</b>	LC.22
<b>Feature being tested</b>	Editing nodes – sql injection
<b>Procedure</b>	<p>The test should be performed using editor and admin account.</p> <ul style="list-style-type: none"> <li>• Create a node.</li> <li>• Change every of its fields to:           <ul style="list-style-type: none"> <li>◦ ox"'; drop table node;--</li> </ul> </li> <li>or</li> <li>◦ ox"'; drop table node;-- or other strings of this type.</li> <li>• Press apply on them.</li> <li>• Deselect the modified node.</li> <li>• Select the modified node.</li> </ul>
<b>Intended result</b>	<p>No error should appear.</p> <p>The node label in the tree should have appropriate title.</p>

# Skrypt testowy



```
public class SimpleTest {  
    private Collection<Object> collection;  
  
    @Before public void setUp() {  
        collection = new  
            ArrayList<Object>();  
    }  
    @Test public void testEmptyCollection() {  
        assertTrue(collection.isEmpty());  
    }  
    @Test public void testOneItemCollection() {  
        collection.add("itemA");  
        assertEquals(1, collection.size());  
    }  
}
```

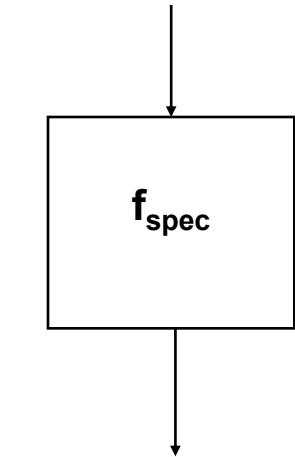
Źródło: <http://www.junit.org>

# Podstawowe techniki testowania

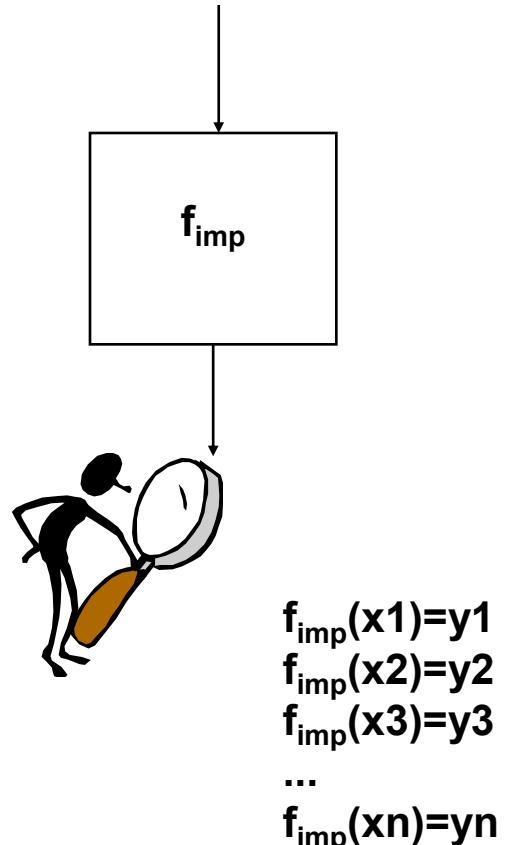
- Testowanie „czarnej skrzynki” (ang. *black box testing*)
  - funkcjonalne – oparte na specyfikacji, system/program widziany z zewnątrz, przez swoje interfejsy (użytkownika, wymiany danych, API)
- Testowanie „białej skrzynki” (ang. *white box testing*)
  - strukturalne – wykorzystuje wiedzę o wewnętrznej strukturze systemu czy też jego konkretnego modułu, o postaci kodu

# Testowanie czarnej skrzynki

specyfikacja:



implementacja:



Odniesienie np. do  
przypadków użycia!

# Testy czarnej skrzynki - podejście

Aby zapewnić systematyczność i możliwą kompletność testów można dążyć do tego, żeby osiągnąć:

- Pokrycie danych wejściowych
- Pokrycie możliwych wyników (danych wyjściowych)
- Pokrycie testowanej funkcjonalności

Przykładowe techniki:

- Podział na klasy równoważności (*equivalence partitioning*)
- Analiza wartości granicznych (*boundary value analysis*)
- Analiza wartości specjalnych (*special value analysis*)
- Tablice decyzyjne (*decision tables*)

# Testowanie czarnej skrzynki

## Podział na klasy równoważności

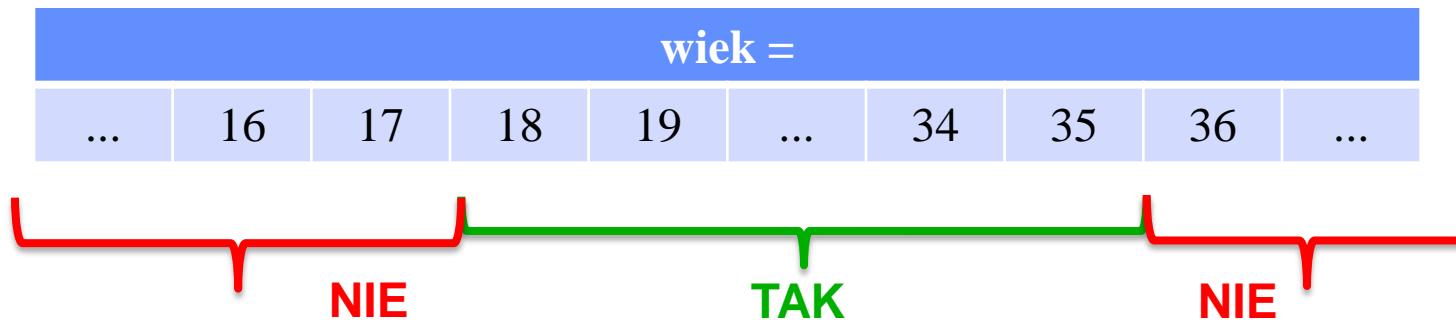
Różne przypadki wykonania programu można pogrupować tak, że poprawne działanie dla reprezentanta grupy oznacza poprawne działanie dla całej grupy

Podział na klasy równoważności może dotyczyć:

- dziedziny wejściowej
- dziedziny wyjściowej

Przykład:

„Do udziału w konkursie uprawnione są osoby w wieku od 18 do 35 lat włącznie”



Otrzymujemy 3 klasy równoważności – teoretycznie wystarczy sprawdzić po 1 przypadku z każdej klasy, w praktyce pewnie więcej, ale mamy istotną wskazówkę, jak zaplanować przypadki testowe

# Testowanie czarnej skrzynki

## Analiza wartości granicznych

- Powiązane koncepcyjnie z klasami równoważności
- Dodatkowo jednak bazuje na obserwacji, że błędne zachowania programu najczęściej występują na granicach dziedziny wejściowej
- Stąd wskazówki dla doboru testów:
  - jeżeli dana wejściowa przebiega zakres między X i Y, to testuj dla X, dla Y oraz dla wartości bezpośrednio sąsiadujących z X i Y
  - jeżeli dziedzina wejściowa obejmuje skończony zbiór uporządkowanych wartości, to testuj dla wartości MAX i MIN oraz w ich bezpośrednim sąsiedztwie
  - powyższe zasady należy również zastosować do wartości wyjściowych, np. aby wyprodukować wartość MAX i MIN na wyjściu
  - jeżeli zadano ograniczenia pojemności (ilość danych, obciążenie), to powinny być one również przetestowane dla ich wartości granicznych

Dla przykładu z poprzedniego slajdu należały zatem sprawdzić:

- przynajmniej 17, 18 oraz 35, 36, a pewnie lepiej 16, 17, 18, 19 oraz 34, 35, 36, 37

# Testowanie czarnej skrzynki

## Metody specjalnych wartości

Sprawdzenie działania programu dla wartości granicznych, wyjątkowych, pozwalających jak najlepiej scharakteryzować badaną funkcję

Wartości graniczne z poprzedniego slajdu plus dodatkowe wartości, które potencjalnie wypada sprawdzić np.

- **operacje arytmetyczne**
  - jedynka, zero, MIN i MAX wartości operandów, liczby pierwsze
- **konwersje typów**
  - podstawienia, argumenty wywołania, wartości zwracane, konwersje niejawne, konwersje jawne
- **tablice**
  - obiekt pusty, prawidłowe i nieprawidłowe wartości indeksów
- **listy**
  - lista pusta, lista jednoelementowa, lista wieloelementowa

# Testowanie czarnej skrzynki

## Tablice decyzyjne

- Stosowane w celu pełnego pokrycia testami wszystkich możliwych przyczyn i skutków dla programu (lub jego części/aspektu), dodatkowym zyskiem może być identyfikacja niepełnych wymagań

	W1	W2	W3	W4	W5	W6	W7	W8	W9
<b>WARUNKI WEJŚCIA</b>									
Aktualny klient banku	N	T	T	T	T	T	T	T	T
Aktywa powyżej 75000 PLN	*	N	N	N	N	T	T	T	T
Zaległe zobowiązania wg BIK	*	N	N	T	T	N	N	T	T
Deklarowane dochody powyżej 9000 PLN	*	N	T	N	T	N	T	N	T
<b>AKCJE</b>									
Przedstawić indywidualną ofertę kredytu	N	N	T	N	N	T	T	N	N
Przydzielić doradcę osobistego	N	N	N	N	N	N	T	T	T

# Testowanie białej skrzynki (strukturalne)

- **Testowanie z wykorzystaniem wiedzy o strukturze wewnętrznej badanego programu / modułu**
  - wykonanie wszystkich niezależnych ścieżek sterowania w programie
  - wykonanie wszystkich decyzji dla instrukcji warunkowych
  - wykonanie wszystkich pętli w programie dla warunku granicznego (0 razy, Max razy) oraz dla przypadku w założonych granicach powtórzeń pętli ( $0 < n < \text{Max}$ )
  - wykonanie dostępu do wszystkich wewnętrznych struktur danych

# Testowanie białej skrzynki

## Testowanie ścieżek

- Struktura przepływu sterowania w programie może być reprezentowana w postaci *grafo przepływu sterowania*
- *Niezależna ścieżka* - różni się co najmniej jedną instrukcją lub warunkiem od innych ścieżek
- *Liczba cyklowatyczna* programu - podaje liczbę niezależnych ścieżek w grafie przepływu sterowania programu

**Liczba cyklotomyczna jest równa  
liczbie rozłącznych obszarów w  
grafie przepływu sterowania**

$$C = E - N + 2$$

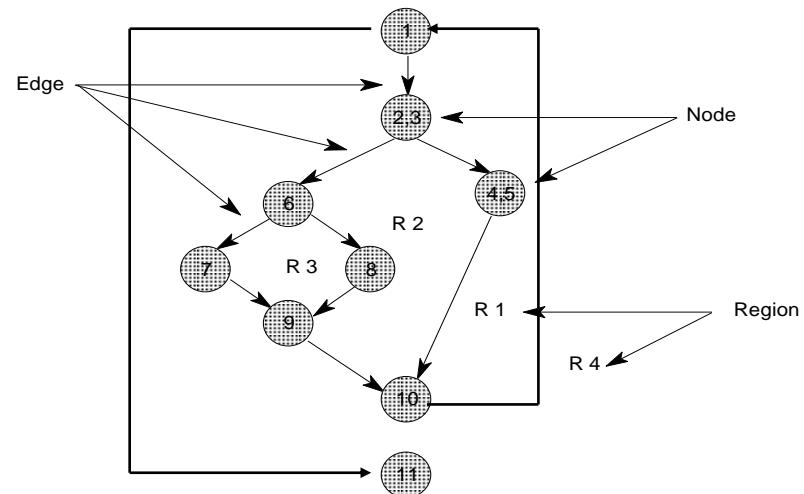
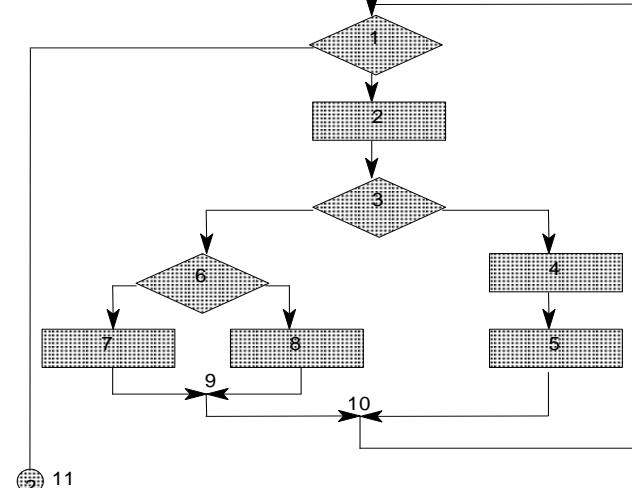
gdzie

E – liczba łuków

N – liczba węzłów

$$C = PR + 1$$

PR – liczba węzłów decyzyjnych  
(warunki logiczne)



INDEPENDENT PATHS

P 1: 1 - 11

P 2: 1-2-3-4-5-10-1-11

P 3: 1-2-3-6-8-9-10-1-11

P 4: 1-2-3-6-7-9-10-1-11

$$C=4$$

# Testowanie białej skrzynki

## Testowanie ścieżek - budowa przypadków testowych

- zbuduj graf przepływu sterowania
- określ złożoność cykłomatyczną
- dobierz zestaw niezależnych ścieżek
- dobierz przypadki testowe (dane wejściowe), które wymuszają wykonanie tych ścieżek

**Potrzebne wspomaganie narzędziowe!**

**Pokrycie testami:**

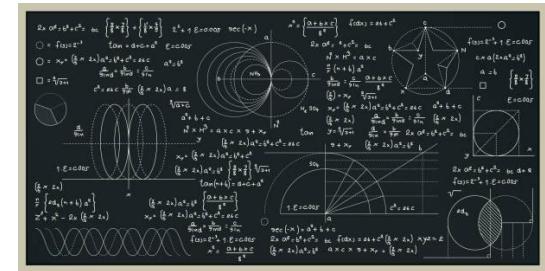
$$Tc = N/C * 100\%$$

**Tc – pokrycie (procent przetestowanych ścieżek)**

**N – liczba przetestowanych niezależnych ścieżek**

**C – liczba cykłomatyczna badanego programu**

# Problem: Jak wykazać poprawność testowanego programu?



- Pomijając trywialne przypadki, nigdy nie da się przetestować wszystkich możliwości dla wszystkich możliwych danych wejściowych i warunków środowiskowych
  - Dla krótkiego programu wykonującego jakąś operację matematyczną na dwóch liczbach Integer (32 bity), należałyby wykonać  $1,84467 * 10^{19}$  testów
- Testowanie nie jest w stanie wykazać poprawności programu - może tylko pokazać jego błędy, a więc obecność w nim defektów
- Pewna analogia do wykazywania poprawności twierdzeń matematycznych:
  - Znalezienie dowolnie dużej liczby przypadków, dla których twierdzenie jest spełnione, nie powoduje jego udowodnienia
  - Znalezienie dowolnego kontr-przykładu obala twierdzenie

# Problem: Kto ma testować?



- **Deweloper, który jest autorem kodu, najlepiej zna jego budowę, ale:**
  - Niekoniecznie musi być zainteresowany wykazaniem własnych pomyłek
  - Częsty pogląd: wytwarzanie - postawa twórcza, testowanie - postawa destrukcyjna, testowanie to nadmierny krytycyzm etc.
  - Jeżeli autor pomylił się przy projektowaniu czy implementacji, to może powtórzyć tę samą pomyłkę przy testowaniu (np. pominięcie jakiejś specyficznej sytuacji)
- **Strategie praktyczne - podejście mieszane (dla różnych rodzajów testów lub/i poziomów testowania):**
  - deweloper
  - ktoś niezależny (odpowiednio motywowany np. opłacany za znajdowane defekty, obejmuje to również crowdsourcing)

# Problem: Kiedy zakończyć testowanie?

- Co może oznaczać sytuacja kiedy testuję i nie znajduję (już) żadnych błędów?
- Ile defektów zostało w systemie?
- Możliwe kryteria zakończenia?
  - Koniec zasobów (zmęczenie, deadline)
  - Wykonanie planu testowania
  - „Dostateczne” pokrycie przypadkami testowymi
  - Szacowanie przez analogię
  - Dynamika zgłoszeń znalezionych błędów
  - Zasiewanie defektów



Fajrant?

# Testowanie – inne problemy

- Czy z każdym znalezionym i poprawionym defektem jakość programu/systemu się zwiększa?
- Maskowanie defektów
- Testy regresji
- Paradoks pestycydów



# Test Driven Development (TDD)

- Cały proces wytwarzania (nie tylko testowania!) bardzo mocno oparty o testy
- Testy powstają przed kodem
  - Najpierw pisane są przypadki testowe (zautomatyzowane, wykonywane jako skrypty testowe)
  - Takie przypadki testowe stanowią więc szczegółową reprezentację wymagań, do ich napisania konieczne jest dobre przemyślenie problemu
  - Kiedy powstanie kod – poddawany jest testom
  - Kiedy kod podlega późniejszej ewolucji – dzięki testom można sprawdzać czy nie przestał realizować wymagań
- TDD skupia się na automatycznych testach jednostkowych. Dobrą praktyką jest jednak wczesne specyfikowanie różnych testów np.
  - Testy akceptacyjne przy specyfikowaniu wymagań
  - Testy integracyjne przy projekcie architektury

# Testowanie mutacyjne



- **Jak stwierdzić, że nasz zestaw testów automatycznych jest skuteczny?**
- **Jedną z technik jest tzw. **testowanie mutacyjne****
- **Jest to „testowanie testów”, nie programu!**
- **Generowane są mutanty programu (klasy, modułu itp.)**
- **Mutant to program po niewielkiej transformacji np.**
  - zmianie operatora np. + na - , < na <=
  - zmianie jakiejś wartości np. iteratora w pętli
  - usunięciu jednej linii
- **Sprawdzenie czy mutant przeżyje tzn. przejdzie pomyślnie testy...**

# Testowanie oprogramowania

*Testowanie nie zastąpi dobrej konstrukcji programu !*

*Jakości nie da się wbudować w program wyłącznie poprzez testowanie !*

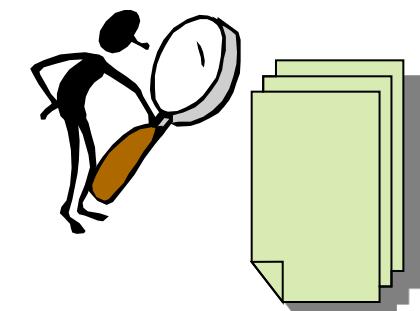


# Analiza statyczna

- **Testowanie (analiza dynamiczna) wymaga eksperymentowania z działającym (wykonywalnym) kodem programu**
- **Możliwe jest jednak poszukiwanie defektów innymi sposobami**
  - Analiza kodu źródłowego za pomocą narzędzi (choćby kompilator, ale i bardziej specjalistyczne narzędzia)
  - Formalne dowodzenie poprawności programów (modele matematyczne, symulacje)
  - Działania wykonywane przez ludzi (przeglądy, inspekcje)

# Przeglądy (ang. reviews)

- Nie wymagają wykonywania kodu, ponadto ich zastosowanie może dotyczyć nie tylko kodu źródłowego, ale wszelkich produktów procesu wytwarzania oprogramowania
  - specyfikacje wymagań, modele analityczne i projektowe, plany testów, dokumenty menedżerskie, ...
- Przeglądy – kontrole artefaktów ukierunkowane na wykrywanie defektów prowadzone przez ludzi
- Spotykane jest również tłumaczenie *review* jako *recenzja*



# Dlaczego przeglądy? - Motywacja

- Ludzie popełniają błędy i wprowadzają defekty do tworzonych przez siebie artefaktów
- Mechanizacja wykrywania i usuwania tych defektów jest możliwa jedynie w ograniczonym zakresie
- Niezbędny jest więc wysiłek człowieka w celu wykrycia i usunięcia tych defektów
- Idea przeglądów:
  - Najtrudniej jest zauważać własne błędy, dlatego dobrze jest, by poszukał ich ktoś inny



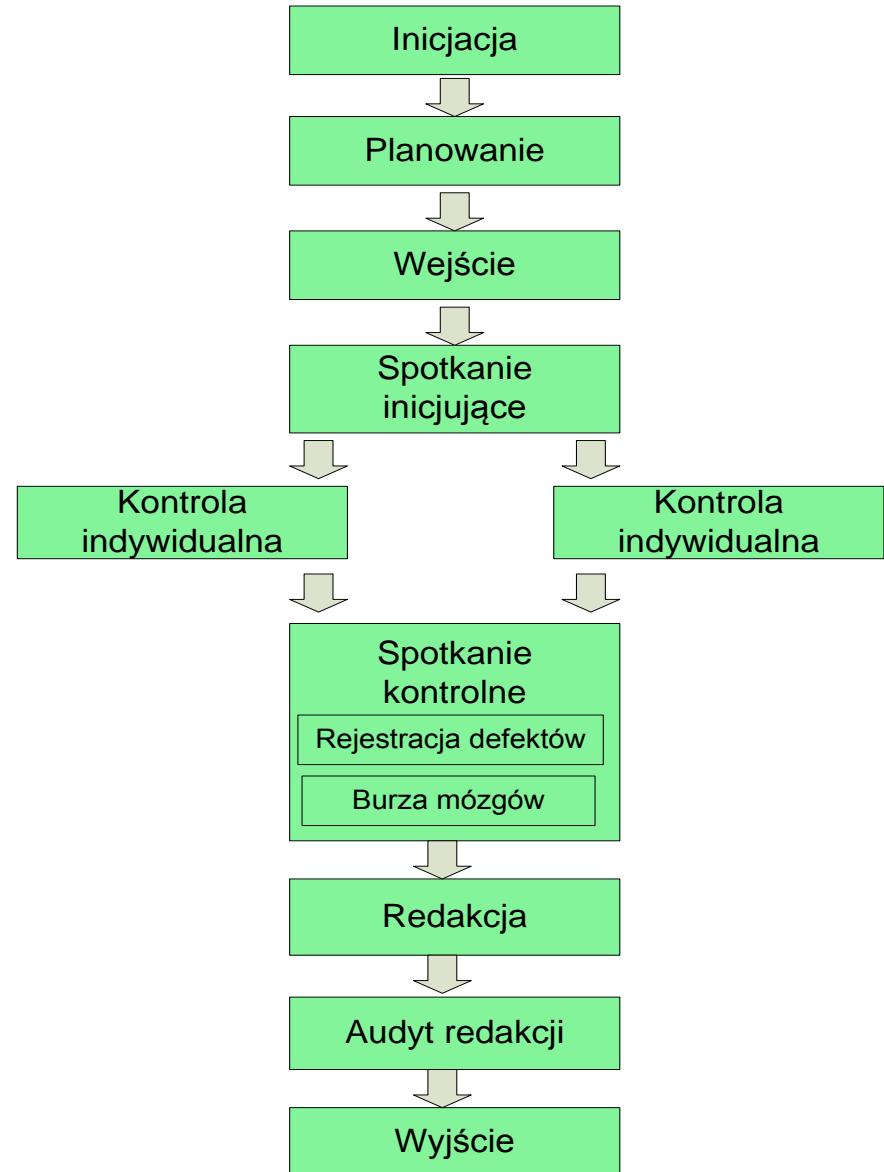
# Przeglądy

- **Przegląd to działanie zespołowe, ukierunkowane na:**
  - wykrycie (szeroko rozumianych) defektów w dokumencie, którego autor jest członkiem zespołu
  - potwierdzenie dobrej jakości dokumentu w zakresie nie wymagającym poprawy
- **Przegląd danego produktu polega na:**
  - przeanalizowaniu go (indywidualnie lub/i zespołowo) przez grupę osób niezaangażowanych w jego wytworzenie,
  - rejestrację zauważonych defektów i innych wątpliwości,
  - przekazanie wyników autorowi, który dokonuje niezbędnych poprawek



# Inspekcje oprogramowania

- **Inspekcja - najbardziej sformalizowany rodzaj przeglądu**
  - Zdefiniowane mechanizmy detekcji
  - Sformalizowany proces
  - Ustalone parametry inspekcji (np. tempo kontroli)
  - Statystyczna kontrola procesu
- **Inspekcje mają zastosowanie w bardziej uporządkowanych i zarządzanych organizacjach i procesach wytwarzania**
- **W mniejszych przedsięwzięciach wystarczające mogą być zwykłe przeglądy koleżeńskie**



# Mechanizmy detekcji defektów

- **Ad hoc – brak wspomagania, ale i brak ograniczenia zakresu**
- **Lista kontrolna (ang. *checklist*) – lista zagadnień do sprawdzenia przez inspektora**
  - Czy wszystkie rozmiary tablic są zadeklarowane jako stałe?
  - Sprawdź czy wszystkie zmienne są zainicjalizowane
- **Scenariusze do wykonania (ang. *scenario-based reading*) – scenariusze zgodnie z którymi ma działać inspektor**
  - Dodatkowe zadania mające za cel umożliwienie inspektorowi lepszego zrozumienia kontrolowanego artefaktu
  - Przykład: *Perspective-Based Reading* – kontrola specyfikacji wymagań z perspektywy projektanta, testera, użytkownika

# Przykłady list kontrolnych (1)

## NASA Formal Inspections Guidebook Software requirements checklist (*fragment*)

### LEVEL OF DETAIL

1. Are the requirements free of design?
2. Have all "TBDs" been resolved?
3. Have the interfaces been described to enough detail for design work to begin?
4. Have the accuracy, precision, range, type, rate, units, frequency, and volume of inputs and outputs been specified for each function?
5. Have the functional requirements been described to enough detail for design work to begin?
6. Have the performance requirements been described to enough detail for design work to begin?

# Przykłady list kontrolnych (2)

## C++ Inspection Checklist by Christopher Fox (*fragment*)

### Variable and Constant Declaration Defects (VC)

- Are descriptive variable and constant names used in accord with naming conventions?
- Are there variables with confusingly similar names?
- Is every variable correctly typed?
- Is every variable properly initialized?
- Could any non-local variables be made local?
- Are there literal constants that should be named constants?
- Are there macros that should be constants?
- Are there variables that should be constants?

### Storage Usage Defects (SU)

- Is statically allocated memory large enough?
- Is dynamically allocated memory large enough?
- Is all dynamically allocated memory freed, and freed when appropriate
- Is there a free for every *malloc* and a delete for every new?

# Modele cyklu życia oprogramowania

*Aleksander Jarzębowicz*

*Katedra Inżynierii Oprogramowania  
Politechnika Gdańskia*

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich rozpowszechnianie  
jest zabronione.

# Główne obszary IO



autor: J.Miler

# Obszary a proces

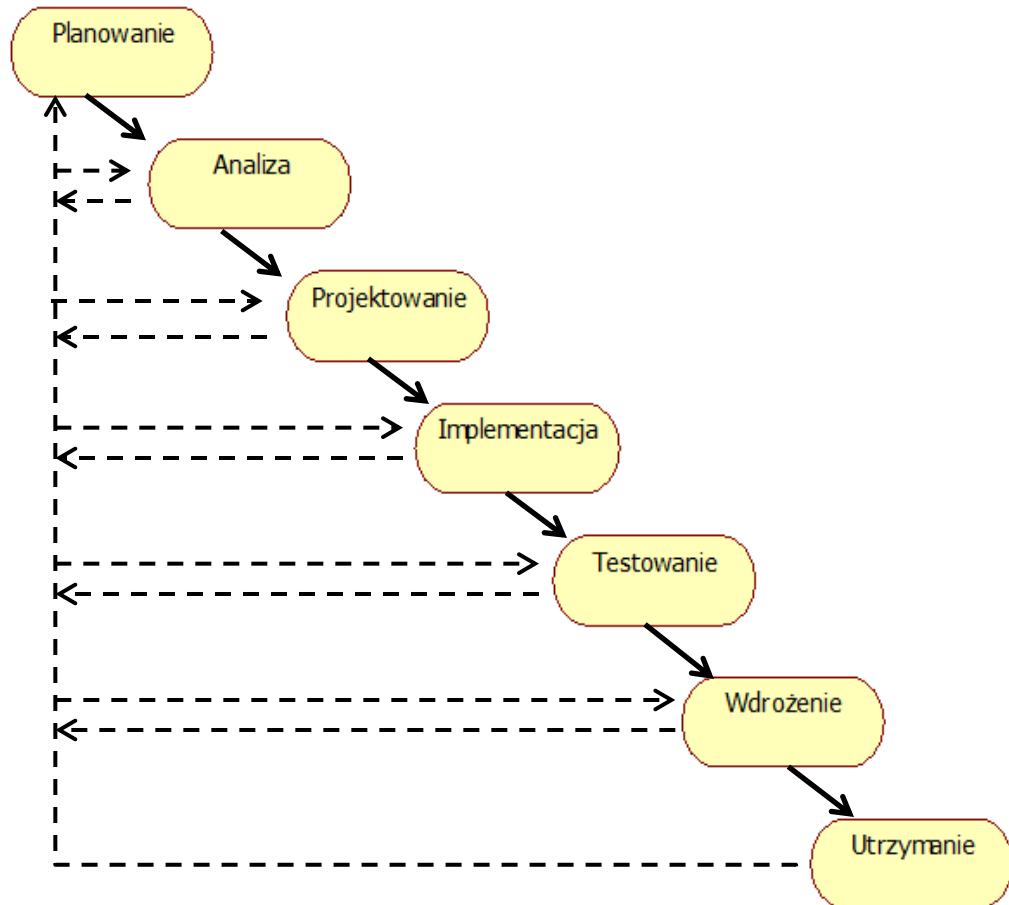
- Jak „poskładać” działania w poszczególnych obszarach IO w sensowną całość?
- Jak „zgrać” to z zagadnieniami zarządzania i zapewniania jakości?
- Podejście „na żywioł” (ang. *cowboy coding, code & fix*)
  - problemy w pracy zespołowej (>1 deweloper)
  - problemy z współpracą z interesariuszami
  - trudność z zapewnieniem jakości projektu i kodu
  - źródło Software Crisis i impuls do rozwoju IO
- Różne propozycje modeli wytwarzania

# Pojęcia

- **Cykl życia oprogramowania / model cyklu życia / model wytwarzania** (*różne funkcjonujące nazwy, dalej stosowane zamiennie*) – ogólna metoda organizacji procesu wytwarzania systemu, zawierająca podział na etapy i kryteria przechodzenia między etapami, może również sugerować produkty i praktyki
- **Metodyka** – konkretna metoda, bazująca na jednym lub kilku modelach wytwarzania, definiująca szczegółowo etapy, praktyki, produkty, role, zwykle firmowana przez określonego autora/gremium

# Klasyczny (kaskadowy, ang. waterfall) model cyklu życia

- Wyróżnione i odseparowane etapy odpowiadające poszczególnym podstawowym obszarom IO
- W każdym etapie przedmiotem zainteresowania jest cały wytwarzany system np.
  - robimy projekt architektoniczny i szczegółowy całego systemu
  - implementujemy całość
  - testujemy i wdrażamy cały system
- Etapy ustawione są sekwencyjnie, kolejny zaczyna się po zamknięciu poprzedniego
  - np. zaczynamy implementację, kiedy skończyliśmy już całe projektowanie
- Jeżeli we wcześniejszym etapie zrobiono coś źle (np. ujawni się błąd czy problem z wymaganiami) powinno się tam wrócić, dokonać korekty i ponownie przejść wszystkie pośrednie etapy
  - w ten sposób np. wykryty przy testowaniu błąd wynikający z niezrozumienia wymagań klienta wymaga powrotu do etapu analizy



# Zalety modelu kaskadowego

- Pierwsza propozycja reakcji na *Software Crisis*
- Łatwy do zrozumienia / wytłumaczenia
- Obejmuje wszystkie obszary IO i wprowadza systematykę
- Narzuca dobre praktyki - dobre zrozumienie problemu (analiza) i zdefiniowanie architektury (projekt) przed implementacją
- Pozwala na wyraźne zdefiniowanie etapów (np. możliwe punkty kontrolne – *milestones* i śledzenie postępów)
- Pozwala na dekompozycję pracy dla różnych ról (np. projektant, programista, tester) oraz rozłożenie ich zaangażowania w czasie
- Powstały oparte na nim standardy z konkretnymi metodykami

# Problemy modelu kaskadowego

- Zakłada, że od razu będzie możliwe uchwycenie całego problemu i wymagań, co zwykle jest trudne – w trakcie całego projektu „odkrywane” są kolejne szczegóły
- Zakłada, że pozyskane na początku projektu wymagania będą stabilne i nie będą podlegały zmianom – w praktyce rzadko kiedy jest to możliwe
- Działające oprogramowanie powstaje późno, pod koniec projektu, wcześniej nie ma niczego do pokazania interesariuszom
- Wytwarzanie jest w ogólności oderwane od interesariuszy - pomiędzy analizą a walidacją produktu w końcowych krokach (testowanie, wdrożenie) brak kontaktu
- Błąd lub żądanie zmiany powoduje konieczność powrotu do odpowiedniego etapu i przejścia procesu jeszcze raz



Modele nieklasyczne

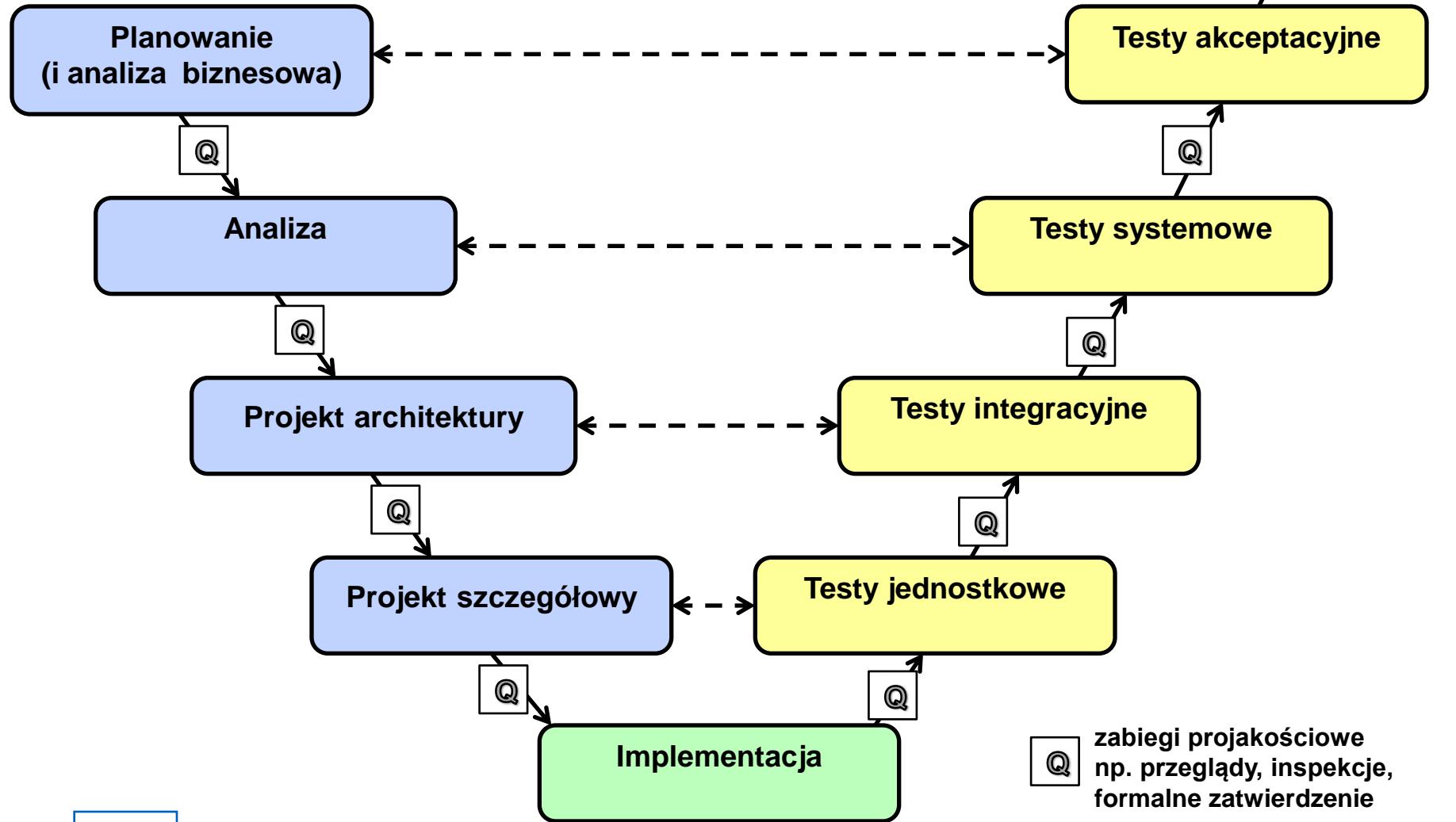
# Model kaskadowy - filmik

- **Z przymrużeniem oka, ale jednocześnie prawdziwie...**
- **<https://vimeo.com/18951935>**

# Modele nieklasyczne

- **Model kaskadowy (klasyczny) był propozycją odpowiadającą na „software crisis” (czyli w odległych czasach) i stanowił znaczny postęp**
- **Nie należy go traktować jako podstawowego obowiązującego w praktyce przemysłowej!**
- **Nie należy jednak zupełnie go skreślać – może być warty zastosowania gdy np.:**
  - jest gwarancja niezmienności wymagań
  - mamy bardzo dobrze rozpoznaną dziedzinę np. to kolejny projekt bardzo podobny do poprzednich
- **Z uwagi na znaczące problemy związane z modelem klasycznym, zaproponowane zostały inne modele, eliminujące/minimalizujące przynajmniej niektóre jego problemy**

# Model V



# Ocena modelu V



## Zalety:

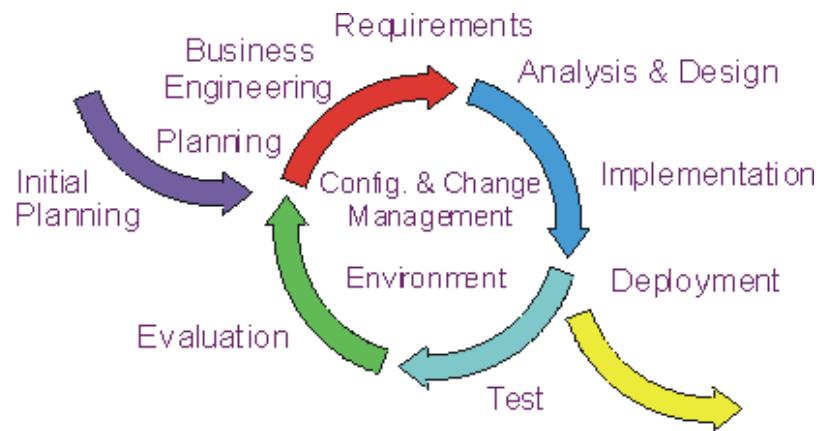
- + Model przenosi wszystkie zalety modelu kaskadowego
- + Jest ukierunkowany na wysoką jakość wytwarzanego produktu (liczne zabiegi projakościowe, testy prowadzone na wielu poziomach)
- + Obniża ryzyko popełnienia dużego błędu

## Wady:

- Przejmuje też większość wad modelu kaskadowego – założenie o dobrze zidentyfikowanych i stabilnych wymagań, małą elastyczność, oderwanie od interesariuszy
- Znaczne narzuty pracy, czasu i kosztów poświęcone na testy i jakość
- Silne rozbudowanie dokumentacji
- W razie zmian też trzeba się cofać, a jeszcze na dodatek modyfikować testy

# Podejście iteracyjne

- Idea: zamiast jednego „przebiegu” jak w modelu kaskadowym, wykonywanych jest kilka iteracji
- W ramach każdej iteracji powstaje jakąś (tymczasowa, nie ostateczna) wersja produktu
- W każdej iteracji system jest rozbudowywany lub/i udoskonalany
- Umożliwia to:
  - uczenie się deweloperów w trakcie przedsięwzięcia (zrozumienie potrzeb),
  - sprzężenie zwrotne od użytkowników i innych interesariuszy
  - dostosowanie produktu do potrzeb (również tych zmieniających się w trakcie przedsięwzięcia)
  - poprawianie defektów w kolejnej iteracji

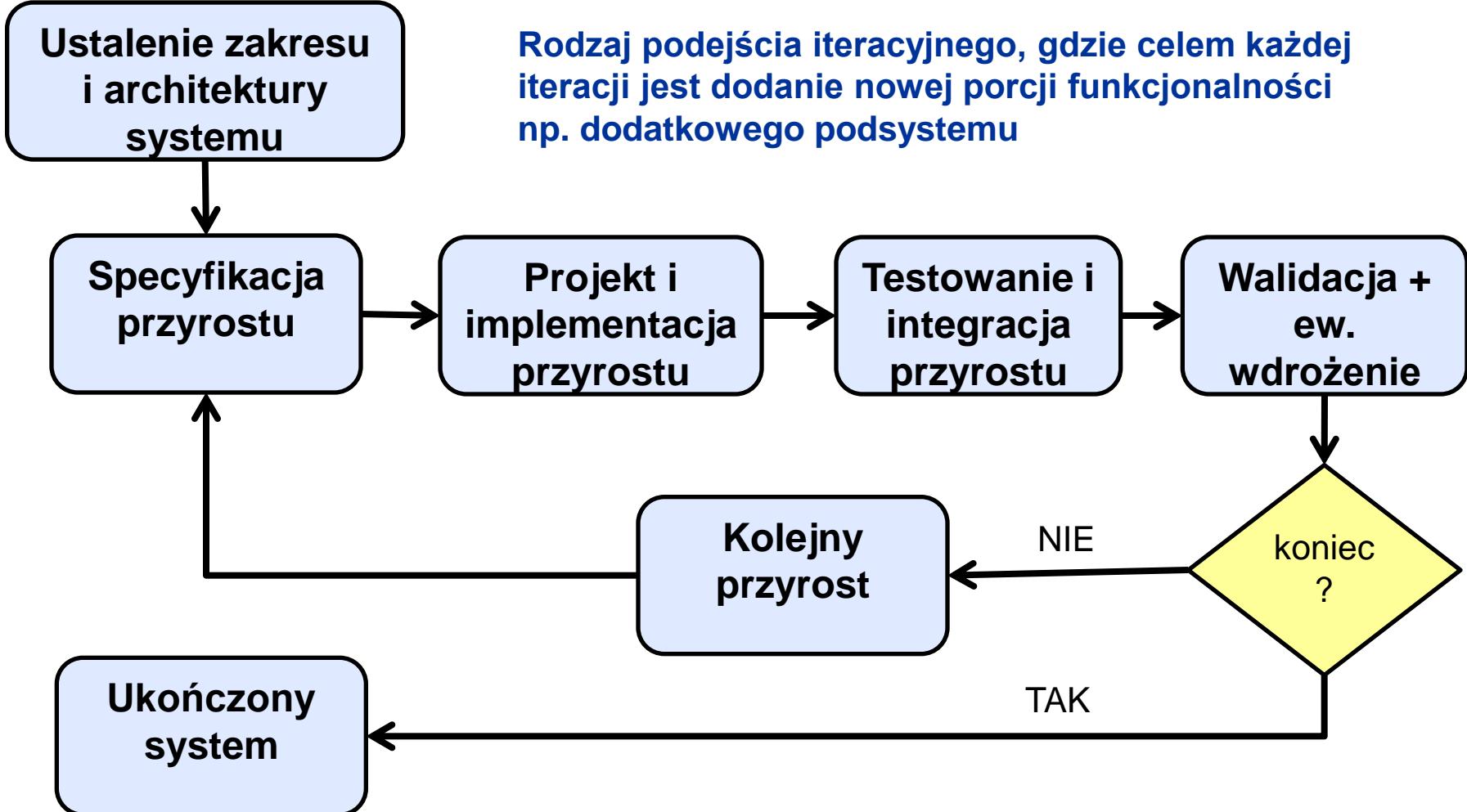


Źródło: Specyfikacja RUP

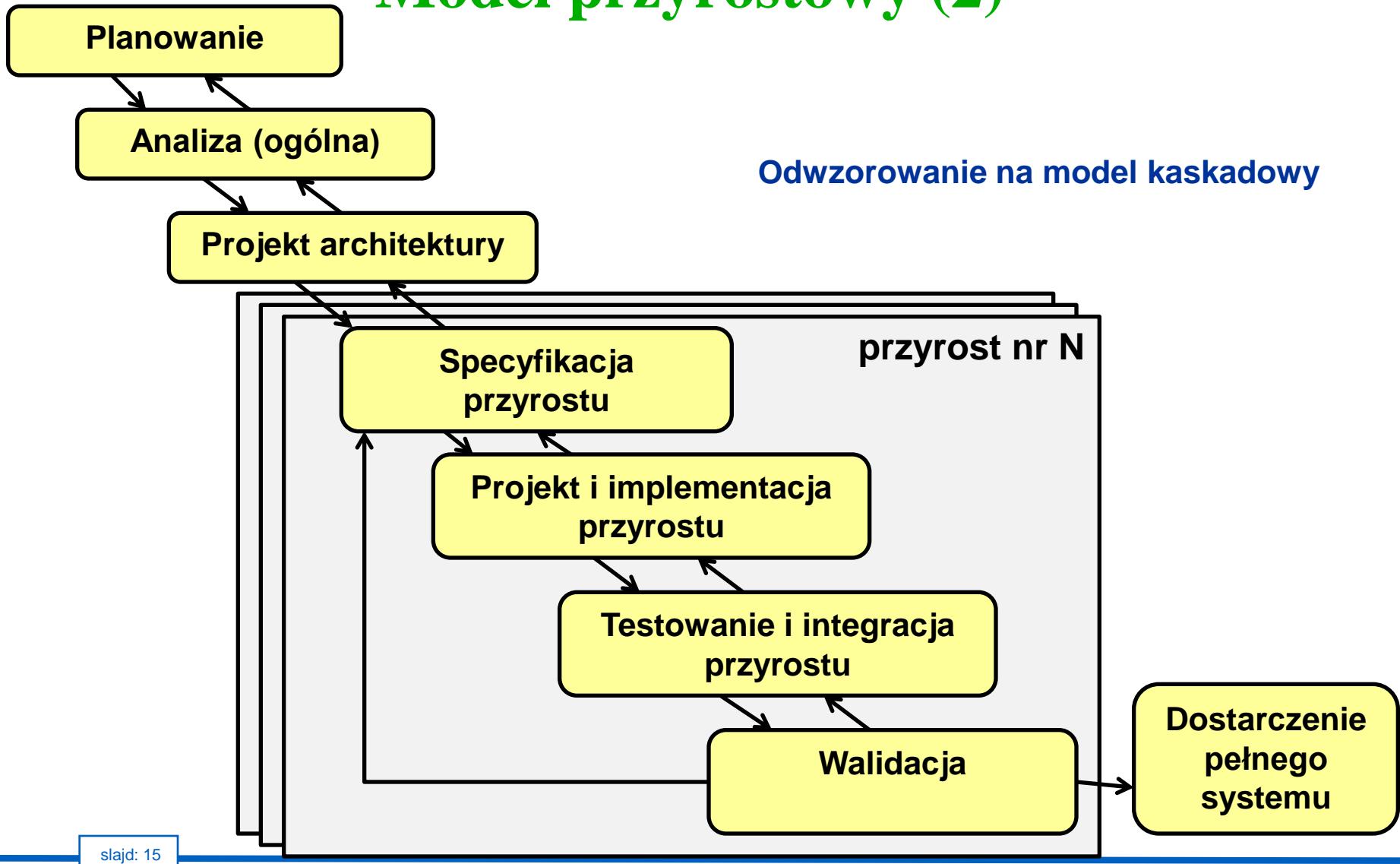
## Podejście iteracyjne - możliwości

- Zgodnie z nazwą projekt podzielony jest na iteracje
- Jednak w zależności od tego, co będzie wykonywane w tych iteracjach, można wyróżnić konkretniejsze modele wytwarzania:
  - dodanie kolejnej porcji funkcjonalności -> model przyrostowy
  - wizualizacja wymagań i ich lepsze zrozumienie -> model prototypowy
  - zaadresowanie największego ryzyka -> model spiralny

# Model przyrostowy (1)

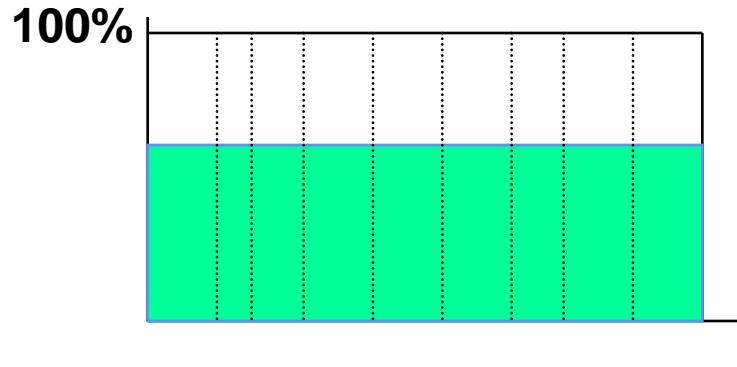


# Model przyrostowy (2)



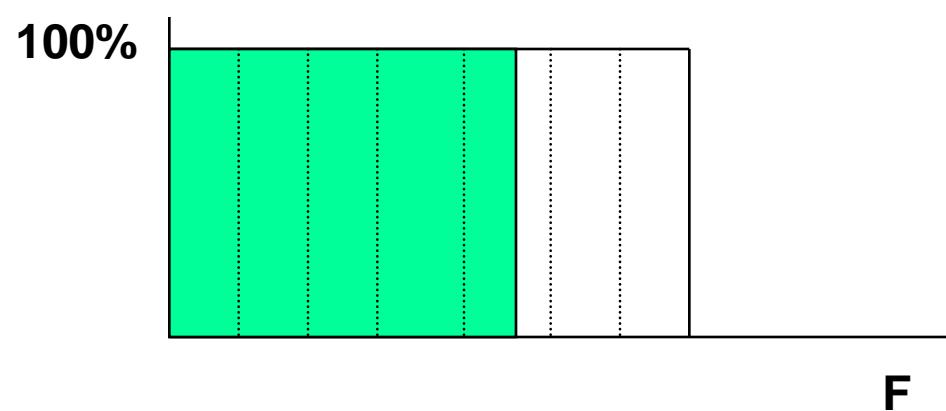
# Przyrostowa budowa systemu

model kaskadowy



W chwili T, 100% systemu  
jest ukończone w 60%

model przyrostowy



W chwili T, 60% systemu  
jest ukończone w 100%

F - funkcjonalność systemu  
T - czas

## Zalety modelu przyrostowego (1)

- **Uniknięcie wielu problemów występujących w podejściu kaskadowym poprzez:**
  - redukcję kosztu zmian w wymaganiach kolejnych modułów/podsystemów
  - uniknięcie jednorazowej integracji („big bang problem”)
  - równomierny rozkład nakładów na testowanie
  - możliwość poprawy procesu w ramach projektu
  - lepsze szacowanie projektu
- **Stabilizacja wymagań na poziomie realizacji danego przyrostu**

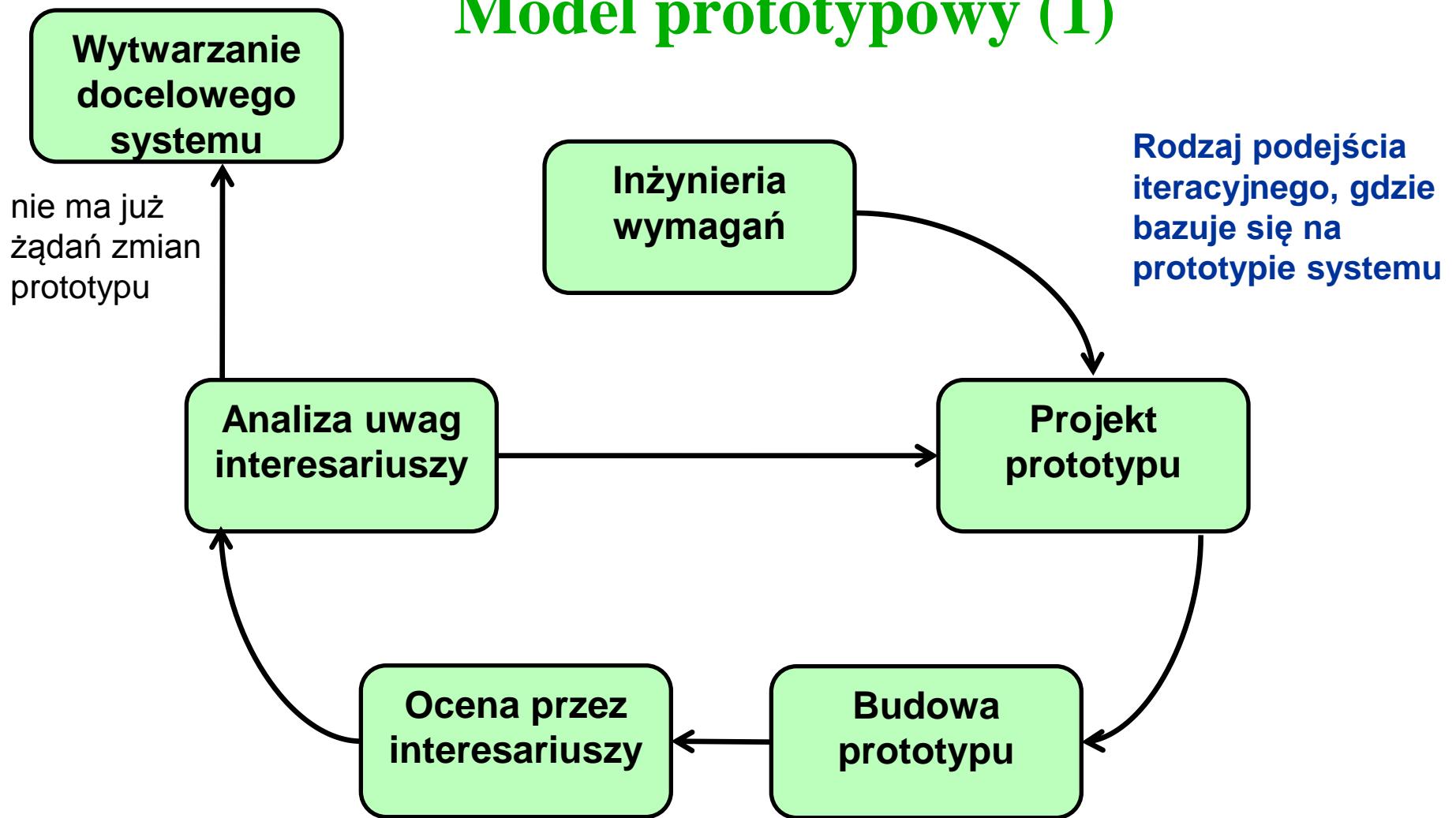
## Zalety modelu przyrostowego (2)

- Nie ma konieczności pełnej specyfikacji wymagań *a priori*, wystarczy zarys, konkretniej specyfikowany jest tylko najbliższy przyrost
- Każdy przyrost:
  - dodaje nowe funkcje
  - może być realizowany oddzielnie
  - może być oddzielnie testowany
  - może być zrealizowany w krótszym okresie (np. kilku miesięcy)
  - intensywność prac można regulować
- Namalczany produkt (część systemu) powstaje szybciej
- Interesariusze są mocniej wciągnięci w proces

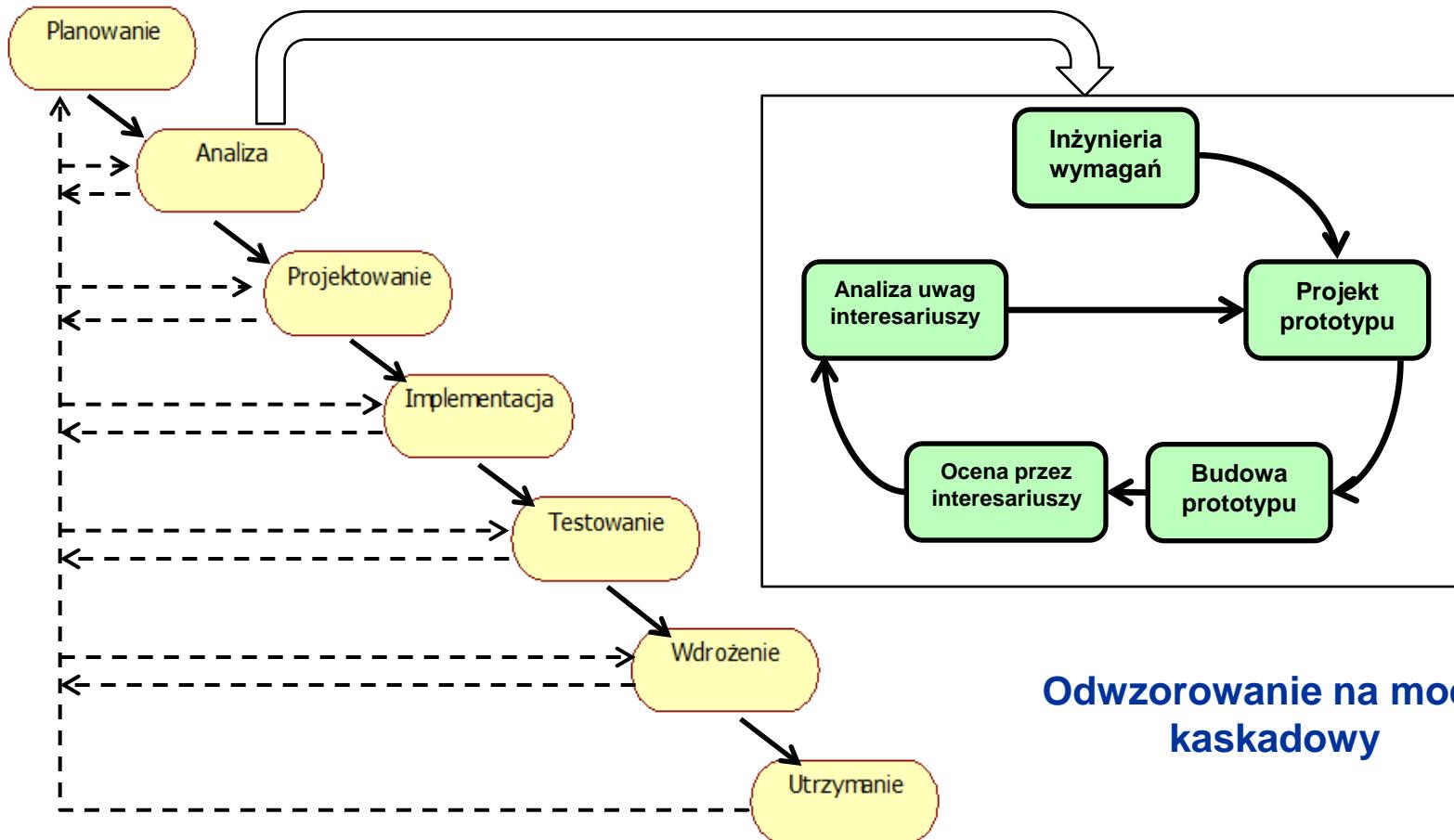
## Model przyrostowy - wady

- Przydatny raczej tylko dla systemów, w których dopuszcza się podzbiory funkcji (możliwość dekompozycji na podsystemy)
- Dojście do rozwiązania docelowego bardziej długotrwałe i kosztowne (wiele iteracji) niż dla „bezbłędnego przebiegu” modelu kaskadowego
- Przy szczegółowej analizie danego przyrostu może się jednak okazać, że inicjalnie określony zakres systemu lub/i jego projekt architektury są niewystarczające

# Model prototypowy (1)



# Model prototypowy (2)



# Prototypowanie

- Podstawowym celem prototypowania jest identyfikacja wymagań poprzez budowę kolejnych przybliżeń systemu
- Możliwości:
  - Model „papierowy” - bazuje na rysunkach interfejsów systemu (np. rysunki ekranów pokazywane użytkownikowi)
  - Model „symulowany” - analityk odgrywa rolę systemu i informuje o jego kolejnych reakcjach na działania użytkownika, może również wspomagać się rysunkami
  - Model „programowy” – oprogramowanie napisane specjalnie w celu demonstracji użytkownikowi określonych cech docelowego systemu (najczęściej interfejsu użytkownika i uproszczonej funkcjonalności)
  - Model „ewolucyjny” - częściowo wykonany system docelowy, posiadający najpilniejsze (z punktu użytkownika) cechy systemu docelowego, który będzie podlegał kolejnym rozszerzeniom zmierzającym do uzyskania pełnej wersji systemu docelowego – **ale to już bardziej inne podejście (ewolucyjne)**
- Poza prototypowaniem wymagań, możliwe jest również prototypowanie konstrukcji (*proof of concept*) – wtedy poświęcone mu iteracje mają miejsce w ramach projektowania

prototypy „dowyrzucenia”

# Model prototypowy - zalety

- Wspomaganie identyfikacji wymagań
- Zwiększenie zaangażowania interesariuszy w procesie wytwarzania
- Lepsza walidacja systemu, możliwość odniesienia się interesariuszy do przedstawianego prototypu
- Możliwość oceny i doboru alternatywnych rozwiązań
- Poprawa cech jakościowych
  - łatwości użycia, ergonomii, dopasowania do potrzeb (prototypowanie wymagań)
  - jakości architektury, utrzymywalności (prototypowanie konstrukcji)

## Model prototypowy - problemy

- Budowa i wprowadzanie zmian do prototypu pochłaniają czas i koszty, co może budzić opór, zwłaszcza jeśli prototyp jest „do wyrzucenia”
- Projektant może „przyzwyczać się” do rozwiązań, które zastosował w prototypie i użyć ich w docelowym systemie, podczas gdy niekoniecznie gwarantują one spełnienie wszystkich wymagań (np. jakościowych typu wydajność)
- Klient widzi coś, co w jego przeświadczenie jest działającym systemem i ma problemy w zrozumieniu, dlaczego ma on wymagać jeszcze dalszej pracy (albo wręcz być zarzucony i budowany od nowa – w przypadku prototypów „do wyrzucenia”)

# Model spiralny



Pierwsza propozycja podejścia iteracyjnego, dość specyficzna, oparta na ryzyku

## Model spiralny - idea

- Strategia zmniejszająca ryzyko
- Cykliczność faz wytwarzania, uwzględnianie oceny interesariuszy (walidacja dotychczas wykonanych prac) w dalszych planach
- Najogólniejszy model wytwarzania, wg autora (Barry'ego Boehma) mieści w sobie pozostałe modele jako specjalne przypadki

Przydatny dla przedsięwzięć obarczonych dużym ryzykiem, gdzie konieczna jest ocena stanu, analiza zagrożeń i dostosowanie na tej podstawie dalszych działań

### Przykład:

Oprogramowanie opracowywane na szeroki rynek, w warunkach dużej konkurencji może wymagać działań wg modelu spiralnego.

W kolejnych obrotach spirali przedmiot zainteresowania mogłyby stanowić np. dodanie kluczowej funkcjonalności, która jest już w konkurencyjnym produkcie albo udoskonalenie części systemu postrzeganej przez użytkowników jako bardzo „denerwująca”.

## Model spiralny - zalety i wady

### Zalety:

- **Jawne wskazanie ryzyka i jego analizy**
- **Działania adresujące ryzyko obejmujące jawne rozważanie alternatyw np. tworzymy/pozyskujemy komponent**
- **Mocne ukierunkowanie na zmiany – kolejna iteracja może obejmować ich wprowadzenie albo wręcz być temu poświęcona**
- **Szybciej wykrywane i rozwiązywane są trudne problemy i wyzwania napotykane w ramach projektu**

### Wady:

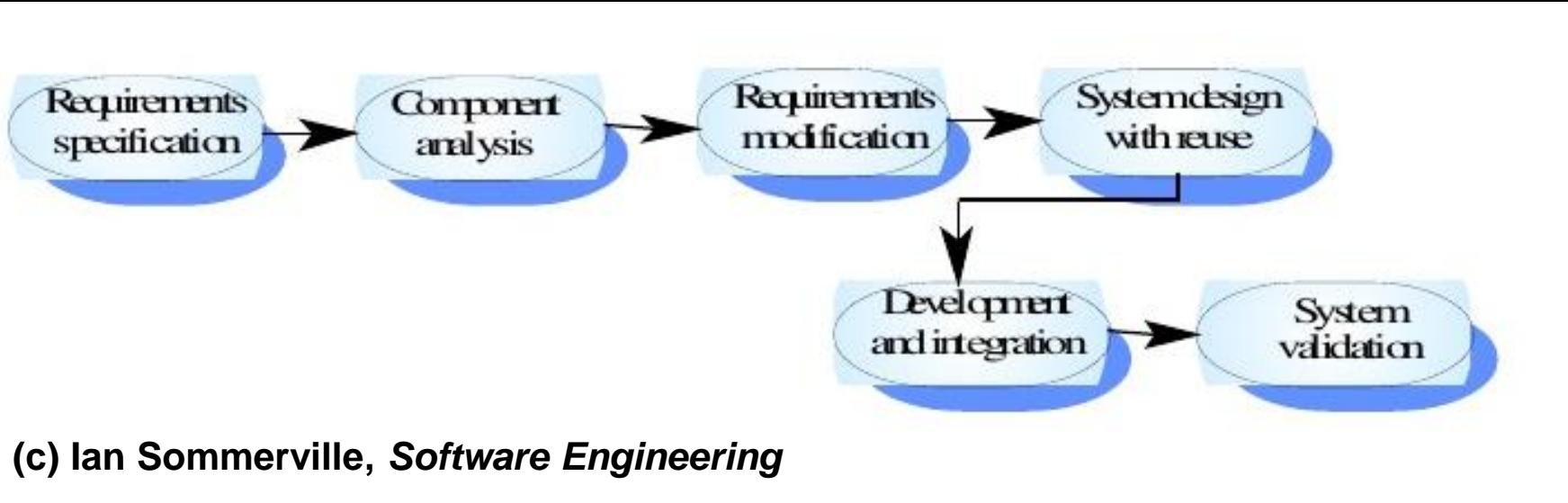
- **Model trudny do zrozumienia/wyjaśnienia**
- **Duży koszt zarządzania ryzykiem**
- **Długotrwałość dojścia do rozwiązania docelowego**

## Inne modele - uzupełniające

- **Niektóre systemy i projekty prowadzące do ich wytwarzania korzystają z pewnych specyficznych zasobów i praktyk np.**
  - wysoki stopień bazowania na gotowych komponentach (reuse)
  - budowa nowego systemu na podstawie już istniejącego (ale np. w przestarzałej technologii)
- **Nie są to podejścia zupełnie różne od przedstawianych wcześniej, ale z uwagi na specyfikę zasługują na odrębne omówienie**

# Komponentowy model wytwarzania

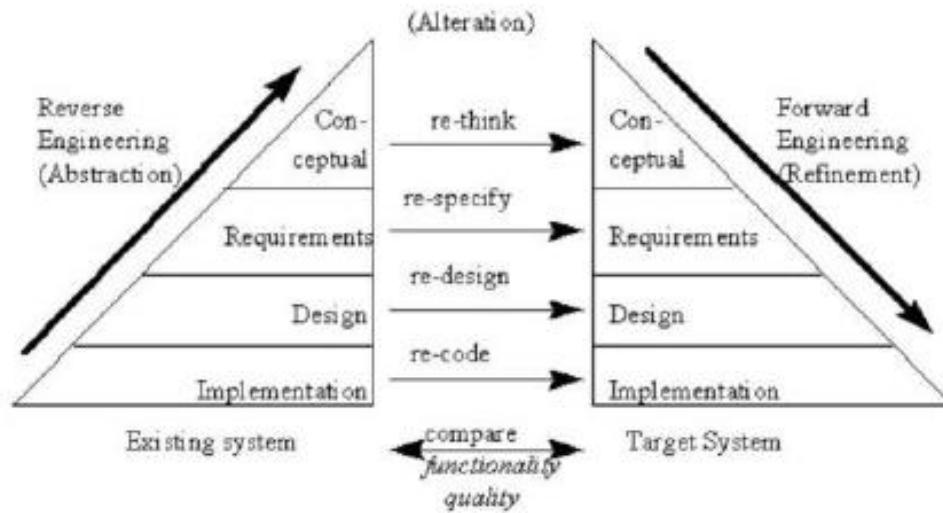
(CBD - Component Based Development)



- **Dla systemów opartych głównie o gotowe komponenty (pojęcie komponentu -> wykład o Software Reuse)**
- **Zamiast projektowania i implementacji „od nowa” zawiera dużo więcej analizy istniejących komponentów, ich wyboru, dostosowywania do potrzeb i integracji**

# Model ponownej inżynierii oprogramowania (*re-engineering*)

Inżynieria  
odwrotna

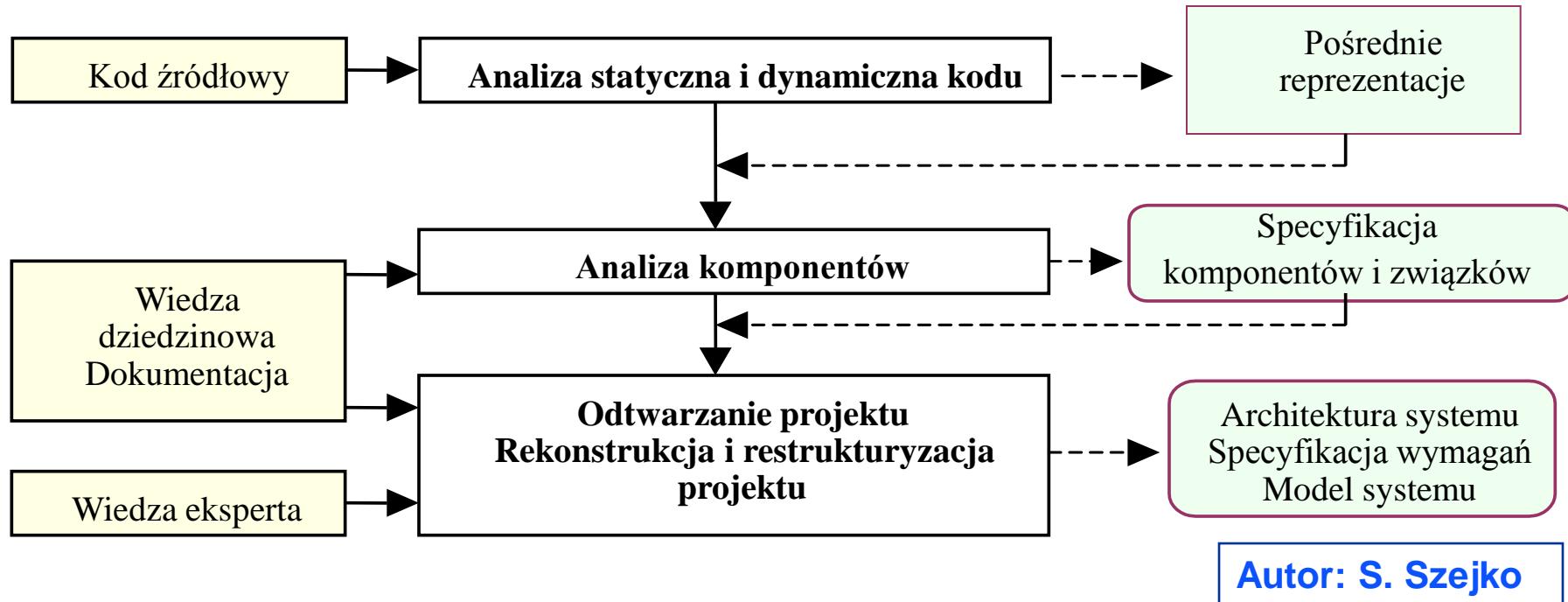


Inżynieria  
„do przodu”

Źródło: E.J. Byrne, „A  
*Conceptual Foundation for  
Software Re-engineering*”

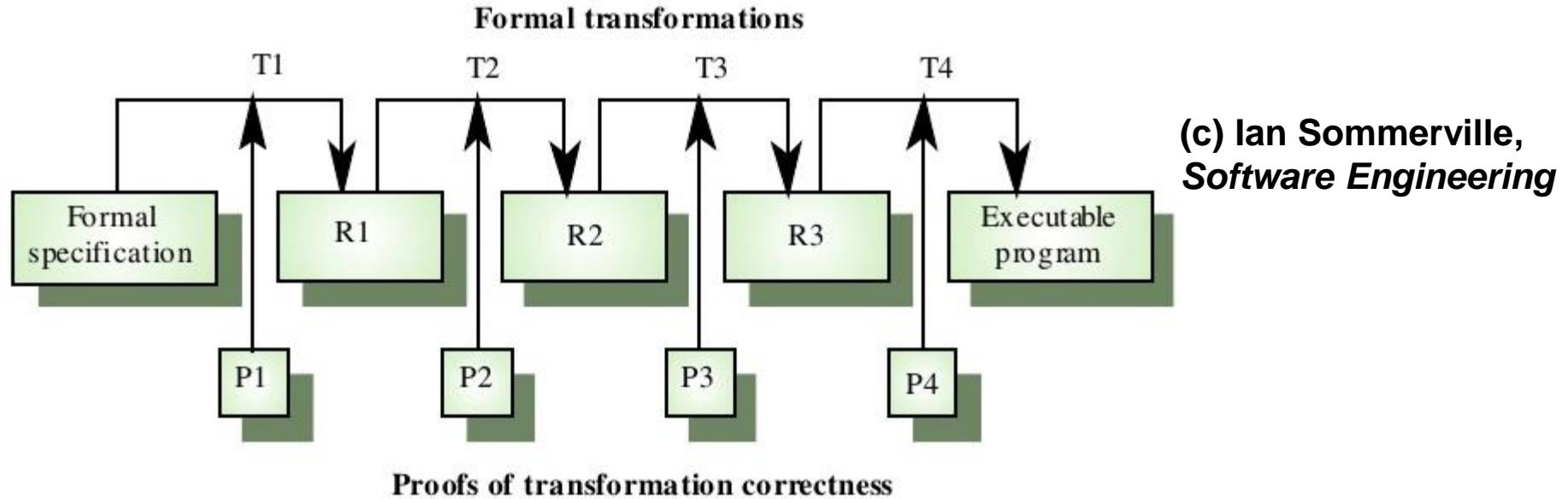
- **Model wytwarzania dla sytuacji, gdzie istnieje już jakiś system, który ma zostać zastąpiony nowym**
- **Dla starego systemu może nie być dokumentacji (która trzeba wówczas odtworzyć na podstawie kodu i działania), a nawet może nie być kodu źródłowego (konieczna dekompilacja)**
- **W inżynierii odwrotnej wchodzi się na wyższe poziomy abstrakcji (np. odtworzenie projektu architektury na podstawie kodu), w inżynierii „do przodu” na poziomy niższe (tak jak zwykle przy wytwarzaniu)**

# Schemat procesu inżynierii odwrotnej (*reverse engineering*)



- **Analiza oprogramowania w celu odtworzenia jego projektu i specyfikacji**
- **Może być częścią restrukturyzacji, a może być podstawą ponownej implementacji systemu**

# Model oparty o przekształcenia formalne (*transformational development*)

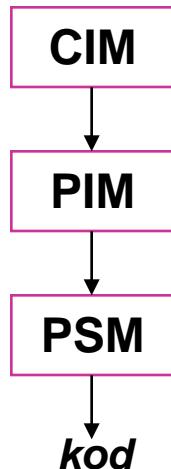


- Budowa najpierw innej reprezentacji systemu np. opartej o formalną notację matematyczną i potem przekształcanie tego do kolejnych reprezentacji i finalnie do postaci kodu
- Stosowane w wąskich dziedzinach zastosowań np. dla systemów krytycznych (ang. *safety-critical systems*)

# Przekształcenia modeli

## (*MDD Model-Driven Development*)

- Również przekształcenia modeli, choć niekoniecznie formalnych (matematycznych)
- Przykład: Architektura MDA (Model Driven Architecture) zaproponowana przez Object Management Group. MDA zakłada ona trzy poziomy modeli:



- modelowanie aplikacji w terminach biznesowych (ang. *Computation Independent Model*, CIM)
- model niezależny od platformy (ang. *Platform-Independent Model*, PIM), odwzorowujący abstrakcję biznesową np. w model funkcjonalności systemu
- model architektoniczny, zależny od platformy i technologii (ang. *Platform Specific Model*, PSM)

<http://www.omg.org/mda/>

# Modele wytwarzania - podsumowanie (1)

- Przydatność modelu wytwarzania jest uzależniona od specyfiki danego przedsięwzięcia
  - np. stabilność wymagań, świadomość interesariuszy, możliwość dekompozycji systemu na porcje funkcjonalności, możliwość użytkowania niedoskonałego systemu, poziom wymagań jakościowych...
- Warto dobierać model, organizację przedsięwzięcia i techniki wytwarzania w zależności od specyfiki danego problemu!
- Modele wytwarzania stanowią pewnego rodzaju abstrakcję, rzeczywiste procesy wytwarzania w konkretnych projektach często łączą elementy różnych modeli

# Modele wytwarzania - podsumowanie (2)

- **Możliwe i jak najbardziej warte rozważenia jest więc świadome łączenie elementów różnych modeli wytwarzania np.**
  - prototypowanie do pełniejszego określenia wymagań, dalej wytwarzanie wg modelu przyrostowego
  - model spiralny gdzie w poszczególnych obrotach spirali wykorzystuje się w różnym zakresie wykorzystanie i dostosowanie gotowych komponentów
- **Konkretnie metodyki wytwarzania często czerpią z kilku modeli cyklu życia i łączą ich zalety**
  - wybór danej metodyki zwykle oznacza (niekoniecznie świadomą) pracę w oparciu o pewien model lub kombinację modeli wytwarzania

# Praktyka?

- **Najczęściej iteracje z podejściem przyrostowo-ewolucyjnym**
  - Wydzielenie pewnych przyrostów - ale niekoniecznie są to osobne podsystemy, mogą być zestawy powiązanych funkcji czy use case'ów)
  - Skupienie w danym czasie (iteracji) na jednym przyroście
  - Nie ma jednak założenia, że to co powstało w czasie poprzednich przyrostów jest już kompletne i niezmienne – podlega to ciągłej ewolucji na podstawie np. sprzężenia zwrotnego od interesariuszy
  - Podsumowując: w każdej iteracji dodawany jest nowy przyrost i udoskonalane wcześniej wykonane elementy systemu
- **Uwaga – powyższe podejście nie musi obowiązywać dla wszystkich systemów i przedsięwzięć**
- **Istnieją chociażby przedsięwzięcia, w których problem jest na tyle dobrze zdefiniowany lub powtarzalny, że można zastosować model kaskadowy**

# Metodyki wytwarzania i zarządzania projektem

*Aleksander Jarzębowicz*

*Katedra Inżynierii Oprogramowania  
Politechnika Gdańsk*

Materiały pomocnicze do wykładu  
z Inżynierii Oprogramowania na Wydziale ETI PG.  
Ich lektura nie zastępuje obecności na wykładzie.  
Wykorzystanie materiałów w innym celu oraz ich rozpowszechnianie  
jest zabronione.

## Pojęcia (przypomnienie)

- **Model cyklu życia / model wytwarzania** – ogólna metoda organizacji procesu wytwarzania systemu, zawierająca podział na etapy i kryteria przechodzenia między etapami, może również sugerować produkty i praktyki
- **Metodyka** – konkretna metoda, bazująca na jednym lub kilku modelach wytwarzania, definiująca szczegółowo etapy, praktyki, produkty itp., zwykle firmowana przez określonego autora/gremium

# Metodyki i procesy

- Przedstawione do tej pory modele wytwarzania (np. kaskadowy, przyrostowy) miały ogólniejszy charakter
- Istnieją jednak bardziej szczegółowe wskazówki co do realizacji projektu lub/i zarządzania nim - **metodyki**
- Metodyki na ogół dokładniej precyzują zadania/czynności, praktyki, produkty i role ludzi
  - Choć poziom szczegółowości może być znacząco różny
- Powyższe elementy wchodzą w skład opisu całego procesu realizacji projektu informatycznego

# Działania w projekcie informatycznym

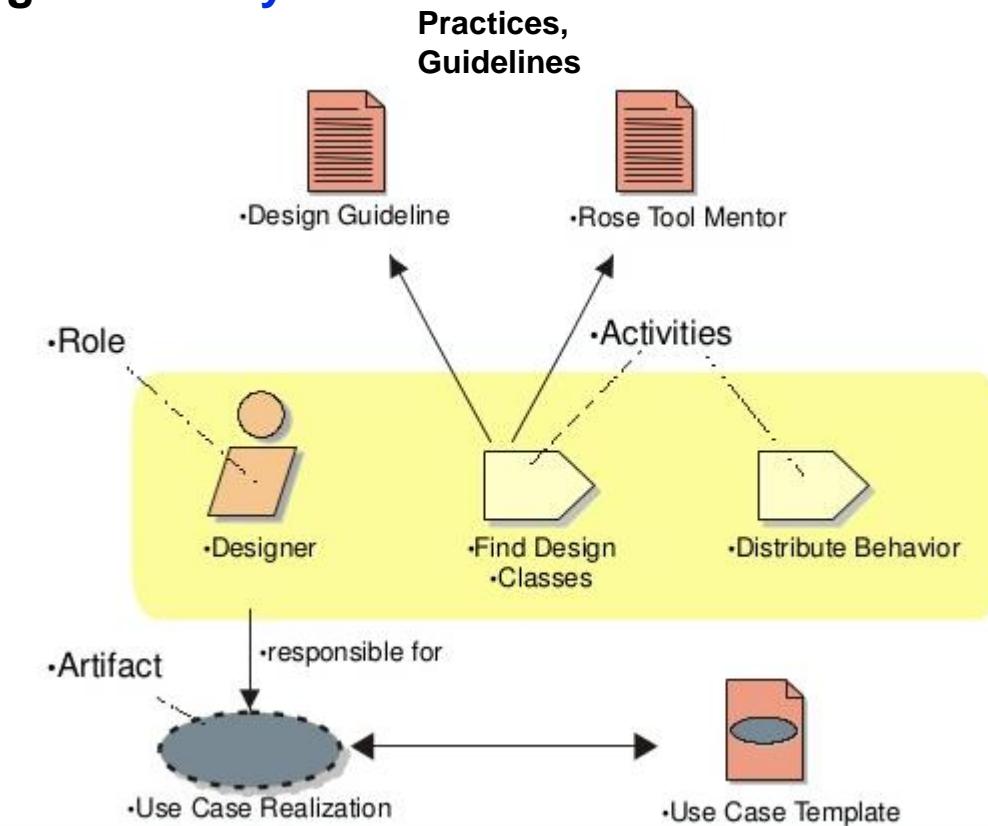
- **Zadania techniczne, m.in.**
  - inżynieria wymagań
  - projektowanie
  - implementacja
  - testowanie
  - wprowadzanie zmian / utrzymanie
  - wybór narzędzi i środowiska wytwórczego
- **Zadania zarządcze, np.**
  - planowanie
  - zarządzanie wymaganiami
  - zarządzanie budżetem/finansami
  - komunikacja w projekcie
  - zarządzanie ludźmi (doborem, motywowaniem, przydziałem zadań, ...)
  - zapewnianie jakości
  - ocena (postępów projektu i zgodności z planem, zaawansowania, testowania, zarządzania,...)
  - współpraca z otoczeniem projektu: klientami, własną firmą, związkami zawodowymi,...

Proces?

# Elementy procesu w metodyce

Proces określa **Co** jest wykonywane, przez **Kogo**, **Kiedy** i **Jak**, oraz jakie są tego **Produkty**

- **Co** – czynności (activities)
- **Kto** – role w projekcie (roles)
- **Kiedy** – wskazania kolejności, następstw przyczynowo skutkowych dla czynności
- **Jak** - towarzyszące im praktyki (practices, guidelines)
- **Produkty** - artefakty (artefacts)



Źródło: RUP

# Kategorie metodyk

- **Wytwórcze (*Software Development Methodology*)**
  - Zwykle bazują na określonym modelu wytwarzania lub kombinacji modeli
  - Koncentrują się na działaniach deweloperskich (programowanie, testowanie, analiza etc.)...
  - ... Siłą rzeczy zawierają jednak również zagadnienia zarządcze, menedżerskie (planowanie, kontrola jakości, zarządzanie ryzykiem etc.) – w mniejszym lub większym zakresie
- **Zarządzania (*Project Management Methodology*)**
  - Punkt widzenia menedżerski, zarządzanie przedsięwzięciem
  - W znacznej mierze w oderwaniu od konkretnych praktyk deweloperskich
  - Często metodyka uniwersalna tzn. dotycząca nie tylko do projektów informatycznych albo adaptowana do dziedziny oprogramowania

**Uwaga – czasami trudno rozgraniczyć!**

# Metodyki wytwarzania

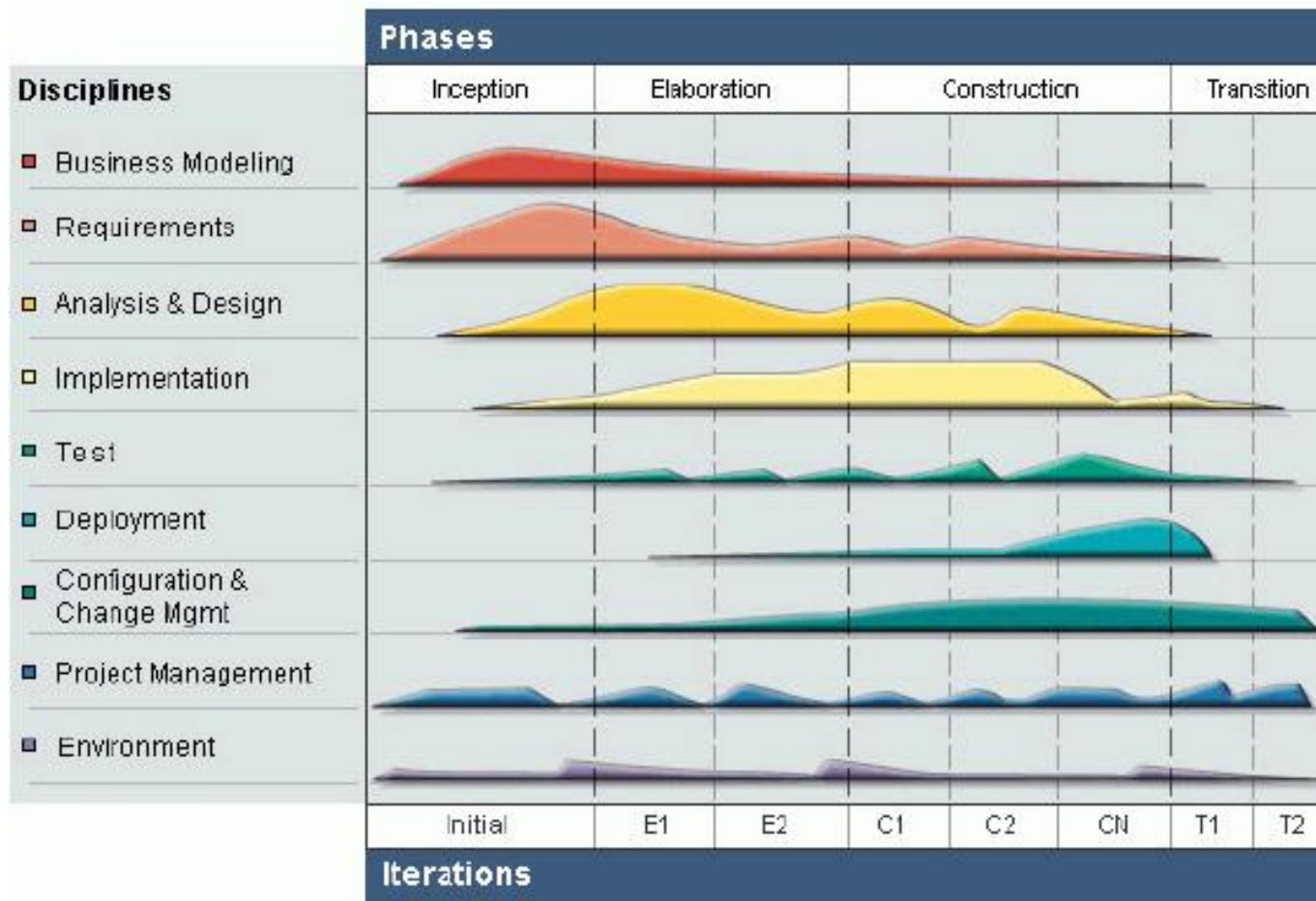
- Występuje powszechnie stosowany podział na:
  - Metodyki sterowane planem / zdyscyplinowane
  - Metodyki zwinne / lekkie
- Metodyki sterowane planem
  - Oparte na planowaniu i zarządzaniu
  - Przeznaczone dla dużych projektów realizowanych przez większe zespoły
  - Precyzyjnie zdefiniowane procesy, z możliwością adaptacji („przykrajania”)
  - Zapewnianie jakości poprzez dodatkowe zadania QA, kontrole, nadzór itp.
- Metodyki zwinne
  - Nastawione na zmiany i adaptację do zmian
  - Oryginalnie dla mniejszych projektów i zespołów (ale później powstały propozycje skalowania tzw. *scaling frameworks*)
  - Procesy „lekkie”, relatywnie mniejsza liczba działań i produktów
  - Zapewnianie jakości poprzez transparentność i wspólne zaangażowanie

# Rational Unified Process (RUP)

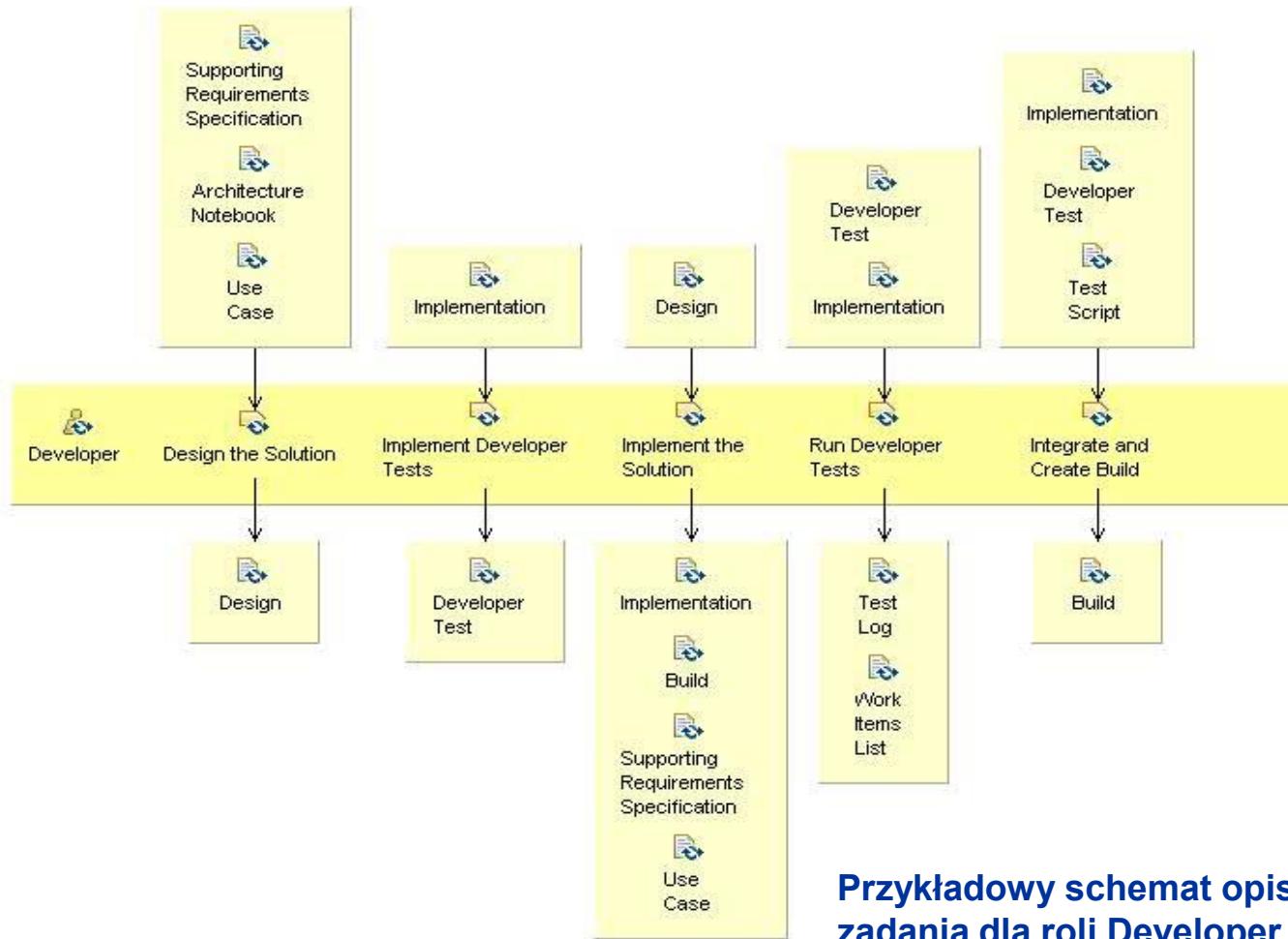
- Przykład metodyki zdyscyplinowanej
- Opracowany przez firmę Rational (obecnie część IBM) znaną z różnych osiągnięć w IO (UML, pakiety narzędzi)
- Rozbudowana metodyka (szerokie pokrycie zadań, praktyk, ról) udostępniona do „przykrajania” (ang. *tailoring*)
- Wytwarzanie iteracyjne, oparte na analizie ryzyka
- Wykorzystanie modeli wizualnych, narzędzi, architektur i komponentów



# Rational Unified Process (1)

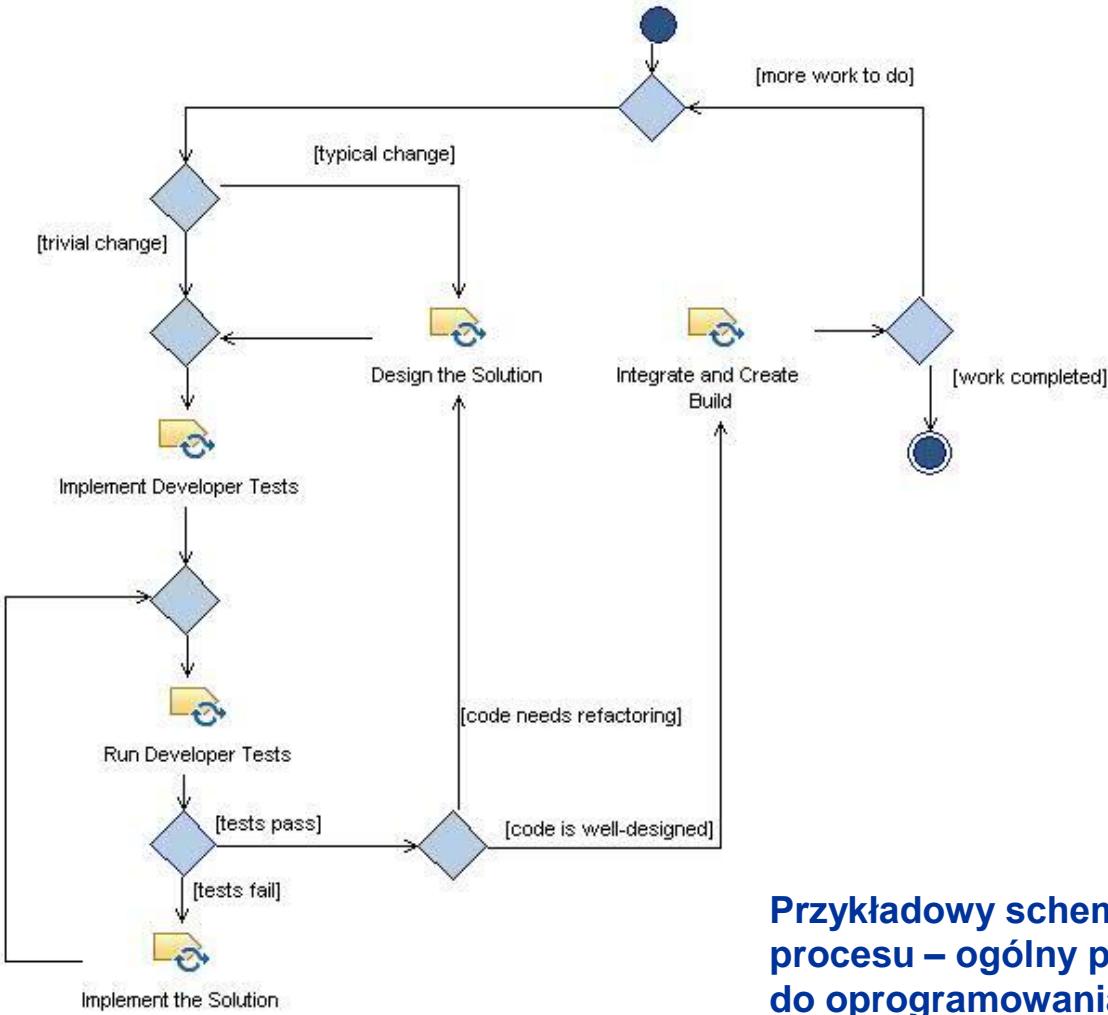


## Rational Unified Process (2)



**Przykładowy schemat opisu fragmentów procesu – zadania dla roli Developer i artefakty wejściowe i wyjściowe dla tych zadań**

## Rational Unified Process (3)



Przykładowy schemat opisu fragmentów procesu – ogólny proces wprowadzania zmian do oprogramowania

## Rational Unified Process (4)

- Z czasem powstały liczne „odmiany” RUP ukierunkowane na różne typy projektów / obszary zastosowań np.
  - Enterprise Unified Process
  - Open Unified Process
  - Agile Unified Process
  - RUP for Service Oriented Modeling and Architecture
- Dostępność narzędzi do definiowania nowych „odmian” oraz do „przykrajania” do własnych potrzeb
  - Rational Method Composer
  - Eclipse Process Framework



# Inne metodyki zdyscyplinowane

- **Microsoft Solution Framework (MSF)**
  - Microsoft, nastawienie na wykorzystanie narzędzi MS (Visual Studio Team Server)
  - Podejście iteracyjne z elementami cyklu spiralnego, choć wyróżnione pewne fazy (analogicznie jak w RUP)
- **Personal Software Process (PSP)**
  - Software Engineering Institute
  - Sformalizowany, mocno oparty na metrykach
  - Rejestrowanie czasu, zadań, defektów itp.
  - Samodoskonalenie
- **Team Software Process (TSP)**
  - Część „zespołowa” oparta na podstawie indywidualnej (PSP)
  - Organizacja zespołu i projektu
  - Planowanie i koordynacja prac

# Podejście zwinne (agile development)

- Podejście będące pewnego rodzaju reakcją na mocno sformalizowane, sterowane planem metodyki IO (które z kolei wynikały z potrzeby uporządkowania procesu i uzyskania jego powtarzalności)
- Agile Manifesto (2001):
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan

# Cechy podejścia agile (1)

- Wiele konkretnych metodyk definiujących zadania, produkty, role, dobre praktyki m.in.:
  - Scrum
  - eXtreme Programming
  - Kanban
  - Feature Driven Development
  - ...
- Wspólne mianowniki:
  - wytwarzanie iteracyjne, bardzo krótkie iteracje z małymi przyrostami
  - praca zespołowa z intensywną współpracą i bezpośrednią komunikacją
  - nastawienie na bliską współpracę z klientem i uwzględnianie na bieżąco jego postulatów
  - dostosowywanie produktu i procesu do potrzeb (agility, adaptability)
  - „agile mindset”



## Cechy podejścia agile (2)

- Nie oznacza działania bez planu i celu czy bałaganu (typu *cowboy coding*)
  - ...choć bywa, że bałaganiarze chętnie „podczepiają się” pod szyld Agile
  - konieczne jest stosowanie wielu praktyk deweloperskich i intensywnej komunikacji, żeby osiągnąć sensowny rezultat przy braku dokumentacji i ciągłych zmianach
- „Coś za coś” np.
  - Nie dokumentujemy wymagań, polegamy na komunikacji z klientem i demonstrowaniu mu kolejnych wersji oprogramowania -> trzeba będzie wprowadzać zmiany w kodzie wg informacji zwrotnych od klienta
  - Nie robimy formalnego projektu systemu, zasada YAGNI -> trudniejsze będzie utrzymanie, możliwa konieczność restrukturyzacji (zmiany architektury)
  - Nie ma kierownika, który rozdziela zadania -> członkowie zespołu muszą mieć większą samomotywację i poczucie odpowiedzialności
- Obserwowalne zjawisko - łączenie elementów agile i elementów z innych metodyk (np. development w oparciu o agile, ale dodatkowe mechanizmy zarządzania „wyżej” plus rozbudowana inżynieria wymagań)
  - tzw. hybrid development approaches
  - wyniki inicjatywy badawczej HELENA Survey - Hybrid dEveLopmENt Approaches in software systems development

# Scrum (1)

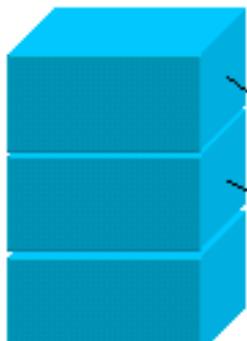
- Najpopularniejsza obecnie metodyka zwinna (czy też framework – niektórzy „czepiają się” słów)
- Definiuje proces wytwarzania w krótkich iteracjach (tzw. Sprintach) oraz role zespołu i sposoby komunikacji
- Względnie mało mówi o praktykach wytwarzania i zapewniania jakości, technice pracy, produktach
- Wymagania reprezentowane w formie bardzo ogólnej i zgrupowane w rejestrach
  - Product Backlog
  - Sprint Backlog

## Scrum (2)

### SCRUM Sprint Cycle

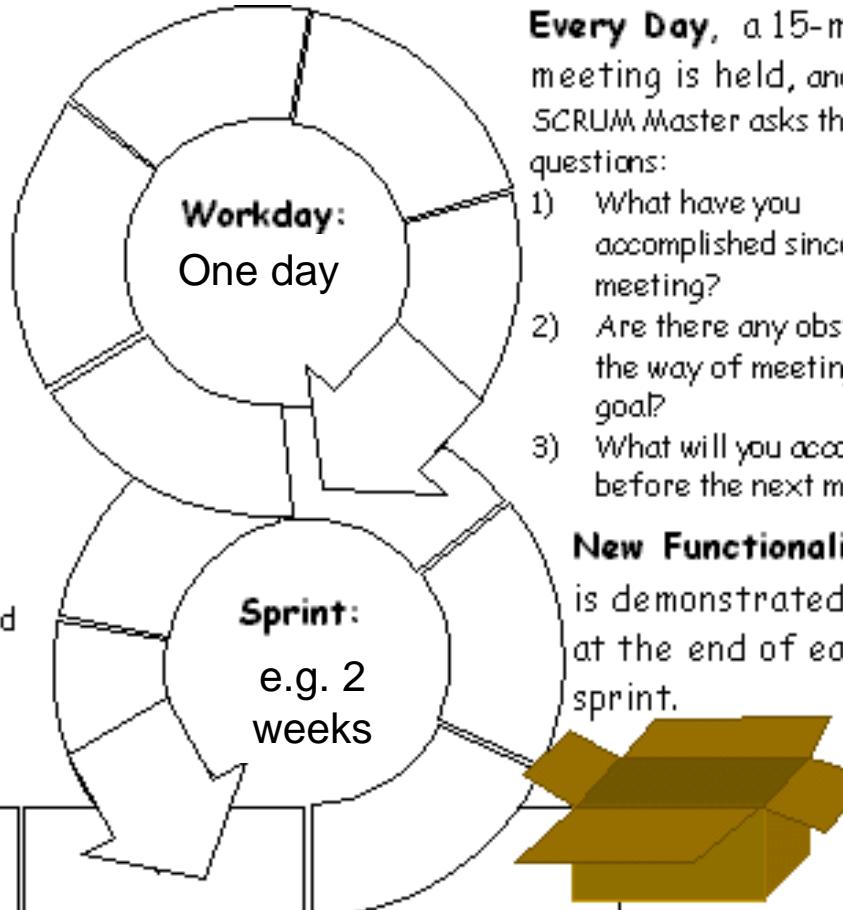
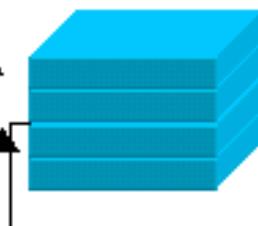
#### Product Backlog:

Prioritized list of features required by the customer



#### Sprint Backlog:

Features to be done this sprint  
Features are expanded into smaller tasks.



**Every Day**, a 15-minute meeting is held, and the SCRUM Master asks the 3 questions:

- 1) What have you accomplished since the last meeting?
- 2) Are there any obstacles in the way of meeting your goal?
- 3) What will you accomplish before the next meeting?

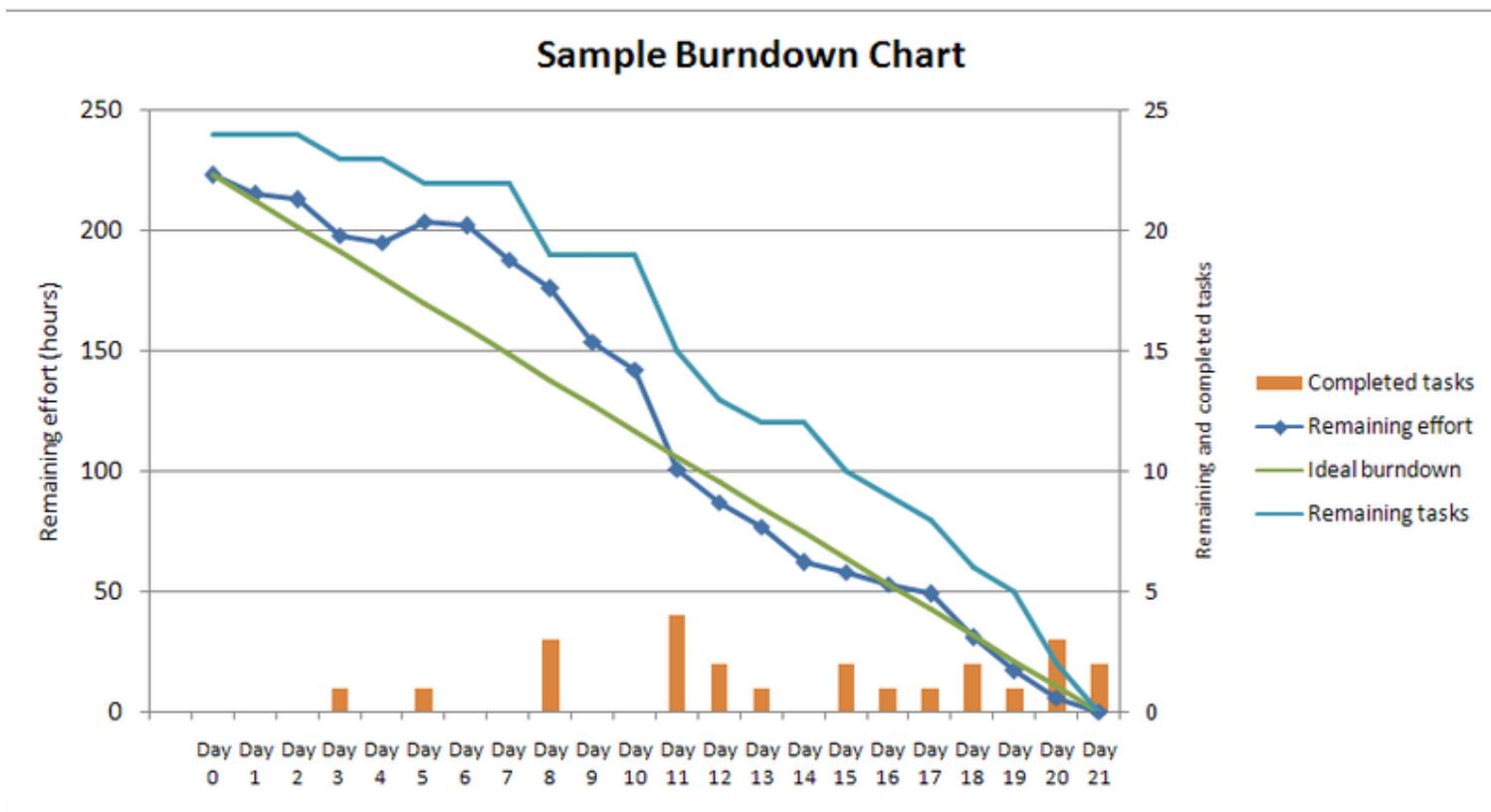
#### New Functionality

is demonstrated at the end of each sprint.



Źródło: [www.codeproject.com](http://www.codeproject.com)

## Scrum (3)

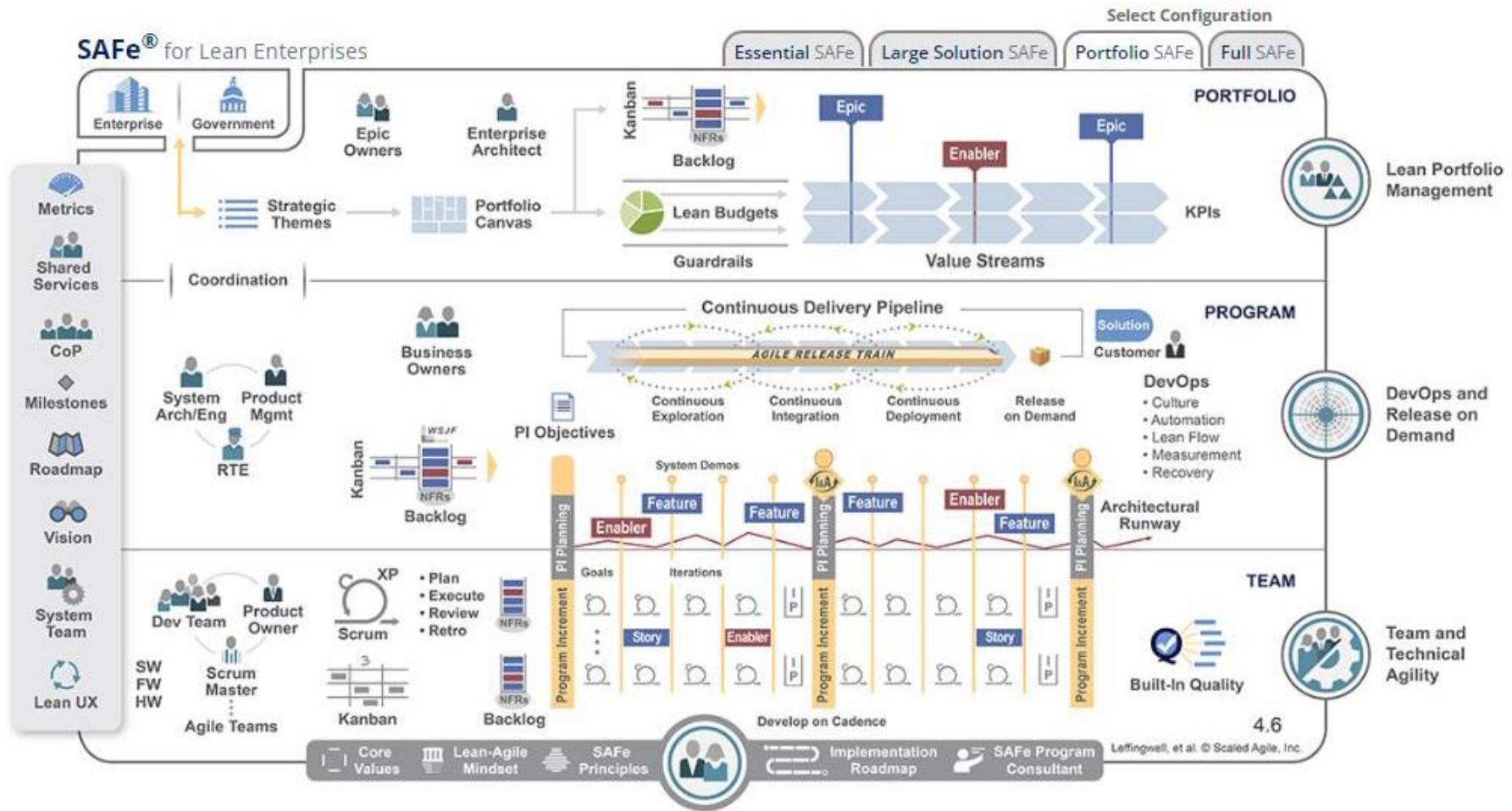


# Extreme Programming (XP)

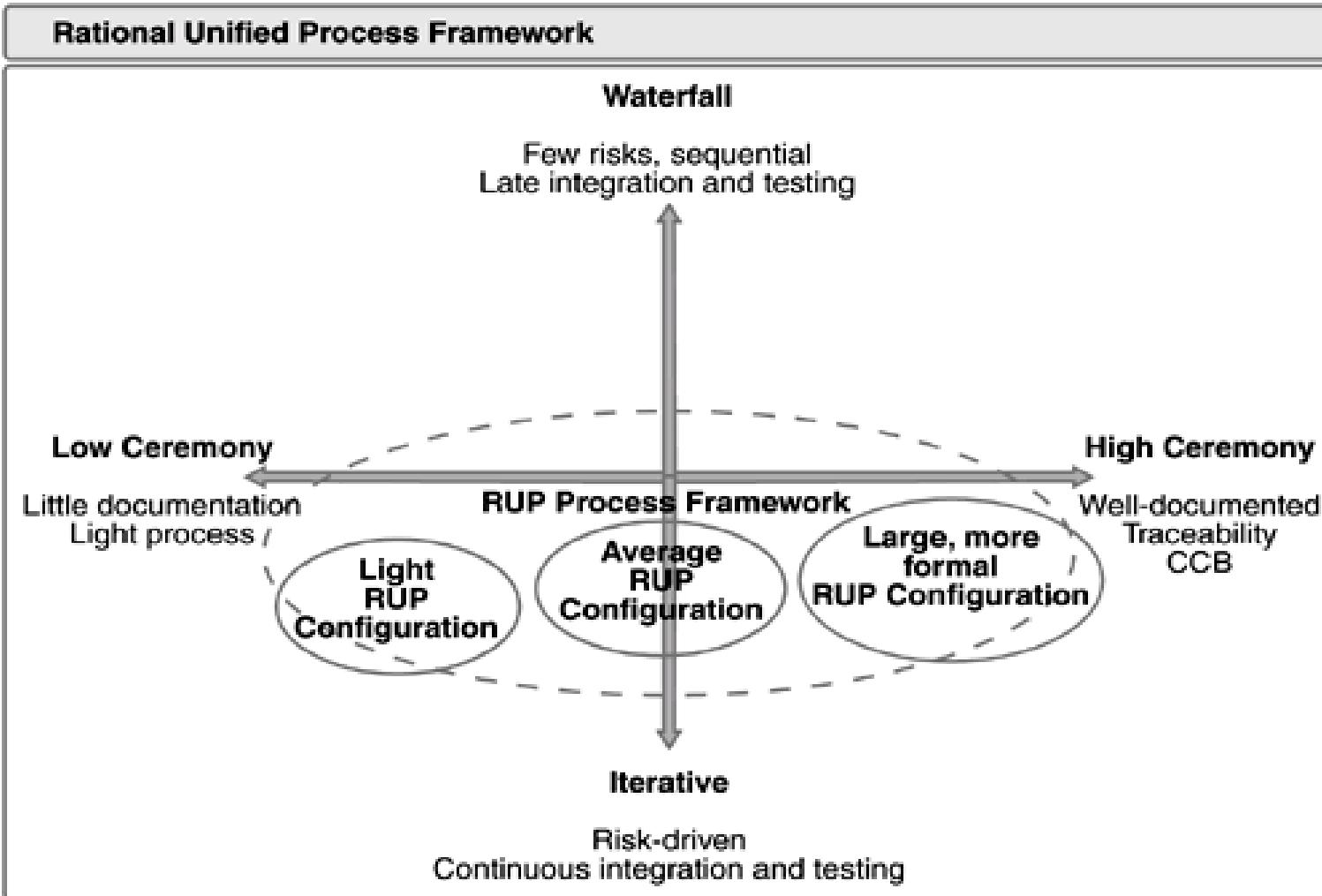
- Pierwsza metodyka zwinna szerzej spopularyzowana
- W mniejszym stopniu definiuje proces wytwarzania – postuluje krótkie iteracje
- Zawiera za to szereg praktyk dotyczących różnych obszarów inżynierskich np.
  - Ciągła integracja
  - Test-driven development
  - Gra planistyczna – „negocjacje” z klientem
  - Nacisk na prostotę projektu
  - Nacisk na refaktoryzację



# Skalowanie metod zwinnych (na przykładzie SAFe)



# „Wymiary” metodyk



# Zarządzanie projektem

- **Co to znaczy, że projekt osiągnął sukces?**
- **Wiele wymiarów:**
  - Cele (<- zakres, <- wymagania)
  - Budżet / koszty
  - Czas / harmonogram
  - Jakość (czasem ujmowana w ramach pierwszego wymiaru, czasem wyróżniana jako osobny wymiar)
- **Nie tylko aspekty bezpośrednio związane z wytwarzaniem!**
- **Im większy projekt, tym więcej wysiłku wymaga planowanie, koordynacja prac, komunikacja, zarządzanie konfiguracją...**

# Przykładowe obszary zarządzania

- Zarządzanie zakresem / interesariuszami
- Zarządzanie czasem
- Zarządzanie kosztami
- Zarządzanie ludźmi / zespołem
- Zarządzanie komunikacją
- Zarządzanie konfiguracją
- Zarządzanie zmianą
- Zarządzanie jakością
- Zarządzanie ryzykiem
- .....



Źródło: PMBOK + opr. wł.

# Zarządzanie zakresem i interesariuszami

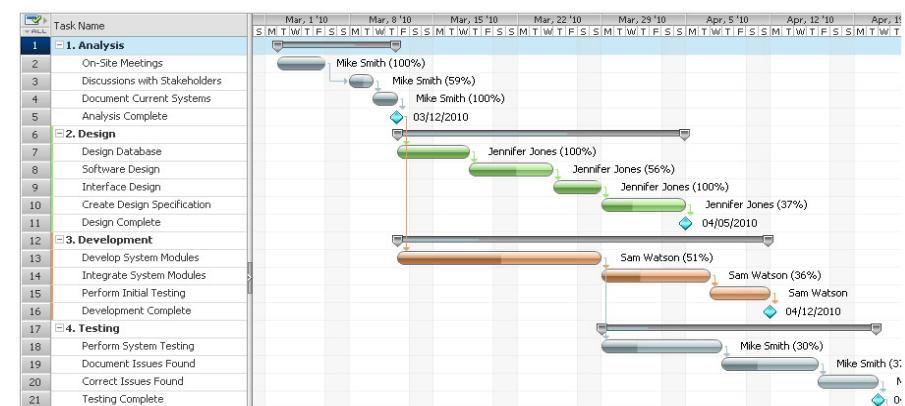
- **Identyfikacja interesariuszy**
- **Utrzymywanie zaangażowania interesariuszy i komunikacja z nimi**
- **Ustalenie i weryfikacja zakresu (systemu, usług)**
- **Przełożenie zakresu na bardziej szczegółowe zadania i produkty**
- **Reagowanie na zmiany zakresu**
- **Zapobieganie „rozpełzaniu zakresu” (ang. *scope creep*)**



**Wykład o analizie biznesowej i systemowej**

# Zarządzanie czasem

- **Identyfikacja zadań i ich wzajemnych powiązań**
- **Oszacowanie czasu dla poszczególnych zadań**
- **Przygotowanie harmonogramu**
  - Milestones (punkty kontrolne)
  - Diagram obrazujący zależności np. diagram Gantta czy PERT
- **Nadzór nad realizacją harmonogramu**
- **Reakcja na zmiany / problemy**



# Zarządzanie kosztami

- **Identyfikacja zasobów (ludzie, narzędzia, materiały)**
- **Oszacowanie kosztów zasobów (również pośrednich!)**
- **Odwzorowanie kosztów na zadania, etapy projektu, harmonogram**
- **Nadzorowanie budżetu, kontrola wydatków**



# Zarządzanie ludźmi / zespołem

- **Organizacja zespołu**
  - Struktura zespołu
  - Role i odpowiedzialności
  - Sposoby komunikacji
- **Dobór ludzi do zespołu**
  - Kompetencje
  - Profil
- **Budowa zespołu**
  - Zasady współpracy
  - Wartości i praktyki
  - Atmosfera
- **Motywowanie**
- **Ocena**



# Zarządzanie komunikacją

- **Przygotowanie kanałów komunikacyjnych**
  - Narzędzia i technologie
  - Procedury
  - Uprawnienia dostępu do informacji
- **Zapewnienie dostępu do informacji, dystrybucja**
  - np. odnośnie przyjętych decyzji projektowych
- **Raportowanie**
  - np. postępów prac, rezultatów danego etapu
- **Zgłaszanie problemów**



# Zarządzanie konfiguracją i zmianą

- Zarządzanie konfiguracją: nadzór nad wszystkimi produktami i zasobami np.
  - Kod źródłowy
  - Przypadki testowe
  - Wymagania
  - Umowy
  - Narzędzia
- Zarządzanie zmianą: zgłoszanie, rozpatrywanie i śledzenie zmian występujących w trakcie trwania projektu np.
  - Nowe wymaganie
  - Zgłoszenie defektu



**Wykład 12c**

# Zarządzanie jakością

- **Planowanie jakości**
  - określenie standardów
  - określenie celów i miar ich osiągania
- **Zapewnienie jakości**
  - plan zapewniania jakości i jego realizacja
  - audyty
- **Kontrola jakości**
  - przeglądy i inspekcje
  - próbkowanie
  - analiza przyczyn
  - zmiany w procesach



# Zarządzanie ryzykiem

- **Ryzyko – szansa wystąpienia niepożądanego zdarzenia o negatywnych konsekwencjach**
- **Zarządzanie ryzykiem – działanie proaktywne zapobiegające lub ograniczające konsekwencje**
- **Proces zarządzania ryzykiem:**
  - identyfikacja ryzyk
  - oszacowanie ryzyka  
(prawdopodobieństwo / konsekwencje)
  - przeciwdziałanie
  - nadzór nad przeciwdziałaniem



# PMBOK

- Project Management Body of Knowledge (obecnie 7 ed.)
- Standard / metodyka zarządzania firmowana przez Project Management Institute
- Certyfikacja (np. Project Management Professional)
- Przedstawia procesy zarządcze z 13 obszarów wiedzy (w większości zgodnych z przedstawianymi na poprzednich slajdach)
- Każdy proces opisuje w kategoriach:
  - artefaktów wejściowych
  - stosowanych w ramach procesu metod i narzędzi
  - produktów procesu



# PRINCE 2

- PRojects IN Controlled Environments 2, obecnie 7th edition
- Metodyka opracowana jako „standard” dla projektów informatycznych realizowanych na zlecenia rządowe w W. Brytanii
- Definiuje strukturę zarządzania projektem (komitet sterujący z reprezentantami klienta i dostawcy, kierownik projektu, kierownicy zespołów) i relacje między nimi
- Duże znaczenie przykłada do inicjacji projektu obejmującej wizję biznesową, planowanie projektu, planowanie jakości, mechanizmy kontroli
- Projekt dzielony na etapy (formalnie rozliczane)
- Znaczna ilość działań kontrolnych i projakościowych oraz związanej z tym dokumentacji
- Adaptacja PRINCE2 Agile (*tailoring*)



# ITIL

- **Information Technology Infrastructure Library (obecnie wersja 4)**
- **Dotyczy świadczenia usług IT dla biznesu (a więc niekoniecznie tworzenia nowych systemów, odbiega od *Project Management*)**
- **Obejmuje 5 głównych obszarów:**
  1. ITIL Service Strategy
  2. ITIL Service Design
  3. ITIL Service Transition
  4. ITIL Service Operation
  5. ITIL Continual Service Improvement
- **Dla poszczególnych obszarów definiuje procesy, które organizacja powinna opracować i realizować dla sensownego świadczenia usług np.**
  - **Service Design**: Availability Management, Service Level Management, ISMS, ...
  - **Service Transition**: Transition planning and support, Service asset and configuration management, ...
  - **Service Operation**: Incident Management, Request Fulfillment, ...



## Metodyki - podsumowanie

- Metodyki stanowią zwykle już całkiem precyzyjne opisy (kto, co, kiedy, jak ...)
- Oczywiście można ścisłe przestrzegać tych zaleceń, ale nie są one nienaruszalną świętością! Możliwe są adaptacje.
  - Choć oczywiście można zmienić tyle, że stosowanie metodyki traci sens
- W ramach określonej metodyki nadal mamy pole manewru co do definicji zadań, produktów, harmonogramu, przypisania ról itp. i możemy uwzględnić specyfikę naszego projektu
  - np. zamiast use-case'ów robimy szkice poszczególnych ekranów, łączymy testowanie integracyjne z systemowym, nie ma dedykowanej roli związanej z zapewnianiem jakości, lecz jest to częścią obowiązków kierownika projektu
- Często metodyki są od razu opracowane w takiej formie, żeby móc je „przykroić” do własnych potrzeb (np. RUP)