INŻYNIERIA OPROGRAMOWANIA

wykład 1: zakres wiedzy

dr inż. Leszek Grocholski
pok. 236

Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

INŻYNIERIA OPROGRAMOWANIA

SWEBOK 2014

Guide to the
Software Body of Knowledge
(http://www.swebok.org ***)***

A project of the IEEE Computer Society
Proffesional Practices Commitee

IEEE Computer Society (http://computer.org **)**
(Institute of Electrical and Electronics Engineers)

Przedmiot inżynierii oprogramowania

INŻYNIERIA OPROGRAMOWANIA jest praktyczną wiedzą techniczną dotyczącą wszystkich procesów dotyczących: wytwarzania, wdrażania i utrzymania oprogramowania.

Traktuje oprogramowanie jako produkt, który ma spełniać potrzeby techniczne, ekonomiczne lub społeczne.

Wiedza praktyczna

- tworzona przez praktyków a nie naukowców,
- przeznaczona dla praktyków,
- wiedza interdyscyplinarna – socjologia, psychologia itd. ….,
- jej wynikiem są najlepsze praktyki, standardy, normy, metodyki, które przeznaczone są do bardzo konkretnych zastosowań.

Przedmiot inżynierii oprogramowania ..

Produkcja oprogramowania jest procesem składającym się z wielu etapów. **Kodowanie (pisanie programów) jest tylko jednym z nich, niekoniecznie najważniejszą.**

W USA spada zapotrzebowanie na programistów (z 17% do 11%) a rośnie rynek pracy dla inżynierów oprogramowania (z 28 % do 33%) dane pochodzą z lat 2014 – 2015.

Inżynieria oprogramowania jest wiedzą empiryczną, syntezą doświadczenia tysięcy ośrodków zajmujących się budową oprogramowania.

Praktyka pokazała, że w inżynierii oprogramowania nie ma miejsca stereotyp „od teorii do praktyki”. Teorie, szczególnie „zmatematyzowane” teorie, okazały się dramatycznie nieskuteczne w praktyce.

Inżynieria oprogramowania - główny powód stosowania

Głównym powodem stosowania wiedzy dostarczanej przez inżynierie oprogramowania jest problem rosnącej złożoności oprogramowania !!!

Trochę historii:
-40 lat temu nie było komputerów osobistych - PC,
-25 lat temu nie było internetu,
-15 lat temu nie było smartfonów, Google, Facebook … .

Co już się rozpoczęło i co nas czeka:
-internet rzeczy małych (np. palety) i dużych (samochody),
-coraz bardziej inteligentne miasta,
-klienci potrzebują rozwiązań a nie oprogramowania, komputerów itd. …,
-ogromna ilość transmitowanych i gromadzonych informacji - co w 2030r ?.

Zasługą inżynierii oprogramowania jest to, że od lat procent przedsięwzięć informatycznych które kończą się sukcesem jest od stały i wynosi ok 30 %

INŻYNIERIA OPROGRAMOWANIA

HISTORIA INŻYNIERII OPROGRAMOWANIA

- **software** 1958 – John Tukey, słynny statystyk użył pierwszy raz słów software i hardware.

- **software engineering** - tytuł konferencji NATO, która odbyła się w Niemczech w roku 1968.

- W roku 1972 IEEE Computer Society (Instytut inżynierów Elektryków i Elektroników) zaczęło wydawać czasopismo **Transactions on Software Engineering**.

- Komitet IEEE Computer Society **odpowiedzialny za stanowienie standardów inżynierii** oprogramowania – 1976 r.

- Pierwsza norma – IEEE Std 730 – dotycząca **zapewnienia jakości oprogramowania** – 1979 r.

INŻYNIERIA OPROGRAMOWANIA

Normy IEEE dot. SE do roku 1999:

- Customer and Terminology Standards - 9
- Process standards – 14
- Product standards – 5
- Resource and technique standards -12

Po roku 1999 – kolejnych kilkadziesiąt norm

INŻYNIERIA OPROGRAMOWANIA

ACM – Association for Computing Machinery

1995 r. - IEEE i ACM zauważyły konieczność opracowania odpowiedniego kompedium wiedzy dla działalności związanej z inżynierią oprogramowania - SWEBOK

Kompedium wiedzy i regulacje prawne, które stanowiły by podstawę: decyzji przemysłowych, certyfikatów zawodowych i programów edukacyjnych.

INŻYNIERIA OPROGRAMOWANIA

Wspólny komitet IEEE i ACM postanowił dla inżynierii oprogramowania:

- Zdefiniować podstawowy zakres wiedzy i rekomendowanych zasad działania (recommended practice).
- Zdefiniować zawodowe standardy etycznego, profesjonalnego postępowania (code of ethical and profecionals practice).
- Zdefiniować programy nauczania
  (eductional curricula) dla szkół zawodowych i wyższych.

INŻYNIERIA OPROGRAMOWANIA

- Pierwsze wydanie SWEBOK - 1998 roku. Aktualnie wersja 3 z 2014 r.

- Kodeks dot. etycznych zasady postępowania zawodowego został opracowany i zaaprobowany przez IEEE i ACM w roku 1998.

- Wytyczne dla celów nauczania oraz wymagane zakresy wiedzy (curriculum guidelines) zostały opracowane w roku 2004 - Software Engineering 2004 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering (http://www. computer.org/ portal/

INŻYNIERIA OPROGRAMOWANIA

- Steve McConnell, Roger Pressman, Ian Sommerville - światowe autorytety, autorzy podręczników inżynierii oprogramowania (wydanych również po polsku)

  stanowili panel ekspertów biorących udział w opracowywaniu przewodnika SWEBOK.

- Z Polski w pracach komisji IEEE/ACM uczestniczył prof. Janusz Górski z Wydziału Elektroniki, Telekomunikacji i Informatyki Katedra Inżynierii Oprogramowania, Politechniki Gdańskiej.

INŻYNIERIA OPROGRAMOWANIA

10 obszarów wiedzy (Knowlege Areas - KA) SE. Każdy z obszarów wiedzy został opisany w poszczególnych rozdziałach przewodnika:

1. **Zarządzanie wymaganiami dot. oprogramowania (inżynieria wymagań)**
2. **Projektowanie oprogramowania**
3. **Wytwarzanie oprogramowania (w tym programowanie)**
4. **Testowanie oprogramowania**
5. **Wdrożenie i eksploatacja (utrzymywanie) oprogramowania**

---

6. Zarządzanie zmianami i konfiguracjami
7. **Zarządzanie wytwarzaniem oprogramowania**
8. Procesy w życiu oprogramowania
9. Metody i narzędzia inżynierii oprogramowania
10. **Jakość oprogramowania**

INŻYNIERIA OPROGRAMOWANIA

Co stanowi źródło wiedzy inżynierii oprogramowania wg SWEBOK ? :

- normy i standardy : IEEE, ISO/IEC, SEI,… zawierają definicje
- podręczniki,
- publikacje,
- strony www.

SWEBOK rekomenduje podstawową literaturę.

INŻYNIERIA OPROGRAMOWANIA

Przykład:

Zarządzanie wymaganiami dot. oprogramowania

Wymaganie jest zdefiniowane jako CECHA, która musi być ustalona w celu rozwiązania pewnego problemu dotyczącego rzeczywistego świata.

W skład obszaru wiedzy dot. zarządzania wymaganiami oprogramowania wchodzą następujące podobszary:

- Podstawy definiowania wymagań na oprogramowanie;
- Proces definiowania wymagań;
- Uzyskiwanie wymagań od zainteresowanych stron;
- Analiza wymagań (np. ważność);
- Specyfikacja wymagań;
- Ocena wymagań;
- Rozważania praktyczne dotyczące inżynierii wymagań.

INŻYNIERIA OPROGRAMOWANIA

Przykład: Zarządzanie wytwarzaniem oprogramowania

Zarządzanie wytwarzaniem oprogramowania odwołuje się do wiedzy dot. zarządzania przedsięwzięciem i pomiarów procesów inżynierii oprogramowania. W skład obszaru wiedzy wytwarzanie oprogramowania wchodzą następujące podobszary:

- Rozpoczęcie i definiowanie zakresu przedsięwzięcia;
- Planowanie przedsięwzięcia;
- Dokumentowanie postępów prac;
- Przeglądy i ocena;
- Zakończenie przedsięwzięcia;
- Miary związane z wytwarzaniem oprogramowania.

INŻYNIERIA OPROGRAMOWANIA

SWEBOK – określa poziom wiedzy inżyniera oprogramowania zdobytej podczas 4 letniej edukacji

Do określania niezbędnego poziomu wiedzy wykorzystano tzw. taksonomie celów edukacyjnych Blooma [5].

Oceniono, że dla skutecznego korzystania z metod inżynierii oprogramowania opisanych w SWEBOK niezbędny jest pewien minimalny poziom znajomości zagadnień.

W załączniku SWEBOK podano, jakie są wymagane minima dla poszczególnych obszarów. Wytyczne te mogą pomóc w przygotowaniu materiałów szkoleniowych, opisie obowiązków, planowaniu rozwoju pracownika, szkoleniach zawodowych, jak również (przede wszystkim?) w tworzeniu programów edukacyjnych na uczelniach.

INŻYNIERIA OPROGRAMOWANIA

Cele nauczania wg Blooma to:

znajomość zagadnień, zrozumienie, stosowanie wiedzy, analiza, synteza oraz ocena.

Cele nauczania 221 podobszarów wiedzy SE wg SWEBOK

- Stosowanie 109 49,32%
- Zrozumienie 80 36,20%
- Analiza 31 14,03%
- Znajomość 1 0,45%
- Synteza 0 0,0 %
- Ocena 0 0,0 %

- Razem 221 100 %

INŻYNIERIA OPROGRAMOWANIA

Przykład 1: Podobszar wiedzy: projektowanie obiektowe w dziedzinie (wiedzy) projektowanie architektury.

Autorzy SWEBOK sugerując cel nauczania na poziomie analizy. Stwierdzają, że niższy poziom nauczania (znajomość, zrozumienie lub stosowanie) są niewystarczające, natomiast wyższe (synteza lub ocena) są niepotrzebne.

INŻYNIERIA OPROGRAMOWANIA

Przykład 2: Konstruowanie testów w dziedzinie (wiedzy) wytwarzanie oprogramowania

Wymagania wiedzy na poziomie stosowania. Wiedza na niższym poziomie (znajomość, zrozumienie) jest niewystarczająca, a na wyższym (analiza, synteza lub ocena) zbędna lub koszt jej zdobycia jest nieadekwatny do pożytku.

INŻYNIERIA OPROGRAMOWANIA

Może zwracać uwagę fakt, że implementacja zaleceń SWEBOK z reguły nie wymaga wiedzy na najwyższych poziomach

- analizy i syntezy !

INŻYNIERIA OPROGRAMOWANIA

Ostatni rozdział SWEBOK określa dziedziny związane z inżynierią oprogramowania, wymieniając wśród nich:

- inżynierię komputerową
- inżynierię systemów
- informatykę teoretyczna (computer science)
- zarządzanie zespołami ludzkimi
- matematykę
- zarządzanie przedsięwzięciem (projektem)
- zarządzanie jakością
- ergonomię oprogramowania

INŻYNIERIA OPROGRAMOWANIA

- Przykładowo w zakresie informatyki teoretycznej SWEBOK wymienia obszary:

struktury dyskretne, podstawy programowania, algorytmy i złożoność, organizacja i architektura, systemy operacyjne, obliczenia sieciowe, języki programowania, interfejs człowiek – maszyna, wizualizacja i grafika, systemy inteligentne, metody i modele numeryczne.

INŻYNIERIA OPROGRAMOWANIA

- Jak nauczać inżynierię oprogramowania ?

  Oczywiście na podstawie SWEBOK

Główna korzyść:

- Zgodność ze standardami (czytaj np.. programami studiów i normami przemysłowymi) światowymi.

INŻYNIERIA OPROGRAMOWANIA - INACZEJ

- Sposoby prowadzenia przedsięwzięć informatycznych.
- Techniki planowania, szacowania kosztów, harmonogramowania i monitorowania przedsięwzięć informatycznych.
- Metody analizy i projektowania systemów.
- Techniki zwiększania niezawodności oprogramowania.
- Sposoby testowania systemów i szacowania niezawodności.
- Sposoby przygotowania dokumentacji technicznej i użytkowej.
- Procedury kontroli jakości.
- Metody redukcji kosztów konserwacji (usuwania błędów, modyfikacji i rozszerzeń)
- Techniki pracy zespołowej i czynniki psychologiczne wpływające na efektywność pracy.

PROBREMY IO (1)

Sprzeczność pomiędzy odpowiedzialnością, jaka spoczywa na współczesnych SI, a ich zawodnością wynikającą ze złożoności i ciągle niedojrzałych metod tworzenia i weryfikacji oprogramowania.

Ogromne koszty utrzymania oprogramowania.

Niska kultura ponownego użycia wytworzonych komponentów projektów i oprogramowania; niski stopień powtarzalności poszczególnych przedsięwzięć.

Długi i kosztowny cykl tworzenia oprogramowania, wysokie prawdopodobieństwo niepowodzenia projektu programistycznego.

Długi i kosztowny cykl życia SI, wymagający stałych (często globalnych) zmian.

Eklektyczne, niesystematyczne narzędzia i języki programowania.

PROBLEMY IO (2)

Frustracje projektantów oprogramowania i programistów wynikające ze zbyt szybkiego postępu w zakresie języków, narzędzi i metod oraz uciążliwości i długotrwałości procesów produkcji, utrzymania i pielęgnacji oprogramowania.

Uzależnienie organizacji od systemów komputerowych i przyjętych technologii przetwarzania informacji, które nie są stabilne w długim horyzoncie czasowym.

Problemy współdziałania niezależnie zbudowanego oprogramowania, szczególnie istotne przy dzisiejszych tendencjach integracyjnych.

Problemy przystosowania istniejących i działających systemów do nowych wymagań, tendencji i platform sprzętowo-programowych.

Walka z problemami

Stosowanie technik i narzędzi ułatwiających pracę nad złożonymi systemami;

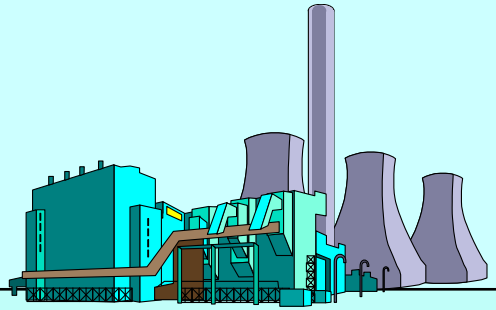
Korzystanie z metod wspomagających analizę nieznanych problemów oraz ułatwiających wykorzystanie wcześniejszych doświadczeń;

Usystematyzowanie procesu wytwarzania oprogramowania, tak aby ułatwić jego planowanie i monitorowanie;

Wytworzenie wśród producentów i nabywców przekonania, że budowa dużego systemu wysokiej jakości jest zadaniem wymagającym profesjonalnego podejścia.

Podstawowym powodem problemów oprogramowania jest złożoność produktów informatyki i procesów ich wytwarzania.

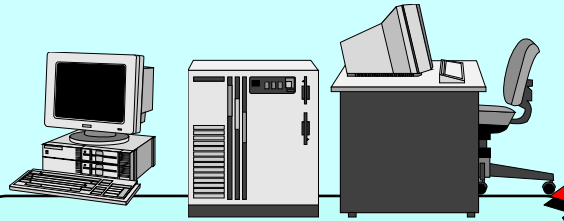
Źródła złożoności projektu oprogramowania



Dziedzina problemowa,
obejmująca ogromną liczbę wzajemnie uzależnionych aspektów i problemów.

Zespół projektantów
podlegający ograniczeniom pamięci, percepcji, wyrażania informacji i komunikacji.

Oprogramowanie:
decyzje strategiczne,
analiza,
projektowanie,
konstrukcja,
dokumentacja,
wdrożenie,
szkolenie,
eksploatacja,
pielęgnacja,
modyfikacja.



Środki i technologie informatyczne:
sprzęt, oprogramowanie, sieć,
języki, narzędzia, udogodnienia.

Potencjalni użytkownicy:
czynniki psychologiczne, ergonomia, ograniczenia pamięci i percepcji, skłonność do błędów i nadużyć, tajność, prywatność.

JAK WALCZYĆ ZE ZŁOŻONOŚCIĄ ?

***Zasada dekompozycji*:**
rozdzielenie złożonego problemu na podproblemy, które można rozpatrywać i rozwiązywać niezależnie od siebie i niezależnie od całości.

***Zasada abstrakcji*:**
eliminacja, ukrycie lub pominięcie mniej istotnych szczegółów rozważanego przedmiotu lub mniej istotnej informacji; wyodrębnianie cech wspólnych i niezmiennych dla pewnego zbioru bytów i wprowadzaniu pojęć lub symboli oznaczających takie cechy.

***Zasada ponownego użycia*:**
wykorzystanie wcześniej wytworzonych schematów, metod, wzorców, komponentów projektu, komponentów oprogramowania, itd.

***Zasada sprzyjania naturalnym ludzkim własnościom*:**
dopasowanie modeli pojęciowych i modeli realizacyjnych systemów do wrodzonych ludzkich własności psychologicznych, instynktów oraz mentalnych mechanizmów percepcji i rozumienia świata.

ZROZUMIENIE TEGO CO TRZEBA ZROBIĆ

Projektant i programista muszą dokładnie wyobrazić sobie problem oraz metodę jego rozwiązania. Zasadnicze procesy tworzenia oprogramowania zachodzą w ludzkim umyśle i nie są związane z jakimkolwiek językiem programowania.

Pojęcia ***modelowania pojęciowego*** (*conceptual modeling*) oraz ***modelu pojęciowego*** (*conceptual model*) odnoszą się procesów myślowych i wyobrażeń towarzyszących pracy nad oprogramowaniem.

Modelowanie pojęciowe jest wspomagane przez środki wzmacniające ludzką pamięć i wyobraźnię. Służą one do przedstawienia rzeczywistości opisywanej przez dane, procesów zachodzących w rzeczywistości, struktur danych oraz programów składających się na konstrukcję systemu.

METODYKA (METODOLOGIA)

Metodyka jest to zestaw pojęć, notacji, modeli, języków, technik i sposobów postępowania służący do analizy dziedziny stanowiącej przedmiot projektowanego systemu oraz do projektowania pojęciowego, logicznego i/lub fizycznego.

Metodyka jest powiązana z **notacją** służącą do dokumentowania wyników faz projektu (pośrednich, końcowych), jako środek wspomagający ludzką pamięć i wyobraźnię i jako środek komunikacji w zespołach oraz pomiędzy projektantami i klientem.

Metodyka ustala:

- fazy projektu, role uczestników projektu,
- modele tworzone w każdej z faz,
- scenariusze postępowania w każdej z faz,
- reguły przechodzenia od fazy do kolejnej fazy,
- notacje, których należy używać,
- dokumentację powstającą w każdej z faz.

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę

INŻYNIERIA OPROGRAMOWANIA

wykład 2:
MODELE PROCESU WYTWARZANIA OPROGRAMOWANIA

(na podstawie wykładów prof. K. Subiety, Instytut Informatyki PAN)

dr inż. Leszek Grocholski

Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

PROCES WYTWARZANIA OPROGRAMOWANIA

Klasyczne elementy procesu :

- Faza wstępna: określenie strategicznych celów, planowanie i definicja projektu
- Określenie wymagań klienta
- Analiza: określenie dziedziny, wymagań systemowych
- Projektowanie: projektowanie pojęciowe, projektowanie logiczne
- Implementacja/konstrukcja: rozwijanie, testowanie, dokumentacja
- Testowanie integracyjne
- Dokumentacja
- Instalacja
- Przygotowanie użytkowników, akceptacja, szkolenie
- Działanie, włączając wspomaganie tworzenia aplikacji
- Utrzymanie, konserwacja, pielęgnacja
- Wycofanie oprogramowania

Wytwarzania oprogramowania a procesy w życiu oprogramowania

Uwaga:
Wytwarzanie oprogramowania to tylko jedna z grup procesów w życiu oprogramowania!

Dokładnie procesy w cyklu życia oprogramowania wymienia i opisuje norma ISO/IEC 12207.

Z kolei procesy w cyklu życia systemów wymienia i opisuje norma ISO/IEC 15288.

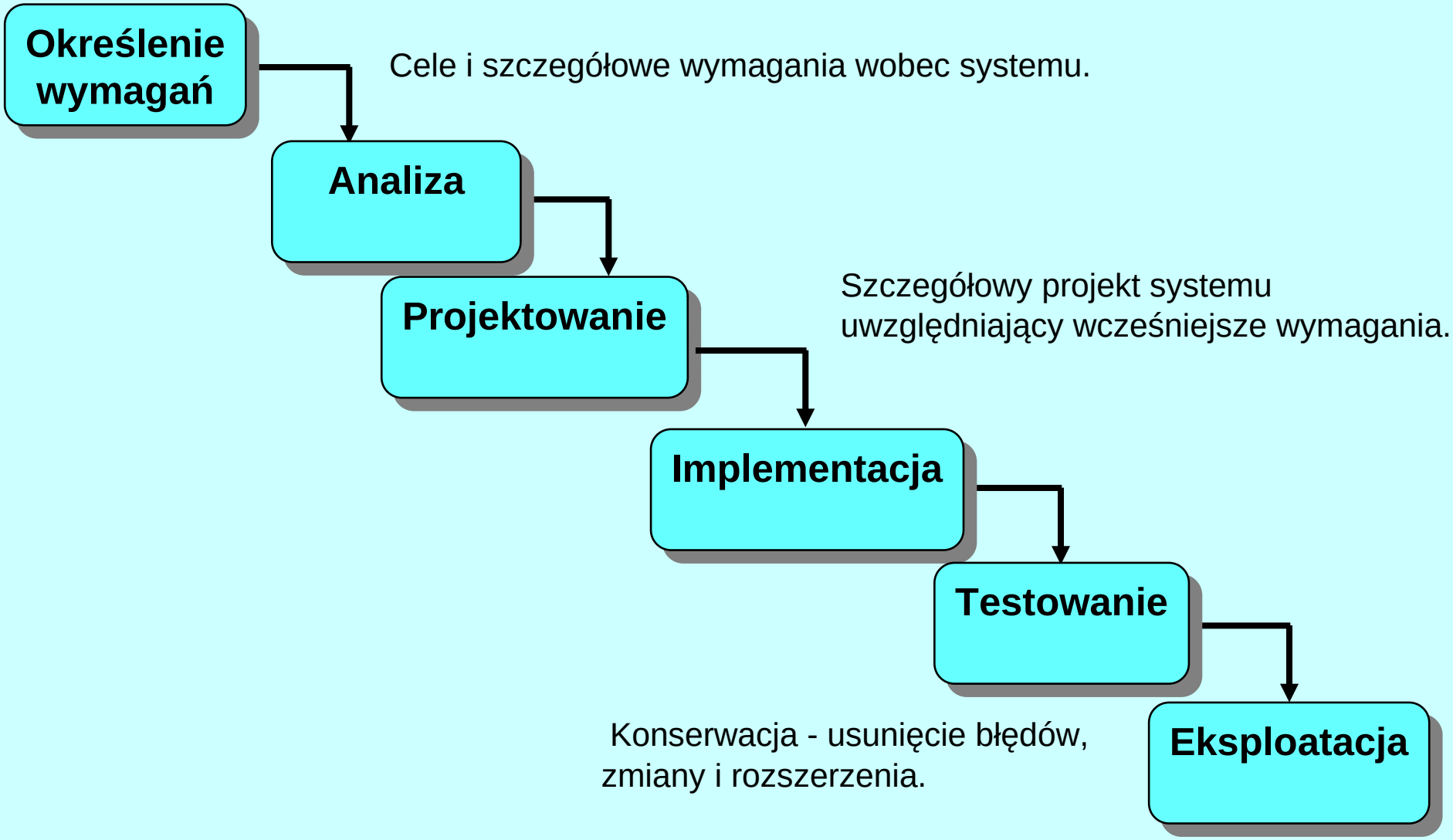
Modele cyklu wytwarzania oprogramowania

- Model kaskadowy (wodospadowy)
- Modele iteracyjne
- Model spiralny
- Prototypowanie
- Montaż z gotowych komponentów

Tego rodzaju modeli (oraz ich mutacji) jest bardzo dużo.

waterfall model

Model kaskadowy (wodospadowy)

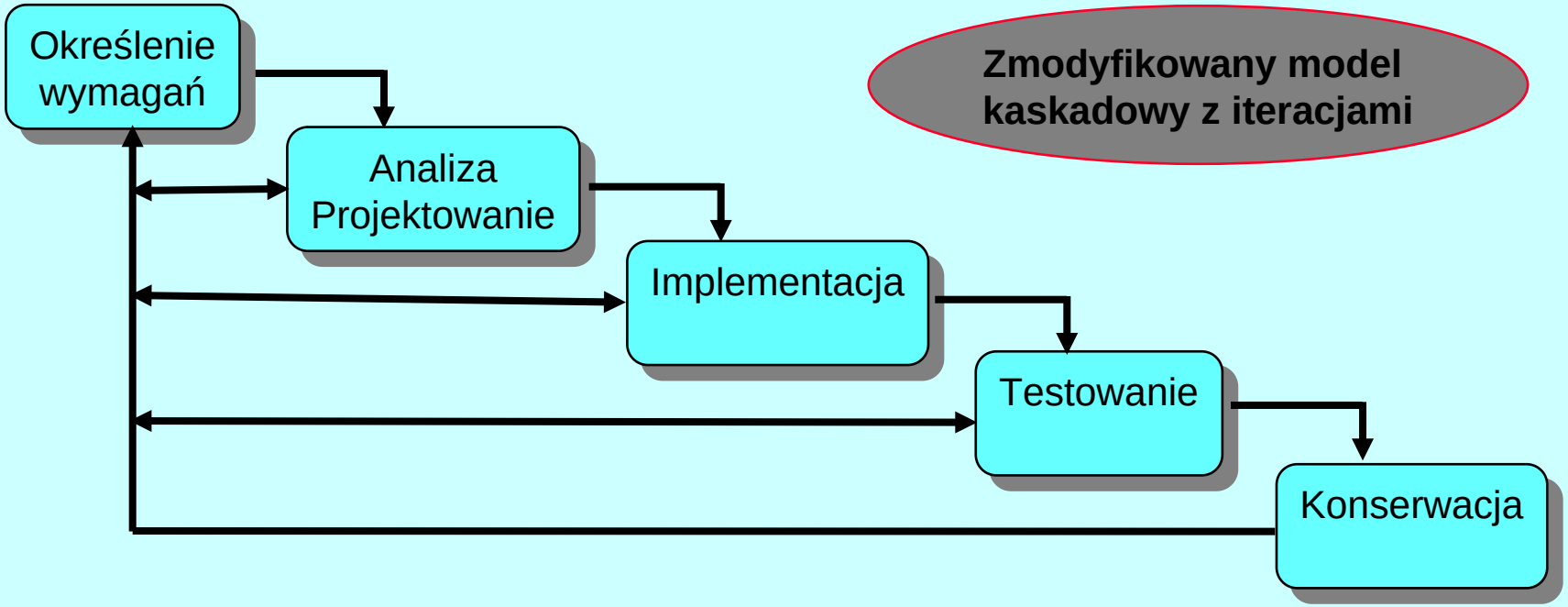


KRYTYKA MODELU KASKADOWEGO

Istnieją zróżnicowane poglądy co do przydatności praktycznej modelu kaskadowego. Podkreślane są następujące wady:

- Niemożliwość określenia (na początku i nie tylko)wszystkich wymagań
- Brak możliwości uwzględniania zmian wymagań
- Wysoki koszt błędów popełnionych we wczesnych fazach

Z drugiej strony, jest on niezbędny dla planowania, harmonogramowania, monitorowania i rozliczeń finansowych.



Model spiralny

spiral model

Planowanie: Ustalenie celów produkcji kolejnej wersji systemu

Analiza ryzyka
(ew. budowa prototypu)

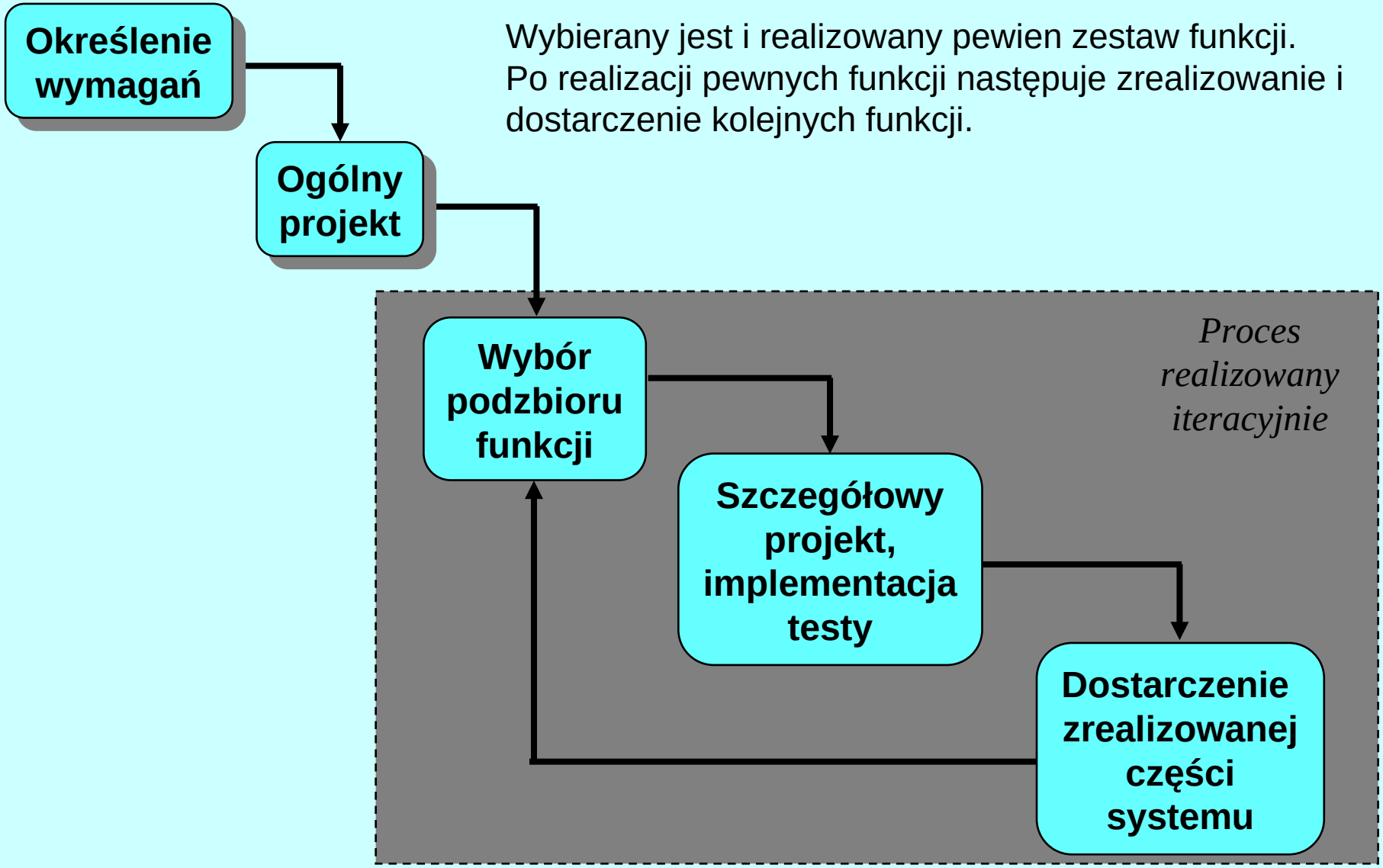
Konstrukcja
(model kaskadowy)

Atestowanie (przez klienta). Jeżeli ocena jest w pełni pozytywna, rozpoczynany jest kolejny cykl.

Istnieje wiele wariantów tego modelu.

Realizacja przyrostowa (odmiana m. spiralnego)

incremental development



Wybierany jest i realizowany pewien zestaw funkcji.
Po realizacji pewnych funkcji następuje zrealizowanie i dostarczenie kolejnych funkcji.

Prototypowanie

prototyping

Sposób na uniknięcie zbyt wysokich kosztów błędów popełnionych w fazie określania wymagań. Zalecany w przypadku, gdy określenie początkowych wymagań jest stosunkowo łatwe. **Przykłady: tworzenie makiet formatek (ekranów)**

Fazy:
- ogólne określenie wymagań
- budowa prototypu
- weryfikacja prototypu przez klienta
- pełne określenie wymagań
- realizacja pełnego systemu zgodnie z modelem kaskadowym/iteracyjnym

Cele
- wykrycie nieporozumień pomiędzy klientem a twórcami systemu
- wykrycie brakujących funkcji
- wykrycie trudnych usług
- wykrycie braków w specyfikacji wymagań

Korzyści:
- możliwość demonstracji makiety systemu
- możliwość szkoleń zanim zbudowany zostanie pełny system

Cechy prototypowania

- **Niepełna realizacja**: objęcie tylko części funkcji

- **„Wyklikiwanie”**: np. Oracle Express

- **Wykorzystanie gotowych komponentów**

- **Generatory interfejsu użytkownika**: wykonywany jest wyłącznie interfejs, wnętrze systemu jest “puste”.

- **Szybkie programowanie** (*quick-and-dirty*): normalne programowanie, ale bez zwracania uwagi na niektóre jego elementy, np. zaniechanie testowania.

Podczas implementacji następuje stopniowe, ewolucyjne przejście od prototypu do końcowego systemu. To oczywiście trwa. Należy starać się nie dopuścić do sytuacji, aby klient miał wrażenie, że prototyp jest prawie ukończonym produktem.
Po fazie prototypowania najlepiej prototyp skierować do archiwum.

Montaż z gotowych komponentów

Kładzie nacisk na możliwość redukcji nakładów poprzez wykorzystanie podobieństwa tworzonego oprogramowania do wcześniej tworzonych systemów oraz wykorzystanie gotowych komponentów dostępnych na rynku. **Klasyka dla systemów ERP.**

Temat jest określany jako ponowne użycie (reuse)

Metody
- zakup elementów ponownego użycia od dostawców
- przygotowanie elementów poprzednich przedsięwzięć do ponownego użycia

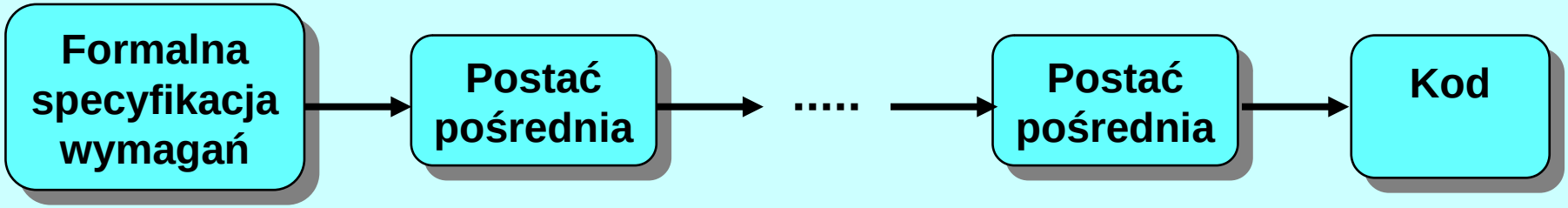
Zalety
- wysoka niezawodność
- zmniejszenie ryzyka
- efektywne wykorzystanie specjalistów
- narzucenie standardów

Wady
- dodatkowy koszt przygotowania elementów ponownego użycia
- ryzyko uzależnienia się od dostawcy elementów
- niedostatki narzędzi wspomagających ten rodzaj pracy.

Formalne transformacje

formal transformations

Postulowany w ramach tzw. nurtu formalnego w inżynierii oprogramowania.



Transformacje są wykonywane bez udziału ludzi (czyli w istocie, język specyfikacji wymagań jest nowym “cudownym” językiem programowania).

Tego rodzaju pomysły nie sprawdziły się w praktyce. Nie są znane szersze (są znane pewne) ich zastosowania. Metody matematyczne nie są w stanie utworzyć pełnej metodyki projektowania, gdyż metodyki włączają wiele elementów (np. psychologicznych) nie podlegających formalnemu traktowaniu. Metody matematyczne mogą jednak wspomagać pewne szczegółowe tematy (tak jak w biologii, ekonomii i innych dziedzinach), np. obliczanie pewnych mierzalnych charakterystyk oprogramowania.

RUP – Rational Unified Process

RUP – proces wytwarzania oprogramowania opracowany przez firmę Rational Software obecnie IBM Rational.

Obecnie zapomniany.

Firma IBM Rational jest autorem popularnego oprogramowania IBM Rational Architect stosowanego w bardzo wielu firmach, też we Wrocławiu
-Siemens, VOLVO IT, Asseco, Signity

Oprogramowanie to zawiera m in. pakiety do:
- zarządzania wymaganiami (Requisite Pro),
- zarządzania wersjami (Clear Case),
- edytor UML.

FAZY RUP


Początkowa
Opracowywanie
Konstrukcja
Wdrażanie
10%
30%
50%
10%
Planowanie
Wymagania
Architektura
Projekt implementacji
Implementacja
Integracja
Testowanie, weryfikowanie
Iter. wstępna
Iter. #1, Iter. #2
Iter.#n, Iter. #..., Iter.#m
Iter. #m+1, Iter .#m+2

Działania a fazy i iteracje (1)

W każdej z faz położony jest różny nacisk na różne działania:

- **Faza początkowa (wstępna) :** tu wysiłki koncentrują się na określeniu kontekstu systemu (aktorów), wymagań funkcjonalnych, zakresu odpowiedzialności (głównych wymagań), ograniczeń (wymagań niefunkcjonalnych), kryteriów akceptacji dla produktu finalnego oraz zgrubnego planu realizacji projektu.

- **Faza opracowywania:** poświęca się wiele uwagi wymaganiom, ale też prowadzone są prace projektowe i implementacyjne (w celu określenia projektu architektury i wytworzenia prototypu, stanowiącego podstawę dla dalszych prac) oraz planuje się MITYGACJE technicznych ryzyk poprzez testowanie różnych rozwiązań i narzędzi.

- **Faza budowy (konstrukcji):** nacisk jest położony na prace projektowe i implementacyjne. W tej fazie prototyp jest rozwijany, aż do uzyskania pierwszego operacyjnego produktu.

- **Faza przekazania (wdrażania):** prace są skoncentrowane na zapewnieniu produktowi odpowiedniej jakości: usuwa się błędy, ulepsza i/lub uzupełnia o brakujące elementy, wypełnia bazę, trenuje użytkowników.

Działania a fazy i iteracje (2)

FAZA OPRACOWANIA

✓Podstawowe zadania: (1) przeprowadzić „bardziej dokładną” analizę, (2) wypracować fundamenty dla architektury, (3) wyeliminować elementy obarczone największym ryzykiem, (4) uszczegółowić plan projektu (czyli skonstruować szczegółowe plany dla iteracji).

Decyzje architektoniczne muszą bazować na zrozumieniu całości systemu: jego funkcjonalności i ograniczeń (“zrozumienie z perspektywy na milę szerokiej i na cal głębokiej”).

✓ **<u>Faza opracowywania – najbardziej krytyczna z czterech faz</u>** (wysoki koszt, wysokie ryzyko): wymagania, architektura, plany powinny osiągnąć stabilną postać, ryzyka muszą być zmitygowane, tak aby była możliwa realna werfikacja zaplanowanych kosztów i czasu potrzebnego na ukończenie projektu.

✓ Bada się różne rozwiązania budując prototyp(y) – w jednej lub kilku iteracjach (zakres, rozmiar, nowości, inne ryzyka). Budowa prototypów ułatwia mitygowanie ryzyk oraz ustanawianie kompromisów między wymaganiami a możliwościami środowiska implementacji.

Planowanie z uwzględnieniem iteracji (1)

Planowanie projektu z uwzględnieniem iteracji wymaga rozstrzygnięcia kwestii, które nie istniały przy zastosowaniu tradycyjnego podejścia kaskadowego, jak np.:

- Jak wiele iteracji będzie potrzebne?
- Jak długo powinna trwać każda z nich?
- W jaki sposób należy ustalić cele i „zawartość” każdej iteracji?
- Jak należy śledzić postęp prac w trakcie realizacji iteracji?

Planowanie dużych, złożonych projektów

Ambitne próby planowania realizacji całości dużych, złożonych projektów z dokładnością „co do minuty”, wyróżniania działania i przypisywania osób do zadań na kilka miesięcy czy kilka lat do przodu, jak wykazała praktyka − są z góry skazane na niepowodzenie.
Tworzono diagramy Gantt’a (czy semantyczne sieci działania), które pokrywały całe ściany, a mimo to projekty kończyły się niepowodzeniem.
Skuteczna realizacja szczegółowych planów wymaga posiadania zarówno dobrego zrozumienia problemu, jak i stabilnych wymagań, stabilnej architektury oraz ukończonego choć jednego podobnego projektu,

Planowanie z uwzględnieniem iteracji (2)

Innymi słowy, jak można planować zbudowanie przez osobę ***X*** w tygodniu ***n*** modułu ***Y*** skoro w momencie planowania, w żaden sposób nie da się w tym czasie w ogóle wydedukować potrzeby posiadania modułu ***Y***? Planowanie szczegółowe jest możliwe w tych dziedzinach inżynieryjnych, gdzie wymagania są mniej lub bardziej zestandaryzowane i stabilne, gdzie kolejność zadań jest wystarczająco uporządkowana. Np. budując budynek nie można wznosić jednocześnie piętra pierwszego i czwartego. Po pierwszym musi powstać drugie, potem trzecie, aby można było zająć się czwartym.

RUP rekomenduje, by realizacja projektu była opierana na dwóch rodzajach planów, różniących się stopniem szczegółowości: **plan faz (zgrubny – nie więcej niż jedna, dwie strony opisu) i seria planów dla iteracji (szczegółowych)**.

Plan faz

Tworzony jest jeden plan faz, **jeden na potrzeby całego projektu**. Plan faz obejmuje z reguły jeden cykl − czasami, stosownie do potrzeb, kilka kolejnych cykli. Tworzony jest bardzo wcześnie, w fazie początkowej. W każdej momencie, może być poddawany modyfikacji.

Planowanie z uwzględnieniem iteracji (3)

Plan faz zawiera poniższe elementy:

- **Daty głównych** kamieni milowych:
  - **LCO** – określenie celów danego cyklu rozwijania projektu (koniec fazy początkowej; projekt ma dobrze określony zakres odpowiedzialności i zapewnione środki na realizację).
  - **LCA** – specyfikacja architektury (koniec fazy opracowywania; stabilizacja architektury).
  - **IOC** – osiągnięta początkowa zdolność operacyjna (koniec fazy konstrukcji; pierwsze beta).
  - Wypuszczenie produktu (koniec fazy wdrażania i koniec danego cyklu).

- **Profile zespołowe** --> jakie zasoby ludzkie są wymagane na poszczególnych etapach cyklu.
- **Specyfikację mniej ważnych kamieni milowych** --> data zakończenia każdej iteracji wraz ze specyfikacją jej celów (o ile można je zidentyfikować).

Planowanie z uwzględnieniem iteracji (4)

Plan iteracji
Tworzony jest **jeden plan dla każdej iteracji** (plan szczegółowy). Projekt z reguły posiada dwa plany iteracji aktywne w danym momencie:

- Plan dla bieżącej iteracji: jest wykorzystywany do śledzenia postępów prac w trakcie realizacji iteracji.
- Plan dla następnej iteracji: jest tworzony począwszy od połowy bieżącej iteracji i jest prawie (?) gotowy przy jej końcu.

Plan iteracji tworzony jest za pomocą tradycyjnych technik (diagramy Gantt'a, itp.) i narzędzi do planowania (Microsoft Project, itp.). Cel – definiowanie zadań, przypisanie zadań do osób i zespołów. Plan zawiera ważne daty, takie jak np.: zakończenie budowy „czegoś", dostarczenie komponentów z innej organizacji czy terminy głównych przeglądów.

RUP – przyjmując za podstawę proces iteracyjny i niekończącą się akomodację zmian celów i taktyki – zakłada tym samym, że nie istnieją racjonalne powody, by marnować czas na przygotowywanie szczegółowych planów dla dłuższych horyzontów czasowych: takie plany są trudne do zarządzania, szybko stają się nieaktualne oraz z reguły są ignorowane przez organizacje wykonawcze).

Planowanie z uwzględnieniem iteracji (5)

Iteracje w fazie początkowej

- W fazie początkowej często uważa się, że iteracji nie ma, tzn. nie ma czegoś, co można by nazwać „prawdziwą iteracją” (mini-projektem): żaden kod nie jest produkowany, wykonywane są przede wszystkim działania związane z rozpoznawaniem, planowaniem i marketingiem.

Czasami jednak iteracja jest potrzebna, po to by:

- Zbudować prototyp, by przekonać siebie czy sponsora, że proponowane rozwiązanie (lepiej: pomysł na rozwiązanie) jest dobre.

- Zbudować prototyp, w celu mitygowania ryzyk technicznych (np. eksploracja nowych technologii czy nowych algorytmów) czy udowodnienia, że cele wydajnościowe są osiągalne.

- Przyspieszyć wdrażanie narzędzi i procesów w organizacji.

Planowanie z uwzględnieniem iteracji (6)

Iteracje w fazie opracowywania

Plany dla iteracji w fazie opracowywania muszą być budowane w oparciu o trzy, najważniejsze w tej fazie elementy: ryzyka, własności krytyczne dla osiągnięcia sukcesu i pokrycie :

- Plany powinny być budowane tak, by większość ryzyk została zmitygowana lub „wysłana na emeryturę” właśnie w fazie opracowywania.

- Ryzyka są bardzo ważne, ale to nie mitygacja ryzyk jest celem ostatecznym. **Ostatecznym celem jest uzyskanie produktu, posiadającego własności uznane za krytyczne dla osiągnięcia sukcesu.**

- Ponieważ podstawowym celem fazy opracowywania jest budowa stabilnej architektury, plany muszą być konstruowane tak, by architektura adresowała wszystkie aspekty oprogramowania (pokrycie). Jest to ważne, bo architektura stanowi bazę dla dalszego planowania.

Podsumowanie (1)

Proces sekwencyjny jest odpowiedni dla realizacji małych projektów, czy tych z niewielką liczbą ryzyk (mało nowości, znane technologie, znana dziedzina problemowa, doświadczeni ludzie), natomiast niezbyt nadaje się dla projektów dużych, z dużą liczbą ryzyk.

Proces iteracyjny jest realizowany w oparciu o sekwencję iteracji. W trakcie każdej iteracji, opartej o model kaskadowy, wykonywane są działania związane z wymaganiami, tworzeniem projektu implementacji, implementacją i szacowaniem rezultatów.

W RUP w celu ułatwienia zarządzania procesem realizacji, iteracje zostały pogrupowane w cztery fazy: początkową, opracowywania, konstrukcji i wdrażania. W każdej z faz położono różny nacisk na różne działania.

Podejście iteracyjne ułatwia dokonywanie zmian (związanych z użytkownikiem, rynkiem czy technologiami), mitygację ryzyk, wprowadzanie technologii ponownego użycia, podnoszenie umiejętności członków zespołu, ulepszanie procesu jako takiego – wszystko to w efekcie skutkuje uzyskaniem lepszej jakości produktu finalnego.

Podsumowanie (2)

RUP − jako proces “prowadzony” w oparciu o przypadki użycia − uczynił z przypadków bazę dla całego procesu budowania produktu tworząc w oparciu o nie rodzaj języka stanowiącego podstawę do komunikacji między wszystkimi osobami zaangażowanymi w realizację projektu.

W RUP przypadki użycia wspomagają członków zespołu przy:

- planowaniu iteracji,
- budowie i walidacji stabilnej architektury systemu,
- definiowaniu przypadków i procedur testowych w modelu testów,
- tworzeniu dokumentacji użytkownika,
- rozmieszczaniu aplikacji (ang. deployment).

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę

INŻYNIERIA OPROGRAMOWANIA

wykład 3:
FAZA WSTĘPNA

dr inż. Leszek Grocholski

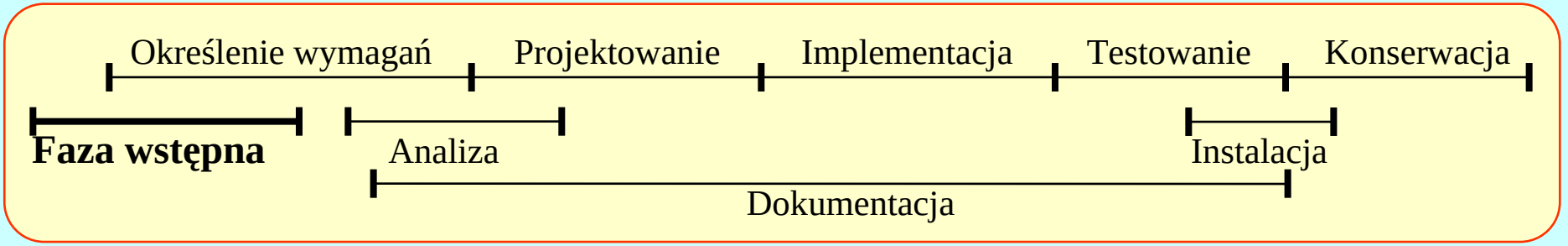
Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

Plan wykładu

1. Analiza biznesowa
2. Analiza potrzeb klienta
3. Zakres i kontekst przedsięwzięcia
4. Podejmowane decyzje
5. Harmonogram przedsięwzięcia
6. Szacowanie kosztu oprogramowania
7. Studium wykonalności

Faza wstępna (studium wykonalności)

feasibility study



Celem FAZY WSTĘPNEJ jest podjęcie decyzji o rozpoczęciu projektu wytworzenia (zamówienia) i wdrożenia oprogramowania.

Nazywana także **strategicznym planem rozwoju informatyzacji** (SPRI) lub **studium osiągalności**.

Czynności w fazie wstępnej -> strona zamawiającego

PRZEPROWADZENIE ANALIZY BIZNESOWEJ,

jej etapy to:

1. Zidentyfikowanie problemu i jego źródła (problem, potrzeba zmiany)
2. Opisanie działania organizacji po wyeliminowaniu problemu
3. Opis możliwych rozwiązań problemu (nie tylko system informatyczny !)
4. Jeżeli problem ma być rozwiązany przy pomocy systemu informatycznego to:
   - Określenie najważniejszych wymagań
   - Propozycja kilku możliwych rozwiązań (sposobów realizacji systemu)
   - Oszacowanie kosztów oprogramowania i projektu
   - Analiza rozwiązań
   - Prezentacja wyników analizy biznesowej
5. Podjęcie decyzji o zamówieniu (ew. nie zamówieniu) oprogramowania

Czynności w fazie wstępnej –> strona wykonawcy

1. Analiza co i dlaczego klient potrzebuje

   Najlepiej przez dokonanie serii rozmów (wywiadów) z przedstawicielami klienta

2. Określenie celów przedsięwzięcia z punktu widzenia klienta

3. Określenie zakresu oraz kontekstu przedsięwzięcia

4. Ogólne określenie wymagań, wykonanie zgrubnej analizy i koncepcji systemu

5. Propozycja kilku możliwych rozwiązań (sposobów realizacji systemu)

6. Orientacyjne oszacowanie kosztów rozwiązań

7. Analiza rozwiązań

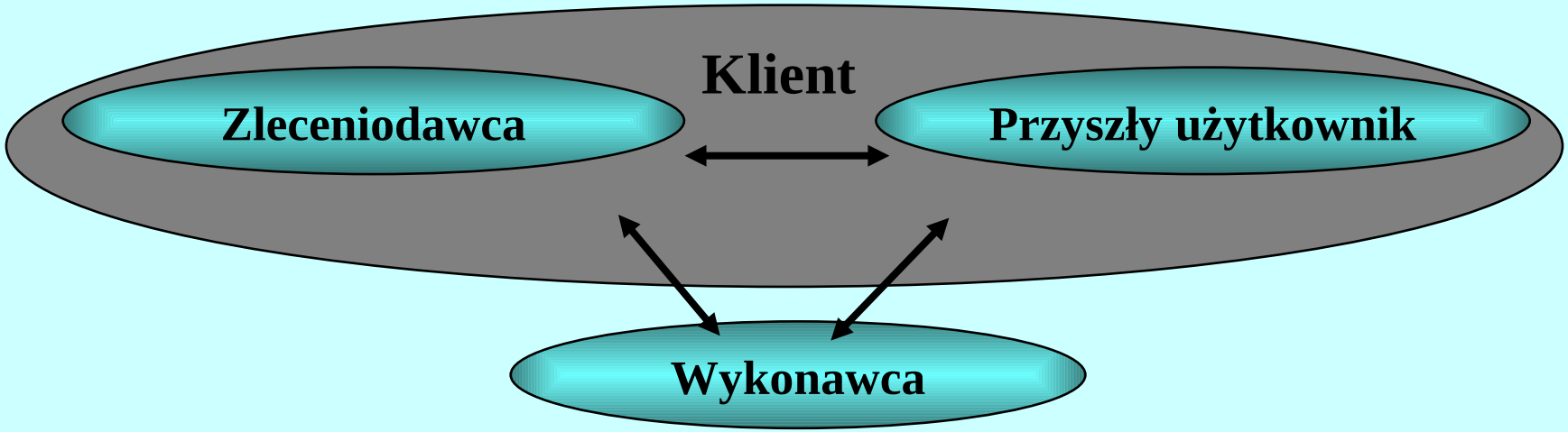
8. Określenie wstępnego harmonogramu przedsięwzięcia oraz struktury zespołu realizatorów

9. Określenie standardów, zgodnie z którymi realizowane będzie przedsięwzięcie

10. Prezentacja wyników fazy wstępnej przedstawicielom klienta oraz korekta wyników

11. Podjęcie decyzji o wytworzeniu oprogramowania dla klienta

Współpraca z klientem



Po stronie klienta wśród interesariuszy warto wyróżnić:
zleceniodawcę (sponsora) i przyszłych użytkowników.
Należy starać się uwzględnić kryteria wielu stron, ale należy pamiętać, że system będzie głównie oceniany przez przyszłych użytkowników.

Ważnym elementem fazy wstępnej jest jasne określenie **celów** przedsięwzięcia z punktu widzenia klienta. Nie zawsze są one oczywiste, co często powoduje nieporozumienia pomiędzy klientem i wykonawcą.

Równie ważne jest określenie **ograniczeń klienta** (np. finansowych, infrastruktury, zasobów ludzkich, czasu wdrożenia, itd.)

Przykład: program podatkowy

Firma rachunkowa zajmuje się m.in. przygotowaniem formularzy zeznań podatkowych (PIT-ów) dotyczących podatku dochodowego dla indywidualnych podatników.

Ponieważ liczba klientów tego rodzaju usługi jest duża, a w dodatku muszą być obsłużeni w większości w marcu i kwietniu, firma widzi konieczność opracowania systemu komputerowego wspomagającego ten typ działalności.

Cele systemu:
Ogólnie świadczenie udoskonalonych usług, tzn w tym przypadku
- przyśpieszenie obsługi klientów,
- zmniejszenie ryzyka popełnienia błędów.

Przykład: system informacji geograficznej - SIG

Firma TURYSTYCZNA widzi możliwość sprzedaży rynkowej prostego systemu informacji geograficznej (mapy komputerowej).

Miałby to być system łączący w sobie możliwość przeglądania bitowej mapy pewnego obszaru (np. mapy fizycznej, zdjęcia satelitarnego) wraz z umieszczonymi na tym tle dodatkowymi informacjami opisującymi pewne obiekty znajdujące się na prezentowanym obszarze.

Cele systemu:

- możliwość łatwego, dialogowego projektowania mapy,
- możliwość łatwego i wygodnego przeglądania mapy wraz z informacjami.

Przykład: system harmonogramowania zleceń

Przedsiębiorstwo farmaceutyczne zleciło wykonanie analizy krytycznych procesów funkcjonowania jednego z wydziałów. Jednym z nich jest harmonogramowanie zleceń, które wydział otrzymuje z działu marketingu. Zlecenie oznacza wyprodukowanie pewnej ilości konkretnego produktu, przy czym możliwe są dodatkowe wymagania, np. ograniczenie terminu wykonania.

Wymagania:
- uwzględnienie wszelkich ograniczeń, zapewniające praktyczną wykonalność proponowanych harmonogramów
- zapewnienie możliwości “ręcznego” modyfikowania harmonogramu
- opracowanie harmonogramu w formie łatwej do wykorzystania przez kadrę kierowniczą wydziału oraz automatyzacja przygotowania zamówień dla magazynu na półprodukty

Cele przedsięwzięcia z punktu widzenia klienta:
- zwiększenie wydajności pracy wydziału,
- zmniejszenie opóźnień w realizowaniu zleceń.

.

Zakres i kontekst przedsięwzięcia

Zakres przedsięwzięcia: określenie fragmentu procesów informacyjnych zachodzących w organizacji, które będą objęte przedsięwzięciem. Na tym etapie może nie być jasne, które funkcje będą wykonywane przez oprogramowanie, a które przez personel, inne systemy lub standardowe wyposażenie sprzętu.

Kontekst przedsięwzięcia: systemy, organizacje, użytkownicy zewnętrzni, z którymi tworzony system ma współpracować.

Bardzo ważne jest wczesne określenie co ma zostać objęte systemem a co nie.

Przykłady zakresu/kontekstu przedsięwzięcia

Program podatkowy

Zakresem przedsięwzięcia jest działalność jednej firmy rachunkowej, która może mieć dowolną liczbę klientów. Pracownik firmy jest jedynym **systemem zewnętrznym**.

System informacji geograficznej

Zakresem przedsięwzięcia jest projektowanie i przeglądanie prostej mapy komputerowej.
Systemami zewnętrznymi, z którymi system ma współpracować jest projektant mapy i osoba przeglądająca mapę.

System harmonogra-mowania zleceń

Zakresem przedsięwzięcia jest funkcjonowanie komórki wydziału obejmującego przygotowanie harmonogramu wykonywania zleceń.
Systemami zewnętrznymi są: system zarządzający zleceniami, System zawierający informacje o technologicznych możliwościach wydziału produkcyjnego, kadra kierownicza.

Decyzje strategiczne - wykonawca

- Wybór modelu, zgodnie z którym będzie realizowane przedsięwzięcie
- Określenie architektury
- Wybór technik stosowanych w fazach analizy i projektowania
- Wybór środowiska (środowisk) implementacji
- Wybór narzędzia CASE
- Określenie stopnia wykorzystania gotowych komponentów
- Podjęcie decyzji o współpracy z innymi producentami lub zatrudnieniu ekspertów

Zidentyfikowane ograniczenia:

Maksymalne nakłady, jakie można ponieść na realizację przedsięwzięcia

- Dostępny personel
- Dostępne narzędzia
- Ograniczenia czasowe

Po prezentacji wyników dla klienta końcowym wynikiem może być przyjęcie lub odrzucenie oferty twórcy oprogramowania. Faza wstępna jest nieodłączną częścią cyklu produkcji oprogramowania, wobec czego nie powinna być wykonywana na koszt i ryzyko producenta oprogramowania

Studium wykonalności

feasibility study

Wynikiem studium wykonalności są m in:

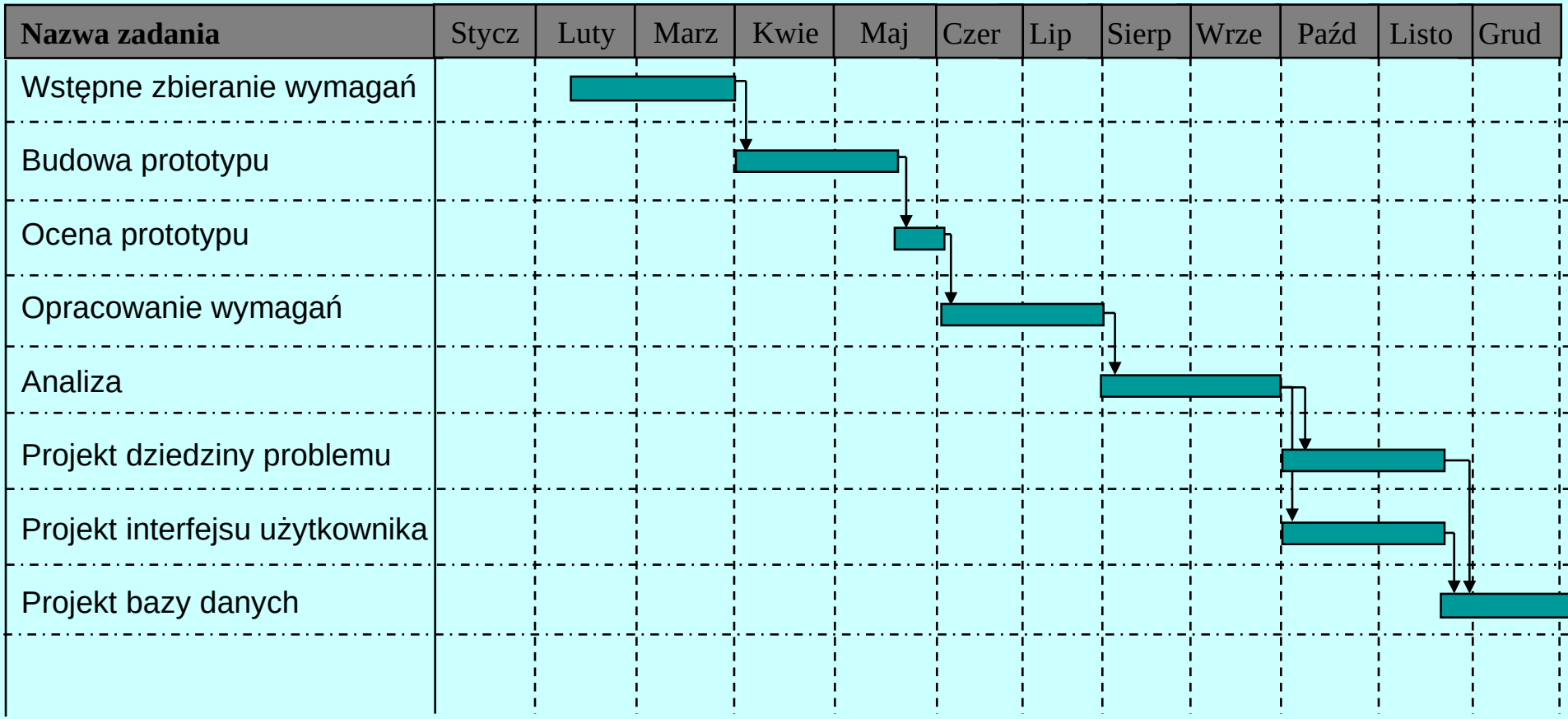
-Rozmiar projektu (np. w punktach funkcyjnych) w porównaniu do rozmiaru zakładanego zespołu projektowego i czasu

- Dostępność zasobów (budżet, personel, kadra)
- Ograniczenia czasowe (krańcowe daty ukończenia projektu, wdrożenia, itd.)
- Warunki wstępne niezbędne do realizacji projektu
- Dostępność oprogramowania oraz narzędzi do rozwoju oprogramowania
- Dostępność sprzętu i sieci
- Dostępność technologii oraz know-how
- Dostępność specjalistów wewnątrz firmy oraz zewnętrznych ekspertów
- Dostępność usług zewnętrznych, kooperantów i dostawców
- Dostępność powierzchni biurowej, środków komunikacyjnych, zaopatrzenia, itd.

Harmonogram przedsięwzięcia

Ustalenie planu czasowego (i/lub wykorzystania zasobów) dla poszczególnych etapów i zadań.

Diagram Gantta



Ocena rozwiązań

Z powodów: wielości celów przedsięwzięcia (czyli kryteriów oceny) lub niepewności (niemożliwości precyzyjnej oceny spodziewanych rezultatów) czy paru możliwych rozwiązań **w fazie wstępnej często rozważa się kilka rozwiązań**.

Częste kryteria oceny:

- koszt
- czas realizacji
- niezawodność
- możliwość ponownego użycia
- przenośność na inne platformy
- wydajność (szybkość)

Prezentacja i porównanie poszczególnych rozwiązań w postaci tabelarycznej

Rozwiązanie	**A**	**B**	**C**
Koszt (tys. zł)	120	80	175
Czas (miesiące)	33	30	36
Niezawodność (błędy/tydzień)	5	9	13
Ponowne użycie (%)	40	40	30
Przenośność (%)	90	75	30
Wydajność(transakcje/sek)	0.35	0.75	1

Wybór rozwiązania

Podczas wyboru optymalnego rozwiązania pomija się rozwiązania zdominowane, tj. gorsze wg danego kryterium (lub wszystkich/prawie wszystkich).

Normalizacja wartości dla poszczególnych kryteriów (sprowadzenie do przedziału [0,1])

Przypisanie wag do kryteriów (również może być trudne).

Przykład: łączna ocena za pomocą sumy ważonej

Rozwiązanie	**A**	**B**	**C**	**Waga**
Koszt (tys. zł)	0.58	1	0	3
Czas (miesice)	0.5	1	0	2
Niezawodność (błędy/tydzień)	1	0.5	0	3
Ponowne użycie (%)	1	1	0	1
Przenośność (%)	1	0.75	0	1
Wydajność(transakcje/sek)	0	0,62	1	1.5
Łączna ocena	**7.74**	**9.17**	**1.5**	

Szacowanie kosztu oprogramowania

Szacowanie kosztów przeprowadza się dla każdego z alternatywnych rozwiązań.

Na koszt oprogramowania składają się następujące główne czynniki:

- robocizna ! → czyli informatycy
- koszt sprzętu będącego częścią tworzonego systemu
- koszt wyjazdów i szkoleń
- koszt zakupu narzędzi

Trzy pierwsze czynniki są dość łatwe do oszacowania.
Oszacowanie kosztów oprogramowania jest praktycznie utożsamiane z oszacowaniem nakładu pracy.

Techniki oszacowania pracochłonności

Modele algorytmiczne. Wymagają opisu przedsięwzięcia przez wiele atrybutów liczbowych i/lub opisowych. Odpowiedni algorytm lub formuła matematyczna daje wynik.

Ocena przez eksperta. Doświadczone osoby z dużą precyzją potrafią oszacować koszt realizacji nowego systemu.

Ocena przez analogię (historyczna). Wymaga dostępu do informacji o poprzednio realizowanych przedsięwzięciach. Metoda podlega na wyszukaniu przedsięwzięcia o najbardziej zbliżonych charakterystykach do aktualnie rozważanego i znanym koszcie i następnie, oszacowanie ewentualnych różnic.

Wycena dla wygranej. Koszt oprogramowania jest oszacowany na podstawie kosztu oczekiwanego przez klienta i na podstawie kosztów podawanych przez konkurencję.

Szacowanie wstępujące. Przedsięwzięcie dzieli się na mniejsze zadania, następnie sumuje się koszt poszczególnych zadań.

Modele szacowania kosztów

Historycznie, podstawą oszacowania jest rozmiar systemu liczony w liniach kodu źródłowego. Metody takie są niedokładne, zawodne, sprzyjające patologiom, np. sztucznemu pomnażaniu ilości linii, ignorowaniu komentarzy, itp.

Obecnie stosuje się wiele miar o lepszych charakterystykach (z których będą omówione punkty funkcyjne). Miary te, jakkolwiek niedokładne i oparte na szacunkach, są jednak konieczne. Niemożliwe jest jakiekolwiek planowania bez oszacowania kosztów. Miary dotyczą także innych cech projektu i oprogramowania, np. czasu wykonania, jakości, niezawodności, itd.

Jest bardzo istotne uwolnienie się od religijnego stosunku do miar, tj. traktowanie ich jako obiektywnych wartości “policzonych przez komputer”. Podstawą wszystkich miar są szacunki, które mogą być obarczone znacznym błędem, nierzadko o rząd wielkości. Miary należy traktować jako latarnię morską we mgle - może ona nas naprowadzić na dobry kierunek, może ostrzec przed niebezpieczeństwem. Obowiązuje zasada patrzenia na ten sam problem z wielu punktów widzenia (wiele różnych miar) i zdrowy rozsądek.

Metoda COCOMO

COnstructive COst MOdel

COCOMO jest oparte na kilku formułach pozwalających oszacować całkowity koszt przedsięwzięcia na podstawie oszacowanej liczby linii kodu. Jest to główna słabość tej metody, gdyż:

- liczba ta staje się przewidywalna dopiero wtedy, gdy kończy się faza projektowania architektury systemu; jest to za późno;

- pojęcie “linii kodu” zależy od języka programowania i przyjętych konwencji;

- pojęcie “linii kodu” nie ma zastosowania do nowoczesnych technik programistycznych, np. programowania wizyjnego.

COCOMO oferuje kilka metod

- Metoda podstawowa: prosta formuła dla oceny osobo-miesięcy oraz czasu potrzebnego na całość projektu.

- **Metoda pośrednia**: modyfikuje wyniki osiągnięte przez metodę podstawową poprzez odpowiednie czynniki, które zależą od aspektów złożoności.

- **Metoda detaliczna:** bardziej skomplikowana, ale jak się okazało, nie dostarcza lepszych wyników niż metoda pośrednia.

Metoda punktów funkcyjnych

Function Point Analysis, FPA

Metoda punktów funkcyjnych oszacowuje koszt projektu na podstawie funkcji użytkowych, które system ma realizować. Stąd wynika, ze metoda ta może być stosowana dopiero wtedy, gdy funkcje te są z grubsza znane.

Metoda jest oparta na zliczaniu ilości wejść i wyjść systemu, miejsc przechowywania danych i innych kryteriów. Te dane są następnie mnożone przez zadane z góry wagi i sumowane. Rezultatem jest liczba „punktów funkcyjnych”.

Punkty funkcyjne mogą być następnie modyfikowane zależnie od dodatkowych czynników złożoności oprogramowania.

Istnieją przeliczniki punktów funkcyjnych na liczbę linii kodu, co może być podstawą dla metody COCOMO.

Metoda jest szeroko stosowana i posiada stosunkowo mało wad.
Niemniej, istnieje wiele innych, mniej popularnych metod, posiadających swoich zwolenników.

Metoda Delphi i inne metody

Metoda Delphi zakłada użycie kilku niezależnych ekspertów, którzy nie mogą się ze sobą w tej sprawie komunikować i naradzać. Każdy z nich szacuje koszty i nakłady na podstawie własnych doświadczeń i metod. Eksperci są anonimowi. Każdy z nich uzasadnia przedstawione wyniki.

Koordynator metody zbiera wyniki od ekspertów. Jeżeli znacznie się różnią, wówczas tworzy pewne sumaryczne zestawienie (np. średnią) i wysyła do ekspertów dla ponownego oszacowania. Cykl jest powtarzany aż do uzyskania pewnej zgody pomiędzy ekspertami.

Metoda analizy podziału aktywności (activity distribution analysis):
Projekt dzieli się na aktywności, które są znane z poprzednich projektów. Następnie dla każdej z planowanych aktywność ustala się, na ile będzie ona bardziej (lub mniej) pracochłonna od aktywności już wykonanej, której koszt/nakład jest znany. Daje to szacunek dla każdej planowanej aktywności. Szacunki sumuje się dla uzyskania całościowego oszacowania.

Metody oszacowania pracochłonności: testowania systemu, dokumentacji, wdrożenia, …

Kluczowe czynniki sukcesu

Szybkość pracy. Szczególnie w przypadku firm realizujących oprogramowanie na zamówienie, opóźnienia w przeprowadzeniu fazy wstępnej mogą zaprzepaścić szansę na wygranie przetargu lub na następne zamówienie. Faza ta wymaga więc stosunkowo niedużej liczby osób, które potrafią wykonać pracę w krótkim czasie.

Zaangażowanie kluczowych osób ze strony klienta. Brak akceptacji dla sposobu realizacji przedsięwzięcia ze strony kluczowych osób po stronie klienta może uniemożliwić jego przyszły sukces.

Uchwycenie (ogólne) całości systemu. Podstawowym błędem popełnianym w fazie wstępnej jest zbytnie przywiązanie i koncentracja na pewnych fragmentach systemu. Niemożliwe jest w tej sytuacji oszacowanie kosztów wykonania całości. Łatwo jest też przeoczyć szczególnie trudne fragmenty systemu.

Rezultaty fazy wstępnej

Udostępniony klientowi raport, ew. wewnętrzna analiza, która obejmuje:

- definicję celów przedsięwzięcia
- opis zakresu przedsięwzięcia
- opis systemów zewnętrznych, z którymi system będzie współpracować
- ogólny opis wymagań
- ogólny model systemu
- opis proponowanego rozwiązania
- oszacowanie wymiarów (czasu, kosztów)
- opis metody zarządzania przedsięwzięciem
- wstępny harmonogram prac

Raport oceny rozwiązań, zawierający informację o rozważanych rozwiązaniach oraz przyczynach wyboru jednego z nich.

Opis wymaganych zasobów - pracownicy, oprogramowanie, sprzęt, lokale, ...

Definicje wymaganych standardów.

Ogólny harmonogram.

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę

INŻYNIERIA OPROGRAMOWANIA

wykład 4:
WYMAGANIA

dr inż. Leszek Grocholski

Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

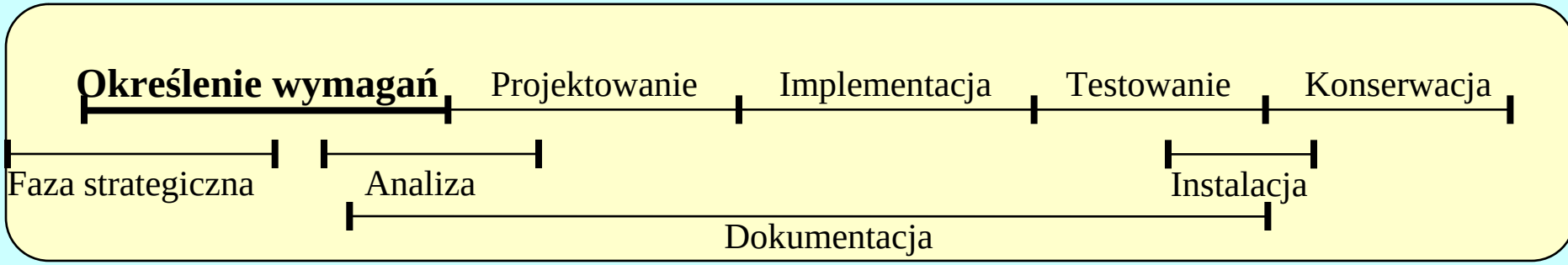
Plan wykładu

1. Problemy dot. określenia wymagań
2. Poziomy ogólności opisu wymagań
3. Ocena jakość opisu wymagań
4. Sposoby identyfikacji wymagań
5. Wymagania funkcjonalne
6. Porządkowanie wymagań wg. ich ważności
7. Wymagania niefunkcjonalne
8. Czynniki uwzględniane przy konstruowaniu wymagań
9. Dokument wymagań

Określenie wymagań

Celem fazy określenia wymagań jest ustalenie wymagań klienta wobec tworzonego systemu. Dokonywana jest zamiana celów klienta na konkretne wymagania zapewniające osiągnięcie tych celów.

Klient NIE ZAWSZE WIE, jakie wymagania zapewnią osiągniecie jego celów. Bardzo często wymagania są sformułowane na wysokim poziomie.
Ta faza nie jest więc prostym zbieraniem wymagań, lecz procesem, w którym klient wspólnie z przedstawicielem producenta konstruuje zbiór wymagań zgodnie z postawionymi celami.



W przypadku dedykowanego (wytwarzanego specjalnie na systemu na zamówienie) analitycy mają bezpośredni kontakt z przedstawicielami klienta. Faza ta wymaga **dużego zaangażowania ze strony klient**a, ze strony przyszłych użytkowników systemu i ekspertów w dziedzinie.

Problemy dot. określenia wymagań

Wymagań nawet dla „małych” systemów jest bardzo dużo. Bardzo trudno je wszystkie zidentyfikować, dokładnie opisać i zweryfikować.

Klient rzadko kiedy wie DOKŁADNIE w jaki sposób osiągnąć założone cele. Tymczasem cele klienta mogą być osiągnięte na wiele sposobów.

Duże systemy są wykorzystywane przez wielu użytkowników.
Ich cele są często sprzeczne.

Różni użytkownicy mogą posługiwać się RÓŻNĄ terminologią mówiąc o tych samych zagadnieniach.

Zleceniodawcy i użytkownicy to często inne osoby. Głos zleceniodawców może być w tej fazie decydujący, chociaż nie zawsze potrafią oni właściwie przewidzieć potrzeby przyszłych użytkowników.

Poziomy opisu wymagań

Definicja wymagań, to ogólny opis w języku naturalnym. Opis taki jest rezultatem wstępnych rozmów z klientem.

Specyfikacja wymagań, to częściowo ustrukturalizowany zapis wykorzystujący zarówno język naturalny, jak i proste, częściowo przynajmniej sformalizowane notacje.

Specyfikacja oprogramowania, to formalny opis wymagań.

Dokładne specyfikacja oznacza bardzo dokładne zdekomponowanie wymagań (najlepiej w pewnym formularzu) na krótkie punkty, których interpretacja nie powinna nastręczać trudności lub prowadzić do niejednoznaczności. Formalna specyfikacja powinna stanowić podstawę dla fazy testowania.

Ocena jakości opisu wymagań

Dobry opis wymagań powinien:

- Być kompletny oraz niesprzeczny.

- Opisywać zewnętrzne zachowanie się systemu a nie sposób jego realizacji.

- Obejmować ograniczenia przy jakich musi pracować system.

- Być łatwy w modyfikacji.

- Brać pod uwagę przyszłe możliwe zmiany wymagań wobec systemu.

- Opisywać zachowanie systemu w niepożądanych lub skrajnych sytuacjach.

Najbardziej typowy błąd w tej fazie: koncentrowanie się na sytuacjach typowych i pomijanie wyjątków oraz przypadków granicznych. Zarówno użytkownicy jak i analitycy mają tendencję do nie zauważania sytuacji nietypowych lub skrajnych.

Zalecenia dla fazy definicji wymagań

Wymagania użytkowników powinny być wyjaśniane poprzez:

-**krytykę,**

-**porównania z istniejącym oprogramowaniem i prototypami.**

Należy osiągnąć porozumienie - consensus pomiędzy projektantami i użytkownikami dotyczący projektowanego systemu.

Wiedza i doświadczenia potencjalnej organizacji podejmującej się rozwoju systemu powinny wspomóc studia nad osiągalnością systemu (omawiane feasibility study).

W wielu przypadkach dużym wspomaganiem jest budowa prototypów.

Uwaga: wymagania użytkowników powinny być: **jasne, jednoznaczne, weryfikowalne, kompletne, dokładne, realistyczne, osiągalne.**

Sposoby rozpoznania wymagań

Studia przeznaczenia systemu. Określenie realistycznych celów systemu, zakresu i metod ich osiągnięcia.

Historyjki użytkownika. Opowiadania użytkowników dot. oczekiwanego sposobu działania systemu.

Wywiady i przeglądy. Wywiady powinny być przygotowane (w postaci listy pytań) i podzielone na odrębne zagadnienia. Podział powinien przykrywać całość tematu a wywiady powinny być przeprowadzone na reprezentatywnej grupie użytkowników. Wywiady powinny doprowadzić do szerokiej zgody i akceptacji projektu.

Studia na istniejącym oprogramowaniem. Dość często nowe oprogramowanie zastępuje stare. Studia powinny ustalić wszystkie dobre i złe strony starego oprogramowania.

Studia wymagań systemowych. Dotyczy sytuacji, kiedy nowy system ma być częścią większego systemu.

Prototypowanie – analiza czy taki system jest potrzebny? Zbudowanie prototypu systemu działającego w zmniejszonej skali, z uproszczonymi interfejsami.

Wymagania funkcjonalne

Opisują „co system ma robić”.
Opisują funkcje (czynności, operacje) wykonywane przez system.
Funkcje te mogą być również wykonywane przy użyciu systemów zewnętrznych.

Określenie wymagania funkcjonalnych obejmuje następujące kwestie:

- Określenie wszystkich rodzajów użytkowników, **którzy będą korzystać** z systemu.

- Określenie wszystkich rodzajów użytkowników, którzy **są niezbędni do działania** systemu (obsługa, wprowadzanie danych, administracja).

- **Dla każdego rodzaju użytkownika określenie funkcji systemu oraz sposobów korzystania z planowanego systemu.**

- Określenie systemów zewnętrznych (obcych baz danych, sieci, Internetu), które będą wykorzystywane podczas działania systemu.

- Ustalenie struktur organizacyjnych, przepisów prawnych, statutów, zarządzeń, instrukcji, itd., które pośrednio lub bezpośrednio określają funkcje wykonywane przez planowany system.

Metody specyfikacji wymagań

Słowa, słowa, słowa …

Język naturalny - najczęściej stosowany. Wady: *niejednoznaczność* powodująca różne rozumienie tego samego tekstu; *elastyczność*, powodująca wyrazić te same treści na wiele sposobów. **Np. historyjki użytkownika.**

Formalizm arytmetyczny (matematyczny). Stosuje się rzadko (dla specyficznych celów).

Język naturalny strukturalny. Język naturalny z ograniczonym słownictwem i składnią. Tematy i zagadnienia wyspecyfikowane w punktach i podpunktach.

Tablice, formularze. Wyspecyfikowanie wymagań w postaci (zwykle dwuwymiarowych) tablic, kojarzących różne aspekty (np. tablica ustalająca zależność pomiędzy typem użytkownika i rodzajem usługi).

Różne rysunki (popularny UML):

- Diagramy blokowe: forma graficzna pokazująca cykl przetwarzania.

- Diagramy kontekstowe: ukazują system w postaci jednego bloku oraz jego powiązania z otoczeniem, wejściem i wyjściem.

- Diagramy przypadków użycia: poglądowy sposób przedstawienia aktorów i funkcji systemu.

Formularz wymagań

W formularzach zapis jest podzielony na konkretne pola, co pozwala na łatwe stwierdzenie kompletności opisu oraz na jednoznaczną interpretację.

Przykład:

Nazwa funkcji	*Edycja dochodów podatnika*
Opis	Funkcja pozwala edytować łączne dochody podatnika uzyskane w danym roku.
Dane wejściowe	Informacje o dochodach pracowników uzyskane uzyskanych z różnych źródeł: kwoty przychodów, koszty uzyskania przychodów oraz zapłaconych zaliczek na poczet podatku dochodowego. Informacje o dokumentach opisujących dochody z poszczególnych źródeł.
Źródło danych wejściowych	Dokumenty oraz informacje dostarczone przez podatnika. Dane wpisywane przez pracownika firmy podatkowej.
Wynik	
Warunek wstępny	Kwota dochodu = kwota przychodu - kwota kosztów (zarówno dla konkretnych dochodów, jak i dla łącznych dochodów podatnika). Łączne kwoty przychodów, kosztów uzyskania dochodów oraz zapłaconych zaliczek są sumami tych kwot dla dochodów z poszczególnych źródeł.
Warunek końcowy	Jak wyżej.
Efekty uboczne	Uaktualnienie podstawy opodatkowania.
Powód	Funkcja pomaga przyśpieszyć obsługę klientów oraz zmniejszyć ryzyko popełnienia błędów.

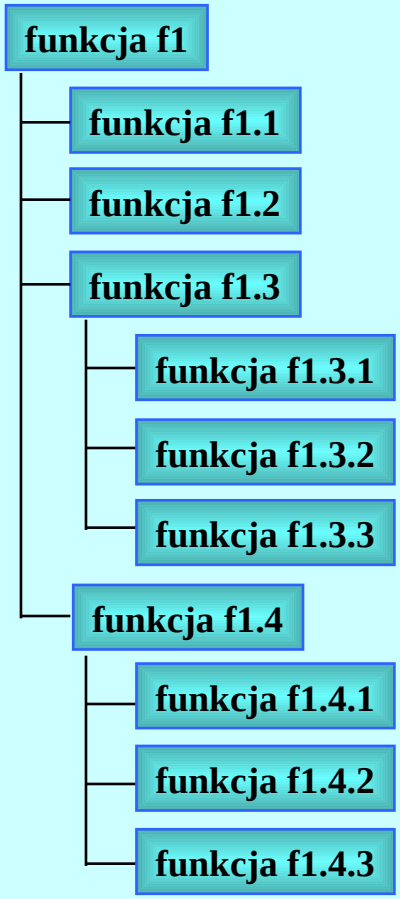
Porządkowanie wymagań

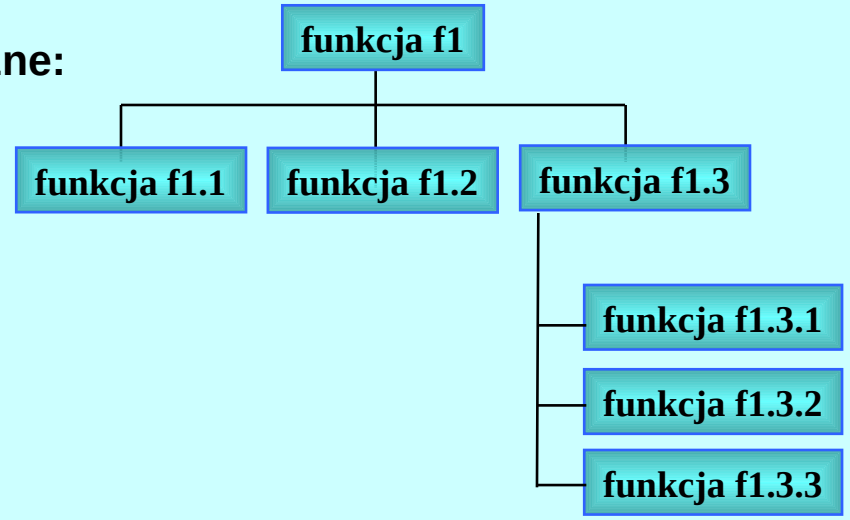
Hierarchia wymagań funkcjonalnych:
Z reguły funkcje można rozbić na podfunkcje.

Format tekstowy:

Funkcja nadrzędna f1
- funkcja f1.1
- funkcja f1.2
- funkcja f1.3
  - funkcja f1.3.1
  - funkcja f1.3.2
  - funkcja f1.3.3
- funkcja f1.4
  - funkcja f1.4.1
  - funkcja f1.4.2
  - funkcja f1.4.3

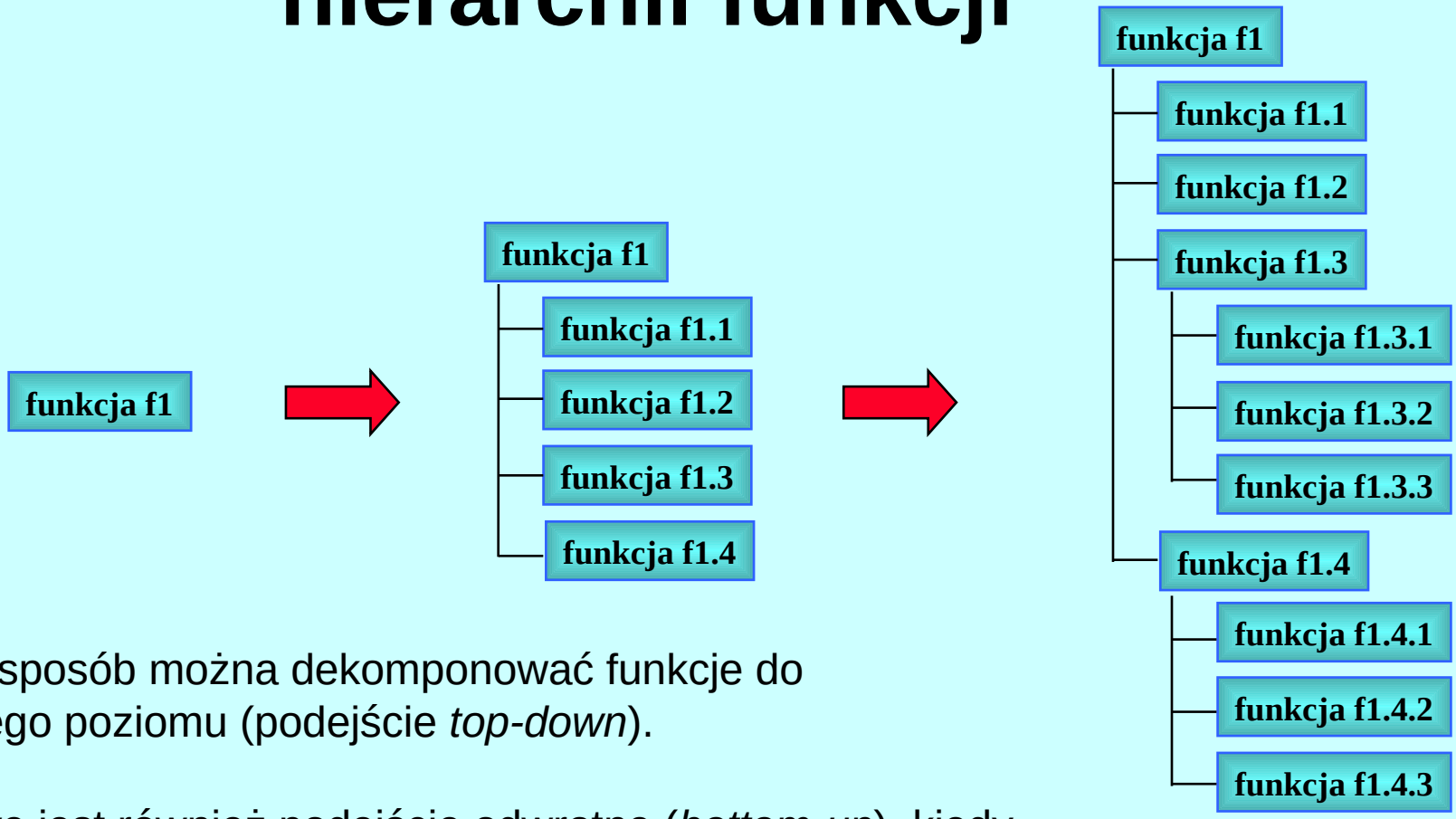
Notacje graficzne:





- Na każdym poziomie ten sam poziom szczegółowości.
- Kolejność funkcji nie ma znaczenia.
- Niektóre funkcje mogą być wykonywane wielokrotnie.
- Wyszczególniać tylko funkcje widoczne dla użytkowników.

Zstępujące konstruowanie hierarchii funkcji



- W ten sposób można dekomponować funkcje do dowolnego poziomu (podejście *top-down*).

- Możliwe jest również podejście odwrotne (*bottom-up*), kiedy składamy kilka funkcji bardziej elementarnych w jedną funkcje bardziej ogólną. Możliwa jest również technika mieszana.

P1 program podatkowy

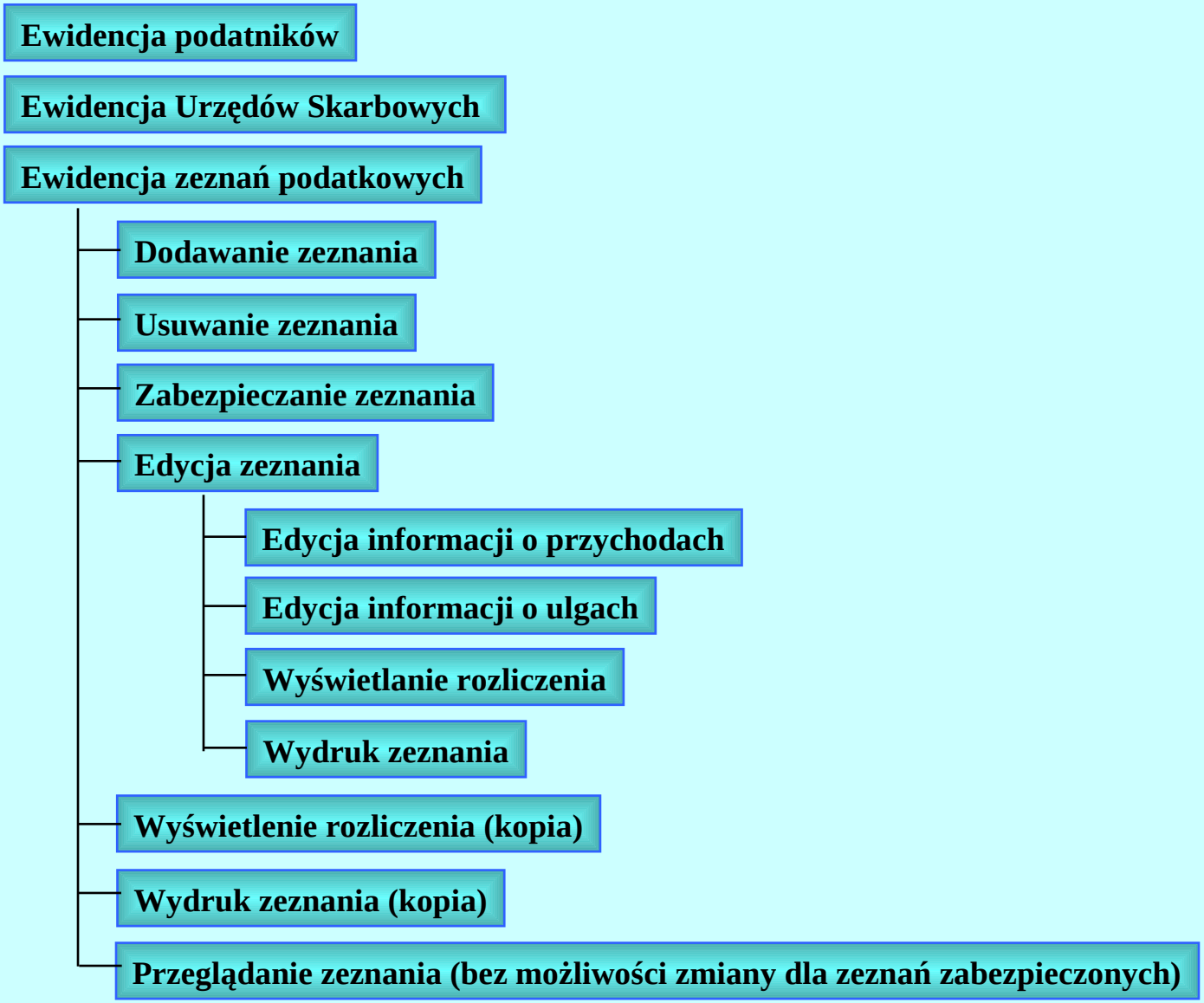
Przykład opowiadania (powieść?) użytkownika

Program ułatwia przygotowanie formularzy zeznań podatkowych (PIT-ów) oraz przechowanie informacji o źródłach przychodów i ulg. Zeznanie może być tworzone przez pojedynczego podatnika lub małżeństwa. Zeznania mogą obejmować informacje o rocznych przychodach (w przypadku małżeństwa z podziałem na przychody męża i żony) oraz o ulgach podatkowych. Przychody podzielone są na klasy ze względu na źródło uzyskania, np. poza-rolnicza działalność gospodarcza, wolny zawód,... . W ramach danej klasy przychodów podatnik mógł osiągnąć szereg przychodów z różnych źródeł.

Wszystkie przychody opisane są przez kwotę przychodu, kwotę kosztów, kwotę zapłaconych zaliczek oraz kwotę dochodu. Powyższe informacje pozwalają obliczyć należny podatek oraz kwotę do zapłaty lub zwrotu. Zeznanie zawiera także informację o podatniku oraz adres Urzędu Skarbowego.

System pozwala wydrukować wzorzec zeznania zawierający wszystkie informacje, jakie podatnik musi umieścić w formularzu. Zeznanie można zabezpieczyć przed dalszymi zmianami (po złożeniu w Urzędzie Skarbowym). System pozwala na tworzenie listy podatników oraz urzędów skarbowych, które mogą być pomocne przy tworzeniu nowego zeznania. Przechowuje także informację o wszystkich złożonych zeznaniach.

Program podatkowy: hierarchia funkcji



P2: system harmonogramowania

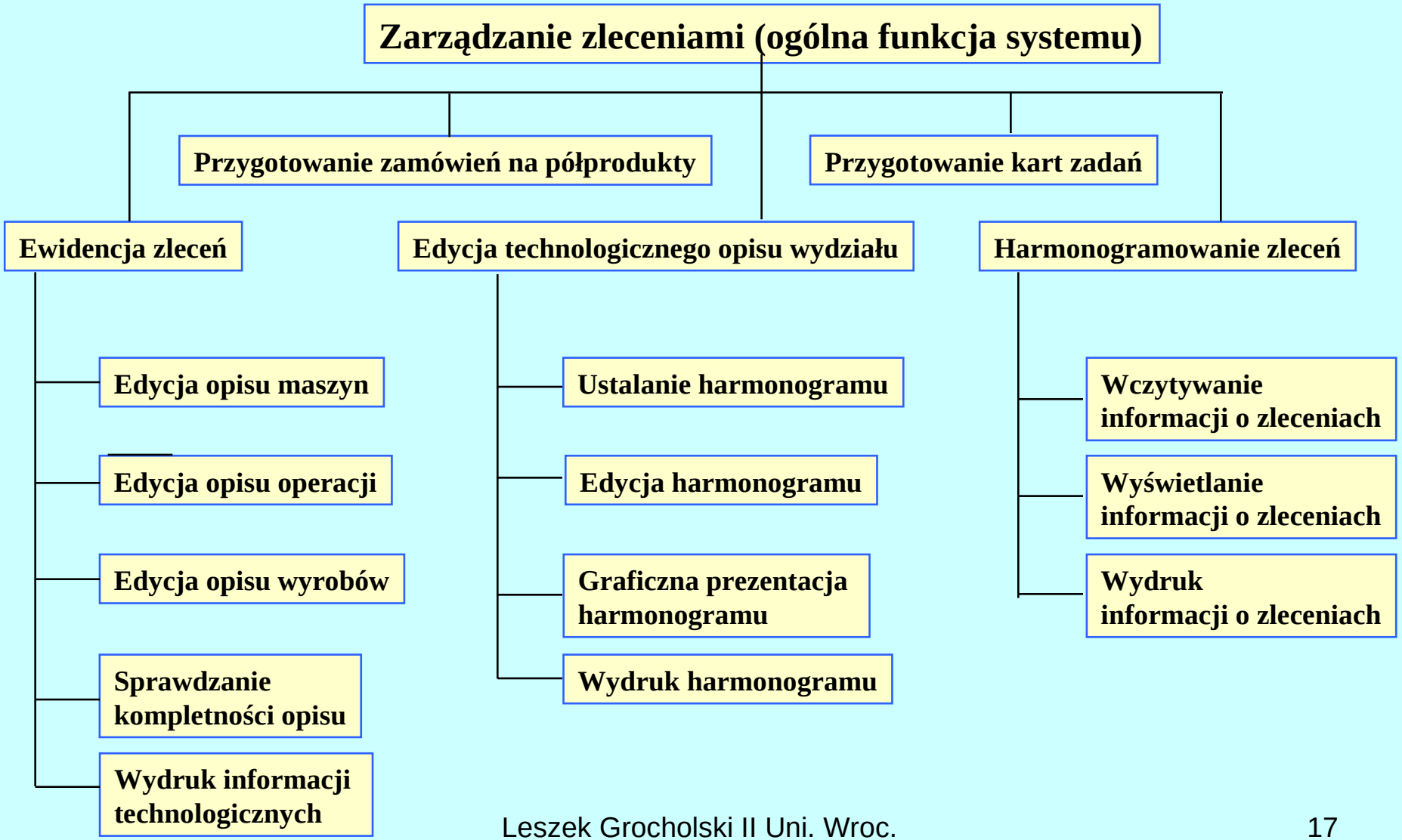
Opowiadanie (powieść?) użytkownika

Zlecenia dla wydziału przygotowywane są przez dział marketingu. Zlecenie oznacza konieczność wyprodukowania konkretnej ilości pewnego wyrobu przed upływem konkretnego terminu. Czasami może być określony najwcześniejszy pożądany termin realizacji. Dział marketingu wykorzystuje własny system informatyczny, w którym między innymi umieszczane są informacje o zleceniach, pożądane jest więc, aby system harmonogramowania zleceń automatycznie odczytywał te informacje.

Wyprodukowanie danego wyrobu (realizacja zlecenia) wymaga wykonania ciągu operacji. Pewne operacje mogą być wykonywane tylko na jednym urządzeniu; w innych przypadkach możliwe jest wykorzystanie kilku maszyn, które mogą różnić się efektywnością pracy. Po wykonaniu pewnych operacji może być konieczna przerwa, zanim rozpocznie się realizacja następnych; z drugiej strony taka przerwa może trwać przez ograniczony czas. Przestawienie maszyn na inne operacje może wymagać czasu. Co pewien czas (np. raz na miesiąc) ustalany jest harmonogram niezrealizowanych zleceń.

System powinien opracować harmonogramy w formie łatwej do wykorzystania przez kadrę kierowniczą wydziału oraz przygotowywać zamówienia do magazynu na półprodukty. Zlecenia wykonane są usuwane ze zbioru niezrealizowanych zleceń.

System harmonogramowania zleceń: funkcje



Diagramy przypadków użycia

Opis funkcji systemu z punktu widzenia jego użytkowników.
Opis wymagań poszczególnych użytkowników jest bardziej przejrzysty niż opis wszystkich wymagań wobec systemu.

Klasy użytkowników:

- projektant
- użytkownik - osoba przeglądająca mapę

Metoda modeluje **aktorów**, a nie konkretne **osoby**. Jedna osoba może pełnić rolę wielu aktorów; np. być jednocześnie projektantem i osobą przeglądającą mapę.
I odwrotnie, jeden aktor może odpowiadać wielu osobom, np. projektant.

Identyfikacja funkcji dla poszczególnych użytkowników.
Przeprowadzając wywiad z konkretna osobą należy koncentrować się na funkcjach istotnych z punktu widzenia roli (ról) odgrywanych przez tę osobę.

Metoda przypadków użycia nie jest sprzeczna z hierarchiczną dekompozycją funkcji.

Przykład: SIG

Opowiadanie (powieść?) użytkownika
SIG jest rodzajem mapy komputerowej, składającej się z tła (np. mapy fizycznej) oraz szeregu obiektów graficznych umieszczonych na tym tle.

Obiekt może być punktem (budynek, firma), linią (rzeka, kolej) lub obszarem (park, osiedle).

Każdy obiekt ma swoją nazwę i ewentualny opis, który może być wyświetlony na żądanie użytkownika. Obiekt można opisać szeregiem słów kluczowych.

Użytkownik ma możliwość wyświetlenia tylko tych obiektów, które opisano słowami kluczowymi.

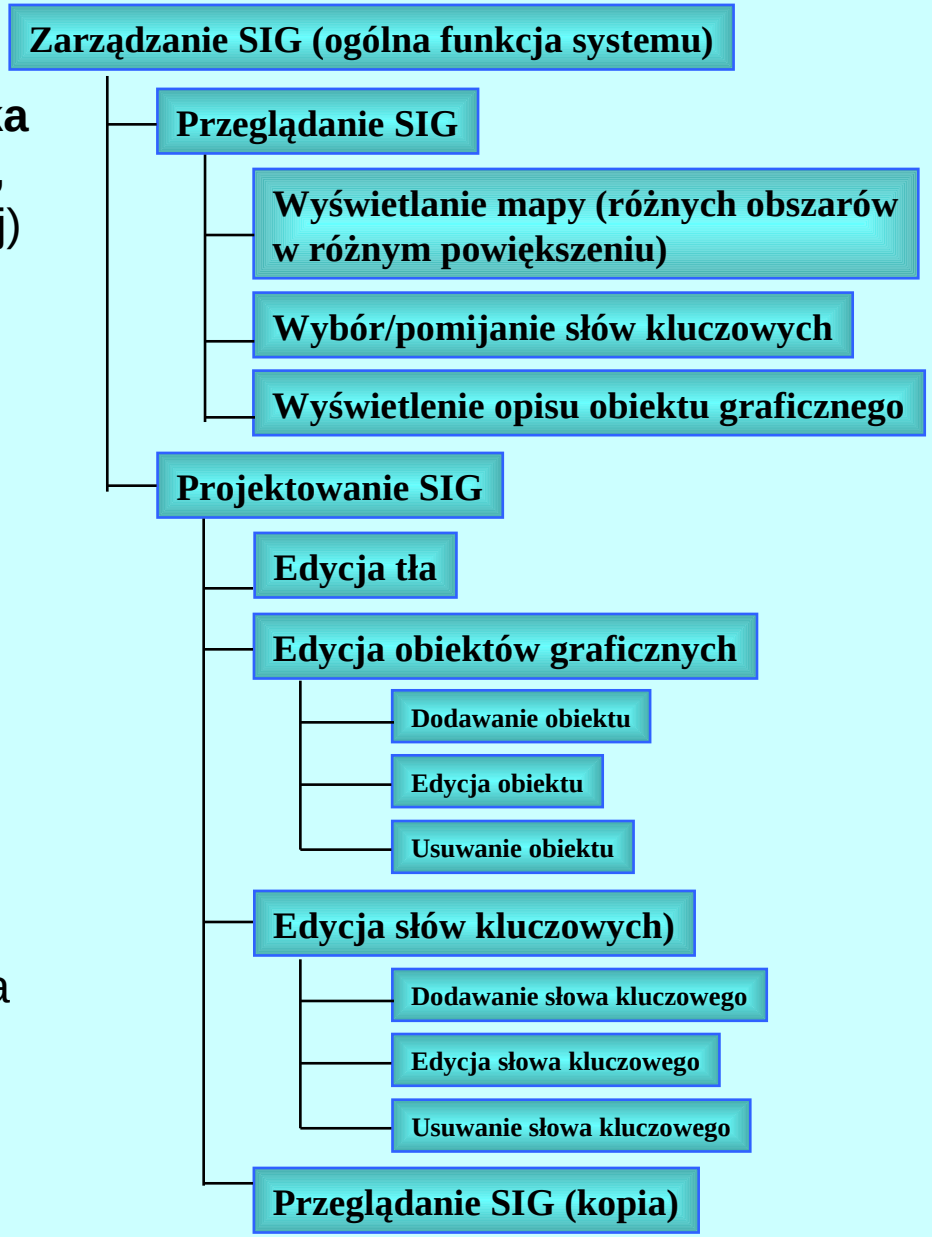
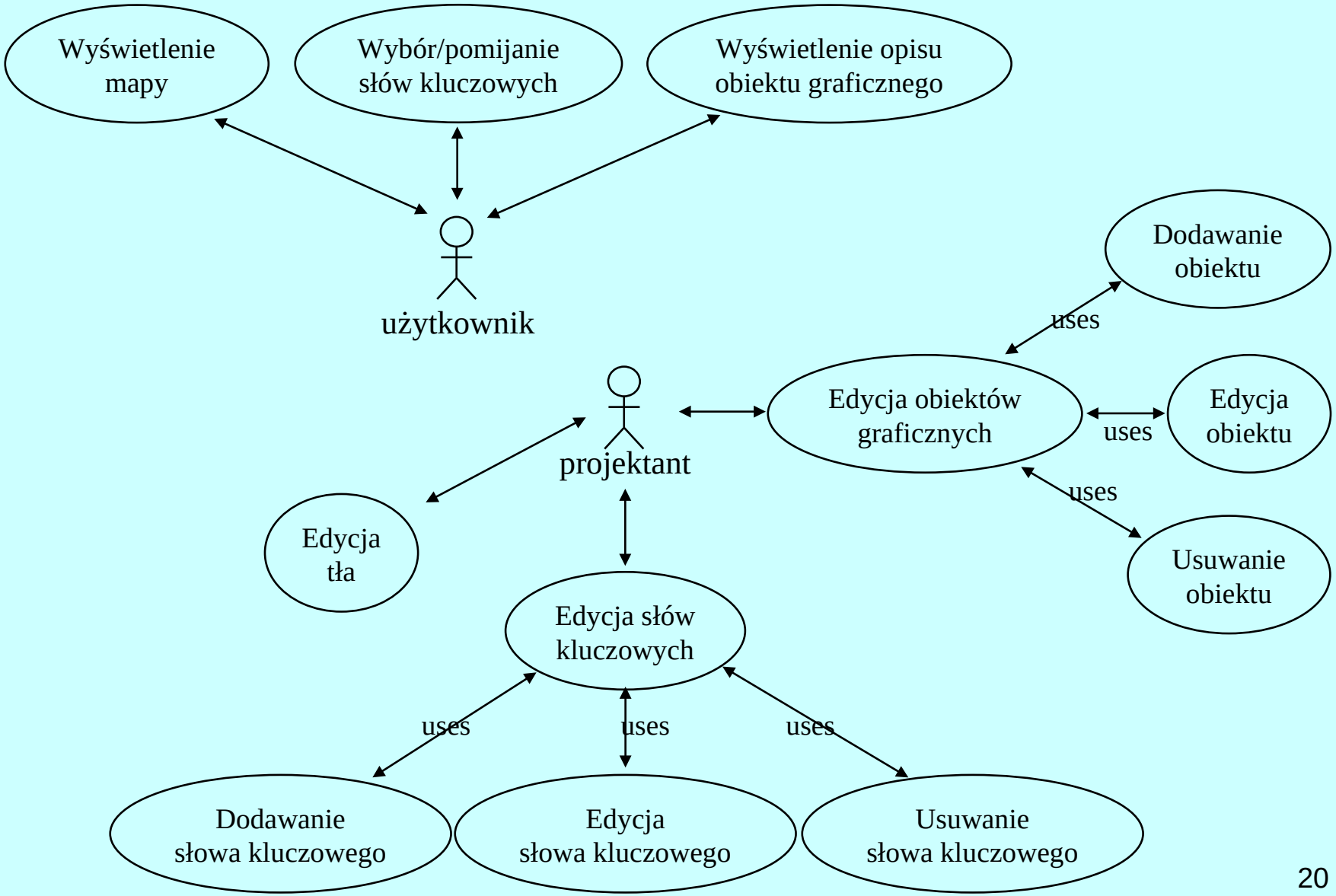


Diagram przypadków użycia dla SIG



Wymagania niefunkcjonalne

Określają „ w jaki sposób system ma realizować funkcje”
Inaczej - opisują ograniczenia, przy których system ma realizować swoje funkcje.

PODZIAŁ NA:

- Wymagania dotyczące produktu

Np. Musi istnieć możliwość operowania z systemem wyłącznie za pomocą klawiatury, np. bankomat.

- Wymagania dotyczące procesu

Np. Zarządzanie projektem ma się odbywać zgodnie z metodyką PRINCE 2.

- Wymagania zewnętrzne

Np. wykorzystanie standardów GPS, GLONASS (standard rosyjski nawigacji satelitarnej obejmujący zasięgiem całą kule ziemską).

Formularz zapisu wymagań nieunkcjonalnych

Nr	Data wprow.	Rozmówca	Wymaganie	Motywacja	Uwagi
1	99/04/14	A.Nowak J.Pietrzak	System powinien zwracać wynik zapytania po max 5-ciu sekundach przy 100 użytkownikach pracujących jednocześnie.	Inaczej system nie będzie konkurencyjny na rynku	Może być niestabilne
2	00/02/05	K.Lubicz	Każdy klient powinien mieć przypisany krótki numer identyfikacyjny	Inne identyfikatory niż PESEL np. numer telefonu są niestabilne, nie unikalne, lub za długie	
3	.....	.....	.....	....	.....

Weryfikowalność wymagań niefunkcjonalnych

Wymagania niefunkcjonalne powinny być **weryfikowalne -** czyli powinna istnieć możliwość sprawdzenia lub zmierzenia czy system je rzeczywiście spełnia.
Np. wymagania "*system ma być łatwy w obsłudze*", „*system ma być niezawodny*", „*system ma być dostatecznie szybki*", itd. nie są weryfikowalne.

Cecha	***Weryfikowalne miary***
Wydajność	Liczba transakcji obsłużonych w ciągu sekundy Czas odpowiedzi
Rozmiar	Liczba rekordów w bazie danych Wymagana pamięć dyskowa
Łatwość użytkowania	Czas niezbędny dla przeszkolenia użytkowników Rozmiar dokumentacji
Niezawodność	Prawdopodobieństwo błędu podczas realizacji transakcji Średni czas pomiędzy błędnymi wykonaniami Dostępność (procent czasu w którym system jest dostępny) Czas restartu po awarii systemu Prawdopodobieństwo zniszczenia danych w przypadku awarii
Przenaszalność	Procent kodu zależnego od platformy docelowej Liczba platform docelowych Koszt przeniesienia na nową platformę

Konstruowanie wymagań niefunkcjonalnych (1)

Możliwości systemu: Zestaw funkcji, które ma wykonywać system, uporządkowany hierarchicznie.

Objętość: Ilu użytkowników będzie pracować jednocześnie? Ile terminali ma być podłączone do systemu? Ile czujników będzie kontrolowanych jednocześnie? Ile danych będzie przechowywane?

Szybkość: Jak długo może trwać operacja lub sekwencja operacji? Liczba operacji na jednostkę czasu. Średni czas niezbędny dla jednej operacji.

Dokładność: Określenie stopnia precyzji pomiarów lub przetwarzania. Określenie wymaganej dokładności wyników. Zastąpienie wyników ilościowych jakościowymi lub odwrotnie.

Ograniczenia: ograniczenia na interfejsy, jakość, skalę czasową, sprzęt, oprogramowanie, skalowalność, itd.

Konstruowanie wymagań niefunkcjonalnych (2)

Interfejsy komunikacyjne: sieć, protokoły, wydajność sieci, poziom abstrakcji protokołów komunikacyjnych, itd.

Interfejsy sprzętowe: specyfikacja wszystkich elementów sprzętowych, które będą składały się na system, fizyczne ograniczenia (rozmiar, waga), wydajność (szybkość, RAM, dysk, inne pamięci), wymagania co do powierzchni lokalowych, wilgotności, temperatury i ciśnienia, itd.

Interfejsy oprogramowania: Określenie zgodności z innym oprogramowaniem, określenie systemów operacyjnych, języków programowania, kompilatorów, edytorów, systemów zarządzania bazą danych, itd.

Interakcja człowiek-maszyna: Wszystkie aspekty interfejsu użytkownika, rodzaj języka interakcji, rodzaj sprzętu (monitor, mysz, klawiatura), określenie formatów (układu raportów i ich zawartości), określenie komunikatów dla użytkowników (język, forma), pomocy, komunikatów o błędach, itd.

Konstruowanie wymagań niefunkcjonalnych (3)

Adaptowalność: Określenie w jaki sposób będzie organizowana reakcja na zmiany wymagań: dodanie nowej komendy, dodanie nowego okna interakcji, itd.

Bezpieczeństwo: założenia co do poufności, prywatności, integralności, odporności na hakerów, wirusy, wandalizm, sabotaż, itd.

Odporność na awarie: konsekwencje błędów w oprogramowaniu, przerwy w zasilaniu, kopie zabezpieczające, częstotliwości składowania, dziennika zmian, itd.

Standardy: Określenie dokumentów standardyzacyjnych, które mają zastosowanie do systemu: formaty plików, normy czcionek, polonizacja, standardy procesów i produktów, itd.

Zasoby: Określenie ograniczeń finansowych, ludzkich i materiałowych.

Skala czasowa: ograniczenia na czas wykonania systemu, czas szkolenia, wdrażania, itd.

Dokument wymagań

- Wymagania powinny być zebrane w dokumencie - opisie wymagań.
- Dokument ten powinien być podstawą szczegółowego kontraktu między klientem a producentem oprogramowania.
- Powinien także pozwalać na weryfikację stwierdzającą, czy wykonany system rzeczywiście spełnia postawione wymagania.
- Powinien to być dokument zrozumiały dla obydwu stron.

Uwaga: Zdarza się, że producenci nie są zainteresowaniu w precyzyjnym formułowaniu wymagań, które pozwoliłoby na rzeczywistą weryfikację powstałego systemu.
Tego rodzaju polityka lub niedbałość może prowadzić do konfliktów.

Zawartość dokumentu SW (1)

Informacje organizacyjne

Streszczenie (maksymalnie 200 słów)
Spis treści
Status dokumentu (autorzy, firmy, daty, podpisy, itd.)
Zmiany w stosunku do wersji poprzedniej

Zasadnicza zawartość dokumentu

1. Wstęp
- 1.1. Cel
- 1.2. Zakres
- 1.3. Definicje, akronimy i skróty
- 1.4. Referencje, odsyłacze do innych dokumentów
- 1.5. Krótki przegląd

2. Ogólny opis
- 2.1. Walory użytkowe i przydatność projektowanego systemu
- 2.2. Ogólne możliwości projektowanego systemu
- 2.3. Ogólne ograniczenia
- 2.4. Charakterystyka użytkowników
- 2.5. Środowisko operacyjne
- 2.6. Założenia i zależności

3. Specyficzne wymagania
- 3.1. Wymagania funkcjonalne (funkcje systemu)
- 3.2. Wymagania niefunkcjonalne (ograniczenia).

Dodatki

Norma
ANSI/IEEE Std 830-1993
„Recommended Practice for Software Requirements Specifications”

Zawartość dokumentu SW (2)

Kolejność i numeracja punktów w przedstawionym spisie treści powinna być zachowana. W przypadku gdy pewien punkt nie zawiera żadnej informacji należy wyraźnie to zasygnalizować przez umieszczenie napisu „Nie dotyczy”.

Dla każdego wymagania powinien być podany powód jego wprowadzenia: cele przedsięwzięcia, których osiągnięcie jest uwarunkowane danym wymaganiem.

Wszelki materiał nie mieszczący się w podanych punktach należy umieszczać w dodatkach.

Często spotykane dodatki

- wymagania sprzętowe
- wymagania dotyczące bazy danych
- model (architektura) systemu
- słownik terminów (użyte terminy, akronimy i skróty z wyjaśnieniem)
- indeks pomocny w wyszukiwaniu w dokumencie konkretnych informacji (dla dokumentów dłuższych niż 80 stron)

Jakość dokumentu wymagań

Styl

Jasność: jednoznaczne sformułowania, zrozumiały dla użytkowników i projektantów. Strukturalna organizacja dokumentu.
Spójność: brak konfliktów w wymaganiach.
Modyfikowalność: wszystkie wymagania są sformułowane w jasnych punktach, które mogą być wyizolowane z kontekstu i zastąpione przez inne.

Ewolucja

Możliwość dodawania nowych wymagań, możliwość ich modyfikacji.

Odpowiedzialność

Określenie kto jest odpowiedzialny za całość dokumentu wymagań.
Określenie kto lub co jest przyczyną sformułowania danego wymagania, istotne w przypadku modyfikacji, np. zmiany zakresu lub kontekstu systemu.

Medium

Dokument papierowy lub elektroniczny.
Staranne kontrolowanie wersji dokumentu.

Słownik

Opis wymagań może zawierać terminy niezrozumiałe dla jednej ze stron. Mogą to być **terminy informatyczne (niezrozumiałe dla klienta)** lub **terminy dotyczące dziedziny zastosowań (niezrozumiałe dla przedstawicieli producenta**).

Wszystkie specyficzne terminy powinny być umieszczone w słowniku, wraz z wyjaśnieniem. Słownik powinien precyzować terminy niejednoznaczne i określać ich znaczenie w kontekście tego dokument (być może nieco węższe).

Termin	**Objaśnienie**	**Synonimy (nie zalecane)**
konto	Nazwana ograniczona przestrzeń dyskowa, gdzie użytkownik może przechowywać swoje dane. Konta są powiązane z określonymi usługami, np. pocztą komputerową oraz z prawami dostępu.	katalog użytkownika
konto bankowe	Sekwencja cyfr oddzielona myślnikami, identyfikująca stan zasobów finansowych oraz operacji dla pojedynczego klienta banku.	konto
klient	Jednostka sprzętowa instalowana w biurach urzędu, poprzez którą następuje interakcja użytkownika końcowego z systemem.	stacja robocza
użytkownik	Osoba używająca systemu dla własnych celów biznesowych nie związanych z obsługą lub administracją systemu.	operator (klient)

Kluczowe czynniki sukcesu

- Zaangażowanie właściwych osób ze strony klienta

- Pełne rozpoznanie wymagań, wykrycie przypadków i dziedzin szczególnych i nietypowych. Błąd popełniany w tej fazie polega na koncentrowaniu się na sytuacjach typowych.

- Sprawdzenie kompletności i spójności wymagań. Przed przystąpieniem do dalszych prac, wymagania powinny być przejrzane pod kątem ich kompletności i spójności.

- Określenie wymagań niefunkcjonalnych w sposób umożliwiający ich weryfikację.

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę

INŻYNIERIA OPROGRAMOWANIA

wykład 5:
ANALIZA WYMAGAŃ

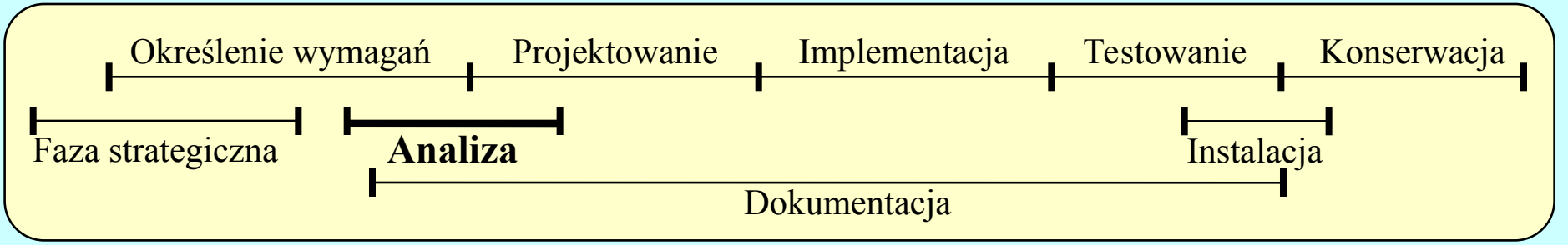
dr inż. Leszek Grocholski

Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

Plan wykładu

1. Model analityczny
2. Czynności w fazie analizy
3. Wymagania na oprogramowanie
4. Notacje w fazie analizie
5. Metodyki strukturalne i obiektowe
6. Metodyki obiektowe → UML
7. Proces tworzenia modelu obiektowego
8. Dokument wymagań na oprogramowanie
9. Kluczowe czynniki sukcesu i rezultaty fazy analizy

Analiza



Analiza jest fazą niezbędną do **przejścia** pomiędzy wymaganiami użytkownika (wynikającymi z jego procesów biznesowych) do wymagań na system informatyczny (wspomaganiem tych procesów).

Celem **początkowych** czynności analizy jest rozpoznaniem wszystkich tych czynników lub warunków w dziedzinie przedmiotowej, w otoczeniu projektu, w istniejących lub planowanych systemach, które mogą wpłynąć na **decyzje projektowe** i na **przebieg procesu projektowego**.

Na początku analizy chodzi nam wyłącznie o **rozpoznanie** wszelkich krytycznych aspektów w dziedzinie biznesowej, które mogą zaważyć na rezultacie lub przebiegu projektu. Nie interesujemy się zmianą procesów biznesowych wynikającą z wprowadzenia do nich systemu informatycznego.

Wynik analizy

Celem **końcowych** czynności analizy jest zamiana wymagań użytkownika oraz wszelkich krytycznych cech procesów biznesowych i otoczenia tych procesów na:

- wymagania wobec projektowanego systemu,
- wymagania wobec procesu jego wytwarzania.

Końcowe czynności analizy są więc początkowymi decyzjami projektowymi. Moment, w którym analiza zamienia się w projektowanie, jest umowny i często nieokreślony.

Istotne w analizie jest ustalenie **logicznego modelu systemu**, opisującego sposób realizacji przez system postawionych wymagań, bez szczegółów implementacyjnych.

Celem końcowego dokumentu analizy jest uzyskanie odpowiedzi na pytania: "jakie cechy ma posiadać przyszły system?"
oraz „jak ma być on tworzony?”.

Model analityczny

Każda dyscyplina techniczna korzysta z modeli.

Proszę podać przykłady.

Dlaczego stosujemy modele ?

Film Altkom Akademia
User Story – zwinne definiowanie wymagań:
https://www.youtube.com/watch?v=NRiaTM9t5oc&t=2175s

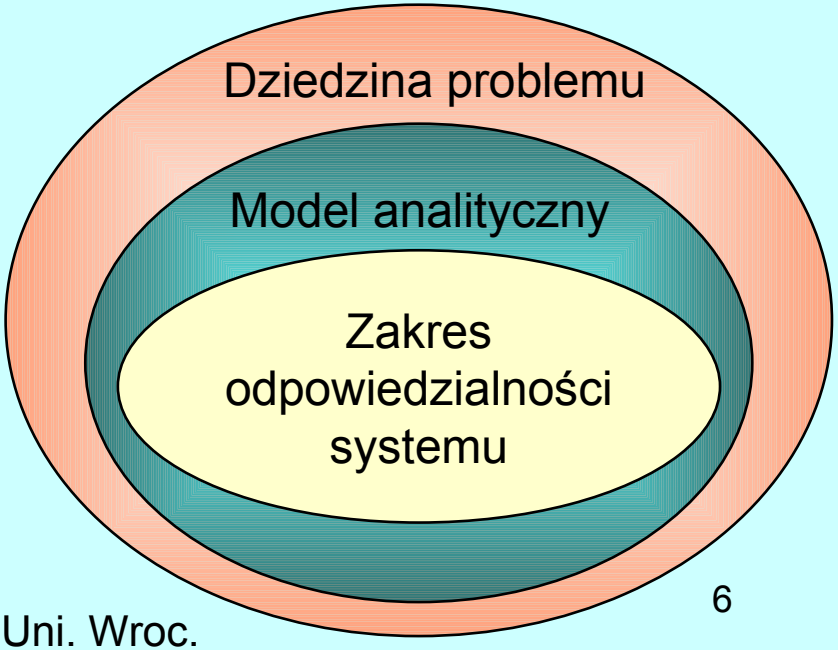
Model analityczny

Z reguły wykracza poza zakres odpowiedzialności systemu. Przyczyny:

Ujęcie w modelu pewnych elementów dziedziny problemu nie będących częścią systemu czyni model bardziej zrozumiałym. Przykładem jest ujęcie w modelu systemów zewnętrznych, z którymi system ma współpracować.

Na etapie modelowania może nie być jasne, które elementy modelu będą realizowane przez oprogramowanie, a które w sposób sprzętowy lub ręcznie.

Dostępne środki mogą nie pozwolić na realizację systemu w całości.
Celem analizy może być wykrycie tych fragmentów dziedziny problemu, których wspomaganie za pomocą oprogramowania będzie szczególnie przydatne.



Cechy modelu analitycznego (logicznego)

- Uproszczony opis systemu
- Hierarchiczna dekompozycja funkcji systemu
- Model logiczny jest opisany przy pomocy notacji zgodnej z pewną konwencją
- Jest on zbudowany przy użyciu dobrze rozpoznanych metod i narzędzi
- Jest on używany do wnioskowania o przyszłym oprogramowaniu

Model oprogramowania powinien być jego uproszonym opisem, opisującym wszystkie istotne cechy oprogramowania na wysokim poziomie abstrakcji.

Model ten jednakże nie zastępuje doświadczenia i wnikliwości projektantów, lecz pomaga projektantom w zastosowaniu tych walorów.

Logiczny model oprogramowania:

- pokazuje co system musi robić,
- jest zorganizowany hierarchicznie, wg poziomów abstrakcji,
- unika terminologii implementacyjnej,
- pozwala na wnioskowanie „od przyczyny do skutku” i odwrotnie.

Czynności w fazie analizy

Rozpoznanie, wyjaśnianie, modelowanie, specyfikowanie i dokumentowanie rzeczywistości lub problemu będącego przedmiotem projektu

Ustalenie kontekstu projektu

Ustalenie wymagań użytkowników

Ustalenie wymagań organizacyjnych

Inne ustalenia, np. dotyczące preferencji sprzętowych, preferencji w zakresie oprogramowania, ograniczeń finansowych, ograniczeń czasowych, itd.

Uwaga:
W zasadzie analiza nie powinna stawiać nacisku na zmianę rzeczywistości poprzez wprowadzenie tam nowych elementów np. w postaci systemu komputerowego. Jej **celem jest rozpoznanie** wszystkich tych aspektów rzeczywistości, które mogłyby mieć wpływ na postać, organizację lub wynik projektu.

Tematy i techniki analizy

- Budowa fizyczna systemu (składniki systemu)
- Analiza funkcji (historyjek użytkownika)
- Analiza przypadków użycia
- Identyfikacja i weryfikacja składowych logicznych systemu
- Identyfikacja i definiowanie algorytmów
- Modelowanie stanów i przejść między stanami
- Modelowanie procesów i przepływów danych
- Modelowanie przepływu sterowania
- Inne

Wiele tych technik jest omówione podczas wykładu nt. obiektowego projektowania oprogramowania.

Wymagania na oprogramowanie

W trakcie analizy **wymagania użytkownika** są przekształcane w **wymagania na oprogramowanie.**

Mogą one dotyczyć:

- Funkcji systemu
- Wydajności systemu
- Zewnętrznych interfejsów
- Wykonywanych operacji
- Wymaganych zasobów
- Sposobów weryfikacji
- Sposobów testowania
- Dokumentacji
- Ochrony (losowe zniszczenie)
- Przenośności
- Jakości
- Niezawodności
- Pielęgnacyjności
- Bezpieczeństwa (włamania)

Uwaga:
Wymagania powinny być zorganizowane hierarchicznie.
Wymagania niefunkcjonalne powinny być skojarzone z wymaganiami funkcjonalnymi (np. poprzez wzajemne odsyłacze).

Reguły modelu logicznego opartego na funkcjach

- Funkcje musza wynikać z historyjek użytkownika

- Funkcje muszą mieć pojedyncze, zdefiniowane cele.

- Funkcje powinny być zdefiniowane na tym samym poziomie (np. funkcja *Oblicz Sumę Kontrolną* jest niższego poziomu niż funkcja *Obsługa Protokołu Sieciowego*).

- Interfejsy do funkcji (wejście i wyjście) powinny być minimalne. Pozwala to na łatwiejsze oddzielenie poszczególnych funkcji.

- Przy dekompozycji funkcji należy trzymać się zasady „nie więcej niż 7 funkcji podrzędnych”.

- Opis funkcji nie powinien odwoływać się do pojęć i terminów implementacyjnych (takich jak plik, zapis, zadanie, moduł, stacja robocza).

- Określenie wskaźników wydajnościowe funkcji (szybkość, częstość, itd).

- Powinny być zidentyfikowane funkcje krytyczne (od których zależy istota systemu).

- Nazwy funkcji powinny odzwierciedlać ich cel i mówić ***co*** ma być zrobione, a nie ***jak*** ma być zrobione.

- Nazwy funkcji powinny mieć charakter deklaracyjny (np. *Walidacja Zlecenia Zewnętrznego),* a nie proceduralny (np. *Czynności Systemu po Otrzymaniu Zlecenia*).

Notacje w fazie analizy

Rodzaje notacji

- Język naturalny
- Notacje graficzne
- Specyfikacje - ustrukturalizowany zapis tekstowy i numeryczny

Szczególne znaczenie maja notacje graficzne. Inżynieria oprogramowania wzoruje się na innych dziedzinach techniki, takich jak elektronika i mechanika. **Zalety notacji graficznych potwierdzają badania psychologiczne.**

Funkcje notacji

- Narzędzie pracy analityka i projektanta, zapis i analiza pomysłów
- Współpraca z użytkownikiem
- Komunikacja z innymi członkami zespołu
- Podstawa implementacji oprogramowania
- Zapis dokumentacji technicznej

Notacje powinny być przejrzyste, proste, precyzyjne, łatwo zrozumiałe, umożliwiające modelowanie złożonych zależności.

Metodyki obiektowe w analizie

Metodyka wykorzystująca **pojęcia obiektowości** dla celów **modelowania pojęciowego** oraz analizy i projektowania systemów informatycznych.

Podstawowym składnikiem jest **diagram klas**, będący zwykle wariantem notacyjnym i pewnym rozszerzeniem diagramów encja-związek.

Diagram klas zawiera: **klasy**, w ramach klas specyfikacje **atrybutów** i **metod**, związki **generalizacji**, związki **asocjacji** i **agregacji**, **liczności** tych związków, różnorodne **ograniczenia** oraz inne oznaczenia.

Uzupełnieniem tego diagramu są inne: **diagramy dynamiczne** uwzględniające **stany** i przejścia pomiędzy tymi stanami, **diagramy interakcji** ustalające zależności pomiędzy wywołaniami metod, **diagramy funkcjonalne** (będące zwykle pewną mutacją diagramów przepływu danych), itd.

Koncepcja **przypadków użycia** (*use cases*) zakłada odwzorowanie struktury systemu z punktu widzenia jego użytkownika.

Różnice pomiędzy metodykami

Podejścia proponowane przez różnych autorów różnią się częściowo, nie muszą być jednak ze sobą sprzeczne. **Nie ma metodyk uniwersalnych.** Analitycy i projektanci wybierają kombinację technik i notacji, która jest w danym momencie najbardziej przydatna.

Poszczególne metodyki zawierają elementy rzadko wykorzystywane w praktyce.

Notacje proponowane przez różnych autorów nie są koniecznie nierozerwalne z samą metodyką.

Narzędzia CASE nie narzucają metodyki; raczej, określają one tylko notację. Twierdzenia, że jakieś narzędzie CASE “jest oparte” na konkretnej metodyce jest często wyłącznie hasłem reklamowym. Nawet najlepsze metodyki i narzędzia CASE nie są w stanie zapewnić jakości projektów.

Uwaga:
Kluczem do dobrego projektu jest zespół doświadczonych, zaangażowanych i kompetentnych osób, dla których metodyki, notacje i narzędzia CASE służą jako istotne wspomaganie.

Notacja a metodyka

Dowolny język, w tym notacje stosowane w metodykach, oprócz *składni* posiada dwa znacznie od niej ważniejsze aspekty: *semantykę* i *pragmatykę*.

Składnia określa, jak wolno łączyć (składać) ze sobą przyjęte oznaczenia.

Semantyka określa, co należy rozumieć pod przyjętymi oznaczeniami.

Pragmatyka określa, w jaki sposób i do czego należy używać przyjętych oznaczeń.

Pragmatyka określa, jak do konkretnej sytuacji dopasować pewien wzorzec notacyjny. Pragmatyka wyznacza więc *procesy* prowadzące do wytworzenia zapisów wyników analizy i projektowania, które są zgodne z intencją autorów tej notacji. Jakakolwiek notacja nie ma większego sensu bez wiedzy o tym, w jaki sposób może być ona użyta w ramach pewnego procesu analizy i projektowania.

W metodykach pragmatyka stosowanej notacji (czyli jak i do czego jej użyć) jest sprawą podstawową. Jest ona zazwyczaj trudna do objaśnienia: nie ma innego sposobu oprócz pokazania sposobów użycia na przykładach przypominających realne sytuacje. **Realne sytuacje są zazwyczaj skomplikowane, co powoduje wrażenie, że przykłady zamieszczane w podręcznikach wydają się banalne.**

UML - przykład notacji

UML (Unified Modeling Language) powstał jako synteza trzech metodyk/notacji:

- OMT (Rumbaugh): metodyka ta była dobra do modelowania dziedziny przedmiotowej. Nie przykrywała jednak dostatecznie dokładnie zarówno aspektu użytkowników systemu jak i aspektu implementacji/konstrukcji.

- OOSE (Jacobson): metodyka ta dobrze podchodziła do kwestii modelowania użytkowników i cyklu życiowego systemu. Nie przykrywała jednak dokładnie modelowania dziedziny przedmiotowej jak i aspektu implementacji/konstrukcji.

- OOAD (Booch): metodyka dobrze podchodziła do kwestii projektowania, konstrukcji i związków ze środowiskiem implementacji. Nie przykrywała jednak dostatecznie dobrze fazy analizy i rozpoznania wymagań użytkowników.

Istniało wiele aspektów systemów, które nie były właściwie przykryte przez żadne z wyżej wymienionych podejść, np. włączenie prototypowania w cykl życiowy, rozproszenie i komponenty, przystosowanie notacji do preferencji projektantów, i inne. Celem UML jest przykrycie również tych aspektów.

Diagramy definiowane w UML

Autorzy UML wychodzą z założenia, że żadna pojedyncza perspektywa spojrzenia na projektowany system nie jest wystarczająca. Stąd wiele środków:

- **Diagramy przypadków użycia (***use case***)**
- **Diagramy klas,** w tym diagramy pakietów
- **Diagramy zachowania się (***behavior***)**
  - Diagramy stanów
  - Diagramy aktywności
  - Diagramy sekwencji
  - Diagramy współpracy (*collaboration*)

- Diagramy implementacyjne

  - Diagramy komponentów
  - Diagramy wdrożeniowe (*deployment*)

Diagramy te zapewniają mnogość perspektyw systemu podczas analizy i rozwoju.

Diagramy definiowane w UML

Film :
Modelowanie oprogramowania z użyciem UML
Altkom Akademia:

https://www.youtube.com/watch?v=G9zTK90ZNXQ

Film:
Diagramy aktywności czyli co łączy procesy i scenariusze
Altkom Akademia:

https://www.youtube.com/watch?v=zpTQN__UPUk

Proces tworzenia modelu obiektowego

Zadania:

- Identyfikacja klas i obiektów
- Identyfikacja związków pomiędzy klasami
- Identyfikacja i definiowanie pól (atrybutów)
- Identyfikacja i definiowanie metod i komunikatów

Czynności te są wykonywane iteracyjnie. Kolejność ich wykonywania nie jest ustalona i zależy zarówno od stylu pracy, jak i od konkretnego problemu.

Inny schemat realizacji procesu budowy modelu obiektowego polega na rozpoznaniu funkcji, które system ma wykonywać. Dopiero w późniejszej fazie następuje identyfikacja klas, związków, atrybutów i metod. Rozpoznaniu funkcji systemu służą modele funkcjonalne (diagramy przepływu danych) oraz model przypadków użycia.

Identyfikacja klas i obiektów

Typowe klasy:

- przedmioty namacalne (samochód, czujnik)
- role pełnione przez osoby (pracownik, wykładowca, student)
- zdarzenia, o których system przechowuje informacje (lądowanie samolotu, wysłanie zamówienia, dostawa),
- interakcje pomiędzy osobami i/lub systemami o których system przechowuje informacje (pożyczka, spotkanie, sesja),
- lokalizacje - miejsce przeznaczone dla ludzi lub przedmiotów
- grupy przedmiotów namacalnych (kartoteka, samochód jako zestaw części)
- organizacje (firma, wydział, związek)
- wydarzenia (posiedzenie sejmu, demonstracja uliczna)
- koncepcje i pojęcia (zadanie, miara jakości)
- dokumenty (faktura, prawo jazdy)
- klasy będące interfejsami dla systemów zewnętrznych
- klasy będące interfejsami dla urządzeń sprzętowych

Obiekty, zbiory obiektów i metadane

W wielu przypadkach przy definicji klasy należy dokładnie ustalić, z jakiego rodzaju abstrakcją obiektu mamy do czynienia.

Należy zwrócić uwagę na następujące aspekty:

- czy mamy do czynienia z konkretnym obiektem w danej chwili czasowej?
- czy mamy do czynienia z konkretnym obiektem w pewnym odcinku czasu?
- czy mamy do czynienia z opisem tego obiektu (dokument, metadane)?
- czy mamy do czynienia z pewnym zbiorem konkretnych obiektów?

Np. klasa „samochód”. Może chodzić o:

- egzemplarz samochodu wyprodukowany przez pewną fabrykę
- model samochodu produkowany przez fabrykę
- pozycję w katalogu samochodów opisujący własności modelu
- historię stanów pewnego konkretnego samochodu

Podobnie z klasą „gazeta”. Może chodzić o:

- konkretny egzemplarz gazety kupiony przez czytelnika
- konkretne wydanie gazety (niezależne od ilości egzemplarzy)
- tytuł i wydawnictwo, niezależne od egzemplarzy i wydań
- partię egzemplarzy danej gazety dostarczaną codziennie do kiosku

Wymagania na oprogramowanie

Informacje organizacyjne

Streszczenie (maksymalnie 200 słów)
Spis treści
Status dokumentu (autorzy, firmy, daty, podpisy, itd.)
Zmiany w stosunku do wersji poprzedniej

Zasadnicza zawartość dokumentu

1. Wstęp
- 1.1. Cel
- 1.2. Zakres
- 1.3. Definicje, akronimy i skróty
- 1.4. Referencje, odsyłacze do innych dokumentów
- 1.5. Krótki przegląd

2. Ogólny opis
- 2.1. Relacje do bieżących projektów
- 2.2. Relacje do wcześniejszych i następnych projektów
- 2.3. Funkcje i cele
- 2.4. Ustalenia dotyczące środowiska
- 2.5. Relacje do innych systemów
- 2.6. Ogólne ograniczenia
- 2.7. Opis modelu

..... cd. na następnym slajdzie

Norma
ANSI/IEEE Std 830-1993 „Recommended Practice for Software Requirements Specifications”

Wymagania na oprogramowanie (2)

..... (*poprzedni slajd*)

3. Specyficzne wymagania (ten rozdział może być podzielony na wiele rozdziałów zgodnie z podziałem funkcji)

3.1. Wymagania dotyczące funkcji systemu
3.2. Wymagania dotyczące wydajności systemu
3.3. Wymagania dotyczące zewnętrznych interfejsów
3.4. Wymagania dotyczące wykonywanych operacji
3.5. Wymagania dotyczące wymaganych zasobów
3.6. Wymagania dotyczące sposobów weryfikacji
3.7. Wymagania dotyczące sposobów testowania
3.8. Wymagania dotyczące dokumentacji
3.9. Wymagania dotyczące ochrony dokumentacji i nośników
3.10. Wymagania dotyczące przenośności
3.11. **Wymagania dotyczące jakości**
3.12. Wymagania dotyczące niezawodności
3.13. Wymagania dotyczące pielęgnacyjności
3.14. Wymagania dotyczące bezpieczeństwa

Dodatki (to, co nie zmieściło się w powyższych punktach)

Analiza – zapewnienie jakości

PLAN ZAPEWNIENIA JAKOŚCI dotyczy WYMAGAŃ odnośnie:

- Sposobów weryfikacji
- Sposobów testowania
- Norm, standardów
- Niezawodności
- Pielęgnacyjności
- Bezpieczeństwa
- Standardów wewnętrznych

PLAN ZAPEWNIENIA JAKOŚCI powinien zakładać MONITOROWANIE następujących aktywności:

- Zarządzanie
- Dokumentowanie (jakość dokumentacji)
- Standardy, praktyki, konwencje i metryki
- Przeglądy i audyty
- Testowanie
- Raporty problemów i akcje korekcyjne
- Narzędzia, techniki i metody
- Kontrolowanie wytwarzanego kodu i mediów
- Kontrolowanie dostaw
- Pielęgnowanie i utrzymywanie kolekcji zapisów
- Szkolenie
- Zarządzanie ryzykiem

Analiza - kluczowe czynniki sukcesu

- Zaangażowanie właściwych osób ze strony klienta

- Kompleksowe i całościowe podejście do problemu, nie koncentrowanie się na partykularnych jego aspektach

- Nie unikanie trudnych problemów (bezpieczeństwo, skalowalność, szacunki objętości i przyrostu danych, itd.)

- Zachowanie przyjętych standardów, np. w zakresie notacji

- Sprawdzenie poprawności i wzajemnej spójności modelu

- Przejrzystość, logiczny układ i spójność dokumentacji

Analiza - podstawowe rezultaty

- Poprawiony dokument opisujący wymagania

- Słownik danych zawierający specyfikację modelu

- Dokument opisujący stworzony model, zawierający:

- diagram klas
- diagram przypadków użycia
- diagramy sekwencji komunikatów (dla wybranych sytuacji)
- diagramy stanów (dla wybranych sytuacji)
- raport zawierający definicje i opisy klas, atrybutów, związków, metod, itd.

- Harmonogram fazy projektowania

- Wstępne przypisanie ludzi i zespołów do zadań

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę

INŻYNIERIA OPROGRAMOWANIA

wykład 6:
PROJEKTOWANIE

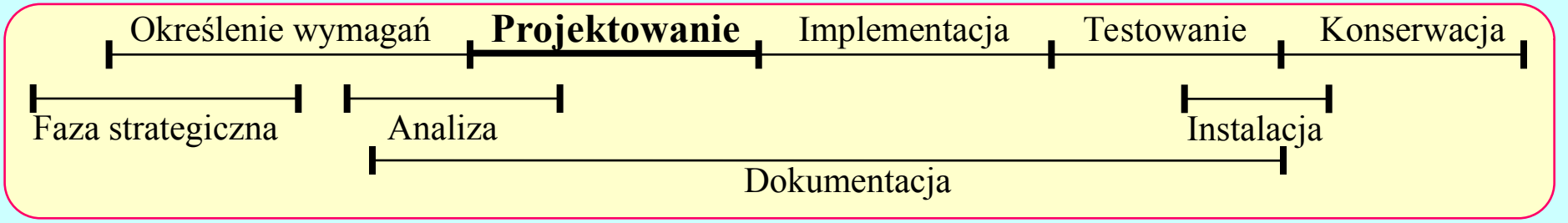
dr inż. Leszek Grocholski

Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

Plan wykładu

1. Zadania wykonywane w fazie projektowania
2. Projektowanie składowych nie/związanych z dziedziną problemu
3. Projektowanie Interfejsu z użytkownikiem
4. Projektowanie składowej zarządzania danymi
5. Optymalizacja projektu
6. Dostosowanie do ograniczeń i możliwości środowiska implementacji
7. Określenie fizycznej struktury systemu
8. Graficzny opis sprzętowej konfiguracji systemu
9. Poprawność i jakość projektu
10. Wymagania niefunkcjonalne dla fazy projektowania
11. Podstawowe rezultaty fazy projektowania

Projektowanie



Celem projektowania jest opracowanie szczegółowego opisu implementacji systemu. Tak szczegółowego aby na jego podstawie można było wykonać system.

W odróżnieniu od analizy, w projektowaniu dużą role odgrywa środowisko implementacji. Projektanci muszą więc posiadać dobrą znajomość języków, bibliotek i narzędzi stosowanych w trakcie implementacji.

Projektowanie chrakteryzuje naturalne dążenie do tego, aby struktura projektu zachowała ogólną strukturę modelu stworzonego w poprzednich fazach (analizie). Niewielkie zmiany w dziedzinie problemu powinny implikować niewielkie zmiany w projekcie.

Zadania wykonywane w fazie projektowania

1. Uszczegółowienie wyników analizy. Projekt musi być wystarczająco szczegółowy aby mógł być podstawą implementacji.
   Stopień szczegółowości zależy od poziomu zaawansowania programistów.

2. Projektowanie składowych dot. wymagań funkcjonalnych (dziedzina)

3. Projektowanie składowych systemów nie związanych z dziedziną problemu

4. Projektowanie rozwiązań zapewniających spełnienie innych wymagań niefunkcjonalnych

3. Optymalizacja systemu

4. Dostosowanie do ograniczeń i możliwości środowiska implementacji

5. Określenie logicznej fizycznej struktury systemu

6. Weryfikacja, walidacja zgodności z wymaganiami i założeniami

Projektowanie składowych systemu nie związanych z dziedziną problemu

Składowa dot. dziedziny problemu – to co system robi z perspektywy klienta. Projekt skonstruowany przez uszczegółowienie modelu opisuje składowe programu odpowiedzialne za realizację podstawowych zadań systemu.

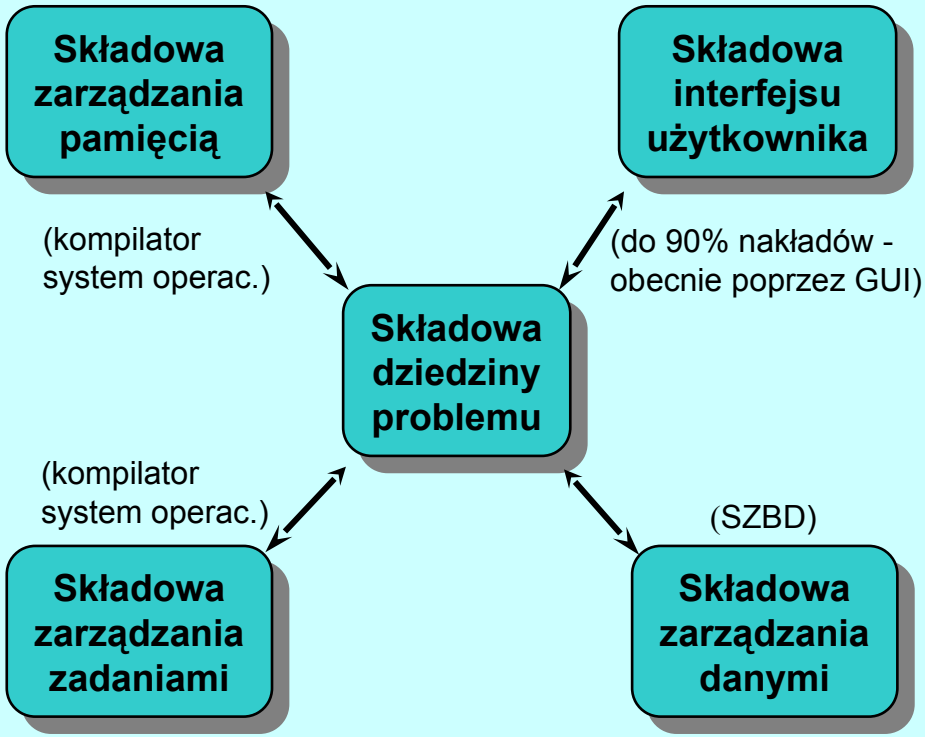
Gotowe oprogramowanie musi się jednak składać z dodatkowych składowych:

- składowej interfejsu użytkownika

- składowej zarządzania danymi (przechowywanie trwałych danych)

- składowej zarządzania pamięcią operacyjną

- składowej zarządzania zadaniami (podział czasu procesora)



RAD - Rapid Application Development

Szybkie rozwijanie? aplikacji

Terminem tym określa się narzędzia i techniki programowania umożliwiające szybką budowę prototypów lub gotowych aplikacji, z reguły oparte o programowanie wizyjne. Termin RAD występuje niekiedy jako synonim języków/środowisk czwartej generacji (4GL).

Przykładami narzędzi RAD są: Microsoft Visual Studio, PowerBuilder Desktop oraz przeróżne framework'i.

Łatwa realizacja pewnych funkcji systemu poprzez tworzenie bezpośredniego połączenia pomiędzy składowymi interfejsu użytkownika (dialogami, raportami) z elementami zarządzania danymi w bazie danych (przeważnie relacyjnej).

Składowa dziedziny problemu w najmniejszym stopniu poddaje się automatyzacji. Niekiedy inne ograniczenia lub nietypowość wykluczają możliwość zastosowania narzędzi RAD.

Projektowanie interfejsu użytkownika

W ostatnich latach nastąpił gwałtowny rozwój narzędzi graficznych służących do tego celu różnych frameworków.

- Interaktywne projektowanie dialogów, okien, menu, map bitowych, ikon oraz pasków narzędziowych z wykorzystaniem bogatego zestawu gotowych elementów.

- Definiowanie reakcji systemu na zajście pewnych zdarzeń, tj. akcji podejmowanych przez użytkownika (np. wybór z menu).

- Symulacja pracy interfejsu.

- Generowanie kodu, często z możliwością wyboru jednego z wielu środowisk docelowych.

Organizacja interakcji z użytkownikiem

Realizacja komunikacji z użytkownikiem:

Za pomocą linii komend

- dla niewielkich systemów,
- dla prototypów,
- dla zaawansowanych użytkowników.

Przykład: komendy „unixowe”, systemów wbudowanych.
Uwaga: Często szybszy od niż interfejs pełnoekranowy (okienkowy).

W pełnoekranowym środowisku okienkowym

Tworzenie ma sens dla dużych systemów.
Wygodny dla początkujących i średnio zaawansowanych użytkowników

Wprowadzanie i wyprowadzanie danych

Wprowadzanie (IN) przez użytkownika:

- Podawanie parametrów poleceń w przypadku systemów z linią komend
- Wprowadzanie danych w odpowiedzi na zaproszenie systemu
- Wprowadzanie danych w dialogach – formatki dialogowe

Wyprowadzanie (OUT) przez system:

- Wyświetlanie informacji w formatkach dialogowych
- Wyświetlanie i/lub wydruki raportów (analizy, planowanie…)
- Graficzna prezentacja danych (bardzo użyteczne)

Prototyp interfejsu użytkownika może powstać już w fazie określenia wymagań.
Pomaga uzyskać odpowiedz na pytanie: czy tak właśnie chce pracować użytkownik.
Systemy zarządzania interfejsem użytkownika pozwalają na wygodną budowę prototypów oraz wykorzystanie prototypu w końcowej implementacji.

Zasady projektowania interfejsu użytkownika (1)

Wytyczne projektowania interfejsu użytkownika – np. 10 heurystyk Nielsena

1. Spójność. Wygląd oraz obsługa interfejsu powinna być podobna w momencie korzystania z różnych funkcji. Poszczególne programy tworzące system powinny mieć zbliżony interfejs, podobnie powinna wyglądać praca z rozmaitymi dialogami, podobnie powinny być interpretowane operacje wykonywane przy pomocy myszy. Proste reguły:
- Umieszczanie etykiet zawsze nad lub obok pól edycyjnych.
- Umieszczanie typowych pól OK i Anuluj zawsze od dołu lub od prawej.
- Spójne tłumaczenie nazw angielskich, spójne oznaczenia pól.

2. Skróty dla doświadczonych użytkowników. Możliwość zastąpienia komend w paskach narzędziowych przez kombinację klawiszy.

3. Potwierdzenie przyjęcia zlecenia użytkownika. Realizacja niektórych zleceń może trwać długo. W takich sytuacjach należy potwierdzić przyjęcie zlecenie, aby użytkownik nie był zdezorientowany odnośnie tego co się dzieje. Dla długich akcji - wykonywanie sporadycznych akcji na ekranie (np. wyświetlanie sekund trwania, sekund do przewidywanego zakończenia, „termometru”, itd.).

Zasady projektowania interfejsu użytkownika (2)

4. Prosta obsługa błędów. Jeżeli użytkownik wprowadzi błędne dane, to po sygnale błędu system powinien automatycznie przejść do kontynuowania przez niego pracy z poprzednimi poprawnymi wartościami.

5. Odwoływanie akcji (*undo*). W najprostszym przypadku jest to możliwość cofnięcia ostatnio wykonanej operacji. Jeszcze lepiej jeżeli system pozwala cofnąć się dowolnie daleko w tył.

6. Wrażenie kontroli nad systemem. Użytkownicy nie lubią, kiedy system sam robi coś, czego użytkownik nie zainicjował, lub kiedy akcja systemu nie daje się przerwać. System nie powinien inicjować długich akcji (np. składowania) nie informując użytkownika co w tej chwili robi oraz powinien szybko reagować na sygnały przerwania akcji (Esc, Ctrl+C, Break,...)

Zasady projektowania interfejsu użytkownika (3)

7. Nieobciążanie pamięci krótkotrwałej użytkownika. Użytkownik może zapomnieć o tym po co i z jakimi danymi uruchomił dialog. System powinien wyświetlać stale te informacje, które są niezbędne do tego, aby użytkownik wiedział, co aktualnie się dzieje i w którym miejscu interfejsu się znajduje.

8. Grupowanie powiązanych operacji. Jeżeli zadanie nie da się zamknąć w prostym dialogu lub oknie, wówczas trzeba je rozbić na szereg powiązanych dialogów. Użytkownik powinien być prowadzony przez ten szereg, z możliwością łatwego powrotu do wcześniejszych akcji.

Reguła Millera 7 +/- 2:
Człowiek może się jednocześnie skupić na 5 - 9 elementach.
Dotyczy to liczby opcji menu, podmenu, pól w dialogu, itd. Ograniczenie to można przełamać poprzez grupowanie w wyraźnie wydzielone grupy zestawów semantycznie powiązanych ze sobą elementów.

Projektowanie składowej zarządzania danymi

Trwałe dane mogą być przechowane w:
- pliku,
- w bazie danych (relacyjnej, obiektowej, lub innej).

Poszczególne elementy danych - zestawy obiektów lub krotek - mogą być przechowywane w następującej postaci:
- w jednej relacji lub pliku,
- w odrębnym pliku dla każdego rodzaju obiektów lub krotek.

Sprowadzenie danych do pamięci operacyjnej oraz zapisanie do trwałej pamięci może być:
-na bieżąco, kiedy program zażąda dostępu i kiedy następuje zapełnienie bufora
- na zlecenie użytkownika

Zalety baz danych

- Wysoka efektywność i stabilność
- Bezpieczeństwo i prywatność danych, spójność i integralność przetwarzania
- Automatyczne sprawdzanie warunków integralności danych
- Wielodostęp, przetwarzanie transakcji

- Rozszerzalność (zarówno dodawanie danych jak i dodawanie ich rodzajów)
- Możliwość geograficznego rozproszenia danych
- Możliwość kaskadowego usuwania powiązanych danych
- Dostęp poprzez języki zapytań (SQL)

Spójność: zgodność danych z rzeczywistością lub z oczekiwaniami użytkownika.

Integralność: poprawność danych w sensie ich organizacji i budowy.

Wady relacyjnych baz danych

-Konieczność przeprowadzenie nietrywialnych odwzorowań przy przejściu z modelu pojęciowego (np. obiektowego w UML) na strukturę relacyjną.
→ Hibernate

- Ustalony format krótki powodujący trudności przy polach zmiennej długości.

- Trudności (niesystematyczność) reprezentacji dużych wartości (grafiki, plików tekstowych, itd.)

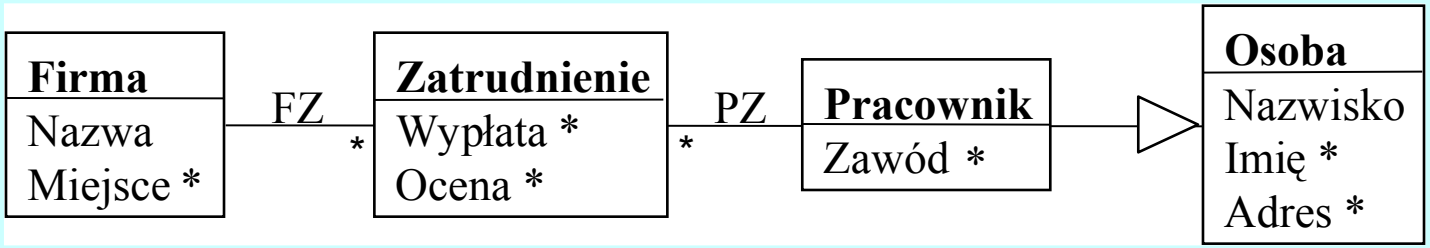
- W niektórych sytuacjach - duże narzuty na czas przetwarzania

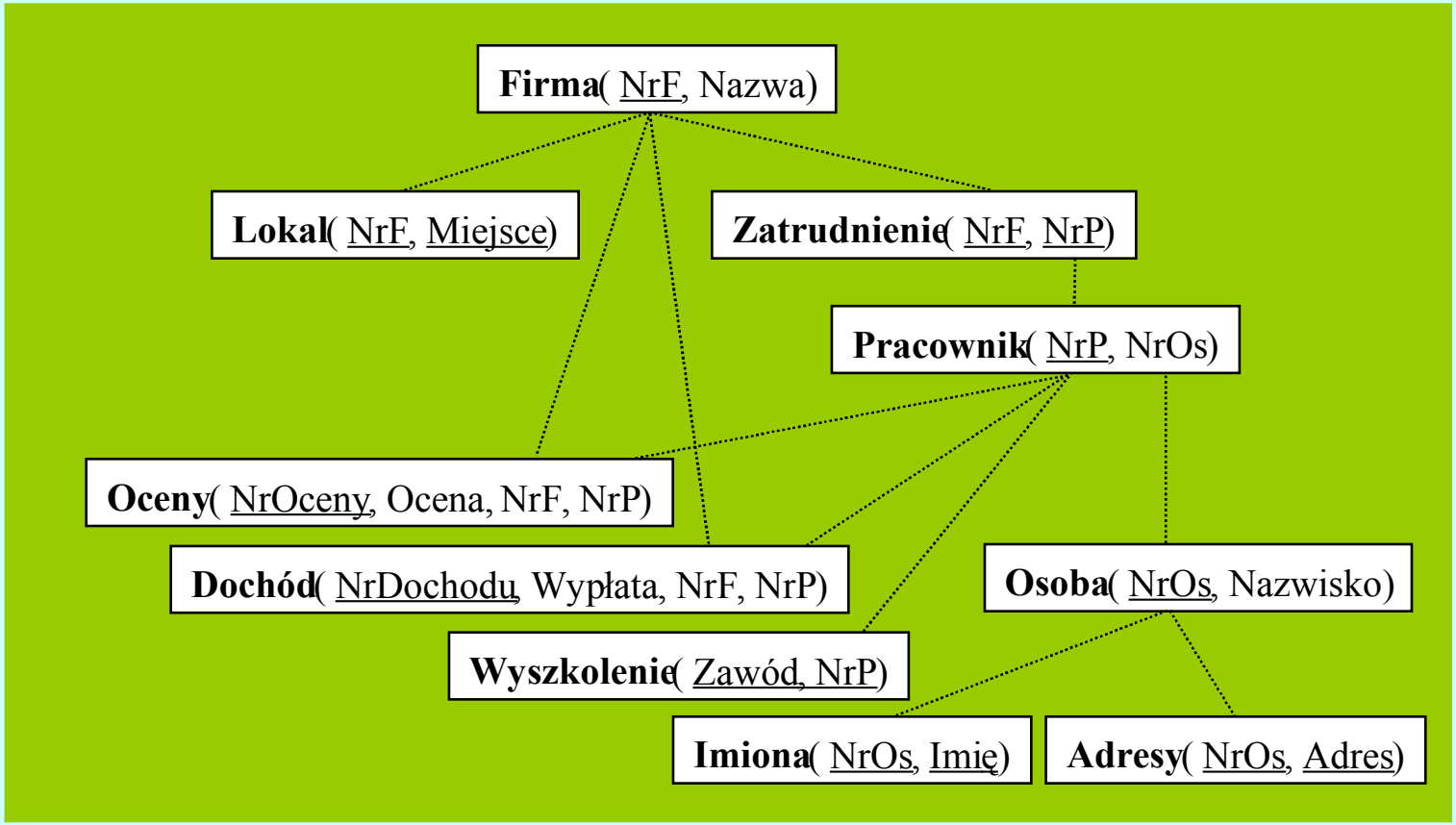
- Niedopasowanie interfejsu dostępu do bazy danych (SQL) do języka programowania (np. C), określana jako “niezgodność impedancji”.

- Brak możliwości rozszerzalności typów (zagnieżdżania danych)

- Brak systematycznego podejścia do informacji proceduralnej (metod)

Niezgodność modelu obiektowego i relacyjnego





Optymalizacja projektu (1)

Bezpośrednia implementacja projektu może prowadzić do systemu o zbyt niskiej efektywności. Najczęstsze problemy:

- Wykonanie pewnych funkcji jest zbyt wolne.
- Struktury danych mogą wymagać zbyt dużej pamięci operacyjnej i masowej.

Optymalizacja może być dokonana:

- Na poziomie projektu
- Na poziomie implementacji

Sposoby stosowane na etapie implementacji:

- Stosowanie zmiennych statycznych zamiast dynamicznych (lokalnych)
- Umieszczanie zagnieżdżonego kodu zamiast wywoływania procedur.
- Dobór typów o minimalnej, niezbędnej wartości.
- Optymalizacja zapytań w SQL.

Uwaga:
Wielu specjalistów jest przeciwna sztuczkom optymalizacyjnym: zyski są bardzo małe (o ile w ogóle są) w stosunku do zwiększenia nieczytelności kodu.

Optymalizacja projektu (2)

o może przynieść zasadnicze zyski optymalizacyjne?

- Zmiana algorytmu przetwarzania. Np. zmiana algorytmu sortującego poprzez wprowadzenie pośredniego pliku zawierającego tylko klucze i wskaźniki do sortowanych obiektów może przynieść nawet 100-krotny zysk.

- Wyłowienie "wąskich gardeł" w przetwarzaniu i optymalizacja tych wąskich gardeł poprzez starannie rozpracowane procedury. Znane jest twierdzenie, że 20% kodu jest wykonywane przez 80% czasu.

- **Zaprogramowanie "wąskich gardeł" w języku niższego poziomu**, np. w C dla programów w 4GL.

- Denormalizacja relacyjnej bazy danych, łączenie dwóch lub więcej tablic w jedną.

- Stosowanie indeksów, tablic wskaźników i innych struktur pomocniczych.

- Analiza mechanizmów buforowania danych w pamięci operacyjnej i ewentualna zmiana tego mechanizmu (np. zmniejszenie liczby poziomów)

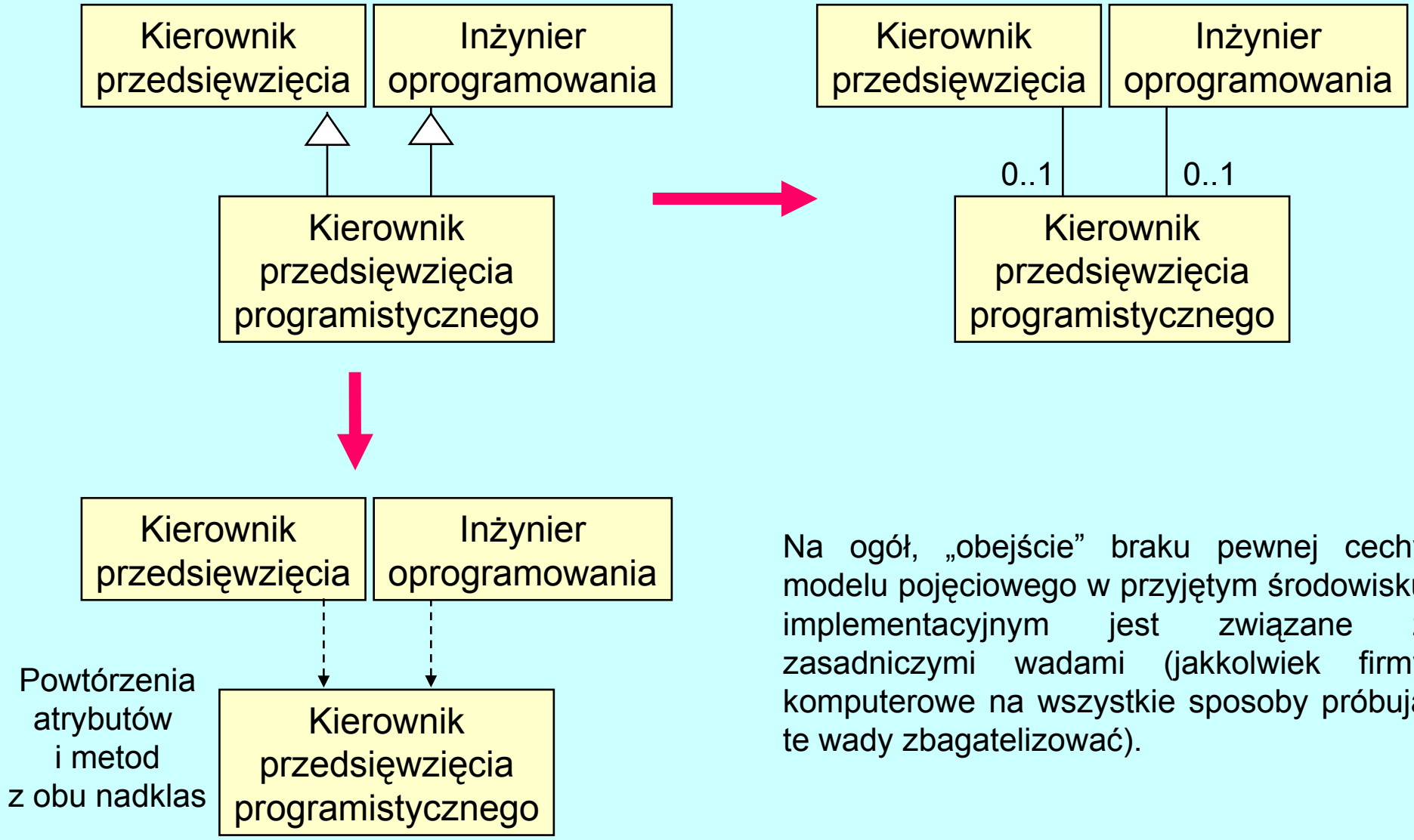
- Optymalizacja zapytań SQL

Dostosowanie do ograniczeń i możliwości środowiska implementacji

Projektant może zetknąć się z wieloma ograniczeniami implementacyjnymi, np:

- Brak dziedziczenia wielokrotnego
- Brak dziedziczenia
- Brak metod wirtualnych (przesłaniania)
- Brak złożonych atrybutów
- Brak typów multimedialnych

Przykład: obejście braku wielodziedziczenia



Na ogół, „obejście” braku pewnej cechy modelu pojęciowego w przyjętym środowisku implementacyjnym jest związane z zasadniczymi wadami (jakkolwiek firmy komputerowe na wszystkie sposoby próbują te wady zbagatelizować).

Określenie fizycznej struktury systemu

Obejmuje:

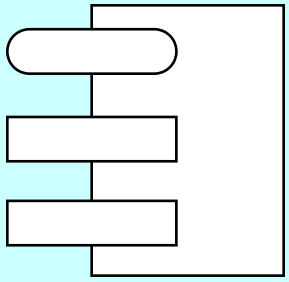
- Określenie struktury kodu źródłowego, tj. wyróżnienie plików źródłowych, zależności pomiędzy nimi oraz rozmieszczenie składowych projektu w plikach źródłowych.

- Podział systemu na poszczególne aplikacje.

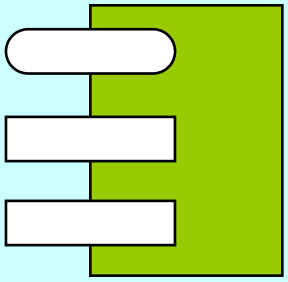
- Fizyczne rozmieszczenie danych i aplikacji na stacjach roboczych i serwerach.

Oznaczenia (Booch)

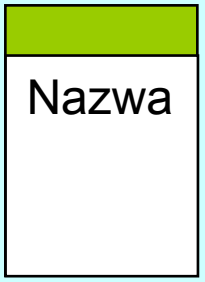
Deklaracja:



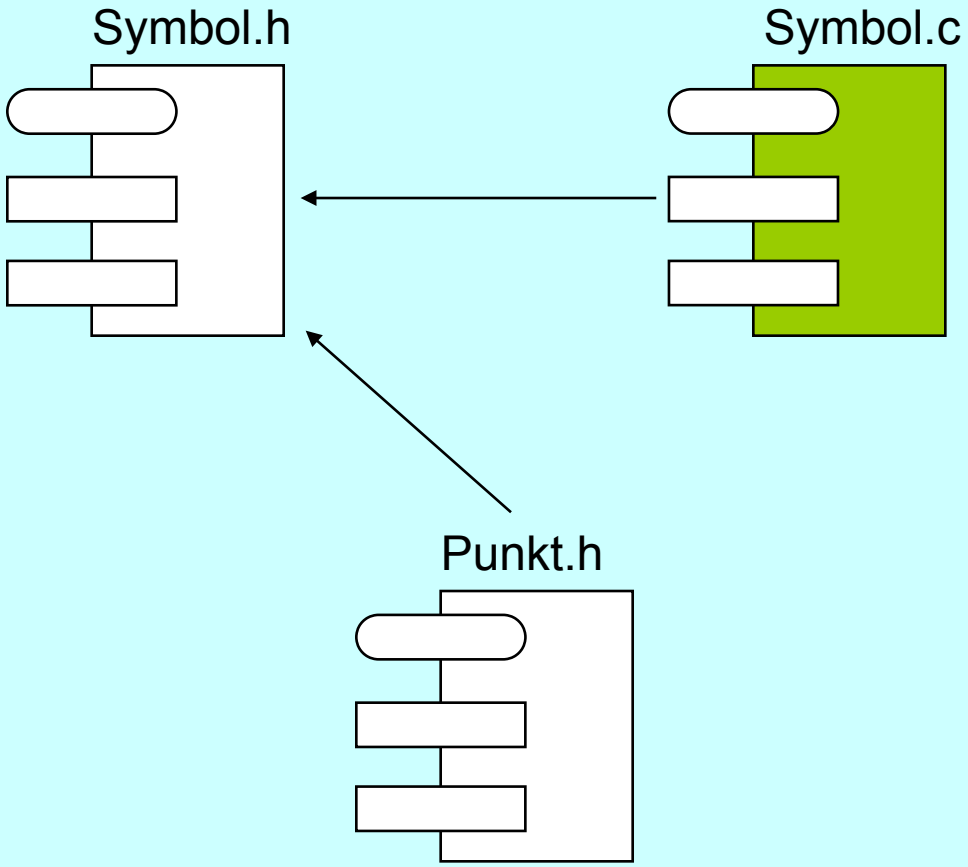
Definicja:



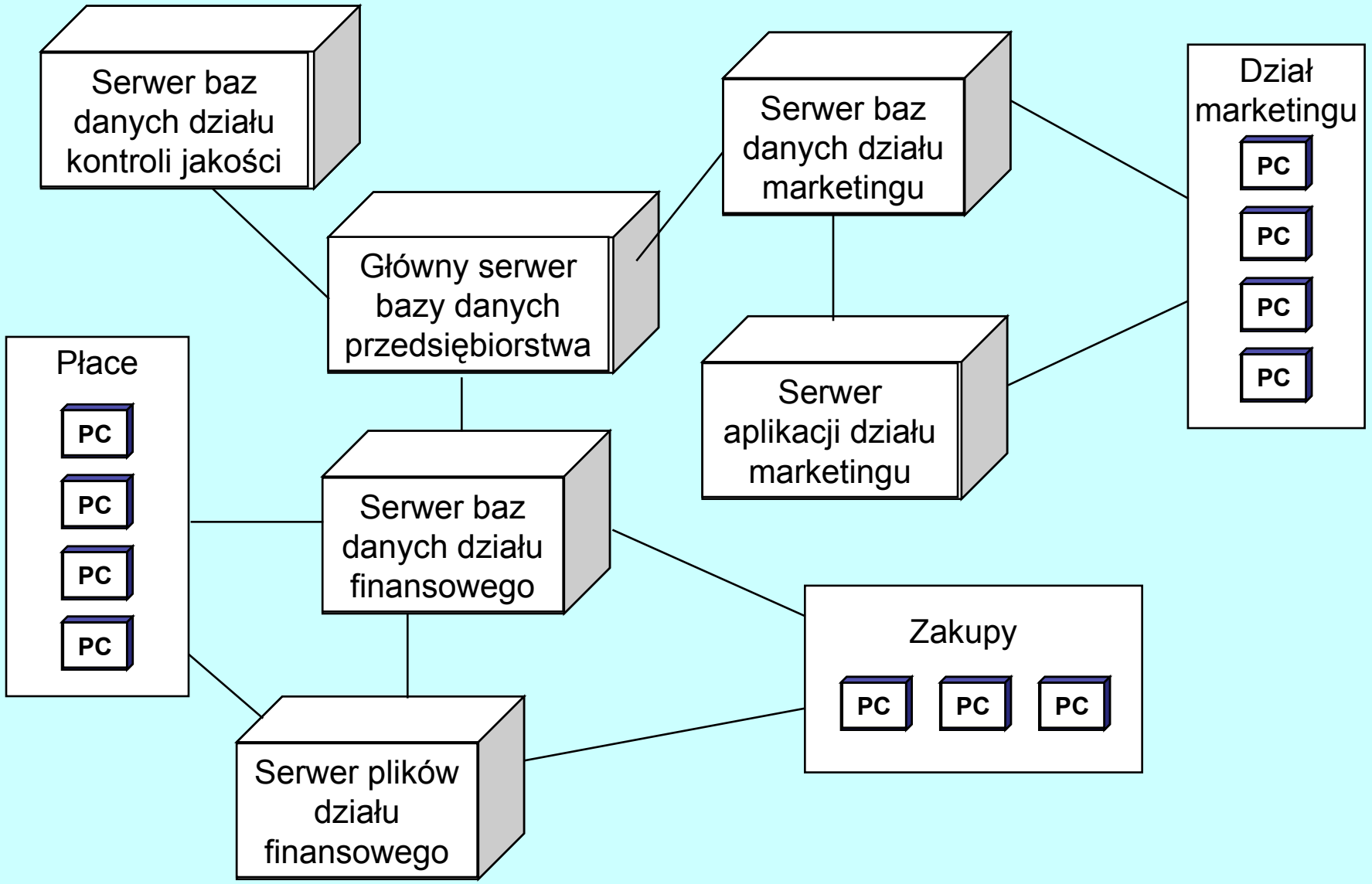
Moduł główny:



Przykład: zależności kompilacji dla C++



Graficzny opis sprzętowej konfiguracji systemu



Poprawność projektu

Poprawność oznacza, że opis projektu jest zgodny z zasadami posługiwania się notacjami. Nie gwarantuje, że projekt jest zgodny z wymaganiami użytkownika.

Poprawny projekt musi być:

* kompletny
* niesprzeczny
* spójny
* zgodny z regułami składniowymi notacji

Np. Kompletność projektu KLAS oznacza, że zdefiniowane są:

* wszystkie klasy
* wszystkie pola (atrybuty)
* wszystkie metody
* wszystkie dane złożone i elementarne

a także że opisany jest sposób realizacji wszystkich wymagań funkcjonalnych.

Spójność projektu oznacza semantyczną zgodność wszystkich informacji zawartych na poszczególnych diagramach i w specyfikacji.

Poprawność diagramów klas i stanów

Diagramy klas:

- Acykliczność związków generalizacji-specjalizacji

- Opcjonalność cyklicznych związków agregacji

- Brak klas nie powiązanych w żaden sposób z innymi klasami. Sytuacja taka może się jednak pojawić, jeżeli projekt dotyczy biblioteki klas, a nie całej aplikacji.

- Umieszczenie w specyfikacji sygnatur metod informacji o parametrach wejściowych, wyjściowych i specyfikacji wyniku

Diagramy stanów:

- Brak stanów (oprócz początkowego), do których nie ma przejścia.

- Brak stanów (oprócz końcowego), z których nie ma wyjścia.

- Jednoznaczność wyjść ze stanów pod wpływem określonych zdarzeń/warunków

Jakość projektu

Metody projektowe i stosowane notacje są w dużym stopniu nieformalne, zaś ich użycie silnie zależy od rodzaju przedsięwzięcia programistycznego.

Jest więc dość trudno ocenić jakość projektu w sensie jego adekwatności do procesu konstruowania oprogramowania i stopnia późniejszej satysfakcji użytkowników: stopień spełnienia wymagań, niezawodność, efektywność, łatwość konserwacji i ergonomiczność.

Pod terminem *jakość* rozumie się bardziej szczegółowe kryteria:

* spójność
* stopień powiązania składowych
* przejrzystość

Istotne jest spełnienie kryteriów formalnych jakości, które w dużym stopniu rzutują na efektywną jakość, chociaż w żadnym stopniu o niej nie przesądzają.
Spełnienie formalnych kryteriów jakości jest warunkiem efektywnej jakości.
Nie spełnienie tych kryteriów na ogół dyskwalifikuje efektywną jakość.

Spójność

Spójność opisuje na ile poszczególne części projektu pasują do siebie.
Istotne staje się kryterium podziału projektu na części.
W zależności od tego kryterium, możliwe jest wiele rodzajów spójności.

Kryteria podziału projektu (i rodzaje spójności):

Podział przypadkowy. Podział na moduły (części) wynika wyłącznie z tego, że całość jest za duża (utrudnia wydruk, edycję, itd)

Podział logiczny. Poszczególne składowe wykonują podobne funkcje, np. obsługa błędów, wykonywanie podobnych obliczeń.

Podział czasowy. Składowe są uruchamiane w podobnym czasie, np. podczas startu lub zakończenia pracy systemu.

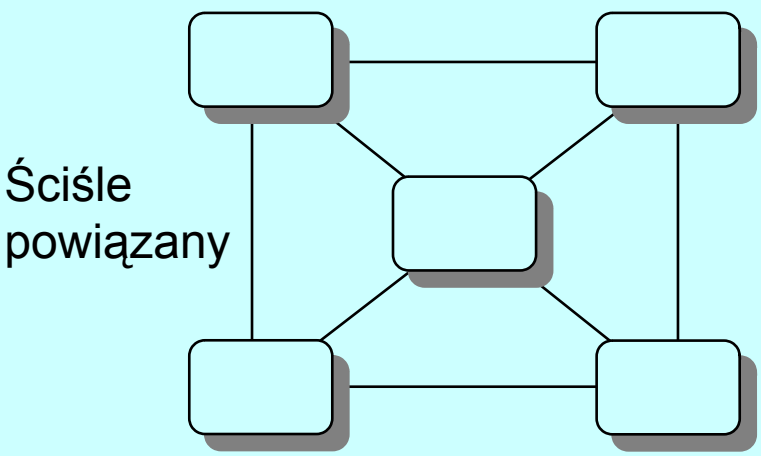
Podział proceduralny (sekwencyjny). Składowe są kolejno uruchamiane. Dane wyjściowe jednej składowej stanowią wejście innej

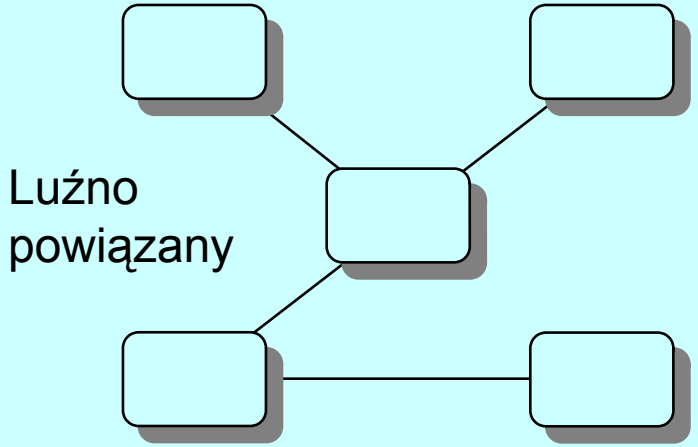
Podział komunikacyjny. Składowe działają na tym samym zbiorze danych wejściowych i wspólnie produkują zestaw danych wyjściowych

Podział funkcjonalny. Wszystkie składowe są niezbędne dla realizacji jednej tej samej funkcji.

Stopień powiązania składowych

W dobrym projekcie powinno dążyć się do tego, aby stopień powiązania pomiędzy jego składowymi był minimalny. To kryterium określa podział projektu na części zaś oprogramowanie na moduły.





Co to są “powiązania pomiędzy składowymi”?

- Korzystanie przez procesy/moduły z tych samych danych
- Przepływy danych pomiędzy procesami/modułami
- Związki pomiędzy klasami
- Przepływy komunikatów
- Dziedziczenie

Stopień powiązań można oceniać przy pomocy miar liczbowych (kohezja).

Przejrzystość

Dobry projekt powinien być przejrzysty, czyli czytelny, łatwo zrozumiały. Na przejrzystość wpływają następujące czynniki:

- Odzwierciedlenie rzeczywistości. Składowe i ich związki pojawiające się w projekcie powinny odzwierciedlać strukturę problemu. Ścisły związek projektu z rzeczywistością.

- Spójność oraz stopień powiązania składowych

- Zrozumiałe nazewnictwo

- Czytelna i pełna specyfikacja

- Odpowiednia złożoność składowych na danym poziomie abstrakcji

Na uwagę zasługuje dziedziczenie oraz przypisanie metod do klas jako czynnik przejrzystości projektu. Pozwala to znacznie uprościć i zdekomponować problem.

Wymagania niefunkcjonalne dla fazy projektowania

- Wymagania odnośnie wydajności
- Wymagania odnośnie interfejsu (protokoły, formaty plików, ...)
- Wymagania operacyjne (aspekty ergonomiczne, języki, pomoce)
- Wymagania zasobów (ilość procesorów, pojemność dysków, ...)
- Wymagania w zakresie weryfikacji (sposoby przeprowadzenia)
- Wymagania w zakresie akceptacji i testowania
- Wymagania odnośnie dokumentacji
- Wymagania odnośnie bezpieczeństwa
- Wymagania odnośnie przenaszalności
- Wymagania odnośnie jakości
    - wybór metod projektowania
    - decyzje dotyczące ponownego użycia
    - wybór narzędzi
    - wybór metod oceny projektu przez ciała zewnętrzne

- Wymagania odnośnie niezawodności
- Wymagania odnośnie podatności na pielęgnację (*maintenance*)
- Wymagania odnośnie odporności na awarie

Kluczowe czynniki sukcesu fazy projektowania

- Wysoka jakość modelu projektowego

- Dobra znajomość środowiska implementacji

- Zachowanie przyjętych standardów, np. konsekwentne stosowanie notacji i formularzy.

- Sprawdzenie poprawności projektu w ramach zespołu projektowego

- Optymalizacja projektu we właściwym zakresie. Powinna być ograniczona do istotnych, krytycznych miejsc

- Poddanie projektu ocenie przez niezależne ciało oceniające jego jakość pod względem formalnym i merytorycznym.

Podstawowe rezultaty fazy projektowania

- Poprawiony dokument opisujący wymagania
- Poprawiony model
- Uszczegółowiona specyfikacja projektu zawarta w słowniku danych
- Dokument opisujący stworzony projekt składający się z (dla m. obiektowych)
  - diagramu klas
  - diagramów interakcji obiektów
  - diagramów stanów
  - innych diagramów, np. diagramów modułów, konfiguracji
  - zestawień zawierających:
    - definicje klas
    - definicje atrybutów
    - definicje danych złożonych i elementarnych
    - definicje metod

- Zasoby interfejsu użytkownika, np. menu, dialogi
- Projekt bazy danych
- Projekt fizycznej struktury systemu
- Poprawiony plan testów
- Harmonogram fazy implementacji

Narzędzia CASE w fazie projektowania

Tradycyjnie stosuje się Lower-CASE (projektowanie struktur logicznych).

- Edytor notacji graficznych
- Narzędzia edycji słownika danych
- Generatory raportów
- Generatory dokumentacji technicznej
- Narzędzia sprawdzania jakości projektu

Narzędzia CASE powinny wspomagać proces uszczegóławiania wyników analizy. Powinny np. automatycznie dodawać atrybuty realizujące związki pomiędzy klasami. Powinny ułatwiać dostosowanie projektu do środowiska implementacji.

Powinna istnieć możliwość automatycznej transformacji z modelu obiektów na schemat relacyjnej bazy danych.

Niektóre narzędzia CASE umożliwiają projektowanie interfejsu użytkownika.

Narzędzia inżynierii odwrotnej (*reverse engineering*), dla odtworzenia projektu na podstawie istniejącego kodu.

Zawartość dokumentu projektowego

Celem (**Dokładnego**) **Dokumentu Detalicznego Projektu** (**DDP**) jest szczegółowy opis rozwiązania problemu określonego w dokumencie wymagań na oprogramowanie. DDP musi uwzględniać wszystkie wymagania. Powinien być wystarczająco detaliczny aby umożliwić implementację i pielęgnację kodu.

Styl DDP powinien być systematyczny i rygorystyczny. Język i diagramy użyte w DDP powinny być klarowne. Dokument powinien być łatwo modyfikowalny.

Struktura DDP powinna odpowiadać strukturze projektu oprogramowania. Język powinien być wspólny dla całego dokumentu. Wszystkie użyte terminy powinny być zdefiniowane i użyte w zdefiniowanym znaczeniu.

Zasady wizualizacji diagramów:

- wyróżnienie ważnych informacji,
- wyrównanie użytych oznaczeń,
- diagramy powinny być czytane od lewej do prawej oraz z góry do dołu,
- podobne pozycje powinny być zorganizowane w jeden wiersz, w tym samym stylu,
- symetria wizualna powinna odzwierciedlać symetrię funkcjonalną,
- należy unikać przecinających się linii i nakładających się oznaczeń i rysunków,
- należy unikać nadmiernego zagęszczenia diagramów.

Modyfikowalność, ewolucja, odpowiedzialność

Modyfikowalność dokumentu. Tekst, diagramy, wykresy, itd. powinny być zapisane w formie, którą można łatwo zmodyfikować. Należy kontrolować nieprzewidywalne efekty zmian, np. lokalnych zmian elementów, które są powtórzone w wielu miejscach dokumentu i powiązane logicznie.

Ewolucja dokumentu. DDP powinien podlegać rygorystycznej kontroli, szczególnie jeżeli jest tworzony przez zespół ludzi. Powinna być zapewniona formalna identyfikacja dokumentów, ich wersji oraz ich zmian. Wersje powinny być opatrzone unikalnym numerem identyfikacyjnym i datą ostatniej zmiany. Powinno istnieć centralne miejsce, w którym będzie przechowywana ostatnia wersja.

Odpowiedzialność za dokument. Powinna być jednoznacznie zdefiniowana. Z reguły, odpowiedzialność ponosi osoba rozwijająca dane oprogramowanie. Może ona oddelegować swoje uprawnienia do innych osób dla realizacji konkretnych celów związanych z tworzeniem dokumentu.

Medium dokumentu Należy przyjąć, że wzorcowa wersja dokumentu będzie w postaci elektronicznej, w dobrze zabezpieczonym miejscu. Wszelkie inne wersje, w tym wersje papierowe, są pochodną jednej, wzorcowej wersji.

Dalsze zalecenia odnośnie DDP (Dokumentu Detalicznego Projektu)

- DDP jest centralnym miejscem, w którym zgromadzone są wszystkie informacje odnośnie budowy i działania oprogramowania.

- DDP powinien być zorganizowany w taki sam sposób, w jaki zorganizowane jest oprogramowanie.

- DDP powinien być kompletny, odzwierciedlający wszystkie wymagania zawarte w Specyfikacji wymagań.

- Materiał, który nie mieści się w podanej zawartości dokumentu, powinien być załączony jako dodatek.

- Nie należy zmieniać numeracji punktów. Jeżeli jakiś punkt nie jest zapełniony, wówczas należy pozostawić jego tytuł, zaś poniżej zaznaczyć "Nie dotyczy."

Zawartość DDP (1)

Informacja organizacyjna

a - Streszczenie
b - Spis treści
c - Formularz statusu dokumentu
d - Zapis zmian w stosunku do ostatniej wersji

CZĘŚĆ 1 - OPIS OGÓLNY

1. **WPROWADZENIE**

Opisuje cel i zakres, określa użyte terminy, listę referencji oraz krótko omawia dokument.

1.1. Cel Opisuje cel DDP oraz specyfikuje przewidywany rodzaj jego czytelnika.

1.2. Zakres

Identyfikuje produkt programistyczny będący przedmiotem dokumentu, objaśnia co oprogramowanie robi (i ewentualnie czego nie robi) oraz określa korzyści, założenia i cele. Opis ten powinien być spójny z dokumentem nadrzędnym, o ile taki istnieje.

1.3. Definicje, akronimy, skróty

1.4. Odsyłacze

1.5. Krótkie omówienie

2. **STANDARDY PROJEKTU, KONWENCJE, PROCEDURY**

2.1. Standardy projektowe

2.2. Standardy dokumentacyjne

2.3. Konwencje nazwowe

2.4. Standardy programistyczne

2.5. Narzędzia rozwijania oprogramowania

Zawartość DDP (2)

CZĘŚĆ II - SPECYFIKACJA KOMPONENTÓW

n [IDENTYFIKATOR KOMPONENTU]
n.1. Typ
n.2. Cel
n.3. Funkcja
n.4. Komponenty podporządkowane
n.5. Zależności
n.6. Interfejsy
n.7. Zasoby
n.8. Odsyłacze
n.9. Przetwarzanie
n.10. Dane

Dodatek A. Wydruki kodu źródłowego
Dodatek B. Macierz zależności pomiędzy zbiorem wymagań i zbiorem komponentów oprogramowania
Dodatek C. Uzasadnienie spełnienia wymagań niefunkcjonalnych

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę

INŻYNIERIA OPROGRAMOWANIA

wykład 7:
MIARY OPROGRAMOWANIA

dr inż. Leszek Grocholski
Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

POMIAR OPROGRAMOWANIA

Pomiar (*measurement*) jest to proces, w którym atrybutom świata rzeczywistego przydzielane są liczby lub symbole w taki sposób, aby charakteryzować te atrybuty według jasno określonych zasad.

Jednostki przydzielane atrybutom nazywane są **miarą** danego atrybutu.

Metryka (*metric*) jest to proponowana (postulowana) miara.

Nie zawsze charakteryzuje ona w sposób obiektywny dany atrybut. Np. ilość linii kodu (LOC) jest metryką charakteryzującą atrybut “długość programu źródłowego”, ale nie jest miarą ani złożoności ani rozmiaru programu (choć występuje w tej roli).

Co mierzyć ?

Proces: każdy określony ciąg działań w ramach analizy, projektowana, wytwarzania lub eksploatacji oprogramowania.

Produkt: każdy przedmiot powstały w wyniku procesu: kod źródłowy, specyfikację projektową, udokumentowaną modyfikację, plan testów, dokumentację, itd.

Zasób: każdy element niezbędny do realizacji procesu: osoby, narzędzia, metody wytwarzania, itd.

Elementy pomiaru oprogramowania - PRODUKTY

Obiekty	**Atrybuty bezpośrednio mierzalne**	**Wskaźniki syntetyczne**
Specyfikacje	rozmiar, ponowne użycie, modularność, nadmiarowość, funkcjonalność, poprawność składniową, ...	zrozumiałość, pielęgnacyjność, ...
Projekty	rozmiar, ponowne użycie, modularność, spójność, funkcjonalność,...	jakość, złożoność, pielęgnacyjność, ...
Kod	rozmiar, ponowne użycie, modularność, spójność, złożoność, strukturalność, ...	niezawodność, używalność, pielęgnacyjność, ...
Dane testowe	rozmiar, poziom pokrycia,...	jakość,...
....	....	....

Elementy pomiaru oprogramowania - PRODUKTY cd ..

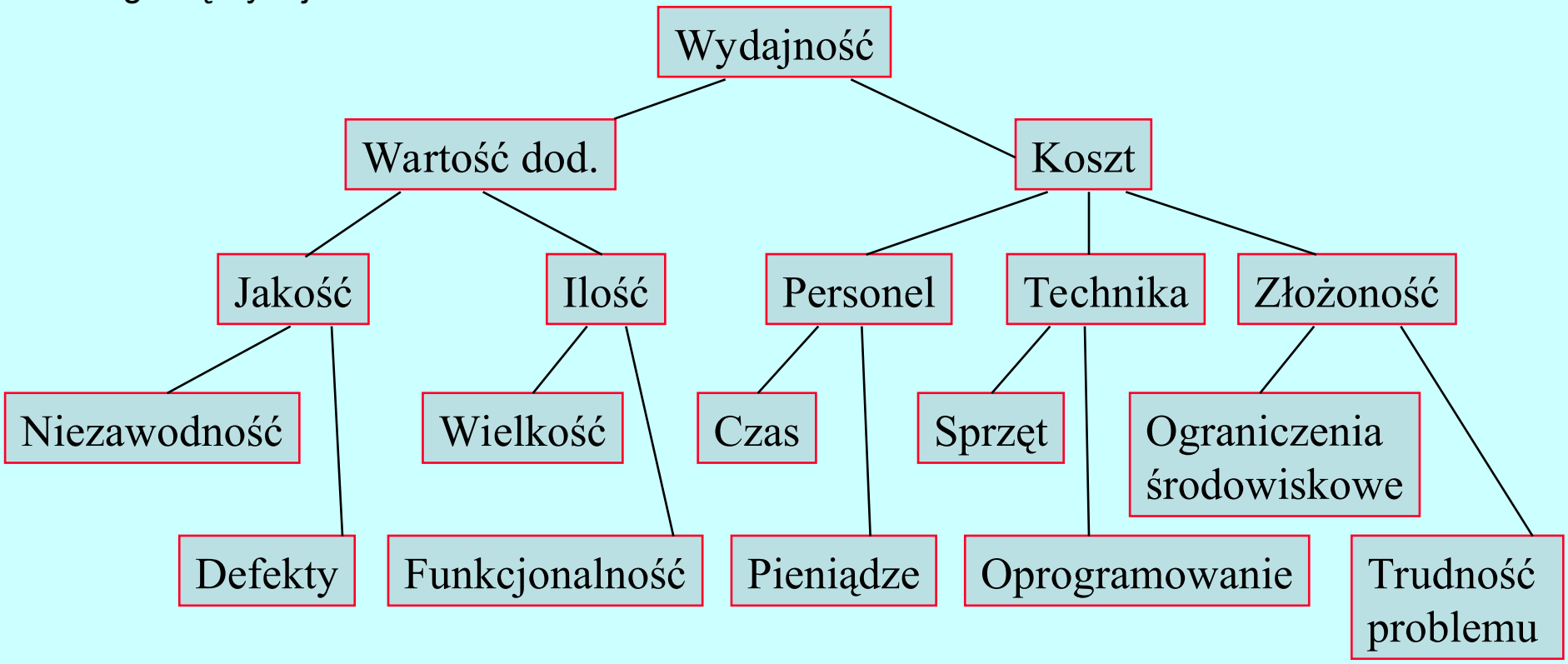
Obiekty	**Atrybuty bezpośrednio mierzalne**	**Wskaźniki syntetyczne**
Specyfikacja architektury	czas, nakład pracy, liczba zmian wymagań, ...	jakość, koszt, stabilność, ...
Projekt szczegółowy	czas, nakład pracy, liczba znalezionych usterek specyfikacji,...	koszt, opłacalność, ...
Testowanie	czas, nakład pracy, liczba znalezionych błędów kodu, ...	koszt, opłacalność, stabilność, ...
....	....	....

Elementy pomiaru oprogramowania - ZASOBY

Obiekty	**Atrybuty bezpośrednio mierzalne**	**Wskaźniki syntetyczne**
Personel	wiek, cena, ...	wydajność, doświadczenie, inteligencja, ...
Zespoły	wielkość, poziom komunikacji, struktura,...	wydajność, jakość, ...
Oprogramowanie	cena, wielkość, ...	używalność, niezawodność, ...
Sprzęt	cena, szybkość, wielkość pamięci	niezawodność, ...
Biura	wielkość, temperatura, oświetlenie,...	wygoda, jakość,...
....	....	....

Model i miara wydajności ludzi

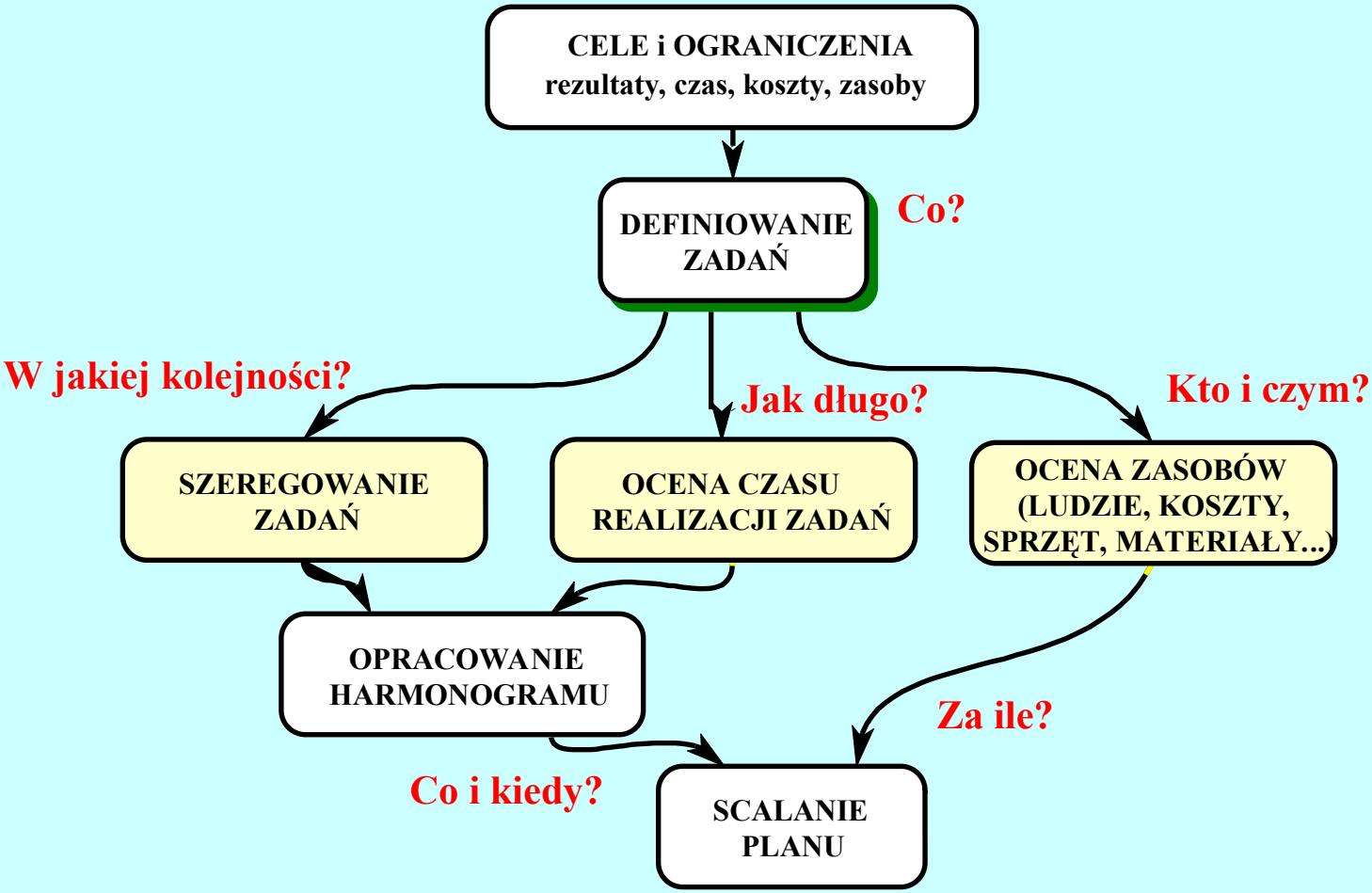
Czynniki wpływające
na ogólną wydajność



Uwaga: Mylące, wręcz niebezpieczne jest zastępowanie wielu miar jedną miarą np. długością wyprodukowanego kodu.

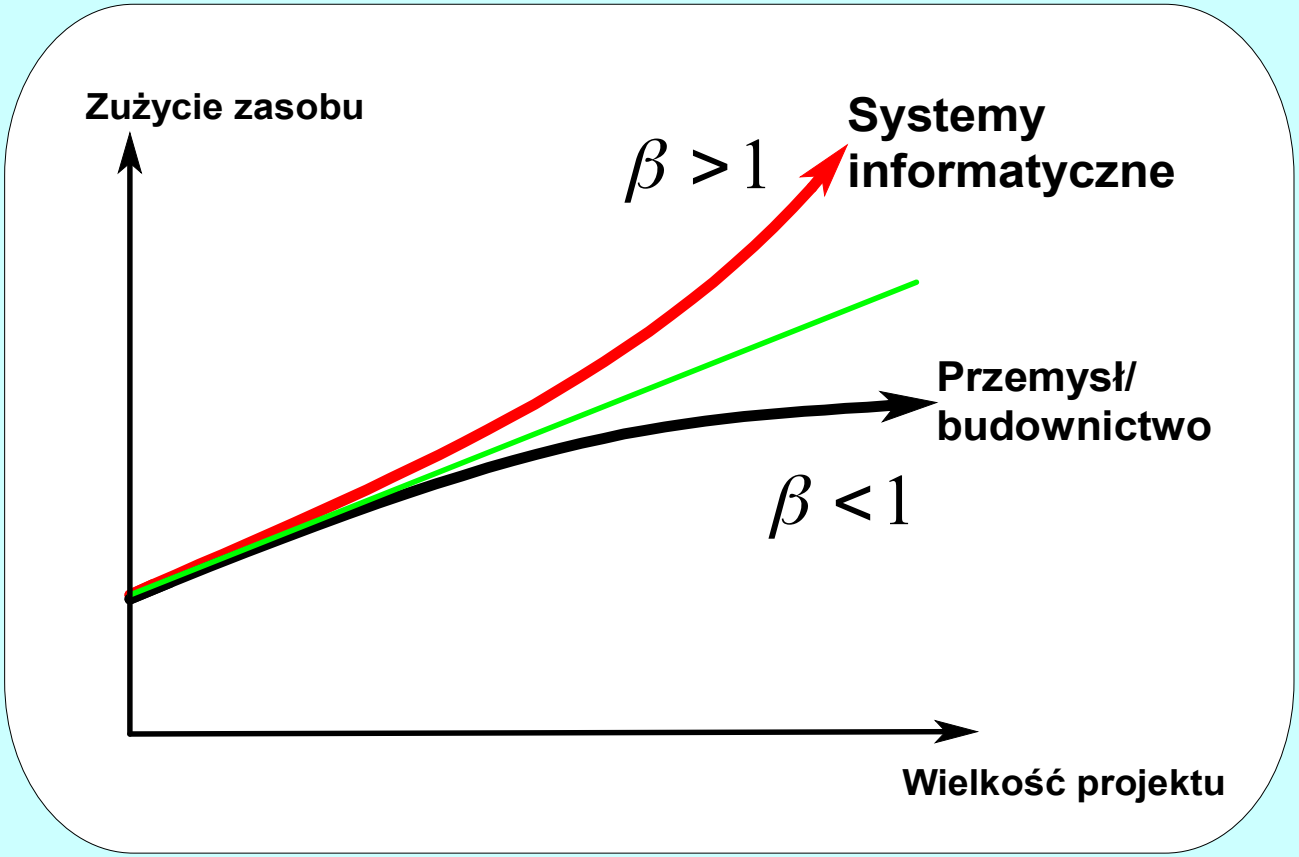
Ocena złożoności w planowaniu projektu

(tak jak odmiana przez przypadki, kto?, co ….)



Efekt skali

Zużycie Zasobu = Zużycie Stałe + K * Wielkość Projektu $^{\beta}$



Czynniki wpływające na efekt skali

Czynniki spłaszczenia krzywej:

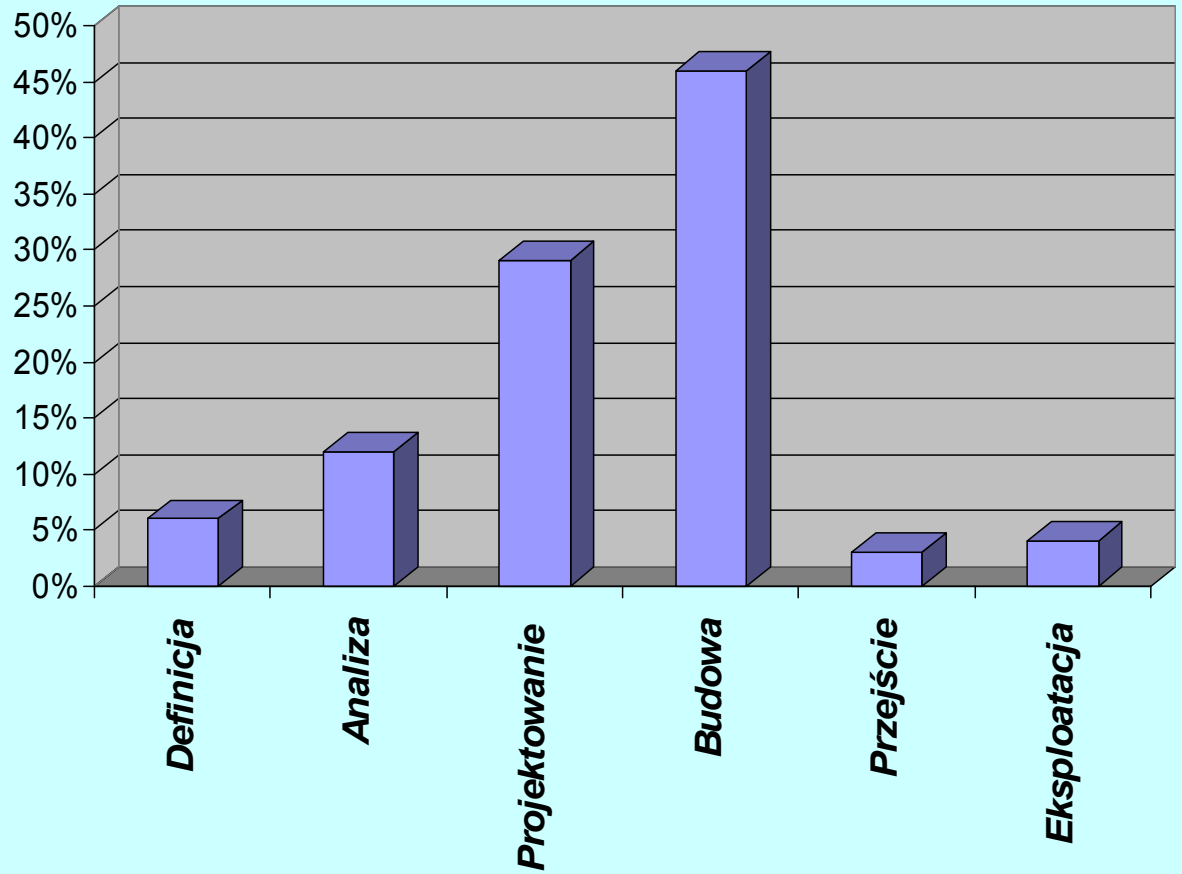
- Specjalizacja
- Uczenie się, doświadczenie
- Narzędzia CASE
- Wspomaganie dokumentowania
- Biblioteki gotowych elementów
- Stałe koszty projektu

Czynniki wzrostu krzywej:

- Koszty zarządzania
  (czas produkcyjny/nie)
- Lawinowy wzrost ilości powiązań
- Komunikacja wewnątrz zespołu
  → czas szybko wzrasta w miarę dodawania nowych członków
- Wzrost złożoności testowania

FAZY a koszty wytwarzania oprogramowania

Empiryczne koszty poszczególnych faz wytwarzania oprogramowania systemów informatycznych:



Źródło: Oracle Corp. Badaniom podlegały realizacje systemów przetwarzania danych, realizowane metodą CDM, prze użyciu narzędzi CASE firmy Oracle.

Metoda szacowania kosztów COCOMO

COnstructive COst MOdel

Wymaga oszacowania liczby instrukcji, z których będzie składał się system. Rozważane przedsięwzięcie jest następnie zaliczane do jednej z klas:

Przedsięwzięć łatwych (organicznych, *organic*). Klasa ta obejmuje przedsięwzięcia wykonywane przez stosunkowo małe zespoły, złożone z osób o podobnych wysokich kwalifikacjach. Dziedzina jest dobrze znana. Przedsięwzięcie jest wykonywane przy pomocy dobrze znanych metod i narzędzi.

Przedsięwzięć niełatwych (pół-oderwanych, *semi-detached*). Członkowie zespołu różnią się stopniem zaawansowania. Pewne aspekty dziedziny problemu nie są dobrze znane.

Przedsięwzięć trudnych (np.. osadzonych, *embedded*). Obejmują przedsięwzięcia realizujące systemy o bardzo złożonych wymaganiach. Dziedzina problemu, stosowane narzędzia i metody są w dużej mierze nieznane. Większość członków zespołu nie ma doświadczenia w realizacji podobnych zadań.

Metoda szacowania kosztów COCOMO (2)

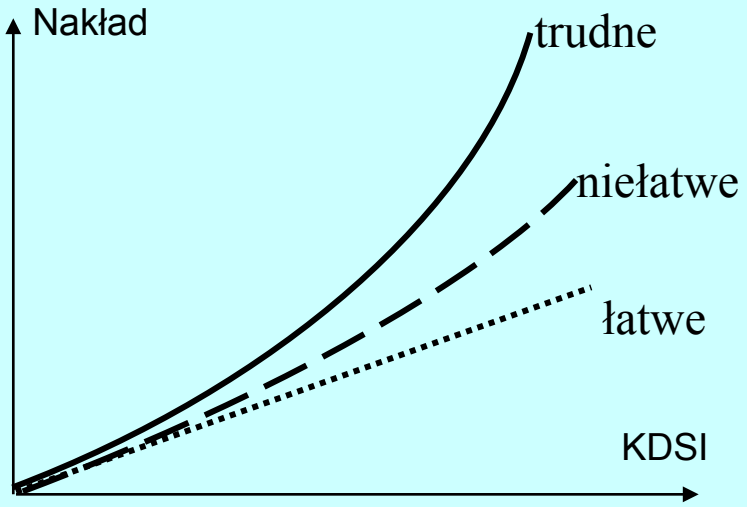
Podstawowy wzór dla oszacowania nakładów w metodzie COCOMO:

$$\text{Nakład [osobomiesiące]} = A * K^{b}$$ (zależność wykładnicza)

K (określane jako KDSI, *Kilo (thousand) of Delivered Source Code Instructions*) oznacza rozmiar kodu źródłowego mierzony w tysiącach linii. KDSI nie obejmuje kodu, który nie został wykorzystany w systemie.
Wartości stałych A i b zależą od klasy, do której zaliczono przedsięwzięcie:

Przedsięwzięcie łatwe:	$\text{Nakład} = 2.4 * K^{1.05}$
Przedsięwzięcie niełatwe:	$\text{Nakład} = 3 * K^{1.12}$
Przedsięwzięcie trudne:	$\text{Nakład} = 3.6 * K^{1.20}$

Dla niewielkich przedsięwzięć są to zależności bliskie liniowym. Wzrost jest szczególnie szybki dla przedsięwzięć trudnych (duży rozmiar kodu).



Metoda szacowania kosztów COCOMO (3)

Metoda COCOMO zakłada, że znając nakład można oszacować czas realizacji przedsięwzięcia, z czego wynika przybliżona wielkość zespołu. Z obserwacji wiadomo, że dla każdego przedsięwzięcia istnieje optymalna liczba członków zespołu wykonawców. Zwiększenie tej liczby może nawet wydłużyć czas realizacji.
Proponowane są następujące wzory:

Przedsięwzięcie łatwe:	Czas [miesiące] = 2.5 * Nakład $^{0.32}$
Przedsięwzięcie niełatwe:	Czas [miesiące] = 2.5 * Nakład $^{0.35}$
Przedsięwzięcie trudne:	Czas [miesiące] = 2.5 * Nakład $^{0.38}$

Otrzymane w ten sposób oszacowania powinny być skorygowane przy pomocy tzw. czynników modyfikujących. Biorą one pod uwagę następujące atrybuty przedsięwzięcia:

- wymagania wobec niezawodności systemu
- rozmiar bazy danych w stosunku do rozmiaru kodu
- złożoność systemu: złożoność struktur danych, złożoność algorytmów, komunikacja z innymi systemami, stosowanie obliczeń równoległych
- wymagania co do wydajności systemu
- ograniczenia pamięci
- zmienność sprzętu i oprogramowania systemowego tworzącego środowisko pracy systemu

Wady metody COCOMO

Liczba linii kodu jest znana dokładnie dopiero wtedy, gdy system jest napisany. Szacunki są zwykle obarczone bardzo poważnym błędem (niekiedy ponad 100%)

Określenie “linii kodu źródłowego” inaczej wygląda dla każdego języka programowania. Jedna linia w Smalltalk’u jest równoważna 10-ciu linii w C. Dla języków 4GL i języków zapytań ten stosunek może być nawet 1000 : 1.

Koncepcja oparta na liniach kodu źródłowego jest całkowicie nieadekwatna dla nowoczesnych środków programistycznych, np. opartych o programowanie wizyjne.

Zły wybór czynników modyfikujących może prowadzić do znacznych rozbieżności pomiędzy oczekiwanym i rzeczywistym kosztem przedsięwzięcia.

Żadna metoda przewidywania kosztów nie jest więc doskonała i jest oparta na szeregu arbitralnych założeń. Niemniej dla celów planowania tego rodzaju metody stają się koniecznością. Nawet niedoskonała metoda jest zawsze lepsza niż “sufit”.

Analiza Punktów Funkcyjnych

Metoda analizy punktów funkcyjnych (FPA), opracowana przez Albrechta z firmy IBM w latach 1979-1983 bada pewien zestaw wartości. Łączy ona własności metody, badającej rozmiar projektu programu z możliwościami metody badającej produkt programowy.

Liczbę nie skorygowanych punktów funkcyjnych wylicza się na podstawie formuły korzystając z następujących danych:

- **Wejścia użytkownika**: obiekty wejściowe wpływających na dane w systemie
- **Wyjścia użytkownika:** obiekty wyjściowe związane z danymi w systemie
- **Zbiory danych wewnętrzne**: liczba wewnętrznych plików roboczych
- **Zbiory danych zewnętrzne:** liczba plików zewnętrznych zapełnianych przez system
- **Zapytania (interfejsy) zewnętrzne:** interfejsy z otoczeniem programu

UFP - Nieskorygowane Punkty Funkcyjne

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} \star n_{ij}$$

UFP- nieskorygowane punkty funkcyjne

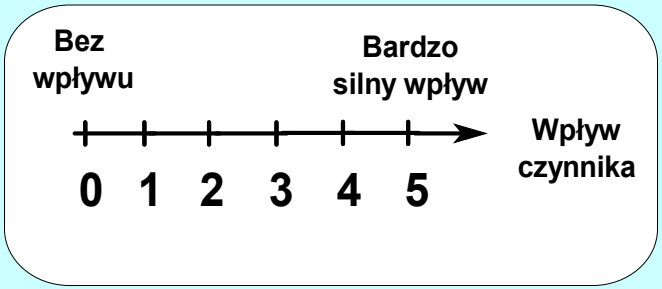
gdzie: w_{ij} - wagi, n_{ij} - ilość elementów

		Wagi przypisywane komponentom:		
	Czynnik złożoności	**prosty**	**średni**	**złożony**
		3	4	6
i = 1	Wejścia użytkownika	4	5	7
i = 2	Wyjścia użytkownika	7	10	15
i = 3	Zbiory danych wewnętrzne	5	7	10
i = 4	Zbiory danych zewnętrzne	3	4	6
i = 5	Zapytania zewnętrzne			
		j = 1	j = 2	j = 3

Korekcja Punktów Funkcyjnych

1. występowanie urządzeń komunikacyjnych
2. rozproszenie przetwarzania
3. długość czasu oczekiwania na odpowiedź systemu (nacisk na szybkość)
4. stopień obciążenia istniejącego sprzętu
5. częstotliwość wykonywania dużych transakcji
6. wprowadzanie danych w trybie bezpośrednim
7. wydajność użytkownika końcowego
8. aktualizacja danych w trybie bezpośrednim
9. złożoność przetwarzania danych
10. możliwość ponownego użycia programów w innych zastosowaniach
11. łatwość instalacji
12. łatwość obsługi systemu
13. rozproszenie terytorialne
14. łatwość wprowadzania zmian - pielęgnowania systemu

Skorygowane Punkty Funkcyjne



kompleksowy współczynnik korygujący

$$VAF = \sum_{k=1}^{14} k_k$$

(Skorygowane) punkty funkcyjne (FPs):

$$FP = (0{,}65 + 0{,}01 * VAF) * UFP$$

$$FP = (0{,}65 \ldots 1{,}35) * UFP$$

Kolejność obliczeń Punktów Funkcyjnych

1. Identyfikacja systemu
2. Obliczenie współczynnika korygującego
3. Wyznaczenie ilości zbiorów danych i ich złożoności
4. Wyznaczenie ilości i złożoności elementów funkcjonalnych (we, wy, zbory we, wy)
5. Realizacja obliczeń
6. Weryfikacja
7. Raport, zebranie recenzujące

Przykład obliczania punktów funkcyjnych

Elementy	Proste	Waga	Średnie	Waga	Złożone	Waga	Razem
Wejścia	**2**	3	**5**	4	**3**	6	**44**
Wyjścia	**10**	4	**4**	5	**5**	7	**95**
Zbiory wew.	**3**	7	**5**	10	**2**	15	**101**
Zbiory zew.	**0**	5	**3**	7	**0**	10	**21**
Zapytania	**10**	3	**5**	4	**12**	6	**122**
							Łącznie 383

Aplikacje a Punkty Funkcyjne

1 FP $\approx$ 125 instrukcji w C

10 FPs - typowy mały program tworzony samodzielnie przez jednego „dobrego” programistę przez 1 miesiąc

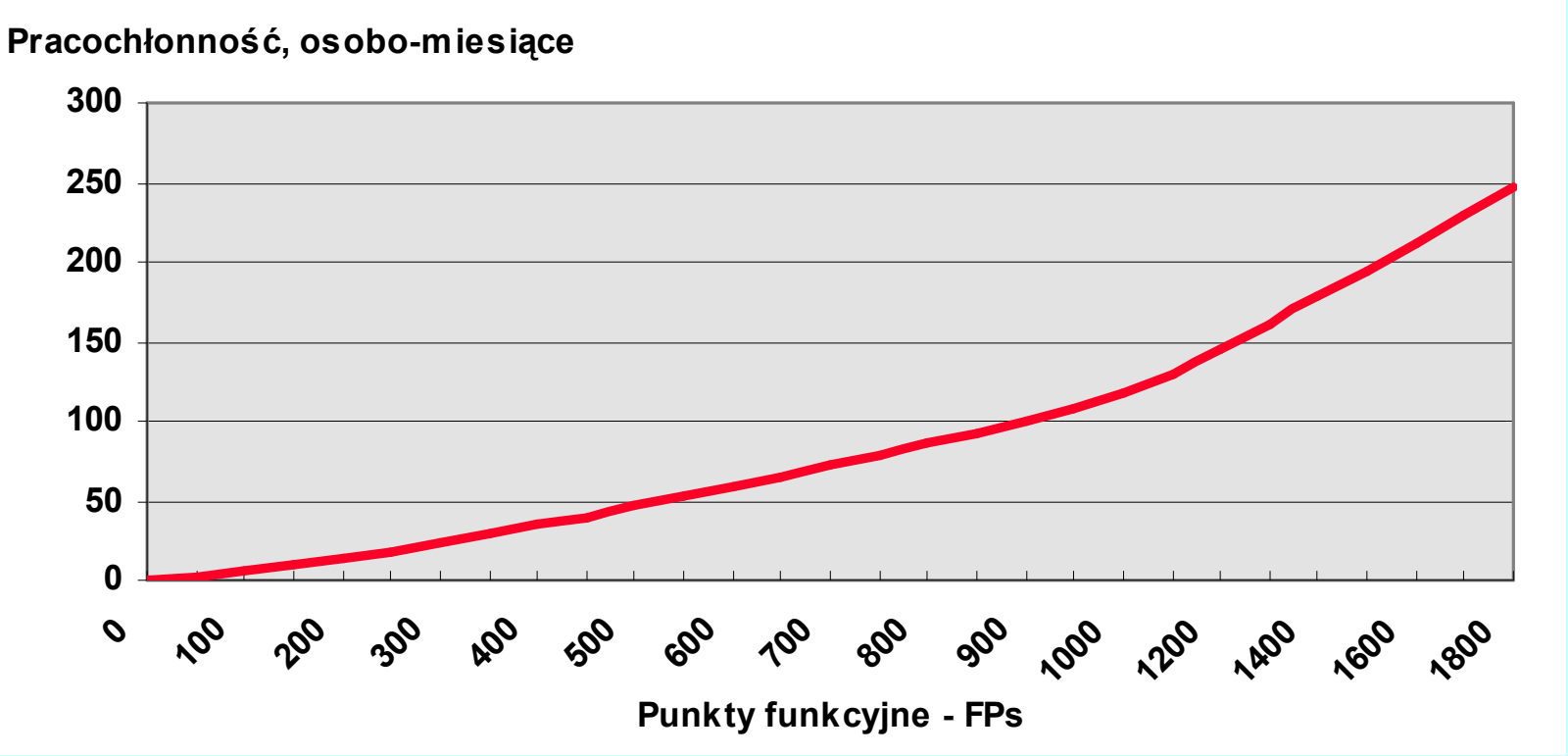
100 FPs - większość popularnych aplikacji; wartość typowa dla aplikacji tworzonych przez klienta samodzielnie (6 m-cy)

1 000 FPs - komercyjne aplikacje w MS Windows, małe aplikacje klient-serwer (10 osób, ponad 12 m-cy)

10 000 FPs - systemy złożone (100 osób, ponad 18 m-cy)

100 000 FPs - MS Windows NT, MVS, systemy militarne

Punkty Funkcyjne a pracochłonność



Wykorzystanie punktów funkcyjnych

- Ocena złożoności realizacji systemów

- Audyt projektów

- Wybór systemów informatycznych funkcjonujących w przedsiębiorstwie do reinżynierii (wg. koszt utrzymania/FPs)

- Szacowanie liczby testów

- Ocena jakości pracy i wydajności zespołów ludzkich

- Ocena stopnia zmian, wprowadzanych przez użytkownika na poszczególnych etapach budowy systemu informatycznego

- Prognozowanie kosztów pielęgnacji i rozwoju systemów

- Porównanie i ocena różnych ofert dostawców oprogramowania pod kątem merytorycznym i kosztowym

Punkty Funkcyjne a języki baz danych

Typ języka lub konkretny język	Poziom języka wg. SPR	Efektywność LOC/FP
Access	8.5	38
ANSI SQL	25.0	13
CLARION	5.5	58
CA Clipper	17.0	19
dBase III	8.0	40
dBase IV	9.0	36
DELPHI	11.0	29
FOXPRO 2.5	9.5	34
INFORMIX	8.0	40
MAGIC	15.0	21
ORACLE	8.0	40
Oracle Developer 2000	14.0	23
PROGRESS v.4	9.0	36
SYBASE	8.0	40

wg. Software Productivity Research

Punkty Funkcyjne a wydajność zespołu

Poziom języka wg. SPR *	Średnia produktywność FPs/osobomiesiąc
1 - 3	5 - 10
4 - 8	10 - 20
9 - 15	16 - 23
16 - 23	15 - 30
24 - 55	30 - 50
>55	40 - 100

** wg. Software Productivity Research*

Uwzglednienie różnorodnych aspektow

Różnorodne metryki uwzględniają m.in. następujące aspekty:

- wrażliwość na błędy,
- możliwości testowania,
- częstotliwość występowania awarii,
- dostępność systemu,
- propagacja błędów,
- ilość linii kodu, złożoność kodu, złożoność programu,
- złożoność obliczeniową, funkcjonalną, modułową,
- łatwość implementacji,
- rozmiar dokumentacji,
- ilość zadań wykonanych terminowo i po terminie,
- współzależność zadań,
- wielkość i koszt projektu,
- czas trwania projektu,
- zagrożenia projektu (ryzyko),
- czas gotowości produktu,
- kompletność wymagań, kompletność planowania,
- stabilność wymagań,
- odpowiedniość posiadanych zasobów sprzętowych, materiałowych i ludzkich,
- efektywność zespołu, efektywność poszczególnych osób.

Przykłady metryk oprogramowania

- Metryki zapisu projektu, kodu programu

- rozmiar projektu, kodu programu (liczba modułów/obiektów, liczba linii kodu, komentarza, średni rozmiar komponentu)
- liczba, złożoność jednostek syntaktycznych i leksykalnych
- złożoność struktury i związków pomiędzy komponentami programu

  (procesy, funkcje, moduły, obiekty itp..)

- Metryki uzyskiwanego produktu

- rozmiar
- architektura
- struktura
- jakość użytkowania i pielęgnacji
- złożoność

Przykłady metryk oprogramowania, cd.

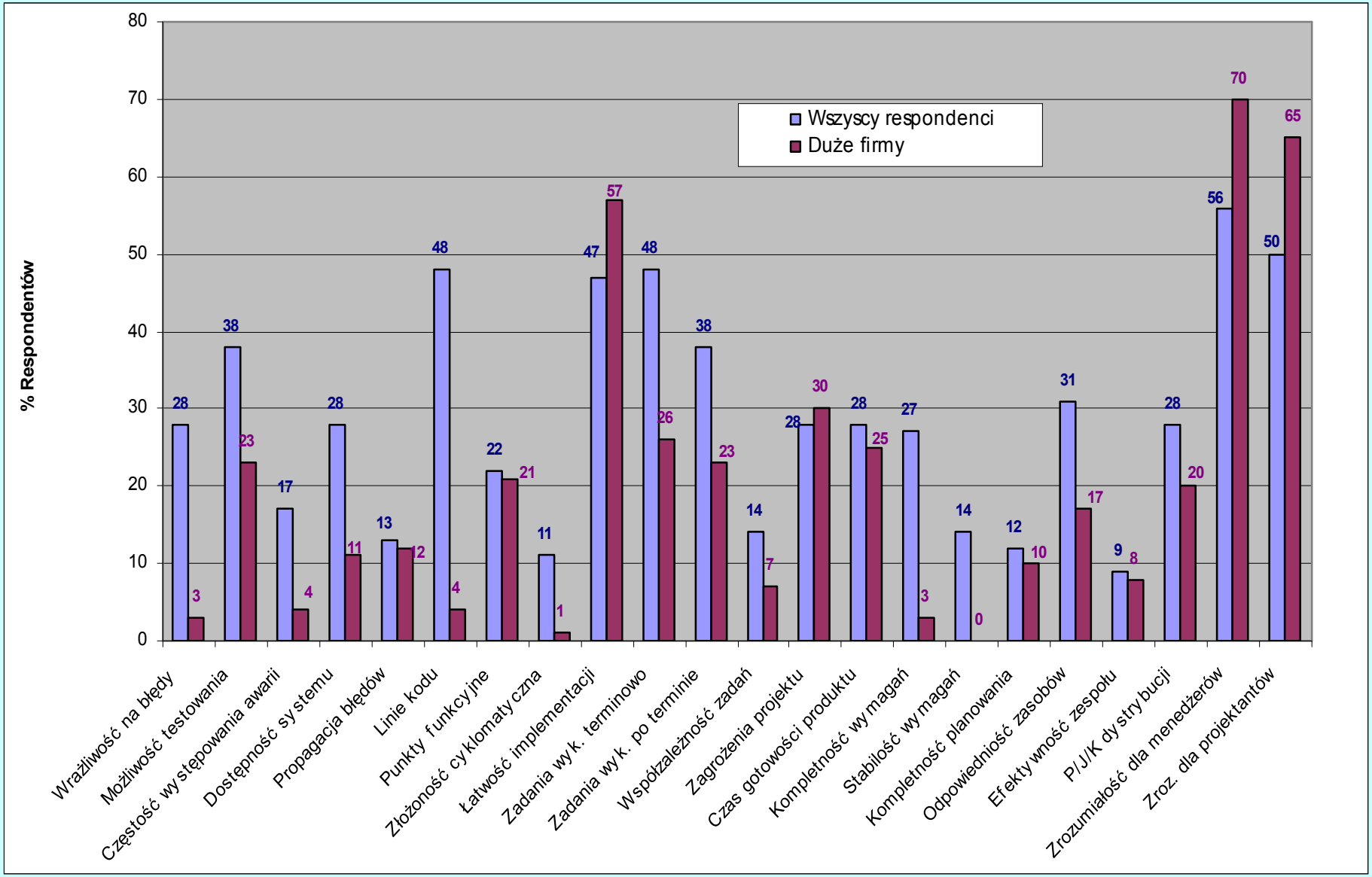
- Metryki procesu wytwarzania

- dojrzałość procesów systemu !
- ocena zarządzanie wytwarzaniem oprogramowania w odniesieniu do cyklu życia oprogramowania

- Metryki zasobów realizacyjnych

- w odniesieniu do personelu „zaangażowanego” w realizację
- narzędzia software’owe, wykorzystywane przy realizacji
- sprzęt, jakim dysponuje wykonawca,
- inne, np. samochody, lokale …..

Wykorzystanie metod estymacyjnych



Podsumowanie

- **Metryki są tworzone i stosowane na bazie doświadczenia i zdrowego rozsądku, co obniża ich wartość dla tzw. „teoretyków informatyki”!**

- Metryki powinny być wykorzystywane jako metody wspomagania ekspertów. Bo metryki stosowane formalistycznie mogą być groźne.
- Najlepiej jest stosować zestawy metryk, co pozwala zmniejszyć błędy pomiarowe.
- Przede wszystkim zdrowy rozsądek i doświadczenie.
- Pomimo pochodzenia empirycznego, metryki skutecznie pomagają w szybkiej i mniej subiektywnej ocenie oprogramowania.
- Specjalizacja metryk w kierunku konkretnej klasy oprogramowania powinna dawać lepsze i bardziej adekwatne oceny niż metryki uniwersalne.
- Wskazane jest wspomaganie metod opartych na metrykach narzędziami programistycznymi.

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę

INŻYNIERIA OPROGRAMOWANIA

wykład 8:
ZARZĄDZANIE PROJEKTEM PROGRAMISTYCZNYM

dr inż. Leszek Grocholski
Zakład Inżynierii Oprogramowania
Instytut Informatyki
Uniwersytet Wrocławski

ZARZĄDZANIE - definicja

Zarządzanie - wyszukiwarka poda linki do ponad 26,5 miliona artykułów.

Zarządzanie jako działanie praktyczne istnieje od dawien dawna !
Szczególnie rozwijało się podczas i po wojnach światowych.

Definicje

1. **Zestaw działań** (obejmujący planowanie i podejmowane decyzji, organizowanie, przewodzenie, tj. kierowanie ludźmi, i kontrolowanie), skierowanych na zasoby organizacji (ludzkie, finansowe, rzeczowe i informacyjne) i **wykonywanych z zamiarem osiągnięcia celów organizacji w sposób sprawny i skuteczny.**
(**Griffin R. W.**, *Podstawy zarządzania organizacjami*, Wydawnictwo Naukowe PWN, Warszawa 2005, s. 6)

2. **Działanie polegające na dysponowaniu zasobami**; ponieważ do zasobów najważniejszych należą ludzie, zasobami są pieniądze, a przez nie oddziałuje się na ludzi. Zarządzanie wiąże się z kierowaniem i bardzo często używa się łącznie terminów „organizacja i zarządzanie", „kierowanie i zarządzanie”.
(**Pszczołowski T.**, *Mała encyklopedia prakseologii i teorii organizacji*, Ossolineum, Wrocław - Warszawa - Kraków - Gdańsk 1978, s. 288)

3. **Jest to działanie zmierzające do spowodowania funkcjonowania rzeczy, organizacji lub osób podległych, zgodnie z celami zarządzającego.**
(**Gliński B.**, *Mała encyklopedia ekonomiczna*, Warszawa 1974, s. 929; za T. Pszczołowskim)

źródło: https://mfiles.pl/pl/index.php/Zarz%C4%85dzanie

PROJEKT - definicja

Projekt jest **złożonym** działaniem o **charakterze jednorazowym**, które jest podejmowane dla osiągnięcia z góry określonych celów.

Złożoność wynika z szeregu działań, jakie należy wykonać w określonej kolejności, aby można było osiągnąć cele. Działania często wymagają złożonych zasobów.

Jednorazowość projektu jest jednym z jego kluczowych atrybutów, nie możemy bowiem nazwać projektem działania powtarzalnego.

W przypadku działań powtarzalnych mówimy raczej o operacjach lub procesach.

Do podstawowych atrybutów projektu należy zaliczyć:

- zdefiniowanie w czasie
- niepowtarzalność (jednorazowość)
- złożoność
- celowość

!! Każdy projekt musi mieć szczegółowo określone terminy. Muszą one dotyczyć nie tylko projektu jako całości, ale każdego zadania w nim wykonywanego. Służy temu plan projektu !!

Projektem nie można nazwać pojedynczego zadania lub też zestawu niepowiązanych zadań. Wszystkie zadania wykonywane w projekcie muszą przyczyniać się do realizacji celów.

Nie wolno w projekcie realizować zadań nie przyczyniających się do osiągnięcia celów, ponieważ zmniejsza to efektywność projektu.

Źródło: https://mfiles.pl/pl/index.php/Projekt

KIEROWNIK PROJEKTU - definicja

Kierownik projektu (*PM – Project Manager*) – specjalista w dziedzinie zarządzania projektami. Jest odpowiedzialny za planowanie, realizację i zamykanie projektu. Podstawowym zadaniem kierownika projektu jest zapewnienie osiągnięcia założonych celów projektu, wytworzenie produktu spełniającego określone wymagania jakościowe. PM jest odpowiedzialny za efekt końcowy realizowanego projektu i musi być aktywny podczas wszystkich etapów projektu.
Kierownik projektu może kierować m.in. projektami w budownictwie, projektami informatycznymi, projektami telekomunikacyjnymi, projektami finansowymi.

Kierownik projektu, aby sprawnie zarządzać projektem i posiadanymi zasobami ludzkimi, często pełni wiele funkcji jednocześnie.

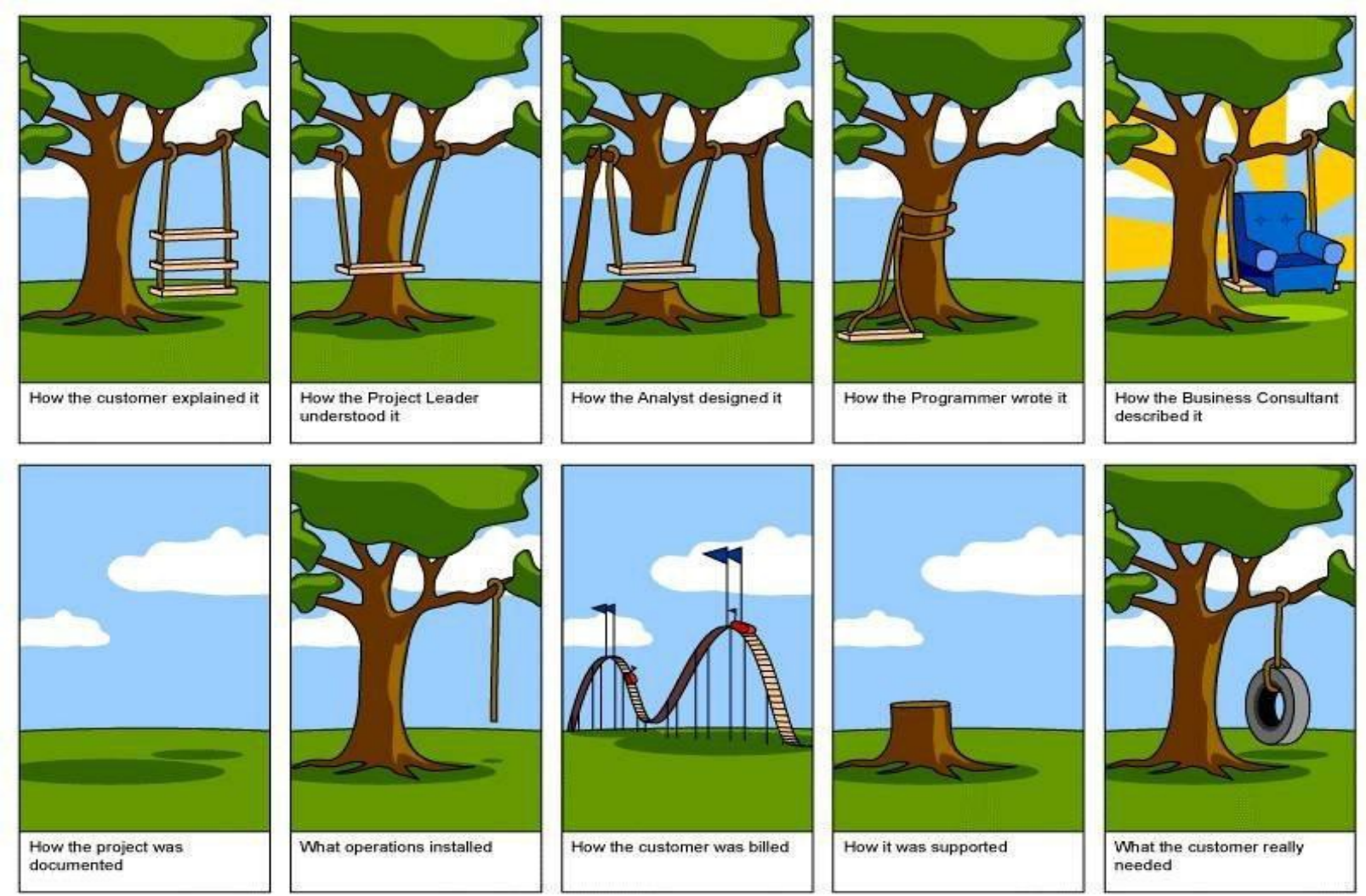
Podstawowe funkcje osoby zarządzającej projektem:

- planowanie
- organizowanie
- motywowanie
- kontrolowanie
- komunikowanie

Źródło: https://pl.wikipedia.org/wiki/Kierownik_projektu

PRAWDA o projekcie IT ☺

- Źródło: znany od ponad 30 lat komiks autorstwa Scott'a Yang'a.
  Osoba potrzebująca opisała produkt następująco: służące do huśtania urządzenie zawieszane na drzewie .
  Zamawiała osoba z działu zamówień.



SPECYFIKA PROJEKTU PROGRAMISTYCZNEGO

Pytanie: kiedy projekt (programistyczny) kończy się sukcesem ?

Trzy proste prawdy:
1.Nie jest możliwe zebranie i opisanie wszystkich wymagań na początku projektu !
2.Jakiekolwiek wymagania zostaną zebrane, na pewno się zmienią !
3.Zawsze będzie więcej do zrobienia, niż pozwalają czas i pieniądze !

Ale kierownictwo klienta potrzebuje i szuka w szacowaniu tego czego tam nie ma:
-Dokładnych przewidywań co do przyszłości: budżet i czas

Również w firmie wytwarzającej trzeba zaplanować: ilu ludzi, czasu, pieniędzy i innych zasobów będzie potrzebnych

Zauważmy, że należy przygotować dokładne oszacowania w sytuacji niepewności !:
-jeszcze niedokładnie opisanego systemu
-często używając nowej albo nieprecyzyjnie określonej technologii
-z zespołem różnych, często nieokreślonych ludzi
-w nieokreślonym nieznanym środowisku biznesowym
-dla projektu, który rozpocznie się w przyszłości - za 3, 6 miesięcy albo za rok

WIEDZA dot. ZARZĄDZANIA PROJEKTAMI

Wiedza jest ogromna.

Zarzadzanie jest kierunkiem studiów, np.:
-Wydział Zarządzania AGH
-Wydział Zarządzania Uniwersytetu Łódzkiego

Na Politechnice Wrocławskiej jest Wydział Informatyki i Zarządzania.

Zarządzanie jest kierunkiem studiów podyplomowych, np.:
-Uniwersytet Ekonomiczny we Wrocławiu,
w programie, np.: - Kompetencje Project Managera
(Microsoft MS Project Certificate)
-WSB (Wyższa Szkoła Bankowa) we Wrocławiu
-WSB we Wrocławiu, też: *Zarządzanie projektem* informatycznym

Dostępnych jest dużo kursów przygotowujących do egzaminów dot. określonego certyfikatu zarzadzania

STANDARDY ZARZĄDZANIA PROJEKTEM

PRINCE2

- Metodyka zarządzania projektami oparta na produktach (dokumentach)
- Opublikowany w 1996 r. (oparty o metodykę opracowaną w latach 70 XX w.
- standard w Wielkiej Brytanii

PMI/PMBOK Guide – ISO 21500

- PMI (Project Management Institute) powstało w 1969 roku w USA.

PMBOK pisuje 9 obszarów zarządzania:

- integralnością projektu
- zakresem
- czasem
- kosztami
- jakością
- zasobami ludzkimi
- komunikacją
- ryzykiem
- zaopatrzeniem

STANDARDY ZARZĄDZANIA PROJEKTEM ..

Standardy zarządzania - projekty programistyczne

SCRUM
- iteracyjna i inkrementalna (= przyrostowa) metodyka prowadzenia projektów
- kolejne "sprinty" dostarczają coraz bardziej dopracowanych wyników
- zespół sam się organizuje
-codzienne Scrum-meeting'i

KANBAN metoda oparta wizualizacji zadań oczekujących na wykonanie i trakcie wykonania.

Programowanie ekstremalne (XP)
-wydajne tworzenie małych i średnich "projektów wysokiego ryzyka", czyli takich, w których nie wiadomo do końca, co się tak naprawdę robi i jak to prawidłowo zrobić.

Inne mniej popularne, np:
- FDD (Feature Driven Development),
- DDD (Domain Driven Design)

4 DEMONY ZRZĄDZANIA PROJEKTEM

4 demony to:
1. CZAS
2. BUDŻET
3. ZAKRES (wymagania funkcjonalne - co system na robić)
4. JAKOŚĆ (wymagania niefunkcjonalne - jak system ma działać)

Uwaga:
Każdym z powyższych czterech demonów trzeba zrządzać.

Przypomnienie:
Zawsze będzie więcej do zrobienia, niż pozwalają czas i pieniądze !

Główny problem:
Jak w powyższej sytuacji postępować ?
Nie okłamywać klienta i siebie – ograniczyć zakres – zrobić to co klientowi jest w rzeczywiście potrzebne w MINIMALNEJ WERSJI.

Zarządzanie interesariuszami

Kierownik projektu ze strony klienta NIE JEST JEDYNĄ osobą u klienta, do której obowiązku należy co najmniej interesowanie się przebiegiem i wynikami projektu.

Społeczność związana z projektem jest zawsze większa, niż nam się wydaje!

Jeżeli będziemy współpracować jedynie z kierownikiem projektu ze strony klienta to może nas czekać coś dziwnego.
Ludzie i grupy, których nigdy nie spotkaliśmy ani nawet widzieliśmy zaczną się pojawiać znikąd i stawiać nam różne, nieznane dotąd wymagania, np:
-Jedna grupa będzie chciała sprawdzać naszą architekturę.
-Inna będzie chciała upewnić się, że spełniamy korporacyjne reguły bezpieczeństwa.
-Jeszcze inna będzie chciała zrobić przegląd dokumentacji.

Każda osobę u klienta, która interesuje się przebiegiem lub wynikami projektu nazywamy interesariuszem.
Interesariuszy należy zidentyfikować i zarządzać nimi !

Zarządzanie interesariuszami ..

Na początku projektu, należy zrobić przegląd społeczności związanej z projektem, namierzyć wszystkich i rozpocząć relacje, zanim wykonawcy (czyli my) będą ich potrzebować.

W ten sposób, gdy przyjdzie ten moment, kiedy oprogramowanie trzeba będzie dostosować do reguł firmy (zasad korporacyjnych) oraz ominąć pułapki organizacyjne interesariusze nie będą obcy i będą w dużo lepszej pozycji aby pomoc.

Podstawą każdej odnoszącej sukcesy organizacji jest zaufanie.
Jednym z elementów zaufania jest poznanie się – interesariuszy trzeba poznać i zapoczątkować dobre stosunki.

Pytanie:
Kto w Państwa projekcie jest interesariuszem ?
Za co on w firmie odpowiada?
Jaki wpływ na projekt może mieć dany interesariusz ?

Zarządzanie interesariuszami..

Przykłady interesariuszy:
-zarząd
-kierownictwo działu, którego dotyczy system
-audytorzy bezpieczeństwa
-architekci
-dział zarzadzania ryzykiem
-specjaliści ds. rekrutacji
-zarządzający zmiana (biznesową i/lub IT)
-administratorzy sieci, serwerów, aplikacji, baz danych
-organizatorzy szkoleń
-dział prawny (sporządzanie/negocjace umowy)
-dział kontroli wewnętrznej
-dział pomocy (Help Desk)
-organizatorzy wdrożeń
-specjaliści ds. dokumentacji

Zwłaszcza w dużych organizacjach, korporacjach (banki, koncerny energetyczne, ..) jest wielu interesariuszy. Np. w banku BZ WBK występują powyżsi interesariusze. Architektów jest 8 w 4 lokalizacjach.

Zarządzanie personelem

PODSTAWOWY ZAKRES OBSZARÓW WIEDZY (KOMPETENCJI) dot. wytwarzania oprogramowania (inżynierii oprogramowania), który powinien posiadać zespół:

1. Zarządzanie wymaganiami dot. oprogramowania (inżynieria wymagań)
2. Projektowanie oprogramowania
3. Wytwarzanie oprogramowania (w tym programowanie)
4. Testowanie oprogramowania
5. Wdrożenie i eksploatacja (utrzymywanie) oprogramowania

---

6. Zarządzanie zmianami i konfiguracjami
7. Zarządzanie wytwarzaniem oprogramowania
8. Procesy w życiu oprogramowania
9. Metody i narzędzia inżynierii oprogramowania
10. Jakość oprogramowania

W związku z powyższym w skład zespołu wytwarzającego oprogramowanie musza wchodzić ludzie posiadający kompetencje dot. wyżej wymienionych zakresów wiedzy.

Zarządzanie personelem..

Wymieniony powyżej zakres obszarów wiedzy to tylko podstawa !

Nie zwiera on wielu innych obszarów wiedzy, np.
-zarzadzania zmianą,
-zarzadzania ryzykiem,
-zarzadzania kryzysem, konfliktami
-zarzadzania wiedzą
-zarzadzania personelem

DOBRY ZESPÓŁ to taki, który może obsłużyć swojego klienta od początku do końca.
Oznacza to posiadanie niezbędnych umiejętności i doświadczenia w zespole, aby być w stanie określić każdą funkcjonalność, jaką może potrzebować klient zespołu i być w stanie w pełni ją zrealizować.

Rekrutują ludzi do zespołu należy szukać takich, którzy czują się komfortowo wykonując wiele różnych zadań. W przypadku programistów oznacza to wyszukiwanie ludzi, którzy maja opanowaną całą technologie (a nie tylko interfejsy czy bazę danych) oraz potrafią zaprojektować architekturę (strukturę) aplikacji oraz potrafią pisać testy jednostkowe oraz dokonywać refaktoryzacji.

Samoorganizacja – zarządzanie personelem

Zasada zwinności – gdzie powstają najlepsze rozwiązania?
Najlepsze architektury, wymagania i projekty (wzory wykonania) pojawiają się w samoorganizujących się zespołach.

źródło zasad zwinności: http://agilemanifesto.org

Samoorganizacja polega na ustaleniu przez zespół:
-Celu jaki należy osiągnąć (to potrzebna jest aprobata klienta)
-Sposobu w jaki cel zostanie osiągnięty
a następnie:
-Determinacji w realizacji działań, które zaplanowano jako prowadzące do osiągnięcia celu.

Zasada zwinności - kogo rekrutować ?
Angażuj do projektów zmotywowanych indywidualistów. Stwórz im dobre środowisko, zaspokajaj ich potrzeby i uwierz im, ze wykonują swoje zadanie.

Zmotywowani idywiadualiści nie mogą być zamknięci – sukces zespołu zależy od tego jak będą się komunikować i jakie relacje (zaufanie !) budować.

Zarządzanie wiedzą

Zasada zwinności – efektywne uczenie się, komunikowanie się.
Pracownicy klienta i twórcy oprogramowania muszą pracować razem każdego dnia w trakcie trwania projektu.

Zespół wytwórczy uczy się od przedstawicieli klienta co należy zrobić
(poprzez dokładne poznanie wymagań funkcjonalnych i niefunkcjonalnych).
Członkowie zespołu uczą się wzajemnie jak spełnić wymagania klienta.

Zasada zwinności – dostarczanie informacji
Najbardziej wydajnym i efektywnym sposobem: 1. dostarczania informacji do zespołu, 2 wymiany informacji wewnątrz zespołu JEST ROZMOWA.

Do najbardziej oczywistych sposobów uczenia się należy wykonanie zaplanowanych działań sprawdzania czy nie popełniono błędów i wyciagnięcie wniosków z popełnionych błędów.

Etapy cyklu uczenia się:
1.Wyznaczenie celu
2.Podjęcie działań na rzecz realizacji tego celu
3.Porównanie wyników tych działań z oryginalnymi założeniami
4.Modyfikacja działań na podstawie dokonanych obserwacji i powrót do kroku 2.

Zarządzanie wiedzą ..

Jak uczyć się szybciej? To proste:
Powinniśmy popełniać (mniejsze) błędy szybciej, a właściwie szybciej je odkrywać.
Natychmiastowe wykorzystywanie zdobytej wiedzy i doświadczeń jest kluczem do skojarzenia nauki z dążeniem do właściwego celu.
Dzięki niezwłocznemu stosowaniu tego czego nauczyliśmy w poprzednim cyklu, możemy najefektywniej korzystać ze skutków tej nauki.

Drugim (oprócz cykli) wymaganym elementem jest komunikacja. Potęguje ona efekt uczenia się przez angażowanie w ten proces wszystkich osób.

Praktyki zwinnego wytwarzania oprogramowania koncentrujące się na komunikacji:
-samoorganizujący się zespół
-zespół pracujący w jednym miejscu
-zespół między funkcjonalny
-programowanie w parach
-radiatory informacji
-dokumentacja wywołująca dyskusje
-poranne spotkania

POZIOMY ZARZĄDZANIA

Zarządzanie operacyjne

Zarządzanie operacyjne to nic innego jak <u>codzienne planowanie i realizacja zadań krótkokresowych.</u>

Charakteryzuje się dużą szczegółowością i krótkim okresem odniesienia Zapewnia warunki realizacji założeń strategicznych i umożliwia podejmowanie decyzji wykonawczych na różnych szczeblach organizacyjnych .

Zarządzanie taktyczne
Koncentrują na ogólnym sposobie realizacji (operacjonalizacji) działań niezbędnych do osiągnięcia celów strategicznych.
Dot. m in. rozwiązywania problemów: doskonalenie przepływu informacji, rozwiązywanie konfliktów, zmiany logistyczne.
Ustalenia dot. średniego horyzontu czasowego na średnim poziomie i przez kierowników średniego szczebla.

Zarządzanie strategiczne
Określa główne cele i metody działania w długim horyzoncie czasowym

ZARZĄDZANIE STRATEGICZNE

Zarządzanie strategiczne oznacza, zależnie od kondycji organizacji następujące działania lub ich kombinację:

- Ustalenie długookresowych celów i sposobów ich osiągania
- Zarządzenie alokacją środków na uprzednio właściwie określone cele
- Wskazanie osób decyzyjnych oraz nakreślenie struktury organizacyjnej
- Nakierowanie na analizy, techniki planowania strategicznego, dalszy rozwój oraz innowacyjność mające zagwarantować ustaloną pozycję na rynku
- Określenie ogólnych zasad postępowania i funkcjonowania przedsiębiorstwa
- Stworzenie mechanizmów identyfikacji nowych wyzwań

źródło: http://ahprofit.pl/zarzadzanie-przedsiebiorstwem/

Zarządzanie strategiczne

Przykłady decyzji strategicznych:

-Uczestnictwo w danym przetargu

-Zgoda na prace zdalne (np. z mieszkania)

-Rozpoczęcie produkcji oprogramowania określonego rodzaju

-Kupno innej firmy

-Sprzedaż działu/firmy

-Otwarcie działalności do innego miasta

-Decyzja o utworzeniu offShore Center

-Decyzja o rozpoczęciu produkcji w Indiach, Pakistanie, Chinach

Zarządzanie operacyjne

Zarządzanie operacyjne – wg osoby odpowiedzialnej za założenie we Wrocławiu Capgemini OffShore Center
Zadawanie pytań (odmiana pojęć przez przypadki) dot. osiągnięcia celu projektu i znajdowanie na nie odpowiedzi. Ważne również w samoorganizacji.

Ćwiczenie:
Dla każdego z przypadku zadaj pytanie dot. projektu. Odpowiedz na nie.

1.mianownik – kto?, co?

2.dopełniacz – kogo?, czego?

3.celownik - komu?, czemu?

4.biernik – kogo?, co?

5.narzędnik – z kim?, z czym?

6.miejscownik – o kim?, o czym?, gdzie?, kiedy?

7.wołacz – o !

Zarządzanie taktyczne

Problemy i sposoby ich rozwiązywania zaczerpnięto z książki
Amr Elssamadisy - AGILE wzorce wdrażania praktyk zwinnych, wydawnictwo Helion 2010.

Problemy biznesowe
Mają bezpośredni związek z odczuciami klienta odkrywającego, że zakupione oprogramowanie nie spełnia jego oczekiwań.

Typowe problemy biznesowe

1. Jakość produktu przekazanego klientowi jest nie do przyjęcia
2. Dostarczanie klientowi nowych funkcji trwa zbyt długo
3. Zaimplementowane funkcje nie są wykorzystywane przez klienta
4. Oprogramowanie okazało się nieprzydatne dla klienta
5. Budowa oprogramowania jest zbyt droga
6. My kontra oni
7. Klient żąda od nas zbyt dużo

Zarządzanie taktyczne

Problemy biznesowe cd …

Analiza problemu 3 - zaimplementowane funkcje nie są wykorzystywane.

Powody wystąpienia

1.W czasie formułowania wymagań klienci nie wiedzieli czego potrzebują.

2.W role klienta wcielił się dział marketingu wytwórcy.

3.Niektóre funkcje wykorzystywane są dużo rzadziej niż przewidywano.

Niezgodność priorytetów wytwórcy i klienta.

4.Programiści dostarczają funkcje, które wg nich są potrzebne.

5.Wymagania są zmieniane

Rozwiązanie problemu – zwiększenie użyteczności !

W jaki sposób (najbardziej skuteczne praktyki)

W tym przypadku - włączenie klienta do zespołu (customer part of team),

A następnie:

- nadanie przez reprezentanta klienta priorytetów wymagań
-prezentacja wersji demonstracyjnej przedstawicielom klienta
-testowanie funkcjonalne wykonywane przez reprezentanta klienta

Zarządzanie taktyczne

Problemy związane z procesami należy traktować jako symptomy występowania wewnętrznych trudności w ramach zespołu wytwórcy

Typowe problemy związane z procesami wytwarzania oprogramowania

1. Wiara w bezpośrednie i regularne sugestie klienta jest nieuzasadniona
2. Zarząd jest zaskoczony – nie widzi co dzieje się w projekcie
3. Niewystarczające zasoby – specjaliści są dzielni między zespołami
4. „Ruchome” tzn przekraczające terminy projekty
5. Setki (lub tysiące) błędów
6. Potrzeba fazy „hartowania” na końcu cyklu wytwarzania
7. Integracja ma miejsce zbyt rzadko (ponieważ jest kłopotliwa)

Występowanie problemów dotyczących procesów skłania do ich doskonalenia. Rozwiązywanie tych problemów jest jednak mniej ważne jak podnoszenie walorów biznesowych poprzez rozwiązywanie wymienionych poprzednio problemów biznesowych.

Zarządzanie taktyczne

Problemy związane z procesami
Klient? Jaki klient ?
Wiara w bezpośrednie i regularne sugestie klienta jest nieuzasadniona !

Powody - scenariusze

1.Firma nie ma kontaktu z rzeczywistymi klientami. Dział marketingu pełni role pseudoklienta. Marketingowcy pracują niezależnie od zespołu wytwarzającego oprogramowanie.

2. Klientami są członkowie zarządu danej firmy. Oni nie maja dużo czasu. Spotykają się tylko chwile. Z tych spotkań sporządzane są notatki.

W obu tych scenariuszach ilość informacji docierających do wytwórcy od klienta jest niewielka.

Problem związany z procesem być powodem kilku problemów biznesowych:

1. Istnienie nieużywanych funkcji, 2. Nieprzydatność oprogramowania dla klienta, 3. trudności (użyteczność) pracy z produktem.

Zarządzanie taktyczne

Rozwiązanie problemu związanego z procesami
Klient? Jaki klient ?
!! Wiara w bezpośrednie i regularne sugestie klienta jest nieuzasadniona !!

Bezpośrednie i regularne, wystarczająco długie spotkania z klientem są niemożliwe. W efekcie niemożliwe jest uzyskiwanie od klienta jego opinii i sugestii

Ograniczenie negatywnych skutków stanu rzeczy – zmniejszenie błędów komunikacyjnych.
-Sporządzanie przez klienta (lub przez dział marketingu wytwórcy) dokumentu opisującego wymagania. Zatwierdzanie projektów (ekranów, raportów)
(wada: duża pracochłonność !).
-Praca przyrostowa z iteracjami kończącymi się sporządzeniem wersji demonstracyjnej. Prowadzenie listy zaległych zadań (product backlog).

Aby uniknąć niepotrzebnego zaskakiwania kierownictwa organizacji wytwórcy/zamawiającego powinno się robić 2 rzeczy:
-budować aplikacje przyrostowo, wysyłanie do akceptacji,
-informować kierownictwo o rzeczywistych postępach prac.

INŻYNIERIA OPROGRAMOWANIA

Dziękuję za uwagę