Budowa i integracja systemów informacyjnych

Wykład 1:
Przedmiot inżynierii oprogramowania

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

subieta@pjwstk.edu.pl

Literatura

Materiał pochodzi z następujących pozycji:

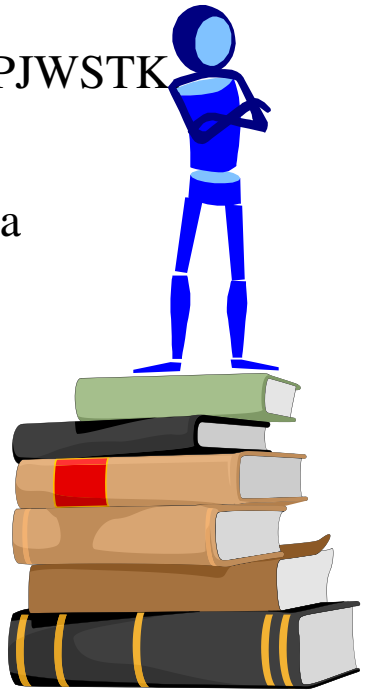
A.Jaszkiewicz. **Inżynieria Oprogramowania.** Wydawnictwo HELION 1997

W.Dąbrowski, K.Subieta**. Podstawy inżynierii oprogramowania.** Wydawnictwo PJWSTK, Warszawa, ISBN 83-89244-46-2, 2005 (314 stron)

K.Subieta. **Wprowadzenie do inżynierii oprogramowania.** Wydawnictwo PJWSTK, Warszawa 2002, ISBN 83-89244-00-4 (218 stron)

K.Subieta. **Słownik terminów z zakresu obiektowości**. Akademicka Oficyna Wydawnicza 1999

C. Mazza, et all. **Software Engineering Guides.** Prentice Hall Europe 1996



Oraz z niemożliwej do ogarnięcia liczby stron internetowych.

Wykłady będą udostępnione w formie slajdów (na życzenie).

subieta@pjwstk.edu.pl

Plan wykładu

- Przedmiot i zagadnienia inżynierii oprogramowania
- Kryzys oprogramowania
- Złożoność projektu oprogramowania
- Modelowanie pojęciowe
- Pojęcie metodyki; metodyki i notacje
- Modele cyklu życiowego oprogramowania

Przedmiot inżynierii oprogramowania

Inżynieria oprogramowania jest wiedzą techniczną dotycząca wszystkich faz cyklu życia oprogramowania. Traktuje oprogramowanie jako produkt, który ma spełniać potrzeby techniczne, ekonomiczne lub społeczne.

Dobre oprogramowanie powinno być:

- zgodne z wymaganiami użytkownika,
- niezawodne,
- efektywne,
- łatwe w konserwacji,
- ergonomiczne.

Produkcja oprogramowania jest procesem składającym się z wielu faz. Kodowanie (pisanie programów) jest tylko jedną z nich, niekoniecznie najważniejszą.

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. Przyczyna …? … dość oczywista.

Zagadnienia inżynierii oprogramowania

- 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.

Kryzys oprogramowania (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.

Kryzys oprogramowania (2)

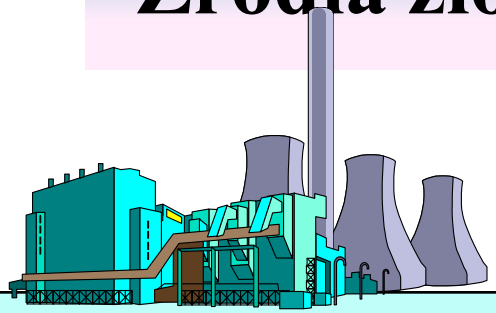
- Frustracje projektantów oprogramowania i programistów wynikające ze zbyt szybkiego postępu w zakresie języków, narzędzi i metod oraz uciążliwości i długotrwałości procesów produkcji, utrzymania i pielęgnacji oprogramowania.
- Uzależnienie organizacji od systemów komputerowych i przyjętych technologii 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 kryzysem oprogramowania

- 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 kryzysu oprogramowania jest złożoność produktów informatyki i procesów ich wytwarzania.

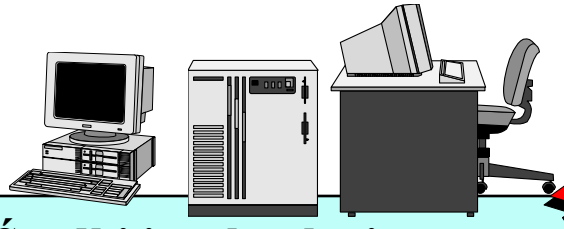
Źródła złożoności projektu oprogramowania



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, standardów, 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.

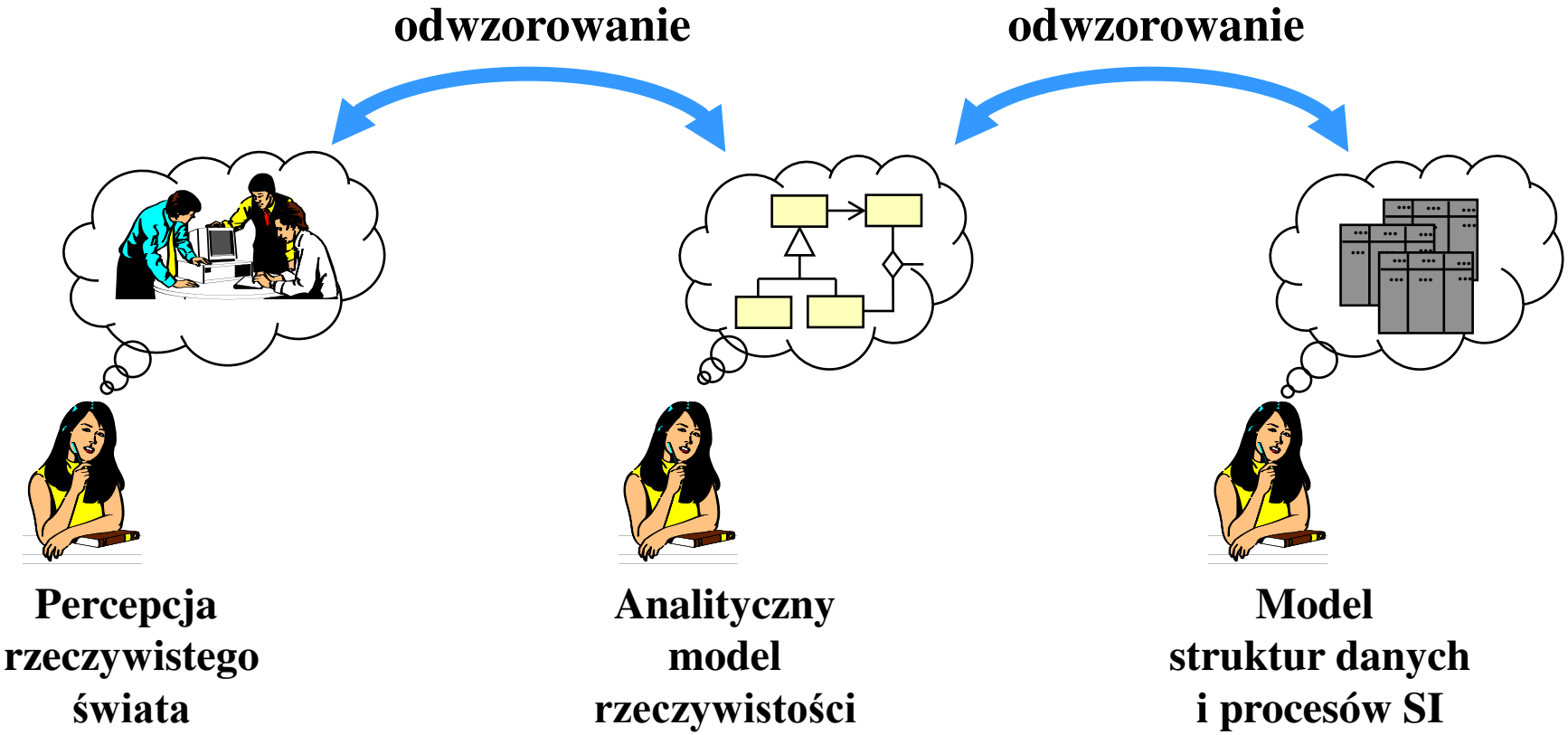
Modelowanie pojęciowe

- 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.

Perspektywy w modelowaniu pojęciowym



Trwałą tendencją w rozwoju metod i narzędzi projektowania oraz konstrukcji SI jest dążenie do minimalizacji luki pomiędzy myśleniem o rzeczywistym problemie a myśleniem o danych i procesach zachodzących na danych.

Co to jest 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 następnej fazy,
- notacje, których należy używać,
- dokumentację powstającą w każdej z faz.

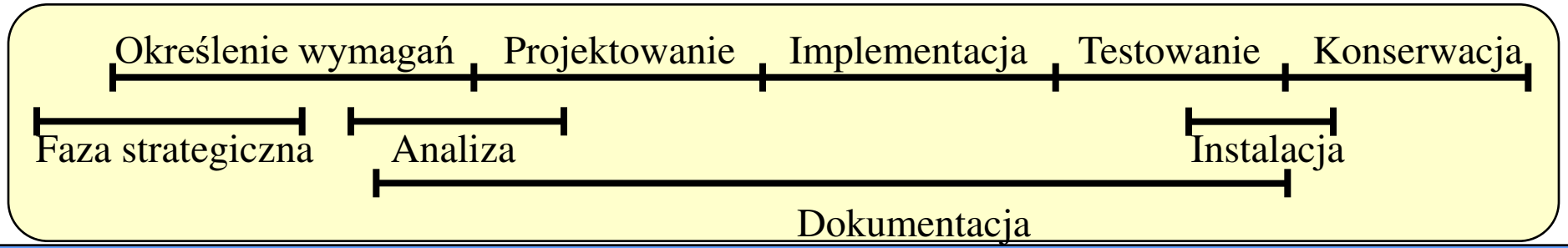
Cykl życiowy oprogramowania

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

Modele cyklu życia oprogramowania

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

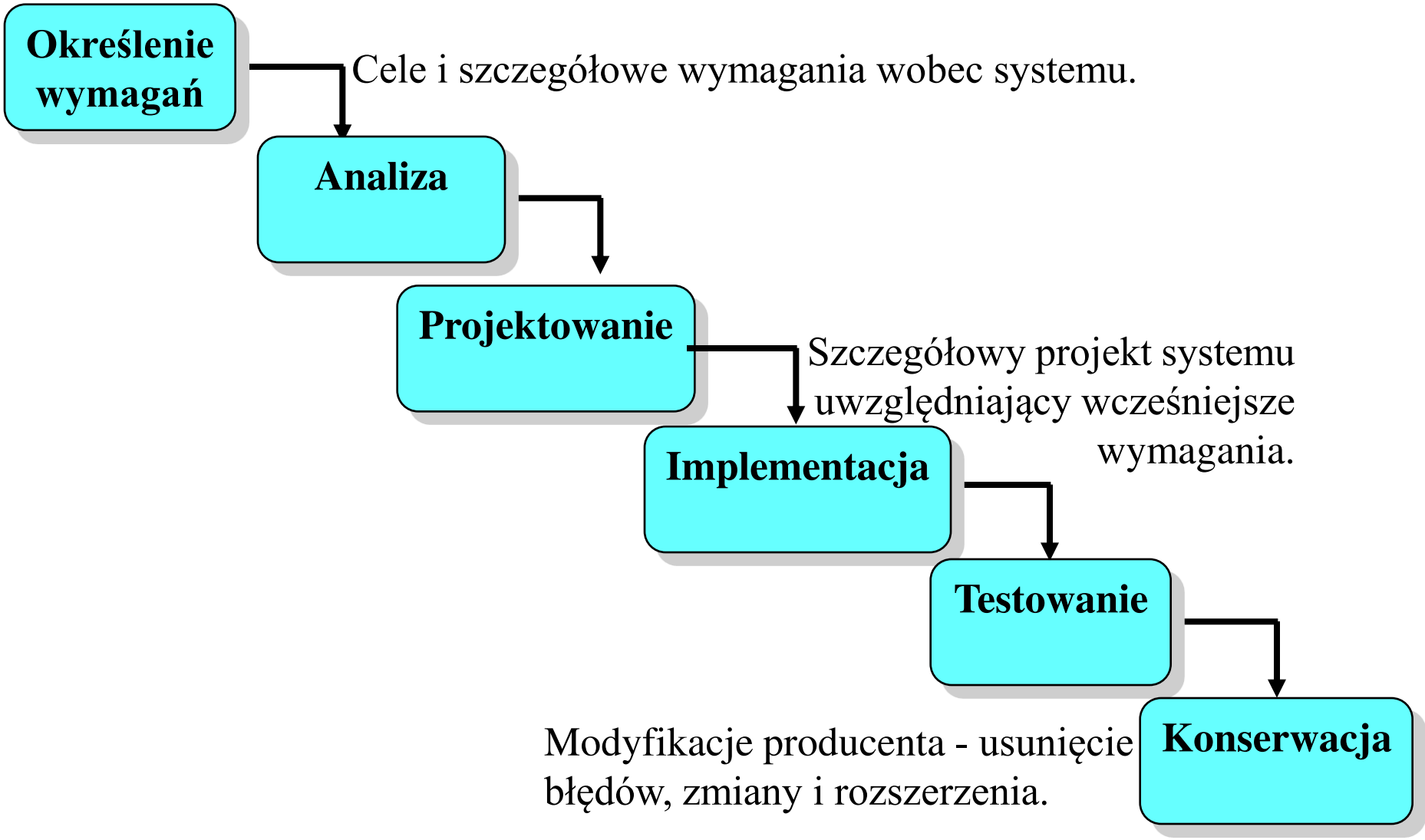
Tego rodzaju modeli (oraz ich mutacji) jest dość dużo.



Model kaskadowy (wodospadowy)

waterfall model

Założenie: nie ma powrotu do poprzednich faz

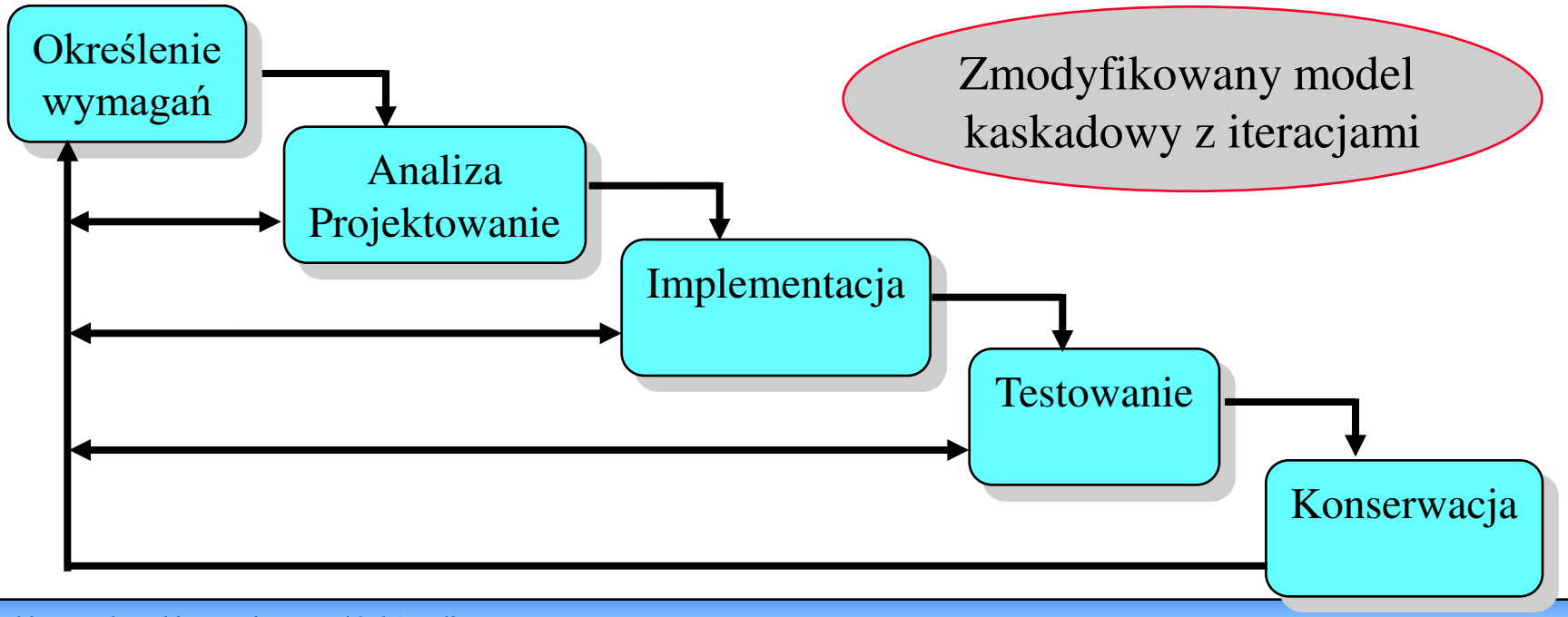


Ocena modelu kaskadowego

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

- Narzucenie twórcom oprogramowania ścisłej kolejności wykonywania prac
- Wysoki koszt błędów popełnionych we wczesnych fazach
- Długa przerwa w kontaktach z klientem

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



Realizacja kierowana dokumentami

- Przyjęty przez armią amerykańską odmiana modelu kaskadowego.
- Każda faza kończy się sporządzeniem szeregu dokumentów, w których opisuje się wyniki danej fazy.
- Łatwe planowanie, harmonogramowanie oraz monitorowanie przedsięwzięcia.
- Dodatkowa zaleta: (teoretyczna) możliwość realizacji dalszych faz przez inną firmę.

Wady

- Duży nakład pracy na opracowanie dokumentów zgodnych ze standardem (DOD STD 2167) - ponad 50% całkowitych nakładów.
- Przerwy w realizacji niezbędne dla weryfikacji dokumentów przez klienta.
- „Tacit knowledge” – wiedza nieudokumentowana.
- W Polsce nie stosowana.

Model spiralny

spiral model

Planowanie: Ustalenie celów produkcji kolejnej wersji systemu

Analiza ryzyka
(ew. budowa prototypu)

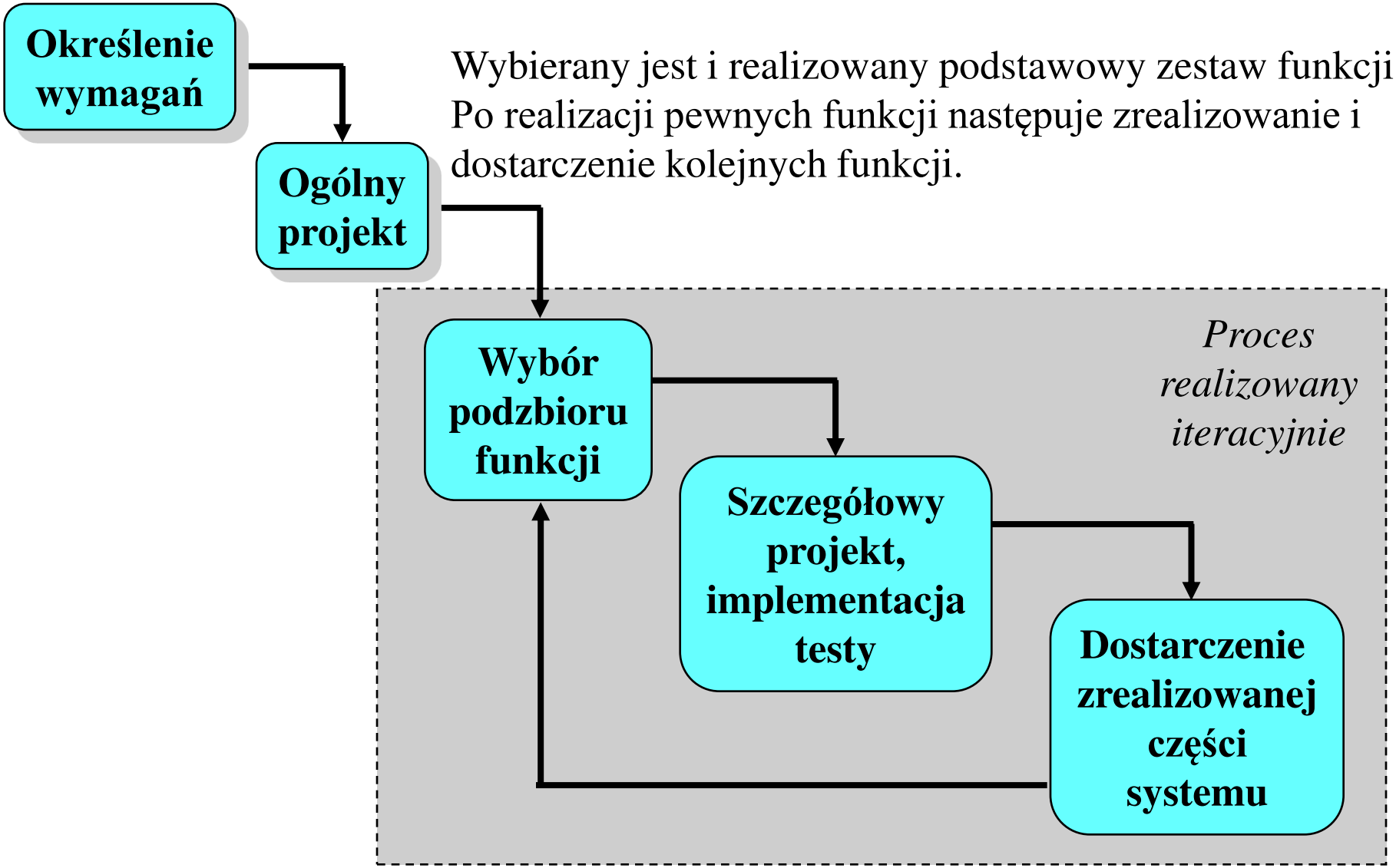
Konstrukcja
(model kaskadowy)

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

Istnieje wiele wariantów tego modelu.

Realizacja przyrostowa (odmiana modelu spiralnego)

incremental development



Prototypowanie

prototyping

Sposób na uniknięcie zbyt wysokich kosztów błędów popełnionych w fazie określania wymagań. Zalecany w przypadku, gdy określenie początkowych wymagań jest stosunkowo łatwe.

Fazy

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

Cele

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

Zalety

- możliwość demonstracji pracującej wersji systemu
- możliwość szkoleń zanim zbudowany zostanie pełny system

Metody prototypowania

- **Niepełna realizacja**: objęcie tylko części funkcji
- **Języki wysokiego poziomu**: Smalltalk, Lisp, Prolog, 4GL, ...
- **Wykorzystanie gotowych komponentów**
- **Generatory interfejsu użytkownika**: wykonywany jest wyłącznie interfejs, wnętrze systemu jest "podróbką".
- **Szybkie programowanie** (*quick-and-dirty*): normalne programowanie, ale bez zwracania uwagi na niektóre jego elementy, np. zaniechanie testowania

Dość często następuje ewolucyjne przejście od prototypu do końcowego systemu. 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.

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

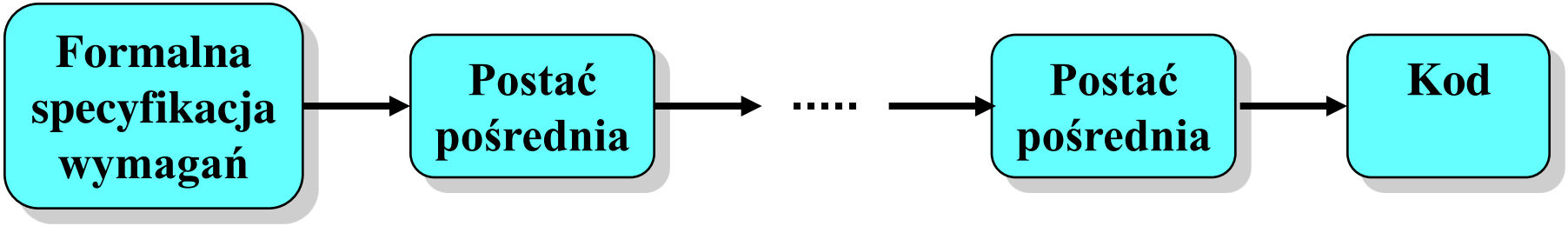
Wady

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

Formalne transformacje

formal transformations

Wymagania na system są formułowane w pewnym formalnym języku, następnie poddawane są kolejnym transformacjom, aż do uzyskania działającego kodu.



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 (lub wręcz *jakiekolwiek*) 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 wyłącznie *wspomagać* pewne szczegółowe tematy (tak jak w biologii, ekonomii i innych dziedzinach).

Matematyka?

Dość często w związku z negacją zastosowań matematyki w inżynierii oprogramowania negowana jest w ogóle rola matematyki w edukacji informatycznej. Dotyczy to m.in. wielu uczelni w USA. Moim zdaniem, jest to pogląd **błędny**.

Matematyka jest niezbędnym elementem informatycznego wykształcenia, przede wszystkim ze względu na wyrabiania nawyków precyzyjnego myślenia.

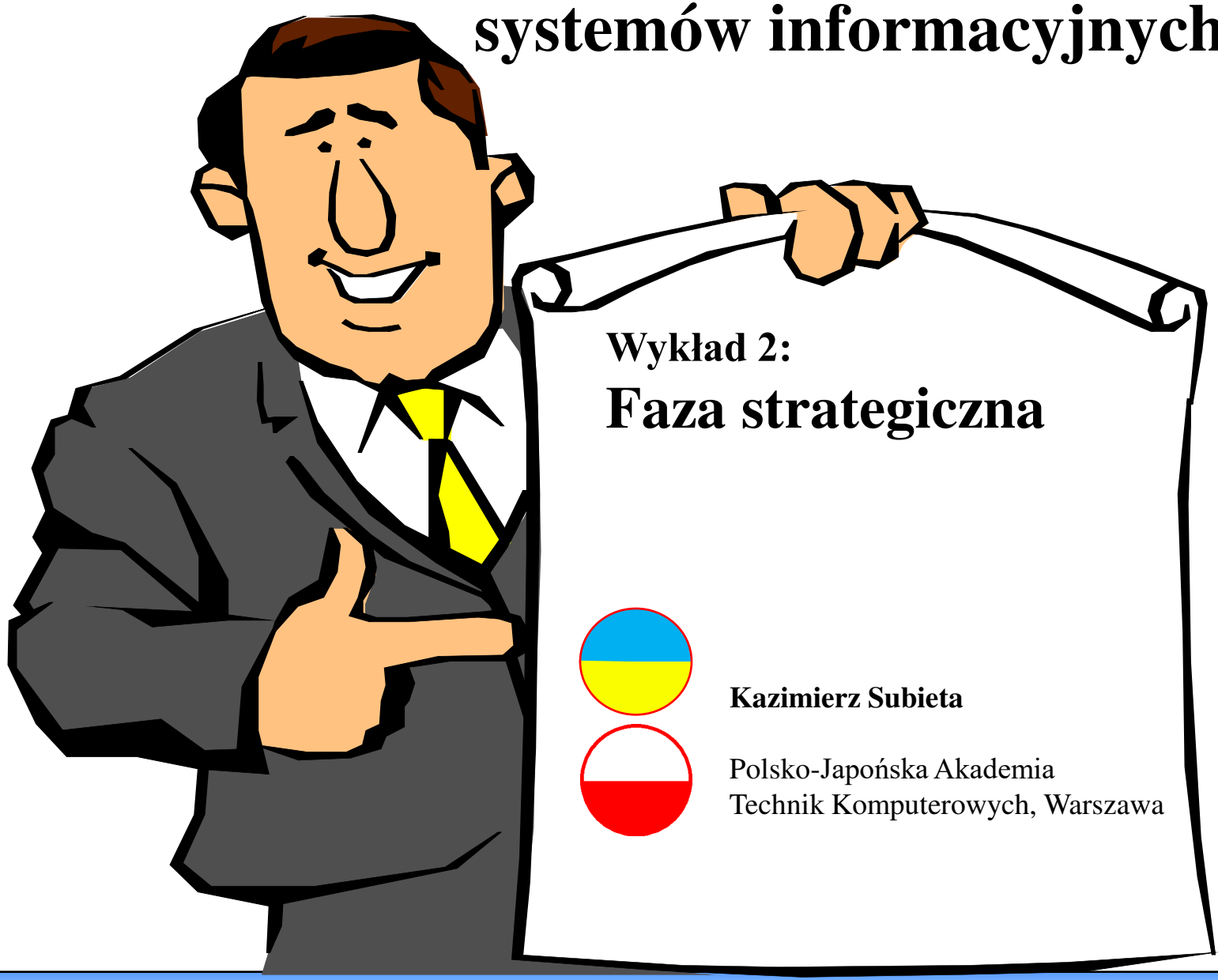
Wg mnie, najważniejszymi działami matematyki niezbędnymi dla informatyka są:

- Rachunek prawdopodobieństwa i statystyka
- Logika matematyczna i dowodzenie twierdzeń
- Teoria gramatyk i automatów skończonych
- Teoria zbiorów, relacji i funkcji
- Teoria równań stało-punktowych i (dla wytrwałych) semantyka denotacyjna

Inne działy matematyki mają mniejsze (marginalne) znaczenie. Zaliczę do nich:

- Rachunek różniczkowy i całkowy
- Algebra liniowa
- Teoria liczb
- …

Budowa i integracja systemów informacyjnych



Wykład 2:
Faza strategiczna

Kazimierz Subieta

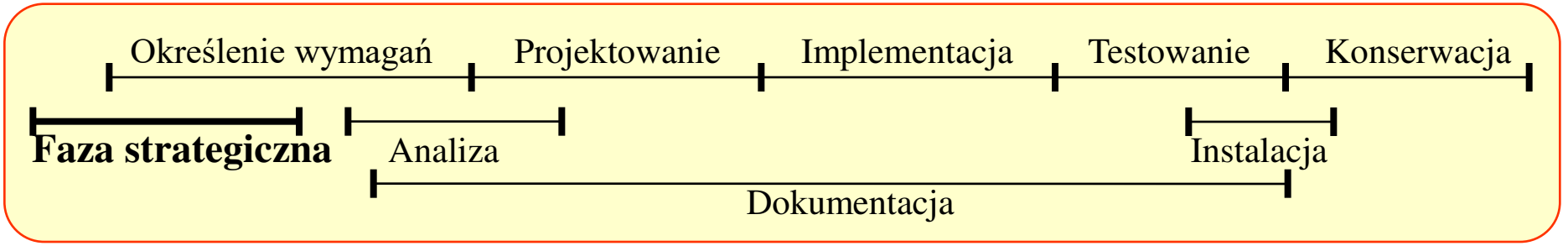
Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Plan wykładu

- Czynności w fazie strategicznej
- Współpraca z klientem
- Zakres i kontekst przedsięwzięcia
- Decyzje strategiczne
- Harmonogram przedsięwzięcia
- Ocena rozwiązań
- Niepewność i ryzyko
- Szacowanie kosztu oprogramowania

Faza strategiczna (studium osiągalności)

strategy phase, feasibility study



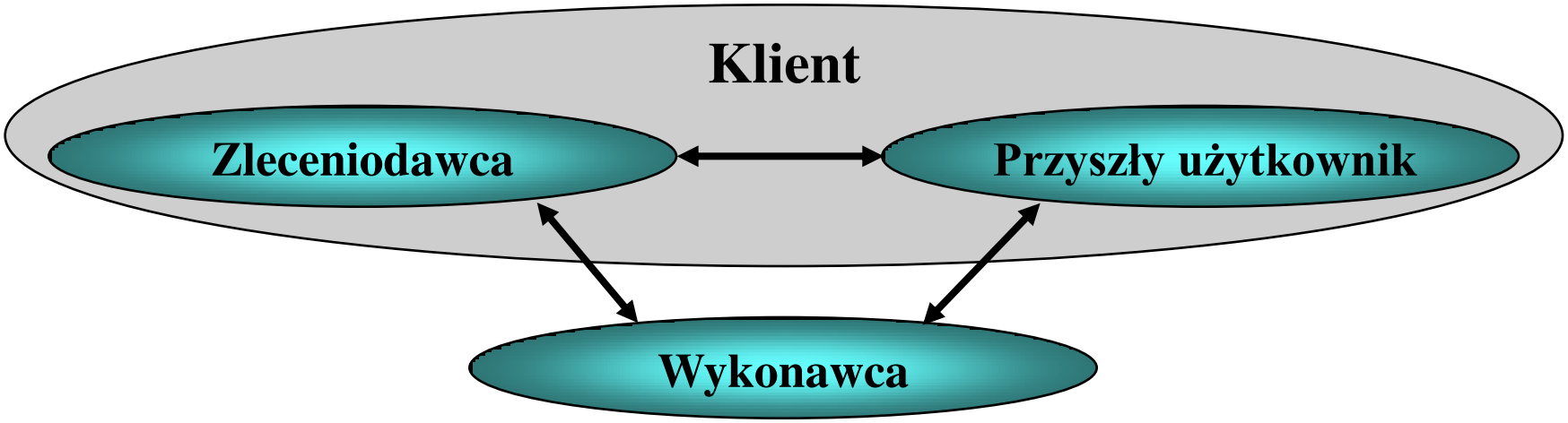
Faza strategiczna jest wykonywana zanim podejmowana jest decyzja o realizacji przedsięwzięcia.

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

Czynności w fazie strategicznej

- Dokonanie serii rozmów (wywiadów) z przedstawicielami klienta
- Określenie celów przedsięwzięcia z punktu widzenia klienta
- Określenie zakresu oraz kontekstu przedsięwzięcia
- Ogólne określenie wymagań, wykonanie zgrubnej analizy i projektu systemu
- Propozycja kilku możliwych rozwiązań (sposobów realizacji systemu)
- Oszacowanie kosztów oprogramowania
- Analiza rozwiązań
- Prezentacja wyników fazy strategicznej przedstawicielom klienta oraz korekta wyników
- Określenie wstępnego harmonogramu przedsięwzięcia oraz struktury zespołu realizatorów
- Określenie standardów, zgodnie z którymi realizowane będzie przedsięwzięcie

Faza strategiczna - współpraca z klientem



Po stronie klienta warto wyróżnić zleceniodawcę i przyszłych użytkowników. Starać się uwzględnić kryteria obydwu stron, ale należy pamiętać, że system będzie głównie oceniany przez przyszłych użytkowników.

- Ważnym elementem fazy strategicznej 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 rachunkow zajmuje się m.in. przygotowaniem formularzy zeznań podatkowych (PIT-ów) dotyczących podatku dochodowego dla indywidualnych podatników.

Ponieważ liczba klientów tegor 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:

➔ przyśpieszenie obsługi klientów
➔ zmniejszenie ryzyka popełnienia błędów

Przykład: system informacji geograficznej - SIG

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

Miałby to być system łączący w sobie mozliwość 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

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.

Cele przedsięwzięcia z punktu widzenia klienta:

- ➔ zwiększenie wydajności pracy wydziału poprzez szybszą i efektywniejszą realizację zleceń,
- ➔ zmniejszenie opóźnień w realizowaniu zleceń
- ➔ 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.

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.

Przykłady zakresu/kontekstu przedsięwzięcia

Program podatkowy	**Zakresem** przedsięwzięcia jest działalność jednej firmy rachunkowej, która może mieć dowolną liczbę klientów. Nie jest określone, czy system ma drukować wypełniony PIT, czy tylko dostarczać dane. 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 zewnetrznymi** są: system komputerowy działu marketingu, osoba definiująca technologiczne możliwości wydziału, kadra kierownicza.

Decyzje strategiczne

- Wybór modelu, zgodnie z którym będzie realizowane przedsięwzięcie
- 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

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 strategiczna jest nieodłączną częścią cyklu produkcji oprogramowania, wobec czego nie powinna być wykonywana na koszt i ryzyko producenta oprogramowania

Studium osiągalności

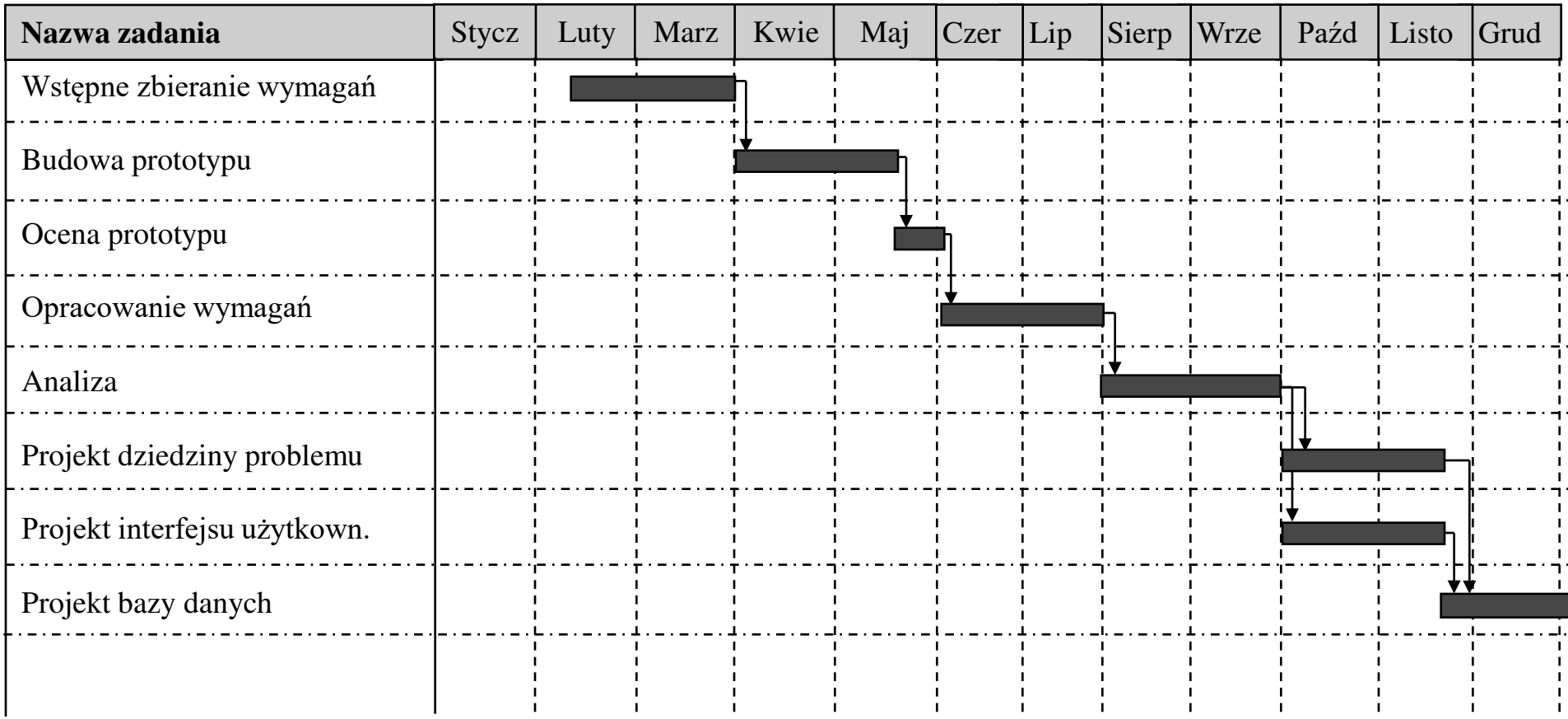
feasibility study

- 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 dla poszczególnych faz i zadań.

Diagram Gantta



Ocena rozwiązań

W fazie strategicznej często rozważa się kilka rozwiązań, z powodów wielości celów przedsięwzięcia (czyli kryteriów oceny) lub niepewności (niemożliwości precyzyjnej oceny spodziewanych rezultatów).

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

Oszacowanie wartości podanych w tabeli może być trudnym problemem.

Wybór rozwiązania

Usunięcie rozwiązań zdominowanych, tj. gorszych wg wszystkich kryteriów (lub 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

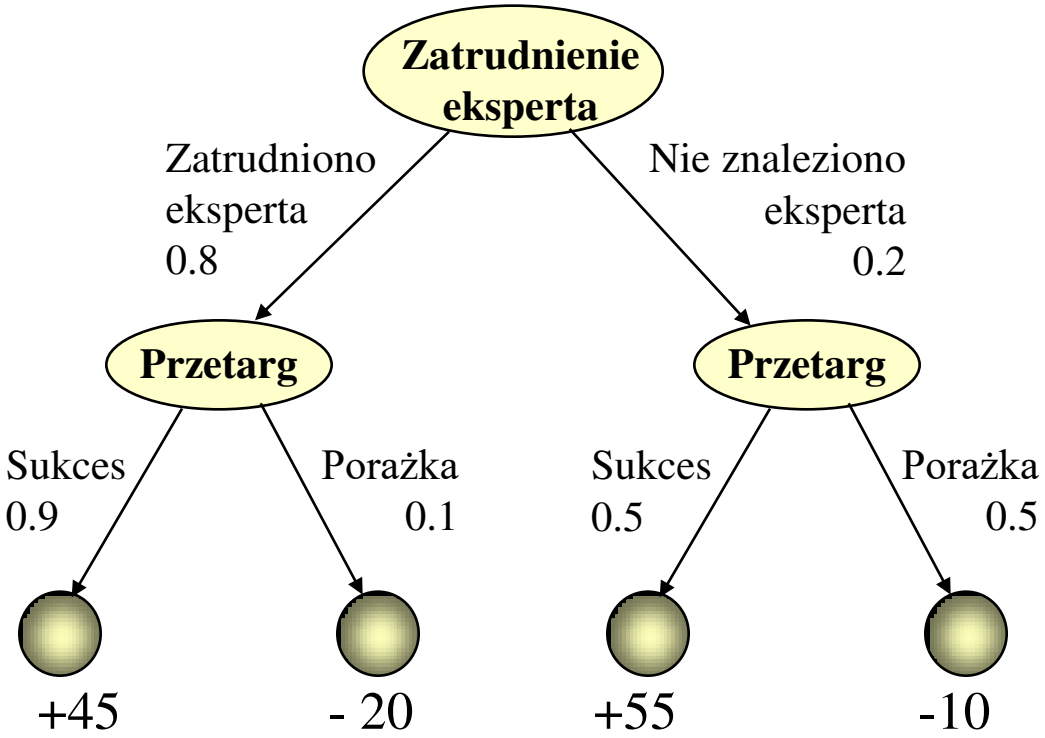
Rozwiązanie	**A**	**B**	**C**	**Waga**
Koszt (tys. zł)	0.58	1	0	3
Czas (miesiące)	0.5	1	0	2
Niezawodność (błędy/tydzień)	1	0.5	0	3
Ponowne użycie (%)	1	1	0	1
Przenośność (%)	1	0.75	0	1
Wydajność(transakcje/sek)	0	0,62	1	1.5
Łączna ocena	**7.74**	**9.17**	**1.5**	

Drzewa ryzyka

Wierzchołki drzewa odpowiadają sytuacjom, w których mogą zajść pewne zdarzenia.
Krawędzie oznaczają przejścia do nowych sytuacji.
Krawędziom są przypisane prawdopodobieństwa.
Każdy scenariusz zdarzeń (liść w drzewie) jest związany z kosztem.

Przykład

Firma chce przystąpić do przetargu. Przygotowanie oferty przetargowej jest kosztowne. Firma może przetarg wygrać lub przegrać. Zatrudnienie dodatkowego eksperta zwiększa szansę firmy. Co robić?



Zysk

Oczekiwany zysk

45*0.8*0.9 + (-20)*0.8*0.1 + 55*0.2*0.5 + (-10)*0.2*0.5 = 35.3

Niepewność szacunków

Każda wielkość liczbowa odnosząca się do przyszłości jest wartością niepewną.

- W większości przypadków nie istnieją żadne wiarygodne metody (np. pomiary) pozwalające je uzyskać
- Dotyczy to szczególnie tzw. „prawdopodobieństw subiektywnych”
- Wielu autorów wskazuje na ewentualność powołania „ekspertów”
  - Zwykle w danej firmie mamy doświadczone osoby, które potrafią przewidzieć różne sprawy z dość dużą dokładnością
  - Powołanie zewnętrznych ekspertów jest jednak zazwyczaj nierealne
- Wszelkie wzory matematyczne, różne metody formalne, w tym oparte o „sztuczną inteligencję” są zazwyczaj bezwartościowe

- **Dlaczego?**

Prawo GIGO

- „Garbage In – Garbage Out”
- Prawo to głosi, że nie da się uzyskać wiarygodnej informacji z szumu informacyjnego
  - Jeżeli na wejściu jest szum informacyjny (losowe sygnały), to nie da się z tego szumu wyabstrahować wiarygodnej informacji niezależnie od metody czy urządzenia przetwarzającego – procedura, reguła, sztuczna inteligencja, metody matematyczne, itp.
  - Informacyjne „perpetum mobile” (zwane też demonem Maxwela 2-go rodzaju - Lem) nie może istnieć
- Wiele metod nie tylko w inżynierii oprogramowania próbuje zaprzeczyć temu prawu
- Do takim metod należy **większość** metod szacowania kosztów, czasu i innych parametrów oprogramowania

Szacowanie kosztu oprogramowania

Obiektywny koszt oprogramowania nie istnieje.

Dlaczego?

- Oszacowania amortyzacji poprzednich inwestycji
- Przyszłe inwestycje (rozwój naszej firmy)
- Oczekiwania płacowe w związku z nowym projektem
- Oczekiwania dotyczące zysku naszej firmy
- Niepewność co do kosztów nowych zatrudnień, zakupów, itd.
- …

Szacowanie kosztów przeprowadza się dla każdego z alternatywnych rozwiązań i dotyczy w zasadzie wyłącznie bieżących kosztów ponoszonych przez wykonawcę.

Na ten koszt składają się następujące główne czynniki:

- koszt sprzętu będącego częścią tworzonego systemu
- koszt wyjazdów i szkoleń
- koszt zakupu narzędzi
- nakład pracy

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

Techniki oszacowania nakładów pracy

- **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.

Algorytmiczne 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 szacowania kosztów 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 określanych jako podstawowa, pośrednia i detaliczna.

- **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**
- **Metody oszacowania pracochłonności dokumentacji**
- **Metody oszacowania obciążenia sieci**
- **....**

Podsumowanie: kluczowe czynniki sukcesu

- **Szybkość pracy**. Szczególnie w przypadku firm realizujących oprogramowanie na zamówienie, opóźnienia w przeprowadzeniu fazy strategicznej 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.

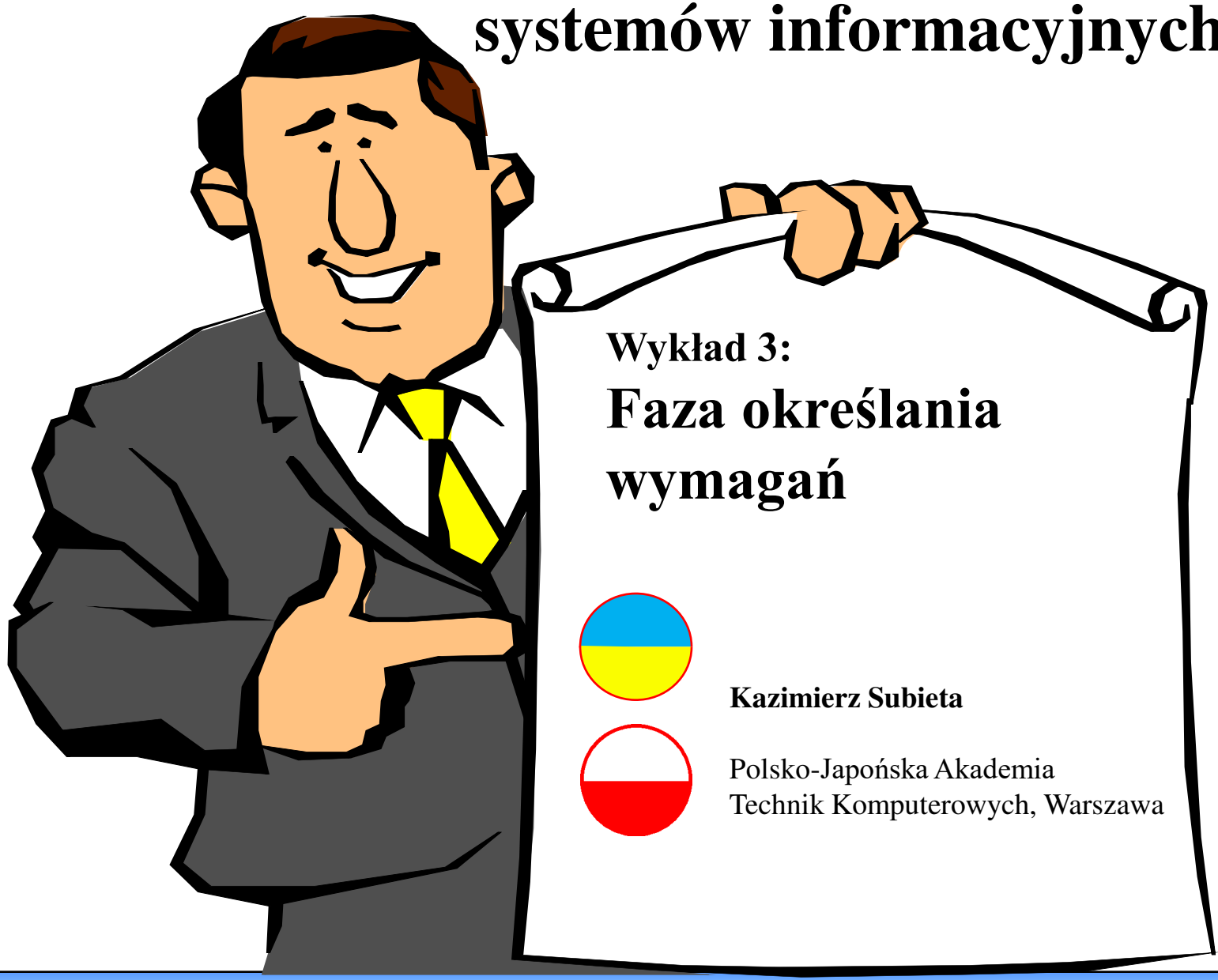
- **Zaangażowanie kluczowych osób ze strony klienta**. Brak akceptacji dla sposobu realizacji przedsięwzięcia ze strony kluczowych osób po stronie klienta może uniemożliwić jego przyszły sukces.

- **Uchwycenie (ogólne) całości systemu**. Podstawowym błędem popełnianym w fazie strategicznej 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.

Podstawowe rezultaty fazy strategicznej

- Udostępniamy klientowi raport, który 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 kosztów
  - 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 standardów.
- Harmonogram fazy analizy

Budowa i integracja systemów informacyjnych



Wykład 3:
Faza określania wymagań

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Plan wykładu

- Trudność określenia wymagań
- Poziomy ogólności opisu wymagań
- Jakość opisu wymagań
- Metody rozpoznania wymagań
- Wymagania funkcjonalne
- Porządkowanie wymagań
- Wymagania niefunkcjonalne
- Czynniki uwzględniane przy konstruowaniu wymagań
- Dokument wymagań

Wymagania określone i nieokreślone

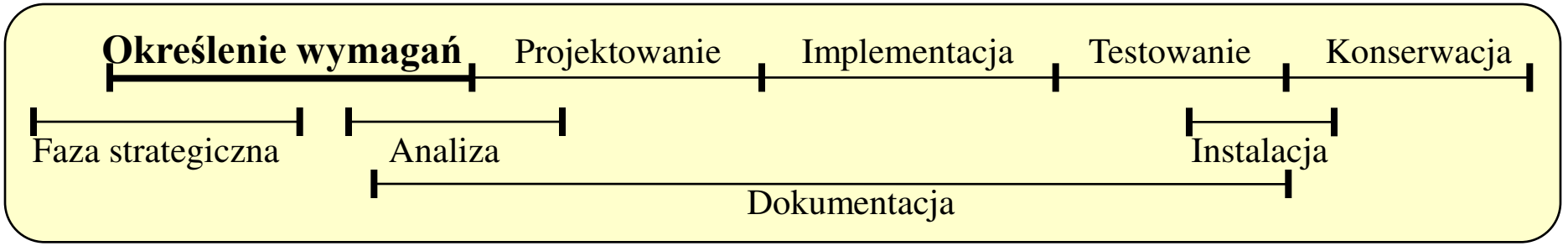
- Spora grupa projektów programistycznych nie ma określonych wymagań z powodu ich innowacyjności
- Są to w szczególności projekty, których założeniem jest nowość cywilizacyjna
- Klient takich projektów nie jest świadomy tego, że budujemy oprogramowanie dla niego, wobec czego niczego sensownego nie może wnieść
- Przykładem są przeglądarki internetowe, kompilatory nowych języków, biblioteki cyfrowe, nowe rozwiązania w zakresie grafiki, oprogramowanie medyczne, itd.
- Istotą takich projektów jest wykorzystanie jakiegoś segmentu rynku, który dotąd nie istniał, lub nowość cywilizacyjna
- W tym wykładzie przyjmujemy, że klient istnieje i ustala czego mu potrzeba

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 rzadko wie, jakie wymagania zapewnią osiągniecie jego celów.

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 systemu na zamówienie analitycy mają bezpośredni kontakt z przedstawicielami klienta. Faza ta wymaga dużego zaangażowania ze strony klienta, ze strony przyszłych użytkowników systemu i ekspertów w dziedzinie.

Trudność określenia wymagań

- Klient z reguły nie wie dokładnie w jaki sposób osiągnąć założone cele. 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 inną terminologią mówiąc o tych samych problemach.
- 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.
- Część wymagań nie jest określona przez klienta, lecz wynika z ogólnej kultury, przepisów prawnych, powszechnych zwyczajów, analogii z istniejącymi rozwiązaniami, utrwalonych przyzwyczajeń, itp.

Poziomy ogólności 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.

Formalna 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.

Jakość 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 i porównania z istniejącym oprogramowaniem i prototypami.

Powinien być uzyskany stan porozumienia pomiędzy projektantami i użytkownikami dotyczący projektowanego systemu.

Wiedza i doświadczenia potencjalnej organizacji podejmującej się rozwoju systemu powinny wspomóc studia nad osiągalnością systemu.

W wielu przypadkach dużym wspomaganiem jest budowa prototypów.

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

Metody rozpoznania wymagań

- **Wywiady i przeglądy.** Wywiady powinny być przygotowane (w postaci listy pytań) i podzielone na odrębne zagadnienia. Podział powinien przykrywać całość tematu i 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.
- **Studia osiągalności**. Określenie realistycznych celów systemu i metod ich osiągnięcia.
- **Prototypowanie**. Zbudowanie prototypu systemu działającego w zmniejszonej skali, z uproszczonymi interfejsami.

Metody specyfikacji wymagań

- **Język naturalny** - najczęściej stosowany. Wady: *niejednoznaczność* powodująca różne rozumienie tego samego tekstu; *elastyczność*, powodująca wyrazić te same treści na wiele sposobów. Utrudnia to wykrycie powiązanych wymagań i powoduje trudności w wykryciu sprzeczności.
- **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).
- **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.
- **Formalizm matematyczny**? Można stosować dla specyficznych celów.

Wymagania funkcjonalne

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.

Formularz wymagań funkcjonalnych

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

Przykład jednej zapełnionej tabeli wg przyjętego formularza:

Nazwa funkcji	*Edycja dochodów pracownika*
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.

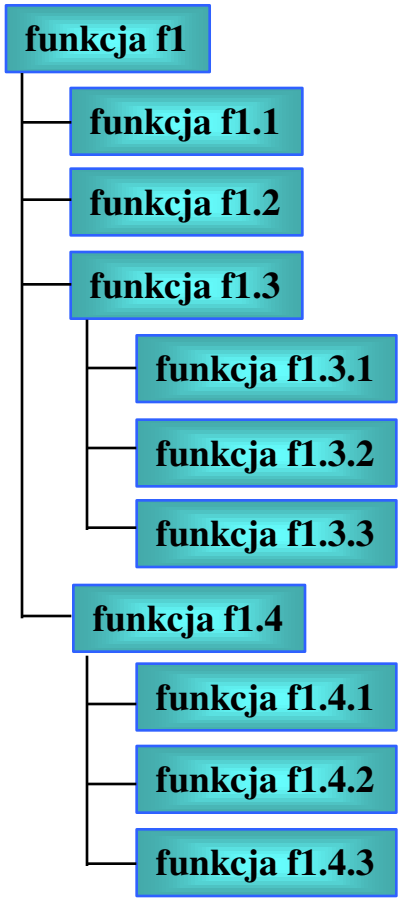
Porządkowanie wymagań

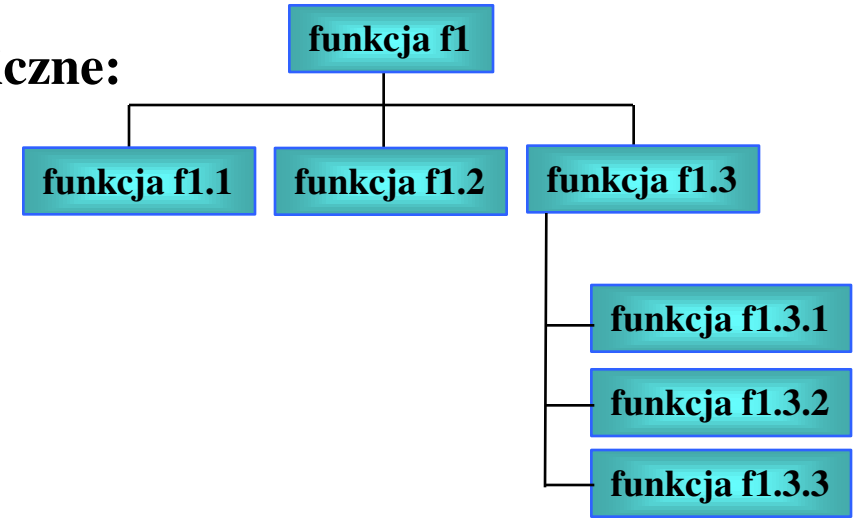
Hierarchia wymagań funkcjonalnych:
Z reguły funkcje można rozbić na podfunkcje.

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

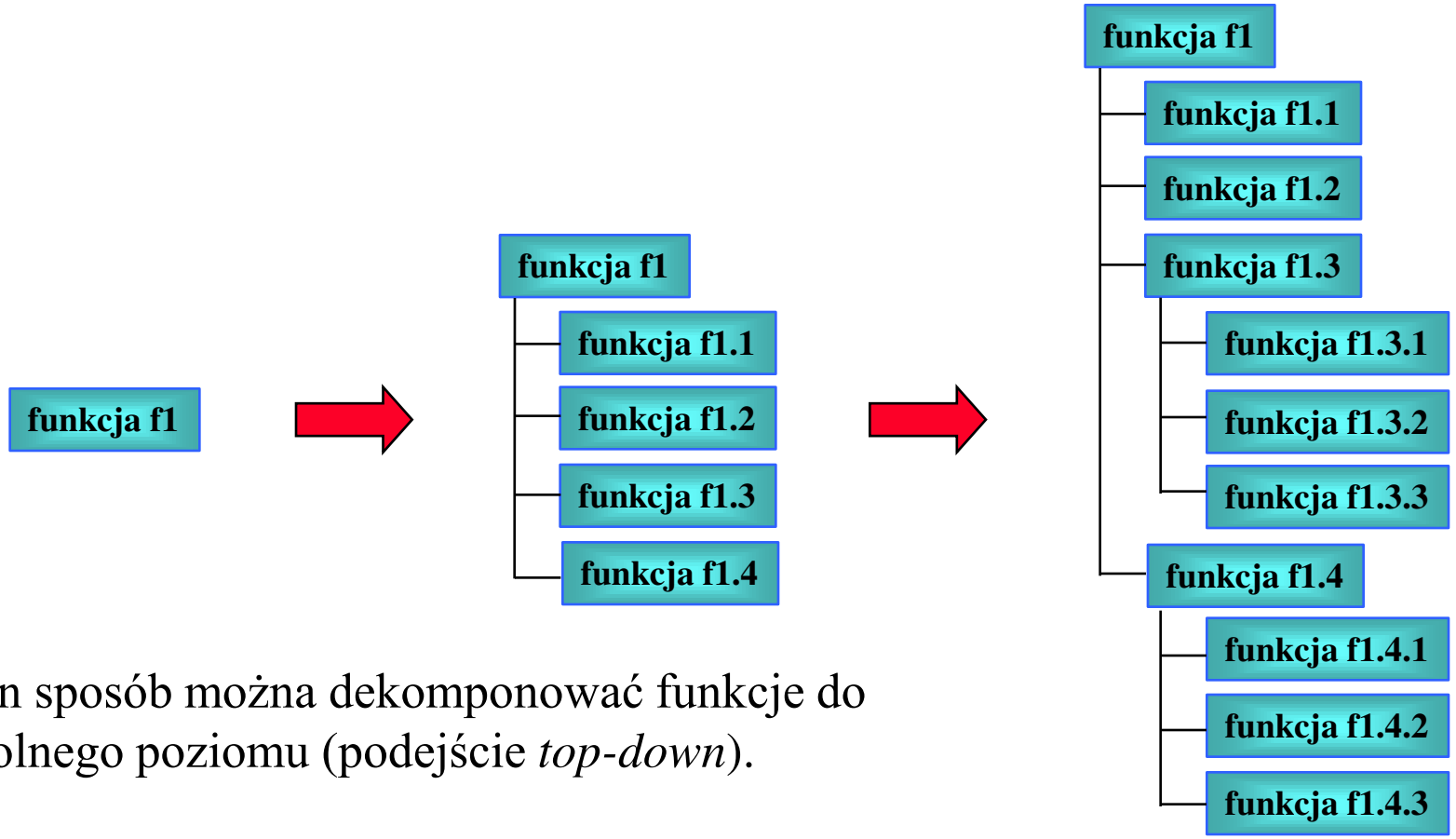
Notacje graficzne:





- 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.

Przykład: program podatkowy

„Surowy” wynik wywiadów z klientem:

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

Przykład: system harmonogramowania zleceń

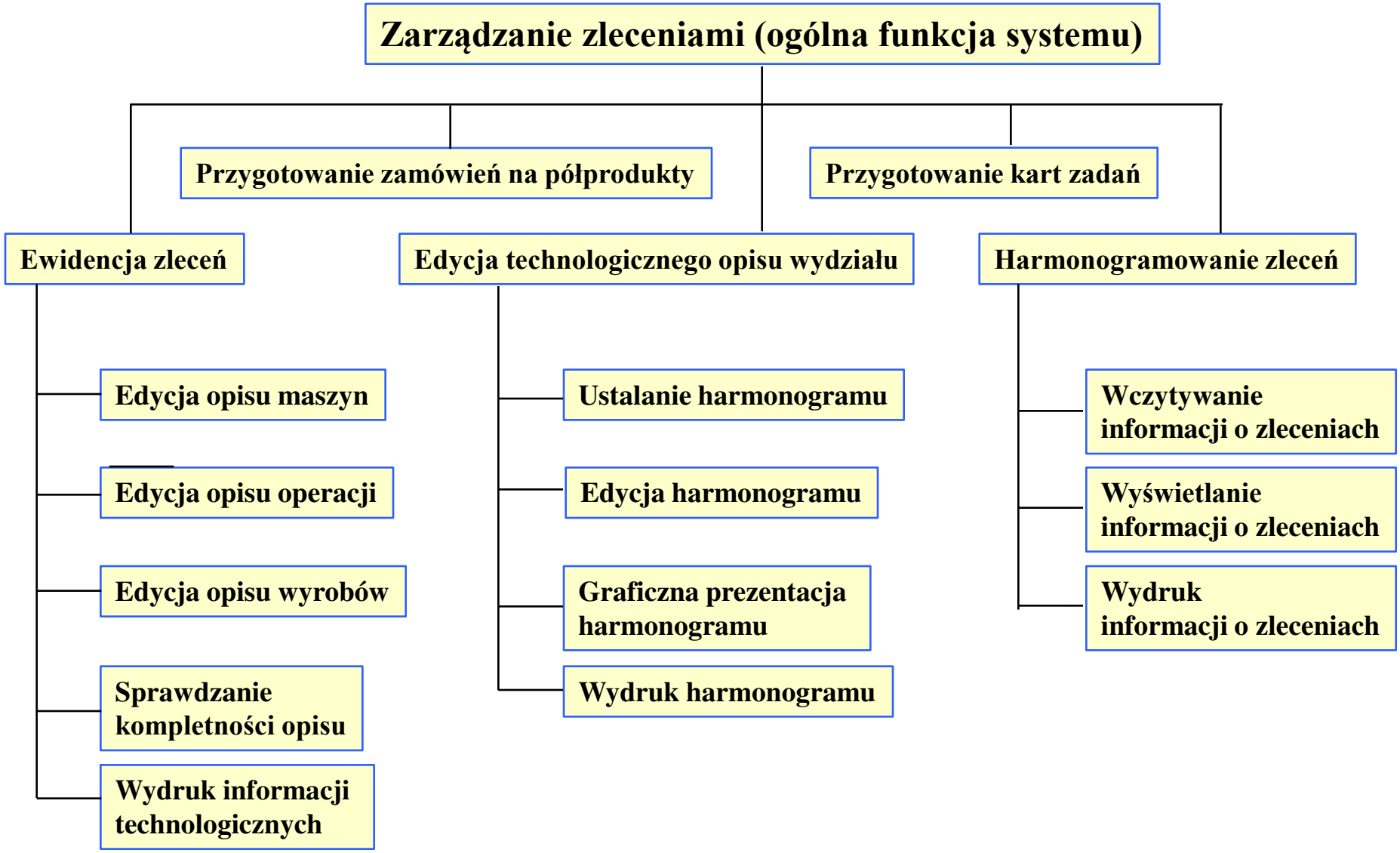
„Surowy” wynik wywiadów z klientem:

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



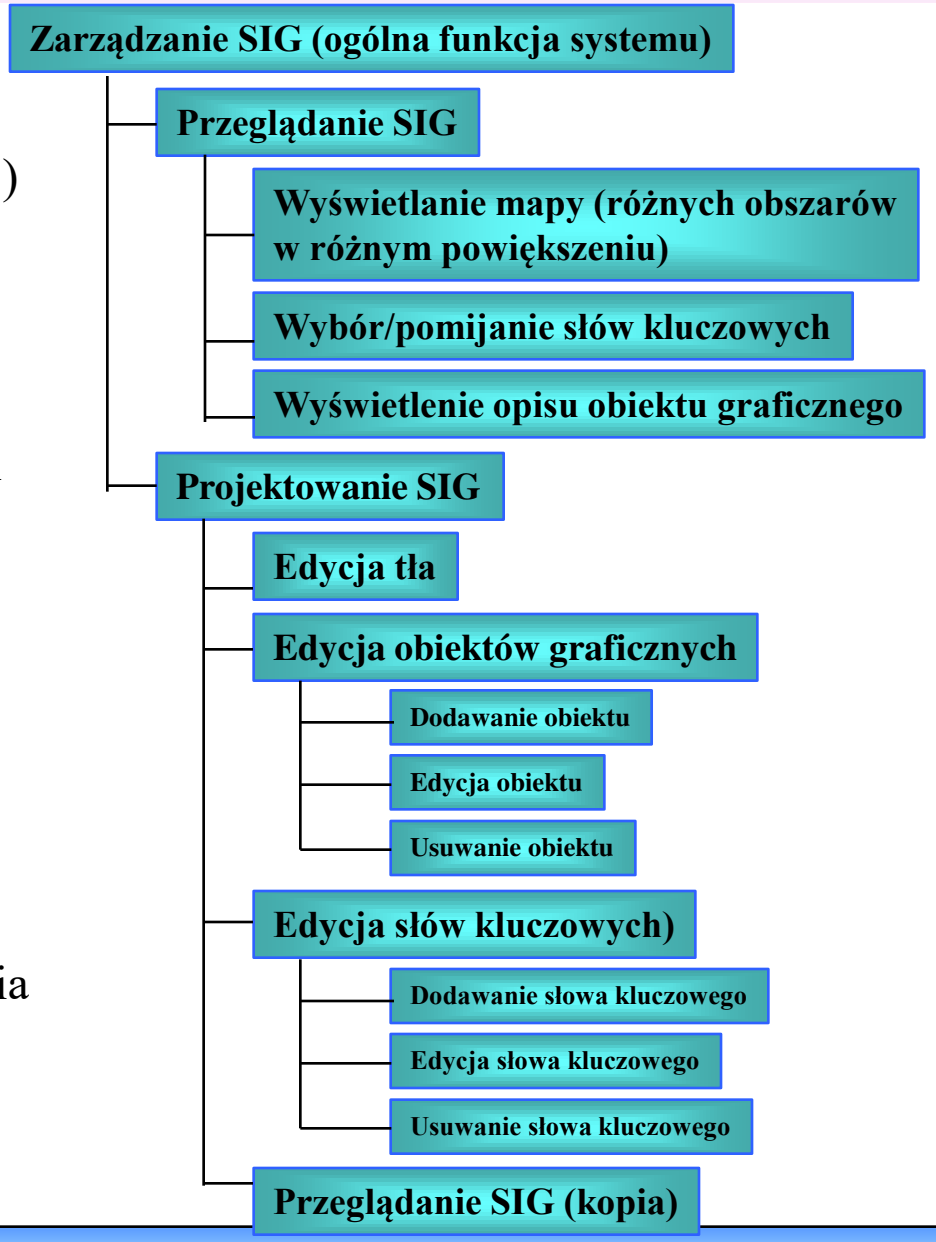
Przykład: system informacji geograficznej - SIG

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.



Diagramy przypadków użycia

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

Klasy użytkowników:

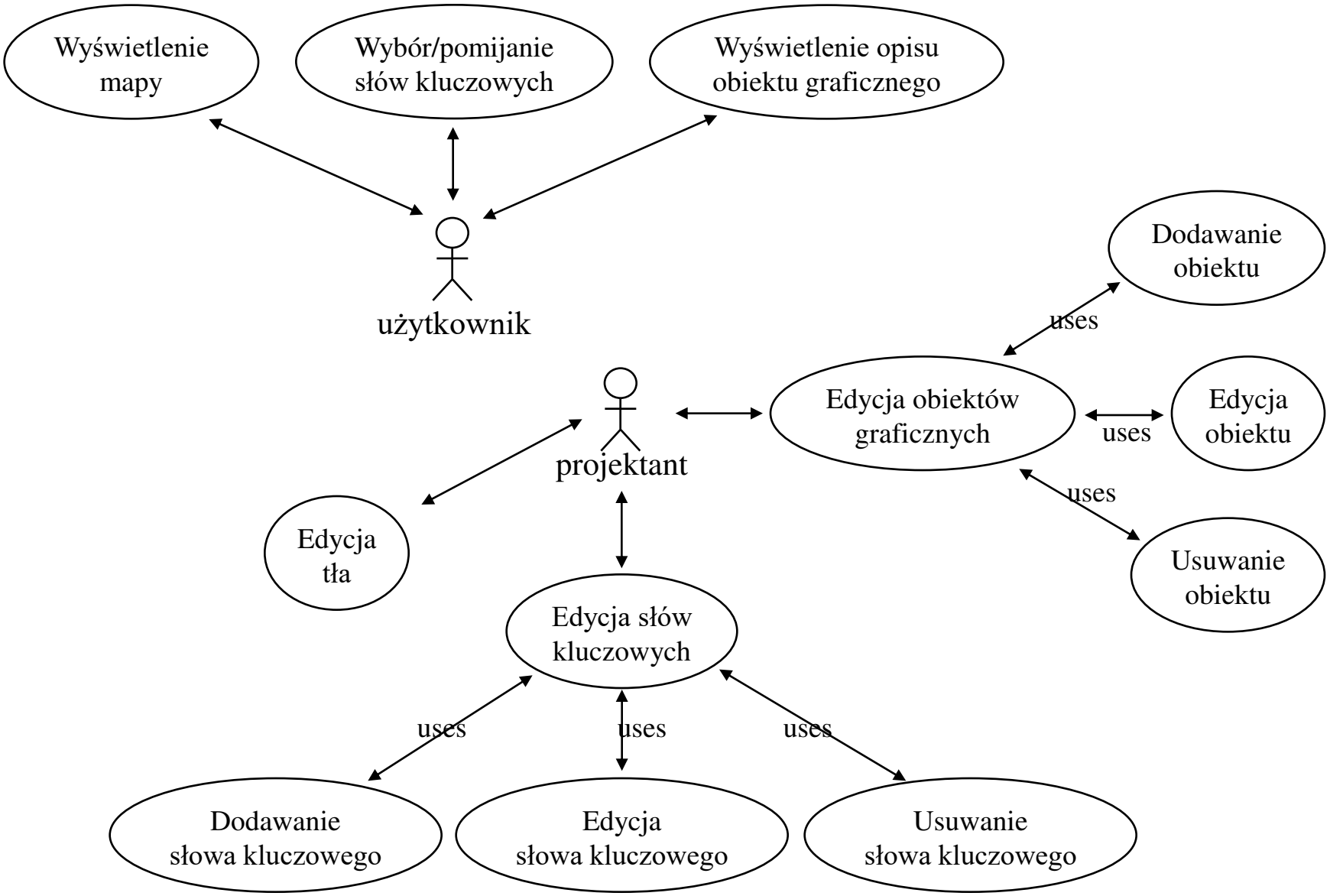
- sekretarka
- projektant
- osoba przeglądająca mapę

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. sekretarka.

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.

Diagram przypadków użycia dla SIG



Wymagania niefunkcjonalne

Opisują ograniczenia, przy których system ma realizować swoje funkcje.

- **Wymagania dotyczące produktu**.

  Np. musi istnieć możliwość operowania z systemem wyłącznie za pomocą klawiatury.

- **Wymagania dotyczące procesu**.

  Np. proces realizacji harmonogramowania zleceń musi być zgodny ze standardem opisanym w dokumencie XXXA/96.

- **Wymagania zewnętrzne**.

  Np. system harmonogramowania musi współpracować z bazą danych systemu komputerowego działu marketingu opisaną w dokumencie YYYB/95. Niedopuszczalne są jakiekolwiek zmiany w strukturze tej bazy.

Formularz zapisu wymagań

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 (nazwisko, PESEL, 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

Czynniki uwzględniane przy konstruowaniu 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.

Czynniki uwzględniane przy konstruowaniu 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.

Czynniki uwzględniane przy konstruowaniu 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.

Często 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 specyfikacji wymagań (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 specyfikacji wymagań(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).

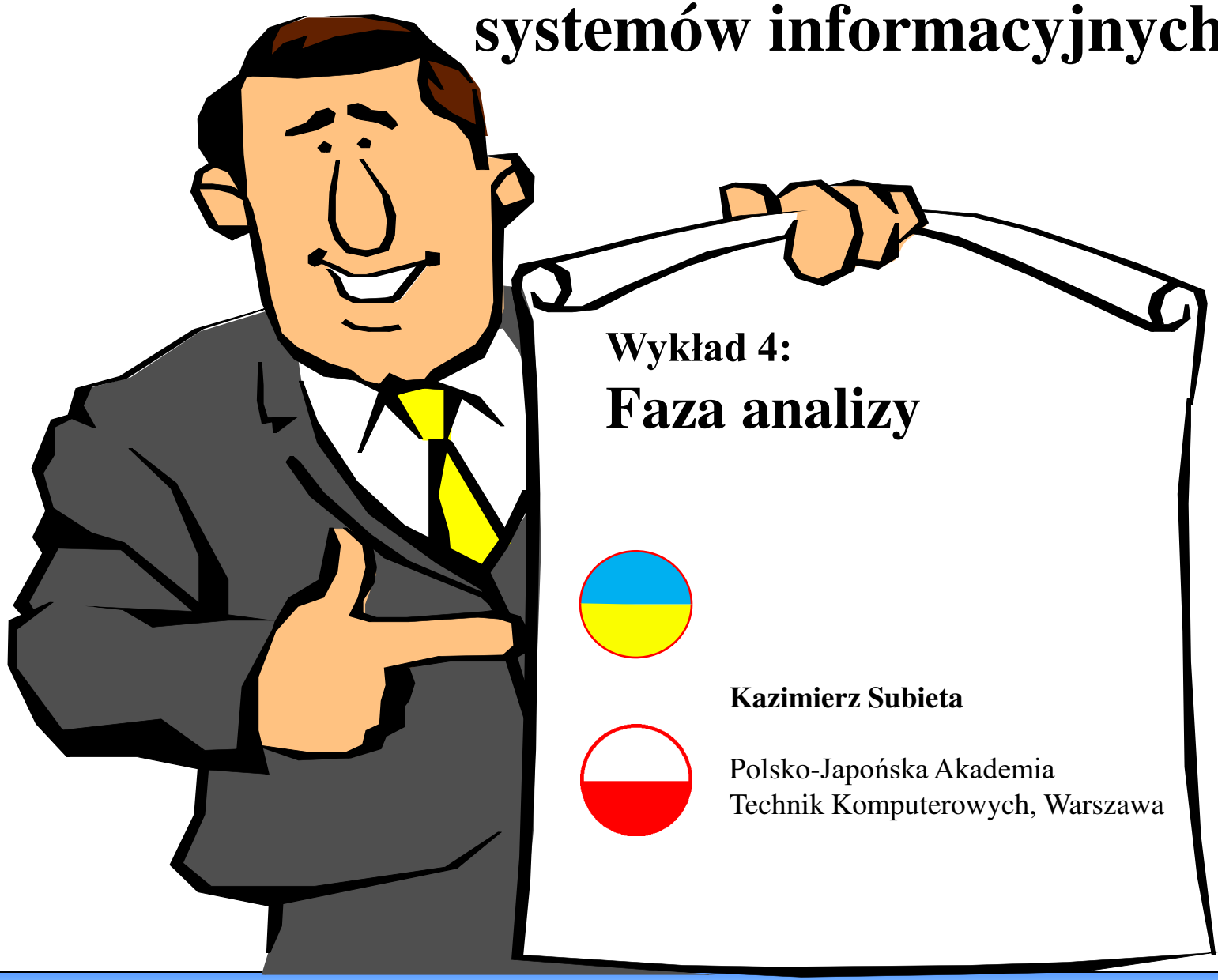
Wszystkie specyficzne terminy powinny być umieszczone w słowniku, wraz z wyjaśnieniem. Słownik powinien precyzować terminy niejednoznaczne i określać ich znaczenie w kontekście tego dokument (być może nieco węższe).

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.	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.

Budowa i integracja systemów informacyjnych



Wykład 4:
Faza analizy

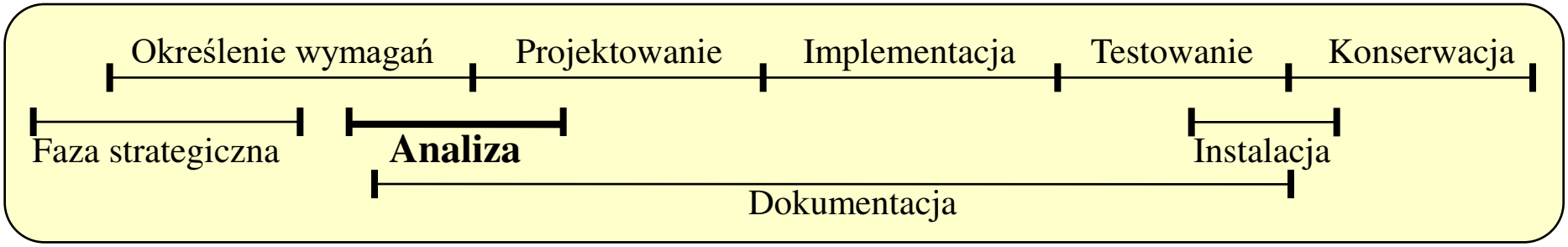
Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Plan wykładu

- Model analityczny
- Czynności w fazie analizy
- Wymagania na oprogramowanie
- Notacje w fazie analizie
- Metodyki strukturalne i obiektowe
- Metodyki obiektowe; UML
- Proces tworzenia modelu obiektowego
- Dokument wymagań na oprogramowanie
- Kluczowe czynniki sukcesu i rezultaty fazy analizy

Faza analizy



- Celem fazy **analizy** jest ustalenie wszystkich tych czynników lub warunków w dziedzinie przedmiotowej, w otoczeniu realizatorów projektu, w istniejących lub planowanych systemach komputerowych, które mogą wpłynąć na decyzje projektowe, na przebieg procesu projektowego i na realizację wymagań.
- Wynikiem jest **logiczny model systemu**, opisujący sposób realizacji przez system postawionych wymagań, lecz abstrahujących od szczegółów implementacyjnych.
  - Faza analizy jest kontynuacją fazy zbierania wymagań, a jednocześnie krystalizacją wizji przyszłego systemu.
- Końcowe dokumenty analizy są pierwszymi dokumentami projektowymi.
  - W odróżnieniu, celem fazy **projektowania** jest udzielenie odpowiedzi na pytanie: Jak system ma być zaimplementowany? Wynikiem jest opis sposobu implementacji.

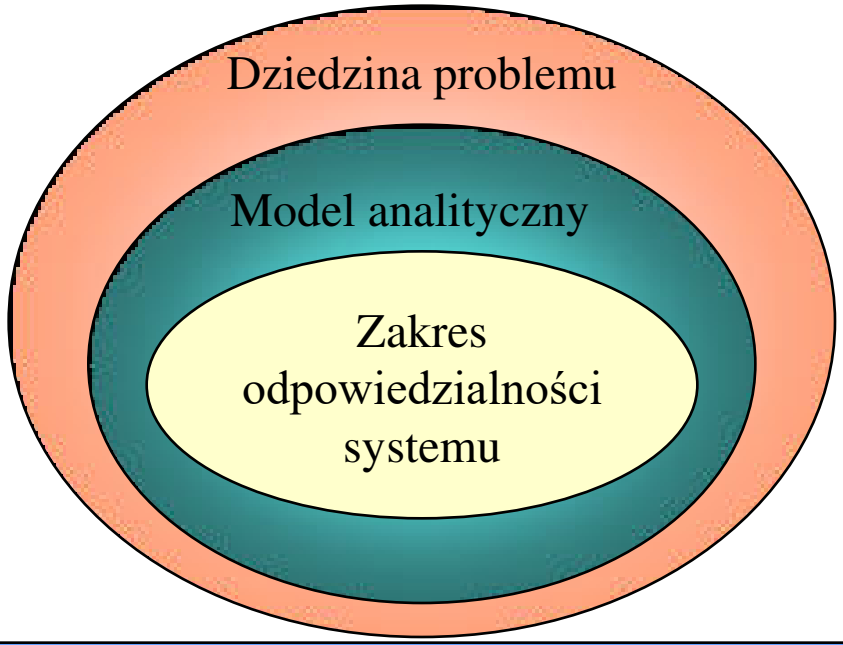
Model analityczny

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

- Uproszczony opis systemu - pokazuje co system musi robić
- Hierarchiczna dekompozycja funkcji systemu
- Opisany przy pomocy notacji zgodnej z pewną konwencją
- Zbudowany przy użyciu dobrze rozpoznanych metod i narzędzi
- Używany do wnioskowania o przyszłym oprogramowaniu
- Unika terminologii implementacyjnej
- Pozwala na wnioskowanie „od przyczyny do skutku” i odwrotnie

- Model oprogramowania powinien być jego uproszonym opisem, opisującym wszystkie istotne cechy oprogramowania na wysokim poziomie abstrakcji.
- Model ten nie zastępuje doświadczenia i wnikliwości projektantów, lecz pomaga projektantom w zastosowaniu tych walorów.

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.**

Na początku analiza nie powinna stawiać nacisku na zmianę rzeczywistości poprzez wprowadzenie 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.

Na pewnym etapie analizy nie da się już uniknąć sformułowań i tez odnoszących się do przyszłego systemu. Dlatego nie da się też uniknąć decyzji projektowych.

Tematy i techniki analizy

- Budowa statycznego modelu klas
- Analiza funkcji i przypadków użycia
- Weryfikacja klas i obiektów
- Identyfikacja i definiowanie metod oraz komunikatów
- Modelowanie stanów i przejść między stanami
- Modelowanie procesów i przepływów danych
- Modelowanie przepływu sterowania
- Inne

Wiele tych technik było omówionych podczas prezentacji UML i metodyki obiektowej.

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
- Bezpieczeństwa
- Przenośności
- Jakości
- Niezawodności
- Pielęgnacyjności

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

Reguły modelu logicznego opartego na funkcjach

- 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).
- Należy podawać wskaźniki wydajnościowe funkcji (szybkość, częstość, itd) wszędzie tam, gdzie jest to możliwe.
- Powinny być zidentyfikowane funkcje krytyczne (od których zależy istota systemu).
- Nazwy funkcji powinny być *deklaracyjne*: odzwierciedlać ich cel i mówić ***co*** ma być zrobione, a nie ***jak*** ma być zrobione, (np. *Walidacja Zlecenia Zewnętrznego,* a nie *Czynności po Otrzymaniu Zlecenia*).

Notacje w fazie analizie

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 strukturalne

structured methodologies, structured analysis

Łączą statyczny opis danych oraz statyczny opis procesów.

- Metodyka Yourdona (DeMarco i Ward/Mellor)
- Structured System Analysis and Design Methodology (SSADM)
- Structured Analysis and Design Technique (SADT)

Zgodnie z DeMarco, **analiza strukturalna** używa następujących technik.

- Diagramy Przepływu Danych (*Data Flow Diagrams, DFD*)
- Słownik Danych (*Data Dictionary*)
- Strukturalny Angielski (*Structured English*) -> strukturalny polski
- Tablice i drzewa decyzyjne (*Decision Tables/Trees*)

Dodatkowo:

- Schemat Transformacyjny (*Transformation Schema*)
- Diagram Przejść Stanów (*State Transition Diagram*)
- Lista Zdarzeń (*Event List*)
- Schemat Danych (*Data Schema*)
- Pre- i post- warunki (*Pre- and Post-Conditions*)
- Diagramy Encja-Związek (*Entity-Relationships Diagrams)*

Uważa się, że wadą metodyk strukturalnych są trudności w zintegrowaniu modeli. Jest to jednak wyłącznie element walki ideologicznej (ideologiczna retoryka).

Metodyki obiektowe

- 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.

Przykłady: Express, OODA(Booch), OMT(Rumbaugh), OOSA(Shlaer-Mellor), Objectory(Jacobson), MOSES/OPEN, OOA/OOD(Coad/Yourdon), Notacja UML, RUP.

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 (np. model przepływu danych w OMT).
- Notacje proponowane przez różnych autorów nie są koniecznie nierozerwalne z samą metodyką. Np. notacji UML można użyć dla metodyki Coad/Yourdon.
- Narzędzia CASE nie narzucają metodyki; raczej, określają one tylko notację. Twierdzenia, że jakieś narzędzie CASE "jest oparte" na konkretnej metodyce jest hasłem reklamowym. Nawet najlepsze metodyki i narzędzia CASE nie są w stanie zapewnić jakości projektów.

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 **kombinować** ze sobą przyjęte oznaczenia.

Semantyka określa, co należy **rozumieć** pod przyjętymi oznaczeniami.

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

Autorzy notacji z reguły specyfikują wyłącznie składnię. Formalna specyfikacja semantyki jest nieosiągalna (brak metod formalnych), więc zastępuje się ją nieformalnymi objaśnieniami, zwykle na przykładach.

W zrozumieniu notacji najważniejsza jest pragmatyka, która określa, jak do konkretnej sytuacji dopasować zapisy notacji. Pragmatyka wyznacza więc *procesy* prowadzące do wytworzenia zapisów wyników analizy i projektowania, które są zgodne z intencją autorów tej notacji.

Nie ma innego sposobu objaśnienia pragmatyki oprócz pokazania sposobów użycia na **przykładach** przypominających realne sytuacje. Pomocne są **anty-przykłady** i **wzorce**. Realne sytuacje są zazwyczaj bardzo skomplikowane, co powoduje pewien infantylizm przykładów zamieszczanych w podręcznikach.

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.

Proces tworzenia modelu obiektowego

Zadania:

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

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

Dokument wymagań na oprogramowanie (1)

Informacje organizacyjne

Zasadnicza zawartość dokumentu

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

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

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

Dokument wymagań 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
- 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)

Jakość, styl oraz odpowiedzialność - podobnie jak dla wymagań użytkownika.

Plan zapewnienia jakości dla fazy analizy

Plan dotyczy wymagań odnośnie:

- Sposobów weryfikacji
- Sposobów testowania
- Jakości
- Niezawodności
- Pielęgnacyjności
- Bezpieczeństwa
- Standardów

Plan 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

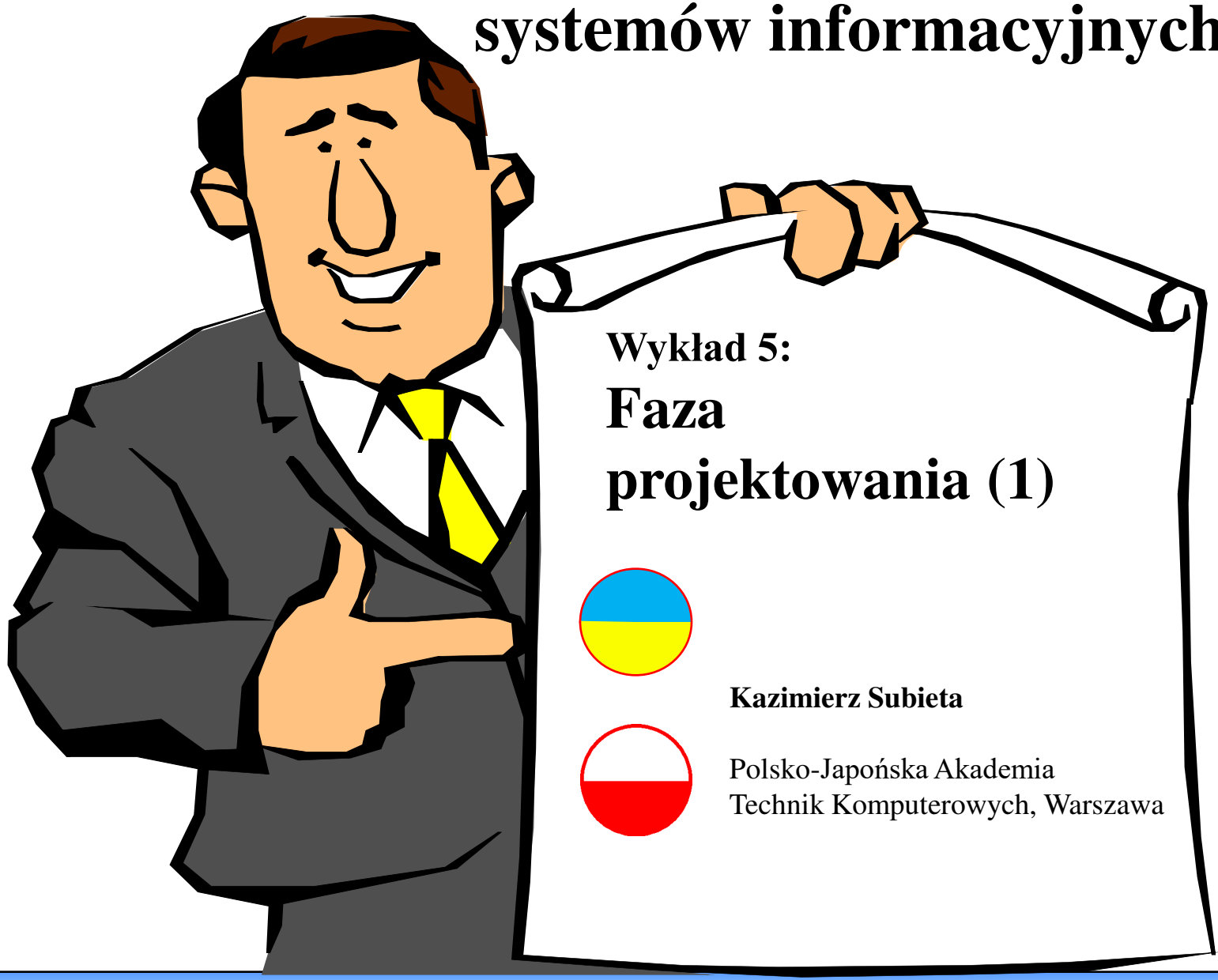
Kluczowe czynniki sukcesu fazy analizy

- **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**

Podstawowe rezultaty fazy analizy

- **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**

Budowa i integracja systemów informacyjnych



Wykład 5:
Faza projektowania (1)

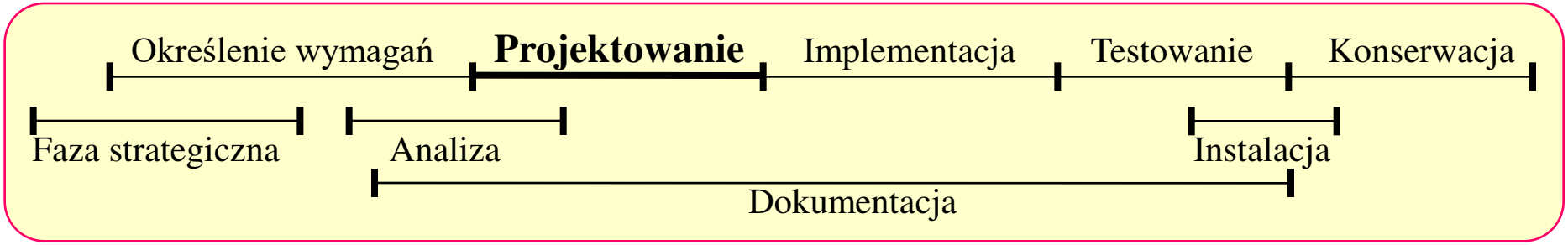
Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Plan wykładu

- Zadania wykonywane w fazie projektowania
- Techniki obiektowe w projektowaniu
- Dodatkowe elementy notacji
- Uszczegółowianie wyników analizy
- Projektowanie składowych systemu nie związanych z dziedziną problemu
- Projektowanie interfejsu użytkownika
- Organizacja interakcji z użytkownikiem
- Zasady projektowania interfejsu użytkownika

Projektowanie



- Celem projektowania jest opracowanie szczegółowego opisu implementacji systemu.
- 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.
- 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.
- Wykorzystanie idei programowania strukturalnego i obiektowego.

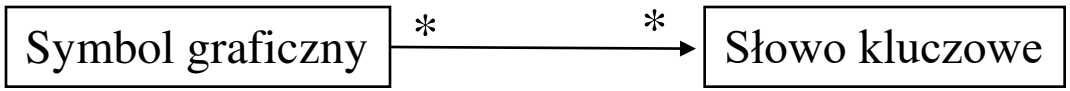
Zadania wykonywane w fazie projektowania

- Uszczegółowienie wyników analizy. Projekt musi być wystarczająco szczegółowy aby mógł być podstawą implementacji. Stopień szczegółowości zależy od poziomu zaawansowania programistów.
- Projektowanie składowych systemów nie związanych z dziedziną problemu
- Optymalizacja systemu
- Dostosowanie do ograniczeń i możliwości środowiska implementacji
- Określenie fizycznej struktury systemu.

Techniki obiektowe w projektowaniu

W projektowaniu często pomocne są specjalne notacje, jako uzupełnienie do notacji stosowanych w analizie.

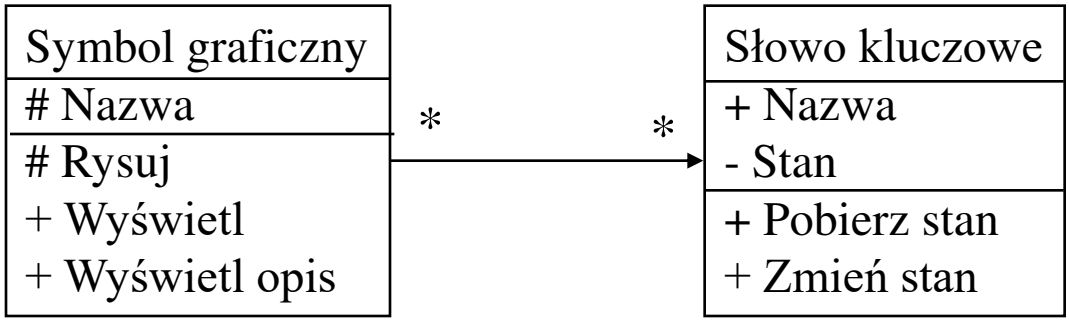
Związek skierowany: oprócz zaznaczenia związku zaznacza się kierunek przesyłania komunikatów. Np w systemie SIG obiekty klasy "Symbol graficzny" wysyłają „komunikaty" do obiektów "Słowo kluczowe". Alternatywnie, strzałka może oznaczać pointer.



Symbole dostępu do pól i metod:

Jest to związane z konwencja C++, gdzie dostęp może być:

- **(+) publiczny** - dla wszystkich funkcji i metod
- **(#) zabezpieczony** (*protected*) - dostęp do metod danej klasy oraz jej specjalizacji
- **(-) prywatny** - dostęp tylko dla funkcji danej klasy



Dodatkowe elementy notacji

Wiele z nich występuje w innych metodykach, np. w metodyce Boocha:

- **Wzorce klas** (*class templates*) – klasy parametryzowane, zwykle typem
- **Klasy kolekcji**, tj. klasy zawierające pola i metody dotyczące klasy jako całości a nie pojedynczych obiektów, np. pola i metody „statyczne”.
- **Wolne funkcje** nie będące metodami żadnej z klas.
- **Sposoby widoczności obiektu** do którego wysyłany jest komunikat. Obiekt ten może być widoczny, gdyż znajduje się w tym samym zakresie, jest przekazany przez parametr lub jest polem klasy, której metoda wysyła komunikat.

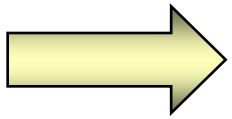
Uszczegółowianie wyników analizy (1)

Uszczegółowianie poprzez podanie odwzorowanie notacji abstrakcyjnej w struktury języka programowania.

Dane z analizy:

Adres
- Ulica +
- Numer domu +
- Numer mieszkania +
- Miasto +
- Kod

Dane osobowe
- Imię +
- Nazwisko +
- Adres



Realizacja w C/C++:

```
typedef struct {
        char Ulica[30];
        char NumerDomu[10];
        char NumerMieszkania[10];
        char Miasto[30];
        char Kod[5];
        } TypAdres;

typedef struct {
        char Imię[30];
        char Nazwisko[30];
        TAdres Adres;
        } TypDaneOsobowe;
```

Uszczegółowianie wyników analizy (2)

Uszczegółowianie metod

- Podanie nagłówków metod oraz ich parametrów.
- Określenie, które z metod będą realizowane jako funkcje wirtualne (późno wiązane, podczas wykonania) a które jako zwyczajne funkcje (wiązane statycznie podczas kompilacji i linkowania). (Dotyczy C++)
- Zastąpienie niektórych prostych metod bezpośrednim dostępem do atrybutów. Np. metody PobierzNazwisko, UstawNazwisko, etc.
- Zastąpienie niektórych atrybutów przez odpowiednie metody, np.
  Wiek = BieżącaData - DataUrodzenia;
  KwotaDochodu = KwotaPrzychodu - KwotaKosztów;
  Niekiedy atrybuty są zastępowane przez tzw. „getery” (odczyt wartości) i „setery” (zapis wartości), co (podobno) czyni program bardziej bezpiecznym.

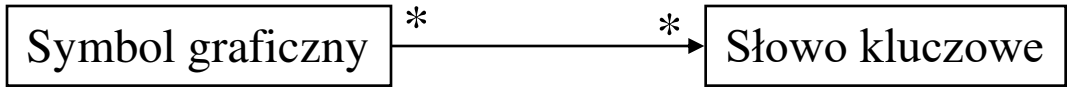
Uszczegółowianie wyników analizy (3)

Określenie sposobów implementacji związków (asocjacji)

Asocjacje (w tym agregacje) są zazwyczaj niedostępne bezpośrednio w językach programowania. Można je zaimplementować na wiele sposobów, z reguły poprzez wprowadzenie dodatkowych atrybutów (pól). Mogą one być następujące:

- obiekty powiązanej klasy
- wskaźniki (referencje, identyfikatory) do obiektów powiązanej klasy
- klucze obiektów powiązanej klasy

W zależności od przyjętego sposobu oraz od liczności związków (1:1, 1:n, n:1, m:n) możliwe są bardzo różne deklaracje w przyjętym języku programowania.



Tablica obiektów: ⟶
```
TypSłowoKluczowe SłowaKluczowe[100];
```
Lista wskaźników: ⟶
```
list< TypSłowoKluczowe *> SłowaKluczowe;
```
Tablica wskaźników: ⟶
```
char * WskaźnikiSłówKluczowych[100];
```

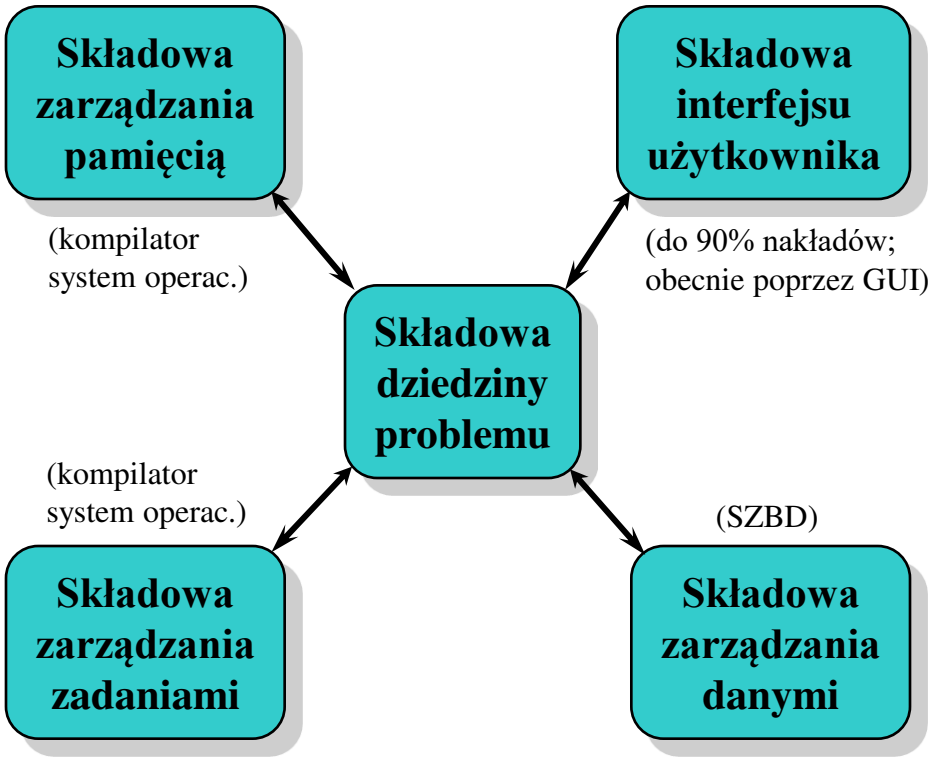
Dodatkowe reguły dla transformacji schematów obiektowych na relacyjne.

Projektowanie składowych systemu nie związanych z dziedziną problemu

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 zastąpił termin „języków/środowisk czwartej generacji" (4GL).

Przykładami narzędzi RAD są: Borland Delphi RAD Pack, IBM VisualAge (for Cobol, Java, C++, Smalltalk), Microsoft Access Developer's Toolkit, Microsoft Visual FoxPro Professional, PowerBuilder Desktop, Power++ i wiele innych.

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.
ODRA – Object Database for Rapid Application development

Projektowanie interfejsu użytkownika

W ostatnich latach nastąpił rozwój narzędzi graficznych służących do tego celu: MS Windows, Object Windows, MS Foundation Class.

Systemy zarządzania interfejsem użytkownika: Zapp Factory, Visual Basic.

- 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

- **Za pomocą linii komend**
  - Dla niewielkich systemów
  - Dla prototypów
  - Dla zaawansowanych użytkowników
- **W pełnoekranowym środowisku okienkowym**
  - Dla dużych systemów
  - Wygodny dla początkujących i średnio zaawansowanych użytkowników
- **Powszechne użycie przeglądarek internetowych spowodowało wykształcenie się standardu (stereotypu) interfejsu użytkownika, akceptowanego powszechnie.**
  - Nie warto od niego odchodzić, ale dla niektórych mniej standardowych przypadków może to być konieczne.

Typowe sposoby wydawania przez użytkownika poleceń systemowi

- Wpisywanie poleceń za pomocą linii komend
- Wybór opcji z menu
- Wciśnięcie odpowiedniej kombinacji klawiszy (skrótu)
- Korzystanie z ikon w paskach narzędziowych
- Wybór przycisku w dialogu
- Korzystanie z nawigacji kursorem myszy i przycisków myszy

Wprowadzanie i wyprowadzanie danych

Wprowadzanie przez użytkownika:

Podawanie parametrów poleceń w przypadku systemów z linią komend

Wprowadzanie danych w odpowiedzi na zaproszenie systemu

Wprowadzanie danych w dialogach

Wyprowadzanie przez system:

Wyświetlanie informacji w dialogach

Wyświetlanie i/lub wydruki raportów

Graficzna prezentacja danych

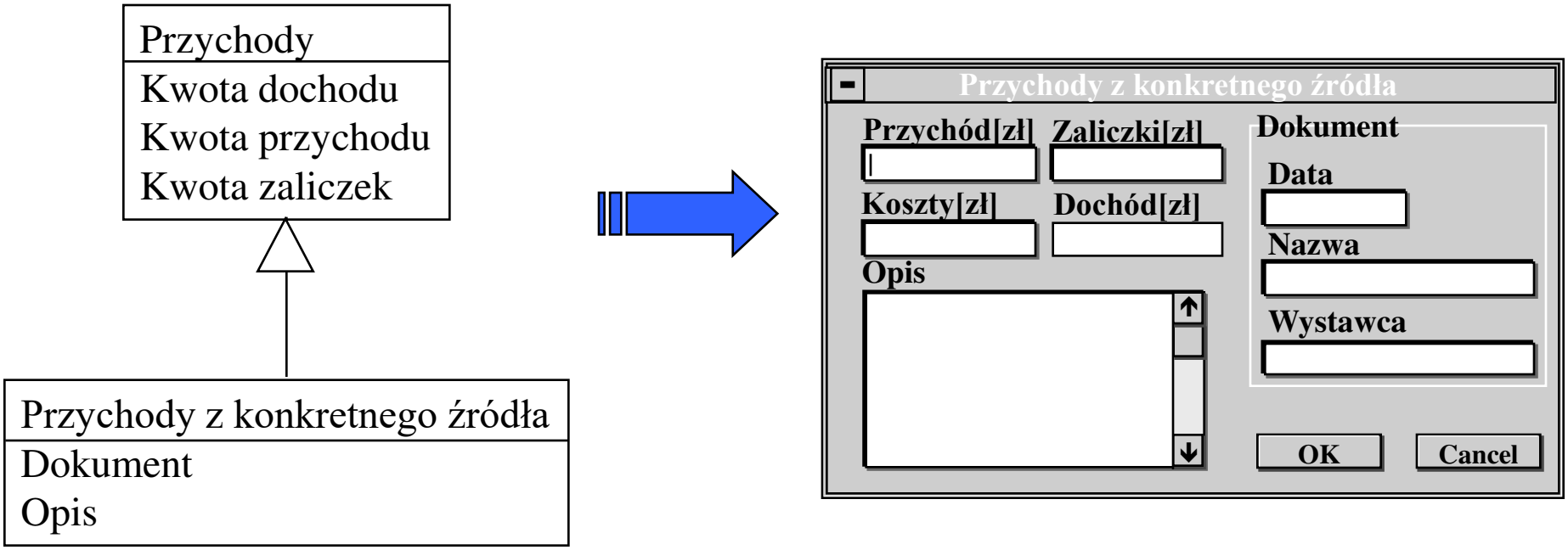
Prototyp interfejsu użytkownika może powstać już w fazie określenia wymagań. Systemy zarządzania interfejsem użytkownika pozwalają na wygodną budowę prototypów oraz wykorzystanie prototypu w końcowej implementacji.

Przykład okna dialogowego

Dialog:

- Przepływ danych pomiędzy użytkownikiem a systemem
- Parametry komunikatów wysyłanych przez użytkownika
- Metody i procesy, które zgodnie ze specyfikacją służą do edycji obiektów, encji lub zbiorników danych

Edycja obiektu “Przychody z konkretnego źródła”:



Zasady projektowania interfejsu użytkownika (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

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

- **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)

- **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.

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

- **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)

- **Nieobciążanie pamięci krótkotrwałej użytkownika** (*awareness*). Użytkownik może zapomnieć 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.

- **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 7 +/- 2. Człowiek może się jednocześnie skupić na 5 - 9 elementach.
Ta reguła powinna być uwzględniana przy projektowaniu interfejsu użytkownika. 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.

Dwa funkcjonalnie równoważne dialogi

Wewnętrzne grupowanie pól:

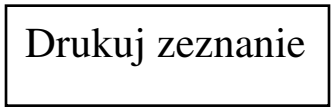
Bez wewnętrznego grupowania pól:

Techniki/diagramy strukturalne (1)

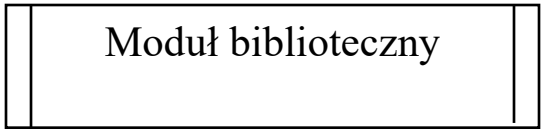
structure charts/diagrams

UML ich nie przewiduje (z mętnym uzasadnieniem przyczyny)

Moduł: aktywna składowa programu, tj. procedura lub funkcja (lub ich zestaw).

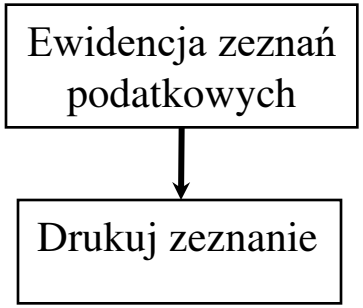


Moduł biblioteczny: gotowa procedura lub funkcja wykorzystywana w systemie.



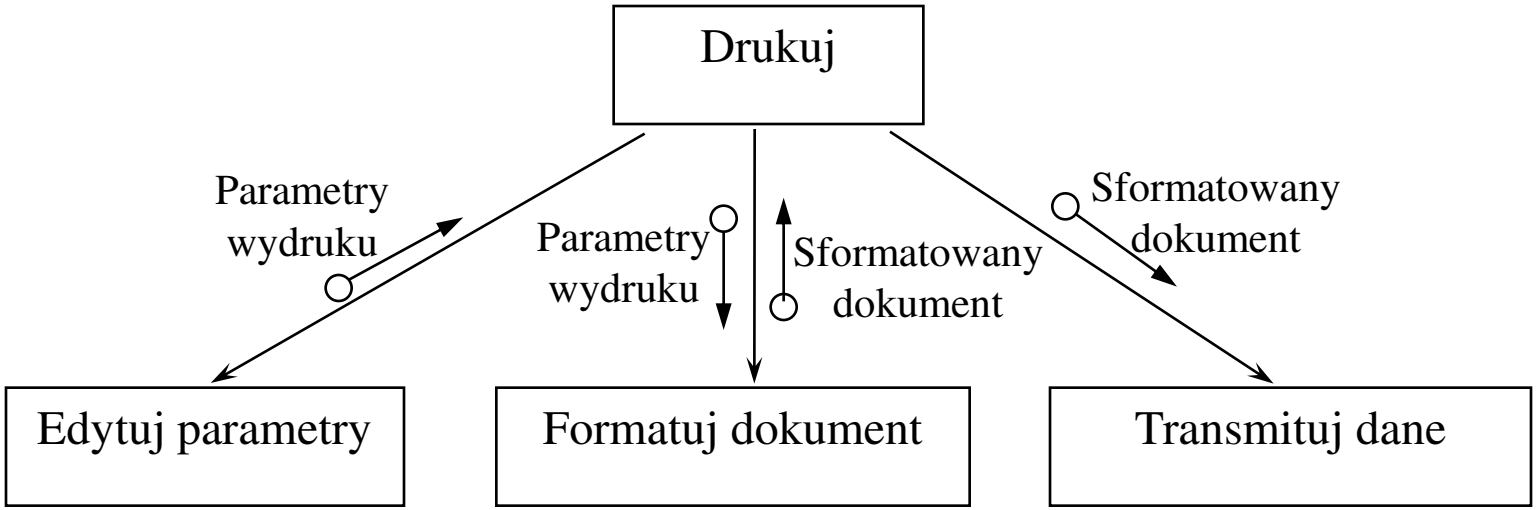
Dane: relacja w bazie danych, plik lub zmienne programu

Wywołanie (call): wywołanie przez pewien moduł innego modułu.



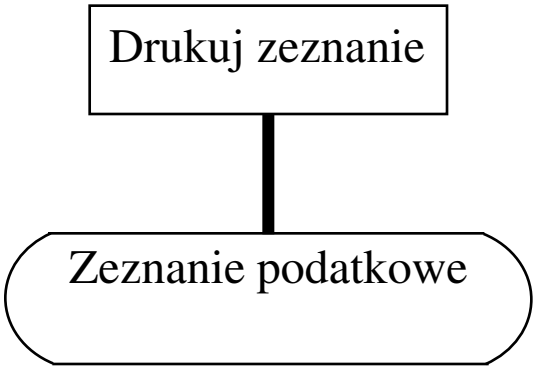
Techniki/diagramy strukturalne (2)

Flagi przepływu danych: z wywołaniem modułu może być związany przepływ danych z modułu wywołującego do wywoływanego i odwrotnie. Pierwszy rodzaj odpowiada parametrom wejściowym, drugi wynikowi i parametrom wyjściowym.



Techniki/diagramy strukturalne (3)

Korzystanie z danych:

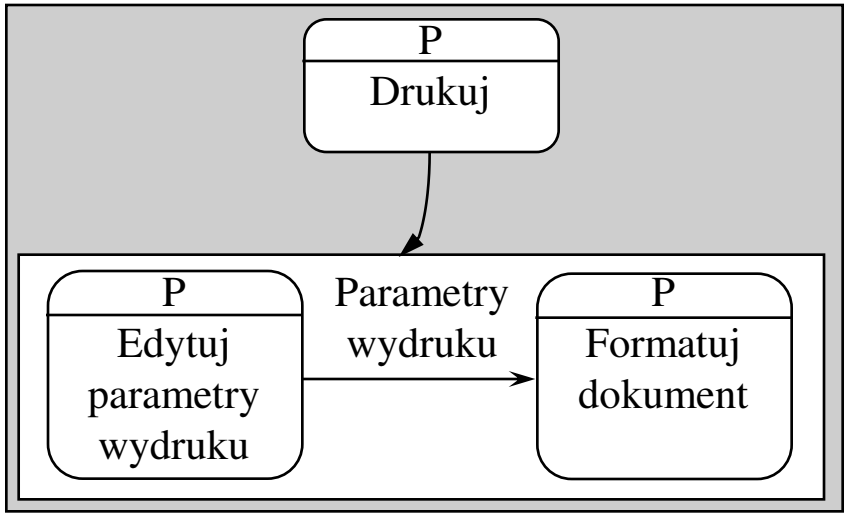


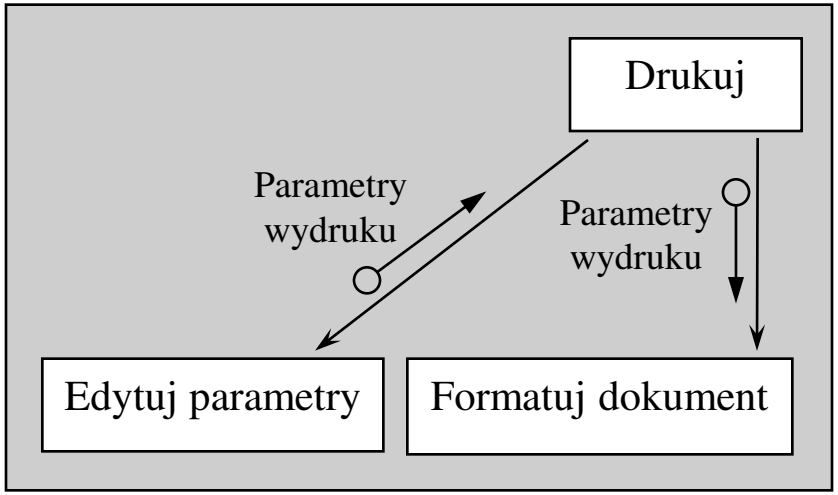
Diagramy strukturalne formatuje się z reguły tak, aby moduły wyższego poziomu - moduły wywołujące - znajdowały się **powyżej** modułów niższego poziomu - modułów wywoływanych.

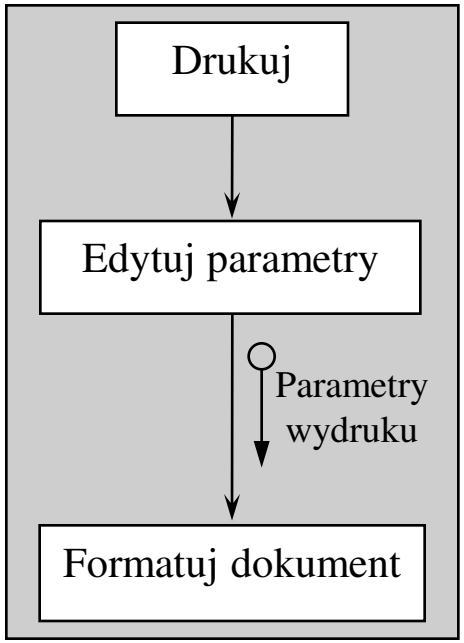
Diagramy strukturalne są uszczegółowieniem **diagramów przepływu danych**.

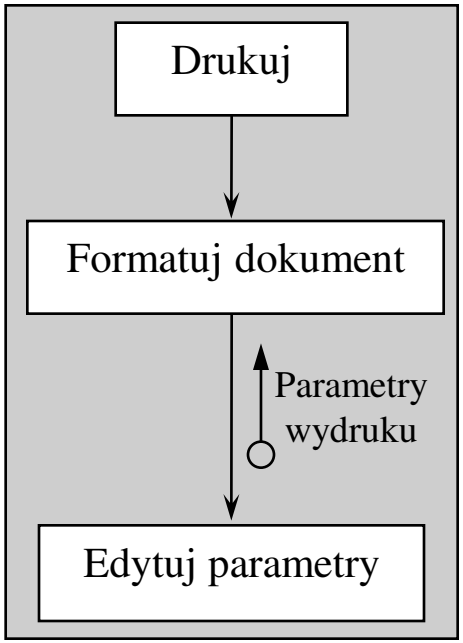
- Moduł odpowiadający procesowi wyższego poziomu wywołuje moduł będący źródłem danych, a następnie moduł będący odbiorcą danych.
- Moduł odpowiadający procesowi wyższego poziomu wywołuje moduł będący źródłem danych, który z kolej wywołuje moduł będący odbiorca danych.
- Moduł odpowiadający procesowi wyższego poziomu wywołuje moduł będący odbiorcą danych, który z kolej wywołuje moduł będący źródłem danych.

Techniki/diagramy strukturalne (4)

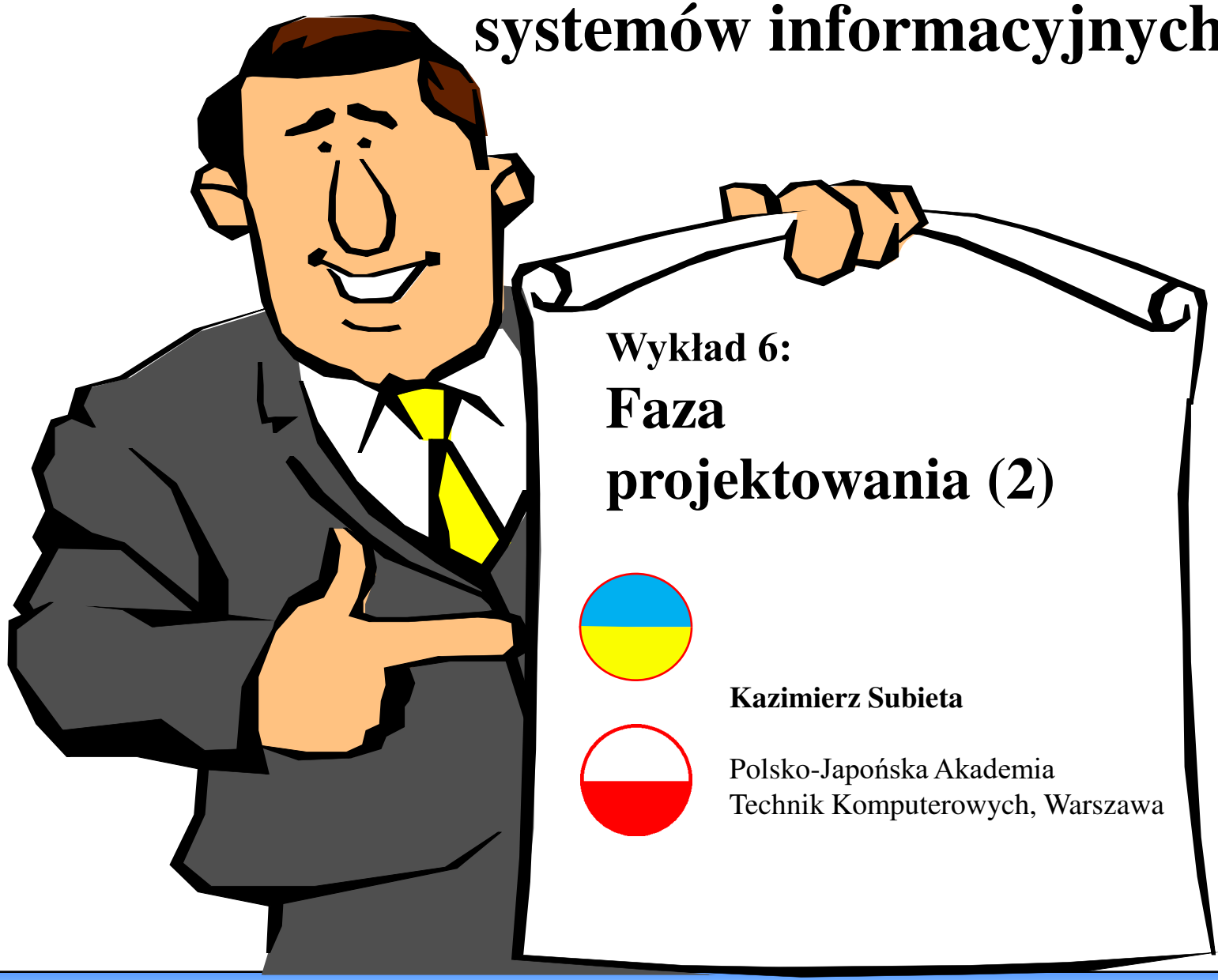








Budowa i integracja systemów informacyjnych



Wykład 6:
Faza projektowania (2)

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Plan wykładu

- Projektowanie składowej zarządzania danymi
- Optymalizacja projektu
- Dostosowanie do ograniczeń i możliwości środowiska implementacji
- Określenie fizycznej struktury systemu
- Graficzny opis sprzętowej konfiguracji systemu
- Poprawność i jakość projektu
- Wymagania niefunkcjonalne dla fazy projektowania
- Podstawowe rezultaty fazy projektowania

Projektowanie składowej zarządzania danymi

Co to znaczy „trwałe dane” (persistent)?
- Żyją dłużej niż program, który je wytworzył
- Podczas swego życia mogą być dostępne dla wielu programów

Trwałe dane mogą być przechowane w:
- plikach
- w bazie danych (relacyjnej, obiektowej, „NoSQL”, lub innej)
- w „chmurze” – czyli gdzieś w Internecie

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

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 programisty
- „wirtualnie”: bez uświadamiania programisty o transferach między pamięciami

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, inne)

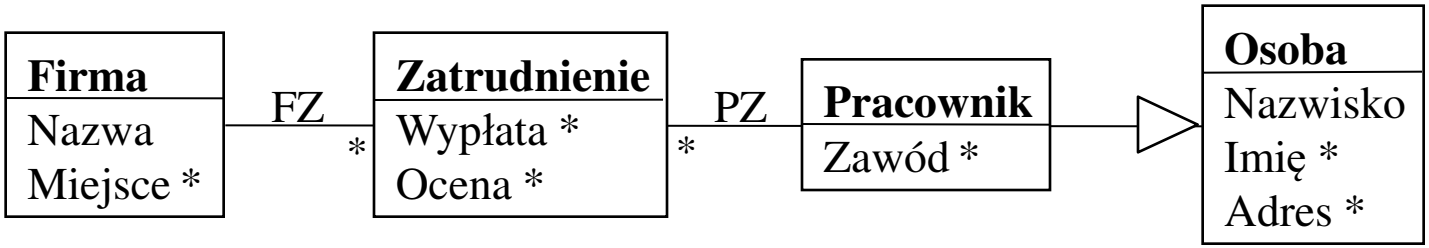
Integralność: poprawność danych w sensie ich organizacji i budowy.

Spójność: zgodność danych z rzeczywistością lub z oczekiwaniami użytkownika.

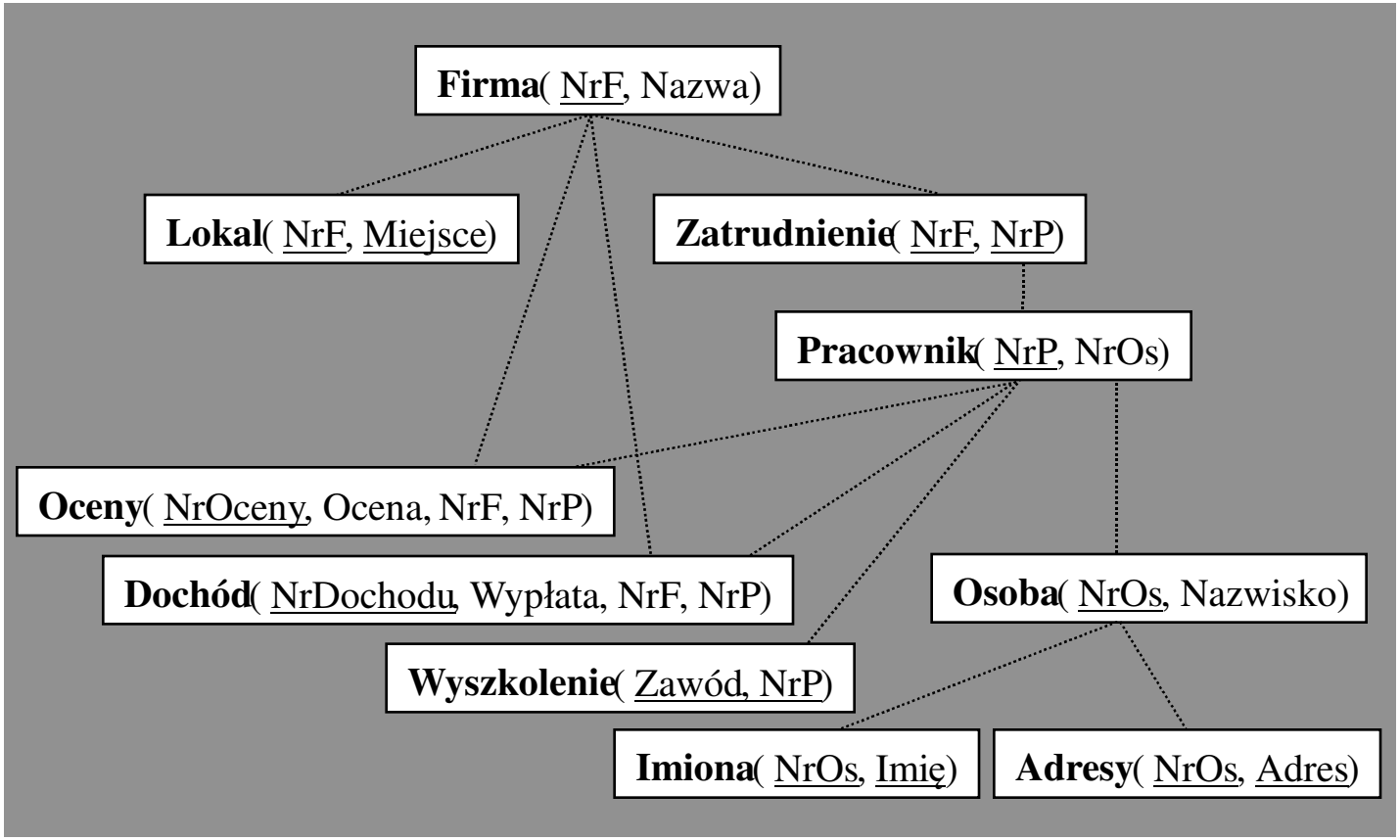
Wady relacyjnych baz danych

- Konieczność przeprowadzenie nietrywialnych odwzorowań przy przejściu z modelu pojęciowego (np. w UML) na strukturę relacyjną.
- Ustalony format krotki (rekordu) może powoduj trudności przy polach zmiennej długości (w niektórych systemach).
- Trudności (niesystematyczność) reprezentacji dużych wartości (grafiki, plików tekstowych, itd.). Zwykle niestandardowa.
- W niektórych sytuacjach - duże narzuty na czas przetwarzania (kosztowne złączenia)
- Niedopasowanie interfejsu dostępu do bazy danych (SQL) do języka programowania (np. Java), określana jako “niezgodność impedancji”.
- Brak możliwości rozszerzalności typów (zagnieżdżania danych)
- Brak systematycznego podejścia do informacji proceduralnej (metod)

Niezgodność modelu obiektowego i relacyjnego



10 sekund aby zrozumieć diagram



10 minut (lub więcej) aby zrozumieć diagram

Optymalizacja projektu (1)

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

- 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.

Wielu specjalistów jest przeciwna sztuczkom optymalizacyjnym: zyski są bardzo małe (o ile w ogóle są) w stosunku do zwiększenia nieczytelności/zawodności kodu.
Np. pierwsza sztuczka jest sprzeczna z zasadą enkapsulacji (podstawą inżynierii)

Optymalizacja projektu (2)

Co może przynieść zasadnicze zyski optymalizacyjne?

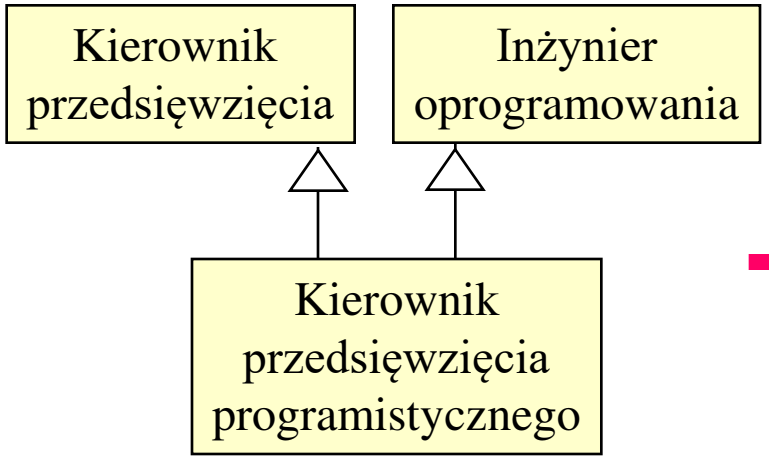
- **Zmiana algorytmu przetwarzania**. Np. zmiana algorytmu sortującego poprzez wprowadzenie pośredniego pliku zawierającego tylko klucze i wskaźniki do sortowanych obiektów może przynieść nawet 100-krotny zysk.
- **Wyłowienie "wąskich gardeł"** w przetwarzaniu i optymalizacja tych wąskich gardeł poprzez starannie rozpracowane procedury. Znana jest teza, że 20% kodu jest wykonywane przez 80% czasu (a la „zasada Pareto").
- **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. Wbrew „teorii", która nakazuje normalizację.
- **Stosowanie indeksów, tablic wskaźników i innych struktur pomocniczych**.
- **Analiza mechanizmów buforowania danych** w pamięci operacyjnej i ewentualna zmiana tego mechanizmu (np. zmniejszenie liczby poziomów)

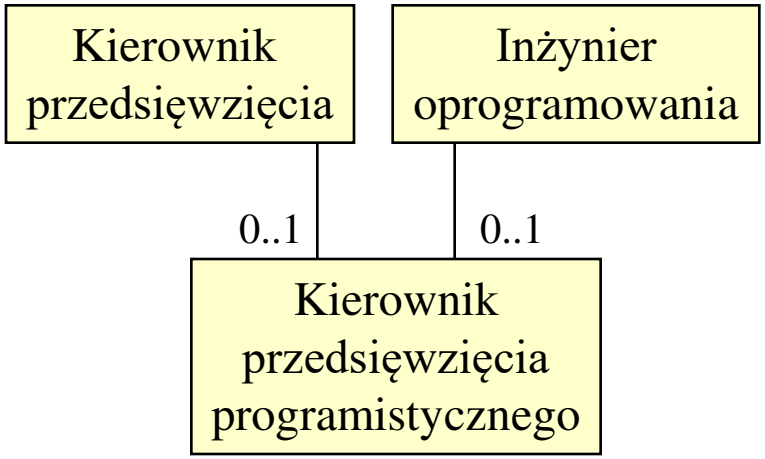
Dostosowanie do ograniczeń i możliwości środowiska implementacji

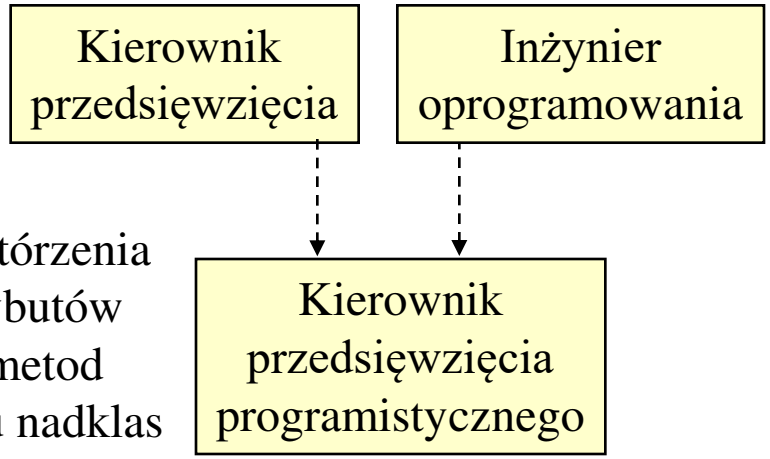
Projektant może zetknąć się z wieloma ograniczeniami implementacyjnymi:

- Brak dziedziczenia wielokrotnego
- Brak dziedziczenia
- Brak metod wirtualnych (przeciążania, przesłaniania)
- Brak złożonych lub powtarzalnych atrybutów
- Brak typów multimedialnych

Przykład: „obejście” braku wielo-dziedziczenia







Powtórzenia atrybutów i metod z obu nadklas

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).

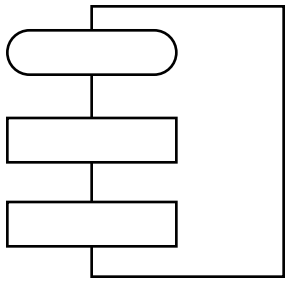
Określenie fizycznej struktury systemu

Obejmuje:

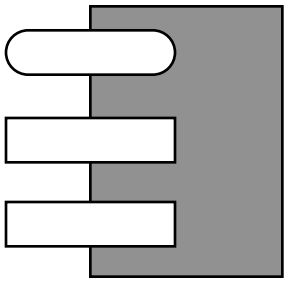
- Określenie struktury kodu źródłowego, tj. wyróżnienie plików źródłowych, zależności pomiędzy nimi oraz rozmieszczenie składowych projektu w plikach źródłowych.
- Podział systemu na poszczególne aplikacje.
- Fizyczne rozmieszczenie danych i aplikacji na stacjach roboczych i serwerach.

Oznaczenia (Booch)

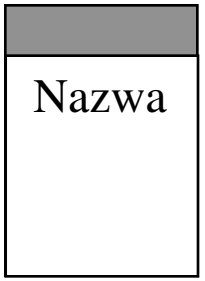
Deklaracja:



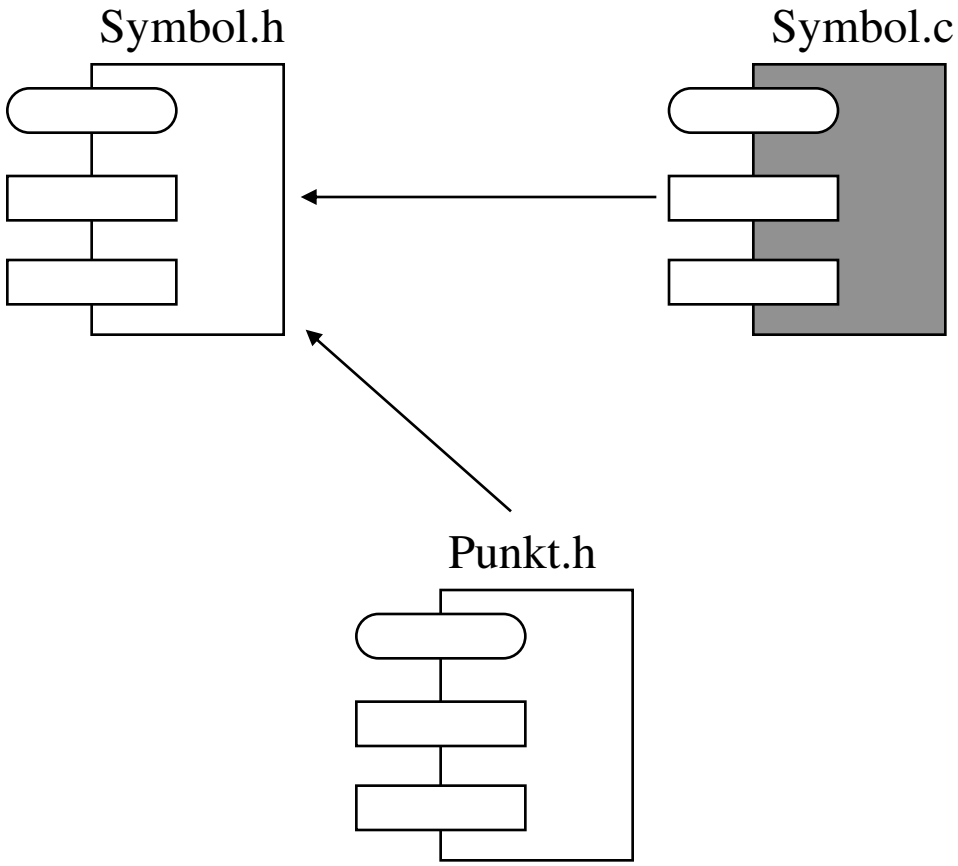
Definicja:



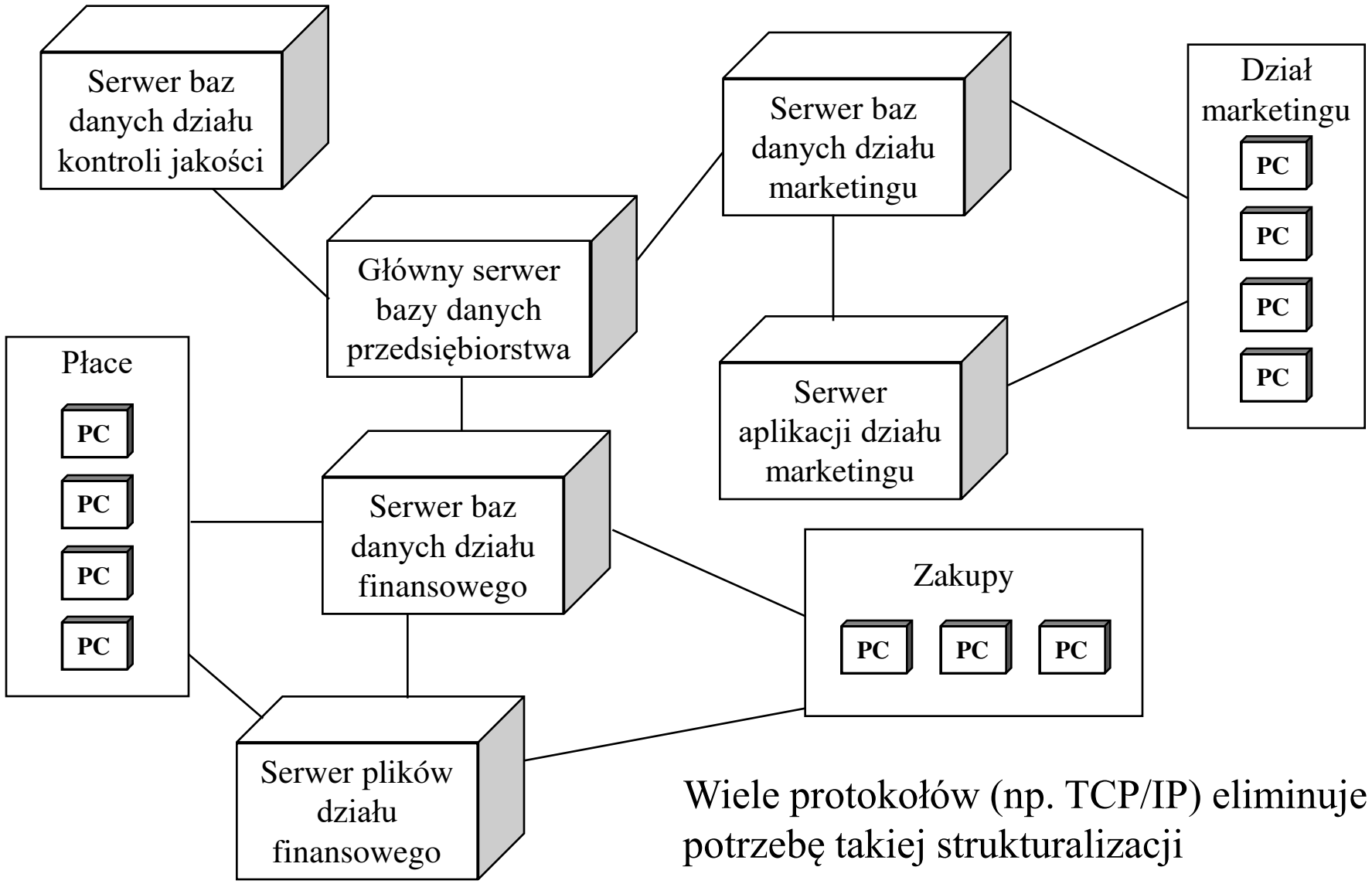
Moduł główny:



Przykład zależności kompilacji dla C++



Graficzny opis sprzętowej konfiguracji systemu



Wiele protokołów (np. TCP/IP) eliminuje potrzebę takiej strukturalizacji

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

Kompletność projektu 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.

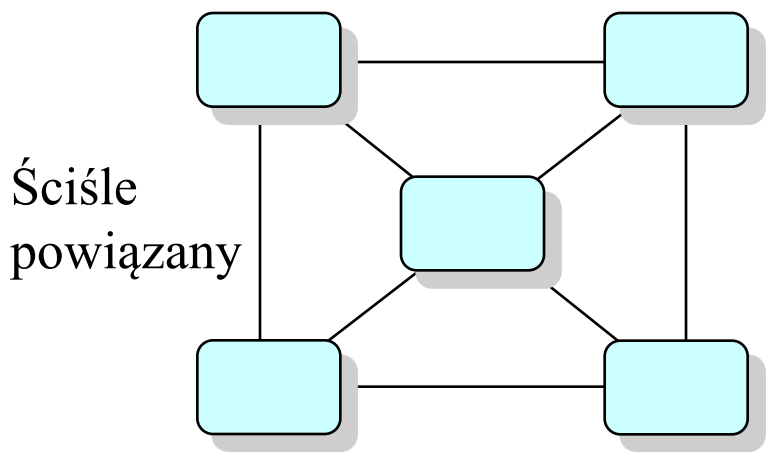
Podział proceduralny (sekwencyjny). Składowe są kolejno uruchamiane. Dane wyjściowe jednej składowej stanowią wejście innej

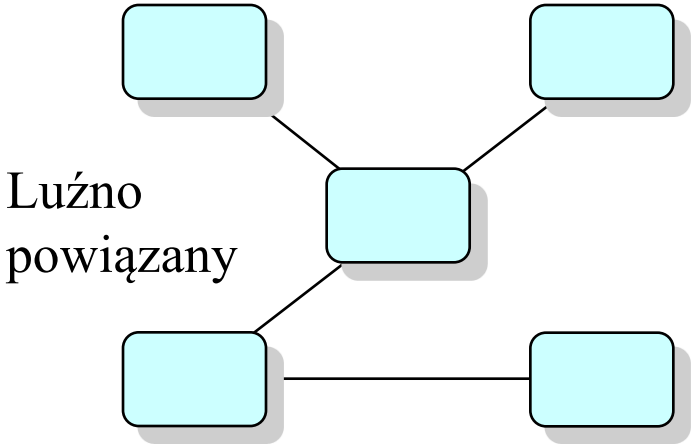
Podział komunikacyjny. Składowe działają na tym samym zbiorze danych wejściowych i wspólnie produkują zestaw danych wyjściowych

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 (z naszym rozumieniem rzeczywistości).
- **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 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 **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

- 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 DWO.
- 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)

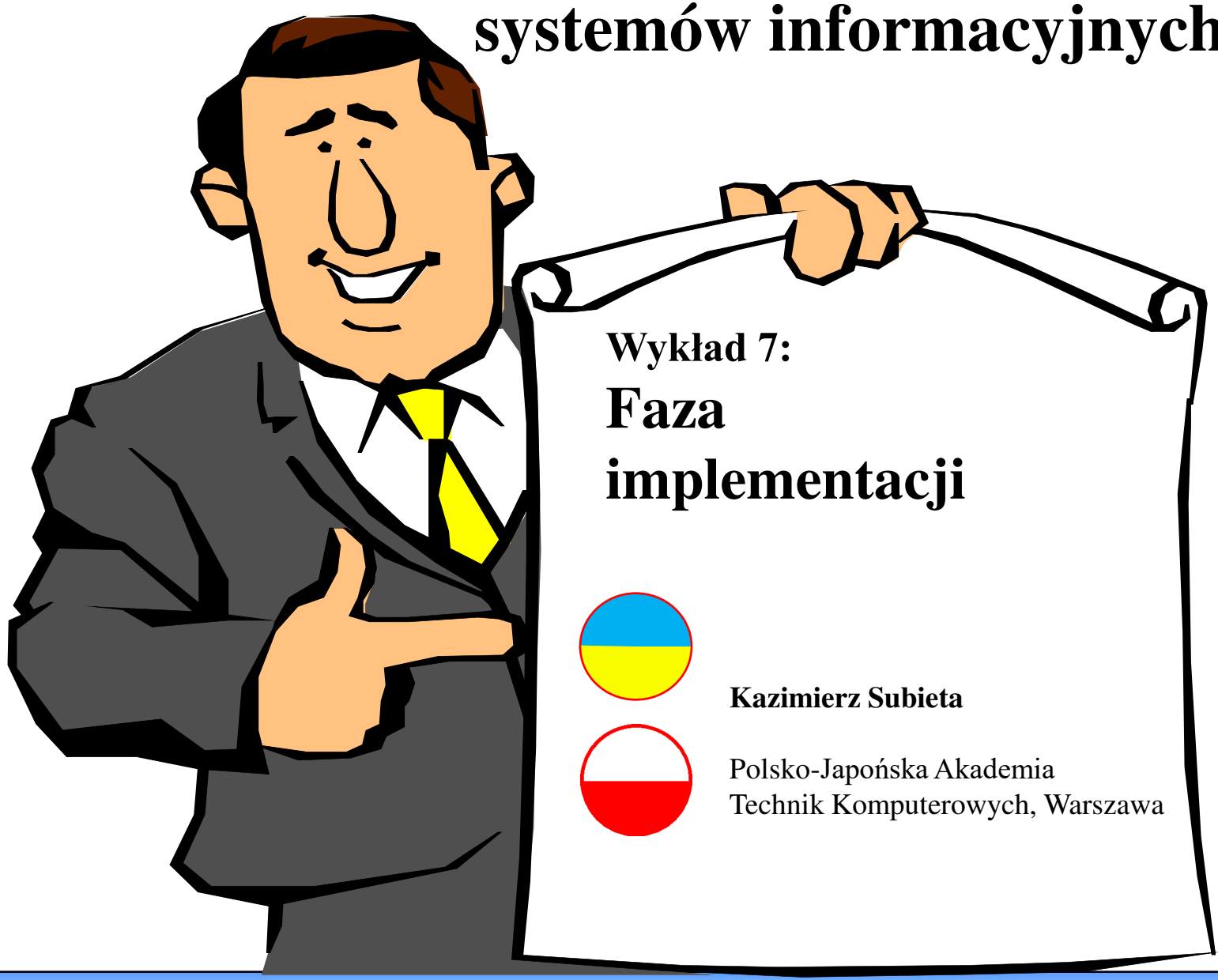
CZĘŚĆ II - SPECYFIKACJA KOMPONENTÓW

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

Budowa i integracja systemów informacyjnych



Wykład 7:
Faza implementacji

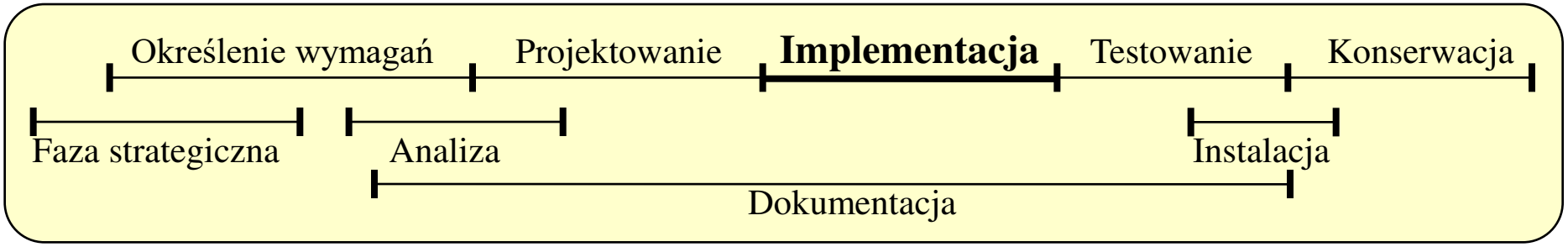
Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Plan wykładu

- Niezawodność oprogramowania
- Unikanie błędów
- Niebezpieczne techniki
- Zasada ograniczonego dostępu
- Mocna kontrola typu
- Tolerancja błędów
- Porównywanie wyników różnych wersji
- Transakcja: jednostka działalności systemu
- Typowe środowiska implementacyjne
- Czynniki sukcesu i rezultaty fazy implementacji

Faza implementacji



Faza implementacji uległa w ostatnich latach znaczącej automatyzacji wynikającej ze stosowania:

- Języków wysokiego poziomu
- Gotowych elementów
- Narzędzi szybkiego wytwarzania aplikacji - RAD (*Rapid Application Development*)
- Generatorów kodu

  Generatory kodu są składowymi narzędzi CASE, które na podstawie opisu projektu automatycznie tworzą kod programu (zwykle tzw. „szkielet” kodu).

Niezawodność oprogramowania

Znaczenie niezawodności:

- Rosnące oczekiwania klientów wynikające m.in. z wysokiej niezawodności sprzętu. Niemniej nadal niezawodność oprogramowania znacznie ustępuje niezawodności sprzętu. Jest to prawdopodobnie nieuchronne ze względu na znacznie mniejszą powtarzalność oprogramowania i stopień jego złożoności.

- Potencjalnie duże koszty błędnych wykonań, wysokie straty finansowe wynikające z błędnego działania funkcji oprogramowania, nawet zagrożenie dla życia.

- Nieprzewidywalność efektów oraz trudność usunięcia błędów w oprogramowaniu. Często pojawia się konieczność znalezienia kompromisu pomiędzy efektywnością i niezawodnością. Łatwiej jednak pokonać problemy zbyt małej efektywności niż zbyt małej niezawodności.

Zwiększenie niezawodności oprogramowania można uzyskać dzięki:
- unikaniu błędów
- tolerancji błędów

Unikanie błędów

Pełne uniknięcie błędów w oprogramowaniu nie jest możliwe.

Można znacznie zmniejszyć prawdopodobieństwo wystąpienia błędu stosując następujące zalecenia:

- Unikanie **niebezpiecznych technik** (np. programowanie poprzez wskaźniki)
- Stosowanie zasady **ograniczonego dostępu** (reguły zakresu, hermetyzacja, podział pamięci, itd.)
- Zastosowanie języków z **mocną kontrolą typów** i kompilatorów sprawdzających zgodność typów
- Stosowanie języków o **wyższym poziomie abstrakcji**
- Dokładne i konsekwentne specyfikowanie **interfejsów pomiędzy modułami** oprogramowania
- Szczególna uwaga na **sytuacje skrajne** (puste zbiory, pętle z zerową ilością obiegów, wartości zerowe, niezainicjowane zmienne, itd.)
- Wykorzystanie **gotowych komponentów** (np. gotowych bibliotek procedur lub klas) raczej niż pisanie nowych (ponowne użycie, *reuse*)
- Minimalizacja różnic pomiędzy modelem pojęciowym i modelem implementacyjnym

Niebezpieczne techniki (1)

- **Instrukcja *go to*** (skocz do) prowadząca do programów, których działanie jest trudne do zrozumienia
- **Stosowanie liczb ze zmiennym przecinkiem**, których dokładność jest ograniczona i może być przyczyną nieoczekiwanych błędów
- **Wskaźniki i arytmetyka wskaźników**: technika wyjątkowo niebezpieczna, dająca możliwość dowolnej penetracji całej pamięci operacyjnej i dowolnych nieoczekiwanych zmian w tej pamięci
- **Obliczenia równoległe**. Prowadzą do złożonych zależności czasowych i tzw. pogoni (zależności wyniku od losowego faktu, który z procesów szybciej dojdzie do pewnego punktu w obliczeniach). Bardzo trudne do testowania. Modne ***wątki*** są niebezpieczne i określane jako *zatrute jabłko*.
- **Przerwania i wyjątki**. Technika ta wprowadza pewien rodzaj równoległości, powoduje problemy j/w. Dodatkowo, ryzyko zawieszenia programu.

Niebezpieczne techniki (2)

- **Rekurencja**. Trudna do zrozumienia, utrudnia śledzenie programu, może losowo powodować przepełnienie stosu wołań (*call stack*)
- **Dynamiczna alokacja pamięci** bez zapewnienia automatycznego mechanizmu odzyskiwania nieużytków (*garbage collection*). Powoduje "wyciekanie pamięci" i w efekcie, coraz wolniejsze działanie i zawieszenie programu.
- Procedury i funkcje, które realizują wyraźnie odmienne zadania w zależności od parametrów lub stanu zewnętrznych zmiennych
- **Niewyspecyfikowane, nieoczekiwane efekty uboczne** funkcji i procedur
- **Złożone wyrażenia bez form nawiasowych**: korzystanie z priorytetu operatorów, który zwykle jest trudny do skontrolowania przez programistę
- **Akcje na danych przez wiele procesów** bez zapewnienia mechanizmu synchronizacji (blokowania, transakcji)

Niektóre z tych technik są przydatne, trzeba je jednak stosować świadomie

Zasada ograniczonego dostępu

Zasada bezpieczeństwa: dostęp do czegokolwiek powinien być ograniczony tylko do tego, co jest niezbędne.
Wszystko, co *może być* ukryte, *powinno być* ukryte.

Programista nie powinien mieć możliwości operowania na tej części oprogramowania lub zasobów komputera, która nie dotyczą tego, co aktualnie robi

Perspektywa (prawa dostępu) programisty powinny być ograniczone tylko do tego kawałka, który aktualnie programuje

Typowe języki programowania niestety nie w pełni realizują te zasady.
Dotyczy to również systemów operacyjnych takich jak Windows.

Zasadę tę realizuje hermetyzacja (znana np. z Modula 2) oraz obiektowość:
- **prywatne** pola, zmienne, metody
- **listy eksportowe** (publiczny interfejs do zasobów do wykorzystania z zewnątrz)
- **listy importowe** (określające zasoby wykorzystywane z zewnętrznych modułów)

Mocna kontrola typu (1)

strong typing

- **Typ** jest wyrażeniem (oraz pewną abstrakcją programistyczną) przypisaną do pewnych bytów programistycznych: zmiennej, danej, obiektu, funkcji, procedury, operacji, metody, parametru, modułu, wyjątku, zdarzenia, …
- Typ specyfikuje rodzaj **wartości**, które może przybierać byt programistyczny, lub „zewnętrzne" cechy tego bytu (interfejs)
- Typ jest formalnym ograniczeniem narzuconym na **budowę** zmiennych lub obiektów. Typy określają również parametry i wyniki procedur, funkcji i metod.
- Typ stanowi ograniczenie **kontekstu**, w którym odwołanie do danego bytu programistycznego może być użyte w programie
- Zasadniczym celem typów jest kontrola **formalnej poprawności programu przed jego uruchomieniem** (mocna) lub po uruchomieniu (słaba)
- Ważnym celem typów jest wspomaganie **modelowania pojęciowego**. Nazwa typu zwykle przenosi nieformalna semantykę zmiennej lub obiektu, któremu jest ten typ przypisany; np. typem zmiennej *D* jest typ *Data*

Mocna kontrola typu (2)

- W językach z mocnym typowaniem każdy deklarowany byt programistyczny musi być wyposażony w deklarację typu.
  - Poprzez deklarację programista wyraża oczekiwania co do roli tego bytu Jest to następnie formalnie sprawdzane podczas kompilacji (lub wykonania).
  - Np. określając typ zmiennej X jako *integer* programista ustala, że ta zmienna ma przechowywać wartości całkowite.
- Mocna typologiczna kontrola poprawności programów okazała się cechą skutecznie eliminującą błędy popełniane przez programistów
  - Według typowych szacunków, po wyeliminowaniu błędów syntaktycznych **około 80%** pozostałych błędów jest wychwytywane przez mocną kontrolę typu
- Języki głównego nurtu (C/C++, Java, C#, Pascal, …) są mocno typowane
- W wielu produktach komercyjnych kontrola typów jest zaniedbywana lub występuje w postaci słabej (Basic, SQL, Pyton, Ruby, PHP, …), tzw. „kaczej”
- **Wady mocnej kontroli typów**: sprzyja monolitycznemu programowaniu w ogromnych blokach, znacznie redukuje możliwość programowania generycznego

Mocna kontrola typu (3)

System mocnej statycznej kontroli typu zawiera następujące elementy:

- **Specyfikacja typów wszystkich zmiennych i obiektów**, które występują w programie, np.:
```
typedef TypPrac=struct{ string nazwisko, int zarobek, Dział pracuje_w, int zarobek_netto()};
TypPrac Pracownik;
```

- **Specyfikacja sygnatur** wszystkich operatorów, procedur, funkcji, metod, np.
```
boolean sprawdź( in TypPrac prac, in TypDział dzial, out int ile_lat_pracuje )
```

- **Specyfikacja interfejsów** modułów, klas i innych hermetyzowanych abstrakcji programistycznych.

- **Dla parametrów procedur, funkcji, metod: określenie które z nich są wejściowe** (czyli komunikowane przez wartość, *call-by-value*), a które wyjściowe (czyli komunikowane przez referencję, *call-by-reference*).

- **Określenie reguł wnioskowania o typie** (*type inference rules*) dla wszystkich konstrukcji składniowych języka programowania, szczególnie dla wyrażeń. W procesie analizy gramatycznej następuje ustalenie jaki będzie wynikowy typ każdej konstrukcji, która w tym programie występuje.

W procesie wnioskowania o poprawności typologicznej ustalone są między innymi typy rzeczywistych parametrów aktualnych poszczególnych wywołań procedur, metod, etc. Sprawdzane jest także, czy nie ma odwołań do bytów, które nie są zadeklarowane lub sa niedostępne. Niezgodności są sygnalizowane jako błędy typu.

Tolerancja błędów

Żadna technika nie gwarantuje uzyskania programu w pełni bezbłędnego.
Tolerancja błędów oznacza, że program działa poprawnie, a przynajmniej sensownie także wtedy, kiedy zawiera błędy.

Tolerancja błędów oznacza wykonanie przez program następujących zadań.

- Wykrycia błędu.
- Wyjścia z błędu, tj. poprawnego zakończenie pracy modułu, w którym wystąpił błąd.
- Ewentualnej naprawy błędu, tj. zmiany programu tak, aby zlikwidować wykryty błąd.

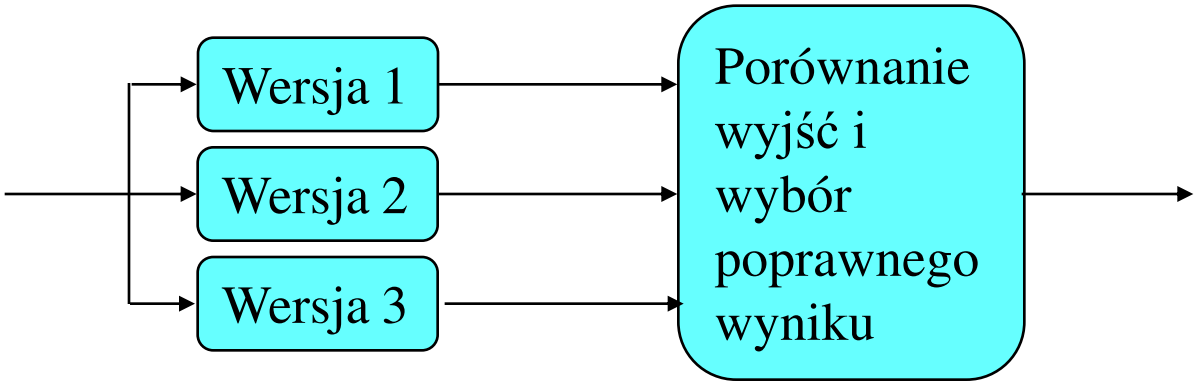
Istotna jest diagnostyka błędu, z dokładnością do linii kodu źródłowego.

Istnieją dwa główne sposoby automatycznego wykrywania błędów:

- sprawdzanie warunków poprawności danych (tzw. **asercje**). Sposób polega na wprowadzaniu dodatkowych warunków na wartości wyliczanych danych, które są sprawdzane przez dodatkowe fragmenty kodu.
- porównywanie wyników różnych wersji modułu.

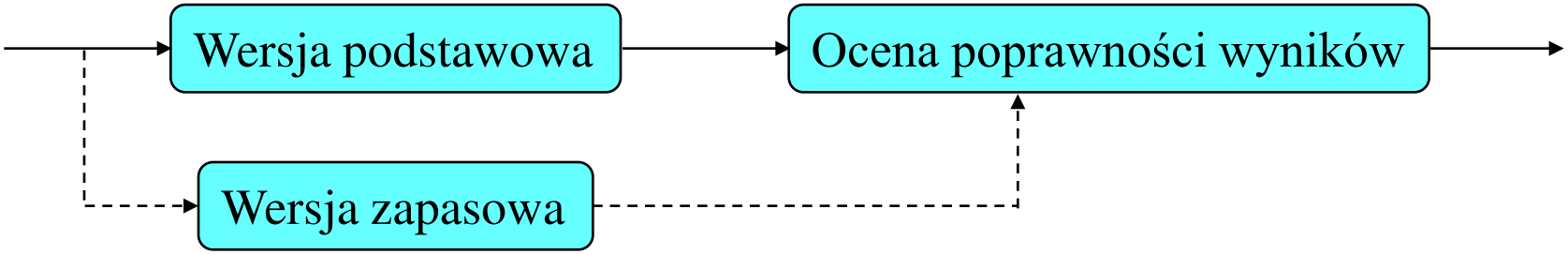
Porównywanie wyników różnych wersji

Programowanie N-wersyjne: ten sam moduł zaprogramowany przez niezależne zespoły programistów. Wersje modułu działają równolegle, wyniki są porównywane:



Narzuty na czas wykonania.

Programowanie z modułem zapasowym:



Po wykryciu błędnego wyniku uruchamia się wersję zapasową.

Transakcje (1)

Jim Grey,
Andreas Reuter, 1981

Po co transakcje? - Współbieżność

Niech procesy A i B działają jednocześnie na zmiennej X zapisanej w bazie danych:

Czas	Proces A	Proces B	Wartość X_A	Wartość X_B	Wartość X w bazie danych
1	Czyta X		5		5
2		Czyta X	5	5	5
3	X :=X+5		10	5	5
4		X := X+1	10	6	5
5	Zapisuje X		10	6	10
6		Zapisuje X	10	6	6 (powinno być 11)

Wynik 6 jest niespójny. Jeżeli te dwa procesy działałyby niezależnie, wynik byłby 11. Brak synchronizacji spowodował zgubienie jednej aktualizacji.

Inny przykład: mamy 4-ch autorów, którzy równolegle aktualizują pewien tekst. Jeżeli nie umówią się, który z nich aktualnie ma prawo wprowadzać poprawki, to część poprawek może zostać zgubiona.

Transakcje umożliwiają zachowanie spójności wielu jednocześnie działających procesów. „Ręczna” synchronizacja lub umawianie się są niepotrzebne.

Transakcje (2)

Po co transakcje? - Przeciwdziałanie awariom

Załóżmy, że mamy system bankowy, w którym operacje na kontach klientów są realizowane w następujący sposób:

1. Klient wczytuje kartę magnetyczną i jest autoryzowany
2. Klient określa sumę wypłaty
3. Konto klienta jest sprawdzane
4. Konto jest zmniejszane o sumę wypłaty
5. Wysyłane jest zlecenie do kasy
6. Kasjerka odlicza sumę wypłaty od stanu kasy
7. Kasjerka wypłaca klientowi pieniądze

Pytanie: co się stanie, jeżeli pomiędzy operacją 4 i 5 wyłączą światło?
Konto zostało zmniejszone, klient nie dostał pieniędzy, dane o aktualizacji przepadły.
Zaczyna się awantura, dyrekcja tłumaczy, że klient może zgłosić pretensje do elektrowni a nie do banku, klient ripostuje, że guzik go obchodzi elektrownia, straszy bank sądem, ...

Transakcje umożliwiają uniknięcie niespójności danych i przetwarzania związanych z dowolnymi awariami sprzętu, błędami w oprogramowaniu, nagłą niedyspozycją personelu, itd.

Transakcja: jednostka działalności systemu

Transakcja umożliwia powrót do stanu sprzed rozpoczęcia jej działania po wystąpieniu dowolnego błędu. Jest to podstawowa technika zwiększenia niezawodności oprogramowania działającego na bazie danych.

Własności transakcji: **ACID**

A **Atomowość** (*atomicity*) - w ramach jednej transakcji wykonują się albo wszystkie operacje, albo żadna

C **Spójność** (*consistency*) - o ile transakcja zastała bazę danych w spójnym stanie, po jej zakończeniu stan jest również spójny. (W międzyczasie stan może być chwilowo niespójny)

I **Izolacja** (*isolation*) - transakcja nie wie nic o innych transakcjach i nie musi uwzględniać ich działania. Czynności wykonane przez daną transakcję są niewidoczne dla innych transakcji aż do jej zakończenia.

D **Trwałość** (*durability*) - po zakończeniu transakcji jej skutki są na trwale zapamiętane (na dysku) i nie mogą być odwrócone przez zdarzenia losowe (np. wyłączenie prądu)

Fakty i mity dotyczące transakcji

- Transakcje są cechą trudną, której nie da się prosto zaimplementować np. w Java
- **Brak transakcji jest wadą dyskwalifikującą system zarządzania bazą danych**
  - Jest to powszechnie negowane w nowych rozwiązaniach, np. NoSQL
  - Popularna teza marketingowa, że systemy przetwarzania dokumentów „nie potrzebują" transakcji jest szkodliwym mitem
- Transakcje w różnych wariantach Web Services są poważnie ograniczone
  - Dominuje rozwiązanie, w którym transakcja blokuje cały serwis, co jest marketingowym *fake*, nie załatwiającym problemu
- Transakcje w systemach rozproszonych, systemach workflow i systemach middleware generują własne problemy, które tylko częściowo są rozwiązane (np. poprzez specjalne protokoły takie jak 2PC)
- Transakcje w systemach interakcyjnych (reagujących natychmiastowo na akcje użytkownika) nie mają dotąd dobrych rozwiązań
- Długie transakcje (trwające dni, tygodnie,…) w ogóle nie mają rozwiązania („isolation levels")

Środowiska języków proceduralnych

- **Jest to tradycyjne i wciąż popularne środowisko implementacji**
- Procesy i moduły wysokiego poziomu mogą odpowiadać całym aplikacjom
- Grupy procedur i funkcji - odpowiadają poszczególnym funkcjom systemu
- Składy/zbiorniki danych w projekcie odpowiadają strukturom danych języka lub strukturom przechowywanym na pliku
- Języki proceduralne dają niewielkie możliwości ograniczenia dostępu do danych
- Poszczególne składowe struktur są dostępne wtedy, gdy dostępna jest cała struktura.
- Istnieją języki o znacznie bardziej rozbudowanych możliwościach ograniczania dostępu do danych, np. Ada, Modula-2, …
- Istnieje możliwość programowania w stylu obiektowym, ale brakuje udogodnień takich jak klasy, dziedziczenie, metody, enkapsulacja, polimorfizm, …

Środowiska języków obiektowych

- Bardzo przydatne jako środowisko implementacji projektu obiektowego, gdyż odwzorowanie pomiędzy modelem projektowym i implementacyjnym jest prostsze
- Z reguły jednak są niewystarczające w przypadku dużych zbiorów przetwarzanych danych i wymagają współpracy z bazą danych
- Większość języków obiektowych to języki hybrydowe, powstające w wyniku dołożenia cech obiektowości do języków proceduralnych. Najbardziej klasycznym przypadkiem takiego rozwiązania jest C++.
- Istnieją zwolennicy i przeciwnicy takiego podejścia. Jak się wydaje, jedyną zaletą języków hybrydowych jest znacznie łatwiejsze ich wypromowanie przy użyciu nie do końca prawdziwych (zwykle naciąganych) twierdzeń o “zgodności”, “kompatybilności” i “przenaszalności”.
- Tendencja do tworzenia języków hybrydowych słabnie, czego dowodem jest Java, C#, Eiffel, Ruby, Python, …

Środowiska relacyjnych baz danych

W tej chwili są to najlepiej rozwinięte środowiska baz danych, pozwalające na zaimplementowanie systemów bazujących na dużych zbiorach danych.

\+

- wielodostęp
- automatyczna weryfikacji więzów integralności
- prawa dostępu dla poszczególnych użytkowników
- wysoka niezawodność
- rozszerzalność (ograniczona)
- możliwość rozproszenia danych
- dostęp na wysokim poziomie (SQL, ODBC, JDBC)

\-

- skomplikowane odwzorowanie modelu pojęciowego
- mała efektywność dla pewnych zadań (kaskadowe złączenia)
- ograniczenia w zakresie typów
- brak hermetyzacji i innych cech obiektowości
- zwiększenie długości kodu, który musi napisać programista

Środowiska obiektowych baz danych

- Zaletą modelu obiektowego baz danych jest wyższy poziom abstrakcji, który umożliwia zaprojektowanie i zaprogramowanie tej samej aplikacji w sposób bardziej skuteczny, konsekwentny i jednorodny
- Uproszczenie i usystematyzowanie procesu projektowania i programowania, minimalizacja liczby pojęć, zwiększenie poziomu abstrakcji, zmniejszenie dystansu pomiędzy fazami analizy, projektowania i programowania oraz zwiększeniu nacisku na rolę czynnika ludzkiego
- W stosunku do modelu relacyjnego, obiektowość wprowadza więcej pojęć, które wspomagają procesy myślowe zachodzące przy projektowaniu i implementacji
- Obiektowe bazy danych (ObjectStore, O2, Versant, Gemstone, Poet, Objectivity/DB, Jasmine, Jade, itd.) osiągają dojrzałość, ale nie pokonały bariery nieufności powszechnego (dużego) klienta
- Na razie są w odwrocie, ale temat jeszcze wróci

Środowiska obiektowo-relacyjnych BD

- Sukces obiektowości w zakresie ideologii i koncepcji spowodował wprowadzenie wielu cech obiektowości, takich jak klasy, metody, dziedziczenie, abstrakcyjne typy danych, do systemów relacyjnych. „Częściowa obiektowość” jest wprowadzona do większości systemów relacyjnych znajdujących się na rynku.

- Takie podejście jest określane jako „hybrydowe” lub „obiektowo-relacyjne”. Ostatnio karierę robi także termin „uniwersalny serwer” (*universal server*), hasło marketingowe eksponujące możliwość zastosowania systemu do przechowywania i przetwarzania obiektów, relacji, danych multimedialnych, itd. Podstawą ideologiczną systemów obiektowo-relacyjnych jest zachowanie sprawdzonych technologii relacyjnych (np. SQL) i wprowadzanie na ich wierzchołku innych własności, w tym obiektowych.

- Systemy obiektowo-relacyjne (Oracle-8, Informix Dynamic Server, itd.), powstają w wyniku ostrożnej ewolucji systemów relacyjnych w kierunku obiektowości. Liczą na pozycję systemów relacyjnych na rynku i odwołują się do swojej klienteli.

Środowiska programów użytkowych

Przykładem może być Microsoft Office, który można przystosować do różnych zastosowań. Np. cechy Microsoft Excel:

- Zawiera pełny proceduralny język Visual Basic dla Aplikacji
- Obejmuje szeroko rozbudowaną bibliotekę obiektów, udostępniającą praktycznie wszystkie możliwości pakietu.
- Pozwala na nagrywanie makrodefinicji w stylu Visual Basic.
- Posiada możliwość dialogowego projektowania interfejsu użytkownika: projektowanie dialogów, menu i pasków narzędziowych, umieszczanie pól dialogowych na arkuszach, definiowanie reakcji na zdarzenia
- Zawiera debugger ułatwiający uruchamianie programów
- Pozwala na dystrybucję aplikacji bez rozprowadzania kodu źródłowego
- Obejmuje rozbudowane możliwości współpracy ze standardami DLL, DDE, OLE, ODBC

Łatwiejsze opracowanie prototypów (montaż z gotowych elementów).

Narzędzia CASE w fazie implementacji

- Programiści mogą bezpośrednio korzystać (przeglądać) diagramy i słownik danych korzystając z narzędzia CASE.
- Niektóre systemy CASE (*lower*) dostarczają generatorów kodu, które generują programy lub ich szkielety. Typowe elementy kodu:
  - skrypty tworzące tabele w bazie danych
  - definicje struktur danych
  - nagłówki procedur i funkcji
  - definicje klas
  - nagłówki metod
- Kod jest uzupełniany wieloma komentarzami na podstawie informacji ze słownika danych. Niektóre narzędzia CASE umożliwiają interfejs do narzędzi RAD.

Czynniki sukcesu i rezultaty fazy implementacji

Czynniki sukcesu:

Wysoka jakość i wystarczająca szczegółowość projektu
Dobra znajomość środowiska implementacji
Zachowanie standardów
Zwrócenie uwagi na środki eliminacji błędów

Rezultaty:

Poprawiony dokument opisujący wymagania
Poprawiony model analityczny
Poprawiony projekt, który od tej pory stanowi już dokumentację techniczną
Kod składający się z przetestowanych modułów
Raport opisujący testy modułu
Zaprojektowana i dostrojona baza danych
Harmonogram fazy testowania

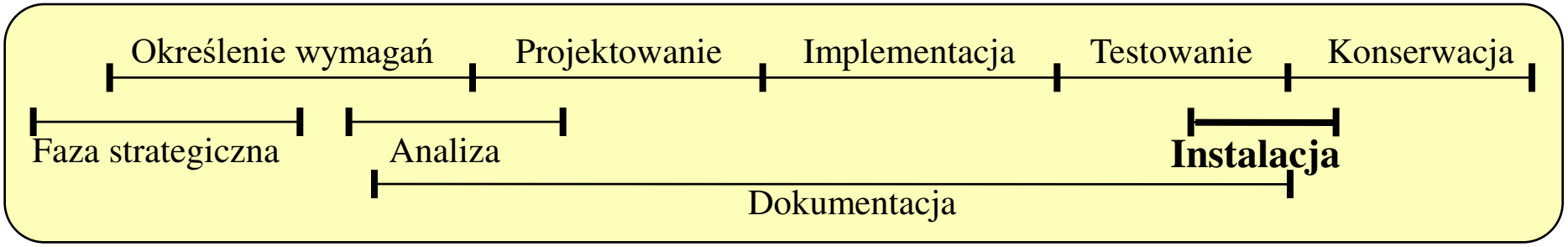
Budowa i integracja systemów informacyjnych

Wykład 8:
Instalacja i konserwacja oprogramowania. Narzędzia CASE

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Instalacja



Na fazę instalacji składają się:

- Szkolenie użytkowników końcowych i administratorów systemu
- Instalacja sprzętu i przeniesienie oprogramowania
- Wypełnienie baz danych
- Nadzorowane korzystanie z systemu, często równoległe z tradycyjnym sposobem pracy
- Usuwanie błędów w oprogramowaniu i dokumentacji użytkowej
- Przekazanie systemu klientowi

Problemy podczas instalacji

Szkolenie użytkowników: zaleca się, aby przeprowadzały je osoby, które były zaangażowane w prowadzenie przedsięwzięcia. Osobom tym będzie łatwiej nawiązać kontakt z przyszłymi użytkownikami.

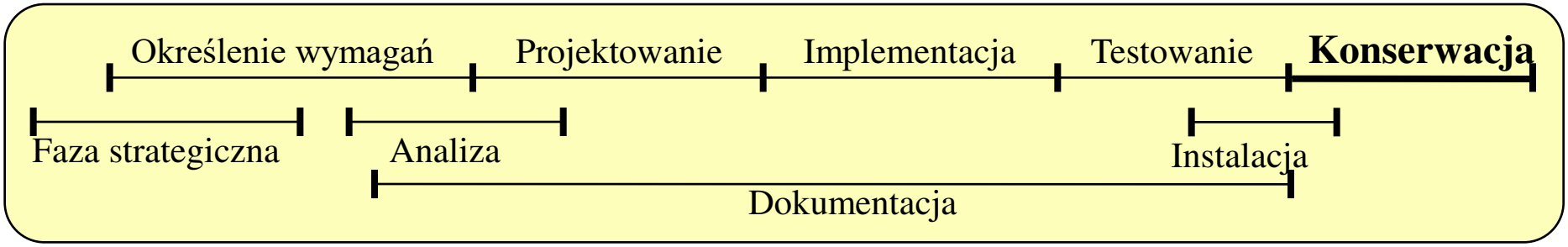
Wypełnienie bazy danych jest często bardzo żmudnym procesem, wymagającym wprowadzenia danych z nośników papierowych. Niekiedy część danych jest w formie elektronicznej - wtedy z reguły potrzebne są specjalne programy konwersji. Konwersja jest łatwiejsza, jeżeli znana jest specyfikacja struktury starej BD.

Ważne jest planowanie i harmonogramowanie prac. W tej fazie pojawia się szereg problemów, np. konieczność usunięcia błędów i wprowadzenia modyfikacji. Z reguły, wykonawcy systemu nie mogą zarezerwować w pełni swojego czasu na prace związane z instalacją. Z drugiej strony, użytkownicy nie mogą zaniechać wykonywania przez nich bieżących prac.

Pewien opór klienta przed zmianą sposobu pracy. Często użytkownicy systemu są to osoby po raz pierwszy stykający się z systemem (inni niż ci, którzy uczestniczyli w poprzednich fazach). Ważne jest uzyskanie ich akceptacji.

Konserwacja oprogramowania

maintenance



(Używa się również terminów “pielęgnacja” oraz “utrzymanie”.)

Konserwacja oprogramowania polega na wprowadzeniu modyfikacji.
Istnieją trzy główne klasy wprowadzanych w oprogramowaniu modyfikacji:

- **Modyfikacje poprawiające**: polegają na usuwaniu z oprogramowania błędów popełnionych w fazach wymagań, analizy, projektowania i implementacji .
- **Modyfikacje ulepszające**: polegają na poprawie jakości oprogramowania.
- **Modyfikacje dostosowujące**: polegają na dostosowaniu oprogramowania do zmian zachodzących w wymaganiach użytkownika lub w środowisku komputerowym.

Modyfikacje ulepszające

- Poprawa wydajności pewnych funkcji
- Poprawa ergonomii interfejsu użytkownika
- Poprawa przejrzystości raportów

Modyfikacje dostosowujące wynikają z:

- Zmiany wymagań użytkowników
- Zmian przepisów prawnych dotyczących dziedziny problemu
- Zmian organizacyjnych po stronie klienta
- Zmian sprzętu i oprogramowania systemowego

Zaleca się, aby wprowadzanie modyfikacji polegało na powrocie do wcześniejszych faz analizy i projektowania, na których rezultaty wpływa przeprowadzana zmiana w oprogramowaniu.

Analiza potrzeby wprowadzania modyfikacji

Analiza powinna uwzględniać:

- Znaczenie wprowadzenia zmiany dla użytkowników
- Koszt wprowadzenia zmiany
- Wpływ zmiany na poszczególne składowe systemu
- Wpływ zmiany na poszczególne składowe dokumentacji technicznej.

Dopiero po dokonaniu oceny zmiany podejmowana jest decyzja o jej ewentualnej realizacji. W przypadku bardzo dużych przedsięwzięć może zostać powołana w tym celu specjalna komisja.

Nie wprowadzać każdej zmiany natychmiast. Zaleca się grupowanie zmian, których wykonanie prowadzi do nowej wersji systemu.

Koszty konserwacji oprogramowania

Występuje tendencja do tego, aby niżej oceniać koszt konserwacji niż koszt wytworzenia oprogramowania.

Zwykle koszty konserwacji wielokrotnie przekraczają koszty wytworzenia.

Niedocenianie nakładów pracy na fazę konserwacji jest jedną z głównych przyczyn opóźnień przedsięwzięć programistycznych.

Obiektywne czynniki wpływające na koszty konserwacji:

- **Stabilność środowiska w którym pracuje system.** Zmiany zachodzące w przepisach prawnych, zmiany struktury organizacyjnej i sposobów działania po stronie klienta prowadzą do zmian wymagań wobec systemu.
- **Stabilność platformy sprzętowej i oprogramowania systemowego**
- **Czas użytkowania systemu**. Całkowite koszty konserwacji rosną, gdy system jest eksploatowany przez dłuższy czas.

Czynniki redukcji kosztów konserwacji (1)

- **Znajomość dziedziny problemu**. Jeżeli analitycy pracujący nad systemem dobrze znają daną dziedzinę problemu, mają mniej trudności z właściwym zebraniem wymagań oraz budową oddającego rzeczywistość modelu.

- **Wysoka jakość modelu i projektu**, w szczególności jego spójność, stopień powiązania składowych oraz przejrzystość.

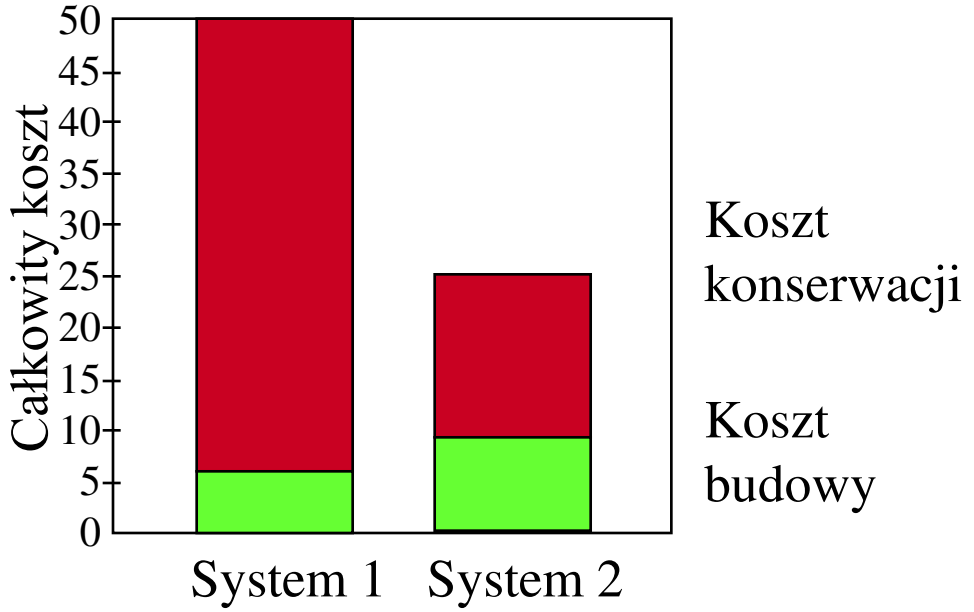
- **Wysoka jakość dokumentacji technicznej**. Powinna ona:
  - w pełni odpowiadać systemowi
  - być wystarczająco szczegółowa
  - być zgodna z przyjętymi w firmie standardami.

- **Stabilność personelu**. Niezależnie od jakości dokumentacji, pewne aspekty systemu są znane tylko osobom bezpośrednio uczestniczącym w realizacji. Jest to tzw. „cicha wiedza” (*tacit knowledge*). Niekoniecznie muszą one same dokonywać modyfikacji, ale mogą istotnie wspomagać konsultacjami.

Czynniki redukcji kosztów konserwacji (2)

- **Środowisko implementacji**. Zaawansowane środowisko implementacji sprzyja skróceniu czasu niezbędnego na wprowadzenie modyfikacji.
- **Niezawodność oprogramowania**. Wysoka niezawodność oprogramowania przekazanego klientowi zmniejsza liczbę modyfikacji.
- **Inżynieria odwrotna**. Pod tym pojęciem rozumie się odtwarzanie dokumentacji technicznej na podstawie istniejącego oprogramowania.
- **Zarządzanie wersjami**.

Wiele działań zmierzających do redukcji kosztów konserwacji musi być podjęte już w fazie budowy systemu.



Czynniki redukcji kosztów konserwacji (3)

Właściwa dekompozycja oprogramowania (podział na części) jest istotnym czynnikiem zmniejszającym koszt konserwacji

- **Podział ma moduły** z dobrze wyspecyfikowanym interfejsem pomiędzy modułami
  - Umożliwia wymianę wadliwego lub przestarzałego modułu
  - Zmniejsza przestrzeń oddziaływania błędu w oprogramowaniu
- **Warstwy systemu** zgodne z ich specjalizacją lub kierunkiem przekazywania sterowania
  - MVC: Model-View-Controller (baza danych, interfejs użytkownika, logika biznesowa)
  - Klient-serwer
  - Warstwy zależne od poziomu abstrakcji oprogramowania
- **Programowanie aspektowe** – niezależne programowanie zaplątanych (*tangled*) cech oprogramowania
  - Dużo szumu, ale niewiele efektów

Narzędzia i metody konserwacji oprogramowania

Mogą być użyte wszystkie narzędzia stosowane do analizy, projektowania, konstruowania i testowania oprogramowania.

W tej fazie pojawiają się także specjalne narzędzia i metody:

- **Narzędzia nawigacyjne.** Umożliwiają inżynierom oprogramowania na szybkie i łatwe odnajdywanie części oprogramowania, które ich interesują. Typowe możliwości: identyfikacja miejsc użycia zmiennych, identyfikacja modułów, które używają danego modułu, wyświetlenie grafu wołań procedur, wyświetlenie drzewa deklaracji typów danych.
- **Narzędzia poprawiania kodu**. Umożliwiają reformatowanie i restrukturalizację kodu programu (*pretty printers*).
- **Narzędzia inżynierii odwrotnej** (*reverse engineering*), umożliwiające odtworzenie bardziej abstrakcyjnej postaci oprogramowania z postaci szczegółowej. Np. odtworzenie dokumentacji technicznej na podstawie kodu programu, odtworzenie źródłowego kodu programu na podstawie kodu skompilowanego (dekompilacja), odtworzeniu modelu logicznego bazy danych na podstawie jej fizycznej struktury, odtworzenie pojęciowego diagramu klas na podstawie deklaracji w języku programowania, itp.

Odtworzenie diagramu klas na podstawie SQL

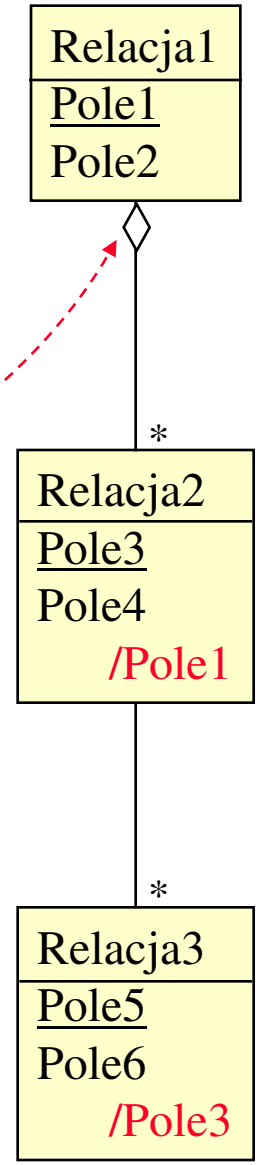
```
CREATE TABLE Relacja1(
         Pole1 TEXT NOT NULL,
         Pole2 TEXT NOT NULL,
         PRIMARY KEY ( Pole1 )
)
CREATE TABLE Relacja2(
         Pole3 TEXT NOT NULL,
         Pole4 DATE NOT NULL,
         Pole1 TEXT NOT NULL,
         PRIMARY KEY ( Pole3 )
)
ALTER TABLE Relacja2 (
         ADD FOREIGN KEY ( Pole1 )
         REFERENCES Relacja1 ( Pole1 )
            ON DELETE CASCADE
)
CREATE TABLE Relacja3(
         Pole5 TEXT NOT NULL,
         Pole6 DECIMAL NOT NULL,
         Pole3 TEXT NOT NULL,
         PRIMARY KEY ( Pole5 )
)
ALTER TABLE Relacja3 (
         ADD FOREIGN KEY ( Pole3 )
         REFERENCES Relacja2 ( Pole3 )
)
```

klucz główny

klucz obcy

klucz główny

klucz obcy

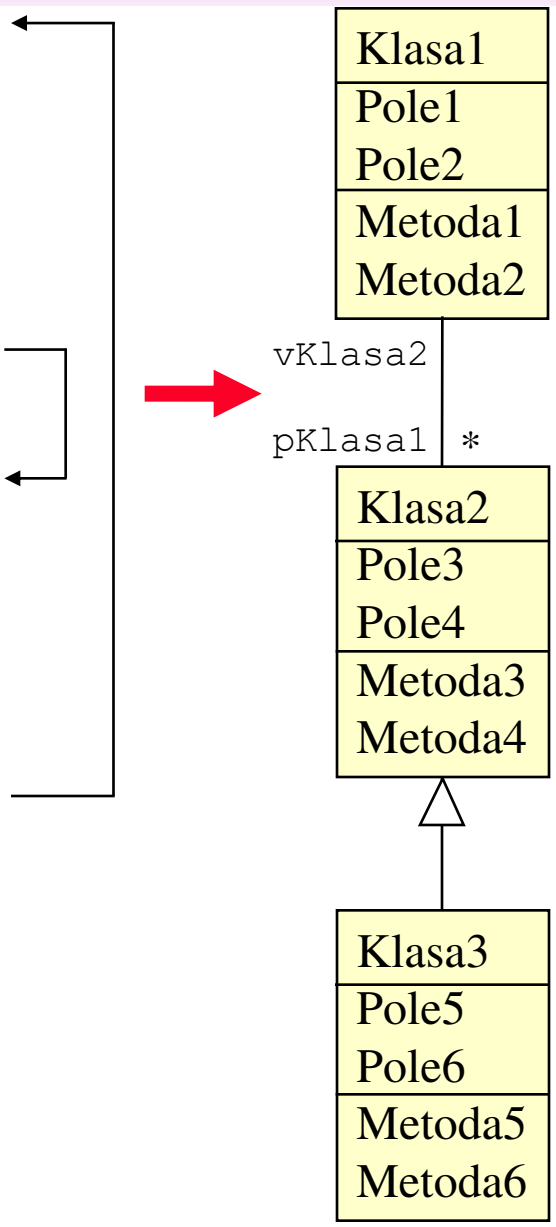


Odtworzenie diagramu klas na podstawie C++

```
class Klasa1
{
private: long Pole2;
         double Pole2;
public:  void Metoda1();
         void Metoda2();
protected:
         Vector<Klasa2*> vKlasa2;
};

class Klasa2
{
private: unsigned short Pole3;
         float Pole4;
public:  void Metoda3();
         void Metoda4();
protected:
         Klasa1* pKlasa1;
};

class Klasa3 : public Klasa2
{
private: long Pole5;
         double Pole6;
public:  void Metoda5();
         void Metoda6();
};
```



W istocie, z podanej specyfikacji w C++ nie wynika, że wskaźniki *pKlasa1* są „bliźniacze” w stosunku do wskaźników *vKlasa2*.

C++ nie definiuje tego ograniczenia.

Faza konserwacji - zgłaszanie problemów

Użytkownicy dokumentują problemy powstałe podczas działania systemu w specjalnym dokumencie **Zgłoszenie Problemu z Oprogramowaniem** (**ZPO**). Nie powinny to być problemy związane z brakiem wiedzy u użytkownika.

Każdy ZPO powinien zgłaszać dokładnie jeden problem. Powinien zawierać:

- Nazwę elementu konfiguracji oprogramowania.
- Wersję lub wydanie tego elementu.
- Priorytet problemu w stosunku do innych problemów (priorytet ma dwa wymiary: **krytyczność** - na ile problem jest istotny dla funkcjonowania systemu, oraz **pilność** - maksymalny czas usunięcia problemu).
- Opis problemu.
- Opis środowiska operacyjnego.
- Zalecane rozwiązanie problemu (o ile użytkownik jest je w stanie określić).

Problemy mogą wynikać z wielu powodów: błędy w oprogramowaniu, brak funkcji, które okazały się istotne, ograniczenia, których nie uwzględniono lub które się pojawiły, zmiany w środowisku systemu. Stąd każdy ZPO powinien być oceniony odnośnie odpowiedzialności za problem (kto ma ponosić koszt).

Faza konserwacji - zlecanie zmian

ZPO jest analizowane przez producenta oprogramowania. Może ono być odrzucone, mogą być podjęte negocjacje z klientem celem ustalenia warunków (np. finansowych, czasowych) wprowadzenia zmian, lub zgłoszenie może być zaakceptowane. Po tych ustaleniach tworzony jest dokument **Zlecenie Zmiany w Oprogramowaniu** (**ZZO**), który zawiera:

- Nazwę elementu konfiguracji oprogramowania.
- Wersję lub wydanie tego elementu.
- Wymagane zmiany.
- Priorytet zlecenia (krytyczność, pilność).
- Personel odpowiedzialny.
- Szacunkową datę początku, datę końca i pracochłonność w osobo-dniach.

ZZO powinien zawierać sekcje dotyczące:

- Zmian w dokumentach wymagań (użytkownika, na oprogramowanie)
- Zmian w dokumentach projektowych (sekcja dla każdego dokumentu)
- Zmian w dokumentach dotyczących zarządzania, testowania, zapewniania jakości.

Faza konserwacji - ocena efektu zmian

ZZO jest realizowane przez wyznaczony personel. Nanoszone są zmiany do kodu i odpowiednich dokumentów opisujących oprogramowanie. Dokument **Raport z Modyfikacji Oprogramowania** (**RMO**) określa wszystkie zmiany w kodzie i w dokumentach. Zmiany są oceniane; ocenie podlegają następujące aspekty:

- **Wydajność** (szybkość) oprogramowania
- **Zużycie zasobów** (pamięci dyskowej, czasu procesora, pamięci operacyjnej.
- **Stopień powiązania** elementów systemu (kohezji).
- **Niezależność zmienianego elementu** (od pozostałych elementów oprogramowania).
- **Złożoność** (na ile została zwiększona).
- **Spójność** (odstępstwa od reguły spójności, np. interfejsów użytkownika).
- **Przenaszalność** (czy oprogramowanie będzie działać na innej platformie).
- **Niezawodność** (czy zmiana mogła spowodować jej obniżenie).
- **Podatność na konserwację** (inaczej pielęgnacyjność - czy nie została obniżona. Oceniane są wszelkie ewentualne odstępstwa od przyjętych standardów.)
- **Bezpieczeństwo** (czy zmiana nie tworzy zagrożenia dla biznesu klienta)
- **Ochrona** (czy zmiana nie powoduje wyłomów w ochronie systemu).

Kluczowe czynniki sukcesu fazy konserwacji

- Wysoka jakość definicji wymagań, modelu i projektu
- Dobra znajomość środowiska implementacji
- Właściwa motywacja osób wykonujacych konserwację oprogramowania
- Właściwe oszacowanie kosztów konserwacji

Podstawowy rezultat:

poprawiony kod, projekt, model i specyfikacja wymagań.

Narzędzia CASE

computer assisted/aided software/system engineering

Pod tym hasłem kryją się wszelkie narzędzia wykorzystywane w trakcie prac nad oprogramowaniem: kompilatory, debuggery, edytory tekstu, narzędzia harmonogramowania przedsięwzięć, arkusze kalkulacyjne.

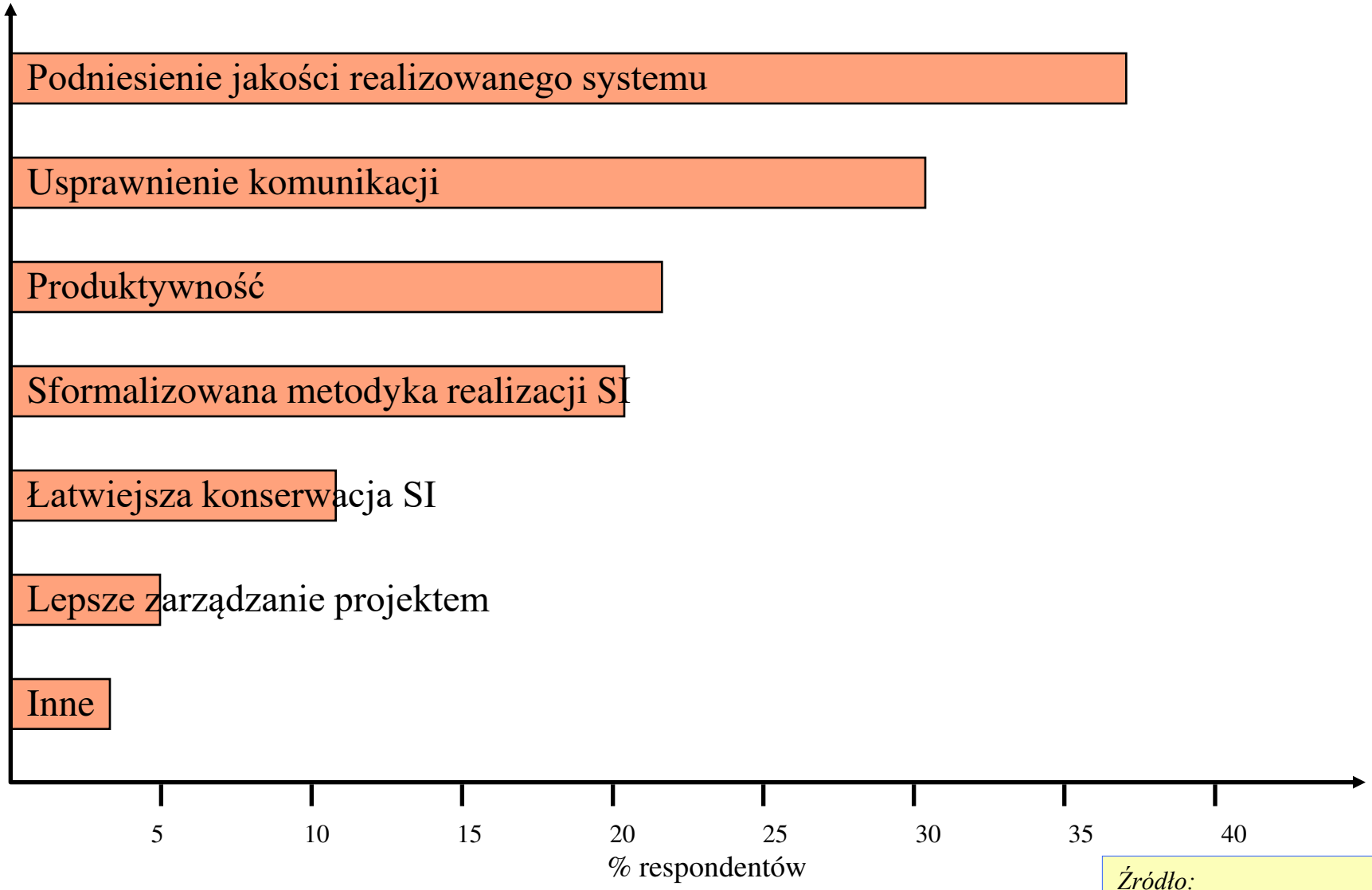
Tradycyjnie pod pojęciem CASE rozumie się narzędzia, które:

- Wspomagają ogólnie rozumiane wytwarzanie oprogramowania
- Koncentrują się na fazach analizy i projektowania oraz bezpośrednim wykorzystaniu wyników tych faz w implementacji.

Dwie główne grupy narzędzi CASE:

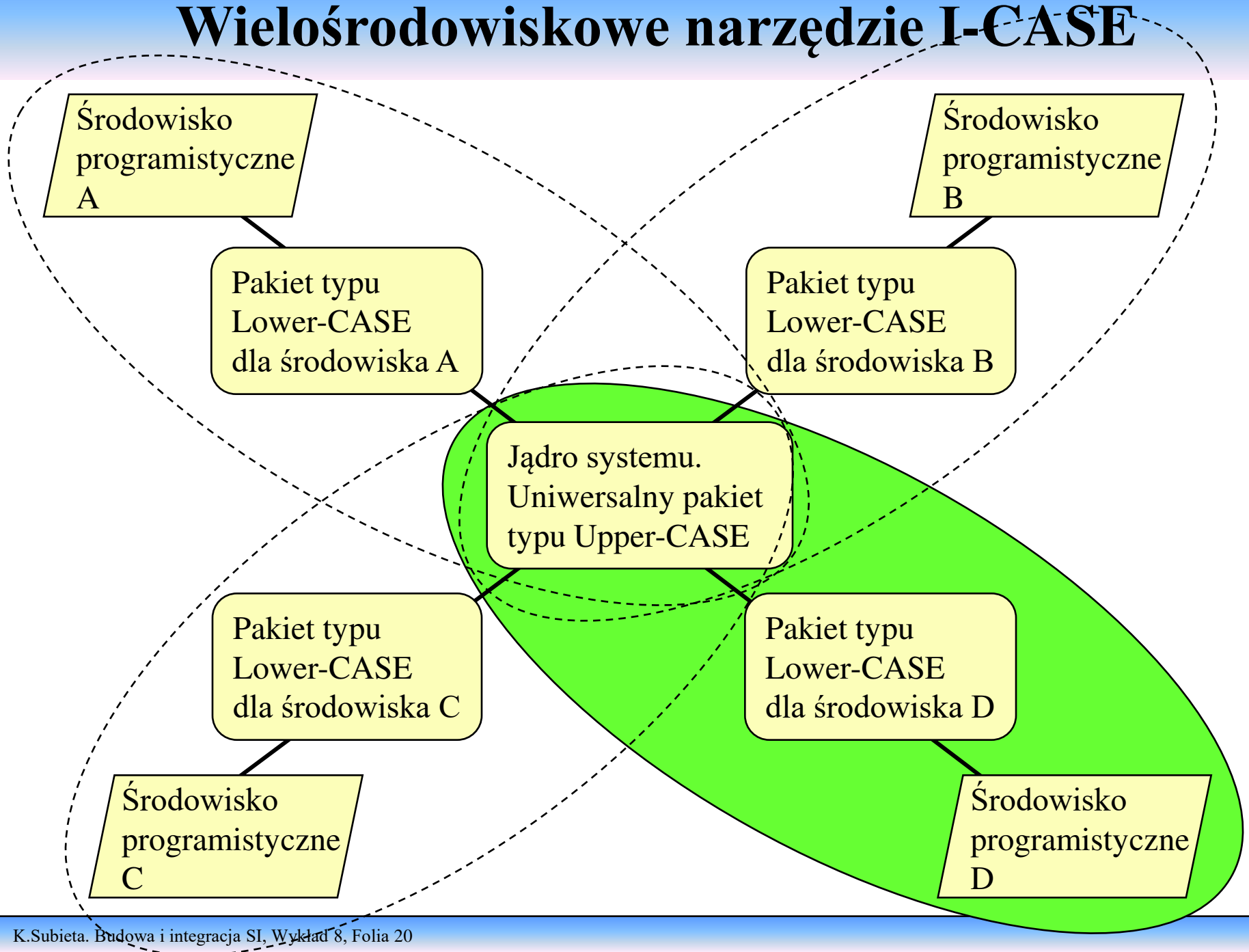
- **Upper-CASE**: wspomaganie wczesnych faz prac nad oprogramowaniem, w szczególności fazy analizy (potrzeby analityków i projektantów). Narzędzia te nie są związane z konkretnym środowiskiem implementacyjnym.
- **Lower-CASE**: wspomaganie faz projektowania i implementacji (potrzeby programistów). Narzędzia te są z reguły ściśle związane z konkretnym środowiskiem implementacji.

Korzyści ze stosowania narzędzi CASE



Źródło:
CASE RESEARCH CORP.

Wielośrodowiskowe narzędzie I-CASE



Aspekty narzędzi CASE

Narzędzia CASE stosują różne techniki wizualizacji projektów, w szczególności notacje diagramów encja-związek (ER), UML i inne.

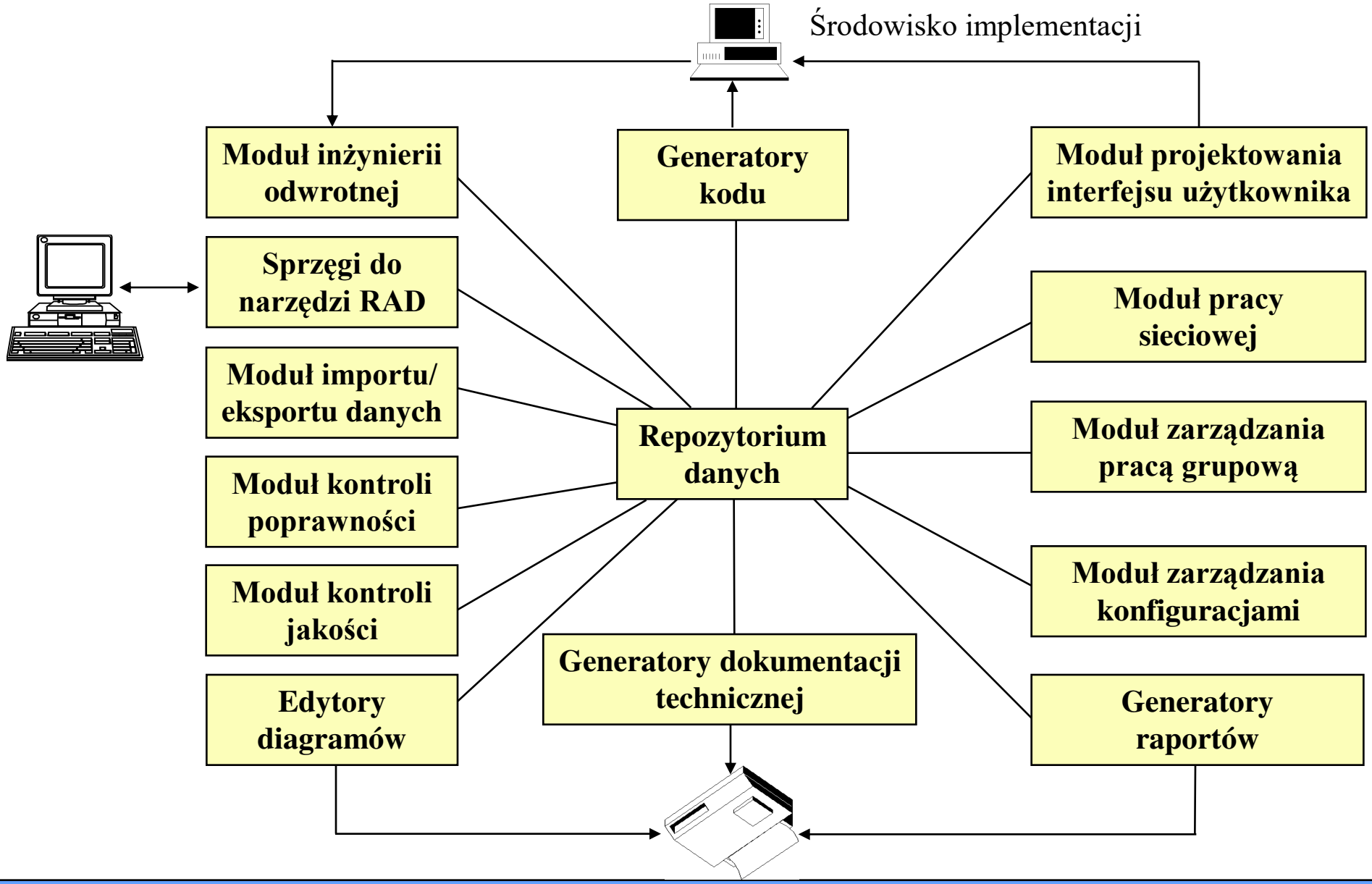
Obecnie większość producentów określa swoje środowiska jako I-CASE (Integrated-CASE). Są to narzędzia łączące w sobie możliwości Lower-CASE i Upper-CASE.

Stosunek narzędzi CASE do konkretnych metodyk i notacji jest dość różny. Istnieje grupa narzędzi uniwersalnych, które umożliwiają pracę z wieloma notacjami i wieloma metodykami. Istnieją również narzędzia CASE przypisane do konkretnych produktów, np. Oracle CASE. Wiele narzędzi CASE łączy elementy znane z wielu metodyk z własnymi pomysłami.

Przykłady narzędzi CASE (są ich dziesiątki):
Oracle CASE, EasyCASE, CASE 4.0, ObjectiF, Select OMT Professional, System Architect, ObjectTeam, Paradigm Plus, Rational Rose, Select Enterprise

Wiele narzędzi jest niczym więcej niż wyspecjalizowanymi edytorami graficznymi.

Składowe narzędzi CASE



Funkcje edytorów notacji graficznych

- Tworzenie i edycja diagramów wykorzystywanych w fazach określania wymagań.
- Tworzenie i edycja powiązań pomiędzy poszczególnymi symbolami i diagramami oraz nawigowanie po sieci powiązanych diagramów.
- Wydruk diagramów.

Ocena notacji graficznych:

- Ergonomia pracy. Diagramy graficzne są jednym z podstawowych narzędzi pracy w fazach analizy i projektowania. Powinny one pozwalać analitykom i projektantom skupić się na pracy, a nie na “zmaganiach” z edytorem.
- Możliwość kontrolowania ilości informacji prezentowanej w sposób graficzny.
- Jakość i możliwość formatowania wydruków.
- Wykrywanie na bieżąco konstrukcji niepoprawnych.
- Zapewnienie spójności informacji umieszczonych na różnych diagramach.

Repozytorium narzędzia CASE (słownik)

Jest to baza danych o realizowanym projekcie oraz narzędzia służące do jej edycji i przeglądania.

Podstawowe funkcje repozytorium:

- Wprowadzenie oraz edycja specyfikacji modelu i projektu, a także innych informacji związanych z przedsięwzięciem.
- Wyszukiwanie pożądanej informacji

W wielu narzędziach CASE repozytorium jest przechowywany w sposób umożliwiający dostęp z programów zewnętrznych pisanych np. w Visual Basic, SQL, itd.

W niektórych narzędziach CASE użytkownik ma możliwość rozbudowania struktury repozytorium oraz wprowadzania własnych funkcji działających na repozytorium.

Pozostałe moduły narzędzi CASE (1)

- **Moduł kontroli poprawności**
  Pewne błędy mogą być wykrywane na bieżąco w trakcie edycji diagramów i słownika danych., np. uczynienie klasy swoja własną specjalizacją.

- **Moduł kontroli jakości**
  Pewne systemy pozwalają na automatyczną ocenę pewnych miar jakości projektu. Dotyczy to szczególnie złożoności oraz stopnia powiązania składowych.

- **Generator raportów**
  Służy do przygotowania raportów na podstawie zawartości słownika danych. Niektóre raporty są parametryczne. Narzędzia CASE zawierają sporo gotowych generatorów raportów. Niektóre z nich pozwalają na definiowanie własnych.

- **Generator dokumentacji technicznej**
  Moduł służący do przygotowania dokumentacji technicznej złożonej z szeregu diagramów. Swobodne formatowanie dokumentów. Przykładowe dokumenty. Łatwe i efektywne uaktualnienie dokumentacji po dokonaniu zmian w projekcie.

Pozostałe moduły narzędzi CASE (2)

- **Generatory kodu**
  Narzędzia służące do generacji kodu w rozmaitych językach programowania. Często generują szkielety, które muszą być uzupełnione przez użytkownika dodatkowym kodem. Wygenerowany kod jest uzupełniony o komentarze i inne informacje. Może także zawierać pewne elementy do modyfikacji. Nazwy użyte w projekcie przechodzą do wynikowego kodu (ewentualnie są skracane).

- **Moduł zarządzania wersjami**
  Umożliwia kontrolę różnych wersji projektu powstających ze względu na konieczność grupowego wprowadzania zmian oraz wskutek wielu środowisk informatycznych (sprzęt i oprogramowania) oraz różnych zastosowań.

- **Moduł projektowania interfejsu użytkownika**
  Dotyczy projektowania dialogów, okien, menu. Zaletą jest wykorzystanie informacji znajdujących się w słowniku danych. Pozwala to np. na automatyczne wygenerowanie dialogu do edycji pewnej struktury danych.
  Integracja z RAD.

- **Moduł inżynierii odwrotnej**

Ocena narzędzi CASE

Kryteria:

- Zakres oferowanych funkcji i ich zgodność z potrzebami firmy
- Koszt
- Niezawodność
- Opinia o producencie i dystrybutorze
- Dostępność na rynku pracy specjalistów znających dany pakiet
- Stopień zintegrowania z przyjętym środowiskiem programistycznym
- Wielośrodowiskowość
- Koszt szkoleń
- Koszt dostosowania sprzętu do wymagań pakietu

Obecnie narzędzia są zwykle oparte o koncepcje obiektowe, ale wiele z nich jest przypisanych do konkretnych systemów relacyjnych lub internetowych (HTML, XML) i nie ma wiele wspólnego z obiektowością

Przyczyny trudności z narzędziami CASE

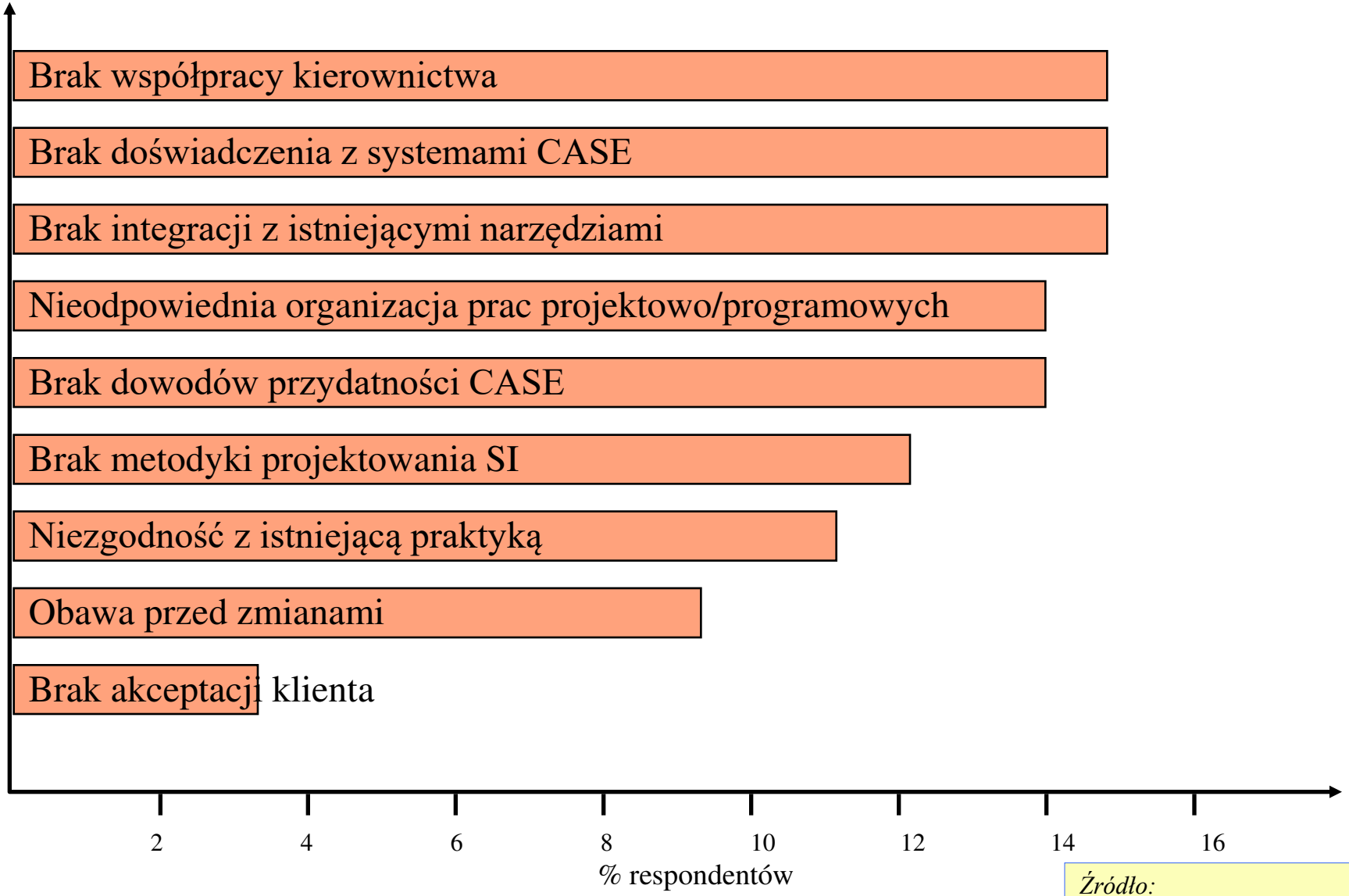
- **Traktowanie narzędzi CASE wyłącznie jako generatorów kodu**. Nie jest to efektywne przy braku rzetelnego podejścia do analizy i projektowania:
  - nakłady na implementację stanowią tylko ok. 15-30% całych nakładów
  - koszt błędów popełnionych w fazie implementacji jest stosunkowo niewielki
  - istnieją inne, tańsze narzędzia programistyczne (RAD)
- **Nieznajomość metodyki analizy i projektowania.** Narzędzia CASE nie zwalniają z myślenia, wiedzy i doświadczenia.
- **Niewłaściwa organizacja i zarządzanie przedsięwzięciem**. Nieuporządkowanie prac, brak planu, brak właściwych ocen, brak monitorowania postępu, itd.
- **Zbyt wysokie oczekiwania w stosunku do narzędzia CASE**. Może ono zredukować koszty co najwyżej o 50%, koszt wdrożenia jest wysoki, efekty pojawiają się z pewnym opóźnieniem, wymaga dyscypliny w przedsięwzięciu.

Narzędzie CASE nie przesądza w auto-magiczny sposób o powodzeniu projektu. Zastosowanie obiektowego CASE niekoniecznie oznacza “nowoczesność” projektu. Stosowanie narzędzi CASE często przynosi znikome efekty.

Wdrażanie i konfigurowanie pakietu CASE

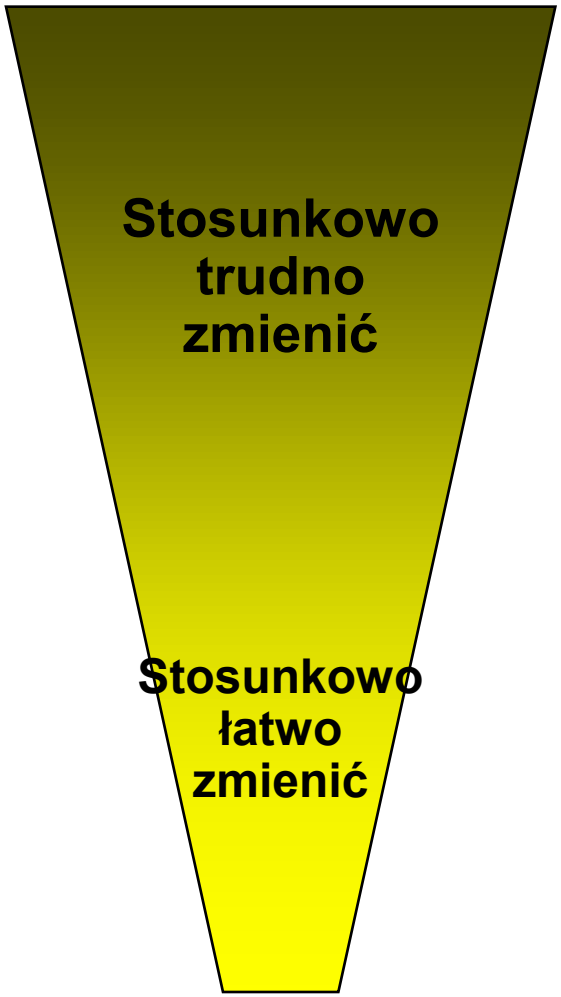
- Skonfigurowanie pakietu stosownie do potrzeb i zgodnie ze standardami już stosowanymi w firmie. Obejmuje skonfigurowanie słownika danych, zdefiniowanie niezbędnych raportów, zdefiniowanie dokumentów, które będą generowane za pomocą generatora dokumentacji technicznej.
- Dostosowanie standardów firmy do nowej technologii.
- Szkolenie pracowników w zakresie metodyki analizy i projektowania wspomaganej przez pakiet.
- Szkolenie pracowników w zakresie obsługi pakietu.
- Odtworzenie dokumentacji technicznej poprzednich przedsięwzięć. Obejmuje funkcje inżynierii odwrotnej oraz możliwości importu danych ze standardowych formatów.
- Wykonanie pilotowego projektu (projektów) z wykorzystaniem narzędzia CASE, często równolegle do stosowanych wcześniej metod.

Przeszkody we wdrażaniu CASE



Źródło:
CASE RESEARCH CORP.

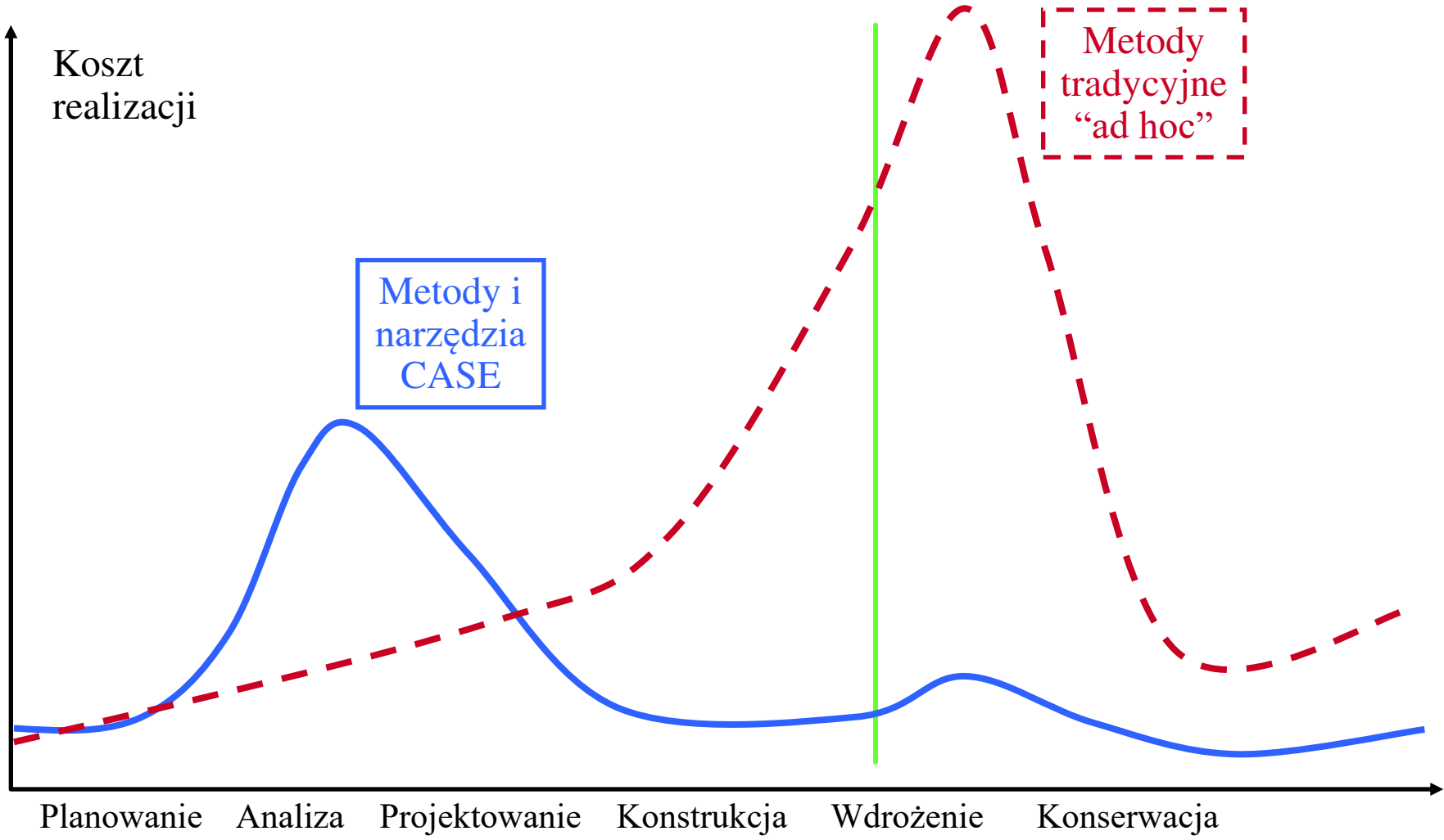
Skala trudności zmian



- Metody rozwiązywania problemów
- Organizacje prac projektowo-programowych
- Podejście do projektowania
- Języki programowania
- Oprogramowanie narzędziowe
- System operacyjny
- Sprzęt komputerowy w ramach jednej rodziny
- Standardy dokumentacyjne
- Styl programowania

Źródło:
Auerbach

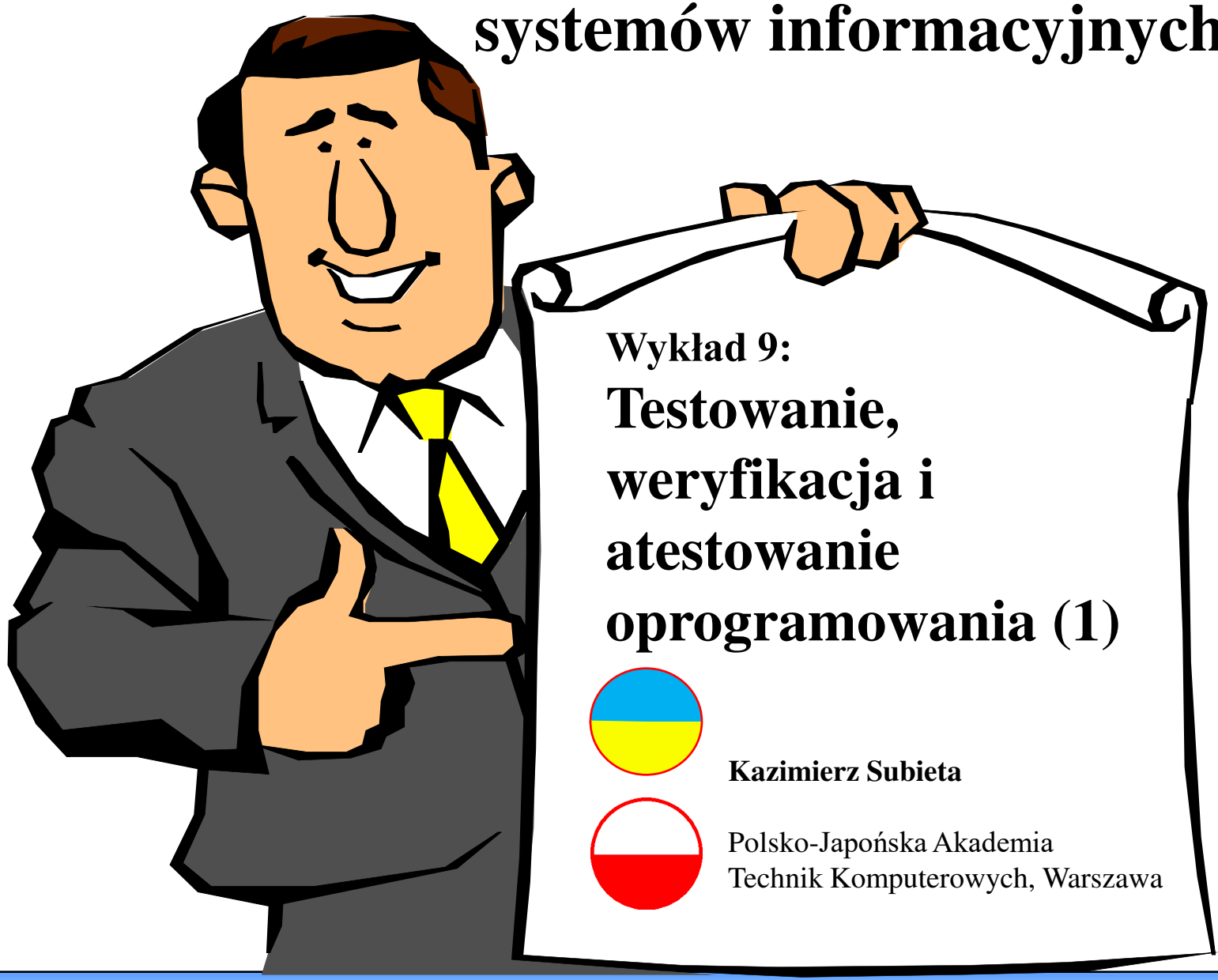
Rozkład kosztów realizacji SI



Kierunki rozwoju narzędzi CASE

- Integracja poszczególnych elementów CASE
- Semantyka modeli wykorzystywanych w CASE
- Interfejs graficzny w CASE
- Inżynieria odwrotna
- Integracja z otoczeniem programistycznym
  - narzędzia, metodyki, zarządzania projektami, ORSZBD, OSZBD
- Projektowanie systemów:
  - Klient-Serwer
  - Obiektowych
  - Komponentowych
  - Multimedialnych
  - Eksperckich
- Projektowanie rozproszonych baz danych
- Dostosowanie narzędzia CASE do projektu (*customization*)
- Elementy sztucznej inteligencji (?)

Budowa i integracja systemów informacyjnych

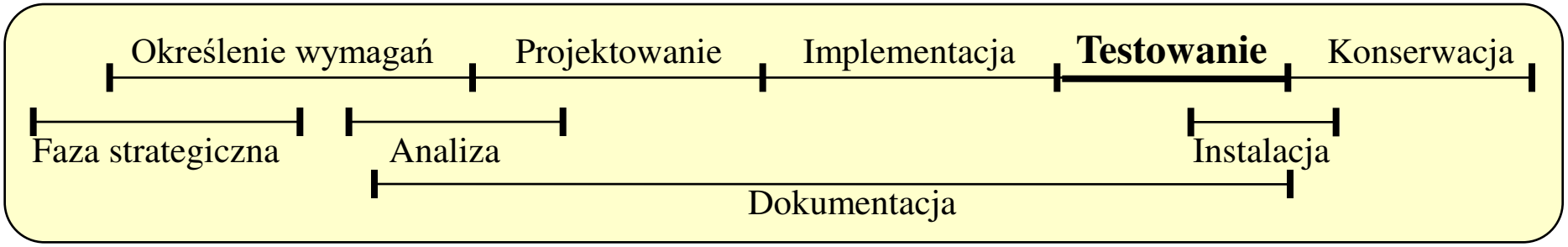


Wykład 9:
Testowanie, weryfikacja i atestowanie oprogramowania (1)

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Faza testowania



Rozróżnia się następujące terminy:

Weryfikacja (*verification*) - testowanie zgodności systemu z wymaganiami zdefiniowanymi w fazie określenia wymagań.

Atestowanie (*validation*) - ocena systemu lub komponentu podczas lub na końcu procesu jego rozwoju na zgodności z wyspecyfikowanymi wymaganiami. Atestowanie jest więc weryfikacją końcową.

Dwa główne cele testowania:

- wykrycie i usunięcie błędów w systemie
- ocena niezawodności systemu

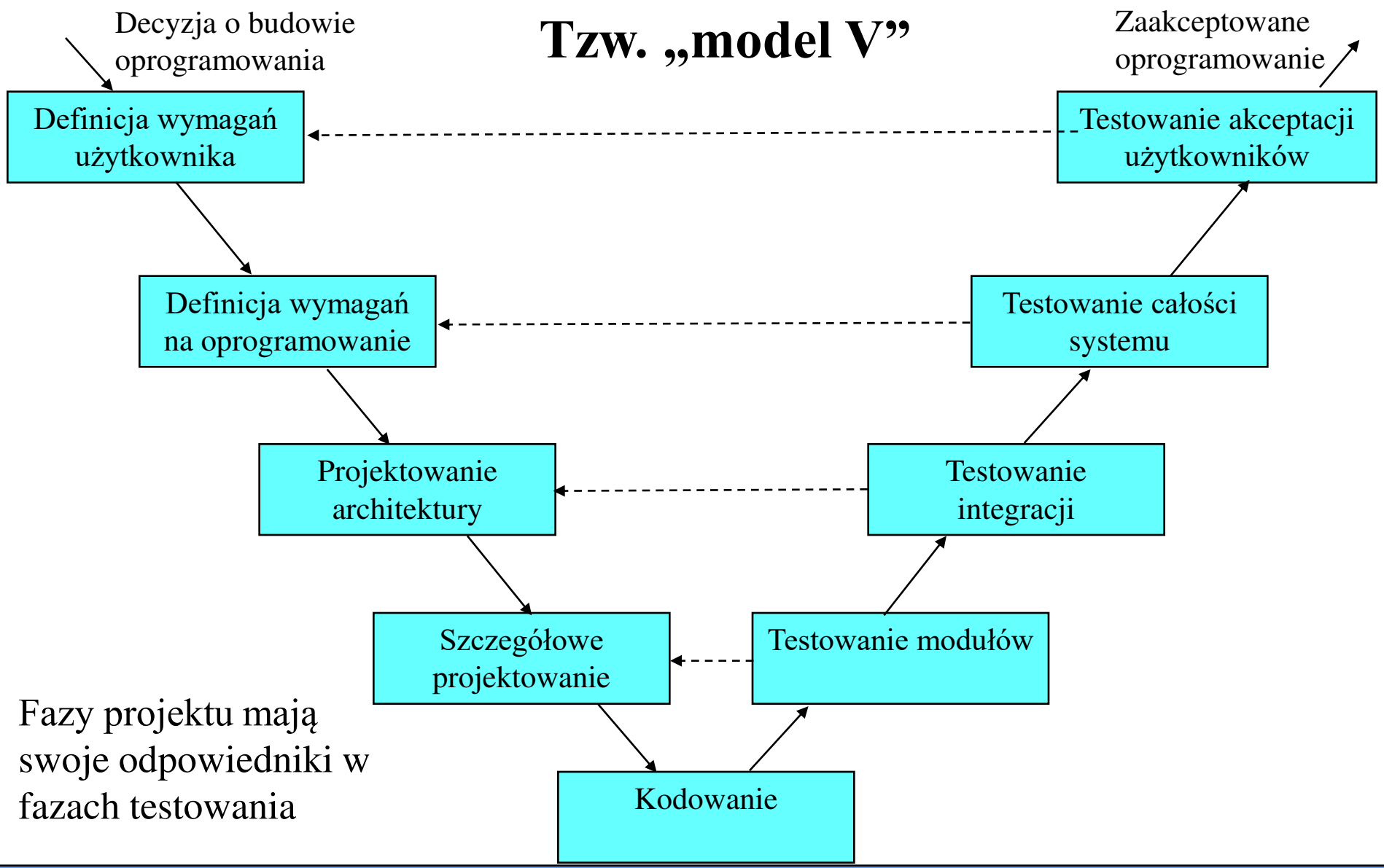
Weryfikacja oznacza...

- Przeglądy, inspekcje, testowanie, sprawdzanie, audytowanie lub inną działalność ustalającą i dokumentującą czy składowe, procesy, usługi lub dokumenty zgadzają się z wyspecyfikowanymi wymaganiami.
- Oceny systemu lub komponentu mające na celu określenie czy produkt w danej fazie rozwoju oprogramowania spełnia warunki zakładane podczas startu tej fazy.

Weryfikacja włącza następujące czynności:

- Przeglądy techniczne oraz inspekcje oprogramowania.
- Sprawdzanie czy wymagania na oprogramowanie są zgodne z wymaganiami użytkownika.
- Sprawdzanie czy komponenty projektu są zgodne z wymaganiami na oprogramowanie.
- Testowanie jednostek oprogramowania (modułów).
- Testowanie integracji oprogramowania, testowanie systemu.
- Testowanie akceptacji systemu przez użytkowników
- Audyt.

Związek faz projektu z fazami testowania



Fazy projektu mają swoje odpowiedniki w fazach testowania

Przeglądy oprogramowania

reviews

Przegląd jest procesem lub spotkaniem, podczas którego produkt roboczy lub pewien zbiór produktów roboczych jest prezentowany dla personelu projektu, kierownictwa, użytkowników, klientów lub innych zainteresowanych stron celem uzyskania komentarzy, opinii i akceptacji.

Przeglądy mogą być formalne i nieformalne.
Formalne przeglądy mogą mieć następującą postać:

- **Przegląd techniczny**. Oceniają elementy oprogramowania na zgodność postępu prac z przyjętym planem. (Szczegóły można znaleźć w ANSI/IEEE Std 1028-1988 „*IEEE Standard for Reviews and Audits*").
- **Przejście** (*walkthrough*). Wczesna ocena dokumentów, modeli, projektów i kodu. Celem jest zidentyfikowanie defektów i rozważenie możliwych rozwiązań. Wtórnym celem jest szkolenie i rozwiązanie problemów stylistycznych (np. z formą kodu, dokumentacji, interfejsów użytkownika).
- **Audyt**. Przeglądy potwierdzające zgodność oprogramowania z wymaganiami, specyfikacjami, zaleceniami, standardami, procedurami, instrukcjami, kontraktami i licencjami. Obiektywność audytu wymaga, aby był on przeprowadzony przez osoby niezależne od zespołu projektowego.

Skład zespołu oceniającego oprogramowanie

Dla poważnych projektów ocena oprogramowania nie może być wykonana w sposób amatorski. Musi być powołany zespół, którego zadaniem będzie zarówno przygotowanie testów jak i ich przeprowadzenie.

Potencjalny skład zespołu oceniającego:

- Kierownik
- Sekretarz
- Członkowie, w tym:
  - użytkownicy
  - kierownik projektu oprogramowania
  - inżynierowie oprogramowania
  - bibliotekarz oprogramowania
  - personel zespołu zapewnienia jakości
  - niezależny personel weryfikacji i atestowania
  - niezależni eksperci nie związani z rozwojem projektu

Zadania kierownika: nominacje na członków zespołu, organizacja przebiegu oceny i spotkań zespołu, rozpowszechnienie dokumentów oceny pomiędzy członków zespołu, organizacja pracy, przewodniczenie posiedzeniom, wydanie końcowego raportu, i być może inne zadania.

Co to jest audyt?

audit

Audytem nazywany jest niezależny przegląd i ocena jakości oprogramowania, która zapewnia zgodność z wymaganiami na oprogramowanie, a także ze specyfikacją, generalnymi założeniami, standardami, procedurami, instrukcjami, kodami oraz kontraktowymi i licencyjnymi wymaganiami.

Aby zapewnić obiektywność, audyt powinien być przeprowadzony przez osoby niezależne od zespołu projektowego.

Audyt powinien być przeprowadzany przez odpowiednią organizację audytu (w Polsce Polskie Stowarzyszenie Audytu), przez osoby posiadające uprawnienia/licencję audytorów. Wymagane jest zdanie egzaminu na audytora.

Reguły i zasady audytu są określone w normie:
ANSI/IEEE Std 1028-1988 *„IEEE Standard for Reviews and Audits”*.

Audyt jest zwykle związany z normami jakości.

Aspekty audytu

- **Przykłady**
  - Analiza stanu projektu
  - Analiza celowości projektu
  - Analiza procesu wytwórczego
  - Analiza dowolnego procesu
  - Analiza systemu jakości
  - Analiza stosowania systemu jakości

- **Relacje odbiorca dostawca**
  - audyt wewnętrzny
  - audyt zewnętrzny
  - audyt „trzeciej strony”

- **Etapy**
  - planowanie i przygotowanie
  - wykonywanie
  - raportowanie
  - zamknięcie

Audyt projektu informatycznego

- Celem audytu projektu informatycznego jest dostarczenie odbiorcy i dostawcy obiektywnych, aktualnych i syntetycznych informacji o stanie całego projektu

- Jest to osiągane przez zbieranie dowodów, że projekt:
  - posiada możliwości (zasoby, kompetencje, metody, standardy) by osiągnąć sukces,
  - optymalnie wykorzystuje te możliwości,
  - rzeczywiście osiąga założone cele (cząstkowe).

- Zebrane informacje służą jako podstawa do podejmowania strategicznych decyzji w projekcie

Przedmioty i perspektywy audytu

- **Przedmioty**
  - **Procesy** projektu informatycznego - ma na celu sprawdzenie czy wykonywane prace oraz sposób ich wykonywania są prawidłowe
  - **Produkty** (cząstkowe) projektu informatycznego - ma na celu sprawdzenie czy rezultaty poszczególnych prac odpowiadają zakładanym wymaganiom
- **Perspektywy**
  - **Technologia** - ma na celu sprawdzenie czy użyte techniki oraz opracowane rozwiązania są prawidłowe i prawidłowo stosowane
  - **Zarządzanie** - ma na celu sprawdzenie czy sposób zarządzania projektem umożliwia jego sukces

Inspekcje

Inspekcja to formalna technika oceny, w której wymagania na oprogramowanie, projekt lub kod są szczegółowo badane przez osobę lub grupę osób nie będących autorami, w celu identyfikacji błędów, naruszenia standardów i innych problemów [IEEE Std. 729-1983]

- Technika o najlepszej skuteczności wykrywania niezgodności (od 50% do 80%; średnia 60%; dla testowania średnia 30%, max 50%)
- Stosowane dla „elitarnych” systemów
- Dlaczego nie są powszechne?
  - trudne w sprzedaży: nie potrzeba narzędzi, potrzeba planowania i kompetentnych ludzi
  - analiza kosztów i zysków nie jest prosta

Cechy inspekcji

- Sesje są zaplanowane i przygotowane
- Błędy i problemy są notowane
- Wykonywana przez techników dla techników (bez udziału kierownictwa)
- Dane nie są wykorzystywane do oceny pracowników (?)
  - Ocena produktu => pośrednia ocena pracownika
- Zasoby na inspekcje są gwarantowane
- Błędy są wykorzystywane w poprawie procesu programowego (prewencja)
- Proces inspekcji jest mierzony
- Proces inspekcji jest poprawiany

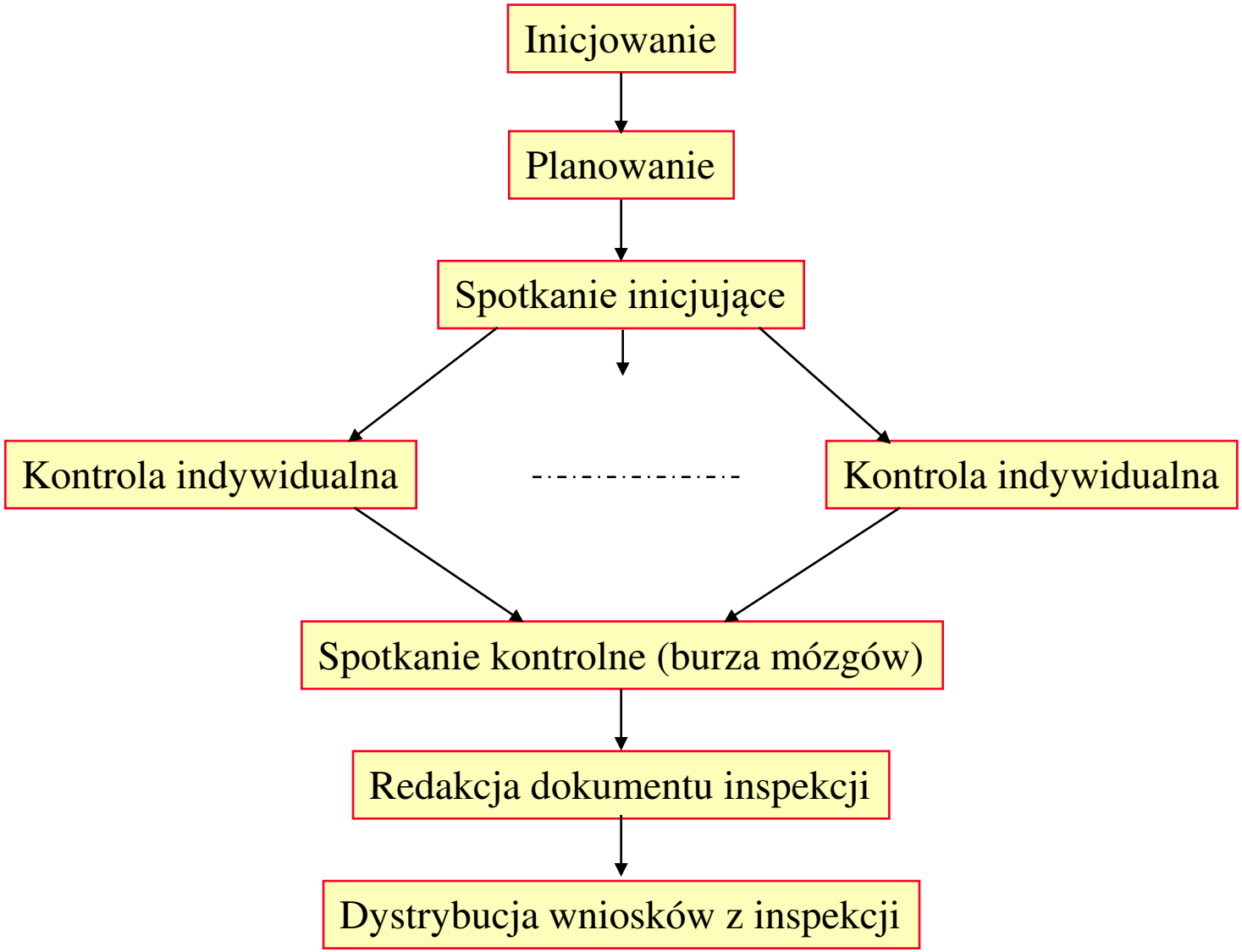
Korzyści z inspekcji

- Wzrost produktywności od 30% do 100%
- Skrócenie czasu projektu od 10% do 30%
- Skrócenie kosztu i czasu wykonywania testów (mniej błędów, mniej testów regresyjnych)
- Znacznie mniejsze koszty pielęgnacji (naprawczej)
- Poprawa procesu programowego
- Dostarczanie na czas (bo wcześnie wiemy o problemach)
- Większy komfort pracy (brak presji czasu i nadgodzin)
- Zwiększenie motywacji
  - świadomość, że produkt będzie oceniany (wybór pomiędzy byciem zażenowanym a dumnym)
  - nauka przez znajdowanie błędów
- Mniejsze koszty marketingu („przykrywanie” braku jakości)

Zagrożenia inspekcji

- Ocena osób na podstawie zebranych metryk
- Złe prowadzenie inspekcji - mała efektywność i skuteczność
- Słabi kontrolerzy
- Kontrola indywidualna niewystarczająca (jakość i ilość)
- Skłonność autora do lekceważenia defektów na etapie opracowywania dokumentów (“inspekcja wskaże błędy...”)
- Dyskusje o rozwiązaniach podczas spotkania kontrolnego
- Poczucie zagrożenia u autora - obrona własnych rozwiązań, często na podstawie drugorzędnych lub wydumanych argumentów (syndrom NIH - *Not Invented Here*)
- Krytyczne nastawienie do autora programu, „czepialstwo”

Przebieg inspekcji



Kroki inspekcji (1)

- Inicjowanie
  zgłoszenie potrzeby inspekcji; wyłonienie lidera inspekcji
- Planowanie
  lider ustala uczestników, listy kontrolne, zbiory reguł, tempo kontroli, daty spotkań kontrolnych
- Spotkanie inicjujące
  ustalenie ról, ustalenie celów i oczekiwań, dystrybucja dokumentu, szkolenie w inspekcjach
- Kontrola indywidualna
  uczestnicy sprawdzają dokument względem zadanych kryteriów, reguł i list kontrolnych (znaleźć jak najwięcej unikalnych błędów)
- Spotkanie kontrolne (burza mózgów)
  Notowanie uwag z kontroli indywidualnej;
  Każda uwaga jest kwalifikowana jako „zagadnienie” (potencjalny błąd), „pytanie o intencję”, „propozycja poprawy procesu”; Szukanie nowych zagadnień; Poprawa procesu inspekcji.

Kroki inspekcji (2)

- Poprawa produktu: edytor (najczęściej autor) rozwiązuje zagadnienia; prawdziwy problem może być inny niż jest to zgłoszone; dokument jest redagowany by uniknąć błędnych interpretacji
- Kontynuacja: lider sprawdza, że obsłużono wszystkie zagadnienia: są poprawione lub są w systemie zarządzania konfiguracją; sprawdza kompletność a nie poprawność
- Decyzja o gotowości: lider podejmuje decyzję czy produkt jest gotowy do przekazania dalej (np. liczba błędów w określonym limicie)
- Rozpowszechnienie dokumentu
- Burza mózgów ma na celu identyfikację przyczyn błędów (max 10 najpoważniejszych) i propozycji poprawy procesu programowego, by błędy te nie powtórzyły się w przyszłości; idee są notowane bez ich głębokiej oceny

Rodzaje testów

Testy można klasyfikować z różnych punktów widzenia:

- **Wykrywanie błędów**, czyli testy, których głównym celem jest wykrycie jak największej liczby błędów w programie
- **Testy statystyczne**, których celem jest wykrycie przyczyn najczęstszych błędnych wykonań oraz ocena niezawodności systemu.

Z punktu widzenia techniki wykonywania testów można je podzielić na:

- **Testy dynamiczne**, które polegają na wykonywaniu (fragmentów) programu i porównywaniu uzyskanych wyników z wynikami poprawnymi,
- **Testy statyczne**, oparte na analizie kodu (wykonywanie kodu w myśli). Dla większych programów (> 20 linii) mało efektywne.
  - *Metody matematyczne weryfikacji programów można uważać za testy statyczne ale w tej roli okazały się nieefektywne*

Błąd i błędne wykonanie

Błąd (*failure, error*) - niepoprawna konstrukcja znajdująca się w programie, która może doprowadzić do niewłaściwego działania.

- Pojęcie „błędu” jest w pewnym stopniu relatywne i subiektywne
- „Błąd” może oznaczać sytuację nie do przewidzenia w trakcie pisania programu, np. stworzenie luki, przez którą wcisnął się haker

Błędne wykonanie (*failure*) - niepoprawne działanie systemu w trakcie jego pracy.

Błąd może prowadzić do różnych błędnych wykonań.
To samo błędne wykonanie może być spowodowane różnymi błędami.

Proces weryfikacji oprogramowania można określić jako poszukiwanie i usuwanie błędów na podstawie obserwacji błędnych wykonań oraz innych testów.

Typowe fazy testowania systemu

- **Testy modułów**: Są one wykonywane już w fazie implementacji bezpośrednio po zakończeniu realizacji poszczególnych modułów

- **Testy systemu**: W tej fazie integrowane są poszczególne moduły i testowane są poszczególne podsystemy oraz system jako całość

- **Testy akceptacji** (*acceptance testing*):
  - Przekazanie do przetestowania przyszłemu użytkownikowi - testy **alfa**
  - W przypadku oprogramowania sprzedawanego rynkowo: nieodpłatne przekazanie oprogramowania grupie użytkowników - testy **beta**
  - Mówi się także o oprogramowaniu w ***wersji beta,*** czyli oprogramowaniu bez gwarancji wytwórcy (i bez obowiązku naprawy błędu)

Co podlega testowaniu (1)?

- **Wydajność systemu** i poszczególnych jego funkcji (czy jest satysfakcjonująca).
- **Interfejsy systemu** na zgodność z wymaganiami określonymi przez użytkowników
- **Własności operacyjne systemu**, np. wymagania logistyczne, organizacyjne, użyteczność/ stopień skomplikowania instrukcji kierowanych do systemu, czytelność ekranów, operacje wymagające zbyt wielu kroków, jakość komunikatów systemu, jakość informacji o błędach, jakość pomocy.
- **Testy zużycia zasobów**: zużycie czasu jednostki centralnej, zużycie pamięci operacyjnej, przestrzeni dyskowej, itd.
- **Zabezpieczenie systemu**: odporność systemu na naruszenia prywatności, tajności, integralności, spójności i dostępności. Testy powinny np. obejmować:
  - zabezpieczenie haseł użytkowników
  - testy zamykania zasobów przed niepowołanym dostępem
  - testy dostępu do plików przez niepowołanych użytkowników
  - testy na możliwość zablokowania systemu przez niepowołane osoby
  - ....

Co podlega testowaniu (2)?

- **Przenaszalność oprogramowania**: czy oprogramowanie będzie działać w zróżnicowanym środowisku (np. różnych wersjach Windows, Unix), przy różnych wersjach instalacyjnych, rozmiarach zasobów, kartach graficznych, rozdzielczości ekranów, oprogramowaniu wspomagającym (bibliotekach), ...

- **Niezawodność oprogramowania**, zwykle mierzoną średnim czasem pomiędzy awariami.

- **Odtwarzalność oprogramowania** (*maintainability*), mierzoną zwykle średnim czasem reperowania oprogramowania po jego awarii. Pomiar powinien uwzględniać średni czas od zgłoszenia awarii do ponownego sprawnego działania.

- **Bezpieczeństwo oprogramowania**: stopień minimalizacji katastrofalnych skutków wynikających z niesprawnego działania. (Przykładem jest wyłączenie prądu podczas działania w banku i obserwacja, co się w takim przypadku stanie.)

- **Kompletność i jakość założonych funkcji systemu**.

- **Nie przekraczanie ograniczeń**, np. na zajmowaną pamięć, obciążenia procesora,...

Co podlega testowaniu (3)?

- **Modyfikowalność oprogramowania**, czyli zdolność jego do zmiany przy zmieniających się założeniach lub wymaganiach

- **Obciążalność oprogramowania**, tj. jego zdolność do poprawnej pracy przy ekstremalnie dużych obciążeniach. Np. maksymalnej liczbie użytkowników, bardzo dużych rozmiarach plików, dużej liczbie danych w bazie danych, ogromnych (maksymalnych) zapisach, bardzo długich liniach danych źródłowych. W tych testach czas nie odgrywa roli, chodzi wyłącznie o to, czy system poradzi sobie z ekstremalnymi rozmiarami danych lub ich komponentów oraz z maksymalnymi obciążeniami na jego wejściu.

- **Skalowalność systemu**, tj. spełnienie warunków (m.in. czasowych) przy znacznym wzroście obciążenia.

- **Akceptowalność systemu**, tj. stopień usatysfakcjonowania użytkowników.

- **Jakość dokumentacji**, pomocy, materiałów szkoleniowych, zmniejszenia bariery dla nowicjuszy.

Testy statystyczne

Podobne do testów z innych dziedzin wytwórczości technicznej.

- Losowa konstrukcja danych wejściowych zgodnie z rozkładem prawdopodobieństwa tych danych
- Określenie wyników poprawnego działania systemu na tych danych
- Uruchomienie systemu oraz porównanie wyników jego działania z poprawnymi wynikami.

Powyższe czynności powtarzane są cyklicznie.

Stosowanie testów statystycznych wymaga określenia rozkładu prawdopodobieństwa danych wejściowych możliwie bliskiemu rozkładowi, który pojawi się w rzeczywistości. Dokładne przewidzenie takiego rozkładu jest trudne, w związku z czym wnioski wyciągnięte na podstawie takich testów mogą być nietrafne.

Założeniem jest przetestowanie systemu w typowych sytuacjach. Istotne jest także przetestowanie systemu w sytuacjach skrajnych, nietypowych, ale dostatecznie ważnych.

Istotną zaletą testów statystycznych jest możliwość ich automatyzacji, a co za tym idzie, możliwości wykonania dużej ich liczby.

Testy statystyczne są mało efektywne, stosuje się je bardzo rzadko.

Testowanie na zasadzie białej skrzynki

white-box testing

Tak określa się sprawdzanie wewnętrznej logiki oprogramowania. (Lepszym terminem byłoby „*testowanie na zasadzie szklanej skrzynki*”.)

Testowanie na zasadzie białej skrzynki pozwala sprawdzić wewnętrzną logikę programów poprzez odpowiedni dobór danych wejściowych, dzięki czemu można prześledzić wszystkie ścieżki przebiegu sterowania programu.

- Tradycyjnie programiści wstawiają kod diagnostyczny do programu aby śledzić wewnętrzne przetwarzanie. Debuggery pozwalają programistom obserwować wykonanie programu krok po kroku.
- Często niezbędne staje się wcześniejsze przygotowanie danych testowych lub specjalnych programów usprawniających testowanie (np. programu wywołującego testowaną procedurę z różnymi parametrami).
- Dane testowe powinny być dobrane w taki sposób, aby każda ścieżka w programie była co najmniej raz przetestowana.
- Ograniczeniem testowania na zasadzie białej skrzynki jest niemożliwość pokazania brakujących funkcji w programie. Wadę tę usuwa testowanie n/z czarnej skrzynki.

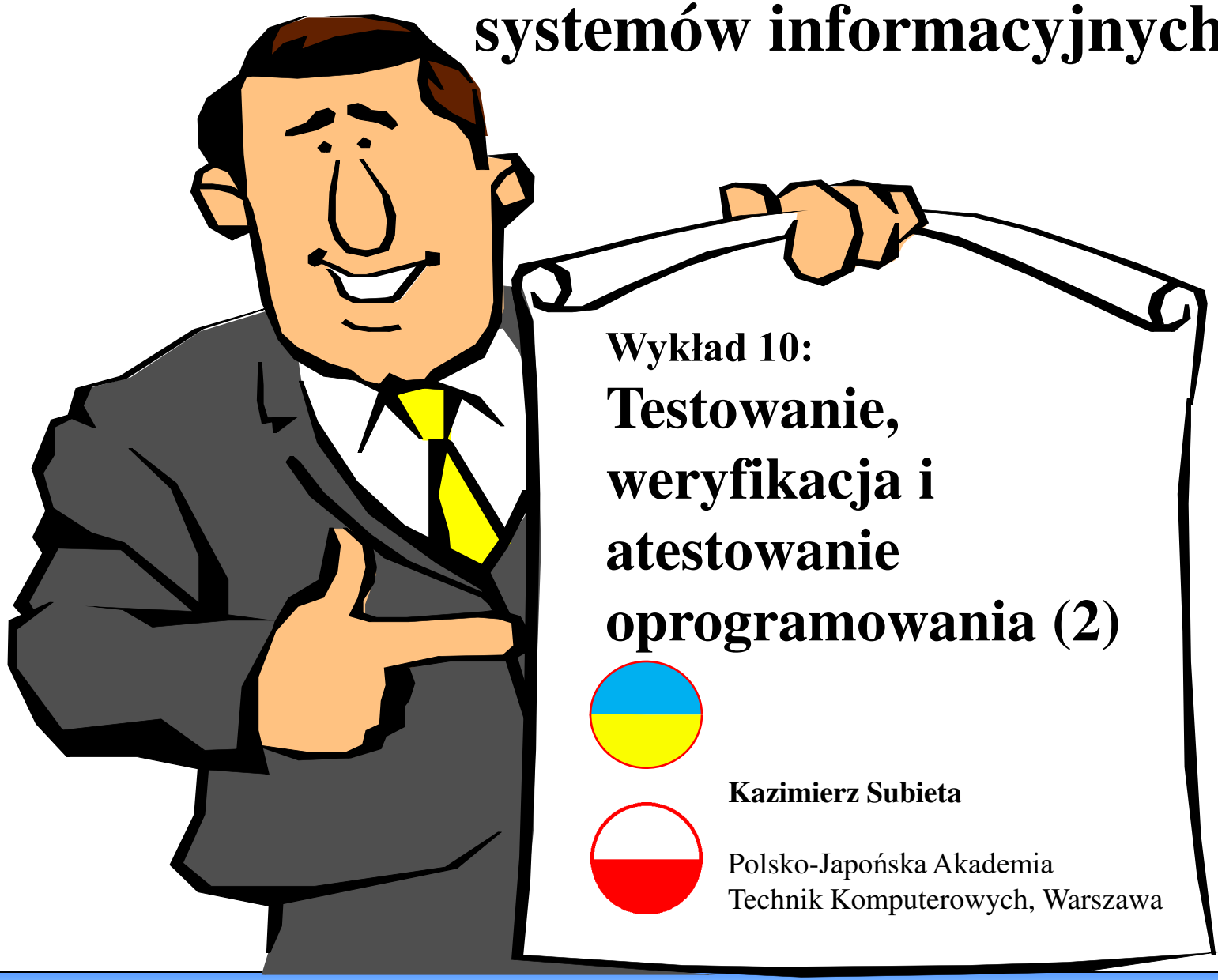
Testowanie na zasadzie czarnej skrzynki

black-box testing

Tak określa się sprawdzanie funkcji oprogramowania bez zaglądania do środka programu. Testujący traktuje sprawdzany moduł jak „czarną skrzynkę", której wnętrze jest niewidoczne.

- Testowanie na zasadzie czarnej skrzynki powinno obejmować cały zakres danych wejściowych.
- Testujący powinni podzielić dane wejściowe w „klasy równoważności", co do których istnieje duże przypuszczenie, że będą produkować te same błędy. Np. jeżeli testujemy wartość „Malinowski", to prawdopodobnie w tej samej klasie równoważności jest wartość „Kowalski". Celem jest zredukowanie „eksplozji danych testowych".
- „Klasy równoważności" mogą być również zależne od wyników zwracanych przez testowane funkcje. Np. jeżeli wejściem jest wiek pracownika i istnieje funkcja zwracająca wartości „młodociany", „normalny" „wiek emerytalny", wówczas implikuje to odpowiednie klasy równoważności dla danych wejściowych.
- Wiele wejść dla danych (wiele parametrów funkcji) może wymagać zastosowania pewnych systematycznych metod określania ich kombinacji, np. tablic decyzyjnych lub grafów przyczyna-skutek.

Budowa i integracja systemów informacyjnych



Wykład 10:
Testowanie, weryfikacja i atestowanie oprogramowania (2)

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Określenie niezawodności oprogramowania

Miary niezawodności:

- **Prawdopodobieństwo błędnego wykonania** podczas realizacji transakcji. Błędne wykonanie powoduje zerwanie całej transakcji. Miarą jest częstość występowania transakcji, które nie powiodły się wskutek błędów.

- **Częstotliwość występowania błędnych wykonań**: ilość błędów w jednostce czasu. Np. 0.1/h oznacza, że w ciągu godziny ilość spodziewanych błędnych wykonań wynosi 0.1. Miara ta jest stosowana w przypadku systemów, które nie mają charakteru transakcyjnego.

- **Średni czas między błędnymi wykonaniami** - odwrotność poprzedniej miary.

- **Dostępność**: prawdopodobieństwo, że w danej chwili system będzie dostępny do użytkowania. Miarę tę można oszacować na podstawie stosunku czasu, w którym system jest dostępny, do czasu od wystąpienia błędu do powrotu do normalnej sytuacji. Miara zależy nie tylko od błędnych wykonań, ale także od narzutu błędów na niedostępność systemu.

Oszacowanie niezawodności

Niekiedy poziom niezawodności (wartość pewnej miary lub miar) jest określany w wymaganiach klienta.

Częściej, jest on jednak wyrażony w terminach jakościowych, co bardzo utrudnia obiektywną weryfikację. Jednakże informacja o niezawodności jest przydatna również wtedy, gdy klient nie określił jej jednoznacznie w wymaganiach.

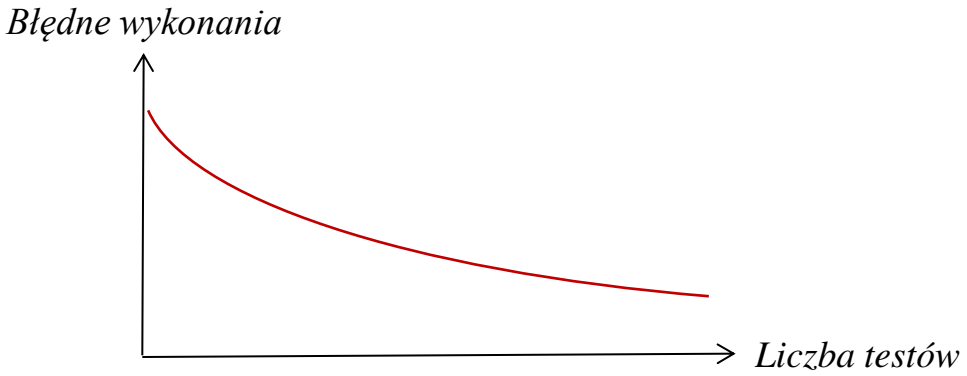
Dlaczego?

- Częstotliwość występowania błędnych wykonań ma duży wpływ na koszt konserwacji oprogramowania (serwis telefoniczny + wizyty u klienta).
- Znajomość niezawodności pozwala oszacować koszt serwisu, liczbę personelu, liczbę zgłoszeń telefonicznych, łączny koszt serwisu.
- Znajomość niezawodności pozwala ocenić i polepszyć procesy wytwarzania pod kątem zminimalizowania łącznego kosztu wynikającego z kosztów wytwarzania, kosztów utrzymania, powodzenia na rynku, reputacji firmy.

Wzrost niezawodności oprogramowania

- Rezultatem wykrycia przyczyn błędów jest ich usunięcie.
- Jeżeli przy tym nie wprowadza się nowych błędów, to można mówić o wzroście niezawodności.
- Jeżeli wykonywane są testy czysto statystyczne to wzrost niezawodności określa się następującym wzorem (logarytmiczny wzrost niezawodności):

Błędne wykonania = Błędne wykonania początkowe × exp(-C × liczba testów)



- Miarą niezawodności jest częstotliwość występowania błędnych wykonań.
- Stała C zależy od konkretnego systemu. Można ją określić na podstawie obserwacji statystycznych.
- Szybszy wzrost niezawodności można osiągnąć jeżeli kolejnych przebiegach testuje się sytuacje, które dotąd nie były testowane.

Wykrywanie błędów - rodzaje testów

Dynamiczne testy zorientowane na wykrywanie błędów dzieli się na:

- **Testy funkcjonalne – „czarnej skrzynki”** (*functional tests, black-box tests*), które zakładają znajomość jedynie wymagań wobec testowanej funkcji.
  - System jest traktowany jako czarna skrzynka, która w nieznany sposób realizuje wykonywane funkcje.
  - Testy powinny wykonywać osoby, które nie były zaangażowane w realizację testowanych fragmentów systemu.

- **Testy strukturalne – „szklanej skrzynki”** (*structural tests, white-box tests, glass-box tests*): zakładają znajomość i analizę sposobu implementacji testowanych funkcji.
  - Śledzenie wykonywania programu przy pomocy debuggera

Testy funkcjonalne

- Pełne przetestowanie rzeczywistego systemu jest niemożliwe z powodu ogromnej liczby kombinacji danych wejściowych i stanów.
  - Nawet dla stosunkowo małych programów.
- Zakłada się, że jeżeli dana funkcja działa poprawnie dla **kilku** danych wejściowych, to działa także poprawnie dla **całej klasy** danych wejściowych.
  - Jest to wnioskowanie czysto heurystyczne.
  - Fakt poprawnego działania w kilku przebiegach nie gwarantuje, że błędne wykonanie nie pojawi się dla innych danych z tej samej klasy.
- Podział danych wejściowych na klasy odbywa się na podstawie opisu wymagań:

  Rachunek o wartości do 1000 zł może być zatwierdzony przez kierownika.
  Rachunek o wartości powyżej 1000 zł musi być zatwierdzony przez prezesa.

- Jednak przetestowanie tylko dwóch wartości, np. 500 i 1500, jest zazwyczaj niewystarczające.
- Konieczne jest także przetestowanie wartości **granicznych**, np. 0, dokładnie 1000 oraz maksymalnej wyobrażalnej wartości.

Kombinacja elementarnych warunków

- Jak widać, testy tylko dla jednej danej wejściowej muszą być przeprowadzone dla pięciu wartości: np. 0, 500, 1000, 1500, max.
- Jeżeli takich danych jest wiele, to mamy do czynienia z **kombinatoryczną eksplozją** przypadków testowych.

Dzieląc dane wejściowe na klasy należy więc brać pod uwagę rozmaite kombinacje elementarnych warunków. Np. do wymienionego warunku dołączony jest następujący:

Kierownik może zatwierdzić miesięcznie rachunek o łącznej wartości do 10 000 zł. Każdy rachunek przekraczający tę wartość musi być zatwierdzony przez prezesa.

Wśród danych wyjściowych można teraz wyróżnić następujące klasy:

- rachunek do 1000 zł nie przekraczający łącznego limitu 10 000 zł.
- rachunek do 1000 zł przekraczający łączny limit 10 000 zł.
- rachunek powyżej 1000 zł nie przekraczający łącznego limitu 10 000 zł.
- rachunek powyżej 1000 zł przekraczający łączny limit 10 000 zł.

Uwzględnienie przypadków granicznych powoduje dalsze rozmnożenie przypadków testowych: (0, 500, 1000, 1500, max) × (<10000, 10000, >10000)

Eksplozja kombinacji danych testowych

Przetestowanie wszystkich kombinacji danych wejściowych (nawet zredukowanych do “typowych” i “granicznych”) jest niemożliwe.

Np. przyjmując, że jest 10 danych wejściowych, każda przyjmuje 5 wartości i przetestowanie jednej kombinacji potrzebuje 10 sec otrzymamy: 5^{10} * 10 sec = ~100 milionów sec = ~75 lat

Konieczny jest wybór kombinacji. **Ogólne zalecenia takiego wyboru**

- **Możliwość wykonania funkcji jest ważniejsza niż jakość jej wykonania**. Brak możliwości wykonania funkcji jest poważniejszym błędem niż np. niezbyt poprawne wyświetlenie jej rezultatów na ekranie.

- **Funkcje systemu znajdujące się w poprzedniej wersji są istotniejsze niż nowo wprowadzone**. Użytkownicy, którzy posługiwali się dana funkcją w poprzedniej wersji systemu będą bardzo niezadowoleni jeżeli w nowej wersji ta funkcja przestanie działać.

- **Typowe sytuacje są ważniejsze niż wyjątki lub sytuacje skrajne**. Błąd w funkcji wykonywanej często lub dla danych typowych jest znacznie bardziej istotny niż błąd w funkcji wykonywanej rzadko dla nietypowych danych.

Testy strukturalne

W przypadku testów strukturalnych, dane wejściowe dobiera się na podstawie analizy struktury programu realizującego testowane funkcje.
Kryteria doboru danych testowych są następujące:

- **Kryterium pokrycia wszystkich instrukcji**. Zgodnie z tym kryterium dane wejściowe należy dobierać tak, aby każda instrukcja została wykonana co najmniej raz. Spełnienie tego kryterium zwykle wymaga niewielkiej liczby testów. To kryterium może być jednak nieskuteczne.

```
if x > 0 then begin ... end; y := ln( x);
```

Dla x >0 wykonane będą wszystkie instrukcje, ale dla x <= 0 program jest błędny.

- **Kryterium pokrycia instrukcji warunkowych**. Dane wejściowe należy dobierać tak, aby każdy elementarny warunek instrukcji warunkowej został co najmniej raz spełniony i co najmniej raz nie spełniony. Testy należy wykonać także dla każdej wartości granicznej takiego warunku. Zastosowanie tego warunku pozwoli wykryć błąd z poprzedniego przykładu, gdyż zmusi do testu dla x = 0 oraz x < 0.

Istnieje szereg innych kryteriów prowadzących do bardziej wymagających testów.

Testowanie programów zawierających pętle

Kryteria doboru danych wejściowych mogą opierać się o następujące zalecenia:

- Należy dobrać dane wejściowe tak, aby nie została wykonana żadna iteracja pętli, lub, jeżeli to niemożliwe, została wykonana minimalna liczba iteracji.
- Należy dobrać dane wejściowe tak, aby została wykonana maksymalna liczba iteracji.
- Należy dobrać dane wejściowe tak, aby została wykonana przeciętna liczba iteracji.

Programy uruchamiające

debuggers

Mogą być przydatne dla wewnętrznego testowania jak i dla testowania przez osoby zewnętrzne. Zakładają testowanie na zasadzie białej skrzynki (znajomość kodu).

Własności programów uruchamiających:

- Wyświetlenie stanu zmiennych programu i interakcja z testującym z użyciem symboli kodu źródłowego.
- Wykonywanie programów krok po kroku, z różną granulowością instrukcji
- Ustanowienie punktów kontrolnych w programie (zatrzymujących wykonanie)
- Ustanowienie obserwatorów wartości zmiennych
- Zarządzanie plikiem źródłowym podczas testowania i ewentualna poprawa wykrytych błędów w tym pliku.
- Tworzenie dziennika testowania, umożliwiającego powtórzenie testowego przebiegu.

Debuggery odwołują się do kodu maszynowego, przez co ich znaczenie dla języków wyższego rzędu (np. Java) jest znacznie ograniczone.
Dla silnie optymalizowanych języków (np. SQL) stworzenie debuggera jest praktycznie niemożliwe.

Analizatory przykrycia kodu

coverage analysers

Są to programy umożliwiające ustalenie obszarów kodu źródłowego, które były wykonane w danym przebiegu testowania. Umożliwiają wykrycie martwego kodu, kodu uruchamianego przy bardzo specyficznych danych wejściowych oraz (niekiedy) kodu wykonywanego bardzo często (co może być przyczyną wąskiego gardła w programie).

Bardziej zaawansowane analizatory przykrycia kodu umożliwiają:

- Zsumowanie danych z kilku przebiegów (dla różnych kombinacji danych wejściowych) np. dla łatwiejszego wykrycia martwego kodu.
- Wyświetlenie grafów sterowania, dzięki czemu można łatwiej monitorować przebieg programu
- Wyprowadzenie informacji o przykryciu, umożliwiające poddanie przykrytego kodu dalszym testom.
- Operowanie w środowisku rozwoju oprogramowania.

Programy porównujące

comparators

Są to narzędzia programistyczne umożliwiające porównanie dwóch programów, plików lub zbiorów danych celem wykrycia cech wspólnych i różnic. Często są niezbędne do porównania wyników testów z wynikami oczekiwanymi. Programy porównujące przekazują w czytelnej postaci różnice pomiędzy aktualnymi i oczekiwanymi danymi wyjściowymi.

Ekranowe programy porównujące mogą być bardzo użyteczne dla testowania oprogramowania interakcyjnego. Są niezastąpionym środkiem dla testowania programów z graficznym interfejsem użytkownika.

Pozostałe narzędzia do testowania:

Duża różnorodność narzędzi stosowanych w różnych fazach rozwoju oprogramowania. Np. wspomaganie do planowania testów, automatyczne zarządzania danymi wyjściowymi, automatyczna generacja raportów z testów, generowanie statystyk jakości i niezawodności, wspomaganie powtarzalności testów, itd.

Testy statyczne

Polegają na analizie kodu bez uruchomienia programu. Techniki są następujące:

- dowody poprawności
- metody nieformalne

Dowody poprawności nie są możliwe dla rzeczywistych programów

- Są to urojenia pseudo-teoretyków informatyki (ignorujące skalę problemu)
- Specyfikacja formalna wymagałaby długiego szkolenia w zakresie jej użycia
- Jest obszerna (zwykle większa niż sam program)
- Może też zawierać błędy
- Programiści nie są przygotowani do przeprowadzania dowodów formalnych
- W programach mogą być sytuacje lub problemy formalnie nierozstrzygalne

Statyczne metody nieformalne polegają na analizie kodu przez programistów.

Dwa niewykluczające się podejścia:

- śledzenie przebiegu programu (wykonywanie programu "w myśli" przez analizujące osoby)
- wyszukiwanie typowych błędów i niebezpiecznych przypadków.

Typowe błędy wykrywane statycznie

- Niezainicjowane zmienne
- Porównania na równość liczb zmiennoprzecinkowych
- Indeksy wykraczające poza tablice
- Błędne operacje na wskaźnikach
- Błędy w warunkach instrukcji warunkowych
- Niekończące się pętle
- Błędy popełnione dla wartości granicznych (np. > zamiast >=)
- Błędne użycie lub pominięcie nawiasów w złożonych wyrażeniach
- Nieuwzględnienie błędnych danych

Strategia testów nieformalnych:

- Programista, który dokonał implementacji danego modułu w nieformalny sposób analizuje jego kod.

- Kod uznany przez programistę za poprawny jest analizowany przez doświadczonego programistę. Jeżeli znajdzie on pewną liczbę błędów, moduł jest zwracany programiście do poprawy.

- Szczególnie istotne moduły są analizowane przez grupę osób.

Ocena liczby błędów

Błędy w oprogramowaniu niekoniecznie są bezpośrednio powiązane z jego zawodnością. Oszacowanie liczby błędów ma jednak znaczenie dla producenta oprogramowania, gdyż ma wpływ na koszty konserwacji oprogramowania. Szczególnie istotne dla firm sprzedających oprogramowanie pojedynczym lub nielicznym użytkownikom (relatywnie duży koszt usunięcia błędu).

Podstawy szacowania kosztu konserwacji związanego z usuwaniem błędów:

- Szacunkowa liczba błędów w programie
- Średni procent błędów zgłaszanych przez użytkownika systemu, na podstawie danych z poprzednich przedsięwzięć.
- Średni koszt usunięcia błędu na podstawie danych z poprzednich przedsięwzięć.

Technika “posiewania błędów”.

Polega na tym, że do programu celowo wprowadza się pewną liczbę błędów podobnych do tych, które występują w programie. Wykryciem tych błędów zajmuje się inna grupa programistów niż ta, która dokonała “posiania” błędów.

Załóżmy, że:

N - liczba posianych błędów
M - liczba wszystkich wykrytych błędów
X - liczba posianych błędów, które zostały wykryte

Szacunkowa liczba błędów przed wykonaniem testów: $(M - X) * N/X$

Szacunkowa liczba błędów po usunięciu wykrytych (“posiane” błędy zostały też usunięte): $(M - X) * (N/X - 1)$

Szacunki te mogą być mocno chybione, jeżeli “posiane” błędy nie będą podobne do rzeczywistych błędów występujących w programie.
Technika ta pozwala również na przetestowanie skuteczności metod testowania. Zbyt mała wartość X/N oznacza konieczność poprawy tych metod.

Testy systemu

Techniki:

- testowanie wstępujące
- testowanie zstępujące

- **Testowanie wstępujące**: najpierw testowane są pojedyncze moduły, następnie moduły coraz wyższego poziomu, aż do osiągnięcia poziomu całego systemu. Zastosowanie tej metody nie zawsze jest możliwe, gdyż często moduły są od siebie zależne. Niekiedy moduły współpracujące można zastąpić implementacjami szkieletowymi.

- **Testowanie zstępujące**: rozpoczyna się od testowania modułów wyższego poziomu. Moduły niższego poziomu zastępuje się implementacjami szkieletowymi. Po przetestowaniu modułów wyższego poziomu dołączane są moduły niższego poziomu. Proces ten jest kontynuowany aż do zintegrowania i przetestowania całego systemu.

Testy pod obciążeniem, testy odporności

Testy obciążeniowe (*stress testing*). Celem tych testów jest zbadanie wydajności i niezawodności systemu podczas pracy pod pełnym lub nawet nadmiernym obciążeniem. Dotyczy to szczególnie systemów wielodostępnych i sieciowych. Systemy takie muszą spełniać wymagania dotyczące wydajności, liczby użytkowników, liczby transakcji na godzinę. Testy polegają na wymuszeniu obciążenia równego lub większego od maksymalnego.

Testy odporności (*robustness testing*). Celem tych testów jest sprawdzenie działania w przypadku zajścia niepożądanych zdarzeń, np.

- zaniku zasilania
- awarii sprzętowej
- wprowadzenia niepoprawnych danych
- wydania sekwencji niepoprawnych poleceń

Bezpieczeństwo oprogramowania

Pewnej systemy są krytyczne z punktu widzenia bezpieczeństwa ludzi, np. aparatura medyczna, oprogramowanie wspomagające sterowanie samolotem. Może to być także zagrożenie pośrednie, np. systemy eksperckie w dziedzinie medycyny, systemy informacji o lekach.

Bezpieczeństwo niekoniecznie jest pojęciem tożsamym z niezawodnością.

- System zawodny może być bezpieczny, jeżeli skutki błędnych wykonań nie są groźne.
- Wymagania wobec systemu mogą być niepełne i nie opisywać zachowania systemu we wszystkich sytuacjach. Dotyczy to zwłaszcza sytuacji wyjątkowych, np. wprowadzenia niepoprawnych danych. Ważne jest, aby system zachował się bezpiecznie także wtedy, gdy właściwy sposób reakcji nie został opisany.
- Niebezpieczeństwo może także wynikać z awarii sprzętowych. Analiza bezpieczeństwa musi uwzględniać oba czynniki.

Analiza bezpieczeństwa

Zaczyna się od określenia potencjalnych niebezpieczeństw związanych z użytkowaniem systemu: możliwości utraty życia, zdrowia, strat materialnych, złamania przepisów prawnych.

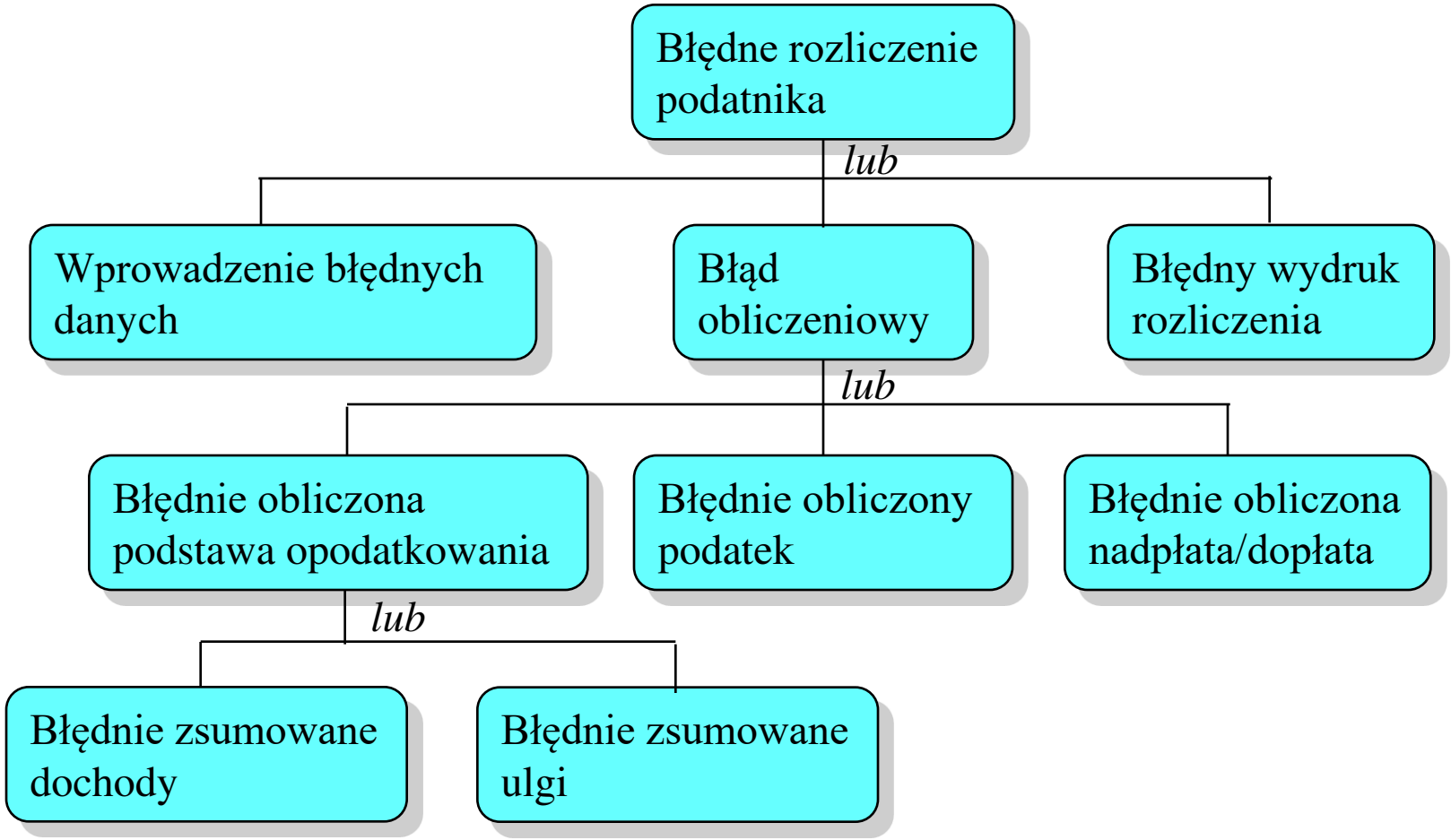
Np. dla programu podatkowego mogą wystąpić następujące niebezpieczeństwa:

- błędne rozliczenie się z urzędem podatkowym
- nie złożenie zeznania podatkowego
- złożenie wielu zeznań dla jednego podatnika

Drzewo błędów

fault tree

Korzeniem drzewa są jest jedna z rozważanych niebezpiecznych sytuacji. Wierzchołkami są sytuacje pośrednie, które mogą prowadzić do sytuacji odpowiadającej wierzchołkowi wyższego poziomu.



Techniki zmniejszania niebezpieczeństwa

- Położenie większego nacisku na unikanie błędów podczas implementacji fragmentów systemu, w których błędy mogą prowadzić do niebezpieczeństw.
- Zlecenie realizacji odpowiedzialnych fragmentów systemu bardziej doświadczonym programistom.
- Zastosowanie techniki programowania N-wersyjnego w przypadku odpowiedzialnych fragmentów systemu.
- Szczególnie dokładne przetestowanie odpowiedzialnych fragmentów systemu.
- Wczesne wykrywanie sytuacji, które mogą być przyczyną niebezpieczeństwa i podjęcie odpowiednich, bezpiecznych akcji. Np. ostrzeżenie w pewnej fazie użytkownika o możliwości zajścia błędu (asercje + zrozumiałe komunikaty o niezgodności).

Czynniki sukcesu, rezultaty testowania

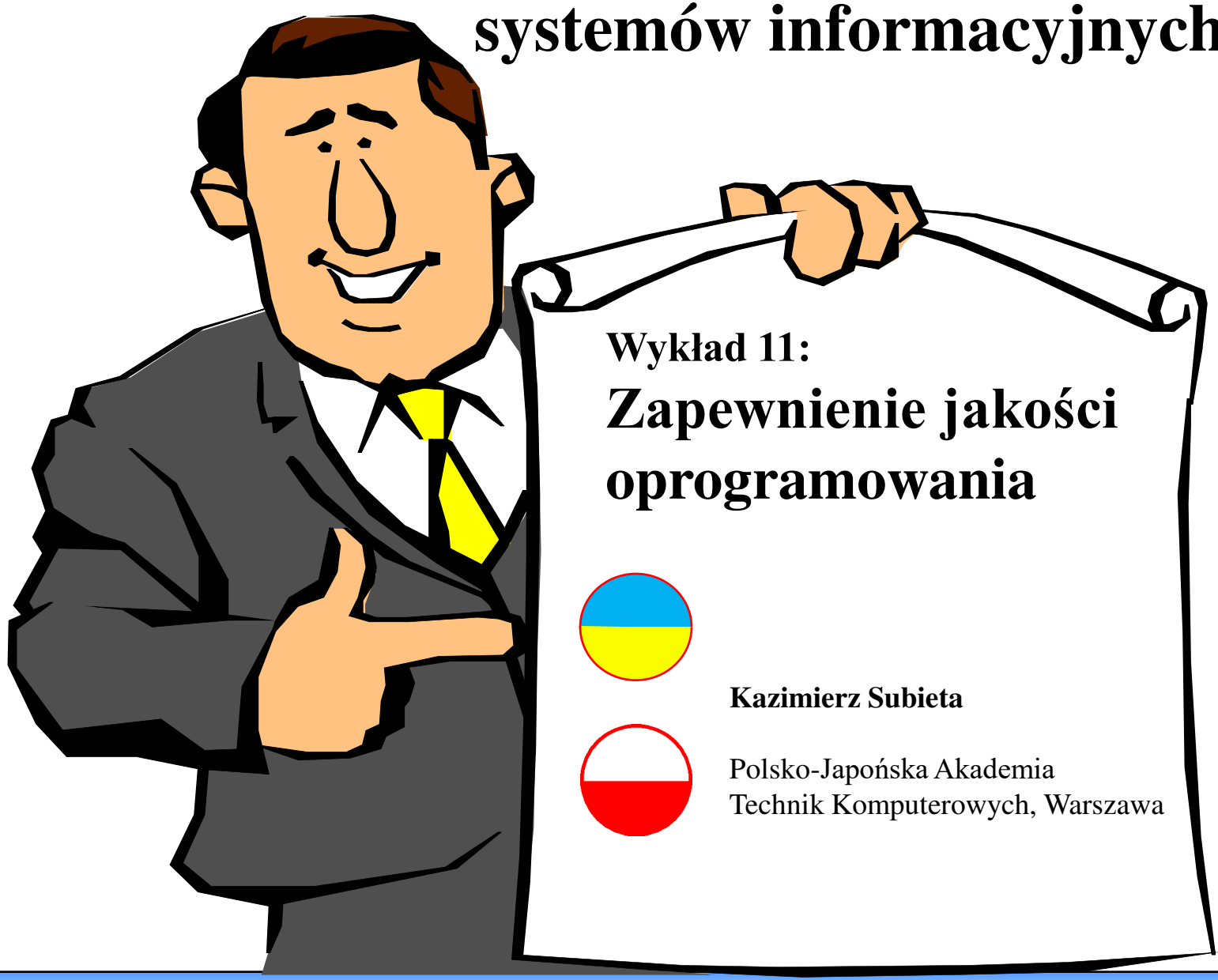
Czynniki sukcesu:

- Określenie fragmentów systemu o szczególnych wymaganiach wobec niezawodności.
- Właściwa motywacja osób zaangażowanych w testowanie. Np. stosowanie nagród dla osób testujących za wykrycie szczególnie groźnych błędów, zaangażowanie osób posiadających szczególny talent do wykrywania błędów

Podstawowe rezultaty testowania:

- Poprawiony kod, projekt, model i specyfikacja wymagań
- Raport przebiegu testów, zawierający informację o przeprowadzonych testach i ich rezultatach.
- Oszacowanie niezawodności oprogramowania i kosztów konserwacji.

Budowa i integracja systemów informacyjnych



Wykład 11:
Zapewnienie jakości oprogramowania

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Co to jest “jakość oprogramowania”?

- Zapewnienie jakości jest rozumiane jako zespół działań zmierzających do wytworzenia u wszystkich zainteresowanych **przekonania**, że dostarczony produkt właściwie realizuje swoje funkcje i odpowiada aktualnym wymaganiom i standardom.
- Jakość, oprócz mierzalnych czynników technicznych, włącza dużą liczbę niemierzalnych obiektywnie czynników psychologicznych.
- Podstawą obiektywnych wniosków co do jakości oprogramowania są **pomiary** pewnych parametrów użytkowych (niezawodności, szybkości, itd.) w realnym środowisku, np. przy użyciu metod statystycznych.
- Obiektywne pomiary cech produktów programistycznych są utrudnione lub niemożliwe (tak jak pomiary jakości fundamentów w budynku).
- Jakość gotowych produktów programistycznych jest nieokreślona ze względu na ich złożoność (eksplozja danych testowych), wieloaspektowość, identyczność wszystkich kopii produktu, oraz niską przewidywalności wszystkich aspektów ich zastosowań w długim czasie.

Trudności z oceną jakości oprogramowania

- Oceny jakości najczęściej muszą być znane zanim powstanie gotowy, działający produkt, co wyklucza zastosowanie obiektywnych metod pomiarowych.
- Wiele czynników składających się na jakość produktu jest niemierzalna.
- Produkty programistyczne są złożone i wieloaspektowe, co powoduje trudności w wyodrębnieniu cech mierzalnych, które odzwierciedlałyby istotne aspekty jakości.
- Produkty programistyczne mogą działać w różnych zastosowaniach, o różnej skali. Pomiary jakości mogą okazać się nieadekwatne przy zmianie skali (np. zwiększonej liczbie danych lub użytkowników), w innym środowisku, itp.
- Pomiary mogą okazać się bardzo kosztowne, czasochłonne lub niewykonalne (z powodu niemożliwości stworzenia środowiska pomiarowego przed wdrożeniem);
- Nie ma zgody co do tego, w jaki sposób pomierzone cechy danego produktu składają się na ogólny wskaźnik jego jakości.

Oceny jakości produktów programistycznych są skazane na metody spekulacyjne, oparte na uproszczeniach oraz dodatkowych założeniach.

Metody zapewnienia jakości w projektach

Opierają się na dyscyplinie całości procesu, raczej niż na pomiarach efektów procesów.

- Pomiary są istotne, o ile dadzą się przeprowadzić.
- Obowiązuje założenie, że brak dyscypliny, bałagan, brak kontroli podczas dowolnego procesu jest czynnikiem uniemożliwiającym jakościowe wykonanie produktu lub usługi.
- Z tego oczywiście **nie wynika**, że dyscyplina, kontrola i porządek zapewniają jakość.
- Ale w zarządzaniu jakością w projektach nie istnieją inne, lepsze metody.
- W zarządzaniu projektami metody zapewnienia jakości mało zależą od dziedziny wytwarzania lub usługi, zatem normy jakości również w dużym stopniu są niezależnie od dziedziny (np. ISO 9000, Prince -2, …)

TQM - zarządzanie przez jakość

- W latach 50-tych produkty japońskie uważano za buble (tak jak obecnie niektórzy postrzegają wyroby chińskie)
- Koncepcja TQM została zaproponowania przez Japończyka **Eiji Toyodę** dla potrzeb naprawy prestiżu japońskiego przemysłu motoryzacyjnego - 1950 r.
- Główna myśl TQM mówiła o tym, że w związku z tym, że to klient stanowi o rentowności przedsiębiorstwa, to należy tak sterować wszystkimi fazami procesu produkcyjnego wyrobu, aby klient był zadowolony z jakości tego wyrobu,
- TQM została rozwinięta przez amerykanów (W.E.Deming, P.Crosby, J.M.Juran, A.V.Feigenbaum), japończyków (E.Toyoda, M.Imai, K.Ishikawa) i brytyjczyka (J.Oaklanda),
- Każdy z powyższych autorów zdefiniował własne zasady TQM. Wszystkie one obracają się jednak wokół zasady Toyody: „Jakość jest najważniejszym kryterium oceny przydatności produktów dla klienta, a to właśnie klient umożliwia funkcjonowanie wytwórcy tych produktów”.
- Producent wytwarzający produkty kiepskie powinien wypaść z rynku.

Jakość w terminologii ISO 9000

- **jakość** - ogół cech i właściwości wyrobu lub usługi decydujący o zdolności wyrobu lub usługi do zaspokojenia stwierdzonych lub przewidywanych potrzeb użytkownika produktu
- **system jakości** - odpowiednio zbudowana struktura organizacyjna z jednoznacznym podziałem odpowiedzialności, określeniem procedur, procesów i zasobów, umożliwiających wdrożenie tzw. *zarządzania jakością*
- **zarządzanie jakością** - jest związane z aspektem całości funkcji zarządzania organizacji, który jest decydujący w określaniu i wdrażaniu *polityki jakości*
- **polityka jakości** - ogół zamierzeń i kierunków działań organizacji dotyczących jakości, w sposób formalny wyrażony przez najwyższe kierownictwo organizacji, będącej systemem jakości
- **audyt jakości** - systematyczne i niezależne badanie, mające określić, czy działania dotyczące jakości i ich wyniki odpowiadają zaplanowanym ustaleniom, czy te ustalenia są skutecznie realizowane i czy pozwalają na osiągnięcie odpowiedniego *poziomu jakości*

Polityka i system jakości

Polityka jakości to ogólne intencje i zamierzenia danej organizacji w odniesieniu do jakości [ISO8402] wyrażana w sposób formalny przez zarząd firmy.

- Musi być zdefiniowana i udokumentowana;
- Muszą być określone cele i zaangażowanie w jakość;
- Musi być zgodna z działaniami przedsiębiorstwa i oczekiwaniami klienta;
- Musi być zakomunikowana i rozumiana na wszystkich szczeblach zarządzania.

System jakości to struktura organizacyjna, przydział odpowiedzialności, procedury postępowania, zasoby użyte do implementacji polityki jakości w danej organizacji [ISO8402]

- **pełnomocnik** lub zespół do spraw jakości;
- **księga jakości**: udokumentowane procedury systemu jakości.

Model jakości ISO 9126

- **Funkcjonalność**
  - odpowiedniość
  - dokładność
  - współdziałanie
  - zgodność
  - bezpieczeństwo
- **Niezawodność**
  - dojrzałość
  - tolerancja błędów
  - odtwarzalność
- **Użyteczność**
  - zrozumiałość
  - łatwość uczenia
  - łatwość posługiwania się
- **Efektywność**
  - charakterystyka czasowa
  - wykorzystanie zasobów
- **Pielęgnowalność**
  - dostępność
  - podatność na zmiany
  - stabilność
  - łatwość walidacji
- **Przenośność**
  - dostosowywalność
  - instalacyjność
  - zgodność
  - zamienność

Atrybut jakościowy to cecha lub charakterystyka mająca wpływ na jakość danego obiektu

Zasady zarządzania jakością

- Ukierunkowanie na klienta (również klient wewnętrzny)
- Przywództwo (budowa wizji, identyfikacja wartości)
- Zaangażowanie ludzi (satysfakcja, motywacja, szkolenia)
- Podejście procesowe (koncentracja na poszczególnych krokach procesu i relacjach pomiędzy tymi krokami, pomiary)
- Podejście systemowe (całe otoczenie procesu wytwórczego)
- Ciągłe doskonalenie (doskonalenie stanu obecnego, ewolucja a nie rewolucja)
- Rzetelna informacja (zbieranie i zabezpieczanie danych do podejmowania obiektywnych decyzji)
- Partnerstwo dla jakości (bliskie związki producentów z klientami)

Zapewnienie Jakości Oprogramowania (ZJO)

software quality assurance, SQA

Zgodnie z normą jest to „**planowany i systematyczny wzorzec wszystkich działań potrzebnych dla dostarczenia adekwatnego potwierdzenia że element lub produkt jest zgodny z ustanowionymi wymaganiami technicznymi**”.

ZJO oznacza sprawdzanie:

- czy plany są zdefiniowane zgodnie ze standardami;
- czy procedury są wykonywane zgodnie z planami;
- czy produkty są implementowane zgodnie z planami.

Kompletne sprawdzenie jest zwykle niemożliwe. Projekty bardziej odpowiedzialne powinny być dokładniej sprawdzane odnośnie jakości.

W ramach ZJO musi być ustalony plan ustalający czynności sprawdzające przeprowadzane w poszczególnych fazach projektu.

Ryzyko utraty jakości

Najbardziej istotnym kryterium przy zapewnianiu jakości jest **ryzyko**.
Najczęstszymi czynnikami ryzyka utraty jakości są:

- nowość projektu,
- złożoność projektu,
- niedostateczne wyszkolenie personelu,
- zbyt małe doświadczenie personelu,
- niesformalizowane (tworzone i zarządzane ad hoc) procedury
- niska dojrzałość organizacyjna wytwórcy.

Dla zmniejszenia ryzyka personel ZJO powinien być zaangażowany w projekt programistyczny jak najwcześniej.

Powinien on sprawdzać wymagania użytkownika, plany, procedury i dokumenty na zgodność ze standardami i przyjętymi procedurami postępowania.

Wynika to z faktu, że dodatkowe koszty związane z problemem lub błędem są tym większe, im później zostanie on zidentyfikowany.

Zadania zapewniania jakości

Firma

- ciągła pielęgnacja procesu wytwarzania
- definiowanie standardów
- nadzór i zatwierdzanie procesu wytwarzania

Projekt

- dostosowywanie standardów
- przeglądy projektu
- testowanie i udział w inspekcjach
- ocena planów wytwarzania i jakościowych
- audyt systemu zarządzania konfiguracją
- udział w komitecie sterującym projektu

Procesy obsługiwane przez personel ZJO

Tworzenie technologii

- tworzenie standardu
- wdrażanie standardu

Kontrola jakości

- ocena produktu
- ocena procesu
- zatwierdzanie jakości

Analiza działalności firmy

- zbieranie danych
- analiza danych

Zarządzanie personelem

Personel ZJO powinien ustalić, czy... (1)

- Projekt jest właściwie zorganizowany, z odpowiednim cyklem życiowym;
- Członkowie zespołu projektowego mają zdefiniowane zadania i odpowiedzialności;
- Plany w zakresie dokumentacji są implementowane;
- Dokumentacja zawiera to, co powinna zawierać;
- Przestrzegane są standardy dokumentacji i kodowania;
- Standardy, praktyki i konwencje są przestrzegane;
- Dane pomiarowe są gromadzone i używane do poprawy produktów i procesów;
- Przeglądy i audyty są przeprowadzane i są właściwie kierowane;
- Testy są specyfikowane i rygorystycznie przeprowadzane;

Personel ZJO powinien ustalić, czy... (2)

- Problemy są rejestrowane i reakcja na problemy jest właściwa
- Projekty używają właściwych narzędzi, technik i metod
- Oprogramowanie jest przechowywane w kontrolowanych bibliotekach
- Oprogramowanie jest przechowywane w chroniony i bezpieczny sposób
- Oprogramowanie od zewnętrznych dostawców spełnia odpowiednie standardy
- Rejestrowane są wszelkie aktywności związane z oprogramowaniem
- Personel jest odpowiednio przeszkolony
- Zagrożenia projektu są zminimalizowane

Zakres działań dla zapewnienia jakości

- Modele i miary służące ocenie kosztu i nakładu pracy
- Modele i miary wydajności ludzi
- Gromadzenie danych
- Modele i miary jakości
- Modele niezawodności
- Ocena i modelowanie wydajności oprogramowania
- Miary struktury i złożoności
- Ocena dojrzałości technologicznej
- Zarządzanie z wykorzystaniem metryk
- Ocena metod i narzędzi

Klasyfikacja zadań zapewnienia jakości

- Certyfikacja systemów przed skierowaniem do produkcji
- Wymuszanie standardów gromadzenia i przetwarzania danych
- Recenzowanie i certyfikacja wytwarzania i dokumentacji
- Opracowanie standardów dotyczących architektury systemu i praktyk programowania
- Recenzowanie projektu systemu pod względem kompletności
- Testowanie nowego lub zmodyfikowanego oprogramowania
- Opracowanie standardów zarządzania
- Szkolenie

Pomiary odgrywają istotną rolę, jednakże są one postrzegane jako jedno z wielu specjalistycznych działań, a nie podstawa całego procesu zapewnienia jakości.

Normy ISO dotyczące jakości

Oprogramowanie jest rozumiane jako jeden z rodzajów wyrobów

ISO 8402

Terminologia

ISO 9000

Wytyczne wyboru modelu

ISO 9001

ISO 9002

ISO 9003

Modele systemu jakości

ISO 9004

Elementy systemu jakości

IEC/TC 56

Niezawodność oprogramowania systemów krytycznych

ISO/IEC 1508

Bezpieczeństwo oprogramowania systemów krytycznych

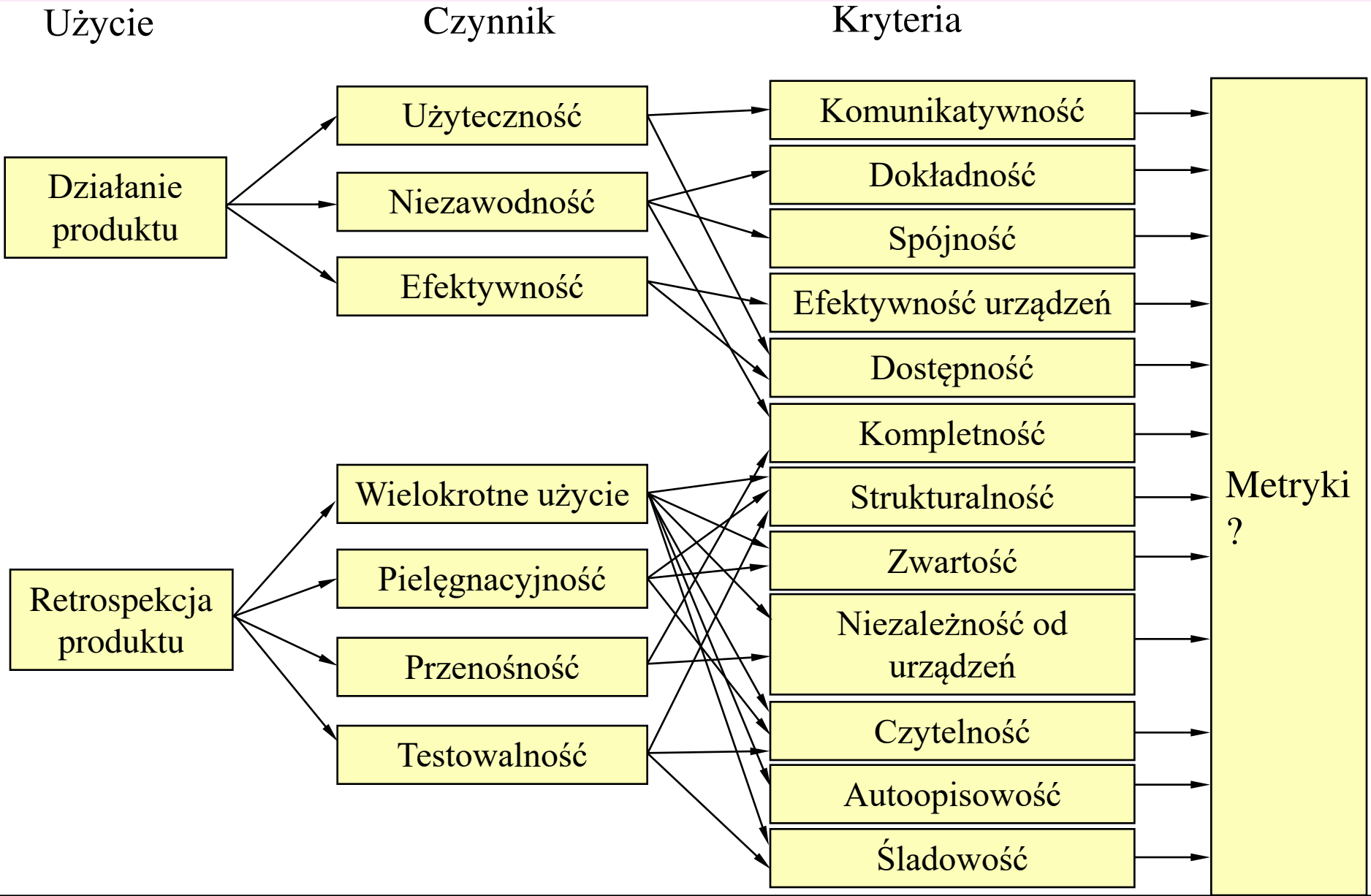
Norma IEEE-730

Norma IEEE-730 podaje ogólne ramy planu zapewniania jakości. Powinien on obejmować następujące zagadnienia:

- analiza punktów widzenia
- referencje wykonawcy
- zarządzanie przedsięwzięciem informatycznym
- dokumentacja
- standaryzacja działań
- przeglądy i audyty
- zarządzanie konfiguracją oprogramowania
- raport napotykanych trudności i podjętych działań prewencyjnych
- wykorzystywane metody i narzędzia
- kontrola kodu, mediów, dostawców
- zarządzanie hurtowniami danych
- pielęgnacja

Norma IEEE-730 została uzupełniona i uszczegółowiona normą IEEE-983.

Model jakości oprogramowania



Niedojrzałość i dojrzałość procesów wytwórczych

Niedojrzałość

- Improwizacja podczas procesu wytwórczego
- Proces jest wyspecyfikowany, ale specyfikacja nie jest stosowana
- Doraźne reagowanie w sytuacji kryzysów
- Harmonogram i budżet są przekraczane
- Funkcjonalność jest stopniowo okrajana
- Jakość produktu jest niska
- Brak obiektywnych kryteriów oceny

Dojrzałość

- Zdolność do budowy oprogramowania jest cechą organizacji a nie personelu
- Proces jest zdefiniowany, znany i wykorzystywany
- Proces jest obserwowany i ulepszany
- Prace są planowane i monitorowane
- Role i odpowiedzialności są zdefiniowane
- Obiektywna, ilościowa ocena

CMM - model dojrzałości procesu wytwórczego

- Wykorzystywany w procedurach klasyfikacji potencjalnych wykonawców oprogramowania dla Departamentu Obrony USA
- Wyróżniono 5 poziomów dojrzałości wytwórców (poczynając od poziomu najniższego):
  - 1 poziom początkowy (proces chaotyczny)
  - 2 poziom powtarzalny (proces zindywidualizowany)
  - 3 poziom zdefiniowany (proces zinstytucjonalizowany)
  - 4 poziom zarządzany (proces + informacje zwrotne dla sterowania procesem)
  - 5 poziom optymalizujący (proces + informacje zwrotne wpływające na ulepszenie procesu
- Niewiele firm uzyskało poziom 3-ci, umożliwiający uzyskanie prawa dostaw dla Departamentu Obrony USA,
- Tylko IBM w zakresie oprogramowania promu kosmicznego dla NASA uzyskał poziom 5-ty (najwyższy) \
- Klasyfikacja postrzegana także jako arbitralna, tendencyjna i protekcjonistyczna

Główne czynniki poprawy jakości

- Poprawa zarządzania projektem
- Wzmocnienie inżynierii wymagań
- Zwiększenie nacisku na jakość
- Zwiększenie nacisku na kwalifikacje i wyszkolenie ludzi
- Szybsze wykonywanie pracy (lepsze narzędzia) 10%
- Bardziej inteligentne wykonywanie pracy (lepsza organizacja i metody) 20%
- Powtórne wykorzystanie pracy już wykonanej (ponowne użycie) 65 %

Plan zapewnienia jakości oprogramowania (PZJO)

PZJO powinien być sporządzany i modyfikowany przez cały okres życia oprogramowania. Pierwsze jego wydanie powinno pojawić się na końcu fazy wymagań użytkownika.

- PZJO powinien ustalać i opisywać wszelkie aktywności związane z zapewnieniem jakości dla całego projektu. Odpowiednie sekcje planu jakości powinny dotyczyć wszystkich ustalonych w danym modelu rozwoju oprogramowania faz cyklu życia oprogramowania.
- Podane dalej zalecenia co do PZJO pochodzą z normy ANSI/IEEE Std 730-1989 „*IEEE Standard for Software Quality Assurance Plan*”.
- Dodatkowe wytyczne co do PZJO pochodzą z ANSI/IEEE Std 983-1989 „*IEEE Guide for Software Quality Assurance Plan*”.
- Rozmiar i zawartość PZJO powinny odpowiadać skali i złożoności projektu.
- Zalecany (podany dalej) spis treści może i niekiedy powinien być uzupełniony o punkty specyficzne dla konkretnego projektu.

Styl, odpowiedzialność, sekcje PZJO

- **Styl**. PZJO powinien być zrozumiały, lakoniczny, jasny spójny i modyfikowalny.
- **Odpowiedzialność**. PZJO powinien być wyprodukowany przez komórkę jakości zespołu podejmujący się produkcji oprogramowania.
- PZJO powinien być przejrzany i zrecenzowany przez ciało, któremu podlega dana komórka jakości oprogramowania.
- **Medium**. Zwykle PZJO jest dokumentem papierowym, może być także rozpowszechniony w formie elektronicznej.
- **Zawartość**. PZJO powinien być podzielony na 4 rozdziały, każdy dla następujących faz rozwoju oprogramowania:
  - PZJO dla fazy wymagań użytkownika i analizy;
  - PZJO dla fazy projektu architektury;
  - PZJO dla fazy projektowania i konstrukcji;
  - PZJO dla fazy budowy, testowania i instalacji oprogramowania.
- **Ewolucja**. PZJO powinien być tworzony dla następnej fazy po zakończeniu fazy poprzedniej.

Spis treści PZJO (1)

Informacje organizacyjne

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

Zasadnicza zawartość dokumentu

1. Cel
2. Referencje, odsyłacze do innych dokumentów
3. Zarządzanie
4. Dokumentacja
5. Standardy, praktyki konwencje i metryki
- 5.1. Standardy dokumentacyjne
- 5.2. Standardy projektowe
- 5.3. Standardy kodowania
- 5.4. Standardy komentowania
- 5.5. Standardy i praktyki testowania
- 5.6. Wybrane metryki ZJO
- 5.7. Ustalenia dotyczące sposobu monitorowania zgodności z planem

..... na następnym slajdzie

Spis treści PZJO (2)

Zasadnicza zawartość dokumentu

..... z poprzedniego slajdu

6. Przeglądy i audyty
7. Testowanie
8. Raportowanie problemów i akcje korygujące
9. Narzędzia, techniki i metody
10. Kontrola kodu
11. Kontrola mediów
12. Kontrola dostawców
13. Zbieranie, pielęgnacja i utrzymanie zapisów
14. Szkolenie
15. Zarządzanie ryzykiem
16. Przegląd pozostałej części projektu
Dodatek A: Słownik pojęć i akronimów

Numeracja punktów nie powinna być zmieniana. Jeżeli pewien punkt nie ma treści, powinna tam znajdować się informacja „Nie dotyczy”.
Informacje nie mieszczące się w tym spisie treści powinny być zawarte w dodatkach.
Punkty 3-15 powinny określać jak plany techniczne i zarządzania będą sprawdzane.

Spis treści PZJO - omówienie (1)

- **Cel**. Sekcja ta powinna krótko określać: cel PZJO, rodzaj czytelnika, produkty programistyczne podlegające PZJO, zamierzone użycie oprogramowania, fazę cyklu życiowego, do którego PZJO się odnosi.
- **Zarządzanie**. Sekcja powinna opisywać organizację zarządzania jakością i związane z nią odpowiedzialności i role, bez określania przypisania ludzi do ról i bez określania pracochłonności i harmonogramu. Zalecana jest następująca struktura tego rozdziału:
  - **Organizacja**: identyfikacja ról w organizacji (kierownik projektu, prowadzący zespołu, inżynierowie oprogramowania, bibliotekarze oprogramowania, prowadzący weryfikację i walidację , inżynier ZJO), opis związków pomiędzy rolami, opis interfejsu z organizacją użytkownika.
  - **Zadania**: Opisuje zadania ZJO, które będą wykonywane w tej fazie.
  - **Odpowiedzialności**: Opisuje odpowiedzialność poszczególnych ról za poszczególne zadania oraz ustala kolejność wybranych zadań.

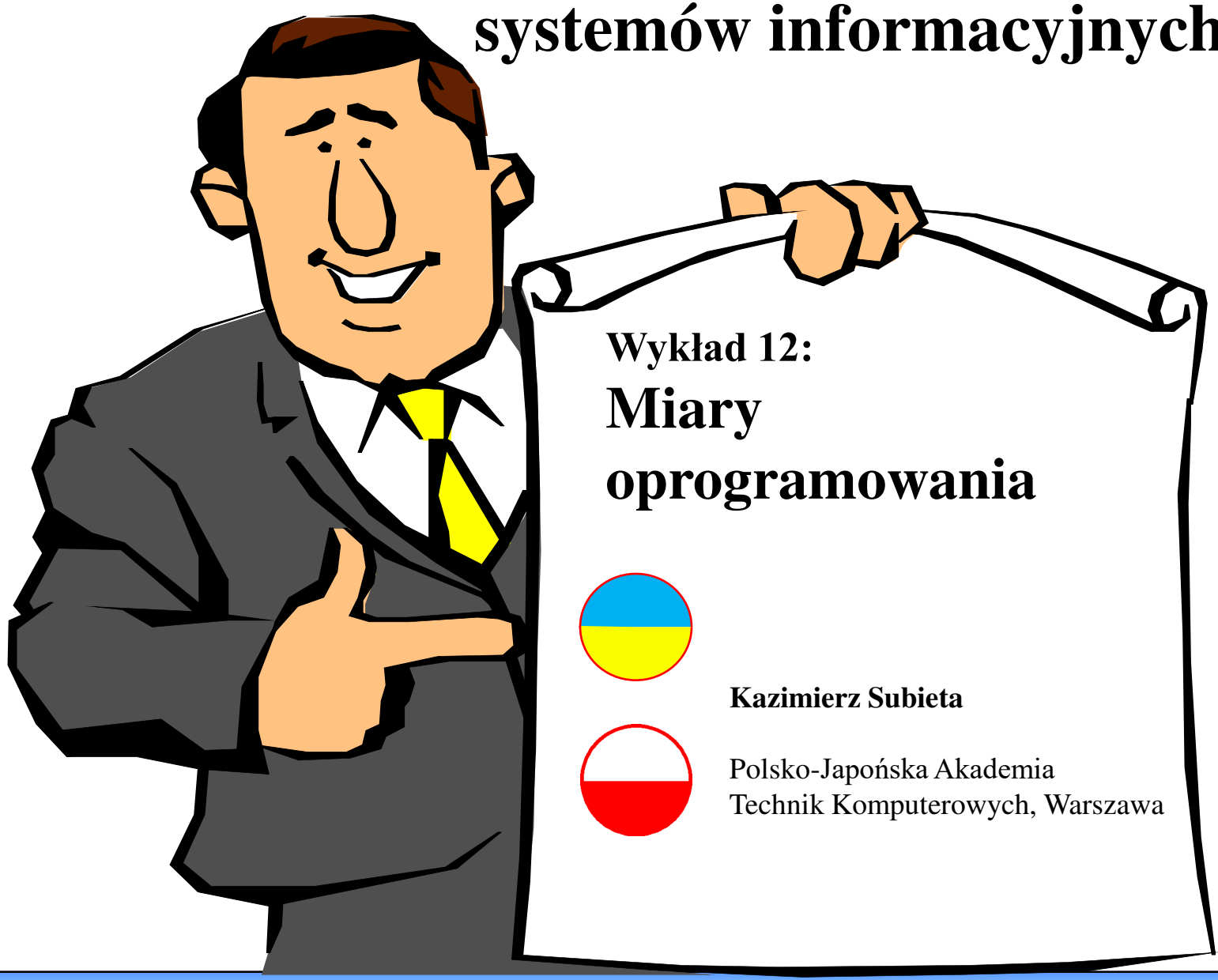
Spis treści PZJO - omówienie (2)

- **Dokumentacja**. Identyfikuje wszystkie dokumenty, które będą wyprodukowane w tej fazie. Sekcja powinna ustalać jak te dokumenty będą sprawdzane na zgodność ze standardami.
- **Standardy, konwencje metryki**. Opisuje je detalicznie lub zawiera odsyłacze do innych dokumentów.
- **Przeglądy i audyty**. Identyfikuje techniczne przeglądy, przejścia, inspekcje, audyty mające zastosowanie w tej fazie, oraz cel każdego z nich. Opisuje sposoby monitorowania zgodności tych procedur z planem oraz rolę personelu ZJO w tych procedurach.
- **Testy**. Opisuje w jaki sposób czynności weryfikacji i walidacji oprogramowania będą monitorowane i sprawdzane.
- **Raportowanie problemów i akcje korygujące**. Identyfikuje procedury zgłaszania problemów oraz podejmowania akcji mających na celu usunięcie problemów. Może opisywać metryki stosowane do procedur zgłaszania problemów mające wpływ na jakość oprogramowania.

Spis treści PZJO - omówienie (3)

- **Kontrola kodu**. Procedury stosowane do pielęgnacji, przechowywania, zabezpieczania i dokumentowania kodu oprogramowania.
- **Kontrola mediów**. Dotyczy mediów, na których oprogramowanie i dokumentacja będą przechowywane.
- **Kontrola dostawców**. Procedury stosowane do kontroli zewnętrznych organizacji lub osób, które rozwijają lub dostarczają oprogramowanie niezbędne dla projektu. Procedury powinny określać standardy, które mają być stosowane przez dostawców, oraz powinny określać sposoby kontroli przestrzegania tych standardów.
- **Zbieranie, pielęgnacja i utrzymanie zapisów**. Identyfikuje procedury stosowane do przechowywania informacji zebranych ze wszelkich aktywności, takich jak spotkania, przeglądy, przejścia, audyty, notatki, korespondencja. Powinny określać gdzie te informacje/dokumenty są przechowywane i jak długo, oraz określać sposób dostępu do tych informacji.

Budowa i integracja systemów informacyjnych



Wykład 12:
Miary oprogramowania

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Informacja z portalu Interia:

Sześć znaków zodiaku, które często się rozwodzą

Chociaż nie każdy musi wierzyć w to, że astrologia ma realny wpływ na nasze życie, na całym świecie jest mnóstwo ludzi, którzy regularnie czytają horoskopy i analizują swoje kosmogramy po to, aby dowiedzieć się o sobie jak najwięcej. Czy znak zodiaku może mieć wpływ na trwałość naszych związków? Okazuje się, że to możliwe!

Astrologia

- W 18-tym wieku wyrzucona ze wszystkich uniwersytetów jak pseudo-nauka nie mająca **żadnych** materialnych lub logicznych podstaw.
- Badania na Uniwersytecie w Manchesterze dla 20 milionów małżeństw ustaliły, że **astrologii przeczy zwykła statystyka** (czego można było oczekiwać).
- Psychologia wyjaśnia zjawisko irracjonalnej wiary w astrologiczne przepowiednie (**efekt Barnuma**).
- W telewizji i Internecie astrologia znowu rozkwitła, ponieważ jest to interesujące dla przeciętnej osoby niewyrobionej intelektualnie (delikatnie mówiąc).
- Są osoby, które na astrologii i przepowiadaniu przyszłości dorobiły się majątku.
- Jako **nieszkodliwa rozrywka** może być akceptowana, ale często jest traktowana poważnie, jako życiowy drogowskaz - a to już jest chore.
- Z punktu widzenia nauki i kultury, **astrologia jest szkodliwa:** pozorując naukę podważa autorytet i misję nauki, daje zbyt duże pole dla wydrwigroszy.

Konkluzja:

Cała wiedza astrologiczna, łącznie ze znakami zodiaku, to szkodliwe średniowieczne bzdury.

Astrologia a miary oprogramowania

- W projektach programistycznych miary oprogramowania są formą przewidywania przyszłości (zatem mają wiele wspólnego z astrologią).
- Astrologii w projektach programistycznych chcielibyśmy uniknąć.
- Ale w naszej strefie kulturowo-cywilizacyjnej **musimy przewidywać przyszłość** (np. ile projekt będzie kosztować).
- Z jednej strony mamy więc ostrzeżenie przed astrologią, z drugiej strony przewidywania przyszłości nie możemy uniknąć!
- Możemy być **ukarani** za złe przewidywanie przyszłości!
  - Rozmazana granica pomiędzy złą prognozą, niegospodarnością i przekrętem
- Czy możemy myśleć o metodach przewidywania przyszłości w terminach obiektywnych, racjonalnych lub naukowych?

Prawo GIGO

- „Garbage In – Garbage Out”
- Prawo to głosi, że nie da się uzyskać wiarygodnej informacji z szumu informacyjnego
  - Jeżeli na wejściu jest szum informacyjny (losowe sygnały), to nie da się z tego szumu wyabstrahować wiarygodnej informacji niezależnie od metody czy urządzenia przetwarzającego – procedura, reguła, sztuczna inteligencja, metody matematyczne, itp.
  - Informacyjne „perpetum mobile” (zwane też demonem Maxwela 2-go rodzaju - Lem) nie może istnieć
- Wiele metod nie tylko w inżynierii oprogramowania próbuje zaprzeczyć temu prawu
- Do takim metod należy **większość** metod szacowania kosztów, czasu i innych parametrów oprogramowania

Pomiary 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: każde określone działanie w ramach projektu, 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 - procesy

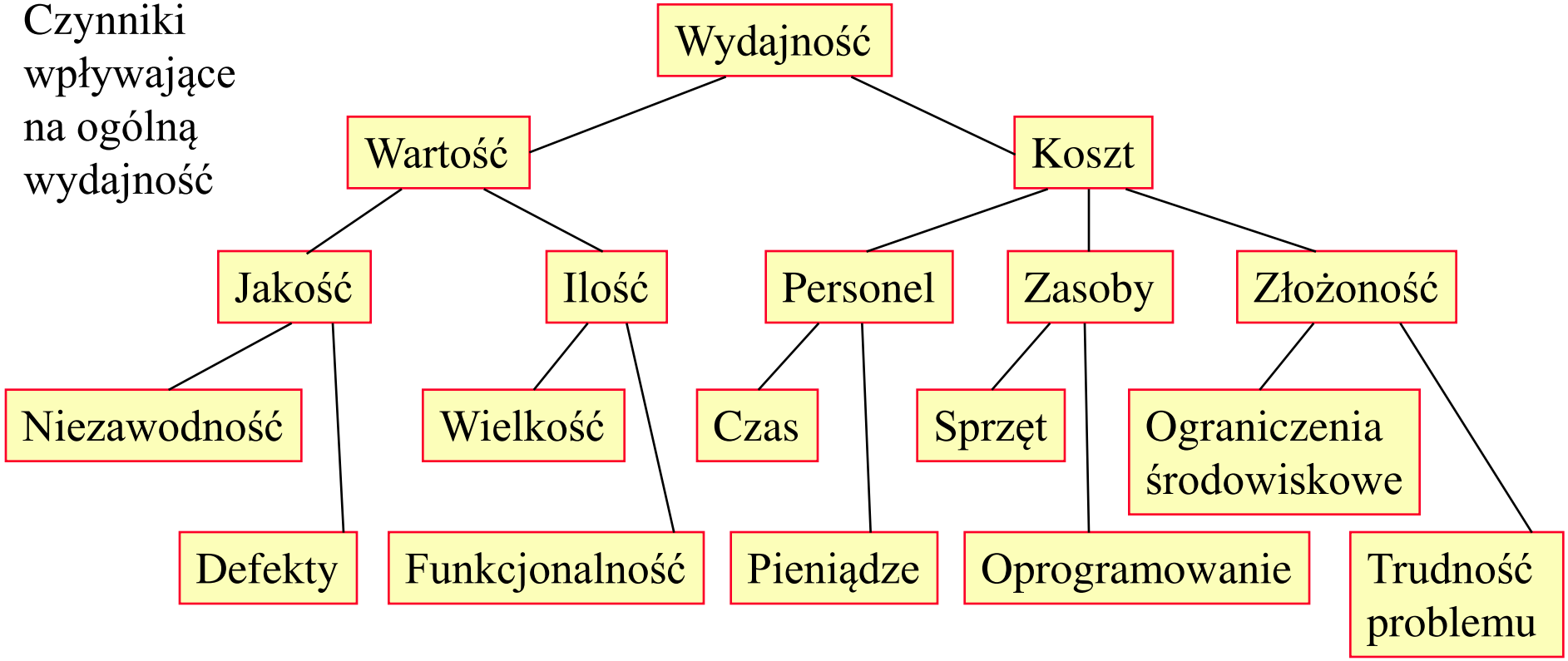
Obiekty	**Atrybuty bezpośrednio mierzalne**	**Wskaźniki syntetyczne**
Specyfikacja architektury	czas, nakład pracy, liczba zmian wymagań, ...	jakość, koszt, stabilność, ...
Projekt szczegółowy	czas, nakład pracy, liczba znalezionych usterek specyfikacji,...	koszt, opłacalność, ...
Testowanie	czas, nakład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

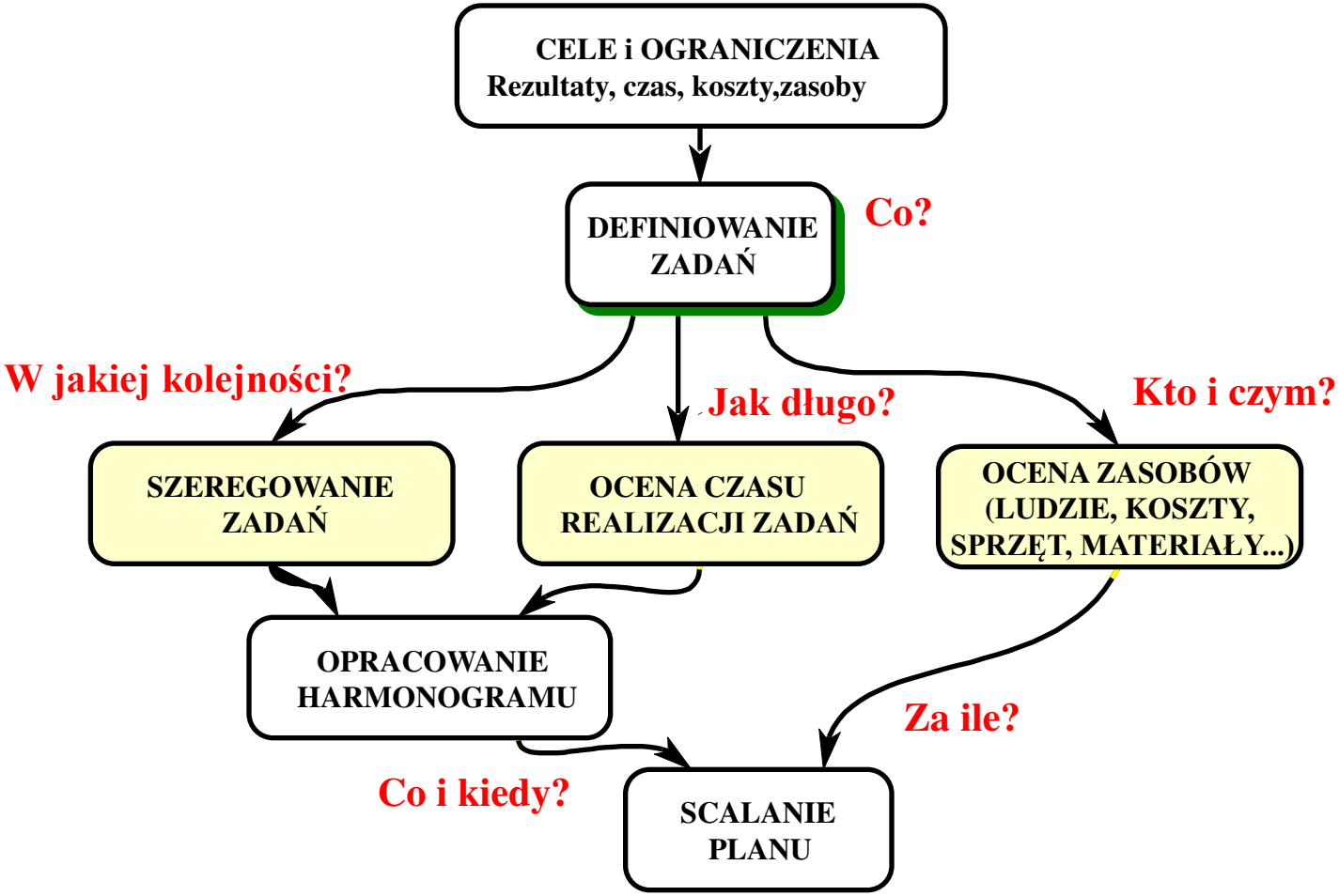
Modele i miary wydajności ludzi

Czynniki wpływające na ogólną wydajność



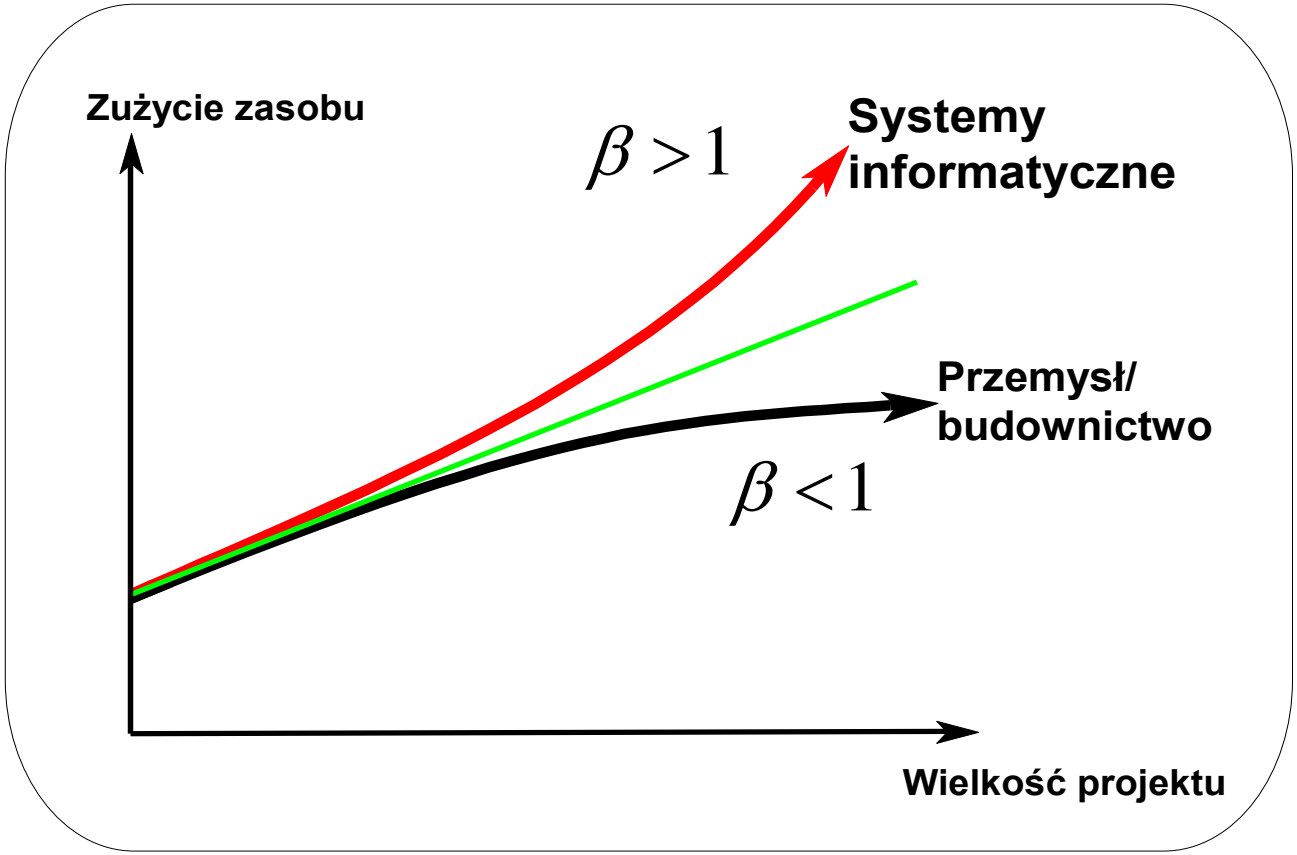
Mylące, wręcz niebezpieczne jest zastępowanie wielu miar jedną miarą, np. długością wyprodukowanego kodu.

Ocena złożoności w planowaniu projektu



Efekt skali

$$\text{ZużycieZasobu} = \text{ZużycieStałe} + K * \text{WielkośćProjektu}^{\beta}$$



Czynniki wpływające na efekt skali

Czynniki spłaszczenia krzywej

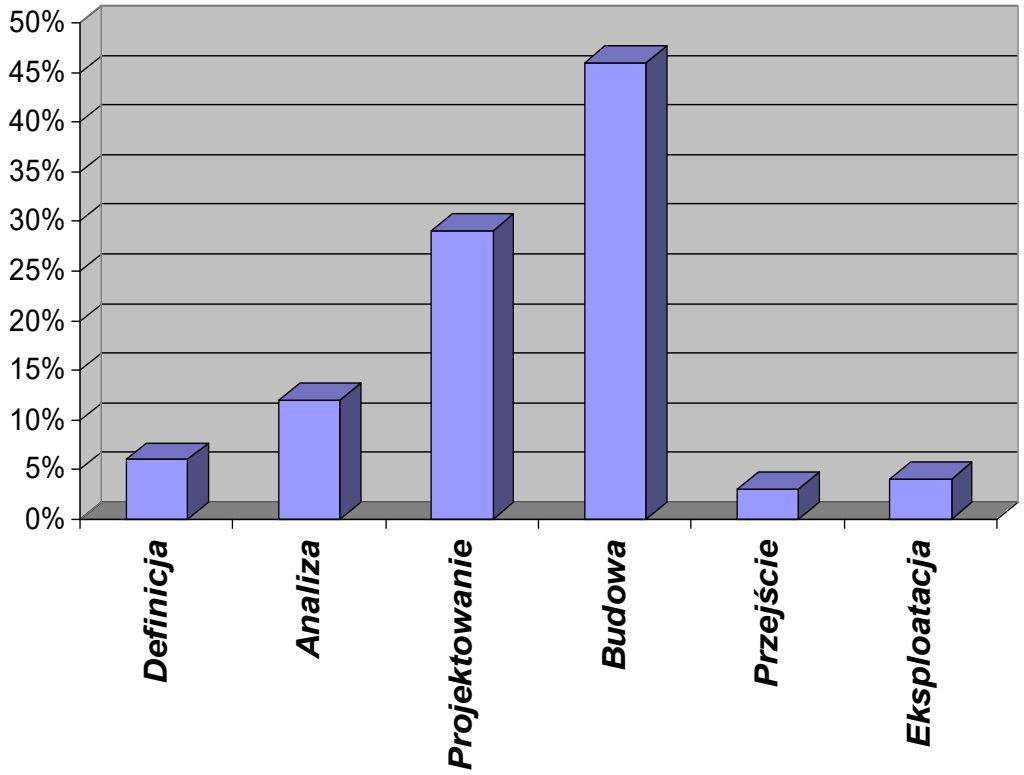
- Specjalizacja
- Uczenie się, doświadczenie
- Narzędzia CASE
- Wspomaganie dokumentowania
- Biblioteki gotowych elementów
- Stałe koszty projektu

Czynniki wzrostu krzywej

- Koszty zarządzania (czas produkcyjny/nie)
- Lawinowy wzrost liczby powiązań
- Integracja modułów
- Komunikacja wewnątrz zespołu
- Eksplozja złożoności testowania

Etapy i 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 (1)

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** (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:

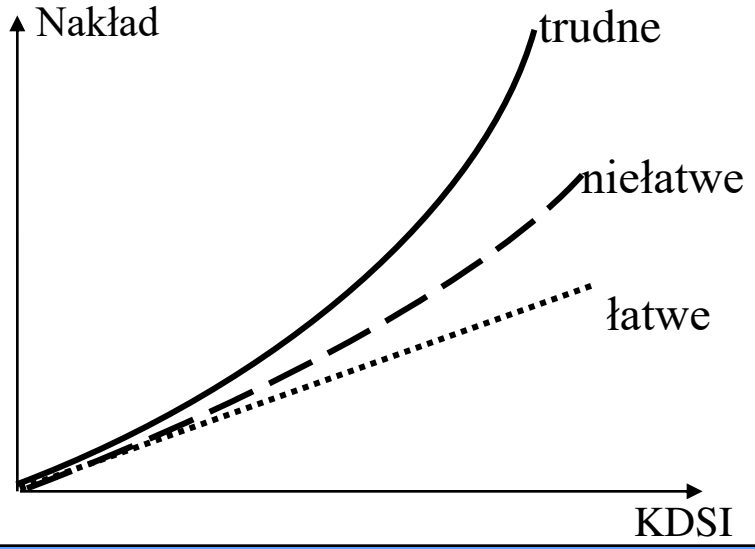
$$\text{Nakład[osobomiesiące]} = A * K^{b}$$

(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:	Nakład = $2.4 * K^{1.05}$
Przedsięwzięcie niełatwe:	Nakład = $3 * K^{1.12}$
Przedsięwzięcie trudne:	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: $\text{Czas[miesiące]} = 2.5 * \text{Nakład}^{0.32}$

Przedsięwzięcie niełatwe: $\text{Czas[miesiące]} = 2.5 * \text{Nakład}^{0.35}$

Przedsięwzięcie trudne: $\text{Czas[miesiące]} = 2.5 * \text{Nakład}^{0.38}$

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. Czy niedoskonała metoda jest lepsza niż “sufit”?

Analiza Punktów Funkcyjnych

Metoda analizy punktów funkcyjnych (FPA), opracowana przez Albrechta 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 produkt programowy
- **Zapytania zewnętrzne:** interfejsy z otoczeniem programu

UFP - nieskorygowane punkty

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

UFP- nieskorygowane punkty funkcyjne

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

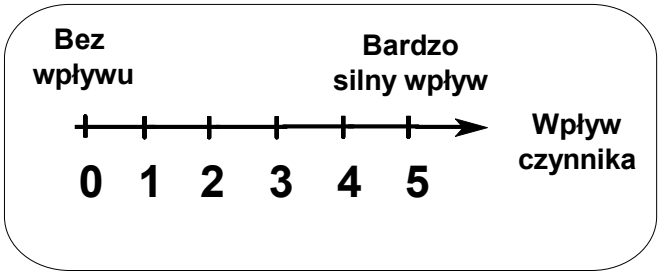
		Wagi przypisywane projektom:		
	Czynnik złożoności	**Projekt prosty**	**Projekt średni**	**Projekt złożony**
i = 1	Wejścia użytkownika	3	4	6
i = 2	Wyjścia użytkownika	4	5	7
i = 3	Zbiory danych wewnętrzne	7	10	15
i = 4	Zbiory danych zewnętrzne	5	7	10
i = 5	Zapytania zewnętrzne	3	4	6
		j = 1	j = 2	j =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

Korekcja Punktów Funkcyjnych

- występowanie urządzeń komunikacyjnych
- rozproszenie przetwarzania
- długość czasu oczekiwania na odpowiedź systemu
- stopień obciążenia sprzętu istniejącego
- częstotliwość wykonywania dużych transakcji
- wprowadzanie danych w trybie bezpośrednim
- wydajność użytkownika końcowego
- aktualizacja danych w trybie bezpośrednim
- złożoność przetwarzania danych
- możliwość ponownego użycia programów w innych zastosowaniach
- łatwość instalacji
- łatwość obsługi systemu
- rozproszenie terytorialne
- łatwość wprowadzania zmian - pielęgnowania systemu

19 - British Mark II

Skorygowane Punkty Funkcyjne



kompleksowy współczynnik korygujący

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

Punkty funkcyjne (FPs):

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

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

Kolejność obliczeń Punktów Funkcyjnych

- Identyfikacja systemu
- Obliczenie współczynnika korygującego
- Wyznaczenie ilości zbiorów danych i ich złożoności
- Wyznaczenie ilości i złożoności elementów funkcjonalnych (we, wy, zapytania)
- Realizacja obliczeń
- Weryfikacja
- Raport, zebranie recenzujące

Przykład obliczania punktów funkcyjnych

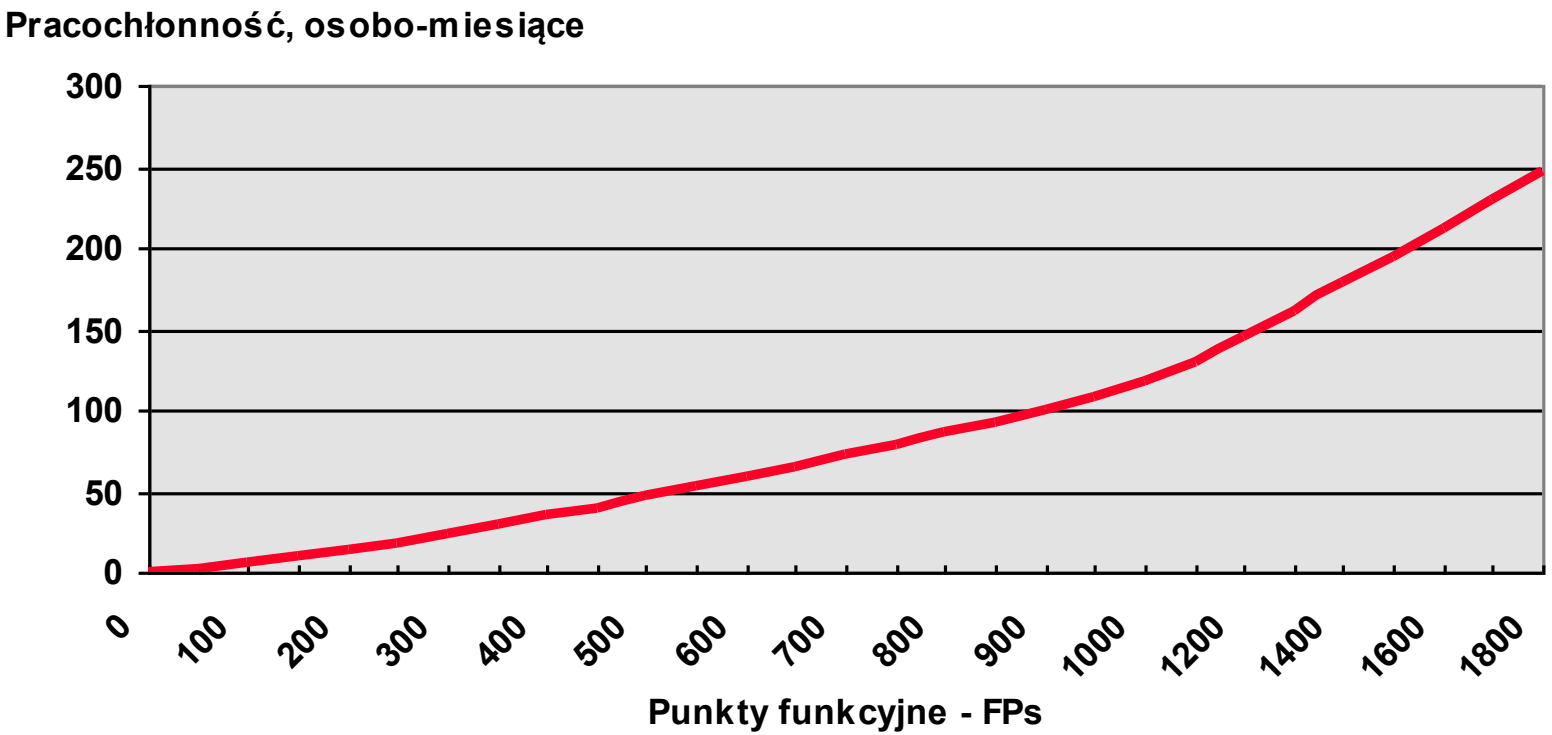
Elementy	**Proste**	Waga	**Średnie**	Waga	**Złożone**	Waga	**Razem**
Wejścia	**2**	3	**5**	4	**3**	6	**44**
Wyjścia	**10**	4	**4**	5	**5**	7	**95**
Zbiory wew.	**3**	7	**5**	10	**2**	15	**101**
Zbiory zew.	**0**	5	**3**	7	**0**	10	**21**
Zapytania	**10**	3	**5**	4	**12**	6	**122**
							Łącznie 383

Aplikacje a Punkty Funkcyjne

- 1 FP ≈ 125 instrukcji w C
- 10 FPs - typowy mały program tworzony samodzielnie przez programistę (1 m-c)
- 100 FPs - większość popularnych aplikacji; wartość typowa dla aplikacji tworzonych przez programistę 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 (100 osób, ponad 18 m-cy)
- 100,000 FPs - MS Windows, MVS, systemy militarne

Podobne metody: Mark IV, Cosmic

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

Inne metody pomiarów i estymacji

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..)

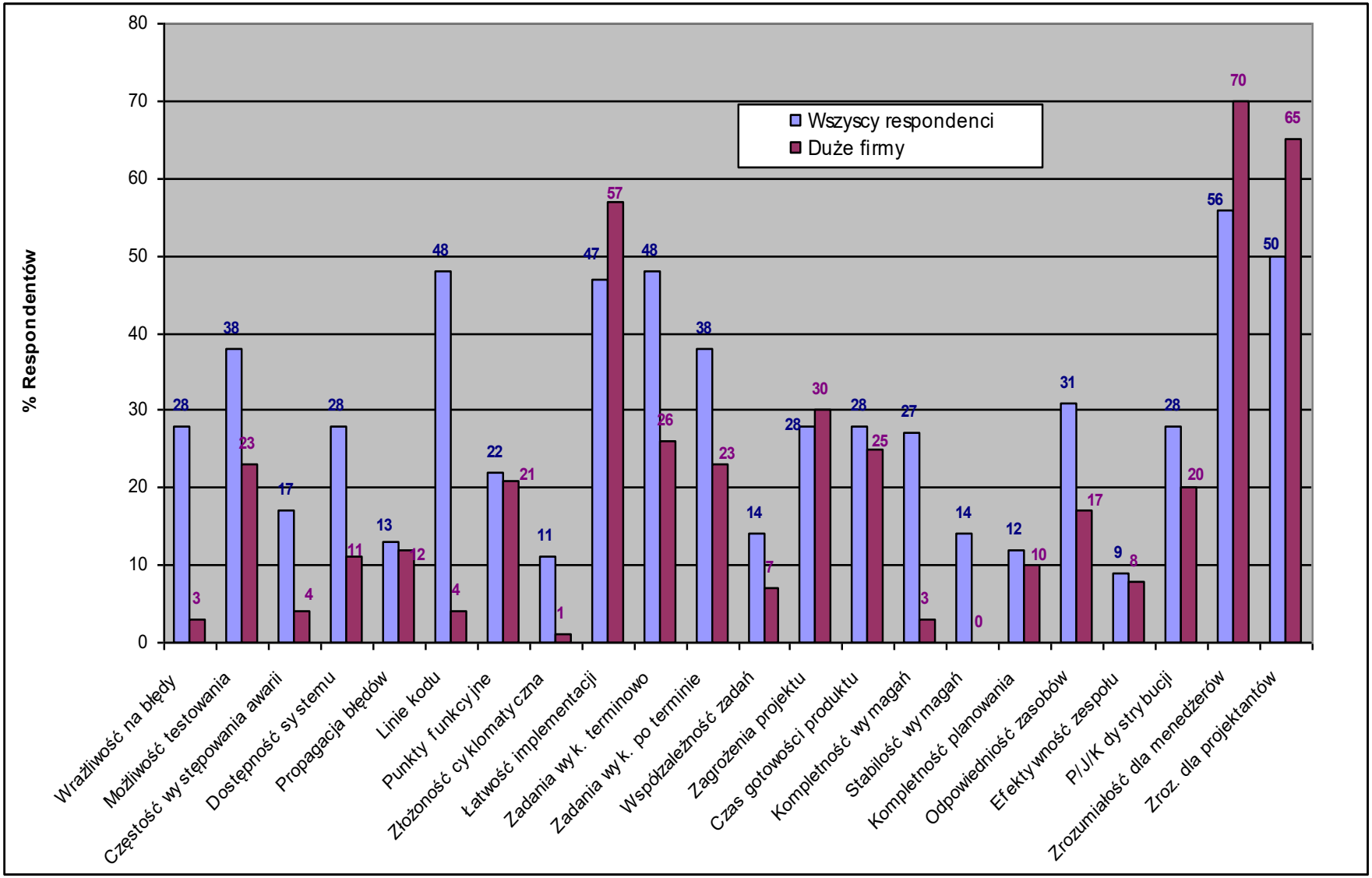
- Metryki uzyskiwanego produktu
  - » rozmiar
  - » architektura
  - » struktura
  - » jakość użytkowania i pielęgnacji
  - » złożoność

Przykłady metryk oprogramowania, cd.

- Metryki procesu wytwarzania
  - » dojrzałość realizacji systemu
  - » zarządzanie wytwarzaniem oprogramowania
  - » w odniesieniu do cyklu życia oprogramowania

- Metryki zasobów realizacyjnych
  - » w odniesieniu do personelu „zamieszanego” w realizację
  - » narzędzia software’owe, wykorzystywane przy realizacji
  - » sprzęt, jakim dysponuje wykonawca

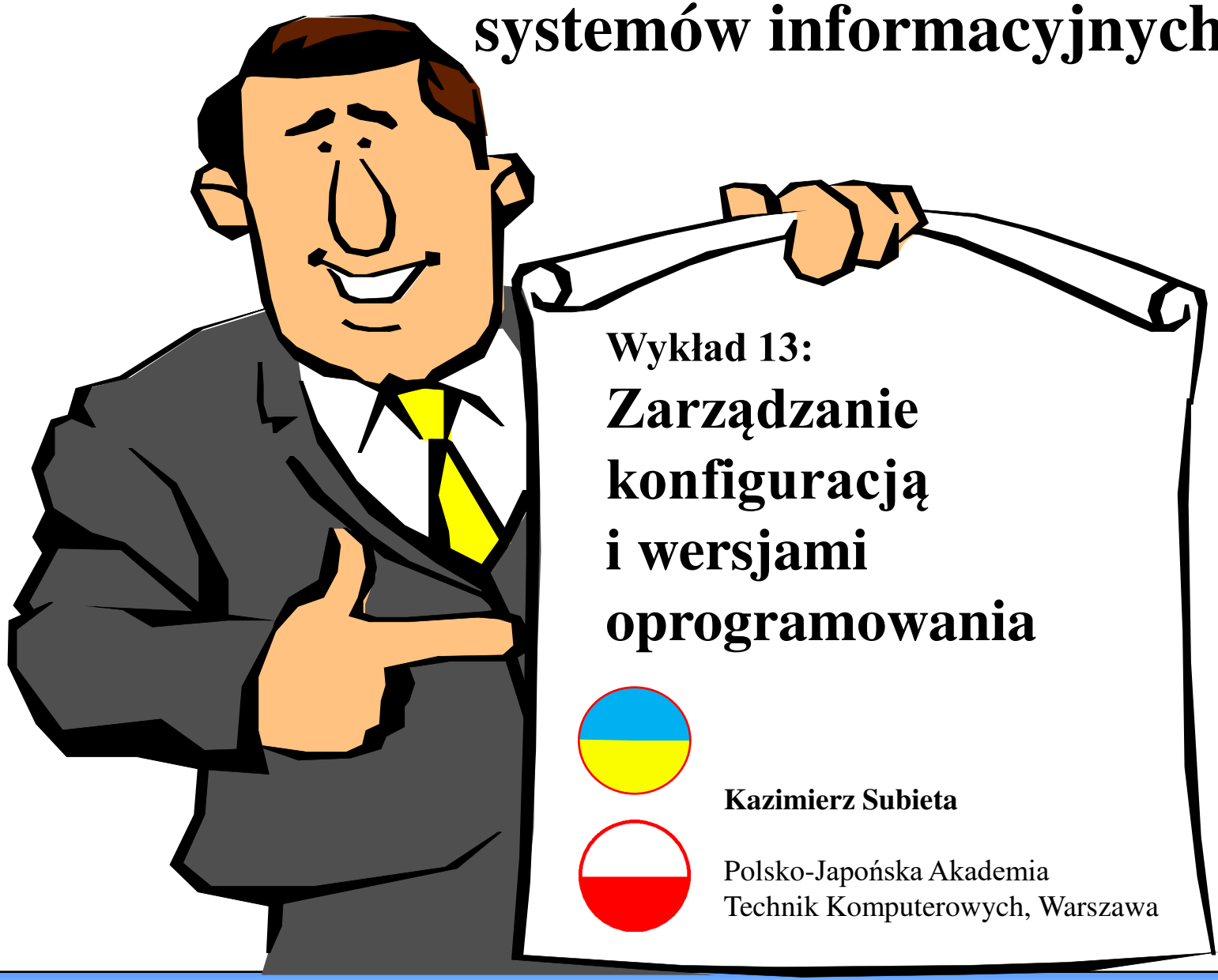
Wykorzystanie metod estymacyjnych



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. Metryki stosowane formalistycznie mogą być zawodne.
- 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.

Budowa i integracja systemów informacyjnych



Wykład 13:
Zarządzanie konfiguracją i wersjami oprogramowania

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Entropia

Termin pochodzący z termodynamiki oznaczający

miarę chaosu

Stosowany też w innych kontekstach, np. w (zapomnianej) teorii informacji.

Prawo entropii:

- **Jeżeli rzeczy pozostawimy losowi i nie będziemy wkładać tam dodatkowej energii, to entropia będzie rosła.**

Co oznacza coraz większy **chaos** (zwany też niekiedy bałaganem).

  - Wiele osób testuje to prawo w swoim mieszkaniu, użytkując go bez jakiejkolwiek dodatkowej energii (na sprzątanie).
- Prawo dotyczy także (a może szczególnie) **projektów oprogramowania**.
- Chaos jest czynnikiem silnie destrukcyjnym.
- Jak zwalczyć chaos? Jak zarządzać, aby nie było bałaganu?

Zarządzanie konfiguracją oprogramowania, ZKO

software configuration management, SCM

Celem zarządzania konfiguracją oprogramowania jest planowanie, organizowanie, sterowanie i koordynowanie działań mających na celu identyfikację, przechowywanie i zmiany oprogramowania w trakcie jego rozwoju, integracji i przekazania do użycia.

Każdy projekt musi podlegać konfiguracji oprogramowania. Ma ono krytyczny wpływ na jakość końcowego produktu. Jest niezbędne dla efektywnego rozwoju oprogramowania i jego późniejszej pielęgnacyjności.

ZKO jest szczególnie ważne, jeżeli projekt może toczyć się przez wiele lat, jeżeli cel lub wymagania na oprogramowanie są niestabilne, jeżeli oprogramowanie może mieć wielu użytkowników, i/lub jeżeli oprogramowanie jest przewidziane na wiele platform sprzętowo-programowych.

W takich sytuacjach złe zarządzanie konfiguracją oprogramowania może całkowicie sparaliżować projekt.

ZKO jest podstawowym składnikiem norm jakości oprogramowania.

ZKO powinno zapewniać, że ...

- Każdy komponent oprogramowania będzie jednoznacznie identyfikowany;
- Oprogramowanie będzie zbudowane ze spójnego zestawu komponentów;
- Zawsze będzie wiadomo, która wersja komponentu oprogramowania jest najnowsza;
- Zawsze będzie wiadomo, która wersja dokumentacji pasuje do której wersji komponentu oprogramowania;
- Komponenty oprogramowania będą zawsze łatwo dostępne;
- Komponenty oprogramowania nigdy nie zostaną stracone (np. wskutek awarii nośnika lub błędu operatora);
- Każda zmiana oprogramowania będzie zatwierdzona i udokumentowana;
- Zmiany oprogramowania nie zaginą (np. wskutek jednoczesnych aktualizacji);
- Zawsze będzie istniała możliwość powrotu do poprzedniej wersji;
- Historia zmian będzie przechowywana, co umożliwi odtworzenie kto i kiedy zrobił zmianę, i jaką zmianę.

Zadania kierownictwa projektu w zakresie ZKO

- **Kierownictwo projektu** jest odpowiedzialne za organizowanie aktywności związanych z ZKO, zdefiniowanie ról personelu (np. bibliotekarza oprogramowania) oraz przypisanie ról do personelu.
- **Kierownictwo projektu** musi wymagać dokładnej identyfikacji wszystkich komponentów składających się na projekt oprogramowania i określania ich statusu (np. wstępny, roboczy, zatwierdzony, końcowy).
- **Personel rozwijający oprogramowanie** powinien dzielić między siebie komponenty w sposób bezpieczny i efektywny.
- **Personel zapewnienia jakości oprogramowania** musi mieć możliwość śledzenia pochodzenia i rozwijania każdego komponentu oraz ustalania kompletności i poprawności każdej konfiguracji.
- Wdrożone przez kierownictwo **procedury lub system ZKO** powinny zapewniać przejrzystość projektu i produktu dla wszystkich zainteresowanych stron.

Pozycja konfiguracji oprogramowania

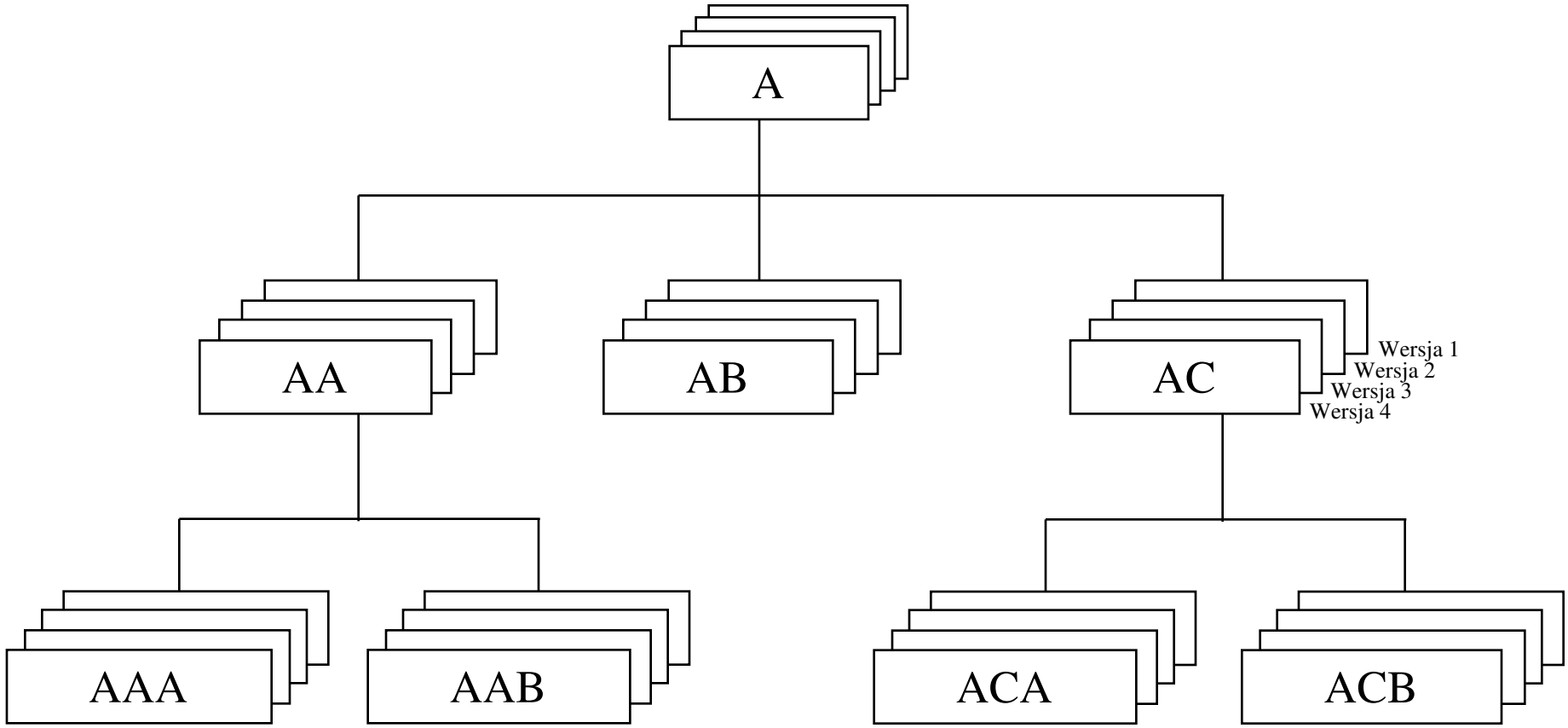
configuration item

Wszystkie elementy projektu i oprogramowania muszą być przedmiotem ZKO, w szczególności:

- dokumentacja: wymagań, analityczna, projektowa, testowania, użytkownika, itd.
- moduły z kodem źródłowym, kody do konsolidowania, kody binarne,
- ekrany interfejsu użytkownika,
- pliki z danymi tekstowymi (np. komunikatami systemu), bazy danych, słowniki, itd.
- kompilatory, konsolidatory, interpretery, biblioteki, protokoły, narzędzia CASE, konfiguracje sprzętowe, itd.
- oprogramowanie testujące, dane testujące,
- serwery WWW wraz z odpowiednimi stronami HTML i oprogramowaniem,
- ...

Wyróżnialny element uczestniczący w projekcie lub produkcie będzie określany jako **„pozycja konfiguracji”.** Jest ona traktowana jako pojedynczy, możliwy do odseparowania komponent projektu lub produktu programistycznego.

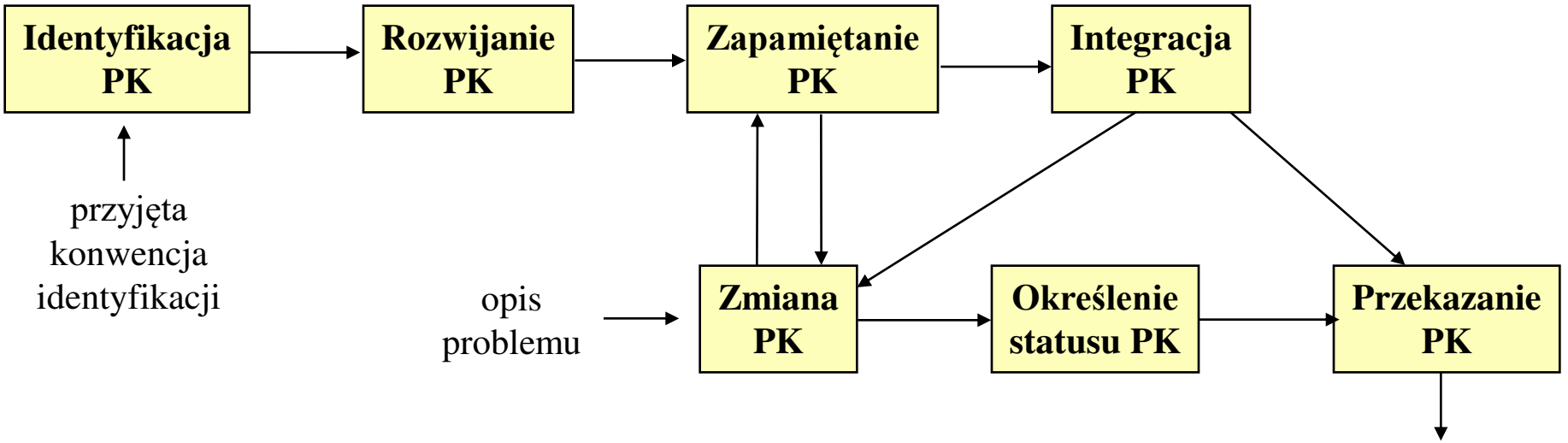
Hierarchia pozycji konfiguracji



Pozycja konfiguracji może istnieć w wielu wersjach oraz może być agregatem złożonym z pozycji konfiguracji. Wszystkie pozycje konfiguracji, od atomowych modułów do całkowitej ukończonej wersji oprogramowania, muszą być zdefiniowane tak wcześnie jak to możliwe i systematycznie oznaczone w momencie utworzenia.

Aktywności ZKO

- **Identyfikacja pozycji konfiguracji (PK)**
- **Przechowywanie pozycji konfiguracji (PK)**
- **Kontrola zmian konfiguracji**
- **Określanie statusu konfiguracji**
- **Przekazanie pozycji konfiguracji na zewnątrz** (*release*)



Produkt bazowy

baseline

Produktem bazowym **jest pozycja konfiguracji oceniona i zaakceptowana formalnie przez odpowiednie ciało weryfikacyjne jako zakończona, stanowiąca podstawę do dalszych faz rozwoju projektu.**

- W szczególności, zakończony projekt jest produktem bazowym.
- Rozwój projektu postępuje od produktów bazowych do kolejnego produktu bazowego. Produkt bazowy jest podstawą formalnego rozliczenia wykonawców.
- Wczesne produkty bazowe zawierają dokumenty analityczne i projektowe. Późniejsze produkty bazowe zawierają również kod.
- Poszczególne PK w produkcie bazowym muszą być wzajemnie spójne.
- Produkty bazowe są niemodyfikowalne. Są one podstawę wymiany informacji pomiędzy uczestnikami projektu oraz podstawę testów nowego oprogramowania.
- Kluczowe pozycje bazowe są przywiązane do kamieni milowych w planie zarządzania projektem.
- Nowe produkty bazowe są określane w miarę integracji nowych komponentów oprogramowania.

Wersje (warianty)

versions (variants)

Termin *wersja* (lub *wariant*) jest używany dla określenia pozycji konfiguracji, która ma prawie identyczną logikę i przeznaczenie, ale różni się w pewnych aspektach, takich jak:

- docelowa platforma/konfiguracja sprzętowa lub system operacyjny;
- protokół komunikacyjny, współdziałanie z innym (zewnętrznym) oprogramowaniem;
- język użytkownika (komend, komunikatów, menu), np. polski, angielski, itd.;
- specyficzne wymagania poszczególnych użytkowników;
- postępujące w czasie ulepszenia;
- realizacja celów diagnostycznych i testowych podczas rozwoju oprogramowania.

Istnienie wielu wersji PK znacznie komplikuje zarządzanie konfiguracjami. Liczba wersji powinna być minimalizowana. Podstawową metodą jest *parametryzowanie* wytwarzanego kodu celem zwiększenia stopnia jego generyczności. Powoduje ona jednak zwiększenie skomplikowania modułów oprogramowania. Konieczne jest uzyskanie kompromisu pomiędzy złożonością wytwarzanych modułów oprogramowania a złożonością konfiguracji.

Konwencja identyfikacji konfiguracji

identification convention

Pierwszym krokiem w stworzeniu systemu zarządzania konfiguracją oprogramowania jest stworzenie **konwencji identyfikacji**.

- Konwencja identyfikacji jest zbiorem stringów, zwykle złożonym z liter, cyfr i znaków kropki, /, -, itd. umożliwiająca jednoznaczne oznaczenie dowolnej pozycji konfiguracji.
- Konwencja powinna odzwierciedlać hierarchiczną strukturę pozycji konfiguracji, rodzaj pozycji i/lub przypisanie pozycji do projektu. Konwencja powinna odzwierciedlać przyjęte w danej firmie formularze, dokumenty i kody.
- Np. SME/ANA/DW 3.2 oznacza: projekt oznaczony jako SME, pakiet prac (zadanie) oznaczony ANA, DW oznacza dokument wymagań, 3-cia wersja, 2-ga rewizja.
- Konwencja identyfikacji powinna:
  - ustalać jak należy nazywać pozycje konfiguracji;
  - określać kto jest odpowiedzialny za nazwanie danej pozycji konfiguracji;
  - odwzorowywać (w miarę możności) historię danej pozycji konfiguracji.

Co należy identyfikować jako PK?

- Na górnym poziomie cały system/projekt jest PK; powinien on otrzymać unikalny identyfikator. System/projekt składa się z PK niższego rzędu; zwykle ich identyfikatory są poprzedzone identyfikatorem PK wyższego rzędu.
- PK danego projektu może włączać PK innych projektów; w tym przypadku identyfikacja „obcych” PK nie ulega zmianie.
- Na dolnym poziomie znajdują się atomowe PK, np.:
  • poszczególne dokumenty analityczne i projektowe;
  • jednostki kodu źródłowego i wynikowego (pliki kodu, pliki nagłówkowe, itd.) traktowane jako niepodzielne przez oprogramowanie narzędziowe (np. kompilator)
- PK mogą odzwierciedlać podział projektu na zadania.
- Konfiguracje muszą być praktyczne z fizycznego punktu widzenia (t.j. muszą być łatwe do tworzenia, kopiowania, modyfikacji i usuwania) oraz muszą być naturalne z logicznego punktu widzenia (tj. ich cel musi być łatwy do zrozumienia).
- Atomowe PK muszą mieć odpowiednia ziarnistość: zbyt duże są trudne do manipulowania, zbyt małe powodują nadmierne rozdrobnienie i w konsekwencji trudności w utrzymaniu i zarządzaniu.

Jak identyfikować pozycję konfiguracji?

- Identyfikator powinien zawierać nazwę, typ i wersję PK. Nowy identyfikator nie może implikować konieczności zmiany poprzednich identyfikatorów.
- Dobre identyfikatory pozwalają na szybkie zorientowanie się o jaką PK chodzi i na łatwe jej odszukanie. Dokumenty powinny mieć standardowe nazwy. Identyfikatory kodu powinny uwzględniać jego jego hierarchiczną budowę.
- Identyfikator powinien uwzględniać również typ PK. Trzy podstawowe typy:
  - źródłowa PK (np. tekst programu);
  - pochodna PK (np. binarny kod programu);
  - narzędzie dla generowania pochodnej PK ze źródłowej PK (np. kompilator).
- Wszelkie poprawki powinny być dokonywane na wersji źródłowej. Poprawki na wersji wygenerowanej są niedopuszczalne.
- Identyfikatory powinny być wyposażone w numery wersji i rewizji. Każda, nawet najmniejsza zmiana powoduje powstanie nowej pozycji z nowym identyfikatorem.
- Niektóre schematy zarządzania konfiguracją rozróżniają „wersje” i „rewizje”. Rewizje dotyczą drobnych zmian, np. usunięcia błędu. W tym przypadku numer (oznaczenie) wersji składa się z dwóch członów: wersji i rewizji, np, „wersja 5.4”.

Odpowiedzialność za pozycje konfiguracji

- Trzy poziomy odpowiedzialności:
  - autor kodu (programista) lub dokumentacji;
  - kierownik projektu;
  - ciało kontrolno-rewizyjne.
- Duże projekty mogą posiadać więcej poziomów, zgodnie z fazami kontroli i weryfikacji oprogramowania.
- Kierownik projektu jest odpowiedzialny za połączenie PK niższego poziomu (kod, dokumentacja) w PK wyższego poziomu (konfiguracje).
- PK niższego poziomu są przechowywane w bibliotece/repozytorium. Kierownik projektu jest odpowiedzialny za udokumentowanie PK wyższego poziomu. Dobre repozytorium powinno także przechowywać informację o PK wyższego poziomu.
- Dla celów większych projektów konieczne jest powołanie funkcji lub stanowiska *bibliotekarza oprogramowania*.
- Ciało kontrolno-rewizyjne aprobuje produkty bazowe i zmiany do produktów bazowych. Ciało to może składać się z kierownika projektu, przedstawiciela klienta, przedstawiciela zarządu firmy, bibliotekarza oprogramowania, personelu zarządzania jakością, itd.

Przechowywanie pozycji konfiguracji

Wszystkie pozycje konfiguracji muszą być przechowywane w sposób bezpieczny, systematyczny i dobrze zorganizowany - jak książki w bibliotece.

- System przechowywania PK musi dotyczyć wszystkich mediów - elektronicznych, papierowych i innych.
- Powinien istnieć system ewidencji i rejestracji zależności pomiędzy pozycjami konfiguracji. Dobrze zorganizowany system powinien być oparty na bazie danych oraz integrować informacje o PK z samymi (elektronicznymi) PK.
- System powinien także rejestrować i przechowywać wszelkie dokumenty administracyjne związane z projektami oprogramowania, takie jak raporty etapowe i końcowe, zlecenia, raporty zaistniałych problemów, raporty z testów, itd.
- Dokumenty administracyjne powinny być powiązane z pozycjami konfiguracji w taki sposób, aby można było prześledzić ich historię oraz związki przyczynowo-skutkowe pomiędzy dokumentami i pozycjami konfiguracji.
- Rodzaje bibliotek konfiguracji oprogramowania:
  - biblioteki związane z bieżącym rozwojem oprogramowania;
  - biblioteki ukończonych (bazowych) produktów programistycznych;
  - archiwa (przechowywanie nieaktualnych kodów i dokumentów)

Biblioteki/repozytoria pozycji konfiguracji

- Dobrze zorganizowana biblioteka/repozytorium PK jest cechą fundamentalną dla zarządzania konfiguracjami oprogramowania.
- Biblioteka powinna umożliwiać łatwe odszukanie, odczytanie, wstawienie, zastąpienie i usuwanie dowolnych pozycji konfiguracji.
- **Kluczową cechą biblioteki jest bezpieczeństwo i autoryzowany dostęp:**
  - zminimalizowanie prawdopodobieństwa nieautoryzowanego dostępu;
  - precyzyjne określenie praw dostępu poszczególnych uczestników projektów;
  - uniemożliwienie jednoczesnej aktualizacji tej samej PK przez dwie osoby;
  - uniemożliwienie zmiany pozycji konfiguracji będących produktami bazowymi;
  - minimum możliwości zniszczenia biblioteki poprzez awarię, błąd lub sabotaż;
  - kwestie bezpieczeństwa nie powinny powodować: niewygody w pracy użytkowników, zwiększenia czasów dostępu, istotnych nakładów, itd.
- **Wszystkie PK, elektroniczne i papierowe, muszą mieć etykietę zawierającą:**
  - nazwę projektu;
  - identyfikator pozycji konfiguracji;
  - datę wprowadzenia do repozytorium;
  - krótki opis lub charakterystykę zawartości PK.

Kontrolowanie zmian konfiguracji

- Zmiany są rzeczą normalną podczas rozwoju i ewolucji systemu. Dobre zarządzanie zmianami jest esencją dobrego zarządzania konfiguracjami.
- Zmiany nie powinny prowadzić do utraty informacji. Wszystkie przestarzałe dokumenty (elektroniczne i papierowe) powinny być archiwizowane.
- Osoba odpowiedzialna za zmiany (np. kierownik projektu) koordynuje ich wprowadzenie.
- Repozytorium powinno utrzymywać odpowiednie powiązania pomiędzy PK w taki sposób, aby było wiadomo, że zmiana jednej PK pociąga za sobą zmianę innych PK. Np. zmiana kodu może pociągać za sobą zmianę dokumentacji projektowej, dokumentacji użytkownika, planu testów, itd.
- Zmiany powinny być zweryfikowane przed wprowadzeniem następnych zmian. Podstawą zmian są odpowiednie dokumenty: formularz zmian w oprogramowaniu oraz formularz zmian w dokumentacji.

Określanie statusu konfiguracji

- Jest to zapisywanie informacji i sporządzanie raportów niezbędnych do efektywnego zarządzania konfiguracjami, włączając listowanie identyfikatorów wszystkich zatwierdzonych pozycji, statusu wszystkich proponowanych zmian do konfiguracji oraz statusu implementacji proponowanych zmian.
- Status wszystkich pozycji konfiguracji musi być zapamiętany. Istotne jest przede wszystkim zapis stanu pozycji bazowych.
- W tablicy na następnym slajdzie pokazano rozwój oprogramowania zgodnie z produktami bazowymi (kamieniami milowymi); każda kolumna tablicy odpowiada pewnej fazie życia oprogramowania. Elementy tablicy przedstawiają pozycje konfiguracji, ich wersje i rewizje.
- Tablica taka jest dokumentem przedstawiającym status konfiguracji. Możliwe są inne formy dokumentowania statusu, o ile są one łatwe do manipulowania i czytania.
- Status konfiguracji powinien być aktualizowany na bieżąco i powinien być na bieżąco dostępny dla wszystkich członków zespołu projektowego. Jeżeli program działał poprzedniego dnia, a dzisiaj nie działa, pierwszym pytaniem jest: "*co zostało zmienione?*".

Przykład tablicy statusu konfiguracji

Produkty bazowe →	1	2	3	4	5	6	7
Kamienie milowe → / **PK ↓**	**Zatwier-dzenie DWU**	**Zatwier-dzenie DWO**	**Zatwier-dzenie DAP**	**Pośredni produkt bazowy**	**Zatwier-dzenie DDP**	**Akcep-tacja wstępna**	**Akcep-tacja końcowa**
PZPO	1.0	2.0	3.0	4.0	4.0	4.1	4.2
PZKO	1.0	2.0	3.0	4.0	4.0	4.0	4.0
PWWO	1.0	2.0	3.0	4.0	4.1	4.2	4.3
PZJO	1.0	2.0	3.0	4.0	4.0	4.0	4.0
DWU	1.0	1.1	1.2	1.3	1.4	1.5	1.6
DWO		1.0	1.1	1.2	1.3	1.4	1.5
DAP			1.0	1.1	1.2	1.3	1.4
DDP				1.0	1.1	1.2	1.3
Podr.użytk.				1.0	1.1	1.2	1.3
Program A				1.0	1.1	1.2	1.3
Program B				1.0	1.1	1.2	1.3
Kompilator				5.2	5.2	5.2	5.2
Linker				3.1	3.1	3.1	3.1
Syst.oper.				6.1	6.1	6.1	6.1
Program C					1.0	1.1	1.2
DIO						1.0	1.0

PZPO - Plan Zarządzania Projektem Oprogramowania
PZKO - Plan Zarządzania Konfiguracją Oprogramowania
PWWO - Plan Weryfikacji i Walidacji Oprogramowania
PZJO - Plan Zapewnienia Jakości Oprogramowania

DWU - Dokument Wymagań Użytkownika
DWO - Dokument Wymagań na Oprogramowanie.
DAP - Dokumentacja Analityczno-Projektowa
DDP - Detaliczny Dokument Projektowy
DIO - Dokument Instalacji Oprogramowania

Recenzje (przeglądy) dokumentów

- Dokumenty opisujące elementy projektu lub dokumentacja użytkowa powinna podlegać recenzjom (przeglądom).
- W zależności od charakteru dokumentu, recenzenci mogą rekrutować się z wewnątrz lub z zewnątrz zespołu projektowego.
- Zadaniem recenzenta jest znalezienie możliwie największej liczby defektów.
- Recenzent przedstawia wynik w postaci dokumentu, gdzie zapisuje:
  - Identyfikator PK
  - Lokalizację defektu (PK niższego poziomu, nr strony, nr wiersza,...)
  - Opis defektu
  - Możliwy sposób usunięcia defektu (rozwiązania problemu).
- Po recenzji (recenzjach) następuje spotkanie wszystkich zainteresowanych stron, gdzie po dyskusji podejmuje się decyzję o zmianach w dokumentach lub o ich zatwierdzeniu.
- Dokumenty podlegają określaniu statusu na podanych wcześniej zasadach.

Wydanie (opublikowanie)

release

Każda pozycja konfiguracji (zwykle cały projekt, ale niekoniecznie), która jest zakończona i oficjalnie przekazana na zewnątrz (zwykle na zewnątrz firmy wytwórcy oprogramowania), jest określana jako wydanie (*release*).

- Wydania muszą być odpowiednio opisane, udokumentowane i zaaprobowane na poziomie kierownictwa projektu, kierownictwa firmy oraz klienta.
- Pozycje konfiguracji będące wydaniami muszą być wyraźnie oznaczone i "zamrożone" w bibliotece/repozytorium konfiguracji. Identyfikacja i rejestracja wydań powinna być zgodna z konwencją identyfikacji. Np. konfiguracja oznaczona SME 1.0 może nie być wydaniem, jest nim np. konfiguracja SME 1.4.
- Na poziomie firmy wytwórcy oprogramowania wszystkie składniki wydania (włączając dokumenty analityczne, plany, kody źródłowe, dokumenty wewnętrzne, dokumentacja testowa, itd.) muszą być przechowywane jako składniki wydania. Zwykle tylko pewna cześć z tych pozycji konfiguracji jest przekazana na zewnątrz (np. wynikowy kod programu plus dokumentacja). Pozycje konfiguracji przekazane na zewnątrz jako składniki wydania powinny być odnotowane w bibliotece/repozytorium konfiguracji.

Plan Zarządzania Konfiguracją Oprogramowania

- Wszystkie aktywności związane z zarządzaniem konfiguracją oprogramowania dla danego projektu lub jego fazy powinny być przewidziane w Planie Zarządzania Konfiguracją Oprogramowania (PZKO).
- Nowe sekcje PZKO muszą pojawiać się w miarę przystępowania do kolejnych faz rozwoju oprogramowania. Każda sekcja powinna dokumentować wszystkie aktywności związane z konfiguracja oprogramowania, w szczególności:
  - organizację zarządzania konfiguracjami;
  - procedury do identyfikacji konfiguracji;
  - procedury do kontroli zmian;
  - procedury do rejestrowania statusu konfiguracji;
  - narzędzia, techniki i metody dla zarządzania konfiguracjami;
  - procedury do kontrolowania dostawców;
  - procedury do gromadzenia i zachowywania zapisów dotyczących konfiguracji.
- Procedury zarządzania konfiguracją powinny być ustanowione przed rozpoczęciem produkcji kodu i dokumentacji.
- W miarę możności PZKO powinien przewidywać pozycje konfiguracji (będące rezultatem danej fazy rozwoju) oraz ustalać ich identyfikacje i odpowiedzialności.

Zawartość PZKO (Wg normy ANSI/IEEE Std 828-1990) (1)

Informacje organizacyjne

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

Zasadnicza zawartość dokumentu

1. Wprowadzenie
- 1.1 Cel
- 1.2 Zakres
- 1.3 Słownik terminów, akronimów i skrótów
- 1.4 Odsyłacze

2. Zarządzanie
- 2.1 Organizacja
- 2.2 Odpowiedzialności w zakresie ZKO
- 2.3 Zarządzanie interfejsami zewnętrznymi
- 2.4 Implementacja PZKO
- 2.5 Stosowane strategie, zalecenia i procedury

3. Identyfikacja konfiguracji
- 3.1 Konwencje
- 3.2 Produkty bazowe

..... c.d. na następnej stronie.....

Zawartość PZKO (2)

Zasadnicza zawartość dokumentu, c.d.

..... c.d. z poprzedniej strony

4. Kontrola konfiguracji

- 4.1 Kontrola kodu i dokumentacji
- 4.2 Kontrola mediów
- 4.3 Kontrola zmian
  - *4.3.1 Poziomy autoryzacji zmian*
  - *4.3.2 Procedury zmian*
  - *4.3.3 Ciało (rada, komisja) przeglądowa*
  - *4.3.4 Kontrola interfejsów*
  - *4.3.5 Procedury zmian oprogramowania obcego*

5. Rejestracja statusu konfiguracji

6. Narzędzia, techniki i metody dla ZKO

7. Kontrola dostawców

8. Gromadzenie i przechowywanie zapisów

Numeracja punktów nie powinna być zmieniana. Jeżeli pewien punkt nie ma treści, powinna tam znajdować się informacja „Nie dotyczy”.
Informacje nie mieszczące się w tym spisie treści powinny być zawarte w dodatkach.

Omówienie zawartości PZKO (1)

1.1 Cel

Krótko omawia cel PZKO (do czego i dla kogo jest przeznaczony).

1.2 Zakres

Określa z grubsza pozycje konfiguracji będące przedmiotem zarządzania, aktywności związane z konfiguracją, organizacje (zespoły projektowe) do których plan ma zastosowanie, oraz fazę cyklu życiowego, której plan dotyczy.

1.4 Odsyłacze

Zawiera listę wszystkich związanych dokumentów, identyfikowanych przez tytuł, autorów, daty, numery, sygnatury, itp.

2. Zarządzanie

Sekcja ta opisuje organizację zarządzania konfiguracją oraz związane z tym odpowiedzialności oraz role.

2.1. Organizacja

Podsekcja ta identyfikuje role organizacyjne, które mogą wpłynąć na funkcje ZKO, np. menedżerów projektów, programistów, personel zapewnienia jakości, ciała przeglądowe, rewizyjne i akceptacyjne. Opisuje także związki pomiędzy rolami oraz interfejsy z organizacjami klienta/użytkownika.

Omówienie zawartości PZKO (2)

2.2 Odpowiedzialności w zakresie ZKO

Określa funkcje w zakresie w zakresie ZKO, za które są odpowiedzialne poszczególne role organizacyjne (np. identyfikacje PK, przechowywanie, kontrola zmian, określanie statusu. Ustala także odpowiedzialności w zakresie przeglądów, audytów i zatwierdzeń, włączając w to rolę użytkowników w tych czynnościach.

2.3 Zarządzanie interfejsami zewnętrznymi

Określa procedury zarządzania interfejsami do zewnętrznego sprzętu i oprogram., organizacje zewnętrzne udostępniające ten sprzęt i oprogram., punkty kontaktowe z tymi organizacjami, oraz grupy odpowiedzialne za poszczególne interfejsy.

2.4 Implementacja PZKO

Ustanawia kluczowe elementy w zakresie wdrożenia PZKO, m.in. gotowość systemu ZKO do użycia, ciała przeglądu oprogram., produkty bazowe, wydania produktu, itd.

2.5 Stosowane strategie, zalecenia i procedury

Identyfikuje wszystkie mające zastosowanie strategie konfiguracji oprogram., zalecenia i procedury będące składową PZKO, oraz określa ich interpretację.

Omówienie zawartości PZKO (3)

3.1 Konwencje

Określa konwencję w zakresie nazywania PK etykietowania PK.

3.2 Produkty bazowe

Dla każdego produktu bazowego sekcja ta określa:

• identyfikator;

• zawartość, np. oprogramowanie, narzędzie, oprogramowanie testowe, raporty o niezgodności, zgłoszenie problemu, itp. dokumenty;

• interfejsy do produktu bazowego;

• zdarzenia związane z przeglądami i akceptacją;

• uczestnictwo producentów i użytkowników podczas określania produktu bazowego.

Opis każdego produktu bazowego powinien wyróżniać oprogramowanie, które jest ponownie użyte lub zakupione, definiować środowisko sprzętowe, oraz określać PK będące składnikami produktu bazowego.

Omówienie zawartości PZKO (4)

4.1 Kontrola kodu i dokumentacji

Określa procedury dla zarządzania biblioteką kodów i dokumentacji: biblioteką rozwoju oprogramowania, biblioteką główną (produktów bazowych, wydań) oraz archiwum.

4.2 Kontrola mediów

Określa na jakich nośnikach i gdzie fizycznie będą przechowywane poszczególne PK (dysk serwera, taśma magnetyczna, dyskietki, dyski optyczne, szafy z dokumentami papierowymi, itd.). Określa także konwencję etykietowania poszczególnych mediów (np. taśm, dyskietek, dysków optycznych, określa sposób i miejsce ich przechowywania (szafy pancerne, sejfy bankowe, itd.) oraz zasady recyklingu mediów (kiedy dane medium może być ponownie użyte).

4.3 Kontrola zmian; *4.3.1 Poziomy autoryzacji zmian*

Określa poziom autorytetu, który może zarządzić zmianę do danego produktu bazowego (bibliotekarz, kierownik projektu, itd.)

4.3 Kontrola zmian; *4.3.2 Procedury zmian*

Określa w jaki sposób zmiany będą nanoszone (kto, kiedy, jak).

Omówienie zawartości PZKO (5)

4.3 Kontrola zmian; *4.3.3 Ciało (rada, komisja) przeglądowa*

Określa członków ciała przeglądowego, poziomy autorytetów, delegacje uprawnień do niższych poziomów.

5. Rejestracja statusu konfiguracji

Definiuje w jaki sposób będzie zbierana, przechowywana i przetwarzana informacja o PK, określa okresowe raporty dotyczące statusu PK.

6. Narzędzia, techniki i metody dla ZKO

Określa narzędzia (np. repozytoria oprogramowania, takie jak CVS lub ClearCase) oraz metody (np. metodyki), które będą użyte do ZKO.

7. Kontrola dostawców

Określa zadania z zakresie ZKO, które będą wykonane przez zewnętrznych dostawców.

8. Gromadzenie i przechowywanie zapisów

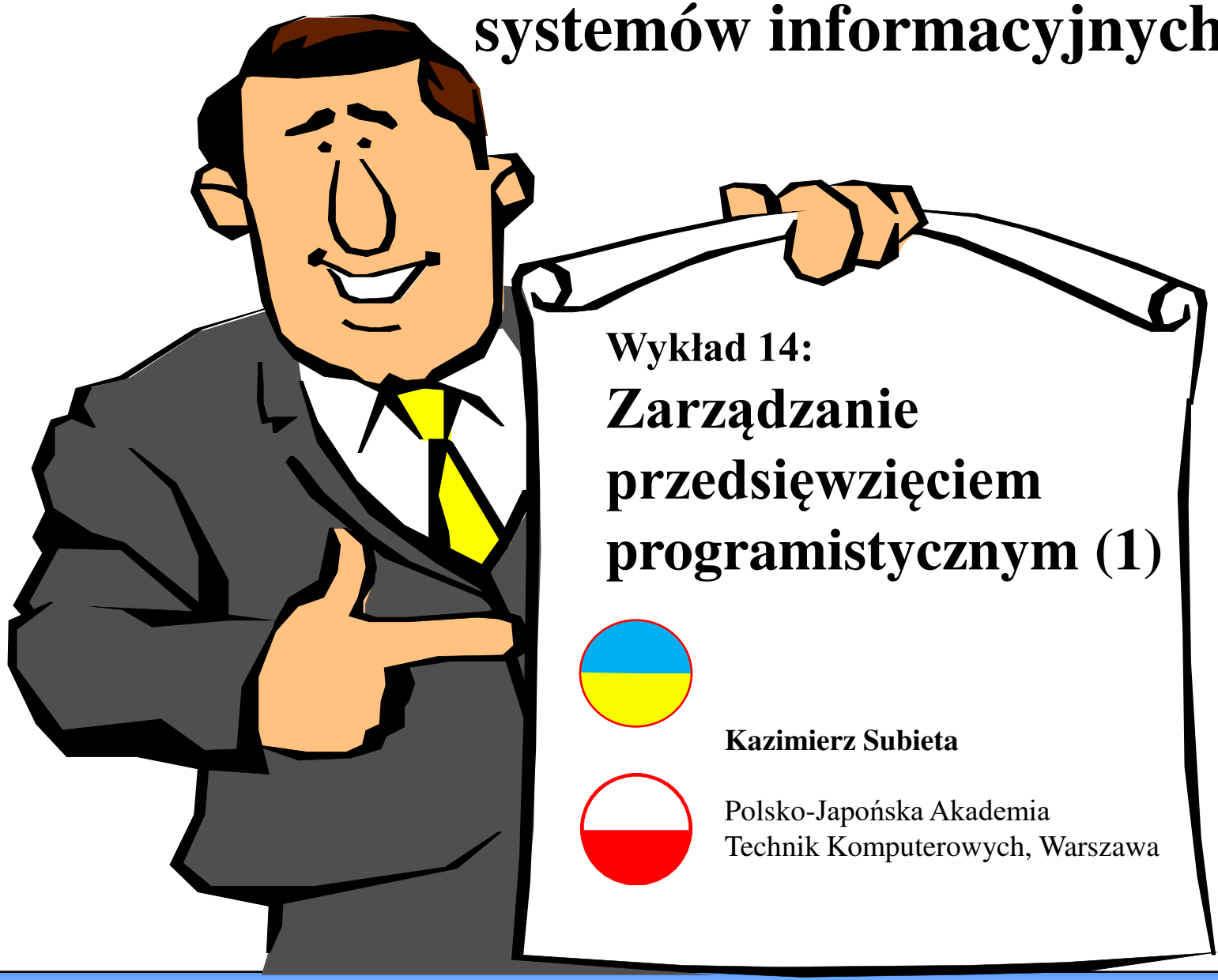
Określa które pozycje konfiguracji będą przechowywane, w jaki sposób, i jak długo.

Przykładowe pytania egzaminacyjne

1. Scharakteryzuj i porównaj model kaskadowy i model spiralny. Oceń wady i zalety obu tych modeli.
2. Jakie znasz algorytmiczne modele kosztów? Określ jaki rodzaj kosztów one liczą.
3. Krótko scharakteryzuj i porównaj wymagania funkcjonalne i niefunkcjonalne.
4. Jakie kryterium określa podział projektu na części, zaś oprogramowania na moduły? Krótko je omów.
5. Po co są transakcje w bazach danych?
6. Co rozumiesz pod pojęciem konserwacja oprogramowania?

Egzamin: 8 lipca (sobota), 15:00 – 16:30, budynek A2020, sala A/1

Budowa i integracja systemów informacyjnych



Wykład 14:
Zarządzanie przedsięwzięciem programistycznym (1)

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Plan wykładu

- Zadania kierownictwa przedsięwzięcia
- Czynniki psychologiczne w inżynierii oprogramowania
- Lojalność grupowa
- Ergonomia pracy
- Struktura zarządzania firmą programistyczną
- Poziomy rozwoju firmy programistycznej
- Dokumentacja, zarządzanie wersjami
- Miary produktywności
- Harmonogramowanie przedsięwzięć
- Ekonomiczne aspekty działalności firmy

Zadania kierownictwa przedsięwzięcia

Niezbędnym warunkiem sukcesu jest właściwe zarządzanie przedsięwzięciem.

Podstawowe zadania kierownictwa przedsięwzięcia programistycznego:

- Opracowanie propozycji dotyczących sposobu prowadzenia przedsięwzięcia
- Kosztorysowanie przedsięwzięcia
- Planowanie i harmonogramowanie przedsięwzięcia
- Monitorowanie i kontrolowanie realizacji przedsięwzięcia
- Dobór i ocena personelu
- Opracowanie i prezentowanie sprawozdań dla kierownictwa wyższego szczebla

Sposoby zarządzania przedsięwzięciem programistycznym nie różnią się od zarządzania innymi przedsięwzięciami, chociaż posiadają swoją specyfikę, np. nieprzejrzystość procesu budowy oprogramowania.

Czynniki psychologiczne w inżynierii oprogramowania

Czynniki te wynikają z faktu, że oprogramowanie jest <u>używane</u> i <u>tworzone</u> przez ludzi.

Użytkowanie - implikuje zasady tworzenia interfejsu użytkownika i dokumentacji użytkowej.
Tworzenie - zagadnienia psychologiczne odgrywające rolę w tworzeniu oprogramowania.

Cechy ludzkiej inteligencji:

- Umiejętność całościowego (syntetycznego) spojrzenia na problem.
- Świadomość celów, środków i (niekiedy odległych) konsekwencji działania.
- Posługiwanie się wiedzą płynącą z doświadczenia, a więc stosowania nieścisłych zasad wnioskowania na bazie wcześniejszych doświadczeń.
- Wynalazczość, adaptowalność: odkrywanie nowych efektywnych rozwiązań w sytuacji całkowitego braku wiedzy lub wzorców.
- Zespołowość: umiejętność efektywnego działania w dużych zespołach, z podziałem ról i specjalizacji.

Sztuczna inteligencja?

- Tak są określane **algorytmy** wzbudzające powszechne zdumienie analogiami z ludzkim myśleniem i działaniem.
  - W istocie jest to **stereotyp, antropomorfizm** wynikający m.in. z dziennikarskich tendencji do tworzenia sensacji
- Ludzkość nie jest na etapie wyjaśnienia czym jest inteligencja
  - Istotne badania w tym zakresie nie istnieją.
  - Czy może być „inteligentny" twór, który nie odwzorowuje i nie rozumie otaczającego go świata?
  - Nie rozumie celu, konsekwencji działania w swoim dalszym otoczeniu?
- „Sztuczną inteligencją" nazywa się zawansowane technicznie rozwiązania, które nie tyle naśladują procesy i zachowania ludzkiego mózgu, ale są od nich lepsze, szybsze, dokładniejsze, odporne na błędy, itp.
  - Tendencje do semantycznej analogii i przesady są tak stare jak komputery: kalkulator jako „mózg elektronowy".
  - „W ciągu dwudziestu lat maszyny będą w stanie wykonać każdą pracę, jaką wykonują ludzie." (1956 r., Herbert Simon, amerykański informatyk i futurolog, laureat Nagrody Turinga)

Osobowość twórców oprogramowania

Istnieją ogromne różnice w predyspozycjach osób dotyczące ich efektywności w produkcji oprogramowania. Wydajność może się różnić o rząd lub więcej.

Testy osobowości:
metody określenia, czy dana osoba posiada cechy przydatne na danym stanowisku.

Stosowanie testów osobowości wiąże się z następującymi trudnościami:

- Osobowość ludzka ma charakter dynamiczny (zmienia się). Wieloletnia praktyka zawodowa nie pozostaje bez wpływu na osobowość. Część cech osobowości może być nabyta i nie da się odkryć wstępnymi testami.
- Różne zadania mogą wymagać różnych cech osobowości. Inne powinien posiadać analityk (kontakt z klientem), inne zaś programista lub osoba testująca oprogramowanie. Ponadto, metody inżynierii oprogramowania ulegają zmianie, co pociąga za sobą inny stosunek pożądanych cech osobowości do aktualnych zadań.
- Osoby poddane testom będą starały się raczej odgadnąć pożądaną przez testujących odpowiedź niż odpowiadać zgodnie ze stanem faktycznym. Test nie będzie więc odzwierciedlał cech osobowości osoby, lecz raczej to, jak ta osoba wyobraża sobie cele i kryteria testowania oraz cechy pożądane przez pracodawcę.

Cechy dobrego inżyniera oprogramowania

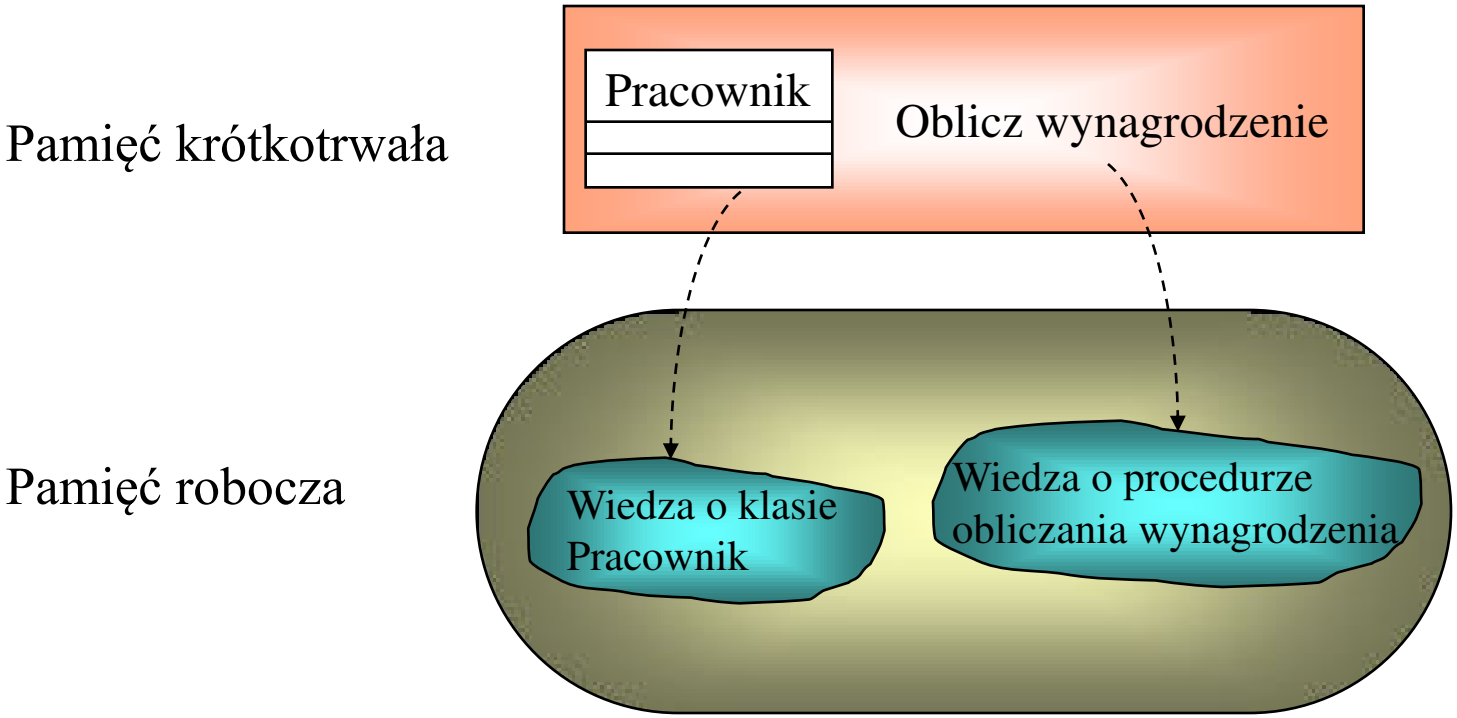
- **Umiejętność pracy w stresie**. W pracy często zdarzają się okresy wymagające szybkiego wykonania złożonych zadań. Dla większości osób niewielki stres działa mobilizująco. Po przekroczeniu jednak pewnego progu następuje spadek możliwości danej osoby. Próg ten jest różny dla różnych osób.

- **Zdolności adaptacyjne**. Informatyka jest jedną z najszybciej zmieniających się dziedzin. Ocenia się, że jeden rok przynosi w informatyce zmiany, które w innych dziedzinach zajmują 5-10 lat. Oznacza to konieczność stałego kształcenia dla wszystkich inżynierów oprogramowania - stałe poznawanie nowych narzędzi, sprzętu, oprogramowania, technologii, metod, sposobów pracy.

- Nie wszyscy to tempo wytrzymują. Uśpienie, zajmowanie się jednym problemem w jednym środowisku przez lata jest w informatyce bardzo groźne.

Rola skojarzeń w pracy nad złożonymi problemami

Możliwość budowania skojarzeń jest cechą ludzkiego umysłu i znacznie wzmacnia zarówno objętość zapamiętywanej informacji jak i szybkość dostępu.



Umysł ludzki sprawnie kojarzy wiedze o klasie pracownik i wiedze o procedurze obliczającej wynagrodzenie pracownika. Posługiwanie się rysunkami i czytelnymi nazwami zdecydowanie ułatwia tworzenie tego typu skojarzeń.

Rodzaje wiedzy

- **Wiedza składniowa**. Polega na mechanicznym zapamiętaniu pewnych faktów, bez ich istotnego przetworzenia. Jest słabo zintegrowana z wcześniej zdobytą wiedzą.
  - Np. do takiej wiedzy zaliczamy reguły składniowe danego języka programowania.
- **Wiedza semantyczna.** Fakty są zapamiętane nie w postaci ich formy, lecz w postaci znaczenia.
  - Np. znajomość zasady instrukcji *while*, znajomość pojęcia klasy i dziedziczenia, itd.
- **Wiedza pragmatyczna**: kiedy, jak i po co użyć tej wiedzy dla osiągnięcia pożądanego celu praktycznego.

Istnienie tych trzech rodzajów wiedzy może mieć wpływ na politykę kadrową. Np. pracownik rozumiejący zasady obiektowości (wiedza semantyczna) może lepiej sobie poradzić niż pracownik dobrze znający składnię C++ (wiedza składniowa). Najbardziej liczy się jednak doświadczenie (wiedza pragmatyczna).

Firmy przywiązują zbyt wielką wagę do wiedzy składniowej, np. znajomości konkretnych języków i systemów. W istocie, ta wiedza może być stosunkowo szybko nabyta (kilka tygodni przeciętnie na opanowanie nowego języka). Natomiast wiedza pragmatyczna może być przedmiotem lat pracy, studiów i doświadczeń.

Nastawienie osób do pracy w zespole

Czynniki psychologiczne mają zasadniczy wpływ na efektywność pracy zespołu. Wyróżnia się następujące typy psychologiczne:

- **1. Zorientowani na zadania** (*task-oriented*). Osoby samowystarczalne, zdolne, zamknięte, agresywne, lubiące współzawodnictwo, niezależne.

- **2. Zorientowani na siebie** (*self-oriented*). Osoby niezgodne, dogmatyczne, agresywne, zamknięte, lubiące współzawodnictwo, zazdrosne.

- **3. Zorientowani na interakcję** (*interaction-oriented*). Osoby nieagresywne, o niewielkiej potrzebie autonomii i indywidualnych osiągnięć, pomocne, przyjazne.

Osoby typu 1 są efektywne, o ile pracują w pojedynkę. Zespół złożony z takich osób może być jednak nieefektywny. Lepsze wyniki dają zespoły złożone z typów 3. Typ 1 i 2 może być także efektywny w zespole, o ile jest odpowiednio motywowany przez kierownictwo. Typy 3 są konieczne w fazie wstępnej wymagającej intensywnej interakcji z klientem.

Lojalność grupowa

Terminem tym określa się silny, osobisty związek pomiędzy poszczególnymi członkami zespołu, grupą i wynikami pracy grupy. Niekorzystne efekty:

- **Trudność zmiany lidera**. Silnie związana grupa nie akceptuje nowego lidera narzuconego z zewnątrz. Często jednak formalny lider źle kieruje pracą.
- **Myślenie grupowe** (*groupthink*). Brak krytycyzmu w stosunku do efektów pracy grupy, nie rozważanie jakichkolwiek pomysłów i rozwiązań nie pochodzących z wnętrza grupy, wzajemne podtrzymywanie się w poglądach, często niesłusznych lub tendencyjnych. Rezultatem jest znaczny spadek jakości wyników pracy.

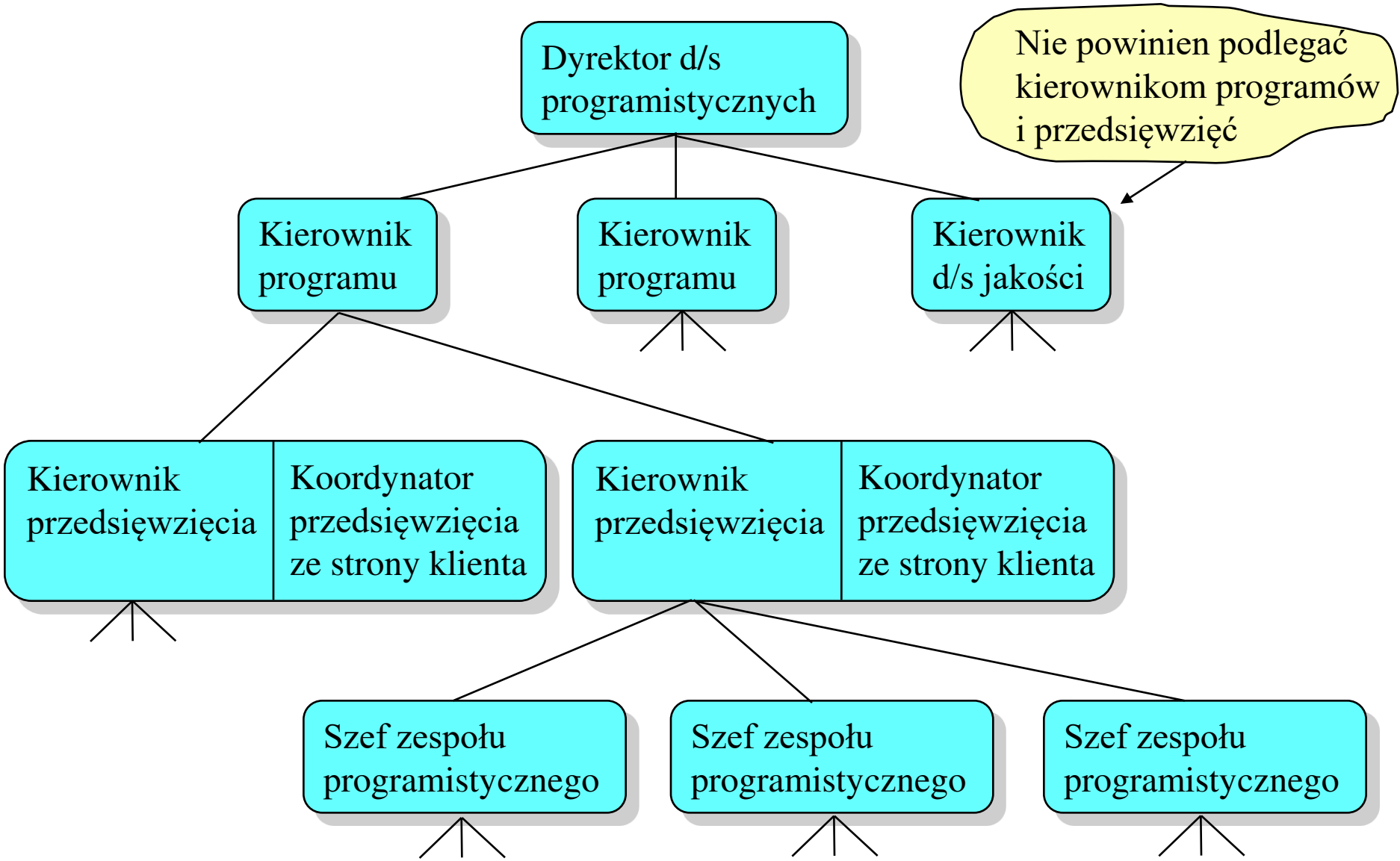
Walka z myśleniem grupowym:

- **Sesje krytyki**, podczas których dozwolona jest jedynie krytyka przyjętych rozwiązań, natomiast zabroniona jest jakakolwiek obrona osiągnięć grupy.
- **Włączanie do zespołu krytycznych osobowości** - osób o szczególnych zdolnościach do wyszukiwania błędów i kwestionowania przyjętych rozwiązań (tzw. “czepialskich”). Osoby te są zwykle nielubiane.

Ergonomia pracy

- Zamiast dużej hali, lepsze wyniki daje umieszczenie dwóch-trzech stanowisk pracy w wielu mniejszych pomieszczeniach.
- Personalizacja stanowiska pracy.
- Pokój zebrań dla organizowania formalnych spotkań pracowników.
- Miejsce dla spotkań nieformalnych (np. omówienie spraw przy kawie).
- Poczucie pracy na nowoczesnym sprzęcie. Wydajność i chęć ludzi do pracy gwałtownie spada, jeżeli odczuwają oni, że pracują na przestarzałym sprzęcie - nawet wtedy, gdy wymiana sprzętu jest merytorycznie nieuzasadniona.
- Komfort psychiczny, właściwa atmosfera w pracy, eliminacja napięć i zadrażnień, nie dopuszczanie do rozmycia odpowiedzialności, sprawiedliwa ocena wyników pracy poszczególnych członków zespołu, równomierny rozkład zadań.

Struktura zarządzania firmą programistyczną



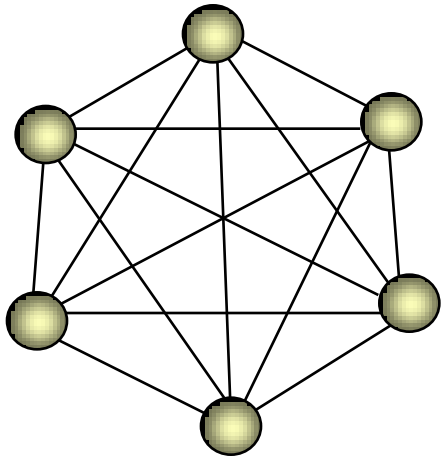
Funkcje osób pracujących nad oprogramowaniem

- **Kierownik programu/przedsięwzięcia**
- **Analityk** - osoba bezpośrednio kontaktująca się z klientem, której celem jest określenie wymagań i budowa modelu systemu
- **Projektant** - osoba odpowiedzialna za realizację oprogramowania. Może posiadać bardziej wyspecjalizowane funkcje:
  - Projektant interfejsu użytkownika
  - Projektant bazy danych
- **Programista** - osoba implementująca oprogramowanie
- **Osoba wykonująca testy**
- **Osoba odpowiedzialna za konserwację oprogramowania**
- **Ekspert metodyczny** - osoba szczególnie dobrze znająca stosowaną metodykę
- **Ekspert techniczny** - osoba szczególnie dobrze znająca sprzęt i narzędzia

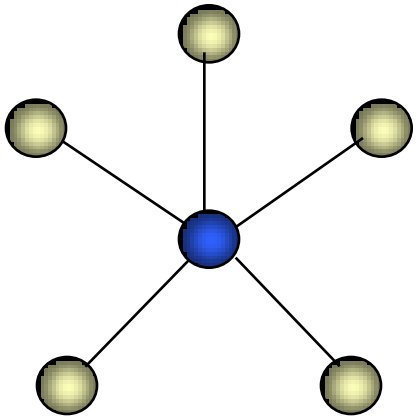
Model analityk/projektant + programista: funkcje analizy i projektu w jednych rękach, funkcje programisty dość niskiego poziomu. W warunkach polskich model nie zdaje egzaminu.
Model analityk + projektant/programista: Model bardziej realistyczny.

Organizacja zespołu programistycznego

Struktura sieciowa



Struktura gwiaździsta



Zalety:

- Dzięki ścisłej współpracy członkowie zespołu wzajemnie kontrolują swoją pracę. Szybko osiągane są standardy jakości.
- Umożliwia realizację idei wspólnego programowania
- Ponieważ praca członków zespołu jest znana dla innych członków, łatwo mogą oni przejąć obowiązki pracownika, który opuścił zespół.

Jest przydatna wtedy, gdy w skład zespołu wchodzi wielu niedoświadczonych pracowników. Szef kontroluje i koordynuje pracę.
Wielkość zespołu może być znacznie większa niż w strukturze sieciowej.

Duże problemy w momencie odejścia szefa zespołu.

Struktura sieciowa nie może liczyć więcej niż 8 osób.

Zapewnianie jakości

Nie powinno być utożsamiane z testowaniem oprogramowania i pomiarami dokonywanymi na gotowym oprogramowaniu.

Podejście procesowe (dyscyplina wytwarzania)**:**
Jakość produktu jest związana z jakością procesu, który go wytwarza. Na tym założeniu oparte są standardy jakości, takie jak ISO 9000, Prince-2, itd.

- **Kryteria jakości:** zgodność z wymaganiami użytkownika, efektywność, łatwość konserwacji, ergonomiczność
- Na jakość oprogramowania wpływają działania podejmowane we wszystkich fazach jego życia.
- Istotne jest określenie kryteriów jakości i ich priorytetu.
- Kryteria te powinny być zawarte w dokumencie zwanym

  planem jakości.

Poziomy rozwoju firmy programistycznej

Pięć poziomów rozwoju organizacji z punktu widzenia dojrzałości procesu:

- **Początkowy**. Na tym poziomie nie istnieją żadne standardy procesu. Decyzje są podejmowane *ad hoc*. Ten poziom mogą mieć również firmy o dobrym zaawansowaniu technicznym.
- **Powtarzalny**. Poszczególne przedsięwzięcia wykonywane sa w podobny sposób. W firmie istnieje co do tego zgoda, można więc mówić o pewnym standardzie firmy, które są jednak standardami *de facto*. Standardy te nie są udokumentowane. Nie istnieją ścisłe procedury kontroli.
- **Zarządzany**. Standardy postępowania są dobrze zdefiniowane i sformalizowane. Istnieje ścisła kontrola przestrzegania standardów. Są osoby odpowiedzialne za opracowanie i uaktualnianie standardów.
- **Mierzony**. Proces nie tylko podlega kontroli na zgodność ze standardami, ale jest mierzony w sposób ilościowy. Mierzona jest np. wydajność. Wyniki są wykorzystywane do poprawy sposobów realizacji przyszłych przedsięwzięć.
- **Optymalizowany**. Standardy są w ciągły sposób uaktualniane tak, aby uwzględnić doświadczenia w przyszłych przedsięwzięciach. Standardy zawierają elementy pozwalające na dostrojenie procesu do aktualnych potrzeb.

Dokumentacja procesu wytwarzania

Podlega sformalizowanemu zarządzaniu konfiguracją oprogramowania.
W trakcie trwania przedsięwzięcia powstają następujące dokumenty:
- Dokumentacja procesu produkcji oprogramowania.
- Dokumentacja techniczna opisująca wytworzony produkt.

Dokumentacja procesu:

- **Plany, szacunki, harmonogramy** - dokumenty tworzone przez kierownictwo przedsięwzięcia Odbiorcami ich są przełożeni wyższego szczebla. Zaakceptowane dokumenty tego typu pełnia rolę poleceń dla wykonawców.
- **Raporty** - Dokumenty przygotowywane przez kierowników dla przełożonych. Opisują przebieg i rezultaty prac.
- **Standardy** - dokumenty opisujące pożądany sposób realizacji
- **Dokumenty robocze** - rozmaite dokumenty zawierające propozycje rozwiązań. Twórcami są członkowie zespołu. Zaakceptowane mogą stać się standardami.
- **Komunikaty** - rozmaite, z reguły krótkie dokumenty służące do wymiany informacji pomiędzy członkami zespołu.

Dokumentacja techniczna

Dokumentacja techniczna przed oddaniem oprogramowania do eksploatacji powinna być poddana weryfikacji celem wyeliminowania błędów i nieścisłości.

Istotne jest wypracowanie w firmie standardów dokumentacji technicznej:

- Procesów wytwarzania dokumentacji: tworzenia wstępnej wersji dokumentów, wygładzania, drukowania, powielania, oprawiania, wprowadzania zmian w istniejących dokumentach. Konieczne jest ścisłe określenie odpowiedzialnych za to osób.
- Treści i formy dokumentów: strona tytułowa, spis treści, budowa rozdziałów, podrozdziałów i sekcji, indeks, słownik.
- Sposobu dostępu do dokumentacji: niezbędne jest stworzenie rodzaju biblioteki dokumentów technicznych, z zapewnieniem sprawnego dostępu do dowolnego dokumentu.

Zarządzanie wersjami

Podlega sformalizowanemu zarządzaniu konfiguracją oprogramowania.

- Każda modyfikacja oznacza powstanie nowej wersji systemu, mniej lub bardziej różnej od wersji poprzedniej.
- Zróżnicowanie potrzeb użytkowników. Mogą być wersje będące kombinacją modułów oprogramowania.
- Istnienie wielu platform sprzętowych i systemów operacyjnych.

System zarządzania konfiguracją oprogramowania powinien zawierać:

- Informację o wszystkich wykonanych i oddanych do eksploatacji wersjach
- Informację o klientach, którzy nabyli daną wersję
- Wymagania sprzętowe i programowe poszczególnych wersji
- Informację o składowych (klasach, encjach, modułach) wchodzących w skład danej wersji
- Informację o żądaniach zmian w stosunku do danej wersji
- Informację o błędach wykrytych w poszczególnych wersjach.

Miary produktywności

Konieczna jest ocena poszczególnych pracowników ze względu na:
- Konieczność odpowiedniego motywowania najbardziej wydajnych osób
- Możliwość wykorzystania zebranych danych do szacowania przyszłych zadań

Tradycyjne miary produktywności:

- Liczba linii uruchomionego i przetestowanego kodu źródłowego (bez komentarzy) napisanych np. w ciągu miesiąca.
- Liczba instrukcji kodu wynikowego wyprodukowanego w pewnym okresie czasu
- Liczba stron dokumentacji napisanej w pewnym okresie czasu
- Liczba przykładów testowych opracowanych w pewnym okresie czasu

Stosowanie nowoczesnych narzędzi może utrudnić lub uniemożliwić posługiwanie się tymi miarami:
(1) Języki wysokiego poziomu => krótki kod, duża wydajność;
(2) Gotowe biblioteki, generatory => duża liczba instrukcji w krótkim czasie.

Miary te mogą także doprowadzić do tendencyjnego wypaczenia procesów produkcji oprogramowania. Miary te są bardzo trudne do zastosowania dla analityków i projektantów („astrologia”).

Harmonogramowanie przedsięwzięć

- Ustalenie kalendarza prac
  - daty rozpoczęcia przedsięwzięcia
  - dni roboczych i wolnych w przewidywanym okresie realizacji przedsięwzięcia
  - czasu pracy w poszczególnych dniach
- Podział przedsięwzięcia na poszczególne zadania
- Określenie parametrów zadań
  - czasu wykonania
  - najwcześniejszy możliwy termin rozpoczęcia
  - pożądany czas zakończenia
  - innych ograniczeń, np. zadań których zakończenie jest niezbędne do rozpoczęcia nowych zadań.
- Określenie zasobów niezbędnych do realizacji poszczególnych zadań
- Ustalenie dostępności zasobów
- Ustaleniu kolejności i czasów wykonania poszczególnych zadań

Przykładowy diagram zależności PERT

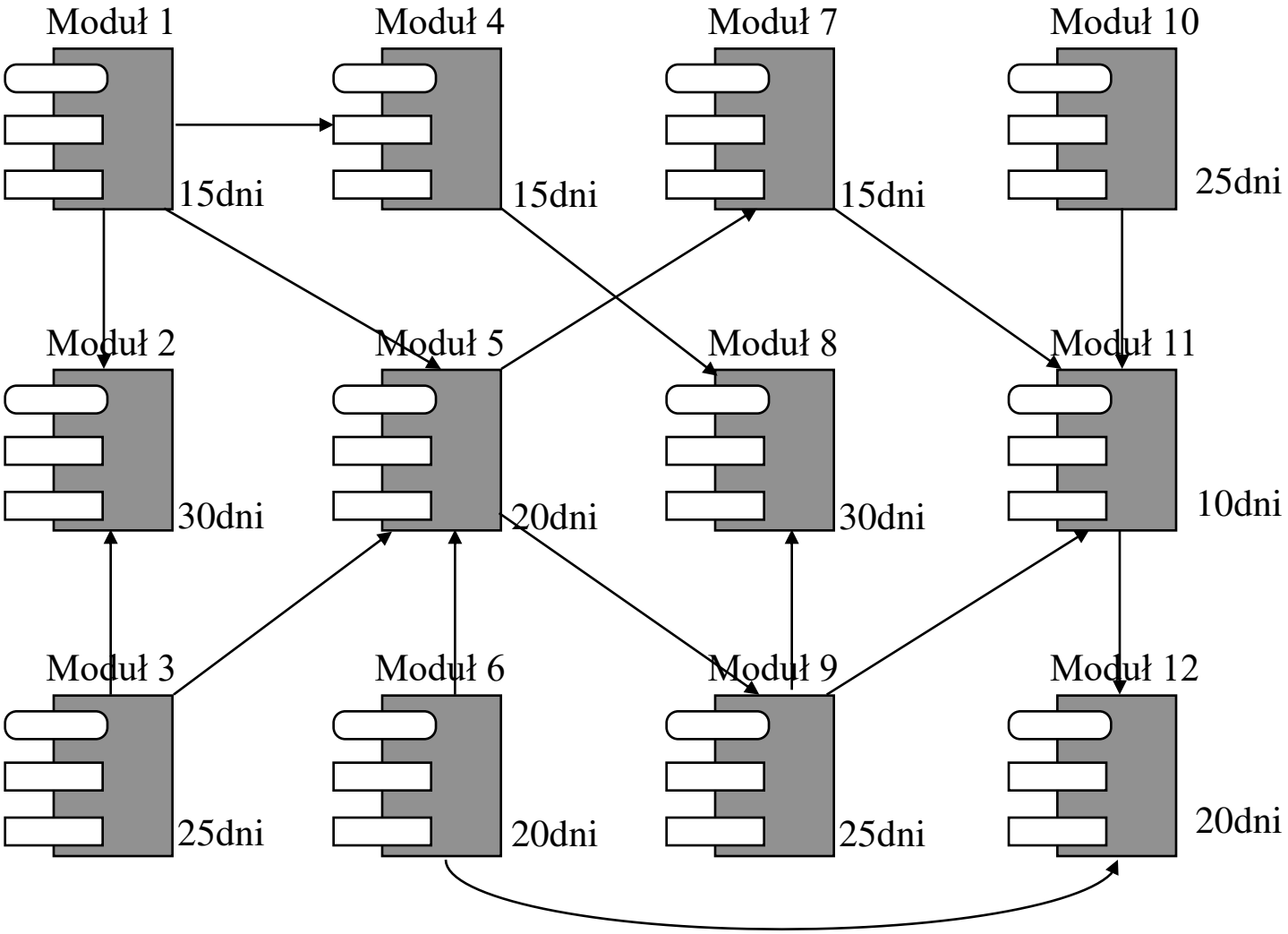
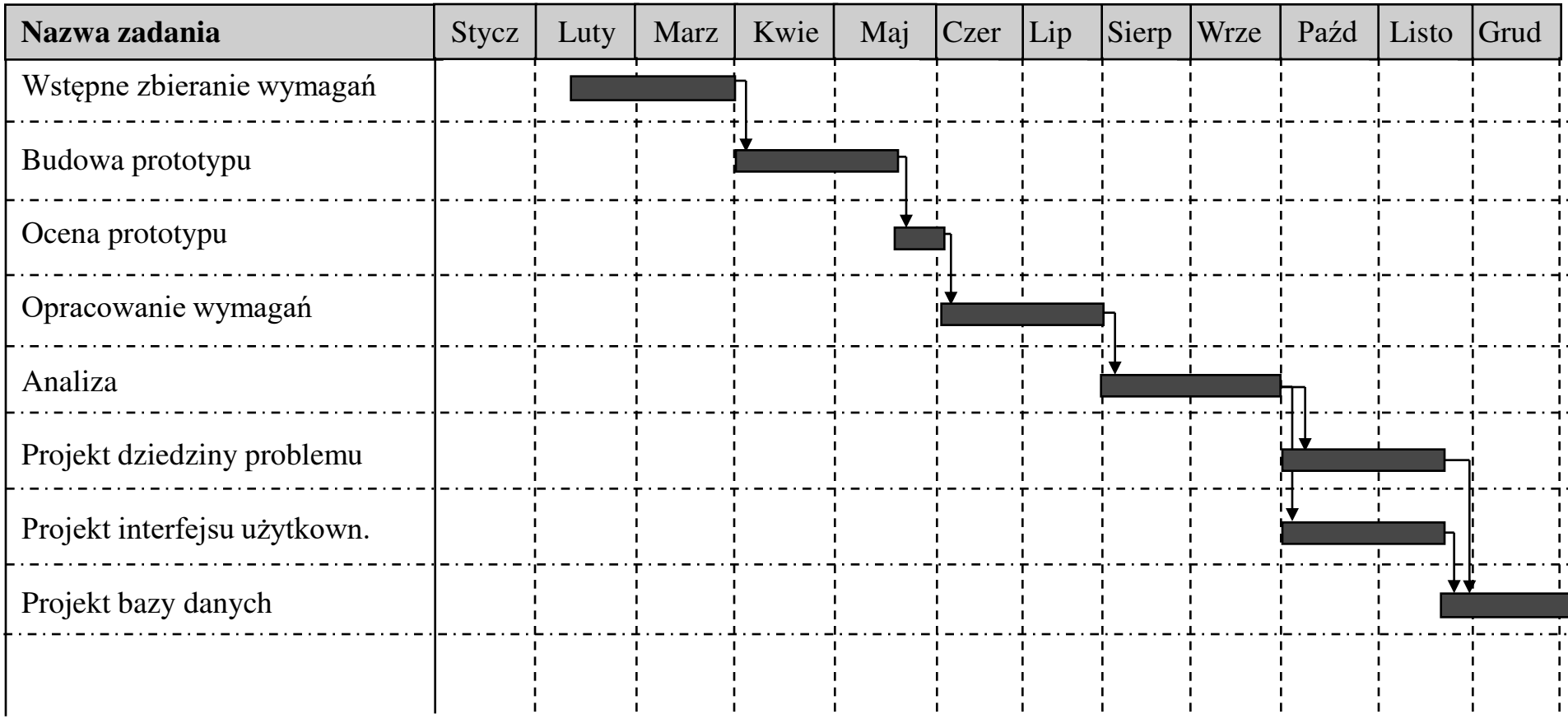


Diagram podlega analizie ścieżki krytycznej (najdłuższa ścieżka = najkrótszy czas)
Diagramy Gantta, diagramy ograniczeń zasobów - inne metody ustalenia harmonogramów.

Diagram Gantta

Ustalenie planu czasowego dla poszczególnych faz i zadań.



Ekonomiczne aspekty działalności firmy

Jakość produktu jest tylko jednym z czynników wpływających na wynik ekonomiczny firmy. Inne aspekty:

- Reklama i promocja produktu
- Renoma producenta
- Rodzaj i zakres gwarancji oraz innych usług dla klientów
- Przyzwyczajenia klientów
- Sposób wyceny rozmaitych wersji produktu.

Na wielkość zysków wpływają także koszty własne poniesione przy produkcji:

- Inwestycje niezbędne wewnątrz firmy
- Koszty reorganizacji firmy
- Koszty szkoleń
- Koszty zakupu narzędzi CASE
- Nakłady na dokładne testowanie oprogramowania

Zwrot nakładów następuje zwykle po pewnym czasie i często nie może być traktowany jako pewny.

Budowa i integracja systemów informacyjnych

Wykład 15:
Zarządzanie przedsięwzięciem programistycznym (2)

Kazimierz Subieta

Polsko-Japońska Akademia
Technik Komputerowych, Warszawa

Aktywności w zarządzaniu projektem

IEEE Standard for Software Project Management Plans
ANSI/IEEE Std 1058.1-1987

Każdy projekt programistyczny podlega:

- **planowaniu**
- **organizowaniu**
- **zarządzaniu personelem**
- **prowadzeniu**
- **monitorowaniu**
- **kontrolowaniu i sterowaniu**

Kierownik projektu ...

- **Tworzy Plan Zarządzania Projektem Programistycznym**
- **Definiuje organizacyjne role i przypisuje do nich personel**
- **Steruje projektem poprzez informowanie personelu o jego roli w ramach planu**
- **Prowadzi projekt poprzez podejmowanie głównych decyzji oraz przez motywowanie personelu do ich właściwego wykonywania**
- **Monitoruje projekt poprzez pomiary postępu prac**
- **Sprawozdaje postęp prac dla inicjalizatorów projektu i zwierzchnictwa**

Danymi wejściowymi dla Planu Zarządzania Projektem Programistycznym są standardy (wewnętrzne lub zewnętrzne) i wymagania użytkownika. Plan dotyczy:
- zarządzania projektem,
- zarządzania konfiguracją,
- weryfikacji i walidacji,
- zapewnienia jakości.

Cel i odpowiedzialność kierownika projektu

Celem stojącym przed kierownikiem projektu jest dostarczenie produktu w wymaganym czasie, w ramach danego budżetu i posiadającego odpowiednią jakość.

Odpowiedzialność kierownika projektu może zmieniać się w zależności od firmy i rodzaju projektu. Zawsze włącza **planowanie** i **prognozowanie**.

Dodatkowe obszary odpowiedzialności:

- **odpowiedzialność interpersonalna**: prowadzenie zespołu, porozumiewanie się z inicjalizatorami, zwierzchnictwem i dostawcami, reprezentowanie projektu na formalnych posiedzeniach i prezentacjach;
- **odpowiedzialność za stan informacji**: monitorowanie wydajności personelu, monitorowanie zgodności postępu prac z planem projektu, informowanie zespołu o bieżących zadaniach, informowanie inicjalizatorów i kierownictwa o stanie projektu.
- **odpowiedzialność decyzyjna**: alokacja zasobów (np. budżetu) zgodnie z planem projektu, korekta alokacji zasobów, negocjacje z inicjalizatorem odnośnie interpretacji warunków kontraktu, negocjacje z zarządem firmy odnośnie zasobów (np. czasu komputerów), negocjacje z zespołem odnośnie zadań, rozstrzyganie wszelkich zakłóceń w toku projektu takich np. jak awaria sprzętu lub problemy z personelem.

Interfejsy projektu

Kierownik projektu musi zidentyfikować i udokumentować **grupy ludzi** (interfejsy projektu), którzy są lub mogą być związani z projektem. Ludzie ci mogą pochodzić z wnętrza lub z zewnętrz firmy.

Rodzaje interfejsów:
- inicjalizatorzy,
- użytkownicy końcowi,
- dostawcy,
- podwykonawcy,
- główny wykonawca (nadwykonawca),
- wytwórcy innych podsystemów danego systemu.

Zdefiniowanie zewnętrznych interfejsów oznacza:
- pojedynczy, nazwany punkt kontaktowy zespołu projektowego z każdą zewnętrzną grupą;
- zredukowanie kanału przesyłania informacji pomiędzy zespołem a każdą zewnętrzną grupą do możliwie minimalnej liczby osób;
- żaden z członków zespołu projektowego nie może kontaktować się z więcej niż siedmioma grupami („reguła siedmiu”).

Planowanie projektu

Niezależnie od rozmiaru projektu, dobre planowanie jest istotne dla jego sukcesu.

Główne aktywności w planowaniu:

- Zdefiniowanie produktów
- Zdefiniowanie aktywności
- Oszacowanie zasobów i czasów wykonania
- Zdefiniowanie sieci aktywności (np. PERT)
- Zdefiniowanie harmonogramu i kosztu ogólnego

Proces planowania powinien być stosowany zarówno do całego projektu, jak i do każdej jego fazy. Każda czynność planowania powinna być powtarzana kilka razy aby uzyskać osiągalny plan. Z reguły konieczne są nawroty do poprzednich czynności.

Dane wejściowe do planowania projektu

- Dokument wymagań użytkownika, dokument wymagań na oprogramowanie, dokument projektu architektury systemu (zgodnie z fazą projektowania)
- Standardy w zakresie oprogramowania dla produktów i procesów wytwarzania
- Dane historyczne dla oszacowania zasobów i czasów trwania
- Dane odnośnie kosztów związanych z dostawami zewnętrznymi
- Dane odnośnie rozważanych czynników ryzyka
- Dane odnośnie środowiska wykonania, takie jak opisy nowych technologii
- Dane odnośnie ograniczeń czasowych, np. data dostarczenia produktu
- Dane odnośnie ograniczeń zasobów, np. dostępność personelu

Zawartość Planu Zarządzania Projektem

- **Definicje produktów**, które będą dostarczone (funkcje, fizyczne charakterystyki, nośnik, język opisu, język programowania).
- **Model procesów** definiujący podejście do cyklu życiowego oprogramowania oraz metod i narzędzi, które będą użyte (aktywności w rozwoju oprogramowania, wejścia i wyjścia każdej aktywności, role osób w każdej czynności, model (wodospadowy, przyrostowy, ewolucyjny, zwinny)).
- **Struktura prac** w postaci hierarchii czynności.
- **Organizacja projektu** definiująca role oraz związki sprawozdawczości.
- **Sieć aktywności** ustalająca kolejność i wzajemne uwarunkowanie czasowe prac.
- **Harmonogram projektu** określający początkowe i końcowe momenty poszczególnych prac.
- **Lista zasobów** wymaganych do realizacji projektu.
- **Estymacja kosztu**.

Dokument Planu Zarządzania Projektem

a - Streszczenie
b - Spis treści
c - Formularz statusu dokumentu
d - Zmiany w dokumencie od ostatniego wydania

1 Wstęp (krótkie omówienie, wyniki projektu, ewolucja planu zarządzania projektem, odsyłacze, definicje i akronimy)

2 Organizacja projektu (model procesów, struktura organizacyjna, granice organizacyjne i interfejsy, odpowiedzialności uczestników projektu)

3 Procesy zarządzania (cele i priorytety zarządzania, założenia, zależności i ograniczenia, zarządzanie ryzykiem, mechanizmy monitorowania i sterowania, plan personalny)

4 Procesy techniczne (metody, narzędzia i techniki, dokumentacja oprogramowania, funkcje wspomagające projekt)

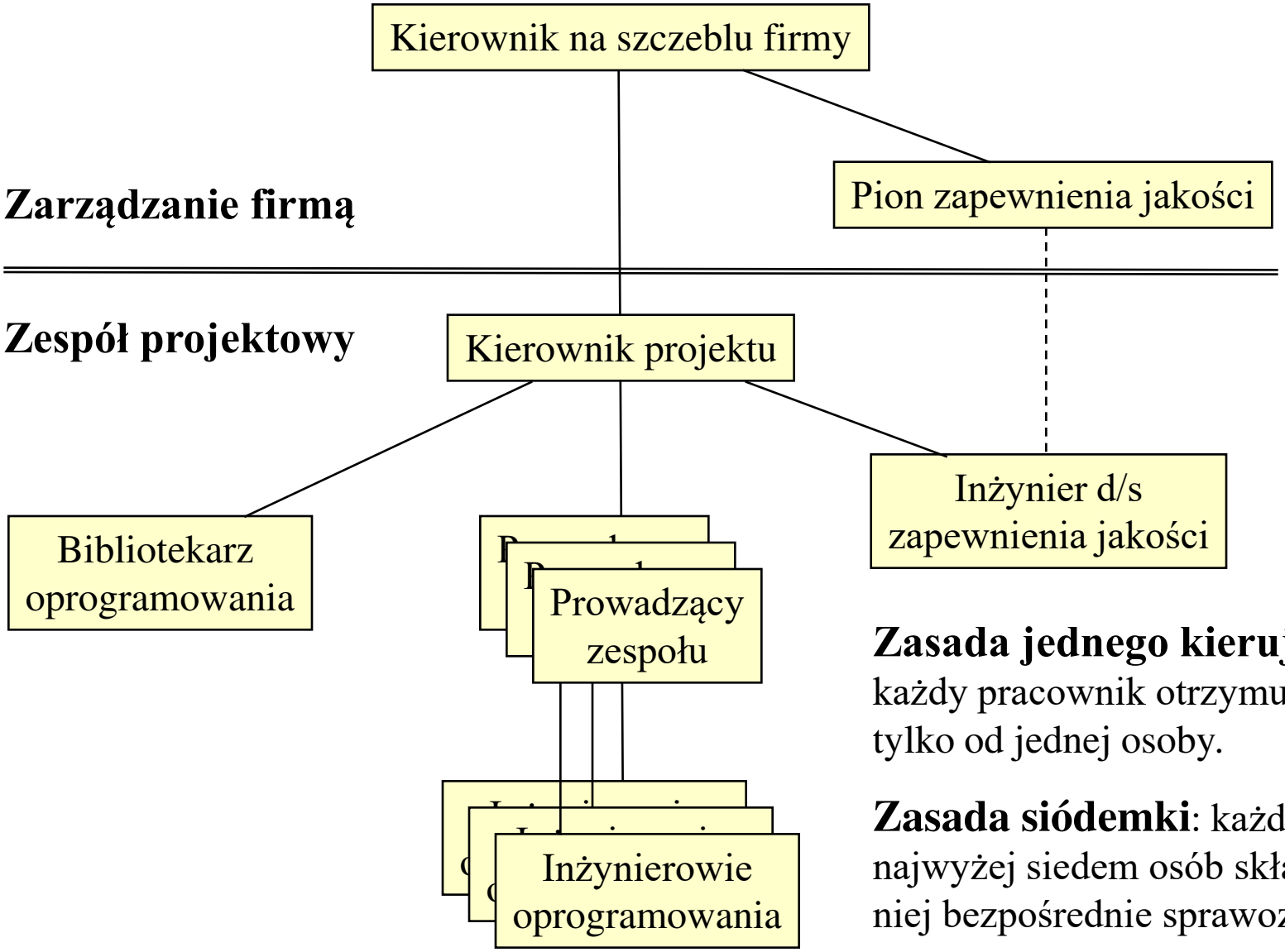
5 Pakiety prac, harmonogram i budżet (pakiety prac, zależności pomiędzy pracami, wymagania odnośnie zasobów, alokacja budżetu i zasobów, harmonogram)

Dodatki: Materiał nie mieszczący się w powyższym wykazie treści

Oszacowanie zasobów i czasu trwania

- **Zdefiniowanie zasobów ludzkich**
- **Określenie ról osób w projekcie**
- **Zdefiniowanie związków pomiędzy rolami celem koordynowania i sterowania projektem**
- **Oszacowanie pracochłonności:**
  - metoda COCOMO, Analiza Punktów Funkcyjnych (FPA), …
- **Oszacowanie kosztów nie związanych z pracą**
  - produkty komercyjne wchodzące w skład końcowego produktu
  - komercyjne produkty użyte do wytworzenia produktu
  - materiały
  - wewnętrzne udogodnienia (np. komputery użyte do testów)
  - zewnętrzne usługi (np. kopiowanie)
  - podróże i delegacje
  - pakowanie i wysyłka
  - ubezpieczenie
- **Oszacowanie czasu trwania**
  - metoda COCOMO, Analiza Punktów Funkcyjnych (FPA). …

Przykładowy diagram organizacji projektu



Zasada jednego kierującego: każdy pracownik otrzymuje polecenia tylko od jednej osoby.

Zasada siódemki: każda osoba ma co najwyżej siedem osób składających do niej bezpośrednie sprawozdania.

Techniczne zarządzanie projektem

Kierownik projektu **powinien rozumieć** projekt od strony technicznej.

Jest on/ona odpowiedzialny (-a) za główne decyzje techniczne:

- metody i narzędzia
- standardy projektowania i kodowania
- model logiczny
- wymagania na oprogramowanie
- model fizyczny
- projekt architektury
- szczegółowe projektowanie
- zarządzanie konfiguracjami
- weryfikację i walidację
- zapewnienie jakości.

W średnich i dużych projektach kierownik może oddelegować niektóre z tych kompetencji do osób prowadzących zespoły.

Zarządzanie ryzykiem

Wszystkie projekty są obarczone ryzykiem.

Zarządzanie ryzykiem polega na:

- zredukowaniu prawdopodobieństwa wystąpienia okoliczności zagrożeń,
- zminimalizowaniu skutków zagrożeń, które wystąpiły.

Aktywności kierownika:

- ciągłe śledzenie okoliczności, które mogą stać się zagrożeniami projektu,
- poprawianie planu celem zminimalizowania prawdopodobieństwa zagrożenia,
- określenie planu awaryjnego na wypadek okoliczności zagrożenia,
- wdrożenie planu w wypadku wystąpienia okoliczności zagrażającej.

Zarządzanie ryzykiem nigdy nie powinno zaczynać się od optymistycznego założenia „wszystko pójdzie dobrze” („jakoś to będzie”), ale raczej od pytania „co najprawdopodobniej może pójść źle?”. Nie jest to pesymizm, ale realizm.

Czynniki ryzyka (1)

Czynniki doświadczenia

- **brak doświadczenia i/lub kwalifikacji kierownika projektu** (niedoświadczony kierownik jest poważnym zagrożeniem dla projektu),
- **brak doświadczenia i/lub kwalifikacji personelu** (personel powinien być sprawdzony pod względem kwalifikacji, powinien być przypisany do odpowiednich zadań, ...)
- **niedojrzałość dostawców** (brak sukcesów w rozwijaniu podobnych projektów, brak standardów, brak certyfikatu ISO 9000, ...).

Czynniki planowania

- **niedokładność metod szacowania** czasu, kosztów, zasobów,
- **zbyt krótka skala czasowa** (niemożliwość zrównoleglenia pewnych prac),
- **zbyt długa skala czasowa** (zmiany wymagań, personelu, technologii),
- **zależność od awarii losowych, wandalizmu i sabotażu** (zniszczenie sprzętu, zniszczenie danych, itd.),
- **zła lokalizacja personelu** (utrudnienia w komunikacji),
- **zła definicja odpowiedzialności** (brak odpowiedzialnych za kluczowe zadania, wykonywanie niepotrzebnych lub drugorzędnych zadań, ...),
- **częste zmiany personelu** (nowy personel wymaga czasu dla zapoznania się z dotychczasowymi pracami).

Czynniki ryzyka (2)

Czynniki technologiczne

- **nowość technologiczna** (brak doświadczeń, konieczność dodatkowego wysiłku na rozpoznanie, ...),
- **niedojrzałość lub nieodpowiedniość stosowanych metod** (nowe metody są często niesprawdzone, konieczne jest praktyczne doświadczenie, ...),
- **niedojrzałość lub nieodpowiedniość narzędzi** (personel powinien umieć je używać, mogą być nieodpowiednie w stosunku do metod, są zmieniane w trakcie projektu, ...),
- **niska jakość użytego komercyjnego oprogramowania** (może być przereklamowane, może nie być niezawodne, pielęgnowalne, bezpieczne, stabilne, ...),

Czynniki zewnętrzne

- **niska jakość lub niestabilność wymagań użytkownika,**
- **słabo zdefiniowane, niestabilne lub niestandardowe interfejsy zewnętrzne**,
- **niska jakość lub słaba dostępność systemów zewnętrznych** (od których zależy powodzenie projektu; może być konieczne rozwijanie możliwości symulujących systemy zewnętrzne).

Tabela ryzyka

Kroki do wykonania tabeli ryzyka:

- wylistowanie zagrożeń projektu
- zdefiniowanie prawdopodobieństwa dla każdego zagrożenia
- zdefiniowanie akcji zmierzających do zmniejszenia ryzyka lub akcji alternatywnej
- określenie daty decyzji
- zdefiniowanie możliwego wpływu wystąpienia zagrożenia dla celów projektu

Ryzyko	Opis	Prawdop.	Akcja	Data decyzji	Wpływ
1	Nowe wymagania użytkownika	Wysokie	Zmiana cyklu na rozwoju oprogramowania na ewolucyjne	1.06.2020	wysoki
2	Instalacja urządzenia klimatyzacyjnego	Średnie	Przemieszczenie personelu do czasu zakończenia instalacji	5.04.2020	średni

Macierz ryzyka

Przedstawia w sposób poglądowy ryzyka projektu, ich prawdopodobieństwo oraz wpływ.

prawdopodobieństwo	mały	średni	wysoki
wysokie			Nowe wymagania użytkownika
średnie		Instalacja urządzenia klimatyzacyjnego	
małe			

wpływ

Pomiary procesów i produktów projektu

Metody pomiarów liczbowych powinny być stosowane wszędzie tam, gdzie jest to możliwe.

Kroki pomiarowe:

- Oszacowanie środowiska projektowego i zdefiniowanie głównych celów (cele finansowe, jakość, niezawodność, ...)
- Analiza głównych celów celem wydzielenia pod-celów, które mogą być efektywnie zmierzone; zdefiniowanie metryk pomiarowych dla każdego pod-celu.
- Kolekcjonowanie danych pomiarowych oraz pomiar ich wpływu na cele projektu.
- Poprawienie działań w ramach projektu poprzez korygowanie odchyleń od celów projektu.
- Poprawienie działania całości organizacji poprzez przesłanie danych pomiarowych do odpowiednich zespołów odpowiedzialnych za standardy i procedury stosowane w projekcie.

Analiza celów i definiowanie metryk

Powszechne pod-cele w stosunku do głównego celu:

- nie przekraczać planowanej pracochłonności dla każdej aktywności
- nie przekraczać planowanego czasu dla każdej aktywności
- zapewnić kompletność produktu
- zapewnić niezawodność produktu

Metryki procesów:

- ilość użytych zasobów
- ilość niezużytych zasobów
- okres danej aktywności w dniach, tygodniach lub miesiącach
- przesunięcie czasowe aktywności (*aktualny początek* - *planowany początek*)
- ilość zakończonych prac
- ilość rozwiązanych problemów programistycznych

Metryki produktów:

- liczba linii napisanego kodu bez komentarzy
- liczba zakodowanych i przetestowanych modułów
- liczba zaimplementowanych punktów funkcyjnych
- liczba napisanych stron dokumentacji
- procentowe pokrycie kodu testami
- złożoność cyklomatyczna modułów źródłowych
- złożoność integracji programów
- ilość krytycznych problemów do rozwiązania
- ilość nie-krytycznych problemów do rozwiązania
- ilość zmian produktu od pierwszego wydania lub wdrożenia

Metody estymacyjne

- **Porównanie historyczne**: porównanie bieżącego projektu z poprzednimi projektami i na tej podstawie oszacowanie kosztu/czasu/pracochłonności. **Wady**: (1) trudności z uwzględnieniem nowych technologii; (2) nie uwzględnianie doświadczenia zespołu (3) zła praca podlega instytucjonalizacji.
- **COCOMO**: oparta na formułach ustalających koszt/czas na podstawie linii kodu. **Wady**: (1) liczba linii kodu jest nieznana przed zakończeniem prac; (2) może być sztucznie pomnażana przez wykonawców; (3) nie ma zastosowania dla nowoczesnych technik programistycznych.
- **Analiza punktów funkcyjnych**: suma punktów za funkcjonalne (zewnętrzne) cechy produktu.
- **Analiza podziału aktywności**: skorzystanie z poprzednich doświadczeń i porównanie obecnych cząstkowych prac z analogicznymi wykonywanymi poprzednio. Wady: analogiczne do wad porównania historycznego.
- **Metoda Delphi:** wykorzystanie ekspertów i koordynowana negocjacja pomiędzy ekspertami.
- **Estymacje pracochłonności integracji systemu i testowania**
- **Estymacje pracochłonności dokumentowania.**

Bieżące raportowanie

Dokładne i aktualne raportowanie jest istotne dla sterowania projektem.
Kierownicy projektu dają raporty dla inicjalizatorów i zwierzchnictwa projektu.
Członkowie zespołu dają formalne raporty dla swoich bezpośrednich kierowników.

Raporty odnośnie postępu prac (rutynowe, np. co miesiąc):
- stan techniczny
- stan zasobów
- stan harmonogramu
- napotkane problemy
- stan finansowy

Raporty odnośnie zakończonych prac
Kierownik projektu powinien potwierdzić ten raport poprzez wydanie „certyfikatu zakończonej pracy

Raporty obciążenia czasowego pracowników
Raport opisuje (z dokładnością do dni i godzin) czas przeznaczony przez poszczególnych pracowników przeznaczony lub zużyty na wykonanie poszczególnych prac.

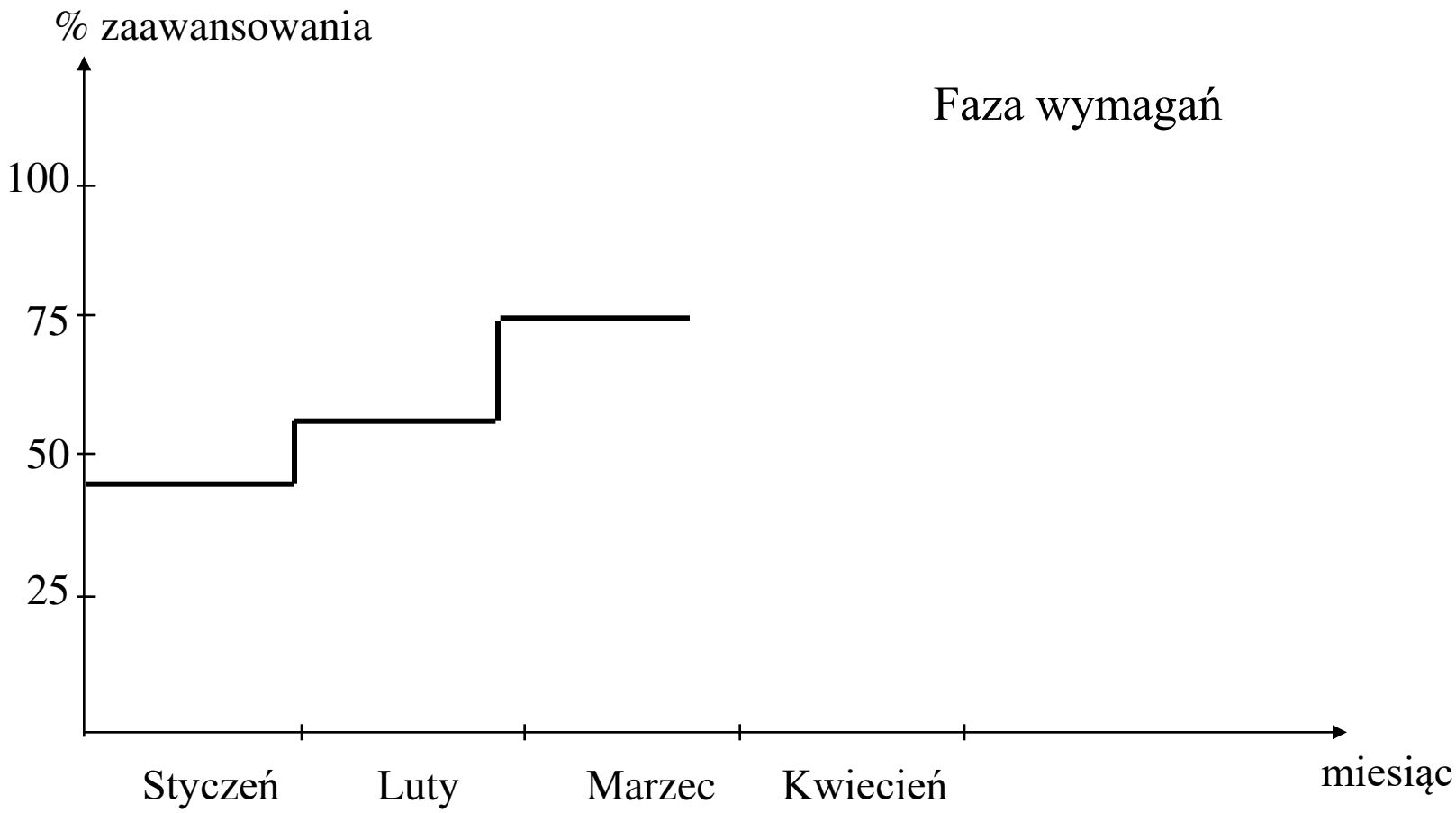
Metody raportowania - tabela postępu prac

Ile czasu wykorzystano i ile pozostało.

Id	**Nazwa**	**Plan**	**Styczeń** PoSz DoKo WwO SumW	**Luty** PoSz DoKo WwO SumW	**Marzec** PoSz DoKo WwO SumW	**Razem**
2210	Model logiczny	20	20 10 20 20	30 0 10 30	30 0 0 30	30
2220	Prototyp	30	30 15 20 20	35 5 15 35	40 0 5 40	40
2230	Dokument wymagań	25	25 25 0 0	25 15 10 10	25 0 15 25	25
2200	Razem faza wym.	75	75 50 40 40	90 20 35 75	95 0 20 95	95

PoSz - poprzednie oszacowanie
WwO - wydatki w okresie
DoKo - wydatki pozostałe do dońca
SumW - wydatki sumaryczne

Metody raportowania - wykres postępu



Narzędzia do zarządzania projektem

Rozliczne narzędzia i metody do planowania, analizy ryzyka, raportowania i wspomagania procesu wytwarzania.

Wspomaganie (np. ze strony narzędzi CASE) m.in. dla:

- Zdefiniowania pakietów prac
- Zdefiniowania zasobów i ich dostępności
- Przypisania zasobów do pakietów prac
- Określenia czasu trwania pakietów prac
- Konstruowania sieci aktywności z użyciem metody PERT
- Ustalenia ścieżki krytycznej (najdłuższej) w sieci aktywności
- Zdefiniowania harmonogramu przy użyciu diagramu Gantta.
- Obliczenie sumarycznej ilości wymaganych zasobów
- Ustalenia czasu trwania aktywności
- Podziału projektu na pod-projekty
- Modelowania procesów (definicji procedur i ich analiza)
- Estymacji kosztu, czasu, pracochłonności

Przykładowe pytania na egzamin

1. Jakie elementy zasobów można pomierzyć, jakie ich cechy mogą być mierzalne bezpośrednio?
2. Omów cel zarządzania konfiguracją oprogramowania.
3. Na czym polega metoda COCOMO i z jakich danych wejściowych korzysta?.
4. Jakie rodzaje zagrożeń mogą wystąpić w procesie zarządzania przedsięwzięciem programistycznym?
5. Czym różni się metryka od pomiaru ?
6. Wymień elementy projektu i oprogramowania, które powinny być przedmiotem zarządzania konfiguracją oprogramowania.
7. Do czego wykorzystuje się metodę punktów funkcyjnych?
8. W jaki sposób zarządzający przedsięwzięciem programistycznym powinni stawić czoła możliwym zagrożeniom?
9. Czego mogą dotyczyć metryki oprogramowania? Podaj przykłady.
10. Omów pojęcie *wersja*, związane z zarządzaniem konfiguracją oprogramowania.
11. Omów, na czym polega metoda punktów funkcyjnych i z jakich danych wejściowych korzysta się w tej metodzie.
12. Na czym polega zarządzanie ryzykiem i jak jest w tym rola kierownika projektu?

Przykładowe pytania egzaminacyjne

1. Scharakteryzuj i porównaj model kaskadowy i model spiralny. Oceń wady i zalety obu tych modeli.
2. Jakie znasz algorytmiczne modele kosztów? Określ jaki rodzaj kosztów one liczą.
3. Krótko scharakteryzuj i porównaj wymagania funkcjonalne i niefunkcjonalne.
4. Jakie kryterium określa podział projektu na części, zaś oprogramowania na moduły? Krótko je omów.
5. Po co są transakcje w bazach danych?
6. Co rozumiesz pod pojęciem konserwacja oprogramowania?