SEMESTR LETNI 2023

PROJEKTOWANIE SYSTEMÓW INFORMATYCZNYCH

NIEZBĘDNY PODRĘCZNIK – OD STUDENTA DLA STUDENTA

KANAŁ STUDENTCKI

SOFTWARE ENGINEERING

1. Formalna definicja software engineering
2. Istotne cechy software engineering
3. Fazy software engineering

Problem: Jak kontrolować proces powstawania oprogramowania? Czy jest to możliwe tak, jak w przypadku tradycyjnych dziedzin inżynieryjnych?

SOFTWARE ENGINEERING

- **Software engineering jest systematycznym, ustrukturyzowanym, mierzalnym podejściem do tworzenia, działania i utrzymania oprogramowania; jest zastosowaniem technik inżynieryjnych do tworzenia oprogramowania.**
- Metody i techniki niezbędne do stworzenia dużych systemów informatycznych
- Wykorzystanie technik inżynieryjnych dla otrzymania w sposób efektywny oprogramowania, które jest wiarygodne i funkcjonuje efektywnie.

Efekt: Już nie tworzymy Odtwarzacza w javie po omacku pół semestru, tylko mamy konkretne metody przybliżające nas do celu - działającego oprogramowania.

ISTOTNE CECHY SOFTWARE ENGINEERING

DLA DUŻYCH APLIKACJI
aplikacje wymagające udziału więcej niż jednej osoby oraz większego nakładu czasu i pracy

ZARZĄDZANIE KOMPLEKSOWOŚCIĄ
podział systemu na moduły i komunikacja między nimi

WSPOMAGANIE WSPÓŁPRACY
jasne, określone techniki i zasady ułatwiają dzielenie zadań i zarządzanie współpracą

ZARZĄDZANIE WYMAGANIAMI
radzenie sobie z zmieniającą się rzeczywistością oraz wymaganiami klientów

EFEKTYWNOŚĆ
optymalizacja wykorzystywanych zasobów, takich jak pamięć, ludzie czy czas; optymalizacja algorytmów i struktur - oprogramowanie o wyższej wydajności

FAZY SOFTWARE ENGINEERING

1.Definicja problemu

Identyfikacja problemu rozwiązywanego przez oprogramowanie i zrozumienie jego natury.

2. Specyfikacja wymagań

Określenie wymagań funkcjonalnych (jakie funkcje powinien mieć system) i niefunkcjonalnych (jaka wydajność, bezpieczeństwo, interfejs).

3. Projektowanie

Tworzenie planu struktury i architektury systemu, podjęcie decyzji o komponentach systemu i interakcji między nimi. To tutaj używamy wiedzy z UML i ćwików.

4. Programowanie

Tworzenie kodu źródłowego na podstawie specyfikacji wymagań i projektu.

5. Testowanie

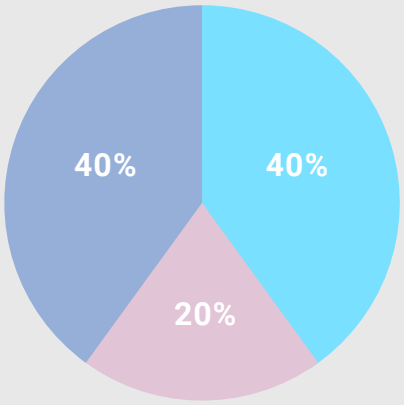
Weryfikacja i walidacja oprogramowania, znalezienie błędów, niezgodności.

7. Utrzymanie systemu

Dostosowywanie systemu do pojawiających się w czasie wymagań, zarządzanie zmianami, wsparcie techniczne użytkowników. Koszty utrzymania często wynoszą 50-75% kosztów potrzebnych na stworzenie systemu.

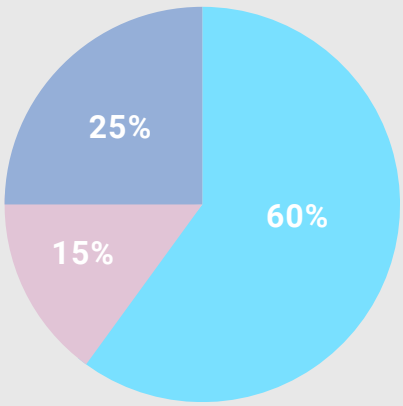
ZASADA 40-20-40

- 40% zasobów - definicja problemu, specyfikacja wymagań, projektowanie
- 20% zasobów - programowanie
- 40% zasobów - testowanie



ZASADA 60-15-25

- 60% zasobów - definicja problemu, specyfikacja wymagań, projektowanie
- 15% zasobów - programowanie
- 25% zasobów - testowanie



FUNDAMENTALNE ZASADY SOFTWARE ENGINEERING

1. Minimalizowanie interakcje między komponentami.
2. Tworzenie oprogramowania metodycznie, krok po kroku.
3. Definiowanie mierników jakości dla każdego aspektu.
4. Zarządzanie zmianami – by nie były ani zbyt sztywne, ani pływające.
5. Dokumentowanie kompromisów i porozumień.
6. Analizowanie analogicznych przypadków z przeszłości.
7. Zarządzanie niepewnościami – są nieuniknione.
8. Budowanie komponentów oprogramowania wykorzystując te istniejące i nadających się się do wykorzystania w przyszłości.
9. Kontrolowanie kompleksowości w oparciu o wiele perspektyw i wiele poziomów abstrakcji.
10. Ścisłe definiowanie tego, co chcemy osiągnąć oprogramowaniem (artefaktów).
11. Elastyczność procesu wytwarzania oprogramowania.
12. Dyscyplina procesu wytwarzania oprogramowania.
13. Inwestowanie w zrozumienie problemu.
14. Zarządzanie jakością projektu w trakcie całego cyklu życia oprogramowania tak formalnie, jak to tylko możliwe.
15. Wykorzystywanie miar ilościowych w podejmowaniu decyzji.

SPRAWDŹ SIĘ!

Cechą oprogramowania uzyskanego w procesie Software Enginnering, zgodnie z definicją jest....

- Wiarygodność
- Bezpieczeństwo
- Niezaprzeczalność
- Użyteczność

W modelu tworzenia oprogramowania 40-20-40, 20 oznacza, że

- 20 procent kosztów związanych jest z testowaniem
- 20 procent czasu należy poświęcić na testowanie
- 20 procent kosztów związanych jest z programowaniem
- 20 procent czasu tworzony jest kod dla rozwiązania

Zasady software engineering: jak tworzyć nowe systemy?

WYMAGANIA UŻYTKOWNIKA

1. Specyfikacja wymagań użytkownika
2. Przykłady wymagań jakościowych
3. Model satysfakcji klienta: diagram Kano

Jak widzimy wymagania użytkownika są niezwykle istotne w tworzeniu oprogramowania i SE. Warto im poświęcić szczególną uwagę.

SPECYFIKACJA (INŻYNIERIA) WYMAGAŃ UŻYTKOWNIKA

- Pierwszy etap każdego dowolnego projektu IT.
- Przeprowadzamy analizę (inżynierię) wymagań, aby móc określić ich specyfikację i ruszyć z projektem.
- **Zebrane w formie dokumentu, będącego kontraktem dla klienta oraz punktem startowym dla projektowania rozwiązania.**

Punkt startowy dla projektowania rozwiązania

Kontrakt dla klienta

WYMAGANIA FUNKCJONALNE

oczekiwane efekty funkcjonowania systemu

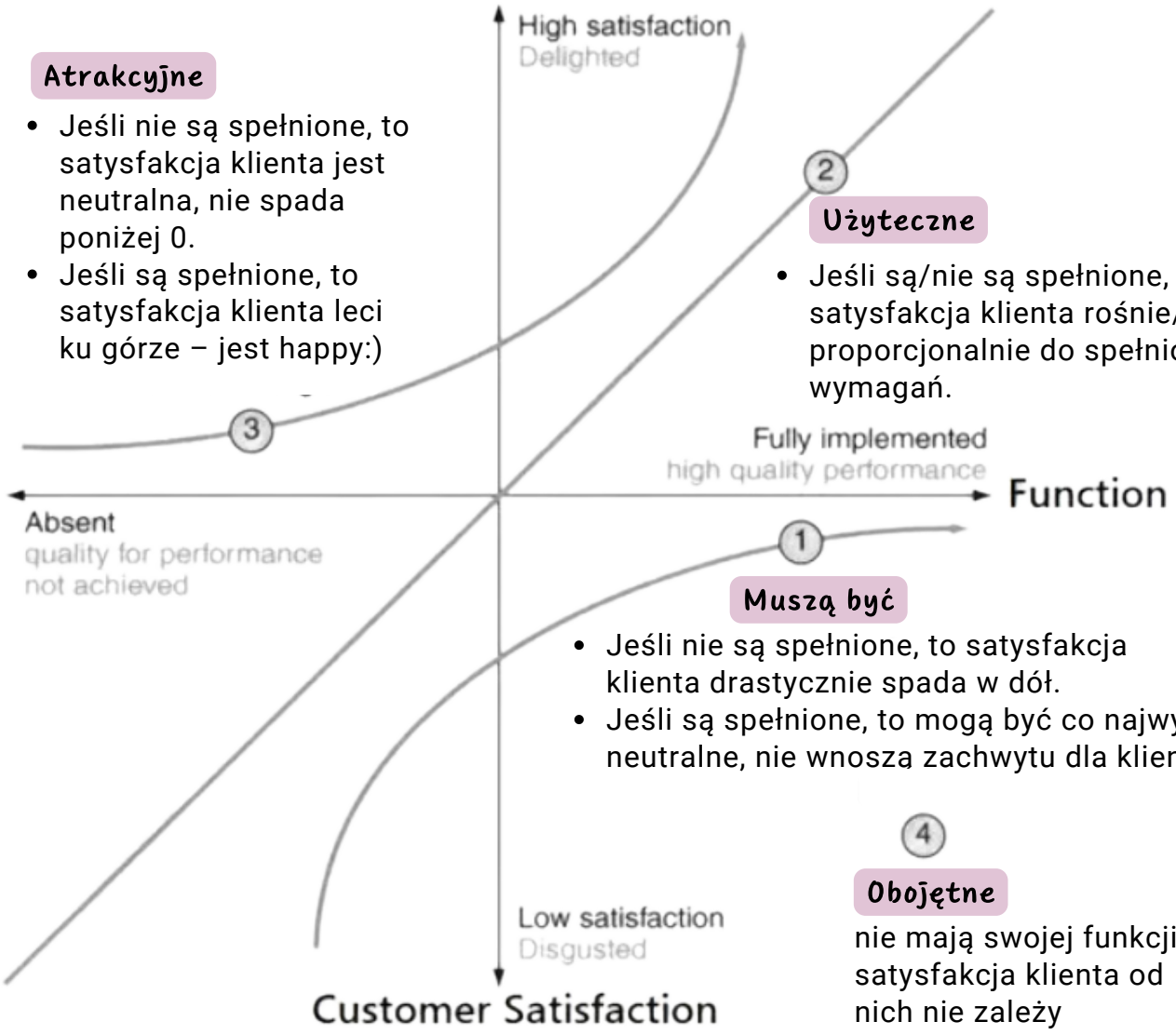
- **Zakres systemu** – główne moduły, użytkownicy docelowi, obszar zastosowania i ograniczenia systemu
- **Konieczne funkcjonalności** – podstawowe funkcje, które system musi posiadać, aby sprostać wymaganiom użytkowników
- **Pożądane dane** – przechowywane i generowane

WYMAGANIA JAKOŚCIOWE

oczekiwana jakość systemu, ograniczenia i zakres implementacji

- **Użyteczność** – dokumentacja, łatwość użycia, dostępność materiałów szkoleniowych
- **Wielorazowość** komponentów
- **Niezawodność** – częstość błędów, szybkość odtworzenia
- **Wsparcie** – help desk, obsługa informatyczna
- **Wydajność** – czas odpowiedzi, przepustowość, zużycie zasobów
- **Interfejs** – wizualny i funkcjonalny

MODEL SATYSFAKCJI KLIENTA: DIAGRAM KANO



Atrakcyjne

- Jeśli nie są spełnione, to satysfakcja klienta jest neutralna, nie spada poniżej 0.
- Jeśli są spełnione, to satysfakcja klienta leci ku górze – jest happy:)

Użyteczne

- Jeśli są/nie są spełnione, to satysfakcja klienta rośnie/spada proporcjonalnie do spełnionych wymagań.

Muszą być

- Jeśli nie są spełnione, to satysfakcja klienta drastycznie spada w dół.
- Jeśli są spełnione, to mogą być co najwyżej neutralne, nie wnoszą zachwytu dla klienta.

4

Obojętne

nie mają swojej funkcji, satysfakcja klienta od nich nie zależy

MOSCOW

Mo - must have	S - should have	Co - could have	W - will not have
najbardziej potrzebne, bez nich system nie ma sensu	rzeczy istotne, ale nie kluczowe do działania systemu	rzeczy, które miło mieć, jak starczy czasu i ochoty	poza budżetem, spoko są, ale nie mają istotnego wpływu

GRZECHY ANALITYKA

- Przyjmowanie własnych założeń – słuchamy piąte przez dziesiąte
- Brak wiedzy, zapominanie o wymaganiach (brak adresowania)
- Opisywanie rozwiązania zamiast problemu
- Sprzeczności między wymaganiami oraz wewnętrzne samego wymagania
- Niejasne, wieloznaczne wymagania – utrata istoty problemu
- Referencje do jeszcze nieopisanych wymagań
- Rozważanie cech niemożliwych do zaimplementowania

SPRAWDŹ SIĘ!

Użyteczność systemu jest wymaganiem pozafunkcyjnym określającym
- Wszystkie odpowiedzi są poprawne
- Łatwość użycia
- Dostępność materiałów szkoleniowych
- Dostępność dokumentacji dla systemu

Łatwość nauczenia, szybkość działania, odporność na błędy użytkownika, czy zdolność wycofania wskazują na następujące wymagania jakościowe
- Czytelność
- Prostota
- Użyteczność
- Efektywność

Typowym „grzechem" analityka w procesie specyfikacji wymagań nie jest
- Doprowadzenie do sprzeczności między wymaganiami
- Brak zaadresowania wymagania
- Opis rozwiązania problemu użytkownika
- Tworzenie powiązań między funkcjonalnościami (tworzenie hierarchii wymagań)

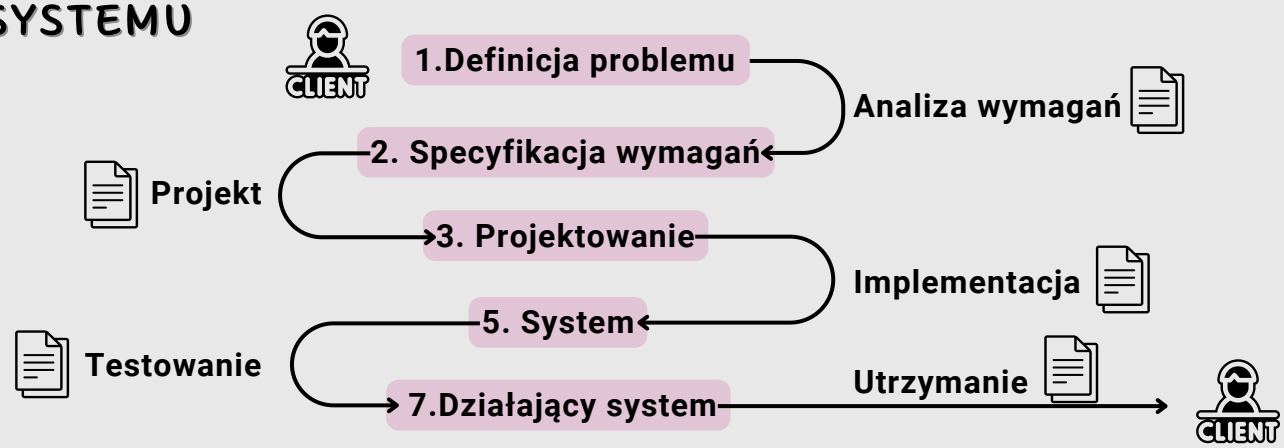
Specyfikacja wymagań użytkownika
- Stanowi punkt startowy dla definicji testów systemu
- Jest kontraktem dla klienta systemu
- Stanowi punkt startowy dla projektu systemu
- Wszystkie odpowiedzi są prawidłowe

W modelu Kano wymagania konieczne (must have) oznaczają wymagania, które
- Muszą być uwzględnione przy tworzeniu a ich uwzględnienie powoduje proporcjonalny wzrost satysfakcji
- Żadna z odpowiedzi nie jest prawidłowa
- Gdy spełnione są neutralne dla użytkownika
- Muszą być uwzględnione i ich spełnienie powoduje wykładniczy wzrost satysfakcji użytkownika

W modelu MoSCoW, C określa
- Wymagania o wysokim priorytecie realizacji
- Wymagania, które zostaną zaimplementowane jeśli starczy czasu
- Wymagania kluczowe
- Wymagania dla kolejnej wersji systemu

METODYKI PROJEKTOWE

PROSTY CYKL ŻYCIA SYSTEMU

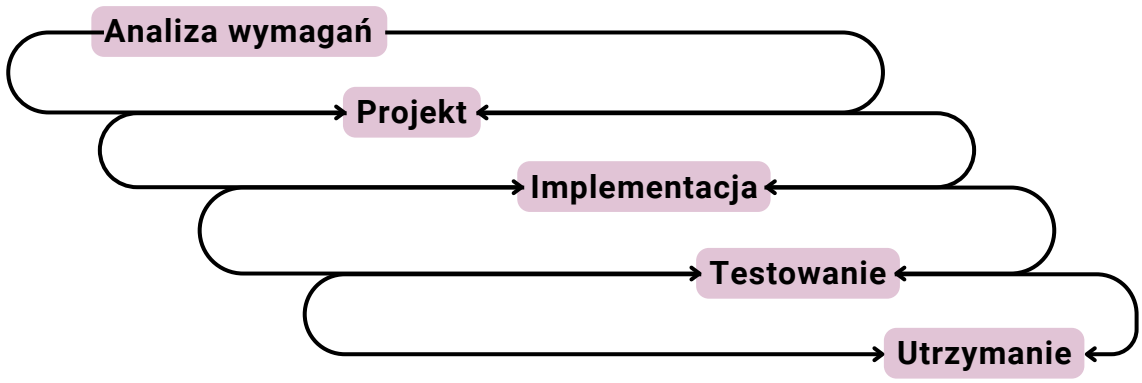


- duża ilość dokumentacji
- mała elastyczność
- szczegółowe kontrakty i plany

Problemy:

- mała styczność z klientem (jedynie na początku i na końcu), ograniczony dostęp do komentarzy i uwag
- nie zakłada ewolucji systemu, utrzymanie dotyczy finalnej wersji

MODEL KASKADOWY (WATERFALL) - LINIOWY



- **każdy etap zakończony weryfikacją i walidacją** - czy system spełnia wymagania i czy wymagania te są słuszne, istotne dla rozwiązania problemu
- **stabilny zestaw potrzeb i celów użytkownika**, potrzeby nie zmieniają się w trakcie życia systemu
- **stopniowe uszczegóławianie oprogramowania**, stopniowy proces budowy, krok po kroku, jeden etap się kończy, drugi zaczyna

METODY ELASTYCZNE - AGILE

Dostarczanie wartości klientowi tak szybko, jak to możliwe. Postawienie na kontakt z klientem i częste iteracje zadań w oparciu o jego feedback. Zmiana nieodłącznym elementem procesu, nie boimy się jej.

MANIFEST AGILE

- **Indywidualne jednostki i ich interakcje są ważniejsze** niż proces i narzędzia.
- **Działające oprogramowanie jest ważniejsze** niż szczegółowa dokumentacja.
- **Współpraca z klientem jest ważniejsza** od negocjacji warunków kontraktu.
- **Reagowanie na zmianę jest ważniejsze** niż plan.

WNIOSKI - CECHY AGILE

- Iteracje krótkie, system tworzony jest inkrementalnie, narastająco dodajemy kolejne funkcjonalności
- Planowanie maksymalnie jednego cyklu w przód, nie wybiegamy daleko
- Działające oprogramowanie na koniec każdego cyklu
- Architektura nie jest szczegółowo projektowana przed kodowaniem
- Refactoring: poprawianie projektu po każdej iteracji
- Wiedza ukryta vs. dokumentacja - nie wszystko jest dokumentowane

PROTOTYPOWANIE

- **Konieczny, gdy wymagania nie są znane**
- Duża interakcja z użytkownikiem
- Redukcja czasu oczekiwania na rezultaty
- Priorytetyzacja interfejsu – klient musi widzieć efekty
- Języki programowania dające szybko widoczne rezultaty

Zalety

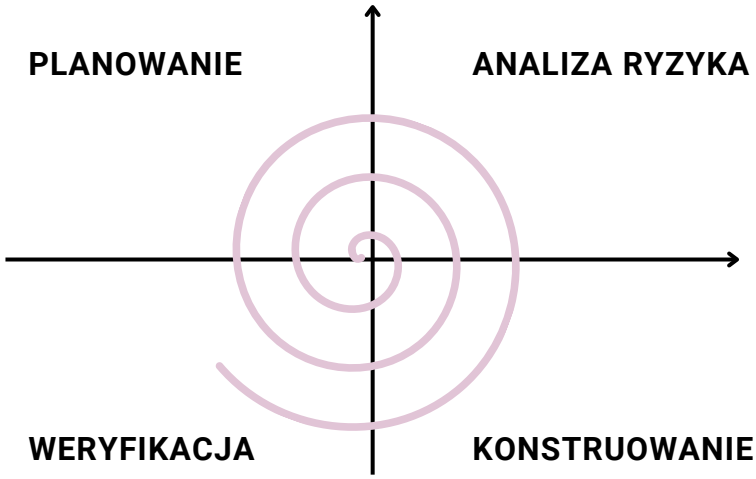
- Łatwy w użyciu system o mniejszej liczbie cech (gdy klienci się ograniczają)
- Szansa dla użytkownika na lepsze wyrażenie swoich potrzeb i wizji
- Wyższa jakość projektu
- Łatwiejsze wykrywanie błędów
- Łatwiejsze utrzymanie systemu

Wady

- Więcej cech (użytkownicy mogą się nie ograniczać)
- Niższa jakość, jeśli finalny produkt dalej będzie tylko prototypem
- Cięższy w utrzymaniu
- Wymaga doświadczonych programistów

MODEL SPIRALNY

- Dobry, aby uniknąć przedefiniowania systemu, **zaczyna się od najbardziej istotnych funkcjonalności oraz problemu najwyższego ryzyka.**
- Etapy powtarzane, dzięki czemu następuj doskonalenie kolejnych wersji.



Zalety

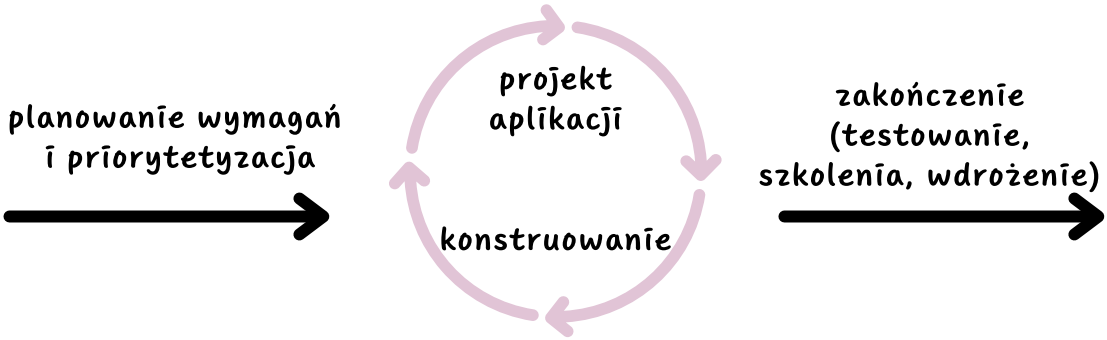
- Da się określić kamienie milowe
- Elastyczność
- Łatwiejsze wykrywanie błędów
- Zarządzanie ryzykiem (zaczynamy od najbardziej ryzykownych)

Wady

- Trudności w zarządzaniu - harmonogramowanie i budżetowanie utrudnione ze względu na ilość iteracji

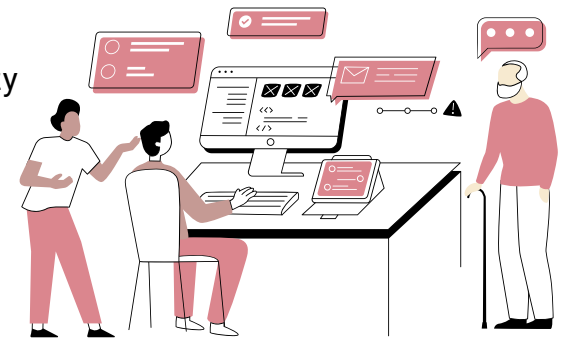
RAPID APPLICATION DEVELOPMENT

- Tworzenie oprogramowania w małych grupach, wielofunkcyjnych grupach.
- Odwrócona kolejność planowania – najpierw wyznaczany jest okres czasu, a dopiero potem, co w tym czasie ma powstać.



EXTREME PROGRAMMING

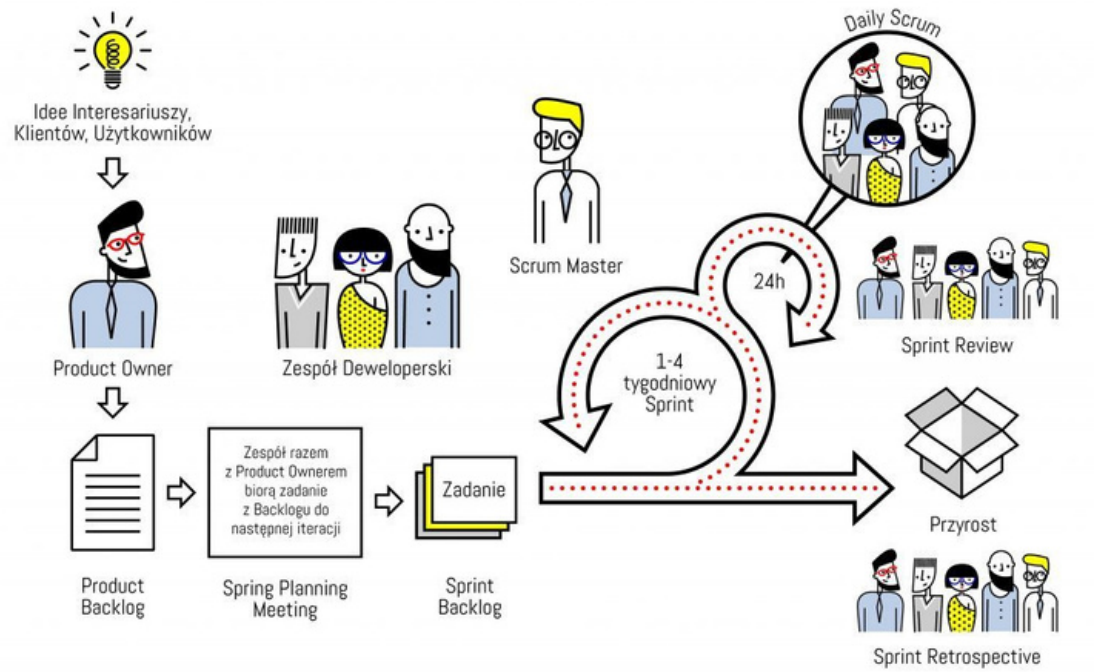
- Programowanie w parach – dwie osoby przy jednym monitorze
- Wiele wersji, małe zmiany
- Szybkie określanie kolejnych interakcji
- Ciągłe testowanie i integracja kolejnych elementów
- Ciągła obecność klienta w zespole



Szybka informacja zwrotna od zespołu i klienta, stawianie na prostotę, inkrementalne tworzenie systemu, zajmowanie się zmianami wtedy, gdy trzeba, koncentrowanie się na jakości.

SCRUM

- Podzielenie rozwoju produktu na mniejsze, 2-6 tygodniowe sprinty. Podczas sprintu wykonuje się określone na początku sprintu zadania.
- Codziennie podczas sprintu odbywają się spotkania "daily", na których omawiane są postępy.
- Po każdym sprincie dostarcza się działający prototyp oraz dokonuje ewaluacji sprintu.
- **Product backlog** – lista wymagań, funkcjonalności i zadań, które muszą zostać zrealizowane w projekcie. Dynamiczny dokument, ewoluujący wraz z postępem projektu i rozwojem wymagań.
- **Sprint backlog** – lista funkcjonalności i zadań realizowanych w ciągu sprintu.



RATIONAL UNIFIED PROCESS

- Wykorzystywanie UML - Use Case do iteracyjnej definicji wymagań, połączenie podejścia iteracyjnego i dokumentacji (wykorzystywanie jej dobrych aspektów)
- Wygląd iteracji zmienia się w czasie w zależności od dojrzałości systemu
- Możliwość ulepszania systemu w czasie trwania projektu
- Możliwość oceny statusu projektu

LEAN

Tworzenie systemu wertykalnie - każde wymaganie po trochu, żeby mieć sprawnie działający system o rosnących poziomach skomplikowania.

SPRAWDŹ SIĘ!

Ocena systemu lub komponentu podczas lub na końcu procesu jego rozwoju na zgodności z wyspecyfikowanymi wymaganiami to (wybierz najpełniejszą odpowiedź)

- Atestowanie
- Weryfikacja
- Testowanie
- Formalny odbiór przez klienta

Brak możliwości zmian w działaniu etapów zakończonych to cecha modelu cyklu życia systemu informatycznego

- Typu Agile
- Przyrostowego
- Iteracyjnego
- Liniowego

Podejście extreme programming zakłada

- Wszystkie powyższe są prawidłowe
- Ciągły udział klienta w tworzeniu oprogramowania
- Ciągłe monitorowanie jakości tworzonego oprogramowania
- Podejście iteracyjne do tworzenia oprogramowania

W sytuacji projektowej, gdy nieznane są zarówno produkt, proces tworzenia, jak i zasoby, które możemy wykorzystać najlepszą strategią wykonania będzie

- Podejście agile
- Prototypowanie
- Podejście inkrementalne
- Wszystkie odpowiedzi są prawidłowe

Cechą kaskadowego podejścia do tworzenia oprogramowania nie jest

- Ciągły kontakt z klientem
- Łatwość rozliczeń z klientem
- Techniczna łatwość harmonogramowania i budżetowania
- Fromalny odbiór poszczególnych etapów prac

W modelu MoSCoW, C określa

- wymagania o wysokim priorytecie realizacji
- wymagania, które zostaną zaimplementowane jeśli starczy czasu
- wymagania kluczowe
- wymagania dla kolejnej wersji systemu

Następujące cechy, działające modele systemu, duża interakcja z użytkownikiem, redukcja czasu oczekiwania na rezultaty, czy nacisk na interfejs użytkownika to klasyczne cechy

- Prototypowania
- Podejście spiralne
- Rapid Application Development
- eXtreme Programming

ARCHITEKTURA OPROGRAMOWANIA

Po co nam architektura?

- Komunikacja między osobami projektującymi i implementującymi system
- Forma wizualizacji wczesnych decyzji projektowych, umożliwiająca nadawanie priorytetów. Już na kartce możemy dojść do porozumienia między wydajnością a bezpieczeństwem

ARCHITEKTURA SYSTEMU

Architektura to fundamentalna organizacja systemu, jego komponentów i ich relacji w stosunku do siebie i środowiska projektu, a także zbiór zasad zarządzających ich projektowaniem i ewaluacją.

Architektura oprogramowania programu lub systemu komputerowego to struktura lub struktury systemu, na które składają się komponenty programistyczne, widoczne z zewnątrz właściwości tych komponentów oraz występujące między nimi zależności.

ISTOTNE CECHY ARCHITEKTURY

ZESPÓŁ ISTOTNYCH DECYZJI projektowych dotyczących organizacji systemu, odnośnie wymagań ilościowych/jakościowych

WYBÓR ELEMENTÓW STRUKTURALNYCH i ich interfejsów,a także ich kompozycji w większe podsystemy

DEFINICJA KONTEKSTU dla projektu i implementacji systemu

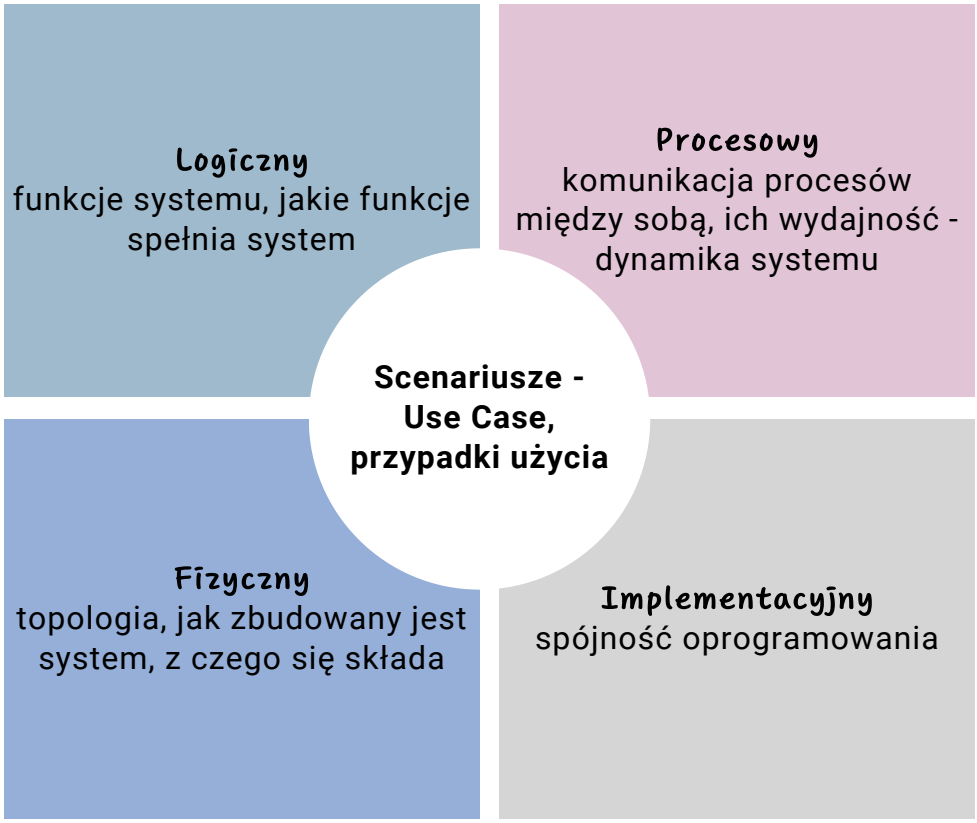
STYLU ARCHITEKTONICZNY przyjęty przez organizację

FUNDAMENTALNE DECYZJE których zmiana pociąga dziesiątki innych zmian

PRZENOSZALNA, UNIWERSALNA nie powinna być ściśle związana z określoną technologią

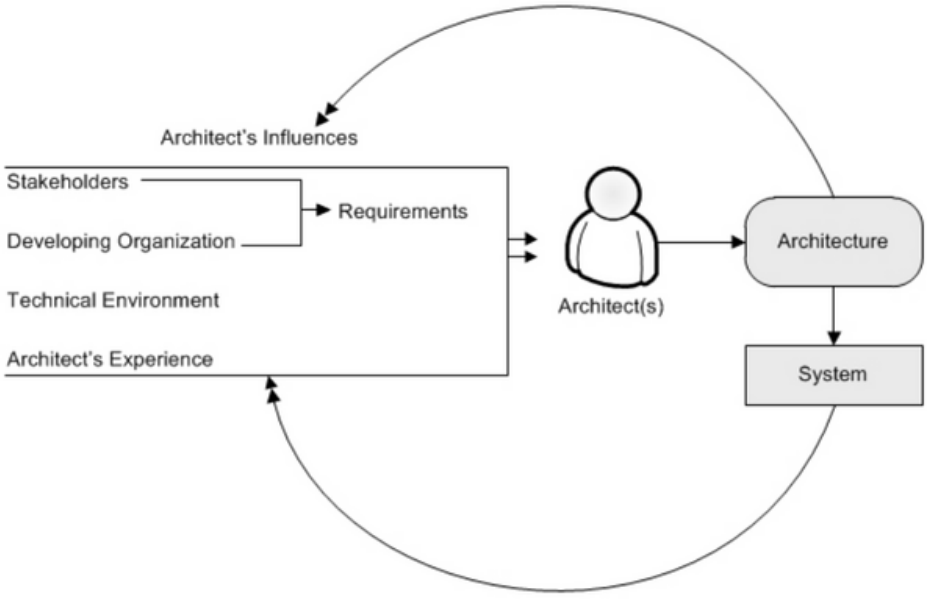
MODEL 4 + 1

Widoki na architekturę, w jakich aspektach warto ją rozważać.



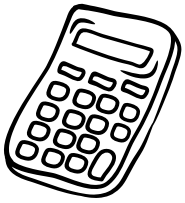
ABC (ARCHITECTURE BUSINESS CYCLE)

- **Proces wzajemnego, cyklicznego (iteracyjnego) oddziaływania środowiska na architekturę i architektury na środowisko - architekci nieustannie się uczą.**
- Budowanie doświadczenia przez daną organizację na określonych obszarach, np. gdy dany bądź podobny system implementowany jest dwukrotnie.
- Każde kolejne rozwiązanie tworzone przez organizację, będzie coraz lepsze.



TYPY KOMPONENTÓW ARCHITEKTONICZNYCH

OBLICZENIOWE



- Wywoływane argumentami
- Odpowiedzialne za operacje obliczeniowe, matematyczne
- Pobieranie i zwracanie parametrów określonego typu
- Bezstanowe - brak zapamiętywania wyników poprzednich operacji, czyszczenie pamięci po zwróceniu wyniku

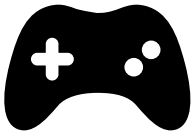
PAMIĘCIOWE

- Przechowywujące dane współdzielone z innymi komponentami, takie jak baza danych, system plików, tablica symboli; zapewniają trwałość danych

ZARZĄDZAJĄCE

- Przechowywujące rezultat, aktualizację stanu
- Zawierające stan i powiązane z nim operacje
- Aktualizowanie stanu w momencie wywołania operacji i przechowywanie do kolejnego wywołania

KONTROLER



- Komponent zarządzający sekwencją zdarzeń, np. moduł kontroli
- Nie zapamiętujące zdarzeń

WZORCE PROJEKTOWE

Wzorce projektowe — kolekcje modułów wspólnie proponujących rozwiązania znanych problemów.

Architecture framework — częściowo gotowy system wdrażany w konkretnych dziedzinach, ze względu na cechy wspólne jego instancje tworzą rodzinę podobnych systemów.

Idiomy — rozwiązania problemów charakterystyczne dla konkretnych technologii, np. wskaźniki w C++.

Styl architektoniczny

- sposób projektowania elementów systemu oraz powiązań między nimi, narzuca pewne ograniczenia
- charakteryzowany przez wzorce opisane przez:
  - problem, kontekst (ograniczenia),
  - rozwiązania (komponenty + powiązania),
  - wariantów rozwiązania i przykładów wykorzystania

STYLE ARCHITEKTONICZNE

SYSTEM Z PODSYSTEMAMI

Problem: System opisany jako szereg wywoływanych po sobie procedur. Możliwy do zrealizowania przez system z kontrolerem (komponentem nadrzędnym), wywołującym kolejne podkomponenty do określonych zadań.

Kontekst: System tworzony w językach programowania, pozwalających na zagnieżdżanie definicji procedur i modułów.

Rozwiązanie: Hierarchia komponentów zależnych od siebie oraz hierarchia danych lokalnych i globalnych. Zwykle jedno repozytorium danych dla komponentów.

Typy komponentów: Grupy procedur (głównie komponenty obliczeniowe).

Typy powiązań: Wywołania procedur, współdzielenie danych za pośrednictwem metod.

WARSTWOWY

Problem: W budowie systemu można wyróżnić różne rodzaje usług, które można powiązać ze sobą hierarchicznie

Kontekst: Każda usługa musi zostać powiązana z określoną warstwą.

Rozwiązanie: System warstwowy o ograniczonej widoczności warstw, czasem tylko do jednej (warstwa widzi usługi jedynie z warstwy podrzędnej). Pogrupowanie w warstwy ułatwia zarządzanie.

Typy komponentów: Obliczeniowe.

Typy powiązań: Wywołanie procedury.

REPOZYTORIUM

Problem: Zarządzanie i przechowywanie ustrukturyzowanej informacji przetwarzanej na wiele różnych sposobów. Dane o długiej ważności – muszą być integralne. Tak jak na GitHubie.

Kontekst: System wzbogacony o bazę danych.

Rozwiązanie: System oparty na wspólnej bazie danych + wiele komponentów. Spójne, długotrwałe przetwarzanie danych. Baza danych w centrum.

Typy komponentów: Jeden pamięciowy i wiele obliczeniowych.

Typy powiązań: Wywołanie procedury, przekazanie danych.

POŚREDNIE WYWOŁANIE

Problem: Słabo powiązana kolekcja komponentów wykonujących określone operacje i umożliwiających wykonanie innych. Brak powiązań generujących i odbierających sygnały.

Kontekst: Komponent przechwytujący zdarzenia i informujący inne komponenty o pojawieniu się zdarzeń.

Rozwiązanie: Model oparty o zdarzenia.

Typy komponentów: Obliczeniowe.

Typy powiązań: Niejawne wywołanie, elastyczność w systemie, brak powiązań między komponentami.

WORKFLOW

Problem: Zarządzanie przepływem danych. Seria niezależnych, sekwencyjnych transformacji danych, zwykle inkrementalne przetwarzanie danych wejściowych. Każdy komponent bazuje na outpucie poprzedniego komponentu, np przetwarzanie danych z Twittera.

Kontekst: System zdekomponowany na serię obliczeń (filtrów), inkrementalnie przetwarzających zadane dane.

Rozwiązanie: System zapewniający przepływ danych pomiędzy komponentami. Komponenty są niezależne.

Typy komponentów: Obliczeniowe.

Typy powiązań: Przekazanie danych.

ABSTRAKCYJNY TYP DANYCH

Problem: Ochrona danych - możliwe zmiany modelu danych w trakcie życia systemu. Enkapsulacja relewantnych obiektów świata rzeczywistego i ich operacji.

Kontekst: Identyfikacja obiektów świata rzeczywistego wchodzących w skład komponentów. Proponowanie przez języki programowania obiektowego pojęcia klasy i ponowne wykorzystywanie kodu. Często obsługa zdarzeń zamiast bezpośredniego wywołania metody.

Rozwiązanie: Obiektówka - każdy komponent zarządza swoimi danymi. Jeśli inny komponent chce ich użyć – musi odwołać się do określonej metody, a nie do określonych danych.

Typy komponentów: Zarządzające.

Typy powiązań: Wywołanie procedury.

OCENA ARCHITEKTURY

Po co nam ocena architektury?
Przewiduje jakość i sprawność systemu. Ponadto może wykryć problemy na wczesnym etapie, co zmniejsza ewentualne koszty poprawek w przyszłości,

Architektura jest dobra, jeśli zbudowany system spełnia wymagania funkcjonalne i jakościowe oraz jest możliwy do zbudowania z dostępnych zasobów.

ATAM - METODA ANALIZY KOMPROMISÓW ARCHITEKTONICZNYCH

Ocena spełnienia wymagań jakościowych oraz ich wzajemny wpływ na siebie - kompromisy. Przed procesem należy stworzyć zespół ewaluacyjny.

ETAPY ATAM

Prezentacja (etapy 1-3)
informacje o projekcie
- prezentacja metody ATAM
- motywacje i kontekst projektu
- architektura

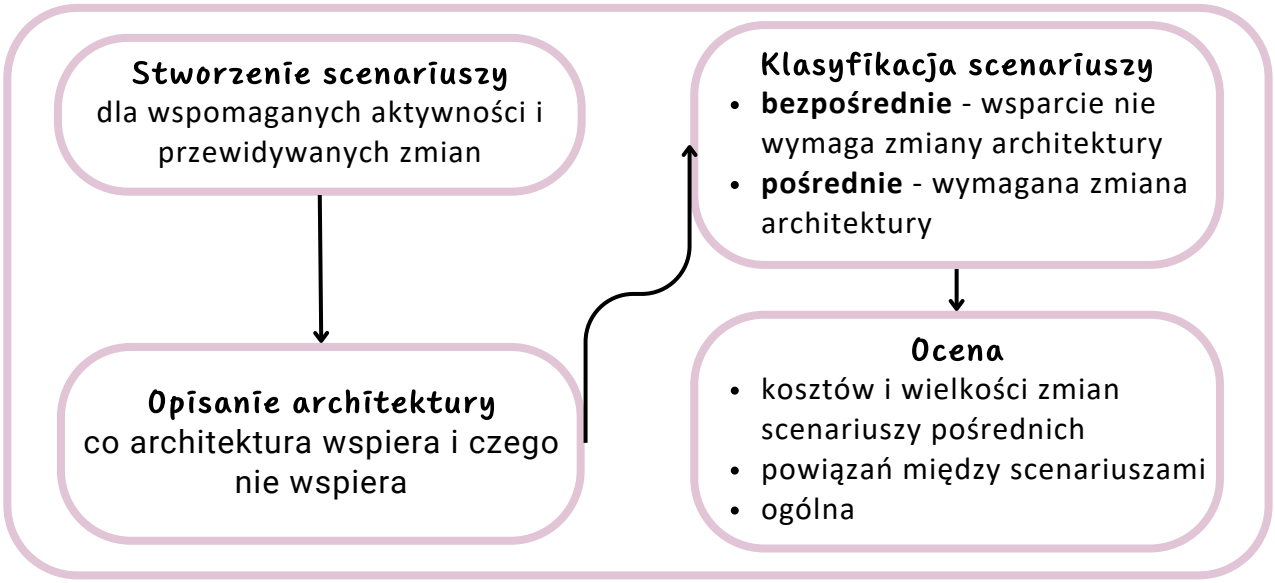
Badanie i analiza (etapy 4-6)
ocena kluczowych atrybutów jakościowych i odniesienie ich do znanych architektur
- identyfikacja decyzji architektonicznych
- drzewo użyteczności cech jakościowych, u=określających użyteczność systemu
- analiza metod architektonicznych

Testowanie (etapy 7-8)
sprawdzanie czy otrzymane rezultaty spełniają potrzeby zainteresowanych
- priorytetyzacja scenariuszy podczas burzy mózgów
- powtórna analiza metod architektonicznych

Raportowanie (etap 9)
przedstawienie wyników ATAM
- prezentacja rezultatów, stworzenie raportu, uaktualnienie repozytoriów

SAAM - SOFTWARE ARCHITECTURE ANALYSIS METHOD

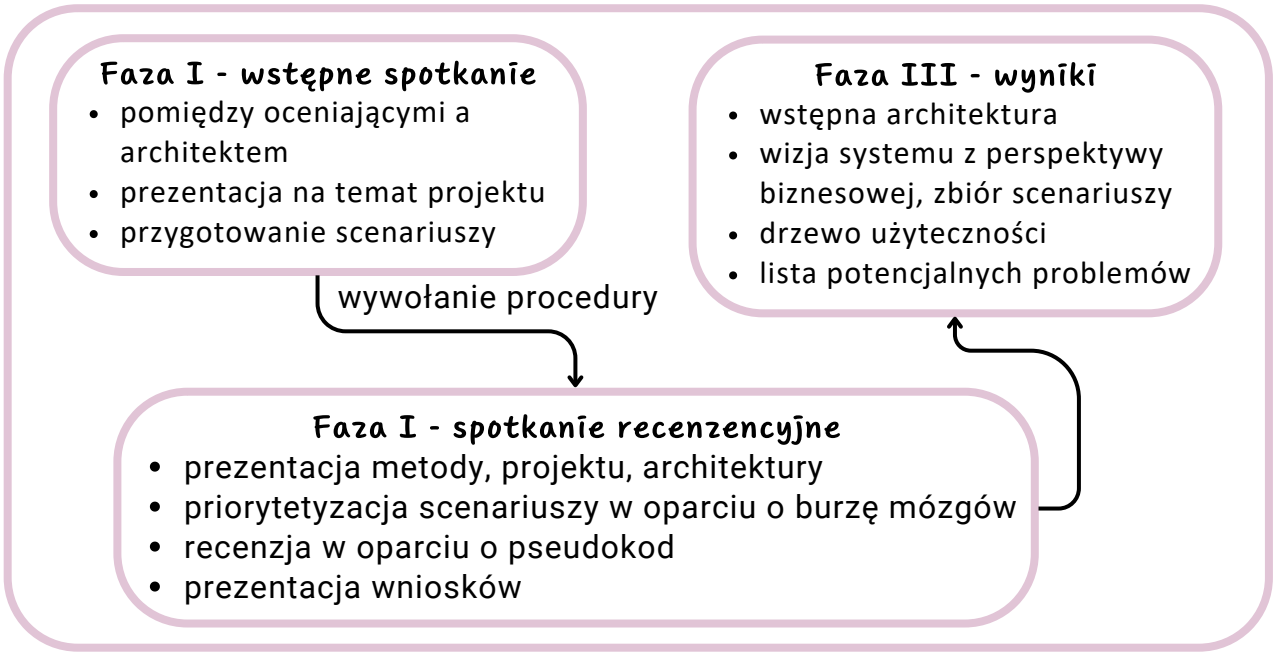
Mniej formalny niż ATAM. Ocena wymagań funkcjonalnych.



ARID - ACTIVE REVIEW FOR INTERMEDIATE DESIGNS

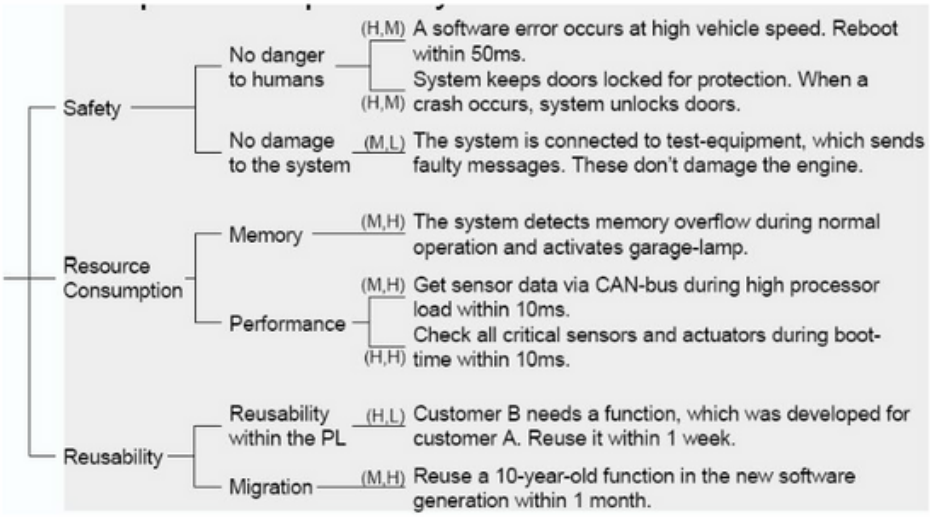
wymagania jakościowe i funkcjonalne

Grupa oceny, obserwatorzy procesu | **Architekt systemu** | **Recenzenci (aktorzy)**



DRZEWO UŻYTECZNOŚCI

Graficzne narzędzie pomagające w identyfikowaniu i hierarchizowaniu wymagań funkcjonalnych i jakościowych systemu, a także w określaniu ich znaczenia dla użytkowników.Składa się z korzenia (początkowego wymagania) oraz gałęzi i liści, które reprezentują kolejne poziomy wymagań.



- H (High) – wysoki priorytet
- M (Medium) – średni priorytet
- L (Low) – niski priorytet
- Oceniamy pod kątem ważności i trudności

JAKIE WYRÓŻNIAMY RODZAJE SCENARIUSZY W PROCESIE OCENY ARCHITEKTURY SYSTEMU?

Metody i techniki związane z ilościową oceną architektury (np. poprzez symulację). Testowanie, jak architektura reaguje w określonych sytuacjach (np. z wykorzystaniem scenariuszy).

Jakie wyróżniamy rodzaje scenariuszy w procesie oceny architektury systemu?

- **scenariusze„zwykłe"**,
- **scenariusze„zmiany”** – wskazują na przewidywane zmiany zachodzące w systemie (np. dodanie nowego typu komunikatów do repertuaru systemu w czasie krótszym niż osobotydzień),
- **migracja systemu operacyjnego** należącego do tej samej rodziny bądź nowej edycji istniejącego systemu operacyjnego w czasie krótszym niż jeden osoborok (pływa na wydajność, bezpieczeństwo i niezawodność),
- **sytuacje stresowe**,
- **scenariusze badawcze** – „bardzo przyszłościowe”,
- **wyznaczenie granic** lub **warunków brzegowych** dla danego projektu.

SPRAWDŹ SIĘ!

Styl architektoniczny określa

- Rozwiązania problemów związanych z konkretnymi technologiami
- Żadna z odpowiedzi nie jest poprawna
- Kolekcje modułów implementujących dane rozwiązane
- Sposób projektowania elementów systemu i ich powiązań

Jaki styl architektury najlepiej zaproponować w sytuacji gdy: „transformacje polegają na inkrementalnym przetwarzaniu danych wejściowych”

- Workflow
- System z podsystemami
- Abstrakcyjny typ danych
- Repozytorium

Decyzja architektoniczna może dotyczyć

- Zastosowania określonego wzorca
- Wszystkie odpowiedzi są prawidłowe
- Wykorzystania konkretnego stylu architektonicznego
- Wykorzystania określonej technologii do implementacji komponentu

Komponent zarządzający sekwencją zdarzeń nazywamy komponentem

- Kontrolerem
- Zarządzającym
- Obliczeniowym
- Pamięciowym

Jakie rozwiązanie najlepiej zaproponować dla następującego problemu „słabopowiązana kolekcja komponentów, z których każdy wykonuje określone operacje i umożliwia wykonania innych. Brak powiązania generujących i odbierających sygnały (korzystne np. przy zamianie dostawców)”

- Każdy komponent zarządza swoimi danymi. Jeśli inny komponent chce ich użyć musi odwołać się do określonej metody, a nie do określonych danych.
- Stworzenie hierarchii komponentów (zależy od siebie), oraz hierarchii danych (danych lokalnych, globalnych)
- System zapewniający przepływ danych pomiędzy komponentami
- Model oparty o zdarzenia

Ocena architektury polega na określeniu

- Niebezpieczeństw dla realizacji systemu zgodnie z wymaganiami
- Możliwości realizacji systemu z wykorzystaniem dostępnych zasobów
- Zgodności systemu z wymaganiami jakościowymi
- Wszystkie odpowiedzi są prawidłowe

Założenia dla decyzji architektonicznej związane są z

- Środowiskiem, w którym podejmowana jest decyzja
- Złożenia dotyczą status decyzji
- Kwestiami architektonicznymi, których dotyczy decyzja
- Ewentualnych powiązań decyzji z innymi decyzjami

Styl architektoniczny określony jest przez następujące elementy (wybierz najpełniejszą odpowiedź)

- Ograniczenia dla systemu wynikające ze środowiska dla tworzonego systemu oraz propozycji rozwiązania ograniczeń
- Sytuację, której dotyczy proponowane rozwiązanie wraz z proponowanym rozwiązaniem
- Problem, ograniczenia dla środowiska i rozwiązanie
- Zbiór decyzji architektonicznych dotyczących komponentów

Poprawnie zaprojektowana architektura pozwala spełnić następujące wymagania pozafunkcjonalne

- Wszystkie odpowiedzi są prawidłowe
- Wydajność
- Użyteczność
- Bezpieczeństwo

Jaki styl architektoniczny najlepiej zaproponować dla następującego problemu : "System może być opisany jako szereg wywołanych po sobie procedur"

- Pośrednie wywołanie
- Abstrakcyjny typ danych
- System z podsystemami
- Repozytorium

Wzorce projektowe dostarczają

- Rozwiązań nieznanych problemów
- Komponentów programowych zbudowanych zgodnie z najlepszymi praktykami
- Kodyfikacji doświadczenia z dziedziny
- Jednoznacznych I precyzyjnych rozwiązań znanych problemów

Baza danych jest przykładem komponentu

- Pamięciowego
- Zarządzającego
- Obliczeniowego
- Kontrolera

Efektem fazy elaboration (uszczegółowienie) w Rational Unified Process jest

- Kod produktu
- Business case
- Architektura systemu
- Kontekst projektu

Elementami drzewa użyteczności w metodzie ATAM są

- Korzeniem jest dowolne wymaganie funkcjonalne , dla którego określane są wymagania jakościowe wraz z wartościami
- Korzeniem jest użyteczność , elementami scenariusze I wymagania jakościowe a liśćmi konkretne wartości
- Korzeniem jest użyteczność , elementami wymagania jakościowe z określonego katalogu wymagań dla metody ATAM , zaś liścmi konkretne ich wartości
- Korzeniem jest użyteczność , elementami dowolne wymagania jakościowe a liścmi konkretne ich wartości

Komponent pobierający i zawracający parametry określonego typu nazywamy komponentem

- Pamięciowym
- Kontrolerem
- Obliczeniowym
- Zarządzającym

Architektura systemu określa

- Wszystkie odpowiedzi są poprawne
- stanowi medium komunikacyjne wyłącznie między projektantami a testerami systemu
- Umożliwia analizę priorytetów nadawanych funkcjonalnościom systemu
- Komponenty systemu i ich powiązania

TWORZENIE OPROGRAMOWANIA

PODSTAWOWE ZASADY OPROGRAMOWANIA

ABSTRAKCJA

Koncentracja na istotnych problemach; abstrakcja od detali.

- **proceduralna** – kroki, w jakich powinien być wykonywany program; wymuszona przez języki programowania; podział problemu na podproblemy.
- **danych** – dane użyte w programie oddające ideę problemu, ich hierarchia i kombinacje w bardziej złożone struktury.

MODULARNOŚĆ

Definicja komponentów i powiązań między komponentami; ułatwia to komunikację między programistami, przeprowadzanie testów, łatwość oceny systemu i daje możliwość ponownego wykorzystania komponentów.

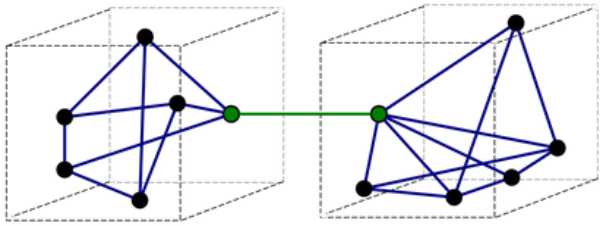
COUPLING

Miara siły powiązań pomiędzy komponentami. Wysoki coupling - silna zależność pomiędzy komponentami, niski coupling - niewielkie zależności.

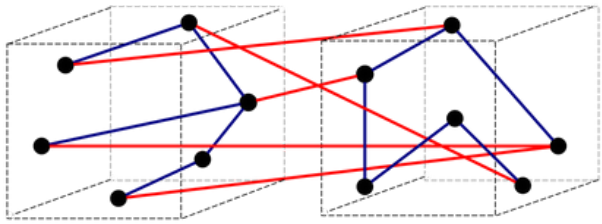
zależność

STOPNIE COUPLINGU

- **zawartości** – jeden komponent swoją działalnością wymusza zmiany na innym komponencie; powinno się go unikać w projektowaniu,
- **zwyczajny** – dwa komponenty posiadają współdzielone dane np. w postaci zmiennych globalnych,
- **zewnętrzny** – komponenty komunikują się przez zewnętrzne medium np. plik z danymi,
- **kontroli** – jeden komponent „dyryguje” wykonaniem zadania przez inny komponent (generując zdarzenia),
- **struktur danych** – kompletna struktura danych jest przekazywana między komponentami,
- **danych** – jedynie proste dane są przekazywane między komponentami.

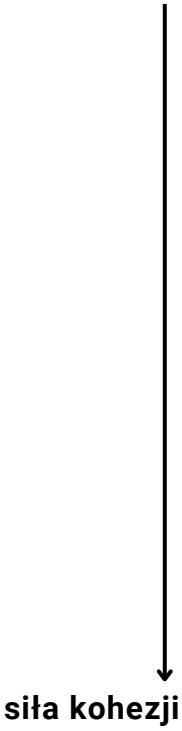


a) Good (loose coupling, high cohesion)



b) Bad (high coupling, low cohesion)

KOHEZJA



Wewnętrzna spójność komponentu.

- **przypadkowa** – pogrupowanie elementów przypadkowo,
- **logiczna** – realizowanie logicznie powiązanych zadań przez elementy komponentu np. komponent realizujący wszystkie operacje wejścia/wyjścia,
- **temporalna** – elementy komponentu są niezależne, lecz aktywowane dokładnie w tym samym czasie,
- **proceduralna** - elementy komponentu wywoływane w określonym porządku,
- **komunikacyjna** – elementy komponentu operują na tych samych (zewnętrznych) danych,
- **sekwencyjna** – komponent składa się z sekwencji elementów, w której wyjście z jednego elementu jest wejściem kolejnego,
- **funkcjonalna** - wszystkie elementy komponentu realizują wspólną funkcję,
- **kohezja danych** – Wszystkie elementy komponentu powiązane są z tym samym, abstrakcyjnym typem danych. Kohezja danych jest silniejsza od kohezji funkcjonalnej.

UKRYWANIE INFORMACJI

Każdy komponent posiada „sekret” niedostępny dla innych komponentów. Widzą one jedynie wynik, ale nie mają wglądu do szczegółów – algorytmu i sposobu. Powiązane z poprzednimi cechami (abstrakcją i modularnością).

KOMPLEKSOWOŚĆ

Kompleksowość dotyczy ilości zasobów niezbędnych do rozwiązania problemu. Atrybut oprogramowania określający ilość zasobów **niezbędnych do stworzenia bądź zmiany** kawałka oprogramowania – rozwiązania problemu np. czasu, ludzi. Przydatny przy wycenie oprogramowania. Jak bardzo nasze oprogramowanie jest kompleksowe/złożone?

Problemy miar kompleksowości

- nie są kontekstowe
- biorą pod uwagę ograniczoną liczbę cech – metoda Halsteada nie zajmuje się np. kompleksowością przepływu danych
- nie ma miary kompleksowej, mierzącej wszystkie istotne aspekty

Metryki

- **wielkościowe** – np. **metoda Halsteada**
- **strukturalne** – struktura wewnętrzna komponentu jako miernik jakości - np. **McCabe’s Cyclomatic Complexity**

Typy kompleksowości

- **wewnętrzne** (dla komponentów)
- **pomiędzy komponentami** (dla systemu rozumianego jako kolekcja komponentów)

METODA HALSTEADA

Wykorzystuje **operatory i operandy zamiast liczenia linijek kodu.**

- Każdy kolejny projekt szacujemy z perspektywy doświadczenia.
- Koszt tworzenia oprogramowania zależy od złożoności kodu.
- Od kompleksowości kodu zależy czas, który przeznaczymy na jego wytworzenie.
- **OPERATOR** – arytmetyczny separator oraz słowa kluczowe.
- **OPERAND** – zmienne i stałe.

```
public static void sort(int x[]) {
    for (int i=0; i<x.length - 1; i++) {
        for (int j = i+1; j<x.length; j++) {
            if (x[i]>x[j]) {
                int save = x[i];
                x[i] = x [j];
                x[j] = save;
            }
        }
    }
}
```

- **n1** – liczba unikalnych operatorów
- **n2** – liczba unikalnych operandów (zmiennych i stałych)
- **N1** – całkowita liczba operatorów
- **N2** – całkowita liczba operandów
- Obszerność programu – **V**
- Poziom programu – **L**
- Złożoność programistyczna – **E**
- Czas programowania (w sekundach) **T=E/18**

Operator	Liczba wystąpień	Operand	Liczba wystąpień
public	1	x	9
sort()	1	length	2
int	4	i	7
...	...	...	...
n1 = 17	N1 = 39	n2 = 7	N2 = 29

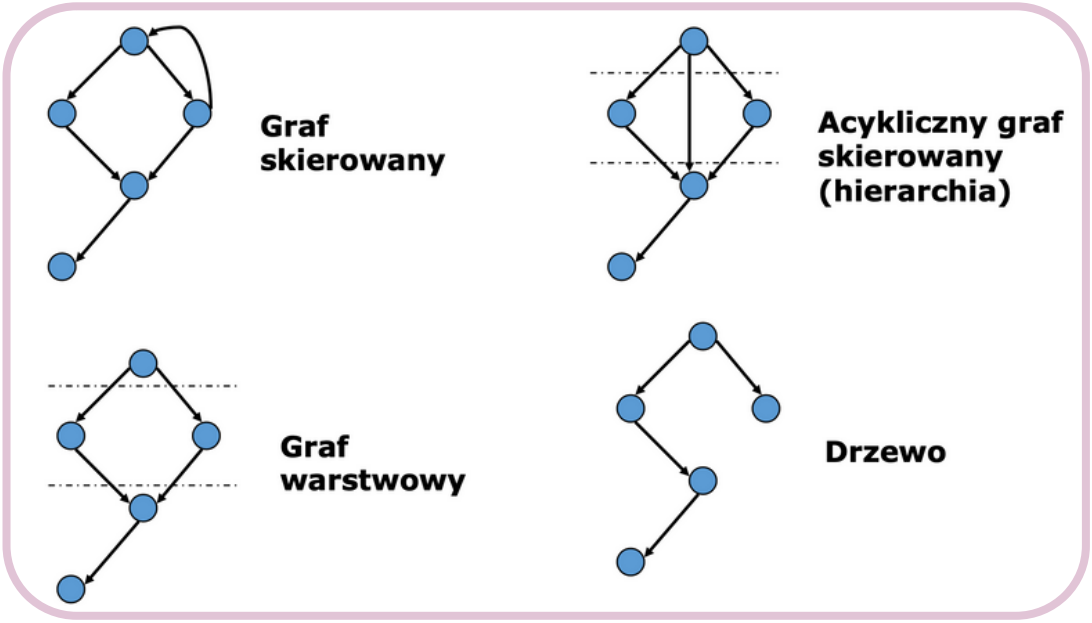
T=1155 sekund = 19 minut

MCCABE'S CYCLOMATIC COMPLEXITY

Miara złożoności kodu źródłowego określająca minimalną liczbę niezależnych ścieżek decyzyjnych w programie (nie będących pętlami). Im wyższa wartość złożoności cyklomatycznej, tym bardziej skomplikowany jest kod. **Liczba punktów decyzyjnych (np. instrukcje warunkowe, pętle) wpływa na zwiększenie tej wartości**. Wyższa złożoność cyklomatyczna sugeruje większą liczbę potrzebnych testów i może wskazywać na trudności w zrozumieniu, utrzymaniu i testowaniu kodu.

STRUKTURA SYSTEMU

GRAFY POWIĄZAŃ



MIARY STRUKTURY SYSTEMU

- **Rozmiar** – ilość węzłów, przejść
- **Głębokość** – długość najdłuższej ścieżki od wierzchołka początkowego do liścia
- **Szerokość** – maksymalna ilość liści na danym poziomie drzewa
- **Miara tree impurity** – odstępstwa grafu od „czystego" drzewa (minimalna liczba wierzchołków, jakie należy usunąć, aby graf stał się drzewem)
- **DIT (depth of class in inheritance tree)** – Określa głębokość drzewa dziedziczenia (liczbę klas, po których dana klasa dziedziczy). W przypadku dziedziczenia wielokrotnego DIT jest najdłuższą drogą do korzenia.
- **NOC (number of children)** – Wyznacza liczbę bezpośrednich potomków danej klasy, określając, ile razy kod z nadklasy został powtórnie wykorzystany w podklasach.
- **CBO (coupling between object classes)** – Mierzy stopień zależności danej klasy do innych klas (które nie są związanie przez dziedziczenie). CBO powinno być < 5 – ścisłe granice między klasami i czytelny sposób ich komunikacji.

PROJEKTOWANIE OPROGRAMOWANIA

DEKOMPOZYCJA FUNKCJONALNA

- **Dana funkcja dekomponowana na podfunkcje rozwiązujące fragmenty problemu. Podstawą dekompozycji jest zbiór wymagań użytkownika.**
- Kryterium podziału/połączenia są np.: czas, dane, kohezja, architektura.

Podejście TOP DOWN
gdy wymagania są dobrze zdefiniowane,
od ogółu do szczegółu, początkowe decyzje projektowe najważniejsze

Podejście BOTTOM UP
gdy wymagania nie są dobrze zdefiniowane,
od szczegółu do ogółu, zaczynamy od dostępnych funkcji szczegółowych i łączymy je z innymi, by adresowały wymagania

POŁĄCZENIE
dwie techniki nie są zbyt dobre oddzielnie, użytkownicy rzadko znają wymagania od początku, a pójście od szczegółu do ogółu może się źle skończyć

MODELOWANIE PRZEPŁYWU DANYCH

- **Dekompozycja funkcjonalna wykorzystująca przepływ danych albo kohezję jako kryterium podziału. Dana funkcja dekomponowana na podfunkcje rozwiązujące fragmenty problemu.**
- Wykorzystywanie diagramów przepływu danych.

SA - Structured Analysis

- **model logiczny, diagramy przepływów danych, analiza struktury**
- Krok 1: Ogólny diagram przepływu danych (kontekst)
- Krok 2: Szczegółowy DPD
- Krok 3: Kolejne szczegóły na DPD
- Uszczegółowienia (dokładne kroki procedury) zwane są minispecyfikacjami
- Struktury danych definiowane w słowniku danych

SD - Structured Design

- **struktura na podstawie powiązań między danymi, krok drugi po SA**
- przejście od DPD powstałego z analizy struktury do projektu (diagramu) struktury, np. techniką JSP

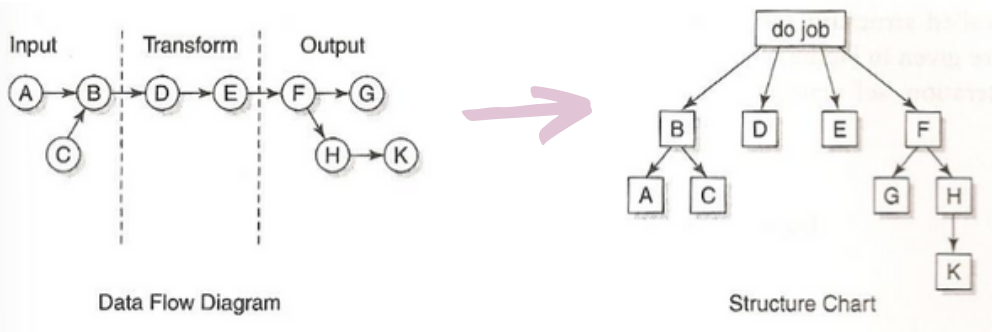
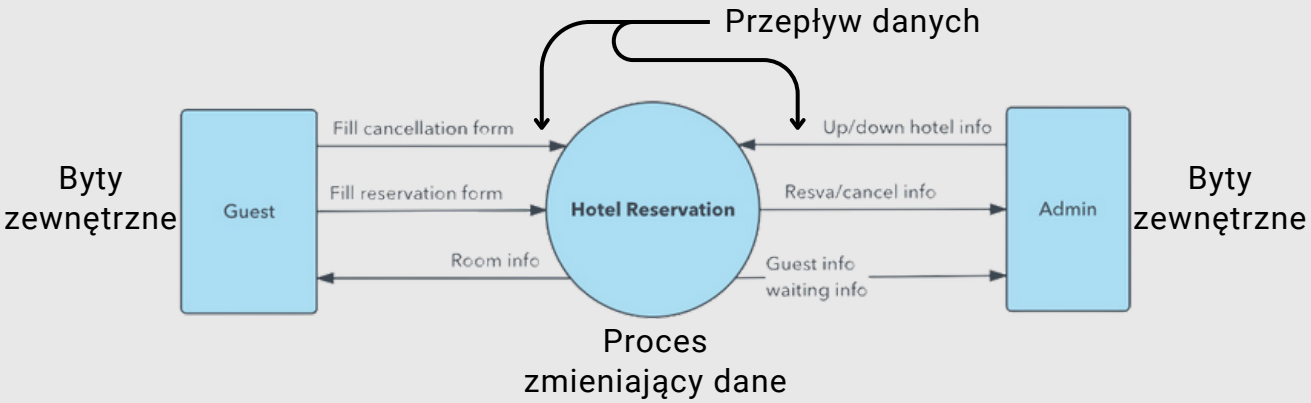
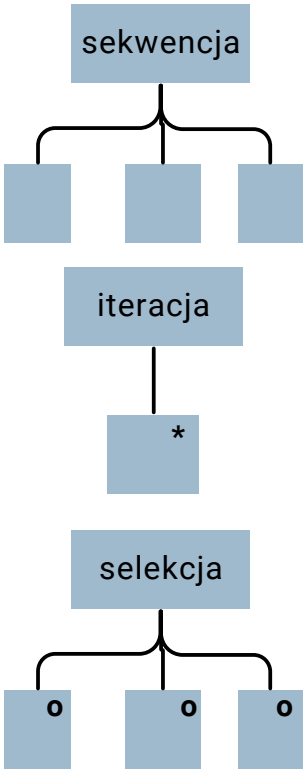


DIAGRAM PRZEPŁYWU DANYCH



PROJEKTOWANIE STRUKTUR DANYCH - JSP

- **Jackson Structured Programming**
- Skoro program składa się z wejść i wyjść to na podstawie dobrego modelu danych można zaprojektować program.
- Zmierza do ustalenia struktury programu na podstawie struktur danych.
- **Komponenty elementarne** – "atomowe", nie podlegają dalszej dekompozycji
- **Komponenty złożone** – sekwencje, interakcje, selekcje, reprezentowane przez diagramy struktur

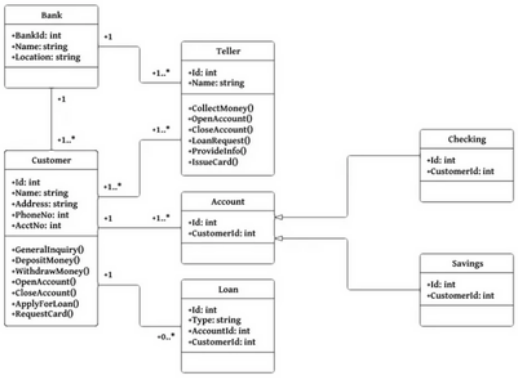


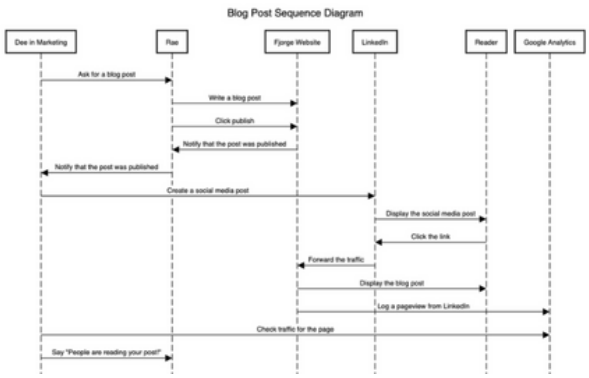
KROKI

1. Modelowanie danych wejściowych i wyjściowych.
2. Łączenie ze sobą diagramów dla danych wejściowych i wyjściowych w celu stworzenia struktury programu.
3. Eliminacja ewentualnych problemów (konfliktów, duplikatów, itp.).
4. Optymalizacja programu.

OBJECT-ORIENTED ANALYSIS AND DESIGN

- **Analiza i projektowanie zorientowane na obiekty - programowanie obiektowe.**
- Tworzenie wizji systemu w oparciu o diagram klas i diagram sekwencji - jakie obiekty są, jakie parametry przechowują i jak się komunikują.





METODA BOOCHA

Skupia się głównie na modelowaniu obiektowym, analizie wymagań, projektowaniu i implementacji systemów. Koncentruje się na identyfikowaniu obiektów, ich atrybutów, zachowań i relacji między nimi.

FUSION HP

Skupia się na wysokiej wydajności systemów - analizie, projektowaniu i implementacji systemów, które mają zapewnić maksymalną wydajność i efektywność.

faza analizy - perspektywa wejścia i wyjścia użytkownika
definiowanie obiektów i interakcji między nimi, tworzenie interfejsów na tej podstawie

faza projektowania - perspektywa wejścia i wyjścia użytkownika
komunikacja między obiektami i połączeń między nimi, sposób realizowanej komunikacji (visibility)

WNIOSKI - RÓŻNICE

Główną różnicą między metodą Boocha a Fusion HP jest ich główne skupienie. Metoda Boocha skupia się na ogólnym modelowaniu, analizie wymagań i projektowaniu systemów, podczas gdy Fusion HP skupia się na osiąganiu wysokiej wydajności i efektywności w implementowanych systemach

UTRZYMANIE OPROGRAMOWANIA

Po co nam utrzymanie oprogramowania?
Do eliminacji błędów i poprawy wydajności, dostosowywania systemu do zmieniającego się środowiska i wymagań jakościowych.

KROKI DO UNIKNĘCIA PROBLEMÓW Z UTRZYMANIEM

1. **DOKUMENTACJA** - ułatwia odnalezienie się innym, zrozumienie kodu po czasie.
2. **PRZEWIDYWANIE ZMIAN** - już na etapie analizy, jeszcze przed projektowaniem
3. **KOMPLETNA ANALIZA WYMAGAŃ** - dostosowywanie oprogramowania do użytkownika, żeby mu się podobało.
4. **ZARZĄDZANIE KODEM** - wykorzystywanie tych samych komponentów przez różne systemy zmniejsza problem utrzymania. Nie trzeba wszystkiego pisać na nowo.

RODZAJE UTRZYMANIA

Korygujące naprawa błędów, np. nie da się zapisać na przedmiot w usosie.

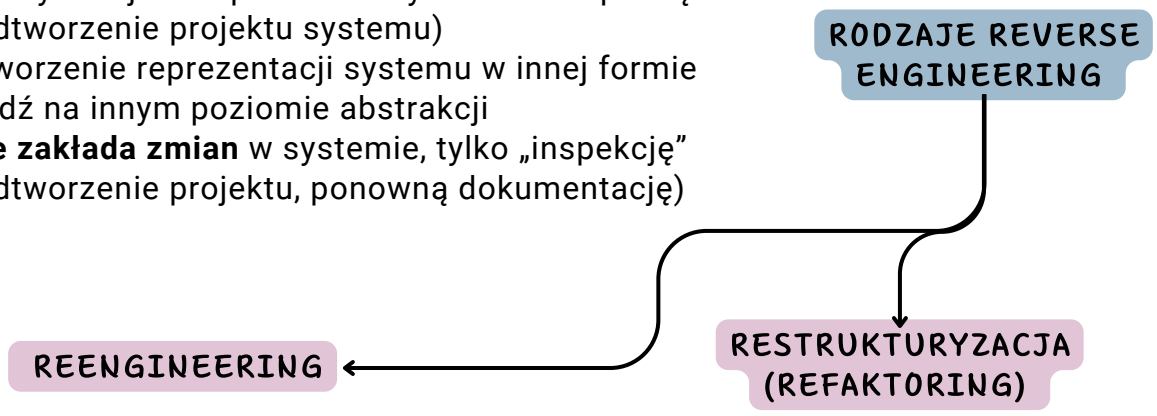
Adaptatywne adaptowanie systemu do zmian otoczenia, np. chcemy korzystać z ususa na telefonie.

Ulepszające nadążające za nowymi wymaganiami, np. reels na instagramie.

Prewencyje ułatwiające potomnym – przyszłe utrzymanie systemu, aktualizacja dokumentacji, lepsze komentarze czy struktury.

REVERSE ENGINEERING

- identyfikacja komponentów systemu i ich powiązań (odtworzenie projektu systemu)
- stworzenie reprezentacji systemu w innej formie bądź na innym poziomie abstrakcji
- **nie zakłada zmian** w systemie, tylko „inspekcję” (odtworzenie projektu, ponowną dokumentację)



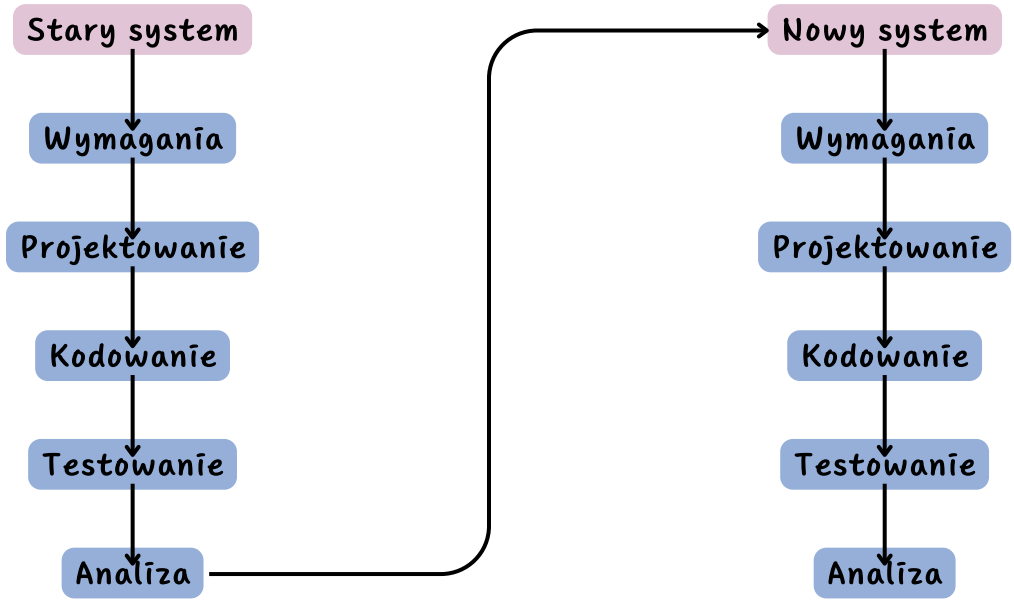
- Prawdziwe zmiany w systemie
- Przeprojektowanie systemu
- Nowych funkcjonalności oprócz wypielęgnowania kodu

- Poprawianie efektywności (zmiana lokalna)
- Zmiana kodu bez zmiany funkcjonalności
- Pielęgnowanie kodu, stworzenie struktury kodu, sprawdzenie zmian w kodzie, zmiana sposobu reprezentacji.

METODYKA WPROWADZANIA ZMIAN

KLASYCZNY SPOSÓB WPROWADZANIA ZMIAN

Tworzenie nowej wersji oprogramowania z Change request.



Change request to dokument wnioskujący o wprowadzenie zmiany w systemie.

QUICK-FIX

- naprawianie oprogramowania, błędu w systemie
- najpierw kod – potem dokumentacja i reszta ferajny

SPRAWDŹ SIĘ!

Kohezja temporalna określna sytuację, w której elementy w komponencie powinny być wywoływane w określonym porządku

- Prawda
- Fałsz

Kohezja funkcjonalna określa sytuację, w której wszystkie elementy komponentu realizują wspólną funkcję

- Prawda
- Fałsz

Coupling struktury danych oznacza sytuację, w której komponenty komunikują się przez zewnętrzne medium np. plik z danymi

- Prawda
- Fałsz

Elementem diagramu przepływu danych nie jest

- Przepływ danych
- Byt zewnętrzny
- Proces przekształcenia danych
- Element struktury systemu (komponent)

W metodzie Halsteada przykładem operatora nie jest

- J
- Sort()
- Public
- Int

Najsilniejsze powiązanie między komponentami określa sytuację, w której jeden komponent swoją działalnością wymusza zmiany na innym komponencie

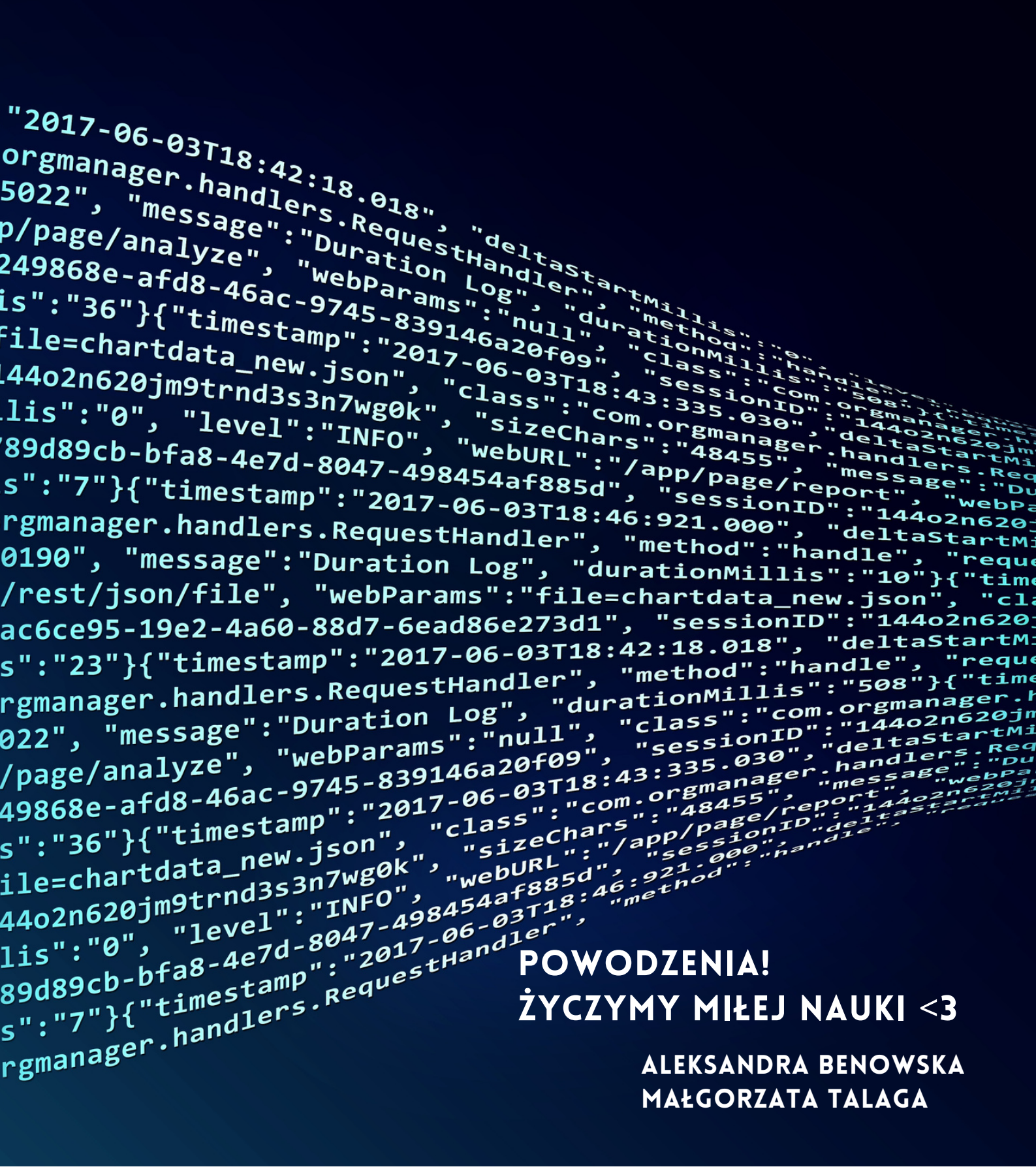
- Prawda
- Fałsz

Metoda Halsteada służy do oceny kompleksowości oprogramowania . Jej celem jest ustalenie wielkości zespołu projektowego jak powinien realizować zadanie

- Prawda
- Fałsz

Założeniem leżącym u podstaw Jackson Structured Programming jest, że dobry program odzwierciedla strukturę wejść i wyjść (czyli na podstawie dobranego modelu danych można stworzyć odpowiedni program)

- PRAWDA
- FAŁSZ



POWODZENIA!
ŻYCZYMY MIŁEJ NAUKI <3

ALEKSANDRA BENOWSKA
MAŁGORZATA TALAGA

WSZYSTKIE ŹRÓDŁA PODLINKOWANE
W MIEJSCACH UŻYCIA + PREZENTACJE
PANI DR AGATY FILIPOWSKIEJ :)

KANAŁ
STUDENTCKI