‘Wykaz tematów do egzaminu

### Spis treści

[T1](#id.f4qcwvqj4x4x) - Done

[T2](#id.9k2bk8k7y492) - Modele wytwarzania - Done

[T3](#id.jh3nxiy95nfj) - Faza strategiczna - Done

[T4](#id.k654fi2qpwj4) - Modelowanie wymagań - Brak: 5,6,8,9,10,

[T5](#id.9qkwwxdvh9a2) - Inżynieria wymagań - Brak: 6,

[T6](#id.qufwgbtju2n) - Wzorce projektowe - Done

[T7](#id.shpiiano0sm8) - Zarządzanie konfiguracją - Done

[T8](#id.co8hozb1sn0r) - Testy - Brak: 2,3,9,10,11,12

[T9](#id.m3b52uuwjy4v) - Metryka, pomiar, szacowanie kosztów - Brak: 3,4, 5,9,10

[T10](#id.372g7dnjwbli) - Bazy danych - Done~!

### T1[↑](#id.5nzygkfqi063)

1. **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.
Jest również wiedzą empiryczną, syntezą doświadczeń tysięcy ośrodków zajmujących się budową oprogramowania. Nie działa tu stwierdzenie "od teorii do praktyki".

Dobre oprogramowanie powinno być zgodne z wymaganiami, niezawodne, efektywne, łatwe w konserwacji, ergonomiczne.
Produkcja oprogramowania jest procesem skłądającym się z wielu faz. Programowanie jest tylko jedną z nich.
2. **Zagadnienia inżynierii oprogramowania**
	1. sposoby prowadzenia przedsięwzięć informatycznych
	2. techniki planowania, szacowania kosztów, harmonogramowania, monitorowania
	3. metody analizy i projektowania
	4. techniki zwiększania niezawodności oprogramowania
	5. sposoby testowania, szacowania niezawodności
	6. sposoby przygotowania dokumentacji technicznej i użytkowej
	7. procedury kontroli jakości
	8. metody redukcji kosztów konserwacji (utrzymania systemu)
	9. techniki pracy zespłowej, czynniki psychologiczne wpływające na efektywność
3. **Źródła złożoności projektu oprogramowania**
	1. zróżnicowanie dziedzin problemowych
	2. zróżnicowanie zespołu projektantów + trudności w komunikacji
	3. ogromna dynamika technologii, środków technicznych
	4. ciągła zmienność wymagań i niezdecydowanie potencjalnych userów
4. **Metody walki ze złożonością**
	1. zasada dekompozycji -
	podział problemu na niezależne podproblemy
	2. zasada abstrakcji -
	eliminacja/ukrycie/pominięcie szczegółów problemu; wyodrębnienie cech wspólnych i stałych, wprowadzenie pojęć by je definiować.
	3. zasada ponownego użycia -
	używanie tego co już mamy, wzorców i metod dobrze znanych i sprawdzonych
	4. zasada sprzyjania ludzkim własnościom
5. **Modelowanie pojęciowe**

Projektant i programista muszą dokładnie wyobrazić sobie problem oraz metodę jego rozwiązania - dąży się do tego, by luka między myśleniem o rzeczywistym problemie a myśleniem o danych i procesach w systemie była jak najmniejsza.

### T2[↑](#id.5nzygkfqi063)

1. **Co to jest cykl życiowy oprogramowania**

Zbiór faz składających się na życie oprogramowania. Modele cykli życia oprogramowania pozwalają uporządkować przebieg prac, ułątawiając planowanie zadań oraz monitorowanie przebiegu ich realizacji.
Fazy to np.:

* 1. Faza strategiczna
	2. Określenie wymagań
	3. Analiza
	4. Projektowanie
	5. Implementacja
	6. Testowanie
	7. Instalacja
	8. Konserwacja
	9. Dokumentacja
1. **Modele cyklów życiowych (wodospadowy, spiralny, przyrostowy, prototypowy, code-and-fix, sterowany dokumentami), charakterystyka, zalety i wady, zastosowania**
	1. wodospadowyj
		1. charakterystyka -
		wszystkie fazy jedna po drugiej. Istnieją pewne modyfikacje - np. cofanie się o max 1 faze wstecz (by wprowadzać poprawki), lub możliwosc cofnięcia się gdziekolwiek. Powroty to jednak sytuacje wyjątkowe
		2. zalety

- jeżeli nic się nie zmienia, to wychodzi dobrze

- łatwość ustalenia harmonogramu i jego monitorowania

* + 1. wady

- narzuca scisłą kolejność wykonywania prac

- wysoki koszt błędów z wczesnych faz

- długa przerwa w kontaktach z klientem

* + 1. zastosowania
		tam gdzie raczej na pewno nic się nie będzie zmieniać, bądź zmiany są bardzo kosztowne/niemożliwe (co by wszystko przewidywali z góry np. programy kosmiczne)
	1. spiralny
		1. charakterystyka
		składa się z 4 głównych faz wykonywanych cyklicznie (**planowanie, analiza, konstrukcja, atestowanie**). Rozważamy opcje i cele budowy nowej wersji systemu. Analizujemy je pod względem ryzyka ich realizacji. Konstruujemy kolejna wersję systemu, zgodnie z modelem kaskadowym. Klient ocenia nasze dzielo. Zapętlić
		2. zalety
		- minimalizacja ryzyka związanego z produkcją oprogramowania (m.in. przez częste kontakty z klientem)
		3. wady
		- wymaga większych nakładów organizacyjnych (nie koniecznie wada)
		- może prowadzić do przykładania mniejszej wagi do identyfikacji wymagań czy budowy modeli ( bo zawsze można poprawić w następnycm cyklu spirali)
		4. zastosowania
		Używane do firm wytwarzających kolejne rynkowe wersje oprogramowania - atestowanie odpowiada eksploatacji softu przez użytkowników.
	2. przyrostowy
		1. charakterystyka
		Bardzo podobne do spiralnego. Polega na definiowaniu zestawu funkcjonalności do realizacji w kolejnych cyklach. Po zrealizowaniu “dostarczamy” częściowo gotowy soft, i dobieramy kolejne funkcjonalności.
	3. prototypowy
		1. charakterystyka

Ma minimalizować ryzyko związane z niewłaściwym określeniem wymagań. Głównymi celami prototypu jest wykrycie nieporozumień, brakujących funkcji, trudnych usług, braków w specyfikacji.

Fazy ~:
- wstępne określenie wymagań

- prototyp + akceptacja

- pełne określenie wymagań + kaskada

* + 1. zalety

- możliwość szybkiej demonstracji pracującej wersji systemu

- możliwość szkoleń znaim zbudowany zostanie pełen system

* + 1. wady
		- dodatkowy koszt budowania prototypu/ów
		- klient może być zdziwiony, który musi długo czekać i sporo płacić za coś, co widział w ‘działającym’ stanie (i to po tak krótkim czasie)
		2. zastosowania
		- tworzenie systemów automatyzujących prace, która do tej pory wykonywana była bez komputerów.

- tam, gdzie klient ma problem z określeniem wymagań, więc spojrzenie na prototyp może mu pomóc

* 1. code-and-fix
		1. charakterystyka
		Weź to i zrób to. Jest plan - super. Nie ma planu - super. Programujemy! Jak nie wyjdzie to trudno.
		2. zalety
		- fun
		- szybko
		3. wady

- jeśli robimy coś większego to masakra

* 1. sterowany dokumentami
		1. charakterystyka
		Ścisła realizacja modelu kaskadowego. Dodatkowo, każda faza kończy się opracowaniem szeregu dokumentów, w pełni opisujących wyniki tej fazy. Dokumenty powinny być wystarczającą podstawą do realizacji dalszych faz - są prezentowane klientowi do akceptacji. (bez niej nie przechodzimy dalej)
		2. zalety
		- jak kasakodwy, łatwo planować, harmonogramować i monitorować przedsięwzięcie.
		- teoretycznie, można przerwać w połowie i dać komuś innemu do kontynuacji za np. rok
		3. wady
		- jak kaskadowy
		- ogromny nakład pracy nad dokumentacją
		4. zastosowania
		używała/używa tego amerykańska armia
	2. formalnych transformacji
		1. charakterystyka
		wymagania formułowane są w pełni formalnym języku, poddawane transformacjom, aż do uzyskania działającego kodu. Transformacje odbywają się bez udziału ludzi.
		2. zalety

- brak

* + 1. wady

 - nie działa

* + 1. zastosowania
		same metody matematyczne mogą jedynie wpomagać pewne szczegółowe tematy, np. obliczanie pewnych mierzalnych charakterystyk oprogramowania
1. Dobór cyklu życiowego do projektu
Musimy ocenić co dokładnie jest nam potrzebne. Rózne modele cykli życiowych sprawują się różnie w różnych sytuacjach.


### T3[↑](#id.5nzygkfqi063)

1. **Czym jest faza strategiczna**

Określenie startegicznych celów, planowanie i definicja projektu.
2. **Umiejscowienie fazy strategicznej w cyklu życiowym**Wykonywana zanim podjęta zostanie decyzja o realizacji przedsięwzięcia. Nazwana również strategicznym planem rozwoju informatyzacji (SPRI) lub studium osiągalności
3. **Działania podejmowane w fazie strategicznej, czynniki sukcesu i rezultaty**
	1. działania
		1. dokonanie serii wywiadów z przedstawicielami klienta
		2. określenie celów przedsięwzięcia z puktu widzenia klienta
		3. określenie zakresu oraz kontektsu
		4. ogólne określenie wymagań, wykonanie zgrubej analizy i projektu systemu
		5. propozycja kilku możliwych rozwiązań (sposobów realizacji)
		6. oszacowanie kosztów
		7. analiza rozwiązań
		8. prezentacja wyników fazy strategicznej oraz korekty
		9. określenie wstępnego harmonogramu oraz struktury zespołu
		10. określenie standardów, których będziemy się trzymać przy realizacji
	2. czynniki sukcesu
		1. szybkość pracy -
		opóźnienie w przeprowadzeniu fazy strategicznej może prowadzić do utraty klienta
		2. zaangażowanie kluczowych osób ze strony klienta -
		brak akceptacji dla sposobu realizacji od kluczowych osob może uniemożliwić wykonanie/sukces projektu.
		3. uchywcenie ogólne całości systemu -
		błędem jest zbytnie czepianie się szczegółów. Musimy uchwycić całość systemu, by móc dobrze oszacować koszty/stopień skomplikowania wykonania całości.
	3. rezultaty
		1. raport, obejmujący (definicje celów przedsięwzięcia, opis zakresu, opis systemów zewnętrznych, ogólny opis wymagań, ogólny model systemu, opis proponowanego rozwiązania, oszacowanie kosztów, wstępny harmonogram prac)
		2. raport oceny rozwiązań (info o rozważanych rozwiązaniach + przyczynach wyboru)
		3. opis wymaganych zasobów - pracownicy, soft, sprzęt, lokum itp
		4. definicje standardów
		5. harmonogram fazy analizy
4. **Zależność Użytkownik – Zleceniodawca**
Zleceniodawca płaci za projekt, jednak to użytkownicy będą mieli kontakt z naszym tworem. Przy zbieraniu celów przedsięwzięcia ważne jest zebranie ich od użytkowników, a nie tylko od zleceniodawcy.
5. **Czym jest zakres projektu, umiejętność definiowania celu i zakresu przedsięwzięcia informatycznego**
	1. zakres przedsięwzięcia -
	określenie fragmentu procesów zachodzących w organizacji, które będą objęte przedsięwzięciem. Na etapie fazy strategicznej może nie być jasne co dokładnie ma robić system, a co ma robić personel/inne systemy/sprzęt.
	2. definiowanie celu -
6. **Zawartość studium osiągalności**decyzje, od których zależy przebieg i powodzenie projektu + ważne pytania
Decyzje :

- wybór modelu cyklu życia

- wybór technik analizy i projektowania

- wybór środowisk implementacji

- wybór narzędzia CASE

- określenie wykorzystywania ponownego już gotowych rzeczy

- o współprac z innymi producentami/ekspertami

Ograniczenia:

- nakłady na realizacje

- dostępność personelu + narzędzi

- czasowe

Pytania w studium osiągalności

- jak duży będzie projekt?

- ile czego i czy będziemy mieć? (kasa, ludzie, kadra, lokum, komunikacja, zaopatrzenie)

- jakie sa dedlajny

- warunki niezbędne do realizacji (rozpoczęcia)

- dostępność narzedzi/oprogramowania do rozwoju tego, co mamy zrobić

- jak jest ogólnie ze sprzętem/siecią?

- dostępność know-how i technologii, specjalistów wewnętrznych i zewnętrznych

- dostępnośćusług zewnętrznych, kooperatorów, dostawców

1. **Metody oceny rozwiązań w studium osiągalności, drzewa ryzyka i ich zastosowanie**By wybrać najlepsze z kilku rozwiązań, zazwyczaj porównujemy parę czynników (zwykle przydzielając im odpowiednie wagi) :

- koszt

- czas realizacji

- możliosc ponownego użycia

- przenośność na inne platformy

- wydajność

**Drzewa ryzyka**

wierzchołki odpowiadają sytuacjom, w któryc mogą zajść pewne zdarzenia.

krawędzie to przejścia między sytuacjami - przypisujemy im prawdopodobieństwa.

każdy liść odpowiada za osobny scenariusz zdarzeń i jest związany z kosztem.



1. **Techniki szacowania kosztu oprogramowania**
	1. Modele algorytmiczne
	opis przedsięwzięcia liczbami, toną liczb i/lub opisów + przeliczenie tego algorytmem/formułą matematyczną
		1. metoda COCOMO - na podstawie linii kodu
		2. metoda punktów funkcyjnych -
		oceniamy na podstawie wymaganych od projektu funkcji, ilości wyjść i wejść do systemu. Ogólnie dość dobra metoda, mało wad, daje niezłe pojęcie o koszcie.
	2. Ocena przez eksperta
	ktoś, kto robił już coś podobnego przychodzi i mówi
	3. Ocena przez analogie
	jeżeli było robione coś po dobnego, sprawdzamy co było podobne, i przeliczamy
	4. Wycena dla wygranej
	nasz koszt ma zawierać się między wyceną klienta a wyceną konkurencji
	5. Szacowanie wstepujące
	podział zadania na mniejsze, ocena kosztu mniejszych

### T4[↑](#id.5nzygkfqi063)

1. **Czym jest model, cechy modelu**
**Model** to uproszczenie rzeczywistości. Wszystkie modele systemu muszą być ze sobą niesprzeczne oraz spójne. Ich celem jest przedsawienie różnych aspektów systemu. Mamy na przykład model analityczny, projektowy, statyki, dynamiki systemu
2. **Zawartość i obszar modelu analitycznego**
Zawiera sposób realizacji przez system postawionych wymagań, lecz abstrahuje od szczegółów implementacyjnych. Jest bardziej ogólny od zakresu odpowiedzialności systemu. Ujmuję również użycie systemów zewnętrznych, albo decyduje co z zakresu odpowiedzialności będzie realizowane przez zewnętrzny system.
3. **Miejsce budowy modelu analitycznego w cyklu życiowym**
W fazie analizy, po zbieraniu wymagań.
4. **Czynności w fazie analizy, rezultaty fazy analizy**
	1. czynności
	- rozpoznanie, wyjaśnianie, modelowanie, specyfikowanie i dokumentowanie rzeczywistości/problemu będącego przedmiotem projektu
	- ustalenie kontekstu projektu
	- uszczegółowienie wymagań użytkowników
	- ustalenie lub szczegółowienie wymagań organizacyjnych
	- dokonanie innych ustalen (kasa, czas, preferencje softu, sprzętu)
	2. wyniki
	- statyczny model klas
	- zanalizowane funkcje i przypadki użycia
	- zweryfikowane klasy i obiekty
	- zidentyfikowane i zdefiniowane metody oraz komunikaty
	- zamodelowane stany, przejścia
	- zamodelowane procesy, przepływ danych, przepływ sterowania
5. **Transformacja wymagań użytkownika w wymagania na oprogramowanie**
6. **Pożądane i rzeczywiste cechy wymagań**
7. **Notacje stosowane w fazie analizy**
Dzięki notacjom analityk zapisuje i dokonuje analizy modelu, oraz może komunikować się z innymi członkami zespołu projektowego i klientem. Mają być jasne i przejrzyste, proste i w ogóle.
	1. język naturalny
	2. notacje graficzne
	3. specyfikacje - ustrukturalizowany zapis tekstowy i numeryczny
8. **Metoda dekompozycji funkcjonalnej – zasada działania i zastosowania**
9. **Pojęcie spójności i powiązania**
10. **Podejście obiektowe do zmienności wymagań**
11. **Struktura normy IEEE 830-1993 – Dokument wymagań na oprogramowanie**


### T5[↑](#id.5nzygkfqi063)

1. **Metody pozyskiwania (rozpoznawania) wymagań**
	1. wywiady i przeglądy
		1. listy pytań pokrywające całość zagadnienia
		2. powinny doprowadzić do szerokiej zgodności i akcpetacji projektu
	2. analiza starego oprogramowana, które ma zostać zastąpione przez nasz program
	3. analiza wymagań systemowych
		1. gdy nasz program ma być częścią większego systemu
	4. studia osiągalności
		1. określenie realistycznych celów i metod ich osiągniecia
	5. prototypowanie
2. **Technika map myśli, cechy map myśli, zastosowania**
	1. autor: Tony Buzan
	2. taki pająk, dużo kolorów i obrazków - podobno mózg się mniej nudzi
	3. hasła zamiast zdań
	brzydkie i chaotyczne
	4. technika wspomaga kreatywne myślenie, więc pewnie można wykorzystać przy zbieraniu wymagań (i masie innych rzeczy)
3. **Trudności w pozyskiwaniu i dokumentowaniu wymagań – źródła i metody przeciwdziałania**
	1. **trudności w pozyskaniu:**
		1. Klient ma cel ale nie wie jak go osiągnąć, a cele mogą być osiągnięte na wiele sposobów
		2. Cele różnych użytkowników mogą być sprzeczne
		3. Użytkownicy stosują różną terminologię
		4. często zleceniodawcy != użytkownicy, zleceniodawca jest decydentem, który może źle przewidzieć potrzeby użytkowników
	2. **przeciwdziałanie**
		1. użycie checklisty (jakie elementy powinny być zawarte we wzorcowym wymaganiu)
		2. konsultacje trójstronne (zleceniodawca - użytkownik - dostawca)
	3. **trudności w dokumentowaniu:**
		1. język naturalny jest niejednoznaczny
		2. niejednorodna terminologia (język informatyka a businessmana)
		3. trudności w uchwyceniu zależności między wymaganiami
	4. **przeciwdziałanie**
		1. porządkowanie wymagań - hierarchia w postaci drzewka
			1. top-down - podział ogólnych funkcji na prostsze
			2. bottom-up - kilka elementarnych funkcji składanych jest w jedną bardziej ogólną
		2. template’y dokumentów
		3. stosowanie diagramów blokowych, UC i.in.
4. **Poziomy abstrakcji opisu wymagań**

 a. Definicja wymagań - opis w języku naturalnym, wynik wstępnych konsultacji z klientem

 b. Specyfikacja wymagań - częściowo ustrukturalizowany zapis - wymieszany język naturalny i proste sformalizowane notacje

 c. Specyfikacja oprogramowania - jednoznaczny formalny opis wymgań, które są dokładnie zdekomponowane na punkty - podstawa do testowania

1. **Jakość opisu wymagań**

Dobry opis wymagań charakterzuje się:

 a. jest kompletny oraz niesprzeczny

 b. opisywać zewnętrzne zachowanie sie systemu, a nie sposów realizacji

 c. obejmować ograniczenia przy jakich musi pracować system

 d. być łątwy w modyfikacji

 e. brac pod uwagę przyszłe możliwe zmiany

 f. opisywać zachowanie systemu w skrajnych sytuacjach.

1. **Etapy i transformacje w analizie wymagań**
2. **Metody dokumentowania wymagań**
	1. język naturalny
	2. język naturalny strukturalny
	3. tablice, formularze
	4. diagramy blokowe
	5. diagramy kontekstowe
	6. diagramy przypadków użycia
	7. formalizm matematyczny
3. **Metoda przypadków użycia – czym jest UC i jego składniki**

- forma opisu funkcji systemu z poziomu jego użytkowników

 - modeluje funkcjonalności grupy użytkowników (grupy aktoró) a nie konkretnego aktora

 - przedstawiają w poglądowy sposób aktorów i funkcje systemu oraz powiązania międy nimi.

1. **Cykl życiowy UC**

Identyfikacja -> krótki opis -> uszczegółowienie -> kompletny opis

1. **Metody dokumentowania przypadków użycia**
	1. diagramy UML
	2. hierarchia funkcji
2. **Metoda aktor – cel**
	1. polega na tworzeniu listy, w której każdemu aktorowi przyporządkowuje się cele biznesowe, jakie może on zrealizować poprzez system. Może być prioretyzowana. Jest stosowana w fazie wczesnej analizy.
3. **Poziomy celów użytkownika – klasyfikacja wg Cockborna**
	1. cel strategiczny (fakturowanie)
	2. cel użytkownika (utwórz faktrę, wyślij fakturę)
	3. podfunkcje (rozpoznaj produkt, rozpoznaj klienta)
4. **Hierarchiczne porządkowanie wymagań**

Jest to rozbijanie wymagania, funkcji na podfunkcje. Np:

 1. Funkcja 1

 1.1 podfunkcja 1

 1.2 podfunkcja 2

 a. Na kazdym poziomie ma być ten sam poziom szczegółowości

 b. kolejność funkcji nie ma znaczenia

 c. wyszczegóławiać funkcje widoczne tylko dla użytkownika

1. **Wymagania funkcjonalne a niefunkcjonalne**

Wymagania funkcjonalne opisuje funkcje wykonywane przez system lub przez system zewnętrzny.

 Wymagania niefunkcjonalne dotyczą środowiska, ograniczeń przy których system będzie realizował swoje funkcje.

### T6[↑](#id.5nzygkfqi063)

1. **Pojęcie wzorca projektowego**

**Wzorzec projektowy** opisuje problem, który stale się powtarza, określa zadadniczą część jego rozwiązania tak, aby można było je zastosować wiele razy za każdym razem w nieco inny sposób.

1. **Geneza wzorców projektowych**

Za Twórcę wzorców projektowych uważany jest **Christopher Alexander**. Wzorce projektowe dotyczyły rozwiązań architektonicznych.

1. **Klasyfikacja wzorców projektowych**

Znane są wzorce projektowe ogólnego przeznaczenia, jak również specjalistyczne wzorce projektowe(specyficzne dla różnych dziedzin zastosowań).

Klasyfikacja ogólnych wzorców projektowych:

* 1. kreacyjne (np. Fabryka Abstrakcyjna)
	2. strukturalne (np.Fasada, Adapter, Most)
	3. czynnościowe
1. **Metody opisu wzorca – szablon opisu (nazwa, intencja, problem, rozwiązanie, uczestnicy, konsekwencje)**
2. **Znajomość wzorców: Adapter, Fasada, Most, Fabryka abstrakcyjna, Strategia**
	1. **Nazwa:** Adapter

 **Intencja**: dopasowanie istniejącego obiektu do określonego interfejsu.

 **Problem:** obiekt zachowuje się tak jak trzeba, ale ma nieodpowiedni interfejs

 **Rozwiązanie:** obudowanie obiektu pożądanym interejsem

**Uczestnicy:** Adapter, Adaptowany, Cel, Użytkownik

**Konsekwencje:** dopasowanie istniejących obiektów do tworzonych struktur, uniknięcie ograniczeń



* 1. **Nazwa:** Fasada

**Intencja:** Uproszczony sposób korzystania z istniejącego systemu.

**Problem:** Potrzeba wykorzystania części możliwości istniejącego systemu

**Rozwiązania:** Nowy interfejs do istniejącego systemu

**Konsekwencje:** Upraszcza korzystanie z systemu

**Uczestnicy:** Uzytkownik, implementator innych czesci systemu



* 1. **Nazwa:** Most

**Intencja:** usunięcie powiązań między zbiorem implementacji, a zbiorem korzystających z nich obiektów.

**Problem:** Klasy pochodne klasy abstrakcyjnej muszą używać wielu róznych implementacji tak, aby nie spowodowalo to nadmiernego wzrostu liczby wyspecjalizowanych klas

**Rozwiązania:** zdefiniowanie wspólnego interfejsu dla wszystkich implementacji i wykorzytsanie go przez klasy pochodne klasy abstrakcyjnej

**Konsekwencje:** usunięcie powiązań między różnymi implementacjami, a obiektami, które je wykorzystują, ułatwia rozbudowę, gdyż obiekty nie posiadają wiedzy na temat swojej implementacji.

**Uczestnicy:** Abstrakcja, Implementator



* 1. **Nazwa:** Fabryka abstrakcyjna

**Intencja:** Dostarczanie interfejsu do tworzenia rodzin powiązanych ze sobą obiektów bez konieczności specyfikowanie ich klas konkretnych

**Problem:** Chcemy dostarczać interfejs do tworzenia całych rodzin spokrewnionych ze sobą, lub zależnych od siebie okiektów bez konieczności definiowania ich klas konkretnych.

**Rozwiązania:** ?

**Konsekwencje:** Ukrywane są szczegóły implementacyjne, klient widzi tylko interfejs

**Uczestnicy:** Fabryka abstakcyjna, Fabryki konkretne, Produkty abstakcyjne, Produkty konkretne



* 1. **Nazwa:** StrategiaSorter zleca operację wybranej strategii sortowania. Każda strategia to jeden algorytm. Zmiana strategii nie wpływa na obiekt Sorter.

Drugi przykład dotyczy wzorca Strategy. Klasa Sorter wykonuje sortowanie wewnętrznej kolekcji. Ponieważ istnieją różne algorytmy sortowania, dlatego realizacja metody sort() jest delegowana do aktywnego algorytmu, stanowiącego implementację klasy SortingStrategy. Metody tej klasy to kroki algorytmu. Każdy algorytm może realizować je w charakterystyczny dla siebie sposób. Zmiana algorytmu sortowania jest realizowana wyłącznie przez zmianę obiektu reprezentującego ten algorytm: jest przezroczysta z punktu widzenia obiektu Sorter.

**Intencja:** Odseparowanie zachowań od klas.

**Problem:** W hierarchi klas, to samo zachowanie musi być implementowane dla wielu. Chcemy to jakoś zhermetyzować

**Rozwiązania:** Zdefiniowanie interfejsu określającego zachowanie, oraz klas, implementujących różne warianty tego zachowania. Klasy naszej hierarchii będą posiadały teraz zachowanie jako obiekt klasy, implementującej nowy interfejs.

**Konsekwencje:** Elastyczniejszy design zgodny z Open/closed principle

**Uczestnicy:** Programista

### T7[↑](#id.5nzygkfqi063)

1. **Pojęcie i zasady zarządzania konfiguracją**

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.

Zarządzanie konfiguracją oprogramowania powinno zapewniać, że (**zasady**):

* 1. Każdy komponent oprogramowanie będzie jednoznacznie identyfikowany
	2. Oprogramowanie będzie zbudowane ze spójnego zestawu komponentów
	3. Zawsze będzie wiadomo, która wersja komponentu oprogramowania jest najnowasza
	4. Zawsze będzie wiadomo, która wersja dokumentacji pasuje do której wersji komponentu oprogramowania
	5. Komponenty oprogramowania będą zawsze łatwo dostępne.
	6. Komponenty oprogramowania nigdy nie zostaną stracone (np. wskutek awarii nośnika lub błędu operatora)
	7. Każda zmiana oprogramowania będzie zatwierdzona i udokumentowana.
	8. Zmiany oprogramowania nie zginą (np. wskutek jednoczesnych aktualizacji)
	9. Zawsze będzie istniała możliwość powrtotu do poprzedniej wersji.
	10. Historia zmian będzie przechowywana, co umożliwi odtworzenie kto i kiedy zrobił zminanę i jaką zminanę.
1. **Pojęcie pozycji konfiguracji, granulacja pozycji konfiguracji**
	1. pozycja konfiguracji -
	Wyróżnialny element uczestniczący w projekcie lub produkcie programistycznym. Traktowana jako pojedyńczy, możliwy do odseparowania komponent.
	2. granulacja pozycji konfiguracji -
	Atrybuty dotyczące danej pozycji konfiguracji, które wpływają na identyfikowanie tej pozycji. (chyba)
	Tu chodzi o to czy pozycją konfiguracji ma być pojedyncza linia kodu (masakra...) czy metoda, czy cała klasa, cały moduł itd.
2. **Hierarchie 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.

1. **Model procesu zarządzania konfiguracją**
	1. Identyfikacja pozycji konfiguracji
	2. Przechowywanie pozycji konfiguracji
	3. Kontrola zmian konfiguracji
	4. Określenie statusu konfiguracji
	5. Przekazanie pozycji konfiguracji na zewnątrz



1. **Pojęcie produktu bazowego i wydania**

***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. Produkty bazowe są niemodyfikowalne. Są one podstawą wymiany informacji pomiędzy uczestnikami projektu oraz podstawą testów nowego oprogramowania. Wczesne produkty bazowe zawierają dokumenty analityczne i projektowe. Późniejsze produkty bazowe zawierają również kod.

**Wydaniem** nazywamy każdą pozycję konfiguracji (zwykle cały projekt, ale niekoniecznie), która jest zakończona i oficjalnie przekazana na zewnątrz (zwykle na zewnątrz firmy wytwórcy oprogramowania). Wydania muszą być odpowiednio opisane, udokumentowane i zaaprobowane na poziomie kierownictwa projektu, kierownictwa firmy oraz klienta.

1. **Wersje pozycji konfiguracji**

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:

* 1. docelowa platforma/konfiguracja sprzętowa lub system operacyjny
	2. protokół komunikacyjny, współdziałanie z innym oprogramowaniem
	3. język użytkownika
	4. specyficzne wymagania poszczególnych użytkowników
	5. postępujące w czasie ulepszenia
	6. realizacja celów diagnostycznych i testowych podczas rozwoju oprogramowania

Istnienie wielu wersji pozycji konfiguracji 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.

1. **Zasady identyfikowania wersji**

Konwencja identyfikacji jest zbiorem stringów, zwykle złożonym z liter, cyfr i znaków kropki, /, -, itd. umożliwiająca jednoznaczne oznaczenie dowolnej pozycji konfiguracji. Konwencja ta 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.

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.
1. **Przechowywanie pozycji konfiguracji – zasady**

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 on zapewniać ewidencję i rejestrację zależności pomiędzy pozycjami konfiguracji. Dobrze zorganizowany system powinien być oparty na bazie danych oraz integrować informacje o PK z samymi (elektronicznymi) PK.

1. **Pojęcie statusu konfiguracji i tablicy statusu konfiguracji**

**Status 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, przy czym istotny jest przede wszystkim zapis stanu pozycji bazowych. Status konfiguracji powinien być na bieżąco aktualizowany i dostępny dla wszystkich członków zespołu projektowego.

**Tablica statusu konfiguracji** pokazuje 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.

1. **Zawartość planu zarządzania konfiguracją (norma IEEE 828-1990)**
	1. Wprowadzenie
		1. Cel

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

* + 1. 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. Osyłacze

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

* 1. Zarządzanie

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

* + 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.

* + 1. Odpowiedzialności w zakresie zarządzania konfiguracją

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.

* + 1. 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.

* + 1. Implementacja pozycji konfiguracji oprogramowania

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

* + 1. Stosowane strategie, zalecenia i procedury

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

* 1. Identyfikacja Konfiguracji
		1. Konwencje
		2. Produkty bazowe
	2. Kontrola kofiguracji
		1. Kontrola kodu i dokumentacji
		2. Kontrola mediów
		3. Kontrola zmian
			1. Poziomy autoryzacji zmian
			2. Procedury zmian
			3. Ciało (rada, komisja) przeglądowa
			4. Kontrola interfejsów
			5. Procedury zmian oprogramowania obcego
	3. Rejestracja statusu konfiguracji
	4. Narzędzia, techniki i metody dla zarządzania konfiguracją
	5. Kontrola dostawców
	6. Gromadzenie i przechowywanie zapisów

### T8[↑](#id.5nzygkfqi063)

1. **Pojęcia błąd, błędne wykonanie, weryfikacja, walidacja**

 **Błąd (failure, error)** - niepoprawna konstrukcja znajdująca się w programie, która

może doprowadzić do niewłaściwego działania.
 **Błędne wykonanie (failure)** - niepoprawne działanie systemu w trakcie jego

pracy.

**Weryfikacja** - Przeglądy, inspekcje, testowanie, sprawdzanie, audytowanie lub inna 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.

**Walidacja** - działanie mające na celu potwierdzenie w sposób udokumentowany i zgodny z założeniami, że procedury, procesy, urządzenia, materiały, czynności i systemy rzeczywiście prowadzą do zaplanowanych wyników.

1. **Źródła błędów**
2. **Metody zapobiegania i obsługi błędów, asercje**
3. **Miary niezawodności i metody jej pomiaru**

**Miara 1**

Jako miarę niezawodności przyjmuje się prawdopodobieństwo błędnego wykonania podczas realizacji transakcji. Każde błędne wykonanie powoduje zerwanie całej transakcji. Miarą jest więc częstość występowania transakcji, które nie powiodły się wskutek błędów.

## Miara 2

Jako miarę niezawodności przyjmuje się częstotliwość występowania błędnych wykonań, czyli liczbę błędów w jednostce czasu. Na przykład 0,1/h oznacza, że w ciągu godziny liczba spodziewanych błędnych wykonań wynosi 0,1. Miara ta jest stosowana w przypadku systemów, które nie mają charakteru transakcyjnego.

## Miara 3

Jako miarę niezawodności przyjmuje się średni czas między błędnymi wykonaniami - odwrotność miary 2.

## Miara 4

Jako miarę niezawodności przyjmuje się dostępność, czyli prawdopodobieństwo, że w danej chwili system będzie dostępny dla użytkowania. Miarę tę można oszacować na podstawie stosunku czasu, którym system jest dostępny, do czasu od wystąpienia błędu do powrotu do normalnej sytuacji. Miara zależy nie tylko od błęd-nych wykonań, ale także od narzutu błędów na niedostępność systemu.

1. **Metody testowania, czarna skrzynka, biała skrzynka,**

**Biała skrzynka** - sprawdzanie wewnętrznej logiki oprogramowania.Testowanie n/z białej skrzynki pozwala sprawdzić wewnętrzną logikę programów poprzez odpowiedni dobór danych wejściowych, dzięki czemu można prześledzić wszystkie ścieżki przebiegu sterowania programu.Programiści wstawiają kod diagnostyczny do programu aby śledzić wewnętrzne przetwarzanie. Debuggery pozwalają programistom obserwować

wykonanie programu krok po kroku. 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.

**Czarna skrzynka** - sprawdzanie funkcji oprogramowania bez zaglądania do środka

programu. Testujący traktuje sprawdzany moduł jak „czarną skrzynkę”, której

wnętrze jest niewidoczne.Testowanie n/z 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.

1. **Klasyfikacja metod testowania: statyczne, dynamiczne, jednostkowe, funkcjonalne, strukturalne, akceptacyjne, posiewowe, wstępujące, zstępujące, obciążeniowe, odpornościowe**

**Test jednostkowy** (ang. *unit test*) – w programowaniu metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu – np. metod lub obiektów w programowaniu obiektowym lub procedur w programowaniu proceduralnym. Testowany fragment programu poddawany jest testowi, który wykonuje go i porównuje wynik (np. zwrócone wartości, stan obiektu, wyrzucone wyjątki) z oczekiwanymi wynikami – tak pozytywnymi, jak i negatywnymi (niepowodzenie działania kodu w określonych sytuacjach również może podlegać testowaniu).

**Testowanie wstępujące** polega na rozpoczęciu testowania od testowania pojedynczych modułów. Następnie testowane są 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** polega na rozpoczęciu testowania 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.

Celem **testów odporności** jest sprawdzenie działania w przypadku zajścia niepożądanych zdarzeń, takich jak na przykład zaniku zasilania, awarii sprzętowej, wprowadzenia niepoprawnych danych czy wydania sekwencji niepoprawnych poleceń.

**Testy obciążeniowe** polegają na wymuszeniu obciążenia równego lub większego od maksymalnego.Celem testów obciążeniowych 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 statyczne** są formą testowania oprogramowania bez uruchamiania programu podczas testów. Test polega na automatycznym i ręcznym sprawdzaniu kodu w celu znalezienia błędów. Najczęściej wykonywany jest przez twórców kodu jako pierwsze i podstawowe sprawdzenie każdego programu. Testowanie statyczne sprawdza podstawową poprawność kodu i pozwala ocenić, czy program jest gotowy na bardziej szczegółowe testowanie.

**Testy statystyczne** - **wcześniej nazwane jako TESTY STATYCZNE które są wyżej** oparte na analizie kodu. Testy statystyczne przebiegają według ściśle określonego schematu i są wykonywane w sposób cykliczny:

* 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.

Zaletą testów statystycznych jest łatwość ich automatyzacji.Wadą zaś jest to, że wnioski dotyczące niezawodności oprogramowania wyciągnięte na podstawie analizy wyników mogą okazać się nie trafione.

 **Testy dynamiczne** - polegają na wykonywaniu (fragmentów) programu

i porównywaniu uzyskanych wyników z wynikami poprawnymi.

**Testy funkcjonalne** - zakładają znajomość wymagań wobec testowanych funkcji. System traktowany jest więc jako tzw. czarna skrzynka, która w nieznany sposób realizuje wymagane funkcje. Testy powinny wykonywać osoby, które nie były zaangażowane w realizacje testowanych fragmentów .systemu.

**Testy strukturalne** - zakładają znajomość sposobu implementacji testowanych funkcji. 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 lak, aby każda instrukcja została wykonana co najmniej raz. Spełnienie tego kryterium zwykle wymaga niewielkiej liczby testów.

* 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.

**Testy akceptacji (acceptance testing)** - w przypadku oprogramowania

realizowanego na zamówienie system przekazywany jest do przetestowania

przyszłemu użytkownikowi. Testy takie nazywa się wtedy testami alfa. W

przypadku oprogramowania sprzedawanego rynkowo testy takie polegają na

nieodpłatnym przekazaniu pewnej liczby kopii systemu grupie użytkowników.

Testy takie nazywa się testami beta.

**Testy posiewowe** - 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 - pozwala sprawdzić skuteczność testerów, ale i oszacować ile normalnych błędów jest niewykrywanych (skoro niewykryto posianych błędów, to prawdziwe też musiały się uchować) - wyniki są mylące, gdy prawdziwe błędy są innego typu niż te posiane

1. **Zależność faz testowania z cyklem życiowym wytwarzania oprogramowania**



1. **Metody testowania specyfikacji oprogramowania**

**Testowanie modułów i jednostek** - testuje się poszczególne komponenty (elementy systemu), aby zapewnić, że działają one poprawnie. Każdy składnik jest testowany niezależnie. Testowaniu podlegają również moduły. Moduł jest zbiorem niezależnych komponentów (kolekcją procedur i funkcji). Moduł można testować bez udziału innych modułów systemu

**Testowanie integracji** - faza obejmująca testowanie zbioru modułów, które zostały zintegrowane w podsystem. Proces testowania takiego podsystemu zwykle prowadzi do wykrycia błędów w interfejsach modułów.

**Testowanie całości systemu** - system jest kolekcją różnych podsystemów. Proces testowania systemu ma wykryć błędy, których przyczyny związane są z nie przewidzianych interakcji między podsystemami. W fazie testowania systemów sprawdza się też, czy system spełnia stawiane mu wymagania.

**Testowanie akceptacyjne** jest końcową fazą procesu testowania, którą wykonuje się przed przyjęciem systemu do użytkowania. Gotowy system testuje się za pomocą danych dostarczonych przez użytkownika, a nie danych symulowanych. Ta faza testów może doprowadzić do wykrycia błędów i zaniedbań w definicji wymagań stawianych systemowi.

1. **Dobór danych testowych, klasy równoważności, krzywa testowania**
	1. dobór danych testowych
		1. ogólnie: dane typowe i graniczne
		2. gdy trzeba ograniczyć, to:
			1. możliwość wykonania funkcji jest ważniejsza niż jej jakość *(hmm raczej nie zawsze...)*
			2. funkcjonalności starego systemu są ważniejsze niż te nowo wprowadzone
			3. typowe sytuacje są ważniejsze niż te skrajne
		3. przy testach strukturalnych - cel: pokrycie wszystkich instrukcji lub wszystkich instrukcji warunkowych
		4. testowanie pętli - cel: wykonanie minimalnej / maksymalnej / przeciętnej liczby iteracji
	2. klasa równoważności - grupuje “podobne” dane testowe. Zakłada się, że jeśli testy przechodzą dla kilku danych z klasy, to przechodzą dla całej klasy. Ale należy jeszcze przetestować sytuacje brzegowe.
	3. krzywa testowania - ?
2. **Metody analizy kodu**
3. **Metody testowania dokumentacji**
4. **Automatyzacja w testowaniu**
5. Przeglądy, inspekcje i audyty oprogramowania

**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 w celu uzyskania komentarzy, opinii i akceptacji. Przeglądy mogą być formalne i nieformalne.

Formalne przeglądy mogą mieć następującą postać przeglądu technicznego, przejścia lub audytu.

**Przegląd techniczny** polega na ocenie elementów oprogramowania na zgodność postępu prac z przyjętym planem.

**Przejście** (walkthrough) jest wczesną oceną dokumentów, modeli, projektów i kodu. Jego celem jest zidentyfikowanie defektów i rozważenie możliwych rozwiązań, a celem wtórnym jest szkolenie i rozwiązanie problemów stylistycznych (np. z formą kodu, dokumentacji, interfejsów użytkownika).

**Inspekcja** jest formalną techniką oceny, w której wymagania na oprogramowanie, projekt lub kod są szczegółowo badane przez osobę lub grupę osób nie będących jej autorami. Inspekcję przeprowadza się w celu identyfikacji błędów, sprawdzenia czy nie zostały naruszone standardy oraz identyfikacji innych problemów powstałych w czasie procesu wytwórczego oprogramowania

**Audyt** jest formą przeglądu potwierdzającym 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. Celem audytu projektu informatycznego jest dostarczenie odbiorcy i dostawcy obiektywnych, aktualnych i syntetycznych informacji o stanie całego projektu.

Przedmiotem audytu informatycznego mogą być procesy lub produkty projektu (lub jedno i drugie).

**Audyt procesów** projektu informatycznego ma na celu sprawdzenie czy wykonywane prace oraz sposób ich wykonywania są prawidłowe, czy przebiegają zgodnie z przyjętymi zasadami lub standardami itp.

**Audyt produktów** (końcowych lub cząstkowych) projektu informatycznego ma na celu sprawdzenie czy rezultaty poszczególnych prac odpowiadają zakładanym założeniom i stawianym przed nimi wymaganiom.

Audyt może koncentrować się na różnych perspektywach projektu.

**- aspekty technologiczne:** Wówczas jego głównym celem jest sprawdzenie, czy użyte techniki oraz opracowane rozwiązania techniczne są prawidłowe i prawidłowo stosowane.

**- aspekty zarządcze:** Wówczas jego głównym celem jest sprawdzenie, czy sposób zarządzania projektem umożliwia osiągnięcie przez niego sukcesu.

1. Refaktoring

**Refaktoryzacja kodu** to proces wprowadzania zmian w projekcie informatycznym, w wyniku którego zasadniczo nie zmienia się funkcjonalność.Celem refaktoryzacji jest utrzymanie wysokiej jakości organizacji systemu, poprawienie pewnych atrybutów niefunkcjonalnych oprogramowania

.

### T9[↑](#id.5nzygkfqi063)

1. **Pojęcie pomiarów, metryk i miary**
	1. Pomiar - jest to proces w którym atrybutą świata rzeczywistego przydzielane są liczby lub symbole w taki sposób aby charakteryzować te atrybuty wg określonych zasad.
	2. Miara - jednostka przydzielana atrybutowi.
	3. Metryka - to proponowana miara, nie zawsze charakteryzuje w sposób obiektywny dany atrybut.
2. **Artefakty podlegające pomiarowi**
	1. Proces - każde określone działanie w ramach projektu, wytwarzania lub eksploatacji oprogramowania.
	2. Produkt - każdy przedmiot powstały w wyniku procesu np. kod źródłowy, specyfikacja, plan testó itp....
	3. Zasób - każdy niezbędny element do realizacji procesu np. osoba, narzędzie.
3. **Budowa strategii pomiarowej**
4. **Przykłady pomiarów i miar , miary bezpośrednie i pośrednie**
5. **Budowa modeli pomiarowych**
6. **Efekt skali w wytwarzaniu oprogramowania**

Polega na wykładniczym wzroście zużycia zasobów w projekcie w funkcji wielkości projektu

$$ZużycieZasobu = ZużycieStałe + K \* WIelkośćProjektu^{β}$$

1. **Metoda COCOMO II**

Metoda **COCOMO** i jej pochodne za podstawę analizy biorą liczbę linii kodu źródłowego LOC. Następnie przedsięwzięcie jest zaliczane do jednej z klas:

 a. **Przedsięwzięcie łatwe (organiczne)**. Obejmuje przedsięwzięcia wykonywane przez małe zespoły, złozone z osób o podobnych kwalifikacjach, dziedzina projektu jest dobrze znana podobnie jak narzędzia i metody.

 b. **Przedsięwzięcie niełatwe (półoderwane)**. Człoonkowie zespołu różnią się stopniem zaawansowania, pewne aspekty dzieziny problemu nie są dobrze znane.

 c. **Przedsięwzięcie trudne (osadzone).** systemy o bardzo dużej złożoności wymagań. Dziedzina problemu, narzędzia i metody w dużej mierze nieznane, a członkowie zespołu nie mają doświadczenia w podobnych zadaniach.

Wzór dla oszacowania nakładów w COCOMO:

 $Nakłąd[osobomiesiące] = A \* K^{}^{b}$ gdzie: K - rozmiar kodu źródłowego mierzony w tysiącach linii, A i b zależy od klasy problemu:

 łatwe - A = 2,4 b = 1,05

 niełatwe - A = 3,0 b = 1,12

 trudne - A = 3,6 b = 1,20

Metoda ta zakłada, że znając nakład można oszacować czas realizacji przedsięwzicia, z czego wynika przyblizona wielkość zespołu i wykorzystania zasobów. To co wyjdzie trzeba skorygować biorąc pod uwagę:

 a. wymagania wobec niezawodności systemu

 b. rozmiary baz danych w stosunku do rozmiaru kodu

 c. złożoność systemu

 d. wymagania co do wielkości systemu

 e. ograniczenia pamięci

 f. zmienność sprzętu i oporogramowania.

WADY:

 a. liczba lini kodu znana jest dokładnie gdy system jest już gotowy

 b. szacunki zależą od wielu subiektywnych czynników np. doświadczenie analityka

 c. zły wybór czynników może prowadzić do rozbierzności pomiędzy oczekiwaniami a rzeczywistym kosztem przedsięwzięcia.

1. **Metoda APF (FPA)**

Metoda punktów funkcyjnych jest metodą wymiarowania oprogramowania z punktu widzenia użytkownika. Estymacja odbywa się w oparciu o dane zawarte we wstępnej specyfikacji wymagań funkcjonalnych. Można nią szacować we wczesnych fazach projektu.

Aby wyliczyć liczbę nieskorygowaną punków funkcyjnych bierze sie pod uwagę dane:

 a. Wejście użytkownika - obiekty wejściowe wpływające na dane w systemie

 b. Wyjście użytkownika - obiekty wyjściowe związane z danymi w systemie

 c. Zbiory danych wewnętrzne - liczba weenetrznych plików roboczych

 d. Zbiory danych zewnętrzne- liczba plików zewnętrznych zapełnianych przez system

 e. Zapytania zewnętrzne - interfejsy z otoczeniem programu

Miarą produktywności oprogramowania jest jednostka zwana *punktem funkcyjnym*.

Jest ona powiązana z określonymi typami funkcjonalności oprogramowania. Wyróznia się 5 abstrakcyjnych klas funkcjonalności:

 1. wejście systemu

 2. wyjście systemu

 3. funkc. wew. struktur danych

 4. funkc. komunikacji z prog. zewnetrznymi

 5. zapytania do systemu

Każdy komponent oprogramowania musi zostać zaklasyfikowanyc do jejenej z grup oraz nalezy przypisać jeden z poziomów złożoności (prosty, średni, złożony).

Zastosowanie FPA:

a. ocena złożónosci realizacji systemu

b. audyt projektów

c. szacowanie liczby testów

d. prognozowanie kosztów pielęgnacji i rozwoju systemu.

WADY:

 1. subiektywność w przypisywaniu punktów funkcyjnych, zalezne od doświadczenia analityka

 2. trzy poziomy (niski, średni i wysoki) mogą okazać się niewsystarczające.

1. **Organizacja zespołu, struktury organizacyjne**

1. **Jakość, model dojrzałości procesów**

### T10[↑](#id.5nzygkfqi063)

1. **Pojęcie bazy danych i warstwy magazynu danych**

**Baza danych** jest to zbiór danych zapisanych zgodnie z określonymi regułami. W węższym znaczeniu obejmuje dane cyfrowe gromadzone zgodnie z zasadami przyjętymi dla danego programu komputerowego specjalizowanego do gromadzenia i przetwarzania tych danych.

**Najniższą warstwę tworzą źródła danych**, czyli zastane bazy danych przedsiębiorstwa, często rozproszone geograficznie, zróżnicowane pod względem sposobu dostępu (zwykłe bazy danych różnych formatów, pliki binarne lub tekstowe, źródła specjalne), a także struktury logicznej, wielkości i jakości danych.
**Środkową warstwę na schemacie zajmuje** **centralna hurtownia danych** (podstawowa, korporacyjna). Stanowi ona podstawowe miejsce przechowywania nieulotnej informacji gromadzonej ze źródeł, jak też częściowych podsumowań przydatnych w zadaniach typu OLAP i we wspomaganiu decyzji. Globalna hurtownia danych rejestruje historię źródła danych i jest cyklicznie, podczas aktualizacji, uzupełniana o nowe, skondensowane informacje dotyczące aktualnego stanu źródła danych, zapisywane obok poprzednich. Dane w hurtowniach danych mogą pochodzić sprzed wielu lat, gdyż hurtownia spełnia także funkcje archiwalne.
**Kolejną warstwę stanowią hurtownie lokalne**, tworzone na potrzeby użytkowników (działów analitycznych), zawierające wyselekcjonowane dane w postaci silnie zagregowanej, pozwalające na szybką prezentację podsumowań wykorzystywanych w zarządzaniu, planowaniu długoterminowym, analizach historycznych, analizach trendów, przetwarzaniu informacji i analizach zintegrowanych. Lokalne hurtownie danych nazywane są **hurtowniami tematycznymi** (data marts, hurtownie oddziałowe). Ze względu na mniejszy rozmiar i możliwość pracy lokalnej, hurtownie tematyczne pozwalają na sprawniejsze operowanie danymi. Mogą być zaimplementowane jako relacyjne bazy danych lub specjalne struktury wielowymiarowe.

1. **Klasyfikacja baz danych**
	1. Bazy proste
		1. bazy kartotekowe
		2. hierarchiczne bazy danych
	2. Bazy złożone
		1. bazy relacyjne
		2. bazy obiektowe
		3. bazy relacyjno-obiektowe
		4. strumieniowe bazy danych
		5. temporalne bazy danych
2. **Model relacyjny, hierarchiczny i obiektowy bazy danych**
	1. relacyjny - dane przechowywane w tabelach o stałej liczbie kolumn i dowolnej liczbie wierszy. Tabela ma określony klucz (jeden lub wiele atrybutów), po którym identyfikuje się konkretny wiersz. Wartością atrybutu może być klucz z innej tabeli (klucz obcy)
	2. hierachiczny - dane zapisane jako rekordy nadrzędne-podrzędne. Każdy rekord (poza korzeniem) jest związany z dokładnie z jednym nadrzędnym. Tworzą strukturę drzewa.
	3. obiektowy - dane przechowywane w formach obiektów, zupełnie jak w obiektowych językach programowania (mają cechy i udostępniają metody).
3. **Zalety i wady modelu relacyjnego i modelu obiektowego**
	1. RELACYJNE
	Zalety
	- bardzo dobrze znany i nieźle dopracowany SQL
	- dużo narzędzi
	- działa? działa!
	Wady
	- brak wsparcia dla skomplikowanych typów
	- wiedza o strukturze bazy wymagana do ad hoc zapytań
	- baza relacyjna + oprogramowanie obiektowe? -> konieczność mapowania obiekt-relacja (vide hibernate)
	2. OBIEKTOWE
	Zalety
	- łatwiejsza nawigacja
	- mniejsza ilość kodu przy używaniu do obiektowo napisanych aplikacji (brak mapowania O-R)
	- dobrze działa na rozproszonych architektur
	- lepsza kontrola wpółbierzności - możemy zalockować pewne hierarchie
	Wady
	- mniej efektywna dla prostych baz
	- bardziej skomplikowana
	- mniej stabilne/popularne
4. **Transakcja jako mechanizm bezpieczeństwa bazy danych**Transakcja, jako pojedyńcza logiczna operacja na danych, ma spełniać postulat **ACID:**- Atomicity - atomowość
- Consistency - spójność
- Isolation - odizolowanie
- Durability - trwałość
Gdy spełnia, jesteśmy pewni że żadna część transakcji nie uchowa się po odrzuceniu transakcji np. przy transakcji “przelej kase z A na B” i odrzuceniu transakcji po odjeciu kasy z A, możemy być pewni, że nasza baza cofnie się do stanu sprzed transakcji.
5. **Operacje elementarne w ramach transakcji**

 Read, Write, Commit, Rollback

1. **Historie przetwarzania i anomalie historii przetwarzania**
	1. anomalie :
		1. nieodwarzalna historia przetwarzania - (wyjasnienie przez przyklad) -
		T1 zapisuje pewną wartość X. T2 odczytuje wartość X, i na podstawie X ustala wartość Y. T2 kończy się sukcesem.
		W tym momencie następuje odrzucenie T1, więc wycofujemy wszystkie zmiany - cofamy X do starej wartości, ale nie damy rady cofnąć T2, bo jest już zatwierdzona.
		Historia przetwarzania transakcji będzie odtwarzalna, jeśli każda transakcja jest zatwierdzana dopiero po zatwierdzeniu wszystkich transakcji, z których czyta - w naszym przypadku T2 czekałoby na zatwierdzenie T1.
		2. kaskada odrzuceń -
		Uzyskanie odtwarzalnej historii przetwarzania wiąże się z możliwością powstania kaskady odrzuceń. T1 zapisuje X, T2 czyta X, T2 zapisuje Y, ale nijak nie zwiazane z X Konczymy T2(czeka na koniec T1). Odrzucamy T1. No i teraz T2 tez musi zostać odrzucone, bo czytało z T1, a szkoda, bo mogło by przejść.
		Jak obejść ? - niech żadna transakcja nie czyta z transakcji niezatwierdzonych.
		3. powtórne czytanie/niepowtarzalne czytanie -
		czytanie danej drugi raz zwróci inną wartość, tj przez jakąś chwilę, wtedy co czytaliśmy, dane pole miało wartość X, ale już nigdy nie doczytamy się tej wartości - bo zaraz ktoś ją zmienił, a my nic o tym nie wiemy.
		Jak obejść? - niech żadna transakcja nie zapisuje w transakcjach niezatwierdzonych.
		4. fantomy -
		przy braku locka na zakres danych (od warunku w WHERE), może zajść dopisanie rekordu (przez inna transakcję) spełniającego warunek where + commit. Wtedy, gdy pierwsza transakcja znowu wykona tego samego selecta otrzyma fantoma - widzi inny wynik tego selecta niż na pooczątku.
2. **Poziomy izolacji transakcji**
	1. 0 - READ UNCOMMITED -
	czytanie danych z tr. niezatwierdzonych.
	2. 1 - READ COMMITED -
	czytanie tylko z zatwierdzonych. Może wystąpić anomalia powtórnego czytania.
	3. 2 - REPEATABLE READ -
	zakaz czytania i zapisywania do niezatwierdzonych. Locki na read i write aż do końca transakcji. Brak locka na zakres. Mogą wystąpić fantomy.
	4. 3 - SERIALIZABLE - zakaz czytania i zapisywania do niezatwierdzonych. Locki na read i write + przy readzie locki na zakres (zależnie od warunku w WHERE). Najbezpieczniejsze ale i najwolniejsze.



1. **Protokół dwufazowy w przewarzaniu transakcji
(**B2F/2PC) - protokół do zatwierdzania transakcjii. Dzieli się na 2 fazy. Mamy koordynatora całej zabawy (moduł planisty). Dostaje on od różnych transakcji różne operacje do wykonania. może on z taką operacją ( Pi[x] ) zrobić pare rzeczy :
- jeżeli Pi[x] może być wykonanie, to ustawia blokadę na x dla transakcji o numerze i, i przekazuje do menadżera danych
- jeżeli nie może, to do kolejki
Blokade zdejmie najwcześniej gdy dostanie odpowiedź od MD o wykonaniu operacji.
I tu uwaga uwaga, jeżeli nastąpiło zdjęcie jakiejkolwiek blokady dla transakcji o numerze i, to nie możemy dla tej transakcji zakładać już zadnych nowych blokad.