

Notatki z kursu Inżynieria Oprogramowania

Małgorzata Dymek

2018/19, semestr letni

1 Podstawowe pojęcia

Inżynieria oprogramowania

Produkcja oprogramowania

Zastosowanie **inżynierskiego** (systematycznego, zdyscyplinowanego, ilościowego) **podejścia** do oprogramowania (rozwoju, eksploatacji, utrzymania).

- analiza,
- wymagania,
- projektowanie,
- wdrożenie, ewolucja systemu

Proces tworzenia oprogramowania

Zbiór czynności i związanych z nimi wyników, **prowadzących do powstania systemu** informatycznego - tworzenie oprogramowania od zera, rozszerzanie i modyfikowanie istniejących systemów.

- specyfikacja
- podręcznik użytkowania
- scenariusze przypadków użycia
- raport o statusie projektu
- podręcznik testera

Aktywności, zadania i **zasoby**.

- zbieranie, analizowanie wymagań
- realizacja przypadku użycia
- projektowanie systemu, obiektów
- implementacja, testowanie
- komunikacja,
- zarządzanie konfiguracją, projektem
- cykl życiowy oprogramowania

Scenariusz przypadku użycia - wy-specyfikowana sekwencja zdarzeń między użytkownikiem a systemem.

- Zdefiniowane w pierwszej kolejności.
- Wyróżnia się jeden **główny scenariusz sukcesu**.
- Może zawierać warunki wstępne, gwarancje lub wyzwalacze.
- W agile development używa się skróconej wersji scenariusza odpowiadającej na pytania: kto, co, dlaczego.

Przypadek użycia - zbiór powiązanych ze sobą scenariuszy opisujących użycie systemu przez aktorów.

Opisujemy je tekstowo, poprzez user stories lub diagramy.

- reprezentuje **funkcjonalne wymaganie** systemu;
- pewna historia; opisuje akcje systemu z punktu widzenia użytkownika;
- specyfikuje jeden aspekt zachowania bez wchodzenia w strukturę systemu;
- jest zorientowany na osiągnięcie celu użytkownika;

2 UML

UML - Unified Modelling Language

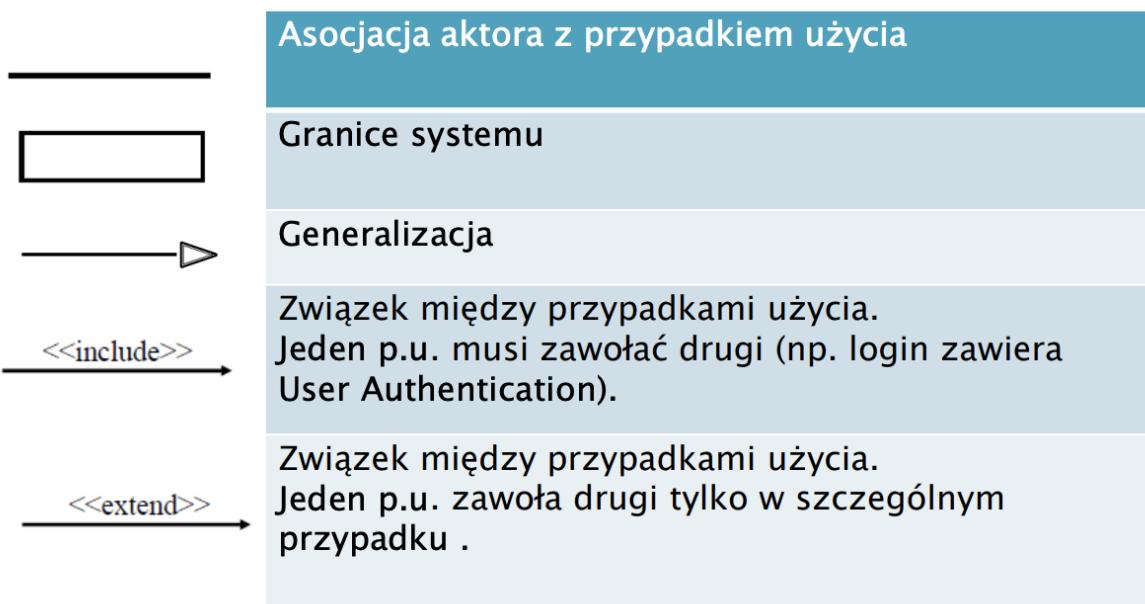
- rodzina **notacji** graficznych; unifikacja wielu obiektowych języków modelowania graficznego.
- służy do opisywania i projektowania systemów informatycznych.
- nadzorowany przez organizację OMG
- dwie podstawowe części:
 - **Notacja** - składnia oraz elementy graficzne.
 - **Metamodel** - semantyka oraz definicje pojęć języka i ich powiązań.

Sposoby użycia UML:

Szkic	Projekt	Język programowania
<ul style="list-style-type: none">• użytkowy podczas tworzenia i odtwarzania,• charakter selektywny, nie-formalny, dynamiczny.	<ul style="list-style-type: none">• użytkowy podczas tworzenia i odtwarzania,• powinien być kompletny i zawierać wszystkie istotne decyzje,• może dotyczyć tylko części systemu,• wymaga bardziej skomplikowanych narzędzi niż szkic.	<ul style="list-style-type: none">• diagramy tak szczegółowe, że można generować kod automatycznie,• wymaga bardzo skomplikowanych narzędzi.

Aktor - użytkownik systemu; ogólnie wszystkie role, które wchodzą w interakcję z systemem, komponenty odpowiedzialne za inicjalizację use case'ów.

Diagramy przypadków użycia są używane do modelowania kontekstu i wymagań systemu.



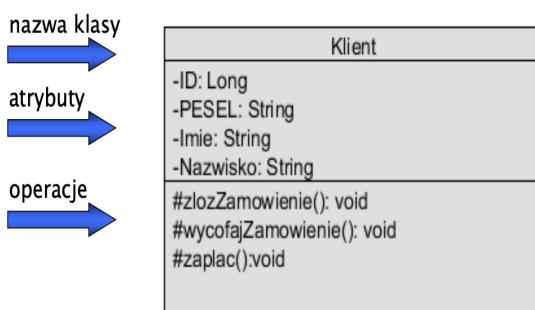
2.1 Diagramy UML

Strukturalne	Behawioralne
<ul style="list-style-type: none"> diagram klas diagram komponentów diagram obiektów diagram pakietów diagram wdrożenia diagram struktur złożonych 	<ul style="list-style-type: none"> diagram czynności diagram stanów diagram przypadków użycia diagram komunikacji diagram sekwencji diagram przeglądu diagram przebiegów czasowych

2.1.1 Diagram klas

- jeden z najczęściej używanych, zwłaszcza do **generowania kodu**.
- ilustracja w modelach obiektowych struktury klas i zależności między nimi,
- podział odpowiedzialności pomiędzy klasy i rodzaj wymienianych komunikatów,

Klasy służą do opanowania słownictwa tworzonego systemu. Używane do przedstawienia bytów programowych, sprzętowych, koncepcyjnych.



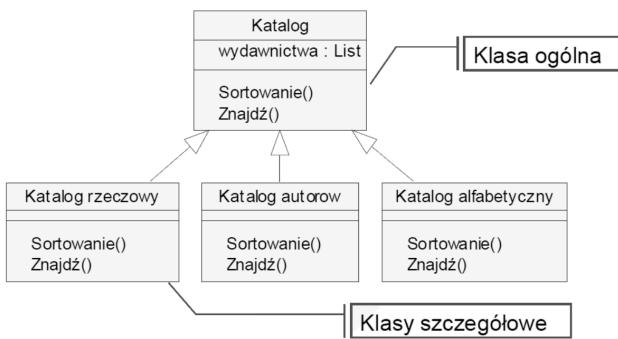
- **Parametr** - przyjmowany przez metodę.
- **Atrybut** - nazwana właściwość klasy. Może mieć podaną klasę i wartość początkową.
 - + public
 - # protected
 - – private
- **Operacja** (metoda) - implementacja pewnej usługi, której wykonania można zarządzać od każdego obiektu klasy. Dokładne określenie przez podanie sygnatury (typy i domyślne wartości parametrów).
- **Odpowiedzialność** - wyrażenie zobowiązań klasy (lista).
- **Instancja** - każdy egzemplarz przechowuje oddzielną wartość
- **Classifier** - jest tylko jedna wartość wspólna dla wszystkich egzemplarzy
- **Ilość wystąpień** - liczba w prawym górnym rogu (1 = *singleton*)



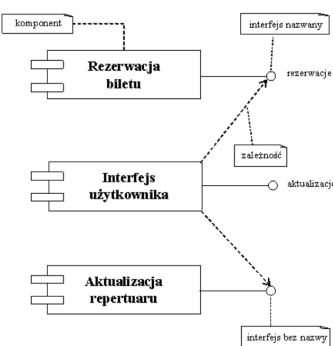
Dodatkowo:

- **Implementacja** - linia przerywana zakończona pustą strzałką.
- **Rozszerzenia**
 - **stereotypy** - służą do stworzenia nowego rodzaju obiektu ma podstawie innego obiektu będącego już w standardzie UML.
 - **ograniczenia** - dotyczą zarówno klas jak i związków między nimi.
- **Szablony** - klasy parametryzowane. Parametry szablony w prostokącie z przerywanych linii obok diagramu klasy.

UML – diagram klas



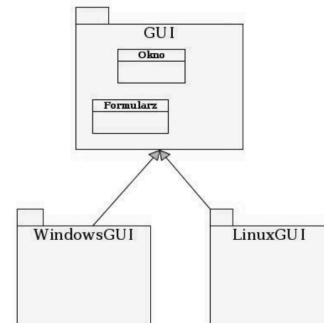
UML – diagram komponentów



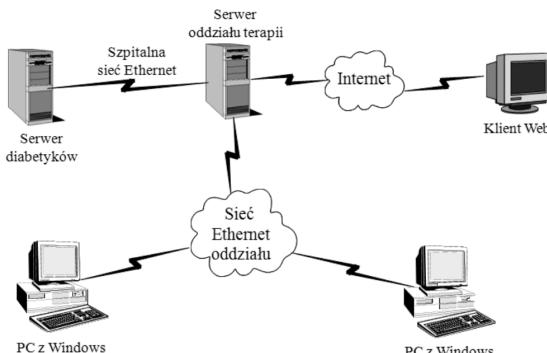
UML – diagram obiektów



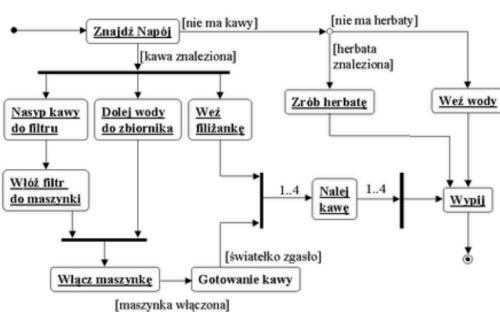
UML – diagram pakietów



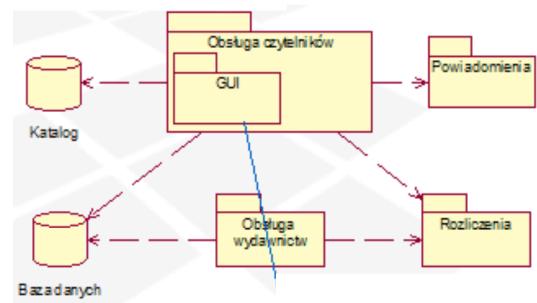
UML – diagram wdrożenia



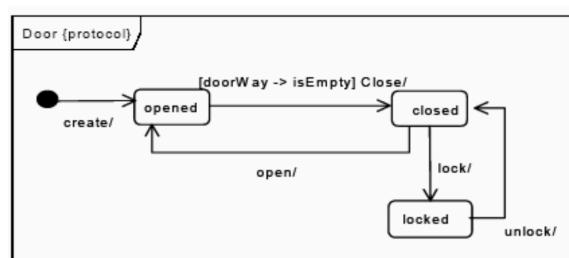
UML – diagram czynności



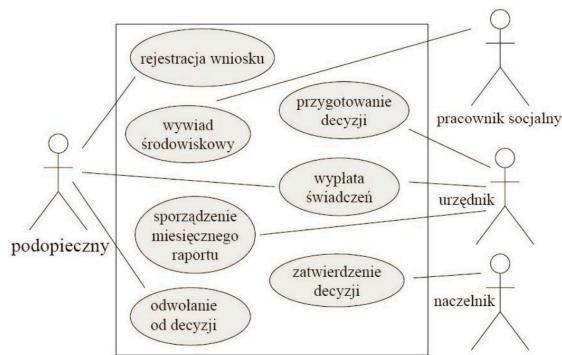
UML – struktury złożonych



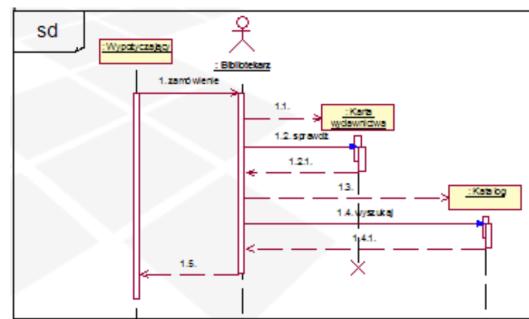
UML – diagram stanów



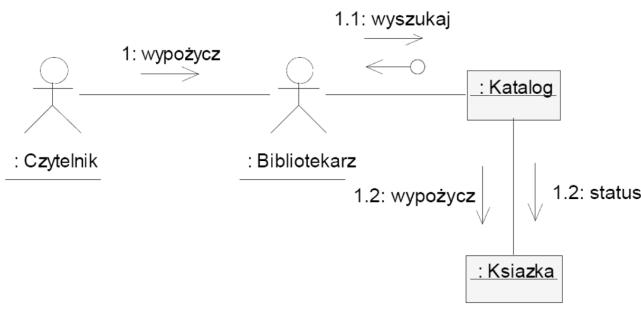
UML – diagram przypadków użycia



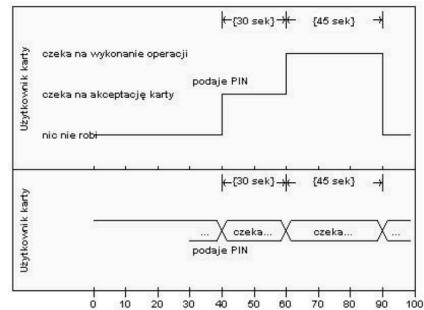
UML – diagram sekwencji



UML – diagram komunikacji



UML – diagram przebiegów czasowych



3 Procesy wytwarzania oprogramowania

Cztery fundamentalne działania wspólne dla wszystkich procesów:

- **Specyfikowanie** oprogramowania
 - zebranie wymagań,
 - definiowanie funkcjonalności i ograniczeń dotyczących tworzonego software'u.
- **Tworzenie** oprogramowania
- **Walidacja** oprogramowania
- **Ewolucja** oprogramowania

Cykl życia systemu - cały okres istnienia systemu:

- studium zastosowalności,
- analiza i specyfikacja,
- projektowanie i tworzenie,
- wdrażanie,
- pielegnacja,
- aspekty usprawnienia.

3.1 Modele procesu wytwarzania oprogramowania

Model cyklu życia systemu informatycznego ma na celu przedstawienie procesu wytwarzania oprogramowania, który prowadzi do stworzenia działającego systemu.

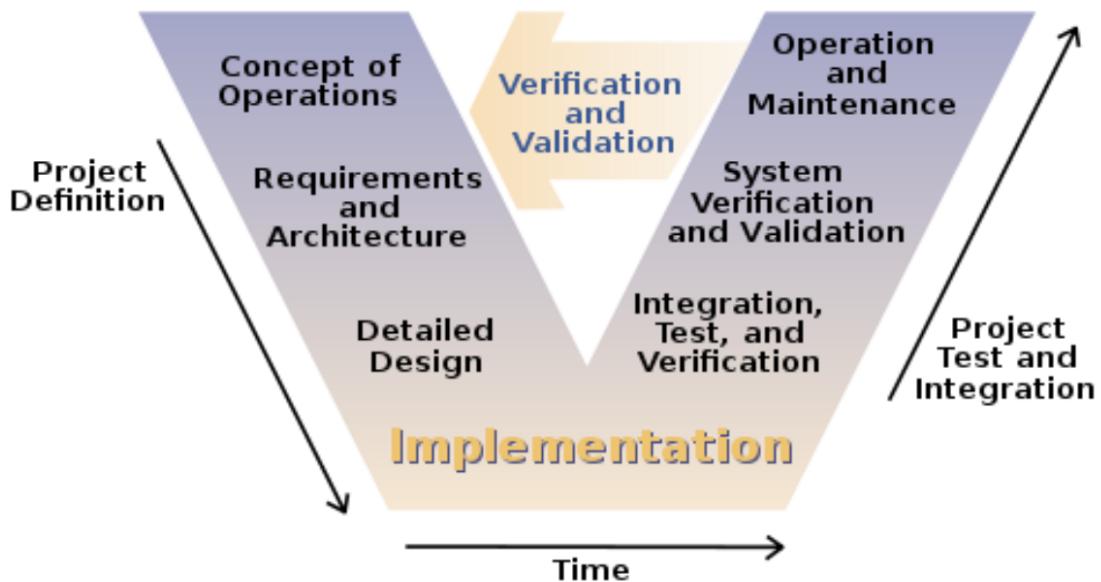
3.1.1 Model kaskadowy

- **Wyizolowane etapy**, każdy musi być zakończony przed rozpoczęciem kolejnego
 - Planowanie: cele biznesowe, podstawowe wymagania, założenia, standardy, parametry,
 - Analiza: zdefiniowanie przeznaczenia systemu → kompletny, spójny i jednoznaczny model systemu,
 - Projekt: dekompozycja na podsystemy, wybór strategii budowania, projektowanie obiektów, wybór komponentów, opis interfejsów,
 - Implementacja: tworzenie kodu źródłowego, mapowanie modeli na kod,
 - Testowanie: znajdowanie różnic między rzeczywistym elementem a jego modelem,
 - Pielegnacja.
- Etapy podzielone na dwie części: twórczą i weryfikacji
- Ponowna praca, jeśli konieczna, jest wykonywana w kolejnych etapach - bardzo wysoki koszt błędów popełnionych we wstępnych etapach, adaptowanie zmian bardzo kosztowne.
- Koszty opracowania i akceptacji dokumentów są wysokie, powinien być używany tylko jeśli wymagania są jasne i zrozumiałe.
- Marginalizacja roli klienta w procesie wytwarzania oprogramowania. Uzyskanie produktu zgodnego z wymaganiami silnie zależne od ich stabilności.

3.1.2 Model V

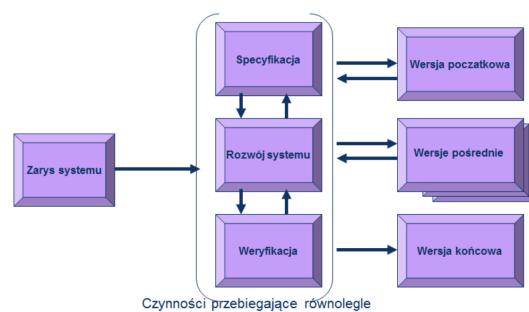
Sekwencyjny model, najczęściej posiadający cztery poziomy testowania odpowiadające czterem poziomom rozwoju oprogramowania.

- wymagania klienta - testy akceptacyjne
- wymagania systemowe - testy systemowe
- ogólny design - testy integracyjne
- szczegółowy design - testy modułowe



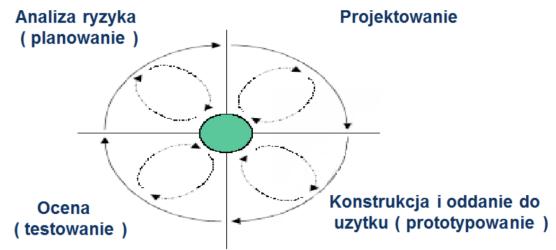
3.1.3 Model Ewolucyjny

- pozwala później określić wymagania do projektowanego systemu,
- prototyp pomaga kształcić przyszłego użytkownika,
- prototyp podnosi koszty w krótszej perspektywie, ale w dłuższej może je obniżać,
- zwykle prototyp jest wyrzucany.



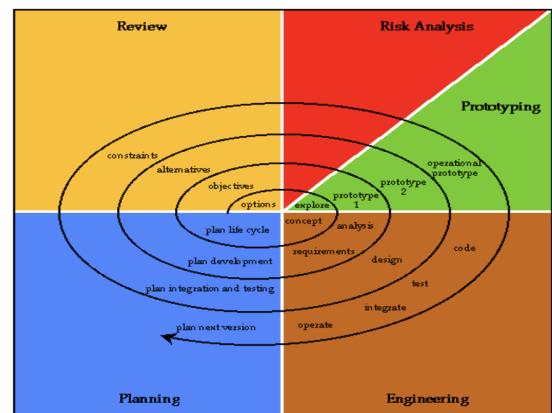
3.1.4 Model iteracyjny

- pozwala na wczesne wykrywanie błędów,
- łączy iteracje z klasycznym modelem kaskadowym,
- łatwość wprowadzania zmian,
- wymogi klienta dotyczące harmonogramu mogą utrudnić korzystanie z tego modelu,
- problemy z oszacowaniem ryzyka.



3.1.5 Model spiralny

- ciągłe monitorowanie i pomiar zmian,
- zmiany poddawane są review użytkownika,
- próba minimalizacji ryzyka niepowodzenia.



4 Standardy jakości

Odpowiedź na syndrom **LOOP** - Late, Over budget, Overtime, Poor quality.

Minusy standaryzacji:

- Ważniejszy proces niż samo oprogramowanie,
- Spora część procesu jest fikcyjna, tworzenie dużej ilości dokumentacji,
- Dyscyplina zabija inicjatywę.

4.1 CMM - Capability Maturity Model

Ocenia proces wytwarzczy służący do produkcji oprogramowania w skali pięciostopniowej - od **charakterystycznego** (nic nie jest sterowane ani kontrolowane), aż do **ścisłego**, zdyscyplinowanego procesu uwzględniającego wszystkie potrzebne aspekty.

Model CMM obejmuje **pięć aspektów**:

- Poziomy dojrzałości
- Kluczowe obszary procesowe
- Cele
- Atrybuty procesu

- Kluczowe praktyki

4.1.1 Poziomy dojrzałości

Poziom 1 - Wstępny

- Działanie tymczasowe, doraźne.
- Organizacja bez stabilnej technologii wytwarzania i utrzymywania produktów.
- Zakres projektów zupełnie nieprzewidywalny.

Poziom 2 - Powtarzalny

- Dokumentowane standardy dokumentacji, szkoleń, utrzymywania.
- W miarę ustabilizowane środowisko pracy i procedury zarządzania.

Poziom 3 - Zdefiniowany

- Spójny zbiór definicji i standardów na poziomie organizacji realizującej projekt.
- Wyodrębnienie w zespole specjalistów od realizacji poszczególnych zadań, a organizacja dąży do wyposażenia ich w niezbędną wiedzę i umiejętności.
- Organizacja zaczyna na podstawie własnych doświadczeń modyfikować sposób prowadzenia projektów, tak aby maksymalnie odpowiadał jej specyfice.

Poziom 4 - Zarządzany

- W jakimś zdefiniowanym obszarze wyniki podejmowanych działań przenoszą określone rezultaty, które można zmierzyć za pomocą wcześniej zdefiniowanych metryk.
- Zadania, których wykonanie nie generuje dużej liczby błędów mogą być kontrolowane z mniejszą częstotliwością, zaś obszary zdefiniowane jako potencjalnie niebezpieczne np. w związku ze zmianą technologii, mogą podlegać ściszej kontroli.

Poziom 5 - Optymalizujący

- Proces jest już tak dobrze zorganizowany i zarządzany, że nie pozostaje nic innego, jak tylko dalsze podnoszenie stawianych przed procesem wymagań.
- Celem stawianym na tym poziomie jest optymalizacja i dalsze ulepszanie procesu, zwiększenie jego efektywności oraz wydajności.

4.2 ISO 9000

Wymaga udokumentowania wszystkich procedur związanych z wytwarzaniem oprogramowania.



Odpowiedzialność kierownictwa

Kierownictwo:

- odpowiada za właściwe funkcjonowanie organizacji,
- ustala misję i politykę organizacji, określa cele,
- opracowuje plan działań do realizacji celów i przyznaje odpowiednie zasoby.

Zarządzanie zasobami

- Zespół procesów związanych z zasobami - ludzkimi, infrastrukturą, środowiskiem pracy.

Realizacja wyrobu

- Zespół procesów bezpośrednio związany z realizacją wyrobu lub usługi.
- Wejściem są wymagania klienta, a wyjściem jest dostarczony wyrób lub usługa.

Pomiar, analiza i doskonalenie

- Procesy w organizacji i zadowolenie klienta wymagają systematycznego monitoringu, analizy i podejmowania działań doskonalących, aby wiedzieć jak postrzega nas klient (zadowolenie) oraz jak funkcjonują procesy w organizacji (zielona strzałka).

Ciągłe doskonalenie systemu zarządzania jakością

- Ciągłe zwiększanie skuteczności i efektywności w realizacji polityki, strategii i celów organizacji.

5 Zwinne procesy wytwarzania oprogramowania

Idea przepływu produktu przez system, podczas którego systematycznie zwiększa się jego wartość.

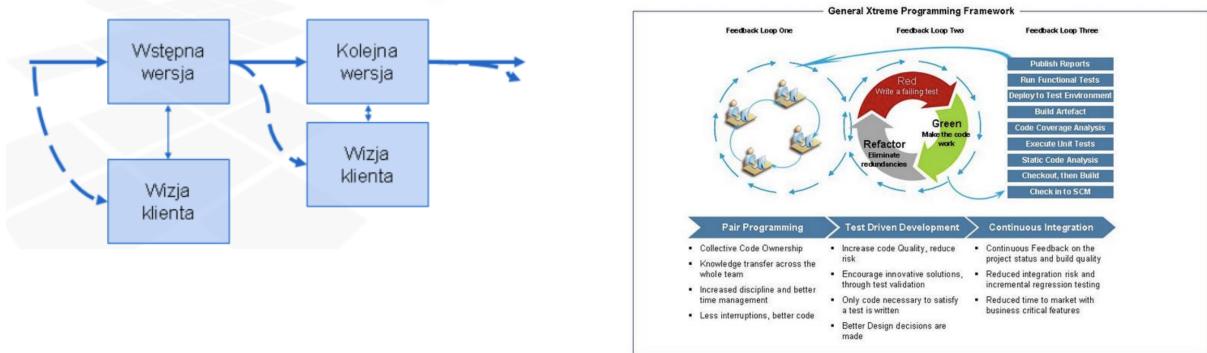
Manifesto for Agile Software Development:

- Individuals and interactions → Teamwork and responsibility
- Working software → Business value
- Customer collaboration → Partnership elaboration
- Responding to change → Prepare for change

Lightweight (XP, Scrum) or fuller (DSDM, AUP) approaches.

5.1 Programowanie ekstremalne - XP

Projekt informatyczny - szczelny systemem czterech zmiennych: daty dostarczenia, kosztu, liczby defektów oraz niekompletności funkcji.



- brak fazy projektowania i dokumentacji,
- krótka perspektywa planowania,
- silne założenie, że klient pracuje cały czas z zespołem.

Wartości

- **Komunikacja** - przede wszystkim werbalna.
- **Prostota** - rozpoczynamy od najprostszego rozwiązania, spełniającego wymagania; refaktoryzacja pozwala na adaptacje oprogramowania do zmian.
- **Sprzężenie zwrotne** - obejmuje kilka aspektów (system, klient, zespół).
- **Odwaga** - potrzebna by: od razu produkować kod; refaktoryzować; wyrzucić zbędny kod.
- **Szacunek** - do pracy i czasu innych; między członkami zespołu.

Struktura zespołu

- **Role podstawowe:** programiści, klient,
- **Role pomocnicze:** tester, coach, tracker.

User Stories

- opisują **funkcje systemu** z punktu widzenia użytkownika,
- ważne by miały wartość dla klienta,
- powinny być testowane.

Gra planistyczna

- *pisanie* user story (klient),
- *oszacowanie* user story (informatycy),
- *dzielenie* user story/wybór zakresu iteracji (klient).

Zapewnianie jakości

- prostota,
- unikanie optymalizacji,
- Test Driven Development - TTD,
- automatyczne testowanie,
- refaktoryzacja.

Testy akceptacyjne

- pochodzą od klienta (w ten sposób dokładnie określa, zachowanie systemu),
- najlepiej gdy mogą być wykonywane automatycznie (tester).

Programowanie parami

- zaleca się, by całość kodu pisana była w parach,
- częste zmiany w parach,
- wspólny standard kodowania,
- kod jest własnością całego zespołu,
- niezbędny system kontroli wersji.

5.2 SCRUM

Metoda przy użyciu której ludzie mogą z powodzeniem rozwiązywać **złożone problemy adaptacyjne**, by w sposób produktywny i kreatywny wytwarzać produkty o najwyższej możliwej wartości.

- lekki;
- łatwy do zrozumienia;
- bardzo trudny do opanowania.

Trzy filary teorii SCRUMa:

- **Adaptacja** - powinna być ciągła. Korekta musi być wykonana jak najszybciej, by ograniczyć dalsze następstwa problemów.
- **Przejrzystość** - wszystkie istotne aspekty procesu muszą być widoczne dla osób odpowiedzialnych za osiągane rezultaty.
- **Inspekcja** – poddawane regularnej inspekcji są zarówno scrumowe artefakty jak i postępy prac.

5.2.1 Role

Właściciel Produktu

- odpowiedzialny za maksymalizację wartości produktu i pracy Zespołu Deweloperskiego,
- jedyna osoba zarządzająca Rejestrem Produktu, tzn:
 - jasne artykułowanie elementów Rejestru Produktu, określanie ich kolejności w sposób zapewniający osiąganie założonych celów i misji;
 - zapewnianie dostępności i przejrzystości Rejestru Produktu dla wszystkich i to, że dobrze opisuje czym Zespół Scrumowy będzie się zajmował.

Zespół deweloperski

- złożony z profesjonalistów (3-9 osób); samorganizujący się, wielofunkcyjny.
- ma za zadanie dostarczenie (na zakończenie każdego Sprintu), gotowego do wydania Przyrostu produktu,
- nie przewiduje tytułów innych niż „Deweloper”,
- odpowiedzialność za wykonywaną pracę ponosi cały Zespół; brak podziału na podzespoły.

Scrum Master

- odpowiedzialny za to, by Scrum był rozumiany i stosowany,
- upewnia się, że Zespół Scrumowy stosuje się do założeń teorii Scruma, jego praktyk i regul postępowania.
- wspiera zarówno Właściciela Produktu, jak i Zespół Deweloperski.

5.2.2 Artefakty

Rejestr Produktu

uporządkowana lista wszystkiego, co może być potrzebne w produkcie oraz jedyne źródło wymaganych zmian. Odpowiedzialny za RP jest Właściciel Produktu. Elementy posiadają atrybuty: opis, kolejność i oszacowanie (estymację).

Rejestr Sprintu

podzbiór elementów RP wybranych do Sprintu rozszerzony o plan dostarczenia Przyrostu produktu. RS definiuje pracę, jaką ZD wykona by przekształcić elementy RP w „Ukończony” Przyrost. RS jest dobrze widocznym, tworzonym w czasie rzeczywistym obrazem pracy, jaką ZD planuje wykonać w trakcie Sprintu. RS należy tylko i wyłącznie do ZD.

Monitorowanie postępów Sprintu

możliwe w każdym momencie Sprintu (Codzienny Scrum).

Przyrost

suma wszystkich elementów RP zakończonych podczas wszystkich sprintów. Na koniec Sprintu nowy Przyrost musi być „Ukończony”.

Definicja Ukończenia

Aby zapewnić przejrzystość, wszyscy członkowie danego zespołu muszą mieć wspólne pojmowanie, co to znaczy, że praca jest skończona. W miarę dojrzewania Zespołu Scrumowego oczekuje się, że ich Definicja Ukończenia będzie zawierała coraz bardziej rygorystyczne kryteria zapewniania jeszcze wyższej jakości.

5.2.3 Zdarzenia

Cztery formalne punkty (ograniczone czasowo, wprowadzające regularność) przeprowadzania inspekcji i dokonania korekty (adaptacji) (każde oprócz Sprintu).

Sprint – serce Scruma

- stale przez okres trwania prac ograniczenie czasowe (≤ 1 mies)
- podczas Sprintu wytwarzany jest Przyrost ukończonej, używalnej i potencjalnie gotowej do wydania funkcjonalności,
- niedozwolone są zmiany, które wpłyną na cel Sprintu,
- niezmienny skład Zespołu Deweloperskiego i jego cel jakościowy.

Przerwanie Sprintu

- tylko Właściciel Produktu ma prawo to zrobić;
- w przypadku dezaktualizacji celu Sprintu,
- zużywa zasoby, bo powoduje przegrupowanie podczas kolejnego Planowania Sprintu.

Planowanie Sprintu - 8h/mies

- Część pierwsza: *Co będzie zrobione w tym Sprincie?* Wejście:
 - Rejestr Produktu;
 - ostatni przyrost;
 - przewidywana pojemność ZD;
 - ostatnie odczyty wydajności;Wyjście:
 - elementu Rejestru Produktu wybrane do zaimplementowania;
 - cel Sprintu.
- Część druga: *Jak wybrana praca będzie wykonana?*
 - zwykle rozpoczęcie od stworzenia projektu systemu i planu prac niezbędnych do przetworzenia elementów Rejestru Produktu w działający Przyrost produktu.
 - zanim Planowanie Sprintu dobiegnie końca, ZD powinien móc wytłumaczyć Właścicielowi Produktu i Scrum Masterowi, w jaki sposób ma zamiar pracować, organizując się samodzielnie, by osiągnąć Cel Sprintu i wytworzyć oczekiwany Przyrost.

Codzienny Scrum - 15 min/d

- Co zostało wykonane od ostatniego potkania?
- Co zostanie wykonane przed kolejnym spotkaniem?
- Jakie przeszkody stoją na drodze?

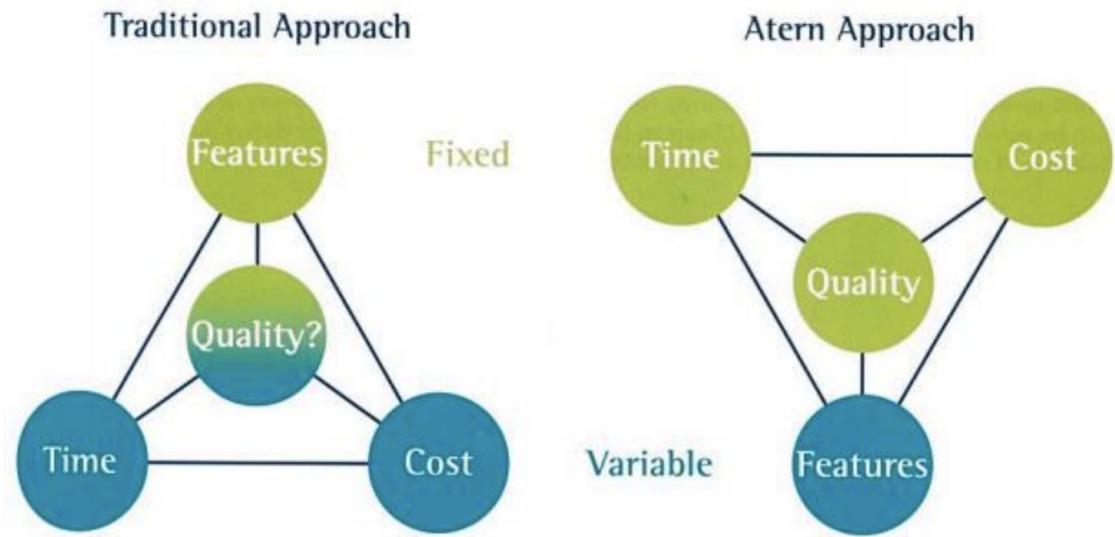
Przegląd Sprintu - 4h na zakończenie Sprintu

- WP stwierdza, które funkcjonalności zostały „Ukończone”, a które nie;
- ZD omawia, co poszło dobrze w trakcie Sprintu; jakie były problemy i jak je rozwiązano;
- ZD prezentuje „Ukończoną” pracę i odpowiada na pytania dotyczące Przyrostu,
- WP omawia Rejestr Produktu w aktualnej jego postaci. Przewiduje termin zakończenia prac.
- Cała grupa omawia kolejne kroki.

Retrospektyna Sprintu - okazja do przeprowadzenia inspekcji swoich działań i opracowania planu usprawnień.

- Sprawdzenie, co działo się w ostatnim Sprincie, biorąc pod uwagę ludzi, zależności, procesy i narzędzia;
- Zidentyfikowanie i uporządkowanie istotnych elementów, które sprawdziły się w działaniu oraz tych, które kwalifikują się do poprawy;
- Stworzenie planu wprowadzania w życie usprawnień sposobu wykonywania pracy przez Zespół Scrumowy.

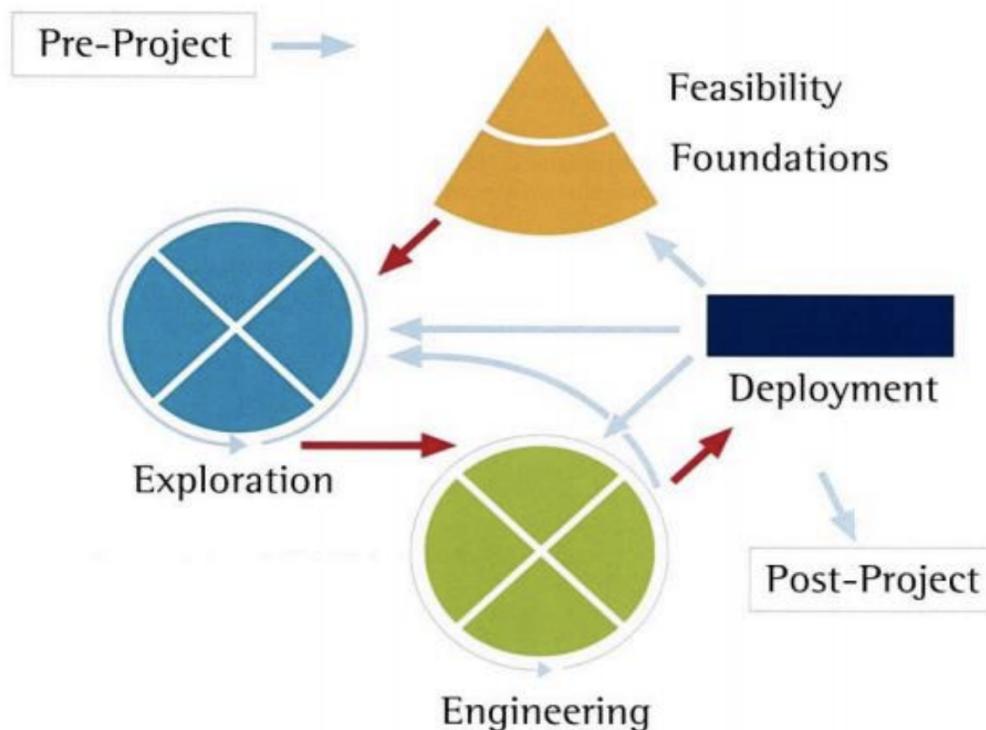
5.3 AGILE PM (DSDM Atern)



PROCESS - PEOPLE - PRODUCTS - PRACTISES

- Focus on the *business need*
- Deliver *on time*
- *Collaborate*
- Never compromise *quality*
- Build incrementally from *firm foundations*
- Develop *iteratively*
- *Communicate continuously* and clearly
- Demonstrate *control*

5.3.1 Fazy projektu



Pre-project

- problem biznesowy; identyfikacja Business Sponsor i Business Visionary,
- zakres, plan i zasoby na fazę Feasibility.

Feasibility

- ustalić wykonalność rozwiązania problemu biznesowego,
- identyfikacja potencjalnych zysków; zarys możliwych podejść do rozwiązania,
- pierwsze estymaty czasowe i kosztowe.

Foundation

- wysoko poziomowe wymagania,
- identyfikacja wspieranych procesów biznesowych,
- podstawy architektury systemu; sposób zapewnienia wysokiej jakości.

Exploration

- uszczegóławianie wymagań; iteracyjnie działające rozwiązanie,
- zarys możliwych podejść do rozwiązania.

Engineering

- rozwijanie rozwiązania z fazy Exploration.

Deployment

- potwierdzenie wydajności rozwiązania,
- dostarczenie (iteracyjnie) rozwiązania,
- dostarczenie potrzebnej dokumentacji.

5.3.2 Role

Business Sponsor

- najwyższy rangą w projekcie; zapewnia finansowanie i zasoby,
- właściciel tzw. przypadku biznesowego.

Business visionary

- definiuje wizję projektu i komunikuje ją,
- ma zapewnić współpracę na wszystkich poziomach projektu,
- wkład w najważniejsze wymagania; arbiter w przypadku sporów.

Project manager

- monitoruje postęp projektu; motywuje zespoły, zatrudnia specjalistów,
- wysoko poziomowe planowanie harmonogramu, zarządzanie ryzykiem w projekcie.

Technical coordination

- definiuje środowisko pracy,
- doradza w sprawach technicznych, pilnuje standardów,
- zajmuje się wymaganiami niefunkcjonalnymi.

Team Leader

- skupiony na zespole, pilnuje dostarczania poszczególnych komponentów na czas,
- raportuje postęp do PM, prowadzi spotkania zespołowe.

Business Ambassador

- rola biznesowa w zespole deweloperskim,
- dzieli się perspektywą biznesową z zespołem, dostarcza scenariusze biznesowe,
- tworzy dokumentacje użytkownika.

Business Analyst

- komunikacja między biznesem a zespołem deweloperskim,
- dystrybucja i wstępna akceptacja dokumentów biznesowych.

Solution Developer

- skupiony na dostarczeniu rozwiązania,
- modele potrzebne do dostarczenia rozwiązania, dokumentacja.

Solution Tester

- definiuje scenariusze testowe, test easy,
- komunikuje wyniki testów do TL, pracuje z BAs nad testami akceptacyjnymi.

5.3.3 Produkty

Levels of priority - MoSCoW

- Must Have
- Should Have
- Could Have
- Won't Have this time

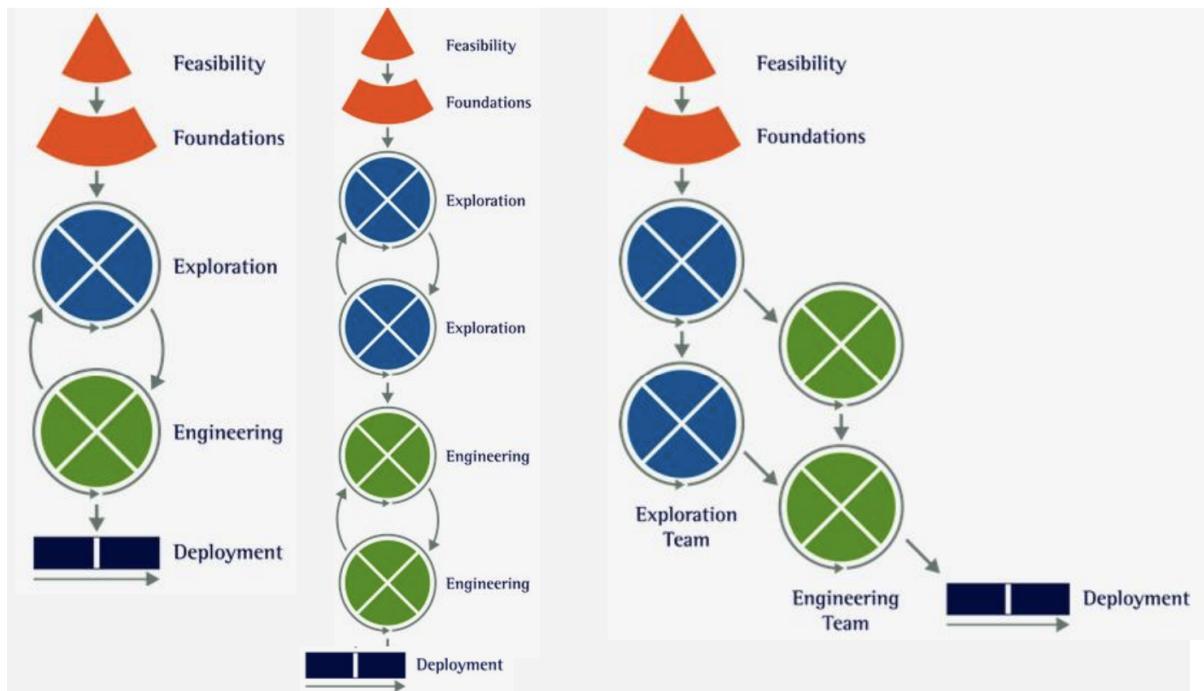
Fazy TIMEBOXu:

- **Kick-off** – krótka sesja, która ma pomóc zrozumieniu celu timeboxa,

- **Investigation** – szczegóły wszystkich produktów, które mamy wykonać,
- **Refinement** – kodowanie i testowanie,
- **Consolidation** – spinanie całości.

Iterative development:

- **Identify**: zespół definiuje cel
- **Plan**: kto powinien zrobić co
- **Evolve**: wykonywanie zaplanowanych czynności
- **Review**: sprawdzanie rezultatów

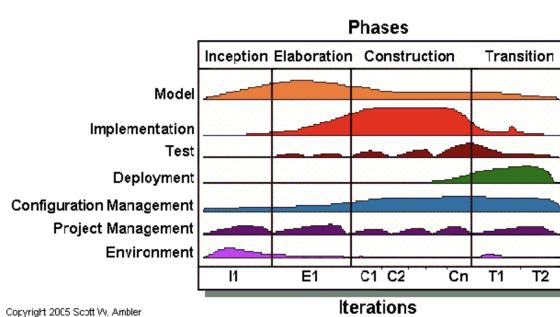


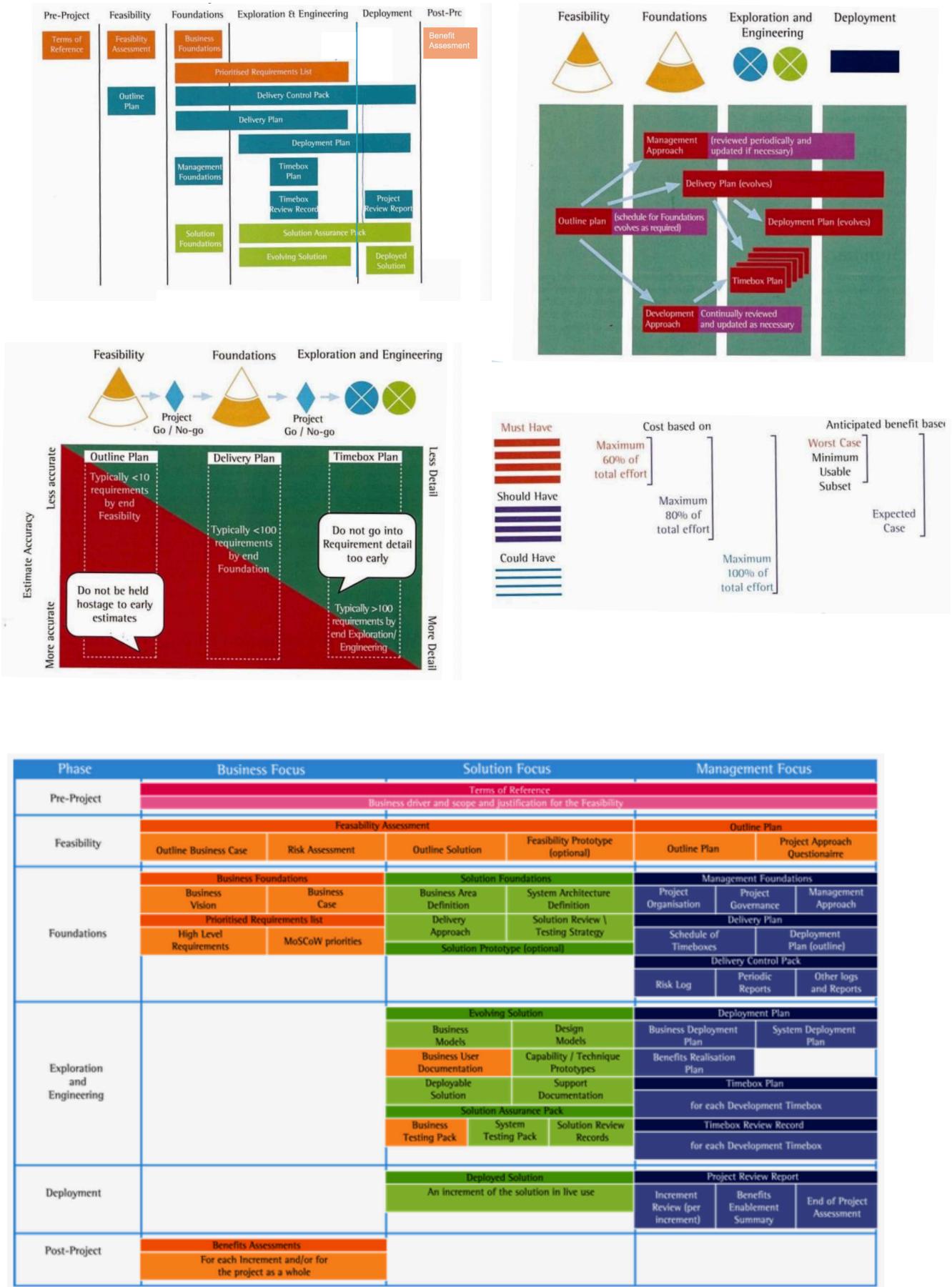
5.4 AUP - Agile Unified Process

- uproszczona wersja Rational Unified Process,
- stosuje zwinne techniki takie jak TDD, refactoring,
- seryjny w dużej skali, iteracyjny w małej.

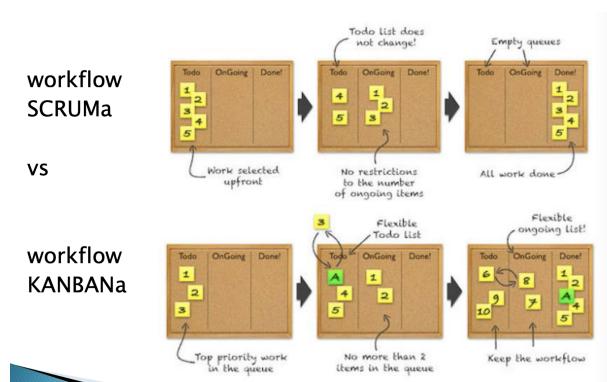
Zasady AUP

- twój zespół wie, co robi;
- prostota;
- zwinność;
- skupienie się na istotnych aktywnościach;
- niezależność od narzędzi;
- możliwość adaptacji.





5.5 KANBAN



- ciągły przepływ produktu przez system produkcyjny.
- podstawa systemów produkcyjnych Toyoty i pochodnych
- system pull sterowany jest przez składane przez odbiorcę zamówienie, a nie ogólny, arbitralny plan produkcji.
- odnosi się do etapowości procesu wytwarzania oprogramowania, przynajmniej trzy stany pracy — do zrobienia, w trakcie, gotowe.

Sześć reguł kanbana:

- odbiorca przetwarza dokładnie tyle elementów, ile opisane jest na karcie kanban;
- dostawca wytwarza dokładnie tyle elementów, ile opisane jest na karcie kanban;
- żaden element nie jest wytwarzany lub przekazywany pomiędzy stanowiskami bez karty kanban;
- karta kanban musi towarzyszyć każdemu elementowi czy półproduktowi przetwarzanemu w ramach systemu;
- elementy wadliwe lub występujące w niewłaściwych ilościach, nigdy nie są przekazywane w dół procesu;
- limity obowiązujące na każdym z etapów (fizyczna ilość kart kanban) są stopniowo obniżane aby redukować zapasy i odkrywać nieefektywności procesów produkcji, dążąc do ich doskonalenia.

5.5.1 Kanban vs Scrum

Sposób pracy	
Scrum	Kanban
Rytmiczność pracy	Plynność pracy
Synchronizacja aktywności	Aktywności synchroniczne lub asynchroniczne
Skupienie na celu, minimalizacja przełączania między zadaniami	Niski czas odpowiedzi systemu na zmiany
Pewność realizacji aktywności związanych z Agile SD	Möglichkeit doboru czynności dopasowanych do środowiska
Pełne zaangażowanie zespołu, wiedza ogólna i współpraktywność	Möglichkeit pełnego wykorzystania specjalistów
Trudność w pełnym wykorzystaniu zespołu ze względu na skokowy sposób pracy	Möglichkeit pełnego wykorzystania zespołu ze względu na płynny sposób pracy
Niebezpieczeństwo przeładowania poprzez stosowanie złe wyznaczonych miar podczas planowania, penalizacji związanej z niedostarczeniem funkcjonalności	Niebezpieczeństwo przeładowania poprzez nadmierną optymalizację wykorzystania zespołu i restrykcyjności związanej z niedostarczeniem funkcjonalności
Utrzymanie wydajności zespołu w dalszej perspektywie wymaga znacznego doświadczenia	Utrzymanie wydajności zespołu w dalszej perspektywie wymaga mniejszego doświadczenia.

Definicja ukończenia	
Scrum	Kanban
Wymaga, aby ukończenie zadania oznaczało dla wszystkich to samo	Sama metoda nie identyfikuje definicji ukończenia
Bardziej szczegółowe implementacje korzystają z listy warunków jakie praca musi spełnić, aby może ją uznać za ukończoną	Bardziej szczegółowe implementacje określają warunki uznania zadania za ukończone poprzez przejście przez wszystkie procesy systemu
Estymacja, planowanie i metryki	
Scrum	Kanban
Określony sposób zarządzania zadaniami do realizacji w rejestrze produktu	Zadania do realizacji mogą pochodzić z różnych źródeł, a także być tworzone w ramach procesu
Estymacja zadań prowadzona przez zespół, najczęściej relatywna	Brak wymagań względem estymacji, ewentualnie prosty podział typów zadań
Estymacja zwiększa poziom zrozumienia zadań przez cały zespół	Brak konieczności poświęcania czasu na szczegółową estymację zadań
Planowanie jako ilość pracy realizowanej w trakcie jednej iteracji	Planowanie oparte o przewidywany czas zakończenia zadania
Zespół zobligowany do realizacji całości zaplanowanej pracy w trakci iteracji	Wykrywanie zadań przekraczających średni czas realizacji
Zespół bierze czynny udział w planowaniu pracy	Zespół niekoniecznie musi brać udział w planowaniu pracy
Postęp prac w ramach iteracji jest monitorowany na wykresach spalania	Postęp prac monitorowany na tablicy kanban oraz poprzez analizę średnich czasów wykonania
Velocity - jedna miara dotycząca różnych zadań	Lead time może być określany dla zadań o różnej złożoności
Velocity - określa ilość pracy w okresie czasu	Lead time - określa czas wykonania pewnej ilości pracy w postaci jednego zadania.
Diagram zmiany velocity	Diagram zmiany lead time
Brak konieczności posiadania zdefiniowanego przepływu zadań w ramach iteracji. Jeśli jest zdefiniowany można korzystać z Cumulative flow diagram	Cumulative flow diagram udostępniający informacje o pracy w toku i wpływie decyzji na lead time w sposób ciągły
Możliwość planowania pracy w dalszym terminie	Możliwość gwarantowania czasu obsługi zadania
Możliwość analizowania postępu w dalszym terminie	Dostosowanie procesu charakterystyki obsługiwanych zadań i warunków nałożonych na czas ich realizacji

5.6 SCRUM + KANBAN = SCRUM-BAN

Kiedy używać Scrum-bana?

- W projektach typu maintenance
- W projektach typu helpdesk
- W projektach z często dorzucanymi User stories lub często zgłaszanymi błędami

	Scrum	Scrumban
Board/Artifacts	board, backlogs, burn-downs	board only
Ceremonies	daily scrum, sprint planning, sprint review, sprint retrospective	daily scrum (planning, review and retrospective as needed)
Iterations	yes (sprints)	no (continuous flow)
Estimation	yes	no (similar size)
Teams	must be cross-functional	can be specialized
Roles	Product Owner, Scrum Master, Team	Team + roles needed
Teamwork	collaborative as needed by task	swarming to achieve goals
WIP	controlled by sprint content	controlled by workflow state
Changes	should wait for the next sprint	added as needed on the to-do board
Product Backlog	list of prioritized end estimated stories	just in time cards
Impediments	dealt with immediately	avoided

6 Wymagania

- Wymagania to **opis funkcji** (usług) i ograniczeń dla systemu.
- Wymagania nie opisują jak system ma działać, a **co ma wykonywać**.
- Definiowane na **wczesnych etapach rozwoju** systemu jako specyfikacja tego, co ma być implementowane.
- Inżynieria wymagań to proces pozyskiwania, analizowania, dokumentowania oraz weryfikowania wymagań dla projektowanego systemu.

6.1 Klasyfikacja wymagań

- **Funkcjonalne** - czynność, zadanie.
- **Pozafunkcjonalne** - technikalia mierzone metrykami.

Klasyfikacja wymagań - **FURPS** - Functionality, Usability, Reliability, Performance, Security.

6.1.1 Wymagania funkcyjne - „System powinien...”

- **Zalety**: łatwość spisywania,
- **Wady**: słaba czytelność, trudne sprawdzanie kompletności i spójności.
- **Przypadki użycia**: łatwość spisywania, czytelność, łatwość zrozumienia; forma ustukturalizowana.
- **Historyjki użytkownika** - Who? What? Why?

Cecha **INVEST**:

- **Independent** - zależności powodują problem z estymacją),
- **Negotiable**,
- **Valuable**,
- **Estimable**,
- **Small** (jeden sprint),

- **Testable.**

6.1.2 Wymagania pozafunkcjonalne

- Ograniczenia usług lub funkcji, np. czasowe, procesu rozwoju oprogramowania, standardy.
- Każda cecha to **zbiór atrybutów**.
- Stanowią **niezbędne uzupełnienie wymagań funkcjonalnych** dla oprogramowania,
- Problemy z oprogramowaniem wskazują na silną potrzebę precyzyjnego definiowania atrybutów (charakterystyk) dla tworzonych produktów programistycznych.

Niezawodność - zdolność do spełnienia i utrzymywania określonych wymagań stabilności, przy spełnieniu określonych warunków oraz w określonych ramach czasowych.

- Dojrzałość
- Odporność na błędy
- Zdolność do odtworzenia

Wydajność - opisuje powiązania między poziomem wydajności oprogramowania, a wykorzystywanyimi zasobami przy spełnieniu określonych warunków.

- Wykorzystanie czasu
- Wykorzystanie zasobów

Użyteczność - opisuje nakład pracy niezbędny do swobodnego posługiwania się oprogramowaniem.

- Łatwość zrozumienia
- Łatwość nauki
- Operatywność

Łatwość konserwacji - opisuje nakład pracy niezbędny do wprowadzenia zmian do oprogramowania.

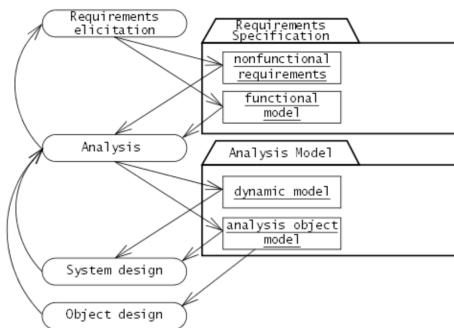
- Łatwość analizy
- Łatwość wprowadzania zmian
- Stabilność
- Łatwość testowania

Przenośność - opisuje zdolność oprogramowania do przenoszenia między różnymi środowiskami/platformami.

- Łatwość adaptacji
- Zgodność
- Łatwość instalacji
- Łatwość zastąpienia

6.2 Analiza wymagań/analiza obiektowa

- Celem jest stworzenie modelu systemu, zwanego **modelem analitycznym**.
- Wysiłek uczestników projektu skupia się na strukturalizowaniu i formalizowaniu zabranych wcześniej wymagań.



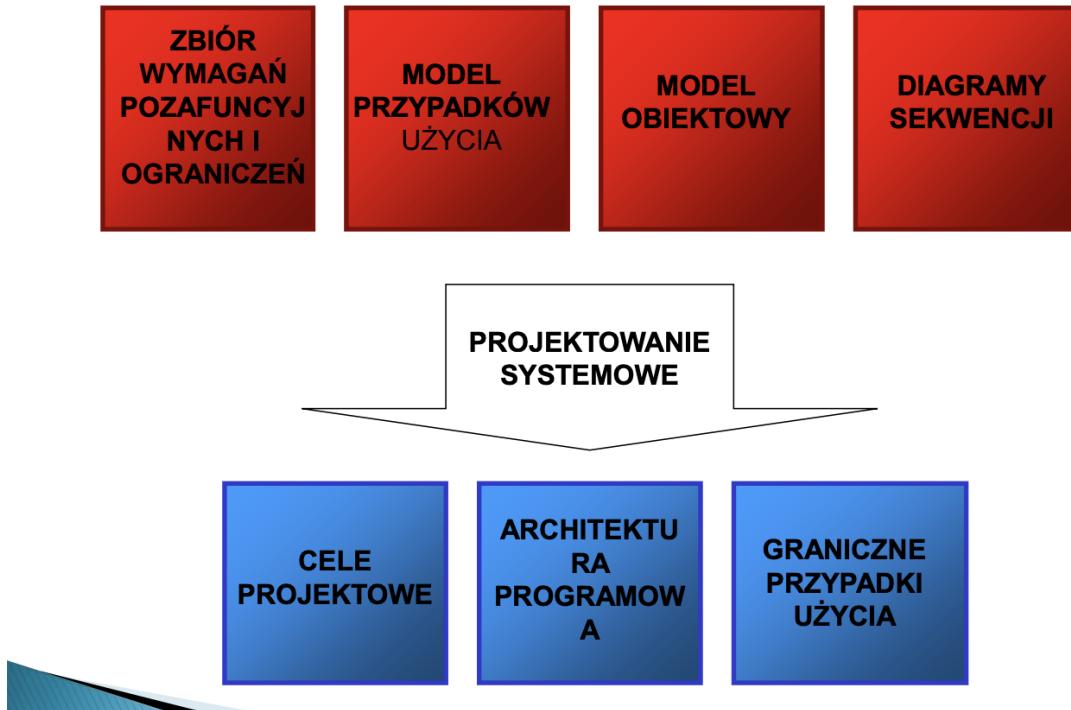
- **Model analityczny** – reprezentuje tworzony system z perspektywy użytkownika; opis co system powinien robić.
- **Analityczny model obiektowy** – odzwierciedla indywidualne koncepcje korzystania z systemu, ich właściwości i relacje między nimi (diagramy klas).
- **Model dynamiczny** - koncentruje się na zachowaniu systemu(diagramy sekwencji i stanów).
- **Obiekty encji** – reprezentują trwałą informację potwarzaną przez system.
- **Obiekty brzegowe** – odzwierciedlają interakcje między aktorami a systemem.
- **Obiekty sterujące** – odpowiedzialne są za realizację przypadków użycia.
- **Relacja dziedziczenia** umożliwia hierarchiczne organizowanie koncepcji.
- **Generalizowanie** - aktywność identyfikowania abstrakcyjnych koncepcji na podstawie przykładów i konkretyzacji.
- **Specjalizowanie** - aktywność odwrotna, czyli identyfikowanie koncepcji bardziej specyficznych na podstawie koncepcji wysokopoziomowej.

7 Projektowanie systemu

Transformowanie modelu analitycznego w model projektu systemu.

Etapy projektowania systemu:

- rozpoznawanie celów projektowych,
- projektowanie wstępnych dekompozycji,
- doskonalenie dekompozycji stosownie do celów projektowych.



7.1 Podstawowe pojęcia i koncepcje.

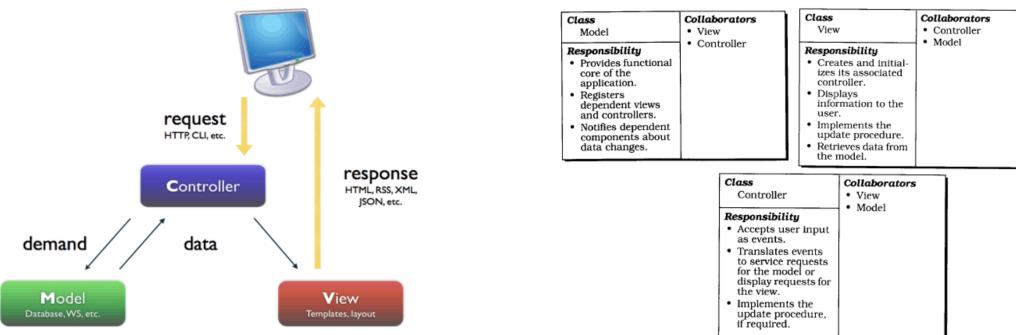
- **Podsystem** - wymienna część systemu, posiadająca dobrze zdefiniowane interfejsy i hermetyzującą stan oraz zachowanie składających się na nią klas.
- Dwa główne typy komponentów: **logiczny** i **fizyczny**.
- **Usługa** jest zbiorem powiązanych operacji podporządkowanych realizacji wspólnego celu.
- **Sprzężeniem** w zbiorze podsystemów nazywamy stopień ich **wzajemnego uzależnienia**. (MINIMALIZACJA)
- Spoistość podsystemu jest miara **uzależnienia jego własnych klas**. (MAKSYMALIZACJA)
- **Warstwa** - zgrupowanie podsystemów oferujących powiązane usługi.
- Efektem **dekompozycji hierarchicznej** jest uporządkowany zbiór warstw.
- **Architektury warstwowa**: otwarta i zamknięta (np. ISO/OSI, TCP/IP).

7.2 Wzorce architektoniczne - poziom integracji komponentów

7.2.1 MVC: model-widok-kontroler

Model zawiera korową funkcjonalność. Widoki wyświetlają funkcjonalności. Kontroler obsługuje żądanie użytkownika. Kontroler z widokami tworzą UI aplikacji.

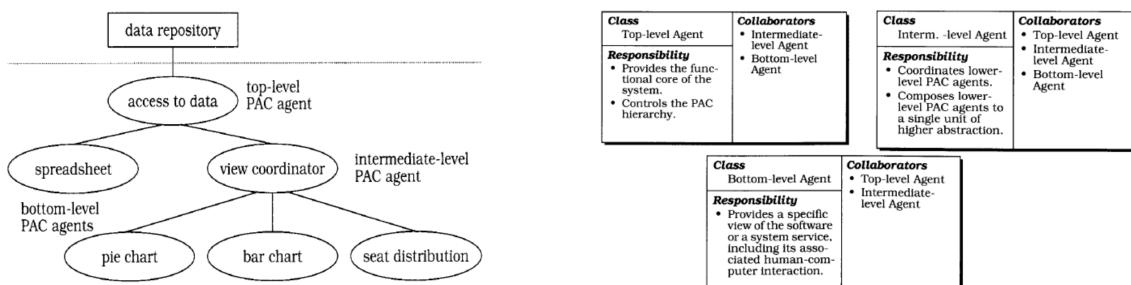
Zastosowania: Smalltalk, Java/Swing.



7.2.2 PAC: prezentacja-abstrakcja-kontrola

Hierarchie kooperujących agentów, podzielonych na trzy komponenty: prezentacji, abstrakcji kontroli.

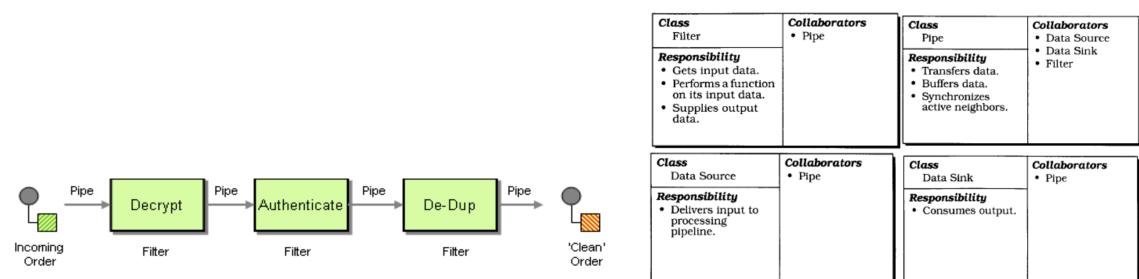
Zastosowania: Network Trafic Management (gathering traffic data, displaying various user-configurable views of the whole network).



7.2.3 Architektura filtry i potoki

Pozwala na uporządkowanie systemu, który przetwarza strumienie danych. Każdy krok przetwarzania jest zamknięty w filtrze. Dane są przesyłane za pomocą potoków. Każdy z podsystemów realizuje przetwarzanie danych otrzymanych od innych podsystemów.

Zastosowania: Unix, WEB, Servlet, Numerical Analysis (filters and data extractions).



7.2.4 Tablica(blackboard)

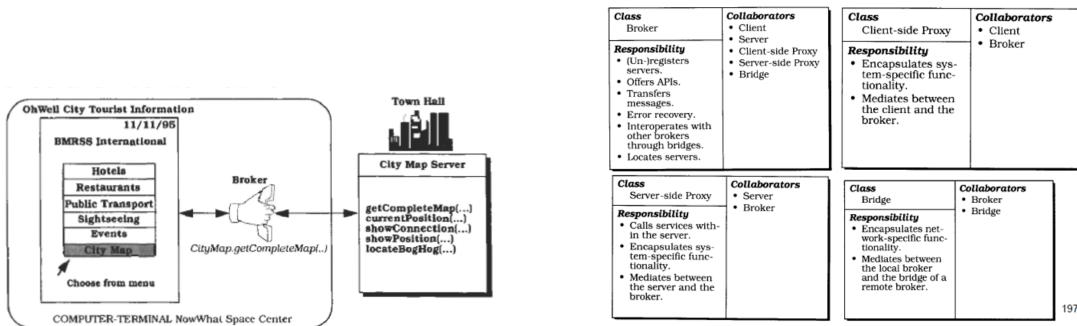
Użyteczna w systemach, gdzie nie są znane deterministyczne rozwiązania danego problemu. W przypadku tablicy kilka wyspecjalizowanych systemów łączy swoja wiedzę w taki sposób, żeby stworzyć częściowe lub przybliżone rozwiązanie problemu.

Zastosowania: working memory, repository data.

Class Blackboard	Collaborators
Responsibility <ul style="list-style-type: none">• Manages central data	
Class Knowledge Source	Collaborator <ul style="list-style-type: none">• Blackboard
Responsibility <ul style="list-style-type: none">• Evaluates its own applicability• Computes a result• Updates Blackboard	
Class Control	Collaborators <ul style="list-style-type: none">• Blackboard• Knowledge Source
Responsibility <ul style="list-style-type: none">• Monitors Blackboard• Schedules Knowledge Source activations	

7.2.5 Broker

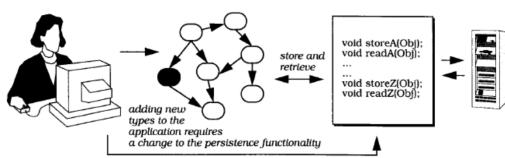
Pozwala na uporządkowanie rozproszonych systemów podzielonych na komponenty współpracujące ze sobą za pomocą zdalnego wywoływania serwisu. Komponent brokera odpowiedzialny jest za koordynację komunikacji.



197

7.2.6 Reflection

Dostarcza mechanizm pozwalający na dynamiczną zmianę zachowania i struktury systemu.



Class Base Level	Collaborators <ul style="list-style-type: none">• Meta Level
Responsibility <ul style="list-style-type: none">• Implements the application logic.• Uses information provided by the meta level.	
Class Meta Level	Collaborators <ul style="list-style-type: none">• Base Level
Responsibility <ul style="list-style-type: none">• Encapsulates system internals that may change.• Provides an interface to facilitate modifications to the meta level.	
Class Metabject Protocol	Collaborators <ul style="list-style-type: none">• Meta Level• Base Level
Responsibility <ul style="list-style-type: none">• Offers an interface for specifying changes to the meta level.• Performs specified changes	

Zastosowania: WWW.

7.2.7 Sieciowe

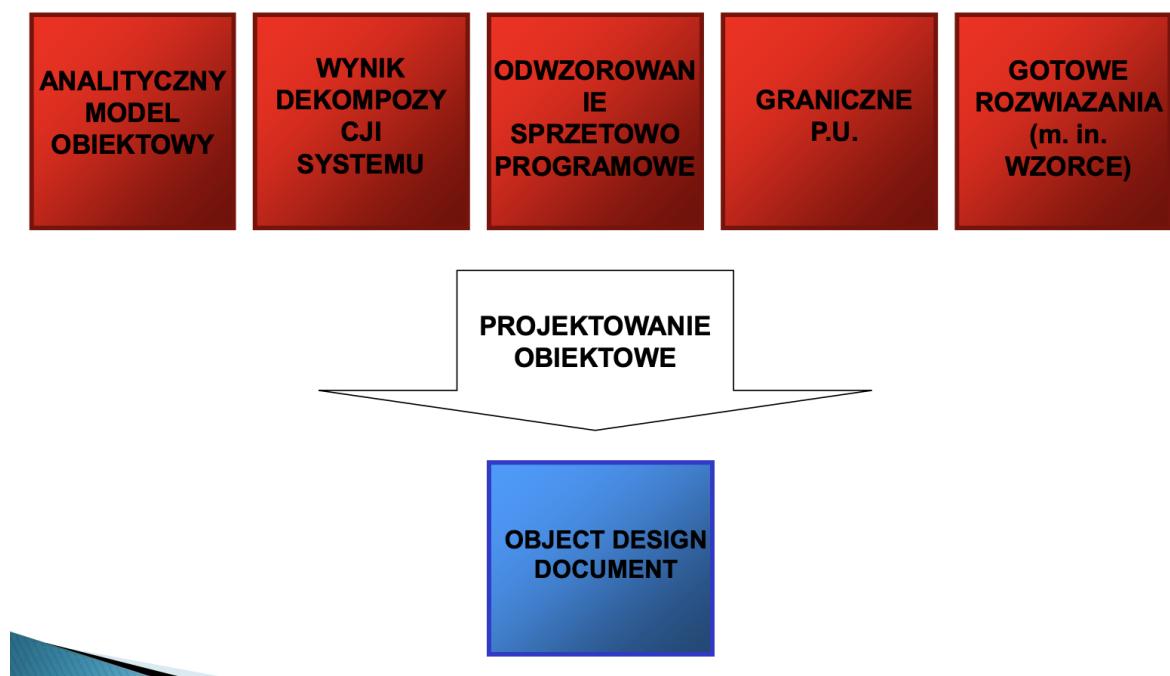
- **Architektura klient-serwer** - podział systemu na dostawce usług (serwer) oraz ich odbiorców (klientów).
- **Architektura peer-to-peer** - każdy z podsystemów może spełniać obie funkcje (klient/serwer).

7.2.8 Wzorce architektoniczne - wady

- Patterns do not lead to direct code reuse.
- Individual Patterns are deceptively simple.
- Composition of different patterns can be very complex.
- Teams may suffer from pattern overload.
- Patterns are validated by experience and discussion rather than by automated testing.
- Integrating patterns into a software development process is a human-intensive activity.

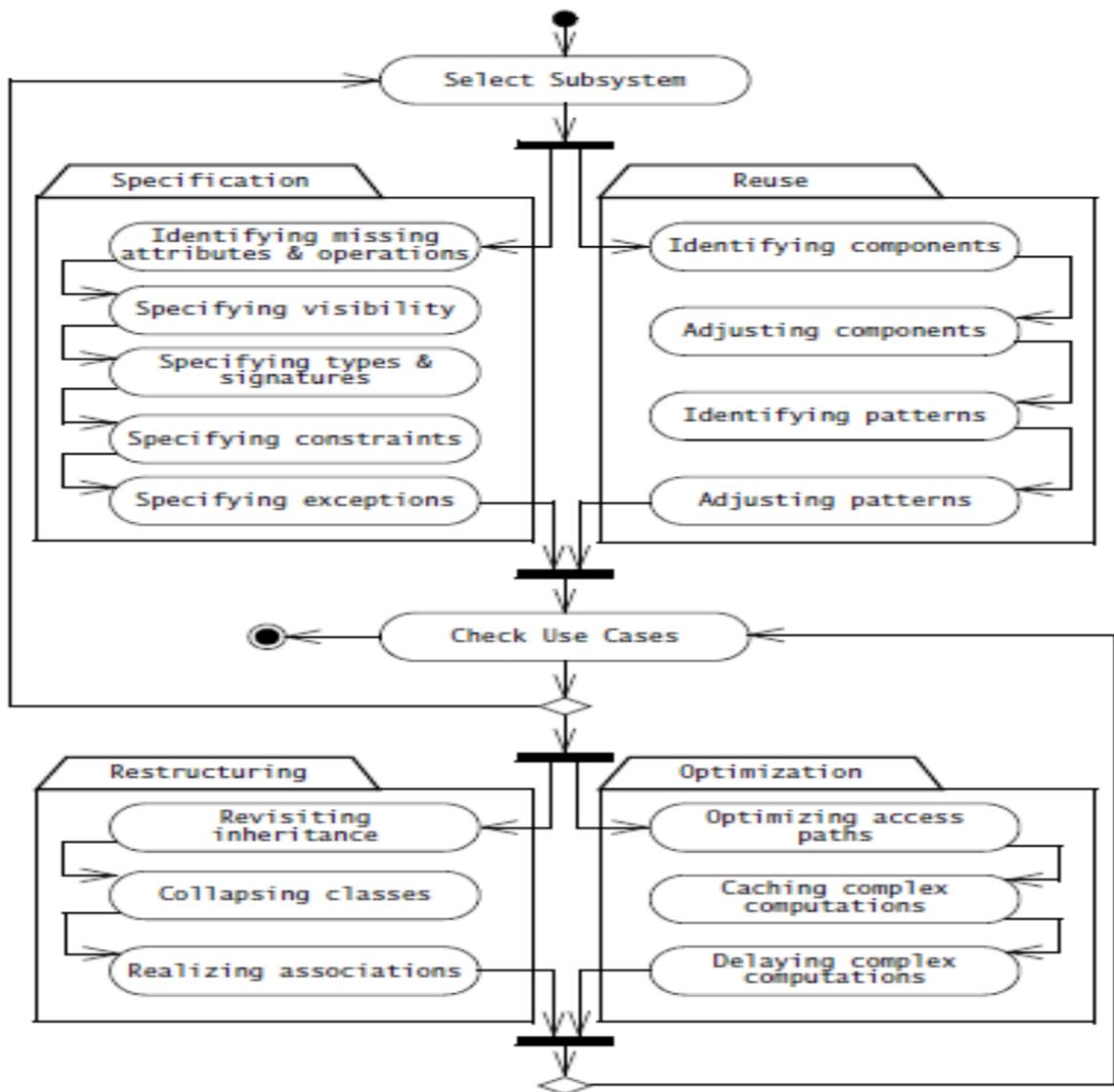
8 Projektowanie obiektów

Ma na celu wypełnienie luki między obiektami dziedziny aplikacyjnej a komponentami wybranymi na etapie projektowania systemu.



Etapy projektowania obiektów

- wykorzystanie gotowych rozwiązań, którymi są zarówno gotowe produkty (komponenty) jak i wzorce projektowe;
- specyfikowanie usług;
- restrukturyzacja modelu obiektowego;
- optymalizacja modelu obiektowego;



Koncepcje wielokrotnego wykorzystywania gotowych rozwiązań:

- **obiekty aplikacyjne** - reprezentują koncepcje problemowe związane z tworzonym systemem.
- **obiekty realizacyjne** - reprezentują komponenty nie mające odpowiedników w dziedzinie aplikacyjnej, na przykład bazy danych czy obiekty interfejsu użytkownika.
- **dziedziczenie implementacyjne** - ma miejsce jeśli sięgamy po dziedziczenie z zamiarem wykorzystania gotowego kodu, mimo różnic koncepcyjnych pomiędzy powiązanymi klasami.
- **dziedziczenie specyfikacyjne** - ma odzwierciedlenie w taksonomii klas (reprezentuje podtypowanie).
- **delegowanie implementacji** - zamiast implementować set jako nadpisywanie metod hashtable, implementujemy go jako set korzystający z instancji hashtable z własnymi metodami. Rozwiązuje problemy dziedziczenia implementacyjnego: rozszerzalność, podtypowanie.
- **zasada zastępowania Liskov** - 'Jeśli obiekt klasy S może stać się substytutem obiektu klasy T w dowolnym miejscu kodu, w którym oczekiwany jest obiekt klasy T , to klasa S jest podtypem klasy T .'

- **wzorce projektowe** (obiektowe);

8.1 Wzorce projektowe - poziom interakcji między klasami

Wzorzec opisuje problem, który powtarza się wielokrotnie w danym środowisku, oraz podaje istotę jego rozwiązania.

- Czy typowe problemy można rozwiązać w powtarzalny sposób?
- Czy te problemy można przedstawić w sposób abstrakcyjny, tak aby były pomocne w tworzeniu rozwiązań w różnych konkretnych kontekstach?

- **Wzorce kreacyjne**

- abstrakcyjne metody tworzenia obiektów,
- uniezależnienie systemu od sposobu tworzenia obiektów.

- **Wzorce strukturalne**

- sposób wiązania obiektów w struktury,
- właściwe wykorzystanie dziedziczenia i kompozycji.

- **Wzorce behawioralne**

- algorytmy i przydział odpowiedzialności,
- opis przepływu kontroli i interakcji.

8.2 Koncepcje specyfikowania interfejsów

- implementator (realize class), ekstender (refine class) i użytkownik (use class) klasy,
- typy, sygnatury (wektory/krotki typów parametrów i typu wyniku) i widzialność (public, private, protected, packet),
- kontrakty: niezmienniki, warunki wstępne i warunki końcowe,
- język OCL (Object Constraint Language) – ograniczenia, zbiory, wielozbiory i ciągi; kwantyfikatory.

8.3 Aktywności specyfikowania interfejsów

- identyfikowanie brakujących atrybutów i operacji;
- definiowanie widzialności i sygnatur;
- specyfikowanie kontraktów;
- dziedziczenie kontraktów;

8.4 SOLID

- **Single responsibility principle**

- Klasa powinna mieć pojedynczą odpowiedzialność.
- Nigdy nie powinien istnieć więcej niż jeden powód do modyfikacji klasy.
- Im mniejsza i bardziej wyspecyfikowana klasa tym łatwiej ją nazwać.
- Łatwiejsze wprowadzanie zmian, testowanie i naprawa

Jak rozpoznać naruszenie SRP?

- Ilość linii kodu (Class:LOC > 250),
- Za dużo zależności, słaba spojność, dużo zagnieżdżeń,

- Opis lub nazwa wymaga “i”,
- Wymaga skomplikowanych testów,
- Modyfikacja może zepsuć inne testy.

- **Open/closed principle**

- klasa powinna być otwarta na rozbudowę, ale zamknięta do jej własnej modyfikacji,
- możemy dodawać nowe pola i metody, ale bez zmiany w wewnętrznej strukturze,
- zmiana istniejącej struktury może mieć wpływ na inne elementy,
- hermetyzacja, dziedziczenie, polimorfizm, delegaty,
- unikamy instrukcji warunkowych.

- **Liskov substitution principle**

- ‘Jeśli obiekt klasy S może stać się substytutem obiektu klasy T w dowolnym miejscu kodu, w którym oczekiwany jest obiekt klasy T, to klasa S jest podtypem klasy T.’

- **Interface segregation principle**

- Kilka konkretnych interfejsów jest lepszych niż jeden ogólny,
- Związki między klasami powinny być ograniczone do minimum,
- Klient klasy powinien mieć dostęp tylko do tych składowych klasy, których rzeczywiście potrzebuje,
- Moduły wysokiego poziomu nie powinny zależeć od modułów niskopoziomowych,
- Obie grupy modułów powinny zależeć od abstrakcji.

- **Dependency inversion principle** - zasada odwracania zależności.

- Software powinien zależeć od abstrakcji a nie od konkretyzacji,
- ‘Hollywood Principle’ - don’t call us, we’ll call you!.

8.5 Wybrane wzorce kreacyjne

Wzorzec	Schemat	
Singleton	<p>• Zapewnienie, że klasa posiada jedną instancję wewnątrz całej aplikacji</p> <p>• Stworzenie punktu dostępowego do tej instancji</p>	
Factory method	<p>• Zdefiniowanie interfejsu do tworzenia obiektów</p> <p>• Umożliwienie przekazania odpowiedzialności za tworzenie obiektów do podklas</p> <p>• Umożliwienie wyboru klasy i konstruktora użytego do utworzenia obiektu</p>	
Builder	<ul style="list-style-type: none"> • Odseparowanie sposobu reprezentacji i metody konstrukcji złożonych struktur obiektowych • Wykorzystanie jednego mechanizmu konstrukcyjnego do tworzenia struktur o różnej reprezentacji 	

8.6 Wybrane wzorce strukturalne

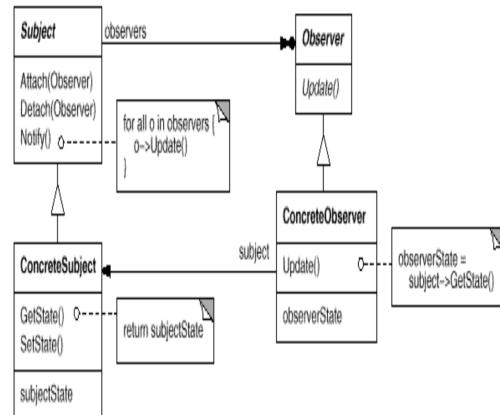
Wzorzec	Schemat
Adapter <ul style="list-style-type: none"> Umożliwia współpracę obiektów o niezgodnych typach Tłumaczy protokoły obiektowe 	<pre> graph LR Client --> Target[Target Request()] Target --- Adaptee[Adaptee SpecificRequest()] Adapter[Adapter Request()] -.-> SpecificRequest[SpecificRequest()] subgraph Implementation [Implementation] Target Adaptee Adapter end Implementation -- "(implementation)" --> Adapter </pre>
Proxy <ul style="list-style-type: none"> Dostarcza zamiennik obiektu w celu jego kontroli i ochrony Przezroczyste odsuniecie inicjalizacji obiektu w czasie 	<pre> graph TD Client[Client] -.-> Subject["<<interface>> Subject"] Client --> DoAction[DoAction()] Subject --> DoAction RealSubject[RealSubject DoAction()] -- delegate --> DoAction </pre>
Fasada <ul style="list-style-type: none"> Dostarczenie jednorodnego interfejsu wyższego poziomu do zbioru różnych interfejsów w systemie Ukrycie złożoności podsystemów przed klientem 	<pre> graph TD Facade[Facade] --> Subsystem[Subsystem] Subsystem --> Comp1[] Subsystem --> Comp2[] Subsystem --> Comp3[] Subsystem --> Comp4[] Subsystem --> Comp5[] </pre>

8.7 Wybrane wzorce behawioralne

Wzorzec

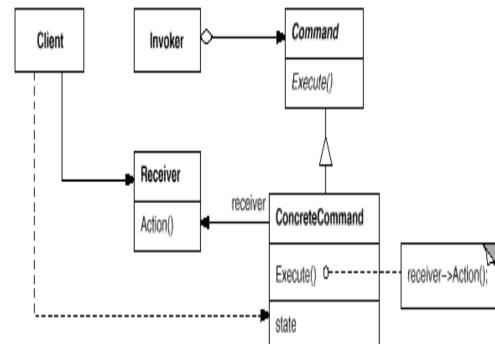
Obserwator

- Tworzy **zależność typu jeden-wiele** pomiędzy obiektami
- Informacja o zmianie stanu** wyróżnionego obiektu jest **przekazywana wszystkim pozostałym obiektom**



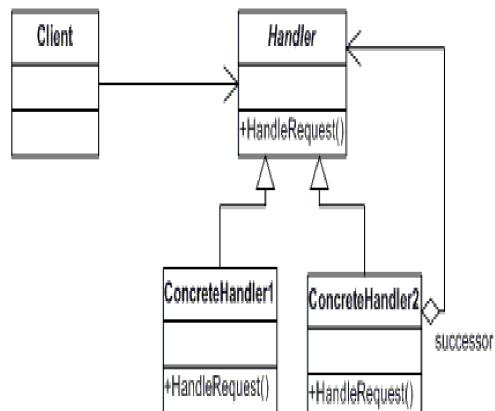
Command

- Hermetyzacja poleceń** do wykonania w postaci obiektów
- Umożliwienie **parametryzacji** klientów obiektami polecień
- Wsparcie dla **polecień odwracalnych**



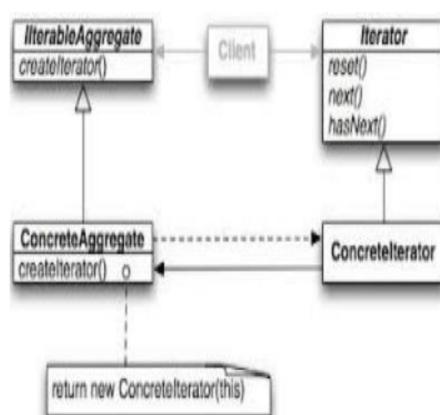
Chain of responsibility

- Usunięcie powiązania pomiędzy nadawcą i odbiorcą żądania**
- Umożliwienie wielu obiektom obsługi żądania



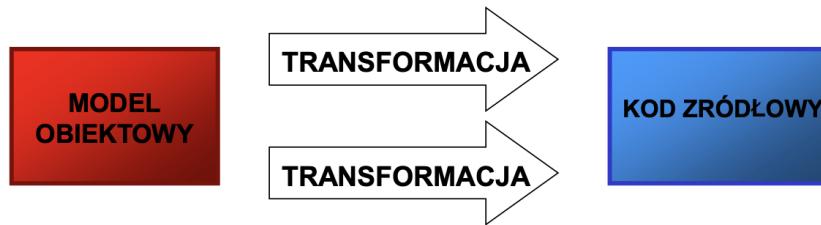
Iterator

- Umożliwienie **sekwencyjnego dostępu** do elementów kolekcji bez ujawniania jej wewnętrznej implementacji



8.8 Implementacja wzorców projektowych

Implementacja to transformowanie modelu w kod źródłowy.



8.8.1 Transformacja

- powinna dotyczyć tylko jednego, ścisłe określonego kryterium,
- musi mieć charakter lokalny, powinna być izolowana od innych zmian,
- musi być poddana weryfikacji.

Transformacja modelu

- ograniczona jest do samego modelu
- celem jest uproszczenie lub zoptymalizowanie istniejącego modelu

Inżynieria postępująca

- tworzenie szablonów kodu źródłowego odpowiadającego modelowi obiektowemu

Najczęściej wykonywane aktywności (transformacje):

- **optymalizowanie modelu** obiektowego,
- odwzorowywanie **skojarzeń w kolekcje**,
- odwzorowywanie **kontraktów w wyjątki**,
- odwzorowywanie modelu obiektowego w **schematy bazy danych**.

8.8.2 Paradygmat programowania

- wzorzec, **najogólniejszy model**, jako wzorcowy przykład,
- **zbiór pojęć** i teorii tworzących **podstawy danej nauki**.

Abstrakcja	System jako układ obiektów rozpatrywanych jako modele abstrakcyjnego elementu, które mogą:
	<ul style="list-style-type: none"> • opisywać i zmieniać swój stan, • komunikować się z innymi obiektami w systemie, • wykonywać pewne czynności na rzecz innych obiektów bez ujawniania, w jaki sposób zaimplementowano dane cechy.
Enkapsulacja	<ul style="list-style-type: none"> • ukrywanie szczegółów implementacji, • obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób, • tylko wewnętrzne metody obiektu są uprawnione do zmiany jego stanu, • każdy typ obiektu dostarcza innym obiektom swój "interfejs", który określa dopuszczalne metody współpracy.
Polimorfizm	<ul style="list-style-type: none"> • wykazywanie różnych form działania podczas wywoływanego metody w zależności od typu obiektu, • referencje i kolekcje obiektów mogą dotyczyć obiektów różnego typu, a wywołanie metody dla referencji spowoduje zachowanie odpowiednie dla pełnego typu obiektu wywoływanego.
Dziedziczenie	<ul style="list-style-type: none"> • porządkuje i wspomaga polimorfizm i enkapsulację, • umożliwia definiowanie i tworzenie specjalizowanych obiektów, • dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tą, której nie mają obiekty ogólniejsze.

8.8.3 GRASP

General Responsibility Assignment Software Patterns - zakres odpowiedzialności.

- **Creator** - określa kiedy podany obiekt powinien tworzyć inny obiekt, tzn. B powinien tworzyć A, jeśli:
 - B agreguje A,
 - B operuje na danych obiektu A,
 - B używa bezpośrednio A,
 - B dostarcza informacji niezbędnej do utworzenia A
- **Information Expert** - określenie danych niezbędnych do wypełnienia nowej odpowiedzialności. Programista powinien delegować ją do obiektów, które zawierają najwięcej informacji pozwalających ją zrealizować.
- **Controller** - jego zadaniem jest: odbieranie informacji od UI, wykonywanie operacji oraz zwracanie ich wyników do UI. Programista deleguje zadania z UI do kontrolera, a kontroler w głąb systemu.
- **Low Coupling** - jak największa niezależność klas.
- **High Cohesion** - obiekt powinien skupiać się na jednej odpowiedzialności, która powinna być jasna i nie rozmyta.
- **Polymorphism**
- **Pure Fabrication** - powstawania w systemie obiektów, które nie reprezentują żadnego obiektu dziedziny, a kondensują funkcje udostępniane na rzecz innych obiektów.
- **Indirection** - aby zapewnić low coupling często zachodzi potrzeba dodania mediatora w komunikacji między obiektami. Jego zadaniem jest jedynie wymiana informacji między obiektami. Taki

obiekt deleguje zadania z jednego obiektu na rzecz drugiego. Projektowanie systemu z użyciem mediatora wpływa na poprawę hermetyzacji elementów systemu. Model MVC jest dobrym tego przykłademastosowania tej zasady. Kontroler jest mediatorem, co izoluje interfejs użytkownika od modelu.

- **Protected variations** - zasada mówiąca o zakresie modyfikacji w systemie wymaganym przez określona zmianę. W systemie powinno się identyfikować punkty niestabilności i budować interfejsy wokół tych punktów. To ograniczy zakres zmian w przypadku, gdyby okazało się, że podany punkt niestabilności systemu wymaga zmian.

8.8.4 Metazasady

- **Don't repeat yourself - DRY**

Jedno miejsce w systemie, na pojedynczą informację, co ułatwia późniejsze zmiany. Inna nazwa to **Single Source Of Truth** (SSOT), każda informacja w systemie powinna być przechowywana dokładnie raz, bo ułatwia to jej modyfikację.

- **Keep it simple, stupid - KISS**

W projektowaniu interfejsów powyższą zasadę można nazwać **zasadą najmniejszego zaskoczenia**, czyli fragment kodu powinien robić dokładnie to co ma robić. Czasem trzeba wybrać, czy dany fragment kodu napisać z wykorzystaniem wzorca projektowego czy prostej konstrukcji.

9 Testowanie i kontrola jakości

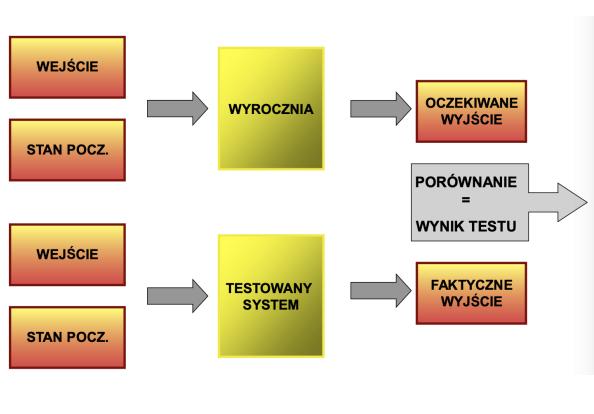
9.1 Testowanie

Testowanie oprogramowania to **wykonywanie kodu dla pewnych kombinacji** danych wejściowych i stanów w celu wykrycia błędów.

Udany test - wykrycie błędu. Efektywność testu - zdolność do znajdywania błędów.

Testowanie + statyczna weryfikacja = **pełne pokrycie dla weryfikacji i walidacji**.

Testowanie jest **jedyną techniką walidacji dla wymagań niefunkcjonalnych**.



Planowanie testów

- im wcześniej rozpoczęniemy tym lepiej;
- statyczna weryfikacja czy testowanie?
- zasada Pareto;
- definiowanie standardów dla testowania;

Dokument planu testów

- Proces testowania
- Śledzenie wymagań
- Testowane elementy
- Harmonogram testów
- Procedury nagrywania testów
- Wymagania odnośnie sprzętu i oprogramowania
- Ograniczenia

Aksjomaty testowania

- **Antyekstencjonalność** - zestaw testów pokrywających nadklasę może nie być odpowiedni dla jakiejś jej implementacji,
- **Antydekompozycja** - pokrycie testami modułu wolanego nie jest takie samo jak pokrycie tego, który woła,

- **Antykompozycja** - testy pokrywające segmenty modułu niekoniecznie są odpowiednie dla modułu jako całości.

Pokrycie kodu

- **Pokrycie instrukcji**: sprawdzana jest każda instrukcja,
- **Pokrycie gałęzi**: odwiedzamy każdą gałąź; instrukcja warunkowa musi być raz spełniona a raz fałszywa.

9.1.1 Rodzaje testów

Jednostkowe

- przeprowadzane na bardzo niskim poziomie aplikacji, bardzo zbliżonym do kodu źródłowego oprogramowania,
- polegają na testowaniu poszczególnych metod i funkcji klas, komponentów lub modułów,
- automatyzacja z reguły dość tania,
- mogą one być bardzo szybko przeprowadzane przez serwer ciągłej integracji.

Integracyjne

- sprawdzają, czy różne moduły lub usługi wykorzystywane przez oprogramowanie dobrze ze sobą współpracują,
- mogą być stosowane na przykład w celu sprawdzania interakcji aplikacji z bazą danych lub upewnienia się, że mikro-usługi działają zgodnie z postawionymi wymaganiami i oczekiwaniemi,
- droższe, ponieważ wymagają uruchomienia wielu elementów aplikacji.

Systemowe - funkcjonalne i nie-funkcjonalne

- sprawdzają funkcjonalne oraz niefunkcjonalne wymagania systemowe oraz jakość testowanych danych,
- technika czarnej skrzynki dla wymagań funkcjonalnych,

Akceptacyjne

- formalne testy sprawdzające spełnienie wymagań biznesowych,
- wymagają uruchomienia i poprawnego działania całości aplikacji i polegają na replikowaniu zachowań użytkowników,
- mogą obejmować również np. wydajność systemu,
- odrzucenie zmian w przypadku, gdy nie pozwalają one na osiągnięcie postawionych celów i wymagań.

Rodzaje testów w fazie pielęgnacji

Regresyjne

- ponowne przetestowanie uprzednio testowanego programu po dokonaniu w nim modyfikacji lub zmianie środowiska pracy
- w celu upewnienia się, że w wyniku zmian nie powstały nowe defekty lub nie ujawniły się wcześniej nie wykryte,

Smoke test

- przetestowanie sprzętu pod kątem tak oczywistego problemu, że wydobywający się z urządzenia dym byłby przewidywanym kryterium niezaliczenia testu,
 - **test pobiczny** (będzie to często test przeszukujący "wszerz", a nie "w głęb"; często będzie to test najbardziej typowej ścieżki czynności, którą może przebyć potencjalny użytkownik),
 - **test zajmujący** niewiele czasu (ma szybko udzielić informacji koniecznych do podjęcia decyzji, co zrobimy dalej w ramach testowania),
 - **test poszukujący** bardzo wyraźnych problemów (test zdany pomyślnie powinien wykluczyć zachodzenie ewidentnej awarii na swojej ścieżce),
 - **test dopuszczający** do kolejnego etapu prac (zwłaszcza w kontekście zaangażowania w tym kolejnym etapie znaczących zasobów).

Techniki testowania

- white box - struktura wewnętrzna
- black box - struktura zewnętrzna

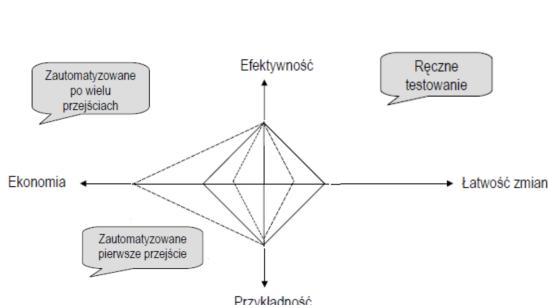
9.1.2 Automatyzacja testowania

- **Bardzo istotny wybór wariantów testów**
- Automatyzacja testów znacznie różni się od testowania.
- Bywa bardzo kosztowna, droższa nawet od ręcznego wykonania testów.

Obietnice automatyzacji testowania

- Zwiększenie testowania (przypadki testowe uruchamiane w minutach)
- Zmniejszenie kosztu testowania aż do 80
- Lepszej jakości oprogramowanie wyprodukowane szybciej

Ocena jakości wariantu testu



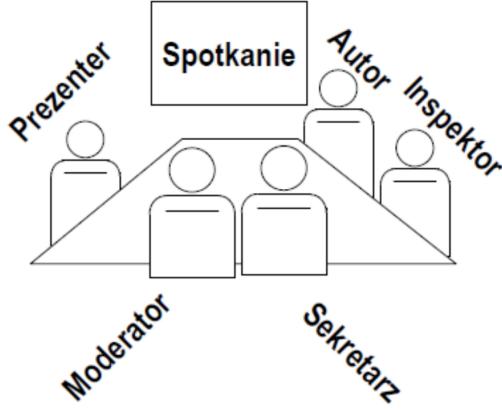
Czynności w ramach testowania

- Identyfikacja warunków testu
- Zaprojektowanie przypadków testowych
- Zbudowanie przypadków testowych
- Uruchomienie przypadków testowych
- Porównanie uzyskanych wyników z oczekiwany

9.2 Kontrola jakości

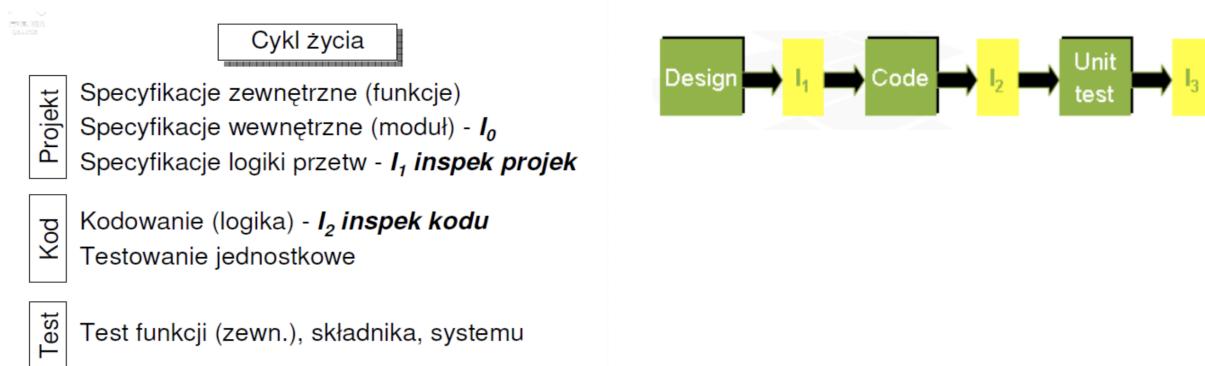
- **Jakość** - zgodność z wymaganiami,
- **Jakość projektu** - wymagania do projektu,

- **Jakość wykonania** - projekt a implementacja,
- **Cztery filary zapewniania jakości**
 - zarządzanie konfiguracją
 - testowanie
 - przeglądy
 - refaktoryzacja
- **Anomalia** - sytuacja różna od oczekiwanej wynikającej ze specyfikacji, standardów lub doświadczenia,
- **Przegląd** - ocena artefaktu realizowana przez grupę osób
 - Czy wszystkie stałe są zdefiniowane?
 - Czy w trakcie manipulacji kolejką może wystąpić przerwanie? Jeśli tak, to czy kolejka jest ujęta w rejon krytyczny?
 - Czy rejesty są odtwarzane przy wyjściu?
 - Czy wszystkie liczniki są odpowiednio inicjowane (0 lub 1)?
 - Czy są literały numeryczne, które powinny być zastąpione stałymi symbolicznymi?
 - Czy wszystkie bloki na schemacie są potrzebne?



- **Inspekcja** - ocena artefaktu przeprowadzana przez współpracowników i kierowana przez moderatora
 - omówienie (cały zespół)
 - przygotowanie (indywidualnie)
 - inspekcja (cały zespół) - akceptacja pełna lub warunkowa/powtórna inspekcja,
 - naprawa
 - sprawdzenie

Inspekcje Fagana



9.2.1 Refaktoryzacja

Refaktoryzacja - **zmiana wewnętrznej struktury programu**, która zwiększa jego **czytelności i obniża koszt pielęgnacji** bez zmiany jego obserwowalnego zachowania.

Motywacja

- Wysoki koszt pielęgnacji oprogramowania
- Naturalny wzrost złożoności i entropii oprogramowania
- Prawa Lehmana: konieczna ciągła restrukturyzacja

Preykat noSideEffectsP

Wejście:

- Program odwołujący się do zmiennej Var o wartości początkowej 1
- Funkcja F potencjalnie modyfikująca wartość Var

Problem 1

- Czy wywołanie funkcji F powoduje efekt uboczny w postaci zmiany wartości zmiennej Var?

Problem 2

- Czy istnieje zbiór wejść, który powoduje zmianę wartości zmiennej Var?

Lemat 1

- Problem 1 (braku efektów ubocznych) jest nie-rozstrzygalny

Lemat 2

- Problem 2 (zmodyfikowany braku efektów ubocznych) jest NP-zupełny.

Proste

- zautomatyzowana weryfikacja,
- weryfikowane poprzez statyczną analizę kodu
- można dowieść ich poprawności
- obecnie w wielu środowiskach IDEs

Trudne

- weryfikacja wymaga testowania
- testy muszą zostać stworzone ręcznie
- nie można dowieść ich poprawności analitycznie
- wymagają testów jednostkowych

10 Ewolucja oprogramowania i zarządzanie konfiguracją

10.1 Przykłady zapachów w kodzie programów

Nazwa	Rozwiążanie
Zduplikowany kod <ul style="list-style-type: none">Identyczny lub podobny kod w wielu miejscach	<ul style="list-style-type: none">w jednej klasie: <i>Extract Method</i>w klasach o wspólnej nadklasie: <i>Extract Method</i> i <i>Pull-up Method</i>w klasach niezwiązanych: <i>Extract Class</i>
Długa metoda <ul style="list-style-type: none">Metoda wykonuje w rzeczywistości wiele czynnościBrak wsparcia ze strony metod niższego poziomu	<ul style="list-style-type: none"><i>Extract Method</i><i>Replace Temp with Query</i><i>Replace Method with Method Object</i><i>Introduce Parameter Object</i> lub <i>Preserve Whole Object</i>
Nadmiernie rozbudowana klasa <ul style="list-style-type: none">Klasa posiada zbyt wiele odpowiedzialności,Liczne klasy wewnętrzne, pola i metody,Duża liczba metod upraszczających.	<ul style="list-style-type: none"><i>Extract Class</i>, <i>Extract Sub/Superclass</i> lub <i>Extract Interface</i><i>Pull up Member</i>, <i>Push Down Member</i>, <i>Move Member</i>.
Długa lista parametrów <ul style="list-style-type: none">Metoda otrzymuje więcej danych niż potrzebuje	<ul style="list-style-type: none"><i>Replace Parameter with Method</i>, <i>Preserve Whole Object</i> lub <i>Introduce Parameter Object</i>.
Nadmiar komentarzy <ul style="list-style-type: none">Komentarze niepotrzebnie opisują znaczenie kodu	<ul style="list-style-type: none"><i>Extract Method</i>, <i>Rename Method</i><i>Introduce Assertion</i>
Niekompletna klasa biblioteczna <ul style="list-style-type: none">Gotowa biblioteka nie posiada pewnej funkcjonalności	<ul style="list-style-type: none"><i>Introduce Foreign Method</i>, <i>Introduce Local Extension</i>
Skomplikowane instrukcje warunkowe <ul style="list-style-type: none">Metoda zawiera złożoną instrukcję if lub switch	<ul style="list-style-type: none"><i>Extract Method</i><i>Replace Conditional with Polymorphism/State</i><i>Replace Conditional with Subclasses</i>
Łańcuchy wywołań metod <ul style="list-style-type: none">łańcuchy wywołań przez delegacjęnaruszenie prawa Demeter	<ul style="list-style-type: none"><i>Hide Delegate</i><i>Extract Method</i> i <i>Move Method</i>

Pojemnik na dane

- Klasa jedynie przechowuje dane i nie posiada użytecznych metod.

Zbitka danych

- Zbiór danych zawsze występujących wspólnie

- *Extract Method* i *Move Method*
- *Inline Class*

Odrzucony spadek

- Podklasa nie wykorzystuje odziedziczonych metod i pól

- *Extract Class, Introduce Parameter Object, Preserve Whole Class*
- podobnie jak z Long Parameter List

Niewłaściwa hermetyzacja

- Klasa bezpośrednio odwołuje się do składowych wewnętrznych innej klasy.

- *Move Member*
- *Change Bidirectional References to Unidirectional*
- *Replace Inheritance with Delegation*

Bezużyteczna klasa

- Klasa nie posiada żadnej lub ograniczoną funkcjonalność.

- *Move Member, Inline Class*
- *Collapse Hierarchy*

Zazdrość o funkcję

- Metoda w klasie częściej korzysta z metod w obcych klasach
- Niska spójność klasy

- *Move Method*
- wzorzec projektowy Visitor

Równoległe hierarchie dziedziczenia

- Utworzenie podklasy powoduje konieczność utworzenia odpowiadającej jej podklasy w innej hierarchii dziedziczenia.

- *Move Method, Move Field, Extract Interface*
- *Inline Class*

Pośrednik

- Klasa deleguje większość funkcjonalności do innych klas

- *Remove Middle Man, Inline Method*
- *Replace Delegation with Inheritance*

Zmiany z wielu przyczyn

- Klasa jest wielokrotnie modyfikowana w różnych celach

- *Extract Class* z elementów zmieniających się z jednego powodu

Odpryskowa modyfikacja

- Zmiana w jednym miejscu powodu konieczność modyfikacji w innych

- *Extract Class, Move Method, Move Field*
- *Inline Class*

Spekulacyjne uogólnienie

- Klasa jest zaprojektowana pod kątem potencjalnej funkcjonalności do zaimplementowania w przyszłości

- *Collapse Hierarchy*
- *Inline Class, Remove Parameter*
- *Rename Method*

10.2 Ewolucja oprogramowania

Ewolucja oprogramowania - proces zmian zachodzących w oprogramowaniu w czasie jego życia.

Pielęgnacja oprogramowania - czynności modyfikujące program po jego dostarczeniu i wdrożeniu.
Cele:

- poprawa błędów
- poprawa wydajności lub innych atrybutów programu
- adaptacja produktu do zmian w środowisku operacyjnym

	Program typu E osadzony rzeczywistości	Program typu P rozwiązuje Problem	Program typu S oparty na Specyfikacji
założenia	system funkcjonuje w rzeczywistym świecie	system w przybliżeniu odzwiera rzeczywistość	dostępna jest pełna specyfikacja systemu
kryterium jakości	subiektywna ocena użytkownika	akceptowalne rozwiązanie problemu	zgodność ze specyfikacją
ewolucja	nieunikniona, program i jego środowisko nieustannie oddziałują na siebie	prawdopodobna – poprawa programu, ewolucja środowiska	brak (modyfikacja nowy problem nowy program)

10.2.1 Prawa Lehmana

Dotyczą systemów typu E, niesprawdzone w typach S i P, raczej obserwacje/hipotezy.

Prawo nieustannej zmiany

Program stosowany w rzeczywistym środowisku musi być stale do niego adaptowany. W przeciwnym przypadku będzie stopniowo coraz mniej użyteczny.

Prawo wzrastającej złożoności

Wraz z ewolucją oprogramowania jego struktura staje się coraz bardziej złożona. Pielęgnacja i upraszczanie struktury wymagają dodatkowych nakładów.

Prawo samoregulacji

Ewolucja oprogramowania jest samoregulującym procesem o rozkładzie atrybutów procesu i produktu bliskim rozkładowi normalnemu. Podstawowe atrybuty procesu ewolucji pozostają stałe dla każdego wydania.

Prawo organizacyjnej stabilności

Tempo rozwoju oprogramowania w całym jego cyklu życia jest stałe, bez względu na dostępność zasobów, jeżeli podczas jego pielęgnacji nie wykorzystano właściwie informacji zwrotnej.

Prawo zachowania przyzwyczajeń

W dłuższej perspektywie, rozmiar kolejnych wydań systemu jest statystycznie niezmienny. Przyswojenie nowych funkcji systemu wymaga czasu.

Prawo ciągłego wzrostu

Funkcjonalność oprogramowania musi rosnąć, aby satysfakcja odbiorców systemu pozostała na tym samym poziomie.

Prawo spadku jakości

Jakość oprogramowania spada, o ile nie jest ono dostosowywane do zmian zachodzących w swoim środowisku operacyjnym.

Prawo przyrostowego rozwoju

Procesy ewolucyjne oprogramowania są wielopoziomowe, posiadają naturę iteracyjną i wymagają informacji z wielu punktów widzenia. Wykorzystanie informacji zwrotnej pozwala osiągnąć istotną poprawę procesu ewolucji.

Wnioski z praw Lehmana

- Oprogramowanie, aby pozostało użyteczne, musi ewoluować;
- Jakość oprogramowania (zdolność do ewolucji) pogarsza się z upływem czasu
- Rosnąca złożoność oprogramowania w pewnym momencie znacznie utrudnia dalszy rozwój systemu
- Tempo rozwoju oprogramowania jest w najlepszym przypadku stałe i nie zależy od sposobu zarządzania

10.2.2 Pielęgnacja oprogramowania

Cztery rodzaje aktywności:

- pielęgnacji doskonalącej (ok. 50%) - implementacja nowych wymagań funkcjonalnych;
- pielęgnacji adaptacyjnej (ok. 25%) - dostosowywanie do zmian zachodzących w środowisku;
- pielęgnacji naprawcze (ok. 20%) - usuwanie błędów
- prewencyjnej (ok. 5%) - restrukturyzacja wewnętrzna.

Model kosztowy Boehma - $AME = 1.0 * ACT * SDT$; AME - roczna pracochnośc związaną z pielęgnacją [PM], ACT - względna liczba zmian [%], SDT - pracochnośc rozwoju oprogramowania [PM].

Czynniki wpływające na koszt pielęgnacji

- Dziedzina zastosowań systemu;
- Stabilność personelu pielęgnacyjnego;
- Czas życia oprogramowania;
- Stabilność sprzętu;
- Jakość kodu i dokumentacji.

Czynniki nie wpływające na koszt pielęgnacji

- Metoda zarządzania przedsięwzięciem;
- Dostępność zasobów.

10.3 Zarządzanie konfiguracją

Problemy

- różnorodność artefaktów
- równoległa praca nad fragmentami kodu
- wiele wersji artefaktów
- wersje artefaktów vs wersje produktu
- analizowanie historii
- praca nad nową wersją systemu i poprawianie starej

Procedury

- kodowanie;
- wydawanie nowej wersji;
- poprawianie defektów;
- łączenia różnych zmian.

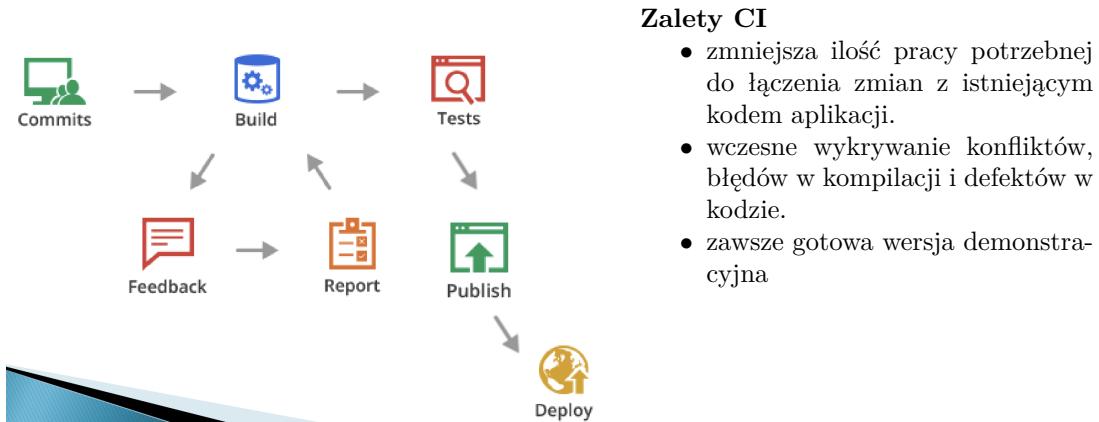
Narzędzia

- CVS
- Subversion
- ClearCase
- Git

11 Ciągła integracja, oprogramowanie w chmurze

11.1 Ciągła integracja

Ciągła Integracja - praktyka polegająca na **jak najczęstszej integracji** zmian z istniejącym już systemem.

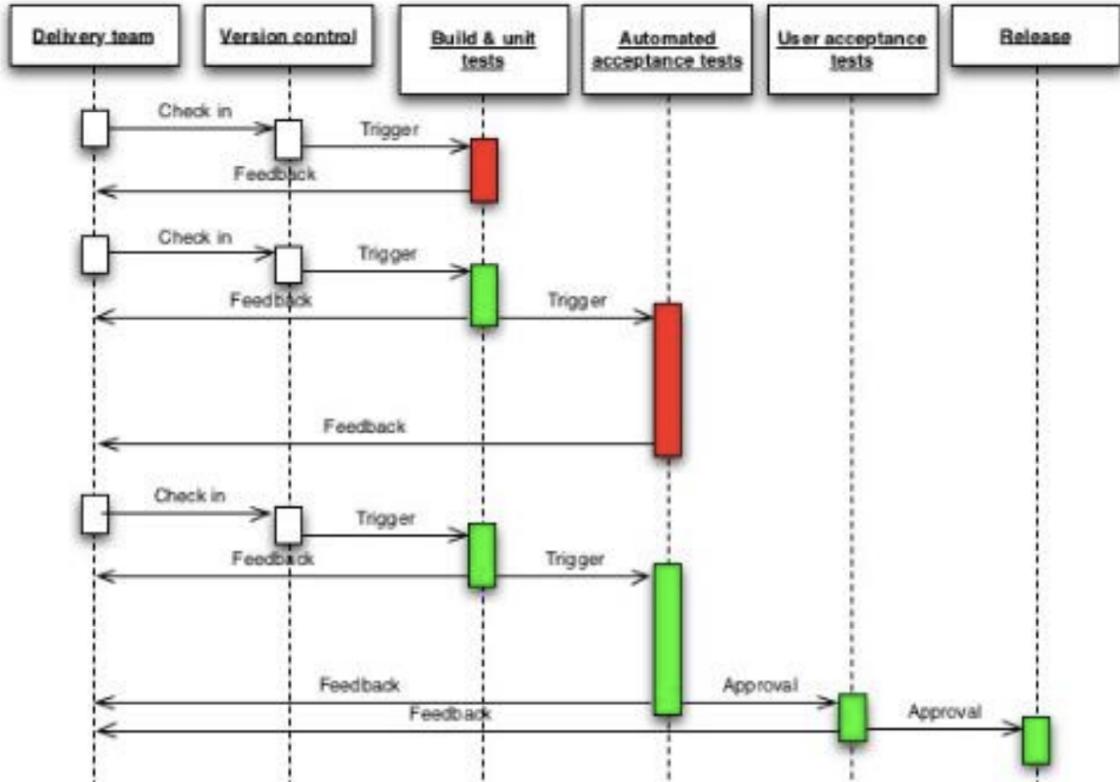


Wymagania CI

- repozytorium kodu
- skrypt umożliwiający automatyczne budowanie aplikacji
- Testy Jednostkowe
- wyzwalacz w postaci włącznika czasowego lub wykrywania zmiany w kodzie, albo połączenia obu.
- system powiadamiania o wynikach procesu i problemach
- zbiór wyników i interfejs dostępnny dla każdego w dowolnym momencie

11.2 Ciągłe dostarczanie oprogramowania

- Za każdym razem, kiedy build pozytywnie przejdzie wszystkie testy, jest automatycznie wdrażany do środowiska testowego, gdzie można go poddać dalszym testom.
- Proces ten może nastąpić tylko raz, zanim oprogramowanie zostanie udostępniane klientom albo może się powtarzać, tworząc wiele nowych funkcji i poprawek zanim nadjejdzie czas wydania.



11.3 Oprogramowanie w chmurze

Praca w chmurze obliczeniowej oznacza **pracę z aplikacjami lub usługami**, udostępnianymi przez internet.

Praca odbywa się na **sieci serwerów** znajdujących się w centrum obliczeniowym.

Zalety

- Skalowalność
- Dostępność
- Wydajność
- Łatwe zarządzanie
- Elastyczność
- Niezawodność
- Ekologia

Wady

- Bezpieczeństwo
- Ograniczone rozwiązania
- Wydajność

11.3.1 Modele dystrybucji

Software as a Service - SaaS

aplikacja jest przechowywana i wykonywana na komputerach dostawcy usługi i jest udostępniana użytkownikom przez Internet.

Infrastructure as a Service - IaaS

polega na dostarczeniu przez dostawcę całej infrastruktury informatycznej, takiej jak np. wirtualizowany sprzęt, skalowany w zależności od potrzeb użytkownika.

Platform as a Service - PaaS

polega na udostępnieniu przez dostawcę wirtualnego środowiska pracy; skierowana jest przede wszystkim do programistów.