

# Notatki z kursu Inżynieria Oprogramowania

Małgorzata Dymek

2018/19, semestr letni

## 1 Podstawowe pojęcia

**Scenariusz przypadku użycia** - wyspecyfikowana 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



## 3 Procesy wytwarzania oprogramowania

Cztery fundamentalne działania wspólne dla wszystkich procesów:

- Specyfikowanie oprogramowania

- **Tworzenie** oprogramowania
- **Walidacja** oprogramowania
- **Ewolucja** oprogramowania

### 3.1 Modele procesu wytwarzania oprogramowania

#### Model kaskadowy

- **wyzolowane etapy:** Planowanie, Analiza, Projekt, Implementacja, Testowanie, Pielęgnacja,
- etapy podzielone na dwie części: twórczą i weryfikacji,
- bardzo wysoki koszt błędów popełnionych we wstępnych etapach, adaptowanie zmian bardzo kosztowne,
- powinien być używany tylko jeśli wymagania są jasne i zrozumiałe,
- marginalizacja roli klienta w procesie wytwarzania oprogramowania.

---

#### Model V

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

---

#### Model Ewolucyjny

- równolegle przeprowadzana specyfikacja, rozwój systemu i weryfikacja
- pozwala później określić wymagania do projektowanego systemu,
- prototyp: pomaga kształcić przyszłego użytkownika, podnosi koszty w krótszej perspektywie, ale w dłuższej może je obniżyć, zwykle jest wyrzucany.

---

#### Model iteracyjny

- planowanie, projektowanie, testowanie, prototypowanie,
- całościowe myślenie o produkcie,
- pozwala na wczesne wykrywanie błędów, łatwość wprowadzania zmian,
- wymogi klienta dotyczące harmonogramu mogą utrudnić korzystanie z tego modelu,
- problemy z oszacowaniem ryzyka.

---

#### Model spiralny

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

## 4 Standardy jakości

Odpowiedź na syndrom **LOOP** - **L**ate, **O**ver budget, **O**vertime, **P**oor quality.

**CMM - Capability Maturity Model**  
**Ocenia proces wytwórczy** w skali pięciostopniowej - od **chaotycznego** do **ściśłego**.

- Poziomy dojrzałości
  - Poziom 1 - Wstępny.
  - Poziom 2 - Powtarzalny.
  - Poziom 3 - Zdefiniowany.
  - Poziom 4 - Zarządzany.
  - Poziom 5 - Optymalizujący.
- Kluczowe obszary procesowe - podzielone na pięć cech
  - podzielone na kluczowe praktyki.

**ISO 9000**  
 Wymaga **udokumentowania wszystkich procedur** związanych z wytwarzaniem oprogramowania.

- Odpowiedzialność kierownictwa
- Zarządzanie zasobami
- Realizacja wyrobu
- Pomiary, analiza i doskonalenie
- Ciągłe doskonalenie systemu zarządzania jakością

## 5 Zwinne procesy wytwarzania oprogramowania

### 5.1 Programowanie ekstremalne - XP

- brak fazy projektowania i dokumentacji,
- krótka perspektywa planowania,
- silne założenie, że klient pracuje cały czas z zespołem.
- **Struktura zespołu** - role podstawowe (programiści, klient) i pomocnicze (tester, coach, tracker).
- **User Stories** - pisują **funkcje systemu** z punktu widzenia użytkownika,
- **Gra planistyczna** - pisanie (klient), oszacowanie (informatycy) i dzielenie (klient) user story,
- **Zapewnianie jakości** - prostota, TTD, automatyczne testowanie, refaktoryzacja.
- **Testy akceptacyjne** - od klienta, najlepiej automatycznie.
- **Programowanie parami** - wspólny standard kodowania, częste zmiany par, system kontroli wersji.

#### Wartości

- **Komunikacja** - przede wszystkim werbalna.
- **Prostota** - rozpoczynamy od najprostszego rozwiązania.
- **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.

## 5.2 SCRUM

### Trzy filary

- **Adaptacja** - powinna być ciągła.
- **Przejrzystość** - istotne aspekty procesu muszą być widoczne dla osób odpowiedzialnych.
- **Inspekcja** - poddawane regularnej inspekcji.

### Role

- **Właściciel Produktu** - odpowiedzialny za pracę ZD, zarządza RP.
- **Zespół deweloperski** - samoorganizujący się, wielofunkcyjny, zarządza RS.
- **Scrum Master**.

### Artefakty

- **Rejestr Produktu** - uporządkowana lista wszystkiego, co może być potrzebne w produkcji.
- **Rejestr Sprintu** - podzbiór RP wybrany do Sprintu rozszerzony o plan dostarczenia Przyrostu produktu.
- **Monitorowanie postępów Sprintu** - możliwe w każdym momencie Sprintu.
- **Przyrost** - suma wszystkich elementów RP zakończonych podczas wszystkich sprintów.
- **Definicja Ukończenia**

### Zdarzenia

- **Sprint** – stały czas, niezmienny cel, niezmienny skład ZD.
- **Przerwanie Sprintu** - tylko przez WP, przy deaktualizacji celu.
- **Planowanie Sprintu** - 8h/mies, wyznaczenie celu, projektu systemu i planu prac.
- **Codzienny Scrum** - 15 min/d.
- **Przegląd Sprintu** - 4h na zakończenie Sprintu.
- **Retrospektywa Sprintu** - inspekcja i opracowanie planu usprawnień.

## 5.3 AGILE PM (DSDM Atern)

### Role

- **Business Sponsor** - najwyższy rangą w projekcie; zapewnia finansowanie i zasoby.
- **Business visionary** - definiuje wizję projektu i komunikuje ją.
- **Project manager** - monitoruje postęp projektu, wysoko poziomowe planowanie.
- **Technical coordinator** - definiuje środowisko pracy, pilnuje standardów, zajmuje się wymaganiami niefunkcjonalnymi.
- **Team Leader**.
- **Business Ambassador** - rola biznesowa w zespole deweloperskim, tworzy dokumentację użytkownika.
- **Business Analyst** - komunikacja między biznesem a zespołem deweloperskim.
- **Solution Developer** - skupiony na dostarczeniu rozwiązania.
- **Solution Tester** - definiuje scenariusze testowe, test case'y.

### Fazy projektu

- **Pre-project** - identyfikacja BS i BV; zakresu, planu i zasobów na Feasibility.
- **Feasibility** - wykonalność, zyskowość, czasowość, kosztowość.
- **Foundation** - wysoko poziomowe wymagania.
- **Exploration** - uszczegóławianie wymagań; iteracyjnie działające możliwe rozwiązanie.
- **Engineering** - rozwijanie rozwiązania z fazy Exploration.
- **Deployment** - potwierdzenie wydajności rozwiązania, dostarczenie rozwiązania i dokumentacji.

## Produkty

- Levels of priority - **MoSCoW**: **M**ust Have, **S**hould Have, **C**ould Have, **W**on't Have this time

## TIMEBOX

- **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

- 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.
- 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.
- możliwość specjalizacji w zespołach
- nie identyfikuje "Ukończenia"

### 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 zredukować zapasy i odkrywać nieefektywności procesów produkcji, dążąc do ich doskonalenia.
- postęp prac monitorowany na tablicy kanban oraz poprzez analizę średnich czasów wykonania

## 5.6 SCRUM-BAN

- board
- tylko daily scrum, płynna praca
- zespoły mogą być specjalizowane, role jak potrzeba
- WIP kontrolowane przez workflow, zmiany dodawane do TODO board na bieżąco
- product backlog tylko w kartkach czasowych

## 6 Wymagania

Klasyfikacja wymagań - **FURPS** - **F**unctionality, **U**sability, **R**eliability, **P**erformance, **S**ecurity.

- **Funkcjonalne** - czynność, zadanie, „System powinien...”.
- **Pozafunkcjonalne** - technikalia mierzone metrykami.
  - **Niezawodność** - odporność na błędy, dojrzałość.
  - **Wydajność**.
  - **Użyteczność** - łatwość zrozumienia i nauki, operatywność.
  - **Łatwość konserwacji** - łatwość analizy, wprowadzania zmian, testowania; stabilność.
  - **Przenośność** - łatwość adaptacji, instalacji.

Cecha **INVEST**: Independent, Negotiable, Valuable, Estimable, Small, Testable.

**Analiza wymagań/analiza obiektowa**

- Celem jest stworzenie modelu systemu, zwanego **modelem analitycznym**.
- Wysilek uczestników projektu skupia się na strukturalizowaniu i formalizowaniu zabranych wcześniej wymagań.
- **Model analityczny** – system z perspektywy użytkownika.
- **Analityczny model obiektowy**.
- **Model dynamiczny** - koncentruje się na zachowaniu systemu.
- **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

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

### 7.1 Podstawowe pojęcia i koncepcje.

- **Podsystem** - wymienna część systemu, posiadającą 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

| Model                                | Opis  | Zastosowanie  |
|--------------------------------------|---|---|
| MVC: Model-Widok-Kontroler           | Model zawiera korowca funkcjonalność. Widoki wyświetlają funkcjonalności. Kontroler obsługuje żądanie użytkownika. Kontroler z widokami tworzą UI aplikacji.  | Smalltalk, Java/Swing.  |
| PAC: Prezentacja-Abstrakcja-Kontrola | Hierarchie kooperujących agentów, podzielonych na trzy komponenty: prezentacji, abstrakcji kontroli.  | Network Traffic Management (gathering traffic data, displaying various user-configurable views of the whole network). |
| 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. | Unix, WEB, Servlet, Numerical Analysis (filters and data extractions).  |
| Tablica (blackboard)                 | Użyteczna w systemach, gdzie nie są znane deterministyczne rozwiązania danego problemu. W przypadku tablicy kilka wyspecjalizowanych systemów łączy swoją wiedzę w taki sposób, żeby stworzyć częściowe lub przybliżone rozwiązanie problemu.         | Working memory, repository data.  |
| Broker                               | Pozwala na uporządkowanie rozproszonych systemów podzielonych na komponenty współpracujące ze sobą za pomocą zdalnego wywoływania serwisu. Komponent brokera odpowiedzialny jest za koordynację komunikacji.  |   |
| Reflection                           | Dostarcza mechanizm pozwalający na dynamiczną zmianę zachowania i struktury systemu.  | WWW.  |
| Sieciowe                             |   |   |
| Architektura klient-serwer           | Podział systemu na dostawcę usług (serwer) oraz ich odbiorców (klientów).   |   |
| Architektura peer-to-peer            | Każdy z podsystemów może spełniać obie funkcje (klient/serwer).   |   |

- 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

- 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** (obektowe);

### 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
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

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

| Wybrane wzorce strukturalne    |  |
|--------------------------------|--|
| <b>Adapter</b>                 | <ul style="list-style-type: none"> <li>• Umożliwia <b>współpracę obiektów o niezgodnych typach</b></li> <li>• Tłumaczy protokoły obiektowe</li> </ul>  |
| <b>Proxy</b>                   | <ul style="list-style-type: none"> <li>• Dostarcza <b>zamiennik obiektu</b> w celu jego kontroli i ochrony</li> <li>• Przezroczyste odsuniecie inicjalizacji obiektu w czasie</li> </ul>   |
| <b>Fasada</b>                  | <ul style="list-style-type: none"> <li>• Dostarczenie <b>jednorodnego interfejsu wyższego poziomu</b> do zbioru różnych interfejsów w systemie</li> <li>• Ukrycie złożoności podsystemów przed klientem</li> </ul>                           |
| Wybrane wzorce behawioralne    |  |
| <b>Obserwator</b>              | <ul style="list-style-type: none"> <li>• Tworzy <b>zależność typu jeden-wiele</b> pomiędzy obiektami</li> <li>• <b>Informacja o zmianie</b> stanu wyróżnionego obiektu jest <b>przekazywana wszystkim pozostałym obiektom</b></li> </ul>     |
| <b>Command</b>                 | <ul style="list-style-type: none"> <li>• <b>Hermetyzacja poleceń</b> do wykonania w postaci obiektów</li> <li>• Umożliwienie <b>parametryzacji klientów</b> obiektami poleceń</li> <li>• Wsparcie dla <b>poleceń odwracalnych</b></li> </ul> |
| <b>Chain of responsibility</b> | <ul style="list-style-type: none"> <li>• <b>Usunięcie powiązania</b> pomiędzy nadawcą i odbiorcą żądania</li> <li>• Umożliwienie wielu obiektom obsługi żądania</li> </ul>   |
| <b>Iterator</b>                | <ul style="list-style-type: none"> <li>• Umożliwienie <b>sekwencyjnego dostępu</b> do elementów kolekcji bez ujawniania jej wewnętrznej implementacji</li> </ul>   |

## 8.5 Implementacja wzorców projektowych

### Transformacja

- powinna dotyczyć tylko jednego, ściśle 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**.

#### 8.5.1 Paradygmat programowania

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

##### Abstrakcja

System jako układ obiektów, które mogą:

- 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

- 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"

##### Polimorfizm

- wykazywanie różnych form działania w zależności od typu obiektu,
- referencje i kolekcje obiektów mogą dotyczyć obiektów różnego typu

##### Dziedziczenie

- porządkuje i wspomaga polimorfizm i enkapsulację,
- umożliwia definiowanie i tworzenie specjalizowanych obiektów,

#### 8.5.2 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.

### 8.5.3 GRASP - General Responsibility Assignment Software Patterns

|  |  |
|--|--|
| <b>Creator</b>                           | Obiekt B powinien tworzyć A, jeśli: <ul style="list-style-type: none"><li>• B agreguje A,</li><li>• B operuje na danych obiektu A,</li><li>• B używa bezpośrednio A,</li><li>• B dostarcza informacji niezbędnej do utworzenia A</li></ul> |
| <b>Information Expert</b>                | <ul style="list-style-type: none"><li>• Określenie danych niezbędnych do wypełnienia nowej odpowiedzialności.</li><li>• Programista deleguje ją do obiektów, które zawierają najwięcej informacji pozwalających ją zrealizować.</li></ul>  |
| <b>Controller</b>                        | <ul style="list-style-type: none"><li>• Odbieranie informacji od UI, wykonywanie operacji (delegowanie zadań w głąb systemu) oraz zwracanie ich wyników do UI.</li></ul>   |
| <b>Low Coupling</b>                      | <ul style="list-style-type: none"><li>• Jak największa niezależność klas.</li></ul>  |
| <b>High Cohesion</b>                     | <ul style="list-style-type: none"><li>• Obiekt powinien skupiać się na jednej odpowiedzialności, która powinna być jasna i nie rozmyta.</li></ul>  |
| <b>Polymorphism<br/>Pure Fabrication</b> | <ul style="list-style-type: none"><li>• Obiekty kondensujące funkcje udostępniane na rzecz innych obiektów.</li></ul>  |
| <b>Indirection</b>                       | <ul style="list-style-type: none"><li>• Dodanie mediatora w komunikacji między obiektami aby zapewnić low coupling.</li></ul>  |
| <b>Protected variations</b>              | <ul style="list-style-type: none"><li>• Zakres modyfikacji w systemie wymagany przez określoną zmianę.</li><li>• Identyfikować punktów niestabilności i obudowanie interfejsami.</li></ul>   |

## 9 Testowanie i kontrola jakości

### Planowanie testów

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

### Aksjomaty testowania

- **Antyekstencjonalność** - zestaw testów pokrywających nadklasę może nie być odpowiedni dla jakiegś jej implementacji,
- **Antydekompozycja** - pokrycie testami modułu wołanego 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.

#### Rodzaje testów

- Jednostkowe
- Integracyjne
- Systemowe
- Akceptacyjne

#### Rodzaje testów w fazie pielęgnacji

- Regresyjne - ponowne przetestowanie uprzednio testowanego programu po dokonaniu zmian.
- Smoke test - przetestowanie sprzętu pod kątem oczywistego problemu.

#### Techniki testowania

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

### 9.1 Kontrola jakości

- **Cztery filary zapewniania jakości**
  - zarządzanie konfiguracją
  - testowanie
  - przeglądy
  - refaktoryzacja
- **Anomalia** - sytuacja różna od oczekiwanej,
- **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 rejestry 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** - po specyfikacji wewnętrznej, po specyfikacji logiki przetwarzania, po kodowaniu.

### 9.1.1 Refaktoryzacja

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

#### Predykat `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`?

#### Lemat 1

- Problem 1 (braku efektów ubocznych) jest nierozstrzygalny

#### Problem 2

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

#### 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 Przykre zapachy w kodzie programów

- |   |   |
|---|---|
| <ul style="list-style-type: none"><li>• Zdublikowany kod</li><li>• Długa metoda</li><li>• Nadmiernie rozbudowana klasa</li><li>• Długa lista parametrów</li><li>• Nadmiar komentarzy</li><li>• Niekompletna klasa biblioteczna</li><li>• Skomplikowane instrukcje warunkowe</li><li>• Łańcuchy wywołań metod</li><li>• Pojemnik na dane</li><li>• Zbitka danych</li></ul> | <ul style="list-style-type: none"><li>• Odrzucony spadek</li><li>• Niewłaściwa hermetyzacja</li><li>• Bezużyteczna klasa</li><li>• Zazdrość o funkcję</li><li>• Równoległe hierarchie dziedziczenia</li><li>• Pośrednik</li><li>• Zmiany z wielu przyczyn</li><li>• Odpryskowa modyfikacja</li><li>• Spekulacyjne uogólnienie</li></ul> |
|---|---|

### 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

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

### Prawa Lehmana

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

- Prawo nieustannej zmiany
- Prawo wzrastającej złożoności
- Prawo samoregulacji
- Prawo organizacyjnej stabilności
- Prawo zachowania przyzwyczajęń
- Prawo ciągłego wzrostu
- Prawo spadku jakości
- Prawo przyrostowego rozwoju

### 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

#### 10.2.1 Pielęgnacja oprogramowania

- **doskonaląca** (ok. 50%) - implementacja nowych wymagań funkcjonalnych;
- **adaptacyjna** (ok. 25%) - dostosowywanie do zmian zachodzących w środowisku;
- **naprawcza** (ok. 20%) - usuwanie błędów
- **prewencyjna** (ok. 5%) - restrukturyzacja wewnętrzna.

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

## 11 Ciągła integracja, oprogramowanie w chmurze

### 11.1 Ciągła integracja

#### Zalety CI

- zmniejsza ilość pracy potrzebnej do łączenia zmian z istniejącym kodem aplikacji.



- wczesne wykrywanie konfliktów, błędów w kompilacji i defektów w kodzie.
- zawsze gotowa wersja demonstracyjna

### 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 dostępny dla każdego w dowolnym momencie

### 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 nadejdzie czas wydania.

## 11.2 Oprogramowanie w chmurze

### Zalety

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

### Wady

- Bezpieczeństwo
- Ograniczone rozwiązania
- Wydajność

### 11.2.1 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.