



Projektowanie obiektowe

Dr Karol Przystalski



Poznajmy się

2017 - doktorat uzyskany w PAN oraz UJ

od 2010 - CTO @ **Codete**

2007 - 2009 - Software Engineer @ **IBM**

Praca naukowa

Multispectral skin patterns analysis using fractal methods}, K. Przystalski and M. J. Ogorzalek. Expert Systems with Applications, 2017

<https://www.sciencedirect.com/science/article/pii/S0957417417304803>

karol.przystalski@uj.edu.pl

kprzystalski@gmail.com

karol@codete.com



Zaliczenie

Średnia ocena ze wszystkich projektów.

Warunki:

- Termin na każdy projekt: max. 2 tygodnie po zajęciach
- Każdy projekt zaliczony na minimum 3.0

Egzamin: test wyboru, 30 pytań. Aby zaliczyć trzeba poprawnie odpowiedzieć na 20 pytań.

Terminy do ustalenia



Paradygmaty programowania

ang. Programming paradigms



Programowanie imperatywne

Komputer pracuje w sposób imperatywny, czyli wykonywania instrukcji jedna po drugiej jednocześnie zmieniając stan programu.

Języki, które realizują paradygmat programowania imperatywny: COBOL, PHP, Ruby, Java.



Programowanie strukturalne

Wprowadzony został przez Edsgera Wybe Dijkstra w 1968 roku. Zastąpił on skoki typu `goto` instrukcjami typu `if-then`, `while`, itp.

Języki, które realizują paradygmat programowania strukturalnego: ALGOL, Pascal, Ada.



Programowanie obiektowe

Programowanie obiektowe jest jeszcze starsze od strukturalnego, ponieważ jego historia sięga 1966, kiedy Ole Johan Dahl oraz Kristen Nygaard zaproponowali, aby przenieść wywołanie funkcji przenieść do stosu, tak aby zmienne w niej istniejące były dostępne również poza funkcją. I tak powstał konstruktor.

Języki, które realizują paradygmat programowania strukturalnego: Smalltalk, Java.



Programowanie funkcjonalne

Jeszcze starszym paradygmatem jest znany od 1936 roku wraz z wprowadzeniem rachunku lambda (ang. λ -calculus). Pierwszy raz został jednak zastosowany dopiero w 1958 w języku LISP.

Języki funkcjonalne nie mają możliwości przypisania (ang. assignment), ponieważ język funkcjonalny z definicji jest niezmienny (immutable).

Języki, które realizują paradygmat programowania strukturalnego: JavaScript, Erlang, LISP, Haskell.



Pozostałe paradygmaty

- Proceduralne: Pascal
- Wizualne: Scratch
- Uogólnione: Java
- Logiczne: Prolog
- Aspektowe: AspectJ
- Deklaratywny: SQL
- Modularne: Fortran90



Zarządzanie pamięcią

ang. Memory management



Heap vs stack

Stack działa na zasadzie LIFO i alokuje pamięć per wątek. Zarządzany przez system operacyjny.

Heap jest wykorzystywany do alokacji pamięci kiedy tylko chcemy, ale jest też wolniejszy. Zarządzany z poziomu aplikacji.

Obie pamięci wykorzystują pamięć RAM.



Manualne zarządzanie pamięcią

Przykładem języka, gdzie zarządzanie pamięcią odbywa się w sposób manualny jest C++.

Służą do tego m.in. funkcje `malloc` i `free`.

Wykorzystujemy do tego również `new` oraz `delete`.

W innych językach też mamy takie funkcje: Java (`new`), Python (`malloc`).



ARC

Jest wykorzystywany m.in. w Swift'cie. Implementuje metody `retain` oraz `release`. Gdy liczba referencji zejdzie do 0, obiekt jest usuwany (`release`).

Więcej o ARC:

<https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html>



Słabe referencje (weak references)

Aby zapobiec zapętleniu się referencji wprowadzono weak references, np. w Swift'cie czy Pythonie. Dzięki temu inaczej liczone są referencje np. w ARC i zapobiega to wyciekowi pamięci.

Python: <https://docs.python.org/3/library/weakref.html>

Swift: <https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html>



Garbage collector

Języki, które go stosują to m.in.: Java oraz inne na JVM, JavaScript, C#, Go, Ruby.

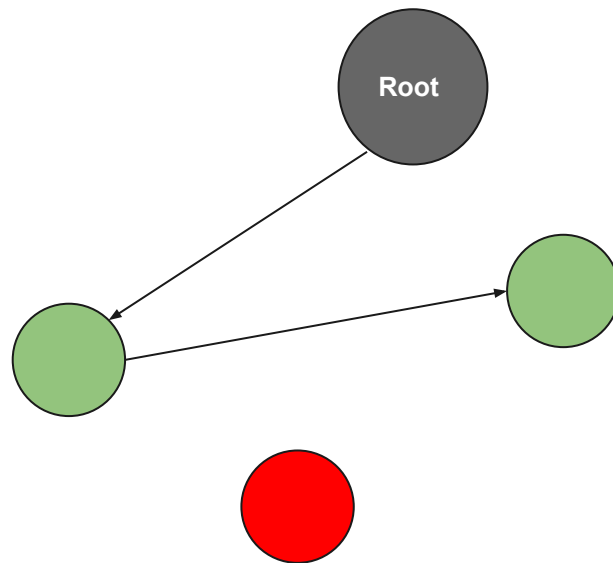
Istnieje wiele implementacji GC:

- Tracing GC - składa się z dwóch etapów: wykrywania „żyjących” obiektów oraz zwalnianie pamięci,
- RC GC - podobne do ARC; posiada osobny proces GC, który czyści pamięć

Garbage collector

Mark - zaznacz (zielone)

Sweep - usuń (czerwone)





Java Garbage collector

Istnieje wiele GC w Javie 11 (-XX:):

- Serial Collector - jeden wątek, dobry dla małych aplikacji
- Parallel Collector - wiele wątków; średnie aplikacje
- Garbage-First Collector - równoległe wątki; wykorzystywany na sprzęcie z wieloma procesorami i dużą ilością pamięci
- Z Garbage Collector - dostępny od Javy 11 zoptymalizowany na ograniczenie opóźnień po stronie aplikacji.



Zarządzanie pamięcią w różnych językach

- Python - wykorzystuje połączenie RC z GC.
- Java - GC
- JavaScript - GC
- Haskell - GC
- Go - GC via Go scheduler
- Ruby - GC
- Swift - ARC
- Lisp - GC
- COBOL - manualne
- Lua - GC



Wzorce architektury



Architektura warstwowa

Jednowarstwowe - przykład: aplikacje desktopowe

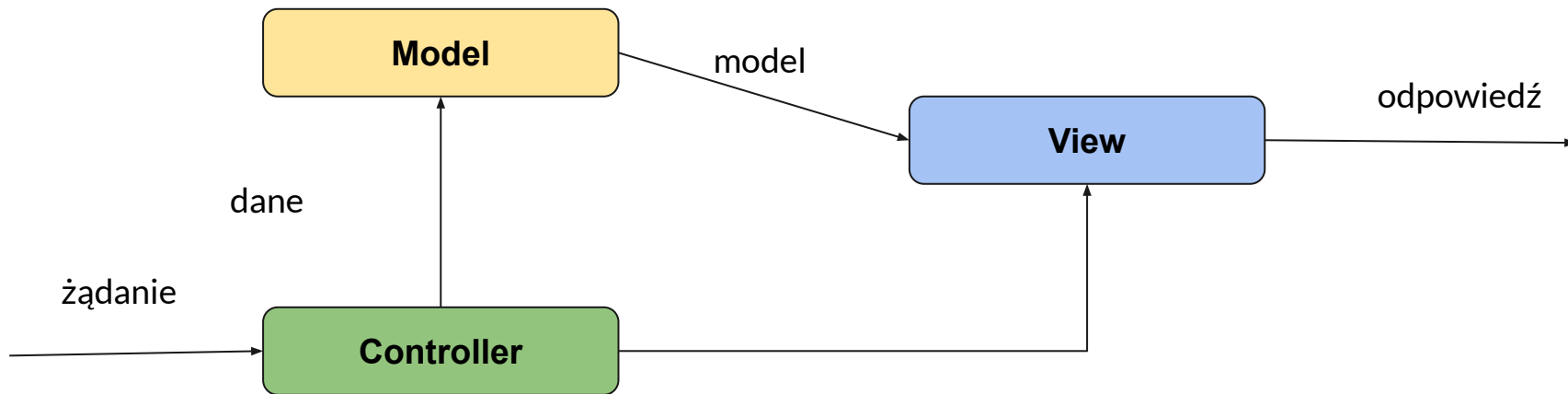
Dwuwarstwowe - przykład: architektura klient-serwer


Trójwarstwowe - przykład: MVC

Wielowarstwowe



MVC





MVC - zalety

Logika biznesowa oddzielona jest od widoku,

Nie ma zależności modelu od widoku

Uporządkowany kod



MVC - wady

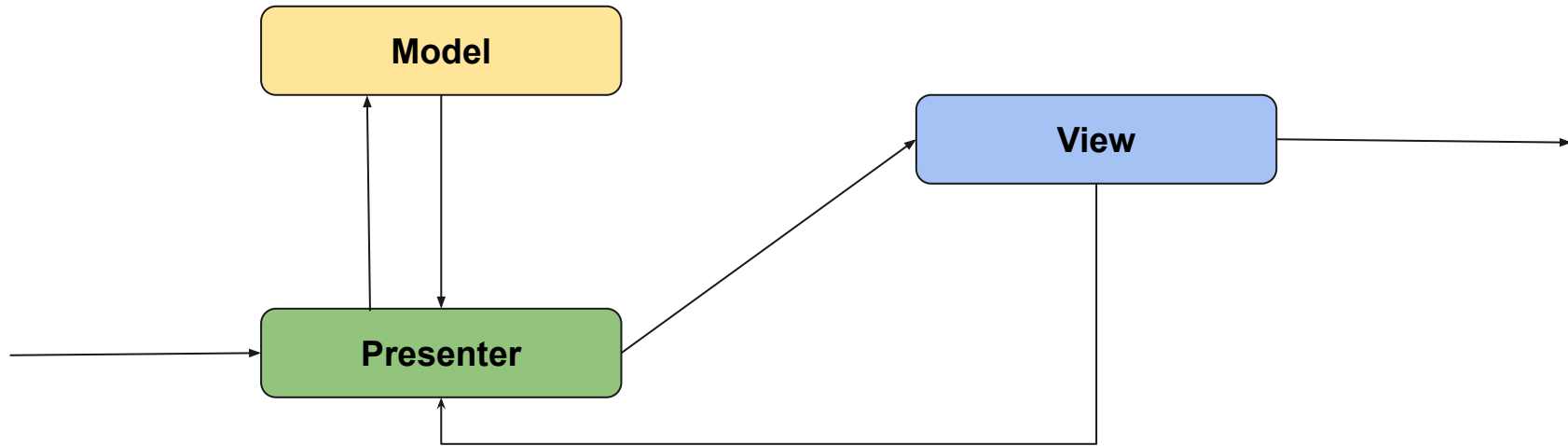
Złożoność aplikacji


Widok zależny od modelu

Widoki są trudne w testowaniu



MVP





MVP - zalety

Pochodna MVC

Oddzielona logika od widoku

Brak zależności modelu od widoku,

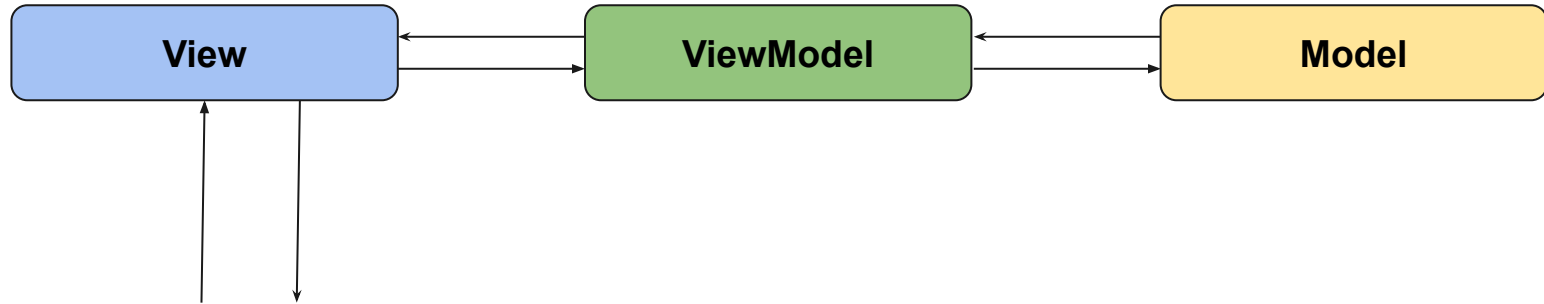



MVP - wady

Skomplikowana dla prostych aplikacji



MVVM






MVVM- zalety

Podział na widok i ViewModel

Testowanie jest prostsze niż w MVC

Asynchroniczność



MVVM - wady

ViewModel nie komunikuje się z warstwą widoku

Dużo klas, ponieważ każdy widok ma więcej niż jedną klasę po stronie widoku i ViewModel.



Inne architektury

Model View Adapter (MVA)

Service Oriented Architecture (SOA)

Presentation-Abstraction-Control (PAC) - hierarchiczna struktura agentów



Zasady

ang. Design Principles



Zasady - po co?

Celem stosowania/wprowadzenia zasad jest tworzenie czystego kodu. Przez czysty kod rozumiemy taki kod, który jest łatwy w rozumieniu i czytaniu przez zespół oraz programistów z zewnątrz. Jednocześnie czysty kod to taki kod, który jest łatwy w utrzymaniu, gdzie można dodać kolejne funkcjonalności bez znacznych zmian w kodzie.

Podstawowe zasady to:

- SOLID,
- GRASP,
- KISS,
- DRY/IDE,
- YAGNI.



Zasady - o czym mówią?

Mówią o tym jak powinniśmy tworzyć kod oraz jak go ustrukturyzować. Nie mówią wprost o samej jakości kodu przez tworzenie odpowiednich testów, a jakości ocenianą przez czytelność.



Zasady - cele

Istnieją trzy cele stosowania zasad:

- czytelność kodu,
- podatność kodu na zmiany,
- przenośność kodu,
- łatwość utrzymania.



Zasady - stosuj z umiarem

Istnieje tzw. *hype*, czyli przesadność w stosowaniu pewnej technologii czy podejścia tylko dlatego, że jest to modne. Wynika to najczęściej z tego, że znane są zalety danego rozwiązania/podejścia i są one nagłaśniane, jednocześnie zapomina się o ich wadach.

Podobnie jest z zasadami, które teraz już nie są tak przesadnie stosowane jak kiedyś, ale nadal warto uważać na to, aby nie przesadzić i nie zamieniać wszystkiego we wzorzec.



SOLID



SOLID - kto i dlaczego?

Zestaw zasad SOLID został zapoczątkowany już w latach 80-tych XX. wieku, ale dopiero w 2004 Michael Fathers uporządkował je i określił mianem SOLID.

SOLID tak jak inne zasady oraz wzorce projektowe wzięły się od programistów jako znane, dobre praktyki tworzenia oprogramowania.



Single Responsibility

"Moduł/Aktor powinien być odpowiedzialny tylko za jedną funkcjonalność "

ang. *"A module should be responsible to one, and only one, actor."*

Zastąp boską klasę osobnymi klasami, każda odpowiedzialna za inną funkcjonalność

Przykład

Klasy nie powinny jednocześnie być odpowiedzialne za autentykację oraz wysyłanie maili



Open/Closed

"Komponenty powinny być otwarte na rozszerzenia, ale zamknięte na ich modyfikację"

ang. "Software components should be open for extension, but closed for modification"

Należy rozszerzać klasy za pomocą dziedziczenia, nie modyfikować ich samych.

Przykład

Cieężko utrzymać takie klasy w 100% kodu. Dla wielu problemów stosuje się strategię



Liskov Substitution Principle

“Typy pochodne powinny być całkowicie zastępowalne w ramach swoich typów bazowych”

ang. "Derived types must be completely substitutable for their base types"

Jeżeli istnieje obiekt **o1** typu **S**, to istnieje taki obiekt **o2** typu **T**, taki że dla każdego kodu **P** zdefiniowanego w ramach **T**, zachowanie **P** nie zmieni się jeżeli podmienimy obiekt z **o1** na **o2**, jeżeli **S** jest podtypem **T**.

Przykład

Problem kwadratu i prostokąta, gdzie kwadrat dziedziczy po prostokącie.



Interface Segregation Principle

"Klienci nie powinni być zmuszani do implementowania niepotrzebnych metod"

ang. "Clients should not be forced to implement unnecessary methods which they will not use"

Jest to powiązane również z zasadą YAGNI, ale w tym przypadku odnosi się do dziedziczenia oraz implementacji interfejsów. Implementowanie niepotrzebnych metod wprowadza bałagan w kodzie.

Przykład

Interfejsy, które zależą od innych interfejsów, a które zawierają metody, które mogą być niepotrzebne należy zastąpić wzorcem fabryki abstrakcyjnej.



Dependency Inversion Principle

ang. "Bądź zależny od abstrakcji, a nie konkretnej implementacji"

ang. "Depend on abstractions, not on concretions"

Starajmy się tworzyć klasy abstrakcyjne tam, gdzie spodziewamy się dziedziczenia po danej klasie. Najgorsze co może się stać do zależności rekurencyjne.

Przykład

Przykładem może być String w Javie, który jest konkretną implementacją i od niego zależymy. Jednocześnie nie możemy zrobić inaczej.



GRASP



GRASP - General Responsibility Assignment Software Patterns

Składa się z dziewięciu zasad: kontroler, twórca, pośrednik, ekspert od informacji, niskie sprzężenie, wysoka spójność, polimorfizm, bezpiecznie opcje, czyste tworzenie.



Pozostałe zasady



DRY/DIE

“Don’t repeat yourself”, “Duplicate is evil”

Nie duplikuj kodu



KISS

“Keep it simple, stupid”

Klasy i metody powinny być proste. Przeciwnieństwo to klasy boskie.



YAGNI

“You ain’t gonna need it”

Częste dodawanie funkcjonalności, której nie ma w historyjkach jest częsta i zbędna ze względu na sposób w jaki tworzone jest oprogramowanie.



Wzorce projektowe



Wzorce projektowe

Jest kilka grup wzorców projektowych:

- strukturalne
 - Adapter
 - Most (Bridge)
 - Kompozyt (Composite)
 - Dekorator (Decorator)
 - Fasada (Facade)
 - Pyłek (Flyweight)
 - Pełnomocnik (Proxy)



Wzorce projektowe

Jest kilka grup wzorców projektowych:

- strukturalne
- kreacyjne
- Fabryka abstrakcyjna (Abstract Factory)
- Budowniczy (Builder)
- Metoda wytwórcza (Factory Method)
- Object Pool (Pula obiektów)
- Prototyp (Prototype)
- Singleton



Wzorce projektowe

Jest kilka grup wzorców projektowych:

- strukturalne
 - kreacyjne
 - behawioralne
- Łańcuch zobowiązań (Chain of responsibility)
 - Polecenie (Command)
 - Interpreter
 - Iterator
 - Mediator
 - Memento
 - Obserwator (Observer)
 - Stan (State)
 - Strategia (Strategy)
 - Metoda szablonowa (Template Method)
 - Wizytator (Visitor)



Wzorce projektowe

Jest kilka grup wzorców projektowych:

- strukturalne
- kreacyjne
- behawioralne

Istnieją również grupy antywzorców:

- tworzenia oprogramowania

- Blob
- Ciągłe starzenie (Continuous Obsolescence)
- Martwy kod (Lava Flow)
- Niejednoznaczność (Ambiguous Viewpoint)
- No OO (Functional Decomposition)
- Duch (Poltergeists)
- Kotwica (Boat Anchor)
- Złoty młot (Golden Hammer)
- Ślepa uliczka (Dead End)
- Spaghetti (Spaghetti Code)
- Bobas (Input Kludge)
- Pole minowe (Walking through a minefield)
- Kopiuj-wklej (Cut-and-Paste programming)
- Na ślepo (Mushroom Management)



Wzorce projektowe

Jest kilka grup wzorców projektowych:

- strukturalne
- kreacyjne
- behawioralne

Istnieją również grupy antywzorców:

- tworzenia oprogramowania
- architektury

- Stożki migracyjne (Autogenerated Stovepipe)
- Zbieranina (Jumble)
- Papierkowa robota (Cover Your Assets)
- Uzależnienie (Vendor Lock-In)
- Wilczy bilet (Wolf Ticket)
- Ciepłe posadki (Warm Bodies)
- Polityczna (Design By Committee)
- Szwajcarski scyzoryk (Swiss Army Knife)
- Odkrywanie koła na nowo (Reinvent The Wheel)
- Abstrakcja vs Implementacja (The Grand Old Duke of York)



Kreacyjne wzorce projektowe



Fabryka abstrakcyjna (Abstract factory)

Założenia

- Dostarcza interfejs do tworzenia całej rodziny powiązanych obiektów bez specyfikowania klasy z implementacją
- Hierarchia, która enkapsuluje wiele klas (produktów)

Przykład

Maszyny produkujące elementy samochodów wykorzystują ten wzorzec. Mogą po podmianie niektórych części maszyny, mogą one produkować różne elementy samochodu do różnych ich modeli.



Budowniczy (Builder)

Założenia

- Oddziela logikę odpowiedzialną za tworzenie złożonych obiektów od ich reprezentacji
- Umożliwia na tworzenie różnych obiektów w zależności ich reprezentacji

Przykład

Tworzenie różnych rodzajów pizzy jest takim przykładem. Główny proces jest taki sam, różnią się jedynie



Fabryka (Factory method)

Założenia

- Definiuje interfejs dla tworzenia obiektu, jednocześnie pozostawia decyzję klasie, którą klasę zainicjalizować
- W odróżnieniu od budowniczego skupia się wokół konstruktora, jest prostsza.

Przykład

Jest wykorzystywana przez budowniczego.



Pula obiektów (Object pool)

Założenia

- Ma również zastosowanie przy projektach, gdzie istotna jest prędkość działania
- Pozwala na alokowanie pamięci dla grupy obiektów, ponieważ ta część jest najbardziej czasochłonna
- Udostępnia instancje obiektów

Przykład

W grach jest często wykorzystywana oraz wszędzie tam, gdzie wykorzystywane są identyczne obiekty wiele razy.



Prototyp (Prototype)

Założenia

- Jest prototypem obiektu
- W wielu językach klonuje albo kopiuje się prototyp

Przykład

W języku JavaScript jest podstawowym pojęciem.



Singleton

Założenia

- Jeden z prostszych wzorców
- Zapewnia istnienie jednej instancji danej klasy

Przykład

Wykorzystywany jest często w autoryzacji czy dostępie do bazy danych.



Strukturalne wzorce projektowe



Adapter

Założenia

- Konwertuje interfejs klasy na interfejs, który jest oczekiwany przez klienta
- Pozwala na pracę pomiędzy klasami, które normalnie nie mogłyby pracować
- Pozwala na współpracę starego/innego rozwiązania z naszym

Przykład

Brak kompatybilności pomiędzy standardami obsługi odpowiedzi od serwera. Do obsługi możemy wykorzystać interfejs, który pozwoli na obsługę konkretnej odpowiedzi z serwera przez klasę, która nie została zaimplementowana do obsługi odpowiedzi danego typu. Innym przykładem jest wtyczka do prądu w UK, US oraz w Europie. Istnieją adaptory, które pozwalają na pracę wtyczek z kontaktami w różnych krajach.



Most (Bridge)

Założenia

- Celem jest rozdzielenie abstrakcji od implementacji
- Udostępnia hierarchię interfejsów oraz odpowiednią hierarchię implementacji

Przykład

Most rozbija część implementacji od jej abstrakcji, w taki sposób że są od siebie niezależne. Przykładem może być przełącznik do włączania/wyłączania światła. Abstrakcja jest jedna, a implementacji wiele.



Kompozyt (Composite)

Założenia

- Kompozyt tworzy reprezentację obiektów za pomocą drzewa. Jednocześnie kompozyt pozwala na dostęp do poszczególnych elementów ze względu na dziedziczenie po interfejsie nadrzędnym
- Upraszcza kod aplikacji
- Rekursywne

Przykład

Operacja arytmetyczna może być kompozytem, ponieważ $(2+3) - (6+2)$ jest tym samym co $2+3-6+2$.



Dekorator (Decorator)

Założenia

- Dodaje dodatkowe funkcjonalności do obecnie istniejących klas opakowując je, wywołując metody wewnątrz
- Jest alternatywą dla tworzenia podklas

Przykład

Dodanie kilku klas CSS do zwracanego przez klasę kodu HTML jest dobrym przykładem dekoratora.



Fasada (Facade)

Założenia

- Opakowuje złożoną część kodu przez proste interfejsy
- Uogólnia system za pomocą prostych interfejsów
- Wysokopoziomowe podejście

Przykład

Fasadą jest niemal każdy call center. Dzwonimy na numer biura obsługi klienta i wybieramy kolejno numery, aby skontaktować się z odpowiednią osobą, która jest odpowiednia dla rozwiązania naszego problemu.



Pyłek (Flyweight)

Założenia

- Pozwala na współdzielenie bardzo wielu małych obiektów
- Ma zastosowanie w usprawnieniach wydajności

Przykład

Przeglądarki wykorzystują ten wzorzec do ładowania dużej liczby rysunków.



Proxy

Założenia

- Tworzy obiekt pośredni w komunikacji pomiędzy dwoma innymi
- Pozwala na dostęp oraz kontrolę drugim obiektem

Przykład

Najprostszy przykładem jest proxy wykorzystywane w przeglądarkach.



Funkcyjne wzorce projektowe



Monoid

Założenia

- Struktura algebraiczna
- Występuje funkcja zwracająca wartość pustą oraz metoda łącząca zwracająca wartość podaną jako argument, np. combine

Przykład

MapReduce wykorzystuje monoidy. Option jest monoidem



Monada

Założenia

- Pozwala na zastąpienie wielu wywołań, które od siebie zależą funkcjami unit oraz bind
- Upraszcza operacje pomiędzy funkcjami zależnymi
- Jest monoidem

Przykład

Monada zamienia zagnieżdżone wyrażenia w proste (flatten)



Funktor

Założenia

- Funktor stosuje operację na elementach listy, funkcjach lub innych strukturach
- Zawiera funkcję mapującą (map, flatmap),
- Wyjście jest typu wejścia

Przykład

Funktory są często wykorzystywane przy listach, np. kiedy je mapujemy.



Behawioralne wzorce projektowe



Łańcuch odpowiedzialności (Chain of responsibilities)

Założenia

- Przekazuje obiekt do kolejnych obiektów w łańcuchu, aż któryś z nich je obsłuży

Przykład

Przykładem jest każdy framework MVC albo bankomat.



Komenda (Command)

Założenia

- Enkapsuluje wywołanie/żądanie w obiekcie, jednocześnie parametryzując żądanie w obiekcie

Przykład

Przykładem jest PyCall, który wywołuje komendę w Pythonie z poziomu języka Ruby.



Interpreter

Założenia

- Mapuje gramatykę języka i interpretuje ją w celu wykonania poszczególnych komend

Przykład

Występuje w każdym języku skryptowym



Iterator

Założenia

- Pozwala na łatwe przejście sekwencyjne przez kolekcję

Przykład

Występuje w większości kolekcji w językach obiektowych.



Mediator

Założenia

- Pozwala na komunikację (kontrolę) się wielu niezwiązanych ze sobą bezpośrednio obiektów.
- Pozwala na komunikację wielu-do-wielu

Przykład

Przykładem jest wieża kontroli lotów.



Memento

Założenia

- Pozwala na zapamiętanie stanu obiektu, dzięki czemu może wrócić do poprzedniego stanu obiektu w przypadku błędów
- Może tworzyć historię zmian i odtwarzać stany z przeszłości (check points)

Przykład

Niektóre debugery realizują ten wzorzec i pozwalają na powrót stanu poprzedniego.



Obserwator (Observer)

Założenia

- Potrafi zdefiniować zależność jeden do wielu, w taki sposób, że gdy jeden z obiektów zmieni swój stan, pozostałe obiekty są o tym informowane

Przykład

View w MVC oraz aukcje.



Stan (State)

Założenia

- Podobnie jak obserwator, ale dotyczące jednego obiektu
- Obiekt zmieniając swój stan zmienia swoje zachowanie
- Obiektowe maszyna stanów

Przykład

Maszyna vendingowa, chatboty



Strategia (Strategy)

Założenia

- Definiuje wiele metod do rozwiązania podobnego problemu w różny sposób.
- Strategia je enkapsuluje i pozwala na wybór odpowiedniej metody w zależności od wejścia

Przykład

Wybór trasy przejazdu z punktu A do B.



Metoda szablonowa (Template method)

Założenia

- Tworzy szablon/szkielet metody, w której niektóre kroki są wywołaniem metod z innych subklas.
- Klasa bazowa określa gdzie znajdują się implementacje poszczególnych kroków metody

Przykład

Budowanie domu.



Wizytator (Visitor)

Założenia

- Wizytator pozwala na zdefiniowanie operacji bez zmiany danej klasy oraz elementów danej klasy na której operuje.
- Pozwala na odzyskanie utraconych informacji

Przykład

Taksówka.



Wzorce projektowe - porównanie



Budowniczy vs Fabryka

Budowniczy

- Zawiera metodę `build()`, która buduje instancję, ale parametry mogą być podawane przez settery
- Ma najczęściej wiele metod, które rozkładają tworzenie obiektu na wiele etapów (funkcjonalności)

Fabryka

- Prostsza od budowniczego
- Wymaga podania wszystkich parametrów w jednej metodzie



Most vs Strategia

Most

- Tworzymy strukturę interfejsów
- Wykorzystywane, gdy chcemy użyć tych interfejsów w innym celu
- Wybór „opcji” wybierany jest na poziomie implementacji

Strategia

- Zmiana implementacji podczas wykonywania kodu, tj. wybieramy, który obiekt wykona daną akcję



Fasada vs Dekorator

Fasada

- Nie dodaje dodatkowych funkcjonalności do istniejącego obiektu

Dekorator

- Dodaje dodatkowych funkcjonalności do istniejącego obiektu



Czysty kod



Zapaszki wg Uncle Boba

Wg Robert C. Martina w *Clean Code* dzieli on zapaszki na kilka typów, dotyczące:

- komentarzy,
- środowiska,
- funkcji,
- ogólne,
- Javy,
- nazewnictwa,
- testów.



Komentarze

1. Błędne informacje
2. Stare informacje
3. Redundantne
4. Komentarze słabej jakości
5. Zakomentowany kod



Środowisko

Zapaszki środowisko opierają się na procesie pisania oraz budowania. Dwie podstawowe zasady to:

1. Budowanie to nie jest jeden prosty krok
2. Testy to nie jest jeden prosty krok



Funkcje

Wiele zapaszków oraz złych praktyk dotyczy pisania funkcji/metod. Uncle Bob wyróżnia cztery zasady:

1. Zbyt wiele argumentów
2. Argumenty wyjściowe
3. Argumenty binarne (Boolean)
4. Martwe funkcje



Ogólne

Jest ich najwięcej i dotyczą wielu aspektów tworzenia kodu. Niektóre z nich to:

- Wiele języków w jednym pliku
- Oczywiste zachowania nie są zaimplementowane
- Błędne zachowanie przy wartościach brzegowych
- Nadpisywać zabezpieczenia
- Duplikacje
- Kod na nieprawidłowym poziomie abstrakcji
- Klasa bazowa zależy od pochodnej
- Potok informacji
- Niekonsystentność
- Sztuczny podział
- Opakuj wartości brzegowe
- Zainteresowanie zmiennymi w klasie
- Intencja w nazwie
- Porozrzucane odpowiedzialności
- Statyczne zmienne niepotrzebne
- Rozumienie algorytmu
- Polimorfizm zamiast switch
- Standardy
- Zmienne zamiast magicznych liczb
- Dokładność
- Unikaj negatywnych porównań
- Funkcje powinny robić jedną rzecz
- Konfiguracja



Java

Jest kilka zasad, które odnoszą się bezpośrednio do Javy:

1. Unikaj długiej listy importów
2. Nie rozszerzaj stałych
3. Enum zamiast stałych



Nazewnictwo

Jest kilka zasad, które wydają się oczywiste i odnoszą się do nazewnictwa:

- nazwy klas oraz funkcji/metod, które coś mówią,
- nazwy, które odnoszą się do poziomu abstrakcji,
- używaj standardowych podejść do nazewnictwa,
- niedwuznaczne nazwy,
- długie nazwy tak, ale tylko przy większej funkcjonalności
- unikaj standardów typu `v_`, `b_` lub innych
- nazwy powinny mówić o rezultacie



Testy

Zasady dotyczące testów:

- pomarańczowe testy,
- wykorzystuj testy do pokrycia testów,
- nie unikaj banalnych testów,
- ignorowanie testów?
- wartości brzegowe,
- pokrycie testami metod, które miały wcześniej bugi,
- testy powinny być szybkie.



Antywzorce



Typy antywzorców

Antywzorce możemy podzielić na które występują w:

- kodzie,
- architekturze aplikacji,
- procesie zarządzania projektem.



Antywzorce kodu



Blob

Dotyczy sytuacji, kiedy mamy „boskie” klasy, czyli takie, które robią w zasadzie wszystko.

Innym przykładem są pliki z kodem, gdzie mamy np. 10k linii kodu.

Rozwiązanie: podziel boską klasę na wiele klas, każda odpowiedzialna za inną funkcjonalność



Ciągłe starzenie

Wraz z rozwojem kolejnych technologii programiści mają problem z kompatybilnością lub kolejnymi rozwiązaniami, które nie są już rozwijane. Czy ktokolwiek pamięta jeszcze taką technologię jak Flash? Gorzej jest jak trzeba wspierać starą technologię.

Rozwiązanie: trzeba odpowiednio dobierać technologię i/lub pamiętać o tzw. *tech debt* i trzeba ciągle poprawiać naszą aplikację i aktualizować



Martwy kod

Zdarzają się czasami takie kawałki kodu (`if(false)`), które nigdy nie zostaną wykonane. Programiści robią to umyślnie, ponieważ chcą „chwilowo” wyłączyć kawałek kodu. Mogą to również robić z innego powodu.

Rozwiązanie: należy wykorzystywać możliwości, które daje nam git



Niejednoznaczność

Na tworzenie aplikacji można spojrzeć z kilku stron: biznesowej, specyfikacji oraz implementacji. Oznacza to, że możemy mieć inne spojrzenie z poziomu klienta, inżyniera jakości oraz programisty. To powoduje, że niektóre klasy mogą mieć zupełnie inne znaczenie w stosunku do innych. To powoduje trudności w zrozumieniu kodu.

Rozwiązanie: refaktoryzacja



No OO

Ostatnio ponownie bardzo popularne jest programowanie funkcjonalne. To dobrze do momentu aż Ci sami programiści nie zaczną projektować i implementować aplikacji, która ma być napisana w sposób obiektowy. Efektem tego jest aplikacja napisana, że mamy mix obu podejść i bardzo często wadliwe nazwy klas czy metod.

Rozwiązanie: lepsze planowanie/projektowanie, a w ostateczności refaktoryzacja



Duch

Z duchami mamy do czynienia w sytuacji, gdy mamy klasy, które przestały lub od samego początku nie stanowiły zbyt wielkiej wartości dla projektu, tj. są mało używane lub w ogóle.

Rozwiązanie: ghostbuster!



Kotwica

Kotwicą nazywamy kawałek kodu, który nie daje zbyt wiele wartości dodanej do aplikacji, ale z różnych powodów jesteśmy zmuszeni do wykorzystywania danego kawałka kodu.

Rozwiązanie: refaktoryzacja, chociaż jest często dość trudna, a czasami niewykonalna



Głowa w piasku

Gdy mamy aplikację/rozwiązanie, które stosujemy do rozwiązania konkretnego problemu i dalej, rozwiązujemy każdy inny problem za pomocą tej aplikacji. Nie widzimy innych rozwiązań, które mogą lepiej rozwiązać dany problem. Często programiści przywiązują się do jednej biblioteki/rozwiązania danej firmy.

Rozwiązanie: rozsądek ;)



Ślepa uliczka

Dochodzi do takich sytuacji, gdy korzystamy z zewnętrznej biblioteki, której wsparcie się zakończyło.

Rozwiązanie: wykorzystanie alternatywnego rozwiązania



Spaghetti

Kod trudny w zrozumieniu, często gdy mamy wiele odwołań do różnych metod w taki sposób jakby to był makaron. Wykorzystanie goto wzmacnia tylko efekt.

Rozwiązanie: głęboka refaktoryzacja



Bobas

Ten antywzorzec dotyczy danych wprowadzanych przez użytkowników. Źle napisana obsługa wejścia użytkownika powoduje, że aplikacja zaczyna się zachowywać nieprzewidywalnie.

Rozwiązanie: dopracowanie metod wejścia



Pole minowe

Do takiej sytuacji dochodzi, gdy mamy aplikację, która zawiera mnóstwo błędów, oraz brakuje jakichkolwiek testów. Naprawa błędów jest bardzo czasochłonna, a przechodzenie przez kod aplikacji przypomina chodzenie po polu minowym. Nigdy nie wiadomo, która zmiana w kodzie nie spowoduje, że aplikacja po prostu przestanie działać.

Rozwiązanie: głęboka refaktoryzacja + napisanie testów regresyjnych



Copy&Paste / Cut&Paste / Stackoverflow

Bardzo znany błąd popełniany przez niemal wszystkich programistów. Może on występować w kilku wariantach, w zależności od tego skąd kopiujemy. Takie podejście jest też często zaprzeczeniem podejścia DRY. W tym trzecim przypadku, korzystamy z rozwiązań dostępnych na Stackoverflow, chociaż poszczególne źródła mają nieco inne zastosowanie niż nasze. Często zdarza się, że takie kawałki kodu nie są dostosowywane do naszych potrzeb i potrafią „wybuchnąć” w nieoczekiwanych momentach.

Rozwiązania:

- zmiana podejścia,
- sprawdzanie tego co się kopiuje,
- refaktoryzacja kodu, aby był zgodny z DRY.



Programowanie na ślepo

Odseparowanie klienta od programistów może mieć potencjalnie dobre strony, ale prowadzi też często do tzw. programowania na ślepo.

Rozwiązanie: analityk biznesowy w projekcie lub inżynierowie jakości



Antywzorce architektury



Stożki migracyjne

Ostatnio modne są trzy typy: cloud, systemowe oraz enterprise. Dotyczy to kwestii przenoszenia danego rozwiązania do chmury. Innym przypadkiem jest wykorzystanie danego rozwiązania jako enterprise. W obu przypadkach mamy do czynienia z aplikacjami, które nie są dostosowane do danego podejścia, a zmiany wymagana są na poziomie architektury, co jest bardzo kosztowne. Istotnym elementem, który należy wziąć pod uwagę jest interoperacyjność.

W przypadku systemowych, mówimy o aplikacji słabej jakości, bez testów oraz ze starym kodem.

Rozwiązanie: głęboka refaktoryzacja



Zbieranina

Dotyczy to rozwiązań, które mają mieszane architektury oraz zależności pomiędzy nimi.

Rozwiązanie: rozdzielić architektury aplikacji - refaktoryzacja



Papierkowa robota

Istnieją projekty, gdzie podstawą jest dokumentacja. Ma to oczywiście pozytywny aspekt, ponieważ wszyscy wiedzą o czym jest projekt. Jednak jeżeli jest jej dużo, a brakuje priorytetyzacji lub inne sposoby na zarządzanie dokumentacją, to stracimy nad nią kontrolę i stworzy się chaos.

Rozwiązanie: priorytetyzacja dokumentacji/zasobów



Uzależnienie

Dotyczy to projektów, które zbyt mocno uzależniają się od jednego dostawcy. Przykładem może być np. npm oraz zależności pomiędzy paczkami. Usunięcie jednej od której zależały spowodowało paraliż wielu projektów opartych o JavaScript.

Rozwiązanie: odizolowanie rozwiązania dostawcy od reszty aplikacji



Wilczy bilet

Produkty, których używamy, które podają się za otwarte/transparentne, ale jednocześnie nie trzymają pewnych standardów. To ostatnie jest o tyle istotne, że bez takich gwarancji nasza aplikacja może być narażona na błędy.

Rozwiązanie: open source tak, ale ostrożnie - nie próbujemy zainstalować najnowszej biblioteki tylko dlatego, że jest „cool”



Ciepłe posadki

Niektóre projekty są na tyle duże, że potrzeba większych zespołów. Niestety nie wszyscy programiści mają dobry *performance* i zdarza się, że mamy zespoły gdzie jest kilku programistów jest dobrych, a reszta wykonuje przeciętną pracę.

Rozwiązanie: trzeba eliminować *ciepłe posadki*



Polityczna

Zdarza się, że architektura aplikacji jest tworzona na podstawie politycznych przekonań, a nie merytorycznie. Kończy się to skomplikowaną dokumentacją lub zastosowaniem nieodpowiednich technologii do danego problemu.

Rozwiązanie: spotkania, retrospekcja



Szwajcarski scyzoryk

Aplikacja, która robi wszystko.

Rozwiązanie: skupienie się na wartościach biznesowych



Odkrywanie koła na nowo

Próba tworzenia czegoś nowego od zera nie zawsze jest najlepszym rozwiązaniem. Zabiera czas i często nie wnosi dodatkowych wartości do projektów.

Rozwiązanie: korzystać z „gotowców”



Abstrakcja vs Implementacja

Nie wszyscy programiści potrafią myśleć abstrakcyjnie na tyle, aby móc to przełożyć na architekturę. Dotyczy to niestety większości programistów. Wiąże się z tym wiele problemów związanych z tworzeniem architektury aplikacji.

Rozwiązanie: odpowiedni dobór architekta



Antywzorce zarządzania projektami



Problemy w zespole



Problemy z komunikacją

Powód

Programiści skupiają się na zbyt wielu detalach

Rozwiązanie

Programiści powinni występować w roli ekspertów



Mowa nienawiści

Powód

Kiedy zespół nie dostarczy historyjek na czas

Rozwiązanie

Retrospekcja



Rambo

Powód

Kiedy jeden programista odpowiada za zbyt wiele części procesu w projekcie

Rozwiązanie

Zamiast jednego Rambo potrzebnych jest wiele *Ninjas*



Dokumentacja w TDD

Powód

Kiedy za dokumentację traktowane są testy

Rozwiązanie

Pisanie dokumentacji poza testami



Potok informacji

Powód

Dokumentacja znajduje się w zbyt wielu miejscach

Rozwiązanie

Unifikacja, osoba odpowiedzialna za utrzymanie dokumentacji



Zjadanie własnego ogona

Powód

Gdy programiści nie znają celu, „big picture”

Rozwiązanie

Retrospektywa



Boomerang lub zombie (bugi)

Powód

„Poprawione” bugi pojawiają się znowu lub wykonane zadania powracają na listę do zrobienia

Rozwiązanie

Czas na przygotowanie opisu zadań/historyjek



Ściana/Mur

Powód

Tworzone są bariery na poziomach menedżerowie - programiści

Rozwiązanie

Transparentność



Zacznij od siebie

Powód

Duże ego

Rozwiązanie

Nawet początkujący programiści są w stanie wnieść do projektu



Usprawnienia w zespole



Podstawowe zasady

Istnieje kilka podstawowych zasad:

1. Plany rozwoju
2. Trzymanie motywacji
3. Dzielenie się sukcesem
4. „Big Picture”



Umiejętności

Kilka istotnych umiejętności nad którymi warto pracować:

1. Prezentacja
2. Konferencje
3. Metryki
4. Kaizen
5. Planowanie długo- i krótkoterminowe



Komunikacja

Kilka zasad komunikacji w zespole:

1. Unikaj *buzzwordów*
2. Komunikacja techniczno<->biznesowa
3. Rysunki
4. Praca z zespołami zdalnymi



Usprawnienie iteracji

Pamiętaj o:

1. Planowaniu
2. Retrospekcji
3. Groomingu
4. Demo
5. Szukaj pola do działania



Jakość



Proces testowania (ang. test process)

Proces testowania składa się z kilku części:

- planowania i zarządzania testami,
- analizy i projektowanie testów,
- implementacja oraz wykonywanie testów,
- ewaluacji kryteriów wyjściowych (ang. exit criterias) oraz raportowania wyników,
- wykonywanie procesów końcowych testów (ang. test closure).



Przypadki testowe (ang. test case)

Przypadek testowy składa się z listy kroków z akcjami oraz oczekiwanym wynikiem każdej akcji. Na przykład, gdy chcemy przetestować autoryzację można stworzyć przypadek jak poniżej.

Krok	Akcja	Oczekiwany wynik
1	Przejdź na stronę: <code>http://localhost/login</code>	Formularz logowania jest widoczny z polami: <i>username</i> oraz <i>password</i> .
2	Wypełnij pola <i>username</i> oraz <i>password</i> danymi: <i>admin</i> oraz <i>secret</i> .	Pole <i>username</i> zostało wypełnione jawnym tekstem. Pole <i>password</i> zostało wypełnione tekstem niejawnym.
3	Kliknij przycisk <i>login</i>	Formularz został wysłany
4	Sprawdź obecną stronę po przekierowaniu	Strona po przekierowaniu to: <code>http://localhost/main</code>

Wykonywanie przypadków testowych jest prosty, ponieważ oczekiwane wyniki są jasno zdefiniowane.

Przypadki testowe mogą zostać wykorzystane do testów regresyjnych czy akceptacyjnych.



Przypadki testowe

Często jednak przypadki testowe pozwalają również pokryć negatywne scenariusze, gdzie wynikiem jest błąd.

Krok	Akcja	Oczekiwany wynik
1	Przejdź na stronę: <code>http://localhost/login</code>	Formularz logowania jest widoczny z polami: <i>username</i> oraz <i>password</i> .
2	Wypełnij pola <i>username</i> oraz <i>password</i> danymi: <i>admin</i> oraz <i>secret</i> .	Pole <i>username</i> zostało wypełnione jawnym tekstem. Pole <i>password</i> zostało wypełnione tekstem niejawnym.
3	Kliknij przycisk <i>login</i>	Formularz został wysłany
4	Sprawdź obecną stronę po przekierowaniu	Strona po przekierowaniu to: <code>http://localhost/main</code>
5	Sprawdź stronę z błędem (popup)	Wyświetlony został komunikat z błędem: "Błędny użytkownik/hasło. Spróbuj ponownie."



Zestawy testów oraz pokrycie testami

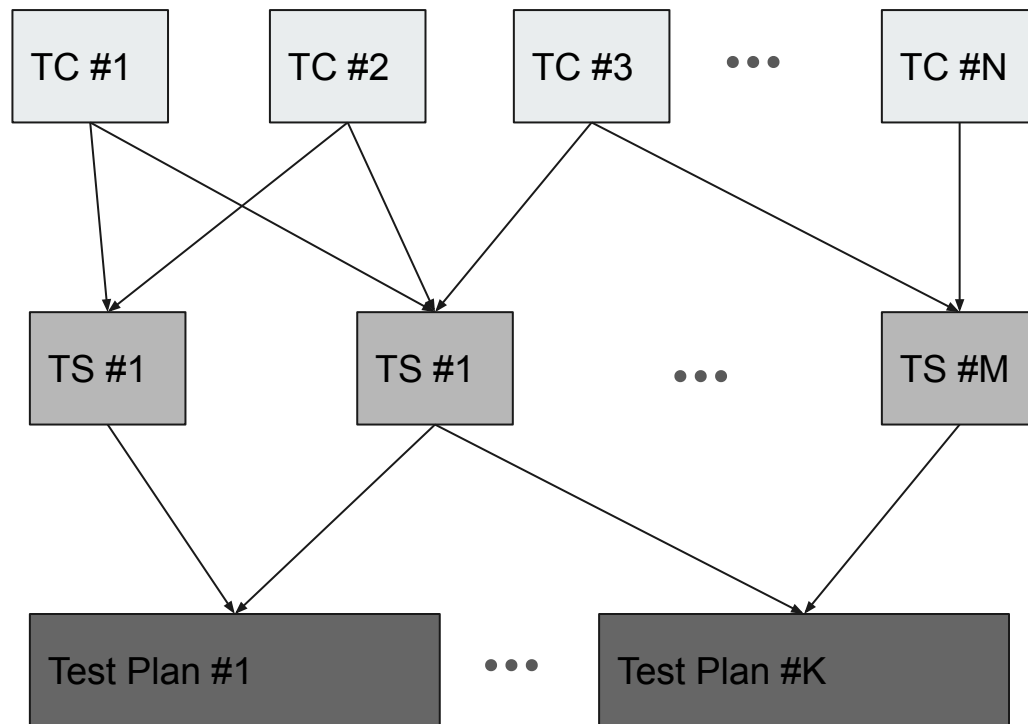
Zestaw testów (ang. test suites) składają się z przypadków testowych. Zestawy zawierają przypadki, które testują konkretną funkcjonalność.

Pokrycie testami (ang. test coverage) jest określane procentowo w jakim stopniu testy pokrywają kod źródłowy. Pokrycie dotyczy zarówno manualnych jak i automatycznych testów, np. 50% kodu może mieć pokrycie w testach automatycznych i jednocześnie dodatkowe 30% w testach manualnych. Możemy pokrycie testami odnieść do konkretnych funkcjonalności aplikacji. Wyższe pokrycie oznacza wyższą jakość aplikacji.

Test plans

TC - przypadek testowy

TS - zestaw testów

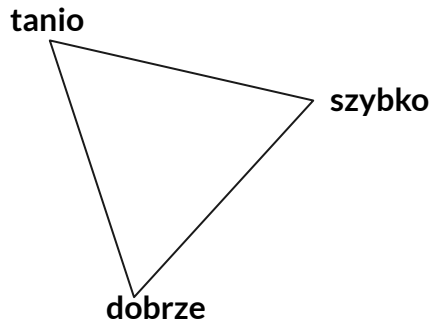




Kiedy należy zakończyć testowanie?

To jest bardzo dobre pytanie!

Istnieją trzy wyznaczniki tworzenia oprogramowania, tj. oprogramowanie można zrobić szybko, tanio albo dobrze. Nie należy znacząco skupiać na tylko jednym z nich. Najczęściej wybierane są dwa wyznaczniki. Najlepsze wyniki osiągamy jednak będąc dokładnie po środku. Jeżeli chcemy skupiać się na jednym z wyznaczników, niech to będzie jakość.





Rodzaje testów



Rodzaje testów

Istnieje wiele typów testów. Poniżej przedstawione zostały niektóre (angielskie nazwy).

- Unit testing
- Integration testing
- System testing
- Sanity testing
- Smoke testing
- Interface testing
- Regression testing
- Alpha testing
- Beta testing
- Acceptance testing
- Security testing
- Performance Testing
- Load testing
- Stress testing
- Functional testing
- Boundary value testing
- Exploratory testing
- Happy path testing
- Monkey testing
- Grey testing
- Browser compatibility testing



Testy czarnoskrzynkowe (ang. black box tests)

Testy czarnej skrzynki działają na takiej zasadzie jak nazwa wskazuje, tzn. aplikacja traktowana jest jak czarna skrzynka, czyli nie znamy kodu aplikacji. Z tego powodu należy przygotować odpowiednie dane wejściowe. Dane wyjściowe otrzymane z aplikacji porównywane są z oczekiwanym wyjściem. Inna nazwa dla testów czarnej skrzynki to testy funkcjonalne, ponieważ dotyczą funkcjonalności aplikacji.





Testy białoskrzynkowe (ang. white box tests)

Testy białej skrzynki różnią się od czarnoskrzynkowych tym, że w przypadku testów białej skrzynki znamy kod aplikacji. W tym przypadku testy automatyczne tworzone są przez programistów (zgodnie z ISTQB). Należy w testach jednostkowych przetestować klasy, metody, funkcje oraz przekazywanych parametrów.



Testy szarej skrzynki (ang. grey box testing)

Testy szarej skrzynki są kombinacją testów czarnej i białej skrzynki. W tym przypadku testujemy aplikację tak za pomocą testów funkcjonalnych mając jednocześnie wiedzę na temat kodu aplikacji.



Wydajnościowe, obciążeniowe i stres testy

(ang. performance, load and stress tests)

Często obciążeniowe i stres testy są często klasyfikowane jako wydajnościowe. Nie jest to jednak prawdą i każdy typ powinien być rozpatrywany jako osobna kategoria.

Testy wydajnościowy mają na celu znalezienie słabych stron aplikacji. Metrykami wykorzystywanymi są m.in. za czas odpowiedzi, skalowalność czy zużycie zasobów. Celem jest znalezienie miejsc, które należy poprawić pod kątem wydajności.

Testy obciążeniowe mają za zadanie sprawdzenie jak zachowa się aplikacja przy maksymalnym obciążeniu. Najczęściej testy takie trwają wiele godzin czy dni, ponieważ niektóre błędy można wykryć przy dłuższym wykorzystaniu aplikacji, np. wycieki pamięci.

Stres testy mają na celu sprawdzenie jak zachowa się aplikacja przy znacznie większym od maksymalnego obciążenia. Celem jest sprawdzenie zachowania aplikacji w przypadku nagłego skoku obciążenia aplikacji, np. sklep internetowy w okresie świątecznym.



Testy regresyjne (ang. regression tests)

Testy regresyjne lub regresywne to takie testy, które są wykonywane zawsze gdy wprowadzamy zmiany w funkcjonalności aplikacji. Najczęściej wszystkie zestawy testów są wykonywane po każdej zmianie, włączając automatyczne jak i manualne testy. Celem testów regresyjnych jest znalezienie błędów w istniejących funkcjonalnościach, które działały poprawnie do tej pory. Chodzi o sprawdzenie czy zmiany nie spowodowały pojawienie się nowych błędów w działających funkcjonalnościach.



Bugi



Przyjaciel Bug

Nie wykryte błędy są niebezpieczne, ponieważ często są tykającymi bombami. Nigdy nie wiadomo, kiedy takie błędy spowodują znaczne straty. Nigdy nie wiadomo ile takich błędów mamy w kodzie, ale staramy się doprowadzić do jak najmniejszej ich liczby. Dlatego więcej wykrytych błędów wcale nie musi znaczyć gorszej jakości oprogramowania.

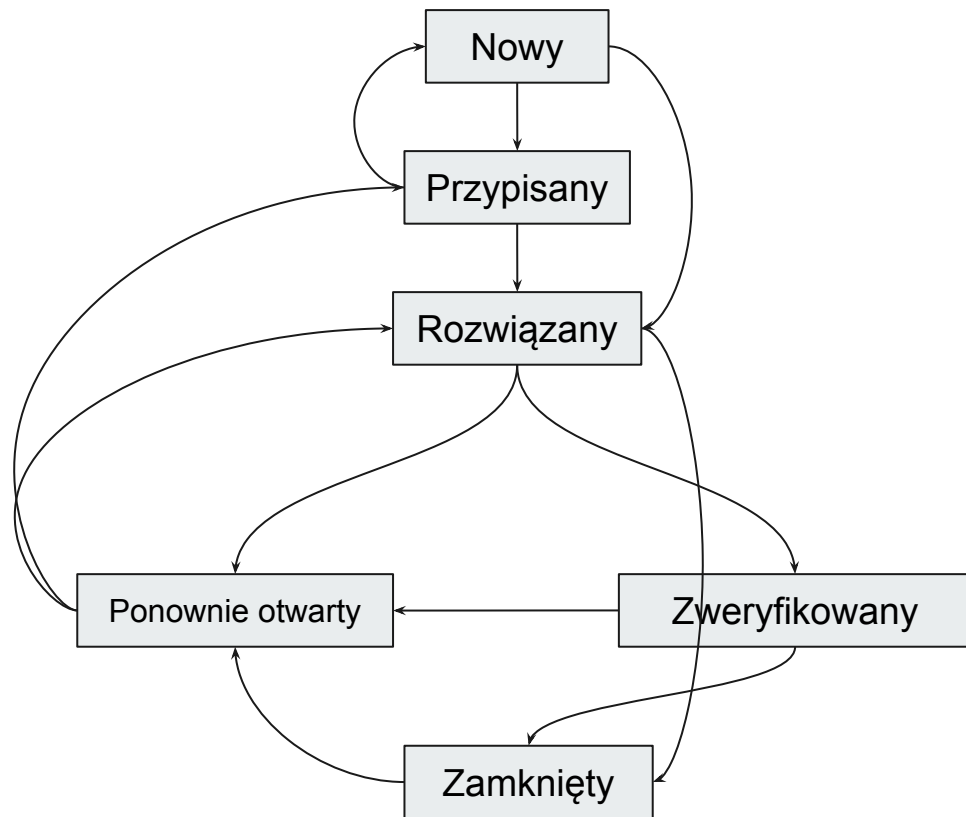
Liczba błędów może być metryką mówiącą o jakości oprogramowania. Wszystko zależy od ilości ich wykrycia i częstotliwości.

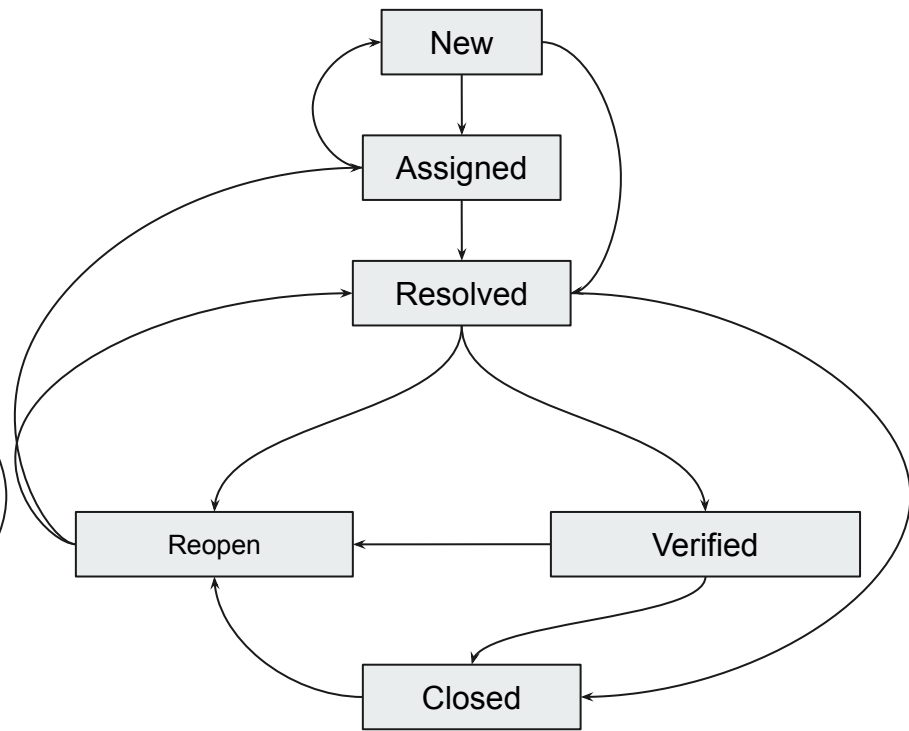
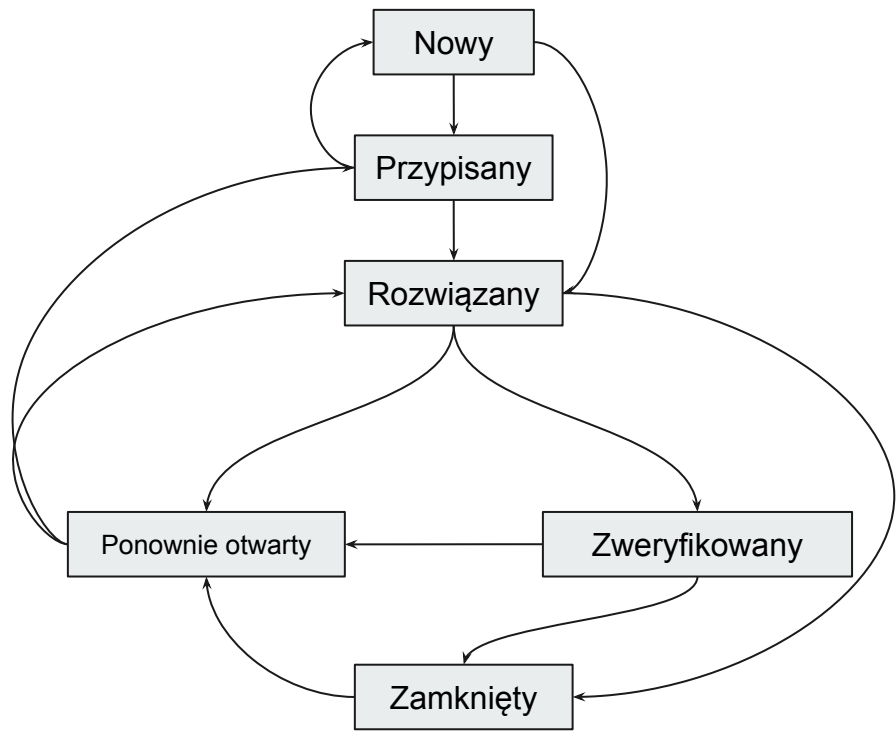
Życie bugów

(ang. Bug lifecycle)

Staramy się zawsze doprowadzić do zamknięcia buga.

Każdy błąd może zostać potraktowany w innych sposób. Rozwiązany może być jako: duplikat (ang. **duplicated**), poprawiony (ang. **fixed**), nieprawidłowy (ang. **invalid**), przeniesiony (ang. **moved**), nierozwiązany (ang. **won't fix**) oraz działa (ang. **works for me**).







Raportowanie *bugów*

Tytuł

Priority

Moduł

Dotkliwość

Przypisany

Status

Kroki

1.
2.
3.
4.

Oczekiwany wynik

Uzyskany wynik



Metodologie



Podejścia w tworzeniu oprogramowania

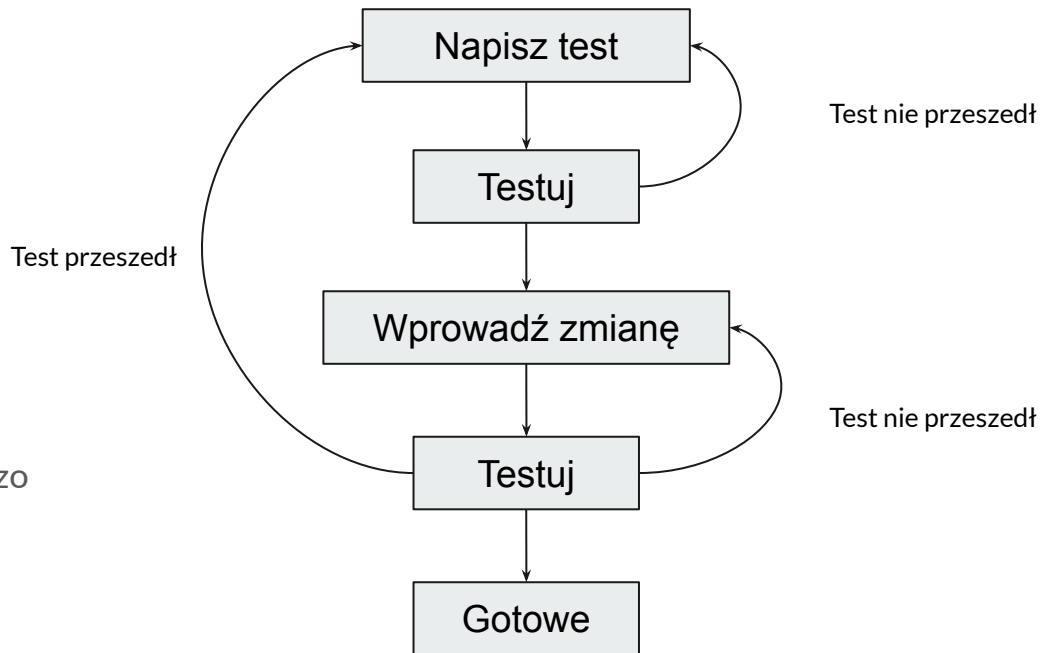
Istnieje wiele podejść, niektóre podane zostały poniżej w j. angielskim:

- acceptance test-driven development (ATDD),
- behavior-driven development (BDD),
- cross-functional team,
- continuous integration (CI),
- domain-driven design (DDD),
- iterative and incremental
- pair programming,
- planning poker,
- refactoring,
- scrum events (sprint planning, daily scrum, sprint review and retrospective),
- test-driven development (TDD),
- agile testing,
- timeboxing,
- user story,
- story-driven modeling,
- retrospective,
- velocity tracking,
- user story mapping.

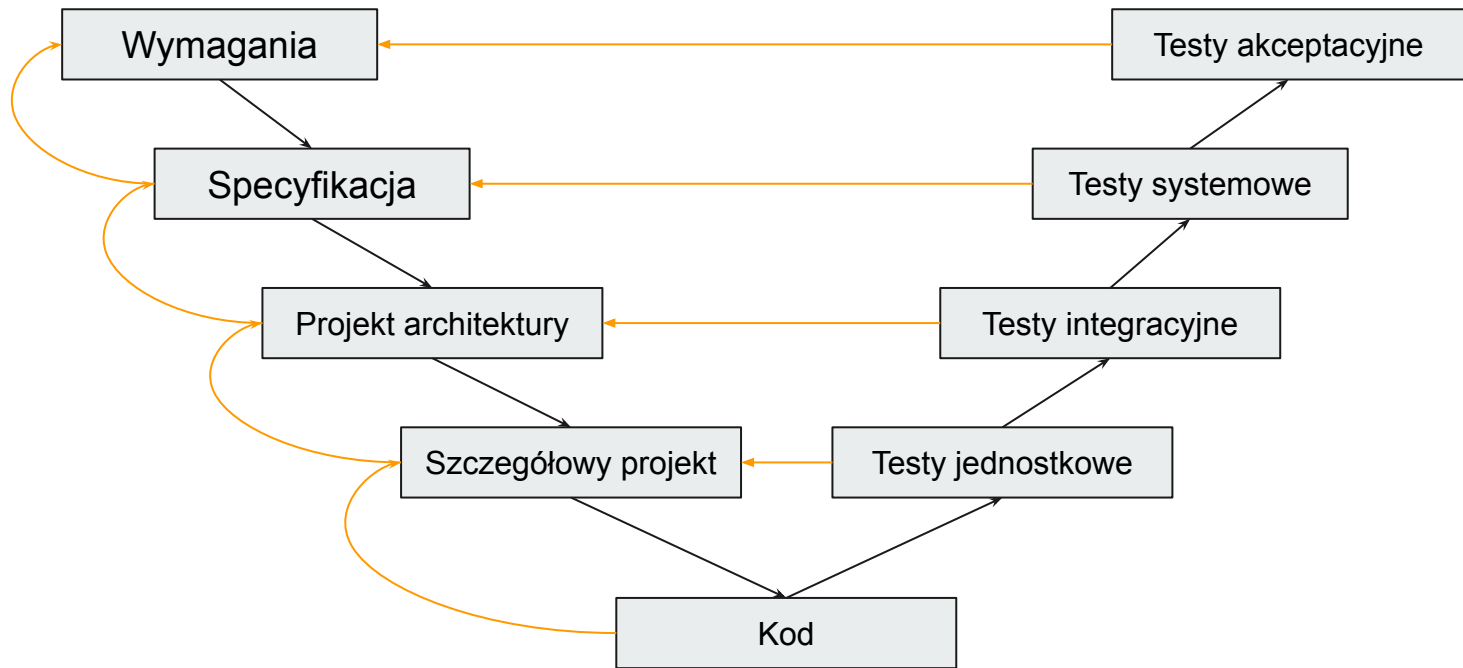
TDD

W podejściu TDD testy pisane są przed implementacją funkcjonalności.

Zaletą tego podejścia jest pokrycie funkcjonalności w testach, ale jest bardzo trudne we wdrożeniu.



V-model

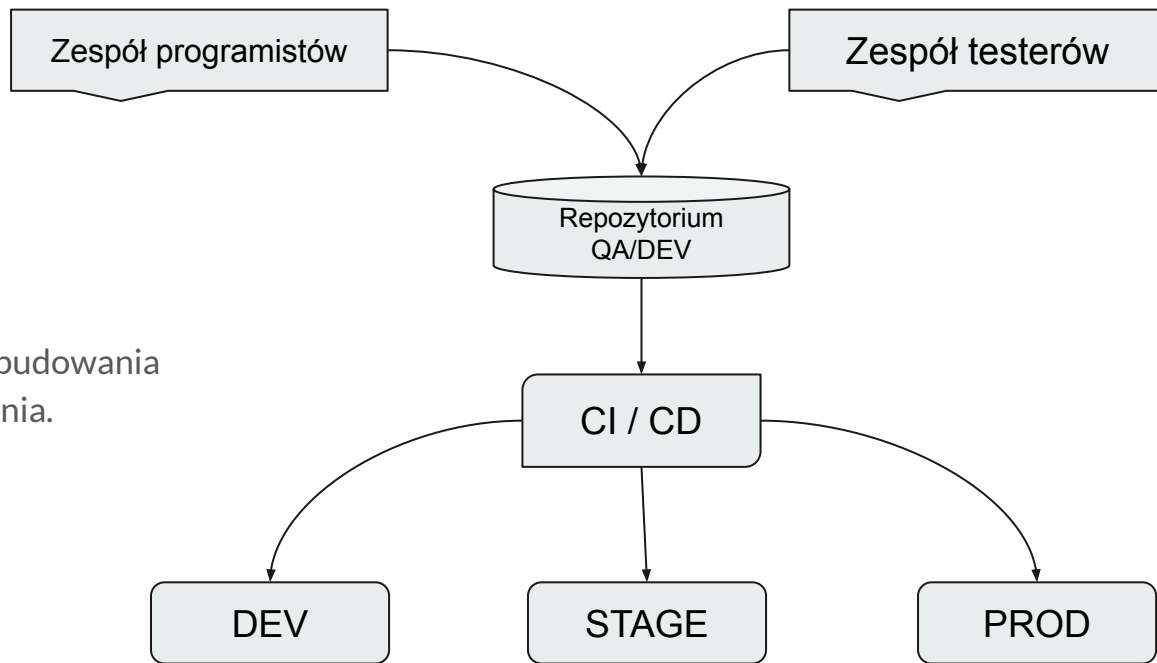


Continuous integration

Mamy trzy środowiska:

- development,
- staging,
- production.

Continuous integration służy do budowania aplikacji, testowania oraz wdrażania.





Continuous delivery

Różnica pomiędzy CI a CD jest taka, że w tym drugim przypadku, aplikacja jest budowana, testowana oraz wdrażana przy każdym wprowadzeniu zmian w kodzie.



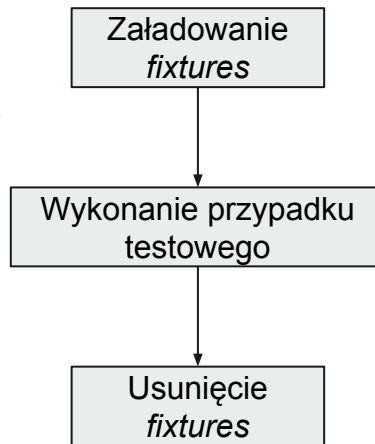
Dodatkowe terminy związane z jakością



Fixtures

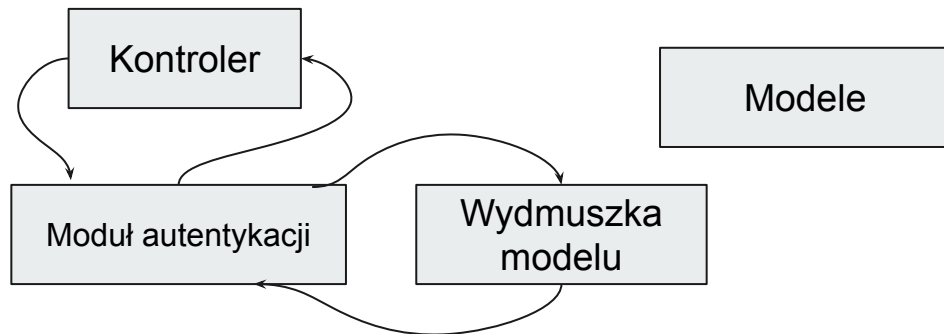
Fixtures to dane przygotowane na potrzeby testów. Bardzo często do przetestowania funkcjonalności wymagane są dane, np. dane do logowania czy dane, które należy pobrać za pomocą aplikacji. Aby test był możliwy do przeprowadzenia, należy najpierw wypełnić bazę odpowiednimi danymi, które są ładowane i znane są jako fixtures. Często przechowywane są w formacie XML, YAML, JSON czy SQL.

Bazda powinna zostać wyczyszczona po wykonaniu każdego przypadku testowego.



Wydmuszka (ang. mock)

Nie zawsze możemy lub chcemy przetestować całą aplikację. Niektóre funkcjonalności trzeba przetestować osobno. Trudno jednak przetestować niektóre funkcjonalności nie angażując w testy inne. Przykładem może być autentykacja. Możemy jednak stworzyć model użytkownika za pomocą wydmuszki (mock), który symuluje działanie modelu użytkownika. To samo możemy zrobić z niemal każdą klasą, serwisem czy biblioteką.





Wzorzec POM (Page Object Model)

Page Object Model jest wzorcem bardzo popularnym w tworzeniu testów automatycznych. Jedną z zalet tego wzorca jest łatwa utrzymywalność kodu napisanego zgodnie ze wzorcem POM. POMem jest każda strona aplikacji, tj. strona reprezentowana jest przez klasę. Dzięki temu wraz ze zmianą elementów na stronie, należy jedynie zmienić kod w danej klasie.



Literatura

1. **Scala Design Patterns - Second Edition**, Ivan Nikolov. Packt 2018
2. **Head First Design Patterns**, Kathy Sierra, Bert Bates, Elisabeth Robson, Eric Freeman. O'Reilly 2014
3. **Design Patterns Explained Simply**, Alexander Shvets. Sourcecmaking.com
4. **Clean Code**, Robert C. Martin. Addison-Wesley Professional 2016
5. **Clean Code Applied**, Robert C. Martin. Addison-Wesley Professional 2017
6. **Refactoring to Patterns**, Joshua Kerievsky, Addison-Wesley Professional 2004
7. **Refactoring JavaScript**, Evan Burchard. O'Reilly 2017
8. **Reactive Design Patterns**, Roland Kuhn, Brian Hanafée, Jamie Allen. Manning 2017
9. **Software Design X-Rays**, Adam Tornhill. Pragmatic 2018
10. **Beyond Legacy Code**, David Scott Bernstein. Pragmatic 2015
11. **The Nature of Software Development**, Ron Jeffries. Pragmatic 2015
12. **Test Driven Development**, Kent Beck. Pragmatic 2002
13. **The Clean Coder: A Code of Conduct for Professional Programmers**, Robert C. Martin. Addison-Wesley Professional 2011
14. **Growing Object-Oriented Software**, Guided by Tests, Steve Freeman, Nat Pryce. Addison-Wesley Professional 2009
15. **TO-DO Team! Simple productivity techniques for improving your team and making the software that matters**, Karol Sójko 2015