



# 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](mailto:karol.przystalski@uj.edu.pl)

[kprzystalski@gmail.com](mailto:kprzystalski@gmail.com)

[karol@codete.com](mailto: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.  $\lambda$ -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.

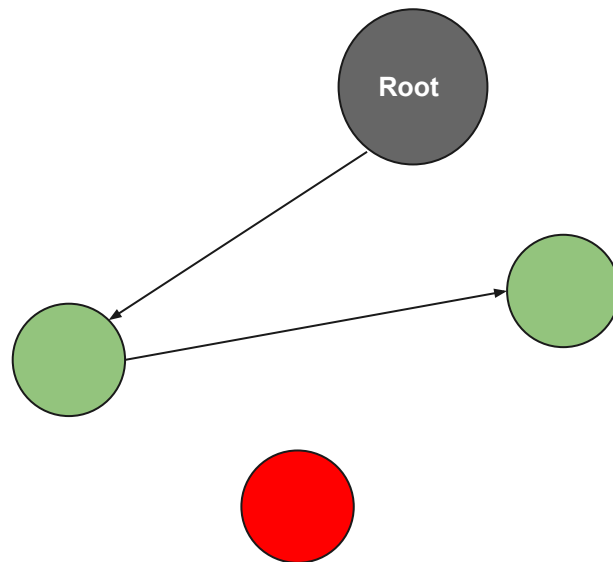
Istnieje wiele implementacji GC:

- 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

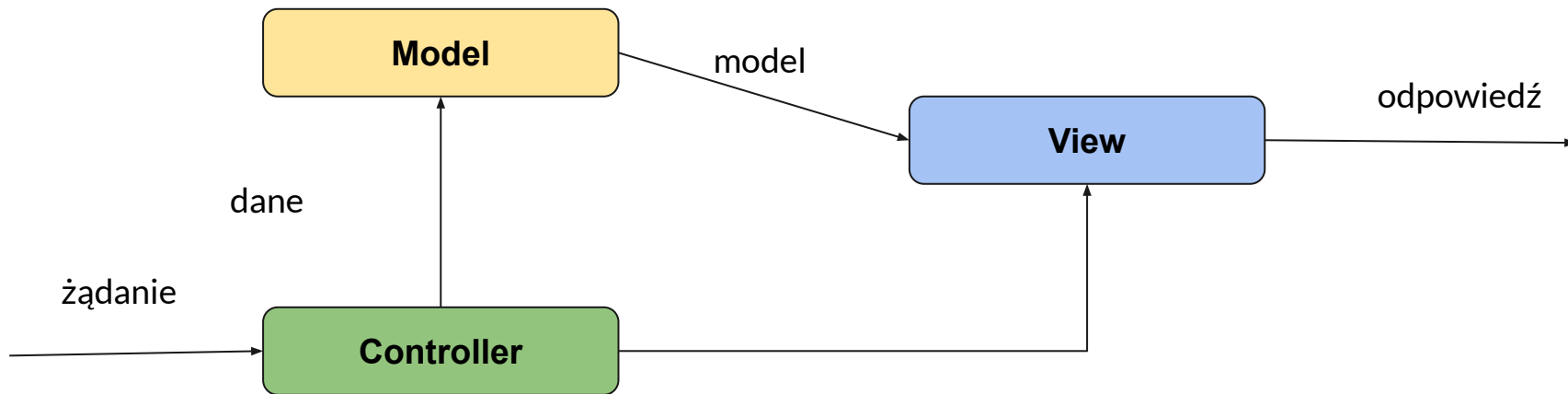
Dwuwarstwowe - przykład: architektura klient-serwer


Trójwarstwowe - przykład: MVC

Wielowarstwowe



# MVC





# MVC - zalety

Logika biznesowa oddzielona jest od widoku,

Nie ma zależności modelu od widoku

Uporządkowany kod



# MVC - wady

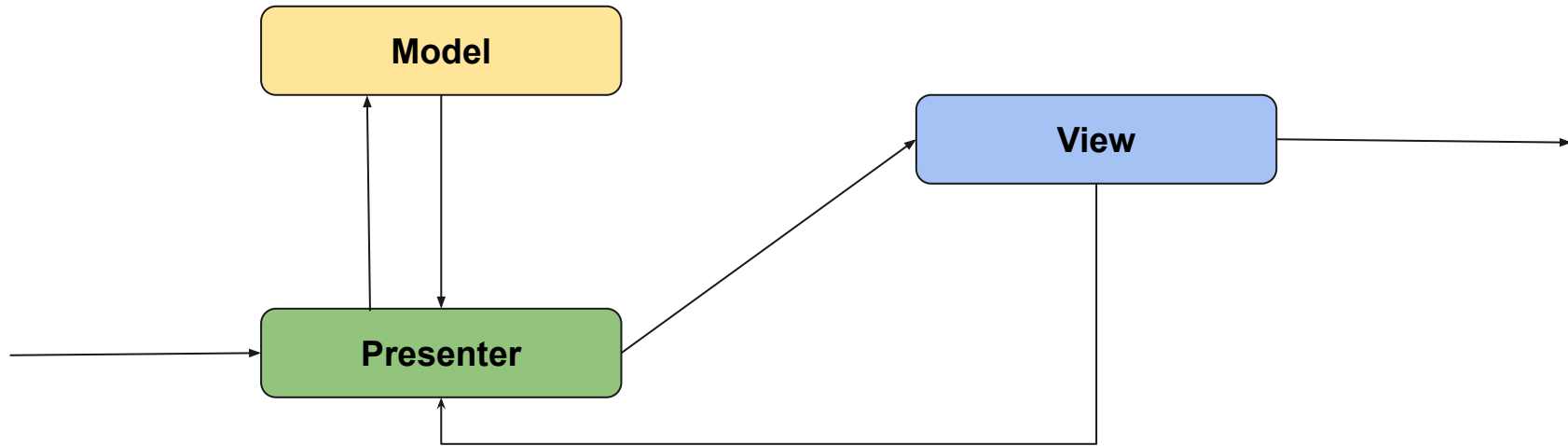
Złożoność aplikacji

Widok zależny od modelu


Widoki są trudne w testowaniu



# MVP







# MVP - zalety

Pochodna MVC

Oddzielona logika od widoku

Brak zależności modelu od widoku,

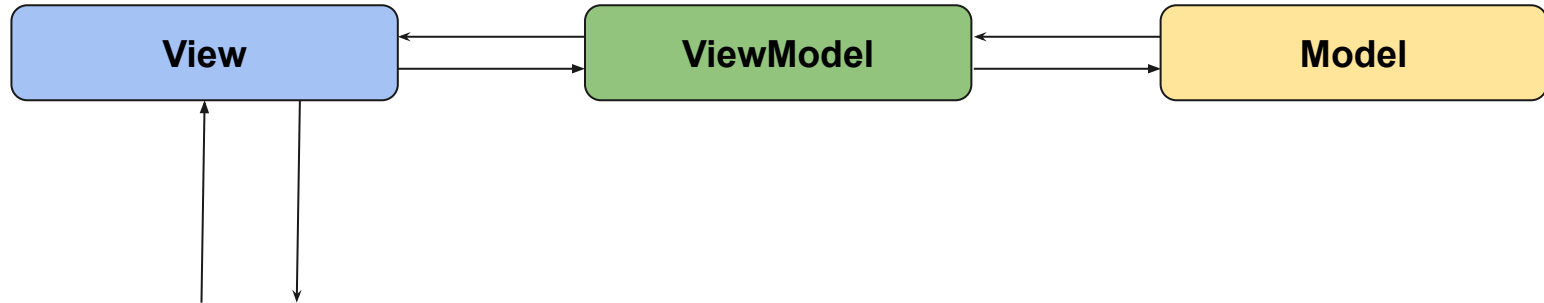



# MVP - wady

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.