software
development
academy

# Clean code i refaktoryzacja

przygotował: Łukasz Bartnicki

# Wstęp

- Łukasz Bartnicki
- Edukacja: informatyka na Wydziale Matematyki i Informatyki UWr (I st), informatyka na Wydziale Informatyki i Zarządzania PWr (mgr)
- Wykładowca na studiach podyplomowych WSKZ (linear algebra for data scientists, software engineering, introduction to machine learning)
- Software engineer i big data developer m.in w Roche (biotech, diagnostic area) oraz w Thomson Reuters
- Mentor, Prelegent na wydarzeniach skierowanych do programistów
- Prowadzący zajęcia, kształcące przyszłych oraz obecnych programistów (współpraca między innymi z SDA)

# Cel zajęć

Przedstawienie **koncepcji, metod** i narzędzi, które mogą być pomocne podczas procesu zwiększania jakości implementacji, abstrahując od optymalizacji

# Agenda

1. Praca z zastanym kodem
2. Refaktoryzacja do wzorców projektowych
3. Dekompozycja klas - rozbijanie olbrzymów
4. Metody testowania kodu przed wprowadzeniem zmian
5. Rekonstrukcja testów po dekompozycji klas
6. Zasady bezpiecznej refaktoryzacji

# Agenda

7. Usuwanie nadmiarowego kodu
8. Techniki usuwania zależności
9. Rozdzielanie danych od logiki
10. Wymuszanie enkapsulacji
11. Wykorzystanie obiektów reprezentujących operacje
12. Wykorzystanie interfejsów do rozluźnienia zależności

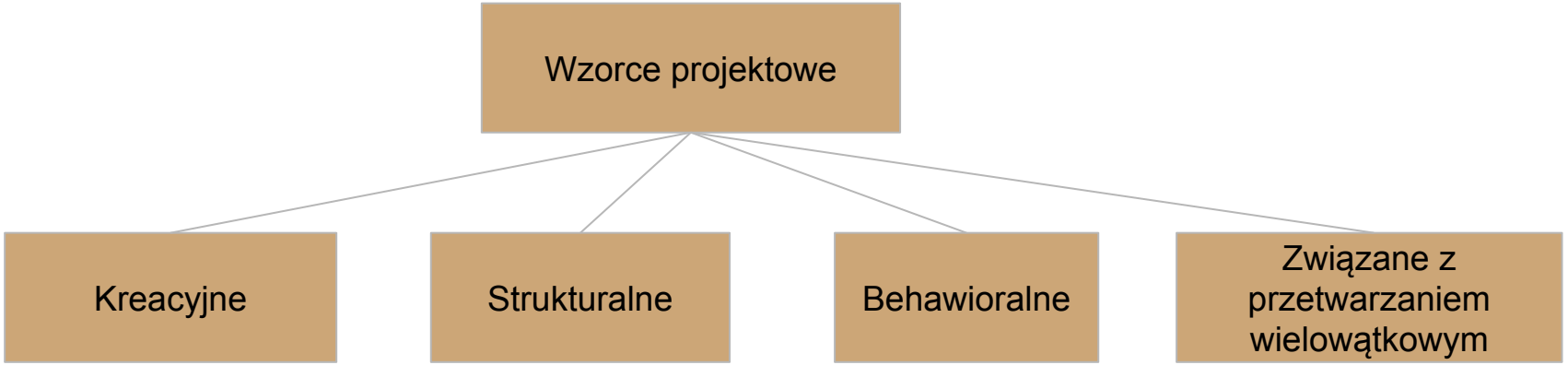
# Praca z zastanym kodem - wstęp

- Ang. legacy code - kod odziedziczony
- W takim kodzie mogą wystąpić problemy, dotyczące
  - odpowiedniego pokrycia testami automatycznymi
  - dokumentacji
  - czytelności kodu
  - stosowania konwencji
  - odpowiedniego projektu logiki biznesowej (jak najmniej zależności, łatwa i bezpieczna modyfikowalność, ...)

# Praca z zastanym kodem - podejścia

- Przegląd dokumentacji
- Refaktor zamiast przepisywania
- Pomiar jakości przed zmianami
- Statyczna analiza kodu => zebranie dodatkowych informacji
- Zasadnicze zmiany (warto zacząć od podziału na elementy w jak najmniejszym stopniu zależne od siebie)
- Pokrycie testami
- Wybór odpowiedniej osoby do code review
- Pomiar jakości po zmianach

# Refaktoryzacja do wzorców projektowych




Singleton
Builder
Factory method
Abstract factory
Prototype
Lazy initialization
Dependency injection
...

Decorator
Adapter
Facade
Bridge
Proxy
Composite
Flyweight
Delegate
...

Strategy
Observer
Mediator
Visitor
Template method
Chain of responsibility
Iterator
Fluent interface
...

Active Object
Lock
Thread Pool
...

# Refaktoryzacja do wzorców projektowych - kreacyjnych

- Instancjonowanie obiektu ⇔ gdy nastąpi odwołanie do niego
- Kontrola ilości instancji
- Dostarczenie abstrakcji dot. kreowania obiektów i klas pochodnych dot. wyspecjalizowanych obiektów
- Ułatwienie robienia instancji bardziej skomplikowanych obiektów
- Oddelegowanie do frameworka / serwisu robienia instancji, aby nie powstawały bezpośrednie zależności
- Usuwanie obiektów, które nie są już potrzebne
- Tworzenie nowych obiektów ze „szkieletu” istniejącego obiektu, zwiększając w ten sposób wydajność i ograniczając zużycie pamięci

# Refaktoryzacja do wzorców projektowych - strukturalnych

- Dostarczenie uproszczonego interfejsu, który “przykrywa” bardziej skomplikowan[y/e] interfejs[y]
- Umożliwienie współpracy klas, które mają niekompatybilne interfejsy
- Dynamiczne rozszerzanie logiki obiekty klasy A, bez modyfikacji kodu A
- Rozdzielenie hierarchii klas, w której jest abstrakcja na dwie hierarchie - jedną związana z abstrakcją, drugą - z implementacją
- Kontrola dostępu do oryginalnego obiektu, umożliwiając wykonanie pewnej logiki, przed lub po wykonaniu metody na oryginalnym obiekcie
- Komponowanie obiektów do struktury drzewiastej (reprezentacja hierarchii część-całość)
- Rozszerzanie klasy o kompozycję zamiast implementacji podklasy
- Współdzielenie identycznych części, należących do dużej liczby obiektów

# Refaktoryzacja do wzorców projektowych - behawioralnych

- Redukuje chaotyczne, bezpośrednie zależności między obiektami, wymuszając komunikację poprzez obiekt pośredniczący
- Separacja algorytmu od obiektu na bazie którego ten algorytm działa
- Implementacja mechanizmu publisher-subscriber
- W klasie nadrzędnej znajduje się szkielet algorytmu (niektóre metody są abstrakcyjne) a w klasach pochodnych znajdują się implementacje metod abstrakcyjnych
- Ciąg powiązanych obiektów, w którym każdy obiekt zajmuje się obsługą pewnej operacji. Obiekt i może oddelegować operację x do obiektu i+1 w przypadku gdy obiekt i nie zajmuje się obsługą x
- Definicja rodziny algorytmów, które mogę być parametrami pewnego obiektu
- Łańcuch wywołań funkcji w postaci f1(..).f2(..). ... .fn(..)

# Refaktoryzacja do wzorców projektowych

Przykład

# Refaktoryzacja do wzorców projektowych

Zadanie
Przeanalizuj kod w pakiecie taxes.beforefactor i zastanów się jakie problemy powoduje / może powodować taka implementacja.
Dokonaj refaktoru, tak aby wyeliminować te problemy. Rezultat umieść w pakiecie  taxes.afterrefactor

# Refaktoryzacja do wzorców projektowych

Zadanie
Przeanalizuj kod w pakiecie newlibrary.beforerefactor. Treść zadania znajduje się w komentarzach w klasie Main. Rozwiązanie umieść w pakiecie newlibrary.afterrefactor.

# Dekompozycja klas - rozbijanie olbrzymów

- Istnieją różne źródła odnoszące się do maksymalnej liczby linii w klasach i metodach
- Dobrą praktyką jest żeby każdy deweloper miał w IDE takie same ustawienia dot konwencji stosowanych w kodzie
- Z dużej liczby linii w klasach może wynikać:
  - złamanie zasady pojedynczej odpowiedzialności
  - utrudnione testowanie
  - utrudniona analiza kodu
  - utrudniony rozwój i modyfikacje

# Dekompozycja klas - podejścia

- Najważniejsze jest podejście że lepiej zapobiegać niż leczyć
- Rozwiązania, które mogą pomóc w redukcji rozmiaru klasy:
  - zastosowanie agregacji / delegate design pattern
  - architektura trójwarstwowa
  - architektura oparta o mikroserwisy
  - strategia
  - ....

# Dekompozycja klas

Przykład

# Dekompozycja klas - duże zbiory

Zadanie

a) Przeanalizuj kod w pakiecie largesetofclasses.beforerefactor
b) Korzystając z https://app.diagrams.net/ owórz plik largesetofclasses.redesig/solution.drawio i przeanalizuj ideę która będzie prowadziła do przeprojektowania rozwiązania z pakietulargesetofclasses.beforerefactor
c) Poprzez funkcję f(n, k) przedstaw wzór na liczbę wszystkich klas w rozwiązaniu “przed refaktorem” na diagramie. Ile klas należałoby dodać do pakietu largesetofclasses.beforerefactor w przypadku dodania obsługi nowego systemu operacyjnego, zakładając że chcemy obsługiwać takie i tylko takie typy kontrolek jak w istniejących systemach ?
d) Analogicznie do punktu c odpowiedz na pytania w kontekście rozwiązania “po refaktorze”
e) Bazując na diagramie klas “po refaktorze” z pliku solution.drawio, przeprojektuj rozwiązanie z pakietu before (nowe rozwiązanie umieść w pakiecie after)

# Metody testowania kodu przed wprowadzaniem zmian

- Przed wprowadzeniem danej zmiany, należy wykonać wszystkie rodzaje testów, które pokrywają te zmiany:
  - testy jednostkowe
  - testy integracyjne
- Można zapisać wyniki pokrycia (raport) używając:
  - IDE -> run unit test with coverage
  - Opowiedniego plugina do generowania raportu pokrycia (np. JaCoco Maven plugin dla projektów w Javie)
- Jeśli zmieniany fragment nie jest pokryty testami jednostkowymi, można:
  - wykonać testy manualne dotyczące logiki biznesowej przed jej zmianą
  - zastosować podejście TDD, implementując najpierw testy, pokrywające logikę, która będzie zmieniona

# Rekonstrukcja testów po dekompozycji klas

- Dla każdej klasy, która powstała w wyniku dekompozycji, powinna powstać odpowiednia klasa, związana z testem automatycznym
- Należy rozważyć zarówno dekompozycję testów jednostkowych jak i integracyjnych
- Do podziału klasy zw. z testami automatycznymi, można użyć wsparcia IDE (ekstrakcja do osobnej klasy, wybranych metod testowych)
- Po dekompozycji klasy testowej, należy uruchomić wszystkie testy (metody testowe), które do niej należą i wynik porównać z raportem pokrycia, wygenerowanym przed refaktorem
- Ten etap jest okazją do zaobserwowania brakujących pokryć - w takim przypadku, należy w odpowiedni sposób zaznaczyć o takim braku w celu uzupełnienia w ramach innego zadania

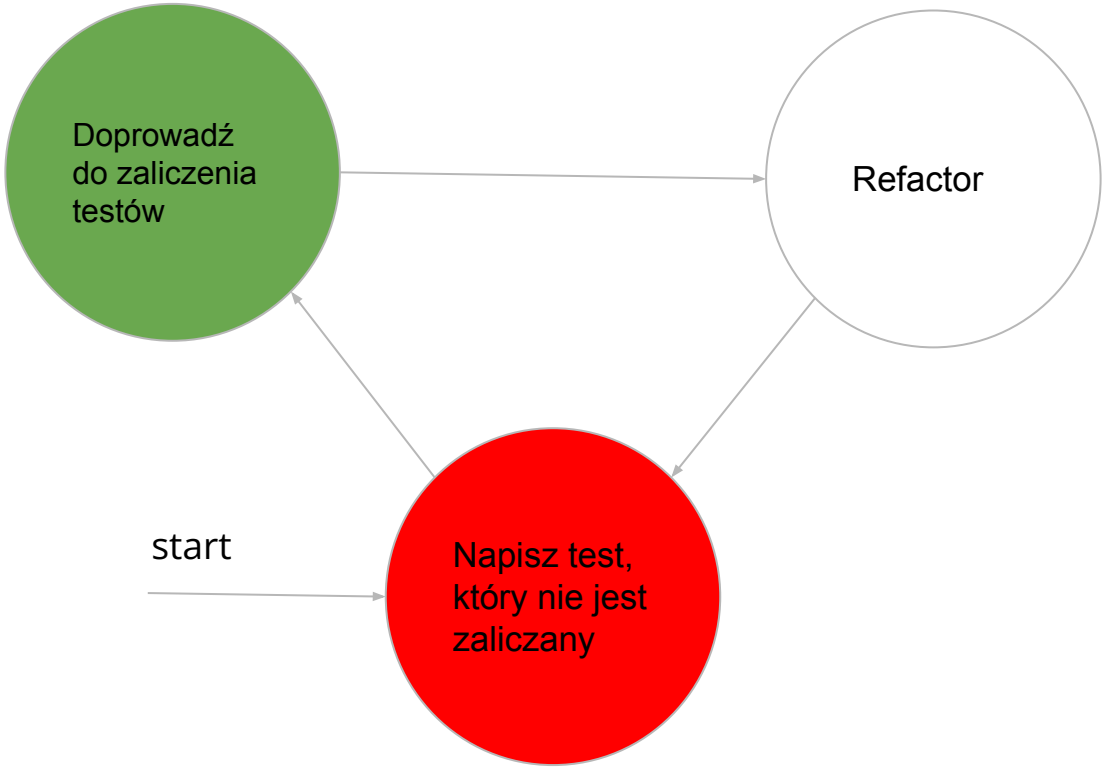
# Zasady bezpiecznej refaktoryzacji

- Dokonywanie zmian małymi krokami
- Należy dobrze zdefiniować i opisać zakres refaktoru
- Szczególnie należy uważać na refaktor fragmentów, które nie są pokryte testami
- Po każdej zmianie, należy od razu wykonać testy automatyczne, które ją pokrywają
- Refaktor należy wykonać przed dodaniem nowej funkcjonalności, szczególnie tej dużej

# Zasady bezpiecznej refaktoryzacji

- Należy unikać sytuacji refaktoru podczas równoległych prac innych developerów (bugfixy, nowe funkcjonalności)
- Przed wykonaniem danej czynności manualnie, należy zastanowić się czy nie istnieje funkcjonalność IDE, która automatyzuje ten proces
- Nie należy popadać w skrajny perfekcjonizm => przed rozpoczęciem refaktoru danego fragmentu, należy zastanowić się jakie korzyści da ten proces

# Zasady bezpiecznej refaktoryzacji - technika oparta o TDD



# Usuwanie nadmiarowego kodu

- Nadmiarowy kod jest jednym z przypadków tzw. “code smells”
- Detekcja przy użyciu wbudowanych funkcjonalności IDE nie zawsze daje dobre rezultaty
- Pomocny może być odpowiedni plugin do IDE (np. dla IntelIJ - SonarLint)
- Znaczne bezpieczniejszym rozwiązaniem może być integracja projektu z narzędziem do statycznej analizy kodu np. SonarCube (wsparcie dla Java i C#)
- Należy pamiętać żeby zwrócić uwagę na duplikaty podczas code review
- Nie należy przesadzać ; )

# Usuwanie nadmiarowego kodu

Przykład

# Techniki usuwania zależności - wstęp

- Bardziej chodzi o redukcję stopnia zależności
- SOLI**D** - dependency inversion principle - obiekty powinny być zależne od abstrakcji a nie od konkretnej klasy
- Powiązania pomiędzy klasami powinny być luźne
- Klasy wyższego poziomu nie powinny być zależne od klas niższego poziomu
- Abstrakcje nie powinny być zależne od szczegółów

# Techniki usuwania zależności

Przykład

# Techniki usuwania zależności

Zadanie
Przy pomocy https://app.diagrams.net/ otwórz diagram dependencies/companychat/before/wrongdesign.drawio
który przedstawia relacje pomiędzy klasami pewnej implementacji.
Załóżmy że wszyscy pracownicy należą do tego samego zespołu, który pracuje nad tym samym projektem, więc dowolny pracownik powinien mieć możliwość komunikacji z dowolnym innym pracownikiem.
W przypadku trzech pracowników, będzie więc sześć powiązań pomiędzy wszystkimi obiektami.
Ile powiązań tego typu  będzie w przypadku n pracowników?
Przeprojektuj rozwiązanie (zaimplementuj kod), do takiej postaci, żeby w przypadku n pracowników, całkowita liczba powiązań (pomiędzy obiektami wszystkich klas) wynosiła 2n

# Techniki usuwania zależności

- Wsparcie IDE, pluginów, do detekcji nieużywanych obiektów
- Dependency injection
- Service Locator
- Inversion of Control
- Mediator

# Rozdzielenie danych od logiki

- W tym momencie poprzez dane rozumiemy:
  - pliki konfiguracyjne
  - dane, które zamiast wpisane w kod, mogą znajdować się w bazie danych
- Korzyści separacji danych od logiki:
  - Brak konieczności deploymentu skompilowanych artefaktów => w przypadku aktualizacji danych, aktualizowany jest tylko plik konfiguracyjny albo tabela w bazie danych
  - Porządek => wiadomo gdzie szukać i gdzie nie szukać
  - Redukcja kodu źródłowego
  - Uproszczenie pracy z danymi i kodem źródłowym

# Rozdzielenie danych od logiki

- Nie należy zawierać w klasach encyjnych ani dto logiki biznesowej
- Pomocnicze wzorce:
  - MVC
  - wizytor
  - architektura trójwarstwowa (kontroler, serwis, dto, encje, dao)

# Wymuszanie enkapsulacji

Korzyści:

- Redukcja dezorientacji programisty
- Redukcja prawdopodobieństwa popełnienia błędu przez dostęp do składowej

Metody:

- Zastosowanie features wbudowanych w język, np. słowo kluczowe record (Java, C#)
- Zastosowanie zewnętrznych bibliotek np. Lobok dla Java
- Statyczna analiza kodu - pluginy, SonarQube

# Wykorzystanie obiektów reprezentujących operacje

Przykład

# Wykorzystanie interfejsów do rozluźnienia zależności

Zadanie domowe
Zaimplementuj klasę VideoChannel. Klasa powinna zawierać: nazwę kanału, listę tytułów filmów oraz metodę void AddMovieTitle(string title), która dodaje nowy tytuł do tej listy.
W momencie dodania nowego filmu, każdy subskrybent, powinien być powiadomiony w odpowiedni sposób.
Subskrybent to obiekt jednej z klas:

EmailSubscriber - w tym przypadku powiadamianie ma się odbywać poprzez wysyłanie wiadomości email
- zasymuluj to poprzez odpowiedni tekst np.
"Email: na kanale <nazwa_kanału> pojawił się nowy film pod tytułem <tytuł filmu>"

SmsSubscriber - w tym przypadku powiadamianie ma się odbywać poprzez wysyłanie wiadomości sms
- zasymuluj to poprzez odpowiedni tekst np.
"Sms: na kanale <nazwa_kanału> pojawił się nowy film pod tytułem <tytuł filmu>"