Daniel Górski
Korporacyjna Java
Wykład 5: Clean coding

Clean coding

- Mikro skala:
  - Formatowanie
  - Porządek w kodzie
  - Nazewnictwo metod, zmiennych, klas
  - Unit testy
- Makro skala:
  - SOLID
  - Uporządkowanie architektoniczne
  - Testy integracyjne

Formatowanie

- Przede wszystkim musi być ustalone
  - Brak ustalenia z dużym prawdopodobieństwem da o sobie kiedyś negatywnie znać
- Najlepiej początkowo przyjąć jakieś standardowe i modyfikować w razie uzasadnionej potrzeby
- Najczęściej modyfikowane:
  - Liczba znaków w linii
  - Ilość importów zastępowana *

Porządek w kodzie

- Co nie wygląda dobrze:
  - Zakomentowany kod
  - Niechlujne / nic nie wnoszące komentarze
  - "Wieczne" TODO
- Można robić testowe branche aby przetestować coś z marszu, nie musi być to stylowe
- Kod jaki wchodzi do mastera powinien być w miarę czysty

Nazewnictwo

- Klasy z wielkiej litery, metody i zmienne z małej
- CamelCase (stałe: SNAKE_CASE)
- Klasy: rzeczowniki
- Metody biznesowe: czasowniki (+ przymiotniki)
- Metody techniczne / zmienne: tak aby inny programista odgadł intencję

Unit testy

- Testy działania pojedynczych metod
  - Testy podstawowej ściezki
  - Testy sytuacji brzegowych
  - Testy możliwych wyjątków
  - Narzędzia do badania pokrycia kodu testami
- JUnit daje wystarczająco możliwości w tym obszarze
- TDD (Test Driven Development)
  - Zaczynamy od zdefiniowania kontraktu i napisania testów

DRY

- Don't Repeat Yourself
- Nie powtarzać tego samego kodu z niewielkimi zmianami
  - Stworzyć generyczną parametryzowaną klasę / metodę
- Nie powtarzać cyklu tych samych czynności
  - Stworzyć skrypty automatyzujące
  - Napisać testy zamiast testować "z palca"

KISS && YAGNI

- Keep It Simple Stupid
- You Aren't Gonna Need It
- Dylemat: czy pisać wielofunkcyjną / parametryzowaną metodę czy też dedykowaną pod konkretny przypadek
  - W momencie tworzenia nie zawsze możemy przewidzieć dalszy rozwój / potrzeby
  - W momencie oglądania powtórzonego kilkukrotnie kodu widzimy, że warto to zrefaktorować...

SOLID: Single responsibility

- Klasa powinna mieć jedną odpowiedzialność
- Klasy wysokopoziomowe mają wysokopoziomową odpowiedzialność
- Agregatory typu Utils należy rozdzielać gdy się rozrastają zbytnio

SOLID: Open / closed

- Klasy powinny być otwarte na rozszerzenia zamknięte na modyfikacje.
  - Klasa ma realizować w sposób kompletny swoją funkcjonalność
  - Dobrze jest oddzielić abstrakcję od konkretnej implementacji

SOLID: Liskov substitution

- Podklasy powinny być spójne działaniem z klasą rozszerzaną
- Użycie instanceof i dedykowanej logiki jest często ręcznym łataniem działania i omijaniem tej zasady
- Standardem jest instanceof w metodzie equals()
- Zdarzają się niespójności pomiędzy compareTo() [z interfejsu Comparable] a equals()

SOLID: Interface segregation

- Wiele mniejszych wyspecjalizowanych interfejsów jest lepsze niż jeden wielki
  - Uzupełnia się z Single responsibility
  - Zbyt duży interfejs: konieczność większej specjalizacji / uszczegółowienia
  - Klasyczny przykład to ConnectionUtils, TextUtils, DateUtils zamiast Utils

SOLID: Dependency Inversion

- Wysokopoziomowe moduły / klasy nie powinny zależeć od niskopoziomowych
- Dobrze jest opakować niskopoziomowe funkcjonalności w interfejs i pracować na abstrakcji
- Naturalną rzeczą jest brak możliwości skonfigurowania pewnych detali technicznych podczas wykonania na wysokim poziomie

MVC: Model View Controller

- Podstawowy wzorzec podziału na warstwy
- Warstwa widoku powinna być pozbawiona logiki
  - To prawda, że pewne rzeczy wydają się prostsze i szybsze do zrobienia bezpośrednio w Widoku bez transferu informacji przez Kontroler
  - Często powoduje to problemy w przyszłości

Testy integracyjne

- Różne metody:
  - Postawienie całego systemu i testowanie
    - Może być niemożliwe do uciągnięcia przez maszynę lokalną
    - Jest to zasobożerne
    - Daje wynik bliższy produkcyjnemu
  - Testowanie niezależnych serwisów komunikujących się przez "Mocki" innych
    - Umożliwia testowanie pojedynczego serwisu
    - Jest jasno zdefiniowany kontrakt z innymi serwisami
    - Utrzymywanie tego kontraktu może stać się z czasem bardzo kosztowne

Pracownia: co wygląda na ciężkie w utrzymaniu

- Brak klasy abstrakcyjnej figury ze współdzieloną logiką
- Skopiowana logika wczytywania w różnych miejscach
- Zahardcodowane teksty w wielu miejscach kodu

Pracownia

- Obsługa błędów ma być bardziej dokładna: wymagane jest rozróżnienie (kolejność od najważniejszych)
  - Nieznane polecenie
  - Niepoprawna liczba parametrów
  - Nieznany parametr
  - Zła wartość parametru:
    - Nie jest to liczba a powinna być
    - Niepoprawna wartość

Pracownia

- Program ma umożliwić rozwiązywanie podstawowych zadań geometrycznych dla prostokąta i rombu
- Ma wyświetlić pełne informacje o danej figurze, na podstawie zadanego wejścia

Pracownia: Prostokąt

- Charakterystyka figury
  - Długości krótszego i dłuższego boku
  - Długość przekątnej
  - Pole powierzchni
- Możliwe wejście: (dowolna dwójka)
  - Długości boków
  - Długość przekątnej
  - Pole powierzchni

Pracownia: Romb

- Charakterystyka figury
  - Długość boku
  - Długości krótszej i dłuższej przekątnej
  - Pole powierzchni
- Możliwe wejście: (dowolna dwójka)
  - Długość boku
  - Długości przekątnych
  - Pole powierzchni