UNIWERSYTET IM. ADAMA MICKIEWICZA W POZNANIU
Wydział Matematyki i Informatyki

Test-Driven Development

Ostatnim razem

Sprostowanie:

- programowanie **nie służy** komunikacji człowieka z komputerem!
- służy komunikacji z **drugim człowiekiem** (kod czytany jest średnio 7 razy po napisaniu)
- pisz tak, jakby twoi współpracownicy byli psychopatycznymi, sadystycznymi mordercami, którzy wiedzą gdzie mieszkasz

Ostatnio:

- poznaliśmy narzędzie Maven
- zaczęliśmy testy

Przypomnienie

Testy jednostkowe:

- muszą być uruchamiane w izolacji (nie zależeć od siebie)
- powinny testować zachowania kodu
- muszą pokrywać przypadki szczególne (błędne dane)
- powinny testować funkcjonalność jeden raz i...
- powinny pozwalać na modyfikację kodu (nie “betonujemy” aplikacji testami!)
- powinny być tak samo czytelne jak kod, albo nawet bardziej
- stanowią formę dokumentacji
- **piszemy przed napisaniem kodu**

Red-Green Refactor

Idea red-green refactor:

- piszemy test szczęśliwej ścieżki (ma działać dla poprawnego przebiegu programu)
  - klasy do których się odnosimy mogą nie istnieć
  - “tworzą się” w trakcie – emergent architecture
- test z założenia ma najpierw “nie przejść” (czerwony)
- dopisujemy kawałek kodu, który zapewnia “przejście” (zielony)
- dopisujemy kolejny warunek (znowu nie “przechodzi”)
- refaktoryzujemy kod, aby przechodził
- powtarzamy czynności aż do wyczerpania warunków

Red-Green Refactor

Gdy podstawowa funkcjonalność „szczęśliwej ścieżki” działa:

- dopisujemy testy na warunki brzegowe
  - niepoprawne dane wejściowe,
  - sytuacje które “na pewno nigdy nie zajdą” (założymy się?),
  - itd.
- w ten sposób otrzymujemy kawałek kodu, który na pewno działa poprawnie (wg naszych założeń, które mogą być błędne)

Red-Green Refactor

Gdy wszystkie testy i kod działają poprawnie:

- refaktoryzujemy kod, aby:
  - był czytelny,
  - klasy i metody miały jedną odpowiedzialność (Single Responsibility principle),
  - nie było zduplikowanego kodu.

Przykład

Dla naszego MagicznegoSystemuUbezpieczeń™ chcemy dodać produkt UbezpieczenieNaŻycieOgrabimyCięZMieszkania™.

Wymagania

- tylko dla osób między 35, a 65 rokiem życia;
- persona musi posiadać mieszkanie o wartości min. 300k PLN;
- gdy persona jest w związku małżeńskim kwota rośnie do 500k;
- gdy mieszkanie jest wartości z przedziału 300k do 350k i 500k do 550k dla osób zamężnych, o możliwości zawarcia musi zadecydować biuro firmy ubezpieczeniowej (backoffice).

Przykład

Czego potrzebujemy?

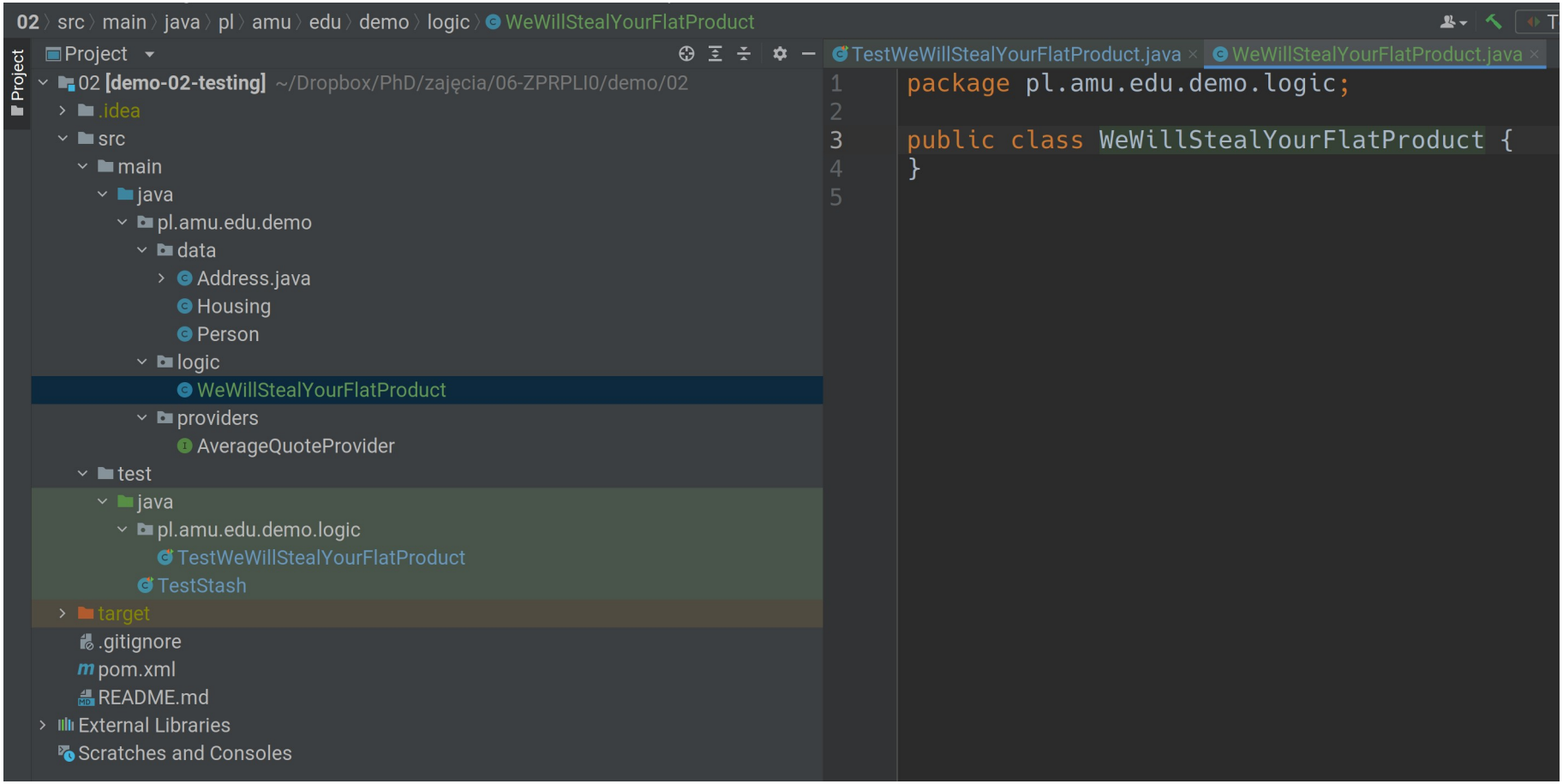
- klas przetrzymujących dane o osobach, ich mieszkaniach, ...
- jakiejś usługi, która na podstawie adresu zwróci nam średni kosz metra kwadratowego
- logiki biznesowej (klasy), która zapewni spełnienie wymagań
- nade wszystko testów ww. logiki

Zaczynamy od testu “szczęśliwej ścieżki” (“happy path”).

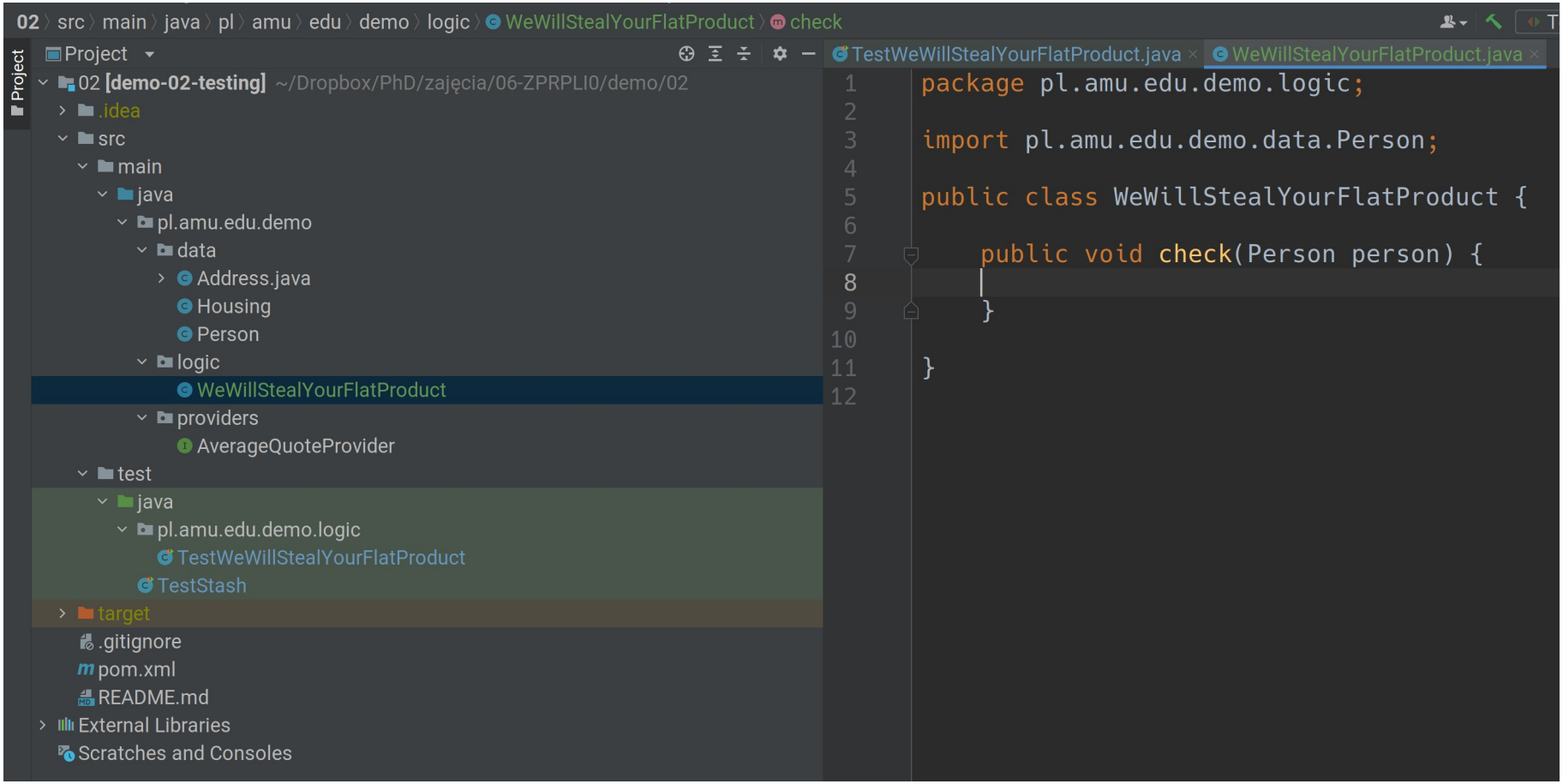
Struktura testu

```
package pl.amu.edu.demo.logic; ← konwencja (ta sama paczka)

class TestNazwaTestowanejKlasy { ← konwencja

    @Test ← adnotacja (metaprogramowanie)
    public void shouldDoSomething() {
        // co dane
        // gdy wywołujemy jakąś metodę
        // asercje (czego oczekujemy w wyniku)
    }

}
```

Przykład – logika biznesowa



Przykład – logika biznesowa



Przykład – logika biznesowa

```
12      @Test
13      public void shouldBeValid() {
14          // given
15          var person : Person = Person.builder()
16                  .displayName("Miś Uszatek")
17                  .firstName("Miś")
18                  .lastName("Uszatek")
19                  .birthDate(LocalDate.of( year: 1957,  month: 3,  dayOfMonth: 6))
20                  .housing(
21                          Housing.builder()
22                                  .address(
23                                          Address.builder()
24                                                  .withAddressLine1("Stumilowy las 5")
25                                                  .withZipCode("99-999")
26                                                  .build()
27                                  ).isApartment(false).area(36.0)
28                                  .build()
29                  ).isMarried(false)
30                  .build();
31          var logic = new WeWillStealYourFlatProduct();
32
33          // when
34          var result : void = logic.check(person);
35
```

Bookmarks

Przykład – logika biznesowa

TestWeWillStealYourFlatProduct.java × WeWillStealYourFlatProduct.java ×

```
1   package pl.amu.edu.demo.logic;
2
3   import pl.amu.edu.demo.data.Person;
4
5   public class WeWillStealYourFlatProduct {
6
7       public WeWillStealYourFlatCheckResult check(Person person) {
8
9       }
10
11  }
12
```

Create Class WeWillStealYourFlatCheckResult

Destination package: pl.amu.edu.demo.d

Target destination directory: .../src/main/java/pl/amu/edu/demo

data
providers
Press Enter to insert, Tab to replace Next Tip

OK Cancel

Przykład – logika biznesowa

TestWeWillStealYourFlatProduct.java × WeWillStealYourFlatProduct.java × WeWillStealYourFlatCheckResult.java ×

```
1  package pl.amu.edu.demo.data;
2
3  public enum WeWillStealYourFlatCheckResult {
4  }
5
```

Przykład – logika biznesowa

TestWeWillStealYourFlatProduct.java × WeWillStealYourFlatProduct.java × WeWillStealYourFlatCheckResult.java ×

```java
package pl.amu.edu.demo.data;

public enum WeWillStealYourFlatCheckResult {
    ELIGIBLE, NOT_ELIGIBLE, BACKOFFICE_DECISION
}

```

Przykład – logika biznesowa

02 › src › main › java › pl › amu › edu › demo › logic › WeWillStealYourFlatProduct › check

TestWeWillStealYourFlatProduct.java × WeWillStealYourFlatProduct.java ×

Project

```
1   package pl.amu.edu.demo.logic;
2
3   import pl.amu.edu.demo.data.Person;
4   import pl.amu.edu.demo.data.WeWillStealYourFlatCheckResult;
5
6   public class WeWillStealYourFlatProduct {
7
8       public WeWillStealYourFlatCheckResult check(Person person) {
9           return null;
10      }
11
12  }
13
```

Ale...

Pojedynczy test jakkolwiek ładny nie jest tym czego szukamy, bo:

- potrzebujemy sprawdzić wartość zwracaną dla różnych przypadków (odmienny wiek, stan cywilny, różna wartość mieszkania w zależności od okolicy, ...)
- chcemy mieć jeden kod do sprawdzania poprawności, bez powielania
- chcemy żeby kod był czysty, a dane testowe wyglądają jak...

Potrzebujemy testów parametrycznych

Struktura testu parametrycznego

```
package pl.amu.edu.demo.logic;

class TestNazwaTestowanejKlasy {

    @ParameterizedTest
    @ArgumentSource(Nazwa_klasy.class)
    public void nazwaOpisującaWymaganie(Data testData) {
        commonTest(testData);
    }

    private void commonTest(Data testData) {
        // właściwy test
    }

}
```

Trochę o parametryzacji

JUnit w wersji 5 ma (wreszcie) testy parametryczne. Idea polega na przekazaniu do metody testu danych, przy czym chcemy dla jednej metody uruchomić test tyle razy, ile danych przekazaliśmy. Potrzebujemy jakiegoś źródła i tu mogą być:

- @ValueSource – statyczne dane zdefiniowane w adnotacji
- @EnumSource – dane pochodzące z klasy enum
- @MethodSource – dane pochodzące z metody w klasie testowej
- @ArgumentSource – dane pochodzące z klasy je dostarczającej (provider)

Literatura: https://www.baeldung.com/parameterized-tests-junit-5

Czy to już wszystko?

Testy jednostkowe muszą działać w izolacji.

- nasza logika potrzebuje zewnętrznej usługi
- nie możemy po prostu jej dostarczyć (nie byłoby izolacji)
- nie mielibyśmy też wpływu na wartości przez nią zwracane

Potrzebujemy czegoś, co nazywa się mokowaniem (po angielsku mocking, stubbing, albo… test doubles [od stunt doubles]).

Na szczęście ktoś w Polsce napisał Mockito

Jeszcze o asercjach

JUnit (o zgrozo) dostarcza własne asercje. Są one nadal dość ubogie, więc ludzie stworzyli biblioteki do tworzenia asercji

- polski Hamcrest
- AssertJ Fluent Assertions

Kiedyś AssertJ po prostu opakowywał Hamcresta… Polecam (i sam używam) AssertJ.

No to przykład całości:
https://git.wmi.amu.edu.pl/pdyda/06-ZPRPLI0/src/branch/master/demo/02

Debugowanie

Związek testów z debugowaniem jest taki, że dają one nam łatwy punkt wejścia do programu, do sprawdzenia fragmentu w izolacji. Istotne koncepcje:

- breakpoint (punkt zatrzymania w wykonywanym programie)
  - może być warunkowy (conditional breakpoint)
- step over (pominięcie linii kodu)s
- step in(to) (wejście do wywoływanej metody)
- step out (wyjście z aktualnie wykonywanego fragmentu kodu na poziom wyżej)
- evaluate expression – ewaluacja wpisanego ręcznie kodu

Lteratura: https://www.jetbrains.com/help/idea/debugging-code.html

Ćwiczenia

Ćwiczenia do prezentacji:

https://link.wmi.amu.edu.pl/j1vtPX

Dziękuję za korporację
(thank you for your corporation)