Hleb Shypula

Praca inżynierska – Plan

WfiIS 2021

Opis pracy:

Celem pracy jest przedstawienie zasadniczych cech metodyki Test Driven Development na przykładzie procesu tworzenia aplikacji internetowej wykorzystującej Java Spring.

Opis aplikacji:

Aplikacja powinna realizować sprzedaż internetową dla firmy farmaceutycznej.

Wymagania funkcjonalne:

- Logowanie użytkowników – strona umożliwia każdemu użytkownikowi systemu powprowadzeniu odpowiednich danych (loginu i hasła) zalogowaniesię do niego. Autoryzacja odbywa się na podstawie bazy danych użytkowników oraz wspiera protokół SSL.
- Rejestracja użytkowników – strona umożliwia każdemu użytkownikowi rejestrację powprowadzeniu odpowiednich danych (dane osobowe, login hasło, adres e-mail) do formularza rejestracyjnego.
- Wyświetlanie katalogu produktów – strona powinna umożliwiać wyświetlanie całego katalogu produktów opartego o bazę danych. Wyświetlone powinny zostać wszystkie produkty wraz zich opisem itd. Produkty powinny zostać czytelnie podzielone na podkategorie.
- Wyszukiwanie produktów – strona powinna umożliwiać użytkownikowi wyszukiwanie konkretnego produktu podając jego nazwę. Ponadto możliwe powinno być wskazanie szczegółowych wymagań odnośnie danego produktu (np. podkategoria) oraz wyświetlenie wszystkich produktów spełniających podane kryteria.
- Koszyk produktów – strona powinna umożliwiać gromadzenie wybranych przez użytkownika produktów oraz ich ilości, przeliczając na bieżąco sumę wszystkich produktów w koszyku. Zatwierdznia mogą dokonać jak użytkownicy zarejestrowane, tak i niezarejestrowane podawając odpowiedzie dane dla jednokrotnego zakupu.
- Dokonywanie zamówienia oraz jego anulowanie –
  złożenie zamówienia powinno się odbywać po określeniu:
  • dodaniu produktów do koszyka produktów oraz zatwierdzenie wyboru
  • sprawdzenie przez system kompatybilności wybranych produktów i poinformowanie użytkownika o ewentualnych błędach
  • wybrania sposobu zapłaty
  • anulowanie jest możliwe do momentu zmiany statusu zamówienia z „Przyjęte do realizacji” na „Realizowane”.
- Wybór płatności –
  strona powinna umożliwiać użytkownikom określenie sposobu dokonania płatności za dokonane zakupy spośród:
  • gotówka w siedzibie firmy przy odbiorze
  • karta kredytowa/płatnicza przy odbiorze osobistym oraz na stronie

- Uprawnienia administratora oraz obsługi serwisu –
  strona powinna umożliwiać administratorowi:
  • dodawanie, usuwanie oraz edycję użytkowników
  • zarządzanie zamówieniami (edycja statusu zamówienia itd)
- Uprawnienia użytkowników zarejestrowanych –
  strona powinna umożliwiać użytkownikom zarejestrowanym:
  • zalogowanie się do sklepu
  • przeglądanie publicznego katalogu produktów
  • dodanie produktów do koszyka
  • dokonanie płatności po zwerefikowaniu zamówienia przez serwis
  • zmianę własnych danych osobowych i dostępowych
  • kontakt z obsługą sklepu
- Uprawnienia użytkowników niezarejestrowanych –
  strona powinna umożliwiać użytkownikom nierejestrowanym:
  • rejestrację (podanie pełnych danych osobowych, loginu, hasła oraz adresu e-mail)
  • przeglądanie publicznego katalogu produktów
  • wyszukiwanie produktów według wybranych kryteriów
  • dodanie wybranych produktów do koszyka z możliwościa złożenia zamówienia bez zalogowania podając dane osobowe
  • kontakt zobsługą sklepu
- Kontakt –
  strona powinna umożliwiać użytkownikom uzyskanie informacji o kontakcie z firmą poprzez oraz sklepem:
  • wyświetlenie mapki dojazdowej
  • adres, telefon, e-mail itd
  • formularz kontaktowy (wysłanie wiadomości email)

Front-end:

ReactJS (TS).

Back-end:

Java Spring.

DB:

Hibernate.

MSSQL Server (Najprawdopodobniej).

TDD:

JUnit, (Selenium dla testowania w późnych etapach).

TDD – Napisanie aplikacji oparte na testy. Staramy się znaleźć najlepsze rozwiązanie na przejście testu, a nie napisać test opierając się na rozwiązanie modelu.

1. Dodaję automatyczny test (lub modyfikuję istniejący test).
2. Przeprowadzam wszystkie testy i sprawdzam, czy nowy test się nie udał.
3. Implementuję funkcjonalność – tak aby przeszedł test.
4. Przeprowadzam testy i sprawdzam czy się powiodły – jeśli tak, to kod spełnia wymagania.
5. Refaktoryzuję kod – dzięki czemu będzie „czystszy". Poprzez ponowne przeprowadzanie testów sprawdzam, że wprowadzone zmiany nie niszczą istniejącej funkcjonalności.
6. Powtarzam całość aby rozszerzać funkcjonalność.

- Jak zakłada TDD, podejmujemy decyzje dotyczące modelu podczas napisania testu. Czyli wymyślamy potrzebne klasy itd, wtedy test wymusi nas te klasy zaimplementować.

- Gold rule – nie implementujemy żadnego kodu, którego test nie wymaga.

- Dobra zasada – „assert first". Czyli najpierw implementujemy odwołanie do jakiegoś obiektu przy asercji, a dopiero potem mamy napisać kod, żeby to odwołanie zadziałało.

```
@Test
public void donateMovie(){
    assertTrue(library.getCatalogue().contains(movie));
}
```

  Wtedy widzimy co musi zawierać nasz model nie aby przejść test, tylko na razie przeprowadzić test.

```
@Test
public void donateMovie(){
    Library library = new Library();
    Movie movie = new Movie();
    library.donate(movie);
    assertTrue(library.getCatalogue().contains(movie));
}
```

  I dopiero potem tworzymy potrzebne nam klasy.

- Dobra zasada – „see the test fail". Czyli sprawdzamy, że test się nie udał przy błędnych danych na wejściu. To jest ważne, ponieważ chcemy być pewni, że jeżeli w przyszłości coś pozmieniamy i tym zepsujemy tą funkcjonalność, to test to wykryje.

- Dobra zasada – „one reason to fail". Test powinien przy niepowodzeniu mieć wyłącznie jedną przyczynę.

- Podczas napisania testów uważamy na powtarzający się kod. Ponieważ jak przez jakiś czas będziemy mieli bardzo wiele testów i coś się zmieni w konstrukcji, która występuje w większości tych testów, to będziemy musieli zmieniać kod we wszystkich tych miejscach. Czyli trzeba uważać na duplikowanie zależności pomiędzy kodem testującym i kodem klas testowanych.

Jednym z dobrych rozwiązań jest tworzenie factory metod. Jeżeli coś się doda w klasie testowanej, to będziemy musieli zmienić tylko ten factory metod.

Kolejne dobre rozwiązanie – użycie junitparams:

```
@RunWith(JUnitParamsRunner.class)
public class RoverTest {

    @Test
    @Parameters({
            "N,E",
            "E,S",
            "S,W",
            "W,N"
    })
    public void turnsRightClockwise(String startsFacing, String endsFacing){
        Rover rover = new Rover(startsFacing);
        rover.go( instructions: "R");
        assertEquals(endsFacing, rover.getFacing());
    }
}
```

Uważamy na zredukowanie duplikatów, żeby nie doszło do nieczytelnego kodu testów (podajemy w nazwach metod to, co test zawiera w środku, żeby nie było potrzeby patrzeć na kod – wszystko da się odczytać z nazwy).

- „The rule of three" – Chcemy mieć 3 powtarzających się występowania jakiejś logiki w testach zanim zaczniemy refaktoryzować duplikaty. 3 a nie 2, ponieważ wtedy będziemy już ogarniać pattern tych metod i będziemy w stanie poprawnie zdefiniować metodę na zredukowanie tych powtórzeń. I jak jest 3, to wzrasta szansa na kolejne występowanie.

- Podejście „Inside-Out" – zaczynamy od „test-driven" wewnętrznych części naszej logiki zanim połączymy to w całość jako końcowe rozwiązanie.
  Zalety – jeżeli test nie przejdzie, to dokładnie wiemy GDZIE to się stało.
  Wady – Słaba enkapsulacja; Większe ryzyko, że te małe kawałki wewnętrznej struktury po połączeniu się nie zgadzą; Jeżeli logika się zmieni (a zatem i kod), to mamy wiele testów do refaktoryzacji.

- Podejście „Outside-In" – zaczynamy od zewnętrznych części aplikacji, od całych funkcjonalności, a nie od małych wewnętrznych kawałków.
  Wada – jeżeli test nie przejdzie, to nie będziemy wiedzieć gdzie jest błąd, ponieważ test nie widzi naszych wewnętrznych metod. Test tylko potrafi nam powiedzieć, że błąd jest „gdzieś tam".
  Zalety – Gwarantowane jest, że małe części logiki są połączone zgodnie z wymaganiami; Lepsza enkapsulacja.

- Łączymy te 2 podejścia, wtedy zapewniamy, że implementacja jest zgodna z wymaganiami (outside-in), a jeżeli test nie przejdzie, to dokładnie wiemy gdzie jest problem. Czyli chcemy mieć podwójne testowanie.

- Testowanie Spring Service – mamy testowanie jednostkowe przy pomocy Mockito. Oraz mamy testowanie integracyjne z dostępem do bazy danych.

- Testowanie Spring Controller – mamy testowanie jednostkowe przy pomocy Mockito. Oraz mamy testowanie integracyjne z rzeczywistym wysłaniem zapytań.

- Testowanie Spring Data Access – tylko testy integracyjne. Nie ma potrzeby w testach jednostkowych, ponieważ większość funkcjonalności jest wbudowana i nie musimy tego sprawdzać.

- Łączymy testy integracyjne w feature test suite.

- Tworzymy CI test suites.

- Jak dostaliśmy zielone światło (test przeszedł), to idziemy głębiej do naszego kodu. Czyli refaktoryzujemy go tak, aby dostać piękny model. Po tym puszczamy testy jeszcze raz oczywiście.

- Nie piszemy testy opierając się o znany wcześniej model. Czyli jak wymyślimy jakąś potrzebną w projekcie encję, to będziemy pisać testy na poszczególne metody, które klasa dla tej encji ma implementować. I to jest błędne podejście – mamy tu do czynienia z „Design-Driven Testing", a nie „Test-Driven Design".

- Pattern – „test list". Przechodzimy przez wymagania (przypadki użycia) i przemyślamy je od strony przypadków testowych (scenariuszy, które nasz kod ma implementować). Czyli chcemy mieć tzw. „meaningful tests", pozwolmy testom wymyślić potrzebny nam model w oparciu o przypadki testowe.

- Zaczynamy testowanie od hardkorowych wartości, czyli po prostu zwracamy żądaną wartość, żeby zobaczyć jak nam test działa. I rozszerzamy się, pisząc kolejne testy i implementując logikę. Ale...

- Jeżeli logika jest oczywista (np. quantity-- czy inne proste operacje), to nie traćmy czas na testowanie takich przypadków. Po prostu implementujemy kod. Jeżeli uważamy, że logika nie jest aż tak trywialna, to wtedy chcemy triangulować ten przypadek użycia.

Sprint 1 (planowane sprint review – 22-23.07.2021):

- Czytanie książki i notowanie najważniejszych zasad – Kent Beck, TDD
- Kurs ReactJS (TS)
- Przejście po kursach ze strony https://www.linkedin.com/learning/paths/become-a-spring-developer
- Stworzenie struktury aplikacji
- Ewentualnie implementacja pierwszych kroków

Pytania organizacyjne:

1. Ogólny system napisania pracy inżynierskiej.
2. Czy wolno korzystać z własnego serwera w późnych etapach pracy? Czy to ma być jakaś darmowa platforma zewnętrzna?
3. Interfejs ma być w języku polskim? (Przykładowa baza towarów, którą mogę uzyskać, jest w języku rosyjskim)