Temat:
Program do gromadzenia informacji o obserwacjach przyrodniczych

Przedmiot „Programowanie w języku Java”
Studia niestacjonarne
Skład zespołu:
Izabela Tarasin
Robert Pierog
Jakub Fik
Michał Piwowarczyk

Opis:
Aplikacja desktopowa z GUI w JavaFX uwzględniająca lokalizację i dokumentację fotograficzną oraz różne kategorie obiektów zapisane w bazie SQLite.
Aplikacja posiada 5 ekranów:
- menu - umożliwia dostęp do pozostałych części programu
- wyświetlenie obserwacji - wyświetla listę obserwacji zawartych w bazie
- dodanie nowej obserwacji - dodanie i zapisanie obserwacji, która zawiera nazwę, opis, lokalizację, kategorię oraz zdjęcie.
- dodanie kategorii - umożliwia dodanie i zapisanie w bazie nowej kategorii
- dodanie lokalizacji - umożliwia dodanie i zapisanie w bazie nowej lokalizacji

Pomiędzy ekranami została zaimplementowana nawigacja realizowana przy pomocy przycisku „wstecz”, który po naciśnięciu przenosi użytkownika do poprzedniego ekranu.
Zdjęcie jest możliwe do wyboru poprzez zaimplementowanie file choosera a obsługiwane formaty plików to .jpg .png.

1. Projekt jest Mavenowy i poprawnie buduje się z lini komend konsoli systemu operacyjnego.

Komendy do uruchomienia:

mvn package - zbudowanie projektu, uruchomienie testów, utworzenie pliku jar
java -jar /…/…(jako ścieżka do pliku jar)
mvn test - uruchomienie samych testów
mvn javafx:run - uruchomienie aplikacji po zbudowaniu kodu źródłowego (bez testów)

2. Plik jar jest możliwy do uruchomienia z poziomu konsoli systemu operacyjnego przy pomocy komendy:
   j**ava -jar /…/…**(jako ścieżka do pliku jar)

3. Wykonano **10 przykładowych testów jednostkowych**.

Do wykonania testów potrzebne było stworzenie sztucznej bazy danych **DatabaseMock** odpowiedzialnej za dostarczanie danych podczas testów jak i sprawdzania czy dane w odpowiednim formacie zostałyby przekazane do prawdziwej bazy.
Testy nie testują warstwy danych.

Testy dotyczą:
- NatureObservationDBOBuilderTest (1 test)- sprawdzenie, czy obiekt utworzony przy pomocy buildera zawiera poprawne dane (NatureObservationDBOBuilder)

- CategoryDBOToCategoryMapperTests, LocalisationDBOToLocalisationMapperTests, NatureObservationDBOToNatureObservationTests (7 testów) - sprawdzenie, czy konwersja obiektu między warstwami aplikacji (warstwa wizualna i warstwa danych) przebiega poprawnie

- CategoryFacadesTests oraz LocalisationFacadesTests (2 testy) - sprawdzenie, czy działanie fasad jest prawidłowe

4. Wszystkie 10 testów jednostkowych mają status PASSED w trakcie budowania projektu.
   Użyta komenda:
   mvn package lub **mvn test**

5. Projekt jest umieszczony pod kontrolą wersji Git.

Platforma: https://bitbucket.org

6. Opis struktury gałęzi repozytorium:

master - finalna wersja projektu oddana do oceny - zabezpieczona w trakcie projektu (nowe wersje mergowane tylko przez jedną osobę)
develop - główna gałąź integracyjna - zabezpieczona w trakcie projektu (pozwala na merge zawiera poprawne dane przez jedną osobę)
feature/… - tutaj tworzone były gałęzie deweloperskie, które dodawały nowe funkcjonalności do projektu. Każda osoba pracowała na osobnych branchach.

bugfix/… - tutaj tworzone były gałęzie dewelopwerskie, które naprawiały błędy w programie/funkcjonalnościach. Każda osoba pracowała na osobnych branchach.

W trakcie budowy projektu występowały konflikty przy merge do gałęzi develop, które z sukcesem były rozwiązywane przez członków zespołu.
Projekt posiada odpowiednio długą historię wersji sięgającą około miesiąca z wieloma commitami oraz mergami.

7. Skorzystano z interfejsów/klas abstrakcyjnych tworząc **IDatabase** oraz **Facade** podczas prac nad projektem.

IDatabase - interface posiadający deklaracje metod użytych w operacjach na bazie danych (tj. Database). Użyty również podczas tworzenia sztucznej bazy danych (DatabaseMock) potrzebnej do testów jednostkowych.
Facade - klasa abstrakcyjna, która stanowi uproszczony mechanizm odczytu i zapisu z i do bazy danych.

8. Wiele razy w projekcie korzystano z polimorfizmu, który wykorzystywał własne klasy abstrakcyjne i interfejs.

Najlepszym przykładem są fasady, które korzystając z generycznej klasy abstrakcyjnej Facade wykorzystują polimorfizm poprzez nadpisywanie metod z konkretnymi typami. Również klasy dziedziczące po ViewController (np. MenuViewController) używają polimorfizmu nadpisując metodę LoadView

9. Wykorzystano **4 wzorce projektowe**:

Facade Pattern - użyty w celu łatwego i ograniczonego dostępu do zasobów w bazie danych. Uzyskaliśmy prosty interface do odczytu i zapisu stworzonych w aplikacji danych.
Aplikacja rozdziela dane na których operuje użytkownik od danych, które trafiają do bazy danych (DBO = Data Base Object). Aby uniknąć trudności i powielania i rozrastania kodu wydzielono część logiki do fasad (Category, Localisation, NatureObservation).
Została czyta ponieważ każda z fasad operuje w pewien sposób na bazie danych. Interfejs nie pozwala na przechowywanie pól wiec jedyną opcją było zastosować jedną abstrakcyjną klasę bazową dla wszystkich z fasad. Jest to klasa abstrakcyjna a nie zwykłą ponieważ instancja fasady sama w sobie nie ma sensu. Jedynie klasy ją poszerzające i operujące na konkretnych modelach mają sens.

Factory Pattern - użyty w celu tworzenia obiektów ViewControllerów z pominięciem ich szczegółowego typu (zwracane jako obiekt typu ViewController, co jest klasą bazową dla każdego ViewControllera). Dodatkowo fabryka zawiera wszystkie zależności jakie są wymagane w poszczególnych ViewControllerach.

Dzięki temu łatwiej jest rozszerzać kod konstruujący ViewControllery bez konieczności ingerencji w resztę kodu. Przykładowo jeżeli jeden w ViewControllerów wymaga większej ilości zależności to wystarczy dodać zależność w ViewControllerFactory podczas jego tworzenia.
Dodatkowo wszystkie zależności jakie trafiają do ViewControllerów są tworzone tylko raz i przekazywane podczas budowania poszczególnych ViewControllerów.

Adapter Patter - użyty kilkukrotnie:
- podczas implementacji wzorca mvc. Klasy i obiekty odpowiadające za warstwę wizualną z użytego frameworka JavaFX zostały opakowane we własne obiekty typu View i ViewController.
Dzięki temu w łatwy sposób można pośredniczyć pomiędzy naszym kodem a klasą pochodzącą z zewnętrznej biblioteki.
- podczas transportu wczytanego obrazka między różnymi częściami kodu.
Jest to spowodowane, że nie udało się w łatwy sposób wyciągnąć danych z klasy Image (JavaFX), które później zostałyby zapisane do bazy jako typ BLOB (Binary Large Object). W jednym części kodu jest potrzeba dostępu do pliku obrazka wczytanego z dysku a w drugiej części kodu potrzebne jest wyświetlenie obrazka zawartego w klasie Image z frameworku JavaFX.
Dzięki takiemu zastosowaniu w zależności od miejsca gdzie trafia obiekt mamy dostęp do obrazka np. w celu wyświetlenia go oraz do pliku (file) w celu konwertowania go i zapisania w bazie danych.

Builder Pattern - użyty podczas budowania obiektu NatureObservationDBO, który trafia do bazy danych. Pozwala on na przekazywanie opcjonalnego parametru zdjęcia obserwacji jeżeli ono nie istnieje (NatureObservationToNatureObservationDBOMapper). Wzorzec ten pozwala konstruować obiekty krok po kroku w miarę jakie ich pola są potrzebne.

Chain of responsibility (łańcuch zobowiązań) - użyty w celu ustalenia porządku obsługi żądania np. dodania obiektu do bazy danych.
Na przykład aby dodać nową obserwację użytkownik tworzy obiekt, który zawiera dane pobrane z interfejsu użytkownika wraz ze zdjęciem (ObservationViewController) a następnie obiekt ten trafia do fasady (NatureObservationFacade) w której następuje zamiana obiektu za pomocą odpowiedniego mappera (NatureObservationToNatureObservationDBOMapper) przez co otrzymujemy obiekt z danymi gotowymi do zapisu:
obrazek będący plikiem zostaje zamieniony na tablice bajtów a relacje z obiektami Category i Localisation zostają zamienione na ich ID.
Następnie obiekt ten trafia do bazy (NatureObservationDBO)
Analogicznie podczas żądania pobrania danych z bazy i wyświetlenia użytkownikowi.

10. W celu usprawnienia tworzenia aplikacji stworzono konieczne do przedstawienia interfejsy i klasy abstrakcyjne. Dokonano podziału na warstwy architektoniczne: widoku, biznesowa i danych. Kod podzielono na odpowiednie grupy paczek realizujące te same funkcjonalności.

11. Zastosowano wzorzec architektoniczny **MVC** wykorzystujący JavaFX jako GUI. Aplikacja została podzielona na poszczególne ekrany widoczne przez użytkownika, zwane View, zawierające kontrolki z JavaFX. Każdy View zarządzany jest przez swój własny ViewController, który stanowi warstwę pomiędzy widokiem a modelem. Model należy traktować jako dane wraz z logiką biznesową.
    Przykładowo użytkownik klikając w przycisk „zapisz nową kategorię” uruchamia metodę obsługująca zdarzenie kliknięcia przypietą w ViewControllerze, która z kolei komunikuje się bezpośrednio z fasadą będącą modelem we wzorcu MVC.