***Wojskowa Akademia Techniczna***

***im. Jarosława Dąbrowskiego***



**Wydział Cybernetyki, kierunek informatyka - inżynieria systemów**

Sprawozdanie z realizacji zadania przedmiotu:

***Event-driven programming***

Temat projektu:

***Aplikacja do monitorowania pogody***

**Opracował:** Radosław Relidzyński, **Grupa:** WCY20IJ1S1

**Prowadzący:** mgr inż. Michał Sobolewski

Spis treści

[Wstęp 3](#_Toc138415355)

[1. Wykorzystane technologie 3](#_Toc138415356)

[2. Prezentacja aplikacji 4](#_Toc138415357)

[3. Schemat aplikacji 7](#_Toc138415358)

[4. Wykorzystane biblioteki oraz sposób ich zaimportowania 8](#_Toc138415359)

[5. Pomoce dydaktyczne w zakresie implementacji 9](#_Toc138415360)

[6. Odniesienie się realizacji projektu do modelu SOLID 9](#_Toc138415361)

[7. Odniesienie się do realizacji wymagań 10](#_Toc138415362)

[7.1. Technologia Java SE 10](#_Toc138415363)

[7.2. Aplikacja GUI (Swing, JavaFX, SWT) 10](#_Toc138415364)

[7.3. Wielowątkowa 10](#_Toc138415365)

[7.4. Programowanie asynchroniczne 12](#_Toc138415366)

[7.5. Zdarzenia (własne zdarzenia) 12](#_Toc138415367)

[7.6. Wykorzystanie danych pobranych z dwóch zewnętrznych usług webowych 13](#_Toc138415368)

[7.7. Bazy danych (JDBC, JPA) 16](#_Toc138415369)

[7.8. Config properties 16](#_Toc138415370)

[7.9. Własne 2 komponenty graficzne 17](#_Toc138415371)

[7.10. Dodatkowo punktowane wykorzystanie reflection API 17](#_Toc138415372)

[7.11. Wykorzystanie bibliotek Apache Commons oraz Magistrali zdarzeniowych 17](#_Toc138415373)

[7.12. Zastosowanie wzorców programowania obiektowego i zdarzeniowego 17](#_Toc138415374)

[7.13. Zastosowanie wzorców projektowych z serii strukturalnych i behawioralnych 18](#_Toc138415375)

[7.14. Wymaga się całkowitego autorstwa kodów źródłowych 18](#_Toc138415376)

[8. Podsumowanie 18](#_Toc138415377)

# Wstęp

Weather Monitor jest to aplikacja desktopowa służąca do monitorowania pogody w krajach Europy centralno-wschodniej. Przy pomocy niej można pobierać aktualną temperaturę w krajach oraz inne informacje pogodowe takie jak poziom zachmurzenia, opady czy ciśnienie atmosferyczne.

Aplikacja wykonana jest w języku java z wykorzystaniem technologii JavaFX do tworzenia GUI aplikacji. W ramach bazy danych wykorzystana została baza posgresql oraz połączenie z nią przy pomocy technologii hibernate. Do pobierania informacji o pogodzie wykorzystano API openweathermap.

# Wykorzystane technologie

* 1. **Java SE** - zestaw narzędzi i bibliotek programistycznych służących do tworzenia aplikacji w języku Java.
  2. **JavaFX** – biblioteka do tworzenia interfejsów graficznych (GUI) w języku Java, umożliwiająca tworzenie aplikacji desktopowych.
  3. **PostgreSQL** - relacyjna baza danych, która obsługuje rozszerzenia SQL.
  4. **Hibernate** - framework do mapowania obiektowo-relacyjnego (ORM) dla języka Java, który ułatwia przechowywanie i pobieranie danych z bazy danych.
  5. **Jakarta** - zestaw specyfikacji i bibliotek służących do tworzenia zaawansowanych aplikacji w języku Java.
  6. **JSON** - format wymiany danych, który jest popularny w aplikacjach internetowych do przesyłania danych w sposób czytelny dla człowieka i łatwy do przetwarzania przez komputery.
  7. **Lombok** - generowanie kodu do standardowych operacji, takich jak gettery, settery i konstruktory, co skraca czas i wysiłek programisty.
  8. **Apache Commons** - zestaw bibliotek, które zawierają wiele użytecznych narzędzi i komponentów ułatwiających programowanie.
  9. **Guava** - biblioteka Java stworzona przez Google, która dostarcza wiele użytecznych narzędzi i funkcji, takich jak kolekcje, operacje na strumieniach, obsługa wyjątków i wiele innych.

# Prezentacja aplikacji

* 1. Wygląd początkowy: Obraz zawierający tekst, mapa, zrzut ekranu, diagram

     Opis wygenerowany automatycznie
  2. Start monitorowania Polski Obraz zawierający tekst, zrzut ekranu, mapa, diagram

     Opis wygenerowany automatycznie
  3. Włączenie rejestru Obraz zawierający tekst, mapa, zrzut ekranu, diagram

     Opis wygenerowany automatycznie
  4. Zatrzymanie rejestru Obraz zawierający tekst, zrzut ekranu, mapa, diagram

     Opis wygenerowany automatycznie
  5. Wyczyszczenie rejestru Obraz zawierający tekst, zrzut ekranu, mapa, diagram

     Opis wygenerowany automatycznie
  6. Rozpoczęcie monitorowania Niemiec i Czech Obraz zawierający tekst, zrzut ekranu, mapa, diagram

     Opis wygenerowany automatycznie
  7. Ponowne włączenie rejestru (w tle cały czas dodawały się rejestry dla Polski)Obraz zawierający tekst, mapa, zrzut ekranu, atlas

     Opis wygenerowany automatycznie

# Schemat aplikacji

Obraz zawierający tekst, diagram, zrzut ekranu, design

Opis wygenerowany automatycznie

# Wykorzystane biblioteki oraz sposób ich zaimportowania

* 1. Sposób dodania bibliotek: biblioteki zostały dodane za pomocą pliku pom.xml jako elementy „dependencies”. Każda biblioteka to osobny element „dependency”, czyli zaimportowany jako zależność dla Mavena.
  2. Lista bibliotek:

<dependencies>  
 <dependency>  
 <groupId>org.openjfx</groupId>  
 <artifactId>javafx-controls</artifactId>  
 <version>19.0.2</version>  
 </dependency>  
 <dependency>  
 <groupId>org.openjfx</groupId>  
 <artifactId>javafx-fxml</artifactId>  
 <version>19.0.2</version>  
 </dependency>  
 <dependency>  
 <groupId>org.controlsfx</groupId>  
 <artifactId>controlsfx</artifactId>  
 <version>11.1.2</version>  
 </dependency>  
 <dependency>  
 <groupId>org.kordamp.bootstrapfx</groupId>  
 <artifactId>bootstrapfx-core</artifactId>  
 <version>0.4.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-api</artifactId>  
 <version>${junit.version}</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-engine</artifactId>  
 <version>${junit.version}</version>  
 <scope>test</scope>  
 </dependency>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.28</version>  
 </dependency>  
 <dependency>  
 <groupId>com.google.guava</groupId>  
 <artifactId>guava</artifactId>  
 <version>31.0.1-jre</version>  
 </dependency>  
 <dependency>  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-core</artifactId>  
 <version>6.2.3.Final</version>  
 </dependency>  
 <dependency>  
 <groupId>org.postgresql</groupId>  
 <artifactId>postgresql</artifactId>  
 <version>42.6.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.json</groupId>  
 <artifactId>json</artifactId>  
 <version>20220320</version>  
 </dependency>  
 <dependency>  
 <groupId>com.google.code.gson</groupId>  
 <artifactId>gson</artifactId>  
 <version>2.8.9</version>  
 </dependency>  
 <dependency>  
 <groupId>name.finsterwalder</groupId>  
 <artifactId>fileutils</artifactId>  
 <version>1.1</version>  
 </dependency>  
 <dependency>  
 <groupId>commons-io</groupId>  
 <artifactId>commons-io</artifactId>  
 <version>2.11.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.commons</groupId>  
 <artifactId>commons-lang3</artifactId>  
 <version>3.12.0</version>  
 </dependency>  
</dependencies>

# Pomoce dydaktyczne w zakresie implementacji

* 1. <https://refactoring.guru/pl/design-patterns>
  2. <https://home.openweathermap.org/>
  3. <https://api.elephantsql.com/console/3bfa7042-df2a-42b8-ad15-bd6cad0f8954/details>
  4. <https://www.samouczekprogramisty.pl/solid-czyli-dobre-praktyki-w-programowaniu-obiektowym/>
  5. <https://www.baeldung.com/guava-eventbus>
  6. <https://www.baeldung.com/guava-eventbus>
  7. <https://www.tutorialspoint.com/h2_database/h2_database_jdbc_connection.htm>
  8. <https://projectlombok.org/setup/maven>
  9. <https://projectlombok.org/features/Data>
  10. <https://hibernate.org/orm/documentation/getting-started/>
  11. <http://adriankurek.pl/2019/08/22/jpa-i-hibernate-tworzenie-projektu-maven/>
  12. <https://mvnrepository.com/artifact/org.postgresql/postgresql/42.6.0>
  13. <https://stackoverflow.com/questions/51478675/error-javafx-runtime-components-are-missing-and-are-required-to-run-this-appli>

# Odniesienie się realizacji projektu do modelu SOLID

* 1. S – samodzielny

Projekt spełnia zasadę samodzielności, ponieważ poszczególne komponenty systemu są niezależne od siebie i mogą być łatwo wymieniane lub rozszerzane bez wpływu na całość. Na przykład, można łatwo zmienić źródło danych o pogodzie lub miejsce ich przechowywania bez konieczności wprowadzania zmian w całym systemie, jedynie w jego konkretnych elementach.

* 1. O – otwarty

Projekt spełnia zasadę otwartości, ponieważ jest zaprojektowany w taki sposób, aby był łatwo rozszerzalny o nowe funkcjonalności. Na przykład, można łatwo zaimplementować nowe pola do pobierania jeszcze bardziej szczegółowych informacji o pogodzie lub stworzyć mechanizm pobierający pogodę z więcej niż jednego źródła.

* 1. L – Liskov Barbara

Zasada podstawienia Liskov mówi o tym, że można używać klas dziedziczących po klasach bazowych. W ramach projektu nie zachodzi dziedziczenie klas, stąd nie ma elementów kodu do wskazania zastosowania zasady Liskov.

* 1. I – interfejsy

W projekcie zasada segregacji interfejsów oznacza, że interfejsy powinny być dostosowane do konkretnych potrzeb komponentów. W ramach projektu każdy kraj posiada swój osoby zestaw obiektów (przycisk, pole tekstowe, pole na obraz), który reprezentuje unikalne informacje o pogodzie tego kraju.

* 1. D – odwrócenie zależności

W projekcie zasada odwrócenia zależności mówi o tym, że moduły powinny polegać na abstrakcjach, a nie na konkretnej implementacji. Główna magistrala używa interfejsu „EventBus”, a klasy nasłuchujące magistrali posiadają interfejs „Subscribe”.

# Odniesienie się do realizacji wymagań

### Technologia Java SE

Cały projekt został zrealizowany w technologii Java SE, aplikacja działa na platformie Java Standard Edition, wykorzystując funkcje i biblioteki Java SE.

### Aplikacja GUI (Swing, JavaFX, SWT)

Aplikacja GUI została wykonana przy użyciu technologii JavaFX, odpowiadająca za wygląd aplikacji i komunikację programu z człowiekiem.

### Wielowątkowa

Aplikacja jest wielowątkowa, w ramach swojego działania powołuje i zatrzymuje wątki odpowiedzialne za monitorowanie pogody lub rejestru.

Powoływanie wątku do monitorowania pogody:

private void startCountryWeatherThread(CountryRecordToggleEvent event) {  
  
 // Create and start thread  
 CountryWeatherThread countryWeatherThread = new CountryWeatherThread(centralEventBus, event.country(), generalPeriod);  
 countryWeatherThread.start();  
  
 // Add countryWeatherThread to list of workingCountryWeatherThreads  
 *workingCountryWeatherThreads*.add(countryWeatherThread);  
}

Implementacja wątku odpowiedzialnego za monitorowanie pogody danego kraju:

package com.example.weather\_monitor;  
  
import com.example.weather\_monitor.data.Country;  
import com.example.weather\_monitor.db.WeatherRecord;  
import com.example.weather\_monitor.event.WeatherRecordEvent;  
  
/\* Thread class that will gather the weather update from api and parse to the application \*/  
public class CountryWeatherThread extends Thread {  
 private final CentralEventBus centralEventBus;  
 private final Country country;  
 private final int period;  
 private volatile boolean running = true;  
  
 public CountryWeatherThread(CentralEventBus centralEventBus, Country country, int period) {  
 this.centralEventBus = centralEventBus;  
 this.country = country;  
 this.period = period;  
 }  
  
 public WeatherRecordEvent generateEvent() {  
 WeatherRecord weatherRecord = APIWeatherManager.*getWeatherData*(country);  
 return new WeatherRecordEvent(weatherRecord);  
 }  
  
 public void publishEvent() {  
 centralEventBus.publishEventFromListener(generateEvent());  
 }  
  
 @Override  
 public void run() {  
 while (running) {  
 publishEvent();  
 try {  
 Thread.*sleep*(period);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 public void stopThread() {  
 running = false;  
 }  
  
 public Country getCountry() {  
 return country;  
 }  
}

### Programowanie asynchroniczne

Wątki w aplikacji działają asynchronicznie. Każdy z nich powoływany jest niezależnie i działają równolegle.

### Zdarzenia (własne zdarzenia)

W projekcie zastosowano własne zdarzenia, przedstawione poniżej:

package com.example.weather\_monitor.event;  
  
import com.example.weather\_monitor.data.Country;  
  
/\* Event to parse country weather recorder change \*/  
public record CountryRecordToggleEvent(Country country) {  
 @Override  
 public String toString() {  
 return "CountryWeatherEvent{" +  
 "country=" + country +  
 '}';  
 }  
}

package com.example.weather\_monitor.event;  
  
import com.example.weather\_monitor.data.RegisterOption;  
  
/\* Event to parse register options changes \*/  
public record RegisterConfigEvent(RegisterOption option) {  
  
 @Override  
 public String toString() {  
 return "RegisterConfigEvent{" +  
 "option=" + option +  
 '}';  
 }  
}

package com.example.weather\_monitor.event;  
  
import com.example.weather\_monitor.db.WeatherRecord;  
  
/\* Event to parse new weather register \*/  
public record WeatherRecordEvent(WeatherRecord weatherRecord) {  
 @Override  
 public String toString() {  
 return "WeatherRecordEvent{" +  
 ", weatherRecord=" + weatherRecord +  
 '}';  
 }  
}

Zdarzenia te są dodawane do głównej magistrali jako reakcja na dane działanie aplikacji, na przykład:

centralEventBus.publishEventFromListener(countryRecordToggleEvent);

centralEventBus.publishEventFromListener(registerConfigEvent);

centralEventBus.publishEventFromListener(generateEvent());

Główna magistrala przekazuje następnie zdarzenia do obiorców zdarzeń, które połączone zostały w następujący sposób:

centralEventBus.registerNewListener(countryThreadsManageListener);  
centralEventBus.registerNewListener(weatherUpdatesListener);  
centralEventBus.registerNewListener(registerListener);

Odbiorcy posiadają obsługę wydarzeń o konkretnych klasach w taki sposób, że jedno zdarzenie zostanie obsłużone przez jednego, konkretnego odbiorcę, na przykład dla WeatherUpdatesListener:

@Subscribe  
 public void handleEvent(WeatherRecordEvent event) {  
 System.*out*.println(this.getClass().getSimpleName() + " " + *eventsHandled*++ + ": " + event.toString());  
 try (DBManager dbManager = new DBManager()) {  
 dbManager.addRecord(event.weatherRecord());  
 } catch (IOException | SQLException e) {  
 throw new RuntimeException(e);  
 }  
 *weatherMonitorController*.setWeatherPromptData(event.weatherRecord());  
 }  
}

Dzięki temu cała aplikacja działa w architekturze zdarzeniowej

### Wykorzystanie danych pobranych z dwóch zewnętrznych usług webowych

Pierwsza usługa to API, z którego czerpane są informacje o pogodzie. Do komunikacji z API służy klasa „APIWeatherManager”. Tutaj generowany jest URL połączenia z API:

public static URL generateQueryUrl(Country country) throws IOException {  
  
 final Properties properties = new Properties();  
 String propFileName = "config.properties";  
 InputStream inputStream = APIWeatherManager.class.getClassLoader().getResourceAsStream(propFileName);  
  
 if (inputStream != null) {  
 properties.load(inputStream);  
 } else {  
 throw new FileNotFoundException("property file '" + propFileName + "' not found in the classpath");  
 }  
  
 String BASE\_URL = properties.getProperty("api.url");  
 String API\_KEY = properties.getProperty("api.key");  
  
 String encodedCountry = URLEncoder.*encode*(String.*valueOf*(country), StandardCharsets.*UTF\_8*);  
 String queryUrl = BASE\_URL + "?q=" + encodedCountry + "&appid=" + API\_KEY + "&units=metric";  
 return new URL(Objects.*requireNonNull*(queryUrl));  
}

Natomiast tutaj tworzony jest obiekt klasy „WeatherRecord” zwracający pobrane informacje o pogodzie:

public static WeatherRecord getWeatherData(Country country) {  
 JSONObject weatherData = *getAllWeatherData*(country);  
 if (weatherData != null) {  
  
 WeatherRecord weatherRecord = new WeatherRecord();  
  
 String countryString = weatherData.getString("name");  
 if (StringUtils.*containsWhitespace*(countryString)) {  
 countryString = switch (countryString) {  
 case "Luxembourg Province" -> "Luxembourg";  
 case "Czech Republic" -> "Czechia";  
 case "Republic of Lithuania" -> "Lithuania";  
 default -> countryString;  
 };  
 }  
 weatherRecord.location = Country.*valueOf*(countryString);  
 weatherRecord.time = LocalDateTime.*now*();  
 weatherRecord.weatherCondition = weatherData.getJSONArray("weather").getJSONObject(0).getString("main");  
 weatherRecord.temperature = weatherData.getJSONObject("main").getDouble("temp");  
 weatherRecord.pressure = weatherData.getJSONObject("main").getInt("pressure");  
 weatherRecord.cloudiness = weatherData.getJSONObject("clouds").getInt("all");  
  
 return weatherRecord;  
 }  
 return null;  
}

Drugą usługą webową jest baza danych ElephantSQL, która przechowuje informacje o historii zarejestrowanej pogody. Tutaj tworzone jest połączenie z bazą:

public DBManager() throws IOException, SQLException {  
 final Properties properties = new Properties();  
 String propFileName = "config.properties";  
 InputStream inputStream = getClass().getClassLoader().getResourceAsStream(propFileName);  
  
 if (inputStream != null) {  
 properties.load(inputStream);  
 } else {  
 throw new FileNotFoundException("property file '" + propFileName + "' not found in the classpath");  
 }  
  
 Configuration configuration = new Configuration();  
 configuration.setProperty("hibernate.dialect", properties.getProperty("db.dialect"));  
 configuration.setProperty("hibernate.connection.url", properties.getProperty("db.url"));  
 configuration.setProperty("hibernate.connection.username", properties.getProperty("db.user"));  
 configuration.setProperty("hibernate.connection.password", properties.getProperty("db.password"));  
 configuration.setProperty("hibernate.hbm2ddl.auto", properties.getProperty("db.auto"));  
 configuration.addAnnotatedClass(WeatherRecord.class);  
  
 this.sessionFactory = configuration.buildSessionFactory();  
 this.session = sessionFactory.openSession();  
  
 maxRowsInRegister = Integer.*parseInt*(properties.getProperty("max\_rows\_in\_register"));  
}

Tutaj jest metoda, która służy to stworzenia odpowiedniej tabeli:

public void uploadInstanceToDatabase() {  
 Transaction transaction = session.beginTransaction();  
 session.persist(new WeatherRecord());  
 transaction.commit();  
}

A tutaj są metody służące dodania, pobrania oraz usuwania wierszy z tabeli:

public void addRecord(WeatherRecord weatherRecord) {  
 Transaction transaction = session.beginTransaction();  
 session.persist(weatherRecord);  
 transaction.commit();  
}  
  
public List<WeatherRecord> getRecords() {  
 Query<WeatherRecord> query = session.createQuery("FROM WeatherRecord ORDER BY time DESC", WeatherRecord.class);  
 query.setMaxResults(maxRowsInRegister);  
 return query.list();  
}  
  
public void clearRecords() {  
 Transaction transaction = session.beginTransaction();  
 session.createQuery("DELETE FROM WeatherRecord").executeUpdate();  
 transaction.commit();  
}

W momencie wykonania danej operacji połączenie jest zamykane:

@Override  
public void close() {  
 session.close();  
 sessionFactory.close();  
}

### Bazy danych (JDBC, JPA)

Aplikacja wykorzystuje technologię hibernate dla ORM-a przy połączeniu z bazą danych. Ze względu na rodzaj bazy danych wybrałem dialect dla hibernate o nazwie „PostgreSQLDialect”.

### Config properties

Aplikacja posiada plik o nazwie „config.properties” posiadająca zbiór kluczy i parametrów systemu potrzebnych do jego uruchomienia.

Poniżej na podstawie połączenia z bazą danych przedstawiony jest sposób pobierania informacji z pliku:

final Properties properties = new Properties();  
String propFileName = "config.properties";  
InputStream inputStream = getClass().getClassLoader().getResourceAsStream(propFileName);  
  
if (inputStream != null) {  
 properties.load(inputStream);  
} else {  
 throw new FileNotFoundException("property file '" + propFileName + "' not found in the classpath");  
}  
  
Configuration configuration = new Configuration();  
configuration.setProperty("hibernate.dialect", properties.getProperty("db.dialect"));  
configuration.setProperty("hibernate.connection.url", properties.getProperty("db.url"));  
configuration.setProperty("hibernate.connection.username", properties.getProperty("db.user"));  
configuration.setProperty("hibernate.connection.password", properties.getProperty("db.password"));  
configuration.setProperty("hibernate.hbm2ddl.auto", properties.getProperty("db.auto"));

Dzięki wykorzystaniu tego pliku można w jednym miejscu w prosty sposób edytować informacje o systemie oraz połączenie z bazą danych.

### Własne 2 komponenty graficzne

GUI aplikacji posiada wiele własnych komponentów graficznych takich jak przycisku, pola tekstowe, pola na obrazy, zbiory komponentów takie jak V-box oraz H-box.

### Dodatkowo punktowane wykorzystanie reflection API

W momencie obsługi wydarzeń zaimplementowana jest taka linijka wyświetlająca w konsoli informację o tym jaki odbiorca otrzymał event i który z kolei event to jest:

System.*out*.println(this.getClass().getSimpleName() + " " + *eventsHandled*++ + ": " + event.toString());

Wykorzystana tu jest metoda „getClass()” pozwalająca na pobranie informacji o nazwie klasy za pomocą „getSimpleName”.

### Wykorzystanie bibliotek Apache Commons oraz Magistrali zdarzeniowych

Do pobierania obrazu dla wyświetlenia stanu pogody wykorzystywana jest taka linijka:

weatherImage.setImage(new Image(weatherImageSource + StringUtils.*capitalize*(weatherRecord.weatherCondition) + ".png"));

Przychodzące informacje z api są z małej litery, natomiast grafiki mają nazwy plików z wielkiej litery, stąd potrzeba użycia metody „capitalize()”.

Oprócz tego wykorzystana jest metoda „containsWhitespace()”:

if (StringUtils.*containsWhitespace*(countryString))

Służy ona do sprawdzenia, czy otrzymana nazwa kraju zawiera spację, co w przypadku zaistnienia takiej sytuacji wymaga dodatkowych działań.

### Zastosowanie wzorców programowania obiektowego i zdarzeniowego

Cała aplikacja składa się z klas, ich tworzeniu oraz wywoływaniu.

Wszystkie działania, jakie może wykonać użytkownik w aplikacji reprezentowane są jako zdarzenia, które aplikacja potem obsługuje. Dodatkowo wątki pobierające informacje o pogodzie również wysyłają zdarzenia do obsługi.

### Zastosowanie wzorców projektowych z serii strukturalnych i behawioralnych

Wzorzec strukturalny: Adapter

Elementy aplikacji działają jak pośredniki pomiędzy komponentami. Klasa „DBManager” służy jako połączenie aplikacji z bazą danych, a klasa „APIWeatherManager” służy jako połączenie aplikacji z API.

Wzorzec behawioralny: Obserwator

Wzorzec ten jest zastosowany w klasach „CountryThreadsManageListener” oraz „WeatherUpdatesListener”, które są oznaczone adnotacją @Subscribe i implementują metodę handleEvent. Obie te klasy są subskrybentami (obserwatorami) zdarzeń publikowanych na magistrali zdarzeń (EventBus). Gdy wystąpi odpowiednie zdarzenie, metoda handleEvent zostanie wywołana, co umożliwia reakcję na zdarzenia i wykonanie odpowiednich operacji.

### Wymaga się całkowitego autorstwa kodów źródłowych

Aplikacja została w pełni i od początku napisana przez autora sprawozdania.

# Podsumowanie

Projekt był bardzo dobrą okazją na poznanie aspektów programowania w zakresie architektury opartej na zdarzeniach. Największą trudność sprawiło samo zrozumienie tematu, natomiast po zapoznaniu się ze specyfikacją architektury oraz podejściem, jakie trzeba wobec niego przyjąć, udało się stworzyć działającą aplikację.