Criterion C: Development

Table of Contents

Word Count: 885 (excluding code listings)

JavaFX for GUI

The JavaFX framework *(Oracle, and Sun Microsystems)* was used to develop the front-end GUI. This was accomplished by creating FXML 'scenes', which use an XML-format file for describing the user interface, and then calling the FXMLLoader class to render these scenes to the user's screen. Below is a snippet of the FXML used:

```
<!-- This is a Text node, representing text, which contains a Font node
describing the font style, and HBox.margin describing the margins for the
container node -->
<Text fill="#dddddd" strokeType="OUTSIDE" strokeWidth="0.0" text="ibia">
    <font>
        <Font name="System Bold" size="28.0" />
    </font>
    <HBox.margin>
        <Insets left="10.0" />
    </HBox.margin>
</Text>
```

Below is a snippet of the code used to load the FXML scenes:

```
// 'FXMLLoader' is imported from javafx.fxml.FXMLLoader;
FXMLLoader loader = new FXMLLoader();
// 'name' is the name of the file containing the FXML describing the GUI
loader.setLocation(getClass().getResource("/fxml/" + name));
// 'content' contains all the nodes of the GUI, as described by the FXML
Parent content = loader.load();
```

In JavaFX, a 'stage' represents the window being displayed to the user, while 'scenes' represent the actual user interface. The above code only loads the FXML into the 'content' variable. The next step is to load this scene onto a stage and display it:

```
Stage stage = new Stage(); // Create a stage instance
Scene scene = new Scene(content); // Use the loaded content to create a Scene
instance
stage.setScene(scene); // initialize the stage with this scene
stage.setTitle(title); // set title of the window
stage.show(); // This method displays the window to the user's screen
```

Use of Object-Relational Mapper (Hibernate)

Hibernate *(Red Hat)* is an Object-Relational Mapper (ORM) for Java that provides a powerful API for interfacing with databases, while dealing with data conversion and querying under the hood. With Hibernate, queries are usually executed through Session objects. The Session object first needs to be obtained from the SessionFactory, which manages the database configurations:

```
private static SessionFactory getSessionFactory() {
    if (sessionFactory == null) {
        try {
            // Create registry and configure database
            // The configuration file is automatically read from the resources/
directory.
            registry = new StandardServiceRegistryBuilder().configure().build();
            // Create MetadataSources
            MetadataSources sources = new MetadataSources(registry);
            // Create Metadata
            Metadata metadata = sources.getMetadataBuilder().build();
            // Create SessionFactory
            sessionFactory = metadata.getSessionFactoryBuilder().build();
        } catch (Exception e) {
            e.printStackTrace();
            if (registry != null) {
                // Close registry if there were errors initializing database
                StandardServiceRegistryBuilder.destroy(registry);
            }
            throw e;
        }
    }
    return sessionFactory;
}

private static Session openSession() {
    // Since session objects are created frequently, this method is used for
convenience
    return getSessionFactory().openSession();
}
```

The session object can then be used to make transactions, which represent queries, with the database, like so:

```
Session session = openSession();
session.beginTransaction();
```

```
session.save(entity); // entity is the object we want to save to the database
session.getTransaction().commit();
session.close();
```

The use of Hibernate made the database management process a lot smoother and provided powerful functionality such as caching database results as well as formulating more powerful queries. One important advantage was that queries could be formed and executed using Java methods instead of SQL query strings. This meant that the Java methods, and thus the database queries, could be type-checked at compile time, leading to less bugs and smoother development. This example shows a query constructed programmatically, which returns all the rows in a table:

```
Session session = openSession();
session.beginTransaction();
CriteriaBuilder cb = session.getCriteriaBuilder(); // CriteriaBuilder is used to
programmatically build complex queries.
CriteriaQuery<T> cq = cb.createQuery(entityClass); // CriteriaQuery represents
one such query
Root<T> rootEntry = cq.from(entityClass); // analogous to SQL's FROM keyword
CriteriaQuery<T> all = cq.select(rootEntry); // analogous to SQL's SELECT keyword

TypedQuery<T> allQuery = session.createQuery(all); // TypedQuery represents the
final query created from the above methods, except this one is type checked by
the compiler
ArrayList<T> results = (ArrayList<T>)allQuery.getResultList(); // execute query
and obtain the results
session.close();
return results; // return the results
```

Enumeration Types

Enumerations were used to encode the main entity types used in the application. It was defined as:

```
public enum EntityType {
    COM, // COM stands for Committee
    CON, // CON stands for Conference
    DEL, // DEL stands for Delegate
    ENT  // ENT stands for some Entity not belonging to the previous three
}
```

This was useful in parts of the application which needed to deal with all possible entities available in the application, so that an entity is not 'forgotten'. This was aided by the type

checker and made development safer. A good example is its use in ID generation, since a unique ID had to be generated for each entity type:

```
public static String generate(EntityType type) {
    String ts = Long.toString(System.currentTimeMillis());
    switch (type) {
        // The cases below deal with all possible values of the EntityType enum,
ensuring that none of them are missed out.
        case COM:
            return "COM" + ts;
        case CON:
            return "CON" + ts;
        case DEL:
            return "DEL" + ts;
        default:
            return "ENT" + ts; // The default case handles the remaining 'ENT'
type.
    }
}
```

Polymorphism: Interfaces

The Entity interface was used to write reusable polymorphic methods that would work regardless of the type provided as long as it implemented the interface. This was particularly useful in writing helper methods for the database queries:

```
public interface Entity {
    public EntityType getType();

    /* GETTERS and SETTERS used by hibernate */
    public String getId();
    public void setId(String id);

    public String getName();
    public void setName(String name);
}
```

The compiler then ensured that these methods were implemented for each entity, for example:

```
public class Conference implements Entity {
    // This is how the getType() method was implemented for Conference:
    private final EntityType type = EntityType.CON;
    public EntityType getType() {
        return type;
    }
```

```
    /* . . . rest of the class implementation . . . */
}
```

Polymorphism: Generic Programming

Generic programming was used to write reusable database methods that would work for any entity type within the application, rather than having to define separate methods or queries for each entity:

```
// Here, T is a type variable, and can be used throughout the method definition
just like any other type.
public static <T> void deleteAll(Collection<T> entities) {
    Session session = openSession();
    session.beginTransaction();
    for (T entity : entities) session.delete(entity);
    session.getTransaction().commit();
    session.close();
}
```

This pattern was used for nearly all database methods, allowing the same methods to be used for any entity defined in the application.

Higher-order Functions

Higher-order functions are a feature of functional programming introduced in Java 8 *(Gosling)* through the addition of anonymous functions. The benefit is that they allow more concise, readable, and reusable code. These were used throughout the codebase, especially for filtering database results. For example:

```
public static <T> ArrayList<T> findAll(Class<T> entityClass, Predicate<T> filter)
 {
    ArrayList<T> found = new ArrayList<>();
    ArrayList<T> entities = fetchAll(entityClass);
    if (entities != null) {
        for (T entity : entities) {
            if (filter.test(entity)) found.add(entity);
        }
    }
    return found.size() > 0 ? found : null;
}
```

Here, the predicate function is a higher order function that is used to test all database entities for a condition and returns all that pass the condition. It can be used like so:

```
ArrayList<Conference> confs = DbDriver.findAll(Conference.class,
                                                    c -> c.isOngoing());
```

This returns an ArrayList of all Conferences that are marked ongoing. This is a simple example, although anonymous functions can be used for much more complex tasks as well.

Caching Behavior through HashMaps

Every time a new JavaFX scene is loaded, the FXML file needs to be read, parsed, and rendered. To improve user experience and the overall speed of the application, simple caching behavior was implemented so that resources are not wasted on Scenes that need to be loaded repeatedly. This was achieved using HashMaps:

```
public final class SceneUtil {
    // cache will store the filename as the key and the loaded FXML content as
the value
    private HashMap<String, Parent> cache = new HashMap<>();

    // the useCache parameter can be used to specify whether the cache should be
used when loading the given FXML file. Sometimes caching may not be desirable
(due to mutability of the cached objects).
    private Parent _loadFXML(String name, boolean useCache) throws IOException {
        // if the FXML is already cached, return that.
        if (useCache && cache.containsKey(name)) return cache.get(name);

        name = name.endsWith(".fxml") ? name : name + ".fxml";
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(getClass().getResource("/fxml/" + name));
        Parent content = loader.load();

        // if the FXML wasn't cached, and useCache is true, then store it for
next time.
        if (useCache) cache.put(name, content);
        return content;
    }
    /* . . . rest of the class implementation . . . */
}
```

File-Handling: Log Files

Log files were included to store output from the database ORM, JavaFX framework, as well as the application's own information, warning, and error messages:

```
public static Logger getLogger() {
    if (logger == null) {
        logger = Logger.getLogger("");
        logger.setUseParentHandlers(false); // No need output to parent loggers
        try {
            // Create the log file if it doesn't exist
            String path = getLogFilePath();
            File file = new File(path);
            if (!file.isFile()) {
                file.getParentFile().mkdirs();
                file.createNewFile();
            }
            // Set the formatter and add handler to logger
            SimpleFormatter fmt = new SimpleFormatter();
            handler = new FileHandler(path, true);
            handler.setFormatter(fmt);
            logger.addHandler(handler);

            // Indicate a new log session is starting.
            info("= = = = = SESSION START = = = = =");
        } catch (Exception e) {
            // Handle error appropriately – do not crash the application if log
file does not load since it is not an essential function of the app
            e.printStackTrace();
            System.out.println("Failed to access log file (data/ibia.log).");
        }
    }
    return logger;
}
```

This method tries to load a log file when called. If it fails, it does not crash the entire application, and simply tries again next time. If the file is not available, it tries to create one.

Within the application, an option was provided to view to contents of the log file to the user, for troubleshooting purposes. This required the log file to be read efficiently, as it can grow quite large. This was done using a buffered file reader:

```
FileReader fr = new FileReader("data/ibia.log");
// Use BufferedReader for fast reading
BufferedReader br = new BufferedReader(fr);
String contents = "";
while (true) {
    String line = br.readLine(); // Read contents line by line
    if (line == null) break; // Stop reading after reaching end of file
    contents += line + "\n"; // Store contents into variable
```

```
}
br.close();
```

File-Handling: Images

Icons for country flags were displayed when possible, to improve the user experience and interface of the application. This required loading image files (all files were of the PNG format), which was achieved by loading them as resources, into InputStreams:

```
public static InputStream getFlag(String code) {
    if (listOfCodes().contains(code)) {
        // First, create the path of the file using the code provided
        String fileName = code.toLowerCase() + ".png";
        String resource = "world-countries/flags/64x64/" + fileName;
        // Load the image from the resources directory, as an InputStream
        InputStream flag = Country.class.getClassLoader().getResourceAsStream(res
ource);
        return flag;
    }
    return null;
}
```

The InputStream can then be used to instantiate a JavaFX Image object, to be rendered to the application window and displayed to the user.

Parsing Json

Data for the country codes and their icon files were obtained from the stefangabos/world_countries GitHub repository *(Gabos)*. This was in JSON format and needed to be parsed before it could be used inside the application. This was achieved using the Gson *(Google)* library. First, the structure of the JSON data needed to be defined using a java class, which would then be used by the library:

```
public static class CountryData {
    public int id;
    public String name;
    public String alpha2;
    public String alpha3;
}
```

Then, the JSON file could be read and parsed into instances of this class:

```
String dataPath = "world-countries/data/en/world.json"; // Path to file
ClassLoader classLoader = Country.class.getClassLoader();
InputStream stream = classLoader.getResourceAsStream(dataPath);
```

```
int b;
StringBuilder str = new StringBuilder();
while ((b = stream.read()) != -1) { // Read stream into StringBuilder
    str.append((char)b);
    stream.close();
}

String json = str.toString(); // Convert StringBuilder to normal string,
containing the json from the file
data = new Gson().fromJson(json, CountryData[].class); // Here, Gson will read
the JSON string and parse its contents into an array of CountryData instances.
```

Sources

Gosling, James, et al. “The Java® Language Specification: Java SE Edition.” *Chapter 15. Expressions*, 13 Feb. 2015, www.docs.oracle.com/javase/specs/jls/se8/html/jls-15.html.

Gabos, Stefan. "Stefangabos/World_Countries". *Github*, 2020, https://github.com/stefangabos/world_countries. Accessed 25 Nov 2020

Google. "Google/Gson". *Github*, 2020, https://github.com/google/gson.

Red Hat. "Hibernate. Everything Data. - Hibernate". *Hibernate.Org*, 2020, http://hibernate.org/.

Oracle, and Sun Microsystems. "Main - Openjdk Wiki". *Wiki.Openjdk.Java.Net*, 2020, https://wiki.openjdk.java.net/display/OpenJFX/Main.