Demystifying the OCMJD Certification

Roberto Perillo[1]

[1]Aeronautical Institute of Technology
Praça Marechal Eduardo Gomes, 50 – Vila das Acácias – CEP 12228-900
São José dos Campos – SP, Brazil

jrcperillo@yahoo.com.br

Abstract. *Of all Java certifications currently available, OCMJD is the only one that effectively demands the candidates to demonstrate in practice their ability to solve problems of the real world using the Java language. This certification demands the candidates to show some level of proficiency in some topics that are not covered in other certifications, such as RMI/Sockets and Swing. This paper shows what candidates can expect when opting for this certification, the steps that have to be taken in order to become certified, the pitfalls hidden behind the requirements and a fictitious assignment partially solved.*

1. Introduction

The Java language currently has eight certifications, where only two of them are based on practical performance, that is, the candidate only succeeds after developing an assignment and achieving a minimal score after evaluation. One of these certifications is the OCMJD (Oracle Certified Master Java Developer, formerly known as SCJD [Camerlengo and Monkhouse 2005]), where the candidate has to develop a programming assignment and take an essay exam based on the assignment that was developed. In the assignment, the candidate has to build a client/server program that must follow some "musts" and prepare several documents related to the assignment. In the essay exam, the candidate has to answer four generic questions (or that apply to any assignment). Although the questions refer to specific points of the resolution of the assignment, the main goal of the essay exam is to verify if the assignment was really developed by the candidate that requested it.

In order to request this certification, the candidate must have any version of the OCPJP certification, and since it is a two-phase certification, it takes two vouchers to finish it. When requesting the certification, the candidate receives a JAR file that contains a file called instructions.html and a .db file that represents a non-relational database. The instructions.html file contains all necessary rules and instructions for developing the assignment, and it also contains the code of the interface that must be implemented by a class that must have a specific name, indicated in the instructions as well. This class will be responsible for accessing the .db file, received in the JAR file.

After developing the assignment and all required documents, the candidate has to take the essay exam and then upload the assignment, which must be done up to one year after the assignment was first downloaded. After the evaluation of the assignment, the candidate receives an email that says whether the candidate was approved or not.

The present paper is organized as follows: first, an analysis is made over the real assignment. After that, it is shown some techniques and patterns that can be used in order

to solve the assignment. The steps that can be taken, with examples of how each component can be developed, are addressed next. Each execution mode that must be supported by the application is addressed right after. Next, it is given a final overview of the assignment, with details of each artifact that must be delivered. The locking mechanism, which is the most complex item of the assignment, is addressed after that. And finally, some final considerations are made, finishing the present paper.

2. The Assignment

Currently, there are two models of assignment: URLyBird and Bodgitt & Scarper. In practice, the essence of these models is pretty much the same, with small variations between their versions. Basically, the assignment consists in the development of a client/server application that accesses a binary file. This application must allow the user to reserve available resources and must also allow the user to search for resources according to a given criteria. The assignment does not contain complex business rules, but the candidate must develop both server (using either RMI or Sockets) and client. The server must be capable of dealing with multiple simultaneous requests and must manage the access to the binary file through a locking mechanism, which must protect the file's integrity, so that one client does not override other client's data.

The application must also be executable in standalone mode, which means that the application must provide the user windows and must access the binary file locally, without using server's code. Regarding the windows, they must be built with Java Swing and may use AWT in the absence of equivalent Java Swing components. For instance, the candidate must use `javax.swing.JButton` instead of `java.awt.Button`, and may also use `java.awt.event.ActionListener`. The windows that must be built are not complex, having only to allow the user to provide configuration data, view the binary file's records and perform searches and bookings. Therefore, the candidate must built simple windows using the Java Swing framework, create a server capable of dealing with multiple simultaneous requests and managing the access to the binary file through a locking mechanism and must also create the client that connects to this server. What defines whether the data should be retrieved locally or remotely is a parameter passed in the command line when the application is executed.

During the development, the candidate faces many questions, due to the fact that the instructions provided by Sun have several ambiguous points. Although it may seem to the candidate that some information might be missing, the fact is that Sun leaves some open questions on purpose. There is a statement in the end of the document that says that it "deliberately leaves some issues unspecified, and some problems unraised". The ability that the candidate has to make decisions in face of these doubts, deal with poor requirements and define solutions with the available options is part of the evaluation.

The candidate must also be capable of filtering what must be done and what does not have to be done. For instance, the candidate might ask whether it is necessary to apply the Observer pattern [Gamma et al. 1994], so that when the application is executed in client mode (connected to a server) and one record is updated, all clients are notified. This is an example of a requirement that does not have to be implemented. Therefore, the main evaluation criteria is to verify whether the candidate is able to do what is asked in a simple way. The candidate does not get extra points for implementing features that

are not asked; instead, might even lose points for that. It is also worth reminding that functionalities provided by the Java API are preferred over own implementations.

One of the first things provided in the instructions document is the text that identifies what are the client's needs. Although the following example is fictitious, it shows what candidates may expect in the real assignment:

"Bob Lennon's Tour is a tourism company that acts in the Brazilian territory. One of its departments organizes bus excursions throughout Brazil. A new governmental law demanded a unique registry of bus excursions, and thus, all companies that organize excursions must register them in this registry. As of a public bidding, Bob Lennon's Tour became responsible for creating this registry. In the first phase, the idea is to run the system only in the company's local network, where the sales representatives will access them from a local server, and thus, companies will have to contact them in order to register their excursions over the phone, but in a time to be defined, the system will be migrated to a web environment, allowing companies to register their excursions by theirselves.

For such, the company's CEO, Bob "John Lennon", decided to engage the development of a new Java application. At this moment, the idea is to develop something only to attend the first phase, since the migration of the application to a web environment is still undefined. So, it is not foreseen whether the developed components will be reused in the web application, but maybe will be. One of the requirements of this new registry is the storage of data in a .txt file, whose format was also provided by the government. Therefore, it will not be possible to make use of a DBMS, such as MySQL, for instance, and thus, it will also be necessary to create a component that accesses this file, respecting the format that was provided by the government.

The new Java application, using the format of the existing file, must allow the CSRs to create data of new excursions, delete data of excursions previously created and show data of a particular excursion. This is the project whose development was assigned to you."

Besides the development, the candidate must list in a file called choices.txt all design and architectural choices made during the development. In this file, the candidate must also list ambiguous points found in the requirements, the corresponding choices and the reasons why the candidate took such decisions. Although Sun recommends that a set of code conventions is used throughout the code, the candidate must preferably use Sun's code conventions, and all elements (at least the ones of the public interface) must contain JavaDoc comments. One good tip is to also provide comments to the non-public elements.

With the JavaDoc comments correctly commenting the class elements, the candidate must use the javadoc tool to generate the JavaDoc documentation, which must also be delivered. The candidate must also create a user guide that should be delivered as a text file, HTML or within the application as a help system. The content of this guide must contain enough information to allow a user who is familiar with the purpose of the application to use it. Another document that must also be delivered is a file called version.txt, where the candidate must simply indicate the version of the JDK used and the operating system where the application was built. Another good tip is to always use the latest JDK released.

3. Solving the Assignment

There are several architectural patterns that can be used in order to solve the assignment. It is a good idea to keep things as simple as possible, since the defined architecture and how easy it is to use and maintain the developed code is also part of the evaluation. It is shown next the patterns that can be considered and used, and finally, it is shown a comparison between the presented domain organization patterns and how everything can be put together, in order to solve the assignment in a flexible and easy way.

3.1. Traditional Four-Tier Layered Architecture

In the excellent book [Panda et al. 2007], the authors address an architectural pattern called "traditional four-tier layered architecture", which is quite popular among designers of enterprise applications. Although the application to be developed is essentially a client/server application, this pattern can be followed to solve the assignment.

The four layers that compose this style of architecture are: presentation layer, business layer, persistence layer and database layer, as shown in Figure 1. The presentation layer is responsible for dealing with UI concerns, such as showing visual messages to the user, enabling or disabling text fields, dealing with radio buttons, etc. This layer must contain only presentation rules, not business rules, which must be concentrated only in the business layer. This layer represents the heart of this style of architecture and is responsible for concentrating all business rules implemented by the application. This layer does not retrieve or persist information to the database; instead, all database-related operations are intermediated by the persistence layer, which is an object-oriented abstraction over the database layer. Like the presentation layer, this layer must not contain any business rules, and must "dumbly" obey all request made by the business layer.

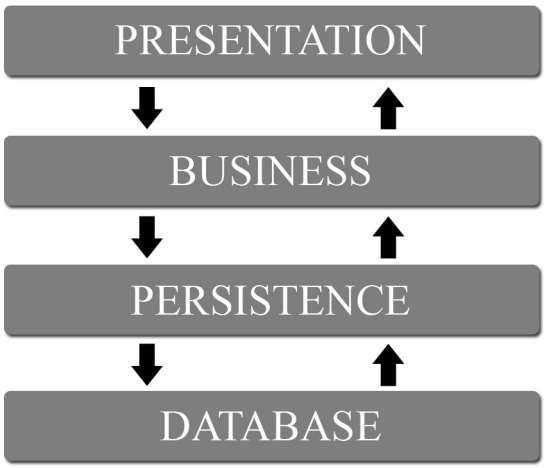

Figure 1. Organization of the four-tier layered architecture.

The layers of the presented architecture are logical, which means that they may be physically gathered (in the same JAR file, for instance). The greatest benefit of separating the application in layers is that each layer may be designed to deal with only one particular concern, promoting reuse, flexibility and easing the maintenance. When opting to organize the application in layers, it must be considered that the components of an upper layer must only communicate between them or with components of a lower layer (when

a component of a lower layer has to communicate with an element of an upper layer, callback mechanisms or the Observer pattern can be used). To design and define the API of a particular layer in isolation helps to keep it simple and flexible. For instance, one can define the components of the business layer without worrying whether these components are to be used in a web or desktop application. How easy it is to maintain and evolve the developed code is an important aspect of the evaluation.

3.2. Transaction Scripts

In the excellent book [Fowler 2002b], Martin Fowler addresses a domain logic organization pattern called Transaction Script, whose key is simplicity. This pattern organizes the business logic in a set of transaction scripts, where transaction script is a method that handles requests from the presentation layer and organizes the domain logic, performing computations and accessing the database through DAOs [Alur et al. 2003]. This pattern promotes a procedural structure, where normally classes tend to be coarse-grained and data become separate from processing.

Ideally, transaction scripts should be gathered in classes that deal with similar tasks. These classes contain only transaction scripts, and thus, contain only behavior. It is very common in application to find classes with transaction scripts in packages called "business" or "services". The former is preferred, since "services" can give the sense of web services. Martin Fowler gave this name to this pattern because, in most cases, the handling of a request from the presentation layer implies in a transaction, or a set of operations that must be processed atomically.

Transaction scripts normally deal with "dumb" objects that exist only to transfer values from one layer to another (known as DTOs [Alur et al. 2003]). Therefore, given a request from the presentation layer, the transaction script receives or fills a DTO, utilizes its data to perform computations, with the possibility of passing it to the persistence layer if the method requires persistence, and also with the possibility of returning another DTO back to the presentation layer. A domain model that contains objects that only exist for data transferring between layers and do not have any intelligence is known as anemic domain model [Fowler 2003].

The usage of transaction scripts is very simple because its appliance and straightforward and it is not necessary to worry about identifying classes and assign responsibilities to them, as it happens when the Domain Model [Richardson 2006] is applied, and thus, it is not necessary to have much experience with object orientation. At the same time, the simplicity proposed by this pattern is also a limitation. Since a transaction script is a routine that handles requests from the presentation layer, code may get big, becoming hard to understand and maintain. Therefore, its appliance is indicated mainly when domain logic is simple.

A transaction script may not be a method of a Java class; it can be, for instance, a CGI routine. If there is a routine that handles a request from the presentation layer and organizes domain logic, where this routine deals with concerns that should be distributed in other classes (or structures), and these classes (or structures) would handle only the concerns of their competence, then this routine can be considered a transaction script.

3.3. Domain Model

In another excellent book [Richardson 2006], Chris Richardson shows in practice how to apply the Domain Model pattern, which is another domain logic organization pattern. Domain Model is Domain-Driven Design's [Evans 2003] greatest product, and is simply an abstraction of the knowledge that the domain experts have. In order to be implemented, it requires a special architectural pattern, which is also called Domain Model. Unlike the Transaction Scripts pattern, the Domain Model pattern proposes to implement business logic in a rich domain objects model that utilizes all forces and benefits offered by object orientation.

The domain classes are invoked directly or indirectly by the presentation layer and the handling of a request results in the invocation of one or more entities, which validate user's inputs, perform computations, deal with business logic and access the database through repositories. When following the structure proposed by this pattern, it becomes easy to maintain and evolve the application, because responsibilities are spread among high cohesive and low-grained classes. Since the classes that implement the domain model are POJOs, it is possible to benefit from all advantages offered by object orientation and utilize any GoF pattern [Gamma et al. 1994] with no restrictions.

The building blocks, that is, classes that implement a domain model, play the following roles:

- Entities: are objects that own an identity that differs them from other objects. For instance, a class called Computer could have an attribute called serialNumber, which could be its identity. Entities represent concepts of the domain model and concentrate most part of the business logic implemented by the application.
- Value objects: are objects that are defined by their values and do not have an identity. They are essentially immutable, that is, once created, cannot be changed, and therefore, must have their attributes initialized in the constructors due to the absence of "setter" methods. In practice, they exist only to compose entities and may result in the gathering of duplicate attributes in two or more entities or that make more sense to be separate from entities. The early J2EE literature referred to DTOs as value objects, whose goal is the data transportation between layers. Besides being immutable, DTO's differ from value objects in that a value object is an object that makes sense in a domain and a DTO is a "dumb" object used for data transportation.
- Factories: are objects that exist exclusively to create other objects. Factories are very useful when the objects graph to be instantiated is complex. Separating the code that creates these objects makes the code more flexible and easier to maintain, but the greatest benefit of factories comes when polymorphism is used. This way, the code that uses the factory can reference a super class or an interface, and thus the object returned by the factory can vary without the need of changing the code that uses the factory. In terms of design patterns, there are officially two patterns, called Abstract Factory and Factory Method [Gamma et al. 1994], but there are also other approaches that are popular among developers, such as static factories.
- Repositories: are objects that manage collections of entities and define methods for finding, creating, updating or deleting them. Since the Domain Model pattern is to be used when the domain logic is complex, then managing them

with JDBC may be overkill, and therefore, repositories are more likely to encapsulate persistence frameworks, which make the persistence mechanism transparent to the rest of the domain layer. A repository is abstracted by an interface and can have one or more implementations. The interface defines methods to be offered to the clients of the repository, and the implementations deal directly with the persistence frameworks. The difference between a DAO and a repository is a fine line: while a DAO encapsulates the data access (i.e. with methods like `save(Object)`, `update(Object)` or `delete(Object)`), a repository manages collections of entities and has a direct with the domain (i.e. in the domain of electronic equipments, it could have methods like `findTvsByManufacter(Manufacturer)`, `createComputer(Computer)`, `updateVideoGame(VideoGame)`, and so on).

- Services: are objects that organize the application's execution flow and act as the domain layer's API. Normally, their clients are the presentation layer or Façades [Gamma et al. 1994] that encapsulate the domain layer. They retrieve entities through repositories and delegate the execution of computations to them. Since they organize the application's execution flow, they normally have the responsibilities found on use cases or user stories. A Service is similar to an Application Service [Alur et al. 2003] because they can also contain specific use case or user story logic.

The reunion of these classes result in the implementation of the domain model, and this implementation corresponds to the "M" of the MVC pattern. For instance, let us consider a web application where a Front Controller [Alur et al. 2003] receives all possible requests and delegates the discovery of the business component that handles a given request to an Application Controller [Alur et al. 2003]. This Application Controller delegates the handling of the request to the model, and after that, redirects the user to the appropriate window. This way, the Controllers receive the requests and delegate their handling to the domain model, and after that, redirect the user to the appropriate window.

3.4. Model-Delegate Pattern

The Model-Delegate pattern is a small variation of the MVC pattern [BluePrints 2002]. While the latter has a clear separation between view and controllers, the former has a more intimate relationship between view and controllers.

Considering a web application that has a Front Controller that receives all possible requests that can be handled, the web pages (view) that send the requests are clearly separate from the controller. Besides that, the web pages run on the client's browser, which is on the client side, and the controller resides in the web environment, on the server side. On the other hand, considering a client/server application that uses the Java Swing framework, the window (view) can have buttons that trigger features performed by the application, and these buttons can have `ActionListeners` that capture the click event and delegate the handling of the user's action to the model. This way, the `ActionListener` acts like a controller, but is intimately bounded to the view as it listens to a particular button and resides on the same side as the view.

In both MVC and Model-Delegate, the model corresponds to the business intelligence implemented by the application. When the Domain Model pattern is used, the

model corresponds to the implementation of the domain model, and when transaction scripts are used, the model corresponds to the gathering of transaction scripts that reside in the business layer and the components that reside in the persistence layer. For a Java Swing application, it is more natural to use the Model-Delegate pattern.

3.5. Dependency Injection

Dependency Injection [Fowler 2002a] is a pattern proposed by Martin Fowler and aims to achieve a higher level of decoupling between classes. In order to execute its responsibilities, an object may require the collaboration of another object, instantiating it directly at runtime, and thus classes become tightly coupled with each other. Dependency Injection suggests that objects do not instantiate their dependencies at runtime, but that they should receive them, exempting these objects from the responsibility of building other objects, and thus, they can execute their main responsibilities. This promotes a high level of flexibility, mainly when interfaces or abstract classes are used, because objects that collaborate with other objects may be changed by other objects without the need of changes in the code that use them.

Obviously, this technique requires a code that is external to the code that receives the dependencies to instantiate the objects and arrange them. Generally, this task is performed by IoC Containers (that is, inversion of control containers). These containers receive this name because they control the life cycle of an amount of objects, that is, the instantiation and arrangement of objects is performed by them. Some frameworks, like Spring [Spring 2011] or PicoContainer [PicoContainer 2011] can play this role, but this can be done inside the application. For instance, when the application is started, a class can arrange the objects, and when they are about to execute their responsibilities, their dependencies will already be instantiated and injected.

In essence, dependency injection can be done in three ways: via constructor, setter method or interface. There is another way to inject dependencies that is supported by some containers, such as Spring or a JEE container, which is dependency injection via annotation. For instance, Spring offers the `@Autowired` annotation to indicate that an annotated element must be injected.

In the constructor injection, the dependencies are injected in the object that receives them via constructor method. One advantage of this approach is that the class API does not have to be changed only to support the approach. A disadvantage is that, if the class requires many dependencies, the constructor may become harder to use, especially if the arguments are of the same type.

When using the setter injection, the class provides a setter method that assigns to a variable at class level the attribute received by the method. This type of implementation requires the class to alter its API.

And finally, the interface injection is similar to the setter injection. The difference is that the setter method is defined in an interface that is implemented by the class. This type of dependency injection implies in the fact that all classes that implement the interface will have the dependency, and thus, this type of dependency injection only makes sense when all classes that must implement the interface must mandatorily have a particular dependency.

3.6. Patterns Reunion Overview

Regarding domain logic organization patterns, it may apparently seem better to use the Domain Model pattern instead of the Transaction Scripts pattern. But in [Richardson 2006], Chris Richardson also addresses situations where it makes more sense to use the Transaction Script pattern:

> The Domain Model pattern is an excellent way to organize complex business logic. However, there are situations where you might not want to use a domain model, such as when the development team lacks the necessary OO design skills to develop one or the business logic is very simple. It also does not make sense to use a domain model when you cannot use a persistence framework because, for example, the architecture does not include one or the application accesses the database in ways that require it to use SQL directly. In these situations, you should consider writing procedural business logic, an approach also known as the Transaction Script pattern.

Analyzing both instructions and Chris Richardson's considerations, the Transaction Script pattern can be considered more adequate to solve the assignment due to its simple nature. In the instructions, it is said that "a clear design, such as will be readily understood by junior programmers, will be preferred to a complex one, even if the complex one is a little more efficient". Therefore, it is possible to organize the overall architecture with the four-tier layered architecture style and build the business layer with transaction scripts, so that each transaction script deals with a request from the presentation layer, and at the same time, the Model-Delegate pattern can also be used, where buttons can have `ActionListeners` that have access to the windows that receive user's inputs and to the classes that have the transaction scripts responsible for handling user's actions. When the application starts, the objects can be instantiated properly and arranged through constructor dependency injection, and thus, the code can be kept flexible and easier to maintain and evolve.

4. Implementation Steps

After defining the architectural patterns to be followed, it is time to effectively think about the implementation. The figures that contain code shown here do not contain imports because the classes used are from the Java Core API, and also to save space. It is shown next the steps that can be taken in order to implement the assignment in an organized, flexible and easy way.

4.1. First Step: Implementation of the Data Access Class

Although in a well-defined architecture it makes no difference which layer the application starts to be implemented, a good place to start the development of the assignment is the class that must implement the interface provided by Sun. **Important: this class must contain the exact name indicated in the instructions and must also be in a package indicated in the instructions as well, otherwise, the candidate will automatically be failed.** This class is responsible for accessing the .db file provided by Sun. The provided interface is limited in some aspects, but how the candidate deals with these limitations is also part of the evaluation.

It is shown in Figure 2 a fictitious example of the interface provided by Sun in the instructions document.

```
package certification.database;

public interface DB {

    // Creates a new record in the database.
    public void createExcursion(Excursion excursion)
            throws DuplicateExcursionException;

    // Updates the fields of a given excursion.
    public void updateExcursion(long recordNumber, Excursion excursion)
            throws RecordNotFoundException;

    // Deletes a particular record.
    public void deleteExcursion(long recordNumber)
            throws RecordNotFoundException;

    // Reads a record from the database.
    public Excursion readExcursion(long recordNumber)
            throws RecordNotFoundException;

    // Locks a particular record, so that it can only be
    // updated or deleted by this client.
    public void lock(long recordNumber)
            throws RecordNotFoundException;

    // Unlocks a particular record.
    public void unlock(long recordNumber)
            throws RecordNotFoundException;

    // Other methods...
}
```

Figure 2. Example of the interface provided by Sun.

Analyzing the code shown in Figure 2, it is possible to have an idea of the format of the interface provided by Sun. The actual interface does not have collections, such as a List, or something like the `Excursion` class, for instance. In the real assignment, a record is represented by String arrays. This interface reflects two important aspects: data management and the locking mechanism. It is important to implement all methods, even if they are not used in any business rule implemented in the assignment. Besides being mandatory, it is a matter of reusability. Considering the example shown in Figure 2, if the DB interface is used somewhere in another application, it is possible to use the implementation made here.

In the example proposed by this paper, one of the requirements is the usage of the .txt file provided by the government. In the real assignment, the data file is a binary file that can be read by using the `java.io.RandomAccessFile` class. A code made by the author of this paper that reads and prints on the console the content of the binary file can be found in the JavaRanch's SCJD FAQ. Candidates can use it and understand how the file can be read and then create their own implementations.

In the real assignment, some methods of the interface provided by Sun receive String arrays as parameters, where each value of the array represents a field of a record in the database. The format of the file is also provided in the instructions document, and has informations like a value that indicates whether the file is valid or not, size in bytes of

each record, number of fields of each record, etc. The database schema is also provided, and has informations like the name and the size in bytes of each field. In the example proposed by this paper, the `Excursion` class reflects exactly the format of a database record.

```
package certification.domain;

// imports omitted…

public class Excursion implements Serializable {

    private int busNumber;

    private String companyName;

    private Date departure;

    private Date arrival;

    private String origin;

    private String destiny;

    private String price;

    private byte capacity;

    // Getters e Setters omitted...
}
```

Figure 3. Excursion domain class.

Regarding the method that creates a new record, the candidate will have to verify if it is possible to create a record with duplicate key, facing the provided schema. For such, the candidate will have to verify if it a field or combination of fields may be considered unique. If so, then the create method will throw the exception indicated in the method's signature, which must also be created. In the example proposed by this paper, busNumber, company and departure can be considered a unique combination of fields (considering that there is a remote possibility that two different companies can have two buses with the same number), and thus it is possible to extract these fields from the `Excursion` class and move them to another class, `ExcursionPrimaryKey`, and if another record with this combination of fields already exists, then the method that creates new records will throw this exception.

Another important point is that all exceptions that can be thrown by the methods of the interface must be created with the constructors indicated in the instructions document. A good approach is to create a hierarchy of exceptions, where the exceptions to be thrown extend another exception of a higher level. An alternative is to create an exception called `DatabaseException`, for instance, that directly extends `java.lang.Exception` and have the exception that can be thrown by the interface extend this exception. This eases the exceptions treatment because if, for some reason, a method has to throw exceptions that are subclass of `DatabaseException`, then this method can declare only `DatabaseException` in its throws clause.

One of the options is to create a Façade that implements the interface provided by Sun and delegates the tasks of data management and locking mechanism to other

```
package certification.database;
public class DatabaseException extends Exception {...}

package certification.database;
public class RecordNotFoundException extends DatabaseException {
    // Constructors indicated in the assignment must be defined...
}

package certification.database;
public class DuplicateExcursionException extends DatabaseException {
    // Constructors indicated in the assignment must be defined...
}
```

Figure 4. Definition of exceptions that can be thrown by the interface.

classes. In order to keep things simple, the implementation of the DB interface in the example proposed by this paper is a class that handles both data management and locking mechanism. There are two ways that can allow the data access to be done: to read the .db file at each request from the presentation layer or to keep all records in a memory cache. If the candidate opts to read the file at each request, then the candidate will have to guarantee the integrity of the file, in order to avoid the records to get messed up. The easiest approach is to read the .db file and put all records in a memory cache when the application is started. Some candidates say that this may consume too much memory if the database's size increases. But the candidate may argue that, besides being the easiest approach, the database's size is not big, and it is not informed whether the database's size may increase, and thus the candidate may assume that this choice will not be a problem.

When opting to store the database records in a memory cache, the candidate will have to decide when the records will be stored in memory and when they will be written back to the file. One option is to have a structure in the class, such as a `Map<Long, Excursion>` that acts as cache, and fill this structure with the database records in the class constructor. Although it was defined in this example a primary key, the methods defined in Sun's interface use record numbers as parameters, and thus it is easier to retrieve a record by its number. This way, given a number, it is easy to retrieve the corresponding record, via get method, defined in the `java.util.Map` class.

In order to write the records back to the file, one option is to define a new method to be invoked when the application finishes. The problem is that this method is not part of the interface provided by Sun **(which must not be altered)**, and thus it is not possible to access it when the interface is used as static type. It would be possible to invoke this method when using the implementation as static type, but one must program to interfaces as much as possible. The solution is to create a new interface that extends Sun's interface and add this method (and other methods that the candidate may judge necessary) to this new interface. This way, Sun's interface is implemented and the part of its limitations is overcome. It is shown in Figure 5 how the interface provided by Sun can be extended and in Figure 6 how the new interface can be implemented. A good tip is to start writing the JavaDoc comments along with the code, so that all this work is not left to be done after the end of the development.

The class that implements the interface provided by Sun must be Thread-safe. The candidate can then use the classical synchronization, always synchronizing on the cache object (that is, the `Map<Long, Excursion>` object) when performing operations in-

```
package certification.database;

public interface LocalDB extends DB {
    public void saveData() throws DatabaseException;

    // Other methods…
}
```

Figure 5. Extending the interface provided by Sun.

```
package certification.database;

// imports omitted...

public class Database implements LocalDB {

    /** Memory cache */
    private final Map<Long, Excursion> data = new HashMap<Long, Excursion>();

    /**
     * Database path, used to write the records back to the file
     * when the application finishes
     */
    private String dbPath;

    public Database(String dbPath) throws DatabaseException {
        super();

        // Reads the given database file and fills the memory cache with its content,
        // possibly throwing DatabaseException
        readData(dbPath);
        this.dbPath = dbPath;
    }

    @Override
    public synchronized void createExcursion(Excursion excursion)
            throws DuplicateExcursionException {
        // Puts the excursion record in the memory cache if
        // a record with the same primary key still does not exist...
    }

    // Implementation of other methods omitted...
}
```

Figure 6. Implementation of the data access class.

side the `Data` class. Of course, the `java.util.concurrent` API can also be used. Since some problems may occur (i.e. the database file, indicated when the application was started, may have been deleted), the `saveData` method, defined in the interface that extends Sun's interface can throw `DatabaseException` (the root class in the hierarchy of exceptions that can occur when performing operations with the database file).

Many candidates opt to implement the `Data` class as a Singleton [Gamma et al. 1994] and mark all methods as synchronized. In order to guarantee the integrity of the database, there must be some synchronization somewhere in the code. In a Java application, it is not possible to synchronize on a file, but it is possible to synchronize on an object. If this object is unique in the application, then it is like the synchronization is on the file. Although it is an easy approach, the Singleton is not scalable: if the application runs on more than one VM, then it is not unique anymore. Besides that, the Singleton is often applied wrongly, acting more like a global variable in the object-oriented world.

Therefore, one possible way to implement the `Data` class is to make it not a Singleton and instantiate it only when the application is started. This object, although not a Singleton so to speak, will be unique if instantiated only when the application starts and used throughout the application. This can be achieved through Dependency Injection. If the `Data` class methods are not marked as synchronized, the code that uses the `Data` object must always synchronize on it or the synchronization can be done inside the `Data` class on the objects that compose its state (such as the `java.io.RandomAccessFile` object, if one is used to read and write to the .db file) in order to guarantee its integrity. The candidate should then add JavaDoc comments to the `Data` class explaining how to use it and list these choices in the choices.txt file.

4.2. Second Step: Definition and Implementation of Business Methods

For the next step, the business methods that will be used by the presentation layer can be defined. The application must be executed in both client mode (that is, connected to a server, where information is retrieved from and sent to) and standalone mode (that is, information is retrieved from a local database, bypassing the server's code). In order to promote abstraction, the candidate can define an interface that can have two implementations, where one of them deals with local data and the other deals with remote data. **Important: in standalone mode, the server code must not be used, otherwise, the candidate will be automatically failed.**

At this point, it is important to define whether a thin or thick client will be implemented. The difference between these two concepts is quite simple. When using a thick client, the business logic is implemented in the client side, and when using a thin client, the business logic is implemented in the server side. For the OCMJD certification, the server code may be implemented with RMI or Sockets. **Important: if the candidate opts to use RMI, will have to use only RMI, or if the candidate opts to use Sockets, will have to use only Sockets. Also, if the candidate opts to use RMI, will have to attend the restrictions described in the instructions document, otherwise the candidate will be automatically failed. In order to keep things simple, the example proposed by this paper uses RMI.**

An advantage of the thin client is that the API used in the client side is simpler, as well as the server's code. In the case of the thick client, in order to correctly implement

the business rules on the client side, the server's API normally has more methods than when using a thin client. It is also necessary to control transactions on the client side, since they are always started and finished on the client side. That means that, if the client crashes between the start and end of the transaction, it will never be finalized. In the case of the thin client, the locking mechanism also becomes easier. In order to keep things simple, the example proposed by this paper implements a thin client.

Since the business layer will be implemented as a set of transaction scripts, it important to identify them at this point. There are some techniques that can be used to identify them in a real scenario. One of them is to analyze use cases, because transaction scripts generally correspond to use case steps. Another one is to verify the features that must be offered by the UI and define one transaction script for each possible request. The assignment itself does not contemplate use cases, but provides all features that must be offered on the project's main window. Therefore, the candidate can analyze these features and define one method in the business layer to handle each feature.

For instance, let us consider that the requirements and features that must be offered in the UI are:

- It must be composed only with Swing components.
- It must display all records in a JTable.
- It must allow to create new excursions.
- It must allow to delete existing excursions.
- It must display data of a particular excursion, given the bus number, the name of the company and a departure date.

In the real assignment, it is not necessary to create or delete records, and it is also not necessary to implement a feature that displays data of a particular record, and thus are examples of features that do not have to be implemented, but since this papers aims to instruct its readers to successfully solve the assignment, these are the features that are shown as examples.

Analyzing the features to be offered by the UI, it is possible to identify one method for creating records, another one for deleting records, another one for retrieving a particular record, based on a given number, and another one for retrieving all records from the database, in order to fill the main window's JTable. Besides these methods, since the records will be kept in a record cache while the application is executed, another method, that writes the records back to the database when the application is finalized, will also have to be defined. These methods will also have to be available remotely (in the example proposed by this paper, through an RMI server), and thus it is also necessary to include `java.rmi.RemoteException` in their throws clause. Therefore, it is possible to define the business component's API as shown in Figure 7.

As mentioned before, the methods defined in the interface provided by Sun do not include objects like `Excursion`, as shown here, but String arrays. In order to ease the API usage, it is possible to define a class that will be a DTO, like the `Excursion` class, in order to represent the String arrays, and this class can reside in the business package, since this DTO will be a domain object. For transforming from String array to domain object and vice-versa, an Adapter [Gamma et al. 1994] can be defined with two methods, where one transforms from String array to domain object and the other one does the opposite.

```
package certification.business;

// imports omitted...

public interface ExcursionServices {

    public void createExcursion(Excursion excursion) throws RemoteException;

    public void deleteExcursion(long recordNumber) throws RemoteException,
            ExcursionNotFoundException;

    public Excursion readExcursion(ExcursionPrimaryKey key)
            throws RemoteException, ExcursionNotFoundException;

    public Map<Long, Excursion> getAllRecords() throws RemoteException;

    public void saveData() throws RemoteException, ServicesException;

}
```

Figure 7. Definition of the business component's API.

Unlike the methods defined in the class that implements the interface provided by Sun, each method of the business component's API throws `ServicesException`. This exception is the root exception of the exceptions that can be thrown in the business layer, and thus, each exception to be defined in the business layer extends `ServicesException`. A good approach is to keep the appropriate level of abstraction in each layer. For instance, while a method from the persistence layer throws `RecordNotFoundException`, another method in the business layer throws `ExcursionNotFoundException`. Since the methods that can throw exceptions in the business layer include `ServicesException` in their throws clause, it is possible to throw every necessary exception without including each exception in the method's throws clause, as long as these exceptions extend `ServicesException`.

The interface defined in Figure 7 will have two implementations: one for the client mode and another one for the standalone mode. In the standalone mode, the presentation layer will access directly the implementation of this interface, and this implementation will use the data access class. In the client mode, the presentation layer will access the server (the implementation of this interface will be the server itself), and the server will use the data access class. Since the domain objects will travel from will VM to another through the network, these objects must implement java.io.Serializable. For the interface defined in Figure 7, classes `Excursion` and `ExcursionPrimaryKey` must implement java.io.Serializable.

The more abstraction applied, the easier it is to maintain and evolve an object-oriented system. To define independent APIs (that is, without worrying with the APIs that will use them or with the dependencies a component may depend on) also promotes flexibility. As shown in Figure 1, the presentation layer accesses the business layer through the interface defined in Figure 7, and thus it will not know whether the data is local or remote. In order for this approach to work correctly, it is necessary to build the code that arranges the objects, connecting them properly. For instance, one can build the code that initializes the proper implementation of the business component and provides it to the presentation layer. As mentioned before, it is a command line parameter that defines

how the application should be started. Therefore, it is possible to verify this parameter when starting the application, initialize the proper business component and provide it to the presentation layer.

```
package certification.business;

// imports omitted...

public class DefaultExcursionServices implements ExcursionServices {

    private LocalDB dbManager;

    public DefaultExcursionServices(LocalDB dbManager) {
        super();
        this.dbManager = dbManager;
    }

    @Override
    public void saveData() throws ServicesException {
        try {
            dbManager.saveData();
        } catch (DatabaseException exception) {
            String message = "It was not possible to save the data "
                    + "back to the file due to the following reason: "
                    + exception.getMessage();
            throw new ServicesException(message, exception);
        }
    }

    @Override
    public void createExcursion(Excursion excursion) {
        // Creates a new Excursion record...
    }

    // Implementation of other methods omitted...
}
```

Figure 8. Default implementation of the business layer component.

Since the remote services will be exposed via RMI, an interface that extends java.rmi.Remote must be defined, and the implementation of this interface will be the server so to speak. All methods of an interface that extends java.rmi.Remote must include in their throws clause `java.rmi.RemoteException`. Since the methods of the `ExcursionServices` interface already include this exception in their throws clause, it is possible to define a new interface that extends both `certification.business.ExcursionServices` and `java.rmi.Remote`. This new interface can also have a method responsible for starting the server that will be invoked when the application is started in server mode. This method can receive an int as parameter, which will correspond to the port in which the server will be started. In order to keep things simple, the example shown in this paper uses a `ShutdownHook`, but the server could also have a window that could graphically allow the user to start and stop the server.

One point to be observed in the business components implementation proposed by this paper is that, as shown in Figure 1, the business layer intermediates the communication between the presentation layer and the persistence layer. This way, the constructors of the business components receive as parameter the persistence layer interface defined in Figure 5, which constitutes constructor dependency injection. In this case, the busi-

```
package certification.business;

// Imports omitted...

public interface RemoteExcursionServices
        extends Remote, ExcursionServices {

    public abstract void startServer(int port) throws RemoteException;
}
```

Figure 9. Definition of the server's API.

```
package certification.business;

// imports omitted...

public class RMIRemoteExcursionServices extends DefaultExcursionServices
            implements RemoteExcursionServices {

    public static final String SERVER_NAME = "ExcursionServer";

    public RMIRemoteExcursionServices(LocalDB dbManager) {
        super(dbManager);
        if (dbManager == null) {
            String message = "The LocalDB object cannot be null.";
            throw new IllegalArgumentException(message);
        }
    }

    @Override
    public void startServer(int port) throws RemoteException {
        RemoteExcursionServices servidor = (RemoteExcursionServices) UnicastRemoteObject
                .exportObject(this, 0);
        Registry registry = LocateRegistry.createRegistry(port);
        registry.rebind(SERVER_NAME, servidor);

        Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                final String message = "Saving database changes, please "
                        + "wait...";
                System.out.println(message);
                try {
                    saveData();
                } catch (ServicesException exception) {
                    final String exceptionMessage = "The database file could "
                            + "not be updated with the latest changes due to the "
                            + "following reason: " + exception.getMessage()
                            + ". Please contact the system administrator for "
                            + "assistence.";
                    System.err.println(exceptionMessage);
                }
            }
        });

        String message = "Server running on port #" + port
                + "\nPress Ctrl + C to exit...";
        System.out.println(message);
    }
}
```

Figure 10. RMI server.

ness components depend only on one object, and defining these constructors eases their utilization, because as soon as they are created, they are ready to be used, what automatically exempts the business component from the task of instantiating the persistence component. Besides that, defining these constructors eases the utilization of these components in frameworks that act as IoC Containers, such as Spring, for instance. For the real assignment, the objects arrangement can be done when the application is initialized, according to the mode the application was run.

After implementing the data access component and the business components, the candidate is ready for the next step, which is the implementation of the application windows so to speak.

4.3. Third Step: Implementation of the Application Windows

After developing the code that resides in the lower layers, the candidate can then develop the code related to the presentation layer. In the real assignment, it is given all requirements, elements and features that must be offered in the application's main window. At the moment the application is run, it must be presented to the user a window where he/she can enter the properties according to the mode the application was run. For instance, when running the application in server mode, the user can provide the database file path and the port in which the server must be started. In standalone mode, the user can provide the database file path, and in client mode, the user can provide the server's IP address and the port where the remote services are available on the server.

The data provided in the configuration dialogs must be saved in a .properties file that must be created in the user's hard disk in a specific place and with a specific name, both indicated in the instructions. When the application is executed for the first time, the file will not naturally exist, and therefore, the file must be created with the configuration provided by the user. After that, when the application is executed in a particular mode, the fields where the user provides configuration data must be populated with the information previously provided. **Important: if the fields present in the configuration dialogs are not persisted between runs of the program, the candidate will automatically be failed.**

The only thing that varies between client mode and standalone mode is the data source. When executing the application in client mode, data is provided by the server (in the example proposed by this paper, by `RMIRemoteExcursionServices`, shown in Figure 10), and in this case, the server must mandatorily be up so data can be retrieved. In standalone mode, data is provided locally, and thus the class `DefaultExcursionServices`, shown in Figure 8, can be used. The components used in the configurations dialogs are responsible for injecting an instance of the business class in the main window's object, so everything works correctly.

In both cases, the business components obey the `ExcursionServices` interface's API. Therefore, it is possible to create an abstract class called `OKButtonListener` that implements `java.awt.event.ActionListener`. This class will have two subclasses: one for the standalone launch dialog and another one for the client launch dialog. It can have an abstract method called `getServices` that returns an implementation of `ExcursionServices`. One of the subclasses will instantiate `RMIRemoteExcursionServices` and will listen to the OK but-

ton of the configuration dialog in client mode, and the other subclass will instantiate `DefaultExcursionServices` and will listen to the OK button of the configuration dialog in standalone mode, what characterizes the Factory Method. Since the data that is provided by the user has to be saved in a .properties file, it is also possible to define an abstract method to execute this task, what makes the `actionPerformed` method a Template Method [Gamma et al. 1994].

Since there might be problems when instantiating the `Data` object and the `ExcursionServices` object, the `getServices` method can throw `LaunchException`, which is an exception that directly extends `Exception`. In case of a problem, the `getServices` method can throw `LaunchException` with a descriptive message of what happened when starting the application, and then, this message can be displayed to the user, as shown in Figure 11.

Ideally, the messages that are displayed on the user's window can be internationalized by using a messages bundle, but, in order to keep this paper simple, the messages that are displayed in this example are hardcoded. If the properties cannot be saved, a message can be displayed to the user, but, in this case, it is not necessary to stop the execution of the application, unlike when, for instance, it is not possible to get a reference to the remote server.

As shown in Figure 11, in order for this design to work, it is necessary to create a hierarchy of interfaces that are implemented by the launch dialogs. The `LaunchManager` interface is the root interface of this hierarchy and has a method called `closeWindow` that closes the dialog being displayed.

In order to access the data provided by the user, the classes that extend the abstract class defined in Figure 11 can receive the classes that represent the dialog windows in their constructors, and these classes can provide methods that return the data provided. After verifying that everything is correct, these classes can correctly instantiate the business components and provide them to the class that represents the main window.

In the real assignment, the main window must display the database records in a `JTable`, must allow users to search for all records or for records that match a particular search criteria and must also allow the user to book a selected record, updating the database properly. For the booking requirement, the locking mechanism must mandatorily be used, in order to guarantee that the data provided by a user do not inappropriately override data provided by another user.

The window to be built in the assignment must be as simple and easy to use as possible. For instance, one can define shortcut keys for the utilization of menu items or buttons. Other than that, in the user interface section of the instructions document, there is a statement that says that the window must be built expecting future enhancements. This is a hint that layout managers that ease the introduction of new UI components, like `BorderLayout`, `FlowLayout` or `GridBagLayout`, should be used. If the candidate judges necessary, he/she can block the insertion of invalid characters in specific fields (such as letters in numerical fields), use masks in text fields or disable buttons if the window is in a particular state (for instance, disable the search button while the search criteria is not filled).

In the example proposed by this paper, the application must create and delete

```
package certification.ui.launch;

// imports omitted...

public abstract class OKButtonListener implements ActionListener {

    /**
     * The LaunchManager interface is the root interface of the interfaces
     * that are implemented by the launch dialogs. It will be either a
     * StandaloneLaunchManager object (which is an interface that extends
     * LaunchManager and is implemented by StandaloneLaunchDialog) or a
     * ClientLaunchManager object (which is another interface that extends LaunchManager
     * and is implemented by ClientLaunchDialog)
     */
    private LaunchManager manager;

    /** Returns an implementation of ExcursionServices */
    protected abstract ExcursionServices getServices() throws LaunchException;

    /** Saves the data provided by the user in a .properties file */
    protected abstract void saveProperties() throws IOException;

    protected OKButtonListener(LaunchManager manager) {
        super();
        this.manager = manager;
    }

    @Override
    public void actionPerformed(ActionEvent event) {
        try {
            ExcursionServices services = getServices();
            manager.closeWindow();
            new MainWindow(services);
        } catch (LaunchException exception) {
            String message = "The application could not be started for the "
                    + "following reason: "
                    + exception.getMessage();
            JOptionPane.showMessageDialog(null, message, "Excursion System",
                    JOptionPane.ERROR_MESSAGE);
        }

        try {
            saveProperties();
        } catch (IOException exception) {
            String message = "The properties could not be saved.";
            JOptionPane.showMessageDialog(null, message, "Excursion System",
                    JOptionPane.ERROR_MESSAGE);
        }
    }

    protected LaunchManager getManager() {
        return manager;
    }
}
```

Figure 11. Abstract listener to be implemented by the listener that listen to the OK buttons in client and standalone mode.

```
package certification.ui.launch;

// imports omitted...

public class StandaloneOKButtonListener extends OKButtonListener {

    public StandaloneOKButtonListener(StandaloneLaunchManager manager) {
        super(manager);
    }

    @Override
    protected ExcursionServices getServices() throws LaunchException {
        StandaloneLaunchManager manager = (StandaloneLaunchManager) getManager();
        String dbLocation = manager.getDbLocation();
        LocalDB db;
        try {
            db = new Database(dbLocation);
        } catch (DatabaseException exception) {
            String message = exception.getMessage();
            throw new LaunchException(message, exception);
        }
        return new DefaultExcursionServices(db);
    }

    @Override
    protected void saveProperties() throws IOException {
        StandaloneLaunchManager manager = (StandaloneLaunchManager) getManager();
        String dbLocation = manager.getDbLocation();
        // Saves the property in a .properties file...
    }
}
```

Figure 12. Listener that listens to the OK button in standalone mode.

records, and read a particular record, given the bus number, the company name and the departure date. Therefore, the main window can have the layout shown in Figure 13.

At the top of the window, it is located the feature that displays data of a particular excursion, that utilizes the `readExcursion` method of the business component. The database records are displayed in the `JTable`. Its `TableModel` can be built so that it only controls the records being displayed. The features that create and delete records are provided below the JTable, via New Excursion and Delete Excursion buttons, and respectively use the `createExcursion` and `deleteExcurion` methods of the business component.

In the implementation of the main window class proposed by this paper, there must be a constructor that receives an `ExcursionServices` object, in order to promote polymorphism. This way, the presentation layer will execute its tasks without knowing whether the manipulated data is local or remote. This way, if there is another data source in the future, it is only necessary to create another implementation of the `ExcursionServices` interface, shown in Figure 7, and provide it to the main window. The services implementation is passed to the `ActionListeners` that listen to the New Excursion and Delete Excursion buttons, so the business methods can be consumed inside these listeners.

The `TableModel` can be built inside the constructor of the main window class and only store the data being displayed, as shown in Figure 13. It can also have a method that returns a record, given a number (which will be the row selected by the user) and

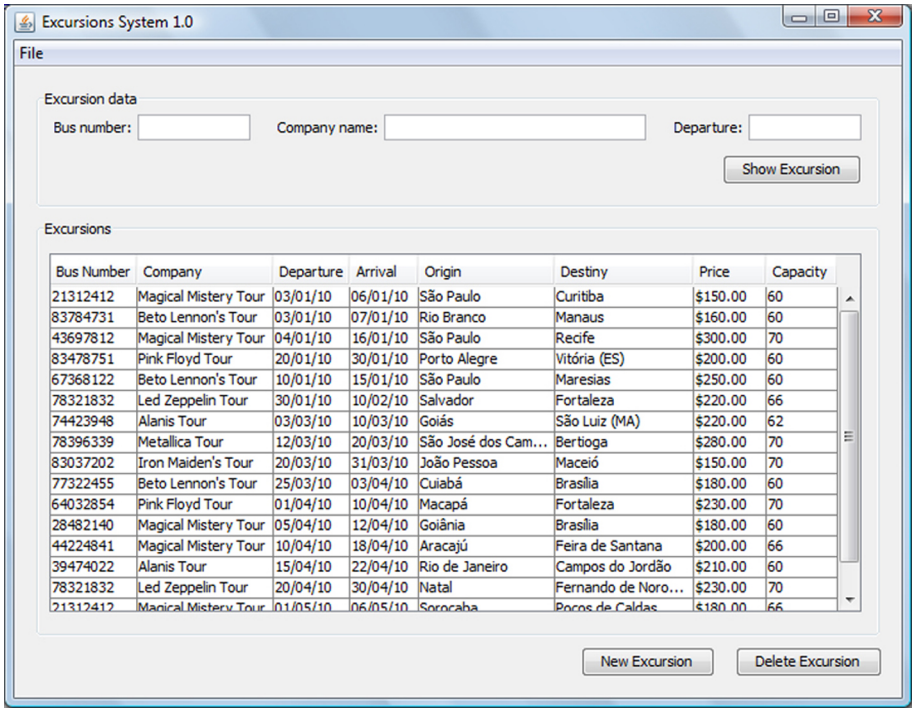

Figure 13. Main window's layout.

other methods the candidate may judge necessary. The main window class can have a method called `updateData` that can receive the data to be displayed in the `JTable` and then update the `TableModel`, which can be customized in a way that it extends `AbstractTableModel` and adds a method called `updateData`, receiving the new data to be displayed in the `JTable`, as shown in Figure 14.

For the `ActionListeners`, it is also necessary to pass a reference to the main window, so the data provided by the user can be retrieved and the services can be invoked properly. Each constructor can be defined according to the tasks it must execute. As an example, the `ActionListener` that listens to the Show Excursion button is shown in Figure 16.

This way, the business component is passed to the listeners that listen to the main window's buttons, so they can execute their tasks correctly. For the `JTable`, it can be created a class that extends `javax.swing.table.AbstractTableModel` and defines methods that ease that controlling of table model's state and that retrieve specific records.

4.4. Forth Step: Packaging the Application

After finishing the implementation, the application must be packaged into a single JAR file. Although each execution mode uses some specific classes of the project, all classes must be packaged into the same JAR file. For instance, in the example proposed by this paper, only the persistence interfaces and the `Database` class, the interfaces and the remote business component are used, and thus, only these classes should be required for the server mode. Normally, this file must be named runme.jar.

Other than that, the candidate must also create a `MANIFEST.MF` file and put it in

```
package certification.ui;

// imports omitted...

public class MainWindow extends JFrame {

    /** The JTextField where the user provides the bus number */
    private JTextField busNumber;

    /** The custom TableModel */
    private AbstractExcursionsTableModel tableModel;

    /** The JTable that renders the data */
    private JTable table;

    // Other fields omitted...

    public MainWindow(ExcursionServices services) {
        super();

        // The construction of the window is done here...
        JButton showExcursion = new JButton("Show Excursion");
        ActionListener showExcursionListener = new ShowExcursionListener(this,
                services);
        showExcursion.addActionListener(showExcursionListener);

        Map<Long, Excursion> records = services.getAllRecords();
        // The table columns can be retrieved from the messages bundle
        String [] columns = ...;
        tableModel = new ExcursionsTableModel(columns, records);
        table = new JTable(tableModel);
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
        JScrollPane pane = new JScrollPane(table);
        // More code here...
        setVisible(true);
    }

    public String getBusNumber() {
        return busNumber.getText();
    }

    public void updateData(Map<Long, Excursion> records) {
        tableModel.updateData(records);
    }
}
```

Figure 14. Main window's class partially implemented.

```
package certification.ui;

// imports omitted...

// The AbstractExcursionsTableModel class extends AbstractTableModel and
// defines two methods: one that returns the Excursion object, given the
// number of the selected row, and another one that updates the data being displayed
public class ExcursionsTableModel extends AbstractExcursionsTableModel {

    private String [] columns;
    private Map<Long, Excursion> records;

    public ExcursionsTableModel(String [] columns, Map<Long, Excursion> records) {
        super();
        this.columns = columns;
        this.records = records;
    }

    @Override
    public Excursion getExcursion(int row) {
        if (row == -1) {
            return null;
        }

        return (Excursion) records.values().toArray()[row];
    }

    @Override
    public void updateData(Map<Long, Excursion> records) {
        this.records = records;
        fireTableDataChanged();
    }

    // Implementation of other methods omitted...
}
```

Figure 15. Main window's table model.

```
package certification.ui;

// imports omitted...

public class ShowExcursionListener implements ActionListener {

    private MainWindow mainWindow;
    private ExcursionServices services;

    public ShowExcursionListener(MainWindow mainWindow,
            ExcursionServices services) {
        super();
        this.mainWindow = mainWindow;
        this.services = services;
    }

    @Override
    public void actionPerformed(ActionEvent event) {
        // Assume that the data provided by the user
        // was previously validated...

        String busNumber = mainWindow.getBusNumber();
        String companyName = mainWindow.getCompanyName();
        String departureString = mainWindow.getDeparture();
        Date departure = convertToDate(departureString);

        ExcursionPrimaryKey key = new ExcursionPrimaryKey();
        key.setBusNumber(Integer.parseInt(busNumber));
        key.setCompany(companyName);
        key.setDeparture(departure);

        try {
            Excursion excursion = services.readExcursion(key);
            // Opens a window that displays the data of the
            // Excursion that corresponds to the data provided
            // by the user
            showExcursion(excursion);
        } catch (ServicesException exception) {
            String message = "No excursions with the given parameters could "
                    + "be found.";
            JOptionPane.showMessageDialog(null, message, "Excursions System",
                    JOptionPane.INFORMATION_MESSAGE);
        } catch (RemoteException exception) {
            String message = "A problem ocurred while trying to communicate "
                    + "with the remote server.";
            JOptionPane.showMessageDialog(null, message, "Excursions System",
                    JOptionPane.INFORMATION_MESSAGE);
        }
    }

    private void showExcursion(Excursion excursion) {
        // Opens a dialog with the Excursion's information
    }

    private Date convertToDate(String departureString) {
        // Converts the data provided by the user to a Date object...
        return null;
    }
}
```

Figure 16. Listener of the Show Excursion button.

the `META-INF` directory, in the root of the JAR file, indicating the class that contains the main method, invoked when the application is executed. The main method must verify the argument that indicates how the application should be executed, passed in the command line, and open the appropriate configuration dialog. **Important: the only argument that is permitted in the command line is the mode that indicates how the application should be executed. If the application requires other arguments, then the candidate will automatically be failed.**

At this point, all components are implemented. It is shown in Figure 17 a high level class diagram that shows the organization of the main components implemented in this paper. Obviously, the real application will contain many other classes, but the base design of the application may look like the design shown in Figure 17.

5. Execution Modes

It should be possible to execute the application in three different modes:

- Client: in this mode, data is retrieved from a server. Obviously, in order to establish communication, the server must be up when the client is started. This mode does not require arguments in the command line and can be executed as follows:
  `java -jar runme.jar`
- Server: in this mode, a server (either RMI or Sockets) should be started, in a way to provide the implemented services to its clients. If the client is a thin client, the server will be responsible for implementing and providing methods to be consumed by the presentation layer (as shown in Figure 10), or, if the client is a thick client, the server will be responsible for providing an API similar to the interface provided by Sun so business rules can be managed on the client side. This mode requires the argument "server" to be passed in the command line and can be executed as follows:
  `java -jar runme.jar server`
- Standalone: in this mode, data is retrieved locally, bypassing the server's code. For instance, in the example proposed by this paper, the presentation layer deals with an object of the `DefaultExcursionServices` class, instantiated when the application is started. This mode requires the argument "alone" to be passed in the command line and can be executed as follows:
  `java -jar runme.jar alone`

Taking into consideration the example proposed by this paper, the execution modes can be represented as shown in Figure 18.

6. Project Overview

After packaging the application, generating the JavaDoc of all classes, generating the user guide, creating the version.txt file, creating the choices.txt files with all significant choices made during the development, the candidate must gather all these artifacts in one single file, whose name must be `scjda-xx9999999.XXX`, where `xx9999999` is the candidate's Prometric ID, and `XXX` is either ZIP or JAR.

It is fundamental that the candidate pays careful attention to the format and content of the file. The final file must have the following format:

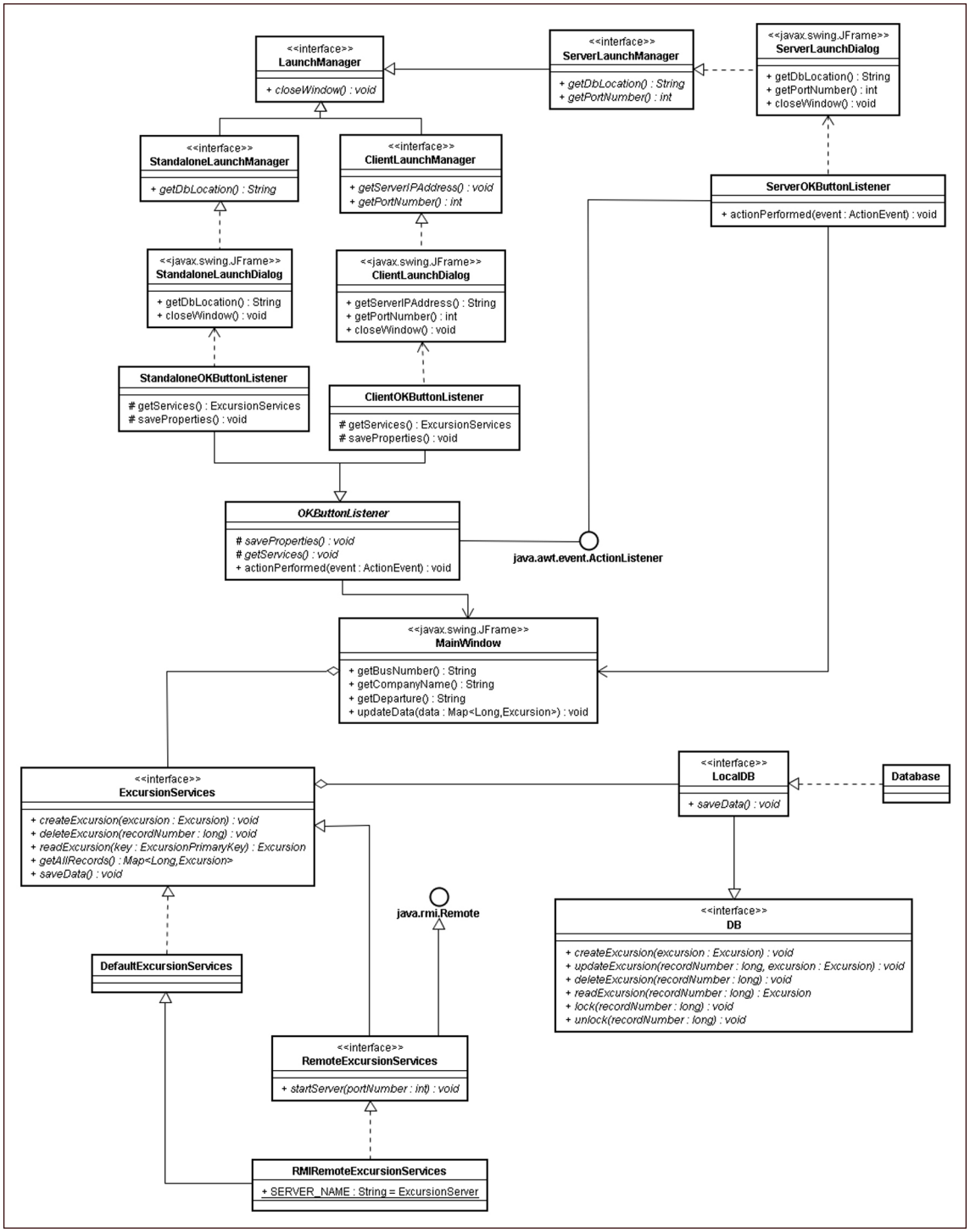

Figure 17. Overall design.

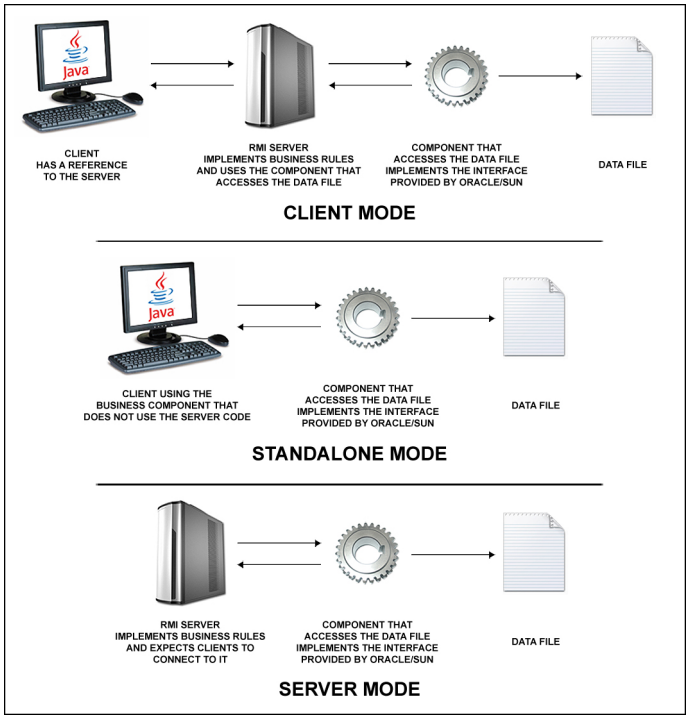

Figure 18. Communication between components in each execution mode.

- The runme.jar file.
- The .db file, received along with the instructions (without any modifications).
- A directory called code, containing the source-code of the application (respecting the packages structure created).
- A file called version.txt, indicating the version of the JDK used and the operating system used for developing the project.
- A directory called docs with the following format:

  The instructions.html file, provided by Sun.

  A subdirectory called javadoc, with the JavaDoc documentation generated by the candidate.

  A file called choices.txt, with the decisions made by the candidate.

  User guide. If this guide is within the application, this file may be omitted. Otherwise, it must be called userguide.txt or may be multiple HTML files, where the home page must be named userguide.html.

7. The Locking Mechanism

Each aspect of this certification is worth an amount of points that can be achieved by the candidate. The hardest part and the one that is worth the biggest amount of points is the locking mechanism, which must guarantee that the data provided by a client do not inappropriately override the data provided by another client. For instance, in the real assignment, the customer field of a given record must be updated with the number of a particular customer if that field is blank. If two clients try to update the same record at the same time, only one must succeed, and the other one must receive a message, informing that the record is no longer available. Therefore, the locking mechanism must guarantee that when a client locks a record, only this client can update or delete this record, in order to avoid the following situation:

- Client 1 verifies that record 1 is available.
- Client 2 verifies that record 1 is available.
- Client 1 updates the record.
- Client 2 updates the record, overriding the data provided by client 1 inappropriately.

With the locking mechanism, a given record must be protected when two clients try to update or delete it at the same time:

- Client 1 locks record 1.
- Client 2 tries to lock record 1, but since it is already locked, waits.
- Client 1 verifies that record 1 is available, updates it, unlocks it and notifies all waiting clients.
- Client 2 wakes up and locks record 1.
- Client 2 verifies that record 1 is no longer available, does not update the record, unlocks it and notifies all waiting clients that may be waiting.

The locking mechanism can also be implemented with the locking classes of the Java Concurrent API (located in the `java.util.concurrent` package). As stated earlier, it is not necessary to offer deletion of records as a feature, but the method of deletion of records, provided in Sun's interface, must be implemented and must verify if the client deleting the record is the same who locked it. Besides that, the locking mechanism must also be used in standalone mode, even if theoretically there is only one Thread running.

The locking mechanism must also guarantee that, if a given record is already locked, then other Threads interested in that record must not consume CPU cycles and must wait for the record to be released. For this requirement, the wait method can be invoked within a while loop. The class that implements the interface provided by Sun can have a `Map<Long, Long>` to control the locked records, where the key is the record number and the value is a number that identifies the client who locked the record. It is possible to have an idea of how the locking mechanism can be implemented by looking at Figure 19.

In the real assignment, the record must be locked before updating it. Therefore, in the business method that updates the record, the record can be locked and it can be verified whether it is still available for an update (that is, if the customer field is still blank). If so, then the record is updated and unlocked. Otherwise, it can be thrown an exception that can extend `ServicesException` and be called `RoomAlreadyBookedException`. It is of high importance to unlock the record in both cases, otherwise the waiting Threads will never be notified and the deadlock will occur. Therefore, the sequence of methods invocation is lock-update-unlock. **Important: if the deadlock can occur in the application, the candidate will be automatically failed.**

In the JavaRanch's SCJD FAQ, it is possible to find a code implemented by the author of this paper that tests the `Data` class and the locking mechanism. This way, the candidate can use it and avoid being failed by possible deadlock occurrences (which is the biggest cause of failures in this certification).

```
package certification.database;

// imports omitted...

public class Database implements LocalDB {

    private final Map<Long, Long> lockedRecords = new HashMap<Long, Long>();

    @Override
    public synchronized void lock(long recordNumber)
            throws RecordNotFoundException {
        long clientId = Thread.currentThread().getId();

        // While the record to be locked is in the
        // lockedRecords Map, the current thread must wait...
        // the notification for the waiting Threads to wake up
        // must be done in the unlock() method, and a good tip
        // is to notify all Threads. After acquiring the record
        // lock, it must be verified if the record to be locked
        // still exists, because the method that deletes records
        // also requires the locking mechanism to be used.
        // In the update and delete methods, it must be verified
        // whether the client requesting the operation is the same
        // client who locked the record, and only this client will
        // be able to unlock it.
        lockedRecords.put(recordNumber, clientId);
    }
}
```

Figure 19. Tips of how to implement the locking mechanism.

```
package certification.business;

// imports omitted...

public class DefaultExcursionServices implements ExcursionServices {

    private LocalDB dbManager;

    public DefaultExcursionServices(LocalDB dbManager) {
        super();
        this.dbManager = dbManager;
    }

    @Override
    public void deleteExcursion(long recordNumber)
            throws ExcursionNotFoundException {
        try {
            dbManager.lock(recordNumber);
            dbManager.deleteExcursion(recordNumber);
            dbManager.unlock(recordNumber);
        } catch (RecordNotFoundException exception) {
            // If it occurs, then the exception was thrown by the lock method.
            // If the deleteExcursion method is invoked alone (that is, without
            // locking the record to be deleted first), then an
            // IllegalStateException can be thrown because it is necessary to lock
            // the record first. The same applies for the unlock method.
            String message = "This excursion has already being deleted.";
            throw new ExcursionNotFoundException(message, exception);
        }
    }
}
```

Figure 20. Code using the locking mechanism.

8. Conclusions

This paper shows what candidates can expect when opting for the SCJD certification. The goal is not to show an assignment solved (what would be against the rules of the certification), but how the candidate can approach the problem, model the application and resolve all steps through the resolution of a fictitious problem. Although it demands hard work, this certification is a great exercise regarding systems modeling and solutions for real world problems. It is also a good opportunity to improve skills in aspects like multi-threaded programming, Swing, design patterns and distributed programming, and even learning Sun's Java code conventions and how to correctly write JavaDoc comments.

A good tip is to try to solve what is asked the simplest possible way, and nothing more. Besides that, the candidate must pay careful attention to all "musts" stated in the instructions document. If one of these rules is not followed, the candidate will automatically be failed. In case of doubts, the candidate may look for answers in JavaRanch's SCJD forum. This certification is a great preparation for the SCEA's second phase, which requires the modeling of a system and demands some proficiency regarding systems modeling. After finishing solving the assignment, the candidate will realize that has considerably evolved as a developer and that all efforts were worth it!

References

Alur, D., Malks, D., Crupi, J., Booch, G., and Fowler, M. (2003). *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Sun Microsystems, Inc., Mountain View, CA, USA, 2 edition.

BluePrints, J. (2002). The mvc pattern. http://java.sun.com/blueprints/patterns/MVC-detailed.html.

Camerlengo, T. and Monkhouse, A. (2005). *SCJD Exam with J2SE 5, Second Edition*. Apress, Berkely, CA, USA.

Evans, E. (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fowler, M. (2002a). Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html.

Fowler, M. (2002b). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fowler, M. (2003). Anemic domain model. http://www.martinfowler.com/bliki/AnemicDomainModel.html.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA.

Panda, D., Rahman, R., and Lane, D. (2007). *Ejb 3 in Action*. Manning Publications Co., Greenwich, CT, USA.

PicoContainer (2011). The picocontainer framework. http://www.picocontainer.org.

Richardson, C. (2006). *POJOs in Action: Developing Enterprise Applications with Lightweight Frameworks*. Manning Publications Co., Greenwich, CT, USA.

Spring (2011). The spring framework. http://www.springsource.org/.

Appendix A – About the Author

Roberto Perillo is a SCJP, SCWCD, SCJD and SCBCD, lives in Brazil, and has been working with Java since 2005. He started programming COBOL at the age of 15 back in high school, and started working with web development in 2001 in a small company. Some years later, went to IBM, where he had the opportunity to co-found the ibm.com GWPS LA Innovation Team and work on several challenging projects. Still in IBM, won 2 prizes for developing and leading innovation projects. In 2007, moved to Avaya, where he also had the opportunity to work on several challenging projects. He currently works as Tech Lead and Senior Java Developer.

Roberto has bachelor's degree in Computer Science and has already finished a Software Engineering Specialization course at the Aeronautical Institute of Technology (ITA), where he is currently post-graduating in Electronic and Computer Engineering as well. He currently has 2 papers published (at RAM-SE/ECOOP 2009 and ACoM/OOPSLA 2009), and is also part of the technical team of the MundoJ magazine (a brazilian Java magazine) where he often publishes articles as well. He has been developing a framework called Daileon which enables domain annotations in Java applications.

In his free time, he likes to run, play basketball, swim, play the guitar (some of his neighbors say he plays just like Jimmy Page!) and listen to music (mostly The Beatles, Pink Floyd, Led Zeppelin, Alanis Morissette, Slayer, Metallica, Iron Maiden, Frank Sinatra and some brazilian stuff, like Tom Jobim, MPB-4 and bossa nova). Besides that, he's a huge fan of the Sao Paulo Futebol Clube soccer team.