Practical 4

Theme

– taking a one-JVM Java application and converting it to a client-server Java RMI application

Key concepts: remote method invocation, remote object, synchronisation, registry

4.1. Start up and essential configuration

a) *Log-in to Ubuntu and start terminal.*

b) *Download and extract source code archive* `lab4-standalone.zip` *from BlackBoard to your* `CS3250-DS-1011` *folder created in Practical 1.*

c) *Open the code in Eclipse.*

Start Eclipse, switch to Java perspective and select File > New > Java Project.

Untick the Use default location check box and click Browse to navigate to the location of the decompressed source code,
ie `CS3250-DS-1011/lab4-standalone`. Click Finish.

d) *Prepare to open BlackBoard assessment called* `Practical 4` *when required.*

Important: The quiz should be started within the **first 20 minutes** of the practical.

Notice that this is an assessed quiz that contributes to your module score.

4.2. Turning given code into a DS — Part I

a) *Examine the structure of the phone book application.*

The aim is to convert the local phone book application into a distributed version. Figure 1 shows a UML class diagram for the structure of the given simple local phone book application. Figure 2 shows a proposed distributed version of the same system.

b) *Copy the three classes from package* `standalone` *into package* `part1` *and correct the import of the* `PhoneType` *enumeration in class* `UsePhoneBook`.

c) *Create a remote* `PhoneBookInterface` *in package* `part1`.

This is a Java RMI remote interface indicating the methods that will be remotely available for instances of the `PhoneBook` class.

✎ **Quiz.** Answer and save **question 1**, which is related to this exercise.

d) *Ensure all methods in* `PhoneBookInterface` *have parameters suitable for RMI but do not edit* `PhoneBook.java`

Recall that all RMI parameters must be either remote objects or `Serializable`. In this case, you need to make all parameters `Serializable`.

✎ **Quiz.** Answer and save **question 2**, which is related to this exercise.

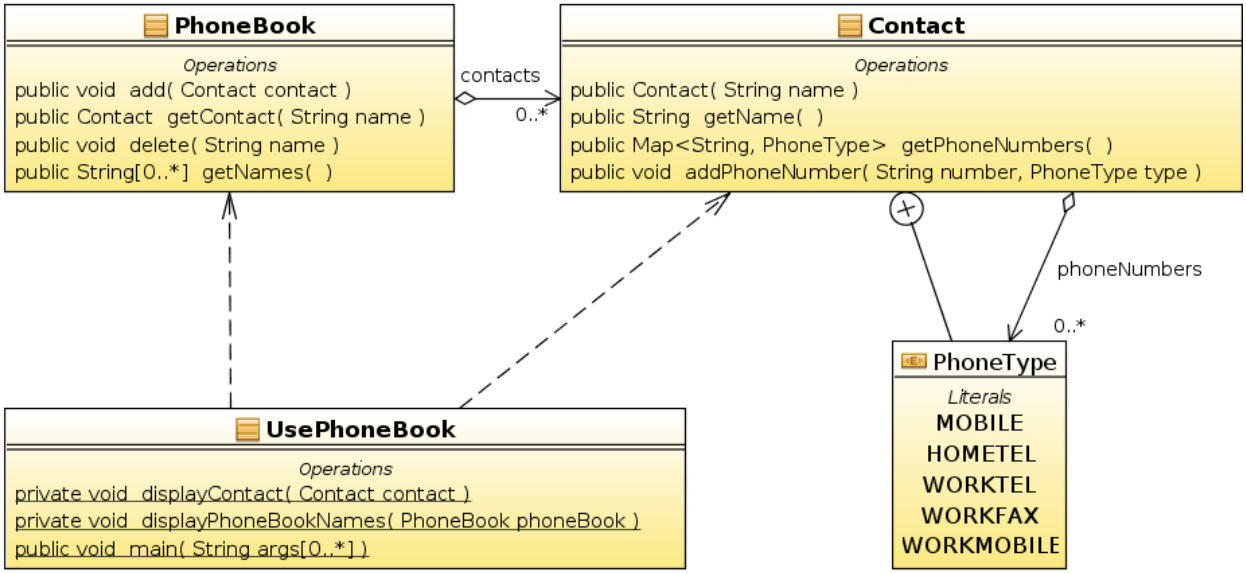

Figure 1: UML class diagram of local phone book.(Attributes have been omitted.)

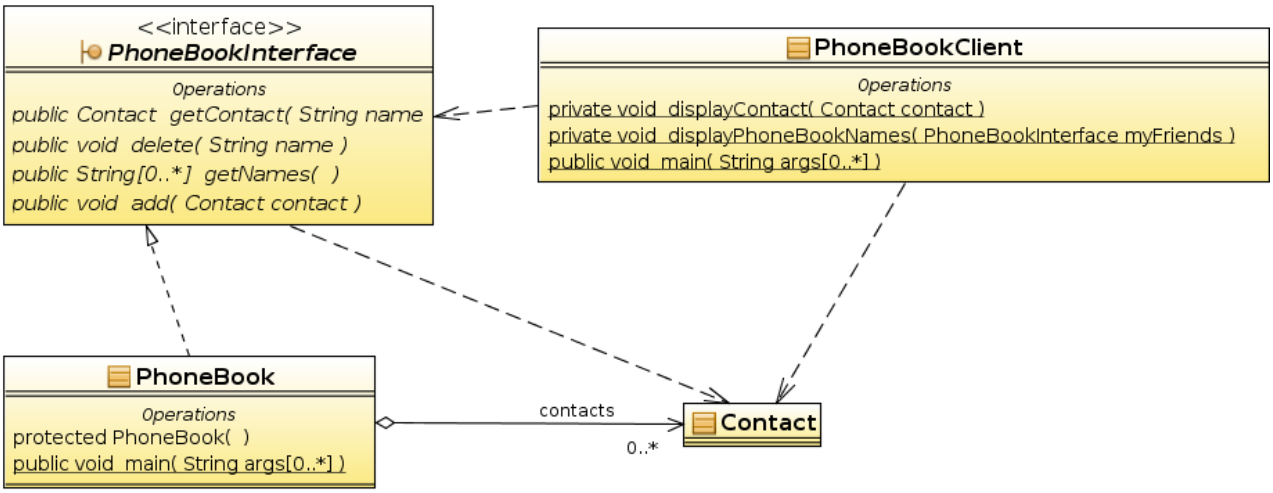

Figure 2: UML class diagram of distributed phone book in Part I.
(Attributes and some methods have been omitted.)

e) *Adjust the class* `part1.PhoneBook` *to make its instances remotely accessible.*

Recall that such a class has to implement a remote interface and extend `UnicastRemoteObject`. Also, you need to write a new constructor that invokes the parent class' constructor.

f) *Add a* `main` *method to class* `PhoneBook`.

The remote version of the phone book differs from the local version in that it is meant to be created in a separate JVM and so it must have its own `main` method. The `main` must:

– create and install a security manager;
– create a phone book instance;
– create an RMI registry;
– use the `Naming` class to bind the instance to a name in the registry.

✎ **Quiz.** Answer and save **question 3**, which is related to this exercise.

g) *Create a client for a remote phone book as class* `PhoneBookClient`.

Start by renaming the class `UsePhoneBook` (which you copied earlier from the `standalone` package) to `PhoneBookClient`.

The main changes are that the client application has to bind to a remote phone book object and handle remote exceptions.

> **Important.** Ensure that `PhoneBookClient` does **not use a locally created** phone book but instead accesses a remote phone book.

✎ **Quiz.** Answer and save **question 4** related to this exercise.

h) *Test the system.*

Execute the `PhoneBook` server and then execute the client application twice and observe its output.

You can execute the programs in terminals — using one terminal to start the server and another one to start the client using the provided scripts, eg `sh part1-RunServer.bat`. Use `Ctrl-C` if you need to stop the server.

> **Important.** To run the server via Eclipse, you need to first use Run Configurations, add or select the Java application PhoneBook and in the Arguments tab enter the following into the VM arguments entry box:
>
> `-Djava.security.policy=policy.all`
>
> When you start `PhoneBook` and it reports an exception, you still need to stop it. In the Eclipse Console view, you can switch between consoles for different Java programs you executed using the icon.

✎ **Quiz.** Answer and save **questions 5–7**, which are related to Part I.

4.3. (Optional) Turning given code into a DS — Part II

The aim of this part is to further modify the client-server system developed in Part I. In Part I the `Contact` objects were passed there and back between the server and the client. Here we want to pass remote references to remote `Contact` objects instead.

(Note that the changes in this part do not improve the system—quite the opposite; nevertheless, it is a good exercise in using Java RMI.)

a) *Copy all Java classes and interface from package* `part1` *into package* `part2`*, correct the import of* `PhoneType` *as before and work in package* `part2` *in the following steps.*

b) *Create a* `ContactInterface` *remote interface.*

c) *Amend class* `Contact`*.*

The `Contact` class now needs to extend `UnicastRemoteObject` as well as implement `ContactInterface`.

d) *Amend* `PhoneBook` *and* `PhoneBookInterface`*.*

Since the client will no longer handle the `Contact` objects directly, the `add` method will need to be replaced with the method `addContact` that takes as parameters the various components of a contact required by its constructor. This method will add a `Contact` object to the `PhoneBook` server. References to remote `Contact` objects can be obtained by calling `getContact(name)` and used to operate on the contacts remotely.

e) *Amend* `PhoneBookClient` *to correctly use the modified* `PhoneBookInterface`*.*

Ensure there is no reference to class `Contact` within `PhoneBookClient`.

f) *Test the system.*

Run the server and then twice the client and observe the output. It should be exactly as before.