CHAPTER 4

PROTOTYPE DESIGN

4.1 Introduction

This chapter discusses the design of our prototype. The use case presents the general idea on the prototype’s GUI design. The prototype architecture is represented by package diagrams that lay the overall structure of the client and server. Next, we construct class diagrams based on the packages identified in the architecture. The final step of our design captures the interactions of the classes within the client and server using sequence diagrams.

4.2 Use case scenario

A use case is an illustration of the actions that can be done by a user on the prototype. For the purpose of experimentation, it is practical for use cases in both server and client to be initiated by the same user. The use cases are presented using UML diagrams with their corresponding use case descriptions.
4.2.1 Use case for the server

![Figure 4.1: Use case for the server](image)

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start server</td>
<td>The user sets the server into a listening mode and waits for incoming connection requests from the client</td>
</tr>
</tbody>
</table>

Table 4.1: Use case description for the server

4.2.2 Use case for the client

![Figure 4.2: Use case for the client](image)

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Client</td>
<td>The user starts device discovery to search for other Bluetooth devices that are within the client’s range.</td>
</tr>
<tr>
<td>Select Device</td>
<td>When the client detects several Bluetooth devices within its proximity, the name of the devices are displayed on the screen. The user chooses one device and request to be connected to the selected device and the security procedures are immediately initiated.</td>
</tr>
</tbody>
</table>

Table 4.2: Use case description for the client
4.3 Architecture

The following diagram shows UML packages representing the system architecture of the prototype.

Figure 4.3: Package diagrams for the prototype architecture

The architecture shows the communication between a client entity and a server entity. Both client and server are divided into 2 generic layers which are the presentation layer and application logic layer. The presentation layer serves as a user interface while the application logic layer provides Bluetooth connectivity and security services.

Part of the application logic for the client is based on the Bluelet component developed by Benhui\(^1\). Bluelet simplifies the development of Bluetooth by delivering features such as device discovery, service discovery and device selection. These functionalities are the implementation of the JSR-82. Bluetooth service record and

\(^1\) http://www.benhui.net
discoverability in the server are provided by the BlueCove API. Finally, cryptographic support on both client and server is provided by the BouncyCastle API.

The client entity consists of three packages which are the gui package, client package and security package. Below are the functionalities of the packages identified in the client:

- The gui package provides the presentation logic needed for the screen display on the client.
- The client package provides the client logic that consists of connectivity and message transfer to the server.
- The security package provides the security logic needed to generate a digital signature.

There is not much difference between the package diagrams for the client and the server. The packages for the server entity are the console package, server package and security package. The following are the functionalities of the packages identified in the server entity:

- The console package provides the presentation layer needed for console display on the server.
- The server package provides the server logic that consists of connectivity and message retrieval from the client.
- The security package provides the functionalities needed to verify a signature.
4.4 Design model

In this section, we model the design of the classes in the client and server using class diagrams.

4.4.1 Client design model

The following UML class diagram shows the classes involved in the client application:-

![Class diagram showing the classes in the client](image.png)

Figure 4.4: Class diagrams showing the classes in the client

- Client_MIDlet implements the programme’s lifecycle, initiates client’s functionalities and provides logging service.
- Client_Screen displays a basic menu on the screen interface.
- Start_Client is a Client_MIDlet’s child class that invokes the Client_Engine.
- Client_Engine handles client connectivity and message retrieval.
- Security_Engine serves as the main security controller for the client.
- Signature_Parameters generates signature parameters that serves as inputs to the signature algorithm.
- Sign provides signature generation.

4.4.2 Server design model

The following UML class diagram shows the classes involved in the server entity:-

![Class Diagram](image)

Figure 4.5: Class diagrams showing the classes in the server

- Server_Main initiates server functionalities and provides logging service.
- Server_Screen displays a basic menu on the console interface.
- Server_Engine provides server connectivity and message retrieval.
- Security_Engine serves as the main security controller for the server.
- Verify provides signature verification.
4.5 Interactions

In this section, we describe the interactions involved between classes in each entity using sequence diagrams. For the purpose of clarity, we dropped minor utility functions which are mostly various data type conversions into byte arrays.

4.5.1 Sequence diagram for the client

The following UML sequence diagram shows the interaction between the classes involved in the client application.

![Sequence diagram for the client](image)

Figure 4.6: Sequence diagrams showing the interactions between classes in the client
<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>The user starts the application on the client. This causes startApp() on Client_MIDlet to be called which in turn causes the Client_Screen to be created. Then, the screen displays a menu for user selection.</td>
</tr>
<tr>
<td>3</td>
<td>When the user selects <strong>Run Client</strong> on the menu, the control is returned to Client_MIDlet and the commandAction() initiates Client_Engine.</td>
</tr>
<tr>
<td>5-9</td>
<td>The constructor of Client_Engine calls send() to request a connection on the server. send() invokes generate Signature() on Security_Engine. Security_Engine generates the parameters needed for the signature by calling getEncodedTime() and getEncodedRandom(). A signature is generated on Sign when doSign() is called.</td>
</tr>
<tr>
<td>10</td>
<td>Client_Engine calls log() on Client_MIDlet to display the results.</td>
</tr>
<tr>
<td>11</td>
<td>Finally, the application terminates when the user chooses <strong>Exit</strong> on the menu.</td>
</tr>
</tbody>
</table>

Table 4.3: Descriptions for the interactions between classes in the client
4.5.2 Sequence diagram for the server

The following UML sequence diagram shows the interaction of the classes involved in the server application.

Figure 4.7: Sequence diagrams showing the interactions between classes in the server
<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>The user starts the application on the server. This causes <code>startApp()</code> on <code>:Server_Main</code> to be called which in turn causes a <code>commandListener()</code> on <code>:Server_Screen</code> to be initiated. Then, the console displays a menu for user selection.</td>
</tr>
<tr>
<td>3</td>
<td>When the user selects <strong>Start Server</strong> on the menu, the control is returned to <code>:Server_Main</code> and the <code>commandAction()</code> invokes <code>run_server()</code> on <code>:Server_Engine</code>.</td>
</tr>
<tr>
<td>4-7</td>
<td><code>run_server()</code> starts a thread that invokes <code>run()</code>. <code>run()</code> opens a server connection and listens for any incoming connection. During the lifetime of <code>run()</code>, this function calls <code>getTimeVerifySignature()</code> on <code>:Security_Engine</code>. <code>:Security_Engine</code> uses <code>doVerify()</code> on <code>:Verify</code> to perform the signature verification.</td>
</tr>
<tr>
<td>8</td>
<td>The thread on <code>:Server_Engine</code> terminates after the results of signature verification are received from <code>:Security_Engine</code>. At this point, <code>:Server_Engine</code> calls <code>log()</code> on <code>:Server_Main</code> to display the results.</td>
</tr>
<tr>
<td>9</td>
<td>Finally, the application terminates when the user chooses <strong>Exit</strong> on the menu.</td>
</tr>
</tbody>
</table>

Table 4.4: Description of the interactions between classes in the server
4.6 Conclusion

Based on our experience, the decision to adopt object-oriented design in our prototype contributes much towards the ease of implementation. Although the initial design process is very time consuming, the pace are tremendously improved as the basic packages and classes are completed. The flow of the programme is easily traced using the sequence diagrams and minimal changes are needed when the design is fully implemented. In conclusion, the design pattern has been constructed in such a way that allows for easy maintenance and later adjustments.