LANCASTER
UNIVERSITY

Matthew Weaver

Clean-Cut Software Development with Aspect-Oriented Programming

B.Sc. Computer Science

19th March 2004

An evaluation of the use of an aspect-oriented approach to software development when adding telephony features to a working telephone application

"I certify that the material contained in this dissertation is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation."

Date: 19th March 2004

Signed:

ABSTRACT

Aspect-oriented programming (AOP) is increasingly becoming a popular alternative to more traditional object-oriented programming (OOP) and is aimed at providing an alternative structuring mechanism for concerns that typically cut across entire systems. This project uses an aspect-oriented extension to Java, AspectJ, to implement various telephony features (such as call forwarding on busy) on top of an existing basic telephone system, written in Java. Both object-oriented and aspect-oriented approaches are used to add a particular feature to the underlying application which gives rise to a comparison of OOP and AOP techniques. The work is concluded with a detailed evaluation of how effective AOP is in removing scattered and tangled code from object-oriented systems.

Table of Contents

Figures

Figures (continued)

Tables

Working documents

The working documents provide full listings of all source code and electronic copies of the project proposal and project report. Point your browser at:

http://www.lancs.ac.uk/ug/weavermj/project/

Chapter 1. Introduction

Aspect-oriented programming (AOP) is increasingly becoming a popular alternative to more traditional object-oriented programming (OOP) and is aimed at providing an alternative structuring mechanism for concerns that typically cut across entire systems. The primary focus of this project was on analysing and evaluating the use of AOP in the particular context of adding telephony features to an existing telephone application.

Software design is based on stringent models that allow a programmer to identify a problem and break it down into a number of manageable tasks or concerns. The object-oriented design paradigm is one such model that has pervaded all aspects of computer science over the past decade or so, as noted by Garside (1998). OOP techniques are based on software models that resemble a natural thought process, where items from the real-world are represented as software objects. Gradecki (2003) observes how this technique has been successful on both small and large scales and how OOP is an integral part of the software development community.

There are certain situations, however, where the OOP model simply cannot encapsulate the requirements of an application in a clean and organised manner. Two fundamental conditions arise from this problem: code scattering and code tangling. Code scattering describes the situation where code relating to a particular concern is scattered across a number of different objects within the system. Code tangling is where multiple concerns are implemented within a single object. Both of these problems violate the basic principles of object-oriented design.

Aspect-oriented programming is a new paradigm that allows concerns to be separated out into distinct modules of functionality, solving the problems of code scattering and tangling within applications. The project set out to perform a fair analysis and evaluation of the use of AOP techniques in order to determine if an aspect-oriented approach to software development could solve the current problems that exist in the OOP domain.

In order to provide some value to the evaluation, a software application was designed using both object-oriented and aspect-oriented development techniques to give the writer first-hand experience of working with both methodologies. A phone system was chosen as the target application as it is an interesting case study, allowing basic functionality to be implemented in an object-oriented language and for telecommunications features such as call-forwarding-on-busy and call-barring to be added using an aspect-oriented language.

1.1 Aims and objectives

In order to accomplish the principal aim of performing an unbiased analysis and evaluation of AOP techniques, it was necessary to outline a number of supplementary objectives that the project should also address. The objectives are divided into two sections: those that exploit fundamental concepts in computer science and those that relate to the relatively new area of aspect-oriented programming.

1.1.1 Objectives : The fundamentals of computer science

Objective 1 *To design a basic model of a working telephone system following an object-oriented design pattern*

Objective 2 *To model the telephone system as a distributed client/server application, with distributed phone clients and a centralised switchboard acting as a server*

Objective 3 *To provide each virtual phone client with a friendly graphical user interface and a look-and-feel that resembles a typical phone from the real-world*

Objective 4 *To incorporate fault-tolerant mechanisms within the system, allowing the switchboard to be relocated in the event of failure whilst causing minimal impact to the operation of the phone network*

Objective 5 *To extend the basic application with a dynamic telephone book capable of automatically maintaining an up-to-date list of all the phones online in the system*

1.1.2 Objectives : Aspect-oriented programming

Objective 6 *To gain a detailed understanding of the concepts behind an aspect-oriented approach to software design*

Objective 7 *To layer a chosen feature on top of the phone application as both an object and an aspect*

Objective 8 *To perform a critical review of the procedures involved when using OOP and AOP techniques to add a feature to the existing phone application*

Objective 9 *To add additional features to the phone application using the favoured technique*

Objective 10 *To conclude the project by performing an evaluation of aspect-oriented techniques, stating the extent to which AOP solves the problems associated with OOP*

1.2 Report overview

Following this short introduction, chapter 2 examines the background to the project in more detail, using an example to motivate the need for AOP techniques, and explains important terminology used in aspect-oriented development. Chapter 3 details significant design decisions and outlines the overall architecture for the basic telephone system. The implementation of the basic system is discussed in Chapter 4, where noteworthy characteristics of the system are demonstrated in more detail. Chapter 5 continues the implementation, adding additional functionality in an incremental manner by layering telephony features on top of the basic application and performing a review of both an object-oriented and an aspect-oriented approach to adding a feature. Chapter 6 presents an overview of the system in operation, providing a walkthrough of screen shots to give the reader a feel for the user interface of the application. Chapter 7 performs a detailed evaluation of AOP techniques, investigating their importance and discussing their impact in five key areas. Finally, chapter 8 concludes the work by reviewing the aims and objectives of the project and predicting a future for aspect-oriented design.

Chapter 2. Background

This background chapter starts by giving an overview of object-oriented programming techniques, focusing on the milestones in its development and drawing on its modern-day limitations. This leads the discussion onto the topic of aspect-oriented programming, where new concepts are introduced and the motivation for AOP techniques is made clear. The chapter goes on to explain how the AOP model fits around the existing object-oriented paradigm and looks at a specific Java implementation of AOP, AspectJ. The notion of IP telephony is introduced and a brief study of existing Voice-over IP technologies is conducted.

2.1 An overview of Object-Oriented Programming

Object-oriented programming is a technique where real-world objects from everyday life are modelled using software. Real-world objects have two principal characteristics: state and behaviour. A software object represents an object from the real-world by storing state information in *variables* and implementing behaviour in *methods* (Sun Microsystems). All the state and behaviour needed to represent a given real-world object is encapsulated within its equivalent software object.

2.1.1 An example object

A very simple real-world object is a desk lamp. To represent the lamp as a software object one must first identify the state and behaviour that the lamp exhibits. The state is simple, it is merely the status of the lamp, "*on*" or "*off*". The behaviour associated with the lamp can be thought of as the actions performed to operate the light, such as "*switch on*" and "*switch off*".

2.1.2 The benefits of an OOP approach

There is no doubt that OOP is a proven methodology. Gradecki (2003) observes that using an OOP approach to software development has dramatically enhanced the quality of software that we see on our machines today. OOP software solutions are developed by creating software objects and grouping them together to form an entire system. Using such a structured approach allows components to be reused, produces a more modularised system and provides developers with a clear model that represents a real-world domain.

2.1.3 Where the OOP model breaks down

OOP has many advantages but in some situations an object-based approach to software design can be problematic for developers. This scenario typically arises when a decision is made to include additional functionality in an existing application, in which case the application model that represents the real-world domain must be altered so that it exploits the new features. The changes made to the model often break the very concept of OOP itself; state and behaviour associated with the new features is cumbersome to encapsulate, and as a result it becomes scattered across different objects within the system. This is a problem that has been termed *code scattering*.

Take the example of the simple desk lamp object. Now envisage the lamp as part of a lighting system for an office. There are other types of lights in the building, namely ceiling lights, wall lights and elevator lights. The lighting system is automated so that lights are turned on and off as a person enters and exits a room. The system is designed using an object-oriented approach, with each type of light being modelled by a software object.

Now, suppose the lighting system has been in successful operation for a number of weeks but the office workers get disturbed by the constant switching on and off of lights around them. They request that the office manager install a dimmer system, so that the lights are dimmed when a room is empty and only completely switched off at night when everyone has left the building. In order to deploy this functionality, each light object must be modified so that the brightness of the bulb dims to 50% instead of it switching off completely when the last person leaves a room. Some additional logic must also be embedded within each light object so that it knows to switch itself off completely after 6pm when the office is empty.

2.1.4 The notion of cross-cutting code

The additional functionality can be added to the application but not in a way that is consistent with object-oriented design patterns. Ideally, the dimmer should be represented as an object in its own right, encapsulating all the state and behaviour associated with dimming, but this is not possible as every light object needs to contain code that will dim the light. The code that controls the dimming of the lights must be distributed across different objects and is said to *cut across* the lighting system.

2.2 The motivation for Aspect-Oriented Programming

The previous section used a simple scenario to demonstrate how the object-oriented design philosophy is easily broken when new functionality is added to existing applications. Realistically, the inclusion of dimmer functionality within the lighting system is a relatively lightweight task, as only four objects need to be modified. However, the task of modifying much larger applications becomes extremely heavyweight, as the following example will demonstrate.

Consider an international bank that has a large number of branches around the world. Banking is a very complex and detailed domain and there would likely be hundreds or possibly thousands of objects in a typical distributed banking system. Suppose that the system is to be upgraded so that it keeps a log of all user activity and transactions that are processed in the system. The logger could of course be realised as an object, perhaps writing user activity and transaction information to a secure file. At all points in the system's objects where code exists to deal with a user gaining access to the system, or dealing with a transaction, additional code must be added to write the details of the activity to the log. The implication of adding the logging functionality may incur changes to more than a hundred objects and potentially there could be a large number of changes to make within each object.

Clearly, there is a need for a more flexible structuring mechanism that is capable of completely encapsulating the requirements of a given concern in a single entity. This need is satisfied by the notion of aspect-oriented programming.

2.3 So what is Aspect-Oriented Programming?

Aspect-oriented programming is described by Gradecki as one of the most promising solutions to the problem of creating clean, well-encapsulated objects without extraneous functionality. In a nutshell, the AOP paradigm addresses the problems of code scattering and code tangling by allowing cross-cutting concerns to be separated out into single units of functionality over and above the object functionality, where each new unit encapsulates all the necessary code required to implement a particular concern. These units are called *aspects*.

2.3.1 Aspect-oriented terminology

The concept of AOP introduces a wealth of new terminology to the object-oriented programmer. This section explains the meaning of these new terms.

concern	A *concern* should be thought of as a requirement. There are typically two types of concerns, classified by Gradecki as “must-have” and “nice-to-have” concerns. The must-have concerns are those that must be implemented by the component language in order to provide an application with an acceptable level of functionality. The nice-to-have concerns are those that would provide an additional degree of functionality to the application but are not necessarily fundamental to the system.
	A given concern may cut across one or more different concerns within a system, resulting in a system that contains scattered code. These types of concerns are termed *cross-cutting concerns*.
aspect	An *aspect* encapsulates all the code related to a particular concern in a single structure. Aspects are analogous to classes in object-oriented languages. Taking the earlier examples, dimming would be an aspect of the lighting system and logging would be an aspect of the banking system.
join point	A *join point* is a well-defined point in the execution of an application. Gradecki regards the join point as one of the most important concepts in the aspect-oriented design paradigm. Example join points are calls to methods or constructors, instantiation of objects, the reading or writing of class attributes and the handling of exceptions.
pointcut	A *pointcut* is identified by one or more join points. Pointcuts are used to determine precise moments where a given concern will cross-cut the underlying application. Typically, pointcuts denote interesting points of execution within a program.

advice *Advice* is the term given to a section of code that executes when a join point in an application is reached. Advice is fundamental to the concept of aspect-oriented programming because it is the code contained within each advice block that adds the cross-cutting functionality to the underlying application. There are three main types of advice: advice that executes *before* a join point, advice that executes *after* a join point, and *around* advice that can be used to execute code both before and after a given join point (or instead of the join point).

2.4 How does AOP fit in with the current OOP model?

Aspect-oriented programming is not a replacement for the proven technique of object-oriented design; rather it is an extension to the object-oriented paradigm. Cross-cutting concerns can be implemented as aspects on top of an existing object-oriented application without needing to modify any of the underlying code.

2.4.1 The distinction between OOP and AOP languages

In order to understand the aspect-oriented development process, it is important to appreciate the distinction between the language used to code the primary concerns and that used to code cross-cutting concerns. The language used to implement the primary concerns of an application is referred to as the *component language*, whilst *aspect language* refers to the language that is used to integrate the cross-cutting concerns.

2.5 Aspect-oriented programming language bindings

Support for aspect-oriented software development is available in a number of different programming languages, in the form of AOP language bindings. Table 2.1 below shows a number of AOP language bindings and their associated programming platform.

All of these AOP language bindings use the same terminology as defined in section 2.3.1; however, the syntax of each binding will differ considerably across the languages.

AOP Language Binding	Native Programming Language
AspectJ	Java
AspectWerkz	Java
AspectC#	C#
Aspect.pm	Perl
AspectC++	C++
AspectC	C
Pythius	Python
AspectS	Smalltalk
AspectR	Ruby

Table 2.1 ***A list of AOP language bindings for popular programming languages***

The list of languages in table 2.1 contains both object-oriented languages, such as Java, C++, C# and Python, and functional languages, namely C. It is interesting to note that cross-cutting concerns can be implemented as aspects on top of languages that are not based on the object-oriented design paradigm.

2.5.1 The chosen component and aspect languages

The component language that was used to implement the primary concerns of the application was Java. As an object-oriented language, Java provided the means to investigate the impact of adding cross-cutting concerns to an existing application. Despite the obvious efficiency problems associated with a language that runs in a virtual machine, this was not seen as a problem, as the application being written was intended to function as a working model and not become a product for release into a commercial environment.

As Java was chosen for the component language, the choice for the aspect language was limited to language bindings that are compatible with Java. The aspect-oriented development community places emphasis on two AOP language bindings for the Java platform; AspectJ and AspectWerkz. AspectJ prevailed as the binding of choice, namely due to its popularity among professional developers, its open-source nature and its extensive on-line support community. At the time of writing, AspectJ has been nominated as a finalist in the 2004 Annual Jolt Product Excellence & Productivity Awards and has previously won an award for “Most Innovative Java Product or Technology” at JavaWorld’s 2003 Editor’s Choice Awards.

2.6 The aspect-oriented development process

When producing a standard Java application, Garside (1998) notes how Java source code is compiled to Java bytecode which is interpreted by a virtual machine at run-time and executed in a protected operating environment. When cross-cutting concerns are implemented as aspects, some work must be done to integrate the AOP code into the underlying Java application. This process is known as *weaving*.

There are four main types of weaving; compile-time, link-time, load-time and run-time weaving. The current version of the AspectJ language uses link-time weaving to weave cross-cutting concerns into an underlying application. Figure 2.1 below demonstrates the link-time weaving process.

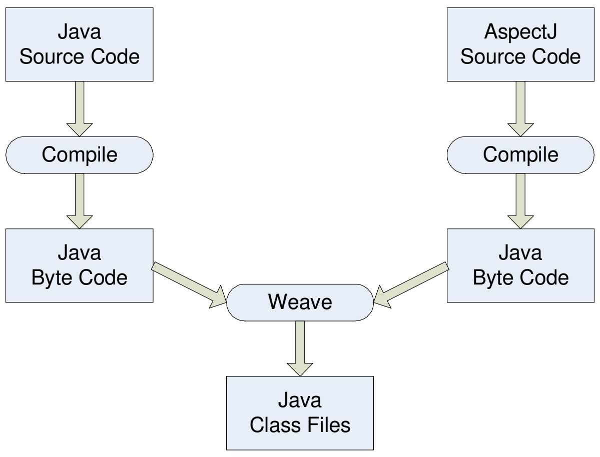


Figure 2.1 ***The link-time weaving process***

AspectJ includes its own compiler, called *ajc*, that compiles both the aspect source code and the source code for the underlying application. ajc is built on top of IBM's Eclipse project compiler which provides strict compliance with Sun Microsystems' Java specification.

2.7 IP telephony

IP telephony is the term used to describe the transmission of voice calls over a data network, such as the internet. In the telephone industry, IP telephony is commonly referred to as Voice-over IP (VoIP).

Traditional telephone systems operate over circuit-switched networks, where a virtual circuit is established between the caller and the receiver for the entire duration of a call. This architecture is the basis of the Public Switched Telephone Network, or PSTN for short. Telephone conversations are transmitted over the PSTN at a fixed rate of 64 kilobits per second (Kbps) in each direction, forming a total transmission rate of 128 Kbps for bi-directional communication. A ten minute conversation over a traditional PSTN equates to the sending and receiving of 9600 kilobytes (KB) of data, which is roughly equal to 9.4 megabytes (MB).

On closer examination of a typical telephone conversation it can be observed that most of the data being transmitted is redundant. Whilst one party is talking the other is usually listening, meaning that only half of the 128 Kbps bandwidth is in use. Most conversations also contain a significant amount of dead air, where neither party is talking, and thus the entire channel is filled with useless information.

VoIP technology drastically reduces the amount of redundant information that naturally occurs in telephone conversations by digitally encapsulating voice traffic into small packets that are transmitted over a packet-switched network, such as the internet, using the Internet Protocol (IP). Each packet contains a destination address so that it can be successfully routed across the network to the intended recipient. At the other end, the recipient obtains the packets from the network and reassembles them into the original voice stream.

In comparison to circuit-switched networks like the PSTN, a packet-switched network does not maintain a permanent connection for the duration of a call. Instead, whenever a user speaks into their phone a set of packets is transmitted across the network, typically using the User Datagram Protocol (UDP). Whenever there is dead air on the line, no data is sent. This approach results in greater utilisation of network bandwidth, allowing multiple calls to occupy the same amount of space that one call did in the circuit-switched PSTN. Further savings can be made by compressing voice data using a suitable codec, such as G.729, which is part of the International Telecommunications Union's (ITU) H.323 protocol suite. *(Tyson, http://www.howstuffworks.com)*

2.8 Voice-over IP protocols

There are two protocols that dominate the world of VoIP: H.323 and SIP.

H.323 was first standardised by the ITU in 1996 and has since been revised several times; the current revision is at version 5 which was released in 2003. The protocol provides a number of complex standards for transmission of real-time audio, video and data over packet-switched networks. The H.323 protocol family was primarily engineered to provide consistent delivery of real-time media in networks that had little support for quality of service (QoS).

In 1999 the Internet Engineering Task Force (IETF) developed an alternative protocol to H.323, the Session Initiation Protocol (SIP), which has since been revised in 2002. SIP provides more efficient mechanisms for setting up an interactive session in which a user can transmit video, audio and text-based traffic. Operating at the application layer, SIP was designed specifically for IP telephony and integrated support for quality features such as call forwarding, caller identification and multicast conferencing, whilst reducing the complexity involved in setting up an end-to-end path between a sender and a receiver. *(http://searchnetworking.techtarget.com)*

The important distinction to be made between the two protocols is that they were engineered for two different industries; the ITU geared H.323 towards the telecommunications sector whereas the IETF focused on providing a simpler and more manageable protocol to deploy over the IP-based internet. Both protocols have emerged as standards for VoIP communication and there is much debate over which standard will prevail when the telephony world becomes primarily IP based. *(http://www.iptel.org/info/trends/sip.html)*

2.9 Voice-over IP solutions

VoIP solutions come in two flavours that are either software or hardware based, although they can be a combination of the two approaches in some cases.

In software-only VoIP solutions, a *soft phone* application runs on the users desktop and mimics the behaviour of a traditional telephone. The telephone handset can be simulated by connecting a microphone and speakers to the machine, although often an all-in-one headset is used by busy office workers who make a lot of calls, as shown in figure 2.2 below.

Figure 2.2 ***A Voice-over-IP headset***

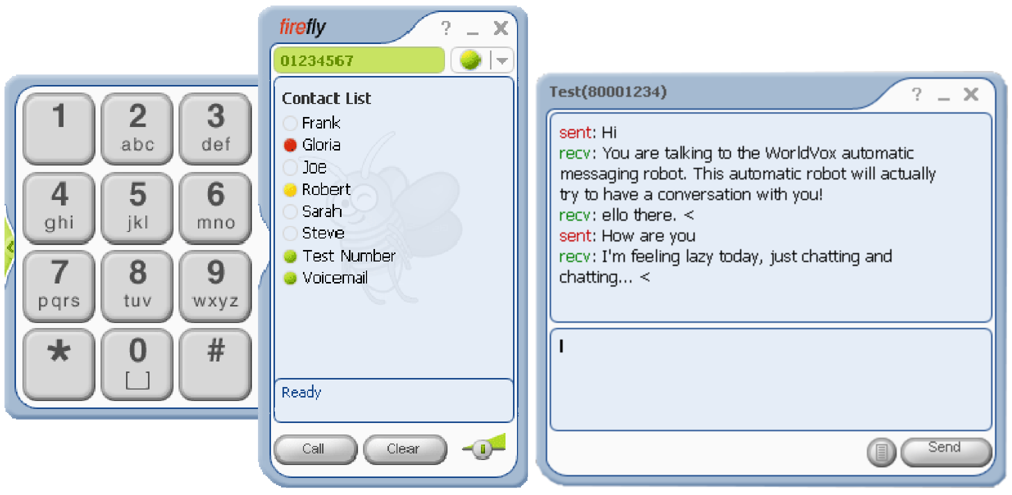
Hardware-based VoIP solutions are often installed in large offices. The phone appears to operate exactly the same as a traditional phone connected to a PSTN; however, it is really a client of an IP-based phone network connected directly with an RJ45 patch cable. Figure 2.3 below shows how a hardware-based IP phone looks no different to an ordinary office phone.

Figure 2.3 ***Alcatel's Easy e-Reflexes IP Phone***

In certain situations, users opt to use both a soft phone client coupled with a physical handset that connects directly to their machine, usually via a USB port. This method gives users greater control over their telephone application whilst providing them with the feel of a quality telephone.

What follows is a brief look at three existing soft phone applications. The graphical user interface of each phone and the features on offer are the main points of interest.

2.9.1 Firefly VoIP Soft Phone 1.3

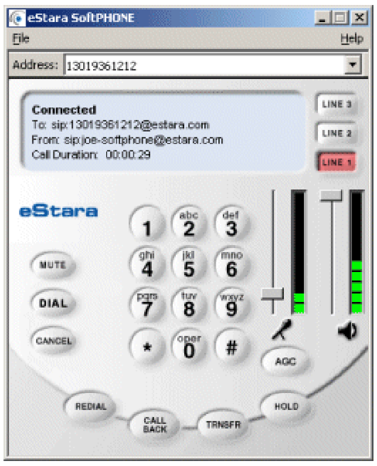


sent: Hi
recv: You are talking to the WorldVox automatic messaging robot. This automatic robot will actually try to have a conversation with you!
recv: ello there. <
sent: How are you
recv: I'm feeling lazy today, just chatting and chatting... <

Figure 2.4 ***Firefly's User Interface***

Figure 2.4 above shows the user interface of Firefly, a freeware application provided by freshTEL (*http://www.freshtel.net*). The application allows users to maintain a contact list of friends and family and provides voice call, instant messaging and voice mail capabilities.

2.9.2 eStara Softphone


Address: 13019361212

Connected
To: sip:13019361212@estara.com
From: sip:joe-softphone@estara.com
Call Duration: 00:00:29

Figure 2.5 *eStara Softphone in operation*

eStara is a professional soft phone application aimed at business use. Figure 2.5 shows the user interface of the phone, which can be customised to the user's preference. The application uses SIP to deliver enhanced capabilities and allow for PC-to-phone, phone-to-PC and PC-to-PC calls. The main features of the phone are call forwarding, call transfer, line hold, last number redial, speed-dial and caller ID. More detailed information on this product can be obtained by visiting *http://www.estara.com/softphone*.

2.9.3 LIPZ4 SIP Soft Phone

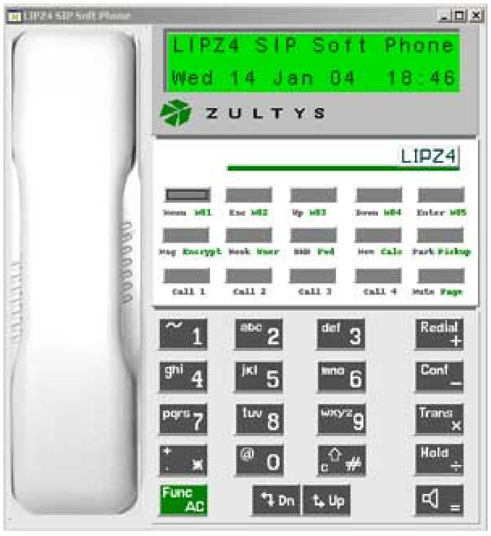


Figure 2.6 *LIPZ4's phone-like user interface*

LIPZ4 is a soft phone designed explicitly for the Linux operating system. LIPZ4 uses SIP, which makes it compatible with existing IP phones and IP networks. Figure 2.6

above is a screen-shot of the user interface provided by the application. For more information on LIPZ4 visit *http://www.lipz4.com/lipz4.htm*.

2.10 Summary

This chapter started by looking at existing object-oriented programming techniques and discussed the problems that can arise when cross-cutting concerns are added to an existing system. Aspect-oriented programming concepts were introduced using a motivating example to show how concerns can be separated out into single units of functionality, called aspects. Various aspect-oriented language bindings were mentioned and AspectJ was chosen for its large success in the aspect-oriented software development community. The aspect-oriented development process was briefly outlined where the notion of weaving AspectJ source code into existing Java source code was explained.

It was far beyond the scope of this project to produce a system that supported the transportation of real-time voice data using protocols such as H.323 or SIP. The model only needed to demonstrate that calls could be established and teared-down, so an important decision was made to simulate conversations using simple text-based instant messages.

The discussion then moved on to the broad topic of IP telephony and Voice-over IP technologies. H.323 and SIP were highlighted as two important protocols that can be used for real-time IP-based communications. The examples of existing VoIP solutions presented in section 2.9 show how soft phone applications typically have well-structured user interfaces and are fundamentally easy to use. The project will build on these principles by producing a basic soft phone application with an intuitive user interface similar to that of a classic phone from the real-world. Advanced functionality, for example fault-tolerant mechanisms, will be included under the bonnet of the application to abstract the user from complicated operations.

Chapter 3 goes on to document important design decisions and discusses implications for the basic telephone system.

Chapter 3. Design

Chapter 1 introduced the aims and objectives of the project, placing emphasis on the evaluation of aspect-oriented programming techniques that constitutes a large part of this project. In order to add some weight to the evaluation, it was decided that a telephone system would be written in an object-oriented programming language (Java) and telephony features added using an aspect-oriented programming language binding (AspectJ). Chapter 2 discussed the notion of IP telephony and VoIP technologies in a fair amount of detail, leaving plenty of scope to develop a sophisticated software telephone system.

As the evaluation of AOP techniques is the focal point of this project, it was decided to produce only a *working model* of a basic telephone system, allowing telephony features to be added as aspects without introducing unnecessary complexities that are present in a complete real-world system. This chapter contains important design decisions and considerations for this basic telephone system, such as:

- the design of the overall system architecture,
- the role of the switchboard,
- how phones communicate with one another,
- whether the phones should include any level of security or authentication,
- how the system can be made fault-tolerant, and
- the mechanisms that can be used to support the dynamic telephone book.

3.1 Overall system architecture

The system was envisaged as a client/server model, with distributed phone clients connecting to a centralised switchboard acting as a server. Figure 3.1 below outlines the basic topology of the phone network.

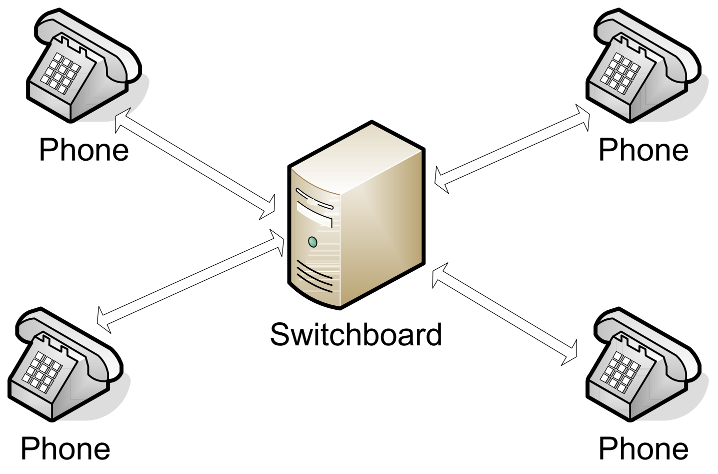


Figure 3.1 ***The basic topology of the phone***

system

As discussed in the background chapter, traditional phone networks such as the PSTN maintain a switched connection between hosts for the duration of a call. This type of connectivity is simply not possible in a packet-switched network such as Ethernet and it would be difficult to model such techniques using software. A more rigorous approach was to envisage each phone first connecting to a centralised switchboard to obtain details of the phone it wished to communicate with and then forming a direct point-to-point connection with the intended recipient.

3.1.1 Point-to-point architecture

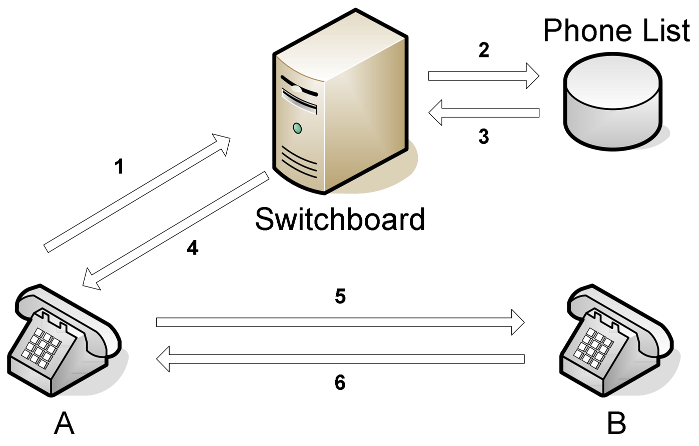


Figure 3.2 ***Example of how two phones communicate with each other***

Figure 3.2 above illustrates how two phones can establish a connection with each other, where phone A wishes to communicate with phone B. The only information known about B by A is B’s phone number. The following steps outline the procedures involved in setting up a direct point-to-point connection between A and B.

1. Phone A sends a message to the switchboard informing it that it wishes to communicate with phone B. The message consists of phone A’s identity and B’s phone number.
2. The switchboard extracts B’s phone number from the message and performs a lookup on a list of all the phones that are connected to the network.
3. Assuming that the lookup is successful, the Phone List returns more detailed information about phone B, namely B’s precise network location, to the switchboard.
4. The switchboard forwards B’s details back to phone A.
5. Phone A receives the message from the switchboard and extracts the exact location of phone B on the network. Phone A then constructs a new message containing its own precise location on the network and sends this to B.
6. Phone B receives the message from A and knows that A wishes to communicate. B extracts A’s location from the message and sends a message back to A. The two phones are now directly connected by a point-to-point link.

3.1.2 Security considerations

Security is an important consideration in any distributed system and telephone systems are no exception. The PSTN relies mainly on physical security; that is securing telecommunications buildings, safeguarding information about how a physical connection is actually made and protecting cable trunks by burying them deep underground. Taylor (2004) explains how "PSTN security isn't exactly airtight" as calls transmitted over analogue lines can easily be tapped, but nevertheless basic security mechanisms are put in place in an effort to deter snoopers.

The proposed point-to-point architecture illustrated in figure 3.2 has two main security implications. Firstly, messages between phones are not encrypted (i.e. they are sent in plain text) which leaves them vulnerable to snoop attacks. Secondly, there is no authentication procedure on the network making it impossible to guarantee that phone A is actually connected to phone B (it could be connected to a malicious phone C at a completely different location).

It is interesting to compare the architecture depicted in figure 3.2 with Needham and Schroeder's secret key authentication protocol, shown below in figure 3.3. The significant characteristics of this architecture are the careful exchange of secret keys (for the encryption and decryption of messages) and the last two steps (4 and 5) that ensure authentication of both hosts A and B.

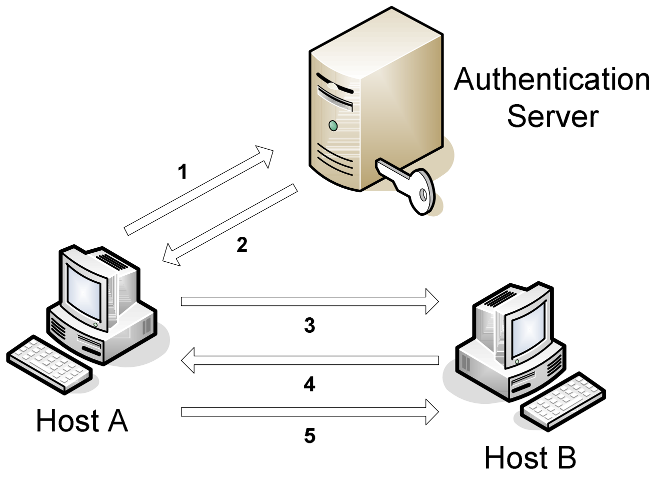


Figure 3.3 ***Needham and Schroeder's secret key authentication protocol***

Blair (2003) outlines the steps involved in the authentication process as follows:

1. Host A sends a unique message (ID1) to the authentication server, telling it that it wishes to communicate with host B.
2. The authentication server acknowledges the message and issues host A with the following information, all encrypted with A's secret key: a key to communicate with host B (Key[A,B]), a ticket to pass to host B encrypted in B's secret key, and the original message for reference (ID1).

3. Host A decrypts the reply from the authentication server and forwards the ticket to host B.
4. Host B receives the ticket from host A and decrypts it. The ticket contains Key[A,B]. Host B then sends a unique message to host A (ID2) encrypted with Key[A,B].
5. Host A returns an agreed transformation of the message from B (ID2 – 1), encrypted with Key[A,B].

At the end of step 5, both hosts A and B are authenticated and can communicate securely by exchanging messages encrypted with Key[A,B]. Implementing a cryptography system such as that used in Needham and Schroeder's secret key authentication protocol is far beyond the scope of this project but it is an interesting analogy to make. Should the model be extended into a fully-working application then it is likely that an authentication system based on the Needham and Schroeder algorithm would be included.

3.1.3 Benefits of a non-switched network model

Although this model of point-to-point connectivity between phones is not a true reflection of a real-world telephone network like the PSTN, it does bring about a number of advantages that can be exploited to provide a level of Quality of Service (QoS) within the network. In essence, the switchboard serves as a central database, maintaining up-to-date information about all of the phones online within the network and only issuing responses to phone clients when they wish to make a call. By not having an active part in the routing of calls to their destination, the load on the switchboard is greatly reduced. This factor alone could provide real benefits if the system were to be installed in a large office, as the machine used to host the switchboard would not need to have an extensive amount of processing power, therefore making IP based communications cost-effective.

3.1.4 Message-passing infrastructure

The process outlined in section 3.1.1 relies on messages being passed around the system that contain specific information about various phones on the network. An object was needed that could both encapsulate this information in a consistent manner and be concise enough so that it could be passed around the network without compromising efficiency. There are three basic pieces of information that constitute a phone's identity: the telephone or extension number of the phone, the name of the phone's owner and the IP address that identifies the unique location of the phone on the network. Figure 3.4 below shows a graphical representation of this information encapsulated into a Phone object.

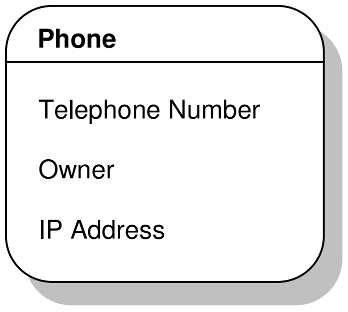


Figure 3.4 ***Structure of the Phone object***

These Phone objects will be the main type of messages passed around the system, although other short text-only commands will be needed to control the state of phones on the network.

3.1.5 Communication requirements

In a typical VoIP system, voice traffic is compressed and encapsulated into packets that are sent out across the network using the User Datagram Protocol (UDP). UDP is a connection-less protocol that operates at the transport layer of the International Standard Organisation's Open System Interconnect (OSI) model. UDP is suitable for transporting real-time VoIP traffic as it does not have any of the latencies that are associated with connection-based protocols such as the Transmission Control Protocol (TCP). TCP is a connection-oriented protocol that maintains state information about a connection between two remote hosts on a network. Although TCP is a reliable protocol, it should not be considered for real-time communication as the process of establishing a connection and sending acknowledgements introduces considerable delay to the stream.

i) Phone-to-phone connectivity

Despite its disadvantages, TCP was chosen as the protocol to be used for establishing point-to-point connections between phones on the network. As a decision had already been made to simulate conversations using text-based instant messages, the delay associated with setting up a connection and maintaining its state became irrelevant. Modelling the connection between two phones as a single TCP session was analogous to the circuit-switched model employed in the PSTN, where a virtual circuit is established between caller and receiver for the duration of the call. The TCP approach also provided a simple means of exchanging messages between phones once the connection had been established, by reading and writing data to the connection's socket.

ii) Phone-to-switchboard connectivity

Connectivity between individual phones on the network and the centralised switchboard was envisaged using Java Remote Method Invocation (RMI). RMI is best described by Coulouris (2001) as a technique that "allows objects to invoke methods on remote objects using the same syntax as for local invocations". Grosso (2002) expands on this definition by stating how "RMI provides Java programmers with a lightweight solution to a heavyweight problem [of writing] efficient fault-tolerant applications with very little time or effort".

Using RMI greatly simplifies the design of client/server systems by providing mechanisms to deal with complicated distributed systems principles, such as marshalling and demarshalling of parameters and return values. By hiding these operations and providing a structured application programming interface (API), RMI helps the programmer to concentrate on implementing the necessary business logic in a traditional object-oriented approach without much consideration for the detailed mechanics of distributed systems design.

At the heart of the RMI infrastructure is a *naming service* that is hosted on a central server and operates like a telephone directory, mapping pseudo names to remote objects. Sun Microsystems provide a default naming service known as the *RMI registry* with their

Java implementation. A client can issue a lookup request to the registry using the pseudo name and obtain a direct reference to an object hosted on a remote server.

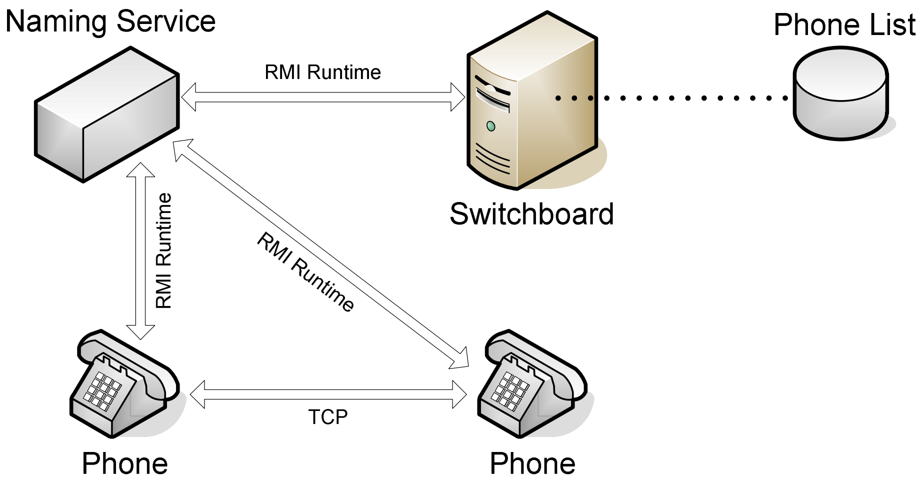


Figure 3.5 ***Overall system architecture with the added RMI naming service***

Figure 3.5 above outlines the overall architecture for the telephone system with the RMI naming service now in place. Notice how this figure differs to figures 3.1 and 3.2, the main difference being that the direct link between each phone client and the switchboard has been removed. Phone to switchboard connectivity is managed by the RMI infrastructure and greatly simplifies client/server interactivity. Note that the point-to-point TCP connection between phones on the network has also been added to the diagram.

Figure 3.6 below demonstrates how the naming service can be used to obtain a reference to a remote object (the Phone List) and how it can be seemingly accessed as if it were a local object. The numbered points 1 through 4 are explained on the next page.

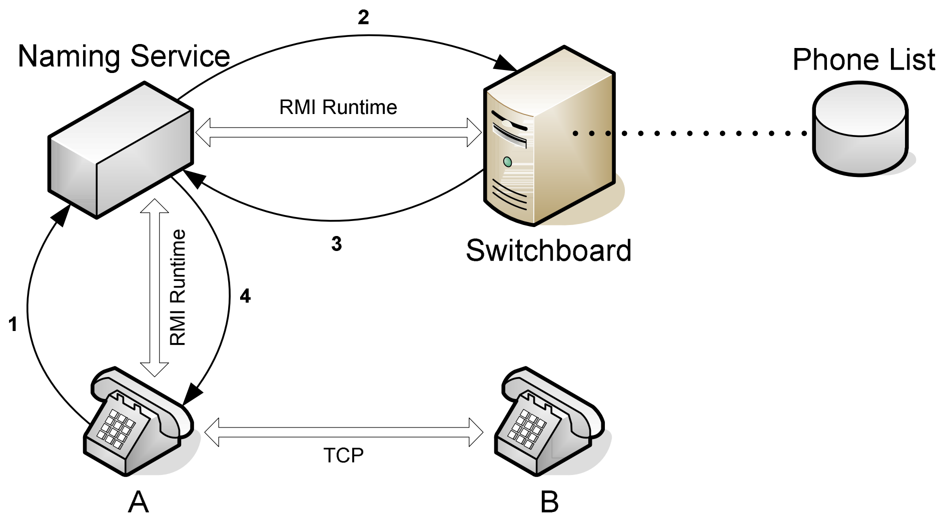


Figure 3.6 ***Example of how the naming service can be used to obtain a reference to a remote object***

1. The phone client issues a lookup request to the naming service, quoting the name of the service that it wishes to access.
2. The naming service resolves the service name into a remote object, the phone list, hosted on the switchboard.
3. The switchboard returns the phone list object to the naming service.
4. The naming service returns the phone list object to the phone client.

The steps outlined above are very primitive and only serve to give the reader a brief overview of the system architecture. However, it is important to mention how stubs and skeletons are used to marshall and demarshall data across the network.

When a phone client makes a lookup request to the naming service, a stub class is obtained automatically from the switchboard server. This stub acts as a simple proxy, allowing the client to call methods on the instance of the stub class as if it were a local object. Behind the scenes, the RMI runtime connects to the switchboard using a shared socket, marshalls all information associated with the method call (namely the method name and its arguments) and passes this over the socket connection to the switchboard's skeleton. The skeleton demarshalls the data and performs the method call on the actual remote object, which is the phone list in the above example. The switchboard returns a value to the skeleton which in turn marshalls the value and passes it back across the network to the stub. At the client side, the stub demarshalls the return value and returns it to the calling code.

3.2 Centralised switchboard

The previous section outlined important design decisions and considerations for the overall architecture of telephone system. This section identifies functional requirements and presents design decisions for the centralised switchboard application.

3.2.1 Phone list service

The main purpose of the switchboard is to provide a centralised location for storing a list of all the phones on the network. Phones contact the switchboard to obtain the exact location of a given phone on the network in order to make a call. This procedure was outlined in section 3.1.1. An important decision had to be made as to how the phone list should be stored: persistently (e.g. in a file on disk) or non-persistently (as an object in memory). As the switchboard was envisaged as an always-on server there was not a great need to store the phone list in a persistent manner. Instead, the phone list was realised as a non-persistent object which proved to be an efficient choice. As the list would need to change every time a phone comes or goes offline, constantly editing a file on disk would consume too many resources, so opting to store the list in main memory meant that it could be updated and accessed much more quickly, placing less demand on the switchboard.

It was decided that the phone list should support the following actions:

- the ability to add a phone to the list when it comes online,
- the ability to remove a phone from the list when it goes offline,
- the ability to return the complete phone list at any time,
- the ability to report the number of phones online at any given moment, and
- the ability to return the specific location of a given phone on the network.

As communication between individual phones and the switchboard is to be implemented with Java RMI, the phone list service will need to be hosted as a remote object on the switchboard and registered with the naming service. There is a standard naming convention that should be followed when assigning names to network services in an RMI infrastructure. Grosso defines the standard format of names in the RMI registry as a single string that takes the following format:

//host-name:port-number/human-readable-name

In the above definition, *host-name* and *port-number* describe the machine that the RMI registry is running on (so that clients can correctly locate the naming service) whilst *human-readable-name* is a pseudo name for describing a particular service registered with the naming service. Both *host-name* and *port-number* have default values of *localhost* and *1099* respectively. If either of these two pieces of information are omitted when registering a service then the default values are used. The phone list service will probably be registered with a name such as *//server/PhoneListService* where *server* is the host name of the machine hosting the switchboard.

3.2.2 Fault-tolerance

Linking back to the initial aims of the project set out in section 1.1, you can see that one of the objectives was to extend the basic telephone system by incorporating fault-tolerant mechanisms to allow the switchboard to be relocated in the event of failure, whilst causing minimal impact to the operation of the phone network.

Phone clients connect to the switchboard by performing a lookup request on the naming service, using the pseudo name of the service that they wish to access. So far, it has been assumed that the pseudo name will be a fixed string value, hard-wired into the application code. In order to relocate the switchboard after a failure, every phone client on the network needs to be aware of the switchboard's new location so that it can re-direct lookup requests to the new machine.

The solution to this problem will rely on a broadcast mechanism. The location of the switchboard will be stored in a string and broadcast to all phones on the network at regular intervals of 5 seconds. Each phone will therefore need to listen for messages on the network and update the location of the switchboard should it relocate to another terminal. Figure 3.7 on the next page demonstrates how this will work.

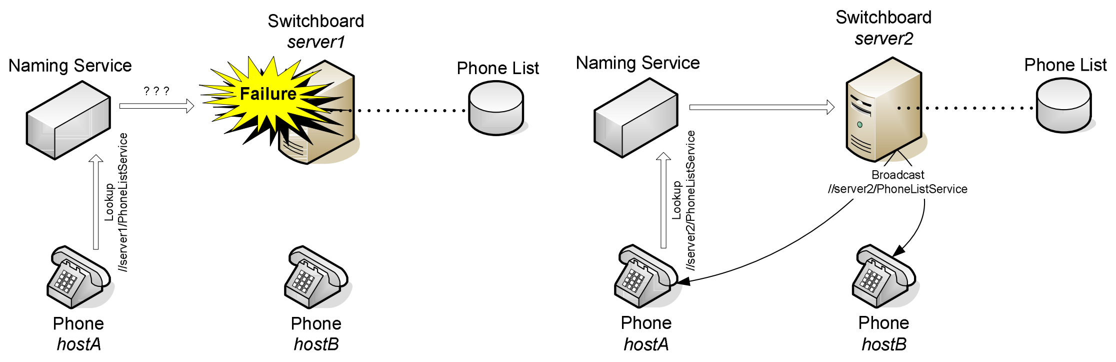


Figure 3.7 *Illustration of how the broadcast mechanism helps to make the system fault-tolerant*

3.2.3 Multi-threading issues

The notion of multi-threading is a complicated one. Grosso makes reference to a humorous remark about multi-threading once seen in an e-mail signature:

Writing a multi-threaded program is hard.
Writing a correct multi-threaded program is impossible.

The essence of this project is not in producing a fully-working application for deployment into a commercial environment, as has already been stated several times. However, the switchboard will need to be capable of handling multiple client requests in a consistent and thread-safe manner. The specifics of writing a multi-threaded application are too involved to discuss here, but considerable effort will be made to ensure that the switchboard can handle multiple requests simultaneously.

3.2.4 Graphical user interface

The switchboard is designed to be an always-on server that requires little or no human intervention. In a real-world system, the switchboard might have a complicated user interface allowing a telephone supervisor to set a multitude of options and configuration settings for the network. In the simple model of the telephone system, the graphical user interface (GUI) of the switchboard will be limited and only exist to provide a simple indication of the number of phones connected to the network. Figure 3.8 below shows a user interface prototype for the switchboard.

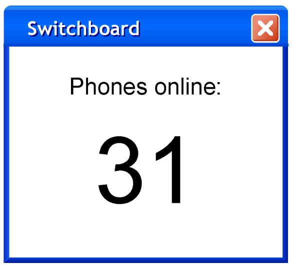


Figure 3.8 *User interface prototype for the switchboard application*

3.3 Distributed phone clients

Now that the fundamental design decisions for the overall system architecture and the centralised switchboard application have been presented, the focus of this section can move to the phone application which will build on the techniques mentioned earlier.

3.3.1 Typical phone operations and behaviour

The phone application will be modelled on a typical phone from the real-world and will allow a user to perform basic functions, such as make outgoing calls, receive incoming calls, take part in one-to-one conversations and terminate calls.

The phone application will inevitably be more complex than the switchboard, as two phones must interact with each other during a call. Phone-to-phone connectivity cannot be modelled using a simple client/server paradigm as a phone can be both a client (when it is making an outgoing call) and a server (when it is accepting an incoming call). To understand this interactivity it is necessary to identify the discrete states that can be associated with a phone. Table 3.1 below lists the possible states that a phone may be in.

State	Explanation
Ready	The phone is ready to make an outgoing call and receive incoming calls
Making an outgoing call	The phone has placed an outgoing call but the intended recipient hasn't answered
Engaged in an outgoing call	The phone has placed an outgoing call, the recipient answered and a conversation is active
Have a pending incoming call	The phone has an incoming call pending but hasn't responded to it
Engaged in an incoming call	The phone answered an incoming call and a conversation is active

Table 3.1 ***A list of states that a phone can be in***

In each of the above states there are a number of operations that a user can perform depending on the state that their phone is in. These are highlighted in table 3.2 below.

State	Valid actions
Ready	Make an outgoing call Accept an incoming call Remove phone from network
Making an outgoing call	Cancel the call
Engaged in an outgoing call	Chat Terminate the call
Have a pending incoming call	Reject the call Accept the call
Engaged in an incoming call	Chat Terminate the call

Table 3.2 ***States and their associated valid actions***

The phone application will need to model the state and behaviour outlined in tables 3.1 and 3.2 carefully. State information will need to be maintained and fully reset when transitioning back to the ready state, ensuring that a phone cannot enter an unknown state or perform invalid operations.

3.3.2 Outgoing calls

Figure 3.9 below illustrates the process of making an outgoing call.

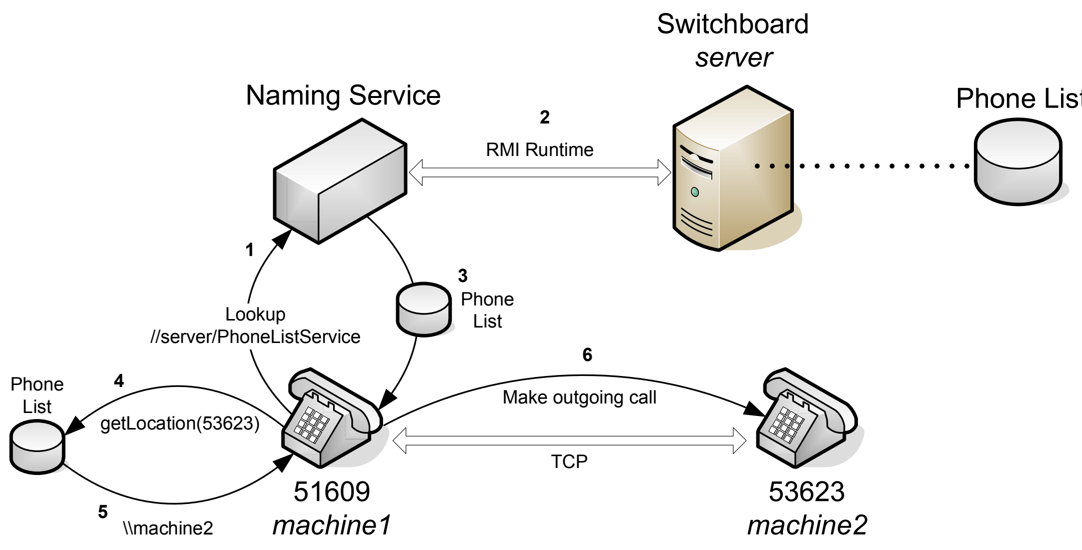


Figure 3.9 ***Making an outgoing call***

1. Phone 51609 on machine1 wishes to call phone 53623 but does not know its location on the network. It issues a lookup request to the naming service.
2. The naming service resolves the request into the remote phone list object hosted on the switchboard.
3. The switchboard skeleton returns the phone list object to the stub on the client phone. Remember that this happens transparently as it is handled by the RMI runtime.
4. The client phone performs a getLocation(53623) method call on the phone list to obtain the location of phone 53623.
5. The phone list returns the specific network location (machine2) of phone 53623.
6. 51609 opens a TCP socket connection to 53623 and makes an outgoing call.

3.3.3 Incoming calls

Outgoing calls are predictable in that they are initiated by the user when they click on a “call” button. Incoming calls are rather different in nature, largely due to the fact that a user cannot predict when they will receive an incoming call. Each phone client will need to actively listen for incoming calls and deal with them appropriately, i.e. either accept or reject them.

To successfully obtain incoming connections, a given phone must listen on a specific port number. All other phones on the network need to know what port this phone is listening on to enable them to make a direct socket connection to the correct port on the machine that is hosting the phone application. This has great implications for the entire phone network and there are two possible approaches to solving this problem.

The first is to use a fixed port number and have all phones listening on the same port, 4444 for example, and hard-wire this value into the application code. The problem with this approach is that other applications running on the host machine may be configured to use port 4444 which would cause a conflict and mean that the user would not be able to use their phone application unless they freed up port 4444.

The second approach is to have each phone application use any available port on the host machine. The problem here is that all other phones on the network would need to be notified of the port number that the phone is listening on. Also, this port number would be likely to change each time the user starts their phone application and registers it on the network. However, this method does provide an advantage in that it does not give rise to conflicts between applications competing for the same port number.

3.3.4 The modified Phone object

The chosen solution to the port number problem was the second approach discussed above; having each phone listen on any free port on the host machine. The most obvious way of informing all other phones of the port number that a given phone is listening on is to include the port number in the phone object and store it in the phone list. This means that whenever a phone makes an outgoing call, it should obtain the precise network location of the recipient and the port number it is listening on from the switchboard. Figure 3.10 below shows the modified Phone object with a port number field added.

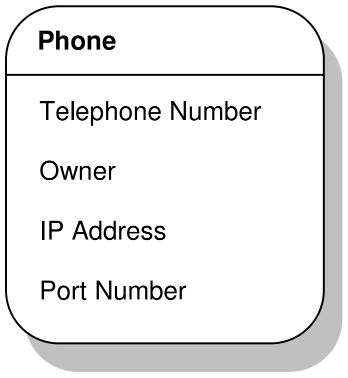


Figure 3.10 ***The modified Phone object***

3.3.5 Fault-tolerance

The fault-tolerant broadcast mechanism of the switchboard was discussed in section 3.2.2. To support the relocation of the switchboard in the event of a failure, each phone client must be capable of receiving the broadcast messages and updating the location of the switchboard if necessary.

3.3.6 Multi-threading issues

Each telephone client will need to be capable of performing multiple operations simultaneously. Two operations that must run continuously have already been identified: listening for incoming calls from other phones and listening for broadcast messages from the switchboard. These tasks must run concurrently, as an incoming call could occur at the exact moment that a broadcast message is received from the switchboard. It is envisaged that a number of other tasks will also need to run concurrently. Semaphores will be used to control access to objects in an effort to make the application thread-safe.

3.3.7 Automated telephone book

Once the basic phone application is complete, a dynamic telephone book will be added to the system. The telephone book must maintain an up-to-date list of all the phones that are connected to the network, correctly adding entries for phones that come online and removing entries for those that disconnect from the network. This process must be handled transparently from the user and must not consume too many resources on either the client or the server.

When a phone initially comes online, it can obtain a list of all the phones connected to the network from the phone list. What it cannot do is predict when other phones will come online after it has obtained the data from the phone list. Each phone could poll the switchboard for the list of online phones at regular intervals, say every 10 seconds, but this would place considerable demand on the switchboard. A more practical approach is to have phones broadcast information about themselves to all other phones when they connect to the network for the first time. Existing phones on the network could receive the broadcasts and update their telephone book whenever a phone connects to or disconnects from the network. This prompts the need for another listener object to be concurrently active in the application background.

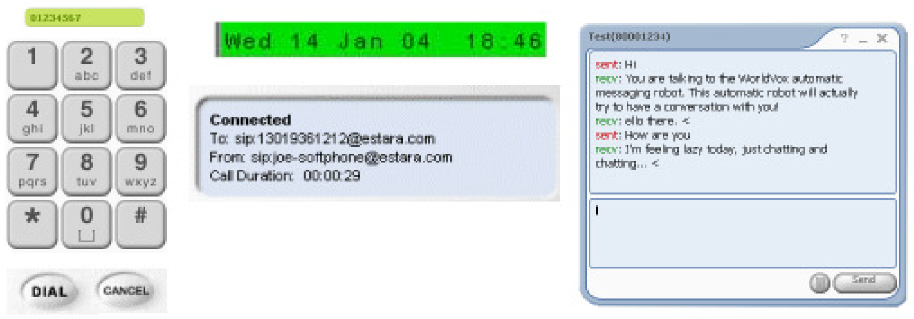
It is important to note that the telephone book is to be an addition to the basic system so that users can easily locate a particular phone on the network when there are a large number of phones online. Calls should still be able to be made without the telephone book, by directly dialling the recipient's number.

3.3.8 Graphical user interface

Section 2.9 outlined some existing VoIP solutions and presented three examples of existing soft phone applications: Firefly, eStara and LIPZ4. One of the objectives set out in section 1.1 was to provide each phone client with a friendly graphical user interface and a look-and-feel that resembles a typical phone from the real-world. The three soft phone examples were studied and both positive and negative aspects of their user interfaces were noted.

The Firefly VoIP soft phone, described in section 2.9.1, had a large, clear number pad and a basic chat window for displaying text-based conversations. The application also featured a dial display showing what numbers had been keyed-in. eStara Softphone outlined in section 2.9.2 incorporated a useful status window showing information about the current call and clear buttons for starting and ending calls. LIPZ4 SIP Soft Phone boasted a small but useful panel indicating the current date and time.

Despite the fact that all of these features were not present in a single application, they were thought by the writer to be desirable for inclusion in a model of a classic phone from the real-world. Figure 3.11 below shows snippets of all three user interfaces joined together to form a complete application.



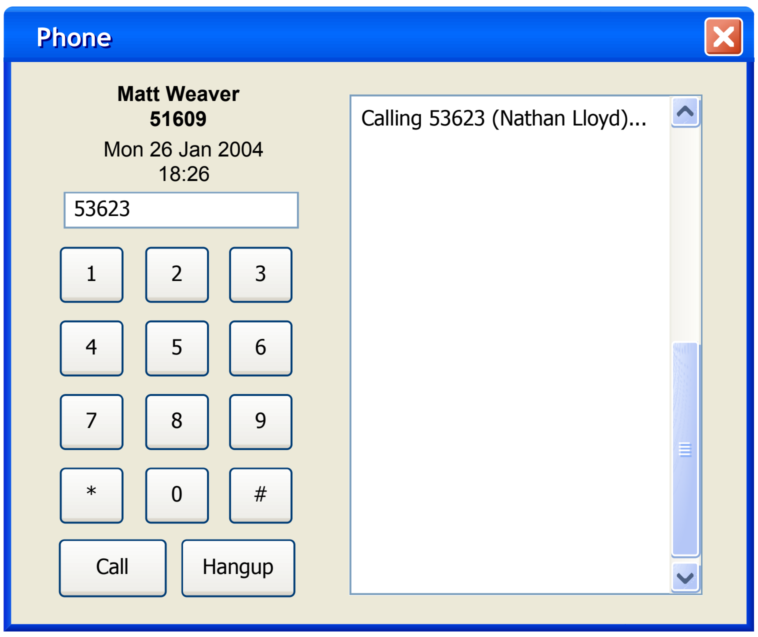
Wed 14 Jan 04 18:46

Connected
To: sip:13019361212@estara.com
From: sip:joe-softphone@estara.com
Call Duration: 00:00:29

sent: Hi
recv: You are talking to the WorldVox automatic messaging robot. This automatic robot will actually try to have a conversation with you!
recv: ello there. <
sent: How are you
recv: I'm feeling lazy today, just chatting and chatting... <

Figure 3.11 ***Noteworthy characteristics of three soft phone applications joined together to form a complete application***

Figure 3.11 above looks very messy as the individual components that make up the interface are all disjoint in their appearance. Figure 3.12 below shows a more consistent user interface prototype.

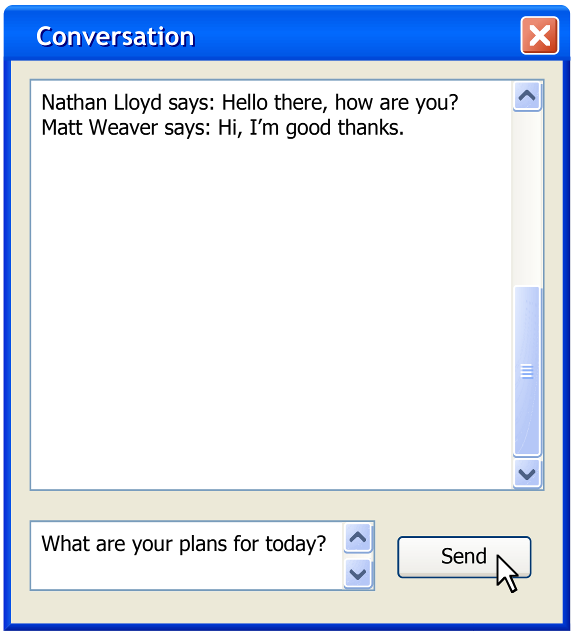


Matt Weaver
51609
Mon 26 Jan 2004
18:26
53623

Calling 53623 (Nathan Lloyd)...

Figure 3.12 ***User interface prototype for the phone application***

Figure 3.12 above presents a very basic interface for the phone application. All the desirable components from the three example soft phones have been included and streamlined to simplify the overall layout. Figure 3.13 below shows a user interface prototype for the conversation window that will be used to exchange messages between users.



Nathan Lloyd says: Hello there, how are you?
Matt Weaver says: Hi, I'm good thanks.

What are your plans for today?

Figure 3.13 ***User interface prototype for the conversation window***

The design for the chat window is based on that from Firefly's VoIP Soft Phone but also exhibits characteristics from Msn Messenger, namely the addition of the friendly "user says:" remarks at the beginning of messages to clearly identify the two parts of the conversation.

3.4 Summary

This chapter began by introducing the overall client/server architecture of the telephone system. The need for point-to-point connections between individual phones was justified and the notion of a centralised list storing details of all phones on the network was put forward. Several security issues were addressed and the point-to-point architecture of the system was compared to that of Needham and Schroeder's secret key authentication protocol. The benefits of a non-switched network model were touched upon and a Phone object to encapsulate all of the information associated with a given phone was created. The discussion moved on to talk about the communication requirements of the system, where decisions were made to use the Transmission Control Protocol for phone-to-phone connectivity and Java Remote Method Invocation for phone-to-switchboard connectivity. A detailed overview of RMI was presented and the concept of an RMI naming service was introduced.

Next, important design decisions for the centralised switchboard were put forward. Switchboard functionality was realised as a remote phone list object registered with the naming service. Techniques to provide a level of fault-tolerance within the system were discussed and a broadcast mechanism was chosen to relay the location of the switchboard to all phones on the network at a regular time interval. The need for the switchboard to handle simultaneous requests was explained and a user interface prototype was drawn up. The final section in this chapter presented important design considerations for the client phone application that will be distributed throughout the network. Permissible phone operations and behaviour were outlined in tables and the process of making an outgoing call was depicted in a diagram. The unpredictability of incoming calls was noted and contrasted to the controlled initiation of outgoing calls. A problem was identified with listening for incoming calls and consequently solved by having each phone listen on any available port number on the host machine. This presented further problems which were addressed by the addition of a port number field in the Phone object. The need for every phone to support the fault-tolerant properties of the switchboard was mentioned and multi-threaded issues were again considered. Finally, the implications of adding an automated telephone book were discussed and a simple graphical user interface was prototyped.

Chapter 4 continues the work by showing how the fundamental concepts detailed here were implemented into a basic working model of a telephone system.

Chapter 4. Implementation : The Basic Phone System

This chapter shows how the important design decisions made in chapter 3 were realised in an object-oriented model of a basic telephone system. The chapter starts by looking at the structure of the centralised switchboard and the distributed phone client applications, presenting various UML diagrams and explaining the key functionality provided by each of the classes in the system.

4.1 Centralised switchboard

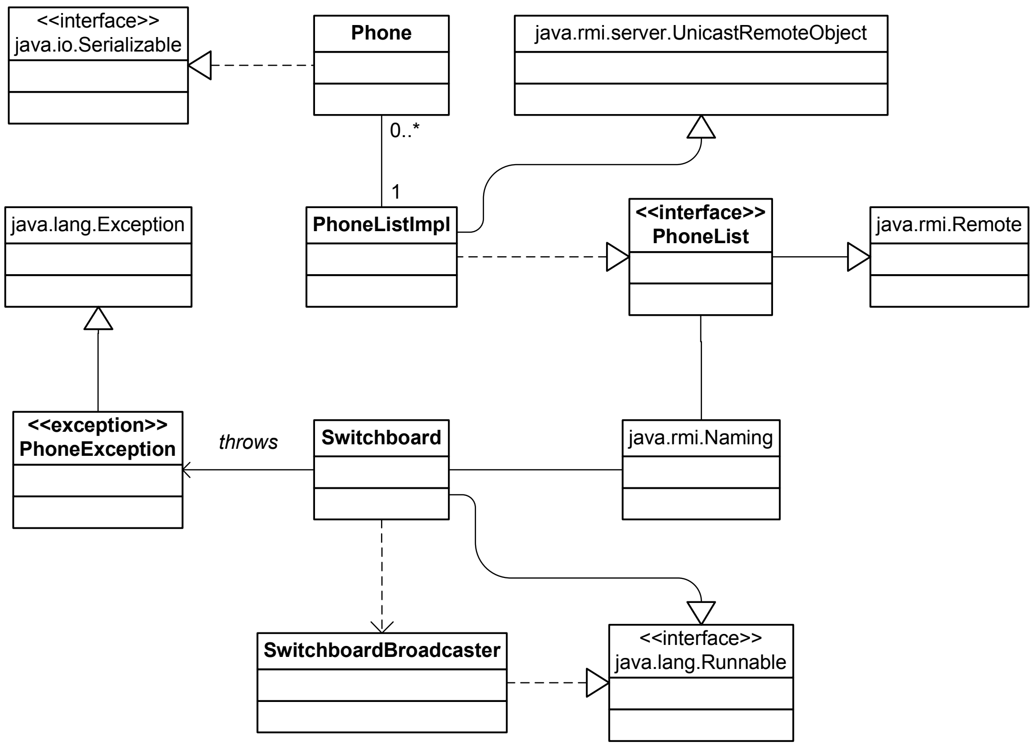


Figure 4.1 ***UML class diagram for the centralised switchboard***

Figure 4.1 above depicts a UML class diagram for the server-based switchboard. The classes shown in **bold** are those that were written specifically for the switchboard system, whilst classes in plain text are those provided by the Java API.

4.1.1 Switchboard

The switchboard application is hosted on one specific machine on the network dedicated as a server. This machine should also be hosting an instance of the RMI registry (rmiregistry.exe) on a known port number, usually 1099. The Switchboard class contains a main method so that the application can be launched from a command window. It should be noted that Switchboard also extends the java.awt.Frame class and implements the java.awt.event.ActionListener and java.awt.event.WindowListener interfaces. These details were omitted from the diagram to avoid unnecessary complexity.

When the switchboard comes online for the first time, it creates a new instance of the PhoneListImpl class and binds it to the registry using *//server/PhoneListService* as the service name. *server* represents the host name of the machine hosting the switchboard which is dynamically discovered at run-time. The switchboard then creates a new instance of the SwitchboardBroadcaster class and starts it running as a new thread in the background.

The switchboard application also provides a simple user interface showing the number of phones online within the network. When the switchboard is launched from the command window, an instance of Switchboard is created and started as a separate thread. The run() method executes continuously in the background, querying the PhoneListImpl object for the number of phones online and updating the display accordingly. To make this task more efficient, the thread is told to sleep for 250 milliseconds so that vital resources are not wasted on a non-critical operation.

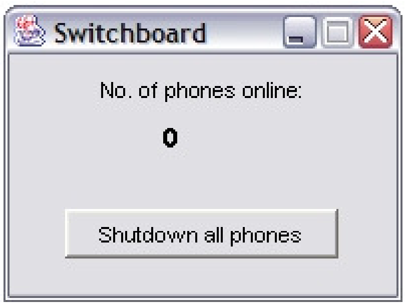


Figure 4.2 *Switchboard GUI*

The switchboard GUI is presented above in figure 4.2. A “shutdown all phones” button was added to greatly simplify the process of shutting down all the phones on the network, as up to 8 phones were hosted simultaneously on a single machine during the development stages.

4.1.2 SwitchboardBroadcaster

The SwitchboardBroadcaster class was created to perform the operations associated with the fault-tolerant properties of the switchboard. In the design section, it was proposed that the switchboard broadcast its location out to all phones on the network at regular intervals of 5 seconds.

The broadcaster is based on the principal of IP multicast. The location of the switchboard is encapsulated into a UDP packet and sent out to a specific multicast group using a multicast address. Multicast addresses will never clash with global host addresses as a portion of the IPv4 address space in the range 224.0.0.0 to 239.255.255.255 is reserved for multicast traffic (*http://ntrg.cs.tcd.ie/undergrad/4ba2/multicast/antony/*). The chosen multicast address was 230.0.0.1 which is well within this range.

An InetAddress object is used to store the address of the switchboard and is passed into the SwitchboardBroadcaster constructor, along with a reference to the switchboard object itself. The host name is obtained from the InetAddress object and converted into a byte array. This array is then packaged into a DatagramPacket and sent out over a DatagramSocket to multicast address 230.0.0.1 on port 4446. Port 4446 is unlikely to be in use on a typical machine as ports greater than 1000 are usually not reserved by the operating system.

The broadcaster supports three operations. It can:

- multicast the location of the switchboard every 5 seconds to all phones on the network,
- multicast a single message containing the location of the switchboard and a request for each phone to return its location on the network, and
- multicast a single "shutdown" message to remove all phones from the network.

The second operation highlighted above only occurs when the switchboard relocates to a different machine (or if it is shutdown and restarted on the same machine) and serves as a mechanism for dynamically rebuilding the phone list.

Each operation is identified by prefixing a string identifier onto the beginning of the switchboard's host name before it is converted into a byte array. When a normal broadcast is being sent, "location:" is applied as the prefix. When the broadcaster runs for the first time, i.e. immediately after the switchboard has been relocated, a prefix of "re-send:" is concatenated onto the beginning of the host name. When a shutdown message is to be broadcast, a single "shutdown" message is sent without a corresponding host name.

4.1.3 PhoneList (interface)

This interface extends java.rmi.Remote and exists to comply with standard conventions that have been established in the world of distributed systems programming and RMI. The jGuru tutorial on RMI (*http://java.sun.com/developer/onlineTraining/rmi/RMI.html*) reminds us that interfaces are used to define behaviour and classes are used to define particular implementations of the required behaviour. Warren (2003) states how "objects should be known by the interfaces they provide and not their implementation classes".

Programming to an interface provides many benefits, namely the flexibility to substitute an implementation of an interface at run-time. When a remote interface is provided, client applications need only know about the interface definition and not care for the specific implementation details. All the necessary logic can be contained at the server, allowing the functionality to be extended at a later date without having to modify the client code.

PhoneList defines the operations needed to add phones to the network, remove phones from the network, return the entire list of phones on the network, obtain the specific location of a given phone, determine the number of phones connected to the network and perform broadcast capabilities that will be required to support the dynamic telephone book.

4.1.4 PhoneListImpl

The PhoneListImpl class is a direct implementation of the PhoneList interface immediately above and implements the remote phone list service that was discussed extensively in the design chapter. The class extends java.rmi.server.UnicastRemoteObject which automatically manages most of the tasks associated with creating a server, such as opening a socket and listening for incoming client requests. PhoneListImpl is also associated with the Phone class as it is capable of storing zero or more Phone objects.

A Vector object is used to store an unordered collection of Phone objects within the PhoneListImpl. The functionality provided by the remote phone list service is an integral part of the telephone system and it is necessary to outline the methods provided in this class and document the operations that they perform.

addPhoneToNetwork(Phone phone, boolean performBroadcast)

This method adds a new phone to the network.

A check is first performed to determine if a Phone object with the same telephone number as that of the phone object passed into the method exists in the vector of online phones. If it does, a PhoneException is thrown with a suitable error message informing the user that a phone with the same telephone number already exists on the network. The IP address and host name of the machine hosting the phone are also included in the message to provide useful troubleshooting information.

If the phone being added does not already exist on the network then it is added to the vector of online phones. If the performBroadcast boolean is set to true, a call is made to the broadcastPhoneToNetwork(phone) method, passing the new phone object in as a reference.

removePhoneFromNetwork(Phone phone, boolean performBroadcast)

This method removes an existing phone from the network.

The vector of online phones is searched until a Phone object is found that has the same telephone number as the phone object passed into the method. This object is then removed from the vector. If the performBroadcast boolean is set to true, a call is made to the broadcastRemovePhoneMessageToNetwork(phone) method, passing in as a reference the phone object that is to be removed from the network.

getPhoneList()

This method simply returns the Vector object containing Phone objects for all the phones online within the network.

getNumberOfPhonesOnline()

This method returns, as an integer, the total number of phones that are currently connected to the network.

getPhoneLocation(Phone phone)

This method returns a complete Phone object from the vector of online phones.

Client phones call this method whenever an outgoing call is made, passing in a minimal Phone object with only the telephone number field set. The remaining fields are all set to null, as the owner of the targeted phone, its unique location on the network and the port number it is listening for incoming calls on are unknown by the calling phone.

The vector of online phones is searched to find a Phone object with a matching telephone number. If a match is found, the complete phone object (containing the owner name, the IP address of the phone on the network and the port number it is listening for incoming calls on) is returned to the calling phone, otherwise a PhoneException is thrown.

broadcastPhoneToNetwork(Phone phone)

Whenever a phone is added to the network this method broadcasts a message (containing an identifier and a byte representation of the phone's Phone object) out to all other phones on the network. The method is needed to support the dynamic telephone book so that existing phones can add entries for phones that subsequently join the network.

In essence, a byte array is generated containing only the byte representation of the character "n" (to signify that this is a new phone). The phone object that is added to the network is passed in and converted into a separate byte array. The two arrays are encapsulated into separate UDP packets and multicast out to address 230.0.0.2 on port 4447. Two UDP packets were used to simplify the process of determining the new location of the phone at the client side. A different multicast address and port number were used to prevent any possible clash with the multicasting process used by the SwitchboardBroadcaster.

broadcastRemovePhoneMessageToNetwork(Phone phone)

This method broadcasts out a message to all other phones on the network whenever a phone is removed from the network. The method serves to inform the telephone book of the phone that is being taken offline so that all other phones on the network can remove its entry from their listing.

Two byte arrays are generated, one containing the byte representation of the character "r" (to signify that this phone is being removed from the network) and the other containing the byte stream that makes up the phone object being removed from the network. Again, both these arrays are packaged into separate UDP packets and multicast out to address 230.0.0.2 on port 4447.

4.1.5 Phone

The Phone object, as discussed in the design section, contains 4 private fields for encapsulating all of the information associated with a given phone on the network. Figure 4.3 below shows the conceptual model of the Phone object and its realisation as a Java class.

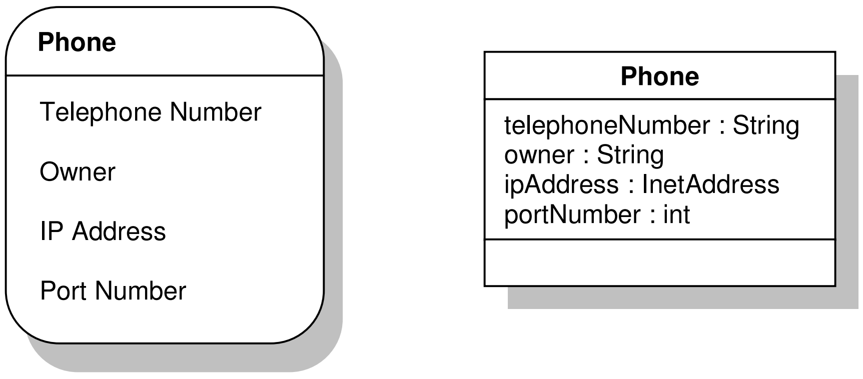


Figure 4.3 *The Phone object, pictured as both a conceptual object and as a Java class*

The Phone class provides 4 constructors to allow new Phone objects to be created with varying levels of detail. A full collection of get and set methods are also provided, as well as an important toString() method that will be realised to its full potential when the dynamic telephone book is discussed in sections 4.2.20 and 4.2.21.

4.1.6 PhoneException (exception)

This is a simple exception that extends java.lang.Exception and provides a means of presenting the user with useful troubleshooting information should a problem arise when performing a remote operation associated with their phone, such as adding it to the network or resolving their phone number into an IP address.

4.2 Distributed phone client

Although a large amount of time was dedicated to explaining the important design decisions surrounding the overall architecture of the telephone system, the majority of the complexity involved was in implementing the phone application. A large number of classes make up the application and it would be difficult to fit them all into a single diagram. Instead, a series of smaller UML diagrams will be presented to show how different areas of the system were implemented.

This section takes a bottom-up approach by first explaining the basics of the phone application and its user interface, then moving on to show how telephony functionality was incorporated into the application and finally demonstrating the mechanics behind the automated telephone book.

Figure 4.4 below shows a UML diagram defining the PhoneUI class which is the central phone application. Each instantiation of PhoneUI is responsible for managing all the associated state and behaviour of a given phone on the network.

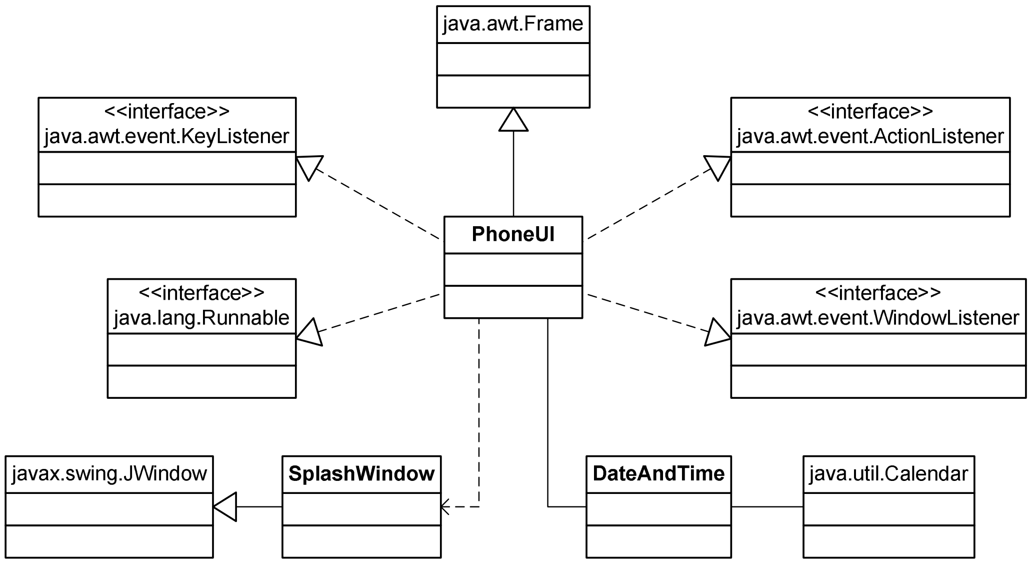


Figure 4.4 *UML diagram showing the user interface structure of the PhoneUI application*

4.2.1 PhoneUI

The PhoneUI application is the program that a user will invoke when they wish to host a soft phone on their desktop. PhoneUI is responsible for:

- setting up the user interface of the application,
- handling a user's interactions with the interface,
- maintaining the state and behaviour of the phone, and
- managing basic telephony operations, such as initiating and tearing-down calls.

To make the application code more manageable, some of these tasks are handled indirectly by other classes. In such cases, a reference to the PhoneUI object is usually passed into the constructor of the child class. These classes will be discussed later in this section.

The graphical components of the user interface were based on those available in the Java Swing API. It can be seen in figure 4.4 that PhoneUI extends the java.awt.Frame class and not the javax.swing.JFrame class. This decision was made to slightly simplify the process of adding components to the application. ActionListener and KeyListener interfaces were implemented to allow the application to detect clicks on buttons and certain key presses on the keyboard, mainly the numbers 0 through 9 on the number pad.

PhoneUI contains a main method to allow the user to invoke the application directly from the command line (or from an icon on their desktop using the *javaw* command which launches a Java virtual machine without an associated terminal window). The class also implements the Runnable interface for the sole purpose of providing a run() method that automatically updates the date and time display every second.

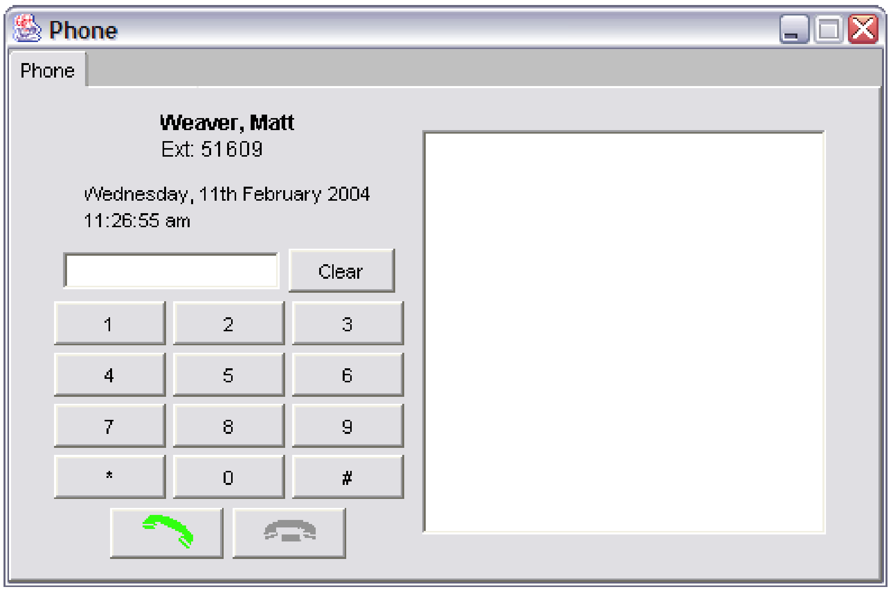


Figure 4.5 ***The basic user interface of the PhoneUI application***

Figure 4.5 above depicts the initial user interface of the PhoneUI application. A JTabbedPane was used to separate the three primary concerns of the application: the basic phone application, the dynamic telephone book and specific telephony features implemented as aspects. The latter two tabs will appear later on in the document when the implementation of the telephone book and aspects is covered.

4.2.2 DateAndTime

The DateAndTime class uses an instance of the java.util.Calendar class to obtain the date and time from the system clock. It contains two void methods, makeDateString() and makeTimeString(), that compute the current date and time and format the values appropriately into two separate strings. These strings are encapsulated within the class and can be returned by calling the corresponding getDate() and getTime() methods.

4.2.3 SplashWindow

The SplashWindow class was added towards the end of the development stage to produce a splash screen on the centre of the display whilst the phone application is loading. The code required to generate the window was copied directly from an article on the JavaWorld website (*http://www.javaworld.com/javaworld/javatips/jw-javatip104.html*) and modified slightly to meet the specific needs of the application.

Figure 4.6 below shows the classes that were implemented to support the main functionality of the phone application and their associated dependencies.

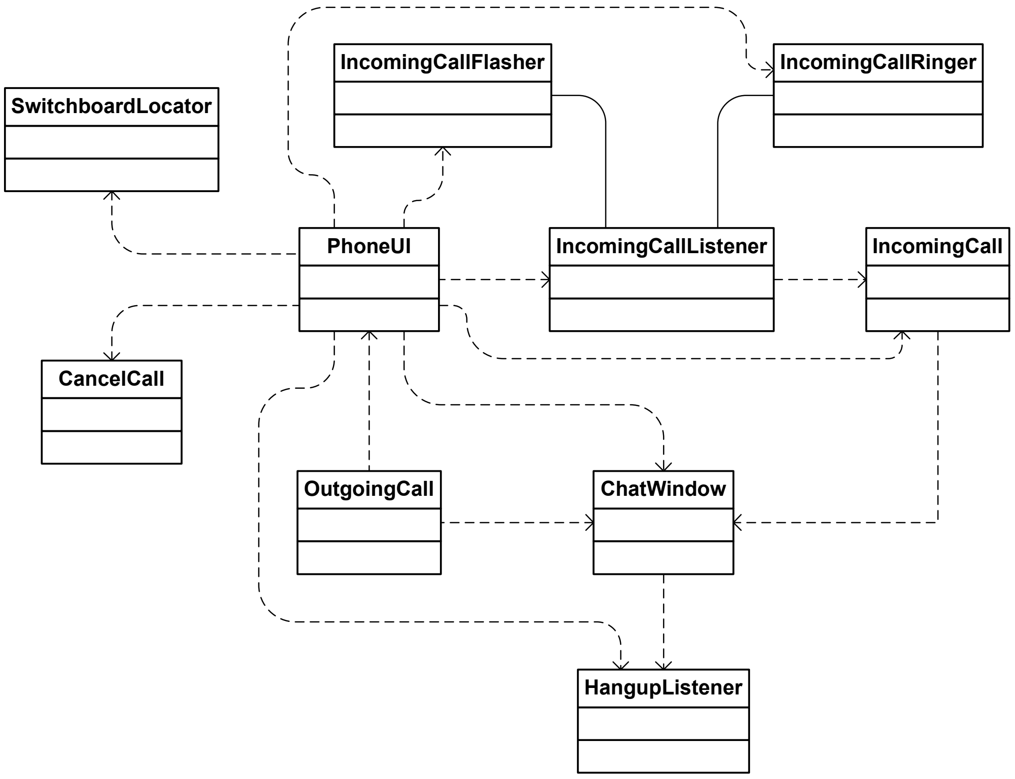


Figure 4.6 *UML diagram showing the classes that make up the main functionality of the phone application*

All the classes shown in the above diagram implement the java.lang.Runnable interface to allow them to run as separate threads in the background of the main phone application. The specifics of the threading will be explained in the relevant sections below.

4.2.4 PhoneUI

The PhoneUI application is started from the command window. Three arguments must be specified when launching the program:

- the phone number or extension number of the phone,
- the forename of the phone's owner, and
- the surname of the phone's owner.

The main method extracts these arguments from the command line and encapsulates them into a new Phone object called thisPhone, along with an InetAddress object representing the phone's unique location on the network. A new instance of the PhoneUI class is started as a thread and new instances of the SwitchboardLocator and IncomingCallListener classes are started by the PhoneUI constructor as background threads. Execution through the rest of the constructor method is halted until the IncomingCallListener has successfully bound itself to an available port and until the location of the switchboard has been identified by the SwitchboardLocator.

Throughout the lifetime of a phone application a user is likely to make a large number of outgoing calls. Outgoing calls are handled as a separate thread by the OutgoingCall class.

4.2.5 SwitchboardLocator

The role of the SwitchboardLocator class is to listen for incoming multicast messages that are broadcast from the switchboard's SwitchboardBroadcaster class. To do this, the SwitchboardLocator opens a new MulticastSocket on port 4446 and joins the multicast group with address 230.0.0.1. The locator then sits in a while loop for the rest of its lifecycle, listening for incoming DatagramPackets and dealing with them accordingly.

Every packet received on port 4446 that is bound for multicast address 230.0.0.1 is examined, paying particular attention to the string identifier that is prefixed to the switchboard's location. If the identifier is found to be "shutdown", the phone makes a call to removePhoneFromNetwork() on the remote phone list to remove the phone from the network and subsequently calls System.exit(0) to close the application.

If the identifier is not "shutdown", the host name of the switchboard is extracted from the packet and used to generate a new InetAddress object that represents the location of the switchboard. This object is stored in the PhoneUI class as a public global variable. The identifier is then checked to see if it is equal to a "re-send" message. If so, the phone calls addPhoneToNetwork() on the phone list to allow the newly relocated switchboard to rebuild its knowledge of online phones. If the locator is running for the first time, it signals the PhoneUI object telling it that it has successfully located the switchboard.

4.2.6 OutgoingCall

An instance of the OutgoingCall class is started as a new thread whenever a user starts a call by pressing the appropriate button in the user interface. OutgoingCall constructs a new Phone object, phoneToCall, containing only the phone number of the phone that is to be called. A lookup request is performed to obtain a reference to the remote phone list object and a call to getPhoneLocation(phoneToCall) is made on the phone list. A complete Phone object is returned for the phone being called, assuming that it exists on the network.

OutgoingCall then creates a direct socket connection to the intended recipient using a Java Socket. A "begin call" message is sent over the socket along with the thisPhone object that represents the calling phone. A message is received from the target phone stating whether it accepted the call, rejected the call or whether it was engaged in another call. If the phone was engaged, or if the call was rejected, then the state of the calling phone is reset. If the call was accepted then a new ChatWindow instance is created and started as a thread. The OutgoingCall thread then waits on a Semaphore object that is only released when the ChatWindow is closed.

4.2.7 CancelCall

This class was created to deal with the situation where a user places an outgoing call and cancels it before the other user has a chance to answer it. In essence, it passes a "cancel call" message to the phone it is calling. This is received by the remote phone's

IncomingCallListener which processes the message and stops the remote phone from ringing.

4.2.8 IncomingCallListener

This class is a listener that runs continuously in the background of the phone application. It listens for incoming calls by opening a ServerSocket on a specific port number that is discovered dynamically at run-time and registered in the thisPhone object. Incoming connections to this port are accepted in the form of Sockets and comprise a single message, either “begin call” or “cancel call”. If the incoming connection is the start of a call, the listener starts instances of IncomingCallFlasher and IncomingCallRinger as threads to alert the user that there is an incoming call. An instance of IncomingCall is then started as a thread to deal with setting up and managing the call.

4.2.9 IncomingCall

An instance of the IncomingCall class is started as a thread by the IncomingCallListener. The ObjectOutputStream and ObjectInputStream objects from the IncomingCallListener are passed into the constructor, allowing IncomingCall to read in the Phone object that was sent across the network by the OutgoingCall thread in the calling phone.

There are three possible conditions that can arise when a phone receives an incoming call: the user can accept the call, reject the call or the calling phone can cancel the call before the user responds to it. This class determines which of these conditions has occurred by comparing the values of boolean variables stored within the PhoneUI instance. If the user accepted the call then an instance of ChatWindow is started as a thread. As with the OutgoingCall class, the IncomingCall class waits on a semaphore until it is released by the closing of the chat window.

4.2.10 IncomingCallRinger

This is a very simple class. It contains a while loop that plays a ringing sound to alert the user that they have an incoming call. The ringing sound is stored on disk as a WAV file and loaded into memory in the form of a Clip object. More information can be found on the Java Sound API at Sun Microsystems website (*http://java.sun.com/products/java-media/sound/*).

4.2.11 IncomingCallFlasher

This is another simple class that runs as a thread and changes the background colour of the phone’s display area to green, and back to white again, every second until the user responds to an incoming call.

4.2.12 ChatWindow

The ChatWindow class provides a simple user interface to allow two users to take part in a conversation using short, text-based instant messages. This is shown in figure 4.7 below.

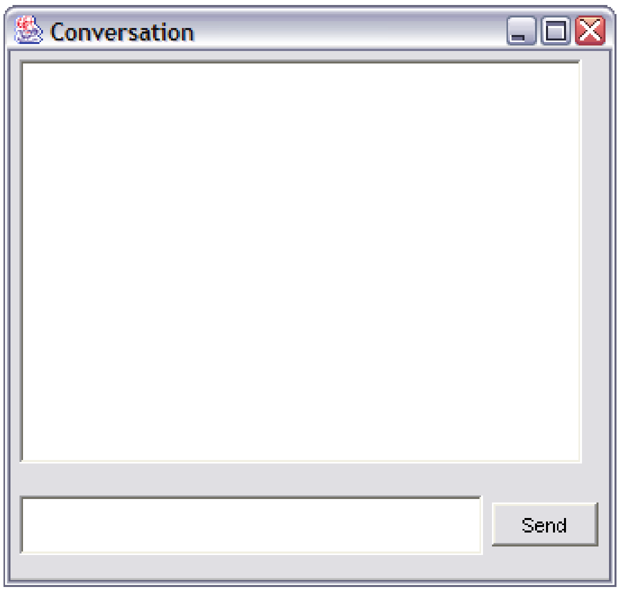


Figure 4.7 *The implemented ChatWindow user interface*

The ChatWindow class has two main roles:

- to allow the user to read messages from another user and type a reply to them, and
- to manage the sending and receiving of messages between two remotely connected phones.

The first role is supported by the graphical user interface components that are initialised in the constructor method. ChatWindow implements the ActionListener, WindowListener and KeyListener interfaces to add support for user interaction, allowing a user to initiate the transfer of a message by clicking on the "send" button or by simply pressing the return key.

The second role of managing the sending and receiving of messages between phones is realised in two separate areas of the class. The sending of messages occurs when a user clicks on the send button, where any text in the message window is stored in a string and sent across the socket connection to the remote phone. The receipt of incoming messages is performed by a run() method that executes continuously in the background. The thread continuously reads in a String object from the ObjectInputStream of the socket connection.

The ChatWindow class also contains some complicated code that determines if the user at the other end of the connection has closed their chat window, which is supported by the HangupListener class discussed on the next page.

4.2.13 HangupListener

The HangupListener class contains a single run() method that waits on a semaphore to be released by the phone application when a user ends a call. When the semaphore is signalled by the PhoneUI class, a method within the ChatWindow class is called to inform the remote user that the conversation has been terminated and to close the chat window.

Figure 4.8 below shows the four remaining classes used by the PhoneUI class that make up the basic phone application.

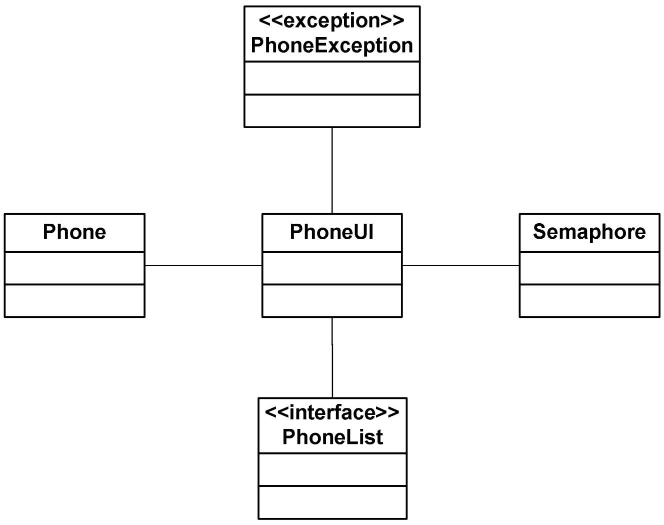


Figure 4.8 *UML diagram showing fundamental classes*

4.2.14 Phone

This is the same Phone class that was defined for the switchboard in section 4.1.5. It is also needed on the client side so that a phone can encapsulate information about itself into a Phone object and pass it to the remote phone list when it is added to the network.

4.2.15 PhoneException (exception)

Again, this exception class is needed at the client side to correctly handle PhoneExceptions that are thrown by the PhoneListImpl class.

4.2.16 PhoneList (interface)

The PhoneList interface described in section 4.1.3 is replicated on the client side so that the phone application can call any of the methods that are defined by the interface. The details of the method implementations are stored at the server side and are made available transparently by the RMI runtime. The only changes that would need to be made to this interface are when additional methods are provided by the switchboard server.

4.2.17 Semaphore

This is a very simple implementation of a Semaphore class. It consists of two methods, P() and V() that perform the necessary wait and signal operations respectively. Both methods P() and V() are synchronized to ensure that the operations are atomic and thread-safe.
Figure 4.9 below presents a UML diagram showing how the dynamic telephone book was integrated into the phone application.

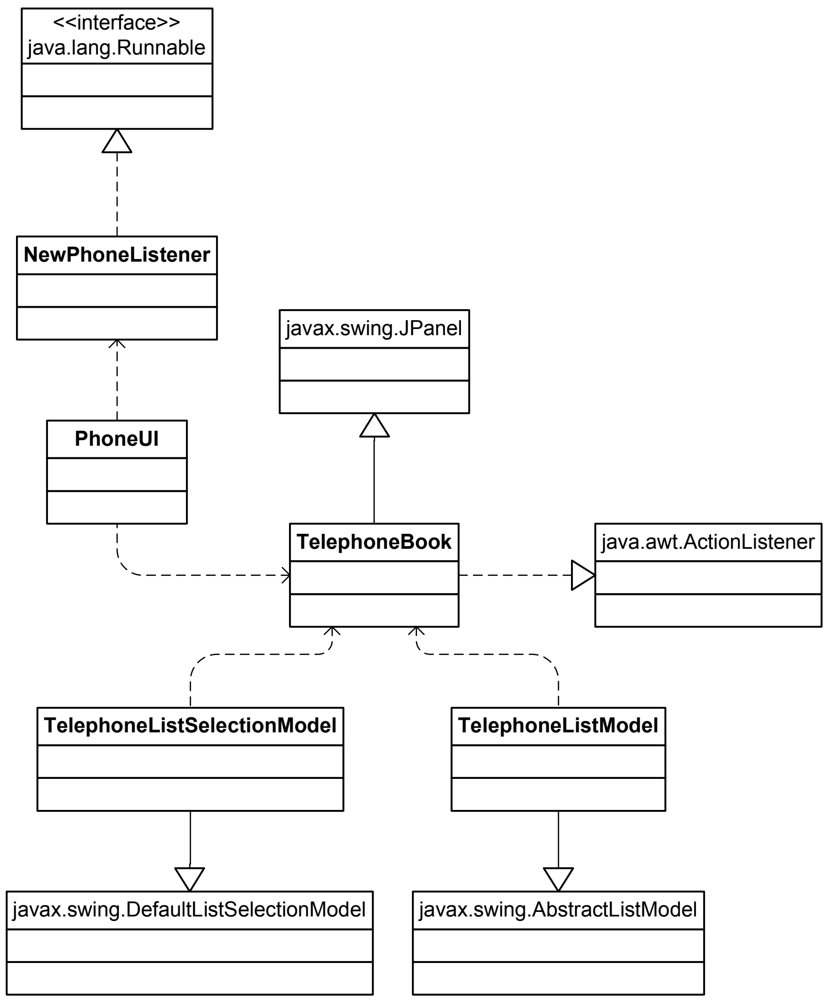


Figure 4.9 *UML diagram for the dynamic telephone book*

4.2.18 PhoneUI

The PhoneUI class starts a NewPhoneListener instance as a thread which runs continuously in the background of the phone application. PhoneUI also maintains a local cache of all the phones that are online within the network. The cache is initially obtained from the phone list when the phone is registered online. Phones that subsequently come online are added to the cache by the NewPhoneListener.

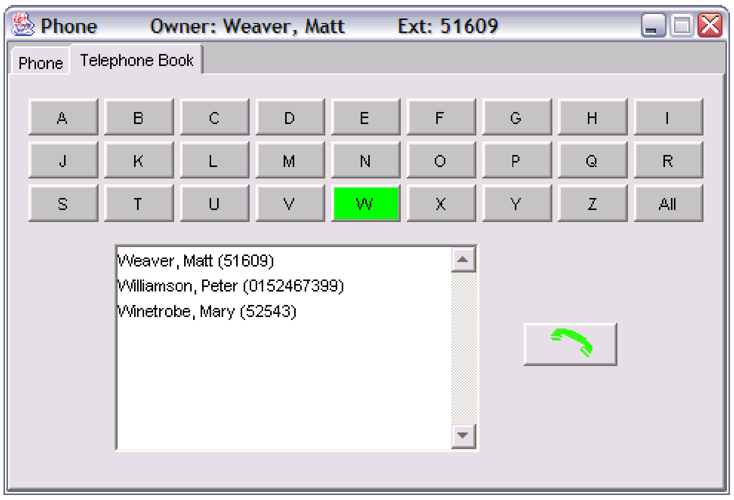
4.2.19 NewPhoneListener

The NewPhoneListener runs as a continuous thread to listen for incoming messages about phones that are being added to and removed from the network, updating the phone's local cache as appropriate.

The listener opens a new MulitcastSocket on port 4447 and joins the multicast group with address 230.0.0.2. The listener then operates in a similar way to the SwitchboardLocator class, sitting in a while loop and constantly listening for incoming DatagramPackets. Each packet is examined to determine whether it contains a byte representation of the character "n" (to signify that a new phone has come online) or "r" (to indicate that a phone is being removed from the network). Another packet is obtained from the network and converted into a Phone object which is either added to or removed from the local phone cache, depending on the value of the identifier ("n" or "r"). To prevent the NewPhoneListener thread from consuming too many resources, a two second sleep command is introduced at the end of each while loop iteration.

4.2.20 TelephoneBook

The TelephoneBook extends the JPanel class and provides a simple interface that users can interact with to find out the telephone number of any phone connected to the network. The virtual book provides a series of 27 alphabetised index tabs that, when clicked, update a JList object with a subset of the phones online. For example, if the "A" tab is pressed than all entries in the phone cache whose owner's surnames start with "A" are populated in the list. Figure 4.10 below depicts the TelephoneBook user interface.



Weaver, Matt (51609)
Williamson, Peter (0152467399)
Winetrobe, Mary (52543)

Figure 4.10 ***User interface of the telephone book***

The main role of the TelephoneBook class is to provide a user interface for the telephone book and abstract over the inner workings of how the list of phones is populated. The task of managing the list and changing it to reflect the user's choice of letter index is delegated to the TelephoneListModel class.

4.2.21 TelephoneListModel

This class extends the AbstractListModel class that provides an abstract definition of a data model that provides a list with its contents. TelephoneListModel provides a getList(String letter) method where *letter* is used to refine the number of phones displayed in the list. There are three types of strings passed into this method: the letters A to Z and two special keywords, "all" and "clear". The letters A to Z are used to display all the phones whose owner's surname starts with a particular letter. The "all" keyword is used to return all the phones online and "clear" is used to empty the list. It is worth nothing that the getList() method is case insensitive.

Encapsulated inside this class is a Vector that is used to contain a specific subset of all the phones online. For example, if the string passed into the getList() method was "C" then this vector would contain a deep copy of all the Phone objects whose owner name begins with the letter "C" or "c". It was mentioned in section 4.1.5 how the Phone object provides a toString() method that is significant to the core functionality of the telephone book. The entries listed in the JList are obtained directly from the toString() method as a Vector is used to populate the list. In order to change the way that entries are displayed in the list it is necessary to modify the toString() method in the Phone class.

4.2.22 TelephoneListSelectionModel

This is a simple class that is required to control how a user selects entries from the list. By default, a user can select any combination of items in a JList. This is not desirable, as it is impossible to call more than one user at a time in the basic phone environment. The selection model was therefore needed to prevent users from selecting more than one entry in the list. TelephoneListSelectionModel extends DefaultListSelectionModel and is instantiated in the TelephoneBook constructor.

4.3 Summary

This chapter has shown how the important design decisions made in chapter 3 were implemented as Java classes in an object-oriented manner. A total of 23 classes were created to fully encapsulate all the necessary state and behaviour of the basic phone system. Various UML class diagrams were presented and each class has been explained in a fair amount of detail, with fundamental classes being explained from different perspectives. Important functionality within these classes has been described where appropriate, keeping the discussion at a relatively high level. To gain a more detailed understanding of the Java concepts used it is recommended that the reader refer to the accompanying website (http://www.lancs.ac.uk/ug/weavermj/project/) where a full listing of the source code is provided.

Chapter 5 continues the implementation by discussing the techniques used to add additional incremental telephony functionality to the basic telephone application.

Chapter 5. Implementation continued : Addition Of Incremental Telephony Functionality

This second implementation chapter focuses on demonstrating how additional telephony functionality was added to the basic phone application using an incremental approach. The ideals of aspect-oriented programming that were so heavily discussed in the background section are finally put to the test by adding a telephony feature to the underlying application using two methods: using traditional object-oriented methodology by modifying existing Java classes and using more recent aspect-oriented methodology by creating a new aspect in AspectJ. The techniques are compared and the favoured approach is then used to generate a further two telephony features that can be incorporated into the phone application.

5.1 AspectJ semantics

The background chapter introduced some new terminology in section 2.3.1 specific to the domain of aspect-oriented programming. This short section expands on those definitions by providing specific AspectJ examples of *join points*, *pointcuts* and *advice*.

5.1.1 Join points

A join point is a well-defined point in the execution of program, such as a call to a method or the setting of a class attribute.

The generic signature of a join point that identifies a call to a particular method is defined as:

<access_type> <return_value> <type>.<method_name>(parameter_list)

So, an example of a join point that defines a call to a method setOwner() in the Phone class would look like this:

public void Phone.setOwner(String)

The signature of a join point that identifies calls to the constructor method of a class is defined as:

<method_access_type> <class_type.new>(<parameter_list>) [throws <exception>]

An example of a join point that defines a call to the constructor of a Phone class would resemble the following:

public Phone.new(String, String, InetAddress, int)

The signature of a join point that sets the value of a particular field is defined as:

<field_type> <class_type>.<field_name>

So, an example of a join point that defines the setting of the telephone number attribute in the Phone class would look like this:

private int Phone.telephoneNumber

5.1.2 Pointcuts

A pointcut is a collection of one or more join points grouped together to determine a precise moment where a given concern will cross-cut an application. The generic signature of a pointcut is fairly complex but a typical pointcut has the following properties:

- an access type, either private or public,
- a unique text string representing the name of the pointcut, analogous to a method name in a Java class,
- zero or more parameters, used to transfer the context pulled from a join point into the aspect, and
- one or more pointcut designators, used to define join points that make up the pointcut.

Pointcuts that define the join points explained above might look like this:

public pointcut1() : call(public void Phone.setOwner(String)) ;

public pointcut2() : call(Phone.new(String, String, InetAddress, int)) ;

public pointcut3() : set(private int Phone.telephoneNumber) ;

where *call* and *set* are the pointcut designators. Other types of common designators, as described by Gradecki (2003), are listed below:

target	Returns the target object of a join point or limits the scope of a join point
args	Exposes the arguments to a join point or limits the scope of the pointcut
within	Matches join points within a specific class type

Multiple designators can be used with a single pointcut to define a very specific point of execution within a program. The following pointcut shows how the *within* designator can be used to define all the points where a new Phone object is instantiated within a particular class called OutgoingCall, and how the *args* designator can be used to pull the arguments to the constructor into the pointcut.

```
public pointcut allPhones(String telephoneNumber, String owner) :

                                        call(Phone.new(String, String)) &&
                                        within(OutgoingCall) &&
                                        args(telephoneNumber, owner) ;
```

5.1.3 Advice

Advice describes a section of code that executes when a join point in an application is reached. Advice is most useful when formal parameters are used to obtain the particular context of a join point. There are four types of context that can be pulled into an advice block using a pointcut. These are:

- arguments, selected with the *args* designator, which are typically the parameters passed to methods and constructors,
- the currently executing object, selected with the *this* designator,
- the target of a method or constructor call, selected with the *target* designator, and
- the return value from a join point, selected with the *returning* designator.

The allPhones pointcut on the previous page could be used to perform additional operations *before*, *after* and *around* the calling of the constructor method. Here are examples of how the three different types of advice might look.

```
before(String telephoneNumber, String Owner) :
                                        allPhones(telephoneNumber, owner)
    {
    // code executed before the Phone() constructor
    }

after(String telephoneNumber, String Owner) :
                                        allPhones(telephoneNumber, owner)
    {
    // code executed after the Phone() constructor
    }

void around(String telephoneNumber, String Owner) :
                                        allPhones(telephoneNumber, owner)
    {
    // code executed before the Phone() constructor
    proceed(telephoneNumber, owner)
    // code executed after the Phone() constructor
    }
```

Around advice is interesting as it can be used instead of *before* and *after* advice to produce the same results. The *proceed* statement above calls the original constructor and executes it as normal. If this were omitted then the original constructor method would not be processed at all and only the code within the around block would execute.

This chapter continues to show how the first telephony feature was added as an aspect using AspectJ.

5.2 Adding the first feature – call forwarding on busy

Figure 4.5 in the previous implementation chapter presented the user interface of the basic phone application. It can be seen how all of the user interface components were added to a single pane using a JTabbedPane object. This was an intentional design decision to allow for additional user interface components, such as the dynamic telephone book, to be added to the interface at great ease without disturbing the positioning of existing components. Before the first feature could be added, a separate aspect was created to add an extra pane to the JTabbedPane of the user interface. This pane will contain sub panes, one for each feature that is implemented, and will allow features to be incrementally added and removed as necessary.

5.2.1 FeaturesAvailable aspect

This is a very simple aspect containing two pointcuts and two pieces of advice. The first pointcut uses a *target* designator with one formal parameter to obtain the instance of the PhoneUI class. A *before* advice block extracts the instance of PhoneUI from the context of the join point and stores a reference to it within the FeaturesAvailable aspect. A second pointcut picks up on a call to the setupFeaturesPane() method within the PhoneUI class and uses *after* advice to set a boolean flag within PhoneUI to inform it that some features are available. The PhoneUI class tests the value of this flag when setting up the user interface components and adds an extra "Features" pane if it equates to true. Figure 5.1 below illustrates this additional empty tab added to the application.

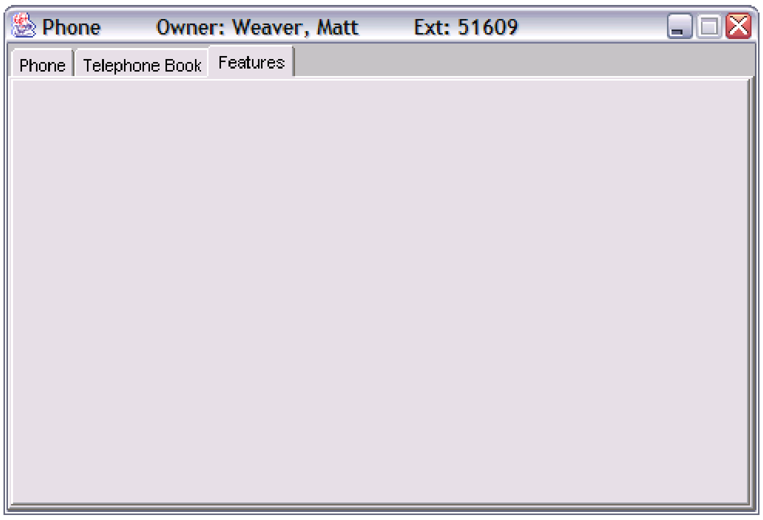


Figure 5.1 ***The phone application with the added "Features" pane***

The features pane is actually another instance of the JTabbedPane class which will allow any number of features to be added to the application as panes without interfering with the layout of the original phone application.

The first feature to be implemented will be call forwarding on busy. Should a user receive a second incoming call whilst they are already engaged in a first then this feature will enable a user to forward the second call onto another phone.

Figure 5.2 below shows, in diagrammatic form, the cross-cutting relationship between the FeaturesAvailable aspect and the PhoneUI class. At the time of writing, there is no established standard for describing aspect-oriented systems using the Unified Modelling Language (UML). Significant effort has been made by the aspect-oriented software development community to make such elements available in future releases of the UML standard. The convention used by the majority of diagrams in this section builds on a proposal by Basch (2003). The circle with the cross inside represents one or more join points that cut across the PhoneUI class.

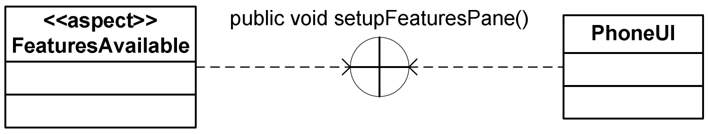


Figure 5.2 ***UML diagram showing the join points that form a relationship between the FeaturesAvailable aspect and the PhoneUI class***

5.2.2 Adding the feature using an aspect-oriented approach

The first approach to implementing the call forwarding on busy feature was to add an aspect to the system using AspectJ. This aspect encapsulated all the necessary functionality, state and behaviour associated with forwarding calls to another user.

Four of the primary classes used to implement the underlying application had to be cross-cut in order to provide the required functionality. Figure 5.3 below outlines the nature of the cross-cutting concerns. The specific detail of how these were modularised into a single aspect is explained below, along with justification for the need to cut across each of the classes.

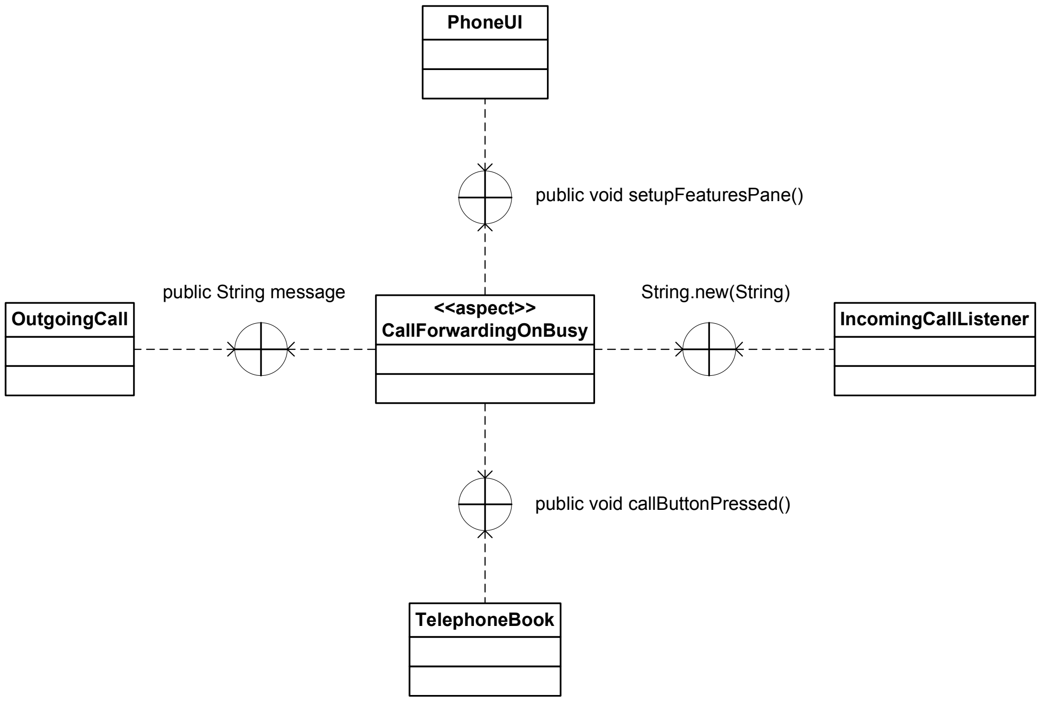


Figure 5.3 ***The cross-cutting nature of the CallForwardingOnBusy aspect***

PhoneUI

The new feature needed to have its own pane within the "Features" pane of the phone application. To successfully add a call forwarding on busy pane, the instance of PhoneUI had to be extracted using a pointcut with a *target* designator. A second pointcut was used to identify the point where the setupFeaturesPane() method is called in the PhoneUI class so that a block of *after* advice could be used to add a new JPanel to the features tab. Figure 5.4 below shows the "Call Forwarding On Busy" pane added to the phone application within the features tab.

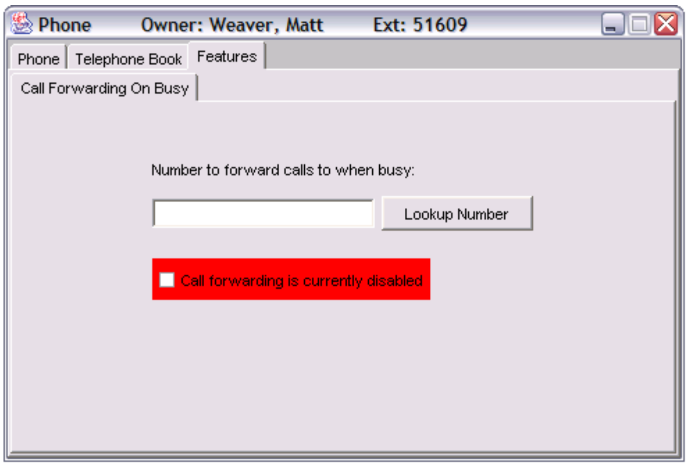

Phone Owner: Weaver, Matt Ext: 51609
Phone
Telephone Book
Features
Call Forwarding On Busy
Number to forward calls to when busy:
Lookup Number
Call forwarding is currently disabled

Figure 5.4 ***The phone application with a pane added for the call forwarding on busy feature***

TelephoneBook

The TelephoneBook class was reused to provide a simple interface for choosing the phone to forward calls to when busy. A separate JFrame window is launched when a user clicks on the "Lookup Number" button. Figure 5.5 below depicts the pop-up phone chooser window.

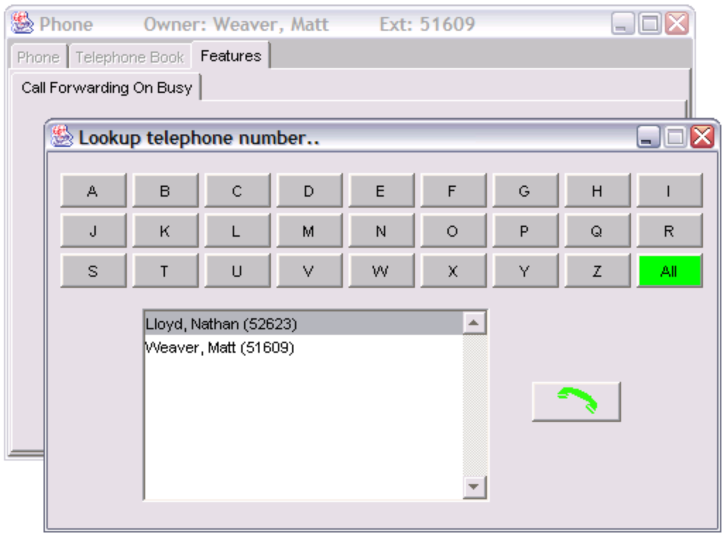


Figure 5.5 ***The pop-up phone chooser window displaying a modified instance of the TelephoneBook class***

The ActionListener detects the button click and contains code to create a new JFrame window containing a new instance of the TelephoneBook class. The TelephoneBook class is modified so that when the green call button is pressed, the phone number of the selected user is entered into the "Number to forward calls to when busy" JTextField instead of being used to make an outgoing call. A single pointcut pinpoints the exact point at which the callButtonPressed() method is called from within the TelephoneBook class using a *target* designator. A void *around* advice block is used to change the behaviour of the call button for this particular instance of the TelephoneBook.

IncomingCallListener

The IncomingCallListener class is modified so that a "forward call" message is sent to phones that attempt to call a phone that is engaged in another call and has the call forwarding on busy feature enabled. A single pointcut is used to determine the point in the IncomingCallListener class where a new String object is created with a "phone engaged" message. A simple piece of *around* advice is used to test if call forwarding is enabled and change the value of the message to "forward call:", followed by the phone number of the phone that will handle the call on behalf of the busy user. Figure 5.6 below shows the exact *around* advice used; it is interesting to see how such a small snippet of code can cause an application to function in a completely different way.

```
String around(String string) : sendingEngagedMessage(string)
    {
    if(string.equals("phone engaged") && featureStatusCheckBox.isSelected())
        {
        string = "forward call:" + forwardingNumber ;
        }
    return string ;
    }
```

Figure 5.6 ***The around advice block that changes the "engaged" message to a "forward call" message***

OutgoingCall

The OutgoingCall class needs to be modified so that it can act accordingly when it receives a "forward call" message from the phone that it is trying to contact. A single pointcut is specified that defines a join point wherever a String variable called message is assigned a value within the OutgoingCall class. A piece of advice executes *after* this pointcut and updates the numberToDial variable within the PhoneUI instance so that it represents the number of the phone that the call should be redirected to. Finally, a new OutgoingCall instance is started as a thread.

CallForwardingOnBusy

The structure of the CallForwardingOnBusy aspect can be summarised by figure 5.7 on the next page which shows the attributes, pointcuts, advice and methods that are defined within the aspect.

```
<<aspect>>
CallForwardingOnBusy
-----------------------------------------------------------------------------------------
<<attributes>>
phone : PhoneUI
forwardingNumberTextField : JTextField
lookupButton : JButton
featureStatusCheckBox : JCheckBox
lookupFrame : JFrame
lookupOpen : boolean
telephoneBook : TelephoneBook
forwardingNumber : String
message : String
-----------------------------------------------------------------------------------------
<<pointcuts>>
getPhone(PhoneUI phone) : target(phone)
featuresPane() : call(public void setupFeaturesPane())
lookupCallButtonPressed(TelephoneBook telephoneBook) :  call(public void callButtonPressed())
                                                            && target(telephoneBook)
sendingEngagedMessage(String string) : call(String.new(String)) && args(string)
getMessage(String message) : set(public String message) && within(OutgoingCall) && args(message)
-----------------------------------------------------------------------------------------
<<advice>>
before(PhoneUI phone) : getPhone(phone)
after () : featuresPane()
void around (TelephoneBook telephoneBook) : lookupCallButtonPressed(telephoneBook)
String around(String string) : sendingEngagedMessage(string)
after(String message) : getMessage(message)
-----------------------------------------------------------------------------------------
<<methods>>
forwardCall()
addFeatureToPane()
featureStatusCheckBoxPressed()
lookupButtonPressed()
callButtonPressedOnLookup()
actionPerformed(event : ActionEvent)
windowClosing(event : WindowEvent)
```

Figure 5.7 *Structure of the CallForwardingOnBusy aspect*

5.2.3 Adding the feature using an object-oriented approach

Once the implementation of the call forwarding on busy feature had been successfully completed in an aspect-oriented manner, the same feature was added to the basic phone application using an object-oriented technique. This involved modifying existing Java classes to add the necessary functionality required to forward calls to a third party when a phone is busy. A copy of the source code was made so that the original code for the basic application could be weaved in with the call forwarding on busy aspect and be directly compared to the equivalent object-oriented version.

The functionality encapsulated within the CallForwardingOnBusy aspect was realised as an extension to each of four key Java classes that make up the phone application: PhoneUI, TelephoneBook, IncomingCallListener and OutgoingCall. The PhoneUI and TelephoneBook classes were modified to provide a pane to host the user interface components associated with the call forwarding on busy feature. The IncomingCallListener and OutgoingCall classes were adapted to adequately manage the process of forwarding calls to a third party when a phone is in an engaged state. An additional LookupFrame class was created to host a new instance of the TelephoneBook that could be launched in a separate window. Figure 5.8 on the next page shows the additional class attributes and methods that were added to each class (in plain text) and existing methods that were modified (in **bold**).

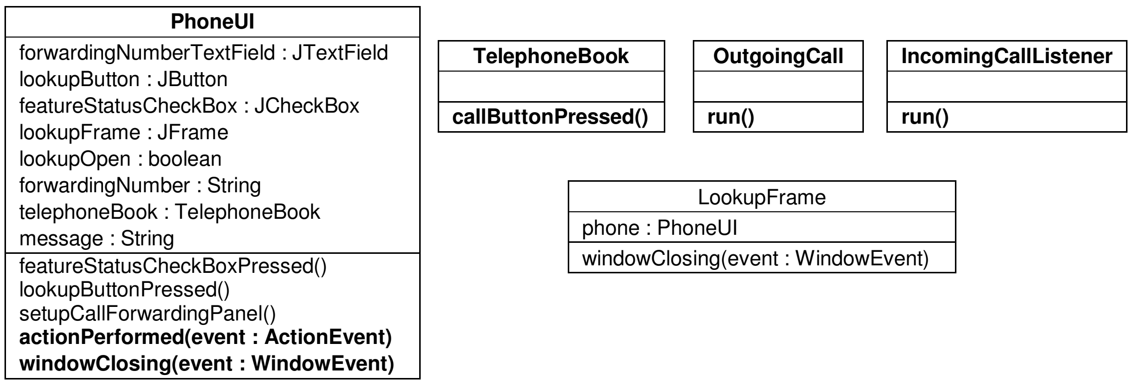

PhoneUI
forwardingNumberTextField : JTextField
lookupButton : JButton
featureStatusCheckBox : JCheckBox
lookupFrame : JFrame
lookupOpen : boolean
forwardingNumber : String
telephoneBook : TelephoneBook
message : String
featureStatusCheckBoxPressed()
lookupButtonPressed()
setupCallForwardingPanel()
actionPerformed(event : ActionEvent)
windowClosing(event : WindowEvent)
TelephoneBook
callButtonPressed()
OutgoingCall
run()
IncomingCallListener
run()
LookupFrame
phone : PhoneUI
windowClosing(event : WindowEvent)

Figure 5.8 *Changes made in the object-oriented approach to adding the first feature*

5.3 Comparison of AOP and OOP techniques

Now that the call forwarding on busy feature has been implemented using both an aspect-oriented and an object-oriented approach, both techniques can be critically reviewed to determine which methodology is more suitable for adding a telephony feature to a working telephone application. This relatively short review will reflect on the writer's own experience at programming with the two techniques by highlighting the positive and negative characteristics of each approach.

The review is needed so that a decision can be made as to which method will be used for adding two more features to the application. A more detailed and extensive evaluation will be performed in chapter 7.

5.3.1 Aspect-oriented approach

This approach encapsulated all of the state and behaviour associated with the call forwarding on busy feature into a single aspect construct.

Initially, the process of learning the new AspectJ language semantics was very tedious. Major difficulty was involved in actually drilling-down a pointcut to an exact point in one of the existing classes, as more often than not the join points were cross-cutting multiple classes. To solve this problem, the *within* designator was used to narrow the scope of the pointcut.

The addition of the "Features" tab to the phone application was not entirely solved by the introduction of a new aspect. A setupFeaturesPane() method had to be added to the PhoneUI class to create a new JTabbedPane instance and add it as a third tab to the existing JTabbedPane of the application. A boolean variable was also introduced into the PhoneUI class to represent whether or not features were available. These additions allowed a pointcut to be defined that picked up on a call to the setupFeaturesPane() method and executed some *after* advice to set the boolean flag to true to indicate that one or more features were available, thus adding a new pane to the phone application.

Once the initial obstacle of applying the new AspectJ semantics was overcome, the remaining task of adding the call forwarding feature was fairly straightforward. Placing all of the code associated with the feature within a single file provided a means for the writer to focus on the task in hand and not get distracted by other code that was irrelevant to call forwarding. This was incredibly beneficial, as a significant amount of time was spent when implementing the basic phone application on continually glancing back over existing code to understand how it worked.

5.3.2 Object-oriented approach

This approach involved editing existing Java classes to incorporate the necessary code that would provide call forwarding on busy functionality as a feature.

The process of adding the feature in an object-oriented fashion actually took less time than its aspect-oriented equivalent. However, this is not a true reflection, as the object-oriented approach to adding the feature was implemented directly after it had already been added as an aspect. Performing the aspect-oriented implementation first meant that the writer already knew where the new call forwarding code would need to cross-cut the existing application, greatly simplifying the task of adding the feature in an object-oriented manner. As a result, the relevant code was merely added in the correct places without paying any regard to the principles of object-oriented design.

Embedding the code for a totally different feature within the application might have shown more interesting results. What can be determined, though, is that the object-oriented implementation of the feature resulted in both scattered and tangled code being introduced across the various classes. Should the call forwarding on busy code need to change then it would prove highly tedious to identify every code fragment that contributes to the functionality of the feature.

5.3.3 The favoured approach

Despite the ease and speed of the object-oriented approach to adding a feature, aspect-oriented development was chosen as the technique to be used for implementing a further two telephony features for the application. Once the writer was familiar with the AspectJ language semantics and AOP design principles, encapsulating all the code required to implement a particular feature within a single aspect was much more efficient than editing a series of existing Java classes. Also, using aspects would offer a way of providing the application with a varying level of functionality, as any number of aspects can be weaved in with existing Java bytecode. This would allow for features to be added to and removed from the underlying phone application at any time, with only a simple recompile operation needed.

The following section describes how two additional telephony features, anonymous calling and anonymous call barring, were added to the basic phone application using aspects.

5.4 Anonymous calling

Anonymous calling allows a user to make an outgoing call to another phone without revealing their own identity. An AnonymousCalling aspect was created to encapsulate the functionality required to implement this feature. Figure 5.9 below shows how the aspect cuts across other classes in the underlying system.

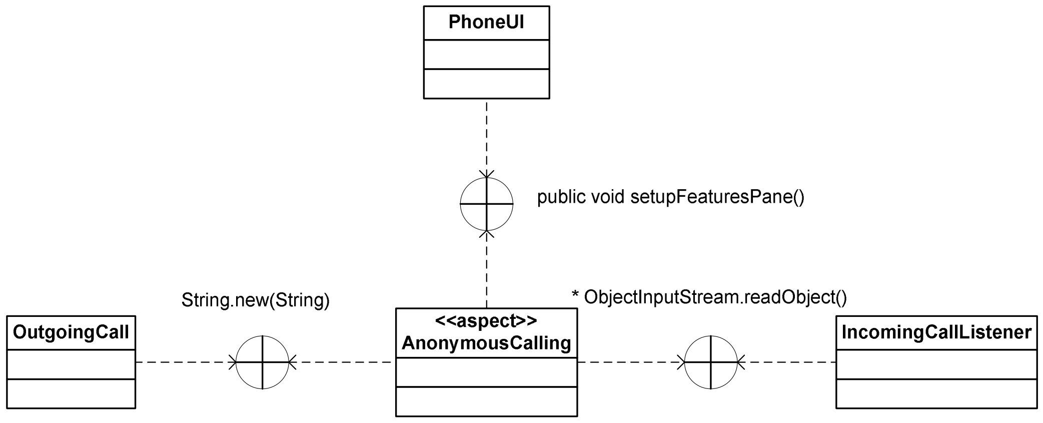


Figure 5.9 ***Cross-cutting nature of the AnonymousCalling aspect***

In figure 5.13 above the join point between the AnonymousCalling aspect and the IncomingCallListener class is a call to the readObject() method within the java.io.ObjectInputStream class. The * denoted at the beginning of the join point represents a wildcard so that any access type (public, protected or private) is included in the join point.

The aspect first uses some primitive before and after advice to obtain a reference to the PhoneUI instance and add a new "Anonymous Calling" pane to the features pane, as shown below in figure 5.10.

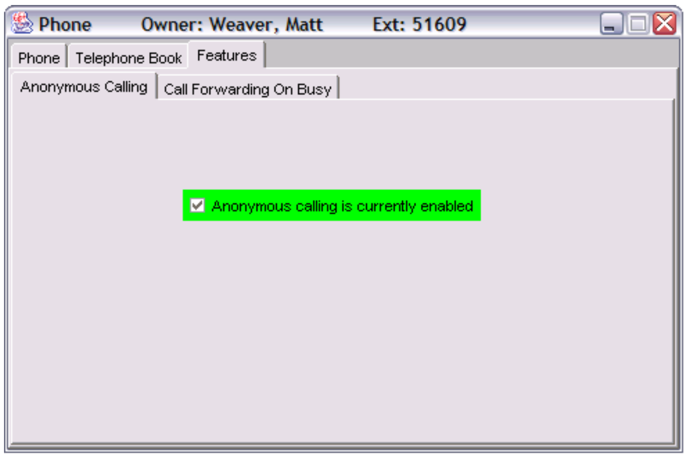

Phone Owner: Weaver, Matt Ext: 51609
Phone
Telephone Book
Features
Anonymous Calling
Call Forwarding On Busy
☑ Anonymous calling is currently enabled

Figure 5.10 ***The phone application with a new***

pane for the anonymous calling feature

Next, a piece of *around* advice is used to modify a message sent by the OutgoingCall class. The message is changed from “begin call” to “begin call:anonymous”, as the IncomingCallListener will need to evaluate the contents of this message to determine if the incoming call is anonymous and bar it if necessary. The aspect adds a new boolean variable to the IncomingCallListener class to store the state of the incoming call, either anonymous or normal.

A second piece of *around* advice is used to test the value of the message and set the boolean to true if the call is anonymous. The “:anonymous” part of the message is removed to allow the rest of the IncomingCallListener class to operate as before. Adding the extra boolean and performing checks on it are not really needed for anonymous calling. However, the anonymous call barring feature that will be added later will need to be aware of anonymous calls so that it can deal with them appropriately.

If the anonymous calling feature checkbox is enabled then the thisPhone object discussed in section 4.2.4 is modified so that the telephone number of the phone becomes “***” and the owner’s name reads “Unknown User”.

Figure 5.11 below summaries the pointcuts and advice within the AnonymousCalling aspect.

<<aspect>> **AnonymousCalling**
<<attributes>> phone : PhoneUI anonymousCallingCheckBox : JCheckBox IncomingCallListener.anonymousCall : boolean

<<pointcuts>>
getPhone(PhoneUI phone) : target(phone)
featuresPane() : call(public void setupFeaturesPane())
sendingBeginCallMessage(String string) : call(String.new(String)) && within(OutgoingCall)
&& args(string)
receivingBeginCallMessage() : call(* ObjectInputStream.readObject())
&& within(IncomingCallListener)

<<advice>> before(PhoneUI phone) : getPhone(phone) after () : featuresPane() String around(String string) : sendingBeginCallMessage(string) String around() : receivingBeginCallMessage()

<<methods>>
addFeatureToPane()
anonymousCallingCheckBoxPressed()
actionPerformed(event : ActionEvent)

Figure 5.11 *The AnonymousCalling aspect*

5.5 Anonymous call barring

This features allows users to block incoming calls whose identity is withheld, i.e. it should bar all calls from users that have their anonymous calling feature enabled. An AnonymousCallBarring aspect was created to contain all the functionality associated with

barring anonymous calls. Figure 5.12 on the next page shows the cross-cutting nature of this new aspect.
As with the AnonymousCalling aspect, *before* and *after* advice is used to obtain a reference to the PhoneUI instance and add a new "Anonymous Call Barring" pane to the features pane, as shown in figure 5.13 below. A short *around* advice block is used to extract the ObjectOutputStream instance and store a reference to it within the aspect so that a "rejected anonymous" message can be sent back to the phone who attempted to make the anonymous call.

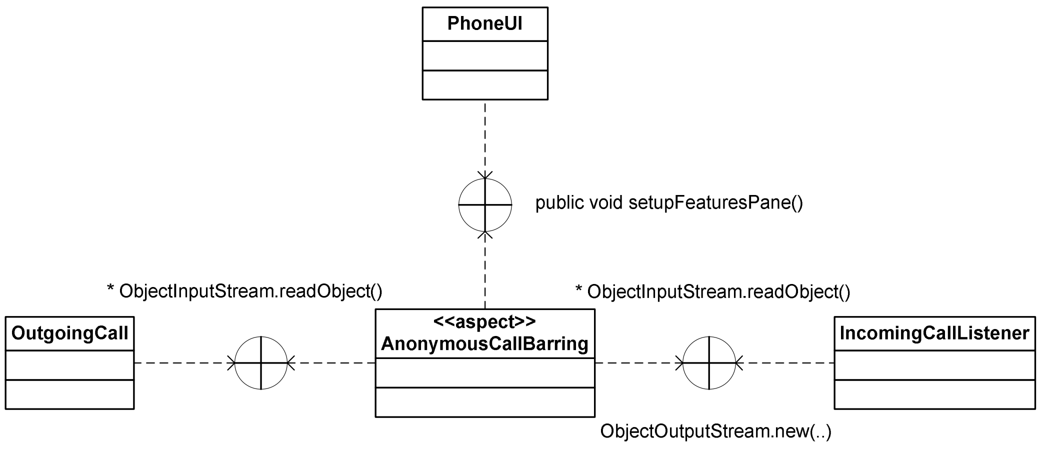

PhoneUI
public void setupFeaturesPane()
* ObjectInputStream.readObject()
* ObjectInputStream.readObject()
OutgoingCall
<<aspect>>
AnonymousCallBarring
IncomingCallListener
ObjectOutputStream.new(..)

Figure 5.12 ***Cross-cutting nature of the AnonymousCallBarring aspect***

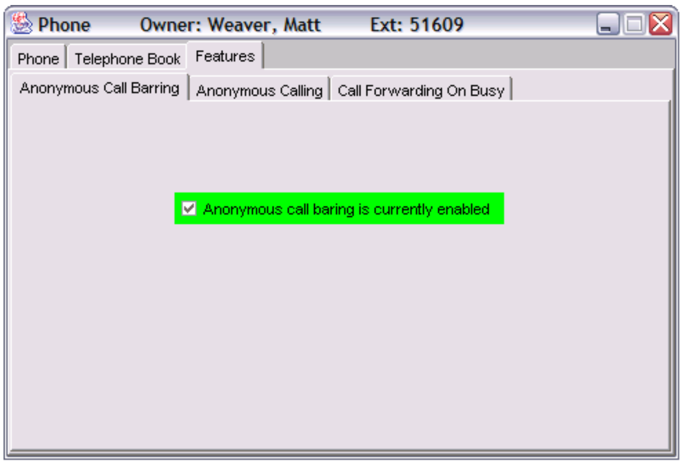
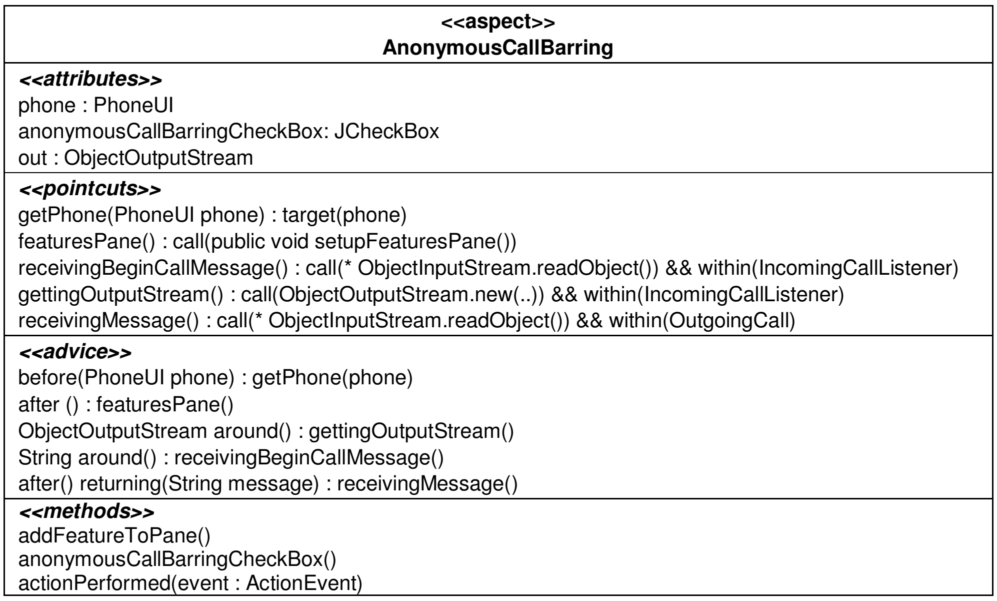


Figure 5.13 ***The phone application with a new pane for the anonymous call barring feature***

A further change is made, this time to the OutgoingCall class, so that it can detect the "rejected anonymous" message sent out by the IncomingCallListener and act on it accordingly. A piece of *after* advice is used to display an indication that the call was rejected on the calling phone's display and reset the phone to a ready state.

The specific details of the pointcuts and advice used in the AnonymousCallBarring aspect are shown in figure 5.14 on the following page.



<<aspect>> **AnonymousCallBarring**
<<attributes>> phone : PhoneUI anonymousCallBarringCheckBox: JCheckBox out : ObjectOutputStream

<<pointcuts>>
getPhone(PhoneUI phone) : target(phone)
featuresPane() : call(public void setupFeaturesPane())
receivingBeginCallMessage() : call(* ObjectInputStream.readObject()) && within(IncomingCallListener)
gettingOutputStream() : call(ObjectOutputStream.new(..)) && within(IncomingCallListener)
receivingMessage() : call(* ObjectInputStream.readObject()) && within(OutgoingCall)

<<advice>> before(PhoneUI phone) : getPhone(phone) after () : featuresPane() ObjectOutputStream around() : gettingOutputStream() String around() : receivingBeginCallMessage() after() returning(String message) : receivingMessage()

<<methods>>
addFeatureToPane()
anonymousCallBarringCheckBox()
actionPerformed(event : ActionEvent)

Figure 5.14 *The AnonymousCallBarring aspect*

5.6 Summary

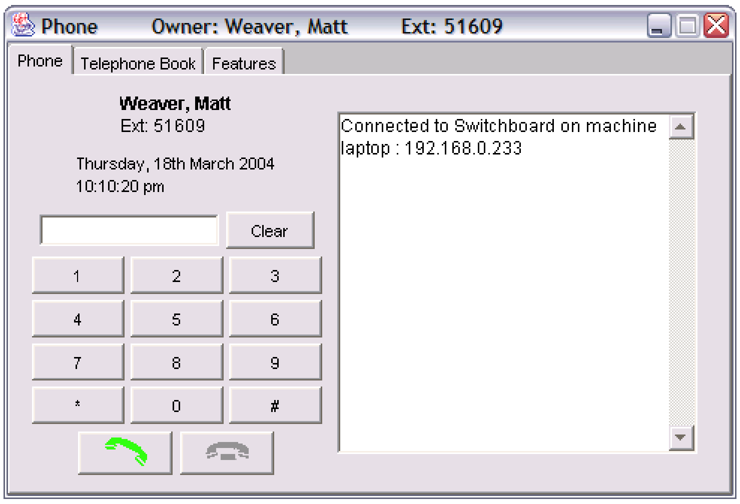
This chapter has presented the implementation details for three telephony features that were added incrementally to the basic telephone application. The semantics of AspectJ were discussed at the start of the chapter using methods and attributes from the Phone class as an example. Following this, the first telephony feature, call forwarding on busy, was implemented using both a trusted object-oriented approach by modifying existing Java classes and using a relatively new aspect-oriented approach by creating a new aspect in AspectJ. Both techniques were compared to one another and a decision was made to implement a further two features, anonymous calling and anonymous call barring, using AspectJ. The chapter finished by presenting the implementation details for these features.

Chapter 6 gives the reader a feel for what the system is like to use by presenting a selection of screen shots highlighting basic operations, such as making an outgoing call, and showing how the three features implemented in this section can be used to alter the default behaviour of the phone application.

Chapter 6. The System In Operation

This chapter presents a walkthrough of screen shots to highlight the basic operations of the phone system and provide the reader with a feel for what the application is like to use.

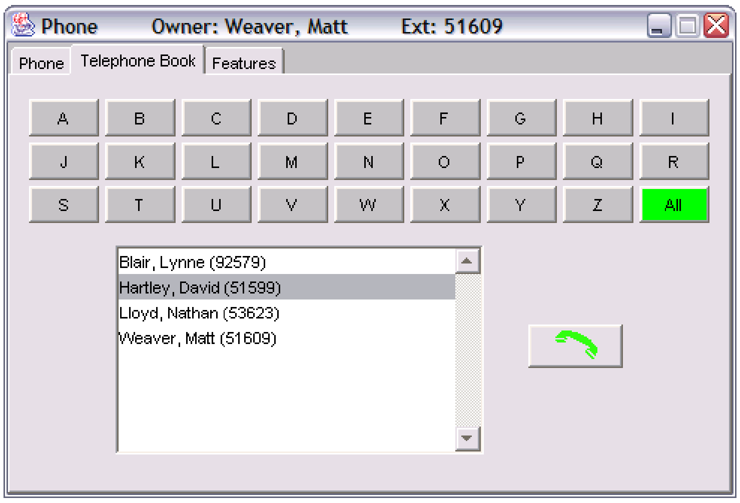
Figure 6.1 below shows the phone application in its initial "ready" state when it is first loaded.



Connected to Switchboard on machine laptop : 192.168.0.233

Figure 6.1 *The phone application in its initial ready state*

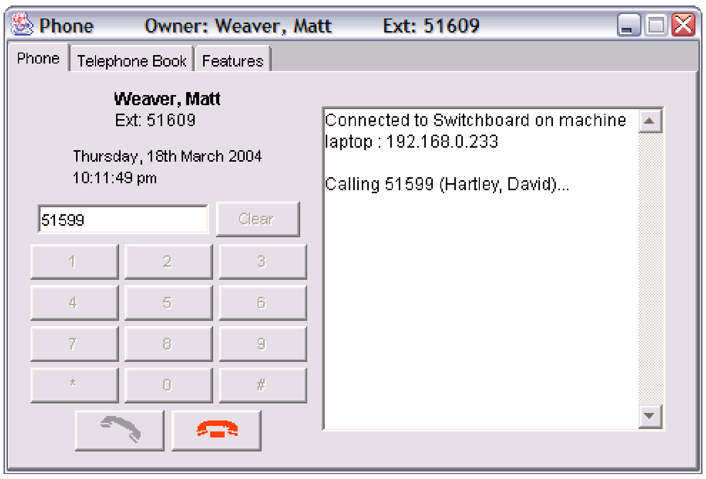
Figure 6.2 below shows the dynamic telephone book integrated into the application as a separate pane. Highlighting a name in the list and clicking the green "call" button places an outgoing call to the selected user.



Blair, Lynne (92579)
Hartley, David (51599)
Lloyd, Nathan (53623)
Weaver, Matt (51609)

Figure 6.2 *The dynamic telephone book*

Figure 6.3 below shows how the user interface of the phone is minimised whilst an outgoing call is being placed.

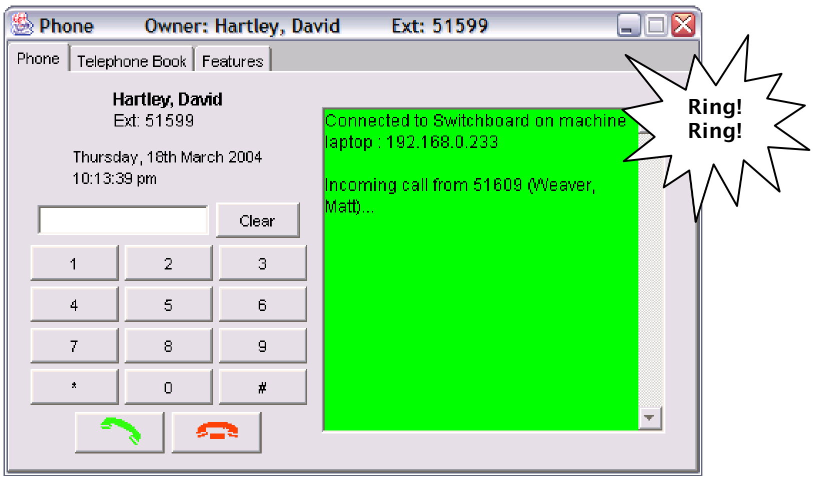


Connected to Switchboard on machine laptop : 192.168.0.233

Calling 51599 (Hartley, David)...

Figure 6.3 ***The phone making an outgoing call***

Figure 6.4 below shows the flashing green background to alert the user of a pending incoming call.

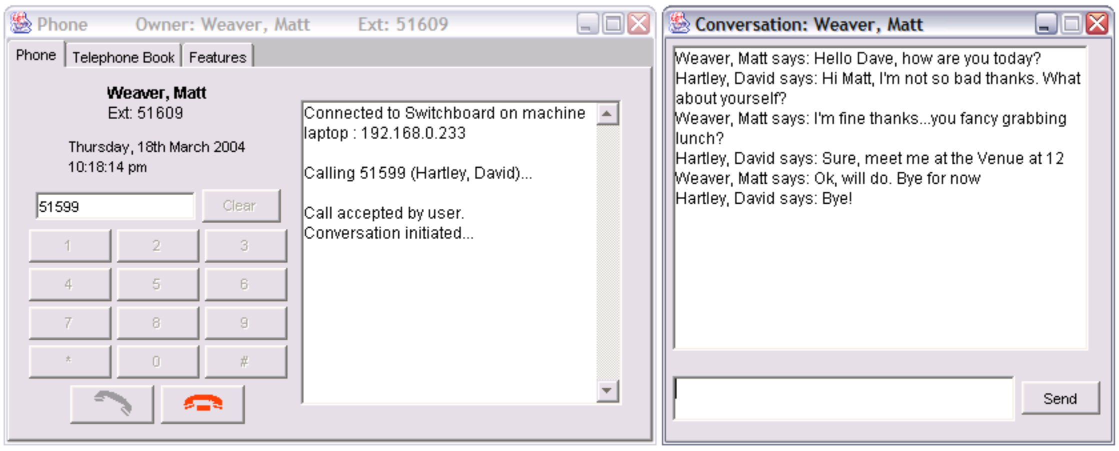


Connected to Switchboard on machine laptop : 192.168.0.233

Incoming call from 51609 (Weaver, Matt)...

Figure 6.4 ***The phone receiving an incoming call***

Figures 6.5 and 6.6 below show the chat window that appears when a call is accepted. The conversation can be seen on both the caller's phone (Matthew Weaver) and the receiver's phone (David Hartley).


Phone Owner: Weaver, Matt Ext: 51609
Phone
Telephone Book
Features
Clear
1 2 3 4 5 6 7 8 9 * 0 #
Conversation: Weaver, Matt
Send

Weaver, Matt
Ext: 51609

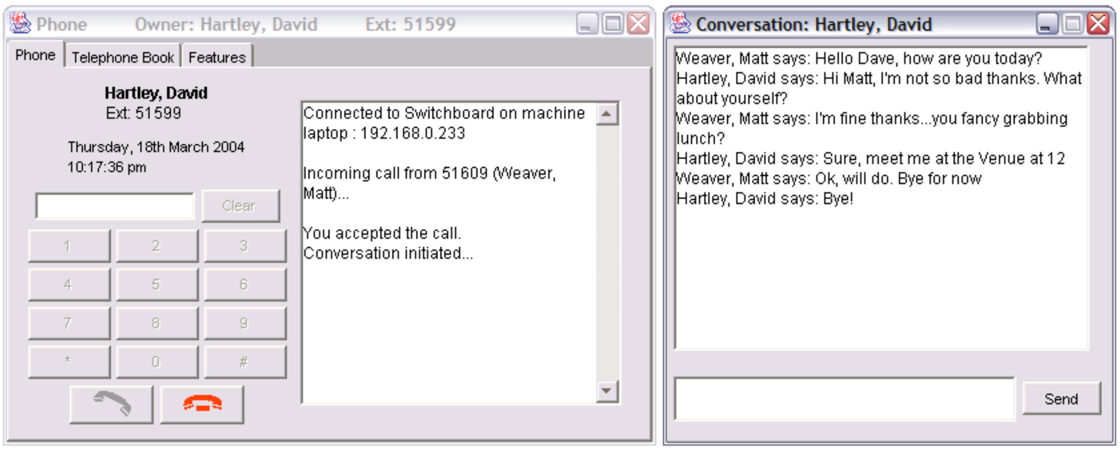
Thursday, 18th March 2004
10:18:14 pm

51599

```
Connected to Switchboard on machine
laptop : 192.168.0.233

Calling 51599 (Hartley, David)...

Call accepted by user.
Conversation initiated...
```

```
Weaver, Matt says: Hello Dave, how are you today?
Hartley, David says: Hi Matt, I'm not so bad thanks. What
about yourself?
Weaver, Matt says: I'm fine thanks...you fancy grabbing
lunch?
Hartley, David says: Sure, meet me at the Venue at 12
Weaver, Matt says: Ok, will do. Bye for now
Hartley, David says: Bye!
```

Figure 6.5 ***The calling phone in a conversation***


Phone Owner: Hartley, David Ext: 51599
Phone
Telephone Book
Features
Clear
1 2 3 4 5 6 7 8 9 * 0 #
Conversation: Hartley, David
Send

Hartley, David
Ext: 51599

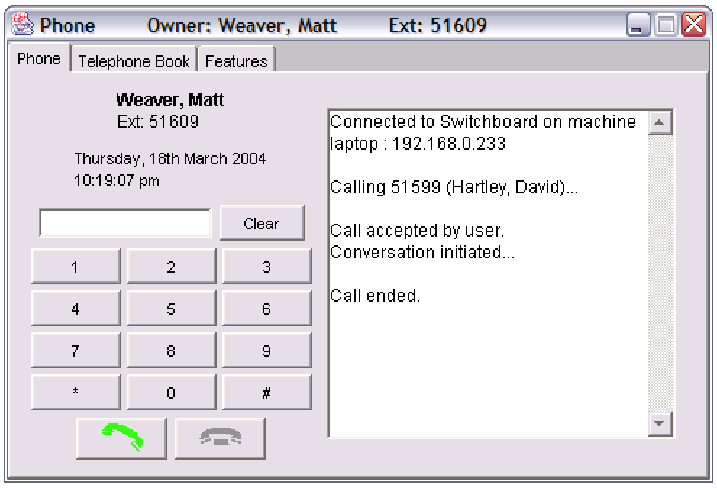
Thursday, 18th March 2004
10:17:36 pm

```
Connected to Switchboard on machine
laptop : 192.168.0.233

Incoming call from 51609 (Weaver,
Matt)...

You accepted the call.
Conversation initiated...
```

```
Weaver, Matt says: Hello Dave, how are you today?
Hartley, David says: Hi Matt, I'm not so bad thanks. What
about yourself?
Weaver, Matt says: I'm fine thanks...you fancy grabbing
lunch?
Hartley, David says: Sure, meet me at the Venue at 12
Weaver, Matt says: Ok, will do. Bye for now
Hartley, David says: Bye!
```

Figure 6.6 ***The receiving phone in a conversation***

Figure 6.7 below shows how the phone is reset to a “ready” state when it ends a conversation.



Connected to Switchboard on machine laptop : 192.168.0.233

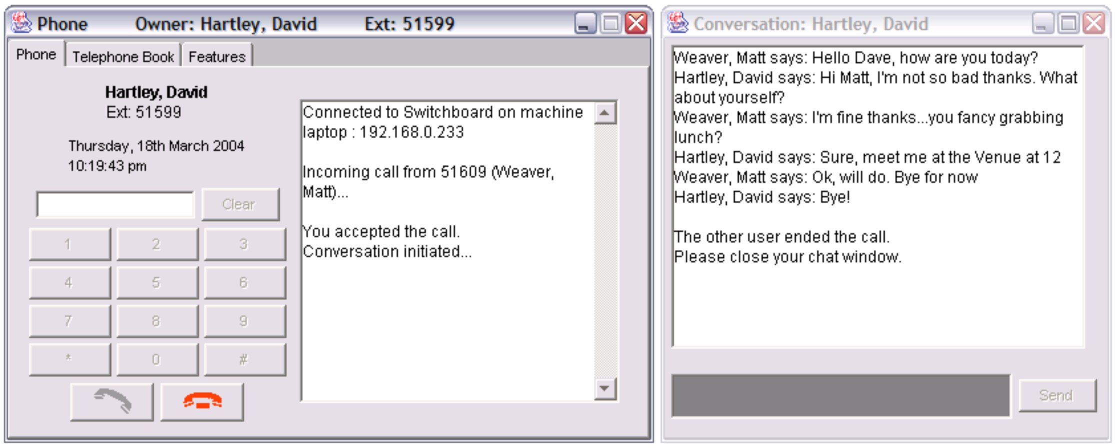
Calling 51599 (Hartley, David)...

Call accepted by user.
Conversation initiated...

Call ended.

Figure 6.7 *The phone returning to a ready state when it ends a conversation*

Figure 6.8 below illustrates how the chat window is disabled when the other user ends a call. A user must close their chat window or press the “hang-up” button to reset their phone to a “ready” state.


Phone Owner: Hartley, David Ext: 51599
Phone
Telephone Book
Features
Hartley, David
Ext: 51599
Thursday, 18th March 2004
10:19:43 pm
Clear
1
2
3
4
5
6
7
8
9
*
0
#
Conversation: Hartley, David
Send

Connected to Switchboard on machine laptop : 192.168.0.233

Incoming call from 51609 (Weaver, Matt)...

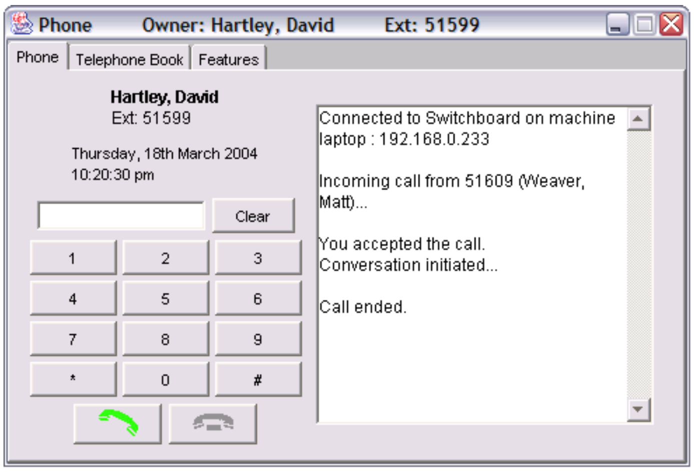
You accepted the call.
Conversation initiated...

Weaver, Matt says: Hello Dave, how are you today?
Hartley, David says: Hi Matt, I'm not so bad thanks. What about yourself?
Weaver, Matt says: I'm fine thanks...you fancy grabbing lunch?
Hartley, David says: Sure, meet me at the Venue at 12
Weaver, Matt says: Ok, will do. Bye for now
Hartley, David says: Bye!

The other user ended the call.
Please close your chat window.

Figure 6.8 *The chat window, disabled, as the other user has ended the call*

Figure 6.9 below shows the receiving phone reset to its “ready” state after closing the chat window.



Connected to Switchboard on machine laptop : 192.168.0.233

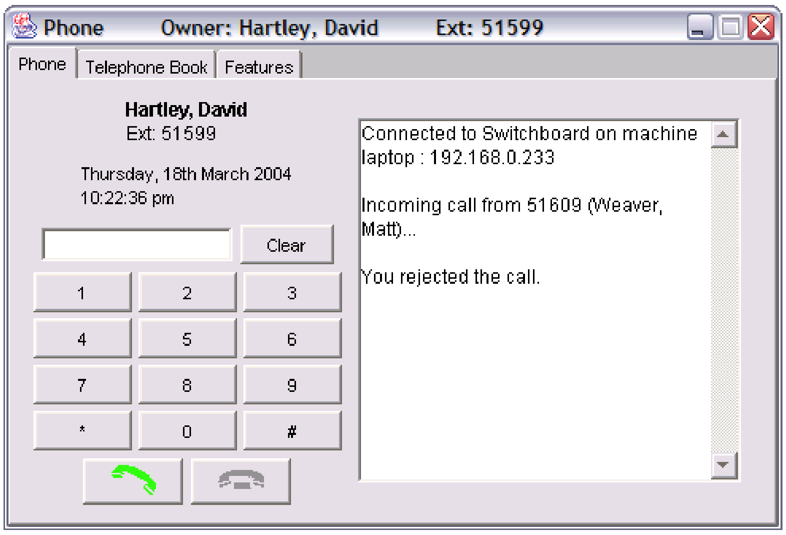
Incoming call from 51609 (Weaver, Matt)...

You accepted the call.
Conversation initiated...

Call ended.

Figure 6.9 ***The receiving phone reset to its ready state***

Figure 6.10 below illustrates how a phone is reset to the “ready” state after it rejects an incoming call.



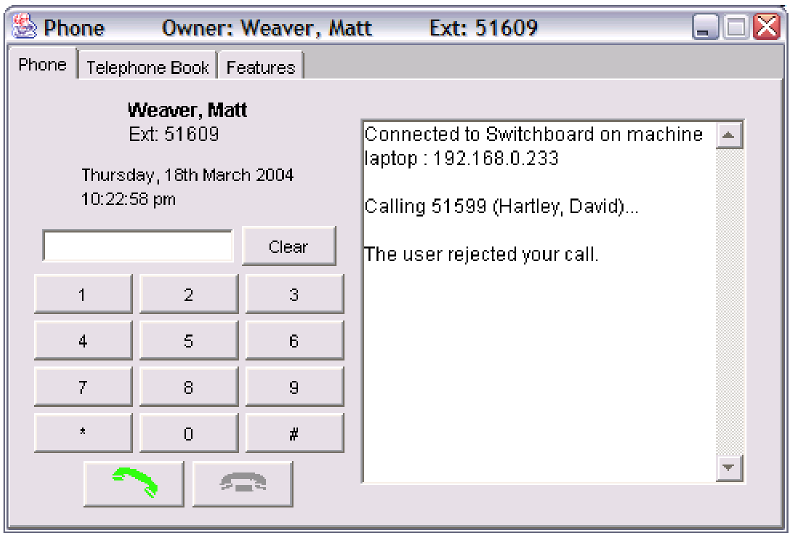
Connected to Switchboard on machine laptop : 192.168.0.233

Incoming call from 51609 (Weaver, Matt)...

You rejected the call.

Figure 6.10 ***The phone returning to a ready state after rejecting a users call***

Figure 6.11 below pictures the phone returning to a "ready" state after the remote user rejects the call.



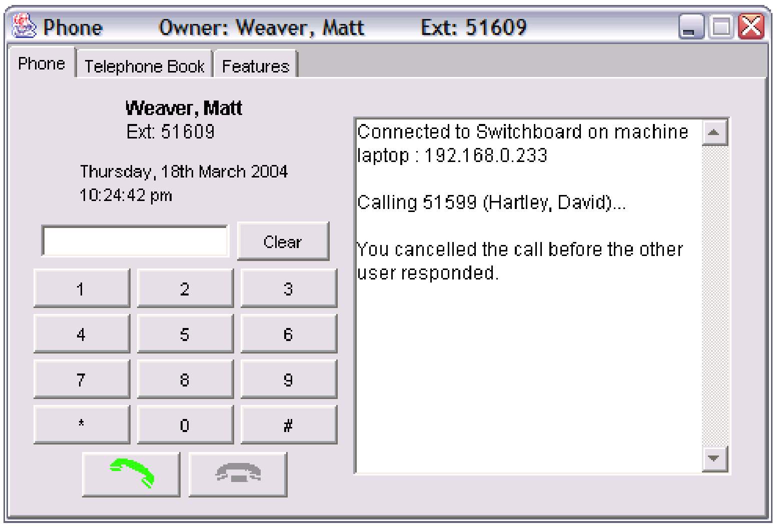
Connected to Switchboard on machine laptop : 192.168.0.233

Calling 51599 (Hartley, David)...

The user rejected your call.

Figure 6.11 ***The phone returning to a ready state after the remote phone rejects a users call***

Figure 6.12 below illustrates how a calling phone can cancel a call before the remote user has a chance to answer it.



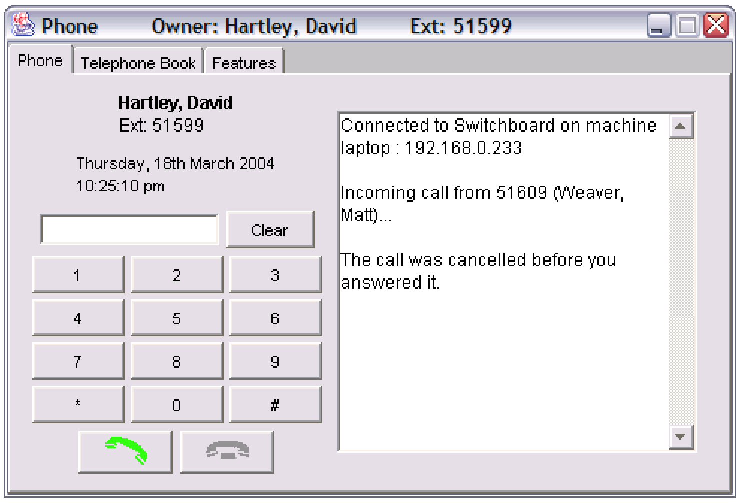
Connected to Switchboard on machine laptop : 192.168.0.233

Calling 51599 (Hartley, David)...

You cancelled the call before the other user responded.

Figure 6.12 ***The phone returning to a ready state after cancelling the outgoing call***

Figure 6.13 below shows the remote phone responding to the fact that the call was cancelled at the other end.



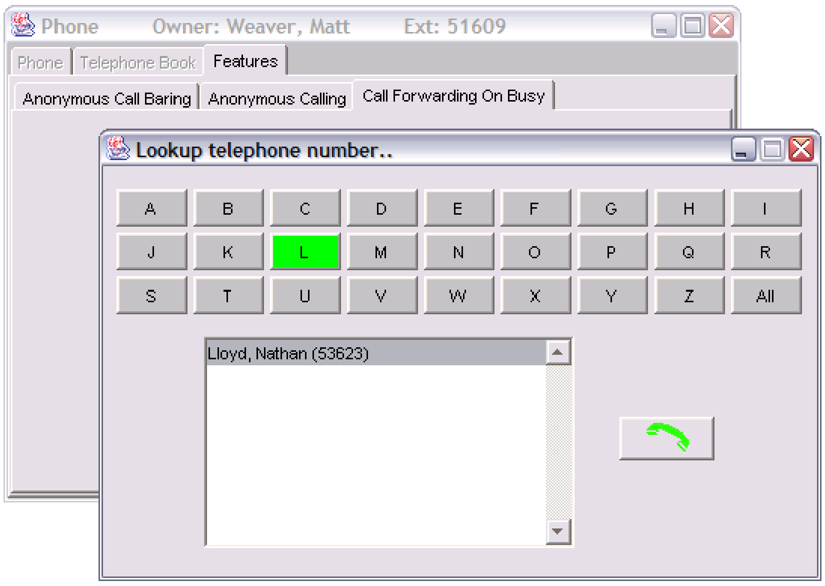
Connected to Switchboard on machine laptop : 192.168.0.233

Incoming call from 51609 (Weaver, Matt)...

The call was cancelled before you answered it.

Figure 6.13 ***The remote phone responding to the cancelled call***

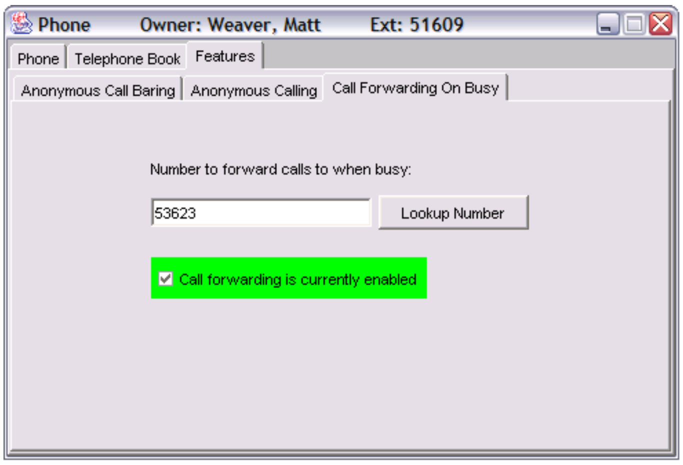
Figure 6.14 below shows the call forwarding on busy feature where a user can select a particular phone to forward calls to when it is engaged in another call.



Lloyd, Nathan (53623)

Figure 6.14 ***The lookup telephone number window within the call forwarding on busy feature***

Figure 6.15 below shows the call forwarding on busy feature enabled.



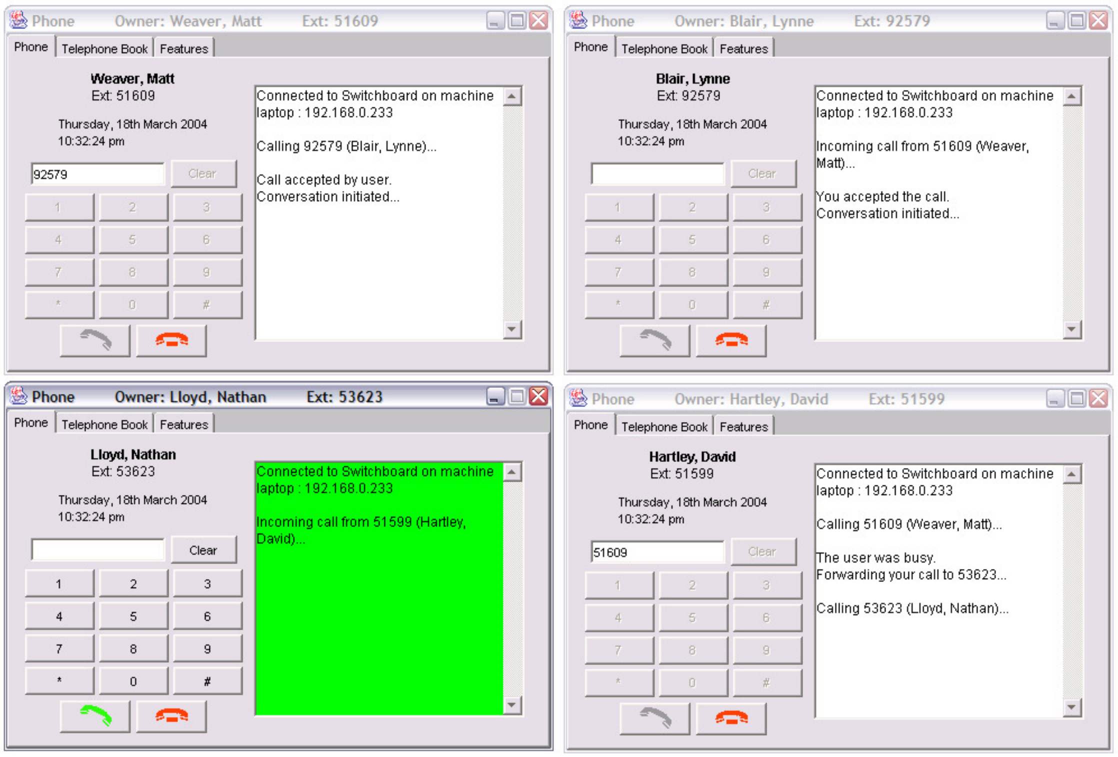
Number to forward calls to when busy:

53623

☑ Call forwarding is currently enabled

Figure 6.15 ***The call forwarding on busy feature enabled***

Figure 6.16 below shows the call forwarding on busy feature in use. The top left phone has call forwarding on busy enabled, specifying that calls be forwarded to the bottom left phone. The bottom right phone tries to call the top left phone but it is already engaged in a conversation with the top right phone and so forwards the call to the bottom left phone.


Phone Owner: Weaver, Matt Ext: 51609
Phone
Telephone Book
Features
Clear
1 2 3 4 5 6 7 8 9 * 0 #
Phone Owner: Blair, Lynne Ext: 92579
Phone Owner: Lloyd, Nathan Ext: 53623
Phone Owner: Hartley, David Ext: 51599

Weaver, Matt
Ext: 51609
Thursday, 18th March 2004
10:32:24 pm
92579

Connected to Switchboard on machine laptop : 192.168.0.233

Calling 92579 (Blair, Lynne)...

Call accepted by user.
Conversation initiated...

Blair, Lynne
Ext: 92579
Thursday, 18th March 2004
10:32:24 pm

Connected to Switchboard on machine laptop : 192.168.0.233

Incoming call from 51609 (Weaver, Matt)...

You accepted the call.
Conversation initiated...

Lloyd, Nathan
Ext: 53623
Thursday, 18th March 2004
10:32:24 pm

Connected to Switchboard on machine laptop : 192.168.0.233

Incoming call from 51599 (Hartley, David)...

Hartley, David
Ext: 51599
Thursday, 18th March 2004
10:32:24 pm
51609

Connected to Switchboard on machine laptop : 192.168.0.233

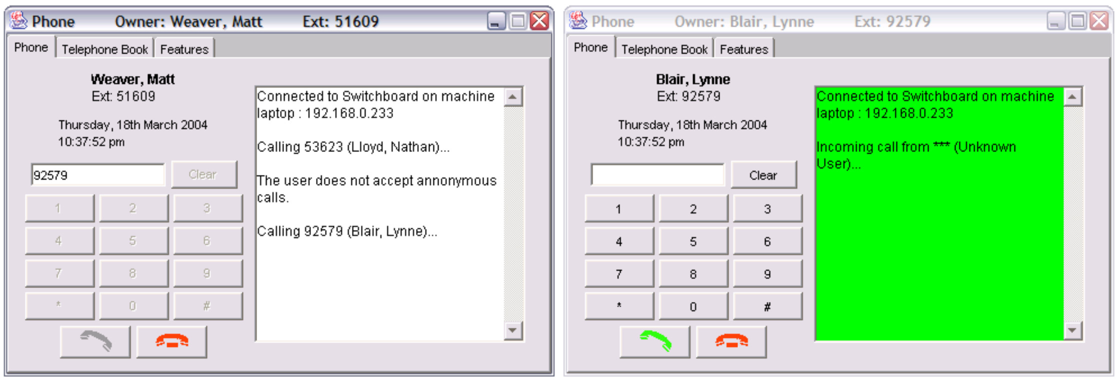
Calling 51609 (Weaver, Matt)...

The user was busy.
Forwarding your call to 53623...

Calling 53623 (Lloyd, Nathan)...

Figure 6.16 ***The call forwarding on busy feature in action***

Figure 6.17 below shows the anonymous calling feature in action, masking the phone number and owner name of the calling phone.


Phone Owner: Weaver, Matt Ext: 51609
Phone
Telephone Book
Features
Clear
Phone Owner: Blair, Lynne Ext: 92579
Phone
Telephone Book
Features
Clear

Weaver, Matt
Ext: 51609

Thursday, 18th March 2004
10:37:52 pm

92579

```
Connected to Switchboard on machine
laptop : 192.168.0.233

Calling 53623 (Lloyd, Nathan)...

The user does not accept annonymous
calls.

Calling 92579 (Blair, Lynne)...
```

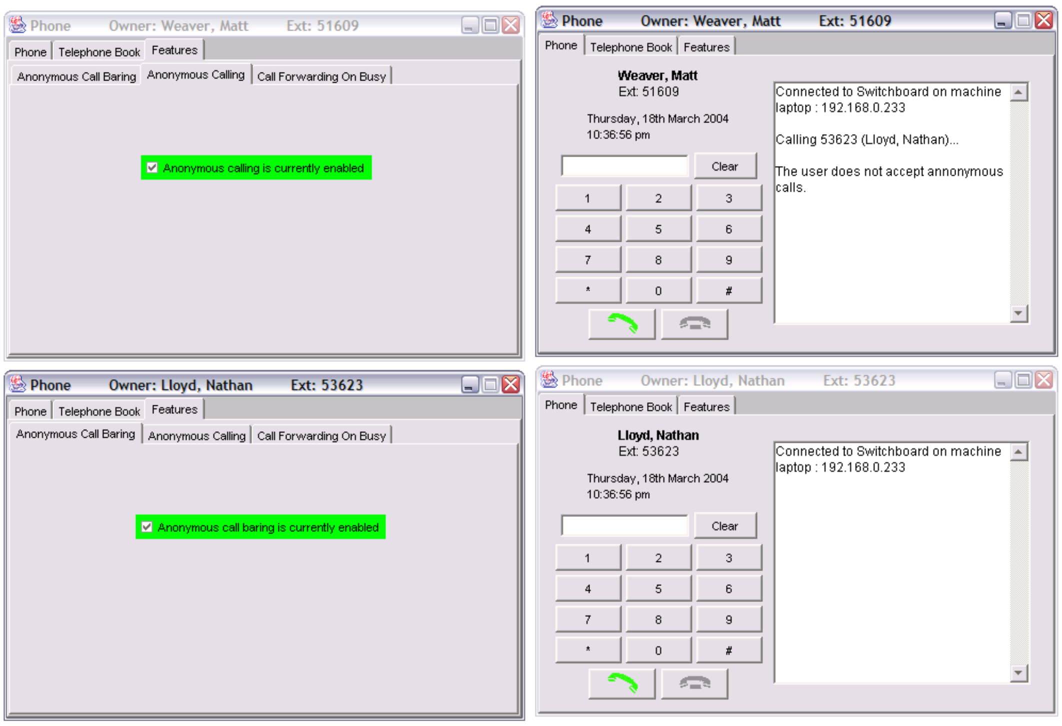
Blair, Lynne
Ext: 92579

Thursday, 18th March 2004
10:37:52 pm

```
Connected to Switchboard on machine
laptop : 192.168.0.233

Incoming call from *** (Unknown
User)...
```

Figure 6.17 ***The anonymous calling feature in operation***

Figure 6.18 below shows the anonymous calling and anonymous call barring features working together to bar anonymous incoming calls.



☑ Anonymous calling is currently enabled

Weaver, Matt
Ext: 51609

Thursday, 18th March 2004
10:36:56 pm

```
Connected to Switchboard on machine
laptop : 192.168.0.233

Calling 53623 (Lloyd, Nathan)...

The user does not accept annonymous
calls.
```

☑ Anonymous call baring is currently enabled

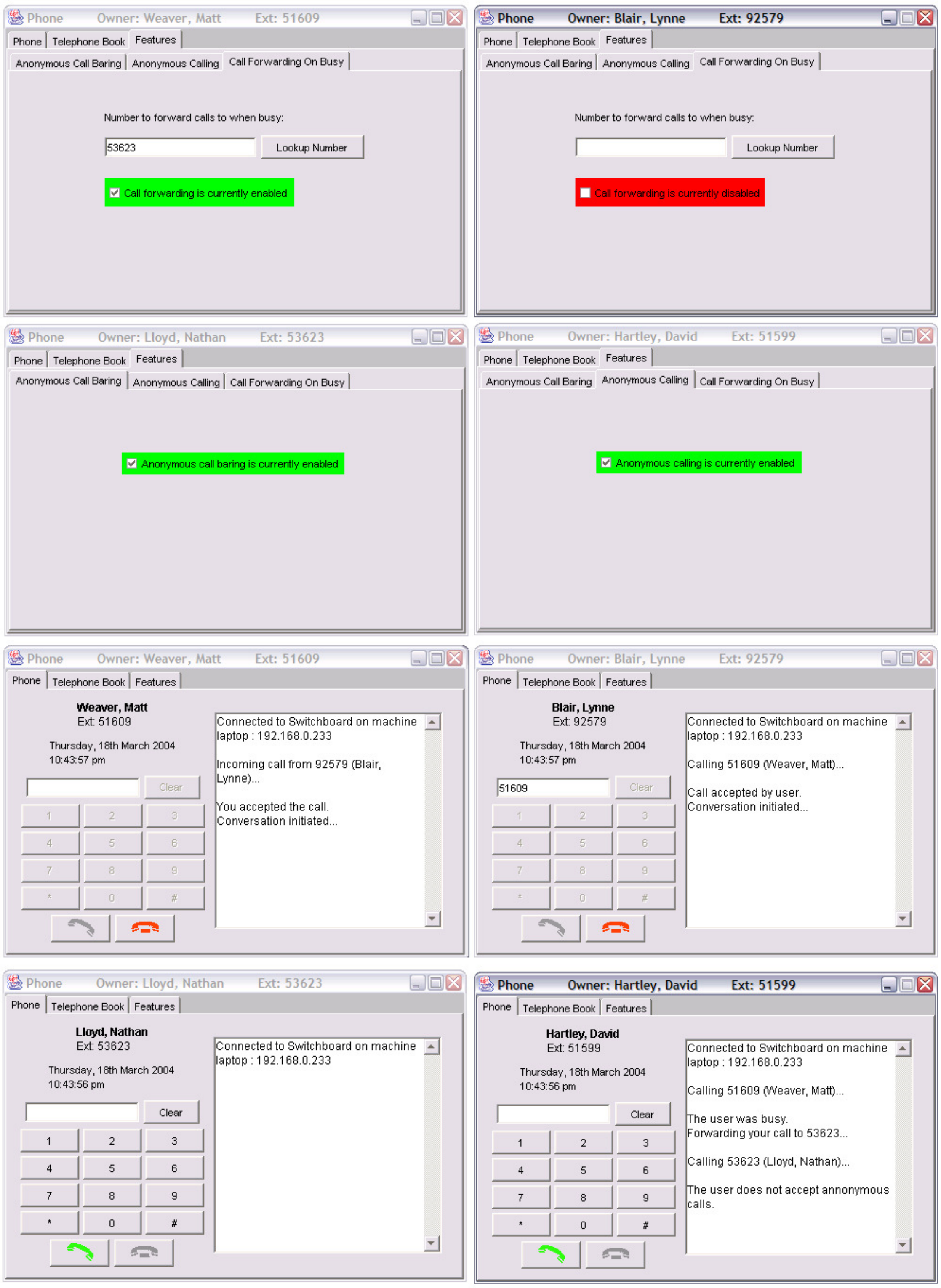
Lloyd, Nathan
Ext: 53623

Thursday, 18th March 2004
10:36:56 pm

```
Connected to Switchboard on machine
laptop : 192.168.0.233
```

Figure 6.18 ***The anonymous calling and anonymous call barring features in operation***

Figure 6.19 below shows how the call forwarding on busy, anonymous calling and anonymous call barring features can be used together.



Phone Owner: Weaver, Matt Ext: 51609
Phone | Telephone Book | Features
Anonymous Call Baring | Anonymous Calling | Call Forwarding On Busy
Number to forward calls to when busy:
53623 Lookup Number
☑ Call forwarding is currently enabled

Phone Owner: Blair, Lynne Ext: 92579
Phone | Telephone Book | Features
Anonymous Call Baring | Anonymous Calling | Call Forwarding On Busy
Number to forward calls to when busy:
Lookup Number
☐ Call forwarding is currently disabled

Phone Owner: Lloyd, Nathan Ext: 53623
Phone | Telephone Book | Features
Anonymous Call Baring | Anonymous Calling | Call Forwarding On Busy
☑ Anonymous call baring is currently enabled

Phone Owner: Hartley, David Ext: 51599
Phone | Telephone Book | Features
Anonymous Call Baring | Anonymous Calling | Call Forwarding On Busy
☑ Anonymous calling is currently enabled

Phone Owner: Weaver, Matt Ext: 51609
Phone | Telephone Book | Features
Weaver, Matt
Ext: 51609
Thursday, 18th March 2004
10:43:57 pm

Connected to Switchboard on machine laptop : 192.168.0.233

Incoming call from 92579 (Blair, Lynne)...

You accepted the call.
Conversation initiated...

Phone Owner: Blair, Lynne Ext: 92579
Phone | Telephone Book | Features
Blair, Lynne
Ext: 92579
Thursday, 18th March 2004
10:43:57 pm
51609

Connected to Switchboard on machine laptop : 192.168.0.233

Calling 51609 (Weaver, Matt)...

Call accepted by user.
Conversation initiated...

Phone Owner: Lloyd, Nathan Ext: 53623
Phone | Telephone Book | Features
Lloyd, Nathan
Ext: 53623
Thursday, 18th March 2004
10:43:56 pm

Connected to Switchboard on machine laptop : 192.168.0.233

Phone Owner: Hartley, David Ext: 51599
Phone | Telephone Book | Features
Hartley, David
Ext: 51599
Thursday, 18th March 2004
10:43:56 pm

Connected to Switchboard on machine laptop : 192.168.0.233

Calling 51609 (Weaver, Matt)...

The user was busy.
Forwarding your call to 53623...

Calling 53623 (Lloyd, Nathan)...

The user does not accept annonymous calls.

Figure 6.19 ***All three features interacting together***

Chapter 7. Evaluation

This chapter completes the project by performing an overall evaluation of aspect-oriented programming techniques. The evaluation will largely be based on the writer's own personal experience of using AspectJ to add telephony features to a basic phone application, but will also draw on other related work that is currently being carried out by enthusiasts in the aspect-oriented software design community.

Initially, a quantitative approach is used to generate interesting statistics from the work carried out by this project, namely in comparing recent aspect-oriented techniques to traditional object-oriented methods and determining how successful each methodology was at incorporating the call forwarding on busy logic into the underlying object-oriented application. What follows is a qualitative discussion highlighting the relative value of aspect-oriented programming techniques in the areas of usability, performance, modularity, maintainability and reusability.

7.1 Source code statistics

Table 7.1 below shows a count of the number of non-commented source statements for each class and aspect used to implement the two versions of the system. The classes/aspects that differed across the implementations are highlighted in **bold**.

Class / Aspect Name	Number of lines (Object-oriented)	Number of lines (Aspect-oriented)
CancelCall	22	22
ChatWindow	145	145
DateAndTime	116	116
HangupListener	9	9
IncomingCall	55	55
IncomingCallFlasher	13	13
IncomingCallListener	**48**	**44**
IncomingCallRinger	13	13
LookupFrame	**32**	**Not used**
NewPhoneListener	40	40
OutgoingCall	**99**	**87**
Phone	43	43
PhoneException	4	4
PhoneList	10	10
PhoneListImpl	89	89
PhoneUI	**610**	**544**
Semaphore	15	15
SplashWindow	32	32
Switchboard	86	86
SwitchboardLocator	43	43
TelephoneBook	**378**	**369**
TelephoneListModel	29	29
TelephoneListSelectionModel	15	15
FeaturesAvailable (aspect)	**Not used**	**8**
CallForwardingOnBusy (aspect)	**Not used**	**151**
Total	**1946**	**1982**

Table 7.1 ***Source code comparison between the OOP and AOP implementation techniques***

Counting the non-commented source statements is fairly common practice and there are various tools available for benchmarking Java classes. However, there are no such tools available yet that are capable of contrasting aspects to classes or even comparing aspects to one another. The method used to obtain the results in table 7.1 was a manual one, where method signatures, method content and import statements were all counted to give a total number of lines per class/aspect.

It is interesting to note that the aspect-oriented approach to adding the call forwarding on busy feature contained more lines of code than its object-oriented equivalent. Part of the reason for this is the inclusion of additional import statements at the beginning of the CallForwardingOnBusy aspect. The need for the aspect to implement the ActionListener and WindowListener interfaces meant that duplicate code was replicated in both PhoneUI and CallForwardingOnBusy. Pointcut declarations were also needed, as were tests based on if statements to determine if the arguments extracted from the join point were the ones that needed to be altered.

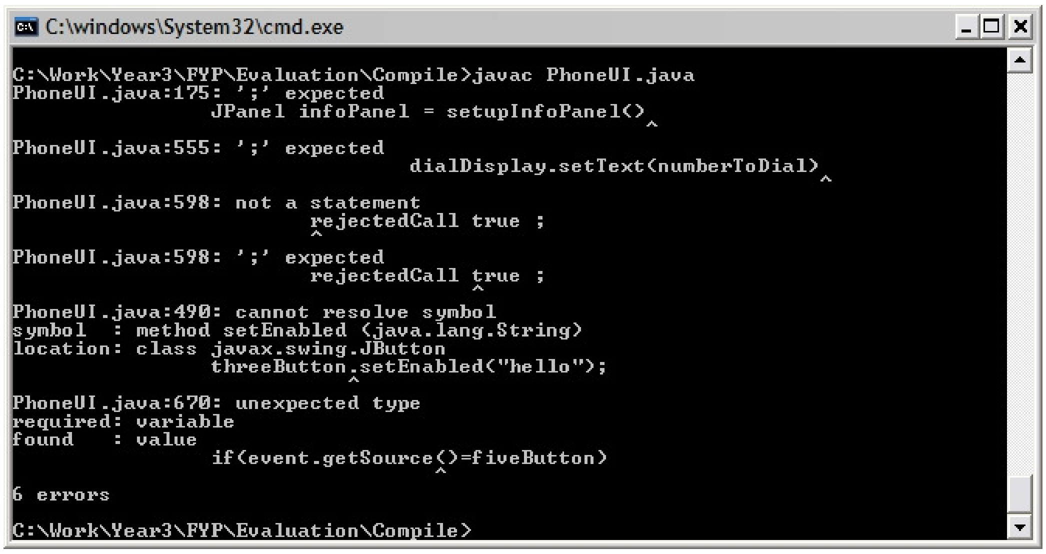
7.2 Usability

The usability of aspect-oriented programming refers to the ease at which cross-cutting concerns can be cleanly encapsulated into a single entity by a programmer using the aspect-oriented design paradigm. This section reflects on the writer's own experience with AspectJ, an AOP language binding for the Java platform.

7.2.1 Using AspectJ

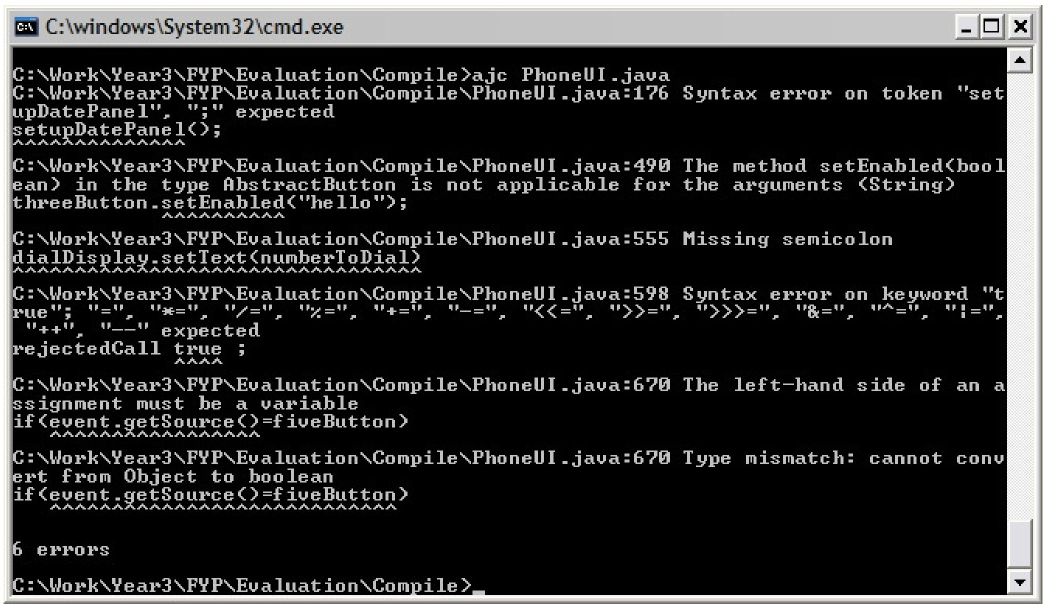
It was mentioned in the short comparison of AOP and OOP techniques in section 5.3 that the process of learning the new language semantics introduced by AspectJ was tedious and cumbersome. Several hours were spent writing a simple "Hello World!" application, using a single pointcut to pick up on all calls to System.out.println(..), which proved ridiculously difficult.

The *ajc* compiler can be used to compile both standard Java class files and AspectJ aspects independently of one another, but primarily it serves to weave aspects into the underlying application's bytecode (as discussed in section 2.6). *ajc* is naturally slower at compiling than *javac*, as it must individually compile both the class file and the aspect file in addition to weaving them together. The main problem encountered when compiling with *ajc* was the output generated in the console window when the source code contained syntactical errors. The errors were correctly identified by *ajc*, however, the error messages produced to indicate the source of error were appallingly structured. For someone who is used to the clear and concise error messages produced by the *javac* compiler, the writer found it increasingly difficult to speedily debug simple errors, such as the slip of a semicolon or a missing parenthesis. Figures 7.1 and 7.2 on the next page show the output generated by the *javac* and *ajc* compilers respectively when used to compile the same erroneous version of the PhoneUI class.



```
C:\Work\Year3\FYP\Evaluation\Compile>javac PhoneUI.java
PhoneUI.java:175: ';' expected
                JPanel infoPanel = setupInfoPanel()
                                                   ^
PhoneUI.java:555: ';' expected
                                dialDisplay.setText(numberToDial)
                                                                 ^
PhoneUI.java:598: not a statement
                        rejectedCall true ;
                        ^
PhoneUI.java:598: ';' expected
                        rejectedCall true ;
                                     ^
PhoneUI.java:490: cannot resolve symbol
symbol  : method setEnabled (java.lang.String)
location: class javax.swing.JButton
                threeButton.setEnabled("hello");
                           ^
PhoneUI.java:670: unexpected type
required: variable
found   : value
                if(event.getSource()=fiveButton)
                                  ^
6 errors

C:\Work\Year3\FYP\Evaluation\Compile>
```

Figure 7.1 *The clear and concise output produced by Sun Microsystems javac compiler*



```
C:\Work\Year3\FYP\Evaluation\Compile>ajc PhoneUI.java
C:\Work\Year3\FYP\Evaluation\Compile\PhoneUI.java:176 Syntax error on token "set
upDatePanel", ";" expected
setupDatePanel();
^^^^^^^^^^^^^^
C:\Work\Year3\FYP\Evaluation\Compile\PhoneUI.java:490 The method setEnabled(bool
ean) in the type AbstractButton is not applicable for the arguments (String)
threeButton.setEnabled("hello");
            ^^^^^^^^^^
C:\Work\Year3\FYP\Evaluation\Compile\PhoneUI.java:555 Missing semicolon
dialDisplay.setText(numberToDial)
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
C:\Work\Year3\FYP\Evaluation\Compile\PhoneUI.java:598 Syntax error on keyword "t
rue"; "=", "*=", "/=", "%=", "+=", "-=", "<<=", ">>=", ">>>=", "&=", "^=", "|=",
 "++", "--" expected
rejectedCall true ;
             ^^^^
C:\Work\Year3\FYP\Evaluation\Compile\PhoneUI.java:670 The left-hand side of an a
ssignment must be a variable
if(event.getSource()=fiveButton)
   ^^^^^^^^^^^^^^^^^
C:\Work\Year3\FYP\Evaluation\Compile\PhoneUI.java:670 Type mismatch: cannot conv
ert from Object to boolean
if(event.getSource()=fiveButton)
   ^^^^^^^^^^^^^^^^^^^^^^^^^^^


6 errors

C:\Work\Year3\FYP\Evaluation\Compile>_
```

Figure 7.2 *The poorly structured and cluttered output produced by AspectJ's ajc compiler*

7.2.2 Tool support

The object-oriented design paradigm is well into its maturity stages as a methodology for representing a problem domain as a set of objects that interact with one another by exchanging messages. As one might expect, there is an extensive range of development tools available to assist a programmer in his or her approach to designing an object-oriented system. These come in many flavours but can be generalised into groups of utilities such as editors, debugging utilities and visual aids. Integrated development environments (IDEs) combine these tools into a single application. IBM's Eclipse project is emerging as a well-known and powerful IDE with a large support base.

Aspect-oriented system design is still in its infancy stage. In comparison to OOP, there are relatively few tools available that support the aspect-oriented development process. Any worthy Java editor will support the notion of syntax highlighting (where keywords are highlighted in different colours) and provide a code auto-complete facility (where the program predicts the name of standard methods from the Java API and offers a list of possible choices). Both of these features help to reduce programmatic errors when coding applications. A standard AOP implementation for the Java platform is yet to emerge, and hence there is no common aspect-oriented syntax that can be adopted by editors to support useful features such as syntax highlighting and code auto-complete. Some applications do offer these features but they are specific to a particular implementation of AOP, such as AspectJ or AspectWerkz.

New programmers who wish to program with Java and AspectJ can use the Eclipse IDE to support their design process. A set of AspectJ development tools are available as a plug-in to Eclipse and provide a complete AspectJ development environment. There are a large number of programmers, the writer included, who prefer to use a Java-aware text editor and perform compilation manually using *javac* and *ajc* commands from a console. At the time of writing, relatively few of these simpler editors support syntax highlighting and code auto-complete.

7.2.3 The aspect-oriented development process

Currently, there is no agreed standard as to what approach to take when designing an entirely aspect-oriented system. The unified modelling language (UML) is standardised for object-oriented design and Warren (2003) reminds us how it can be used to “visualise, specify, construct and document software-intensive systems”. In the AOP world, there is no such notation for blueprinting aspect-oriented systems. This is increasingly becoming of great concern to the aspect-oriented development community, as the AOP ideals of pointcuts, join points and advice need stringent constraints and consistent representations if they are to be globally adopted.

7.3 Performance issues

It is important to address the issue of performance when adding features to a working application using an aspect-oriented approach. Whilst an AOP approach might lead to a an application having totally separated concerns and eliminate the problems of code tangling and code scattering, the overall efficiency of the system should not suffer as a consequence.

An investigation was performed to determine the different file sizes produced by the *javac* and *ajc* compilers. *javac* was used to compile the object-oriented version of the system, initially without any features implemented and subsequently with the call forwarding on busy functionality included in the relevant classes. *javac* was also used to compile the aspect-oriented version of the system without any features added, followed by an *ajc* compile of the same files for comparison. *ajc* was then used to compile just the FeaturesAvailable and CallForwardingOnBusy aspects and finally used to perform a weave of the Java class files with the AspectJ aspects. Table 7.2 on the next page summarises the resulting file sizes.

Approach	Features implemented	Type of files compiled	Compiler	File size (bytes)
Object-oriented	None	Java class files (*.java)	javac	76,598
	Call Forwarding On Busy	Java class files (*.java)	javac	81,355
Aspect-oriented	None	Java class files (*.java)	javac	76,598
	None	Java class files (*.java)	ajc	88,176
	Call Forwarding On Busy	Aspect files (*.aj)	ajc	16,363
	Call Forwarding On Busy	Aspect files (*.aj) weaved with java class files (*.java)	ajc	134,822

Table 7.2 ***Summary of the file sizes generated by javac and ajc when used to compile the source code of the object-oriented and aspect-oriented applications***

The file sizes generated by the various compile operations are interesting to compare. The difference between the *javac* and *ajc* compiles of the basic source code without any features included is surprising in itself; *ajc* produced 15% more bytecode than *javac*.

One would think that the total size of the woven application would be equal to the sum of the *ajc* compile of the java classes plus the *ajc* compile of the aspects. If this were true, the resulting file size would be 88,176 + 16,363 which equals 104,539 bytes. The actual woven total was 134,822 bytes which is a difference of 30,283 bytes. Comparing the file size generated by the *javac* compile of the object-oriented source code with the call forwarding feature included (81,355) and comparing it to the total *ajc* woven file size of the aspect-oriented version (134,822) it can be deduced that 65% additional bytecode is created by using an aspect-oriented approach with AspectJ.

The reason for this large increase in bytecode is due to the way in which the *ajc* compiler weaves the aspect code into the underlying Java class files. Join points in AspectJ are mapped to regions of bytecode which are matched by AspectJ pointcuts, and in essence advice blocks are implemented within these regions. A paper submitted for the 2004 Aspect-Oriented Software Development conference (*http://hugunin.net/papers/aosd-2004-cameraReady.pdf*) extensively discusses the performance impacts of weaving with version 1.1 of the AspectJ compiler. The main issue to note is that the reflection API in the current version of AspectJ contains a deficiency that will be fixed in the 1.2 release using a work around from the underlying Java reflection implementation.

7.4 Modularity

The modularity of aspect-oriented techniques refers to how well AOP can encapsulate the requirements of cross-cutting concerns into well-defined entities.

The work performed in this project has demonstrated a clear motivation for the need to better modularise functionality within applications and has taken significant effort in showing how a telephony feature that cross-cuts a basic phone application can be separated out into a single concern using an aspect. Figure 7.3 on the following page shows a comparison between the object-oriented and aspect-oriented versions of the application with the call forwarding on busy feature implemented.

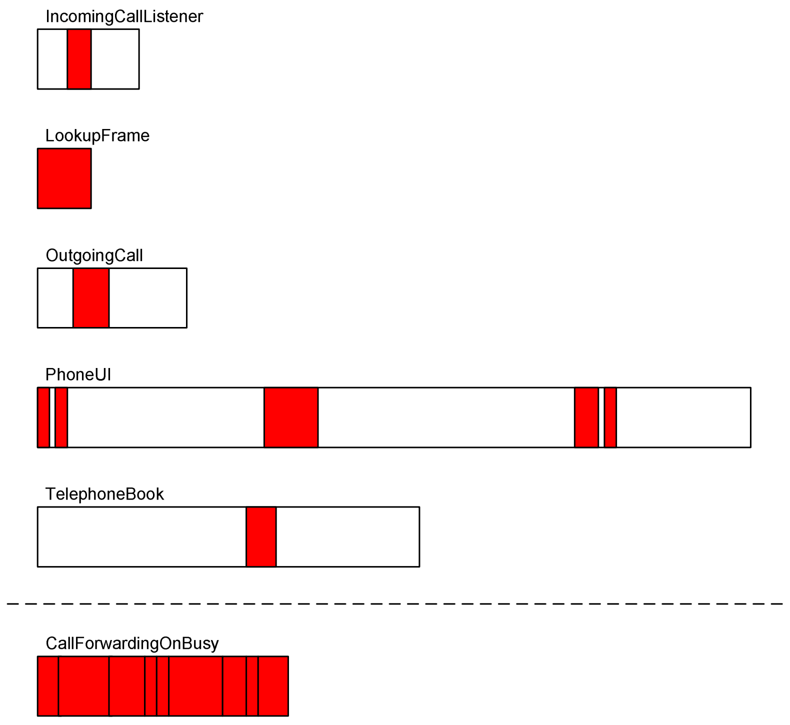


Figure 7.3 ***Comparison of the two approaches to adding the call forwarding on busy feature***

The rectangles in figure 7.3 above represent the relative sizes of the classes that had to be cross-cut in order to implement the call forwarding on busy feature using an object-oriented design pattern. The shaded sections highlight the changes that had to be made to the existing classes, showing the proportion of change required to each class. Depicted underneath the dotted line is the equivalent CallForwardingOnBusy aspect that was created to encapsulate all the functionality associated with forwarding calls to another user in an aspect-oriented manner.

Adding the feature using the two methodologies clearly shows that an aspect-oriented approach can modularise the necessary functionality required to implement a particular cross-cutting concern more cleanly than any technique available in the object-oriented paradigm.

7.5 Maintainability

The maintainability of programs created using an aspect-oriented development approach refers to how easily they can be updated in the future, perhaps to exploit new functionality, extend the business logic or provide compatibility with new applications and services.

The overhead of maintaining a particular concern in an aspect-oriented system is far less than that involved in maintaining an equivalent object-oriented system. AOP techniques drastically reduce the amount of scattered and tangled code present in an application by encapsulating all the state and behaviour associated with a concern into a single aspect.

This task of maintaining this code is relatively lightweight, as all the code is in one place. Should the call forwarding on busy feature need to change, perhaps to forward calls to voice mail instead of another user, then all the modifications would be made within the CallForwardingOnBusy aspect, leaving the remaining classes untouched.

In much larger systems it becomes increasingly difficult to deploy new functionality and features to every single application in a system, primarily due to time and cost constraints. It is likely that upgrades would be performed incrementally over a period of time. The aspect-oriented methodology allows for features to be developed independently of the core application and weaved-in to the underlying classes when needed. A particular subset of applications in a distributed system could be upgraded with new features, leaving other applications untouched until it becomes economically viable to issue the update.

7.6 Reusability

The reusability of an aspect-oriented system relates to how effectively a given component from the system can be used in a different context, programming environment or application with little or no modification.

The behavioural and structural patterns that can be applied to an object can also be used to define aspects. An aspect can be abstract in the same way that a class can, so that common functionality can be abstracted-out and shared by sibling sub classes. Code created to satisfy aspect-oriented design principles is more modular and more cleanly encapsulated into a single concern. It can be seen from the extensive work carried out in the object-oriented design community that well-defined code of this nature tends to be more reusable than code that is poorly defined and scattered across an entire application.

7.7 Other work

Research by Mickelsson (2002) has shown that aspect-oriented techniques are not just beneficial when additional features are added to an existing application but also when an application is designed using entirely aspect-oriented techniques. Mickelsson's work involved creating two applications from scratch, firstly by following the well-defined object-oriented design paradigm and secondly by using a less-formal aspect-oriented approach with AspectJ.

It is interesting to note that some of the problems that Mickelsson identified with AOP have been solved, such as the lack of incremental compiling, which has since been addressed in version 1.1 of AspectJ. Mickelsson's work also addressed some important issues which still have not been resolved at the time of writing. The most fundamental of these is the lack of a standardised modelling language (such as UML) to adopt aspect-oriented theory and provide mechanisms for representing the cross-cutting nature of aspect code.

7.8 Summary

This evaluation chapter started by examining the source code for the basic phone application with the call forwarding on busy aspect implemented as both an object and an aspect. Statistics were generated detailing the number of non-commented source statements used in each approach and the size of the class files generated by the *javac* and *ajc* compilers.

The discussion moved on to evaluate the relative value of aspect-oriented programming methods in the particular areas of usability, performance, modularity, maintainability and reusability. Finally, related work performed by Mickelsson was studied to draw out important similarities and differences with the results from this project.

The key findings from this evaluation are outlined below.

- The aspect-oriented version of the application, with the addition of the call forwarding on busy feature, contained more lines of code than that of the equivalent object-oriented implementation (with the aspect version containing 1.8% more code)
- The aspect-oriented version of the application weaved with *ajc* contained 65% more bytecode than the object-oriented equivalent version compiled with *javac*
- The AspectJ compiler, ajc, was noted for its poorly structured and cluttered output when presenting indications of source errors to the user
- The slight lack of tool support for aspect-oriented development was noted, with great emphasis placed on the need for a standardised modelling language (such as UML) to adopt aspect-oriented principles and provide structures for representing the cross-cutting nature of aspect code and the concepts of pointcuts, join points and advice
- Separating out the call forwarding on busy functionality as an aspect using AspectJ demonstrated that aspect-oriented approaches to software development can encapsulate the functionality of a cross-cutting concern more cleanly and than their object-oriented counterparts
- Encapsulating all of the logic associated with a particular concern into a single aspect makes for an application that is more maintainable, allowing functionality to be incrementally added to large applications in a cost-effective manner
- Aspects, like objects, can apply the same structural and behavioural patterns that exist in the object-oriented paradigm to ensure maximum reusability of individual system components

Chapter 8. Conclusions

This concluding chapter revisits the aims and objectives that were set out in the introductory chapter and shows how, and where, they have been met by the project. Revisions to the basic phone system that should be made if the project is to be continued on a larger scale are considered. A brief statement of what the writer has achieved by carrying out the project is made before determining if aspect-oriented programming truly solves the problems associated with current object-oriented techniques.

8.1 Aims and objectives revisited

Objective 1	*To design a basic model of a working telephone system following an object-oriented design pattern*
Objective 2	*To model the telephone system as a distributed client/server application, with distributed phone clients and a centralised switchboard acting as a server*
Objective 3	*To provide each virtual phone client with a friendly graphical user interface and a look-and-feel that resembles a typical phone from the real-world*
Objective 4	*To incorporate fault-tolerant mechanisms within the system, allowing the switchboard to be relocated in the event of failure whilst causing minimal impact to the operation of the phone network*
Objective 5	*To extend the basic application with a dynamic telephone book capable of automatically maintaining an up-to-date list of all the phones online in the system*
Objective 6	*To gain a detailed understanding of the concepts behind an aspect-oriented approach to software design*
Objective 7	*To layer a chosen feature on top of the phone application as both an object and an aspect*
Objective 8	*To perform a critical review of the procedures involved when using OOP and AOP techniques to add a feature to the existing phone application*
Objective 9	*To add additional features to the phone application using the favoured technique*
Objective 10	*To conclude the project by performing an evaluation of aspect-oriented techniques, stating the extent to which AOP solves the problems associated with OOP*

Table 8.1 below highlights the sections of the report where the aims and objectives of the project have been met.

Obj.	How it was met	Relevant sections	Page Numbers
1	Detailed design considerations discussed	Chapter 3	14 - 29
2	Phone-to-switchboard connectivity achieved with Java RMI Phone-to-phone connectivity established with TCP connections	3.1	14 - 20
3	Existing Voice-over-IP solutions identified Extracted the noteworthy characteristics of each system Simple user interface constructed with Java Swing components	2.9 3.3.8 4.2.1	10 – 12 26 - 28 36 - 37
4	Switchboard location broadcast out to all phones every 5 seconds Each phone listens for broadcast messages in the background Switchboard can dynamically re-build itself after a failure	3.2.2 4.1.2 4.1.4	21 – 22 31 – 32 33 - 35
5	Discussion of possible approaches to building the telephone book Book realised as a JList supported by broadcast messages	3.3.7 4.2.18 – 4.2.22	26 43 - 45
6	Detailed discussion of problems with OOP and motivation for AOP Explanation of specific AspectJ language semantics	2.1 - 2.6 5.1	4 – 9 46 - 48
7	Call forwarding on busy added as the first feature	5.2	49 - 54
8	Short comparison of OOP and AOP techniques	5.3	54 - 55
9	Anonymous calling added as a second feature Anonymous call barring added as a third feature	5.4 5.5	56 – 57 57 - 59
10	Short comparison of OOP and AOP techniques Detailed evaluation of AOP and statistics from the source code	5.3 Chapter 7	54 – 55 70 - 77

Table 8.1 *Summary of how the aims and objectives were met by the project*

8.2 Suggested extensions to the project

There are a number of possible extensions to the project that could be made if it were to be repeated on a larger scale.

An interesting adaptation would be to extend the functionality of the basic phone application by providing real-time voice and video communications with SIP using an aspect-oriented approach. Each type of communication could be modelled as an aspect to fully encapsulate all of the state and behaviour associated with voice and video transmissions. It would be interesting to examine such a system and determine the efficiency of using a real-time protocol such as SIP in conjunction with aspect-oriented design techniques.

Further to the work performed by this project, a completely different target application could be chosen and modelled entirely in both an object-oriented and aspect-oriented manner. This would provide the evaluation of AOP techniques with more quantitative data to support the argument for or against the methodology and give a true comparison of OOP and AOP design methodologies.

Other challenging extensions include studying the heavily research-based area of dynamic aspects. A dynamic aspect is an aspect that changes its behaviour based on varying conditions that occur at run-time. Currently there is no standard language structure for specifying dynamic aspects but great power and flexibility is represented in the ability to dynamically change the way in which a program operates based on non-deterministic characteristics.

8.3 Learning outcomes

Before this project was undertaken, the writer had no knowledge of aspect-oriented programming techniques whatsoever. In a period of about one year, a wealth of knowledge has been accumulated and great pleasure has been experienced in putting new concepts to practical use in Java a programming environment. The learning outcomes from this project are summarised below.

- Aspect-oriented design principles have been extensively researched from many viewpoints
- New knowledge of IP telephony, Voice-over-IP and real-time communications protocols such as SIP and H.323 has been obtained
- Significant obstacles in distributed systems programming have been overcome using Java Remote Method Invocation and TCP socket connections
- The writer's confidence in programming in an object-oriented Java environment has been increased through persistence and dedication

8.4 So, does AOP solve the problems associated with OOP?

This project has spent considerable time and effort discussing aspect-oriented approaches to software development and implementing telephony features on top of a basic telephone application as a means to provide value to an evaluation of OOP and AOP techniques. The question as to whether AOP actually solves any of the problems associated with OOP has yet to be clearly answered; this final section addresses that challenge.

There is little doubt as to the benefit of being able to separate out cross-cutting concerns into individual units of functionality that represent aspects of an entire system, such as logging or authentication. The evaluation section showed how encapsulating all of the state and behaviour associated with a particular concern within a single entity removed the majority of scattered and tangled code from the underlying application. However, a slight "cheat" had to be performed by adding a dummy method to the PhoneUI class so that a pointcut could be adequately defined for a specific location of particular interest to the call forwarding on busy feature. This was not seen as a major problem in the context of adding such a small amount of code to the phone application. It is unclear, though, as to what the impact of this technique would be in much larger systems, but the very notion of having to modify object-oriented code in order to implement a completely different concern should be somewhat frowned upon.

Extensive research performed by professionals such as Mickelsson (2002) and Gradecki (2003) has shown that AOP has already solved some of the problems that developers encounter when translating a problem into a software model. In particular, Mickelsson notes how AOP is mature enough as a technology to be used in larger development projects and in development environments that harness the advanced power of the Java 2 Enterprise Edition platform.

What AOP will never solve is the problem of bad coding, i.e. code that is inevitably written incorrectly by a programmer. There is not a single person in the world that has not accidentally omitted a semicolon off the end of a statement or misplaced a closing bracket at some point in their programming career. This goes to show that humans, in their current state of evolution, will always make random mistakes. Whilst this problem cannot be solved, the impact of it can be considerably reduced by employing a standard convention for designing software solutions in an aspect-oriented manner. IDEs and editors would benefit from a standardised implementation of AOP by being able to incorporate it into their applications for the purpose of providing useful features such as syntax highlighting and code auto-complete, thus reducing the number of errors that we inherently make as humans when coding our applications.

The aspect-oriented software development conference is held every year and brings leading researchers and practitioners together to discuss the latest developments in the field of AOP. The conference is entirely devoted to aspect-oriented technologies and practices and exists to provide demonstrations of leading edge developments, such as dynamic aspects, and hands-on tutorials at varying levels of detail. The current downfalls with AOP are being addressed by the aspect-oriented software development community. Perhaps the most significant drawback concerned with implementing AOP is that no standard convention exists for modelling the cross-cutting nature of concerns. It is speculated, however, that the next release of the unified modelling language (UML 2.0) will include constructs for representing the fundamental AOP principles of join points, pointcuts and advice.

The future of aspect-oriented programming looks very bright indeed. Since its initial definition in 1997, a number of language bindings have been released to support the AOP methodology in different environments. The most popular of these is AspectJ which is in the process of being developed into a new 1.2 revision and is supported by IBM's Eclipse project. To quote Mickelsson, "AOP should be regarded as a very powerful tool for improving application development productivity, that, if used correctly, will pose a considerable business advantage to those who incorporate it into their development process". The impact of this is being realised already, as manufacturers of IDEs that fail to incorporate tools to support the aspect-oriented development process are losing out on revenue, as their customers are losing faith and converting over to better-supported open source environments such as Eclipse.

References

Books

Coulouris G., Dollimore J. & Kindberg T. (2001) "Distributed Systems Concepts And Design", Addison-Wesley Publishers Limited, Essex

Garside R. & Mariani J. (1998) "Java: First Contact", Course Technology, Cambridge

Grosso W. (2002) "Java RMI", O'Reilly, California, USA

Gradecki J.D. & Lesiecki N. (2003) "Mastering AspectJ", Wiley Publishing, Indiana, USA

Warren I. (2003) "Csc222/3 Software Design: Course Notes", Lancaster University

Web Resources

Colston T. (2004) "Java Tip 104: Make a splash with Swing", Online publication, JavaWorld, http://www.javaworld.com/javaworld/javatips/jw-javatip104.html

McKeon B. (2004) "A Tutorial on IP Multicast", Online publication, Networks and Telecommunications Research Group, http://ntrg.cs.tcd.ie/undergrad/4ba2/multicast/antony/

Mickelsson M. (2002), "Aspect-Oriented Programming compared to Object-Oriented Programming when implementing a distributed, web-based application", Online publication, Uppsala University, http://www.bluefish.se/aop/aop_magnus_mickelsson.pdf

Taylor S. & Hettick L. (2004), "PSTN security isn't exactly airtight", Online publication, Network World Fusion, http://www.nwfusion.com/newsletters/converg/2004/0209converge2.html

Tyson J. (2004) "How IP Telephony Works". Online publication, HowStuffWorks, http://computer.howstuffworks.com/ip-telephony.htm

Unknown (2004) "Java Sound API", Onlin publication, Sun Microsystems, http://java.sun.com/products/java-media/sound/

Unknown (2004) "jGuru: Remote Method Invocation (RMI)", Online publication, jGuru, http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Unknown (2004) "LearnIT: VoIP, Part one – the basics", Online publication, TechTarget, http://searchnetworking.techtarget.com/content/0,290959,sid7_gci932903,00.html

Unknown (2004) "SIP versus H.323", Online publication, iptel.org, http://www.iptel.org/info/trends/sip.html