11 Case Study

This chapter will bring together all of the previous chapters showing how these essential concepts work in practise through one example case study.

Objectives

By the end of this chapter you will see…

- how a problem description is turned into a UML model (as described in Chapter 6)
- an example of how typed collections can be effectively used, in particular you will see an example of the use of sets and maps (as described in Chapter 7)
- an example of polymorphism and see how this enables programs to be extended simply (as described in Chapter 4)
- a simple example of the use of inheritance and method overriding (as described in Chapter 3)
- several example of UML diagrams (as described in Chapter 2)
- finally you will see the use of the Javadoc tool (as described in Chapter 8).

The complete working application, developed as described throughout this chapter, is available to download for free as an exported Eclipse project.

Eclipse is freely available from www.eclipse.org.

This chapter consists of seventeen sections:-

1) The Problem
2) Preliminary Analysis
3) Further Analysis
4) Documenting the design using UML
5) Prototyping the Interface
6) Revising the Design to Accommodate Changing Requirements
7) Packaging the Classes
8) Programming the Message Classes
9) Programming the Client Classes
10) Creating and Handling UnknownClientException
11) Programming the Main classes
12) Programming the Interface
13) Using Test Driven Development and Extending the System

14) Generating Javadoc
15) Running the System and Potential Compiler Warnings
16) The Finished System…
17) Summary

11.1 The Problem

User requirements analysis is a topic of significant importance to the software engineering community and totally outside the scope of this text. The purpose of this chapter is not to show how requirements are obtained but to show how a problem statement is modelled using OO principles and turned into a complete working system once requirements are gathered.

The problem for which we will design a solution is 'To develop a message management system for a scrolling display board belonging to a small seaside retailer.'

For the purpose of this exercise we will assume preliminary requirements analysis has been performed by interviewing the shop owner, and the workers who would use the system, and from this the following textual description has been generated:-

Rory's Readables is a small shop on the seafront selling a range of convenience goods, especially books and magazines aimed at both the local and tourist trades. It has recently also become a ticket agency for various local entertainment and transport providers.

Rory plans to acquire an LCD message display board mounted above the shopfront which can show scrolling text messages. Rory intends to use this to advertise his own products and offers and also to provide a message display service for fee-paying clients (e.g. private sales, lost and found, staff required etc.)

Each client is given a unique ID string (e.g. "adams4") and has a name, an address, a phone number and an amount of credit in 'credit units'. A book of clients is maintained to which clients can be added and in which we can look up a client by their ID.

Each message is for a specific client and comprises the text to be displayed, the number of days for which it should be displayed and the cost of that message in units. No duplicate messages (i.e. the same text for the same client) are permitted.

A set of current messages is to be maintained: new messages can be added, the message set can be displayed on the display board, and at the end of each day a purge is performed – each message has its days remaining decremented and its client's credit reduced by the cost of the message, and any messages which have expired or whose client has no more credit are deleted from the message set.

The software is to be written before the display board is installed – therefore the connection to the board should be via a well-defined interface and a dummy display board implemented in software for testing purposes.

Given the description above this chapter describes how this problem may be analysed, modelled and a solution programmed – thus demonstrating the techniques discussed throughout this book.

11.2 Preliminary Analysis

To perform a preliminary analysis of these requirement as (described in Chapter 6) we must…

- List the nouns and verbs
- Identify things outside the scope of the system
- Identify the synonyms
- Identify the potential classes, attributes and methods
- Identify any common characteristics

From reading this description we can see that the first paragraph is purely contextual and does not describe anything specifically related to the software required. This has therefore been ignored.

List of Nouns

From the remaining paragraphs we can list the following nouns or noun phrases:-

- LCD message display board
- shopfront
- scrolling text message
- client
- ID string
- name
- address
- phone number
- credit
- credit unit
- message
- day
- book of clients
- ID
- text
- number of days
- cost of message (units)
- set of current messages
- message set
- display board
- days remaining
- client's credit
- cost of message
- software
- connection
- interface
- dummy display board

List of Verbs

and the following verbs:-

- acquire
- mount
- show
- advertise
- give (a unique ID)
- display
- add (a client)
- look up
- permit (duplicates – NOT!)
- purge
- decrement
- reduce credit
- expire
- delete
- write (the software)
- install
- implement
- test

Outside Scope of System

By identifying things outside the scope of the system we simplify the problem…

- Nouns:
  - shopfront
  - software

- Verbs:
  - acquire, mount (the display board)
  - advertise
  - give (a unique ID)
  - write, install, implement, test (the software)

The shopfront is not part of the system and it is not a part of the system to acquire and mount the displayboard. The ID is assigned by the shop owner – not the system. And writing / installing the software is the job of the programmer – it is not part of the system itself.

Synonyms

The following are synonyms:-

- Nouns:
  - LCD message display board = display board
  - scrolling text message = message
  - ID string = ID
  - credit units = client's credit = credit
  - set of current messages = message set
  - days = number of days = days remaining

- Verbs:
  - show = display

By identifying synonyms we avoid needless duplication and confusion in the system.

Potential Classes, Attributes and Methods

Nouns that describe significant entities for which we can identify properties i.e. data and behaviour i.e. methods could become classes within the system. These include:-

- Client
- Message
- ClientBook
- MessageSet
- DisplayBoard
- DummyDisplayBoard

Nouns that are would be better as attributes of a class rather than becoming a class themselves:-

- For a 'client':
  - ID
  - name
  - address
  - phone number
  - credit

- For a 'message':
  - text
  - days remaining
  - cost of message

Each of these *could* be modelled as a class (which Client or Message would have as an object attribute), but we decide that each of them is a sufficiently simple piece of information that there is no reason to do so – each one can be a simple attribute (instance variable) of a primitive type (e.g. int) or library class (e.g. String).

This is a **design judgement** – introducing classes for significant entities (Client, Message etc.) which have a set of information and behaviour belonging to them, but not overloading the design with trivially simple classes (e.g. credit which would just contain an 'int' instance variable together with a getter and setter!).

Verbs describe candidate methods and we should be able to identify the classes these could belong to. For instance:-

- For a 'client':
  - decrease credit
- For a 'message':
  - decrement days

The other verbs describing potential methods should also be listed:-

- display
- add (client to book)
- add (message to set)
- lookUp (client in book)
- purge
- decrement
- expire
- delete

For each of these the associated class should be identified.

Common Characteristics

The final step in our preliminary analysis is to identify the common characteristics of classes and to identify if these classes should these be linked in an inheritance hierarchy or linked by an interface.

Looking at the list of candidate classes provided we can see that two classes that share common characteristics:-

- DisplayBoard
- DummyDisplayBoard

This either implies these classes should be linked within the system within an inheritance hierarchy or via 'an interface'(see section 4.5 Interfaces). In this case the clue is within the description "These will have a 'connection' to the rest of the system via a 'well-defined interface".

Ultimately our system should display messages in a real display board however it should first be tested on a dummy display board. For this to work the dummy board must implement the same methods as a real display board.

Thus we should define a java 'interface'. No common code would exist between the two classes – hence why we are not putting these within an inheritance hierarchy. However the dummy board and the real display board should both implement the methods defined via a common interface. When our system is working we could replace the dummy board with the real board which implements the same methods. As the connection with the dummy board is via the interface changing the dummy board with the real display board should have no impact on our system.

From our preliminary analysis of the description we have identified candidate classes, interfaces, methods and attributes. The methods and attributes can be associated with classes.

The classes are: -

- Client
- Message
- ClientBook
- MessageSet
- DisplayBoard
- DummyDisplayBoard

The Interface is:-

- DisplayBoardControl (a name we have made up)

And the methods include:-

- display
- add (client to book)
- add (message to set)
- lookUp (client in book)
- purge
- decrement
- expire
- delete

11.3 Further Analysis

We could now document this proposed design using UML diagrams and program a system accordingly. However before doing so it would be better to find any potential faults in our designs as fixing these faults now would be quicker than fixing the faults after time has been spent programming the system. Thus we should now refine our design using CRC cards and elaborate our classes.

CRC cards (see Chapter 6 section 6.10 and 6.11) allow us to role play various scenarios to check if our designs look feasible before refining these designs and documenting them using UML class diagrams.

The two CRC cards below have been developed to describe the Client and ClientBook classes. The panel on the left shows the class responsibilities and the panel on the right shows the classes they are expected to collaborate with.

Client	
Know name, address, phone	
Know current credit	
Decrease credit	

ClientBook	
Hold list of client IDs and associated clients	Client
Add a client	Client
Look up a client	Client

We can now use these to roleplay, or test out, a scenario. In this case what happens when we get a new client in the shop? Can this client be created and added to the client book?

To do this the system must perform the following actions:-

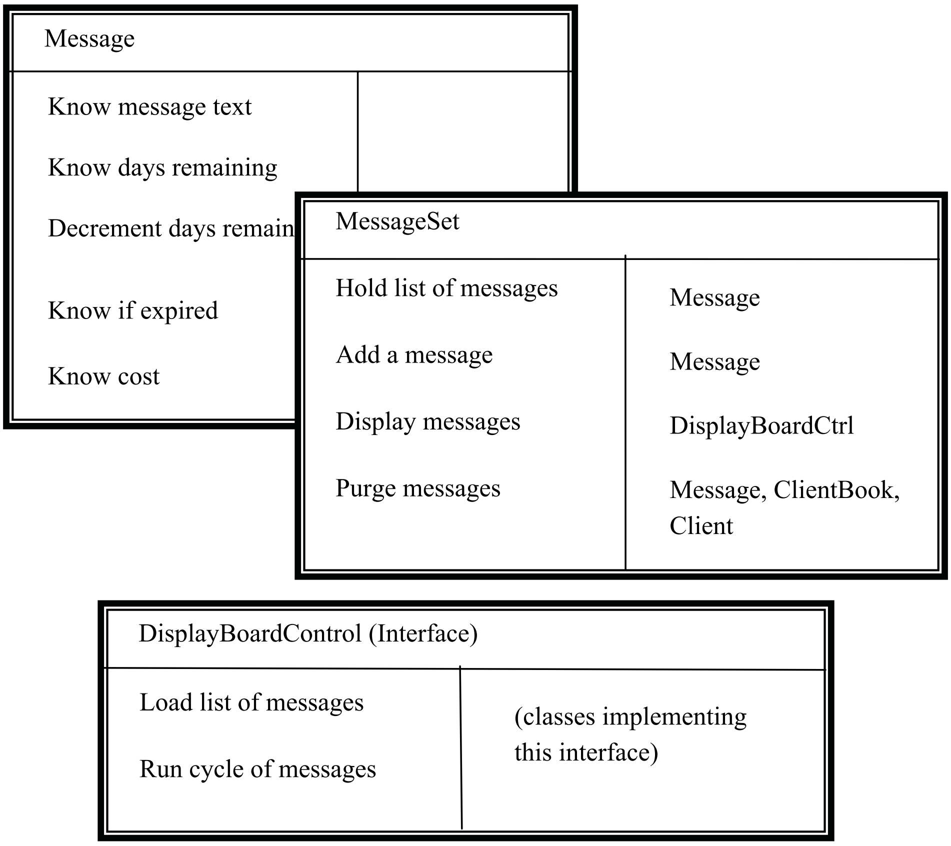
- create a new Client object
- pass it (along with the unique ID to associate with it) to the ClientBook object
- add the client to the client book.

By looking at the CRC cards we can see that:-

- the constructor for Client will be able to create a new client object
- The ClientBook has the capability to add a client and
- the ClientBook can hold the IDs associated with each client.

It would therefore appear that this part of the system will work at least in this respect – of course we need to create CRC cards to describe every class and to test the system with a range of scenarios.

Below are three CRC cards to describe the Message and MessageSet classes and the DisplayBoardControl interface.



Message	
Know message text	
Know days remaining	
Decrement days remain	
Know if expired	
Know cost	

MessageSet	
Hold list of messages	Message
Add a message	Message
Display messages	DisplayBoardCtrl
Purge messages	Message, ClientBook, Client

DisplayBoardControl (Interface)	
Load list of messages Run cycle of messages	(classes implementing this interface)

What we want to 'test' here is that messages can be created, added to the MessageSet and displayed on the display.

A point of requirements definition occurs here. There are two possibilities regarding the interface to the display board:-

a) we load one message at a time, display it, then load the next message, and so on.
b) we load a collection of messages in one go, then tell the board to display them in sequence which it does autonomously

The correct choice depends on finding out how the real display board actually works.

Note that (a) would mean a simple display board and more complexity for the "Display messages" responsibility of MessageSet, while (b) implies the converse.

For this exercise we will assume the answer to this is (b), hence the responsibilities of scrolling through a set of messages will be assigned to the DisplayBoardControl interface.

Looking at these CRC cards it would appear that we can

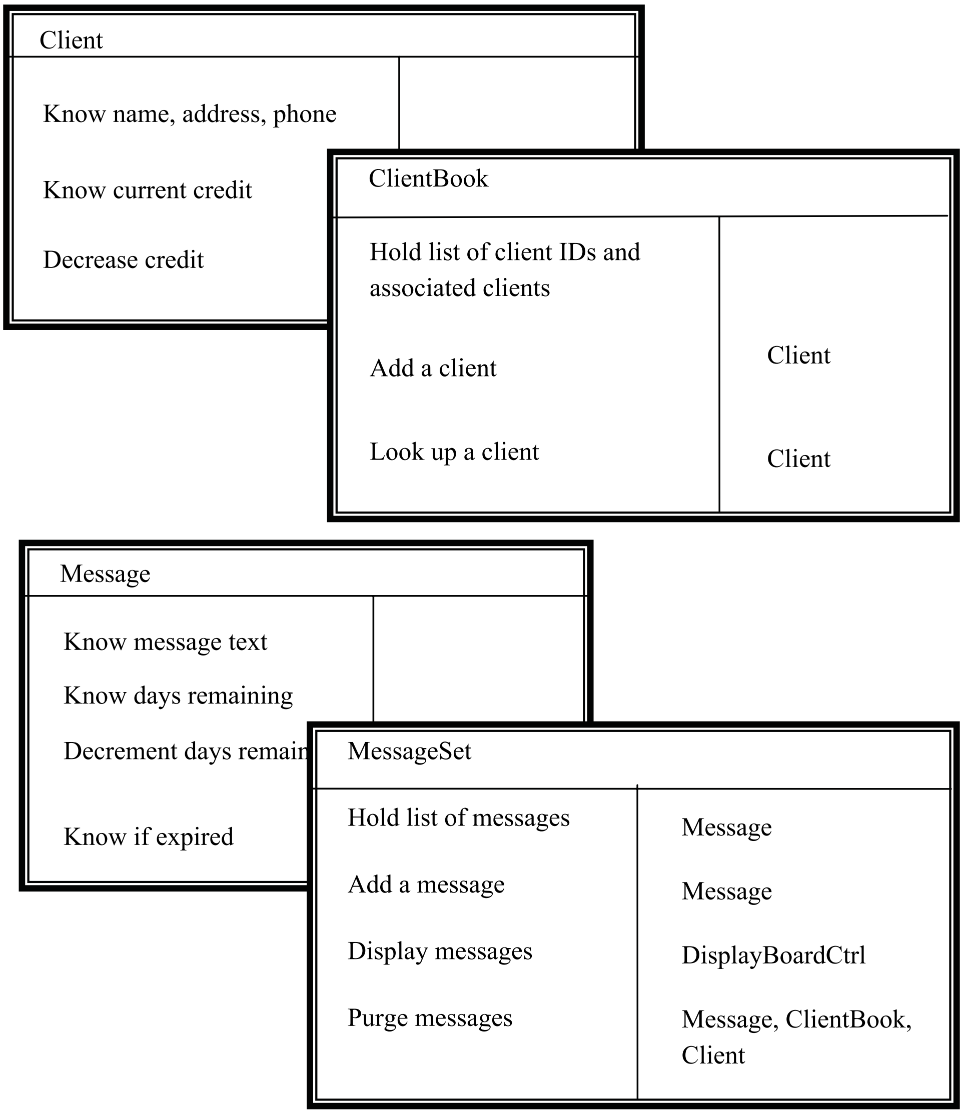
- Create a new message,
- Add this to the message set and
- Display these messages by invoking load 'list of messages' and 'run cycle of messages'

So this part of our design also seems to work.

The Message Purge Scenario

The final scenario that we want to run though here is the message purge scenario. At the end of the day when messages have been displayed the remaining days of the message need to be decremented and the message will need to be deleted if a) the message has run out of days or b) the client has run out of credit.

CRC cards for the classes involved in this have been drawn below…



Client	
Know name, address, phone	
Know current credit	
Decrease credit	

ClientBook	
Hold list of client IDs and associated clients	
Add a client	Client
Look up a client	Client

Message	
Know message text	
Know days remaining	
Decrement days remain	
Know if expired	

MessageSet	
Hold list of messages	Message
Add a message	Message
Display messages	DisplayBoardCtrl
Purge messages	Message, ClientBook, Client

Activity 1

To purge the messages, the MessageBook cycles through its list of messages reducing the credit for the client who 'owns' this message, decrementing the days remaining for that message and deleting messages when appropriate.

Looking at the CRC cards above work through the following steps and identify any potential problems with these classes:-

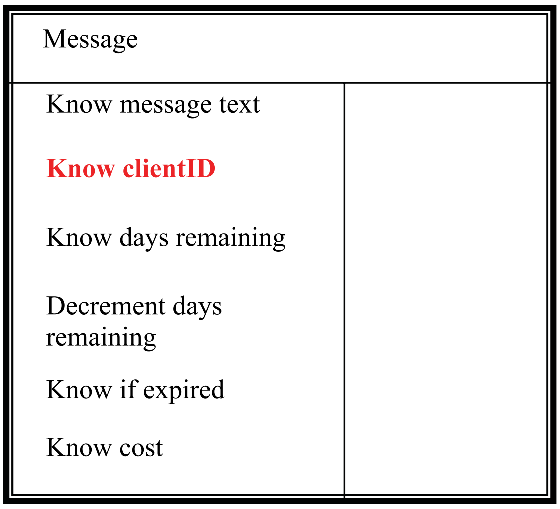
```
For each message
        - tell the Message to decrement its days remaining and
        - tell the relevant Client to decrease its credit
                - ask the Message for its client ID
                - ask the Message for its cost
                - ask the ClientBook for the client with this ID
                - tell the Client to decrease its credit by the cost of the message
        - if either the Client's credit is <= 0 or the Message is
         now expired
                delete the message from the list
```

Feedback 1

A problem becomes evident when we try to find the client associated with a message as Message does not know the client ID.

We therefore need to add this responsibility to the Message class.

A revised design for the Message class is given below…



Message	
Know message text	
Know clientID	
Know days remaining	
Decrement days remaining	
Know if expired	
Know cost	

By drawing out CRC cards for each class and interface and by role playing a range of scenarios we have checked and revised our plans for the system – we can now refine these and document these using UML diagrams.

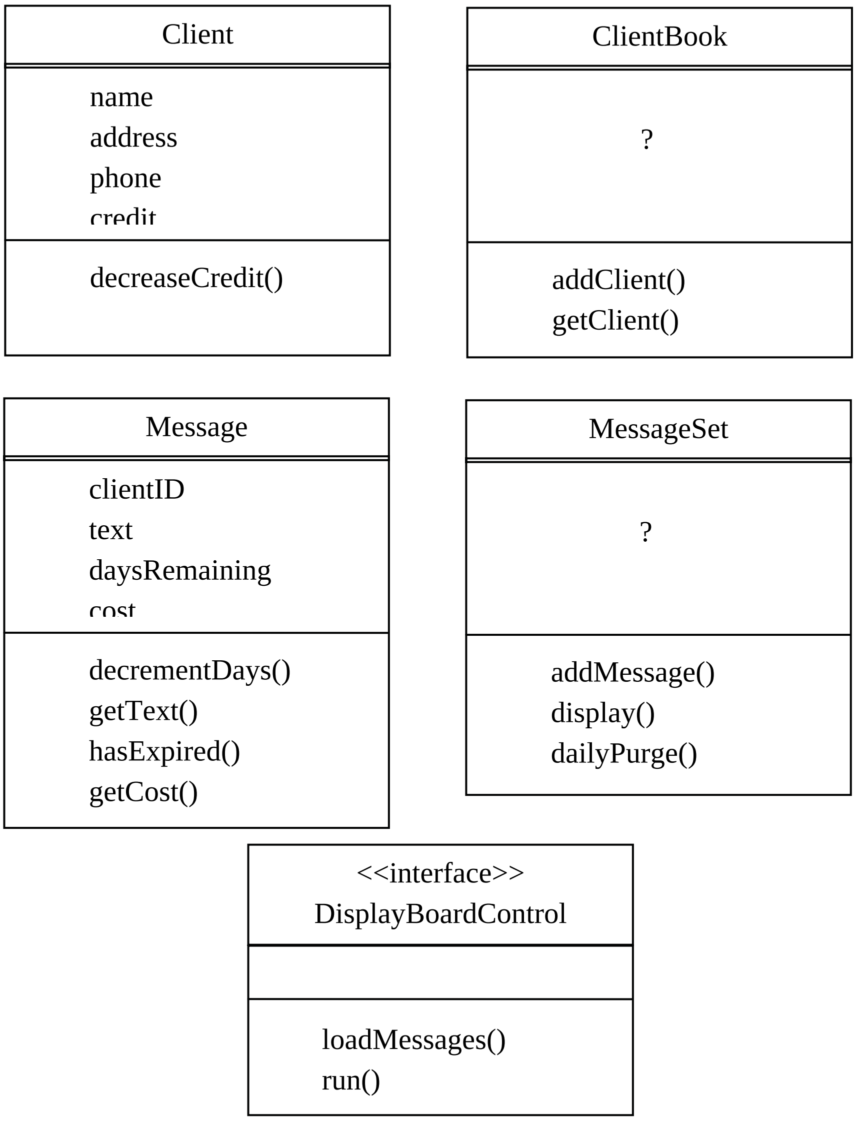
11.4 Documenting the design using UML

To fully document our designs we need to:-

- Determine in detail what attributes go in each class
- Determine how the classes are related and
- Put classes into appropriate packages.

Elaborating the Classes

Having worked through CRC scenarios we can make an initial assignment of instance variables and methods among our classes, including some accessors and mutators whose necessity has become evident (see diagram below).



We don't know of any simple attributes which ClientBook and MessageSet will require, but they will need to be associated with other classes so we still have some work to do there – hence the ?s (which are not an official part of UML)!

Relationships Between Classes

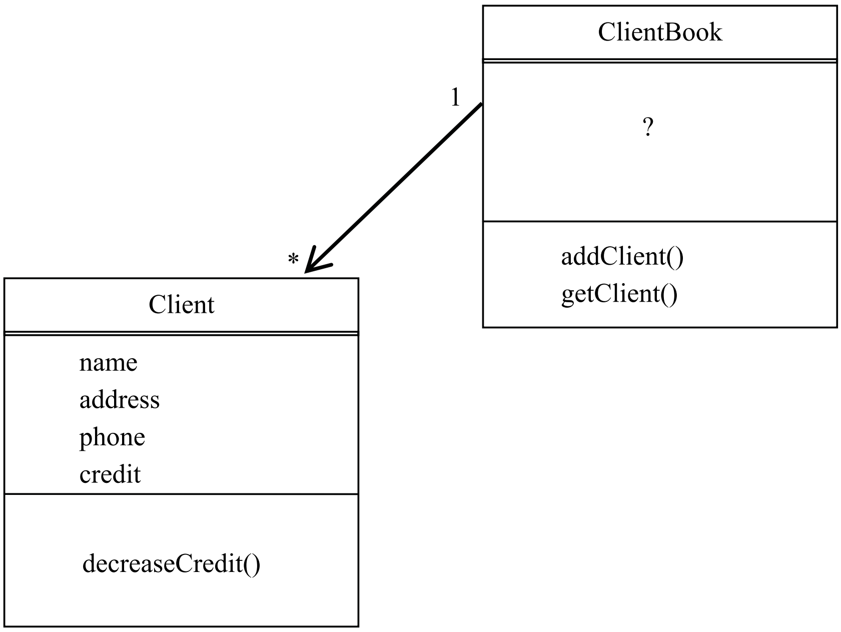
We can now start to work out how these classes are related.

Starting with ClientBook and Client:- a ClientBook will record details of zero or more clients.

The navigability will be from ClientBook to client because the book "knows about" its Clients (in implementation terms it will have references to them) but the individual Clients will not have references back to the book.

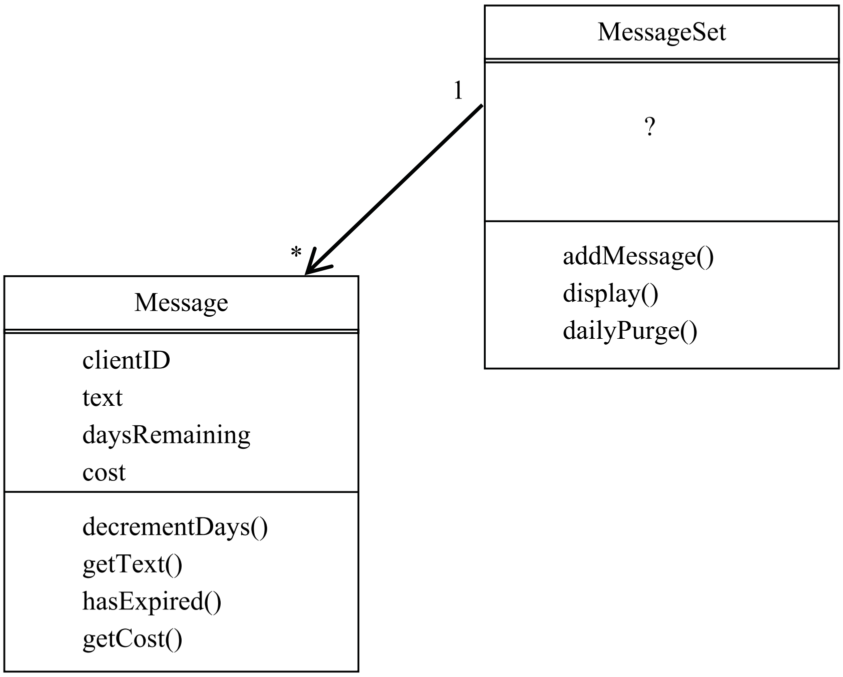
The one-to-many relationship suggests that ClientBook will have a **Collection** (of some kind) of Clients. The specification states that each Client will have a unique ID thus the collection will in fact be a map where each entry is made up of a pair of values – in this case a clientID (a string) and a Client object.

Alcatel·Lucent
www.alcatel-lucent.com/careers
What if you could build your future and create the future?
One generation's transformation is the next's status quo.
In the near future, people may soon think it's strange that
devices ever had to be "plugged in." To obtain that status, there
needs to be "The Shift".



The relationship between MessageSet and Message is very similar to the relationship between ClientBook and Clients.

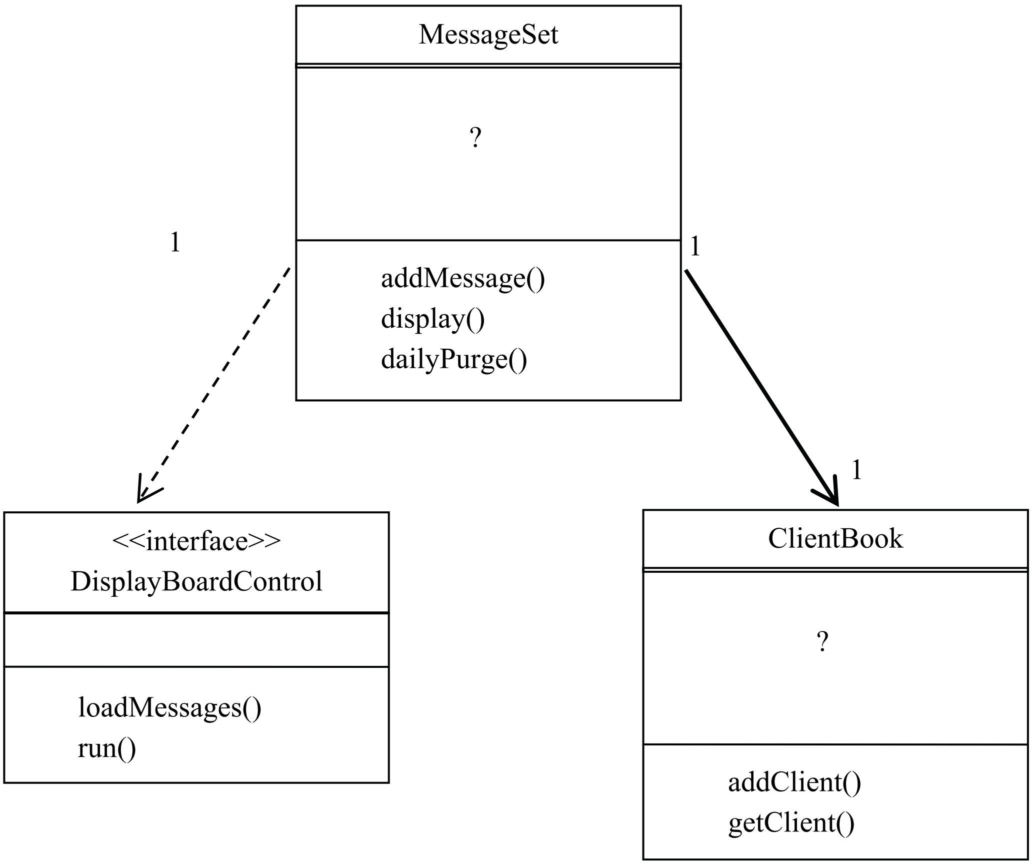
Although MessageSet appears to have no attributes, its one-to-many association with Message again implies an attribute which is a Collection type. The specification states that messages must be unique but does not imply a key value is required thus a simple set will suffice.



Relating the Classes: MessageSet, ClientBook, and DisplayBoardControl

Because MessageSet is responsible for initiating the display of the messages on the display board it has a dependency on a class implementing the DisplayBoardControl interface.

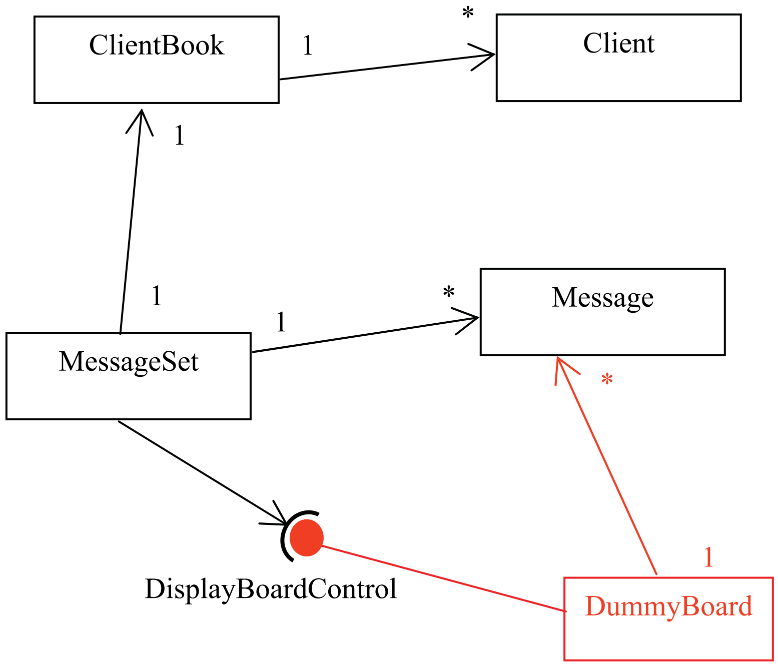
MessageSet also has a relationship with ClientBook because it needs to access and update Client information when the daily purge is carried out. This is shown below.



Relating the Classes Overall

The diagram below shows how all of these classes are related. An additional class, DummyBoard, has been included which will implement the DisplayBoardControl interface for testing purposes.

Since DummyBoard will have a collection of messages loaded it also has a one-to-many relationship with Message.



Note the use of the concise “ball and socket” notation for the DisplayBoardControl interface.

While the classes above will form the heart of the system two additional classes will be required to drive and manage the system as a whole.

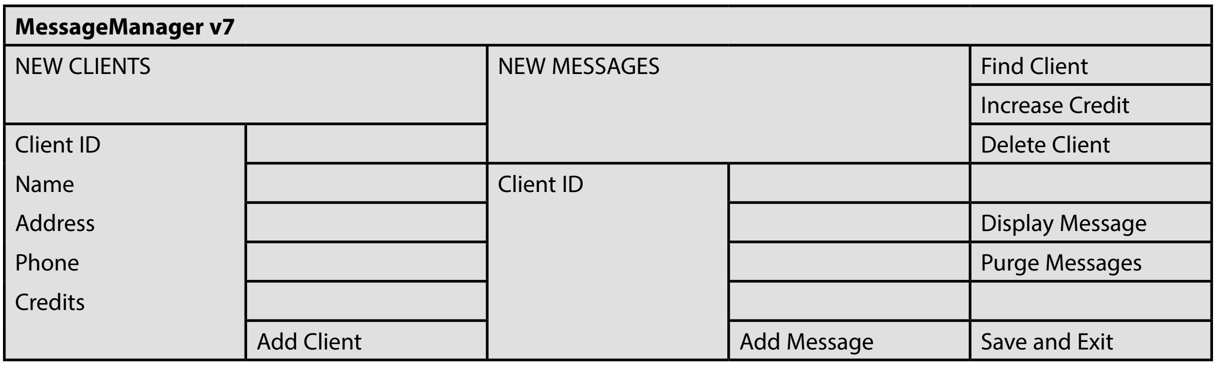
One of these 'GUImain' will have the 'main' method required to run the system and this will invoke a graphical user interface through which the user will interact with the system adding clients, messages etc.

The other additional class 'Manager' will control additional functionality not specified by the shop owner but implicitly required – for instance at the end of the day the details of the ClientBook and MessageSet will need to be saved to file. This data will need to be restored next time the system is run as the shop owner will clearly not want to enter details of all the clients every time they run the program.

11.5 Prototyping the Interface

While methods for gathering user requirements is beyond the scope of this text – it is always a good idea to prototype an interface and get feedback on this before proceeding with the development.

The figure below shows the proposed interface for this system:-



MessageManager v7				
NEW CLIENTS		NEW MESSAGES		Find Client
				Increase Credit
Client ID				Delete Client
Name		Client ID		
Address				Display Message
Phone				Purge Messages
Credits				
	Add Client		Add Message	Save and Exit

This is made up of three areas. From left to right these are a) an area for adding new clients, b) and area for adding new messages and c) an area for buttons dedicated to other essential operations.

Each of these three areas will be implemented using a JPanel placed within one larger JFrame.

11.6 Revising the Design to Accommodate Changing Requirements

Changing software requirements are a fact of life and OO programming is intended to help software engineers make program adaptations easier, quicker, cheaper and with less risk of generating errors. The principles of inheritance, method overriding and polymorphism are essential OO features that help in this manner.

In this project when gaining feedback from the shop owner on the prototype interface they comment that they generally like the interface but that they have an additional system requirement:-

> Some messages are 'urgent messages'. These should be highlighted on the display by placing three stars before and after the message and the cost of these messages will be twice the cost of ordinary messages. Other than that urgent messages are just like ordinary messages.

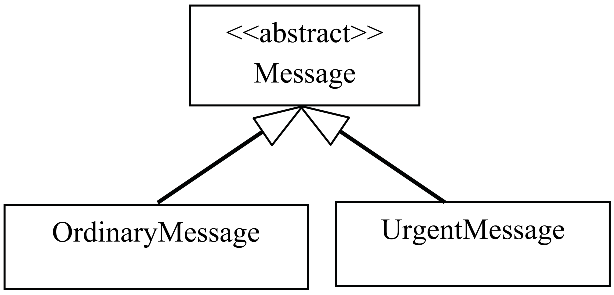
Modifying the interface design to accommodate this change is easy – we can either:-

- create an new panel to accommodate the creation of 'Urgent Messages' or
- since the data required is identical to normal messages we can just add an extra button to the middle panel.

But how will these extra requirements impact on the underlying classes within the system?

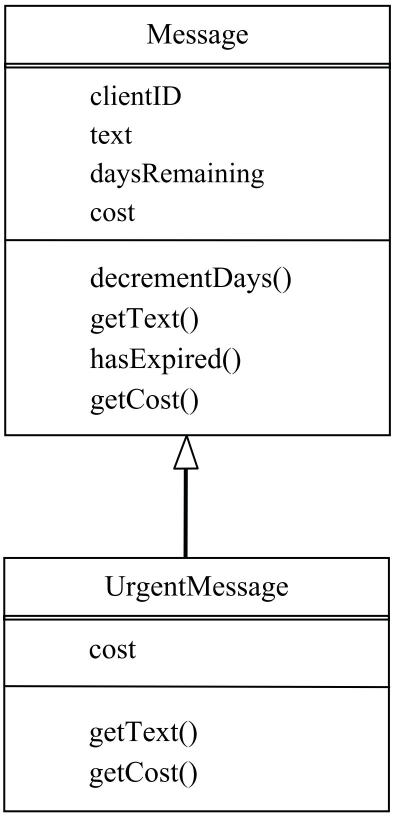
If OO principles work implementing this additional requirement should be relatively simple. Firstly there is clearly a strong relationship between a 'Message' and an 'Urgent Message'

If both classes had some unique features but there was a significant overlap in functionality we could introduce an inheritance hierarchy to deal with this:-

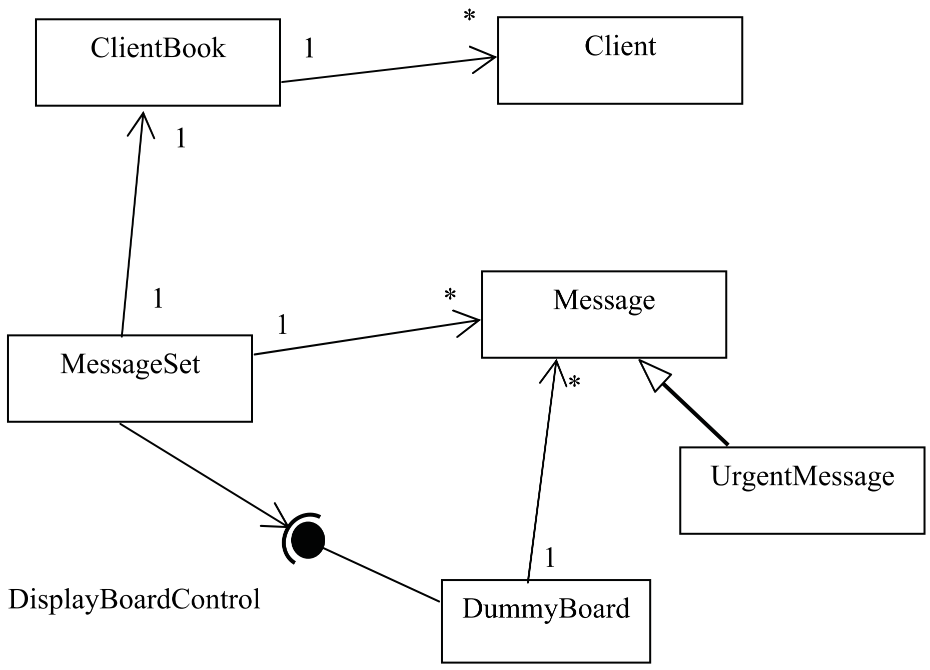


However in this case there are no unique features of an ordinary message – messages have an associated cost, the cost and text can be obtained and new messages can be created. All this is true for urgent messages. An urgent message is just the same as an ordinary message where the text and the cost has been changed slightly. Thus UrgentMessage is a type of Message and can inherit ALL of the features of Message with the cost and text methods being overridden.

Thus the Message and UrgentMessage classes are be related as shown below, with UrgentMessage inheriting all of the values and methods associated with Message but overriding getCost() and getText(0 methods to reflect the different cost and text associated with urgent messages.


Message
clientID
text
daysRemaining
cost
decrementDays()
getText()
hasExpired()
getCost()
UrgentMessage
cost
getText()
getCost()

A revised class diagram is below. But how will this change impact upon other parts of the system?



Thanks to the operation of polymorphism this change will have no impact at all on any other part of the system!

Looking at the class diagram above we can see that MessageSet keeps and manages a set of Messages (DummyBoard also keeps a set of messages – once they have been uploaded for display). But what about UrgentMessages?

Urgent messages are just a specific type of message. When the addMessage() method is invoked within MessageSet it requires an object of type Message i.e. a message to be added – but an object of the subtype UrgentMessage **is still a 'Message'** so the addMessage() method would accept an UrgentMessage object.

Therefore, without making any changes at all to MessageSet, MessageSet can maintain a set of all messages to be displayed (both urgent and ordinary)!

Furthermore when the dailyPurge() method is invoked it invokes the getCost() method on a Message object so that the client can be charged for that message. At run time the JVM will determine whether the object is of type Message or of type UrgentMessage and it will invoke the correct version of the getCost() method – remember this was overridden in UrgentMessage. This is polymorphism in action!

MessageSet requires messages but, thanks to the application of polymorphism and method overriding, MessageSet will happily deal with any Message subtype as though it were a Message object. If later we decided to create new message types (such as a Christmas message) MessageSet would be able to deal with these as well without changing a single line of code!

Thus in this application we are able to extend the system to add the facility for urgent messages by adding only one class and making one small change to the interface.

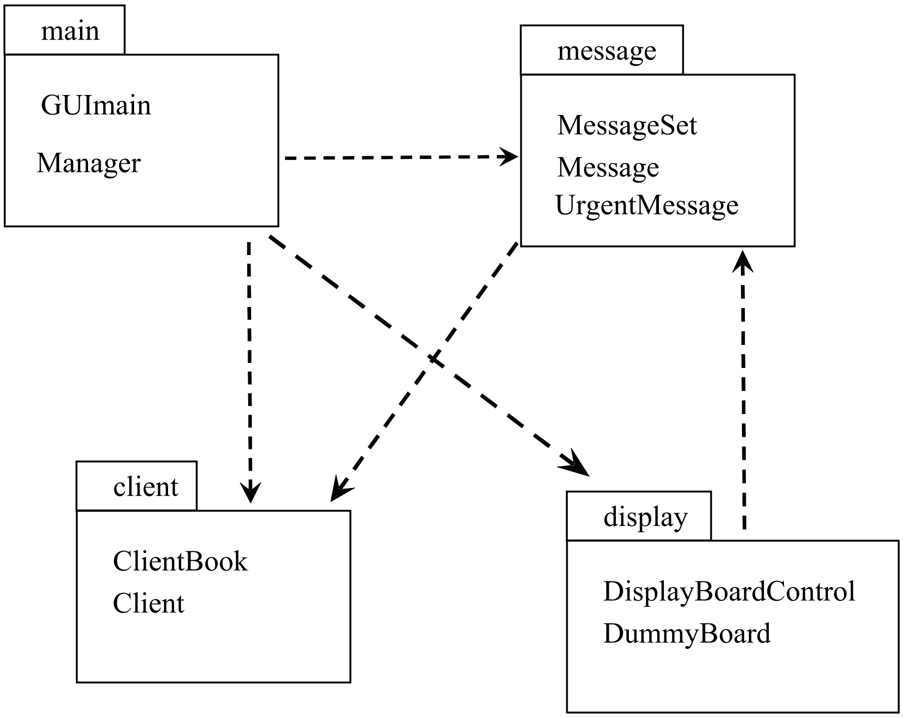
Without the application of polymorphism we would need to have made additional changes to other parts of the system – namely MessageSet and DummyBoard.

Object Orientation has enabled to the system to be extended with minimal effort!

11.7 Packaging the Classes

Large programs should be segmented into packages as this provides an appropriate level of encapsulation and access control (as described in Chapter 2).

The system being used here to demonstrate the theory in this textbook hardly qualifies as large – nonetheless it has been decided to package related classes together as shown below.



This diagram shows the four packages used and the classes within each package. Also shown are associations between the packages. Not surprisingly the main package, which houses the system interface, is associated with all of the other packages – this is because the interface invokes functionality throughout the system.

Having completed the design, and accommodated changing requirements, we can start implementing the system. This will be done in two phases:-

In the first phase a basic system will be implemented which will allow messages and clients to be created, the details written to file and messages to be displayed.

In the second phase the system functionality will be extended to allow clients to be deleted and to allow their credit to be increased. This will be done in a way to allow the demonstration of Test Driven Development (as described in Chapter 10).

11.8 Programming the Message Classes

Message, UrgentMessage and MessageSet are relatively straight forward to program.

Message has various instance variables (Strings: clientID, text and int: daysRemaining). It has a constructor to initialize the instance variables and it has the following methods:-

```
void decrementDays ()
boolean hasExpired()
String getClientID()
int getCost()
String getText ()
```

The Manager class will need to store the ClientBook and MessageSet objects to a file. To do this all Client objects and Message objects will also need to be stored hence these classes (including the Message class) will need to implement the Serializable interface.

Finally the requirements state that "No duplicate messages (i.e. the same text for the same client) are permitted."

Therefore Message must override the equals() and hashCode() methods to ensure that duplicates will not be permitted when the messages are stored in a Set.

The complete code for this class is given below – though comments have been excluded for the sake of brevity.

```
package messages;
import java.io.Serializable;

public class Message implements Serializable
{
      final int COST=1;

      private String clientID;
      private String messageText;
      private int daysRemaining;

      public Message (String pClientID, String pText,
                                          int pDaysRemaining) {
            clientID = pClientID;
            messageText = pText;
            daysRemaining = pDaysRemaining;
      }

      public void decrementDays() {
            daysRemaining--;
      }

      public boolean hasExpired() {
            return (daysRemaining == 0);
      }

      public String getClientID() {
            return clientID;
      }

      public String getText () {
            return messageText;
      }

      public int getCost() {
      return COST;
      }

      public int hashCode () {
            return (clientID + messageText).hashCode();
      }

      public boolean equals (Object pOther) {
            Message otherMsg = (Message)pOther;
            return (clientID.equals(otherMsg.clientID) &&
            messageText.equals(otherMsg.messageText));
      }

      public String toString(){
            return ("Message text: " + messageText +
                  "\nClient: " + clientID +
                  "\nDays left: " + daysRemaining);
      }
}
```

The UrgentMessage class is extremely short and sweet as it inherits almost all of its functionality from Message:-

```
package messages;

public class UrgentMessage extends Message
{

      final int COST=2;

      public UrgentMessage (String pClientID, String pText,
                                          int pDaysRemaining){
            super(pClientID, pText, pDaysRemaining);
      }

      public String getText () {
            return "*** "+super.getText()+ " ***";
      }

      public int getCost() {
            return COST;
      }
}
```

The MessageSet class has a one-to-many relationship with Message. This implies a collection type and the fact that duplicate massages are not allowed (at least for the same client) implies the collection should be a Set.

The MessageSet class requires an instance variable to hold the set of messages and also one to reference a ClientBook – as it needs access to the clients when performing a daily purge.

A constructor is required to assign a new HashSet() to messageSet and to initialize clientBook to a parameter. The following methods are also required:-

void addMessage(Message pMsgToAdd)
void display(DisplayBoardControl db)
void dailyPurge()

Some of the code from this class is shown below:-

```
package messages;

import ...

public class MessageSet implements Serializable
{

      private Set<Message> messageSet;
      private ClientBook clients;

      public MessageSet (ClientBook pClients){
            clients = pClients;
            messageSet = new HashSet<Message>();
      }

      public void addMessage(Message pMsgToAdd) {
            messageSet.add(pMsgToAdd);
      }

      public void display(DisplayBoardControl db)
      {
            db.loadMessages(messageSet);
            db.run();
      }

      public void dailyPurge() {

            // code omitted here

      }

      public void saveToFile(ObjectOutputStream oos) {
            try {
                  oos.writeObject(this);

            } catch (IOException ioe) {
                  JOptionPane.showMessageDialog(null, ""+ioe);
            }
      }

      static public MessageSet readFromFile(ObjectInputStream ois) {
            MessageSet cb = null;

            try {
                  cb = (MessageSet) ois.readObject();

            } catch (IOException ioe) {
                  JOptionPane.showMessageDialog(null, ""+ioe);
                  System.exit(1);
            } catch (ClassNotFoundException cnfe) {
                  JOptionPane.showMessageDialog(null, ""+cnfe);
                  System.exit(1);
            }
            return cb;
      }
}
```

The code above shows the creation of a typed collection of 'Message' and methods to add and display messages.

The method to display messages requires and object of type DisplayBoardControl to be passed as a parameter. Initially a DummyBoard object will be provided however when a real display board is purchased then this object will replace the DummyBoard object. This will have no impact on the code within the display() method as both objects are of the more general type DisplayBoardControl. This is another example of the application of polymorphism.

Two additional methods have been created saveToFile() and readFromFile() which read and write the entire set of messages to file using the technique of object serialisation.

The dailyPurge() method was excluded from the code above so we could concentrate on this method below:-

```
public void dailyPurge() {

      Client client;

      // loop through all current messages

      for (Message msg: messageSet) {

            // deduct 1 from days remaining for message
            msg.decrementDays();

            try
            {
                  // decrease client credit for this message
                  client = clients.getClient(msg.getClientID());
                  client.decreaseCredit(msg.getCost());

                  // if message expired or client out of credit remove it
                  if (msg.hasExpired() || client.getCredit() <= 0) {
                        messageSet.remove(msg);
                  }
            }
            catch (UnknownClientException uce) {
                  JOptionPane.showMessageDialog(null,
                  "INTERNAL ERROR IN MessageSet.Purge()\n" +
                  "Exception details: " + uce +
                  "\nMesssage details:\n" + msg);

                  if (msg.hasExpired()) {
                        messageSet.remove(msg);
                  }

            }
      }
}
```

The dailyPurge() method performs the following actions:-

```
For each message

        Decrement the days remaining for that message
        Find the client who paid for that message
        Find the cost of the message and deduct this from that clients credit
        If the message has expired or if the client has run out of credit then
              Remove the message
```

Note it is possible that a client could not be found – hence the try catch block. This will be discussed in the next section.

11.9 Programming the Client Classes

Programming the Client class is very similar to programming the Message class and is not shown here.

Programming the ClientBook class is also very similar to programming MessageSet, and most of this class is omitted here however there are two significant differences:-

- All clients have a clientID so ClientBook uses a Map instead of a Set.
- The method getClient() could fail if no client exists with the specified clientID. We need to build in protection in case a client cannot be found.

```
package clients;

import ...

public class ClientBook implements Serializable {

      private Map<String,Client> clientMap;

      public ClientBook() {
            clientMap = new HashMap<String,Client>();
      }

      public void addClient(String pClientID, Client pNewClient) {
            clientMap.put(pClientID, pNewClient);
      }

      public Client getClient(String pClientID)
            // details omitted

      }

      public void saveToFile(ObjectOutputStream oos) {
            // details omitted
      }

      static public ClientBook readFromFile(ObjectInputStream ois) {
            // details omitted
      }
}
```

The code above shows the creation of the Map and the other methods required by the ClientBook class.

11.10 Creating and Handling UnknownClientException

The getClient() method in the ClientBook class will return a null value if no client exists with the specified ID. In such a case during the daily purge our program will crash as we try to invoke the decreaseCredit() method without having a client object to invoke this method on (it will crash with a NullPointerException).

We therefore need to build in protection against this eventuality. To protect against this we need to:-

- Create a new kind of exception (as described in Chapter 9) called UnknownClientException
- tell the ClientBook class to throw this exception if a client is not found
- catch and deal with this exception in the dailyPurge() method.

The first step is simple and not shown here.

Telling the getClient() method and deleteClient() method to generate this exception is relatively straight forward (as shown below):-

```
public Client getClient(String pClientID)
                                    throws UnknownClientException {
           Client foundClient;

           foundClient = clientMap.get(pClientID);

           if (foundClient != null) {
                return foundClient;
           } else {
                throw new UnknownClientException(
                      "ClientBook.getClient():
                     unknown client ID:" + pClientID);
           }
      }
```

Firstly we must tell the compiler that this method can generate an exception. Then, under the appropriate condition, we invoke the constructor of the exception using the keyword 'new' and pass a string message required by the constructor. The object returned by the constructor is then 'thrown'.

To be helpful the string specifies the method where this exception was generated from and the clientID for which a client was not found.

The compiler will then ensure that the programmer writing the dailyPurge() method catches this exception – hopefully they will then deal with it to prevent a crash situation.

The final step is to catch and deal with UnkownClientException within the dailyPurge() method – as shown in section 11.8 (Programming the Message Classes).

If a client does not exist we may could remove the message. However in this case we have chosen to be more cautious since we simply don't know how we have come to have an 'unowned' message.

We have therefore decided that if the message has not expired we will not to take any action other than to report the error. The message will continue to be displayed (even without having a client to charge!).

If an unowned message has expired we of course still need to remove it from the display set.

11.11 Programming the Main classes

There are two classes, not shown on the class diagrams given previously, that 'drive' the system.

'Manager' is the main class that manages the system. It performs the following functions:-

- it has a permanent reference to the client book and message set.
- it sets up the data file
- it defines what happens when the system starts and
- it defines what happens when the system shuts down
- finally it has getClientBook() and getMessageSet() methods so the other parts of the system can access the clientbook and message set.

The startup() method is shown below:-

```
public void startUp() {
      try {
            FileInputStream fis = new FileInputStream(MMS_DATA_FILE);
            ObjectInputStream ois = new ObjectInputStream(fis);

            cb = ClientBook.readFromFile(ois);
            ms = MessageSet.readFromFile(ois);
            fis.close();

      } catch (FileNotFoundException fnfe) {
            JOptionPane.showMessageDialog(null, "No existing
                                    client/message data found");
            cb = new ClientBook();
            ms = new MessageSet(cb);
      } catch (IOException ioe) {
            JOptionPane.showMessageDialog(null, ""+ioe);
            System.exit(1);
      }
}
```

The startup() method tries to setup an Object Input stream from which it then tries to reconstruct ClientBook and MessageSet objects.

It has two catch blocks. The first will catch a FileNotFoundException – this will occur if the file of data cannot be found e.g. the first time this program is run. In such a case the system will create a new and empty ClientBook object and a new MessageSet object.

The second catch block will catch any other IO error and in this case will then exit the program.

11.12 Programming the Interface

The other driving class is GUImain. This has a main() method which invokes the createGUI() method. This creates three JPanels and adds buttons, text boxes and labels to these according to the preliminary design. A Grid layout manager was applied to these JPanels (as described in section 10 of chapter 8).

The GUImain class also defines action listeners for each of the buttons on the GUI.

Shown below is the action listener associated with the FindClient button:

```
private class FindClientListener implements ActionListener {
      public void actionPerformed(ActionEvent arg0) {
            String id = JOptionPane.showInputDialog(null,
                                                "Enter client ID");
            if (id == null) id = ""; //in case Cancel pressed

            try {
                  JOptionPane.showMessageDialog(null,
                  manager.getClientBook().getClient(id).toString());
            }
            catch (UnknownClientException uke){
                  JOptionPane.showMessageDialog(null,
                                                "No such client");
            }
      }
}
```

This action listener does the following:-

- It opens a dialog box to ask the user for a clients ID,
- If cancel is pressed it replaces the null returned with an ID of “”.
- It then asks the manager object to return the client book.
- On the client book object it invokes the getClient() method passing the ID as a parameter
- Assuming a client is returned the toString() method is then invoked to get a string representation of the client and this is passed as a parameter to the showMessageDialog() method (which displays the details of the client with that ID).
- If getClient() fails to find a client this method will throw an UnknownclientException – this will be caught here and an appropriate message will be displayed.

11.13 Using Test Driven Development and Extending the System

Now we have a working system – though two important methods have yet to be created. We need a method to increase a clients credit – this should be placed within the Client class. We also need a method to delete a client – as this means removing them from the client book this should be placed in the ClientBook class.

It has been decided to use Test Driven Development to extend the system by providing this functionality (as discussed in Chapter 10 Agile Programming).

In TDD we must:-

1) Write tests
2) Set up automated unit testing, which fails because the classes haven’t yet been written!
3) Write the classes so the tests pass

After creating these methods we must then adapt the interface so that this will invoke these methods.

Two test cases are given below:-

```
public void testIncreaseCredit() {
      Client c = new Client("Simon", "Room 217", "x2756", 10);
      c.increaseCredit(10);
      assertEquals(20, c.getCredit());
}

public void testDeleteClient() {
      ClientBook cb = new ClientBook();
      Client c = new Client("Simon", "Room 217", "x2756", 10);
      cb.addClient("sk", c);
      try {
            cb.deleteClient("sk");
      } catch (UnknownClientException uce) {
            fail();
      }
      try {
            c = cb.getClient("sk");
            fail();
      } catch (UnknownClientException uce) {
}
}
```

The first of these creates a client with 10 units of credit, adds an additional 10 units of credit and then checks that this client has 20 units of credit.

One test does alone not sufficiently prove that the increaseCredit() method will always work so we made need to define additional tests.

The second test creates and empty client book, creates a client, adds the client to the client book and tries to delete this client – if at this point an unknownClientexception is generated then we know there is a problem. Once deleted we try to delete the object for a second time and this should fail as an unkownClientException should now be generated. If an exception is not generated at this point then the class is flawed and the test should 'fail'.

Having created test cases these will generate complier errors as the methods increaseCredit() and deleteClient() do not exist.

We must now create the methods and revise them until these tests pass.

The increaseCredit() method is given below…

```
public void increaseCredit(int pExtraCredit) {
      credit = credit + pExtraCredit;
}
```

Theory suggest that TDD leads to simple code.

In this case by focusing our minds on what the increaseCredit() method needs to achieve we reduce the risk of over complicating the code. Of course we may need a range of test case to make sure the method has all of the essential functionality it needs.

11.14 Generating Javadoc

Documentation is essential and can be generated automatically (as described in Chapter 8 – Java Development Tools) assuming appropriate comments have been placed in the code.

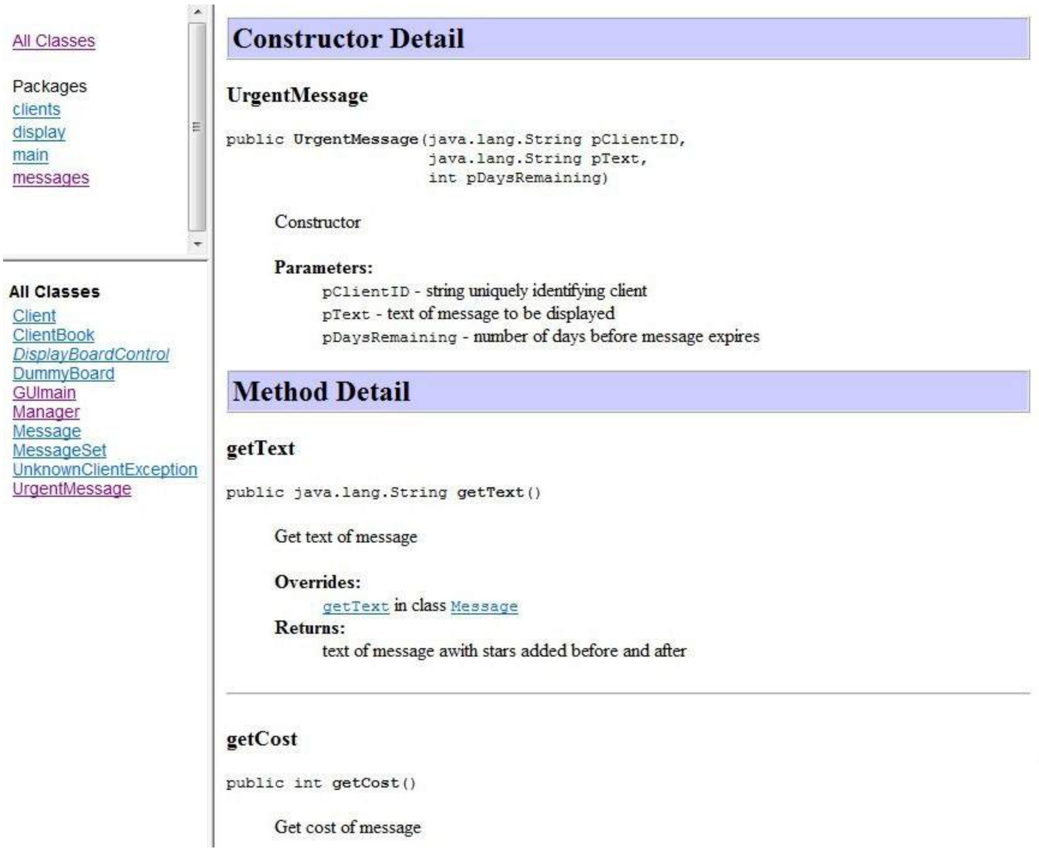
Javadoc comments have been placed in the code to describe all classes, all constructors and all methods. All parameters and return values have been described.

Three of the comments taken from the Client class are shown below:-

```
/************************************************************
 * A customer of the message display service
 *
 * @author Simon Kendal
 * @version 1.0 (26 June 2009)
 ***********************************************************/
public class Client implements Serializable {

      ... lines missing ...

      /**
       * Constructor
       *
       * @param pName name of client
       * @param pAddress client's address
       * @param pPhone client's phone number
       * @param pCredit initial credit for client
       */
      public Client(String pName, String pAddress, String pPhone,
                                                     int pCredit) {

      ... lines missing ...

      /**
       * return the client's current credit
       *
       * @return credit units remaining
       */
      public int getCredit() {

      ... lines missing ...

      }
```

Once Javadoc style comments have been placed throughout the code initiating the javadoc tool will generate a set of web pages to describe the system (in Eclipse this is done by selecting Project | Generate Javadoc menu items).

The following picture shows part of the Java documentation describing the UrgentMessage class:-



All Classes

Packages
clients
display
main
messages

All Classes
Client
ClientBook
DisplayBoardControl
DummyBoard
GUImain
Manager
Message
MessageSet
UnknownClientException
UrgentMessage

Constructor Detail

UrgentMessage

```
public UrgentMessage(java.lang.String pClientID,
                     java.lang.String pText,
                     int pDaysRemaining)
```

Constructor

Parameters:
`pClientID` - string uniquely identifying client
`pText` - text of message to be displayed
`pDaysRemaining` - number of days before message expires

Method Detail

getText

```
public java.lang.String getText()
```

Get text of message

Overrides:
`getText` in class `Message`
Returns:
text of message awith stars added before and after

getCost

```
public int getCost()
```

Get cost of message

11.15 Running the System and Potential Compiler Warnings

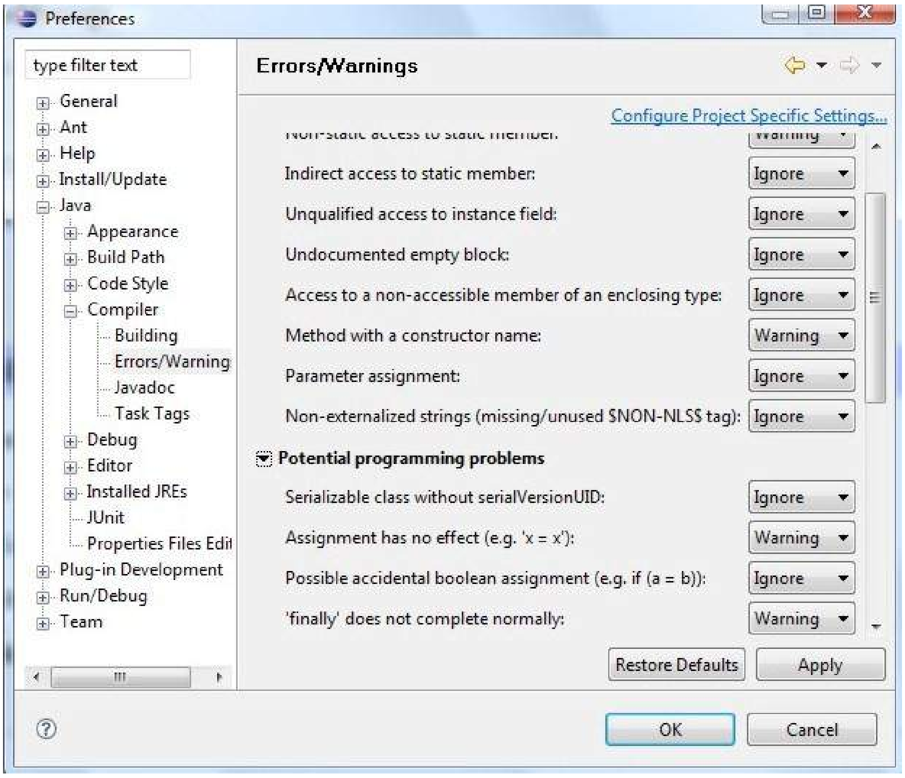
The complete program, as described in this chapter, is available with this textbook as an exported Eclipse project. To run this program:-

- Download the file 'OOP Using Java'
- Import it into Eclipse by selecting File | Import | Existing Project Into Workspace | Select Archive File 'OOP Using Java' and then select the Message Management System project that is inside the archive file.
- Using the Eclipse package explorer window, right click on the GUImain class which is inside the package 'main' and select 'Run as Java application'.

In this exported file are all classes, methods and test cases discussed in this chapter along with the Javadoc generated by the Javadoc tool. To view the Javadoc go to the .doc folder and double click on the index.html page.

When examining this program, depending upon your compiler settings, you may notice some warning messages. One in particular that you are likely to see refers to the serializable classes (MessageSet, Message etc) as I have not given them version numbers.

These warnings should have no impact on running the system and can be switched of if required by going to Window | Preferences |Compiler | Errors & Warnings and selecting ignore for those warnings you wish to suppress (as shown below).



Errors/Warnings

Non-static access to static member: Warning

Indirect access to static member: Ignore

Unqualified access to instance field: Ignore

Undocumented empty block: Ignore

Access to a non-accessible member of an enclosing type: Ignore

Method with a constructor name: Warning

Parameter assignment: Ignore

Non-externalized strings (missing/unused $NON-NLS$ tag): Ignore

Potential programming problems

Serializable class without serialVersionUID: Ignore

Assignment has no effect (e.g. 'x = x'): Warning

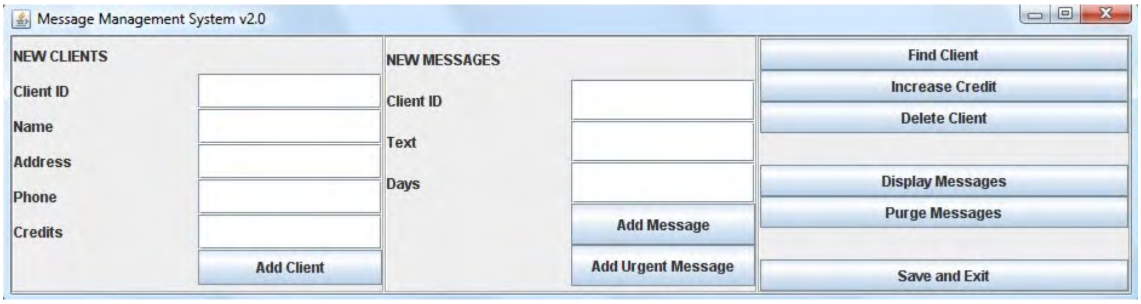
Possible accidental boolean assignment (e.g. if (a = b)): Ignore

'finally' does not complete normally: Warning

11.16 The Finished System…

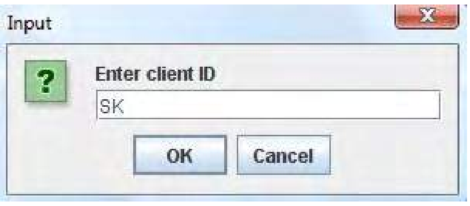
The following screen shots show the finished system.

Firstly the main interface window – this is very similar to the design. The only change was one extra button that was added to allow a message to be designated as an urgent message.

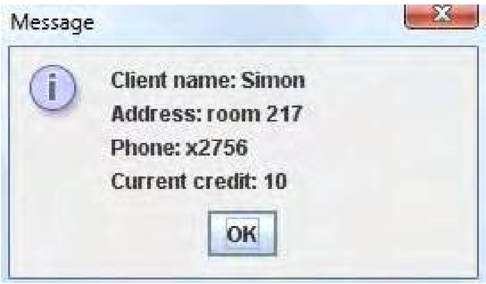


The next two images show the pop up dialogues that appear when the 'Find Client' button is pressed.

Firstly asking for a client ID…



Secondly displaying the client details – assuming a client with this ID has been added.



Client name: Simon
Address: room 217
Phone: x2756
Current credit: 10

The 'Display Messages' button shows each of the messages on the screen that should be display using the DummyBoard class. This is only crudely simulating a real display board and makes no effort to scroll the messages or display them in any graphically interesting way.

'Purge Messages' invokes the purgeMessages() method. It does nothing visible but decrements the days remaining for each message, decreases the clients credits and deletes the messages if appropriate. This can be tested by running Find Client before and after doing a daily purge.

11.17 Summary

The fundamental principles of the Object Orientated development paradigm are

- abstraction
- encapsulation
- generalization/specialization (inheritance)
- polymorphism

These principles are ubiquitous throughout the Java language and library package APIs as well as providing a framework for our own software development projects.

A well-established range of tools and reference support is available for OO development in Java, some of it allied to modern 'agile' development approaches.

Throughout this chapter you will hopefully have seen how Object Orientation supports the programmer by:-

- using abstraction and encapsulation to enables us to focus on and program different parts of a complex system without worrying about 'the whole'.
- using inheritance to 'factor out' common code
- using polymorphism to make programs easier to change
- using tools help document and manage large software projects.

This has been exemplified using Java but the same principles and benefits apply to all OO programming languages and the tools demonstrated here are available in many modern IDE's.

Through reading this book, and doing the small exercises, you will hopefully have gained some understanding of these principles.

I hope you have found this book helpful and I wish you all the best for the future.