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)
- several example of UML diagrams (as described in Chapter 2)
- an example of the use of inheritance and method overriding (as described in Chapter 3)
- an example of polymorphism and see how this enables programs to be extended simply (as described in Chapter 4)
- an example of how generic collections can be effectively used, in particular you will see an example of the use of a set and a dictionary (as described in Chapter 7)
- an example of these collections can be stored in files very simply using the process of serialization (as described in Chapter 7)
- examples of exceptions and exception handling (chapter 9)
- examples of automated unit testing (chapter 10).
- finally you will see the use of the automatic documentation tool (as described in Chapter 8).

The complete working application, developed as described throughout this chapter, is available to download for free as compressed file. This file can be unzipped and the project loaded into Visual Studio 2010.

The express edition of Visual Studio 2010 is available for free download from www.microsoft.com/express/Downloads though this does not include support for the unit testing.

The automatic documentation created to describe the system is available as a set of web pages and can therefore be viewed by any web browser.

This chapter consists of sixteen 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 the Documentation
15) The Finished System
16) Running the System
17) Conclusions

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 number 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 and the number of days for which it should be displayed. The cost of each message is 1 unit per day. 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 c lasses, 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 number
- 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 number = 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 property or accessor method!).

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 C# '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 :-

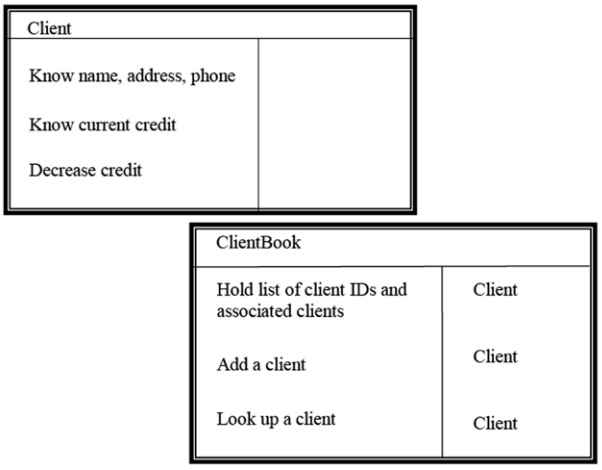
- display
- add (client to book)
- add (message to set)
- lookUp (client in book)
- purge
- decrement
- expire
- delete

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.



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 :-

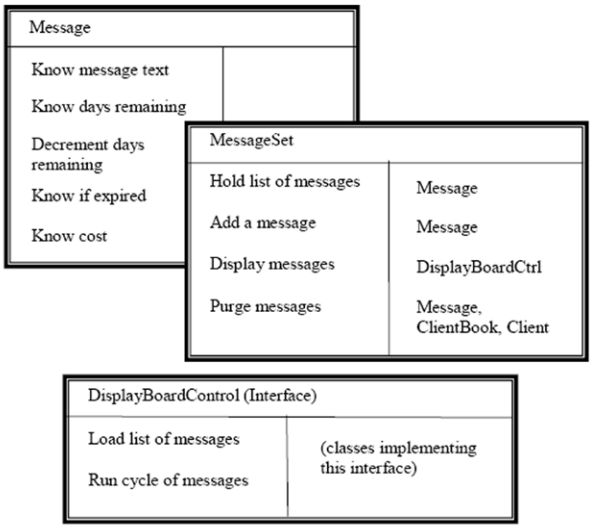
- create a new Client object
- pass it (along with the unique ID to associate with it) to the ClientBook object
- add the client to the client book.

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.



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

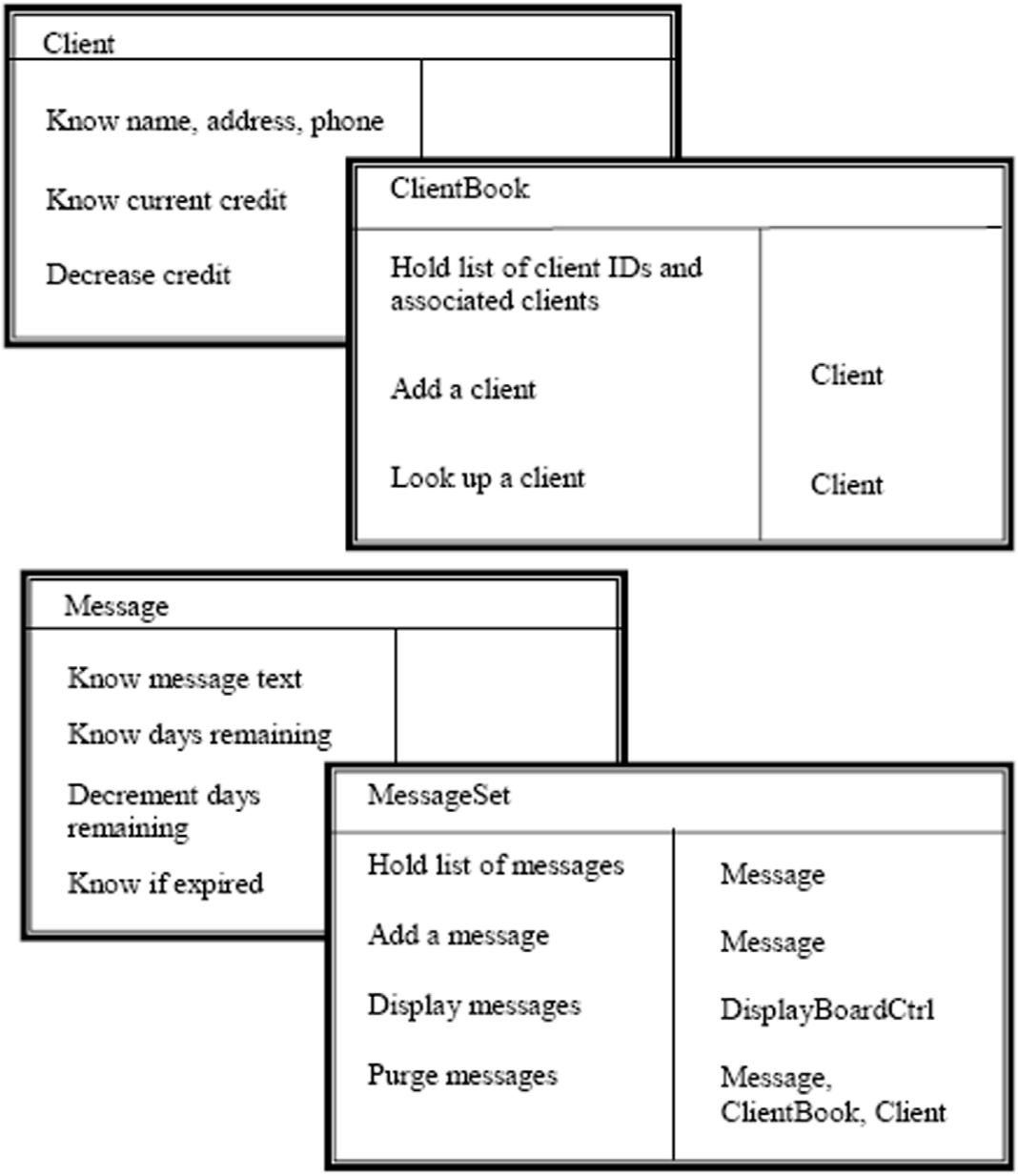
- Create a new message,
- Add this to the message set and
- Display these messages by invoking load 'list of messages' and 'run cycle of messages'

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…



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

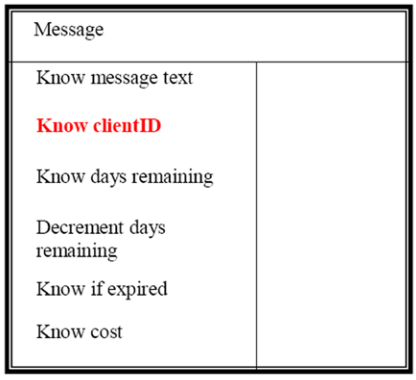
- if either the Client's credit is <= 0 or the Message is now expired
delete the message from the list

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….



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

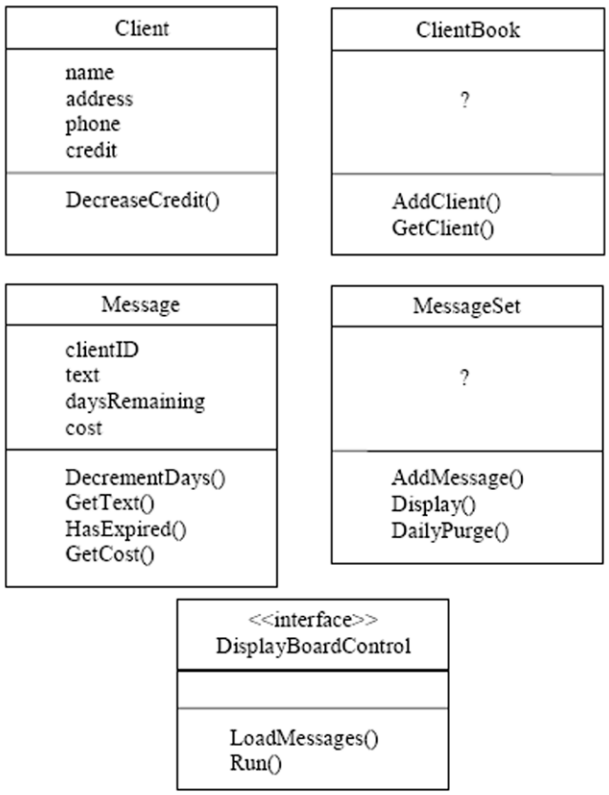
To fully document our designs we need to :-

- 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).

Of course as we are going to program this system in C# we will replace our accessor and mutator methods with properties but as this design could be programmed in any OO language we will leave our design showing appropriate accessor and mutator methods.



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).

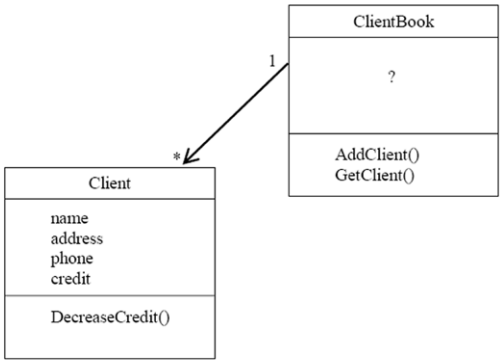
Relationships Between Classes

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.

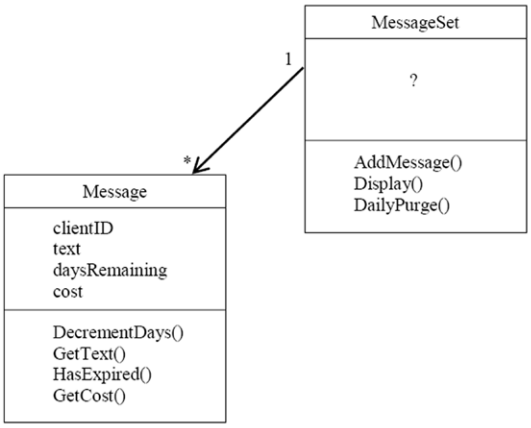
The navigability will be from ClientBook to client because the book "knows about" its Clients (in implementation terms it will have references to them) but the individual Clients will not have references back to the book.

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 dictionary where each entry is made up of a pair of values – in this case a clientID (an int) and a Client object. As we are likely to be searching and retrieving clients by their ID far more often than we will be adding and deleting clients the system will be more efficient if we make this dictionary a sorted dictionary, where the clients will be stored in order of their ID. With a sorted dictionary adding and deleting elements will be slower as the process is more complex but retrieving elements will be quicker.



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

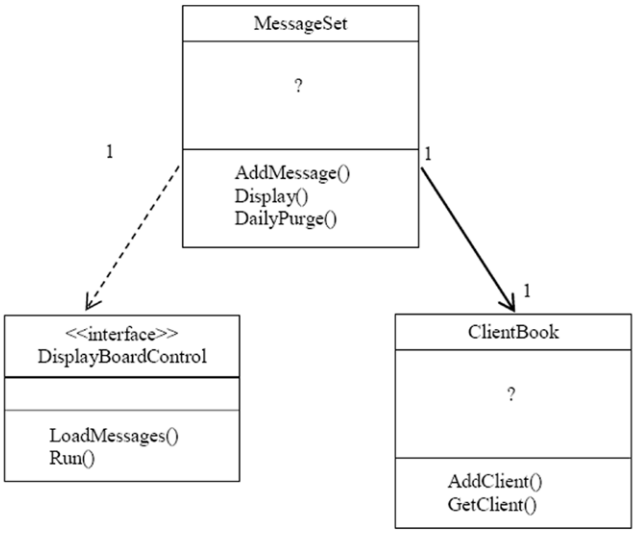
Although MessageSet appears to have no attributes, its one-to-many association with Message again implies an attribute which is a Collection type. The specification states that messages must be unique but does not imply a key value is required thus a simple set will suffice.



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.

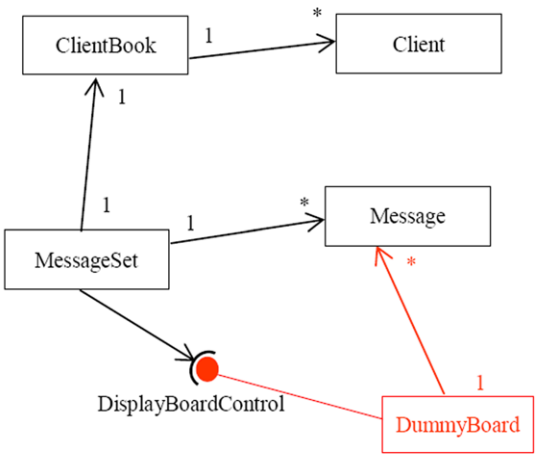
MessageSet also has a relationship with ClientBook because it needs to access and update Client information when the daily purge is carried out. This is shown below..



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 an additional class will be required to drive and manage the system as a whole.

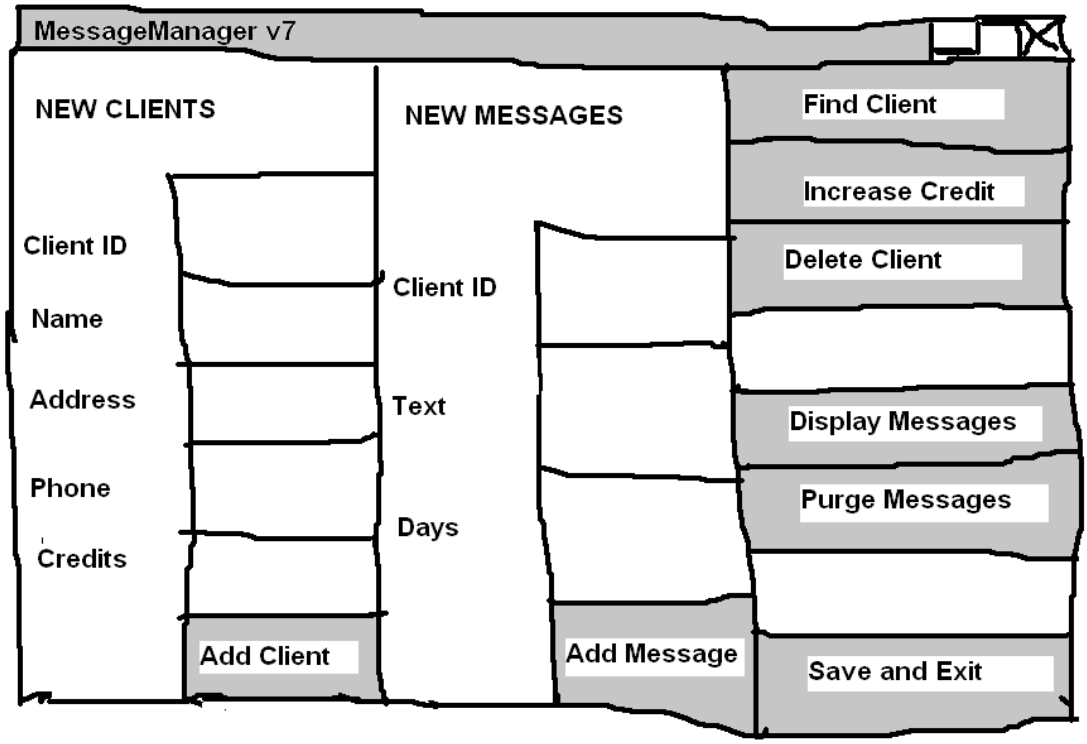
A class 'ApplicationWindow' will be created. This will be the GUI that will run the system and this will allow the user will interact with the system adding clients, messages etc.

Other additional functionality, not specified by the shop owner, is implicitly required. 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. This again will be driven from the interface but the body of the code will be devolved to the relevant classes.

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:-



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.

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.

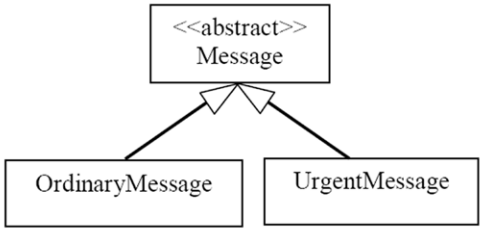
Modifying the interface design to accommodate this change is easy – we can either :-

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

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

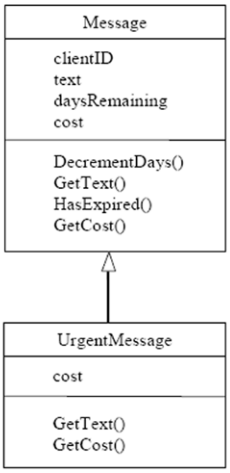
If OO principles work implementing this additional requirement should be relatively simple. Firstly there is clearly a strong relationship between a 'Message' and an 'Urgent Message'

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 :-

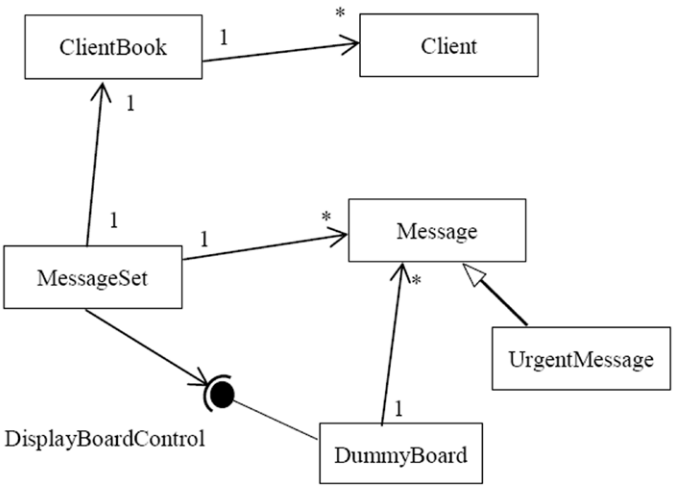


However in this case there are no unique features of an ordinary message – messages have an associated cost, the cost and text can be obtained and new messages can be created. All this is true for urgent messages. An urgent message is just the same as an ordinary message where the text and the cost have 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() methods to reflect the different cost and text associated with urgent messages.



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 Common Language Runtime (CLR) engine 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 or Valentine 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.

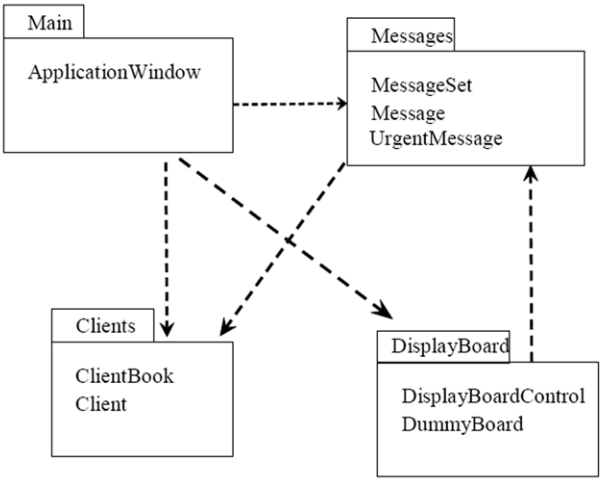
Without the application of polymorphism we would need to have made additional changes to other parts of the system – namely MessageSet and DummyBoard.

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 (String: messageText and int: COST, clientID and daysRemaining). It has appropriate properties to access these private attributes (note only daysRemaining needs a setter) and a constructor to initialize the instance variables. It also has the following methods :-

```
void DecrementDays()   // to reduce the number of days that the message should be displayed for
boolean HasExpired()   // to specify if this message should be removed from the set of messages
String ToString()      // to return the text to be displayed on the displayboard.
```

At some point we 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 be marked as Serializable.

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 GetHashCode() 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.

The source code for the full system, fully commented, can be viewed by following the instructions near the end of this chapter.

```
namespace MessageManagerSystem.Messages
{
     [Serializable]
     public class Message
     {
          const int COST = 1;
          public virtual int Cost
          {
               get { return COST; }
          }

          private int clientID;
          public int ClientID
          {
               get { return clientID; }
          }

          private String messageText;
          public String MessageText
          {
               get { return messageText; }
          }

          private int daysRemaining;
          public int DaysRemaining
          {
               get { return daysRemaining; }
               set { daysRemaining = value; }
          }

          public Message(int clientID, String text, int daysRemaining)
          {
               this.clientID = clientID;
               this.messageText = text;
               this.daysRemaining = daysRemaining;
          }

          public void DecrementDays()
          {
               daysRemaining--;
          }

          public bool HasExpired()
          {
               return (daysRemaining <= 0);
          }

          public override String ToString()
          {
               return (messageText);
          }
          public override bool Equals(object obj)
          {
               Message m = (Message)obj;
               return (clientID.Equals(m.ClientID) &&
                       messageText.Equals(m.MessageText));
          }

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

     }
}
```

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

```
namespace MessageManagerSystem.Messages
{

     [Serializable]
     public class UrgentMessage : Message
     {
          const int COST = 2;
          public override int Cost
          {
               get { return COST; }
          }


          public UrgentMessage(int clientID, String text, int
          daysRemaining):base (clientID, text, daysRemaining)
          {
          }

          public override String ToString()
          {
               return ("*** "+ MessageText + " ***");
          }

     }
}
```

Only the 'Cost' property and ToString() methods are overridden as UrgentMessages cost more and the text to be displayed changes. Note to override the Cost property we must mark it as virtual in the Message Class.

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 it will need access to a ClientBook object as it needs access to the clients when performing a daily purge. The client book object could be stored using an instance variable or passed as a parameter to the DailyPurge() method. As it is only required by this one method the decision was made to pass this as a parameter each time the DailyPurge() is invoked.

A constructor is required to assign a new HashSet() to Messages (the set of messages stored). The following methods are also required :-

void AddMessage(Message msgToAdd)	to add a message to the message set
void display(DisplayBoardControl db)	to display the messages each day on a display board… initially a simulated display board
void DailyPurge(ClientBook clients)	to a) decrement the days remaining at the end of each day for each message, b) charge the client for displaying that message and c) remove all messages that have expired.
private bool ToBeDeleted()	a private method used by the DailyPurge() to denote which messages are to be removed from the message set i.e. those that have expired.

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

```
namespace MessageManagerSystem.Messages
{
     [Serializable]
     public class MessageSet
     {
          private HashSet<Message> messages;
          public HashSet<Message> Messages
          {
               get { return messages; }
          }

          public MessageSet()
          {
               messages = new HashSet<Message>();
          }

          public void AddMessage(Message msgToAdd)
          {
               messages.Add(msgToAdd);
          }

          public void Display(IDisplayBoardControl db)
          {
               db.LoadMessages(messages);
               db.Run();
          }

          public void DailyPurge(ClientBook clients)
          {
                // code omitted here
          }

          private bool ToBeDeleted(Message m)
          {
               return m.HasExpired();
          }

     }

}
```

The code above shows the creation of a typed collection of 'Message' called Messages 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.

The DailyPurge() method was excluded from the code above so we could concentrate on this method now.

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

For each message

If the message has expired or if the client has run out of credit then
Set the DaysReamining for that message to zero.

Remove all expired messages.

See code below for this…

```
public void DailyPurge(ClientBook clients)
{
    Client client;

    // loop through all current messages and decrement credit and days
    foreach( Message m in messages)
    {

        m.DecrementDays();  // deduct 1 from days remaining for message
        try
        {
            // decrease client credit for this message
            client = clients.GetClient(m.ClientID);
            client.DecreaseCredit(m.Cost);
        }
        catch (UnknownClientException uce)
        {

            MessageBox.Show("INTERNAL ERROR IN MessageSet.Purge()
            \r\nException Details: " + uce.Source + "  \r\nMessage
            details " + uce.Message + ": \r\n");       }

    }
    // loop through all current messages
        and expire those whose client credit <=0
    foreach (Message m in messages)
    {
         try
         {
              client = clients.GetClient(m.ClientID);
              if (client.Credit <= 0)
              {
                   m.DaysRemaining = 0;
              }
         }
         catch (UnknownClientException)
         {
              // Do nothing as unknown clients have been reported to
                 error log already
         }

    }

    // Remove all expired messages
    messages.RemoveWhere(ToBeDeleted);
}
```

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

11.9 Programming the Client Classes

The system needs a Client class, with methods to decrease the client's credit (when a message has been displayed for them). Ultimately it will also need a method to allow a client to pay for and increase their credit but we will not add this functionality in the first version of this system. This class is simple and is very similar to programming the Message class and is therefore not shown here.

Programming the ClientBook class is also similar to programming MessageSet, class however there are a few significant differences :-

- All clients have a ClientID so ClientBook uses a sorted dictionary 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.
- The method AddClient() could also fail if a client already exists with the specified ID.

The complete ClientBook class (without comments) is shown below. By downloading the finished program this code can be viewed with embedded comments.

```
namespace MessageManagerSystem.Clients
{
     [Serializable]
     public class ClientBook
     {
          private SortedDictionary<int, Client> clients;
          public  SortedDictionary<int, Client> Clients
          {
               get { return clients; }
          }

          public ClientBook()
          {
               clients = new SortedDictionary<int, Client>();
          }

          public ClientBook(SortedDictionary<int, Client> clients)
          {
               this.clients = clients;
          }

          public void AddClient(int clientID, Client newClient)
          {
               try
               {
                    clients.Add(clientID, newClient);
               }
               catch (ArgumentException)
               {
                    throw new ClientAlreadyExistsException
                    ("ClientBook.AddClient(): a client with this ID
                      already exists in system ID:" + clientID);
               }
          }

          public Client GetClient(int clientID)
          {
               try
               {
                    return clients[clientID];
               }
               catch(KeyNotFoundException)
               {
                    throw new UnknownClientException
                    ("ClientBook.GetClient(): unknown client ID:" +
                     clientID);
               }
          }
     }
}
```

The code above shows the constructors to create a ClientBook object which is a sorted dictionary of ClientID, Client objects and the other methods required by the ClientBook class. Further discussion of this is provided below.

11.10 Creating and Handling UnknownClientException

The GetClient() method will generate a KeyNotFoundException if no client exists with the specified ID. If we do not catch and deal with this exception our program will crash! Furthermore our program will crash as we try to invoke the DecreaseCredit() method without having a client object to invoke this method on.

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 ….

```
public class UnknownClientException :ApplicationException
{
     public UnknownClientException(String message): base(message)
     {
     }
}
```

We next tell the GetClient() method to generate an UnknownClientException if it catches a KeyNotFoundException (as shown below) :-

```
public Client GetClient(int clientID)
{
      try
      {
            return clients[clientID];
       }
       catch(KeyNotFoundException)
       {
            throw new UnknownClientException("ClientBook.GetClient():
                        unknown client ID:" + clientID);
       }
}
```

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 DailyPurge() method should catch this exception and hopefully 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 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 no recognized client 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.

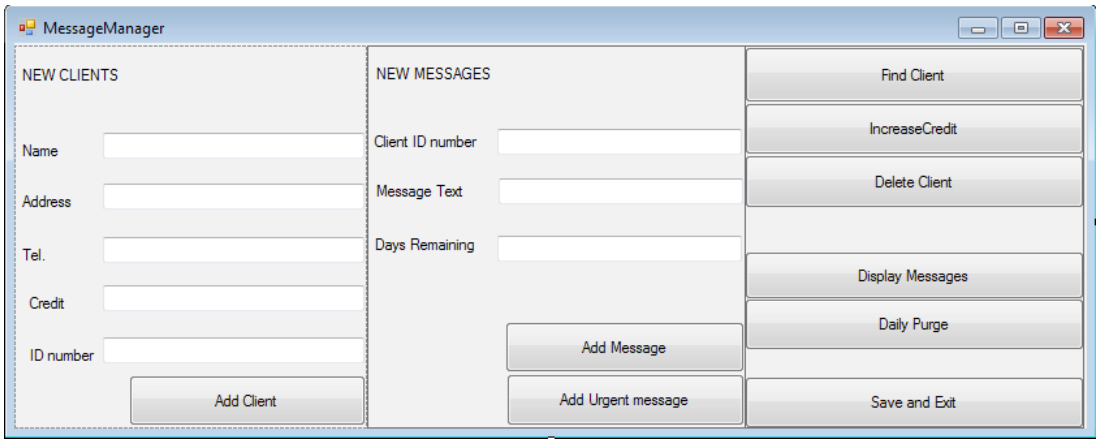
11.11 Programming the Interface

We have still to examine the main application window that will be used to 'drive' 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 and used for storing the ClientBook and Message Set
- it defines what happens when the system starts and
- it defines what happens when the system shuts down
- it invokes the constructors for the Message, UrgentMessage and Client classes whenever the user wants to add a new message or client to the system.
- it displays the messages on a dummy display board and
- it invokes the DailyPurge() method when requested by the user at the end of each day.

The application window will look as shown below….



Note this includes buttons for increasing a client's credit and deleting a client – functionality we realised some time ago that we needed but functionality that was not added to the first version of this system. We will extend our system to add this additional functionality once a basic system is working.

The ApplicationWindow_Load() method is shown below :-

```
private void ApplicationWindow_Load(object sender, EventArgs e)
{
     clientBook = new ClientBook();
     messageSet = new MessageSet();
     try
     {
         FileStream inFile = new FileStream("ClientAndMessageData",
                                            FileMode.Open, FileAccess.Read);
         BinaryFormatter bFormatter = new BinaryFormatter();
         clientBook = (ClientBook)bFormatter.Deserialize(inFile);
         messageSet = (MessageSet)bFormatter.Deserialize(inFile);
         inFile.Close();
         inFile.Dispose();
     }
     catch (FileNotFoundException)
     {
     }
}
```

The ApplicationWindow_Load() method reconstructs any previously stored ClientBook and MessageSet objects.

The action listener for the SaveAndExit button is shown below….

```
private void btnSaveAndExit_Click(object sender, EventArgs e)
{
     FileStream outFile = new FileStream("ClientAndMessageData",
                                         FileMode.Create, FileAccess.Write);
     BinaryFormatter bFormatter = new BinaryFormatter();


     bFormatter.Serialize(outFile, clientBook);
     bFormatter.Serialize(outFile, messageSet);

     outFile.Close();
     outFile.Dispose();
     this.Close();

}
```

Note how with just four lines of code above we can create an appropriate output stream and save all client book and message set data – this includes details of all clients and all messages. While we had to mark the relevant classes, ClientBook, Client, MessageSet and Message, as serializable we did not have to write any code to save this data to file.

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

```
private void btnFindClient_Click(object sender, EventArgs e)
{
    Client client;
    try
    {
        InputBox inputBox = new InputBox("Find Client", "Please enter
                                                              clients ID.");
        DialogResult dialogResult = inputBox.ShowDialog();

        if (dialogResult == DialogResult.OK)
        {
               int clientID;
               if (!Int32.TryParse(inputBox.Answer, out clientID))
               {
                    MessageBox.Show("Invalid client ID, please enter
                    integer number.");
                    return;
               }

                client = clientBook.GetClient(clientID);
                MessageBox.Show(client.ToString());
          }
     }
     catch (UnknownClientException ex)
     {
          MessageBox.Show(ex.Message);
     }
}
```

This action listener performs the following tasks:-

- It opens a dialog box to ask the user for a clients ID. As C# does not contain predefined methods to create input boxes this uses a form called InputBox that was created specifically for this purpose. If cancel is pressed this form returns an empty string.
- It then checks that OK has been pressed and the ID returned is a valid integer.
- On the client book object it invokes the GetClient() method passing the ID as a parameter.
- Assuming a client object 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 MessageBox.Show() method (which displays the details of the client with that ID).
- If GetClient() fails to find a client with the specified ID it will throw an UnknownClientException – this will be caught here and an appropriate message will be displayed. This makes use of the Message property of the Exception class.

11.12 Using Test Driven Development and Extending the System

We now have a working system – though two important methods have yet to be created. We need a method to increase a client's 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.

To do this we will create two test fixtures… one to test the ClientBook class and one to test the Client class.

In the Client text fixture we will initialise a test by setting up a specified client. We will then create a test to test credit can be added and finally we will set the TestCleanup() method to remove the client specified.

The code for this is shown below…

```
namespace MessageManagerTests
{
     TestClass]
     public class TestFixture_ClientTests
     {

          private MessageManagerSystem.Clients.Client c;

          [TestInitialize]
          public void TestInitialize()
          {
               c = new MessageManagerSystem.Clients.Client("Simon",
                    "Room 1234", "x200", 10);
          }

          [TestCleanup]
          public void TestCleanup()
          {
               c = null;
          }

          [TestMethod]
          public void IncreaseCredit_TestAdd5UnitsOfCredit
                                                                                    _
CreditShouldBe15()
          {
               c.IncreaseCredit(5);
               Assert.AreEqual(15, c.Credit, "Credit after adding 5
               units is not as expected. Expected: 15 Actual:
               "+c.Credit);
          }

    }
}
```

This test creates a client with 10 units of credit, adds an additional 5 units of credit and then checks that this client has 15 units of credit.

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

We also need to create test cases to test the DeleteClient() method in the ClientBook class. As this is a separate class we need to create a new test fixture appropriately named and we need to set up a test to test this method.

The code for this is shown below….

```
namespace MessageManagerTests
{
     [TestClass]
     public class TestFixture_ClientBookTests
     {
          public TestFixture_ClientBookTests()
          {
          }

          private MessageManagerSystem.Clients.ClientBook cb;


          [TestInitialize]
          public void TestInitialize()
          {
               cb = new MessageManagerSystem.Clients.ClientBook();

          }

          [TestCleanup]
          public void TestCleanup()
          {
               cb = null;
          }

          [TestMethod]
          public void GetClient_TestDeleteClient
                                                  _ShouldNotGenerateException()
          {
               MessageManagerSystem.Clients.Client c = new
               MessageManagerSystem.Clients.Client
                                        ("Simon", "Room 1234", "x200", 10);
               try
               {
                    cb.AddClient(1, c);
                    cb.DeleteClient(1);

               }
               catch
               (MessageManagerSystem.Clients.UnknownClientException)
               {
                    Assert.Fail("UnknownClient exception should not be
                                                 thrown if client exists");
               }
          }
```

One test we should perform on the DeleteClient() method is to test that it can delete a client … or at least not generate an exception. The test above proves and exception is not thrown inappropriately but it does not demonstrate that the client has been successfully deleted nor does it test what happens if we try to delete a client thqat does not exist… clearly we need to define some more tests.

Having created appropriate test cases our code will generate complier errors as the methods IncreaseCredit() and DeleteClient() do not exist.

We must now add these methods o our program and revise them until these tests pass.

The IncreaseCredit() method is given below…

```
public void IncreaseCredit(int extraCredit)
{
     credit = credit + extraCredit;
}
```

And the DeleteClient() method is given below…

```
public void DeleteClient(int clientID)
          {
               if(clients.Remove(clientID)==false)
               {
                    throw new
                    UnknownClientException("ClientBook.DeleteClient():
                    unknown client ID:" + clientID);
               }
          }
```

Finally we must amend the system GUI to invoke these methods as required.

Theory suggests that TDD leads to simple code.

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

Even if not developing our system using TDD we should define a wide ranging set of test cases for all of the classes within the system. This will ensure that we can undertake regression testing every time we enhance or adapt the system to meet the future and ever changing needs of the client.

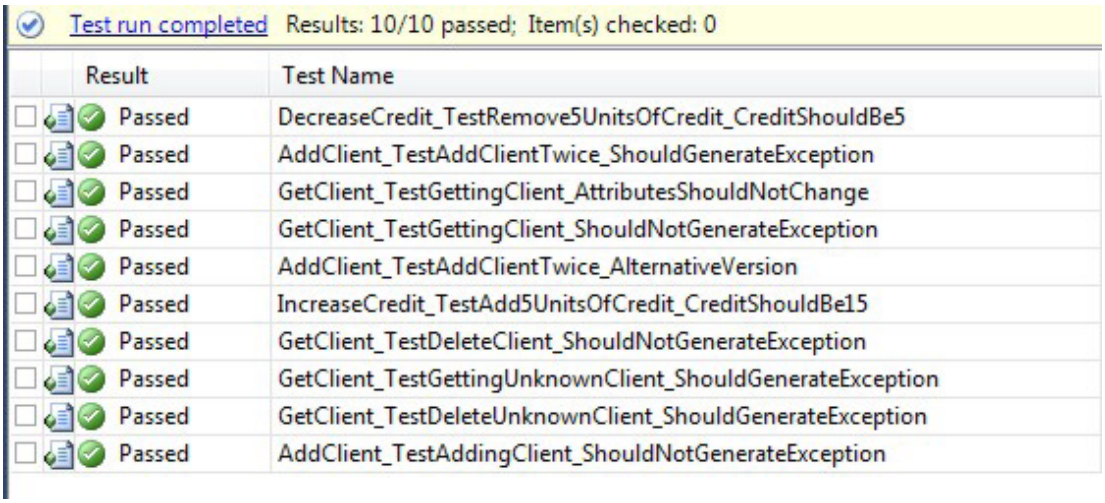
Some more tests for the ClientBook class are shown below….

```
[TestMethod]
public void AddClient_TestAddingClient_ShouldNotGenerateException()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     try
     {
          cb.AddClient(1, c);
     }
     catch
     (MessageManagerSystem.Clients.ClientAlreadyExistsException)
     {
          Assert.Fail("ClientAlreadyExists exception should not be
          thrown for new clients");
     }

}

[TestMethod]
public void AddClient_TestAddClientTwice_ShouldGenerateException()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     try
     {
          cb.AddClient(1, c);
          cb.AddClient(1, c);
          Assert.Fail("ClientAlreadyExists exception should be thrown
          if client added twice");
     }
     catch
    (MessageManagerSystem.Clients.ClientAlreadyExistsException)
     {
     }
}

[TestMethod]                              [ExpectedException(typeof(MessageManagerSystem.Clients.
ClientAlreadyExistsException))]
public void AddClient_TestClientTwice_AlternativeVersion()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     cb.AddClient(1, c);
     cb.AddClient(1, c);
     Assert.Fail("ClientAlreadyExists exception should be thrown if
     client added twice");
}

[TestMethod]
public void GetClient_TestGettingUnknownClient_ShouldGenerateException()
{
     try
     {
         cb.GetClient(1);
          Assert.Fail("UnknownClient exception should be thrown if
          client does not exist");
```

```
     }
     catch(MessageManagerSystem.Clients.UnknownClientException)
     {
     }
}

[TestMethod]
public void GetClient_TestGettingClient_ShouldNotGenerateException()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     MessageManagerSystem.Clients.Client c2 = null;
     try
     {
          cb.AddClient(1, c);
          c2=cb.GetClient(1);
     }
     catch(MessageManagerSystem.Clients.UnknownClientException)
     {
          Assert.Fail("UnknownClient exception should not be thrown if
          client exists");
     }
}

[TestMethod]
public void GetClient_TestGettingClient_AttributesShouldNotChange()
{
     MessageManagerSystem.Clients.Client c = new
     MessageManagerSystem.Clients.Client("Simon",
     "Room 1234", "x200", 10);
     MessageManagerSystem.Clients.Client c2 = null;
     try
     {
          cb.AddClient(1, c);
          c2 = cb.GetClient(1);
          Assert.AreEqual(c2.Credit,10,"Value of returned credit not as
          expected");
     }
     catch (MessageManagerSystem.Clients.UnknownClientException)
     {
     }
}


[TestMethod]
public void GetClient_TestDeleteUnknownClient_ShouldGenerateException()
{
     try
     {
          cb.DeleteClient(1);
          Assert.Fail("UnknownClient exception should be thrown if
          client does not exist");
     }
     Catch (MessageManagerSystem.Clients.UnknownClientException)
     {
     }
}
```

The tests above show numerous tests with an empty client book. They demonstrate that clients can be added, but not twice. They also demonstrate that clients can be deleted and that exceptions are generated appropriately.

The figure below shows the results from running the tests….



By creating automated test fixtures to test all classes and all methods we can run these tests every time the system is adapted to meet the clients changing needs.

11.13 Generating the Documentation

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

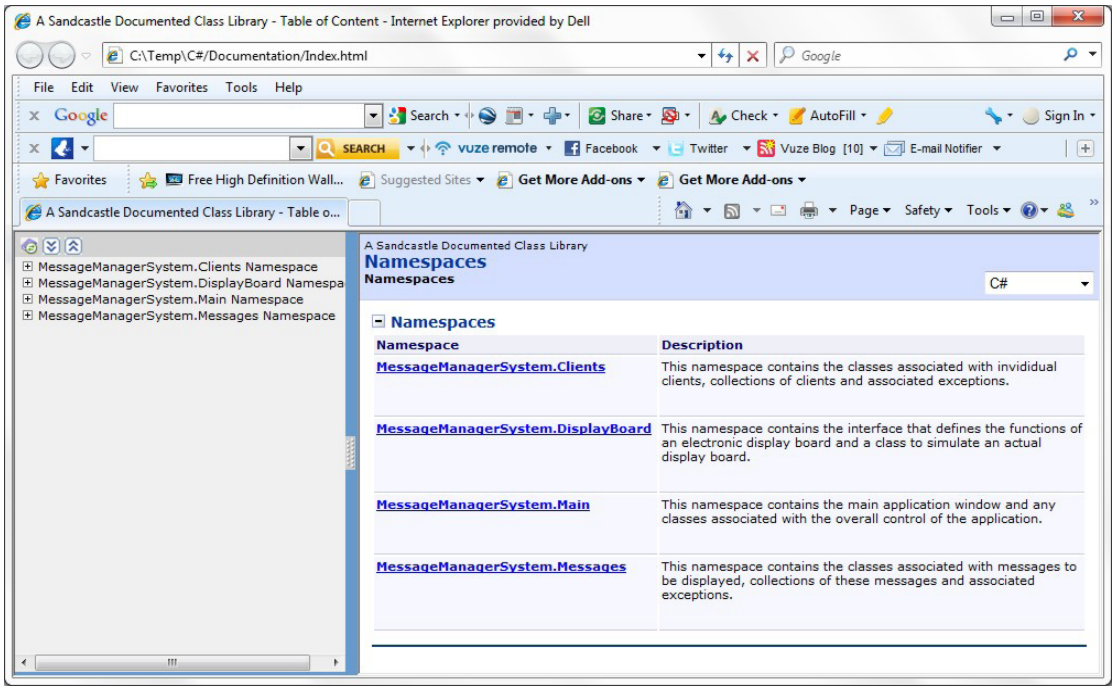
XML comments have been placed in the code to describe all classes, all constructors and all methods. All parameters, return values and exception thrown have also been described.

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

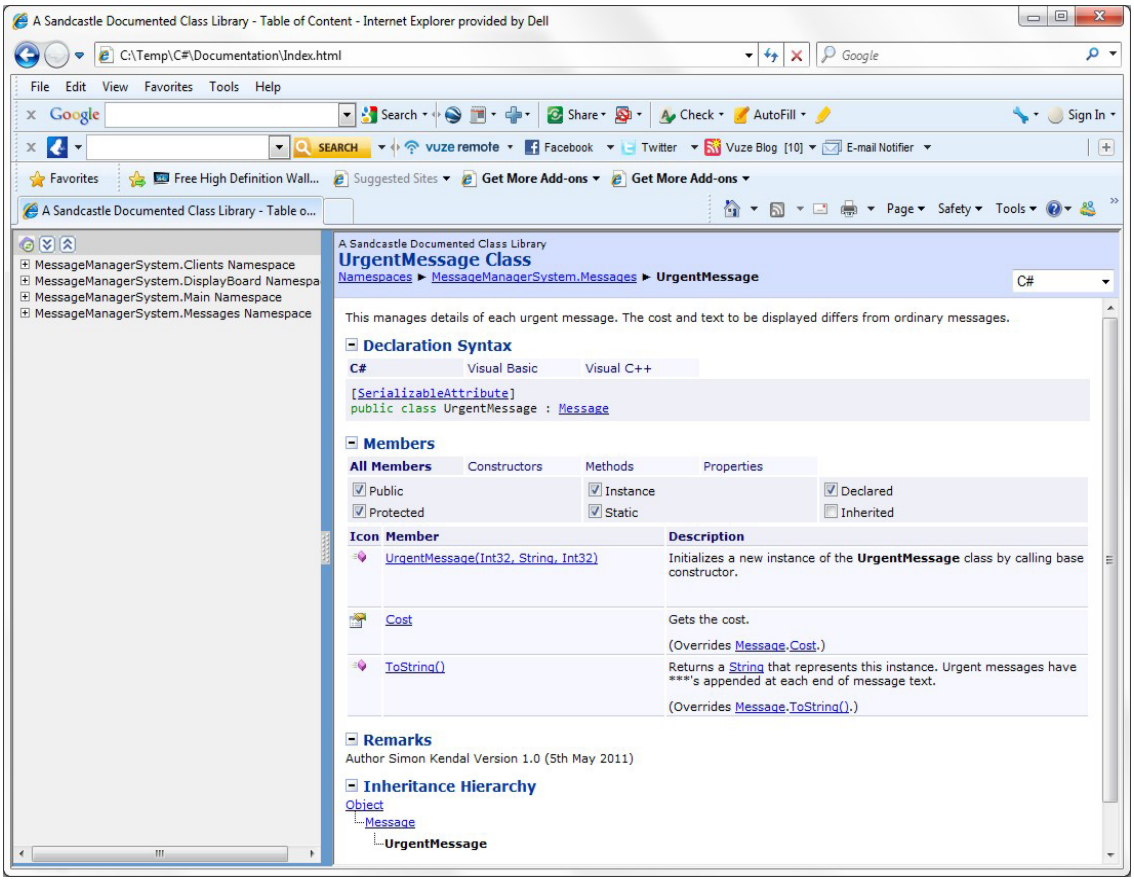
```
/// <summary>
/// Manages a collection (sorted dictionary) of clients where each
/// client has an ID number (int).
/// </summary>
/// <remarks>Author Simon Kendal
/// Version 1.0 (5th May 2011)</remarks>
public class ClientBook
{
      private SortedDictionary<int, Client> clients;

      /// <summary>
      /// Gets the clients.
      /// </summary>
      public  SortedDictionary<int, Client> Clients
      {
           // ... lines missing ...

      }

      /// <summary>
      /// Initializes a new empty instance of the <see
                                                cref="ClientBook"/> class.
      /// </summary>
      public ClientBook()
      {
              // ... lines missing ...
      }

      /// <summary>
      /// Initializes a new instance of the <see cref="ClientBook"/>
                 class and instantiates this to the disctionary passed.
      /// </summary>
      /// <param name="clients">A disctionary of Client ID, Client
                                                              objects.</param>
      public ClientBook(SortedDictionary<int, Client> clients)
      {
           // ... code omitted ...
      }

      /// <summary>
      /// Adds a client to the client book
      /// </summary>
      /// <param name="clientID">The client ID.</param>
      /// <param name="newClient">The new client.</param>
      /// <exception cref="ClientAlreadyExistsException"> Throws
        exception if a client with ClientID already exists</exception>
      public void AddClient(int clientID, Client newClient)
      {
           // ... code omitted ...

      }
```

Once XML comments have been placed throughout the code and exported, and comments have been added to the Sandcastle Help File Builder tool to describe the namespaces then this tool can be used to generate a set of web pages to describe the system…. virtually at the push of a button!

The following picture shows the main help page generated 'Index.html' documentation describing the Message Manager System at its highest most general level i.e. the packages or namespaces within the system.



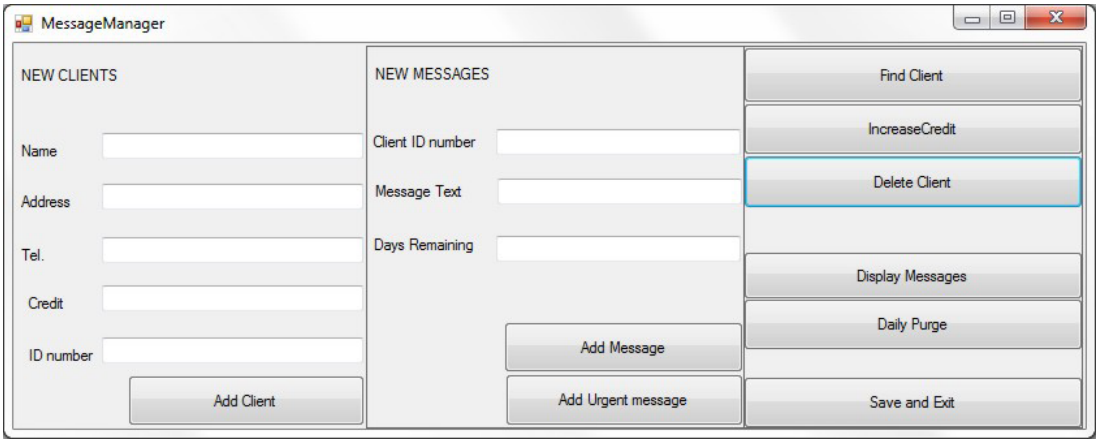
The following picture shows part of the help documentation describing the UrgentMessage class:-



11.14 The Finished System

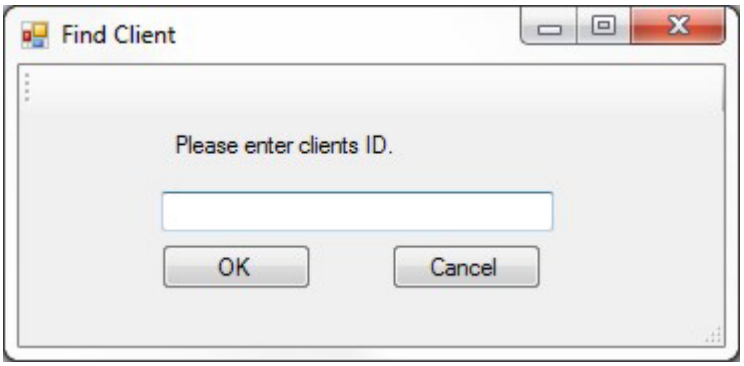
The following screen shots show the finished system.

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.

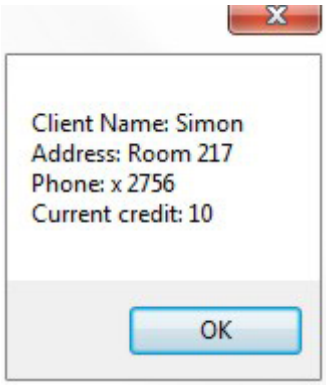


The next two images show the pop up dialogues that appear when the 'Find Client' button is pressed.

Firstly asking for a client ID….



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



The 'Display Messages' button shows each of the messages on the screen 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.

Urgent messages look just like ordinary messages except ***'s are displayed before and after the message.

'Purge Messages' invokes the PurgeMessages() method. Mostly this does nothing visible but decrements the days remaining for each message, decreases the client's credits and deletes the messages if appropriate. Urgent messages are charged at double the rate of ordinary messages. This can be tested by running Find Client before and after doing a daily purge – this should show the clients credit decreasing. If messages exist with an unrecognised client ID and exception will be generated. This exception will be caught by the PurgeMessages() method and an error message will be displayed on the screen.

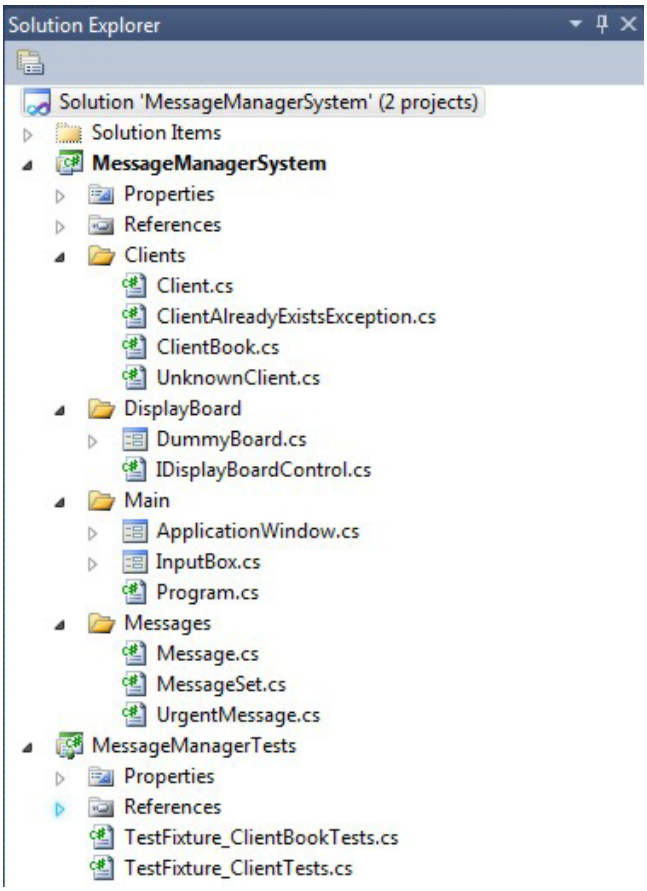
Ultimately of course the idea would be to get the MessageManagerSystem to display the messages on a real display board. This would involve 1) loading the DLL for the real display board 2) creating an object of the real display board in place of the dummy display board 3) passing this object when calling the Display() method. i.e. only two lines of the entire MessageManagerSystem would need to be changed!

11.15 Running the System

The complete, fully commented, source code for the Message Manager system, as described in this chapter, is available with this textbook as zipped file. To view or run the Message Manager system :-

- Install Microsoft Visual Studio 2010 (http://www.microsoft.com/visualstudio/en-us/products/2010-editions). The C# express edition is free and will be perfectly adequate but will not allow you to run the unit tests.
- Download and unzip the file 'OOP Using C#'… available with this book.
- Load the MessageManagerSystem.sln file into Visual Studio, view the code and run within Visual Studio.

In the zip file downloaded are all classes, methods and test cases discussed in this chapter. When viewing the Solution Explorer in Visual Studio you will see all the packages, all of the classes and you should be able to view all of the code with the associated comments (see below..)



You will not able to view to or run the test fixtures with Visual Studio express. Partly to overcome this we have shown many of the test cases in this chapter.

Also inside this zip file is the automated documentation generated by the Sandcastle tool. To view the documentation go to the 'Documentation' folder and double click on the index.html page. This should load the documentation into your web browser software.

If you install Sandcastle Help File Builder (available for free from http://shfb.codeplex.com/) you will be able to load the file MessageManagerSystem.shfbpro available as part of the Message Manager system download. You will then be able to see that the comments for the namespaces have been added to the project properties and if you adjust the output path to an appropriate path for yourself you will be able to rerun this software and see the documentation generated for yourself.

11.6 Conclusions

The fundamental principles of the Object Orientated development paradigm are

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

These principles are ubiquitous throughout the C# language and the .NET 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 C#, 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 automatic tools to help document and test large software projects.

These principles have been exemplified here using C# but the same principles and benefits apply to all OO programming languages and the facilities 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.

If you want a further explanation of the C# language the following book is highly recommended…

Pro C# 2010 and the .NET 4 Platform by Andrew Troelsen

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