Developing Software Agents using .NET

Maria Fasli and Michael Michalakopoulos
University of Essex
Department of Computer Science
Wivenhoe Park, Colchester CO4 3SQ, UK
Email: {mfasli;mmichag}@essex.ac.uk

Table of Contents

1 Introduction ... 1

1.1 Background ... 2

1.1.1 Java ... 2

1.1.2 .NET ... 3

1.2 The .NET Framework ... 3

1.3 Conclusions ... Error! Bookmark not defined.

2 Developing Software Agents ... 6

2.1 Being Social: Handling Communication ... 6

2.1.1 TCP Sockets ... 6

2.1.2 Web Services ... 9

2.2 Constructing Messages: FIPA ACL ... 13

2.3 Reading and Writing XML ... 18

2.3.1 Writing XML ... 18

2.3.2 Reading XML ... 20

2.4 Being Reactive: Responding to Events ... 22

2.5 Being Proactive: Using Threads ... 25

2.5.1 Basic Usage of Threads ... 26

2.5.2 Race Conditions ... 27

2.5.3 Deadlocks, Wait and Pulse ... 31

2.5.4 Threads and GUI ... 36

2.6 Handling Unexpected Errors ... 40

2.7 Being Intelligent ... 43

2.7.1 Making Decisions ... 44

3 A Walkthrough Example Using C# ... 45

3.1 Computer Market Game (CMG) Overview ... 47

3.1.1 Goods ... 48

3.1.2 Auctions ... 49

3.1.3 Utility Functions ... 52

3.2 Requirements and Specifications ... 53

3.2.1 Non-Functional Requirements ... 53

3.2.2 Functional Requirements ... 54

3.3 Deciding on the Modules ... 55

3.4 Creating the Classes ... 55

3.4.1 The GUI ... 55

3.4.2 Handling Communication ... 56

3.4.3 Handling Events ... 58

3.4.4 Agent Description – Decision Making 59

3.5 Running the Agent 65

3.5.1 Game Results 66

4 Conclusions 69

4.1 Discussion – Further Work 69

5 Appendices 70

5.1 Appendix A – Performatives and Their Meaning 70

5.2 Appendix B – Examples of Performatives in an Auction House 71

5.3 Appendix – Zip File 71

6 References 72

1 Introduction

Software agents are computational systems that are capable of autonomous, reactive and proactive behaviour, and they are also capable of interacting with other agents human or artificial (Fasli 2007). The field of agents and multi-agent systems is relatively new even in the short history of Computer Science. It has gained momentum since the mid-1990s as it has been recognised as offering a new software engineering paradigm that will enable us to build complex systems for inherently distributed problems and domains such as the Internet. Despite the many advantages and possibilities of the technology, the full potential of agents and multi-agent systems has not been fully realized. This is partly due to the lack of standard definitions and well-established and accepted methodologies and tools for developing such systems.

Agents and multi-agent systems have been incorporated in both the undergraduate and postgraduate curriculum for a number of years now and institutions choose to offer such specialized courses or incorporate these topics in more general Artificial Intelligence courses. Teaching agents and multi-agent systems presents a number of problems such as:

- a heavy influence of one's own research expertise and specialization in deciding the content of such courses;
- a lack of standard methodologies and tools that practitioners can employ for teaching topics in this area;
- lack of clarity on how concepts and principles usually taught at an abstract level translate at the implementation level.

As a result, textbooks on agents and multi-agent systems (Fasli 2007; Ferber 1999; Wooldridge 2001) examine such topics from a more abstract point of view explaining mainly the concepts and the principles rather than providing more concrete guidelines on how such systems can be implemented.

At the time being, the most widely used language for building software agents in particular for Internet applications, is Java (Bigus & Bigus 2001). A number of toolkits mostly based on Java have been developed to support the agent developer over the last few years such as (Bellifemine et al. 1999; Busetta et al. 1999; Howden et al. 2001; Jeon et al. 2000).

However, it is imperative that the students are exposed and learn how to use other technologies such as .NET, as this we believe may speed up the development process and also will provide the students with additional skills and knowledge important in an increasingly competitive job market. An important part of .the NET framework is related to Internet technologies and applications, and therefore seems to fit well with the domain of software agents.

This project aims to develop practical guidelines for implementing software agents with the Microsoft .NET platform. We seek to investigate and come up with a set of principles that will help the students design and implement agents using the .NET technology. These guidelines are accompanied with code throughout this document and the implementation of a simple agent as a walkthrough example.

1.1 Background

The aim of this section is not to compare the two architectures (Java and .NET), as such comparisons abound in the literature and can also be found on the World Wide Web. However, as Java has been very popular with developers and practitioners, we will provide a brief overview of the two software development systems.

1.1.1 Java

The Java programming language offers a variety of handy features such as automated garbage collection, RMI, network handling, objects serialisation, and at the same time hides a lot of the underlying complexity from the software developer. Additionally, it is available for a wide range of platforms as well as a number of devices, (PDAs, mobile phones), and the generated classes can be run on any platform that provides a Java virtual machine. On top of these features, it has been freely available since its release.

All these features have made Java a popular choice for developers and practitioners around the world that have used it to develop different projects, and more specifically, a series of agent platforms and agent test beds (Bellifemine et al. 1999; Busetta et al. 1999; Howden et al. 2001; Jeon et al. 2000).

1.1.2 .NET

However, other alternatives for agent development do exist, and among these the .NET technology offered by Microsoft. An important part of .the NET framework is related to Internet technologies and applications, and therefore seems to be fitting well within the software agents' domain. .NET is more than just another programming language; it is a development framework and consists of a number of product groups and many different elements (Liberty 2002,2002b; Bagnall et al. 2002):

- The *.NET Framework infrastructure*, which includes the Common Language Runtime (CLR) and the Framework Class Library (FCL). Using software development tools such as Visual Studio, which make use of the .NET infrastructure, it is feasible to develop a variety of standalone or client-server applications. The framework supports a wide set of languages such as C#, J#, and Visual Basic.NET, but is not only limited to these.
- The *.NET Enterprise Servers*, such as new versions of SQL Server, Exchange, Mobile Information Server and more. Software written using the .NET framework can directly interact with these servers.
- *The .NET Compact Framework*, which is a scaled-down version for devices running the Windows CE.NET operating system. Such devices are mostly cell phones, PDAs, etc.

Next an overview of the .NET framework is given, since this is central to the aim of the project.

1.2 The .NET Framework

Figure 1-1 shows the underlying layers of the .NET framework: On top of the framework the higher level components are found such as web services, web forms and window forms. The developers can also access a series of classes that manipulate data either from a database or using XML. Base classes provide standard functionality needed by most programs, such as input/output, string manipulation, security management, network communications, thread management, text management, and user interface design features (MSDN 2006). These three layers constitute the Framework Class Library (FCL),

which consists of more than 4,000 classes, making it one of the largest APIs ever built (Liberty 2002b).

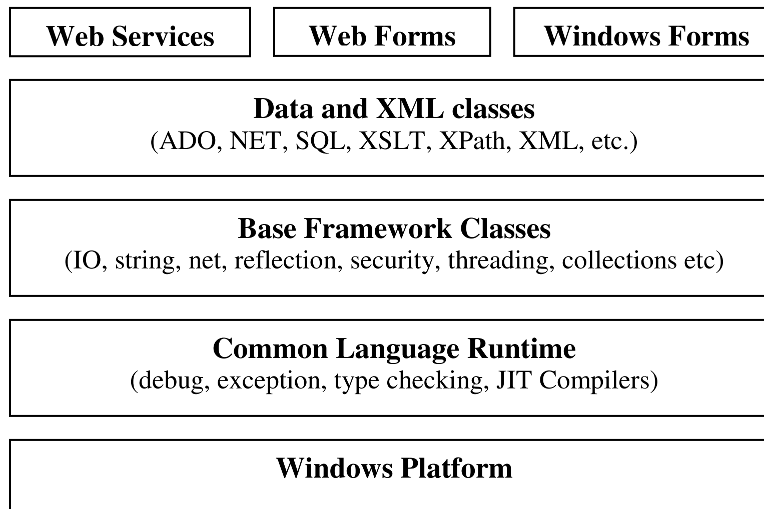


Figure 1-1: Layers of the .NET platform

Probably, the most important layer is the next one, the Common Language Runtime (CLR). The CLR provides the environment in which programs are loaded and run, that is the virtual machine. It is responsible for loading classes, managing security, memory, processes and threads. As the name suggests, the language runtime is common across different languages that support the .NET Framework. Microsoft also provides the Common Language Specification (CLS), which describes how other compilers must produce their output in order to be "compatible" with the .NET framework. Writing such compilers requires that these produce intermediate code which complies with the *Microsoft Intermediate Language* (MSIL). All languages designed to support .NET integration are compiled to MSIL, which is then executed by the CLR. Currently, a number of languages other the ones included in Microsoft Visual Studio offer support for this feature (GDN 2006).

This is an important departure from other frameworks or programming languages: .NET not only supports multiple languages, but also allows them to be integrated in the same project. Developing software in .NET means that one can write a class using VB.NET and subclass it using C#, or any other language that supports .NET; additional features such as catching exceptions, or using polymorphism are also available across different languages.

Whereas Java follows a language-centric approach, meaning that Java is available for different platforms, but at the same time obliges the developer to stick to it, .NET uses a platform-centric approach without caring for the language used: it is mainly aimed at the Windows platform, but allows the integration of different programming languages. This said, there has been a serious attempt to port the .NET platform to Linux (see Mono project at www.mono-project.com).

1.3 Conclusions

This chapter presented the aim of this project, a brief overview of the .NET framework. The goal of this project is to provide the necessary guidelines and teaching material to be used by students in the pursuit of developing software agents.

2 Developing Software Agents

2.1 Being Social: Handling Communication

An agent is rarely useful on its own. Typically, agents will have to coordinate with each other and cooperate to solve a complex problem or accomplish a difficult task or negotiate to share or allocate goods and resources. Central to coordination is the ability of an agent to communicate with others. For instance, communication is very common in virtual marketplaces or distributed problem solving. In most of existing agent platforms or agent infrastructures the most usual way of communication is by means of the TCP/IP protocol suite. Variations of means of communication are RMI (Remote Method Invocation), CORBA, web services and e-mail. Since RMI is a Java technology and CORBA is rather outdated, this section mainly discusses the usage of TCP sockets and web services.

2.1.1 TCP Sockets

The main classes to use when building software agents that need to communicate over TCP/IP are `TCPClient` and `TCPListener`. `TCPClient` is used for agents that want to connect as clients to a specific server such as TAC or e-Game.

The example below illustrates the usage of the `TCPClient` class:

```
public class ClientExample
{
    //hostname and port are the two variables that need to be set in order to
    //achieve a connection to a certain server
    private String hostname;
    private Int32  port;

    //class that provides client network services
    private TcpClient clientConnection;

    //class that provides underlying stream services for a network connection
    private NetworkStream netStream;


    public ClientExample(String h, Int32 p)
    {
        hostname = h;
        port = p ;
        //instantiating the client connection using constructor with hostname, port
        //this also tries to establish a connection;alternatively, the simple,
        //no args constructor can be used, followed by the invocation of connect
        //method. Both ways may generate a SocketException
        try
        {
                clientConnection = new TcpClient(hostname, port);
        }
        catch(Exception ex)
        {
                Console.WriteLine("Exception during connect...{0}",ex);
        }

        //getting the stream associated with the client connection
        netStream = clientConnection.GetStream();
```

```
    } //Constructor: at this point a connection has been achieved


    //simple method to transmit text to a TCP listener socket
    public void send(String cmd)
    {
        // Translate the passed message into ASCII and store it as a Byte array.
        byte[] data = System.Text.Encoding.ASCII.GetBytes(cmd);
        try
        {
                // Send the message to the connected server
                netStream.Write(data, 0, data.Length);
        }
        //there can be many types of exceptions, this one captures them all
        catch (Exception e)
        {
                Console.WriteLine("Exception during send..."+ e);
        }
    }

    //assuming that there has been an appropriately initialised TcpClient object
    // and an appropriately initialised stream
    // and that server is in a state where a message will be transmitted
    // this method will actually read the data from the server
    public String receive()
    {
        // String to store the response ASCII representation.
        String responseData = String.Empty;

        try
        {

                // Buffer to store the response bytes.
                byte[] data = new byte[256];

                //in order to make sure that the next read command
                //will not be executed too quickly
                while(!netStream.DataAvailable)
                {
                        Console.Write(".");
                }
                netStream.Read(data, 0, data.Length);

                // convert data bytes to an ASCII string
                responseData = System.Text.Encoding.ASCII.GetString(data);
                Console.WriteLine("Received: "+ responseData);
        }
        catch (Exception e)
        {
                Console.WriteLine("Exception during read... {0}", e);
        }
        return responseData;
    }

    //convenience method to execute write and then read
    public void submit(String cmd)
    {
        send(cmd);
        receive();
    }

    //closes the stram and associated socket
    public void close()
    {
        netStream.Close();
        clientConnection.Close();
    }

    public static void Main()
    {
        ClientExample ce = new ClientExample("127.0.0.1",5000);
        ce.submit("Hello Remote World");
        ce.close();
    }
}
```

It may be the case that under certain conditions, an agent may want to accept connections from other agents. In such a case, the `TCPListener` class has to be used. The following example shows how to create a TCP socket server:

```
public class MyTcpListener
{
    //main class to use for socket server
    TcpListener myServer=null;

    public MyTcpListener(String ipAddress, Int32 port)
    {
        try
        {
            //server is run on local address and specified port
            IPAddress localAddr = IPAddress.Parse(ipAddress);
            myServer = new TcpListener( localAddr, port);

            // Initiates listening for client requests
            myServer.Start();

            // Buffer for reading data
            Byte[] bytes = new Byte[256];
            String data = null;

            // Enter the listening loop
            while(true)
            {
                Console.Write("Waiting for a connection... ");

                // this is a blocking behaviour:when the next commands is executed
                // the server blocks waiting for a connection
                // or myServer.AcceptSocket()
                TcpClient client = myServer.AcceptTcpClient();

                Console.WriteLine("Connected!");

                data = null;

                // Get a stream object for reading and writing
                NetworkStream stream = client.GetStream();

                //this is the point where the client request should be serviced
                //using threads so as to keep servicing many clients concurrently
                int i;
                // Loop to receive all the data sent by the client
                while((i = stream.Read(bytes, 0, bytes.Length))!=0)
                {
                    // convert data bytes to an ASCII string
                    data = System.Text.Encoding.ASCII.GetString(bytes, 0, i);
                    Console.WriteLine("Received: "+ data);

                    data = data.ToUpper();
                    byte[] msg = System.Text.Encoding.ASCII.GetBytes(data);

                    // Send back a response
                    stream.Write(msg, 0, msg.Length);
                    Console.WriteLine("Sent: "+ data);
                }

                // Shutdown and end connection
                        client.Close();
            }//while
        }
        catch(SocketException e)
        {
            Console.WriteLine("SocketException: "+ e);
```

```
        }
        catch(System.IO.IOException eio)
        {
            Console.WriteLine("SocketException: "+ eio);
        }
        finally
        {
            // Stop listening for new clients.
            myServer.Stop();
        }
    }//constructor

    public static void Main()
    {
        MyTcpListener myTCPListener = new MyTcpListener("127.0.0.1",5000);
    }
}//class
```

When running the MyTcpListener program in a console window the following output is obtained (the program does not exit, but rather waits for incoming connections):

```
Waiting for a connection...
```

Running the ClientExample in another console window generates the following output:

```
......Received: Hello Remote World
```

As it can be seen there are a number of dots that appear before the “Received” text. Obviously the number of dots varies depending on network speed, CPU load, etc. At the same time, the output of the console window of the MyTcpListener program is updated:

```
Waiting for a connection... Connected!
Received: Hello Remote World
Sent: Hello Remote World
Waiting for a connection...
```

2.1.2 Web Services

Another way in which agents may communicate with other agents or a server infrastructure is by making use of web services. By using web services an application can publish its methods, thus allowing for other applications to invoke them over an intranet or the Internet. It may be the case that a client application needs to do some complex calculation but local resources may be inadequate; it can therefore pass its input data to a web service and allow the remote computer to process the data and return the results. In other cases, an agent may be able to invoke a web service that offers certain functionality that the agent does not have, for example, gathering information about road traffic on a certain area, or weather information about a speicifc location.

The main advantage of using web services is that they theoretically are platform and language independent. In practice, there may be certain problems, especially when trying to

pass certain parameters among pieces of code implemented with different languages or on different platforms. Additionally, it is understood that arbitrary objects may not be understood by both ends. For example, a user-developed Bid object in Java may not be passed as a parameter in a method developed using C#, or similarly an object developed in VB or C# does not have a JAVA representation on the other end. Nevertheless, web services are becoming quite popular and as new tools emerge for developing and using them, they offer a transparent way to invoke remote methods.

Invoking web services with .NET is very easy, especially when the developer uses the Microsoft Visual Studio IDE suite. In order to programmatically call an existing web service using .NET, one first needs to know the Uniform Resource Identifier (URI). Then, a reference to the remote web service needs to be established. This can be done by adding a web reference to the existing project, as shown in figure 2-1.

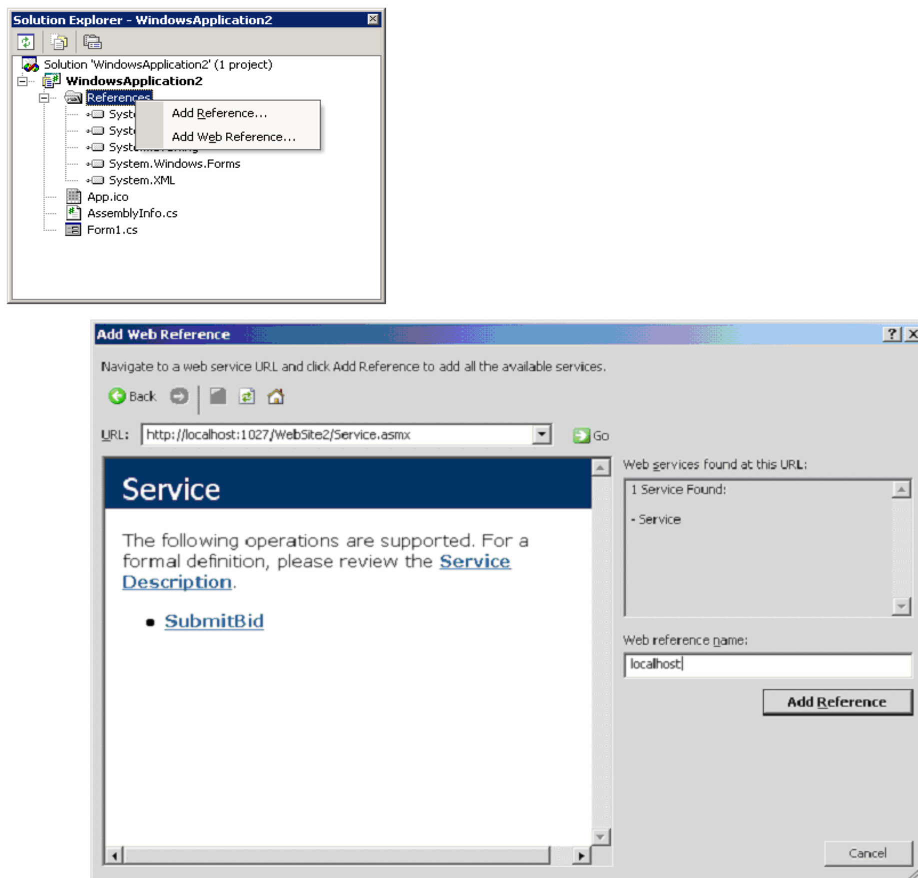


Figure 2-1: Adding a web reference to an existing project

The following example is a web service call implemented using C#. The web service being called has been implemented with VB, but could have been implemented in any

language that supports web services. Once, the web reference is obtained, the web service call is very easy as shown next. In this simple example, there is a form which is used to submit a bid towards an auction using a web service (actually, the web service invoked is a dummy service, which generates a random bid id, but the principle is the same). Figure 2-2 shows this simple form and the result of the web service invocation. The specific web service returns an XML string which encapsulates the newly generated bid id and a command status with a textual description.

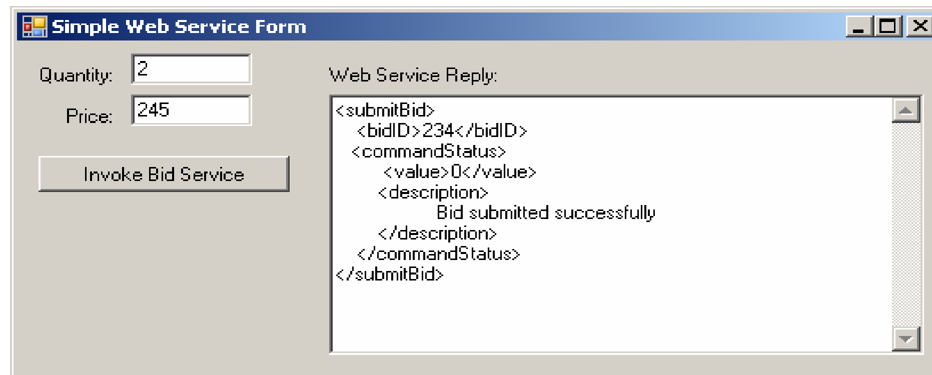


Figure 2-2: Using a simple web service

The code that calls the web service after the web reference has been added to the project is the following:

```
private void button1_Click(object sender, EventArgs e)
        {
           //localhost is the name that has been given to the web reference
           localhost.Service bidService = new localhost.Service();

           int quantity = 0, price = 0; //we need numerical values from the text boxes

           String reply; //web service reply

           bool error = false; //control variable

           try
           {
               quantity = Int16.Parse(txtQuantity.Text);
               price    = Int16.Parse(txtPrice.Text);
           }
           catch(Exception ex)
           {
              MessageBox.Show("There is an error while parsing numerical data,
                                 exception is "+ex);
              error = true;
           }

           //invoke the web service, only if there is no error
           //the web service call should also be wrapped in a try/catch block:
           //server may be down, or another network error may occur
           if (!error)
           {
              reply = bidService.SubmitBid(quantity, price);
              txtReply.Text = reply;
           }
```

```
}
```

The details of the implementation of the actual web service are not important here as the web service could have been implemented using any programming language that provides for web services development. The important thing to know is the URI of the web service, its input parameters and its return values (if there are any). Microsoft Visual Studio makes it easy to develop a web service: all the programmer has to do is to select the appropriate project type when starting a new project and Visual Studio will generate the appropriate declarations, together with a simple “Hello World” service.

The web service code (VB) that is used for the example here is the following:

```
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols

<WebService(Namespace:="http://www.egame.com/")> _
<WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1)> _
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Public Class Service
    Inherits System.Web.Services.WebService

    <WebMethod()> _
    Public Function SubmitBid(ByVal quantity As Integer, ByVal Value As Integer) As
String
        Dim bidID As Integer

        bidID = New Random().Next(10000)

        Return "<submitBid><bidID>" + CStr(bidID)+ "</bidID>" + _
               "<commandStatus><value>0</value><description>" + _
               "Bid submitted successfully</description></commandStatus></submitBid>"
    End Function
End Class
```

2.2 Constructing Messages: FIPA ACL

Once an agent achieves to establish a communication channel with another agent or with a server that provides certain functionality, it may be the case that it needs to interact using a specific protocol or an agent communication language. Among the different communication languages for agents, FIPA ACL (FIPA 2006; FIPA 2006b) and KQML (Labrou & Finin 1997) are the two most popular ones. Another popular format for exchanging messages is XML. Actually, even FIPA ACL allows the content parameter to be set using any language or protocol. Therefore, it is perfectly acceptable to have XML as the content parameter of an ACL message.

The next paragraphs show, one of the many ways to construct an ACL message and also look into constructing an XML message. Typically, a FIPA ACL message contains a series of parameters. Apart from the performative parameter, (which identifies the speech act—that is, the type of the message and the meaning it wants to convey), there is no obligatory parameter that should be included in a message; nevertheless, it is quite usual to include at least the sender, receiver and content parameters. Appendices A and B show the FIPA ACL performatives and their corresponding meaning, together with some examples of their usage. Table 2-1 shows the parameters that may be used in a FIPA ACL message; the example shown here will only include a subset of these parameters.

<table>
<tr><th>Parameter</th><th>Category of Parameters</th></tr>
<tr><td>Performative</td><td>Type of communicative acts</td></tr>
<tr><td>Sender</td><td rowspan="3">Participant in communication</td></tr>
<tr><td>Receiver</td></tr>
<tr><td>reply-to</td></tr>
<tr><td>Content</td><td rowspan="4">Description of Content</td></tr>
<tr><td>Language</td></tr>
<tr><td>encoding</td></tr>
<tr><td>ontology</td></tr>
<tr><td>protocol</td><td rowspan="5">Control of conversation</td></tr>
<tr><td>conversation-id</td></tr>
<tr><td>reply-with</td></tr>
<tr><td>in-reply-to</td></tr>
<tr><td>reply-by</td></tr>
</table>

Table 2-1: Message parameters used in a FIPA ACL message

Here is a typical ACL message with which an agent requests from an auctioneer to process a bid:

```
(request
:sender (agent-identifier :Socrates)
:receiver (set (agent-identifier :eGameServer))
:content
"( (action (agent-identifier :eGameServer)
   (submitBid (auctionid 101)
              (Bid 2 150 )

)"
:language fipa-sl)
```

One of the possible ways to construct a FIPA ACL message is by making use of properties in .NET. Properties allow objects to access data members of a class, while actually implementing the access in a way the user defines. The advantage is that users do not have to call a method to retrieve/set values in data members; they do so, as if they were accessing a data member but at the same time the property mechanism ensures that access is done via a user-defined method.

This is a different approach to the getter/setter methods often found in JAVA and JavaBeans. In JAVA's case the developer actually needs to call these methods to retrieve or set the value of a data member.

The next piece of code shows how properties can be used to provide some simple validation regarding the construction of an ACL message. The validation is about the available performatives: If the programmer creates a message and tries to use a performative which is not supported, then an exception is raised. This can be useful since it allows for early tracing of mistakes while constructing messages. Similar validation checks can be carried out for the content parameter, which can be quite more complex to build.

```
class ACLMessage
{
  //ACL message parameters
  private string sender;
  private string receiver;
  private string performative;
  private string content;

  //this is a readonly property, contains the acutal ACL message
  private readonly string aclMessage;


  public ACLMessage()
  {
      aclMessage = "";
      sender = "";
      receiver = "";
      performative = "";
      content = "";
  }

  //this is how properties are constructed:
  //Notice: 1. Variable in capitals
```

```
//        2. get block returns variable
//        3. set block uses the keyword "value"
//           to set the user value
public string Sender
{
    get
    {
        return sender;
    }
    set
    {
        sender = value;
    }
}




public string Reveiver
{
  get
  {
        return receiver;
  }
  set
  {
        receiver = value;
  }
}


//the Performative property needs to be checked
//so, some extra code goes to the set block
public string Performative
{
  get
  {
      return performative;
  }
  set
  {
     performative = "";
     bool found = checkPerformative(value.ToString());
     //a customised exception at runtime is thrown
     //when the performative is not valid
     if (!found)
         throw new
            InvalidPerformativeException("Performative not in the list:"+value);
     else
         performative = value;
  }
}

public string Content
{
  get
  {
     return content;
  }
  set
  {
     content = value;
  }
}


//this property is readonly,
// so, only, the get block is provided
```

```
public string AclMessage
{
  get
  {
      //formulate the acl message
      //more errors can occur here, if for example,
      //some of the parameters are not set
      //these are not checked here
      string temp = "";

      temp = "(" + performative.ToLower() +
          ":sender (agent-identifier :name " + sender + ") " +
          ":receiver (set (agent-identifier :name " + receiver + ") " +
          ":content " + content +
          " )";
      return temp;
  }
}


//Validation function for performatives
private bool checkPerformative(string candidPerform)
{
    //list with acceptable performatives
    //this can obviously be extended
    string[] performatives = { "accept proposal", "agree", "request" };
    int i = 0;

    //assume that the performative is not in the list
    bool found = false;

    if (candidPerform == null)
      i = performatives.Length + 1;

    candidPerform = candidPerform.ToLower();
    while (i < performatives.Length && !found)
    {
      if (candidPerform.CompareTo(performatives[i]) == 0)
          found = true;
      i++;
    }
    return found;
}
}
```

The class used for the exception is a very simple one:

```
//Very simple, user defined exception
class InvalidPerformativeException : System.Exception
{
     public InvalidPerformativeException(string message):
            base(message)
     {

     }
}
```

By building a very simple test program, the property mechanism can be tested. The form in figure 2-3 is used to set the values of the coresponding parameters and the button is used to retrieve the message from the AclMessage property. This is the property that the agent will use when transmitting the ACL message to another agent.

The code that is actually run when the "Check & Generate" button is pressed is very simple:

```
private void button1_Click(object sender, EventArgs e)
{
    ACLMessage aclMessage = new ACLMessage();

    //Properties are accessed as if they are data variables:
    aclMessage.Performative = txtPerformative.Text;
    aclMessage.Sender = txtSender.Text;
    aclMessage.Reveiver = txtReceiver.Text;
    aclMessage.Content = txtContent.Text;

    //readonly property of aclMessage
    txtReply.Text = aclMessage.AclMessage;
}
```

As figure 2-4 shows, when the performative is set to "delete", (a hypothetical example of an agent asking another agent to delete a file), the exception is thrown, (as the "delete" performative is not supported), and no ACL message is generated.

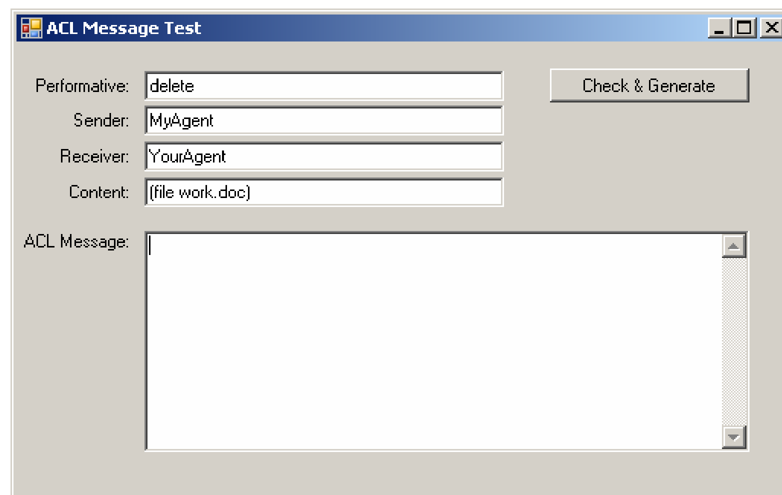


Figure 2-3: Testing the construction of an ACL message

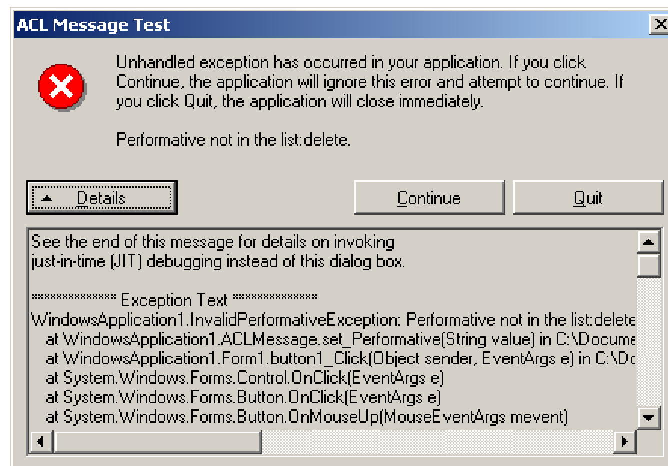


Figure 2-4: Exception thrown during the construction of an ACL message

On the contrary, when the performative is changed to request, which is in the list of the values that the "Performative" property supports, the message is appropriately generated (Figure 2-5).

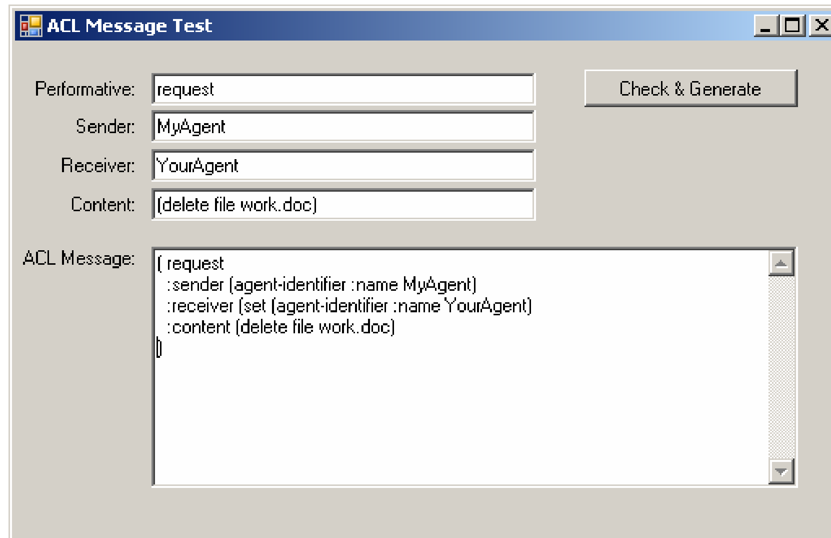


Figure 2-5: Successful example of constructing an ACL message

2.3 Reading and Writing XML

The .NET platform offers a wide variety of classes in order to manipulate data using XML related technologies. For example, there are classes to treat XML as datasets via ADO related classes, handle XML Schema Definitions (XSDs), Extensible Stylesheet Language Transformations (XSLT), and more. However, this section focuses exclusively on reading and writing XML data.

The core classes for handling XML are available in the System.Xml namespace. The main classes involved when reading/writing XML data as text are `XmlTextReader`, `XmlTextWriter` whereas `XmlNodeReader` and `XmlTextWriter` are used when reading and writing XML data as in memory Document Object Model (DOM) trees. DOM allows handling (reading, inserting, deleting elements) of XML data as an object using a tree-like structure.

2.3.1 Writing XML

The next piece of code is used to generate some XML data that are put into a file:

```
static void Main(string[] args)
{
   //used to specify certain settings during XML generation
   XmlWriterSettings settings = new XmlWriterSettings();
   settings.Indent = true;
   settings.IndentChars = ("    ");

   //instantiates the XML writer
   // data will be written to file named books.xml using the
   // settings object
   //*Notice* that XmlWriter is an abstract class, there is no
   //  constructor here, we get back a reference to an XmlTextWriter
   //  object by using the XmlWriter.Create method
   //Alternatively use:
   //    XmlTextWriter writer = new XmlTextWriter("books.xml");
   XmlWriter writer = XmlWriter.Create("books.xml", settings);

   // Write XML data
   writer.WriteStartElement("book");
   writer.WriteElementString("title", "Does anything eat wasps?");
   writer.WriteElementString("price", "10");
   writer.WriteStartElement("author");
   writer.WriteElementString("name", "Various");
   writer.WriteElementString("info", "A collection of New Scientist questions");
   writer.WriteEndElement();
   writer.WriteEndElement();
   writer.Flush();
   writer.Close();
}
```

When run, the program above creates a file with the following content:

```
<?xml version="1.0" encoding="utf-8" ?>
<book>
  <title>Does anything eat wasps?</title>
  <price>10</price>
  <author>
    <name>Various</name>
    <info>A collection of New Scientist questions</info>
  </author>
</book>
```

As it can be seen, the structure of the XML file corresponds to the elements of the program. Additionally, there is appropriate indentation as specified by the settings object. The XmlWriter class also provides other WriteXXXX methods so as to write attributes, comments, integer data and more. For example, the author and names elements can be rewritten as:

```
writer.WriteStartElement("author");
writer.WriteAttributeString("name", "Michael");
```

This would generate the following XML data:

```
<author name="Michael">
```

So, in this case the name of the author appears as an attribute in the author element. Nevertheless, it is often the case that an agent will need to generate XML data as part of an ACL message for example. In order to achieve this, the agent should output the generated

XML to a string, which can later be transmitted over a network stream. XmlWriter offers a constructor which takes a `StringBuilder` object as a parameter:

```
StringBuilder sb = new StringBuilder();
XmlWriter writer = XmlWriter.Create(sb);
.
. Xml writing commands
.
writer.Flush();
String msg = sb.ToString();//put generated Xml into a string variable
```

Alternatively, another XmlWriter constructor allow for the creation of an XML writer that outputs directly to an output stream, such as a stream over the network:

```
XmlWriter writer = XmlWriter.Create(tcpClient.getNetworkStream());
```

2.3.2 Reading XML

An `XmlTextReader` can be used to read the contents of an XML file, or as it was shown earlier with the `XmlTextWriter`, read from a stream, such as a network stream:

```
//Equivalent to using the abstract class XmlReader:
// XmlReader reader = XmlReader.Create("books.xml");
XmlTextReader reader = new XmlTextReader("books.xml");

while (reader.Read())
{
    //First, get the type of the node, other values here maybe:
    //Attribute, Cdata, Comment,EndElement,Entity,XmlDeclaration,etc.
    if (reader.NodeType==XmlNodeType.Element)
    {
        if (reader.LocalName.Equals("title"))
        {
            Console.WriteLine("Title: {0}" ,
                    reader.ReadElementContentAsString());
        }
        if (reader.LocalName.Equals("price"))
        {
            Console.WriteLine("Price: {0}" ,
                    reader.ReadElementContentAsDouble());
        }
}
```

The code above generates the following output:

```
Title: Does anything eat wasps?
Price: 10
```

There are however, alternative ways to read an XML document. For example, there is alternative easy way to retrieve all the book titles from the following XML file, without using the XmlTextReader:

```
<?xml version="1.0" encoding="utf-8" ?>
<books>
  <book>
    <title>Does anything eat wasps?</title>
    <price>10.0</price>
    <author name="Various">
       <info>A collection of New Scientist questions</info>
    </author>
  </book>

  <book>
    <title>The curious incident of the dog in the night
        time</title>
    <price>6.0</price>
    <author name="Mark Haddon">
       <info>From his series: books for children</info>
    </author>
  </book>
  <book>
    <title>The ruins</title>
    <price>16.0</price>
    <author name="Scott Smith">
      <info>From his series: books NOT for children</info>
    </author>
  </book>
  </books>
```

In order to extract the titles, the developer has to make use of the `XmlDocument`, `XmlNodeList`, `XmlReader` and `XmlNode` classes. The following piece of code shows how this can be achieved:

```
//an XmlDocument object is used to load the XML file into the memory
XmlDocument doc = new XmlDocument();
doc.Load("books.xml");

XmlNodeList titlesList;
XmlNodeReader nodeReader;

//it is necessary to show the XPath to the element we need to retrieve
titlesList = doc.SelectNodes("books/book/title");

//now a list with the titles has been created,
//an XmlNodeReader can be used to retrieve the element
//we need to iterate through the list
foreach (XmlNode titleNode in titlesList)
{
    //instantiate an XmlNodeReader from each node
    //*read* paragraph following the code here, an alternative way exists
    nodeReader = new XmlNodeReader(titleNode);
    nodeReader.Read();
    Console.WriteLine("Title : {0}", nodeReader.ReadElementString());
}
```

In this example, the goal was to retrieve the title of each book; delcaring and instantiating an XmlNodeReader for this was an overkill, as the titleNode already contains the information we need. In the previous example, the XmlNodeReader object is rather redudant, instead the InnerXml property of the titleNode can be used:

```
foreach (XmlNode titleNode in titlesList)
```

```
{
    Console.WriteLine("Title : {0}", titleNode.InnerXml);
}
```

The output in both cases is the same:

```
Title : Does anything eat wasps?
Title : The curious incident of the dog in the night time
Title : The ruins
```

Using the XmlNodeReader can be useful, when more information needs to be retrieved, rather than a single element. Consider the case, where each node actually represents a `<book>` rather than a `<title>`:

```
XmlDocument doc = new XmlDocument();
doc.Load("books.xml");
XmlNodeList booksList;
XmlNodeReader nodeReader;
//now the XPath points to a book,so each node is a book
booksList = doc.SelectNodes("books/book");
foreach (XmlNode bookNode in booksList)
{
    //each node is a book, so we need to retrieve its elements using an XmlNodeReader
    nodeReader = new XmlNodeReader(bookNode);
    nodeReader.Read();
    Console.WriteLine("title : {0}", nodeReader.ReadElementString());
    Console.WriteLine("price : {0}", nodeReader.ReadElementString());
    Console.WriteLine("author: {0}", nodeReader.ReadElementString());
}
```

2.4 *Being Reactive: Responding to Events*

During its lifespan, an agent will typically have to respond to different situations and handle different events. These events may be generated by the actions of the agent itself, or may even be generated by other agents or the environment in which the agent lives and acts.

Event handling is very common in applications, which come with a Graphical User Interface (GUI), and developers very frequently make use of event handling when they need their applications to respond to user actions such as clicking buttons, closing windows, etc.

Event handling in the .NET platform is based on the delegates' model. Actually, making use of delegates is mostly related when using C#, but as it is a fundamental concept in event handling, it is presented here as the method to use when handling events. In other languages of the .NET platform, such as VB, the way to handle events is more simplified, although behind the scenes the same principles apply.

A delegate in C# is a reference type used to encapsulate a method with a specific signature and return type. It allows the developer to declare methods, which conform to the

declared signature and use these in a variety of situations, for example, when implementing callbacks or handling events.

Suppose for example, that a developer wants to create a utility class that provides a generic sort method. The method is generic in that it allows for sorting of an array that can hold objects of a certain class, which are unknown until the runtime. In order to create such a class, it is necessary to provide a method, which is able to compare any type of objects. Of course, such a method is very much related to the specific objects. This can be resolved using a delegate type, as shown in the next example of using delegates as a callback method.

```
class GenericSort
{
    //delegate method: any method that matches the signature and return type
    // can be used in the Sort method
    public delegate int CompareObjects(Object o1, object o2);

    //the objects array to be sorted
    object[] arrayObj;

    public void setArray(Object[] aO)
    {
       arrayObj = aO;
    }


    public void Sort(CompareObjects compare)
    {
       bool sorted = false;
       Object temp;
       while(!sorted)
       {
          sorted = true;
          for(int i=0; i<arrayObj.Length-1; i++)
          {
              //1 means that the second object (arrayObj[i+1]), should come
              // before the first one (arrayObj[i]).
              if( compare(arrayObj[i], arrayObj[i+1])==1 )
              {
                 temp         = arrayObj[i];
                 arrayObj[i]  = arrayObj[i+1];
                 arrayObj[i+1]= temp;
                 sorted = false;
              }
          }
      }
}//GenericSort
```

In order to create arrays of objects that can be sorted using the above class, the developer needs to create a class and write a method which conforms to the signature and return type of the delegate method **CompareObjects**. For example:

```
class Student
{
   String name;
```

```
//simple property definition
public string Name
{
    get
    {
       return name;
    }
    set
    {
       name = value;
    }
 }


  public Student(String n)
  {
     name = n;
  }

  // students are ordered alphabetically
  public static int CompareStudents(Object o1, Object o2)
  {
      Student s1 = (Student) o1;
      Student s2 = (Student) o2;
      return (String.Compare(s1.name, s2.name) < 0 ? 0:1);
   }
}
```

From another class the developer now instantiates the delegate as follows:

```
GenericSort.CompareObjects   theStudentDelegate   =
    new GenericSort.CompareObjects(Student.CompareStudents)
```

Assuming that studentsSort (an instance of class GenericSort) is properly instantiated the developer can write (see example on accompanying material):

```
studentsSort.Sort(theStudentDelegate);
```

Typically, delegates are used for handling events in C#. Handling events in C# involves the following steps:

- Declaring the name of the event using the keyword `event` and associating it with a delegate method.
- Declaring the delegate method using the keyword `delegate`, that is, stating the return type and signature of the method.
- Writing the code that raises the event to the “subscribed” methods that want to receive the specific event.

The following piece of code shows these steps:

```
public class MyAgent
{
      //BidSubmittedHandler   is   the   delegate   which   is   associated
      // with the event named OnBidSubmitted. It is usual but not required
      // to start the names of events with the word On
      public event BidSubmittedHandler OnBidSubmitted;
```

```
        //when handling events and declaring the delegate, the first parameter
        // is the object generating the event, the second one is a type
        // inheriting from EventArgs, usually defined by the user as it is
        // the case here.
        public delegate void BidSubmittedHandler(Object o,BidInfoEventArgs bi);
     .
     .
     .
       //the next piece of code shows how this class would generate an
       // OnBidSubmitted event
       public void submitBid(int quantity, int price)
       {
           .
           .
           //this if statement actually checks that there are subscribed
           // methods that want to receive events of type OnSubmitBid
           if(OnBidSubmitted!=null)
           {
               //the event args would also include some info with regards to
               // the efficieny of the bid, for example, if it was rejected,
               // if it won any items, etc.
               BidInfoEventArgs bia = new BidInfoEventArgs(quantity, price);
               OnBidSubmitted (this, bia);
           }//if
       }//submitBid
   }
```

Assuming that there is another class in the application that wants to receive event of type OnSubmitBid, the following code would have to be written in it:

```
class MonitorAgent
{
   MyAgent myAgent = new MyAgent();
   myAgent.OnSubmitBid += new MyAgent.BidSubmittedHandler(this.logEvent);

   public void addText(Object o,BidInfoEventArgs bi)
   {
     .
     .
     .
     logger.write(bi.quantity + " " + bi.price)
   }
}
```

2.5 Being Proactive: Using Threads

Being proactive is another important feature of software agents. Proactiveness refers to anticipating the users' actions and acting on their behalf in advance. This behaviour can often be achieved by handling tasks in parallel, or by proposing certain actions at certain times. Threads, (as well as the Timer class), offer the ability to perform a number of tasks at the same time. This is handy when building software agents and in many cases this ability is very important for demonstrating intelligent and autonomous behaviour. This

section shows how to use threads in .NET and discusses some of the problems that may come up during their usage.

2.5.1 Basic Usage of Threads

Using threads in .NET mainly involves the usage of `Thread` class and the `ThreadStart`, `ParameterizedThreadStart` delegates, which all reside in the System.Threading namespace. The piece of code that needs to be executed in a thread needs to be in a method which is instantiated via a delegate and then passed as a parameter in the constructor of a Thread object.

The following example shows has to start a simple thread:

```
class Program
 {
     static void Main(string[] args)
     {
        //ThreadStart is a delegate method that takes no parameters
        //...so SomeThreadJob is the actual method that does the job
        //it needs to be instantiated via the delegate and then
        //passed on in the constructor of the Thread class

        //notice that here the method SomeThreadJob is static
        //... it is merely passd in the delegate
        ThreadStart myWork = new ThreadStart(SomeThreadJob);
        Thread thread = new Thread(myWork);

        //this is how the thread is started
        thread.Start();

        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine("Main Counting... {0}", i);
            Thread.Sleep(700); //suspends thread execution for 700millis
        }
        Console.WriteLine("Finished Main...");
        Console.ReadLine();
     }

     //this method is actually executed in a thread
     public static void SomeThreadJob()
     {
        Console.WriteLine("Inside SomeThreadJob, Thread started...");
        for (int i = 1; i <= 5; i++)
        {
            Console.WriteLine("Thread Counting... {0}", i);
            Thread.Sleep(500); //suspends thread execution for 500millis
        }
        Console.WriteLine("Finished thread...");
     }
 }
```

Running the program generates the following output:

```
Main Counting... 1
Inside SomeThreadJob, Thread started...
Thread Counting... 1
Thread Counting... 2
Main Counting... 2
Thread Counting... 3
```

```
Main Counting... 3
Thread Counting... 4
Thread Counting... 5
Main Counting... 4
Finished thread...
Main Counting... 5
Finished Main...
```

In addition to the `ThreadStart` parameter, the `ParameterizedhreadStart` allows for the creation of threads in which an object can be passed as a parameter. The following lines of code show how:

```
static void Main(string[] args)
{
    ParameterizedThreadStart myWork = new ParameterizedThreadStart(SomeThreadJob);
    Thread thread = new Thread(myWork);

    //An object of type SomeClass is created, and its state is altered
    SomeClass sc = new SomeClass();
    sc.someData = 10;

    //the thread is started passing the object as a parameter, the Thread
    // uses this object when calling the method SomeThreadJob
    thread.Start(sc);
}

//Notice however, that the method actually takes an Object parameter
//The object needs to be converted back to the original class type
public static void SomeThreadJob(Object sc)
{
     SomeClass temp = (SomeClass)sc;

     Console.WriteLine("Inside SomeThreadJob, Thread
                       started,obj.SomeData={0}",temp.someData);
     for (int i = 1; i <= 5; i++)
     {
         Console.WriteLine("Thread Counting... {0}", i);
         Thread.Sleep(500);
     }
     Console.WriteLine("Finished thread...");
}

class SomeClass
{
    public int someData;
}
```

2.5.2 Race Conditions

Naturally, the usage of threads poses certain problems with regards to shared resources. Consider the common scenario where two travel agents make a seat reservation for the same flight. Making a reservation mainly involves two steps: a) checking availability and b) making the reservation. The step of making the reservation implies that the number of available seats is reduced by the number of available seats. Race conditions may occur when more than one thread have an interest to modify the same resource, in the case of booking a flight, the common resource is the number of seats. Consider the following

timeline of events between two travel agents that want to make a ticket reservation on the same flight:

Agent A wants to make 1 ticket reservation for flight XXX.

Agent B wants to make 1 ticket reservation for flight XXX.

Agent A checks seats availability, there is 1 ticket available, `avail_ticket=1`.

Agent B checks seats availability, and sees there is 1 ticket available, `avail_ticket=1`.

Agent A proceeds with the reservation: `avail_ticket=avail_ticket-`1. Now the number of available tickets is 0, but Agent B does not know this.

Agent B now makes a ticket reservation too, and the number of available tickets is now –1, which is a problem for any programmer that develops ticket reservation systems. The problem is demonstrated in the code below, where two threads invoke the `BookTickets` method in which they repeatedly try to make ticket reservations.

```
class Program
{
    static int availableTickets = 5;

    static void Main(string[] args)
    {
       //We use ParameterizedThreadStart delegate to pass on
       //the agentID to the BookTicket method
       ParameterizedThreadStart myWork = new
                                ParameterizedThreadStart(BookTickets);
       Thread thread1 = new Thread(myWork);
       Thread thread2 = new Thread(myWork);

       //agent IDs
       AgentIdentifier agID1 = new AgentIdentifier("AgentA");
       AgentIdentifier agID2 = new AgentIdentifier("AgentB");

       //Threads start
       thread1.Start(agID1);
       thread2.Start(agID2);

       Console.ReadLine();
    }

    public static void BookTickets(object objID)
    {
       //use this info when outputting text to the console
       AgentIdentifier agID = (AgentIdentifier)objID;

       //the code is repeatedly executed by each thread,
       //simulating a number of (independent) ticket requests
       while (true)
       {
           if (availableTickets > 0)
           {
              Console.WriteLine("{0} - There are available tickets({1}), I will
                                  book one!", agID.agentName,availableTickets);
              //simulate some working load, for example network traffic
              Thread.Sleep(100);
```

```
                //make the reservation
                availableTickets = availableTickets - 1;
                Console.WriteLine("{0} - Available tickets after booking:{1}",
                                  agID.agentName, availableTickets);
            }
            else
                break; //if there are no available tickets,thread exits
        }//end while
    }//end BookTickets
}//end class


class AgentIdentifier
{
   public string agentName;

   public AgentIdentifier(String name)
   {
      agentName = name;
   }
}
```

The code generates the following output, where the problem of race conditions on the shared vaiable availableTickets is shown in the last line:

```
AgentA - There are available tickets(5), I will book one!
AgentB - There are available tickets(5), I will book one!
AgentA - Available tickets after booking:4
AgentA - There are available tickets(4), I will book one!
AgentB - Available tickets after booking:3
AgentB - There are available tickets(3), I will book one!
AgentA - Available tickets after booking:2
AgentA - There are available tickets(2), I will book one!
AgentB - Available tickets after booking:1
AgentB - There are available tickets(1), I will book one!
AgentA - Available tickets after booking:0
AgentB - Available tickets after booking:-1
```

As it is evident from the last line, AgentB made a booking while there were no available tickets! Despite AgentB having checked for ticket availability, AgentA at the same time instant, managed to book the last ticket; nevertheless AgentB decremented the available tickets. This is a problematic situation and is clearly a bug.

The solution to the problem is to protect the shared resource (available tickets), by allowing a single agent to perform a full reservation operation (check availability, book the ticket), before allowing the other agent to do the same thing. .NET offers such a locking mechanism by using the `Monitor` class; protecting a crucial section is feasible by wrapping it with the Monitor.Enter() and Monitor.Exit() methods. They both take an object as a parameter; this object is used to identify the lock, and if two or more threads are to share a resource they should acquire their locks on a common object.

The code below is the new version of the BookTickets method:

.
.

```
.
//object used for locking, declared at class level
public readonly static object countLock = new object();

public static void BookTickets(object objID)
{
    //use this info when outputting text to the console
    AgentIdentifier agID = (AgentIdentifier)objID;

    //the code is repeatedly executed by each thread,
    //simulating a number of (independent) ticket requests
    while (true)
    {
       //countLock is a reference to an object declared
       // and instantiated once at the class level
       Monitor.Enter(countLock);
       if (availableTickets > 0)
       {
          Console.WriteLine("{0} - There are available tickets({1}), I will book
                                 one!", agID.agentName, availableTickets);
          //simulate some working load, for example network traffic
          Thread.Sleep(100);

          //make the reservation
          availableTickets = availableTickets - 1;
          Console.WriteLine("{0} - Available tickets after booking:{1}",
          agID.agentName, availableTickets);
       }
       else
       {
          //IMPORTANT:The lock needs to be released before exiting!
          Monitor.Exit(countLock);
          break; //if there are no available tickets,thread exits
       }
       Monitor.Exit(countLock);
       //it is very important to use Monitor.Exit to release the lock!
    }//end while
}//end method
```

When the new version is run the correct, expected output is produced:

```
AgentA - There are available tickets(5), I will book one!
AgentA - Available tickets after booking:4
AgentB - There are available tickets(4), I will book one!
AgentB - Available tickets after booking:3
AgentA - There are available tickets(3), I will book one!
AgentA - Available tickets after booking:2
AgentB - There are available tickets(2), I will book one!
AgentB - Available tickets after booking:1
AgentA - There are available tickets(1), I will book one!
AgentA - Available tickets after booking:0
```

Notice how there is no –1 at the last line, and also notice that either AgentA, or AgentB gets to check the ticket availability and books the ticket; the output alternates between the two agents, (the actions of checking availability and booking the ticket has now become an atomic action—one that cannot be broken apart).

Nevertheless, there is one more issue with the solution shown here: If the `Monitor.Exit(countLock)` command before the break is removed, the code will almost definetely "hang". The reason is that one thread acquires the lock, but before `break`ing

does not appropriately release it, therefore not allowing the other thread to continue not allowing the program exit appropriately. Additionally, if an unhandled runtime error occurs at any point between the Monitor.Enter(), Monitor.Exit() commands (such as a network timeout), then the Monitor.Exit() command will not be executed and the program will not appropriately terminate. Admittedly, the code presented here is not very elegant, due to the reasons that runtime exceptions are not caught, and having to place Monitor.Exit() in two points is not particularly nice, yet it demonstrates the solution to the shared resource problem and pinpoints the issues that may arise by not allowing the Monitor methods to execute appropriately.

An alternative mechanism in C# exists by using the lock keyword. Essentially, the lock keyword operates by locking an object and replaces the pair of the Monitor commands:

```
while (true)
{
    //alternative method to perform a lock
    lock (countLock)
    {
         if (availableTickets > 0)
         {
            Console.WriteLine("{0} - There are available tickets({1}), I will book
                              one!", agID.agentName, availableTickets);
            //simulate some working load, for example network traffic
            Thread.Sleep(100);

            //make the reservation
            availableTickets = availableTickets - 1;
            Console.WriteLine("{0} - Available tickets after booking:{1}",
                              agID.agentName, availableTickets);
          }
          else
          {
            break; //if there are no available tickets,thread exits
          }
     }//end of lock
}//end of while
```

2.5.3 Deadlocks, Wait and Pulse

In the case of the shared resource problem as described previously, not having a way to turn a series of operations to an atomic action resulted a program bug. The worst-case scenario however, is that inappropriate usage of the locking mechanisms will result a deadlock, a situation in which each thread holds a lock to a resource and tries to acquire another lock on a different resource, which in its own turn is held up by another thread which has no intention of releasing it.

This problem is illustrated below, where two threads are started and they acquire locks on two shared resources, whereas at the same time, they make a request to acquire a second

resource. The time that thread1 sleeps is such that it allows for the demonstration of the deadlock. Thread1 acquires the lock on shared resource, an object named lockOne, then is suspended momentarily, before trying to acquire the lock on lockTwo object. The other thread firstly acquires lockTwo, and then tries to acquire lockOne. Running the program is mostly likely to cause thread1 to wait forever while attempting to lock lockTwo, the reason being that thread2 holds that lock while waiting for thread1 to release lockOne (which will not happen).

```
class Program
{
   //these are the two shared resources
   static readonly object lockOne = new object();
   static readonly object lockTwo = new object();

   static void Main(string[] args)
   {
      ThreadStart myWork1 = new ThreadStart(ThreadJobOne);
      ThreadStart myWork2 = new ThreadStart(ThreadJobTwo);
      Thread thread1 = new Thread(myWork1);
      Thread thread2 = new Thread(myWork2);

      //Threads start
      thread1.Start();
      thread2.Start();

      Console.ReadLine();
   }


   public static void ThreadJobOne()
   {
       Console.WriteLine("ThreadJobOne - Locking on lockOne object");
       lock(lockOne)
       {
           Console.WriteLine("ThreadJobOne - Locked on lockOne object");
           Console.WriteLine("ThreadJobOne - About to lock lockTwo object");
           Thread.Sleep(1000);
           lock (lockTwo)
           {
                Console.WriteLine("ThreadJobOne - Got a lock on lockTwo object");
           }
           Console.WriteLine("ThreadJobOne - Lock on lockTwo object released");
       }
       Console.WriteLine("ThreadJobOne - Lock on lockOne object released");
   }


   public static void ThreadJobTwo()
   {
       Console.WriteLine("ThreadJobTwo - Locking on lockTwo object");
       lock(lockTwo)
       {
           Console.WriteLine("ThreadJobTwo - Got a lock on lockTwo object");
           Console.WriteLine("ThreadJobTwo - About to lock lockOne object");
           lock (lockOne)
           {
                Console.WriteLine("ThreadJobOne - Got a lock on lockOne object");
           }
           Console.WriteLine("ThreadJobOne - Lock on lockOne object released");
       }
       Console.WriteLine("ThreadJobOne - Lock on lockTwo object released");
```

```
    }
}
```

The outcome of this code is that the program does not terminate and the user has to terminate the program by hitting the CRTL-C keys; the output is shown below:

```
ThreadJobOne - Locking on lockOne object
ThreadJobOne - Locked on lockOne object
ThreadJobOne - About to lock lockTwo object
ThreadJobTwo - Locking on lockTwo object
ThreadJobTwo - Got a lock on lockTwo object
ThreadJobTwo - About to lock lockOne object
```

Unfortunately, deadlocks may be hard to reproduce and debug, which is why extra caution is required when programming with threads. In the example above, the deadlock is quite easy to detect, but there can be cases where a deadlock is the result of many different classes involved and is harder to detect. The `Monitor` class provides some more methods for cases like these. Modifying one of the two methods as shown below solves the problem of the deadlock:

```
public static void ThreadJobTwo()
{
      Console.WriteLine("ThreadJobTwo - Locking on lockTwo object");
      Monitor.TryEnter(lockTwo);
      Console.WriteLine("ThreadJobTwo - Got a lock on lockTwo object");
      Console.WriteLine("ThreadJobTwo - About to lock lockOne object");
      if( Monitor.TryEnter(lockOne) )
      {
        Console.WriteLine("ThreadJobOne - Got a lock on lockOne object");
         Monitor.Exit(lockOne);
       }
       else
          Console.WriteLine("ThreadJobOne -- UNABLE TO ACQUIRE A LOCK on
                             lockOne object");
       Console.WriteLine("ThreadJobOne - Lock on lockOne object released");
       Console.WriteLine("ThreadJobOne - Lock on lockTwo object released");
       Monitor.Exit(lockTwo);
}
```

By making use of the method `Monitor.TryEnter()`, the lock may not be acquired, since the program will not wait forever for the lock to be released. An overloaded version of this method allows for a controlled attempt to acquire the lock within a certain time period, before continuing execution. The method returns a boolean value which is used to test whether the lock was acquired successfully or not. Runnning the previous program with the new piece of code, results in smooth execution of the program and generates the following output:

```
ThreadJobOne - Locking on lockOne object
ThreadJobOne - Locked on lockOne object
ThreadJobOne - About to lock lockTwo object
ThreadJobTwo - Locking on lockTwo object
ThreadJobTwo - Got a lock on lockTwo object
ThreadJobTwo - About to lock lockOne object
```

```
ThreadJobOne -- UNABLE TO ACQUIRE A LOCK on lockOne object
ThreadJobOne - Lock on lockOne object released
ThreadJobOne - Lock on lockTwo object released
ThreadJobOne - Got a lock on lockTwo object
ThreadJobOne - Lock on lockTwo object released
ThreadJobOne - Lock on lockOne object released
```

Two other methods found in the `Monitor` class are also very important when handling race conditions and deadlocks: `Monitor.Wait` and `Monitor.Pulse`. When the first one is used on a thread, the thread releases the current lock held on the specific object and blocks the thread until it is reacquires it. The second command is used to signal any waiting thread that the lock status of the object has changed.

The code shown next –the Consumer and Producer example– demonstrates the usage of these two commands. The problem in this case is that the consumer should wait for the producer to produce items before consuming any items, and similarly the producer should wait for the consumer to consume some products when the queue is full.

```
class Program
{
    static ProducerConsumer productionLine = new ProducerConsumer();

    static void Main(string[] args)
    {
        //the two threads
        ThreadStart consumerWork = new ThreadStart(ConsumerJob);
        ThreadStart producerWork = new ThreadStart(ProducerJob);

        Thread thread1 = new Thread(consumerWork);
        Thread thread2 = new Thread(producerWork);

        //Threads start
        thread1.Start();
        thread2.Start();

        Console.ReadLine();
    }


    //the job that produces the products
    public static void ProducerJob()
    {
       Random rnd = new Random();

       for (int i = 1; i <= 10; i++)
       {
          String s = "Product " + i;
          productionLine.produce(s);//put it in the queue

          //sleep for a random time
          Thread.Sleep(rnd.Next(200));
       }
    }

    //the job that consumes
    public static void ConsumerJob()
    {
       Random rnd = new Random();
```

```
        for (int i = 1; i <= 10; i++)
        {
           //consume the product
           String s = productionLine.consume();
           //sleep for a random time
           Thread.Sleep(rnd.Next(1000));
        }
     }

  }


     class ProducerConsumer
     {
        //object used for locking
        readonly object listLock = new object();

        //maximum capacity
        const int FULL = 3;
        Queue<String> queue = new Queue<String>();


        public void produce(String o)
        {
           //producer enters a shared resource, queue
           lock (listLock)
           {
              while (queue.Count == FULL)
              {
                 Console.WriteLine("Queue is full, producer is waiting...");
                 Monitor.Wait(listLock);
              }
              queue.Enqueue(o);
              Console.WriteLine("Produced:{0}", o);

              //before releasing the lock,
              //producer signals waiting thread to continue
              //as there is now at least one product available
              Monitor.Pulse(listLock);
           }
        }


        public String consume()
        {
           String s;
           //consumer enters a shared resource, queue
           lock (listLock)
           {
              //consumer will try to consume all available products
              while (queue.Count == 0)
              {
                 //if there are no available products, the lock
                 //is released and the thread goes into wait state
                 //only continues if a pusle signal is caught
                 Console.WriteLine("\t\tQueue is empty, consumer is waiting...");
                 Monitor.Wait(listLock);
              }
              Monitor.Pulse(listLock);
              s = queue.Dequeue();
              Console.WriteLine("\t\tConsumed:{0}", s);
           }
           return s;
         }
           }//class ProducerConsumer
```

The code above generates the following output, and as it can be seen, the two threads alternate their execution in the intended way:

```
                    Queue is empty, consumer is waiting...
Produced:Product 1
                  Consumed:Product 1
Produced:Product 2
Produced:Product 3
Produced:Product 4
Queue is full, producer is waiting...
                  Consumed:Product 2
Produced:Product 5
Queue is full, producer is waiting...
                  Consumed:Product 3
Produced:Product 6
Queue is full, producer is waiting...
                  Consumed:Product 4
Produced:Product 7
Queue is full, producer is waiting...
                  Consumed:Product 5
Produced:Product 8
Queue is full, producer is waiting...
                  Consumed:Product 6
Produced:Product 9
Queue is full, producer is waiting...
                  Consumed:Product 7
Produced:Product 10
                  Consumed:Product 8
                  Consumed:Product 9
                  Consumed:Product 10
```

The topic of synchronising threads can become quite complex; in order to cope with more complex issues, there are even more classes related to threads such as the Mutex class or the ManualResetEvent class. However, the purpose of this section was to pinpoint the most common syncronisation problems that may occur while using threads.

2.5.4 Threads and GUI

Another type of problem that may come up while using threads is related to the Graphical User Interface (GUI).

The most common problem is quite easy to explain and understand: Consider a very simple GUI in which, when a button is clicked, a time consuming and computationally demanding operation takes place, (for example, calculating the number π with a very high precision). When the code is to be executed in the same thread as the thread that created the GUI, then it takes up all the available time, not allowing the original thread to handle other GUI messages/events. The usual behaviour is that when a button is clicked that initiates such a computationally intense operation, the whole GUI becomes unresponsive and seems frozen. The solution in such cases is to use a thread: when the user clicks on a button, the

computationally intense operation is carried out using a thread. This however, may lead to other problems as is next described:

When starting a thread, after having clicked on a button, the GUI is not frozen; this allows the user to click the same button again a number of times. Depending on the case, this may be a serious bug, overwriting data in primary or secondary memory, filling up main memory, etc. How the problem is handled depends on the each time context. It may be the case that this behaviour is intended, and indeed a new thread has to be created each time the button is pressed; for example, there may be a button named “Spawn Pac-Man Enemy” that when clicked, creates a new ghost enemy in a Pac-Man game. It may be the case, that the request is queued, rather than creating a thread straight away, or most commonly it may be the case that no other thread should be started, while the first one is running.

A usual approach for this last case, is to disable the button, so the user cannot click it again and upon the completion of the thread to enable the button again. As it will later be shown, this leads to other problems. Another solution is to have some kind of indication that the thread is actually running and display some kind of informative message to the user when the button is clicked: The thread is started with a `ParameterizedThreadStart` delegate, which allows a parameter to be passed to the method to execute. This parameter can be an object that contains a status field indicating the progress of the thread operation.

A different way would be to allow the user to actually cancel the lengthy operation, such an approach is probably the most difficult one, as the programmer will have to take care of any resources that were used up to that point. Moreover, it may be the case that certain operations cannot be cancelled, for example, think of the case where a complex SQL query is sent to a database server. Upon issuing the command of executing the query, the user has no option but to wait for the query to be executed.

However, this section is concerned with a particular type of problem that may come up when using threads in a GUI application: Often, a thread will need to access some GUI element, which was created from within a different thread. This can lead to a problem when the two threads try to perform an operation on the same GUI element, a problem which resembles the race conditions described earlier. However, the issue with the GUI is more complicated as the programmer cannot predict how a user will interact with the different GUI elements, making synchronization regarding GUI elements quite complex. Moreoever,

it would not be a wise thing to lock controls prior to accessing their members as the .NET underlying mechanism needs to access these GUI elements on its own, for example when refreshing them, repainting them, listening for events, etc.

It may be the case, that if the programmer does not take care of this issue, nothing will go wrong, nevertheless, this will be a non-deterministic behaviour: error-free behaviour cannot be guaranteed as a different thread other than the one that created the GUI element keeps accessing them.

The solution to the problem is to have all the commands that affect a control's state in a method and then pass this method via a delegate to the control's Invoke method. The code below shows how this can be done:

```
public partial class Form1 : Form
{
       public delegate void NoParamsDelegate();
       public delegate void SomeParamsDelegate(int i, bool test,
                                                DateTime dt);
       public Form1()
       {
          InitializeComponent();
       }

       private void button1_Click(object sender, EventArgs e)
       {
          //accessing a control from the same thread that created it
          textBox1.Text = "Inside button1_Click, Invoke Required:" +
                           textBox1.InvokeRequired;

          //start a thrad that performs some lengthy operation
          //this thread will have to access the textbox
          //the textbox was created from a different thread other than
          //the simpleThread thread
          ThreadStart job = new ThreadStart(this.someWork);
          Thread simpleThread = new Thread(job);

          textBox1.Text = "Inside button1_Click, starting thread...";
          simpleThread.Start();
       }

       //this is the method that carries out the thread's task
       //although not implemented here, this could be a lengthy operation...
       public void someWork()
       {
          //assuming at some point that some text needs to be outputted on
          // the control that was created from within another thread...
           // This could lead to problems:
           // textBox1.AppendText("The result of the computation is:"...);

           //we can check if the textBox1 needs to be accessed via a delegate
           if (textBox1.InvokeRequired)
           {
               MessageBox.Show("InvokeRequired for textBox1 at this point!");
           }

           //declaring and using delegates
           NoParamsDelegate myWork = new NoParamsDelegate(MyWork);
           SomeParamsDelegate myWork2 = new SomeParamsDelegate(MyWorkParams);
```

```
            //no params delegate
            textBox1.Invoke(myWork);

            //a delegate with parameters,
            //the Obects array holds the parameters that the
            //myWork2 method will use
            textBox1.Invoke(myWork2, new Object[]{ 1, true, new DateTime() });

        }

        //first simple method to safely access control
        public void MyWork()
        {
           bool req = textBox1.InvokeRequired;
           textBox1.AppendText("\r\nMyWork: Text Appended...");
           textBox1.AppendText("\r\nMyWork: Invoke required here:"+req);
        }

        //second method with params to safely access control
        public void MyWorkParams(int varInt, bool varBool, DateTime objDT)
        {
           bool req = textBox1.InvokeRequired;

           textBox1.AppendText("\r\nMyWorkParams: Text Appended...");
           textBox1.AppendText("\r\nMyWorkParams : Invoke required here:" + req);
           textBox1.AppendText("\r\nMyWorkParams: varInt :" + varInt);
           textBox1.AppendText("\r\nMyWorkParams: varBool:" + varBool);
           textBox1.AppendText("\r\nMyWorkParams: obDT:" + objDT.ToLongDateString());
        }

}
```

When the code above is run, the user is initially presented with the windows form of figure 2-6 (top). When the button is clicked, then the thread is started. As soon as the thread is started, the programmer needs to update the contents of the textbox. However, the textbox was created from within a different thread, and as the dialog box of figure 2-6 shows, (dialog box stating that InvokeRequired is true), it is necessary to perform this operation via the Invoke method of the control. Finally, the remaining code is executed and the output appears, the InvokeRequired method returns false when executed via the delegate. This happens because the code of the methods MyWork and MyWorkParams are executed from within the thread that actually created the textbox.

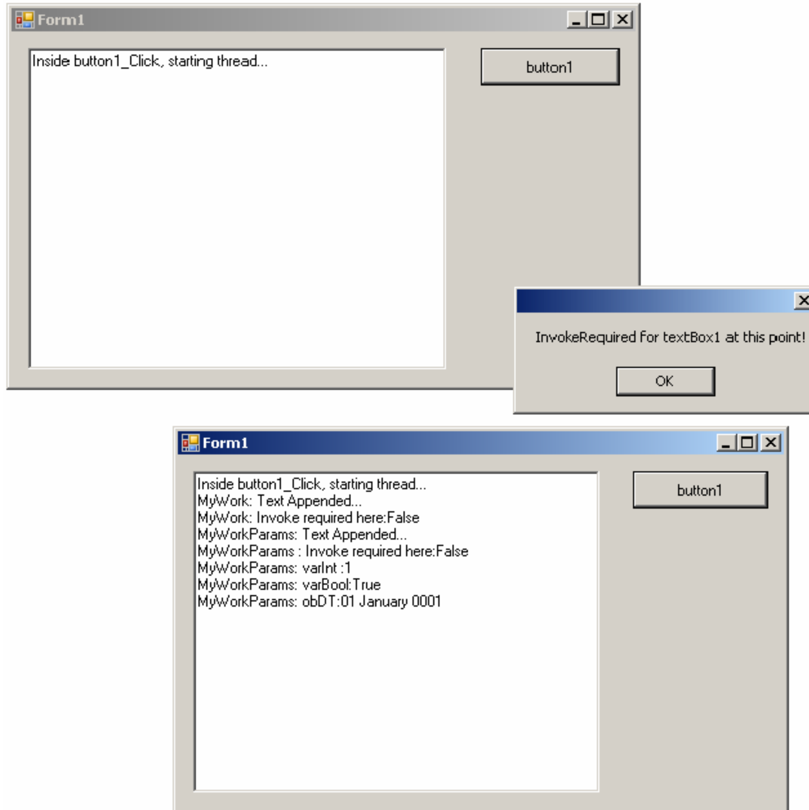


Figure 2-6: Testing threads and GUI

Apart from the Control.Invoke method, another method called Control.BeginInvoke can be used to asynchronously execute code that affects the state of a control. Asynchronous execution implies that this method simply posts a message to the handler of the GUI to execute the method, but rather than waiting for it to execute, it returns immediately.

There are many more methods and issues that may come up when using threads and the Thread class, nevertheless the ones presented here are the most important ones.

2.6 Handling Unexpected Errors

In order to be autonomous, the agent needs to be able to cope with unexpected erroneous situations. These may come up either due to bugs, or due to predictable but unpreventable situations. As far as bugs are concerned, there is not much that software can do when is up and running, after all, no matter how smart software is, it cannot compensate for design flaws, or implementation errors.

However, the developer can, and should, make provisions for handling certain problematic situations that are predictable but may be unpreventable. For example, a navigator robot should be able to operate, even if one of its sensors is damaged. Of course, its efficiency may be reduced, nevertheless, reduced efficiency is preferred over a total inability to operate due to software failure.

In order to handle such situations .NET offers the mechanism of exception handling. The following piece of code handles the runtime erroneous situation that may occur when trying to use a string as an integer:

```
int port;
try
{
   port = Int16.Parse(txtPort.Text);
}
catch(Exception e)
{
   MessageBox.Show(this,"Port should be an integer!","Error");
   return;
}
openConnection(address, port);
```

In the above code, prior to calling the method openConnection, the text of the field `txtPort` needs to be converted to an integer. Since this operation may fail, it is enclosed in a try block and in the case where the conversion does not take place appropriately, the exception is handled in the catch block, and the openConnection method is not called.

As there are many things that may go wrong during execution, there are many types of exceptions. In the above example, the Parse method can actually “throw” the exceptions ArgumentNullException (when the parameter is a null reference), FormatException (when the parameter is not of the correct format, for example a string with characters other than digits), or OverflowException (when the numerical representation of the string is either too small or too big). All these exceptions can be handled by including multiple catch blocks. The point to remember however, is that as all exceptions inherit from Exception. When handling multiple exceptions the more “specific” exception handlers should be placed at the top, followed by the more general ones. The following piece of code is wrong and does not compile:

```
int port;
try
{
   port = Int16.Parse(txtPort.Text);
}
//putting the more generic exception on top causes
//the code not to compile,since it will catch *any*
```

```
//type of exception not allowing the more specific ones
//to be captured.
catch(Exception e)
{
   MessageBox.Show(this,"General Exception!","Error");
   return;
}
catch(ArgumentNullException ae)
{
   MessageBox.Show(this,"Argument exception!","Error");
   return;
}
catch(FormatException fe)
{
   MessageBox.Show(this,"Format exception!","Error");
   return;
}
catch(OverflowException oe)
{
   MessageBox.Show(this,"Overflow exception!","Error");
   return;
}
openConnection(address, port);
```

Whenever an exception occurs, .NET tries to find an appropriate exception handler by unwinding the stack:

```
try
{
       Console.WriteLine("Some operation (1)");
       someFunction();
       Console.WriteLine("Some operation (2), will not be executed!");
}
catch(Exception ex)
{
       Console.WriteLine("General error!","Error");
}

public void someFunction()
{
       int a=3/0;//division by 0 raises an DivideByZeroException
       Console.WriteLine("This will never be executed!")
}
```

In the code above, someFunction raises an exception which is not handled inside the method. The exception is handled at a higher level from the calling block. If an exception handler is not found there, .NET will try to handle the exception at a higher level. If no handler is found, .NET will provide a default handler which will notify the user for the exception.

Apart from the try/catch blocks, there also exists another block, the `finally` block. Any piece of code that is put in the finally block, will be executed regardless of the occurrence of any exceptions or not. This makes the finally block an ideal place to perform tidying up of resources that may have been used inside the try block. For example:

```
try
{
   Console.WriteLine("1. Open a TCP socket here");
   Console.WriteLine("2. Performing some other operations");
   Console.WriteLine("3. Transmit a message over TCP socket");
}
catch(Exception e)
{
   Console.WriteLine("General exception error {0}",e);
   return;
}

finally
{
   Console.WriteLine("4. Close the socket");

}
```

What may not be so obvious here is that if the code that closes the socket, (command marked as 4.), is placed inside the try block, then it may never get to execute if one of the operations 2 or 3 raise an exception; in such a case the socket will not be closed. Of course, there can be more exceptions raised in the finally block, which in their own turn can be caught. The presence of the finally block is not mandatory: if the user is not interested in executing any code that performs tidying up, then the finally block is redundant. Notice however that the finally block is required when there is a try block that is not followed by a catch block (this means that the catch block can be omitted as long as there is a finally block—in such a case the exception is caught by the next available exception handler and then the finally block is executed—however, this is not a common practice).

Apart from catching exceptions, the developer can also create his own exceptions and throw them as necessary. The only prerequisite is that the newly created exception class inherits from `System.ApplicationException`. Such a custom exception has already been used in section 2.2, where an exception of type `InvalidPerformativeException` was created and thrown using the throw keyword when the performative used did not match one of the performatives in the accepted performatives list.

2.7 Being Intelligent

The previous sections in this chapter discussed some of the technical aspects related to building software agents. Certain points such as the ones discussed in the threads and exceptions sections are also related to developing software in general. This should not come as a surprise, developing agents is about developing software, and all the principles that apply for engineering good software, also apply for engineering software agents too.

The issue that was not covered in the previous sections however is one that is mostly related to general principles of developing agents, rather than developing agents with specific tools and/or for a specific platform. These principles, their theoretical background and their implementation have been hot research topics for many decades and this is not likely to change in the years to come. The reason is that pursuing the goal of making intelligent software that has the qualities of what we call "agent" is a very hard one. Agent technology draws knowledge from many different fields —for example, the section about agent communication referred to "performatives", which is a term used in linguistics—and this makes the tasks of agent design and implementation quite hard. Other fields from which agents and multi-agent systems have drawn upon are computer science, robotics, mathematics, economics, physics, psychology, statistics and game theory. Furthermore, depending on the particular problem and domain of application, knowledge on more specialized fields is often necessary, for example, the software agent that needs to make decisions about space probe navigation makes use of specialized physics, aeronautics, etc.

2.7.1 Making Decisions

The successful performance of an agent in a particular environment depends very much on the agent's decision making process. But developing good decision making processes is inevitably a domain dependent issue, and there seems to be no unique, ultimate guidelines on how to build a software agent in this respect, or what algorithms to use for the decision making component of an agent.

There are certain important decisions the developer needs to consider which are related to the goals she wants to achieve, for example will the agent be reactive, proactive, autonomous and if so, up to what extent each of these attributes/qualities should persist in the lifespan of an agent?

A common category of agents are the ones that conform to the Belief-Desire-Intention (BDI) architecture (Russell & Norvig 1995; Weiss 1999):

- Beliefs represent the agent's subjective information about the state it is in, the state of the world, and possibly the state of the other agents. It is important to make a distinction between beliefs and knowledge: Beliefs are subjective and may differ among different agents, even when these agents are presented with the same scenarios. For example, an agent whose goal is to accomplish a task within limited

time may have different beliefs from another agent who tries to achieve the same task but has no time restrictions. Typically, beliefs are stored in a database-like structure from which the agent can derive information as necessary and also may be able to infer new beliefs. Knowledge represents in essence information that can be verified or facts and is often characterized as true belief.

- Desires represent the options or possible goal states of the agent, for example a bidding agent has the desire to obtain the auctioned item. Certain desires however maybe in conflict with other desires. In the case of the bidding agent, the desire to obtain the item may be in conflict with the desire not to pay too much for it.
- Intentions represent those goal states that the agent has committed to bringing about. This often means coming up with a plan and being committed to following it. Quite often, coming up with a plan requires breaking down the task to smaller tasks and solving these ones first. Intentions may be materialized using various algorithms and techniques from the broader field of Artificial Intelligence. Certain problems fall into the category of planning, (e.g. navigation problems), others fall into the category of classification, (e.g. optical character recognition), others in the category of searching (e.g. allocation of bundles to clients), etc.

In order to develop a good software agent, the developer has to take into account the above issues and in most cases, this is not an easy task. Furthermore, it may be the case, that in a dynamic environment, the agent constantly perceives new stimuli, updates its beliefs, revises its desires and consequently needs to come up with new plans that will enable it to achieve its intentions. Obviously, there are many factors that affect an agent's behaviour, for example, availability of information, available computational resources, complex dynamic environments, inability to immediately perceive the outcomes of an action and more.

3 A Walkthrough Example Using C#

This chapter demonstrates how the principles and guidelines described in the previous chapter can be put into practice when building a software agent. The aim is to build an agent that can interact in a virtual marketplace and negotiate over the prices of the items it wants to get hold off. Virtual marketplaces such as e-Game (Fasli & Michalakopoulos 2004) and TAC (Wellman & Wurman 1999), offer agent developers the opportunity to test

various aspects of decision making algorithms, strategies as well as the effectiveness of the market infrastructure.

The testbed used for the development of the walkthrough example is e-Game. e-Game is a flexible auction platform which allows for the setup of different types of auctions, in which participants can be web users or software agents. Additionally, e-Game offers a testbed for agents, since it allows the development of different market scenarios which are based mainly on auctions. In these scenarios, software agents try to obtain different bundles of items in order to satisfy their customers. The items they need to obtain are auctioned in different types of auctions and as customers may have more or less similar likings, the agents need to compete against other agents in order to be able to put together good deals for them.

Therefore, the agents need to exhibit “smart” behaviour in order to achieve the best result. Most of the times they will start with an initial plan that suits the needs of their customers, but as prices go higher, they may have to come up with alternative strategies in order to keep a balance between satisfying their clients and making a profit.

The e-Game platform has been successfully used for teaching agent related topics such as agent communication, game theory, economics, e-commerce and more (course CC435-Agent Technology for e-Commerce, Department of Computer Science, University of Essex). More information about the e-Game platform can be found in (Fasli & Michalakopoulos 2004).

The agent that will be built here is a simple agent that participates in computer market. The next sections describe the game in more detail, the specifications of the agent, its design and its implementation.

3.1 Computer Market Game (CMG) Overview

In the Computer Market Game (CMG), an agent is a supplier/intermediate agent whose task is to assemble PCs for its clients. A CMG game (instance) lasts 9 minutes and features six agents competing against each other. Each agent is acting on behalf of five clients/clients who express their preferences for the various types of components of the PC they would like to obtain. The objective of the agent is to maximize the total satisfaction of its clients by providing them the best configuration of PC possible while at the same time keeping costs at a minimum. A fully assembled PC consists of 3 types of components:

- **A motherboard**: There are three types of motherboards, each one bundled with CPUs of 1.5GHz (MB1), 2.0GHz (MB2) and 2.5 GHz (MB3) respectively.
- **A case**: Cases come in two different types, one packed with a DVD player (C1) and another one with a DVD/RW (C2) combo drive.
- **A monitor**: For the purposes of this game there is only one monitor type.

All three types of goods are traded in separate markets with different rules. The agents communicate via a TCP-based API with the e-Game server via exchanging message strings in a simple protocol of the form: *command:parameter*s, where parameters re *parameter1=value1&Parameter2= value2 ¶meter3 =value3...* A detailed description of the commands that can be used to communicate with the e-Game platform is given at the game's web pages (http://csres43.essex.ac.uk:8080/egame) under the link **Agent Games**.

At the beginning of the game agents receive their clients' preferences about the desired set-up of the computers together with a bonus value for upgrading to a better motherboard (MB2or MB3) and a better case (C2). So for instance, in a particular game instance an agent participating in the game may receive the following as its clients' preferences:

Agent	Client	**MB2**(1.5 GHz)	**MB3** (2 GHz)	**C2** DVD/RW
1	1	110	150	200
1	2	130	200	300
1	3	120	170	250
1	4	150	184	296
1	5	100	156	201

Table 3-1: Example of client preferences for agent 1 in a game

Hence, client 2 of agent 1 has a bonus of 130 for obtaining a 1.5 GHz motherboard and a bonus of 200 for upgrading to a 2.0 GHz motherboard. For upgrading to a better case the

one with the DVD/RW drive the client is offering 300. The Server generates values in the following ranges:

MB2 = [100..150], MB3=[150..200], C2=[200..300]

Each component is available in a limited quantity and can only be obtained via specific types of auctions. The next table summarizes the quantities of the items and the auction types in which they are auctioned:

Component Type	Available Quantity	Available from Auction
Motherboard 1.0GHz (MB1)	17	Mth Price Auction
Motherboard 1.5 GHz (MB2)	8	Mth Price Auction
Motherboard 2.0 GHz (MB3)	5	Mth Price Auction
Case with DVD (C1)	20	Mth Price Auction
Case with DVD/RW (C2)	10	Mth Price Auction
Monitor (M)	30	Continuous Single Seller

Table 3-2: Component types, availability and auctions

3.1.1 Goods

3.1.1.1 Motherboards

There are three types of motherboards: Motherboard MB1 is the standard one and comes with a 1.0 GHz CPU. There are 17 MB1 motherboards available. MB2 is a more advanced motherboard and comes with 1.5 GHz CPU, there are 8 of these available. Finally motherboard MB3 is the most advanced one and comes with a 2.0 GHz CPU and in total there are only 5 such motherboards available. Motherboards are traded in ascending price auctions: one auction for each particular type of motherboard. Only the motherboard suppliers (represented by the e-Game auctioneers) can sell these components. There is no minimum bid for either type of motherboard. The price difference among the motherboards is based on preference for the better ones, as defined by the client utility functions (see below).

3.1.1.2 Cases

There are two types of cases: Case C1 is the standard one and comes bundled with a DVD drive. There are 20 C1 cases available. C2 is a more advanced case and comes bundled with a DVD/RW combo drive. There are 10 of these available. Cases are traded in ascending price auctions: one auction for each particular type of case. Only the case suppliers (represented by the e-Game auctioneers) can sell these components. There is no

minimum bid for either type of case. The price difference among the case is based on preference for the case C2, as defined by the client utility functions (see below).

3.1.1.3 Monitors

There is only one type of motherboard in the Computer Market Game scenario. There are 30 monitors which are sold in a single seller auction. The auction will clear continuously. The monitor supplier is represented in the marketplace by an agent that updates monitor prices using a random walk, starting between 150 and 300. Every time, there is a new match, the price is guaranteed to change again (between 150 and 300) within a minute.

3.1.2 Auctions

All of the auctions run according to the following high-level protocol:

1. An agent submits a bid to the auction.
2. The auction updates its price quote, indicating the current going prices.

The rules for a particular auction specify when, or under what conditions the auction will match the bids and record the transactions.

3.1.2.1 3.1 Bid Format

A bid contains a bid string, representing an agent's willingness to buy and sell the good in an auction. A bid string containing a list of bid points in the following form:

"((q1 p1) (q2 p2) ... (qn pn))"

where qi is a quantity and pi is a price. If there is a point (qi pi) with qi > 0, then it means that the agent is willing to buy qi units of the good at the auction for no more than pi price units per unit of the good. If there is a pair (qj pj) with qj < 0, then it means that the agent is willing to sell qj units of the good at the auction for no less than pj price units per unit of the good. The prices should always be nonnegative.

Note that in the Computer Market Game agents cannot place sell bids.

3.1.2.2 3.2 Auction Scheduling

The auctions are scheduled as follows during the 9-minute period of the CMG game:

- The Mth price auction for the case C1 starts on minute 3 and lasts for a random period between 4-5 minutes.

- The Mth price auction for the case C2 starts on minute 3 and lasts for a random period between 3-4 minutes.
- The Mth price auction for the motherboard MB1 opens in the beginning of the game and lasts for a random period between 6-7 minutes.
- The Mth price auction for the motherboard MB2 starts on minute 4 of the game and lasts for a random period between 4-5 minutes.
- The Mth price auction for the motherboard MB3 starts on minute 6 of the game and lasts for a random period between 3-4 minutes.
- The Continuous Single Seller auction for the sale of the monitors starts in the beginning of the game and closes when all the monitors have been sold or the game ends.

The next figure illustrates the auction types that run during the game. Numbers illustrate the time in minutes.

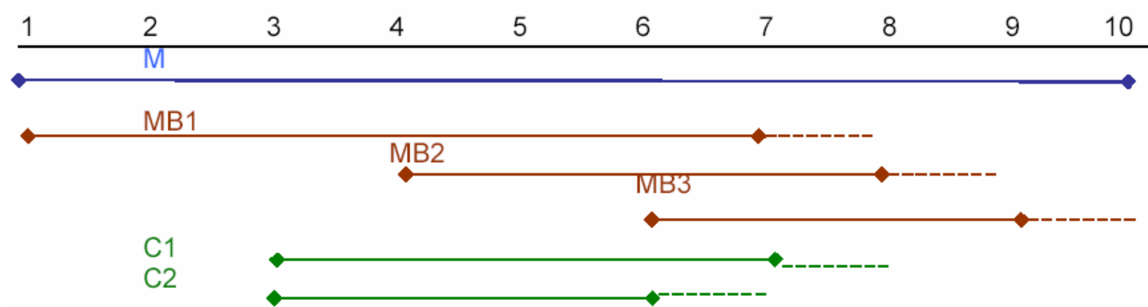


Figure 3-1: Auction Scheduling in the Computer Market Game

3.1.2.3 Motherboard Auctions

Motherboard auctions are standard English ascending multi-unit. These auctions clear and match bids only once, when they close. Price quotes are generated upon the arrival of new bids. Agents may submit buy bids, but not sell bids. Only the motherboard auctioneers may submit sell bids. There are three motherboard auctioneers, one for each type of motherboard, i.e. MB1, MB2 and MB3. The MB1 auctioneer submits bids to provide up to 17 motherboards of that type, for a minimum price of 0. The MB2 auctioneer submits bids to provide up to 8 motherboards of that type, for a minimum price of 0, and finally the MB3 auctioneer submits bids to provide up to 5 motherboards of the respective type.

The price quote is the ask price, calculated as the m-th highest price among all buy and sell bid units, where m = {17, 8, 5} for the MB1, MB2 and MB3 auctions respectively. For

instance, if the following bids were in a MB1 motherboard auction, the ask price would be 0:

((-17 0)), ((2 40) (6 60)), ((4 80)).

If however, the following bids were in a motherboard auction, the ask price would be 60:

((-17 0)), ((2 40) (6 60)), ((4 80)), ((7 100)).

When agents submit new bids, they must "beat the quote", that is, they should submit a buy bid, which is at least equal to the Mth price (ask quote).Agents may not withdraw bids from motherboard auctions. When a motherboard auction clears, the highest price buy units will be matched and the agents will pay the ask price for the motherboards. For instance, assume the following bids were in the MB1 motherboard auction at the closing time:

MB1 Auctioneer bid: ((-17 0)),

- Agent 1: ((8 20))
- Agent 2: ((2 40) (6 60)),
- Agent 3: ((4 80)),
- Agent 4: ((7 100)).

In this example, Agent 4 would win 7 motherboards, Agent 3 would win 4 motherboards, Agent 2 would win 6 motherboards, and Agent 1 would not win any motherboards. The price of the motherboards would be 60.

3.1.2.4 Case Auctions

Case auctions are standard English ascending multi-unit similar to the motherboard auctions. These auctions clear and match bids only once, when they close. Price quotes are generated upon the arrival of new bids. Agents may submit buy bids, but not sell bids. Only the case auctioneers may submit sell bids. There are two case auctioneers, one for each type of case i.e. C1 and C2. The C1 auctioneer submits bids to provide up to 20 cases of that type, for a minimum price of 0. The C2 auctioneer submits bids to provide up to 10 cases of the respective type, for a minimum price of 0. The price quote is the ask price, calculated as the m-th highest price among all buy and sell bid units, where m = {20, 10} for the C1 and C2 auctions respectively. The clearing price is calculated in the same way as in motherboard auctions

3.1.2.5 Monitor Auctions

Monitor auctions are continuous one-sided auctions, and close at the end of the game. Agents may submit buy bids, but not sell bids. Only the Monitor auctioneer may submit sell bids. Price quotes are issued immediately in response to new bids. The price quote is specified as the ask price, calculated as the Mth price among the bids. Monitor auctions clear continually. Any buy bid points that are at least as high as the current ask price will match immediately at the ask price. Any buy bids that do not match immediately remain in the auction as standing bids. A standing buy bid remains in the auction until it is matched by a sell bid with a price at or below the buy bid A standing buy bid matches at the mth-price of the auction. For instance, assume that the Monitor auctioneer places a sell bid of (-30 300), then:

- A bid of (5 370) would match five units at 300 each.
- A bid of (3 290) would not match, and would remain standing in the auction. A subsequent sell bid at 290 or lower would match three units at 290 each from the buy bid.
- A bid of ((2 370) (3 290)) would match two units at 300 each. Since the entire bid would not match, the remaining portion, (3 290), would remain standing in the auction.

3.1.3 Utility Functions

3.1.3.1 Client Utility

When the game starts e-Game issues the Clients' preferences for each agent. Each client is willing to pay a certain amount for upgrading to a better motherboard or case. The agent is only going to receive this bonus provided that a full PC for that client has been assembled and the component is of the advanced type (MB2 or MB3 or C2).

A fully assembled PC is one that has a motherboard, case and monitor. A client's utility, measured in dollars, from a feasible travel package and a feasible entertainment package is given by the following:

CU(Client Utility) = 1000 + Motherboard Bonus + Case Bonus

where Motherboard Bonus = MB2 Bonus or MB3 Bonus, Case bonus = C2 Bonus

A client receives zero utility for an incomplete PC.

3.1.3.2 Agent Utility

At the end of the game, the agent holds several motherboards, cases and monitors. The CMG scorer allocates the agent's goods to its individual clients in order to construct fully assembled PCs. The scorer attempts to construct an optimal allocation, and usually succeeds or comes very close. The Utility Function that an individual agent should attempt to maximise is the sum of all the individual client utilities minus the expenses, minus the penalties:

(CU1+CU2+CU3+CU4+CU5) – Expenses - Penalties

If the agent purchases more than the required quantity of goods for satisfying its 5 clients, then a penalty is applied for each additional item bought which is calculated at 30% of the price that the agent has paid for that item.

3.2 Requirements and Specifications

The agent that will be built will participate in a market scenario related to acquiring a series of interrelated goods by participating in auctions. Next we describe the functional and non-functional requirements of a simple computer market agent.

For the case of this walkthrough example, it seems that functional and non-functional requirements are rather limited. This happens because the server software which provides the market infrastructure, inevitably, narrows down or dictates part of both non-functional and functional requirements. For example, it is assumed that there will be enough network bandwidth to run the agent, or that the agent will have to use TCP sockets to connect to the server.

Nevertheless, there is a minimum set of requirements that the software will have to comply with. We list these in the next two sections.

3.2.1 Non-Functional Requirements

Typically, non-functional requirements specify criteria which can be used to judge the operation of a system as a whole, rather then specifying the requirements for every single component (Pfleeger 1998). So for example, factors such as network bandwidth availability, minimum/maximum response times, reliability, scalability and cost, fall into the category of non-functional requirements. The following requirements are listed:

NFR-1. Robustness. The agent should be able to recover from unexpected conditions that are due to factors beyond its control. For example, during the development phase, it may be necessary to stop and start the agent a number of times. When the agent is re-started, it is desirable to recover its state at the moment prior to exiting and continue from there.

NFR-2. Bounded rationality. The agent should be able to layout a strategy and execute it within reasonable time. Moreover, it should not make excessive use of network resources, (that is, it should not behave in a malicious way regarding its interaction with the server).

NFR-3. Scalability. The software should be built in such a way, so it scales up well to future extensions. Although the market scenario is unlikely to change, it should be relatively easy for the agent developer to make small changes to the code, without breaking existing code. For example, it should be relatively easy to add a strategy for bidding for a new type of resource, or replace the mechanism used for submitting commands to the server, (in case the server specifications regarding communication change).

NFR-4. Reusability. This is linked to the previous requirement and it refers to the fact that parts of the agents should be re-usable by other agents built for the same platform. This means that certain parts of the agent, (for example, the bits of the code that handle communication), should be plugged in to new agents without making any changes to them.

NFR-5. Model-View-Controller Application. The software should be built in such a way, so that the different functions of the agent do not interleave with each other and therefore, these should be modular and independent. For example, if a utilities library is created, this should not be bound to the GUI.

3.2.2 Functional Requirements

Regarding functional requirements, the following points should be noted:

FR-1. The agent should have a minimal interface that would allow for the following:

- Make use of a different login name, password and connection address (hostname and port).

- A connect and abort button. The abort button should abort gracefully, that is closing down the connection and releasing the resources used at that time.
- Display the outcome of interactions with the server, as well as the outcome of the agent's actions in a text area.
- Display any errors that may come up during the interaction with the server or generally during the agent's life span, in a text area. This text area should be distinct from the one mentioned previously.
- Provide a way to view data which are related to the agent's strategy, for example the price changes, or the requested quantities of each time.

FR-2. The agent should be able to exit gracefully when a critical error occurs, (for example, if the server terminates the connection. In such a case the agent should not be trapped for ever in a loop).

3.3 Deciding on the Modules

Since there is a request by the specifications to have a GUI and at the same time, favour scalability and reusability the following elements will have to be created:

- A simple GUI, where information regarding interaction with the server and user intervention for the user will be available, (this is analogous to the View, in the MVC paradigm).
- A number of classes to handle the communication and the methods that may be used by future agents as well, (these classes constitute the model, according to the MVC).
- The agent itself, that, is the class that lays out the bidding strategies and implements them, (the controller of MVC).

3.4 Creating the Classes

Next, a description of the different classes that will have to be constructed is given.

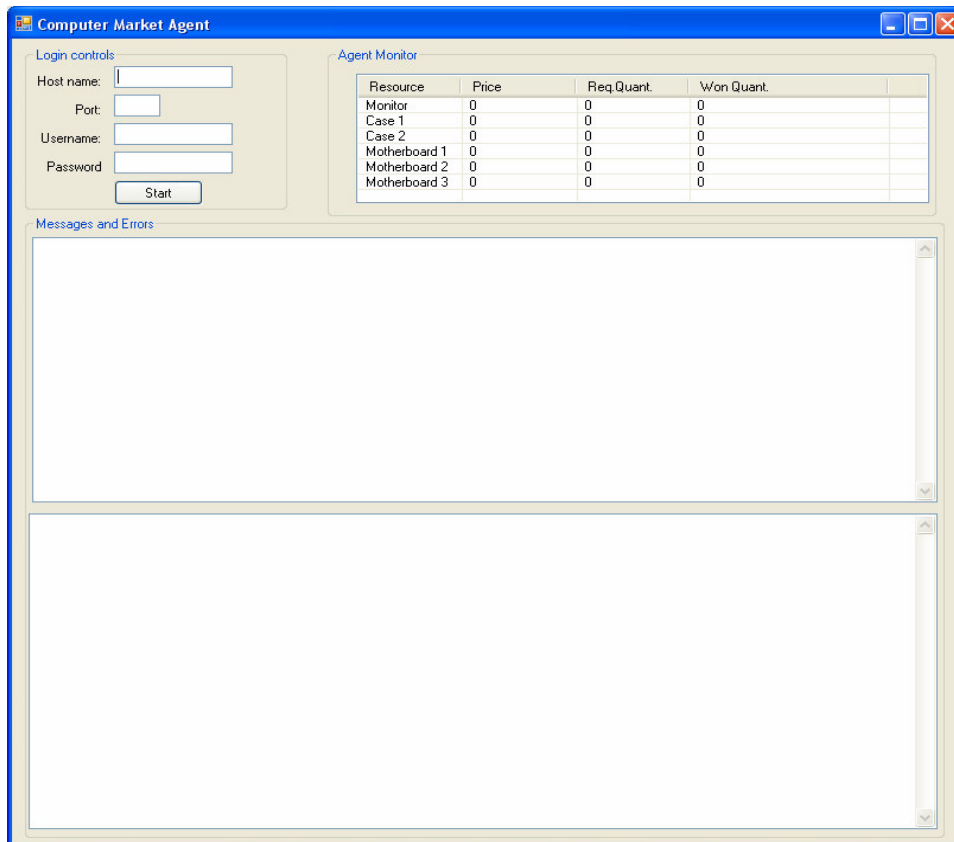
3.4.1 The GUI

Following the requirements, the following GUI is a good candidate for a simple start. It is mainly composed of three control groups, the login controls, the agent monitor and the two text areas.

The login controls allow the agent to connect to different instances of the e-Game server, by providing a different connection address and different user's credentials.

The Agent Monitor, contains only a table, which lists the different resources the agent is interested in. For each resource, the following attributes are monitored: The resource's price, (as retrieved by the e-Game Server), the requested quantity the agent is after, and the quantity the agent would win, in case the auction closed at that moment.

The messages and errors group is composed by two text areas: on the upper area, the "informative" text messages are displayed (messages regarding the decision of the agent and its interaction with the server), whereas in the lower text area, the possible error messages are displayed. Such error messages may be for example, errors due to bad credentials, or due to a connection error with the server.



3.4.2 Handling Communication

Communication with the server is carried out via TCP Sockets. The protocol used for communication is quite simple: Assuming that the agent has achieved a connection with the server, the agent transmits a string to the server, and the server immediately sends back the

reply informing the agent about the outcome of the action. Strings should be null terminated ("\0"). The code for handling the communication is given with the accompanying zip file. However, a short description of the way communication is handled is given in the next algorithm:

```
Function SubmitCommand
Input parameters: cmd    //string Command to submit to server
Returns         : reply //string Reply from the server
Objects used    : netStream  //the stream used for input, output
Vars            : byteArr    //bytes array
                  replyReceived //Boolean var to show end of transmission

//send the command to the server, command should be null terminated!
byteArr = getBytes( cmd )
netStream.write(byteArr )
reply   = empty string

//now we have to wait for the reply from the server, server may take some
// time to post the reply
while NOT !netStream.Available
{
    //wait... do nothing really, this should not take long
}

replyReceived = false
while NOT replyReceived
{
    netStream.read( byteArr )
    reply = reply + getString(byteArr)
    if reply contains null character then
       replyReceived = true
}

return reply
End Function SubmitCommand
```

The important points here have to do with null terminating the strings (this is server specific), allowing the server some time to prepare the answer, and making sure that the whole reply from the server has been transmitted.

3.4.3 Handling Events

One issue that may arise from building software agents has to do with events that may be generated during the lifespan of the agent. Events may be generated by the agent itself when a certain action occurs, or may be generated on the server side and need to captured by the agent on the client side.

In the simple agent developed in this section, events are used to allow the programmer to decide how to handle certain information generated by the agent. The specifications made it clear that there must be a separation of view and model. In order to manipulate with information the GUI presented earlier, the model should be able to access the GUI components were necessary. However, such an implementation would mean that the model would intervene with the GUI, and furthermore the code related to the model would not be reusable; in the future it may be desired that certain info does not appear in the components but is logged to a file. Such a requirement would mean that the model would have to be changed.

In order to overcome the above problem it is feasible to make use of events during the implementation. When the agent performs a certain action an even is generated. Then, it is up to the programmer to handle that event; for example, the programmer may want to log the outcome of the action, or in the example described here, display some information in a certain component.

This section describes how event handling is used in this example.

Deciding on the Events

The example presented here, makes use of events in two classes:

- EGameUtils class: This class provides a series of methods that can potentially be used for building different agents for the e-Game platform. For example, this class provides the implementation for methods such as SubmitBid, GetAskPrice, GetHQWon and more. In this class events are used when a command is submitted to the server. This allows the programmer to decide what to do with the submitted command; for example the programmer may want to add some informative text to a textbox, or write to a log file.
- MyAgent class: This is the class that actually provides the functionality of the agent. The events generated here have to do with handling information during

the interaction of the agent with the server and handling the updated auction prices received from the server.

In both cases, events are triggered when certain information is submitted to the server. This allows the user to decide what to do with this information, for example, append it in a textbox or log it to a file.

Additional information about the event that occurred is encapsulated in an appropriate object: For example, events are put in a `MessageEvent` object, which contains the message generated, plus an indication whether this message is actually an informative message or describes an error condition. For such a simple project, the same object is used for both messages and errors that may occur and is up to the programmer to handle this event as required. In our case, the method that handles these events checks this indication and appends the message either in the textbox area that shows information messages, or in the textbox area that shows error messages.

The other type of event used in this agent is named `DataAuctionEvent` and contains information about the desired quantities of each item the agent wants to obtain, the quantities it would get if the auction closed at that given time, and the current asking prices of the goods. The implementation together with comments on the code can be found in the accompanying zip file.

3.4.4 Agent Description – Decision Making

The example shown here is a fairly simple version of an agent that participates in the computer market game of the e-Game platform. The agent is "complete" in that it connects to the system and participates in a game, nevertheless, its strategy is rather simple.

Regarding the architecture of the agent, it is rather simple and quite basic. The beliefs of the agent about the prices of the auctioned items are directly derived from the information the server provides regarding current price quotes. A more advanced implementation would probably make a prediction about how prices will fluctuate according to past experiences. Nevertheless, in the specific case, this would be quite hard to implement, since such a prediction also depends on the strategies of other players. So, making a prediction about prices would also mean that such a prediction would be made with relation to the other existing players: For example, the agent may come to believe that when agents A, B and C also participate in the auctions, then prices for Motherboard 3 tend

to end up very high. The implementation of such beliefs can be quite hard for such a walkthrough example and additionally, learning about the other agents' behaviour would require many runs.

Regarding the desires of the agent, these are quite simple and in this walkthrough example: at the beginning of the game, the agent decides what its desires will be, that is, it makes a decision about what types of resources to bid for. The agent presented here is not very flexible in that it does not revise its desires as the game progresses. Such a behaviour would be highly desirable as the agent also needs to consider the cost. One can say that the agent in this example has no desire about minimizing the costs and therefore will always stick to its initial desires. It also means that if the agent plays against other agents with a similar behaviour, then acquiring the items can not be guaranteed as the quantities are limited; with no revision of beliefs and desires, the agent is not reactive to any changes that may occur during the game.

Finally, with regards to intentions, these are materialized via the simple strategy the agent follows:

- Firstly, the agent decides on what to bid
- Secondly, the agent follows a very simple plan up to materialize its goals; this plan is carried out for the whole out of the game, till all the auctions close:
  - Request current prices for the auctions it has an interest to place bids
  - Check its inventory for any items it currently wins
  - Bid for the type and number of items it desires to acquire

The strategy of the agent is very simple and most of the decisions in the beginning of the game are based on randomness rather than reasoning. A more advanced agent would make choices by taking into account the bonuses of the clients and would also respond to events that occur during the game (for example, prices going very high). Notice also, that there is no need for the agent to decide on the allocation of the items to customers, as this is carried out by the Computer Market Game on the server side; had it been the case that the agent would need to do the allocation itself, it would have to solve a search space problem (finding the optimum combination of items for each client). The example here demonstrates how a simple agent for e-Game can be built. The high level algorithm is next shown:

```
Algorithm BidderAgent

Objects & variables used:
    GameInfo   : String returned by the server that contains information
                 about the game:  id, startTime, stopTime
    GameParams : String returned by the Server that contains information
                 about game parameters such as client preferences and
                 auction ids
    gameStartTime: long, starting time of the game on the server (millis)
    gameStopTime : long, stopping time of the game on the server (millis)

    //asking prices for monitors, motherboards and cases
    askPriceMonitor : int
    askPriceMB      : int[3], there are three types of Motherboards
    askPriceCase    : int[3], there are three types of Cases

    //asking quantities, how items of each type the agent wants to obtain
    quantMonitor : int
    quantMB      : int[3]
    quantCase    : int[3]

    //hypothetical quantities won, how many items the agent would get if each
    //auction closed down at current time
    hqWonMonitor: int
    hqWonMB     : int[3]
    hqWonCase   : int[3]

    //starting time, auction id and auction state for the monitors auction
    auctionMonStart : long
    auctionMonID    : long
    stateMon        : int, state=1,auction is active,

    //starting times, auction ids and auction state for the motherboards
    //auctions
    auctionMBStart : long[3]
    auctionMBID    : long[3]
    stateMB        : int[3], state=1,auction is active,
                             state=2,auction has closed
    //starting times, auction ids and auction state for the cases auctions
    auctionCaseStart : long[3]
    auctionCaseID    : long[3]
    stateCase        : int[3], state=1,auction is active,
                               state=2,auction has closed

    //boolean variables that tell the agent to bid or not
    bidForMonitor        : boolean
    bidForMB, bidForCase : boolean[3]
//----- End of declarations



    Connect to e-Game and authorise
    GameInfo = Ask info for next game instance

    currentServerTime = Ask server time

    //Wait for scheduled game to start
    gameStartTime = parse(gameInfo,"startTime")
    Sleep(gameStartTime - currentServerTime)

    GameParams = Retrieve game parameters for the specific game

    //auction ids, start and close times for the auctions
    Parse the GameParams object and extract the following info:
        Auction start times: auctionMonitorStart
                             auctionMBStart array
                             auctionCaseStart array
        Auction ids:         auctionMonID
                             auctionMBID
                             auctionCaseID
```

```
    //set up initial strategy: how many items of each item I need, what prices
    //I will pay for them. In this version, these are random decisions, but we
    //always //make sure they sum up to five
    quantMonitor = 5 // always asking for 5 monitors

    //random quantities for motherboards and cases, always sum up to 5.
    quantMB[0]   = Random(5)
    quantMB[1]   = Random(5 - QuantMB[0])
    quantMB[2]   = 5 - (quantMB[0] + quantMB[1])

    quantCase[0] = Random(5)
    quantCase[1] = 5 - quantCase[0]

    currentServerTime = request server's time
    While(gameStopTime<currentServerTime) do
    {
          //retrieve the ask prices for the auctions the agent is interested in
          GetAskPrices()

          //retrieve the
          GetWinningBids()
          SubmitBids()
          currentServerTime = request server's time
    }
End BidderAgent

Method GetAskPrices
{
   parameters used from main algorithm:
      quantMonitors, auctionMonitorStart, askPriceMonitor, stateMonitor
      quantCase, auctionCaseStart, askPriceCase, stateCase
      quantMB,   auctionMBStart,   askPriceMB,   stateMB

   Preconditions: auction ids have been properly updated
   Post Effects : Prices and auction states are updated
                   ( askPriceMonitor, askPriceCase, askPriceMB
                     stateMonitor, stateMB, stateCase)

   //in order to ask for prices for a certain auction,
   // the auction should be running
   if (quantMonitors > 0 AND currentServerTime >= auctionMonitorStart)
   {
      askPriceMonitor = getAskPrice(monitorAuctionID);
      stateMonitor    = getAuctionState(monitorAuctionID);
   }

   //Cases
   for (int i = 0; i < 2; i++)
   {
       if (quantCase[i] > 0 AND currentServerTime > auctionCaseStart[i])
       {
          askPriceCase[i] = getAskPrice(casesAuctionID[i]);
          stateCase[i]    = getAuctionState(casesAuctionID[i]);
       }
   }
   //Motherboards
   for (int i = 0; i < 3; i++)
   {
       if (quantMB[i] > 0 AND currentServerTime > auctionMBStart[i])
       {
          askPriceMB[i] = getAskPrice(casesMBID[i]);
          stateMB[i]    = getAuctionState(casesMBID[i]);
       }
    }
}
```

```
Method GetWinningBids()
{
   parameters used from main algorithm:
      auction ids: auctionMonID, auctionMBID, auctionCaseID

   Preconditions: auction ids have been properly updated
   Post Effects : 1) The boolean variables that tell the agent whether to bid or not,
                     are updated (bidForMonitor, bidForMB, bidForCase)
                  2) The hypothetical quantity won variables are updated
                  3) The auction state variables are updated

    //assume that no bidding will take place
    bidForMonitor  = false;
    bidForCase[C1] = false;
    bidForCase[C2] = false;
    bidForMB[MB1]  = false;
    bidForMB[MB2]  = false;
    bidForMB[MB3]  = false;

    //retrieve hypothetical quantity won and auction state
    hqWonMonitor = GetHQWon(auctionMonID)
    stateMonitor = getAuctionState(auctionMonID)

    //Agent only bids, if it's not winning the quantity it wants
    if(hqWonMonitor < quantMonitors)
        bidForMonitor = true

    for (int i = 0 i < 2 i++)
    {
       hqWonCase[i] = GetHQWon(auctionCaseID[i])
       stateCase[i] = getAuctionState(auctionCaseID[i])

       if(quant < quantCase[i])
           bidForC[i] = true;
    }

    for (int i = 0; i < 3; i++)
    {
        hqWonMB[i] = GetHQWon(auctionMBID[i]);
        stateMB[i] = getAuctionState(auctionMBID[i]);
    }
}
```

```
Method SubmitBids
{
    if (bidForMonitor && stateMonitor == 1)
    {
        quant = 5 - hqWonMonitor;
        price = askPriceMonitor;
        //some "strategy"; I will wait until the ask price is below my threshold
        if(askPriceMonitor <= monitorThreshold && quant > 0)
            SubmitBid(auctionMonitorID, quant, price);
     }

     for (int i = 0; i < 2; i++)
     {
         if (bidForCase[i] && stateCase[i] == 1)
         {
            //check Inventory, do the biddings
            quant = quantCase[i] - hqWonCase[i];
            price = askPriceCase[i] + 5 + rand.Next(25);
            if(quant > 0)
               SubmitBid(auctionCaseID[i], quant, price);
          }
      }

      for (int i = 0; i < 3; i++)
      {
          if (bidForMB[i] && stateMB[i] == 1)
          {
                 //check Inventory, do the biddings
                 quant = quantMB[i] - hqWonMB[i];
                 price = askPriceMB[i] + 5 + rand.Next(25);
                 if(quant > 0)
                    SubmitBid(auctionMBID[i], quant, price);
           }
        }
}
```

3.5 Running the Agent

Figure 3-2 shows the e-Game server, the computer market game and the agent in action. In this example, the server and the agent are run locally, nevertheless, the agent can also connect to any valid e-Game server address.

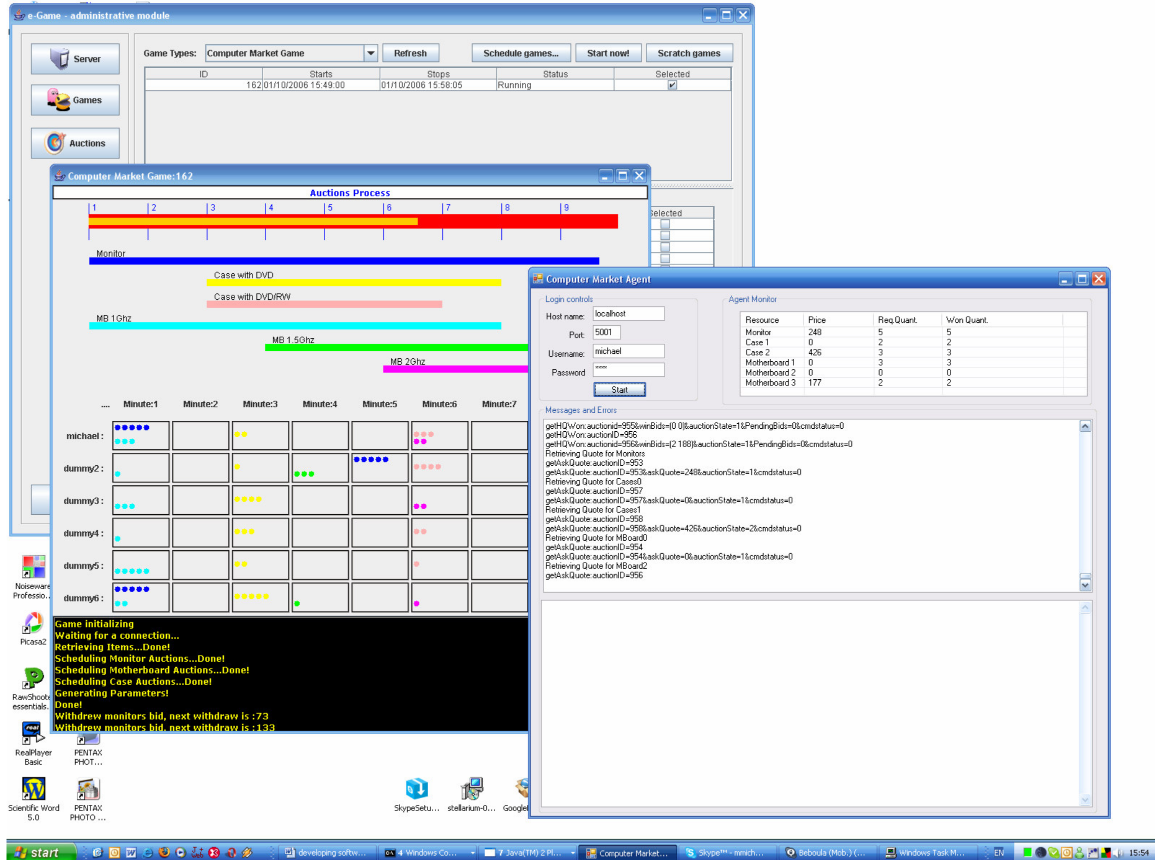


Figure 3-2: Running the e-Game server and the agent

As it is shown, the agent competes against other dummy agents, at the shown stage, the agent wins 5 monitors, 2 simple cases, 3 cases with DVD/RW, 3 motherboards of type 1, and 2 motherboards of type 2. At the end of the game, users can see the results of a specific game instance by using the e-Game web site and navigating to the scores web page.

What is shown next are the allocation of the goods, and the details of the score for each agent. In this case the agent developed here, (agent named “michael”) managed to gather a score of 1001 points. The reason the score is quite low is because the agent built here is not willing to revise its desires and goals. It set a goal of acquiring 2 motherboards of type 3 and remained with this goal even when prices went too high. The next section shows the scores in more detail.

3.5.1 Game Results

CB stands for Case Bonus, MB2 and MB3 stand for Motherboard2, 3 bonuses according to the preferences of each client. The next two columns show the actual bonuses the agent got for the case and the motherboard (Actual Case Bonus—ABC, and Actual Bonus Motherboard—ABM). For the case bonus the agent either gets 0, or the value of CB andfor the case of the motherboard bonus the agent can get 0, the value of MB2 or the value of MB3. The sum of ABC and ABM give the total bonus per client.

What follows is the breakdown of the winning bids, that is how many items of each type and at what prices the agent managed to obtain. The final score is given by subtracting the total expenses and any storage costs, (a random value defined by the server at the beginning of each game to act as an incentive to buy more items than necessary), from the sum of all bonuses.

As the results show for agent named "michael", the agent managed to acquire enough items to build PCs for all the customers. Nevertheless, the prices the agent paid where rather high, and in the case of the motherboards the price was way too high. This resulted a poor overall utility for the agent.

```
Agent:michael
Client     CB     MB2     MB3.  ABC   ABM   Bonus
  0        237    146     178   0     0     1000
  1        235    118     181   235   181   1416
  2        222    102     190   0     0     1000
  3        246    106     159   246   0     1246
  4        264    138     189   264   189   1453
Monitors Bids    : (5 170) =850
Cases Bids (1)   : (2 0) =0
Cases Bids (2)   : (1 426) (1 426) (1 426) =1278
Mboards Bids (1) : (3 0) =0
Mboards Bids (2) : =0
Mboards Bids (3) : (2 1493) =2986
Storage Cost (penalty), for extra items, per unit :234
Extra cases:0, penalty:0
Extra motherboards:0, penalty:0
Extra Monitors:0, penalty:0
Total Punishment= 0
Total Bonuses   = 6115
Total Expenses  = 5114
Utility         :1001
```

```
Agent:dummy2
Client     CB     MB2     MB3.  ABC   ABM   Bonus
  0        222    143     164   222   143   1365
  1        226    105     195   0     0     0
  2        238    139     155   238   139   1377
  3        254    138     167   254   138   1392
  4        235    126     164   235   0     1235
Monitors Bids    : (5 204) =1020
Cases Bids (1)   : (1 0) =0
Cases Bids (2)   : (1 426) (1 426) (2 426) =1704
Mboards Bids (1) : (1 0) =0
Mboards Bids (2) : (3 0) =0
Mboards Bids (3) : =0
Storage Cost (penalty), for extra items, per unit :234
Extra cases:0, penalty:0
Extra motherboards:0, penalty:0
Extra Monitors:0, penalty:0
Total Punishment= 0
Total Bonuses   = 5369
Total Expenses  = 2724
Utility         :2645
```

```
Agent:dummy3
Client     CB     MB2     MB3.  ABC   ABM   Bonus
  0        239    130     186   0     186   1186
  1        285    138     163   0     0     1000
  2        254    107     159   0     0     1000
  3        240    112     153   0     0     0
  4        257    148     171   0     171   1171
Monitors Bids    : (5 187) =935
Cases Bids (1)   : (4 0) =0
Cases Bids (2)   : =0
Mboards Bids (1) : (3 0) =0
Mboards Bids (2) : =0
Mboards Bids (3) : (2 1493) =2986
Storage Cost (penalty), for extra items, per unit :234
Extra cases:0, penalty:0
Extra motherboards:0, penalty:0
Extra Monitors:0, penalty:0
Total Punishment= 0
Total Bonuses   = 4357
Total Expenses  = 3921
Utility         :436
```

```
Agent:dummy4
Client     CB     MB2     MB3.  ABC   ABM   Bonus
  0        215    121     182   0     0     0
  1        248    149     179   248   0     0
  2        279    122     167   279   0     0
  3        241    114     174   0     0     0
  4        237    120     179   0     0     0
Monitors Bids    : =0
Cases Bids (1)   : (3 0) =0
Cases Bids (2)   : (1 426) (1 426) =852
Mboards Bids (1) : (1 0) =0
Mboards Bids (2) : =0
Mboards Bids (3) : =0
Storage Cost (penalty), for extra items, per unit :234
Extra cases:0, penalty:0
Extra motherboards:0, penalty:0
Extra Monitors:0, penalty:0
Total Punishment= 0
```

```
Total Bonuses   = 0
Total Expenses  = 852
Utility         :-852


Agent:dummy5
Client     CB     MB2     MB3.  ABC   ABM   Bonus
  0        232    142     184   0     0     1000
  1        205    126     177   0     0     1000
  2        227    119     185   0     0     0
  3        297    119     183   297   0     1297
  4        290    104     150   0     0     0
Monitors Bids    : (5 187) =935
Cases Bids (1)   : (2 0) =0
Cases Bids (2)   : (1 426) =426
Mboards Bids (1) : (5 0) =0
Mboards Bids (2) : =0
Mboards Bids (3) : =0
Storage Cost (penalty), for extra items, per unit :234
Extra cases:0, penalty:0
Extra motherboards:0, penalty:0
Extra Monitors:0, penalty:0
Total Punishment= 0
Total Bonuses   = 3297
Total Expenses  = 1361
Utility         :1936


Agent:dummy6
Client     CB     MB2     MB3.  ABC   ABM   Bonus
  0        258    124     173   0     0     1000
  1        267    138     160   0     0     1000
  2        292    139     152   0     0     0
  3        237    112     194   0     194   1194
  4        262    148     190   0     148   1148
Monitors Bids    : (5 267) =1335
Cases Bids (1)   : (5 0) =0
Cases Bids (2)   : =0
Mboards Bids (1) : (2 0) =0
Mboards Bids (2) : (1 0) =0
Mboards Bids (3) : (1 1493) =1493
Storage Cost (penalty), for extra items, per unit :234
Extra cases:0, penalty:0
Extra motherboards:0, penalty:0
Extra Monitors:0, penalty:0
Total Punishment= 0
Total Bonuses   = 4342
Total Expenses  = 2828
Utility         :1514
```

4 Conclusions

4.1 Discussion – Further Work

This document presented a set of guidelines for developing software agents. These guidelines were mostly related to how to use various elements from the .NET framework in order to develop functionalities essential to software agents. Such functionalities included communication, constructing messages using xml and threads, responding to events and handling unexpected situations. An effort was made to address the most common issues that may come up when using these structural elements and to provide examples of code that demonstrate correct or incorrect use.

Additionally, a simple agent was built that demonstrated the majority of features presented in the previous sections. The agent was a simple market agent whose goal was to put together certain bundles of computer components to satisfy its clients. The agent must connect and communicate with an existing agent platform (e-Game), lay out a simple bidding strategy, make use of the command set offered by e-Game, use threads so as not to freeze or crash the GUI, and lastly, it made use of the events mechanism. All these elements that were put together had been presented and explained in the previous sections.

The topic of software agents is a very large one and covers a diverse range of fields. Naturally, this document only covers a very basic subset of the elements that are necessary to develop a software agent. For each of the topics presented here, many more issues may come up.

A number of topics were not discussed including principles of object oriented programming, and topics related to algorithms and in particular approaches to Artificial Intelligence such as neural networks, genetic algorithms, planning, scheduling etc.

Nevertheless, with the guidelines presented here and study of the relevant bibliography, the readers should have a basic understanding on how to use the structural elements of .NET in order to develop a software agent.

5 Appendices

5.1 Appendix A – Performatives and Their Meaning

Performative	**When used**
Accept Proposal	The action of accepting a previously submitted proposal to perform an action.
Agree	The action of agreeing to perform some action, possibly in the future.
Cancel	The action of one agent informing another agent that the first agent no longer has the intention that the second agent perform some action.
Call for Proposal	The action of calling for proposals to perform a given action.
Confirm	The sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition.
Disconfirm	The sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true.
Failure	The action of telling another agent that an action was attempted but the attempt failed.
Inform	The sender informs the receiver that a given proposition is true.
Inform If	A macro action for the agent of the action to inform the recipient whether or not a proposition is true.
Inform Ref	A macro action for sender to inform the receiver the object which corresponds to a descriptor, for example, a name.
Not Understood	The sender of the act (for example, i) informs the receiver (for example, j) that it perceived that j performed some action, but that i did not understand what j just did. A particular common case is that i tells j that i did not understand the message that j has just sent to i.
Propagate	The sender intends that the receiver treat the embedded message as sent directly to the receiver, and wants the receiver to identify the agents denoted by the given descriptor and send the received propagate message to them.
Propose	The action of submitting a proposal to perform a certain action, given certain preconditions.
Proxy	The sender wants the receiver to select target agents denoted by a given description and to send an embedded message to them.
Query If	The action of asking another agent whether or not a given proposition is true.
Query Ref	The action of asking another agent for the object referred to by a referential expression.

Refuse	The action of refusing to perform a given action, and explaining the reason for the refusal.
Reject Proposal	The action of rejecting a proposal to perform some action during a negotiation.
Request	The sender requests the receiver to perform some action. One important class of uses of the request act is to request the receiver to perform another communicative act.
Request When	The sender wants the receiver to perform some action when some given proposition becomes true.
Request Whenever	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again.
Subscribe	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes.

5.2 Appendix B – Examples of Performatives in an Auction House

Performative	Situation
Query	Do I have winning bids in this auction?
Request	Process this bid for me
Accept	Ok, I will process the bid
Inform	The bid is now processed
Failure	I am unable to process the bid
Reject	I refuse to process the bid
Subscribe	Tell me when the bid has been processed
Call for proposal	Your attention please, I am starting an auction
Propose	I want to get this for a price of xxx
Not understood	Bid? Auction? I don't get it…

5.3 Appendix – Zip File

The zip file contains the code of the current document including the code for the software agent presented in chapter 4. The tools that were used to develop the code were the freely available versions of the Visual Studio suite, more specifically, the express versions of C# and VB.

6 References

Bagnall B., Chen, P., Goldberg, S., Faircloth, J., and Cabrera, H. (2002). C# for Java Programmers, Syngress Publising.

Bellifemine, F., Rimassa, G., and Poggi, A. (1999). JADE: A FIPA-compliant agent framework. In Proceedings of the Fourth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agent Technology (PAAM-99), pages 97-108, London, UK.

Bigus, J. P. and Bigus J. (2001). Constructing Intelligent Agents Using Java. John Wiley and Sons, Chichester.

Busetta, R., Ronquist, A., Hodgson, A., and Lucas, A. (1999). JACK Intelligent Agents: Components for intelligent agents in Java. AgentLink Newsletter}, pages 2-5.

Fasli, M. and Michalakopoulos, M. (2004). e-Game: A generic auction platform supporting customizable market games. In Proceedings of the IEEE/WIC/ACM Intelligent Agent Technology Conference (IAT 2004), pages 190-196, Beijing, China.

Fasli, M. Agent Technology for E-commerce (2007). John Wiley and Sons, Chichester. (to appear).

Ferber, J. (1999). Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Longman, Reading, MA.

FIPA. (2006). Foundation for Intelligent Physical Agents. http://www.fipa.org

FIPA. (2006b). FIPA ACL Message Structure Specification. FIPA specifications, document identifier 00061, available from www.fipa.org.

GDN. (2006). .NET framework community website http://www.gotdotnet.com/team/lang/.

Howden, N., Ronnquist, R., Hodgson, A., and Lucas, A. (2001). JACK Intelligent Agents - summary of an agent infrastructure. Available at http://www.agentsoftware.com/.

Jeon, H., C. Petrie, C., and Cutkosky, M. R. (2000). JATLite: A Java agent infrastructure with message routing. IEEE Internet Computing, 2(4):87-96.

Labrou Y. and Finin T. (1997). A Proposal for a new KQML Specification, Technical Report TR CS-97-03, February 1997, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD21250.

Liberty J. (2002). Learning C#, O'Reilly, Sebastopol, CA.

Liberty J. (2002b). Building .NET Applications, Programming C#. O'Reilly, Sebastopol, CA.

MSDN. (2006). Technology Overview for the .NET Framework http://msdn.microsoft.com/netframework/technologyinfo/overview/default.aspx.

Pfleeger S. L. (1998). Software Engineering - Theory and Practice. Prentice Hall, Upper Saddle River, NJ.

Russell S. and Norvig P. (1995). Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River, NJ.

Weiss, G. (ed), (1999). “Multiagent Systems, a Modern Approach to Distributed Artificial Intelligence”, MIT Press, Cambridge, MA.

Wellman M. P. and Wurman P. R. 1999. “A trading agent competition for the research community”, In Proceedings of the IJCAI-99 Workshop on Agent-Mediated Electronic Trading, Stockholm, August 1999.

Wooldridge, M. (2001). An Introduction to Multiagent Systems. John Wiley and Sons, Chichester.