Chapter 5 DoIT Development Tool with Generative Programming Concept

5.1 Basic Concept

The total workload of a MMOG development is very huge and need various developers like network programmers, art designers, musicians, and game contents designers and system maintenance workers.

To build a real MMOG, after discuss with developers who have real experience, it can be concluded as the following steps. In the general MMOG development flow, the first step is to define the game contents, for example, game roles, environment, events, game objects, NPCs and game playing scenario. After decide the game contents, the next step is to develop the server side and client side code.

Therefore, in the architecture purposed in previous chapter, the development process can be division into four parts there are client, gateway, server and database. Each part of the development must focus on different characteristics. The client's development focuses on the presentation, control interface, encryption mechanism and the game content. The gateway's and server's development focus on high performance NetEngine, message processor, load balance mechanism, and fault tolerance. And database focuses on data reliability and persistent storage.

Figure 5-1 shows the essential part of a MMOG development from developer's view. The red components (includes the client network engine, gateway network engine, server core and network engine components) are provided by the MMOG platform solution, and the developer will not necessary to pay attention to their operation mechanism, so they can be used directly. The light blue with white dot components are game based components, according to different kinds of MMOG they will have different way to be designed, including the client side presentation, control, game logic and database components. The green components with oblique line are platform based, to design them must according to the assignment way defined by platform, so each MMOG which using the same platform will have the same way and format to design these components.

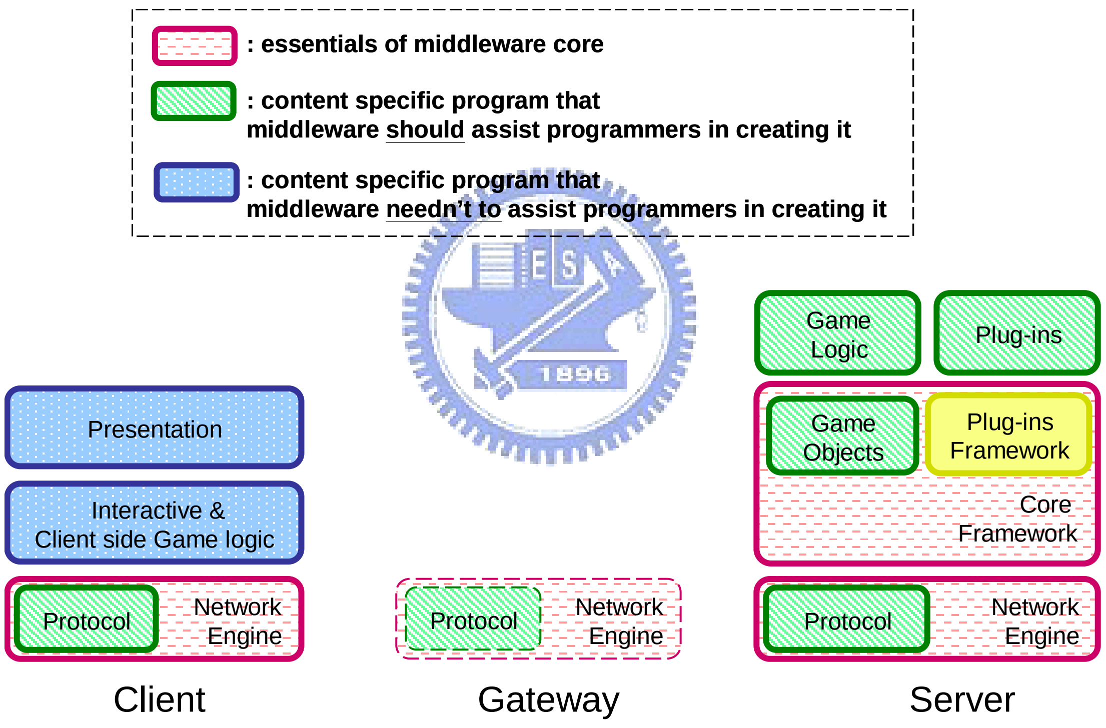

: essentials of middleware core
: content specific program that middleware should assist programmers in creating it
: content specific program that middleware needn't to assist programmers in creating it
Game Logic
Plug-ins
Presentation
Game Objects
Plug-ins Framework
Core Framework
Interactive & Client side Game logic
Protocol
Network Engine
Protocol
Network Engine
Protocol
Network Engine
Client
Gateway
Server

Figure 5-1. Developer's view for MMOG Development

Based on the essential components and development flow, some parts that can be done to accelerate the development speed and reduce the development complexity. First, the game content description document format should be provided by the platform that uses a popular language to define. Therefore, the game content designers could write the description document by using the language and the programmers could understand the document's content more easy and it also reduce the communication time between content designers and programmers which wasted on talk about the content's definition.

Second, programmers will write a lot of the duplicate code during the development process. Because the development of the MMOG component's program will follow an architecture which assigned by the platform and these programs may have the some methods or attributes are the same. Under this condition, the same MMOG component's programs will have a part of codes are similarity and the differentiations between them are the method operation logic and other attributes which are defined by themselves.

Therefore, we draw out the similarity part of the programs and use the XML description document to describe the different parts like methods and attributes. Then we design a code generation engine according to the document's format and the regular of the programs. The code generation engine will generate the code according to the distribution of the document. Therefore, we can load the document into the code generation engine, and then the programs will be generated automatically. At last, the programmers must insert the code into the programs which can't be generated by the engine, and then the development work will be finished.

Therefore, this chapter introduces a development system framework for the DoIT MMOG platform. This dissertation use XML to edit a MMOG content description document and also design a code generation engine, then we can generate the code by load the document into the engine. Therefore, the whole MMOG development works will become more simply, fast and elasticity.

5.2 Code Generation Engine

The development system we introduced will focus on supporting the development of deep blue part components in Fig 5-1 and implementing it on the framework described in previous chapter.

Figure 5-2 shows the overview of MMOG Code generation system. Each generator handle a kind of code, it will store this kind of program's architecture and the part of the same code and will have some method to generate the part of the different code according to received data.

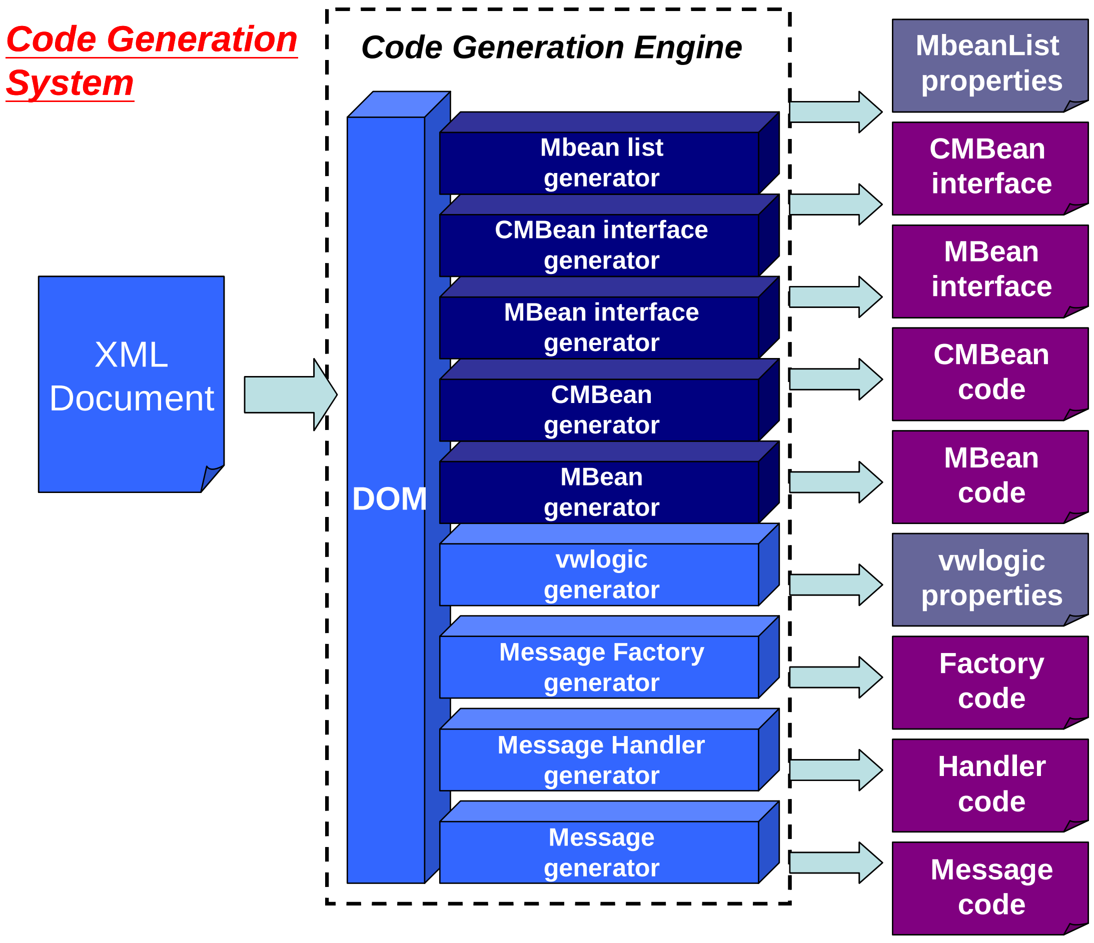


Figure 5-2. Code Generation System of DoIT platform

In traditional development process without generative programming support. After defined the game contents document, the programmers must development the programs according to the definition of contents. The programmers must development all components' programs of the MMOG, the whole work is very complexity and complicated. In order to reduce the programmers' load, development tool here loads the document (which is written in XML by system analyst or programmers) into the code generation engine and generates part of the programs automatically. The generated programs still need to be edit by programmers, because the code generation engine only generate the code which can be described, and others like method operation logic which could not be described in the description document must be write into the program by them self. The code generation engine use a XML parser to parser the XML description document, and send the data which were got in the parser process to the corresponding generator, then the generator will be responsible to generate the codes.

5.2.1 Protocol-driven code generation

The DoIT platform provides the high performance game servers and gateways but doesn't include the clients. Because the client side development will be different according to different kind of the MMOG. The platform keeps the elasticity of the client side design, it allows the client side development according to game developer design themselves. The client side design can be implement by JAVA or C++ or other program language and can use the high performance network engine which provided by platform. The only thing which the programmer should be considered in the client side development process is the message protocol definition; it must be implemented follow the server side definition.

At the server side, there are some components must be implemented according to the definition of the platform. There are message, message handler, message factory, NPC configure, and server configure, vwlogic list document and game objects. Figure 5-3 shows the DOIT platform inner message process flow, it explains the relationship between the message, message handler and message factory. When an unknown message is sand to the server, it will be process at the ControlMessageHandler first. According to the AVID of the message, the message will be delivery to the relative region. AVID is the player's avatar id in the virtual world, and each AVID will be assign to one region according to the player's position in the virtual world. The region will find out the message handler at the Object Adapter according to the type value of the message, and then send the message to the handler. The handler will cast the unknown message to the original message type by using the message factory. And then execute the message operation, and create an update Message return to the client.

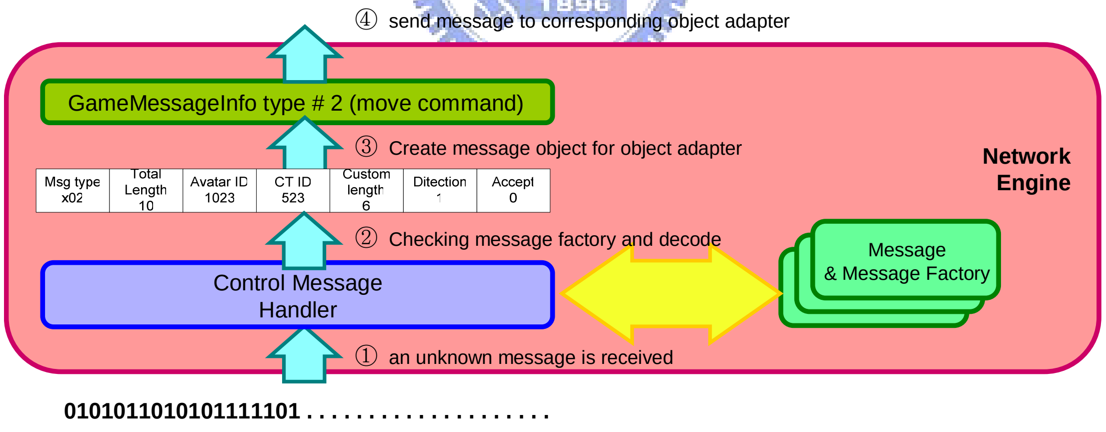

④ send message to corresponding object adapter
GameMessageInfo type # 2 (move command)
③ Create message object for object adapter
Network Engine
Msg type x02
Total Length 10
Avatar ID 1023
CT ID 523
Custom length 6
Ditection 1
Accept 0
② Checking message factory and decode
Message & Message Factory
Control Message Handler
① an unknown message is received
0101011010101111101 . . . .

Figure 5-3. DOIT Platform Inner Message Process Flow

The components which programmers must development are the messages, message handlers, and message factories in the inner message process flow and they are the generality part of the work load in the server side development. Therefore, the development system which we presented will focus on to these components' development. In order to descript these components, the first step is to define the XML schema document according to the format of the message, and find out the relationship between these components. In the DOIT platform, each message is defined as a byte buffer. In the Figure 5-3, there is an unknown type message be sent to the server. This message is composed by two components. First is the fixed component, they include a message type, total message length, avatar id, player id and the length of custom component. Each message will have the same format at the first component and next component is the custom format. According to the different messages' function, the client will need to send some different attributes to the servers for the game logic operation. Therefore, the custom component is composed by some attributes. In the DOIT platform, we support some types of the attributes include int, string, Boolean, byte, short, long and float. For example, a player login message is sent when a new client join to the virtual world and it must include the player id and password attribute. So the custom component in the player login message is composed by a player id string and a password string, the full format is show in figure 5-4.

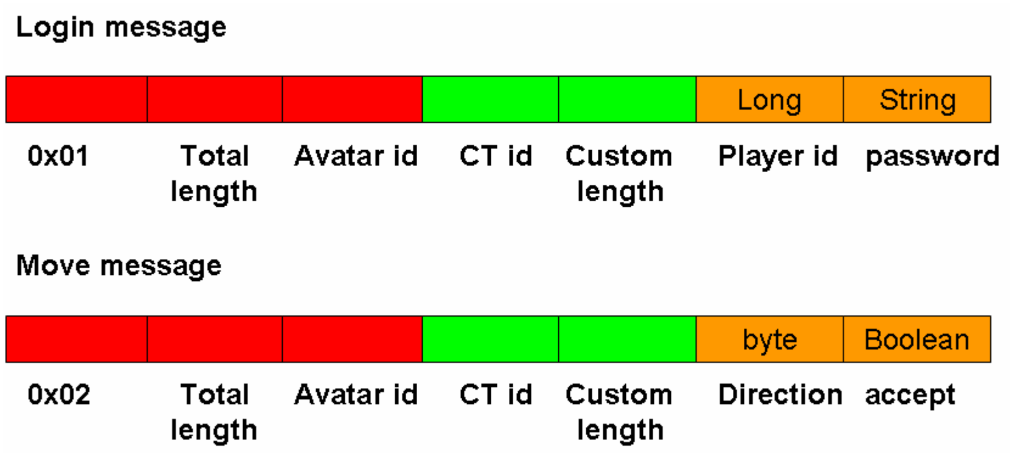
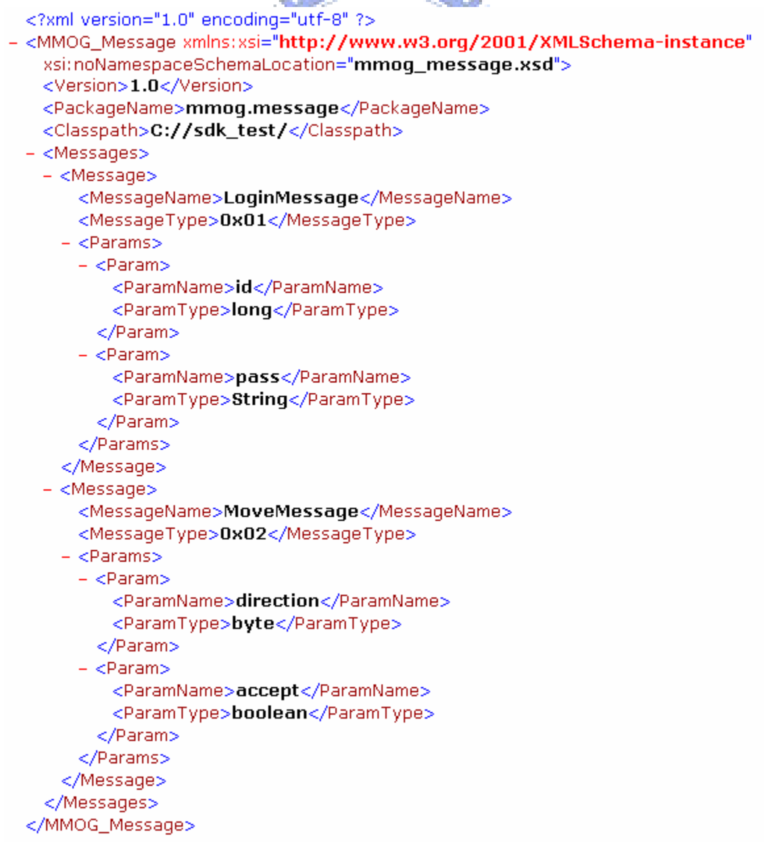


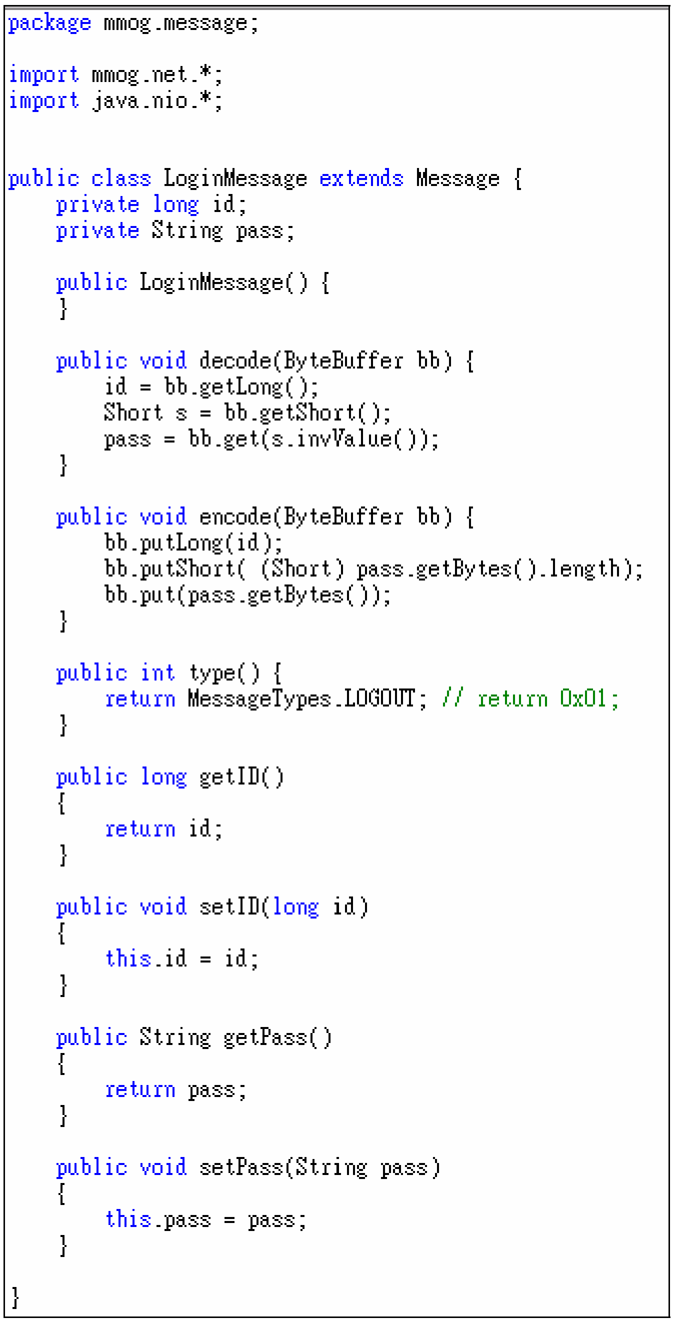
Figure 5-4. Examples of the Message Format

Because of the total length, avatar id, ct id, custom length attributes' value are assigned dynamically during the message sending process, so we will not necessary to describe them. We must describe the others, include the message type, and custom attributes and we also describe the message version number, package name and the class path. *Figure 5-5* is a MMOG message description document; it describes the login and move message protocol. And we defined an xml schema document to verify it; the xml schema document is show in the Appendix B. After defined the MMOG script document, the next step is to design the code generation engine. The code generation engine will generate the code according to the attributes' value which we described in the script document. Therefore, we analyze the code which we want to generate to induce some regular and relationship between these codes.



```
  <?xml version="1.0" encoding="utf-8" ?>
- <MMOG_Message xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="mmog_message.xsd">
    <Version>1.0</Version>
    <PackageName>mmog.message</PackageName>
    <Classpath>C://sdk_test/</Classpath>
  - <Messages>
    - <Message>
        <MessageName>LoginMessage</MessageName>
        <MessageType>0x01</MessageType>
      - <Params>
        - <Param>
            <ParamName>id</ParamName>
            <ParamType>long</ParamType>
          </Param>
        - <Param>
            <ParamName>pass</ParamName>
            <ParamType>String</ParamType>
          </Param>
        </Params>
      </Message>
    - <Message>
        <MessageName>MoveMessage</MessageName>
        <MessageType>0x02</MessageType>
      - <Params>
        - <Param>
            <ParamName>direction</ParamName>
            <ParamType>byte</ParamType>
          </Param>
        - <Param>
            <ParamName>accept</ParamName>
            <ParamType>boolean</ParamType>
          </Param>
        </Params>
      </Message>
    </Messages>
  </MMOG_Message>
```

Figure 5-5. MMOG message script document



```
package mmog.message;

import mmog.net.*;
import java.nio.*;


public class LoginMessage extends Message {
    private long id;
    private String pass;

    public LoginMessage() {
    }

    public void decode(ByteBuffer bb) {
        id = bb.getLong();
        Short s = bb.getShort();
        pass = bb.get(s.invValue());
    }

    public void encode(ByteBuffer bb) {
        bb.putLong(id);
        bb.putShort( (Short) pass.getBytes().length);
        bb.put(pass.getBytes());
    }

    public int type() {
        return MessageTypes.LOGOUT; // return 0x01;
    }

    public long getID()
    {
        return id;
    }

    public void setID(long id)
    {
        this.id = id;
    }

    public String getPass()
    {
        return pass;
    }

    public void setPass(String pass)
    {
        this.pass = pass;
    }

}
```

Figure 5-6. The Code of MoveMessage and LoginMessage

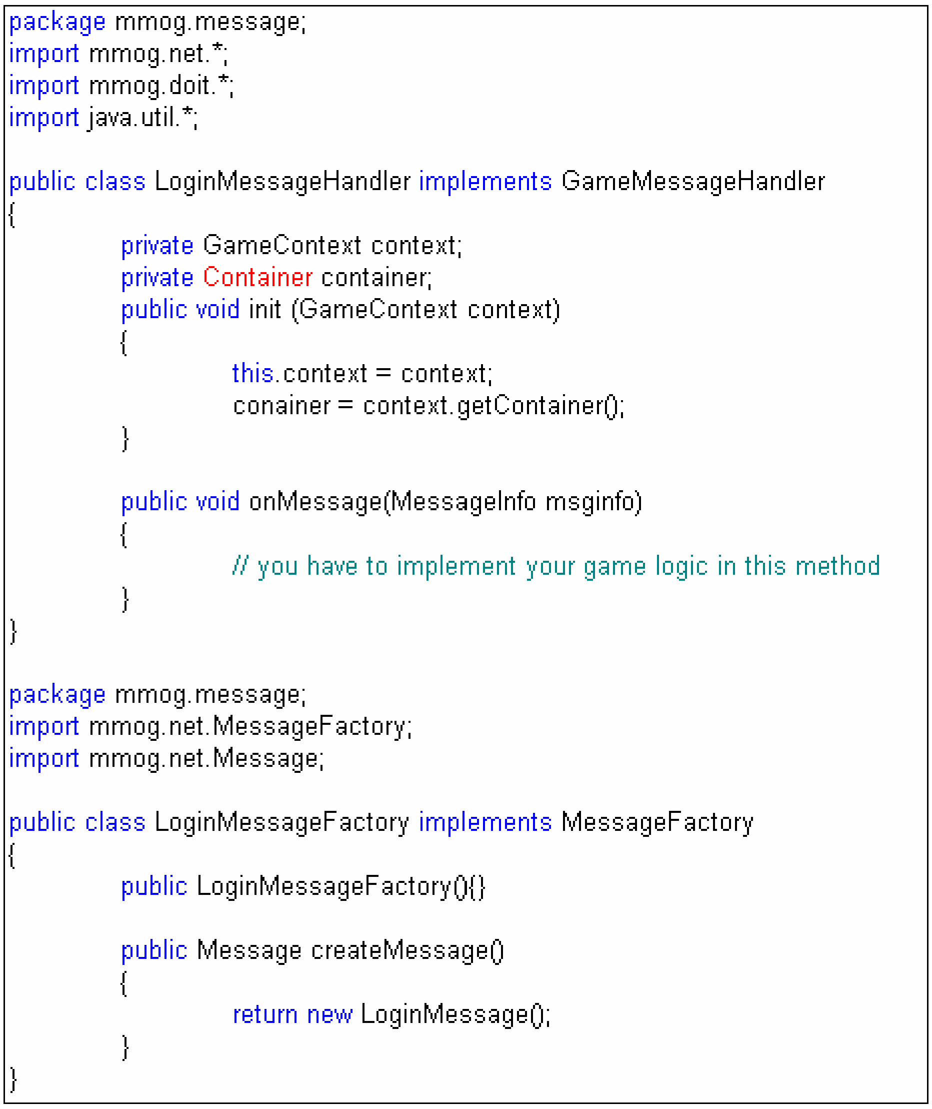
Figure 5-6 shows the code of the login message. It includes the message name and attributes' name and type and the message type value which all described in the MMOG script document. Because each message class will have encode and decode method, and the method logic operation will be different according to the attributes included by the message. The decode method function is to put the attributes' value from the byte buffer, and the encode method is doing the inverse work. Therefore, we must insert the code into the method according to the type of the attributes. In the decode method, we must use the JAVA byte buffer API to get the values from the byte buffer, and the sequence which we get the attributes must follow the sequence of the attributes which we described in the MMOG script document. And the encode method is also. For example, we described the login message in the Figure 5-6, it defined the login message have two attributes; id and password, and they will have the same order in the byte buffer.

Therefore, in the decode method, we assign direction attribute value by using the get() method to get a byte value from the byte buffer and then we use some code to get the accept attribute value. Because the JAVA byte buffer API doesn’t provide the method to get or put the Boolean and string value, we use a shot value to replace the Boolean value, and we use a short value and a byte array to replace the string. In the decode method, we use getShort() method to get a short value from the byte buffer and to differentiate the value. If it is equal to zero then it represent the Boolean value false, if it is equal to one then it represent the Boolean value true. If the attribute type is the string, then we use a short value to record the length of the string and use the bytes to store the string. By the same way, in the encode method we also need to use some JAVA byte buffer API to put the value into the byte buffer according to the type and sequence of the attributes.

The last part of the program is the attributes composed with the get and set methods. Each attributes in the message will accompany a get and a set method. We must insert these methods according to the attributes’ name and type. For example, in the login message, the attribute id will have the getID() and the setID() methods.

Besides the message code, we will also generate the message factory and message handler code. The code of the login message factory and handler is show in the ***Figure 5-7***. They are also follow a fixed format and the only thing that different is the class name. We just only insert the message name into the program where the red block area. Therefore, we can generate these codes fast without do any logic determine. After generated the code, the last work is to generate the vwlogic properties file. The game server will load the message handlers dynamically when system startup according to the description of the vwlogic properties file. The vwlogic properties file will store the entire handler and factories name and class path, and the record format is show in the next page. Each line represents a message handler or factory and it distributes the class path and message type:

```
1=cis.game.common.message.LoginMessageFactory/0x01
2=cis.game.server.handler.LoginMessageHandler/0x01
1=cis.game.common.message.MoveMessageFactory/0x02
2=cis.game.server.handler.MoveMessageHandler/0x02
```



```
package mmog.message;
import mmog.net.*;
import mmog.doit.*;
import java.util.*;

public class LoginMessageHandler implements GameMessageHandler
{
            private GameContext context;
            private Container container;
            public void init (GameContext context)
            {
                        this.context = context;
                        conainer = context.getContainer();
            }

            public void onMessage(MessageInfo msginfo)
            {
                        // you have to implement your game logic in this method
            }
}

package mmog.message;
import mmog.net.MessageFactory;
import mmog.net.Message;

public class LoginMessageFactory implements MessageFactory
{
            public LoginMessageFactory(){}

            public Message createMessage()
            {
                        return new LoginMessage();
            }
}
```

Figure 5-7. The Code of LoginMessageHandler and LoginMessageFactory

5.2.2. Management function code generation

At the game server layer, the architecture and work flow were shown in the Figure 4-18. We designed a management plug-in program named MbeanLoader. This program is designed follow the DOIT platform plug-in module and will be executed when the server is startup. It will do some works; it will startup an Mbean server and also creates a Serveranager Mbean which implemented by DOIT platform server management API, then it will register this Mbean to the Mbean server and connect to the central management server and make the register mechanism. Therefore, there are some programs we need to development includes the Serveranager Mbean and MbeanLoader.

In the DOIT platform, the server provides some management API. We use these API to implement the Serveranager Mbean. Therefore, we defined a ServerManagement MBean interface follow the JMX definition [45]. The interface defined the server state get and set debug and remote register invoke methods. The Serveranager Mbean will implement this interface and will be registered to the Mbean server.

```
public interface ServerManagerMBean {
    public Integer getregioncount();
    public void setregionnumber(Integer regionnumber);
    public String getregionname();
    public void setdebug(Integer debug);
    public void Debug();
    public Integer getAVAcount();
    public Integer getNPCcount();
    public Integer getRegionID();
    public void resetRegion();
    public void register();
}
```

There is a part of the ServerManager code in the below. The getregioncount() method is to get the server region count. The method content is to use the MBeanLoader object reference to lookup the RegionManager object reference, and then use the RegionManager's getRegionCount() method to get the count. So, each of the method in the ServerManager will be implemented by the same way.

```
public class ServerManager implements ServerManagerMBean {
    public MBeanLoader mbeanloader;
    public Integer regioncount;
    public ServerManager() {
    }
    public Integer getregioncount() {
        RegionManager a = (RegionManager) mbeanloader.context.
                          lookupComponent(
                RegionManager.COMPONENT_NAME);
        return new Integer(a.getRegionCount());
    }
}
```

After implemented the Serveranager Mbean, we must implement the MbeanLoader program. Because it is a plug-in for the DOIT platform, so we must development it according to the platform plug-in definition. The DOIT platform defined a plug-in interface; each plug-in program will implement this interface, and they will get the server component manager object reference. By this object reference, the plug-in program can get the server state or call some component method. Therefore, the entire MbeanLoader code can be divided into four parts. First is the platform defined part, this part is composed by a init(ServerContext context, Hashtable properties) method. This method will be invoked by the game server when this plug-in program was start. The game server will send the ServerContext object reference and a Hashtable reference to the method, and then the MbeanLoader will get these object reference.

Here is the init() method content:

```
public void init(ServerContext context, Hashtable properties) throws
MMOGComponentException {
    this.properties = properties;
    this.context=context;
}
```

The next part is to startup an Mbean server. During this process, we must assign a port number to the Mbean server, and also create a JMXConnectorServer for remote connection:

```
LocateRegistry.createRegistry(port);
MBeanServer mbs = MBeanServerFactory.createMBeanServer();
JMXServiceURL url = new
JMXServiceURL("service:jmx:rmi:///jndi/rmi://localhost:"+port+"/serve
r");
JMXConnectorServer cs
=JMXConnectorServerFactory.newJMXConnectorServer(url, null, mbs);
cs.start();
```

The third part is to create a ServerManager Mbean, and register it to the Mbean server:

```
sm = new ServerManager();
sm.mbeanloader = this;
mbs.registerMBean( sm,new
ObjectName( "MBeans:type=mmog_management.ServerManager" ) );
```

The last part is to connect to the central management server and make a register procedure. We will use standard JMXConnector to connect to the central server, and then set the local server name, port and IP to it. At last, we will invoke the addserver method to create a server factory which represents this game server's remote control component.

5.3 Benefits

Below are presented some of the results after testing the development and management systems on the DOIT MMOG platform:

(1) Separating out the various tasks of the system administration, application programming and communication configuration.

In a large scale distributed computing environment, it is important to separate out the various tasks in the software engineering process. In the model presented here, the system administrators/analyzers can define only the interfaces necessary for management tasks. The programmers who employ the interface generated by the code generation engine do not need to know the detail of the communication technology. The model provides a scheme of isolation from implementation and simple and easily understood representation. Essentially the idea is based on the my' previous work: the Ghostwriter engine [50] , which provided a lower technical learning curve, help for concentrating on system design, easily reusable components, and easily integrated applications.

(2) The code generation model helping programmers build a manageable object rapidly and easily.

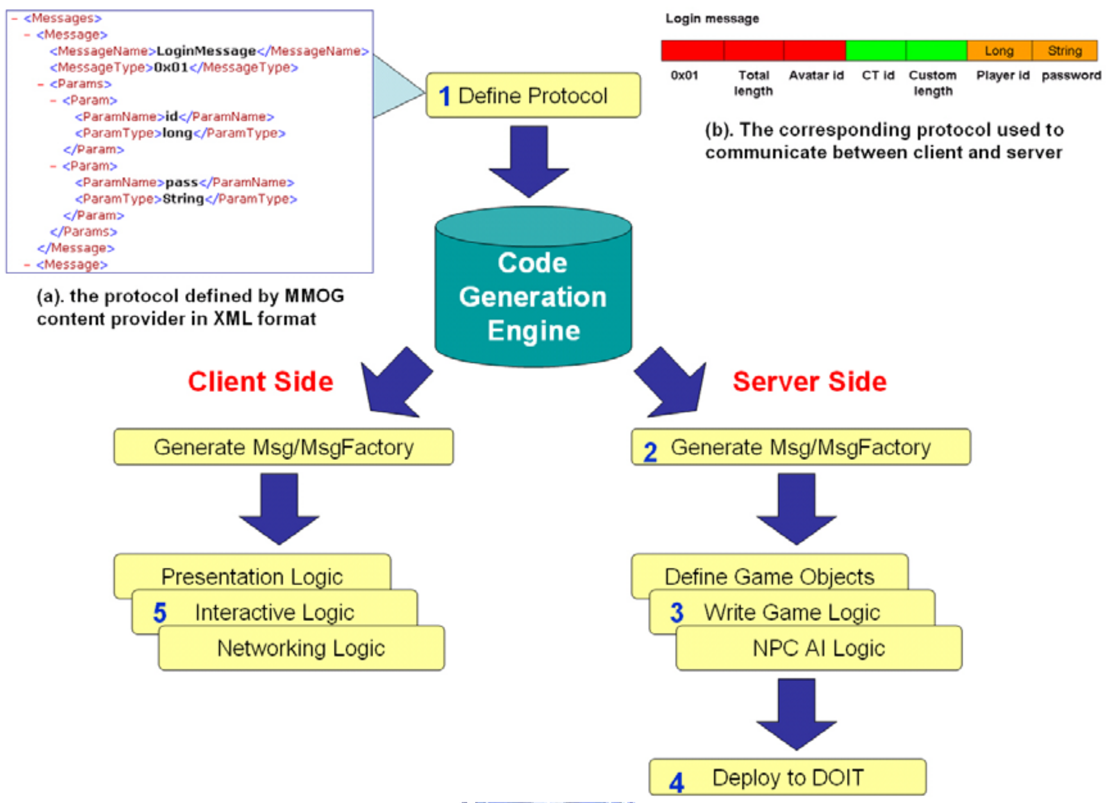
Using XML as the MMOG content description language offers certain advantage. First, the game content designer has simply to learn the description syntax, and can then use any text editing software to edit a MMOG description document. Second, the Unicode encoding of XML allows the use of any language code for editing the document. Third, much of the data in the MMOG is structured data and can be suitably described by XML. On completion of the editing, an XML schema file can verify if the description document is valid while the existing XML parser can analyze the content and process the data.

(3) An efficient and customizable message-based network engine, which reduces complexity for application programmers designing the protocol.

A main goal of DoIT's is "simple is better" [35]. Our message-based network engine helps developers focus on a simple and impact protocol description. Developers need only define the protocol and have no worries over network programming. We also introduce a code generator model (shown in Figure 5-8) to help the developer generate appropriate content protocol. First, the content developer describes the message format in an XML file. Next, our engine parses the protocol description. Third, the engine generates corresponding message factory and handler codes for both clients and servers. Fourth, the developer implements the detailed content code in handlers (for example, how to handle a PLAYER_MOVE message sent by a client). Finally, the codes can be deployed by both client and server.

An example of protocol defined in this platform is shown as Fig 5-8. MMOG developers define the detailed protocol in XML format. Figure 5-8(a) shows that the content protocol number 1 is "*LoginMessage*", the parameters include a long typed id and a string type password. Figure 5-8(b) is the real protocol generated by our code generator. In our opinion, defining protocol precisely makes the generated protocol handlers more compact, and thus increases the network transmission performance without too much of the overhead that is encountered in CORBA.

Security issues and content updating makes changes of content protocols are commonly encountered during the development and operation of MMOGs, so the network engine should be flexible enough to meet such changes easily. In our DoIT platform, the content-oriented protocols are modeled as a set of message field offsets. Therefore, a random shuffling of the field offsets by the code generator engine increases the protection against hacking message-oriented protocols and faking messages. Moreover, the combination of messaging protocol and encryption algorithms, such as SSL, can greatly assist the vendor against hackers.



```
- <Messages>
  - <Message>
      <MessageName>LoginMessage</MessageName>
      <MessageType>0x01</MessageType>
    - <Params>
      - <Param>
          <ParamName>id</ParamName>
          <ParamType>long</ParamType>
        </Param>
      - <Param>
          <ParamName>pass</ParamName>
          <ParamType>String</ParamType>
        </Param>
      </Params>
    </Message>
  - <Message>
```

Figure 5-8. Code Generator Model for DoIT platform content developers

(4) A lightweight real-time virtual world logic adapter makes development, deployment, and change of content easier.

Messages sent to virtual world logics (VWLogic, i.e., game rules) are processed by this component. A control message received by the network engine is asynchronously put into the Virtual World logic adapter. The MMOG developer creates the VWLogic, which can be plugged into or removed from the VW logic adapter at runtime, and this makes any changes simple. This feature allows developers to change the class, for instance by replacing bugged VWLogic by a bug-free class, even when the whole MMOG application is running. Asynchronous adapter design makes 'Hot swap' feature possible to DoIT platform, and more maintainable once the service is online.

(5) The n-tiers hierarchy model providing a scalable management architecture. While most mechanisms or architecture focus on integrating certain network devices/services by a unified protocol or presentation (with such as XML, SNMP, GMPLS technologies), it is very important to present a more scalable and flexible architecture/framework. The framework here can scale from small objects to a large distributed computing environment and all the services can be dynamically loaded, unloaded, or updated in the management infrastructure. In the development of the MMOG management system, JMX is extended to solve the network communication problems and an Mbean server is used to implement the n-tier management server architecture. The workload of the MMOG management system development is reduced and all the resources needing to be managed are packaged into the Mbean. Where there are any new resources that must be managed, the management interfaces are simply added into the Mbean and registered to the delegation server. The manager can connect to the management server by an http connection or other JMX support connection protocols. The method should make the management task more flexible convenient.

(6) Leveraging future management concepts. While SNMP and MUWS focus on packaging legacy components into a specified protocol (SNMP and Web Services, respectively), the approach in this paper is to focus on making components manageable in n-tier architecture. As protocol transformation is done at the delegation layer, manageable objects could be retrieved by any possible future technologies..