Appendix A. How to develop content on DoIT platform

A.1 Packages in DoIT platform

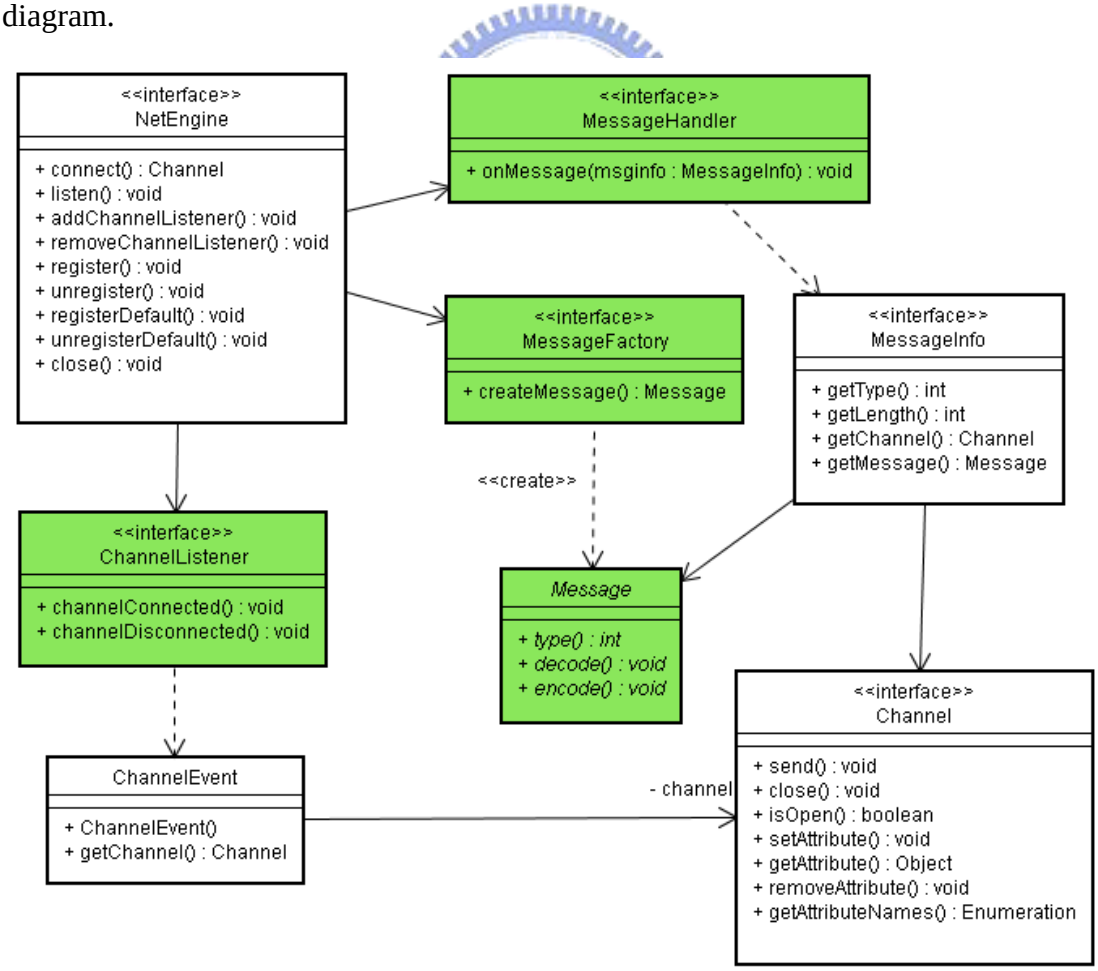
Description of DoIT middleware library:

- mmog.net:
  Abstract interface of NetEngine. It only define the NetEngine interfaces, without the implementation.
- mmog.net.tcp
  Implementation of NetEngine with multiplexing-I/O. It is implemented by non-blocking I/O and selector within java.nio packages.
- mmog.net.ttcp
  Implementationf of NetEngine with multi-thread. Every connection creates a thread.
- mmog.conf:
  this package will analysis the configuration file " <DOIOT>/game/mmog.xml", and provide the information in mmog.xml to other components.
- mmog.gateway:
  The implementation of Gateway components.
- mmog.server:
  The implementation of Server components.
- mmog.server.net:
  The implementation of network component in Server side.
- mmog.server.region:
  The region implementation. It consists most of the implementation of DoIT API.
- mmog.server.rm:
  The implementation of Region Migration.
- mmog.coordinator
  The implementation of Coordinator.
- mmog.message
  It consists most of the necessary message and message factory in the system.
- mmog.doit
  DOIT API interfaces.
- mmog.util
  Some utility class in DoIT platform.
- mmog.util.thread

The implementation of Thread pool.

Dependent third party library
Log4j – http://jakarta.apache.org/log4j
Concurrent Utility by Doug Lea –
http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html
JUnit – http://www.junit.org/index.htm

NetEngine in DoIT Platform
As in chapter 4.3.1 Network Engine, NetEngine is a message-oriented messaging library. Each data transmission is based on message. Every custom message must inheritance mmog.net.Message to transfer by our NetEngine.
In mmog.net, we provide the abstract interfaces of NetEngine, and two implementations. The implementations are mmo.net.tcp (multiplexing I/O) and mmo.net.ttcp (multithread version). The following figure is the UML diagrams of NetEngine. The content developers must implement or inherit the green part of the diagram.



- NetEngine:

The façade interface of NetEngine. It provides the registration method of message handler and listener. NetEngine also provides two methods for active connection (.connect()) and passive connection (.listen()).

- Channel:
  Channel is a virtual connection created by invoking NetEngine.connect() or ChannelListener.channelConnected(). Content developer can invoke send(Message) of Channel to transfer the message, and invoke xxxAttribute() to set attributes on the Channel.
- Message:
  The basic class of all message in DoIT environment. All of the custom message type must inherit from this Class. 3 methods need to be overridden in custom message. First is type(), it will return the message type (from 0 to 65535). The others are decode() and encode(), it handles the read and write of the message content. The parameter is java.nio.ByteBuffer.
- MessageFactory
  It handles the creation of the messages. While we registers a MessageHandler, we must provide a MessageFactory for network engine at the same time.
- MessageHandler
  It consists the interface of how to process messages. Content developers must implement the MessageHandler to process the message. When a message is received by NetEngine, NetEngine will invoke onMessage() of Message Handler with a MessageInfo Object as parameter.
- MessageInfo
  MessageInfo interface defines what will be transfer to MessageHandler (type. Channel, length, message…ect.)
- ChannelListener
  It defines the interface that will handle the event when channel creation of destroy.
- ChannelEvent
  It defines the event object of Channel event

A.2 How to Use NetEngine

The 3 Steps of using NetEngine:

1. Define Message
2. Create Connection
3. Send Message

Define Message

The basic class of all message in DoIT environment. All of the custom message type must inherit from this Class. 3 methods need to be overridden in custom message. First is type(), it will return the message type (from 0 to 65535). The others are decode() and encode(), it handles the read and write of the message content. The parameter is java.nio.ByteBuffer. Here is a sample of simple ADD message. In here, we define a message named AddMessage. Add Message will receive two parameters that need to be added together, and type is set as 1

```
package add;
import mmog.net.Message;
import java.nio.*;
public class AddMessage extends Message{
    public static final int MESSAGE_TYPE = 1;
    private int x;
    private int y;

    public void decode(ByteBuffer bb){
      x = bb.getInt();
      y = bb.getInt();
    }
    public void encode(ByteBuffer bb){
      bb.putInt(x);
      bb.putInt(y);
    }
    public int type(){
      return MESSAGE_TYPE;
    }
    public void setX(int x){
      this.x = x;
    }
    public int getX(){
      return x;
    }
    public void setY(int y){
     this.y = y;
    }
    public int getY(){
      return y;
    }
}
```

Create Connection

To create connection, we must implement the NetEngine (mmog.net.tcp.TCPNetEngine or mmog.net.ttcp.TTcpNetEngine). If the connection is at server side, we simply invoke the listen() method and provide a ChannelListener. If the connection is as client connection, we simply invoke the connection() method.

Server Side:

```
NetEngine engine = new mmog.net.tcp.TCPNetEngine();
InetSocketAddress sockaddr =
      new InetSocketAddress(13579);
engine.listen(sockaddr);
```

Client Side:

```
NetEngine engine = new mmog.net.tcp.TCPNetEngine();
InetSocketAddress sockaddr =
    new InetSocketAddress("localhost", 13579);
Channel channel = engine.connect(sockaddr);
```

Send / Receive Message

Send Message

Both client and server can send / receive message. Here is a sample, we create a AddMessage, set 2 parameters that are to be added, and then invoke channel.send(message) to send it to server side.

```
AddMessage msg = new AddMessage ();
msg.setX(3);
msg.setY(5);
channel.send(msg);
```

Receive Message

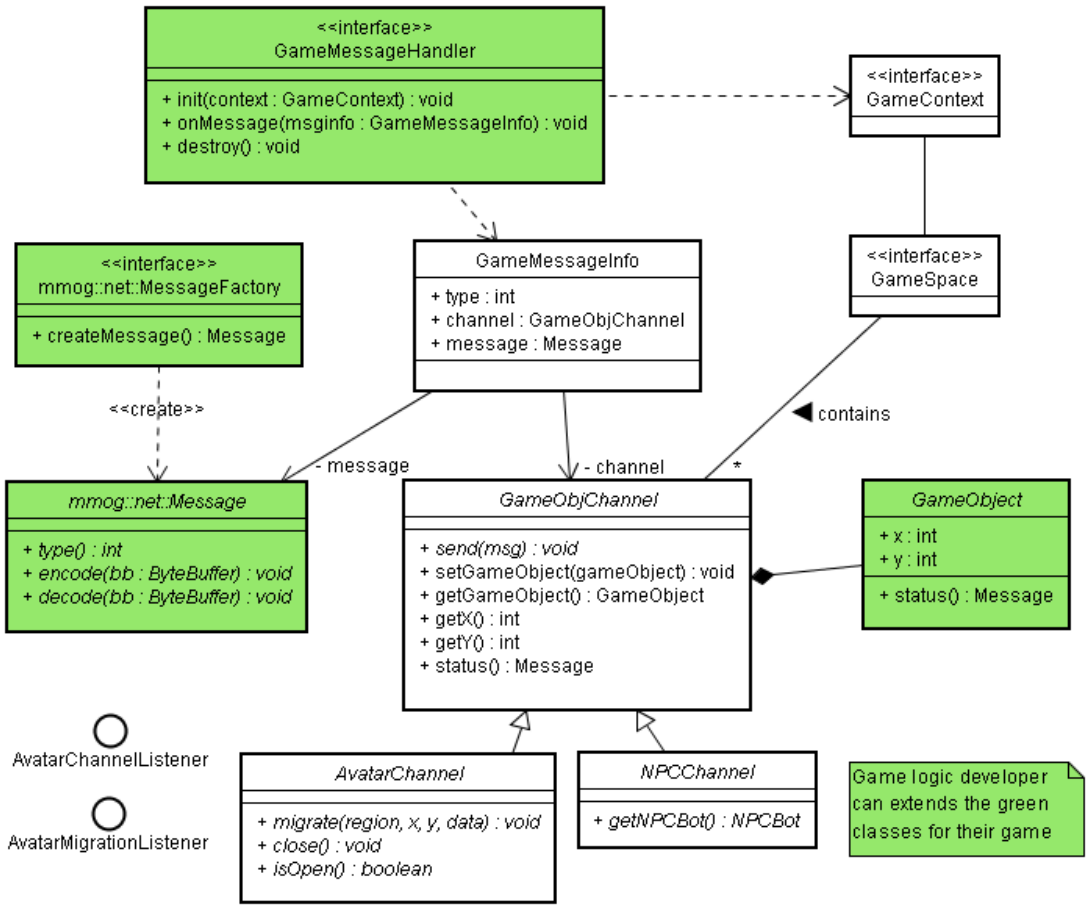
The first step is to register the message to NetEngine. This will notify the NetEngine, what message type should be process by what MessageHandler and MessageFactory. When the message is arrived, NetEngine will invoke corresponding MessageFactory to create the message by message type. Then, NetEngine will pass the created message to corresponding MessageHandler by invoking onMessage() of MessageHandler. Here is a sample:

```
//AddMessageFactory.java
package add;
import mmog.net.*;
public class AddMessageFactory implements MessageFactory{
    public Message createMessage(){
        return new AddMessage();
    }
```

```
}
//AddMessageHandler.java
package add;
import mmog.net.*;
public class AddMessageHandler implements MessageHandler{
    public void onMessage(MessageInfo msginfo) {
        AddMessage msg = (AddMessage)msginfo.getMessage();
        int x = msg.getX();
        int y = msg.getY();
        System.out.println(x + "+" + y + "=" + (x+y));
    }
}
//Main
engine.register(1, new AddMessageFactory(),new AddMessageHandler());
```

A.3 DOIT API

DoIT API is for content developers. API also includes message, message factory, message handler, channel, and listener. The following is the UML diagram of DoIT API.



- **GameMessageHandler**:
  The same as MessageHandler of NetEngine, GameMessageHandler also contains onMessage() method, however, the parameter is GameMessageInfo. In addition, content developer have to implement init() and destroy() of GameMessageHandler. When GameMessageHandler is starting up, DoIT will invoke init() and give GameContext as parameter.
- **GameObjChannel**:
  Similar to Channel of NetEngine, GameObjChannel contains send(). However, GameObjChannel does not contain attributes method, it contains setGameObject() method. GameObjChannel handles the communication mechanism of all game objects. By using GameObjChannel, we can invoke setGameObject() to set the relationship of a game object. All getX() , getY(), status() will delegate to GameObject.getX(), GameObject.getY(), GameObject.status(), respectively.
- **GameObject:**
  Each game object in DoIT must encapsulate into a class that is inherited from GameObject Class. GameObject contains the most fundamental data, such as x-axis and y-axis, and content developers can add new method in sub-class.

Developers can override status() method, this method will return a Message type object, that is responsible for the “update message” of current game object.

- **GameMessageInfo:**
  Similar to MessageInfo of NetEngine. However, it returns GameObjChannel, and it doesn’t have length.
- **GameContext:**
  It represents the context of game environment. Each Region contains a GameContext object. Developers can get GameContext from GameMessageHandler or init() from listeners. Developers can get all attributes of region and GameSpace object from GameContext.
- **GameSpace:**
  GameSpace provides the interfaces that developers can manipulate the virtual environment. GameSpace contains a 2-dimension of data structure; the size of this 2-dimension map is defined by mmog.xml configuration file. GameSpace provides add / remove / move / find operations, and include GameObjChannel element. Before assign a GameObjChannel into a GameSpace, we have to set the relationship between GameObjChannel and GameObject.
- **GameSpaceUtil:**
  Ultility of GameSpace. Developers can send message to certain channel within certain range by GameSpaceUtil. Also, it can get all of the update messages of GameObject within certain range by GameObjectChannel.status() method.
- **AvatarChannel:**
  It is a channel object that is represented Avatar. AvatarChannel can migrate by invoking migrate().
- **NPCChannel:**
  It is a channel object that is represented NPC. Developers can get information of NPC object by invoking getNPCConfig().
- **NPCBot:**
  This Object resides NPC AI program. Programmers have to inherit NPCBot to develop their own NPC object. Programmers have to override : onInit(), onTimeOut(), onUpdate(), that handles initialize, timer trigger, notify when update message is received, respectively. Programmer can invoke sendCommand() to send message into the virtual environment.
- **NPCConfig:**
  This class stores some of the initial status of NPC object.

A.4 The Complete Development Flow in DoIT:

Development flow:

1. Define Message
2. Program Game Logic
3. Compile
4. Deploy
5. Execute

Define Message

As describe in NetEngine part.

Program Game Logic

Game Logic is included in GameMessageHandler. By leveraging DoIT API, content developer can program online game by GameContext and GameSpace easier. GameLogic also can reside in AvatarChannellistener and AvatarMigrationListener to handle the logic when avatar login, migrate in or migrate out.

Compile

Due to game logic always contain DoIT API, you have to place corresponding DoIT library in classpath when you compile your program. For example:

```
javac –classpath lib/mmog2.jar *.java
```

Deploy

The directory of deployment is as following:

/game/mmog.xml	Define the components and attributes of server, gateway, and coordinator. It also contains the information about which region belongs to which sever.
/game/vwlogic.properties	Define the game logic class and corresponding message type.

/game/classes	Reside all of the compiled classes.
/game/lib	directory of 3rd party library
game/npc.properties (optional)	Define the configuration of NPC.

Here are the sample of each properties file:

mmog.xml

```
<?xml version="1.0"?>
<!DOCTYPE mmog SYSTEM "mmog.dtd">
<mmog>
    <components>
        <!--  set up servername, must at least have one -->
        <server name="server1" host="localhost" port="8765"/>
        <server name="server2" host="localhost" port="8764"/>
        <!-- set up gateway name, must at least have one -->
        <gateway name="gateway1" host="localhost" port="5678"/>
        <gateway name="gateway2" host="localhost" port="5679"/>
        <!-- set up servername, none is ok -->
        <coordinator host="localhost" port="8778"/>
    </components>
    <maps>
        <map name="map0">
            <region name="regionlogin" x1="0" y1="0" x2="0" y2="0"
login="true"/>
        </map>
        <map name="map1">
            <region name="region1" x1="0" y1="0" x2="100" y2="100"/>
            <region name="region2" x1="100" y1="0" x2="200" y2="100"/>
            <region name="region3" x1="0" y1="100" x2="100" y2="200"/>
            <region   name="region4"   x1="100"   y1="100"   x2="200"
y2="200"/>
        </map>
    </maps>
    <assignments>
        <!—each region must reside in one server-->
        <assignment region-ref="regionlogin" server-ref="server1" />
```

```
        <assignment region-ref="region1" server-ref="server1" />
        <assignment region-ref="region2" server-ref="server1" />
        <assignment region-ref="region3" server-ref="server2" />
        <assignment region-ref="region4" server-ref="server2" />
    </assignments>
</mmog>
```

vwlogic.properties

```
# 1=MessageFactory
# 2=MessageHandler
# 3=AvatarChannelListener
# 5=AvatarMigrationListener
1=cis.game.common.message.LoginMessageFactory/0x01
2=cis.game.server.handler.LoginMessageHandler/0x01
1=cis.game.common.message.MoveMessageFactory/0x03
2=cis.game.server.handler.MoveMessageHandler/0x03
1=cis.game.common.message.NpcBornMessageFactory/0x06
2=cis.game.server.handler.NpcBornMessageHandler/0x06
1=cis.game.common.message.LogoutMessageFactory/0xFF
2=cis.game.server.handler.LogoutMessageHandler/0xFF
3=cis.game.server.listener.MyAvatarListener/0x00
5=cis.game.server.listener.MyMigrationListener/0x00
```

npc.properties

```
mmog.npc.good=cis.game.server.npc.GoodNPCBot
mmog.npc.bad=cis.game.server.npc.BadNPCBot
```

Execute

We have to start up the server components in this order:

Coordinator → Server* → Gateway* → Client *: all of the same componets.

Execution command:

Coordinator:

java -DMMOG_HOME=<DOIT_HOME> -cp <Classpath>
mmog.coordinator.Coordinator

Server:

java -DMMOG_HOME=<DOIT_HOME> -cp <Classpath>
mmog.server.GameServer <server-name>

Gateway:

java -DMMOG_HOME=<DOIT_HOME> -cp <Classpath>
mmog.gateway.Gateway <gateway-name>

DOIT_HOME is root directory of DOIT platform. Classpath must include all of the path of library (mmog2.jar, concurrent.jar, log4j.jar). Server-name is defined in mmog.xml.

A.5 Programming the Client Side

If client is developed by Java language, it is full compatible with our NetEngine packages. The message define of server side can also reuse in client side. If client is developed by C++ language, we also provide a C++ version NetEngine. If client is developed by other languages, it is still easy to communicate with DoIT server, just follow the same protocol that is defined in server side.

A.6 How to develop your own Plug-ins

As in chapter 4.3.5, DoIT is a extensible framework by developing your own plugins at Server Scope and Region Scope. Due to the component-based framework of DoIT platform, your own plugins must inherit mmog.server.ServerScopeComponent or mmog.server.RegionScopeComponent according to the function you want to develop. Plugins in DoIT are just like common components in DoIT environment, however, you have to follow a specific process to install them.

Process of developing a plugins

1. Write your own Plugins program
2. Compile and package it into jar file
3. Deploy the plugins

Write your own Plugins program:

Your own plugins must inherit ServerScopeComponent or RegionScopeCoponent depends on its scope level. Here is a sample:

```
//MyPlugin.java
package hello;
import mmog.server.*;
```

```
import java.util.Hashtable;

public class MyPlugin extends ServerScopeComponent{
    public void init(ServerContext context,
                     Hashtable prop){
        System.out.println("hello");
        System.out.println("foo=" + prop.get("foo"));
        System.out.println("foo2=" + prop.get("foo2"));
    }
}
```

Compile and package it into jar file:

Compile your plugins with mmog2.jar within your classpath:

javac -cp mmog2.jar -d . MyPlugin.java

Package it into a jar file:

jar –cvf MyPlugin.jar hello/*

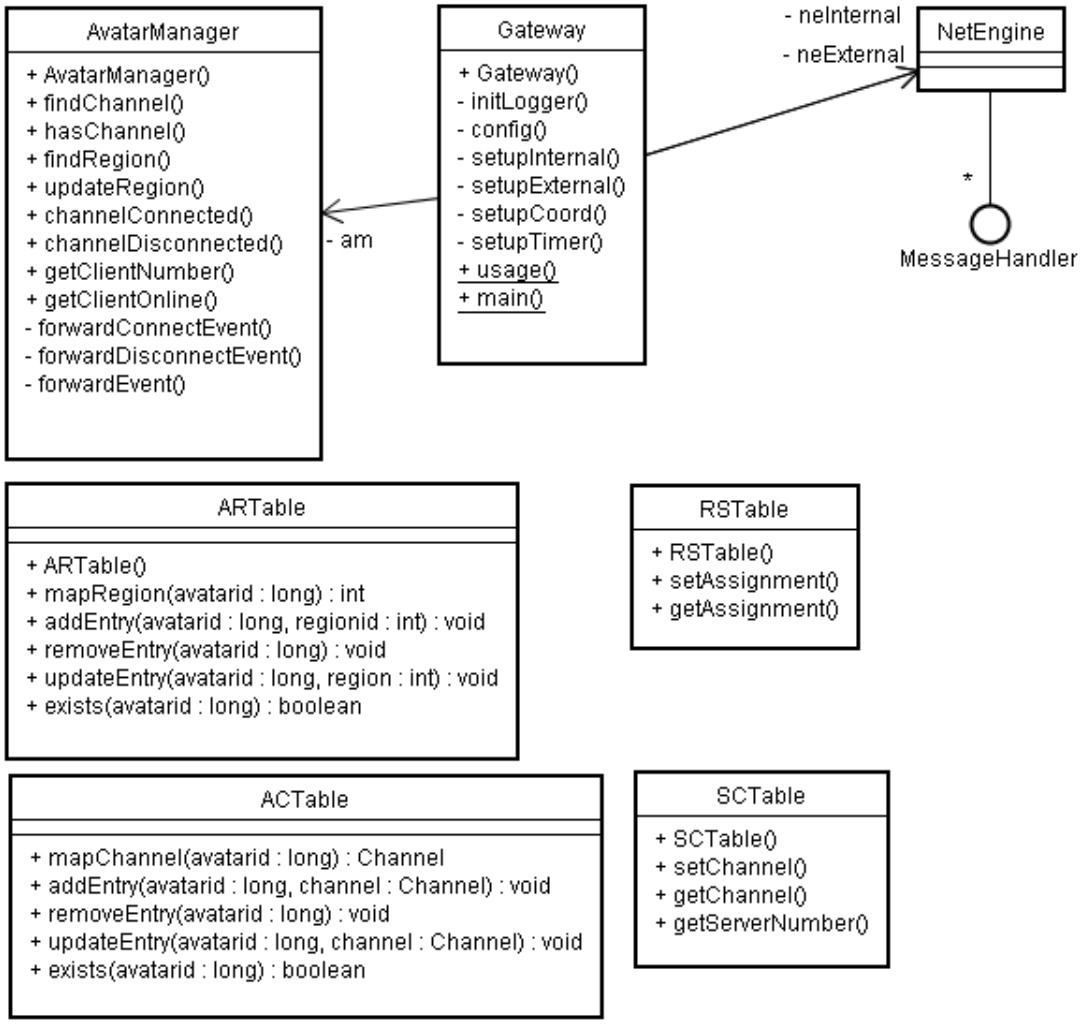
Deploy the plugins:

You have to deploy the jar file in the directory named : <DOIT_HOME>/plugin. At the same time, you have to write a properties file that has the same name with the jar file. This properties file will provide the information that is essential (Class name and initialization information) for DoIT platform. Here is a sample:

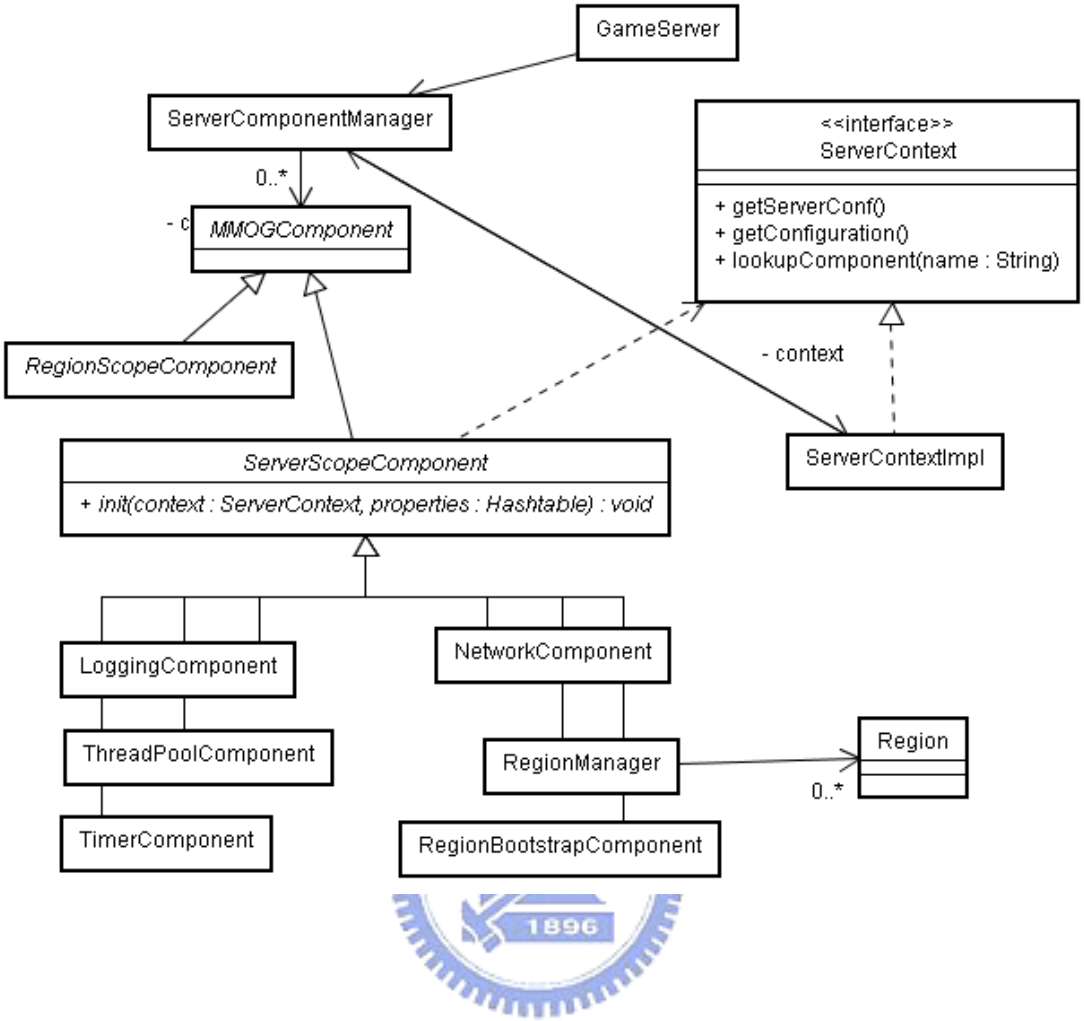
```
#MyPlugin.properties
# all of the “server plugin” must with prefix “mmog.plugin.server”
# all of the “region plugin” must with prefix “mmog.plugin.region”
# here we defind a plugin named “test”, and assign the class name
mmog.plugin.server.test = hello.MyPlugin
# here we give some parameters for the “test” plugin
mmog.plugin.server.test.foo = bar
mmog.plugin.server.test.foo2 = bar2
```

A.7 UML diagram of other Server Components

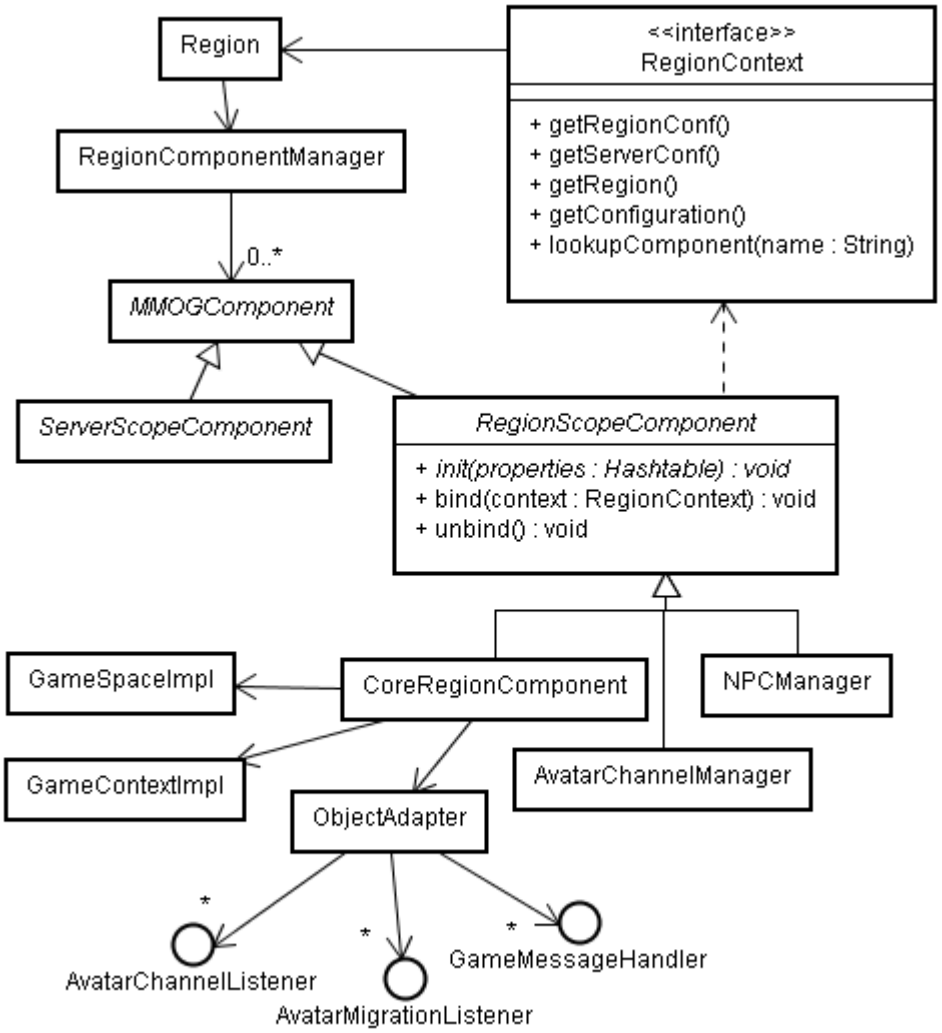
Gateway



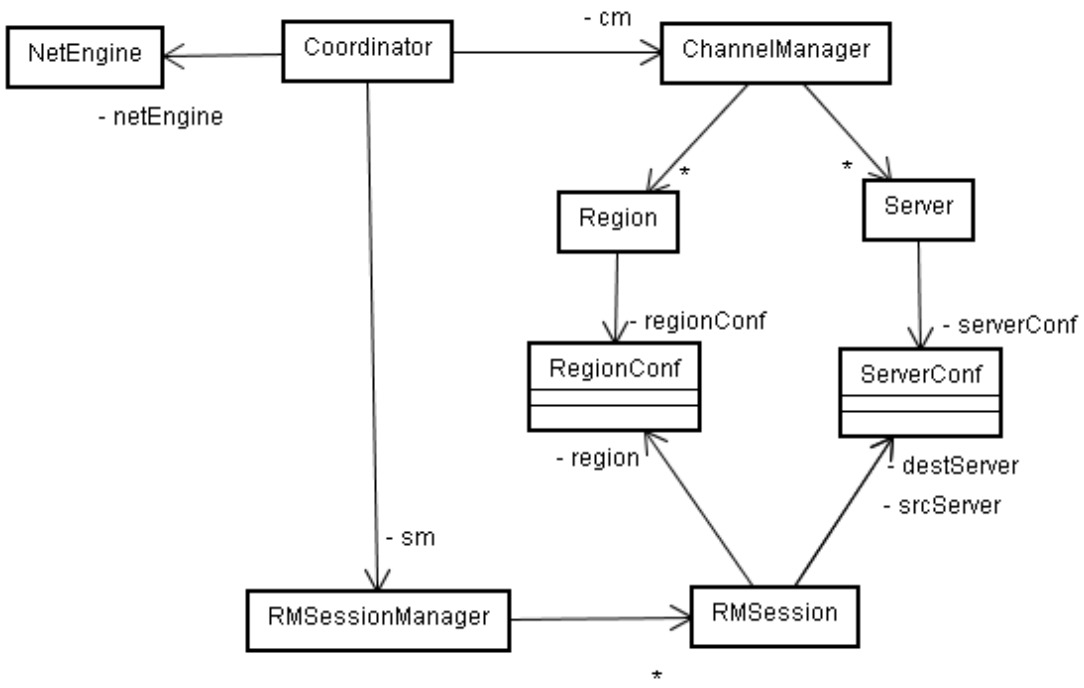
Server


GameServer
ServerComponentManager
0..*
- c
MMOGComponent
<<interface>>
ServerContext
+ getServerConf()
+ getConfiguration()
+ lookupComponent(name : String)
RegionScopeComponent
- context
ServerScopeComponent
+ init(context : ServerContext, properties : Hashtable) : void
ServerContextImpl
LoggingComponent
NetworkComponent
ThreadPoolComponent
RegionManager
Region
0..*
TimerComponent
RegionBootstrapComponent

Region


Region
<<interface>>
RegionContext
+ getRegionConf()
+ getServerConf()
+ getRegion()
+ getConfiguration()
+ lookupComponent(name : String)
RegionComponentManager
0..*
MMOGComponent
ServerScopeComponent
RegionScopeComponent
+ init(properties : Hashtable) : void
+ bind(context : RegionContext) : void
+ unbind() : void
GameSpaceImpl
CoreRegionComponent
NPCManager
GameContextImpl
AvatarChannelManager
ObjectAdapter
*
*
*
AvatarChannelListener
AvatarMigrationListener
GameMessageHandler

Coordinator



How to use Coordinator

When you start up the coordinator, a shell interface will pop up. You can perform "region migration" action with the shell interface. We provide 2 commands currently:

> status

Check current status of the region information

> migrate [region] [server]

migrate certain region to a certain server

Appendix B. DoIT Message definition XML schema

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="MMOG_Message" type="mmog_message_type"/>
  <xsd:complexType name="mmog_message_type">
 <xsd:element name="Version" type="xs:string"/>
 <xsd:element name="PackageName" type="xs:string"/>
 <xsd:element name="Classpath" type="xs:string"/>
 <xsd:element name="Messages" type="messages_type">
 </xsd:complexType>
  <xsd:complexType name="messages_type">
 <xsd:element name="Message" type="message_type">
 </xsd:complexType>
 <xsd:complexType name="message_type">
 <xsd:element name="MessageName" type="xs:string"/>
 <xsd:element name="MessageType" type="xs:string"/>
 <xsd:element name="Params" type="Params_type">
 </xsd:complexType>
 <xsd:complexType name="Params_type">
 <xsd:element name="Param" type="Param_type">
 </xsd:complexType>
 <xsd:complexType name="Param_type">
 <xsd:element name="ParamName" type="xs:string"/>
 <xsd:element name="ParamType" type="xs:string"/>
 </xsd:complexType>
</xsd:schema>
```