Game Environment for Command and Control Operations (GECCO)

System Description

Joel Brynielsson, Henrik Bäärnhielm, Andreas Enblom, Jing Fu Zi, Niklas Hallenfur, Karl Hasselström, Henrik Hägerström, Oskar Linde, Klas Wallenius and Jon Åslund

Department of Numerical Analysis and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm
Sweden
gecco@nada.kth.se

14th May 2001

Contents

1 Introduction

1.1 Background

GECCO is a platform for simulating strategic games. It is written in Java, and this document contains information about the Java implementation.

The three related documents *GECCO General Description*[2], *GECCO Developer's manual*[1] and *GECCO User's Manual*[3] offer additional information about the system, and should be read for general understanding of the functionallity of the platform.

1.2 The Java code

The system is written for Java 2 version 1.3, and consists of a number of packages that are, in some sense, unaware and independent of each other.

The system is documented in detail by comments in the source files. Every part of the system is commented for use with the Javadoc tool. The generated Javadoc web pages are accessible at

`http://www.nada.kth.se/projects/proj01/krigsspel/docs.html`

The following description will be kept on a high abstract level, mostly avoiding the implementation details, and we refer to the Javadoc documentation for the details.

1.3 Components of the system

The system consists of three large parts; the server (documented in section 2), the communication layer (section 3) and the client (section 4).

The server simulates the game, and is aware of the clients only as different roles, that may know and control some part of the game. The server handles all updates of the game (e.g. moving units) and sends messages about those updates to selected roles via the communication layer.

The communication layer handles the TCP/IP network communication between the server and the clients. This layer is aware of the clients, and how many of them that are connected to every role.

The client presents the game to the players, and send messages via the communication layer about control operations of the units.

2 The server

Our wish when we started working on the design of the platform was to make everything as modular and generic as possible, so that further development would be possible by just replacing certain modules. What we came up with is something that we are quite satisfied with.

2.1 Layers of the game

The game consists of two *layers*, not to confuse with the communication layer between the server and the clients. First, there is the *automaton layer*, also known as the map. Every pixel on the map is a finite state automaton, and thus the map can change, adding a special kind of dynamic behaviour to the game. The automaton layer has a discrete coordinate system, due to the use of raster images. Above the map is the *unit layer* where all units dwelve. The unit layer has a continous coordinate system (well, at least approximately), but they are always located withing the bounds of one specific automaton.

The two layers can interact in two ways. Either can the units manipulate the automatons when they execute *actions*, like "Set ground on fire", or the automatons can do the same thing on the units, by sending *events*. Naturally, there can also be interactions inside the layers, because units can affect other units with actions, like "Attack", and automatons can affect other nearby automatons. See the description of the Action Processor and the Queue Manager below.

As of genericity, both units and automatons can and should be subclassed by every real game on the platform, creating whole new rules of interaction between and inside the layers. Thus, we have achieved quite much modularity.

2.2 Server modules

The server consists of a number of more or less separable modules: The Action Processor, the Queue Mananager, the Visibility Manager, the Unit Manager, the map of automatons, the initialization and the Communication Interface. Here is a brief description of each module. For more detailed explanations, we refer to the Javadoc documentation.

- The initialization is the class that one uses to start the server process. Among all the things it does is to read all configuration files of the current game, the background image and all unit images, and allocate memory for the map. It initializes the automaton queue and the action queue, puts all units at their initial positions and starts all threads that will listen for incoming connections from clients. The initialization is contained in the package `server.startup`.

- The Unit Manager keeps track of all units in the game, maintaining data structures of them and providing various ways to get and set properties of units, like the commander and the observers. The Unit Manager is contained in the package `server.unitmanager`.

- The Visibility Manager keeps track of all areas of visibility of all units and ranges, providing methods that compute if a role can see a certain position of the map. To do that effeciently it maintains certain data structures which it keeps updated as the units move. The Visibility Manager is contained in the package `server.visibility`.

- The map of automatons is similar to the Unit Manager, but for the automatons instead. It maintains references to every automaton on the map, and provides methods for allocating and initializing the automatons, given a normal raster image, like a GIF file. There are also ways to access each automaton, get and set its properties, send events to it, and other things. The map of automatons is contained in the package `server.automaton`.

- The Communication Interface is the server's implementation of the Callback Interface of the communication layer, described more thoroughly in the next section. It consists of methods that take care of the messages from the clients, like initiating all actions the clients have executed, connecting a new role and similar things. The Communication Interface is contained in the package `server.core`.
- The Action Processor and The Queue Manager are the modules where everything happens (maybe in conjunction with the Communication Interface). The Action Processor maintains a queue of all ongoing actions in the game and from time to time take out one and calls its *checkpoint* routine. Then, depending on the return value, the action is reinserted in the queue, completed or aborted. The Action Processor is contained in the package `server.actionprocessor`.
- The Queue Manager is basically the same thing as the Action Processor, but for automatons and events to or from automatons. When an automaton is updated, it can choose to insert itself and possibly also its neighbours, thus spreading the update across the map. In this way the automaton layer can be very dynamic, and model widespread activities like fire, radioactivity, explosions and *Conway's Game of Life*. The Queue Manager is contained in the package `server.qmanager`.

The different modules are not as modular as the three different system components, but anyway we have achieved relatively much modularity. For example, if one wants to change some global behaviour about units, it should be almost only the Unit Manager, and its surrounding classes that must be changed, rather than many places all over the server.

3 The communication layer

3.1 Introduction

The communication is the glue between the server and the client. The main purpose of the communication part is to provide the possibility of having the server on one computer and a client on another.

The communication could have been integrated in the client and in the server, but that would mean that they had to handle the details of the TCP/IP communication all by themselves. By constructing communication interfaces for both the client and the server, the server and client communicate via these interfaces instead and just have to worry about their respective part.

3.2 The role model

Except for the raw details of TCP/IP, there are other things that the server and client doesn't need to know. In our current model, we have servers, roles and clients. The server only knows which roles are connected. There can be many clients for each role (you can start several clients for the role God's eye, which sees everything), but the server only sends messages to a role. It's the job of the communication part to send the message to the right clients.

The client on the other hand only knows what server it is attached to and have no idea that there may be other clients with the same role.

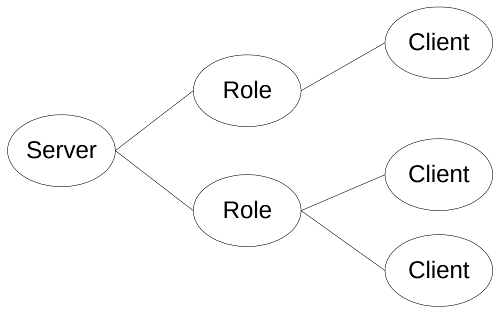

Figure 1: The Role Model.

3.3 The Interface from the outside

The messages that the client and server sends and receives are completely different (what the server sends, the client receives, and vice versa), but they both have interfaces with similar purpose. Both the client side and the server side have:

- Ambassador interface. A java interface, implemented in the communication part. The server and client uses this interface to speak with the communication part. The methods that the server and client can use to send messages are here.
- Callback interface. A java interface, implemented by both the server and client. The communication part uses this interface to speak with the client and the server when a new message has arrived.

The ambassador interface on the client side is called `Server` because it addresses the server, and the ambassador interface on the server side is likewise called `Client`.

The callback interface on client side is called `Client`, and the callback interface on the server side is called `Server`.

3.4 The Interface from the inside

The client and the server part of the communication have many similarities when it comes to handling the communication. Here follows the part that looks almost the same on both sides.

- Connection. Called `ClientConnection` on the server side, and `ServerConnection` on the client side. It is created by the ambassador and contains the sockets and data streams needed for network communication.

- DataInputExchangeHandler. Separate thread that is started on both sides when a client sends a startGame message. The thread waits for incoming messages, by a blocking read on the input stream of the connection. When a message is received, it calls the appropriate method in the callback interface.
- DataOutputExchangeHandler. A thread that is started together with the `DataInputExchangeHandler`. This thread waits for messages to appear in its message queue. Messages are put in the queue when the server or client calls its respective ambassador interface. When a message is in the queue, the `DataOutputExchangeHandler` sends it to the output stream of the connection.

3.4.1 The client

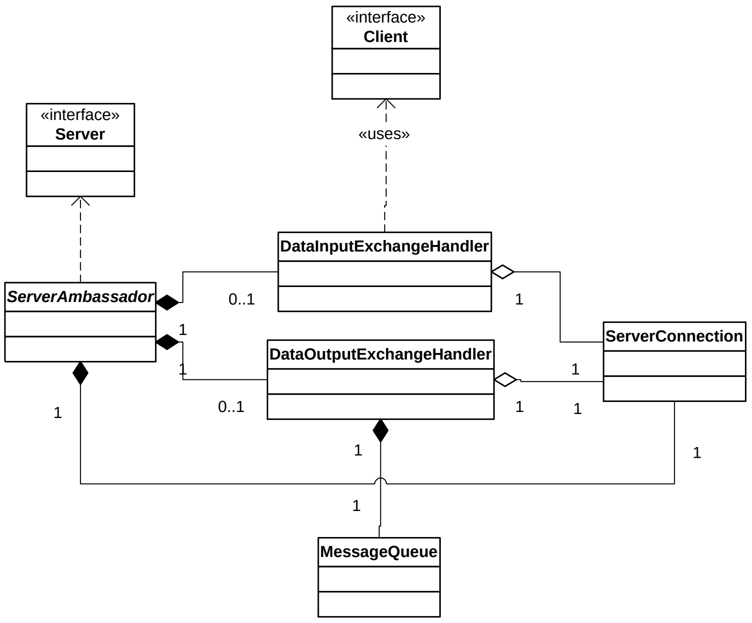

Figure 2: Client UML chart.

The client creates a `ServerAmbassador` and implements the interface `client.Client`. The `ServerAmbassador` creates a `ServerConnection`. When the client starts the game it creates a `DataInputExchangeHandler` which listens to incoming messages from the server and a `DataOutputExchangeHandler` which sends messages to the server. When creating the `DataOutputExchangeHandler`, the `ServerAmbassador` gives a `MessageQueue` as an argument. That is the queue where the newly created thread will look for messages to send.

3.4.2 The server

The server creates a `ClientAmbassador`, and implements the interface `server.core.Server`. When a `ClientAmbassador` is created, it takes a vector containing `RoleDefinition`

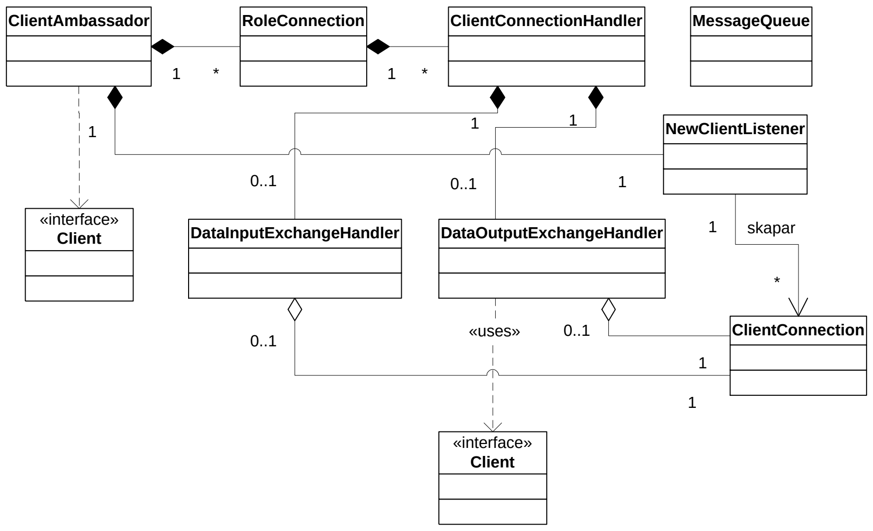

Figure 3: Server UML chart.

objects as an argument. For each `RoleDefinition`, a `RoleConnection` is created and put into a hashtable.

The constructor for `ClientAmbassador` also takes an int, representing a port number, as an argument. The `ServerAmbassador` creates a `NewClientListener`, that will listen for incoming connections on that port, and handle the initial handshaking with the clients.

When a client has successfully joined a game, the `NewClientListener` creates a `ClientConnection` and tells the `ClientAmbassador` to add that to the connected clients. `ClientAmbassador` calls add in the `RoleConnection` representing the client's role, and a new `ClientConnectionHandler` is then created by `RoleConnection`.

3.5 The way of a message

3.5.1 Starting the server

The server creates a `ClientAmbassador`. The constructor takes three arguments; a `Vector` containing RoleDefinition objects, an object that implements `server.core.Server`, and an int representing the port number, to which the server will listen for incoming connections. For each `RoleDefinition` in the `Vector`, a corresponding `RoleConnection` is created and put into a `HashTable` representing connections to all the roles. The `ClientAmbassador` creates a `NewClientListener`, that will listen for incoming connections on the specified port, and handle the initial handshaking with the clients.

The `ClientAmbassador` is now ready to start receiving messages from clients, and sending messages from the server, but since no clients are connected yet, nothing will happen if the server sends a message.

3.5.2 Starting and connecting the client

The client creates a ServerAmbassador, with an object implementing client.Client as argument. The ServerAmbassador in turn creates a ServerConnection.

When the client is ready to connect to the server, it calls the method ServerAmbassador.connect, which takes the server name as a String, and the port number as an int. The ServerAmbassador in turn calls the connect method in its ServerConnection object. On the server side, the NewClientListener accepts the incoming socket. If the connection was successful, the client gets the return value true, and is now able to join as a role by calling ServerAmbassador.joinAsRole.

But first the client might want to know what roles are available to join as, so it calls getAvailableRoles. The ServerAmbassador now writes the int MessageType.getAvailableRoles to the ServerConnection's DataOutputStream. On the server side, the NewClientListener reads that int, and asks the ClientAmbassador for the available roles by calling rolesLeft, and sends the roles as a UTF-string. The ClientAmbassador reads that String, and splits it up into the roles by separating on whitespaces (See StringTokenizer). The StringTokenizer is then returned to the client.

When the client is ready to join it calls joinAsRole with a specified role. The ServerAmbassador sends the message. NewClientListener checks if the role is available for joining and sends true back to the ServerAmbassador if it is.

Now the client wants to start the game, but it has to receive the map first so it calls getMap. The ServerAmbassador sends the message as per usual and receives the map from NewClientListener.

After receiving the map the client finally calls startGame. The ServerAmbassador sends the startGame message to the NewClientListener and then starts the threads DataOutputExchangeHandler and DataInputExchangeHandler. When the NewClientListener reads the startGame message it creates a new ClientConnection and tells the ClientAmbassador to add it. After creating the DataOutputExchangeHandler the ServerAmbassador puts a startGame message in DataOutputExchangeHandler's MessageQueue. The handshaking is now completed and the game can begin.

3.5.3 Messages sent during the game

After the server has received the startGame message it sends many refresh messages so that the client is up to date. This makes it possible to join a game any time.

We are now going to describe the path of a typical message that is sent from the server to the client. All messages are sent exactly the same way.

Assume that the server calls unitInvisible in ClientAmbassador to indicate that a unit no longer is visible to a role. The arguments are a role and a unit handle. The ClientAmbassador creates a UnitDeletedMessage and sends it to the RoleConnection specified by the role argument. RoleConnection in turn calls sendMessage for each ClientConnectionHandler connected to the role. The ClientConnectionHandler then calls addMessage in DataOutputExchangeHandler, which puts the message in the message queue. The DataOutputExchangeHandler discovers the message in the queue, and calls the corresponding sendUnitDeleted method. This method first writes the int MessageType.UnitDeleted, and then the handle of the unit which is no longer visible, to the DataOutputStream in

`ClientConnection`.

The `DataInputExchangeHandler` on the client side now discovers the int `MessageType.UnitDeleted` in the `DataInputStream` in `ServerConnection`. The method `getUnitDeleted` is called, and reads the unit's handle from the `DataInputStream`. Then the client is told about the deleted unit via the method `removePiece` in the callback interface `Client`.

Communicating the other way is handled in almost the same way. The client calls a method in `ServerAmbassador`, a message is created, and put in the message queue of `DataOutputExchangeHandler`. From here on, the message is passed in exactly the same way as in the example above.

4 The client

The client consist of the Java packages `client`, `client.animation`, `client.dialogs` and `client.infopanels`.

The package `client.dialogs` contains the dialogs of the game (such as a dialog for connecting to a game, and an error dialog), whereas `client.infopanels` contain panels where information about game units (or *pieces* as they will be referred to in this section) is displayed. The `client.animation` package supply support for animated pieces (such as explosions). These three packages are rather self-explainative and need no further comments, apart from the Javadoc documentation.

The `client` package (see figure 4) is the main client package. It contain most of the classes that control the client operations (i.e. displaying information to the user, and offering control over some parts).

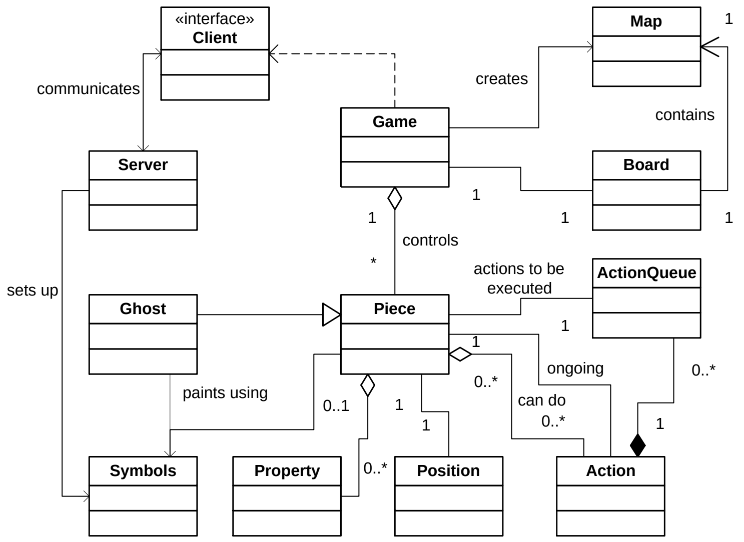

Figure 4: UML chart of the client package.

4.1 Pieces

A most central concept of the client is a *piece*, the client's view of a controllable game unit. A piece has a name, type and a unique identifier known as a handle. The piece also has a *position*, several *properties* (such as health and fuel), and a number of *actions* that represent things that the piece can do.

4.2 Actions

The piece keeps track of actions assigned by the user to be executed in order. These actions are entered by the user and may take an arguments (another piece, or a position on the map). The actions are usually chosen among those possible for the piece to execute. Some actions are marked instantaneous and are executed the moment the user assign them.

4.3 The client – server abstraction

From the client's point of view the *server* is an interface (`Server`) towards the communication. It contains the necessary methods for obtaining the required information and controlling the controllable pieces. When the game starts up, the class `communication.client.ServerAmbassador` that implements the `Server` interface is instantiated, and all server operations are executed through that instance.

The communication uses the interface `Client` as an abstraction of some client in a similar manner (see section 3).

To create alternate clients, simply implement the `Client` interface, and make sure it contacts some class implementing the `Server` interface (normally `communication.client.ServerAmbassador`).

4.4 Game control

The game is controlled by the class `Game`. It contains a game board that interacts with the user. The board displays the map and the pieces of the game, together with scrollbars and zoom buttons. This class implements the `Client` interface.

The symbols used for painting pieces are handled by the class `Symbols`. The symbols are loaded and added to the rest by the server.

4.5 Ghost pieces

Ghost pieces are semi-transparent pieces that are used for client-specific purposes (see *GECCO User's Manual*[3]). Such pieces are instances of the class `Ghost`. The ghosts are painted with semi-transparent symbols.

References

[1] Joel Brynielsson, Henrik Bäärnhielm, Andreas Enblom, Jing Fu Zi, Niklas Hallenfur, Karl Hasselström, Henrik Hägerström, Oskar Linde, Klas Wallenius, and Jon Åslund. *GECCO Developer's Manual.* Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, Sweden, May 2001.

[2] Joel Brynielsson, Henrik Bäärnhielm, Andreas Enblom, Jing Fu Zi, Niklas Hallenfur, Karl Hasselström, Henrik Hägerström, Oskar Linde, Klas Wallenius, and Jon Åslund. *GECCO General Description.* Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, Sweden, May 2001.

[3] Joel Brynielsson, Henrik Bäärnhielm, Andreas Enblom, Jing Fu Zi, Niklas Hallenfur, Karl Hasselström, Henrik Hägerström, Oskar Linde, Klas Wallenius, and Jon Åslund. *GECCO User's Manual.* Department of Numerical Analysis and Computer Science, Royal Institute of Technology, Stockholm, Sweden, May 2001.