Sun Game Server
Server API
Programming Guide

03/20/06

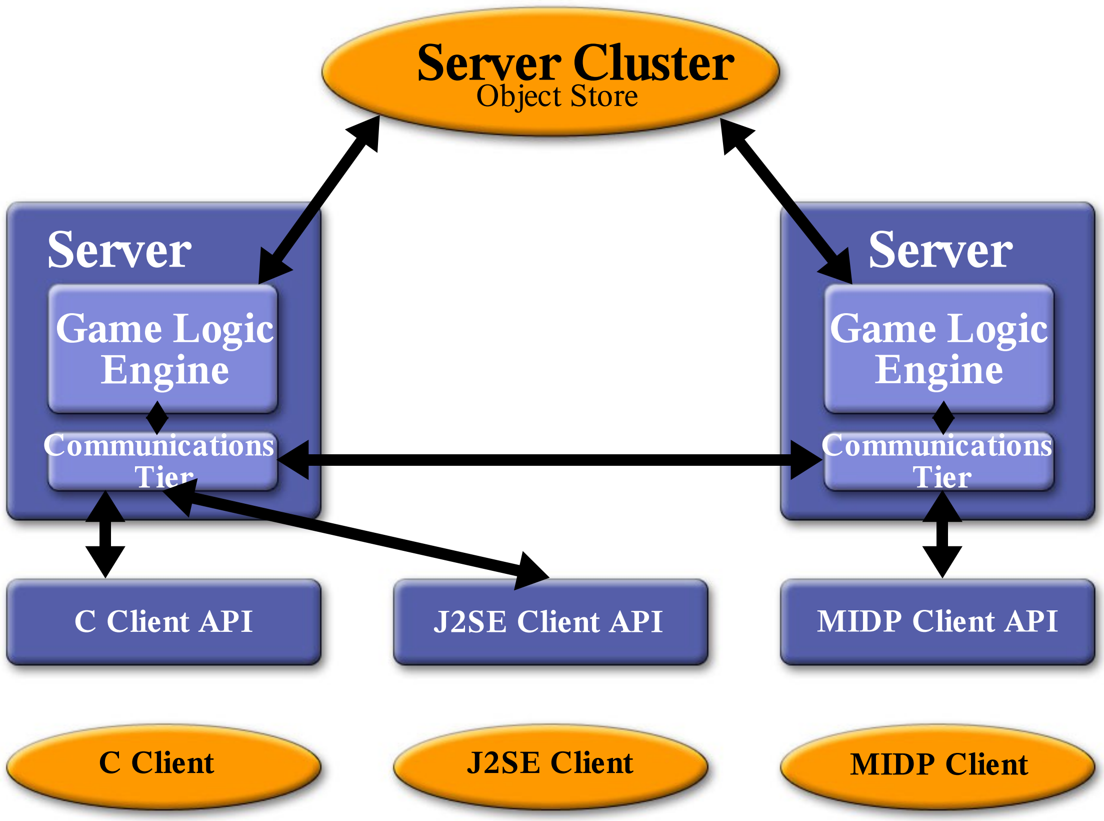
Contents

Introduction to the Sun Game Server

The Sun Game Server (SGS) is a distributed software system built specifically for the needs of game developers. It is an efficient, scalable, reliable, and fault-tolerant architecture that simplifies the coding model for the game developer.

The SGS is a distributed system. Each host contains an SGS *slice* that consists of a single stack of software running in a single process space. Each slice can handle 200-500 users, depending on the game and the hardware.

The figure below shows the architecture of the Sun Game Server:



This architecture consists of the following four tiers:

- The top tier (Tier 3) is a common data repository called the Object Store, which contains the entire game state and logic, implemented as a set of persistent *Game Logic Objects* (GLOs). Each SGS slice talks to this repository. Typically, GLOs include a “player” GLO that listens for packets from a given player and represents that player in the game world, GLOs that represent other objects in the game world (monsters, tools, and the like), and data structures for handling positioning within the world.

- Tier 2, on the Servers, is the Game Logic Engine. The Game Logic Engine is a container that executes event-driven code provided by the game developer. This event handling code is farmed out across the slices such that the entire set of active Game Logic Engines looks to the developer like a single execution environment. Race conditions and deadlock avoidance are handled automatically by the system so the user can write their code as if it were a single-threaded system.

- Tier 1, the communications tier, provides a highly efficient publish/subscribe channel system to the game clients. It also handles passing data from the clients up to the servers and from the servers back to the clients.

- Tier 0 is the Client API. This is a code library that client applications (game clients) use to communicate with each other and with the communications tier. As part of the SGS SDK we make Tier 0 available for J2SE, J2ME, and C++ applications. The C++ code is portable C++ and has been ported by us to two targets—Win32 and the Sony Playstation Portable (PSP).

  Tier 0 takes care of discovering and connecting to SGS servers, as well as handling fail-over to a new server should the one it is currently connected to suddenly fail.

For peer-to-peer communications, data passes only through Tier 0 and Tier 1.

More on the Communications Tier

The communications tier contains the following main sections:

- The *Router* does user authentication through pluggable Login Handlers. Once a user has been authenticated, the Router provides a publish/subscribe channel mechanism to facilitate both peer-to-peer communications among clients and client/server communications with the Game Logic Engine.

- The *Discoverer* collects information on available game connection points**.** (A *connection point* is a well-defined place where a client application can connect to a server application—for example, a particular *user manager* (discussed below) on a particular host.) It makes this information available in the form of a regularly updated XML document that is accessible at a well-known URL. This document is the starting point for all client communications. From the XML document, the client-side API extracts all the possible connection points for a given game, examines included parameters (such as the host and port), and chooses a connection point.

- The *User Manager* encapsulates all knowledge of how a client connects to and communicates with the SGS. The user manager is defined by interfaces and encapsulate a particular manner of communication (protocol, connection type, and so on). The SGS software distribution provides a TCP/IP-based user manager implementation, suitable for J2SE and C++ clients, and an HTTP-based implementation for J2ME clients.

There is an API wrapper that the client side of the user manager plugs into. This API wrapper knows about protocols defined by the Router for user authentication and failover and by the Discoverer for discovery information.

Coding the Sun Game Server

This chapter presents the fundamental concepts of how to code for the Sun Game Server (SGS). Understanding these concepts is the first step on the path to building massively scalable, reliable, fault-tolerant, and persistent network games.

Goals and Philosophy

In order to understand the SGS coding model it is useful to understand the system's goals. The fundamental goals are as follows:

1. Make server side game code reliable, scalable, persistent and fault tolerant in a manner that is transparent to the game developer.
2. Present a simple single-threaded event-driven programming model to the developer. The developer should never have to think about interactions of different threads of control.

Approach to Execution

The approach taken in the SGS is very similar to an "active objects" system, whereby each object has a single thread of control. The system makes all changes in state of that object on that thread of control. In this way, the system avoids race conditions.

SGS expands the active objects concept such that, rather than holding their own thread of control, objects are assigned as a group to a single thread of control that owns them until that thread is finished. This allows the atomicity to expand to groups of objects. Transactional storage ensures an all or nothing update of all the states of the aggregated objects. This thread of control, paired with a transactional context, is called an SGS *task*.

An application gets access to the SGS through methods on an object called SimTask, which represents the execution context of the current task**.** A task is fully transactional. This means that, in the wider scope of the SGS back end, operations on SimTask do not actually occur until a task ends and does a commit.

Working with SGS Tiers

As we know from previous discussion, the SGS is made up of four tiers (that is, layers) of code. Each layer provides a specific set of functions and communicates with the tiers above and/or below it. For the purpose of server-side programming, it is useful to understand how you interact with the code tiers.

The Client API (Tier 0) is a code library that client applications (that is, game clients) use to communicate with each other and with the Communications tier. It takes care of discovering and connecting to SGS servers, as well as handling fail over to a new server should the one it is currently connected to suddenly fail.

The Communications tier (Tier 1) serves two significant services: It provides a publish/subscribe channel system to game clients, and it handles passing data back and forth between clients and servers.

Except for being aware that the Server API classes ChannelID and UserID communicate with the router (which is part of the Communications tier), both Tier 0 and Tier 1 are transparent to server-side programming.

On the other hand, as a server-side programmer, most of your time will be spent working with the Game Logic Engine (Tier 2) and to a lesser extent the ObjectStore (Tier 3). The Game Logic Engine (GLE) executes your event-driven code, while the ObjectStore supports the GLE as a data repository providing the service of fast object storage and retrieval. Objects are checked out with either a GET (write-locked retrieval) operation or a PEEK (non-repeatable read) operation.

There can be many GLEs in an SGS back end. Each GLE can be a single or multi processor core system. Therefore, an SGS can gain additional processing capabilities and can handle more users simply by adding GLEs---all invisible to the application code.

There is only one ObjectStore per application in an SGS back end. All of the GLEs running events for a given application share a designated ObjectStore. As a result, all GLEs act on the same data, giving the appearance that all events are being processed by a single system.

Game Logic Objects

The GLE maintains a persistent world of Game Logic Objects (GLOs). Each GLO represents an entity in this world, and contains both data and the methods to act upon that data. Each GLO is created as an instance of a Game Logic Class (GLC), which is simply a Java class that implements the GLO interface and who's method code adheres to a few simple rules.

GLOs typically fall into three general types of entity:

- Actual objects in your game's simulated environment such as a sword, a monster, or an environment such as a room.
- Purely logical or data constructs such as a quad-tree for determining player-proximity or a walk-mesh to determine movement paths.
- Proxies for human players in a world of GLOs

Figure 1 illustrates a very basic world consisting of a single room containing two players and a sword.

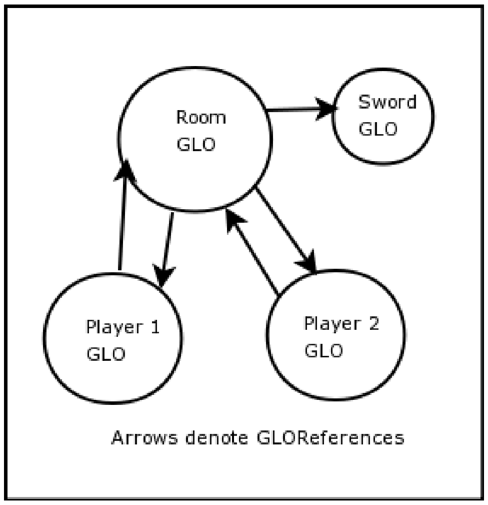


Figure 1: Example of a simple GLO world

Designing Your GLOs

When deciding where to break data up into multiple GLOs, consider these questions:

1. How big is the data? The more data a single GLO encompasses in its state the more time it takes to load and save.
2. How closely coupled is the data? Data that are generally accessed together are more efficiently stored in the same GLO. Data that are accessed independently are candidates for separation onto different GLOs. Data that have to be modified atomically are best stored the same GLO.
3. How many simultaneous Tasks are going to need access to this data? It is best to split up data

that has to be locked with a get() from data that can be shared with a peek(). Over all the goal is to have as few bottleneck get()s on the same GLO as possible.

Of all of these considerations, 3 is the most critical to a well running an SGS application.

Events and Tasks

By itself, a world of GLOs is static and uninteresting. However, GLOs are more then just data, they also contain code. This code gets executed in response to things that happen outside of the world of GLOs. These outside occurrences are referred to as *events* in the game server. System events are defined in a set of well known interfaces. Event processing usually begins with a get() of a GLO that implements the appropriate interface.

Events trigger the execution of a method on a GLO that has registered as an event listener. This in turn can inspect, modify, and/or call methods on other GLOs. The entire chain of GLO execution that occurs in response to one event is called a *task.*

Tasks occur without race conditions or deadlock. If one task modifies a GLO, no other task may modify that GLO until the first is finished. For this reason, Tasks are time-limited, and must be fairly short. If a task needs to do more then it can accomplish in its allotted execution time, it can continue the processing by submitting a child task to the system for execution after it is has finished.

GLO References

GLOs refer to other GLOs using a special reference type called a GLOReference. A GLOReference is much like a standard Java Reference Type (such as SoftReference and Weak Reference), but with additional properties and features.

A GLOReference contains the identifying information for a GLO in the ObjectStore. A task uses a GLOReference to get a copy of the object to inspect or modify. This copy is valid until the end of the task.

It is very important to refer to a GLO through a GLOReference. A GLOReference is a database reference to a GLO in storage, while a Java reference is a reference to an in-memory copy of the GLO. So for example, when GLO A, who is holding the Java reference to GLO B, is itself sent to storage, a copy of GLO B (through a Java reference) is stored with it as part of its own state. The next time a task pulls GLO A out of storage, the stored Java reference will refer to a brand new object (that is, the current in-memory copy of GLO B), and not the previously-stored copy of GLO B.

You may at times need to keep in a field, a Java reference returned from a GLOReference.get() (or peek() or attempt()) for the life of a task. You are free to put that into a Java reference and it will remain valid for the life of that task. However, since storing it is of no value, you should either null the field out before the end of the task, or define the field as Transient in the GLO's GLC.

Accessing GLOs

The GLOReference has three access methods: get(), peek() and attempt().

A get() returns a task-local copy of the GLO that can be modified and will be written out at the successful conclusion of the task. If another task has already done a get() (or attempt()) on the same GLO, then you will be blocked until it is available. In actuality, a task attempting a get() that could result in deadlock, will either abort and re-queue itself, or abort and re-queue the task that currently holds the lock on the GLO in question. From a coding perspective though, this looks the same as a blocking get().

A peek() returns a task-local copy of the GLO that will **not** get written back on task completion. Peek() takes no locks on the GLO and leaves other tasks free to access the GLO as well.

An attempt() is like a get() except that it does not block. Instead, it returns null if another currently executing task has already done a successful attempt() or get() on that GLO.

The Player GLO

In order to get executed when outside events occur, GLOs register themselves as event handlers. One very important type of GLO is the Player GLO. A Player GLO implements the SimUserDataListener interface and registers itself as the UserDataListener listening to incoming data packets from a userDataReceived event.

The Player GLO acts as a proxy for the player in the world of GLOs. The player sends data packets to the server using the SGS Client API. This causes a userDataReceived event in the system, which results in a task that calls the Player GLO's userDataReceived method. The Player GLO parses the packet to find out what it is supposed to do, and then acts on itself and other GLOs in order to accomplish the requested task.

Figure 2 shows our simple GLO world, with the addition of two players connected to the SGS as clients. to listen to the Game Clients,, the Player GLOs are registered as UserDataListeners.

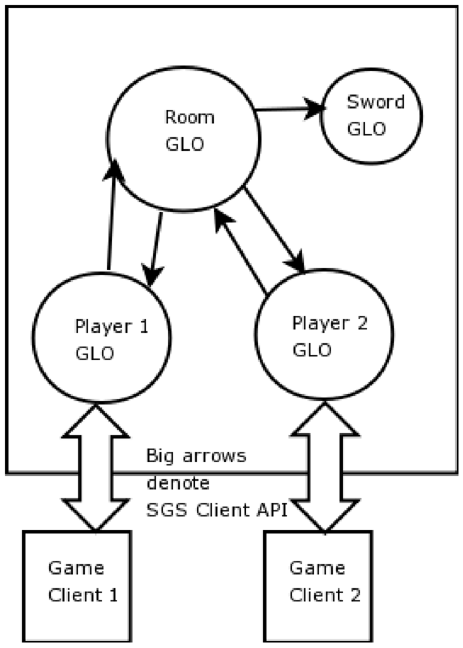


Figure 2: Client connections to the simple GLO world

The Player GLOs have a “current room” field, which is a GLOReference that points to the Room GLO. The Room GLO has an inventory list, which is a list of GLOReferences. Currently, there are three items in the list: the two players and a sword. Each is represented by a GLO (Player 1 GLO, Player 2 GLO, and Sword GLO).

The Boot GLO

Above we had a world of GLOs consisting of a Room GLO, a Sword GLO and a couple of Player GLOs. However when we start the game in the SGS for the first time, the world of GLOs doesn't look like that. In fact it looks like this:

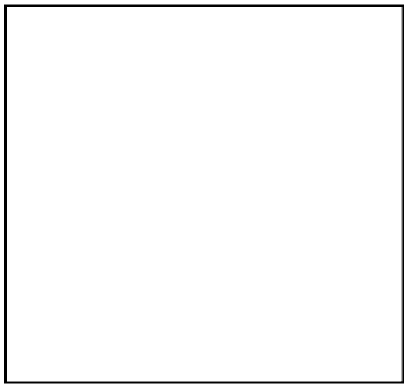


Figure 3: World of GLOs at first time start of game

Which is to say, it is empty.

How then do the GLOs get into the ObjectStore in the first place?

The answer is a special GLO called the Boot GLO. There are two special things about the GLC that defines the Boot GLO:

- It implements the SimBoot interface. This interface defines one method:

  ```
  public void boot(GLOReference<? extends T> thisGLO, boolean firstBoot);
  ```

  This rather arcane bit of syntax says that the boot method takes two parameters: The first is a GLOReference to the Boot GLO itself; the second is a boolean called firstBoot.

- It is defined in the XML deployment descriptor used to deploy an application into the SGS. The XML deployment descriptor is the file that contains all the installation information for the application. All the deployment descriptors installed into the SGS are themselves listed in the Install.txt file.

These two properties combine in the following way:

- On the boot of the SGS (or the installation of a new application into the SGS), the SGS attempts to locate the Boot GLO in the ObjectStore.
- If the application has never been booted before, then its ObjectStore in the SGS is blank (like in Figure 3) , and the SGS will fail to find the Boot GLO. In that case it creates the Boot GLO itself, and then starts a task that calls the boot method on the new Boot GLO, with firstBoot set to True.
- If on the other hand, the application has been booted at least once, the ObjectStore contains the Boot GLO. In this case, the task is started and it calls the boot method on the existing Boot GLO, with firstBoot set to False.

In the case of our little demo app, the boot method will have a block in it that in pseudo code, looks something like this:

```
if (firstBoot){
        CREATE ROOM GLO
        CREATE SWORD GLO
        ADD REF TO SWORD GLO TO ROOM'S INVENTORY
        SAVE A GLO REF TO ROOM FOR LATER
}
```

In general, it is the responsibility of the Boot GLO to create the initial world of GLOs during firstBoot.

Now we have something that is beginning to look like our game:

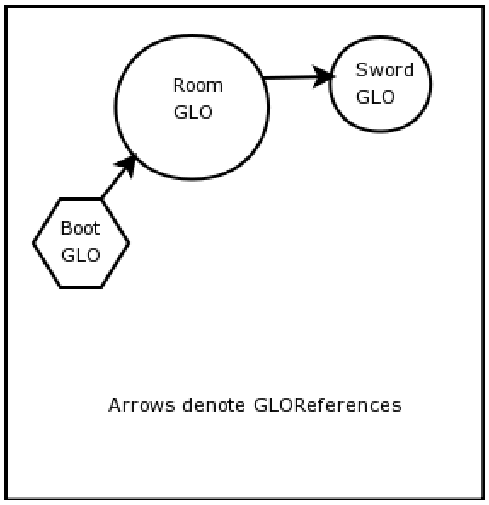


Figure 4: Boot GLO creates initial GLO world

We still don't have Player GLOs however. We will create the Player GLOs as users join much the same way the Boot GLO was created. The first time we see a user log in, we create a new Player GLO for them. After that, every time the user logs in, we just reconnect him to his existing Player GLO. Thus, the system creates Player GLOs as needed, and remembers user information between log ins.

So how do we find out when a user has logged in?

The answer is that there is another event listener interface called SimUserListener. A GLO that implements the SimUserListener interface has the following methods:

```
public void userJoined(UserID uid, Subject subject);
public void userLeft(UserID uid);
```

Every time a user logs into an SGS application, a task is started that calls the userJoined() method on all the registered SimUserListeners for that game. Every time a user logs out from the application, a task is started that calls the userLeft() methods on all the registered SimUserListeners for that game.

In the case of our simple example game, we have made our Boot GLO also a SimUserListener by defining its GLC as implementing both SimBoot and SimUserListener. Note that these don't have to be the same GLO. In its firstBoot block the Boot GLO could have created a GLO of a different GLC that implements SimUserListener. However, combining them is often a convenience, and thus is a common pattern in SGS application development.

The boot object is predefined as the listener for the boot event. In order to receive the userJoined and userLeft events, we need to tell the system that Boot GLO wants to listen to them. This is done in the boot(...) method every time the system boots. In pseudo code, our entire boot method looks like this:

```
if (firstBoot){
      CREATE ROOM GLO
      CREATE SWORD GLO
      ADD GLO REF TO SWORD GLO TO ROOM'S INVENTORY
      SAVE A GLO REF TO ROOM FOR LATER
 }
REGISTER  thisGLO AS A SimUserDataListener
```

Note that `thisGLO` is a GLOReference to the Boot GLO that was passed in on the boot(...) method call. When we register objects as listeners, we pass a GLOReference to them, not a Java reference to the object itself.

When the userJoined callback is called on our Boot object, it executes the following (in pseudo code):

```
player_name = subject.GetPlayerName();
glo_name = "player_"+PLAYER_NAME;
if DOES_GLO_EXIST(glo_Name){
    FIND_GLO(glo_name);
```

```
        } else {
            CREATE NAMED PLAYER GLO(glo_name);
        }
        SET currentRoom on PLAYER GLO to SAVED GLO REF TO ROOM
        GET ROOM GLO
        ADD PLAYER REF TO ROOM GLO's PLAYERS LIST
        REGISTER PLAYER GLO AS SimUserDataListener(uid);
```

SimUserDataListener is another event interface. It defines methods that get called on Tasks to respond to actions the client takes with the client API. Among these are the client sending data to the server for processing. Figure 5 illustrates that our GLO world is starting to look the way we wanted it to.

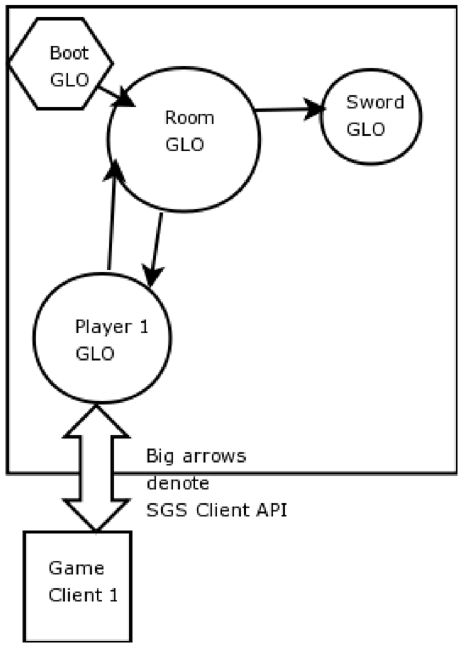


Figure 5: Client sends data to server

When a second user logs in, we will be back to our original world. Figure 6 illustrates our world after the restarting the game with our previous players:

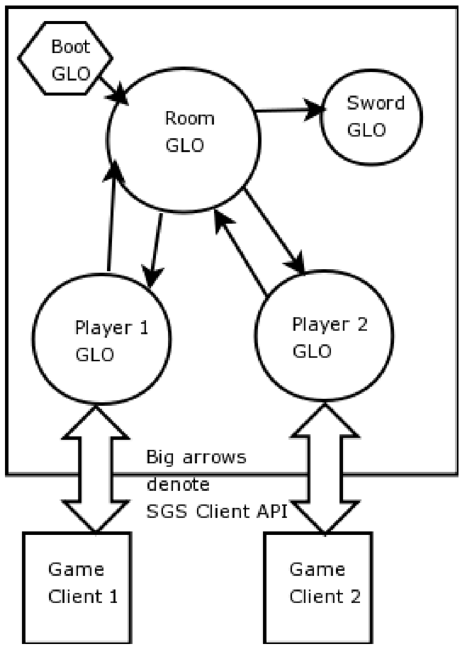


Figure 6: Boot GLO reestablishes simple GLO world

So far, the logic has been laid out in pseudo code. However the actual code to implement this application, both client and server is attached at the end of this document and is included in the SDK examples as the "SwordWorld" application. SwordWorld goes a bit further in that it also implements a "look" command, in order to show you how the Player GLO actually handles commands being sent from the client.

Using the SimTask Object

We have described several interactions with the server system. These Tasks include creating GLOs, looking up GLOs by name, and registering GLOs as event listeners.

In practice, any time a GLO has to talk to the system, it does so by utilizing the SimTask object. You can think of the SimTask kind of like the Java AWT Graphics object. For example, when you respond to a paint() call in Java, you do the actual drawing by calling methods on a Graphics object that the system provides to you. Similarly, when you respond to SGS events, you do so with a SimTask object that the system provides you.

A SimTask object has a life cycle that lasts from the beginning to the end of the task on which it is being used. After that, it is invalid. Thus, **it is very important that you do not try to store the SimTask from one event to the next.** Instead, you must fetch it from the system within your event handling code by calling SimTask.getCurrent().

UserID and ChannelID Life Cycle

The SimTask uses UserIDs to identify connected users and ChannelIDs to identify open channels. As with SimTasks, it is very important to be aware of the UserID and ChannelID life cycle. UserIDs are really session IDs. What this means, is that a particular session ID is valid only from user login to logout. On the next login, the user gets a totally different ID. In coding terms, **the life cycle of a UserID is from userJoined until userLeft.**

How do you recognize a returning user? The answer is that their login name and other login information is all in the Subject object that gets passed to the userJoin event handler. If you look at the

`SwordWorldBoot.java` code, you will see an example of how to use that information.

ChannelIDs also have a limited life cycle. A ChannelID is valid from the GLE boot until the GLE process ends. What this means in practical terms is that **you must reopen the channels in your boot event handler.** `SwordWorldBoot.java` contains an example of this as well.

Task Execution Priorities

Although Figure 6 on page 10 shows the logical flow of data, it could be a bit misleading. At best it is incomplete in that it does nothing to show how events actually get processed. In reality, events are processed and Tasks execute in a more or less simultaneous order. It is probably easiest to think of this as a set of task Processors surrounding the world of objects. Figure 7 illustrates this concept.

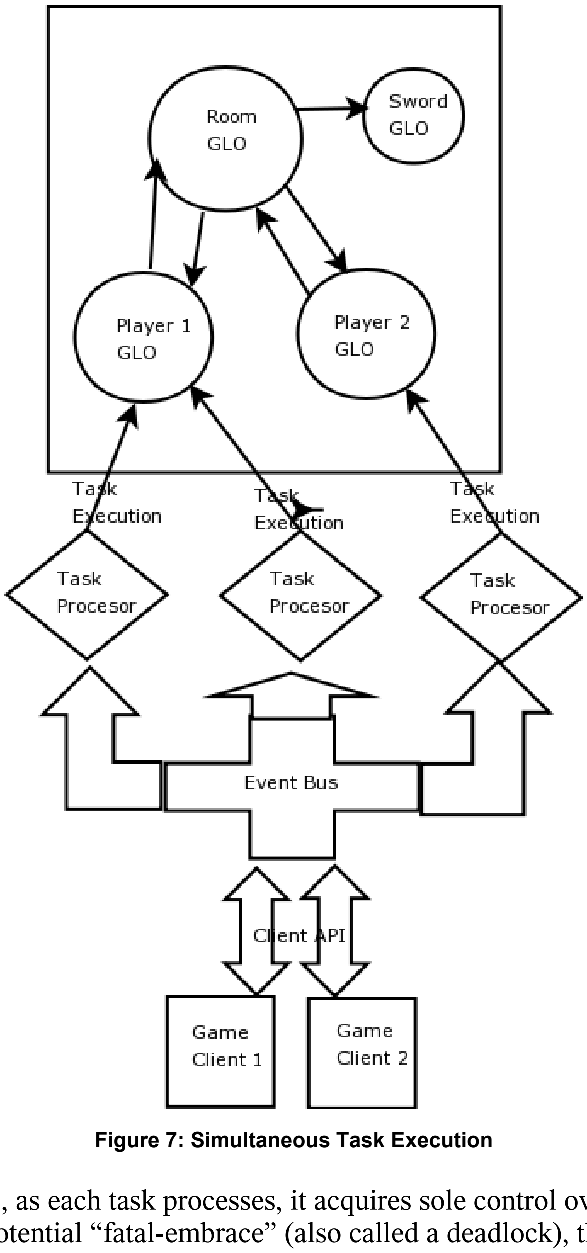


Figure 7: Simultaneous Task Execution

For the sake of this example, as each task processes, it acquires sole control over the state of the GLOs it uses. If two Tasks reach a potential "fatal-embrace" (also called a deadlock), the task that arrives last is aborted to get it out of the way of the first task. The system will attempt to process the later task at another time. An aborted task acts like it never began with regard to input/output and system state (in database talk, it is "rolled back").

Tasks are atomic, that is, you will never get just part of a task executed and visible to another task. The code within a task is guaranteed to execute in order (within the limits of Java's own in-order guarantees).

Should they hit a potential deadlock, Tasks have a relative order. They are otherwise executed simultaneously, and might be reordered at the convenience of the SGS. The only guarantee on task ordering is that such reordering will never permanently starve a task (in scheduling parlance, this property is called “fairness”).

Tasks can request that other child Tasks be run. Child tasks are guaranteed to begin after their parent has finished. Other than that and the fairness property, there are no other order guarantees on child Tasks.

The results of a task are guaranteed durable within a few moments of task conclusion. Should the system totally fail, referential integrity is ensured, though the results of the last few Tasks might be lost.

These are the **only** guarantees the system makes. This is important enough to bear repeating in a summation:

Important: Although all tasks are race and deadlock proof, ordering is only guaranteed on a per-user basis.

What this means is that all the Tasks pertaining to a single user are guaranteed to be processed in the order that their events arrive in the system. A later event does not start processing until all of that user's earlier events have processed. However, ordering of tasks between users is **not** guaranteed. An event pertianing to User 1 that arrived later then one pertiaining to user 2 may well get executed as a task first, depending on object contention.

Creating GLOs

A new GLO is created in the back-end database with a call to createGLO(template_object). A template object holds all the values you want the GLO to contain. Calling createGLO effectively creates a duplicate of the template object inside of the database. What is returned to you is a GLOReference to the object in the database. To do further work with that object you would use GLOReference.get() (or peek() or attempt()) just like you would for an existing GLO.

When a GLO is created, it can be assigned a name. This is done using the createGLO(name, template_object) call. Unlike createGLO, it is possible for a named create to fail. If there already exists a named object in the database with the same name, a named create will fail and return null. Otherwise it is exactly like the basic unnamed createGLO call.

Note: A newly created GLO is in an indeterminate state until commit. For that reason, any attempt by another task to get it, or to create another GLO with the same name, will block until the creating task commits. A peek on the object by another task will return null, as if it did not exist.

The Named GLO Create Pattern

Properly accessing a new named GLO can be tricky. The right way to do it is as follows:

```
GLOReference ref = findGLO(“myName”);
if (ref==null) { // doesnt exist yet
    MyGLC  template = new MyGLC(....);
    ref = createGLO(“myName”,template);
    if (ref==null) { //we raced with another task and lost
       ref = findGLO(“myName”);
    }
}
```

This pattern is guaranteed to leave you with a valid GLOReference you can access.

Note: Nothing keeps another task from deleting an object that your task does not have a GET lock on, so even though you have a valid GLOReference its always possible that what it refers to will go away before you can peek or get it.

If you prefer to use static instance constructors, it can get a bit trickier. It is particularly tricky if, as part of your setup, you need to pass a GLOReference to the newly constructed GLO to other GLOs.

For that case, the game server team recommends the following pattern:

```
public class Foo implements GLO {

  // ... private Foo fields ...
  private GLOReference thisRef; // the GLORef for this GLO
  private GLOReference myBazRef; // some Baz GLO that Foo uses

  // Factory method to create a new Foo from the given arguments
  // and register it as a GLO with the system.
  public static GLOReference create(SimTask task, int n) {

    // The GLO name for this Foo, if it is named.
    String name = "Foo-" + n;

    // Create a new GLO. We assume it will succeed in this
    // example,
    // and perhaps have designed this app so that we know it will.
    // We pass in a template object that has done as much
    // initialization as it can without knowing its own
    // GLOReference,
    // such that the object is a valid Foo.
    // Note that we do not keep a reference to this template
    // object,
    // since the new, locked GLO must be retrieved from "ref".
    GLOReference ref = task.createGLO(new Foo(task, n), name);

    // Get the modifiable, locked Foo GLO we just created.
    Foo foo = (Foo) ref.get(task);

    // Allow foo to perform any remaining init that needs the ref
    foo.init(task, ref);

    // Return a reference to the fully-initialized Foo to the
    // caller.
    return ref; // (or foo itself, if we prefer)
  }

  protected Foo(SimTask task, int x) {
    // Foo's constructor does some of the work...
    myBazRef = task.findGLO("Baz-" + x);
  }

  protected void init(SimTask task, GLOReference ref) {
    // ...but work that requires the GLOref must be
    // done outside the constructor.

    // Remember my own GLOReference
    thisRef = ref;

    // Register myself with my Baz object.
    Baz baz = (Baz) myBazRef.get(task);
    baz.setFooReference(thisRef);
  }

  // ... public, Foo-specific methods...
}
```

The above code satisfies a number of very important design considerations:

1. createGLO uses the template passed in as a pattern to create the GLO, but after that the template has no further connection to the GLO. Changes to the template made after the createGLO call do not effect the GLO. To make changes to the GLO, the actual GLO must be fetched using a get() operation on the returned GLOReference.
2. Java specifically outlaws passing the "this" pointer in a call from out of a constructor. This is because the object is not fully constructed until the constructor returns. Code other than the constructor itself acting on a not-fully-constructed object is defined by the language spec as having undefined and potentially dire consequences.
3. Because of 2 above, createGLO(this) must **never** be called from an object's constructor.
4. It is good object-oriented coding to keep the details of initialing a Foo within the Foo class, instead of exposing the details to users of Foo. Since Foo cannot call createGLO during construction and cannot return a GLOReference during construction, a Factory Method on class Foo is a logical place to encapsulate the full creation of a Foo that needs to make itself known to other objects.

GLC Dos and Dont's

When writing your GLCs (that is, the Java class that implements the GLO interface), there are a number of rules you must remember. Breaking these rules can result in nasty and hard to find bugs, the server forcefully ending the run of a task, or in extreme cases the halt of your entire application.

A valid GLC must adhere to the following restrictions:

1. It **cannot have static fields**. (The one exception being final static constants.) There are a number of reasons for this. The most important one is that static fields only exist within the scope of a single VM, and an SGS back end floats GLOs between many different VMs.
2. It **must not use the synchronized keyword.** First, this is unnecessary in properly written GLOs. Second, it won't work as synchronization is also relative to a single VM. Lastly, it causes interactions between Tasks that can defeat the system deadlock proofing feature.
3. It **should not make blocking IO calls or stay in a loop for a long period of time**. The system contains the assumption that tasks are short lived. If a task lives too long, it will be force ably terminated by the back end.
4. It **must not catch java.lang.Error**. There are sub-classes of Errors defined by the system that it needs to see, and thus must not be swallowed by application code. If you really need to catch some other Error, then write your catch statement to specifically catch that Error alone.
5. It **cannot carry a non-transient Java reference to another GLO**. Instead use GLOReference. Any object that is referred to by a Java reference chain that starts at the GLO is assumed to be a private state of that particular GLO. This means that, while you may set two Java References on two different GLOs to the same Java object during a task, they will each end up with their own copy of that object at the termination of the task.
6. It **cannot save a SimTask on a non-transient field for later use**. This is because a SimTask's life cycle ends with the task in which it was fetched. After that, it is no longer valid, and attempts to use it will cause the current task to be abandoned as illegal code.

Locating the Server API Classes

The com.sun.gi package contains the entire Sun Game Server system in a variety of sub directories. The server API classes reside in the `apps`, `comm`, `framework`, `glutils`, and `logic` sub directories. Architecturally, the `comm` and `framework` classes are part of the Communications layer (Tier 1), and the `apps, glutils, and logic` classes are part of the Game Logic layer (Tier 2).

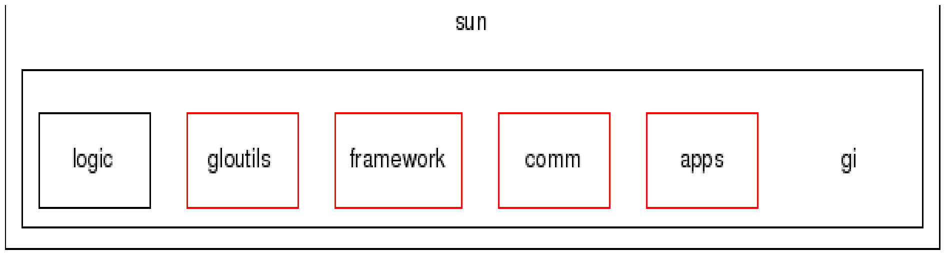


Figure 8: SGS directories containing server API classes

These are the SGS Server API classes with brief descriptions:

Class	**Description**
com.sun.gi.apps.chattest.ChatTestBoot	
com.sun.gi.comm.routing.ChannelID	
com.sun.gi.comm.routing.UserID	An opaque type that represents a user connection to the router system.
com.sun.gi.framework.rawsocket.SimRawSocketListener	Objects interested in communicating with sockets to arbitrary hosts should implement this interface
com.sun.gi.gloutils.pdtimer.PDTimer	PDTimer is the primary GLO in the Persistant/Distributed timer system
com.sun.gi.gloutils.pdtimer.PDTimerEvent	
com.sun.gi.gloutils.pdtimer.PDTimerEventList	
com.sun.gi.logic.GLO	All GLOs (Game Logic Objects) must implement this tagged interface
com.sun.gi.logic.GLOReference< T extends GLO >	All GLOs must refer to other GLOs through GLOReferences
com.sun.gi.logic.SimBoot< T extends SimBoot >	SimBoot must be implemented by the boot object of a Game Server application
com.sun.gi.logic.SimChannelListener	
com.sun.gi.logic.SimTask	
com.sun.gi.logic.SimTimerListener	
com.sun.gi.logic.SimUserDataListener	The SimUserDataListener interface must be implemented by any that will be registered to receive game user data events
com.sun.gi.logic.SimUserListener	The SimUserListener interface must be implemented by any that is registered to handle game user events

Appendix A: SwordWorld Code

SwordWorld Server

The SwordWorld server is made up of the following GLCs:

- SwordWorldBoot.java
- Room.java
- RoomObject.java
- Player.java

In order to process the text commands, Player.java additionally uses this utility class:

- StringUtils.java

Below are each of these classes.

SwordWorldBoot.java

```
/*
 Copyright (c) 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 Clara, California 95054, U.S.A. All rights reserved.

 Sun Microsystems, Inc. has intellectual property rights relating to
 technology embodied in the product that is described in this document.
 In particular, and without limitation, these intellectual property rights
 may include one or more of the U.S. patents listed at
 http://www.sun.com/patents and one or more additional patents or pending
 patent applications in the U.S. and in other countries.

 U.S. Government Rights - Commercial software. Government users are subject
 to the Sun Microsystems, Inc. standard license agreement and applicable
 provisions of the FAR and its supplements.

 This distribution may include materials developed by third parties.

 Sun, Sun Microsystems, the Sun logo and Java are trademarks or registered
 trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

 UNIX is a registered trademark in the U.S. and other countries, exclusively
 licensed through X/Open Company, Ltd.

 Products covered by and information contained in this service manual are
 controlled by U.S. Export Control laws and may be subject to the export
 or import laws in other countries. Nuclear, missile, chemical biological
 weapons or nuclear maritime end uses or end users, whether direct or
 indirect, are strictly prohibited. Export or reexport to countries subject
 to U.S. embargo or to entities identified on U.S. export exclusion lists,
 including, but not limited to, the denied persons and specially designated
 nationals lists is strictly prohibited.

 DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS,
 REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF
 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT,
 ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE
 LEGALLY INVALID.

 Copyright © 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara,
 California 95054, Etats-Unis. Tous droits réservés.

 Sun Microsystems, Inc. détient les droits de propriété intellectuels
 relatifs à la technologie incorporée dans le produit qui est décrit dans
 ce document. En particulier, et ce sans limitation, ces droits de
 propriété intellectuelle peuvent inclure un ou plus des brevets américains
 listés à l'adresse http://www.sun.com/patents et un ou les brevets
 supplémentaires ou les applications de brevet en attente aux Etats -
 Unis et dans les autres pays.

 Cette distribution peut comprendre des composants développés par des
 tierces parties.

 Sun, Sun Microsystems, le logo Sun et Java sont des marques de fabrique
 ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans
 d'autres pays.

 UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et
 licenciée exlusivement par X/Open Company, Ltd.

 see above Les produits qui font l'objet de ce manuel d'entretien et les
 informations qu'il contient sont regis par la legislation americaine en
 matiere de controle des exportations et peuvent etre soumis au droit
 d'autres pays dans le domaine des exportations et importations.
 Les utilisations finales, ou utilisateurs finaux, pour des armes
 nucleaires, des missiles, des armes biologiques et chimiques ou du
 nucleaire maritime, directement ou indirectement, sont strictement
 interdites. Les exportations ou reexportations vers des pays sous embargo
 des Etats-Unis, ou vers des entites figurant sur les listes d'exclusion
 d'exportation americaines, y compris, mais de maniere non exclusive, la
 liste de personnes qui font objet d'un ordre de ne pas participer, d'une
 facon directe ou indirecte, aux exportations des produits ou des services
 qui sont regi par la legislation americaine en matiere de controle des
 exportations et la liste de ressortissants specifiquement designes, sont
 rigoureusement interdites.

 LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS,
 DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES,
 DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE
 GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE
 UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.
*/


package com.sun.gi.apps.swordworld;

import com.sun.gi.comm.routing.ChannelID;
import com.sun.gi.comm.routing.UserID;
import com.sun.gi.logic.GLOReference;
import com.sun.gi.logic.SimBoot;
import com.sun.gi.logic.SimTask;
import com.sun.gi.logic.SimUserListener;

import java.nio.ByteBuffer;
import java.security.Principal;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import javax.security.auth.Subject;

/**
 * The bootstrap class for the toy text-mud minimal SGS example app
 * SwordWorld.  <p>
 *
 * The Boot method is invoked by the system when an SGS stack starts
 * up.  <p>
 *
 * It implements {@link SimBoot} so it can be a boot object.  It
 * implements {@link SimUserListener} so it can handle users joining
 * and leaving the system.</p>
 *
 * @author Jeff Kesselman
 *
 * @version 1.0
 */
public class SwordWorldBoot
       implements SimBoot<SwordWorldBoot>, SimUserListener {

    /**
```

```
     * All GLOs should define a <code>serialVersionUID</code> because
     * they are serialized.  This turns off version checking so we can
     * change the class and still load old data that might already be
     * in the ObjectStore.
     */
    private static final long serialVersionUID = 1L;

    /**
     * This field holds a GLOReference to the one room in our
     * minimal MUD world.
     *
     * All fields on a GameLogicObject (GLO) that refer to other GLOs
     * must be GLOReferences.  If we just said "Room roomRef" then
     * we would end up with a copy of the room as part of the state
     * of this GLO rather then a reference to the other GLO.
     */
    private GLOReference<Room> roomRef;

    /**
     * Currently all data sent from the server to users must be sent
     * on a channel. (This is likely to change  in the next major
     * version of the API.)
     *
     * This field holds the ID of a channel that is used to send data
     * back to all the users.
     */
    private ChannelID appChannel;

    /**
     * We get informed of who events regarding users such as
     * userJoined and userLeft refer to by an identifying UserID. <p>
     *
     * This map lets us retrieve the Player GLO assoigned to that
     * User.  For more information, see the Player.java file.
     */
    private Map<UserID,GLOReference<Player>> uidToPlayerRef =
            new HashMap<UserID,GLOReference<Player>>();

    /**
     * This is the boot method which gets called when the SGS stack
     * starts up.
     *
     * Fundementally we get two paremeters in the boot call.  The
     * first is of type GLOReference<SwordWorldBot> and is a
     * GLOReference to this boot object.  (You almost always need it
     * here so for convenience we pass it in.)
     *
     * The second is a boolean called "firstBoot".  When the SGS stack
     * is started up the stack looks for an existing boot object.  if
     * the boot object is there then it just gets it and calls boot()
     * on it with firstTime set to false.  If it is *not* there, it
     * creates it and calls boot() with firstTime set to true.
     *
     * This means that you can tell if the ObjectStore is empty by the
     * value of firstTime.  If the ObjectStore is empty, you will want
     * to do set up and create the other GLOs your app requires.  is
     * not there
     *
     * @see
     * com.sun.gi.logic.SimBoot#boot(com.sun.gi.logic.GLOReference,
     * boolean)
     */

    public void boot(GLOReference<? extends SwordWorldBoot> thisGLO,
            boolean firstBoot) {

        /*
         * The SimTask object is your window back into the system.
         * All actions on the system such as creating or destroying
         * GLOs, registering event handlers, and so forth require you
         * to call methods on the SimTask.
         *
         * A SimTask object refers to the task that is performing the
         * access to this object and therefore it is ONLY valid during
         * the execution of the Task in which SimTask.getCurrent() is
         * called.  For this reason you should NEVER store a SimTask
         * object in a GLO field unless that field is marked transient
         * and thus not saved past the end of the task.
         */
        SimTask simTask = SimTask.getCurrent();

        if (firstBoot){

            /*
             * firstBoot is true, so we need to create our game GLOs.
             * In order to create a GLO, you first create a template
             * object.  You then pass that template into
             * simTask.createGLO().
             */

            Room roomTemplate = new Room("A big brightly lit room.");
            roomRef = simTask.createGLO(roomTemplate);

            /*
             * IMPORTANT:  The template is *not* the GLO.  If you want
             * to make changes to the fields of the GLO you MUST get
             * the GLO itself with a GLOReference.get().  Changing
             * fields on the template after the createGLO has been
             * called will have NO effect on the GLO itself.
             */

            Room roomGLO = (Room)roomRef.get(simTask);

            /*
             * Now that we have a room GLO, let's make a sword GLO
             */
            RoomObject sword = new RoomObject("A shiny new sword");
            GLOReference<RoomObject> swordRef = simTask.createGLO(sword);

            /*
             * We are going to add the sword to the inventory list of
             * the room.  Note that we do *not* need to get() the
             * swordGLO itself for this.  Since we never want to store
             * Java refernces to GLOs in the fields of other GLOs but
             * only GLOReferences, what we have is all we need.
             */
            roomGLO.addToInventory(swordRef);
        }

        /*
         * A channel is required in the current version of the API in
         * order to send data back to users.  This code and the client
         * code have agreed apriori to call the one channel we will
         * use "GAMECHANNEL".  By opening the channel both here and in
         * the client, we establish a data path from client to server.
         */
        appChannel = simTask.openChannel("GAMECHANNEL");

        /*
         * Finally, we want to know when users log in to the server so
         * we can hook them up to a Player object.  Therefore, we
         * register ourselves as the SimUserListener.
         */
        simTask.addUserListener(thisGLO);

    }

    /**
     * This callback is called by the system when a user logs into the
     * game.  It takes 2 parameters.  The first is the UserID of the
     * user.  IMPORTANT:  A UserID is really a session ID and is valid
     * until the current session ends (the user is disconnected from
     * the system). The next time the same user logs in he or she will
     * get a *different* UserID. <p>
     *
     * The second parameter is a Subject.  This is a JDK class and can
     * be looked up in the Sun J2SE API docs.  It is bascially a
     * container for ids (called Principals) and permissions. <p>
     *
     * When the user logs in, the Validator for this game sets all
     * this information in the Subject.  By convention, the first
     * Principal always contains a String denoting the user's login
     * name.  @see
     * com.sun.gi.logic.SimUserListener#userJoined(com.sun.gi.comm.routing.UserID,
     * javax.security.auth.Subject)
     */
    public void userJoined(UserID uid, Subject subject) {

        /*
         * first we get the first principal
         */
        Set<Principal> principles = subject.getPrincipals();
        Principal principal = principles.iterator().next();

        /*
         * next we create a name for a named GLO by combining the
         * prefix "player_" with the player's login name as
         * returned by the first principal.
         * For any given login this will always result in the same
         * name.
         */
        String playerName = "player_"+principal.getName();
        SimTask simTask = SimTask.getCurrent();

        /*
         * The following "three phase lookup" pattern is a common
         * GLO pattern and should be used whenever you want to
         * create a named GLO if it doesn't already exist:
         * First we ask the system if a named GLO already exists that
         * has this name.
         */
        GLOReference<Player> playerRef = simTask.findGLO(playerName);
        if (playerRef == null){

            /*
             * If the findGLO returns null then there was no named GLO
             * with that name, so we attempt to create it with a new
             * instance of Player.
             *
             * Note the design pattern of creating the object for
             * which we want to create a GLO inside the call to
             * createGLO.  This prevents a reference to that object
             * from "leaking" into our own code, where we might
             * accidently use it.  GLO objects should ONLY be
             * referenced via a GLOReference.
             */
            playerRef = simTask.createGLO(
                    new Player(principal.getName(), appChannel));
            if (playerRef == null) {

                /*
                 * Because of the concurrent nature of the system, it
                 * is possible that we ended up in a race with another
                 * task to create this object.  We both tried to get
                 * it at the same time, then since it didn't exist we
                 * both tried to create it.  The system enforces that
                 * a named object can only be created if it doesn't
                 * exist, and only once if two tasks try to create at
                 * the same time.  The task the system denied creation
                 * to gets a null reference back from the createGLO
                 * call.  In this case, the other task created it so
                 * now we can safely just ask for it by name again,
                 * knowing it now exists.
                 *
                 * Note that there is one condition under which this
                 * assumption is not true, and that's if another task
                 * could have destroyed it between when we called
                 * createGLO and the second attempt to find.  In this
                 * app we know that will never happen, but a more
                 * general solution would actually wrap a while
                 * (playerRef == null) around this and keep trying
                 * until it either created or found the GLO.
                 */
                playerRef = simTask.findGLO(playerName);
            }
        }

        /*
         * we are going to add the player to the room's
         * player list.  For that we need the room GLO so
         * we ask the GLOReference to the room to get the GLO
         * itself for us.
         */
        Room roomGLO= roomRef.get(simTask);
        roomGLO.addPlayer(playerRef);

        /*
         * When we get a userLeft message we are going to want to
         * remove the player from the room.  In order to do that we
         * will need to get the GLOReference to the player listening
         * to that UID.  We do that by storing that relationship in a
         * map for later lookup.
         */
        uidToPlayerRef.put(uid, playerRef);

        /*
         * Finally we tell the system that we want the playerGLO we
         * created to receive UserDataListener events reating to the
         * user we hav associated with that Player GLO.
         */
        simTask.addUserDataListener(uid,playerRef);
         String out = "Sworld welcomes "+playerName;
         ByteBuffer outbuff = ByteBuffer.allocate(out.length());
         outbuff.put(out.getBytes());
         simTask.broadcastData(appChannel, outbuff, true);
    }

    /**
     * This callback gets called to tell us a User's session has ended
     * because they are disconnected/logged out from the system.
     *
     * @param uid the user to remove
     *
     * @see
     * com.sun.gi.logic.SimUserListener#userLeft(com.sun.gi.comm.routing.UserID)
     */
    public void userLeft(UserID uid) {
        SimTask simTask = SimTask.getCurrent();

        /*
         * We fetch the GLOReference to the player who was listening
         * to SimUserData events for this user from the map where we
         * stored that information in userJoined and then remove it
         * from the map because it is going away.
         */
        GLOReference<Player> playerRef = uidToPlayerRef.remove(uid);

        /*
         * We fetch the roomGLO so we can tell it he left, and remove
         * the Player GLO from the Room's list of players.
         */
        Room roomGLO= roomRef.get(simTask);
        roomGLO.removePlayer(playerRef);
    }
}
```

Room.java

```
/*
 Copyright (c) 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 Clara, California 95054, U.S.A. All rights reserved.

 Sun Microsystems, Inc. has intellectual property rights relating to
 technology embodied in the product that is described in this document.
 In particular, and without limitation, these intellectual property rights
```

```java
/**
 * <p>Title: Room.java</p>
 * <p>Description: </p>
 *
 * @author Jeff Kesselman
 * @version 1.0
 */
package com.sun.gi.apps.swordworld;

import java.util.LinkedList;
import java.util.List;

import com.sun.gi.logic.GLO;
import com.sun.gi.logic.GLOReference;
import com.sun.gi.logic.SimTask;

/**
 * The Game Logic Class (GLC) that defines Room Game Logic Objects (GLOs) in our
 * toy MUD example.
 * <p>
 *
 * In this toy example a room is simply a container that has a description, a
 * list of items in the room, and a list of players in the room.
 * <p>
 *
 * @author Jeff Kesselman
 *
 * @version 1.0
 */
public class Room implements GLO {

        /**
         * All GLOs should define a <code>serialVersionUID</code> because they are
         * serialized. This turns off version checking so we can change the class
         * and still load old data that might already be in the ObjectStore.
         */
        private static final long serialVersionUID = 1L;

        /**
         * The description of the room itself.
         */
        private String description;

        /**
         * The list of items in the room. All items are GLOs that are instances of
         * the GLC "RoomObject"
         */
        private List<GLOReference<RoomObject>> inventory = new
LinkedList<GLOReference<RoomObject>>();

        /**
         * The list of players in the room. All players are GLOs that are instances
         * of the GLC "Player"
         */
        private List<GLOReference<Player>> players = new LinkedList<GLOReference<Player>>();

        /**
         * Creates a Room instance.
         *
         * @param description
         *            the description of the room
         */
        public Room(String description) {
                this.description = description;
        }

        /**
         * The only command rooms currently support. It returns a text string that
         * describes the contents of the Room in which the player is.
         * <p>
         *
         * The parameter is a GLOReference to the Player asking for the list, which
         * is used to make sure that we do not put that player in the list of
```

```
     * "others in the room". Note that this method explicitly assumes that the
     * asking player is in the room, and phrases its responses according to this
     * assumption (i.e., "You are in ..." and "You are alone").
     *
     * @param meRef
     *            A GLOReference to the player looking at the room
     *
     * @return text description of the contents of the room
     */
    public String getDescription(GLOReference<Player> meRef) {

        /*
         * We are going to reference things relative to a simTask, so we get the
         * current one here.
         */
        SimTask simTask = SimTask.getCurrent();

        // Initialize the string with the description of the room
        String out = "You are in " + description;

        /*
         * Add either a list of the descriptions off the RoomObjects referred to
         * by the GLOReferences in the inventory list, or, if the list is empty,
         * a statement that there are none.
         */
        out += " containing";
        if (inventory.isEmpty()) {
            out += " nothing.";
        } else {
            out += ":\n";

            /*
             * Loop through all the GLOReferences in the inventory list. For
             * each reference, get the RoomObject GLO, and add its description.
             */
            for (GLOReference<RoomObject> objectRef : inventory) {
                RoomObject objectGLO = objectRef.get(simTask);
                out += objectGLO.getDescription() + "\n";
            }
        }

        /*
         * Now do the same thing with the Player GLOReferences in the player
         * list, with the exception that we skip the given player from the
         * string.
         */
        if (players.size() == 1) {
            // only one player here, must be us.
            out += "You are alone";
        } else {
            out += "With you in the room are:\n";

            /*
             * Loop through the GLORefrences in the players list. If the
             * GLOReference in the list is equal to the GLOreference passed in,
             * then it is a reference to the "looker" and skip it.
             */
            for (GLOReference<Player> playerRef : players) {
                if (!playerRef.equals(meRef)) {
                    Player playerGLO = playerRef.get(simTask);
                    out += playerGLO.getName() + "\n";
                }
            }
        }
        return out;
    }

    /**
     * Adds an item (referenced via its GLOReference) to the inventory of a
     * Room.
     *
     * @param swordRef
     *            the item to add
     */
```

```
        public void addToInventory(GLOReference<RoomObject> swordRef) {
               inventory.add(swordRef);
        }

        /**
         * Adds a GLOReference to a Player GLO into the players list. It is called
         * by the SwordWorldBoot GLO when a player logs into the system.
         *
         * @param playerRef
         *            the player to add
         */
        public void addPlayer(GLOReference<Player> playerRef) {
               players.add(playerRef);
               SimTask simTask = SimTask.getCurrent();
               Player player = playerRef.get(simTask);

            player.setCurrentRoom(simTask.lookupReferenceFor(this));
               player.setCurrentRoom(simTask.lookupReferenceFor(this));

        }

        /**
         * Removes a GLOReference to a Player GLO from the players list. It is
         * called by the SwordWorldBoot GLO when a player logs out or disconnects
         * from the system.
         *
         * @param playerRef
         *            the player to remove
         */
        public void removePlayer(GLOReference<Player> playerRef) {
               players.remove(playerRef);
               SimTask simTask = SimTask.getCurrent();
               Player player = playerRef.get(simTask);
               player.setCurrentRoom(null);
        }
}
```

RoomObject.java

```
 ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE
 LEGALLY INVALID.

 Copyright © 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara,
 California 95054, Etats-Unis. Tous droits réservés.

 Sun Microsystems, Inc. détient les droits de propriété intellectuels
 relatifs à la technologie incorporée dans le produit qui est décrit dans
 ce document. En particulier, et ce sans limitation, ces droits de
 propriété intellectuelle peuvent inclure un ou plus des brevets américains
 listés à l'adresse http://www.sun.com/patents et un ou les brevets
 supplémentaires ou les applications de brevet en attente aux Etats -
 Unis et dans les autres pays.

 Cette distribution peut comprendre des composants développés par des
 tierces parties.

 Sun, Sun Microsystems, le logo Sun et Java sont des marques de fabrique
 ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans
 d'autres pays.

 UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et
 licenciée exlusivement par X/Open Company, Ltd.

 see above Les produits qui font l'objet de ce manuel d'entretien et les
 informations qu'il contient sont regis par la legislation americaine en
 matiere de controle des exportations et peuvent etre soumis au droit
 d'autres pays dans le domaine des exportations et importations.
 Les utilisations finales, ou utilisateurs finaux, pour des armes
 nucleaires, des missiles, des armes biologiques et chimiques ou du
 nucleaire maritime, directement ou indirectement, sont strictement
 interdites. Les exportations ou reexportations vers des pays sous embargo
 des Etats-Unis, ou vers des entites figurant sur les listes d'exclusion
 d'exportation americaines, y compris, mais de maniere non exclusive, la
 liste de personnes qui font objet d'un ordre de ne pas participer, d'une
 facon directe ou indirecte, aux exportations des produits ou des services
 qui sont regi par la legislation americaine en matiere de controle des
 exportations et la liste de ressortissants specifiquement designes, sont
 rigoureusement interdites.

 LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS,
 DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES,
 DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE
 GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE
 UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.
*/

package com.sun.gi.apps.swordworld;

import com.sun.gi.logic.GLO;

/**
 * Implements Game Logic Objects that represent items in the Room.
 * Currently only one item is created on startup by SwordWorldBoot-- a
 * bright, shiny new sword.  (Hence the name of the demo.)
 *
 * @author Jeff Kesselman
 *
 * @version 1.0
 */
public class RoomObject implements GLO {
    /**
     * All GLOs should define a <code>serialVersionUID</code> because
     * they are serialized.  This turns off version checking so we can
     * change the class and still load old data that might already be
     * in the ObjectStore.
     */
    private static final long serialVersionUID = 1L;

    /**
     * The description of the item.
     */
    private String description;
    /**
     * Creates a RoomObject with the given description.
     *
     * @param description a description of the object
     */
    public RoomObject(String description) {
           this.description = description;
    }

    /**
     * Returns the description of the item. <p>
     *
     * Used by the {@link Room} GLO in order to assist
     * it in putting together a list of room contents.
     *
     * @return The description of the item
     */
    public String getDescription() {
       return description;
    }
}
```

Player.java

```
 tierces parties.

 Sun, Sun Microsystems, le logo Sun et Java sont des marques de fabrique
 ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans
 d'autres pays.

 UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et
 licenciée exlusivement par X/Open Company, Ltd.

 see above Les produits qui font l'objet de ce manuel d'entretien et les
 informations qu'il contient sont regis par la legislation americaine en
 matiere de controle des exportations et peuvent etre soumis au droit
 d'autres pays dans le domaine des exportations et importations.
 Les utilisations finales, ou utilisateurs finaux, pour des armes
 nucleaires, des missiles, des armes biologiques et chimiques ou du
 nucleaire maritime, directement ou indirectement, sont strictement
 interdites. Les exportations ou reexportations vers des pays sous embargo
 des Etats-Unis, ou vers des entites figurant sur les listes d'exclusion
 d'exportation americaines, y compris, mais de maniere non exclusive, la
 liste de personnes qui font objet d'un ordre de ne pas participer, d'une
 facon directe ou indirecte, aux exportations des produits ou des services
 qui sont regi par la legislation americaine en matiere de controle des
 exportations et la liste de ressortissants specifiquement designes, sont
 rigoureusement interdites.

 LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS,
 DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES,
 DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE
 GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE
 UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.
*/


package com.sun.gi.apps.swordworld;

import java.nio.ByteBuffer;

import com.sun.gi.comm.routing.ChannelID;
import com.sun.gi.comm.routing.UserID;
import com.sun.gi.logic.GLO;
import com.sun.gi.logic.GLOReference;
import com.sun.gi.logic.SimTask;
import com.sun.gi.logic.SimUserDataListener;

/**
 * The class is the Game Logic Class that defines Player-GLOs in the
 * SwordWorld demo.  <p>
 *
 * Player GLOs are a very important concept in the SGS.  They act like
 * the user's "body" in the world of the GLOs.  They receieve commands
 * from the client program and based on those commands they act on and
 * change other GLOs.  <p>
 *
 * In order to be a Player-GLO, a GLO's defining GLC must implement
 * the {@link SimUserDataListener} event and the GLO must be
 * registered as the {@link SimUserDataListener} for that user's
 * session.  <p>
 *
 * See {@link SwordWorldBoot#userJoined()} to see how registration of
 * a SimUserDataListener is accomplished.  <p>
 *
 * @see SwordWorldBoot#userJoined()
 *
 * @author Jeff Kesselman
 *
 * @version 1.0
 */
public class Player implements SimUserDataListener {

    /**
     * All GLOs should define a <code>serialVersionUID</code> because
     * they are serialized.  This turns off version checking so we can
     * change the class and still load old data that might already be
     * in the ObjectStore.
     */
    private static final long serialVersionUID = 1L;

    /**
     * For extra debugging verbosity.
     */
    private boolean debug = true;

    /**
     * The name of the player, used by Room to
     * help create the room description.
     */
    String name;

    /**
     * A GLOReference to our current Room.  This is
     * so we can be removed from the room when we leave.
     */
    GLOReference<Room> currentRoomRef;

    /**
     * In the current version of the API, the only way for
     * the server to send data back down to the client is through
     * a channel.  (this is likely to chnage in future versions)
     * For now, both the client and the SwordWorldBoot open the
     * channel "GAMECHANNEL".  This is a field to store the
     * ChannelID that SwordWorldBoot was given for that channel.
     */
    private ChannelID appChannel;

    /**
     * Creates a player GLO for this game.
     *
     * @param name the screen name of the player
     */
    public Player(String name, ChannelID cid) {
       this.name = name;
       this.appChannel = cid;
    }

    /**
     * This is called to set what room the player is in.
     *
     * @param room a GLOReference to the room
     */
    public void setCurrentRoom(GLOReference<Room> room) {
       currentRoomRef = room;
    }

    /**
     * This is the callback that is called when data arrives at the
     * server from the user for which we registered ourselves as the
     * SimUserDataListener.
     *
     * @see
     * com.sun.gi.logic.SimUserDataListener#userDataReceived(com.sun.gi.comm.routing.UserID,
     * java.nio.ByteBuffer)
     */
    public void userDataReceived(UserID from, ByteBuffer data) {

       /*
        * We are going to reference things relative to a simTask, so
        * we get the current one here.
        */
       SimTask simTask = SimTask.getCurrent();

       /*
        * In this app, all the data are string commands so we read
        * all the bytes from the buffer and turn them back into a
        * string and then split the string into the individual
        * tokens.
        */
       byte[] inbytes = new byte[data.remaining()];
       data.get(inbytes);
```

```
        String commandString = new String(inbytes).trim();

        if (debug) {
            System.out.println("CommandString = " + commandString);
        }

        /*
         * we split up the string into indvidual words
         */
        String[] words = StringUtils.explode(commandString," ");

        /*
         * words[0] is the basic command.  The only command
         * implemented by this app is "look"
         */

        if (words[0].equalsIgnoreCase("look")){

            /*
             * we get the actual Room GLO for the room we are
             * currently in.
             */

            Room roomGLO = currentRoomRef.get(simTask);

               /*
                * the getDescription command requires we pass in a
                * GLOReference to ourselves so it knows who not to
                * put in the description.  We do not already have a
                * GLOReference to ourselves, but we can use the
                * SimTask to find it.
                */

               GLOReference<Player> myRef = simTask.lookupReferenceFor(this);

               /*
                * We ask the Room GLO for the description.
                */
               String out = roomGLO.getDescription(myRef);

               /*
                * Write the string description to a ByteBuffer so we
                * can send it back to the client.
                */
               ByteBuffer outbuff = ByteBuffer.allocate(out.length());
               outbuff.put(out.getBytes());

               /*
                * Use the SimTask to send the data to the client
                * from whom we received the command.  We send it on
                * the "GAMECHANNEL" channel that SwordWorldBoot
                * opened and told us about.  We want gauranteed
                * delivery of the message, so we set the reliable
                * delivery flag to true.
                */
               simTask.sendData(appChannel, from, outbuff, true);



        } else {
            if (debug) {
               System.out.println("unknown command: " + words[0]);
            }
        }
    }

    /**
     * A callback telling us when the user who we are registered as
     * the SimUserDataListener for joins a channel.  We do not need that
     * information in this app, so we ignore it.
     *
     * @see
com.sun.gi.logic.SimUserDataListener#userJoinedChannel(com.sun.gi.comm.routing.ChannelID,
     * com.sun.gi.comm.routing.UserID)
     */
    public void userJoinedChannel(ChannelID cid, UserID uid) {
       // Not needed for this app.
    }

    /**
     * A callabck telling us when the user who we are registered as
     * the SimUserDataListener for leaves a channel.  We dont need
     * that information in this app so we ignore it.
     *
     * @see
com.sun.gi.logic.SimUserDataListener#userLeftChannel(com.sun.gi.comm.routing.ChannelID,
     * com.sun.gi.comm.routing.UserID)
     */
    public void userLeftChannel(ChannelID cid, UserID uid) {
       // Not needed for this app.
    }

    /**
     * The server can "snoop" channels by using the SimTask to tell
     * the SGS to call this method every time data is sent by a
     * particular user on a particular channel.  We do not need this
     * functionality for this app, so we leave this callback empty.
     *
     * @see
     *
com.sun.gi.logic.SimUserDataListener#dataArrivedFromChannel(com.sun.gi.comm.routing.ChannelID,
     * com.sun.gi.comm.routing.UserID, java.nio.ByteBuffer)
     */
    public void dataArrivedFromChannel(ChannelID cid, UserID from,
           ByteBuffer buff) {
       // Not needed for this app.
    }

    /**
     * This method returns our player name.  Its used when others ask
     * the Room GLO to describe the room.
     *
     * @return our player name
     */
    public String getName() {
       return name;
    }
}
```

StringUtils.java

```
/*
 Copyright (c) 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 Clara, California 95054, U.S.A. All rights reserved.

 Sun Microsystems, Inc. has intellectual property rights relating to
 technology embodied in the product that is described in this document.
 In particular, and without limitation, these intellectual property rights
 may include one or more of the U.S. patents listed at
 http://www.sun.com/patents and one or more additional patents or pending
 patent applications in the U.S. and in other countries.

 U.S. Government Rights - Commercial software. Government users are subject
 to the Sun Microsystems, Inc. standard license agreement and applicable
 provisions of the FAR and its supplements.

 This distribution may include materials developed by third parties.

 Sun, Sun Microsystems, the Sun logo and Java are trademarks or registered
 trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

 UNIX is a registered trademark in the U.S. and other countries, exclusively
 licensed through X/Open Company, Ltd.

 Products covered by and information contained in this service manual are
 controlled by U.S. Export Control laws and may be subject to the export
 or import laws in other countries. Nuclear, missile, chemical biological
 weapons or nuclear maritime end uses or end users, whether direct or
 indirect, are strictly prohibited. Export or reexport to countries subject
 to U.S. embargo or to entities identified on U.S. export exclusion lists,
 including, but not limited to, the denied persons and specially designated
 nationals lists is strictly prohibited.

 DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS,
 REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF
 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT,
 ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE
 LEGALLY INVALID.

 Copyright © 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara,
 California 95054, Etats-Unis. Tous droits réservés.

 Sun Microsystems, Inc. détient les droits de propriété intellectuels
 relatifs à la technologie incorporée dans le produit qui est décrit dans
 ce document. En particulier, et ce sans limitation, ces droits de
 propriété intellectuelle peuvent inclure un ou plus des brevets américains
 listés à l'adresse http://www.sun.com/patents et un ou les brevets
 supplémentaires ou les applications de brevet en attente aux Etats -
 Unis et dans les autres pays.

 Cette distribution peut comprendre des composants développés par des
 tierces parties.

 Sun, Sun Microsystems, le logo Sun et Java sont des marques de fabrique
 ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans
 d'autres pays.

 UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et
 licenciée exlusivement par X/Open Company, Ltd.

 see above Les produits qui font l'objet de ce manuel d'entretien et les
 informations qu'il contient sont regis par la legislation americaine en
 matiere de controle des exportations et peuvent etre soumis au droit
 d'autres pays dans le domaine des exportations et importations.
 Les utilisations finales, ou utilisateurs finaux, pour des armes
 nucleaires, des missiles, des armes biologiques et chimiques ou du
 nucleaire maritime, directement ou indirectement, sont strictement
 interdites. Les exportations ou reexportations vers des pays sous embargo
 des Etats-Unis, ou vers des entites figurant sur les listes d'exclusion
 d'exportation americaines, y compris, mais de maniere non exclusive, la
 liste de personnes qui font objet d'un ordre de ne pas participer, d'une
 facon directe ou indirecte, aux exportations des produits ou des services
 qui sont regi par la legislation americaine en matiere de controle des
 exportations et la liste de ressortissants specifiquement designes, sont
 rigoureusement interdites.

 LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS,
 DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES,
 DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE
 GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE
 UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.
*/

package com.sun.gi.apps.swordworld;

import java.util.ArrayList;

/**
 * Utility class containing string utilities used by the {@link
 * Player} object to break up the input string from the client into
 * tokens.
 *
 * @author Jeff Kesselman @version 1.0
 */
public abstract class StringUtils {

    /**
     * Breaks up a string into sub-strings delimited by the given
     * delimiter string.  Ignores any instances of the delimiter at
     * the start and end of the string.
     *
     * @param str The string to break up
     *
     * @param delim The string to be used as a delimiter (most
     * commonly " ")
     *
     * @return the list of sub-strings
     */
    public static String[] explode(String str, String delim) {
        return explode(str, delim, true);
    }

    /**
     * Breaks up a string into sub-strings delimited by the given
     * delimiter string.  If <code>trim</code> is true then it also
     * trims any instances of the delimiter off the start and end of
     * the string before breaking it.
     *
     * @param str The string to break up
     *
     * @param delim The string to be used as a delimiter (most
     * commonly " ")
     *
     * @param trim if <code>true</code>, remove remove leading and
     * trailing instances of the delimeter
     *
     * @return the list of sub-strings
     */
    public static String[] explode(String str, String delim, boolean trim) {
        ArrayList<String> plist = new ArrayList<String>();
        if (str.indexOf(delim) == -1){
           return new String[] { str };
        }
        while (str.length() > 0) {
            int delimpos = str.indexOf(delim);

            String leftString = str.substring(0, delimpos);
            if (leftString.length() > 0) {
                if (trim) {
                    leftString = leftString.trim();
                }
                plist.add(leftString);
            }
            str = str.substring(delimpos + delim.length());
        }
        String[] retarray = new String[plist.size()];
        return plist.toArray(retarray);
    }
}
```

SwordWorld Client

This is a very simple textual GUI client that connects to the SwordWorld server and allows you to enter commands and see responses. SwordWorld as listed below supports one whole command: "look"

SwordWorldClient.java

```
/*
 Copyright (c) 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 Clara, California 95054, U.S.A. All rights reserved.

 Sun Microsystems, Inc. has intellectual property rights relating to
 technology embodied in the product that is described in this document.
 In particular, and without limitation, these intellectual property rights
 may include one or more of the U.S. patents listed at
 http://www.sun.com/patents and one or more additional patents or pending
 patent applications in the U.S. and in other countries.

 U.S. Government Rights - Commercial software. Government users are subject
```

```
package com.sun.gi.apps.swordworld.client;

import com.sun.gi.comm.discovery.impl.URLDiscoverer;
import com.sun.gi.comm.users.client.ClientAlreadyConnectedException;
import com.sun.gi.comm.users.client.ClientChannel;
import com.sun.gi.comm.users.client.ClientChannelListener;
import com.sun.gi.comm.users.client.ClientConnectionManager;
import com.sun.gi.comm.users.client.ClientConnectionManagerListener;
import com.sun.gi.comm.users.client.impl.ClientConnectionManagerImpl;
import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.net.MalformedURLException;
import java.net.URL;
import java.nio.ByteBuffer;
import javax.security.auth.callback.Callback;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;

/**
 */
public class SwordWorldClient extends JFrame
       implements ClientConnectionManagerListener
{
    private static final long serialVersionUID = 1L;

    JTextArea outputArea = new JTextArea();
    ClientConnectionManager mgr;

    public SwordWorldClient(String discoveryURL)
       throws MalformedURLException, ClientAlreadyConnectedException{
       super("SwordWorldClient");
       Container c = getContentPane();
       c.setLayout(new BorderLayout());
       final JTextField inputField = new JTextField();
       inputField.addActionListener(new ActionListener() {

           public void actionPerformed(ActionEvent e) {
              sendCommand(inputField.getText());
               inputField.setText("");
           }

       });
       c.add(inputField,BorderLayout.SOUTH);
       outputArea.setEditable(false);
       c.add(new JScrollPane(outputArea),BorderLayout.CENTER);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       pack();
       setSize(400,400);
       setVisible(true);
       connect(discoveryURL);
    }

    /**
     * Sends a command to the server.
     *
     * @param text the command to send
     */
    protected void sendCommand(String text) {
       ByteBuffer buff = ByteBuffer.allocate(text.length());
       buff.put(text.getBytes());
       mgr.sendToServer(buff,true);
    }

    public void connect(String discoveryURL)
           throws MalformedURLException, ClientAlreadyConnectedException {
       mgr = new ClientConnectionManagerImpl("SwordWorld",
              new URLDiscoverer(new URL(discoveryURL)));
       mgr.setListener(this);
       mgr.connect("com.sun.gi.comm.users.client.impl.TCPIPUserManagerClient");
    }

    //All the below are methods defined by ClientConnectionManagerListener

    /** {@inheritDoc} */
```

```
public void validationRequest(Callback[] callbacks) {
   ValidatorDialog dialog = new ValidatorDialog(this,callbacks);
   mgr.sendValidationResponse(callbacks);
}

/** {@inheritDoc} */
public void connected(byte[] myID) {
   System.out.println("Connected");
   mgr.openChannel("GAMECHANNEL");
}

/** {@inheritDoc} */
public void connectionRefused(String message) {
   System.out.println("Connection failed: " + message);
   System.exit(1);
}

/** {@inheritDoc} */
public void failOverInProgress() {
}

/** {@inheritDoc} */
public void reconnected() {
}

/** {@inheritDoc} */
public void disconnected() {
   System.out.println("Disconnected");
   System.exit(2);
}

/** {@inheritDoc} */
public void userJoined(byte[] userID) {
}

/** {@inheritDoc} */
public void userLeft(byte[] userID) {
}

/** {@inheritDoc} */
public void joinedChannel(ClientChannel channel) {
   channel.setListener(new ClientChannelListener() {

       /** {@inheritDoc} */
       public void playerJoined(byte[] playerID) {
       }

       /** {@inheritDoc} */
       public void playerLeft(byte[] playerID) {
       }

       /** {@inheritDoc} */
       public void dataArrived(byte[] from, ByteBuffer data,
              boolean reliable) {
          byte[] inbytes = new byte[data.remaining()];
          data.get(inbytes);
          outputArea.append(new String(inbytes));
          outputArea.append("\n");
       }

       /** {@inheritDoc} */
       public void channelClosed() {
       }
   });
}

/** {@inheritDoc} */
public void channelLocked(String channelName, byte[] userID) {
}

/**
 * @param args
 */
public static void main(String[] args) {
        if (args.length == 0){
            args = new String[1];
            args[0]="file:discovery.xml";
        }
        try {
            new SwordWorldClient(args[0]);
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (ClientAlreadyConnectedException e) {
            e.printStackTrace();
        }
    }
}
```

Validator Dialog.java

```
/*
 Copyright (c) 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa
 Clara, California 95054, U.S.A. All rights reserved.

 Sun Microsystems, Inc. has intellectual property rights relating to
 technology embodied in the product that is described in this document.
 In particular, and without limitation, these intellectual property rights
 may include one or more of the U.S. patents listed at
 http://www.sun.com/patents and one or more additional patents or pending
 patent applications in the U.S. and in other countries.

 U.S. Government Rights - Commercial software. Government users are subject
 to the Sun Microsystems, Inc. standard license agreement and applicable
 provisions of the FAR and its supplements.

 This distribution may include materials developed by third parties.

 Sun, Sun Microsystems, the Sun logo and Java are trademarks or registered
 trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

 UNIX is a registered trademark in the U.S. and other countries, exclusively
 licensed through X/Open Company, Ltd.

 Products covered by and information contained in this service manual are
 controlled by U.S. Export Control laws and may be subject to the export
 or import laws in other countries. Nuclear, missile, chemical biological
 weapons or nuclear maritime end uses or end users, whether direct or
 indirect, are strictly prohibited. Export or reexport to countries subject
 to U.S. embargo or to entities identified on U.S. export exclusion lists,
 including, but not limited to, the denied persons and specially designated
 nationals lists is strictly prohibited.

 DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS,
 REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF
 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT,
 ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE
 LEGALLY INVALID.

 Copyright © 2006 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara,
 California 95054, Etats-Unis. Tous droits réservés.

 Sun Microsystems, Inc. détient les droits de propriété intellectuels
 relatifs à la technologie incorporée dans le produit qui est décrit dans
 ce document. En particulier, et ce sans limitation, ces droits de
 propriété intellectuelle peuvent inclure un ou plus des brevets américains
 listés à l'adresse http://www.sun.com/patents et un ou les brevets
 supplémentaires ou les applications de brevet en attente aux Etats -
 Unis et dans les autres pays.

 Cette distribution peut comprendre des composants développés par des
 tierces parties.

 Sun, Sun Microsystems, le logo Sun et Java sont des marques de fabrique
 ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans
 d'autres pays.

 UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et
 licenciée exlusivement par X/Open Company, Ltd.

 see above Les produits qui font l'objet de ce manuel d'entretien et les
 informations qu'il contient sont regis par la legislation americaine en
 matiere de controle des exportations et peuvent etre soumis au droit
 d'autres pays dans le domaine des exportations et importations.
 Les utilisations finales, ou utilisateurs finaux, pour des armes
 nucleaires, des missiles, des armes biologiques et chimiques ou du
 nucleaire maritime, directement ou indirectement, sont strictement
 interdites. Les exportations ou reexportations vers des pays sous embargo
 des Etats-Unis, ou vers des entites figurant sur les listes d'exclusion
 d'exportation americaines, y compris, mais de maniere non exclusive, la
 liste de personnes qui font objet d'un ordre de ne pas participer, d'une
 facon directe ou indirecte, aux exportations des produits ou des services
 qui sont regi par la legislation americaine en matiere de controle des
 exportations et la liste de ressortissants specifiquement designes, sont
 rigoureusement interdites.

 LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS,
 DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES,
 DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE
 GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE
 UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.
*/

package com.sun.gi.apps.swordworld.client;

import java.awt.BorderLayout;
import java.awt.Component;
import java.awt.Container;
import java.awt.Frame;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.ChoiceCallback;
import javax.security.auth.callback.ConfirmationCallback;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.callback.TextInputCallback;
import javax.security.auth.callback.TextOutputCallback;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JPasswordField;
import javax.swing.JTextField;

/**
 * A Swing GUI for server validation fulfillment.  <p>
 *
 * When connecting to a server application via a UserManager, the
 * UserManager will attempt to validate the user based on the
 * applications validation settings as specified in its deployment
 * descriptor.  In the case of the SwordWorld application, a name and
 * a password are required.
 *
 */
public class ValidatorDialog extends JDialog {

    private static final long serialVersionUID = 1L;

    // The array of javax.security.auth.callbacks.Callbacks
    Callback[] callbacks;

    // an array of UI components that parallel callbacks[] in size and order.
    List<Component> dataFields = new ArrayList<Component>();

    /**
     * Creates a new ValidatorDialog for the given frame and
     * callbacks.  <p>
     *
     * The dialog iterates through the Callback array and displays the
     * appropriate UI based on the type of Callback.  In the case of
     * CommTest, the Callbacks are of type NameCallback and
     * PasswordCallback.  This causes both a "username" textfield, and
     * a "password" input field to be rendered on the dialog.
     *
     * @param parent the dialog's parent frame
     *
     * @param cbs an array of Callbacks
     */
    public ValidatorDialog(Frame parent, Callback[] cbs) {
        super(parent, "Validation Information Required", true);
        callbacks = cbs;
        Container c = getContentPane();
        JPanel validationPanel = new JPanel();
        validationPanel.setLayout(new GridLayout(2, 0));
        c.add(validationPanel, BorderLayout.NORTH);
        JButton validateButton = new JButton("CONTINUE");
        c.add(validateButton, BorderLayout.SOUTH);

        /*
         * when pressed, set the data from the UI components to the
         * matching Callbacks.
         */
        validateButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                transcribeToCallbacks();
                ValidatorDialog.this.setVisible(false);
                ValidatorDialog.this.getParent().remove(ValidatorDialog.this);
            }
        });

        /*
         * Iterate through the javax.security.auth.callback.Callbacks
         * and render the appropriate UI accordingly.  For each
         * Callback, the matching UI Component is stored in the
         * dataFields array.  The order is important, as they will be
         * retrieved along side their matching Callback.
         */
        for (Callback cb : cbs) {
            if (cb instanceof ChoiceCallback) {
                ChoiceCallback ccb = (ChoiceCallback) cb;
                validationPanel.add(new JLabel(ccb.getPrompt()));
                JComboBox combo = new JComboBox(ccb.getChoices());
                combo.setSelectedItem(ccb.getDefaultChoice());
                validationPanel.add(combo);
                dataFields.add(combo);
            } else if (cb instanceof ConfirmationCallback) {
                ConfirmationCallback ccb = (ConfirmationCallback) cb;
                validationPanel.add(new JLabel(ccb.getPrompt()));
                JComboBox combo = new JComboBox(ccb.getOptions());
                combo.setSelectedItem(ccb.getDefaultOption());
                validationPanel.add(combo);
                dataFields.add(combo);
            } else if (cb instanceof NameCallback) {
                NameCallback ncb = (NameCallback) cb;
                validationPanel.add(new JLabel(ncb.getPrompt()));
                JTextField nameField = new JTextField(ncb.getDefaultName());
                validationPanel.add(nameField);
                dataFields.add(nameField);
            } else if (cb instanceof PasswordCallback) {
                PasswordCallback ncb = (PasswordCallback) cb;
                validationPanel.add(new JLabel(ncb.getPrompt()));
                JPasswordField passwordField = new JPasswordField();
                validationPanel.add(passwordField);
                dataFields.add(passwordField);
            } else if (cb instanceof TextInputCallback) {
                TextInputCallback tcb = (TextInputCallback) cb;
                validationPanel.add(new JLabel(tcb.getPrompt()));
                JTextField textField = new JTextField(tcb.getDefaultText());
                validationPanel.add(textField);
                dataFields.add(textField);
            } else if (cb instanceof TextOutputCallback) {
                TextOutputCallback tcb = (TextOutputCallback) cb;
                validationPanel.add(new JLabel(tcb.getMessage()));
            }
        }
        pack();
        setVisible(true);
    }

    /**
     * Called when the validation button is pressed.  It iterates
     * through the array of Callbacks and takes the data from the
     * matching UI component and sets it in the Callback.
     */
    protected void transcribeToCallbacks() {
        Iterator iter = dataFields.iterator();
        for (Callback cb : callbacks) {
            if (cb instanceof ChoiceCallback) {
                // note this is really wrong, should allow for multiple
                // select
                ChoiceCallback ccb = (ChoiceCallback) cb;
                JComboBox combo = (JComboBox) iter.next();
                ccb.setSelectedIndex(combo.getSelectedIndex());
            } else if (cb instanceof ConfirmationCallback) {
                ConfirmationCallback ccb = (ConfirmationCallback) cb;
                JComboBox combo = (JComboBox) iter.next();
                ccb.setSelectedIndex(combo.getSelectedIndex());
            } else if (cb instanceof NameCallback) {
                NameCallback ncb = (NameCallback) cb;
                JTextField nameField = (JTextField) iter.next();
                ncb.setName(nameField.getText());
            } else if (cb instanceof PasswordCallback) {
                PasswordCallback ncb = (PasswordCallback) cb;
                JPasswordField passwordField = (JPasswordField) iter.next();
                ncb.setPassword(passwordField.getPassword());
            } else if (cb instanceof TextInputCallback) {
                TextInputCallback tcb = (TextInputCallback) cb;
                JTextField textField = (JTextField) iter.next();
                tcb.setText(textField.getText());
            } else if (cb instanceof TextOutputCallback) {
                // no response required
            }
        }
    }
}
```