Final Project: Universe

The Assignment

In your project you should design and implement an interactive graphics-based game for one or more players, where the player communicates with other players or with the game administrator over the Internet.

The game administrator controls the server in this network architecture. That means, it starts first, opens up a network port and listens for anyone who would like to connect to the port and send messages to it. We implement the game administrator's actions by defining a class that extends the `Universe`.

The game participant designs the game as a class that extends the `World` and includes the methods that respond to clock ticks and to the key presses, as we have done before. Additionally, the game participant starts by connecting to the *universe* server, indicating the desire to join the game, reacts to messages sent to it from the server, and sends other massages to the server, as the game progresses.

A Sample Game

To illustrate how a simple game can be implemented we include a complete code for a simple game together with an explanation of its design.

The player's only action is to roll a die. To illustrate the graphic display during the game and the use of the `onTick` method, the game progress is shown by flashing dot that changes its color on each tick, and displays the last roll of the die as a `String`.

The player starts by connecting to the server. When it receives a message *roll* from the server, it marks its state as `ready`, indicating that the human player can now roll a die. Currently, the player only hits the space bar and the program generates the random number that is rolled. If the reply to the server generated the next roll immediately, the game would proceed very fast and humans would have no way to observe the progress.

When the player rolls two numbers that are the same in a row, the server sends a congratulatory message – and the player shows this as rolling 1000.

The universe allows an arbitrary number of players. For each player it remembers the last die that was rolled, accepts new rolls, checks if it was a repeat of the previous roll, and sends messages back to the player when it is ready to receive the next roll.

The tick count is used solely to limit the duration of the game. At the end of the time the `universe` disconnects all players and the game ends.

The Universe

We show the complete code for the class that extends the `Universe`. The `abstract class Universe` is parametrized by two types, the players type (this is a class that extends `Player`), and the type of the messages that will be sent between the player and the universe, which must extend the `java.io.Serializable` interface. The simplest type of message is a `String` and it requires no additional work.

Once you are comfortable with the `String` messages, you may explore how to send more complex messages that can represent entire objects or a collection of objects.

```
abstract class Universe<P extends Player,
                        M extends java.io.Serializable>>
```

When defining the class that extends the `Universe`, you also need to define how the *universe* is going to keep track of its connections, *the players*. So, you may need to define a class that extends the `Player` class. The `Player` class already in the library gives each player a name (`String name`) and defined two methods that can be used by the `Universe`:

```
/** Disconnect this player from the universe.
  */
public void disconnect();

/** Send the given message to this player
  * @param msg the message to send
  */
public void sendMessage(java.io.Serializable msg);
```

The following methods are declared as `abstract` in the `Universe` class and must be implemented in the class that extends it:

```
/** Called by the Universe when a new Player object
 *  is needed The subclass can use this method to set
 *  the initial state of the Player
 *  @return a new instance of the class that extends Player
 */
public P initializePlayer();
```

```
 /** Called when a new player connects.
   * The Universe state may need to note
   * the number of players or some other information.
   * @param player the new player
   */
 public void onConnect(P player);

 /** Called when a new player disconnects.
   * The Universe state may need to note
   * the number of players or some other information.
   * @param player the player who disconnected
   */
 public void onDisconnect(P player);

 /** Called when a message from a player is received.
   * The Universe processes the massage and may then
   * send one or more messages to this or other players.
   * @param message te message that has been delivered
   * @param player the player who sent the message
   */
 public void onReceive(M message, P player);

 /** What the universe does on each tick
   */
 public void onTick();
```

The following code shows a simple implementation of the class that extends `Universe`:

```
import edu.neu.universe.*;

/**
 * A simple example of a game server that accepts
 * an arbitrary number of players, keeps track of
 * the tick count, and ends the game after 10000
 * ticks.
 *
 * @author Viera K. Proulx
 * @since 9 April 2010
 */
public class MyUniverse extends Universe<MyPlayer, String>{
  int tickcount = 0;

  /**
```

```
 * When the player disconnects, print an announcement
 * with the player's name.
 *
 * @param p the player that has disconnected
 */
public void onDisconnect(MyPlayer p){
  System.out.println("Player " + p.name +
                     " disconnected.");
}

/**
 * After the player successfully connected, send the
 * initial "roll" message.
 */
public void onConnect(MyPlayer p){
  p.sendMessage("roll");
}

/**
 * A Factory method that allows us to construct a new
 * <code>Player</code> instance for the
 * <code>Universe</code> to add to its player list.
 */
public MyPlayer initializePlayer(){
  return new MyPlayer();
}

/**
 * If the player rolls the same die twice,
 * send him a message "deuce".
 * Record the last roll and let the player roll again.
 *
 * @param message player's roll
 * @param p the player who just rolled
 */
public void onReceive(String message, MyPlayer p){
  int newroll = Integer.valueOf(message);
  if (newroll == p.roll)
    p.sendMessage("deuce");
  p.roll = newroll;
  // roll again
  p.sendMessage("roll");
}
```

```
  /**
   * Stop the game after 10000 ticks
   */
  public void onTick(){
    if (++this.tickcount >= 10000)
      while (this.getClients().size() > 0)
        this.getClients().get(0).disconnect();
  }
}
```

The Player

The only thing our *universe* needs to record about each player is the value that has been rolled on the last roll. So, our class that extends the `Player` class is defined as:

```
import edu.neu.universe.*;

/**
 * A class that represents a player
 * it records that last roll
 * @author Viera K. Proulx
 * @since 9 April 2010
 */
public class MyPlayer extends Player{

  /** a record of the last roll of the dice */
  int roll;

  MyPlayer(){
    super();
    this.roll = 0;
  }
}
```

The World

The `abstract class World` actually appears in the library twice. The first one provides the functionality needed for a single player interactive game on one computer - similar to what we have done with the *idraw* library. This is in the package `edu.neu.world.desktop` It requires that the programmer provides a public default constructor with no arguments

and implements the following methods (declared as `abstract` in the parent class):

```
/**
  * Called when the World starts.
  * Use this instead of the constructor to perform tasks
  * that should run when the World starts.
  */
public void init();

/** Draw this world on the given Canvas
  */
public void onDraw(Canvas c);

/** What the world does in reposnce to the key event
  * @param key the key that triggered the event
  * @param the key action: KEY_PRESSED or KEY_RELEASED
  */
public void onKeyEvent(IWorld.Key key, int type);

/** What the world does on each tick
  */
public void onTick();
```

The `abstract class World` in the package `edu.neu.universe` extends the first `World`. It implements three methods for communicating with the server *universe*:

```
/** Attempts a connection to the server.
  * @param ip the IP address of the server
  * @param port the port on which we wish to communicate
  * @param name the name by which the universe will call us
  */
public void connect(java.lang.String ip, int port,
                    java.lang.String name);

/** Forcefully disconnect this world from the universe
  */
public void disconnect();

/** Send the given message to the universe
  * @param msg the massage to send
  */
public void sendMessage(java.io.Serializable msg);
```

Additionally, the class that extends this `World` must implements the following methods declared as `abstract`:

```
 /** What the world does in response to a successful
   * connection to the universe
   */
 public void onConnect();

 /** What the world does in response to being disconnected
   * from the universe
   */
 public void onDisconnect();

 /** What the world does when the given message is received
   * from the universe
   * @param msg the given message
   */
 public void onReceive(M msg);
```

The following example shows our class that extends `World`.

```
import java.util.Random;
import edu.neu.world.*;

/**
 * A class to represent a dice player in a universe
 * After being connected, on receiving message "roll"
 * the user hits a space bar to roll again.
 *
 * @author Viera K. Proulx
 * @since 9 April 2010
 */
@WorldSettings(height=200,width=100,tick=5,title="MyWorld")
public class Dice extends edu.neu.universe.World<String>{
  /** silly toggle to flip color on every tick */
  boolean even = true;

  /** are we ready to roll again? */
  boolean ready = false;

  /** the current roll of the die to display and send to universe */
  int current = 0;

  /**
```

```
 * The required public default constructor
 */
public Dice(){}

/**
 * Connect to the server when the world starts
 */
public void onInit(){
  this.connect("127.0.0.1", 7070, "Dice client");
}

/**
 * On connection tell the universe you are ready to play
 */
public void onConnect(){
  sendMessage(randomRoll());
}

/**
 * Print a message indicating we have been disconnected
 */
public void onDisconnect(){
  System.out.println("We have been disconnected.");
}

/**
 * Allow the user to roll again
 * when told to do so
 * Disconnect if told to stop
 * Ignore any other messages
 * @param s the message received
 */
public void onReceive(String s){
  if (s.equals("roll")){
    this.ready = true;
  }
  else if(s.equals("deuce")){
    this.current = 1000;
  }
  else if (s.equals("stop")){
    disconnect();
  }
}
```

```
/**
 * Hit the space bar to roll again
 * and disable roll until the next message comes
 * from the universe
 */
public void onKeyEvent(IWorld.Key ke, int mode){
  if (ke.equals(IWorld.Key.SPACE) && this.ready
      && mode == KEY_PRESSED){
    this.current = randomRoll();
    sendMessage("" + this.current);
    this.ready = false;
  }
}

/**
 * Change the color of the dot on each tick
 */
public void onTick(){
  this.even = !this.even;
}

/**
 * Fill a red rectangle 50 by 100 in the top left corner
 * on the world canvas,
 * Display the last number rolled, (or 1000 if deuce)
 * Show a flashing dot - changing the color on each tick
 * @param c the world canvas
 */
public void onDraw(edu.neu.world.desktop.Canvas c){
  c.fillRect(0, 0, 50, 100, edu.neu.world.Color.RED);
  c.drawString("" + this.current, 20, 20, 12);
  if (this.even)
    c.fillCircle(25, 50, 5, edu.neu.world.Color.BLUE);
  else
    c.fillCircle(25, 50, 5, edu.neu.world.Color.YELLOW);
}

/**
 * A helper method to generate a random number
 * in the range 0 to n
 */
int randomRoll(){
  return new Random().nextInt(6) + 1;
}
```

```
}
```

The latest release allow the programmer to play an audio file for some period of time, or just once — see the Javadocs for details.

Advice

The most difficult part here is the design of the communication between the universe and the world. Make sure your design is clean and carefully thought out. Start small — to make sure you understand how the whole system works.

Make sure you test everything. There may be some parts where there is no support for simple tests. If that is the case, identify the problems, and run the 'trial tests' — just checking that the communications are working out correctly.

Enjoy.

Acknowledgments

The entire package that supports the design of the universe and world has been designed by Chris Souvey and Griffin Schneider. Our thanks to them for giving us this great toll.