CS032 Lab 05 - Mythical Man Month

1 Introduction

Welcom to the Mythical man month lab. In this lab you'll learn the importance of working in teams. There's almost no programming job anywhere where you'll be dealing only with code written by you. Therefore this lab is provided to give you some experience in working in groups. The objective of this lab is not necessarily to finish the assigment but to understand the mechanics of working as a team to complete a project.

That being said, your assignment will be to program a game called Gem Quest, made up by one of your very own TAs. In this game you are a traveler wandering through dungeons collecting gems while avoiding monsters. The mouse is used to control the character moving through the dungeon. You will need to parse files written in xml to load levels.

Keep these few things in mind as you work with your group:

- Design: How will the different pieces of this project fit together?
- Interaction: What do your teammates need from your code? What do you need from them?
- Integration: When should you start putting together all the pieces of the project?
- Communication: How can you make sure everyone is on the same page?

2 Getting Started

There are 6 main pieces to this project. They are the GUI Layout, the Game Engine, the Map Logic, the Map Graphics, the Object Factory, and the Parser. Depending on how many people are in your group you'll want to divide up the work as evenly as possible. You may choose to code in groups or code individually and then integrate. As you can see already there are many decisions your team will need to make which may help or hinder your progress.

2.1 The GUI Layout

This person should take care of graphical layout of panels and buttons. Your going to start out by subclass either the `javax.swing.JFrame` or the `javax.swing.JPanel` class. You must also display the number of Gems and Lives in the game and display messages when the player dies, when there are no more lives or when all the gems have been collected.

2.2 The Game Engine

This person will take care of the main game loop. Every moving object displayed on the map implements the `edu.brown.cs.cs032.gemquest.GemQuestActor` interface. In order for them to act they must be told to do so every 10 milliseconds. To do this you'll need to create a timer which tells all the actors to act every time it times out. To keep track of the actors you'll need to implement the `edu.brown.cs.cs032.gemquest.GemQuestActorController`. This interface will be needed when the factory starts producing actors. Since only you will know about the player during the game you should also implement the `edu.brown.cs.cs032.gemquest.GemQuestClickHandler` interface which has three methods (but you only have to implement one), each of which receives a click position from the Map Graphics for you to pass on to the player.

2.3 The Map Logic

This person keeps track of where objects are on the map and makes them interact with each other. You should implement the `edu.brown.cs.cs032.gemquest.GemQuestMap` interface so that all the game objects can communicate with the map and each other. All objects on the map (including walls) implement the `edu.brown.cs.cs032.gemquest.GemQuestObject` interface, allowing for uniform communication between the map logic and other objects.

2.4 The Map GUI

This class draws all the graphics to the screen. Although most of this is taken care of for you by the support code, you must tell all the graphics on the map when to draw. To do so you'll subclass the `edu.brown.cs.cs032.gemquest.GemQuestMapGUI` abstract class. It extends a `javax.swing.JPanel` so it can be added to the GUI Layout. Additionally, every graphic in the game implements the `edu.brown.cs.cs032.gemquest.GemQuestGraphic` interface which enables you to call the `paint()` method on the graphic.

2.5 The Object Factory

This person should create everything for the game. They should have references to the map, the actor controller and the gem counter so they can pass these on to whomever needs them as they're being created.

2.6 The Parser

This person is responsible for parsing the level file. The levels are in xml so you should be able to reuse your pinball2 code to parse them. The parser should have a reference to the factory so that it can tell it to make the appropriate objects.

3 In Depth Descriptions

The next six pages are detailed descriptions of what needs to be done for each part. NO ONE PERSON ON YOUR TEAM SHOULD HAVE TO READ ALL OF THEM! After everyone is done reading the summaries you should assign jobs and read only the pages you need for your part of the project. A key part of working in teams is commmunication. If you need to know what any of your team mates are doing ASK THEM! Not commicating can lead to confusion and aggravation which will only slow your team down. Its ok to stop coding and talk with your team mates about what is going on. Almost every decision you make will affect your whole group, so keep those who need to know in the know.

3.1 The GUI Layout

As the GUI Layout, you should be focused mostly on working in swing to lay out how the game will look. You may want to review what you've learned from lab04 if you don't feel comfortable enough to tackle swing on your own. You should start by making a `javax.swing.JFrame` and setting up a `javax.swing.JPanel` that will hold the Map Graphics and the labels you need to display the life counter and gem counters. The GUI Layout will be the top-level class, creating the logic, the parser, the factory and the map gui. You will also need to call `parse()` on the parser, passing in either a command line parameter or a hard coded String. After you've parsed the level, don't forget that you have to call `setup()` on the Map Logic before any of the graphics will load.

Important interfaces to know about:

`edu.brown.cs.cs032.gemquest.GemQuestGemCounter`

Methods:

- `public void gotGem()`: Called when a gem is picked up by the player. If there are no more gems on the map the game is won.
- `public void addGem()`: Called when a new gem has been added to the map.
- `public int numGems()`: Should return the current number of gems on the map.
- `public void reset()`: Should be called when the player has died so you can reset the number of gems on the board.

Important swing classes to know about:

- `java.awt.BorderLayout`: See javadocs...
- `java.awt.FlowLayout`: Can be used to align swing components horizontally.
- `javax.swing.JButton`: A Button. You will need to add a `java.awt.event.ActionListener` to make it do something when clicked.

Anonymous class syntax:
In some cases in swing making an anonymous class is more efficient than creating an inner class. For example making an action listener for a quit button. There is very little code that will be in this class and there is only one other class that will be communicating with it. Usually anonymous classes are a bad idea as you they are very hard to debug, but in this case there is no chance of anything serious going wrong. Here is an example of how to write an anonymous class:

```
JButton quit = new JButton("Quit");
quit.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(java.awt.event.ActionEvent e)
        {
            System.exit(0);
        }
    });
```

The `java.awt.event.ActionListener` is actually an interface so this tells java that your anonymous class will be implementing that interface. Then within the brackets you define all the methods (and even a constructor) that your anonymous class will have.

3.2 The Game Engine

The Engine will contain the main game loop, which will be controlled by a timer. You will need to create a collection of actors, whom you will call the `act()` method on periodically. You must implement the `edu.brown.cs.cs032.gemquest.GemQuestActorController` so that the factory can properly add all of the actors to your collection, . These are the methods you must implement:

- `public void addActor(GemQuestActor actor)`: Add actor to a collection of actors.
- `public void removeActor(GemQuestActor actor)`: Remove actor from the actor collection.
- `public void setPlayer(GemQuestPlayer player)`: Tell the ActorController who the player is.

You will also need to implement the `edu.brown.cs.cs032.gemquest.GemQuestClickHandler` interface. Since the engine is the only class that will know explicitly about the player during the game, it is the only class which can tell the player when the Map Graphics have been clicked. There are three methods you must declare, but you will only need to fill in one for this assignment. They are:

- `public void clicked(java.awt.Point p)`: Called when one of the mouse buttons is clicked on the Map Graphics. (If you fill in this method, you don't need to fill in pressed().)
- `public void pressed(java.awt.Point p)`: Called when one of the mouse buttons is pressed down on the Map Graphics. (If you fill in this method, you don't need to fill in clicked().)
- `public void released()`: Called when the mouse button from pressed has been released. (You shouldn't have to write anything for this method)

Whichever method you choose to use (clicked or pressed) be sure to convert the position from pixels to tiles. You can do this by dividing the x and y axis by 50 like this:

```
GemQuestPosition move = new GemQuestPosition(p.x * 50, p.y * 50);
the_player.moveTo(move);
```

Finally, you'll need to create a `javax.swing.Timer` for the main game loop. Set its delay to 10 and add an `java.awt.event.ActionListener` to listen for when it has timed out. When the timer does time out you'll want to go through your collection of actors and call `act()` on each of them (order does not matter). Because there is a possibility that the player may encounter a monster when you call `act()`, you will need to try calling the `act()` method and catch an `edu.brown.cs.cs032.gemquest.PlayerDeathException`. In this situation you should decrement the amount of lives (if lives < 1 then game over) and call reset on all the actors, which brings them back to their starting position for the next round. You will also need to reset the number of gems on the board.

Don't forget to call `start()` on your timer!!!

Important Interfaces:

`edu.brown.cs.cs032.gemquest.GemQuestActor`

Methods:

- `public void act(int time)`: Do whatever it does.
- `public void reset()`: Moves the actor back to its starting position and resets its animation.

Extra Note: Time is required for use in calculating the speed at which the actors move. It should go from 0 to 99 every 10 milliseconds. To do this add an instance variable of type `int` called `the_time` (or whatever you want to call it). Then within your main game loop add the line:

```
the_time = (the_time + 1) \% 100;
```

Then pass `the_time` as the `time` parameter for the actors `act()` method.

3.3 The Map Logic

This person takes care of interactions between objects on the map. We suggest you make a 2D array of `java.util.Vectors` which will contain `edu.brown.cs.cs032.gemquest.GemQuestObjects`.

All objects on the map require a reference to the map. For this reason you must implement the `edu.brown.cs.cs032.gemquest.GemQuestMap` interface. These are the methods you must implement:

- `public GemQuestMapGUI getGUI()`: returns a reference to the Map GUI.
- `public boolean isPassable(GemQuestPosition p)`: Returns `true` if tile at position `p` is passable and `false` if the tile is outside the bounds of the map.
- `public void removeObjectFrom(GemQuestObject o, GemQuestPosition p)`: remove the object from tile at position `p`.
- `public void moveObject(GemQuestObject o, GemQuestPosition from, GemQuestPosition to)`: remove object from tile at position `from` and add it to tile at position `to`.
- `public boolean addObjectTo(GemQuestObject o, GemQuestPosition p)`: Add the object `o` to tile at position `p`. Should return `false` if there is an impassable object on that tile.
- `public void setup()`: call `setup()` on all the objects on the map.
- `public void visit(GemQuestObject o, GemQuestPosition p) throws PlayerDeathException`: Tell all objects on tile at position `p` they have been visited by `o`. (The support code may throw a `PlayerDeathException`)

To easily loop through all objects on a given tile you can use `java.util.Iterators`. Here is an example of how to loop through all the objects on a tile on the map:

```
for (Iterator it = the_map[p.x][p.y].iterator(); it.hasNext(); ) {
    GemQuestObject o = (GemQuestObject)it.next();
    o.setup();
}
```

Where `p` is of type `edu.brown.cs.cs032.gemquest.GemQuestPosition`.

In your constructor you should also call `setSizeStuff(java.awt.Dimension d)` on the map graphics, passing the width and height of the map multiplied by 50 (the size of a tile). (`setSizeStuff()` is a method in the `edu.brown.cs.cs032.gemquest.GemQuestMapGUI` abstract class)

Important support code:

`edu.brown.cs.cs032.gemquest.GemQuestObject`

Important Methods:

- `public boolean isPassable()`: returns `true` if the object is not an obstacle.
- `public void setup()`: Loads graphics on this object.
- `public void visit(GemQuestObject o)`: Tells the object that object `o` has entered the same tile.

`edu.brown.cs.cs032.gemquest.GemQuestPosition`

Public Instance Variables:

- `int x`: x axis
- `int y`: y axis

3.4 The Map Graphics

This person should subclass the `edu.brown.cs.cs032.gemquest.GemQuestMapGUI` which extends a `javax.swing.JPanel`. You'll need to keep a collection of `edu.brown.cs.cs032.gemquest.GemQuestGraphics`, which you will aquire by implementing the methods required by the abstract super class. They are:

- `public void addGraphic(GemQuestGraphic g)`: Add the graphic `g` to the collection.
- `public void removeGraphic(GemQuestGraphic g)`: Remove the graphic `g` from the collection.

In addition, you'll need to partially override the `paintComponent(java.awt.Graphics g)` method (from `javax.swing.JPanel`) to also call `paint()` on all the graphics in your collection. Be sure to cast the `java.awt.Graphics` to a `java.awt.Graphics2D` before you pass it to your collection of graphics.

Finally, you should take a `edu.brown.cs.cs032.gemquest.GemQuestClickHandler` in your constructor since the `JPanel` will be recieving the clicks to pass on to whomever the click handler is. You also need to set up a mouse adaptor, and since we haven't taught you how to do so here is the code:

Somewhere in your constructor add:

```
addMouseListener(new MouseClick());
```

Now add this inner class:

```
class MouseClick extends MouseAdapter {

    public void mouseClicked (MouseEvent e)
    {
        click_handler.clicked(e.getPoint());
    }

    public void mousePressed (MouseEvent e)
    {
        click_handler.pressed(e.getPoint());
    }

    public void mouseReleased(MouseEvent e)
    {
        click_handler.released();
    }
}
```

Make sure to import `java.awt.event.*` or you'll have to put the whole class path for the `MouseAdaptor` and `MouseEvents`.

Important Interfaces:

`edu.brown.cs.cs032.gemquest.GemQuestGraphic`

Methods:

- `public void paint(java.awt.Graphics2D g)`: draws the graphic to the screen.

3.5 The Factory

This person will need references to the actor controller, and the gem counter so it can pass these on to whomever needs them. You should write create methods for each of the following objects:

- The Map Logic (you'll need to hold on to this after you've created it)
- The Player
- Gems
- Walls
- All the Monsters

Don't forget you must add the objects to their appropriate controllers (eg all the actors should be added to the actor contoller). Here is an example of what your create methods should look like:

```
public void createFireTree(GemQuestPosition p) throws LoadException
{
    GemQuestMonster tree = new GemQuestMonsterFTree(the_map, p);
    if (!the_map.addObjectAt(tree, p)) {
        throw LoadException("Bad Firetree position");
    }
    actor_controller.addActor(tree);
}
```

This method creates a Fire Tree Monster, adds it to the map and then adds it to the actor controller. If the `addObjectAt()` method returns `false`, meaning the position was invalid it throws an exception which should terminate the program. You will also need to write your own LoadException. To write an exception you need to subclass `java.lang.Throwable`. Then in your constructor, you'll want to take a string which you pass on to the superclass' constructor.
Keep in mind your exception will need to be public so that other classes can catch it.

Constructors
(all these classes are in package `edu.brown.cs.cs032.gemquest.*`)

`GemQuestPlayer(GemQuestMap map, GemQuestPosition start)`
 `map` - Reference to an `edu.brown.cs.cs032.gemquest.GemQuestMap`.
 `start` - Start position on the map.

`GemQuestMonsterFTree(GemQuestMap map, GemQuestPosition start)`
`GemQuestMonsterSnake(GemQuestMap map, GemQuestPosition start, boolean vert)`
 `vert` - `true` if the snake should move vertically.

`GemQuestMonsterEye(GemQuestMap map, GemQuestPosition start)`
`GemQuestMonsterFrog(GemQuestMap map, GemQuestPosition start)`
`GemQuestMonsterLeech(GemQuestMap map, GemQuestPosition start)`
`GemQuestGem(GemQuestMap map, GemQuestPosition start, GemQuestGemCounter gemcounter)`
 `gemcounter` - Reference to an `edu.brown.cs.cs032.gemquest.GemQuestGemCounter`.

`GemQuestWall(GemQuestMap map, GemQuestPosition p)`

3.6 The Parser

This person will parse the level file. You should write a method called `parse()` which takes in a string which will be the file to parse. Most of the parsing code you should be able to copy from your pinball2. The grammer for Gem Quest is different from pinball2, but the code should look very similar. Here is a description of the grammer:

```
<GAME NAME = "STRING">
    <MAP HEIGHT = "INTEGER" WIDTH = "INTEGER">
        <ROW WALLS = "STRING"/>
        \\More walls
    <\MAP>

    <OBJECTS>
        <OBJECT TYPE = "STRING" POSX = "INTEGER" POSY = "INTEGER" [VERTICAL = "STRING"]/>
        \\More Objects
    </OBJECTS>
</GAME>
```

You should start by parsing the `GAME` and ensuring that there is only one. Then you can begin to parse the `MAP`. The attribute `WALLS` is actually a String containing 0s and 1s (0 for blank space and 1 for a wall). To parse a String you can use the `charAt()` method built into all Strings. It should look something like this:

```
for (int j = 0; j < walls.length(); ++j) {
    if (walls.charAt(j) == '1') {
        the_factory.createWall(new GemQuestPosition(j, i));
    }
}
```

Where `walls` is the String representing the `WALLS` attribute and `i` is the current row.

Most `OBJECT`s need only an x and y position with the exception of the `SNAKE`, which also needs the `VERTICAL` attribute. It should be either 'T' or 'F' (for `true` or `false`).

You will also be responsible for throwing exceptions when there are errors in the grammer. We won't be testing your parser for mistakes, but some grammer errors may cause other errors in your program which may be harder to track down.

Object Types

- `PLAYER`: There should always be 1 and only 1.
- `GEM`: A gem...
- `FIRETREE`: Monster type.
- `SNAKE`: Monster type. Needs `VERTICAL` attribute.
- `LEECH`: Monster type.
- `EYE`: Monster type.
- `FROG`: Monster type.

4 Wrapup

You may notice later that certain design choices have been left undescribed. These will be up to you and your team to decide. Because no single person on your team should need to know exactly how the entire program works, its important that everyone understands the design of the project. Take some time before you start coding to make sure eveyone knows whats needs to be done an what parts they are responsable for.

5 Stuff

Stuff you'll need to get started...

5.1 The Demo

To run the demo and see what your project should look like run `/course/cs032/demos/lab05` in a shell. By default it loads a simple level file. If you type the name of another file it will load that instead. Your project doesn't have to be exactly like the demo, but it should have similar functionality.

5.2 Level files

The level files are located in `/course/cs032/pub/lab05/`. Feel free to make your own level files (time permitting).

5.3 Build file

The `build.xml` file you will need is also in `/course/cs032/pub/lab05/`. You'll need to change the `main.class` property to the name of the GUI Layout class.

5.4 Java docs

You shouldn't really need them very much, but if you'd like to understand more about how the support code works the java docs can be found at http://cs.brown.edu/courses/cs032/labs/mmm_docs/.

A Monster Behaviors

In case you want to know how the monsters behave. You do not have to worry about coding any of these behaviors.

- Fire Tree: The player dies if he/she steps on it. Kind of boring, but it looks cool...
- Snake: Moves vertical or horizontal. When it hits a wall it reverses direction.
- Eye: Chases the player if the player moves 1 tile away.
- Frog: Hops in a random direction. When it hits a wall it changes direction.
- Leech: Flops 1 tile in a random direction every few seconds.