CS 1331 Homework 10

Due: Monday, November 30, 2009

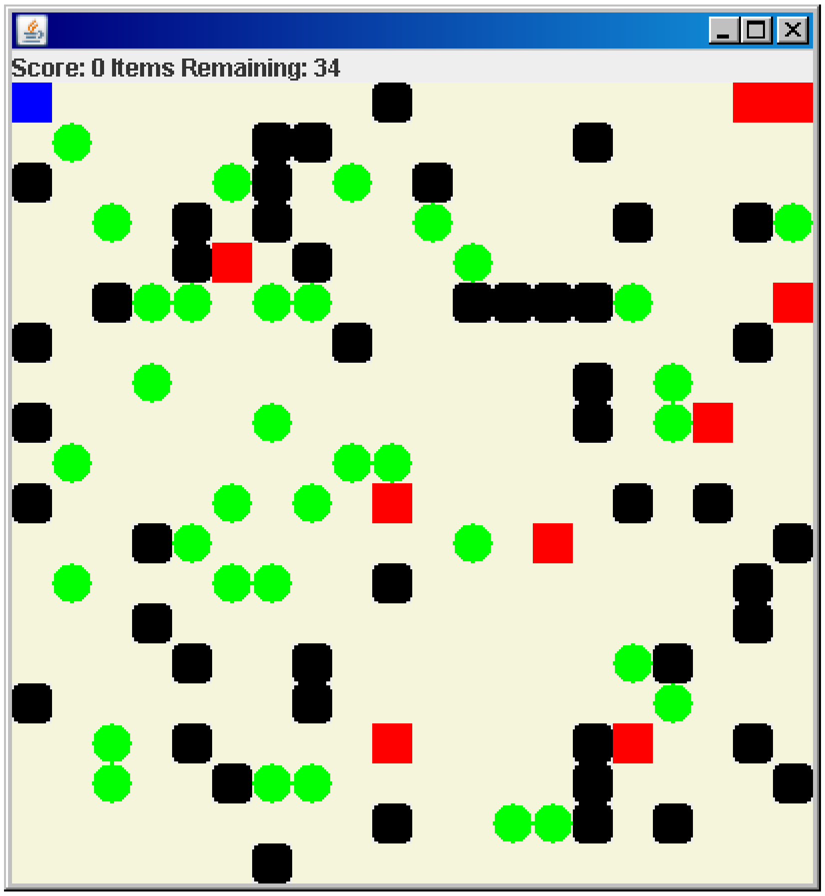
Introduction

This assignment will test various skills you have used in this class, including GUI construction, graphics, abstract classes, inheritance, polymorphism, dynamic binding, timers and KeyListeners. You will create a fully functional game using object-oriented design.

This is the final homework, and will have twice the weight as the previous homeworks.

Description

You will create a game where you control a character that collects gems while avoiding monsters:



Score: 0 Items Remaining: 34

There is a status bar at the top with the score and number of items left, and the playing field takes up the rest of the screen. You control the blue square, using the arrow keys to move. Black squares are walls; you cannot walk through them. Green dots are gems and are worth points. Red squares are monsters – if one touches you, you die.

Architecture

This is one possible design. Read through it to get a feel for the game mechanics, keeping in mind that you have a lot of flexibility in your design.

Game.java

Game.java is the main driver class and information holder. Game's constructor needs to create new Level and Player objects, then create the ArrayList and add some Monsters to it. Finally, Game starts a timer. The timer should call the update method of all monsters, call the level panel's repaint method, and update the score and status bar at the top of the screen.

The Level, Player and Monster classes need constructors which take in a Game object (among other things) as a parameter. Game.java will pass itself to each of those classes.

Game should have win() and lose() methods which get called when the player has either collected all the items, or is killed by a monster. Both methods should stop the timer and display some kind of message to the user (either through the status bar next to the score, or JOptionPane).

Game.java should have a main method which creates a JFrame and adds a LevelPanel and a *JLabel statusLabel*. LevelPanel is explained further below. statusLabel is an instance variable of Game and just displays the score and status.

Here's a recap of Game's instance data. Provide getters for all of these (except the timer):

- A Level object.
- A Player object.
- An ArrayList of Monsters.
- A JLabel that displays the status and score.
- A LevelPanel.
- A Timer (don't need a getter for this)

and a recap of the methods:

- Game() -- constructor, takes no parameters
- void win() -- called when player wins. Stop the timer and display a congratulatory message.
- void lose() -- called when player loses. Stop the timer and display a message.
- static void main(String[] args) -- create a JFrame, adds the LevelPanel and statusLabel.
- Getters for all instance variables (except the Timer)

(abstract) Item.java

Item.java represents an item on the field (i.e. something a player can pick up). It should have:

- Item() -- constructor, takes no parameters
- An abstract method public void draw(Graphics g, int row, int col).
- A variable *int value* of this object (i.e. how many points it's worth)
- A method public int getValue() which returns the value.

Gem.java

Gem.java represents a gem item on the field (the green dots). It should be a subclass of Item. A gem is worth one point (no need to take it in as a parameter--you can either hard-code this in the constructor or you could just hard-code the return value in getValue() and omit the instance variable completely).

(abstract) Tile.java

Tile.java represents a space on the game field. Therefore, a level contains a 2D array of Tiles. A Tile represents a space on the board, but the exact type may be either a Floor or a Wall, or other subclasses.

Tile should have:

- Tile(int row, int col) -- constructor which takes in the row and column
- Protected instance variables *int row* and *int col*.
- public abstract void draw(Graphics g) -- draw the tile
- public abstract boolean isPassable() -- returns true or false depending on whether a player or monster can enter this Tile
- public int getRow() and getCol() -- getters for the row and column.
- public abstract void playerEnters(Player p) – Optionally, if a tile does something to the player, you might include this method to perform that action on the player.

Floor.java

Floor.java represents a floor tile on the field. Players and monsters can both pass through a Floor tile.

In addition to the methods inherited from Tile, Floor should have:

- Instance variable *Item item* -- an Item on this tile which the player can pick up.
- Floor(int row, int col, Item item) -- constructor. Chain to Tile's constructor to set up the row and column, then assign item as an instance variable.
- A method Item collectItem() -- returns a reference to the Floor's item as well as removing the item from the Floor (think carefully about this! If you set item to null, you cannot also return it...how do you resolve that?)

Wall.java

Wall.java represents a wall tile. Players and monsters cannot pass through a Wall tile.

Wall should have:

- Wall(int row, int col) -- constructor, chain to superclass
- Other methods inherited from Tile

(abstract) Creature.java

Creature.java represents a living object in the game, such as players and monsters.

Creature should have:

- Creature(Game game, int row, int col) -- constructor should take in a game, a row and a column.
- Protected instance data for the *game, row* and *col* (assigned in the constructor from the parameters given).
- public abstract void draw(Graphics g) -- in subclasses, this method will draw a representation of the Creature. Use the row and column parameters to figure out where to draw.
- public abstract void update() -- gets called once every time the timer fires. Does stuff like move a monster.
- public boolean runsInto(Creature c) -- return true if two Creatures are overlapping (i.e. same row and column).

Player.java

Player.java represents the player (the blue square that you control). It should be a subclass of Creature.

In addition to the methods inherited from Creature, Player should have:

- Player(Game game, int row, int col) -- constructor, chain to superclass
- An instance variable, *int score*.
- A method int getScore()
- A method void move(int drow, int dcol) -- explained below

Player should have a move(int drow, int dcol) method which takes in the number of rows and columns to move. That method should first determine whether the move is legal (checking to make sure you're not moving off the map or into a wall). If the move is legal, the method should do this:

- Update the Player's row and col to the new location.
- Try to collect an item from this spot. Get the level from the Game object, then use the collectItem method.
- If an item was there, update the score.
- Check whether any Monsters overlap the Player. Get the list of Monsters from the Game object, then check each Monster using the runsInto method in Creature.
    - If any Monsters overlap, call that Monster's attackPlayer method.
- If there are no items left in the world, call the game's win() method.

(abstract) Monster.java

Monster.java represents monsters in the game. It should be a subclass of Creature and will itself have various subclasses, so Monster should also be abstract.

Monster should have:

- Monster(Game game, int row, int col) -- constructor, chains to constructor in superclass with same parameters
- Monster(Game game) -- a second constructor, should chain to the first constructor and generate random numbers for the row and col. You can get the maximum by getting the level from the game parameter, then retrieving the rows or cols. Remember that the call to the constructor must be the first line, so you need to generate the random numbers in the same line as the constructor call.
- An abstract method public abstract void attackPlayer(Player p). In subclasses, the attackPlayer method should do something to the Player object (decrease its score, remove lives, end the game, etc.).

Game.java should use the second constructor to randomly place the Monsters on the field.

SimpleMonster.java

SimpleMonster represents a basic monster. It should be a subclass of Monster. SimpleMonster should have:

- Instance data *int drow* and *int dcol* -- the amount to move each turn for the row and column.
- SimpleMonster(Game game, int row, int col, int drow, int dcol) -- constructor that takes in the game, row, col, and the amount to move. So if it's called with (game, 5, 5, -1, 1), the monster will start at (5,5) and every turn it will move one square left and one square down (remember that increasing y is in the down direction). This constructor should first chain to the superclass to set up the row and col, then assign drow and dcol as instance data.
- SimpleMonster(Game game) -- chains to the superclass constructor to get a random row and column, then should generate random values for drow and dcol ranging from -1 to positive 1 inclusive.

Every turn, the update method will advance the SimpleMonster based on the amount to move. If the SimpleMonster cannot move any further, it should reverse direction and go the other way (i.e. multiply your drow and dcol by -1). Thus, your SimpleMonster will appear to patrol back and forth along a row, column or diagonal.

Your update method should also check whether the monster is running into the player (get the player object from the Game reference, then use the runsInto method).

If you find that the monster moves too quickly, you can slow it down using a counter. Add a counter to SimpleMonster and every time the update method is called, increment the counter. If the counter is less than 5, for example, exit the update method without doing anything further. If the counter is equal to 5, reset the counter to zero and then do the action. This will make the monster move five times slower.

Level.java

Level.java holds all information needed for a level. It should have as instance data:

- 2D array of Tiles
- A reference to a Game object
- Number of items left in the world (this is used in the status bar and determines when the game is over)
- Instance variables for the number of rows and columns

Level will not be extended, so all instance data can be private.
Level also needs a few methods:

- <u>Level(Game game, int rows, int cols)</u> -- constructor. Assign the parameters as instance data. The constructor must also create the level, see below.
- <u>void draw(Graphics g)</u> -- unlike all other draw methods, this method should do three things (in this order):
  - Draw all Tiles (by iterating over the array and calling the draw method for each)
  - Draw all Monsters
  - Draw the Player. Get the player through the reference to the Game object.
- <u>boolean canMoveToSquare(int row, int col)</u> -- this method should check first whether (row, col) is in bounds, and secondly whether a Creature can pass through that tile (hint: use the isPassable() method in Tile).
- <u>Item collectItem(int row, int col)</u> -- this method should collect an item from the Tile at (row, col) (check to see whether it's in bounds first). Remember that only Floor tiles have a collectItem method, so you'll need to check the type. Don't forget to update the item count.
- Getters for the number of rows, columns, and item count.

Finally, Level's constructor has to create the actual level (i.e. fill the 2D array of Tiles). Your program should be able to create two kinds of levels. **First you should read in a level from a file on disk**. You can hard-code the file name, or you can provide a control panel with a button and JFileChooser to choose a file. One way to encode a level in a text file is:

```
oooooxoogoo
oooooxooooo
ooooooooooo
oogoooxoooo
```

etc., where o is a floor tile, x is a wall tile, and g is a floor tile with a gem. This is just a sample, you may use whatever representation you like. By reading in this file, your Level class should be able to fill in the 2D array of Tiles. (The location of Monsters can be randomly generated in the Monster constructor, and you can assume the Player always starts in the upper left).

You can use serialization instead if you like, although then you won't be able to edit the level as easily.

Be sure to use relative paths when doing file i/o so it will work on your TA's computer.

Second, if the file is empty or missing, you should generate a random level. To generate a random level, iterate over your Tile[][] array. For about 15% of the spots (use a Random object to decide this), make them a wall. For the other 85%, make it a Floor tile. Add items to about 10% of those Floor tiles. Those numbers are not requirements; you can play around with them to find good numbers that work.

With randomly generated levels, it's possible you might generate an unwinnable game (maybe your Player is hemmed in by Walls). That is ok.

LevelPanel.java

LevelPanel.java represents a single level and its GUI.

LevelPanel should have:

- LevelPanel(Game game) -- constructor. Set the instance variables and add a KeyListener. If you think back to recitation, you need to do special things with focus in order to make KeyListener work.
- Instance variable for the Game object.
- public void paintComponent(Graphics g) -- call the superclass paintComponent method, then call the level's draw method. Get the Level through the Game object.
- A KeyListener. The KeyListener should listen for the arrow keys and when pressed, call the Player's move method. Get the Player through the Game object.

Requirements

There are several classes in this program:

- Game (contains main)
- Item
- Gem (extends Item)
- Tile
- Floor (extends Tile)
- Wall (extends Tile)
- Creature
- Player (extends Creature)
- Monster (extends Creature)
- SimpleMonster (extends Monster)
- Level.java
- LevelPanel.java

At minimum you should have all those classes (or some equivalent in your own design). They should do roughly what's described above, although you can customize many parts as you see fit (such as the movement pattern of a SimpleMonster). You do not have to follow the architecture described above, as long as your own design shows object-oriented principles, and the game works basically as in the description section.

Your Level class, however, must be able to read in from a file in addition to creating a random level.

The graphics for each object can be simple (squares, circles, etc.) or they can be as elaborate as you wish. You may also use external images.

The recommended design has three different class hierarchies (Item, Tile and Creature are the parents of each). **Your final program must have at least three new subclasses beyond the ones listed in the architecture described above**. The three classes may span all three hierarchies or be all from the same one. The subclasses should have distinct behavior from the other subclasses in the hierarchy (their siblings). For example, having a Gem subclass that is worth 1 point and another subclass of Gem that is worth 3 is a not very good, because that can be taken care of simply with instance data.

Overall, we are looking to see if you've learned the basic principles of Java and object-oriented programming and design, and if you put thought and effort into the assignment. Minor details are left up to you.

When you are finished, **write a file called readme.txt that explains the design and rules of your game** (if you changed it from the rules described here) and contains a description of any extra features you added. If you want to include screenshots, you can use an HTML file (readme.htm) or PDF instead. If you're only including text, please use a .txt file.

Extra Credit

You can earn for adding additional features. Here are some possible ideas for additional features and the extra credit you would earn:

- Adding the ability to pause (maybe pressing "P" on the keyboard)
- Turn this program into an applet (if your program is designed well, it's not difficult). If you do this, you must submit the HTML file as well so we can test it. Your program still needs to work if we run Game.java.
- Having intelligent monsters which chase after the player instead of moving on a fixed pattern
- Having a time limit to collect all the items
- Having lives so when a monster touches you, you don't die instantly
- Saving levels as well as loading them (you would need to provide a control panel so the user can choose to save)
- Saving a high score list
- Multiple players (i.e. another human can use WASD to control a second player). Depending on your design, this could be relatively straightforward or rather complex
- Level editor with the ability to create custom levels (i.e. place different types of Tiles and Monsters on the map) and save them (this extra credit includes the points for the saving levels feature)
- Making elaborate images for Monsters or Tiles

You may also add any feature not listed here and the extra credit value will be determined on a case-by-case basis (the more complex it is, the more points you'll get...most additional features would get around 3-6 points).

Be sure to document all extra features in your readme file so the TAs know to look for it.

Turn-in Procedures

After you have finished your program, turn in through T-Square:

- ALL files required to make your program run.
- readme file, explained under the Requirements section.
- A text file holding a level that gets read by your program. If your program can handle multiple files, you only need to submit one.

Don’t forget to javadoc the classes and important methods of your program.

You may zip all your files and upload the zip file to make sure you don't forget to upload something. In either case, be sure to double-check by downloading your submission off T-Square and compiling and running it. The TA should be able to download your submission to a folder, compile and run it without requiring additional files or making any changes to your code.