Matriculation number 40070877

Module title ....Software Development 2

Module number.........................SET11103

Title of Assignment ....Coursework: 'Get out of my swamp'

Lecturer ....Dr J Owens

Date of Submission ....2012_April_24

Word count ....3905 words

HOW I CREATED MY SWAMP (HOW IT WORKS)

The swamp task revolves around being able to keep track of and change the locations of the ogre and his enemies. Once this can be done, it should be fairly easy to display the swamp and its inhabitants and to find out how many enemies are at the same location as the ogre. When the number of 'immediate' enemies can be found, it should then be fairly easy to program the swamp and its inhabitants to act accordingly, i.e.

- no immediate enemies: no battle, go on to next turn
- 1 immediate enemy: battle, enemy loses and is removed from game
- \> 1 immediate enemy: battle, ogre loses, is removed from game, game ends.

After a number of false starts (see page 10), I realised that the only objects which need to know the creatures' locations are the creatures themselves: if the creatures know their locations and have public methods for movement, these methods can be invoked whenever the swamp/system needs. Further, when the swamp/system needs to know where the creatures are, e.g. to draw the current state, it can ask them, so long as the creatures' locations are available via public methods.

Hence the first task was to create a suitable *Creature* class.

***Creature* class**

This has instance variables:

- *xCoordinate*
- *yCoordinate*
- *type* (i.e. "OGRE" or "enemy")
- *subtype* (i.e. "OGRE", "donkey", "parrot" or "snake") - used to distinguish between the different types of enemy
- *swampSize* - to determine the maximum x- and y-coordinates

The class's constructor receives and sets the coordinates and *swampSize* when a *Creature* object is created. The necessary setters and getters are present.

Creature was initially a concrete class while but soon became abstract with descendent classes for *Ogre*s and different kinds of enemy.

Creature class methods
changeCoordinate()
This method is passed a coordinate and the *swampSize*, then calculates a new value for this coordinate. The coordinate is randomly left as-is, incremented by 1 or decremented by 1, using a do-while loop to keep the *Creature* in the *Swamp*.

actualMove()
This uses *changeCoordinate()* on the *Creature*'s x- and y-coordinates. A do-while loop is used to ensure that at least one of these coordinates has changed and hence a legal move has occurred.

Ogre and *Enemy* subclasses

These classes both receive their x- and y-coordinates and *swampSize* from the code that creates such objects. However, any *Ogre* object sets its *type* and *subType* to "OGRE" (capitals are used to make it easier for the user to follow the *Ogre* moving around the *Swamp*), while *Enemy* is an abstract class (parent to *Donkey*, *Parrot*, and *Snake*). All *Enemies* set their *types* to "enemy" and their *subtypes* to the type of enemy they are supposed to be.

Swamp class

This is basically a class containing an ArrayList of *Creatures*. It has two instance variables:

- *swampSize* - received from the code which creates a *Swamp* object, to be passed into any *Creatures* it creates
- *ogreAlive* - so long as the ogre is alive, the game continues

When created (by a *SwampInterface*), a *Swamp* object's constructor automatically creates an *Ogre* and adds this object to its ArrayList. *Swamp* class has the following methods:

Ogre creation
createOgre()
This creates an *Ogre* object, using a do-while loop to ensure that the Ogre is at any random location apart from (0,0), the top-left corner of the swamp.

addOgre()
This method adds an *Ogre* object to the *Swamp*'s ArrayList. This method and *createOgre()* are called by *Swamp*'s constructor to ensure that all new *Swamp* objects each have exactly 1 *Ogre*.

Enemy creation
addEnemyIfAppropriate()
Because there is to be a 1 in 3 chance of adding an enemy and the 3 kinds of enemy have the same chance of arriving, there is a 1 in 9 chance of any particular kind of enemy arriving

Drawing the swamp
drawSwampMap()
The method seems somewhat inefficient: each *Creature* in the *Swamp* is 'asked'
'Are you in (0,0)? If so, I will add your subType to the String representing that location.
Are you in (0,1)? If so, I will add your subType to the String representing that location.
Are you in (0,2)? If so, I will add your subType to the String representing that location.
Are you...'
So each *Creature* is asked swampSize^2 'questions'. For a large *Swamp*, this might become slow.

If the Swamp had consisted of gridsquares, each having its own collection of Creatures, it would be fairly efficient to add each gridsquare's Creatures' *subTypes* to a String which would be the map.

Moving the *Creatures*
The *moveAllCreatures()* method simply tells each *Creature* in the swamp to move according to their own rules.

Battles

ogreXcoord(), ogreYcoord(), numberOfEnemiesInOgreCurrentLocation()

To find out whether there should be a battle, it's necessary to find out how many enemies are at the ogre's location, i.e. how many enemies have the same coordinates as the ogre. To do that, the ogre's coordinates are needed, so the ogre is 'asked' to supply them. This means 'asking' each *Creature* 'if you are the ogre, what is your x-coordinate? If you are the ogre, what is your y-coordinate?'

POSSIBLE IMPROVEMENT	To alleviate the resulting processing demands, it may be possible to make *Ogre*s return their coordinates to instances variable within *Swamp*. The Swamp would still need to 'ask' each enemy 'are you at the ogre's coordinates. If so, I will increment the number of immediate enemies' - making sure not to include the *Ogre* itself in this number. Further, *Ogre* objects would need slightly different movement methods to other *Creature*s, and so *Swamp*'s *moveAllCreatures()* would need a part for moving the *Ogre* and another part for moving the other *Creature*s.

battle()

This uses *numberOfEnemiesInOgreCurrentLocation()* to decide whether to do nothing (0 immediate enemies), call *onlyOneImmediateEnemy()* or call *moreThanOneImmediateEnemy()*.

onlyOneImmediateEnemy(), moreThanOneImmediateEnemy()

These both use for-each statements to find the subTypes of the ogre's immediate enemy or enemies.

- If there is only 1 immediate enemy, *removeCreature*() is called to remove it from the *Swamp*'s ArrayList. The user is then informed.
- If there is more than 1 immediate enemy, their subTypes are found. This is then used to inform the user that the ogre has been beaten. The *Swamp*'s *ogreAlive* variable is then set to *false*, thus ending the game.

removeCreature()

This is passed the battle's coordinates and the subType of *Creature* to be removed. It uses yet another for-each block to find the removee's position (i.e. its index) within the *Swamp*'s ArrayList. Once the for-each is finished, the resulting index used by a single line of code that removes the appropriate *Creature*.

It's possible for the variable *indexOfCreatureToBeRemoved* to have several successive values while the for-each block is running, i.e. one for every *Creature* at the 'battle site' with the appropriate *subType*. Were this to happen, the *Creature* with the highest index would be the one removed. However, apart from their indices, all 'donkeys' (for example) at the same location are identical so it shouldn't matter which one would be removed. Also, because this method is only called when there is 1 immediate enemy, *indexOfCreatureToBeRemoved* will only ever have 1 value apart from the initial value.

Running the game

actualGame()

This method is called by *SwampInterface* whenever a game is (re)started and hence a *Swamp* is in action. There are three basic components:

1. The first map of the swamp and its inhabitants is drawn. For a brand-new game, this will show the swamp contains only a brand-new *Ogre*. For a restarted game, this will show the *Ogre* and any *Enemies* that have been created.

2. a do-while loop
    1. calls *addEnemyIfAppropriate()*, so a new enemy might appear at (0,0)
    2. asks the user if he or she wants another go. If the user wishes to stop, the loop will be exited as soon as possible.
    3. calls *battle()* to decide whether to whether there should be a battle and, if so, the outcome and consequences
    4. if the *Ogre* is still alive, moves all the *Swamp*'s inhabitants by calling *moveAllCreatures()*
    The loop is exited if *ogreAlive* is false, i.e. the ogre had more than 1 immediate enemy or if *anotherGo* is 1/No.

3. If the do-while loop has been exited:
    • if the *Ogre* is still alive, the *Swamp* is saved to disk and the user is taken back to the *SwampInterface* menu.
    • if the Ogre has been killed, a final map showing the positions of all *Creature*s when the Ogre died is drawn

Saving the game

writeSwampToDisk()

This simply writes the *Swamp* object to disk, using serialization. The corresponding *readSwampFromDisk()* method is in the *SwampInterface* because an object can't do anything unless it exists. Specifically, a *Swamp* doesn't exist until it has been recalled from disk or created by the *SwampInterface*.

INTERMITTENT BUG	Either this method or *readSwampFromDisk()* has an intermittent bug. When a *Swamp* is read back, *numberOfEnemiesInOgreCurrentLocation()* occasionally 'misfires'. In such cases, • an *Ogre* that has 1 immediate enemy 'ignores' it • an *Ogre* that has 2 immediate enemies 'ignores' one of them and kills the other • an *Ogre* that has 3 immediate enemies either 'ignores' 2 of them and kills the other or ignores 1 of them and is killed by the other 2. I'd be very grateful for any hint as to why this bug occurs, and why it occurs intermittently.

POLYMORPHIC PROGRAMMING

*Ogre*s and enemies all inherit the movement methods from the abstract class *Creature*. So when the Swamp has to move all the *Creature*s, it can tell them to do so without knowing in advance what kinds of *Creature* there are or anything else about them (e.g. their locations and movement methods).

Enemy is an abstract class with descendent concrete classes *Donkey*, *Parrot* etc. Because all *Creature*s move in exactly the same way, it would be possible to omit the movement methods from the concrete classes, leaving them in *Creature* only. However, having these methods in the concrete classes allows the possibility of giving each kind of *Creature* its own movement rules, over-ruling the movement methods in *Creature*.

This has been verified by changing the distance *Donkey*s and *Snake*s can move to 2 squares. (The do-while loop in *changeCoordinate()* still keeps them in the swamp). This works so long as the movement methods also remain in *Creature*. If they are removed from *Creature*, the line *tempCreature.actualMove()* in *Swamp*'s *moveAllCreatures()* reports that 'The method actualMove is undefined for the type Creature.'

A related, advantage of this polymorphism/inheritance is that it is easy to increase the number of kinds of enemies. I have verified this by creating a class similar to an existing kind of enemy but with its own subtype and suitably extending the switch statement in *addEnemyIfAppropriate()*.

It is almost certainly possible to do away with the *subType* instance variable and the *Ogre* and *Enemy* subclasses (and *Donkey*, *Parrot*, and *Snake* sub-subclasses), by making *Creature* concrete and using *Type* to record both
• whether a *Creature* is an *Ogre* or an enemy
• if it is an enemy, what sort of enemy it is.
However, doing so would remove polymorphism and its advantages from this project.

PROGRAMMING STYLE

Interfaces

- The type of interface that forces classes to implement methods is used in the different kinds of *Creature* to ensure these implement *actualMove()* and *changeCoordinate()* methods. As previously noted, these methods are currently the same throughout the project and could be removed from all classes apart from *Creature*. (This would remove any need for the *actualMove* and *changeCoordinate* interfaces.) However, to maintain extensibility, these interfaces have been left in.

- The type of interface that is separate to a class but enables interaction with it (the user-interface) is present primarily because this programming style allows very simple applications (classes with *main()* methods) and the possibility of creating other interfaces that use *Swamp*'s methods differently. Good user-interfaces help improve user-experiences. (This type of interface was also a coursework requirement.)

Exceptions
are used as follows:

In **Swamp**
- in *writeSwampToDisk()*
  IOException catches IO failures, i.e. failures to write the *Swamp* to disk. I believe the most likely cause of this is the user not having a folder at /Users/bruceryan/Documents/.

In *SwampInterface*
- in *readSwampFromDisk()*
  - *IOException* catches IO failures, i.e. failures to read from disk. The most likely cause of this is the user not having a previously saved game in the right place. (Either the user has not saved a game previously or the appropriate folder is now missing.)
  - *ClassNotFoundException*. I believe such exceptions would be caused by the program trying to use a class which is not properly available. (See http://juddsolutions.blogspot.co.uk/2008/06/tip-causes-of-javalangclassnotfoundexce.html)

In *SwampTest* (the set of JUnit tests)
- in *testWriteSwampToDisk()*
  - *FileNotFoundException*. This will do what it says. I thought it might be sensible to have it in *Swamp*'s *writeSwampToDisk()* method but Eclipse informed me 'Unreachable catch block for FileNotFoundException. It is already handled by the catch block for IOException'
  - *IOException*
  - *ClassNotFoundException*

Enumerators
are not used:

EXTENSIBILITY

The swamp is extensible in two ways:

- It is easy to amend the value of *swampSize* (in *SwampInterface*). It should be easy to add a part to the user-interface to allow the user to decide the size of the swamp.

- As noted under 'POLYMORPHIC PROGRAMMING', polymorphism allows the number of kinds of enemy to be increased fairly easily:
  - Create a new class similar to *Donkey* or another existing kind of enemy,
  - Amend *addEnemyIfAppropriate()* along the following lines:

```java
public void addEnemyIfAppropriate() {
        //Declare variables
        //chance of enemy arriving = 1 in 3
        int arrivalProb = 3;

        //All enemies have equal probability (i.e. 1 in 4)
        int enemyProb = 4

        //So chance of any particular enemy = 1 in (3 * 4)
        int decider;

        //Code
        decider = (int)(arrivalProb * enemyProb * Math.random());

        switch (decider) {
        case 0:
                Donkey d1 = new Donkey(0, 0, this.swampSize);
                this.creaturesInSwamp.add(d1);
                break;
        case 1:
                Parrot p1 = new Parrot(0, 0, this.swampSize);
                this.creaturesInSwamp.add(p1);
                break;
        case 2:
                Snake s1 = new Snake(0, 0, this.swampSize);
                this.creaturesInSwamp.add(s1);
                break;
        case 3:
                PussInBoots pib1 = new PussInBoots(0, 0, this.swampSize);
                this.creaturesInSwamp.add(pib1);
                break;
        } //end switch
} //end addEnemyIfAppropriate
```

- The following allows for enemies having different probabilities:

```java
public void addEnemyIfAppropriate() {
        //there is a 1-in-3 chance of an enemy each turn
        //there is a 40% chance of an enemy being a donkey
        //there is a 30% chance of an enemy being a parrot
        //there is a 20% chance of an enemy being a snake
        //there is a 10% chance of an enemy being a puss_in_boots
        //Hence assess chances out of 30

        //Declare variables
        int factor = 30;
        int decider;

        //Code
        decider = (int) (factor * Math.random());

        switch (decider) {
        case 0:
        case 1:
        case 2:
        case 3:
                Donkey d1 = new Donkey(0, 0, this.swampSize);
                this.creaturesInSwamp.add(d1);
```

```
                break;
        case 4:
        case 5:
        case 6:
                Parrot p1 = new Parrot(0, 0, this.swampSize);
                this.creaturesInSwamp.add(p1);
                break;
        case 7:
        case 8:
                Snake s1 = new Snake(0, 0, this.swampSize);
                this.creaturesInSwamp.add(s1);
                break;
        case 9:
                PussInBoots pib1 = new PussInBoots(0, 0, this.swampSize);
                this.creaturesInSwamp.add(pib1);
                break;
        } //end switch
    } //end addEnemyIfAppropriate
```

- Given that it's fairly easy to create extra kinds of enemy, it should be possible to have a few more, then allow the user to pick which ones to use in the game. I'd want to learn more about Swing so I could add these choices to the UI as checkboxes.

PROBLEMS ENCOUNTERED DURING THE CODING OF THE GAME

Stumbling block 1

My initial thought was that the *Swamp* should be modelled by a 2-D array of gridsquares. To set this up, I wanted to do something like

```
final int SIZE_LIMIT = 3;
String label;
for (int xCounter = 0; xCounter < SIZE_LIMIT; xCounter++) {
        for (int yCounter = 0; yCounter < SIZE_LIMIT; yCounter++) {
                label = Integer.toString(xCounter) + Integer.toString(yCounter);
                label = "a" + label;
                GridSquare label = new GridSquare(xCounter, yCounter);
        } //end y for-loop
} //end x for-loop
```

However, the underlined line does not compile.

Stumbling block 2

I next tried making my swamp a collection of *Row*s, each having a collection of *GridSquare*s and each *GridSquare* being able to house the ogre and his enemies. Adding an *Ogre* to the swamp seemed to require code to tell the relevant *Row* to tell its relevant *GridSquare* to add the *Ogre*. Then finding the *Ogre* seemed to require *Row*s to have a variable indicating the presence or absence of the *Ogre*.

I tried to keep things simple by modelling the numbers of *Creature*s as simple integer instance variables which could be incremented and decremented as needed. This was to avoid problems when moving *Creature*s. (If a *Creature* was an object, it would need to know where it was to calculate where it could move to. If it changed its coordinates, it would then need to tell its original *GridSquare* 'remove me' and its destination *GridSquare* 'add me'. I felt this would be problematic.) However, this approach had problems:

- separate, but very similar, code was needed for each type of *Creature*.
- while working through all the *GridSquare*s to move the *Creature*s, if a *Creature* was to move to a *GridSquare* that hadn't yet been handled, the *Creature* would need to go into some kind of 'holding pen' in the destination *GridSquare* to avoid being moved on again when it was time to move this *GirdSquare*'s own *Creature*s. At the end of the sequence of moves, the contents of the holding pens would need to be moved into the *GridSquare*s' 'real' areas.
- This method seemed to require a lot of Exceptions, i.e. 4 for alerting if the program tried to have negative numbers of *Creature*s at all, 4 for alerting if the program had negative numbers of *Creature*s in its holding pens, 1 for alerting if the *Swamp* had more than 1 *Ogre*, 1 for alerting if a battle didn't involve an Ogre...
- Even my code for moving the *Ogre* was suspect:
  - An ogre is placed at random:

- A suitable number of enemies arrive in the left-most square:
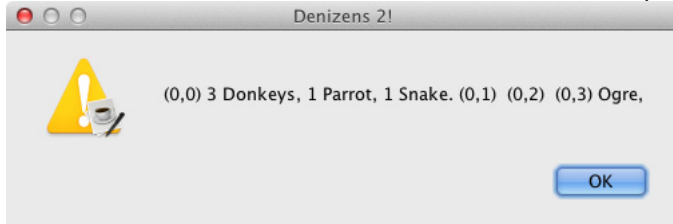
- The *Creature*s move - but the *Ogre* disappears!

I got as far as coding *Row*s and making the *Ogre* move before I realised there was a much simpler way.

Realisation

I eventually realised (on 12 April) that I didn't need to explicitly model the 'places' in the *Swamp*: the *Creature*s themselves were the only things that needed to 'know' their locations. Provided they also had rules for legal moves, they could move themselves on command. Then the system could 'ask' them where they were, either to draw the *Swamp* or to find whether there were any enemies in the *Ogre*'s location. After this, the stumbling blocks became much less fearsome: it took about half a day to write the basic *Creature* and *Swamp*, including the methods to add and move *Creature*s and to count how many immediate enemies the *Ogre* had.

Stumbling block 3

My initial method to remove a *Creature* was

```java
public void removeCreature(int xCoord, int yCoord, String subTypeToBeRemoved) {
        //Declare variables
        int indexOfCreatureToBeRemoved = 200000000;

        //Code
        for(Creature tempCreature : creaturesInSwamp) {
                if (    tempCreature.getxCoord() == xCoord
                        && tempCreature.getyCoord() == yCoord
                        && tempCreature.getSubType() == subTypeToBeRemoved) {
                        this.creaturesInSwamp.remove(indexOfCreatureToBeRemoved);
                } //end if
    } //end removeCreature
```

This almost always crashed. It took several hours to work out that the for-each block needed to finish before the removal step took place. (I also took a while to remember that indices start at 0 - this led to some lost time.)

Stumbling block 4

My methods were typed in the order I'd realised I needed them. They were long and messy, so finding the bits that needed to change was problematic. My solution was order the methods logically, using `/** comments */` to break up the code into 'chapters', then invest time in refining and reducing the number of methods.

Stumbling block 5

Occasionally my Ogre killed himself, crashing the program. I realised this was because I wasn't checking the who was fighting whom properly. This is part of the reason for having both *type* and *subType*.

Stumbling block 6

I noticed a bug in the save and restore methods - after a restore, the system intermittently miscounts the number of immediate enemies. (See also page 5). Despite spending 2 days trying to find and eliminate the cause, this bug is still present. I've tried saving the *Swamp* itself or just its ArrayList, and putting *JOptionPane* statements after every line so I can watch the program in 'slow motion'. I did notice that *moveAllCreatures()* appearing to call *moveCreature()* in *Creature* **at least twice** per Creature. An exact multiple would be more understandable.

Stumbling block 7: JUnit tests

Testing methods that return booleans

During development, I had a separate method to decide whether an enemy should be added. This method randomly returned *true* or *false*. Because the returned value is random, it is only possible to test that it is one of *true* or *false*. It would be possible to test whether the method returned the same value 50 times in a row. (The probability of this is around 1 in 10^{13}.) However, it's not impossible for this to happen. Fortunately, I was able to take this random decision into another method which was more properly testable.

Testing *drawSwampMap()*

It was frustrating to find that in Java ("AB" = "AB") = false! That is, two identical looking strings are apparently not equal. My proof of this is the following:

```
public class testString {
     public static void main(String[] args) {
          String piece1 = "donkeyOGREsnake";
          String piece2 = "donkeysnake";
          String piece3 = piece1.replace("OGRE", "");
          if (piece3 == piece2) {
               System.out.println("equal");
          }
          else {
               System.out.println("NOT EQUAL");
          }
          System.out.println("piece2: " + piece2);
          System.out.println("piece3: " + piece3);
     } //end main
} //end class
```

Running this gives
```
NOT EQUAL
piece2: donkeysnake
piece3: donkeysnake
```

Fortunately, I eventually found the *assertEquals* command.

Testing *moveAllCreatures()*

I'm not convinced this test goes far enough - it only tests whether 1 *ogre* has moved.

Testing *Battle()*, *onlyOneImmediateEnemy()*, *moreThanOneImmediateEnemy* and *actualGame()*(

Because these will display JOptionPanes to tell a user that an Ogre has killed something or has been killed, it seems reasonable to warn the tester. Is this good practice? *actualGame* requires the tester runs through a complete game. Again, is this good practice?

Software Development 2 (SET11103) Coursework: 'Get out of my swamp'

Stumbling block 8: flow of control and user-interface

At first, my interface had a dialog which required the user to enter a choice (1 to display instructions, 2 to start a new game, 3 to recall a previously saved game and 4 to exit). It was easy to trap invalid numeric input, using a while loop and techniques from last semester. It was also easy to trap non-numeric input using *NumberFormatException*.

However it wasn't easy to do both simultaneously and reliably. If the user first entered non-numeric input, he or she would receive first a response that the input was non-numeric, then a response that the input should be a number between 1 and 4. If the user then entered 1, he or she would then see the instructions, then be taken back to the main menu. If the user then entered another invalid input, the instructions would be displayed again, followed by the main menu. This was not the flow I wanted:
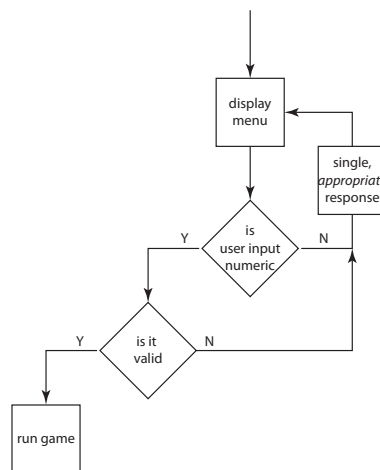
The answer was to remove any chance of inappropriate user-input. Fortunately, I had been looking fro a way to replace the standard Java dialog icons with something more appropriate to this game and had found this: http://docs.oracle.com/javase/tutorial/uiswing/components/dialog.html. While I didn't get icons to work - I think I need to understand package and Java file systems better - I was able to build a dialog with customised buttons. So now the flow is
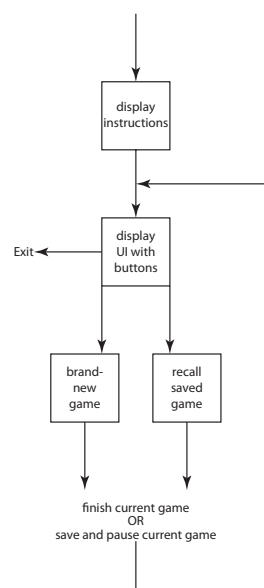
I now see the value of graphical UIs and menus to the programmer as well as to the user.