# COMP202 Winter 2005 Assignment 6

Distributed: **April 2nd**, due: **April 18th, 23:55**

## Introduction

In this assignment, you will be programming a game simulation that takes place on a hexagonal board. The game is described below: it involves creatures of several types with specific rules for capturing (battle) and moving. You will use an package previously made my Marc Lanctot which provides an API for modelling on hexagonal boards.

The assignment demands a thorough application of the Inheritance and Polymorphism concepts of object-oriented programming. It is also meant to give you an idea on the amount of effort involved in and appreciation for a real-world (games) programming task.

The assignment involves understanding and use of several items. We will describe each item on a high-level first, and then mention a few technical requirements that your program must adhere to, as well as discuss some approaches to problems you may face during your implementation.

We will first begin by explaining the rules of the game itself, then go into the technical requirements of this programming assignment, such as explaining the hexIT package that you will be using and the specific requirements to the computer simulation such as the details required (supporting only human play, suggestions for class structure, etc.).

## Hexagonal Turncoat-Chess 2

In his Ph.D. thesis Barney Pell used his METAGAME project to generate rules for a game which, after some analysis, he called Turncoat-Chess [2]. The game you will be implementing is an extended in-house version of Turncoat-Chess (TC2) on a hexagonal board called HexTC2. It was also influenced by the classic computer game Archon [1].

HexTC2 is a two-player turn-based game played on a hexagonal board with a side length of 5 hexagons (see Figure 1). The first player is randomly determined. After the first player is determined, the turn sequence continues alternately between both players. On a turn, before a player moves, there are chances that random events occur (for more details see below). After any events have occured (if any at all) and their effects applied, the player whose turn it is can choose any one piece that belongs to him and move it. A hexagon can only occupy 1 piece at most, so if a player decides to move a piece onto a hexagon that is occupied by an opponent's piece, the pieces immediately enter into a battle which will determine who gets to keep occupation of the hexagon.

There are 3 major types of creatures: slugs, termites, and fireflies. Each creature type determines how the creature moves. Slugs can only move in straight lines (one of the 6 directions dictated by the hexagonal board) and must stop if blocked either by an opponent's piece or the end of the board. Slugs can move into an occupied cell, but cannot keep moving. Termites can hop (meaning 'jump over' other creatures like the knight in Chess) in any direction as long as its path remains straight and it hops over (or onto) another termite of any color (team). Fireflies can fly, meaning they can go in all directions (not only straight) and can fly over all types of other creatures, including other fireflies.

Each creature has the following characteristics: maximum and current hitpoints (as in Assignment 4), maximum movement (MV), armor level (AL), base attack bonus (BAB), and damage potential (DP). The characteristics for each of these creatures are as follows shown in Table 1. The values in the rightmost column are special abilities, they are explained below.

| Name | Type | # | MV | MaxHP | AL | BAB | DP | Spc |
|---|---|---|---|---|---|---|---|---|
| Snale | Slug | 3 | 2 | 10 | 14 | +3 | 6 | |
| Coquille | Slug | 2 | 3 | 12 | 18 | +4 | 8 | HH |
| Ruque | Slug | 1 | 3 | 25 | 24 | +8 | 18 | H |
| Frogue | Termite | 2 | 8 | 13 | 12 | +4 | 10 | R |
| Crockpoache | Termite | 2 | 10 | 5 | 15 | +12 | 12 | |
| Nightboog | Firefly | 2 | 5 | 11 | 16 | +9 | 8 | R+H |
| Feanisk | Firefly | 1 | 7 | 20 | 14 | +13 | 11 | |

Table 1: HexTC2 Creature Information

Players start with 13 pieces in all. The number of pieces of each type is indicated in the table under the column marked #. Each piece has a color, black or white. The color of the pieces correspond to which player owns the piece: one player plays white and one player plays black. The number of each creature they get is shown in the column labeled #. Before the game begins, the players must choose where to put their creatures, but they can only do so on the 3 extreme rows (white pieces on the 3 topmost: $\{ (x, y) \mid 0 \leq x \leq 2 \}$, and black pieces on the 3 bottom most: $\{ (x, y) \mid 6 \leq x \leq 8 \}$ ).

Movement works like so: The distance on a hexagonal grid is the (minimum) number of hexagons it takes to reach one hexagon from another, considering that moving to adjacent hexagon is moving a distance of 1. A creature cannot move outside of its range: meaning that no mater what happens, a creature can only move to squares that are a distance of its maximum movement or less. For example, a Nightboog can move from (3,4) to (2,1) which is a move of 3 squares (1 NW followed by 2 W– see below), even if there are creatures along the path it takes. A termite couldn't do that because it involves two directions which makes the path not straight. A slug cannot move from (3,4) to (2,1) because its maximum movement is 2 while this move requires a movement of 3 hexagons.

Battle works like so: one creature is determined at random to go first. Then, creatures exchange attacks alternatively until one dies. When a creature attacks another, an attack roll is generated. An attack roll is a random number from 1-20 that represents the quality of the attack (1 is low, 20 is high). The attacking creature's base attack bonus is added to this roll to determine the modified attack roll. If the modified attack roll is equal to or above the opponent's armor level, then the attacked creature takes damage. The amount of damage is determined by the damage roll, in which a random number is generated from 1 to the damage potential of the attacking creature. The amount of damage is subtracted from the target creature's current hitpoints. If any creature reaches 0 or less hitpoints, it is considered dead. It is removed from the game (unless it regenerates, see below). This means that its piece is removed from the board entirely: you should no longer see it present during play, so it can no longer move, attack, or be attacked.

Each turn before a player can make a move, there is a chance that a random event occurs. Each action has a probability attached to it: this probability represents how likely the event is to occur each turn. There could be zero, one or more such events in each turn. The number of such events is determined according to the probabilities given below. How the event occurrence of events should be implemented is inclided in the next section.

- Earthquake. Pr = 0.013. All non-flying creatures take 5 damage.
- Insect's Grace. Pr = 0.034. All creatures on a random row heal 7 hitpoints.
- Worm Hole. Pr = 0.021. All creatures within a distance of 2 hexes away from some random location die instantly (their current hitpoints are set to 0). Regenerating creatures do not come back to reincarnate the next turn as they usually do after dying from a battle.
  (Hint: use `HexagonalBoard.getSurroundingArray()`)

The last items of concern in HexTC2 is the special creatures. Each player has 13 creatures, some of which are special for different reasons. The special traits are: regeneration(R), hard-hitting(HH), and healing(H).

The creatures that have special abilities are listed in the table above: for instance, Nightboogs are both regenerators and healers. A regenerator, once killed, will come back to life in one turn as randomly different creature. The player chooses where to put this creature on the board (before moving but after any other events occur). A hard-hitter's damage roll is actually 2 of its damage rolls summed up. A healer recuperates regularly gaining 1 hitpoint per turn, but never above its maximum.

Finally, a player wins by eliminating all the opponent's pieces [1] or when the opponent no longer has any valid moves.

## hexIT

You are not responsible for creating any classes or objects that represent the actual hexagonal board that the game is played on. You will be using an API that is already made for this purpose called `hexIT`, which is available here:

```
http://www.sable.mcgill.ca/~mlanct2/projects/hexIT-0.61.tar.gz
```

The documentation for the package is here:

```
http://www.sable.mcgill.ca/~mlanct2/projects/hexIT/docs/
```

Note that you need not concern yourself with all the classes in the hexIT package: mostly just hexIT.Test, hexIT.HexagonalBoard, hexIT.Hexagon, and hexIT.DrawableAdapter. As well, the important higher-level concepts follow.

There are two kinds of boards that hexIT supports that have hexagonal tiles: a diamond-shaped board called a DiamondBoard, and an actual hexagonally-shaped board called a HexagonalBoard as depicted in Figure 1. You need not concern yourself with the diamond-shaped boards since HexTC2 uses a hexagonally-shaped board.

Each hexagon is allowed to have a piece on it. In the package, the generic class for objects that can occupy a space on a hexagon is the DrawableAdapter, which implements the Drawable interface. This class has a constructor which takes a String as a parameter: the string is a file name that represents the image file that will be used by the graphics part of the package to display the piece on the board. The graphic files for this assignment are located at:

```
http://www.sable.mcgill.ca/~mlanct2/COMP202/assignment6_gfxfiles.zip
```

The files must be placed in a directory accessible by the running program. For simplicity, you might as well put them in the same directory as the source files, so that when you implement your creatures, the image files could be accessed without refering to an absolute path first. Don't worry at all about displaying the graphics such as resizing etc.: all these headaches are taken care of for you by hexIT.

The class you will be using the most is in fact the HexagonalBoard and Hexagon class. A Hexagon object represents a particular unit of space that can be occupied by or not occupied by a creature. A Hexagonal board is a collection of hexagons, where each hexagon has a x-y position on the board. All the methods in HexagonalBoard and Hexagon might be useful except the ones about field-of-view and line-of-sight which are not part of HexTC2. The documentation for these methods are available from the URL given above. The 6 directions that are present in the package are represented as public integer constants inside the HexagonalBoard class: `DIR_W`, `DIR_E`, `DIR_NW`, `DIR_NE`, `DIR_SW`, and `DIR_SE`. These are short terms for the cardinal directions north, east, north-east, etc..

The coordinate system on a hexagonal board is 2-dimensional. The x coordination represents the row, and the y coordinate represents the diagonal rank. This can be visualized using a textual representation like

[1] Interestingly enough, this is actually the opposite goal of the original Turncoat-Chess which was to win by being the first to have no legal moves. The goal in the original game is counter-intuitive which is largely due to the fact that the game rules were generated by a program (METAGAME), so we chose to stick with the traditional definition of winning.

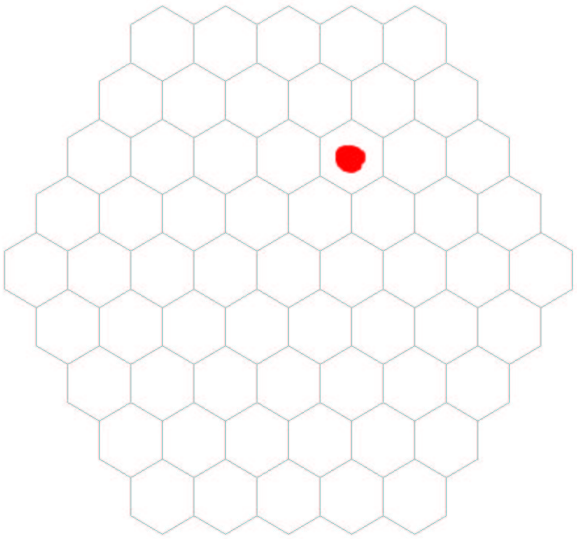

Figure 1: A hexagonal board with a side length of 5 hexagons, and a red dot at location $(x, y) = (2, 4)$

so (dots are free hexagons):

```
* Coord System & Example:
*                                        y-coord
*                                  0 1 2 3 4
*                                 / / / / / 5
*                            0-  . . . . . / 6
*                           1-  . . . . . . / 7
*             x-coord      2-  . . . . . . . / 8
*                         3-  . . . . x . . . /     <-- that 'x' is at position (3,4)
*                       4-  . . . . . . . . . .
*                        5-  . . . . . . . . .
*                         6-  . . . . . . . .
*                          7-  . . . . . . .
*                           8-  . . . . . .
*
```

As we can see, since we're using a 5x5 board, both the x-coordinates and y-coordinates range from 0 to 8. However, there are still possibly invalid coordinates, such as (6,0) and (1,8). The `HexagonalBoard.isValidCoord()` method can be used to verify the validity of arbitrary coordinates. Although, you shouldn't need to use this method very much because of all the helpful methods already available for you in HexagonalBoard.

There are other useful methods in hexIT that you might like to use. For example, given any Hexagon object, you can get its x and y coordinates using `getHexX()` and `getHexY()`. Once you create creature objects and assign them a particular color (white or black), you will have to set the image for the creature. Luckily, a method in DrawableAdapter `setImage(String filename)` – which will be inherited by all your creature classes, see below – will take care of all the details that deal with rendering the graphics.

There are much more that will help you in the package, too much to list here. You should browse the API documentation before starting your programming because it's likely that some generically useful methods are already implemented for you.

## Implementing a Simulation of HexTC2

First suggestion: Don't Panic! The assignment seems intimidating but is much easier than it seems if the concepts presented in class are used properly. This section will help you get a head start by laying down some guidelines that should be following to make your implementation more pleasant.

### Importing External Packages

The first requirement that must be met when importing classes from external packages is that the package must be in a directory name exactly the same as its package (hexIT, note the case). This directory must be somewhere in one of the directories listed in the CLASSPATH environment variable. Usually, the directory containing your source files is in the CLASSPATH by default, so the easiest way to deal with this is to put the hexIT subdirectory in the same directory as all of your source files.

For every class in your implementation that uses hexIT, the class must import the classes it needs. For example, if a source file uses HexagonalBoard, you must import it by using the import declaration: `import hexIT.HexagonalBoard; // again, notice the case`. You can also import all the classes in a package by using `import hexIT.*;`. Finally, if you desire to change the color or use the List data structures instead of arrays, you may have to also import `java.awt.Color` and/or `java.util.List`.

### Code Structure

As explained above, the generic description of a hexagonal board contains pieces on each hexagon. To implement HexTC2, the creature pieces that you intend to use must therefore be subclasses of hexIT.DrawableAdapter.

The classes and methods suggested in this section (except HexFrame, HexagonalBoard, Hexagon, DrawableAdapter which are part of hexIT) are not at all required. They are suggestion merely to help guide you through the design process. If you feel like you have a better way to implement the programming task, feel free to use your design instead.

A Creature class should extend DrawableAdapter. Three classes should extend Creature: Slug, Termite, and Firefly. The classes for each creature should extend the creature types and not be abstract since we will need objects of these types. However, this class hierarchy promotes a lot of code re-use, cutting down the amount of work for the implementation.

The simulation itself will be handled by a Simulator class. The Simulator class has one constructor that takes in 2 Player references. The Simulator class will contain only one public method: `simulate()` which will be responsible for the actual simulation the game. Since this is likely to contain a great deal of code and logic, you should decompose the method into several (private) helper methods such as `setupBoard()`, `simulateTurn(Player p)` etc. Note that because they are now part of an object, they need not be static like in Assignment 4. The Simulator class should have an instance reference of an EventEngine object (see below) for dealing with the effects of events. The simulate method should instantiate a HexFrame object and set it to be visible. And then immediately get the HexagonalBoard object that was created for you upon creating of the frame. The way to do this is like so (except your references should be instance variables instead of local variables):

```
HexFrame hf= new HexFrame("HexTC2 Simulator", 1024, 768, HexFrame.TYPE_HEXAGONAL, 5, Color.black);
HexagonalBoard board= (HexagonalBoard)hf.getHexPanel().getBoard();
// ... setup the board ...
hf.getHexPanel().setScale(true);  // scales all the graphics
hf.setVisible(true); // sets the board to visible (draws board on the screen)
```

Note that you are dealing with Java graphics, that we have not explained much of at all in class. The only thing you must know about Java graphics is this: **whenever the HexagonalBoard object is changed (ie. a piece is moved or removed), you must call the `hf.repaint()` method so that the graphics are updated properly to reflect the recent changes**.

You should have one Main driver class which contains only a main method. The main method creates 2 Player objects and 1 Simulator object, and just calls `Simulator.simulate()` which then does the majority of the game simulation work. This is a handy alternative to doing everything statically.

Use the methods in the HexagonalBoard class as much as possible! These methods will save you some time. There are methods that return arrays of Hexagons that are currently occoupied, or not occupied, and even all the valid hexagons. The typical way to randomly choose an item in an array is to randomly generate an integer index and then refer to the object by using the index:

```
Random RNG= new Random();
int randomIndex= RNG.nextInt(array.length);
Object o= array[randomIndex];
```

There is only one type of player: a human player which reads its moves from the keyboard. There are two phases, the initial placing of the pieces and the moving. The pieces are placed in order (ie. 3 snales, 3 coquilles, 1 ruque, ...). Therefore the easiest way to do this is to class called Player which has 2 methods called `int[] getPlacing(HexagonalBoard b)` and `int[] getMove(HexagonalBoard b)`. Then the `Simulator.simulatorTurn()` method can contain code for a player. The `getPlacing()` method would return 2 integers (one x-coord, one y-coord) while the `getMove()` method returns 4 integers (2 x-coords, 2 y-coords representing 2 positions: start and finish). If an illegal move is made, simply count it as a pass, but print out a message to the screen indicating that an illegal move was entered.

You should have a class called EventEngine which contains one public method called `determineEvents(HexagonalBoard b)`. This method goes through each one of the events in the list probabilistically determining if the event occurs or not (an event can only occur once per turn if at all). If so, it calls a private helper method (ie. `eventInsectsGrace(HexagonalBoard b)` that applies the effects of the event to the pieces on the board. A very helpful method for determining something with a given probability is the following:

```
/** Returns true with probability pr. */
public boolean prob(double pr)
{
   Random rng= new Random();  // note: no need to recreate this object every method invocation
                              // keep an instance of it in your object and reuse the instance
   double roll= rng.nextDouble();

   if (roll <= pr)
   { return true; }
   else
   { return false; }
}
```

You should have a Battle class which has one constructor which takes as parameters two Creature objects. It should have a public method `Creature resolve();` which simulates the battle as described above and returns the winner. This class should contain private helpers such as `boolean resolveSingleAttack(Creature c1, Creature c2)` which returns true if c1's single attack on c2 succeeded in causing damage, `causeDamage(Creature c1, Creature c2)` which determines and applies the damage c1 does to c2 on a successful hit, `Creature checkWinner()` which returns the winner if one of the creatures is dead or `null` if they are both still alive. Note that this is different than in assignment 4 where the attack methods were actually part of the Creature class. In fact, it might be intuitive to have methods in Creature such as `int getModifiedAttackRoll()` and/or `int getDamageRoll()` so that relatively little code logic need be included in the actual Battle class.

Finally, you should have 3 interfaces (one for each special ability) called Regeneratable, Healable, and HardHitter. These contains methods such as `Creature regenerate()`, `void heal()`, and `int hardhitDamage(int dp)`. The creaure classes with the corresponding abilities should implement these interfaces. One way of determining if an object has a type that implements an interface at run-time is by using the `instanceof` operator:

```
Creature c= new ....
if (c instanceof Regeneratable)
{
   Regeneratable rc= (Regeneratable)c;
   Creature newC= rc.regenerate();
}
```

However; this approach makes the implementation dependent on the type structure which in general might change in the future (which would mean going back and modifying old implementations). A slighty better way to determine if a creature is a special type is by using accessor predicates, such as having a `boolean canRegenerate()` method within the Creature class.

Your simulations need not print out the status of the operation as previous assignments have because Java will handle that via graphical display output. However, it is always a good idea to do so anyway to help with your debugging.

One hint that will help you when debugging: do the implementation step-by-step. Don't aim to implement the entire thing and then debug it all at once because that will get you nowhere. Break the large task into the following subtasks or phases, compile the program and test the functionality at each phase before continuing to the next one. If there is a bug in the core parts of the implementation, it will cause utter chaos because the rest of the code depends on the core code. Therefore, it is fundamental to build upon working implementations. If this is not done, then many errors will be present and dependent on each other; you won't even know where to begin debugging. By advancing from the foundation upwards step-by-step like this, you can deal with errors one at a time, not a slew of them all at once.

- First, aim to implement only the Player (with no actual code for selecting a move... just return an invalid one ie. null), Main, and a Simulator that only creates a HexFrame but does nothing with it other than making it visible. Compile these, and test them by running your Main class: the only thing that should happen is that a hexagonal board show up on the screen.
- Next, implement the basics of the Creature classes. Forget about the complex implementations such as the special abilities... just code the hitpoints, armor level, etc. just enough so we can use them in our next step. You can even just start off with only Snales. Then come back to do the rest after the next subtask is working properly.
- Once you have have basic creatures, start implementing the placement phase. Implement the `Player.getPlacing()` method first, compile it and test it. Once it is working with Snales, add the rest of the Creatures one-by-one testing and compiling at each step.
- Implement the `getMove()` method in Player. Implement the `Simulator.applyMove(HexagonalBoard)` helper which applies a move chosen by the player. Compile & test. Make sure the players are not allowed to make invalid moves.
- Implement the Battle class. Compile & test the battles between creatures.
- Implement the EventEngine class. Compile & test to make sure that the events are working properly.
- Implement the special abilities. Compile & test to make sure they are working properly.

# References


[1] FreeFall Associates and Electronic Arts. Archon, 1983. http://www.answers.com/topic/archon-computer-game.

[2] Barney Pell. Strategy Generation and Evaluation for Meta-Game Playing, 1993. http://satirist.org/learn-game/projects/metagame.html.