

CPSC 501 W25 T02 A1 Report
Devon Harstrom, 30132397

The original code is by Ethan McCorquodale. I was given permission to use it: "... I give you guys (and anyone else who is in the same situation) permission to use this project."

The repository of the project is here:

https://csgit.ualgary.ca/devon.harstrom/CPSC-501/-/tree/main/Assignment_1?ref_type=heads

The refactoring I performed on the original code was: 1. De-nesting, Method extraction, Creating subclasses, renaming and removing unused code, and commenting. *Each refactor is discussed below, and the writing is roughly half a page long. However, code takes up quite a bit of space on the document, so each refactor has around two pages.*

Firstly, the refactoring I performed for the first refactor was replacing nested conditionals with guard clauses. When looking at the original code, almost every function in the World.java file contained heavily nested conditional statements. This made it very difficult to determine the flow and purpose of the code. This was the code smell, long complex nested conditionals. Immediately, I knew that implementing guard clauses would improve the readability of the code, which is one of the main issues with the original code. The steps I took to complete this refactor are as follows: I first found which conditional statements resulted in the function terminating or loops continuing. This often includes the else section of the original code. For example (not part of code, just visual example):

```
if (a){  
    if (b){  
        if (c){  
            //do something  
        }  
    }  
}
```

In this case, if the first conditional never passes, then the others won't, so instead, we can separate the condition and convert it to a guard clause by taking the negation of the condition. From there, if the negation of the condition is true, then we can return. This only works if there is one main nested conditional tree and not if there are else-if statements. So, for the example above, it would look like this:

```
if (!a){  
    return  
}  
if (!b){  
    return  
}  
if (c){  
    //do something  
}
```

The original code had plenty of places in the World.java file with this code structure. After this refactoring, the code has a flat list of conditionals. This heavily improves the readability of the code as the flow is now apparent: 'a' and 'b' must be true to get to the main conditional. From this refactoring, many more refactor types can build off of it, such as method extraction separating conditionals and blocks of code into other functions and/or consolidating Conditional Expression, which moves all the conditionals into a single expression if they share the same outcome. In fact, for my refactor 2, I built off this refactor and extracted methods, which are discussed later. I altered every function in the World.java file to be de-nested, but will show one of them as an example. Originally, the function sortDead() in World.java file looked like this:

```
public void sortDead() {
    for (int i = 0; i<=elfCounter+orcCounter;i++) {
        if (loc[i] != null) {
            if ((aWorld[loc[i].getRow()][loc[i].getColumn()] == null)&&(loc[i+1] != null)) {
                loc[i] = loc[i+1];
                loc[i+1] = null;
                if(i-2 >= 0) {
                    if(loc[i-2]== null) {
                        sortDead();
                    }
                }
            }
        }
    }
}
```

Using this refactoring, I adjusted it to have a flat conditional list.

```
public void sortDead() { //supposed to try and sort null elements and bring them to the end
    doesn't always work as intended
    for (int i = 0; i<=elfCounter+orcCounter;i++) {
        if (loc[i] == null) {
            continue;
        }
        if ((aWorld[loc[i].getRow()][loc[i].getColumn()] != null)|| (loc[i+1] == null)) {
            continue;
        }
        loc[i] = loc[i+1];
        loc[i+1] = null;
        if(i-2 >= 0 && loc[i-2]== null) {
            sortDead();
        }
    }
}
```

All the code was tested through junit tests for each altered function (all of them) to see if the same values/results were achieved. Since it is mainly a GUI project, I compared the outcome of the GUI across both the original and refactored versions. For example, this code used this test:

```
@Test
void sortDeadTest(){
    ... load map by entering code in terminal (removed code for report)

    world.sortDead();
    assertNull(world.location[world.location.length - 1]);
    assertNull(world.location[world.location.length - 2]);
    assertNull(world.location[world.location.length - 20]);
    System.out.println(world.location[world.location.length - 20]);
}
```

Which checks if the null is at the end of the location array. All these tests passed. All these refactors were done on the refactor_1 branch and 05eeace9b8fb646f577847264f1b165eaec127a5 is the SHA of the commit for the completion of this function while this is for the entire complete refactor: e0863597e3e809388191adf14b843be33559d112

The next refactor (refactor 2) was built off refactor one and is “method extraction.” The other immediate code smell was long methods. Even after de-nesting the conditionals, the methods were still too long and confusing. For all the functions in World.java except for sortDead(), sortLocation(), and checkPerim(), I applied this refactor. The methods exceeded 50+ lines in some places and were generally large and confusing, so extracting methods would greatly benefit the code. On top of this, there were some places where code was being duplicated, which could benefit from being one function call. For example in the move() function there are these conditionals

```
if (loc[index].getCanMove()) {
    if ((loc[index].getRow()-1 != -1)&&(loc[index].getColumn()-1 != -1)) {
        if ((loc[index].getType() ==
            'E')&&(aWorld[loc[index].getRow()-1][loc[index].getColumn()-1]== null)) {
            oldRow = loc[index].getRow();
            oldCol = loc[index].getColumn();
            loc[index].setRow(oldRow-1);
            loc[index].setColumn(oldCol-1);
            aWorld[oldRow-1][oldCol-1] = aWorld[oldRow][oldCol];
            aWorld[oldRow][oldCol] = null;
        }
    }
    if ((loc[index].getRow()+1 != 10)&&(loc[index].getColumn()+1 != 10)) {
        if ((loc[index].getType() ==
            'O')&&(aWorld[loc[index].getRow()+1][loc[index].getColumn()+1]== null)) {
            oldRow = loc[index].getRow();
            oldCol = loc[index].getColumn();
            loc[index].setRow(oldRow+1);
            loc[index].setColumn(oldCol+1);
            aWorld[oldRow+1][oldCol+1] = aWorld[oldRow][oldCol];
            aWorld[oldRow][oldCol] = null;
        }
    }
}
```

These can be extracted into one method using this refactoring method which I did here:

```
public void moveLogic(char entityType, int index, int increment){
    if ((loc[index].getType() == entityType)&&(aWorld[loc[index].getRow()+
increment][loc[index].getColumn()+increment]== null)) {
        oldRow = loc[index].getRow();
        oldCol = loc[index].getColumn();
        loc[index].setRow(oldRow+increment);
        loc[index].setColumn(oldCol+increment);
        aWorld[oldRow+increment][oldCol+increment] = aWorld[oldRow][oldCol];
        aWorld[oldRow][oldCol] = null;
    }
}
```

This removes duplicate code and increases readability by reducing the size of methods. The move function went from this:

```
public void move(int index) { //changes the value of aWorld[][] must set old position to null
then update it to the new position.
    if (checkPerim(index) == true){ //checks the perim for things not null if it finds
something it returns true and the move code below doesn't run
        return;
    }
    if (loc[index] == null) {
        return;
    }
}
```

```

    }
    if (loc[index].getCanMove()==false){
        return;
    }
    if((loc[index].getRow()-1 != -1)&&(loc[index].getColumn()-1 != -1)) {
        if ((loc[index].getType() ==
'E')&&(aWorld[loc[index].getRow()-1][loc[index].getColumn()-1]== null)) {
            oldRow = loc[index].getRow();
            oldCol = loc[index].getColumn();
            loc[index].setRow(oldRow-1);
            loc[index].setColumn(oldCol-1);
            aWorld[oldRow-1][oldCol-1] = aWorld[oldRow][oldCol];
            aWorld[oldRow][oldCol] = null;
        }
    }
    if((loc[index].getRow()+1 != 10)&&(loc[index].getColumn()+1 != 10)) {
        if ((loc[index].getType() ==
'O')&&(aWorld[loc[index].getRow()+1][loc[index].getColumn()+1]== null)) {
            oldRow = loc[index].getRow();
            oldCol = loc[index].getColumn();
            loc[index].setRow(oldRow+1);
            loc[index].setColumn(oldCol+1);
            aWorld[oldRow+1][oldCol+1] = aWorld[oldRow][oldCol];
            aWorld[oldRow][oldCol] = null;
        }
    }
    if ((GameStatus.debugModeOn == false)|| (loc[index] == null)) {
        return;
    }
    if (loc[index].getType() == 'E') { //elf move data
        status.elfMoveDebug(oldRow,oldCol,loc[index].getRow(),loc[index].getRow());
    } else if (loc[index].getType() == 'O') { //orc move data
        status.orcMoveDebug(oldRow,oldCol,loc[index].getRow(),loc[index].getRow());
    }
}

```

To this:

```

public void move(int index) { //changes the value of aWorld[][] must set old position to null
then update it to the new position.
    if (checkPerim(index) == true){ //checks the perim for things not null if it finds
something it returns true and the move code below doesn't run
        return;
    }
    if (loc[index] == null) {
        return;
    }
    if (loc[index].getCanMove()==false){
        return;
    }

    if((loc[index].getRow()-1 != -1)&&(loc[index].getColumn()-1 != -1)) {
        moveLogic('E', index, -1);
    }
    if((loc[index].getRow()+1 != 10)&&(loc[index].getColumn()+1 != 10)) {
        moveLogic('O', index, 1);
    }
    if ((GameStatus.debugModeOn == true)&&(loc[index] != null)) {
        debugMovement(index);
    }
}

public void debugMovement(int index){

```

```

        if (loc[index].getType() == 'E') { //elf move data
            status.elfMoveDebug(oldRow,oldCol,loc[index].getRow(),loc[index].getRow());
        } else if (loc[index].getType() == 'O') { //orc move data
            status.orcMoveDebug(oldRow,oldCol,loc[index].getRow(),loc[index].getRow());
        }
    }
}

public void moveLogic(char entityType, int index, int increment){
    if ((loc[index].getType() == entityType)&&(aWorld[loc[index].getRow()+
increment][loc[index].getColumn()+increment]== null)) {
        oldRow = loc[index].getRow();
        oldCol = loc[index].getColumn();
        loc[index].setRow(oldRow+increment);
        loc[index].setColumn(oldCol+increment);
        aWorld[oldRow+increment][oldCol+increment] = aWorld[oldRow][oldCol];
        aWorld[oldRow][oldCol] = null;
    }
}

```

The steps I took were to look for conditionals, loops, duplicate code, or long portions of code. From there, I first created a new method with a good name, pulled those chunks of code out, removed it from its prior spot and calling the new method. Next, I put the code chunk in the new method and passed the required parameters, such as local variables. This refactoring could go further by potentially moving these functions to separate classes if the file ends up being too long, but for the simplicity of the project, I believed this would be an unnecessary change. The code was again tested on the same test file for example, move was tested with this, which sets up a mock world and checks if the orc moved after calling the function.

```

@Test
void testMove(){
    String simulatedInput = "Assignment_1/refactored_code/assignment3GUI -
Classmate's/src/moveTest.txt\n";
    ... using system.in to set world

    assertEquals('O', world.aWorld[0][0].getAppearance(), "should be an Orc at 0,0");
    System.out.println(world.aWorld[0][0].getAppearance()+ " at: 0:0");
    world.move(1);
    assertEquals('O', world.aWorld[1][1].getAppearance(), "should be an Orc");
    System.out.println(world.aWorld[1][1].getAppearance() + " at: 1:1, from move function" );
}

```

All these functions were refactored on the refactor_2 branch, and a144509c43132183846d504308374bed8bf2e49c is the SHA of the commit for the completion of the move function, which is the last function I completed.

Next, for refactor 3, the prominent code smell was a “large class.” There was too much information within the class for such a simple purpose. This was within the Entity.java class, which is for both orcs and elves. From this ‘smell,’ I decided it could be refactored to implement subclasses. Orc and Elf could be a subclass of entity and have the health, damage, and appearance in their own classes. This would improve readability and understanding. While also decreasing the complexity of the Entity class. This is what the original Entity class code looked like:

```
public class Entity
{
    public static final char DEFAULT_APPEARANCE = 'X';
    public static final char ELF = 'E';
    public static final char EMPTY = ' ';
    public static final char ORC = 'O';
    public static final int DEFAULT_HP = 1;
    public static final int ORC_DAMAGE = 3;
    public static final int ELF_DAMAGE = 7;
    public static final int ORC_HP = 10;
    public static final int ELF_HP = 15;

    private char appearance;
    private int hitPoints;
    private int damage;

    boolean attacking= false;

    public Entity()
    {
        setAppearance(DEFAULT_APPEARANCE);
        setHitPoints(DEFAULT_HP);
    }
}
```

```

public Entity(char newAppearance)
{
    appearance = newAppearance;

    hitPoints = DEFAULT_HP;

    damage = ORC_DAMAGE;
}

public Entity(char newAppearance, int newHitPoints, int newDamage)
{
    setAppearance(newAppearance);

    setDamage(newDamage);

    setHitPoints(newHitPoints);
}

// ... plenty of getter setters

```

This class is too long and can be split up, so only the Orc subclass has information on Orcs. The main refactoring techniques I used were “extracting subclass” and some “polymorphism.” My steps were to create two new classes and transfer all the relevant variables to the new subclasses, such as ELF_HP or ELF_DAMAGE, etc. I then create a unidirectional relationship with Entity for both Orc and Elf. They inherit properties through extension but not the other way around. I then use polymorphism to move the debug statements specific to the two entities into the subclasses. Lastly, I moved the elf and orc counter from the world.java file into the entity class, as it made more sense to have entity counts in the entity class. These refactors are beneficial as they establish a relationship that improves overall readability and understanding of the code structure and enhances the readability of the entity class by moving variables to the necessary subclass. This also adds the start of producing code that could be extended, for example, other entity types like dwarves, which further benefitted from polymorphism. Other refactors could come from this, such as dealing with primitive obsession and replacing static variables with local subclass variables. After the refactoring, the new entity class looks like this:

```

public abstract class Entity
{
    // public static final char DEFAULT_APPEARANCE = 'X';

    public static final char EMPTY = ' ';
}

```



```
//      public static final int DEFAULT_HP = 1;

    private char appearance;

    private int hitPoints;

    private int damage;

    private static int orcCounter = 0;

    private static int elfCounter = 0;

    public Entity(char appearance, int hitPoints, int damage) {

        this.appearance = appearance;

        this.hitPoints = hitPoints;

        this.damage = damage;

    }

    public abstract void debugAttack(GameStatus status, int tempRow, int tempColumn, int
targetRow, int targetColumn, int HP, int HPAfter);

    public abstract void debugLossConditions(GameStatus status, int entityCounter);

    public abstract void debugMovement(GameStatus status, int oldRow, int oldCol, int newRow,
int newCol);

// ... more getters and setters
```

While Elf looks like this (Orc is the same except with orc variables, etc.):

```
public class Elf extends Entity {

    public static final char ELF = 'E';

    public static final int ELF_DAMAGE = 7;

    public static final int ELF_HP = 15;

    public Elf() {

        super(ELF, ELF_HP, ELF_DAMAGE);

    }

}
```

```

@Override

    public void debugAttack(GameStatus status, int tempRow, int tempColumn, int targetRow, int
targetColumn, int HP, int HPAfter) {

        status.elfAttackDebug(tempRow, tempColumn, targetRow, targetColumn, HP, HPAfter);

    }

@Override

    public void debugLossConditions(GameStatus status, int entityCounter){

        status.orcLoseDebug(entityCounter);

    }

@Override

    public void debugMovement(GameStatus status, int oldRow, int oldCol, int newRow, int
newCol){

        status.elfMoveDebug(oldRow,oldCol,newRow,newCol);

    }

}

```

Testing for these changes was with two tests called testEntities() and testEntitiesCounters():

```

@Test

void testEntities(){

    Entity testElfEntity = new Elf();

    Entity testOrcEntity = new Orc();

    assertNotNull(testElfEntity, "should not be null");

    assertNotNull(testOrcEntity, "should not be null");

    assertEquals('E', testElfEntity.getAppearance());

    assertEquals('O', testOrcEntity.getAppearance());

}

```

```

    assertEquals(7, testElfEntity.getDamage());

    assertEquals(3, testOrcEntity.getDamage());

    assertEquals(15, testElfEntity.getHitPoints());
    assertEquals(10, testOrcEntity.getHitPoints());
}

@Test
void testEntityCounters() {

    assertEquals(0, Entity.getElfCounter());

    assertEquals(0, Entity.getOrcCounter());

    Entity.incrementElfCounter(5);

    Entity.incrementOrcCounter(-1);

    assertEquals(5, Entity.getElfCounter());
    assertEquals(-1, Entity.getOrcCounter());

    Entity.setElfCounter(0);

    Entity.setOrcCounter(0);

    assertEquals(0, Entity.getElfCounter());

    assertEquals(0, Entity.getOrcCounter());
}

```

They check things like if the entity is not null, appearance, damage, health, counter, and increment. It also uses the same tests as before to make sure nothing has changed. And again

comparing GUI outputs with the original. All these refactors were done on the refactor_3 branch, and f48e707d4dd492bba6c38908c0350463543e4f3e is the SHA of the commit for the completion of the entire refactor.

Next, for refactoring 4, I did many small refactoring changes in many of the files for various ‘code scents.’ These were mainly dead code, poor variable naming, unnecessary global variables, warnings, and a lack of general code consistency. I refactor these issues by removing dead code, changing variable names, fixing warnings, and changing some conditionals. These are all very simple. The steps for dealing with dead code are first to see if certain parts of code are being called or used anywhere, such as this, from the original code: (in world.java)

```
public static final int SIZE = 10;

public static final int ORCS_WIN = 0;

public static final int ELVES_WIN = 1;

public static final int DRAW = 2;

public static final int CEASE_FIRE = 3;

public static final int NOT_OVER = 4;
```

Which I found to not be used or called anywhere and then I simply removed it from the code. This helps reduce confusion with code that isn’t being used, improving readability. The other refactor is changing names, the steps are to find poorly named variables, methods, classes, etc. and giving them a better name. For example, I change this (in places like GamePanel.java):

```
for (int r = 0; r<world.SIZE;r++) {

    for (int c = 0; c<world.SIZE; c++) {
```

To this, which makes it more readable.

```
for (int row = 0; row<world.SIZE;row++) {

    for (int column = 0; column<world.SIZE; column++) {
```

In the World.java file, I moved these two variables to local variables

```
int HP;
int HPAfter;
```

The steps I took were very similar to the name changes. I looked to see where these were called and if they needed to be global. I could create them locally if I found that it wasn’t. I also refactored conditionals like this: (from GamePanel.java)

```
if (GameStatus.debugModeOn == false) {...}
```

And changing it to more conventional code:

```
if (!GameStatus.debugModeOn) {
```

The steps are just to find code that doesn’t fit these conventions and adjust them. All of these small refactors improve the overall readability of the code and reduce confusion and clutter. They

also solve the code smell of comments where good code should not need comments. These types of refactoring are more from the results of other refactors and not so much a branch for other refactoring techniques. These refactors were tested using the same test file and comparing GUI to make sure nothing had changed. These refactors were done on the refactor_4 branch, and 1dc732eab1b8aa6b3c35d2950b89096ac7024f42 is the SHA of the commit for the completion of the entire refactor list.

Lastly, for Refactor 5, I dealt with the code smell of comments. However, I added comments and will argue that adding comments is a valid refactoring method to improve readability. The Fowler text discusses writing code without the need for comments. While I agree, adding comments enhances the readability of code. Comments certainly can be overdone and cannot make bad code good. However, if you aim to create good code, comments separately amplify the readability of the code. Code should not rely on comments, but they can improve it. So, I added comments to some of the functions in the World.java file for this refactor. For example, the victoryLogic() function: (before and after):

```
public void victoryLogic(Entity attackingEntity) {
    if (attackingEntity instanceof Elf && Entity.getOrcCounter() > 0) {
        return;
    }
    else if (attackingEntity instanceof Orc && Entity.getElfCounter() > 0) {
        return;
    }
    System.out.println(attackingEntity.getClass().getSimpleName() + " wins, no
more opponents left.");
    programRunning = false;
}
```

```
// if there is no more enemies, then the attackers have won so print out who
won and end the game
public void victoryLogic(Entity attackingEntity) {
    if (attackingEntity instanceof Elf && Entity.getOrcCounter() > 0) {
        return; // if there still exists orcs and elves are attacking then no
victory
    }
    else if (attackingEntity instanceof Orc && Entity.getElfCounter() > 0) {
        return; // same exact thing but opposite
    }
    // get the name of the attacking entity since they won
    System.out.println(attackingEntity.getClass().getSimpleName() + " wins, no
more opponents left.");
    programRunning = false; //end game
}
```

The comments in this allow readers to quickly understand the idea of a function without actually needing to read through the function. Also adding occasional comments inside to explain further. The reason behind this is that while yes it is true that good code doesn't need comments, describing a piece of code in a sentence shows you truly understand the code you are writing, and if you know the code better, you write better. The steps I took for this are to simply find areas of code that could use explanation and write a sentence in your own words as if you are explaining it. Like in victoryLogic(), what would this function do without reading it? Well, in a sentence, I can write what it does so you know before looking at the code. This is usually the end result of refactors and not a branch for more, similar to refactors made in Refactor 4. This

refactor did not require testing as it's just comments. All these functions were refactored on the refactor_5 branch, and 2f79f59ff4852b4dec916b7a30842e506495b49e is the SHA of the commit for the completion of the move function, which is the last function I completed.

These are all the refactors I made for Assignment 1, but many more could be possible; these are just the ones I chose to do for the assignment.