Assignment 1 Report:

Design report:

Since this was the first time we properly were creating a program from scratch with knowledge of design patterns I'm quite happy how it turned out.

For my game loop, I knew if I didn't decide on something smart it would turn out to be a disgusting mess. I looked at my previous assignments and used a great website (http://gameprogrammingpatterns.com/contents.html) as inspiration.

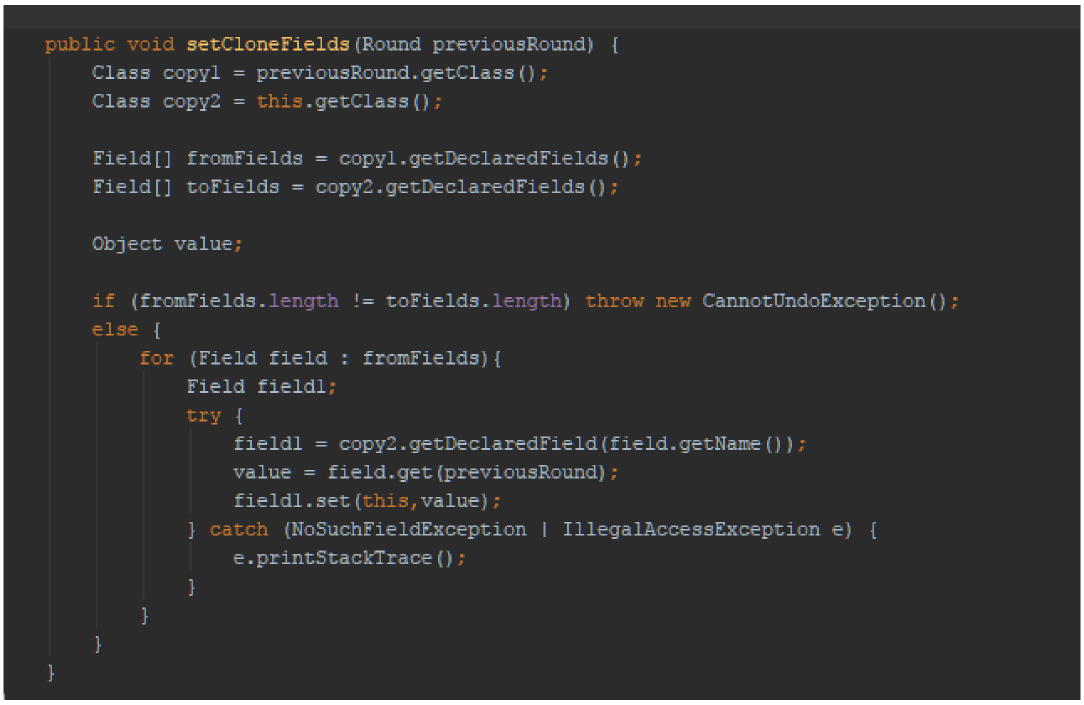
I decided on using a mix of a command pattern to implement a call-back system and using the knowledge from COMP261, a recursive descent parser to parse my user input and return my call back in the form of a generic 'ParseNode'. These ParseNodes all have their individual execute methods which are executed via the callback of the command pattern.

Also, I attempted to use a Model View Controller system so my Controller (inputController) returned a method to manipulate the model. The model as a result updates the board views. I'm not sure if I designed this perfectly and separated them equally as well but I think I made it work OK enough. I'm happy with the way my dependencies ended up and the way my class diagram looks. Admittedly I did auto generate using IntelliJ, but this was good because it allowed me to visually see where I could clean up the design of my program and where I had unnecessary references to classes that could just be passed into method bodies. I also put a lot of thought into what methods should go where to improve cohesion so each class only really handled events that would change its own state.

Besides the boardDrawer class which was a monolith structure to draw the board (sorry), I think the method names were pretty good and what they did was concise and explained well in JavaDocs. I also used method overloading too to make my code more flexible.

Smart Code highlight:

I think given I started undo at about 10pm on Sunday night I thought my implementation was quite good. I used a stack that contained Rounds. These rounds were pushed onto the stack every successful move. If undo was called, it would pop off the round on the top of the stack, I.E the previous state of the round. Simple enough.
I needed a way to reassign the fields of my current round instance to the fields of the one I had just popped off. I immediately thought of using Java reflection. Hence, the following code:



```java
public void setCloneFields(Round previousRound) {
    Class copy1 = previousRound.getClass();
    Class copy2 = this.getClass();

    Field[] fromFields = copy1.getDeclaredFields();
    Field[] toFields = copy2.getDeclaredFields();

    Object value;

    if (fromFields.length != toFields.length) throw new CannotUndoException();
    else {
        for (Field field : fromFields){
            Field field1;
            try {
                field1 = copy2.getDeclaredField(field.getName());
                value = field.get(previousRound);
                field1.set(this,value);
            } catch (NoSuchFieldException | IllegalAccessException e) {
                e.printStackTrace();
            }
        }
    }
}
```

I thought in the grand scheme of things this was good because I was using something that beyond the SWEN221 labs I hadn't really seen the need for and two, it meant any change to the round class meant that this code would still work and wouldn't break the undo functionality.

I suppose I could have commented on the fact I parsed in all my data for my pieces so I didn't need to create heaps of very similar code. Or how I used a recursive descent parser to return a generic ParseNode to be utilised in a command pattern.

Testing Discussion:

So as the way the program worked was using user input, I obviously couldn't write tests using user input. I tried using a inputByteStream and passing that in which worked. But it wasn't ideal, it would only allow me to pass one command and it meant I had to use my game loop which meant it never quit the test… infinite loop yay.

My solution was to make a new state for my program, 'testing'. This state allowed me to not use my game loop. Problem one solved. Due to the fact my Parser returns nodes that could be executed by my controller to alter my Model, it meant I could create different types of nodes and execute them to alter my game state. i.e createNode and moveNode etc. For each test, I added to this queue and ran the test which popped off all the nodes on the queue and executed them, altering my game state. Thus, no user input but workable testing.

I tested as I was going along, some bugs I just fixed and didn't add tests for because they were trivial, some I did. I could have done more tests but I spent a decent amount of time debugging alongside that I feel that the gameplay should be correct, or very accurate. I was struggling to create bugs.