D7032E Home Exam

by Viktor Magnusson

1. Unit Testing

1.

For the first requirement, the game is currently programmed to have either two local players or one local and one remote player failing the test but we can test player count by asserting *server.Players.size()* == 2. We can't test the code since there is no functionality to test.

2 c)

This requirement fails since instead of shuffling in Forest 6 & 2 it shuffles in Forest 6 & 4.

```
Vector<Card> expectedCards = {new Card("Field", null, …..), new Card("Field", …..), …..);
expectedCards.get(0).diceRoll = 3;
…
Vector<Card> actualCards = Card.regions;
actualCards.sort(diceRoll, name); //shortend sudo Code
expectedCards.sort(diceRoll, name); //shortend sudo Code
assertEqual(actualCards, expectedCards);
```

4 a ii 1)

The code only counts wool and gold when it should count all resources.

```
List<List<Card>> priTest = {....} //What principality you want to test
List<List<Card>> priExpected = {....} // What the result should look like
Server server = gameSetup(priTest) //Setup the game with custom principality till were first eventide is cast
while( !System.out.contains("[EventDie] -> 1") ){ server.gameSetup(priTest); }
assert( server.players.get(0).principality == priExpected );
// redo above code for multiple setups.
```

4 a ii 2)

The code calculates trade advantage incorrectly, awarding any player who has more than 3 commerce points. Also a player may receive 0 resources by entering an invalid resource (This also goes for celebration & plentiful harvest). Tested the same way as 4 a ii 1) but you need to set commerce points on both players and change the event die expected.

4 b)

The cards *Brigitta the Wise Woman* and *Scout* can both be played during the action face when they should be playable in other circumstances. Proper use is untestable since functionality does not exist in code. Some action cards result in the default (+1 vp) action. The city cannot be placed on top of a settlement

Player actual, other = new Player(....), new Player(...);
boolean result = new Card(.....).applyEffect(actual,other,col,row);
assert(result = expectedBoolean);
assert(actual = new Player(....));
assert(other = new Player(.....));

4 d)

You cannot see what cards you can choose from when exchanging for 2 resources or what cards you got from the replenish step, you also cannot choose what stack you draw from untestable since functionality is not implemented in code.

5)

The vp for the initial principality setup is not counted, should be tested under 2).

2. Software Architecture design and refactoring.

When it comes to SOLID principles I would say that the original code could be argued to not follow an object oriented design which is the basis for the SOLID principles, by that I mean that if you put all of the code in one files and made the objects into structs it would not change the overall design of the code that much and thereby the benefits of object oriented programming is lost. When it comes to BOSH metrics the original code fails on every point at least somewhat. Cohesion, coupling and primitiveness is very bad since the classes are very dependent on each other. Then the program has several features not implemented.

The main design patterns are for handling the cards and for handling the turns. For the cards I went with an entity component system design with cards replacing entities.

d & c.)

With ESC every card only consists of a map of cardComponents which can add any type of information to the cards and cardSystems which add functionality by using these cardComponents with a CardHandeler so there is only one entryway for managing cards. The reason I use an ESC is because it fulfills **extensibility** extremely well, all you need to add a card with new functionality is to make a new cardComponent to mark the card for the new functionality and possibly store some information and then add a cardSystem to handle the new functionality. When it comes to **testability** an ESC makes it easy to test the different cardSystems with the coupling between them being very low (if implemented correctly) and therefore making it easy to separate the functionality that needs to be tested.

For the loading/setup of the cards, strategy design pattern has been used with it being three parts, each part replaceable by another future implementation to accommodate for expansions. The first part picks out the cards from the jsonfile that will be used. The second part builds the cards by adding the right card components and the third part is a post processing step needed primarily to add the die values to the regions. The first part is meant to be replaced if the cards needed cannot be simply understood by the card theme, tournament mode for example. The second class is meant to be extended as new components are added to functionality with it maybe not being easily readable from the json file what type of component should be added. These three separated steps are meant to accommodate **Modifiability.**

For going through the game turn I have made a design pattern meant to make the turn heavily **modifiable** by either different modes or by card effects. This is done by having mostly everything done in a game turn be done as interfaces with the **gameTurn** class having a list of what current methods are to be used. This allows manly cards to heavily modify how a game turn works by for example changing how a die roll works to it being predetermined. This also allows for very good **testability** as every part of a turn is a separate interface and each implementation can be separately tested.

b.)

Completeness is very bad as my code isn't finished.

Single responsibility is something that have been chief were even though my not finished code has been separated enough to warrant 98 files, there is no file I would combine except how some of the cardComponents are implemented specially region/rotate/hasSymbolReward/symbolHolding cardComponents since they are unnecessarily separated in some instances. There is nowhere in the code where you would need to change the code to implement probable new features (although some would probably pop up if I finished the code) only some parts could need extension (mostly factories) fulfilling **open-closed principle**. When it comes to **Liskov's Substitution Principle** there is only one case of subtyping where the root is used (the aforementioned cardComponents) where this unfortunately breaks at the last subtype. **Interface segregation** is done by not ever using multiple interfaces on any class. **Dependency inversion** has not become much of a problem or consideration since the high level modules have not been implemented.

Coupling is very low in the code mostly thanks to the ECS, making it unnecessary to reference specific cards outside of factories. **Cohesion** is held very high by having separate systems communicate through narrow rules, examplewise having all card functionality move through the cardHandler and its cardSystems. **Primitiveness** has not been followed as well as it should be as most of the information holding classes have been given unnecessary functionality, this is not an issue of the architecture and could easily be fixed by a refactor.

Since one requirement was to have network capabilities and such is not done, **Sufficiency** is not done.