Testing report

Original Documents:
[Test Report]
[Other Test Documents]

Unit Tests

We executed Fractal’s initial Unit Test suite which had been created in JUnit as they described. The results of this combined with our extended test suite can be found on the website. We had a 100% pass rate of these tests. This was critical as the game should be in a finished state at this point. Ready for open days. Alongside these automated tests we had to remove some of their tests which failed due to the game not interfacing correctly with the headless application. Discussed more in Manual Testing (See below). All of these tests were created through our development process and then the project was tested with them in its finished state to confirm it was in fact complete.

Manual Testing

Following a discussion with Fiona after realising Fractal had not made their application testable with either their included headless application or the variant we used last assessment. We decided instead to forgo the efficiency of automatic testing of certain elements because the game simply did not allow for it in that state. Therefore some elements have been tested manually, the testing methodology and results for these exceptions can be found listed below. Any class which requires the main game to be functioning, cannot be tested with the standard JUnit config.

AIPlayer

Methodology:

These functions were both tested in context (Playing a vs AI game) and tested thoroughly during development and upon completion in debugging mode. Context can be replicated simply by having the phase descriptions available to you and playing the game, check that at every Human turn the AI has clearly made the correct progress for its last turn. So check it owns another tile and roboticon, check the market has either been bought or sold from. This is essentially black box testing. The white box tests we ran involved placing breakpoints on each other functions below, and the case select statement in RoboticonQuest’s implementPhase. These points provide sufficient information about the states of the system whilst also making progress between them.

Functions & Tests:

Alongside all the individual tests for phases, they are able to correctly implement the nextPhase button from the UI. Phase 4 is handled automatically so no interaction is required. Gameflows as expected with the game vs AI.

takeTurn(int phase)

> Function is intended to take a parameter between 1 and 5 and execute call the according phase as expected. It performs as follows:

Integer Passed	Expected	Actual
1	Calls phase1()	Calls phase1()
2	Calls phase2()	Calls phase2()
3	Calls phase3()	Calls phase3()
4	Default // no action	Default // no action
5	Calls phase5()	Calls phase5()
Other	Default // no action	Default // no action

random(int max)

Uses the java library Random, this function is merely a convenience and provides no real contribution to the architecture.

phase1()

Intended to simulate the interaction a player has with phase 1 (Buy a single land plot). Does not check for money as the AI is started with 20000 money, this is to compensate for it making less than optimal decisions with buying and selling. This compensation ensures it makes visible progress every turn and makes the player feel like they are actually playing against an opponent.

Works exactly as expected both anecdotally and whilst examining a trace table of all variables and objects. The class will always buy an open land plot, it never buys more than one land plot. These are selected at random. The selection is also completely dynamic, in a map with only 4 tiles, it performs identically to a map with 20 tiles as all variables are taken directly from the plotManager.

phase2()

Intended to simulate player interaction with phase 2 (purchase and upgrade roboticons). The AI purchases a roboticon and upgrades in accordance with the first tile it owns which doesn't have a roboticon. I.e. If it's first tile without a roboticon has 10 ORE, 3 ENERGY and 2 FOOD. It will purchase an ORE upgrade for the corresponding Roboticon. In the event the market is out of roboticons, the AI will try and purchase one, be caught and head to the next phase. This doesn't affect gameplay significantly, however every turn the AI is unable to purchase a roboticon it will never catch up to each tile having a roboticon as it only buys one per turn. This situation arises rarely due to the consistent selling to the market from the AI. And honestly adds to the strategy of the player with denying ORE to the market.

phase3()

Intended to simulate interaction with phase 3 (Install any roboticons purchased). The AI will iterate through all its currently owned tiles, if any don't have a roboticon, it will then iterate through its roboticons and if any are uninstalled it will install them on that tile. This works as intended, and due to it sharing the same iteration as phase 2, it will implement the roboticon previously bought on the correct tile without having to stored the value of the tile alongside the roboticon.

phase5()

Intended to simulate interaction with phase 5 (buy and sell from market).
The AI does not gamble as it doesn't contribute to end score and is also not immediately visible to the Player. The AI makes a simplistic decision to either buy or sell resources from the market just to keep it alive and interesting for the Human player. It either buys 5 of every resource it can, or it sells half of all of its resources with amounts greater than 1.

It performs exactly as expected in this regard. Reliably alternating between selling and buying.

The following two functions are utility functions to make phase5 more maintainable should any changes occur to market or Player classes.

sellResources(ResourceType type, int amount)

Calls Player.sellresourceToMarket and prints what it is doing. Allows for easy following of process in debugging and maintainability in the future.

buyResources(ResourceType type)

See above description. Has hard coded value of 5, all calls to this function expect 5 of the resource to be purchased.

RoboticonQuest

Methodology:

These tests were specified by Fractal at Assessment 2. However now that we are aware of the flaw in their testing we decided to test them again to ensure correctness. As with AI we tested these both in the full context of the game and following all variables and objects in debugging mode with break points on the functions of interest.

Functions & Tests:

phaseTest()

Check Game starts in correct phase, progresses, and can reset
Breakpoints placed on:

reset(boolean AI)
implementPhase()
nextPhase()

RoboticonQuest performed as expected. Initialising at phase 1 and progressing correctly upon interaction with the UI. When phase enters 6, a wincheck is performed to determine if the game should continue. If this fails then phase is set to 1 and the next player is called to take their turn. As Fractal showed last time, all phases work correctly when called.

playerTest()

Test game starts at player 0 and increases at appropriate interval
Breakpoints placed on:

implementPhase()

nextPhase()

Monitor the currentPlayerIndex variable in particular

This test was conducted alongside the previously discussed test. Every time phase 6 is reached the currentPlayerIndex is swapped and the next player is able to take their turn. We also confirmed the game initialises at currentPlayerIndex = 0.

PlayerEffect

Methodology:

Our work in this assessment led to the introduction of random effects, which were divided into PlayerEffects and PlotEffects. PlayerEffects are designed to make permanent changes to players' resource-counts when imposed, so testing them out involves nothing more than triggering them and checking whether their encoded changes do indeed make it through to a player's inventory.

There are five assertions that need to be made about PlayerEffects to prove that they will work as intended. These are as follows...

- Imposing an additive effect with a positive parameter for a particular resource-type will add the value of that parameter to the specified player's count of that specific resource-type
- Imposing an additive effect with a negative parameter for a particular resource-type will subtract the value of that parameter from the specified player's count of that specific resource-type
- Imposing a multiplicative effect with a positive parameter for a particular resource-type will multiply the specified player's count of that specific resource-type by the value of the parameter in question
- Imposing a multiplicative effect with a negative parameter for a particular resource-type will divide the specified player's count of that specific resource-type by the inverse of the parameter in question
- Imposing an additive effect that would ultimately take one or more of the specified player's resource-counts below 0 will instead leave them with 0 of those resources

The simplest way of imposing a PlayerEffect on a player is through the **impose(Player)** method, which accesses the specified player's current resource-counts and adds or multiplies the effect's internal parameters to/by them. Effects can also be imposed by configuring their internal Runnable objects to impose them based on specific preconditions and executing those Runnable objects instead.

PlayerEffect

```
...

public void impose(Player player) {
   if (multiply) {
       player.setResource(ResourceType.ORE, (int) ((float) player.getOre() *
modifiers[0]));
       player.setResource(ResourceType.ENERGY, (int) ((float) player.getEnergy() *
modifiers[1]));
       player.setResource(ResourceType.FOOD, (int) ((float) player.getFood() *
modifiers[2]));
```

```
        player.setMoney((int) ((float) player.getMoney() * modifiers[3]));
    } else {
        player.setResource(ResourceType.ORE, player.getOre() + (int) modifiers[0]);
        player.setResource(ResourceType.ENERGY, player.getEnergy() + (int) modifiers[1]);
        player.setResource(ResourceType.FOOD, player.getFood() + (int) modifiers[2]);
        player.setMoney(player.getMoney() + (int) modifiers[3]);
    }
}

...
```

Hence, the method that we followed to test the assertions stated above is as follows...

- Place a breakpoint at the end of the **impose()** method shown above (to the side of the bracket highlighted in red)
- Temporarily change the local *effectChance* variable in the RoboticonQuest class' **setupEffects()** method to 1 (to ensure that all effects added to the PlayerEffectSource class' internal array are always imposed)

RoboticonQuest

```
...

private void setupEffects() {
  //Initialise the fractional chance of any given effect being applied at the start of a
round
  effectChance = (float) 1;

...
```

- Remove the restriction in the **implementPhase()** method preventing the **setEffects()** method from being run if the local *turnCounter* variable was below 3 (to allow for effects to be imposed as soon as the game starts)

RoboticonQuest

```
...

case 1:
  ...

  if (turnNumber > 0) {
     clearEffects();
     setEffects();
  }
  //Only consider imposing effects once each player has claimed at least 1 tile

...
```

- Change the values of the global *money*, *ore*, *energy* and *food* variables declared in the Player class to 50

Player

```
public class Player {
   ...

   private int money = 50;
   private int ore = 50;
   private int energy = 50;
   private int food = 50;
```

```
    ...
```

- Create four temporary effects in the PlayerEffectSource class as follows…

PlayerEffectSource

```
public PlayerEffect test1;
public PlayerEffect test2;
public PlayerEffect test3;
public PlayerEffect test4;

...

private void configureEffects() {

    ...

    test1 = new PlayerEffect("Test 1", "Test 1", 10, 10, 10, 10, false, new Runnable() {
        @Override
        public void run() {
            test1.impose(game.getPlayer());
        }
    });

    test2 = new PlayerEffect("Test 2", "Test 2", -10, -10, -10, -10, false, new Runnable() {
        @Override
        public void run() {
            test2.impose(game.getPlayer());
        }
    });

    test3 = new PlayerEffect("Test 3", "Test 3", 2, 2, 2, 2, true, new Runnable() {
        @Override
        public void run() {
            test3.impose(game.getPlayer());
        }
    });

    test4 = new PlayerEffect("Test 4", "Test 4", (float) 0.5, (float) 0.5, (float) 0.5,
(float) 0.5, true, new Runnable() {
        @Override
        public void run() {
            test4.impose(game.getPlayer());
        }
    });
}
```

- Change the **implementPhase()** method in the PlayerEffectSource class to make it add the *test1* effect (and nothing else) to the internal array

PlayerEffectSource

```
    ...

    private void implementEffects() {
        add(test1);
    }

    ...
```

- Run the game and check that the following variables bear these values at the breakpoint
    - **Test 1:** [*Effect*: test1] [*Player:* player {money = 60 | ore = 60 | energy = 60 | food = 60}]

- Then, do the same for the other three tests, and check that the same variables appear like so at the breakpoint
    - **Test 2:** [*Effect*: test2] [*Player:* player {[Money = 40 | Ore = 40 | Energy = 40 | Food = 40}]
    - **Test 3:** [*Effect*: test3] [*Player:* player {Money = 100 | Ore = 100 | Energy = 100 | Food = 100}]
    - **Test 4:** [*Effect*: test4] [*Player:* player {Money = 25 | Ore = 25 | Energy = 25 | Food = 25}]
- Now, open the Player class again and change the values of the global *money*, *ore*, *energy* and *food* to 0

Player

```
public class Player {
   ...

   private int money = 0;
   private int ore = 0;
   private int energy = 0;
   private int food = 0;

   ...
```

- Run *test2* again in much the same way that you did before and check that the current player owns the following amounts of resources at the breakpoint
    - **Test 5:** [*Effect*: test2] [*Player:* player {Money = 0 | Ore = 0 | Energy = 0 | Food = 0}]
    (This final test is done to ensure that effects do not withdraw more than what players can afford to give)

Test Results:

Test 1: Pass [*Player:* player {money = 60 | ore = 60 | energy = 60 | food = 60}]
Test 2: Pass [*Player:* player {money = 40 | ore = 40 | energy = 40 | food = 40}]
Test 3: Pass [*Player:* player {money = 100 | ore = 100 | energy = 100 | food = 100}]
Test 4: Pass [*Player:* player {money = 25 | ore = 25 | energy = 25 | food = 25}]
Test 5: Pass [*Player:* player {money = 0 | ore = 0 | energy = 0 | food = 0}]

Plot Effects

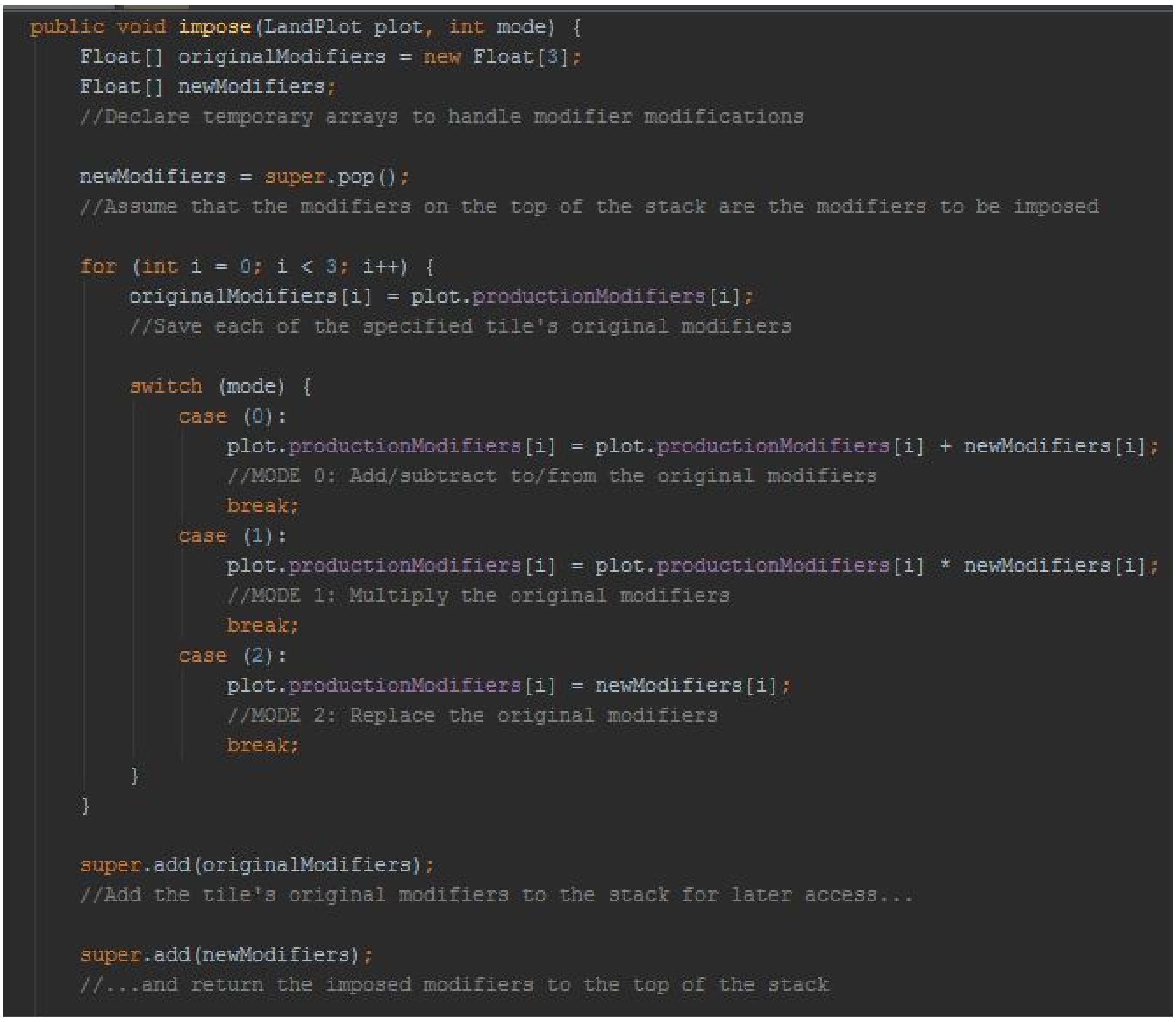
Methodology:

Plot effects work specifically by altering the production modifiers of land plots. Therefore to manually test the effects, we would have to make sure that the correct amount of resources are produced by a land plot based on the effect that has been applied to them.

The following assertion are to be made about plot effects to prove that they work
- For every land plot that an effect is imposed on,:
    - If the plot effect modifier adds/subtracts to the original modifiers, the new modifiers must equal the old modifiers with the effect imposed modifiers added/subtracted to it.
    - If the plot effect modifier multiplies to the original modifiers, the new modifiers must equal the product of old modifiers and the effect imposed modifiers.
    - If the plot effect modifier replaces the original modifiers, the new modifiers must equal the effect imposed modifiers.

    ○ Once an effect is reversed, the tile's modifiers should be the same as they were before the effect was applied

Like player effects, plot effects are invoked via the impose(LandPlot, Mode) function. LandPlot is the plot that is being affected and the Mode is the type of change being applied to the modifiers as described in the comments of the following code.



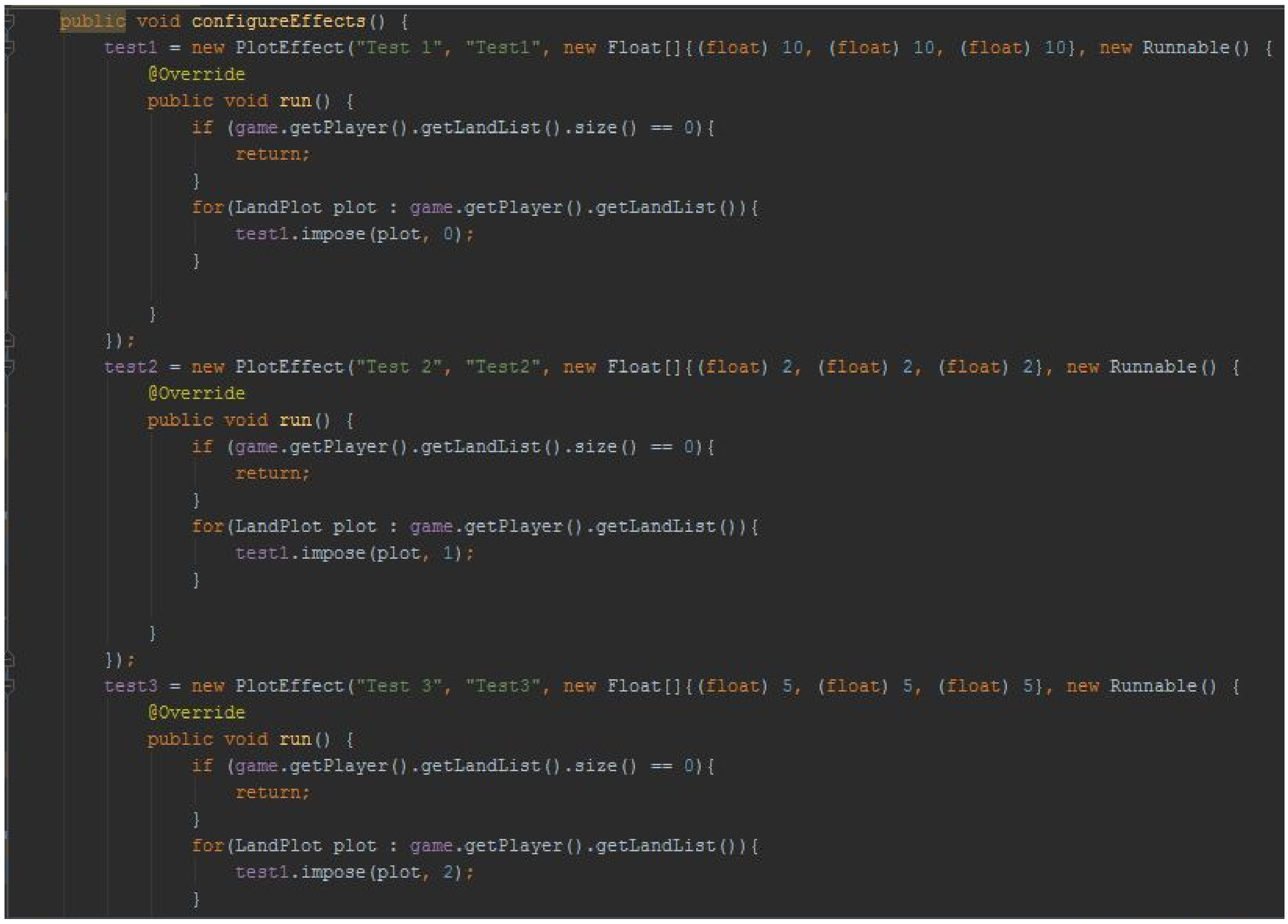
```java
public void impose(LandPlot plot, int mode) {
    Float[] originalModifiers = new Float[3];
    Float[] newModifiers;
    //Declare temporary arrays to handle modifier modifications

    newModifiers = super.pop();
    //Assume that the modifiers on the top of the stack are the modifiers to be imposed

    for (int i = 0; i < 3; i++) {
        originalModifiers[i] = plot.productionModifiers[i];
        //Save each of the specified tile's original modifiers

        switch (mode) {
            case (0):
                plot.productionModifiers[i] = plot.productionModifiers[i] + newModifiers[i];
                //MODE 0: Add/subtract to/from the original modifiers
                break;
            case (1):
                plot.productionModifiers[i] = plot.productionModifiers[i] * newModifiers[i];
                //MODE 1: Multiply the original modifiers
                break;
            case (2):
                plot.productionModifiers[i] = newModifiers[i];
                //MODE 2: Replace the original modifiers
                break;
        }
    }

    super.add(originalModifiers);
    //Add the tile's original modifiers to the stack for later access...

    super.add(newModifiers);
    //...and return the imposed modifiers to the top of the stack
```

Just like the test for the player effects above, the code shall be changed so that effects have a 100% chance of triggering each turn. However, they'll be triggered in the player's second turn. This is because we want to record a tiles resource production before an effect is triggered, so it can be compared to the resource production of a tile after an effect is applied to it. This can be achieved by the following method:

- Acquire a random tile
- Buy a roboticon
- Choose a customisation for the roboticon that produces the resource that the effect affects
- Record the amount of resources that have been produced

- Apply an effect
- Assert that the amount of resources produced is correct
- On the next turn, make sure that the amount of resources produced are the same as they were before the effect was applied to ensure that the effects have been reversed

This will be tested with the following 3 test effects



```
    public void configureEffects() {
        test1 = new PlotEffect("Test 1", "Test1", new Float[]{(float) 10, (float) 10, (float) 10}, new Runnable() {
            @Override
            public void run() {
                if (game.getPlayer().getLandList().size() == 0){
                    return;
                }
                for(LandPlot plot : game.getPlayer().getLandList()){
                    test1.impose(plot, 0);
                }

            }
        });
        test2 = new PlotEffect("Test 2", "Test2", new Float[]{(float) 2, (float) 2, (float) 2}, new Runnable() {
            @Override
            public void run() {
                if (game.getPlayer().getLandList().size() == 0){
                    return;
                }
                for(LandPlot plot : game.getPlayer().getLandList()){
                    test1.impose(plot, 1);
                }

            }
        });
        test3 = new PlotEffect("Test 3", "Test3", new Float[]{(float) 5, (float) 5, (float) 5}, new Runnable() {
            @Override
            public void run() {
                if (game.getPlayer().getLandList().size() == 0){
                    return;
                }
                for(LandPlot plot : game.getPlayer().getLandList()){
                    test1.impose(plot, 2);
                }
```

Results

Test	**Production before effect is applied**	**Production during effect**	**Production after effect has been reverted**
test1	Ore = 2, Energy = 3, Food = 5	Ore = 12, Energy = 13, Food = 15	Ore = 2, Energy = 3, Food = 5
test2	Ore = 7, Energy = 2, Food = 4	Ore = 14, Energy = 4, Food = 8	Ore = 7, Energy = 2, Food = 4
test3	Ore = 3, Energy = 1, Food = 5	Ore = 5, Energy = 5, Food = 5	Ore = 3, Energy = 1, Food = 5

As you can see, all test passed successfully.

Requirements Acceptance Tests

As in our last project, we created a qualitative list of all the requirements and individually filled it in. The evidence of this can be found here. This tests both the requirements being met and is ultimately usability testing as performed in our previous submission. This is a lot less granular of a test as the entire requirement must be met at this point. If any part of a requirement fails here then the system has failed in meeting the bar set for a complete game. To ensure we had finalised any issues Fractal had at the last Submission, we also updated their original document for this part. However any further requirement testing was done in the aforementioned links.

As seen in the document, the game has met all of the requirements. It is worth noting that some of the requirements were modified at the start of this assessment and we have logged the changes in our Change3 document.

GUI Tests

“To ensure that the GUI behaved correctly and as expected, we created a list of test scenarios that describe actions that the player should be able to complete, how the GUI should behave when they are completed, and how it should behave when they are unable to be completed. We have so far only written tests for those features that we have implemented at this stage, as the GUI behaviour for unimplemented features is not yet fully defined. The full plan and results of these tests can be found at http://teamfractal.github.io/assessment2/GUITesting.pdf

All of these tests pass, so we are confident that our GUI, as implemented so far, behaves as expected. We believe this tests are complete because they cover all aspects of the GUI. We believe these tests give us the correct results because we ran them several times as we were developing the code.” -Team Fractal @ Assessment 2

100% Pass rate

DRTN Changes:

Gambling screen

ID	Test	Pass/Fail
29	Input field allows to type in the amount of money for gambling	Pass

30	After hitting “Gamble Money” player’s and AI’s dice values are displayed	Pass
31	After hitting “Gamble Money” player’s money updates according to gambling outcomes	Pass
32	“Money Won” and “Money Lost” fields updates according to gambling outcomes	Pass
33	“W/L” field shows number of win and loss and updates during the gambling	Pass

Effect being imposed

ID	Test	Pass/Fail
34	PlayerEffect occurs. PlayerEffect is clearly visible when imposed. It is clear what has happened both in exposition and game terms.	Pass
35	PlotEffect occurs. PlotEffect is clearly visible when imposed. It is clear what has happened both in exposition and game terms.	Pass

Start screen:

ID	Test	Pass/Fail
1	If the player clicks on the start game button, the game moves to phase 1 of player 1’s turn	Pass

2	If the player clicks on the exit button, the game quits	Pass

Buying a plot:

ID	Test	Pass/Fail
3	If the player clicks on an unbought plot the buy plot button appears	Pass
4	If the player then clicks somewhere else the buy plot button disappears	Pass
5	If the player clicks on the buy plot button and has sufficient gold, the plot will gain a coloured border (blue for player 1, red for player 2)	Pass
6	If the player clicks on a tile that has already been bought, or cannot buy a tile due to already having bought one that turn, or not having enough money, the buy plot button appears greyed out	Pass
7	If the player clicks on the next button, the game moves to the roboticon market screen	Pass

Roboticon market:

ID	Test	Pass/Fail
8	The player can increase and decrease the number of roboticons to buy with the left and right buttons	Pass
9	The player can buy the specified number (provided they have enough money and the market has not run out)	Pass

	buy clicking on the buy roboticons button	
10	When the player clicks on the buy roboticons button, the specified number appears in the list on the right hand side	Pass
11	The player can scroll through their roboticons using the left and right buttons	Pass
12	The player can pick a customisation from the list and can buy it, the customisation then appears on the roboticon in view	Pass
13	If the player clicks on the next button, the game moves to the install roboticon screen	Pass
14	If the player does not click next before they have spent 30 seconds on this screen, the game automatically moves to the install roboticon screen	Pass

Roboticon installation:

ID	Test	Pass/Fail
15	If the player clicks on a tile they own which does not already have a roboticon, the install roboticon menu appears	Pass
16	The player can pick one of their uninstalled roboticons and click to install it on that plot	Pass
17	The player can click cancel on the install roboticon menu to close the menu	Pass

18	When the player installs a roboticon the image of the relevant roboticon appears on the plot tile in question	Pass
19	If the player clicks on the next button, the game moves to the resource production phase	Pass
20	If the player does not click next before they have spent 30 seconds on this screen, the game automatically moves to the resource generation screen	Pass

Resource production:

ID	Test	Pass/Fail
21	On this screen the resource amounts generated appear in the bottom left hand corner	Pass
22	The resource amounts are added to the totals in the top left corner	Pass
23	If the player clicks the next button, the game moves to the resource market screen	Pass
24	If the resource production finishes before the the player clicks next, the game automatically moves to the resource market screen	Pass

Resource Market:

ID	Test	Pass/Fail
25	The player can use the left and right buttons to increase and decrease the	Pass

	amounts of resources to buy or sell	
26	If the player clicks the buy button for a transaction, the relevant amount of gold is and resource is removed/added to the player’s totals in the top left corner	Pass
27	The player cannot buy or sell more resources than the market or they have, or they have money for	Pass
28	If the player clicks the next button the game moves on to the plot buying screen, and switches to the other player	Pass