Exercise 36.48

1. Describe a list of the relevant conditions that influence moveUnit.

Tile
Unit
Position.getColumn()
Position.getRow()

2. Document an equivalence class table that enumerates all your found equivalence classes and briefly argue for the representation property and the set's coverage property.

Condition	Invalid ECs	Valid ECs
Tile	not {P, F, H} [1]	{P} [2], {F} [3], {H} [4]
Unit(to)	playerInTurn [5]	not playerInTurn [6]
Position.getColumn()	< 0 [7]; > 15 [8]	0 - 15 [9]
Position.getRow()	< 0 [10]; > 15 [11]	0 - 15 [12]
Unit(from)	not playerInTurn [13]	playerInTurn[14]
Distance(d)	d = 0 [15], d > 1 [16]	d = 1 [17]

The Tile property, is defined by the ability to move onto a tile, given a type of the specific Tile, which is being moved onto. The only time you're not allowed to move onto a tile because of it's type, is when the type is mountains or ocean. As such, the only invalid case, is when Tile.getTypeString() returns mountains or ocean. This also means, that all other cases are valid and represented as a set of 3 EC's.

The Unit(to), is defined by the owner of the potential unit on destination Tile. The invalid EC for this property is if the player in turn is yourself, as units cannot stack. Valid EC's are, if the owner of the unit is different from yourself, which will lead to the moving unit to attack the enemy unit (which is not handled in this test case scenario). The complete opposite is true for the Unit(from), because only the player should be allowed to move his own units, therefore the invalid EC is now that the owner of the unit is not playerInTurn.

The Position property should be restricted by the worldSize, and therefore it should not be allowed to move to or from position which reside outside of the world. By that logic everything outside of the range 0 - 15 should be invalid ECs. Lastly it should only be allowed to move to a tile next to it.

3. Outline a test case table of concrete test cases defined from the previous analysis, and argue for the heuristics applied to generate them.

For the purpose of these tests, playerInTurn is player.RED

ECs covered	Test case	Expected output
[1], [6], [9], [12], [14], [17]	(M, BLUE, 7, 8, BLUE, 1)	egal
[2], [5], [9], [12], [14], [17]	(P, RED, 7, 8, BLUE, 1)	egal
[2], [6], [7], [12], [14], [17]	(P, BLUE, -1, 8, BLUE, 1)	egal
[2], [6], [8], [12], [14], [17]	(P, BLUE, 16, 8, BLUE, 1)	egal
[2], [6], [9], [10], [14], [17]	(P, BLUE, 7, -1, BLUE, 1)	egal
[2], [6], [9], [11], [14], [17]	(P, BLUE, 7, 16, BLUE, 1)	egal
[2], [6], [9], [12], [13], [17]	(P, BLUE, 7, 8, RED, 1)	egal
[2], [6], [9], [12], [14], [15]	(P, BLUE, 7, 8, BLUE, 0)	egal
[2], [6], [9], [12], [14], [16]	(P, BLUE, 7, 8, BLUE, 2)	egal
[2], [6], [9], [12], [14], [17]	(P, BLUE, 7, 8, BLUE, 1)	egal

We use the heuristics and guidelines adapted from Myers(1979), stating the following:

1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.

2. Until all unvalid ECs have been covered, define a test case whose element only lies in a single invalid EC.

We use this, to generate only one legal test case, as there are 4 valid ECs defined in 4 properties. We define 6 illegal test cases, covering all of the 6 invalid ECs.

4. Present a short comparison to your previously developed TDD test cases from the movevUnit method. Did TDD by itself produce adequate testing?

In TDD we haven't tested the case of moving outside of the worlds boundaries. Outside of this the TDD test cases did cover the moveUnit as well as blackbox testing. We should say, that normally there would probably be a test in TDD for testing if it was possible to move outside of the world boundaries, and that it is not a flaw in the TDD process, but a slip-up made by us.

Exercise 36.50

1. Outline a list of the relevant conditions that influence O.

Unit to

Unit from

Tile to

Tile from

Support to

Support from

2. Document an equivalence class table that enumerates all your found equivalence classes and briefly argue for the representation property and the set's coverage property.

Condition	Invalid ECs	Valid ECs
Unit(from)	not {A, L, S} [1]	{A} [2], {L} [3], {S} [4]
Unit(to)	not {A, L, S} [5]	{A} [6], {L} [7], {S} [8]
Tile(from)	not {C, O, P, F, H, M} [9]	{C} [10], {P} [11], {F} [12], {H} [13], {O} [14], {M} [15]
Tile(to)	not {C, O, P, F, H, M} [16]	{C} [17], {P} [18], {F} [19], {H} [20], {O} [21], {M} [22]
Supcori(from)	s > 7 [23]	s = 0 [24], s = 1 - 7 [25]
Supcori(to)	s > 7 [26]	s = 0 [27], s = 1 - 7 [28]
die(d)		d = 1 - 6 [29]

With respect to coverage it is obvious that the unit and tile equivalent classes cover everything, since they define the allowed and then take the rest as invalid. For the supporting value, we assume that this can't be negative and thus we all the positive numbers, divided into the three sets {s=0}, {1<=s<=7}, and {s>7} and have complete coverage. Finally for the die value, the problem statement tells us that we only have to cover the equivalence class d=1-6.

Representations again for the unit and tile types all of our valid equivalence classes only contain one element and thus will always produce the same result. When we consider the supporting values, we split it up into two intervals, because if we had 0-7 and tested 0 we might not be using the support value at all. However, we do not believe we need to split up the 1-7 interval because if any of these provide the correct result it means that we are using the support strength, and we would assume we do it correctly. Finally, the die value is also representive since we would see errors if they are not in this interval.

3. Document a test case table of concrete test cases defined from the previous alaysis, and argue for the heuristics applied to generate them.

ECs covered	Test case	Expected output
[2][6][10][17][25][28][29]	(A, A, C, C, 1, 1, 2)	FALSE
[3][7][11][17][24][27][29]	(L, L, P, C, 0, 0, 4)	TRUE
[1]	null	illegal
[5]	null	illegal
[9]	null	illegal
[16]	null	illegal
[23]	8	illegal
[26]	8	illegal

Again, we use the heuristics and guidelines adapted from Myers(1979), stating the following:

1. Until all valid ECs have been covered, define a test case that covers as many uncovered valid ECs as possible.

2. Until all unvalid ECs have been covered, define a test case whose element only lies in a single invalid EC.

We use this, to generate two legal test cases - one which returns false, and one which returns true (we only generate two, we know there are lots of other examples of legal test cases though). This time, we also list the valid test cases first, and then list the varying EC, which makes the test case illegal, in all illegal test cases.

4. Make a short comparison to your previously developed TDD test cases from the EpsilonCiv exercise. Did TDD produce adequate testing?

In our previously developed test cases in TestEpsilonCiv we only test if successful attacks increase the number of battles won, and if battles are won/lost correctly. We don't test for all the illegal test cases, mostly because nearly all invalid ECs are null.

Exercise 36.51

The only condition relevant for boundary value analysis is support, as unit and tile can only be a specific set of values, and a die cannot be outside the range of 1-6 (this is a precondition).

Support, EC [23]-[28], has the boundaries 0 and 7. We already test the cases 0 and 8 so the question is, if it would make any sense to test the boundaries -1 and 7.

If we look closer at the implementation of how the support is calculated, we see an iteration over the nearby friendly units. An iteration over this, would generally indicate, that boundary value analysis would be a good idea, as we could run into the "off-by-one" error (which happens during iterations if ever, fx. if you try to get an object from an array at a specific index, which is out of range). As such, we have the opinion that it would make sense to make a boundary value analysis in this case.

We would augment the test case table, by inserting test cases, which cover the boundary 7.

XML-Builder

Make an XML builder that builds an XML representation of the document defined by (fake object) class WordProcessor, that uses an XML storage format as examplified on page 297 in FRS.

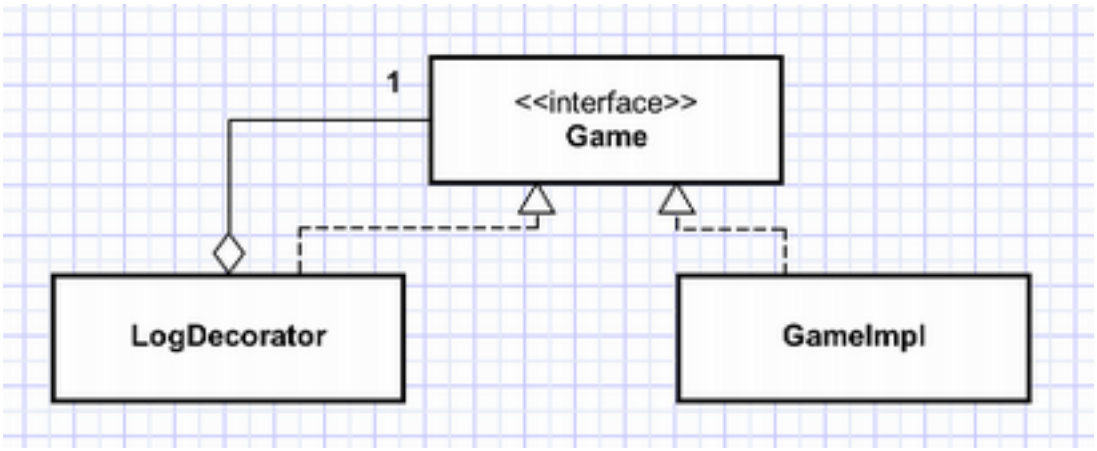
Exercise 36.30

1. Find a suitable design pattern to implement this requirement so any HotCiv variant can be transcribed. Your implementation must be purely change by addition, not by modification.

Inspired by the logging mechanism introduced to the Pay Station, we chose the decorator pattern.

2. Describe the pattern you have chosen and outline why it is appropriate.

The decorator pattern allows us to add responsibilities and behavior to individual objects without modifying its class. This is exactly what we want in our scenario, we want to be able to add the behavior of logging to our HotCiv game. The decorator pattern tell us to create a Decorator class that implements the same interface as our component, in HotCiv this is the MutableGame interface. The Decorator class is then given a field of the component type to which it will delegate all requests it receives. Then, ConcreteDecorators are made which extends the Decorator, the ConcreteDecorators responsibility is to provide some added behavior and then call the ask our Delegator class to request the response from the game.



4. Explain how you can turn transcribing on and off during the game.

We can turn it off on runtime by implementing a toggle function, which listens to the user toggling logging on and off. The toggling executes the following code:

```
if (game == decoratee) {
      game = new LogDecorator(game);
} else {
      game = decoratee;
}
```

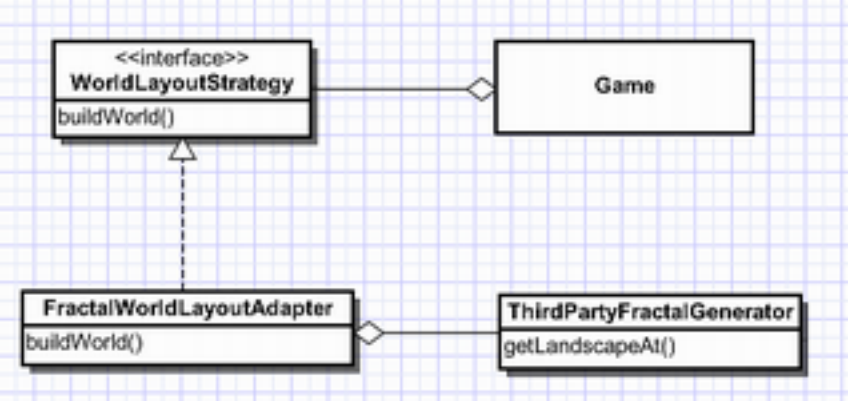
We have two pointers to the same game implementation when the object is first instantiated. We can then change one of the pointers - the pointer executing the code (game), to enable or disable logging.

Exersice 36.31

1. Find and describe a suitable design pattern that will allow you to integrate it into HotCiv for generating fractal maps. Your implementation must be purely change by addition, not by modification. Specifically you are not supposed to modify the developed production code for DeltaCiv.

The implementation of the ThridPartyFractalGenerator is difficult because our WorldLayoutStrategy has a very different interface. More generally we have a problem of incompatibility between two classes interface and protocol. This is exactly the problem that the Adapter pattern can help us solve.
The adapter pattern tells us to make a new class with the adapter role. This adapter class will implement the interface which the client is requesting something from. In our HotCiv implementation, the GameImpl class is the client and it is requesting the WorldLayoutStrategy to setup the world. Thus, we must make the adapter class implement the WorldLayoutStrategy. The adapter class also needs an instance of the class that it is using to provide the behavior that the target interface demands, in our case this is the ThridPartyFractalGenerator. The adapter class will then, whenever requested through the interface, translate the information that the ThridPartyFractalGenerator provides to the format expected by the WorldLayoutStrategy.

UML-Diagram of the implementation of the Adapter Pattern:


<<interface>>
WorldLayoutStrategy
buildWorld()
Game
FractalWorldLayoutAdapter
buildWorld()
ThirdPartyFractalGenerator
getLandscapeAt()