dSoftArk Afl. 1

The final test list.

Player tests

- Red is the first player in turn
- Player Turn shifts between Red and Blue

World Layout Tests

- There is Ocean at (1,0)
- There are Hills at (0,1)
- There are Mountains at (2,2)
- There are Plains everywhere else fx at (0,0)

Unit Tests

- Red has archer and settler at start
- Blue has legion at start
- Units cannot move over mountains
- Red Cannot move Blue’s Units and vice versa
- There can only be one Unit on a Tile

Attack Tests

- Attacker always wins

Unit Action Tests

- As far as I know it is not possible to assert getting no output whatsoever.

Cities

- Red’s city is at (1,1)
- Blue’s city is at (4,1)

- Cities’ population size is always 1
- Cities produce 6 ‘production’ after a round has ended

Unit Production

- Produce a unit selected for production and deduct unit’s cost from the city’s treasury of production
- The unit is placed on the city til ei fno other unit is present, otherwise it is placed on the first non-occupied adjacent tile, starting from the tile just north of the city and moving clockwise.

Aging

- Game starts at 4000 BC
- Each round advances the game age 100 years

Winning

- Noone wins before the year 3000 BC
- Red wins in year 3000 BC

Outline of TDD iterations in detail.

public void shiftsPlayerOnNextTurn()

We start by writing our test-method, named as above:

```
@Test
  public void shiftsPlayerOnNextTurn(){
        Player p = game.getPlayerInTurn();
        assertEquals("RED should be in turn if the game has just 
started", Player.RED, p);
        game.endOfTurn();
        p = game.getPlayerInTurn();
        assertEquals("BLUE should be in turn 2", Player.BLUE, p);
        game.endOfTurn();
        p = game.getPlayerInTurn();
        assertEquals("RED should be in turn 3", Player.RED, p);
  }
```

As expected, this assertion gives a red-bar. After this, we apply the principle “fake it till you make it”. First we simply make game.endOfTurn get the next object in our array of users as such:

```
playerInTurn = playerIterator.next();
```

This gives us another red flag, saying *“expected Player.RED but got Player.BLUE”*. After discovering this, we tried another time, implementing a conditional statement to our code:

```
 if(playerIterator.hasNext()){
      playerInTurn = playerIterator.next();
}else{
      //End of Round
      playerIterator = playerList.iterator();
      playerInTurn = playerIterator.next();
}
```

After this, it worked.

Observed benefits and liabilities of the TDD approach.

Benefits of the TDD approach, in comparison to a rather normal approach to programming is, that the code you write, will give very visible output displaying if it works or not.
It’s always easy to add another test, and then afterwards add the code, making the test work in that particular case, then adding code to make it work generically afterwards, trying to prove another test true. This makes the scalability of the code, using the approach good.

A liability is, that in comparison to just writing the perfect code the first time, it takes a lot of time. If the program you’re writing is very small, the amount of time it takes to write a test in comparison to the time it takes to implement the actual code will be large. When you try to upscale the program though, one might feel sorry for oneself, having not done those initial tests, if the code later on begins to go bad.

Backlog

We did not make the last test in City Production which revolves the placing of new units in/around the City Tile.