1. The design patterns used and argumentation why they are used.

For the implementation of EpsilonCiv, we extend WinnerStrategy with yet another implementation - EpsilionWinnerStrategy, giving EpsilonCiv a strategy-pattern based approach to solving the problem of the new strategy for winning the game (fig.1).
For the implementation of ZetaCiv, we take a state-pattern approach, extending WinnerStrategy with a new implementation - ZetaWinnerStrategy (fig. 2), which then decides which WinnerStrategy to use, depending of the inner state of the game, which changes on runtime (after 20 rounds).
Furthermore, we found a variability point in attacking, where, depending on the version HotCiv, the algorithm for deciding if an attack is won or not, varies. As a result of this, we decided to follow the 3-1-2 approach, and encapsulated this behaviour in an interface, and then implemented different versions of these (also shown in fig. 2)

2. UML class diagrams for resulting designs. I advice making one for each exercise instead of one big diagram for the final system as the latter gets pretty confusing to read.

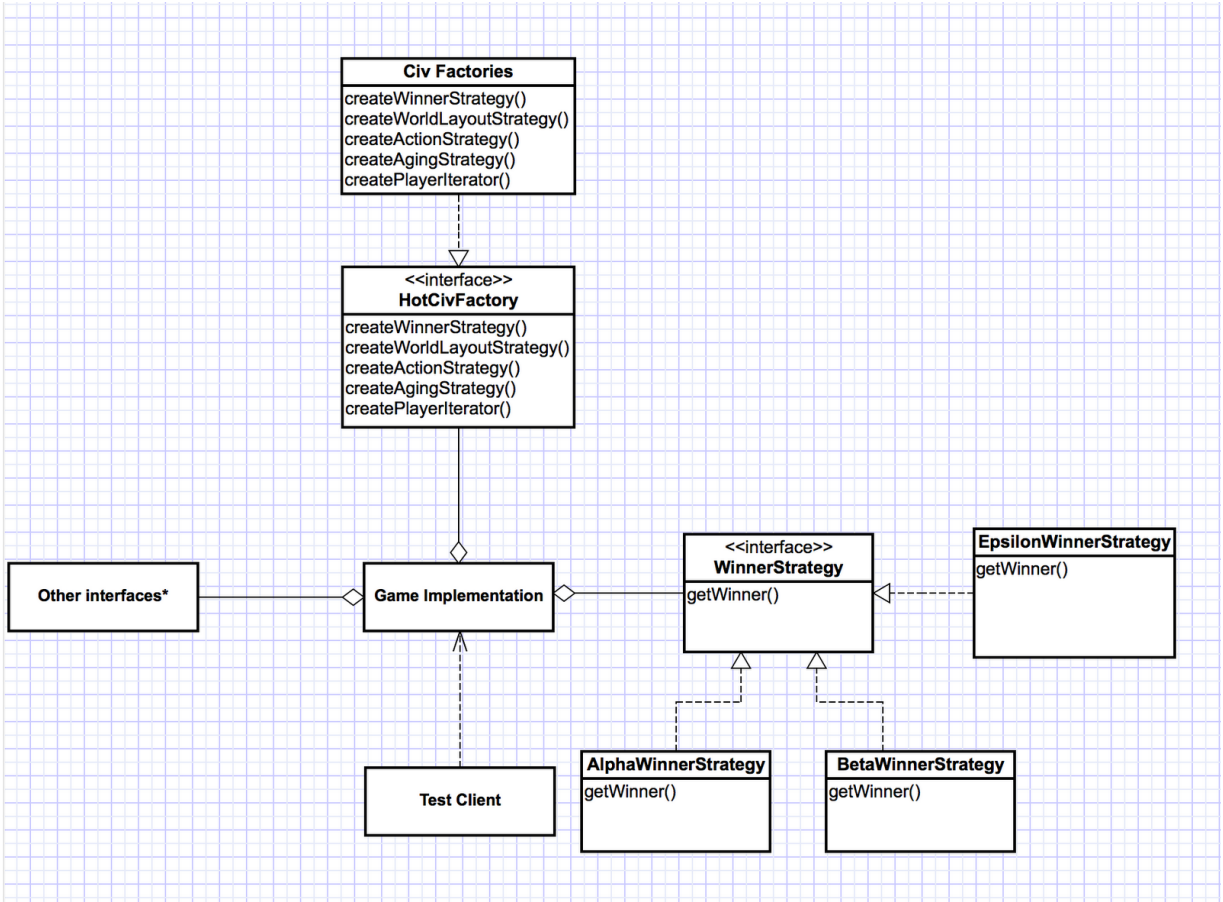

Fig 1. Final EpsilonCiv UML-Diagram

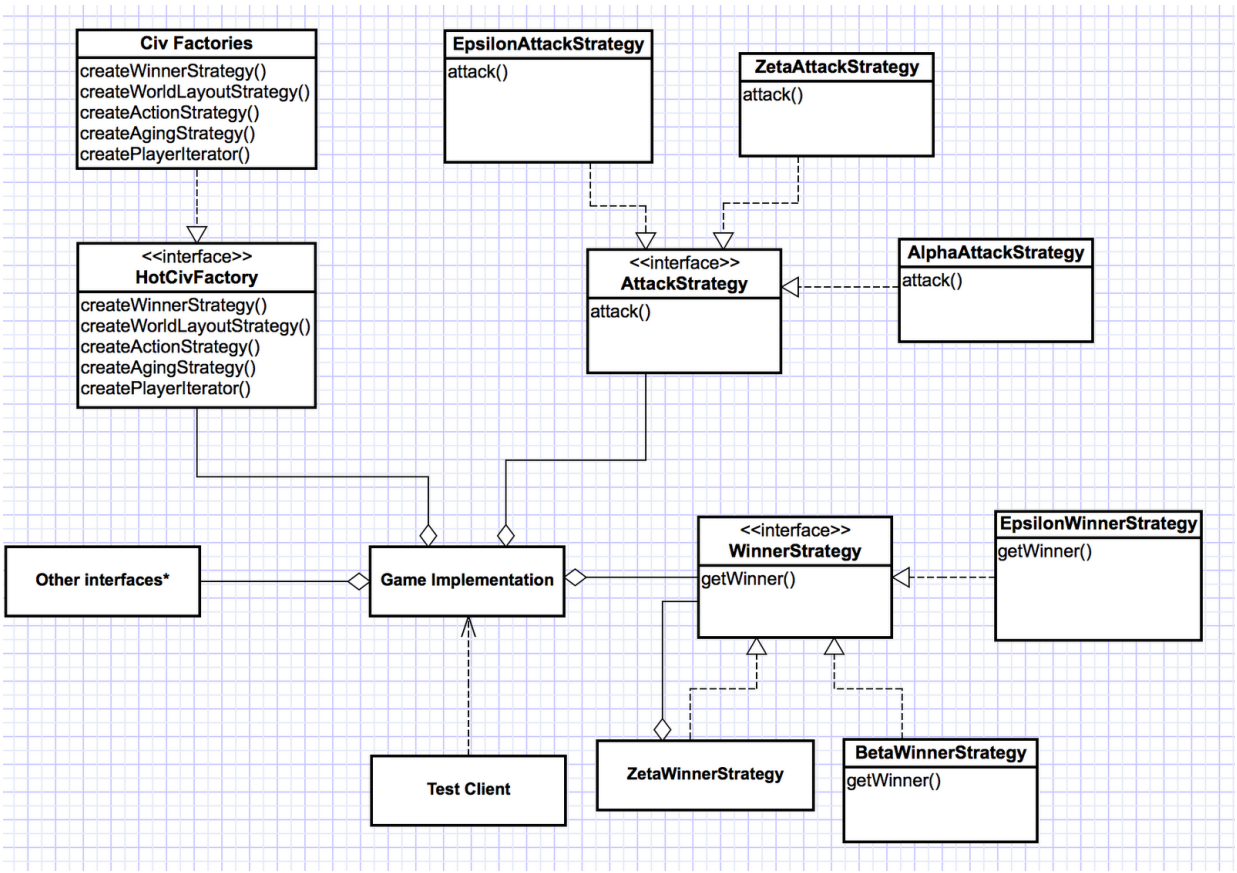

Fig 2. Final ZetaCiv UML-Diagram

3. A discussion of the use of test stubs.

We have used test stubs in order to test code that would not give the correct results 100% of the time. For example to simulate our attacks in Epsilon- and ZetaCiv. In the code stub below we can see that Epsilon- and ZetaCiv use an attackstrategy that provides us with a more-or-less random winner of the battle:

```
Random rng = new Random();

int aStrength = a.getAttackingStrength() + Utility.getFriendlySupport(game, attacker,
a.getOwner()) + Utility.getTerrainFactor(game, attacker);
int dStrength = d.getDefensiveStrength() + Utility.getFriendlySupport(game, defender,
a.getOwner()) + Utility.getTerrainFactor(game, defender);

//Calculate the winner, false if defender wins, true if attacker does.
if (aStrength * (rng.nextInt(6) + 1) > dStrength * (rng.nextInt(6) + 1)) {
        System.out.println("OWNER: " + a.getOwner());
        game.incrementBattlesWon(a.getOwner());
        return true;
}

return false;
```

This code is of course the correct implementation, as it would be a pretty onesided game if the same player always won all of the skirmishes, but does not give reliable testresults because of the fact that we

multiply the attackStrength and defensiveStrength, respectively, with a random number between 1 and 6. Because of this we use a test stub to control the attackStrength and defensiveStrength of the units in battle:

```
Random rng = new Random();

int aStrength = 10;
int dStrength = 1;

//Calculate the winner, false if defender wins, true if attacker does.
if (aStrength * (rng.nextInt(6) + 1) > dStrength * (rng.nextInt(6) + 1)) {
        game.incrementBattlesWon(a.getOwner());
        return true;
}

return false;
```

So to sum up; the use of test stubs in these kinds of situations allows us to simulate results that would not otherwise be reliable/testable.

4. A short reflection on the TDD and refactoring process.

Most of the refactoring we've done, was when implementing the abstract factories. This resulted in having to completely rewrite the constructor of GameImpl, and thus having to also refactor whereever we've instantiated GameImpl, as the preceding parameterized solution required six different parameters, and the abstract factory solution only requires one, leaving the work for the abstract factories.

As a result, we now have a much cleaner and easier construction of Game:

```
public GameImpl(HotCivFactory factory)
```

vs.

```
public GameImpl(Player player1, Player player2, Winnerstrategy ws, Agingstrategy as,
…...)
```

Furthermore, we added attacking to moveUnit(Position from, Position to) and implemented attack(Position attacker, Position defender) as follows:

```
public boolean moveUnit( Position from, Position to ) {

        ...
                //If enemy unit on tile.
                if (this.getUnitAt(to) != null &&
                this.getUnitAt(to).getOwner().equals(playerInTurn)){
                        if(this.attack(from, to) == false){
                                this.deleteUnitAtPosition(from);
                                return false;
                        }
                }

        ...
                return true;
        ...
}



@Override
public boolean attack(Position attacker, Position defender) {
```

```
            return attackStrategy.attack(this, attacker, defender);
        }
```

Exercise 36.18:

> *"Use the definition and properties of test stubs to verify if this statement is true."*

The definition of a test stub is: "*A test stub is a replacement of a real depended-on unit that feeds indirect input, defined by the test code, into the unit under test*."

Following this definition the world layout of AlphaCiv is not a test stub, as this particular world layout is not a replacement of the real *depend-on unit* (DOU), but in fact is the DOU itself.