1. *A short outline of the TDD refactoring iteration that refactors your AlphaCiv variant into a design that will support the BetaCiv requirement (not the actual BetaCiv feature adding iteration!)*

First off, we had to change the code in the GameImpl class, to support the new strategy pattern approach. We do this by adding parameters to the constructor of GameImpl, and setting strategy variables, for dynamic use in the code, which handles the different types of the game - AlphaCiv and BetaCiv. Below we have the constructor code for GameImpl:

```
private AgingStrategy agingStrategy;
private WinnerStrategy winnerStrategy;

public GameImpl(Player p1, Player p2, AgingStrategy as, WinnerStrategy ws){


        playerList.add(p1);
        playerList.add(p2);

        this.agingStrategy = as;
        this.winnerStrategy = ws;
        ...
}
```

The strategies are then used as follows:

AgingStrategy:

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

                age = agingStrategy.doAging(this.age);
                ...
}
```

The job of aging in the game is now delegated to the strategy responsible for the type of game, which is played.

WinnerStrategy:

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

                age = agingStrategy.doAging(this.age);

                winner = winnerStrategy.getWinner(this);

                ...
}

public boolean moveUnit( Position from, Position to ) {

                //if unit there is a unit present at Position from.
                ...
                        //Conquer city if city is there.
                        City c = cityArray[to.getRow()][to.getColumn()];
                        if(c != null){
                                c.setOwner(playerInTurn);
                                winner = this.winnerStrategy.getWinner(this);
                        }

                        return true;
                }
        ...
        return false;
}
```

Winning the game is handled when you move (moveUnit()), if you conquer all cities, you win. It's also handled in at the end of a turn (endOfTurn()), which seems like a more natural approach for AlphaCiv, as this is completely dependant on the age of the game.

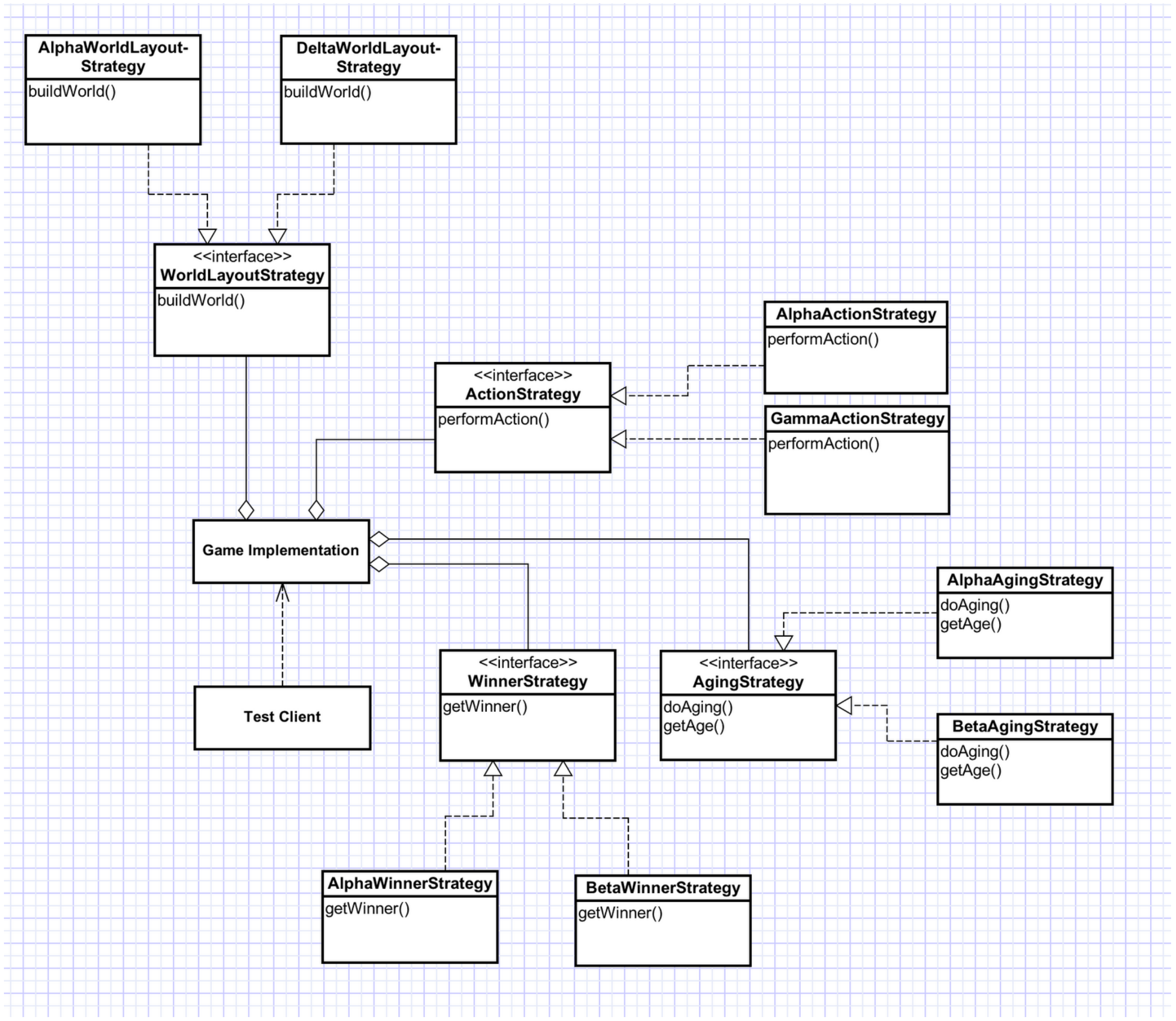
2. *A short outline of the variability points you have identified for Beta-, Gamma-, and DeltaCiv. For one of them, discuss the use of the 3-1-2 process and the Strategy pattern.*

The variability points that we have identified for BetaCiv are the games way of finding a winner and its way of handling the aging that takes place inside the game. The difference here is BetaCivs non-linear aging strategy and meaningful model of winning compared to AlphaCiv.

The variability point for GammaCiv are is the `performAction()` method that makes units in the game do something when called. In AlphaCiv the `performAction()` method specifically had to do nothing, whereas the requirement of GammaCiv was that Settlers and Archers should have actions associated with them.

The variability points for DeltaCiv is its way of generating the layout of the world. AlphaCiv had a static world layout whereas the requirement of DeltaCiv is that the layout of the world has to as showed in the book per default but also be customizable to the programmer outside of the production code. After having identified this variability point we expressed this as an interface through the WorldLayoutStrategy interface with a method `buildWorld()` which sets the layout of the world. We then used the Strategy pattern to make the behaviour of the `buildWorld()` method variable depending on the WorldLayoutStrategy used.

3. *An UML class diagram outlining the final compositional design including all variants.*



4. *The answers to the questions outlined in exercise 36.11.*
    1. *Count the number of relations between instances of Unit and other abstractions in your design for the two solutions.*

In the strategy-based approach, the unit instance only has one relation to GameImpl, as the unit type is handled by a variable in the Unit implementation. In the approach proposed by 36.11, Unit has 2 different implementations, giving the unit instance 2 different relations to the GameImpl class; ArcherUnit and SettlerUnit.

    2. *How and in which abstractions is the "destroy settler unit" responsibility handled in the two solutions?*

In the case of the present strategy-based approach, the strategy pattern itself, handles the calling of
the method, which destroys the settler. In the approach given in 36.11, the Game handles the destruction of the settler.

    3. *How and in which abstractions is the "create city at position" responsibility handled in the two solutions?*

In the case of the present strategy-based approach, the strategy pattern itself, handles the calling of the method, which destroys the settler. In the approach given in 36.11, the Game handles the destruction of the settler.