Exercise 36.22

1. Produce a configuration table outlining all variants and their variability points.

	Variability points				
Product	**Aging**	**Winner**	**WorldLayout**	**Action**	**Attack**
AlphaCiv	Alpha	Alpha	Alpha	Alpha	Alpha
BetaCiv	Beta	Beta	Alpha	Alpha	Alpha
GammaCiv	Alpha	Alpha	Alpha	Gamma	Alpha
DeltaCiv	Alpha	Alpha	Delta	Alpha	Alpha
EpsilonCiv	Alpha	Epsilon	Alpha	Alpha	Epsilon
ZetaCiv	Alpha	Zeta	Alpha	Alpha	Zeta
SemiCiv	Beta	Epsilon	Delta	Gamma	Epsilon

2. Analyze the amount of modifications in the HotCiv production code that are necessary to configure the SemiCiv variant. Analyze if any design changes are necessary.

No changes to the production code are necessary. As the compositional design pattern is followed throughout the code is already made dynamically to be able to use all different strategies implemented.

Exercise 36.23

1. Analyze this solution with respect to the amount and type of parameters to control variability.

In a parametric solution, you would use an enumerator stating which version of the game is being implemented.

The construction of the game implementation would look something like this:

```
public enum GameVersion{
      ALPHA, BETA, GAMMA, DELTA, EPSILON, ZETA, SEMI;
}

private GameVersion gameVersion;

public GameImpl(GameVersion gameVersion){
      this.gameVersion = gameVersion;
}
```

The code initially, when constructing a new instance of the implementation, is easy to understand. You simply state which version of the game you want to implement, meaning you only have one parameter to control initially.

2. Sketch the code of a Game method with emphasis on the variant handling code.

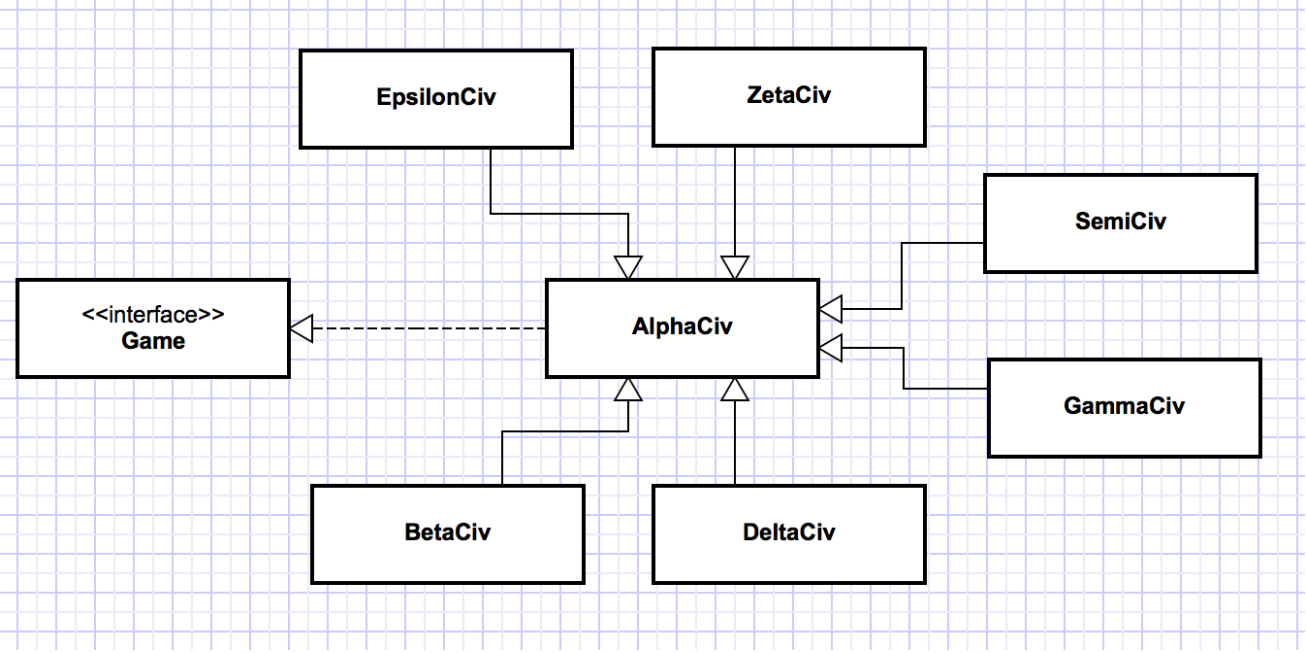
Check Annex 1 to view the code of endTurn() implemented using a parametric solution

As easily seen, the code gets confusing really quickly, and this is only one method - meaning if we had multiple of these, our entire game implementation would get impossible to navigate in.

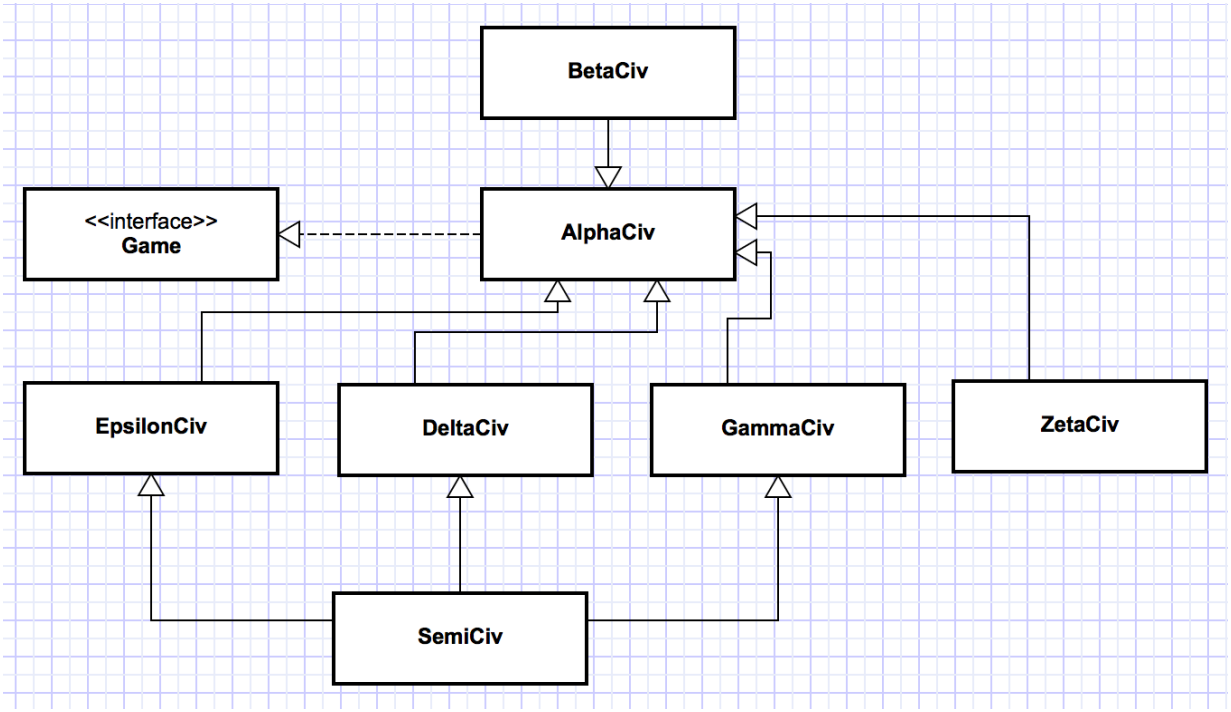
Apart from the above the game itself is also given a lot of responsibility that logically should not be the games'.

Exercise 36.24

1. Sketch two different design proposals for how to implement SemiCiv based upon a purely polymorphic design. You may consider a design in a language that supports multiple implementation inheritance, like C++.



Polymorphic design proposal 1.



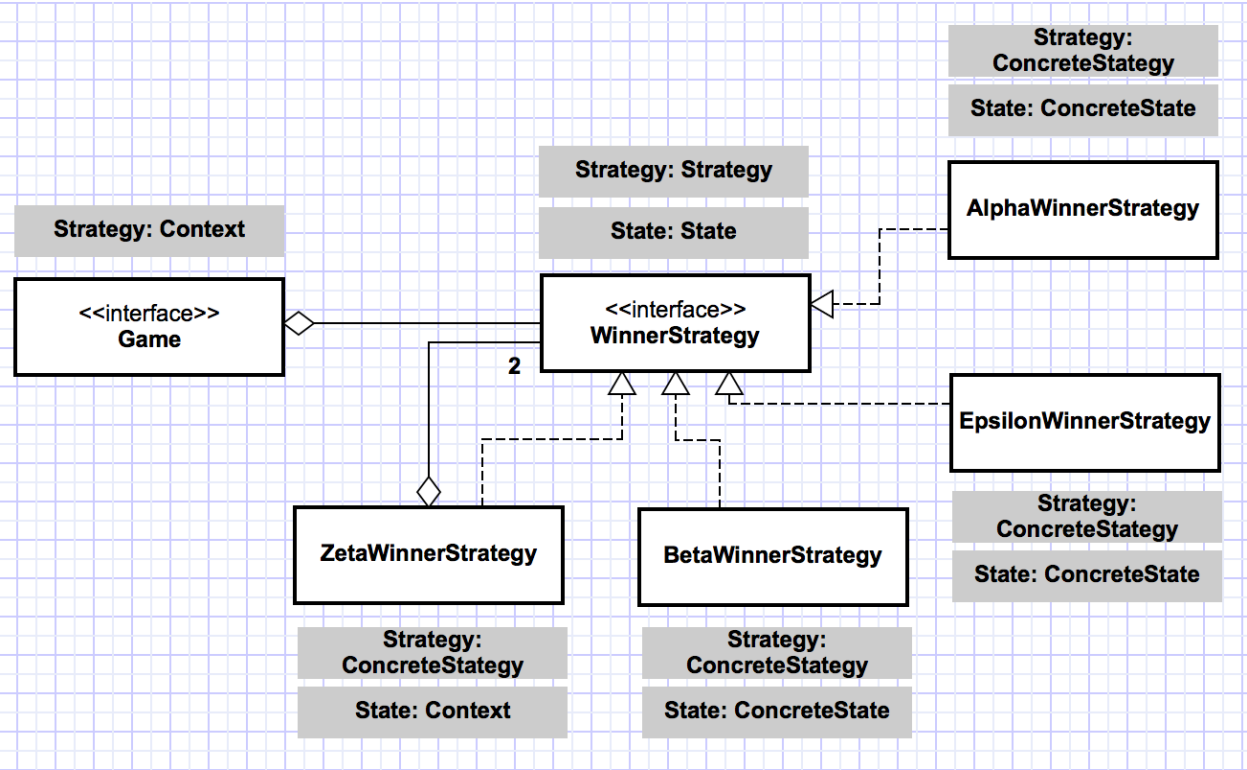
Polymorphic design proposal 2 (using multiple implementation inheritance).

2. Discuss benefits and liabilities of the polymorphic design in comparison with the compositional design.

Benefits of the polymorphic design solution is that only have one single code base, and when you create a new game, you don't have to include any parameters, where as a compositional solution depends on a factory/one or several parameters to define the different strategies to use.

The downsides of the design though are: it's very difficult to reuse algorithms defined by one implementation in another, as different implementations end up overriding each others methods and such. Another liability to the polymorphic solution is compile time binding; meaning that you can't change how the f.x. attack algorithm behaves without resolving to a "sort of" parametric-style solution.

Exercise 36.25



Role diagram of Winner strategy pattern.

Exercise 36.26

1. Find examples of abstractions, objects, and interactions in your HotCiv system that are covered by each of the concepts above.

Behaviour:

Behaviour describes how an object behaves. In our system we can take Unit as an example. Unit f.x. has the getAttackingStrength() method returning the attackingStrength of the Unit-object in question as seen below:

```
public class UnitImpl implements Unit{
	...
	private int attackingStrength;
	...
	@Override
	public int getAttackingStrength() {
		return attackingStrength;
	}
}
```

Responsibility:

Responsibility describes the state of being accountable and dependable to answer a requst. An example of could be WinnerStrategy. WinnerStrategy is responsible for returning the winner, when asked for it. See below:

```
public interface WinnerStrategy {
	public Player getWinner(Game game);
}
```

Role:

A role describes the function an object has in a process.

```
if (this.getUnitAt(to) != null &&
this.getUnitAt(to).getOwner().equals(playerInTurn)){
	if(this.attackStrategy.attack(this, from, to) == false){
		this.deleteUnitAtPosition(from);
		return false;
	}
}
```

Here, Unit takes on the role of being the one, who gives the process a player in order to decide which of the paths of the algorithm, the program should follow. City, f.eks. has a method called getOwner() as well, meaning that city can take on the same role, if it is expected to in the process.

Protocol:

Protocol describes the sequence(s) of interactions or actions expected by a set of roles. In our system this comes to show in all of our strategies f.x. where the game asks for the winner of an attack, and the attackStrategy reacts, calculating the answer and responding back to the game. See below:

Game asking to find the winner of an attack:

```
this.attackStrategy.attack(this, from, to)
```

The attackStrategy responding with the winner of the attack:

```
public boolean attack(Game game, Position attacker, Position defender) {
	Unit a = game.getUnitAt(attacker);
	Unit d = game.getUnitAt(defender);

	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);
```

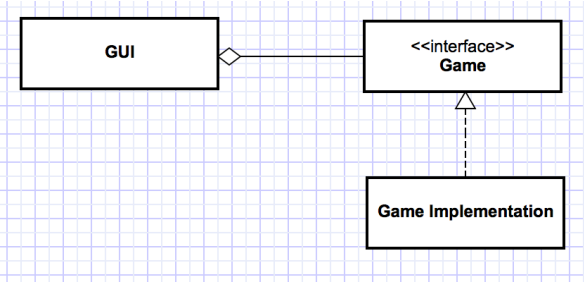
```
        //Calculate the winner, false if defender wins, true if attacker does.
        if (aStrength*(rng.nextInt(6) + 1) > dStrength*(rng.nextInt(6) + 1)) {
              game.incrementBattlesWon(a.getOwner());
              return true;
        }

        return false;
}
```

Exercise 36.29

1. Identify the design pattern that the Game interface represents as seen from the perspective of a graphical user interface. Argue for benefits and liabilities of this design.

We've drawn the design pattern as follows:



The design pattern that we've identified is a Facade-pattern. The intent of the facade pattern is, to provide an interface to a subsystem. The subsystem should only be accessable from the higher level interface; the client in the facade pattern.

Only being able to access the subsystem from the interface, in our case the GUI of the game is beneficial to the system, as this provides security for the subsystem in the way that the user can only access the subsystem in the ways the programmer wants it to. A liability is, that the facade patterns client bloats easily, trying to execute all of the functionality in the subsystem, in this case the Game.

Annex 1.

```
public void endOfTurn() {

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

		round++;

		if (this.gameVersion == GameVersion.ALPHA) {
			age += 100;

			if (game.getAge() == -3000){
				winner = Player.RED;
			}

		}else if(this.gameVersion == GameVersion.BETA){
			int gameAge = 0;

			if (age < -100) {
				gameAge = age + 100;
			} else if(age >= -100 && age < 50) {
				switch(age) {
				case -100:
					gameAge = -1;
					break;
				case -1:
					gameAge = 1;
					break;
				case 1:
					gameAge = 50;
					break;
				}
			} else if (age >= 50 && age < 1750) {
				gameAge = age + 50;
			} else if (age >= 1750 && age < 1900) {
				gameAge = age + 25;
			} else if (age >= 1900 && age < 1970) {
				gameAge = age + 5;
			} else if (age >= 1970) {
				gameAge = age + 1;
			}

			age = gameAge;

			boolean win = true;

			for(City[] cArray : cityArray){
				for(City city : cArray){
					if(city != null){
```

```
                        if(!city.getOwner().equals(playerInTurn)){
                              win = false;
                        }
                  }
            }
      }

}else if(this.gameVersion == GameVersion.DELTA){
      age += 100;

      if (game.getAge() == -3000){
            winner = Player.RED;
      }

}else if(this.gameVersion == GameVersion.EPSILON){
      age += 100;

      Iterator i = battlesWon.entrySet().iterator();

      while (i.hasNext()) {
            Map.Entry me = (Map.Entry)i.next();
            if ((Integer)me.getValue() == 3) {
                  winner = (Player)me.getKey();
            };
      }

}else if(this.gameVersion == GameVersion.GAMMA){
      age += 100;

      if (game.getAge() == -3000){
            winner = Player.RED;
      }

}else if(this.gameVersion == GameVersion.SEMI){
      int gameAge = 0;

      if (age < -100) {
            gameAge = age + 100;
      } else if(age >= -100 && age < 50) {
            switch(age) {
            case -100:
                  gameAge = -1;
                  break;
            case -1:
                  gameAge = 1;
                  break;
            case 1:
                  gameAge = 50;
                  break;
            }
      } else if (age >= 50 && age < 1750) {
            gameAge = age + 50;
      } else if (age >= 1750 && age < 1900) {
            gameAge = age + 25;
      } else if (age >= 1900 && age < 1970) {
            gameAge = age + 5;
```

```
        } else if (age >= 1970) {
              gameAge = age + 1;
        }

        age = gameAge;

        Iterator i = battlesWon.entrySet().iterator();

        while (i.hasNext()) {
              Map.Entry me = (Map.Entry)i.next();
              if ((Integer)me.getValue() == 3) {
                    winner = (Player)me.getKey();
              };
        }

  }else if(this.gameVersion == GameVersion.ZETA){
        age += 100;

        if (round > 20) {
              Iterator i = battlesWon.entrySet().iterator();

              while (i.hasNext()) {
                    Map.Entry me = (Map.Entry)i.next();
                    if ((Integer)me.getValue() == 3) {
                          winner = (Player)me.getKey();
                    };
              }
        }else{
              boolean win = true;

              for(City[] cArray : cityArray){
                    for(City city : cArray){
                          if(city != null){
                                if(!city.getOwner().
                                equals(playerInTurn)){
                                      win = false;
                                }
                          }
                    }
              }

              if(win){
                    winner = playerInTurn;
              }
        }

        for (int i = 0; i < cityArray.length; i++) {
              for (int j = 0; j < cityArray[i].length; j++) {
                    if (cityArray[i][j] != null){
                          City c = cityArray[i][j];
                          c.setMoney(c.getMoney() + 6);
                    }
              }
        }

  }
```

```
        }
}
```