36.22

Varibility Points					
Varianter	World Aging	Unit Actions	World Layout	Winner	Attacking
AlphaCiv	AlphaCiv	AlphaCiv	AlphaCiv	AlphaCiv	AlphaCiv
BetaCiv	BetaCiv	AlphaCiv	AlphaCiv	BetaCiv	AlphaCiv
GammaCiv	AlphaCiv	GammaCiv	AlphaCiv	AlphaCiv	AlphaCiv
DeltaCiv	AlphaCiv	AlphaCiv	DeltaCiv	AlphaCiv	AlphaCiv
ZetaCiv	AlphaCiv	AlphaCiv	AlphaCiv	Betaciv til og med turn 20 derefter EpsilonCiv	AlphaCiv
SemiCiv	BetaCiv	GammaCiv	DeltaCiv	EpsilonCiv	EpsilonCiv

Der var ikke brug for at ændre noget i den kode vi allerede havde for at kunne implementere SemiCiv. Det blev lavet et nyt factory og en test klasse til SemiCiv som det eneste nye.

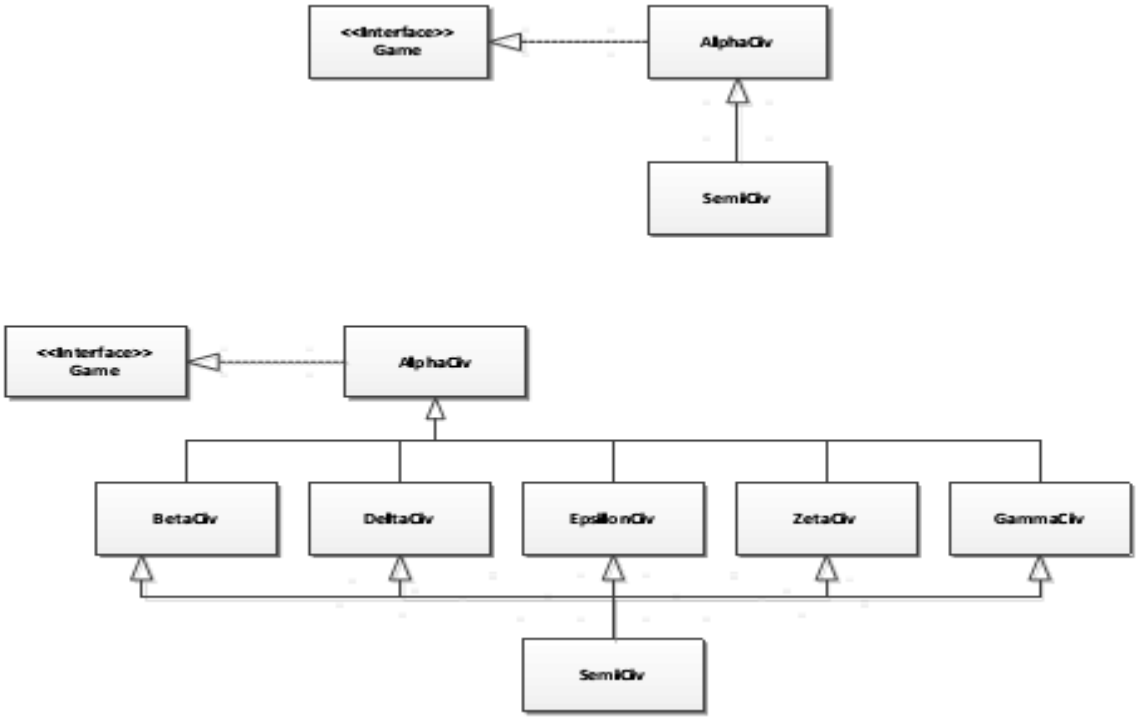
36.23

Fordele og ulemper ved det parametriske design til de forskellige Civ-versioner. En af fordelene er, at et parametriske design bygger på if-conditionals som alle kender til.
En anden fordel er at der kun er en del af koden som skal ordnes frem for at skulle vedligeholde flere forskellige varianter.
Af ulemper nævnes fra bogen :Responsibility Erosion og Reliability concerns.
Responsibility går ud på at en klasse får mere Responsibility eller ansvar hvilket ikke var meningen. Det sker fordi på grund af nye ting bliver tilføjet/ændret i koden
Reliability er at når ny ting skal tilføjes i koden så kan der være fejl i den nye kode.
Vores eksempler:
Age:Der er to varianter af algoritmen til game-age, så her kunne bruge en boolean, men også med fordel en int da der i fremtiden muligvis kunne komme tilføjelser.
Map:Med de nuværende varianter bruges der 2 typer map, men da der sikkert kan komme flere ville vi nok bruge en int til at adskille typerne.
Units:Der er tre typer units men på kun mulighed en eller ingen action så en boolean til hver, igen kunne int bruges til fremtidige udvidelser.
Combat: int eller boolean, samme grund: fremtidige varianter
Movement: int eller boolean.
CivWin:pt. 3 varianter, derfor ville en int være bedst her.

```
private Integer getEndOfGameAge()
{
   Integer age = null;
   if (ageing)
 {
     return (-3000) // fixed
  }
   else {
    if (age < -100)
   {
   age = age + 100;
   // setting age in a progressive way
   // many lines
  } else if (age >= 1970)
   age = age + 1;
   return age;
   }
}
```

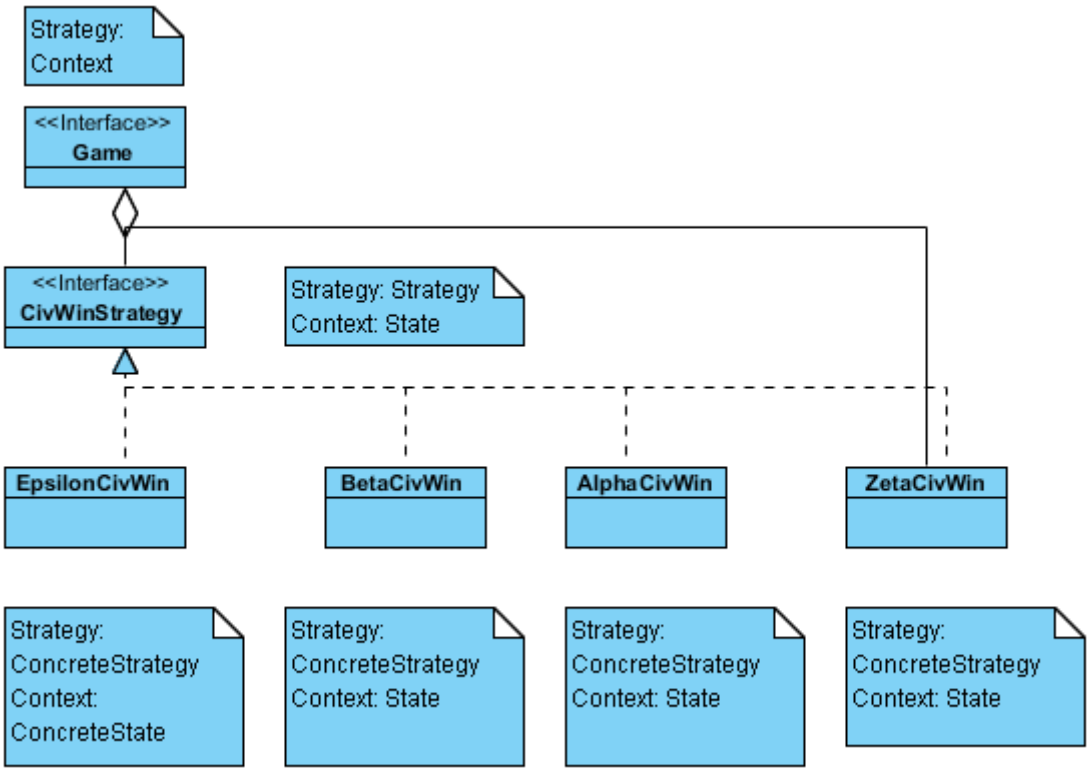
```
private Player getWinner() {
   if (winningVariance == 0)
  {
   // test if all cities are owned by same player
   }
   if (winningVariance == 1) {
   // red wins
   }
   if (winningVariance == 2) {
   // check if a players has won three attacks
   }
   if (winningVariance == 3) {
   // test if anyone has conqured alle cities if round < 20
   // otherwise test who has won 3 attacks (change winning variance to
   }
}
```

36.24



Fordele:
Et polomorfisk design giver færre klasser at holde styr på. Sagt på en anden måde at der er ikke mange klasser og interfaces og det er nemt, hurtigt og beskidt.
Ulemper:
Da koblingen mellem klasser er meget hård kan det være svært at introducere nye varianter uden at lave kode duplikering, da selvom C++ kan have multiple nedarvning er det svært til umuligt at kombinationer uden kode duplikation. Ydermere er det svært at definere hvor den nedarvede funktionalitet stammer fra da det er svært at finde rundt i, når man bruger flere nedarvninger. En anden ulempe er at klasserne kommer til at blive unødvendigt store

36.25



36.26

Behaviour

Objekter der har metoder som udfører et stykke arbejdet og defineres af de metoder på et objekt der bliver kaldt. Vi har endOfTurn der indkremerer alderen og sørger for at by og enheder får production og movement, i hver omgang.

Responsibility

Som navnet antyder har Responsibility ansvaret for at et eller andet krav opretholdes. Det er mere eller mindre lige meget hvordan det løses bare kravet opretholdes.
I CivAttackStrategy interfacet er metoden outcomeOfBAttle som ser på resultatet af en kamp. Ydermere definerer den klart hvad den tager og ikke tager ansvar for.

Role

Ifølge bogen er dette mutual responsibilities som er samarbejdende objekter ved program udførelse skal enes om hvilke roller de hver især påtager sig.
Vorses eksemepl på role er sammenspillet imellem GameImpl og CivAgeStrategy. Hvor GameImpl giver en alder med som argument til metoden ageProgress hvorefter alderen udregnes og returneres. GameImpl forventer at alderen bliver ændret og CivAgeStrategy forventer en alder som ”betaling”

Protocol
Ifølge bogen er dette mutual expectations som er objekter fra Role afsnittet som samarbejdermed andre objekter. Samarbejdet består i at modtage og brug specifikke actions som i vores eksempel kunne være Utility/EpsilonCivAttack da enhederne afhængigt af hvad for en tile de står på og venligsindet enheder omkring dem bliver en bonus udregnet. Reglerne for udregning for denne bonus er klart defineret i ovennævnte klasser.

36.29

Hvis man sammenligner det facade pattern som er på side 282 med det pattern man ser på side 460 ser man at det grafiske interface ser game som en facade. GUI'en er Client'en, game er Facade og klasserne under interfaces og klassen under game er da ”SomeClass” og ”SomeInterface. Fordelen er at GUI’en kun skal tage et game object og har derved uddelegeret ansvaret for alle udregningerne til game. En ulempe kunne være at man måske giver game ansvar over for meget, hvilket kunne være en større mængde metoder udelukkende for klienter kunne for adgang til det underliggende system