dSoftArk

Afleveringsopgave - uge 4 - SemiCiv

Hold: DA4; Gruppe: B

Emil Madsen - 20105376
Rasmus - 20105109
Sverre - 20083549

28. november 2011

36.22

Table of variability points.

Below is our variability table, ie. a table of the things that variant between our different version.

Product	World Aging	UnitActions	World Layout	Winner	Attacking	City
Alphaciv	Default	Default	Default	Default	Default	Default
Betaciv	**Beta**	Default	Default	**Beta**	Default	Default
Deltaciv	Default	Default	**Delta**	Default	Default	Default
Gammaciv	Default	**Gamma**	Default	Default	Default	Default
Epsilonciv	Default	**Epsilon**	Default	**Epsilon**	**Epsilon**	Default
Zetaciv	Default	Default	Default	**Beta/Epsilon**	Default	Default
Etaciv	Default	Default	Default	Default	Default	**Eta**
Semiciv	**Beta**	**Gamma**	**Delta**	**Epsilon**	**Epsilon**	**Eta**

Required changes to production cofe to support semiciv.

Due to the fact that we're using a factory to produce our strategies, and because we've decided to keep our production code free of variant specific code, it does actually not require ANY changes to production code to support the semiciv variant. All there's needed is to produce two classes, a strategy factory, and a unitfactory somewhat alike the gammaunit factory, but with specialized behavior for the settler.

Implementation of the SemiCiv variant.

Implementation of the semiciv variant is, like we mentioned earlier, actually very simple, as it's just a matter of implementing two classes. The first one, the strategy factory, is simply a matter of copy/pasting from the different other strategy factories, such that our semiciv factory returns the correct strategies for everything. However since we do not have a unitfactory, that only modifies the settler, we'll have to write a new unitfactory.

The development of the unitfactory is also really simple, as it's just a matter of copying the gammaunitfactory, and deleting a few lines of code (those that are within the Archer if statement). This is possible because unit's within our unit factories are handled without affecting eachother.

36.23

Analyze the solution that every variant is implemented using parameters.

The design, where all the variant specific code is within the class Game, results in a largely unanalyzable code, as you always have to check which if statement is for what. This is simply because all kind of specific code is within general code, and one will always have to keep track of which code is specific and which code is generic.

Also there's the issue that the game will have a larger set of responsabilities, as it has to handle all of the variants.

And obviously there's the worst issue: that each change to variant code is actually a change within the generic codebase, and it could possibly introduce a defect wihtin the generic code.

Sketch the code of a few methods from game using parameters.

An obvious way to find which method(s) to sketch is by inspecting our variability table, to find the method(s) that differ the most, and in our implementation there's three winning strategies, so let's sketch that one;

```
1  public Player getWinner(String gameType)
2  {
3      // If Betaciv
4      if (gameType = GameConstants.BetaCiv)
5      {
6          // Code
7      }
8      // Otherwise if EpsilonCiv
9      else if(gameType = GameConstants.EpsilonCiv)
10     {
11         // Code
12     }
13     // Otherwise if ZetaCiv
14     else if(gameType = GameConstants.ZetaCiv)
15     {
16         // Code
17     }
18     // Otherwise
19     else
20     {
21         // Code
22     }
23 }
```

Now all of the 'code' could either be inline, or it could be in functions, however both of these alternatives are bad: The inline because it makes it hard to read the code and know which code is for what, and because moving it into functions, makes variant specific code available to every variant within game.

Another issue is that a change to one of the gameTypes, can possibly affect the others, either because of an error, or simply because the change is in some of the code that is shared.

Only a single function was sketched here, as other functions would simply be alike this one.

36.24

The actual design suggested in the exercise, is somewhat alike what we have developed. Instead of making a purely polymorphic design, we have developed a composition strategy factory, for the other strategies. However, these concrete strategies are implemented with inheritance, such that we only has to define the non-default strategies.

Sketch a design implementation of SemiCiv based upon a purely polymorphic approach.

In a programming language that supports multiple inheritance alike C++ or Eiffel, one could implement SemiCiv simply by doing alike what we did, but instead of just loading the default behavior, one could also load the specific behavior of some functions by deciding which super class' function to call.

For instance, in semiciv, we could simply extend all of the variants that we make use of, and then override them, with calls to the specific super. Fx for the winning strategy, we could simply call the betasuper.getWinningStrategy().

Discuss the benefits and liabilities.

The main issue of using multiple inheritance to implement the semiciv variant, is that we'll experience, what is known as 'the diamond problem' that is, the inheritance UML diagram forms a diamond, and as such, the semiciv would have multiple copies of it's super.super class, this is an issue, because we'll have to keep all of these objects in sync. Beside of this issue is also the issue of which super's function to call, an issue that can be resolved in several ways. In C++ this is handled with what's called virtual inheritance.

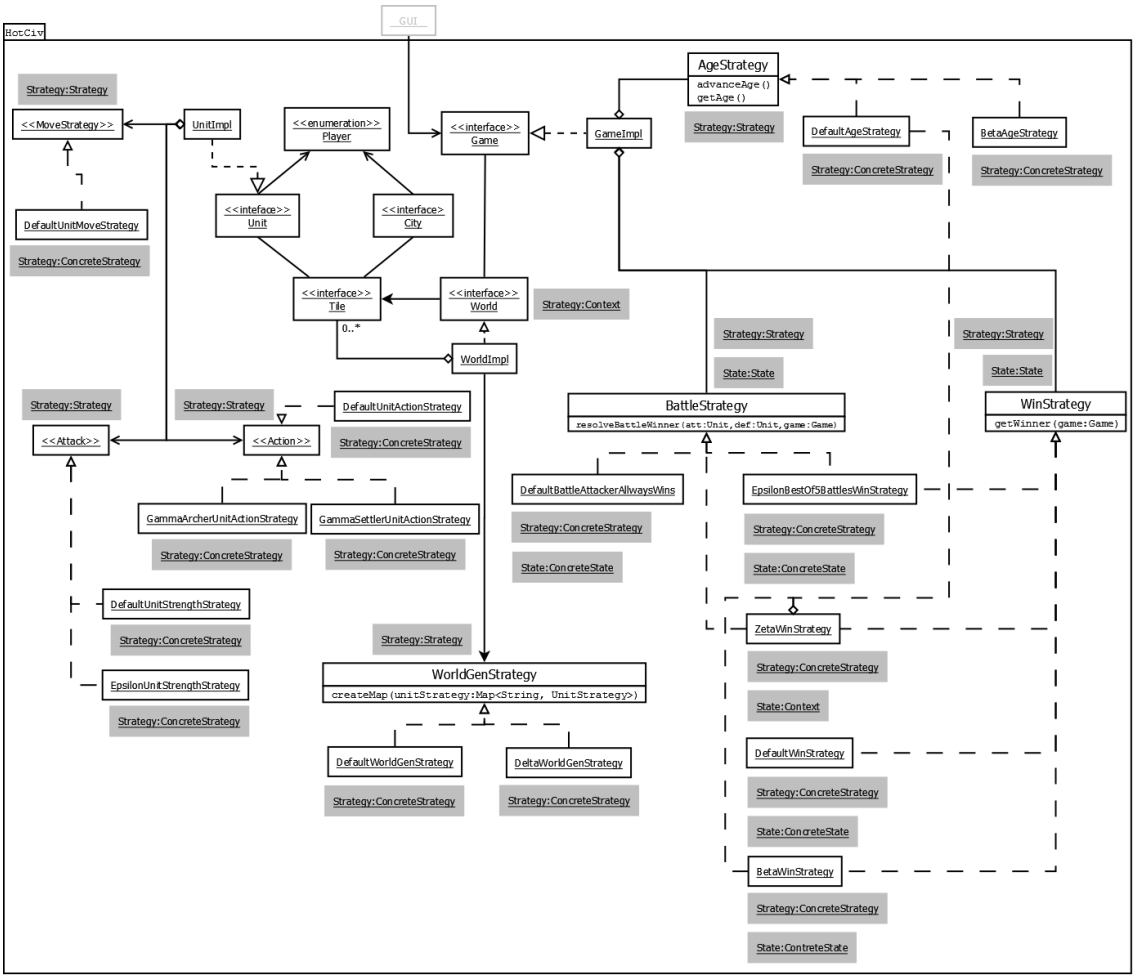
However generally this is just a way to avoid an issue, that one really shouldn't get into, as the need for the UML diamond is a desperate warning of a possibly bad design.

But the benefits of this, compared to the approach we decided to use when we implemented our semiciv design, is that in our design there's actually some duplicate code, and as such, if there's a change to the way that betaciv loads it's strategies, we'll have to update our semiciv strategy accordingly. Whereas in the inheritance approach, it is automatically updated to reflect this change.

Do note however that some languages (for instance lua), can handle this multiple inheritance in a somewhat prettier way, using a multiple inheritance control function, which is basically a function that determins how the multiple inheritance should be performed. That is if there should be multiple super-super classes, or if there should be a shared one, or even a hybrid wheres some super classes share super-supers in groups. With this function you can also decide which functions are to be overloaded, that is which ones are obtainable and which one is the default. However this is mainly possibly because Lua is a dynamically typed language, and because you implement inheritance youself in Lua using the metatable. Hence you have full control as to how.

36.25

Draw a role diagram (see Chapter 18) showing the part of the HotCivv design that handles finding a winner of the game. The diagram must include all winner variants.



The above graphics is also viewable **online**.

36.26

Our design is very much made with responsibility in mind.
The Game interface is responsible for gamelogic.
World is responsible for gameworld consistency, maintainance and modification.
The implementation of Game, GameImpl, delegates some of it's responsibilities in strategies such as determining winners of battles, the aging and winning of the game or/and worldGeneration.

GameImpl additionally takes the role of initializing the game and all related components. It supplies all the needed strategies to where these are used and needed.

To accomplish these responsibilites and roles, there are protocols in place between game and world.
GameImpl expects to recieve all the needed strategies through a Factory pattern.
WorldImp expects all modifications to the gameworld to be done through its methods and expects to recieve a world gen strategy from Game.

36.29

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

Game is the facade to our hotciv subsystem, accessed by the client/user interface.

Benefits:

- It shields the user interface from the subsystem classes, and thereby the specifics of the different variants and resources in the package.
- The relation between game and the client is weakly coupled. This means that we can develop the game subsystem without affecting the client. We just need to keep the well-defined facade interface.
- Any client interface may use any variant of the game subsystem, as relevant for the application.

Liabilities:

- The facade needs to be a well defined interface, or there is a danger of the facade interface being bloated.
- Clients can still access classes in the subsystem. Usage of readonly interfaces is advised.