Visitor Pattern

Gruppe 15 - patterns

Gruppemedlemmer	**Studienummer**
Nicholas Ladefoged	*201500609*
Søren Katborg	*201501911*
Max Barly	*201401694*

Purpose of the pattern

The pattern is use to avoid pollution classes with certain functionality (Single responsibility principle) . And Also to help keep up the Open/Close principle, by being able to add functionally later on.

Like in our example the units doesn't have anything to do with the fighting. So when the fighting is in progress the visitor is in full control over what happens. After we made our fighting we add a building method without having to change the unit classes.

Structure

For the visitor pattern to do work on the different classes, those classes will have to inherit from an abstract class or an interface. This is so that the function that all element needs to have, can call a visitor in the way that the next picture shows. This enables the visit to override what kind of visit the user needs. In our case we have made it so, for instance, that a Viking with axes gets a bonus when he fights and a roman soldier get bonus if he has a spear and a large shield.

```
public interface IVisitorFight : IVisitor
{
    5 references | Katborg, 47 minutes ago | 1 author, 1 change
    new double Visit(IEnglishSoldier unit);
    5 references | Katborg, 47 minutes ago | 1 author, 1 change
    new double Visit(IRomanSoldier unit);
    6 references | Katborg, 47 minutes ago | 1 author, 1 change
    new double Visit(IUnit unit);
    5 references | Katborg, 47 minutes ago | 1 author, 1 change
    new double Visit(IViking unit);
}
```

The visitor specific class itself, will also have to inherit from a common visitor interface, in case more visitor classes a needed. this enables to call accept visit, with the kind of visit the user wishes very easy, next picture illustrates this.

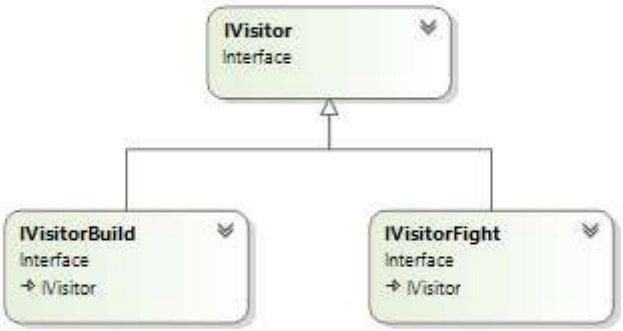
```
public override double AcceptVisit(IVisitor visitor) => visitor.Visit(this);
```

Doing this, every derived class needs to override the AcceptVisit function, so the call gets down to the specific class instead of the base class. An example of this:
“in our case we have a base class call Unit and a derived class called Viking. If Viking did not override AcceptVisit then the visit function would be called with IUnit and the viking would not gets its bonus.”

Design

The design of the visitor pattern uses the pattern down to a degree of 2 ways of visiting. This will show the basic features and advantages of the pattern, without taking it out too far.

Visit:

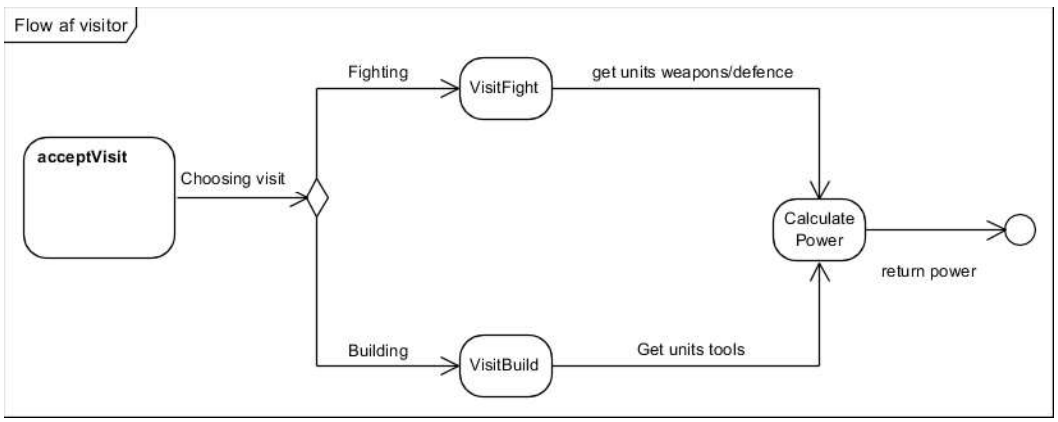


In order to insure that we are following the visitor pattern, we have started off by assigning a general interface IVisitor. This opens up, so that we can implement two different types of scenarios where we want to illustrate.

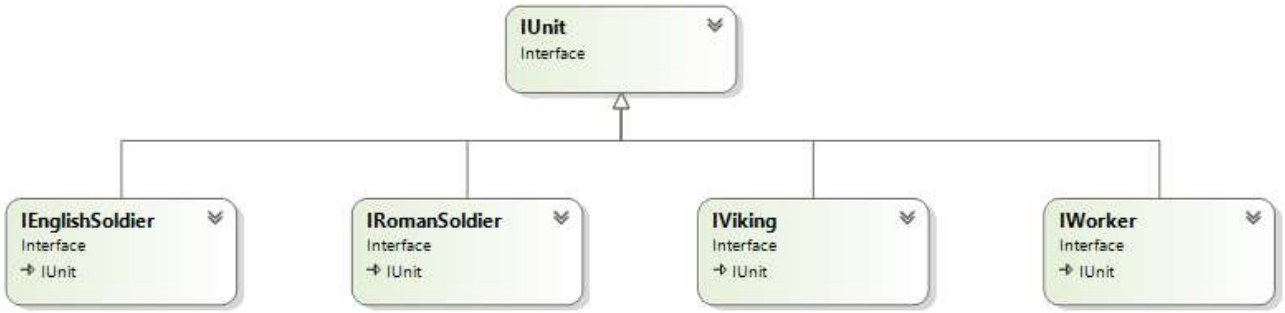
We have chosen to give our units an option to either compete in fights or in building skills. Depending on which scenario chosen, each unit will gain an advantage depending on the scenario (Vikings are always overpowered, as basic logic dictates) However, if we set a worker against a roman soldier, the roman soldier will defeat the worker in combat, however if we set the same units to build, the worker will be superior.

Basics of our visitor-pattern

To illustrate the flow of our visitor pattern, we've created a flow diagram. The essence of visitor pattern is strongly visible here, as it gets very obvious how the visit method, is different based on what visit class is chosen, however, it can manipulate the same units nonetheless.



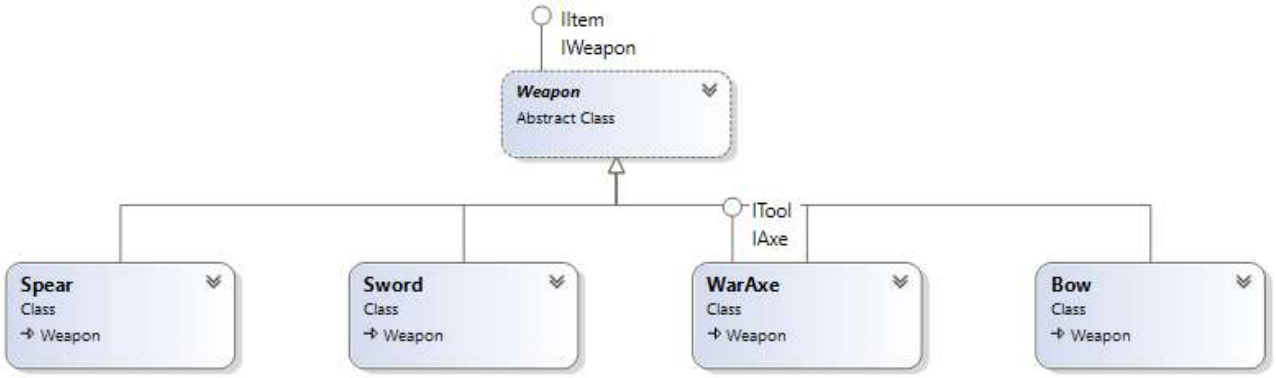
Units:



For the units we have created four different types, where either unit has the same capabilities, the difference is which type of case we decide to visit. The unit is special, because a worker would be assigned default values in a combat scenario.

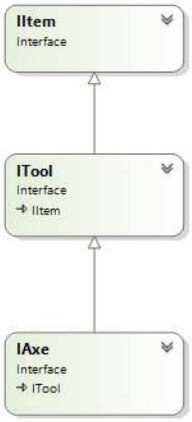
Weapons:

Weapons are something that each unit can equip, to further enhance their fighting capabilities, and in the base concept of visitor, this has not much to do with the pattern, but it adds a nice aspect to our specific design.

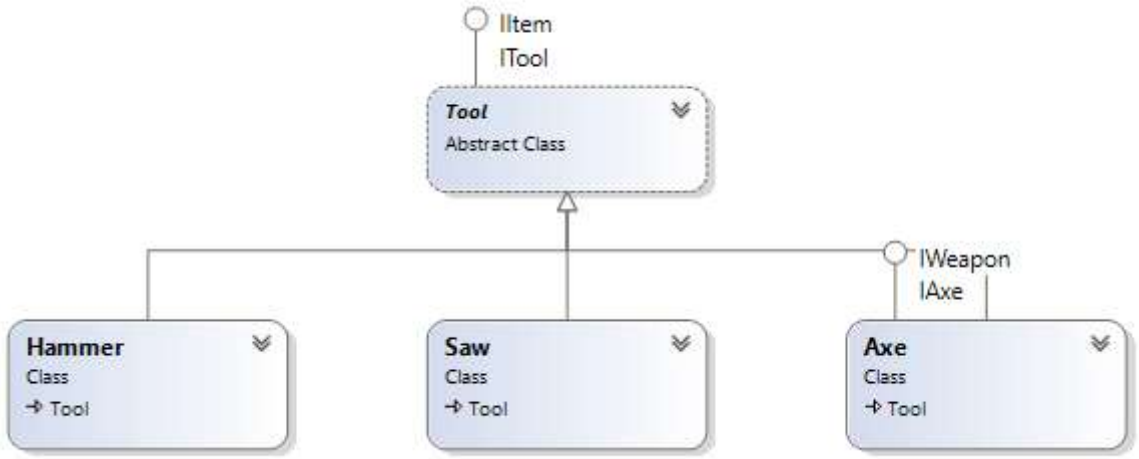


Items:

As with weapons, items are also not part of the actual visitor pattern. We have chosen a few types of common tools, however Axe is a special case, because there is either a axe for the purpose of war, and the axe for the purpose of building. This adds some interesting aspects to the design as well.

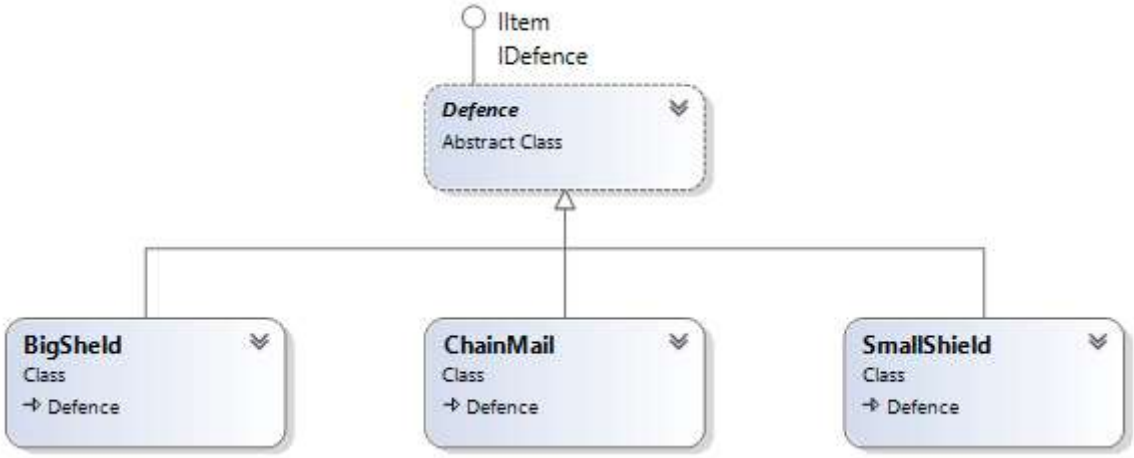


Breaking down the tool to each class, we can see that a total of 3 tools are part of this game. each tool has their own advantages in different scenarios (as explained)



Defence:

Defence is not really what one should think it is in our solution. That is because we didn't want to spend more time on this! So basically defence just adds more fighting power. (Lazy students designing a game). The specific class diagram is shown below!



Fights:

For fighting each unit will receive a base power for their type (viking is op) , also their items (defence and weapons) adds to their power, through visit.

Visitor vs Command vs Servant

Other ways to add/remove functions to a class is using the Command or Servant patterns.

Command:

The command pattern is also a way to move work from a class to another class.
Commands main object is to delegate work not as much adding function. The Command will store information and can be queued until another class has time to do work with the command.
Like if a class put in a order. The order will be the command class that then a cashier can do something with.

Servant:

The Servant pattern is a way to a functionality to a group of classes without putting the functionality in every class.
Like if we have some cars and the have in common that the drive from A to B. So every class can use a Servant class that does the driving.

They all look a bit alike but there are different:
servant: is a simple function that the class needs to know and use.
Command: is used to store information and maybe do some work and pass the info to another class.
Visitor: can be used to add function after a class is done.

When is it useful and when is it not.

The classes that get visited (called the Elements), need to be stable, a lot of revisiting would require alot of elements to be visited again (code heavy)

All Elements needs to inherited from a base class, as all elements, must have the acceptvisit method. furthermore all classes has to have specific interface to determine where they belong in the overridden visit methods.

The Visitor only uses the public interfaces, so if a function needs access to private values Visitor is not gonna work.