Getting Context Information For Strategies.

Finn Rosenbech Jensen

November 11, 2009

1 The Problem.

Introducing different strategies in **HotCiv** we often face the problem, that the *Concrete Strategy* class needs information from it's *Context*. Normally we then send the *Context* class along as a method parameter to the *Concrete Strategy* class. But which type should we send along - a **Game** or a **GameImpl** ? And what should we do if neither of these have methods that will deliver the *Concrete Strategy* class the information it needs ? On the other hand do the *Concrete Strategy* classes really need all the possibilities that a **Game** offers ?

2 An Example.

Lets look at a concrete example. We want to introduce a *WinnerStrategy* interface which should have the responsibility of determining who (if anyone) is the winner of a **HotCiv** game. This means we want the interface to look something like:

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

So far so good. Then we have our *Concrete Strategy* classes. Well how should the **AlphaCivWinnerStrategy** look ? It has to decide whether the game have reached year 3000 BC or not. Hmm - but **Game** very conveniently has a **getAge()** method so we just send our **Game** along:

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

```
public class AlphaCivWinnerStrategy implements WinnerStrategy {
  private final int WINNING_AGE = -3000;
  @Override
  public Player getWinner(Game game) {
    if(game.getAge() >= WINNING_AGE)
      return Player.RED;
    return null;
  }
}
```

That was easy and now for the **BetaCivWinnerStrategy**. It should just check if a player has conquered all cities. Okay, we get the **Game** as a parameter so we just.....?? Well, what do we do ? **Game** doesn't have a **getCities()** or **getOwners()** method.
Now we have a couple of possibilities:

- We could add a **getOwners()** method to the **Game** interface.
  - But this seems wrong as we start to bloat our **Game** interface I.e. we add more and more methods (and thereby responsibilities) to the **Game** interface. This leads to lower cohesion and *Responsibility Erosion* ([1] section 14.5). Besides the only class using this method is **BetaCivWinnerStrategy**.
- We could add a **getOwners()** method to the **GameImpl** class.
  - This does solve the problem. But we would still like only to pass an interface along as parameter to the **getWinner()** method, which means that we will have to downcast the **Game** instance to a **GameImpl** instance in the **BetaCivWinnerStrategy**. This would work but having to downcast is annoying and rests on the implicit precondition that **GameImpl** is the only implementation of **Game** we will work with.
- Actually we could solve the problem with the **Game** interface alone. We would have to iterate over all **Positions**, for each of them check whether there is a **City** on it and check if all cities are owned by the same **Player**.
  - This would definitely work but it does seem like a lot of work to get hold of two cities.

Coming to think of it. Passing a **Game** instance along also is a bit unsatisfying. Why should these *Concrete Strategy* classes have the possibility of calling mutator methods like **moveUnit()** or **endOfTurn()** ? It seems like we have gotten ourselves into yet another fine design mess....

> *Computer science is the discipline that believes all problems can be solved with one more layer of indirection*
>
> Dennis De Bruker

3 Private Interface Pattern comes to rescue !

As often is the case when we have a design problem, the solution is to introduce yet another interface. Here it is actually a design pattern introduced by James Newkirk [2]. We define an interface to be responsible for "publishing the information" that the different *Concrete Strategy* classes need - ***and nothing more*** than that. This will look something like:

```
/** Publishes the methods available for concrete WinnerStrategy classes.
*   This is an application of "Private Interface" Pattern.
*/
public interface WinnerStrategyContext {
  public int getAge();
  public Collection<Player> getOwners();
}
```

Now the **WinnerStrategy** interface will look like:

```
public interface WinnerStrategy {
  public Player getWinner(WinnerStrategyContext context);
}
```

And our *Concrete Strategy* implementations will be:

```
public class AlphaCivWinnerStrategy implements WinnerStrategy {
  private final int WINNING_AGE = -3000;

  @Override
  public Player getWinner(WinnerStrategyContext context) {
    if(context.getAge() >= WINNING_AGE)
      return Player.RED;
    return null;
  }
}
```

```
public class BetaCivWinnerStrategy implements WinnerStrategy {

  @Override
  public Player getWinner(WinnerContext context) {
      Player candidate = null;
      for(Player owner: context.getOwners()) {
          if(candidate == null)
              candidate = owner;
          if (owner != candidate)
              return null;
      }
      return candidate;
  }
}
```

This is all quite simple and nice. The only trickery comes when we call the strategy objects in **GameImpl**. We can do this by using "Ad hoc" interface implementations I.e the creation of an anonymous interface implementation exactly at the spot where we need an implementation:

```
public class GameImpl implements Game {
  ...
  public Player getWinner() {
    return _winnerStrategy.getWinner(new WinnerContext() {
            public int getAge() {
              return GameImpl.this.getAge();
            }
```

```
            public Collection<Player> getOwners() {
              ArrayList<Player> result = new ArrayList<Player>();
              result.add(_redCity.getOwner());
              result.add(_blueCity.getOwner());
              return result;
            }
          });
    }
    ...
}
```

As can be seen my **GameImpl** implementation has a **_winnerStrategy** attribute of type **WinnerStrategy** and furthermore two **City** attributes called **_redCity** and **_blueCity** respectively. Instead of the "on demand" syntactic sugar, we could of course also use a private inner class attribute implementing **WinnerContext**.

Considerations.

So what have we achieved by using Private Interface ? Well, we have gotten some advantages:

1. The *Concrete Strategy* classes are nicely decoupled from **Game** and **GameImpl**. I.e. we have achieved lower coupling.
2. We didn't have to bloat the **Game** interface or use type checking and down-casting in our *Concrete Strategy* classes implementations.
3. The *Concrete Strategy* classes get exactly the amount of information they need.

But there are also some liabilities

1. We had to introduce yet another interface and we might have to do this for most of our Strategy Patterns. This will lead to more complexity in the design.
2. The methods in the **WinnerStrategyContext** interface may be victim to modifications and additions as we get new concrete **Winnerstrategy** implementations.
    - *This problem would also be present without the use of Private Interface Pattern. We could actually argue, that using the pattern exactly pinpoints the place to make these changes I.e. we have achieved higher cohesion.*

4 Interface Segregation Principle comes to rescue !

Private Interface Pattern isn't the only solution to the problem. If we defer from bloating the **Game** interface with new methods we can ensure that *Concrete Strategy* classes don't get access to mutator methods. How ?

Well of course by introducing yet another interface ;-) This idea is to partition the **Game** interface into a mutator and a read-only part (see [1] chapter 19). This can also be seen as an application of the *Interface Segregation Principle.* [3] which can be framed in two ways:

Many client specific interfaces are better than one general purpose interface.

Clients should not be forced to depend on interfaces they don't use.

This means that we introduce a new interface holding all the accessor methods of the **Game** interface (javadoc comments removed):

```
public interface GameContext {
  public Tile getTile( Position p );
  public Unit getUnitAt( Position p );
  public City getCityAt( Position p );
  public Player getPlayerInTurn();
  public Player getWinner();
  public int getAge();
}
```

and then let the original **Game** extend this interface:

```
public interface Game extends GameContext {
  ...
}
```

Now our *Concrete Strategy* classes can be given parameters of the type **GameContext** which hopefully have all the methods needed.

Considerations.

This solution also have it's pros and cons:

1. The *Concrete Strategy* classes have no access to irrelevant mutator methods like **moveUnit()** and **endOfTurn()**.
2. We only have to introduce one more interface.

But there are also some liabilities

1. In case the **GameContext** interface shouldn't hold the methods the *Concrete Strategy* classes need we might have to use type checking and down-casting.
2. The methods in the **GameContext** interface may be victim to modifications and additions as we get new concrete **Winnerstrategy** implementations.

Comparing the solutions we can say that this solution is more *"coarse-grained"* than the Private Interface Pattern solution, as we try to satisfy all possible Strategy Pattern classes with only one interface.

5 Conclusion.

So all in all will the design profit from introducing either of these solutions ? This is difficult to answer conclusive and you might very well get different answers from different people. What you should do is to weight the advantages and liabilities against each other and make up your own mind based on sound judgment.

> *"To write a good program you need intelligence, taste and patience."*
>
> Bjarne Stroustrup

Watch out for the right timing. A word of warning may be justified. All these thoughts and code refactorings belong at step 5 of the TDD rhythm. It's a **bad** idea trying to introduce the solutions before we have a working implementation backed up by a suite of tests. We also are in no real position to judge whether the design improves if we start with introducing the solutions "up front".

References

[1] Henrik Bærbak Christensen *Flexible, Reliable Software: Design Patterns and Frameworks - Agile Development*. To be published by CRC Press 2010.
http://www.baerbak.com

[2] James Newkirk *Private Interface*. PLOP'97 proceedings
http://www.objectmentor.com/resources/articles/privateInterface.pdf

[3] Robert C. Martin *Interface Segregation Principle*.
http://www.objectmentor.com/resources/articles/isp.pdf
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf