Strategy pattern

This is a showcase on few ways to use a strategy pattern. But in this cases we are dynamically chosing the right implementation.

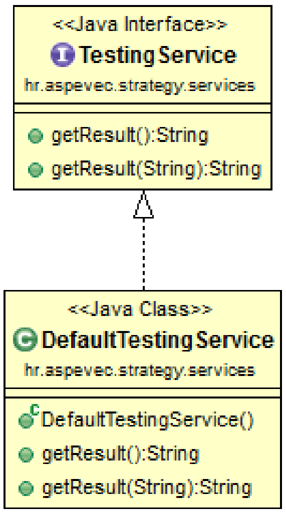
Author: Anđelko Spevec

Year: 2018

Contents

Strategy pattern - implementation

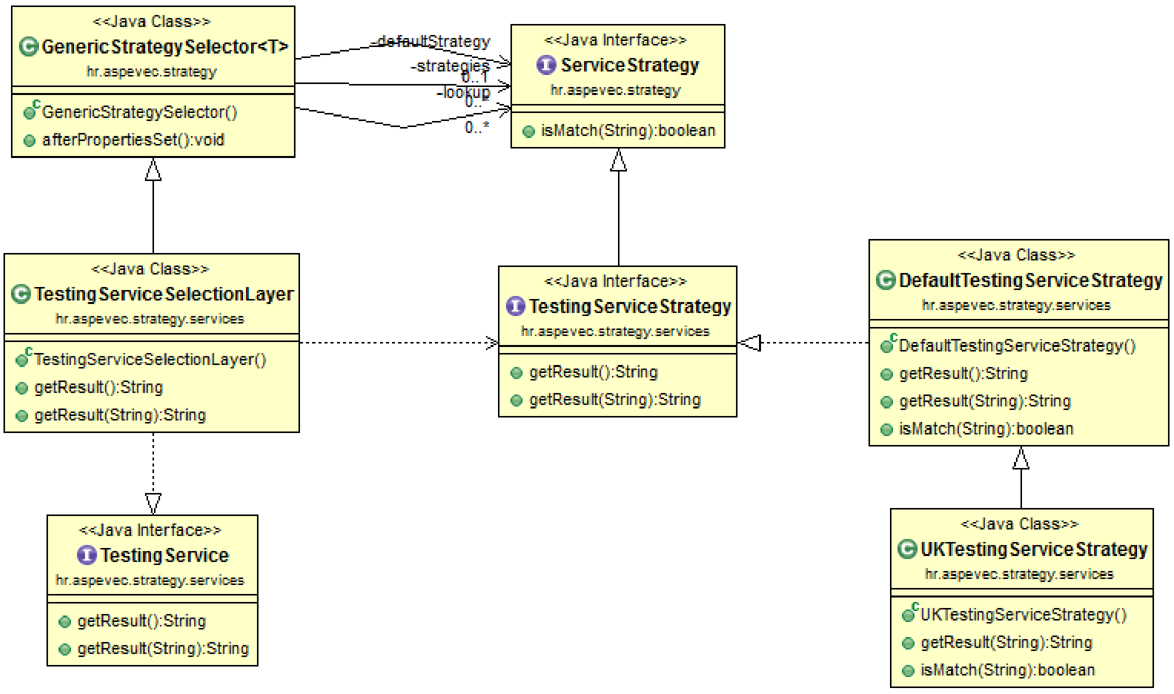
Strategy pattern allows us to have different implementations for the same interface (e.g. each country will have it's own method implementations). By using this pattern we are dynamically selecting appropriate implementation and call it. This is the request that often comes when application is already live and we want to expand our market and add another country. This is the starting point.



Because this *TestingInterface* is already used in other beans, it would not be a good idea to have a selection there. So we need to create a selection layer between interface and the implementations (default and for the other countries). I'm saying default implementation because this will be a common behaviour. Other country may just change one method the common, but other 3 may still want to use the common one. Now I will write steps here to create a selection layer and add a new country implementation.

1) Copy the existing interface and create new one *TestingServiceStrategy* and make that interface extend *ServiceStrategy* interface
2) Rename *DefaultTestingService* to *DefaultTestingServiceStrategy*
    a. implement only *TestingServiceStrategy* (created in step 1)
    b. implement isMatch method and return false
    c. annotate it with @Primary
    d. change the access modifier of everything that is private to protected
3) Create new class and name it *TestingServiceSelectionLayer*
    a. extend *GenericStrategySelector* and add *TestingServiceStrategy* as generic
    b. implement the old TestingService interface
    c. now add all the methods that needs to be implemented
    d. in every method implementation do selectStrategy().method(...);
4) Create new class for UK implementation and name it *UKTestingServiceStrategy*
    a. extend *DefaultTestingServiceStrategy*

b. implement *isMatch method (return CountryEnum.UK.getCode().equals(code);)*
c. change method which you want for UK implementation



The most special class here is GenericStrategySelector which uses *TestingServiceStrategy* interface to inject all the strategies and map them to specific country. So in selection layer we just need to call selectStrategy() method and we will get the right strategy and just forward the method. This is a lot of steps but when this is done once, for adding more countries we just need to repeat step 4.

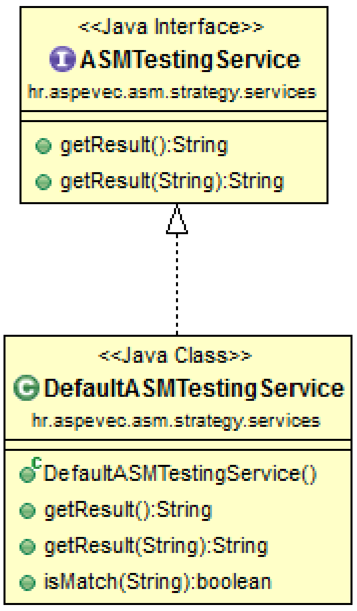
Some drawbacks here are that we have two interfaces with same methods that we need to maintain and that we need to write that selection layer that doesn't do anything smart but just forwarding. Selection layer may give us some more options in the future, but I decided to see if I can do it with less code in the next chapter.

Strategy pattern – ASM implementation

The idea here was that I keep the arhitecture from previous chapter, but to eliminate boilerplate code. It would be nice to create a class dynamically and register it as a bean, right? By using ASM we can do it. This is working with bytecode, so if you are not sure what you are doing or how this will be maintained in the future of your project, perhaps this is not the best approach for your project.

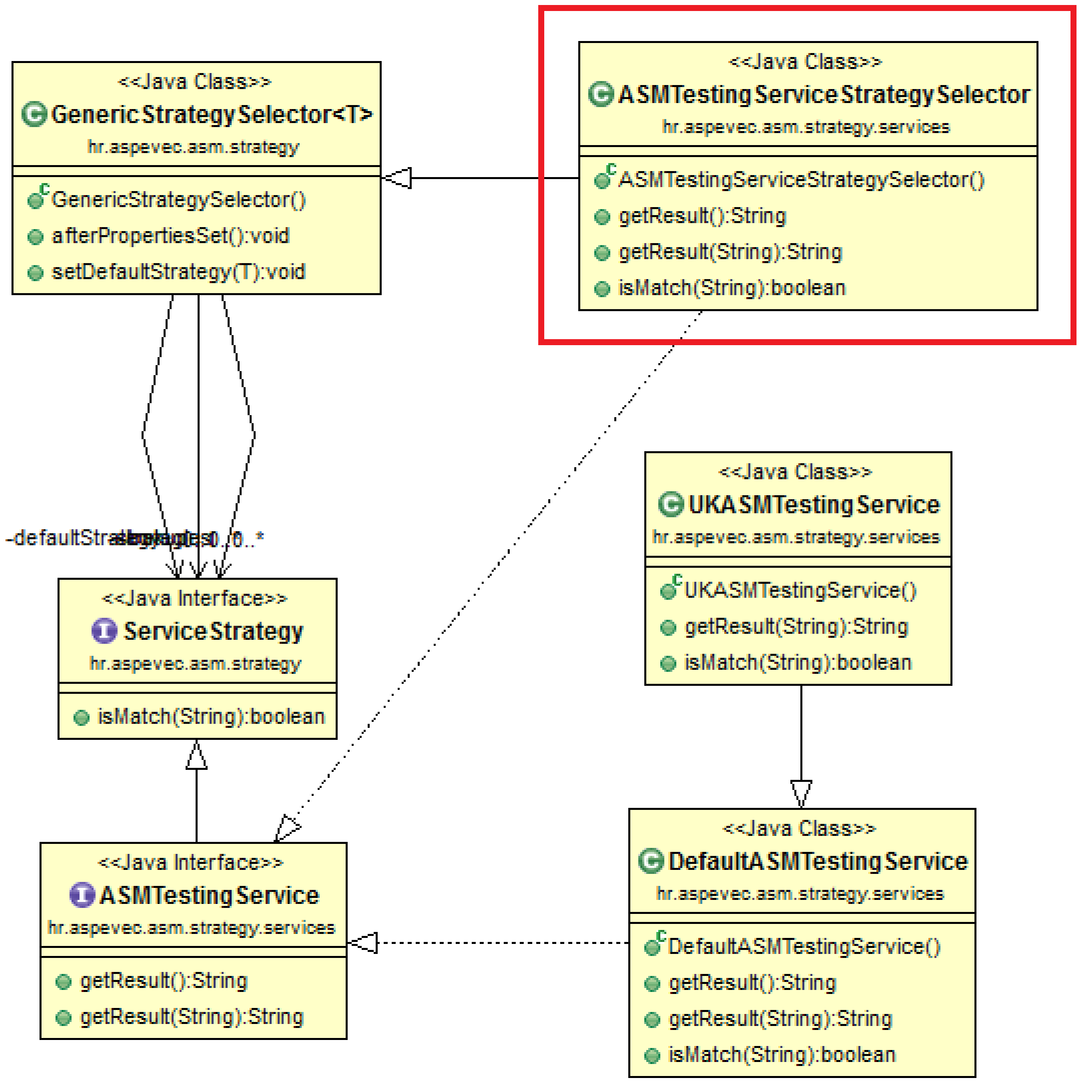
How I did it:

1) I created the new annotation *EnableStrategySelectionLayer* which will be put on Service with common behaviour. And this annotation takes Interface class of this services as a parameter.
2) I created a BeanDefinitionRegistryPostProcessor which checks for my new annotation. When it finds it will:
    a. creates a ClassWriter for my selection layer
    b. creates bean from this class
    c. since this class will also implement the same interface, mark it as primary so it will be @Autowired correctly
    d. fill the default strategy in processor (list of all strategies will be injected by Spring)
3) I needed to create a writer class which will create a selection layer bytecode just by using an interface. When I have a interface I can define the class signature, all the method signatures and method implementation, since it's always calling selectStrategy().method(...); .
4) And I also needed my own ClassLoader which will expose defineClass method.



Now I will also add the implementation for the UK.

1) First go to the *ASMTestingService* interface and extend *ServiceStrategy* interface
2) Go to *DefaultASMTestingService*
    a. implement isMatch method and return false
    b. add EnableStrategySelectionLayer annotation and add *ASMTestingService.class*
3) Create new *UKASMTestingService*
    a. extend *DefaultASMTestingService*
    b. implement *isMatch method (return CountryEnum.UK.getCode().equals(code);)*
    c. change method which you want for UK implementation



Class in red square is the most interesting one because it is created on the runtime. It will extend all the logic for selecting the right strategy from GenericStrategySelector. We just marked it as @Primary, so when ASMTestingService is @Autowired somewhere, it will inject the selection layer which does the forwarding to the right implementation - strategy.

Some literature on ASM/CGLib:

- http://asm.ow2.io/
- https://dzone.com/articles/fully-dynamic-classes-with-asm
- http://www.baeldung.com/java-asm
- https://www.beyondjava.net/blog/quick-guide-writing-byte-code-asm/
- http://download.forge.objectweb.org/asm/asm3-guide.pdf
- https://zeroturnaround.com/rebellabs/java-bytecode-fundamentals-using-objects-and-calling-methods/
- http://andrei.gmxhome.de/bytecode/
- https://dzone.com/articles/cglib-missing-manual
- http://www.baeldung.com/cglib