Tasks design pattern with conditions

All the snippet code will be in java while my code will be implemented in scala, so maybe some solution could get use of the advantages of that language.

I'm trying to perform some post-conditions over some data I get after executing some code and would like to know the best pattern to use.

Let's see that when executing the command I can pass by parameter (this is java) certain information.

```
-DallTheSame=true -DconcreteExpectedResultPerUnit=3,1,1 -DminimumThreshold=60
```

Then I started to look into the best approach to solve the problem of how to be able to extend that in some future.

1. Visitor Pattern

All of the parameter will have an associated class

```
class AllTheSameCheck
class ConcreteExpectedResultPerUnitCheck
class MinimumThresholdCheck
```

all will implement the visit and then the time to execute the checks come, the same object will visit all of them and if required, will execute the check

```
class AllTheSameCheck {
    public void visit(Object object) {
        if(isAllTheSameDefined()){
            // do the check
        }
    }
}
```

and the same code for the others.

2. Decorator Pattern

Same as before in terms of classes

```
class BasicCheck
class AllTheSameDecoratorCheck extends BasicCheck
class ConcreteExpectedResultPerUnitDecoratorCheck extends BasicCheck
class MinimumThresholdDecoratorCheck extends BasicCheck
```

but it will not be a decorator as per see, because in the main execution code they will need to be defined with conditionals

Main Code:

```
BasicCheck basicCheck = new BasicCheck()
if(isAllTheSameDefined()){
    basicCheck = new AllTheSameDefined(basicCheck)
}
if(isConcreteExpected...Defined()){
    basicCheck = new ConcreteExpected....(basicCheck)
}
```

So, having into account that all the post-conditions are defined when executing the tests, which is the best design pattern in that case?

Thanks a lot!

java scala design-patterns

asked Jul 29 '16 at 14:10

RamonBoza

5,889 3 28 44

1 It depends on your intentions. In my opinion, Visitor is not elegant enough and is better applied to problems when you need complex interactions between objects. Decorator is simple and scalable, you can easily add new steps before and after each action. However, it imposes constraints to the execution order (it would be hard to maintain this code if execution order of decorators does matter) and interactions between parameters. – NikolayKondratyev Aug 1 '16 at 7:24

nah the execution order does not matter, its just a matter of checks to be executed over a result that is immutable. – RamonBoza Aug 1 '16 at 7:30

Thus I would prefer Decorator, simple and obvious solution. However you can do it even simpler, create a `List` of classes that will run the checks and call of them (the result will be passed to them as an argument). Pretty simple, but still meet all the requirements, don't you think? – NikolayKondratyev Aug 1 '16 at 7:58

Also the visitor was not the good implementation of it, because the object needs to accept the visitor and it was more like the one you said, a list of Commands to be executed. – RamonBoza Aug 1 '16 at 8:44

Don't use the command line for these checks. Make them part of the API and invoke them with code. – jaco0646 Aug 1 '16 at 14:53

1 Answer

As your checks are obviously certain strategies that should be executed on the presence of certain properties, why not use a strategy pattern therefore?

```
public interface CheckStrategy {
    String getName();
    boolean validate(Object toValidate, String arguments);
}
```

A concrete implementation could now be something like:

```
public class AllTheSameCheck implements CheckStrategy {
    @Override
    public String getName() {
        return "allTheSame";
    }

    @Override
    public boolean validate(Object toValidate, String arguments) {
        ...
    }
}
```

As strategies should only execute some logic but should not hold any state they can simply be registered with a singleton which can also be asked for handling the managment of the validation check.

```
public enum PostValidation {
    INSTANCE;

    private Map<String, CheckStrategy> registeredStrategies = new HashMap<>();

    public void registerNewStrategy(String propertyName, CheckStrategy strategy)
{
        registeredStrategies.put(propertyName, strategy);
    }
```

```
    public boolean check(Object toValidate, Properties properties) {
        boolean checkSucceeded = true;
        for (String key : properties.stringPropertyNames()) {
            CheckStrategy strategy = registeredStrategies.get(key);
            if (null != strategy) {
                checkSucceeded = stategy.validate(toValidate,
properties.getProperty(key));
            }
            if (!checkSucceeded) {
                LOG.warn(strategy.getClass().getSimpleName() + " failed
validation");
                break;
            }
        }
        return checkSucceeded;
    }
}
```

On calling `PostValidation.INSTANCE.check(someObject, somePropertiesMap);` all matching strategies will be executed. If one validation fails, the remaining ones are skipped as proceeding with the validation does not make much sense, but changing this behavior is quite easy though.

All of the available validation strategies should be registered once before the `check(...)` operation is invoked. This would also allow for dynamically adding new strategies on runtime through a plugin mechanism if needed, though the name the strategy should listen for needs to be set within the (system) properties as well.

As you pass the information as system properties (`-Darg=val`), I passed these properties as is to the `check(...)` method. This properties however include plenty more data then actually needed, i.e. certain runtime environment data like the name and version of the operating system, the classpath, ... Plenty of data in the system properties are not really needed for the check method, therefore providing them as application arguments is probably cleaner. On application start you could then store them into a property object like this:

```
public static void main(String ... args) {
    // Register the strategies
    CheckStrategy strat1 = new AllTheSameCheck();
    CheckStrategy strat2 = new ConcreteExpectedResultPerUnitCheck();
    CheckStrategy strat3 = new MinimumThresholdDecoratorCheck();

    PostValidation.INSTANCE.registerNewStrategy(strat1.getName(), strat1);
    PostValidation.INSTANCE.registerNewStrategy(strat2.getName(), strat2);
    PostValidation.INSTANCE.registerNewStrategy(strat3.getName(), strat3);
    ...
```

```
    // Parse the arguments
    Properties prop = new Properties();
    for (String arg : args) {
        String[] kv = arg.split("=");
        if (kv.length == 2) {
            prop.setProperty(kv[0], kv[1]);
        }
    }
    // Pass the passed arguments to the application
    App app = new App(prop);
    ...
}
```

As the actual values stored inside the properties are of string nature, they obviously need to be converted to the respective Java type within the `validate(...)` method. An heuristic for converting the string to Java types could look like this:

```
public class BaseType<T> {
    public T value;
    public Class<T> type;

    public BaseType(T value, Class<T> type) {
        this.value = value;
        this.type = type;
    }

    public T getValue() {
        return this.value;
    }

    public Class<T> getType() {
        return this.type;
    }

    public static BaseType<?> getBaseType(String string) {
        if (string == null) {
            throw new IllegalArgumentException("The provided string must not be
null");
        }

        if ("true".equals(string) || "false".equals(string)) {
            return new BaseType<>(Boolean.getBoolean(string), Boolean.class);
        }
        else if (string.startsWith("'")) {
            return new BaseType<>(string, String.class);
        }
        else if (string.contains(",")) {
            List<BaseType<?>> baseTypes = new ArrayList<>();
```

```
            String[] values = string.split(",");
            for (String value : values) {
                baseTypes.add(getBaseType(value));
            }
            return new BaseType<>(baseTypes, List.class);
        }
        else if (string.contains(".")) {
            return new BaseType<>(Float.parseFloat(string), Float.class);
        }
        else {
            return new BaseType<>(Integer.parseInt(string), Integer.class);
        }
    }
}
```

`getBaseType(String)` is a factory method which has a simple heuristic to determine the type the string may represent. As I do not know your needs and the actual problem description is a bit limited you might need to adapt the `BaseType` class to your needs.

Within the `AllTheSameCheck` validate method you can now convert the passed string argument value using the proposed `BaseType` class from above like this:

```
public boolean validate(Object toValidate, String arguments) {
    boolean value = false;
    BaseType<?> baseType = BaseType.getBaseType(arguments);
    if (Boolean.class.equals(baseType.getType)) {
        value = baseType.getValue();
    }
    ...
}
```

while the `ConcreteExpectedResultPerUnitCheck`, which gets a series of int values passed, may look like this:

```
public boolean validate(Object toValidate, String arguments) {
    List<Integer> values = new ArrayList<>();
    BaseType<?> baseType = BaseType.getBaseType(arguments);
    if (List.class.equals(baseType.getType)) {
        List<BaseType<?>> listType = (List<BaseType<?>>)baseType.getValue();
        for (BaseTyep<?> base : listType) {
            if (Integer.class.equals(base.getType())) {
                values.add(base.getValue());
            }
        }
    }
```

```
    ...
}
```

As you asked for an extensible solution for future usage, the proposed strategy pattern decouples the actual implementation to its own class. If you are capable of adding new classes on the fly i. e. through providing a plugin mechanism, you can simply add the JAR containing the new strategy to your application, register the contained strategy with the `PostValidation` singleton and add the name the strategy should trigger to the properties.

The only downside for future needs I currently see is, when a strategy needs multiple inputs. Here you probably should pass the whole parameters object to the strategy rather then the string containing the respective argument value.

As I haven't programmed in Scala yet, I kept my ideas in plain Java. Transforming the code to Scala should however not be a big problem I guess.

edited Aug 1 '16 at 12:14

answered Aug 1 '16 at 11:50

Roman Vottner
5,119 1 24 34