Lab 1, Design Patterns (TDDB84). Fall 2015

Goal

In this first lab, the aim is that you familiarize yourselves with how design patterns can be implemented in existing code, how they can be used to make changes to an existing codebase, and what effect they will have on an application.

Tasks

In your teams, you will divide the three main tasks (A, B and C) on your three pairs. The tasks are related to the application of design patterns in various contexts. The source code for each task is given on the course web.

Task A: The Builder and State Patterns

A.1 This task requires you to use an existing design pattern and try to use the design pattern to modify and extend the behavior of an existing application. The code for this task is included in the `builder` Java package.

The Builder pattern is intended to simplify multi-step build processes involving the construction of composite objects. Here, we have an application of the design pattern in a small example where we create two sets of objects: a list of e-mail addresses and a list of supervisor names.

Your task is to modify the application to create a set of `Project` objects using the design pattern. The `Project` objects should contain all information that is available from the XML file, and your solution should conform to the structure of a Builder design.

Ensure that there is a clear separation of concerns between the *Builder* and the *Director* in this case.

Also, answer the following questions:

1. **What class has the role of the *Director* in the given example? What class has the role of an *AbstractBuilder* and a *ConcreteBuilder*?**

2. **What functionality does the *Director* have to provide, and what functionality does a *ConcreteBuilder* have to provide?**

A.2 This task requires you to implement the *State* pattern in the code example (package `state`). There is already an interface `State`, and an implementation of that interface `NoQuarterState`. You will need to identify what a *State* pattern is, how it is usually implemented, and how it is intended to be implemented in the given case. Then, you will have to implement the missing parts in the design given.

Also, answer the following questions:

1. **If a new *State* was to be introduced, what other parts of the application would be affected?** Create a new *State* to illustrate your answer.

2. **Why does it matter that you use the *State* design pattern instead of constants and switch statements?** Justify by referring to one of the design principles in SOLID.

Task B: The Bridge and Factory Method patterns

In this example, two software companies have merged their operations and want to reuse the same build systems for their software. Both have started to implement functionality to use build configuration files, in different formats. In one company, the developers have created an Ant-based build system that is programmatically interpreted to provide information for build servers, and allow customizations of the build process. Ant in turn uses XML files for build configurations. The other company uses Yaml for their build configuration files. The implementations are not complete, and somewhat different, but still provide information about the build system. Some merge operations have been performed, such as moving all domain classes that describe concepts in the configuration files, to a separate package "domain". Still, it is awkward to developers who use the configuration parsers to have to switch between the two implementations depending on what configuration file format is used. Your task is to simplify this using the structural design pattern *Bridge* and possibly *Adapter* that will allow developers to choose different implementations without worrying about details.

Also, you should be able to answer the following two questions:

1. **How can the *Factory Method* design pattern enable access to these two different implementations?** Explain with an example.

2. **How can your solution benefit a client that uses this implementation?** Justify your design by explaining how the design is intended to be used, and what information is required by the client (application that uses your *Bridge* implementation).

Task C: The Interpreter and Visitor Patterns

C.1 Explaining the Visitor

When the *Visitor* design pattern is used, responsibility is divided between the structure being traversed and the *Visitor* who performs actions on the structure. It is a pattern that relies on knowledge of how polymorphism works, that is, how methods are invoked at runtime. In this example, you will explore a non-functioning *Visitor* implementation. You will do this by first familiarizing yourself with the basics of polymorphism and single dispatch, and later explain behavior related to the *Visitor* design pattern.

Read how subtype polymorphism works and explain the behaviour in the `polymorphism.PlymorphismTest` class.

1. Explain why the actual runtime type of subAsBase is not taken into account when invoking the aMethod method. Try the other invokations that are deactivated currently and explain the results of invoking a method called `aMethod` there.

2. Explain why the implementation in the visitor package does not work as intended. In the `VisitorTest` class, there is a small program that is intended to be used to illustrate the *Visitor* design pattern, but it does not work correctly. There are two errors: one that gives error messages and another that gives an incorrect result of the printed representation of the expression. Identify what is wrong, explain the error and apply a fix to eliminate the errors.

C.2 Using the Interpreter

The Interpreter pattern is exemplified in packages `interpreter` and `interpreter.lisp`, where two small languages are interpreted. One of them is a mini-version of the programming language Scheme.

The programming language uses parentheses to delimit all compound expressions, and the first term in a compound expression is the operator/function name/language keyword. Thus, instead of “1+2” you would write “(+ 1 2)”. This form is also called *prefix form*, compared to *infix form* where the operator is between the operands, and *postfix form*, where the operator is listed after the operands (as in package `interpreter`). All expressions are evaluated by first resolving the first symbol that is encountered in the expression (“+” in “(+ 1 2)”), and then resolving the remaining subexpressions.

Look at class `interpreter.lisp.InterpreterTest` for a description of the syntax of some expressions in the language.

Try to write your own expressions and use the interpreter to evaluate the results.

1. What types of expressions are evaluated to themselves? Why?

You are to add another type of expression:

```
(case <cond>
  (<val1> <expr1>)
  (<val2> <expr2>)
```

```
  ...
)
```

<cond> is an expression that will evaluate to a value that will have to be equal to either <val1>, <val2> or any other value in the case expression. Depending on the value of <cond>, either <expr1> or <expr2> may be evaluated and used as the value of the expression. The default result will be `Constants.FALSE`.

Here are some examples:

```
(case 1
  (2 0)
  ((* 1 1) 1)
)
=> 1

(case (* 3 (+ 1 2))
  ((+ 6 2) 0)
  (14 2)
  ((+ 6 3) (- 2 1))
)
=> 1

(def f (x)
  (case x
    (1 0)
  )
)

(f 0)
=> FALSE
```

How would you need to change the existing structure?

2. Implement the new conditional expression using the Interpreter design pattern.

When implementing this addition, you will notice that the `CompoundExpression` class will need to be modified. Make sure that you decouple `CompoundExpression` from the concrete subclasses such as `Conditional`, `FunctionCall` and your new addition. There should be no direct connections between `CompoundExpression` and its subclasses, so that it is possible to add other expression types later on without violating the Open/Closed principle here: `CompoundExpression` should be closed for modifications that pertain to other classes.

Comparing this design pattern to *Visitor*, would it be possible to implement the `interpreter.lisp` functionality with visitors?

3. Try to create visitors for evaluating expressions and explain how it works when you evaluate compound expressions such as function calls. It is not imperative that you solve problems you encounter when implementing the Lisp interpreter using Visitors, but you must include sufficient code examples to justify your reasoning in your answers.

In sum, provide solutions and answers to each of the highlighted sentences above.

Reporting your results

You are to

1. submit your results to each other in your team of six,
2. provide comments on each others' solutions, and
3. submit a report where you describe the initial answers, along with a summary of your peers comments and a reflection on the outcome from your discussions, prior to the seminar pertaining to lab 1.

See specific instructions on the course web for submissions and deadlines.