Software Engineering — Lab 3

Uppsala University

Spring 2013

It's time to get in to design patterns. During this lab we'll practice UML, explore design patterns, and eventually refactor the chat systems we built during the first block. Using our design pattern skills.

Work in pairs and take turns writing code!

Contents

1 Modelling Use Cases with UML (optional)

Consider the requirements for the following library system:

- The library lends books and magazines to borrowers who registered in the system (as are the books and magazines).
- The library handles the purchase of new titles. Old books and magazines are removed when they are out of date or in poor condition.
- The librarian is an employee of the library who interacts with the customs (borrowers) and whose work is supported by the system.
- A borrower can reserve a book or magazine that is not currently available, so that when it is returned they're notified.
- The reservation is canceled when the borrower checks and the book or magazine, or through an explicit cancelling procedure.
- The librarian can create, update, and delete information about titles, borrowers, loans, and reservation in the system.

Task: Identify the actors and create a use case diagram of the library system.

2 UML Class Diagrams (optional)

A customer wants to make an order from a retail catalog. The customer can make the payment in one of these three kinds: cash, cheque, or credit card. The order contains order details, each with its associated item. Each order carries the date in which it has been posted and its status (pending, delivered, etc).

Task: Create a class diagram from the above description, begin with a noun and verb analysis.

3 Reverse engineering with UML (optional)

In this exercise, you are expected to "reverse engineer" the chat system you built in block one. If you managed to get the chat system fully functional during the first block then use that version. If not, use the chat-mock that you built.

Task: Model *class diagrams* that represent the way your solution currently works.

4 Observer Pattern

Implement the *Observer* pattern that we discussed in the lectures using the Weather Station Project as inspiration. You will have to change the code from Java to C#. Add the Heat Index display element as described on page 61.

5 Command Pattern

The fragment below is the core of a calculator implemented using the *Command Pattern*. Develop the rest of the code and demonstrate it running on a variety of sample input.

```
class CalculatorCommand : ICommand
{
  public char Operator{ get; private set; }
  public int Operand{ get; private set; }

  private Calculator _calculator;

  // Constructor
  public CalculatorCommand(Calculator calculator,
    char @operator, int operand)
  {
    _calculator = calculator;
    Operator = @operator;
    Operand = operand;
  }

  // Execute new command
  public void Execute()
  {
    _calculator.Operation(Operator, Operand);
  }

  // Unexecute last command
  public void UnExecute()
  {
    _calculator.Operation(undo(Operator), Operand);
  }

  // Returns opposite operator for given operator
  private char undo(char @operator)
  {
    switch (@operator)
    {
      case '+': return '-';
```

```
            case '-': return '+';
            case '*': return '/';
            case '/': return '*';
            default: throw new
              ArgumentException("@operator");
        }
    }
}
```

6 Visitor Pattern

Develop a *Visitor* pattern to calculate the salary of employees, that can be composed of secretaries and managers, like we saw in the second week of the course.

7 Choose a pattern (optional)

Implement one other of the following patterns according to the course book:

Decorator pattern

Factory pattern

Singleton pattern

Adapter or Facade pattern

Template method pattern

Composite pattern

State pattern

Proxy pattern

8 Applying design patterns to existing code

When building the chat system in block 1, little attention was paid to design principles and patterns of software design. In this assignment you are expected to systematically analyze your chat system and make changes to the design.

Task: Take at least *three patterns into consideration*, writing down your reflections. Implement all of which of the three you find appropriate, and keep a written log of the changes you make. *At least one* pattern has to be implemented.

Larmans creator pattern Are objects created in the right place?

Larmans expert pattern Is the responsibility to do things assigned to the correct classes?

Dependency Inversion principle Are there dependencies between concrete system parts that may cause problems to maintain and extend the code? Could these dependencies be removed, e.g. through strategically design interfaces?

Observer pattern How is different parts of the system being notified of change in state? Could an event-driven solution be beneficial?

Decorator pattern Does any class need a change in its responsibility during runtime?

Factory pattern Is the system suffering from high coupling and dependencies or repetitive complex instantiation of objects?

Singleton pattern Do your system need to keep track of objects from a specific type? Should the solution be simplified with just having a single object of that type?

Command pattern Could some part of the system benefit from encapsulate method invocation?

Model View Controller pattern Is managing the entire system overwhelming, could MVC keep it manageable.

Further you should take this following principles into consideration:

The single responsibility principle Does every single class in the solution have a single responsibility?

Open-closed principle Is the code possible to change without making changes to the existing code?

Interface segregation principle Are there interfaces in the code that could be logically separated into several smaller interfaces?

Be sure you keep a copy of the original chat system. So that after solving this assignment you have two different designs of the chat system. During the re-design, keep a written log of the changes you make and the reason for them. An example of an entry in this log could be:

May 1, 2013: New class named X . The class X was introduced since the creation of objects of type Y didn't fit into the responsibility of any of the existing classes. Class X creates objects of the types Y in accordance with Larman's creator pattern.

Good luck!