Refactoring To Patterns

Added by Amiruddin Nagri, last edited by Amiruddin Nagri on Aug 09, 2012 (view change)
Labels: (None) EDIT

Target Audience

This section of Bootcamp is for students who are familiar with the basic OO concepts, basic Refactoring, basic awareness of IDE features and have basic sense of clean code.

Instructor's Notes

You can find the problem for workshops on Github. The branch master have the problem to be solved, the branch solutions have the final state after implementing the suggested refactoring. More specific notes are included in the solutions branch along with the respective module.

It is suggested that the trainers solve the sample problems before the session to get better understanding of issues faced by trainee

Session Objectives/Key Learning Points

By the end of this session students should -

- Understand the advantages of clear naming and clean code
- Understand how comments are maintenance overhead and does not help in understanding code
- Learn Refactoring tools supported by IDE
- Understand the benefits and disadvantages of using Design Patterns
- Be able to describe Design Patterns using
    - Title
    - Example Problem
    - Pattern/Solution
    - Benefits/Liabilities
    - Refactoring steps

Session Overview

Each of the problem can be solved in a session of 1:30 min. Assuming there are 4 such session during the day, 4 of the problems can be covered with their learnings.

Activity	Time	Elapsed Time (hh:mm)
First attempt at reviewing the code	00:10	00:10

Discuss and list out issues with the given code	00:15	00:25
First attempt at solving the issues	00:15	00:40
Discuss the roadblocks while solving the problem	00:10	00:50
Facilitated discovery of how to solve	00:10	01:00
Second attempt at solving	00:10	01:10
Show and Tell solutions	00:15	01:25
Close	00:05	01:30

Session Notes

What Design Patterns provide us with -

- Provides a template (good, well understood) solution to common problems
- Provides a common language when talking about a solution
- Allows for code changes - breaks brittle or tightly coupled code into something easier to digest

Elicit what disadvantages Design Patterns might have

- Extra complexity
- Break encapsulation a bit and stealing work
- Often overused
- Misuse of common language - people having different ideas of what a Strategy is

Explain how we will be doing Refactoring and Design Patterns:

- Example Problem
- Pattern/Solution
- Trade Offs
- Refactoring Steps

Examples - 1-comments-and-naming

Title - Comments, Naming, Long Methods

Problem -

- Comment hints at the code not being communicative
- Commented out code is out of date, does not compile with current code
- Naming using unclear, short names makes code uncommunicative

Pattern/Solution -

- Rename variables
- Remove comments
- Extract methods

Benefits/Liabilities -

- Removes unneeded comments, forces code to be more communicative
- Some comments are helpful, specially for complex algorithm/optimization in code

Refactoring Steps -

- Introduce Rename Variable (+ shortcut)
- Introduce Extract Method (+ shortcut)

Example - 2-duplication

Title - Duplication in code

Problem -

- There is a repetitive pattern in code
- If the change is needed in this pattern, have to make same changes in multiple places

Pattern/Solution -

- Extract the duplicated code in a method
- Parameterize the parts which are not same

Benefits/Liabilities -

- If there is change in logic, don't have to make changes in multiple places
- May drive DRY principle too far, code becomes too hard to work with

Refactoring Steps -

- Extract method
- Introduce as parameter (+ shortcut)

Example - 3-0-state

Title - State Pattern

Problem -

- The next state of the object is dependent on the previous state
- There is a complex conditional (if-else-switch) statement to determine next state

Pattern/Solution -

- Decouple the behaviors from the object and isolate them into their own class that shares an interface
- Implement the specific logic for next state in the implementation of interface method
- Delegate the next state determination to this method

Benefits/Liabilities -

- Have a clear and separated logic
- State and Strategy are if-killers, reduces the number of conditional logic in code

Refactoring Steps -

- Create type safe enums (pre java enum) or enums for different states
- Create an abstract method that will return the next state
- Move the logic from the original method to the implementation of abstract method for each state
- Delegate to the implemented method

Example - 3-conditionals

Optional Exercise

This problem is very similar to 3-0-state, so skip it if you have done the previous one.

Title - State Pattern (same as state, slightly involved)

- The next state of the object is dependent on the previous state
- There is a complex conditional (if-else-switch) statement to determine next state

Pattern/Solution -

- Decouple the behaviors from the object and isolate them into their own class that shares an interface
- Implement the specific logic for next state in the implementation of interface method
- Delegate the next state determination to this method

Benefits/Liabilities -

- Have a clear and separated logic
- State and Strategy are if-killers, reduces the number of conditional logic in code

Refactoring Steps -

- Create type safe enums (pre java enum) or enums for different states
- Create an abstract method that will return the next state
- Move the logic from the original method to the implementation of abstract method for each state
- Delegate to the implemented method

Example - 4-strategy

Title - Strategy pattern

- Object needs to have different ways of doing the same task. You want to be able to choose between different algorithms/behaviours for an object
- Numerous of if/else statements determining behavior

Pattern/Solution -

- Decouple the behaviours from the object and isolate them into their own classes that share an interface
- Make the original object delegate to these classes
- Tell the object which strategy to use when creating it or elsewhere (can be runtime)
- Strategies typically
  - Are implemented as anonymous inner classes
  - Don't hold state
  - Take the original object as a parameter
  - Are declared as constants in the original object

Benefits/Liabilities -

- Enables one algorithm to be swapped for another at runtime.
- Complex design - intention of code is not as clear
- Often static/constant implementations. This forces you to pass in containing object to work on, polluting your interface slightly.

Refactoring Steps -

- Extract the sequence of if/else statements into a class (call it Strategy) - delegate to this class

- One if statement at a time, extract the logic into a separate object which is a subclass of Strategy - let the if/else dictate which object to use
- Once each block is a separate object, make the original code store only one strategy object and delegate to this object
- Make Strategy an interface

Example - 5-feature-envy

Title - Feature Envy

- The parent object uses information from the Pattern/Solution -
- Delegate the calculation to the concerned object

Benefits/Liabilities -

- Avoids breaking encapsulation
- The same calculation used other places can be avoided

Refactoring Steps -

- Extract the method where the information from enclosing object is being pulled out
- Introduce Move method (+shortcut) to the enclosing object

Equipment Required

- Git to be installed on graduates computers
- Internet connectivity
- IntelliJ IDEA

Activity/Exercise Set up and Requirements

- Projector

0 comments | Add Comment