This page last changed on Jun 27, 2010 by smoghe.

Introduction to Design Patterns

Instructor's Note

In TWUs we used this Story list to drive this session.
Also take a look at this sample implementation for these stories.

Jonathan and Deepthi felt that it was best to baseline code after each pattern was implemented and discussed. Our baseline code at all points started with the observer pattern implemented. We think it makes sense to try and derive each design pattern separately.
We managed to get through Observer and Strategy Pattern implementation and discussion. And we managed to get through discussion of the composite pattern but did not get to implementing it with the entire class.

Session Objectives

By the end of this session students should ...

- Understand the benefits and disadvantages of using Design Patterns
- Be able to describe Design Patterns using
  - title
  - problem
  - pattern/solution
  - benefits/liabilities
  - examples
  - refactoring steps

Session Overview

Activity
• Introduce Patterns • Describe the Strategy Pattern • Describe the NullObject Pattern

Session Notes

Introduce Patterns

Today we start Part II of Object Bootcamp. Part I focused on Object Principles and Agile Practices. Part II is going to focus on Design Patterns. We already saw a couple Design Patterns during Part I. What were they? (Students should list Strategy and NullObject). Design Patterns originated as an architectural concept with Christopher Alexander. He looked for patterns in the design of buildings and cities. As he collected more and more he developed a pattern language with which to describe the patterns. His ideas have had a limited impact on the construction industry. However they have had a profound influence within our world - information technology. In the late 80s Kent Beck and

Ward Cunningham (remember them from Tektronics - they also invented XP, JUnit, the wiki, TDD and FIT) began experimenting with the idea of applying patterns to programming. The idea really took hold with the Gang of Four book (Design Patterns) in 1994, and the next year the Portland Pattern Repository (c2.com) was set up. Recently Design Patterns have been extended beyond core programming to EAI patterns, SOA patterns, etc.

Ask the students to define what a Design Pattern is? A solution to commonly occurring problems.

Elicit why we might want Design Patterns

- 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 describing Design Patterns:

- Title
- Problem
- Pattern/Solution
- TradeOffs
- Example
- Refactoring Steps

Describe the Strategy Pattern

Title

- Strategy

Problem

- 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

Example

- Coster in Graph - strategies allow us to define different methods to calculating the cost of traversing nodes

Benefit/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

1. Extract the sequence of if/else statements into a class (call it Strategy) - delegate to this class
2. 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
3. Once each block is a separate object, make the original code store only one strategy object and delegate to this object
4. Make Strategy an interface

Describe the NullObject Pattern

Title

- NullObject

Problem

- Many nulls running all over system
- Numerous if (obj == null) checks before any logic on an object

Solution

- Create an object with no behavior (null object) to avoid the null checks
- Null object has the same methods as the actual object, but does nothing

Example

- In the Node example, we have Path and NO_PATH objects. The caller can call hops(), cost(), etc.

without worrying about whether the object is NO_PATH or just a Path.

Benefit/Liabilities

- Can implement as either (1) anonymous inner subclass or (2) as a separate class implementing the same interface
    - A null object doesn't really meet the two criteria needed for inheritance - Grandmother test and wanting most of the behavior of the super class.
    - Don't want to use real implementation – using an interface forces you to handle changes to the interface in the NullObject implementation
- Do not introduce a null object to replace only one or two null checks
- Complicates maintenance. Must override all methods of superclass/interface.

Refactoring steps

1. Either subclass the original object or extract an interface
2. Implement all methods to have no unintended side effects (ie. add(Link) does not actually add the link, cost() always returns Integer.MAX_VALUE)
3. Replace references to null with references to the null object
4. Remove the null checks