Dashboard > Training > ... > Object Boot Camp > Design Pattern Conclusion

Search

ThoughtWorks

Training

Welcome Luca Minudel | History | Preferences | Log Out

Design Pattern Conclusion

View Info Review

Browse Space

Added by Rolf Russell, last edited by David S Wood on Dec 07, 2006 (view change)
Labels: (None) EDIT

Design Pattern Conclusion

Session Objectives

By the end of this session students should ...

- Be aware of Singleton
- Be aware of State
- Be aware of Proxy
- Be aware of the danger of overusing patterns

Session Overview

Activity
Singleton State Proxy Memento Overuse of Patterns

Session Notes

Give the students a brief overview of some more commonly used patterns that you did not have a chance to get to. Start with any sessions that you may not have gotten to, then move to the patterns below.

Singleton

Problem
Need to ensure a class has only one instance and provide a global point of access to that instance.

Pattern/Solution
Ask the students why doesn't a global variable solve this? A global variable doesn't prevent you from creating multiple instances.

Example
Singleton sample code for projecting to class

Singletons are often 'registered' in a registry, which allows you to change the behaviour of the singleton with a subclass or with a test stub easily. For example you could configure your registry to load test stubs whenever you are running tests.

Benefits/Liabilities
Singletons can be addictive. Joshua Kerviersky coined the term *singletonitis* for the addition to the singleton pattern. It is probably the most overused pattern of all. If you are thinking about implementing a singleton, stop and think really hard:

- Do you truly need a global point of access? Couldn't you pass the singleton object as a parameter instead? Often it turns out that you don't need to pass it around as much as you thought.
- Do you truly need to ensure that you only have one instance of this object in your system?

State

Problem

- Behaviour of an object depends on its state
- Numerous of if/else statements based on the state. (Ex: if you notice a lot of ifs like 'if I am in state B')

Pattern/Solution

- Introduce an object to represent the states. Each state is a different class.
- Where you previously had if statements checking the state, now delegate to the state object. (Along with Strategy, State is one of our if killers)
- State objects manage state transitions as well.
- States are completely encapsulated by their enclosing objects. The outside world does not know they exist.
- States typically
  - Do not have their own state. Instead they modify the enclosing object.
  - Are implemented as anonymous inner classes
  - Are declared as constants in the original object
  - Take the enclosing object as a parameter

Example
A loan object with 2 states: active & inactive
State sample code for projecting to class

Proxy

TBC

Memento

TBC

Overuse of Patterns

> Patterns are easily be overused. Do not add them to code just to try them out or because they are a possible solution. They add complexity and can easily turn what should have been a simple solution into something difficult to understand. Let design patterns naturally show themselves in the code.

Project this code on the screen.

> Up on the screen is an example of design patterns gone wild, taken from slashdot. Take a few minutes and figure out what it does and list all the patterns you find.

The answer is that the program prints *Hello World*

0 comments | Add Comment