Tuesday, October 4

Announcements

- www.singularsource.net
  - Donate to my short story contest
- UCI Delta Sigma Pi
  - Accepts business and ICS students
  - See Facebook page for details

Design Patterns

Design Patterns

- Reusable design component
- First codified by the Gang of Four in 1995
  - Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- Concept taken from architecture
  - “A Pattern Language” by Christopher Alexander
  - “…a three-part rule, which expresses a relation between a certain context, a problem, and a solution.”
- Original Gang of Four book described 23 patterns
  - More have been added
  - Other authors have written books

Design Patterns Template

- Context
  - General situation in which the pattern applies
- Problem
  - The main difficulty being tackled
- Forces
  - Issues or concerns that need to be considered. Includes criteria for evaluating a good solution.
- Solution
  - Recommended way to solve the problem in the context. The solution "balances the forces"
- The following are optional
- Antipatterns
  - Common mistakes to avoid
- Related Patterns
  - Similar patterns; could be alternated solutions or work with the pattern
- References
  - Source of pattern
  - Who developed or inspired the pattern

Gang of Four Design Patterns

- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Patterns in Java

- Chain of Responsibility
  - Exception handling
  - Try/catch/throw blocks
- Iterator
  - Container classes
- Observer
  - Listeners in GUIs

Gang of Four Design Patterns

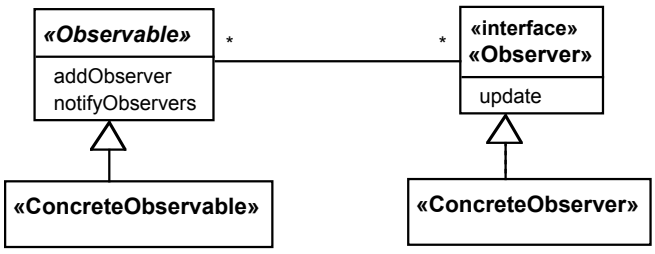
- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - Flyweight
  - Proxy
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

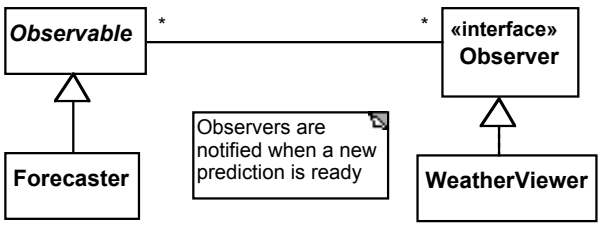
The Observer Pattern

- Context
  - When an association is created between two classes, the code for the classes becomes inseparable.
  - If you want to reuse one class, then you also have to reuse the other.
- Problem
  - How do you reduce the interconnection between classes, especially between classes that belong to different modules or subsystems?
- Forces
  - You want to maximize the flexibility of the system to the greatest extent possible

The Observer Pattern

- Solution





Observer

- Antipatterns (Don't do this)
  - Connect an observer directly to an observable so that they both have references to each other.
  - Make the observers *subclasses* of the observable.
- Reference
  - Gang of Four

Observer in Java

- Observer interface and Observable class exist
  - java.util.Observer and java.util.Observable
- But people usually implement their own
  - Usually can't or don't want to sub-class from Observable
  - Can't have your own class hierarchy and multiple inheritance is not available
  - Has been replaced by the Java Delegation Event Model (DEM)
    - Passes event objects instead of update/notify
- Listener is specific to GUI classes

Thursday, October 4

Announcements

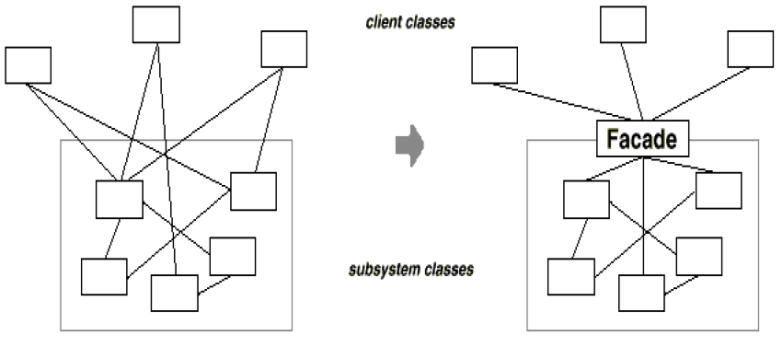
- Tuesday is not Remembrance Day nor Veteran's Day
  - But you still get the day off

The Façade Pattern

- Context
  - Often, an application contains several complex packages.
  - A programmer working with such packages has to manipulate many different classes
- Problem
  - How do you simplify the view that programmers have of a complex package?
- Forces
  - It is hard for a programmer to understand and use an entire subsystem
  - If several different application classes call methods of the complex package, then any modifications made to the package will necessitate a complete review of all these classes.

The Façade Pattern

- Solution



The Façade Pattern

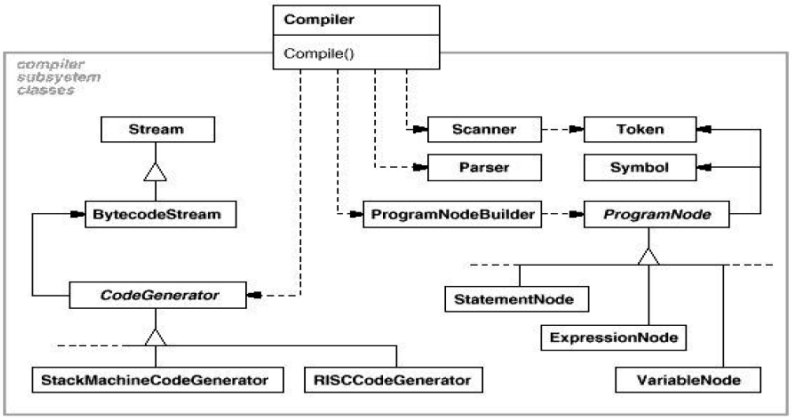
- Solution
  - Provide a simple interface to a complex subsystem.
  - Decouple the classes of the subsystem from its clients and other subsystems, thereby promoting subsystem independence and portability

Using the Façade Pattern

- Hides implementation details
- Promotes weak coupling between the subsystem and its clients.
- Reduces compilation dependencies in large software systems

- Does not add any functionality, it just simplifies interfaces
- Does not prevent clients from accessing the underlying classes.

Façade Example

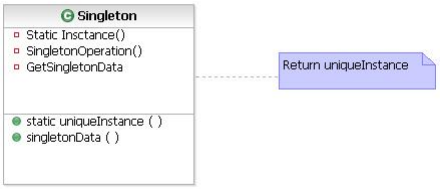


The Singleton Pattern

- Context
  - It is very common to find classes for which only one instance should exist (singleton)
- Problem
  - How do you ensure that it is never possible to create more than one instance of a singleton class?
- Forces
  - The use of a public constructor cannot guarantee that no more than one instance will be created.
  - The singleton instance must also be accessible to all classes that require it

The Singleton Pattern

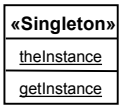
- Solution



Singleton

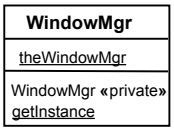
- Example

Pattern



This is the code for getInstance

Instantiation of Pattern




```
if (theWindowMgr==null)
  theWindowMgr= new WindowMgr();

return theWindowMgr;
```

Constructor for WindowMgr is private
getInstance is public and static
theWindowMgr is private and static

Singleton Design Pattern

```
public class WindowMgr {
    private static WindowMgr theWindowMgr;
    private String windowLabel;

    private WindowMgr (){
    }

    // Lazy instantiation
    public static synchronized WindowMgr getInstance(){
        if (theWindowMgr == null){
            theWindowMgr = new WindowMgr();
        }
        return theWindowMgr;
    }

...
}
```


Singleton Design Pattern

```
public class WindowMgr {
    // Eager instantiation
    private static WindowMgr theWindowMgr = new WindowMgr();
    private String windowLabel;

    private WindowMgr (){
   }

    public static synchronized WindowMgr getInstance(){
        return theWindowMgr;
    }

...
}
```

Questions

- Why do you need the getInstance method? Why isn't it enough to just make theWindowMgr static (i.e. one per class)?
  - This results in extra instances of WindowMgr, but still only one underlying theWindowMgr
- Why do you need an instance of WindowMgr at all? Why not just make all the methods static?
  - May need an instance, e.g. as an observer, for callbacks
  - More flexible when you discover later that you don't want WindowMgr to be a singleton any more

Drawbacks

- Need to add synchronization to getInstance
  - Race condition could occur in if block
- Sub-classing becomes complicated
  - Private constructor violates normal Java design principles
  - Could change constructor to protected, but that would violate the security provided
    - Make a sub-class that is identical to parent
    - Can have lots of pseudo-WindowMgrs running around
  - Alternatively, each sub-class has own getInstance method
- Also need to prevent cloning by overriding Cloneable interface
- Erich Gamma doesn't like Singleton any more

Singleton Design Pattern

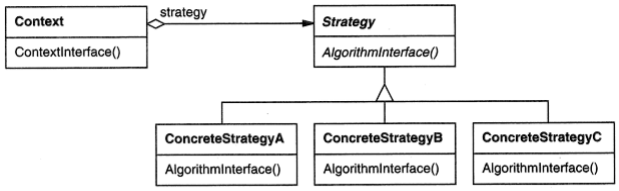
- Related Patterns
  - Factory and Façade
- Reference
  - Gang of Four

Strategy Design Pattern

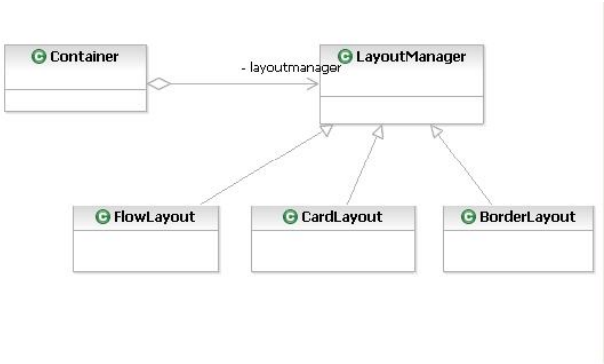
- Context
  - Define a family of algorithms, so they are interchangeable.
- Also Known As
  - Policy
- Problem
  - How to design for varying, but related algorithms or policies? How to design for the ability to change the algorithms or policies?
- Solution
  - Define each algorithm/policy/strategy in a separate class with a common interface

Strategy Design Pattern

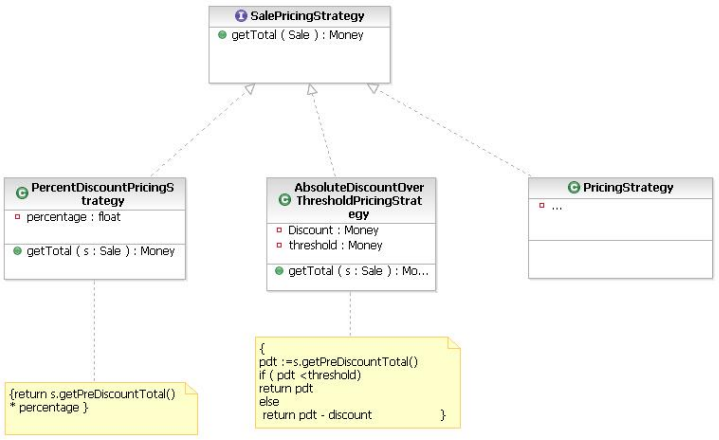
- Structure



Example



Example



Strategy Design Pattern

- Participants
  - Strategy interface, concrete Strategy, and Context/client
- Consequences
  - Provides an alternative to subclassing the Context class to get a variety of algorithms or behaviors
  - Eliminates large conditional statements
  - Provides a choice of implementations for the same behavior
  - Increases the number of objects
  - All algorithms must use the same Strategy interface
- Implementation
  - Can use an Abstract Factory to create a Strategy

The Importance of Ignorance

Context

- Everyone is ignorant about something-- often many things
- The first step in becoming an intelligent ignoramus is to understand ignorance

Armour's Orders of Ignorance

- Zeroth Order Ignorance (0OI): Lack of ignorance.
  - I have 0OI when I provably know something.
- First Order Ignorance (1OI): Lack of knowledge.
  - I have 1OI when I do not know something.
- Second Order Ignorance (2OI): Lack of awareness.
  - I have 2OI when I do not know that I do not know something.
- Third Order Ignorance (3OI): Lack of Process.
  - I have 3OI when I do not know of a suitably efficient way to find out that I do not know that I do not know something.
- Fourth Order Ignorance (4OI): Meta ignorance.
  - I have 4OI when I do not know about the Five Orders of Ignorance.

Zeroth Order Ignorance (0OI)

- Lack of Ignorance
  - I have Zeroth Order Ignorance (0OI) when I know something and can demonstrate my lack of ignorance in some tangible form.
  - 0OI is provable and proven knowledge that is deemed "correct" by some qualified agency. In software this means that the knowledge is invariably factored into usable form. In all forms of knowledge there must be some external "proof" element that qualifies the knowledge as being correct.
  - Examples
    - Trivia
    - Building a system that satisfies the user
    - Ability to sail

First Order Ignorance (1OI)

- Lack of Knowledge
  - I have First Order Ignorance (1OI) when I do not know something and I can readily identify that fact. 1OI is basic ignorance or lack of knowledge.
  - Example
    - Speaking Russian

Second Order Ignorance (2OI)

- Lack of Awareness
  - I have Second Order Ignorance (2OI) when I do not know that I do not know something. That is to say, not only am I ignorant of something (I have 1OI), I am unaware of what it is I am ignorant about. I do not know enough to know what it is that I do not know.
  - Example
    - I cannot give a good example of 2OI, of course.
    - Do you need a will or a trust?

Third Order Ignorance (3OI)

- Lack of Process
  - I have Third Order Ignorance (3OI) when I do not know of a suitably efficient way to find out that I do not know that I do not know something, which is lack of a suitable knowledge-gathering process.
  - This presents me with a major problem: If I have 3OI, I do not know of a way to find out that there are things that I do not know that I do not know. Therefore, I cannot change those things that I do not know that I do not know into either things that I know, or at least things that I know that I do not know, as a step toward converting the things that I know that I do not know into things that I know.
  - Example
    - Design
    - Investigative journalism

Fourth Order Ignorance (4OI):

- Meta Ignorance
  - I have Fourth Order Ignorance (4OI) when I do not know about the Five Orders of Ignorance.
  - I do not have this kind of ignorance, and now neither do you.
  - Knowledge is highly and intrinsically recursive-- to know about anything, you must first know about other things which define what you know.

Asking Questions

- Reveals
  - Ignorance
  - Intelligence
- Reduces
  - Ignorance
  - Assumptions
- Examples
  - Richard Feynman
  - Dan Berry