Agenda

- Lecture
  - Design Patterns
  - UML

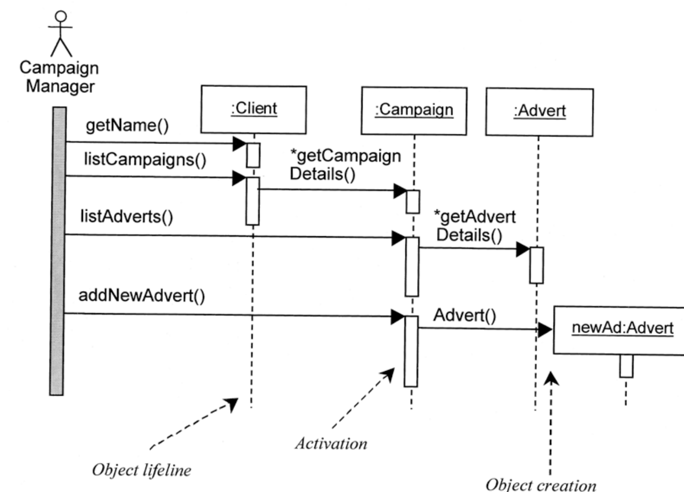
Sequence Diagram of Observer

- Shows runtime interactions

Sequence Diagrams

- Sequence diagrams provide a more detailed look of the sequence of steps executed in a use case
  - Normally used for lower-level design
  - If you wanted to specify all of your application's scenarios with sequence diagrams, you would need one for each of its features' ramifications
    - So we are usually interested in key scenarios only
- Sequence diagrams show:
  - The actors and software classes/objects that intervene in the scenario
  - The step-by-step interactions between them
    - Chronologically, from top to bottom
  - Details regarding when objects are created and activated

Sequence Diagrams (cont)



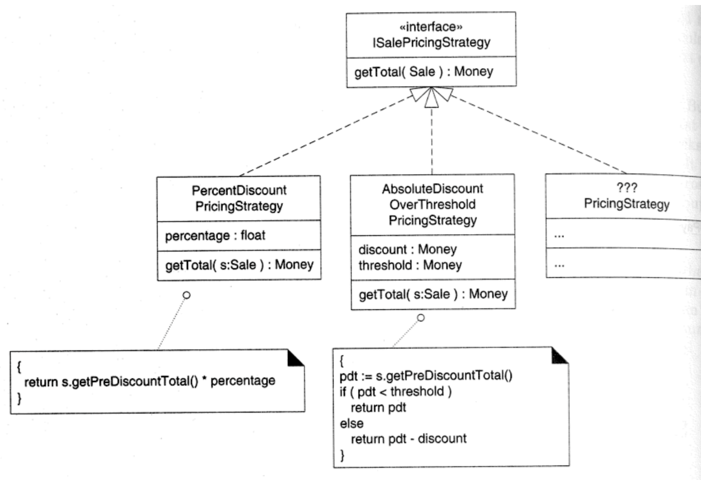
Sequence Diagrams (cont)

- This is not the full story
  - We can illustrate branching, *guards* (conditions necessary for the execution of a call), asynchronous messaging, and more
  - In UML 2.0, sequence diagrams went through a major overhaul
    - Conditionals, loops, etc.
- We don't need the full story for this course
  - These basics are enough
  - But if you want to invest time in learning more about UML, sequence diagrams are the place to start
    - Along with class diagrams, they are the most frequently used kind of model

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

Example



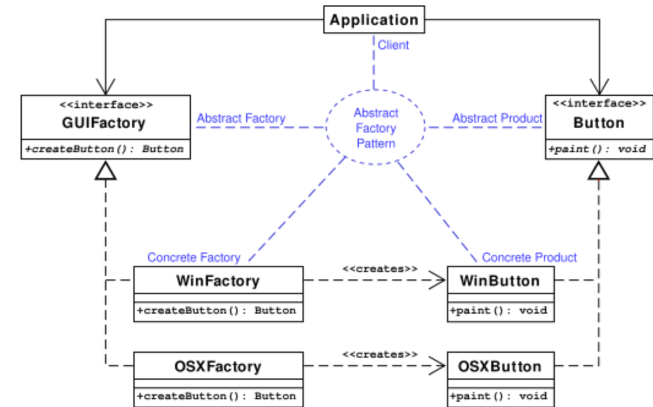
Strategy Design Pattern

- 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

Abstract Factory

- Context
  - Related classes that implement a common interface
- Problem
  - Need to encapsulate the instantiation of the related classes
- Forces
  - Information hiding
  - Keep related classes together
- Solution
  - Define a factory interface (the abstract factory). Define a concrete factory class for each family of things to create.
  - Optionally, define a true abstract class that implements the factory interface and provides common services to the concrete factories that extend it.

- Structure



Abstract Factory (cont.)

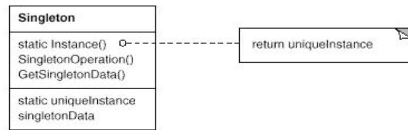
- Participants
  - Abstract Factory
  - Concrete Factory
  - Abstract Product
  - Concrete Product
  - Client
- Consequences
  - Isolates concrete classes.
  - Simplifies exchanging families
  - Promotes consistency
  - Supporting new kinds of products is difficult

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

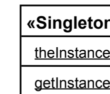
- Solution



Singleton

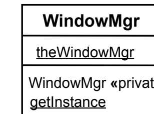
- Example

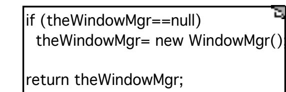
Pattern



This is the code for getInstance

Instantiation of Pattern





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

What is UML and why should I care?

- The Unified Modeling Language is an industry standard for specifying and visualizing the artifacts of software systems
  - A collection of diagrammatic languages to express everything from class structures to execution scenarios
  - A joint effort by object-oriented modeling researchers to merge their different approaches
    - James Rumbaugh, Grady Booch, Ivar Jacobson
    - UML 1.0 came out in 1997
    - Current version, UML 2.0
    - http://www.uml.org/
- If there is one modeling language that you need to know to get a job, this is it
  - Although frankly you may not need to use it once you get that job
  - If "Model-Driven Development" takes off, you will need this
- Easy to learn the basics, very hard to master it
  - Especially the newest version
  - For now all you need are those easy-to-learn basics

The many diagrams of UML