Lecture 6: Design Patterns

1. Introduction
2. Creational Patterns
3. Structural Patterns
4. Behavioral Patterns

Design Patterns

- Design patterns constitute a set of rules describing how to accomplish certain task in the realm of software development.
- A pattern describes a problem, which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without doing it the same way twice.
- Patterns identify and specify abstractions that are above the level of single classes and objects, or of components.
- Design patterns are not just about the design of objects, but also about the communication between objects. It is the design of simple, but elegant, methods of communication that makes many design patterns so important.

- Design patterns can exist at many levels from very low-level specific solutions to broadly generalized system issues.
- The researchers collected 23 of the most commonly used general purpose design patterns that are application domain independent and classified them in three categories:

  Creational Patterns
  Structural Patterns
  Behavioral patterns

Resources:

Design Patterns: Elements of Reusable OO Software
Erich Gamma,Richard Helm,Ralph Johnson, John Vlissides;
Addison-Wesley, 1995

Creational Patterns

- Deal with the best way to create instances of objects.
- Create objects at run-time not at compile-time
- Include:
  1. The factory pattern
  2. The abstract factory pattern
  3. The builder pattern
  4. The prototype pattern
  5. The singleton pattern

Structural Patterns

- Structural patterns describe how classes and objects can be combined to form larger structures.
- Structural patterns include:
    1. **Adapter** makes things work after they are designed
    2. **Bridge** makes things work before they are
    3. **Composite** is a composition of objects
    4. **Decorator** provides an enhanced interface
    5. **Facade** represents an entire subsystem
    6. **Flyweight** is a pattern for sharing objects
    7. **Proxy** represents another object

Behavioral Patterns

- Behavioral patterns are concerned with the assignment of responsibilities between objects, or, encapsulating behavior in an object and delegating requests to it.
- Behavioral patterns include:
    1. **Observer** defines the way a number of classes can be notified of a change.
    2. **Mediator** uses a class to simplify communication between other classes
    3. **Memento** provides a way to capture an object's state

4. **Chain of responsibility** allows a number of classes to attempt to handle a request
5. **Template** provides an abstract definition of an algorithm
6. **Interpreter** provides a definition of how to include language elements in a program
7. **Strategy** encapsulates an algorithm inside a class
8. **Visitor** adds functions to a class
9. **State** provides a memory for a class 's instance variables
10. **Command** encapsulates an action as an object
11. **Iterator** formalizes the way we move through a list of data

---

The Factory Pattern

- Intent
  Define a method that returns one of several possible subclasses of an abstract class depending on the data that are provided.
- Problem
  A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.

The Abstract Factory Pattern

- Intent
  Provides an interface to create and returns one of several families of related objects without specifying their classes directly.
- Problem
  If an application is to be portable, it needs to encapsulate platform dependencies. These "platforms" might include: windowing system, operating system, database, etc. Too often, this encapsulation is not engineered in advance.

The Builder Pattern

- Intent
  Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Problem
  An application needs to create the elements of a complex aggregate. The specification for the aggregate exists on secondary storage and one of many representations needs to be built in primary storage.

The Prototype Pattern

- Intent
  Specify the kinds of objects to create using a

prototypical instance, and create new objects by copying this prototype.

The Singleton Pattern

- Intent
  Ensure a class has only one instance, and provide a global point of access to it.
- Problem
  Application needs one, and only one, instance of an object. Additionally, lazy initialization and global access are necessary.
- Example
  One cannot elect two presidents, the presidency has only one instance.

```
public class Presidency
{
  public static boolean instance_flag = false; //true if 1 instance

  private Presidency() { }
  public static Presidency election() {
    if (! instance_flag)  {
      instance_flag = true;
      return new Presidency();
    }
    else return null;
  }
  //----------------------------------------
  public void finalize() {
    instance_flag = false;
  }
}
```

The Adapter Pattern

- Intent
  Convert the interface of a class into another interface clients expect.
- Problem
  An "off the shelf" component offers compelling functionality that you would like reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.
- Example
  Java use the adapter pattern to implement a simple class that can be used to encapsulate a number of events. They include ComponentAdapter, ContainerAdapter, FocusAdapter, KeyAdapter, MouseAdapter, MouseMotionAdapter and WindowAdapter.

The Bridge Pattern

- Intent
  Decouple an abstraction from its implementation so that the two can vary independently.
- Problem
  "Hardening of the software arteries" has occurred by using subclassing of an abstract base class to

provide alternative implementations. This locks in compile-time binding between interface and implementation. The abstraction and implementation cannot be independently extended or composed.

The Composite Pattern

- Intent
  Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- Problem
  Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

The Decorator Pattern

- Intent
  Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

- Problem
  You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.
- Example
  The decorator pattern provides us a way to modify the behavior of individual objects without having to create a new derived class.

The Facade Pattern

- Intent
  Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Problem
  A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

The Flyweight Pattern

- Intent
  Use sharing to support large numbers of fine-grained objects efficiently.
- Problem
  Designing objects down to the lowest levels of system "granularity" provides optimal flexibility,

but can be unacceptably expensive in terms of performance and memory usage.

- <u>Example</u>
  You can reduce the number of different instances of a class by moving some variables outside the class and pass them in as part of a method call. Flyweights are sharable instance of a class.

The Proxy Pattern

- <u>Intent</u>
  Provide a surrogate or placeholder for another object to control access to it.
- <u>Problem</u>
  You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.
- <u>Example</u>
  When an image needs time to be loaded, you can make a proxy object that displays the area you want to display the image and write a message "Image is loading ..." until the image is loaded completely.

The Chain of Responsibility Pattern

- <u>Intent</u>
  Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and

pass the request along the chain until an object handles it.

- Problem
  There is a potentially variable number of "handler" objects and a stream of requests that must be handled. Need to efficiently process the requests without hard-wiring handler relationships and precedence, or request-to-handler mappings.

The Command Pattern

- Intent
  Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.
- Problem
  Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

The Interpreter Pattern

- Intent
  Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

- Problem
  A class of problems occurs repeatedly in a well-defined and well-understood domain. If the domain were characterized with a "language", then problems could be easily solved with an interpretation "engine".

The Iterator Pattern

- Intent
  Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Problem
  Need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently.
- Example : Find elements by going through a list

The Mediator Pattern

- Intent
  Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Problem
  We want to design reusable components, but

dependencies between the potentially reusable pieces demonstrates the "spaghetti code" phenomenon (trying to scoop a single serving results in an "all or nothing clump").

The Memento Pattern

- Intent
  Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later.
- Problem
  Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations).
- Example : Undo an action

The Observer Pattern

- Intent
  Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- Problem
  A large monolithic design does not scale well as new graphing or monitoring requirements are levied.

The State Pattern

- Intent
  Allow an object to alter its behavior when its

internal state changes. The object will appear to change its class.

- Problem
  A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterixed by large and numerous case statements that vector flow of control based on the state of the application.
- Example : switch a button on and off

The Strategy Pattern

- Intent
  Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.
- Problem
  If clients have potentially generic algorithms embedded in them, it is difficult to: reuse these algorithms, exchange algorithms, decouple different layers of functionality, and vary your choice of policy at run-time. These embedded policy mechanisms routinely manifest themselves as multiple, monolithic, conditional expressions.
- Example : A Strategy for drawing

The Template Pattern

- Intent
  Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Problem
  Two different components have significant similarities, but demonstrate no reuse of common interface or implementation. If a change common to both components becomes necessary, duplicate effort must be expended.

The Visitor Pattern

- Intent
  Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Problem
  Many distinct and unrelated operations need to be performed on node objects in a heterogeneous aggregate structure. You want to avoid "polluting" the node classes with these operations. And, you don't want to have to query the type of each node and cast the pointer to the correct type before performing the desired operation.