Design Patterns

SE3A04 – Tutorial

Jason Jaskolka

Department of Computing and Software
Faculty of Engineering
McMaster University
Hamilton, Ontario, Canada
`jaskolj@mcmaster.ca`

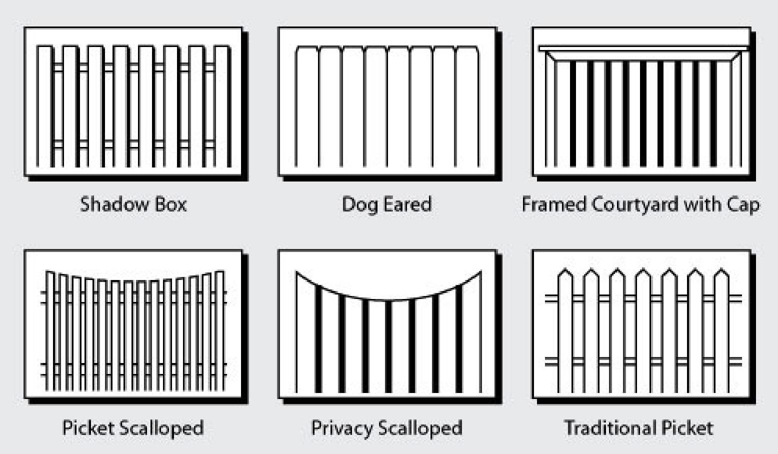
November 13, 2012

Outline

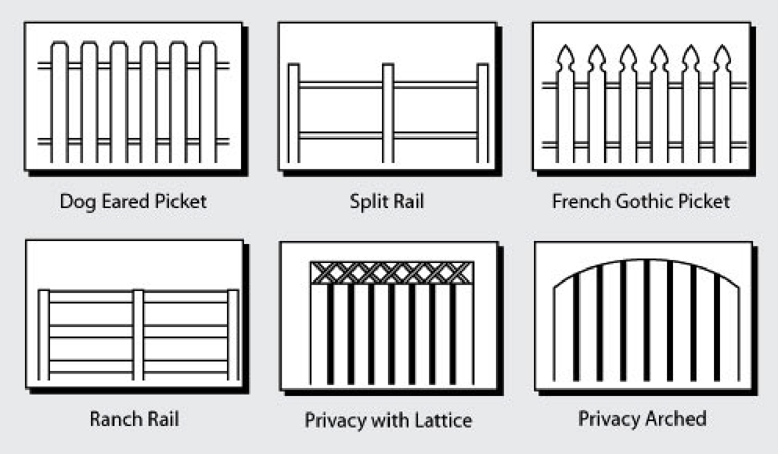
Design Patterns

- Suppose that you wish to build a wooden fence in your yard
  - What kind of fence are you going to build?
- The choice may depend on the purpose of the fence
  - Perhaps you want privacy
  - Perhaps you want the fence to give the house a particular look and feel
  - The list can go on and on . . .
- For fences, there are typical design patterns which can be used to meet different requirements or to fit a particular purpose

Design Patterns



Design Patterns



Design Patterns

- In a similar example, an architect has a collection of design patterns in order to meet particular requirements of to fit different purposes for buildings
- Numerous other design tasks comes with a collection of standard design patterns to aid in fitting a purpose and meeting requirements
- Why should software design be any different?

Design Patterns

Definition (Design Pattern)

A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design.

- It is not a finished design that can be transformed directly into code
- It is a description or template for how to solve a problem that can be used in many different situations
- Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved

Creational Patterns

Definition (Creational Design Pattern)

Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

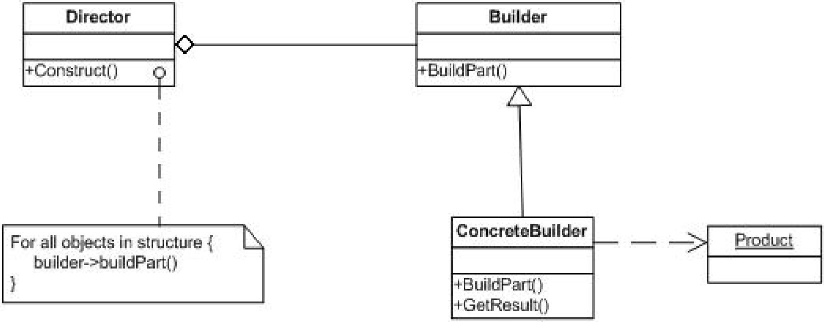
List of Some Creational Patterns

- **Abstract Factory pattern**: centralizes the decision of what factory to instantiate
- **Factory Method pattern**: centralizes the creation of an object of a specific type by choosing one of several implementations
- **Builder pattern**: separates the construction of a complex object from its representation so that the same construction process can create different representations
- **Lazy Initialization pattern**: delays the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed

List of Some Creational Patterns

- **Object Pool pattern**: avoids expensive acquisition and release of resources by recycling objects that are no longer in use
- **Prototype pattern**: used when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects
- **Singleton pattern**: restricts instantiation of a class to one object

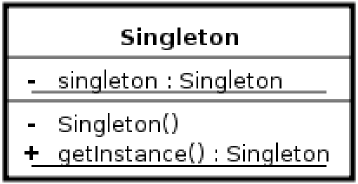
Builder Pattern



Builder Pattern

- **Builder** is an abstract interface for creating objects (product)
- **ConcreteBuilder** provides the implementation for **Builder**; an object able to construct other objects
- **Director** is responsible for managing the correct sequence of object creation; receives a **ConcreteBuilder** as a parameter and executes the necessary operations on it
- **Product** is the final object that will be created by the **Director** using **Builder**

Singleton Pattern



- Create a class with a method that creates a new instance of the class if one does not exist
- If an instance already exists, simply return a reference to that object
- To make sure that the object cannot be instantiated any other way, the constructor is made private

Structural Patterns

Definition (Structural Design Pattern)

Structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

List of Some Structural Patterns

- **Adaptor pattern**: "adapts" one interface for a class into one that a client expects
- **Aggregate pattern**: provides a version of the Composite pattern with methods for aggregation of children
- **Bridge pattern**: decouples an abstraction from its implementation so that the two can vary independently
- **Composite pattern**: provides a tree structure of objects where every object has the same interface

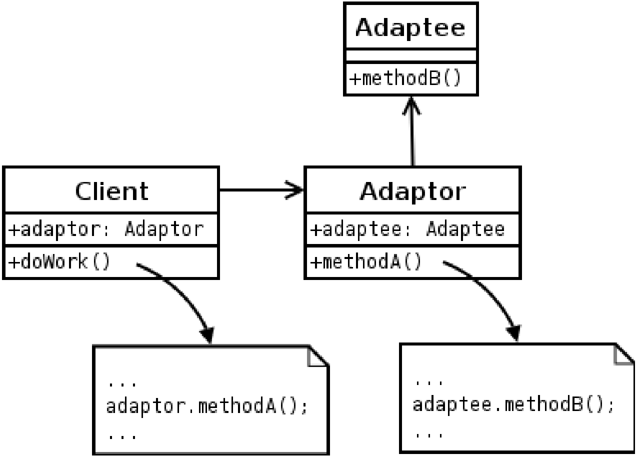
List of Some Structural Patterns

- **Decorator pattern**: adds additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes
- **Extensibility pattern**: hides complex code behind a simple interface
- **Facade pattern**: creates a simplified interface of an existing interface to ease usage for common tasks
- **Flyweight pattern**: allows a high quantity of objects to share a common properties object to save space

List of Some Structural Patterns

- **Proxy pattern**: provides a class functioning as an interface to another thing
- **Pipe and Filter pattern**: provides a chain of processes where the output of each process is the input of the next
- **Private Class Data pattern**: restricts accessor/mutator access

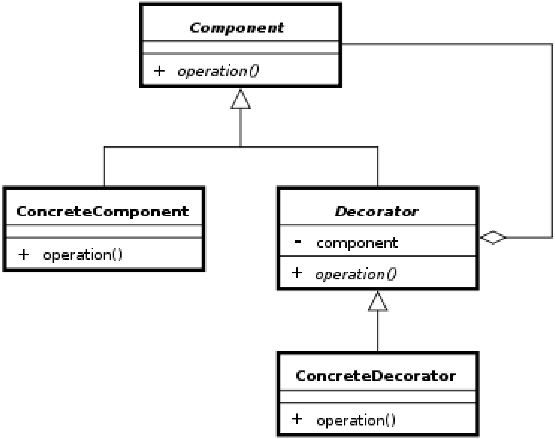
Adaptor Pattern



Adaptor Pattern

- The adaptor translates calls to its interface into calls to the original interface
- The amount of code necessary to do this translation is typically small
- The adaptor is also responsible for transforming data into appropriate forms
    - **Example**: transforming the format of dates from YYYYMMDD to MM/DD/YYYY or DD/MM/YYYY

Decorator Pattern



Decorator Pattern

- Design a new **Decorator** class that wraps the original class by doing the following:
  1. Subclass the original **Decorator** class into a **Component** class
  2. In the **Decorator** class, add a **Component** pointer as a field;
  3. Pass a **Component** to the **Decorator** constructor to initialize the **Component** pointer
  4. In the **Decorator** class, redirect all **Component** methods to the **Component** pointer
  5. In the **ConcreteDecorator** class, override any **Component** method(s) whose behaviour needs to be modified

Behavioural Patterns

Definition (Behavioural Design Pattern)

Behavioural design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

List of Some Behavioural Patterns

- **Chain of Responsibility pattern**: allows command objects to be handled or passed on to other objects by logic-containing processing objects
- **Command pattern**: used when command objects encapsulate an action and its parameters
- **Interpreter pattern**: implements a specialized computer language to rapidly solve a specific set of problems
- **Iterator pattern**: used to access the elements of an aggregate object sequentially without exposing its underlying representation

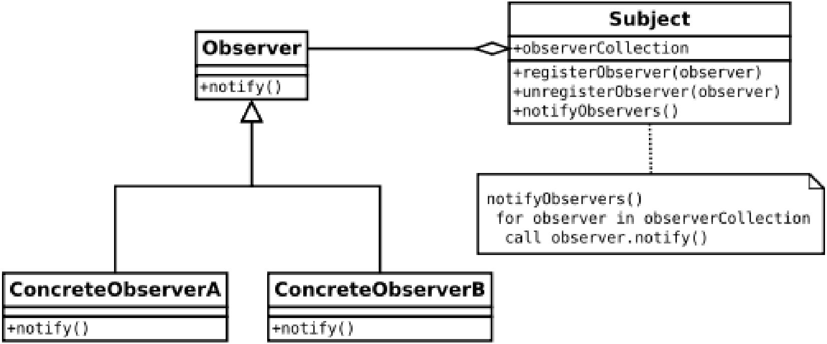
List of Some Behavioural Patterns

- **Mediator pattern**: provides a unified interface to a set of interfaces in a subsystem
- **Memento pattern**: provides the ability to restore an object to its previous state (rollback)
- **Null Object pattern**: designed to act as a default value of an object
- **Observer pattern**: objects register to observe an event which may be raised by another object
- **Protocol Stack pattern**: communications are handled by multiple layers, which form an encapsulation hierarchy

List of Some Behavioural Patterns

- **State pattern**: provides a clean way for an object to partially change its type at runtime
- **Strategy pattern**: allows for algorithms to be selected on the fly
- **Specification pattern**: allows for recombinable business logic in a boolean fashion
- **Template Method pattern**: describes the program skeleton of a program
- **Visitor pattern**: provides a way to separate an algorithm from an object
- **Scheduled-Task pattern**: allows a task to be scheduled to be performed at a particular interval or clock time (used in real-time computing)

Observer Pattern

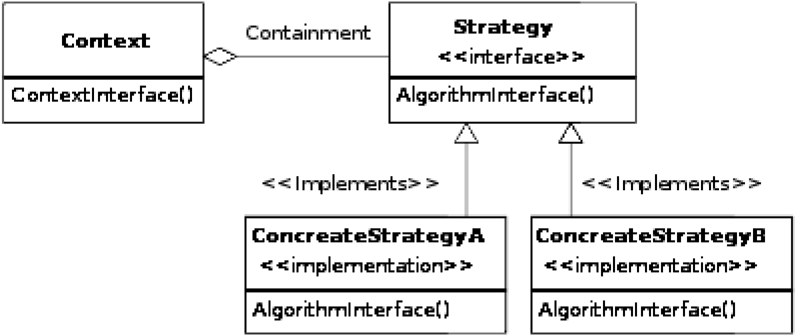


```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

Observer Pattern

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Mainly used to implement distributed event handling systems

Strategy Pattern



Strategy Pattern

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Strategy lets the algorithm vary independently from clients that use it
    - **Example**: Validating incoming data

Concurrency Patterns

Definition (Concurrency Design Pattern)

Concurrency design patterns are those types of design patterns that deal with the multi-threaded programming paradigm.

List of Some Concurrency Patterns

- **Active Object pattern**: decouples method execution from method invocation that reside in their own thread of control; introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests
- **Balking pattern**: only executes an action on an object when the object is in a particular state
- **Binding Properties pattern**: combines multiple observers to force properties in different objects to be synchronized or coordinated in some way

List of Some Concurrency Patterns

- **Double-Checked Locking pattern**: reduces the overhead of acquiring a lock by first testing the locking criterion (the ‘lock hint’) in an unsafe manner; only if that succeeds does the actual lock proceed
- **Guarded Suspension pattern**: manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed
- **Lock pattern**: allows one thread to put a “lock” on a resource, preventing other threads from accessing or modifying it

List of Some Concurrency Patterns

- **Monitor Object pattern**: provides an object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time
- **Reactor pattern**: provides an asynchronous interface to resources that must be handled synchronously
- **Read-Write Lock pattern**: allows concurrent read access to an object, but requires exclusive access for write operations
- **Scheduler pattern**: explicitly controls when threads may execute single-threaded code

List of Some Concurrency Patterns

- **Thread Pool pattern**: allows for a number of threads to be created to perform a number of tasks, which are usually organized in a queue; typically, there are many more tasks than threads
- **Thread-Specific Storage pattern**: used when static or "global" memory is local to a thread

Questions

- Questions?