Proxy Pattern: Motivation

- It is 15:00pm. I am sitting at my 256 kbps connection and retrieve a fancy web site from the US. This is prime web time all over the US. So I am getting 8 bits/sec.
- What can you do?

Proxy Pattern

- What is expensive?
  - Object Creation
  - Object Initialization
- Defer object creation and object initialization to the time you need the object
- Proxy pattern:
  - Reduces the cost of accessing objects
  - Uses another object ("the proxy") that acts as a stand-in for the real object
  - The proxy creates the real object only if the user asks for it

Proxy Pattern (207)

- Interface inheritance is used to specify the interface shared by Proxy and RealSubject.
- Delegation is used to catch and forward any accesses to the RealSubject (if desired)
- Proxy patterns can be used for lazy evaluation and for remote invocation.
- Proxy patterns can be implemented with a Java interface.

Proxy Applicability

- Remote Proxy
  - Local representative for an object in a different address space
  - Caching of information: Good if information does not change too often
- Virtual Proxy
  - Object is too expensive to create or too expensive to download
  - Proxy is a stand-in
- Protection Proxy
  - Proxy provides access control to the real object
  - Useful when different objects should have different access and viewing rights for the same document,
  - Example: Grade information for a student shared by administrators, teachers and students.

Virtual Proxy example

- Images are stored and loaded separately from text
- If a RealImage is not loaded a ProxyImage displays a grey rectangle in place of the image
- The client cannot tell that it is dealing with a ProxyImage instead of a RealImage
- A proxy pattern can be easily combined with a Bridge
Towards a Pattern Taxonomy

- **Structural Patterns**
  - Adapters, Bridges, Façades, and Proxies are variations on a single theme:
    - They reduce the coupling between two or more classes
    - They introduce an abstract class to enable future extensions
    - Encapsulate complex structures
  - Concerned with algorithms and the assignment of responsibilities between objects: Who does what?
    - Characterize complex control flow that is difficult to follow at runtime.

- **Behavioural Patterns**
  - Concerned with algorithms and the assignment of responsibilities between objects: Who does what?
  - Characterize complex control flow that is difficult to follow at runtime.

- **Creational Patterns**
  - Abstract the instantiation process.
  - Make a system independent from the way its objects are created, composed and represented.

Observer pattern (293)

- “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.” (p. 293)
- Also called “Publish and Subscribe”

- Uses:
  - Maintaining consistency across redundant state
  - Optimizing batch changes to maintain consistency
Observer pattern (continued)

- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observers vector) not an interface.

Observer pattern implementation in Java

```java
// import java.util;
public class Observable extends Object {
    public void addObserver(Observer o);
    public void deleteObserver(Observer o);
    public boolean hasChanged();
    public void notifyObservers();
    public void notifyObservers(Object arg);
}
public abstract interface Observer {
    public abstract void update(Observable o, Object arg);
}
public class Subject extends Observable{
    public void setState(String filename);
    public String getState();
}
```

Abstract Factory Motivation

- Consider a user interface toolkit that supports multiple looks and feel standards such as Motif, Windows 95 or the finder in MacOS.
  - How can you write a single user interface and make it portable across the different look and feel standards for these window managers?
- Consider a facility management system for an intelligent house that supports different control systems such as Siemens’ Instabus, Johnson & Control Metasys or Zumtobe’s proprietary standard.
  - How can you write a single control system that is independent from the manufacturer?

Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation:
  - The system should be independent of how its products are created, composed or represented
- Manufacturer Independence:
  - A system should be configured with one of multiple family of products
  - You want to provide a class library for a customer (“facility management library”), but you don’t want to reveal what particular product you are using.
- Constraints on related products
  - A family of related products is designed to be used together and you need to enforce this constraint
- Cope with upcoming change:
  - You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.
The Singleton Pattern (127)

- The Singleton Pattern is used to ensure a class has only one instance and provide a global access point to it. [Gamma et al 95]

- If we write a class to manage a file system, we do not want someone to make multiple instances of the class, as changes in the file system may not be properly recorded.

- In Java a Singleton uses static operations and access modifiers to prevent clients from making multiple instances of the class.

**Singleton in Java**

```java
public class Singleton{
    // declare the unique instance of the class
    private static Singleton unique = new Singleton();

    //private constructor can only be access from this class
    private Single(){//Constructor code here}

    //public method to return a reference to the unique instance of the class
    public Single getInstance(){
        return unique;
    }
}
```

**Summary**

- **Structural Patterns**
  - Focus: How objects are composed to form larger structures
  - Problems solved:
    - To realize new functionality from old functionality,
    - To provide flexibility and extensibility

- **Behavioral Patterns**
  - Focus: Algorithms and the assignment of responsibilities to objects
  - Problem solved:
    - Too tight coupling to a particular algorithm

- **Creational Patterns**
  - Focus: Creation of complex objects
  - Problem solved:
    - Hide how complex objects are created and put together

**Other Design Patterns**

- **Builder (97)** - Separate the construction of a complex object from its representation so the same process can create different representations.
- **Flyweight (195)** - Use sharing to support large numbers of fine-grained objects efficiently.
- **Command (223)** - Encapsulate requests as objects, allowing you to treat them uniformly.
- **Iterator (257)** - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- **State (305)** - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

**Conclusion**

- **Design Patterns**
  - Provide solutions to common problems.
  - Lead to extensible models and code.
  - Can be used as is or as examples of interface inheritance and delegation.
  - Apply the same principles to structure and to behavior.

- Design patterns solve all your software engineering problems???

- Reading and studying design patterns will give you a library of solutions, and an awareness of consequences in object oriented software design.