Review of last class

1. Class inheritance versus interface inheritance

2. Inheritance versus composition
# Design patterns

<table>
<thead>
<tr>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Factory</td>
<td>Adapter</td>
<td>Command</td>
</tr>
<tr>
<td>Builder</td>
<td>Bridge</td>
<td>Iterator</td>
</tr>
<tr>
<td>Prototype</td>
<td>Composite</td>
<td>Observer</td>
</tr>
<tr>
<td>Singleton</td>
<td>Façade</td>
<td>Strategy</td>
</tr>
<tr>
<td>etc.</td>
<td>Decorator</td>
<td>Interpreter</td>
</tr>
<tr>
<td>etc.</td>
<td>Proxy</td>
<td>Mediator</td>
</tr>
<tr>
<td>etc.</td>
<td>etc.</td>
<td>etc.</td>
</tr>
</tbody>
</table>
Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Proxy
Problem 1

- **DrawingEditor**
  - **Shape**
    - `BoundingBox()`
    - `Createmanipulator()`
  - **Line**
    - `BoundingBox()`
    - `Createmanipulator()`
  - **PolygonShape**
    - `BoundingBox()`
    - `Createmanipulator()`
  - **RevisedTextView**
    - `BoundingBox()`
    - `Createmanipulator()`
    - `getExtent()`

- **TextView**
  - `getExtent()`

---

Dr. Xiaolin Hu
Problem 1

Another way?
Problem 1

- DrawingEditor
  - Shape
    - BoundingBox()
    - Createmanipulator()
  - TextView
    - getExtent()
  - Line
    - BoundingBox()
    - Createmanipulator()
  - PolygonShape
    - BoundingBox()
    - Createmanipulator()
  - TextShape
    - BoundingBox()
    - Createmanipulator()
    - Return getExtent()
  - Return new TextManipulator
Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces.

Also known as wrapper
Delegation is used to bind an **Adapter** and an **Adaptee**.

- Interface inheritance is used to specify the interface of the **Adapter** class.
- **Target** and **Adaptee** (usually called legacy system) pre-exist the Adapter.
- **Target** may be realized as an interface in Java.
A class adapter uses multiple inheritance to adapt one interface to another.

Not as flexible as object adapter – cannot work with multiple adaptees.

Let adapter to override some of adaptee’s behavior. Object adapter cannot override adaptee’s behavior.
Collaborations

- Sequence diagram.
- How much adapting does Adapter do?
  - Range from simple interface conversion to supporting an entirely different set of operations.
Applicability

Use the Adapter pattern when

• You want to use an existing class, and its interface does not match the one you need

• You want to create a reusable class that cooperates with unrelated or unforeseen classes

• (object adapter only) You need to use several existing subclasses, but it’s impractical to adapt their interface by subclassing every one.
Sample code

P 146
Problem 2

- It is inconvenient to extend the window abstraction to cover different kinds of windows.
- It makes client code platform-dependent. For example, creating an MS_Window object binds the Window abstraction to the MS Window implementation.
- Clients should be able to create a window without committing to a concrete implementation.
All operations on Window subclasses are implemented in terms of abstract operations from the WindowImp interface.
Bridge

Decouple an abstraction from its implementation so that the two can vary independently

Also known as Handle/Body
Applicability

Use the bridge pattern when

• You want to avoid a permanent binding between an abstraction and its implementation.
• Both the abstractions and their implementations should be extensible by subclassing.
• Changes in the implementation of an abstraction class should have no impact on clients; that is, their code should not have to be recompiled.
• You want to hide the implementation of an abstraction completely from clients.
• Nested generalizations. Example: consider a game design, robot (behavior) – fighters, worker,… robot (type) – flying robot, walking robot, stationary robot…. 
The Robot Example
Consequences

• Decoupling interface and implementation – the implementation of an abstraction can be configured at run-time.
  – This decoupling also encourages layering that can lead to a better structured system.

• Improved extensibility – you can extend the abstraction and implementor hierarchies independently

• Hiding implementation details from clients
Some Implementation Issues

• Only one implementor – the abstract implementor class is not necessary

• Creating the right implementor object
  – Instantiate one of them in its constructor (such as based on parameters passed to the constructor)
  – Choose a default implementation initially and change it later according to usage
  – Delegate the decision to another object, such as a factory object.
Sample code

P. 156

```
Diagram:

- Window
  - Icon_Window
  - Application_Window
  - PM_Window
  - X_Window
- WindowImp
```
Problem 3

- In graphics applications, the user can group components to form larger components.

- A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

- Consider the example of a group of figures in PPT.

- Problem: Code that uses these classes must treat primitives and container objects differently, even if most of the time the user treats them identically.
The key to the composite pattern is an abstract class that represents both primitives and their containers.
Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
A sample composite object diagram, page 165
A sample composite object diagram
Other examples

• Software System:
  – Definition: A software system consists of subsystems which are either other subsystems or collection of classes

• Software Lifecycle:
  – Definition: The software lifecycle consists of a set of development activities which are either other activities or collection of tasks

• File Folder:
Java's AWT library can be modeled with the composite pattern.
Collaborations

Clients use the component class interface to interact with objects in the composite structure.

If the recipient is a Leaf, then the request is handled directly.

If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.
Consequences

• Defines class hierarchies consisting of primitive objects and composite objects.
• Makes the clients simple. Clients can treat composite structures and individual objects uniformly. Clients normally don’t know whether they are dealing with a leaf or a composite component.
• Makes it easier to add new kind of components.
• Can make your design overly general – makes it harder to restrict the components of a composite.
Some Implementation Issues

- Explicit parent reference to simplify the traversal and management (such as deleting a component) of a composite structure.
- Maximizing the component interface – trade-off between client awareness and class hierarchy.
- Declaring the child management operations (in component interface or composite class) – trade-off between safety and transparency.
- Should component implement a list of components? – defining the set of children as an instance variable in the component class?
Sample code

P. 170
Problem 4

• Sometimes we want to add responsibilities to individual objects, not to an entire class.

• A graphical user interface toolkit should let you add properties like borders or behaviors like scrolling to any user interface component

• One way to add responsibilities is with inheritance -- inflexible
Problem 4

A more flexible approach is to enclose the component in another object that adds the scroll bar. The enclosing object is called a decorator. The decorator forwards requests to the component and may perform additional actions before or after forwarding. The decorator conforms to the interface of the component itdecorates so that its presence is transparent to the component’s client. This lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.
How about switching the order of the two decorators?

Draw the class diagram
Problem 4

Client

VisualComponent
Draw()

TextView
Draw()

Decorator
component
Draw()

Component.draw()

ScrollDecorator
Draw()
ScrollTo()

BorderDecorator
Draw()
DrawBorder()

Decorator:Draw()
DrawBorder();
Decorator

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

Also known as Wrapper.
Consequences

More flexibility than static inheritance – responsibilities can be added and removed at run-time simply by attaching and detaching them. You can even attach the same decorator (such as a border decorator) twice.

Avoids heavy featured classes high up in the hierarchy. Also good for unforeseen extensions.

A decorator and its component aren’t identical. They have different object identities. Can you cast a decorator to its component?

Lots of little objects – easy to customize but hard to learn and debug.
Some Implementation Issues

• Interface conformance – A decorator object’s interface must conform to the interface of the component it decorates.

• Omitting the abstract decorator class if you only need to add one responsibility

• Keeping component classes lightweight – it should focus on defining an interface, not on storing data. Otherwise, the decorators will be too heavy-weighted to use in quantity.

• Different from Strategy pattern
Sample code

P. 180
Problem5

- Subsystem 1 can look into the Subsystem 2 and call on any component or class operation at will.
- Why is this good?
  - Efficiency
- Why is this bad?
  - Can’t expect the caller to understand how the subsystem works or the complex relationships within the subsystem.
  - The subsystem will be easily misused.
  - Lead to non-portable code. For example, if needing to replace subsystem 2 with a new subsystem.
Problem 5

• The subsystem decides exactly how it is accessed.
• No need to worry about misuse by callers
• If a façade is used the subsystem can be used in an early integration test
  – We need to write only a driver
Facade

• Provides a unified interface to a set of objects in a subsystem.
• A facade defines a higher-level interface that shields clients from the internal classes, and makes the subsystem easier to use (i.e. it abstracts out the gory details)
Applicability

• Use the Façade pattern when
  – You want to provide a simple interface to a complex subsystem.
  – There are many dependencies between clients and the implementation classes of an abstraction.
  – You want to layer your subsystem.
Discussion

Commonly used if you want to layer your subsystems.

Façade promotes weak coupling between the subsystem and its clients. This can eliminate complex or circular dependencies.

It does not prevent applications from using subsystem classes if they need to.

Reducing client-subsystem coupling further by making Façade an abstract class with concrete subclasses for different implementations of the subsystem.

Public versus private subsystem classes.
Sample code

P. 188
Problem 6

- One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.
- Consider a document editor that can embed graphical objects in a document.
- Some graphical objects, like large images, can be expensive to create.
- Goal: opening a document should be fast
- Solution: delay creation of each expensive object until it is on demand.
- What do we put in the document in place of the image? (we still need to know the total pages of the document, for example)
- Transparency to client?
The proxy stores image extent – width and length

The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation.

Class diagram?
- **Images** are stored and loaded separately from text.
- The client cannot tell that it is dealing with a **ProxyImage** instead of a **RealImage**.
Proxy

Provide a surrogate or placeholder for another object to control access to it. Also known as Surrogate
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 standin
- 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 students shared by administrators, teachers and students.
- Smart reference. Typical uses include:
  - Counting the number of references to the real object so that it can be freed automatically when there are no more references.
  - Checking that the real object is locked before it is accessed.
• 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)
Consequences

A remote proxy can hide the fact that an object resides in a different address space.

A virtual proxy can perform optimizations such as creating an object on demand.

Both Protection proxies and smart references allow additional housekeeping tasks when an object is accessed.
Sample code

P. 213
Discussion of structural patterns

• Adapter vs Bridge
  – Both promote flexibility by providing a level of indirection to another object. Both involve forwarding requests to this object from an interface other than its own.
  – The key difference lies in their intents.
  – Adapter focuses on resolving incompatibilities between two existing interfaces
  – Adapter pattern makes things work after they’re designed.
  – Bridge makes them work before they are designed.
Discussion of structural patterns

• Adapter vs Facade
  – Can we think of a façade is an adapter to a set of other objects?
  – Façade defines a new interface.
  – Adapter reuses an old interface. It makes two existing interfaces work together as opposed to defining an entirely new one.
  – Used at different stages of software development.
Discussion of structural patterns

• Composite vs Decorator
  – Both rely on recursive composition to organize an open-ended number of objects. Can we think of a decorator object as a degenerate composite?
  – Decorator is designed to let you add responsibilities to objects without subclassing.
  – Composite focuses on structuring classes so that many related objects can be treated uniformly, and multiple objects can be treated as one. Its focus is not on embellishment but on representation.
  – They can be used in concert: there will be an abstract class with some subclasses that are composite, and some are decorators, and some are fundamental building blocks.
  
• How do they view each other in this scenario?
Discussion of structural patterns

• Proxy vs Decorator
  – Both compose an object and provide an identical interface to clients
  – The Proxy pattern is not concerned with attaching or detaching properties dynamically, and it is not designed for recursive composition.
  – Decorator addresses the situation where an object's total functionality can’t be determined at compile time, at least not conveniently.
  – The proxy focuses on one relationship – between the proxy and its subject – and that relationship can be expressed statically.
Creational Patterns

- Creational patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed and represented.
- Abstract Factory
- Builder
- Factory Method
- Singleton
Problem

• 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?
  – Different look-and-feel define different appearances and behaviors for user interface “widgets” like scroll bars, windows, and buttons.
  – An application should not hard-code its widgets for a particular look and feel.
An application should not hard-code its widgets for a particular look and feel.
Why do we need the Button and ScrollBar interfaces?

To make the Client unaware of the specific type of button/scrollbar.
Problem

– The abstract WidgetFactory class declares an interface for creating each basic kind of widget
– There is also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards.
– Clients call WidgetFactory’s operations to obtain widget instances, but clients aren’t aware of the concrete classes they’re using.
Applicability for Abstract Factory Pattern

- The system should be independent of how its products are created, composed or represented.
- A system should be configured with one family of products, where one has a choice from many different families.
- You want to provide a class library for a customer ("facility management library"), and you want to reveal just their interfaces, not their implementations.
- A family of related products is designed to be used together and you need to enforce this constraint.
- 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.
AbstractFactory

CreateProductA()
CreateProductB()

ConcreteFactory1
CreateProductA()
CreateProductB()

ConcreteFactory2
CreateProductA()
CreateProductB()

AbstractProductA

ProductA1
ProductA2

AbstractProductB

ProductB1
ProductB2

Client

Initiation Association:
Class ConcreteFactory2 initiates the associated classes ProductB2 and ProductA2

Question: How about adding another Product? Another Factory?
Another example

- 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?
Example: A Facility Management System for the Intelligent Workplace

- **Facility Mgt System**
  - **IntelligentWorkplace**
    - **InitLightSystem**
    - **InitBlindSystem**
    - **InitACSystem**
  - **LightBulb**
    - **InstabusLightController**
    - **ZumbobelLightController**
  - **Blinds**
    - **InstabusBlindController**
    - **ZumbobelBlindController**

- **SiemensFactory**
  - **InitLightSystem**
  - **InitBlindSystem**
  - **InitACSystem**

- **ZumtobelFactory**
  - **InitLightSystem**
  - **InitBlindSystem**
  - **InitACSystem**
Consequences

It isolates concrete classes from clients
It makes exchanging product families easy
It promotes consistency among products
Supporting new kinds of products is difficult. It requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses.
Sample code

p. 92
Problem 3

• It’s important for some classes to have exactly one instance. For example, there should be only one file system and one window manager.

• How do we ensure that a class has only one instance and that the instance is easily accessible?

• A global variable makes an object accessible, but it doesn’t keep you from instantiating multiple objects.

• The solution is to make the class itself responsible for keeping track of its sole instance.
Singleton

Ensure a class only has one instance, and provides a global point of access to it.
Applicability

Use the singleton when

• There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.

• When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
Clients access a singleton instance solely through singleton’s instance operation
Some Implementation Issues

- Ensure a unique instance
- Subclassing the singleton class – see sample code in textbook

Class Singleton {
    Public static Singleton instance();
    Protected Singleton();
    Private static Singleton _instance;
}

Singleton instance() {
    If(_instance == 0 ){
        _instance = new Singleton;
    }
    Return _instance;
}
Consequences

• Controlled access to sole instance
• Permits refinement of operation and representation
• Permits a variable number of instances
• Now consider an intelligent server that can be created “on demand” and close itself automatically if it has no clients.
  – Who will create the server (assuming the server is not running if not created)? How?
  – How to keep track of the number of clients?
  – Add/delete references if receiving connect/disconnect request. And query clients periodically.
Sample code

P. 132
Other patterns to be covered

Builder
Factory Method
Command
Iterator
Observer
Strategy
What is Due Next

Abstract of Literature Review Topic

• A template