Objectives

- To introduce structural design patterns
  - Facade
  - Decorator
  - Composite
  - Adapter
  - Flyweight
  - Proxy
Facade Design Pattern

● Design Purpose

  ● Provide an interface to a package of classes.

● Design Pattern Summary

  ● Define a singleton which is the sole means for obtaining functionality from the package.
Clients communicate with the package (subsystem) by sending requests to Facade, which forwards them to the appropriate package object(s).
Facade: Applicability

- To provide simple interface to a complex package, which is useful for most clients.

- To reduce the dependencies between the client and the package, or dependencies between various packages.
Facade: Structure
Facade: Sequence Diagram
Facade: Consequences

- It shields clients from package components, thereby reducing the number of objects that clients deal with and making the package easier to use.

- It promotes weak coupling between the package and its clients and other packages, thereby promoting package independence and portability.

- It doesn't prevent applications from using package classes if they need to.
Decorator Design Pattern

- **Design Purpose**
  - Add responsibilities to an object at runtime.

- **Design Pattern Summary**
  - Provide for a linked list of objects, each encapsulating responsibility.
Decorator creates an aggregated linked list of *Decoration* objects ending with the basic *Substance* object.
Linked Objects in Decorator

client: Client

decoration1: Decoration

decoration1.objectDecorated: Decoration

...: Decoration

....: Substance
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with...
Text View Example (Cont’d)
Decorator: Consequences

- Why use Decorator when a simple vector of Component objects seems to suffice?
  - Instead of simple interface with a single Component object that Decorator provides, the client would have to know about the subclasses Substance and Decoration, thereby making the client code more specialized.
  - If other classes replaced Substance or Decoration, the client code would have to change. This is not the case when Decorator is used.

- Therefore, Decorator would make the client’s interface more flexible.
Composite Design Pattern

- Design Purpose
  - Represent a Tree of Objects.

- Design Pattern Summary
  - Use a Recursive Form in which the tree class aggregates and inherits from the base class for the objects.
Composite: Class Model

- **Component**
  - add(Component c)
  - doIt()

- **LeafNode**
  - doIt()

- **NonLeafNode**
  - doIt()

- **TypeANonLeafNode**
  - doIt()

- **TypeBNonLeafNode**
  - doIt()

- **Objects**
  - non-leaf node
  - leaf node
Composite: Sequence Diagram
Bank/Teller Example

```
Client
  └── Employee
    └── Clerk
    └── Teller
    └── Supervisor
      └── Manager
      └── President

1..n
reports
```
Composite: Consequences

- Defines class hierarchies consisting of primitive objects and composite objects.
- Makes the client simple.
- Makes it easier to add new kinds of components.
Adapter Design Pattern

- **Design Purpose**
  
  - Allow an application to use external functionality in a retargetable manner.

- **Design Pattern Summary**
  
  - Write the application against an abstract version of the external class; introduce a subclass that aggregates the external class.
Adapter: 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, that is, classes that don't necessarily have compatible interfaces.
Adapter: Class Model

- Adapter is based on the delegation form because an Adapter object delegates the command to the targeted command.
Adapter: Sequence Diagram

- **Client**
  - `clientNameForRequiredMethod()`

- **AbstractClass**

- **Adapter**
  - `RequiredMethod()`

- **adaptee**
  - **RequiredClass**
Financial Application Example
Adapter: Consequences

- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Flyweight Design Pattern

- **Design Purpose**
  - Manage a large number of objects without constructing them all.

- **Design Pattern Summary**
  - Share representatives for the objects; use context to obtain the effect of multiple instances.
Flyweight: Class Model

- **Client**
  - `FlyweightFactory`
    - `getFlyweight(Characteristic)`
  - `Flyweight`
    - `doAction(Context)`
  - `ConcreteFlyweight`
    - `doAction(Context)`

1..n relationship between `Client` and `FlyweightFactory`
Flyweight: Sequence Diagram

[Sequence diagram showing the interaction between Client, FlyweightFactory, and flyweight.]
Text Magnifier Example

Input

ABBRA CADABBRAA ARE THE FIRST TWO OF MANY WORDS IN THIS FILE ...

Input color: RED .... Starting character: 2 ... Ending character: 3

Output

```
  o   v   v   v   v   v   v
  o   o   v   v   v   v   v
  o   o   v   v   v   v   v
  o   o   o   - R E D -   - R E D -   . . . . . . .
  o   o   o   o   v   v   v   v
  o   o   v   v   v   v   v
  o   o   v   v   v   v   v
  o   o   v   v   v   v   v
```
Text Magnifier Example (Cont’d)

**Client Responsibilities**

Use string to determine which flyweight. Use color information to form the context (parameter value).

**DP Responsibilities**

Make (shared) BigA, BigB, … flyweight object available to clients.

- **bigA**: BigA
- **bigB**: BigB

Flyweights (1 each)

- getMatrix(“black”)
- getMatrix(“red”)

ABBRA CADABBRA...

color “RED” begins 0…

Line for output

......
Text Magnifier Example (Cont’d)
Flyweight: Consequences

- Space savings increase as more flyweights are shared.
Proxy Design Pattern

- **Design Purpose**
  - Avoid the unnecessary execution of expensive functionality in a manner transparent to clients.

- **Design Pattern Summary**
  - Interpose a substitute class which accesses the expensive functionality only when required.
Proxy: Class Model

```
// One way to check if really needed:
if (realActiveObject == null) // never referenced
{
    realActiveObject = getRealActiveObject();
    realActiveObject.expensiveMethod();
}
else // try to avoid calling the real expensiveMethod()
```
Proxy: Sequence Diagram

- Client
- Proxy
- RealActiveClass

expensiveMethod() -> (if needed:) realExpensiveMethod()
Telephone Record Example

Console

Please pick a command from one of the following:
quit
middle
all

-------------- Retrieving from the Internet ---------------
9049249 John Doss
9049250 James Dossey

Please pick a command from one of the following:
quit
middle
all

=== No need to retrieve from the Internet ===
9049249 John Doss
9049250 James Dossey

Please pick a command from one of the following:
quit
middle
all

======= No need to retrieve from the Internet =======
9049031 John Don
9049032 John Dol
9049033 John Don
9049034 John Dop
9049035 John Dor
9049036 John Dos
Telephone Record Example (Cont’d)

Ensures that `TelephoneApp` makes calls with `TelNumsProxy` instance

```java
... // One way to check if really needed:
if (value == null)  // never referenced
    remoteTelNums.getTelNums();
else  // no need to call 'getTelNums()' 
```

Diagram:

- **TelephoneApp**
  - display(TelNums)
  - displayMiddleRecord()

- **TelNums**
  - value: Vector
  - getTelNums(): Vector
  - showMiddleRecord()

- **RemoteTelNums**
  - getTelNums()

- **TelNumsProxy**
  - getTelNums()

- **Setup**
  - 1 remoteTelNums
Proxy: Consequences

- Proxy promotes:
  - Efficiency: avoids time-consuming operations when necessary.
  - Correctness: separates design and code that are independent of retrieval/efficiency from parts concerned with this issue.
  - Reusability: design and code that are independent of retrieval efficiency are most likely to be reusable.
  - Flexibility: we can replace one module concerned with retrieval with another.
  - Robustness: isolates parts that check for the validity of retrieved data.

- The penalties we pay can sometimes be too high:
  - If the proxy forces us to keep very large amount of data in the memory and its use is infrequent.
Summary of Structural Design Patterns

- **Structural Design Patterns** relate objects (as trees, lists etc.)
  
  - *Facade* provides an interface to collections of objects
  - *Decorator* adds to objects at runtime
  - *Composite* represents trees of objects
  - *Adapter* simplifies the use of external functionality
  - *Flyweight* gains the advantages of using multiple instances while minimizing space penalties
  - *Proxy* avoids calling expensive operations unnecessarily