Object-Oriented Programming
Decorator Pattern
Proxy Pattern

CSIE Department, NTUT
Woei-Kae Chen
Decorator Pattern
Structural pattern
Decorator: Intent

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

- Also known as
  - Wrapper
Decorator: Motivation (1)

How to add a scrollbar and a border to an object?

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...
Option 1: subclass TextView, ShapeView, ...

Problems
• Class explosion
• Scroll + Border?
Option 2: Implement border and scrollbar in base class (e.g., by a Template Method)

Problems
- Fat base class
- Too much responsibility for base class
- Mixing border and scroll codes
- Impossible: Border + Border

Can also be designed as using separate strategy objects

Template Method

`VisualComponent`
- `doDraw()`
- `Draw()`
- `DrawBorder()`
- `DrawScroll()`
- `ScrollTo()`

`ShapeView`
- `doDraw()`

`TextView`
- `doDraw()`

```java
if (hasBorder)
    DrawBorder();
if (hasScroll)
    DrawScroll();
```
Option 3: apply Decorator Pattern

```
VisualComponent
  Draw()

Decorator
  Draw()

TextView
  Draw()
  ScrollTo()

ScrollDecorator
  scrollPosition
  Draw()

BorderDecorator
  borderWidth
  Draw()
  DrawBorder()

Decorator::Draw();
  DrawBorder();
  Draw();

component->Draw();
```
Decorator: Applicability

- **Use Decorator**
  - to add responsibilities to individual objects dynamically and transparently
    - without affecting other objects
  - for responsibilities that can be withdrawn
  - when extension by subclassing is impractical.
  - Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.
  - A class definition may be hidden or otherwise unavailable for subclassing.
Decorator: Participants

- **Component** (VisualComponent)
  - defines the interface for objects that can have responsibilities added to them dynamically.

- **ConcreteComponent** (TextView)
  - defines an object to which additional responsibilities can be attached.

- **Decorator**
  - maintains a reference to a Component object and defines an interface that conforms to component’s interface.

- **ConcreteDecorator** (BorderDecorator, ...)
  - adds responsibilities to the component.
Decorator: Collaboration

- **Decorator forwards requests to its Component object.**
  - It may optionally **perform additional operations** before and after forwarding the request.
Decorator: Consequences (1)

• Benefits
  – More flexibility than static inheritance
    • responsibilities can be added and removed at run-time
    • easy to add a property twice (e.g., double border)
  – Avoids feature-laden classes high up in the hierarchy
    • support all features in a complex, customizable class
      ⇔ define a simple class and add functionality with Decorator objects
    • Decorator offers a pay-as-you-go approach to adding responsibilities
      an application need not pay for features it does not use.
    • Easy to define new kinds of Decorators independently.
Liabilities

- A decorator and its components are not identical.

  A decorator act as a transparent enclosure. But the object identity of a decorated component is not identical to the component itself (not transparent).
  - You should not rely on object identity when you see decorators (e.g., C++ RTTI: typeid).

- Lots of little objects

  Decorator often results in systems composed of lots of little objects that all look alike.
  - The objects differs in the way they are interconnected, not in their class or in the value of their variables.
  - Easy to customize, but can be hard to learn and debug.
Decorator: Implementation (1)

- **Interface conformance**
  - An decorator object’s interface must conform to the interface of the components it decorates.

- **Omitting the abstract Decorator class**
  - If there is only one responsibility to add.

- **Keeping Component classes lightweight**
  - Component class should focus on defining an interface, not on storing data.
    - The definition of data representation should be deferred to subclasses.
  - If Component class is heavyweight (storing a lot of data), decorators becomes too heavyweight to use in quantity.
  - A lot of functionality in Component concrete subclasses pay for features they do not need.
**Decorator: Implementation (2)**

- **Changing the skin of an object versus changing its guts**
  - Decorator pattern
    - A decorator is a skin over an object that changes its behavior.
    - A Component does not know anything about its decorators.
  - Strategy pattern
    - Strategy pattern is a pattern for changing the guts.
    - Strategy pattern is better when the Component class is intrinsically heavy weight → making Decorator pattern too costly to apply.
    - Example: support different border styles by having the component delegate border-drawing to a separate Border object.
    - Extend the number of strategies from one to an open-ended list → achieve the same effect as nesting decorators recursively.
    - Component has to reference and maintain the corresponding strategies.
    - A strategy can have its own specialized interface.
Decorator: Related Patterns

● Adapter
  – Interface
    ● Decorator \(\rightarrow\) changes an object’s responsibilities, not its interface.
    ● Adapter \(\rightarrow\) gives an object a completely new interface

● Composite
  – A decorator looks like a degenerate composite with only one component. However, they have different intent:
    ● Decorator \(\rightarrow\) adds additional responsibilities.
    ● Composite \(\rightarrow\) object aggregation.

● Strategy
  – Two alternative ways of changing an object
    ● Decorator \(\rightarrow\) change the skin of an object.
    ● Strategy \(\rightarrow\) change the guts of an object.
Proxy Pattern
Structural pattern
Proxy: Intent

- Provide a surrogate or placeholder for another object to control access to it.

- Also known as
  - Surrogate
Proxy: Motivation (1)

- **DocumentEditor**
  - Opening a document should be fast
    - Problem: creating a raster image can be expensive
    - Solution: avoid creating expensive objects when the document is opened
  - *Creation on demand* for raster images
    - Create an image when it becomes visible
    - What do we put in document in place of the image?
    - How to hide creation on demand so that the editor remains simple?
  - Solution: image proxy
    - An image proxy act as a stand-in for the real image.
    - The image proxy creates the real image only when the document asks it to display itself.
    - The image proxy forwards subsequent requests directly to the image.

![Diagram of DocumentEditor, anImageProxy, and anImage](image)
Proxy: Motivation (2)

Image
- imageImp
- extend
- Draw()
- GetExtend()
- Store()
- Load()

ImageProxy
- fileName
- extend
- Draw()
- GetExtend()
- Store()
- Load()

if (image == 0)
    image = LoadImage(fileName);
    image->Draw();
else
    return extend;

if (image == 0)
    return extend;
else
    return image->GetExtend();

Load image until draw() is called
Proxy: Applicability

- When you need a more versatile or sophisticated reference to an object than a simple pointer.
  - remote proxy
    - provides a local representative for an object in a different address space (e.g., network).
  - virtual proxy
    - creates expensive objects on demand (e.g., ImageProxy).
  - protection proxy
    - controls access to the original object.
    - is useful when a object should have different access rights.
  - smart reference
    - performs additional actions when an object is accessed. For example:
      - counting the number of references to the real object
      - loading a persistent object when it is first referenced
      - checking that the real object is locked before it is accessed
Proxy: Structure (1)

Client

Subject
Request()
...()

RealSubject
Request()
...()

Proxy
Request()
...()

realSubject->Request();
...

aClient
subject

aProxy
realSubject

aRealSubject
Proxy: Structure (2)

In the Proxy design pattern, the `Client` interacts with a `Proxy` object, which acts as a wrapper for the `realSubject`. The `Proxy` can intercept requests from the `Client` and perform additional actions before delegating them to the `realSubject`. This is similar to the Decorator pattern, where additional responsibilities can be added to an object without changing its interface.

The `Proxy` class has a method `Request()` that delegates to the `realSubject->Request();` method. This structure allows for flexibility in adding or removing functionality without altering the `Client` code.

Key Classes:
- `Client`
- `Subject`
- `Proxy` (contains `realSubject`)
- `RealSubject`

Key Methods:
- `Client::Request()`: Invoked by the `Client`
- `Subject::Request()`: Invoked by the `Proxy`
- `Proxy::Request()`: Delegates to `realSubject->Request();`

Diagram:
- `Client` -> `Proxy`
- `Proxy` -> `realSubject`
- `realSubject` -> `realSubject->Request();`

Note: The diagram shows a simplified overview and actual implementation details may vary.
Proxy: Participants

● Proxy (ImageProxy)
  – maintains a reference that lets the proxy access the real subject.
  ● Proxy may refer to a Subject if the RealSubject and Subject interface are the same.
  – provides an interface identical to Subject’s so that a proxy can be substituted for the real subject.
  – controls access to the real subject and may be responsible for creating and deleting it.

● Subject (Graphic)
  – defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

● Real Subject (Image)
  – defines the real object that the proxy represents.
Proxy: Consequences

- **Proxy pattern introduces a level of indirection**
  - 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 proxy and smart references allow additional housekeeping tasks when an object is accessed.

- **Copy on write**
  - It is related to Creation on demand.
  - Copying a large object can be expensive.
  - If the copy is never modified → do not need to copy.
  - Use a proxy to postpone the copying process until the object is really modified.
Proxy: Implementation

- **Overloading the member access operator in C++**
  - When a proxy behaves just like a pointer
  - Overload `operator->` and `operator*` to simplify coding

- **Using `doesNotUnderstand` in Smalltalk**

- **Proxy does not always have to know the type of real subject**
  - If an abstract interface is sufficient.
  - If Proxies need to instantiate RealSubjects (e.g., virtual Proxies) they have to know the concrete class.
Proxy: Related Patterns

- **Adapter**
  - Interface
    - Proxy → provides the same interface as its subject
      - A protection proxy may provide an interface that is effectively a subset of its real subject’s interface.
    - Adapter → gives an object a completely new interface

- **Decorator**
  - Intent
    - Proxy → control access to an object
    - Decorator → adds responsibilities to an object
  - Implementation
    - A protection proxy might be implemented exactly like a decorator.
    - A remote proxy does not contain a direct reference to its real subject → contains indirect reference (e.g., IP, Port).
    - A virtual proxy starts off with an indirect reference (e.g., file name) → eventually obtain and use a direct reference.
Composite Pattern

\[ \Leftrightarrow \]

Decorator Pattern

\[ \Leftrightarrow \]

Proxy Pattern

(GoF: page 219)
Composite: Structure

Client → Component

- Operation()
- Add(Component)
- Remove(Component)
- GetChild(int)

Composite

- Operation()
- Add(Component)
- Remove(Component)
- GetChild(int)

Leaf

Operation()

for all g in children

g.Operation() ;
Decorator: Structure

Component
  Operation()

ConcreteComponent
  Operation()

Decorator
  Operation()

ConcreteDecoratorA
  addedState
  Operation()

ConcreteDecoratorB
  Operation()
  AddedBehavior()

component->Operation();
Decorator::Operation();
AddedBehavior();
Proxy: Structure

Client

RealSubject
Request()
...()

Proxy
Request()
...()

Subject
Request()
...()

realSubject->Request();
...

virtual proxy
Composite ↔ Decorator

- Similar structure diagrams
  - Reason: both rely on recursive composition

- Different intents
  - Decorator → adds responsibilities to an object
    - avoid explosion of subclassing
  - Composite → object aggregation
    - treats one and multiple objects uniformly

- Composite and Decorator are often used in concert
  - An abstract class with some subclasses that are composites, decorators, and leaf components.
  - Composite and Decorator have a common interface.
  - Decorator view: a composite is a ConcreteComponent.
  - Composite view: a decorator is a leaf.
Decorator ↔ Proxy

● Similarity
  – Both provide a level of indirection to an object.
  – Both keep a reference to another object.
  – Both provide an identical interface to clients.

● Different intents
  – Proxy → control access to an object
    ● A subject defines functionality.
    ● A proxy provides (or refuses) access of functionality of a subject.
    ● Focus on the relationship between the proxy and its subject → static relationship
  – Decorator → adds responsibilities to an object
    ● A component provides only part of the functionality.
    ● One or more decorators add functionalities to a component.
    ● Address the situation where an object’s total functionality can not be determined at compile-time → dynamic relationship

● Cooperation: Proxy-Decorator or Decorator-Proxy