Object-Oriented Design

Lecturer: Raman Ramsin

Lecture 21

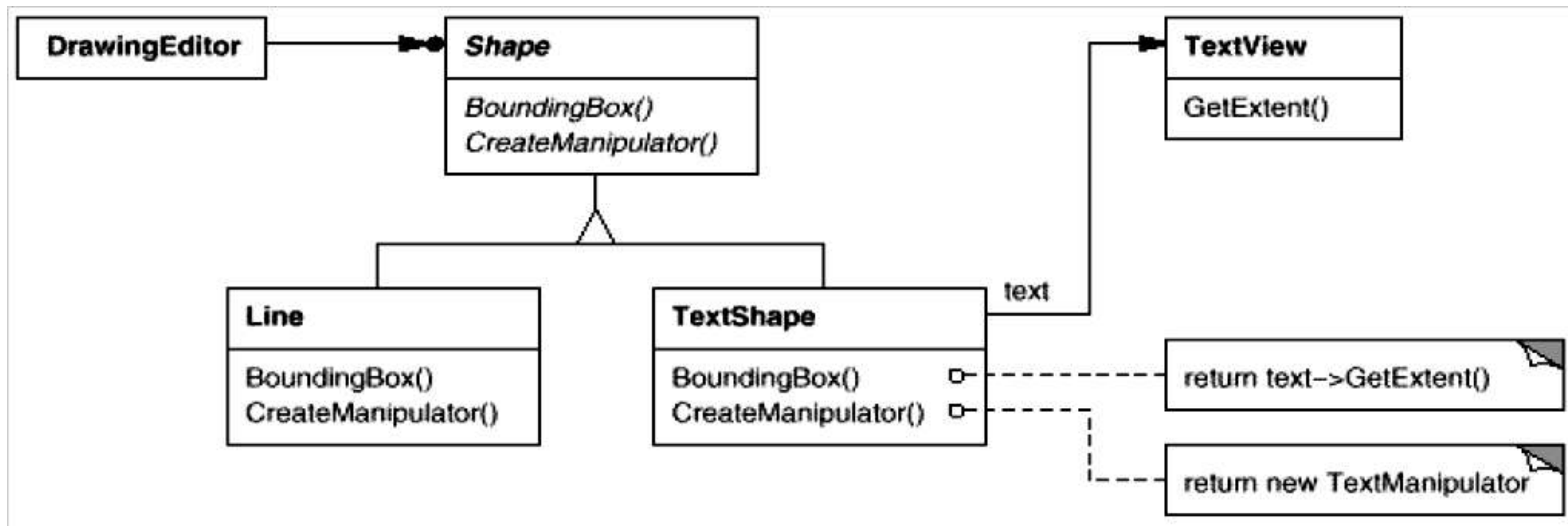
GoF Design Patterns – Structural

GoF Structural Patterns

- **Class/Object**
  - ***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.
- **Object**
  - ***Bridge:*** Decouple an abstraction from its implementation so that the two can vary independently.
  - ***Composite:*** Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  - ***Decorator:*** Attach additional responsibilities to an object dynamically.
  - ***Façade:*** Provide a unified interface to a set of interfaces in a subsystem.
  - ***Flyweight:*** Use sharing to support large numbers of fine-grained objects efficiently.
  - ***Proxy:*** Provide a surrogate or placeholder for another object to control access to it.

Adapter

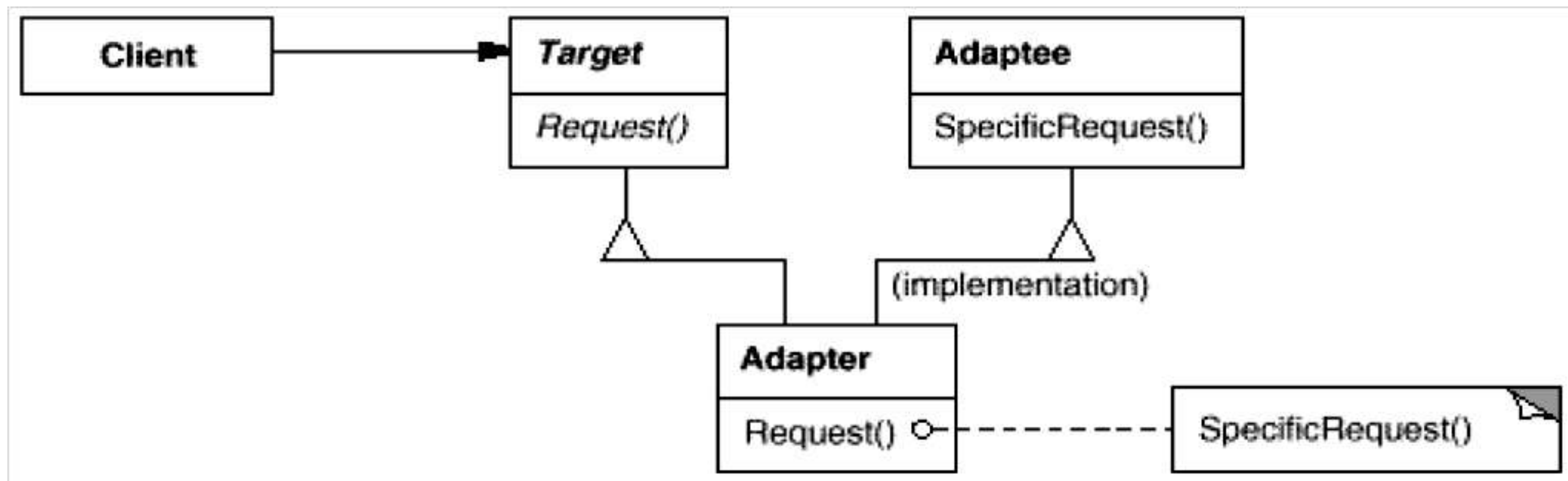
- Intent:
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.



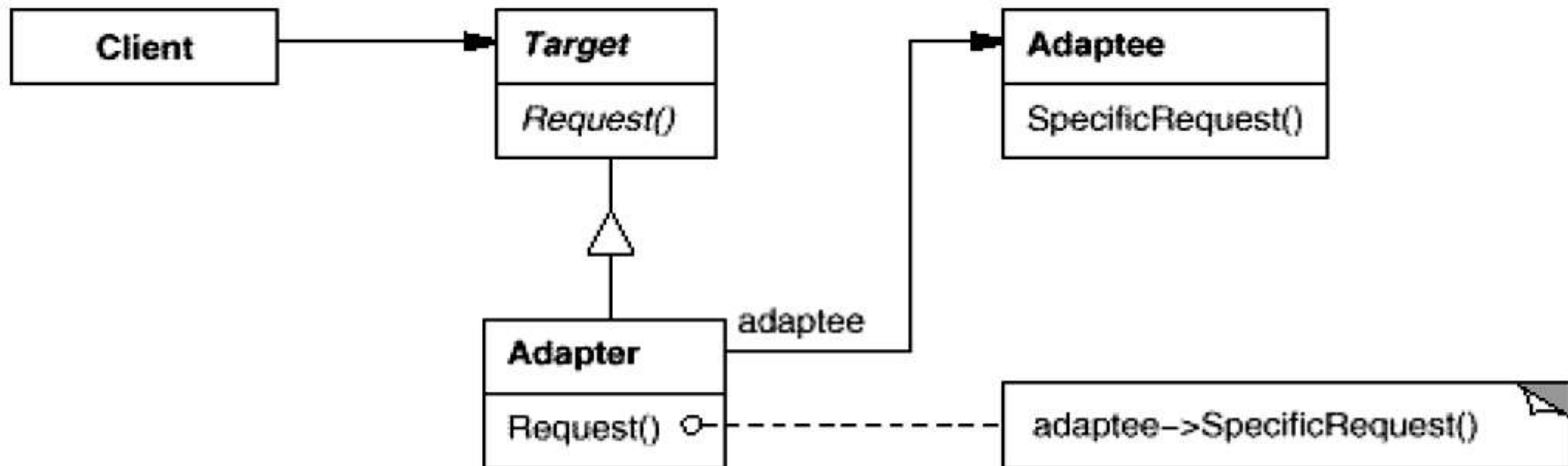
Adapter: Applicability

- 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.
  - *(object adapter only)* you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Adapter (Class): Structure

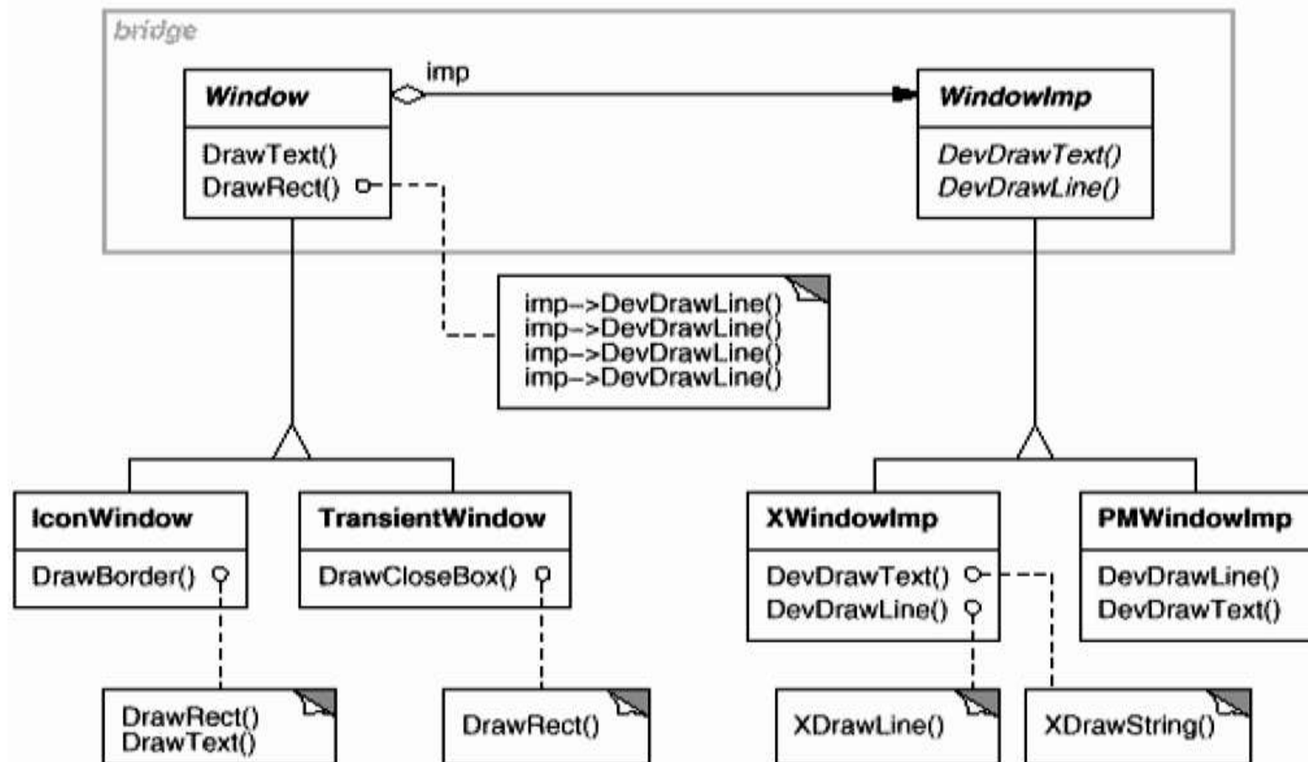


Adapter (Object): Structure



Bridge

- Intent:
  - Decouple an abstraction from its implementation so that the two can vary independently.



Bridge: Applicability

- Use the Bridge pattern when
  - you want to avoid a permanent binding between an abstraction and its implementation; for example, when the implementation must be selected or switched at run-time.
  - both the abstractions and their implementations should be extensible by subclassing; combine different abstractions and implementations and extend them independently.
  - changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
  - (C++) you want to hide the implementation of an abstraction completely from clients. In C++ the representation of a class is visible in the class interface.
  - you want to share an implementation among multiple objects and this fact should be hidden from the client.

Bridge: Structure

Client

Abstraction

Operation()

imp

imp->OperationImp();

Implementor

OperationImp()

RefinedAbstraction

ConcreteImplementorA

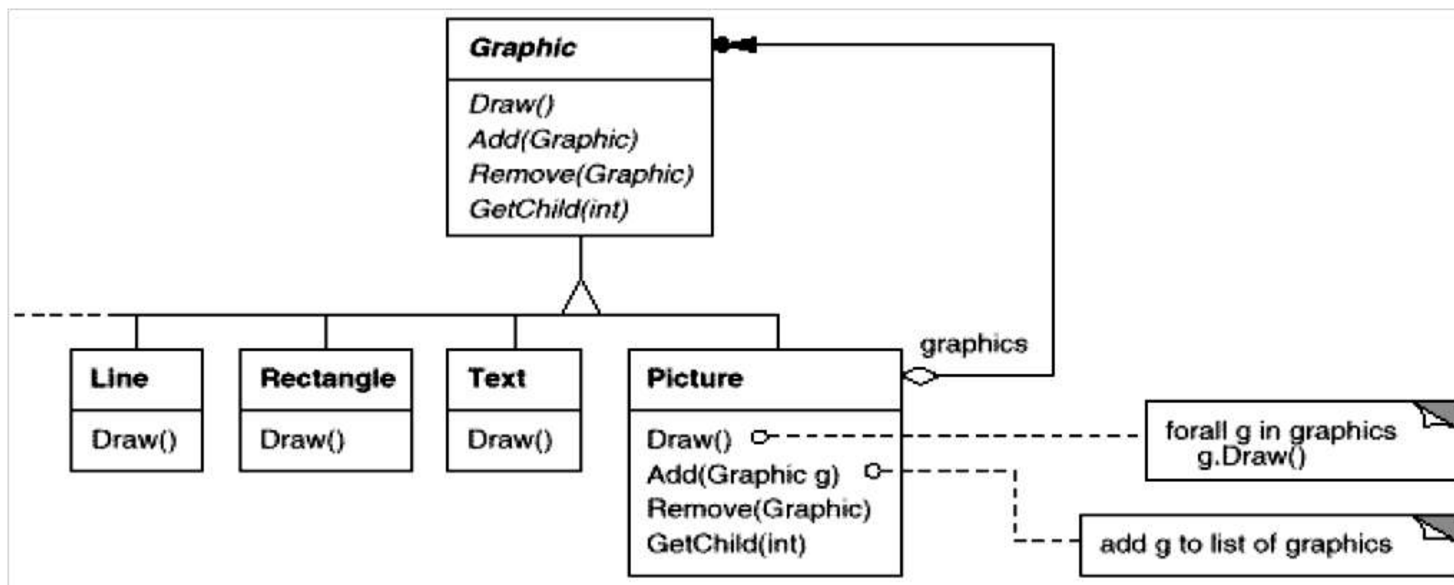
OperationImp()

ConcreteImplementorB

OperationImp()

Composite

- Intent:
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.



Composite: Applicability

- Use the Composite pattern when
  - you want to represent whole-part- hierarchies of objects.
  - you want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite: Structure

Client

Component

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

Leaf

Operation()

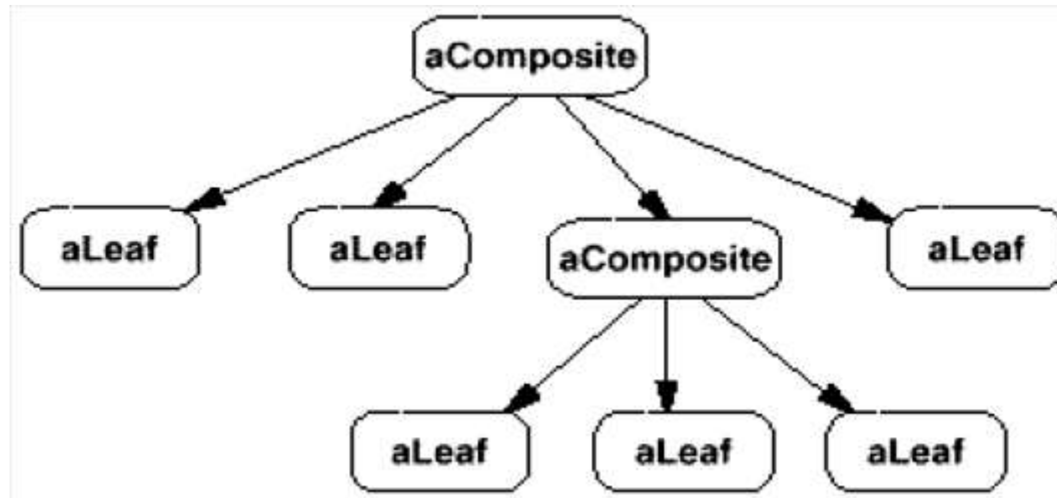
Composite

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

children

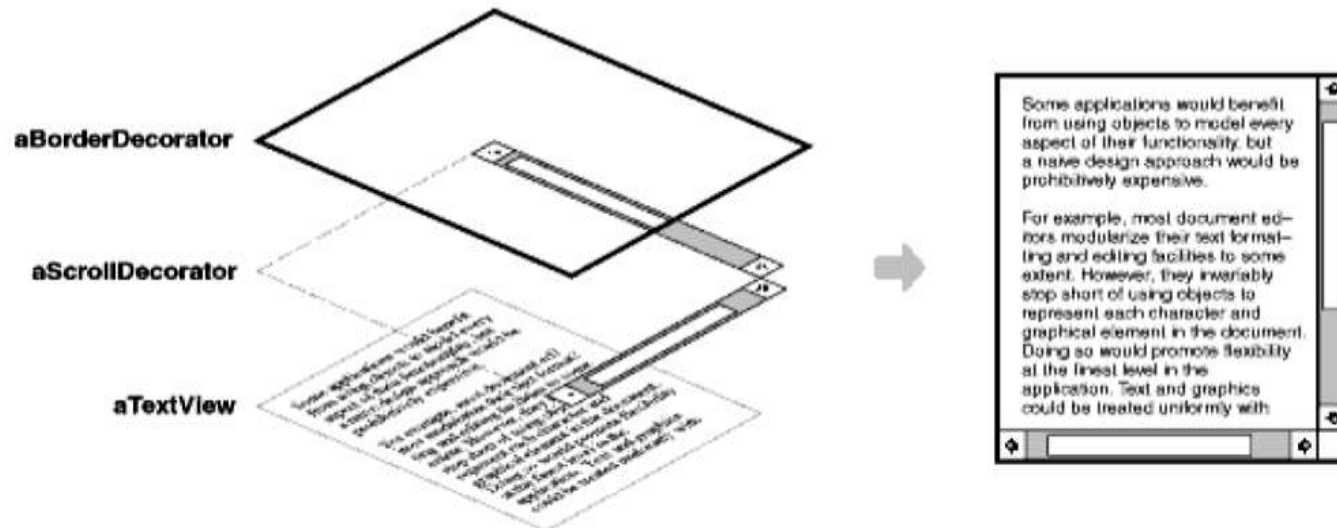
forall g in children
g.Operation();

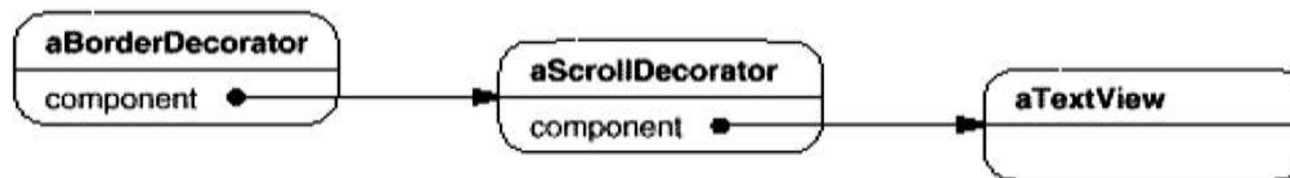
Composite: Typical Object Structure



Decorator

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





Decorator: Class Hierarchy

VisualComponent

Draw()

TextView

Draw()

Decorator

Draw()

component

component->Draw()

ScrollDecorator

Draw()
ScrollTo()

scrollPosition

BorderDecorator

Draw()
DrawBorder()

borderWidth

Decorator::Draw();
DrawBorder();

Decorator: Applicability

- Use the Decorator pattern
  - to add responsibilities to individual objects dynamically and transparently, that is, 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.

Decorator: Structure

Component
Operation()

ConcreteComponent
Operation()

Decorator
Operation()

component

component->Operation()

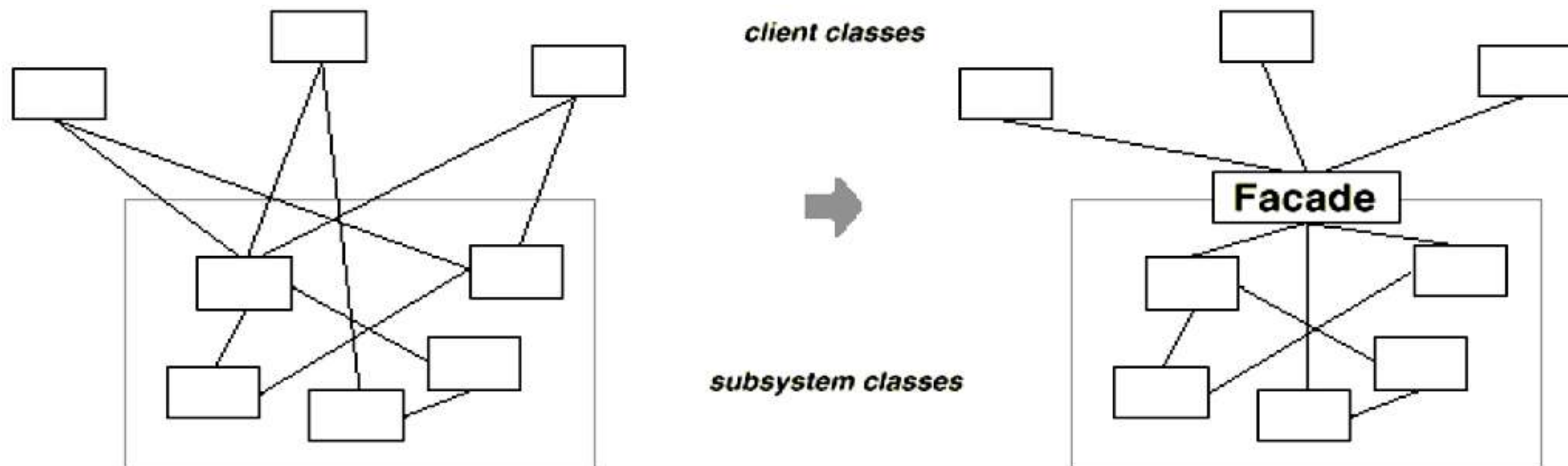
ConcreteDecoratorA
Operation()
addedState

ConcreteDecoratorB
Operation()
AddedBehavior()

Decorator::Operation();
AddedBehavior();

Façade

- Intent:
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



Façade: Class Hierarchy

Compiler

Compile()

compiler subsystem classes

Stream

Scanner

Token

Parser

Symbol

BytecodeStream

ProgramNodeBuilder

ProgramNode

CodeGenerator

StatementNode

ExpressionNode

StackMachineCodeGenerator

RISCCodeGenerator

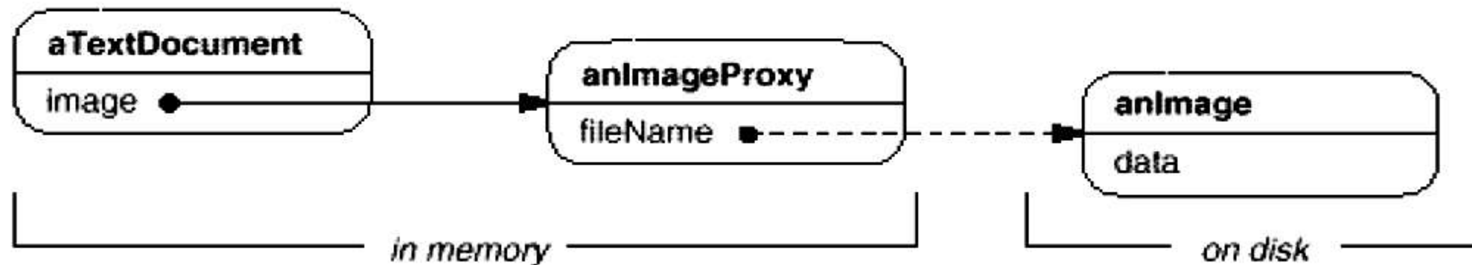
VariableNode

Façade: 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 subsystems. Use a facade to define an entry point to each subsystem level.

Proxy

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



Proxy: Class Hierarchy

DocumentEditor

Graphic
Draw()
GetExtent()
Store()
Load()

Image
Draw()
GetExtent()
Store()
Load()
imageImp
extent

image

ImageProxy
Draw()
GetExtent()
Store()
Load()
fileName
extent

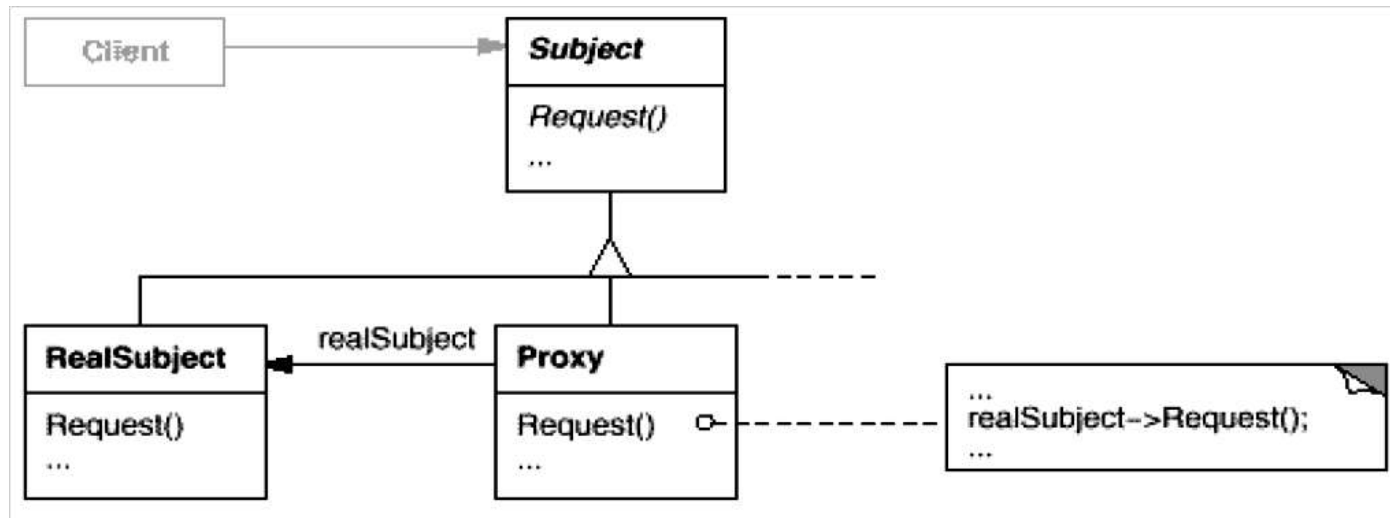
```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

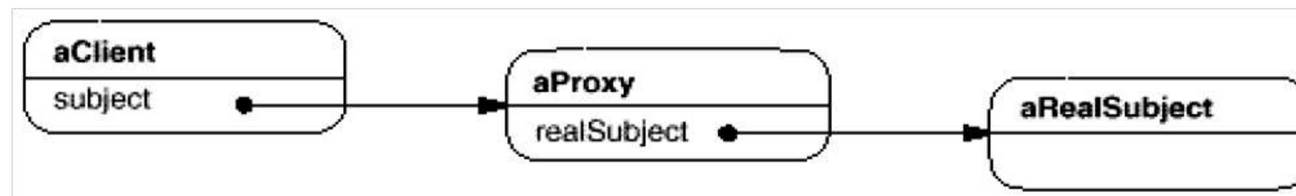
```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

Proxy: Applicability

- Use the Proxy pattern when a surrogate is needed:
  - **Remote proxy:** provides a local representative for an object in a different address space.
  - **Virtual proxy:** creates expensive objects on demand.
  - **Protection proxy:** controls access to the original object.
  - **Smart reference:** a replacement for a bare pointer that performs additional actions when an object is accessed:
    - counting the number of references to the real object so that it can be freed when there are no more references.
    - loading a persistent object into memory when it's first referenced.
    - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Proxy: Structure





Reference

- Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, 1995.