Structural Patterns

Prakash Badhe

prakash.badhe@vishwasoft.in

Go Structural…

- Structural patterns describe how classes and objects can be combined to form larger structures.

Catalog Structural - I

- **Adapter pattern** : used to make one class interface match another to make programming easier
- **The Composite :** is exactly that a composition of objects, each of which may be either simple or itself a composite object.

Catalog Structural - II

- Proxy pattern is frequently a simple object that takes the place of a more complex object that may be invoked later, for example when the program runs in a network environment.

Catalog Structural - III

- The **Flyweight** pattern is a pattern for sharing objects, where each instance does not contain its own state, but stores it externally. This allows efficient sharing of objects to save space, when there are many instances, but only a few different types.

Catalog Structural - IV

- The **Façade** pattern is used to make a single class represent an entire subsystem.
- the **Bridge** pattern separates an object's interface from its implementation, so you can vary them separately.
- The **Decorator** pattern, which can be used to add responsibilities to objects dynamically.
- You'll see that there is some overlap among these patterns and even some overlap with the behavioral patterns.

Incompatible Interfaces

- We know that the interface abstracts the way the client looks at the object,but in cases where the interface is not compatible can the client still use the same object ?
- Adapter pattern makes it possible.

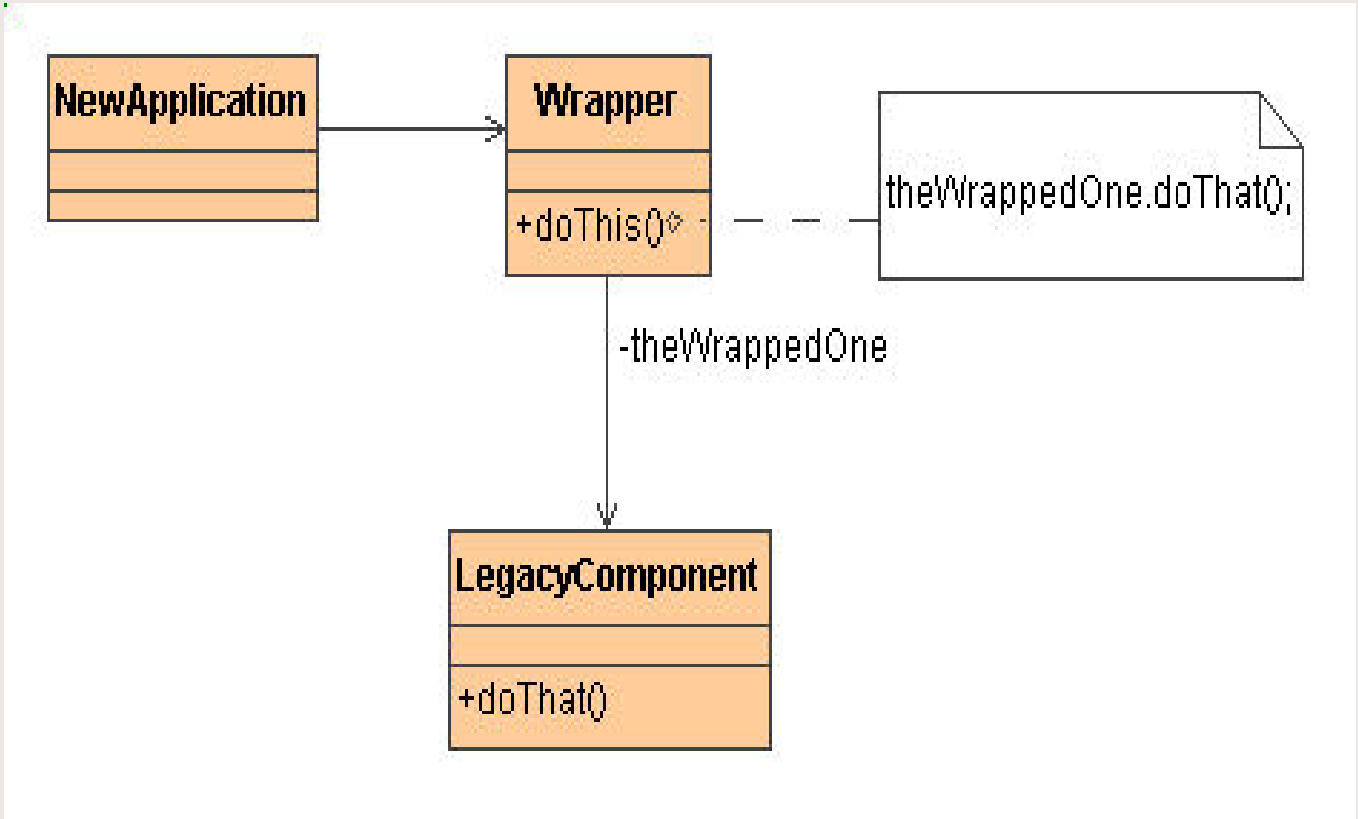
Adapter Pattern

- **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 functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class. It effectively offers an ***impedance-matching*** facility

Creating an Adapter

- Create an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

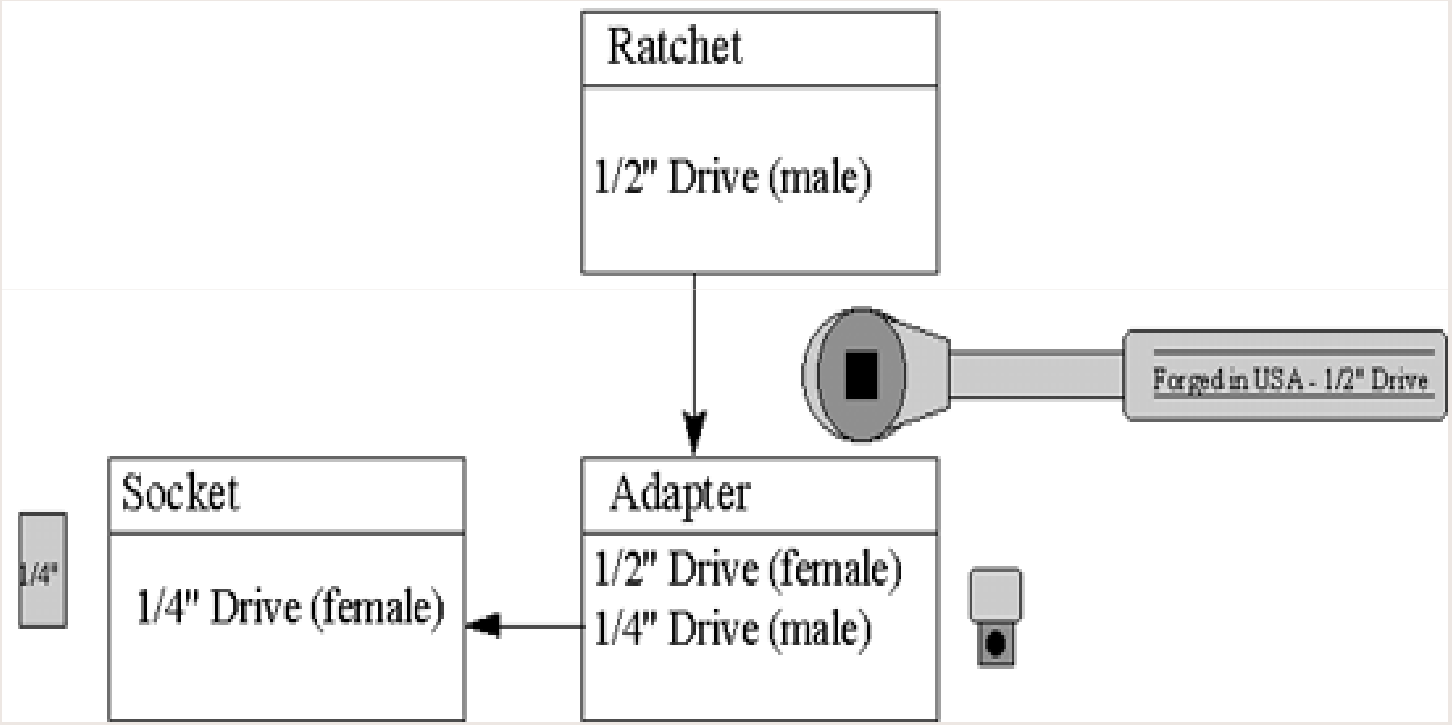
Adapter Class diagram



Adapter Example

- The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.
- Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same.

Adapting itself



Class and Object Adapter

The Class adapter

Won't work when we want to adapt a class and all of its subclasses, since you define the class it derives from when you create it.

Lets the adapter change some of the adapted class's methods but still allows the others to be used unchanged.

An Object adapter

Could allow subclasses to be adapted by simply passing them in as part of a constructor.

Abstraction and implementation

- In class hierarchies '***Software Hardening***' occurs by using sub classing of an abstract base class to provide alternative implementations. This locks in compile-time binding between interface and implementation.
- The abstraction and implementation cannot be *independently extended or composed*.

Bridge between them..

- Bridge pattern decouples an abstraction from its implementation so that the two can vary independently.

How Bridge Works…

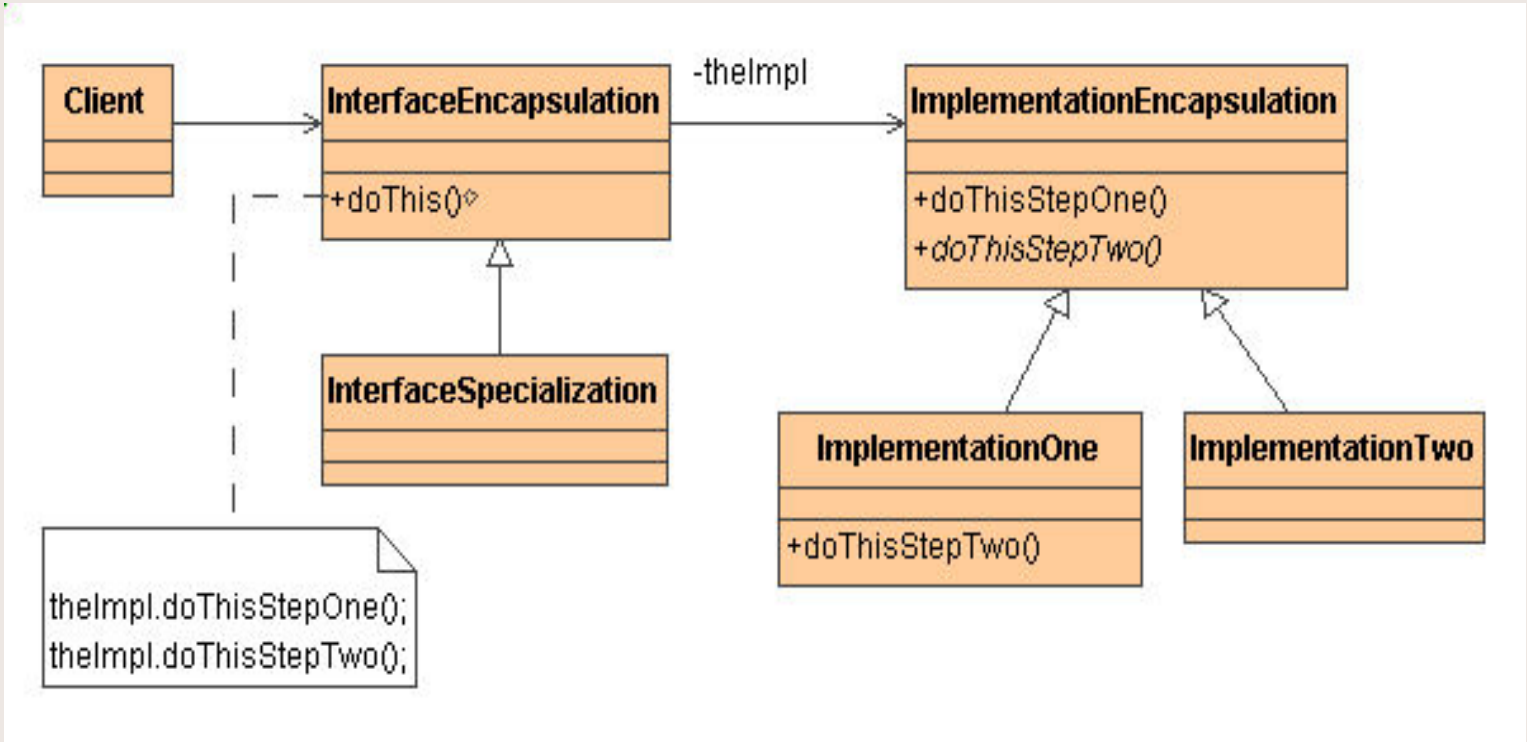
- The interface class contains a pointer to the abstract implementation class. This pointer is initialized with an instance of a concrete implementation class, but all subsequent interaction from the interface class to the implementation class is limited to the abstraction maintained in the implementation base class.
- The client interacts with the interface class, and it in turn "delegates" all requests to the implementation class.

Separation by Bridge…

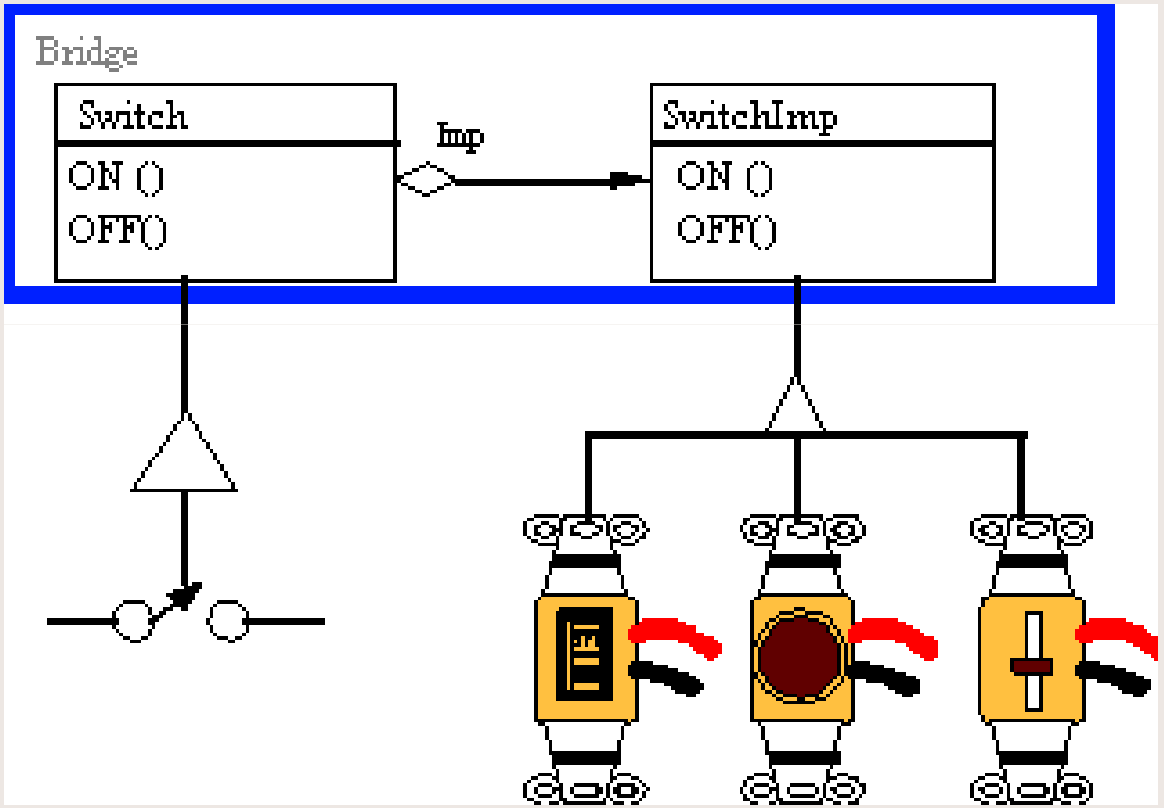
The interface object is the "handle" know and used by the client; while the implementation object, or "body", is safely encapsulated to ensure that it may continue to evolve, or be entirely replaced (or shared at run-time.

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently.

Bridge Diagram



Bridge Example



Bridge Use..

Use the Bridge pattern when:

- you want run-time binding of the implementation
- you have a proliferation of classes resulting from a coupled interface and numerous implementations,
- you want to share an implementation among multiple objects,
- you need to map orthogonal class hierarchies.

Bridge Consequences..

- Decoupling the object's interface,
- Improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),
- Hiding details from clients.

Bridge idiom

- Bridge is a synonym for the "handle/body" idiom
- This is a design mechanism that encapsulates an implementation class inside of an interface class. The **implementation** class is the body, and the **Interface** is the handle. The handle is viewed by the user as the actual class, but the work is done in the body.

Bridge and adapter

- Adapter makes things work after they're designed.
- Bridge makes them work before they are. Bridge is designed up-front to let the abstraction and the implementation vary independently.
- Adapter is retrofitted to make unrelated classes work together.

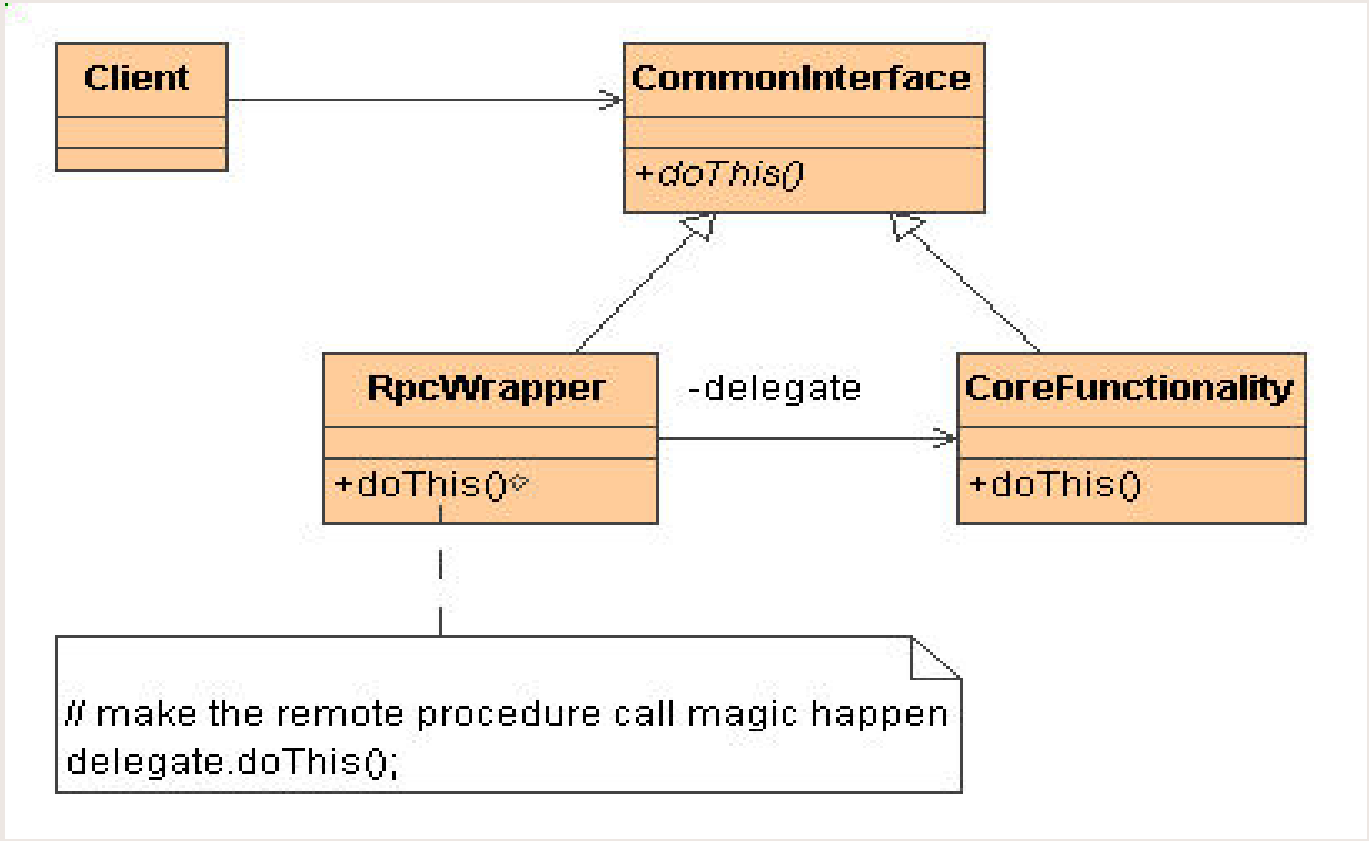
Expensive Objects

- Sometimes the object creation process by client is costly and you don't want the client to remain engaged in object creation process.
- Instead you provide a **Dummy/Proxy** object to the client which creates the real object only when it is required by the client.
- The Proxy provides a surrogate or place holder to provide access to an object

Proxy Pattern

- Design a surrogate, or proxy, object that: instantiates the real object the first time the client makes a request of the proxy.
- Remembers the identity of this real object.
- Forwards the instigating request to this real object. Then all subsequent requests are simply forwarded directly to the encapsulated real object.

Proxy Class Diagram



Proxy Examples

- A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.
- **Stub** is the proxy for RMI server in RMI Client applications.

Proxy uses

- If an object, such as a large image, takes a long time to load,proxy acts as intermediary to create the real object.
- the proxy can load the real object only when it is needed.
- If the object is on a remote machine and loading it over the network may be slow, especially during peak network load periods.
- If the object has limited access rights, the proxy can validate the access permissions for that user.

Proxy Pattern…

Both the Adapter and the Proxy constitute a thin layer around an object.

The Adapter provides a different interface for an object,while the Proxy provides the same interface for the object, but interposes itself where it can save processing effort.

Object Composition

- Frequently programmers develop systems in which a component may be an individual object or it may represent a collection of objects.
- The problem that can arise out of having a single, simple interface to access all the objects in a collection or single individual objects and the client having no knowledge of its composition!
- Applications which needs to manipulate a "**primitive**" and "**composite**" objects, have to handle processing of a primitive objects in one way, and processing of a composite objects in differently. They have to query the "type" of each object before attempting to process

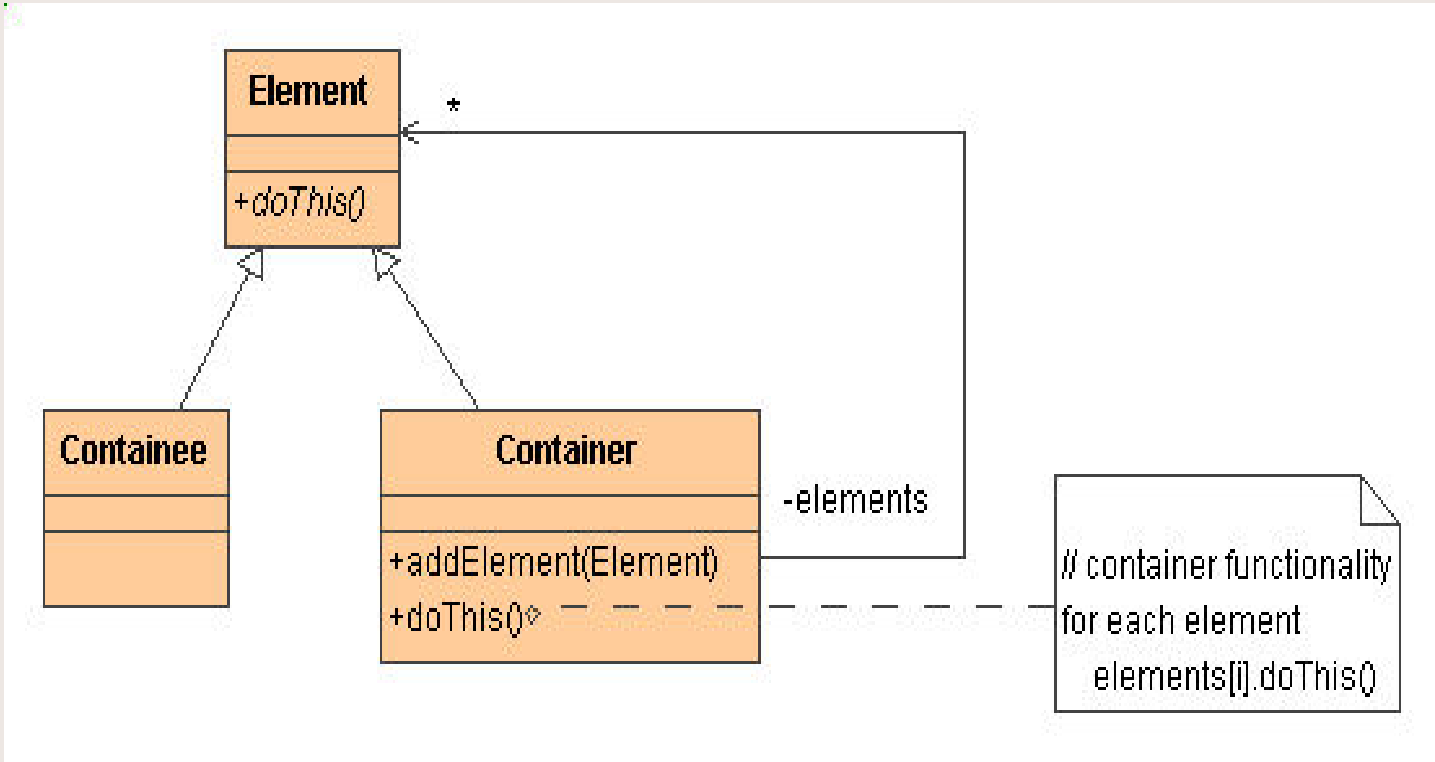
‘Composite’ is the solution…

- Composite is a collection of objects, any one of which may be either a composite, or just a primitive object.
- In tree nomenclature, some objects may be nodes with additional branches and some may be just leaves.
- Composite provides the same interface for primitive or composite object creation.

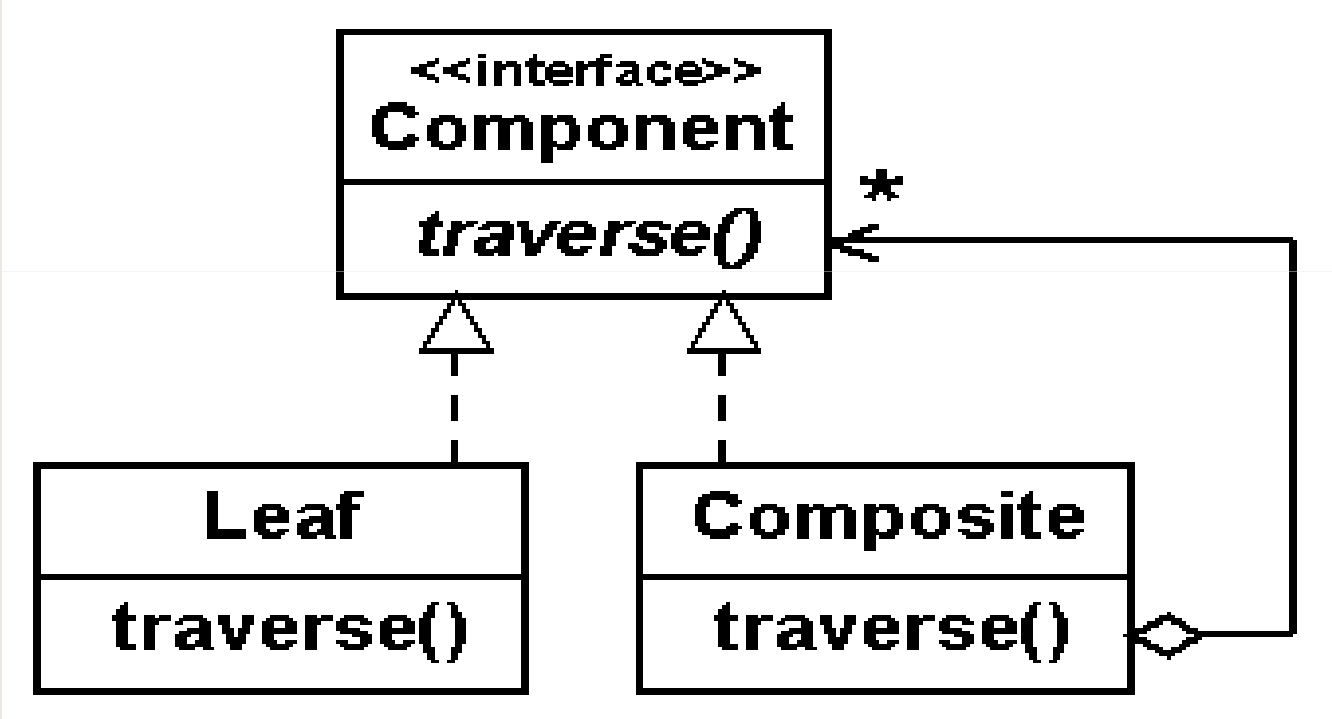
Composite and Client

- The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly
- Use this pattern whenever you have "composites that contain components, each of which could be a composite".

Composite class diagram



Composite Structure



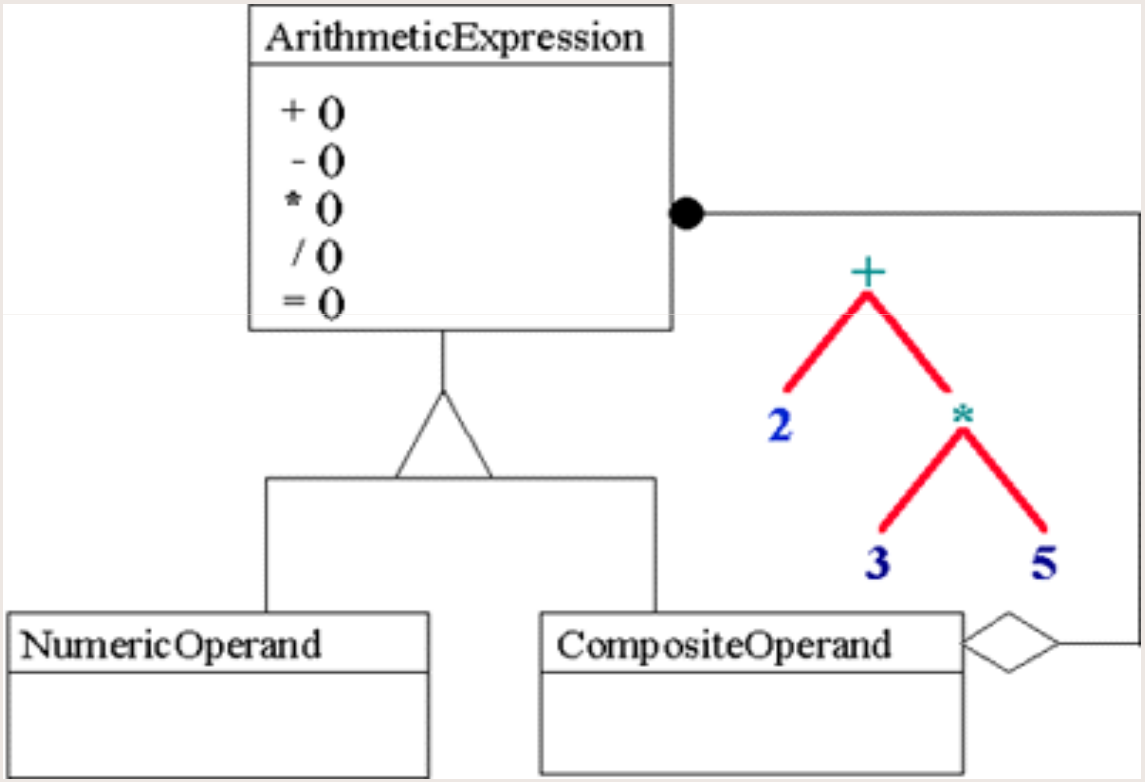
Composite implementation

- Define an abstract base class (*Component*) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects.
- Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as it manages its "children".
- Child management methods should defined in the abstract Component class.

Composite Example

- The Composite composes objects into tree structures and lets clients treat individual objects and compositions uniformly.
- Arithmetic expressions are Composites. An arithmetic expression consists of an operand, an operator (+ - * /), and another operand. The operand can be a number, or another arithmetic expression. Thus, 2 + 3 and (2 + 3) + (4 * 6) are both valid expressions

Composite Expression



Simple Interface to Complex System

- Sometimes the client needs a simplified interface to the overall functionality of a complex subsystem.
- This will reduce the learning curve necessary to successfully leverage the entire subsystem. It will also promote the decoupling of the subsystem from its potentially many clients

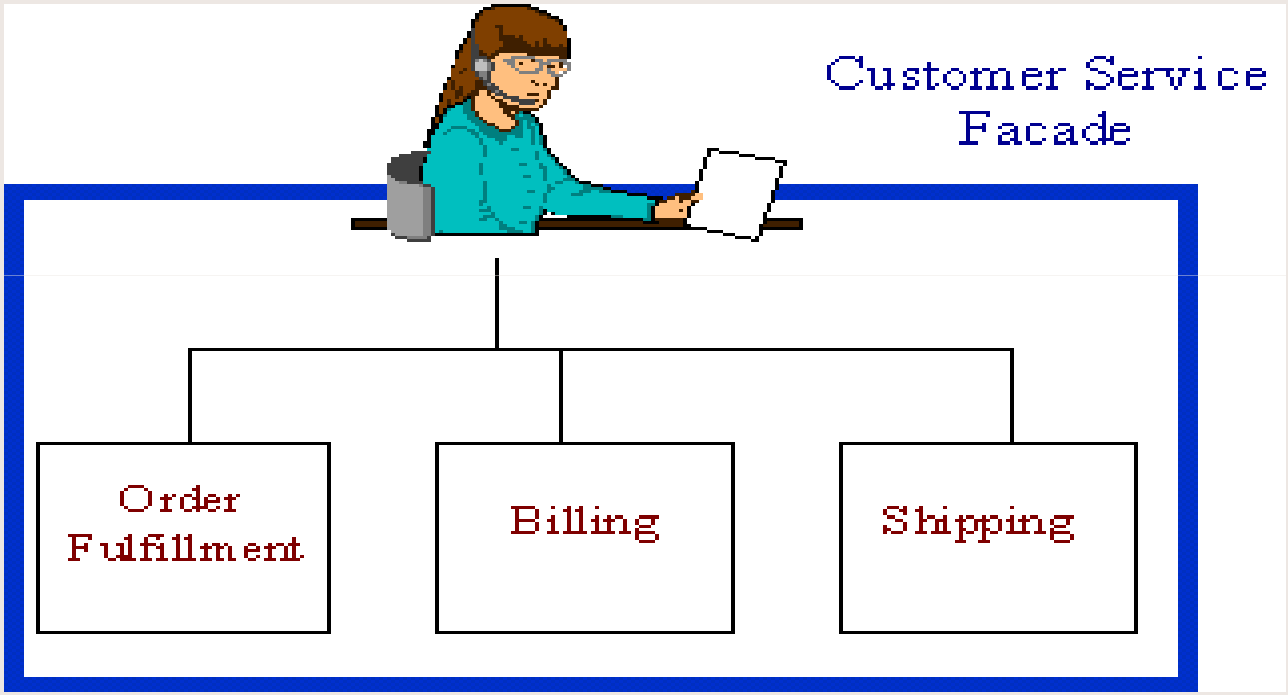
Solution is ‘Facade pattern ’

- The Façade pattern allows you to simplify this complexity by defining a unified, higher level interface to a subsystem that makes it easier to use.
- The Facade object is a fairly simple advocate or facilitator.

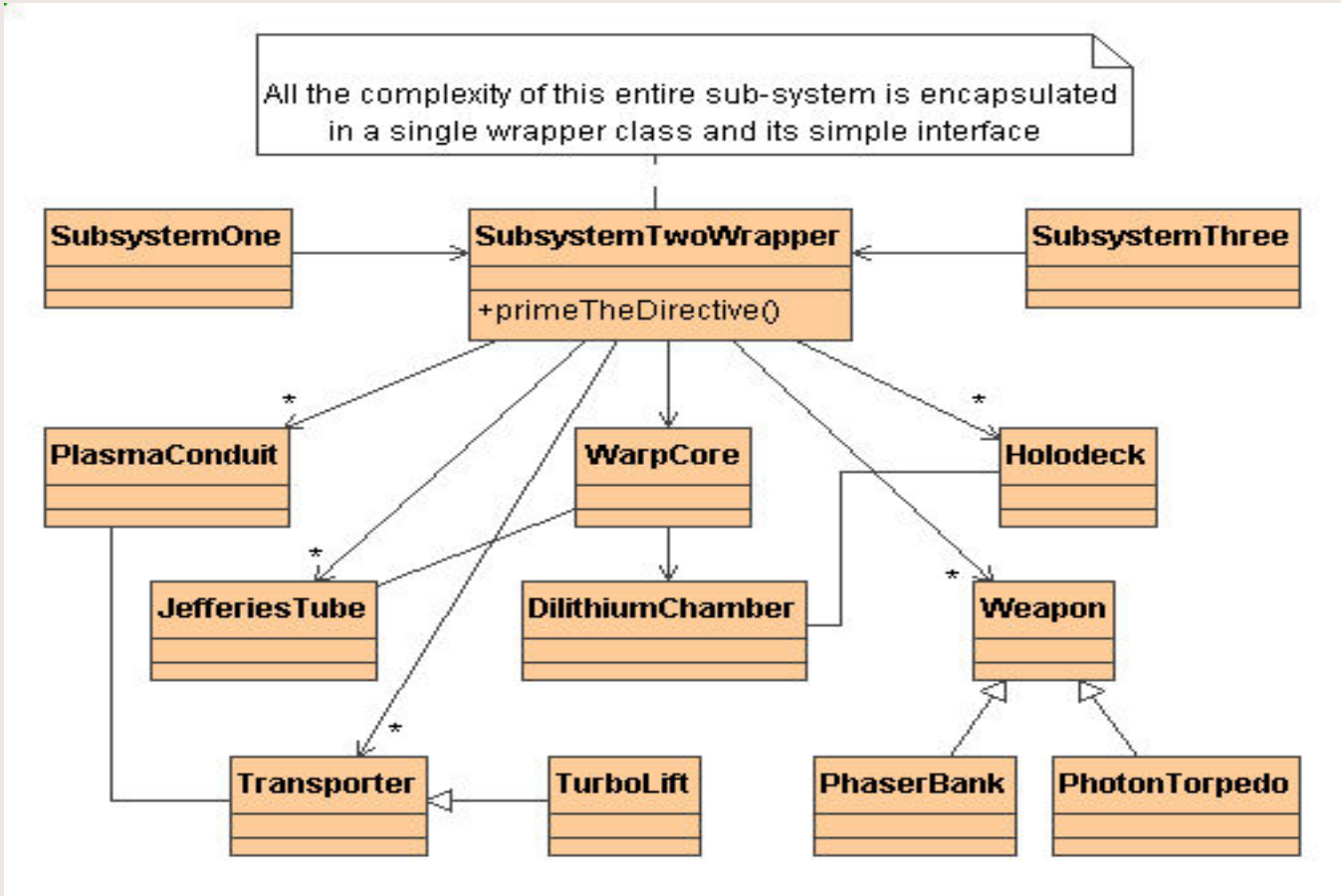
Facade example

- Consumers places the orders from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a ***Facade***, providing an interface to the order fulfillment department, the billing department, and the shipping department.

Customer Service Facade



Facade class diagram



Facade consequences

- Façade allows you to make changes in the underlying subsystems without requiring changes in the client code, and reduces the compilation dependencies.
- It provides a simpler programming interface for the general user. However, it does not prevent the advanced user from going to the deeper, more complex classes when necessary.
- Facade is always singleton

Complex system design

- Sometimes we need to design the objects down to the lowest levels of system "granularity" providing optimal flexibility.
- But this can be expensive in terms of performance and memory usage.
- So we need sharing of objects to support large numbers of fine-grained objects efficiently.

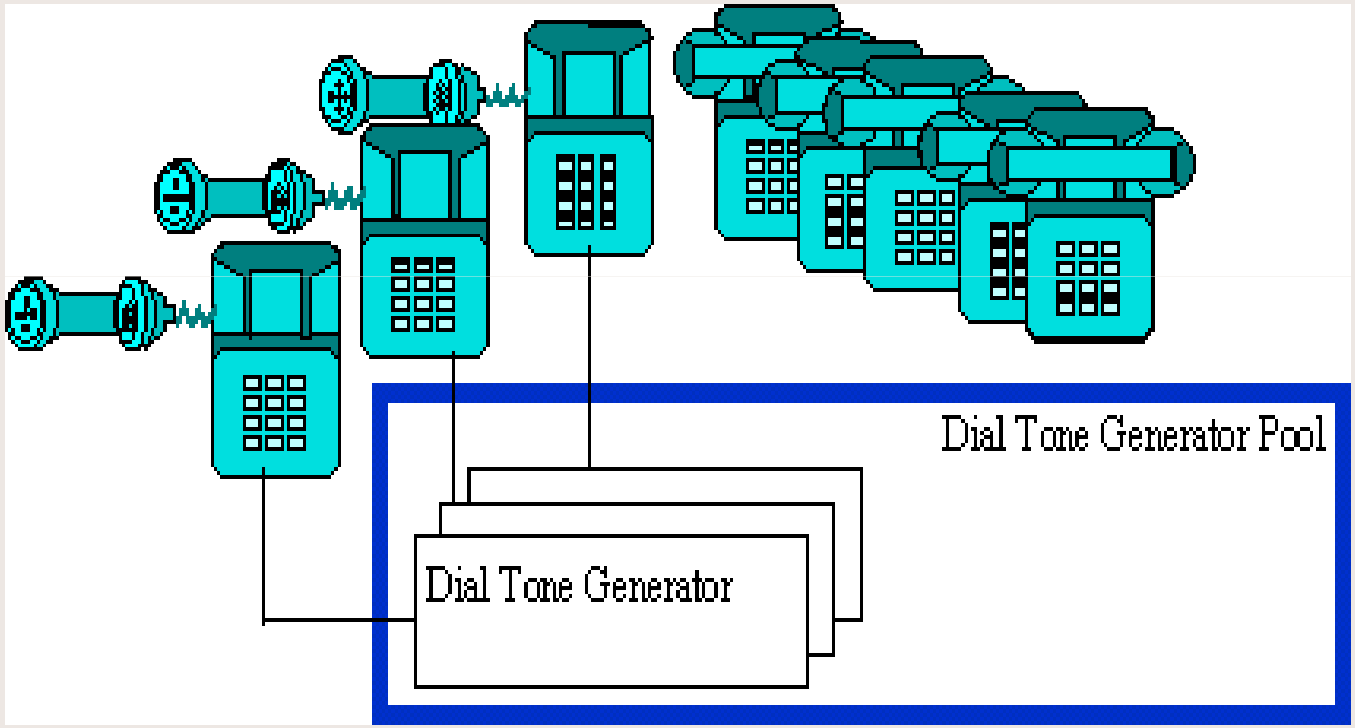
Flyweight sharing

- The ***Flyweight*** pattern describes how to share objects to allow their use at fine granularities without large cost.
- Each "flyweight" object is divided into two pieces: the state-dependent (extrinsic) part, and the state-independent (intrinsic) part.
- Intrinsic state is stored (shared) in the Flyweight object. Extrinsic state is stored or computed by client objects, and passed to the Flyweight when its operations are invoked.

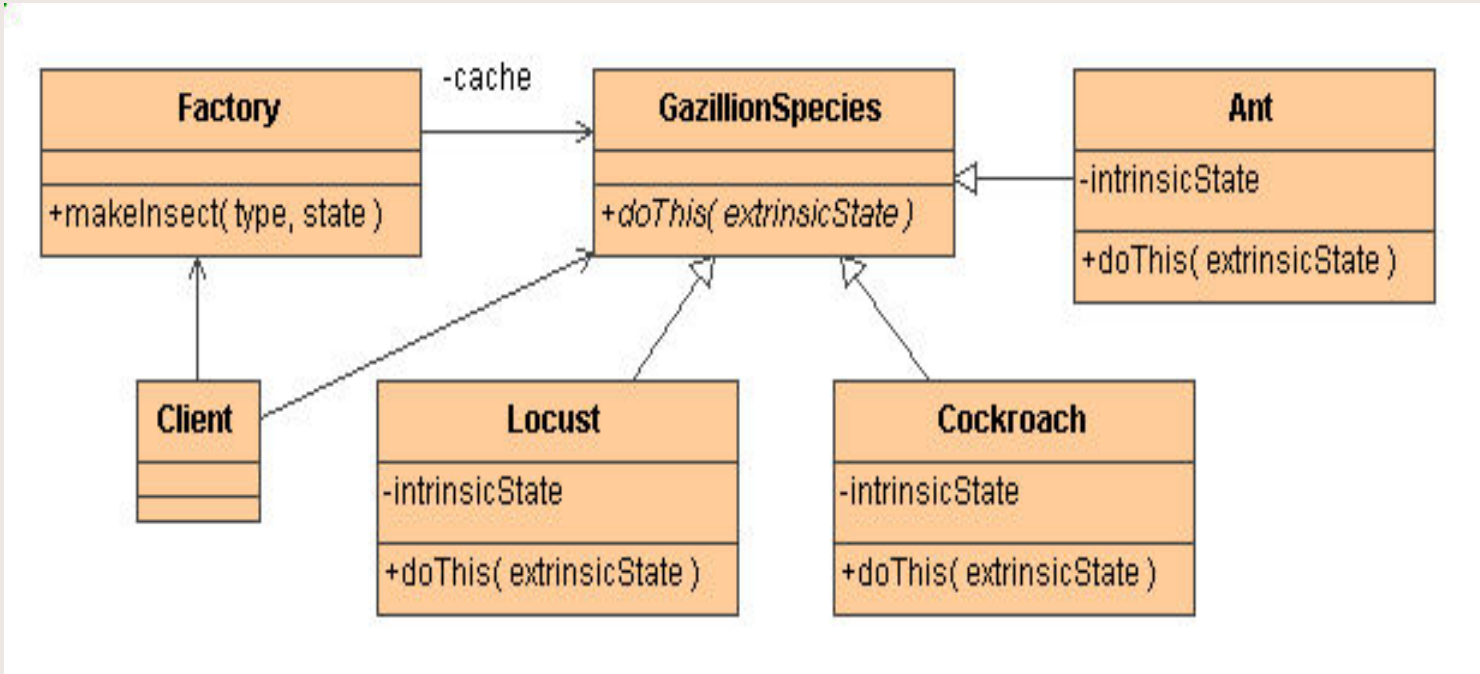
Flyweight example

- The Flyweight uses sharing to support large numbers of objects efficiently.
- In public switched telephone network,there are several resources such as dial tone generators, ringing generators, and digit receivers that must be shared between all subscribers. A subscriber is unaware of how many resources are in the pool when he or she lifts the handset to make a call.
- All that matters to subscribers is that a dial tone is provided, digits are received, and the call is completed.

PT Network



Flyweight class diagram



Flyweight another example

- Each character in a font is represented as a single instance of a *character* class, but the positions where the characters are drawn on the screen are kept as external data.
- So that there needs to be only one instance of *character* class for each character, rather than one for each appearance of that character.

Flyweight Factory

- Flyweights are sharable instances of a class. It might at first seem that each class is a *Singleton*, but in fact there might be a small number of instances, such as one for every character, or one for every icon type.
- The number of instances that are allocated must be decided as the class instances are needed, and this is usually accomplished with a ***FlyweightFactory*** class.
- This factory class usually is a Singleton, since it needs to keep track of whether or not a particular instance has been generated yet. It then either returns a new instance or a reference to one it has already generated.

Flyweight

- ***Flyweight*** shows how to make lots of little objects and ***Facade*** shows how to make a single object to represent an entire subsystem.
- ***Flyweight*** is often combined with Composite to implement shared leaf nodes.
- Flyweight explains when and how State objects can be shared

To inherit or not ?

- Sometimes when we want to add extra behavior or state to individual objects,inheritance is the choice but not always feasible(some languages don't provide multiple inheritance and the language providing it adds overhead to the application.
- We may want the extra behavior only at run-time. Inheritance makes it static and applies to an entire class.

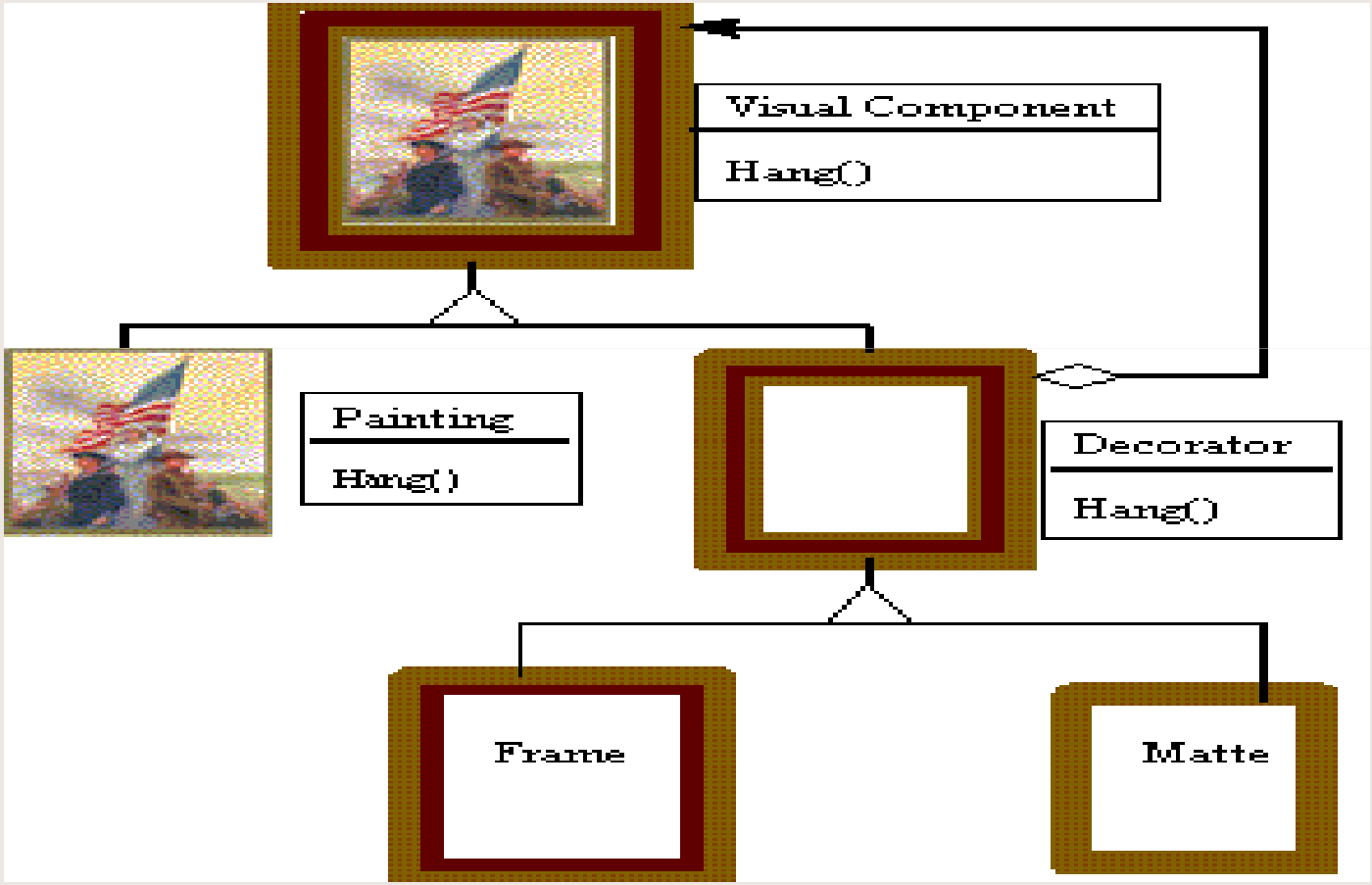
‘Decorator’ is the solution

- ***Decorators*** attach additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to sub classing for extending functionality.

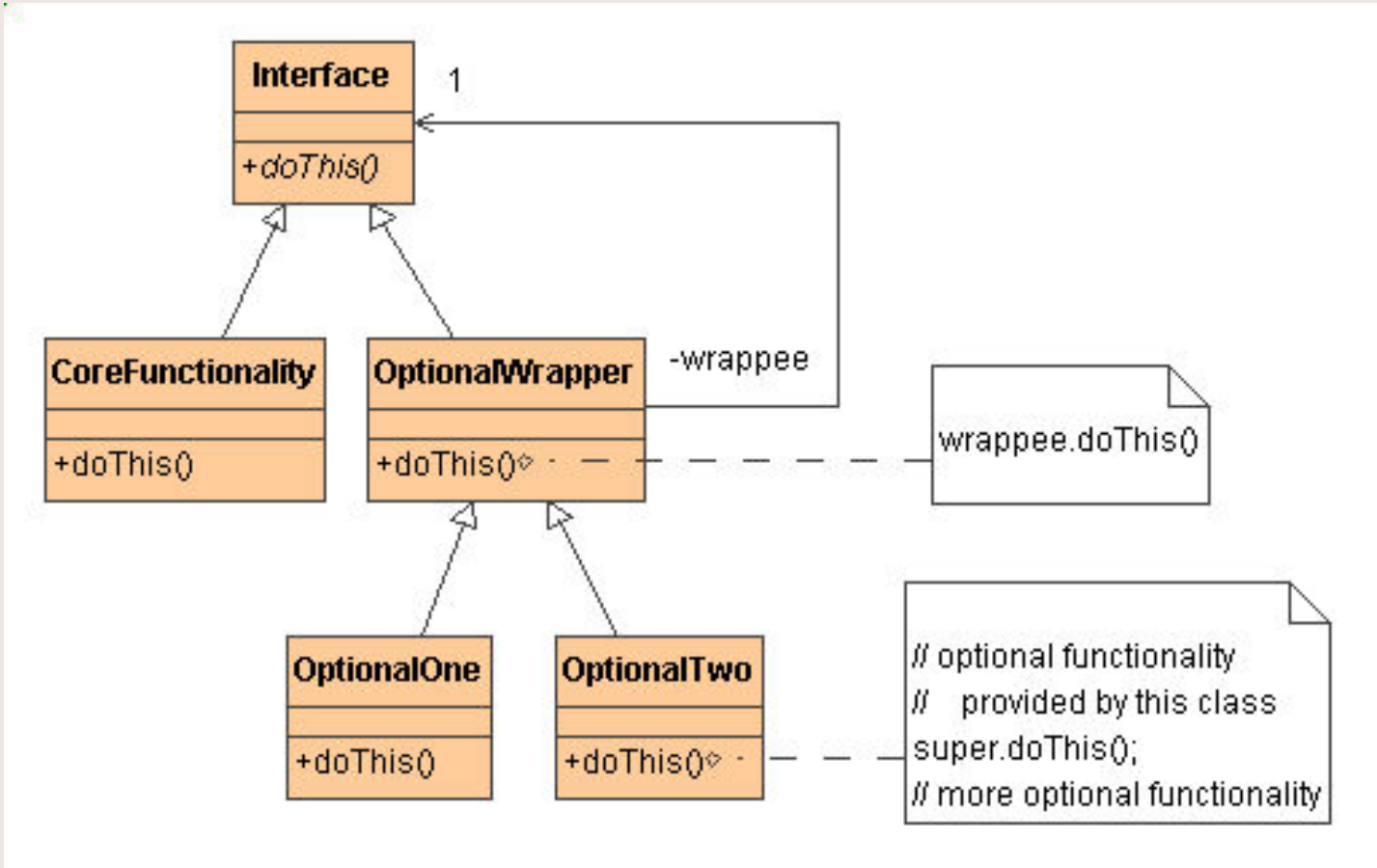
Decorator example

- Lots of ornaments are added to *Christmas* tree are examples of ***Decorators***. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look.
- The ornaments do not change the tree itself which is recognizable as a a Christmas tree regardless of particular ornaments used.

Frame as Decorator for paintings



Decorator class diagram



Decorator

- The solution involves encapsulating the original object inside an abstract wrapper interface.
- Both the ***decorator*** objects and the core object inherit from this abstract interface.
- The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.
- This pattern allows responsibilities to be added to an object, not methods to an object's interface.
- The interface presented to the client ***must remain constant*** as successive layers are specified.

Decorators,Adapters and Composites

- Some essential similarity among them…
- Adapters also seem to “decorate” an existing class.however, their function is to change the interface of one or more classes to one that is more convenient for a particular program. Decorators add methods to particular instances of classes, rather than to all of them.
- You could also imagine that a composite consisting of a single item is essentially a decorator, however, the intent is different.

Decorator consequences

- The inside of a decorator object it *‘hides’* core object's identity .
- Decorator enhances an object's responsibilities.
- Decorator supports recursive composition, which isn't possible with pure Adapters.

Decorator consequences - II

- Decorator and its enclosed component are not identical, thus tests for object type will fail.
- Decorators can lead to a system with “lots of little objects” that all look *alike* to the programmer trying to maintain the code. This can be a maintenance headache.