IOE 321 Software Design Patterns

Chapter IV

Structural Patterns

Presented by
Dr. Koppala Guravaiah
Assistant Professor
IIIT Kottayam

Syllabus

Indian Institute of Information Technology Kottayam

Structural Patterns

- Adaptor,
- Bridge,
- Composite,
- Decorator,
- Facade,
- Flyweight,
- Proxy

Implementation in various languages like Python, Java

Reference:

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns: Elements of Reusable Object oriented Software Addison-Wesley

Introduction

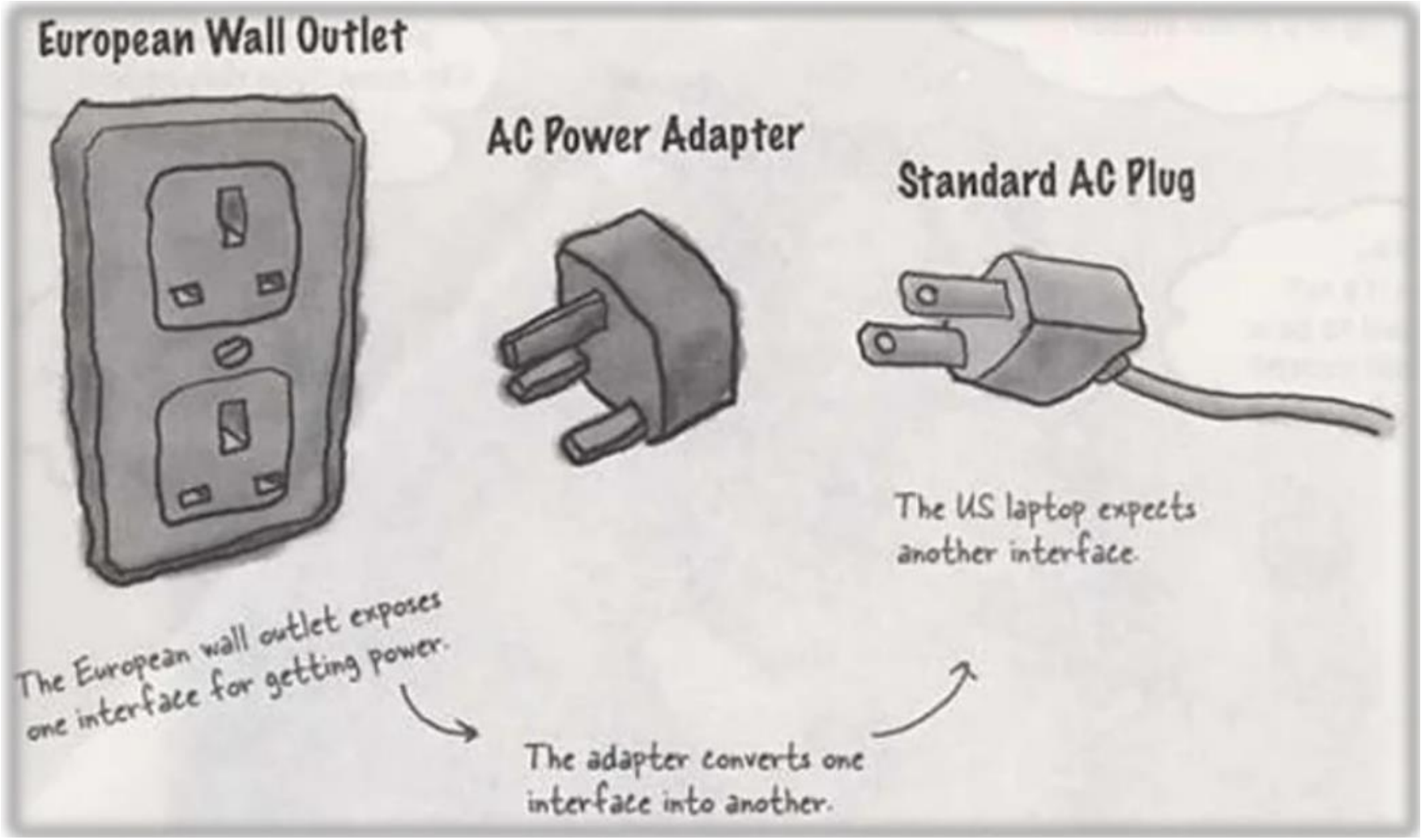
- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Structural *class* patterns use inheritance to compose interfaces or implementations.
- As a simple example, consider how multiple inheritance mixes two or more classes into one.
- The result is a class that combines the properties of its parent classes. This pattern is particularly useful for making independently developed class libraries work together.

Introduction … Contd.

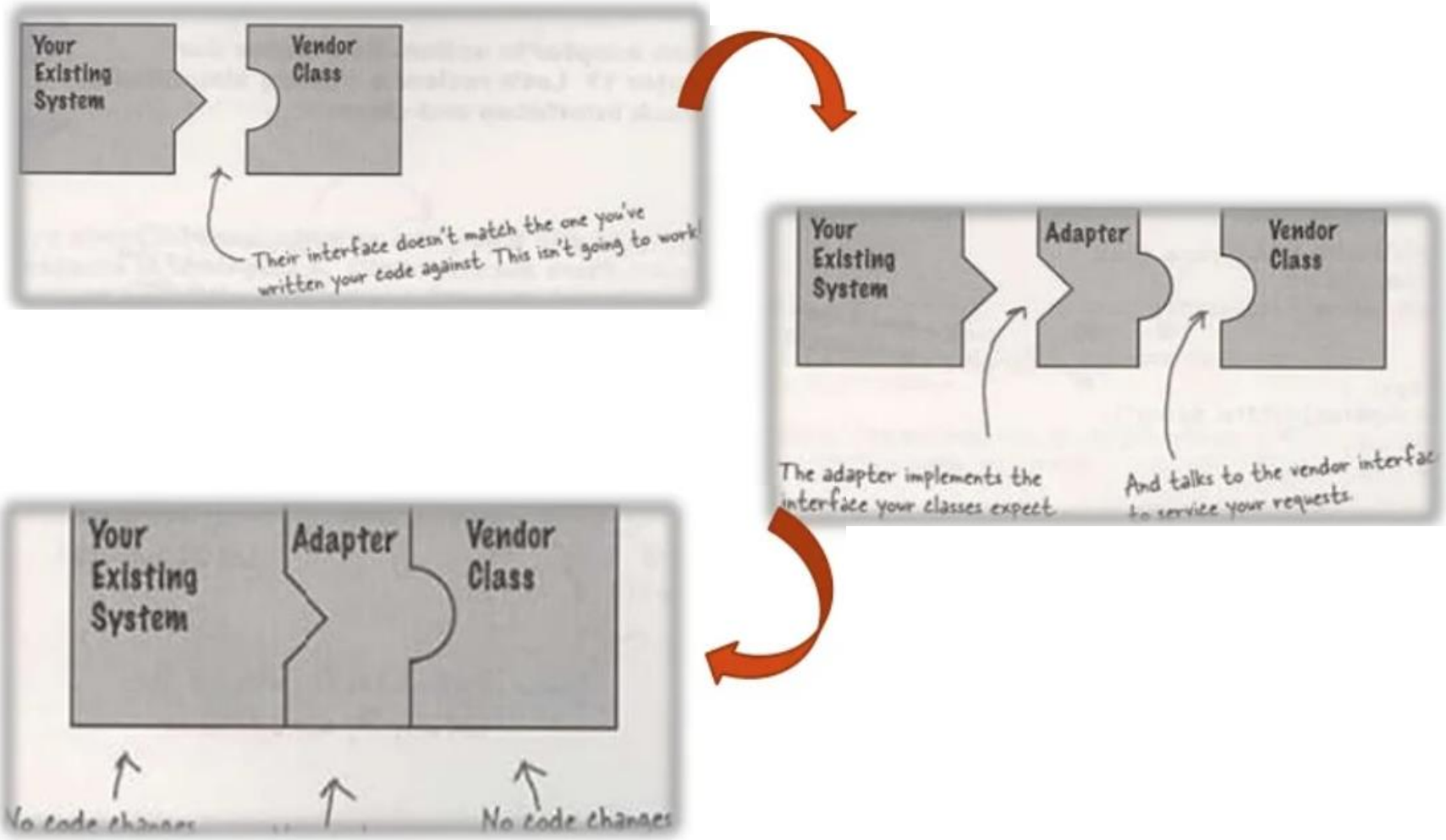
Indian Institute of Information Technology Kottayam

- Rather than composing interfaces or implementations, structural *object* patterns describe ways to compose objects to realize new functionality.
- The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition.

Adapter Pattern



Adapter Pattern … Contd.



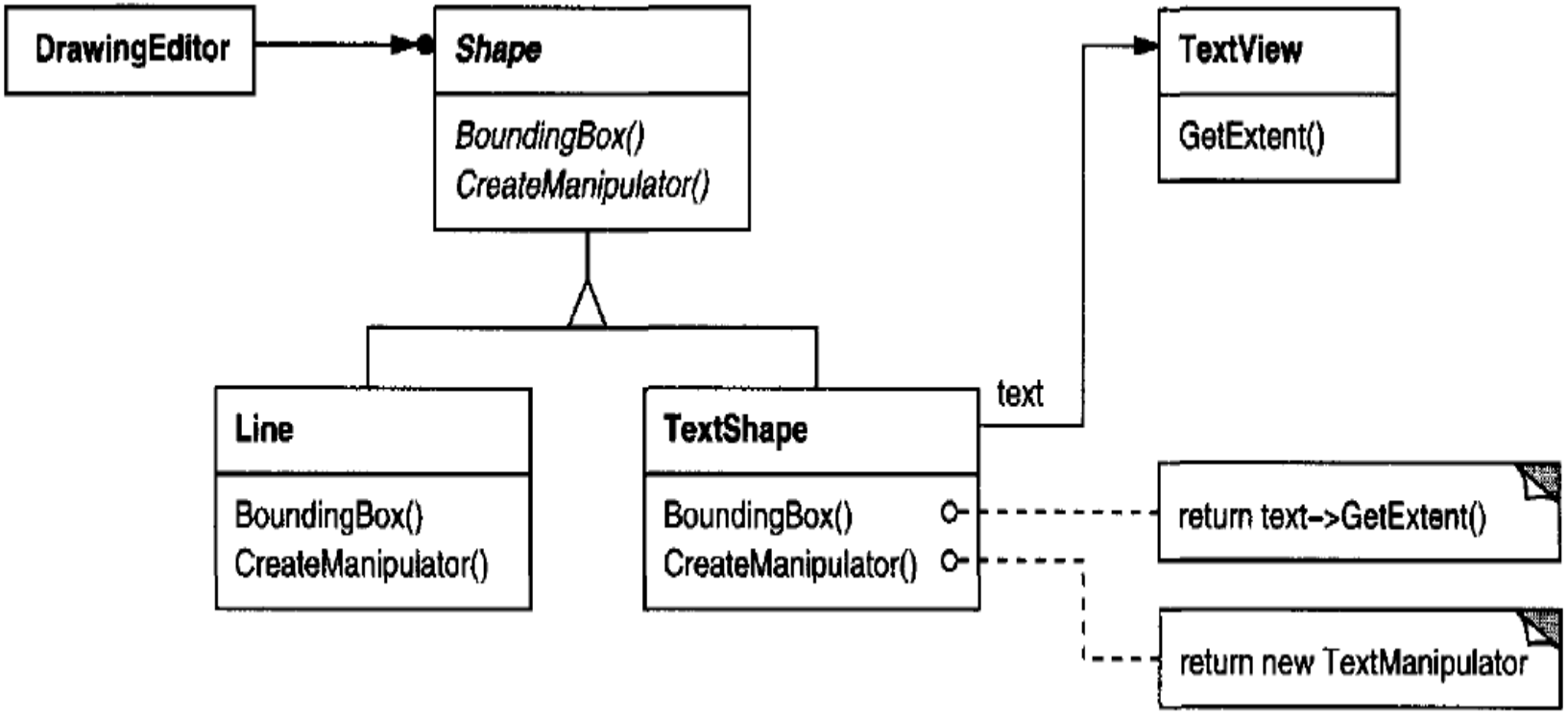
Adapter Pattern … Contd.

- Also Known As ***Wrapper***
- **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.
- **Motivation**
  - Sometimes a toolkit or class library cannot be used because its interface is incompatable with the interface required by an application.
  - We can not change the library interface since we may not have its source code
  - Even if we did have the source code, we probably should not change the library for each domain-specific application

Adapter pattern … Contd.

Indian Institute of Information Technology Kottayam

- Motivation

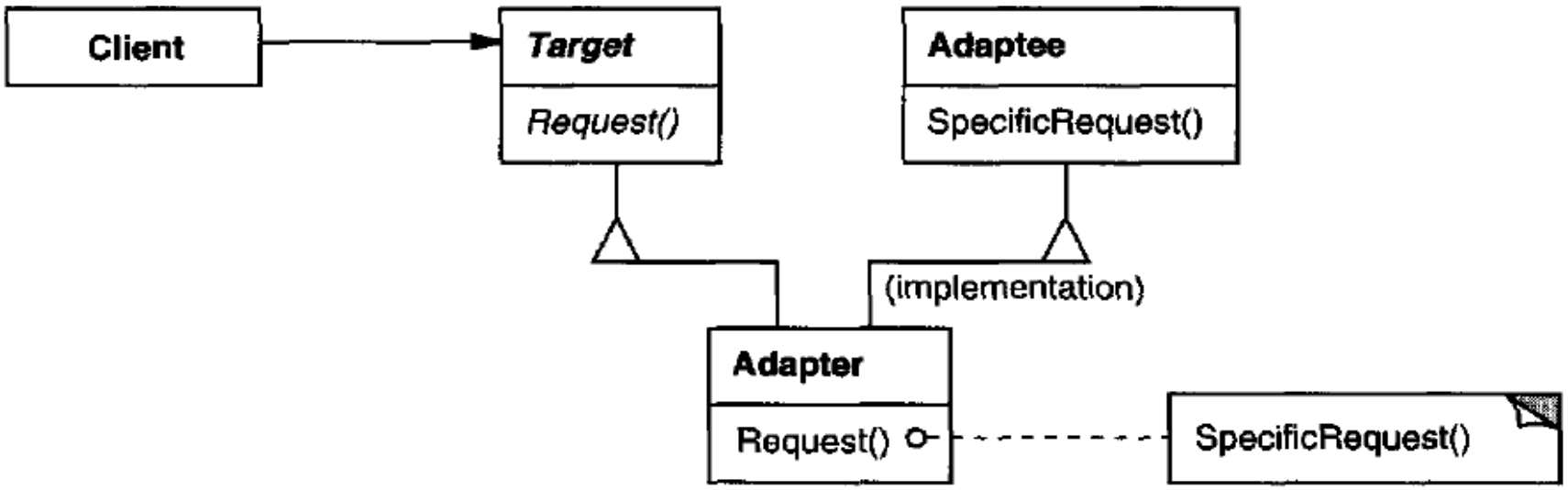


Adapter Pattern … Contd.

- **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 sub classing every one. An object adapter can adapt the interface of its parent class.

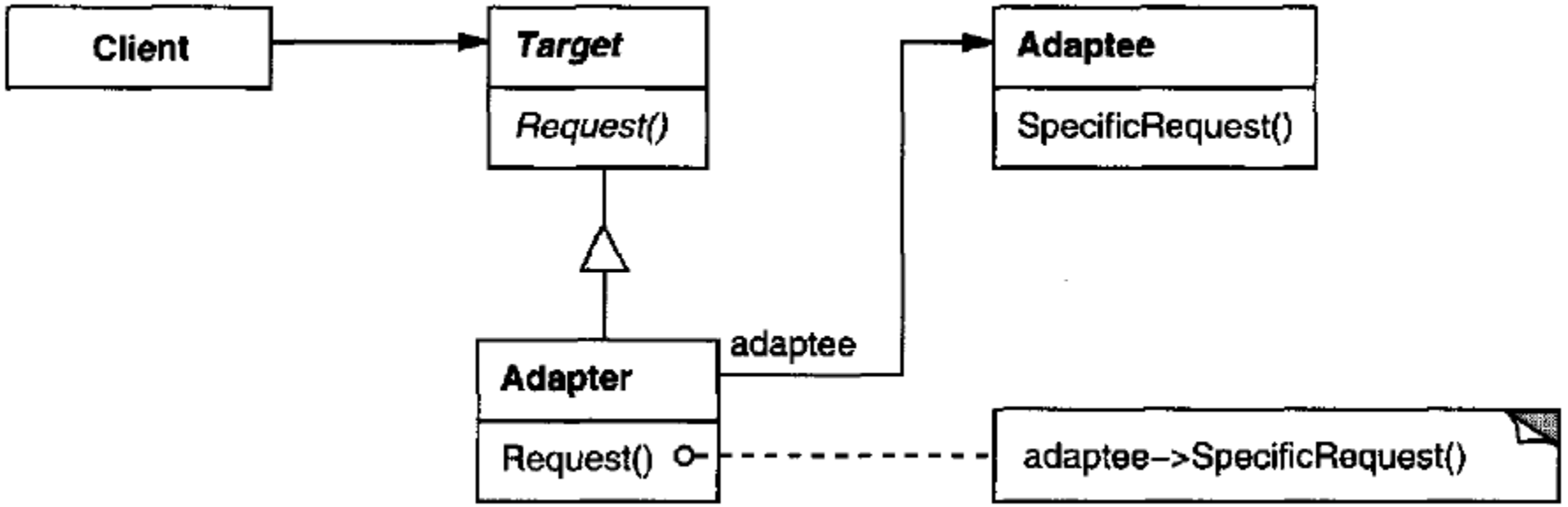
Adapter Pattern … Contd.

- **Structure**
  - A class adapter uses multiple inheritance to adapt one interface to another:



Adapter Pattern … Contd.

- **Structure**
  - An object adapter relies on object composition:



Adapter Pattern … Contd.

- **Participants**
  - Target (Shape)
    - defines the domain-specific interface that Client uses.
  - Client (DrawingEditor)
    - collaborates with objects conforming to the Target interface.
  - Adaptec (TextView)
    - defines an existing interface that needs adapting.
  - Adapter (TextShape)
    - adapts the interface of Adaptee to the Target interface.
- **Collaborations**
  - Clients call operations on an Adapter instance. In turn, the adapter calls Adaptec operations that carry out the request.

Adapter Pattern … Contd.

- **Consequences**
  - Class Adapter
    - Concrete Adapter class
    - Unknown Adaptee Subclasses
  - Object Adapter
    - Adapter can service many different adaptees
    - May require the creation of Adaptee subclasses and referring those objects

Adapter Pattern … Contd.

Indian Institute of Information Technology Kottayam

- **Implementation**
  - How much adapting should be done?
    - Simple interface conversion that just changes operation names and order of arguments
    - Totally different set of operations
  - Does the adapter provide two-way transparency?
    - A two-way adapter supports both the target and the Adaptee interface. It allows an adapted object (Adaptor) to appear as an adaptee object or a Target object

Bridge Pattern

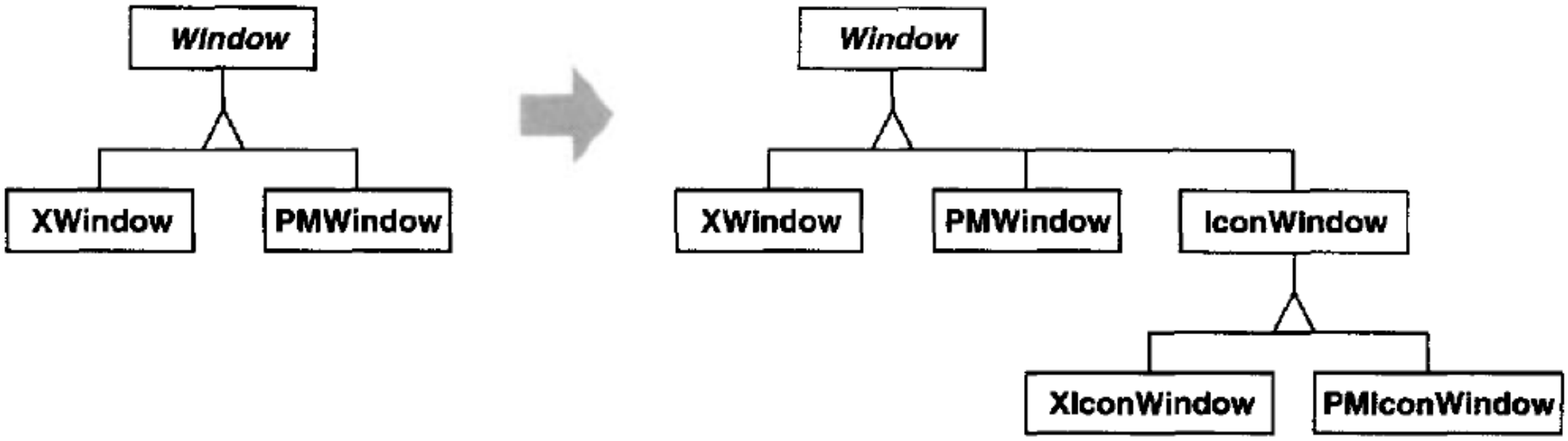
Indian Institute of Information Technology Kottayam

- Also Known As ***Handle/Body***
- **Intent**
  - Decouple an abstraction from its implementation so that the two can vary independently.
  - Multiple dependent implementations
  - Single Independent Interface
- **Motivation**
  - When an abstraction can have one of several possible implementations, inheritance is used to accommodate them.
  - Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.
  - But this approach isn't always flexible enough.
  - An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.

Bridge Pattern … Contd.

Indian Institute of Information Technology Kottayam

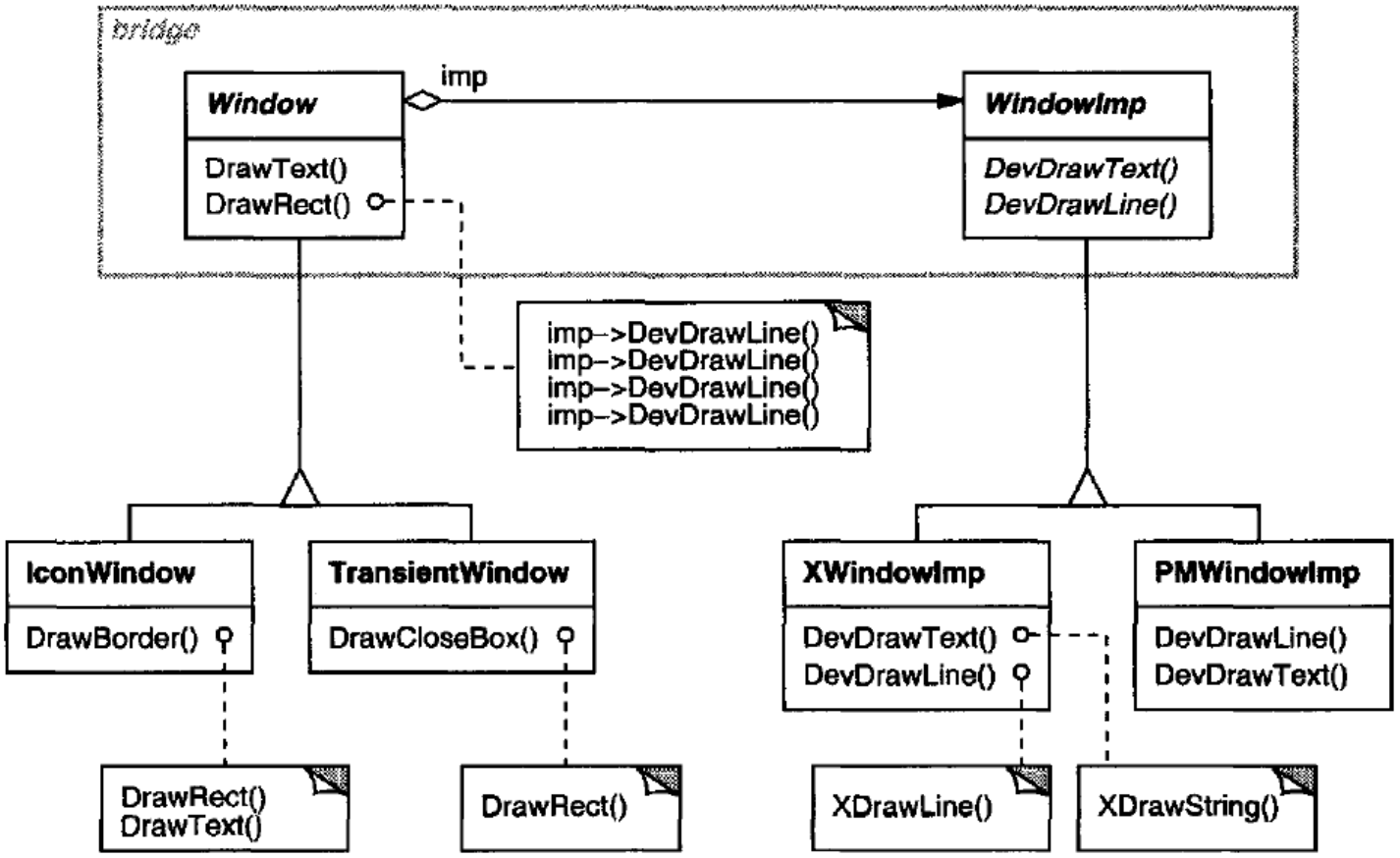
- ***Motivation***



Bridge Pattern … Contd.

Indian Institute of Information Technology Kottayam

- ***Motivation***



Bridge Pattern … Contd.

Indian Institute of Information Technology Kottayam

- **Applicability**

Use the Bridge pattern when

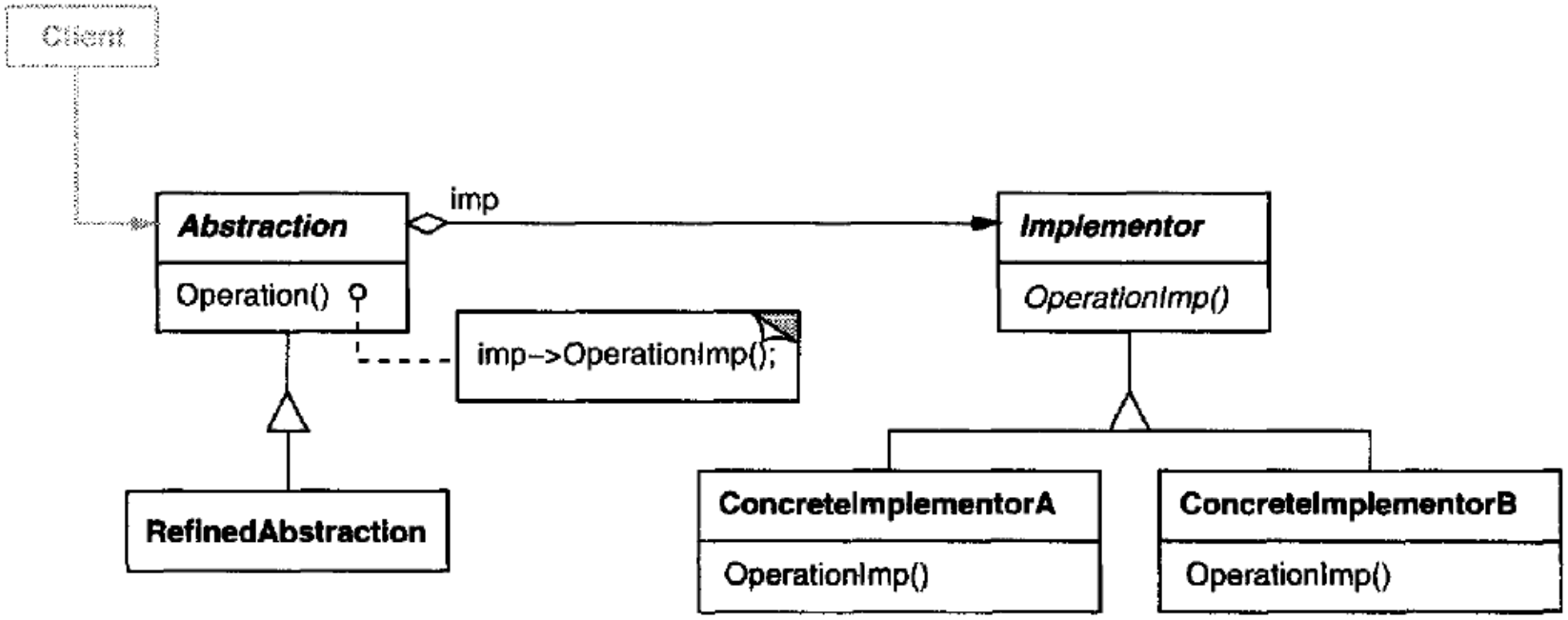
- you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- Both the abstractions and their implementations should be extensible by sub classing. In this case, the Bridge pattern lets you combine the 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.

Bridge Pattern … Contd.

- **Applicability**
- (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 have a proliferation of classes as shown earlier in the first Motivation diagram. Such a class hierarchy indicates the need for splitting an object into two parts. Rumbaugh uses the term "nested generalizations" to refer to such class hierarchies.
- you want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client. A simple example is Coplien's String class, in which multiple objects can share the same string representation (StringRep).

Bridge Pattern … Contd.

- ***Structure***



Bridge Pattern … Contd.

Indian Institute of Information Technology Kottayam

- **Participants**
- **Abstraction** (Window)
  - defines the abstraction's interface.
  - maintains a reference to an object of type Implementor.
- **RefinedAbstraction** (IconWindow)
  - Extends the interface defined by Abstraction.
- **Implementor** (WindowImp)
  - Defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operation based on these primitives.
- **ConcreteImplementor** (XWindowImp, PMWindowImp)
  - implements the Implement or interface and defines its concrete implementation.

Bridge Pattern … Contd.

- **Collaborations**
  - Abstraction forwards client requests to its Implementor object.
- **Consequences**
  - The Bridge pattern has the following consequences:
  - *Decoupling interface and implementation.* An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.
  - *Improved extensibility.* You can extend the Abstraction and Implementor hierarchies independently.
  - *Hiding implementation details from clients.* You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism (if any).

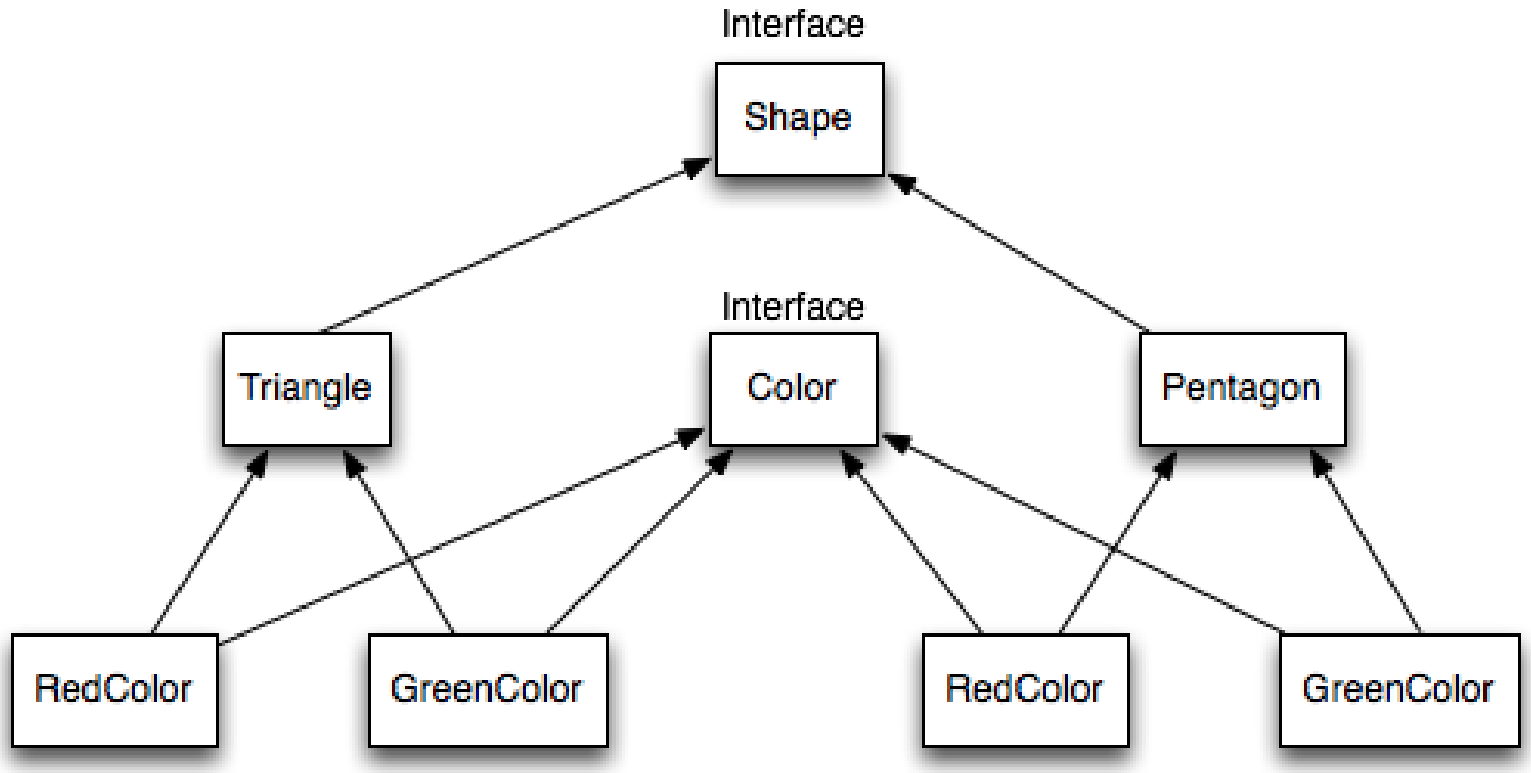
Bridge Pattern … Contd.

- **Implementation**
- Consider the following implementation issues when applying the Bridge pattern:
  - *Only one Implementor*
  - *Creating the right Implementor object*
  - *Sharing implementors*
  - *Using multiple inheritance*

Bridge Pattern … Contd.

- **Related Patterns**
- An AbstractFactory can create and configure a particular Bridge.
- The Adapter pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed.
- Bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

Bridge Pattern … Contd.



- Notice the bridge between Shape and Color interfaces and use of composition in implementing the bridge pattern.

Composite Pattern

- Composite pattern is one of the Structural design pattern.
- Composite design pattern is used when we have to represent a part-whole hierarchy.
- When we need to create a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern.
- Lets understand it with a real life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc. When we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations.

Composite Pattern … Contd.

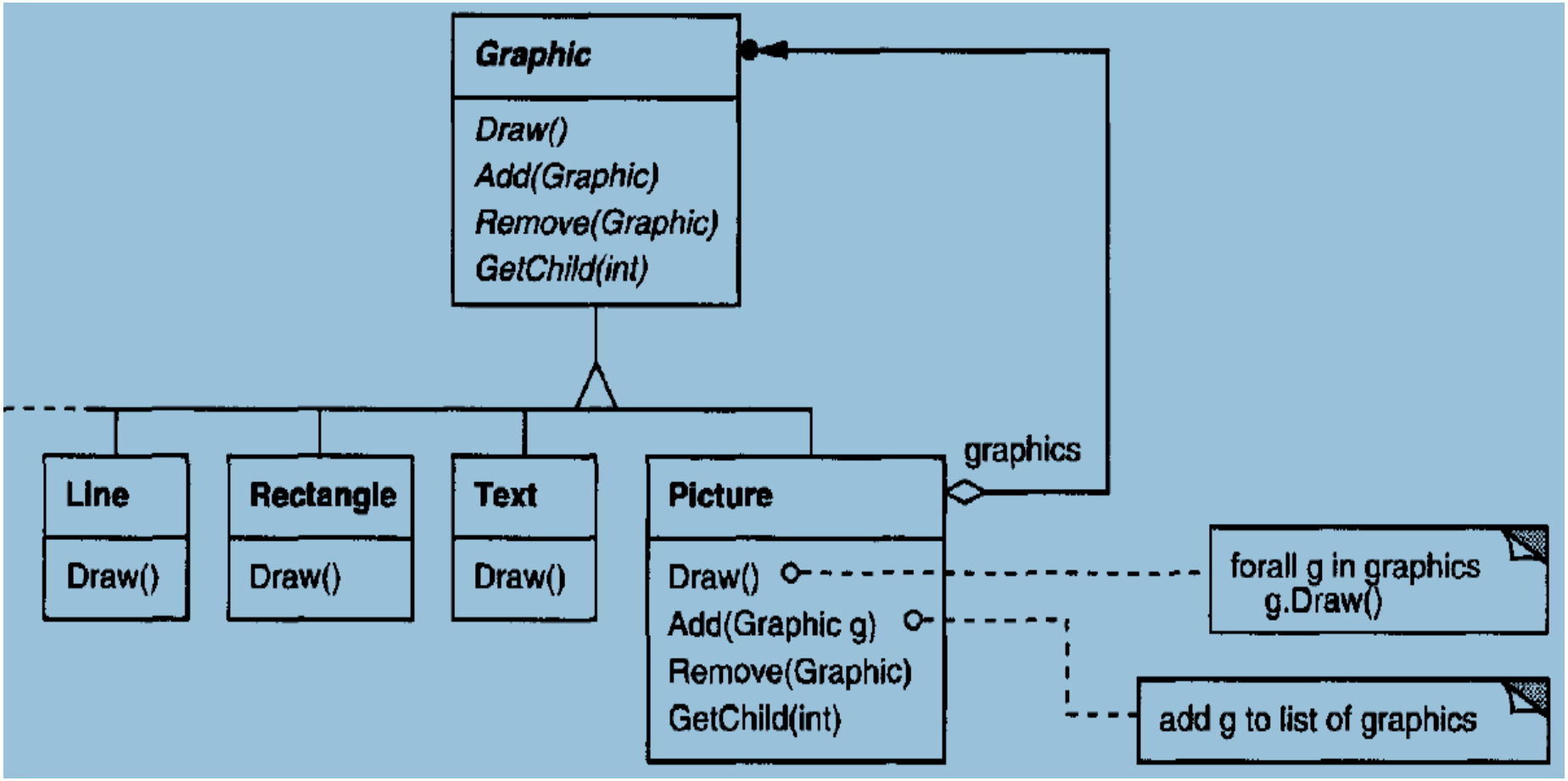
- Composite Pattern consists of following objects.
  - **Base Component** – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an ***abstract class*** with some methods common to all the objects.
  - **Leaf** – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn't have references to other Components.
  - **Composite** – It consists of leaf elements and implements the operations in base component.
- Composite pattern should be applied only when the group of objects should behave as the single object.

Composite Pattern… Contd.

- **Intent**
  - Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  - Recursive composition
  - "Directories contain entries, each of which could be a directory."
  - 1-to-many "has a" up the "is a" hierarchy
- **Problem**
  - Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

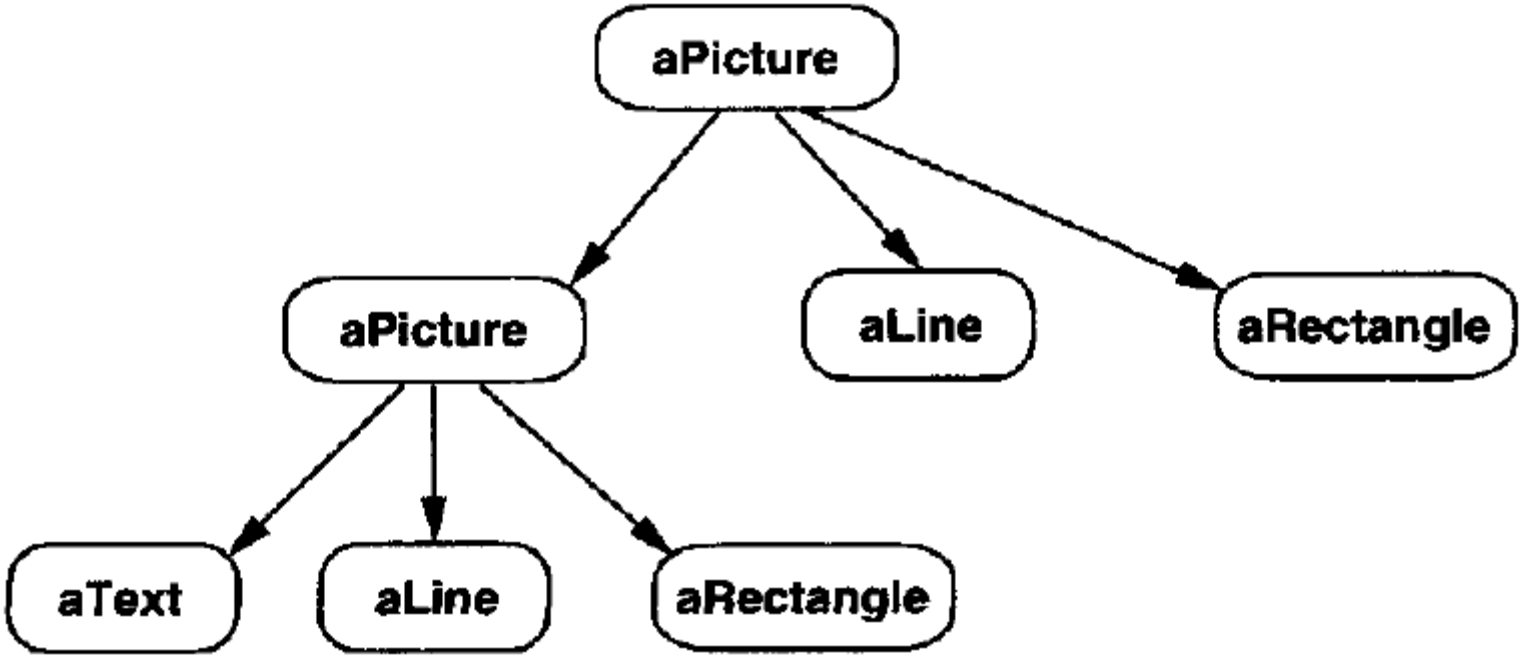
Composite Pattern… Contd.

Indian Institute of Information Technology Kottayam



Composite Pattern… Contd.

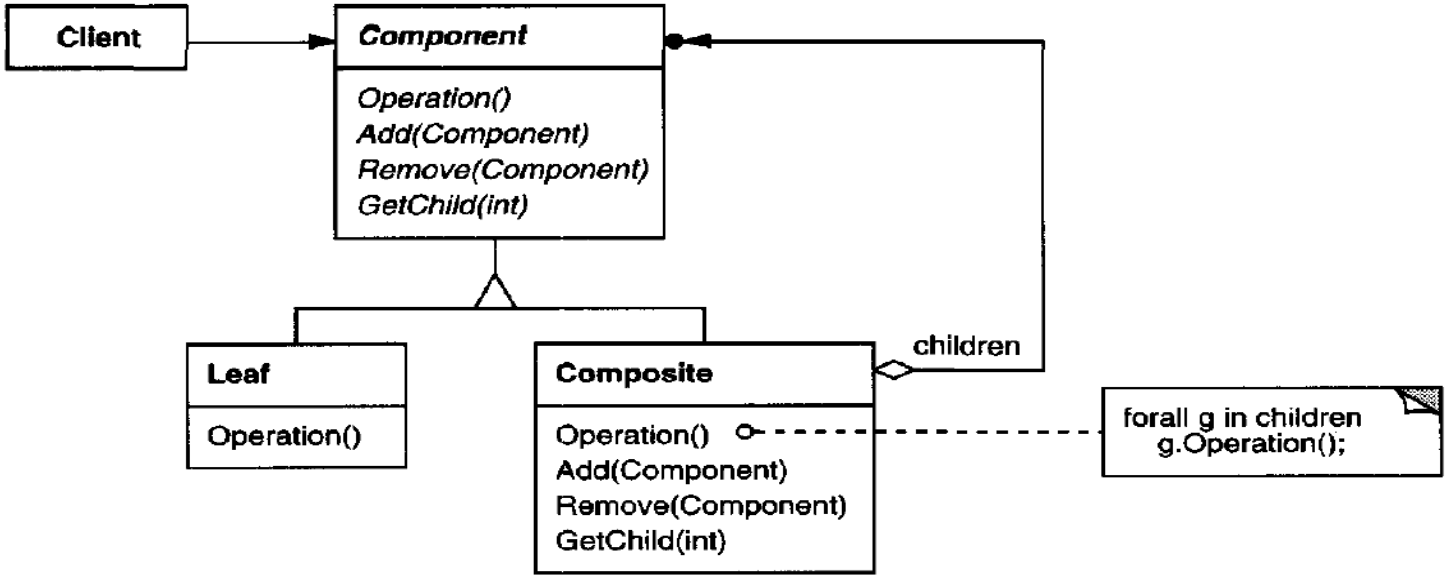
- A typical composite object structure of recursively composed Graphic objects:



Composite Pattern… Contd.

Indian Institute of Information Technology Kottayam

- **Applicability**
  - Use the Composite pattern when
    - you want to represent part-whole 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 Pattern… Contd.

- **Participants**
  - **Component (Graphic)**
    - declares the interface for objects in the composition.
    - implements default behavior for the interface common to all classes, as appropriate.
    - declares an interface for accessing and managing its child components.
    - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
  - **Leaf (Rectangle, Line, Text, etc.)**
    - represents leaf objects in the composition. A leaf has no children.
    - defines behavior for primitive objects in the composition.

Composite Pattern… Contd.

- **Participants**
  - **Composite (Picture)**
    - defines behavior for components having children.
    - stores child components.
    - implements child-related operations in the Component interface.
  - **Client**
    - manipulates objects the composition through the Component interface.
- **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.

Composite Pattern… Contd.

Consequences

The Composite pattern

- defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively. Wherever client code expects a primitive object, it can also take a composite object.
- makes the client simple.
- makes it easier to add new kinds of components.
- can make your design overly general. The is advantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Composite Pattern… Contd.

- **Implementation**
  - There are many issues to consider when implementing the Composite pattern:
    - *Explicit parent references.*
    - *Sharing components.*
    - *Maximizing the Component interface.*
    - *Declaring t he child management operations.*
    - *Should Component implement a list of Components?*
    - *Child ordering.*
    - *Caching to improve performance*
    - *Who should delete components?*
    - *What's the best data structure for storing components?*

Composite Pattern… Contd.

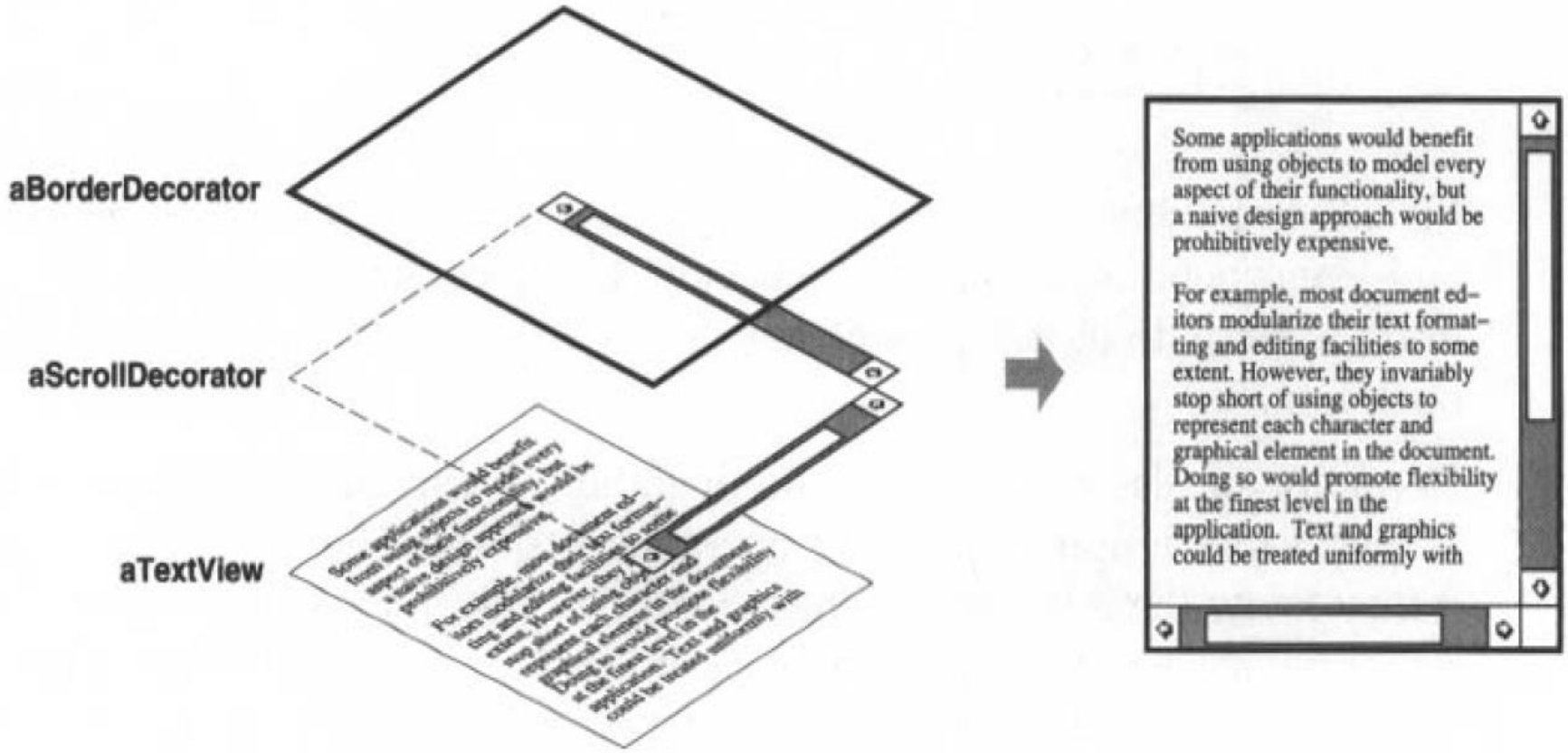
- ***Applications***
  - Composite design pattern can be used to create a tree like structure.
  - Great example of Composite pattern in java and used a lot in Swing: **java.awt.Container#add(Component)**

Decorator Pattern

Indian Institute of Information Technology Kottayam

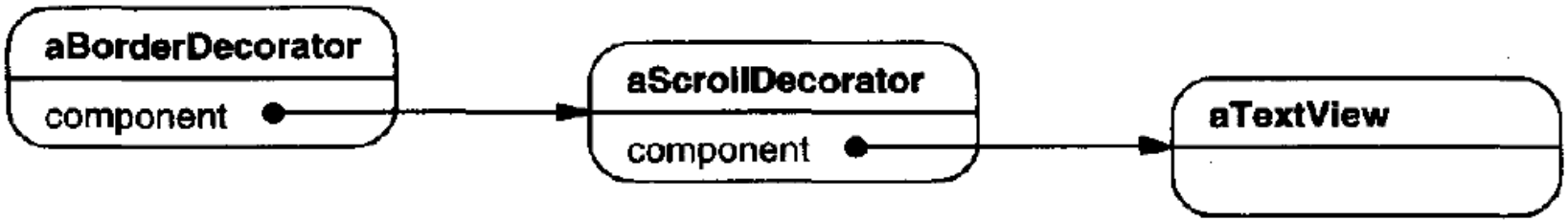
- **Also Known As** Wrapper
- **Intent**
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality.
- **Motivation**
  - A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a decorator.

Decorator Pattern … Contd.



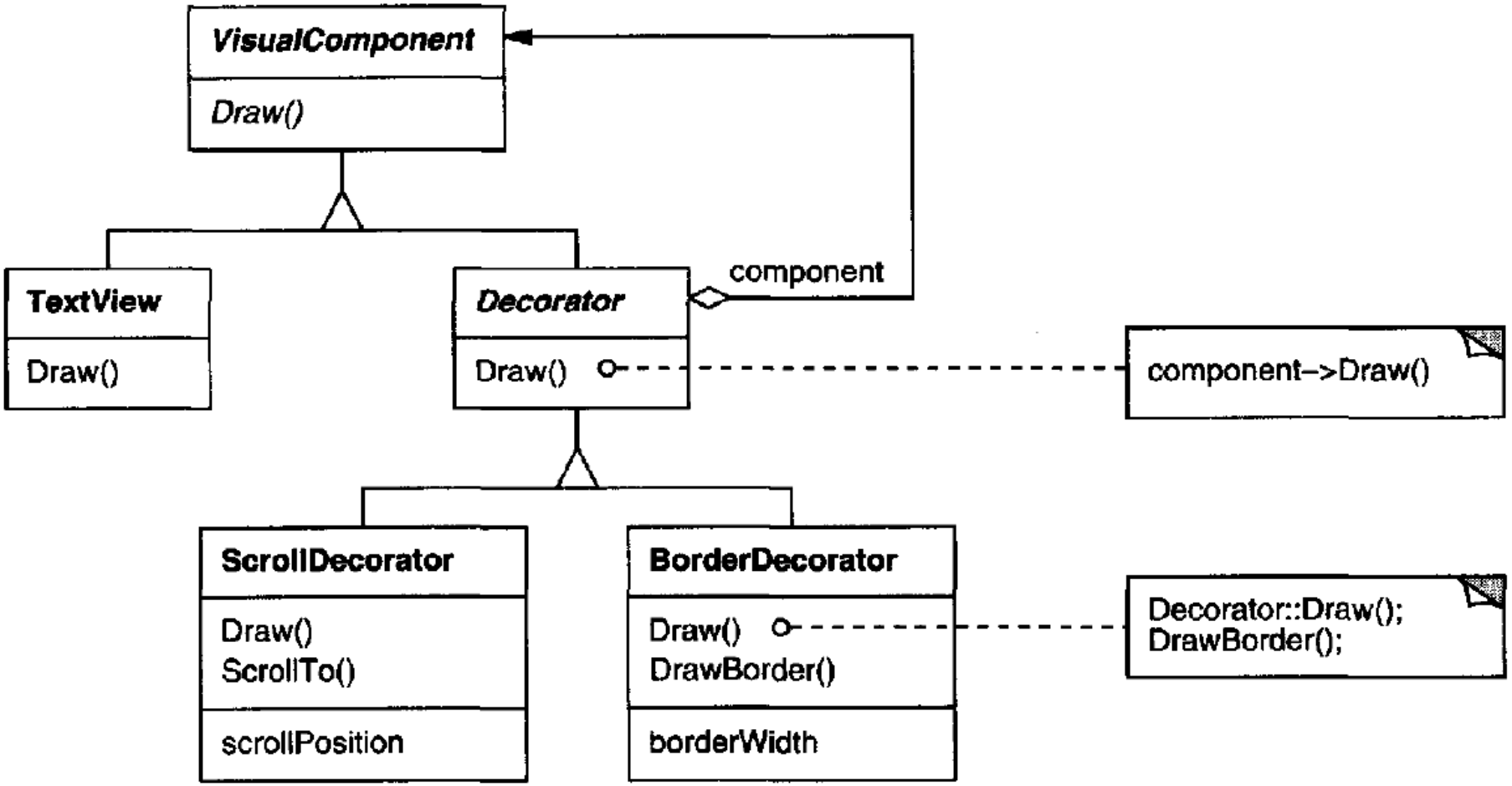
Decorator Pattern … Contd.

- The following object diagram shows how to compose a TextView object with BorderDecorator and ScrollDecorator objects to produce a bordered, scrollable text view:



Decorator Pattern … Contd.

Indian Institute of Information Technology Kottayam



Decorator Pattern … Contd.

- **Applicability**

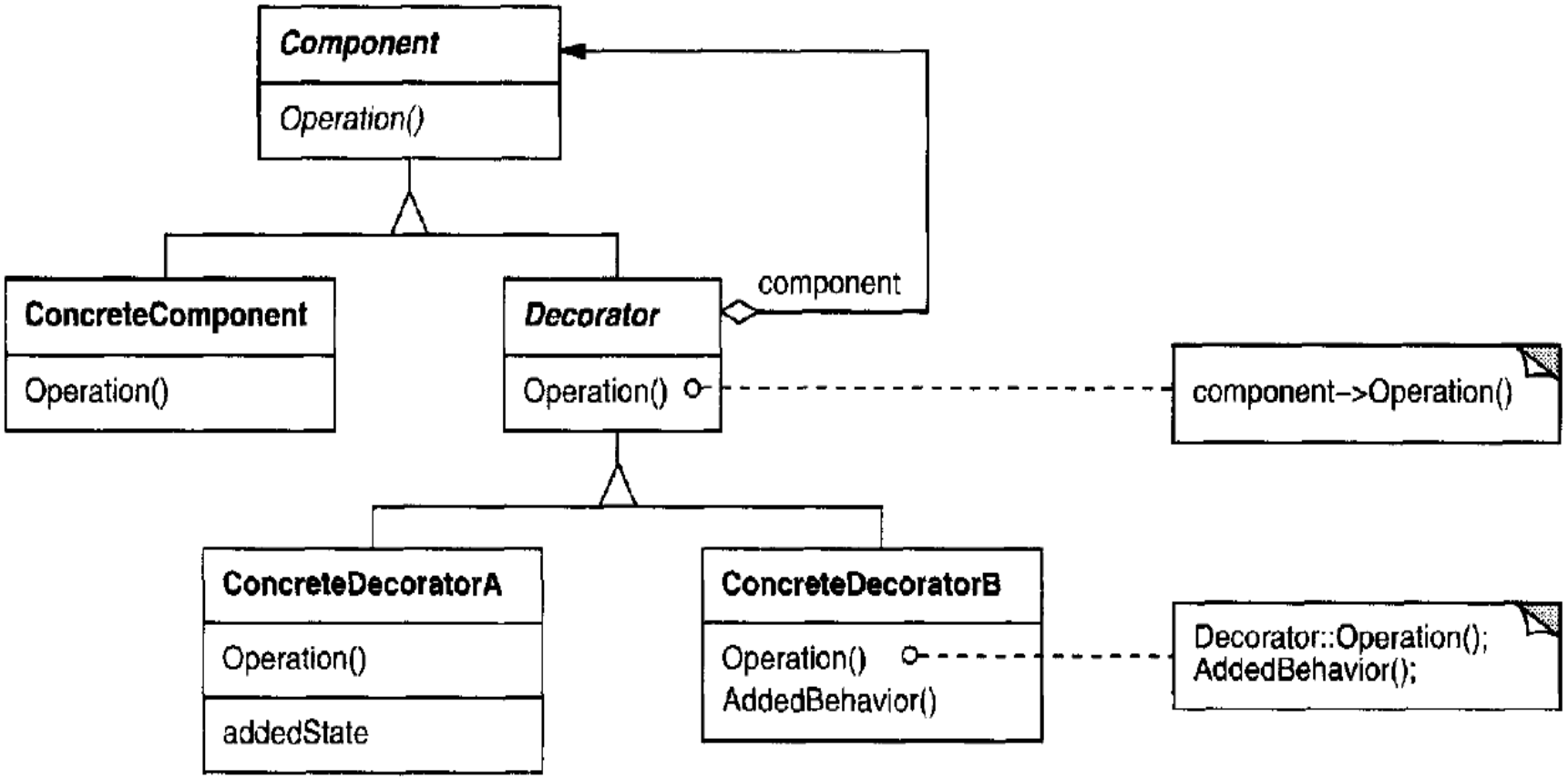
Use Decorator

- to add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
- for responsibilities that can be withdrawn.
- when extension by sub-classing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support ever y combination. Or a class definition maybe hidden or otherwise unavailable for subclassing.

Decorator Pattern … Contd.

Indian Institute of Information Technology Kottayam

- **Structure**



Decorator Pattern … Contd.

- **Consequences**
  - The Decorator pattern has at least two key benefits and two liabilities:
    - *More flexibility than static inheritance*
    - *Avoids feature-laden classes high up in the hierarchy*
    - *A decorator and its component aren't identical.*
    - *Lots of little objects.*

Decorator Pattern … Contd.

- **Implementation**
  - Several issues should be considered when applying the Decorator pattern:
    - *Interface conformance.*
    - *Omitting the abstract Decorator class*
    - *Keeping Component classes lightweight*
    - *Changing the skin of an object versus changing its guts*

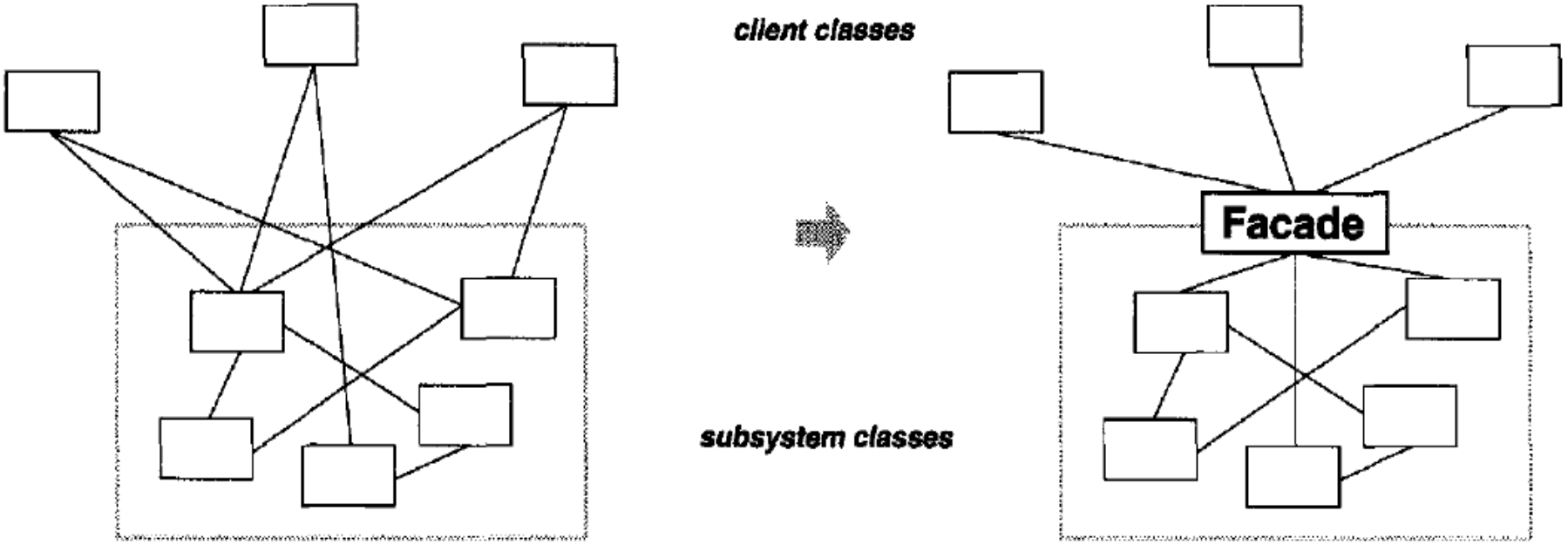
Key Points:

- Helpful in providing runtime modification abilities hence more flexible. Its easy to maintain and extend when the number of choices are more.
- Used in Java IO classes such as FileReader, BufferedReader etc.
- Disadvantage: uses a lot of similar kind of objects (decorators)

Facade Pattern

- **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.
- **Motivation**
  - Structuring a system into subsystems helps reduce complexity.
  - A common design goal is to minimize the communication and dependencies between subsystems.
  - One way to achieve this goal is to introduce a **facade** object that provides a single, simplified interface to the more general facilities of a subsystem.

Facade Pattern … Contd.



Facade Pattern … Contd.

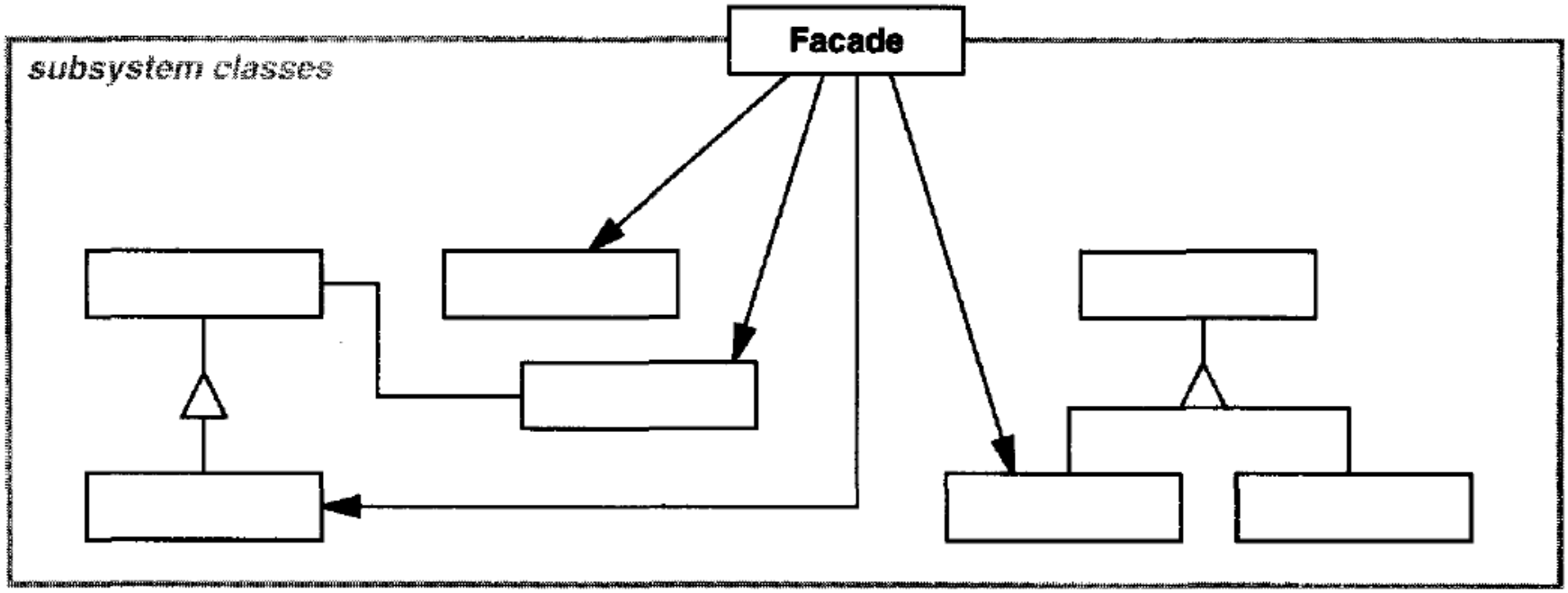
- **Applicability**

Use the Façade pattern when

- Want to provide a simple interface to a complex subsystem.
- There are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
- Want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades

Facade Pattern … Contd.

- **Structure**



Facade Pattern … Contd.

Indian Institute of Information Technology Kottayam

- **Participants**
  - **Facade**
    - knows which subsystem classes are responsible for a request.
    - delegates client requests to appropriate subsystem objects.
  - **subsystem classes**
    - implement subsystem functionality.
    - handle work assigned by the Facade object.
    - have no knowledge of the facade; that is, they keep no references to it.
- **Collaborations**
  - Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s).
  - Clients that use the facade don't have to access its subsystem objects directly

Facade Pattern … Contd.

- **Consequences**

The Facade pattern offers the following benefits:

- It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
- It promotes weak coupling between the subsystem and its clients.
- It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Facade Pattern … Contd.

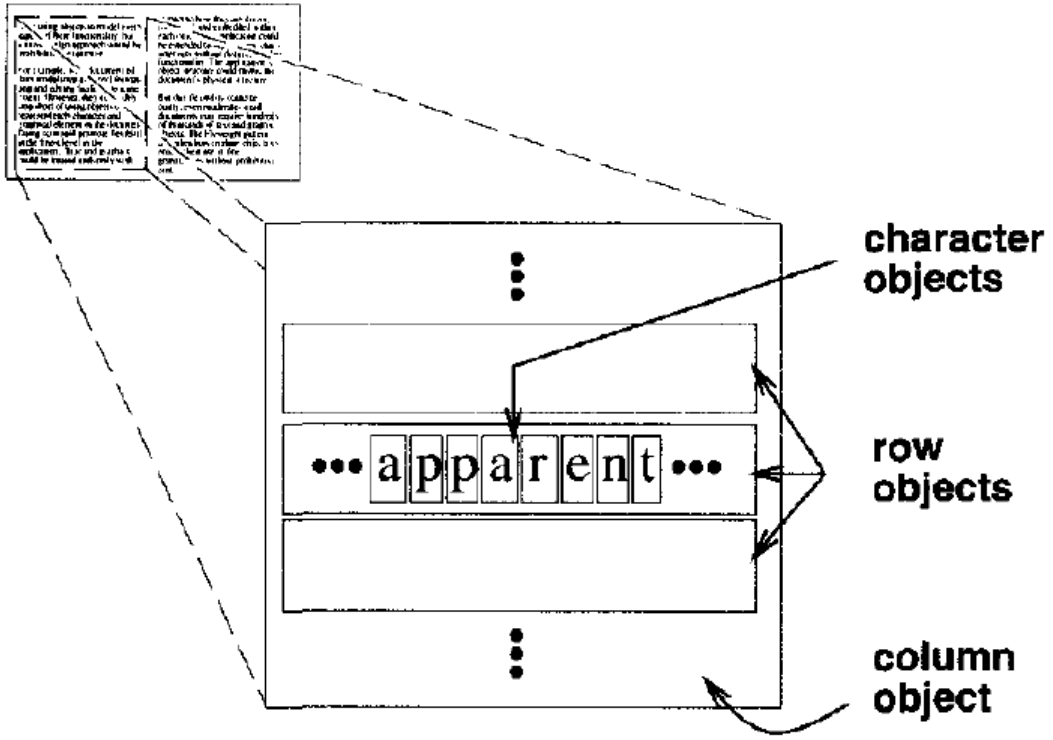
Indian Institute of Information Technology Kottayam

- **Implementation**
- Consider the following issues when implementing a facade:
  - *Reducing client-subsystem coupling*
  - *Public versus private subsystem classes*
- **Applications**
  - In Java, the interface JDBC can be called a facade because, we as users or clients create connection using the "java.sql.Connection" interface, the implementation of which we are not concerned about. The implementation is left to the vendor of driver.
  - Startup of a computer. When a computer starts up, it involves the work of cpu, memory, hard drive, etc.

Flyweight Pattern

Indian Institute of Information Technology Kottayam

- **Intent**
  - Use sharing to support large numbers of fine-grained objects efficiently.
- **Motivation**



Flyweight Pattern … Contd.

- The drawback of such a design is its cost.
- Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time overhead.
- The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost.
- Structuring a system into subsystems helps reduce complexity.

Flyweight Pattern … Contd.

Before we apply flyweight design pattern, we need to consider following factors:

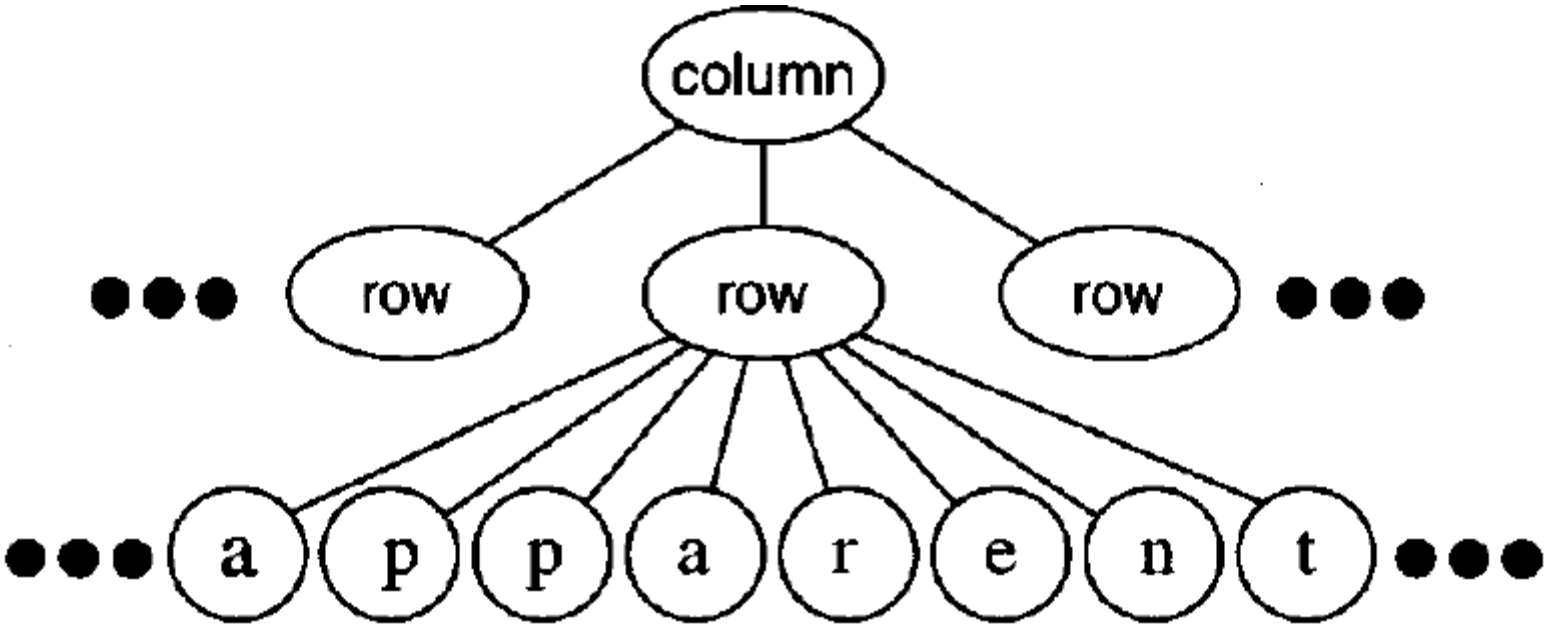
- The number of Objects to be created by application should be huge.
- The object creation is heavy on memory and it can be time consuming too.
- The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program.

Flyweight Pattern … Contd.

- To apply flyweight pattern, we need to divide Object property into intrinsic and extrinsic properties.
- Intrinsic properties make the Object unique
- Extrinsic properties are set by client code and used to perform different operations.
- For example, an Object Circle can have extrinsic properties such as color and width.

Flyweight Pattern … Contd.

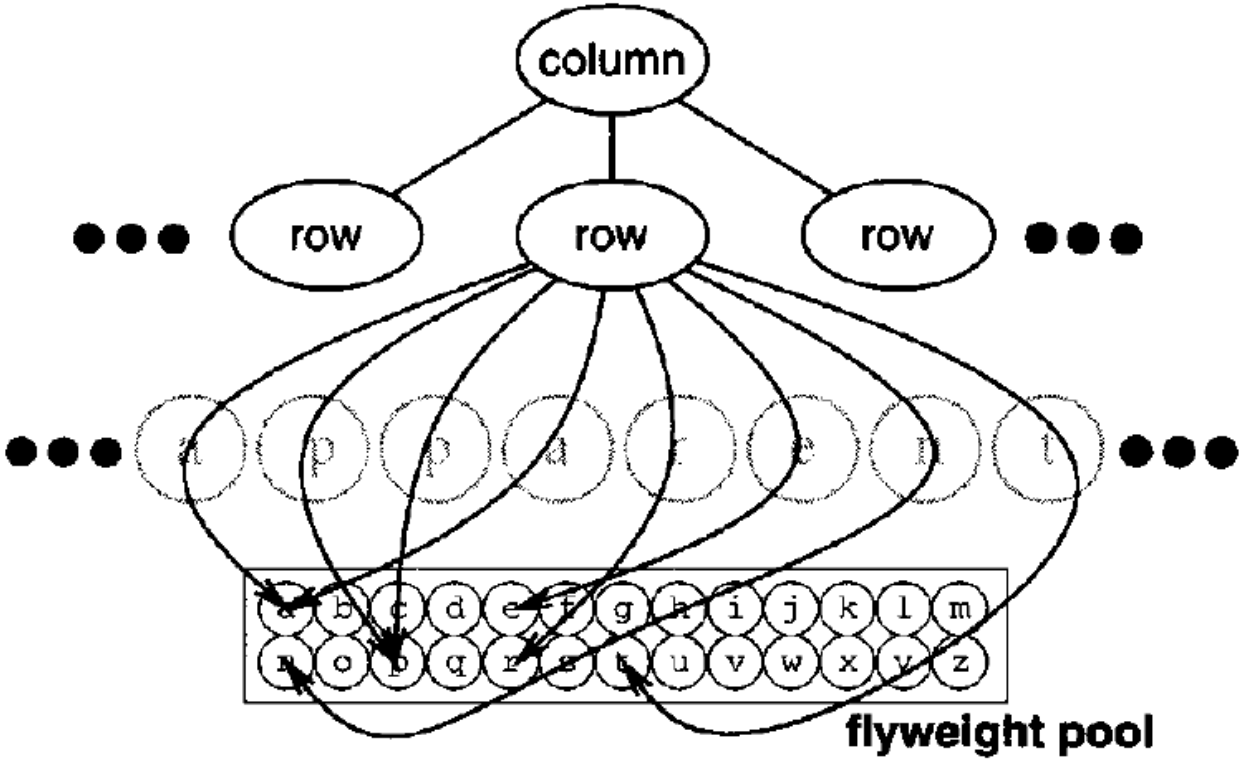
- Logically there is an object for every occurrence of a given character in the document



Flyweight Pattern … Contd.

Indian Institute of Information Technology Kottayam

- there is one shared flyweight object per character, and it appears in different contexts in the document structure



Flyweight Pattern … Contd.

- **Applicability**

The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when *all* of the following are true:

- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on object identity. Since flyweight objects may be shared, identity test will be return true for conceptually distinct objects

Flyweight Pattern … Contd.

- **Participants**
  - **Flyweight** (Glyph)
    - declares an interface through which flyweights can receive and act on extrinsic state.
  - **ConcreteFlyweight (Character)**
    - implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
  - **UnsharedConcreteFlyweight (Row ,Col umn)**
    - not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing; it doesn't enforce it. It's common for UnsharedConcrete- Flyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).

Flyweight Pattern … Contd.

- **Participants**
  - **FlyweightFactory**
    - creates and manages flyweight objects.
    - ensures that flyweights are shared properly. When a client request s a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
  - **Client**
    - maintains a reference to flyweight(s).
    - computes or stores the extrinsic state of flyweight(s).

Flyweight Pattern … Contd.

- **Collaborations**
  - State that a flyweight needs to function must be characterized as either intrinsic or extrinsic. Intrinsic state is stored in the ConcreteFlyweight object; extrinsic state is stored or computed by Client objects. Clients pass this state to the flyweight when they invoke its operations.
  - Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Flyweight Pattern … Contd.

- **Consequences**

  Storage savings are a function of several factors:

  - The reduction in the total number of instances that comes from sharing
  - The amount of intrinsic state per object
  - Whether extrinsic state is computed or stored

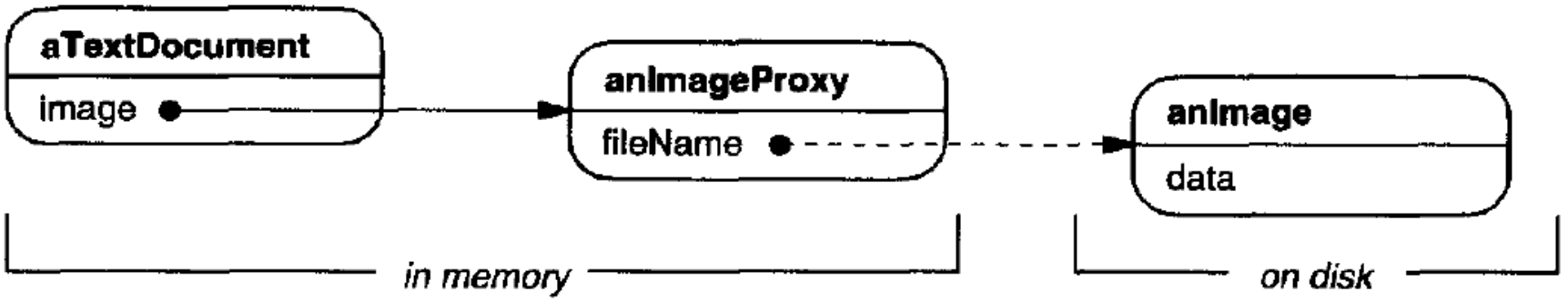
Flyweight Pattern … Contd.

- **Implementation**
- Consider the following issues when implementing the Flyweight pattern:
  - *Removing extrinsic state*
  - *Managing shared objects.*

Proxy Pattern

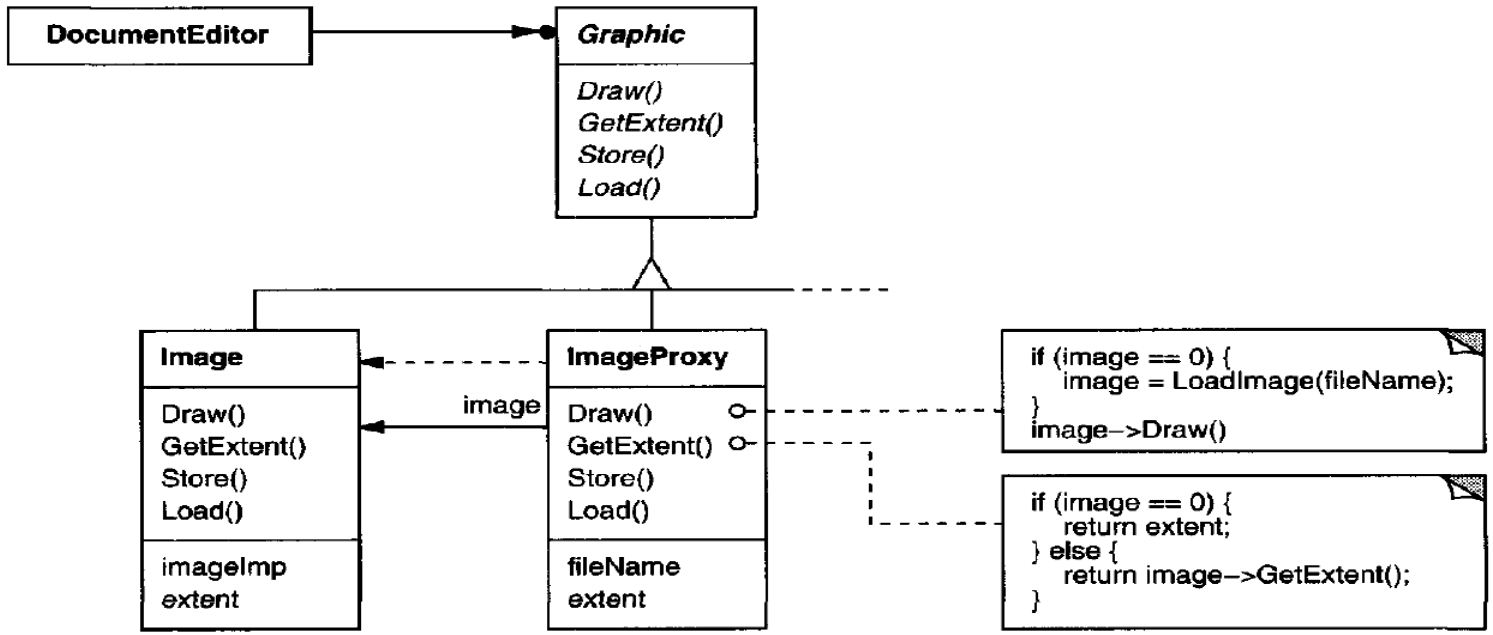
Indian Institute of Information Technology Kottayam

- **Also Known As** Surrogate
- **Intent**
  - Provide a surrogate or placeholder for another object to control access to it.
- **Motivation**
  - 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.



Proxy Pattern … Contd.

- The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation.
- The proxy also stores its extent, that is, its width and height

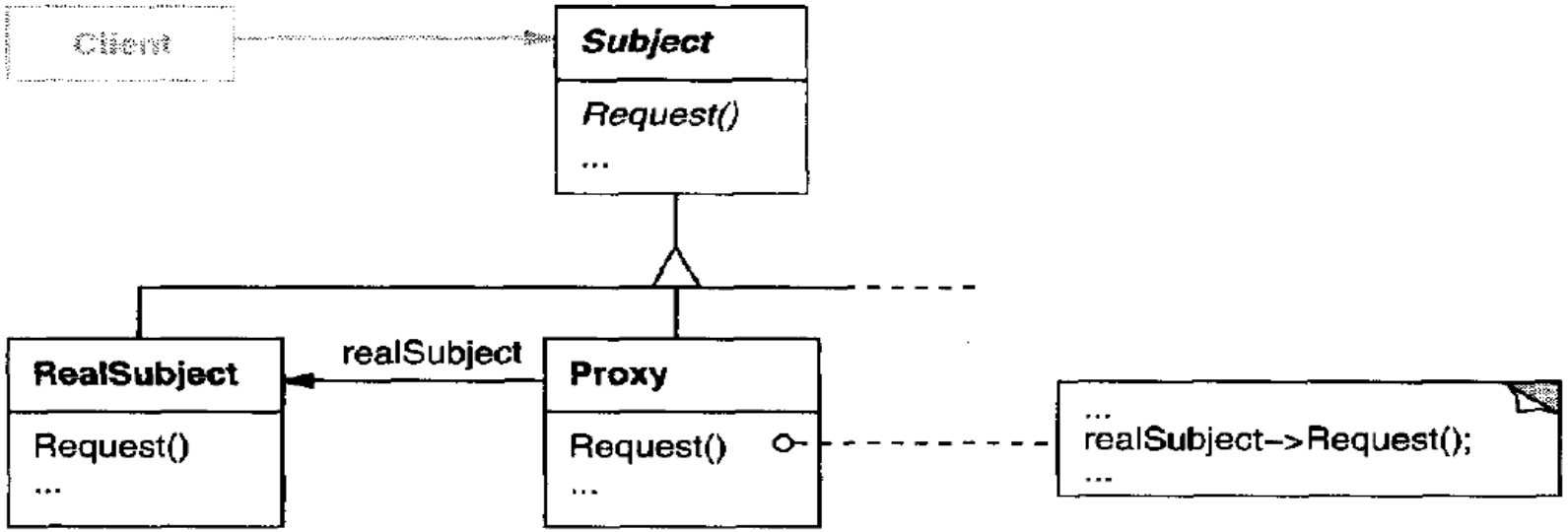


Proxy Pattern … Contd.

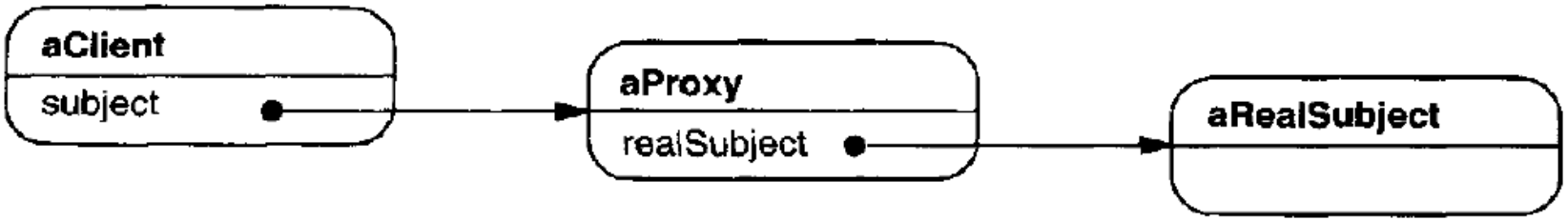
Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:

1. A remote proxy provides a local representative for an object in a different address space.
2. A virtual proxy creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.
3. A protection proxy controls access to the original object. Protection proxies are useful when objects should have different access rights.
4. A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed

Proxy Pattern … Contd.



Object Diagram of Proxy Structure



Proxy Pattern … Contd.

Proxy (ImageProxy)

- Maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the Real Subject and Subject interfaces are the same.
- Provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
- Controls access to the real subject and may be responsible for creating and deleting it.
- other responsibilities depend on the kind of proxy:
  - remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
  - virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
  - protection proxies check that the caller has the access permissions required to perform a request.

Proxy Pattern … Contd.

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.

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending o n the kind of proxy.

Proxy Pattern … Contd.

Consequences

The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:

1. A remote proxy can hide the fact that an object resides in a different address space.
2. A virtual proxy can perform optimizations such as creating an object on demand.
3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

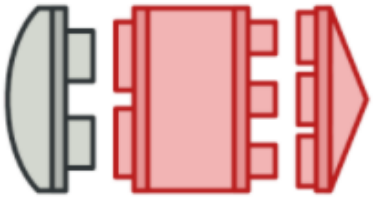
Proxy Pattern … Contd.

Implementation

1. Overloading the member access operator
2. Proxy doesn't always have to know the type of real subject. The Proxy pattern can exploit the language features:

Discussion of Structural Patterns

- Structural patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.



Adapter

Allows objects with incompatible interfaces to collaborate.

Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Ref.: https://refactoring.guru/design-patterns/structural-patterns

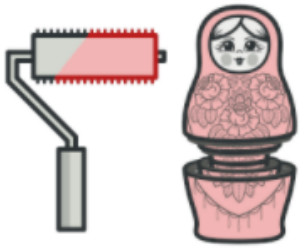
Discussion of Structural Patterns

Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.

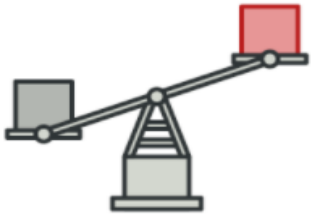


Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

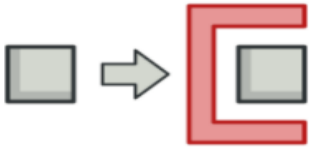
Ref.: https://refactoring.guru/design-patterns/structural-patterns

Discussion of Structural Patterns



Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Ref.: https://refactoring.guru/design-patterns/structural-patterns

Discussion of Structural Patterns

Similarities between Adapter and Bridge

The Adapter and Bridge patterns have some common attributes.

- 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.

Discussion of Structural Patterns

Adapter versus Bridge

- The key difference between these patterns lies in their intents.
- Adapter focuses on resolving incompatibilities between two existing interfaces. It doesn't focus on how those interfaces are implemented, nor does it consider how they might evolve independently.
- It's a way of making two independently designed classes work together without reimplementing one or the other.
- Bridge, on the other hand, bridges an abstraction and its (potentially numerous) implementations.
- It provides a stable interface to clients even as it lets you vary the classes that implement it.
- It also accommodates new implementations as the system evolves.

Discussion of Structural Patterns

Adapter versus Bridge … contd.

- As a result of these differences, Adapter and Bridge are often used at different points in the software lifecycle.
- An adapter often becomes necessary when you discover that two in compatible classes should work together, generally to avoid replicating code.
- In contrast, the user of a bridge understands up-front that an abstraction must have several implementations, and both may evolve independently.
- The Adapter pattern makes things work *after* they're designed; Bridge makes them work *before* they are. That doesn't mean Adapter is somehow inferior to Bridge; each pattern merely addresses a different problem.

Discussion of Structural Patterns

Adapter versus facade

- You might think of a facade as an adapter to a set of other objects. But that interpretation overlooks the fact that a facade defines a *new* interface, whereas an adapter reuses an old interface.
- Remember that an adapter makes two *existing* interfaces work together as opposed to defining an entirely new one.

Discussion of Structural Patterns

Composite versus Decorator versus Proxy

- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- This commonality might tempt you to think of a decorator object as a degenerate composite, but that misses the point of the Decorator pattern.
- The similarity ends at recursive composition, again because of differing intents.

Discussion of Structural Patterns

Composite versus Decorator versus Proxy

- Decorator is designed to let you add responsibilities to objects without sub-classing.
- It avoids the explosion of subclasses that can arise from trying to cover every combination of responsibilities statically.
- Composite has a different intent. It 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.
- These intents are distinct but complementary.

Discussion of Structural Patterns

Composite versus Decorator versus Proxy

- Consequently, the Composite and Decorator patterns are often used in concert. Both lead to the kind of design in which you can build applications just by plugging objects together without defining any new classes.
- There will be an abstract class with some subclasses that are composites, some that are decorators, and some that implement the fundamental building blocks of the system.
- In this case, both composites and decorators will have a common interface.
- From the point of view of the Decorator pattern, a composite is a Concrete Component. From the point of view of the Composite pattern, a decorator is a Leaf.

Discussion of Structural Patterns

Composite versus Decorator versus Proxy

- Another pattern with a structure similar to Decorator's is Proxy. Both patterns describe how to provide a level of indirection to an object, and the implementations of both the proxy and decorator object keep a reference to another object to which they forward requests.
- however, they are intended for different purposes.
- Like Decorator, the Proxy pattern composes an object and provides an identical interface to clients.
- Unlike Decorator, the Proxy pattern is not concerned with attaching or detaching properties dynamically, and it's not designed for recursive composition.

Discussion of Structural Patterns

Composite versus Decorator versus Proxy

- Its intent is to provide a stand-in for a subject when it's inconvenient or undesirable to access the subject directly because, for example, it lives on a remote machine, has restricted access, or is persistent.
- In the Proxy pattern, the subject defines the key functionality, and the proxy provides (or refuses) access to it.
- In Decorator, the component provides only part of the functionality, and one or more decorators furnish the rest.
- Decorator addresses the situation where an object's total functionality can't be determined at compile time, at least not conveniently.

Discussion of Structural Patterns

Composite versus Decorator versus Proxy

- That open-endedness make s recursive composition an essential part of Decorator.
- That isn't the case in Proxy, because Proxy focuses on one relationship—between the proxy and its subject—and that relationship can be expressed statically.
- These differences are significant because they capture solutions to specific recurring problems in object-oriented design. But that doesn't mean these patterns can't be combined.
- You might envision a proxy-decorator that adds functionality to a proxy, or a decorator-proxy that embellishes a remote object is called hybrids patterns, are useful.

Conclusion

Indian Institute of Information Technology Kottayam

Structural Patterns

- Adaptor,
- Bridge,
- Composite,
- Decorator,
- Facade,
- Flyweight,
- Proxy

Implementation in various languages like Python, Java

Reference:

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides Design Patterns: Elements of Reusable Object oriented Software Addison-Wesley

Thank you