VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT - IV

Dr. T. K. Rao

VVIT

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

Contents:

STRUCTURAL PATTERNS:

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- PROXY

Structural patterns

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- Structural class patterns concern Class and Object composition
- These patterns
  - Use inheritance to compose interfaces / implementations
  - Define ways to compose objects to obtain new functionality
- Rather than composing interfaces/ implementations, structural *object* patterns describe ways to compose objects to realize new functionality

- Added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class composition
- Another e.g. is, 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 useful for making independently developed class libraries work together

Adapter

VVIT
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

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

Also Known As

- Wrapper

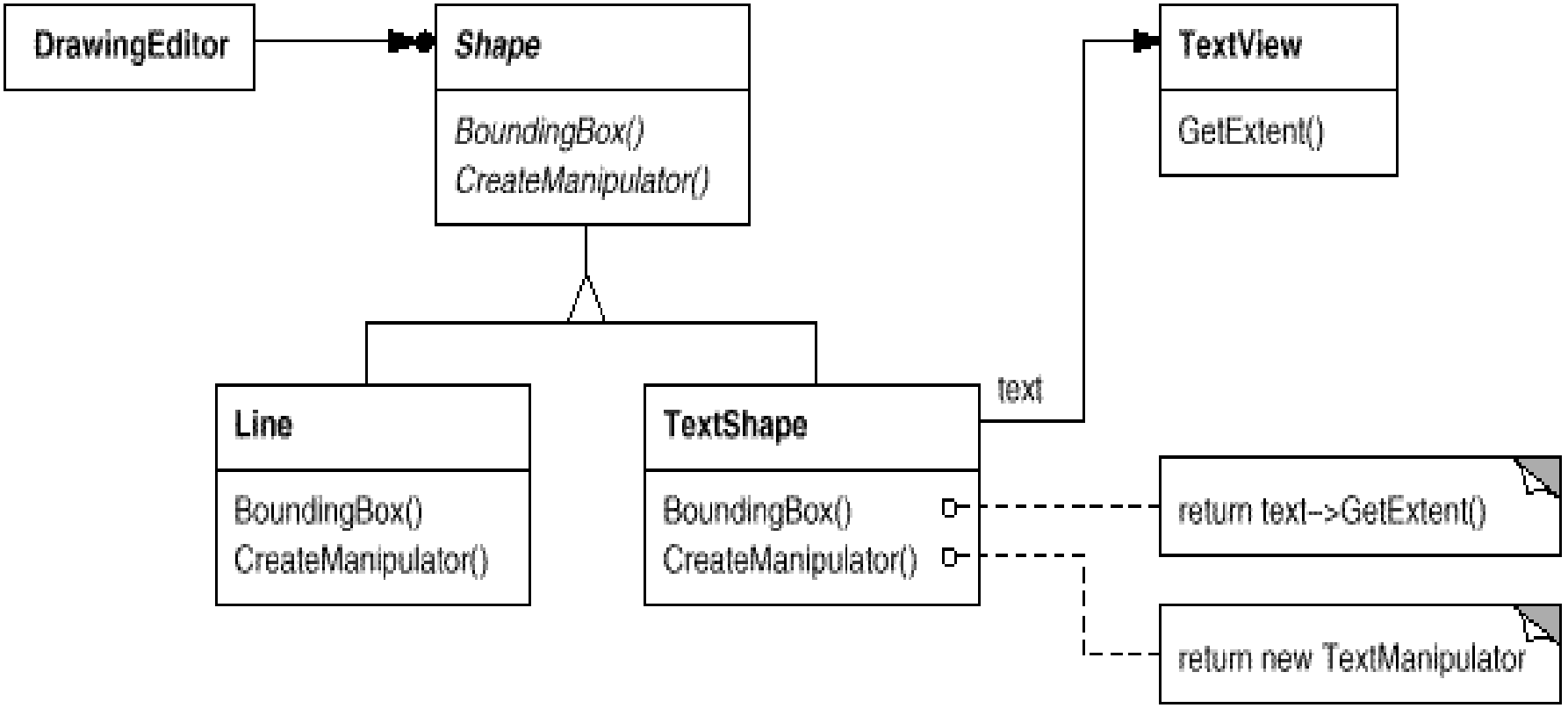
Motivation

- Sometimes a toolkit class (designed for reuse) isn't reusable since its interface doesn't match the domain-specific interface of an app
- E.g., a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams
- The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself
- Interface for graphical objects is defined by an abstract class ‘Shape’
- The editor defines a subclass of Shape for each kind of graphical object:
  - a LineShape class for lines, a PolygonShape class for polygons, etc.

- Classes for shapes like LineShape / PolygonShape are easy to implement, because their drawing and editing capabilities are inherently limited
- But a TextShape subclass that can display and edit text is considerably more difficult to implement, since even basic text editing involves complicated screen update and buffer management
- Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated TextView class for displaying and editing text
- Ideally we'd like to reuse TextView to implement TextShape, but the toolkit wasn't designed with Shape classes in mind
- So we can't use TextView and Shape objects interchangeably.

- How can unrelated classes like TextView work in an app that expects classes with incompatible interface?
- Change the ‘TextView’ class so that it conforms to ‘Shape’ interface, but that isn't an option unless we have the toolkit's source code
- Even if we did, it wouldn't make sense to change TextView
- The toolkit shouldn't have to adopt domain-specific interfaces just to make one application work.

- Instead, we could define TextShape so that it *adapts* the TextView interface to Shape's
- We can do this in one of two ways:
  - by inheriting Shape's interface and TextView's implementation
  - by composing a TextView instance within a TextShape and implementing TextShape in terms of TextView's interface
- These two approaches correspond to the class and object versions of the Adapter pattern
- We call TextShape an **adapter**


DrawingEditor
Shape
BoundingBox()
CreateManipulator()
TextView
GetExtent()
Line
BoundingBox()
CreateManipulator()
TextShape
BoundingBox()
CreateManipulator()
text
return text->GetExtent()
return new TextManipulator

- This diagram illustrates the object adapter case
- It shows how BoundingBox requests, declared in class Shape, are converted to GetExtent requests defined in TextView
- Since TextShape adapts TextView to the Shape interface, the drawing editor can reuse the otherwise incompatible TextView class

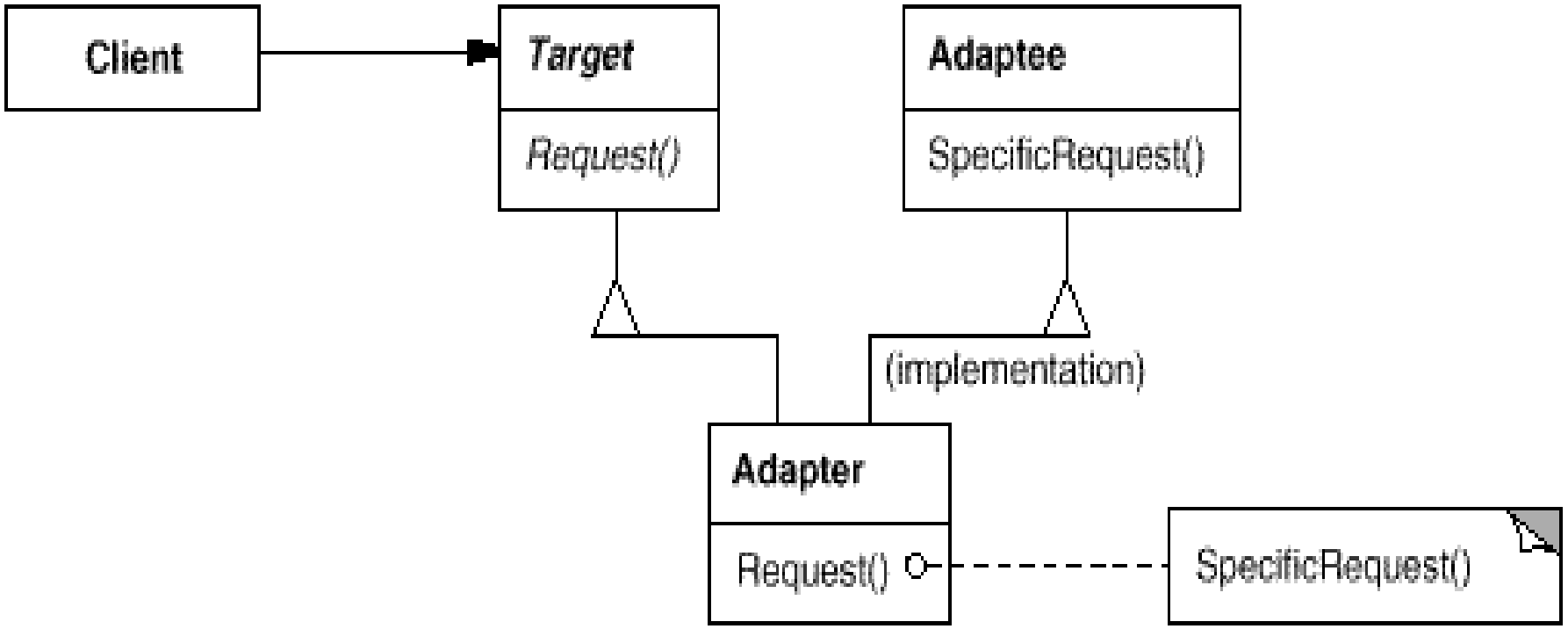
- The adapter is responsible for functionality the adapted class doesn't provide
- Fig. shows how an adapter can fulfill such responsibilities
- User should be able to "drag" every Shape object to a new location interactively, but TextView isn't designed to do that
- TextShape can add missing functionality by implementing Shape's CreateManipulator operation, which returns an instance of the appropriate Manipulator subclass

- Manipulator is an abstract class for objects that know how to animate a Shape in response to user input, like dragging the shape to a new location
- There are subclasses of Manipulator for different shapes;
- E.g. TextManipulator, is the corresponding subclass for TextShape
- By returning a TextManipulator instance, TextShape adds the functionality that TextView lacks

Applicability

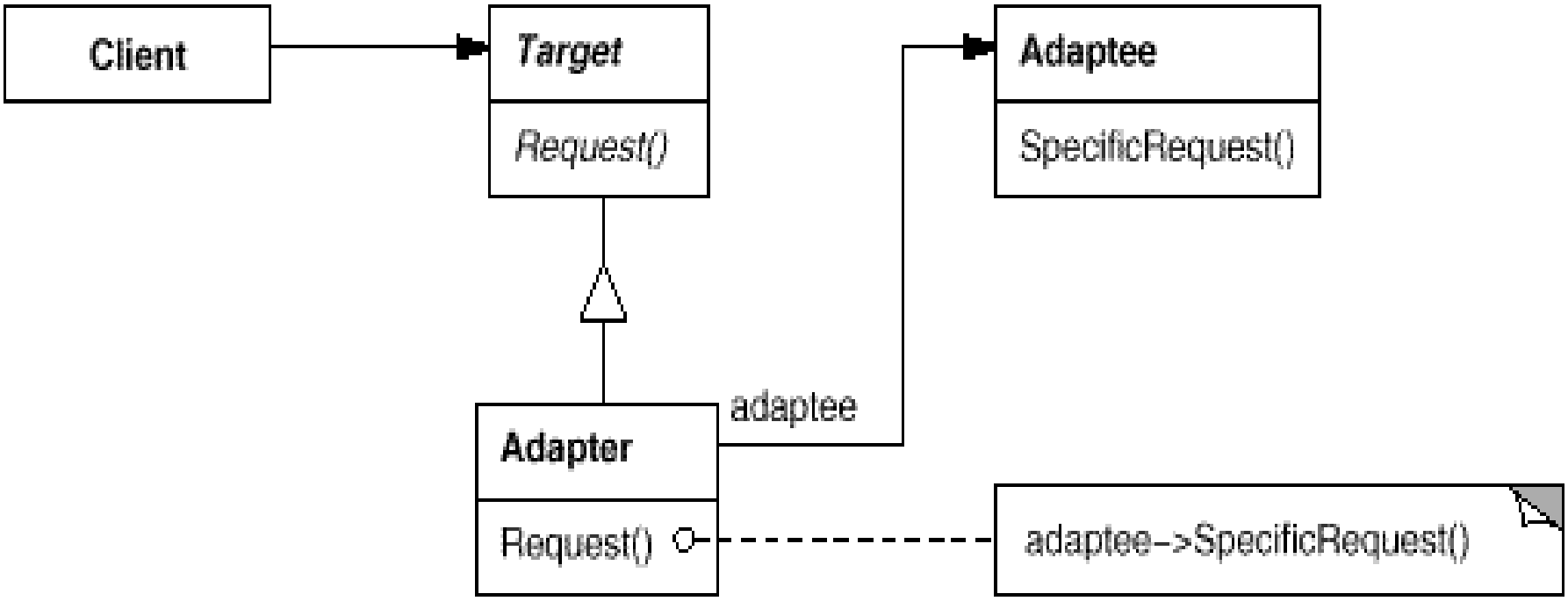
- Use the Adapter pattern when
  - you want to use an existing class, and its interface does not match the one you need.
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
  - *(object adapter 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.

Structure



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

An object adapter relies on object composition:



Participants

- **Target** (Shape)
  - defines the domain-specific interface that Client uses.
- **Client** (DrawingEditor)
  - collaborates with objects conforming to the Target interface.
- **Adaptee** (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 Adaptee operations that carry out the request.

Consequences

- Class and object adapters have different trade-offs
- **A class adapter**
  - adapts Adaptee to Target by committing to a concrete Adapter class, then, a class adapter won't work to adapt a class *and* all its subclasses
  - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee
  - introduces only one object, and no additional pointer indirection is needed to get to the adaptee

- **An object adapter**
  - lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any)
  - Adapter can also add functionality to all Adaptees at once
  - makes it harder to override Adaptee behavior.
  - It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Here are other issues to consider when using the Adapter pattern:

1. *How much adapting does Adapter do?*

- Adapters vary in the amount of work they do to adapt Adaptee to the Target interface.
- There is a spectrum of possible work, from simple interface conversion
- E.g., changing the names of operations—to supporting an entirely different set of operations
- The amount of work Adapter does depends on how similar the Target interface is to Adaptee's

2. *Pluggable adapters*

- A class is more reusable when you minimize the assumptions other classes must make to use it
- By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface
- Put another way, interface adaptation lets us incorporate our class into existing systems that might expect different interfaces to the class
- ObjectWorks\Smalltalk [Par90] uses the term **pluggable adapter** to describe classes with built-in interface adaptation

3. Using two-way adapters to provide transparency

- A potential problem with adapters is that they aren't transparent to all clients
- An adapted object no longer conforms to the Adaptee interface, so it can't be used as is wherever an Adaptee object can
- **Two-way adapters** can provide such transparency
- Specifically, they're useful when two different clients need to view an object differently.

Bridge

Intent

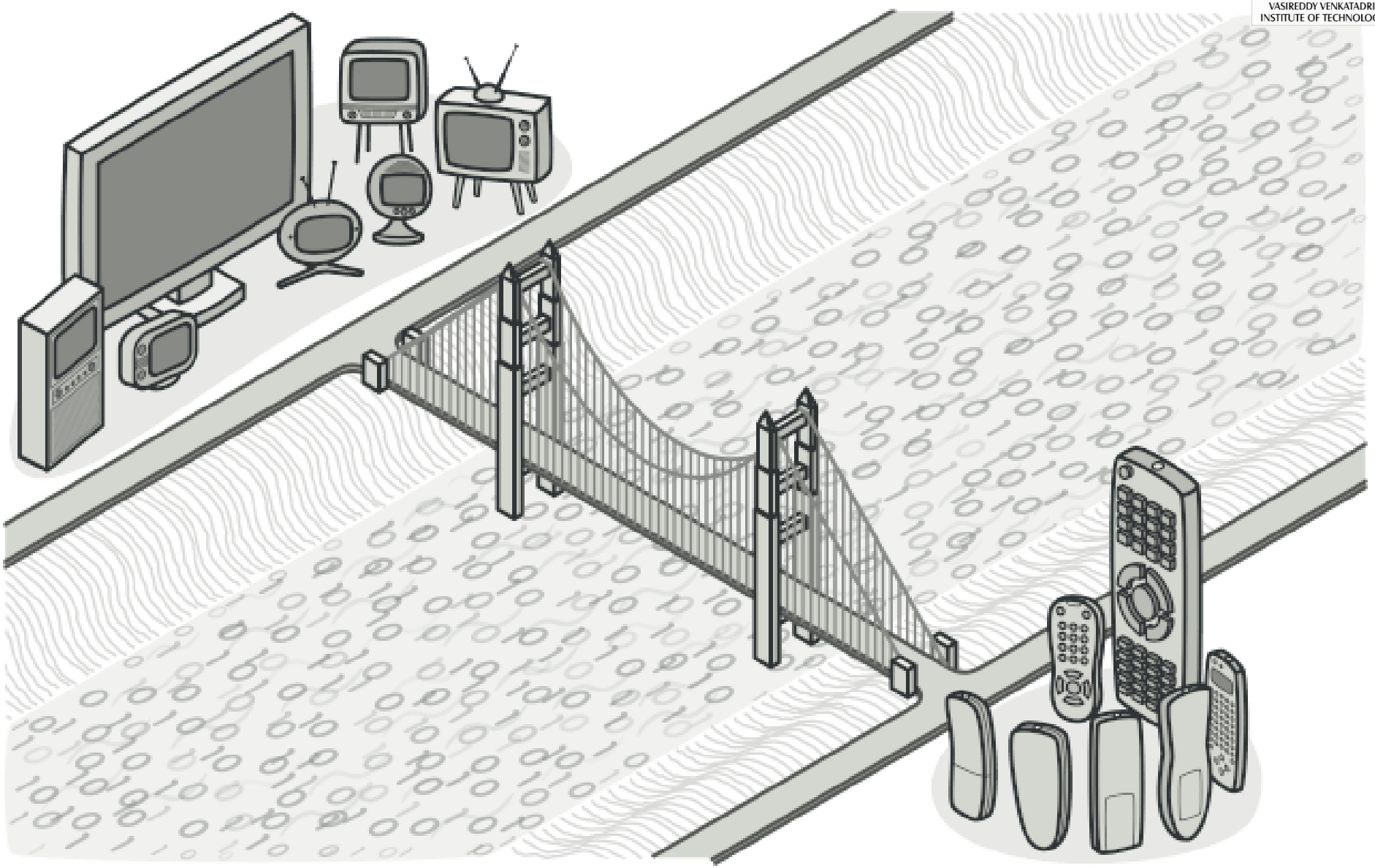
- Decouple an abstraction from its implementation so that both can vary independently
- **L**ets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other

Also Known As

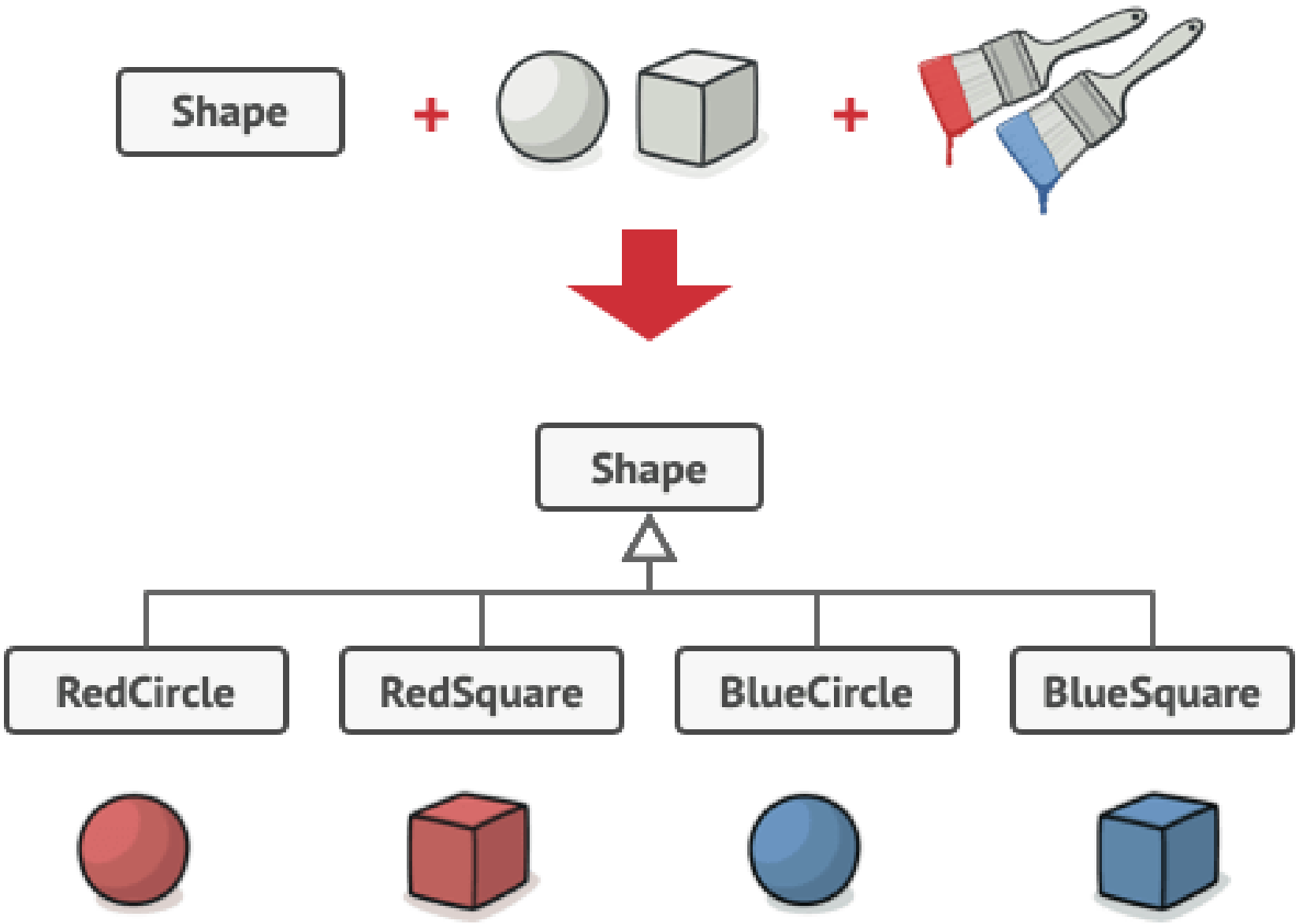
- Handle/Body

Motivation

- When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance



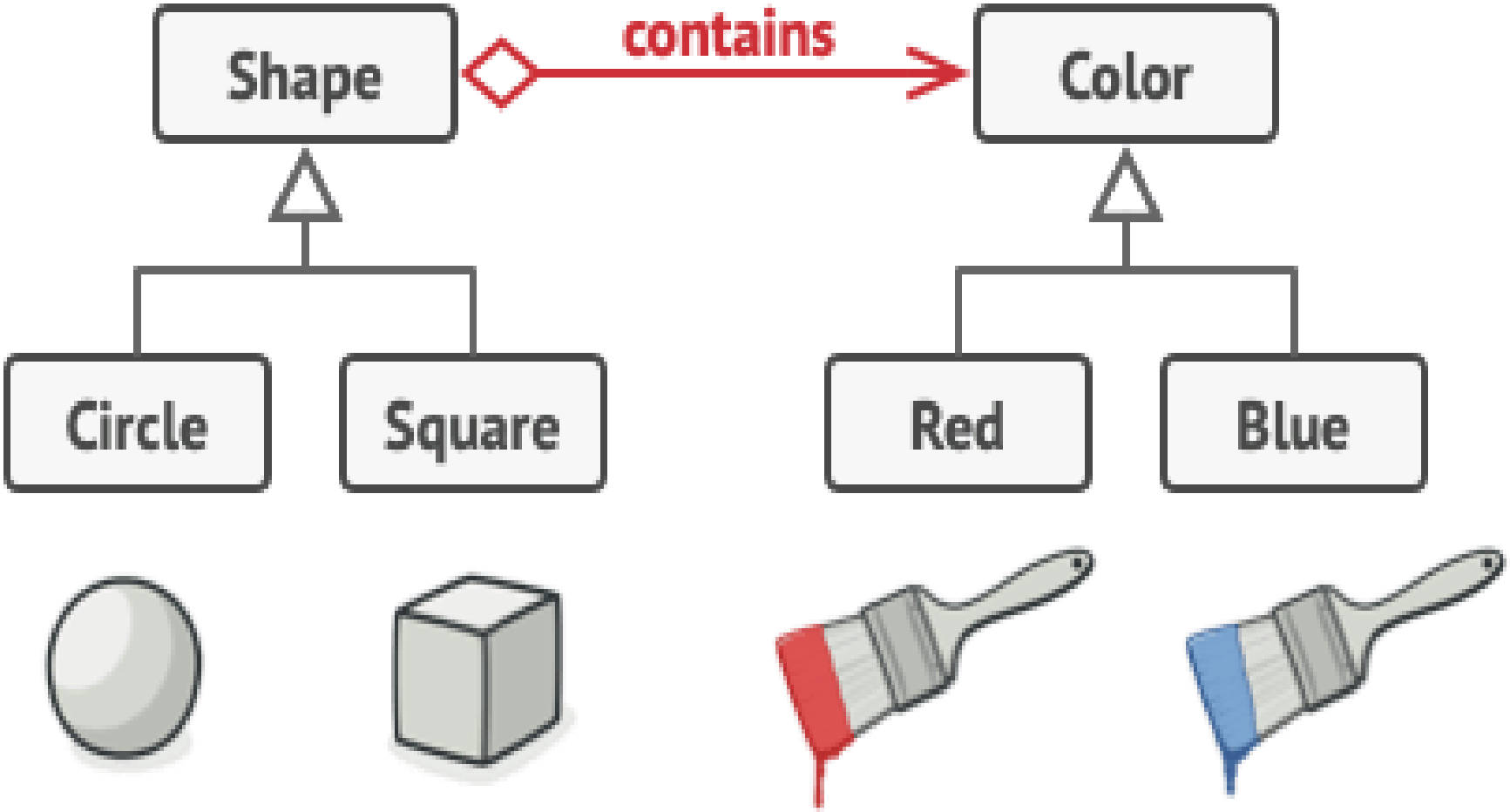
- An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways
- But this approach isn't always flexible enough
- Say you have a geometric Shape class with a pair of subclasses - Circle and Square.
- You want to extend this class hierarchy to incorporate colors, so you plan to create Red and Blue shape subclasses.
- However, since you already have two subclasses, you'll need to create four class combinations such as BlueCircle and RedSquare.


Shape
Shape
RedCircle
RedSquare
BlueCircle
BlueSquare

- Adding new shape types and colors to the hierarchy will grow it exponentially
- E.g., to add a triangle shape you'd need to introduce two subclasses, one for each color
- Adding a new color would require creating three subclasses, one for each shape type
- The further we go, the worse it becomes

Solution

- This problem occurs because we're trying to extend the shape classes in two independent dimensions: by form and by color
- That's a very common issue with class inheritance.
- The Bridge pattern attempts to solve this problem by switching from inheritance to the object composition.
- Extract one of the dimensions into a separate class hierarchy
- So that the original classes will refer an object of the new hierarchy, instead of having all of its state and behaviors within one class

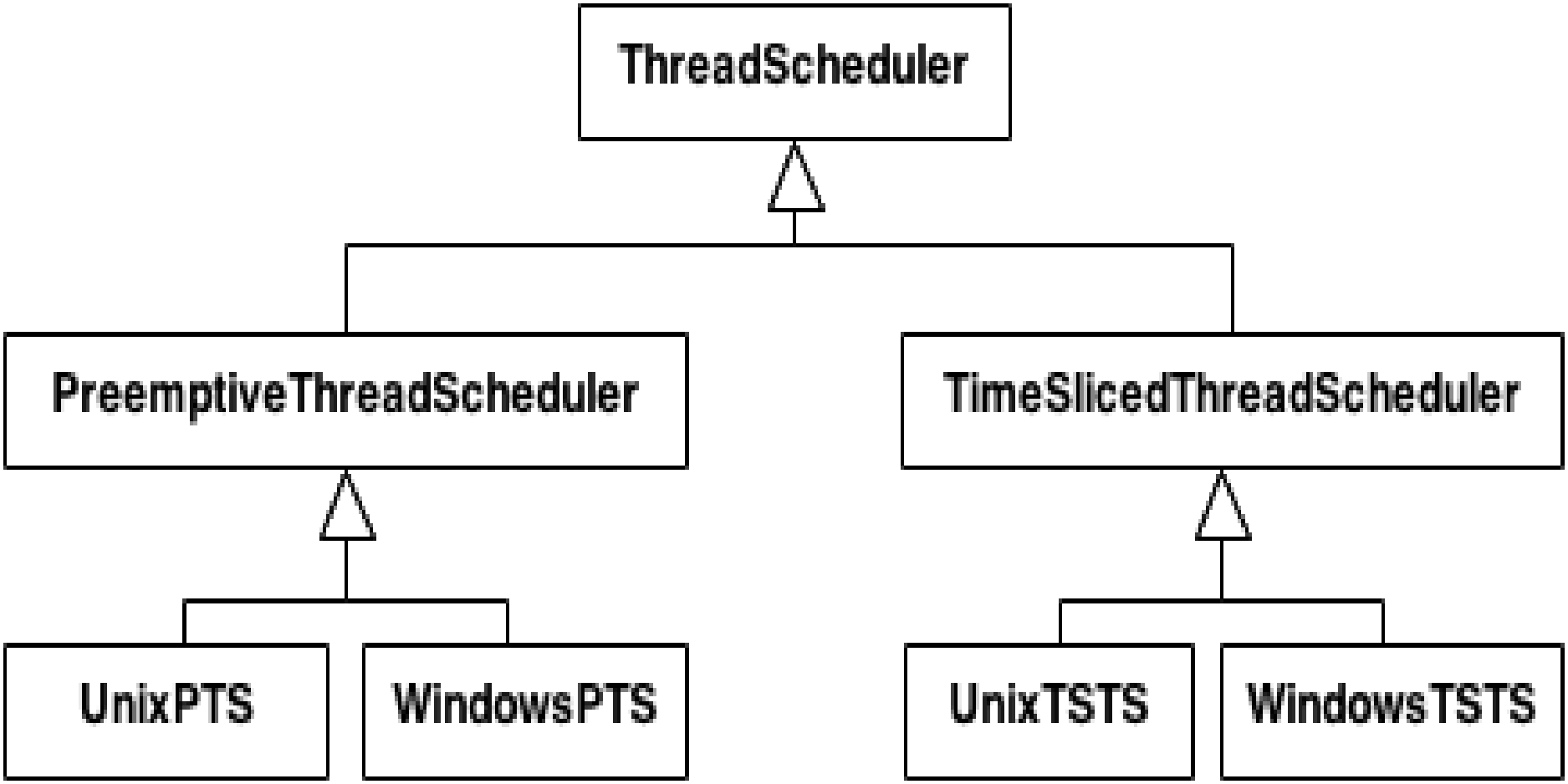

Shape
contains
Color
Circle
Square
Red
Blue

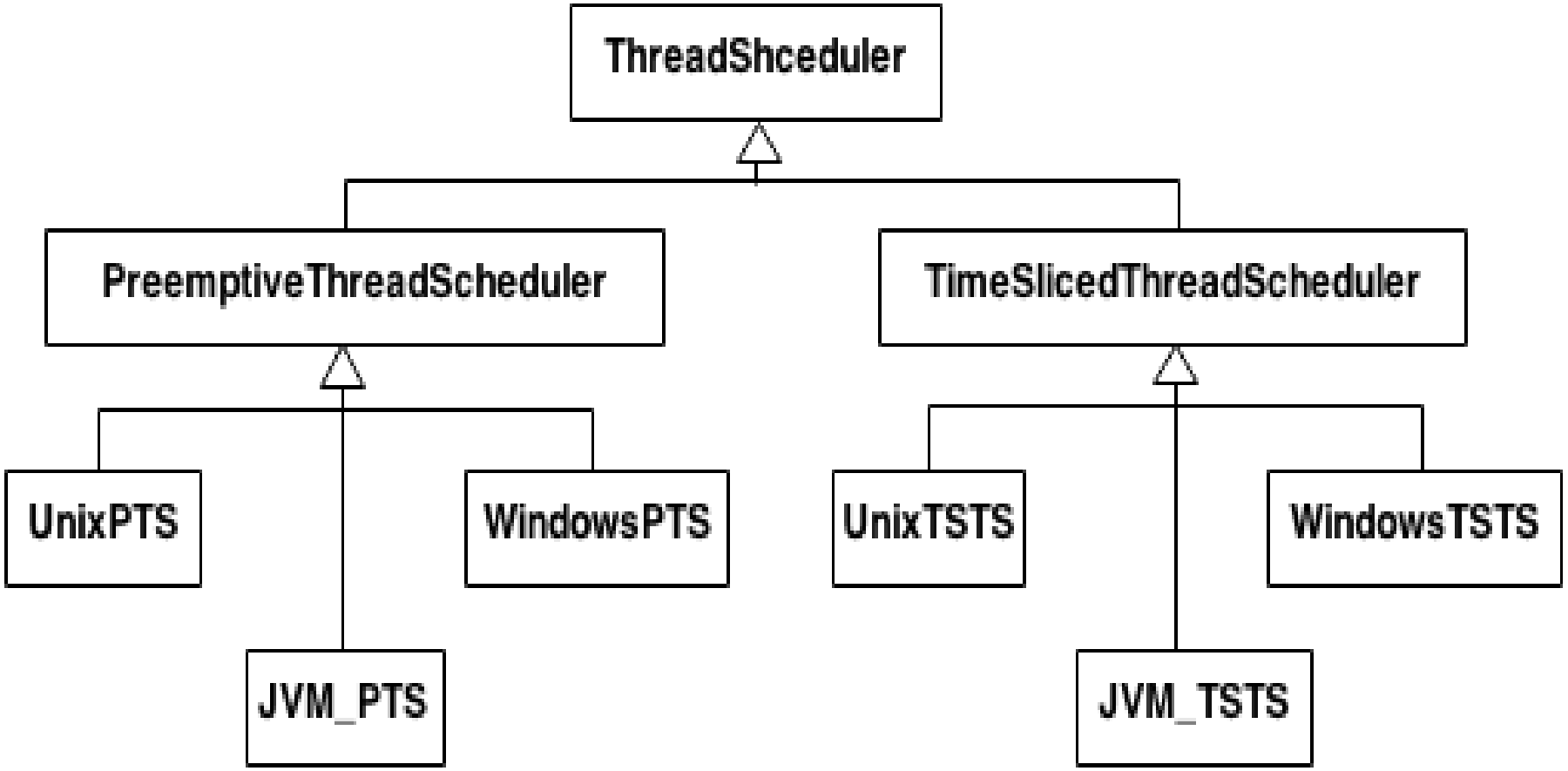
- Following this approach, we can extract the color-related code into its own class with two subclasses: Red and Blue.
- The Shape class then gets a reference field pointing to one of the color objects.
- Now the shape can delegate any color-related work to the linked color object.
- That reference will act as a bridge between the Shape and Color classes.
- From now on, adding new colors won't require changing the shape hierarchy, and vice versa

Another motivation

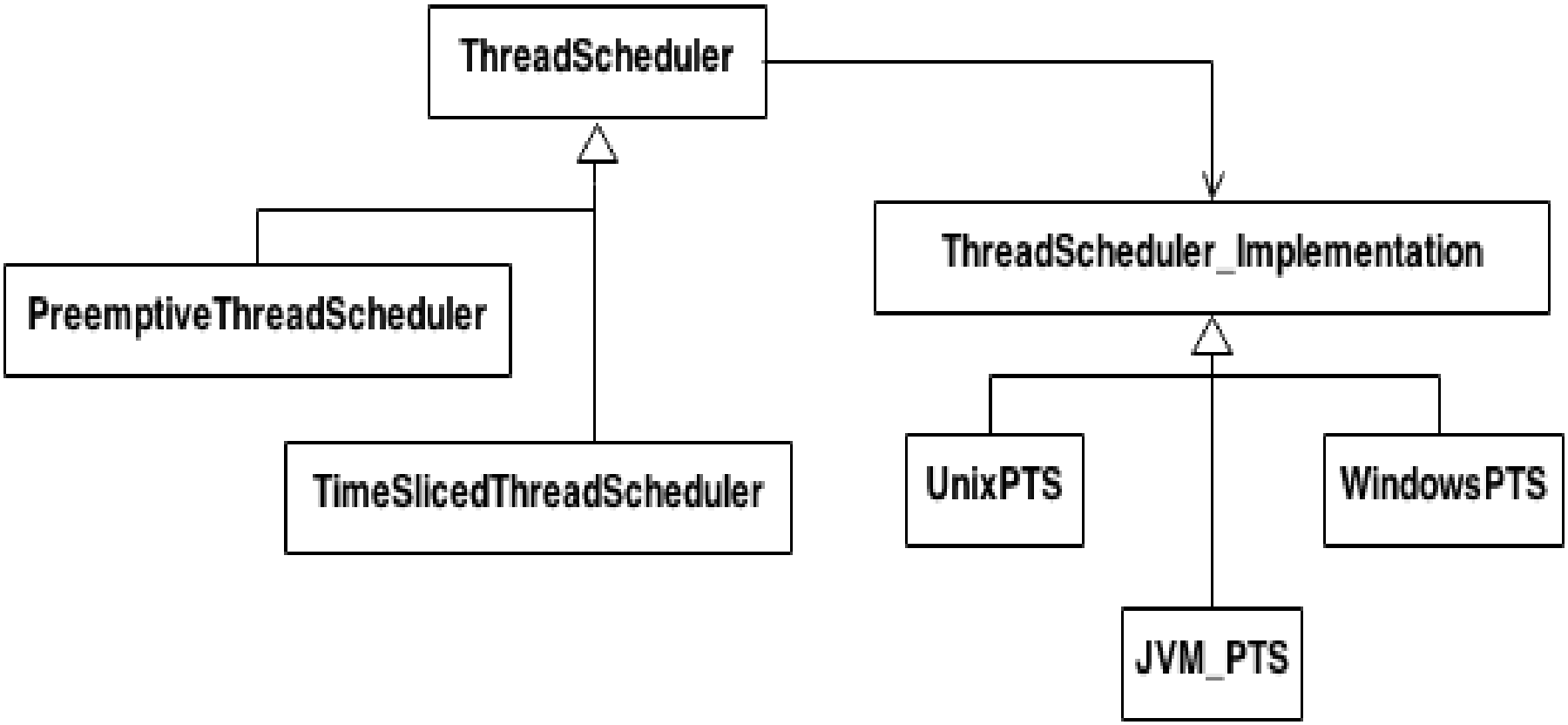
- There are two types of thread schedulers, and two types of operating systems or "platforms“
- Given this approach to specialization, we have to define a class for each permutation of these two dimensions
- If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?

Consider the domain of "thread scheduling".




ThreadShceduler
PreemptiveThreadScheduler
TimeSlicedThreadScheduler
UnixPTS
WindowsPTS
UnixTSTS
WindowsTSTS
JVM_PTS
JVM_TSTS

- What if we had three kinds of thread schedulers, and four kinds of platforms?
- What if we had five kinds of thread schedulers, and ten kinds of platforms?
- The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.
- The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies
  - one for platform-independent abstractions, and
  - the other for platform-dependent implementations

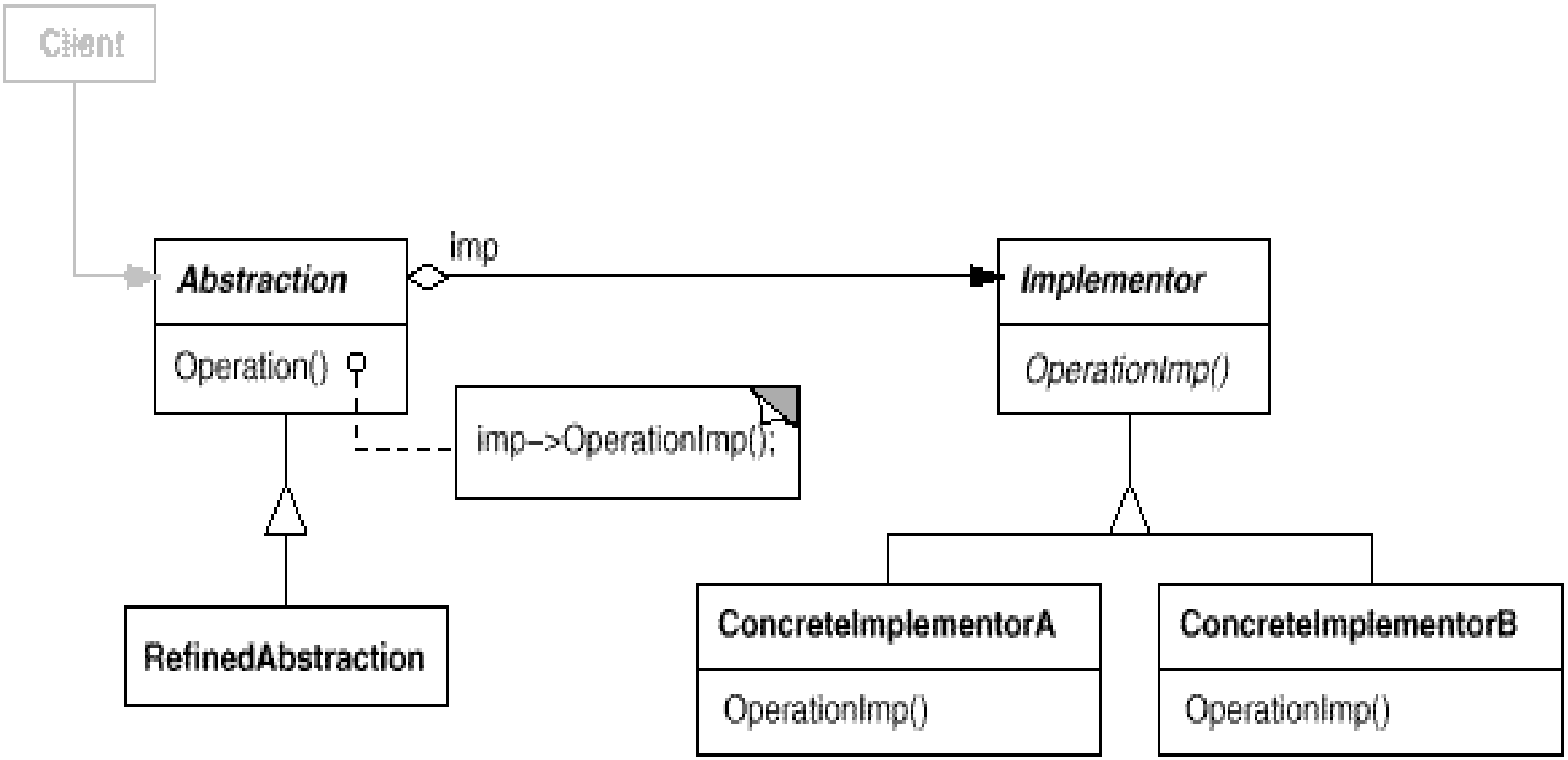

ThreadScheduler
PreemptiveThreadScheduler
TimeSlicedThreadScheduler
ThreadScheduler_Implementation
UnixPTS
WindowsPTS
JVM_PTS

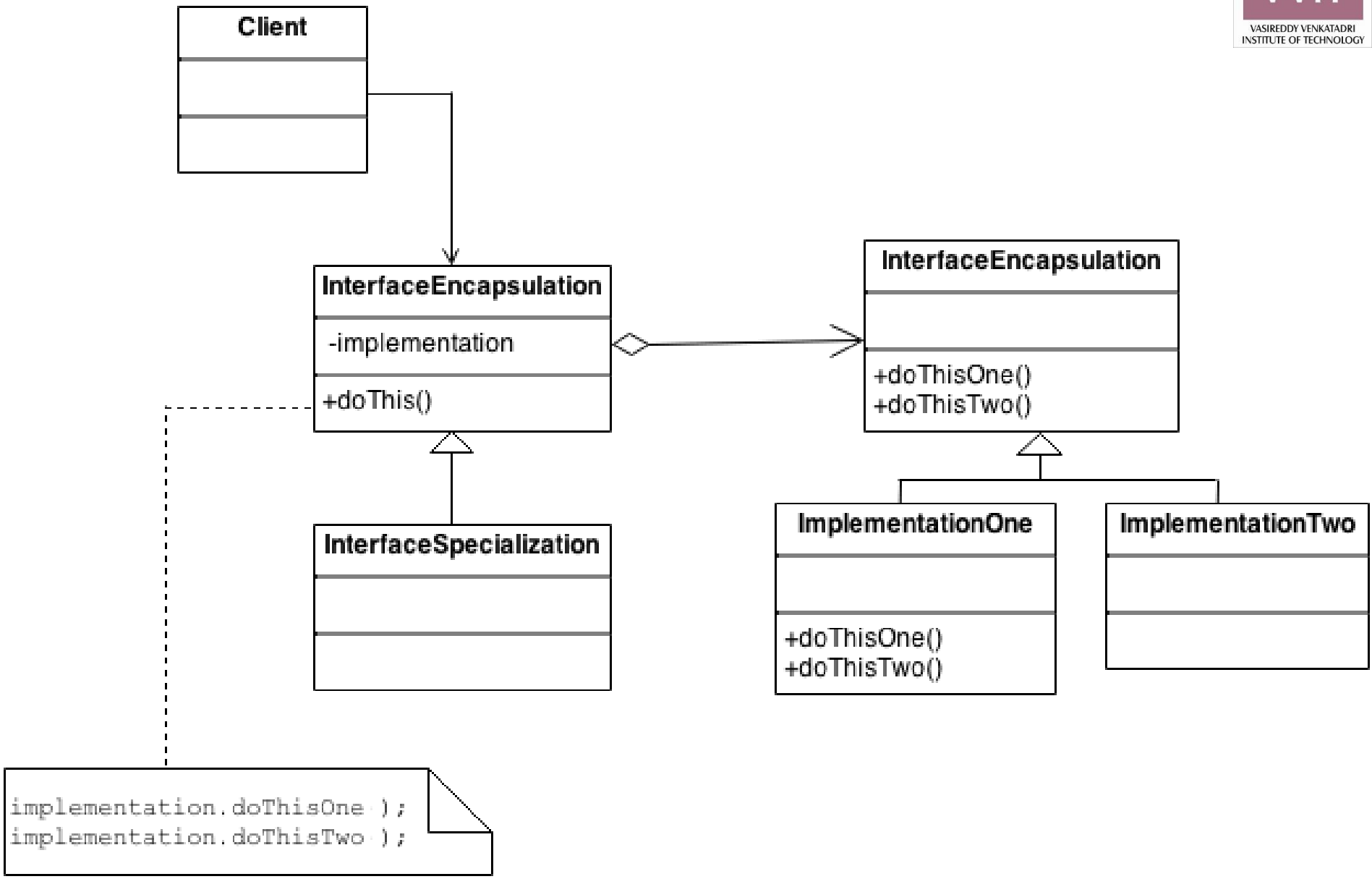
Applicability

Use the Bridge pattern when

- you want to avoid a permanent binding between an abstraction and its implementation
- both the abstractions and their implementations should be extensible by subclassing
- changes in the implementation of an abstraction should have no impact on clients
- you have an expansion of classes, such a class hierarchy indicates the need for splitting an object into two parts
- you want to share an implementation among multiple objects (perhaps using reference counting), and this fact should be hidden from the client

Structure


Client
Abstraction
Operation()
imp
Implementor
OperationImp()
imp->OperationImp();
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()


Client
InterfaceEncapsulation
-implementation
+doThis()
InterfaceEncapsulation
+doThisOne()
+doThisTwo()
InterfaceSpecialization
ImplementationOne
+doThisOne()
+doThisTwo()
ImplementationTwo
implementation.doThisOne);
implementation.doThisTwo);

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 operations based on these primitives.
- **ConcreteImplementor** (XWindowImp, PMWindowImp)
  - implements the Implementor interface and defines its concrete implementation.

Collaborations

- Abstraction forwards client requests to its Implementor object

Consequences

The Bridge pattern has the following consequences:

1. *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 possible for an object to change its implementation at run-time
   - Decoupling Abstraction and Implementor also eliminates compile-time dependencies on the implementation
   - Changing an implementation class doesn't require recompiling the Abstraction class, its clients
   - The high-level part of a system only has to know about Abstraction and Implementor.
2. *Improved extensibility*
   - can extend Abstraction and Implementor hierarchies independently
3. *Hiding implementation details from clients*
   - You can shield clients from implementation details

Implementation

Consider the following implementation issues when applying the Bridge pattern:

- 1. *Only one Implementor.*
  - In situations where there's only one implementation, creating an abstract Implementor class isn't necessary
  - there's a one-to-one relationship between Abstraction and Implementor
  - this separation is useful when a change in the implementation of a class must not affect its existing clients
- *2. Creating the right Implementor object*
  - How, when, and where do you decide which Implementor class to instantiate when there's more than one?
  - If Abstraction knows about all ConcreteImplementor classes, then it can instantiate one of them in its constructor;
  - it can decide between them based on parameters passed to its constructor
  - E.g., If, a collection class supports multiple implementations, the decision can be based on the size of the collection

Related Patterns

- An **Abstract Factory** 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**, is used up-front in a design to let abstractions and implementations vary independently

Composite

Intent

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

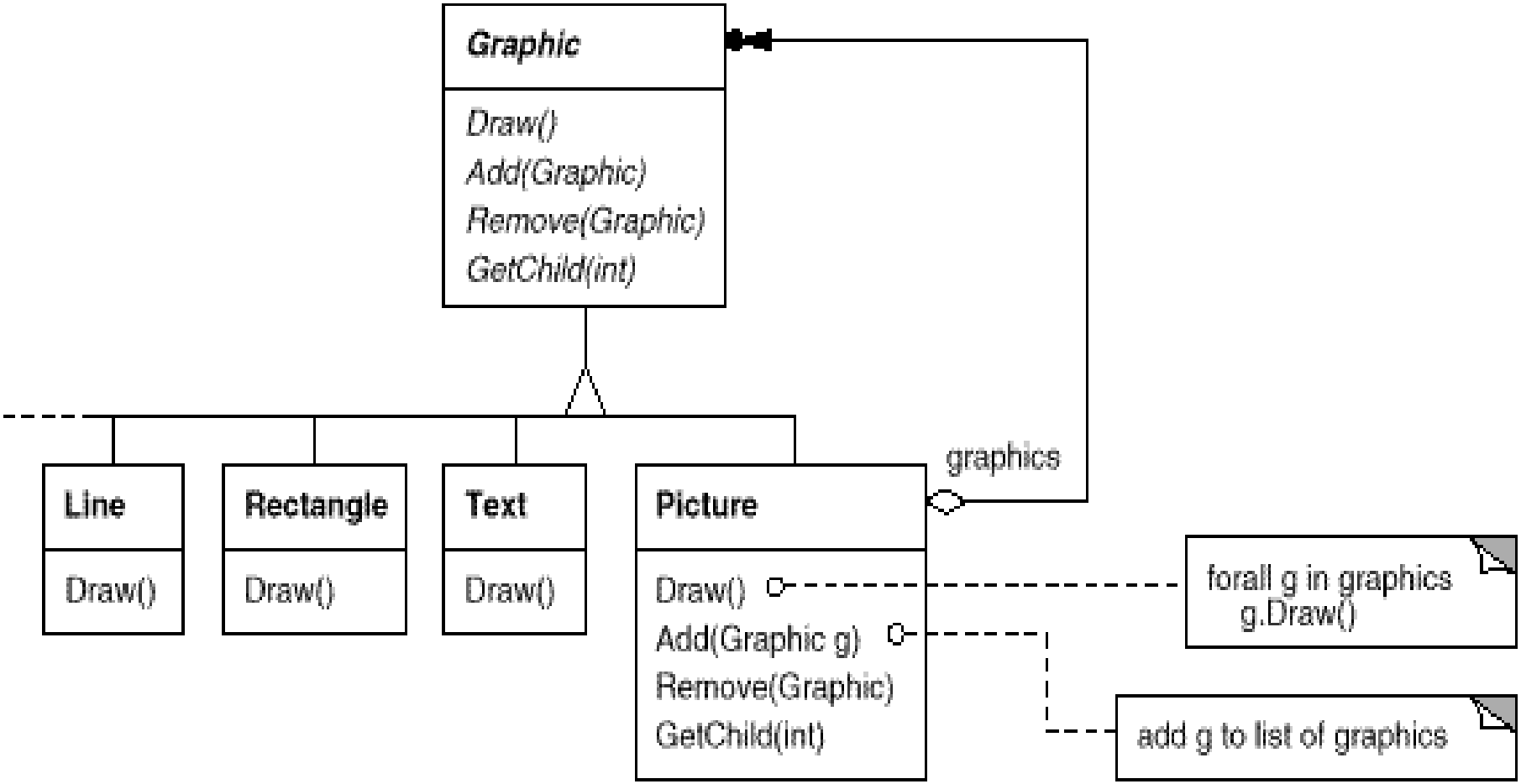
Motivation

- Graphics apps like Drawing editors let users build complex diagrams out of simple components
- A simple implementation could define classes for graphical primitives (e.g. Text/ Lines etc.) that act as containers for these primitives

- Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy
- This is a structural pattern since it creates a tree structure of group of objects

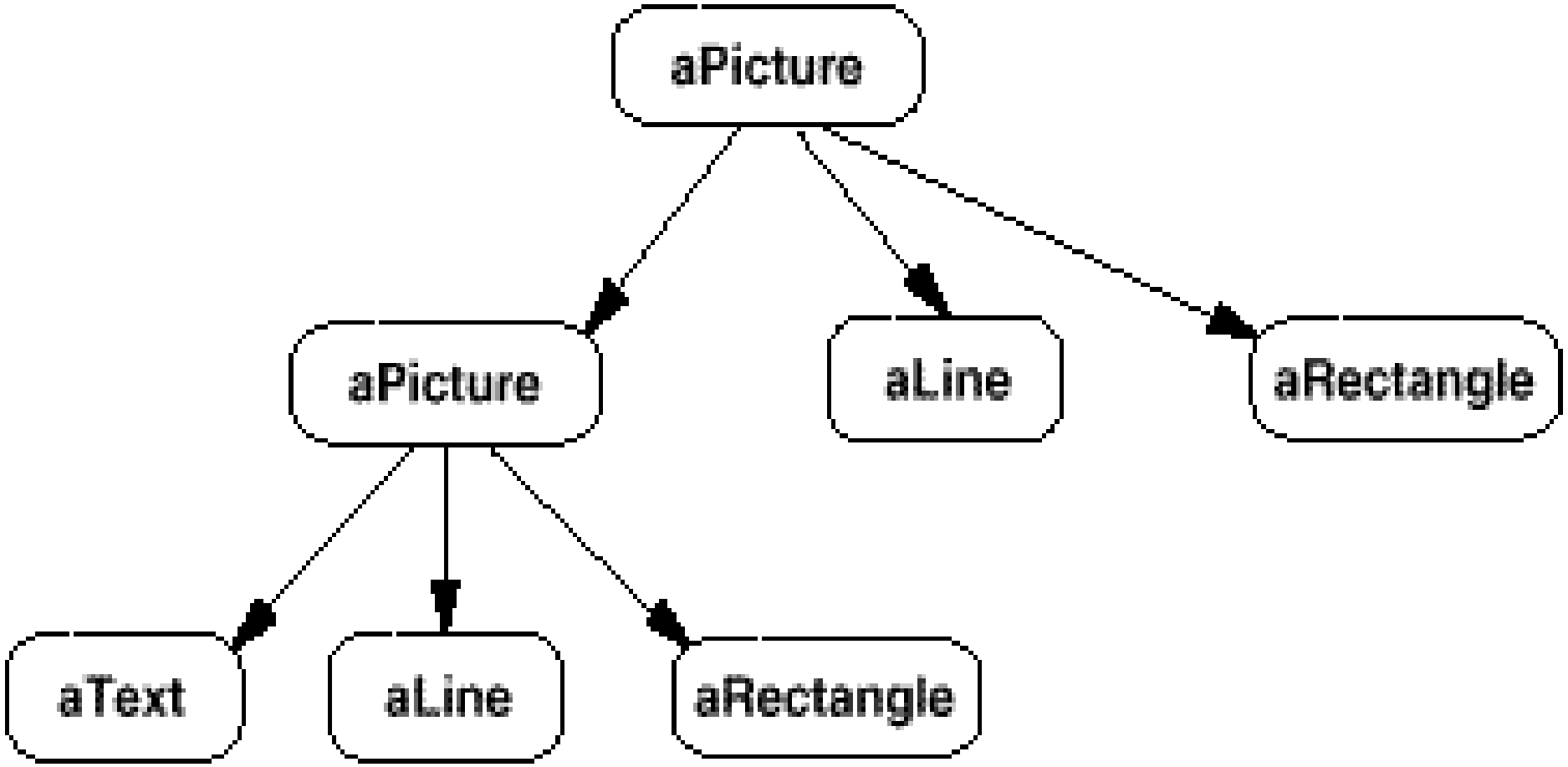
Problem with this approach:

- Code that uses these classes must treat primitive and container objects differently, even if the user treats them identically
- Having to distinguish these objects makes the application more complex
- Composite pattern describes how to use recursive composition so that clients don't have to make this distinction


Graphic
Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)
graphics
Line
Draw()
Rectangle
Draw()
Text
Draw()
Picture
Draw()
Add(Graphic g)
Remove(Graphic)
GetChild(int)
forall g in graphics
g.Draw()
add g to list of graphics

- An abstract class that represents primitives and their containers is the key to the Composite pattern
- For the graphics system, this class is Graphic
- Graphic declares operations like Draw that are specific to graphical objects
- It also declares operations that all composite objects share, such as operations or accessing and managing its children

- Subclasses Line, Rectangle, and Text (see fig) define primitive graphical objects
- These classes implement ‘Draw’ to draw lines, rectangles, and text, respectively
- The Picture class defines an aggregate of Graphic objects
- Picture implements Draw to call Draw on its children, and it implements child-related operations accordingly
- The following diagram shows a typical composite object structure of recursively composed Graphic objects:

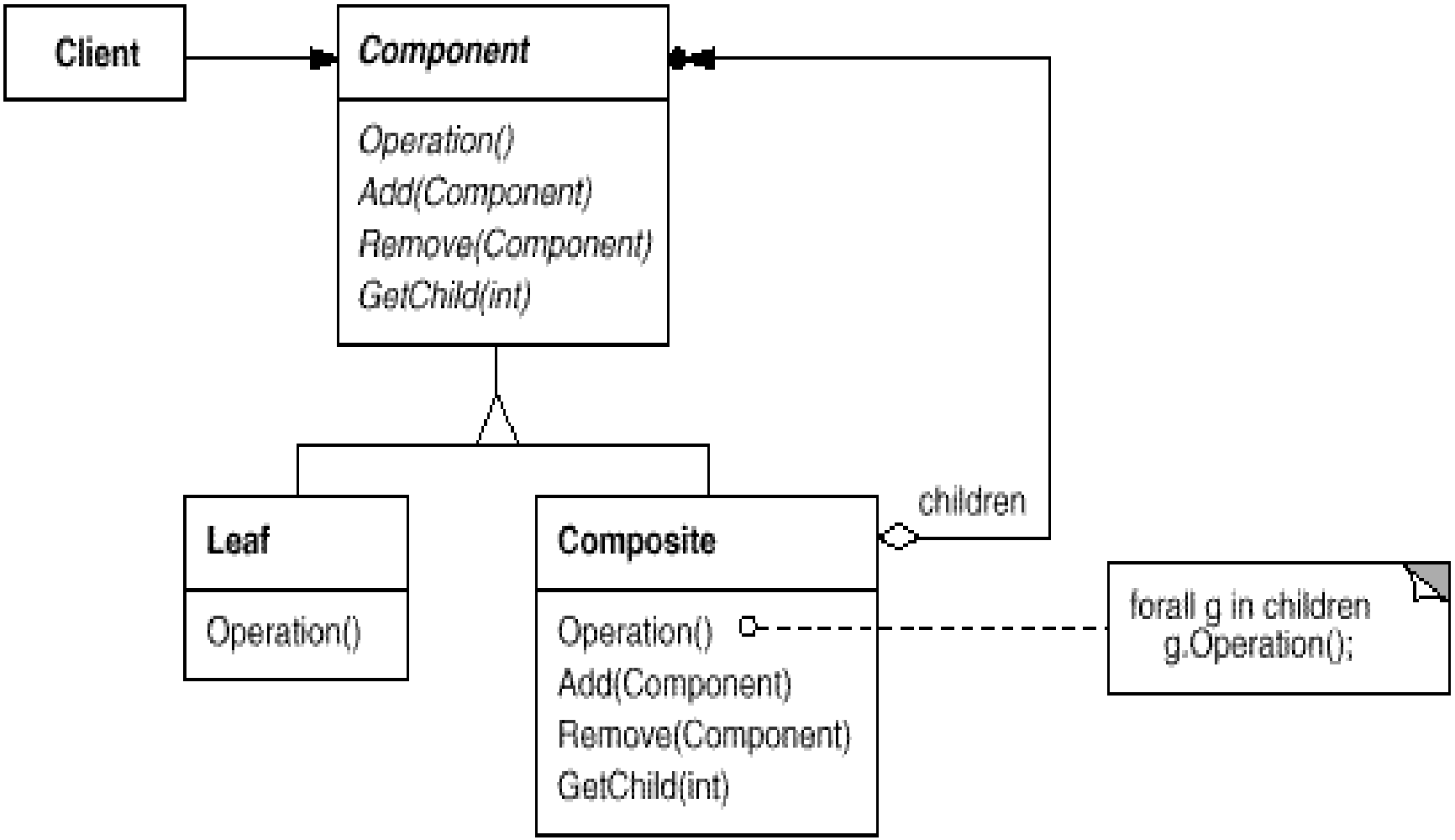

aPicture
aPicture
aLine
aRectangle
aText
aLine
aRectangle

Applicability

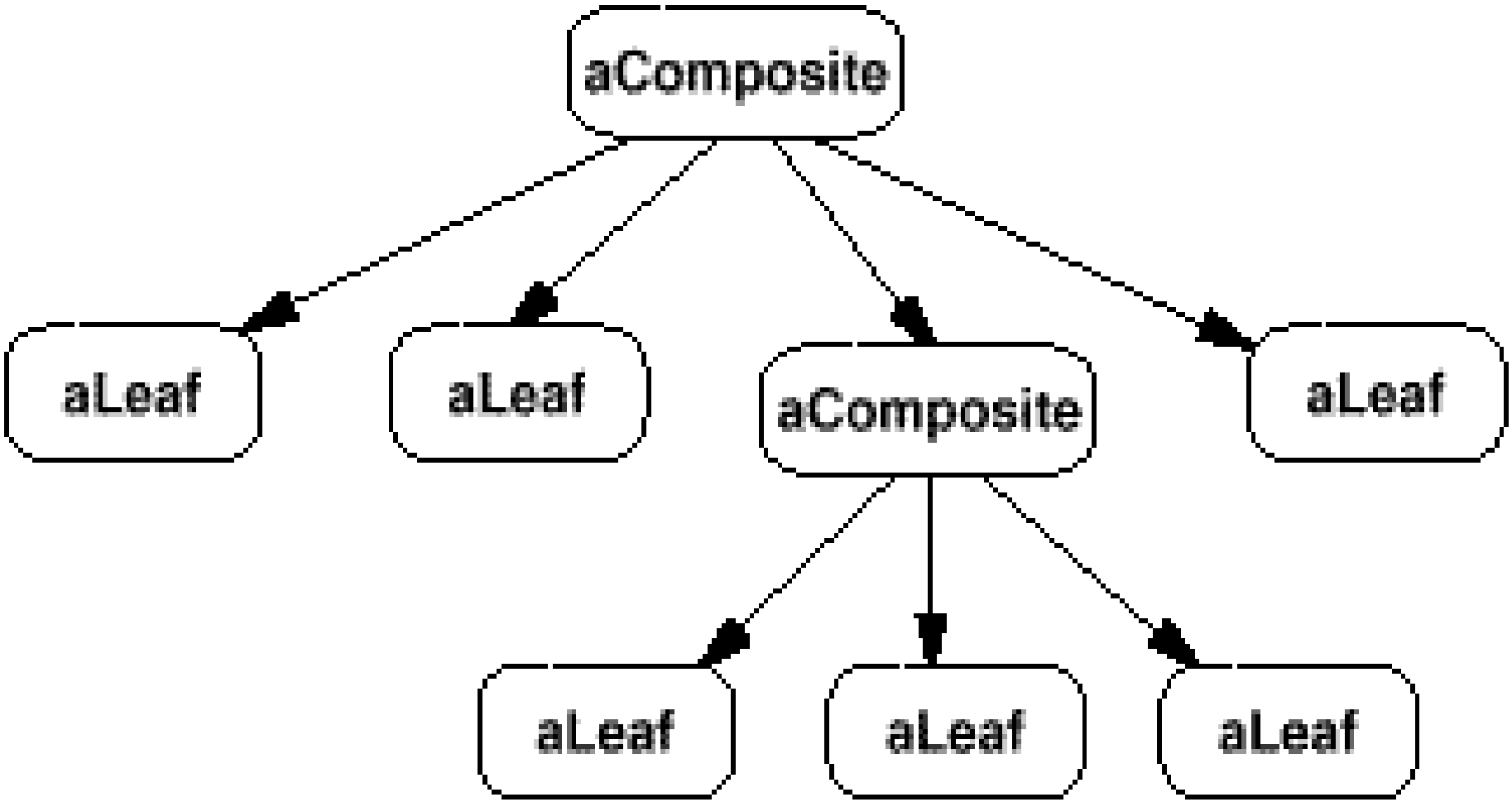
Use the Composite pattern when

- 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.

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();

A typical Composite object structure might look like this:



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** (Picture)
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client**
  - manipulates objects in 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
- Else it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding

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
  - Clients can treat composite structures and individual objects uniformly

- makes it easier to add new kinds of components
  - Newly defined Composite or Leaf subclasses work automatically with existing structures and client code
- can make your design overly general
  - The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite

Implementation

There are many issues to consider when implementing the Composite pattern:

- ***Explicit parent references.***
- ***Sharing components.***
- ***Maximizing the Component interface***
- ***Declaring the child management operations***

Explicit parent references:

- Maintaining references from child components to their parent can simplify the traversal and management of a composite structure.
- The parent reference simplifies moving up the structure and deleting a component
- Leaf and Composite classes can inherit the reference and the operations that manage it

Sharing components:

- It's often useful to share components
- E.g., to reduce storage requirements

Maximizing the Component interface:

- One of the goals is to make clients unaware of the Leaf or Composite classes they're using
- To attain this, the Component class should define as many common operations for Composite and Leaf classes as possible
- The Component class usually provides default implementations for these operations, and Leaf and Composite subclasses will override them

Declaring the child management operations:

- Although the Composite class *implements* the Add and Remove operations for managing children, which classes *declare* these operations in the Composite class hierarchy?
- Should we declare, or should we declare and define them only in Composite and its subclasses?

The decision involves a trade-off between safety and transparency:

- Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly.
  - It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.
- Defining child management in the Composite class gives you safety, because any attempt to add or remove objects from leaves will be caught at compile-time in a statically typed language like C++
  - But you lose transparency, because leaves and composites have different interfaces.

- We have emphasized transparency over safety in this pattern
  - If you opt for safety, then at times you may lose type information and have to convert a component into a composite
  - How can you do this without resorting to a type-unsafe cast?
- One approach is to declare an operation Composite* GetComposite() in the Component class
  - Component provides a default operation that returns a null pointer
  - The Composite class redefines this operation to return itself through the this pointer: c

Decorator

Intent

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

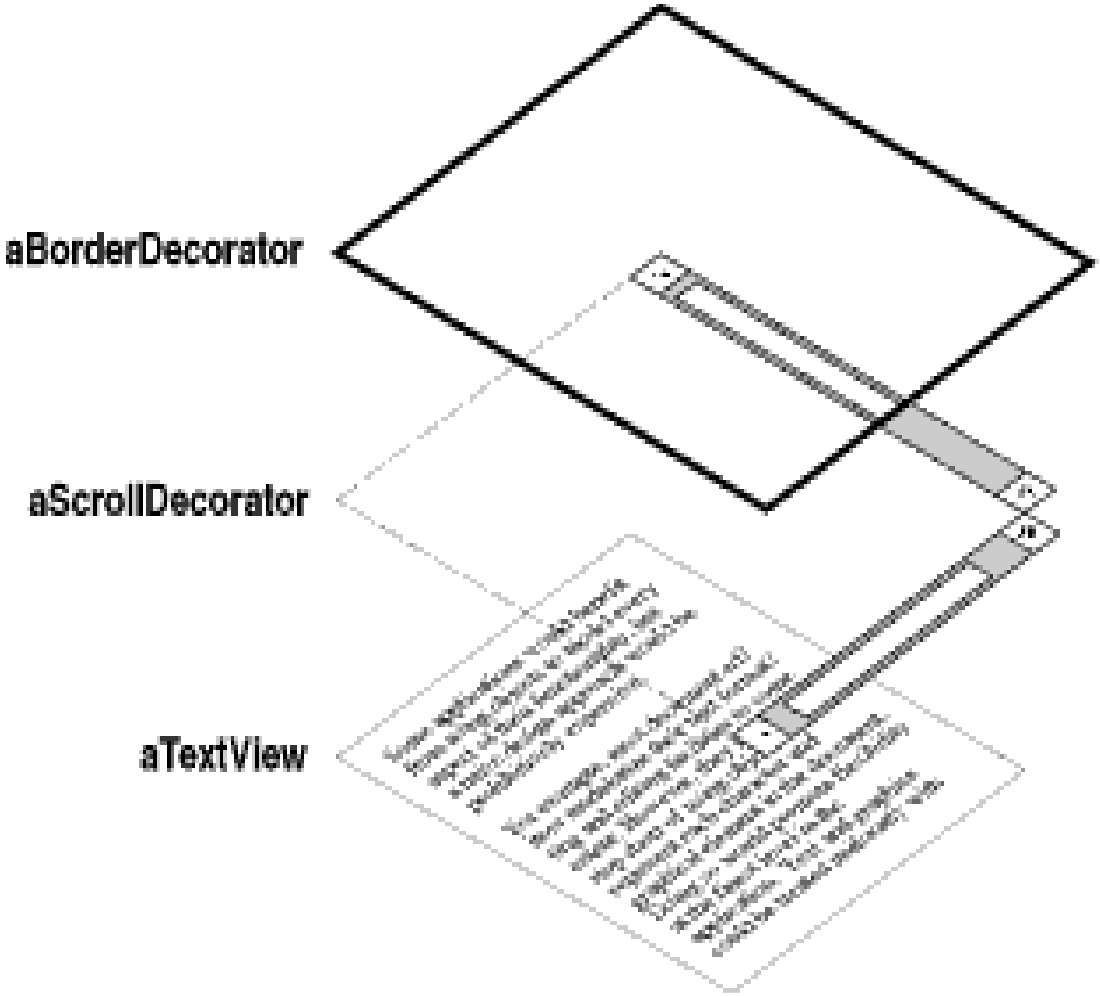
Also Known As

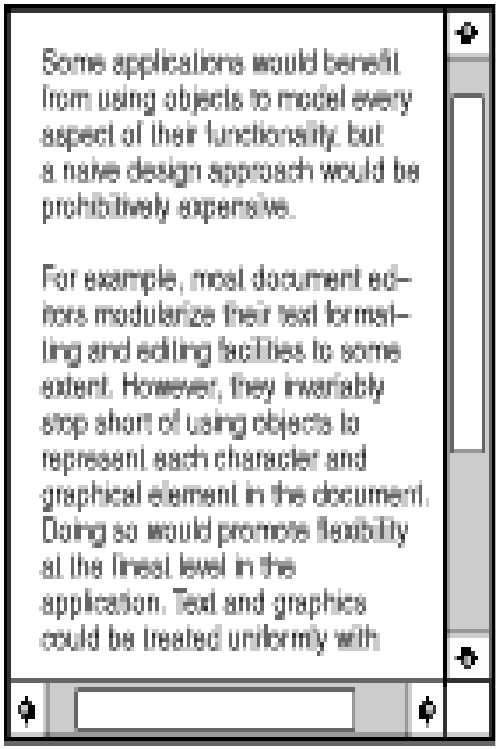
- Wrapper

Motivation

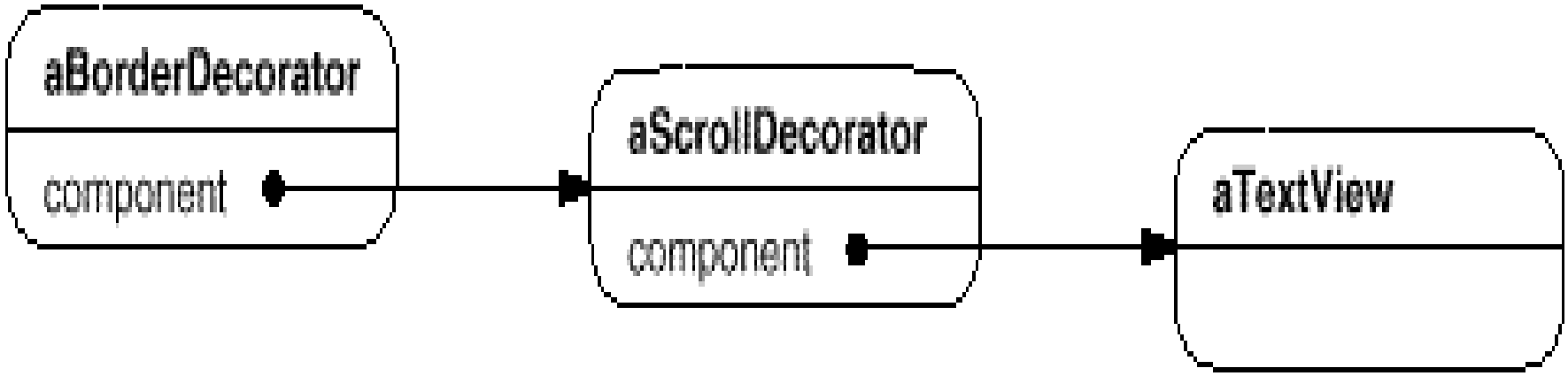
- Sometimes it might be required to add responsibilities to individual objects, not to an entire class
- E.g., A GUI toolkit, should let you add properties like borders or behaviors like scrolling to any user interface component
- Inheriting a border from another class puts a border around every subclass instance
- A client can't control how and when to decorate the component with a border.

- Enclose the component in another object that adds the border
- Enclosing object is called a **decorator**
- Decorator forwards requests to the component and may perform additional actions
- Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities
- E.g., suppose we have a TextView object that displays text in a window
  - TextView has no scroll bars by default, because we might not always need them and we can use a ScrollDecorator to add them
  - Suppose we also want to add a thick black border around the TextView, use a BorderDecorator to add
  - We simply compose the decorators with the TextView to produce the desired result

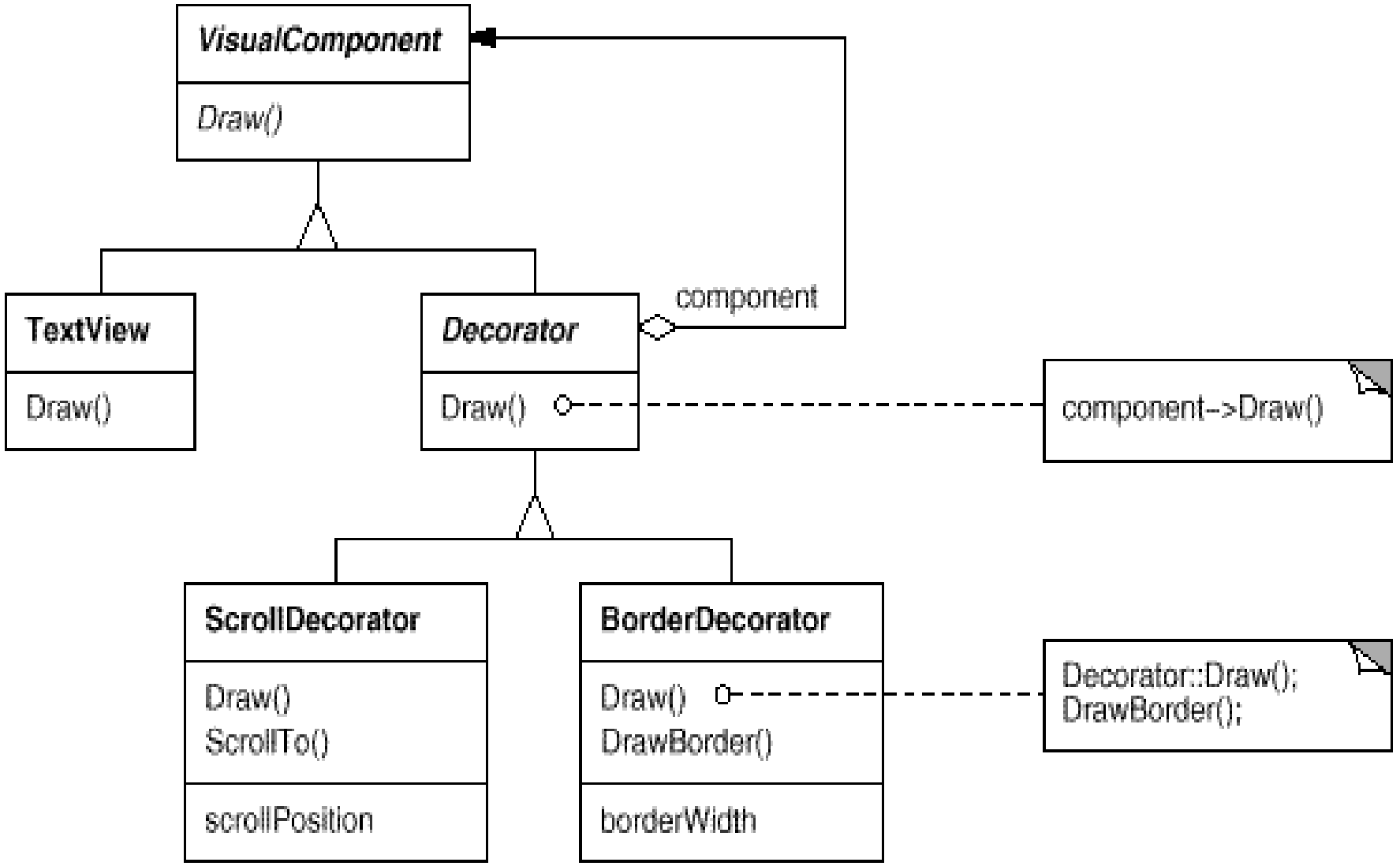

aBorderDecorator
aScrollDecorator
aTextView


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

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



The **ScrollDecorator** and **BorderDecorator** classes are subclasses of Decorator, an abstract class for visual components that decorate other visual components.

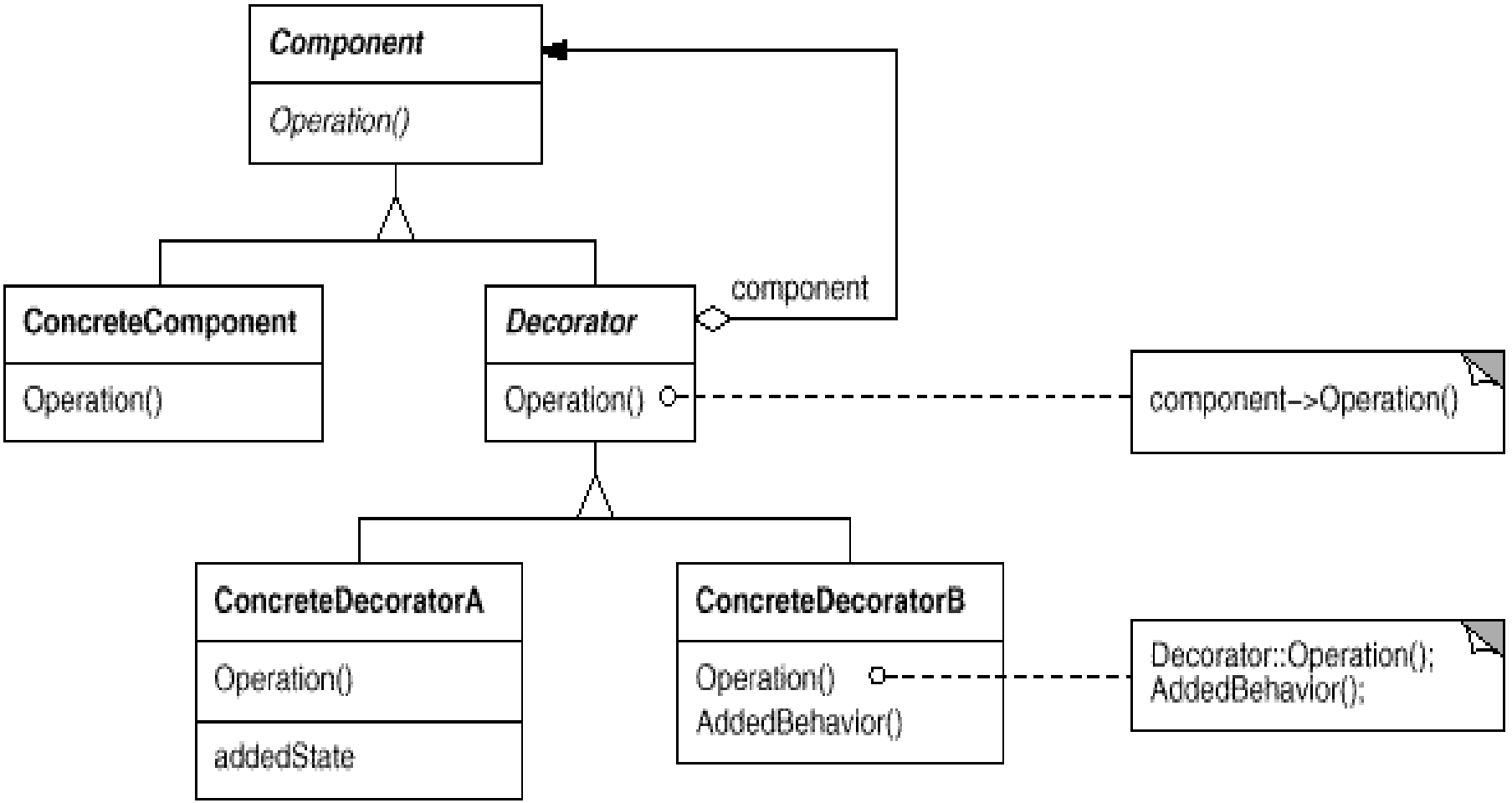

VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

- VisualComponent is the abstract class for visual objects
- It defines their drawing and event handling interface
- Decorator subclasses are free to add operations for specific functionality
- E.g., ScrollDecorator's ScrollTo operation lets other objects scroll the interface *if* they know there happens to be a ScrollDecorator object in the interface.
- The important aspect of this pattern is that it lets decorators appear anywhere a VisualComponent can.

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 subclassing is impractical
  - Sometimes numerous independent extensions are possible and would bombard the subclasses to support every combination


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();

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, ScrollDecorator)
  - adds responsibilities to the component.

Collaborations

- Decorator forwards requests to its Component object
- It may perform additional operations before and after forwarding request

Consequences

The Decorator pattern has at least two key benefits and two liabilities:

1. *More flexibility than static inheritance.*

- Decorator pattern provides more flexible way to add responsibilities to objects at run-time simply by attaching and detaching them
- Inheritance creates new class for each additional responsibility which creates many classes and increases the complexity of a system
- Decorators also make it easy to add a property twice
- E.g., to give a TextView a double border, attach two BorderDecorators
- Inheriting from a Border class twice is error-prone at best.

2. *Avoids feature-laden classes high up in the hierarchy*

- Instead of trying to support all foreseeable features in a complex, customizable class, define a simple class and add functionality incrementally with Decorator objects
- Functionality can be composed from simple pieces
- It's also easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions
- Extending a complex class tends to expose details unrelated to the responsibilities you're adding

3. *A decorator and its component aren't identical*

- Decorator acts as a transparent enclosure
- But a decorated component is not identical to the component itself
- Hence you shouldn't rely on object identity when you use decorators

4. *Lots of little objects*

- The objects differ only in the way they are interconnected, not in their class or in the value of their variables
- Although these systems are easy to customize by those who understand them, they can be hard to learn and debug

Implementation

Several issues should be considered when applying the Decorator pattern:

1. *Interface conformance*

- A decorator object's interface must conform to the interface of the component it decorates.
- ConcreteDecorator classes must inherit from a common class

2. *Omitting the abstract Decorator class*

- There's no need to define an abstract
- Decorator class when you only need to add one responsibility

3. *Keeping Component classes lightweight*

- To ensure a conforming interface, components and decorators must descend from a common Component class
- It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data
- Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.

4. *Changing the skin of an object versus changing its guts*

- Decorator is a skin over an object that changes its behaviour

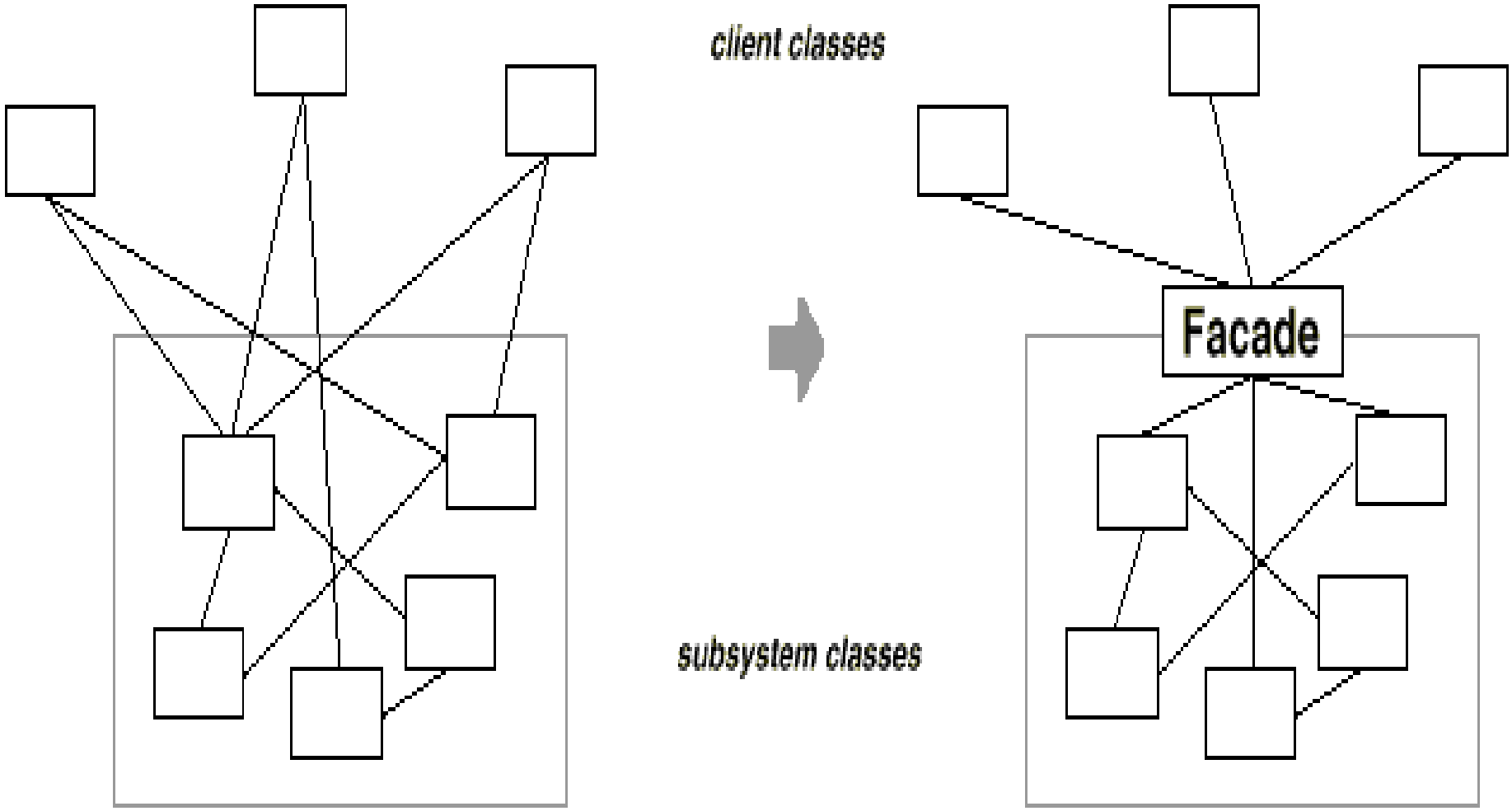
Façade

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

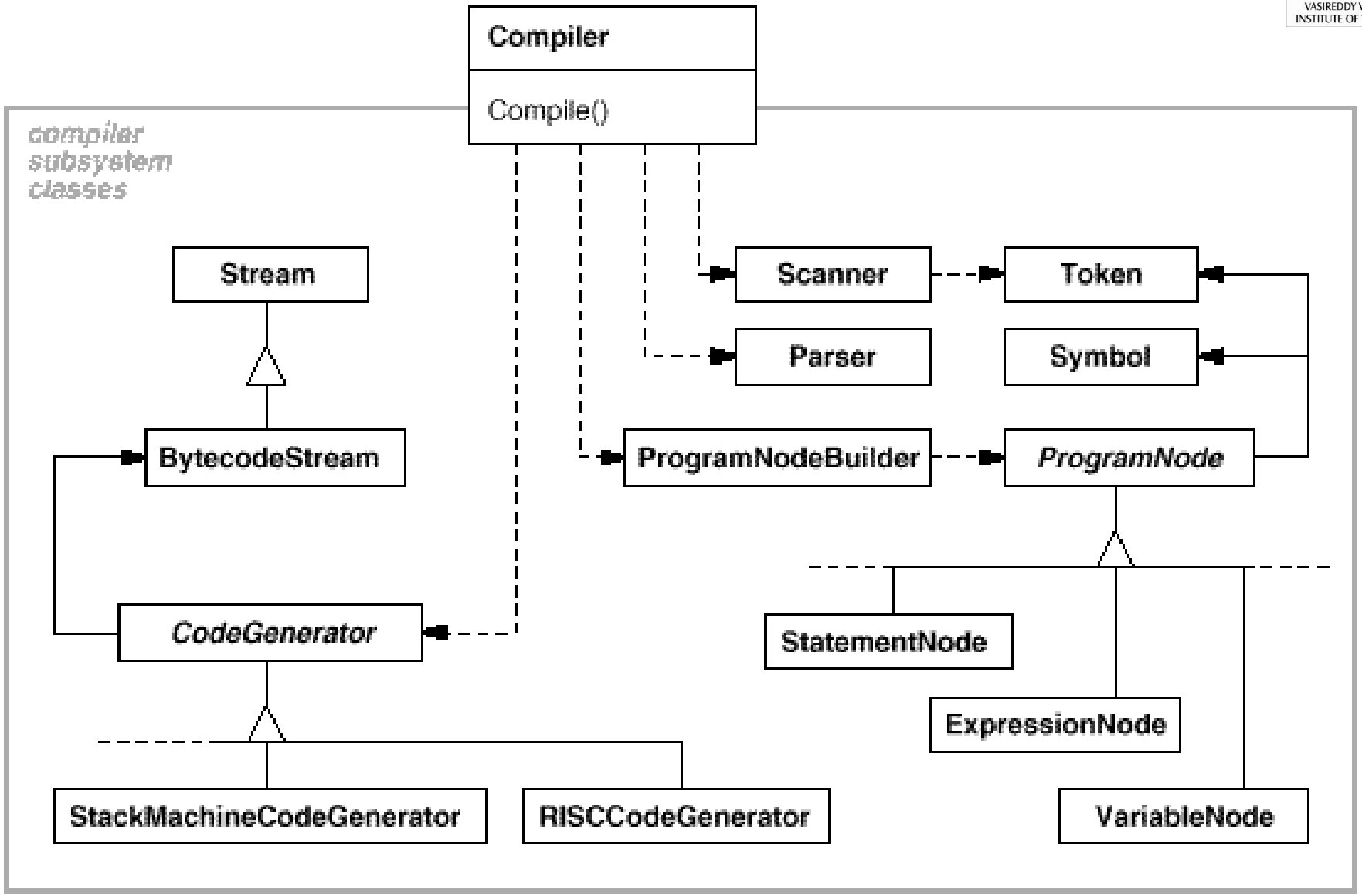
Motivation

- Structuring a system into subsystems helps reduce complexity
- Goal is to minimize communication & 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.


client classes
Facade
subsystem classes

- E.g.,a programming environment that gives applications access to its compiler subsystem
- Subsystem contains classes such as:
  - Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler
- Some specialized applications might need to access these classes directly
- Clients of a compiler don't care about details like parsing and code generation, they just want to compile some code
- For them, powerful but low-level interfaces in the compiler subsystem only complicate their task

- To provide a higher-level interface, compiler subsystem includes a Compiler class, which defines a unified interface to the compiler's functionality
- The Compiler class acts as a façade, It offers clients a single, simple interface to the compiler subsystem
- It glues together the classes that implement compiler functionality without hiding them completely
- Compiler facade makes it easier for most programmers without hiding the lower-level functionality


Compiler
Compile()
compiler subsystem classes
Stream
Scanner
Token
Parser
Symbol
BytecodeStream
ProgramNodeBuilder
ProgramNode
CodeGenerator
StatementNode
ExpressionNode
StackMachineCodeGenerator
RISCCodeGenerator
VariableNode

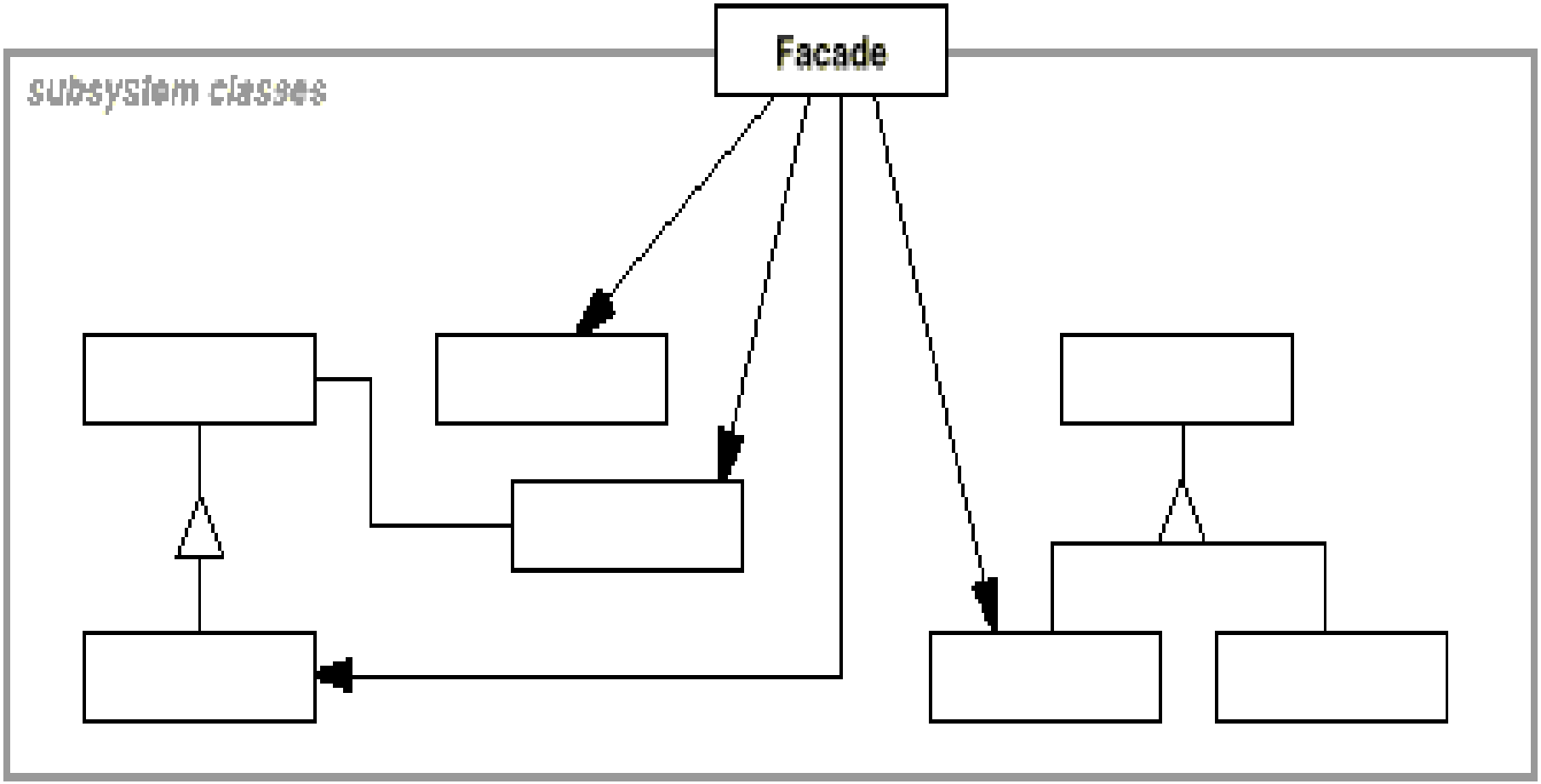
Applicability

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem.
  - Subsystems often get more complex as they evolve
  - Most patterns, when applied, result in more and smaller classes
  - This makes the subsystem more reusable & easier to customize, but it also becomes harder to use for clients that don't need to customize it
  - A facade can provide a simple default view of the subsystem that is good enough for most clients
  - Only clients needing more customizability will need to look beyond the facade.

- 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.
- you want to layer your subsystems
  - Use facade to define an entry point to each subsystem
  - If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Structure



Participants

- **Facade** (Compiler)
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **subsystem classes** (Scanner, Parser, ProgramNode, etc.)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of façade i.e., they keep no references to it

Collaborations

- Clients send requests to Facade, which forwards them to the appropriate subsystem object(s)
  - Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
- Clients that use the facade don't have to access its subsystem objects directly.

Consequences

The Facade pattern offers the following benefits:

1. It shields clients from subsystem components, by reducing the no.of objects that clients deal with & making the subsystem easier to use
2. It promotes weak coupling between the subsystem and its clients
    1. Facades help layer a system and the dependencies between objects
    2. They can eliminate complex or circular dependencies.
3. It doesn't prevent applications from using subsystem classes if they need to
    1. Thus you can choose between ease of use and generality.

Implementation

Consider following issues in implementing facade:

- 1. *Reducing client-subsystem coupling:*
  - Coupling between clients & subsystem is reduced by making Facade an abstract class with concrete sub-classes for different implementations of a subsystem
  - Clients communicate with subsystem through the interface of the abstract Facade class
- 2. *Public versus private subsystem classes:*
  - Public interface to a subsystem consists of classes that all clients can access, Private interface is just for subsystem extenders
  - The Facade class is part of the public interface, of course, but it's not the only part
  - Making subsystem classes private would be useful, but few O-O languages support it.

Flyweight

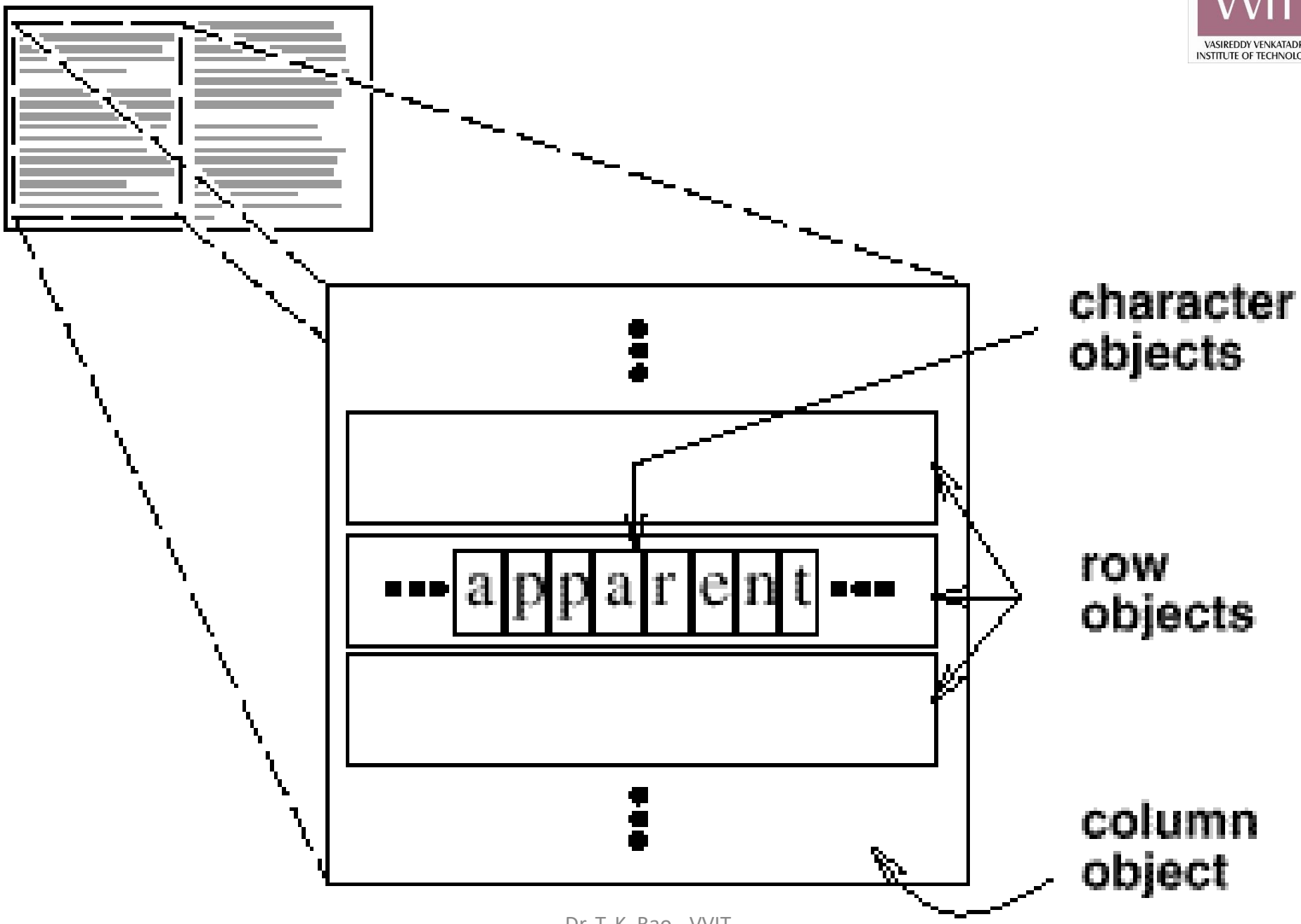
Intent

- Use sharing to support large numbers of fine-grained objects efficiently

Motivation

- Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive
- E.g., most document editor implementations have text formatting and editing facilities that are modularized to some extent
- O-O document editors typically use objects to represent embedded elements like tables and figures

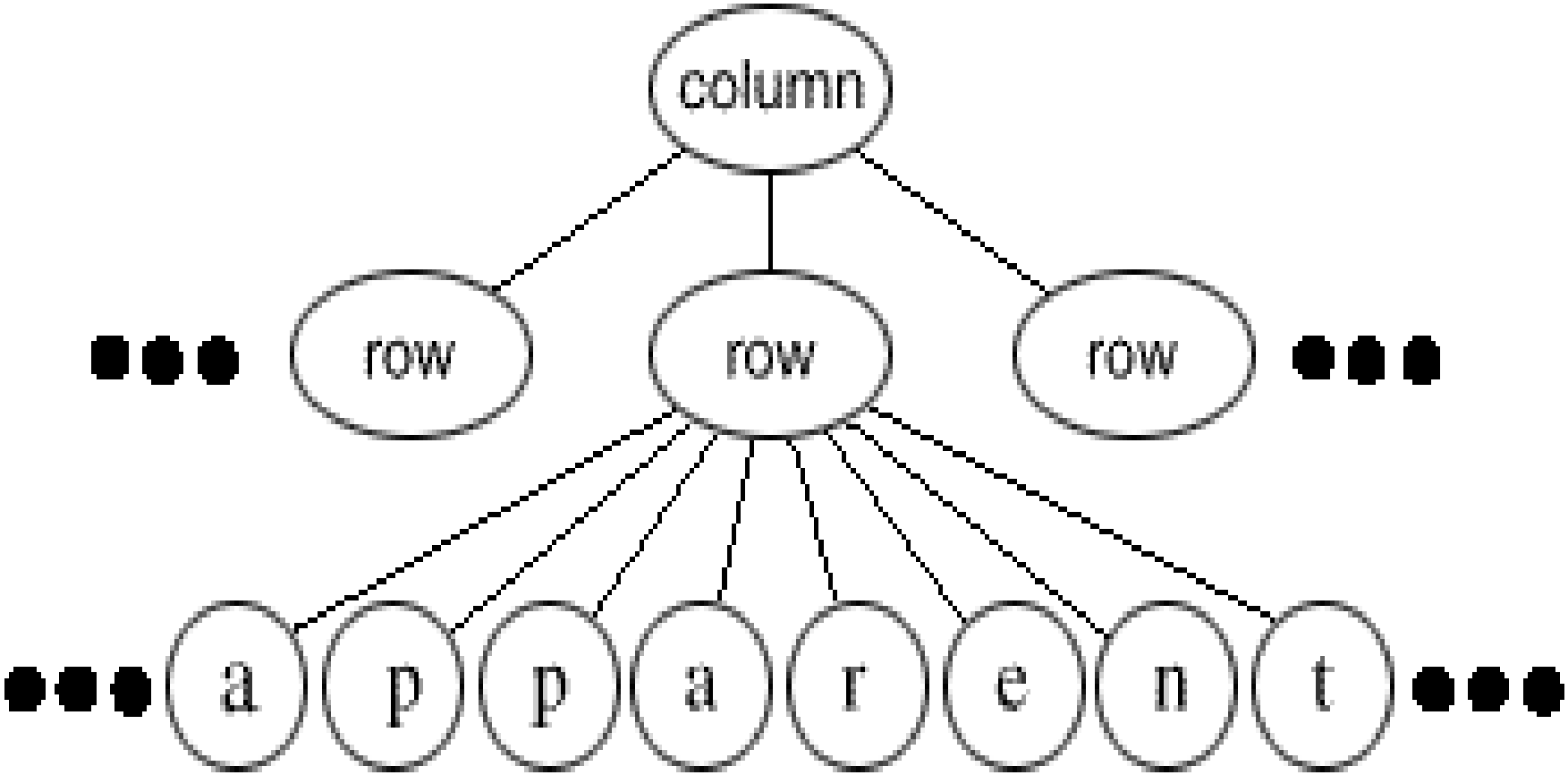
- They usually stop short of using an object for each character in the document, even though doing so would promote flexibility at the finest levels in the application
- Characters and embedded elements could then be treated uniformly with respect to how they are drawn and formatted
- The application could be extended to support new character sets without disturbing other functionality
- The application's object structure could mimic the document's physical structure
- The following diagram shows how a document editor can use objects to represent characters


character objects
row objects
column object
apparent

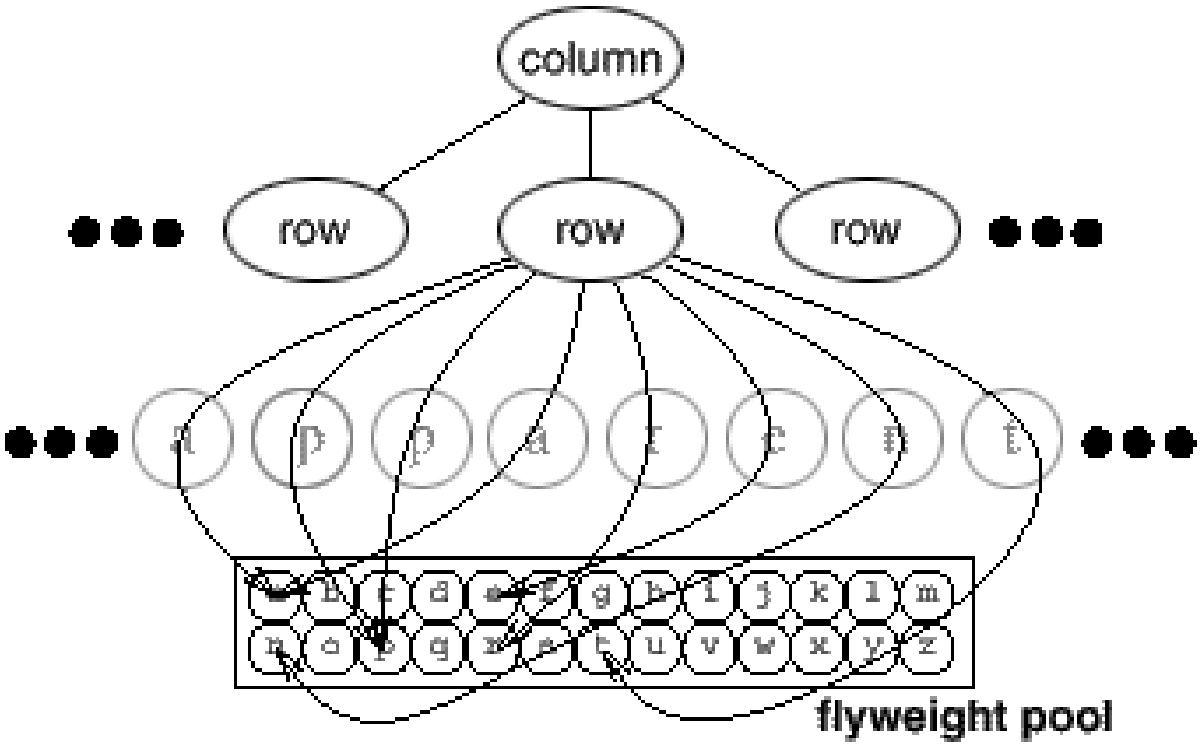
- The drawback of such a design is its cost
- Documents require thousands of character objects, which consume huge memory and unacceptable run-time
- Flyweight pattern describes how to share objects to allow their use at fine granularities at low cost.
- A **flyweight** is a shared object that can be used in multiple contexts simultaneously
- Flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared

- Flyweights cannot make assumptions about the context in which they operate
- The key concept here is the distinction between **intrinsic** and **extrinsic** state
- Intrinsic state is stored in the flyweight
- It consists of information that's independent of the flyweight's context, thereby making it sharable
- Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared
- Client objects are responsible for passing extrinsic state to the flyweight when it needs it

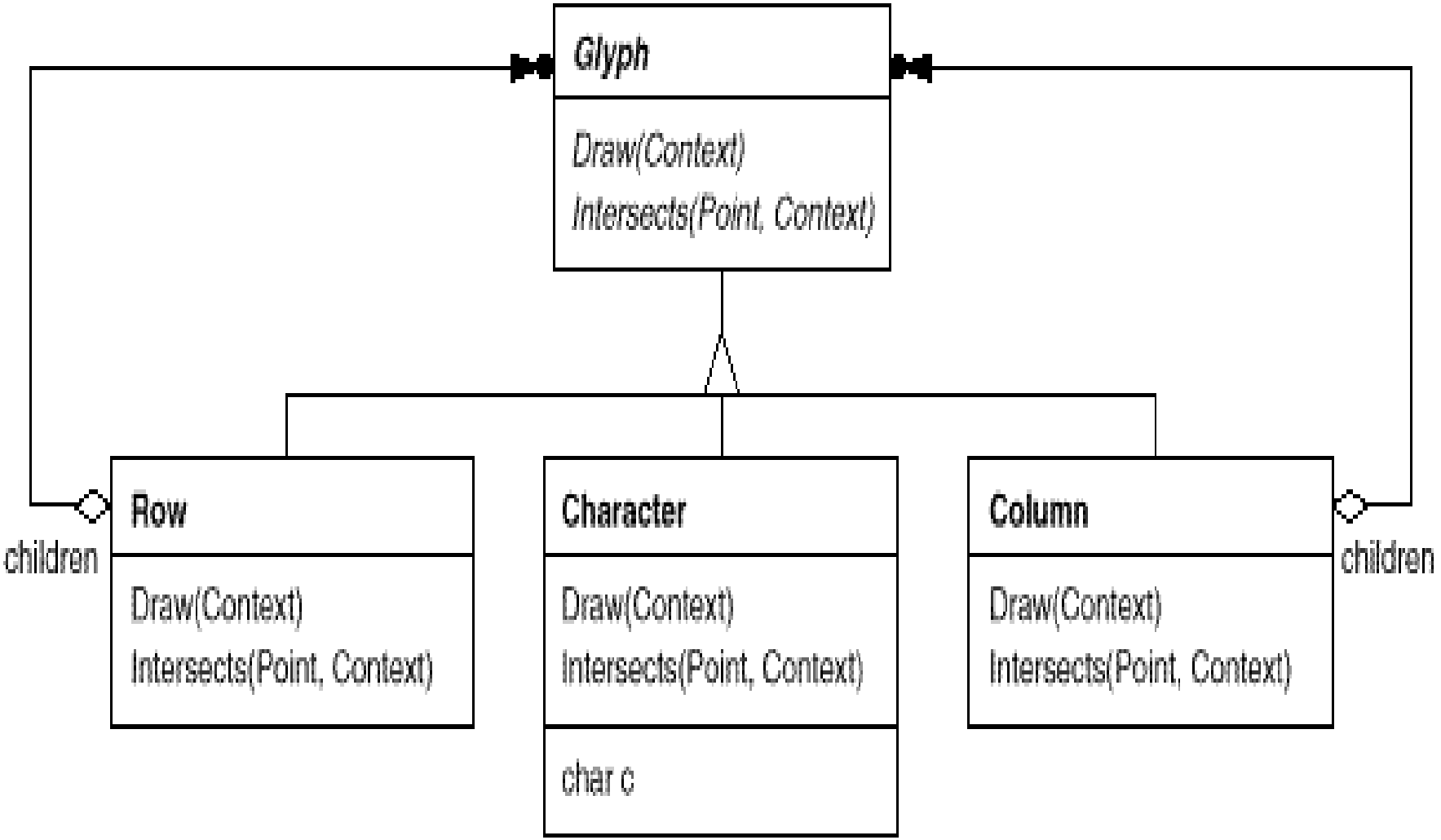
- Flyweights model concepts or entities that are normally too plentiful to represent with objects.
- E.g., a document editor can create a flyweight for each letter
- Each flyweight stores a character code
- But its coordinates, its style can be determined from the text layout algorithms and formatting commands in effect wherever it appears
- The character code is intrinsic state, while the other information is extrinsic
- Logically there is an object for every occurrence of a given character in the document:


column
row
row
row
a
p
p
a
r
e
n
t

- Physically, there is one shared flyweight object per character, and it appears in different contexts
- Each occurrence of a particular character object refers to the same instance in the shared pool of flyweight objects:



- The class structure for these objects is shown next
- Glyph is the abstract class for graphical objects, some of which may be flyweights.
- Operations that may depend on extrinsic state have it passed to them as a parameter
- E.g., Draw and Intersects must know which context the glyph is in before they can do their job.


Glyph
Draw(Context)
Intersects(Point, Context)
Row
Draw(Context)
Intersects(Point, Context)
Character
Draw(Context)
Intersects(Point, Context)
char c
Column
Draw(Context)
Intersects(Point, Context)
children
children

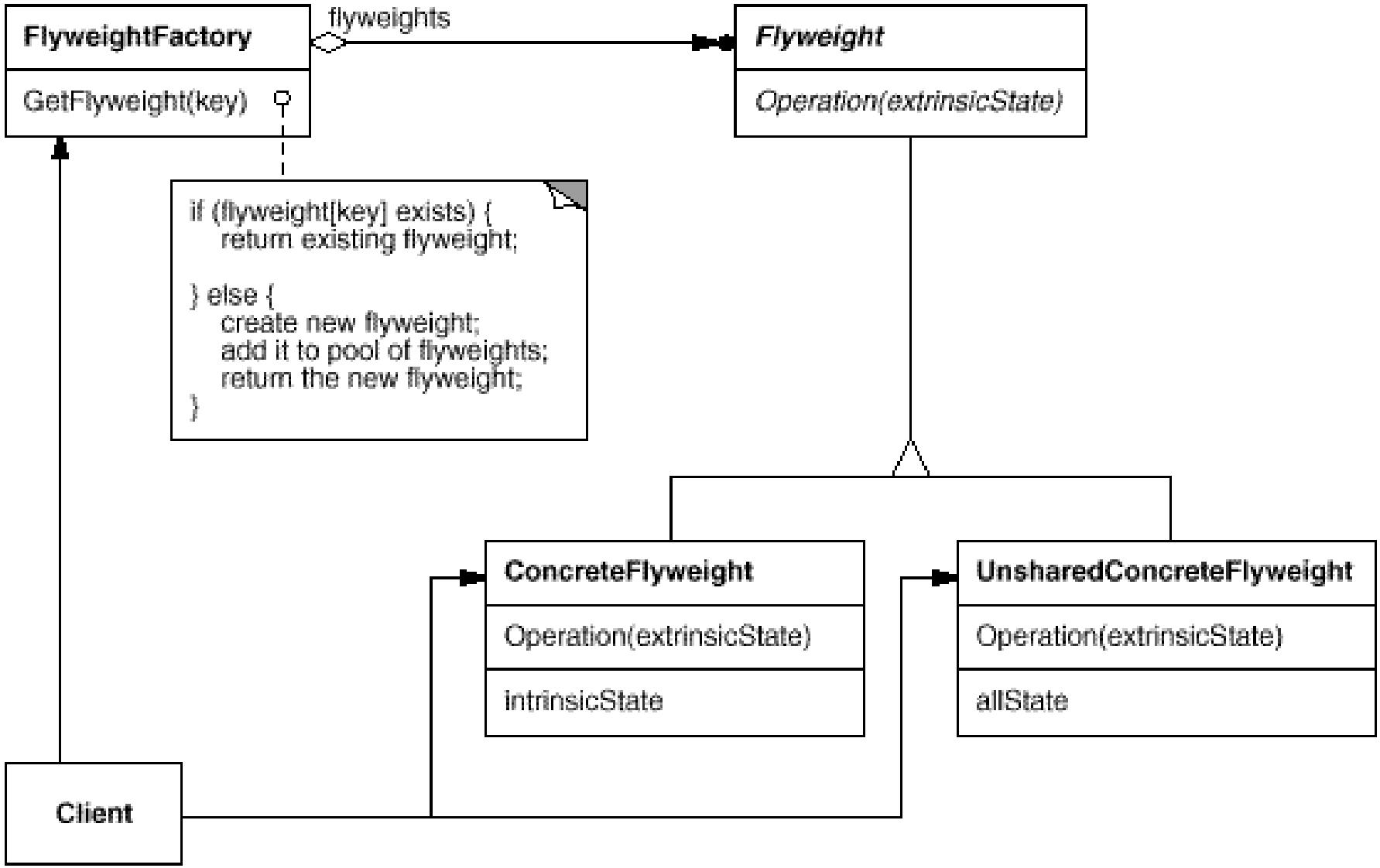
- A flyweight representing the letter "a" only stores the corresponding character code;
- It doesn't need to store its location or font
- Clients supply the context-dependent information that the flyweight needs to draw itself
- E.g., a Row glyph knows where its children should draw themselves so that they are tiled horizontally
- Thus it can pass each child its location in the draw request
- Since no.of different character objects is << no.of characters, total no.of objects is << what a naive implementation would use

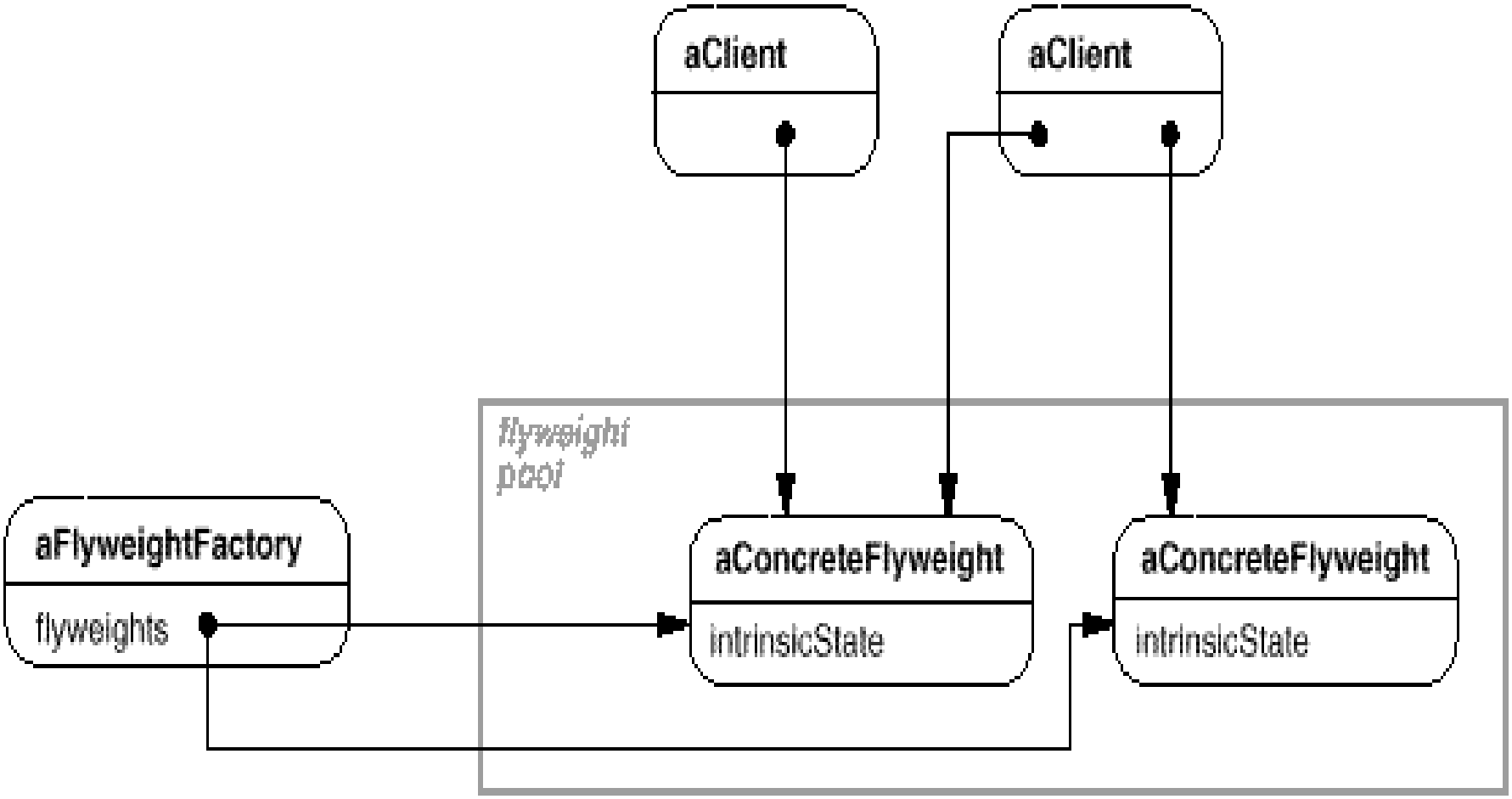
Applicability

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 tests will return true for conceptually distinct objects.

Structure


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client


aClient
aClient
flyweight pool
aFlyweightFactory
flyweights
aConcreteFlyweight
intrinsicState
aConcreteFlyweight
intrinsicState

Participants

- **Flyweight**
  - 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, Column)
  - not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).

- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests 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).

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

Consequences

- Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state
- However, such costs are offset by space savings, which increase as more flyweights are shared
- 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.

Implementation

Consider the following issues when implementing the Flyweight pattern:

- *Removing extrinsic state*
  - The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects
  - Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing
  - Ideally, extrinsic state can be computed from a separate object structure, one with far smaller storage requirements.

- *Managing shared objects*
  - Because objects are shared, clients shouldn't instantiate them directly
  - FlyweightFactory lets clients locate a particular flyweight
  - FlyweightFactory objects often use an associative store to let clients look up flyweights of interest
  - E.g., the flyweight factory in the document editor example can keep a table of flyweights indexed by character codes
  - The manager returns the proper flyweight given its code, creating the flyweight if it does not already exist.

Proxy

Intent

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

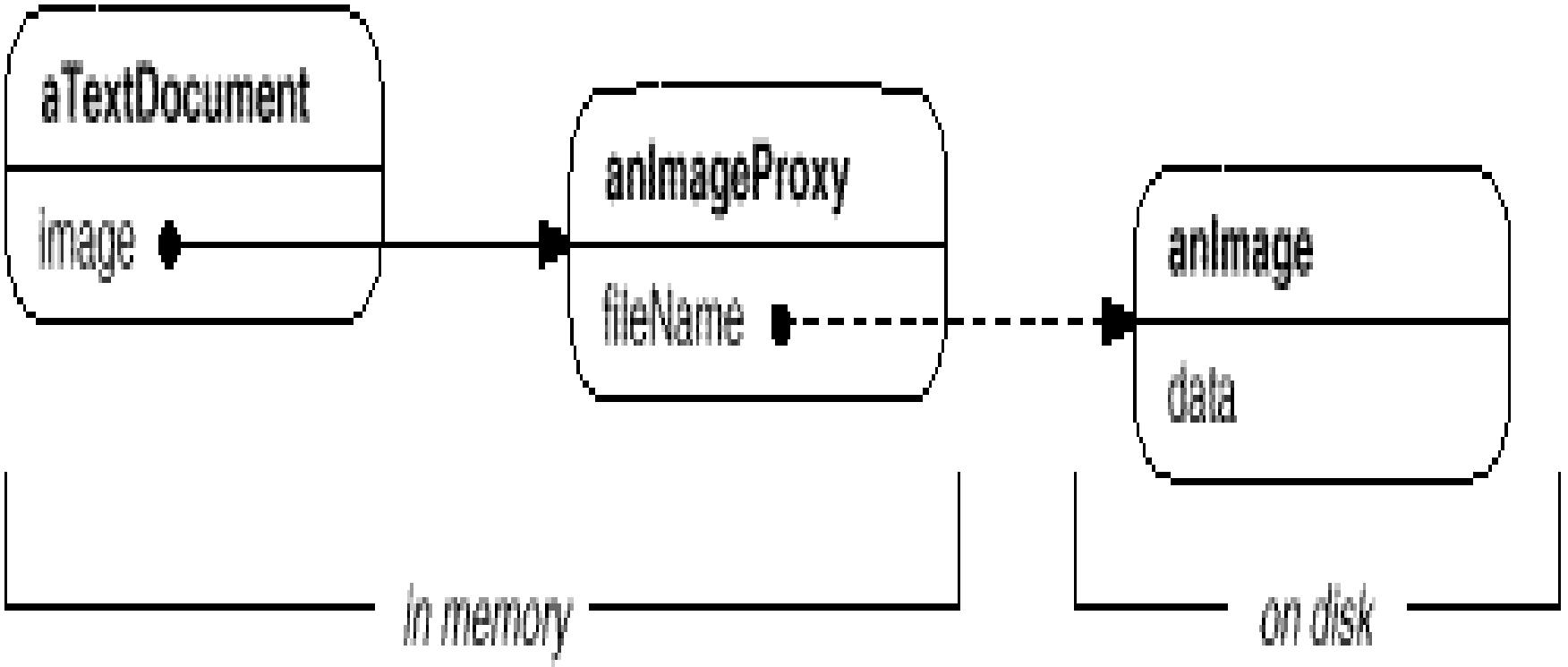
Also Known As

- Surrogate

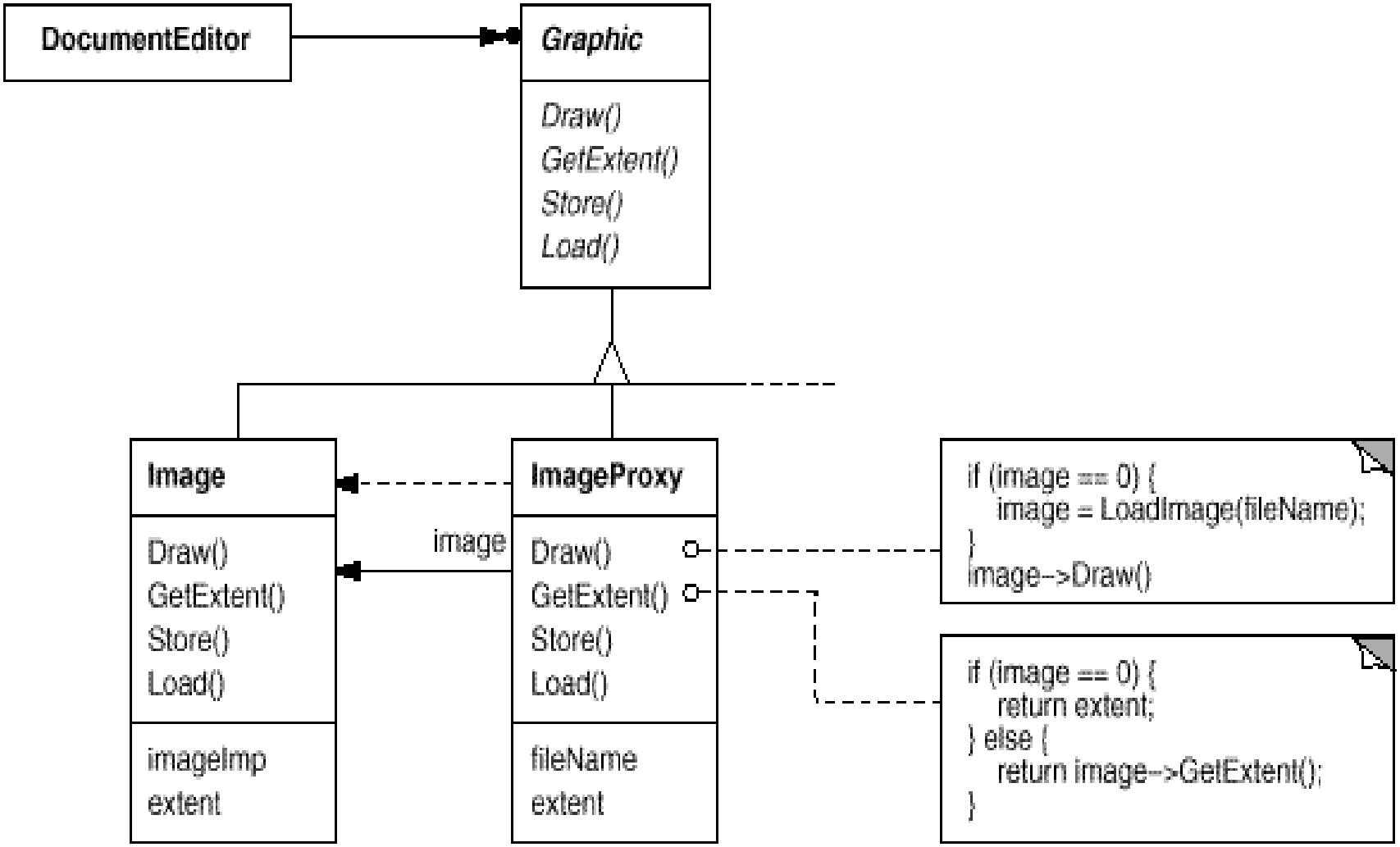
Motivation

- One reason for controlling access to an object is to suspend the cost of its creation and initialization until we actually need to use it
- E.g., graphical objects, (large raster images), are expensive to create, but opening a document should be fast
- creating expensive objects at once should be avoided when document is opened
- Not all of these objects will be visible in the document at the same time

- Constraints suggests creating expensive objects *on demand*
- But what do we put in the document in place of the image?
- How image is created on demand so that we don't complicate the editor's implementation?
- This optimization shouldn't impact rendering and formatting code
- Solution is to use another object, an image **proxy**, that acts as a stand-in for the real image
- Proxy acts as image and takes care of instantiating it when required.


aTextDocument
image
anImageProxy
fileName
anImage
data
In memory
on disk

- Image proxy creates real image, only when document editor requests it to display
- Proxy forwards subsequent requests to the image
- It must keep a reference to the image after creating it
- Assume that images are stored in separate files
- File name is used as the reference to real object
- Proxy also stores its **extent**, i.e., width & height
- Extent lets the proxy respond to requests for its size from the formatter without instantiating the image


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();
}

- Document editor accesses embedded images through the interface defined by abstract Graphic class
- ImageProxy is a class for images that are created on demand, maintains file name as a reference to image
- Reference is invalid until proxy instantiates real image
- Draw operation makes sure the image is instantiated before forwarding it the request
- GetExtent forwards request to image only if it is instantiated; else ImageProxy returns extent it stores

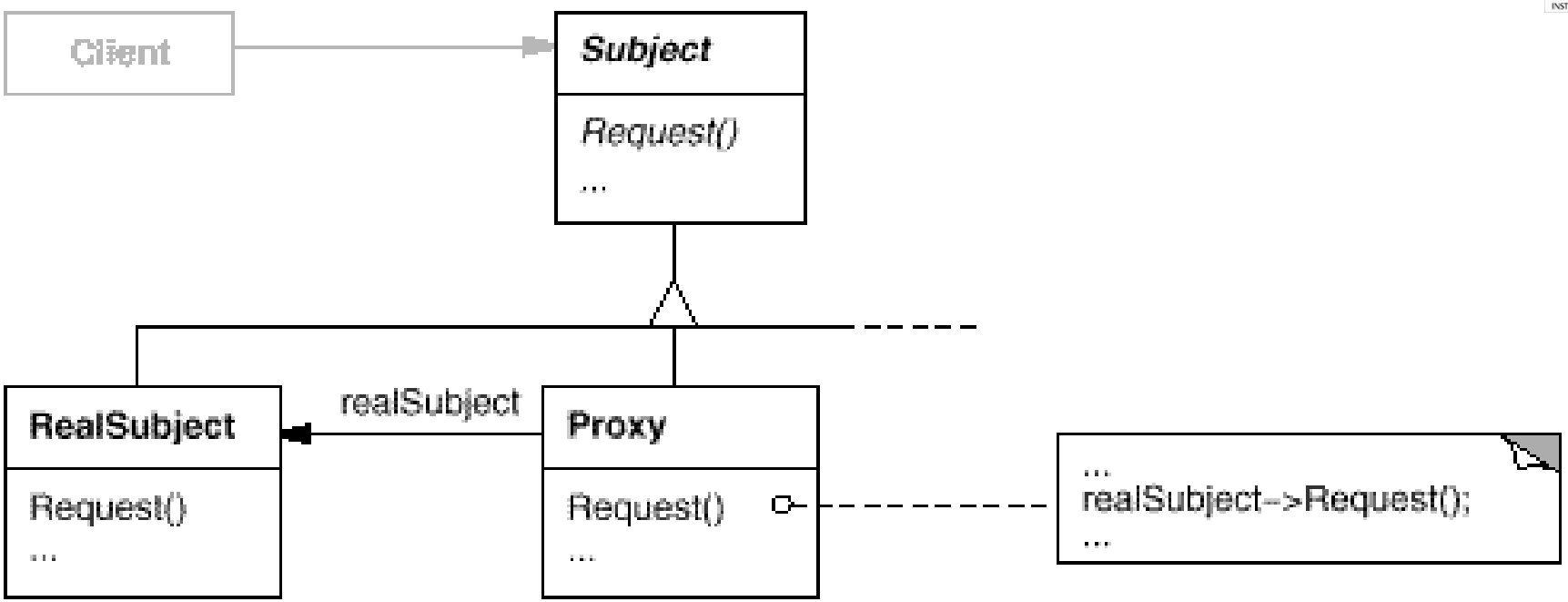
Applicability

- Proxy is applicable whenever there is a need for a more versatile reference to an object than a simple pointer
- Few common situations in which 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 (e.g. ImageProxy)
  - 3. A **protection proxy** controls access to the original object

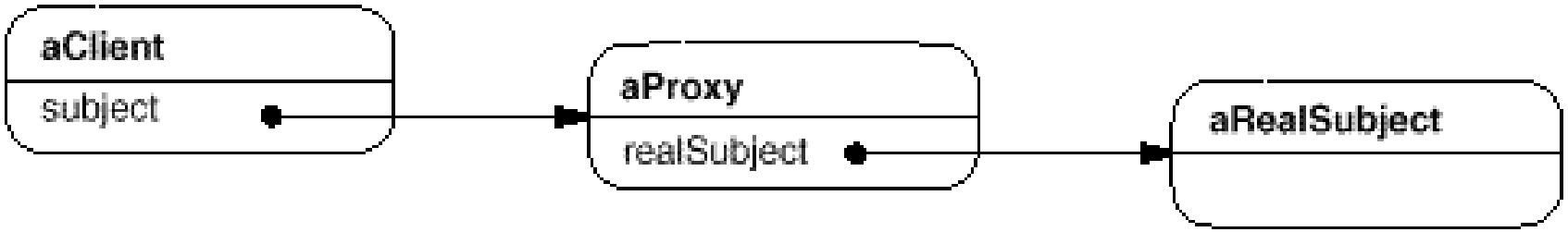
    These are useful when objects should have different access rights

    E.g., KernelProxies in Choices OS provide protected access to OS objects

- 4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed
- Typical uses include:
  - counting the number of references to real object so that it can be freed automatically 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.



Possible object diagram of a proxy structure at run-time:



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 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.

- **Subject** (Graphic)
  - defines common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected
- **RealSubject** (Image)
  - defines the real object that the proxy represents

Collaborations

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

Consequences

- 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