

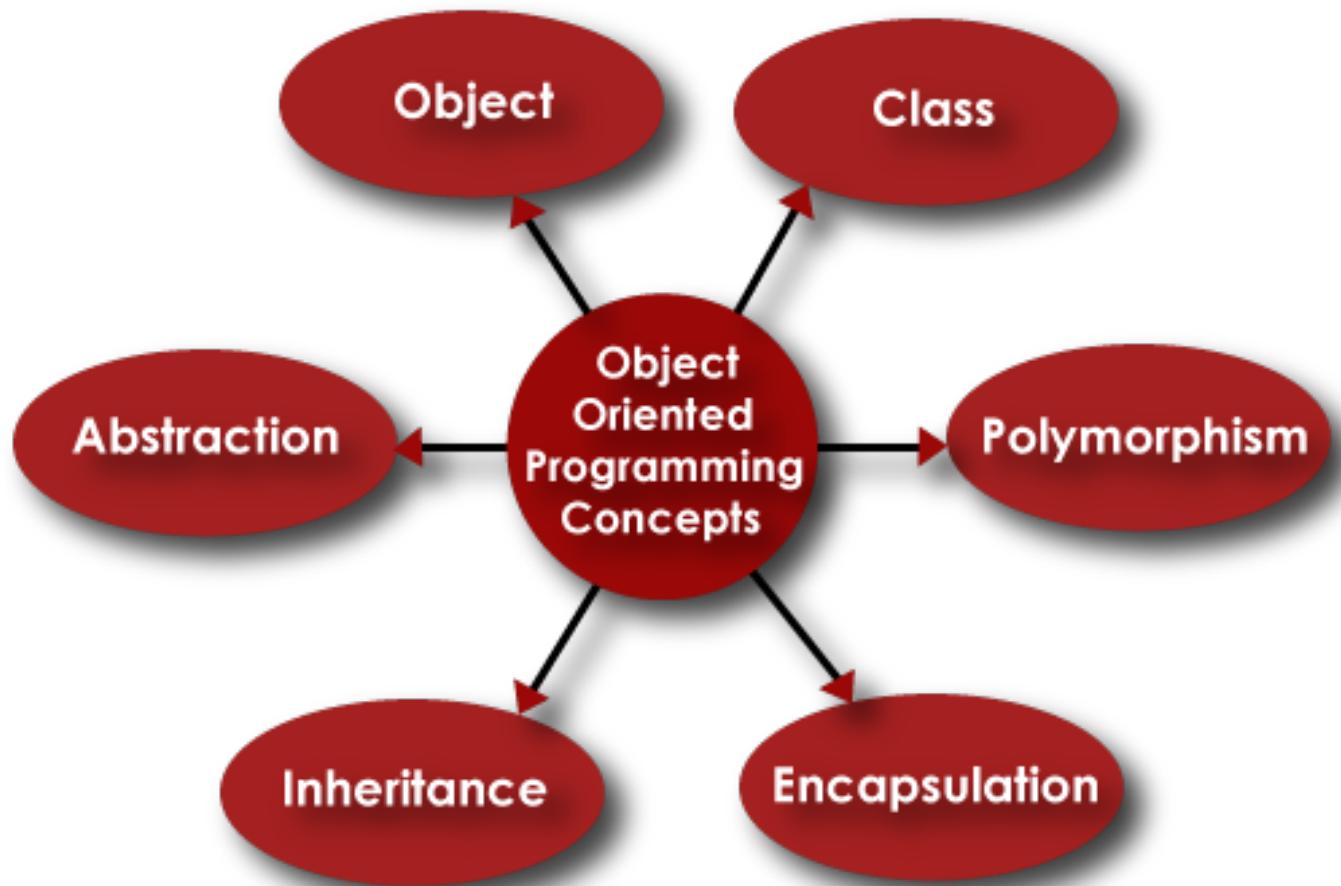
# **Module 3**

# **OOP Concepts**

- Object Oriented is a popular design approach for analyzing and designing an application
- Most of the languages like C++, Java, .net are use object oriented design concept
- Object-oriented concepts are used in the design methods such as classes, objects, polymorphism, encapsulation, inheritance, dynamic binding, information hiding, interface constructor destructor

- The main advantage of object oriented design is that **improving the software development and maintainability**
- Another advantage is that **faster and low cost development**, and **creates a high quality software**
- The disadvantage of the object-oriented design is that **larger program size** and **it is not suitable for all types of program**

- The different terms related to object design are:

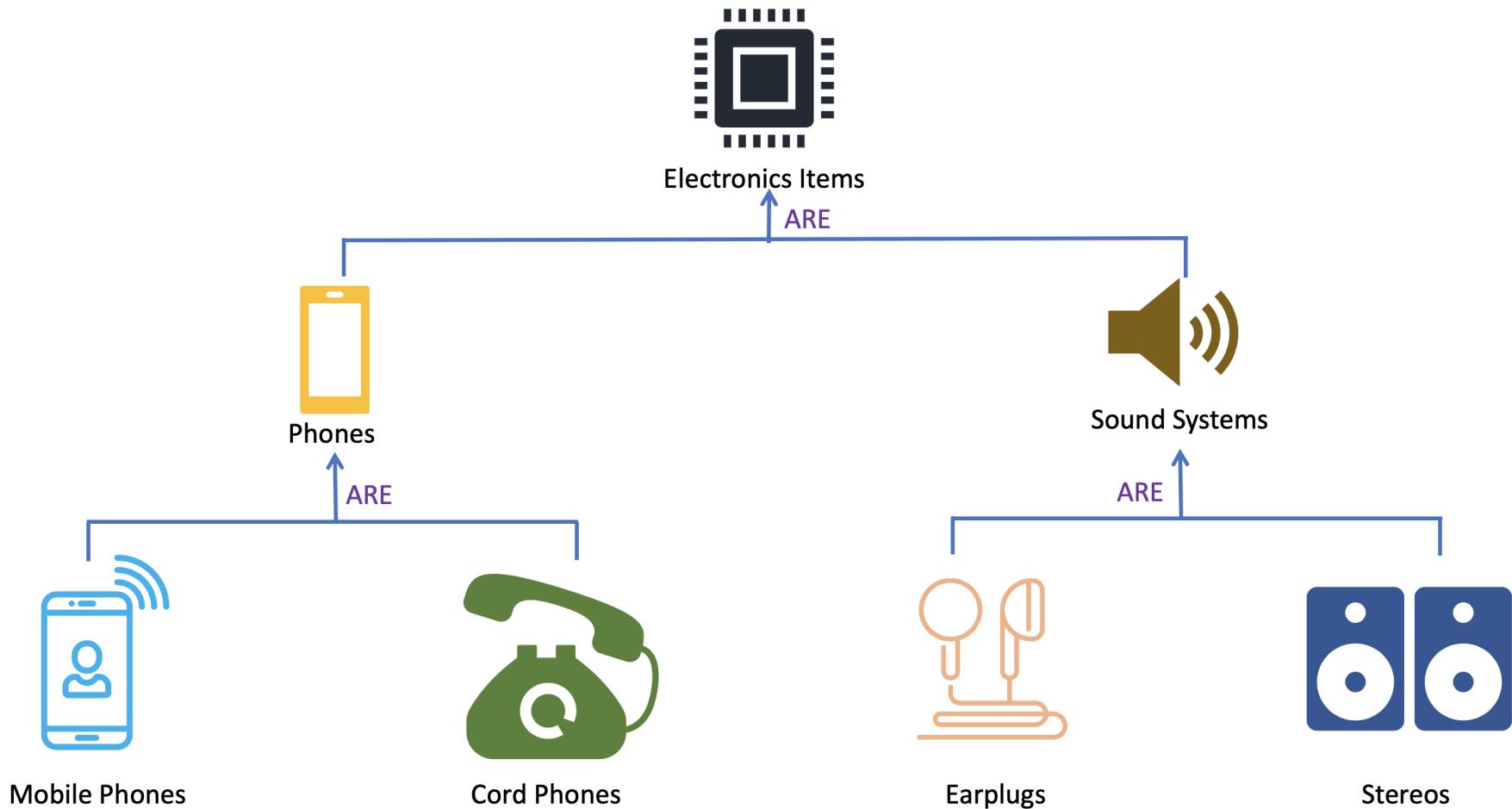


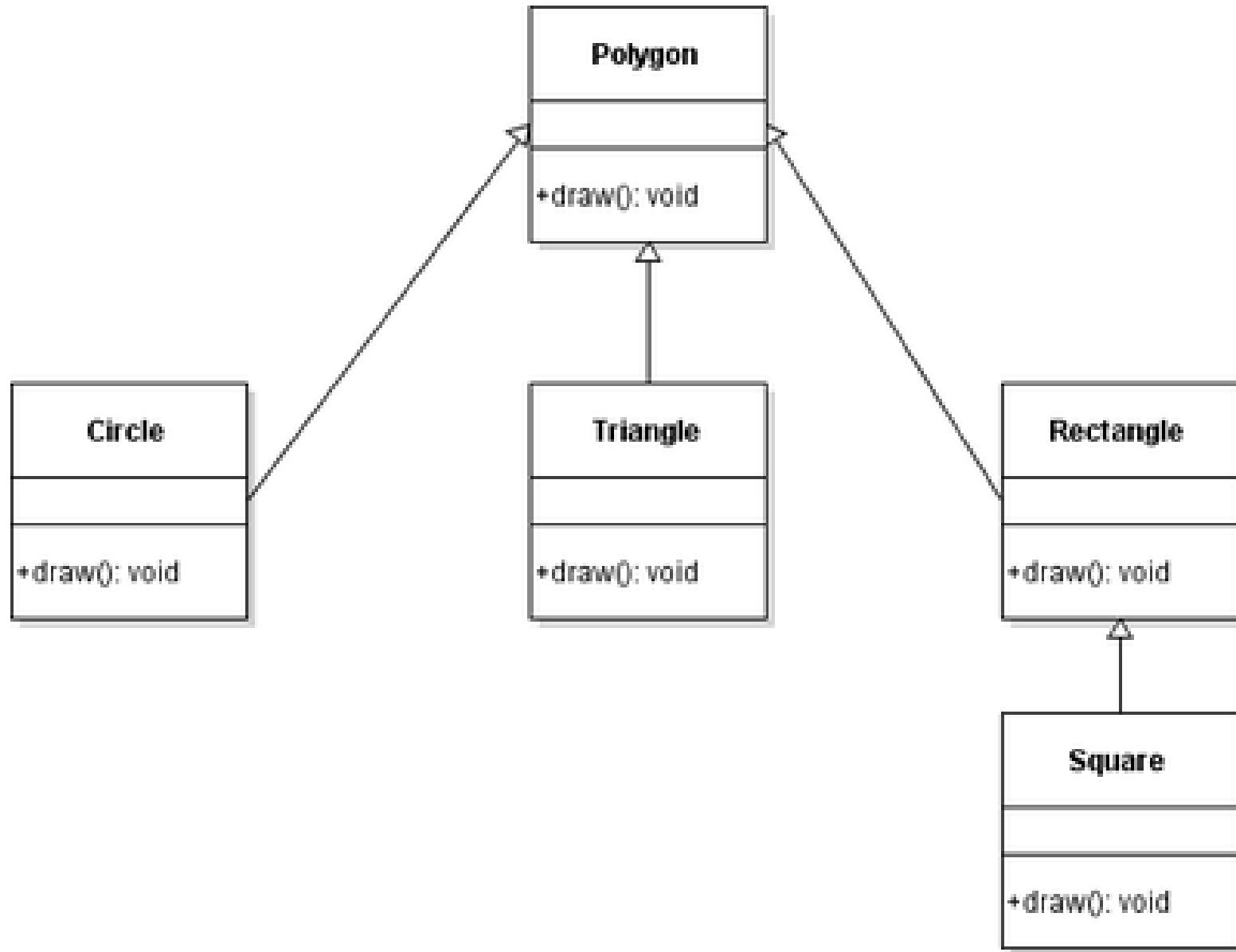
## ❖ Class :

- A class is a collection of method and variables
- It is a blueprint that defines the data and behavior of a type
- Let's take **HumanBeing** as a class
- A class is a blueprint for any functional entity which defines its properties and its functions
- Like **HumanBeing**, having body parts, performing various actions

## ❖ Inheritance :

- Inheritance is a feature of object-oriented programming that allows **code reusability** when a class includes property of another class
- Considering **HumanBeing** a class, which has properties like hands, legs, eyes, mouth, etc, and functions like walk, talk, eat, see, etc.
- Man and Woman are also classes, but most of the properties and functions are included in **HumanBeing**
- Hence, they can inherit everything from class **HumanBeing** using the concept of Inheritance





## ❖ Objects :

- My name is **Akhil**, and I am an instance/object of class **Man**

## ❖ Abstraction :

- Abstraction means, showcasing only the required things to the outside world while hiding the details
- Continuing our example, **HumanBeing's** can talk, walk, hear, eat, but the details of the muscles mechanism and their connections to the brain are hidden from the outside world

## ❖ **Encapsulation :**

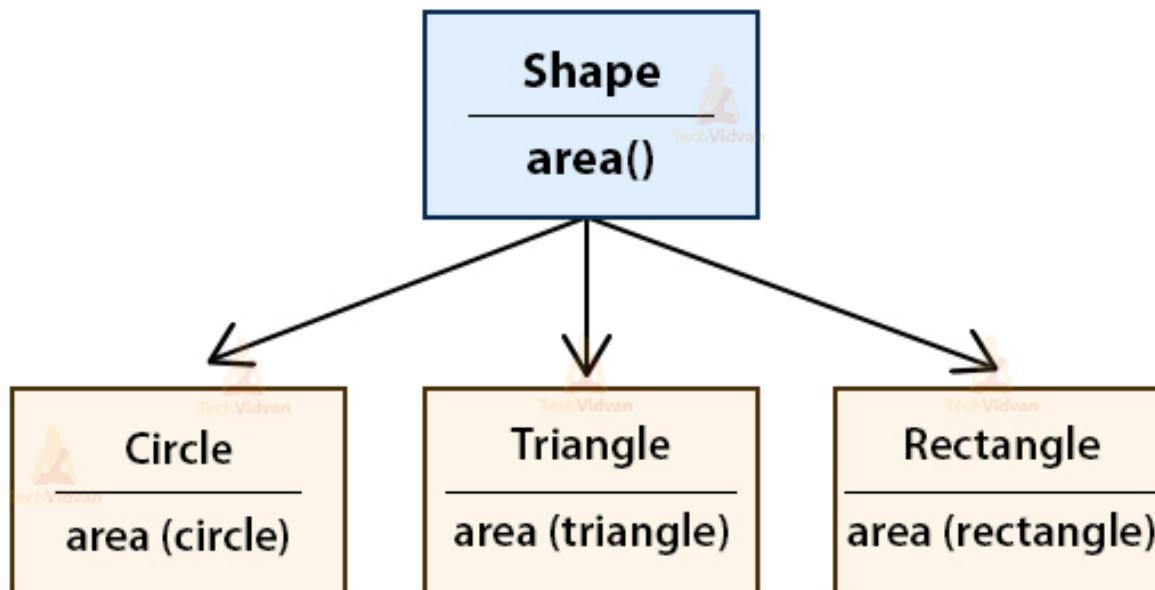
- Encapsulation means that we want to **hide unnecessary details from the user**
- For example, when we call from our mobile phone, we select the number and press call button
- But the entire process of calling or what happens from the moment we press or touch the call button to the moment we start having a phone conversation is hidden from us

<b>Abstraction</b>	<b>Encapsulation</b>
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the internal details or mechanics of how an object does something.
<b>4. Abstraction-</b> Outer layout, used in terms of design.  For Example:-  Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number.	<b>4. Encapsulation-</b> Inner layout, used in terms of implementation.  For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits.

## ❖ Polymorphism :

- Polymorphism is a feature of object-oriented programming languages that allows a specific routine to use variables of different types at different times

### Example of Polymorphism in Java



## ❖ Design classes

- A set of design classes refined the analysis class by **providing design details**
- There are **five different types of design classes** and each type represents the layer of the design architecture these are as follows:
  - 1) User interface classes :**
    - These classes are designed for **Human Computer Interaction(HCI)**
  - 2) Business domain classes :**
    - These classes are required to **implement the elements of the business domain**

### **3) Process classes :**

- Which is needed to completely manage the business domain class

### **4) Persistence classes :**

- It shows data stores that will persist behind the execution of the software

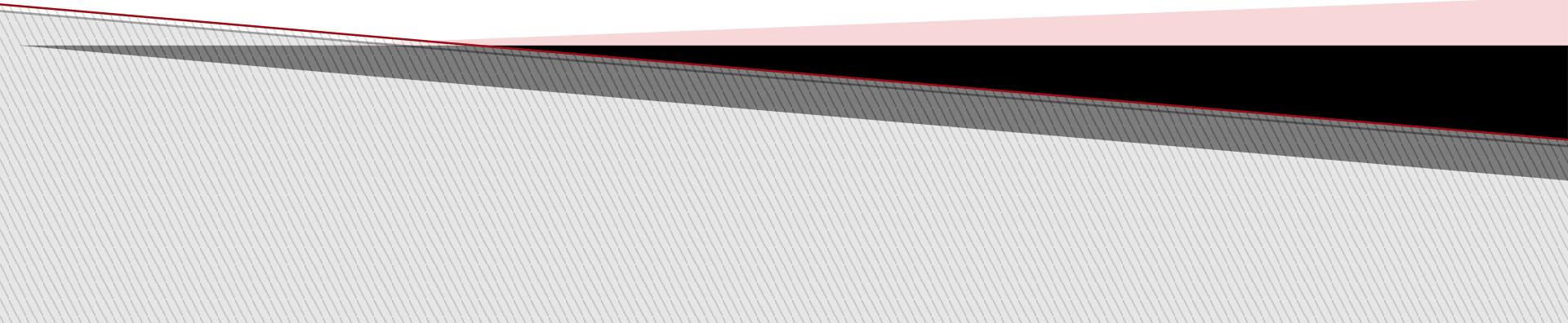
### **5) System Classes :**

- System classes implement software management and control functions

## ❖ Design class characteristics

- **Complete and sufficient**
- **Primitiveness** : Fulfill one service for the class
- **High cohesion** : A cohesion design class has a small and focused set of responsibilities
- **Low-coupling** : The minimum acceptable of collaboration must be kept in the model. If a **design model is highly coupled** then the **system is difficult to implement**, to test and to maintain over time

# **DESIGN PATTERNS**



# Index

- ❖ **Basic concepts of Design patterns**
- ❖ **How to select a design pattern**
- ❖ **Creational patterns**
- ❖ **Structural patterns**
- ❖ **Behavioral patterns**
- ❖ **Concept of Anti-patterns**

## ❖ Basic concepts of Design patterns

- ▶ In software engineering, a **design pattern** is a general repeatable solution to a commonly occurring problem in software design
- ▶ A design pattern isn't a finished design
- ▶ It is a description or template for how to solve a problem that can be used in many different situations

- ▶ Each design pattern systematically names, explains, and evaluates an important and recurring design in **object-oriented systems**
- ▶ Our goal is to capture design experience in a form that people can **use effectively**
- ▶ To this end we have documented some of the most important design patterns and present them as a **catalog**

- ▶ In general, a pattern has four essential elements:
  - 1) Pattern name**
  - 2) Problem**
  - 3) Solution**
  - 4) Consequences**
- **Pattern name**
  - Use to **describe a design problem, its solutions, and consequences** in a word or two
  - Naming a pattern immediately **increases our design vocabulary**
  - It makes it easier to think about designs and communicate to others
  - Finding good names has been one of **the hardest part**

## ❑ Problem

- Describes when to apply the pattern
- It explains the problem and its context

## ❑ Solution

- The solution doesn't describe a particular concrete design or implementation
- Because a pattern is like a template that can be applied in many different situations
- Describes the elements that make up the design, their relationships, responsibilities, and collaborations

## ❑ Consequences

- They are the results and trade-offs of applying the pattern

- ▶ Design patterns can **speed up the development process** by providing tested, proven development paradigms
- ▶ Reusing design patterns helps to **prevent subtle issues** that can cause major problems
- ▶ **Improves code readability** for coders and architects familiar with the patterns

- ▶ Often, people only understand how to apply certain software design techniques to certain problems
- ▶ These techniques are difficult to apply to a broader range of problems
- ▶ Design patterns provide general solutions, documented in a format that doesn't require specifics tied to a particular problem
- ▶ In addition, patterns allow developers to communicate using well-known, well understood names for software interactions
- ▶ Common design patterns can be improved over time

## ❖ How to select a design pattern

- ▶ With more than 20 design patterns in the catalog to choose from, it might be **hard to find the one** that addresses a particular design problem, especially if the catalog is new and unfamiliar to you
- ▶ Here are several different approaches to finding the design pattern that's **right for your problem**:

- ❑ **Consider how design patterns solve design problems**
  - Find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems
- ❑ **Scan Intent sections**
  - Read through each pattern's intent to find one or more that sound relevant to your problem
- ❑ **Study how patterns interrelate**
  - Studying these **relationships between design patterns graphically** can help direct you to the right pattern or group of patterns

## ❑ Study patterns of like purpose

- Study the similarities and differences between **creational patterns**, **structural patterns** and **behavioral patterns**

## ❑ Examine a cause of redesign

- Look at the patterns that help you avoid the causes of redesign
- Eg : **Algorithmic dependencies**
  - Algorithms are often extended, optimized, and replaced during development and reuse
  - Objects that depend on an algorithm will have to change when the algorithm changes
  - Therefore algorithms that are likely to change should be isolated
  - Design patterns: Builder, Iterator, Strategy, Template, Method, Visitor

## ❑ Consider what should be variable in your design

- Consider what you want to be able to change without redesign
- The focus here is on encapsulating the concept that varies
- Design aspect(s) (eg : steps of an algorithm) of design patterns can vary independently, thereby letting you change them without redesign

# ❖ Organizing the Catalog

- ▶ There are many design patterns, we need a way to organize them

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

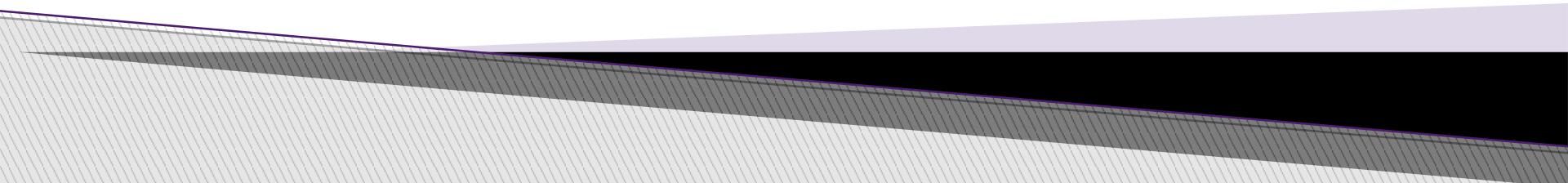
Table 1.1: Design pattern space

- ▶ We classify design patterns by two criteria (Table 1.1)
- ▶ The first criterion, called **purpose**, reflects what a pattern does
- ▶ Patterns can have either **creational, structural, or behavioral** purpose
- ▶ **Creational patterns** concern the process of object creation
- ▶ **Structural patterns** deal with the composition of classes or objects
- ▶ **Behavioral patterns** characterize the ways in which classes or objects interact and distribute responsibility

- ▶ The second criterion, called **scope**, specifies whether the pattern applies primarily **to classes or to objects**
- ▶ Class patterns deal with relationships between classes and their **subclasses**
- ▶ These relationships are established through inheritance, so they are **static (fixed at compile-time)**
- ▶ Object patterns deal with **object relationships**, which can be changed **at run-time and are more dynamic**
- ▶ Only patterns labeled "class patterns" are those that focus on class relationships
- ▶ Note that most patterns are in the Object scope

- ▶ Creational class patterns defer some part of object creation to subclasses, while Creational object patterns defer it to another object
- ▶ The Structural class patterns use inheritance to compose classes, while the Structural object patterns describe ways to assemble objects
- ▶ The Behavioral class patterns use inheritance to describe algorithms and flow of control, whereas the Behavioral object patterns describe how a group of objects cooperate to perform a task that no single object can carry out alone

# Creational Patterns



- ▶ Creational design patterns **abstract the instantiation process**
- ▶ They help make a **system independent of how its objects are created, composed, and represented**
- ▶ This pattern can be further divided into **class-creation patterns** and **object-creational patterns**
- ▶ While class-creation patterns use **inheritance** effectively in the instantiation process
- ▶ Object-creation patterns use delegation effectively to get the job done

<b>Creational</b>	
<b>Class</b>	Factory Method
<b>Object</b>	Abstract Factory Builder Prototype Singleton

## ❖ **Abstract Factory**

- ▶ **Object Creational**

### **Intent**

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

### **Also Known As**

Kit

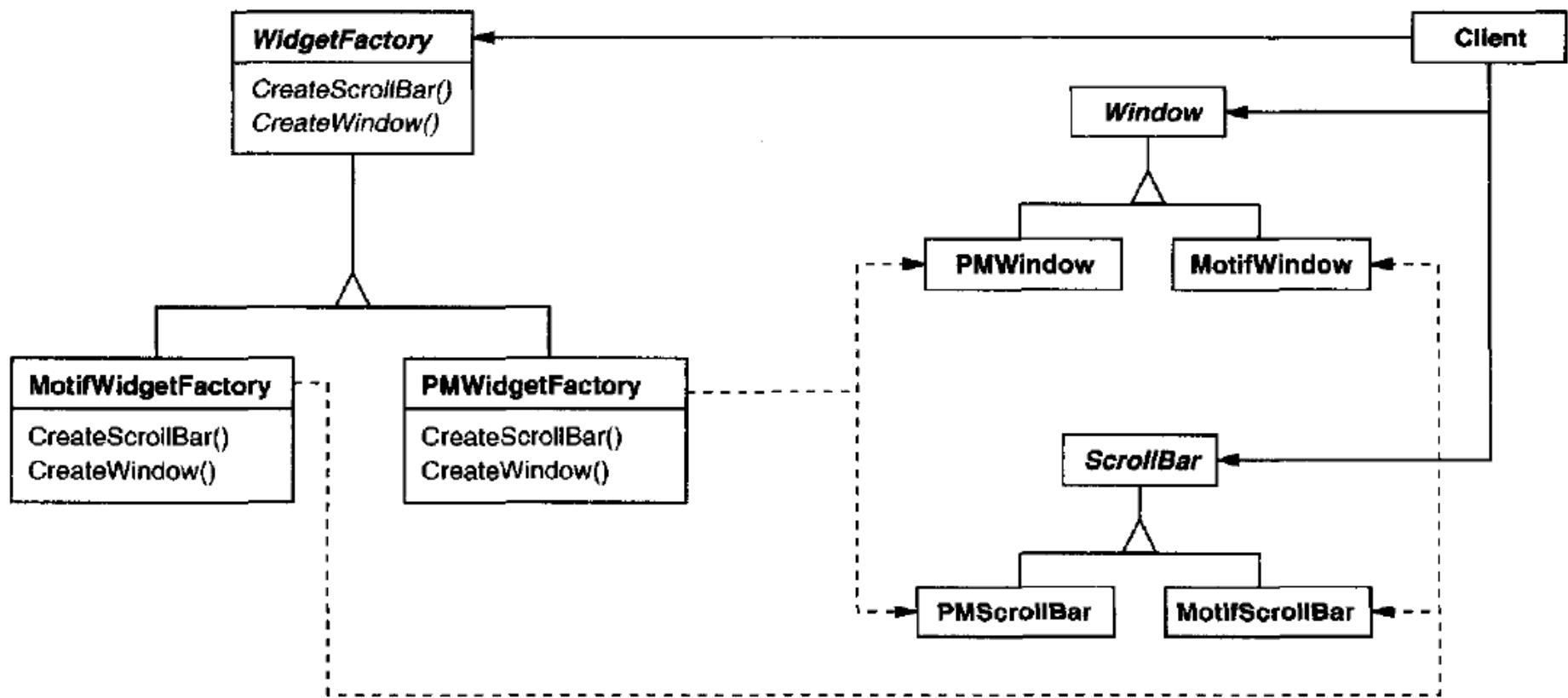
# Motivation

Consider a user interface toolkit that supports multiple look-and-feel standards, such as Motif and Presentation Manager. Different look-and-feels define different appearances and behaviors for user interface “widgets” like scroll bars, windows, and buttons. To be portable across look-and-feel standards, an application should not hard-code its widgets for a particular look and feel. Instantiating look-and-feel-specific classes of widgets throughout the application makes it hard to change the look and feel later.

We can solve this problem by defining an abstract `WidgetFactory` class that declares an interface for creating each basic kind of widget. There's also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards. `WidgetFactory`'s interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they're using. Thus clients stay independent of the prevailing look and feel.

There is a concrete subclass of WidgetFactory for each look-and-feel standard. Each subclass implements the operations to create the appropriate widget for the look and feel. For example, the CreateScrollBar operation on the MotifWidgetFactory instantiates and returns a Motif scroll bar, while the corresponding operation on the PMWidgetFactory returns a scroll bar for Presentation Manager. Clients create widgets solely through the WidgetFactory interface and have no knowledge of the classes that implement widgets for a particular look and feel. In other words, clients only have to commit to an interface defined by an abstract class, not a particular concrete class.

A WidgetFactory also enforces dependencies between the concrete widget classes. A Motif scroll bar should be used with a Motif button and a Motif text editor, and that constraint is enforced automatically as a consequence of using a MotifWidgetFactory.

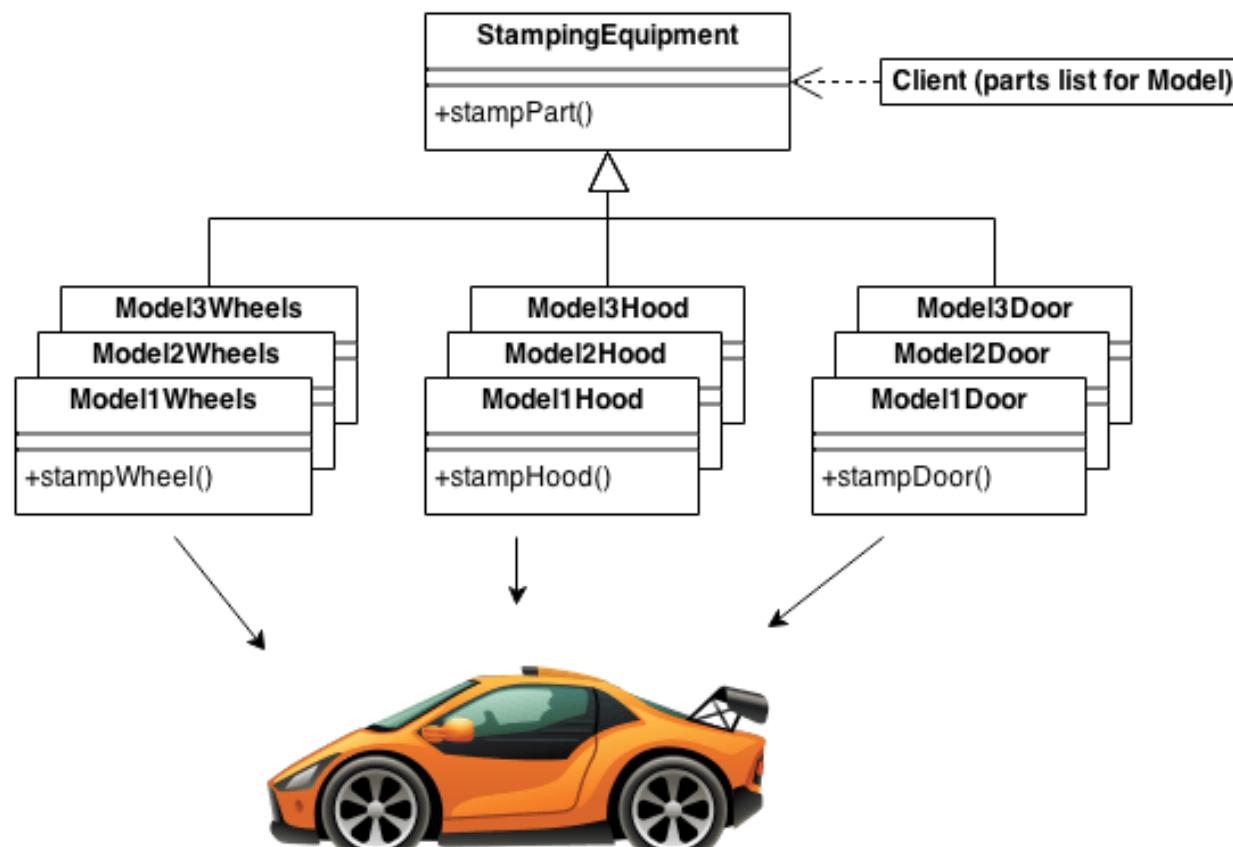


# Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- **AbstractProduct** (Window, ScrollBar)
  - declares an interface for a type of product object.
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

## Example

The purpose of the Abstract Factory is to provide an interface for creating families of related objects, without specifying concrete classes. This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles. The stamping equipment is an Abstract Factory which creates auto body parts. The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars. Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.



## ❖ **Builder**

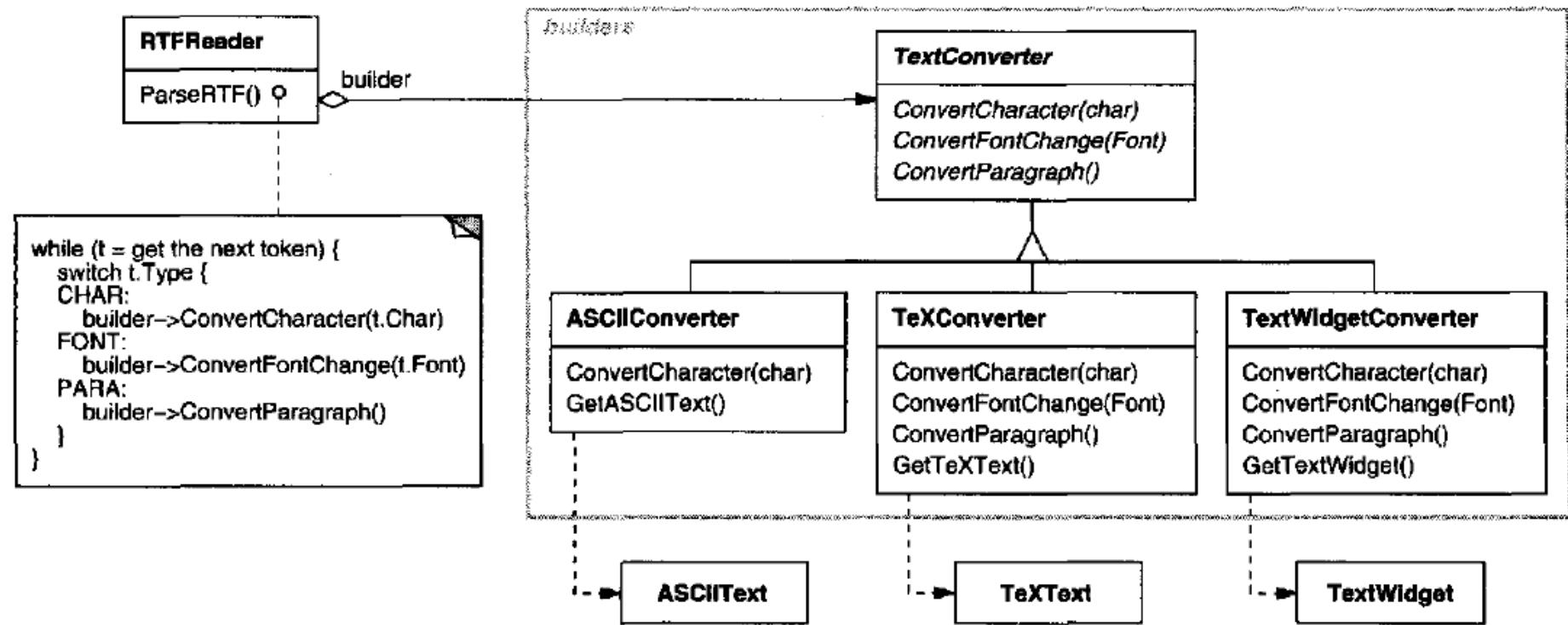
- ▶ **Object Creational**

### **Intent**

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats. The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

A solution is to configure the RTFReader class with a TextConverter object that converts RTF to another textual representation. As the RTFReader parses the RTF document, it uses the TextConverter to perform the conversion. Whenever the RTFReader recognizes an RTF token (either plain text or an RTF control word), it issues a request to the TextConverter to convert the token. TextConverter objects are responsible both for performing the data conversion and for representing the token in a particular format.

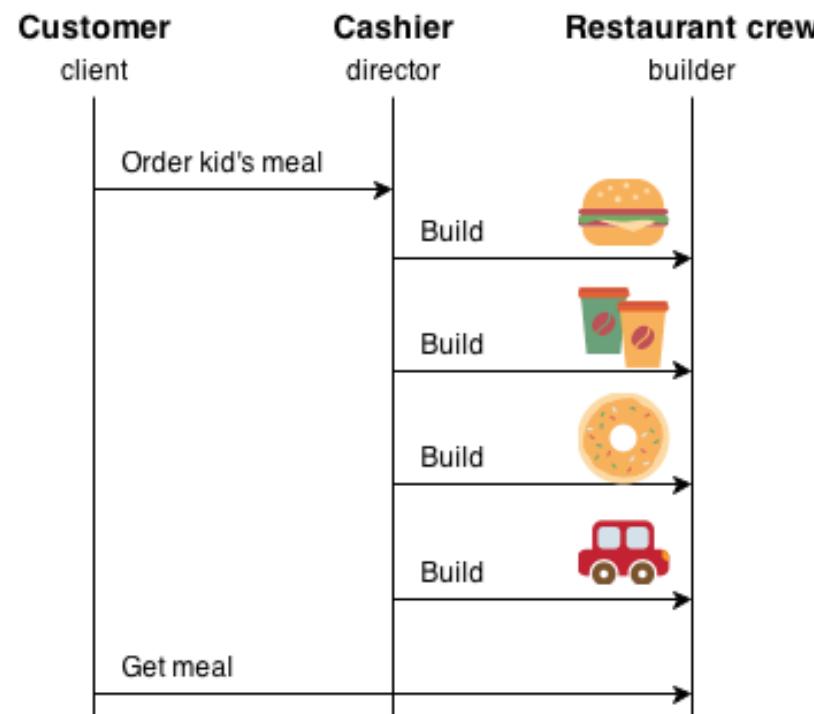


# Participants

- **Builder** (`TextConverter`)
  - specifies an abstract interface for creating parts of a **Product** object.
- **ConcreteBuilder** (`ASCIIConverter`, `TeXConverter`, `TextWidgetConverter`)
  - constructs and assembles parts of the product by implementing the **Builder** interface.
  - defines and keeps track of the representation it creates.
  - provides an interface for retrieving the product (e.g., `GetASCIIText`, `GetTextWidget`).
- **Director** (`RTFReader`)
  - constructs an object using the **Builder** interface.
- **Product** (`ASCIIText`, `TeXText`, `TextWidget`)
  - represents the complex object under construction. **ConcreteBuilder** builds the product's internal representation and defines the process by which it's assembled.

# Example

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy dinosaur). Note that there can be variation in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.



# ❖ Factory Method

- ▶ Class Creational

## Intent

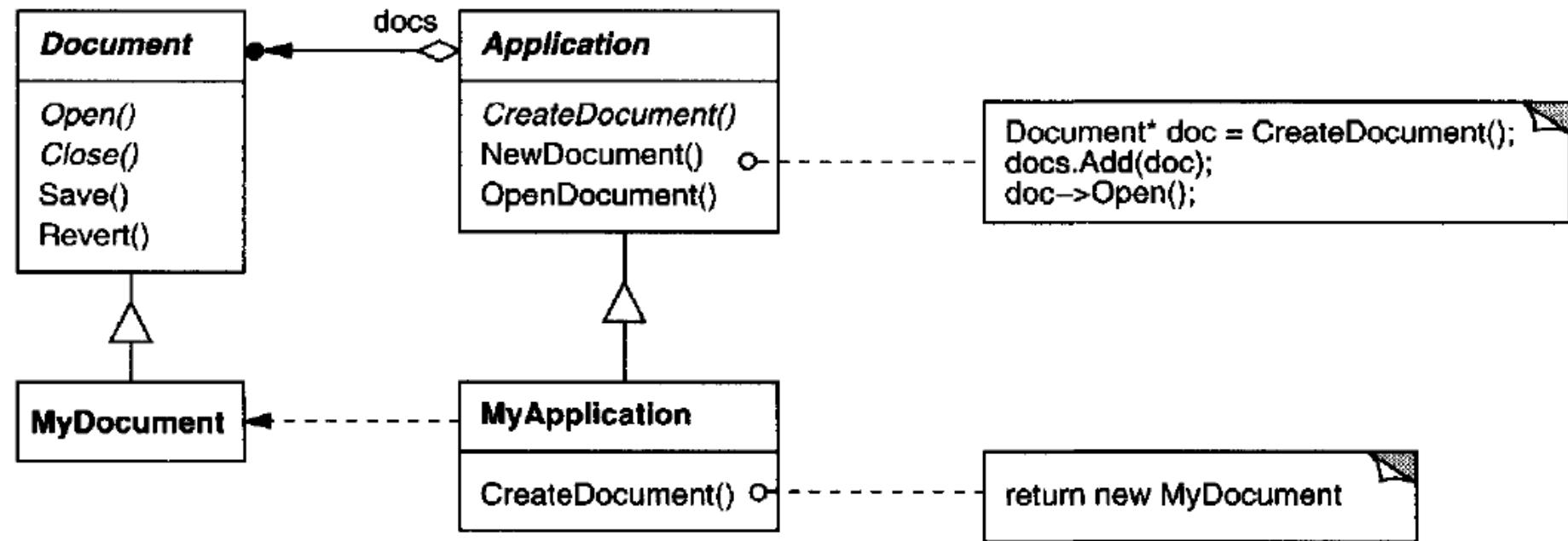
Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

## Also Known As

Virtual Constructor

Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well.

Application subclasses redefine an abstract `CreateDocument` operation on Application to return the appropriate Document subclass. Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class. We call `CreateDocument` a **factory method** because it's responsible for "manufacturing" an object.

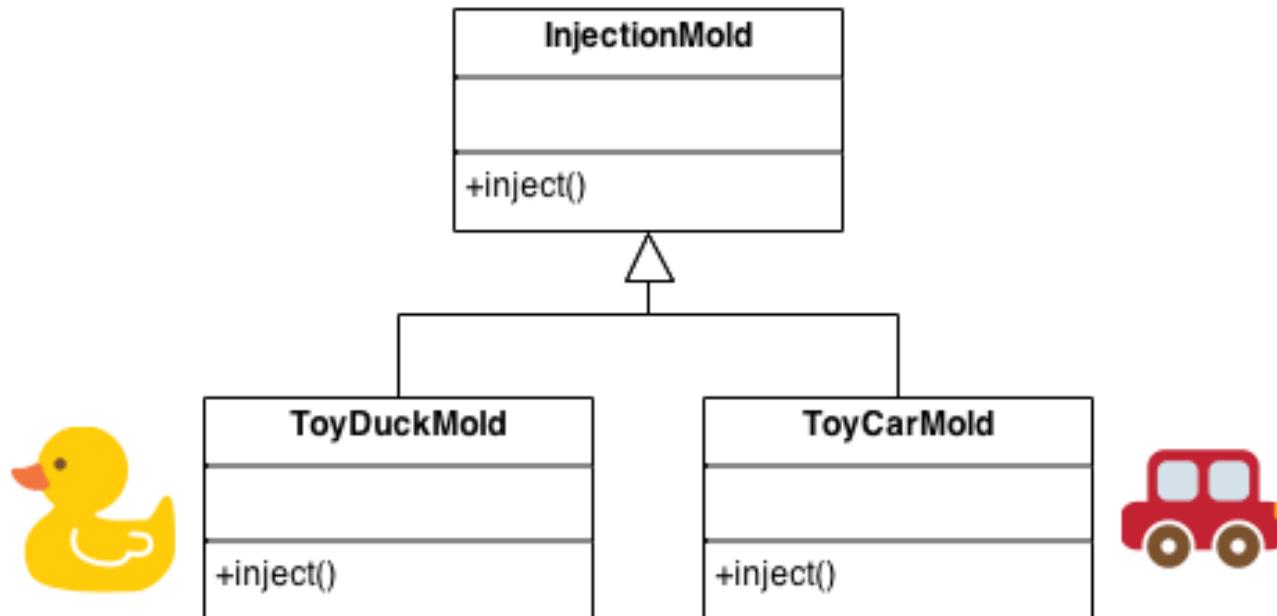


# Participants

- **Product** (Document)
  - defines the interface of objects the factory method creates.
- **ConcreteProduct** (MyDocument)
  - implements the Product interface.
- **Creator** (Application)
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.
- **ConcreteCreator** (MyApplication)
  - overrides the factory method to return an instance of a ConcreteProduct.

# Example

The Factory Method defines an interface for creating objects, but lets subclasses decide which classes to instantiate. Injection molding presses demonstrate this pattern. Manufacturers of plastic toys process plastic molding powder, and inject the plastic into molds of the desired shapes. The class of toy (car, action figure, etc.) is determined by the mold.



## ❖ Prototype

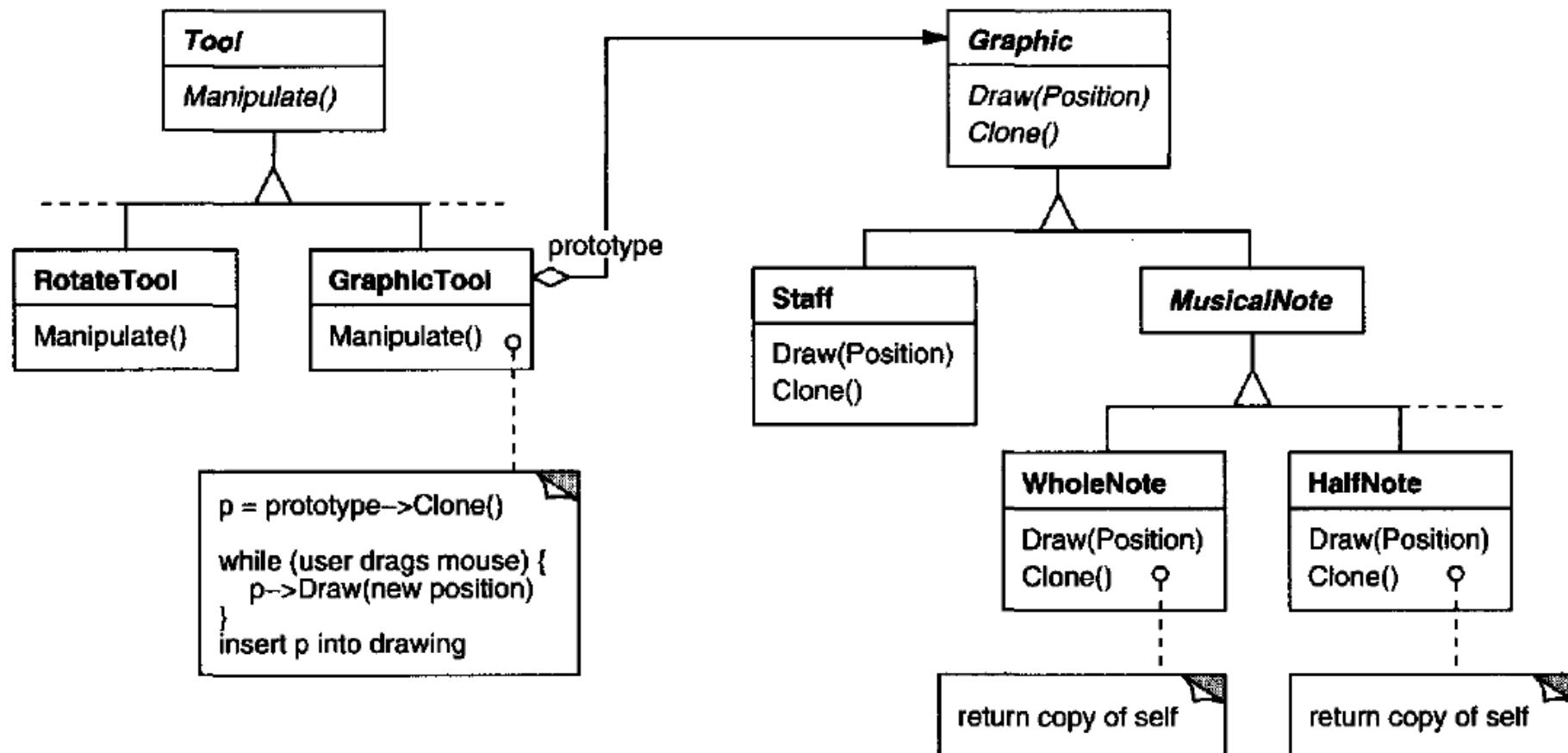
- ▶ Object Creational

### Intent

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves. The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music objects. Users will click on the quarter-note tool and use it to add quarter notes to the score. Or they can use the move tool to move a note up or down on the staff, thereby changing its pitch.

So in our music editor, each tool for creating a music object is an instance of `GraphicTool` that's initialized with a different prototype. Each `GraphicTool` instance will produce a music object by cloning its prototype and adding the clone to the score.



# Participants

- **Prototype** (Graphic)
  - declares an interface for cloning itself.
- **ConcretePrototype** (Staff, WholeNote, HalfNote)
  - implements an operation for cloning itself.
- **Client** (GraphicTool)
  - creates a new object by asking a prototype to clone itself.

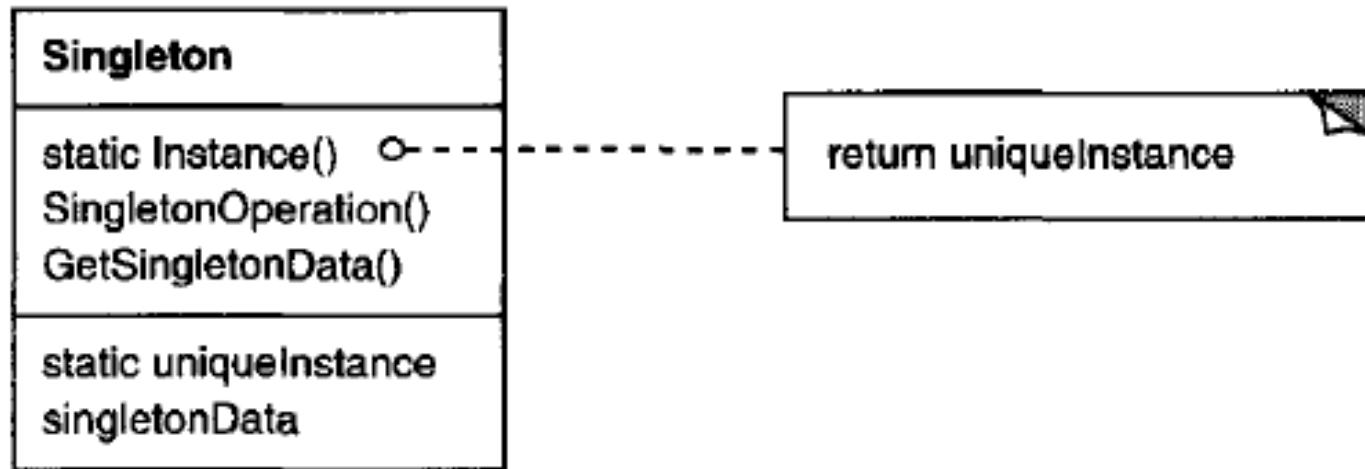
## ❖ Singleton

- ▶ Object Creational

### Intent

Ensure a class only has one instance, and provide a global point of access to it.

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

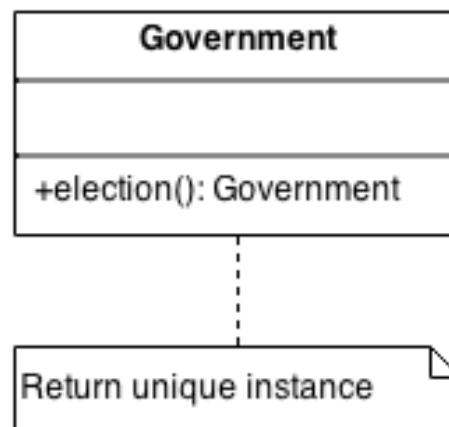


## Participants

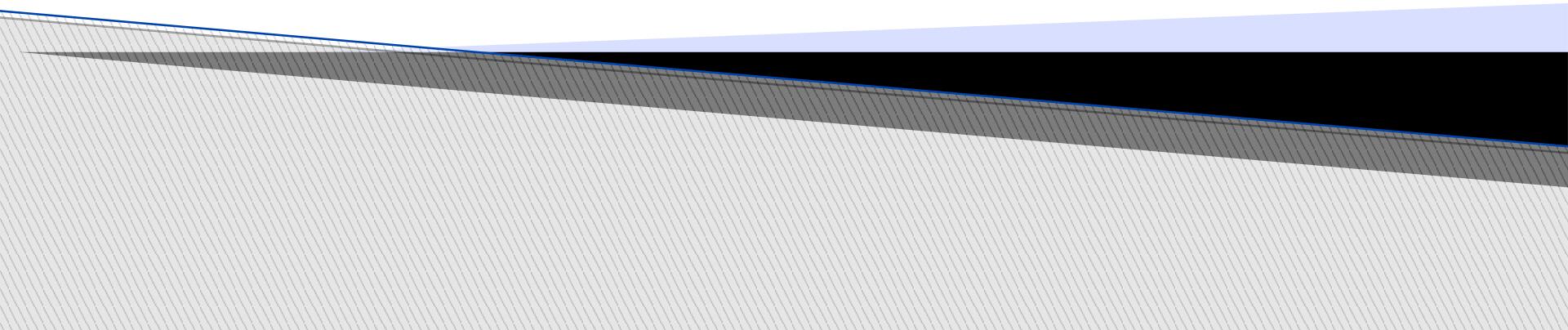
- **Singleton**
  - defines an **Instance** operation that lets clients access its unique instance. **Instance** is a class operation (that is, a class method in Smalltalk and a static member function in C++).
  - may be responsible for creating its own unique instance.

# Example

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. It is named after the singleton set, which is defined to be a set containing one element. The office of the President of the United States is a Singleton. The United States Constitution specifies the means by which a president is elected, limits the term of office, and defines the order of succession. As a result, there can be at most one active president at any given time. Regardless of the personal identity of the active president, the title, "The President of the United States" is a global point of access that identifies the person in the office.



# Structural Patterns



- ▶ These design patterns are all about Class and Object composition
- ▶ Structural class patterns use inheritance to compose interfaces
- ▶ Structural object-patterns define ways to compose objects to obtain new functionality

Structural	
Class	Adapter (class)
Object	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy

# ❖ Adapter

- ▶ Class, Object Structural

## 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.

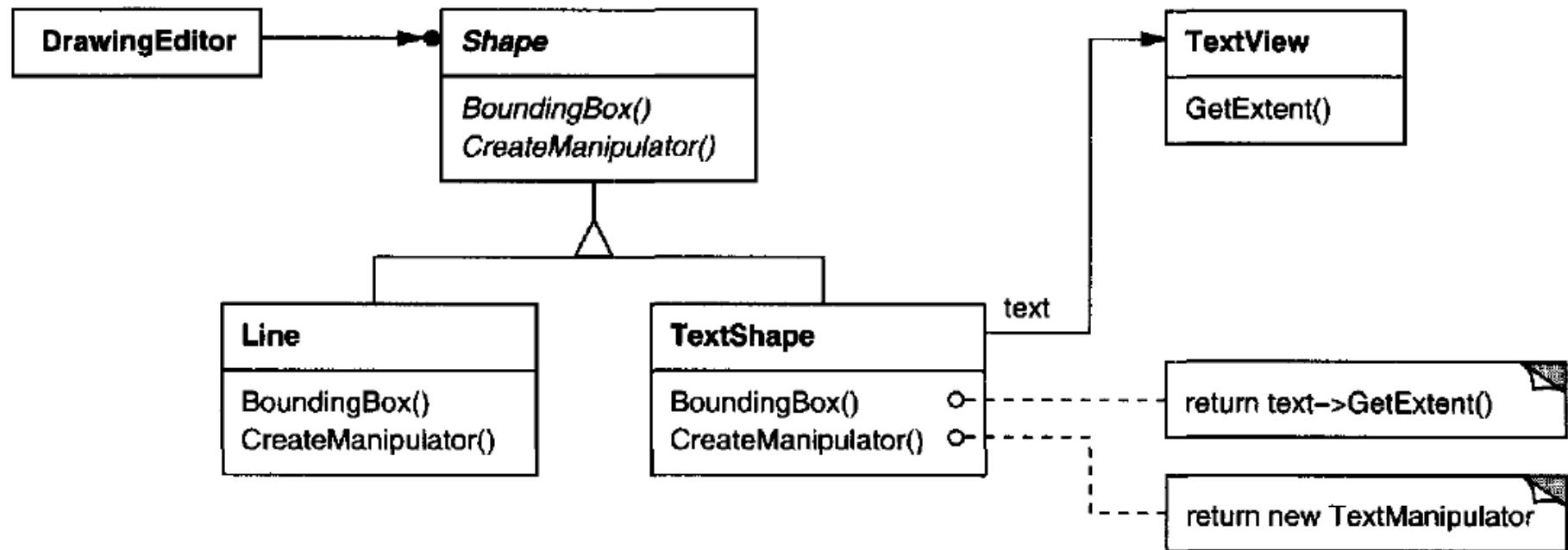
## Also Known As

Wrapper

Consider for example 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. The interface for graphical objects is defined by an abstract class called Shape. The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.

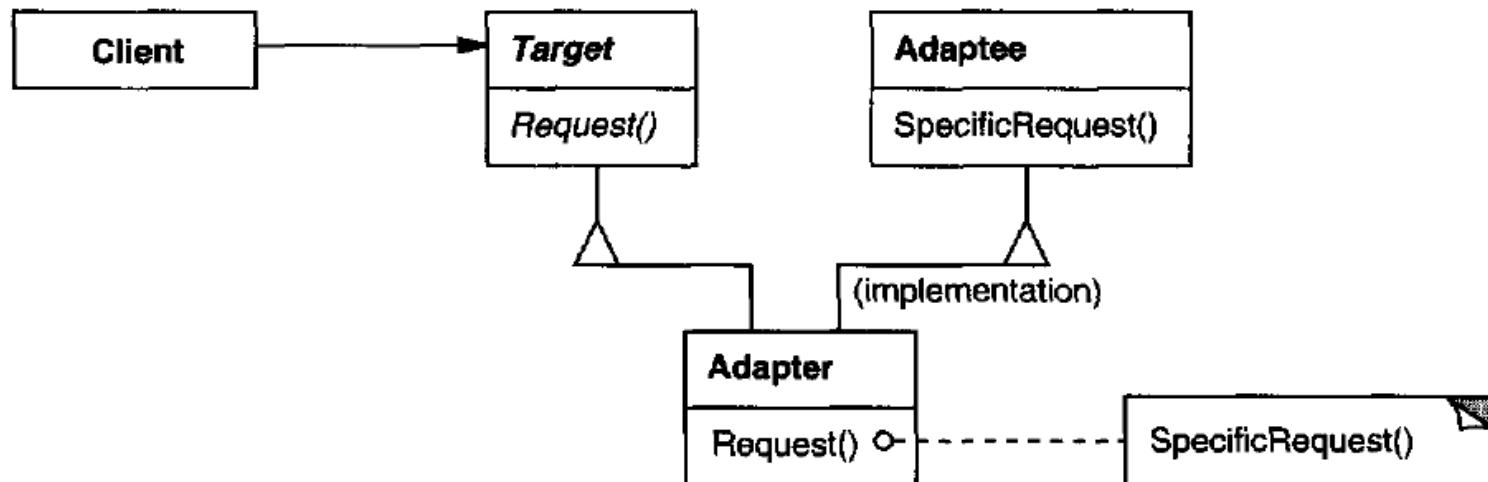
Classes for elementary geometric shapes like LineShape and PolygonShape are rather easy to implement, because their drawing and editing capabilities are inherently limited. But a TextShape subclass that can display and edit text is considerably more difficult to implement, since even basic text editing involves complicated screen update and buffer management.

Instead, we could define `TextShape` so that it *adapts* the `TextView` interface to `Shape`'s. We can do this in one of two ways: (1) by inheriting `Shape`'s interface and `TextView`'s implementation or (2) 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**.

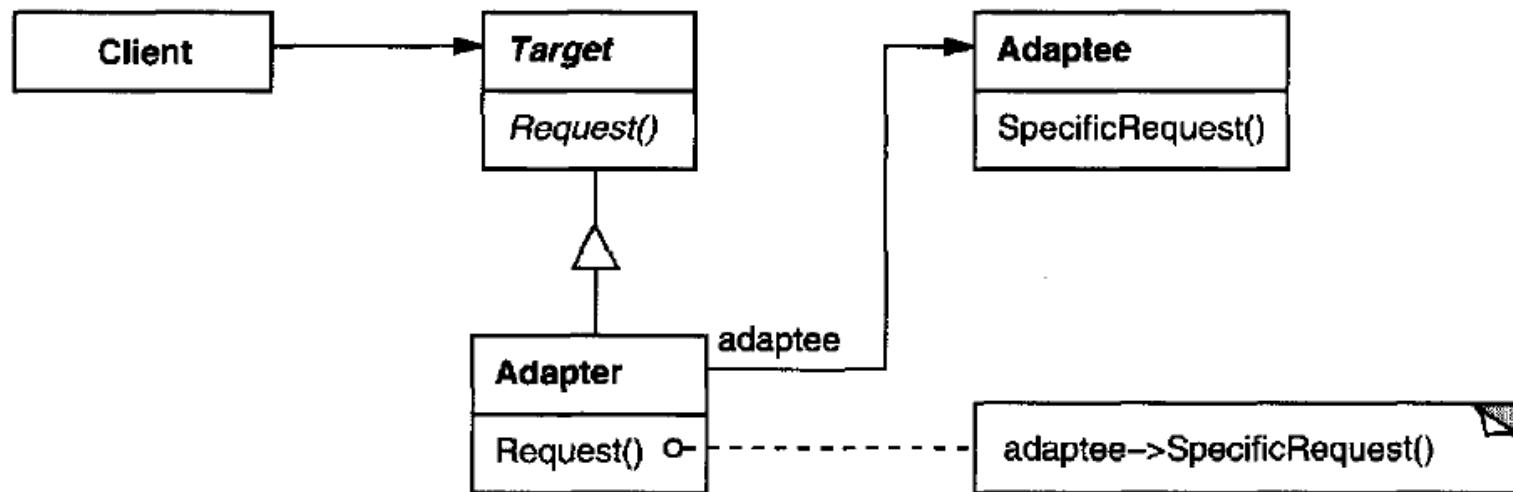


This diagram illustrates the object adapter case. It shows how `BoundingBox` requests, declared in class `Shape`, are converted to `GetExtent` requests defined in `TextView`. Since `TextShape` adapts `TextView` to the `Shape` interface, the drawing editor can reuse the otherwise incompatible `TextView`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

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



An object adapter relies on object composition:

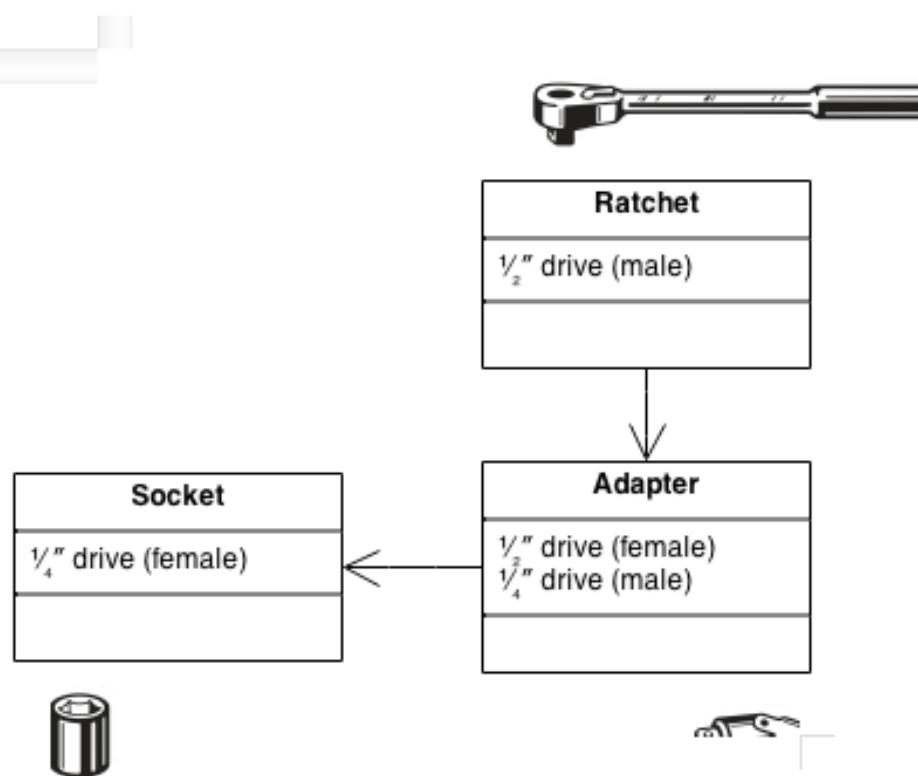


# Participants

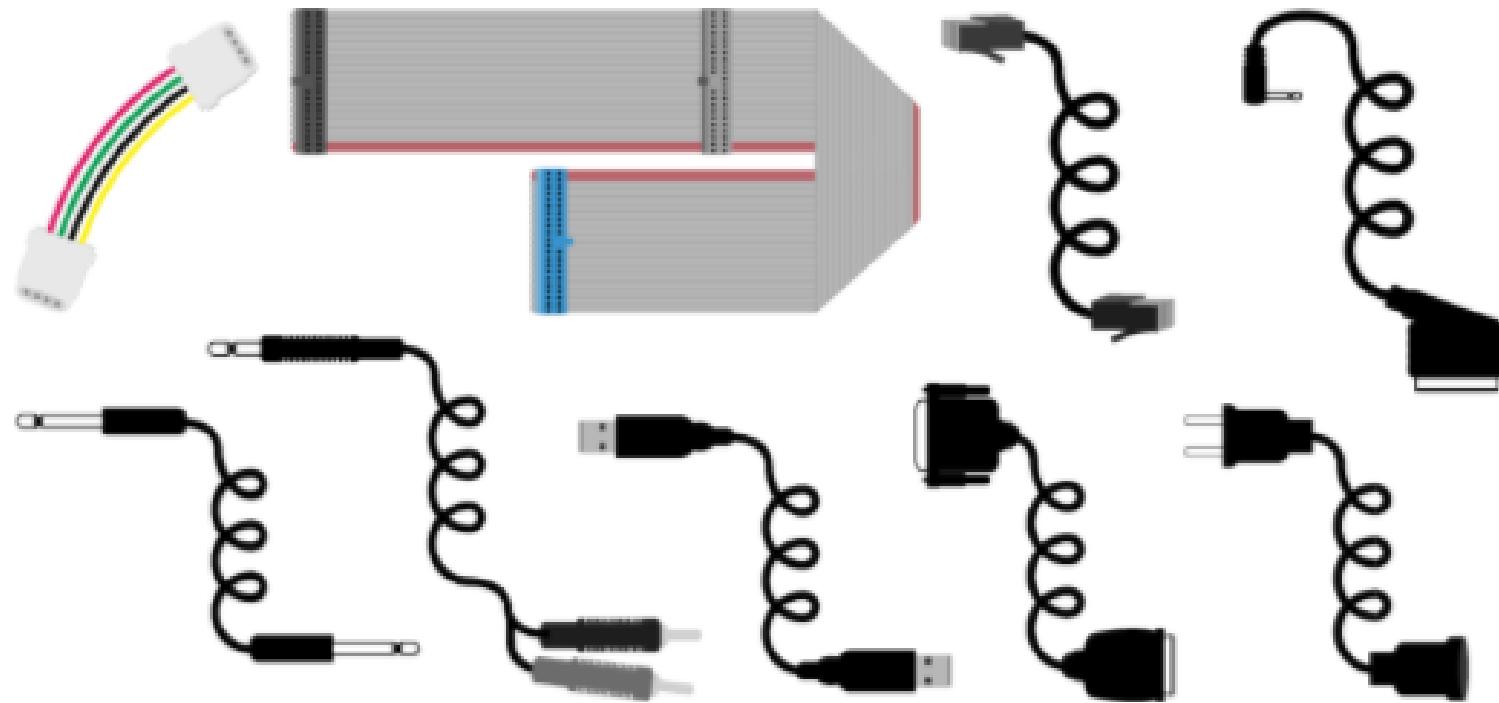
- **Target** (Shape)
  - defines the domain-specific interface that Client uses.
- **Client** (DrawingEditor)
  - collaborates with objects conforming to the Target interface.
- **Adaptee** (TextView)
  - defines an existing interface that needs adapting.
- **Adapter** (TextShape)
  - adapts the interface of Adaptee to the Target interface.

# Example

The Adapter pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. Socket wrenches provide an example of the Adapter. A socket attaches to a ratchet, provided that the size of the drive is the same. Typical drive sizes in the United States are 1/2" and 1/4". Obviously, a 1/2" drive ratchet will not fit into a 1/4" drive socket unless an adapter is used. A 1/2" to 1/4" adapter has a 1/2" female connection to fit on the 1/2" drive ratchet, and a 1/4" male connection to fit in the 1/4" drive socket.



It is like the problem of inserting a new three-prong electrical plug in an old two-prong wall outlet – some kind of adapter or intermediary is necessary.



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

## ❖ Bridge

- ▶ Object Structural

### Intent

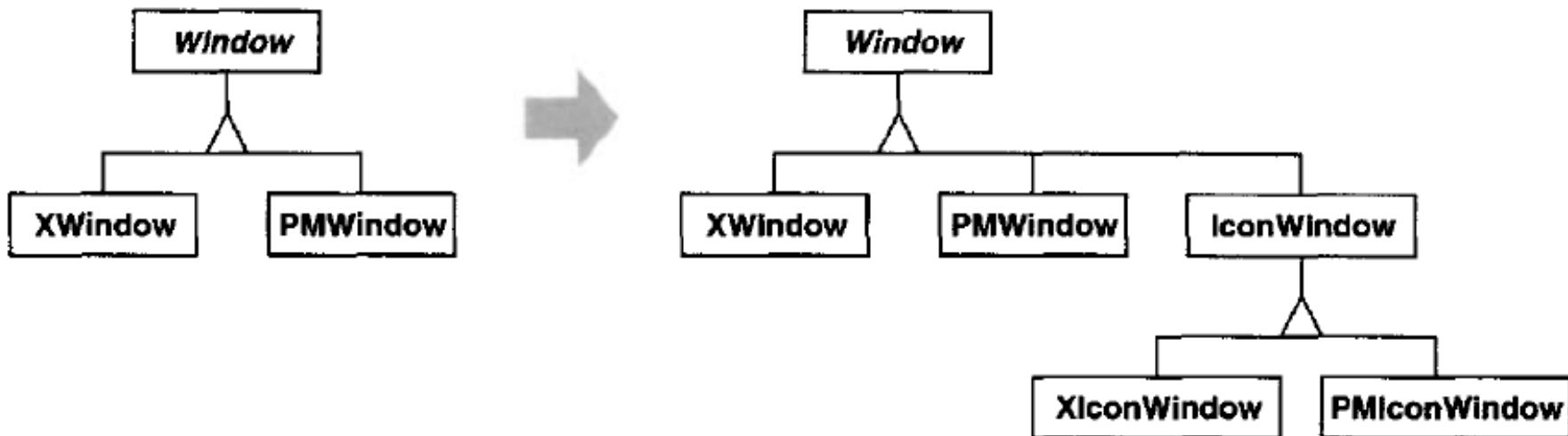
Decouple an abstraction from its implementation so that the two can vary independently.

### Also Known As

Handle/Body

Consider the implementation of a portable Window abstraction in a user interface toolkit. This abstraction should enable us to write applications that work on both the X Window System and IBM's Presentation Manager (PM), for example. Using inheritance, we could define an abstract class `Window` and subclasses `XWindow` and `PMWindow` that implement the `Window` interface for the different platforms. But this approach has two drawbacks:

1. It's inconvenient to extend the Window abstraction to cover different kinds of windows or new platforms. Imagine an IconWindow subclass of Window that specializes the Window abstraction for icons. To support IconWindows for both platforms, we have to implement *two* new classes, XIconWindow and PMIconWindow. Worse, we'll have to define two classes for *every* kind of window. Supporting a third platform requires yet another new Window subclass for every kind of window.

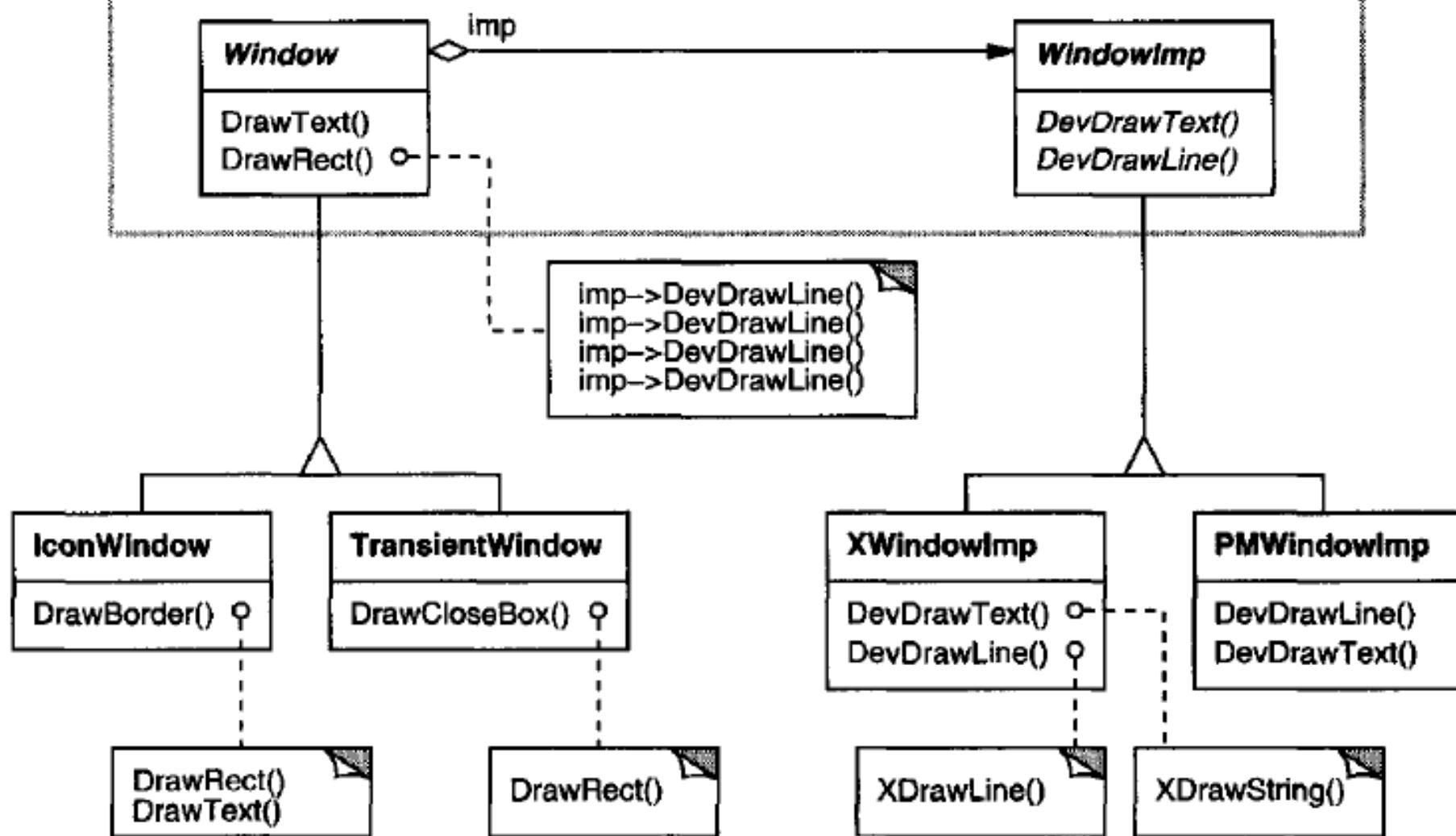


2. It makes client code platform-dependent. Whenever a client creates a window, it instantiates a concrete class that has a specific implementation. For example, creating an XWindow object binds the Window abstraction to the X Window implementation, which makes the client code dependent on the X Window implementation. This, in turn, makes it harder to port the client code to other platforms.

Clients should be able to create a window without committing to a concrete implementation. Only the window implementation should depend on the platform on which the application runs. Therefore client code should instantiate windows without mentioning specific platforms.

The Bridge pattern addresses these problems by putting the Window abstraction and its implementation in separate class hierarchies. There is one class hierarchy for window interfaces (Window, IconWindow, TransientWindow) and a separate hierarchy for platform-specific window implementations, with WindowImp as its root. The XWindowImp subclass, for example, provides an implementation based on the X Window System.

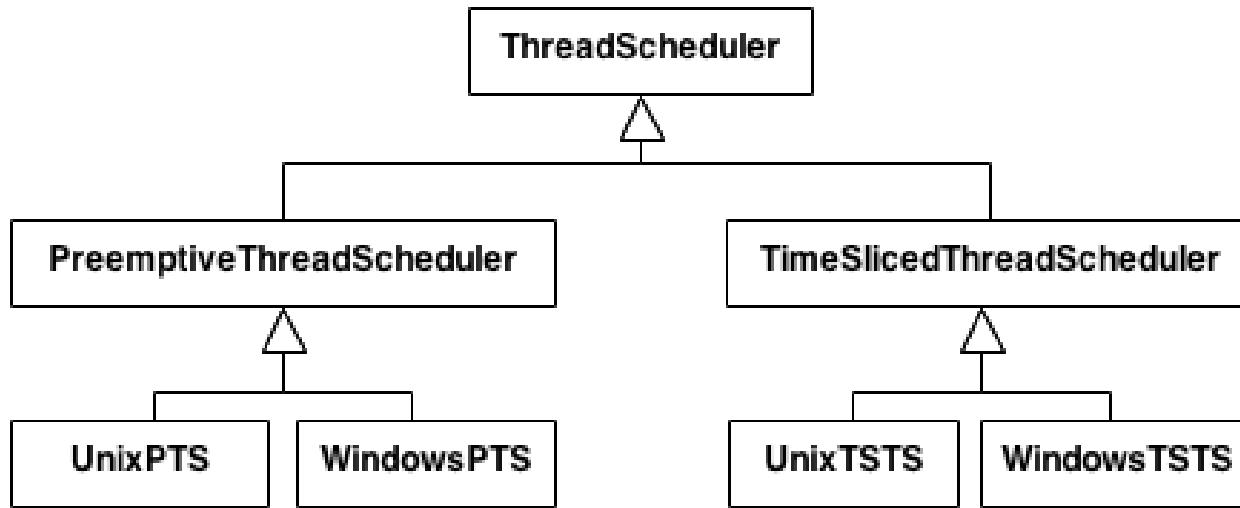
*bridge*



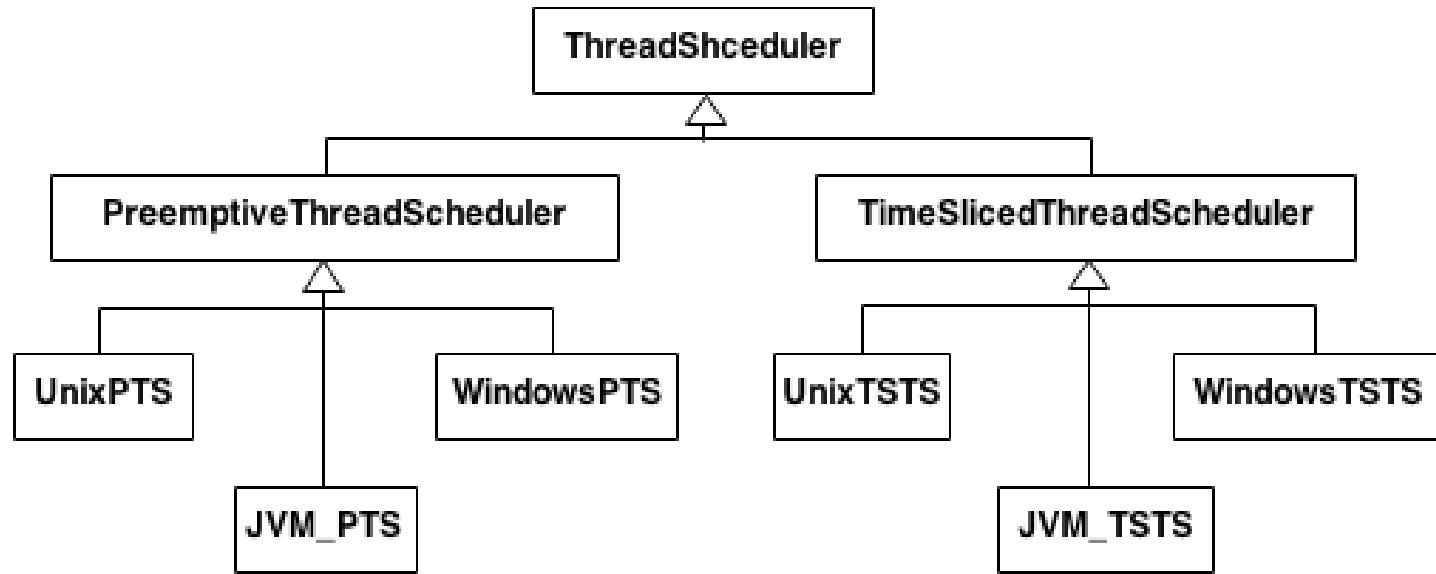
# Participants

- **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.

Consider the domain of "thread scheduling".

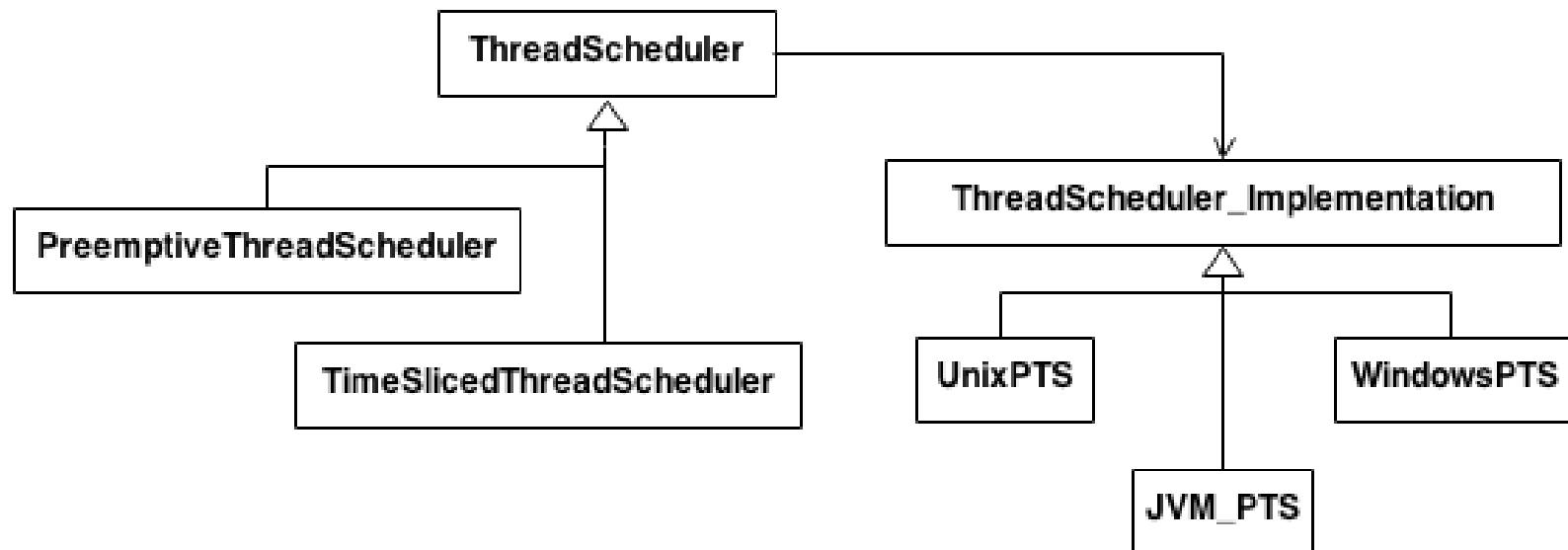


There are two types of thread schedulers, and two types of operating systems or "platforms". Given this approach to specialization, we have to define a class for each permutation of these two dimensions. If we add a new platform (say ...Java's Virtual Machine), what would our hierarchy look like?



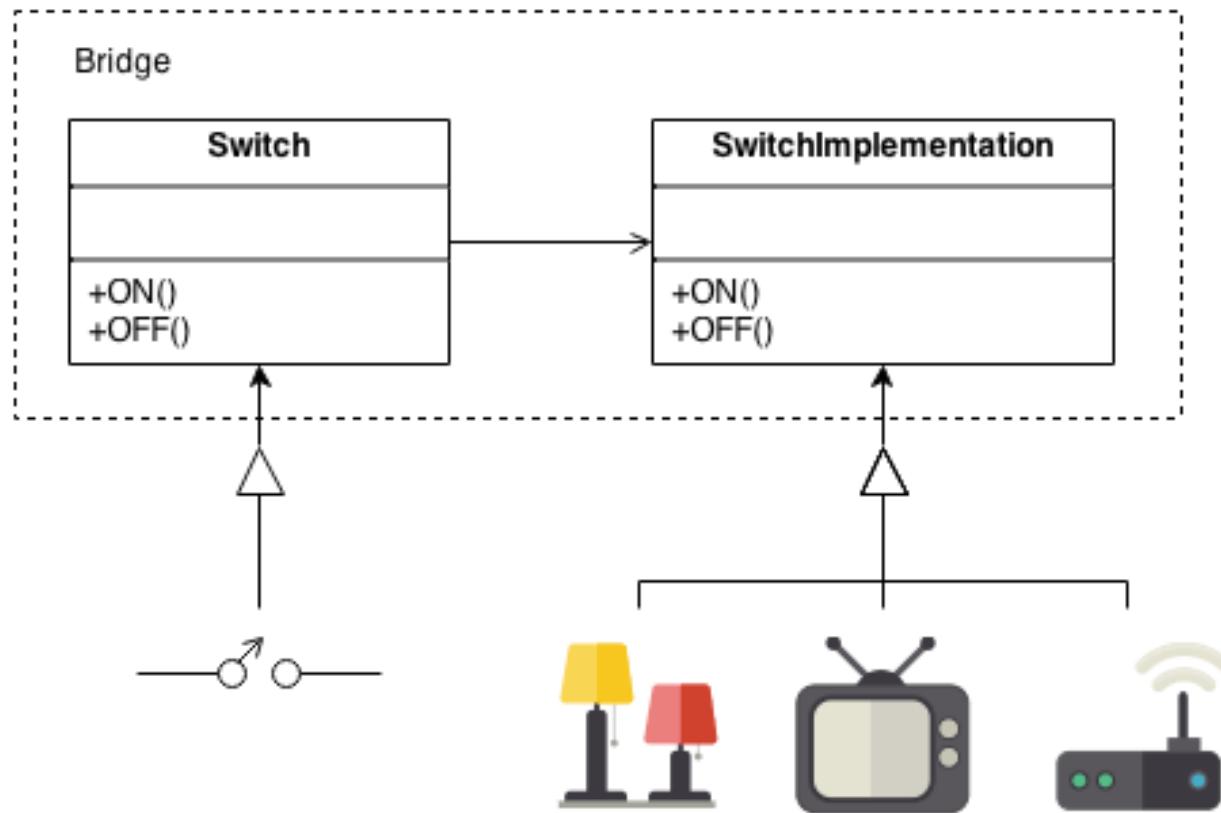
What if we had three kinds of thread schedulers, and four kinds of platforms? What if we had five kinds of thread schedulers, and ten kinds of platforms? The number of classes we would have to define is the product of the number of scheduling schemes and the number of platforms.

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.



# Example

The Bridge pattern decouples an abstraction from its implementation, so that the two can vary independently. A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two position switch, or a variety of dimmer switches.



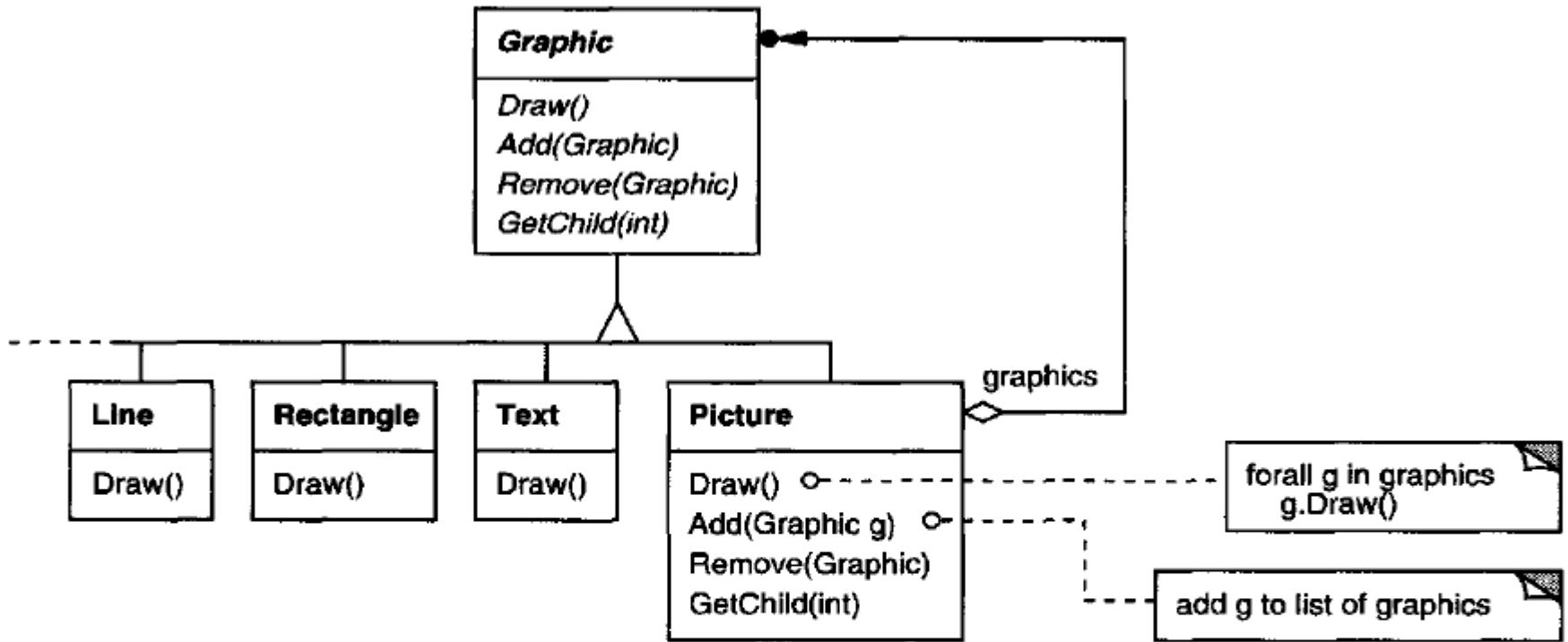
## ❖ Composite

- ▶ Object Structural

### Intent

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

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components. A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.

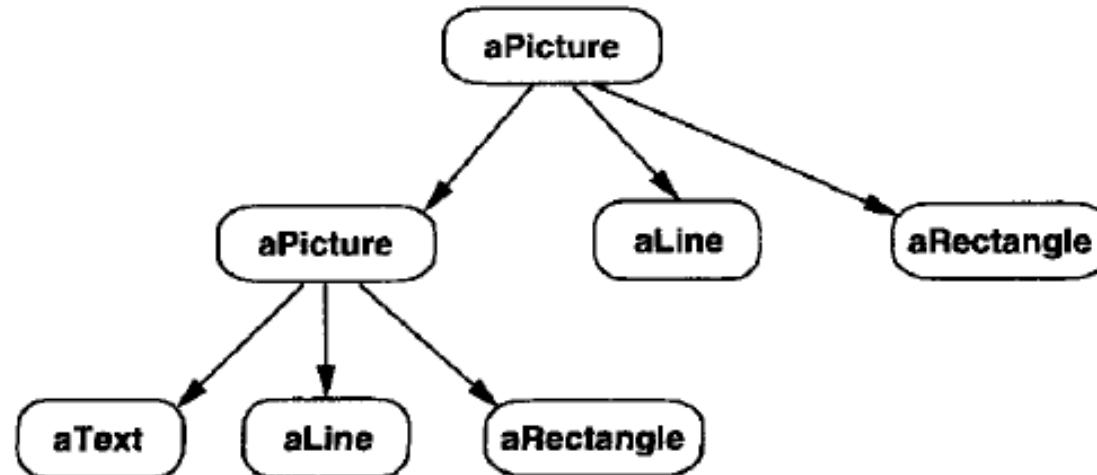


The key to the Composite pattern is an abstract class that represents *both* primitives and their containers. For the graphics system, this class is **Graphic**. **Graphic** declares operations like **Draw** that are specific to graphical objects. It also declares operations that all composite objects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

The subclasses Line, Rectangle, and Text (see preceding class diagram) define primitive graphical objects. These classes implement Draw to draw lines, rectangles, and text, respectively. Since primitive graphics have no child graphics, none of these subclasses implements child-related operations.

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. Because the Picture interface conforms to the Graphic interface, Picture objects can compose other Pictures recursively.

The following diagram shows a typical composite object structure of recursively composed Graphic objects:

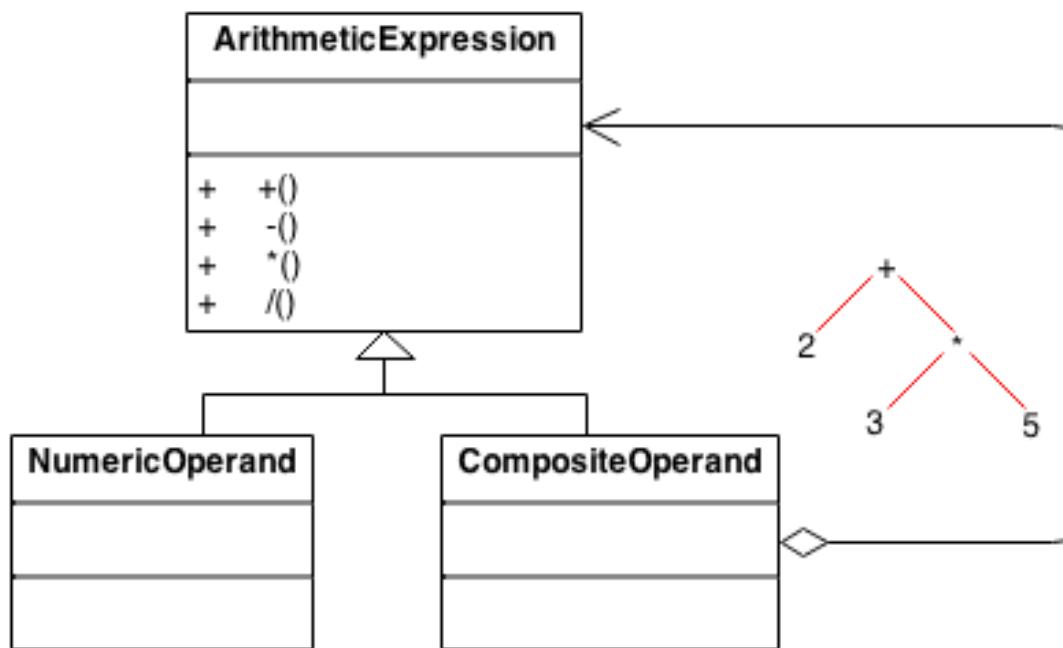


# Participants

- **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.

# Example

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



## ❖ **Decorator**

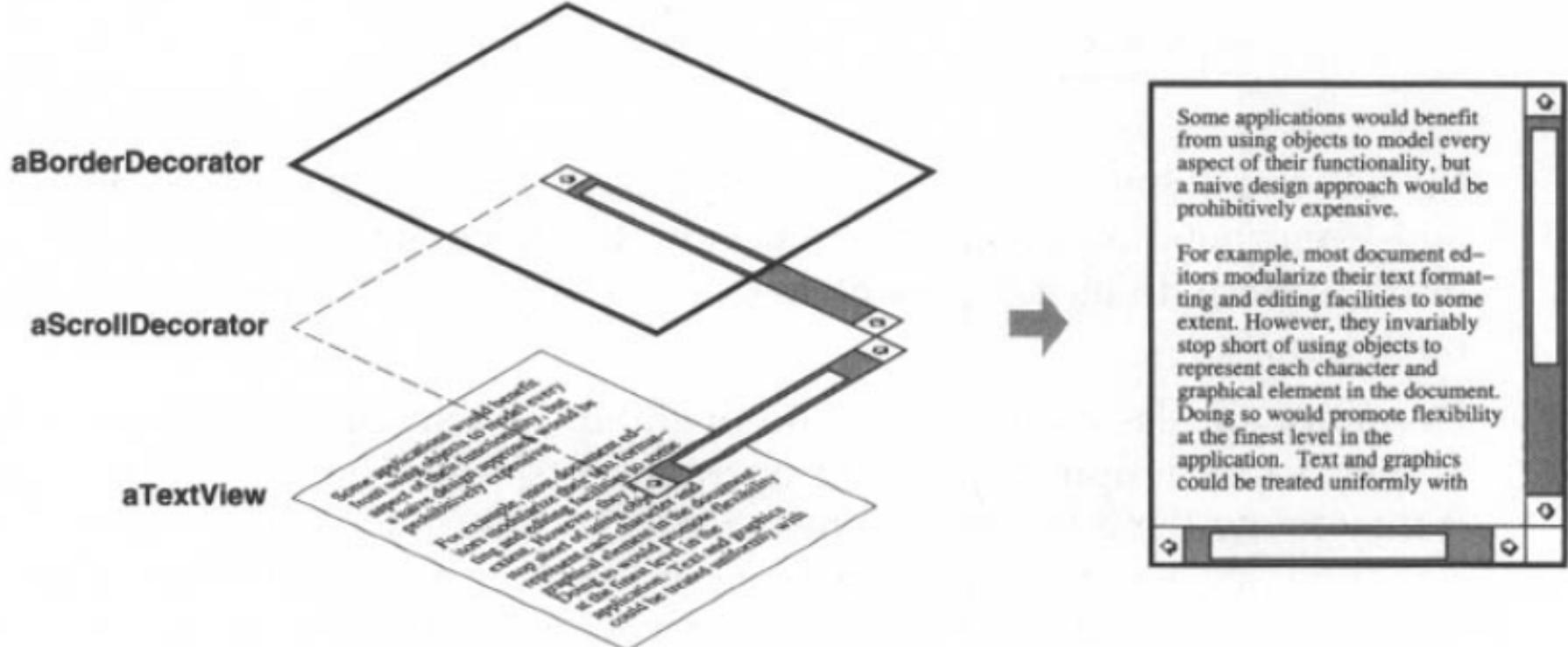
- ▶ Object Structural

### **Intent**

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

### **Also Known As**

Wrapper

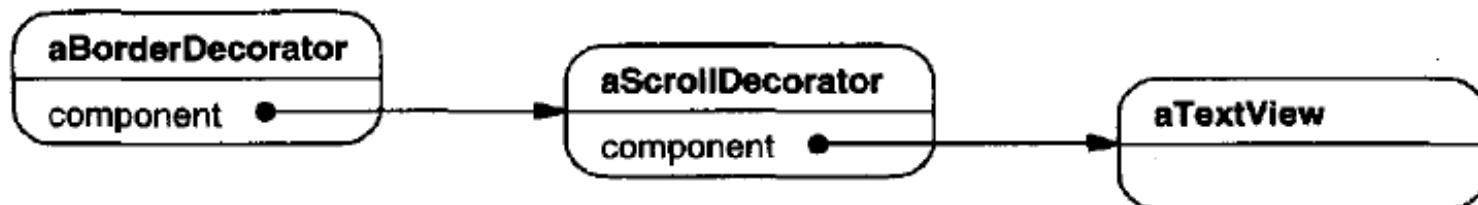


Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.

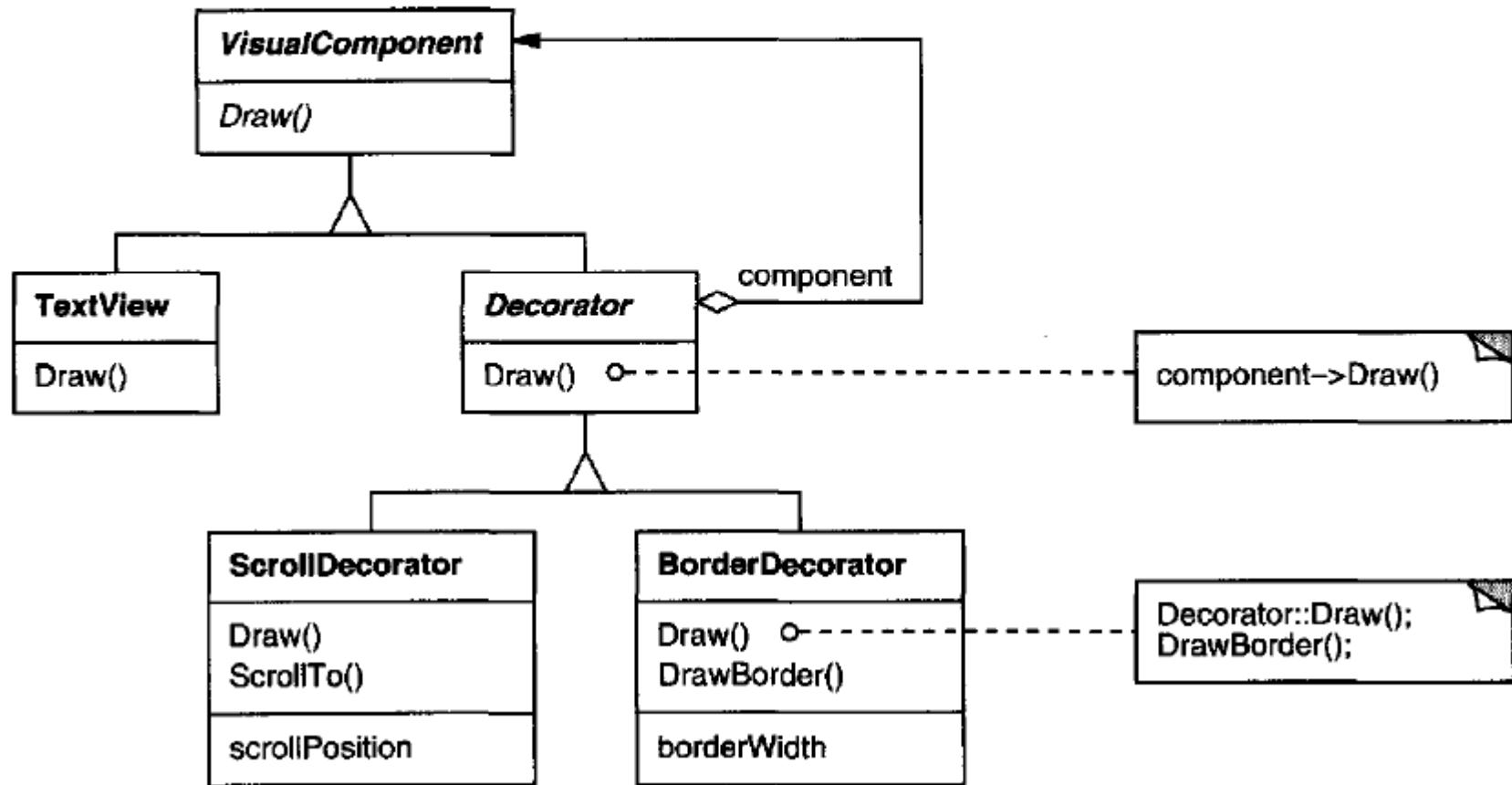
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with

For example, 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. When we do, we can use a `ScrollDecorator` to add them. Suppose we also want to add a thick black border around the `TextView`. We can use a `BorderDecorator` to add this as well. We simply compose the decorators with the `TextView` to produce the desired result.

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.



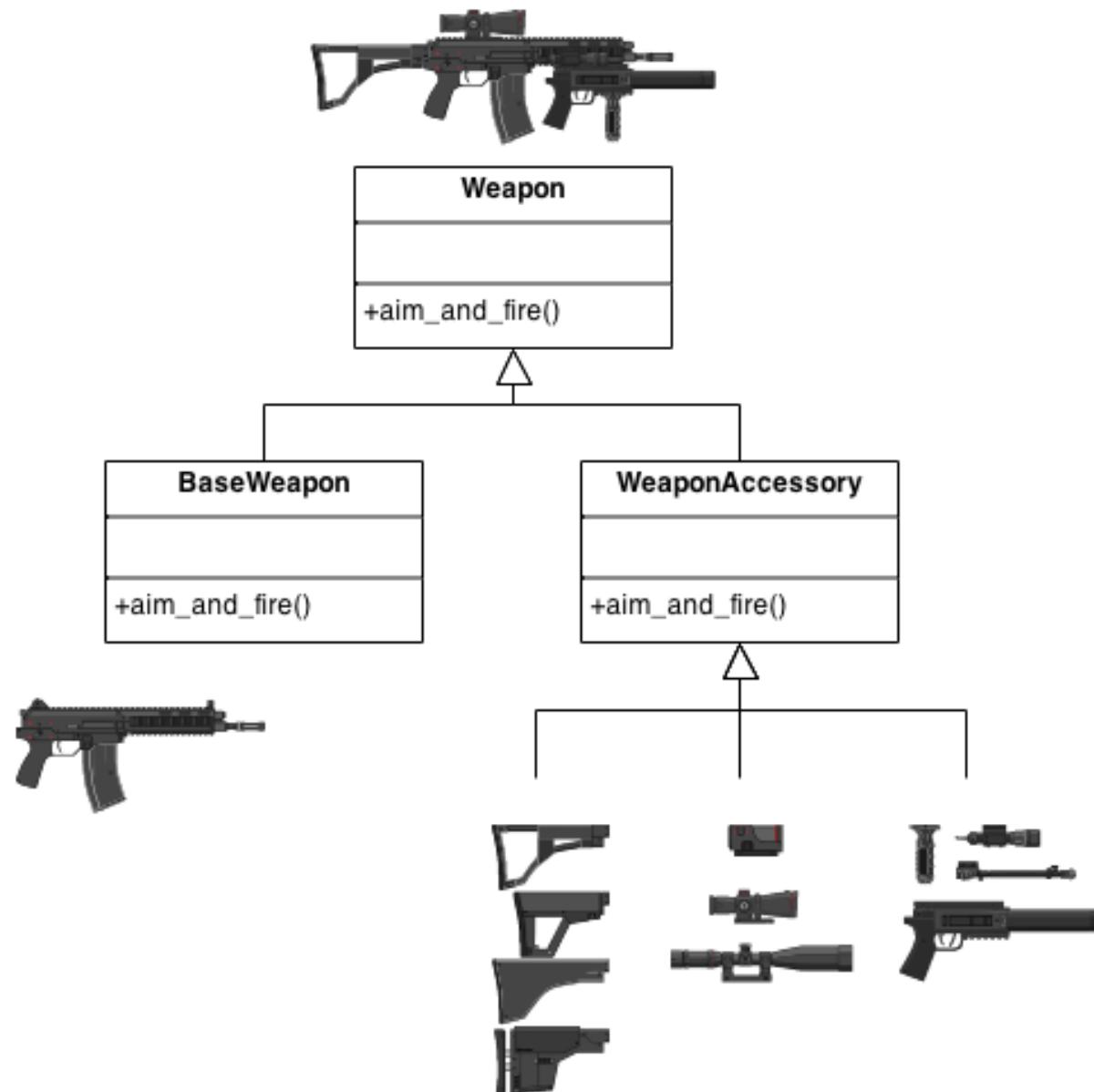
# Participants

- **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.

## Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.

Another example: assault gun is a deadly weapon on it's own. But you can apply certain "decorations" to make it more accurate, silent and devastating.



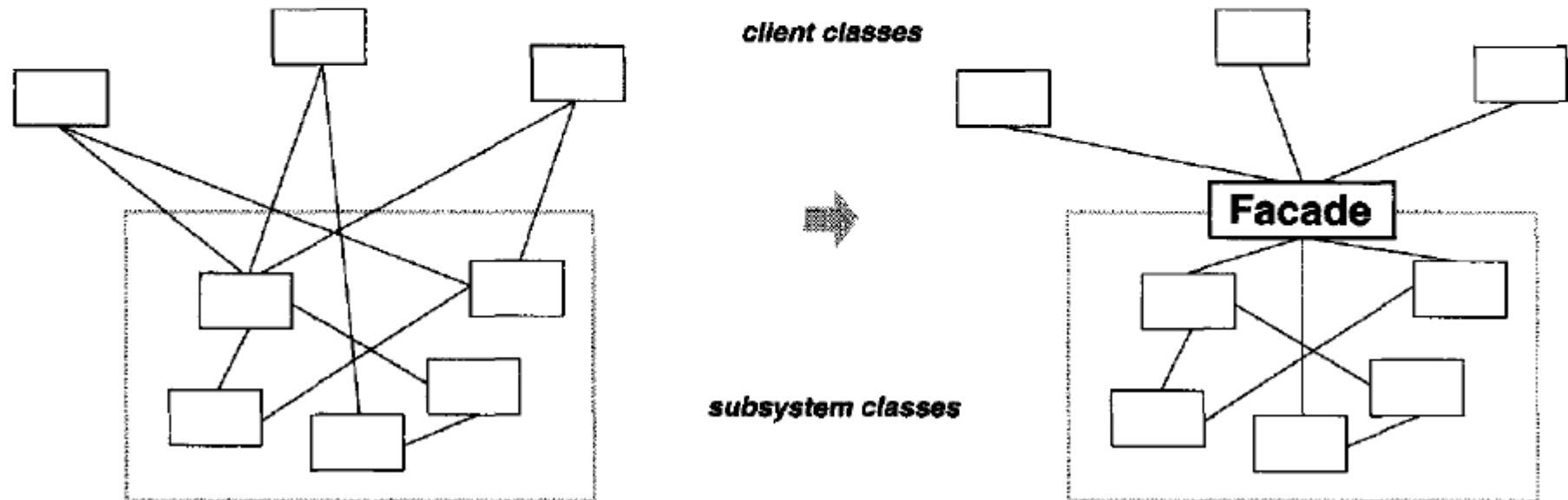
## ❖ Facade

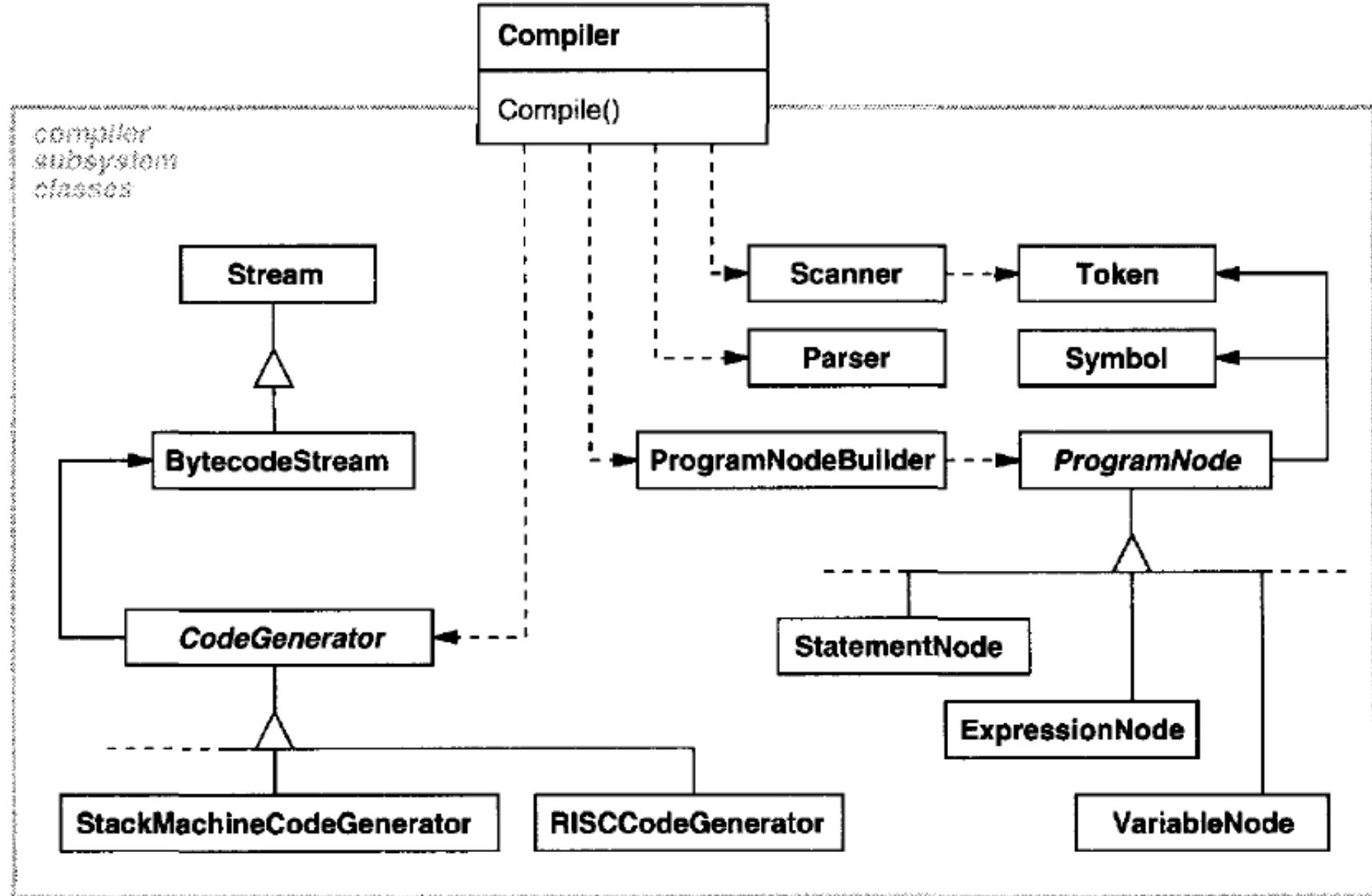
- ▶ Object Structural

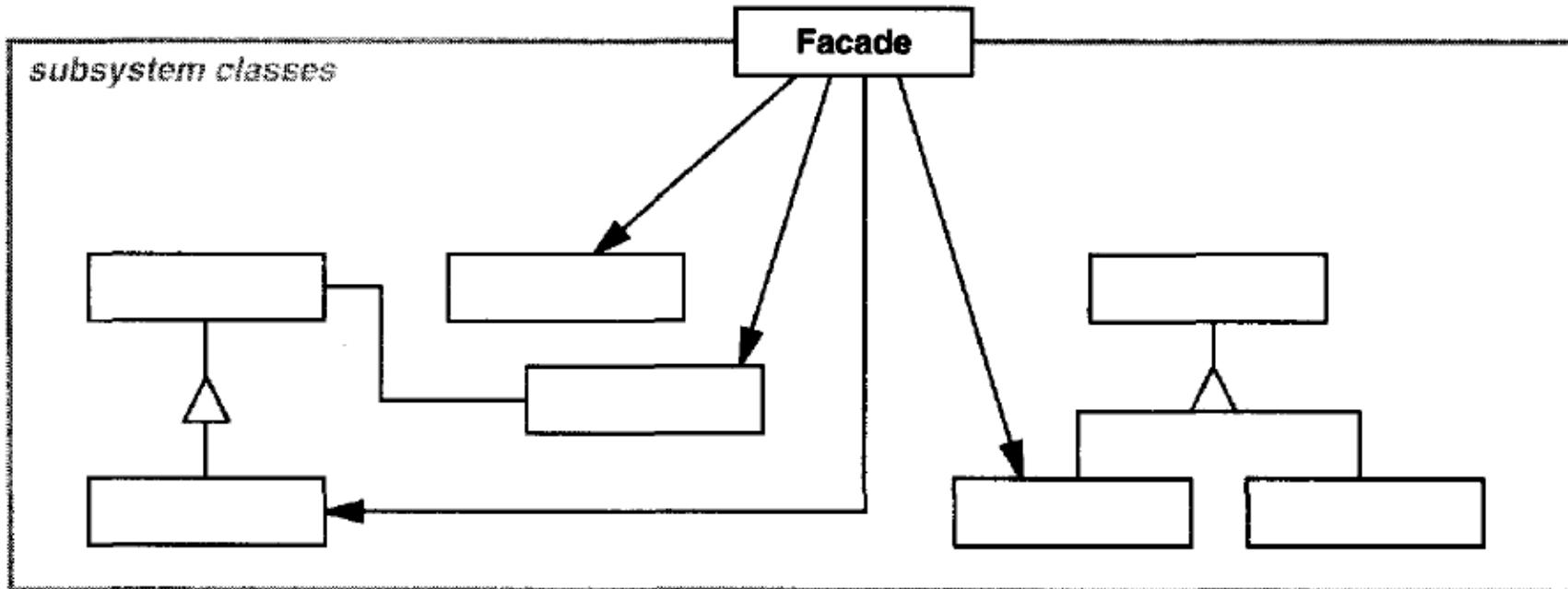
### Intent

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.

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.





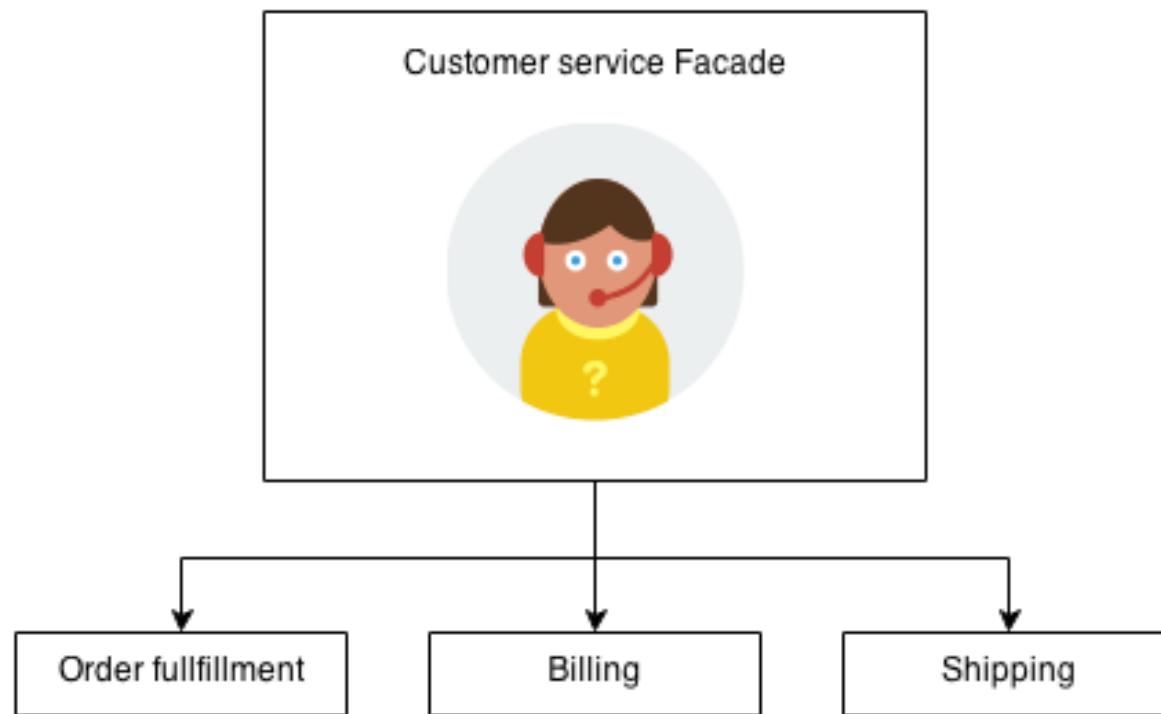


## Participants

- **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 the facade; that is, they keep no references to it.

# Example

The Facade defines a unified, higher level interface to a subsystem that makes it easier to use. Consumers encounter a Facade when ordering from a catalog. The consumer calls one number and speaks with a customer service representative. The customer service representative acts as a Facade, providing an interface to the order fulfillment department, the billing department, and the shipping department.



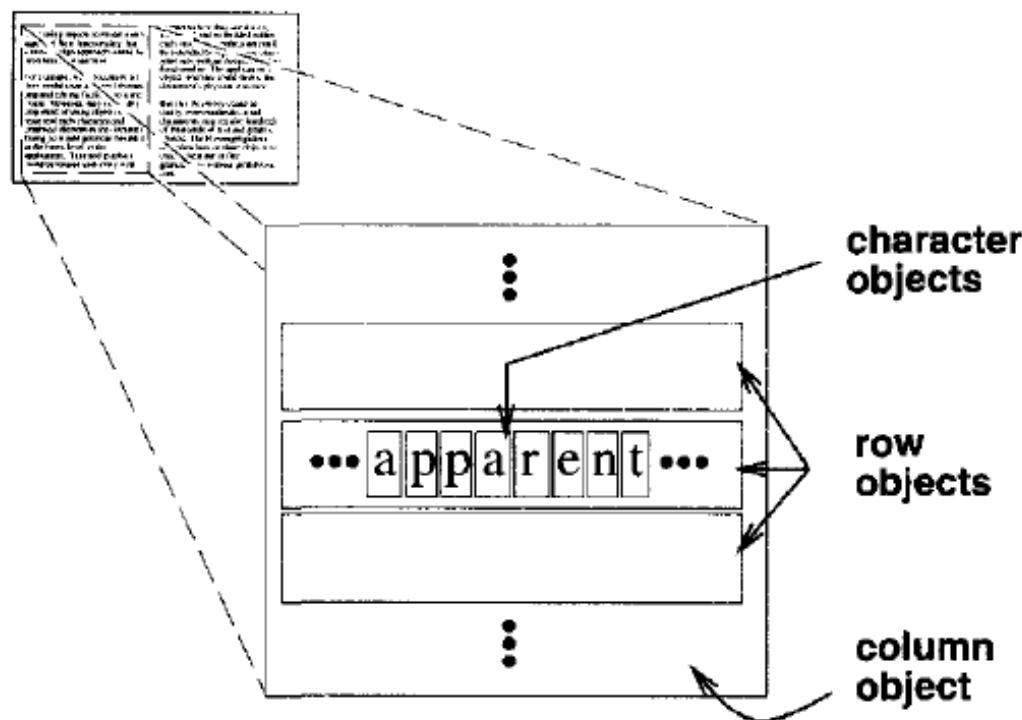
## ❖ Flyweight

- ▶ Object Structural

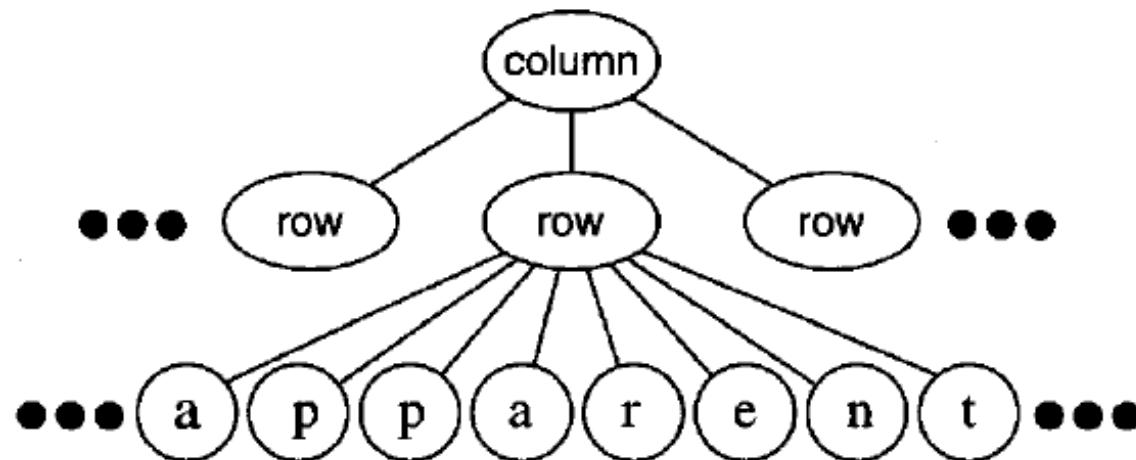
### Intent

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

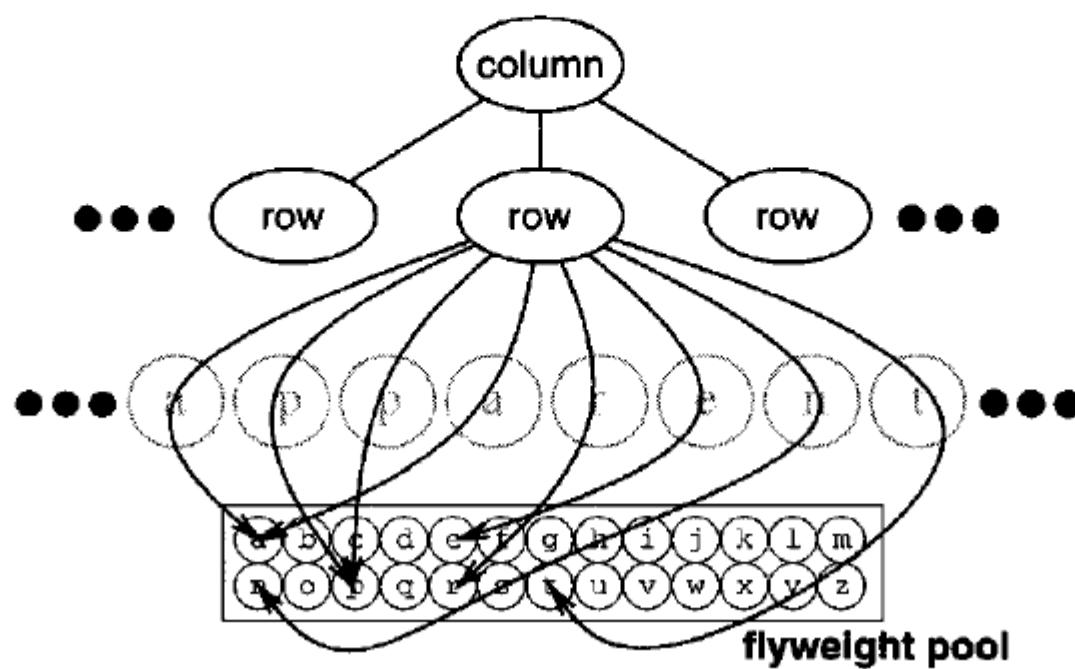
The following diagram shows how a document editor can use objects to represent characters.



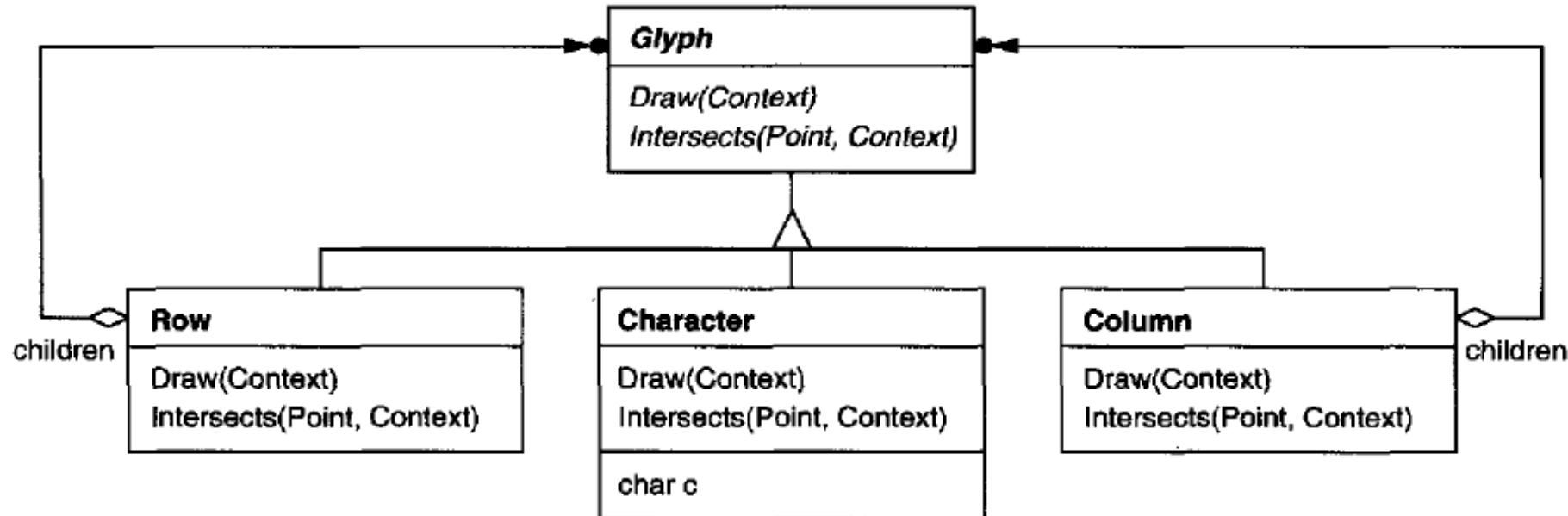
Logically there is an object for every occurrence of a given character in the document:



Physically, however, there is one shared flyweight object per character, and it appears in different contexts in the document structure. 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. For example, Draw and Intersects must know which context the glyph is in before they can do their job.



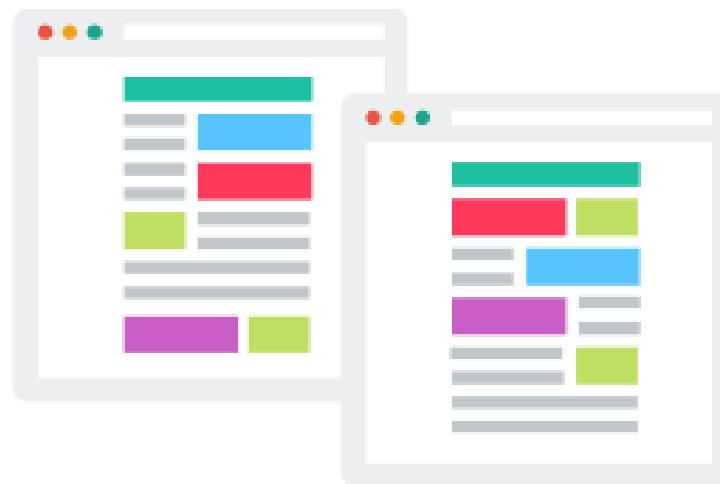
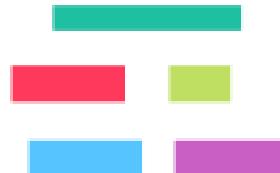
# Participants

- **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, 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).

# Example

The Flyweight uses sharing to support large numbers of objects efficiently. Modern web browsers use this technique to prevent loading same images twice. When browser loads a web page, it traverse through all images on that page. Browser loads all new images from Internet and places them the internal cache. For already loaded images, a flyweight object is created, which has some unique data like position within the page, but everything else is referenced to the cached one.

Browser loads images  
just once and then  
reuses them from pool:



## ❖ **Proxy**

- ▶ Object Structural

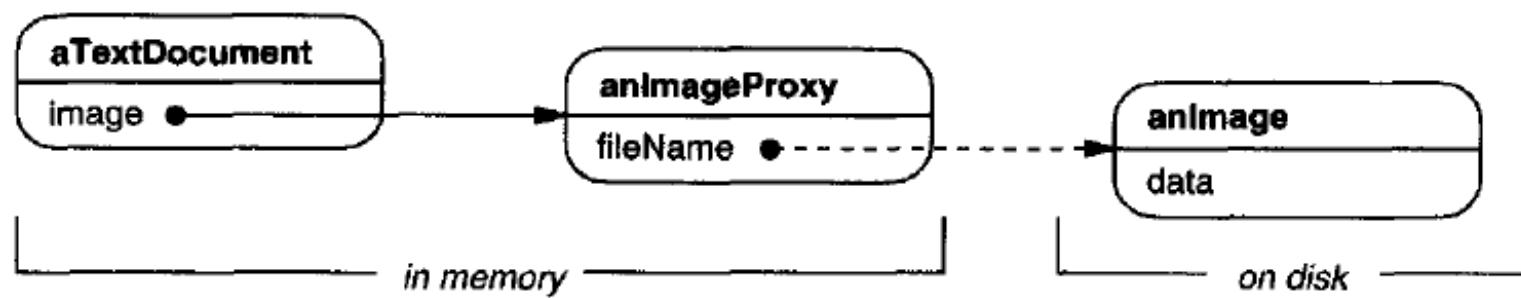
### **Intent**

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

### **Also Known As**

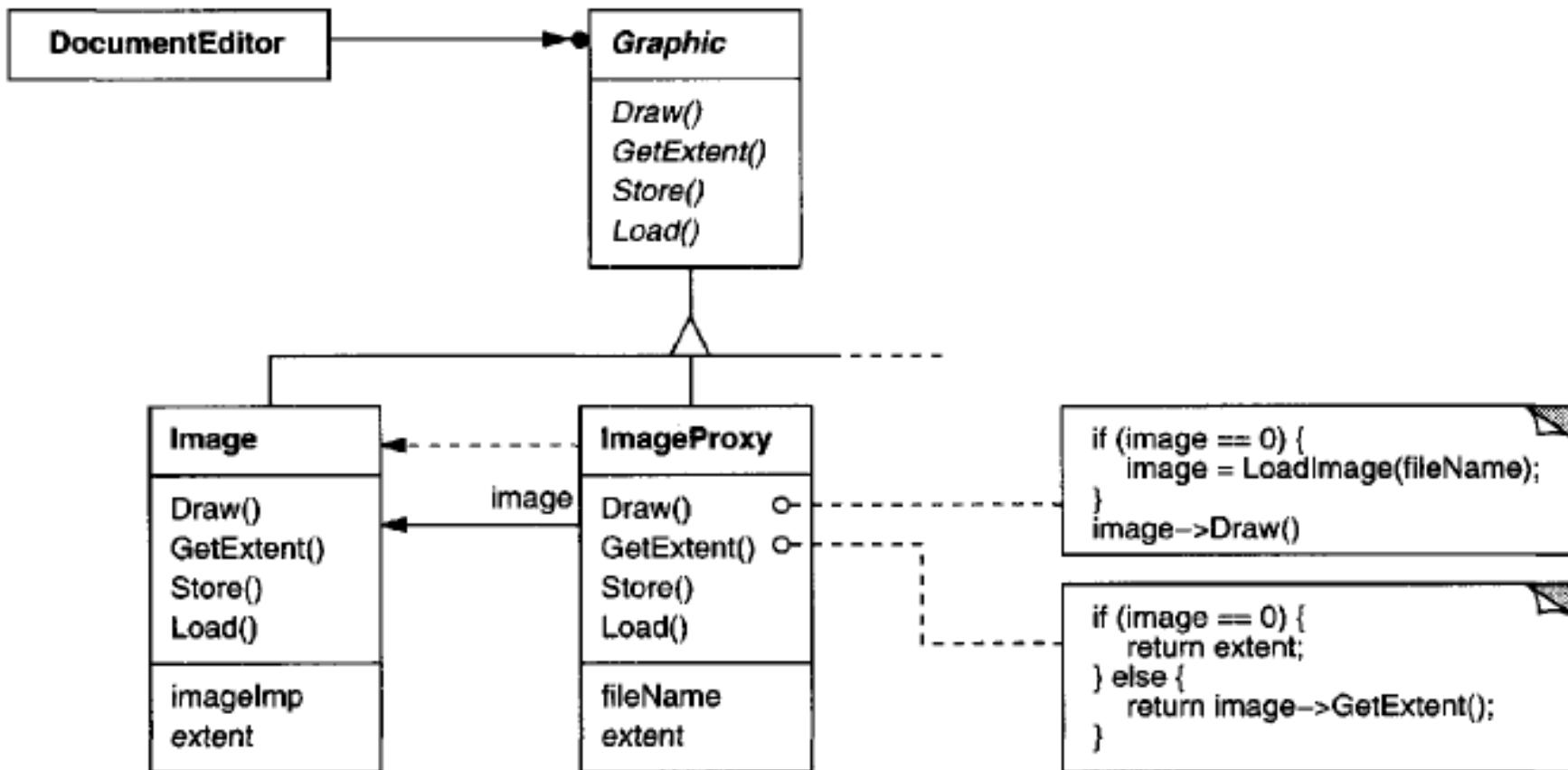
Surrogate

The solution is to use another object, an **image proxy**, that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required.



The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation. The proxy forwards subsequent requests directly to the image. It must therefore keep a reference to the image after creating it.

The following class diagram illustrates this example in more detail.



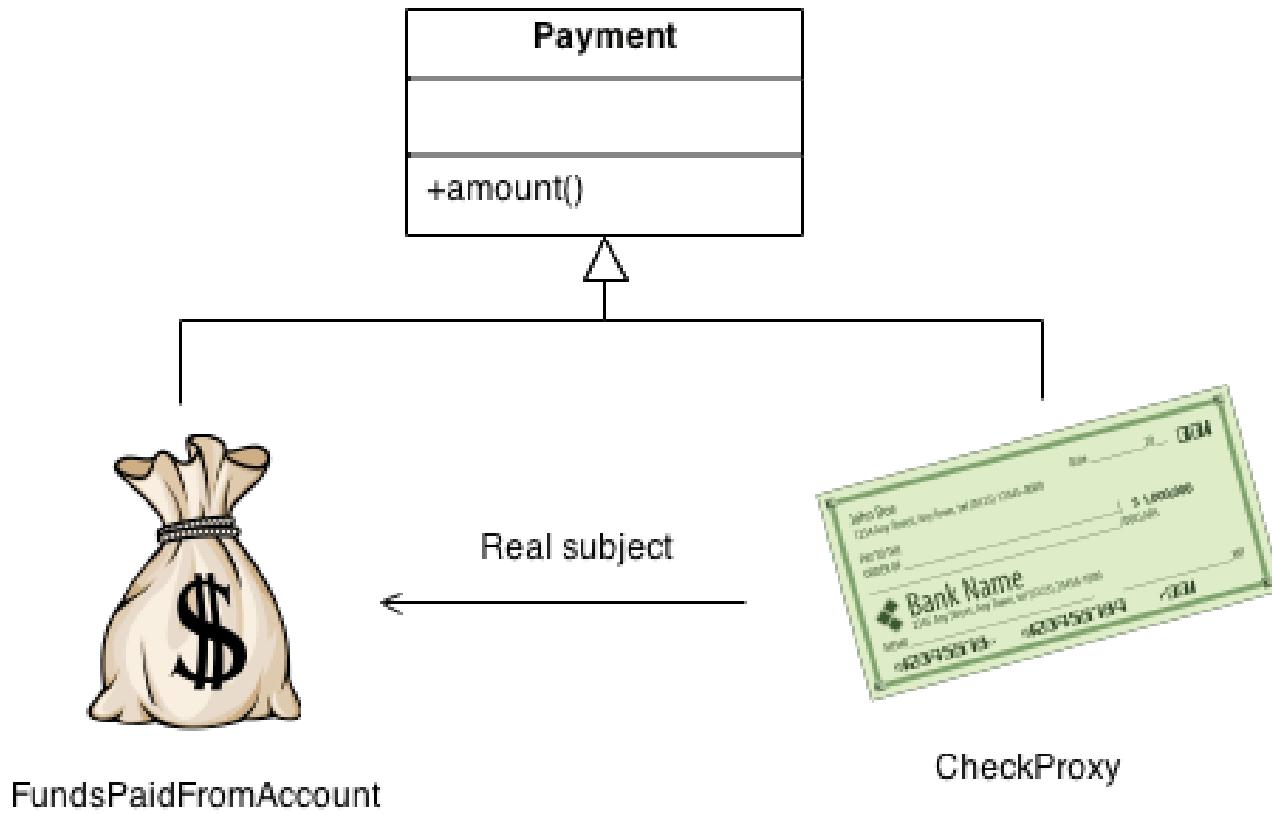
The document editor accesses embedded images through the interface defined by the abstract **Graphic** class. **ImageProxy** is a class for images that are created on demand. **ImageProxy** maintains the file name as a reference to the image on disk. The file name is passed as an argument to the **ImageProxy** constructor.

# Participants

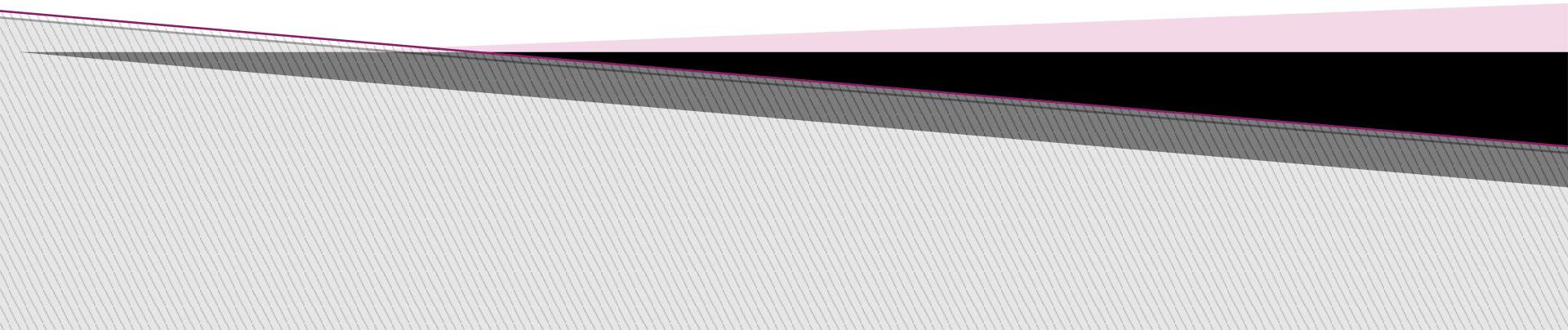
- **Proxy** (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can be substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
- **Subject** (Graphic)
  - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- **RealSubject** (Image)
  - defines the real object that the proxy represents.

# Example

The Proxy provides a surrogate or place holder to provide access to an object. A check or bank draft is a proxy for funds in an account. A check can be used in place of cash for making purchases and ultimately controls access to cash in the issuer's account.



# Behavioral Patterns



- ▶ Behavioral patterns are concerned with **algorithms** and the assignment of **responsibilities between objects**
- ▶ Behavioral patterns describe not just patterns of objects or classes but also the patterns of **communication between them**
- ▶ Behavioral class patterns use **inheritance** to distribute behavior between **classes**
- ▶ Behavioral object patterns use **object composition** rather than inheritance

<b>Behavioral</b>	
<b>Class</b>	Interpreter Template Method
<b>Object</b>	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

## ❖ **Chain of Responsibility**

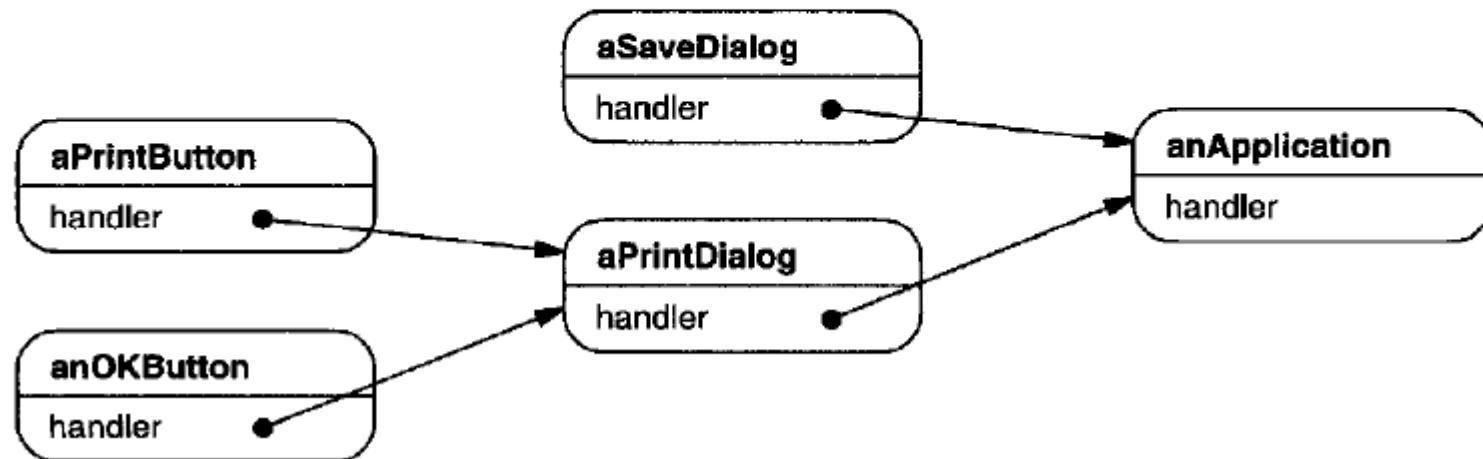
- ▶ **Object Behavioral**

### **Intent**

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

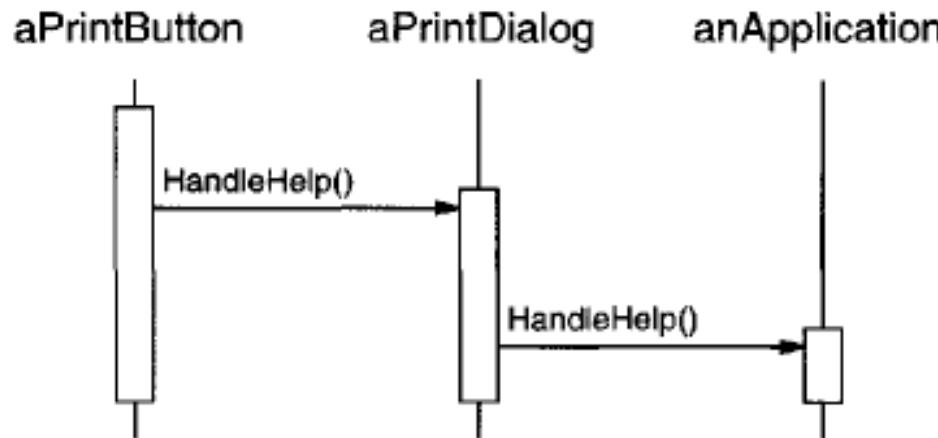
Consider a context-sensitive help facility for a graphical user interface. The user can obtain help information on any part of the interface just by clicking on it. The help that's provided depends on the part of the interface that's selected and its context; for example, a button widget in a dialog box might have different help information than a similar button in the main window. If no specific help information exists for that part of the interface, then the help system should display a more general help message about the immediate context—the dialog box as a whole, for example.

The idea of this pattern is to decouple senders and receivers by giving multiple objects a chance to handle a request. The request gets passed along a chain of objects until one of them handles it.



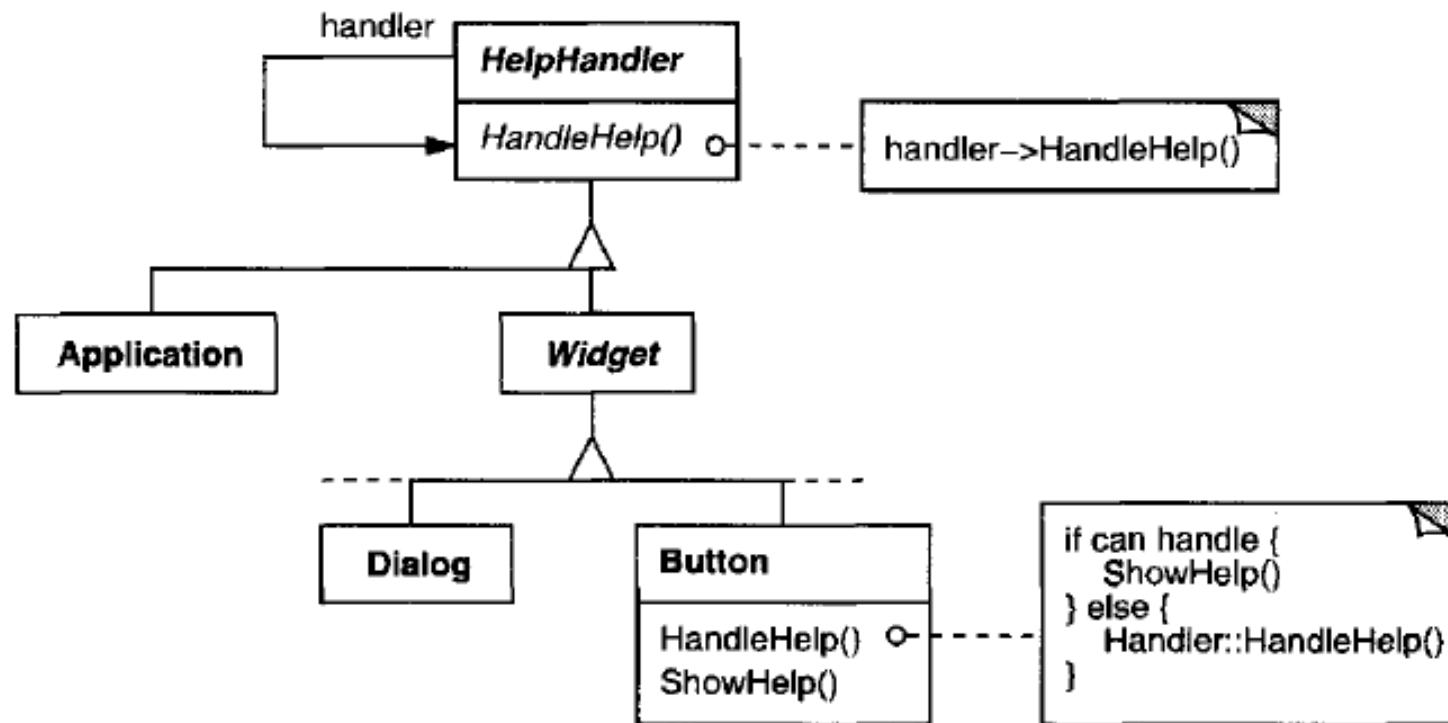
The first object in the chain receives the request and either handles it or forwards it to the next candidate on the chain, which does likewise. The object that made the request has no explicit knowledge of who will handle it—we say the request has an **implicit receiver**.

Let's assume the user clicks for help on a button widget marked "Print." The button is contained in an instance of PrintDialog, which knows the application object it belongs to (see preceding object diagram). The following interaction diagram illustrates how the help request gets forwarded along the chain:

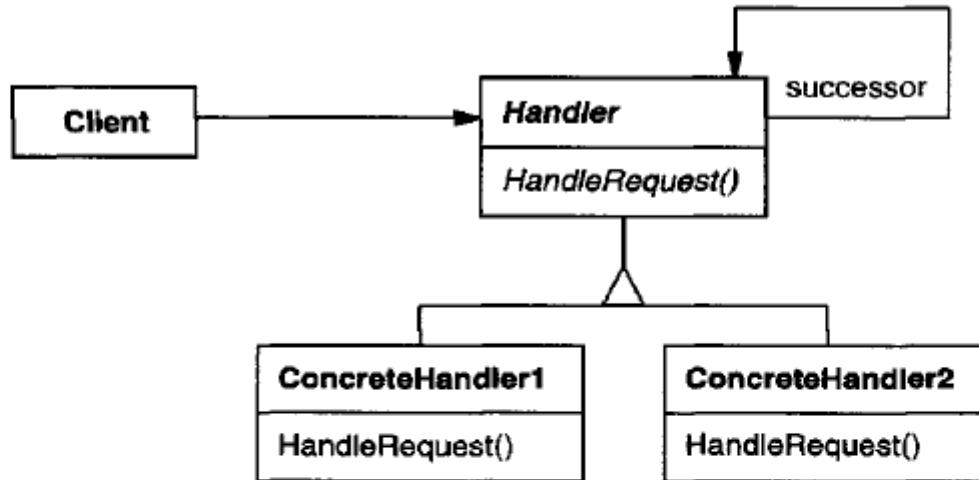


In this case, neither aPrintButton nor aPrintDialog handles the request; it stops at anApplication, which can handle it or ignore it. The client that issued the request has no direct reference to the object that ultimately fulfills it.

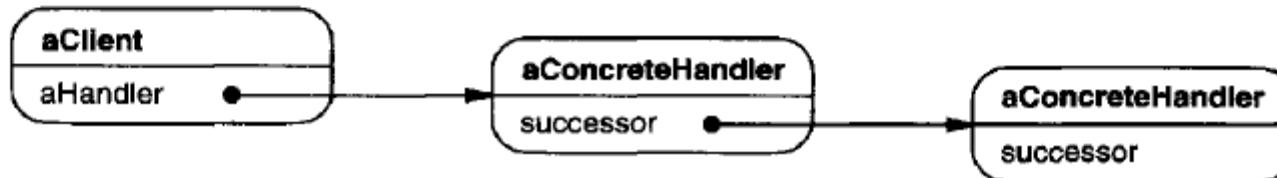
For example, the help system might define a HelpHandler class with a corresponding HandleHelp operation. HelpHandler can be the parent class for candidate object classes, or it can be defined as a mixin class. Then classes that want to handle help requests can make HelpHandler a parent:



The Button, Dialog, and Application classes use HelpHandler operations to handle help requests. HelpHandler's HandleHelp operation forwards the request to the successor by default. Subclasses can override this operation to provide help under the right circumstances; otherwise they can use the default implementation to forward the request.



A typical object structure might look like this:

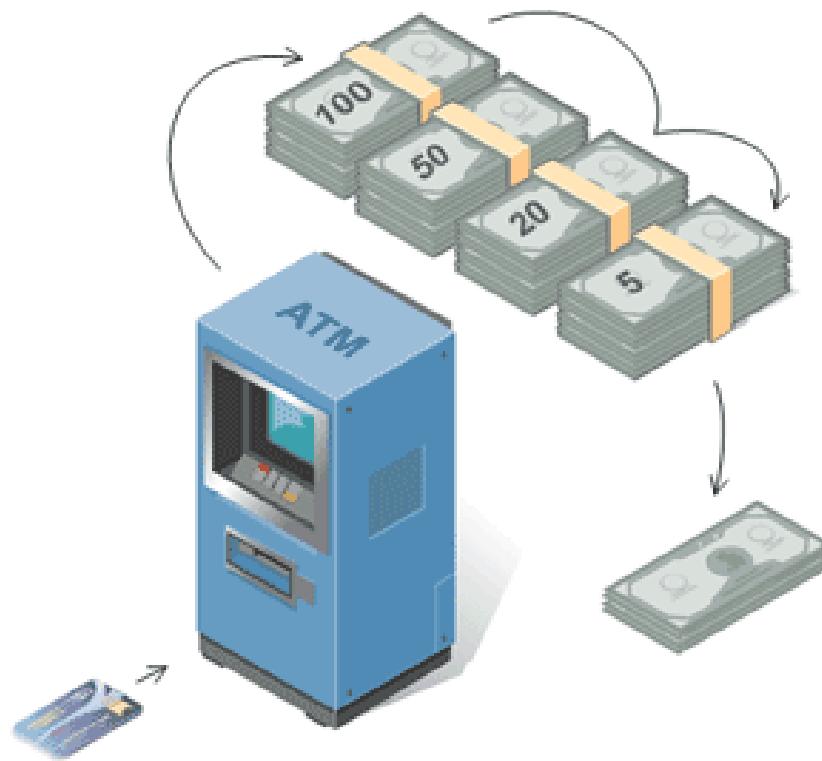


# Participants

- **Handler** (`HelpHandler`)
  - defines an interface for handling requests.
  - (optional) implements the successor link.
- **ConcreteHandler** (`PrintButton`, `PrintDialog`)
  - handles requests it is responsible for.
  - can access its successor.
  - if the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- **Client**
  - initiates the request to a ConcreteHandler object on the chain.

# Example

The Chain of Responsibility pattern avoids coupling the sender of a request to the receiver by giving more than one object a chance to handle the request. ATM use the Chain of Responsibility in money giving mechanism.



## ❖ Command

- ▶ Object Behavioral

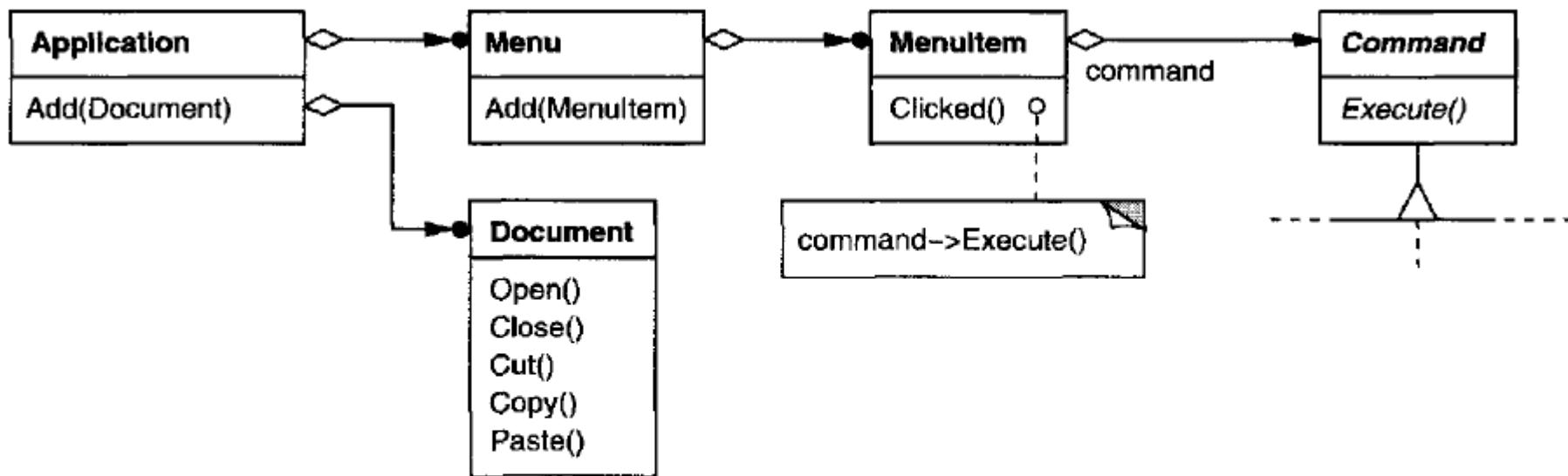
### Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

### Also Known As

Action, Transaction

Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request. For example, user interface toolkits include objects like buttons and menus that carry out a request in response to user input. But the toolkit can't implement the request explicitly in the button or menu, because only applications that use the toolkit know what should be done on which object. As toolkit designers we have no way of knowing the receiver of the request or the operations that will carry it out.

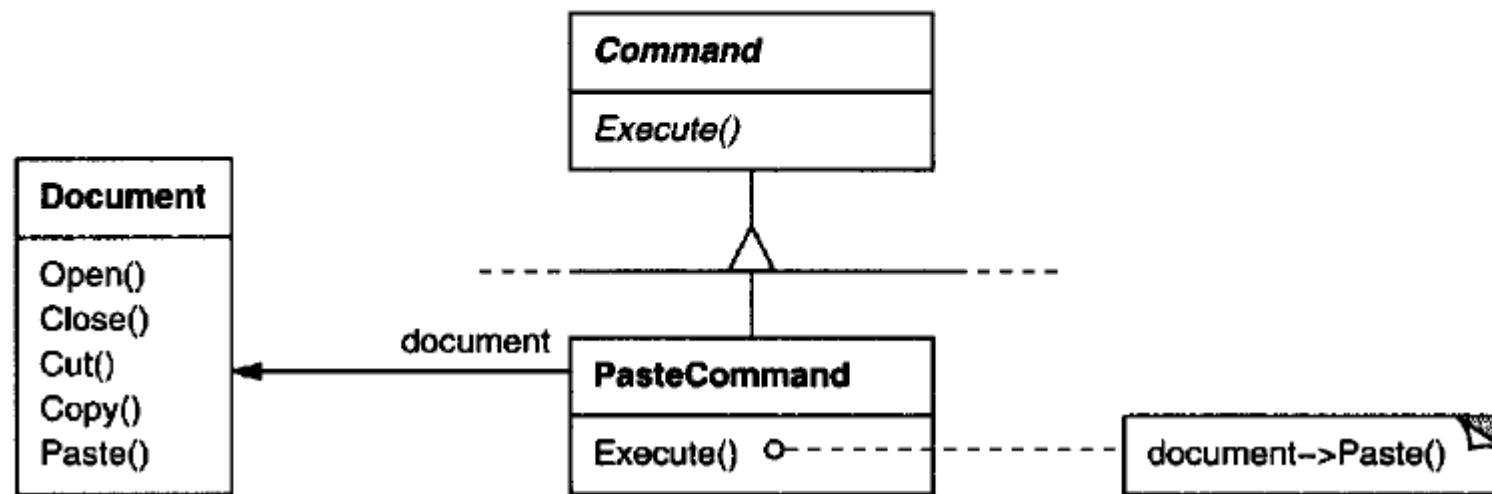


Menus can be implemented easily with Command objects. Each choice in a Menu is an instance of a MenuItem class. An Application class creates these menus and

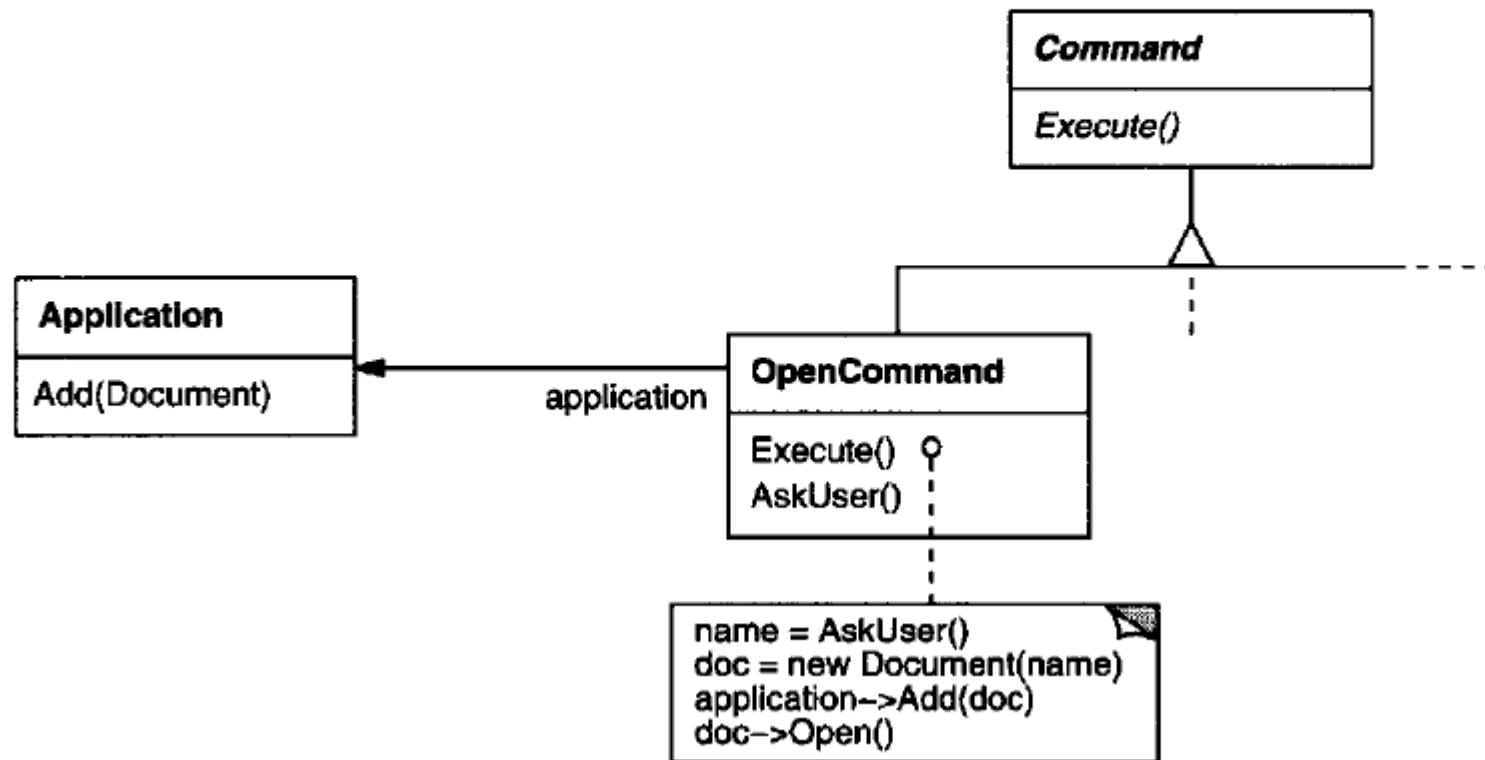
their menu items along with the rest of the user interface. The Application class also keeps track of Document objects that a user has opened.

The application configures each MenuItem with an instance of a concrete Command subclass. When the user selects a MenuItem, the MenuItem calls Execute on its command, and Execute carries out the operation. MenuItems don't know which subclass of Command they use. Command subclasses store the receiver of the request and invoke one or more operations on the receiver.

For example, PasteCommand supports pasting text from the clipboard into a Document. PasteCommand's receiver is the Document object it is supplied upon instantiation. The Execute operation invokes Paste on the receiving Document.



OpenCommand's Execute operation is different: it prompts the user for a document name, creates a corresponding Document object, adds the document to the receiving application, and opens the document.

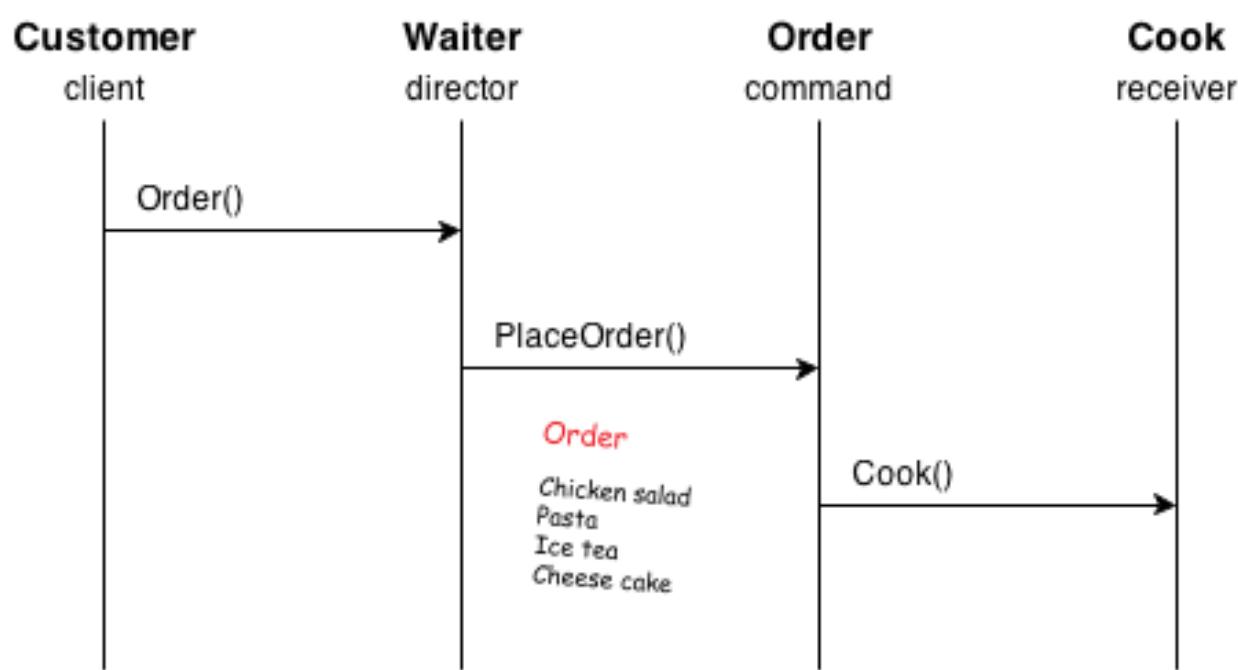


# Participants

- **Command**
  - declares an interface for executing an operation.
- **ConcreteCommand** (PasteCommand, OpenCommand)
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client** (Application)
  - creates a ConcreteCommand object and sets its receiver.
- **Invoker** (MenuItem)
  - asks the command to carry out the request.

# Example

The Command pattern allows requests to be encapsulated as objects, thereby allowing clients to be parametrized with different requests. The "check" at a diner is an example of a Command pattern. The waiter or waitress takes an order or command from a customer and encapsulates that order by writing it on the check. The order is then queued for a short order cook. Note that the pad of "checks" used by each waiter is not dependent on the menu, and therefore they can support commands to cook many different items.



## ❖ Interpreter

- ▶ Class Behavioral

### Intent

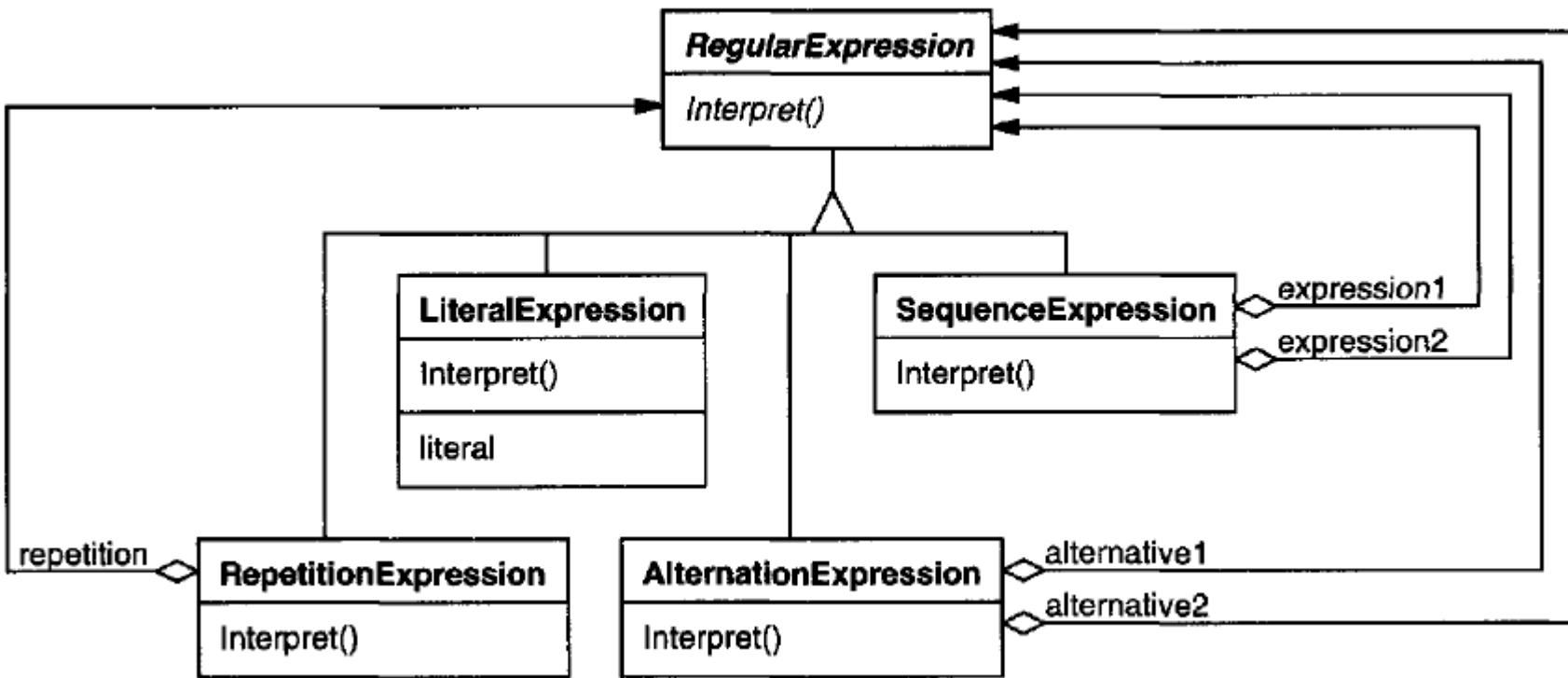
Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

The Interpreter pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences. In this example, the pattern describes how to define a grammar for regular expressions, represent a particular regular expression, and how to interpret that regular expression.

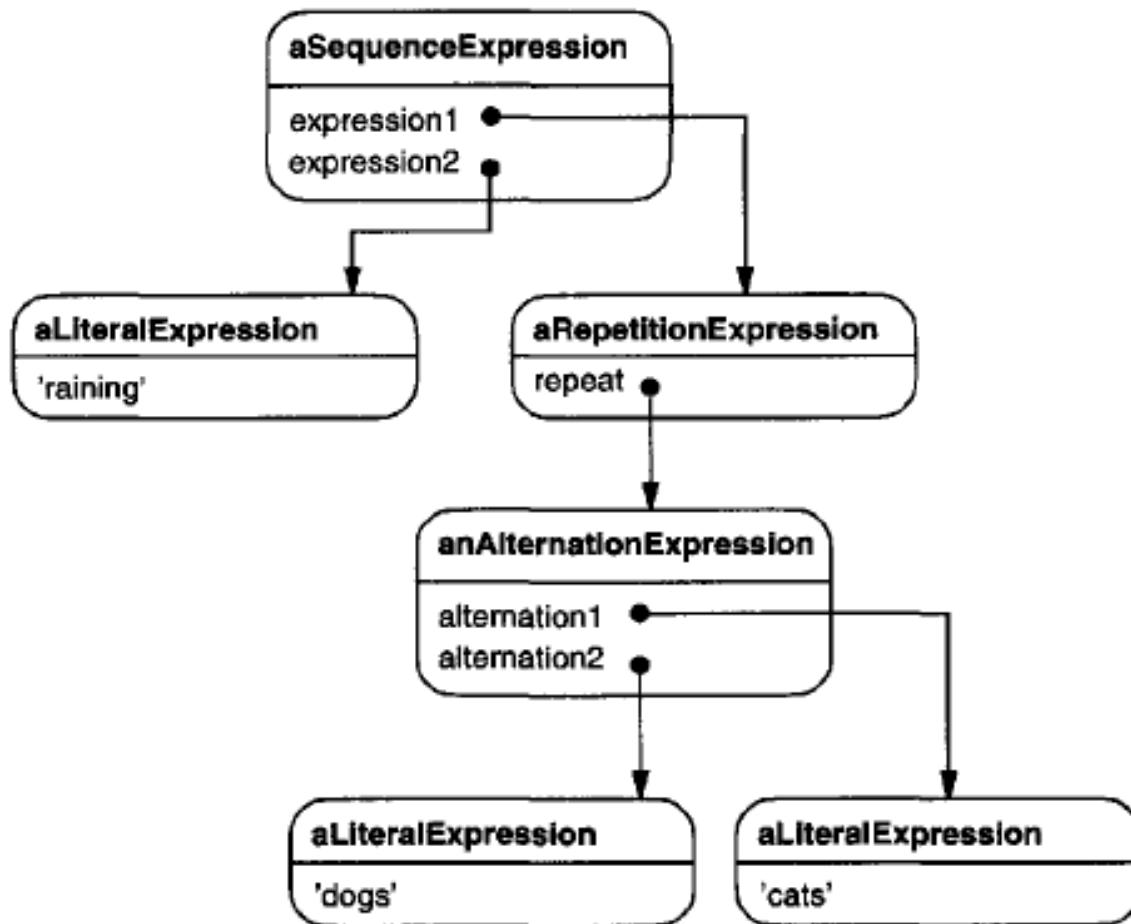
Suppose the following grammar defines the regular expressions:

```
expression ::= literal | alternation | sequence | repetition |
              '(' expression ')'
alternation ::= expression ' | ' expression
sequence ::= expression '&' expression
repetition ::= expression '**'
literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*
```

The symbol **expression** is the start symbol, and **literal** is a terminal symbol defining simple words

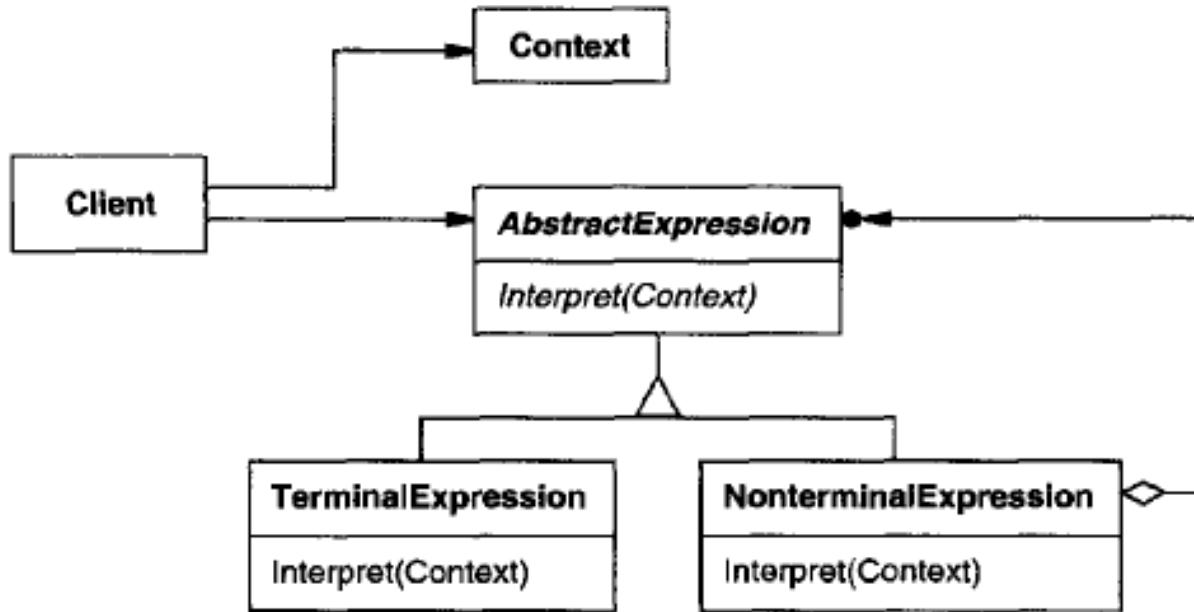


Every regular expression defined by this grammar is represented by an abstract syntax tree made up of instances of these classes. For example, the abstract syntax tree



represents the regular expression

```
raining & (dogs | cats)|*
```



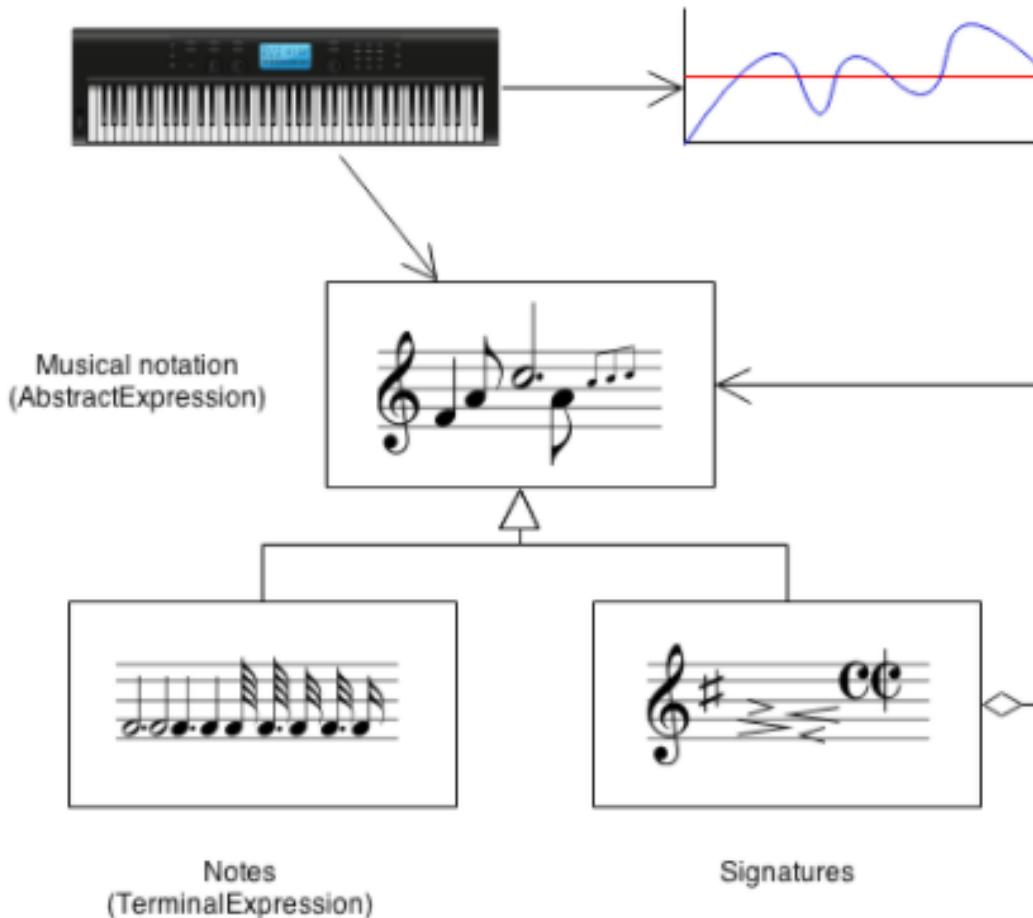
## Participants

- **AbstractExpression** (RegularExpression)
  - declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

- **TerminalExpression** (`LiteralExpression`)
  - implements an `Interpret` operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in a sentence.
- **NonterminalExpression** (`AlternationExpression`, `RepetitionExpression`, `SequenceExpressions`)
  - one such class is required for every rule  $R ::= R_1 R_2 \dots R_n$  in the grammar.
  - maintains instance variables of type `AbstractExpression` for each of the symbols  $R_1$  through  $R_n$ .
  - implements an `Interpret` operation for nonterminal symbols in the grammar. `Interpret` typically calls itself recursively on the variables representing  $R_1$  through  $R_n$ .
- **Context**
  - contains information that's global to the interpreter.
- **Client**
  - builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines. The abstract syntax tree is assembled from instances of the `NonterminalExpression` and `TerminalExpression` classes.
  - invokes the `Interpret` operation.

# Example

The Interpreter pattern defines a grammatical representation for a language and an interpreter to interpret the grammar. Musicians are examples of Interpreters. The pitch of a sound and its duration can be represented in musical notation on a staff. This notation provides the language of music. Musicians playing the music from the score are able to reproduce the original pitch and duration of each sound represented.



## ❖ Iterator

- ▶ Object Behavioral

### Intent

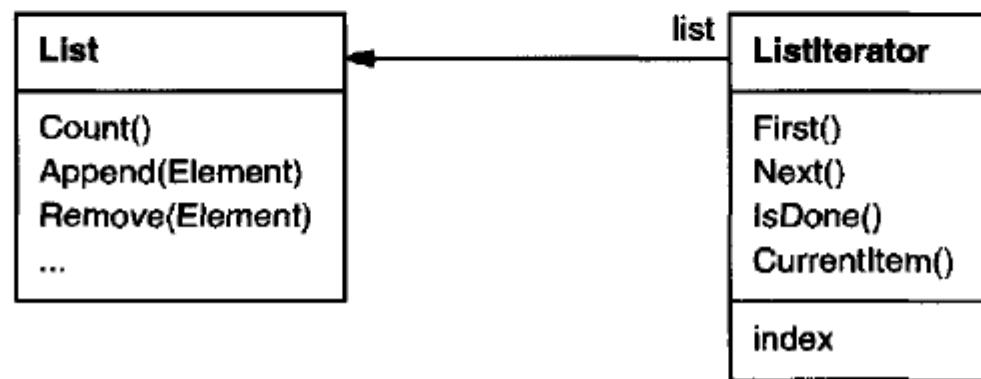
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

### Also Known As

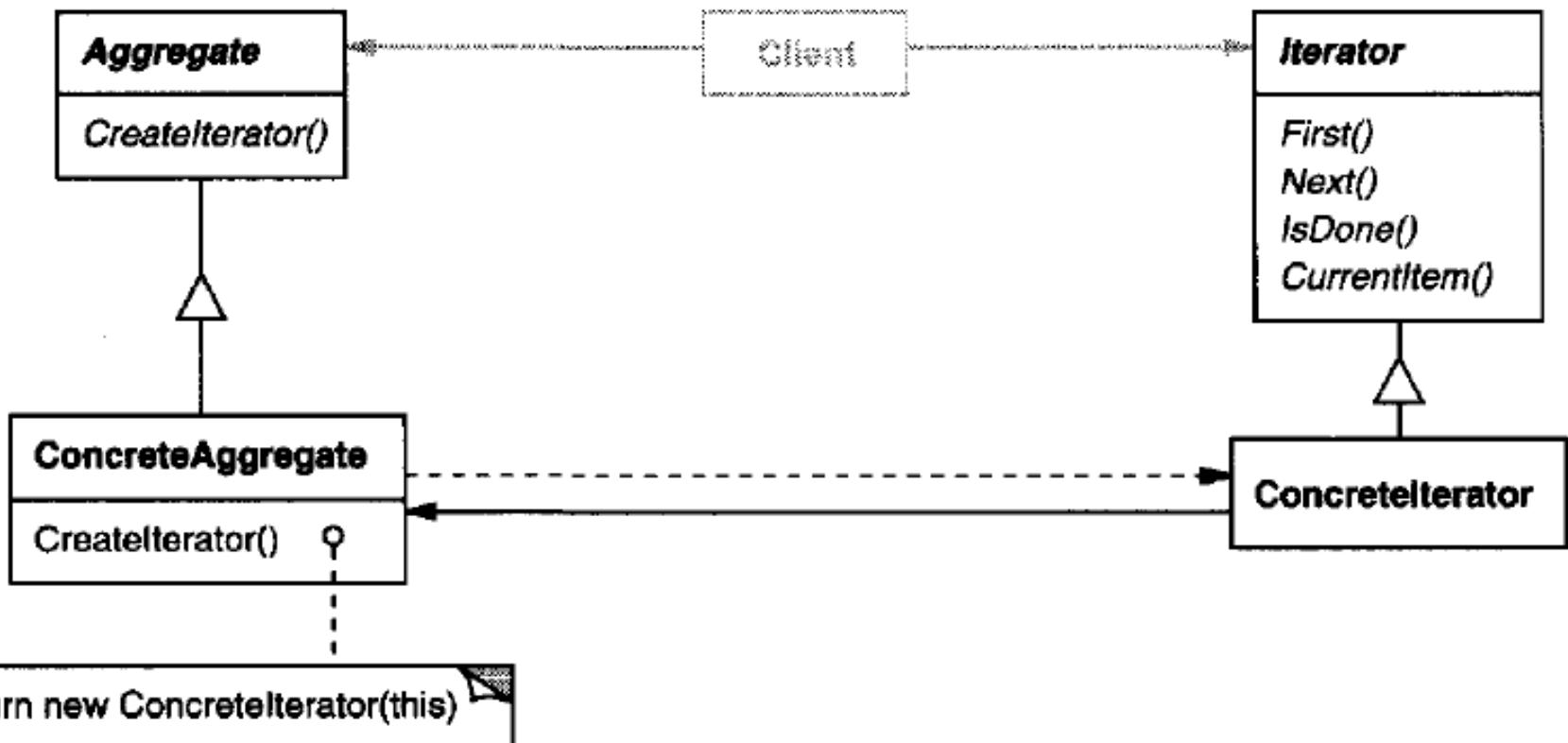
Cursor

The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an **iterator** object. The Iterator class defines an interface for accessing the list's elements. An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

For example, a List class would call for a ListIterator with the following relationship between them:



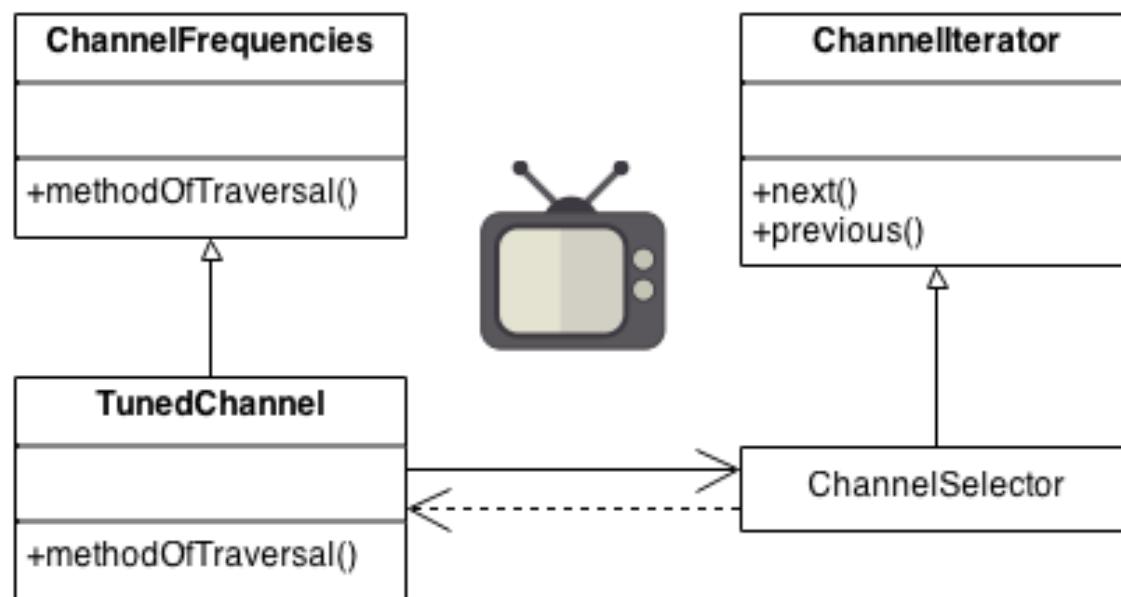
Before you can instantiate ListIterator, you must supply the List to traverse. Once you have the ListIterator instance, you can access the list's elements sequentially. The CurrentItem operation returns the current element in the list, First initializes the current element to the first element, Next advances the current element to the next element, and IsDone tests whether we've advanced beyond the last element—that is, we're finished with the traversal.



# Participants

- **Iterator**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate**
  - defines an interface for creating an Iterator object.
- **ConcreteAggregate**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

On early television sets, a dial was used to change channels. When channel surfing, the viewer was required to move the dial through each channel position, regardless of whether or not that channel had reception. On modern television sets, a next and previous button are used. When the viewer selects the "next" button, the next tuned channel will be displayed. Consider watching television in a hotel room in a strange city. When surfing through channels, the channel number is not important, but the programming is. If the programming on one channel is not of interest, the viewer can request the next channel, without knowing its number.



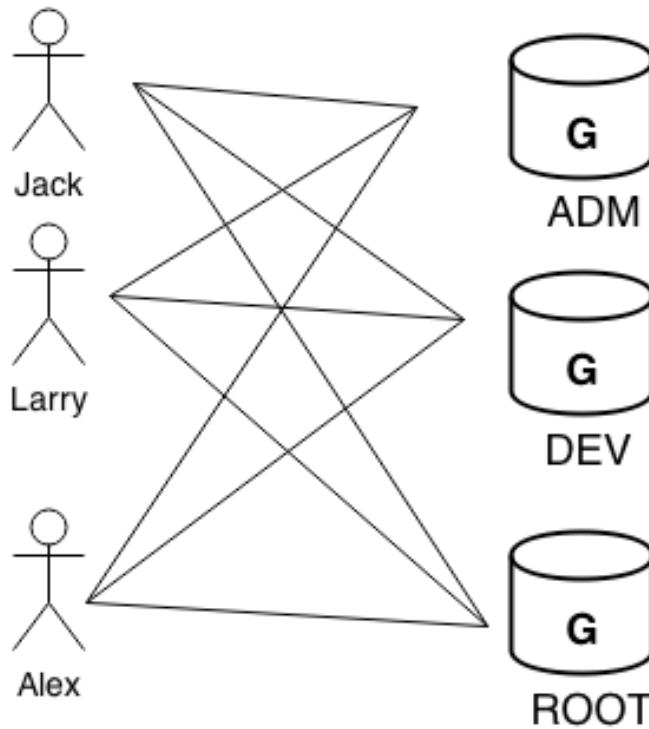
## ❖ Mediator

- ▶ Object Behavioral

### Intent

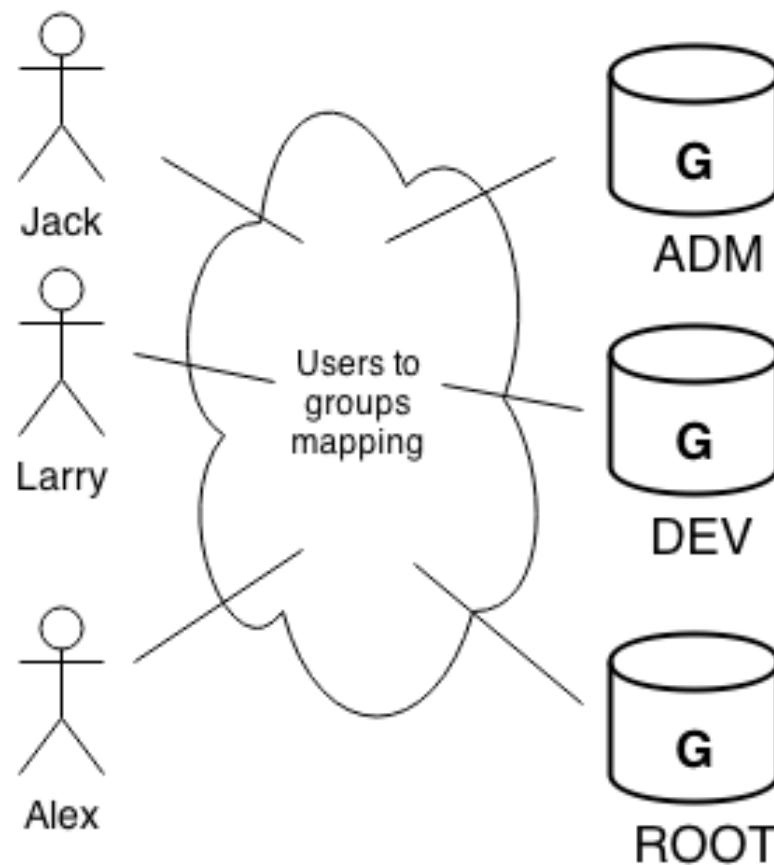
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

In Unix, permission to access system resources is managed at three levels of granularity: world, group, and owner. A group is a collection of users intended to model some functional affiliation. Each user on the system can be a member of one or more groups, and each group can have zero or more users assigned to it. Next figure shows three users that are assigned to all three groups.



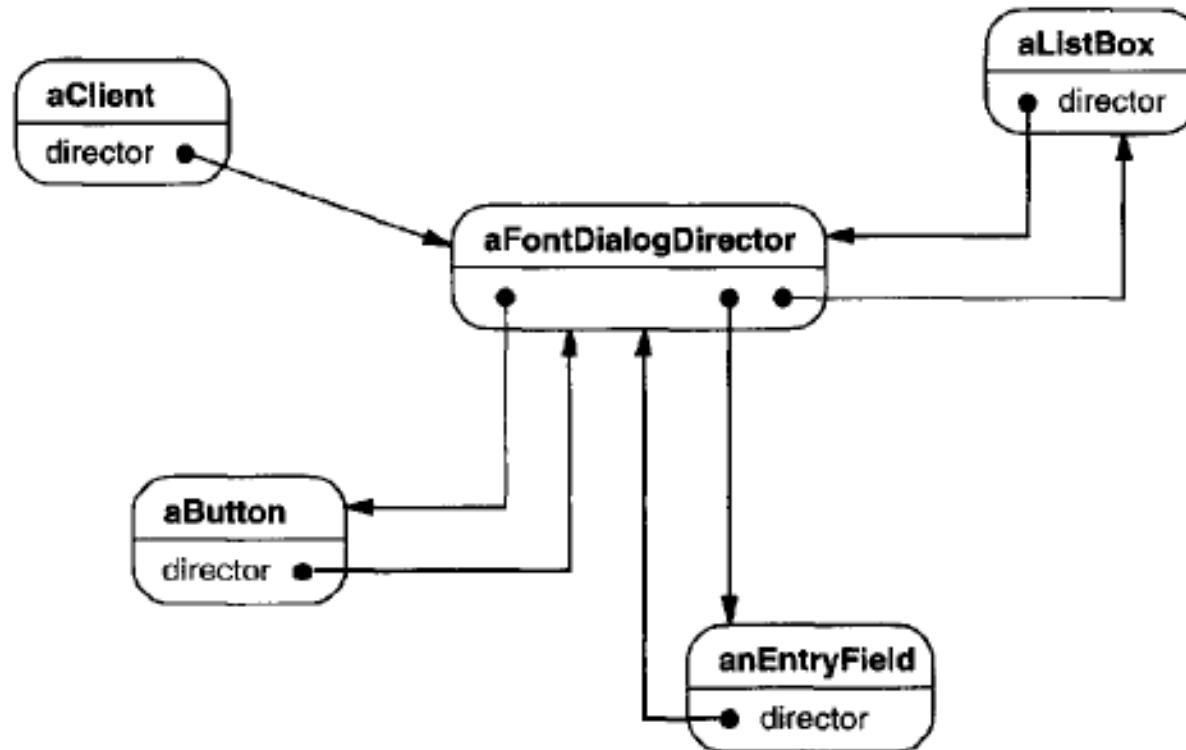
If we were to model this in software, we could decide to have User objects coupled to Group objects, and Group objects coupled to User objects. Then when changes occur, both classes and all their instances would be affected.

An alternate approach would be to introduce "an additional level of indirection" - take the mapping of users to groups and groups to users, and make it an abstraction unto itself. This offers several advantages: Users and Groups are decoupled from one another, many mappings can easily be maintained and manipulated simultaneously, and the mapping abstraction can be extended in the future by defining derived classes.

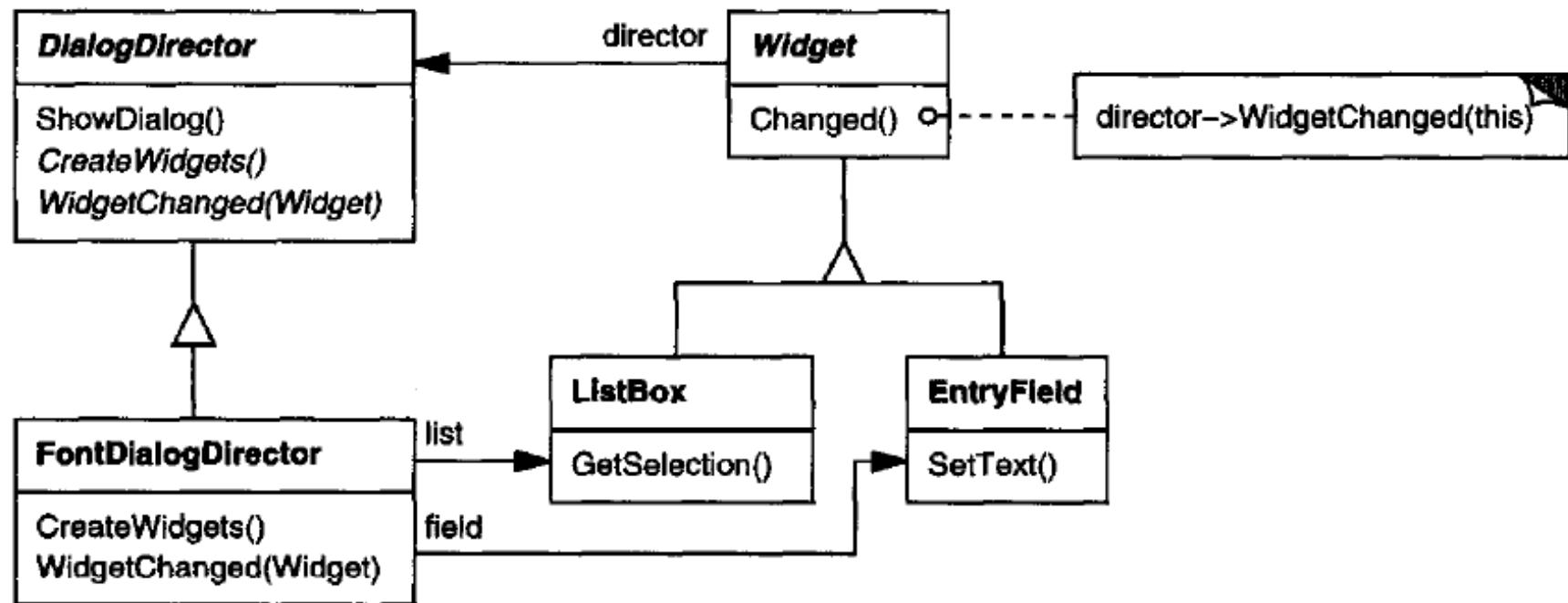


A mediator is responsible for controlling and coordinating the interactions of a group of objects. The mediator serves as an intermediary that keeps objects in the group from referring to each other explicitly. The objects only know the mediator, thereby reducing the number of interconnections.

For example, **FontDialogDirector** can be the mediator between the widgets in a dialog box. A FontDialogDirector object knows the widgets in a dialog and coordinates their interaction. It acts as a hub of communication for widgets:



Here's how the `FontDialogDirector` abstraction can be integrated into a class library:

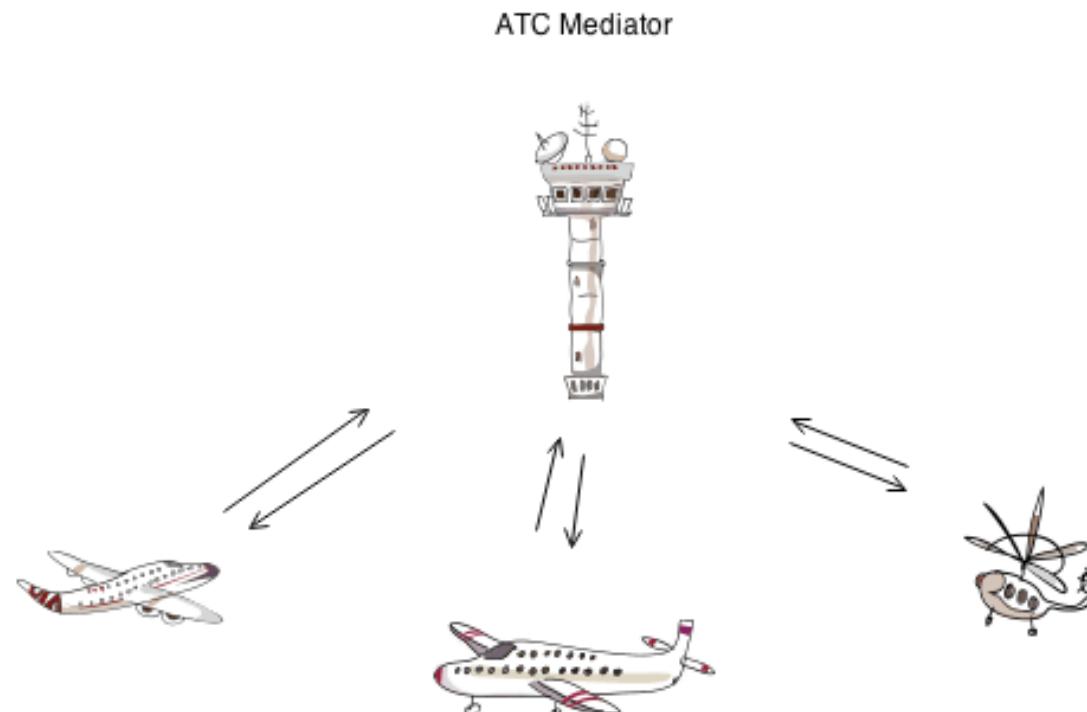


# Participants

- **Mediator** (DialogDirector)
  - defines an interface for communicating with Colleague objects.
- **ConcreteMediator** (FontDialogDirector)
  - implements cooperative behavior by coordinating Colleague objects.
  - knows and maintains its colleagues.
- **Colleague classes** (ListBox, EntryField)
  - each Colleague class knows its Mediator object.
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

# Example

The Mediator defines an object that controls how a set of objects interact. Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other. The control tower at a controlled airport demonstrates this pattern very well. The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another. The constraints on who can take off or land are enforced by the tower. It is important to note that the tower does not control the whole flight. It exists only to enforce constraints in the terminal area.



## ❖ Memento

- ▶ Object Behavioral

### Intent

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

### Also Known As

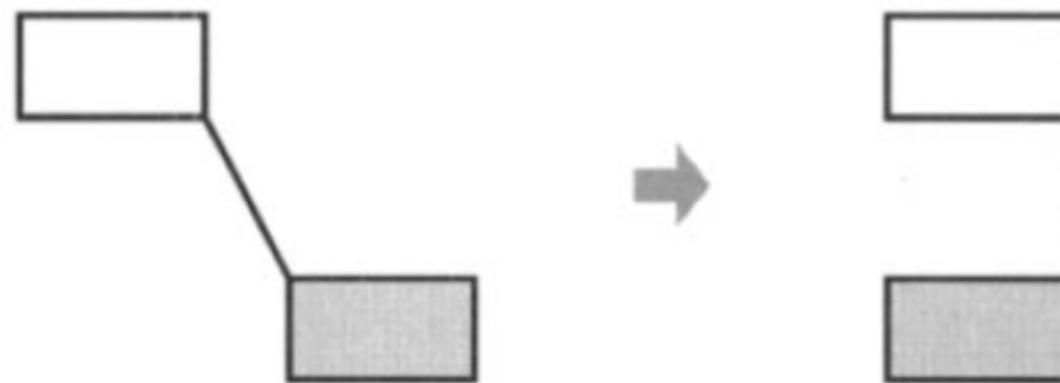
Token

Consider for example a graphical editor that supports connectivity between objects. A user can connect two rectangles with a line, and the rectangles stay connected when the user moves either of them. The editor ensures that the line stretches to maintain the connection.



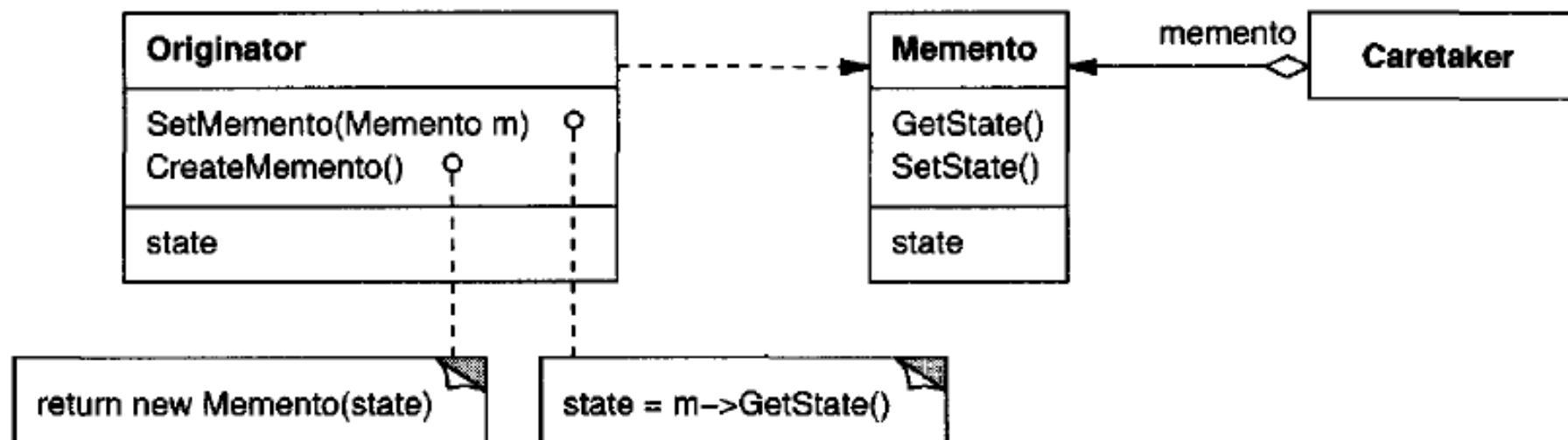
A well-known way to maintain connectivity relationships between objects is with a constraint-solving system. We can encapsulate this functionality in a **Constraint-Solver** object. ConstraintSolver records connections as they are made and generates mathematical equations that describe them. It solves these equations whenever the user makes a connection or otherwise modifies the diagram. ConstraintSolver uses the results of its calculations to rearrange the graphics so that they maintain the proper connections.

However, this does not guarantee all objects will appear where they did before. Suppose there is some slack in the connection. In that case, simply moving the rectangle back to its original location won't necessarily achieve the desired effect.



We can solve this problem with the Memento pattern. A memento is an object that stores a snapshot of the internal state of another object—the memento's originator. The undo mechanism will request a memento from the originator when it needs to checkpoint the originator's state. The originator initializes the memento with information that characterizes its current state. Only the originator can store and retrieve information from the memento—the memento is "opaque" to other objects.

In the graphical editor example just discussed, the ConstraintSolver can act as an originator.

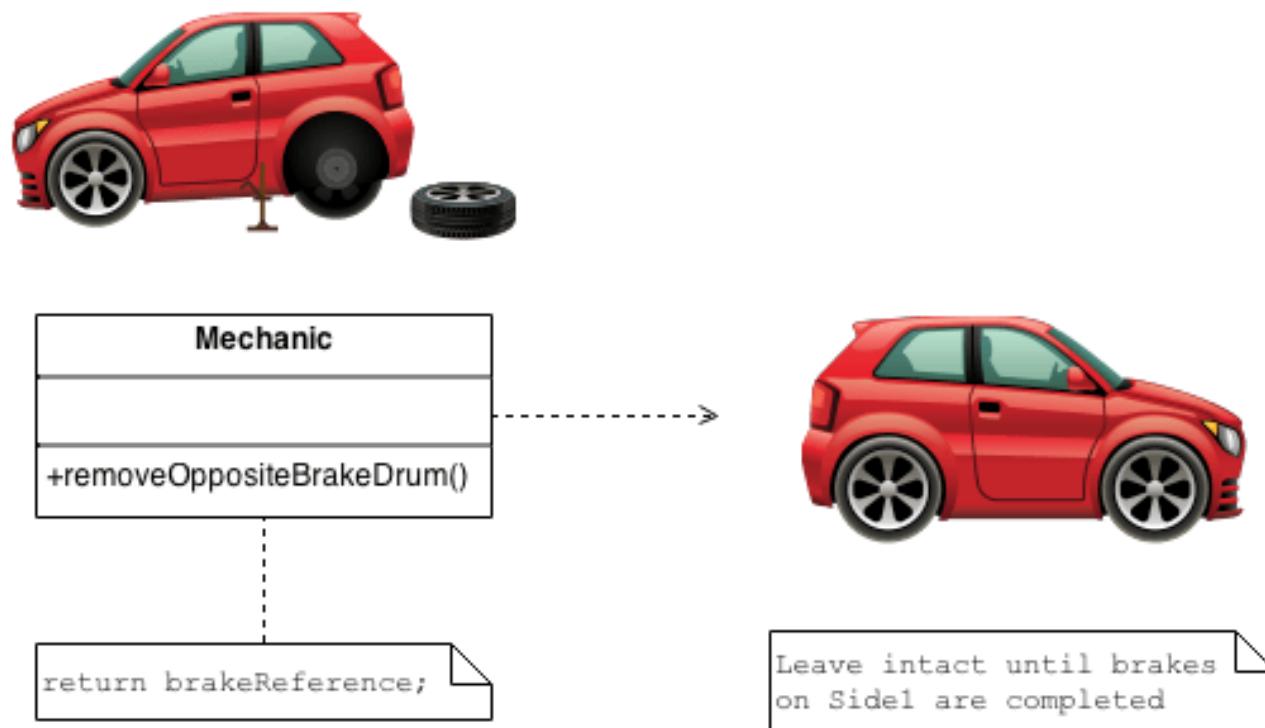


# Participants

- **Memento** (SolverState)
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protects against access by objects other than the originator. Mementos have effectively two interfaces. Caretaker sees a *narrow* interface to the Memento—it can only pass the memento to other objects. Originator, in contrast, sees a *wide* interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produced the memento would be permitted to access the memento's internal state.
- **Originator** (ConstraintSolver)
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state.
- **Caretaker** (undo mechanism)
  - is responsible for the memento's safekeeping.
  - never operates on or examines the contents of a memento.

# Example

The Memento captures and externalizes an object's internal state so that the object can later be restored to that state. This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars. The drums are removed from both sides, exposing both the right and left brakes. Only one side is disassembled and the other serves as a Memento of how the brake parts fit together. Only after the job has been completed on one side is the other side disassembled. When the second side is disassembled, the first side acts as the Memento.



## ❖ Observer

- ▶ Object Behavioral

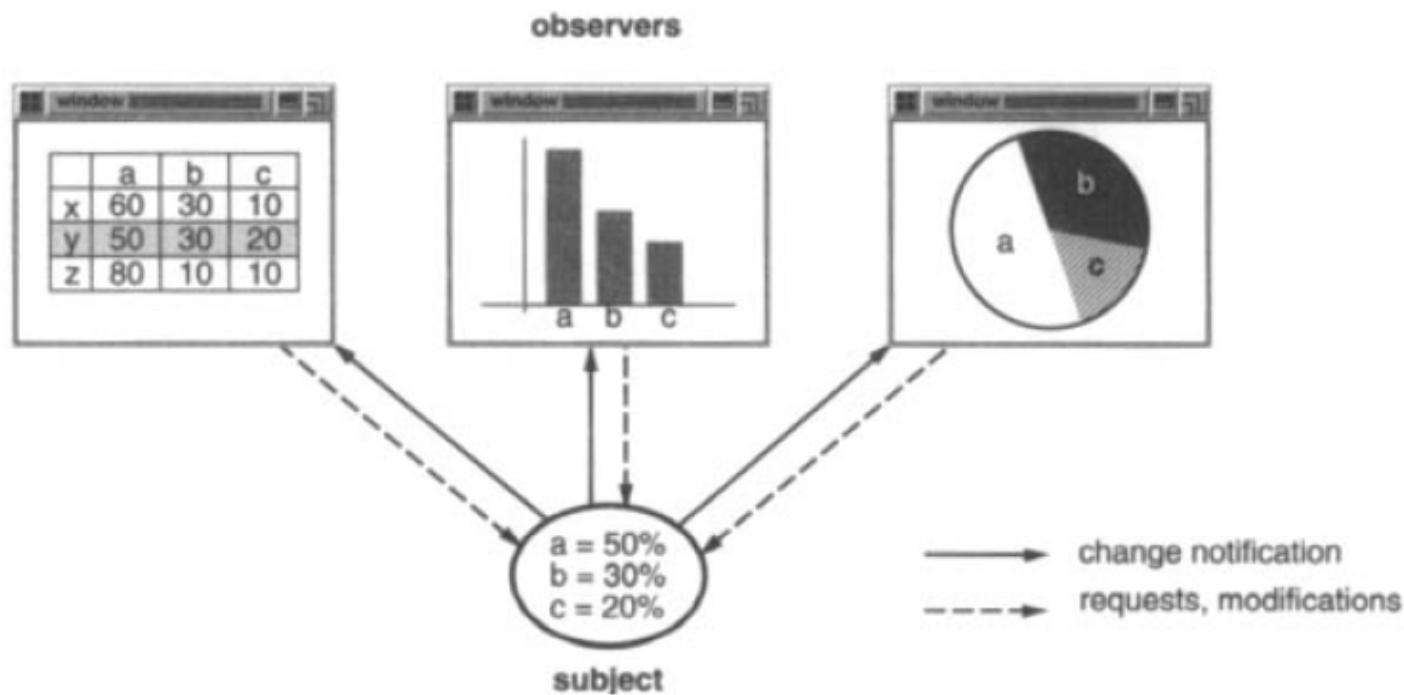
### Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

### Also Known As

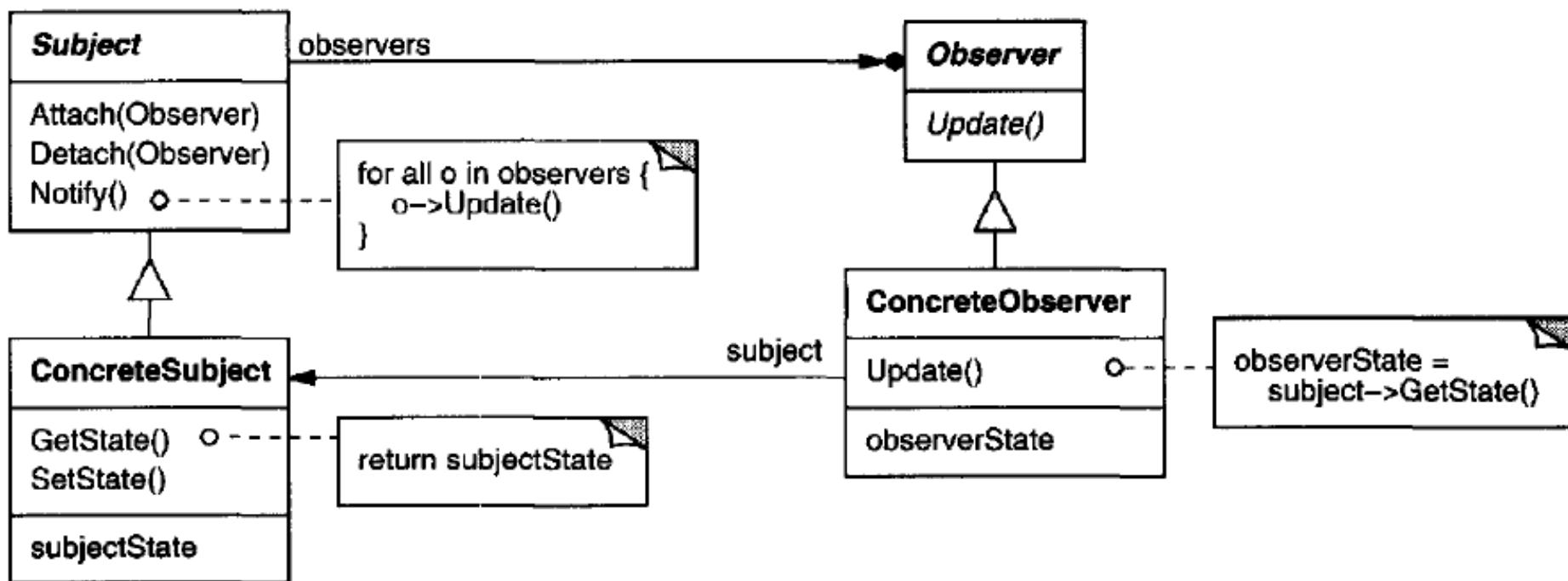
Dependents, Publish-Subscribe

Both a spreadsheet object and bar chart object can depict information in the same application data object using different presentations. The spreadsheet and the bar chart don't know about each other, thereby letting you reuse only the one you need. But they *behave* as though they do. When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.



The Observer pattern describes how to establish these relationships. The key objects in this pattern are **subject** and **observer**. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

This kind of interaction is also known as **publish-subscribe**. The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are. Any number of observers can subscribe to receive notifications.

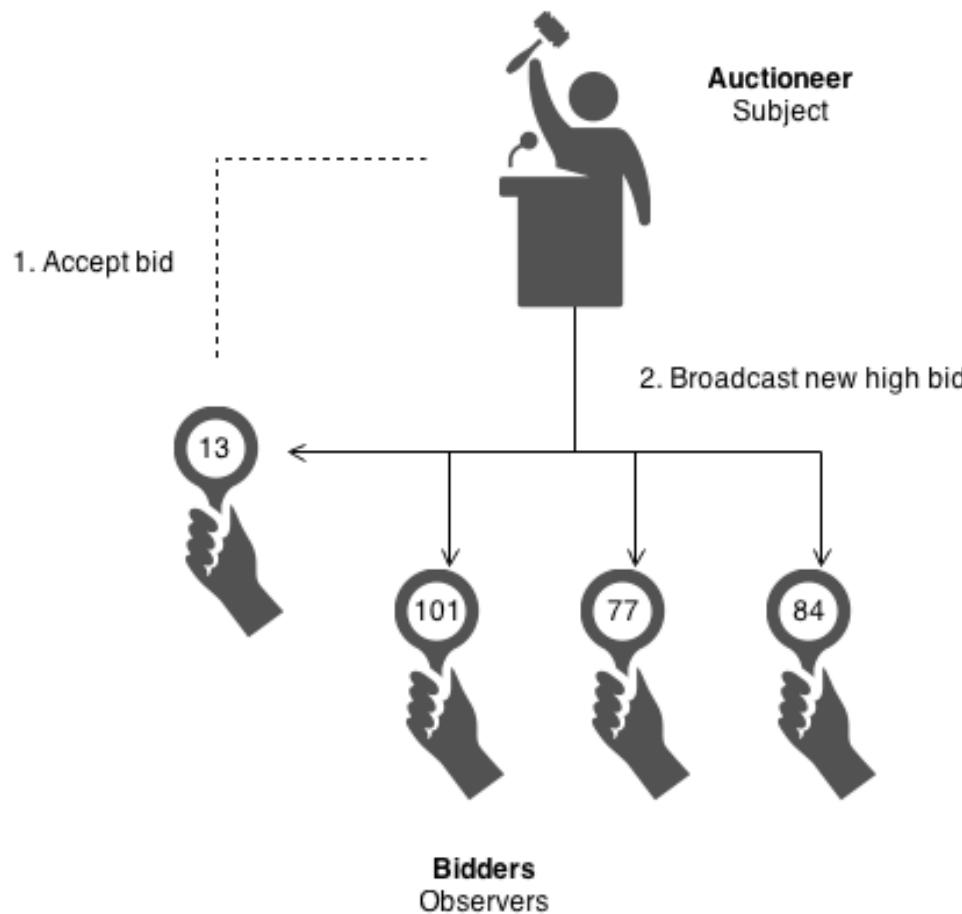


# Participants

- **Subject**
  - knows its observers. Any number of Observer objects may observe a subject.
  - provides an interface for attaching and detaching Observer objects.
- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.

# Example

The Observer defines a one-to-many relationship so that when one object changes state, the others are notified and updated automatically. Some auctions demonstrate this pattern. Each bidder possesses a numbered paddle that is used to indicate a bid. The auctioneer starts the bidding, and "observes" when a paddle is raised to accept the bid. The acceptance of the bid changes the bid price which is broadcast to all of the bidders in the form of a new bid.



## ❖ State

- ▶ Object Behavioral

### Intent

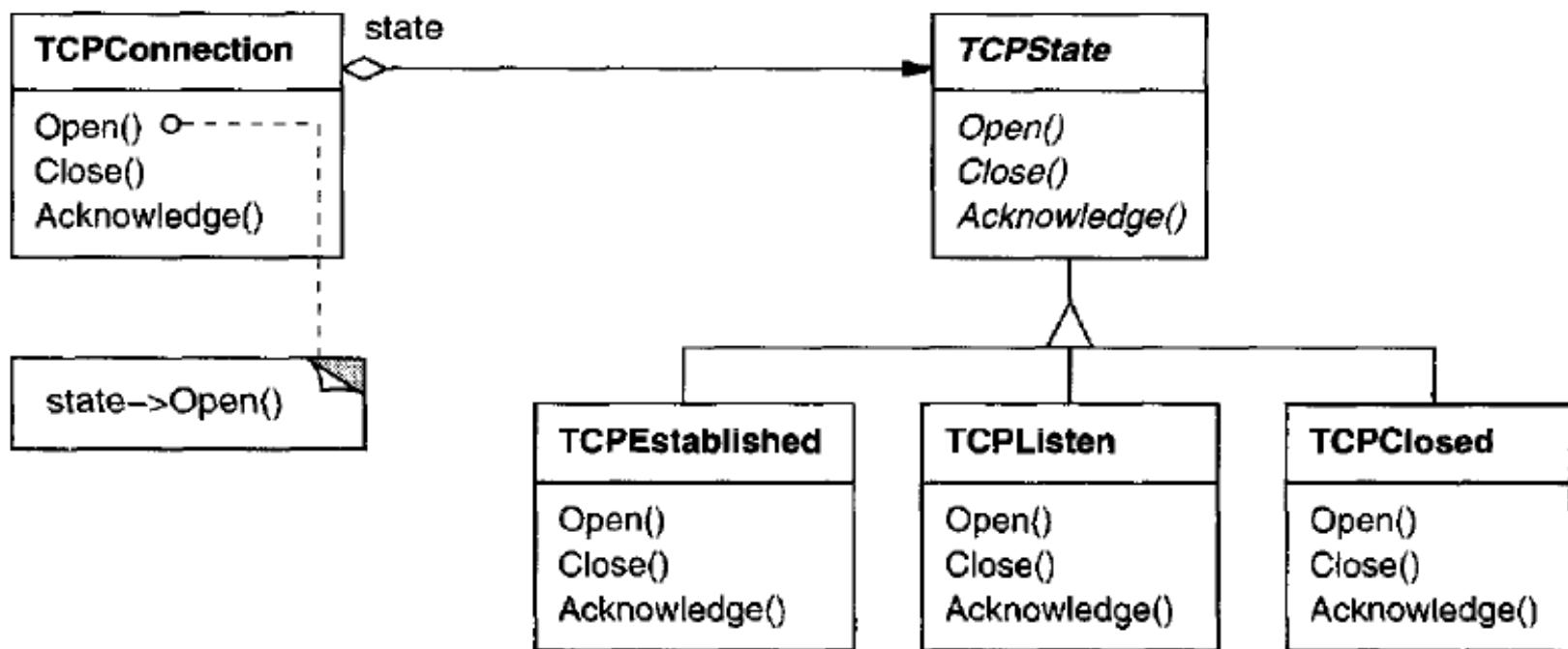
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

### Also Known As

Objects for States

Consider a class `TCPConnection` that represents a network connection. A `TCPConnection` object can be in one of several different states: `Established`, `Listening`, `Closed`. When a `TCPConnection` object receives requests from other objects, it responds differently depending on its current state. For example, the effect of an `Open` request depends on whether the connection is in its `Closed` state or its `Established` state. The `State` pattern describes how `TCPConnection` can exhibit different behavior in each state.

The key idea in this pattern is to introduce an abstract class called `TCPState` to represent the states of the network connection. The `TCPState` class declares an interface common to all classes that represent different operational states. Subclasses of `TCPState` implement state-specific behavior. For example, the classes `TCPEstablished` and `TCPClosed` implement behavior particular to the `Established` and `Closed` states of `TCPConnection`.



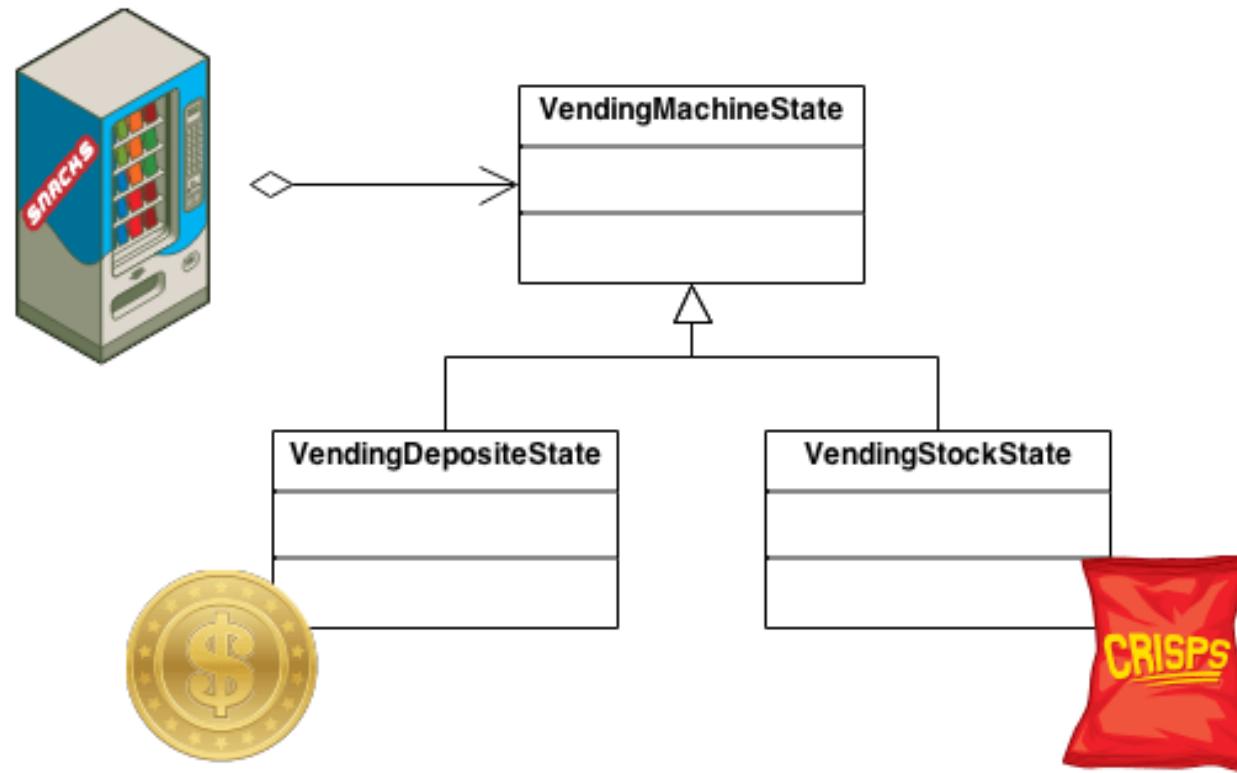
The class **TCPConnection** maintains a state object (an instance of a subclass of **TCPState**) that represents the current state of the TCP connection. The class

# Participants

- **Context** (TCPConnection)
  - defines the interface of interest to clients.
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State** (TCPState)
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **ConcreteState subclasses** (TCPEstablished, TCPListen, TCPClosed)
  - each subclass implements a behavior associated with a state of the Context.

# Example

The State pattern allows an object to change its behavior when its internal state changes. This pattern can be observed in a vending machine. Vending machines have states based on the inventory, amount of currency deposited, the ability to make change, the item selected, etc. When currency is deposited and a selection is made, a vending machine will either deliver a product and no change, deliver a product and change, deliver no product due to insufficient currency on deposit, or deliver no product due to inventory depletion.



## ❖ **Strategy**

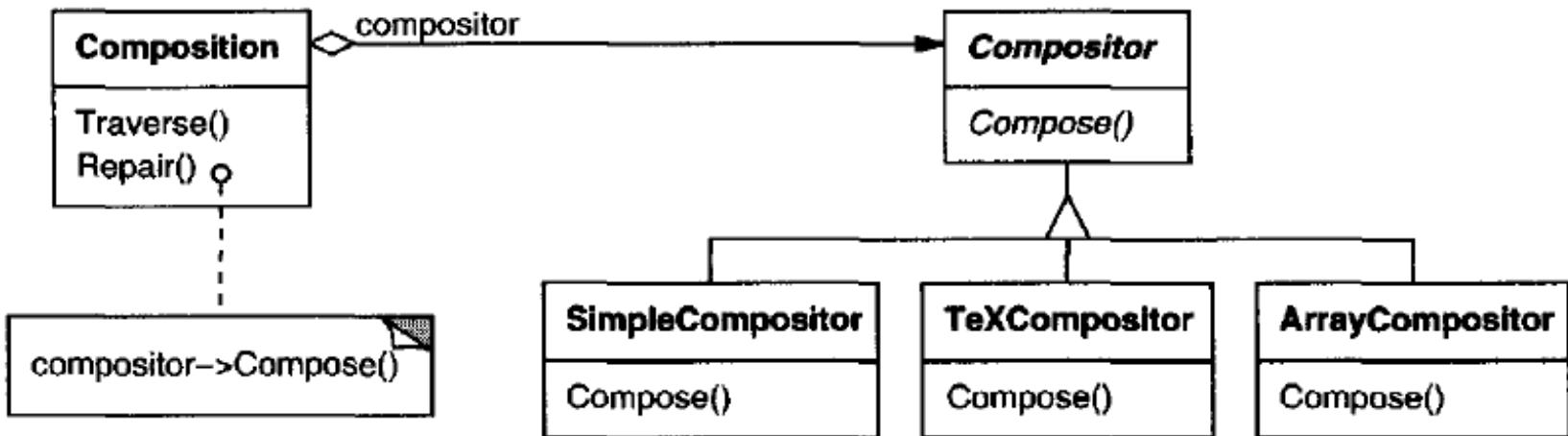
- ▶ Object Behavioral

### **Intent**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

### **Also Known As**

Policy



Suppose a **Composition** class is responsible for maintaining and updating the linebreaks of text displayed in a text viewer. Linebreaking strategies aren't implemented by the class **Composition**. Instead, they are implemented separately by subclasses of the abstract **Compositor** class. **Compositor** subclasses implement different strategies:

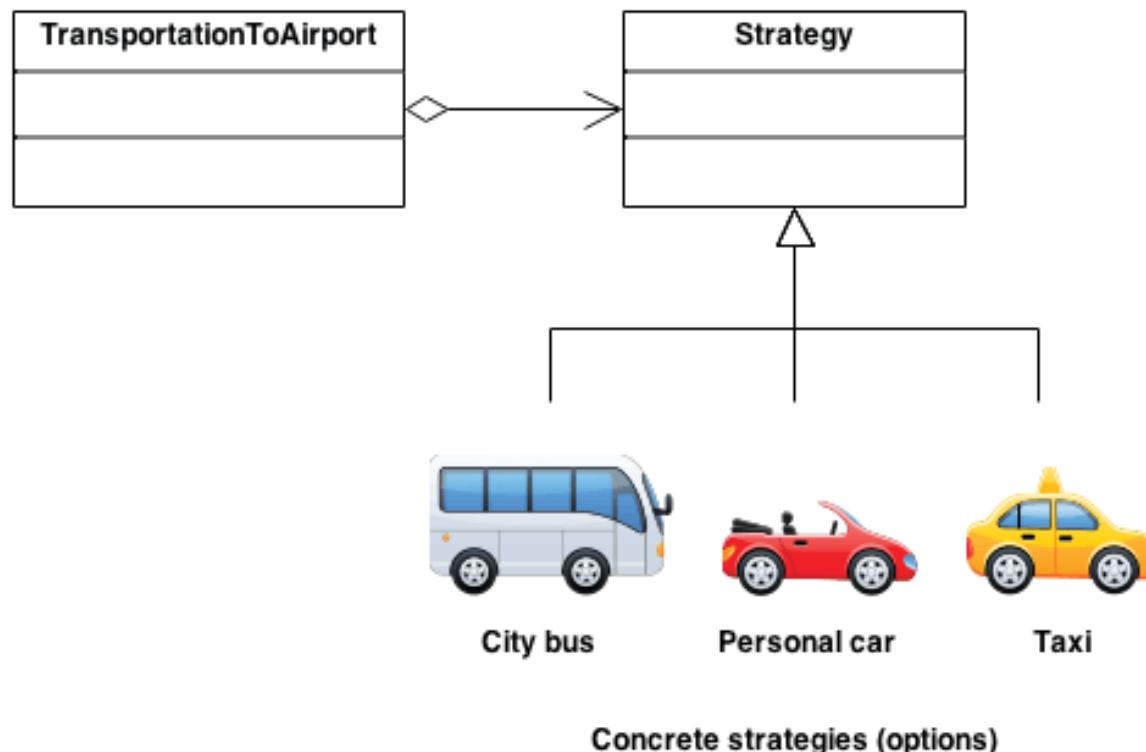
- **SimpleComposer** implements a simple strategy that determines linebreaks one at a time.
- **TeXComposer** implements the TeX algorithm for finding linebreaks. This strategy tries to optimize linebreaks globally, that is, one paragraph at a time.
- **ArrayComposer** implements a strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.

# Participants

- **Strategy** (Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy** (SimpleComposer, TeXComposer, ArrayComposer)
  - implements the algorithm using the Strategy interface.
- **Context** (Composition)
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface that lets Strategy access its data.

# Example

A Strategy defines a set of algorithms that can be used interchangeably. Modes of transportation to an airport is an example of a Strategy. Several options exist such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. For some airports, subways and helicopters are also available as a mode of transportation to the airport. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must choose the Strategy based on trade-offs between cost, convenience, and time.



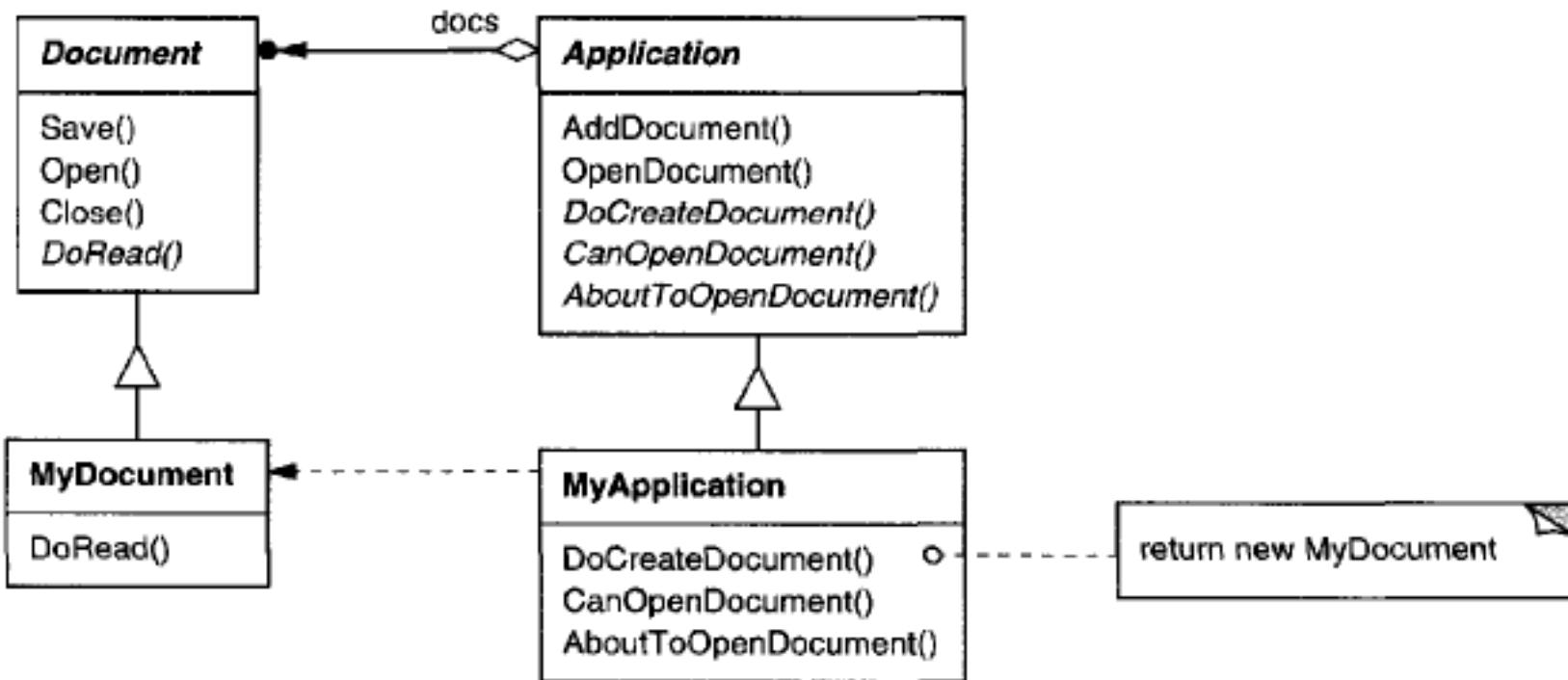
## ❖ **Template Method**

- ▶ **Class Behavioral**

### **Intent**

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Applications built with the framework can subclass Application and Document to suit specific needs. For example, a drawing application defines DrawApplication and DrawDocument subclasses; a spreadsheet application defines SpreadsheetApplication and SpreadsheetDocument subclasses.

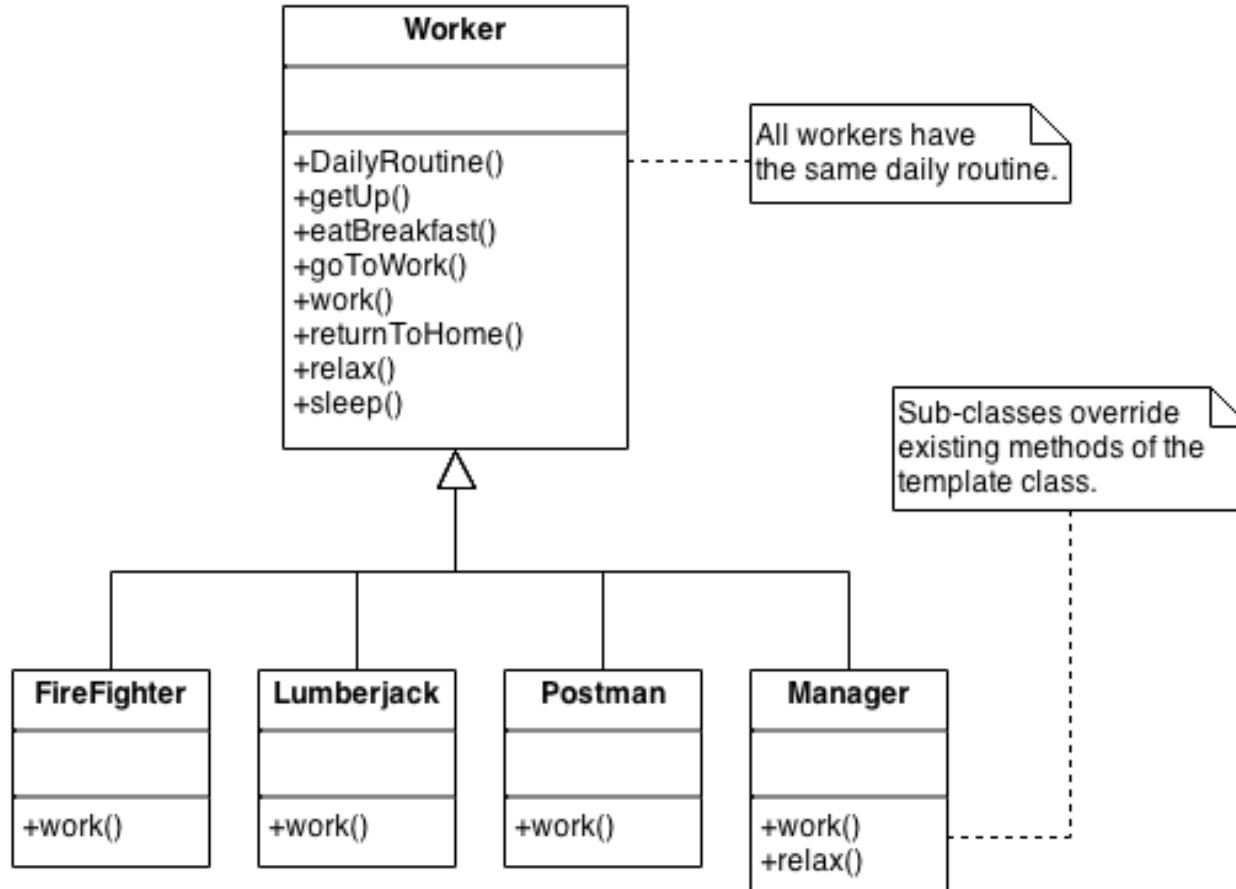


The abstract Application class defines the algorithm for opening and reading a document in its `OpenDocument` operation:

# Participants

- **AbstractClass** (Application)
  - defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass** (MyApplication)
  - implements the primitive operations to carry out subclass-specific steps of the algorithm.

Another example: daily routine of a worker.

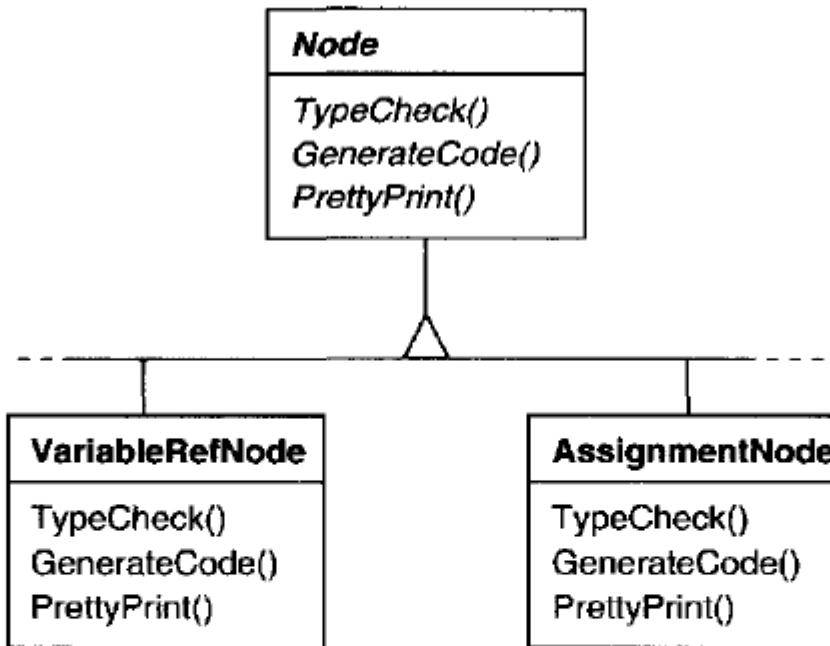


## ❖ **Visitor**

- ▶ **Object Behavioral**

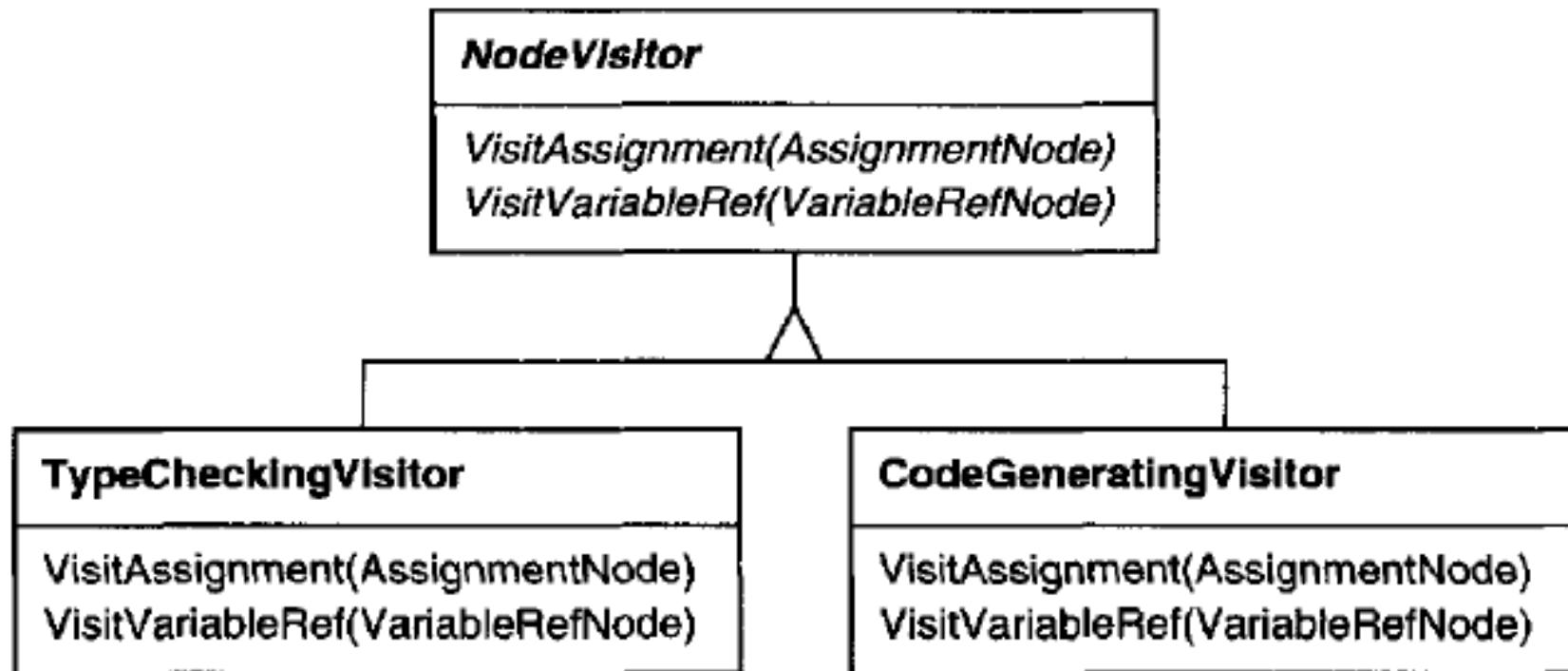
### **Intent**

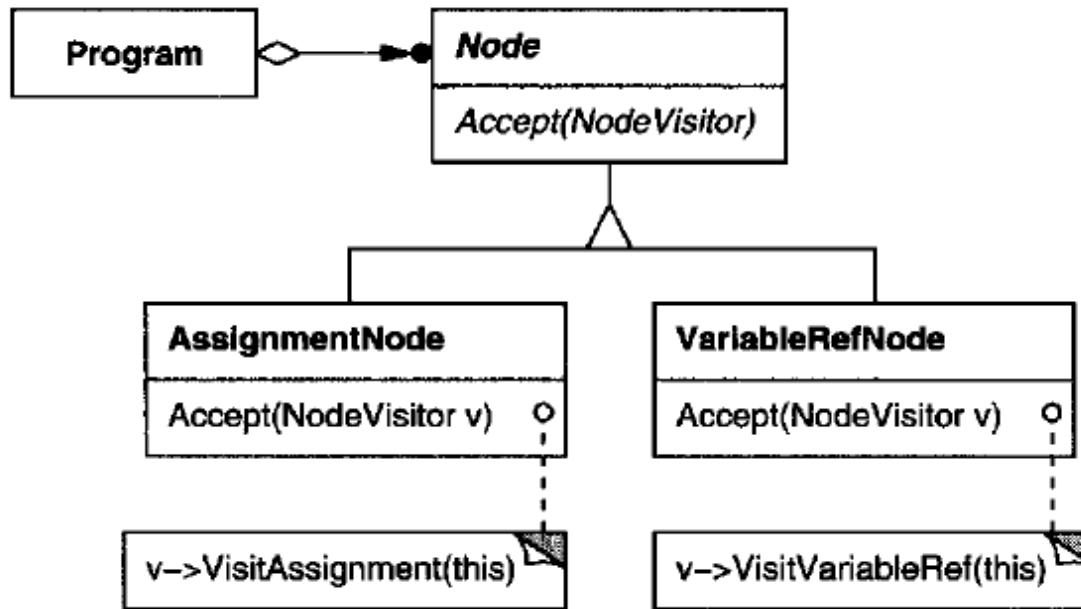
Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.



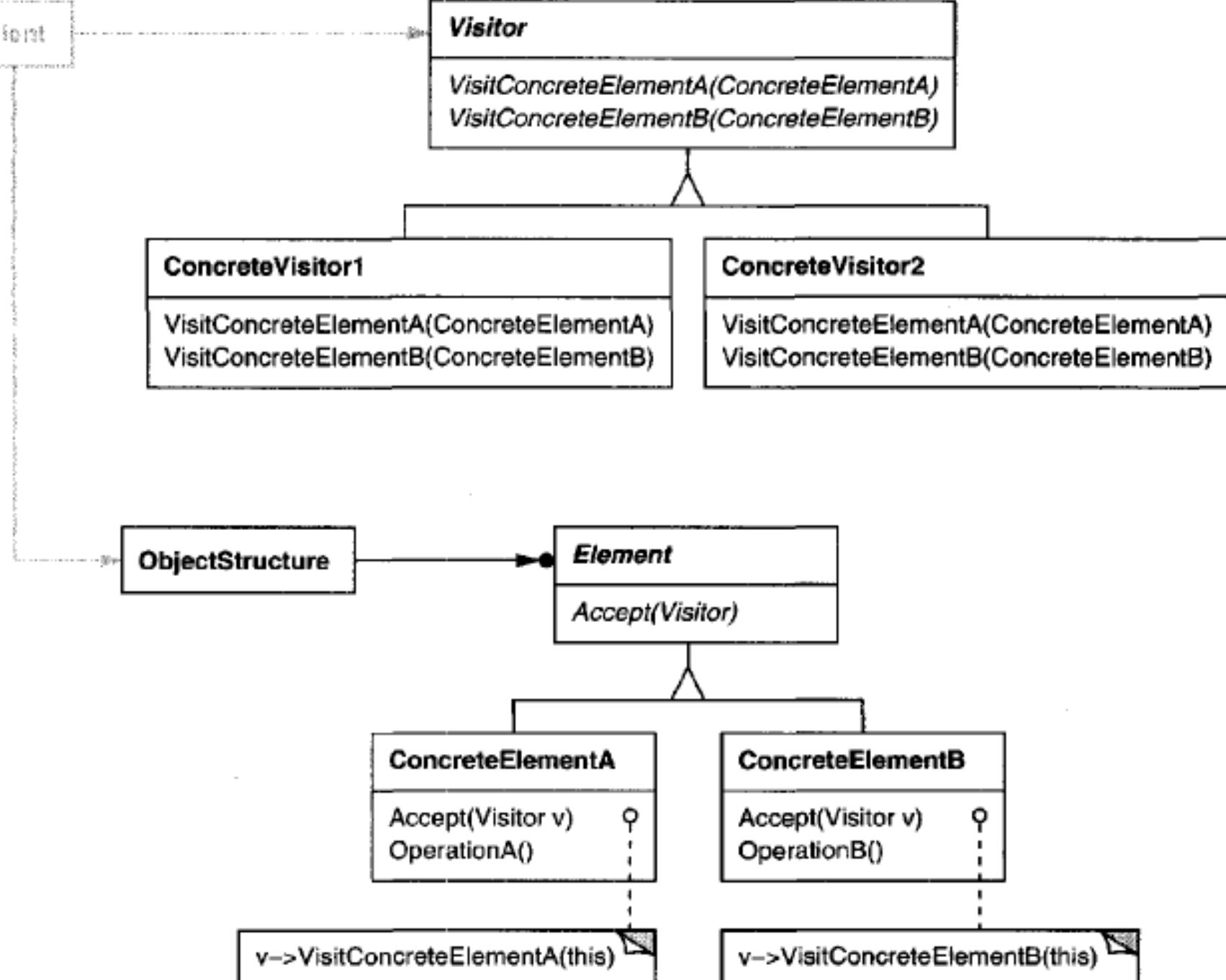
This diagram shows part of the **Node** class hierarchy. The problem here is that distributing all these operations across the various node classes leads to a system that's hard to understand, maintain, and change. It will be confusing to have type-checking code mixed with pretty-printing code or flow analysis code. Moreover, adding a new operation usually requires recompiling all of these classes. It would be better if each new operation could be added separately, and the node classes were independent of the operations that apply to them.

We can have both by packaging related operations from each class in a separate object, called a **visitor**, and passing it to elements of the abstract syntax tree as it's traversed. When an element “accepts” the visitor, it sends a request to the visitor that encodes the element's class. It also includes the element as an argument. The visitor will then execute the operation for that element—the operation that used to be in the class of the element.





With the Visitor pattern, you define two class hierarchies: one for the elements being operated on (the Node hierarchy) and one for the visitors that define operations on the elements (the NodeVisitor hierarchy). You create a new operation by adding a new subclass to the visitor class hierarchy. As long as the grammar that the compiler accepts doesn't change (that is, we don't have to add new Node subclasses), we can add new functionality simply by defining new NodeVisitor subclasses.

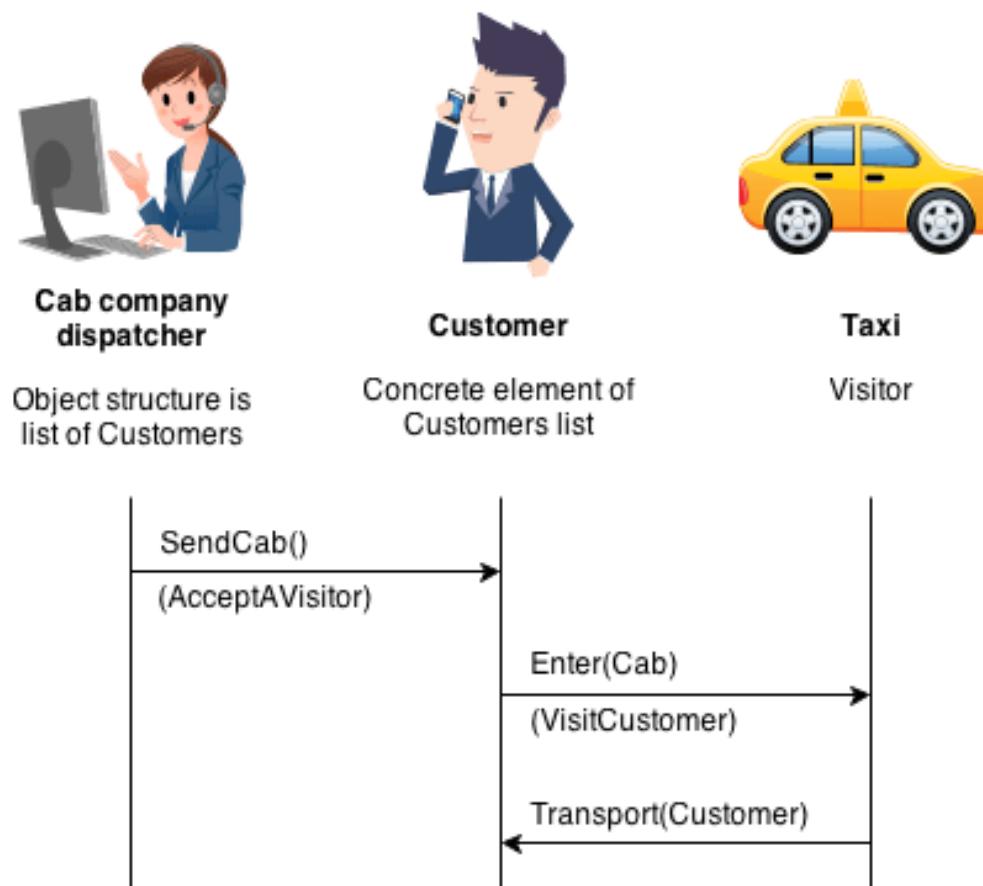


# Participants

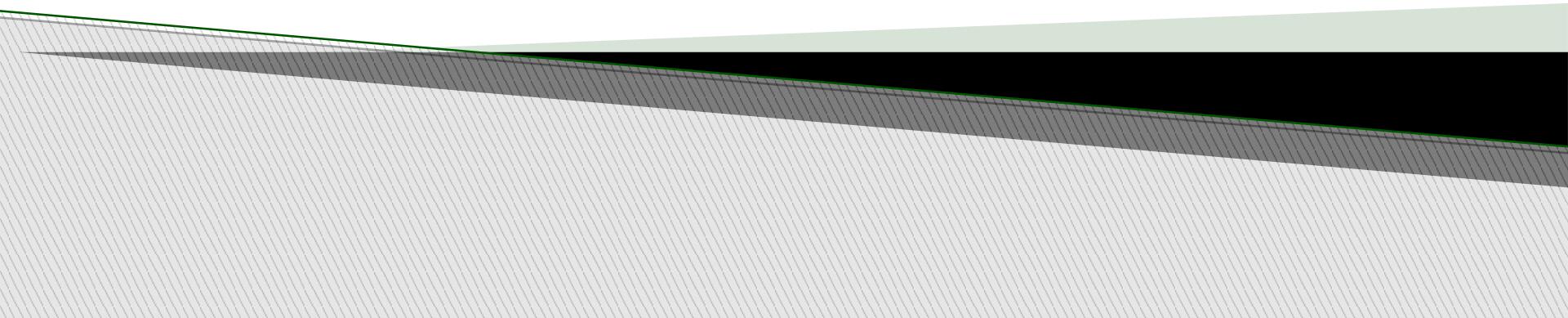
- **Visitor** (NodeVisitor)
  - declares a Visit operation for each class of ConcreteElement in the object structure. The operation's name and signature identifies the class that sends the Visit request to the visitor. That lets the visitor determine the concrete class of the element being visited. Then the visitor can access the element directly through its particular interface.
- **ConcreteVisitor** (TypeCheckingVisitor)
  - implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This state often accumulates results during the traversal of the structure.
- **Element** (Node)
  - defines an Accept operation that takes a visitor as an argument.
- **ConcreteElement** (AssignmentNode, VariableRefNode)
  - implements an Accept operation that takes a visitor as an argument.
- **ObjectStructure** (Program)
  - can enumerate its elements.
  - may provide a high-level interface to allow the visitor to visit its elements.
  - may either be a composite (see Composite (163)) or a collection such as a list or a set.

# Example

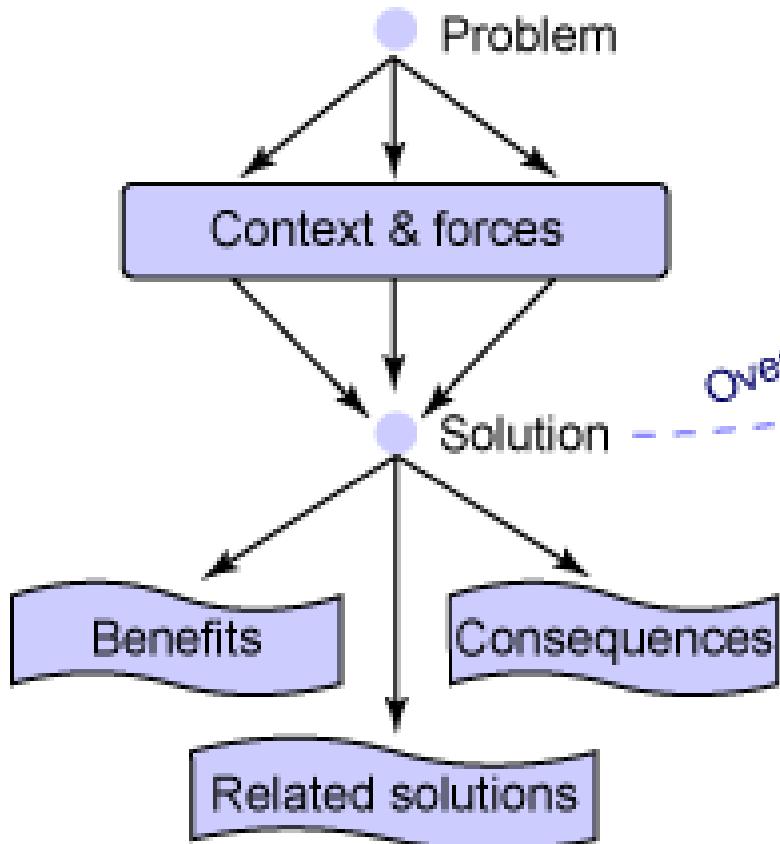
The Visitor pattern represents an operation to be performed on the elements of an object structure without changing the classes on which it operates. This pattern can be observed in the operation of a taxi company. When a person calls a taxi company (accepting a visitor), the company dispatches a cab to the customer. Upon entering the taxi the customer, or Visitor, is no longer in control of his or her own transportation, the taxi (driver) is.



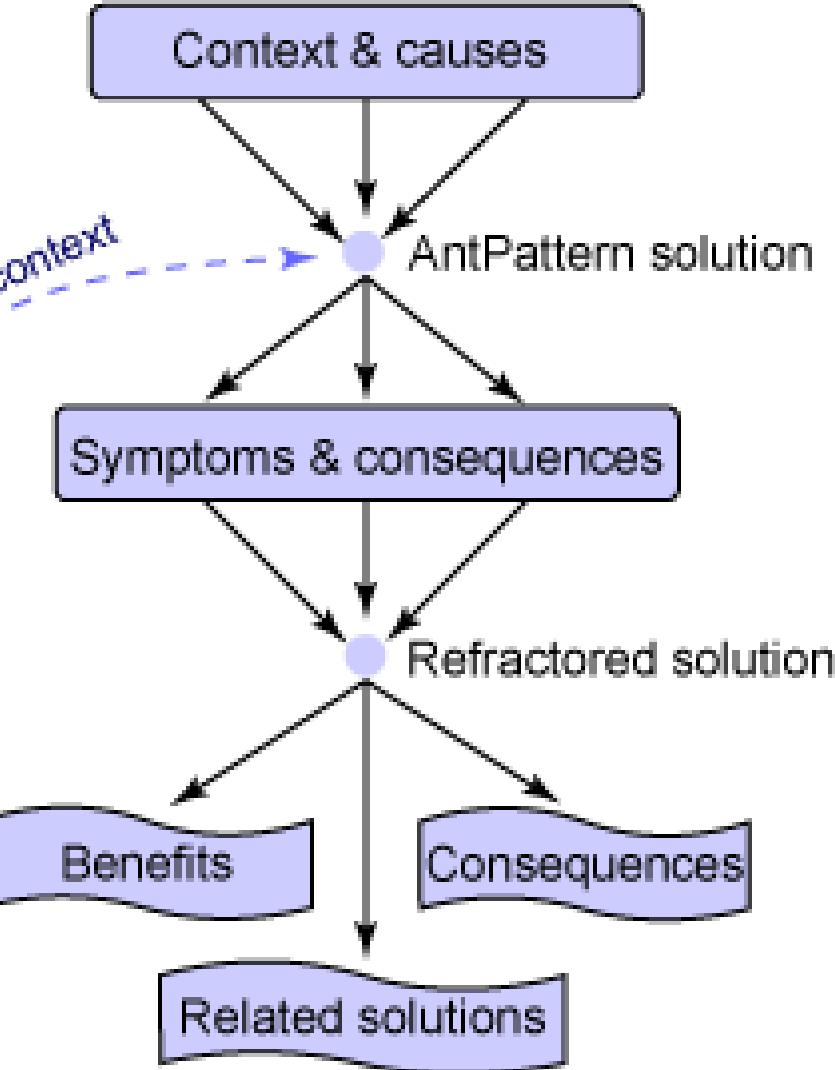
# **Concept of Anti-patterns**



## Design patterns



## AntiPatterns

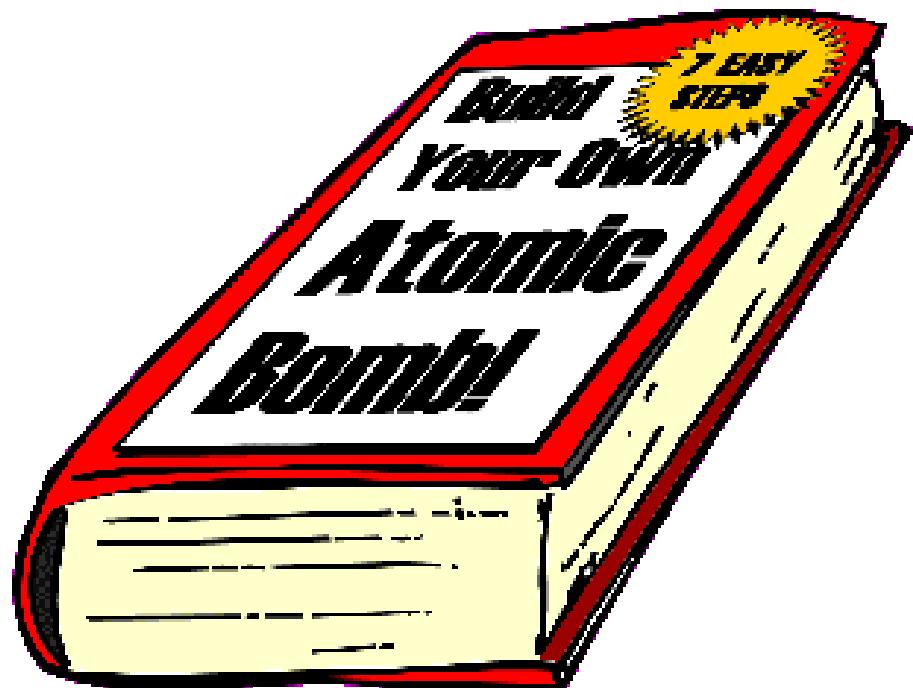


# AntiPattern Examples



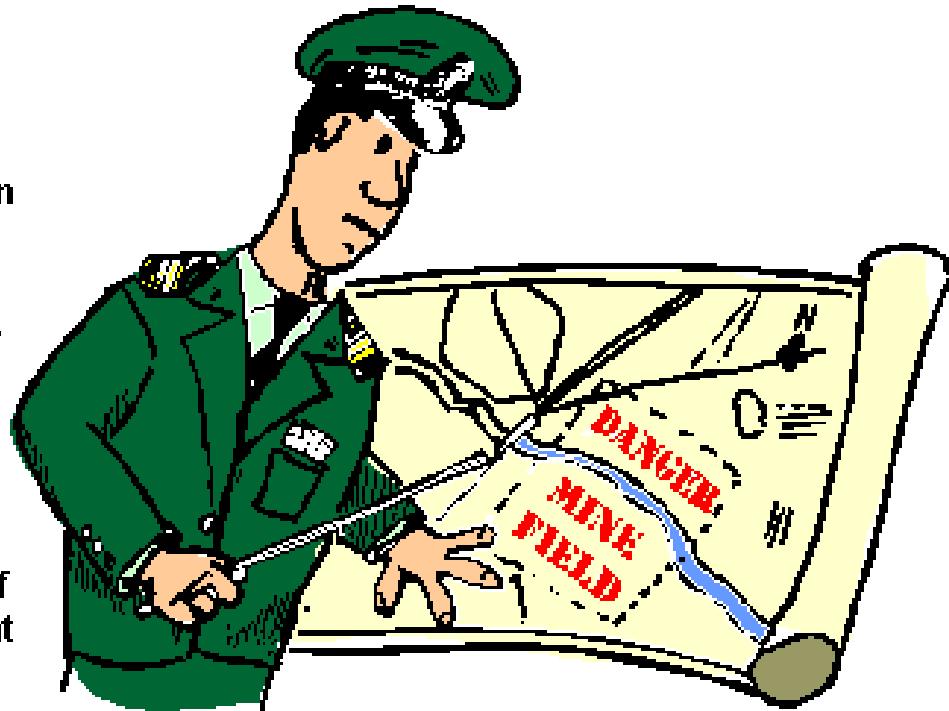
# What is an AntiPattern (and why should I care)?

- AntiPatterns are *Negative* Solutions that present more problems than they address
- AntiPatterns are a natural extension to design patterns
- AntiPatterns bridge the gap between architectural concepts and real-world implementations.
- Understanding AntiPatterns provides the knowledge to prevent or recover from them



# Why Study AntiPatterns?

- AntiPatterns are a method of efficiently mapping a general situation to a specific class of solutions.
- AntiPatterns provide real world experience in recognizing recurring problems in the software industry, providing a detailed remedy for the most common predicaments.
- AntiPatterns provide a common vocabulary for identifying problems and discussing solutions.
- AntiPattern support the holistic resolution of conflicts utilizing organizational resources at several levels, where possible.
- AntiPatterns provide stress release in the form of shared misery for the most common pitfalls in the software industry.



- ▶ AntiPatterns, like their design pattern counterparts, define an industry vocabulary for the common defective processes and implementations within organizations
- ▶ A higher-level vocabulary simplifies communication between software practitioners and enables concise description of higher-level concepts
- ▶ Describes a commonly occurring solution to a problem that generates decidedly negative consequences
- ▶ The AntiPattern may be the result of a manager or developer not knowing any better, not having sufficient knowledge or experience in solving a particular type of problem, or having applied a perfectly good pattern in the wrong context

- ▶ Recognizing recurring problems in the software industry
- ▶ Provide a detailed remedy for the most common predicaments
- ▶ Highlight the most common problems that face the software industry and provide the tools to enable you to recognize these problems and to determine their underlying causes
- ▶ Implementing productive solutions
- ▶ Improve the developing of applications, the designing of software systems
- ▶ Effective management of software projects

## □ Software Development AntiPatterns



- ▶ A key goal of development AntiPatterns is to **describe useful forms of software refactoring**
- ▶ Software refactoring is a form of code modification, used to improve the **software structure** in support of subsequent extension and **long-term maintenance**

## □ Software Architecture AntiPatterns



- ▶ Architecture AntiPatterns focus on the **system-level and enterprise-level structure of applications and components**



## □ Project Management AntiPatterns

- ▶ In the modern engineering profession, more than half of the job involves human communication and resolving people issues
- ▶ The management AntiPatterns identify some of the key scenarios in which **these issues are destructive to software processes**