LECTURE NOTES

ON

DESIGN PATTERNS(15A05603)

III B.TECH II SEMESTER

(JNTUA-R15)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VEMU INSTITUTE OF TECHNOLOGY:: P.KOTHAKOTA

Chittoor-Tirupati National Highway, P.Kothakota, Near Pakala, Chittoor (Dt.), AP - 517112

(Approved by AICTE, New Delhi Affiliated to JNTUA Ananthapuramu. ISO 9001:2015 Certified Institute)

UNIT -I

Introduction to Design Patterns

1.1 Design Pattern Definition?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

In general, a pattern has four essential elements:

1. **Pattern Name:**
   - It is a handle we can use to illustrate a design problem, its solutions, and consequences in a word or two.
   - Naming a pattern instantly increases our design vocabulary. It lets us design at a higher level of abstraction.
   - It makes it easier to think about designs and to communicate them and their trade-offs to others.
   - Finding good names has been one of the hardest parts of developing our catalog.
2. **Problem:**
   - It describes when to apply the pattern.
   - It explains the problem and its context.
   - It might describe specific design problems such as how to represent algorithms as objects.
   - It might describe class or object structures that are indicative of an inflexible design.
   - Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
3. **Solution:**
   - It describes the elements that make up the design, their relationships, responsibilities, and collaborations.

- 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.
- Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements solves it.

4. Consequences:

- They are the results and trade-offs of applying the pattern.
- Though consequences are often understood when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.
- The consequences for software often concern space and time trade-offs.
- They may address language and implementation issues as well.
- Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability.
- Listing these consequences explicitly helps you understand and evaluate them.

Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block.

The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

- A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.
- The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.
- Each design pattern focuses on a particular object-oriented design problem or issue.
- It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.
- Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages like Smalltalk and C++ rather than procedural languages(Pascal, C, Ada) or more dynamic object-oriented languages (CLOS, Dylan, Self).

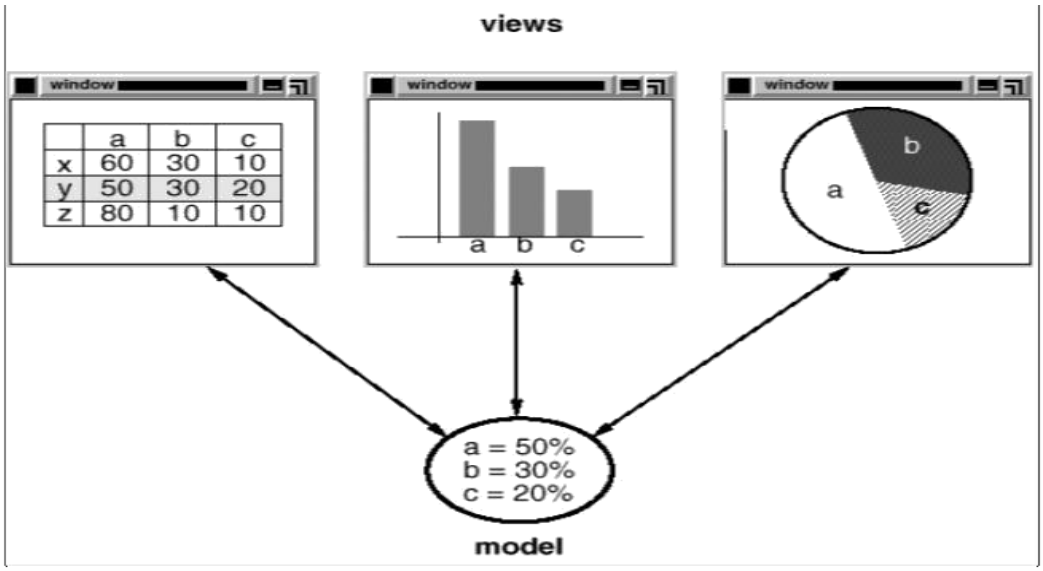
- The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily.
- If we assumed procedural languages, we might have included design patterns called "Inheritance", "Encapsulation," and "Polymorphism."

1.2 Design Patterns in Smalltalk MVC

The Model/View/Controller (MVC) triad of classes is used to build user interfaces in Smalltalk-80.MVC consists of three kinds of objects. The **Model** is the application object, the **View** is its screen presentation, and the **Controller** defines the way the user interface reacts to user input. Before MVC, user interface designs tended to combine these objects together. MVC decouples them to increase flexibility and reuse.

- MVC decouples views and models by establishing a **subscribe/notify** protocol between them.
- A view must ensure that its appearance reflects the state of the model.
- Whenever the model's data changes, the model notifies views that depend on it.
- In response, each view gets an opportunity to update itself.
- This approach lets you attach multiple views to a model to provide different presentations.
- We can also create new views for a model without rewriting it.

The following diagram shows a model and three views.



- The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways.
- The model communicates with its views when its values change, and the views communicate with the model to access these values.

This example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others. This more general design is described by the **Observer** design pattern.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. MVC supports nested views with the **CompositeView** class, a subclass of **View**.

But the design is applicable to a more general problem, which occurs whenever we want to group objects and treat thegroup like an individual object. This more general design is described by the **Composite** design pattern.

MVC also lets you change the way a view responds to user input without changing its visual presentation. You might want to change the way it responds to the keyboard, for example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a **Controller** object.

A view uses an instance of a **Controller** subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller.

The View-Controller relationship is an example of the **Strategy** design pattern. A **Strategy** is an object that represents an algorithm.

MVC uses other design patterns, such as **Factory Method** to specify the default controller class for a view and **Decorator** to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite, and Strategy design patterns.

1.3 Describing Design Patterns

Describing the design patterns in graphical notations, simply capture the end product of the design process as relationships between classes and objects.

We describe design patterns using a consistent format. Each pattern is divided into sections according to the following template. The template lends a uniform structure to the information, making design patterns easier to learn, compare, and use.

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary.

Intent

This describes, what does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

Also Known As

Other well-known names for the pattern, if any.

Motivation

A scenario that illustrates a design problem and how the class and object structures in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability

What are the situations in which the design pattern can be applied? What are examples of poor designs that the pattern can address? How can you recognize these situations?

Structure

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT). We also use interaction diagrams to illustrate sequences of requests and collaborations between objects.

Participants

The classes and/or objects participating in the design pattern and their responsibilities.

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

How does the pattern support its objectives? What are the trade-offs and results of using the pattern? What aspect of system structure does it let you vary independently?

Implementation

What pitfalls, hints, or techniques should you be aware of when implementing the pattern? Are there language-specific issues?

Sample Code

Code fragments that illustrate how you might implement the pattern in C++ or Smalltalk.

Known Uses

Examples of the pattern found in real systems. We include at least two examples from different domains.

Related Patterns

What design patterns are closely related to this one? What are the important differences? With which other patterns should this one be used?

1.4 The Catalog of Design Patterns

The catalog contains 23 design patterns. Their names and intents are listed below.

1. **Abstract Factory**

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

2. **Adapter**

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

3. **Bridge**

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

4. **Builder**

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

5. **Chain of Responsibility**

   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.

6. **Command**

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

7. **Composite**

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

8. **Decorator**

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

9. **Facade**

   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.

10. **Factory Method**

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

11. **Flyweight**

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

12. **Interpreter**

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

13. **Iterator**

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

14. **Mediator**

    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.

15. **Memento**

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

16. **Observer**

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

17. **Prototype**

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

18. **Proxy**

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

19. **Singleton**

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

20. **State**

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

21. **Strategy**

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

22. **Template Method**

    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.

23. **Visitor**

    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.

1.5 Organizing the Catalog

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them. With this classification of design patterns we can easily identify related patterns. The classification helps you learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

We classify design patterns by two criteria:

1) P**urpose:** 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.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Table 1.1:Design pattern space

2) **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.

So the 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.

There are other ways to organize the patterns.

a) Some patterns are often used together: **Composite** is often used with **Iterator** or **Visitor**.
b) Some patterns are alternatives: **Prototype** is often an alternative to Abstract **Factory**.
c) Some patterns result in similar designs even though the patterns have different intents: the structure diagrams of **Composite** and **Decorator** are similar.

Yet another way to organize design patterns is according to how they reference each other in their "Related Patterns" sections. Clearly there are many ways to organize design patterns. Having multiple ways of thinking about patterns will deepen your insight into what they do, how they compare, and when to apply them.

1.7 Selection of 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:

1) Consider how design patterns solve design problems.

Discussions about how design patterns help you find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems. Referring to these discussions can help guide your search for the right pattern.

2) Scan Intent sections.

Read through each pattern's intent to find one or more that are relevant to your problem. You can use the classification scheme to reduce your search.

3) Study how patterns interrelate.

Studying the relationships between design patterns can help you to select right pattern or group of patterns.

4) Study patterns of like purpose.

Each pattern concludes with a section that compares and contrasts with other patterns. These sections help you to identify the similarities and differences between patterns of like purpose.

5) Examine a cause of redesign.

Study the causes of redesign to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign.

6) Consider what should be variable in your design.

This approach is the opposite of focusing on the causes of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies, a theme of many design patterns.

Purpose	**Design Pattern**	**Aspect(s)That Can Vary**
Creational	AbstractFactory	families of product objects
	Builder	How a composite object gets created
	Factory Method	subclass of object that is instantiated
	Prototype	class of object that is instantiated
	Singleton	The sole instance of a class
Structural	Adapter	interface to an object
	Bridge	implementation of an object
	Composite	structure and composition of an object
	Decorator	Responsibilities of an object without subclassing
	Facade	interface to a subsystem
	Flyweight	storage costs of objects
	Proxy	How an object is accessed; its location
Behavioral	Chain of Responsibility	Object that can fulfill a request
	Command	when and how a request is fulfilled
	Interpreter	grammar and interpretation of a language
	Iterator	how an aggregate's elements are accessed,

		traversed
	Mediator	how and which objects interact with each other
	Memento	what private information is stored outside an object, and when
	Observer	number of objects that depend on another object; how the dependent objects stay up to date
	State	States of an object
	Strategy	an algorithm
	TemplateMethod	steps of an algorithm
	Visitor	operations that can be applied to object(s) without changing their class(es)

Table1.2: Design aspects that design patterns let you vary

1.8 Use of a Design Pattern

Once you've picked a design pattern, how do you use it? Here's a step-by-step approach to applying a design pattern effectively:

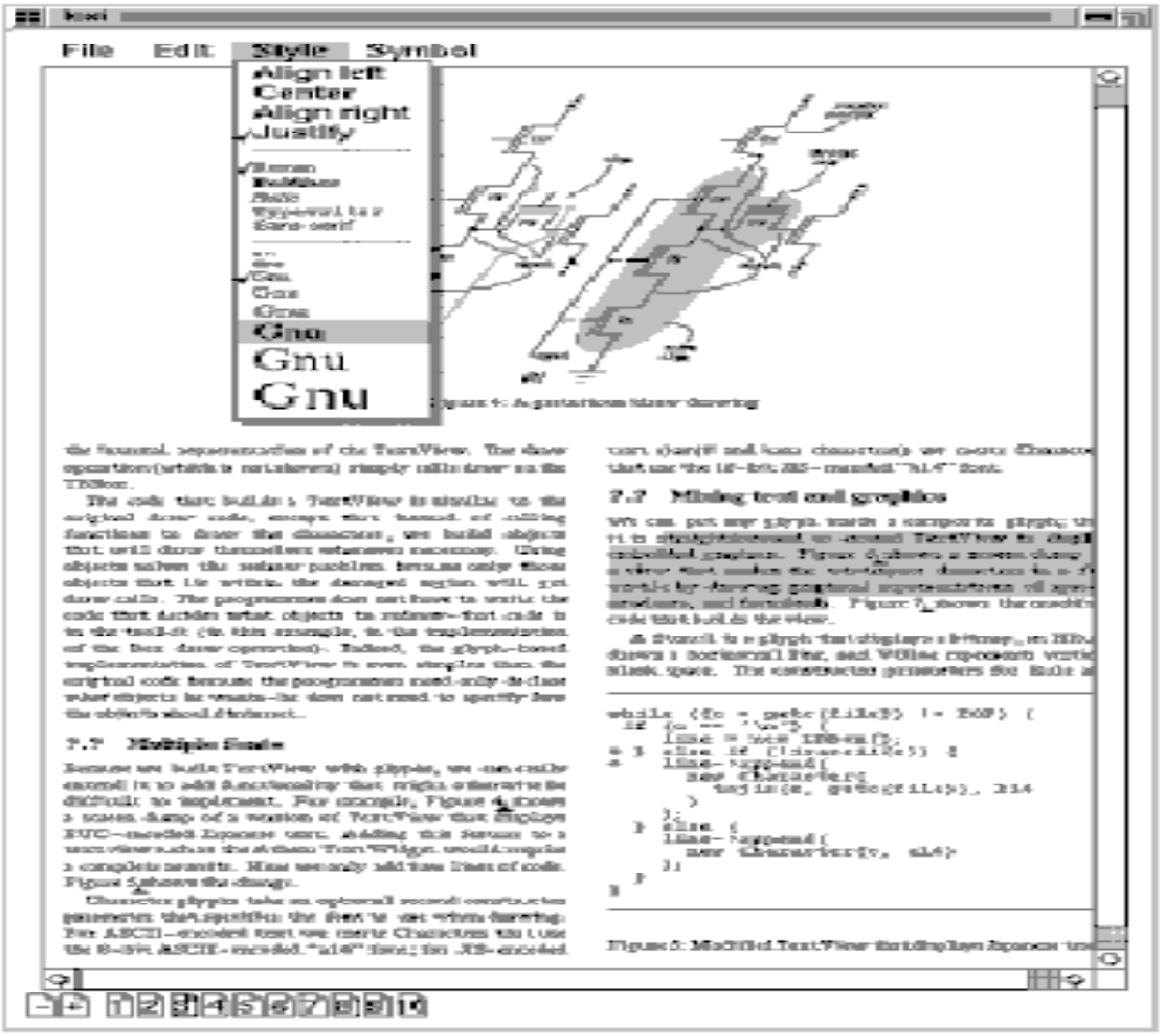
1. Read the pattern once through for an overview. Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.
2. Study the Structure, Participants, and Collaborations sections. Make sure you understand the classes and objects in the pattern and how they relate to one another.
3. Look at the Sample Code section to see a concrete example of the pattern in code. Studying the code helps you learn how to implement the pattern.
4. Choose names for pattern participants that are meaningful in the application context. The names for participants in design patterns are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application. That helps make the pattern more explicit in the implementation.
5. Define the classes. Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.

6. Define application-specific names for operations in the pattern. Here again, the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions. For example, you might use the "Create-"prefix consistently to denote a factory method.
7. Implement the operations to carry out the responsibilities and collaborations in the pattern. The Implementation section offers hints to guide you in the implementation. The examples in the Sample Code section can help as well.

UNIT-II

Design a Document Editor :A Case Study

- The case study is to analyse the design of a "What-You-See-Is-What-you-Get" (WYSIWYG) document editor called **Lexi**.
- Lexi Features are: WYSIWYG Document Editor, Mix text and graphics in a variety of styles, pull-down menus, scrollbars, Icons for jumping to a particular page.



2.1 Design Problems:

1. *Document structure.*
   - How do we represent a document? The choice of internal representation for the document affects nearly every aspect of Lexi's design.
   - All editing, formatting, displaying, and textual analysis will require traversing the representation.

2. *Formatting.*
   - How do we arrange text and graphics on the screen (or paper)?
   - How does Lexi actually arrange text and graphics into lines and columns?

- What objects are responsible for carrying out different formatting policies?
- How do these policies interact with the document's internal representation?

3. *Embellishing the user interface.*
   - Lexi's user interface includes scroll bars, borders, and drop shadows that embellish the WYSIWYG document interface.
   - Such embellishments are likely to change as Lexi's user interface evolves.
   - Hence it's important to be able to add and remove embellishments easily without affecting the rest of the application.
4. *Supporting multiple look-and-feel standards.*
   - Lexi should adapt easily to different look-and-feel standards such as Motif and Presentation Manager (PM) without major modification.
5. *Supporting multiple window systems.*
   - Different look-and-feel standards are usually implemented on different window systems.
   - Lexi's design should be as independent of the window system as possible.
6. *User operations.*
   - Users control Lexi through various user interfaces, including buttons and pull-down menus.
   - The functionality behind these interfaces is scattered throughout the objects in the application.
   - The challenge here is to provide a uniform mechanism both for accessing this scattered functionality and for undoing its effects.

7. *Spelling checking and hyphenation.*
   - How does Lexi support analytical operations such as checking for misspelled words and determining hyphenation points?
   - How can we minimize the number of classes we have to modify to add a new analytical operation?

2.2 Document Structure:

- A document is ultimately just an arrangement of basic graphical elements such as characters, lines, polygons, and other shapes.
- These elements capture the total information content of the document.
- The internal representation should support the following:
  - Maintaining the document's physical structure, that is, the arrangement of text and graphics into lines, columns, tables, etc.
  - Generating and presenting the document visually.
  - Mapping positions on the display to elements in the internal representation.
  - This lets Lexi determine what the user is referring to when he points to something in the visual representation.

Recursive Composition:

- A common way to represent hierarchically structured information is through a technique called **recursive composition,** which entails building increasinglycomplex elements out of simpler ones.
- Recursive composition gives us a way to compose a document out of simple graphical elements.

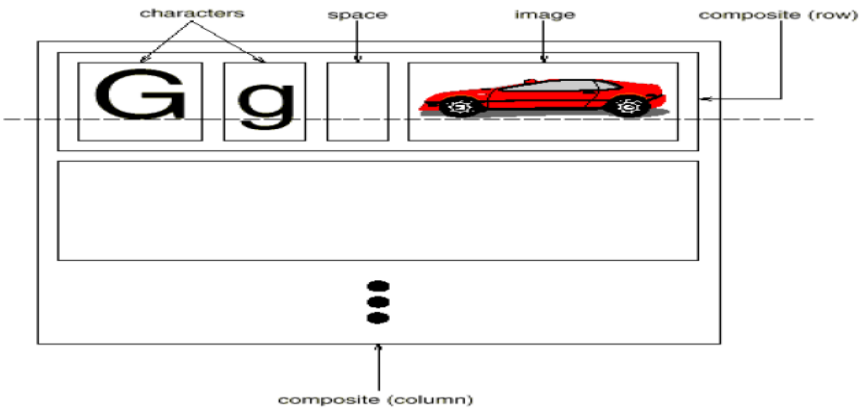


Fig: Recursive composition of text and graphics

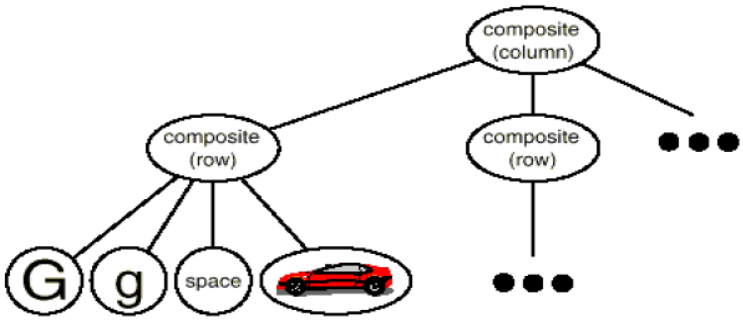


Fig: Object structure for recursive composition of text and graphics

Glyphs: **Glyph** is **abstract** class for all objects that can appear in a document structure.

- Its subclasses define both primitive graphical elements (like characters and images) and structural elements (like rows and columns).

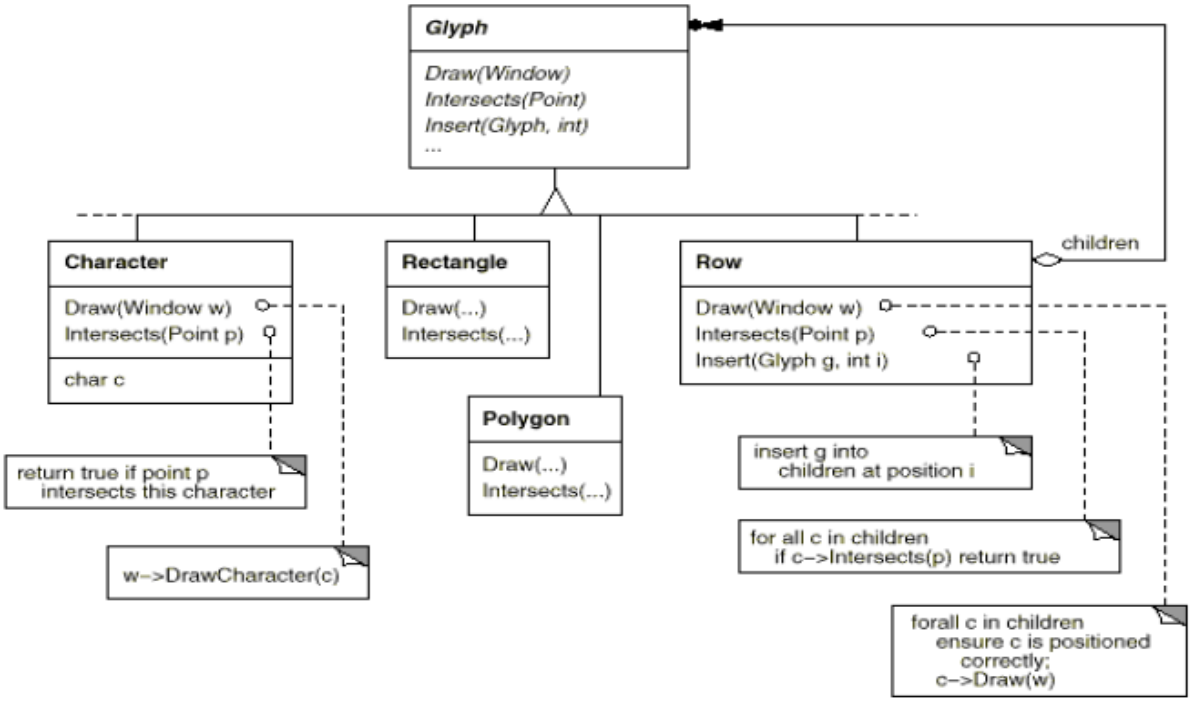


Fig: Partial Glyph class hierarchy

Responsibility	Operations
appearance	virtual void Draw(Window*) virtual void Bounds(Rect&)
hit detection	virtual bool Intersects(const Point&)
structure	virtual void Insert(Glyph*, int) virtual void Remove(Glyph*) virtual Glyph* Child(int) virtual Glyph* Parent()

Table 2.1: Basic glyph interface

Glyphs have three basic responsibilities. They know
(1) how to draw themselves,
(2) what space they occupy, and
(3) their children and parent.

Composite Pattern:

- Recursive composition is good for more than just documents.
- We can use it to represent any much complex, hierarchical structure.
- The Composite pattern captures the essence of recursive composition in object-oriented terms.

2.3Formatting:

- How to construct a particular physical structure, one that corresponds to a properly formatted document.
- Representation and formatting are distinct.
- **Lexi** takes the responsibility of how to arrive at a particular structure.
- It must break text into lines, lines into columns, and so on, taking into account the user's higher-level desires.
- For example, the user might want to vary margin widths, indentation, and tabulation; single or double space; and probably many other formatting constraints.
- We'll restrict "formatting" to mean breaking a collection of glyphs into lines.

Encapsulating the Formatting Algorithm:

- The formatting process, with all its constraints and details, isn't easy to automate.
- There are many approaches to the problem, and people have come up with a variety of formatting algorithms with different strengths and weaknesses.
- Because Lexi is a WYSIWYG editor, an important trade-off to consider is the balance between formatting quality and formatting speed.
- Important Trade-Off
  - Formatting quality vs. formatting speed
  - Formatting speed vs. Storage requirements
- It will be **complex**… so our goals:

    - Keep it well-contained
    - Independent of document structure
        - Add a new glyph… not have to worry about changing format code
        - Add new formatting algorithm – not have to change glyphs
- We'll define a separate class hierarchy for objects that encapsulate formatting algorithms.
- Needs to be easy to change the formatting algorithm
    - If not at run-time, at least at compile-time
- We can make it independent, self-contained and replaceable by putting it in its own class.
- We can make it run-time replaceable by creating a class hierarchy for formatting algorithms.
- The root of the hierarchy will define an interface that supports a wide range of formatting algorithms, and each subclass will implement the interface to carry out a particular algorithm.
- Then we can introduce a Glyph subclass that will structure its children automatically using a given algorithm object.

Compositor and Composition:

- We'll define a **Compositor** class for objects that can encapsulate a formatting algorithm.
- The interface (Table 2.2) lets the compositor know what glyphs to format and when to do the formatting.
- The glyphs it formats are the children of a special Glyph subclass called **Composition**.
- A composition gets an instance of a Compositor subclass (specialized for a particular line-breaking algorithm) when it is created, and it tells the compositor to Compose its glyphs when necessary.

Responsibility	Operations
what to format	void SetComposition(Composition*)
when to format	virtual void Compose()

Table 2.2 Basic compositor interface

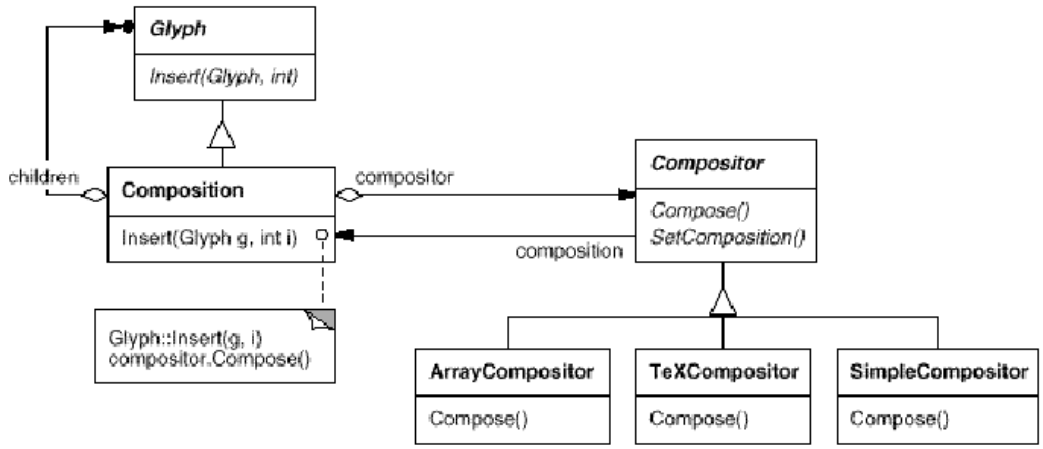


Fig: Composition and Compositor class relationships

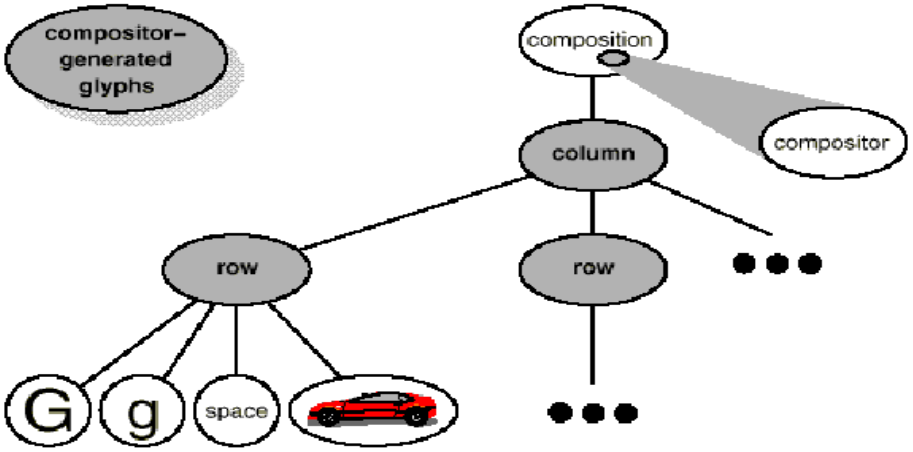


Fig: Object structure reflecting compositor-directed line-breaking

- Each Compositor subclass can implement a different line breaking algorithm.
- The Compositor-Composition class split ensures a strong separation between code that supports the document's physical structure and the code for different formatting algorithms.
- We can add new Compositor subclasses without touching the glyph classes, and vice versa.
- We can change the line breaking algorithm at run-time by adding a single Set Compositor operation to Composition's basic glyph interface.

Strategy Pattern:

- Encapsulating an algorithm in an object is the intent of the Strategy pattern.
- The key participants in the pattern are Strategy objects (which encapsulate different algorithms) and the context in which they operate.
- Compositors are strategies; they encapsulate different formatting algorithms.
- A composition is the context for a compositor strategy.
- The key to applying the Strategy pattern is designing interfaces for the strategy and its context that are general enough to support a range of algorithms.
- In our example, the basic Glyph interface's support for child access, insertion, and removal is general enough to let Compositor subclasses change the document's physical structure, regardless of the algorithm they use to do it.

2.4 Embellishing the User Interface:

- We consider two embellishments in Lexi's user interface.
- The first adds a border around the text editing area to demarcate the page of text.
- The second adds scroll bars that let the user view different parts of the page.
- To make it easy to add and remove these embellishments (especially at run-time), we shouldn't use inheritance to add them to the user interface.
- We need to add and remove the embellishments without changing other classes for providing flexibility.

Transparent Enclosure:

- From a programming point of view, embellishing the user interface involves extending existing code.
- We could add a border to Composition by sub classing it to yield a Bordered Composition class.
- Or we could add a scrolling interface in the same way to yield a Scrollable Composition.
- If we want both scrollbars and a border, we might produce a Bordered Scrollable Composition, and so forth.
- **Transparent enclosure** is a concept, which combines the notions of (1) single-child (or single-component) composition and (2) compatible interfaces. Clients generally can't tell whether they're dealing with the component or its enclosure (i.e., the child's parent), especially if the enclosure simply delegates all its operations to its component.

Monoglyph:

- We can apply the concept of transparent enclosure to all glyphs that embellish other glyphs.
- To make this concept concrete, we'll define a subclass of Glyph called MonoGlyph to serve as an abstract class for "embellishment glyphs," like Border.
- **MonoGlyph stores a reference to a component and forwards all requests to it.**
- That makes MonoGlyph totally transparent to clients by default.
- MonoGlyph implements the Draw operation like this:

```
void MonoGlyph::Draw (Window* w) {_component->Draw(w);}

void Border::Draw (Window* w) { MonoGlyph::Draw(w); DrawBorder(w);   }
```

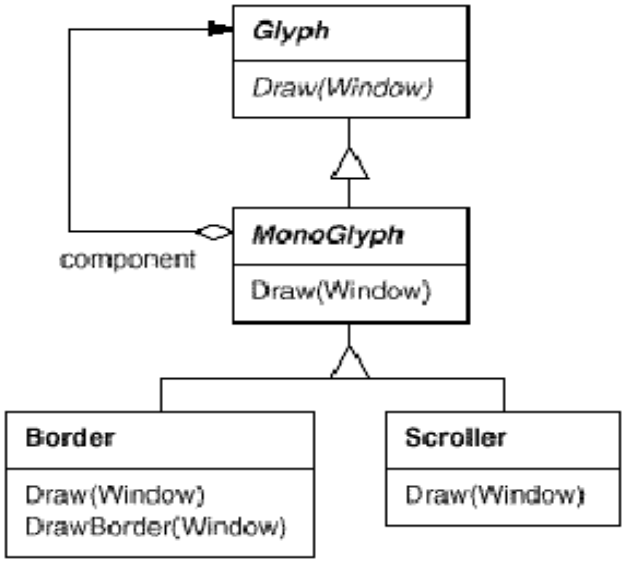


Fig: MonoGlyph class relationships

- MonoGlyph subclasses re-implement atleast one of these forwarding operations.
- **Scroller** is a MonoGlyph that draws its component in different locations based on the positions of two scroll bars, which it adds as embellishments.

- We compose the existing Composition instance in a **Scroller** instance to add the scrolling interface, and we compose that in a Border instance.
- The resulting object structure appears in the following figure:

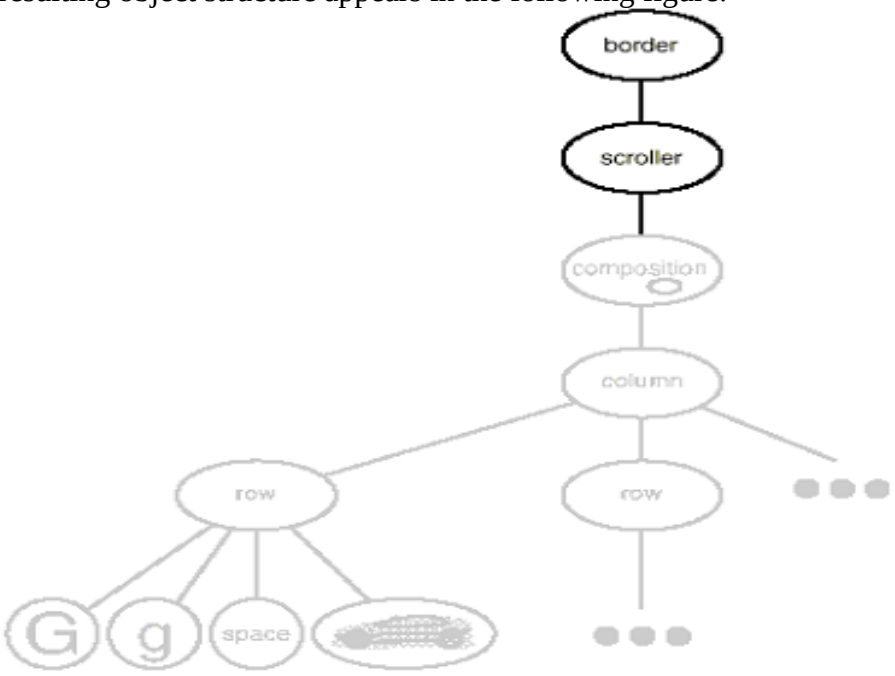


Fig: Embellished object structure

Decorator Pattern:

- The Decorator pattern captures class and object relationships that support embellishment by transparent enclosure.
- In the Decorator pattern, embellishment refers to anything that adds responsibilities to an object.

2.5 Supporting Multiple Look-and-Feel Standards:

- Achieving portability across hardware and software platforms is a major problem in system design.
- One difficulty to portability is the diversity of look-and-feel standards, which are intended to enforce uniformity between applications.
- These standards define guidelines for how applications appear and react to the user.
- Our design goals are
  - to make Lexi conform to multiple existing look-and-feel standards and to make it easy to add support for new standards
  - to support the ultimate in flexibility: changing Lexi's look and feel at run-time.

Abstracting Object Creation:

- Everything we see and interact with in Lexi's user interface is a glyph composed in other, invisible glyphs like Row and Column.
- The invisible glyphs compose visible ones like Button and Character and lay them out properly.

- Style guides have much to say about the look and feel of so-called "widgets," another term for visible glyphs like buttons, scroll bars, and menus that act as controlling elements in a user interface.
- Widgets might use simpler glyphs such as characters, circles, rectangles, and polygons to present data.
- We'll assume we have two sets of widget glyph classes with which to implement multiple look-and-feel standards:
  1. A set of abstract Glyph subclasses for each category of widget glyph.
  2. A set of concrete subclasses for each abstract subclass that implement different look-and-feel standards.
- Lexi must distinguish between widget glyphs for different look-and-feel styles.
- Lexi needs away to determine the look-and-feel standard that's being targeted in order to create the appropriate widgets.

Factories and Product Classes:

```
// Creating a scrollbar…

ScrollBar* sb = guiFactory->CreateScrollBar();
```

- where **guiFactory** is an instance of a MotifFactory class. CreateScrollBar returns a new instance of the proper ScrollBar subclass for the look and feel desired.
- MotifFactory is a subclass of GUIFactory, an abstract class that defines a general interface for creating widget glyphs.
- It includes operations like CreateScrollBar and CreateButton for instantiatingdifferent kinds of widget glyphs.

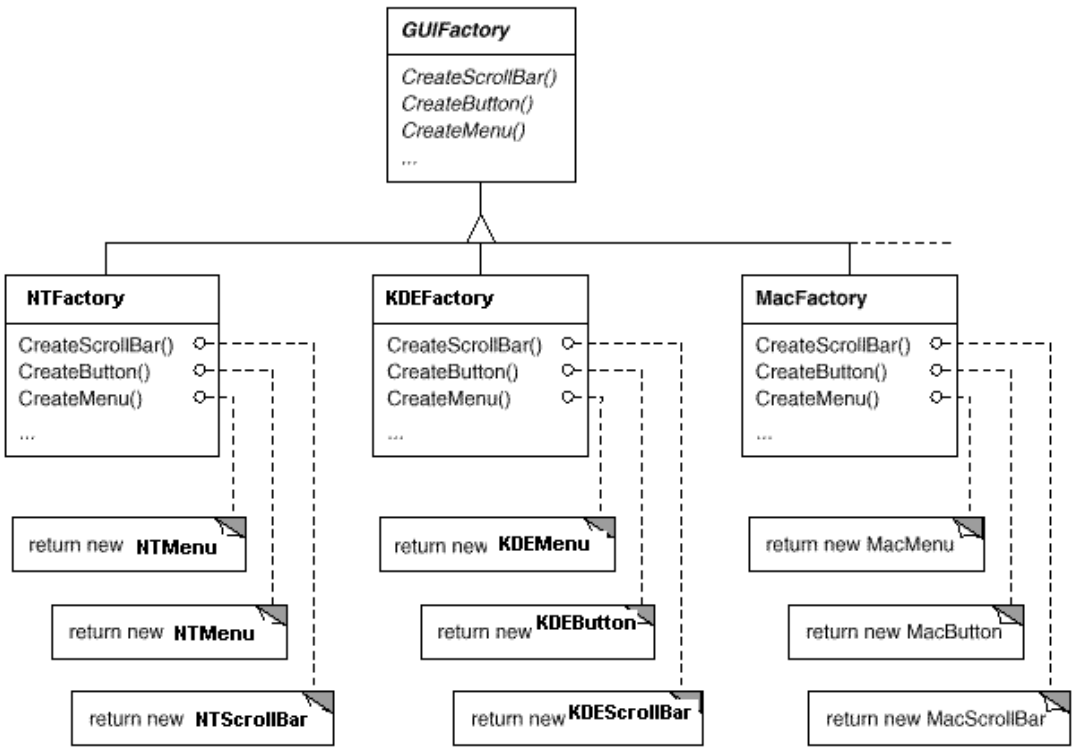


Fig: GUIFactory class hierarchy

- We say that factories create product objects. Moreover, the products that a factory produces are related to one another.

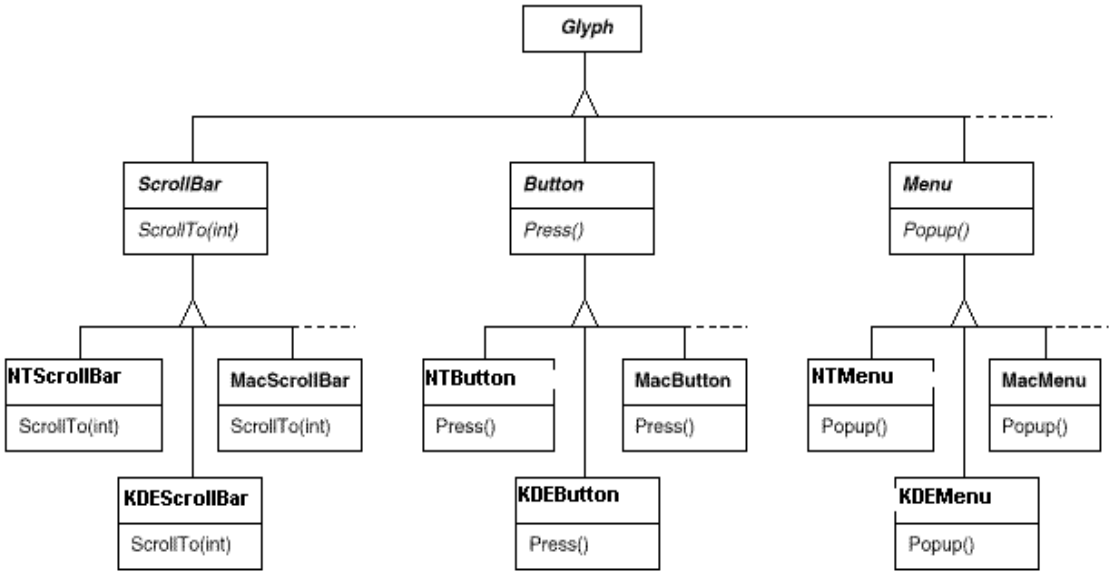


Fig: Abstract product classes and concrete subclasses

Abstract Factory Pattern:

- Factories and products are the key participants in the Abstract Factory pattern.
- This pattern captures how to create families of related product objects without instantiating classes directly.
- It's most appropriate when the number and general kinds of product objects stay constant, and there are differences in specific product families.

2.6 Supporting Multiple Window Systems:

- Look and feel is just one of many portability issues.
- Another is the windowing environment in which Lexi runs.
- A platform's window system creates the illusion of multiple overlapping windows on a bitmapped display.
- It manages screen space for windows and routes input to them from the keyboard and mouse.
- Several important and largely incompatible window systems exist today (e.g., Macintosh, Presentation Manager, Windows, X).

Can We Use an Abstract Factory?

- In applying the Abstract Factory pattern, we assumed we would define the concrete widget glyph classes for each look-and-feel standard.
- That meant we could derive each concrete product for a particular standard (e.g.,MotifScrollBar and MacScrollBar) from an abstract product class (e.g., ScrollBar).
- We have to make the different widget hierarchies adhere to a common set of abstract product interfaces.

- Only then could we declare the Create... operations properly in our abstract factory's interface.
- We solved this problem for widgets by developing our own abstract and concrete product classes.

Encapsulating Implementation Dependencies:

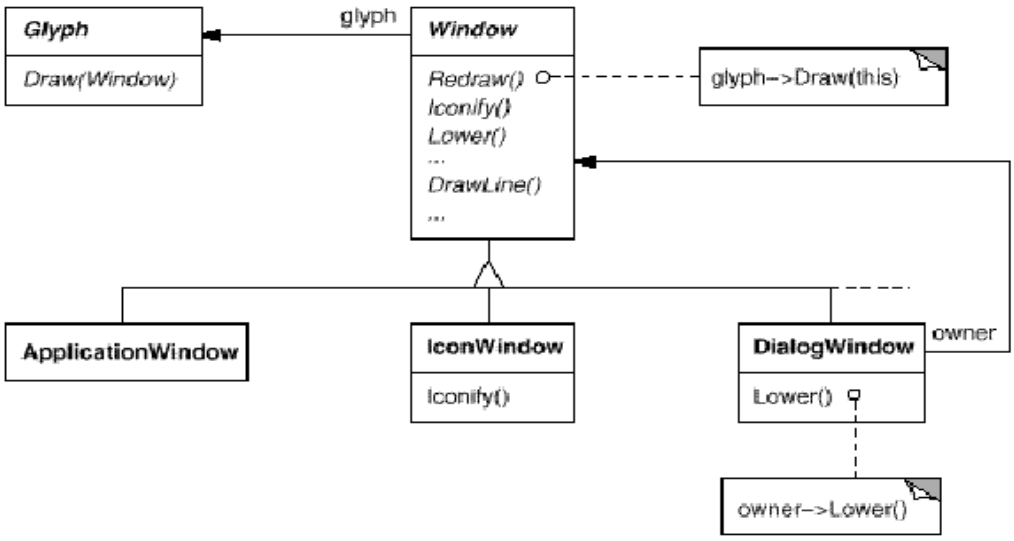
The Window class must span the functionality of windows from different window systems. Let's consider two extreme philosophies:

1. ***Intersection of functionality***. The Window class interface provides only functionality that's common to all window systems. The problem with this approach is that our Window interface winds up being only as powerful as the least capable window system. We can't take advantage of more advanced features even if most (but not all) window systems support them.
2. ***Union of functionality***. Create an interface that incorporates the capabilities of all existing systems. The trouble here is that the resulting interface may well be huge and incoherent.

Responsibility	Operations
window management	virtual void Redraw() virtual void Raise() virtual void Lower() virtual void Iconify() virtual void Deiconify() ...
graphics	virtual void DrawLine(...) virtual void DrawRect(...) virtual void DrawPolygon(...) virtual void DrawText(...) ...

Table 2.3: Window class interface

The resulting class hierarchy gives applications like Lexi a uniform and intuitive windowing abstraction, one that doesn't depend on any particular vendor's window system:



One approach is to implement multiple versions of the Window class and its subclasses, one version for each windowing platform.

Window and WindowImp:

We'll define a separate WindowImp class hierarchy in which to hide different window system implementations. WindowImp is an abstract class for objects that encapsulate window system-dependent code. The following diagram shows the relationship between the Window and WindowImphierarchies.

CREATIONAL PATTERNS

Creational design patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will delegate instantiation to another object.

Creational patterns become important as systems evolve to depend more on object composition than class inheritance.

There are two recurring themes in these patterns.

- They all encapsulate knowledge about which concrete classes the system uses.
- They hide how instances of these classes are created and put together.

2.1ABSTRACTFACTORY: (Object Creational)

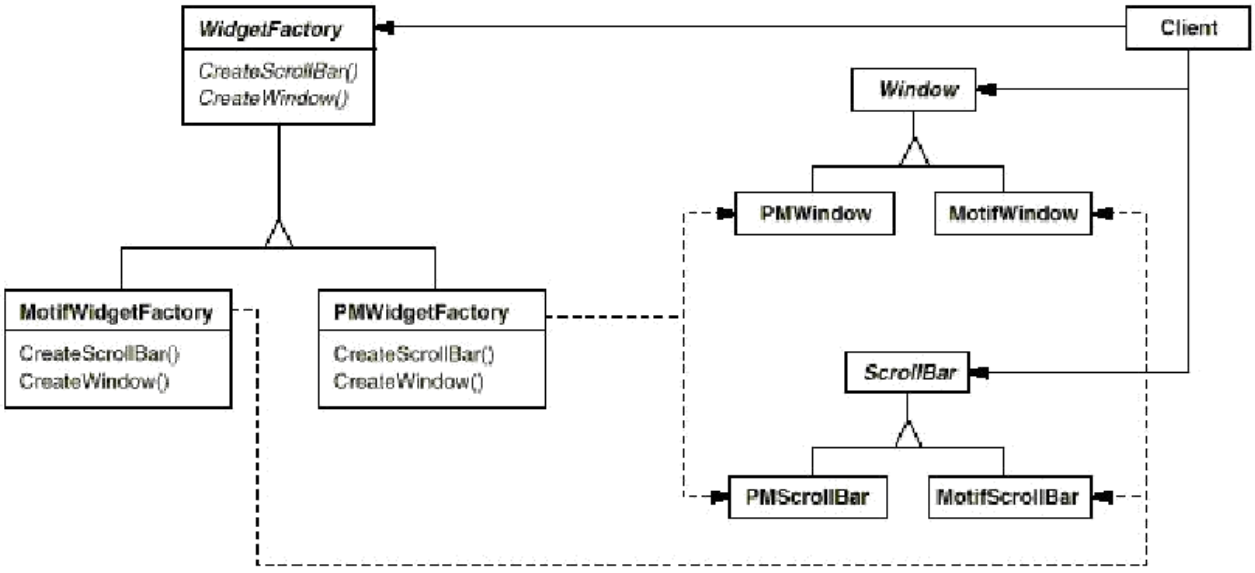
Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

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 usual 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 scrollbar, 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.

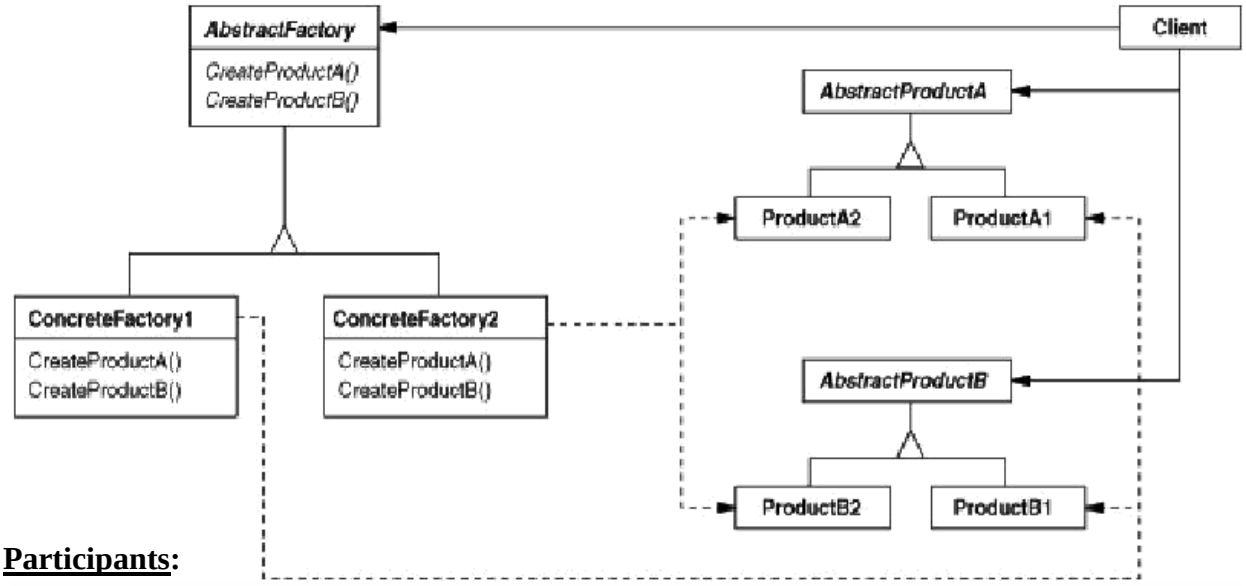


Applicability:

Use the Abstract Factory pattern when

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and we need to enforce this constraint.
- We want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure:



Participants:

- **Abstract Factory** (WidgetFactory)
  - Declares an interface for operations that create abstract product objects.
- **Concrete Factory** (MotifWidgetFactory, PMWidgetFactory)
  - Implements the operations to create concrete product objects.
- **Abstract Product** (Window, ScrollBar)
  - Declares an interface for a type of product object.
- **Concrete Product** (MotifWindow,MotifScrollBar)
  - Defines a product object to be created by the corresponding concrete factory.
  - Implements the Abstract Product interface.
- **Client**
  - Uses only interfaces declared by Abstract Factory and Abstract Product classes.

Collaborations:

- Normally a single instance of a Concrete Factory class is created at run-time. This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
- Abstract Factory defers creation of product objects to its Concrete Factory subclass.

Consequences:

The Abstract Factory pattern has the following benefits and liabilities:

1. **It isolates concrete classes:** The Abstract Factory helps you control the classes of objects than an application creates. Because a factory encapsulates the responsibility and the process of creating product objects.
2. **It makes exchanging product families easy:** The classes of a concrete factory appears only once in an application- i.e., where its instantiated. This makes it easy to change the concrete factory an application uses.
3. **It promotes consistency among products:** When product objects in a family are designed to work together. An application use objects from only one family at a time. Abstract Factory makes this easy to enforce.
4. **Supporting new kinds of products is difficult:** An Abstract Factory interfaces fixes the set of products that can be created. Extending Abstract Factories to produce new kinds of products isn't easy.

Implementation:

Here are some useful techniques for implementing the Abstract Factory pattern.

1. **Factories as singletons:** An application typically needs only one instance of a concrete factory per product family.
2. **Creating the products**: Abstract factory only declares an **interface** for creating products. Its up to concrete product subclasses to actually create them. A concrete factory will specify its products by overriding the factory method for each.
   In many product families are possible, the concrete factory can be implemented using the **prototype** pattern.
3. **Defining extensible factories:** Abstract Factory usually defines a different operation for each kind of product it can produce. The kinds of products are encoded in the operations signatures. Adding new kind of product requires changing the Abstract Factory interface and all the classes that depend on it.

Sample Code:

ClassMazeFactory can create components of mazes. It builds rooms, walls, and doors between rooms. It might be used by a program that reads plans for mazes from a file and builds the corresponding maze. Or it might be used by a program that builds mazes randomly. Programs that build mazes take a MazeFactory as an argument so that the programmer can specify the classes of rooms, walls, and doors to construct.

```
classMazeFactory {

public:

MazeFactory();

virtual Maze* MakeMaze() const

{ return new Maze; }

virtual Wall* MakeWall() const

{ return new Wall; }

virtual Room* MakeRoom(int n) const

{ return new Room(n); }

virtual Door* MakeDoor(Room* r1, Room* r2) const

{ return new Door(r1, r2);
} };
```

Known Uses:

InterViewsuses the "Kit" suffix to denote Abstract Factory classes. It defines WidgetKit and DialogKit abstract factories for generating look-and-feel-specific user interface objects. InterViews also includes a LayoutKit that generates different composition objects depending on the layout desired.

ET++uses the Abstract Factory pattern to achieve portability across different window systems.

Related Patterns:

Abstract Factory classes are often implemented with factory methods (Factory Method), but they can also be implemented using Prototype.

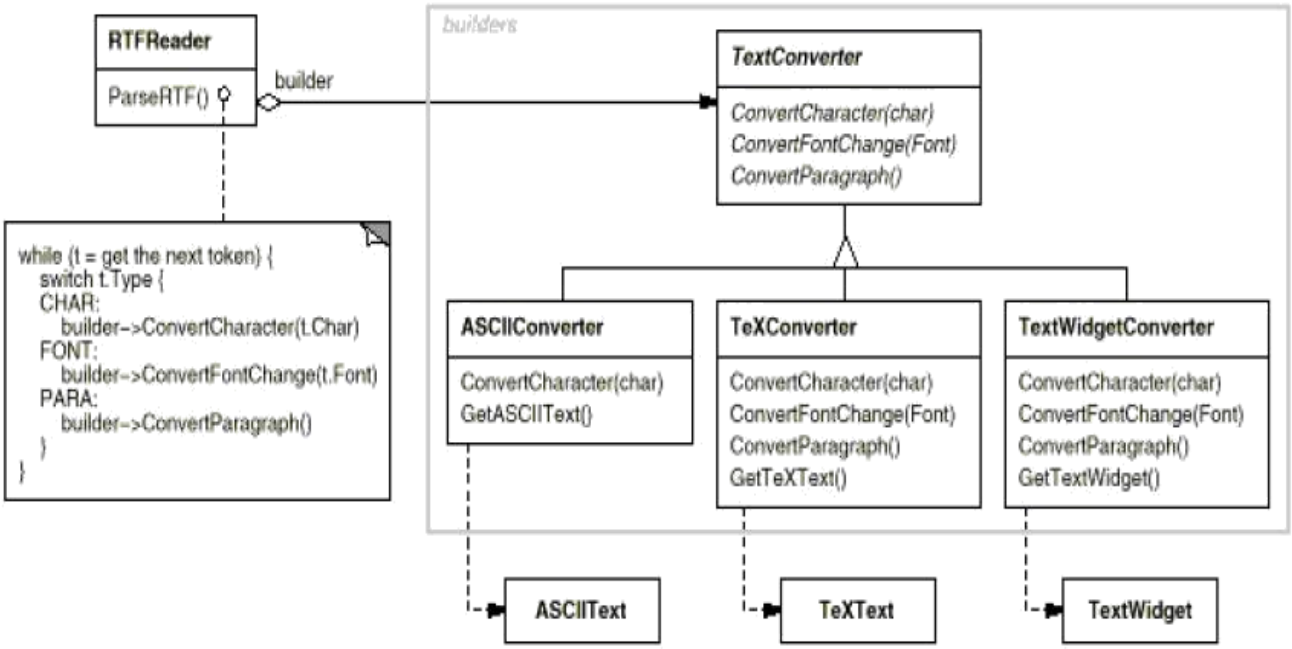
A concrete factory is often a singleton (Singleton).

2.2Builder:

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

Motivation: A reader for the RTF 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 RTFdocument, it uses the TextConverter to perform the conversion. Whenever the RTFReader recognizes an RTF token, 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.



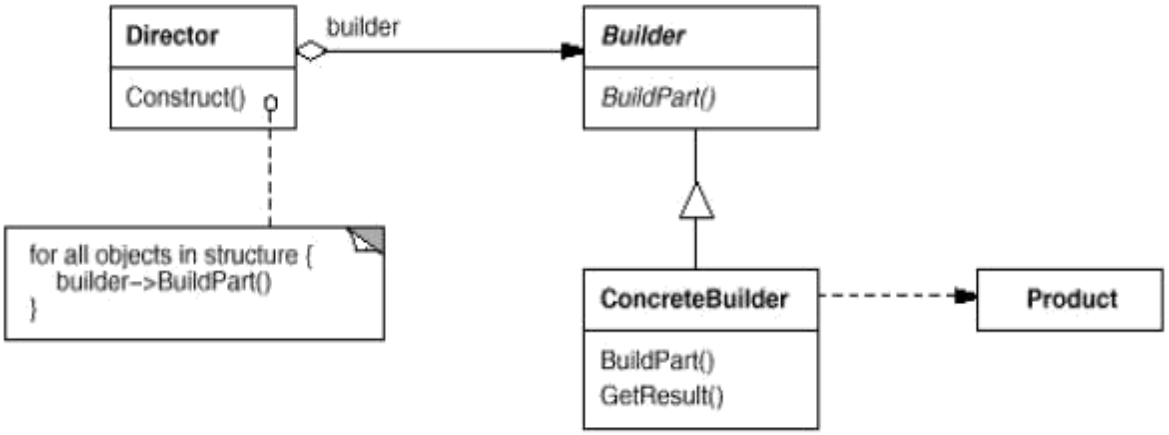
Subclasses of TextConverter specialize in different conversions and formats. Each kind of converter class takes the mechanism for creating and assembling a complex object and puts it behind an abstract interface. The converter is separate from the reader, which is responsible for parsing an RTF document.

The Builder pattern captures all these relationships. Each converter class is called a **builder** in the pattern, and the reader is called the **director**.

Applicability: Use the Builder pattern when

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
- The construction process must allow different representations for theobject that's constructed.

Structure:



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.
- Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

Collaborations:

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Consequences: Here are key consequences of the Builder pattern:

1. **It lets you vary a product's internal representation:** The builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product.
2. **It isolates code for construction and representation:** The builder pattern improves the modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes. The code is written once; then different directors can reuse it to build product variants from the same set of parts.
3. **It gives you finer control over the construction process:** The builder pattern constructs the product step by step under the director's control. Only when the product is finished does the director retrieve it from the builder.

Implementation: Typically there's an abstract Builder class that defines an operation for each component that a director may ask it to create.

Here are other implementation issues to consider:

1. **Assembly and construction interface:** Builders can construct their products in step by step fashion. The builder class interface must be general enough to allow the construction of products for all kinds of concrete builders. The builder return child nodes to the director, which then would pass them back to the builder to build the parent nodes.

2. **Why no abstract class for products?** The products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class. The client is in the position to know which concrete subclass of builder is in use and can handle its products accordingly.
3. **Empty methods as default in Builder:** In C++, the builder methods are intentionally not declared pure virtual member functions. They are defined as empty methods instead.

Sample Code:

We'll define a variant of the CreateMaze member function that takes a builder of class MazeBuilder as an argument.

```
classMazeBuilder {
public:
virtual void BuildMaze() { }
virtual void BuildRoom(int room) { }
virtual void BuildDoor(int roomFrom, int roomTo) { }
virtual Maze* GetMaze() { return 0; } protected:
MazeBuilder();
};
```

This interface can create three things: (1) the maze, (2) rooms with a particular room number, and (3) doors between numbered rooms. The GetMaze operation returns the maze to the client. Subclasses of MazeBuilder will override this operation to return the maze that they build.

Known Uses: The RTF converter application is from ET++. Its text building block uses a builder to process text stored in the RTF format.

Builder is a common pattern in Smalltalk-80:

- The Parser class in the compiler subsystem is a Director that takes a ProgramNodeBuilder object as an argument.
- ClassBuilder is a builder that Classes use to create subclasses for themselves. In this case a Class is both the Director and the Product.
- ByteCodeStream is a builder that creates a compiled method as a byte array. ByteCodeStream is a nonstandard use of the Builder pattern, because the complex object it builds is encoded as a byte array, not as a normal Smalltalk object.

Related Patterns:

Abstract Factory is similar to Builder in that it too may construct complex objects. The primary difference is that the Builder pattern focuses on constructing a complex object step

by step. Abstract Factory's emphasis is on families of product objects (either simple or complex). Builder returns the product as a final step, but as far as the Abstract Factory pattern is concerned, the product gets returned immediately.

A Composite is what the builder often builds.

2.3 Factory Method

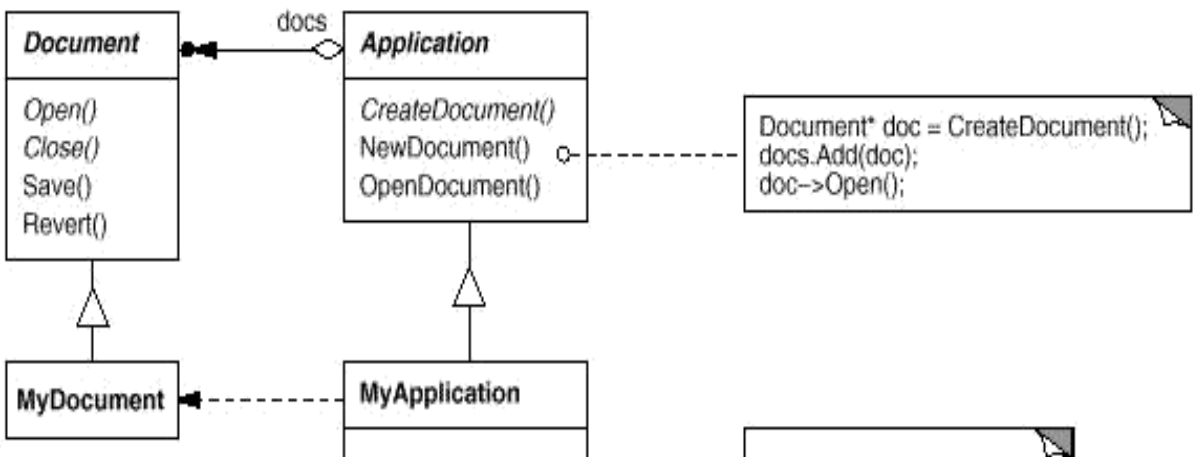
Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Known As: Virtual Constructor

Motivation: Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects as well. Consider a framework for applications that can present multiple documents to the user. Two key abstractions in this framework are the classes Application and Document. This creates a dilemma: The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.

The Factory Method pattern offers a solution. It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.

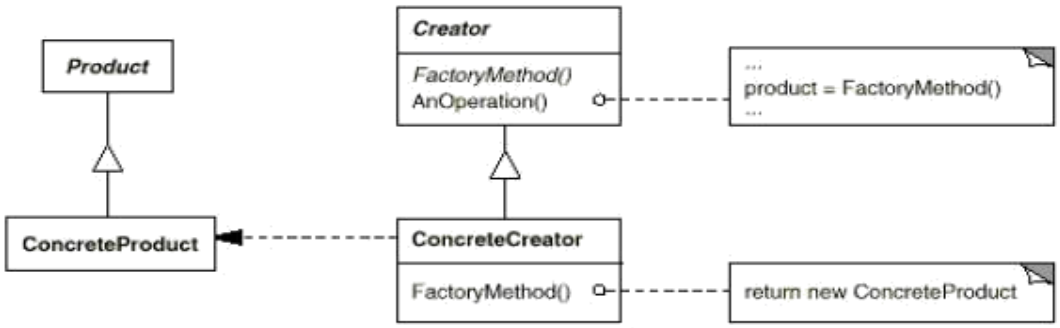
Application subclasses redefine an abstract **CreateDocument** operation on Application to return the appropriate Document subclass. Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class. We call **CreateDocument** a factory method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



Applicability: Use the Factory Method pattern when

- A class can't expect the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure:



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.

Collaborations: Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product.

Consequences: Factory methods eliminate the need to bind application-specific classes into our code. The code only deals with the Product interface; therefore it can work with any user-defined Concrete Product classes.

A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular Concrete Product object.

Here are two additional consequences of the Factory Method pattern:

1. **Provides hooks for subclasses:** Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory method gives sub classes a hook for providing an extended version of an object.
2. **Connects parallel class hierarchies:** The factory method is only called by creators. Parallel class hierarchies results when a class delegates some of its responsibilities to a separate class.

Implementation:

Consider the following issues when applying the FactoryMethodpattern:

1. **Two major varieties.** (1) the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares, and (2) the case when the Creator is a concrete class and provides a default implementation for the factory method.

2. **Parameterized factory methods.** Another variation on the pattern lets the factory method create multiple kinds of products. The factory method takes a parameter that identifies the kind of object to create. All objects the factory method creates will share the Product interface.

3. **Language-specific variants and issues.** Different languages lend themselves to other interesting variations and requirements.

4. **Using templates to avoid sub classing.** Another potential problem with factory methods is that they might force you to subclass just to create the appropriate Product objects. Another way to get around this in C++ is to provide a template subclass of Creator that's parameterized by the Product class.

5. **Naming conventions**. It's good practice to use naming conventions that make it clear you're using factory methods.

Sample Code: The function Create Maze builds and returns a maze. One problem with this function is that it hard-codes the classes of maze, rooms, doors, and walls. We'll introduce factory methods to let subclasses choose these components.

First we'll define factory methods in MazeGame for creating the maze, room, wall, and door objects:

```
class MazeGame {

    public:

    Maze* CreateMaze();

    // factory methods:

    virtual Maze* MakeMaze() const

    { return new Maze; }

    virtual Room* MakeRoom(int n) const

    { return new Room(n); } virtual
    Wall* MakeWall() const {
    return new Wall; }

    {return new Door(r1, r2); }

    };
```

Known Uses: Factory methods provide toolkits and frameworks. The preceding document example is a typical use in MacApp and ET++. The manipulator example is from Unidraw.

ClassView in the Smalltalk-80 Model/View/Controller framework has a method default Controller that creates a controller, and this might appear to be a factory method.

Another example in Smalltalk-80 is the factory method parser Class defined by Behavior (a super class of all objects representing classes).This enables a class to use a customized parser for its source code

The Orbix ORB system from IONA Technologies uses Factory Method to generate an appropriate type of proxy when an object requests a reference to a remote object.

Related Patterns: Abstract Factory is often implemented with factory methods.

Factory methods are usually called within Template Methods.

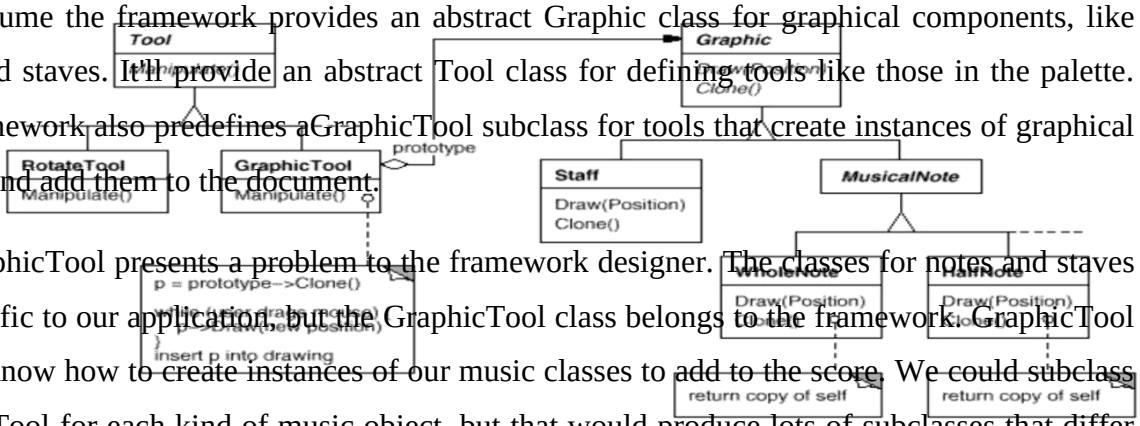
Prototypes don't require sub classing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object.

2.4 Prototype

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

Motivation: We 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.

Let's assume the framework provides an abstract Graphic class for graphical components, like notes and staves. It'll provide an abstract Tool class for defining tools like those in the palette. The framework also predefines aGraphicTool subclass for tools that create instances of graphical objects and add them to the document.



But GraphicTool presents a problem to the framework designer. The classes for notes and staves are specific to our application, but the GraphicTool class belongs to the framework. GraphicTool doesn't know how to create instances of our music classes to add to the score. We could subclass GraphicTool for each kind of music object, but that would produce lots of subclasses that differ only in the kind of music object they instantiate.

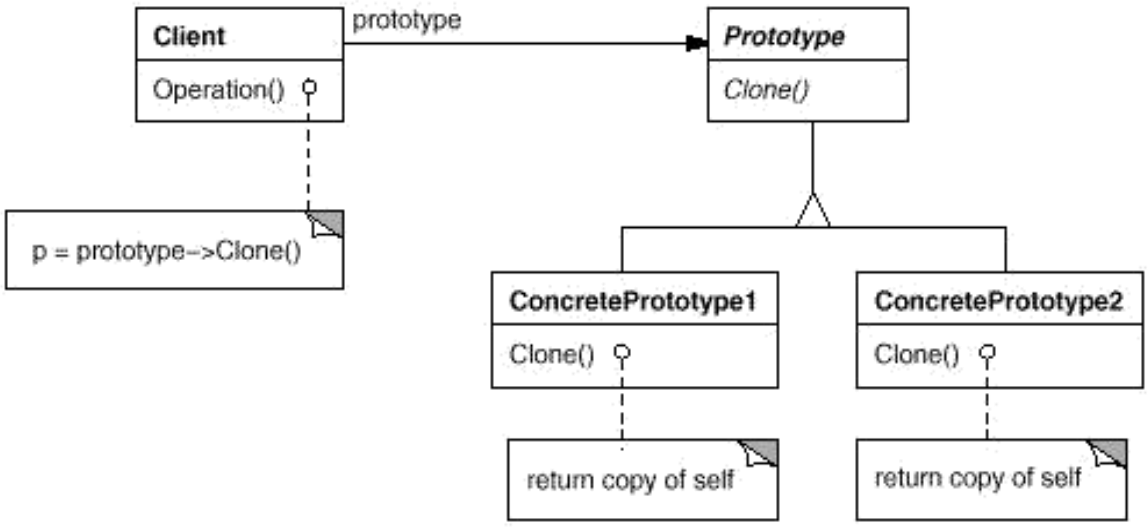
The solution lies in making GraphicTool create a new Graphic by copying or "cloning" an instance of a Graphic subclass. We call this instance a **prototype**. GraphicToolis parameterized by the prototype it should clone and add to the document. If all Graphic subclasses support a Clone operation, then the GraphicTool can clone any kind of Graphic. 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.

A tool for creating whole notes becomes just a GraphicTool whose prototype is a MusicalNote initialized to be a whole note. This can reduce the number of classes in the system dramatically. It also makes it easier to add a new kind of note to the music editor.

Applicability: Use the Prototype pattern when a system should be independent of how its products are created, composed, and represented; and

- when the classes to instantiate are specified at run-time, for example, by dynamic loading; *or*
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products; *or*
- When instances of a class can have one of only a few different combinations of state.

Structure:



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.

Collaborations:

- A client asks a prototype to clone itself.

Consequences:

Prototype has many of the same consequences that Abstract Factory and Builder have: It hides the concrete product classes from the client, thereby reducing the number of names clients know about. Moreover, these patterns let a client work with application-specific classes without modification.

Additional benefits of the Prototype pattern are listed below.

1. **Adding and removing products at run-time:**
2. **Specifying new objects by varying values:**
3. **Specifying new objects by varying structure:**
4. **Reduced sub classing:**
5. **Configuring an application with classes dynamically:**

Implementation:

Consider the following issues when implementing prototypes:

1. **Using a prototype manager**. When the number of prototypes in a system isn't fixed (that is, they can be created and destroyed dynamically), keep a registry of available prototypes. Clients won't manage prototypes themselves but will store and retrieve them from the registry. A client will ask the registry for a prototype before cloning it. We call this registry a **prototype manager**.

2. **Implementing the Clone operation.** The hardest part of the Prototype pattern is implementing the Clone operation correctly.

3. **Initializing clones.** While some clients are perfectly happy with the clone as is, others will want to initialize some or all of its internal state to values of their choosing.

Sample Code: We'll define a MazePrototypeFactory subclass of the MazeFactory class. MazePrototypeFactory will be initialized with prototypes of the objects it will create so that we don't have to subclass it just to change the classes of walls or rooms it creates.

MazePrototypeFactory augments the MazeFactory interface with a constructor that takes the prototypes as arguments:

```
class MazePrototypeFactory : public MazeFactory {
    public:
        MazePrototypeFactory(Maze*, Wall*, Room*, Door*);
        virtual Maze* MakeMaze() const;
        virtual Room* MakeRoom(int) const;
        virtual Wall* MakeWall() const;
        virtual Door* MakeDoor(Room*, Room*) const;
    private:
        Maze* _prototypeMaze;
                Room* _prototypeRoom;
                Wall* _prototypeWall;
                Door* _prototypeDoor;
        };
```

Known Uses: Perhaps the first example of the Prototype pattern was in Ivan Sutherland's Sketchpad system. The first widely known application of the pattern in an object-oriented language was in ThingLab.

Etgdb is a debugger front-end based on ET++ that provides a point-and-click interface to different line-oriented debuggers. Each debugger has a corresponding DebuggerAdaptor subclass.

The "interaction technique library" in Mode Composer stores prototypes of objects that support various interaction techniques. The music editor example discussed earlier is based on the Unidraw drawing framework

Related Patterns:

- Prototype and Abstract Factory are competing patterns in some ways. However, they can also be used together.
- An Abstract Factory might store a set of prototypes from which to clone and return product objects.
- Designs that make heavy use of the Composite and Decorator patterns often can benefit from Prototype as well.

2.5 Singleton:

Intent: Ensure a class only has one instance, and provide a global point of access toit.

Motivation: It's important for some classes to have exactly one instance. Examples:

- 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.

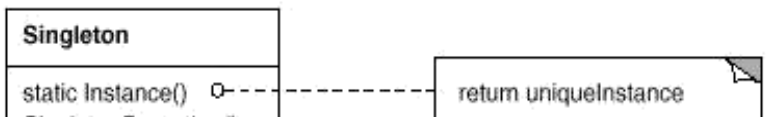
How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created, and it can provide a way to access the instance. This is the **Singleton** pattern.

Applicability: Use the Singleton pattern when

There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

Structure:



Participants:

- **Singleton**
  - Defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - May be responsible for creating its own unique instance.

Collaborations: Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences:

The Singleton pattern has several benefits:

1. *Controlled access to sole instance.* Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
2. *Reduced name space.* The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
3. *Permits refinement of operations and representation.* We can configure the application with an instance of the class we need at run-time.
4. *Permits a variable number of instances.* The pattern makes it easy to change your mind and allow more than one instance of the Singleton class.
5. *More flexible than class operations.* Another way to package a singleton's functionality is to use class operations.

Implementation: Here are implementation issues to consider when using the Singleton pattern:

1. *Ensuring a unique instance.* The Singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can

ever be created. A common way to do this is to hide the operation that creates the instance behind a class operation that guarantees only one instance is created.

2. *Subclassing the Singleton class.* The main issue is not so much defining the subclass but installing its unique instance so that clients will be able to use it. The simplest technique is to determine which singleton you want to use in the Singleton's Instance operation.

Another way to choose the subclass of Singleton is to take the implementation of Instance out of the parent class and put it in the subclass.

A more flexible approach uses a **registry of singletons**. Instead of having Instance define the set of possible Singleton classes, the Singleton classes can register their singleton instance by name in a well-known registry. This approach frees Instance from knowing all possible Singleton classes or instances.

Sample Code:

MazeFactory defines an interface for building different parts of a maze. Subclasses can redefine the operations to return instances of specialized product classes, like BombedWall objects instead of plain Wall objects.

```
class MazeFactory {

public:

static MazeFactory* Instance();
                        // existing interface goes
                        here protected:

                        MazeFactory();

private:

static MazeFactory* _instance; };
```

The corresponding implementation is

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance ()

 {

if (_instance == 0)

{

_instance = new MazeFactory;

}

return _instance;

}
```

Known Uses:

An example of the Singleton pattern in Smalltalk-80is the set of changes to the code, which is ChangeSet current. A more subtle example is the relationship between classes and their meta classes.

The InterViews user interface toolkit uses the Singleton pattern to access the unique instance of its Session and WidgetKit classes, among others.

Related Patterns:

Many patterns can be implemented using the Singleton pattern: AbstractFactory, Builder, and Prototype.

2.6Discussion of Creational Patterns

There are two common ways to parameterize a system by the classes of objects it creates.

1. *Subclass the class that creates the objects*; this corresponds to using the Factory Method pattern. The main drawback of this approach is that it can require creating a new subclass just to change the class of the product. Such changes can cascade.
2. *The other way to parameterize a system relies more on object composition*: Define an object that's responsible for knowing the class of the product objects, and make it a parameter of the system. This is a key aspect of the Abstract Factory, Builder, and Prototype patterns.

All three involve creating a new "factory object" whose responsibility is to create product objects.

- Abstract Factory has the factory object producing objects of several classes.
- Builder has the factory object building a complex product incrementally using a correspondingly complex protocol.
- Prototype has the factory object building a product by copying a prototype object.
- In this case, the factory object and the prototype are the same object, because the prototype is responsible for returning the product.

Consider the drawing editor framework described in the Prototype pattern. There are several ways to parameterize a GraphicTool by the class of product:

- By applying the Factory Method pattern, a subclass of GraphicTool will be created for each subclass of Graphic in the palette. GraphicTool will have a NewGraphic operation that each GraphicTool subclass will redefine.
- By applying the Abstract Factory pattern, there will be a class hierarchy of GraphicsFactories, one for each Graphic subclass. Each factory creates just one product in this case: CircleFactory will create Circles, LineFactory will create

Lines, and so on. A GraphicTool will be parameterized with a factory for creating the appropriate kind of Graphics.

- By applying the Prototype pattern, each subclass of Graphics will implement the Clone operation, and a GraphicTool will be parameterized with a prototype of the Graphic it creates.

Which pattern is best depends on many factors.

- In our drawing editor framework, the Factory Method pattern is easiest to use at first.

- Abstract Factory doesn't offer much of an improvement, because it requires an equally large GraphicsFactory class hierarchy.

- Overall, the Prototype pattern is probably the best for the drawing editor framework, because it only requires implementing a Clone operation on each Graphics class. That reduces the number of classes, and Clone can be used for purposes other than pure instantiation.

- Factory Method makes a design more customizable and only a little more complicated.

- Other design patterns require new classes, whereas Factory Method only requires a new operation.

- Designs that use Abstract Factory, Prototype, or Builder are even more flexible than those that use Factory Method, but they're also more complex.

- Knowing many design patterns gives you more choices when trading off one design criterion against another.

UNIT-3
Structural Patterns

Introduction:

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- **Structural class patterns** use inheritance to compose interfaces or implementations.
- Example: The class form of the **Adapter** pattern.
    - In general, an adapter makes one interface (the adaptee's) conform to another, thereby providing a uniform abstraction of different interfaces.
    - A class adapter accomplishes this by inheriting privately from an adaptee class.
    - The adapter then expresses its interface in terms of the adaptee's.
- Rather than composing interfaces or implementations, **structural object patterns** describe ways to compose objects to realize new functionality.
- The object composition becomes flexible due to the ability to change the composition at run-time, which is impossible with static class composition.
- Example: Composite pattern.
    - How to build a class hierarchy made up of classes for two kinds of objects: primitive and composite.
    - The composite objects let you compose primitive and other composite objects into arbitrarily complex structures.

3.1 Adapter:

Intent:

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

Also Known As: Wrapper

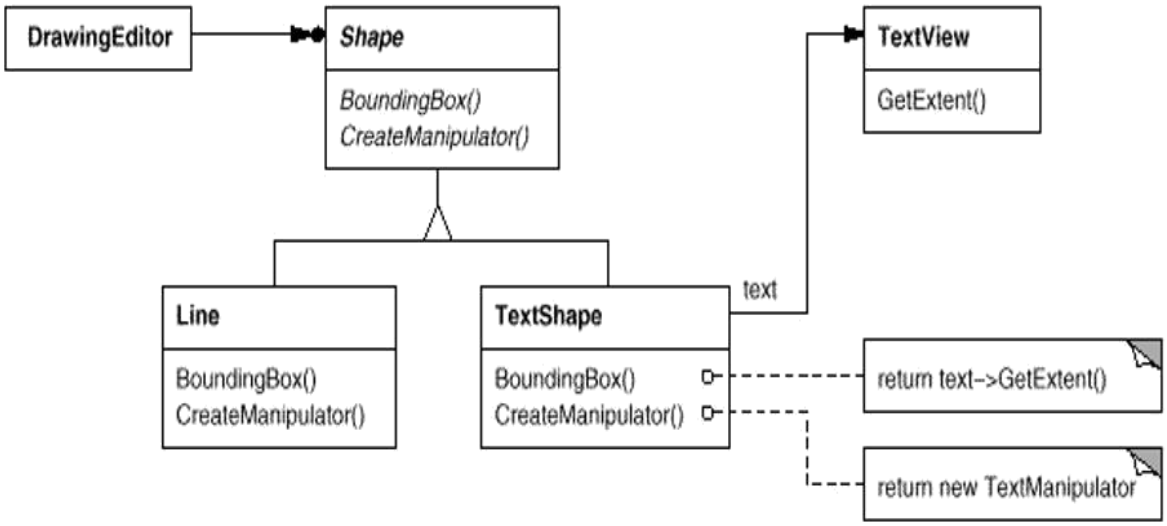
Motivation:

- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.
- Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams.

- 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.
- Classes for elementary geometric shapes are easy to implement. But a **TextShape** subclass that can display and edit text is more difficult to implement. (screen updates, buffer updates)
- How can existing and unrelated classes like **TextView** work in an application that expects classes with a different and incompatible interface?
- We could define **TextShape** so that it adapts the **TextView** interface to Shape's.

  (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.



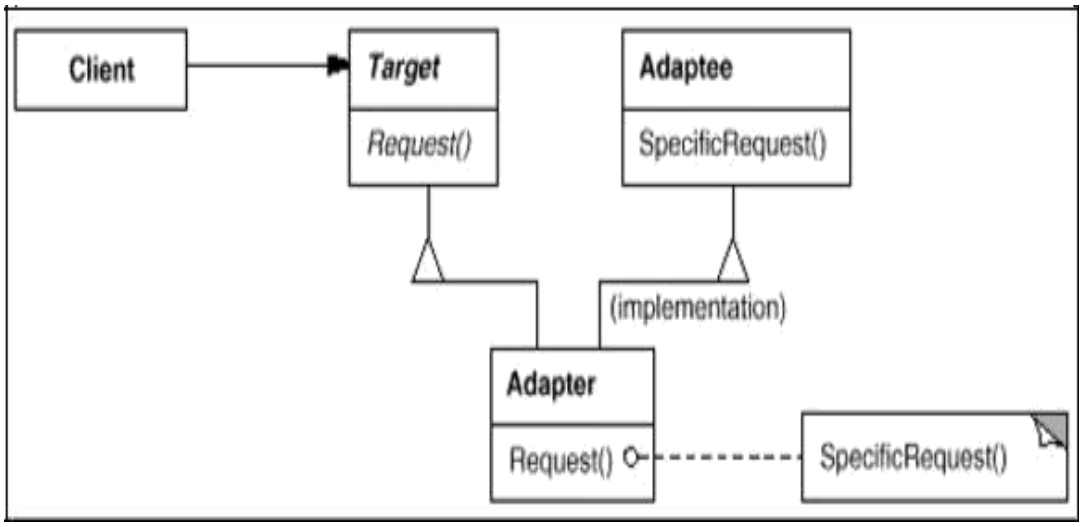
Applicability:

Use the Adapter pattern when

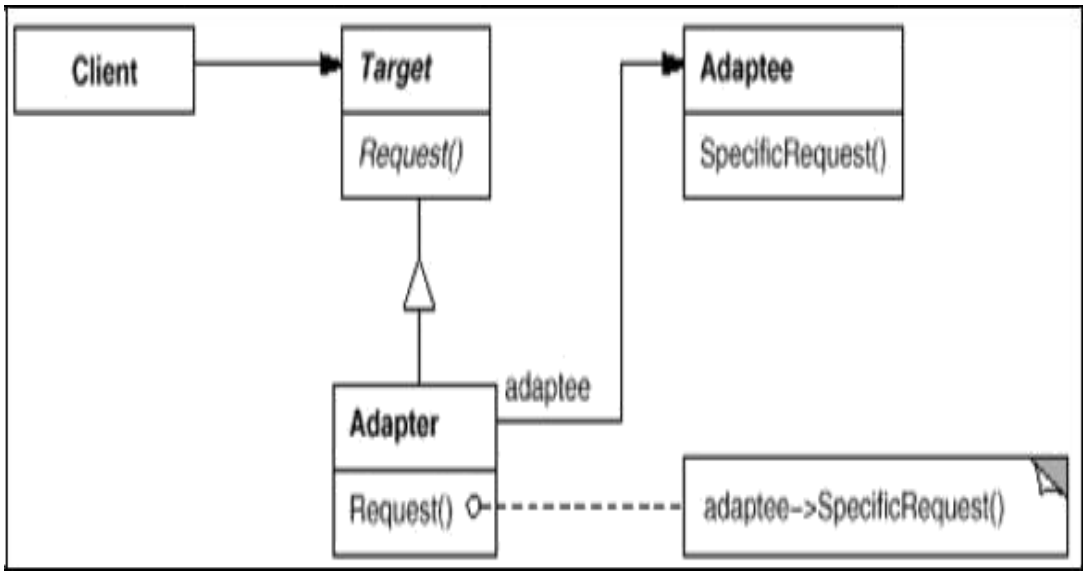
- you want to use an existing class, and its interface does not match the one you need.
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
- (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structure:

Class adapter



Object adapter



Participants:

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

- adapts the interface of Adaptee to the Target interface.

Collaborations:

Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.

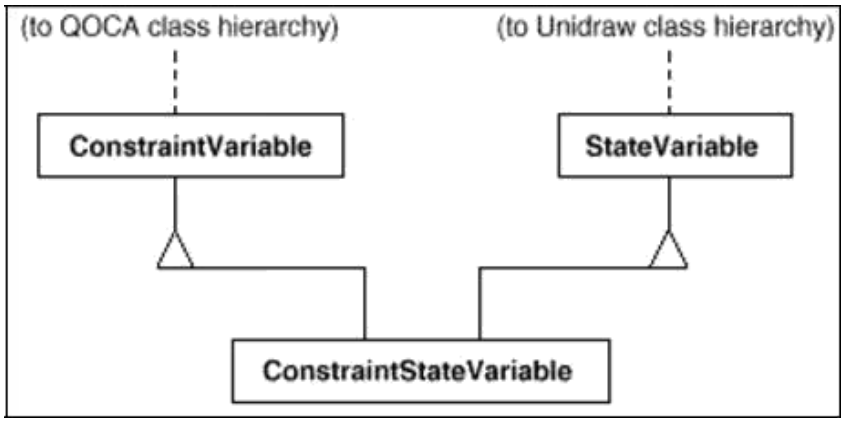
Consequences:

Class and object adapters have different trade-offs. A class adapter

- adapts Adaptee to Target by committing to a concrete Adapter class.
- lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

- lets a single Adapter work with many Adaptees
- makes it harder to override Adaptee behavior.
- How much adapting does Adapter do?
- Pluggable adapters: describe classes with built-in interface adaptation.
- Using two-way adapters to provide transparency.



Implementation:

Here are some issues to keep in mind while implementing Adapter:

1. **Implementing class adapters in C++**. Adapter would inherit publicly from Target and privately from Adaptee. Thus adapter would be a subtype of target but not of adaptee.
2. **Pluggable adapters.**

- find a "narrow" interface for Adaptee

The narrow interface leads to three implementation approaches:

a**. Using abstract operations:** Sub classes must implements the abstract operations and adapt the hierarchically structured object.

b. **Using delegate objects.** Tree displays forwards the requests for accessing the hierarchical structure to a delegate object.

c. **Parameterized adapters.** The usual way to support pluggable adapters in smalltalk is to parameterize an adapter with one or more blocks. The block construct supports adaption without subclassing.

Sample Code:

```
class Shape {
    public:
        Shape();
        virtual void BoundingBox(Point& bottomLeft, Point& topRight)
        const; virtual Manipulator* CreateManipulator() const;};
class TextView {
    public:
        TextView();
        voidGetOrigin(Coord& x, Coord& y) const;
        voidGetExtent(Coord& width, Coord& height) const;
        virtual bool IsEmpty() const;
};
class TextShape : public Shape, private TextView {
    public:
        TextShape();
        virtual void BoundingBox(Point& bottomLeft, Point& topRight)
        const; virtual bool IsEmpty() const;
        virtual Manipulator* CreateManipulator() const;};
```

Known Uses: This pattern is used in the following toolkits: ET++Draw, InterViews 2.6, ObjectWorks\Smalltalk, NeXT'sAppKit

Related Patterns:

- Bridge: has same structure but different intent.
- Decorator: enhances another object without changing its interface.
- Proxy: representative or surrogate for another object and does not change its interface.
- Factory Method is also a related pattern.

3.2 Bridge:

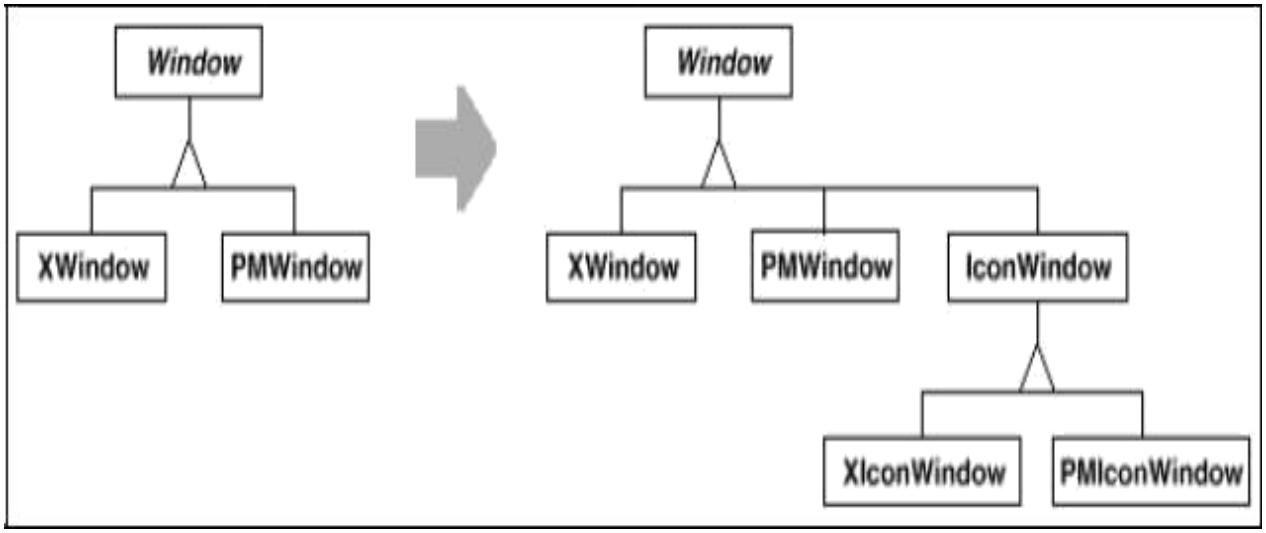
Intent:

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

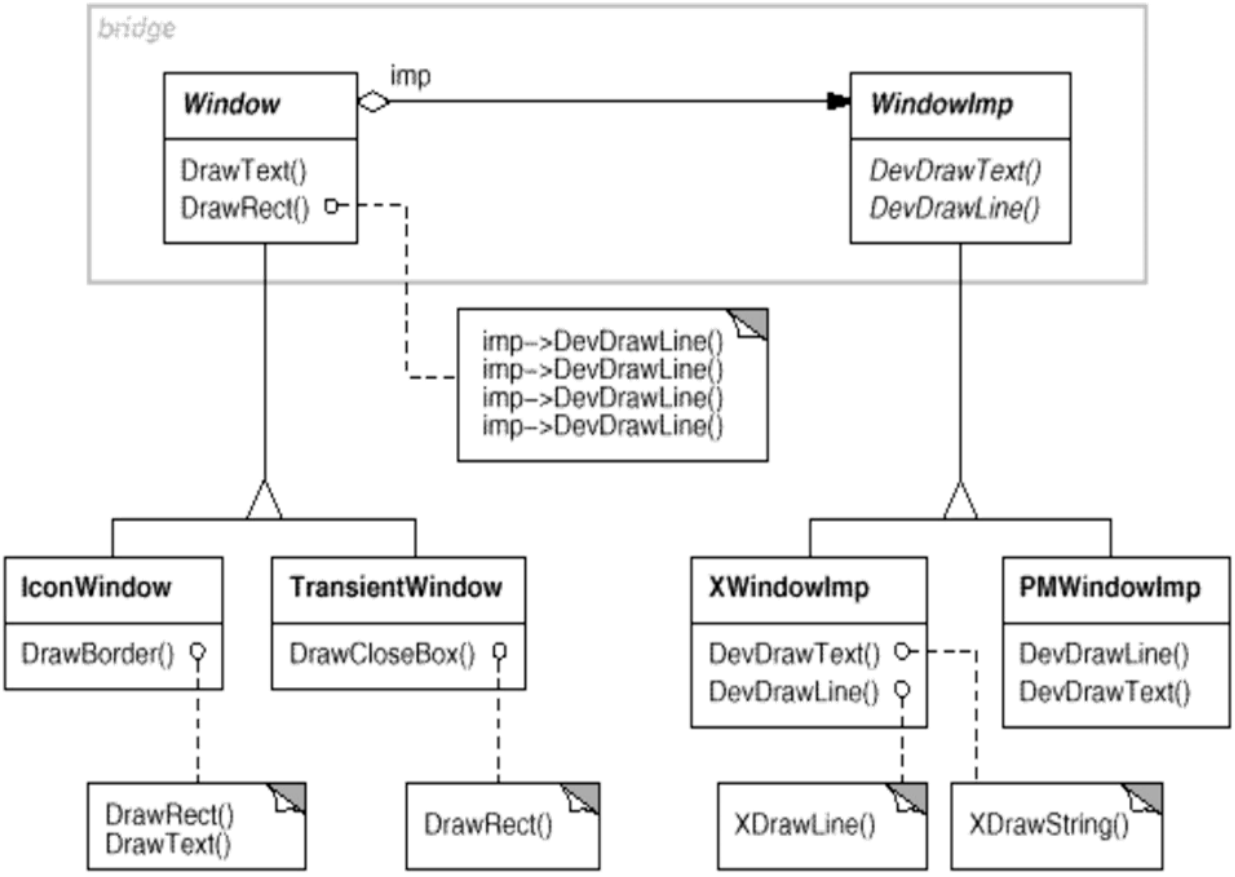
Also Known As: Handle/Body

Motivation:

- When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.
- An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.
- But this approach isn't always flexible enough.



- But this approach has two drawbacks:
  - It's inconvenient to extend the Window abstraction to cover different kinds of windows or new platforms.
  - It makes client code platform-dependent.

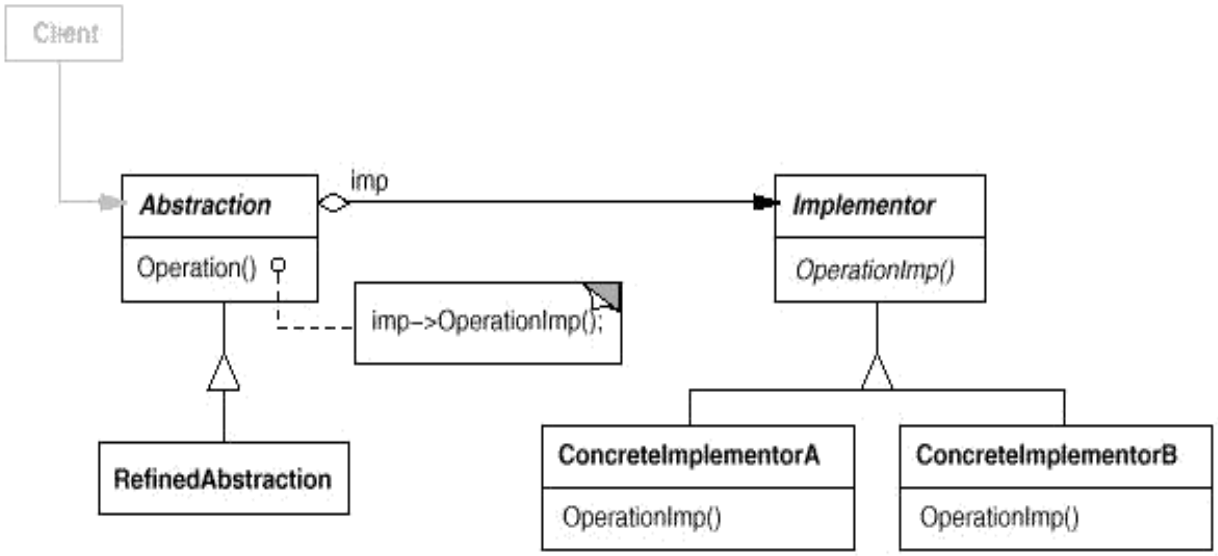


Applicability:

Use the Bridge pattern when

- you want to avoid a permanent binding between an abstraction and its implementation.
- both the abstractions and their implementations should be extensible by subclassing.
- changes in the implementation of an abstraction should have no impact on clients;
- (C++) you want to hide the implementation of an abstraction completely from clients.
- you have a proliferation of classes as shown earlier in the first Motivation diagram.
- you want to share an implementation among multiple objects, and this fact should be hidden from the client.

Structure:



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. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor** (XWindowImp, PMWindowImp)
  - implements the Implementor interface and defines its concrete implementation.

Collaborations:

- Abstraction forwards client requests to its Implementor object.

Consequences:

The Bridge pattern has the following consequences:

1. Decoupling interface and implementation.

- An implementation is not bound permanently to an interface. Decoupling Abstraction and Implementor also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't require recompiling the abstraction class and its clients.

2. **Improved extensibility.**
   - You can extend the Abstraction and Implementor hierarchies independently.

3. **Hiding implementation details from clients.**
   - You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism (if any).

Implementation:

Consider the following implementation issues when applying the Bridge pattern:

1. ***Only one Implementor.*** This is a degenerate case of the bridge pattern ; there is one-to-one relationship between abstraction and implementor. There is one-to-one relationship between abstraction and implementor.
   - Useful when a change in the implementation of a class must not affect its existing clients

2. ***Creating the right Implementor object.***
   - How, when, and where do you decide which Implementor class to instantiate when there's more than one?
   - If abstraction knows about all concrete implementor classes, then it can instantiate one of them in its constructor.
   - To choose a default implementation initially and change it later according to usage.

3. ***Sharing implementors.*** The Body stores a reference count that the Handle class increments and decrements.

4. ***Using multiple inheritance.*** You can use multiple inheritance in C++ to combine an interface with its implementation.

Sample Code:

```
class Window {
    public:
        Window(View* contents);
        // requests handled by window
        virtual void DrawContents();
        virtual void Open();
        virtual void Close();
        virtual void Iconify();
        virtual void Deiconify();
```

```
        // requests forwarded to implementation
        virtual void SetOrigin(const Point& at);
        virtual void SetExtent(const Point& extent);
        virtual void Raise();
        virtual void Lower();
        virtual void DrawLine(const Point&, const Point&);
        virtual void DrawRect(const Point&, const Point&);
        virtual void DrawPolygon(const Point[], int n);
        virtual void DrawText(const char*, const Point&);
    protected:
        WindowImp* GetWindowImp();
        View* GetView();
    private:
        WindowImp* _imp;
        View* _contents; // the window's contents
};
```

Known Uses:

- In ET++, **WindowImp** is called "WindowPort" and has subclasses such as XWindowPort and SunWindowPort.
- The ET++ Window/WindowPort design extends the Bridge pattern in that the WindowPort also keeps a reference back to the Window.
- libg++ defines classes that implement common data structures, such as Set, LinkedSet, HashSet, LinkedList, and HashTable.
- NeXT's AppKit uses the Bridge pattern in the implementation and display of graphical images.

Related Patterns:

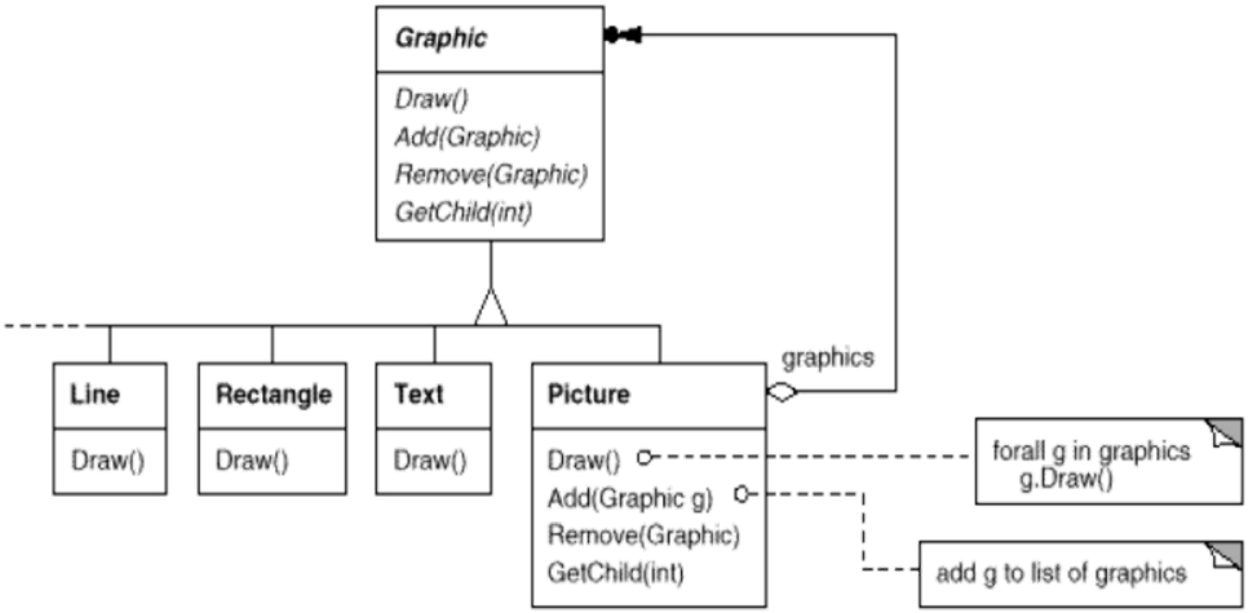
- An Abstract Factory can create and configure a particular Bridge.
- The Adapter pattern is geared toward making unrelated classes work together.

3.3 Composite:

Intent:

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

Motivation: Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components.



Problem:

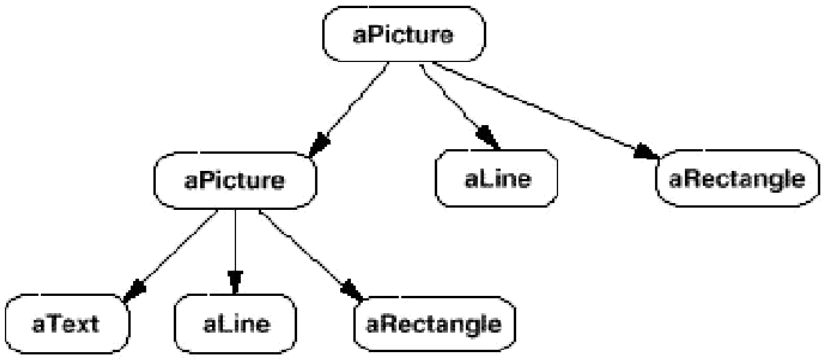
Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically. Having to distinguish these objects makes the application more complex.

Solution:

The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.

- The key to the Composite pattern is an abstract class (**Graphic**) that represents both primitives and their containers.
- **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 define primitive graphical objects.
- Since primitive graphics have no child graphics, none of these subclasses implements child-related operations.
- The Picture class defines an aggregate of Graphic objects.
- 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:

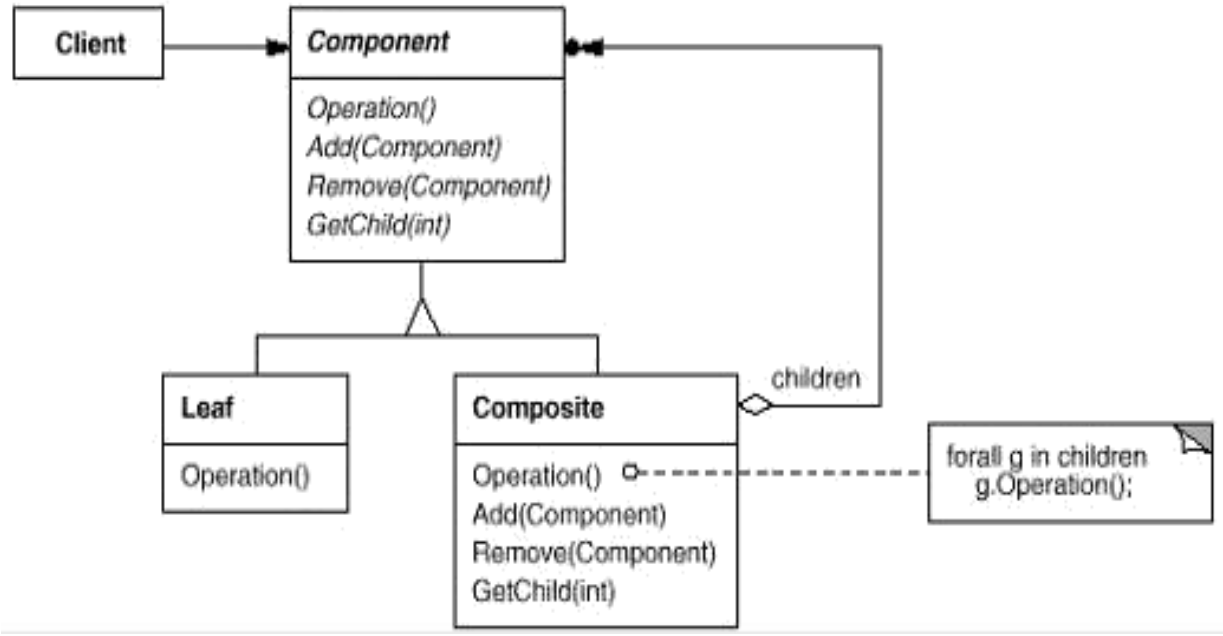


Applicability:

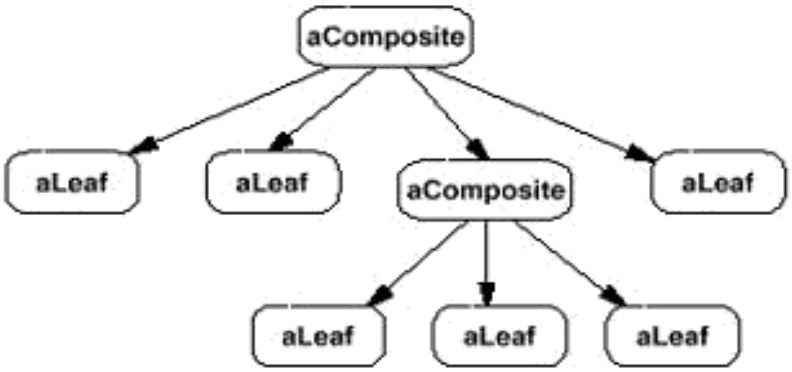
Use the Composite pattern when

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

Structure:



A typical Composite object structure might look like this:



Participants:

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

Collaborations:

- Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Consequences:

The Composite pattern

- defines class hierarchies consisting of primitive objects and composite objects.
- makes the client simple. Clients can treat composite structures and individual objects uniformly.
- makes it easier to add new kinds of components. Clients don't have to be changed for new Component classes.
- can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. You'll have to use run-time checks instead.

Implementation:

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

- **Explicit parent references:** Maintaining references from child components to their parents can simplify the traversal and management of a composite structure. The parent reference simplifies moving up the structure and deleting a component.
- **Sharing components.** When a component can have more than one parent, sharing components becomes difficult. A possible solution is for children's to store multiple parents. The flyweight pattern shows how to rework a design to avoid storing parents together.
- **Maximizing the Component interface.** The composite pattern is to make clients unaware of the specific leaf or composite classes they are using. The component class usually provides default implementation for the operations and leaf and composite sub classes will override them.

Declaring the child management operations.

- Defining the child management interface at the root of the class hierarchy gives you transparency.
- Defining child management in the Composite class gives you safety.

- **Should Component implement a list of Components?** Putting the child pointer in the base class incurs a space penalty for every leaf, even though a leaf never has children.
- **Child ordering.** Many designs specify an ordering on the children of composite.
- **Caching to improve performance.** If you need to traverse or search compositions frequently, the composite class can cache traversal or search information about its children's.

- **Who should delete components?** In languages, without garbage collection, its usually best to make a composite responsible for deleting its children when it is destroyed.
- **What's the best data structure for storing components? The choice of data structure depends on efficiency.**

Sample Code:

```
class Equipment {
public:
        virtual ~Equipment();
        const char* Name(){ return _name; }
        virtual Watt Power();
        virtual Currency NetPrice();
        virtual Currency DiscountPrice();
        virtual void Add(Equipment*);
        virtual void Remove(Equipment*);
        virtual Iterator* CreateIterator();
protected:
        Equipment(const char*);
private:
        const char* _name;
};

class FloppyDisk: public Equipment {
public:
        FloppyDisk(const char*);
        virtual ~FloppyDisk();
        virtual Watt Power();
        virtual Currency NetPrice();
        virtual Currency DiscountPrice();
};

Currency CompositeEquipment::NetPrice () {
        Iterator* i = CreateIterator();
        Currency total = 0;
        for (i->First(); !i->IsDone(); i->Next()) {
```

```
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
    }
class CompositeEquipment : public Equipment {
public:
        virtual ~CompositeEquipment();
        virtual Watt Power();
        virtual Currency NetPrice();
        virtual Currency DiscountPrice();
        virtual void Add(Equipment*);
        virtual void Remove(Equipment*);
        virtual Iterator* CreateIterator();
protected:
        CompositeEquipment(const char*);
private:
        List _equipment;
};
```

Known Uses:

- The original View class of Smalltalk Model/View/Controller was a Composite, and nearly every user interface toolkit or framework has followed in its steps, including ET++ and InterViews, Graphics, and Glyphs.
- The RTL Smalltalk compiler framework uses the Composite pattern extensively.
- Another example of this pattern occurs in the financial domain, where a portfolio aggregates individual assets.
- The Command pattern describes how Command objects can be composed and sequenced with a MacroCommand Composite class.

Related Patterns:

- Often the component-parent link is used for a Chain of Responsibility.
- Decorator is often used with Composite.
- Flyweight lets you share components, but they can no longer refer to their parents.
- Iterator can be used to traverse composites.

- Visitor localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.

3.4 Decorator:

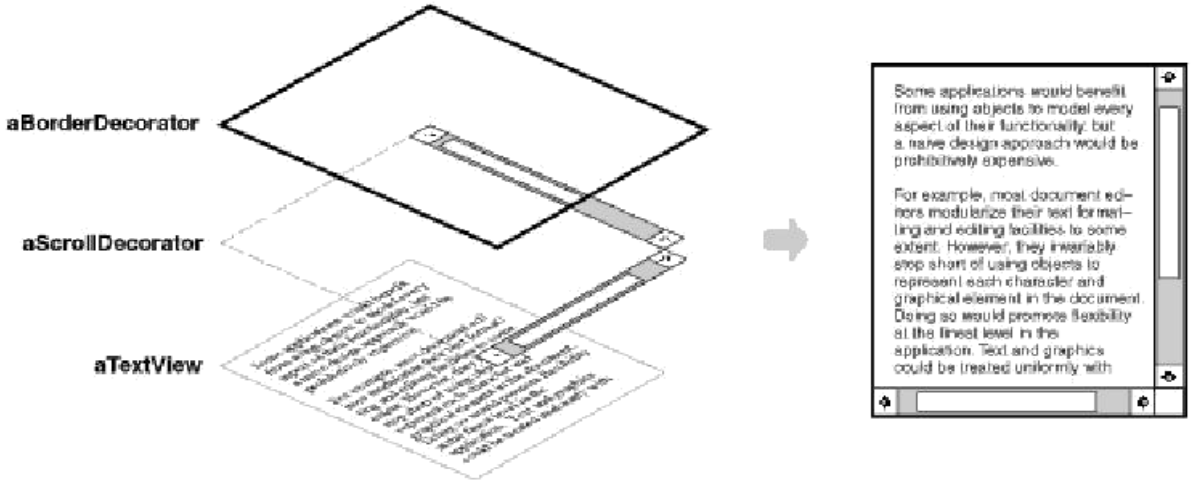
Intent:

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

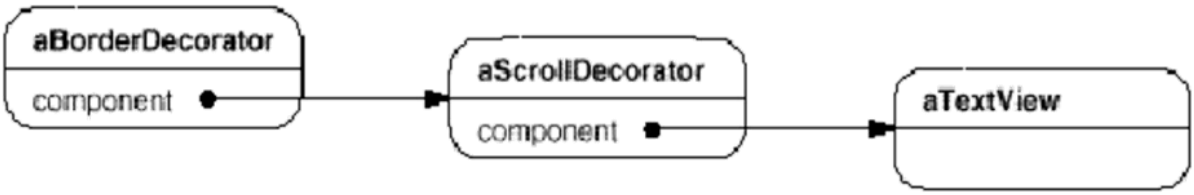
Also Known As: Wrapper

Motivation:

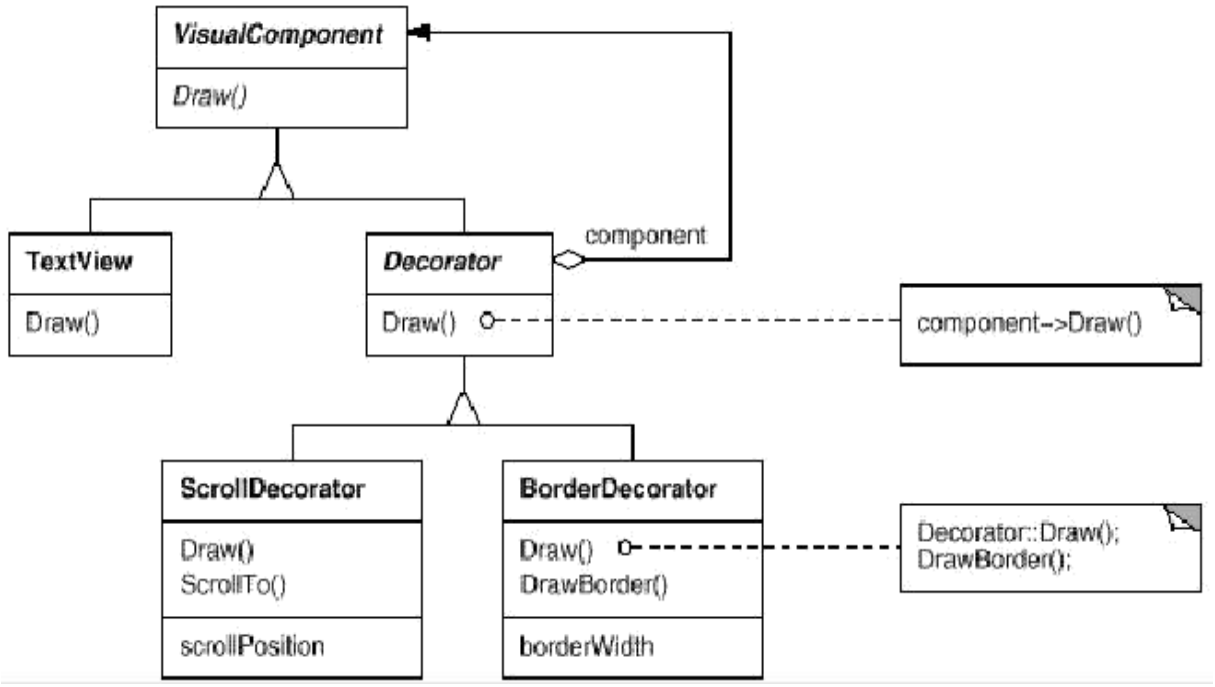
- Sometimes we want to add responsibilities to individual objects, not to an entire class.
- One way to add responsibilities is with inheritance.
- Inheriting a border from another class puts a border around every subclass instance.
- This is inflexible, a client can't control how and when to decorate the component with a border.
- A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator**.
- The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients.
- The decorator forwards requests to the component and may perform additional actions before or after forwarding.
- Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.



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



- VisualComponent is the abstract class for visual objects.
- It defines their drawing and event handling interface.
- We can see how the Decorator class simply forwards draw requests to its component, and how Decorator subclasses can extend this operation.
- Decorator subclasses are free to add operations for specific functionality.

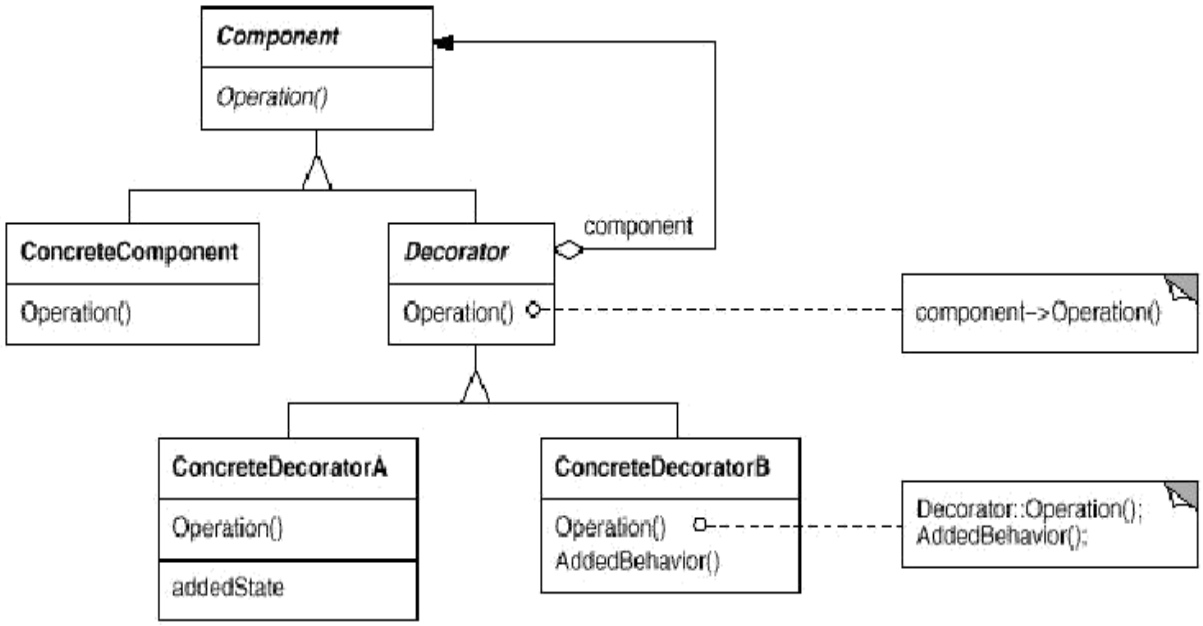


Applicability:

Use Decorator

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

Structure:



Participants:

- **Component** (Visual Component)
  - defines the interface for objects that can have responsibilities added to them dynamically.
- **Concrete Component** (Text View)
  - 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.
- **Concrete Decorator** (Border Decorator, Scroll Decorator)
  - adds responsibilities to the component.

Collaborations:

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

Consequences:

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

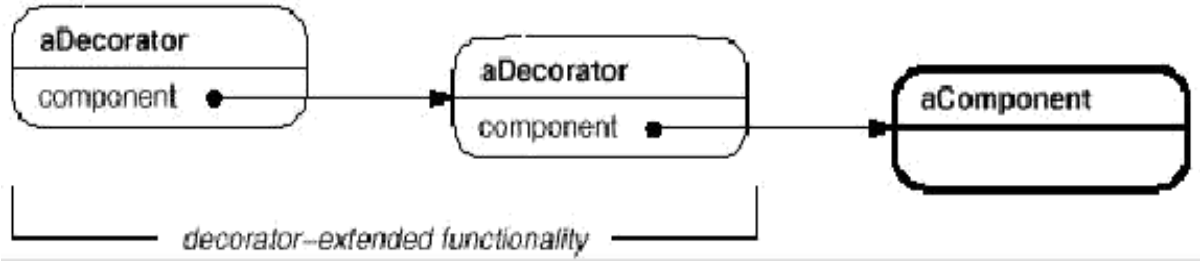
1. **More flexibility than static inheritance:** The decorator pattern provides more flexible way to add responsibilities to objects that can be had with static inheritance. With decorators, responsibilities can be added and removed at run time.

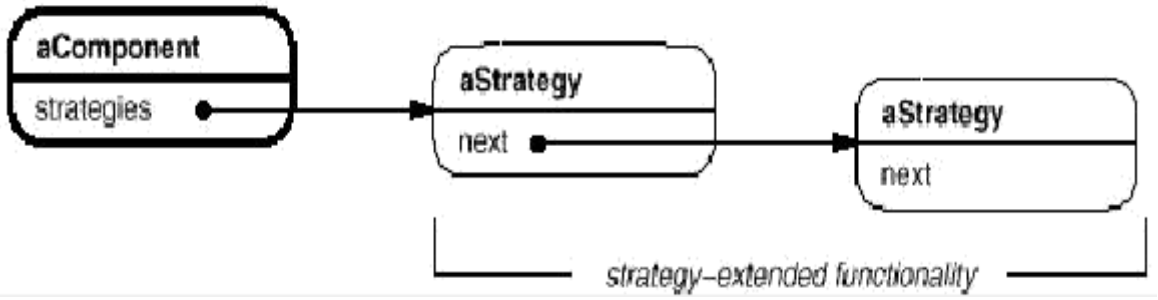
2. **Avoids feature-laden classes high up in the hierarchy:** Decorator offers a pay-as-you-go approach to adding responsibilities.
3. **A decorator and its component aren't identical:** A decorator act as a transparent enclosure. But from an object identity point of view, a decorator component is not identical to the component itself.
4. **Lots of little objects. (that all look alike):** A design that uses decorator often results in systems composed of lot of littlie objects that look alike. The objects differ only in the way they are interconnected, not in their class or in the value of their variables.

Implementation:

Several issues should be considered when applying the Decorator pattern:

1. **Interface conformance:** A decorator object interface must conform to the interface of the component it decorates. Concrete decorator classes must therefore inherit from a common class.
2. **Omitting the abstract Decorator class:** There is no need to define an abstract decorator class when you only need to add one responsibility. You can merge decorator responsibility for forwarding requests to the component into the concrete decorator.
3. **Keeping Component classes lightweight**: To ensure a conforming interface, components and decorators must descend from a common component class. It is important to keep this common class as light weight.
4. **Changing the skin of an object versus changing its guts:** We can think of a decorator as skin over an object that changes its behavior. An alternative is to change the objects guts. The strategy pattern is a good example of a pattern for changing the guts.





Sample Code:

```
class VisualComponent {
public:
    VisualComponent();
    virtual void Draw();
    virtual void Resize();
    // ...
};

class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);
    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
 void Decorator::Draw () {
     _component->Draw();
}
 void Decorator::Resize () {
     _component->Resize();
}

class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);
    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};
```

```
Void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

Known Uses:

Many object-oriented user interface toolkits use decorators

Ex. InterViews, ET++, ObjectWorks\Smalltalk class library

Related Patterns:

Adapter: a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

Composite: A decorator can be viewed as a degenerate composite with only one component.

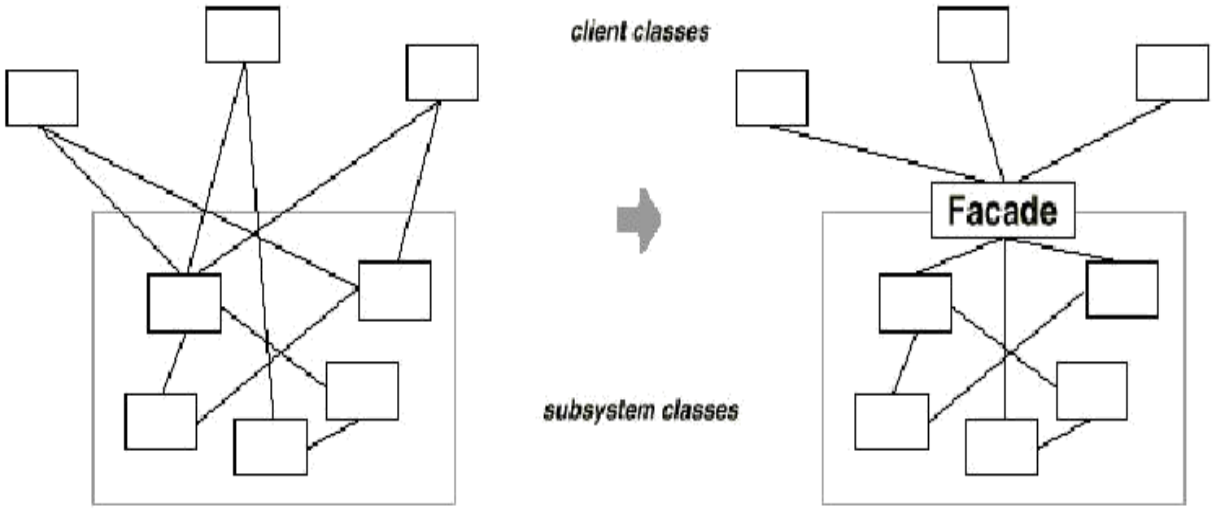
Strategy: A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

3.5 Facade:

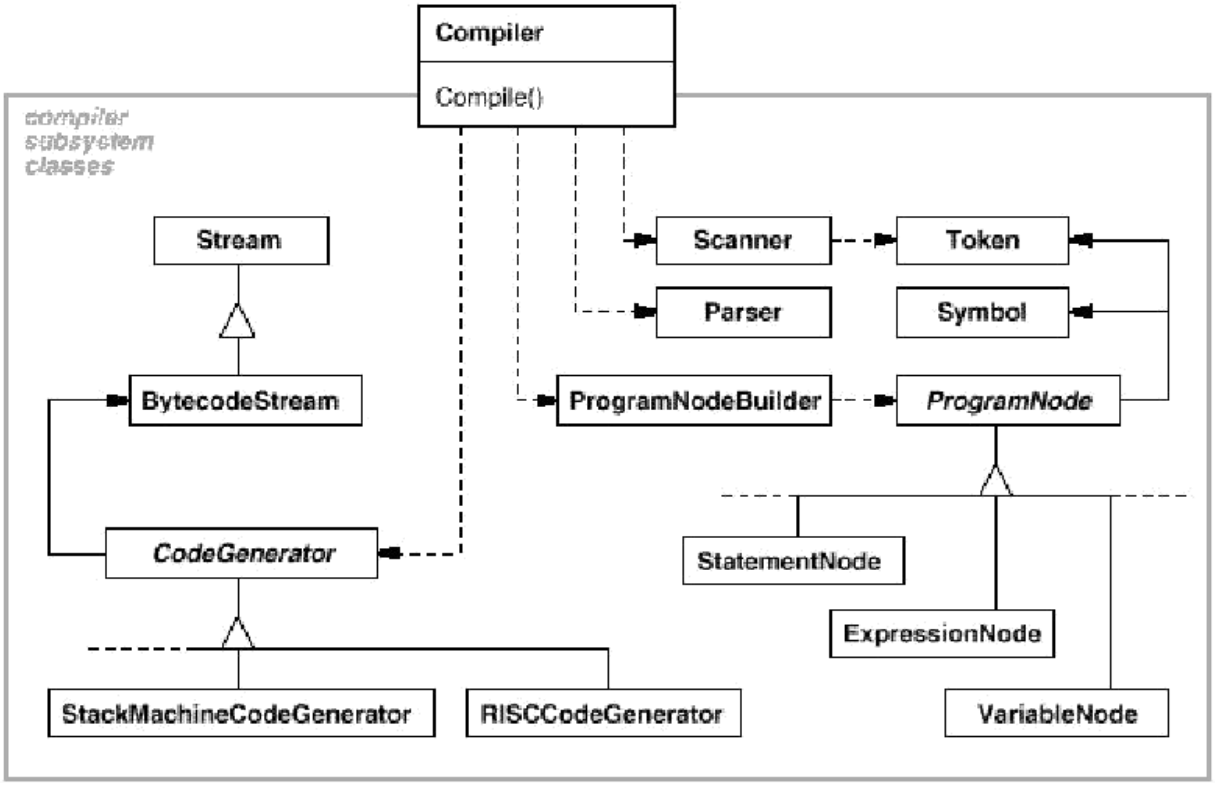
Intent: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation:

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



- The Compiler class acts as a facade: It offers clients a single, simple interface to the compiler subsystem.
- It joins together the classes that implement compiler functionality without hiding them completely.
- The compiler facade makes life easier for most programmers without hiding the lower-level functionality from the few that need it.

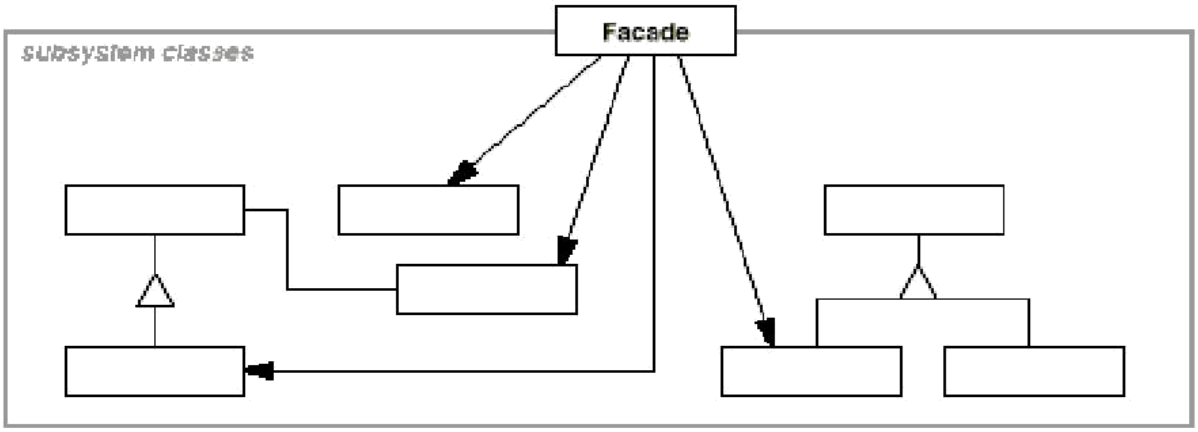


Applicability:

Use the Facade pattern when

- you want to provide a simple interface to a complex subsystem.
- there are many dependencies between clients and the implementation classes of an abstraction.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.

Structure:



Participants:

- **Facade** (Compiler)
  - Knows which subsystem classes are responsible for a request.
  - Delegates client requests to appropriate subsystem objects.
- **Subsystem classes** (Scanner, Parser, ProgramNode, etc.)
  - Implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

Collaborations:

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

Consequences:

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

Implementation:

Consider the following issues when implementing a facade:

1. **Reducing client-subsystem coupling:** The coupling between clients and the subsystem can be reduced even further by making façade an abstract class with concrete subclasses for different implementation of a subsystem. Clients that use the facade don't have to access its subsystem objects directly.
2. **Public versus private subsystem classes:** The public interface to a subsystem consists of classes that all clients can access. The private interface is just for subsystem extenders. A class encapsulates state and operations, while a subsystem encapsulates classes.

Sample Code:

The Scanner class takes a stream of characters and produces a stream of tokens, one token at a time.

```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();
    virtual Token& Scan();
private:
    istream& _inputStream;
};
```

The class Parser uses a ProgramNodeBuilder to construct a parse tree from a Scanner's tokens.

```
class Parser {
public:
    Parser();
    virtual ~Parser();
    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

Known Uses :

- ObjectWorks\Smalltalk compiler system
- ET++ application framework

- The Choices operating system

Related Patterns:

- Abstract Factory: It can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Façade to hide platform-specific classes.
- Mediator: It is similar to Facade in that it abstracts functionality of existing classes.
- Singleton: Usually only one Facade object is required. Thus Façade objects are often Singletons.

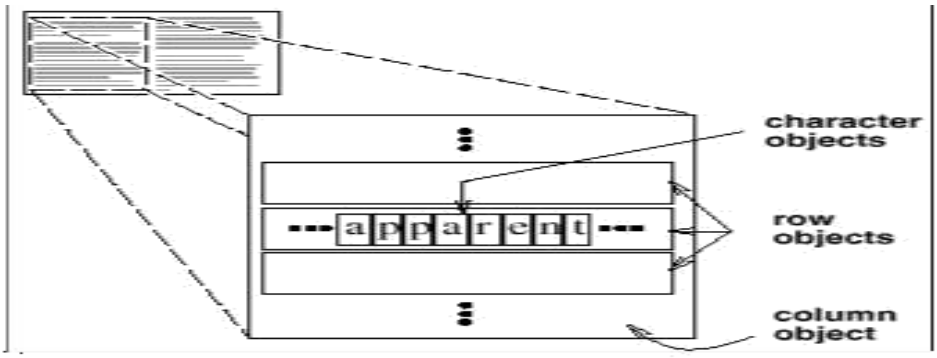
3.6 Flyweight:

Intent:

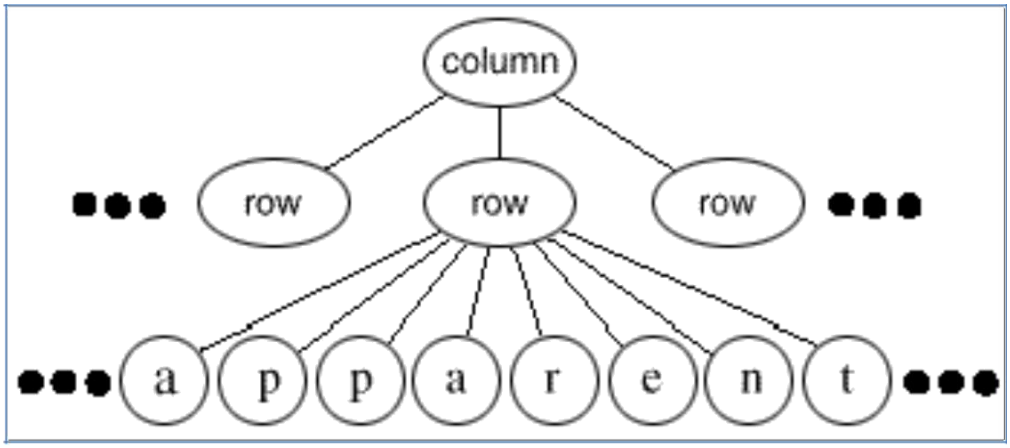
Use sharing to support large numbers of fine-grained objects efficiently.

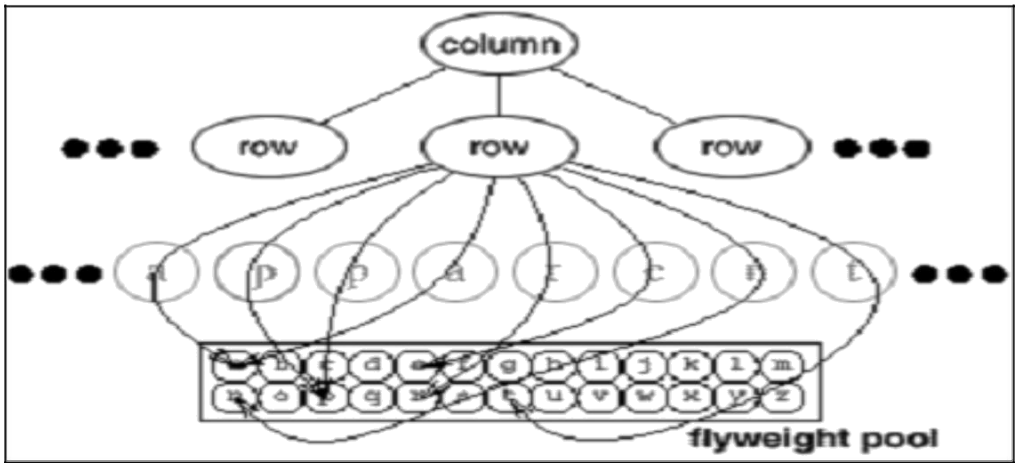
Motivation:

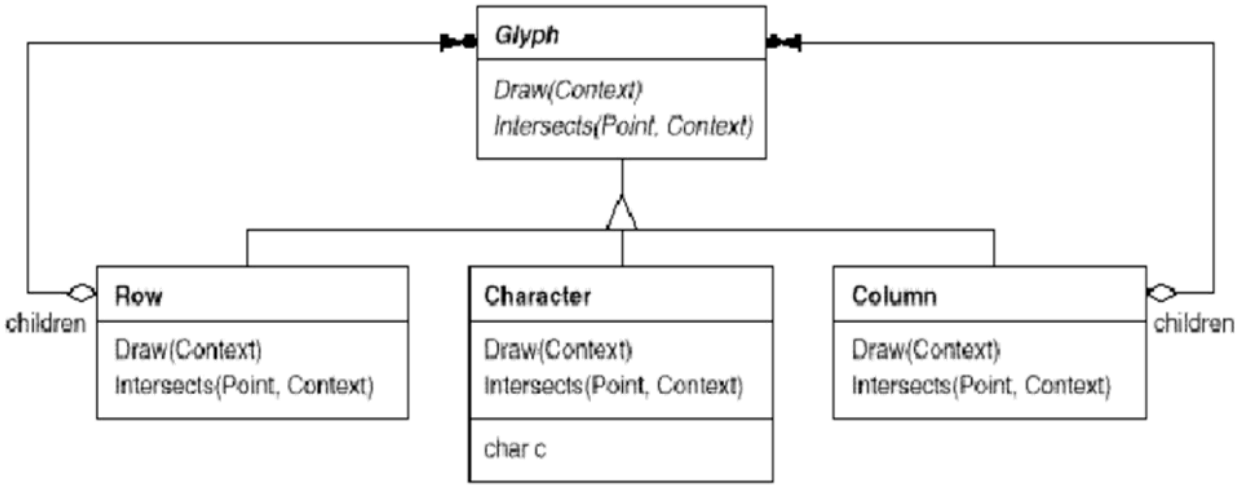
- Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.
- Example: most document editor implementations have text formatting and editing facilities that are modularized to some extent.
- Object-oriented document editors typically use objects to represent embedded elements like tables and figures.
- However, they usually stop short of using an object for each character in the document, even though doing so would promote flexibility at the finest levels in the application.
- Characters and embedded elements could then be treated uniformly with respect to how they are drawn and formatted.
- The following fig. shows how a document editor can use objects to represent characters.



- The drawback of such a design is its cost.
- Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time overhead.
- The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost.
- A **flyweight** is a shared object that can be used in multiple contexts simultaneously.
- Flyweights cannot make assumptions about the context in which they operate.
- The key concept here is the distinction between **intrinsic** and **extrinsic state.**
- Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight,,s context, thereby making it sharable.
- Extrinsic state depends on & varies with the flyweight's context and therefore can't be shared.
- Client objects are responsible for passing extrinsic state to the flyweight when it needs it.
- Logically there is an object for every occurrence of a given character in the document:





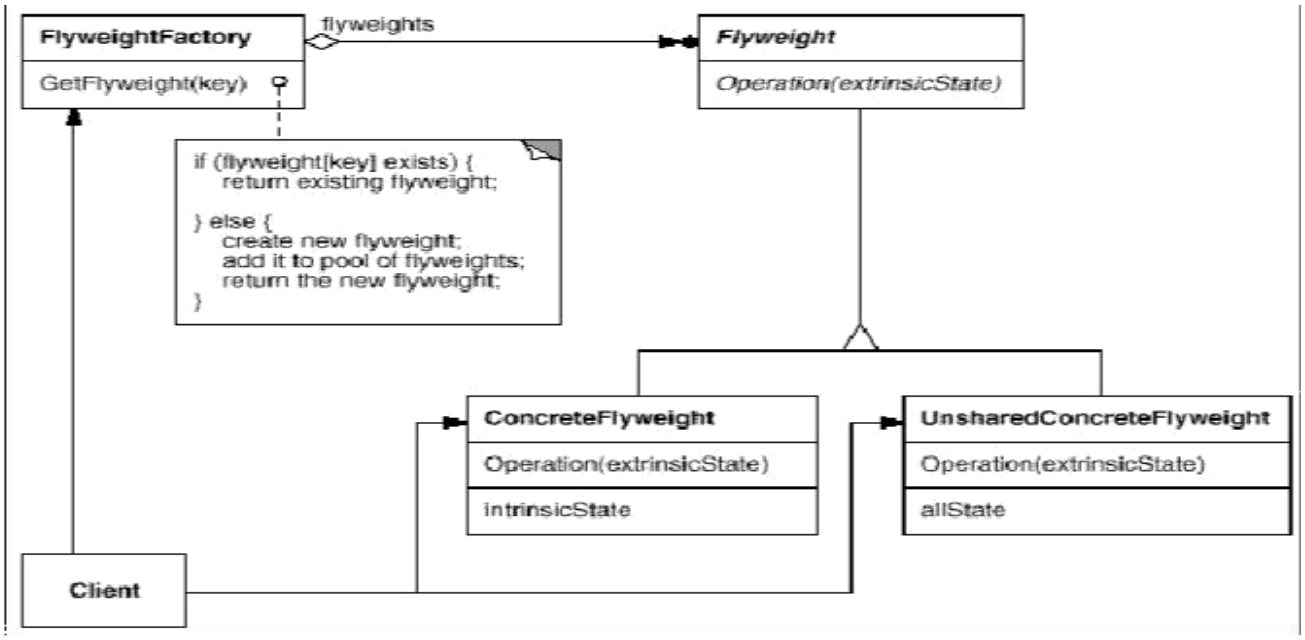


Applicability:

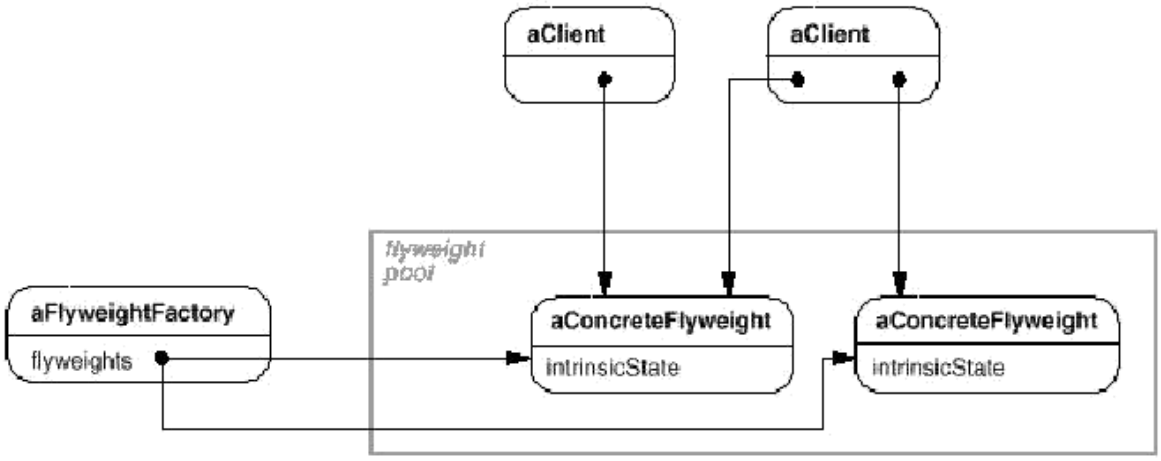
Apply the Flyweight pattern when all of the following are true:

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

Structure:



The following object diagram shows how flyweights are shared:



Participants:

- **Flyweight**
  - declares an interface through which flyweights can receive and act on extrinsic state.
- **ConcreteFlyweight** (Character)
  - implements the Flyweight interface and adds storage for intrinsic state, if any. Its object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.
- **UnsharedConcreteFlyweight**(Row, Column)
  - not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn‟t enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).
- **FlyweightFactory**
  - creates and manages flyweight objects.
  - ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.
- **Client**
  - maintains a reference to flyweight(s).
  - computes or stores the extrinsic state of flyweight(s).

Collaborations:

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

Consequences:

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

Storage savings are a function of several factors:

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

- The Flyweight pattern is often combined with the Composite pattern to represent a hierarchical structure as a graph with shared leaf nodes. A consequence of sharing is that flyweight leaf nodes cannot store a pointer to their parent.

Implementation:

Consider the following issues when implementing the Flyweight pattern:

1. ***Removing extrinsic state***.
   - The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects.
   - Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing.
   - Ideally, extrinsic state can be computed from a separate object structure, one with far smaller storage requirements.
2. ***Managing shared objects***.
   - Because objects are shared, clients shouldn't instantiate them directly.
   - Flyweight Factory lets clients locate a particular flyweight.

- Flyweight Factory objects often use an associative store to let clients lookup flyweights of interest.
- Sharability also implies some form of reference counting or garbage collection to reclaim a flyweight's storage when it's no longer needed.
- However, neither is necessary if the number of flyweights is fixed and small.
- In that case, the flyweights are worth keeping around permanently.

Sample Code:

```
class Glyph {
public:
    virtual ~Glyph();
    virtual void Draw(Window*, GlyphContext&);
    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);
    virtual void First(GlyphContext&);
    virtual void Next(GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);
    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);
protected:
    Glyph();
};
```

The Character subclassjust stores a character code:

```
class Character : public Glyph {
public:
    Character(char);
    virtual void Draw(Window*,
GlyphContext&); private:
    char _charcode;
};
```

```
class GlyphContext {
public:
    GlyphContext();
    virtual ~GlyphContext();
    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);
    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);
private:
    int _index;
    BTree* _fonts;
};
```

Known Uses :

- The concept of flyweight objects was first described and explored as a design technique in InterViews 3.0
- ET++ uses flyweights to support look-and-feel independence.

Related Patterns:

- The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes.
- It's often best to implement State and Strategy objects as Flyweights.

3.7 Proxy:

Intent:

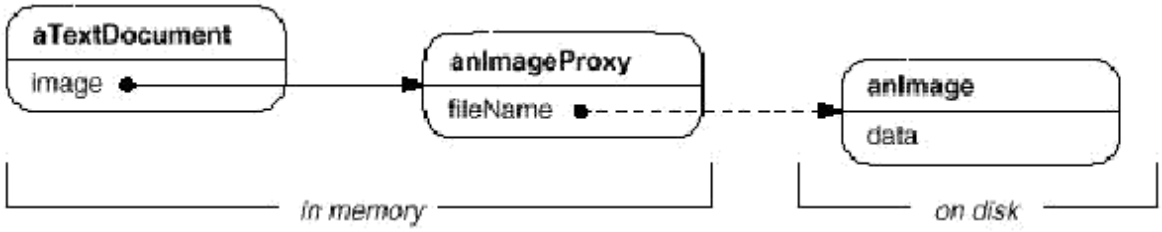
Provide a surrogate or placeholder for another object to control access to it.

Also Known As: Surrogate

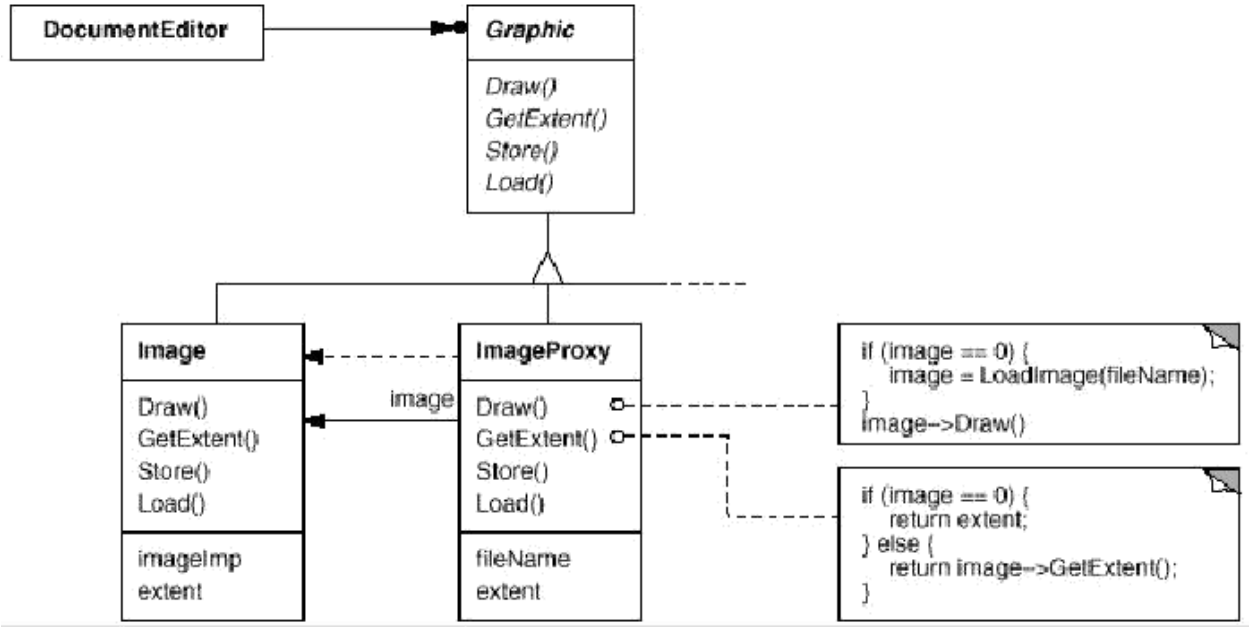
Motivation:

- One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.
- These constraints would suggest creating each expensive object on demand.

- Solution: 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 document editor accesses embedded images through 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.

- ImageProxy also stores the bounding box of the image and a reference to the real Image instance. This reference won't be valid until the proxy instantiates the real image.
- The Draw operation makes sure the image is instantiated before forwarding it the request.
- GetExtent forwards the request to the image only if it's instantiated; otherwise ImageProxy returns the extent it stores.

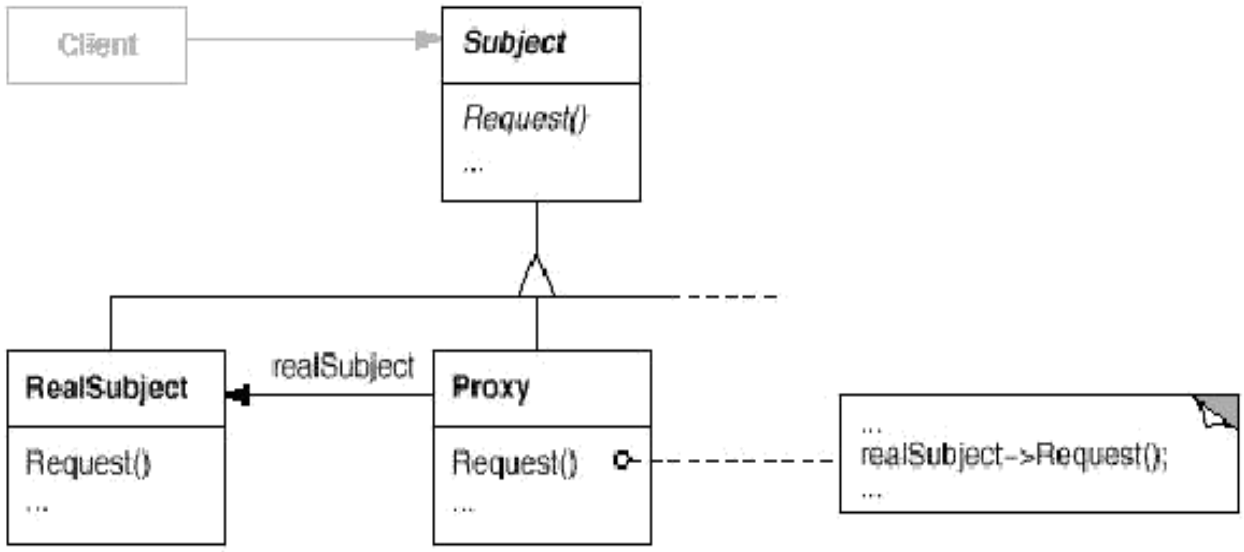
Applicability:

- Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer.

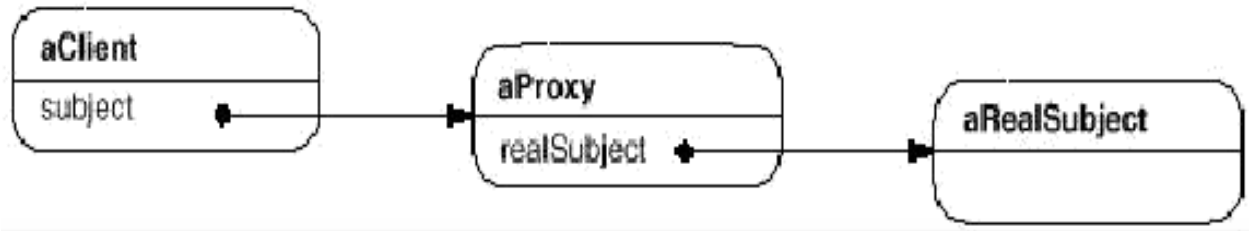
Here are several common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP uses the class NXProxy for this purpose. Coplien calls this kind of proxy an "Ambassador."
2. A **virtual proxy** creates expensive objects on demand. The ImageProxy described in the Motivation is an example of such a proxy.
3. A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights. For example, KernelProxies in the Choices operating system provide protected access to operating system objects.
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include
   - counting the number of references to the real object so that it can be freed automatically when there are no more references (also called smart-pointers).
   - loading a persistent object into memory when it's first referenced.
   - checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure:



Here's a possible object diagram of a proxy structure at run-time:



Participants:

- **Proxy** (ImageProxy)
  - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
  - provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
  - controls access to the real subject and may be responsible for creating and deleting it.
  - remote proxies are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
  - virtual proxies may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.

- protection proxies check that the caller has the access permissions required to perform a request.

- **Subject** (Graphic)
  - defines 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.

Collaborations:

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

Consequences:

- The Proxy pattern introduces a level of indirection when accessing an object.
- **The additional indirection has many uses, depending on the kind of proxy:**
  1. A remote proxy can hide the fact that an object resides in a different address space.
  2. A virtual proxy can perform optimizations such as creating an object on demand.
  3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.
- There's another optimization that the Proxy pattern can hide from the client.
- It's called copy-on-write, and it's related to creation on demand. Copying a large and complicated object can be an expensive operation.

Implementation:

1. **Overloading the member access operator in C++:** *C++* supports overloading operator ->, the member access operator. *T*he proxy behaves just like a pointer.
2. **Using doesNotUnderstand in Smalltalk:** Small talk provides a hook that tou can use to support automatic forwarding of requests. Small talk calls “doesNotUnderstand”: a Message when a client sends a message to a receiver that has no corresponding method. The proxy class can redefine doesNotUnderstand so that the message is forwarded to its subject.
3. **Proxy doesn't always have to know the type of real subject:** If a proxy class can deal with its subject solely through an abstract interface, then there is no need to make a proxy class for each real subject class. The proxy can deal with all real subject classes uniformly.

Sample Code:

1. *A virtual proxy.* The Graphic class defines the interface for graphical

objects: class Graphic {

public:

```
    virtual ~Graphic();
    virtual void Draw(const Point& at) = 0; virtual
    void HandleMouse(Event& event) = 0;
    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
protected:
    Graphic();
};
```

Known Uses:

- The virtual proxy example in the Motivation section is from the ET++ text building block classes.
- NEXTSTEP uses proxies (instances of class NXProxy) as local representatives for objects that may be distributed.

Related Patterns:

- Adapter: An adapter provides a different interface to the object it adapts, whereas a proxy provides the same interface as its subject.
- Decorator: A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

UNIT-4
Behavioral Patterns

Introduction:

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- They 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.
- Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.

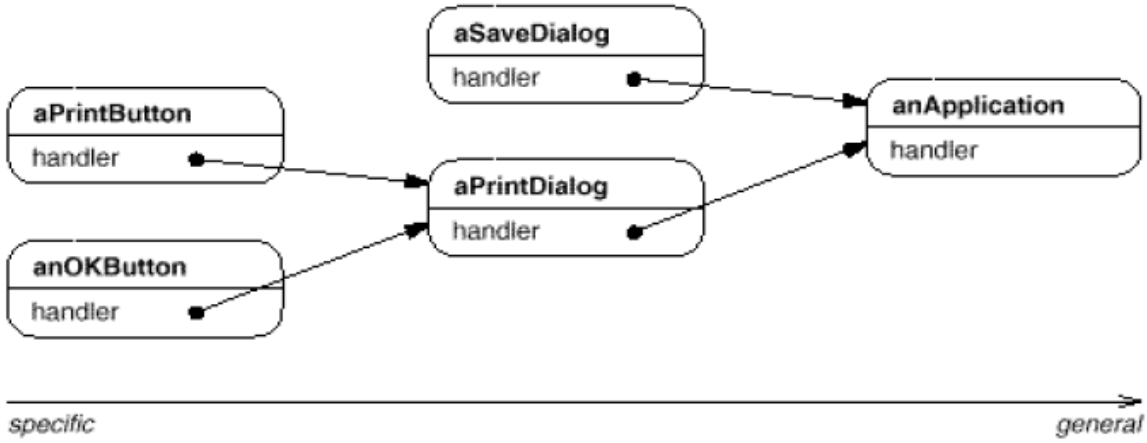
4.1Chain of Responsibility:

Intent:

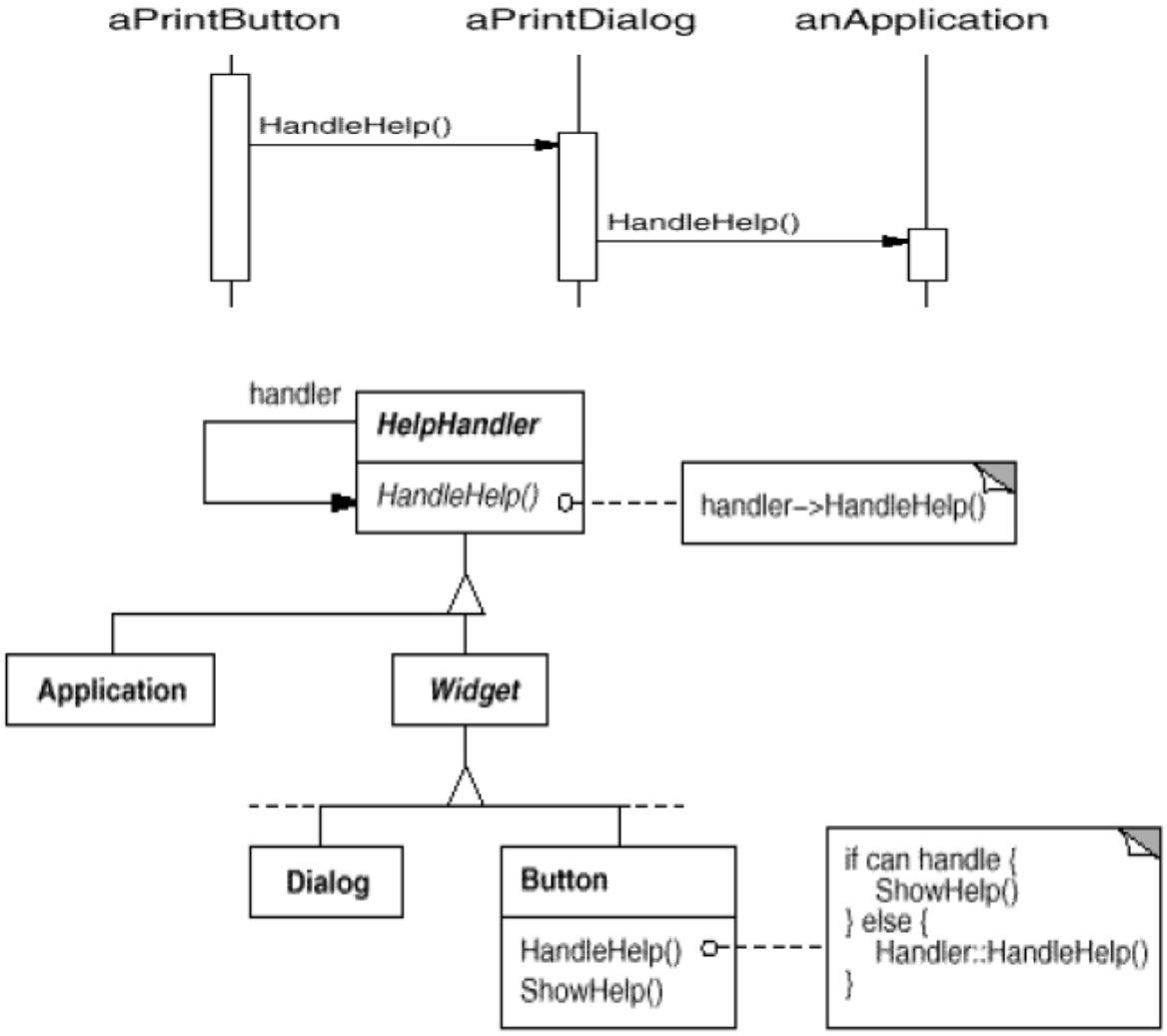
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Motivation:

- 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;
- **Problem:** the object that ultimately provides the help isn't known explicitly to the object (e.g., the button) that initiates the help request.
- **Solution:** decouple the button that initiates the help request from the objects that might provide help information.
- 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 following interaction diagramillustrates how the helprequest gets forwarded along the chain:

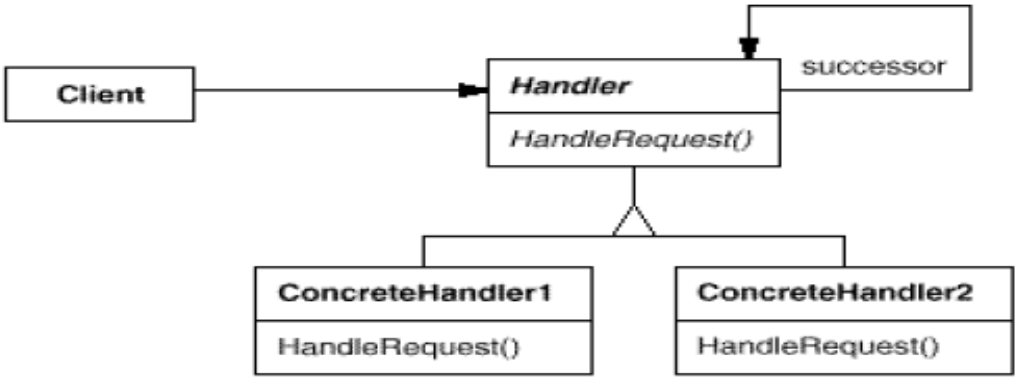


Applicability:

Use Chain of Responsibility when

- more than one object may handle a request, and the handler isn't known apriori. The handler should be ascertained automatically.
- you want to issue a request to one of several objects without specifying the receiver explicitly.
- the set of objects that can handle a request should be specified dynamically.

Structure:



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.

Collaborations:

When a client issues a request, the request propagates along the chain until a Concrete Handler object takes responsibility for handling it.

Consequences:

Chain of Responsibility has the following benefits and liabilities:

1. ***Reduced coupling.*** The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled "appropriately." Both the sender and receiver doesn't have to know about the chain's structure.
2. ***Added flexibility in assigning responsibilities to objects.*** *Chain of* Responsibility gives you added flexibility in distributing responsibilities among objects. You can add or change responsibilities for handling a request by adding to the chain at run-time.
3. ***Receipt isn't guaranteed***. Since a request has no explicit receiver, there's no guarantee it'll be handled—the request can fall off the end of the chain without ever being handled.

Implementation:

1. ***Implementing the successor chain.*** *There are two possible ways to implement* the successor chain:
    - a. Define new links (usually in the Handler, but ConcreteHandlers could define them instead).
    - b. Use existing links.
2. ***Connecting successors.*** *If there are no preexisting references for defining* a chain, then you'll have to introduce them yourself. The handler not only defines the interface for the request but usually maintains the successors as well. The handler provides a default implementation of Handle Request that forwards the request to the successor.
3. ***Representing requests.*** *Different options are available for representing* requests.
    - The request is a hard-coded operation invocation. This is convenient and safe , but you can forward only the fixed set of requests that the handler class defines.

- Use a single handler function that takes a request code as parameter. It supports open ended set of requests. The only requirement is that the sender and the receiver agree on how the request should be encoded.

4. ***Automatic forwarding in Smalltalk.*** *You can use the* ***doesNotUnderstand***mechanism in Smalltalk to forward requests. Messages that have no corresponding methods are trapped in the implementation of **doesNotUnderstand**, which can be overridden to forward the message to an object's successor.

Sample Code:

```
typedefint Topic;
const Topic NO_HELP_TOPIC = -1;

classHelpHandler {
public:
    HelpHandler(HelpHandler* = 0, Topic = NO_HELP_TOPIC);
    virtual bool HasHelp();
    virtual void SetHandler(HelpHandler*, Topic);
    virtual void HandleHelp();
private:
    HelpHandler* _successor;
    Topic _topic;
};

HelpHandler::HelpHandler (HelpHandler* h, Topic t): _successor(h), _topic(t) { }
boolHelpHandler::HasHelp () {
    return _topic != NO_HELP_TOPIC;
}
voidHelpHandler::HandleHelp () {
if (_successor != 0) {
successor->HandleHelp();
}
}
```

Known Uses:

- The Unidraw framework for graphical editors defines Command objects that encapsulate requests to Component and ComponentView objects.
- ET++ uses Chain of Responsibility to handle graphical update.

Related Patterns:

- Chain of Responsibility is often applied in conjunction with Composite.There, a component's parent can act as its successor.

4.2 Command:

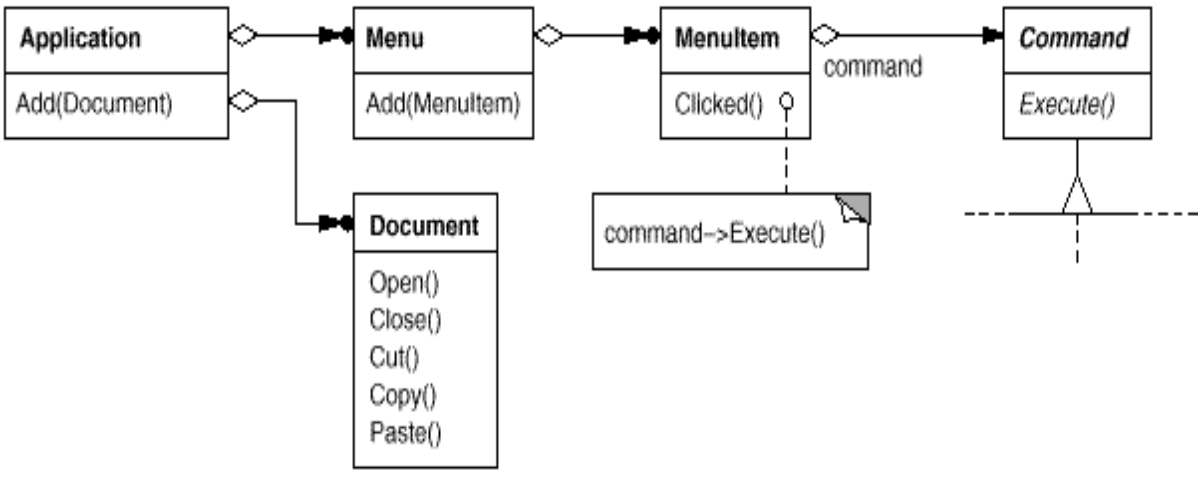
Intent:

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

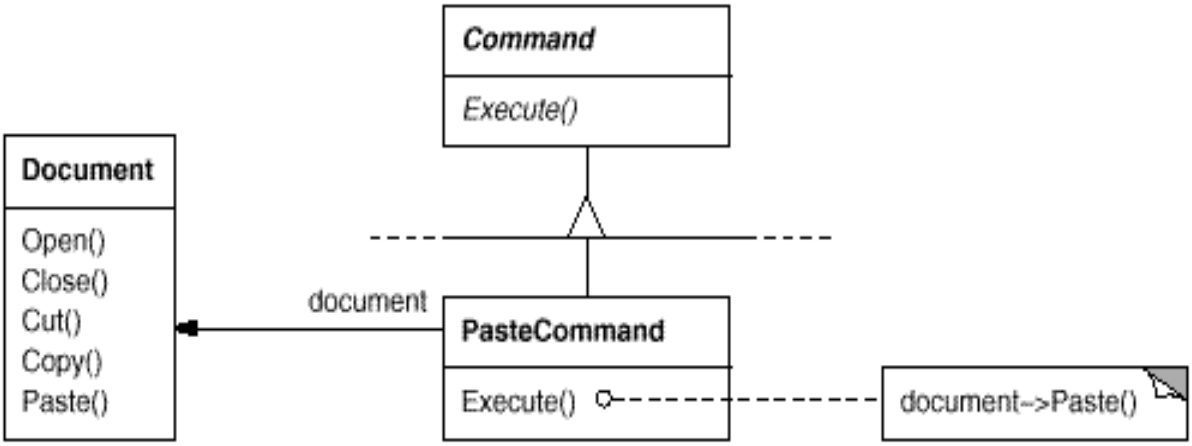
Also Known As: Action, Transaction

Motivation:

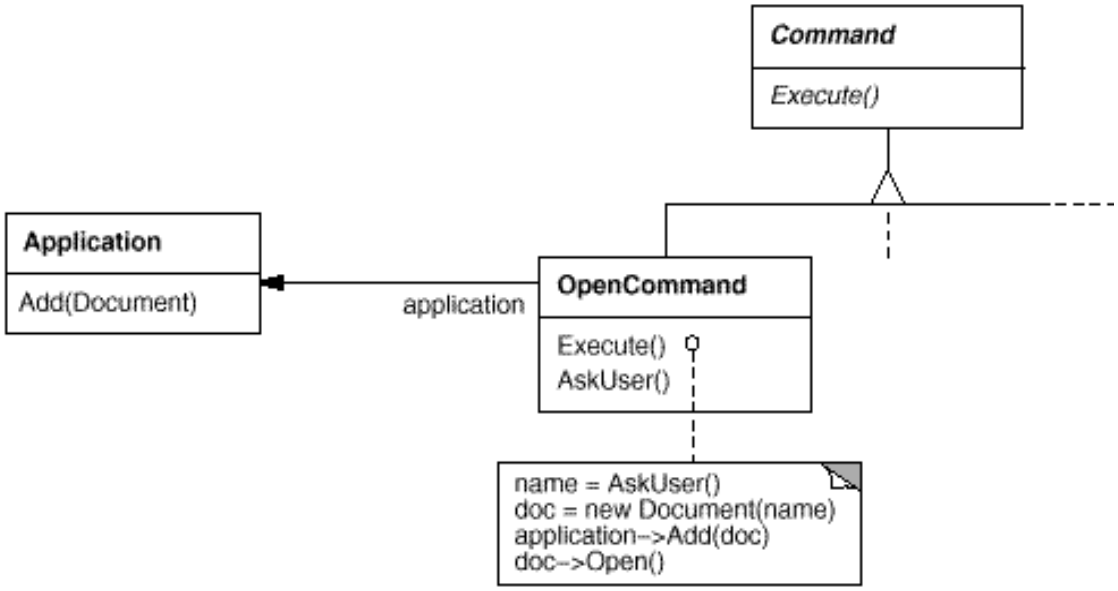
- Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
- The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object.
- This object can be stored and passed around like other objects.



For example, PasteCommand supports pasting text from the clipboardinto a Document.



OpenCommand's Execute operation is different: it prompts the userfor a documentname, creates a corresponding Document object, adds thedocument to the receivingapplication, and opens the document.



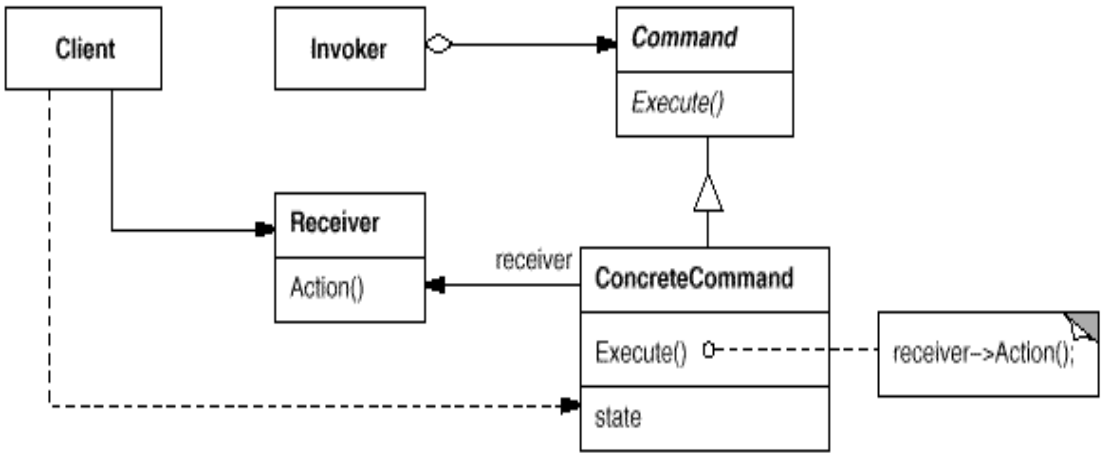
Applicability:

Use the Command pattern when you want to

- parameterize objects by an action to perform, as MenuItem objects did above. Commands are an object-oriented replacement for callbacks.
- specify, queue, and execute requests at different times. A Command object can have a lifetime independent of the original request.

- support undo. The Command's Execute operation can store state for reversing its effects in the command itself.
- support logging changes so that they can be reapplied in case of a system crash.
- structure a system around high-level operations built on primitives operations. Such a structure is common in information systems that support **transactions.** A transaction encapsulates a set of changes to data. The Command pattern offers a way to model transactions.

Structure:

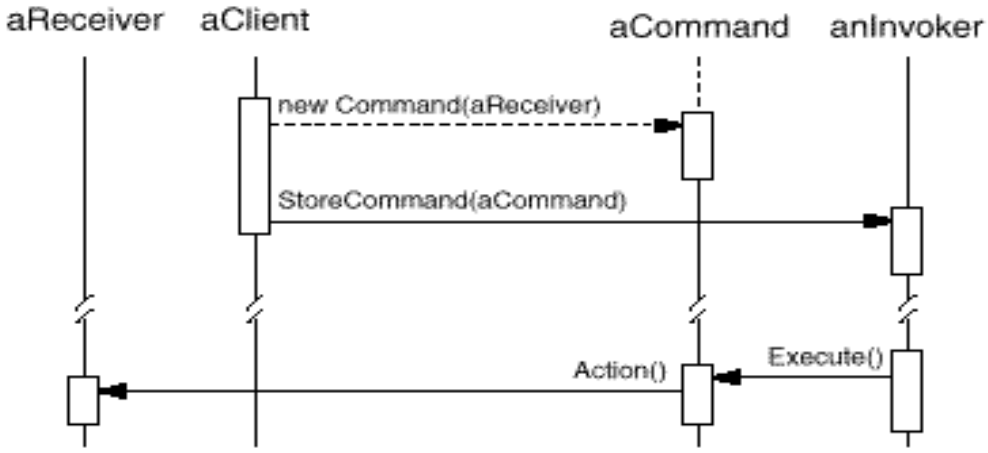


Participants:

- **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.
- **Receiver (Document, Application)**
  - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Collaborations:

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The invoker issues a request by calling Execute on the command. When commands are undoable, ConcreteCommand stores state for undoing the command prior to invoking Execute.
- The ConcreteCommand object invokes operations on its receiver to carryout the request.



Consequences:

The Command pattern has the following consequences:

1. Command decouples the object that invokes the operation from the one that knows how to perform it.
2. Commands are first-class objects. They can be manipulated and extended like any other object.
3. You can assemble commands into a composite command. An example is the MacroCommand class described earlier. In general, composite commands are an instance of the Composite pattern.
4. It's easy to add new Commands, because you don't have to change existing classes.

Implementation:

Consider the following implementation issues when applying the Command pattern:

1. ***How intelligent should a command be?*** A command can have a wide range of abilities.
    - It merely defines a binding between a receiver and the actions carry out the request.
    - It implements everything itself without delegating to a receiver at all.
2. ***Supporting undo and redo***. *Commands can support undo and redo capabilities* if they provide a way to reverse their execution

- A ConcreteCommand class might need to store additional state to do so. This state can include
    - the Receiver object, which actually carries out operations in response to the request,
    - the arguments to the operation performed on the receiver, and
    - any original values in the receiver that can change as a result of handling the request. The receiver must provide operations that let the command return the receiver to its prior state.

3. ***Avoiding error accumulation in the undo process***. *Hysteresis can be a problem* in ensuring a reliable, semantics-preserving undo/redo mechanism. Errors can be accumulate as commands are executed, unexecuted and re executed repeatedly so that application states eventually diverges from original values.
4. ***Using C++ templates***. For commands that (1) aren't undoable and (2) don‘t require arguments, we can use C++ templates to avoid creating a Command subclass for every kind of action and receiver.

Sample Code:

Theabstract Command class:

```
class Command {
public:
    virtual ~Command();
    virtual void Execute() = 0;
```

```
protected:
    Command();
};

classOpenCommand : public Command {
public:
    OpenCommand(Application*);
    virtual void Execute();
protected:
    virtualconst char* AskUser();
private:
    Application* _application;
    char* _response;
};

OpenCommand::OpenCommand (Application* a) {
_application = a;
}

voidOpenCommand::Execute () {
const char* name = AskUser();
if (name != 0) {
Document* document = new Document(name);
_application->Add(document);
document->Open();
}
}
```

Known Uses:

- MacApppopularized the notion of commands for implementingundoable operations.
- ET++, InterViews, and Unidrawalsodefine classes that follow theCommand pattern.
- The THINK class library also uses commands to support undoable actions.
- Coplien describes how to implement **functors**, objects that are functions, in C++.

Related Patterns:

- A Composite can be used to implement MacroCommands.
- A Memento can keep state the command requires to undo its effect.
- A command that must be copied before being placed on the history list acts as a Prototype.

4.3 Interpreter:

Intent:

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

Motivation:

- For example, searching for strings that match a pattern is a common problem. Regular expressions area standard language for specifying patterns of strings.
- 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.

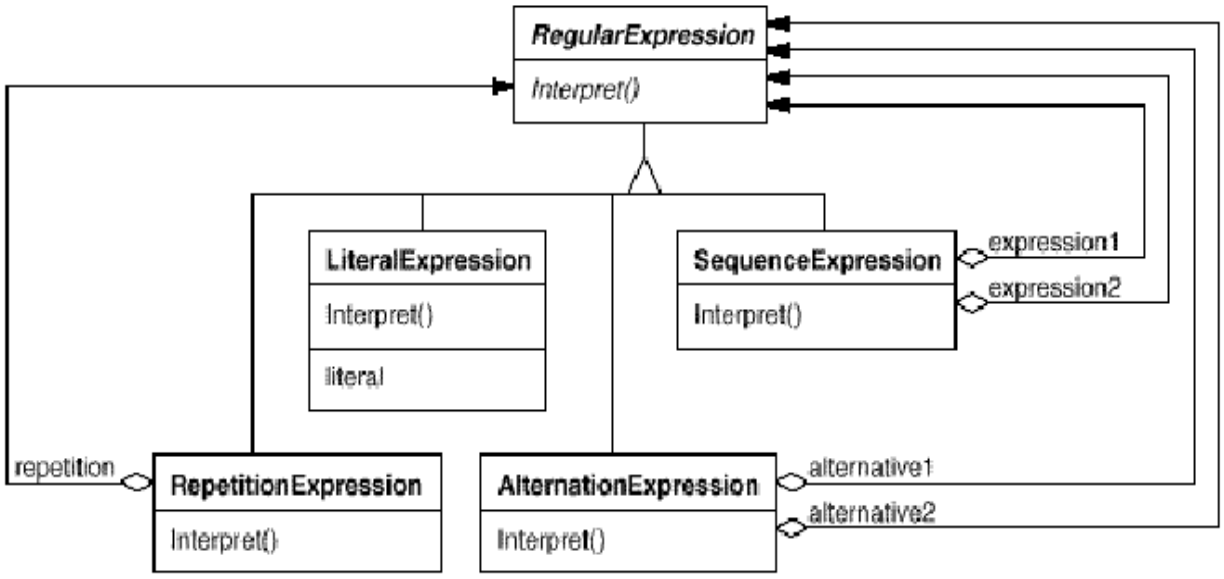
  expression ::= literal | alternation | sequence | repetition | '(' expression ')'

  alternation ::= expression '|' expression

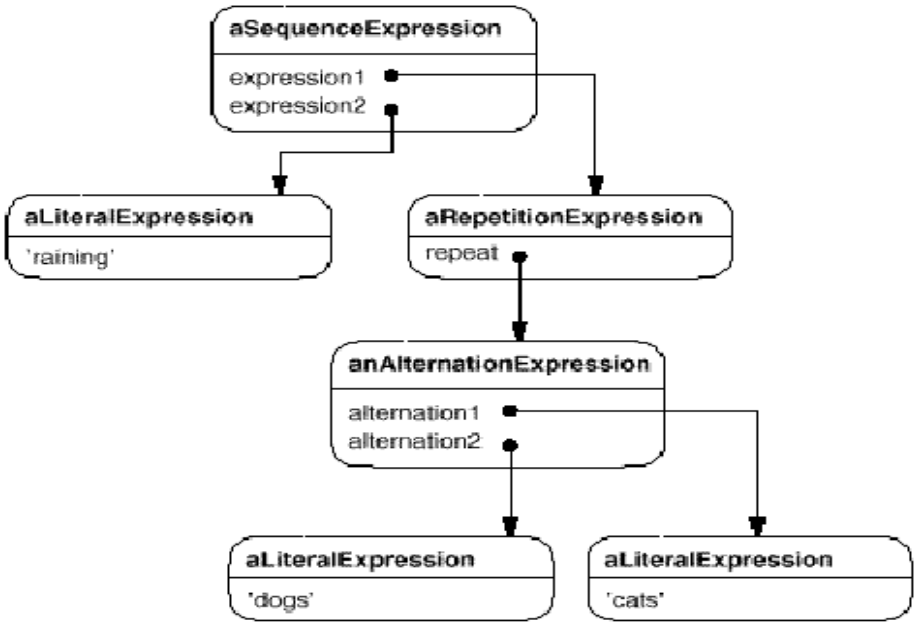
  sequence ::= expression '&' expression

  repetition ::= expression '*'

  literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*



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) ***



- We can create an interpreter for these regular expressions by defining the Interpret operation on each subclass of Regular Expression.
- Each subclass of Regular Expression implements Interpret to match the next part of the input string based on the current context.

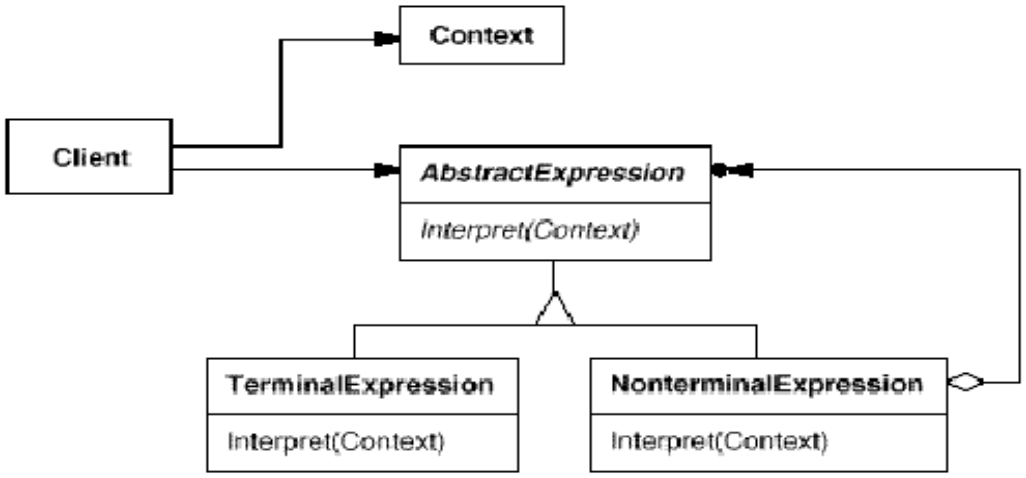
- For example,
  - Literal Expression will check if the input matches the literal it defines,
  - Alternation Expression will check if the input matches any of its alternatives,
  - Repetition Expression will check if the input has multiple copies of expression it repeats, and so on.

Applicability:

The Interpreter pattern works best when

- The grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable.
- Efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form.

Structure:



Participants:

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

- **Terminal Expression** (Literal Expression)
  - Implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in a sentence.
- **Non terminal Expression** (Alternation Expression, Repetition Expression, Sequence Expressions)
  - one such class is required for every rule $R ::= R_1 R_2 ... R_n$ in the grammar.
  - maintains instance variables of type Abstract Expression for each of the symbols R_1 through R_n
  - Implements an Interpret operation for non terminal 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 Non terminal Expression and Terminal Expression classes.
  - Invokes the Interpret operation.

Collaborations:

- The client builds (or is given) the sentence as an abstract syntax tree of Non terminal Expression and Terminal Expression instances. Then the client initializes the context and invokes the Interpret operation.
- Each Non terminal Expression node defines Interpret in terms of Interpret on each sub expression. The Interpret operation of each Terminal Expression defines the base case in the recursion.
- The Interpret operations at each node use the context to store and access the state of the interpreter.

Consequences:

The Interpreter pattern has the following benefits and liabilities:

- ***It's easy to change and extend the grammar*.** Because the pattern uses classes to represent grammar rules, you can use inheritance to change or extend the grammar.
- ***Implementing the grammar is easy, too***. Classes defining nodes in the abstract syntax tree have similar implementations. These classes are easy to write, and their generation can be automated with a compiler generators.
- ***Complex grammars are hard to maintain*.** The Interpreter pattern defines at least one class for every rule in the grammar. Hence grammars containing many rules can be hard to manage and maintain.
- ***Adding new ways to interpret expressions***. The Interpreter pattern makes it easier to evaluate an expression in a new way. Using the Visitor pattern to avoid changing the grammar classes.

Implementation:

The Interpreter and Composite patterns share many implementation issues.

The following issues are specific to Interpreter:

- ***Creating the abstract syntax tree*.** The Interpreter pattern doesn't explain how to create an abstract syntax tree. It doesn't address the parsing. The abstract syntax tree can be created by a table driven parser or directly by the client.
- ***Defining the Interpret operation***. You don't have to define the Interpret operation in the expression classes. For example, a grammar for programming language will have many operations on abstract syntax tree, such as type-checking, optimization, code generation and so on.
- ***Sharing terminal symbols with the Flyweight pattern*.** Grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol. For example, each program variable will appear in many places throughout the code.

Sample Code:

The regular expression matcher tests whether a string is in the language defined by the regular expression. The regular expression is defined by the following grammar:

expression::= literal | alternation | sequence | repetition | '(' expression ')'

alternation ::= expression '|' expression

sequence ::= expression '&' expression

repetition ::= expression 'repeat'

literal ::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

For example, the regular expression ***(('dog ' | 'cat ') repeat & 'weather')*** matches the input string *"dog dog cat weather"*.

Known Uses:

- Widely used in compilers implemented with object-oriented languages (Smalltalk compilers).
- SPECTalk uses the pattern to interpret descriptions of input file formats.
- The QOCA constraint-solving toolkit uses it to evaluate constraints.

Related Patterns:

- **Flyweight** shows how to share terminal symbols within the abstract syntax tree.
- **Iterator**: The interpreter can use an Iterator to traverse the structure.
- **Visitor** can be used to maintain the behaviour in each node in the abstract syntax tree in one class.
- **Composite** : The abstract syntax tree is an instance of the Composite pattern.

4.4 Iterator:

Intent:

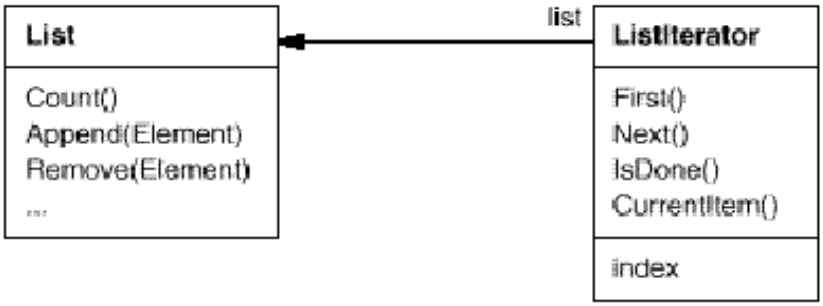
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Also Known As: Cursor

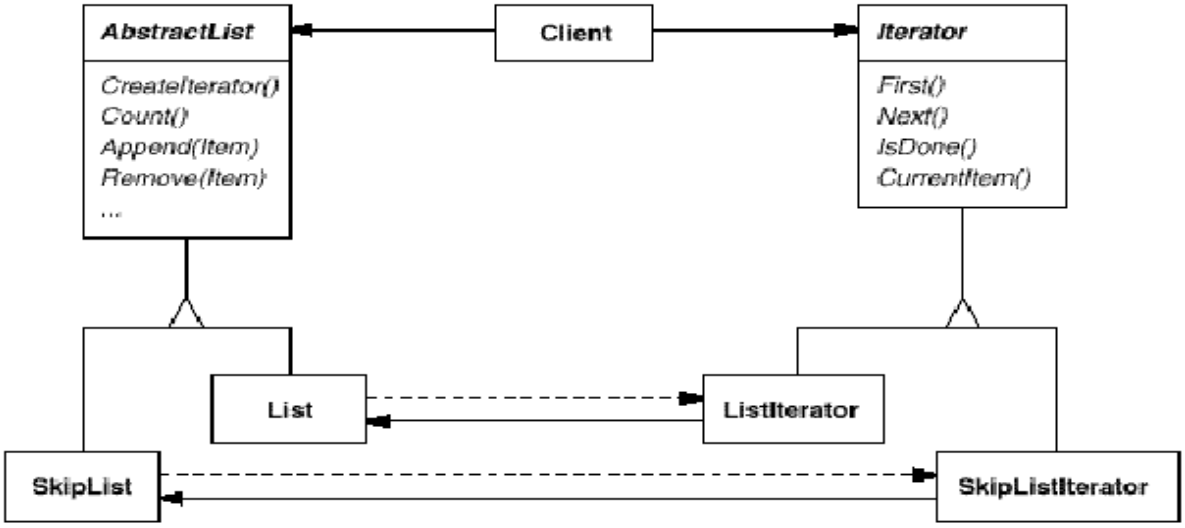
Motivation:

- An aggregate object such as a list should give you a way to access its elements without exposing its internal structure.
- you might want to traverse the list in different ways, depending on what you want to accomplish.
- But you probably don't want to bloat the List interface with operations for different traversals.
- You might also need to have more than one traversal pending on the same list.

- The **Iterator** pattern lets you do all this. 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.
- Forexample, 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.
- We can change the aggregate class without changing client code by generalizing the iterator concept to support ***polymorphic iteration***.



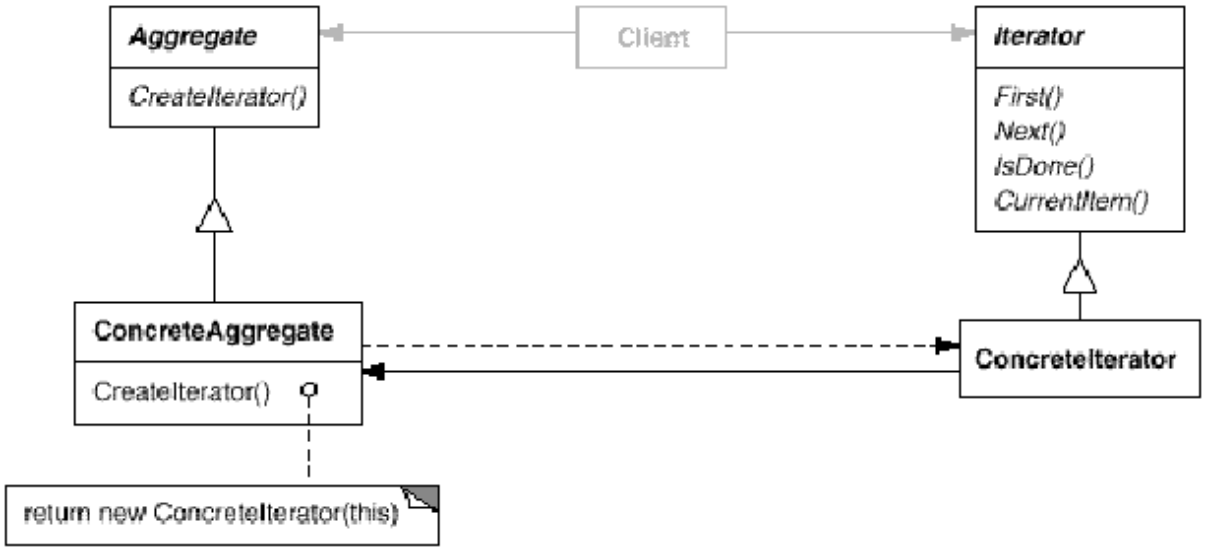
Applicability:

Use the Iterator pattern

- To access an aggregate object's contents without exposing its internal representation.
- To support multiple traversals of aggregate objects.

- To provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

Structure:



Participants:

- **Iterator**
  - Defines an interface for accessing and traversing elements.
- **Concrete Iterator**
  - 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.
- **Concrete Aggregate**
  - Implements the Iterator creation interface to return an instance of the proper Concrete Iterator.

Collaborations:

- A Concrete Iterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.

Consequences:

The Iterator pattern has three important consequences:

1. ***It supports variations in the traversal of an aggregate.*** Complex aggregates may be traversed in many ways. For example, code generation and semantic checking involve traversing parse trees. Code generation may traverse the parse tree inorder or preorder.
2. ***Iterators simplify the Aggregate interface.*** Iterator's traversal interface avoids the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
3. ***More than one traversal can be pending on an aggregate***. An iterator keeps track of its own traversal state. Therefore you can have more than one traversal in progress at once.

Implementation:

Iterator has many implementation variants and alternatives.

1. ***Who controls the iteration*****?** the iterator or the client that uses the iterator.

    * When the client controls the iteration, the iterator is called an **external iterator**, and when the iterator controls it, the iterator is an **internal iterator**.
    * External Iterators are more flexible than internal Iterators. It's easy to compare two collections for equality with an external iterator.
    * Internal iterators are easier to use, because they define the iteration logic for us.

2. ***Who defines the traversal algorithm?*** The iterator or the aggregate.
    * The aggregate might define the traversal algorithm and use the iterator to store just the state of the iteration. We call this kind of iterator a cursor, since it only points to the current position in the aggregate. A client will invoke the Next operation on the aggregate with the cursor as an argument, and the Next operation will change the state of the cursor.
    * If the iterator is responsible for the traversal algorithm, then it's easy to use different iteration algorithms on the same aggregate, and it can also be easier to reuse the same algorithm on different aggregates.

- Putting the traversal algorithm in the iterator violates the encapsulation of the aggregate.

3. ***How robust is the iterator?*** It can be dangerous to modify an aggregate while you're traversing it.
   - A **robust iterator** ensures that insertions and removals won't interfere with traversal, and it does it without copying the aggregate.
4. ***Additional Iterator operations.*** The minimal interface to Iterator consists of the operations First, Next, IsDone, and CurrentItem.
   - Some additional operations might prove useful.
   - Ordered aggregates can have a Previous operation that positions the iterator to the previous element.
   - A SkipTo operation is useful for sorted or indexed collections. SkipTo positions the iterator to an object matching specific criteria.
5. ***Using polymorphic iterators in C++.*** Polymorphic iterators have their cost.
   - They require the iterator object to be allocated dynamically by a factory method.
   - Hence they should be used only when there's a need for polymorphism.
   - Otherwise use concrete iterators, which can be allocated on the stack.
6. ***Iterators may have privileged access.*** An iterator can be viewed as an extension of the aggregate that created it. The iterator and the aggregate are tightly coupled.
   - such privileged access can make defining new traversals difficult, since it'll require changing the aggregate interface to add another friend.
7. ***Iterators for composites.***
   - External iterators can be difficult to implement over recursive aggregate structures like those in the Composite pattern, because a position in the structure may span many levels of nested aggregates.
   - Therefore an external iterator has to store a path through the Composite to keep track of the current object.
   - Sometimes it's easier just to use an internal iterator. It can record the current position simply by calling itself recursively, thereby storing the path implicitly in the call stack.

8. *Null iterators.*

- A **NullIterator** is a degenerate iterator that's helpful for handling boundary conditions.
- A NullIterator is always done with traversal; i.e., its **IsDone** operation always evaluates to true.
- NullIterator can make traversing tree-structured aggregates easier.
- At each point in the traversal, we ask the current element for an iterator for its children.
- Aggregate elements return a concrete iterator as usual. But leaf elements return an instance of NullIterator.

Sample Code:

1. List and Iterator interfaces.

```
template<class Item>
class List {
    public:
        List(long size = DEFAULT_LIST_CAPACITY);
        long Count() const;
        Item&Get(long index) const;
    // ...
};

template<class Item>
class Iterator {
    public:
        virtual void First() = 0;
        virtual void Next() = 0;
        virtual bool IsDone() const = 0;
        virtual Item CurrentItem() const = 0;
    protected:
        Iterator();
};
```

2. Iterator subclass implementations.ListIterator is a subclass of Iterator.

```
template<class Item>
classListIterator : public Iterator<Item> {
    public:
        ListIterator(const List<Item>* aList);
        virtual void First();
        virtual void Next();
        virtual bool IsDone() const;
        virtual Item CurrentItem() const;
    private:
        const List<Item>* _list;
        long _current;
};
```

3. Using the iterators.
4. Avoiding commitment to a specific list implementation.
5. Making sure iterators getdeleted.
6. Aninternal ListIterator.

Known Uses:

- Iterators are common in object-oriented systems.
- Most collection class libraries provide iterators in one form or another.
- Smalltalk also includes a set of Stream classes that support an iterator-like interface.
- ReadStream is essentially an Iterator, and it can act as an external iterator for all the sequential collections.
- Polymorphic iterators and the clean-up Proxy described earlier are provided by the ET++ container classes.
- The Unidraw graphical editing framework classes use cursor-based iterators.
- ObjectWindows 2.0 provides a class hierarchy of iterators for containers.

Related Patterns:

- **Composite**: Iterators are often applied to recursive structures such as Composites.

- **Factory Method**: Polymorphic iterators depend on factory methods to instantiate the appropriate Iterator subclass.
- **Memento** is often used in conjunction with the Iterator pattern. An iterator can use a memento to capture the state of an iteration. The iterator stores the memento internally.
- Booch refers to external and internal iterators as active and passive iterators, respectively.
- The terms "active" and "passive" describe the role of the client, not the level of activity in the iterator.
- Cursors are a simple example of the Memento pattern.
- The Traverse operation in these examples is a **TemplateMethod** with primitive operations TestItem and ProcessItem.

4.5 Mediator:

Intent:

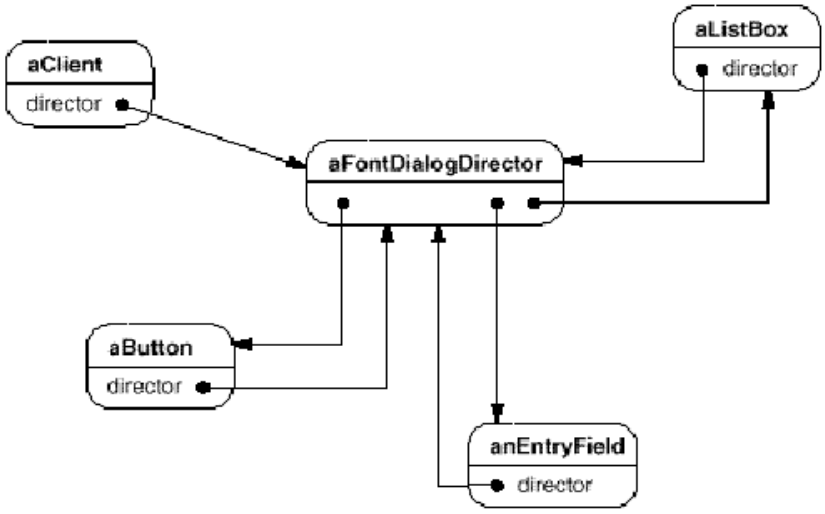
Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Motivation:

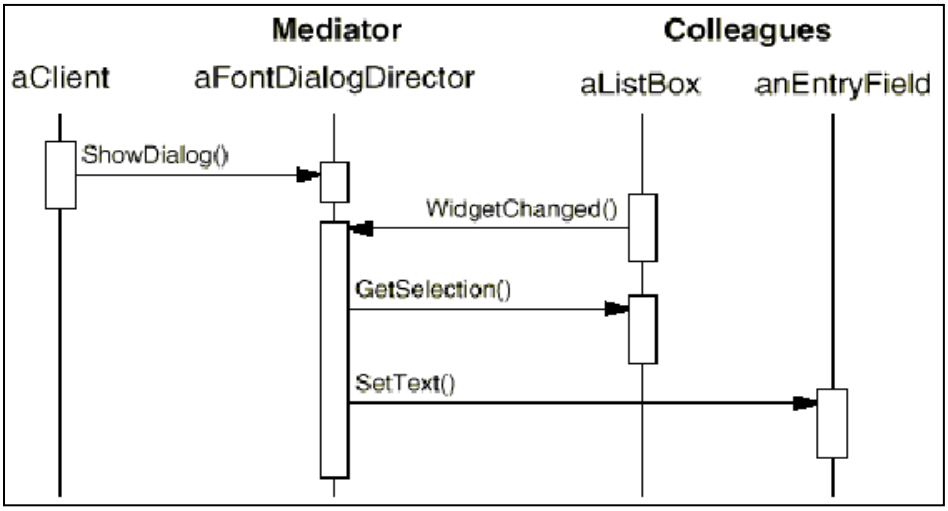
- Object-oriented design encourages the distribution of behavior among objects.
- Such distribution can result in an object structure with many connections between objects; in the worst case, every object ends up knowing about every other.
- Though partitioning a system into many objects generally enhances reusability, increasing inter connections tend to reduce it again.
- As a result, you may be forced to define many subclasses to customize the system's behavior.
- **Example:** consider the implementation of dialog boxes in a graphical user interface.
- A dialog box uses a window to present a collection of widgets such as buttons, menus, and entry fields, as shown here:

- Often there are dependencies between the widgets in the dialog.
- For example, a button gets disabled when a certain entry field is empty.
- Different dialog boxes will have different dependencies between widgets.
- Customizing them individually by subclassing will be tedious, since many classes are involved.
- You can avoid these problems by encapsulating collective behavior in a separate mediator object.
- 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 inter connections.

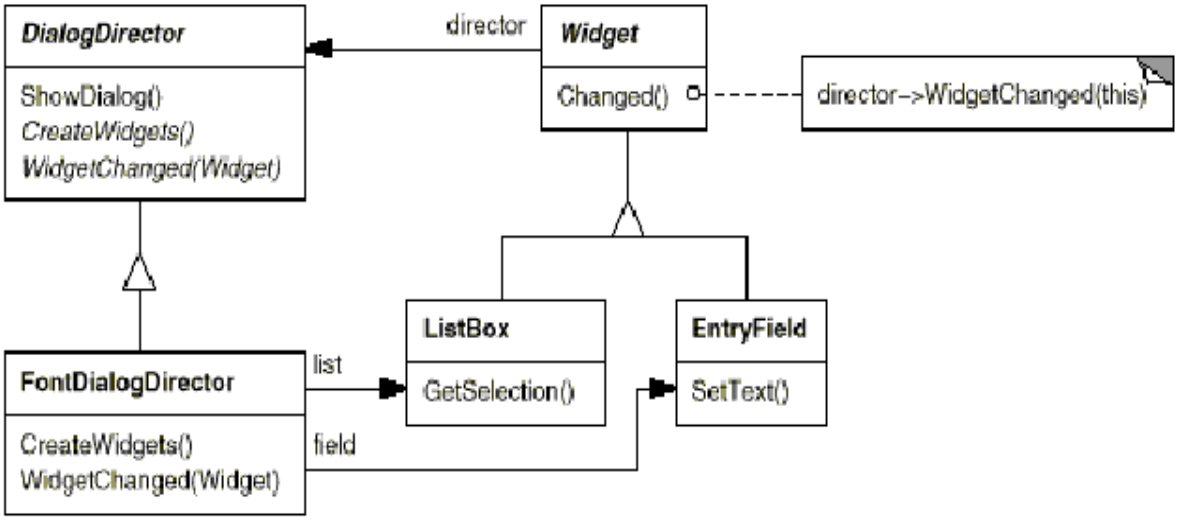
For example, FontDialogDirector can be the mediatorbetween the widgets in a dialogbox. A FontDialogDirectorobjectknowsthe widgetsina dialog and coordinates theirinteraction. It acts asa hub of communication forwidgets:



The following interaction diagram illustrates how the objects cooperate tohandlea change in a list box's selection:



Here's how the FontDialogDirector abstractioncanbeintegrated into aclass library:

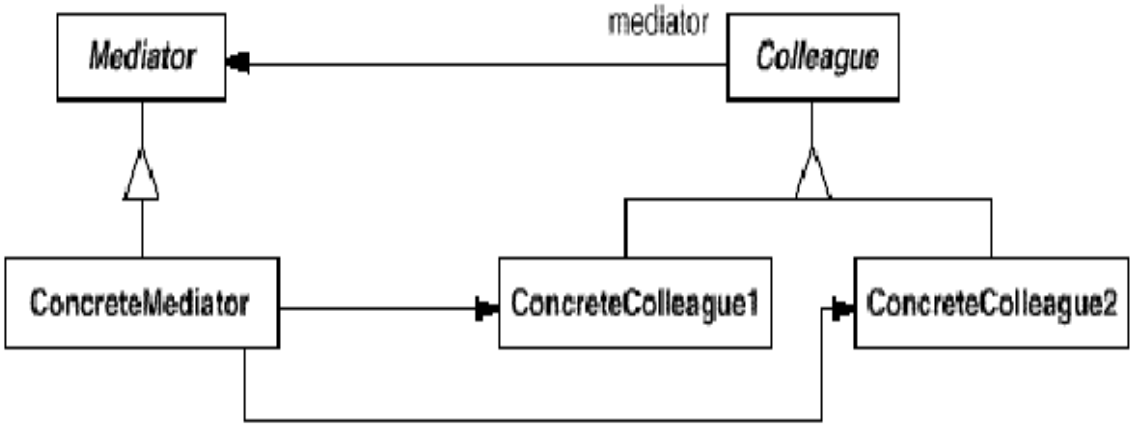


Applicability:

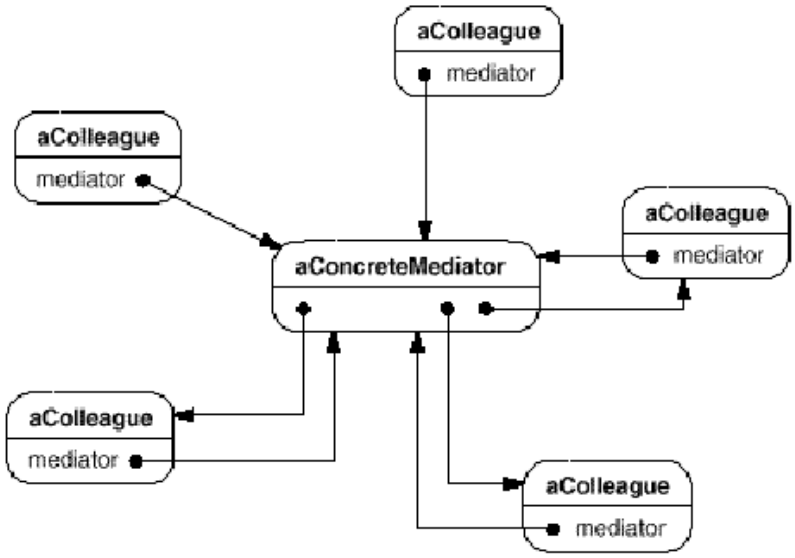
Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways. The resulting inter dependencies are unstructured and difficult to understand.
- reusing an object is difficult because it refers to and communicates with many other objects.
- abehavior that's distributed between several classes should be customizable without a lot of subclassing.

Structure:



A typical object structure might look like this:



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.

Collaborations:

- Colleagues send and receive requests from a Mediator object. The mediator implements the cooperative behavior by routing requests between the appropriate colleague(s).

Consequences:

The Mediator pattern has the following benefits and drawbacks:

1. ***It limits subclassing.***
   - A mediator localizes behaviour that otherwise would be distributed among several objects.

- Changing this behaviour requires subclassing Mediator only; Colleague classes can be reused as is.

2. ***It decouples colleagues.***
   - A mediator promotes loose coupling between colleagues.
   - You can vary and reuse Colleague and Mediator classes independently.
3. ***It simplifies object protocols.***
   - A mediator replaces many-to-many interactions with one-to-many interactions between the mediator and its colleagues.
   - One-to-many relationships are easier to understand, maintain, and extend.
4. ***It abstracts how objects cooperate.***
   - Making mediation an independent concept and encapsulating it in an object lets you focus on how objects interact apart from their individual behavior.
   - That can help clarify how objects interact in a system.
5. ***It centralizes control.***
   - The Mediator pattern trades complexity of interaction for complexity in the mediator.
   - Because a mediator encapsulates protocols, it can become more complex than any individual colleague.
   - This can make the mediator itself a monolith that's hard to maintain.

Implementation:

The following implementation issues are relevant to the Mediator pattern:

1. ***Omitting the abstract Mediator class.***
   - There's no need to define an abstract Mediator class when colleagues work with only one mediator.
2. ***Colleague-Mediator communication.***
   - Colleagues have to communicate with their mediator when an event of interest occurs. One approach is to implement the Mediator as an Observer using the Observer pattern.
   - Colleague classes act as Subjects, sending notifications to the mediator whenever they change state.

- The mediator responds by propagating the effects of the change to other colleagues.
- Another approach defines a specialized notification interface in Mediator that lets colleagues be more direct in their communication.

Sample Code :

We'll use a DialogDirector to implement the font dialog box shown inthe Motivation. Theabstractclass DialogDirector definesthe interface for directors.

```
classDialogDirector {
public:
        virtual ~DialogDirector();
        virtual void ShowDialog();
        virtual void WidgetChanged(Widget*) = 0;
protected:
        DialogDirector();
        virtual void CreateWidgets() = 0;
};
```

Widget is the abstractbase class for widgets. Awidget knows its director.

```
class Widget {
public:
        Widget(DialogDirector*);
        virtual void Changed();
        virtual void HandleMouse(MouseEvent& event);
        // ...
private:
        DialogDirector* _director;
};
```

Changed calls the director's WidgetChangedoperation. Widgets call WidgetChanged ontheirdirector toinform itofa significant event.

```
void Widget::Changed ()
{       _director->WidgetChanged(this);  }
```

Known Uses:

- Both **ET++** and the **THINK C** class library use director-like objects in dialogs as mediators between widgets.
- The application architecture of Smalltalk/V for Windows is based on a mediator structure.
- Unidraw drawing framework uses a class called **CSolver** to enforce connectivity constraints between "connectors."CSolver is a mediator between connectors. It solves the connectivity constraints and updates the connector's positions to reflect them.

Related Patterns:

- **Facade** differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface.Its protocol is unidirectional; but Mediator's protocol ismultidirectional.
- Colleagues can communicate with the mediator using the **Observer** pattern.

4.6 Memento:

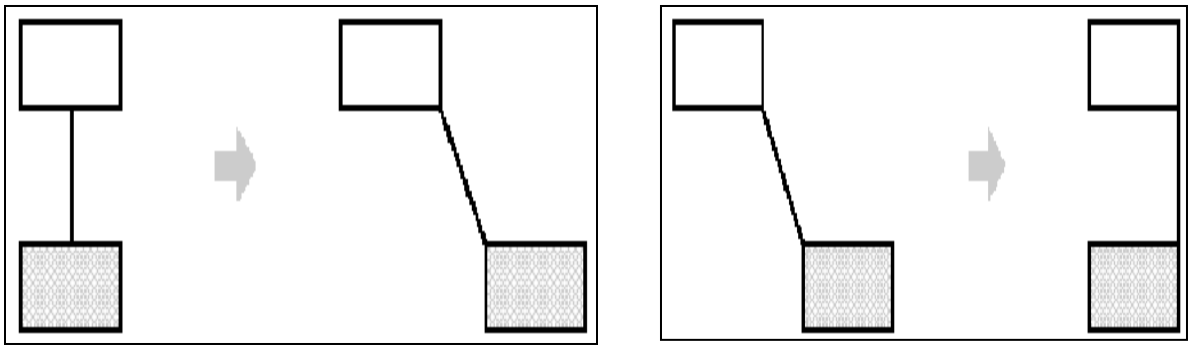
Intent:

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

Motivation:

- Sometimes it's necessary to record the internal state of an object.
- This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.
- But objects normally encapsulate some or all of their state, making it inaccessible to other objects and impossible to save externally.
- Exposing this state would violate encapsulation, which can compromise the application's reliability and extensibility.
- 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.



- A well-known way to maintain connectivity relationships between objects is with a constraint-solving system. We can encapsulate this functionality in a **ConstraintSolver** object.
- ConstraintSolver records connections as they are made and generates mathematical equations that describe them.
- An obvious way to undo a move operation is to store the original distance moved and move the object back an equivalent distance.
- However, this does not guarantee all objects will appear where they did before.
- In general, the ConstraintSolver's public interface might be insufficient to allow precise reversal of its effects on other objects.
- 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.
- The Constraint Solver can act as an originator.
- The following sequence of events characterizes the undo process:

1. The editor requests a memento from the Constraint Solver as aside-effect of the move operation.
2. The Constraint Solver creates and returns a memento, an instance of a class Solver State in this case. A Solver State memento contains data structures that describe the current state of the Constraint Solver's internal equations and variables.

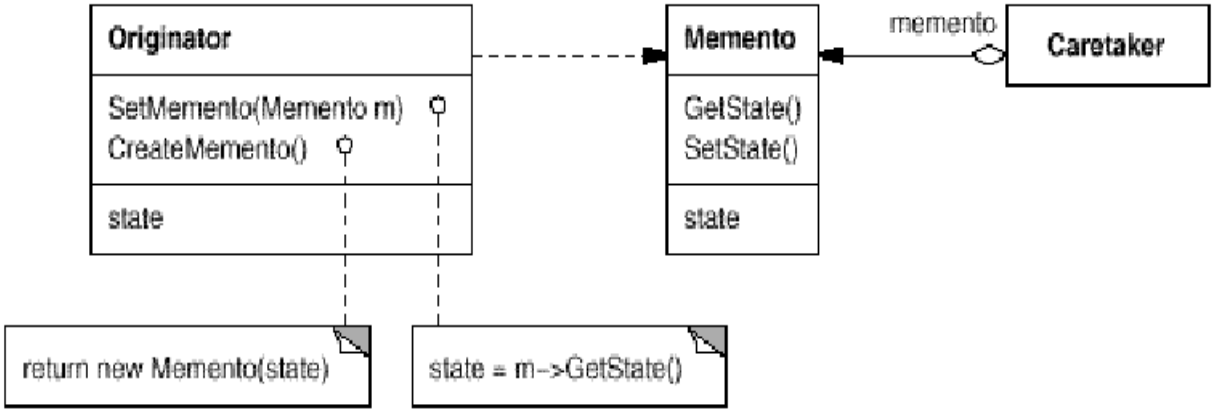
3. Later when the user undoes the move operation, the editor gives the Solver State back to the Constraint Solver.
4. Based on the information in the Solver State, the Constraint Solver changes its internal structures to return its equations and variables to their exact previous state.

Applicability:

Use the Memento pattern when

- a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, and
- a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.

Structure:



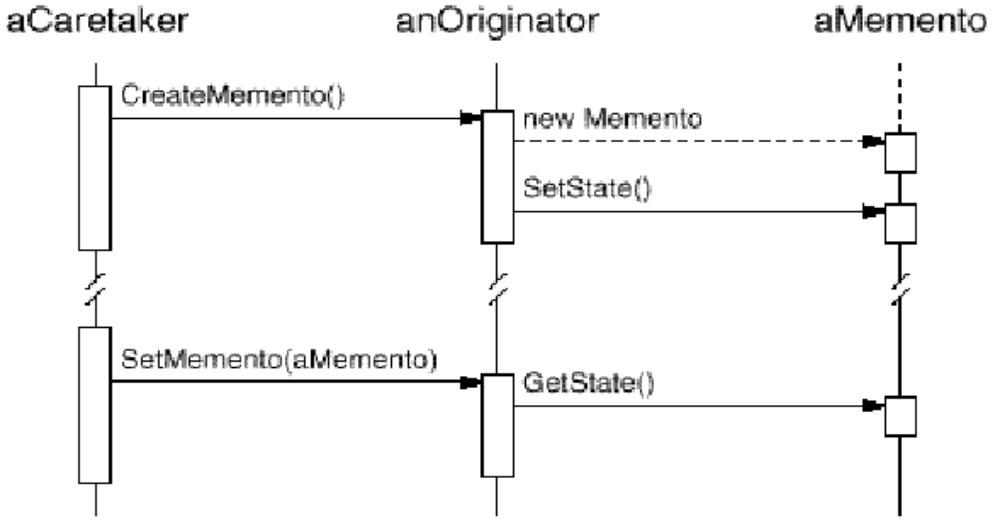
Participants:

- **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 preference.
  - protects against access by objects other than the originator. Mementos have effectively two interfaces. Narrow interface seen by Caretaker—it can only pass the memento to other objects. Wide interface-seen by Originator, one that lets it access all the data necessary to restore itself to its previous state. 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 safe keeping.
  - never operates on or examines the contents of a memento.

Collaborations:

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator, as the following interaction diagram illustrates:



- Sometimes the caretaker won't pass the memento back to the originator, because the originator might never need to revert to an earlier state.
- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

Consequences:

The Memento pattern has several consequences:

1. ***Preserving encapsulation boundaries.***
   - Memento avoids exposing information that only an originator should manage but that must be stored nevertheless outside the originator.
   - The pattern shields other objects from potentially complex Originator internals, thereby preserving encapsulation boundaries.

2. ***It simplifies Originator.***
    - In other encapsulation-preserving designs, Originator keeps the versions of internal state that clients have requested.
    - That puts the entire storage management burden on Originator.
    - Having clients manage the state they ask for simplifies Originator and keeps clients from having to notify originators when they're done.
3. ***Using mementos might be expensive.***
    - Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough.
    - Unless encapsulating and restoring Originator state is cheap, the pattern might not be appropriate.
4. ***Defining narrow and wide interfaces.***
    - It may be difficult in some languages to ensure that only the originator can access the memento's state.
5. ***Hidden costs in caring for mementos.***
    - A caretaker is responsible for deleting the mementos it cares for.
    - However, the caretaker has no idea how much state is in the memento.
    - Hence an otherwise lightweight caretaker might incur large storage costs when it stores mementos.

Implementation:

Here are two issues to consider when implementing the Memento pattern:

1. ***Language support.***
    - Mementos have two interfaces: a wide one for originators and a narrow one for other objects.
    - Ideally the implementation language will support two levels of static protection. C++ lets you do this by making the Originator a friend of Memento and making Memento's wide interface private. Only the narrow interface should be declared public.

2. ***Storing incremental changes.***

- When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just the incremental change to the originator's internal state.
- That means mementos can store just the incremental change that a command makes rather than the full state of every object they affect.

Sample Code:

The C++ code given here illustrates the ConstraintSolver example discussed earlier.

```
class Graphic;
// base class for graphical objects in the graphical editor
classMoveCommand {
public:
        MoveCommand(Graphic* target, const Point& delta);
        void Execute();
        voidUnexecute();
private:
        ConstraintSolverMemento* _state;
        Point _delta;
        Graphic* _target;
};
```

The connection constraints are established by the classConstraintSolver. Its key member functionisSolve, which solves the constraints registered withthe AddConstraint operation.

```
classConstraintSolver {
public:
    staticConstraintSolver* Instance();
    void Solve();
    voidAddConstraint( Graphic* startConnection, Graphic* endConnection);
    voidRemoveConstraint( Graphic* startConnection, Graphic* endConnection );
    ConstraintSolverMemento* CreateMemento();
    voidSetMemento(ConstraintSolverMemento*);
```

```
private:
    // nontrivial state and operations for enforcing
    // connectivity semantics  };

classConstraintSolverMemento {
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();
    // private constraint solver state
};
```

Known Uses :

- Unidrawsupports for connectivity through its CSolver class.
- Collections in Dylan provide an iteration interface that reflects the Memento pattern.

  The memento-based iteration interface has two interesting benefits:

  1. More than one state can work on the same collection.
  2. It doesn't require breaking a collection's encapsulation to support iteration. The memento is only interpreted by the collection itself; no one else has access to it.
- The QOCA constraint-solving toolkit stores incremental information in mementos. Clients can obtain a memento that characterizes the current solution to a system of constraints. The memento contains only those constraint variables that have changed since the last solution.

Related Patterns:

- **Command:** Commands can use mementos to maintain state for undoable operations.
- **Iterator:** Mementos can be used for iteration as described earlier.

4.7 Observer:

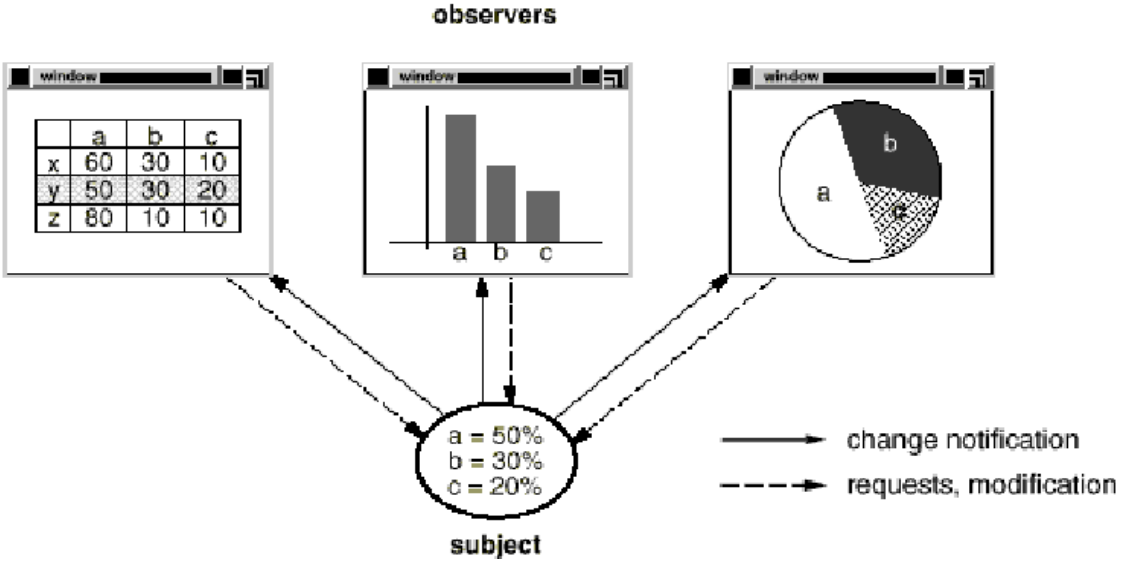
Intent:

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

Motivation:

- A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.
- For example, many GUI toolkits separate the presentational aspects of the user interface from the underlying application data.
- Classes defining application data and presentations can be reused independently. They can work together, too.



- 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.
- This behavior implies that the spreadsheet and bar chart are dependent on the data object and therefore should be notified of any change in its state.
- And there's no reason to limit the number of dependent objects to two; there may be any number of different user interfaces to the same data.
- 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.

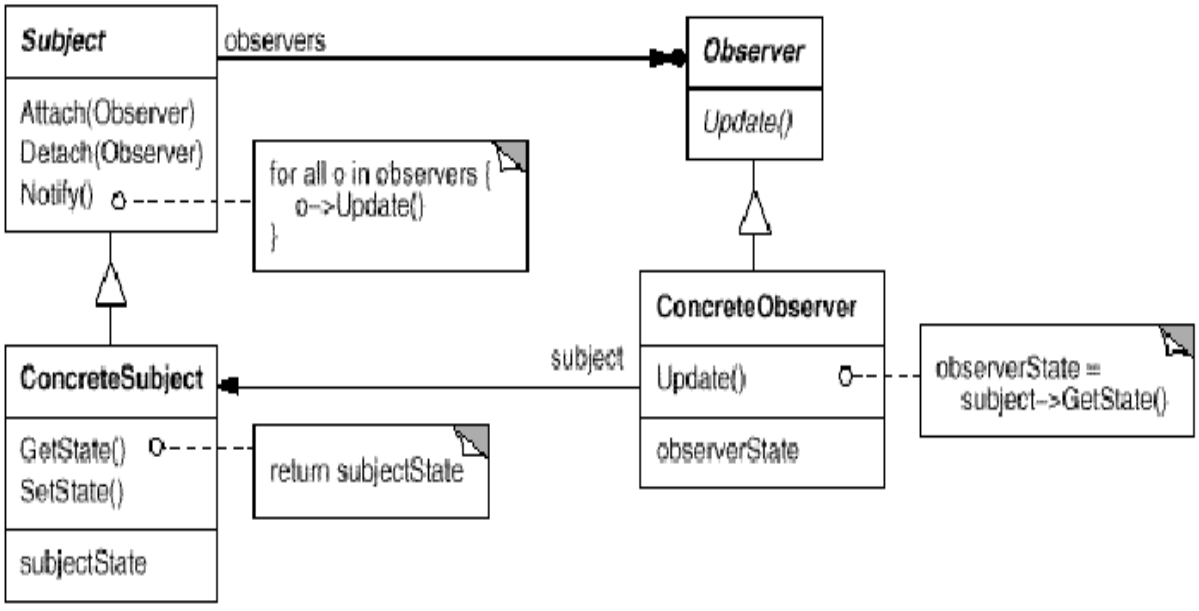
- All observers are notified whenever the subject undergoes a change in state.
- In response, each observer will query the subject to synchronize its state with the subject's state.
- This kind of interaction is also known as **publish-subscribe**.
- The subject is the publisher of notifications. It sends out these notifications without having to know who its observers are.
- Any number of observers can subscribe to receive notifications.

Applicability:

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Structure:

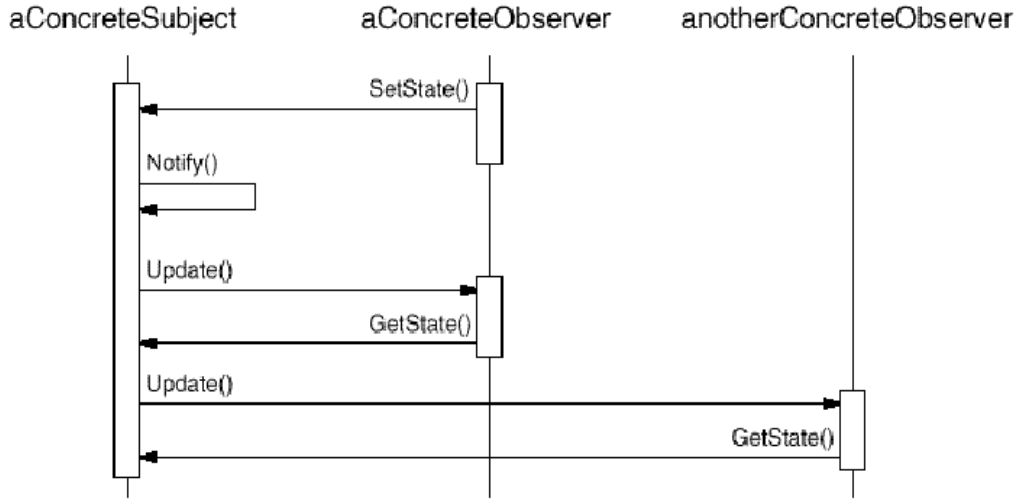


Participants:

- **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.

Collaborations:

- ConcreteSubject notifies its observers whenever a change occurs that could make its observer's state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.



- The Observer object that initiates the change request postpones its update until it gets a notification from the subject.
- Notify is not always called by the subject. It can be called by an observer or by another kind of object entirely.

Consequences:

- The Observer pattern lets you vary subjects and observers independently.
- You can reuse subjects without reusing their observers, and vice versa.
- It lets you add observers without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following:

1. ***Abstract coupling between Subject and Observer.***
   - The subject doesn't know the concrete class of any observer. Thus the coupling between subjects and observers is abstract and minimal.
   - Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system. A lower-level subject can communicate and inform a higher-level observer.
2. ***Support for broadcast communication.***
   - Unlike an ordinary request, the notification that a subject sends needn't specify its receiver.
   - The notification is broadcast automatically to all interested objects that subscribed to it.
   - The subject doesn't care how many interested objects exist; its only responsibility is to notify its observers.
   - This gives you the freedom to add and remove observers at any time.
3. ***Unexpected updates.***
   - Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.

Implementation:

Several issues related to the implementation of the dependency mechanism are:

1. ***Mapping subjects to their observers.***
   - The simplest way for a subject to keep track of the observers it should notify is to store references to them explicitly in the subject.

- Such storage may be too expensive when there are many subjects and few observers.
- One solution is to trade space for time by using an associative look-up to maintain the subject-to-observer mapping.
- Thus a subject with no observers does not incur storage overhead.
- On the other hand, this approach increases the cost of accessing the observers.

2. *Observing more than one subject.*

- In some situations an observer depends on more than one subject.
- Example: a spreadsheet may depend on more than one data source.
- It's necessary to extend the Update interface in such cases to let the observer know which subject is sending the notification.
- The subject can simply pass itself as a parameter in the Update operation, there byletting the observer to know which subject to examine.

3. *Who triggers the update?*

- The subject and its observers rely on the notification mechanism to stay consistent.
- But what object actually calls Notify to trigger the update? Here are two options:

a. Have state-setting operations on Subject call Notify after they change the subject's state.

   Advantage: clients don't have to remember to call Notify on the subject.

   Disadvantage: several consecutive operations will cause several consecutive updates, which may be inefficient.

b. Make clients responsible for calling Notify at the right time.

   Advantage: the client can wait to trigger the update until after a series of state changes has beenmade, thereby avoiding needless intermediate updates.

   Disadvantage: clients have an added responsibility to trigger the update. That makes errors more likely, since clients might forget to call Notify.

4. *Dangling references to deleted subjects.*

- Deleting a subject should not produce dangling references in its observers.
- One way to avoid dangling references is to make the subject notify its observers as it is deleted so that they can reset their reference to it.

- In general, simply deleting the observers is not an option, because other objects may reference them, or they may be observing other subjects as well.

5. ***Making sure Subject state is self-consistent before notification.***
   - It's important to make sure Subject state is self-consistent before calling Notify, because observers query the subject for its current state in the course of updating their own state.
   - This self-consistency rule is easy to violate unintentionally when Subject subclass operations call inherited operations.
6. ***Avoiding observer-specific update protocols:***
   - The **push** and **pull** models.
   - Implementations of the Observer pattern often have the subject broadcast additional information about the change.
   - The subject passes this information as an argument to Update.
   - The amount of information may vary widely.
   - **Push model**: the subject sends observers detailed information about the change, whether they want it or not.
   - **Pull model:** the subject sends nothing but the most minimal notification, and observers ask for details explicitly thereafter.
7. ***Specifying modifications of interest explicitly.***
   - You can improve update efficiency by extending the subject's registration interface to allow registering observers only for specific events of interest.
   - One way to support this uses the notion of aspects for Subject objects.

     void Subject::Attach(Observer, Aspect& interest);*

     void Observer::Update(Subject, Aspect& interest);*
8. ***Encapsulating complex update semantics.***
   - When the dependency relationship between subjects and observers is particularly complex, an object that maintains these relationships might be required. We call such an object a **ChangeManager**.
   - Its purpose is to minimize the work required to make observers reflect a change in their subject.
   - **ChangeManager** has three responsibilities:

1. It maps a subject to its observers and provides an interface to maintain this mapping. This eliminates the need for subjects to maintain references to their observers and vice versa.
2. It defines a particular update strategy.
3. It updates all dependent observers at the request of a subject.

9. *Combining the Subject and Observer classes.*

- Class libraries written in languages that lack multiple inheritances (like Smalltalk) generally don't define separate Subject and Observer classes but combine their interfaces in one class.
- That lets you define an object that acts as both a subject and an observer without multiple inheritances.
- In Smalltalk, for example, the Subject and Observer interfaces are defined in the root class Object, making them available to all classes.

Sample Code:

Anabstract class definestheObserver interface:

```
class Subject;


class Observer {
public:
    virtual ~ Observer();
    virtual void Update(Subject* theChangedSubject) = 0;
protected:
    Observer();
};
```

This implementationsupports multiple subjects for eachobserver.Thesubject passed to the Update operation lets the observerdetermine which subject changed when it observes more than one.

Similarly, an abstractclass defines the Subject interface:

```
class Subject {
public:
    virtual ~Subject();
    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
```

```
protected:
    Subject();
    private:
    List<Observer*> *_observers;
};
void Subject::Attach (Observer* o) {        _observers->Append(o);  }
void Subject::Detach (Observer* o) {        _observers->Remove(o);  }
void Subject::Notify () {
ListIterator<Observer*>i(_observers);
for (i.First(); !i.IsDone(); i.Next()) {
i.CurrentItem()->Update(this);
}
}
```

ClockTimer is a concrete subject for storing andmaintaining the time of day. Itnotifies itsobserversevery second.

```
classClockTimer : public Subject {
public:
        ClockTimer();
        virtualintGetHour();
        virtualintGetMinute();
        virtualintGetSecond();
        void Tick();
};
```

Known Uses:

- *Smalltalk Model/View/Controller* (MVC): MVC's Model class plays the role of Subject, while View is the base class for observers.
- *ET++*, and the *THINK class library* provide a general dependency mechanism.
- Other user interfaces like *InterViews*, the *AndrewToolkit*, and *Unidraw* employ this pattern.

Related Patterns:

- **Mediator**: By encapsulating complex update semantics, the Change Manager acts as mediator between subjects and observers.
- **Singleton**: The Change Manager may use the Singleton pattern to make it unique and globally accessible.

UNIT-5
Behavioral Patterns Part-II

5.1State:

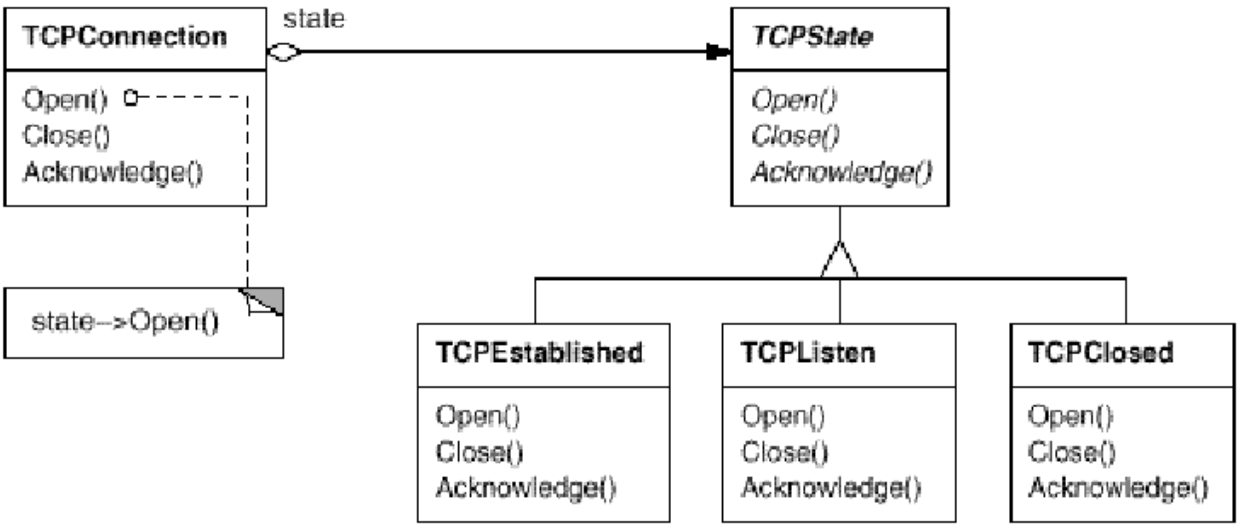
Intent:

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

Motivation:

- 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.
- 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.



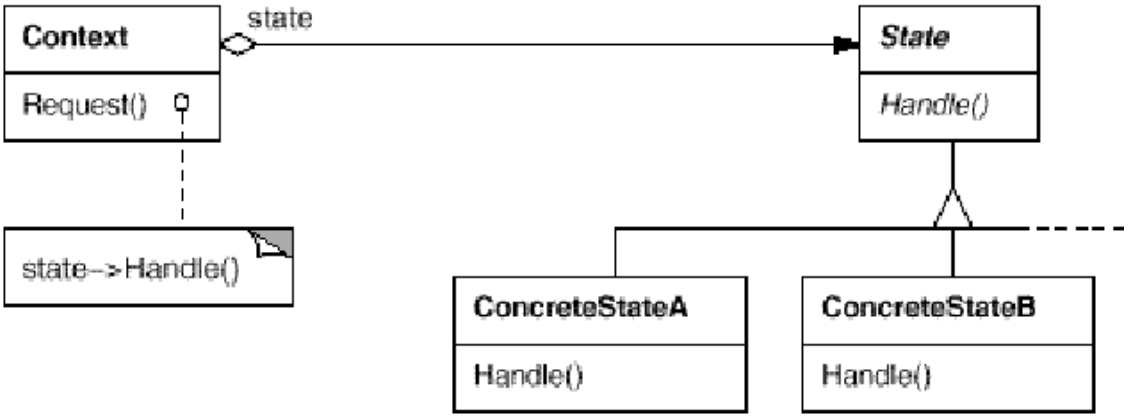
- Whenever the connection changes state, the TCPConnection object changes the state object it uses.
- When the connection goes from established to closed, for example, TCPConnection will replace its TCPEstablished instance with a TCPClosed instance.

Applicability:

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern puts each branch of the conditional in a separate class.

Structure:



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.

Collaborations:

- Context delegates state-specific requests to the current ConcreteState object.
- A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.
- Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.
- Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

Consequences:

The State pattern has the following consequences:

1. *It localizes state-specific behavior and partitions behavior for different states.*
    - The State pattern puts all behavior associated with a particular state into one object.
    - Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclasses.
    - This increases the number of classes and is less compact than a single class.
    - An alternative is to use data values to define internal states and have Context operations check the data explicitly.
    - Adding a new state could require changing several operations, which complicates maintenance.
2. *It makes state transitions explicit.*
    - When an object defines its current state solely in terms of internal data values, its state transitions have no explicit representation; they only show up as assignments to some variables.
    - Introducing separate objects for different states makes the transitions more explicit.
3. *State objects can be shared.*
    - If State objects have no instance variables—that is, the state they represent is encoded entirely in their type—then contexts can share a State object.

- When states are shared in this way, they are essentially flyweights with no intrinsic state, only behavior.

Implementation:

The State pattern raises a variety of implementation issues:

1. *Who defines the state transitions?*
   - The State pattern does not specify which participant defines the criteria for state transitions.
   - If the criteria are fixed, then they can be implemented entirely in the Context.
   - It is generally more flexible and appropriate, however, to let the State subclasses themselves specify their successor state and when to make the transition.
   - This requires adding an interface to the Context that lets State objects set the Context's current state explicitly.
   - Advantage: Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses.
   - Disadvantage: one State subclass will have knowledge of at least one other, which introduces implementation dependencies between subclasses.
2. *A table-based alternative.*
   - For each state, a table maps every possible input to a succeeding state.
   - In effect, this approach converts conditional code into a table look-up.
   - The main advantage of tables is their regularity: You can change the transition criteria by modifying data instead of changing program code.

   Disadvantages:
   - less efficient than a function call.
   - makes the transition criteria less explicit and therefore harder to understand.
   - It's usually difficult to add actions to accompany the state transitions.
3. *Creating and destroying State objects.*

   (1) Create State objects only when they are needed and destroy them there after
   - the request is a hard-coded operation invocation.
   - use a single handler function that takes a request code as parameter.

   (2) Creating State objects ahead of time and never destroying them.

- This approach is better when state changes occur rapidly, in which case you want to avoid destroying states, because they may be needed again shortly.
- Instantiation costs are paid once up-front, and there are no destruction costs at all.
- This approach might be inconvenient, though, because the Context must keep references to all states that might be entered.

4. *Using dynamic inheritance.*
    - Changing the behavior for a particular request could be accomplished by changing the object's class at run-time, but this is not possible in most object-oriented programming languages.
    - Objects in Self can delegate operations to other objects to achieve a form of dynamic inheritance.
    - Changing the delegation target at run-time effectively changes the inheritance structure.
    - This mechanism lets objects change their behavior and amounts to changing their class.

Sample Code:

First, we define the class TCPConnection, which provides an interface for transmitting data and handles requests to change state.

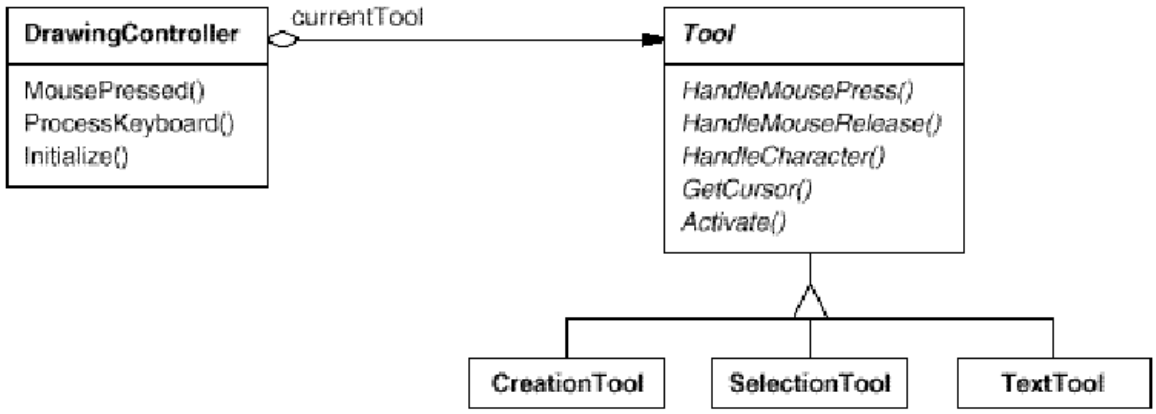
```
class TCPOctetStream;
class TCPState;

class TCPConnection {
    public:
        TCPConnection();
        void ActiveOpen();
        void PassiveOpen();
        void Close();
        void Send();
        void Acknowledge();
        void Synchronize();
        void ProcessOctet(TCPOctetStream*);
    private:
        friend class TCPState;
        void ChangeState(TCPState*);
    private:
        TCPState* _state;
};
```

```
class TCPState {
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

Known Uses:

- It is applicable to TCP connection protocols.
- Most popular interactive drawing programs provide "tools" for performing operations by direct manipulation.
- This technique is used in both the HotDraw and Unidraw drawing editor frameworks.
- It allows clients to define new kinds of tools easily.



- Coplien's Envelope-Letter idiom is related to State.
- Envelope-Letter is a technique for changing an object's class at run-time.
- The State pattern is more specific, focusing on how to deal with an object whose behavior depends on its state.

Related Patterns:

- The **Flyweight** pattern explains when and how State objects can be shared.
- State objects are often **Singletons.**

5.2 Strategy:

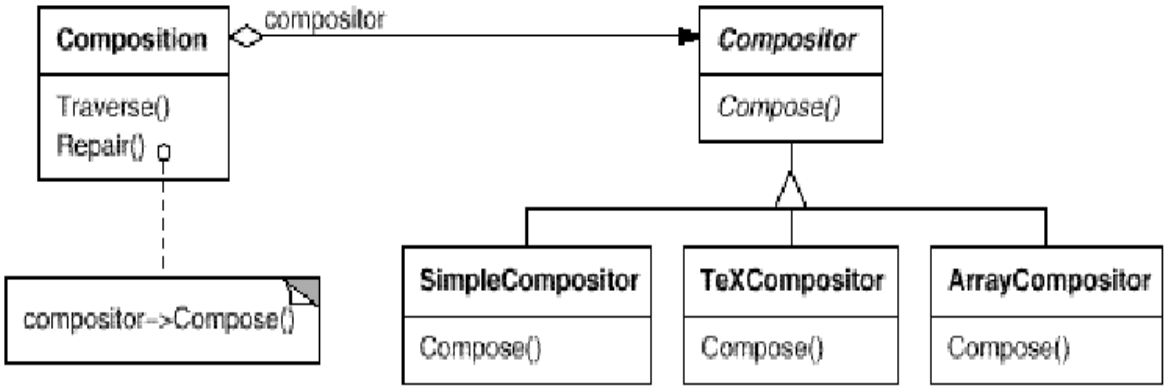
Intent:

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

Motivation:

- Many algorithms exist for breaking a stream of text into lines.
- Hard-wiring all such algorithms into the classes that require them isn't desirable for several reasons:
  - Clients that need line breaking get more complex if they include the line breaking code. That makes clients bigger and harder to maintain, especially if they support multiple line breaking algorithms.
  - Different algorithms will be appropriate at different times. We don't want to support multiple line breaking algorithms if we don't use them all.
  - It's difficult to add new algorithms and vary existing ones when line breaking is an integral part of a client.
- We can avoid these problems by defining classes that encapsulate different line breaking algorithms. An algorithm that's encapsulated in this way is called a strategy.



- Suppose a Composition class is responsible for maintaining and updating the line breaks of text displayed in a text viewer.
- Line breaking strategies aren't implemented by the class Composition.
- Instead, they are implemented separately by subclasses of the abstract Compositor class.

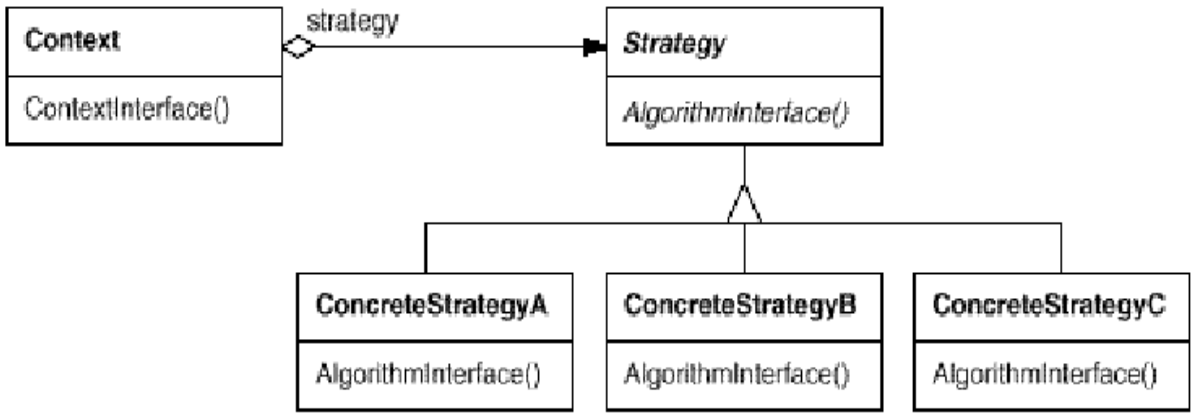
- Compositor subclasses implement different strategies:
  - **SimpleCompositor** implements a simple strategy that determines line breaks one at a time.
  - **TeXCompositor** implements the TeX algorithm for finding line breaks. This strategy tries to optimize line breaks globally, that is, one paragraph at a time.
  - **ArrayCompositor** 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.

Applicability:

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Structure:



Participants:

- **Strategy** (Compositor)
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - 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.

Collaborations:

- Strategy and Context interact to implement the chosen algorithm. A context may pass all data required by the algorithm to the strategy when the algorithm is called. Alternatively, the context can pass itself as an argument to Strategy operations. That lets the strategy call back on the context as required.
- A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context; thereafter, clients interact with the context exclusively. There is often a family of ConcreteStrategy classes for a client to choose from.

Consequences:

The Strategy pattern has the following benefits and drawbacks:

1. *Families of related algorithms.*
   - Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse.
   - Inheritance can help factor out common functionality of the algorithms.
2. *An alternative to subclassing.*
   - Inheritance offers another way to support a variety of algorithms or behaviors.
   - You can subclass a Context class directly to give it different behaviors.
   - But this hard-wires the behavior into Context.

- It mixes the algorithm implementation with Context's, making Context harder to understand, maintain, and extend. And you can't vary the algorithm dynamically.
- Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend.

3. *Strategies eliminate conditional statements.*
   - The Strategy pattern offers an alternative to conditional statements for selecting desired behavior.
   - When different behaviors are lumped into one class, it's hard to avoid using conditional statements to select the right behavior.
   - Encapsulating the behavior in separate Strategy classes eliminates these conditional statements.
4. *A choice of implementations.*
   - Strategies can provide different implementations of the same behavior.
   - The client can choose among strategies with different time and space trade-offs.
5. *Clients must be aware of different Strategies.*
   - The pattern has a potential drawback in that a client must understand how Strategies differ before it can select the appropriate one.
   - Clients might be exposed to implementation issues.
   - Therefore you should use the Strategy pattern only when the variation in behavior is relevant to clients.
6. *Communication overhead between Strategy and Context.*
   - The Strategy interface is shared by all ConcreteStrategy classes whether the algorithms they implement are trivial or complex.
   - Hence it's likely that some ConcreteStrategies won't use all the information passed to them through this interface; simple ConcreteStrategies may use none of it!
   - If this is an issue, then you'll need tighter coupling between Strategy and Context.

7. *Increased number of objects.*
   - Strategies increase the number of objects in an application.
   - Sometimes you can reduce this overhead by implementing strategies as stateless objects that contexts can share.

Implementation:

Consider the following implementation issues:

1. *Defining the Strategy and Context interfaces.*
   - The Strategy and Context interfaces must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
     1. Context pass data in parameters to Strategy operations. This keeps Strategy and Context decoupled. On the other hand, Context might pass data the Strategy doesn't need.
     2. A context pass itself as an argument, and the strategy requests data from the context explicitly. Alternatively, the strategy can store a reference to its context, eliminating the need to pass anything at all.
2. *Strategies as template parameters.*
   - In C++ templates can be used to configure a class with a strategy.
   - This technique is only applicable if (1) the Strategy can be selected at compile-time, and (2) it does not have to be changed at run-time.
   - In this case, the class to be configured (e.g., Context) is defined as a template class that has a Strategy class as a parameter.
3. *Making Strategy objects optional.*
   - The Context class may be simplified if it's meaningful not to have a Strategy object.
   - Context checks to see if it has a Strategy object before accessing it.
   - If there is one, then Context uses it normally.
   - If there isn't a strategy, then Context carries out default behavior.
   - The benefit of this approach is that clients don't have to deal with Strategy objects at all unless they don't like the default behavior.

Sample Code:

We'll give the high-level code for the Motivation example, which is based on the implementation of Composition and Compositor classes in InterViews.

The Composition class maintains a collection of Component instances, which represent text and graphical elements in a document. A composition arranges component objects into lines using an instance of a Compositor subclass, which encapsulates a line breaking strategy.

```
class Composition {
public:
    Composition(Compositor*);
    void Repair();
private:
    Compositor* _compositor;
    Component* _components;      // the list of components
    int _componentCount;         // the number of components
    int _lineWidth;              // the Composition's line width
    int* _lineBreaks;            // the position of linebreaks
    // in components
    int _lineCount;              // the number of lines
};
```

```
class Compositor {
public:
    virtual int Compose( Coord natural[], Coord stretch[], Coord shrink[],
    int componentCount, int lineWidth, int breaks[] ) = 0;
protected:
    Compositor();
};
```

The Compositor is an abstract class. Concrete subclasses define specific line breaking strategies. The composition calls its compositor in its Repair operation.

```
void Composition::Repair () {
    Coord* natural;
    Coord* stretchability;
    Coord* shrinkability;
    int componentCount;
    int* breaks;
// prepare the arrays with the desired component sizes
// ...
// determine where the breaks are:
    int breakCount;
    breakCount = _compositor->Compose( natural, stretchability, shrinkability,
    componentCount, _lineWidth, breaks );
// lay out components according to breaks
```

```
// ...
}

class TeXCompositor : public Compositor {
public:
    TeXCompositor();
    virtual int Compose( Coord natural[], Coord stretch[], Coord shrink[],
    int componentCount, int lineWidth, int breaks[] );
// ...
};
```

TeXCompositor uses a more global strategy. It examines a paragraph at a time, taking into account the components' size and stretchability.

Known Uses:

- Both ET++ and InterViews use strategies to encapsulate different line breaking algorithms.
- The Booch components use strategies as template arguments.
- RApp is a system for integrated circuit layout. Routing algorithms in Rapp are defined as subclasses of an abstract Router class which is a Strategy class.
- Borland's ObjectWindows uses strategies in dialogs boxes to ensure that the user enters valid data.
- ObjectWindows uses Validator objects to encapsulate validation strategies.
- Validators are examples of Strategy objects.

Related Patterns:

- Flyweight: Strategy objects often make good flyweights.

5.3 Template Method:

Intent:

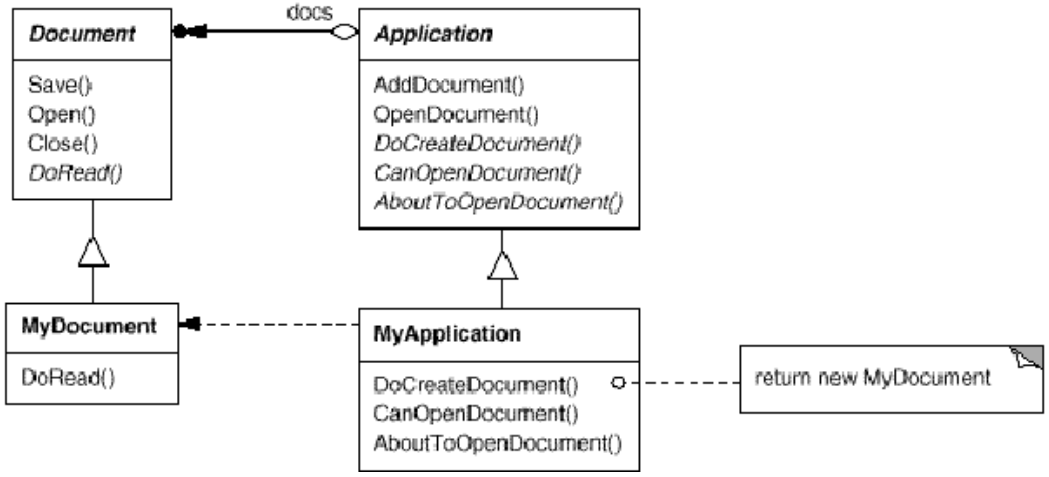
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Motivation:

- Consider an application framework that provides **Application** and **Document** classes.
- The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file.

- 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:




```
void Application::OpenDocument (const char* name) {
if (!CanOpenDocument(name)) {
        // cannot handle this document
        return; }
Document* doc = DoCreateDocument();
if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead(); }
 }
```

- OpenDocument defines each step for opening a document.
- It checks if the document can be opened, creates the application-specific Document object, adds it to its set of documents, and reads the Document from a file.
- We call **OpenDocument** a template method. It defines an algorithm in terms of abstract operations that subclasses override to provide concrete behavior.
- Application subclasses define the steps of the algorithm that check if the document can be opened (CanOpenDocument) and that create the Document (DoCreateDocument).
- Document classes define the step that reads the document (DoRead).

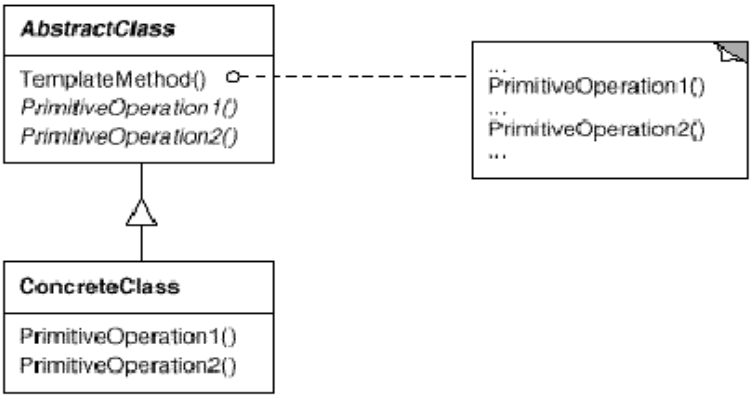
- The template method also defines an operation that lets Application subclasses know when the document is about to be opened (AboutToOpenDocument).
- By defining some of the steps of an algorithm using abstract operations, the template method fixes their ordering, but it lets Application and Document subclasses vary those steps to suit their needs.

Applicability:

The Template Method pattern should be used

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- when common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
- to control subclasses extensions. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.

Structure:



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.

Collaborations:

- ConcreteClass relies on AbstractClass to implement the invariant steps of the algorithm.

Consequences:

- Template methods are a fundamental technique for code reuse.
- Template methods lead to an inverted control structure that's sometimes referred to as "**the Hollywood principle**," that is, "Don't call us, we'll call you".
- This refers to how a parent class calls the operations of a subclass and not the other way around.
- Template methods call the following kinds of operations:
  - concrete operations (either on the ConcreteClass or on client classes);
  - concrete AbstractClass operations (i.e., operations that are generally useful to subclasses);
  - primitive operations (i.e., abstract operations);
  - factory methods; and
  - **hook operations**, which provide default behavior that subclasses can extend if necessary. A hookoperation often does nothing by default.
- It's important for template methods to specify which operations are hooks(may be overridden) and which are abstract operations (must be overridden).
- To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

Implementation:

Three implementation issues are important:

1. *Using C++ access control.*In C++, the primitive operations that a template method calls can be declared protected members. This ensures that they are only called by

the template method. Primitive operations that must be overridden are declared pure virtual. The template method itself should not be overridden;

2. *Minimizing primitive operations.*An important goal in designing template methods is to minimize the number of primitive operations that a subclass must override to flesh out the algorithm. The more operations that need overriding, the more tedious things get for clients.
3. *Naming conventions.*You can identify the operations that should be overridden by adding a prefix to their names. For example, the MacApp framework for Macintosh applications prefixes template method names with "Do-": "DoCreateDocument", "DoRead", and so on.

Sample Code:

The example comes from **NeXT'sAppKit**. Consider a class **View** that supports drawing on the screen.We can use a Display template method to set up this state. View defines two concrete operations, SetFocus and ResetFocus, which set up and clean up the drawing state, respectively.

```
void View::Display () {
SetFocus();
DoDisplay();
ResetFocus();
}
```

View's **DoDisplay** hook operation performs the actual drawing. Display calls **SetFocus** before DoDisplay to set up the drawing state; Display calls **ResetFocus** afterwards to release the drawing state.

DoDisplay does nothing in View:

```
void View::DoDisplay () { }
```

Subclasses override it to add their specific drawing behavior:

```
void MyView::DoDisplay () { // render the view's contents }
```

Known Uses:

- Template methods are so fundamental that they can be found in almost every abstract class.
- NeXT's AppKit uses this pattern.

Related Patterns:

- Factory Methods are often called by template methods.
- In the Motivation example, the factory method DoCreateDocument is called by the template method OpenDocument.
- Strategy: Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.

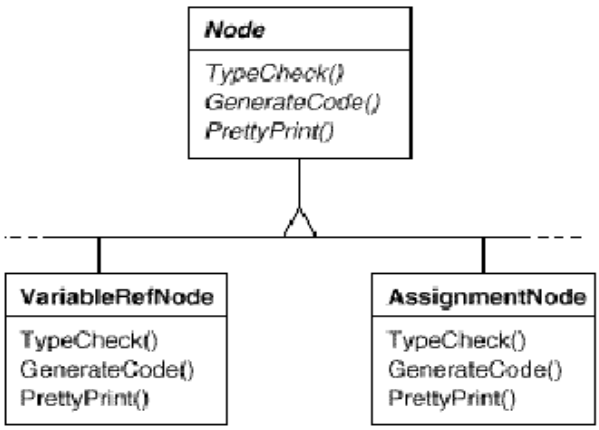
5.4 Visitor:

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.

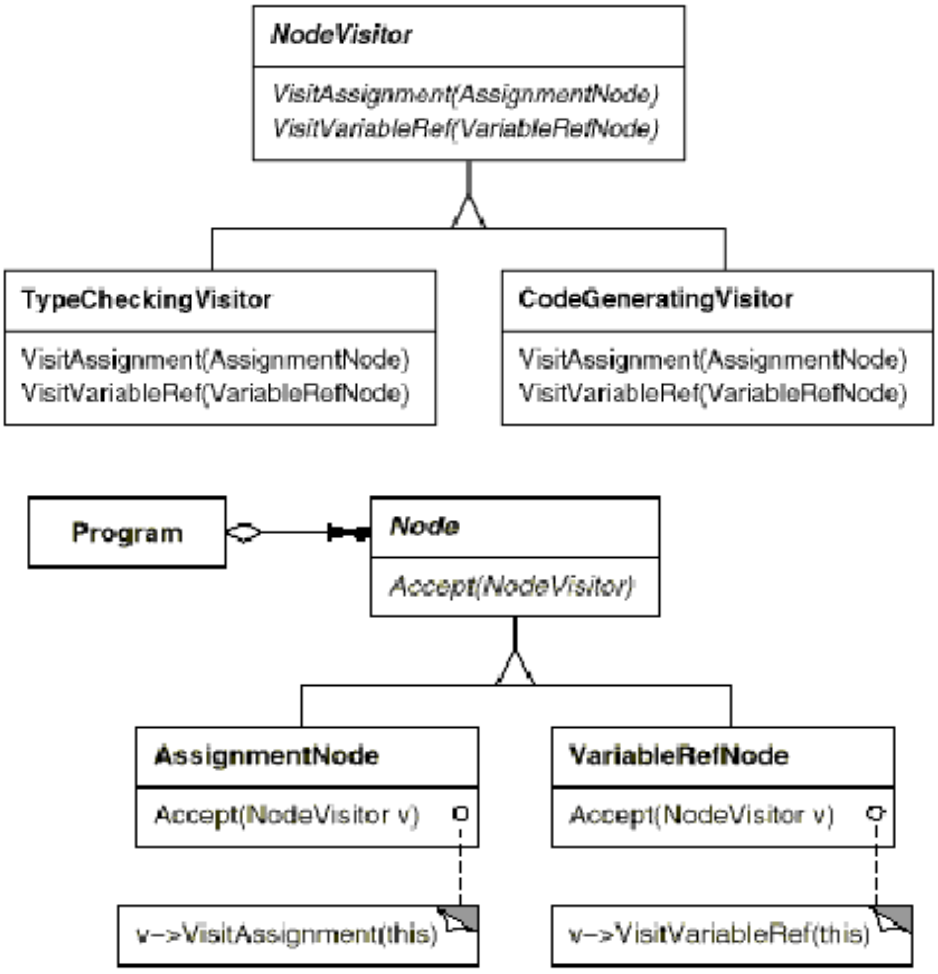
Motivation:

- Consider a compiler that represents programs as abstract syntax trees.
- It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined.
- It will also need to generate code. So it might define operations for type-checking, code optimization, flow analysis, checking for variables being assigned values before they're used, and so on.
- Most of these operations will need to treat nodes that represent assignment statements differently from nodes that represent variables or arithmetic expressions.



- 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.
- 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.
- To make visitors work for more than just type-checking, we need an abstract parent class NodeVisitor for all visitors of an abstract syntax tree.
- 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).

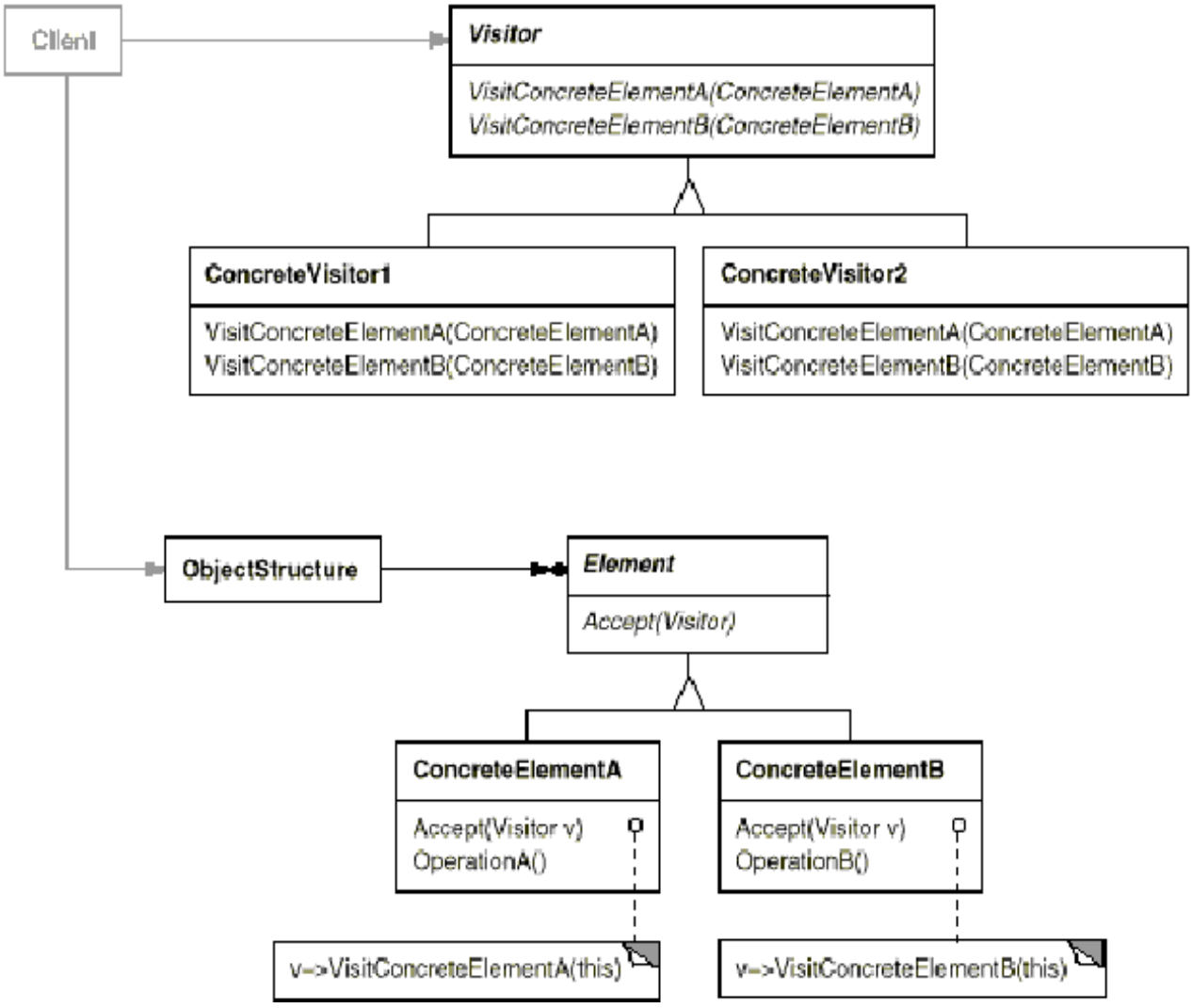


Applicability:

Use the Visitor pattern when

- an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
- the classes defining the object structure rarely change, but you often want to define new operations over the structure. Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly.

Structure:



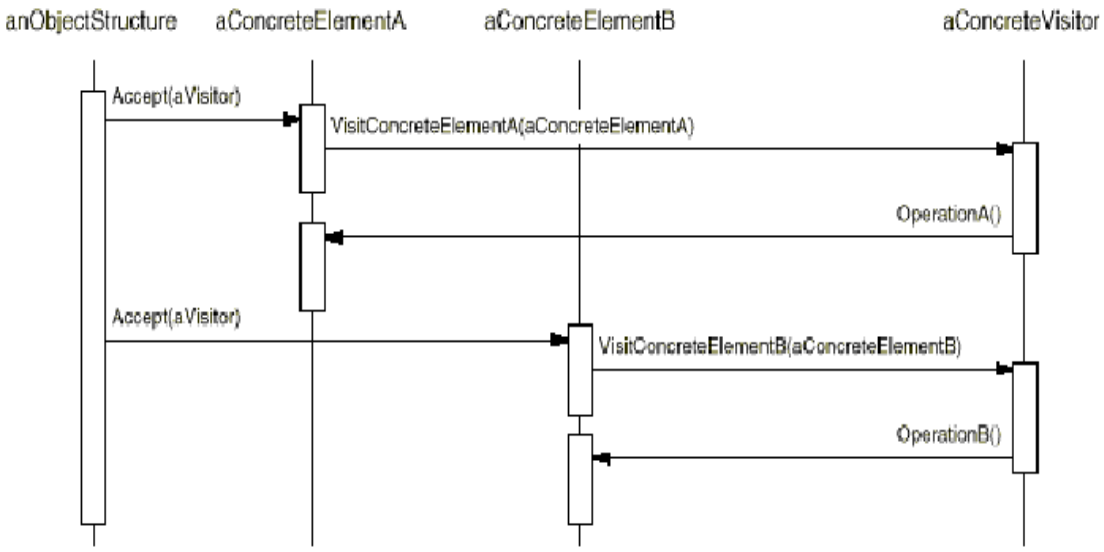
Participants:

- **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 or a collection such as a list or a set.

Collaborations:

- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

- The following interaction diagram illustrates the collaborations between an object structure, a visitor, and two elements:



Consequences:

Some of the benefits and liabilities of the Visitor pattern are as follows:

1. *Visitor makes adding new operations easy.*Visitors make it easy to add operations that depend on the components of complex objects. You can define a new operation over an object structure simply by adding a new visitor.
2. *A visitor gathers related operations and separates unrelated ones.*Related behavior isn't spread over the classes defining the object structure; it's localized in a visitor. Any algorithm-specific data structures can be hidden in the visitor.
3. *Adding new ConcreteElement classes is hard.*Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class.
4. *Visiting across class hierarchies.* An iterator can visit the objects in a structure as it traverses them by calling their operations. But an iterator can't work across object structures with different types of elements.
5. *Accumulating state.* Visitors can accumulate state as they visit each element in the object structure. Without a visitor, this state would be passed as extra arguments to the operations that perform the traversal, or they might appear as global variables.

6. *Breaking encapsulation.* Visitor's approach assumes that the ConcreteElement interface is powerful enough to let visitors do their job. As a result, the pattern often forces you to provide public operations that access an element's internal state, which may compromise its encapsulation.

Implementation:

The operation that ends up getting called depends on both the class of the element and the class of the visitor.

1. *Double dispatch.*
   - Effectively, the Visitor pattern lets you add operations to classes without changing them. Visitor achieves this by using a technique called **double-dispatch.**
   - The "Double-dispatch" simply means the operation that gets executed depends on the kind of request and the types of two receivers. It lets visitors request different operations on each class of element.
2. *Who is responsible for traversing the object structure?*
   - We can put responsibility for traversal in any of three places: in the object structure, in the visitor, or in a separate iterator object.
   - Often the object structure is responsible for iteration.

Sample Code:

```
class Equipment {
public:
    virtual ~Equipment();
    const char* Name() { return _name; }
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
    virtual void Accept(EquipmentVisitor&);
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

The Equipment operations return the attributes of a piece of equipment, such as its powerconsumption and cost.

```
class EquipmentVisitor {
public:
    virtual ~EquipmentVisitor();
    virtual void VisitFloppyDisk(FloppyDisk*);
    virtual void VisitCard(Card*);
    virtual void VisitChassis(Chassis*);
    virtual void VisitBus(Bus*);
// and so on for other concrete subclasses of Equipment
protected:
    EquipmentVisitor();
};
```

Known Uses:

- Smalltalk-80 compiler has a Visitor class called ProgramNodeEnumerator. It's used primarily for algorithms that analyse source code.
- IRIS Inventor (a toolkit for developing 3-D graphics applications).

Related Patterns:

- Composite: Visitors can be used to apply an operation over an object structure defined by the Composite pattern.
- Interpreter: Visitor may be applied to do the interpretation

5.5 Discussion of Behavioral Patterns

Encapsulating Variation:

- Encapsulating variation is a theme of many behavioral patterns.
- When an aspect of a program changes frequently, these patterns define an object that encapsulates that aspect.
- Then other parts of the program can collaborate with the object whenever they depend on that aspect.
- The patterns usually define an abstract class that describes the encapsulating object, and the pattern derives its name from that object.

For example,

- a Strategy object encapsulates an algorithm
- a State object encapsulates a state-dependent behavior
- a Mediator object encapsulates the protocol between objects

- These patterns describe aspects of a program that are likely to change.

- Most patterns have two kinds of objects: the new object(s) that encapsulate the aspect, and the existing object(s) that use the new ones.

Objects as Arguments:

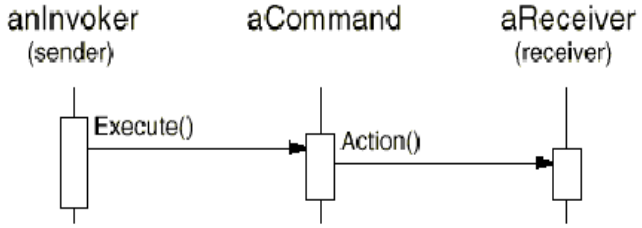
- Several patterns introduce an object that's always used as an argument. One of these is Visitor.
- A Visitor object is the argument to a polymorphic Accept operation on the objects it visits.
- Other patterns define objects that act as magic tokens to be passed around and invoked at a later time. Both Command and Memento fall into this category.
- Polymorphism is important in the Command pattern, because executing the Command object is a polymorphic operation.
- The Memento interface is so narrow that a memento can only be passed as a value. So there are no polymorphic operations at all to its clients.

Should Communication be Encapsulated or Distributed?

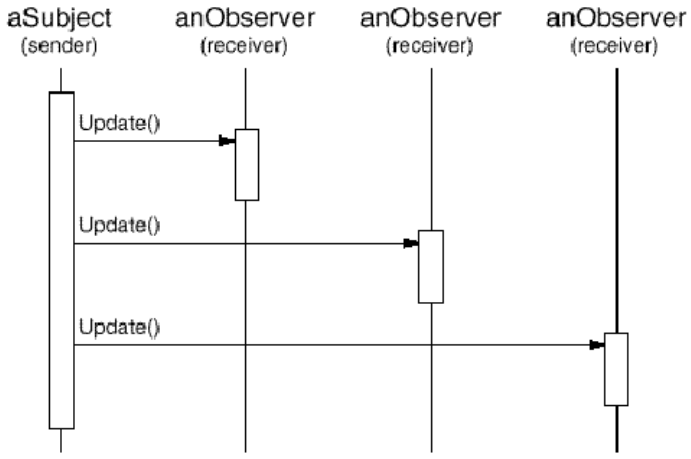
- **Mediator** and **Observer** are competing patterns.
- The difference between them is that Observer distributes communication by introducing Observer and Subject objects, whereas a Mediator object encapsulates the communication between other objects.
- It is easier to make reusable Observers and Subjects than to make reusable Mediators.
- The Observer pattern promotes partitioning and loose coupling between Observer and Subject, and that leads to finer-grained classes that are more apt to be reused.
- It's easier to understand the flow of communication in Mediator than in Observer.
- Observers and subjects are usually connected shortly after they're created, and it's hard to see how they are connected later in the program.
- Observers in Smalltalk can be parameterized with messages to access the Subject state, and so they are even more reusable than they are in C++.
- Thus a Smalltalk programmer will often use Observer where a C++ programmer would use Mediator.

Decoupling Senders and Receivers:

- When collaborating objects refer to each other directly, they become dependent on each other, and that can have more impact on the layering and reusability of a system.
- Command, Observer, Mediator, and Chain of Responsibility address how you can decouple senders and receivers, but with different trade-offs.
- The **Command** pattern supports decoupling by using a Command object to define the binding between a sender and receiver:

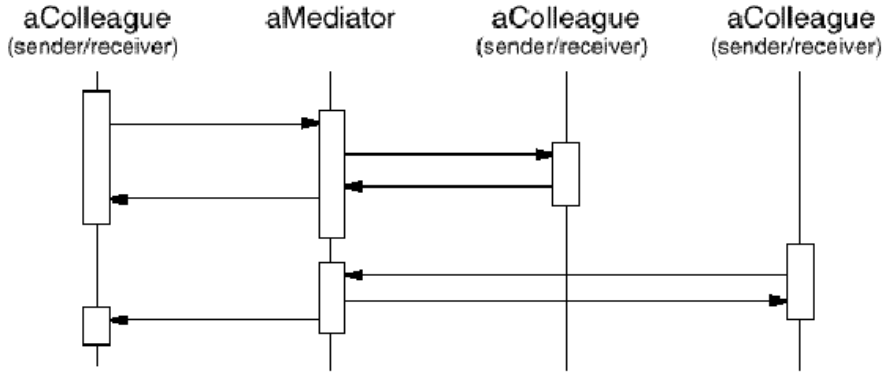


- The Command object provides a simple interface for issuing the request (i.e., Execute( )).
- Defining the sender-receiver connection in a separate object lets the sender work with different receivers.
- You can reuse the Command object to parameterize a receiver with different senders.
- The **Observer** pattern decouples senders (subjects) from receivers (observers) by defining an interface for signalling changes in subjects.
- Observer defines a looser sender-receiver binding than Command, since a subject may have multiple observers, and their number can vary at run-time.

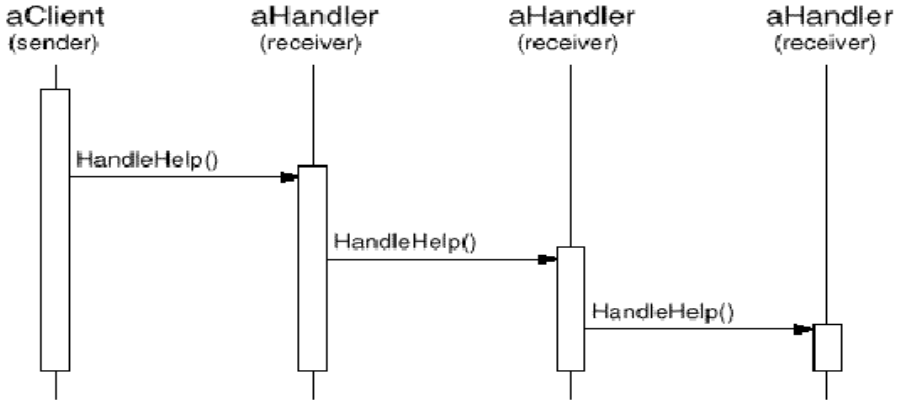


- The Subject and Observer interfaces in the Observer pattern are designed for communicating changes.

- Therefore the Observer pattern is best for decoupling objects when there are data dependencies between them.
- The **Mediator** pattern decouples objects by having them refer to each other indirectly through a Mediator object.
- A Mediator object routes requests between Colleague objects and centralizes their communication.



- The Mediator pattern can reduce subclassing in a system, because it centralizes communication behavior in one class instead of distributing it among subclasses.
- However, ad hoc dispatching schemes often decrease type safety.
- The Chain of Responsibility pattern decouples the sender from the receiver bypassing the request along a chain of potential receivers:



- Since the interface between senders and receivers is fixed, Chain of Responsibility may also require a custom dispatching scheme.
- Hence it has the same type-safety drawbacks as Mediator.

- Chain of Responsibility is a good way to decouple the sender and the receiver if the chain is already part of the system's structure, and one of several objects may be in a position to handle the request.
- The pattern offers added flexibility in that the chain can be changed or extended easily.