COURSE MATERIAL
ON
DESIGN PATTERNS(IV-I)

CONTENTS

UNIT-I

Introduction:

1. What is a design pattern?
2. Design patterns in Smalltalk MVC
3. Describing Design Patterns
4. The Catalog of Design Patterns
5. Organizing the Catalog
6. How Design Patterns Solve Design Problems
7. How to Select a Design Pattern
8. How to use a Design Pattern.

UNIT-II

A Case Study:

1. Designing a Document Editor Design Problems,
2. Document Structure
3. Formatting
4. Embellishing the User Interface
5. Supporting Multiple Look-and Feel Standards
6. Supporting Multiple Window Systems,

7. User Operations Spelling Checking and Hyphenation,
8. Summary.
9. Creational Patterns: Abstract Factory
10. Builder
11. Factory Method
12. Prototype
13. Singleton
14. Discussion of Creational Patterns

UNIT-III

Structural Pattern Part-I :

1. Adapter
2. Bridge
3. Composite

Structural Pattern Part-II :

1. Decorator
2. Façade
3. Flyweight
4. Proxy

UNIT-IV

Behavioural Patterns Part-I :

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator

Behavioural Patterns Part-II :

1. Mediator
2. Memento
3. Observer

UNIT-V

Behavioural Patterns Part-II (cont'd):

1. State
2. Strategy
3. Template Method
4. Visitor
5. Discussion of Behavioral Patterns
6. What to Expect from Design Patterns
7. A Brief History
8. The Pattern Community An Invitation
9. A Parting Thought

Unit 1

What is a Design Pattern?

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" .Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns.

In general, a pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction.

2. The **problem** 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 symptomatic 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. The **solution** 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 (classes and objects in our case) solves it.

4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced 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.

Design Patterns in Smalltalk MVC

The Model/View/Controller (MVC) triad of classes is used to build user interfaces in Smalltalk-80. Looking at the design patterns inside MVC should help you see what we mean by the term "pattern." 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.

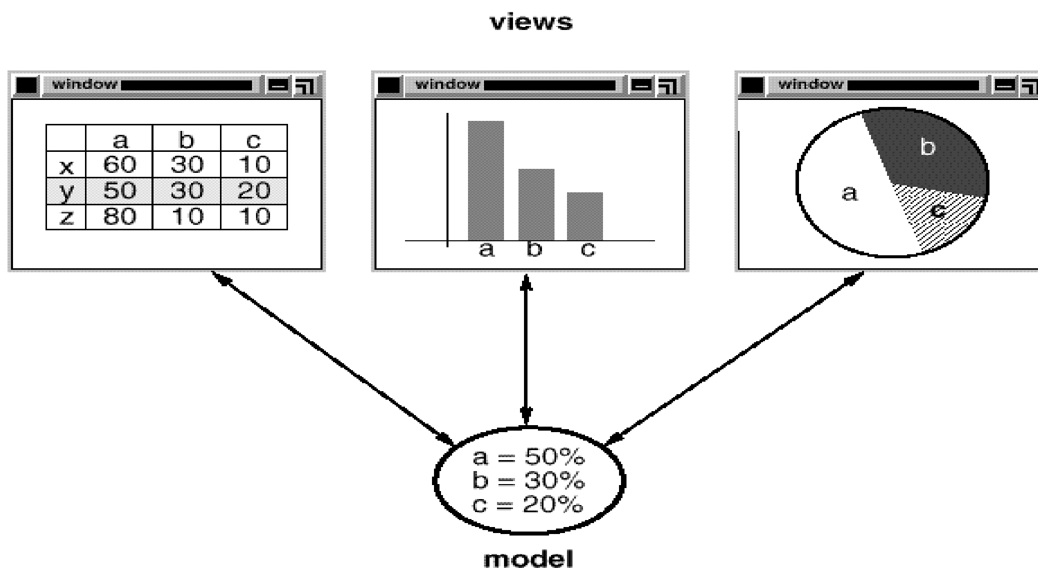
MVC decouples them to increase flexibility and reuse.

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. You can also create new views for a model without rewriting it.

The following diagram shows a model and three views. (We've left out the controllers

for simplicity.) 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 communicatewith 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.

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.

The user interface for an object inspector can consist of nested views that may
be reused in a debugger.

MVC supports nested views with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

MVC also lets you change the way a view responds to user input without changing its visual presentation. for example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a Controller object. There is a class hierarchy of controllers, making it easy to create a new controller as a variation on an existing one.

The View-Controller relationship is an example of the Strategy design pattern. A Strategy is an object that represents an algorithm. It's useful when you want to replace the algorithm either statically or dynamically, when you have a lot of variants of the algorithm, or when the algorithm has complex data structures that you want to encapsulate.

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
…………………………………………………………………………………………………..

Describing Design Patterns

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.
These are …

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. The pattern's classification reflects the scheme we introduce in Section
Intent
A short statement that answers the following questions: 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 objectstructures 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. Appendix B describes these notations in detail.

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?

…………………………………………………………………………………………………

The Catalog of Design Patterns

The catalog contain 23 patterns……..

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes. Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge

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

Builder

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

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.

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

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

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

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

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.

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

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

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

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.

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

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

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

Proxy

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

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

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

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

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.
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.

……………………………………………………………………………………………….

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. This section classifies design patterns so that we can refer to families of 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 (Table 1.1).

The first criterion, called **purpose**, reflects what a pattern does. Patterns can have either **creational**, **structural**, or **behavioral** purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

The second criterion, called **scope**, specifies whether the pattern applies primarily to classes or to objects.
*Class patterns deal with relationships between classes and their subclasses. These relationships are established through inheritance, so they are static—fixed at compile-time.
*Object patterns deal with object relationships, which can be changed at run-time and are more dynamic. Almost all patterns use inheritance to some extent. So the only patterns labeled "class patterns" are those that focus on class relationships.
*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.

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

Table 1.1: Design pattern space

How Design Patterns Solve Design Problems
Design patterns solve many of the day-to-day problems object-oriented designers face, and in many different ways. Here are several of these problems and how design patterns solve them.

Finding Appropriate Objects

Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called **methods** or operations. An object performs an operation when it receives a request (or **message**) from a **client**. Requests are the *only* way to get an object to execute an operation. Operations are the *only* way to change an object's internal data. Because of these restrictions, the object's internal state is said to be encapsulated; it cannot be accessed directly, and its representation is invisible from outside the object. The hard part about object-oriented design is decomposing a system into objects.
Design patterns help you identify less-obvious abstractions and the objects that can capture them. For example, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs. The Strategy (349) pattern describes how to implement interchangeable families of algorithms. The State (338) pattern represents each state of an entity as an object. These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.

Determining Object Granularity

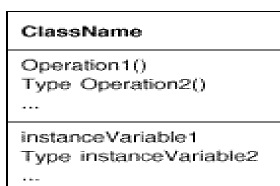
Objects can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. How do we decide
what should be an object?
Design patterns address this issue as well. The Facade pattern describes how to represent complete subsystems as objects, and the Flyweight pattern describes how to support huge numbers of objects at the finest granularities.
Other design patterns describe specific ways of decomposing an object into smaller objects. Abstract Factory and Builder yield objects whose only responsibilities are creating other objects. Visitor and Command yield objects whose only responsibilities are to implement a request on another object or group of objects.

Specifying Object Interfaces

Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's signature. The set of all signatures defined by an object's operations is called the interface to the object. An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object. A type is a name used to denote a particular interface. We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window."
An object's interface says nothing about its implementation—different objects are free to implement requests differently. That means two objects having completely different implementations can have identical interfaces. When a request is sent to an object, the particular operation that's performed depends on *both* the request *and* the receiving object. Different objects that support identical requests may have different implementations of the operations

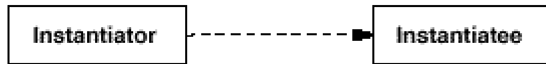
Specifying Object Implementations

An object's implementation is defined by its class. The class specifies the object's internal
data and representation and defines the operations the object can perform. Our OMT-based notation (summarized in Appendix B) depicts a class as a rectangle with the class name in bold. Operations appear in normal type below the class name. Any data that the class defines comes after the operations. Lines separate the class name from the operations and the operations from the data:



Return types and instance variable types are optional, since we don't assume a statically typed implementation language.
Objects are created by **instantiating** a class. The object is said to be an **instance** of the class. The process of instantiating a class allocates storage for the object's internal data (made up of instance variables) and associates the operations with these data. Many similar instances of an object can be created by instantiating a class.
A dashed arrowhead line indicates a class that instantiates objects of another

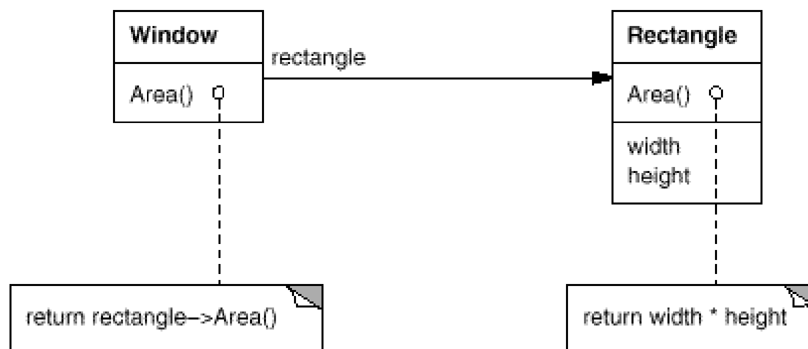
class. The arrow points to the class of the instantiated objects.



Putting Reuse Mechanisms to Work

Most people can understand concepts like objects, interfaces, classes, and inheritance. The challenge lies in applying them to build flexible, reusable software, and design patterns can show you how.

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition. As we've explained, class inheritance lets you define the implementation of one class in terms of another's. Reuse by subclassing is often referred to as white-box reuse. The term "white-box" refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.
Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or *composing* objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as "black boxes."
The following diagram depicts the Window class delegating its Area operation to a Rectangle instance



How to Select a Design Pattern

Here are several different approaches to finding the design pattern that's right for your problem:

1. Consider how design patterns solve design problems.
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 sound relevant to your problem. You

3. Study how patterns interrelate.

. Studying these relationships can help direct you to the right pattern or group of patterns.

4. Study patterns of like purpose.

The catalog has three chapters, one for creational patterns, another for structural patterns, and a third for behavioral patterns. Each chapter starts off with introductory comments on the patterns and concludes with a section that compares and contrasts them. These sections give you insight into the similarities and differences between patterns of like purpose.

5. Examine a cause of redesign.

Look at the causes of redesign starting on 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.

How to Use 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:

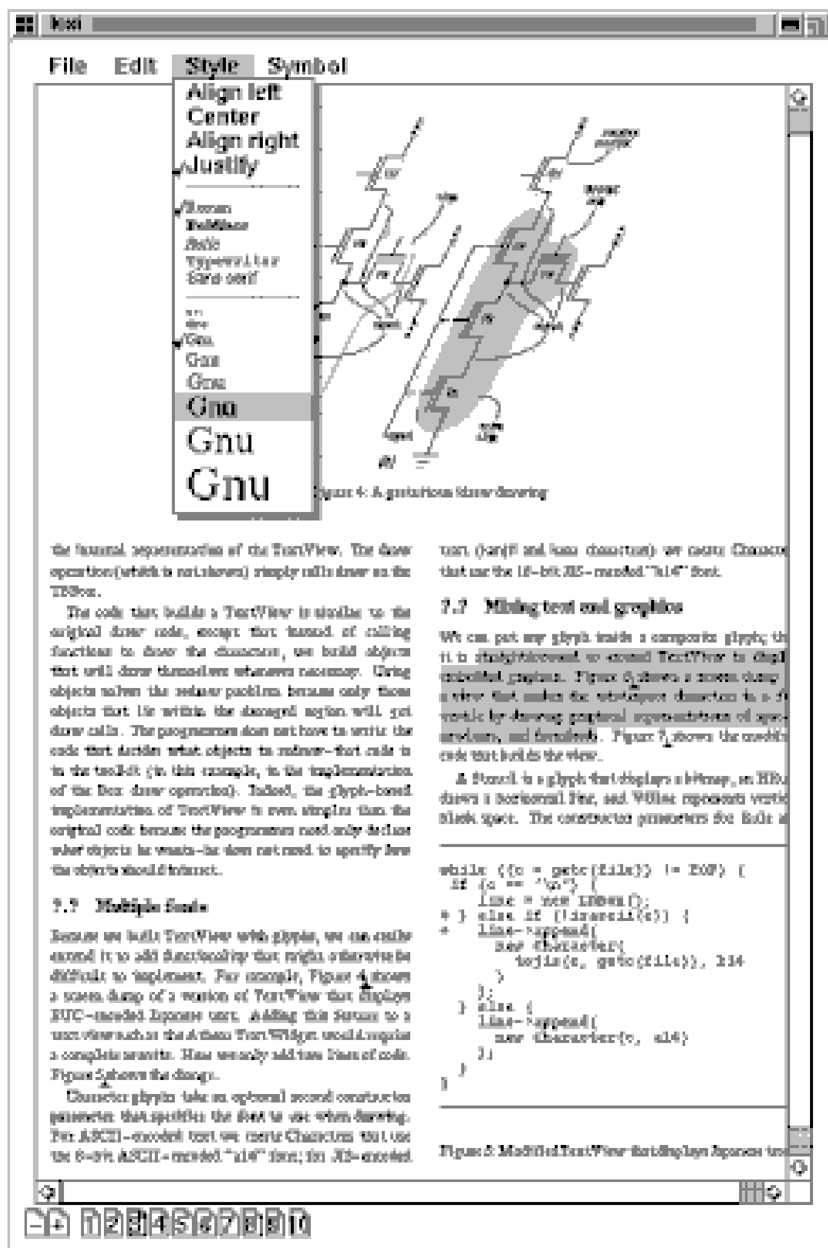
1. Read the pattern once through for an overview. Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.
2. Go back and study the Structure, Participants, and Collaborations sections. Make sure you understand the classes and objects in 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. For example, if you use the Strategy pattern for a text compositing algorithm, then you might have classes SimpleLayoutStrategy or TeXLayoutStrategy.
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-2

A Case Study: Design a Document Editor

his chapter presents a case study in the design of a "What-You-See-Is-What-You-Get" (or "WYSIWYG") document editor called **Lexi**.1
Figure 2.1 depicts Lexi's user interface. AWYSIWYG representation of the document occupies the large rectangulararea in the center. The document can mix text and graphics freely ina variety of formatting styles. Surrounding the document are the usual pull-down menus and scroll bars, plus a collection of page icons for jumping to a particular page in the document.



……………………………………………………………..

Design Problems

We will examine seven problems in Lexi's design:

1. *Document structure.*The choice of internal representation for the document affects nearlyevery aspect of Lexi's design. All editing, formatting, displaying,and textual analysis will require traversing the representation.
Theway we organize this information will impact the design of the rest ofthe
application.

2. *Formatting.* How does Lexi actually arrange text and graphics into lines andcolumns? What objects are responsible for carrying out differentformatting policies? How do these policies interact with thedocument's internal representation?

3. *Embellishing the user interface.*Lexi's user interface includes scroll bars, borders, and drop shadowsthat embellish the WYSIWYG document interface. Such embellishments arelikely to change as Lexi's user interface evolves. Hence it'simportant to be able to add and remove embellishments easily withoutaffecting the rest of the application.

4. *Supporting multiple look-and-feel standards.*Lexi should adapt easily to different look-and-feel standardssuch as Motif and Presentation Manager (PM) without major modification.

5. *Supporting multiple window systems.*Different look-and-feel standards are usually implemented on differentwindow systems. Lexi's design should be as independent of the windowsystem as possible.

6. *User operations.*Users control Lexi through various user interfaces, includingbuttons and pull-down menus. The functionality behind theseinterfaces is scattered throughout the objects in the application.The challenge here is to provide a uniform mechanism both foraccessing this scattered functionality and for undoing its effects.

7. *Spelling checking and hyphenation.*How does Lexi support analytical operations such as checking formisspelled words and determining hyphenation points? How can weminimize the number of classes we have to modify to add a newanalytical operation?

……………………………………………………………………………..

Document Structure

A document is ultimately just an arrangement of basic graphicalelements such as characters, lines, polygons, and other shapes. Theseelements capture the total information content of the document. Yet anauthor often views these elements not in graphical terms but in terms of the document's physical structure—lines, columns, figures,tables, and other substructures.

Lexi's user interface should let users manipulate these substructures directly.
For example, a user should be able to treat adiagram as a unit rather than as a collection of individual graphical primitives. The user should be able to refer to a table as a whole,not as an unstructured mass of text and graphics

In particular, the internal representation should support thefollowing:

· Maintaining the document's physical structure, that is, thearrangement of text and graphics into lines, columns, tables, etc.

· Generating and presenting the document visually.
· Mapping positions on the display to elements in the internalrepresentation.
This lets Lexi determine what the user isreferring to when he points to something in the visual representation.

Recursive Composition

A common way to represent hierarchically structured information isthrough a technique called **recursive composition**, whichentails building increasingly complex elements out of simpler ones.

Recursive composition gives us a way to compose a document out ofsimple graphical elements. As a first step, we can tile a set ofcharacters and graphics from left to right to form a line in thedocument.

Then multiple lines can be arranged to form a column,multiple columns can form a page, and so on (seeFigure 2.2).

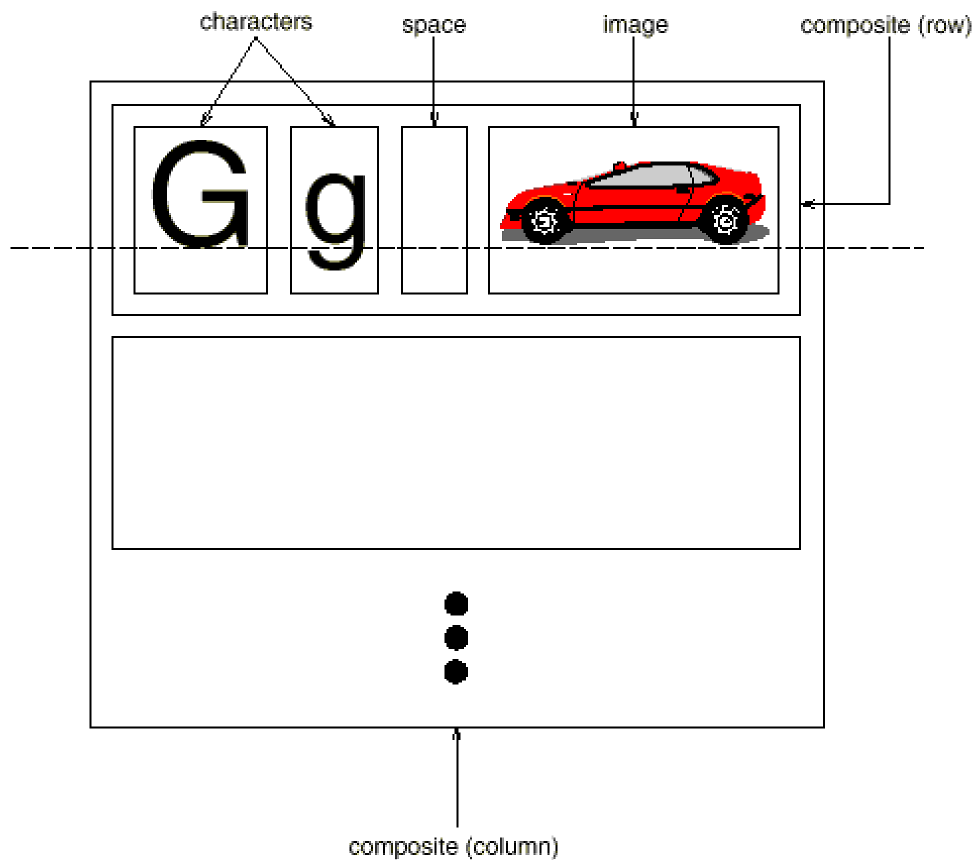


Figure : Recursive composition of text and graphics

We can represent this physical structure by devoting an object to eachimportant element. That includes not just the visible elements likethe characters and graphics but the invisible, structural elements as well—the lines and the column. The result is the object structures hown in below Figure .

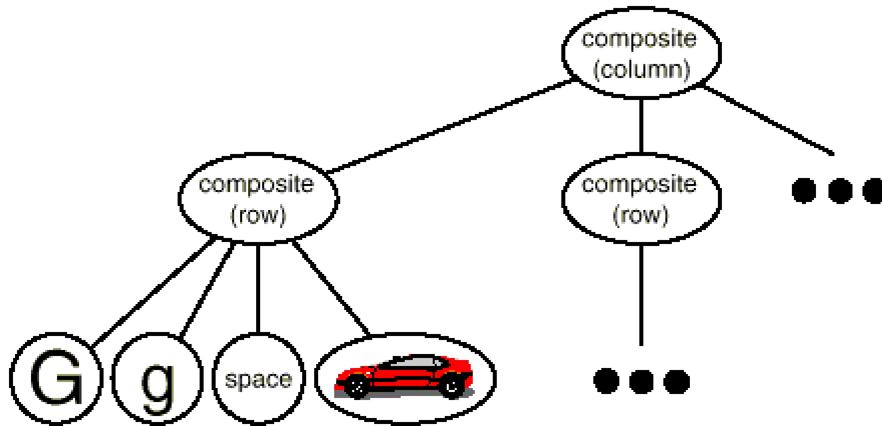


Figure : Object structure for recursive composition oftext and graphics

Glyphs

We'll define a **Glyph** abstract class for all objects that can appear in a document structure
Its subclasses define bothprimitive graphical elements (like characters and images) andstructural elements (like rows and columns).

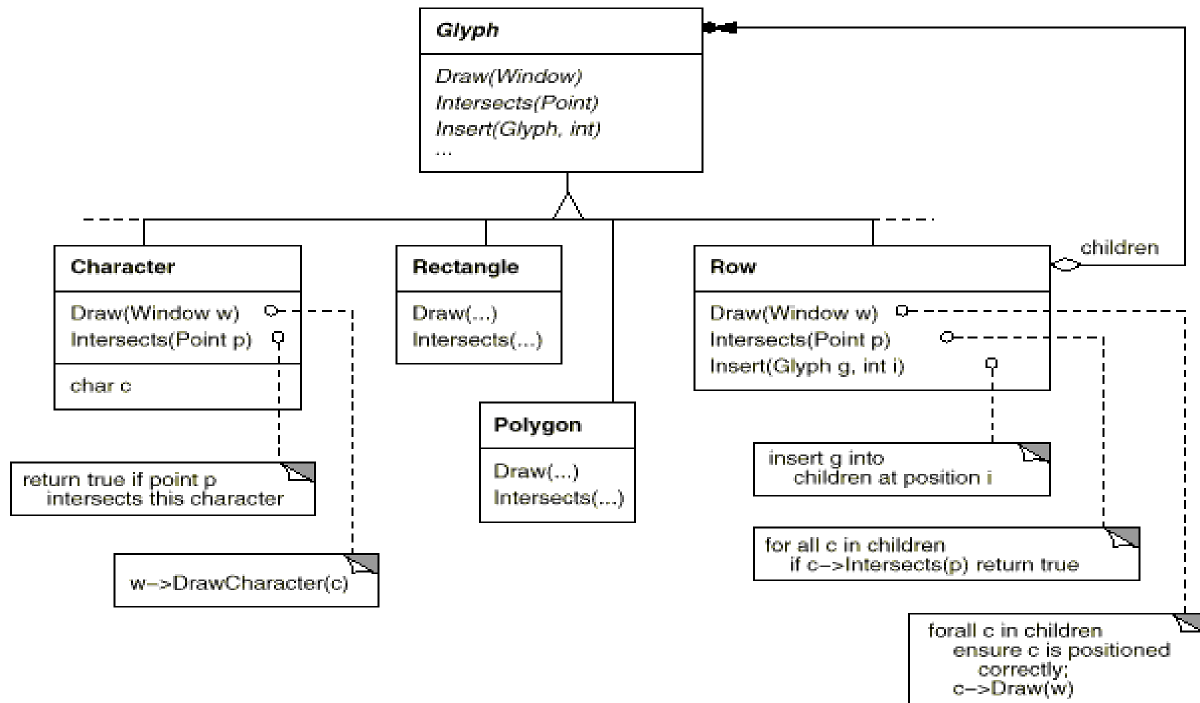


Figure : Partial Glyph class hierarchy

…………………………………………………………………………………………… …………………… Formatting

.

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 upwith a variety of formatting algorithms with different strengths and weaknesses. Because Lexi is a WYSIWYG editor, an important trade-off toconsider is the balance between formatting quality and formatting speed.
We want generally good response from the editor without sacrificing how good the document looks.
we could add a new kind of Glyphsubclass without regard to the formatting algorithm. Conversely,adding a new formatting algorithm shouldn't require modifying existing glyphs. These characteristics suggest we should design Lexi so that it'seasy to change the formatting algorithm at least at compile-time, if not at run-time as well.
We can isolate the algorithm and make iteasily replaceable at the same time by

Compositor and Composition
We'll define a **Compositor** class for objects that can encapsulate a formatting algorithm. The interface 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 (specializedfor a particular linebreaking algorithm) when it is created,
And it tells the compositor to Compose its glyphs whennecessary, for example, when the user changes a document
.

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

… Basic compositor interface

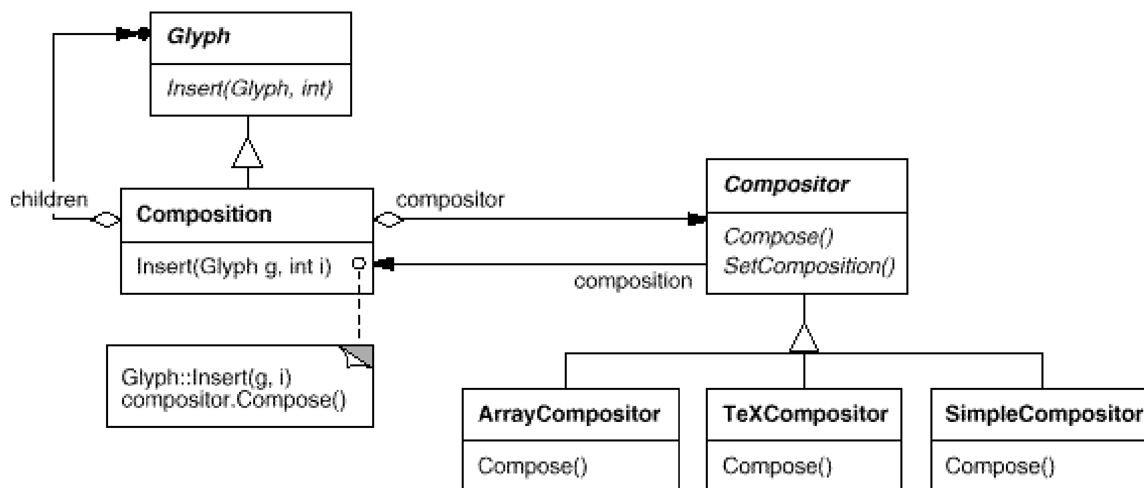


Figure : Composition and Compositor class relationships

………………………………………………………………………………

Embellishing the User Interface

We consider two embellishments in Lexi's user interface…

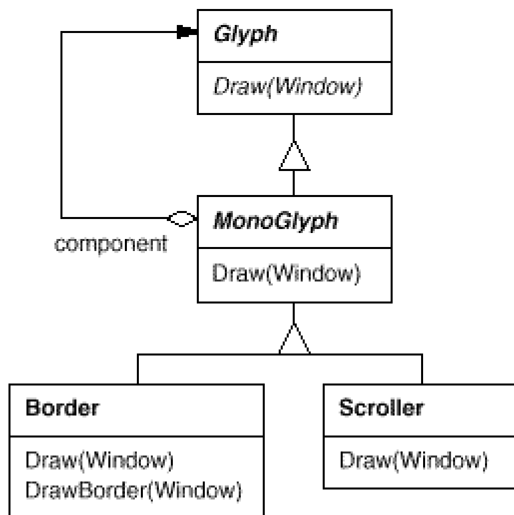
Thefirst adds a border around the text editing area to demarcate the pageof text.

The second adds scroll bars that let the user view differentparts of the page. To make it easy to add and remove theseembellishments (especially at run-time),
we shouldn't use inheritance to add them to the user interface. We achieve the most flexibility if other user interface objects don't even know the embellishments arethere. That will let us add and remove the embellishments without changing other classes

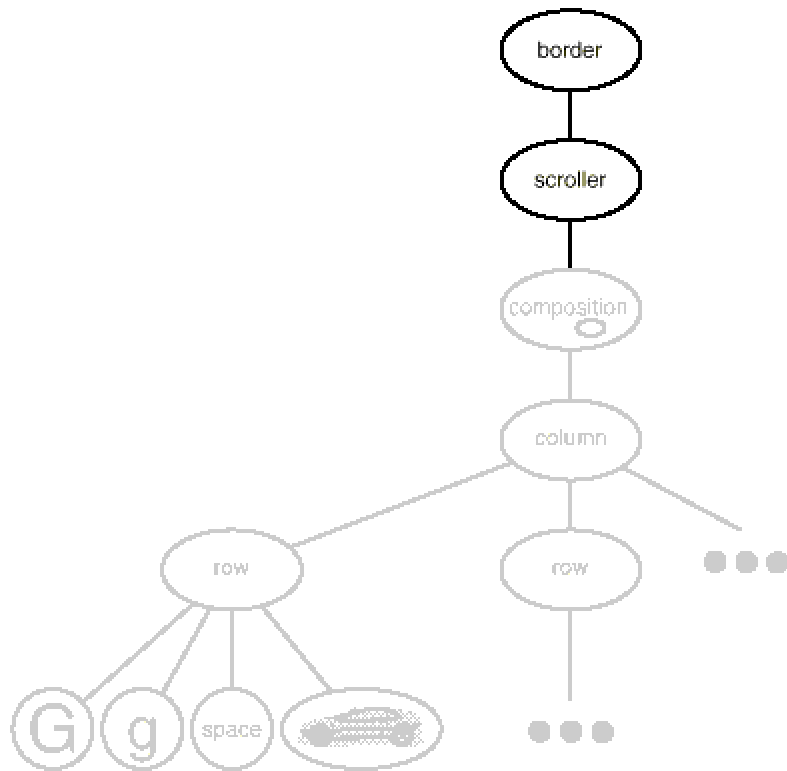
Monoglyph

We can apply the concept of transparent enclosure to all glyphs that embellish other glyphs. To make this concept concrete, we'll define asubclass of Glyph called **MonoGlyph** to serve as an abstractclass for "embellishment glyphs," likeBorder (see Figure 2.7).MonoGlyph stores a reference to a component and forwards all requests to it. That makes MonoGlyph totally transparent to clients by default.For
example, MonoGlyph implements the Draw operation like this:

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



We compose the existingComposition instance in a Scroller instance to add the scrolling
nterface, and we compose that in a Border instance.
The resultingobject structure appears in below Figure



Embellished object structure

Note that we can reverse the order of composition, putting the bordered composition into the Scroller instance. In that case theborder would be scrolled along with the text, which may or may not be desirable.

…………………………………………………………………………………….

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. As that happens, emphasis shifts away from hard-coding a fixed set of behaviors toward defining a smaller set of fundamental behaviors that can be composed into any number of more complex ones. Thus creating objects with particular behaviors requires more than simply instantiating a class.

There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, they hide how instances of these classes are created and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes. Consequently, the creational patterns give you a lot of flexibility in *what* gets created, *who* creates it, *how* it gets created, and *when*.

Abstract Factory

Intent

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

Also Known As

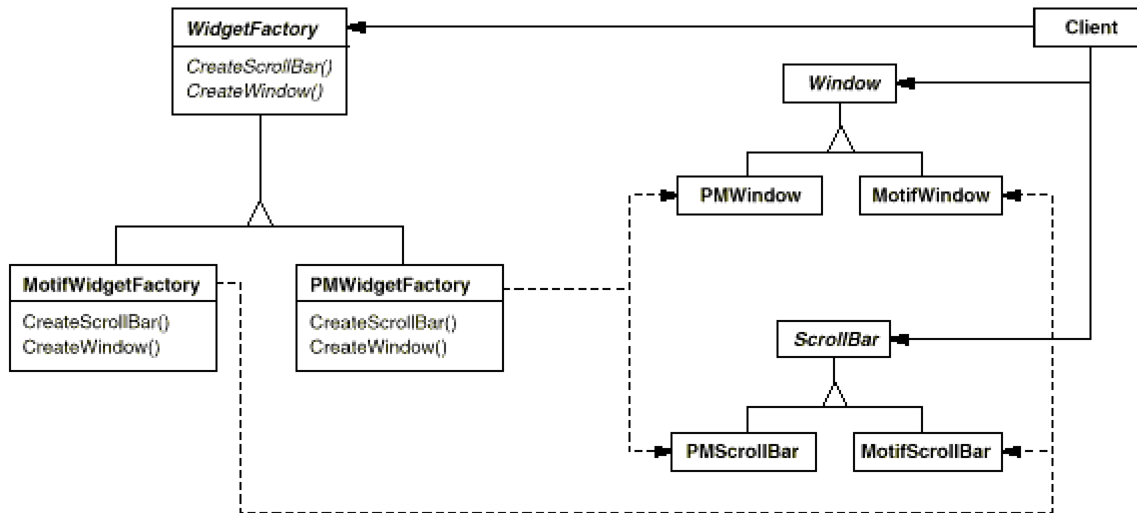
Kit

Motivation

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

We can solve this problem by defining an abstract WidgetFactory class that declares an interface for creating each basic kind of widget. There's also an abstract class for each kind of widget, and concrete subclasses implement widgets for specific look-and-feel standards.

WidgetFactory's interface has an operation that returns a new widget object for each abstract widget class. Clients call these operations to obtain widget instances, but clients aren't aware of the concrete classes they're using. Thus clients stay independent of the prevailing



Applicability

Use the Abstract Factory pattern when

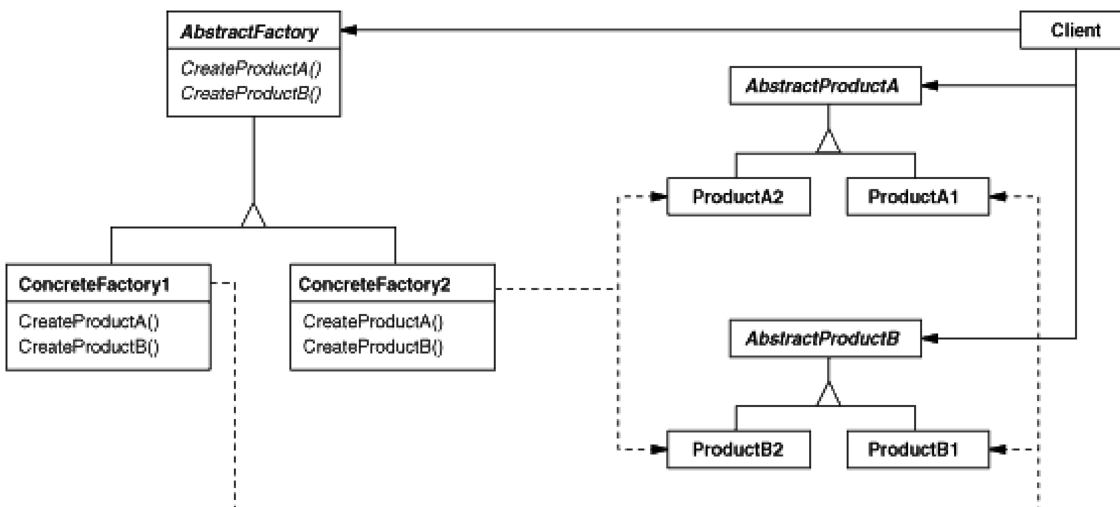
· a system should be independent of how its products are created, composed, and represented.

· 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 you need to enforce this constraint.

· you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure



Participants

· **AbstractFactory** (WidgetFactory)
declares an interface for operations that create abstract product objects.

· **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
implements the operations to create concrete product objects.

· **AbstractProduct** (Window, ScrollBar)
declares an interface for a type of product object.

· **ConcreteProduct** (MotifWindow, MotifScrollBar)
defines a product object to be created by the corresponding concrete factory.
implements the AbstractProduct interface.

· **Client**
uses only interfaces declared by AbstractFactory andAbstractProduct 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.

· AbstractFactory 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 pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate

.

2. *It makes exchanging product families easy.* The class of a concrete factory appears only once in an application—that is, where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use different product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once.

3. *It promotes consistency among products.* When product objects in a family are designed to work together, it's important that an application use objects from only one family at a time. AbstractFactory makes this easy to enforce.

4. *Supporting new kinds of products is difficult.* Extending abstract factories to produce new kinds of Products isn't easy. That's because the

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 ConcreteFactory per product family. So it's usually best implemented as a Singleton .

2. *Creating the products.* AbstractFactory only declares an *interface* for creating products. It's up to ConcreteProduct subclasses to actually create them. The most common way to do this is to define a factory method for each product.

Sample Code

.

Class MazeFactory 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.

```
class MazeFactory {
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); }
};
```

Recall that the member function CreateMaze builds a small maze consisting of two rooms with a door between them. CreateMaze hard-codes the class names, making it difficult to create mazes with different components. Here's a version of Create Maze that remedies that shortcoming by taking a MazeFactory as a parameter:

Known Uses

InterViews uses the "Kit" suffix [Lin92] to denote AbstractFactory classes. It defines WidgetKit and Dialog Kit abstract factories for generating look-and-feel-specific user interface objects. InterViews also includes a Layout Kit that generates different composition objects depending on the layout
desired. For example, a layout that is conceptually horizontal may require different composition objects depending on the document's orientation (portrait or landscape).

Related Patterns

AbstractFactory classes are often implemented with factory methods Factory Method but they can also be implemented using Prototype

A concrete factory is often a singleton

Builder

Intent

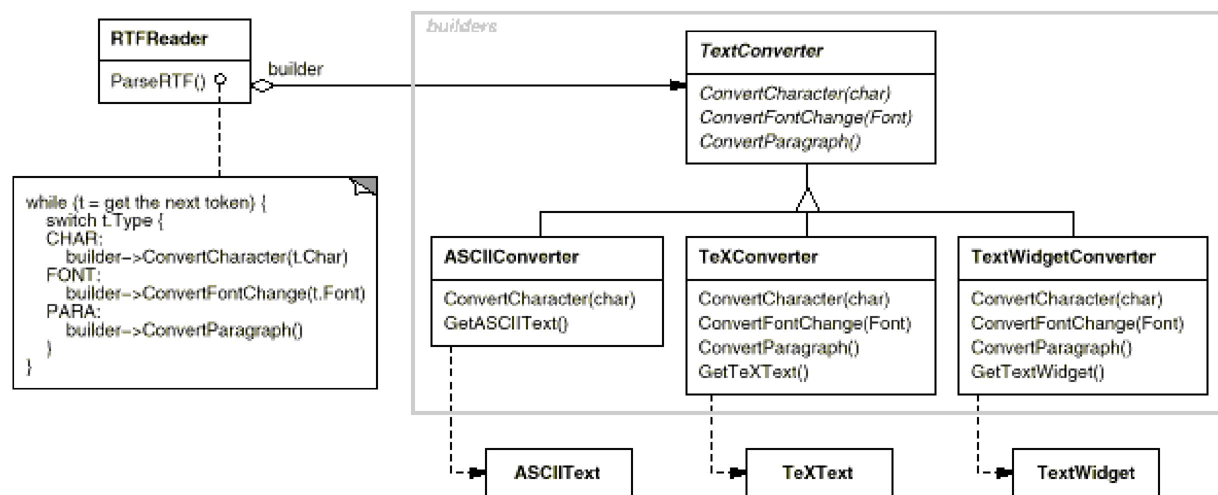
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Motivation

A reader for the RTF (Rich Text Format) document exchange format should be able to convert RTF to many text formats.

The reader might convert RTF documents into plain ASCII text or into a text widget that can be edited interactively. The problem, however, is that the number of possible conversions is open-ended. So it should be easy to add a new conversion without modifying the reader.

A solution is to configure the RTFReader class with a TextConverter object that converts RTF to another textual representation. As the RTFReader parses the RTF document, it uses the TextConverter to perform the conversion. Whenever the RTFReader recognizes an RTF token (either plain text or an RTF control word), it issues a request to the TextConverter to convert the token.



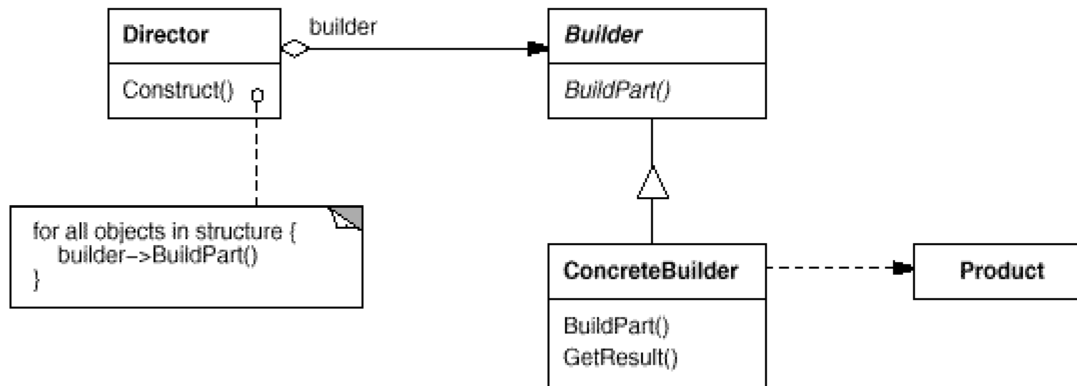
Applicability

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 the object 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).

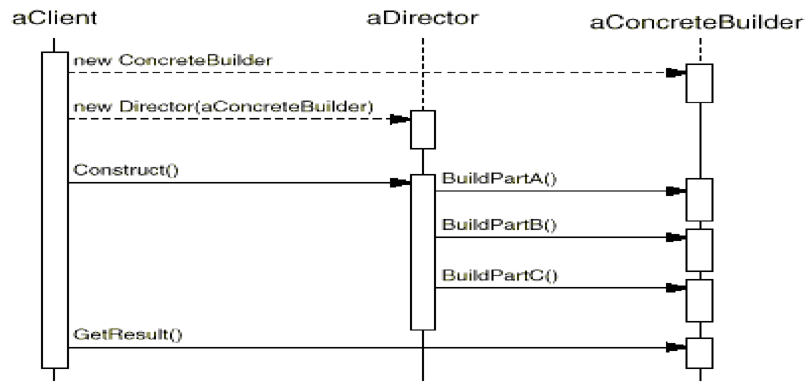
· **Director** (RTFReader)
constructs an object using the Builder interface.

· **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.

The following interaction diagram illustrates how Builder and Director cooperate with a client.


aClient
aDirector
aConcreteBuilder
new ConcreteBuilder
new Director(aConcreteBuilder)
Construct()
BuildPartA()
BuildPartB()
BuildPartC()
GetResult()

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 modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure; such classes don't appear in Builder's interface.

3. *It gives you finer control over the construction process.* Unlike creational patterns that construct products in one shot, the Builder pattern constructs

Implementation

Typically there's an abstract Builder class that defines an operation for each component that a director may ask it to create. The operations do nothing by default.

A ConcreteBuilder class overrides operations for components it's interested in creating.

Here are other implementation issues to consider:

1. *Assembly and construction interface.* Builders construct their products in step-by-step fashion. Therefore the Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders.

2. *Why no abstract class for products?* In the common case, 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.

Sample Code

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

The MazeBuilder class defines the following interface for building mazes:

```
class MazeBuilder {
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 Get Maze operation returns the maze to the client. Subclasses of Maze Builder will override this operation to return the maze that they build.

Given the MazeBuilder interface, we can change the CreateMaze member function to take this builder as a parameter.

Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
builder.BuildMaze();

```
builder.BuildRoom(1);
builder.BuildRoom(2);
builder.BuildDoor(1,   2);
return builder.GetMaze();
}
```

Known Uses

The RTF converter application is from ET++ [WGM88]. Its text building block uses a builder to process text stored in the RTF format.

Builder is a common pattern in Smalltalk-80 [Par90]:

· The Parser class in the compiler subsystem is a Director that takes a ProgramNodeBuilder object as an argument. A Parser object notifies its ProgramNodeBuilder object each time it recognizes a syntactic construct.

When the parser is done, it asks the builder for the parse tree it built
and returns it to the client.

Related Patterns

Abstract Factory is similar to Builder in that it too may construct complex objects.

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

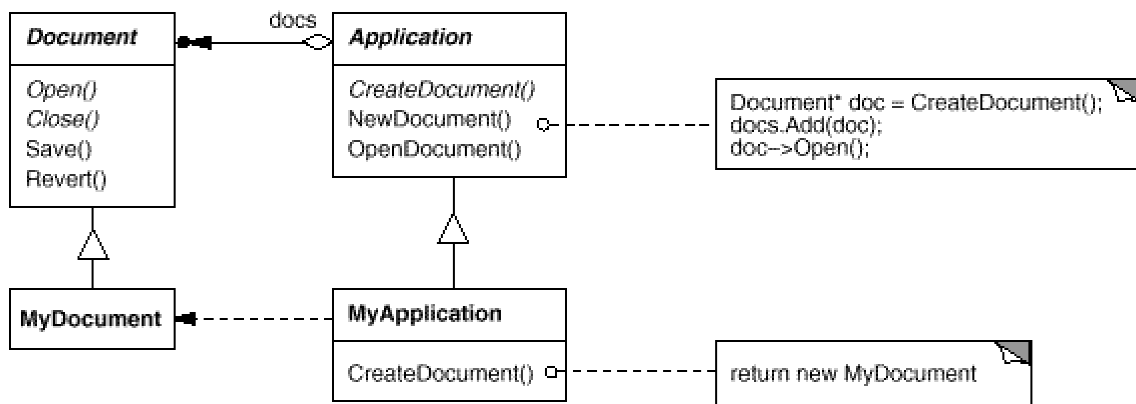
Virtual Constructor

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. Both classes are abstract, and clients have to subclass them to realize their application-specific implementations.

To create a drawing application, for example, we define the classes Drawing Application and DrawingDocument. The Application class is responsible for managing Documents and will create them as required.



Applicability

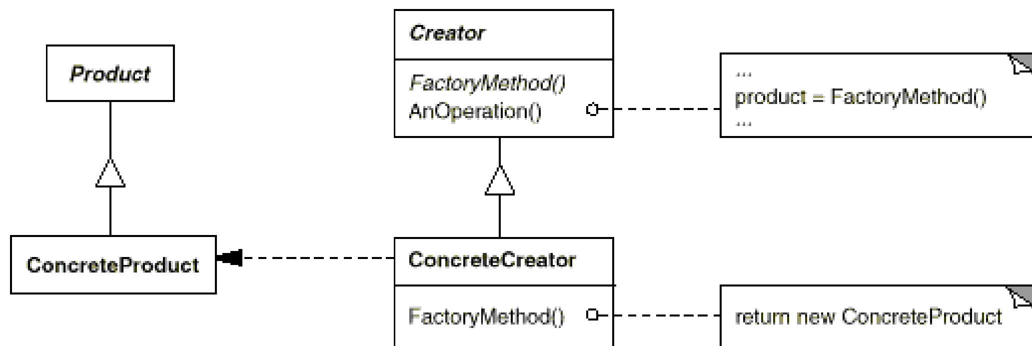
Use the Factory Method pattern when

· a class can't anticipate 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 ConcreteProduct.

Consequences

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

A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Subclassing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution.

Here are two additional consequences of the Factory Method pattern:

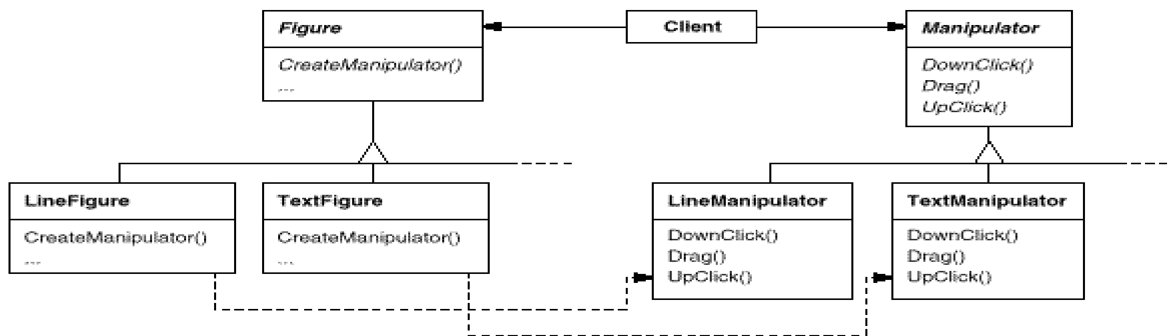
1. *Provides hooks for subclasses.* Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.

In the Document example, the Document class could define a factory method called CreateFileDialog that creates a default file dialog object for opening an existing document. A

Document subclass can define an application-specific file dialog by overriding this factory method. In this case the factory method is not abstract but provides a reasonable default implementation.

2. *Connects parallel class hierarchies.* In the examples we've considered so far, the factory method is only called by Creators. But this doesn't have to be the case; clients can find factory methods useful, especially in thecase of parallel class hierarchies.

Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class. Consider graphical figures that can be manipulated interactively; that is, they can be stretched, moved, or rotated using the mouse.
Figure class hierarchy:



Implementation

Consider the following issues when applying the Factory Method pattern:
1. *Two major varieties.* The two main variations of the Factory Method pattern are (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. It's also possible to have an abstract class that defines a default implementation, but this is less common.
The first case *requires* subclasses to define an implementation, because there's no reasonable default.
. In the second case, the concrete Creator uses the factory method primarily for flexibility. It's following a rule that says, "Create objects in a separate operation so that subclasses can override the way they're created." This rule ensures that designers of subclasses can change the class of objects their parent class instantiates if necessary.

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. In the Document example, Application might support different kinds of Documents. You pass CreateDocument an extra parameter to specify the kind of document to create.

A parameterized factory method has the following general form, where MyProduct and YourProduct are subclasses of Product:
class Creator {

public:

```
virtual Product* Create(ProductId);
};
Product* Creator::Create (ProductId id) {
if (id == MINE) return new MyProduct;
if (id == YOURS) return new YourProduct;
// repeat for remaining products...
return 0;
```

Overriding a parameterized factory method lets you easily and selectively extend or change the products that a Creator produces. You can introduce new identifiers for new kinds of products, or you can associate existing identifiers with different products.

Sample Code

The function CreateMaze 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; }
virtual Door* MakeDoor(Room* r1, Room* r2) const
{ return new Door(r1, r2); }
};
```

Each factory method returns a maze component of a given type. MazeGame provides default implementations that return the simplest kinds of maze, rooms, walls,
and doors. Now we can rewrite CreateMaze to use these factory methods:

```
Maze* MazeGame::CreateMaze () {
Maze* aMaze = MakeMaze();
Room*  r1  =  MakeRoom(1);
Room* r2 = MakeRoom(2);
Door* theDoor = MakeDoor(r1, r2);
aMaze->AddRoom(r1);
```

```
aMaze->AddRoom(r2);
r1->SetSide(North, MakeWall());
```

```
r1->SetSide(East, theDoor);
r1->SetSide(South, MakeWall());
r1->SetSide(West,  MakeWall());
r2->SetSide(North, MakeWall());
r2->SetSide(East,    MakeWall());
r2->SetSide(South, MakeWall());
r2->SetSide(West,        theDoor);
return aMaze;
}
```

Known Uses

Factory methods uses toolkits and frameworks. The preceding document example is a typical use in MacApp and ET++ [WGM88]. The manipulator example is from Unidraw. Class View 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 But subclasses of View specify the class of their default controller by defining defaultControllerClass,

A more esoteric example in Smalltalk-80 is the factory method parserClass defined by Behavior (a superclass of all objects representing classes). This enables a class to use a customized parser for its source code. For example, a client can define a class SQLParser to analyze the source code of a class with embedded SQL
statements.

Related Patterns

Abstract Factory is often implemented with factory methods. The Motivation example in the Abstract Factory pattern illustrates Factory Method as well.
Factory methods are usually called within Template Methods ..

……
.

Prototype

Intent

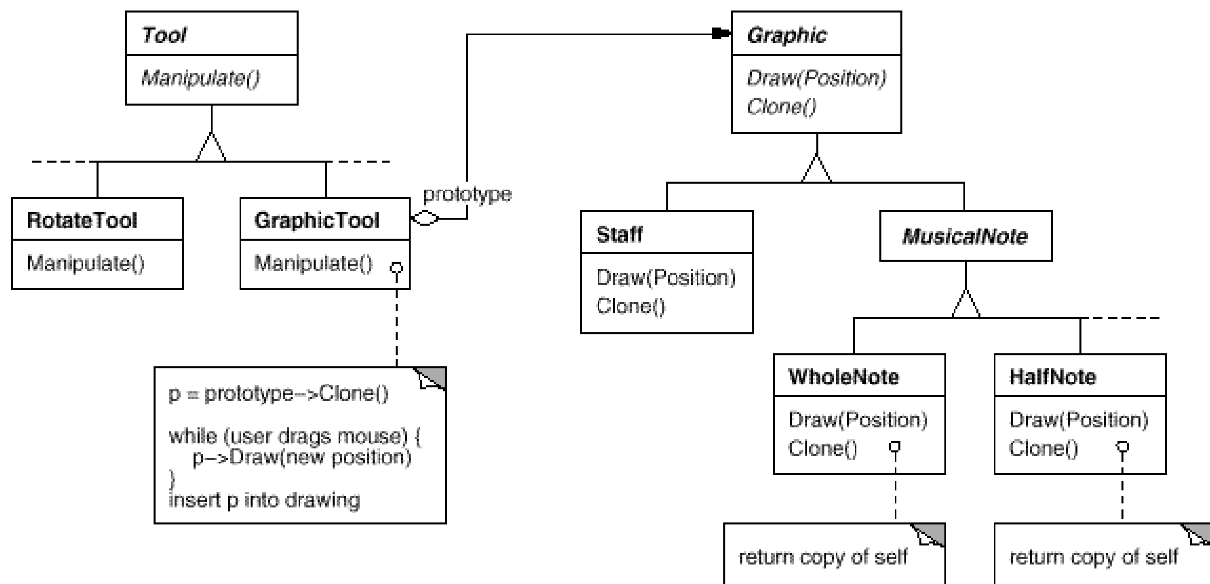
Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Motivation

You could build an editor for music scores by customizing a general framework for graphical editors and adding new objects that represent notes, rests, and staves.
The editor framework may have a palette of tools for adding these music objects to the score. The palette would also include tools for selecting, moving, and otherwise manipulating music

objects.

Users will click on the quarter-note tool and use it to add quarter notes to the score. Or they can use the move tool to move a note up or down on the staff, thereby changing its pitch.
Let's assume the framework provides an abstract Graphic class for graphical components, like notes and staves.
Moreover, it'll provide an abstract Tool class for defining tools like those in the palette. The framework also predefines a GraphicTool 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.
. The question is, how can the framework use it to parameterize instances of GraphicTool by the *class* of Graphic they're supposed to create?
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**. GraphicTool is 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.

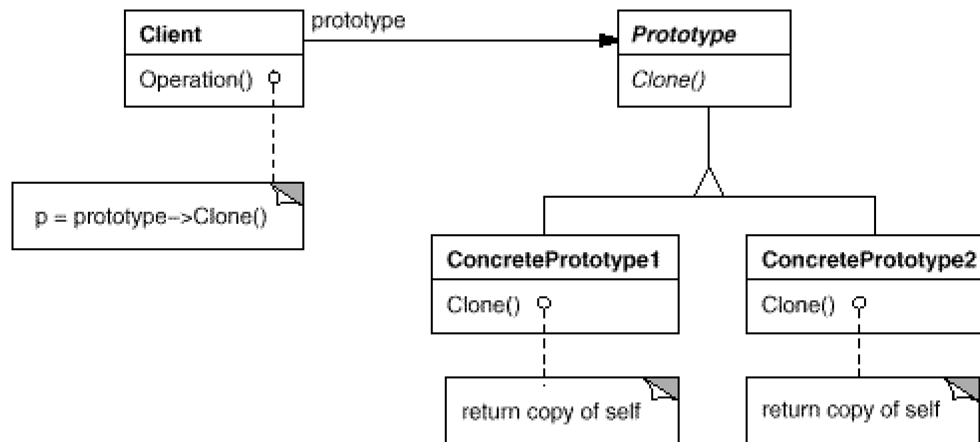


Applicability

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. It may be more convenient to install a corresponding number of prototypes and clone them rather than

instantiating the class manually, each time with the appropriate 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

·

Additional benefits of the Prototype pattern are listed below.

1. *Adding and removing products at run-time.* Prototypes let you incorporate a new concrete product class into a system simply by registering a prototypical instance with the client. That's a bit more flexible than other creational patterns, because a client can install and remove prototypes at run-time.

2. *Specifying new objects by varying values.* Highly dynamic systems let you define new behavior through object composition—by specifying values for an object's variables .In music editor, one GraphicTool class can create a limitless variety of music objects.

3. *Specifying new objects by varying structure.* Many applications build objects from parts and subparts. Editors for circuit design, for example, build circuits out of subcircuits.1 For convenience, such applications often let you instantiate complex, user-defined structures, say, to use a specific subcircuit again and again.

The Prototype pattern supports this as well. We simply add this subcircuit as a prototype to the palette of available circuit elements. As long as the composite circuit object implements Clone as a deep copy, circuits with different structures can be prototypes.
4. *Reduced subclassing.* Factory Method (121) often produces a hierarchy of Creator classes that parallels the product class hierarchy. The Prototype pattern lets you clone a prototype instead of asking a factory method to make a new object.

Implementation
Prototype is particularly useful with static languages like C++, where classes are not objects, and little or no type information is available at run-time.
It's less important in languages like Smalltalk or Objective C that provide what amounts to a prototype (i.e., a class object) for creating instances of each class. This pattern is built into prototype-based languages like Self [US87], in which all object creation happens by cloning a prototype.
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**. A prototype manager is an associative store that returns the prototype matching a given key. It has operations for registering a prototype undera key and for unregistering it. Clients can change or even browse through
the registry at run-time. This lets clients extend and take inventory on
the system without writing code.
2. *Implementing the Clone operation.* The hardest part of the Prototype pattern is implementing the Clone operation correctly. It's particularly tricky when object structures contain circular references.
Most languages provide some support for cloning objects. For example, Smalltalk provides an implementation of copy that's inherited by all subclasses of Object.
C++ provides a copy constructor. But these facilities don't solve the "shallow copy versus deep copy" problem [GR83].

Sample Code

We'll define a Maze Prototype Factory subclass of the Maze Factory class.
Maze Prototype Factory 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.
Maze Prototype Factory augments the Maze Factory 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;
};
```

The new constructor simply initializes its prototypes:

```
MazePrototypeFactory::MazePrototypeFactory        (
Maze* m, Wall* w, Room* r, Door* d
) {
_prototypeMaze = m;
_prototypeWall = w;
_prototypeRoom = r;
_prototypeDoor = d;
}
```

The member functions for creating walls, rooms, and doors are similar: Each clones a prototype and then initializes it. Here are the definitions of MakeWall and
MakeDoor:

Known Uses

The first example of the Prototype pattern was in Ivan Sutherland's Sketchpad system. an object-oriented language was in ThingLab, where users could form a composite object and then promote it to a prototype by installing it in a library of reusable
objects

The "interaction technique library" in Mode Composer stores prototypes of objects that support various interaction techniques . Any interaction technique created by the Mode Composer can be used as a prototype by placing it in this library. The Prototype pattern lets Mode Composer support an unlimited set of interaction techniques.

Related Patterns

Composite and Decorator

……………………………………………………………………………………………………

Singleton

**Inten
t**

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

Motivation

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler.

There should be only one file system and one window manager. A digital filter will have

one A/D converter. An accounting system will be dedicated to serving one company. 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 (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

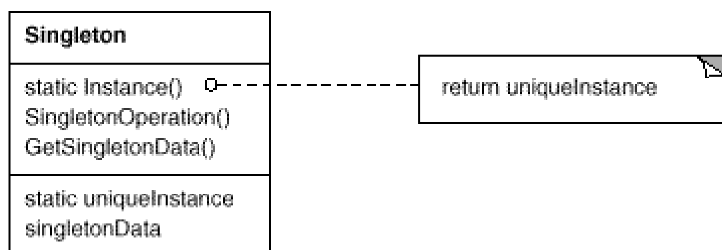
Applicability

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 (that is, a class method in Smalltalk and a static member function in C++).

. 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.* The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you 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. Moreover, you can use the same approach to control the number of instances that the application uses.
5. *More flexible than class operations.* Another way to package a singleton's functionality is to use class operations (that is, static member functions in C++ or class methods in Smalltalk). But both of these language techniques make it hard to change a design to allow more than one instance of a class. Moreover, static member functions in C++ are never virtual, so subclasses can't override them polymorphically.

Implementation

Here are implementation issues to consider when using the Singleton pattern:
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 is, either a static member function or a class method) that guarantees only one instance is created.
Singleton also defines a static member variable _instance that contains a pointer to its unique instance.
The Singleton class is declared as

```
class Singleton {
public:
static Singleton* Instance();
protected:
Singleton();
private:
static Singleton* _instance;
};
```

The corresponding implementation is

```
Singleton* Singleton::_instance = 0;
Singleton* Singleton::Instance () {
if (_instance == 0) {
_instance = new Singleton;
}
return _instance;
}
```

Sample Code

where the Singleton pattern comes in. By making the MazeFactory a singleton, we make the

maze object globally accessible without resorting to global variables.

For simplicity, let's assume we'll never subclass MazeFactory. (We'll consider the alternative in a moment.) We make it a Singleton class in C++ by adding a static Instance operation and a static _instance member to hold the one and only instance.

We must also protect the constructor to prevent accidental instantiation, which might lead to more than one instance.

```
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;
}
```

Now let's consider what happens when there are subclasses of MazeFactory, and the application must decide which one to use. We'll select the kind of maze through an environment variable and add code that instantiates the proper MazeFactory
subclass based on the environment variable's value. The Instance operation is a good place to put this code, because it already instantiates MazeFactory:

```
MazeFactory* MazeFactory::Instance () {
if (_instance == 0) {
const char* mazeStyle = getenv("MAZESTYLE");
if (strcmp(mazeStyle, "bombed") == 0) {
_instance = new BombedMazeFactory;
} else if (strcmp(mazeStyle, "enchanted") == 0) {
_instance = new EnchantedMazeFactory;
// ... other possible subclasses
} else { // default
_instance = new MazeFactory;
}
}
return _instance;
}
```

.

Known Uses

An example of the Singleton pattern in Smalltalk-80 [Par90] is the set of changes to the code, which is ChangeSet current. A more subtle example is the relationship between classes and their

metaclasses. A metaclass is the class of a class, and each metaclass has one instance. Metaclasses do not have names .
The InterViews user interface toolkit [LCI+92] uses the Singleton pattern to access the unique instance of its Session and WidgetKit classes, among others.

Related Patterns

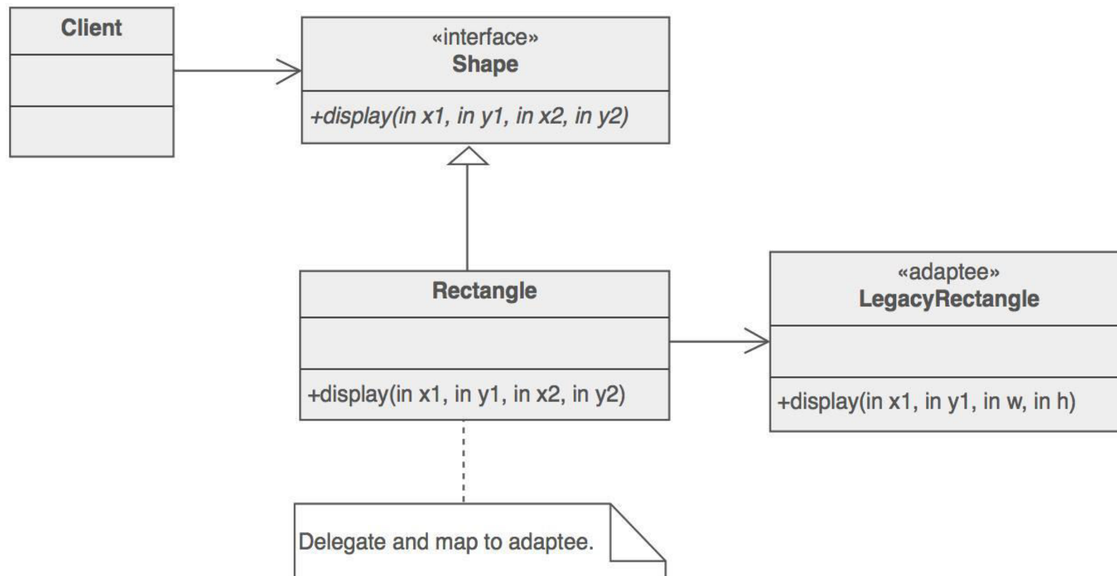
Many patterns can be implemented using the Singleton pattern. See Abstract Factory, Builder , and Prototype .

UNIT-3

Structural Pattern Part-I

Structural patterns

In Software Engineering, Structural Design Patterns are Design Patterns that ease the design by identifying a simple way to realize relationships between entities.



- Adapter
  Match interfaces of different classes

- Bridge
  Separates an object's interface from its implementation

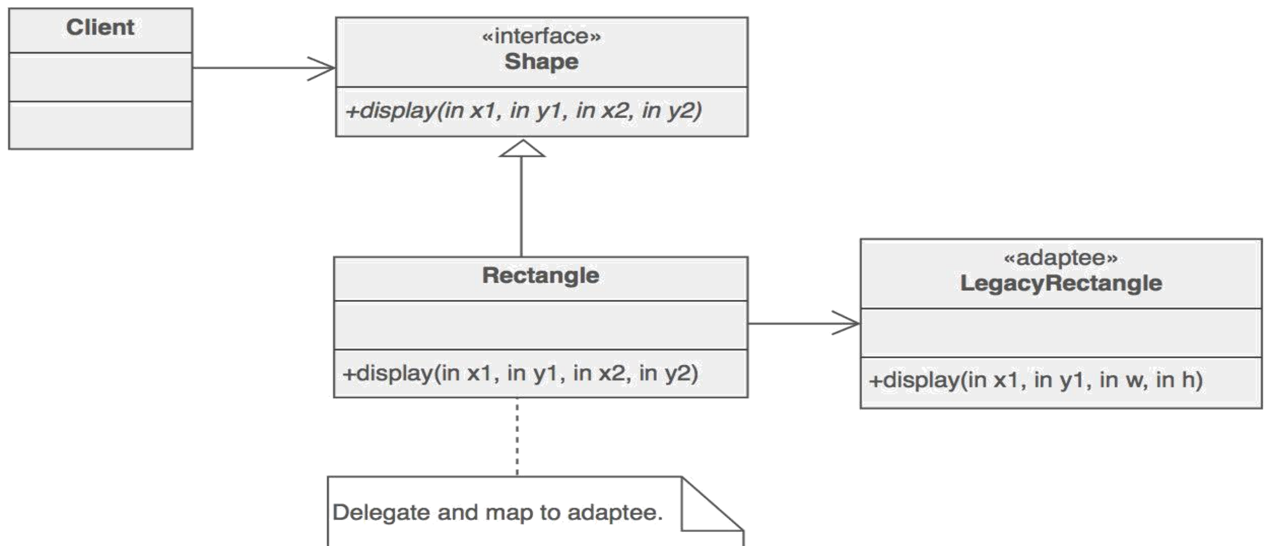
- Composite
  A tree structure of simple and composite objects

- Decorator

  Add responsibilities to objects dynamically

- Facade
  A single class that represents an entire subsystem

- Flyweight
  A fine-grained instance used for efficient sharing



Private Class Data
Restricts accessor/mutator access

Proxy
An object representing another object

Rules of thumb

1. Adapter makes things work after they're designed; Bridge makes them work before they are.

2. Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.

3. Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

4. Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.

5. Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.

6. Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes.

7. Composite can let you compose a Mediator out of smaller pieces through recursive composition.

8. Decorator lets you change the skin of an object. Strategy lets you change the guts.

9. Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.

10. Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

11. Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.

12. Facade objects are often Singleton because only one Facade object is required.

13. Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.

14. Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.

15. Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.

16. Flyweight is often combined with Composite to implement shared leaf nodes.

17. Flyweight explains when and how State objects can be shared.

Adapter Design Patterns

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.

- Wrap an existing class with a new interface.

- Impedance match an old component to a new system

Problem

An "off the shelf" component offers compelling functionality that you would like to reuse, but its "view of the world" is not compatible with the philosophy and architecture of the system currently being developed.

Discussion

Reuse has always been painful and elusive. One reason has been the tribulation of designing something new, while reusing something old. There is always something not quite right between the old and the new. It may be physical dimensions or misalignment. It may be timing or synchronization. It may be unfortunate assumptions or competing standards.

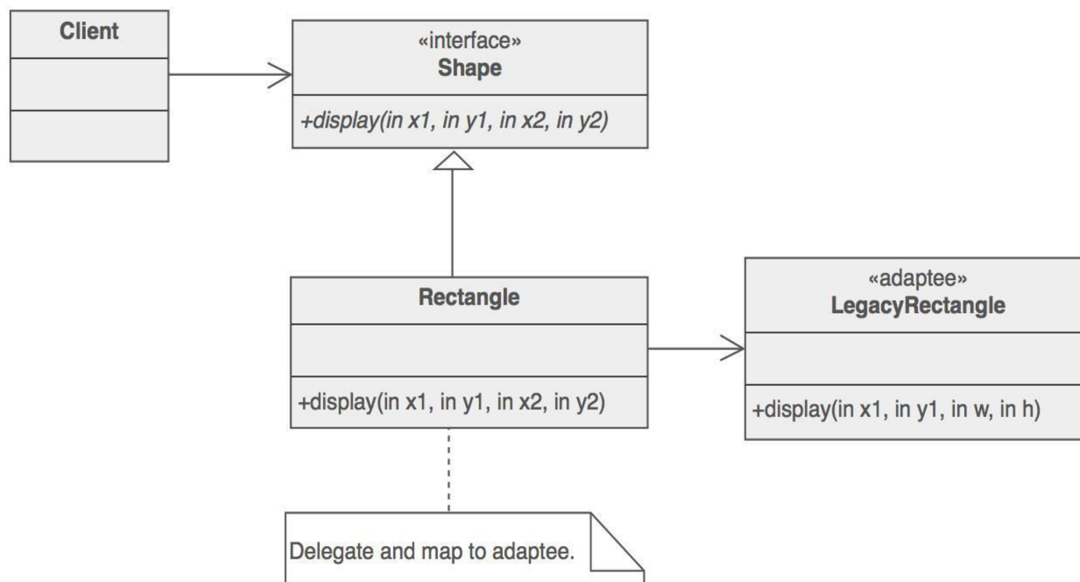
It is like the problem of inserting a new three-prong electrical plug in an old two-prong wall outlet – some kind of adapter or intermediary is necessary.

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

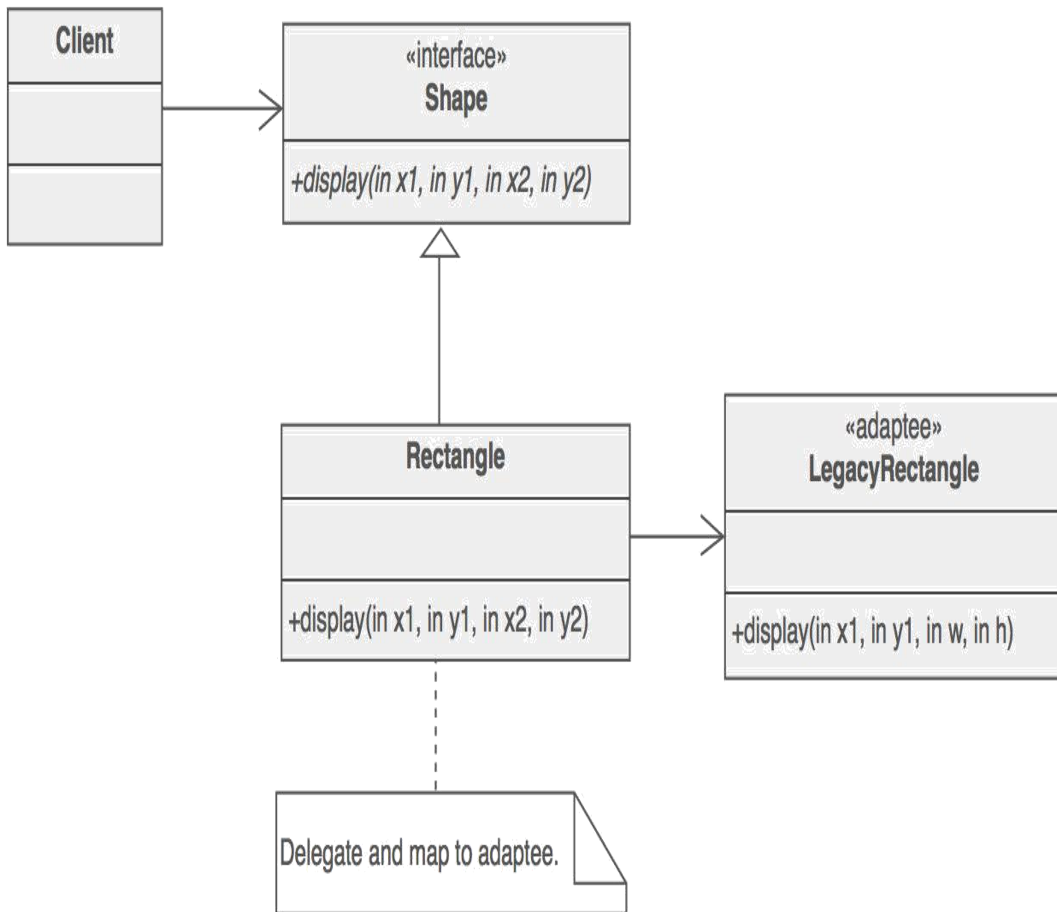
Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class.

Structure

Below, a legacy Rectangle component's display() method expects to receive "x, y, w, h" parameters. But the client wants to pass "upper left x and y" and "lower right x and y". This incongruity can be reconciled by adding an additional level of indirection – i.e. an Adapter object.

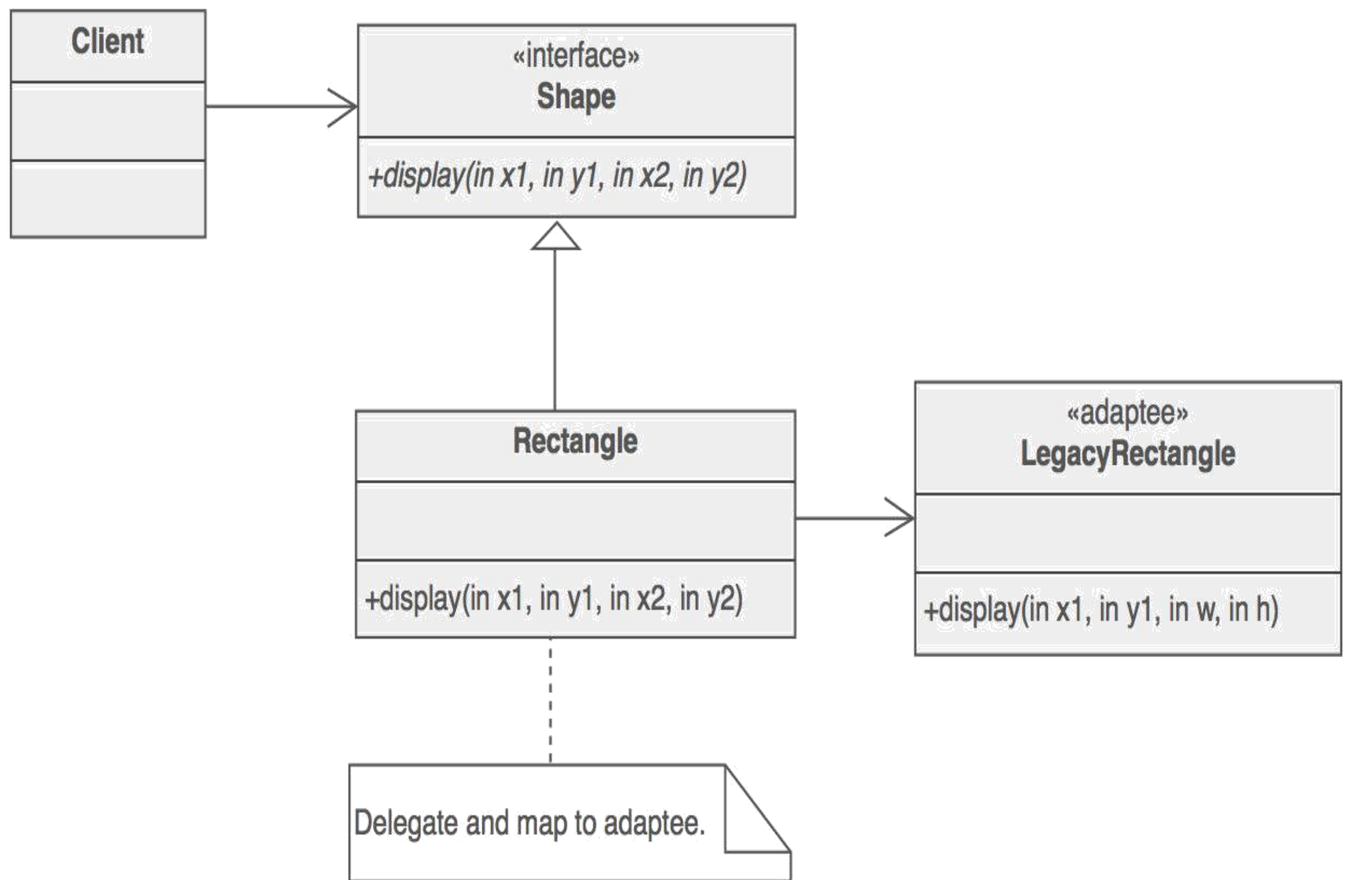


The Adapter could also be thought of as a "wrapper".



Example

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



Check list

1. Identify the players: the component(s) that want to be accommodated (i.e. the client), and the component that needs to adapt (i.e. the adaptee).

2. Identify the interface that the client requires.

3. Design a "wrapper" class that can "impedance match" the adaptee to the client.

4. The adapter/wrapper class "has a" instance of the adaptee class.

5. The adapter/wrapper class "maps" the client interface to the adaptee interface.

6. The client uses (is coupled to) the new interface

Rules of thumb

- Adapter makes things work after they're designed; Bridge makes them work before they are.

- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

- Adapter is meant to change the interface of an existing object. Decorator enhances another object without changing its interface. Decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.

- Facade defines a new interface, whereas Adapter reuses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.

Bridge Design Pattern

Intent

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

- Publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.

- Beyond encapsulation, to insulation

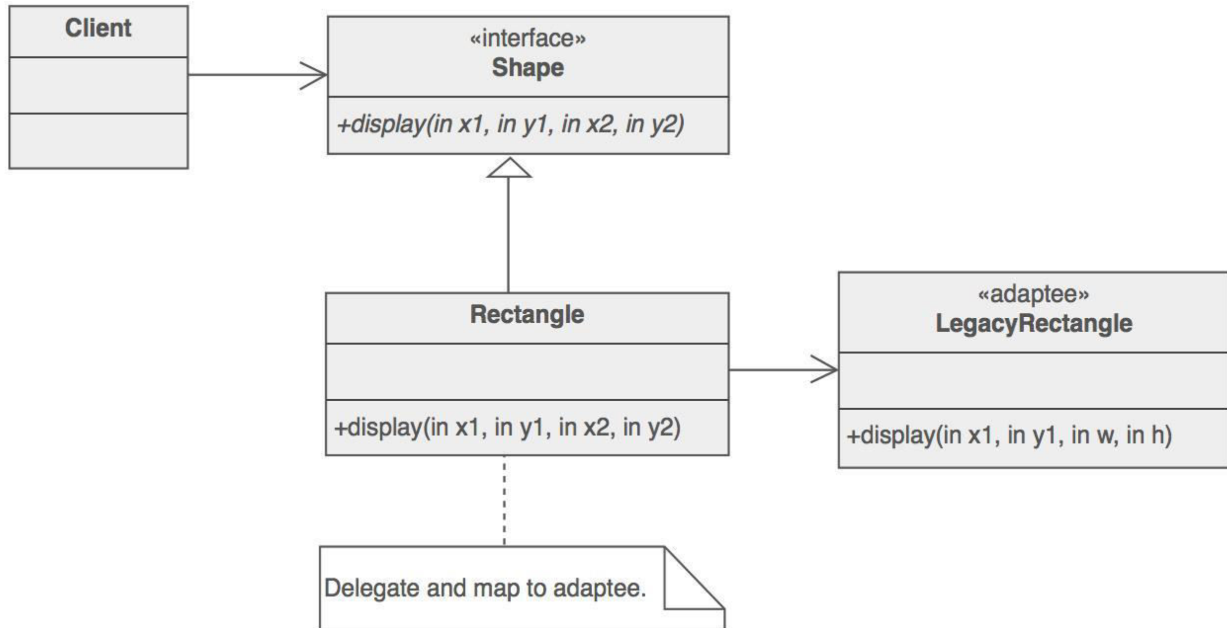
Problem

"Hardening of the software arteries" has occurred by using subclassing of an abstract base class to provide alternative implementations. This locks in compile-time binding between

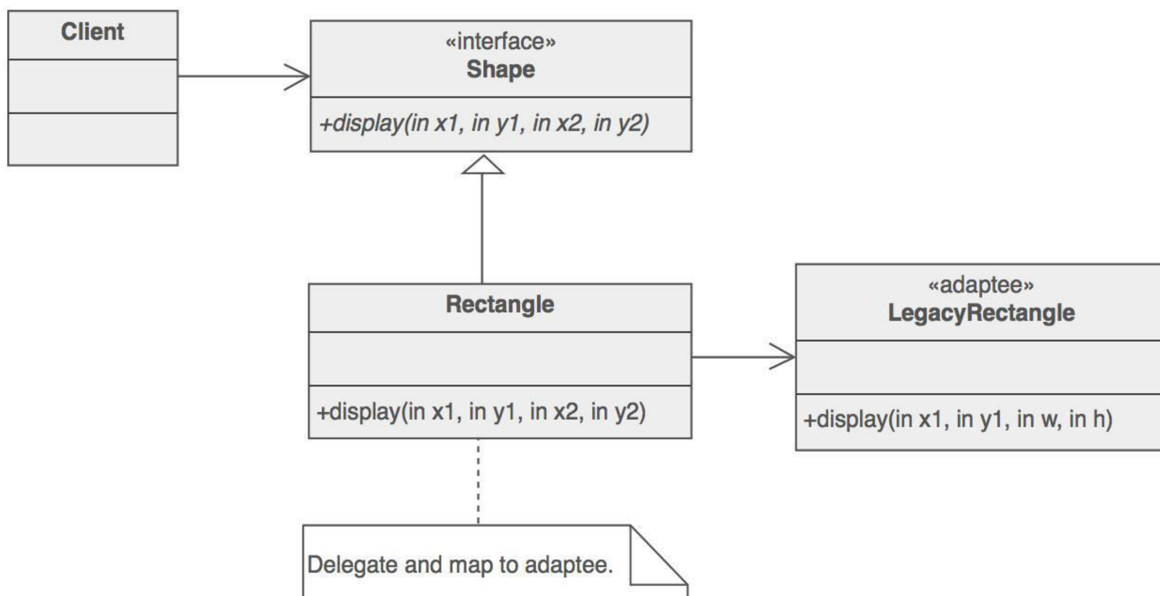
interface and implementation. The abstraction and implementation cannot be independently extended or composed.

Motivation

Consider the domain of "thread scheduling".

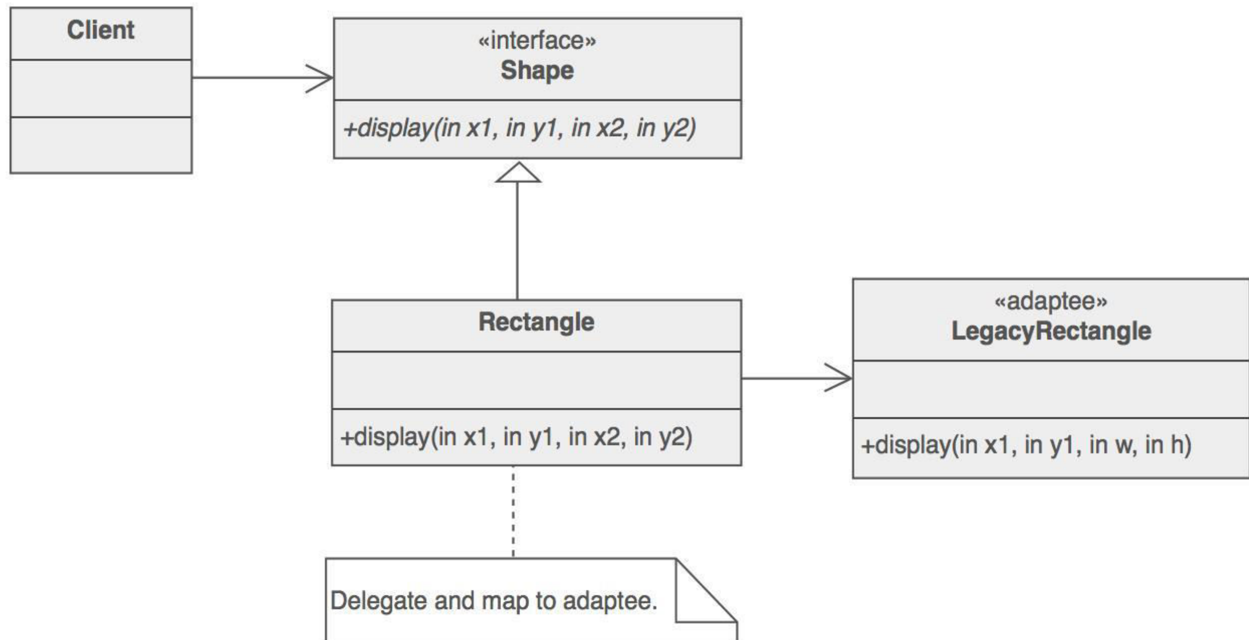


There are two types of thread schedulers, and two types of operating systems or "platforms". Given this approach to specialization, we have to define a class for each permutation of these two dimensions. If we add a new platform (say ... Java's Virtual Machine), what would our hierarchy look like?



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

The Bridge design pattern proposes refactoring this exponentially explosive inheritance hierarchy into two orthogonal hierarchies – one for platform-independent abstractions, and the other for platform-dependent implementations.



Discussion

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

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

Use the Bridge pattern when:

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

Consequences include:

- decoupling the object's interface,
- improved extensibility (you can extend (i.e. subclass) the abstraction and implementation hierarchies independently),

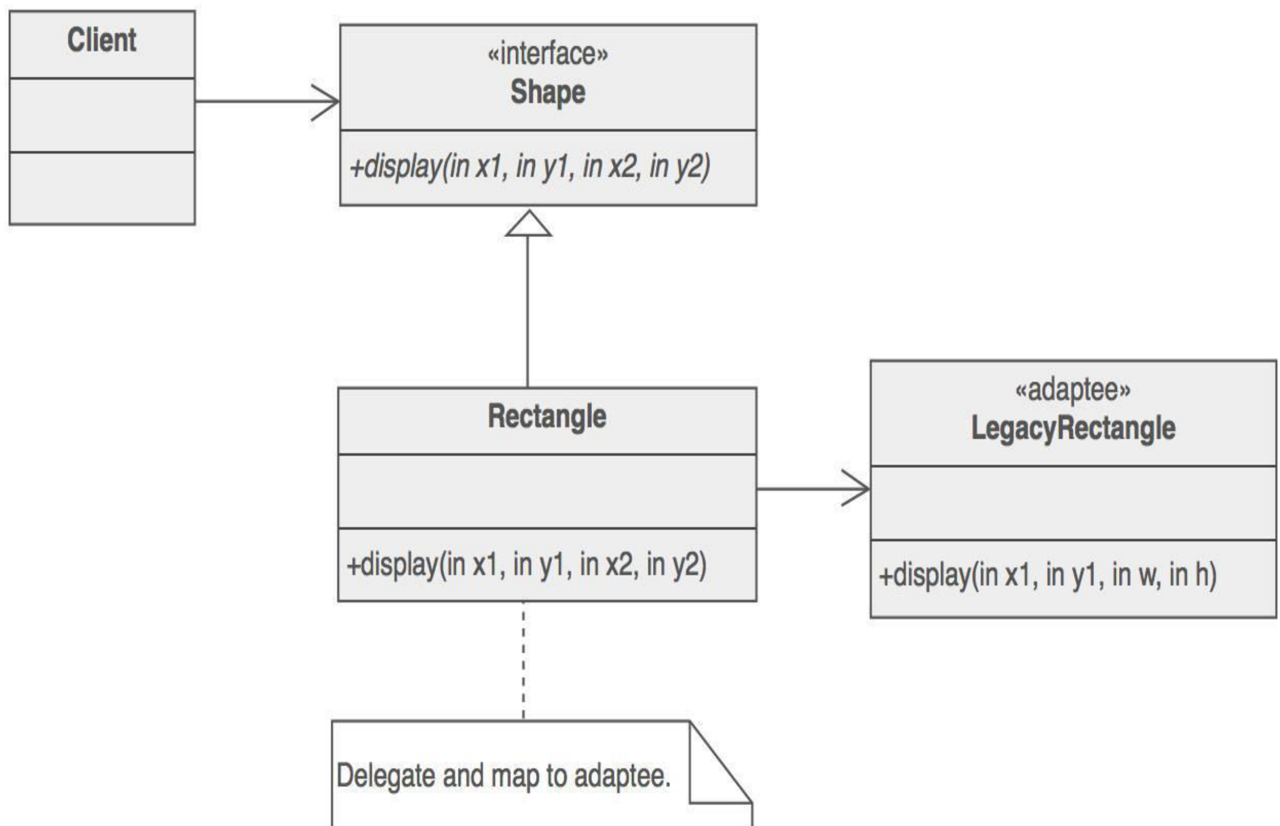
- hiding details from clients.

Bridge is a synonym for the "handle/body" idiom. This is a design mechanism that encapsulates an implementation class inside of an interface class. The former is the body, and the latter is the handle. The handle is viewed by the user as the actual class, but the work is done in the body. "The handle/body class idiom may be used to decompose a complex abstraction into smaller, more manageable classes. The idiom may reflect the sharing of a single resource by multiple classes that control access to it (e.g. reference counting)."

Structure

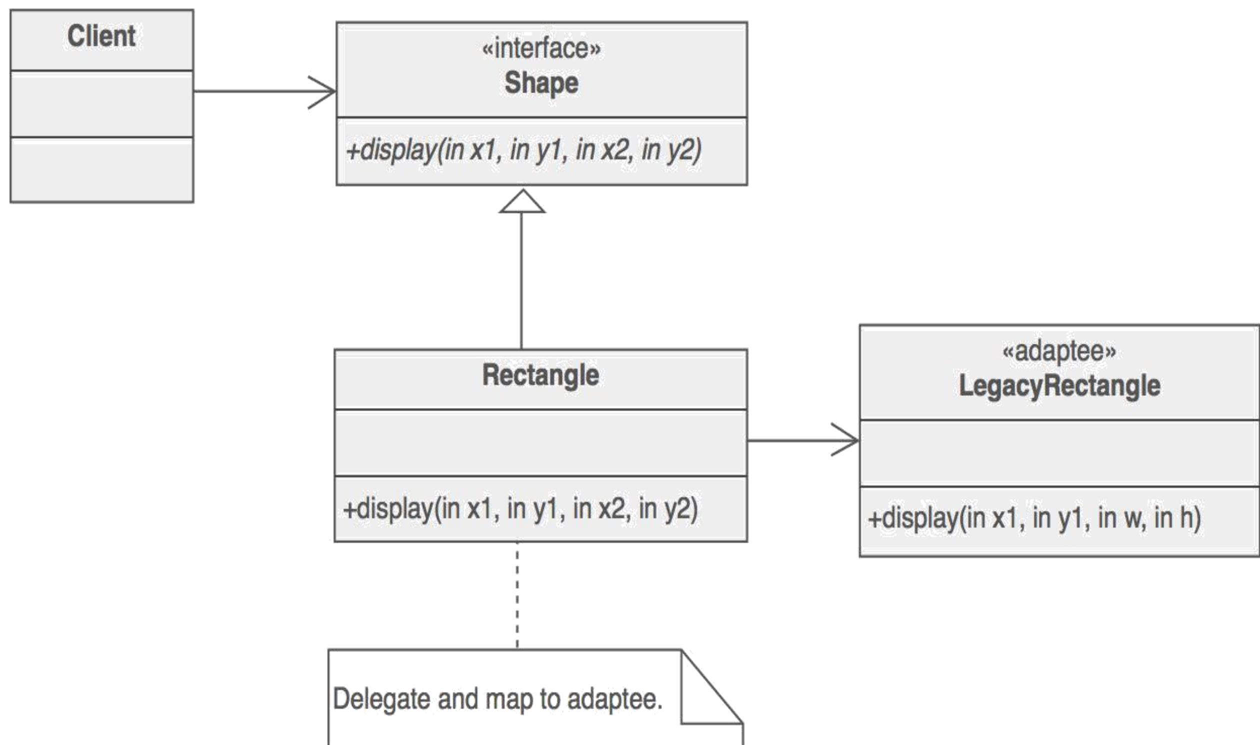
The Client doesn't want to deal with platform-dependent details. The Bridge pattern encapsulates this complexity behind an abstraction "wrapper".

Bridge emphasizes identifying and decoupling "interface" abstraction from "implementation" abstraction.



Example

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



Check list

1. Decide if two orthogonal dimensions exist in the domain. These independent concepts could be: abstraction/platform, or domain/infrastructure, or front-end/back-end, or interface/implementation.

2. Design the separation of concerns: what does the client want, and what do the platforms provide.

3. Design a platform-oriented interface that is minimal, necessary, and sufficient. Its goal is to decouple the abstraction from the platform.

4. Define a derived class of that interface for each platform.

5. Create the abstraction base class that "has a" platform object and delegates the platform-oriented functionality to it.

6. Define specializations of the abstraction class if desired.

Rules of thumb

- Adapter makes things work after they're designed; Bridge makes them work before they are.

- Bridge is designed up-front to let the abstraction and the implementation vary independently. Adapter is retrofitted to make unrelated classes work together.

- State, Strategy, Bridge (and to some degree Adapter) have similar solution structures. They all share elements of the "handle/body" idiom. They differ in intent - that is, they solve different problems.

- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.

- If interface classes delegate the creation of their implementation classes (instead of creating/coupling themselves directly), then the design usually uses the Abstract Factory pattern to create the implementation objects.

Composite Design Pattern

Intent

- Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Recursive composition
- "Directories contain entries, each of which could be a directory."
- 1-to-many "has a" up the "is a" hierarchy

Problem

Application needs to manipulate a hierarchical collection of "primitive" and "composite" objects. Processing of a primitive object is handled one way, and processing of a composite object is handled differently. Having to query the "type" of each object before attempting to process it is not desirable.

Discussion

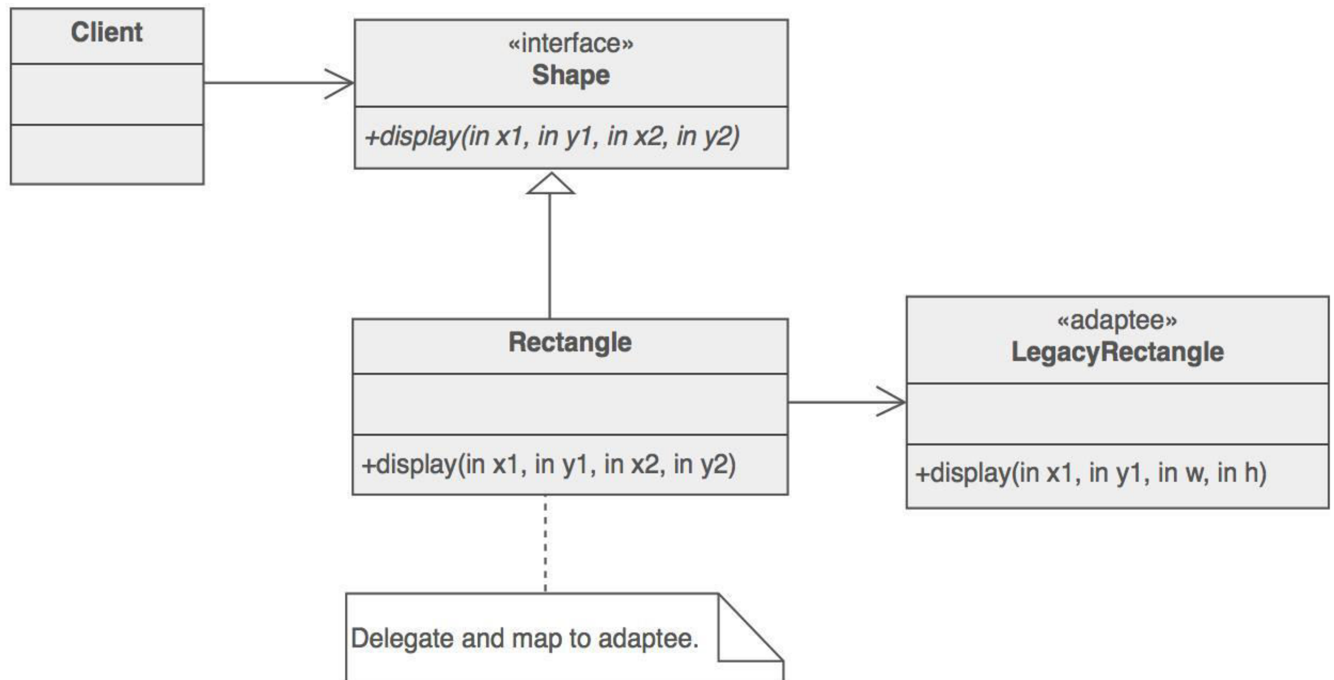
Define an abstract base class (Component) that specifies the behavior that needs to be exercised uniformly across all primitive and composite objects. Subclass the Primitive and Composite classes off of the Component class. Each Composite object "couples" itself only to the abstract type Component as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

Use this pattern whenever you have "composites that contain components, each of which could be a composite".

Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Primitives and Composites uniformly requires that these methods be moved to the abstract Component class. See the "Opinions" section below for a discussion of "safety" versus "transparency" issues.

Structure

Composites that contain Components, each of which could be a Composite.



Menus that contain menu items, each of which could be a menu.

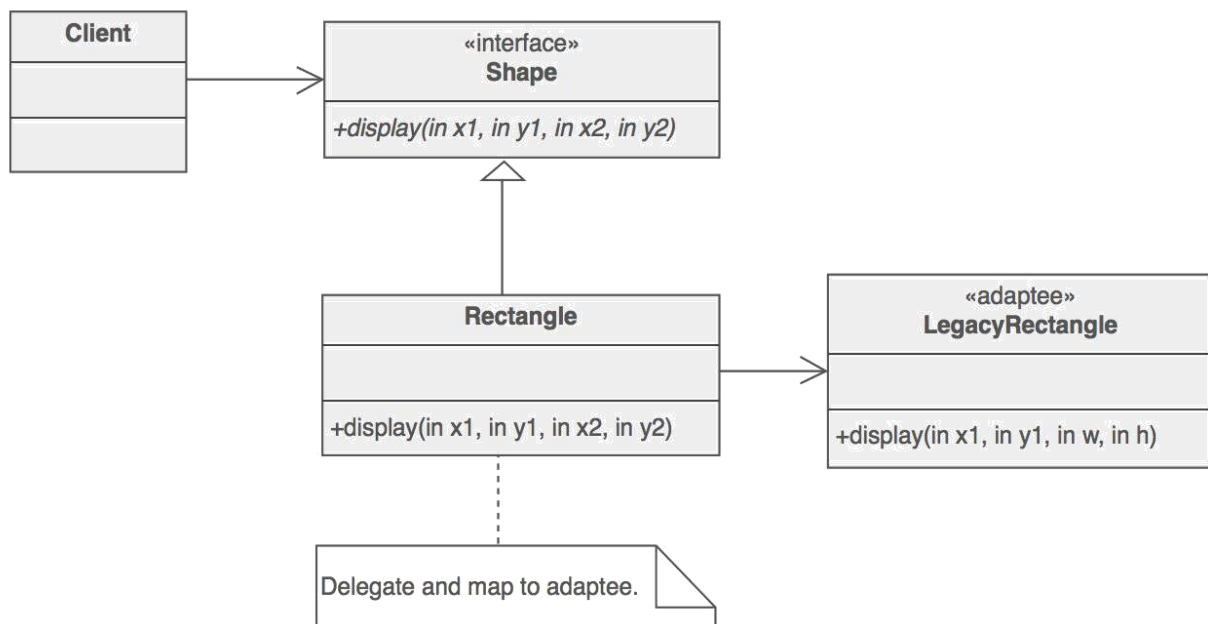
Row-column GUI layout managers that contain widgets, each of which could be a row-column GUI layout manager.

Directories that contain files, each of which could be a directory.

Containers that contain Elements, each of which could be a Container.

Example

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



Check list

1. Ensure that your problem is about representing "whole-part" hierarchical relationships.

2. Consider the heuristic, "Containers that contain containees, each of which could be a container." For example, "Assemblies that contain components, each of which could be an assembly." Divide your domain concepts into container classes, and containee classes.

3. Create a "lowest common denominator" interface that makes your containers and containees interchangeable. It should specify the behavior that needs to be exercised uniformly across all containee and container objects.

4. All container and containee classes declare an "is a" relationship to the interface.

5. All container classes declare a one-to-many "has a" relationship to the interface.

6. Container classes leverage polymorphism to delegate to their containee objects.

7. Child management methods [e.g. addChild(), removeChild()] should normally be defined in the Composite class. Unfortunately, the desire to treat Leaf and Composite objects uniformly may require that these methods be promoted to the abstract Component class. See the Gang of Four for a discussion of these "safety" versus "transparency" trade-offs.

Rules of thumb

- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.

- Composite can be traversed with Iterator. Visitor can apply an operation over a Composite. Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these

properties on parts of the composition. It could use Observer to tie one object structure to another and State to let a component change its behavior as its state changes.

- Composite can let you compose a Mediator out of smaller pieces through recursive composition.

- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.

- Flyweight is often combined with Composite to implement shared leaf nodes.

Opinions

The whole point of the Composite pattern is that the Composite can be treated atomically, just like a leaf. If you want to provide an Iterator protocol, fine, but I think that is outside the pattern itself. At the heart of this pattern is the ability for a client to perform operations on an object without needing to know that there are many objects inside.

Being able to treat a heterogeneous collection of objects atomically (or transparently) requires that the "child management" interface be defined at the root of the Composite class hierarchy (the abstract Component class). However, this choice costs you safety, because clients may try to do meaningless things like add and remove objects from leaf objects. On the other hand, if you "design for safety", the child management interface is declared in the Composite class, and you lose transparency because leaves and Composites now have different interfaces.

Smalltalk implementations of the Composite pattern usually do not have the interface for managing the components in the Component interface, but in the Composite interface. C++ implementations tend to put it in the Component interface. This is an extremely interesting fact, and one that I often ponder. I can offer theories to explain it, but nobody knows for sure why it is true.

My Component classes do not know that Composites exist. They provide no help for navigating Composites, nor any help for altering the contents of a Composite. This is because I would like the base class (and all its derivatives) to be reusable in contexts that do not require Composites. When given a base class pointer, if I absolutely need to know whether or not it is a Composite, I will use `dynamic_cast` to figure this out. In those cases where `dynamic_cast` is too expensive, I will use a Visitor.

Common complaint: "if I push the Composite interface down into the Composite class, how am I going to enumerate (i.e. traverse) a complex structure?" My answer is that when I have behaviors which apply to hierarchies like the one presented in the Composite pattern, I typically use Visitor, so enumeration isn't a problem - the Visitor knows in each case, exactly what kind of object it's dealing with. The Visitor doesn't need every object to provide an enumeration interface.

Composite doesn't force you to treat all Components as Composites. It merely tells you to put all operations that you want to treat "uniformly" in the Component class. If add, remove, and similar operations cannot, or must not, be treated uniformly, then do not put them in the Component base class. Remember, by the way, that each pattern's structure diagram doesn't define the pattern; it merely depicts what in our experience is a common realization thereof.

Structural Pattern Part-II

Decorator Design Pattern

Intent

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

- Client-specified embellishment of a core object by recursively wrapping it.

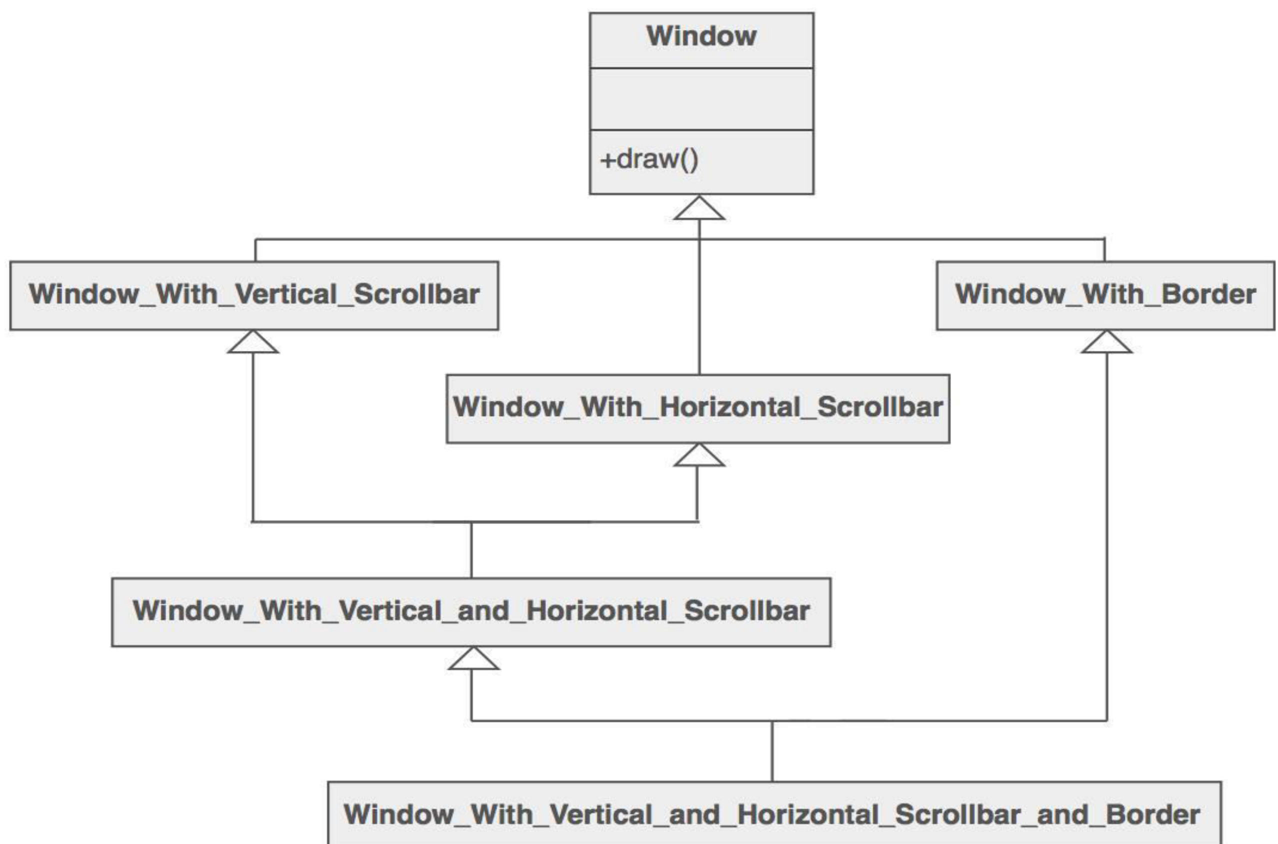
- Wrapping a gift, putting it in a box, and wrapping the box.

Problem

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

Discussion

Suppose you are working on a user interface toolkit and you wish to support adding borders and scroll bars to windows. You could define an inheritance hierarchy like ...



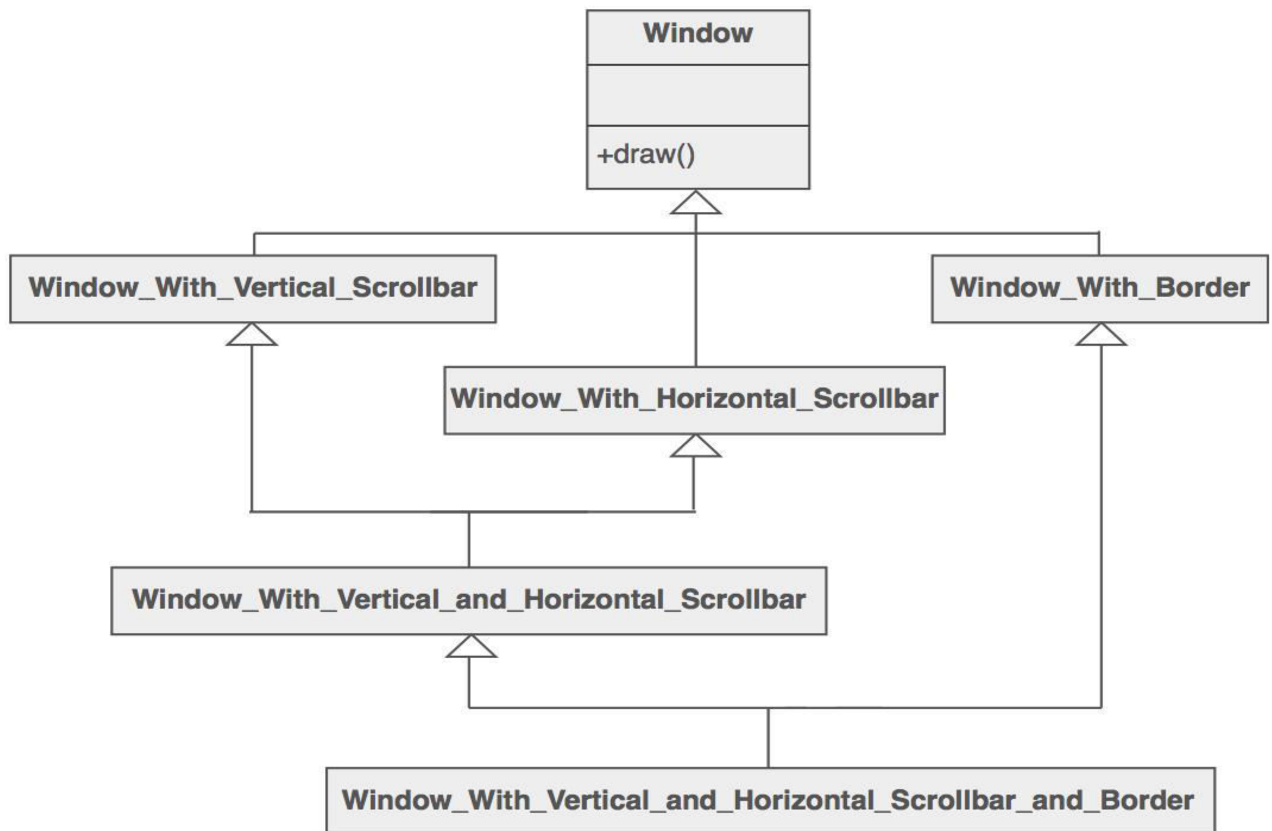
But the Decorator pattern suggests giving the client the ability to specify whatever combination of "features" is desired.

```
Widget* aWidget = new BorderDecorator(

  new HorizontalScrollBarDecorator(
```

```
  new VerticalScrollBarDecorator(

   new Window( 80, 24 ))));
```

aWidget->draw();

This flexibility can be achieved with the following design



Another example of cascading (or chaining) features together to produce a custom object might look like ...

```
Stream* aStream = new CompressingStream(
  new ASCII7Stream(
    new FileStream("fileName.dat")));
aStream->putString( "Hello world" );
```

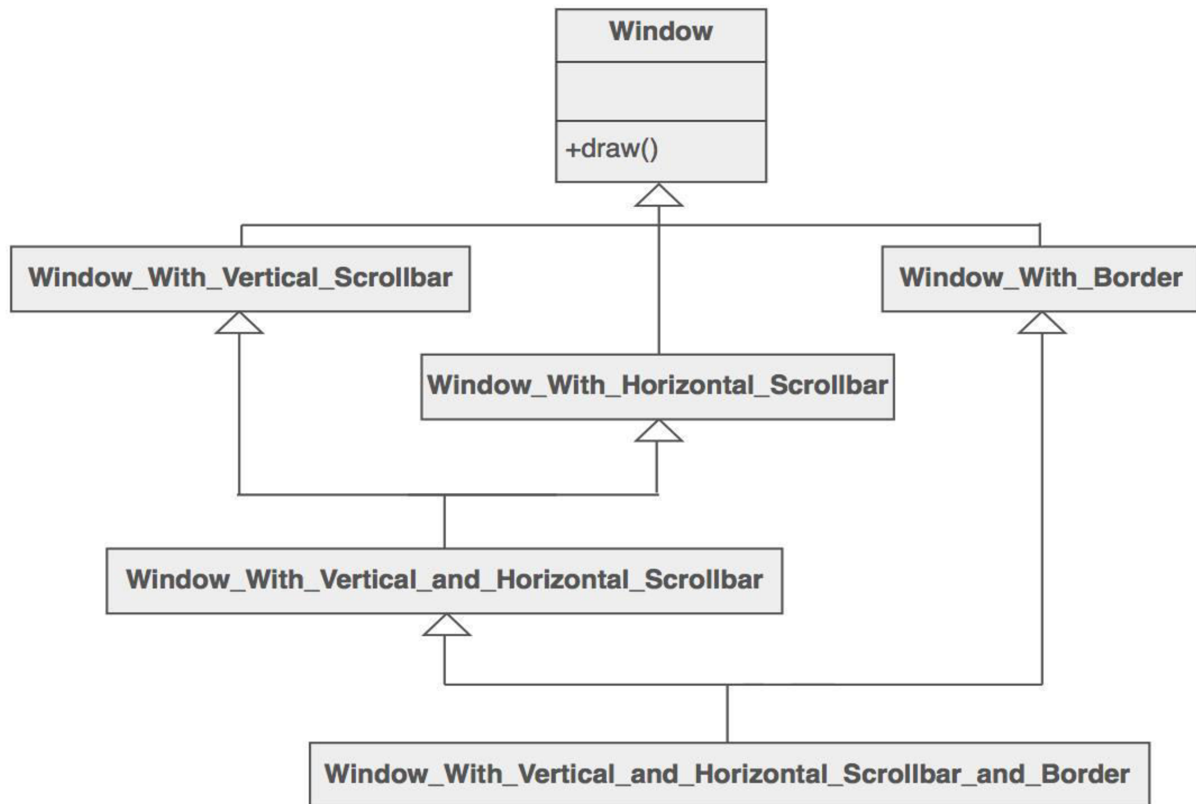
The solution to this class of problems involves encapsulating the original object inside an abstract wrapper interface. Both the decorator objects and the core object inherit from this abstract interface. The interface uses recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

Note that this pattern allows responsibilities to be added to an object, not methods to an object's interface. The interface presented to the client must remain constant as successive layers are specified.

Also note that the core object's identity has now been "hidden" inside of a decorator object. Trying to access the core object directly is now a problem.

Structure

The client is always interested in CoreFunctionality.doThis(). The client may, or may not, be interested in OptionalOne.doThis() and OptionalTwo.doThis(). Each of these classes always delegate to the Decorator base class, and that class always delegates to the contained "wrappee" object.

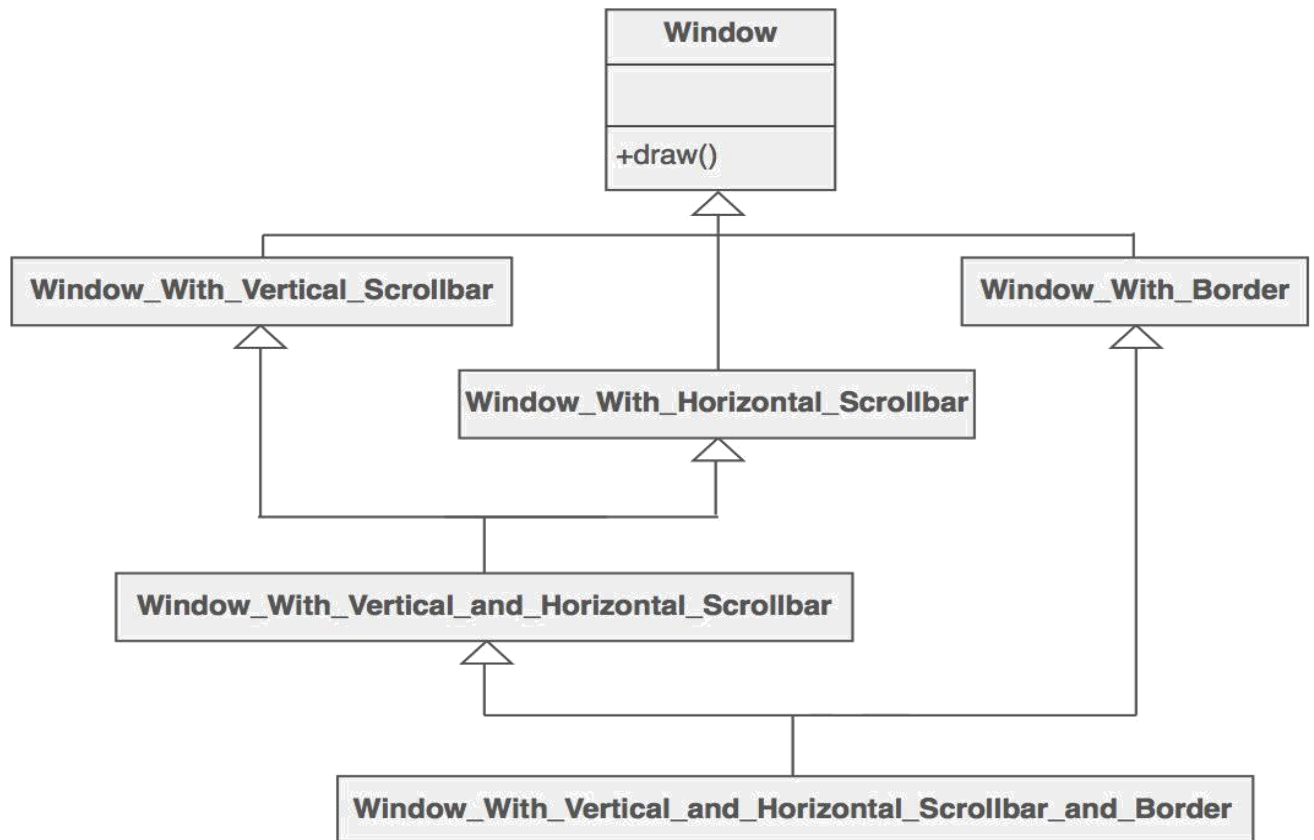


Example

The Decorator attaches additional responsibilities to an object dynamically. The ornaments that are added to pine or fir trees are examples of Decorators. Lights, garland, candy canes, glass ornaments, etc., can be added to a tree to give it a festive look. The ornaments do not

change the tree itself which is recognizable as a Christmas tree regardless of particular ornaments used. As an example of additional functionality, the addition of lights allows one to "light up" a Christmas tree.

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



Check list

1. Ensure the context is: a single core (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all.

2. Create a "Lowest Common Denominator" interface that makes all classes interchangeable.

3. Create a second level base class (Decorator) to support the optional wrapper classes.

4. The Core class and Decorator class inherit from the LCD interface.

5. The Decorator class declares a composition relationship to the LCD interface, and this data member is initialized in its constructor.

6. The Decorator class delegates to the LCD object.

7. Define a Decorator derived class for each optional embellishment.

8. Decorator derived classes implement their wrapper functionality - and - delegate to the Decorator base class.

9. The client configures the type and ordering of Core and Decorator objects.

Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

- Adapter changes an object's interface, Decorator enhances an object's responsibilities. Decorator is thus more transparent to the client. As a consequence, Decorator supports recursive composition, which isn't possible with pure Adapters.

- Composite and Decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.

- A Decorator can be viewed as a degenerate Composite with only one component. However, a Decorator adds additional responsibilities - it isn't intended for object aggregation.

- Decorator is designed to let you add responsibilities to objects without subclassing. Composite's focus is not on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.

- Composite could use Chain of Responsibility to let components access global properties through their parent. It could also use Decorator to override these properties on parts of the composition.

- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

- Decorator lets you change the skin of an object. Strategy lets you change the guts.

Facade Design Pattern:-

Intent

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

- Wrap a complicated subsystem with a simpler interface.

Problem

A segment of the client community needs a simplified interface to the overall functionality of a complex subsystem.

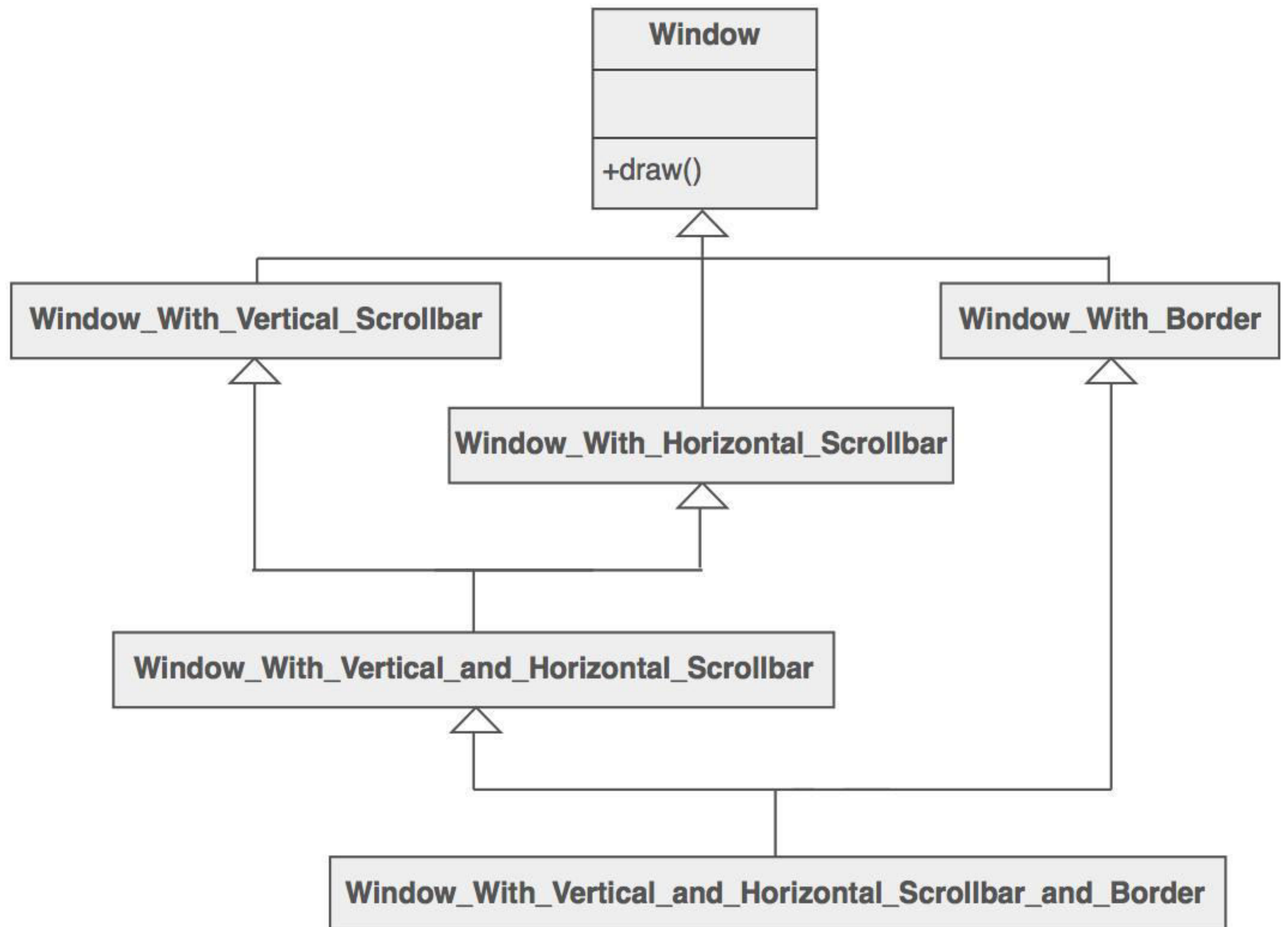
Discussion

Facade discusses encapsulating a complex subsystem within a single interface object. This reduces the learning curve necessary to successfully leverage the subsystem. It also promotes decoupling the subsystem from its potentially many clients. On the other hand, if the Facade is the only access point for the subsystem, it will limit the features and flexibility that "power users" may need.

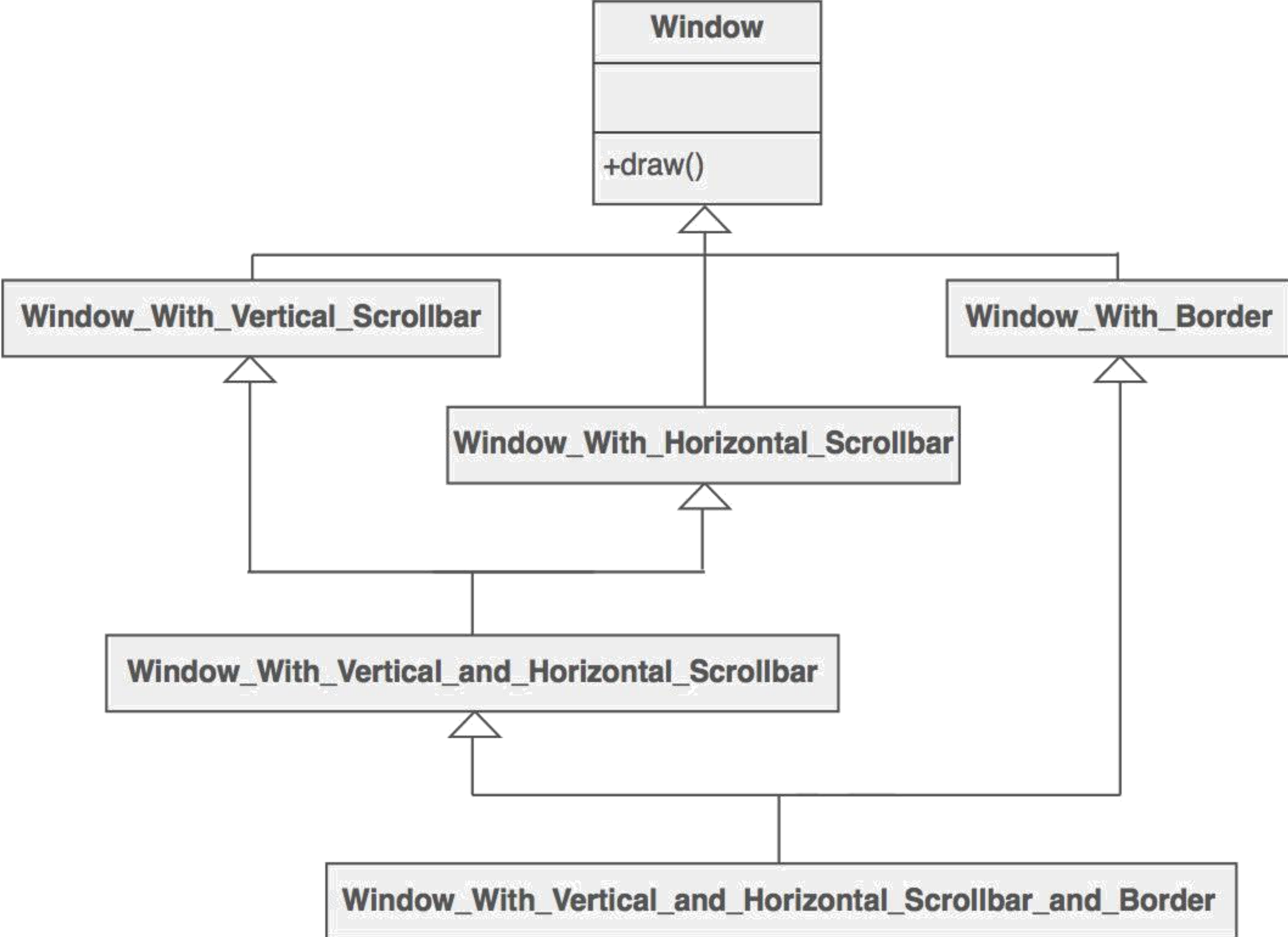
The Facade object should be a fairly simple advocate or facilitator. It should not become an all-knowing oracle or "god" object.

Structure

Facade takes a "riddle wrapped in an enigma shrouded in mystery", and interjects a wrapper that tames the amorphous and inscrutable mass of software.

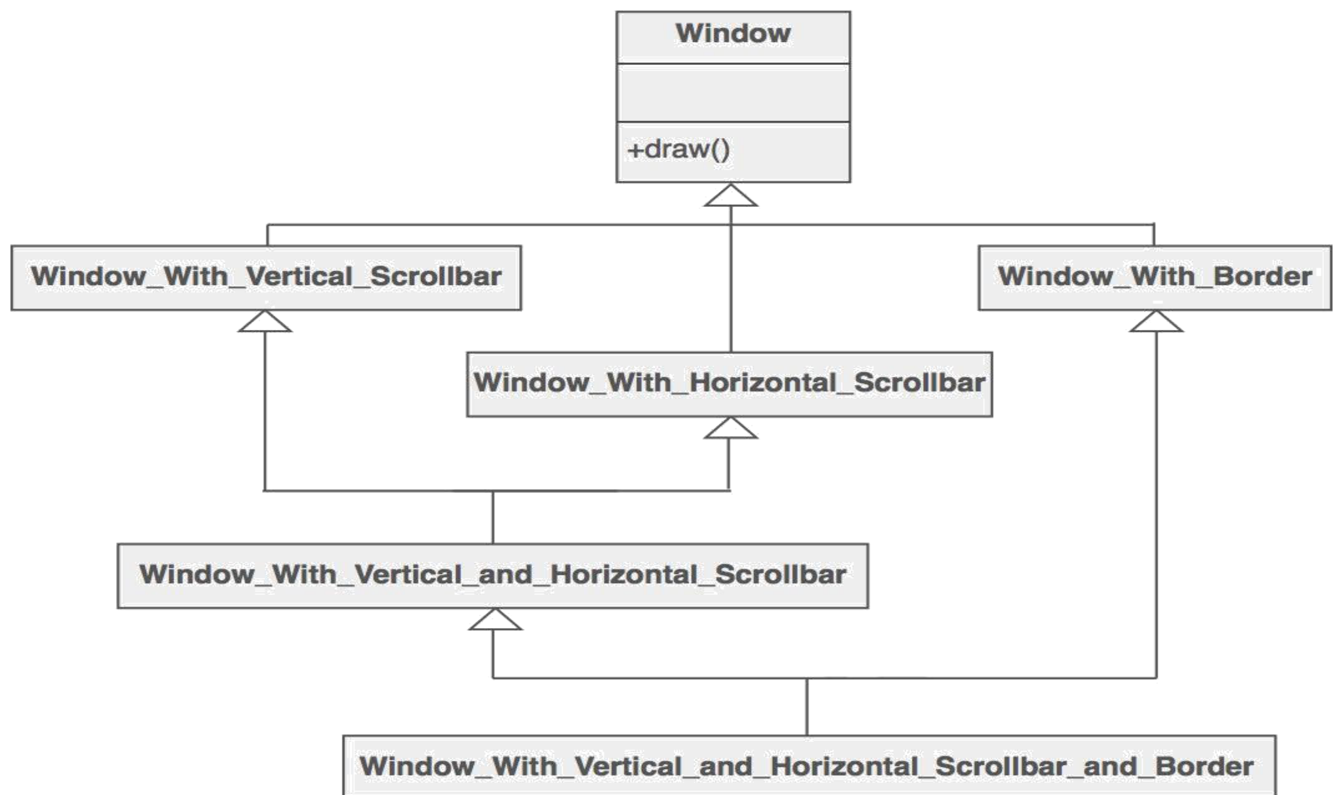


SubsystemOne and SubsystemThree do not interact with the internal components of SubsystemTwo. They use the SubsystemTwoWrapper "facade" (i.e. the higher level abstraction).


Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Border
Window_With_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

Example

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



Check list

1. Identify a simpler, unified interface for the subsystem or component.

2. Design a 'wrapper' class that encapsulates the subsystem.

3. The facade/wrapper captures the complexity and collaborations of the component, and delegates to the appropriate methods.

4. The client uses (is coupled to) the Facade only.

5. Consider whether additional Facades would add value.

Rules of thumb

- Facade defines a new interface, whereas Adapter uses an old interface. Remember that Adapter makes two existing interfaces work together as opposed to defining an entirely new one.

- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.

- Mediator is similar to Facade in that it abstracts functionality of existing classes. Mediator abstracts/centralizes arbitrary communications between colleague objects. It routinely "adds value", and it is known/referenced by the colleague objects. In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes.

- Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.

- Facade objects are often Singletons because only one Facade object is required.
- Adapter and Facade are both wrappers; but they are different kinds of wrappers. The intent of Facade is to produce a simpler interface, and the intent of Adapter is to design to an existing interface. While Facade routinely wraps multiple objects and Adapter wraps a single object; Facade could front-end a single complex object and Adapter could wrap several legacy objects.

Question: So the way to tell the difference between the Adapter pattern and the Facade pattern is that the Adapter wraps one class and the Facade may represent many classes?

Answer: No! Remember, the Adapter pattern changes the interface of one or more classes into one interface that a client is expecting. While most textbook examples show the adapter adapting one class, you may need to adapt many classes to provide the interface a client is coded to. Likewise, a Facade may provide a simplified interface to a single class with a very complex interface. The difference between the two is not in terms of how many classes they "wrap", it is in their intent.

Flyweight Design Pattern:-

Intent

- Use sharing to support large numbers of fine-grained objects efficiently.
- The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.

Problem

Designing objects down to the lowest levels of system "granularity" provides optimal flexibility, but can be unacceptably expensive in terms of performance and memory usage.

Discussion

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

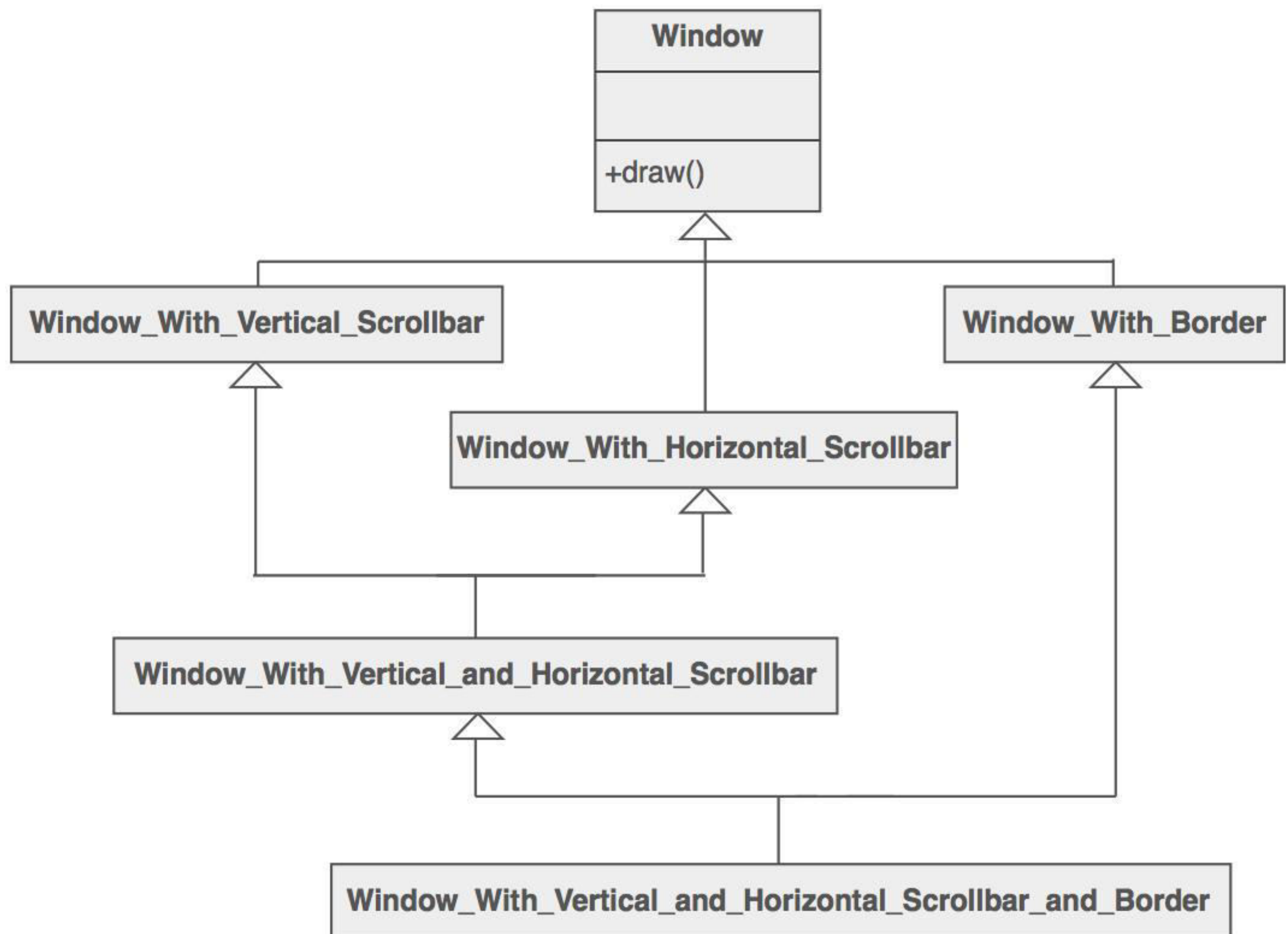
An illustration of this approach would be Motif widgets that have been re-engineered as light-weight gadgets. Whereas widgets are "intelligent" enough to stand on their own; gadgets exist in a dependent relationship with their parent layout manager widget. Each

layout manager provides context-dependent event handling, real estate management, and resource services to its flyweight gadgets, and each gadget is only responsible for context-independent state and behavior.

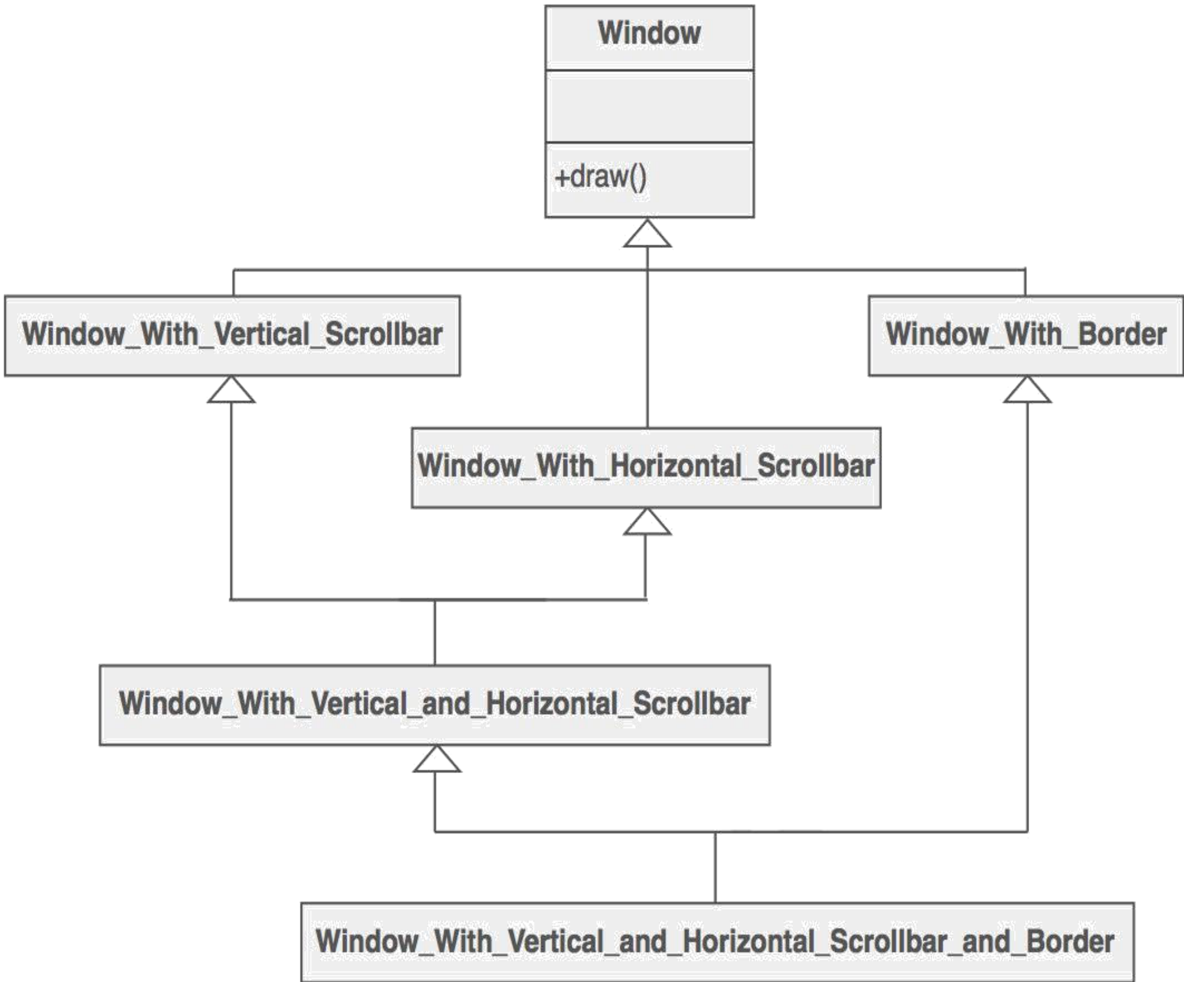
Structure

Flyweights are stored in a Factory's repository. The client restrains herself from creating Flyweights directly, and requests them from the Factory. Each Flyweight cannot stand on its

own. Any attributes that would make sharing impossible must be supplied by the client whenever a request is made of the Flyweight. If the context lends itself to "economy of scale" (i.e. the client can easily compute or look-up the necessary attributes), then the Flyweight pattern offers appropriate leverage.

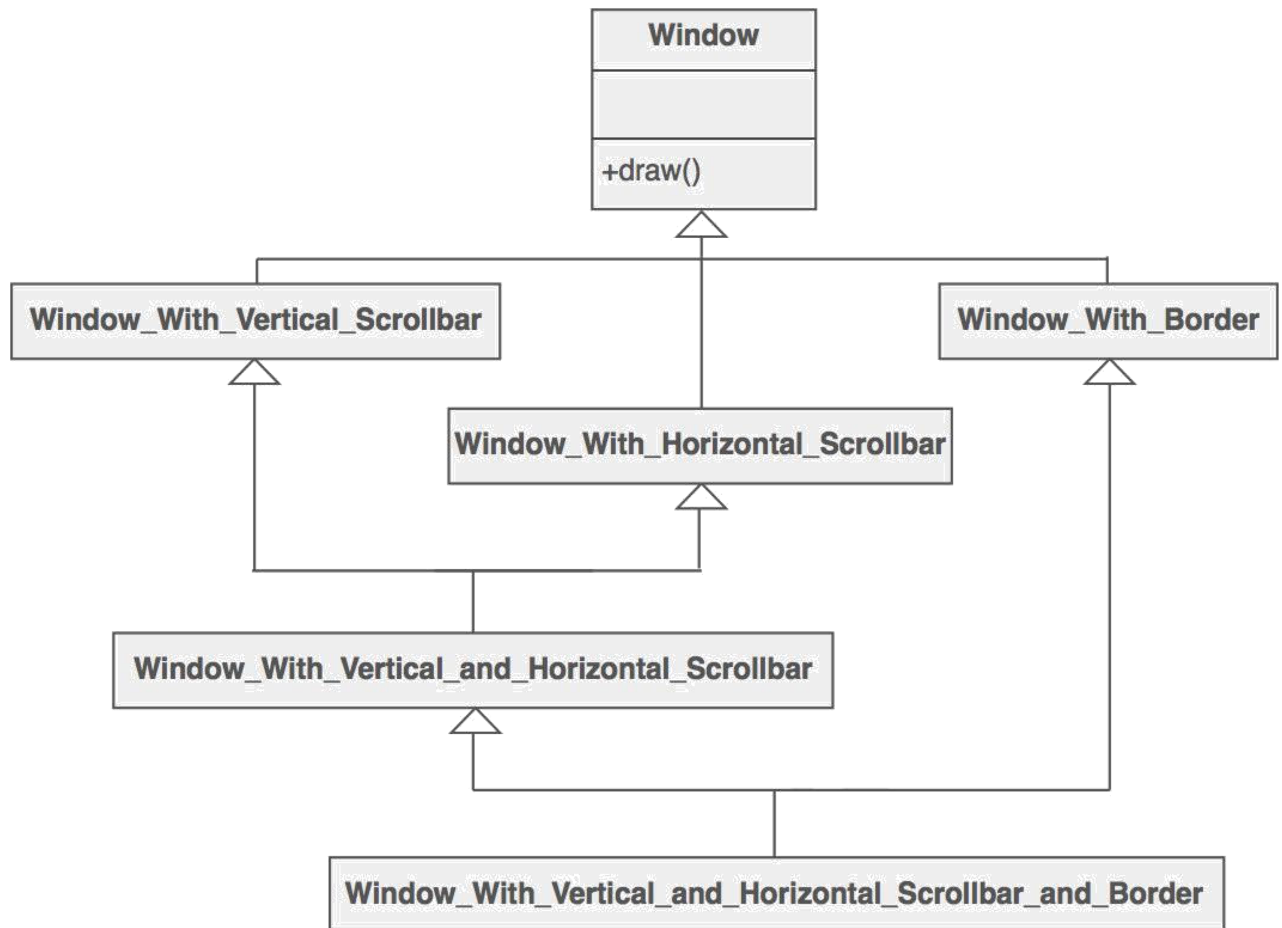


The Ant, Locust, and Cockroach classes can be "light-weight" because their instance-specific state has been de-encapsulated, or externalized, and must be supplied by the client.


Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Border
Window_With_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

Example

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



Check list

1. Ensure that object overhead is an issue needing attention, and, the client of the class is able and willing to absorb responsibility realignment.

2. Divide the target class's state into: shareable (intrinsic) state, and non-shareable (extrinsic) state.

3. Remove the non-shareable state from the class attributes, and add it the calling argument list of affected methods.

4. Create a Factory that can cache and reuse existing class instances.

5. The client must use the Factory instead of the new operator to request objects.

6. The client (or a third party) must look-up or compute the non-shareable state, and supply that state to class methods.

Rules of thumb

- Whereas Flyweight shows how to make lots of little objects, Facade shows how to make a single object represent an entire subsystem.

- Flyweight is often combined with Composite to implement shared leaf nodes.

- Terminal symbols within Interpreter's abstract syntax tree can be shared with Flyweight.

- Flyweight explains when and how State objects can be shared.

Proxy Design Pattern:-

Intent

- Provide a surrogate or placeholder for another object to control access to it.
- Use an extra level of indirection to support distributed, controlled, or intelligent access.
- Add a wrapper and delegation to protect the real component from undue complexity.

Problem

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Discussion

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

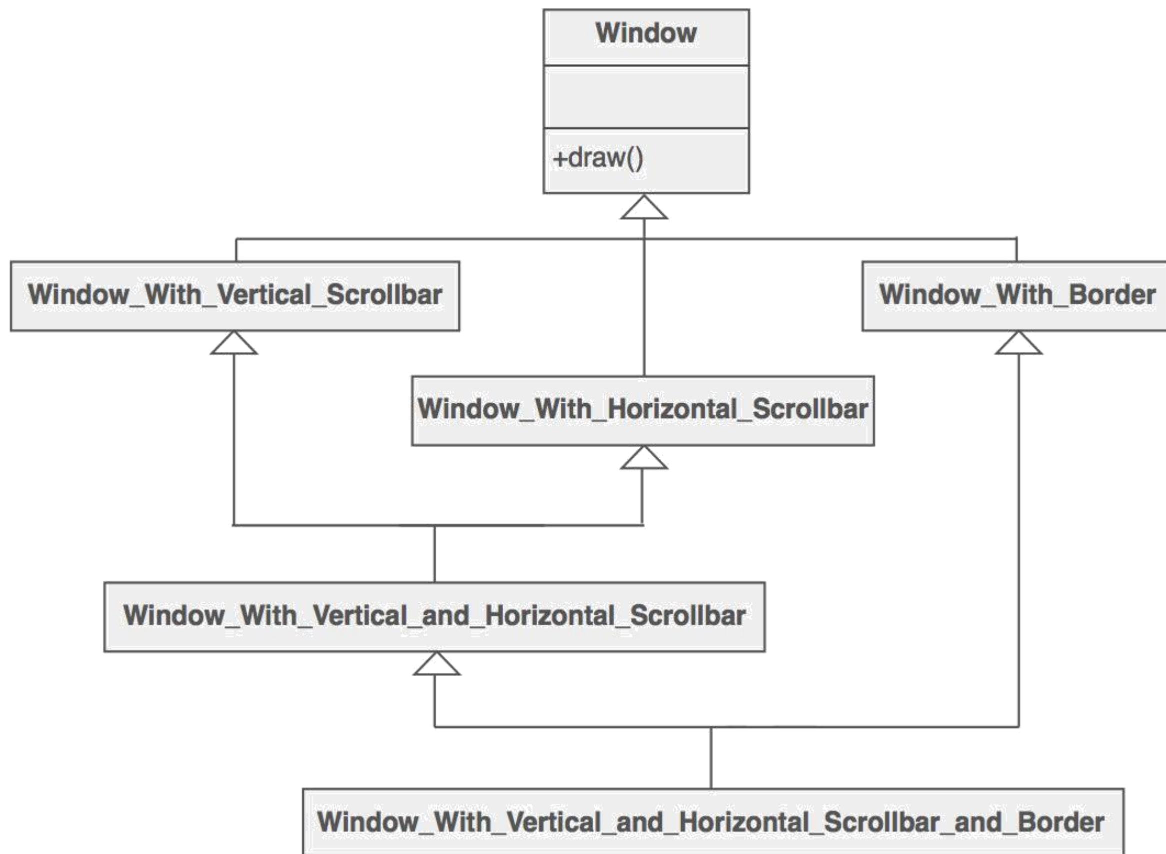
There are four common situations in which the Proxy pattern is applicable.

1. A virtual proxy is a placeholder for "expensive to create" objects. The real object is only created when a client first requests/accesses the object.
2. A remote proxy provides a local representative for an object that resides in a different address space. This is what the "stub" code in RPC and CORBA provides.
3. A protective proxy controls access to a sensitive master object. The "surrogate" object checks that the caller has the access permissions required prior to forwarding the request.
4. A smart proxy interposes 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 (aka smart pointer),

- Loading a persistent object into memory when it's first referenced,
- Checking that the real object is locked before it is accessed to ensure that no other object can change it.

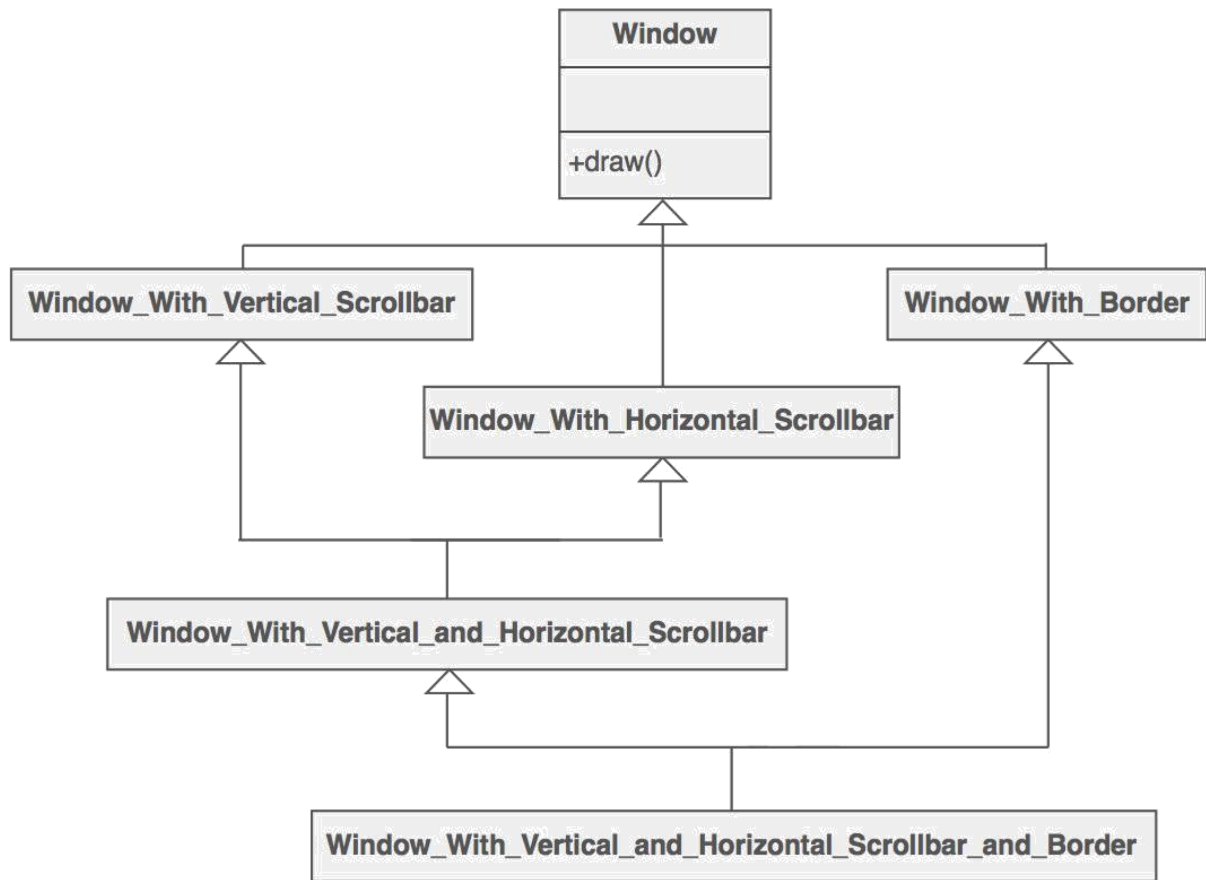
Structure

By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client.



Example

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



Check list

1. Identify the leverage or "aspect" that is best implemented as a wrapper or surrogate.

2. Define an interface that will make the proxy and the original component interchangeable.

3. Consider defining a Factory that can encapsulate the decision of whether a proxy or original object is desirable.

4. The wrapper class holds a pointer to the real class and implements the interface.

5. The pointer may be initialized at construction, or on first use.

6. Each wrapper method contributes its leverage, and delegates to the wrappee object.

Rules of thumb

- Adapter provides a different interface to its subject. Proxy provides the same interface. Decorator provides an enhanced interface.

- Decorator and Proxy have different purposes but similar structures. Both describe how to provide a level of indirection to another object, and the implementations keep a reference to the object to which they forward requests.

UNIT-IV

Behavioral Patterns

Behavioural Patterns Part-I : Chain of Responsibility, Command, Interpreter, Iterator.

Behavioral Patterns (1)

• Deal with the way objects interact and distribute responsibility.

• 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.

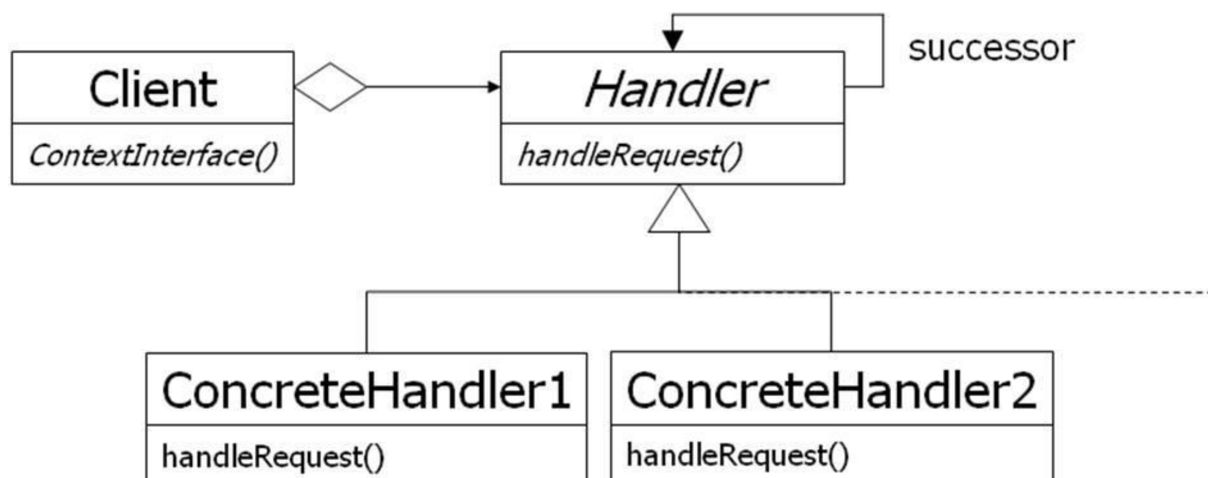
objects an dpass the request along the chain until an object handles it.

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

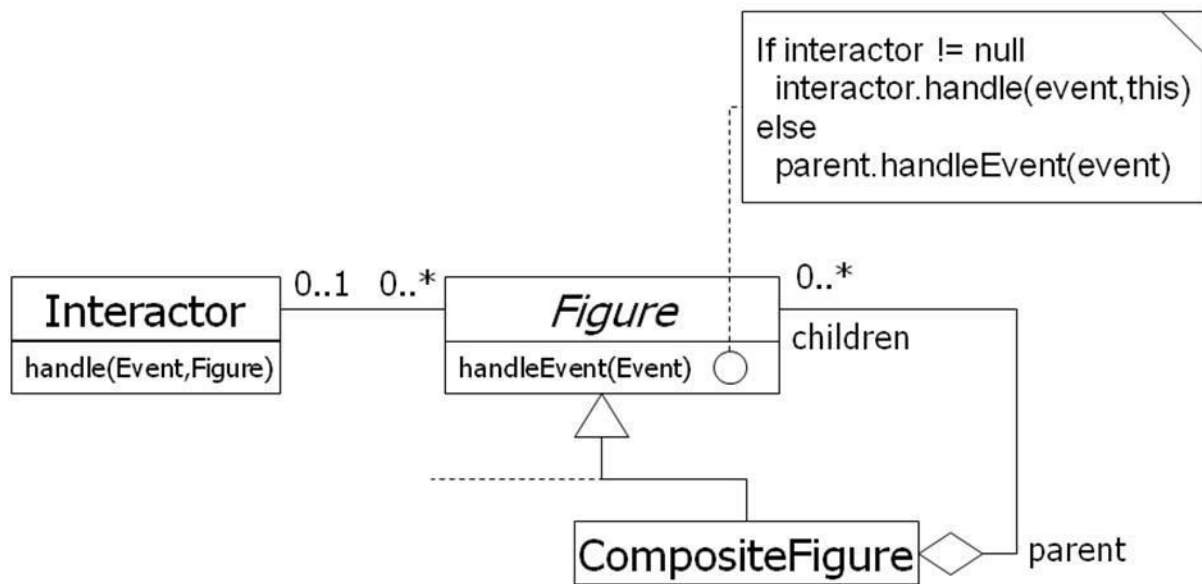
• Interpreter: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Chain of Responsibility:

- Decouple sender of a request from receiver.
- Give more than one object a chance to handle.
- Flexibility in assigning responsibility.
- Often applied with Composite.



Chain of Responsibility (2)



Command:Encapsulating Control Flow :

Name: Command design pattern

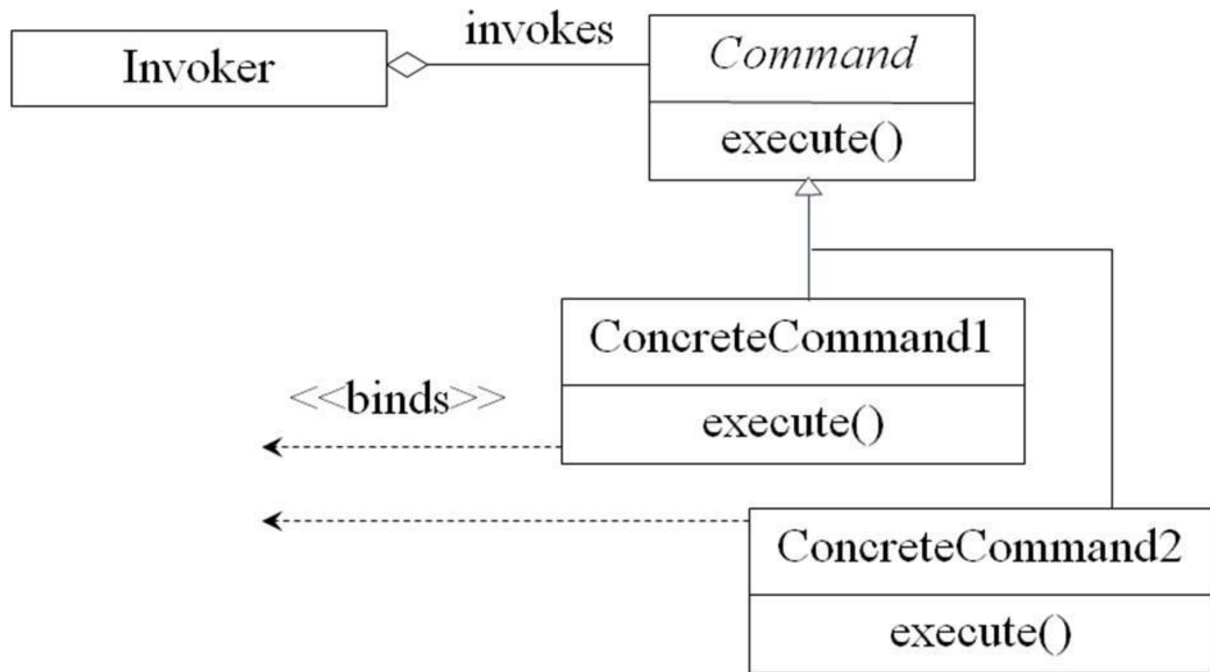
Problem description:

Encapsulates requests so that they can be executed, undone, or queued independently of the request.

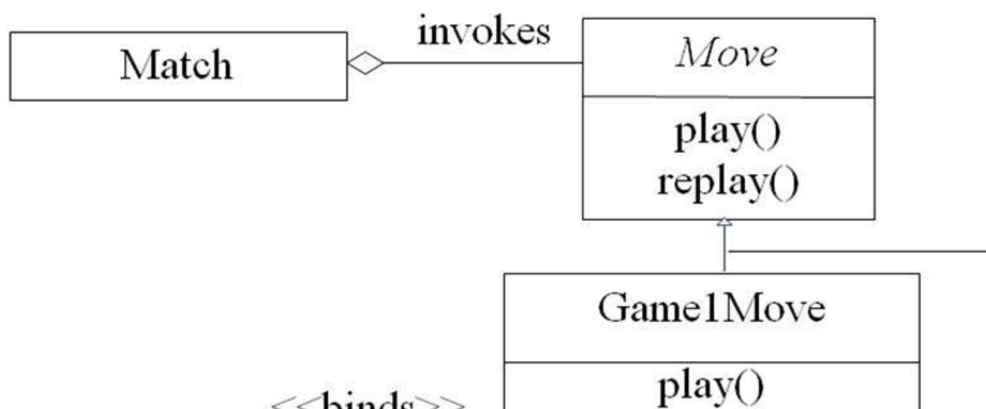
Solution:

A Command abstract class declares the interface supported by all ConcreteCommands. ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates ConcreteCommands and binds them to specific Receivers. The Invoker actually executes a command.

Command: Class Diagram



Command: Class Diagram for Match



Command: Consequences

Consequences:

The object of the command (Receiver) and the algorithm of the command.

(ConcreteCommand) are decoupled.

Invoker is shielded from specific commands.

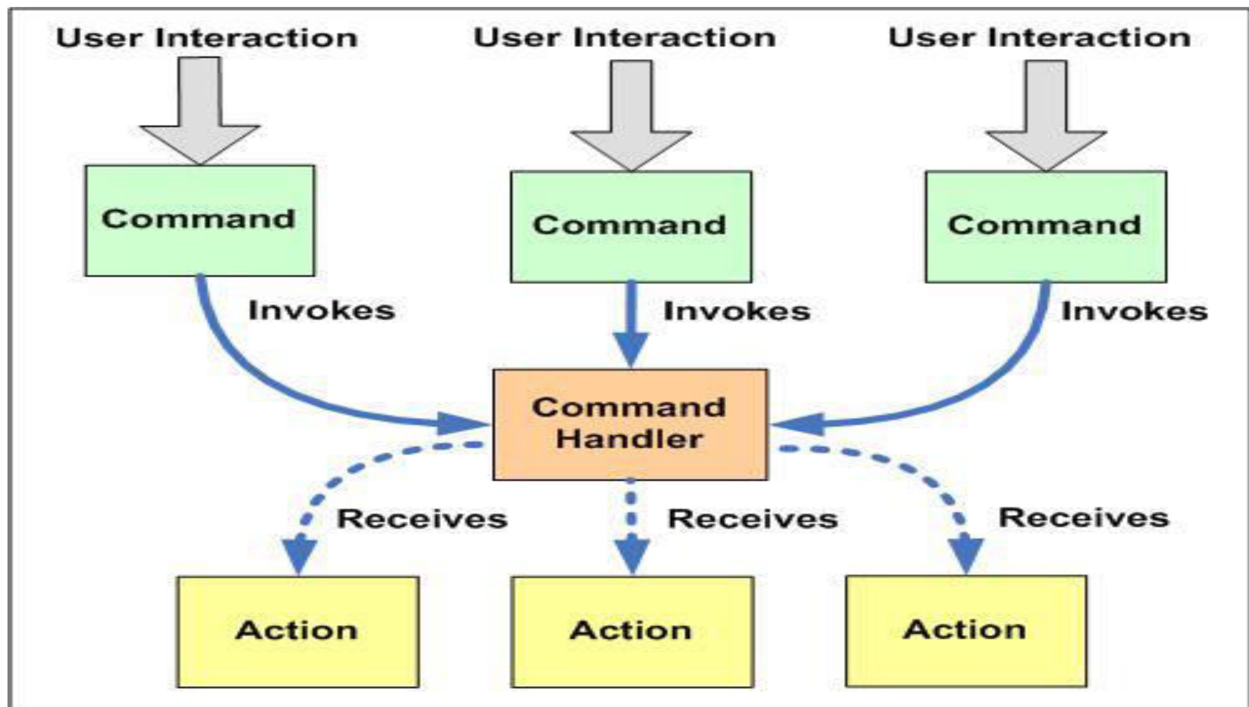
ConcreteCommands are objects. They can be created and stored.

New ConcreteCommands can be added without changing existing code.

Command:

- You have commands that need to be
  - executed,
  - undone, or
  - queued
- Command design pattern separates
  - Receiver from Invoker from Commands
- All commands derive from Command and implement do(), undo(), and redo().

Command Design Pattern :



- Separates command invoker and receiver.

Pattern: Interpreter:

- Intent: Given a language, interpret sentences.
- Participants: Expressions, Context, Client.
- Implementation: A class for each expression type

  An Interpret method on each class

A class and object to store the global state (context)

- No support for the parsing process

  (Assumes strings have been parsed into exp trees)

Pattern: Interpreter with Macros:

- Example: Definite Clause Grammars.
- A language for writing parsers/interpreters.
- Macros make it look like (almost) standard BNF.
  Command(move(D)) -> “go”, Direction(D).
- Built-in to Prolog; easy to implement in Dylan, Lisp.
- Does parsing as well as interpretation.

- Builds tree structure only as needed.

  (Or, can automatically build complete trees)

- May or may not use expression classes.

Method Combination:

- Build a method from components in different classes

- Primary methods: the “normal” methods; choose the most specific one

- Before/After methods: guaranteed to run;
  No possibility of forgetting to call super

  Can be used to implement *Active Value* pattern

- Around methods: wrap around everything;
  Used to add tracing information, etc.

- Is added complexity worth it?

  Common Lisp: Yes; Most languages: No

Iterator pattern :

- **iterator**: an object that provides a standard way to examine all elements of any collection.

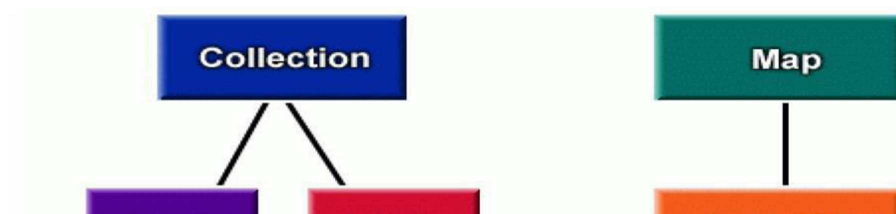
- uniform interface for traversing many different data structures without exposing their implementations.

- supports concurrent iteration and element removal.

- removes need to know about internal structure of collection or different methods to access data from different collections.

Pattern: Iterator

objects that traverse collections



Iterator interfaces in Java:

```
public interface java.util.Iterator {

  public boolean hasNext();

  public Object next();

  public void remove();

}

public interface java.util.Collection {

  ...  // List, Set extend Collection

  public Iterator iterator();

}

public interface java.util.Map {

  ...

  public Set keySet(); // keys,values are Collections public
  Collection values(); // (can call iterator() on them)

}
```

Iterators in Java:

- all Java collections have a method iterator that returns an iterator for the elements of the collection.

- can be used to look through the elements of any kind of collection (an alternative to for loop).

```
List list = new ArrayList();

... add some elements ...
```

```
for (Iterator itr = list.iterator(); itr.hasNext()) {
  BankAccount ba = (BankAccount)itr.next();
  System.out.println(ba);
}
```

Adding your own Iterators :

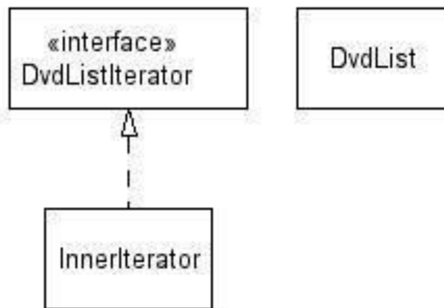
- when implementing your own collections, it can be very convenient to use Iterators

  – **discouraged (has nonstandard interface):**

```
public class PlayerList {

  public int getNumPlayers() { ... }
  public boolean empty() { ... }
  public Player getPlayer(int n) { ...
  }
}
```

  – **preferred:**

```
public class PlayerList {

  public Iterator iterator() { ... }

  public int size() { ... }

  public boolean isEmpty() { ... }

}
```



Command:Encapsulating Control Flow:

Name: Command design pattern

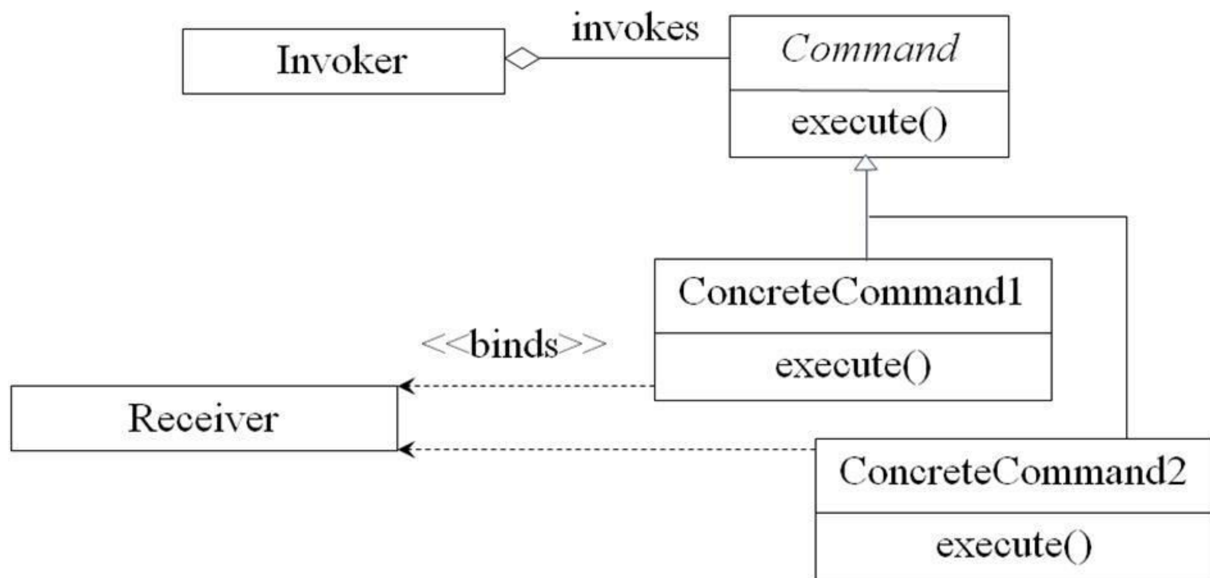
Problem description:

Encapsulates requests so that they can be executed, undone, or queued independently of the request.

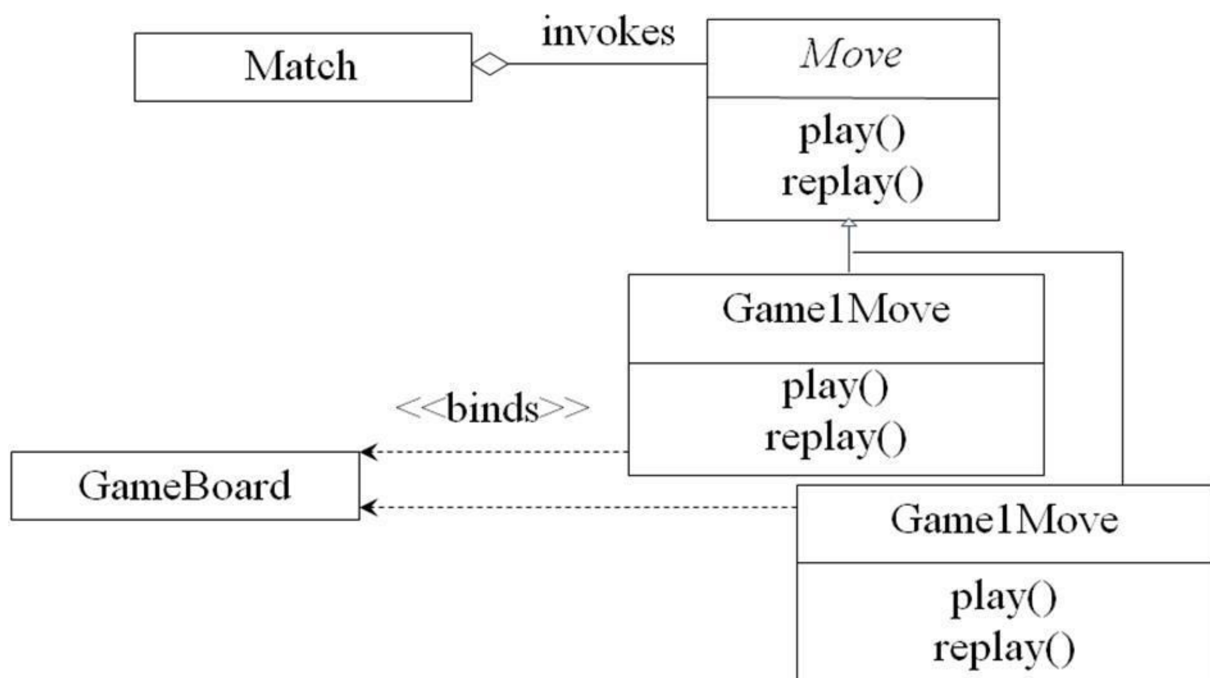
Solution:

A Command abstract class declares the interface supported by all ConcreteCommands. ConcreteCommands encapsulate a service to be applied to a Receiver. The Client creates ConcreteCommands and binds them to specific Receivers. The Invoker actually executes a command.

Command: Class Diagram



Command: Class Diagram for Match



Command: Consequences

Consequences:

The object of the command (Receiver) and the algorithm of the command (ConcreteCommand) are decoupled.

Invoker is shielded from specific commands.

ConcreteCommands are objects. They can be created and stored.

New ConcreteCommands can be added without changing existing code.

- **Intent**: Given a language, interpret sentences
- **Participants**: Expressions, Context, Client
- **Implementation**: A class for each expression type An Interpret method on each class

  A class and object to store the global state (context)
- No support for the parsing process

  (Assumes strings have been parsed into exp trees)

Pattern: Interpreter with Macros

- **Example**: Definite Clause Grammars
- A language for writing parsers/interpreters
- Macros make it look like (almost) standard BNF Command(move(D)) -> "go", Direction(D).
- Built-in to Prolog; easy to implement in Dylan, Lisp
- Does parsing as well as interpretation.
- Builds tree structure only as needed.

  (Or, can automatically build complete trees)

- May or may not use expression classes.

Method Combination:

- Build a method from components in different classes

- Primary methods: the "normal" methods; choose the most specific one

- Before/After methods: guaranteed to run;
  No possibility of forgetting to call super

  Can be used to implement *Active Value* pattern

- Around methods: wrap around everything;
  Used to add tracing information, etc.

- Is added complexity worth it?

  Common Lisp: Yes; Most languages: No

Behavioural Patterns Part-II

Part-II : Mediator, Memento, Observer

Behavioral Patterns (1):

• Deal with the way objects interact and distribute responsibility

• 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 an dpass the request along the chain until an object handles it.

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

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

Behavioral Patterns (2):

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

• 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 lets you vary their interaction independently.

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

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

Behavioral Patterns (3)

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

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

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

- 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.

The Mediator Pattern:

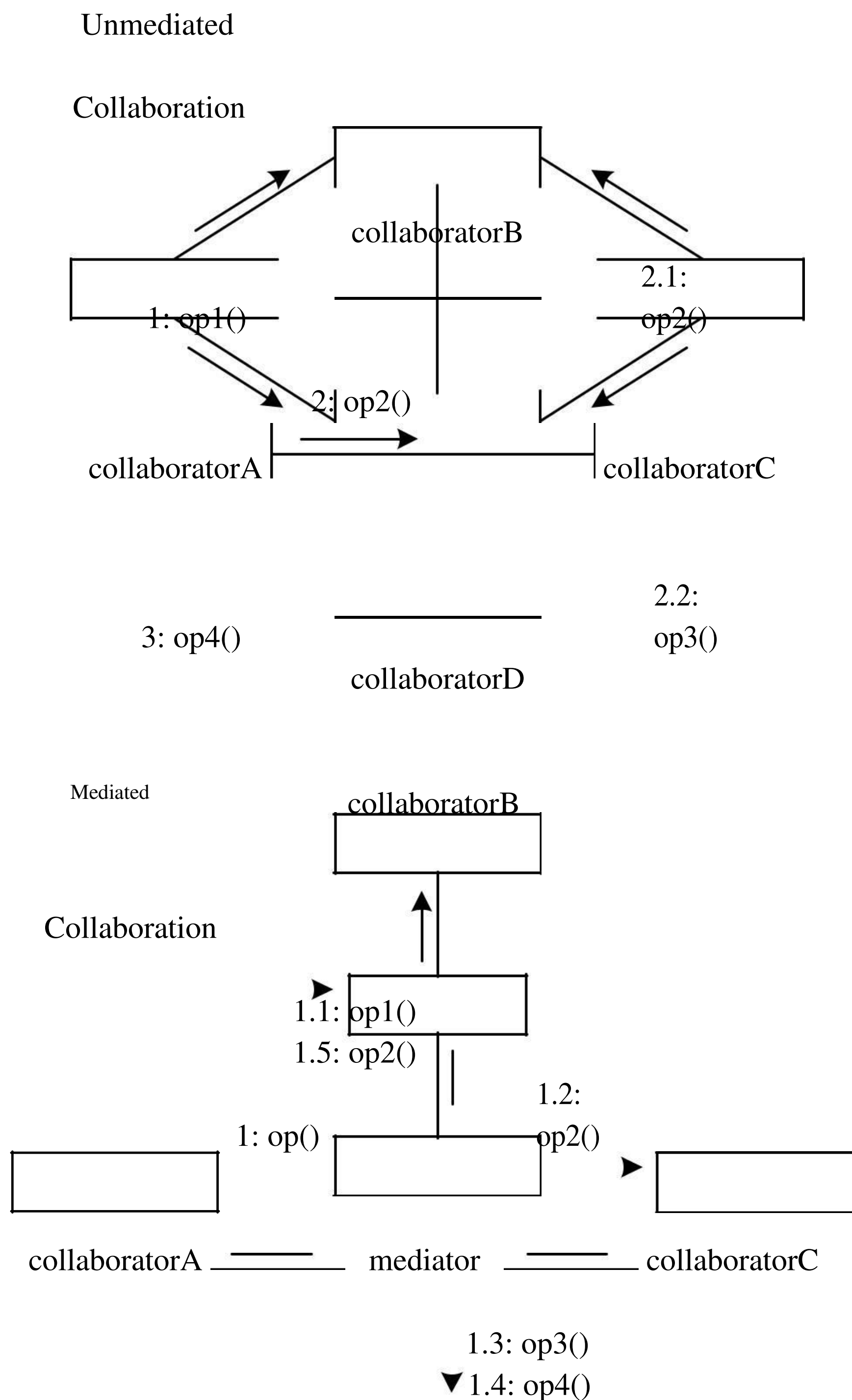
- The Mediator pattern reduces coupling and simplifies code when several objects must negotiate a complex interaction.

- Classes interact only with a mediator class rather than with each other.

- Classes are coupled only to the mediator where interaction control code resides.

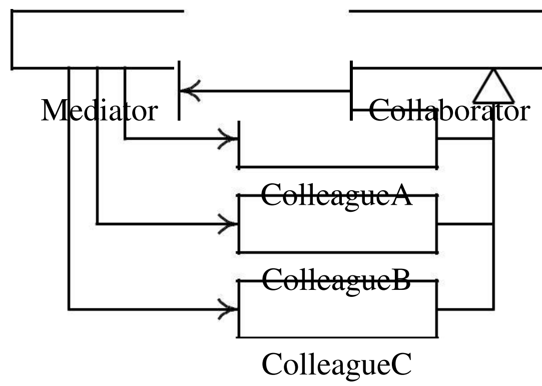
- Mediator is like a multi-way Façade pattern.

Analogy: a meeting scheduler.

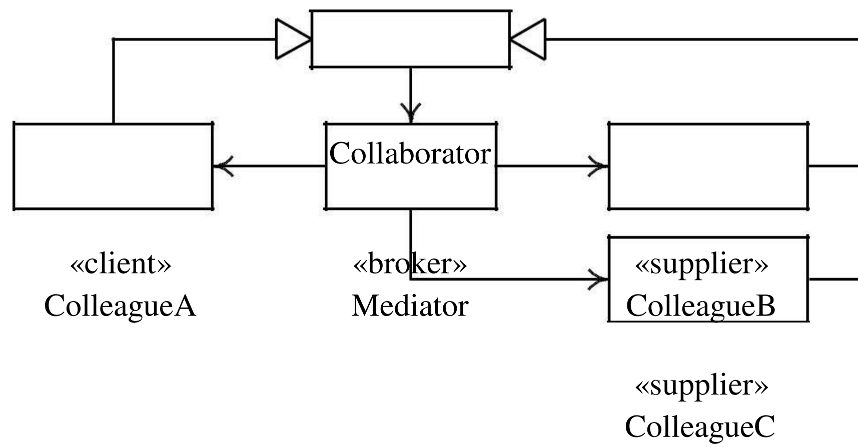


collaboratorD

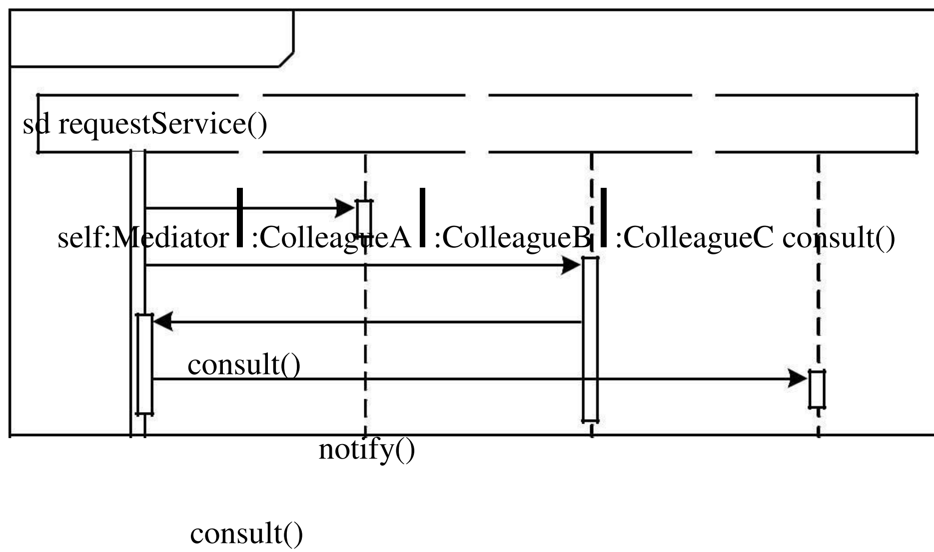
Mediator Pattern Structure:



Mediator as a Broker:



Mediator Behavior:



When to Use a Mediator:

- Use the Mediator pattern when a complex interaction between collaborators must be encapsulated to

  - Decouple collaborators,
  - Centralize control of an interaction, and
  - Simplify the collaborators.
- Using a mediator may compromise performance.

Mediators, Façades, and Control Styles:

- The Façade and Mediator patterns provide means to make control more centralized.
- The Façade and Mediator patterns should be used to move from a dispersed to a delegated style, but not from a delegated to a centralized style.

Summary :

- Broker patterns use a Broker class to facilitate the interaction between a Client and a Supplier.
- The Façade pattern uses a broker (the façade) to provide a simplified interface to a complex sub-system.
- The Mediator pattern uses a broker to encapsulate and control a complex interaction among several suppliers.

Memento Pattern

Intent:

- Capture and externalize an object's state without violating encapsulation.
- Restore the object's state at some later time.
  - Useful when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors.
  - Entrusts other objects with the information it needs to revert to a previous state without exposing its internal structure and representations.

Forces:

- Application needs to capture states at certain times or at user discretion. May be used for:
  - Undue / redo
  - Log errors or events

- Backtracking

- Need to preserve encapsulation
  - Don’t share knowledge of state with other objects
- Object owning state may not know when to take state snapshot.

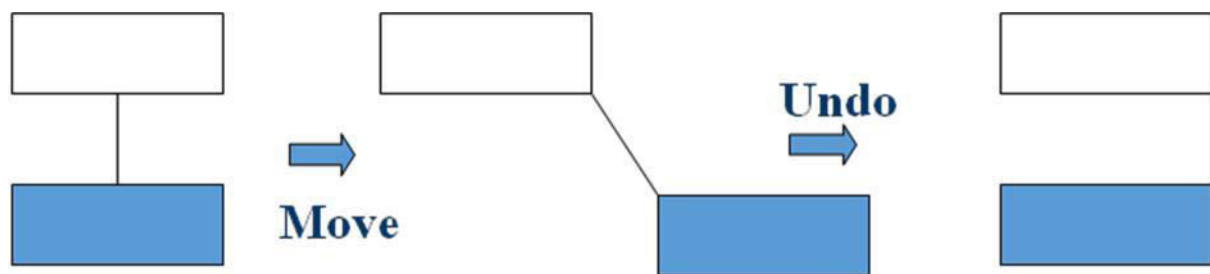
Motivation:

- Many technical processes involve the exploration of some complex data structure.
- Often we need to backtrack when a particular path proves unproductive.

Examples are graph algorithms, searching knowledge bases, and text navigation.

Memento stores a snapshot of another object's internal state, exposure of which would violate encapsulation and compromise the application's reliability and extensibility.

A graphical editor may encapsulate the connectivity relationships between objects in a class, whose public interface might be insufficient to allow precise reversal of a move operation.



Memento pattern solves this problem as follows:

- The editor requests a memento from the object before executing *move* operation.
- Originator creates and returns a memento.
- During *undo* operation, the editor gives the memento back to the originator.

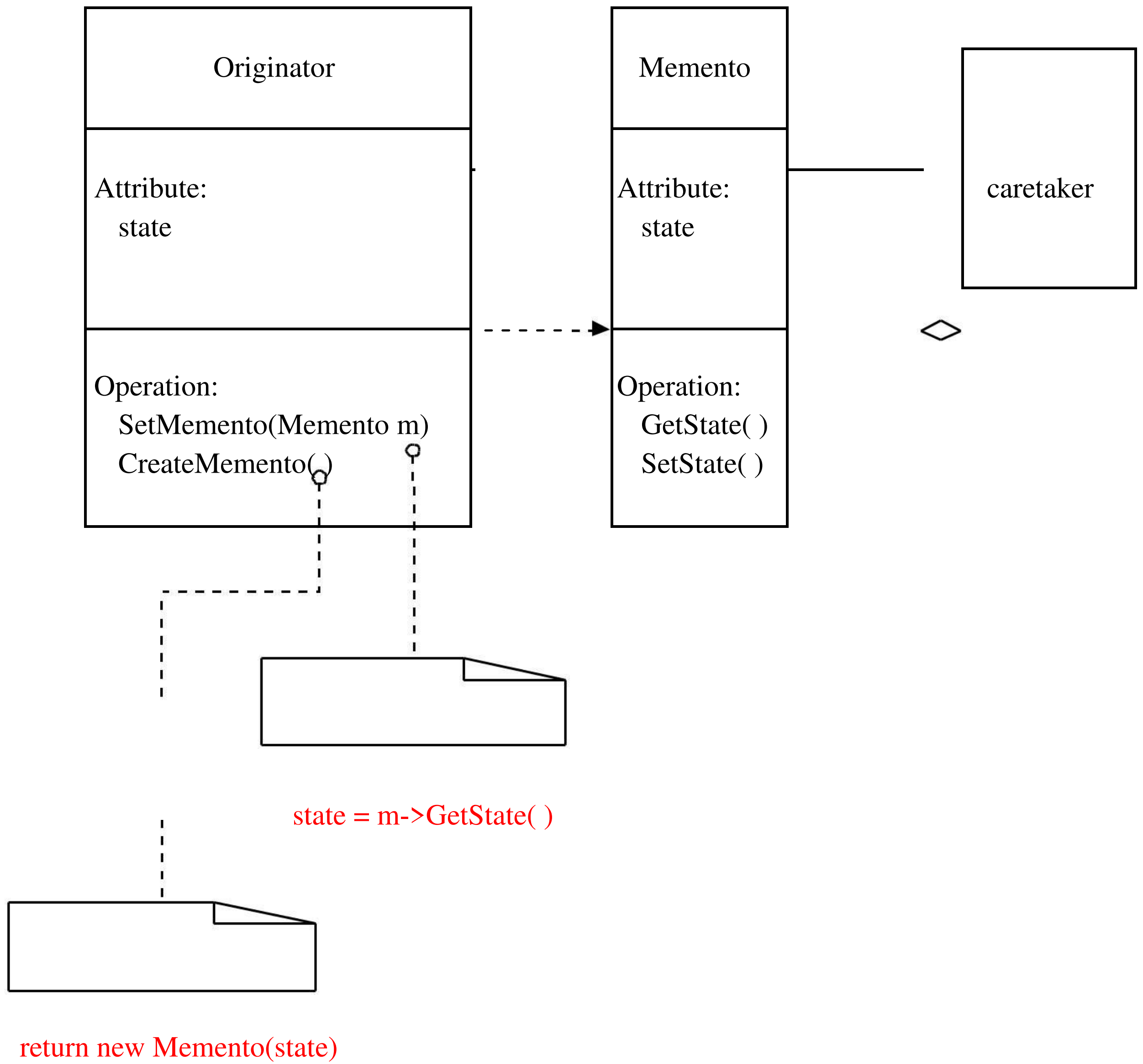
- Based on the information in the memento, the originator restores itself to its previous state.

Applicability:

- Use the Memento pattern when:

    – A snapshot of an object's state must be saved so that it can be restored later, and direct access to the state would expose implementation details and break encapsulation.

Structure:



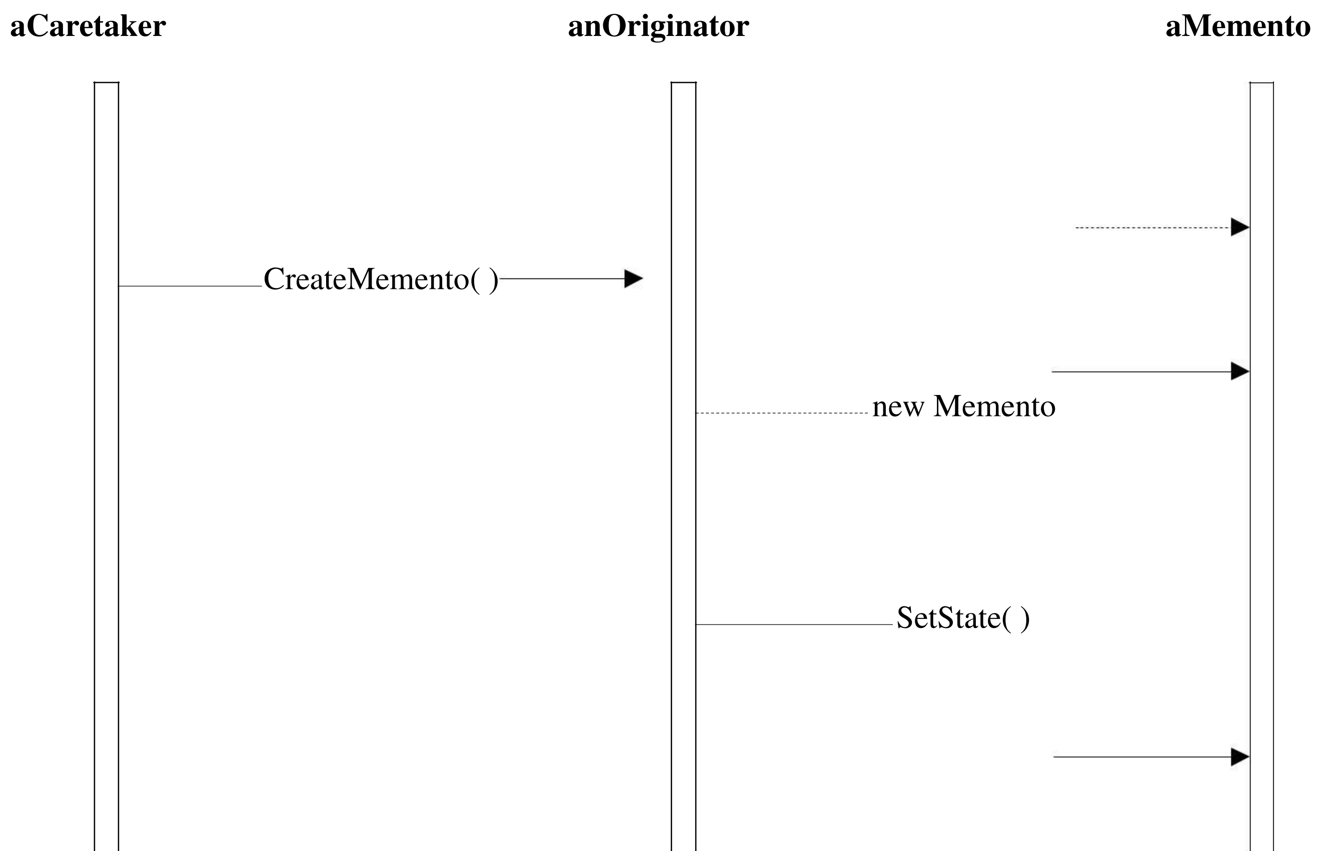
Participants:

- Memento
  - Stores internal state of the Originator object. Originator decides how much.
  - Protects against access by objects other than the originator.
  - Mementos have two interfaces:
    - Caretaker sees a narrow interface.
    - Originator sees a wide interface.
- Originator
  - Creates a memento containing a snapshot of its current internal state.
  - Uses the memento to restore its internal state.

Caretaker:

- Is responsible for the memento's safekeeping.
- Never operates on or examines the contents of a memento.

Event Trace:



SetMemento(aMemento)

GetState( )

Collaborations:

- A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator.

- Mementos are passive. Only the originator that created a memento will assign or retrieve its state.

Consequences:

Memento has several consequences:

- – Memento avoids exposing information that only an originator should manage, but for simplicity should be stored outside the originator.
- – Having clients manage the state they ask for simplifies the originator.

- Using mementos may be expensive, due to copying of large amounts of state or frequent creation of mementos.
- A caretaker is responsible for deleting the mementos it cares for.

A caretaker may incur large storage costs when it stores mementos.

Implementation:

When mementos get created and passed back to their originator in a predictable sequence, then Memento can save just incremental changes to originator's state.

Known Uses:

Memento is a 2000 film about Leonard Shelby and his quest to revenge the brutal murder of his wife. Though Leonard is hampered with short-term memory loss, he uses notes and tatoos to compile the information into a suspect.

Known Use of Pattern

- Dylan language uses memento to provide iterators for its collection facility.
    - Dylan is a dynamic object oriented language using the functional style.
    - Development started by Apple, but subsequently moved to open source.

Related Patterns

- Command

  Commands can use mementos to maintain state for undo mechanisms.

- Iterator

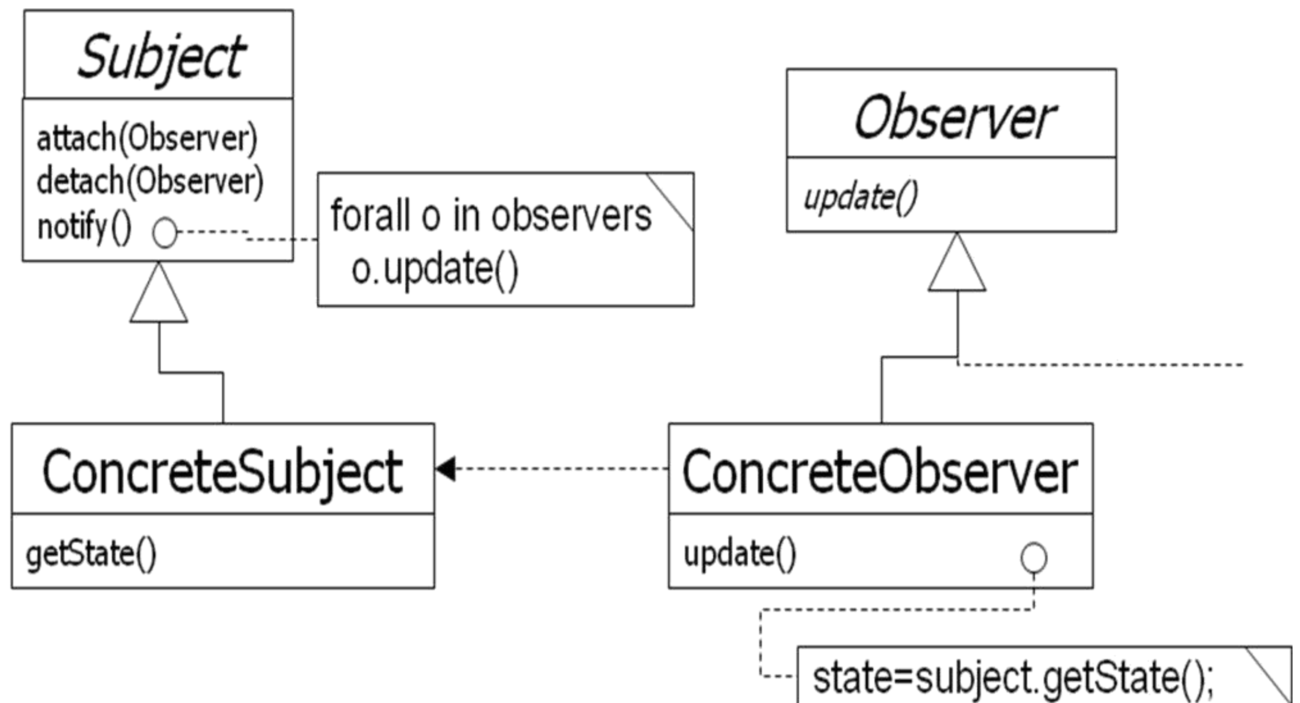
  Mementos can be used for iteration.

Observer Pattern

- Define a one-to-many dependency, all the dependents are notified and updated automatically
- The interaction is known as **publish-subscribe** or **subscribe-notify**
- Avoiding observer-specific update protocol: **pull model** vs. **push model**
- Other consequences and open issues
- **Intent:**
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- **Key forces:**
  - There may be many observers
  - Each observer may react differently to the same notification
  - The subject should be as decoupled as possible from the observers to allow observers to change independently of the subject

Observer

- Many-to-one dependency between objects
- Use when there are two or more views on the same "data"
- aka "Publish and subscribe" mechanism
- Choice of "push" or "pull" notification styles



Observer: Encapsulating Control Flow

Name: Observer design pattern

Problem description:

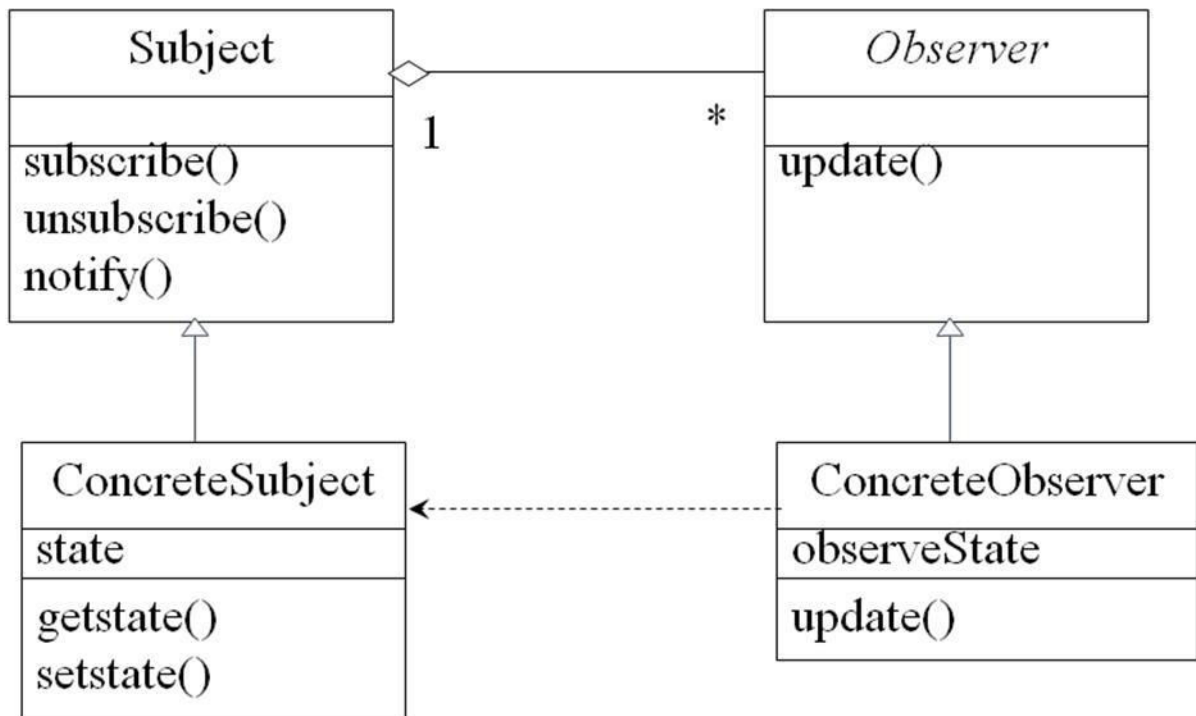
Maintains consistency across state of one Subject and many Observers.

Solution:

A Subject has a primary function to maintain some state (e.g., a data structure). One or more Observers use this state, which introduces redundancy between the states of Subject and Observer.

Observer invokes the subscribe() method to synchronize the state. Whenever the state changes, Subject invokes its notify() method to iteratively invoke each Observer.update() method.

Observer: Class Diagram



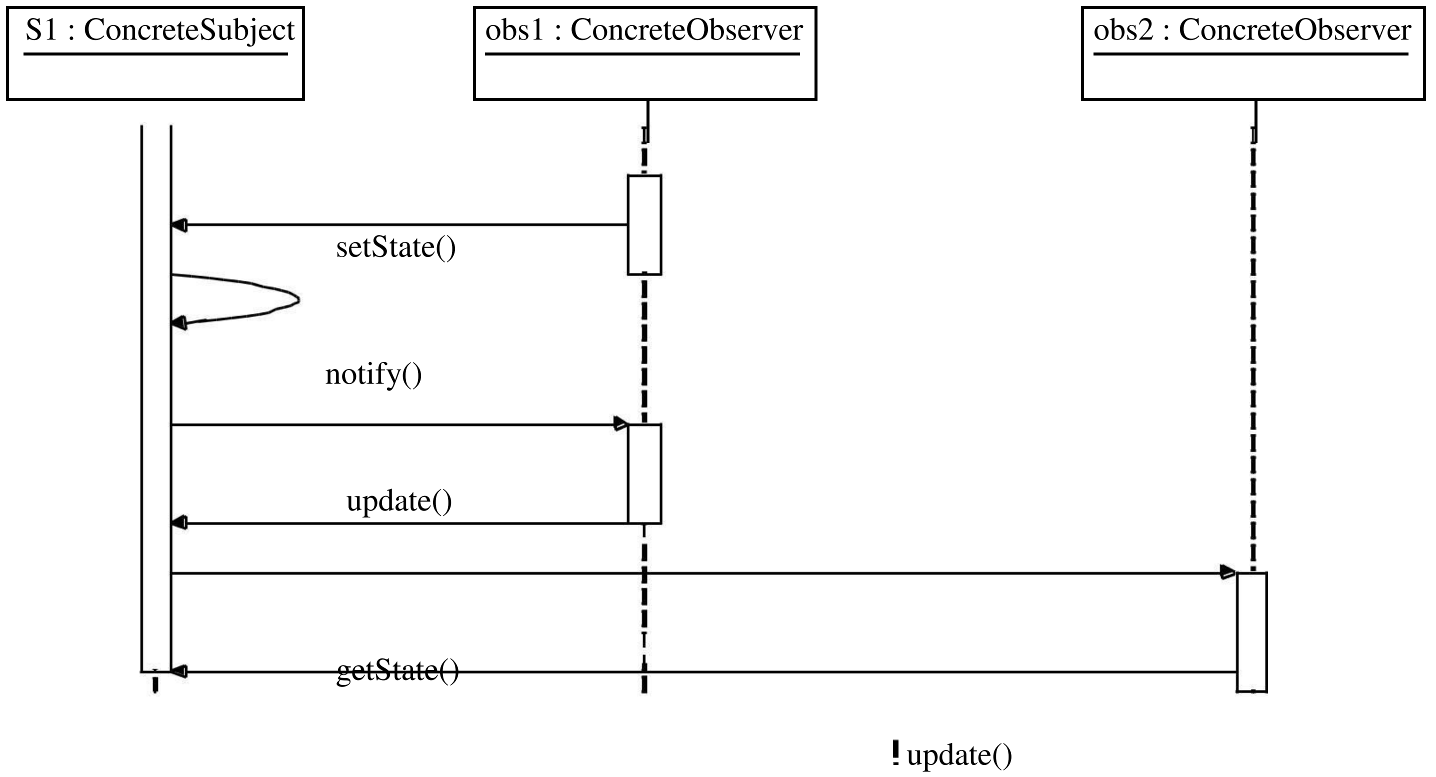
Observer: Consequences

Consequences:

Decouples Subject, which maintains state, from Observers, who make use of the state.

Can result in many spurious broadcasts when the state of Subject changes.

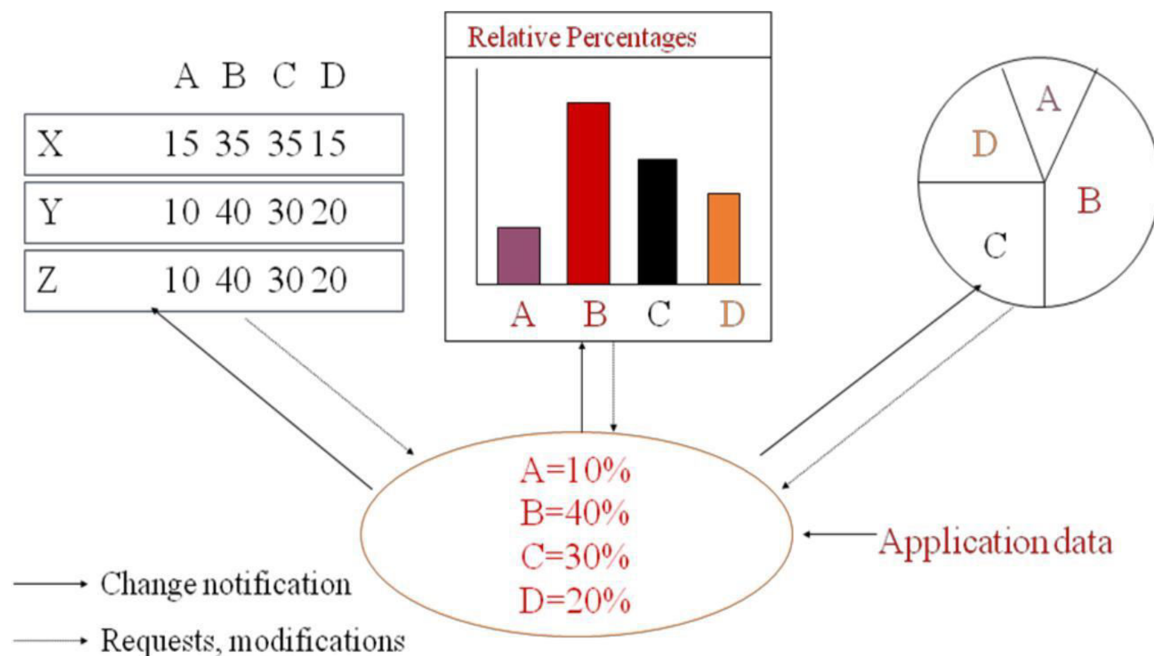
Collaborations in Observer Pattern

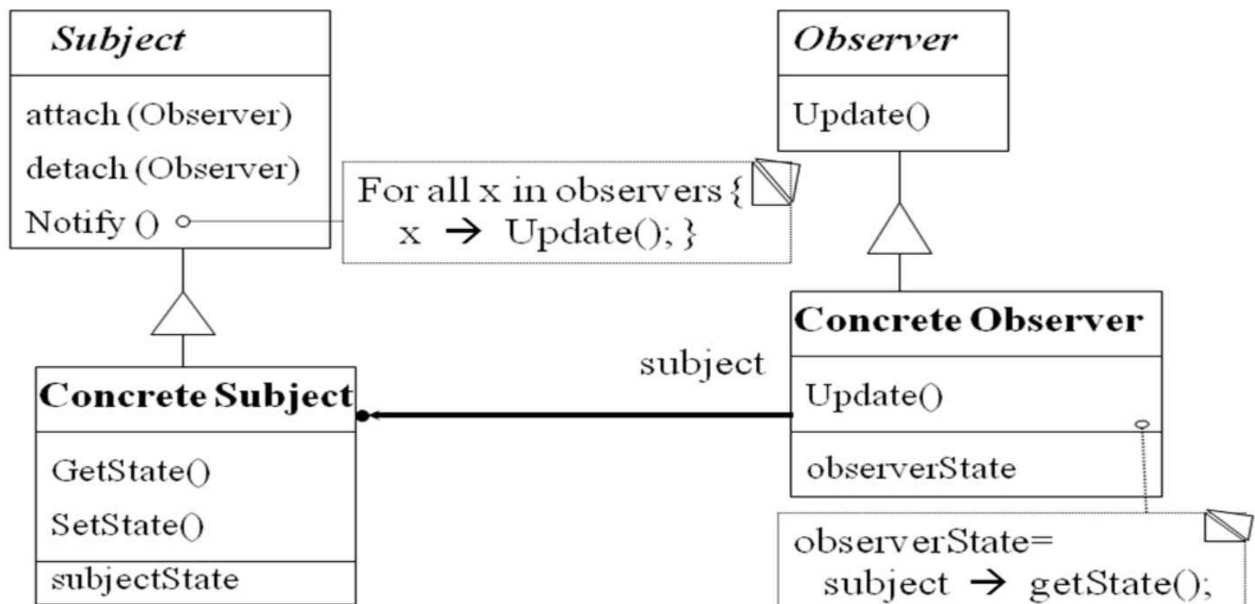


getState()

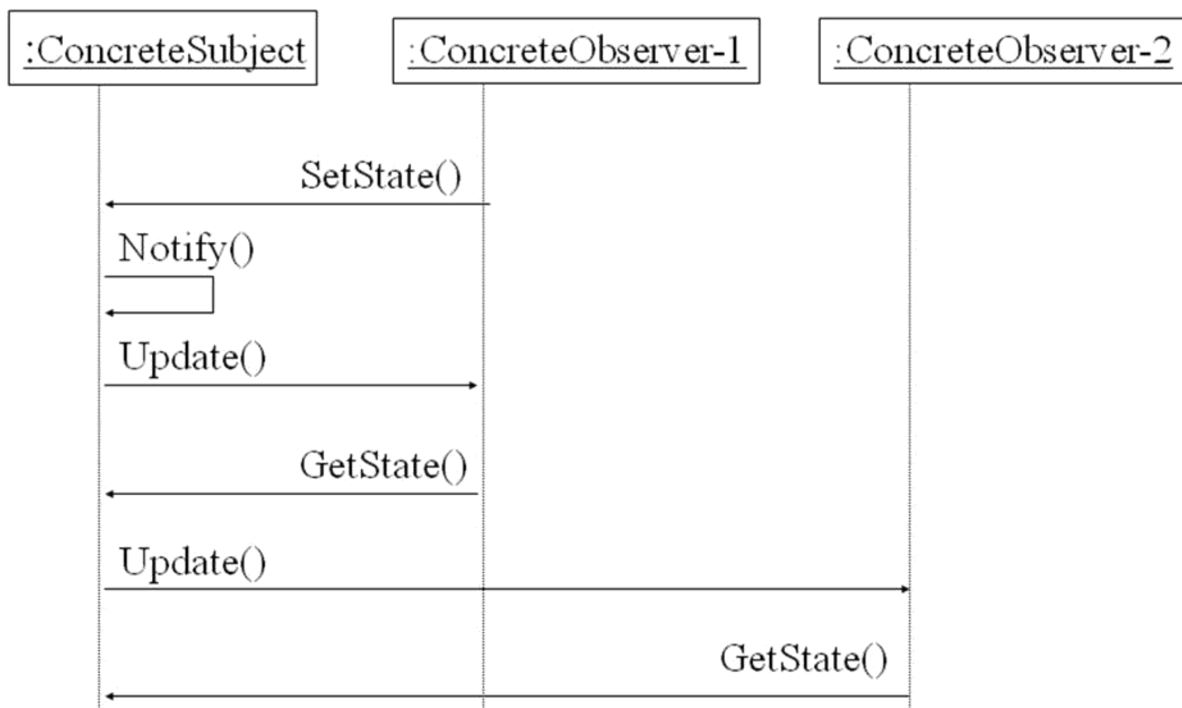
Observer Pattern [1]

- Need to separate presentational aspects with the data, i.e. separate views and data.
- Classes defining application data and presentation can be reused.
- Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.
- Defines one-to-many dependency amongst objects so that when one object changes its state, all its dependents are notified.





Class collaboration in Observer



Observer Pattern: Observer code

```
class Subject;

class observer {
public:
        virtual ~observer;

        virtual void Update (Subject* theChangedSubject)=0;

protected:

        observer ();

};
```

Abstract class defining the Observer interface.

Note the support for multiple subjects.

Observer Pattern: Subject Code

```
class Subject {
public:
        virtual ~Subject;
        virtual   void Attach (observer*);
        virtual   void Detach (observer*) ;
        virtual   void Notify();
protected:
        Subject ();
private:
        List <Observer*> *_observers;
};
```

Abstract class defining the Subject interface.

```
void Subject :: Attach (Observer* o){
        _observers -> Append(o);
}
void Subject :: Detach (Observer* o){
        _observers -> Remove(o);
}
void Subject :: Notify (){

                iter.CurrentItem() -> Update(this);

        }
}
```

Observer Pattern: A Concrete Subject [1]

```
class ClockTimer : public Subject {
public:
                ClockTimer();

                virtual int GetHour();

                virtual int GetMinutes();

                virtual int GetSecond();

                void Tick ();

}
```

```
ClockTimer :: Tick {

    // Update internal time keeping state.
    // gets called on regular intervals by an internal
       timer.

              Notify();

 }
```

Observer Pattern: A Concrete Observer [1]

```
class DigitalClock: public Widget, public
    Observer {
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void  Update(Subject*);
    virtual void  Draw();
private:
    ClockTimer* _subject;
    }
```

Override Observer operation.

Override Widget operation.

```
DigitalClock ::DigitalClock (ClockTimer* s) {
        _subject = s;
        _subject→Attach(this);
}

DigitalClock ::~DigitalClock() {
        _subject->Detach(this);
}
```

```
void DigitalClock ::Update (subject* theChangedSubject ) {
    If (theChangedSubject == _subject) {
        Draw();
    }
}
```

Check if this is the clock's subject.

```
void DigitalClock ::Draw () {
  int hour = _subject->GetHour();
  int minute = _subject->GeMinute(); // etc.
  // Code for drawing the digital clock.
}
```

Observer Pattern: Main (skeleton)

```
ClockTimer* timer = new ClockTimer;

DigitalClock* digitalClock = new DigitalClock (timer);
```

Observer Pattern: Consequences

- Abstract coupling between subject and observer. Subject has no knowledge of concrete observer classes. (What design principle is used?)
- Support for broadcast communication. A subject need not specify the receivers; all interested objects receive the notification.

- Unexpected updates: Observers need not be concerned about when then updates are to occur. They are not concerned about each other's presence. In some cases this may lead to unwanted updates.

When to use the Observer Pattern?

- *When* an abstraction has two aspects: one dependent on the other. Encapsulating these aspects in separate objects allows one to vary and reuse them independently.

- *When* a change to one object requires changing others and the number of objects to be changed is not known.

- When an object should be able to notify others without knowing who they are. Avoid tight coupling between objects.

UNIT-V

Behavioral Patterns

Behavioural Patterns Part-II(cont'd) : State, Strategy, Template Method, Visitor, Discussion of Behavioural Patterns.

General Description

- A type of Behavioral pattern.
- Allows an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Uses Polymorphism to define different behaviors for different states of an object.

When to use STATE pattern ?

- State pattern is useful when there is an object that can be in one of several states, with different behavior in each state.
- To simplify operations that have large conditional statements that depend on the object's state.

```
if (myself = happy) then
{
        eatIceCream();
        ....
        }
else if (myself = sad) then
{
        goToPub();
        ....
```

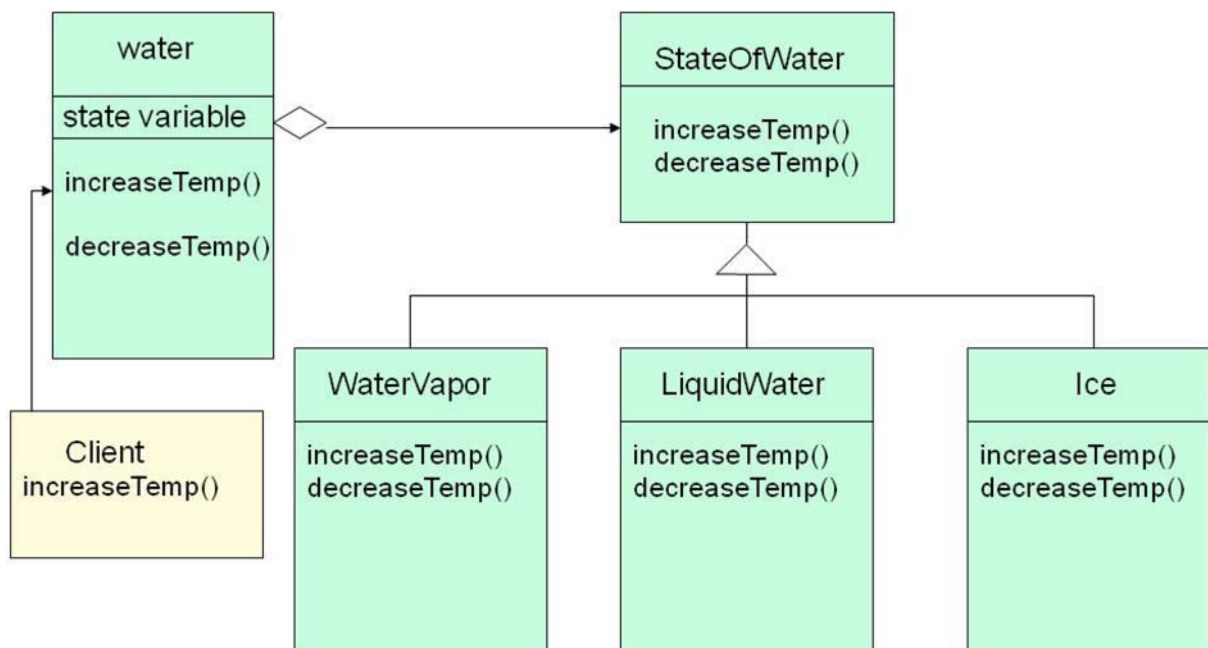
}

else if (myself = ecstatic) then

{

....

Example I



How is STATE pattern implemented ?

- “Context” class:

  Represents the interface to the outside world.

- “State” abstract class:

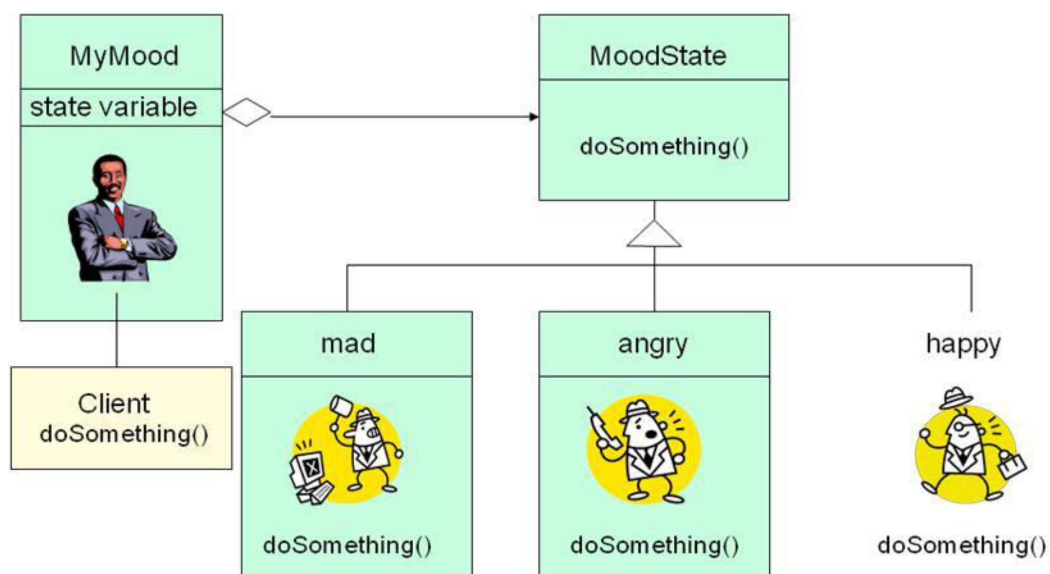
  Base class which defines the different states of the “state machine”.

- “Derived” classes from the State class:

  Defines the true nature of the state that the state machine can be in.

Context class maintains a pointer to the current state. To change the state of the state machine, the pointer needs to be changed.

Example II



Benefits of using STATE pattern

- **Localizes all behavior associated with a particular state into one object.**
  - New state and transitions can be added easily by defining new subclasses.
  - Simplifies maintenance.

- **It makes state transitions explicit.**
  - Separate objects for separate states makes transition explicit rather than using internal data values to define transitions in one combined object.
  - **State objects can be shared.**
  - Context can share State objects if there are no instance variables.

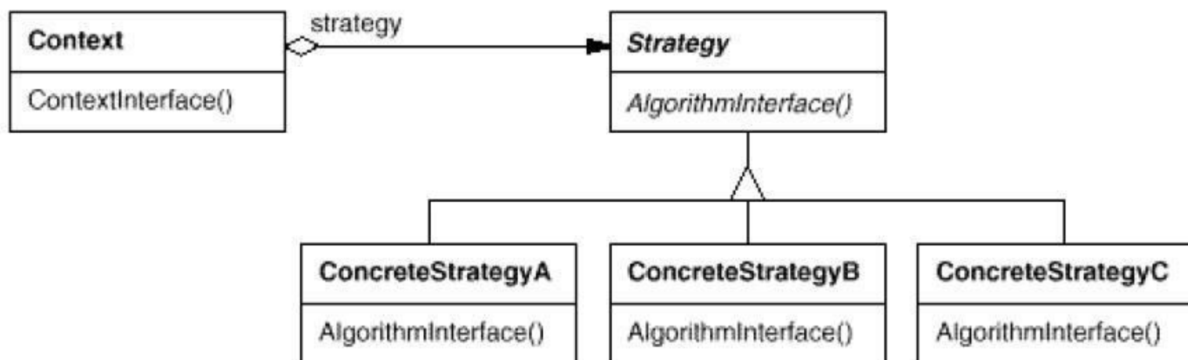
Food for thought…

- **To have a monolithic single class or many subclasses ?**
  - Increases the number of classes and is less compact.
  - Avoids large conditional statements.
  - **Where to define the state transitions ?**

- If criteria is fixed, transition can be defined in the context.
- More flexible if transition is specified in the State subclass.
- Introduces dependencies between subclasses.
- **Whether to create State objects as and when required or to create-them-once-and-use-many-times ?**
- First is desirable if the context changes state infrequently.
- Later is desirable if the context changes state frequently.

Pattern: Strategy

objects that hold alternate algorithms to solve a problem



Strategy pattern

- pulling an algorithm out from the object that contains it, and encapsulating the algorithm (the "strategy") as an object
- each strategy implements one behavior, one implementation of how to solve the same problem
  - how is this different from **Command** pattern?

- separates algorithm for behavior from object that wants to act
- allows changing an object's behavior dynamically without extending / changing the object itself
- **examples**:
  - file saving/compression
  - layout managers on GUI containers
  - AI algorithms for computer game players

Strategy example: Card player

```
// Strategy hierarchy parent

// (an interface or abstract
class) public interface Strategy
{
  public Card getMove();

}

// setting a strategy
player1.setStrategy(new
SmartStrategy());
// using a strategy

Card p1move = player1.move(); // uses strategy
```

Strategy: Encapsulating Algorithms

Name: Strategy design pattern

Problem description:

Decouple a policy-deciding class from a set of mechanisms, so that different mechanisms can be changed transparently.

Example:

A mobile computer can be used with a wireless network, or connected to an Ethernet, with dynamic switching between networks based on location and network costs.

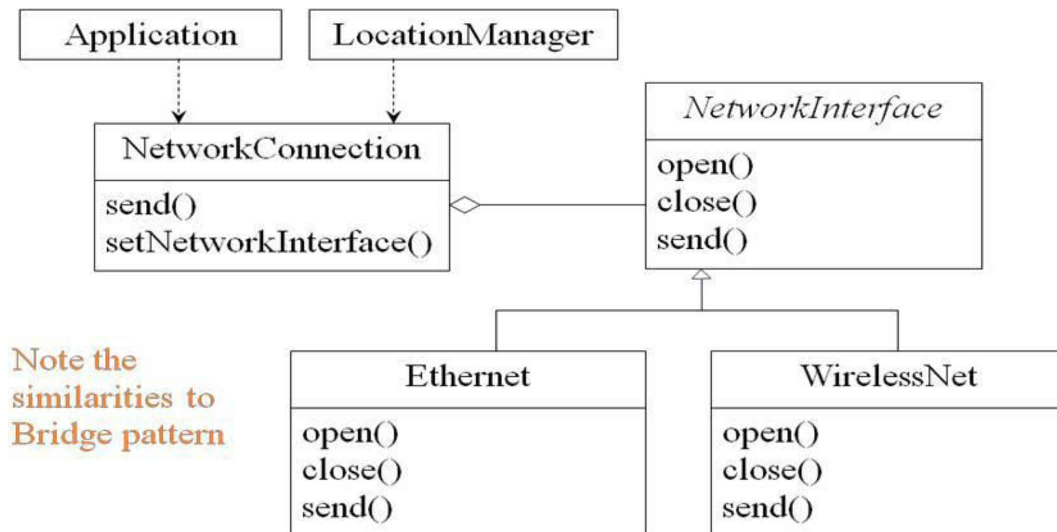
Solution:

A Client accesses services provided by a Context.

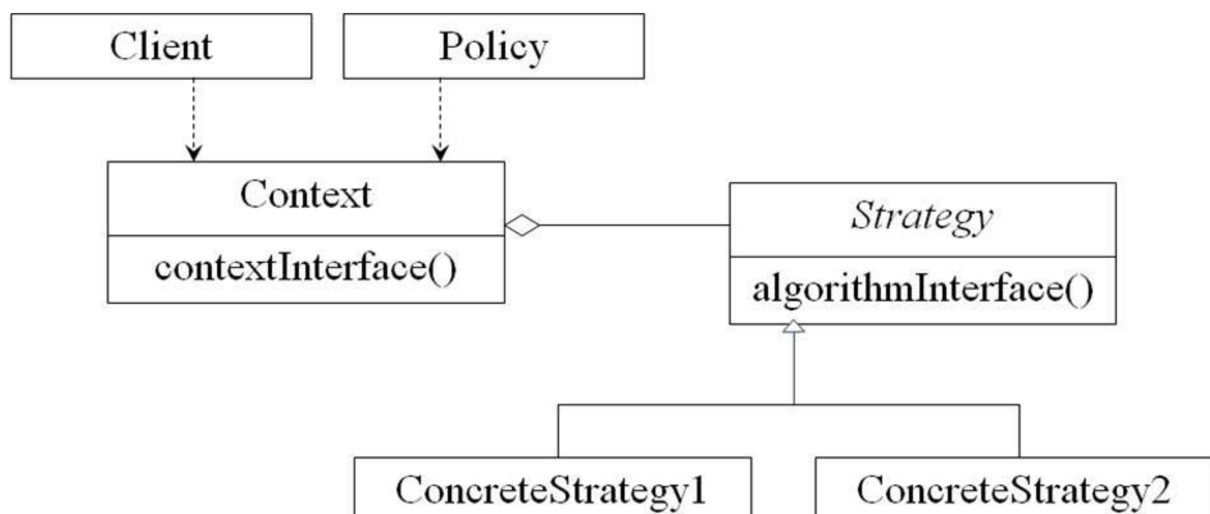
The Context services are realized using one of several mechanisms, as decided by a Policy object.

The abstract class Strategy describes the interface that is common to all mechanisms that Context can use. Policy class creates a ConcreteStrategy object and configures Context to use it.

Strategy Example: Class Diagram for Mobile Computer



Strategy: Class Diagram



Strategy: Consequences

Consequences:

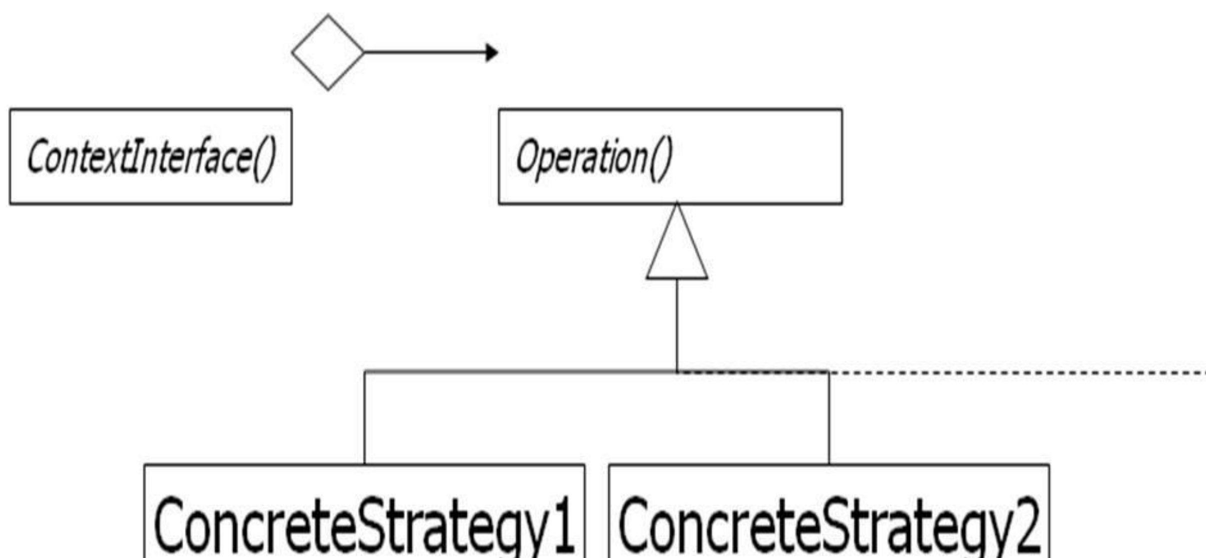
ConcreteStrategies can be substituted transparently from Context.

Policy decides which Strategy is best, given the current circumstances.

New policy algorithms can be added without modifying Context or Client.

Strategy

- **You want to**
  - use different algorithms depending upon the context
  - avoid having to change the context or client
- **Strategy**
  - decouples interface from implementation
  - shields client from implementations
  - Context is not aware which strategy is being used; Client configures the Context
  - strategies can be substituted at runtime
  - example: interface to wired and wireless networks
- Make algorithms interchangeable---"changing the guts"
- Alternative to subclassing
- Choice of implementation at run-time
- Increases run-time complexity



Template Method

Conducted By Raghavendar Japala

Topics – Template Method

- Introduction to Template Method Design Pattern
- Structure of Template Method
- Generic Class and Concrete Class
- Plotter class and Plotter Function Class

Introduction

The DBAnimationApplet illustrates the use of an **abstract class** that serves as a template for classes with shared functionality.

An abstract class contains behavior that is common to all its subclasses. This behavior is encapsulated in nonabstract methods, which may even be declared ***final*** to prevent any modification. This action ensures that all subclasses will inherit the same common behavior and its implementation.

The abstract methods in such templates ensure the interface of the subclasses and require that context specific behavior be implemented for each concrete subclass.

Hook Method and Template Method

The abstract method paintFrame() acts as a placeholder for the behavior that is implemented differently for each specific context.

We call such methods, *hook* methods, upon which context specific behavior may be hung, or implemented.

The paintFrame() hook is placed within the method update(), which is common to all concrete animation applets. Methods containing hooks are called *template* methods.

Hook Method and Template Method (Con't)

The abstract method paintFrame() represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call paintFrame() a hook method. Using the hook method, we are able to define the update() method, which represents a behavior common to all the concrete animation applets.

Frozen Spots and Hot Spots

A template method uses hook methods to define a common behavior.

Template method describes the fixed behaviors of a generic class, which are sometimes called **frozen spots**.

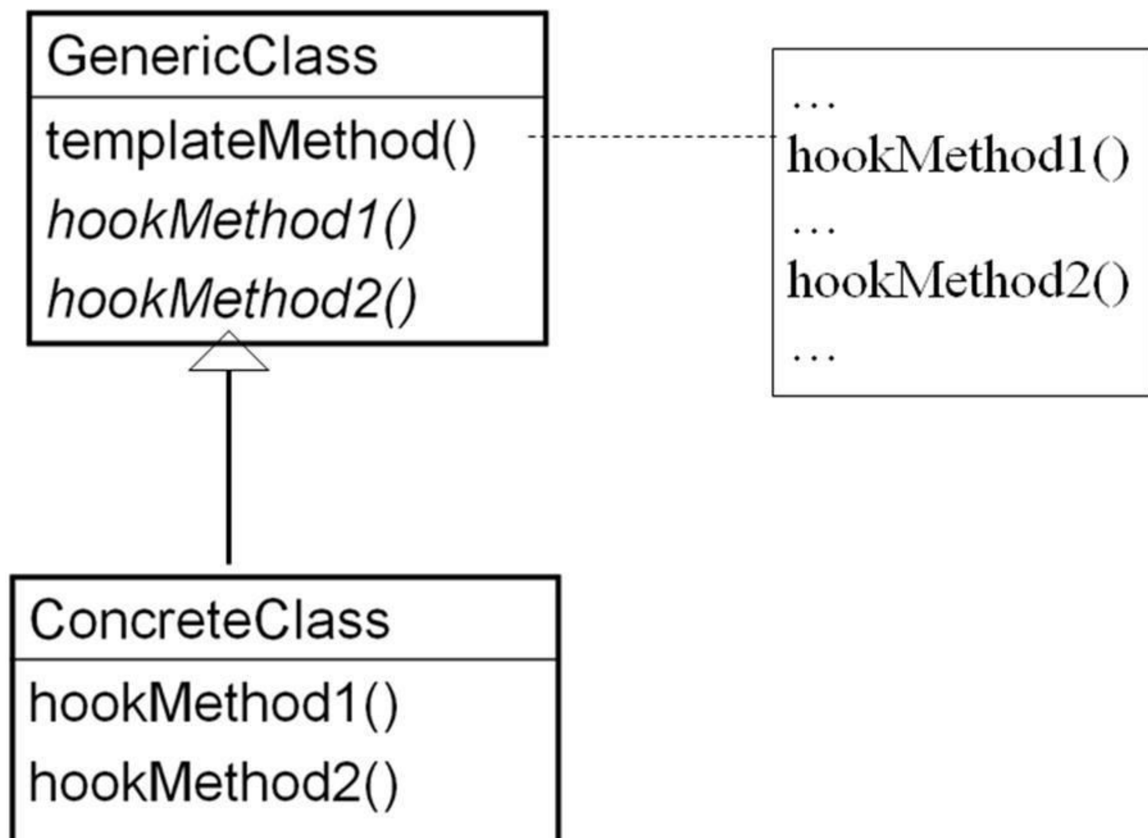
Hook methods indicate the changeable behaviors of a generic class, which are sometimes called **hot spots**.

Hook Method and Template Method (Con't)

The abstract method paintFrame() represents the behavior that is changeable, and its implementation is deferred to the concrete animation applets.

We call paintFrame() a hook method. Using the hook method, we are able to define the update() method, which represents a behavior common to all the concrete animation applets.

Structure of the Template Method Design Pattern



Structure of the Template Method Design Pattern (Con't)

GenericClass (e.g., DBAnimationApplet), which defines abstract hook methods (e.g., paintFrame()) that concrete subclasses (e.g., Bouncing-Ball2) override to implement steps of an algorithm and implements a template method (e.g., update()) that defines the skeleton of an algorithm by calling the hook methods;

ConcreteClass (e.g., Bouncing-Ball2) which implements the hook methods (e.g., paintFrame()) to carry out subclass specific steps of the algorithm defined in the template method.

Structure of the Template Method Design Pattern (Con't)

In the Template Method design pattern, *hook methods* **do not** have to be abstract.

The generic class may provide default implementations for the hook methods.

Thus the subclasses have the option of overriding the hook methods or using the default implementation.

The initAnimator() method in DBAnimationApplet is a
nonabstract hook method with a default implementation.
The init() method is another template method.

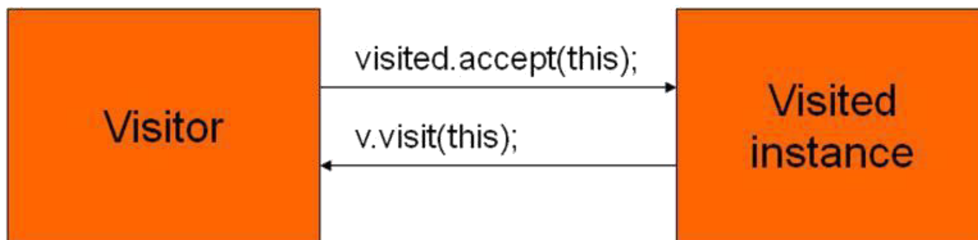
A Generic Function Plotter

The generic plotter should factorize all the behavior related to drawing and leave only the definition of the function to be plotted to its subclasses.

A concrete plotter PlotSine will be implemented to plot the function
$y = \sin x$

Pattern Hatching

Visitor pattern



L8

Pattern Hatching

Visitor Pattern

```
class ClockTimer : public Subject {
public:
    ClockTimer();
    virtual int GetHour();
    virtual int GetMinutes();
    virtual int GetSecond();
    void Tick ();
}
```

```
Class Visitor {
public:
        Visitor();
        void visit(File*);
        void visit(Directory*);
        void visit (Link*);
};

Visitor cat;
node->accept(cat);
```

```
void Visitor::visit (File* f)
        {f->streamOut(cout);}
void Visitor::visit (Directory* d)
        {cerr << "no printout for a directory";}
void Visitor::visit (Link* l)
        {l->getSubject()->accept(*this);}
```

```
void File::accept (Visitor& v)
        {v.visit(this);}
void Directory::accept (Visitor& v)
        {v.visit(this);}
void Link::accept (Visitor& v)
        {v.visit(this);}
```

What to Expect from Design Patterns, A Brief History, The Pattern Community
An Invitation, A Parting Thought.

What to Expect from Design Patterns?

- A Common Design Vocabulary.
- A Documentation and Learning Aid.
- An Adjunct to Existing Methods.
- A Target for Refactoring.

A common design vocabulary

1. Studies of expert programmers for conventional languages have shown that knowledge and experience isn't organized simply around syntax but in larger conceptual structures such as algorithms, data structures and idioms [AS85, Cop92, Cur89, SS86], and plans for fulfilling a particular goal [SE84].

2. Designers probably don't think about the notation they are using for recording the designing as much as they try to match the current design situation against plans, data structures, and idioms they have learned in the past.

3. Computer scientists name and catalog algorithms and data structures, but we don't often name other kinds of patterns. Design patterns provide a common vocabulary for designers to use to communicate, document, and explore design alternatives.

A document and learning aid:

1. Knowing the design patterns makes it easier to understand existing systems.

2. Most large object-oriented systems use this design patterns people learning object-oriented programming often complain that the systems they are working with use inheritance in convoluted ways and that it is difficult to follow the flow of control.

3. In large part this is because they do not understand the design patterns in the system learning these design patterns will help you understand existing object-oriented system.

An adjacent to existing methods:

1. Object-oriented design methods are supposed to promote good design, to teach new designers how to design well, and standardize the way designs are developed.

2. A design method typically defines a set of notations (usually graphical) for modeling various aspects of design along with a set of rules that govern how and when to use each notation.

3. Design methods usually describe problems that occur in a design, how to resolve them and how to evaluate design. But then have not been able to capture the experience of expert designers.

4. A full fledged design method requires more kinds of patterns than just design patterns there can also be analysis patterns, user interface design patterns, or performance tuning patterns but the design patterns are an essential part, one that's been missing until now.

A target for refactoring:

1. One of the problems in developing reusable software is that it often has to be recognized or refactored [OJ90].

2. Design patterns help you determine how to recognize a design and they can reduce a amount of refactoring need to later.

The life cycle of object-oriented software has several faces. Brain Foote identifies these phases as the prototyping expansionary, and consolidating phases [Foo92**]**.

Design Patterns Applied:

Example: An Hierarchical File System

Tree Structure □ Composite

Patterns Overview

Symbolic Links □ Proxy

Extending Functionality □ Visitor

Single User Protection □ Template Method

Multi User Protection □ Singleton

User and Groups □ Mediator

A Brief History of Design Patterns

- 1979--Christopher Alexander pens The Timeless Way of Building
  - Building Towns for Dummies
  - Had nothing to do with software
- 1994--Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (the Gang of Four, or GoF) publish Design patterns: Elements of Reusable Object-Oriented Software
  - Capitalized on the work of Alexander
  - The seminal publication on software design patterns.

What's In a Design Pattern—1994

- The GOF book describes a pattern using the following four attributes:
  - The *name* to describes the pattern, its solutions and consequences in a word or two
  - The *problem* describes when to apply the pattern
  - The *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - The *consequences* are the results and trade-offs in applying the pattern
- All examples in C++ and Smalltalk.

What's In a Design Pattern – 2002

- Grand's book is the latest offering in the field and is very Java centric. He develops the GOF attributes to a greater granularity and adds the Java specifics
  - Pattern name—same as GOF attribute
  - Synopsis—conveys the essence of the solution
  - Context—problem the pattern addresses
  - Forces—reasons to, or not to use a solution

- Solution—general purpose solution to the problem
- Implementation—important considerations when using a solution
- Consequences—implications, good or bad, of using a solution
- Java API usage—examples from the core Java API
- Code example—self explanatory
- Related patterns—self explanatory

Grand's Classifications of Design Pattern:

- Fundamental patterns
- Creational patterns
- Partitioning patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns

The Pattern Community An Invention

- Christopher Alexander is the architect who first studied
Patterns in buildings and communities and developed

A PATTERN LANGUAGE for generating them.

- His work has inspired time and again. So it's fitting worth

while To compare our work to his.

- Then we'll look at other's work in software-related patterns.

Alexander's Pattern Languages

There are many ways in which our work is like Alexander's

Both are based on observing existing systems and looking for patterns in them.
Both have templates for describing patterns although our templates are quite different)..

But there are just as many ways in which our work different.

- People have been making buildings for thousands of years, and there are many classic examples to draw upon. We have been making Software systems for a
- Alexander gives an order in which his patterns should be used; we have not.
- Alexander's patterns emphasize the problems they adderss ,
- where as design patterns describes the solutions in more detail.
- Alexander claims his patterns will generate complete buildings.

We do not claim that our patterns will generate complete programs.

When Alexander claims you can design a house simply applying his patterns one after Another ,he has goals similar to those of object-oriented design methodologies who Gives step-by-step rules for design,

In fact ,we think it’s unlikely that there will ever be a compete pattern language for soft -ware.

But certainly possible to make one that is more complete.

A Parting Thought.

The best designs will use many design patterns that dovetail And intertwine to produce a greater whole.

As Alexander says:

It is possible to make buildings by stringing together pattern’s,
In a rather loose way,

A building made like this , is an assembly of patterns. it is not Dense.

It is not profound. but it is also possible to put pattern’s together

In such a way that many patterns overlap in the same physical

Space: the building is very dense; it has many meaning captured

In a small space; and through this density, it becomes profound.