SOFTWARE ARCHITECTURE AND DESIGN PATTERNS (Effective from the academic year 2018 -2019) SEMESTER – VII			
Course Code	**18CS731**	**CIE Marks**	40
Number of Contact Hours/Week	3:0:0	**SEE Marks**	60
Total Number of Contact Hours	40	**Exam Hours**	03
CREDITS –3			

Course Learning Objectives: This course (18CS731) will enable students to:

- Learn How to add functionality to designs while minimizing complexity.
- What code qualities are required to maintain to keep code flexible?
- To Understand the common design patterns.
- To explore the appropriate patterns for design problems

Module 1	**Contact Hours**
Introduction: what is a design pattern? describing design patterns, the catalog of design pattern, organizing the catalog, how design patterns solve design problems, how to select a design pattern, how to use a design pattern. A Notation for Describing Object-Oriented Systems **Textbook 1: Chapter 1 and 2.7** **Analysis a System**: overview of the analysis phase, stage 1: gathering the requirements functional requirements specification, defining conceptual classes and relationships, using the knowledge of the domain. Design and Implementation, discussions and further reading. **Textbook 1: Chapter 6** **RBT: L1, L2, L3**	08
Module 2	
Design Pattern Catalog: Structural patterns, Adapter, bridge, composite, decorator, facade, flyweight, proxy. **Textbook 2: chapter 4** **RBT: L1, L2, L3**	08
Module 3	
BehavioralPatterns: Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Template Method **Textbook 2: chapter 5** **RBT: L1, L2, L3**	08
Module 4	
Interactive systems and the MVC architecture: Introduction, The MVC architectural pattern, analyzing a simple drawing program, designing the system, designing of the subsystems, getting into implementation, implementing undo operation, drawing incompleteitems, adding a new feature, pattern-based solutions. **Textbook 1: Chapter 11** **RBT: L1, L2, L3**	08
Module 5	
Designing with Distributed Objects: Client server system, java remote method invocation, implementing an object-oriented system on the web (discussions and further reading) a note on input and output, selection statements, loops arrays. **Textbook 1: Chapter 12** **RBT: L1, L2, L3**	08

Course Outcomes: The student will be able to :

- Design and implement codes with higher performance and lower complexity
- Be aware of code qualities needed to keep code flexible

- Experience core design principles and be able to assess the quality of a design with respect to these principles.
- Capable of applying these principles in the design of object oriented systems.
- Demonstrate an understanding of a range of design patterns. Be capable of comprehending a design presented using this vocabulary.
- Be able to select and apply suitable patterns in specific contexts

Question Paper Pattern:

- The question paper will have ten questions.
- Each full Question consisting of 20 marks
- There will be 2 full questions (with a maximum of four sub questions) from each module.
- Each full question will have sub questions covering all the topics under a module.
- The students will have to answer 5 full questions, selecting one full question from each module.

Textbooks:

1. Brahma Dathan, Sarnath Rammath, Object-oriented analysis, design and implementation, Universities Press,2013
2. Erich Gamma, Richard Helan, Ralph Johman, John Vlissides , Design Patterns, Pearson Publication,2013.

Reference Books:

1. Frank Bachmann, ReginеMeunier, Hans Rohnert "Pattern Oriented Software Architecture" –Volume 1, 1996.
2. William J Brown et al., "Anti-Patterns: Refactoring Software, Architectures and Projects in Crisis", John Wiley, 1998.

Module 1 - Introduction

Design pattern

"A proven solution to a common problem in a specified context"

Example: We can light a candle if light goes out at night **Christopher Alexander (Civil Engineer) in 1977 wrote**

"A 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"

Essential Elements:

The **pattern name** is a handle we can use to describe a design problem, 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 pattern provides an abstract description of a design problem and how a general arrangement of classes and objects solves it.

The **consequences** are the results and trade-offs of applying the pattern.

Example Pattern:

Pattern Name – Iterator

Problem – How to serve Patients at a Doctor's Clinic

Solution – Front-desk manages the order for patients to be called

- By Appointment
- By Order of Arrival
- By Extending Gratitude
- By Exception

Consequences

- Patient Satisfaction
- Clinic's Efficiency
- Doctor's Productivity

Describing Design Patterns

Pattern Name & Classification – Conveys the essence of the pattern concisely

Intent – What design issue the pattern addresses

Also Known As – Other well-known names for this pattern

Motivation – A scenario illustrating a design problem and how it's being solved by the pattern

Applicability – Known situations where the pattern can be applied

Structure – OMT (Object Modelling Technique) based graphic representation of the classes in the pattern

Participants – Classes and objects in the pattern with 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 – Hints on implementation of the pattern like language dependency

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

Sample Code – Code fragments to implement the pattern in specific language (C++or C# or java).

Known Uses – Examples of the pattern found in real systems.

Related Patterns – Other patterns closely related with the pattern under consideration

- 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 Pattern

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

Adapter:

- Convert the inter face of a class into another interface client's expect.
- Adapter lets classes work together

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

Builder: Separates the construction of the complex object from its representation so that the same construction process can create different representations.

Chain of Responsibility: Avoid coupling the sender of a request to it's 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 objects handles it.

Command: Encapsulate a request as an object, thereby letting parameterize clients with different request, 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 sub classing for extending functionality.

Façade: 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.

Factory Method: Defines 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: For the given language, it defines the representation of its grammar to interpret sentences in the language.

Iterator: Provide a way to access the element 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 of objects and allows to vary their interaction independently.

Memento: Without violating encapsulation, capture and externalize an object's internal state so that 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 it's dependents are notified and updated automatically.

Prototype: Create new objects by copying existing objects.

Proxy: Provide a surrogate or placeholder (substitute) to control the access to the original object.

Singleton: Ensure a class has only one instance, and provide a 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 them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method: Define the Skelton of an operation, deferring some steps to subclasses. Template method subclasses redefine certain steps of an algorithm without changing the algorithms 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

We classify the design patterns by two criteria.

The ***first criterion,*** called **purpose,** reflects what a pattern does.

1. **Creational patterns** concern the process of object creation.
2. **Structural patterns** deal with the composition of classes or objects.
3. **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.

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

Table 1.1: Design pattern space

How Design Patterns solve design problems

- **Finding Appropriate Objects**
- **Determining Object Granularity**
- **Specifying Object Interfaces**
- **Specifying Object Implementations**
- **Class versus Interface Inheritance**
- **Programming to an Interface, not an Implementation**
- **Putting Reuse Mechanisms to Work**
- **Relating Run-Time and Compile-Time Structures**
- **Designing for Change**

1. Finding Appropriate Objects

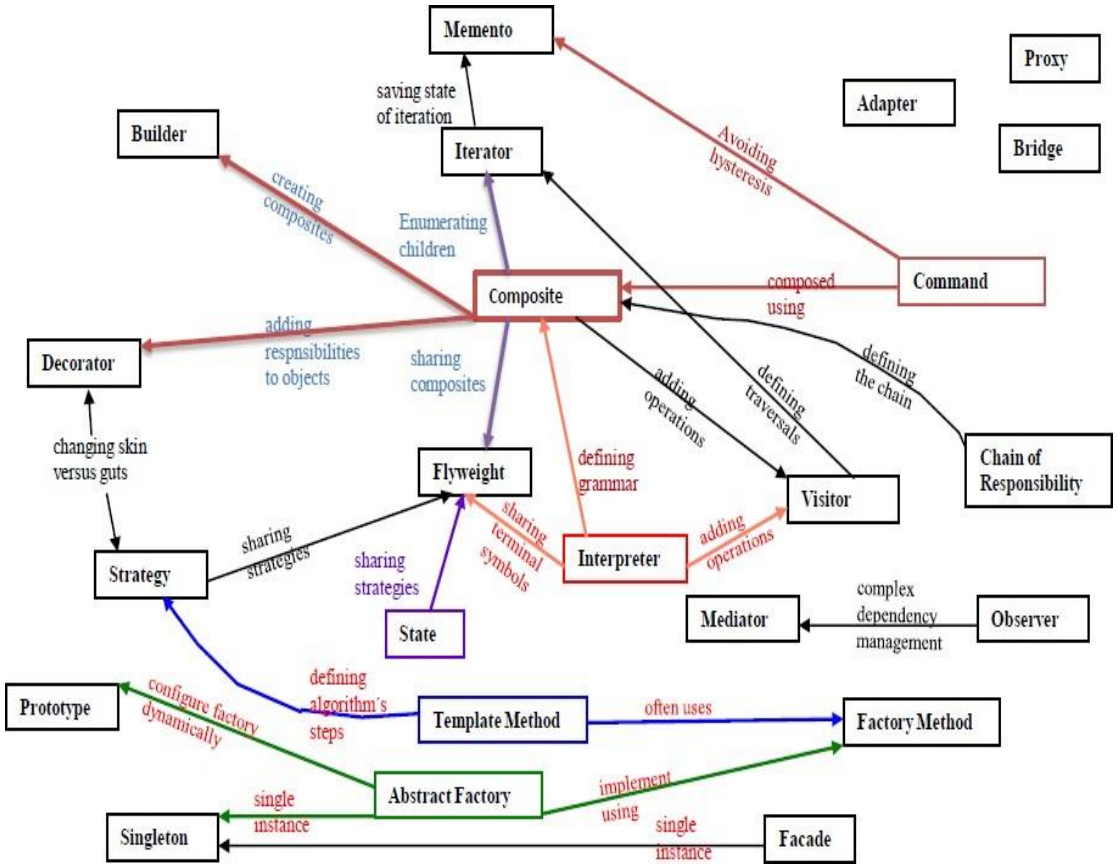
- An object packages both data and the procedures (code), where the Procedures are the methods or operations to be performed.
- Objects are encapsulated during the execution and therefore objects cannot be accessed directly, and its representation is invisible from outside.
- Decomposing a system into objects is the hard part because the parameters like encapsulation, granularity, dependency, flexibility, performance, evolution, reusability era to be considered in the object-oriented design.
- Object-oriented design methodologies includes different approaches
  - ✓ We can write problem statement, single out nouns and verbs, and create corresponding classes and operations

- ✓ Focus the collaborations and responsibilities in the system
- ✓ Strict modeling of the real world and translating the objects found during analysis into design.

- Object-oriented design end up with low level classes like arrays
- The abstractions are necessary to make the design flexible
- Design pattern helps us to identify less-obvious abstractions.
  - ✓ Strategy pattern describes how to implement interchangeable families of algorithms.
  - ✓ State pattern represents each state of an entity as an object

2. Determining Object Granularity

- Objects can vary tremendously in size and number
- Facade pattern describes how to represent subsystems as objects
- Flyweight pattern describes how to support huge numbers of objects
- Abstract Factory and Builder take the responsibilities of creating other objects
- Visitor and Command pattern implement a request on another object or group of objects.



3. 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.
- 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.
- **Subtype** *inheriting* the interface of its **Supertype.**
- Objects are known only through their interfaces.
- The run-time association of a request to an object and one of its operations is known as **dynamic binding.**
- Design patterns help programmers to define interfaces by identifying their key elements and the kind of data that get sent across an interface. A design pattern can also tell what not to put in the interface

Interface:

- Set of all signatures defined by an object's operations
- Any request matching a signature in the objects interface can be sent to the object
- Interfaces may contain other interfaces as subsets

Type:

- Denotes a particular interfaces
- An object may have many types
- Widely different object may share a type
- Objects of the same type need only share parts of their interfaces
- A subtype contains the interface of its super type

Dynamic Binding, Polymorphism

Binding

- Operation to be performed depends on the request and the object
- Run-time association of a request to an object and this operation is known as dynamic binding
- Requests does not allow to a particular implementation until run-time

Polymorphism

- Simplifies the definitions of Clients
- Decouples the objects from each other
- Objects vary their relationships to each other at run-time

An object's implementation is defined by its class

The class specifies the object's internal data and defines the operations the object can perform

Objects is created by instantiating a class

- An object = An instance of a class

Class inheritance

- Parent class and subclass

Memento Pattern define two interfaces

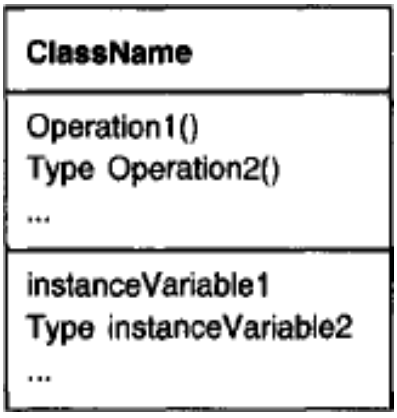
- Restricted one that lets clients hold and copy
- Privileged one that only the original object can reuse to store and retrieve state

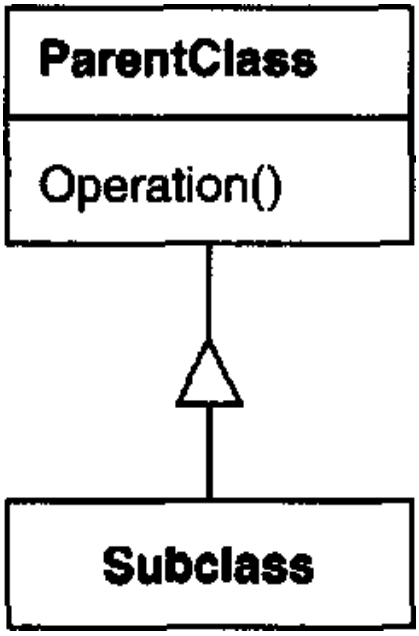
Decorator and Proxy patterns are used for interfaces of objects

Visitor is used to reflect all classes of objects that visitors can visit

4. 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 that the object can perform.
- A dashed arrowhead line indicates a class that instantiates objects of another class. The arrow points to the class of the instantiated objects.





Inheritance

- New classes can be defined in terms of existing classes using class **inheritance.** When a subclass inherits from a parent class, it includes the definitions of all the data and operations that the parent class defines. Objects that are instances of the subclass will contain all data defined by the subclass and its parent classes.
- We indicate the subclass relationship with a vertical line and a triangle.

Abstract Class

- **Abstract Class** is one whose main purpose is to define a common interface for its subclasses. The operations that an abstract class declares but doesn't implement are called **abstract operations.**
- The names of abstract classes appear in slanted type. Slanted type is also used to denote abstract operations.
- The implementation of the operation is represented by dog-eared box, the code will appear connected with a dashed line to the operation it implements

Concrete classes

- Classes that are not abstract are called concrete classes
- A concrete classes implement creation methods of the abstract factory

Override an operation

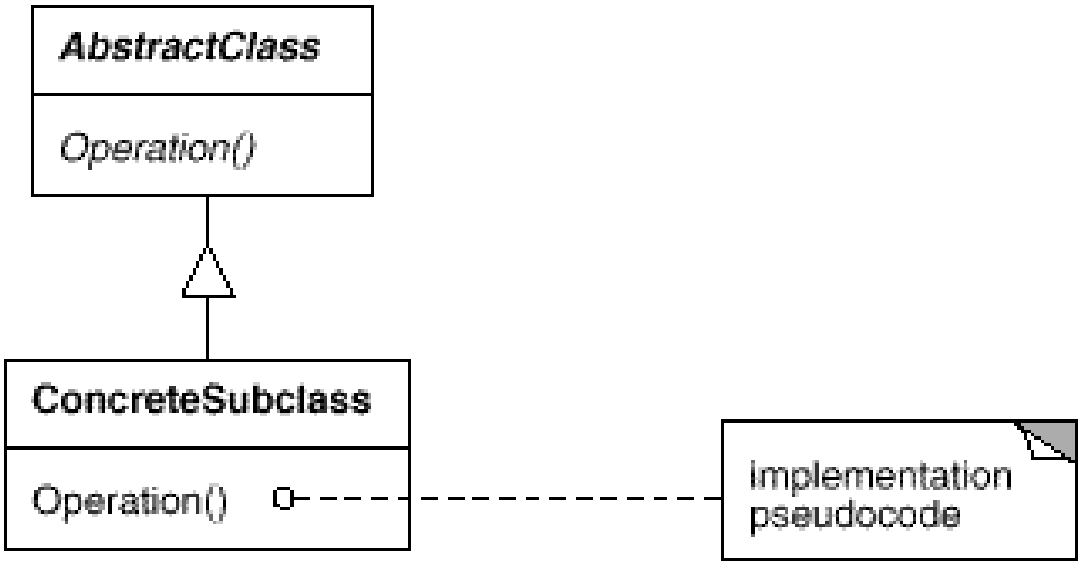
- Subclasses override an operation defined by its parent classes
- Subclasses redefines the behavoirs of their parent classes

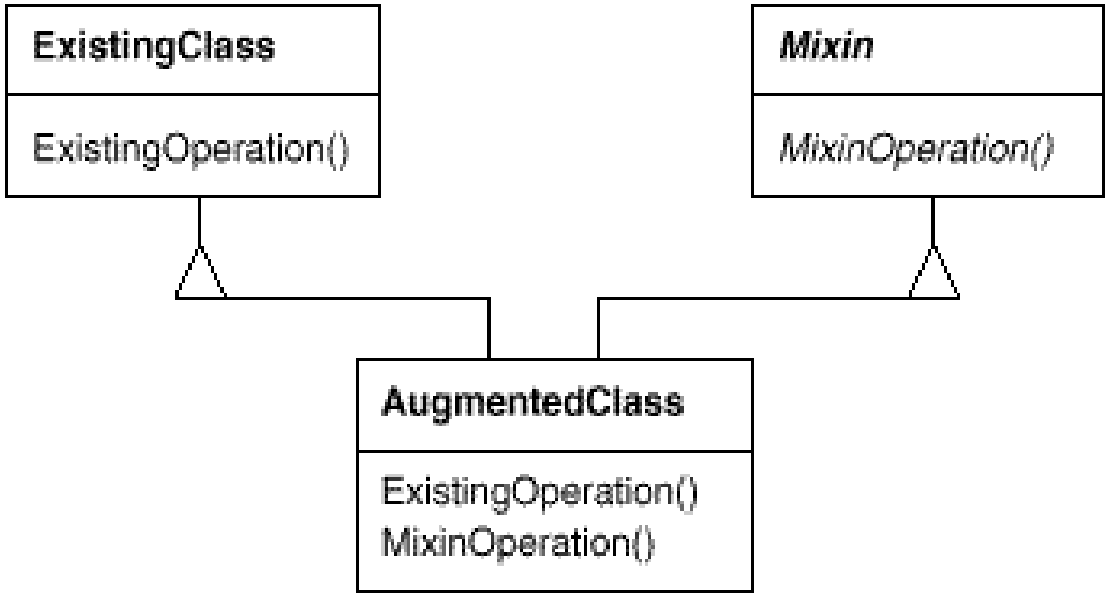
Mixin Class

A Mixin class is a class that's intended to provide an optional interface or functionality to other classes. Mixin classes require multiple inheritances

Augmented class:

Allows user to create own projects without having any previous knowledge





5. Class versus Interface Inheritance

- The class defines the object's internal state and the implementation of its operations.
- In contrast, an object's type only refers to its interface—the set of requests to which it can respond.
- An object can have many types, and objects of different classes can have the same type.
- An object is an instance of a class; we imply that the object supports the interface defined by the class.
- Class inheritance defines an object's implementation in terms of another object's implementation.
- Interface inheritance (or subtyping) describes when an object can be used in place of another.

Examples: Chain of Responsibility, Composite pattern, Command, Observer, State, and Strategy.

6. Programming to an Interface, not an Implementation

- Class inheritance is a mechanism for extending an application's functionality by reusing functionality in parent classes.
- When inheritance is used all classes derived from an abstract class will share its interface.
- All subclasses can then respond to the requests in the interface of this abstract class

Benefits

- Clients remain unaware of the specific types of objects they use
- Clients remain unaware of the classes that implement these objects, clients only know about the abstract classes defining the interface.

This leads to the first principle of reusable object-oriented design:

Instantiation of Concrete classes

- Abstract Factory, Builder, Factor method, Prototype and Singleton are the creational patterns
- Creational patterns ensures that the system is written in terms of interfaces, not implementations

7. Putting Reuse Mechanisms to work

The challenge lies in applying the concepts like objects, interfaces, classes and inheritance to build the design patterns to be flexible and reusable

- **Inheritance** versus **Composition**
- **Delegation**
- **Inheritance** versus **Parameterized Types**

Inheritance verses composition

- Two techniques for reusing the functionality in object-oriented systems are class inheritance and object composition
    - class inheritance
        - ✓ White-box reuse
    - object composition
        - ✓ Black-box reuse

White-box reuse:

- ✓ Reuse by sub classing (class inheritance)
- ✓ Internals of parent classes are often visible to subclasses
- ✓ works statically, compile-time approach
- ✓ Inheritance breaks encapsulation

Black-box reuse:

- ✓ Reuse by object composition
- ✓ Requires objects to have well-defined interfaces
- ✓ No internal details of objects are visible

- Class inheritance define the implementation of one class in terms of the other
- **Class inheritance:** Reuse by sub classing 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.
- Defined at compile-time. and straightforward to use
- "Inheritance breaks encapsulation" (superclass implementation exposed to subclasses)

Advantages

- ✓ Static, straightforward to use
- ✓ Make the implementations being reuse more easily

Disadvantages

- ✓ The implementations inherited can't be changed at run time, because inheritance is defined at compile time
- ✓ Parent classes often define at least part of their subclasses physical representation
    - Breaks encapsulation
- ✓ Implementation dependencies can cause problems(limits flexibility and reusability) when we try to reuse a subclass

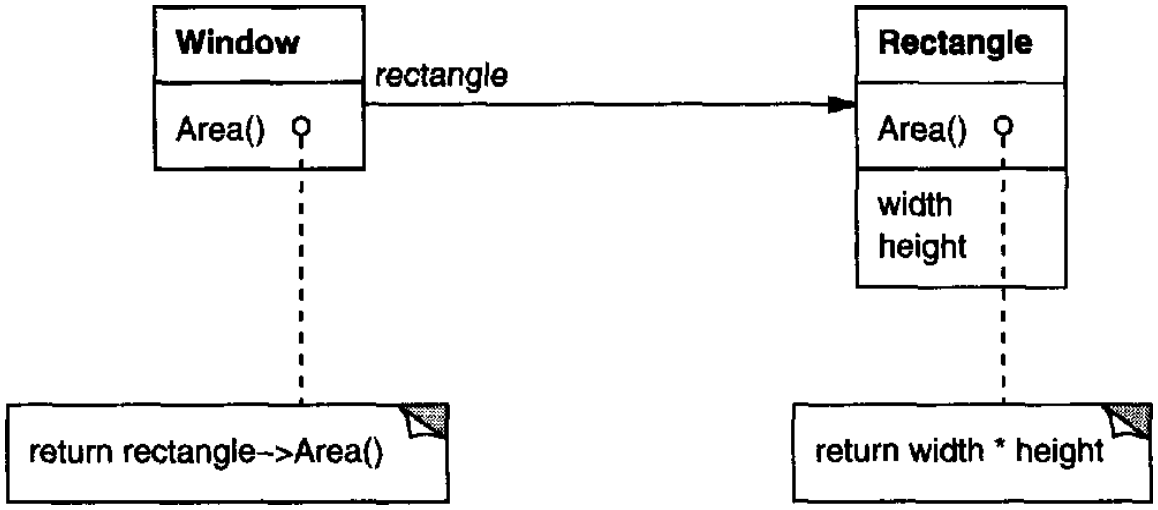
Object composition:

New functionality is obtained by assembling or *composing* objects to get more complex functionality. This style of reuse is called "***black-box reuse"***, because no internal details of objects are visible.

- Defined at run-time by objects acquiring references to other objects.
- Must program to interfaces, so interfaces must be well thought-out and stable.
- Emphasis on interface stability encourages granular objects with single responsibilities

Delegation

In delegation, *two* objects are involved in handling a request: receiving object delegates operations to its delegate



- **Advantages:** Makes it easy to compose behaviors at run-time and to change the way they are composed.
- **Disadvantages:** Dynamic, highly parameterized software is harder to understand than more static software and there are also run-time inefficiencies
- Delegation is a good design choice only when it simplifies more than it complicates
- Delegation is an extreme example of object composition

Example: Several design patterns use delegation, such as:

a. State: Here an object delegates requests to a State object that represents its current state
b. Strategy: Here an object delegates a specific request to an object that represents a strategy for carrying out the request.

Inheritance versus Parameterized Types

- Another technique for reusing functionality is through parameterized types, also known as generics in ADA and templates in C++
- Allows to define a type without specifying all the other types it uses, the unspecified types are supplied as parameters at the point of use
- **For example** :
  - To declare a list of integers, we supply the type "integer" as a parameter
  - To declare a list of String objects, we supply the "String" type as a parameter.
- Parameterized types, generics, or templates
- Parameterized types gives us a third way to compose behavior in object-oriented systems
- Many designs can be implemented using any of these three techniques.

- ✓ An operation implemented by subclasses (an application of Template Method)
- ✓ The responsibility of an object that is passed to the sorting routine (Strategy)
- ✓ An argument of a C++ template or Ada generic that specifies the name of the function is called to compare the elements.

- There are important differences between these techniques.
  - **Object composition** lets us to change the behavior being composed at run-time, but it requires indirection and can be less efficient
  - **Inheritance** lets us to provide default implementations for operations and lets subclasses override them
  - **Parameterized** types let us to change the types that a class can use

8. Relating Run-Time and Compile-Time Structures

- An object-oriented program's **run-time structure** often bears little resemblance to its **code structure**
- The code structure is frozen at compile-time
- A program's run-time structure consists of rapidly changing networks of communicating objects

❖ **Aggregation** versus **Acquaintance** (Association)

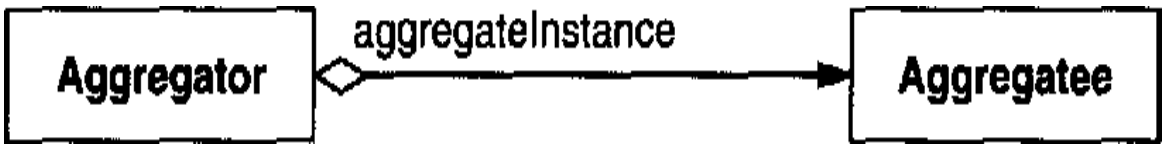
Aggregation

- ✓ Aggregation implies that one object owns or responsible for another object
- ✓ Aggregation implies that an aggregate object and its owner have identical lifetimes
- ✓ Generally we speak of an object *having* or being *part* of another object.
- ✓ Aggregation relationships tend to be permanent than acquaintance.

Acquaintance

- ✓ Acquaintance implies that an object merely knows of another object
- ✓ Acquainted objects request operations of each other, but they are not responsible for each other.
- ✓ Acquaintance is a weaker relationship than aggregation and suggests much looser coupling between the objects
- ✓ Acquaintances are made and remade more frequently,
- ✓ Sometimes Acquaintance is called "Association" or the "using" relationship.

- The distinction between acquaintance and aggregation is determined more by intent than by explicit language mechanisms
- The system's run-time structure must be imposed more by the designer than the language



9. Designing for Change

- The key to maximizing reuse lies in anticipating new requirements and changes to existing requirements, and in designing your systems so that they can evolve accordingly.
- A design that doesn‘t take change into account risks major redesign in the future
- These changes involve class redefinition and reimplementation, client modification and retesting
- Redesign affects many parts of the software system and unanticipated changes are invariably expensive
- Design patterns help us to avoid this by ensuring that a system can change in specific ways
- Each design pattern lets some aspect of system structure vary independently of other aspects
- Here are some common causes of redesign along with the design pattern(s) that address them:

Common Causes of Redesign

- Creating an object by specifying a class explicitly
- Dependence on specific operations
- Dependence on hardware and software platform
- Dependence on object representations or implementations
- Algorithmic dependencies
- Tight coupling
- Extending functionality by sub classing
- Inability to alter classes conveniently

Creating an object by specifying a class explicitly: Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface.

Dependence on specific operations: When you specify a particular operation, you commit to one way of satisfying a request. By avoiding hard-coded requests, you make it easier to change the way a request gets satisfied both at compile-time and at run-time.

Dependence on hardware and software platform: External operating system interfaces and application programming interfaces (APIs) are different on different hardware and software platforms. Software that depends on a particular platform will be harder to port to other platforms. It may even be difficult to keep it up to date on its native platform. It's important therefore to design your system to limit its platform dependencies.

Dependence on object representations or implementations: Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes. Hiding this information from clients keeps changes from cascading.

Algorithmic dependencies: Algorithms are often extended, optimized, and replaced during development and reuse. Objects that depend on an algorithm will have to change when the algorithm changes. Therefore algorithms that are likely to change should be isolated.

Tight coupling: Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes.

Extending functionality by subclassing: Customizing an object by sub classing often isn't easy. Every new class has a fixed implementation overhead (initialization, finalization, etc.). Defining a subclass also requires an in-depth understanding of the parent class. For example, overriding one operation might require overriding another.

Inability to alter classes conveniently: Sometimes you have to modify a class that can't be modified conveniently. Perhaps you need the source code and don't have it (as may be the case with a commercial class library).

Design patterns in Application programs

- If you're building an application program such as a document editor or spreadsheet, then *internal reuse, maintainability, and extension* are high priorities.
- Internal reuse ensures that you don't design and implement any more than you have to.
- Design patterns that reduce dependencies can increase internal reuse.
- Design patterns also make an application more maintainable when they are used to limit platform dependencies and to layer a system
- Looser coupling boosts the likelihood that one class of object can cooperate with several others

- Reduced coupling also enhances extensibility

For example, when you eliminated dependencies on specific operations by isolating and encapsulating each operation, you make it easier to reuse an operation in different contexts.

Design patterns in Toolkits

- A **toolkit** is a set of related and reusable classes designed to provide useful, general-purpose functionality.
- An example of a toolkit is a set of collection classes for lists, associative tables, stacks, and the like.
- The C++ I/O stream library is another example.
- Toolkits emphasize code reuse
- Toolkits are the object-oriented equivalent of subroutine libraries
- Toolkit design is arguably harder than application design
- Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do its job.
- Toolkit design is arguably harder than application design, because toolkits have to work in many applications to be useful.
- Moreover, the toolkit writer isn't in a position to know what those applications will be or their special needs.

Design patterns in Frameworks

- A **framework** is a set of cooperating classes that makeup a reusable design for a specific class of software.
- **For example**, a framework can be geared toward building graphical editors for different domains like artistic drawing, music composition, and mechanical.
- Another framework can help you build compilers for different programming languages and target machines.
- We can customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework
- The framework dictates the **architecture** of the application. It will define the overall structure; it’s partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control.
- The framework captures the design decisions that are common to its application domain.

- Frameworks thus emphasize design reuse over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.
- Frameworks emphasize **design reuse** over code reuse
- When we use a **toolkit**, we can write the main body of the application and call the code which we want to reuse. When we use a framework, we reuse the main body and write the code **it** calls.
- **Advantages:** Builds an application faster, easier to maintain, and more consistent to their users
- Mature frameworks usually incorporate several design patterns
- The patterns help make the framework's architecture suitable to many different applications without redesign
- An added benefit comes when the framework is documented with the design patterns it uses.
- People who know the patterns gain insight into the framework faster.
- Even people who don't know the patterns can benefit from the structure they lend to the framework's documentation.
- Enhancing documentation is important for all types of software, but it's particularly important for frameworks.
- Frameworks often pose a steep learning curve that must be overcome before they're useful.

Differences between framework and design pattern

Patterns and frameworks differ in three ways

1. ***Design patterns are more abstract than frameworks***

   ✓ Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code.

   ✓ A strength of frameworks is that they ca n be written down in programming languages and not only studied but executed and reused directly.

   ✓ Design patterns also explain the intent, trade-offs, and consequences of a design.

2. ***Design patterns are smaller architectural elements than frameworks***

   A typical framework contains several design patterns, but the reverse is never true.

3. Design patterns are less specialized than frameworks

- ✓ Frameworks always have a particular application domain.
- ✓ In contrast, the design patterns in this catalo g can be used in nearly any kind of application.

How to Select a Design Pattern

- ➢ Consider how design patterns solve design Problems
- ➢ Scan Intent sections
- ➢ Study how patterns interrelate
- ➢ Study patterns of like purpose
- ➢ Examine a Cause of redesign
- ➢ Consider what should be variable in the design

Consider how design patterns solve design problems.

Determine object granularity; specify object interfaces, and several other ways in which design patterns solve design problems.

Scan Intent sections

Read through each pattern's intent (purpose) to find one or more that should relevant to your problem.

Study how patterns interrelate

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

Study patterns of like purpose

Study only those patterns which are of specific purposes (creational patterns, structural patterns, and behavioural patterns).

Examine a cause of redesign.

Look at the patterns that help you avoid the causes of redesign

Consider what should be variable in your design.

Consider what you want to be able to change without redesign.

How to Use a Design Pattern

- Read the pattern once through for an overview.
- Go back and study the Structure, Participants and Collaborations sections.
- Look at the Sample Code section to see a concrete
- Example of the pattern in code.
- Choose names for pattern participants that are meaningful in the application context.
- Define the classes.
- Define Application-specific names for operations in the Pattern
- Implement the operations to carry out responsibilities and collaborations in the pattern.

1. **Read overview of pattern**

   Pay 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**

   Helps you learn how to implement the pattern.

4. **Choose names for pattern participants that are meaningful in the application context**

   It is useful to incorporate the participant name into the name that appears in the application.

5. **Define the classes**

   Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references.

6. **Define application-specific names for operations in the pattern**

   Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions

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.

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (87)	families of product objects
	Builder (97)	how a composite object gets created
	Factory Method (107)	subclass of object that is instantiated
	Prototype (117)	class of object that is instantiated
	Singleton (127)	the sole instance of a class
Structural	Adapter (139)	interface to an object
	Bridge (151)	implementation of an object
	Composite (163)	structure and composition of an object
	Decorator (175)	responsibilities of an object without subclassing
	Facade (185)	interface to a subsystem
	Flyweight (195)	storage costs of objects
	Proxy (207)	how an object is accessed; its location
Behavioral	Chain of Responsibility (223)	object that can fulfill a request
	Command (233)	when and how a request is fulfilled
	Interpreter (243)	grammar and interpretation of a language
	Iterator (257)	how an aggregate's elements are accessed, traversed
	Mediator (273)	how and which objects interact with each other
	Memento (283)	what private information is stored outside an object, and when
	Observer (293)	number of objects that depend on another object; how the dependent objects stay up to date
	State (305)	states of an object
	Strategy (315)	an algorithm
	Template Method (325)	steps of an algorithm
	Visitor (331)	operations that can be applied to object(s) without changing their class(es)

Table 1.2: Design aspects that design patterns let you vary

What is Object-Oriented Development?

First computers

- **First computers** are developed mainly to automate a well-defined process (i.e., an algorithm) for numerical computation, as systems became more complex, its effectiveness in developing solutions became suspect.
- software applications developed in later years had two differentiating characteristics:
  - ✓ Behavior that was hard to characterize as a process
  - ✓ Requirements of reliability, performance, and cost that the original developers did not face
- The 'process-centred' approach to software development used what is called top down functional decomposition.
  - ✓ The first step in such a design was to recognize what the process had to deliver which was followed by decomposition of the process into functional modules.
  - ✓ Structures to store data were defined and the computation was carried out by invoking the modules, which performed some computation on the stored data elements.
  - ✓ The life of a process-centred design was short because changes to the process specification required a change in the entire program.
  - ✓ This resulted in an inability to reuse existing code without considerable overhead
- Thus engineering disciplines started soon after, and the disciplines of 'software design' and 'software engineering' came into existence.
- The reasons for this success are easy to see:
  - ✓ Easily understandable designs
  - ✓ Similar (standard) solutions for a host of problems
  - ✓ An easily accessible and well-defined 'library' of 'building-blocks'
  - ✓ Interchangeability of components across systems,
  - ✓ A software component is also capable of storing data,
  - ✓ The components can also communicate with each other as needed to complete the process

Key Concepts of Object-Oriented Design

1. The Central Role of Objects
2. The notion of a Class
3. Abstract specification of functionality
4. A language to define the System
5. Standard Solutions
6. An analysis process to model a system
7. The notions of extendibility and adaptability

Other Related Concepts

Modular Design and Encapsulation

Modular Design

- Modularity refers to the idea of putting together a large system by developing a number of distinct components, independently and then integrating these to provide the required functionality.
- This approach is easier to understand than one that is designed as a monolithic structure. Such a design must be modular.
- The system's functionality must be provided by well-designed, Cooperating modules.
- Each module must perform functionality that is clearly specified by an interface.
- The interface also defines how other components may interact or communicate with the module.
- We would like that a module clearly specify what it does, but not expose its implementation. This separation of concerns gives rise to the notion of encapsulation,

Encapsulation

- Encapsulation, which means that the module hides details of its implementation from external agents. Example of applying encapsulation.
- The abstract data type (ADT), is generalization of primitive data types such as integers and characters.
- The programmer specifies the collection of operations on the data type and the data structures that are needed for data storage.
- Users of the ADT perform the operations without concerning themselves with the implementation.

Cohesion and Coupling

Cohesion

- Cohesion of a module tells us how well the entities within a module work together to provide functionality. Cohesion is a measure of how focused the responsibilities of a module are.
- If the responsibilities of a module are unrelated or varied and use different sets of data, cohesion is reduced.
- Highly cohesive modules tend to be more reliable, reusable, and understandable than less cohesive ones.
- In contrast, the worst approach would be to arbitrarily assign entities to modules, resulting in a module whose constituents have no obvious relationship.

Coupling

- Coupling refers to how modules are dependent on each other.
- The very fact that we split a program into multiple modules introduces some coupling into the system.
- Coupling could result because of several factors: a module may refer to variables defined in another module or a module may call methods of another module and use the return values.
- The amount of coupling between modules can vary.
- In general, if modules do not depend on each others implementation we say that the coupling is low
- Low coupling allows us to modify a module without worrying changes on the rest of the system.

- By contrast, high coupling means that changes in one module would necessitate changes in other modules, which may make it harder to understand the code.

Modifiability and Testability

Modifiability

- The modification in software can be done to change both functionality and design.
- The ability to change the functionality of a component allows for systems to be more adaptable;
- Improving the design through incremental change is accomplished by refactoring.
- In both cases, the organization of the system in terms of objects and classes has helped develop systematic procedures that mitigate the risk.

Testability

- Testability refers to both falsifiability, and ease with which we can find bugs in
- Software and the extent to which the structure of the system facilitates the detection of bugs.

Benefits and Drawbacks of the Paradigm

Advantages

1. Objects often reflect entities in application systems. This makes it easier for a designer to come up with classes in the design. In a process-oriented design, it is much harder to find such a connection that can simplify the initial design.
2. Object-orientation helps increase productivity through reuse of existing software. Inheritance makes it relatively easy to extend and modify functionality provided by a class. Language designers often supply extensive libraries that users can extend.
3. It is easier to accommodate changes. One of the difficulties with application development is changing requirements. With some care taken during design, it is possible to isolate the varying parts of a system into classes.
4. The ability to isolate changes, encapsulate data, and employ modularity reduces the risks involved in system development.

Drawbacks

1. Object creation and destruction is expensive.
2. Interactions of many objects are complex Example: Banking application, Video game that has often a large number of objects.
3. Objects tend to have complex associations, which can result in non-locality, leading to poor memory access times.
4. Programmers and designers schooled in other paradigms, usually in the imperative paradigm, find it difficult to learn and use object-oriented principles.
5. Programmers may need a year to start feeling comfortable with these concepts.
6. Some researchers are of the opinion that the programming environments also have not kept up with research in language capabilities.
7. Editors and testing and debugging facilities do not directly support many of the advances such as design patterns.

Analyzing a System

2.1 Overview of the Analysis Phase

The major goal of this phase is to address this basic question: what should the system do? Requirements are often simple and any clarifications can be had via questions in the classroom, e- mail messages, etc.

However, as in the case of the classroom assignment, there are still two parties: the user community, which needs some system to be built and the development people, who are assigned to do the work.

The process could be split into three activities:

1. Gather the requirements: this involves interviews of the user community, reading of any available documentation, etc.
2. Precisely document the functionality required of the system.
3. Develop a conceptual model of the system, listing the conceptual classes and their relationships.

It is not always the case that these activities occur in the order listed.

2.2 Stage 1: Gathering the Requirements

The purpose of *requirements analysis* is to define what the new system should do. Since the system will be built based on the information garnered in this step, any errors made in this stage will result in the implementation of a wrong system. Once the system is implemented, it is expensive to modify it to overcome the mistakes introduced in the analysis stage.

Imagine the scenario when you are asked to construct software for an application. The client may not always be clear in his/her mind as to what should be constructed.

First reason for this is that it is difficult to imagine the workings of a system that is not yet built.
Second reason Incompleteness and errors in specifications can also occur because the client does not have the technical skills to fully realize what technology can and cannot deliver
Third reason for omissions is that it is all too common to have a client who knows the system very well and consequently either assumes a lot of knowledge on the part of the analyst or simply skips over the 'obvious details'.

Requirements can be classified into two categories:

- ***Functional requirements:*** These describe the interaction between the system and its users, and between the system and any other systems, which may interact with the system by supplying or receiving data.
- ***Non-functional requirements:*** Any requirement that does not fall in the above category is a non-functional requirement. Such requirements include response time, usability and accuracy. Sometimes, there may be considerations that place restrictions on system development; these may include the use of specific hardware and software and budget and time constraints.

2.2.1 Case Study Introduction

Let us proceed under the assumption that developers of our library system have available to them a document that describes how the business is conducted. This functionality is described as a list of what are commonly called ***business processes.***

The business processes of the library system are listed below.

1 Add a member
2 Add books
3 Issue books
4 Return books\
5 Remove books
6 Place a hold on a book
7 Remove a hold on a book
8 Process Holds: Find the first member who has a hold on a book
9 Renew books
10 Print out a member's transactions
11 Store data on disk
12 Retrieve data from disk
13 Exit

In addition, the system must support **three** other requirements that are not directly related to the workings of a library, but, nonetheless, are essential.

- A command to save the data on a long-term basis.
- A command to load data from a long-term storage device.
- A command to quit the application. At this time, the system must ask the user if data is to be saved before termination.

  - A real library would have to perform additional operations like generating reports of various kinds, impose fines for late returns, etc.
  - Many libraries also allow users to check out books themselves without approaching a clerk.
  - Whatever the case may be, the analysts need to learn the existing system and the requirements. As mentioned earlier, they achieve this through interviews, surveys, and study.

2.3 Functional Requirements Specification

The requirements specification document serves as a contract between the users and the developers. we attempt to create a precise documentation of the requirements, we will discover errors and omissions. An accepted way of accomplishing this task is the use case analysis which we study now.

Use Case Analysis: It is a powerful technique that describes the kind of functionality that a user expects from the system.

- It is essentially a narrative describing the sequence of events (actions) of an external agent (actor) using the system to complete a process.

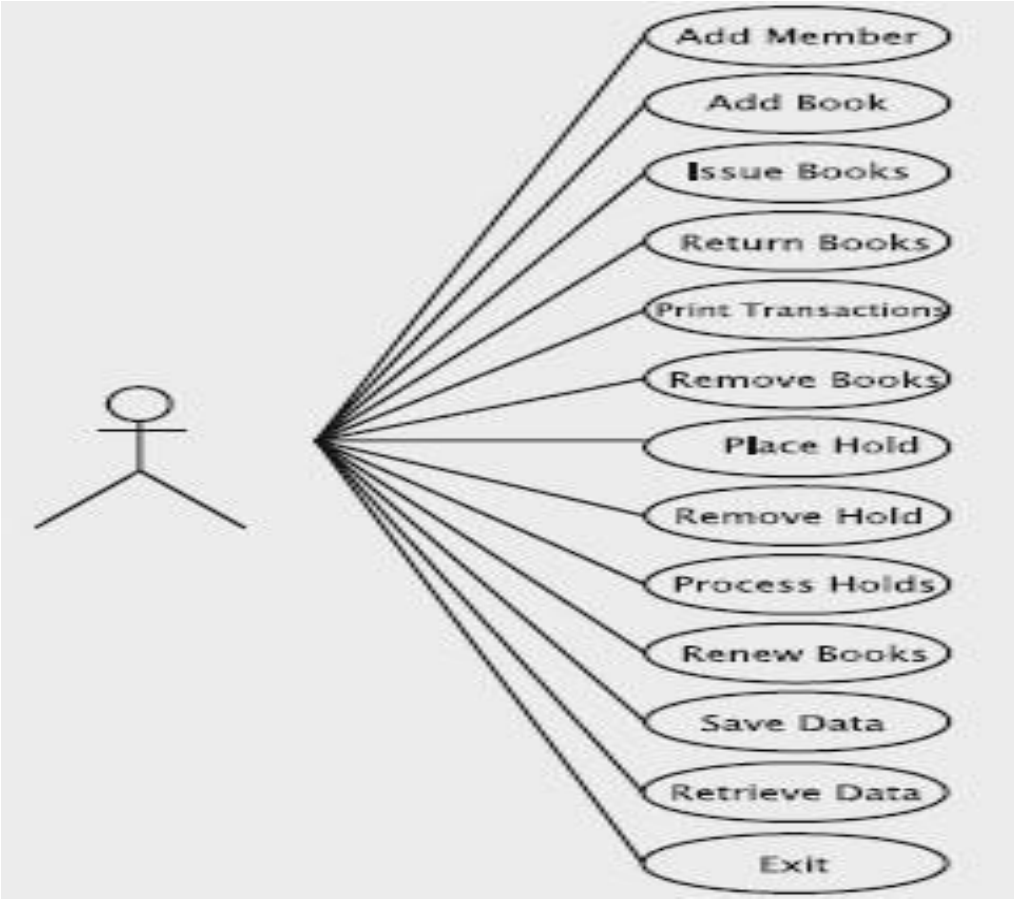
- It is a powerful technique that describes the kind of functionality that a user expects from the system.
- Use cases have two or more parties: *agents* who interact with the system and the *system* itself.

In our simple library system, the members do not use the system directly. Instead, they get services via the library staff.

To initiate this process, we need to get a feel for how the system will interact with the end-user.We assume that some kind of a user-interface is required, so that when the system is started, it provides a **menu with the following choices**:

1. Add a member
2. Add books
3. Issue books
4. Return books
5. Remove books
6. Place a hold on a book
7. Remove a hold on a book
8. Process Holds: Find the first member who has a hold on a book
9. Renew books
10. Print out a member's transactions
11. Store data on disk
12. Retrieve data from disk
13. Exit

Use case diagram for the library system



Use case for registering a user

Actions performed by the actor	Responses from the system
1. The customer fills out an application form containing the customer's name, address, and phone number and gives this to the clerk	
2. The clerk issues a request to add a new member	
	3. The system asks for data about the new member
4. The clerk enters the data into the system	
	5. Reads in data, and if the member can be added, generates an identification number (which is not necessarily a number in the literal sense just as social security numbers and phone numbers are not actually numbers) for the member and remembers information about the member. Informs the clerk if the member was added and outputs the member's name, address, phone and id
6. The clerk gives the user his identification number	

Steps:

1. Member will give the details of name, address, phone number to the clerk
2. Clerk process the request through the system
3. System asks the details of the customer to be registered.
4. Clerk enters the necessary information of the member into the system.
5. System check the details of the member and if the member is a valid person, then generates member identification number and display the necessary information at the output
6. Clerk provides the identification number to the user.

Use case for adding books

Actions performed by the actor	Responses from the system
1. Library receives a shipment of books from the publisher	
2. The clerk issues a request to add a new book	
	3. The system asks for the identifier, title, and author name of the book
4. The clerk generates the unique identifier, enters the identifier, title, and author name of a book	
	5. The system attempts to enter the information in the catalog and echoes to the clerk the title, author name, and id of the book. It then asks if the clerk wants to enter information about another book
6. The clerk answers in the affirmative or in the negative	
	7. If the answer is in the affirmative, the system goes to Step 3. Otherwise, it exits

Steps:

1 Library receives the information about the books shipped from the publisher.
2 Clerk receives a request to process the addition of books to the catalog.
3 System asks the details of the identifier, title, and author name of the book to be added.
4 Clerk generates the necessary information of the book to be added into the system.
5 System adds the details of the book and displays the necessary information of the book at the output and asks for any more books to be added.
6 If clerk replies affirmative, then same procedure is followed for the next set of books to be added. Otherwise system quits the application.

Use case for issuing books

Table 6.3 Use case Book Checkout

Actions performed by the actor	Responses from the system
1. The member arrives at the check-out counter with a set of books and supplies the clerk with his/her identification number	
2. The clerk issues a request to check out books	
	3. The system asks for the user ID
4. The clerk inputs the user ID to the system	
	5. The system asks for the ID of the book
6. The clerk inputs the ID of a book that the user wants to check out	
	7. The system records the book as having been issued to the member; it also records the member as having possession of the book. It generates a due-date. The system displays the book title and due-date and asks if there are any more books
8. The clerk stamps the due-date on the book and replies in the affirmative or negative	
	9. If there are more books, the system moves to Step 5; otherwise it exits
10. The customer collects the books and leaves the counter	

Steps:

1 Member gives the set of books with the member identification number to the clerk at the checkout counter and requests clerk to check out the books.
2 Clerk receives a request to check out the books and start checking in the system.
3 System asks the details of the user ID.
4 Clerk enters the user ID
5 System asks the details of the book ID.
6 Clerk enters the ID of the book to be checked out.
7 i) System checks whether the member possesses the book and generates a due date.
ii) System displays the book title, due date and asks if there is any more books to be processed
8 Clerk stamps due date on the book say yes, if there are books to be checked out. Otherwise no when there are no books to be processed.
9 If yes, system continues to process from step 5 and asks only for book ID since customer ID is same, otherwise system exits.
10 Customer collects the books and leave the checkout counter

Table 6.4 *Use case* Book Checkout *revised*

Actions performed by the actor	Responses from the system
1. The member arrives at the check-out counter with a set of books and supplies the clerk with his/her identification number.	
2. Clerk issues a request to check out books.	
	3. The system asks for the user id.
4. Clerk inputs the user ID to the system.	
	5. If the ID is valid, the system asks for the ID of the book; otherwise it prints an appropriate message and exits the use case.
6. The clerk inputs the identifier of a book that the user wants to check out.	
	7. If the ID is valid and the book is issuable to the member, the system records the book as having been issued to the member; It records the member as having possession of the book and generates a due-date as in *Rule 1*. It then displays the book's title and due-date. If the book is not issuable as per *Rule 2*, the system displays a suitable error message. The system asks if there are more books.
8. The clerk stamps the due-date, prints out the transaction (if needed) and replies positively or negatively.	
	9. If there are more books for checking out, the system goes back to Step 5; otherwise it exits.
10. The clerk stamps the due date and gives the user the books checked out. The customer leaves the counter.	

Steps:

1 Member gives the set of books with the member identification number to the clerk at the checkout counter and requests clerk to check out the books.
2 Clerk receives a request to check out the books and start checking in the system.
3 System asks the details of the user ID.
4 Clerk enters the user ID
5 System asks the details of the book ID.
6 Clerk enters the ID of the book to be checked out.
7 i) System checks whether the member is a valid person or not and then records the member has a possession on the book and generates a due date based on the result of Rule 1.
ii) System displays the book title and due date.
iii) The system displays error message, if the Rule 2 is not satisfied and asks if there is any more books to be processed.
8 i) Clerk stamps due date on the book and take the print out of the transactions, if the user is requesting for print out.
ii) Clerk says yes, if there are books to be checked out. Otherwise no when there are no books to be processed.
9 If yes, system continues to process from step 5 and asks only for book ID since customer ID is same, otherwise system exits.
10 Customer collects the books and leave the checkout counter

Use case Return Book

Table 6.5 *Use case* Return Book

Actions performed by the actor	Responses from the system
1. The member arrives at the return counter with a set of books and leaves them on the clerk's desk.	
2. The clerk issues a request to return books.	
	3.The system asks for the identifier of the book.
4. The clerk enters the book identifier.	
	5. If the identifier is valid, the system marks that the book has been returned and informs the clerk if there is a hold placed on the book; otherwise it notifies the clerk that the identifier is not valid. It then asks if the clerk wants to process the return of another book.
6. The clerk answers in the affirmative or in the negative and sets the book aside in case there is a hold on the book. (See Rule 5.)	
	7. If the answer is in the affirmative, the system goes to Step 3. Otherwise, it exits.

Steps:

1. Member gives the set of books to be returned to the clerk at the checkout counter.
2. On receiving the request, clerk process the return books request in the system.
3. System asks the details of the book ID.
4. Clerk enters the ID of the book to be returned.
5. i) System checks whether the book is valid or not and then records that book is returned.
   ii) System informs the clerk about the deadline of the book to be returned and asks clerk that if any more books are to be returned.
6. i) Clerk says yes, if there are books to be returned, Otherwise no when there are no books to be returned.
   ii) Clerk checks the deadline of the book to be returned based on Rule 5.
7. If yes, system continues to process from step 3, otherwise system exits.

Use case Removing Books

Table 6.6 *Use case* Removing Books

Actions performed by the actor	Responses from the system
1. Librarian identifies the books to be deleted.	
2. The clerk issues a request to delete books.	
	3. The system asks for the identifier of the book.
4. The clerk enters the ID for the book.	
	5. The system checks if the book can be removed using *Rule 3*. If the book can be removed, the system marks the book as no longer in the library's catalog. The system informs the clerk about the success of the deletion operation. It then asks if the clerk wants to delete another book.
6. The clerk answers in the affirmative or in the negative.	
	7. If the answer is in the affirmative, the system goes to Step 3. Otherwise, it exits.

Steps:

1 Librarian gives the list of books to be deleted for the clerk.
2 On receiving the list, the clerk starts processing the deletion of books.
3 The system asks ID of the book.
4 The clerk enters the ID of the book.
5 i) The system checks whether the book can be removed as per Rule 3 by verifying the check out and deadline of the book.
ii) If the book ID is valid, the system removes the book ID from the library's catalog.
iii) The system displays the success of the deletion operation to the clerk.
iv) System asks the clerk that, there are any more books are to be deleted.
6 The clerk answers yes, if there are books to be processed or no, when there are no books to be processed.
7 If the answer is yes, then the system goes to Step 3 Otherwise, it exits

Use case Member Transactions

Table 6.7 *Use case* Member Transactions

Actions performed by the actor	Responses from the system
1. The clerk issues a request to get member transactions.	
	2. The system asks for the user ID of the member and the date for which the transactions are needed.
3. The clerk enters the identity of the user and the date.	
	4. If the ID is valid, the system outputs information about all transactions completed by the user on the given date. For each transaction, it shows the type of transaction (book borrowed, book returned or hold placed) and the title of the book.
5. Clerk prints out the transactions and hands them to the user.	

Steps:

1 The clerk receives a request from the user to give member transactions
2 The system asks for the user ID of the member and the date for which the transactions are required.
3 The clerk enters the ID of the user and the date of the transactions required
4 If the ID is valid, the system outputs information about all transactions completed by the user on the given date along with the details of book borrowed, book returned, hold placed and the title of the book.
5 Clerk prints out the transactions and hands them to the user

Use case Place a Hold and Remove a Hold

Table 6.8 *Use case* Place a Hold

Actions performed by the actor	Responses from the system
1. The clerk issues a request to place a hold.	
	2. The system asks for the book's ID, the ID of the member, and the duration of the hold.
3. The clerk enters the identity of the user, the identity of the book and the duration.	
	4. The system checks that the user and book identifiers are valid and that Rule 6 is satisfied. If yes, it records that the user has a hold on the book and displays that; otherwise, it outputs an appropriate error message.

Table 6.9 *Use case* Remove a Hold

Actions performed by the actor	Responses from the system
1. The clerk issues a request to remove a hold.	
	2. The system asks for the book's ID and the ID of the member.
3. The clerk enters the identity of the user and the identity of the book.	
	4. The system removes the hold that the user has on the book (if any such hold exists), prints a confirmation and exits.

Steps:

1 On receiving the request, clerk start processing to place a hold
2 The system asks details of the book such as book ID, the ID of the member and the duration of the hold
3 The clerk enters all the necessary details.
4 i) The system checks that the user and book ID's whether it is valid or not as per Rule 6
ii) If Rule 6 is satisfied, then the system records that the user has a hold on the book and displays that; otherwise, it outputs an appropriate error message

Steps:

1 On receiving the request, clerk start processing to remove a hold
2 The system asks details of the book such as book's ID and the ID of the member
3 The clerk enters the ID of the user and ID of the book
4 The system removes the hold that the user has on the book, prints a confirmation and exits

Use case Process Holds

Table 6.10 *Use case* Process Holds

Actions performed by the actor	Responses from the system
1. The clerk issues a request to process holds (so that Rule 5 can be satisfied).	
	2. The system asks for the book's ID.
3. The clerk enters the ID of the book.	
	4. The system returns the name and phone number of the first member with an unexpired hold on the book. If all holds have expired, the system responds that there is no hold. The system then asks if there are any more books to be processed.
5. If there is no hold, the book is then shelved back to its designated location in the library. Otherwise, the clerk prints out the information, places it in the book and replies in the affirmative or negative.	
	6. If the answer is yes, the system goes to Step 2; otherwise it exits.

Steps:

1 On receiving the request, clerk start processing to place a hold as per Rule 5 by notifying the member who crosses the deadline.
2 The system asks book ID
3 The clerk enters book ID
4 i) The system checks for the hold whether it is expired or not
ii) If yes, the system records that there is no hold and ask for next books to be processed
5 i) If there is no hold, the book is then kept back to its designated location in the library and no notification generated.
ii) Clerk replies the system yes or no for the next books to be processed
6 If the answer is yes, the system goes to Step 2; otherwise it exits

Use case Renew Books

Table 6.11 *Use case* Renew Books

Actions performed by the actor	Responses from the system
1. Member makes a request to renew several of the books that he/she has currently checked out.	
2. Clerk issues a request to renew books.	
	3. System asks for the member's ID.
4. The clerk enters the ID into the system.	
	5. System checks the member's record to find out which books the member has checked out. If there are none, the system prints an appropriate message and exits; otherwise it moves to Step 6.
	6. The system displays the title of the next book checked out to the member and asks whether the book should be renewed.
7. The clerk replies yes or no.	
	8. The system attempts to renew the book using *Rule 4* and reports the result. If the system has displayed all checked-out books, it reports that and exits; otherwise the system goes to Step 6.

Steps:

1 Member requests for the renew of the books
2 On receiving the request, clerk start processing the renew of books in the system
3 System asks for the member's ID
4 The clerk enters the member ID into the system
5 i) System checks the record to find out which book is availed by the member
ii) If there are none, the system prints an appropriate message and exits; otherwise it moves to Step 6
6 The system displays the title of the next book to be renewed
7 The clerk replies yes or no
8 i) The system renews the book based on Rule 4 by checking holds on the book and reports the result.
ii) If the system has displayed all checked-out books, it reports that and exits; otherwise the system goes to Step 6

Different Rules for the Library System

Rule number	Rule
Rule 1	Due-date for a book is one month from the date of issue
Rule 2	All books are issuable
Rule 3	A book is removable if it is not checked out and if it has no holds
Rule 4	A book is renewable if it has no holds on it
Rule 5	When a book with a hold is returned, the appropriate member will be notified
Rule 6	Holds can be placed only on books that are currently checked out

Guidelines to write use cases

- A use case must provide something of value to an actor or to the business.
- Use case should be functionally cohesive, i.e., they encapsulate a single service that the system provides.
- Use case should be temporally cohesive. This notion applies to the time frame over which the use case occurs.
- If a system has multiple actors, each actor must be involved in at least one, and typically several use cases.
- The model that we construct is a set of use cases.
- Use cases are written from the point of view of the actor.
- A use case describes a scenario.
- Use cases change over the course of system analysis.

Defining Conceptual Classes and Relationships

The last major step in the analysis phase involves the determination of the conceptual classes and the establishment of their relationships. Example, in the library system, some of the major conceptual classes include members and books. Members borrow books, which establish a relationship between them.

1 Design facilitation: Via use case analysis, we determine the functional requirement of the system. Obviously, the design stage must determine how to implement the functionality. For this, the designers should be in a position to determine the classes that need to be defined, the objects to be created, and how the objects interact. This is better facilitated if the analysis phase classifies the entities in the application and determines their relationships.

2 Added knowledge: The use cases do not completely specify the system. Some of these missing details can be filled in by the class diagram.

3 Error reduction: In carrying out this step, the analysts are forced to look at the system more carefully. The result can be shown to the client who can verify its correctness.

4 Useful documentation: The classes and relationships provide a quick introduction to the system for someone who wants to learn it. Such people can join the project to carry out the design or implementation or subsequent maintenance of the system.

While using this approach, we must remember that natural languages are imprecise and that synonyms may be found. We can eliminate the others as follows:

- Customer: Becomes a member, so it is effectively a synonym for member.
- User: The library refers to members alternatively as users, so this is also a synonym.
- Application form and request: Application form is an external construct for gathering information, and request is just a menu item, so neither actually becomes part of the data structures. Customer's name, address, and phone number: They are attributes of a customer, so the Member class will have them as fields.
- Clerk: An agent for facilitating the functioning of the library, so it has no software representation.
- Identification number: Become a part of a member.
- Data: Gets stored as a member.
- Information: Same as data related to a member.
- System: Refers to the collection of all classes and software.

UML Diagram
(Unified Modeling Language Diagram)

Figure 2.2 UML diagram for the class library

- In the above figure, system implies a conceptual class that represents all of the system.
- This class is Library UML without any attributes and methods.

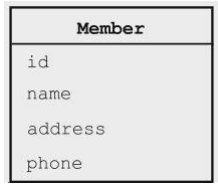


Figure 2.3 UML diagram for the class Member

- The UML convention is to write the class name at the top with a line below it and the attributes listed just below that line.

Figure 2.4 UML diagram showing the association of Library and Member

- An association between the conceptual classes Library and Member.
- The line between the two classes and the labels 1, *, and 'maintains a collection of' just above it.
- There is only one instance of the Library that maintains a collection of zero or more members.

Figure 2.5 UML diagram for the class Book

Figure 2.6 UML diagram showing the association of Library and Book

Figure 2.7 UML diagram showing the association Borrows between Member and Book

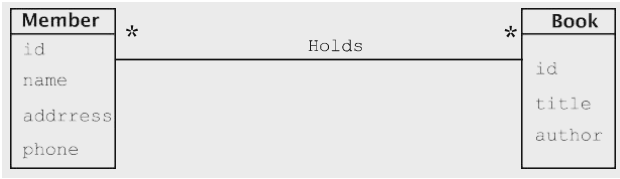


Figure.2.8 UML diagram showing the association Holds between Member and Book

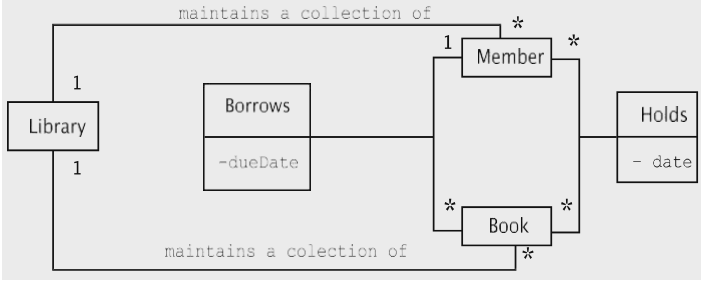


Figure 2.9 Conceptual classes and their associations

All the conceptual classes and their associations are captured into a single diagram. To reduce complexity, attributes of the Library, Member, and Book are omitted. As seen before, a relationship formed between two entities is sometimes accompanied by additional information. This additional information is relevant only in the context of the relationship.

2.4 Using the Knowledge of the Domain

- Domain analysis is the process of analysing related application systems in a domain so as to discover what features are common between them and which parts are variable. Thus, one of the goals of this approach is reuse.
- Any area in which we develop software systems qualifies to be a domain.
- Examples include library systems, hotel reservation systems, university registration systems, etc. It is possible to divide a domain into several interrelated domains.
- Where does the knowledge of a specific domain come from? It could be from sources such as surveys, existing applications, technical reports, user manuals, and so on.
- A domain analyst analyses this knowledge to come up with Specifications, designs, and code that can be reuse in multiple projects

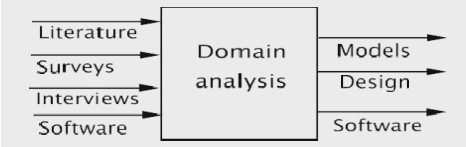


Fig. 2.10 Domain analysis

2.5 Design and Implementation

The main UML tool employed here is the sequence diagram. In a sequence diagram, the designer specifies the details of how the behaviour specified in the model will be realized. This process requires the system's actions to be broken down into specific tasks, and the responsibility for these tasks to be assigned to the various players in the system.

Design
During the design process, a number of questions need to be answered:

1 On what platform(s) (hardware and software) will the system run?
2 What languages and programming paradigms will be used for implementation?
3 What user interfaces will the system provide? These include GUI screens, printouts, and other devices.
4 What classes and interfaces need to be coded? What are their responsibilities?
5 How is data stored on a permanent basis? What medium will be used? What model will be used for data storage?
6 What happens if there is a failure? Ideally, we would like to prevent data loss and corruption. What mechanisms are needed for realizing this?

7 Will the system use multiple computers? If so, what are the issues related to data and code distribution?

8 What kind of protection mechanisms will the system use?

Major subsystems
The first step in our design process is to identify the major subsystems. We can view the library system as composed of two major subsystems:

1 Business logic: This part deals with input data processing, data creation, queries, and data updates. This module will also be responsible for interacting with external storage, storing and retrieving data.
2 User interface: This subsystem interacts with the user, accepting and outputting information. It is important to design the system such that the above parts are separated from each other so that they can be varied independently.

Creating the Software Classes

The next step is to create the software classes. During the analysis, after defining the use case model, We came up with a set of conceptual classes and a conceptual class diagram for the entire system.
In this phase there are two major activities.

1. Come up with a set of classes.
2. Assign responsibilities to the classes and determine the necessary data structures and methods.

In general, it is unlikely that we can come up with a design simply by doing these activities exactly once. Several iterations may be needed and classes may need to be added, split, combined, or eliminated.

Member and Book

- ✓ Each Member object comprises several attributes such as name and address, stays in the system for a long period of time and performs a number of useful functions.
- ✓ Books stay part of the library over a long time and we can do a number of useful actions on them. We need to instantiate books and members quite often. Clearly, both are classes that require representation in software.

Library: Do we really need to make a class for this? To answer the question, let us ask what real library. When a member thinks of a library, he/she thinks of borrowing and returning books, placing and removing holds, i.e., the functionality provided by the library.
Borrows: This class represents the one-to-many relationship between members and books. In typical one-to-many relationships, the association class can be efficiently implemented as a part of the two classes at the two ends.
Holds: Unlike Borrows, this class denotes a many-to-many relationship between the Member and Book classes. *In typical many-to-many relationships,* implementation of the association without using an additional class is unlikely to be clean and efficient.

Assigning Responsibilities to the Classes

- Having decided on an adequate set of software classes, our next task is to assign responsibilities to these. Since the ultimate purpose of these classes is to enable the system to meet the responsibilities specified in the use case, we shall work with these system responsibilities to find the class responsibilities.

- **Sequence diagrams**
  - Describe interactions among classes in terms of an exchange of messages over time.

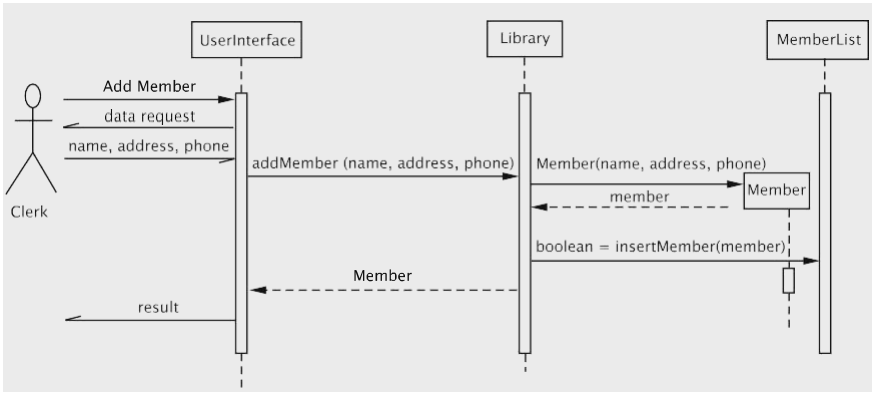


Figure. 2.11 Sequence diagram for adding a new member

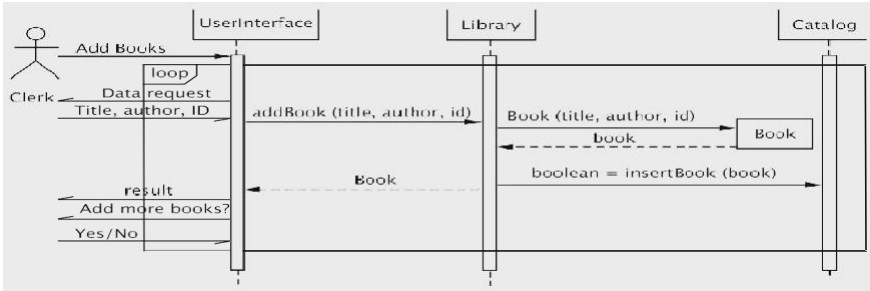


Figure. 2.12 Sequence diagram for adding books

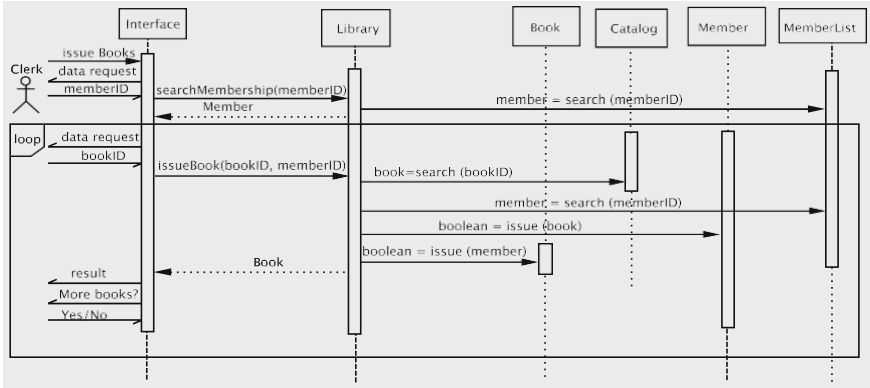


Figure. 2.13 Sequence diagram for issuing books

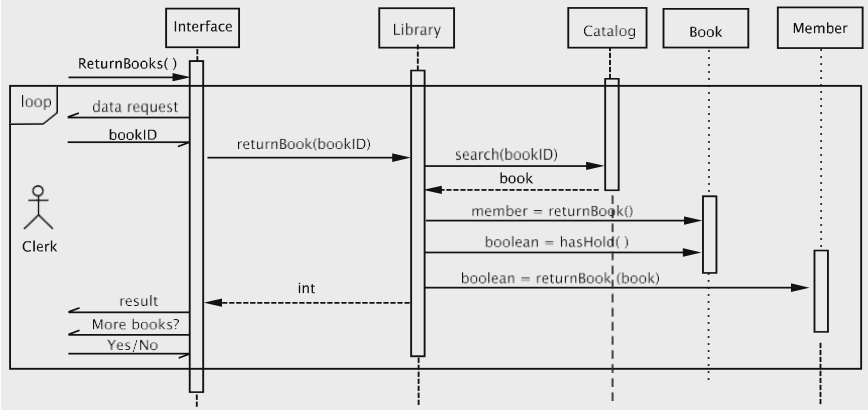


Figure. 2.14 Sequence diagram for returning books

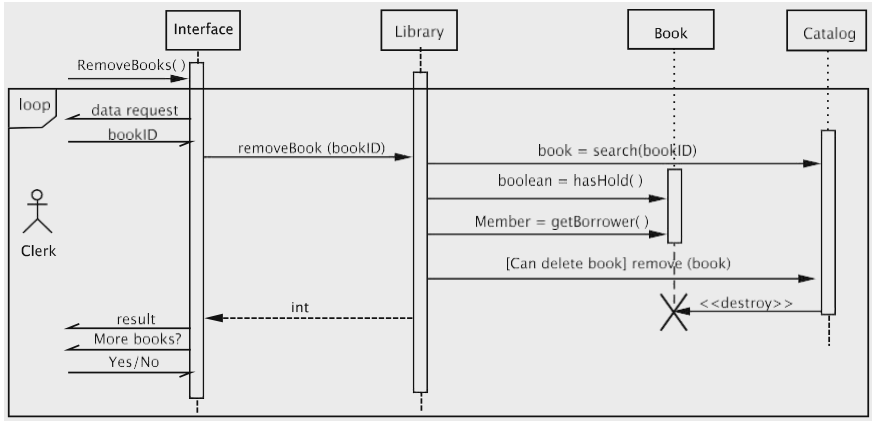


Figure. 2.15 Sequence diagram for removing books

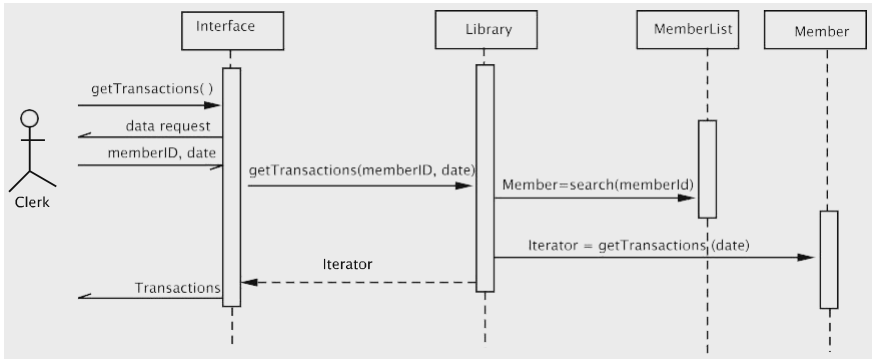


Figure. 2.16 Sequence diagram for printing a member's transactions

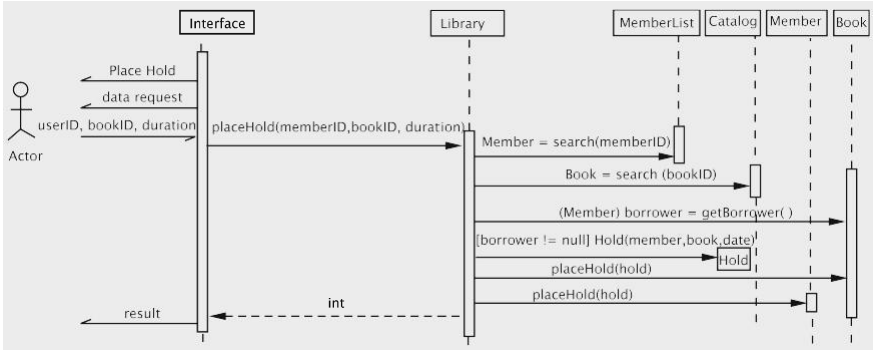


Figure.2.17 Sequence diagram for placing a hold

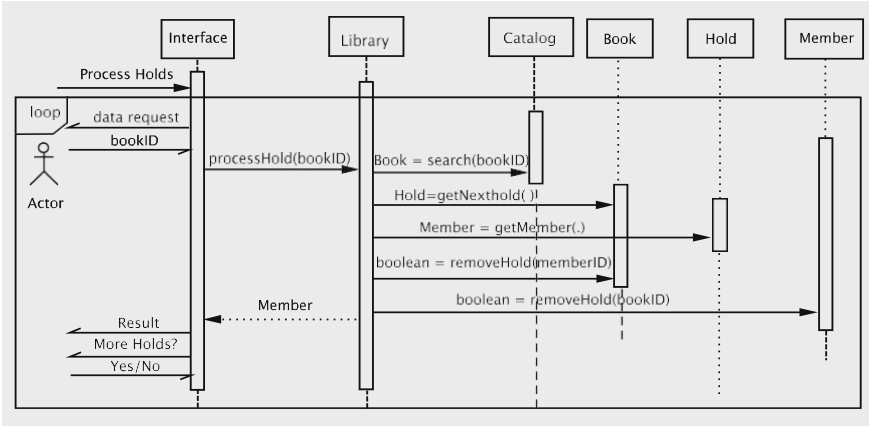
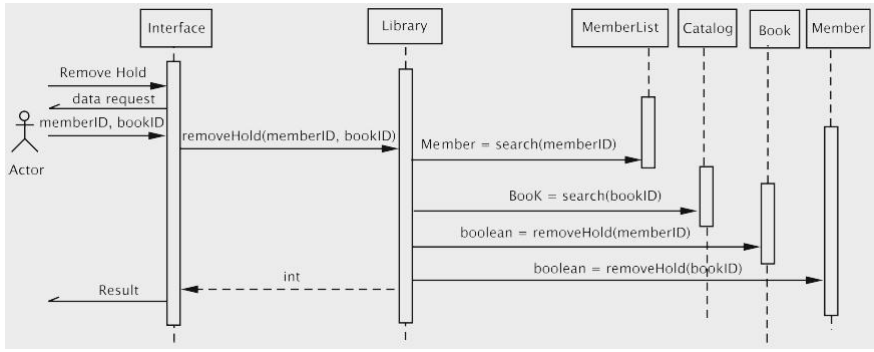


Figure. 2.18 Sequence diagram for processing holds



.

Figure. 2.19 Sequence diagram for removing a holds

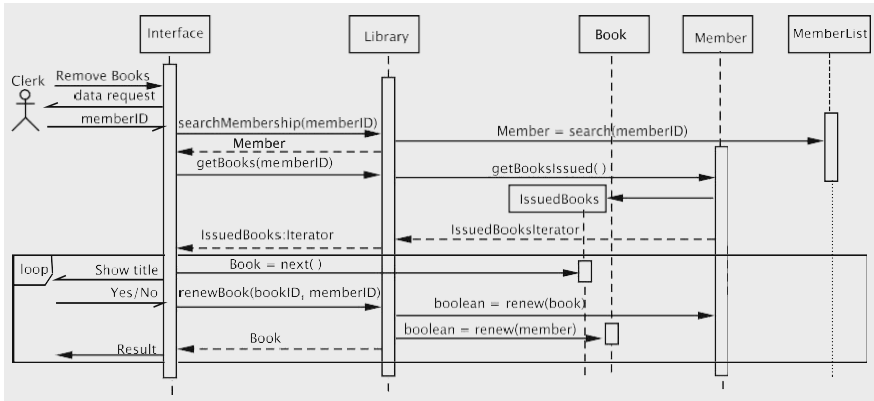


Figure. 2.20 Sequence diagram for renewing books

2.6.1.1 Class Diagrams

Hopefully, at this stage, we have come up with all the software classes. To review:

1. Library
2. MemberList
3. Catalog
4. Member
5. Book
6. Hold
7. Transaction

- The relationships between these classes are shown in Figure.
- Note that Hold is not shown as an Association class, but an independent class that connects Member and Book.
- The new class Transactions added to record transactions; this has a dependency on Book since it stores the title of the book.

Class Diagram for Library

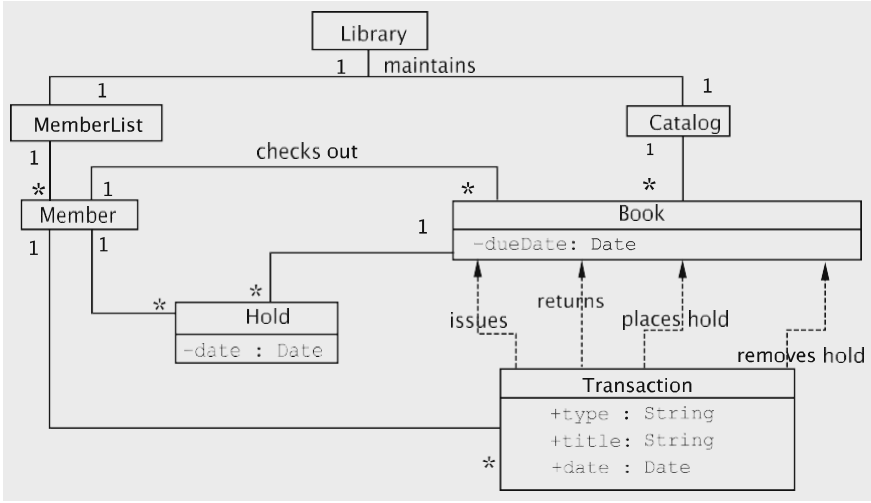


Figure. 2.21 Relationships between the software classes

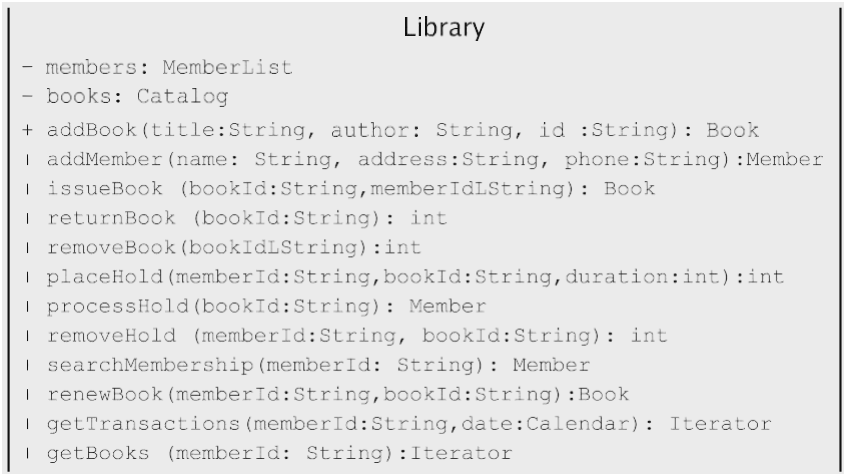


Figure. 2.22 Class diagram for Library

Class Diagram for Member

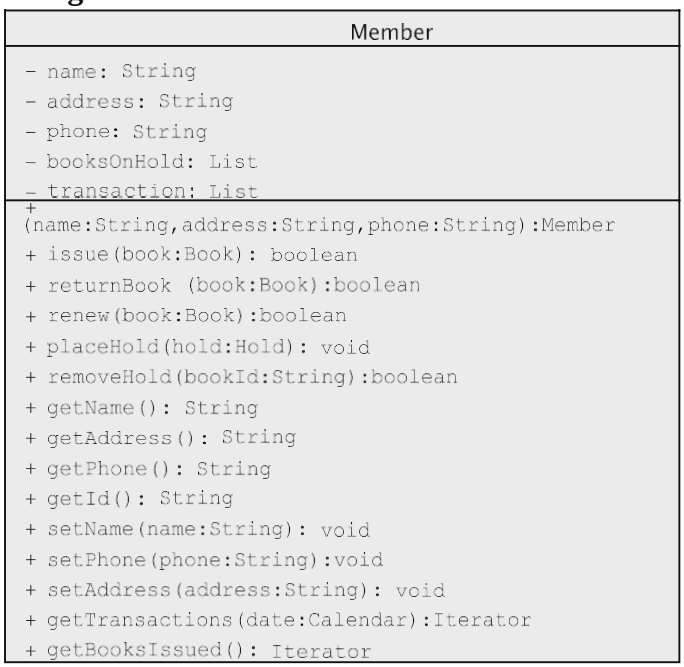
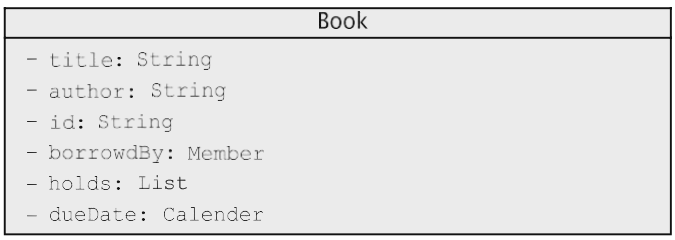


Figure. 2.23 Class diagram for Member

Class Diagram for Book



Class Diagram for Catalog

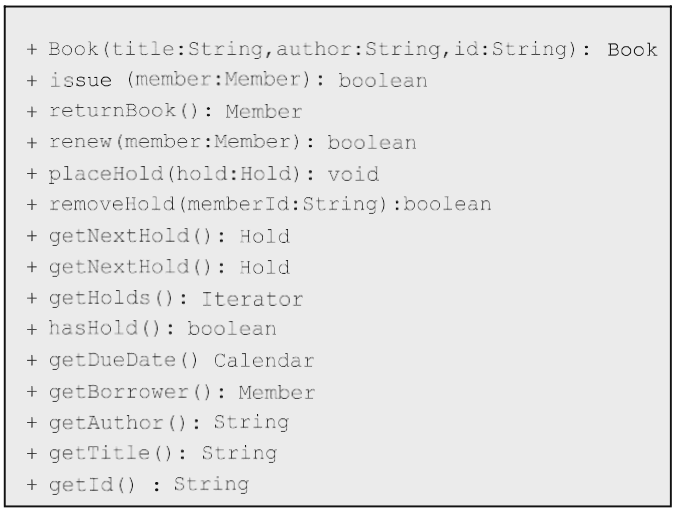


Figure. 2.24 Class diagram for the Book class

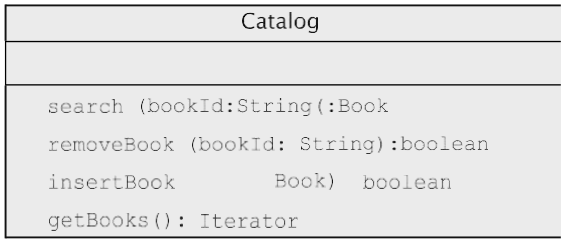


Figure.2.25 Class diagram for Catalog class

Class Diagram for MemberList

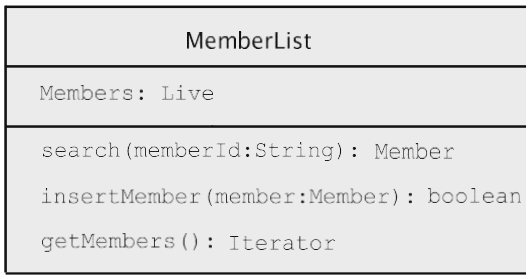


Figure. 2.26 Class diagram for the MemberList class

Class Diagram for Hold

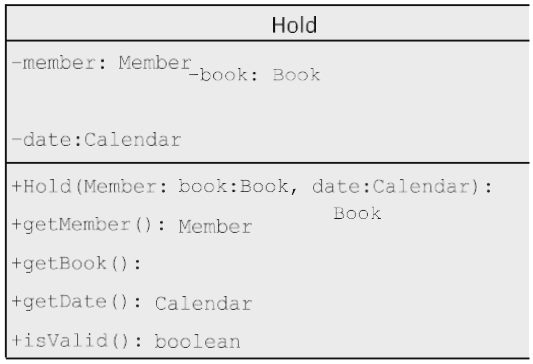


Figure.2.27 Class diagram for Hold

Class Diagram for Transaction

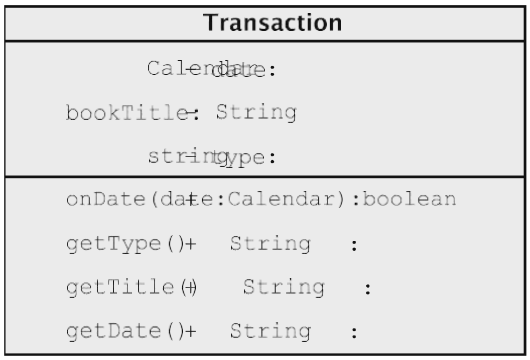


Figure 2.28 Class diagram for Transaction

User Interface

As discussed earlier, our UI provides a menu with the following options:

1 Add a member
2 Add books
3 Issue books 4 Return books
4 Renew books
5 Remove books
6 Place a hold on a book
7 Remove a hold on a book
8 Process holds
9 Print a member's transactions on a given date
10Save data for long-term storage
11Retrieve data fromstorage
12Help
13Exit

2.6.1.2 Data Storage

Following commands in our UI

1. A command to save the data on a long-term basis.
2. A command to load data from a long-term storage device.
    - When the first command is executed, we will copy all of the data onto secondary storage. Similarly,
    - when the second command is executed, the data stored on the storage device is copied to recreate the object.

2.6.2 Implementing Our Design

1 In this phase, we code, test, and debug the classes that implement the business logic (Library, Book, etc.) and UserInterface.

2 An important issue in the implementation is the communication via the return values between the different classes: in particular between Library and UserInterface.

Adding New Books

The addBooksmethod in UserInterfaceis shown below:

```
public void addBooks() { Book result;
  do {
    String title = getToken("Enter book title"); String author = getToken("Enter author");
    String bookID = getToken("Enter id");
    result = library.addBook(title, author, bookID); if (result != null) {
      System.out.println(result);
    } else {
      System.out.println("Book could not be added");
    }
    if (!yesOrNo("Add more books?")) {
      break;
    }
  } while (true);
}
```

Issuing Books

```
public void issueBooks() { Book result;
  String memberID = getToken("Enter member id"); if
  (library.searchMembership(memberID) == null) {
    System.out.println("No such member"); return;
  }
  do {
    String bookID = getToken("Enter book id"); result = library.issueBook(memberID,
    bookID); if (result != null){
      System.out.println(result.getTitle()+ "          " + result.getDueDate());
    } else {
        System.out.println("Book could not be issued");
    }
    if (!yesOrNo("Issue more books?")) {

      break;
    }
  } while (true);
}
```

The issueBookmethod in Librarydoes the necessary processing and returns a reference to the issued book.

```
public Book issueBook(String memberId, String bookId) { Book book = catalog.search(bookId);
  if (book == null) { return(null);
  }
  if (book.getBorrower() != null) { return(null);
  }
  Member member = memberList.search(memberId); if (member == null) {
    return(null);
  }
  if (!(book.issue(member) && member.issue(book))) { return null;
  }
  return(book);
}
```

Printing Transactions

```
public void getTransactions() { Iterator result;
  String memberID = getToken("Enter member id");
  Calendar date = getDate("Please enter the date for which you want " +
                          "records as mm/dd/yy"); result =
  library.getTransactions(memberID,date); if (result == null) {
    System.out.println ("Invalid Member ID");
  } else {while (result.hasNext ()) {
      Transaction transaction = (Transaction) result.next (); System.out.println (transaction.getType () + "
                                                                  " +
                          transaction.getTitle () + "\n");
    }
    System.out.println ("\n There are no more transactions \n");
  }
}
```

2.6.2.1 Placing and Processing Holds

```
public void placeHold(Hold hold) {
  transactions.add(new Transaction ("Hold Placed", hold.getBook().getTitle()
));  booksOnHold.add(hold);
}
```

Toprocess a hold, Libraryinvokes the getNextHold method in Book, which returns the first valid hold.

```
public Hold getNextHold() {
  for (ListIterator iterator = holds.listIterator(); iterator.hasNext();) {Hold hold = (Hold) iterator.next ();
    iterator.remove(); if (hold.isValid()) {
      return hold;
    }
  }
  return null;
}
```

2.6.2.2 Storing and Retrieving the Library Object

Java Serialization

- Our approach to long-term storage of the library data uses the Java serialization mechanism.
- In our Current example, Book and Hold can be serialized by simply declaring them to be Serializable.

Storing the Data

- Library has references to both the Catalog and member List objects, which in turn have references to the Book and Member objects respectively.
- The Hold objects are referred by the Book objects and the Member objects.
- Thus, if we simply store the Library object, all of the data will be stored.

Maintaining the Singleton Property

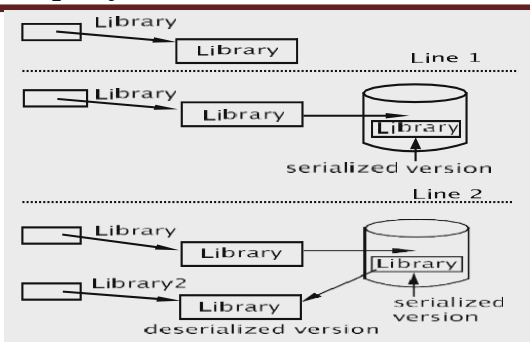


Figure.2.29 A pitfall in using serialization with a singleton

- The Library, MemberList and Catalog objects are singletons: they cannot have more than one instance. Using the serialization mechanism, *it is now possible to serialize an object and then deserialize it to get a second instance.*
- Library library = Library.instance(); Serialize library onto a disk file "library1";
- Library library2 = Deserialized version of "library1"; Update library (add a member);
- Update library2 (delete a book);

Dealing with Static Fields in Non-singletons

- The static fields in non-singletons pose a different challenge. Since the static field idCounter in Member stores the value that is used to generate the ID for each new member, this value must be saved along with the library. Since static fields are not serialized, this value will have to be explicitly written in the write Object method of Member.
- One simple solution to this is to circumvent the problem by encapsulating the static field as a separate class. The singleton Member Id Server, shown below, holds the idCounter and also increments it each time getId is invoked.

```
class MemberIdServer implements Serializable
{ private int idCounter;
private static MemberIdServer server; private MemberIdServer()
{
        idCounter = 1;
}
public static MemberIdServer instance()
  {
         if (server == null)
                {
                        return (server = new MemberIdServer());
                }
        else
                {
                        return server;
                }
  }
public int getId() { return idCounter++;
}/
        / other code not shown
}
```

2.6 Discussion and Further Reading

Converting the model into a working design is the most complex part of the software design process.
The sequence of topics so far suggests that the design would progress linearly from analysis to design to implementation.

- **Conceptual, Software and Implementation Classes:**

Finding the classes is a critical step in the object-oriented methodology. In the analysis phase, we found the conceptual classes. These correspond to real- world concepts or things, and present us with a conceptual or essential perspective.
As we go further into the design process and construct the sequence diagrams, we need to deal with the issue of these conceptual classes. we are now dealing with **software** classes.
The last step is the implementation class, which is a class created using a specific programming language such as Java or C++. The last step is the implementation class, which is a class created using a specific programming language such as Java or C++.

- ***Building a Commercially Acceptable System:***
  - **Non-functional Requirements:** Some issues like portability are automatically resolved since Java is interpreted and is thus platform independent. Response time (run-time performance) is a sticking point for object-oriented applications.
  - **Functional Requirements:** It can be argued that for a system to be accepted commercially, it must provide a sufficiently large set of services, like –
    - ✓ *Additional features can be easily added*
    - ✓ *Allowing for variability among kinds of books/members*
    - ✓ *Having a more sophisticated interface*
    - ✓ *Allowing remote access*

- ***The Facade Pattern:*** Library class that provided a set of methods for the interface and thus served as a single point of entry to and exit from the business logic module. In the language of design patterns, what we created is known as a **facade.**
  - The primary motivation behind using a façade is to reduce the complexity by minimizing communication and dependencies between a subsystem and its clients . The facade not only shields the client from the complexity but also enables loose coupling between the subsystem and its clients. Facades are not typically designed to prevent the client from accessing the components within the subsystem.

- **Using a Facade**

Where do we employ this? A situation in which we have:

1. A system with several individual classes, each with its own set of public methods.
2. An external entity interacting with the system requires knowledge of the public methods of several classes.

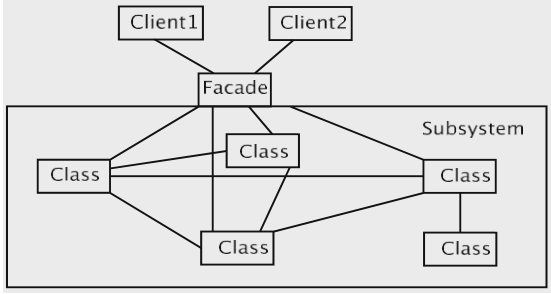


Figure 2.30 Structure diagram for facade

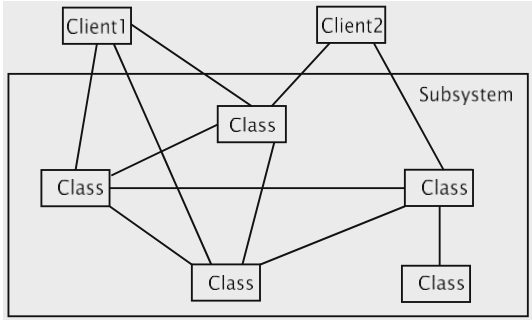


Figure 2.31 Interactions with a subsystem without a facade

Note: Explanation for sequence diagrams can be referred from use case analysis.

MODULE 2
Structural Patterns

- ✓ Structural patterns are concerned with how classes and objects are composed to form larger structures.
- ✓ Structural class patterns use inheritance to compose interfaces or implementations.

Example: Multiple inheritance mixes two or more classes into one.

Popular structural design patterns include**:**

1. Adaptor
2. Bridge
3. Composite
4. Decorator
5. Façade
6. Flyweight
7. Proxy

ADAPTOR

It is both class and object structural pattern.

Intent: To convert the interface of one class into another interface that the client expects. Adapter pattern allows two incompatible classes to communicate with one another.

Also knows as: Wrapper

Motivation:

- ✓ Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams.
  - ➢ The interface for graphical objects is defined by an abstract class called **Shape.**
  - ➢ The editor defines a subclass of Shape:
    - a LineShape class for lines ,
    - a PolygonShape class for polygons, and so forth.
- ✓ Geometric shapes like LineShape and PolygonShape are easy to implement, because their drawing and editing capabilities are inherently limited. But a TextShape subclass is difficult to implement, since even basic text editing involves :
  - ➢ complicated screen update and
  - ➢ buffer management.
- ✓ We can reuse TextView to implement TextShape, but the toolkit wasn't designed with Shape classes in mind. So we can't use TextView and Shape objects interchangeably.

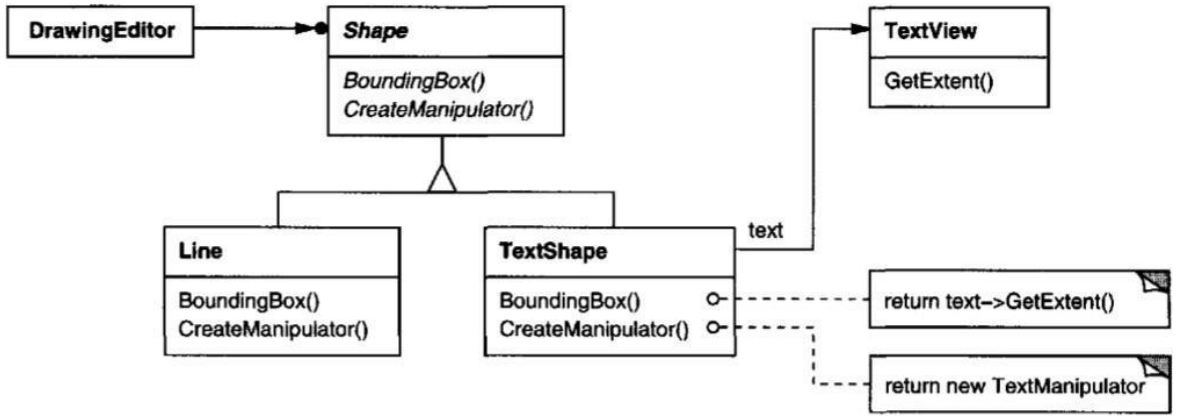
- How can existing and unrelated classes like TextView work in an application that expects classes with a different and incompatible interface?

- We could change the TextView clas s so that it conforms to the Shape interface, but that isn't an option unless we have the toolkit's source code . Even if we did, it wouldn't make sense to change TextView; the toolkit shouldn't have to adopt domain-specific interfaces just to make one application work.
- Instead, we could define TextShape so that it adapts the TextView interface to Shape's.

We can do this in one of two ways:

(1) by inheriting Shape's interface and Text View's implementation or

(2) by composing a TextView instance within a TextShape and implementing TextShape in terms of Text View's interface.



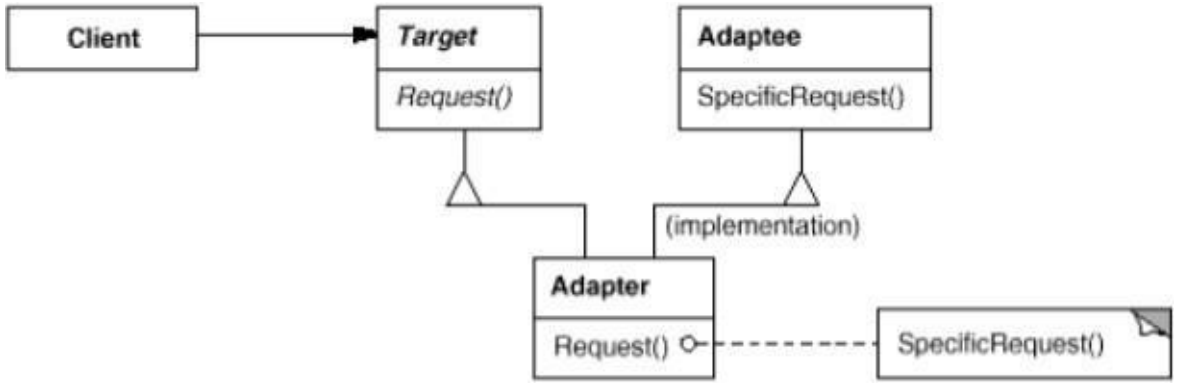
The above diagram illustrates the object adapter case .

- It shows BoundingBox requests, declared in class Shape , are converted to GetExtent requests defined in Text View .
- Since TextShape adapts Text View to the Shape interface, the drawing editor can reuse the otherwise incompatible Text View class.
- CreateManipulator operation, which returns an instance of the appropriate Manipulator subclass.
  - Manipulator is an abstract clas s for objects that know how to animate a Shape in response to user input, lik e dragging the shape to a new location.

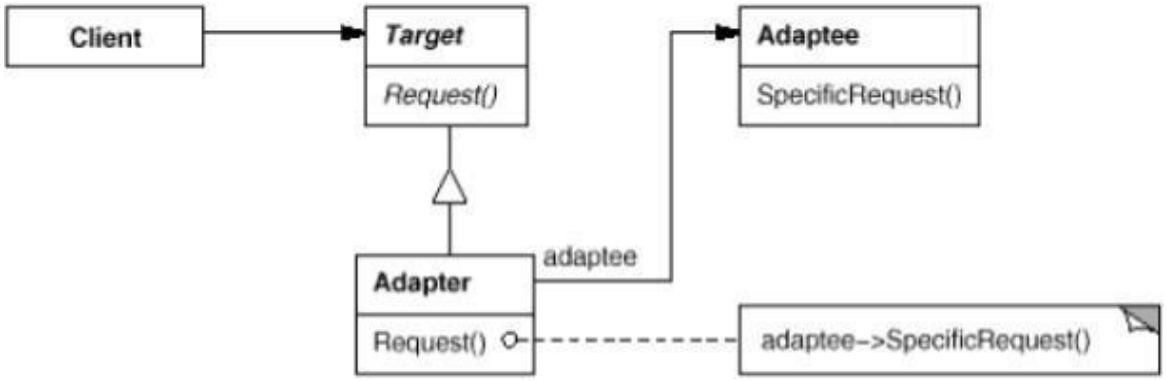
Applicability: Use adapter pattern when:

1. You want to use an existing class, and its interface is not what you needed.
2. You want to create a reusable class that cooperates with the incompatible classes.
3. You need to use several subclasses (object adapter only) by adapting to theirinterfaces (by sub classing each subclass) which is impractical. An object adapter can adapt the interface of their parent class.

Structure: A class adapter uses multiple inheritance to adapt one interface to another. The structure of class adapter is shown below:



An object adapter relies on object composition. The structure of an object adapter is as shown below:



Participants:

- Target (shape): Defines the domain specific interface the client uses.
- Client (Drawing Editor): Collaborates with the objects conforming to the Target interface.
- Adaptee (TextView): Defines an existing interface that needs to be adapted.
- Adapter(TextShape): Adapts the interface of the Adaptee to the Traget interface.

Collaborations:

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

Consequences: Class and object adapters have different trade-offs.

A class adapter:

- ✓ Adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and its subclasses.
- ✓ Let Adapter to override some of the behavior of the Adaptee since it is a subclass of Adaptee.
- ✓ Introduces only one object, and no additional pointer indirection is needed to get to the Adaptee.

An object adapter:

- ✓ Lets a single Adapter work with many Adaptees i.e the Adaptee itself and all of its subclasses. The Adapter can also add functionality to all Adaptees at once.
- ✓ Makes it harder to override Adaptee behavior.

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

1. How much adapting does Adapter do ?
   The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.
2. Pluggable adapters.
   Are classes with built-in interface adaptation.
3. Using two-way adapters to provide transparency.
   They're useful when two different client s need to view an object differently.

Two-way class adapter conforms to both of the adapted classes and can work in either system.

Implementation: Some of the issues to keep in mind while implementing adapter pattern are given below:

1. Implementing class adapters in C++: Adapter would inherit publicly from Target and privately from Adaptee. Thus Adapter would be a subtype of Target but not of Adaptee.

2. Pluggable adapters: Three ways to implement pluggable adapters for the TreeDisplay

A narrow interface consisting of only a couple of operations is easier to adapt than an interface with dozens of operations

The narrow interface leads to three implementation approaches:

a. Using abstract operations: Define corresponding abstract operations for the narrow Adaptee interface in the TreeDisplay class. Subclasses must implement the abstract operations and adapt the hierarchically structured object.

b. Using delegate objects: In this approach, TreeDisplay forwards requests for accessing the hierarchical structure to a delegate object.

c. Parameterized adapters: The usual way to support pluggable adapters in Smalltalk is to parameterize an adapter with one or more blocks.. A block can adapt a request, and the adapter can store a block for each individual request.

Sample code:

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
```

Shape assumes a bounding box defined by its opposing corners .
In contrast , TextView is defined by an origin , height , and width. Shape also defines a CreateManipulator operation for creating a Manipulator object , which knows how to animate a shape when the user manipulates it.

```
class TextShape : public Shape, private TextView {
public:
    TextShape();

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

The BoundingBox operation converts Textview's interface to conform to Shape's.

```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

Th e IsEmpty operation demonstrates the direct forwarding of request s common in adapter implementations:

```
bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}
```

Finally , we define CreateManipulator that supports manipulation of a TextShape.

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

The adapter Text Shape maintains a pointer to Text View.

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

TextShape must initialize the pointer to the TextView instance , and it does so in the constructor .

```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

Known Uses

i. **ET++**Draw reuses the ET++ classes for text editing by using a TextShape adapter class.
ii. **Interviews 2.6** defines an object adapter called GraphicBlock, a subclass of Interactor that contains a Graphic instance. The GraphicBlock adapts the interface of the Graphic class to that t of Interactor.
iii. **ObjectWorks\Smalltalk** includes a subclass of ValueModel called PluggableAdaptor. A PluggableAdaptor object adapts other objects to the ValueModel interface (value , value:)
iv. **NeXT's AppKit [Add94]** use delegate objects to perform interface adaptation.

Related Patterns

- **Bridge** has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an existing object.
- **Decorator** enhances another object without changing its interface. A 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.
- **Proxy** defines a representative or surrogate for another object and does not change its interface.

BRIDGE(Object Structural)

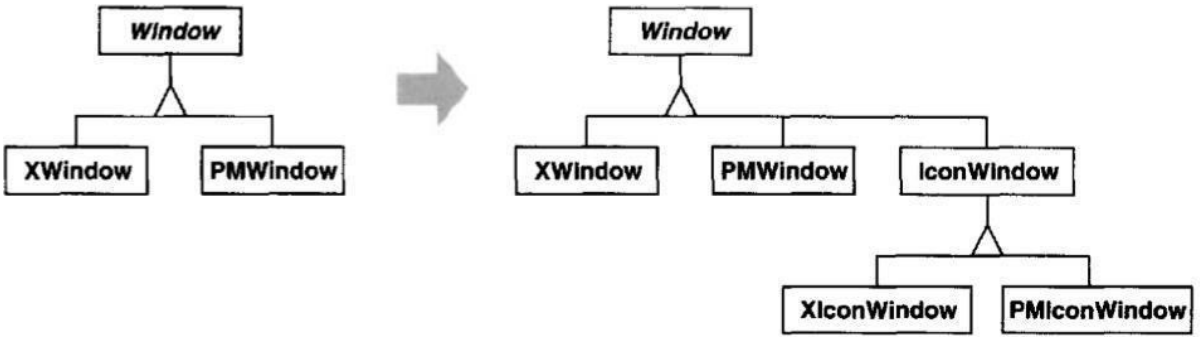
Intent: To decouple an abstraction from its implementation so that both can be changed independently.

Also knows as: Handle/Body

Motivation:

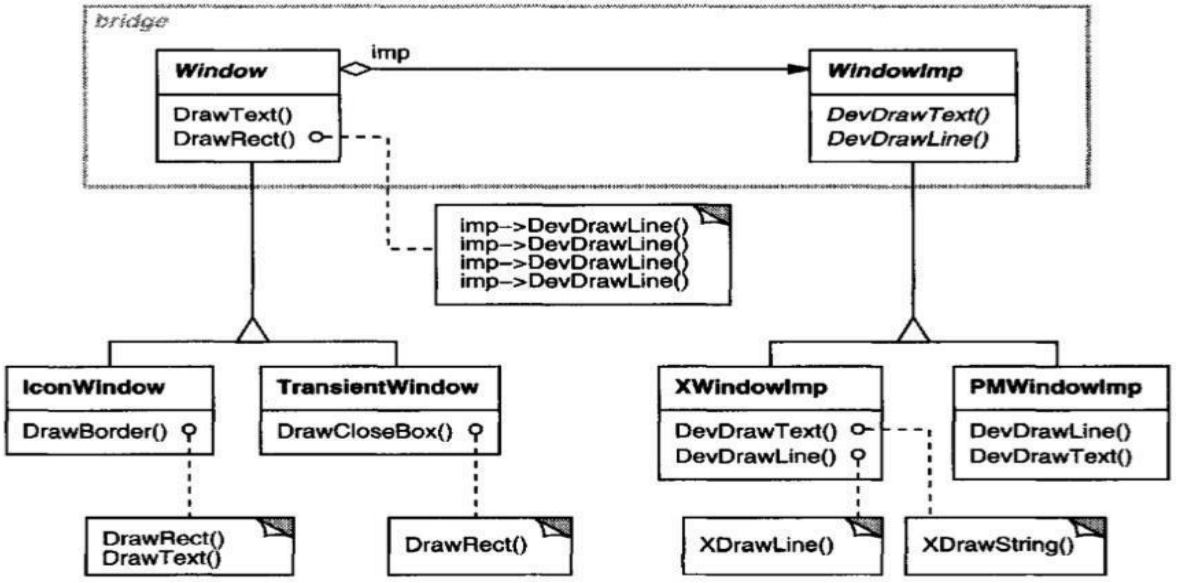
Consider the implementation of a portable Window abstraction in a user interface toolkit. This abstraction should enable us to write applications that work on both the X Window System and IBM's Presentation Manager (PM) platform. But this approach has two drawbacks:

1. To support Icon Windows for both platforms, we have to implement two new classes , XlconWindow and PMIconWindow.



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

The Bridge pattern addresses these problems by putting the Window abstraction and its implementation in separate class hierarchies.

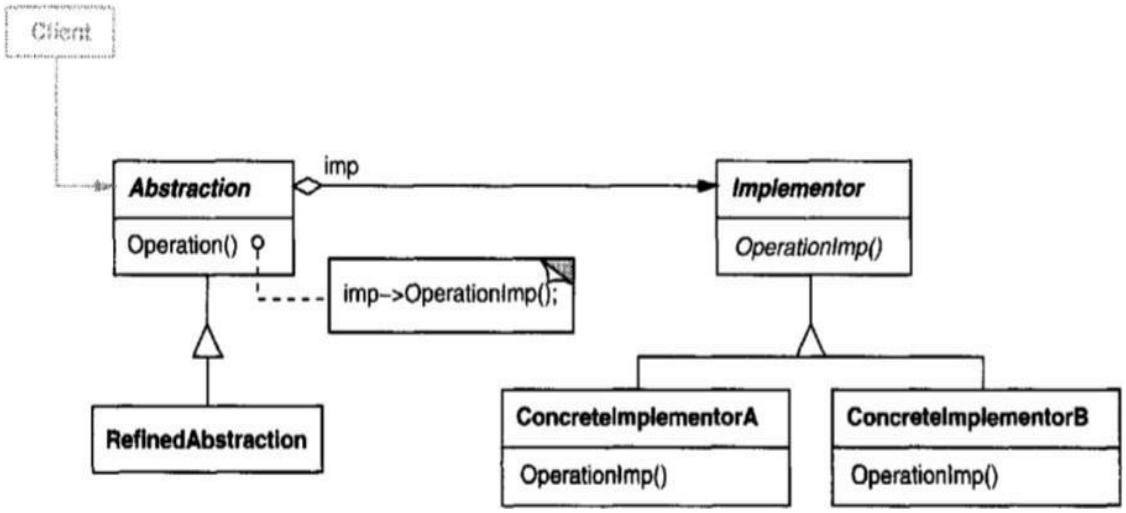


We refer to the relationship between Window and Windowlmp as a bridge , because it bridges the abstraction and its implementation, letting them vary independently.

Applicability: Use bridge pattern when:

- To avoid permanent binding between abstraction and implementation.
- Both abstractions and implementations should be extensible by creating subclasses.
- Changes in implementation should have no impact on client.
- To share and implementation among multiple objects, and this fact should be hidden from client.
- (C++) you want to hide the implementation of an abstraction completely from clients.

Structure: The structure of bridge pattern is as shown below:



Participants: Following are the participants in bridge pattern:

- Abstraction(window): Defines the abstraction interface and maintains a reference to an object of type Implementor
- RefinedAbstraction (Iconwindow): Extends the interface defined by Abstraction.
- Implementor (WindowImp): Defines the interface for implementation classes.
- ConcreteImplmentor (XWindowImp, PMWindowImp): Implements the Implementor interface and defines its concrete implementation

Collaborations: Abstraction forwards client requests to its Implementor object.

Consequences: The bridge pattern has the following consequences:

1. **Decoupling interface and implementation**: An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.

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

3. **Hiding implementation details from clients**: You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism.

Implementation: Following issues should be considered while implementing bridge pattern:

1. **Only one Implementor:** In situations where there's only one implementation, creating an abstract Implementor class isn't necessary. This is a degenerate case of the Bridge pattern; there's a one-to-one relationship between Abstraction and Implementor. Nevertheless, this separation is still useful when a change in the implementation of a class must not affect its existing clients—that is, they shouldn't have to be recompiled, just relinked.

2. **Creating the right Implementor object**: How, when, and where do you decide which Implementor class to instantiate when there's more than one? If Abstraction knows about all ConcreteImplementor classes, then it can instantiate one of them in its constructor; it can decide between them based on parameters passed to its constructor. If, for example, a collection class supports multiple implementations, the decision can be based on the size of the collection. A linked list implementation can be used for small collections and a hash table for larger ones.

3.**Sharing implementors.** The code for assigning handles with shared bodies has the following general form:

```
Handle& Handle::operator= (const Handle& other)  {
    other._body->Ref();
    _body->Unref();

    if (_body->RefCount() == 0) {
        delete _body;
    }
    _body = other._body;

    return *this;
}
```

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

Sample Code

The Window class defines the window abstraction for client applications:

```
class Window {
public:
    Window(View* contents);

    // requests handled by window
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();

    // requests forwarded to implementation
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();

    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);

protected:
    WindowImp* GetWindowImp();
    View* GetView();

private:
    WindowImp* _imp;
    View* _contents; // the window's contents
};
```

Window maintains a reference to a WindowImp, the abstract class that declares an interface to the underlying windowing system.

```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // lots more functions for drawing on windows...
protected:
    WindowImp();
};
```

For example, Application Window will implement DrawContents to draw the View instance it stores:

```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

IconWindow stores the name of a bitmap for the icon it displays...

```
class IconWindow : public Window {
public:
    // ...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

..and it implements DrawContents to draw the bitmap on the window:

```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}
```

Window operations are defined in terms of the Windowlmp interface

```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

The XWindowImp subclass supports the X Window System:

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // remainder of public interface...
private:
    // lots of X window system-specific state, including:
    Display* _dpy;
    Drawable _winid;  // window id
    GC _gc;           // window graphic context
};
```

For Presentation Manager (PM), we define a PMWindowlmp class:

```
class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // remainder of public interface...
private:
    // lots of PM window system-specific state, including:
    HPS _hps;
};
```

For example, DeviceRect is implemented for X as follows:

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

The PM implementation might look like this:

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left;    point[0].y = top;
    point[1].x = right;   point[1].y = top;
    point[2].x = right;   point[2].y = bottom;
    point[3].x = left;    point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR)  ||
        (GpiEndPath(_hps) == false)
    ) {
        // report error

    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}
```

GetWindowImp operation gets the right instance from an abstract that effectively encapsulates all window system specifics.

```
WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}
```

Known Uses

1) ET++ Window/WindowPort design extends the Bridge pattern in that the WindowPort also keeps a reference back to the Window.
2) NeXT's AppKit [Add94] uses the Bridge pattern in the implementation and display of graphical images.
3) AppKit provides an NXImage/NXImageRep bridge. NXImage defines the interface for handling images. The implementation of images is defined in a separate NXImageRep class hierarchy having subclasses such as NXEPSImageRep, NXCachedImageRep, and NXBitMapImageRep.

Related Patterns

1) An Abstract Factory can create and configure a particular Bridge.
2) The Adapter pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed. Bridge, on the other hand, is use d up-front in a design to let abstractions and implementations vary independently.

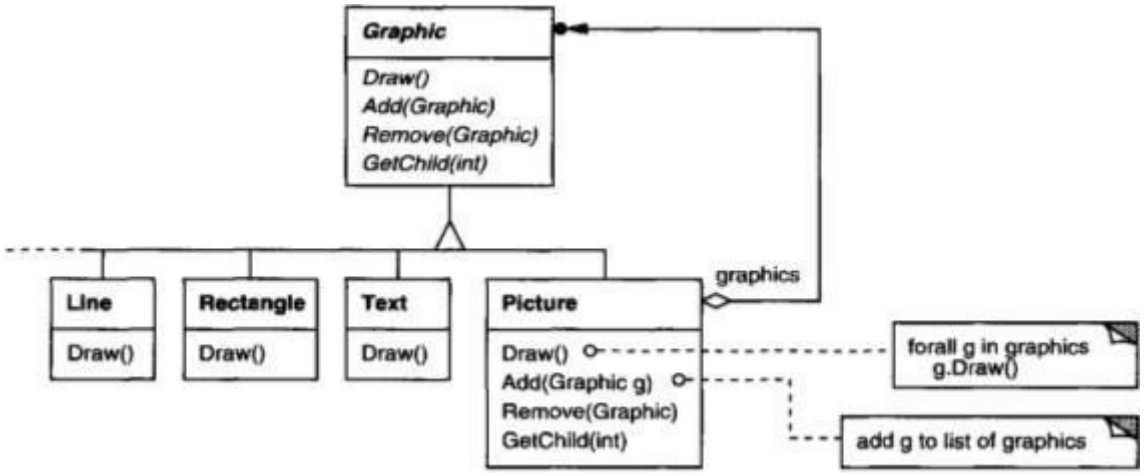
COMPOSITE(Object Structure)

Intent: To compose objects into tree structures to represent part-whole hierarchies. Composite pattern lets client treat individual objects and compositions of objects uniformly.

Motivation:

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

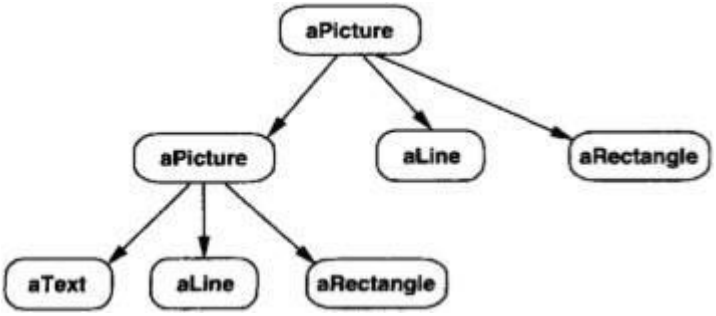
- ❖ Bu t there's a problem with this approach :
  - o Code that uses these classes must treat primitive and container objects differently,. Having to distinguish these objects makes the application more complex . The Composite pattern describe s how to use recursive composition so that clients don't have to make this distinction.



- ✓ Graphic declares operations like Draw that are specific to graphical objects. It also declares operations that all composite objects share , such as operations for accessing and managing its children.
- ✓ The subclasses Line , Rectangle, and Text define primitive graphical objects. These classes implement draw to draw lines , rectangles , and text, respectively .

- ✓ Since primitive graphics have no child graphics, none of these subclasses implements child-related

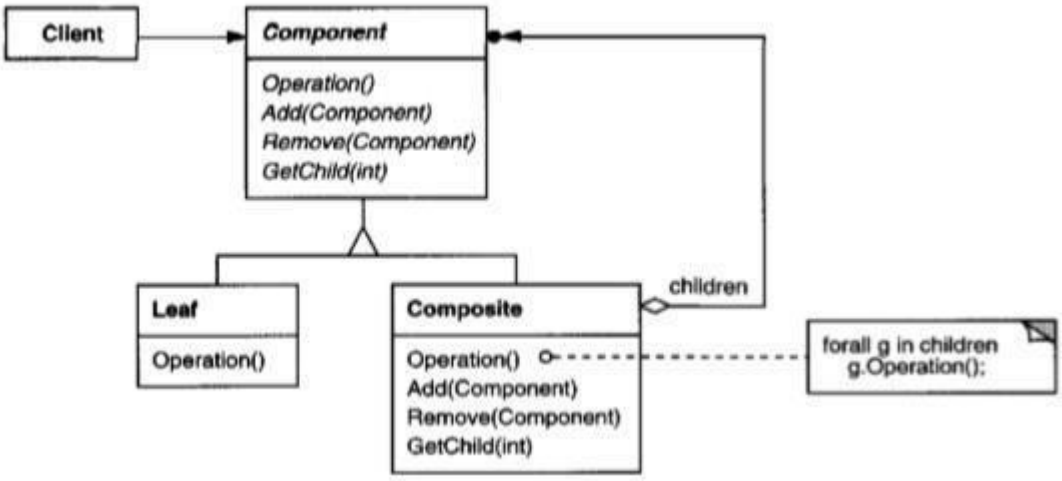
The following diagram shows a typical composite object structure of recursively composed Graphic objects:



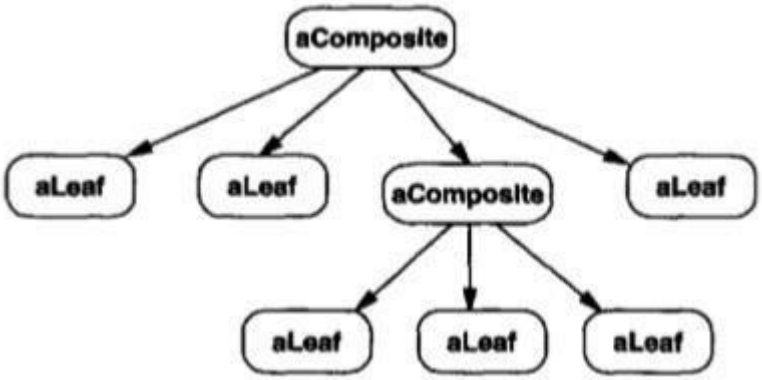
Applicability: Use composite pattern when:

1. You-want to represent part-whole hierarchies of objects.
2. You want clients to be able to ignore the difference between compositions of objects and individual objects.

Structure: The structure of composite pattern is as shown below:



A typical Composite object structure might look like this:



Participants: Following are the participants in composite pattern:

1. **Component:** Declares the interface for objects in the composition. Declares an interface for accessing and managing its child components.
2. **Leaf**: Represents leaf objects in the composition. A leaf has no children. Defines behavior for primitive objects in the composition.
3. **Composite:** Defines behavior for components having children. Stores child components. Implements child-related operations in the composite interface.
4. **Client:** Manipulates objects in the composition through the Component interface.

Collaborations: Clients use the Component class interface to interact with objects in the composite structure.

Consequences: The composite pattern:

1. Defines class hierarchies consisting of primitive objects and composite objects. Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.
2. Makes the client simple. Clients can treat composite structures and individual objects uniformly.
3. Makes it easier to add new kinds of components. Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.
4. Can make your design overly general. The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite. Sometimes you want a composite to have only certain components.

Implementation: Following are the issues to consider when implementing composite pattern:

1. Explicit parent references: Maintaining references from child components to their parent can simplify the traversal and management of a composite structure. The parent reference simplifies moving up the structure and deleting a component. Parent references also help support the Chain of Responsibility pattern.

2. Sharing components: It's often useful to share components, for example, to reduce storage requirements. But when a component can have no more than one parent, sharing components becomes difficult.

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

4. Declaring the child management operations: Although the Composite class implements the Add and Remove operations for managing children, an important issue in the Composite pattern is which classes declare these operations in the Composite class hierarchy.

5. Should Component implement a list of Components: You might be tempted to define the set of children as an instance variable in the Component class where the child access and management operations are declared. But putting the child pointer in the base class incurs a space penalty for every leaf, even though a leaf never has children.

6. Child ordering: Many designs specify an ordering on the children of Composite. In the earlier Graphics example, ordering may reflect front-to-back ordering. If Composites represent parse trees, then compound statements can be instances of a Composite whose children must be ordered to reflect the program.

7. Caching to improve performance: If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children. The Composite can cache actual results or just information that lets it short-circuit the traversal or search.

8. Who should delete components: In languages without garbage collection, it's usually best to make a Composite responsible for deleting its children when it's destroyed. An exceptionto this rule is when Leaf objects are immutable and thus can be shared.

9. What's the best data structure for storing components: Composites may use a variety of data structures to store their children, including linked lists, trees, arrays, and hash tables. The choice of data structure depends (as always) on efficiency.

Sample code: A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc and when we fill the drawing with color (say Red), the same color also gets applied to the Objects in the drawing. Here drawing is made up of different parts and they all have same operations.

Composite Pattern consists of following objects.

1. Base Component – Base component is the interface for all objects in the composition, client program uses base component to work with the objects in the composition. It can be an interface or an abstract class with some methods common to all the objects.

2. Leaf – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn't have references to other Components.

3. Composite – It consists of leaf elements and implements the operations in base component.

Here I am applying composite design pattern for the drawing scenario.

Base Component: Base component defines the common methods for leaf and composites, we can create a class Shape with a method draw(String fillColor) to draw the shape with given color.

```
//Shape.java
public interface Shape
 {
        public void draw(String fillColor);
 }
```

Leaf Objects: Leaf implements base component and these are the building block for the composite. We can create multiple leaf objects such as Triangle, Circle etc.

```
//Triangle.java
public class Triangle implements Shape
 {
        @Override
         public void draw(String fillColor)
         {
                System.out.println("Drawing Triangle with color "+fillColor);
          }
  }

//Circle.java
public class Circle implements Shape
{
 @Override
  public void draw(String fillColor)
 {
   System.out.println("Drawing Circle with color "+fillColor);
 }
```

```
}
```

Composite: A composite object contains group of leaf objects and we should provide some helper methods to add or delete leafs from the group. We can also provide a method to remove all the elements from the group.

```
//Drawing.java
import java.util.ArrayList;
import java.util.List;
public class Drawing implements Shape
{       //collection of Shapes
private List<Shape> shapes = new ArrayList<Shape>();
@Override
public void draw(String fillColor)
 {
  for(Shape sh : shapes)
 {            sh.draw(fillColor);        }
}          //adding shape to drawing


public void remove(Shape s)

{         shapes.remove(s);     }
  //removing all the shapes
 public void clear()
{
   System.out.println("Clearing all the shapes from drawing");
     this.shapes.clear();
   }
 }

   public void add(Shape s)
{
      this.shapes.add(s);
 }          //removing shape from drawing


//TestCompositePattern.java

public class TestCompositePattern
{       public static void main(String[] args)
{         Shape tri = new Triangle();
   Shape tri1 = new Triangle();
   Shape cir = new Circle();
        Drawing drawing = new Drawing();
```

```
  drawing.add(tri1);
    drawing.add(tri1);
drawing.add(cir);
      drawing.draw("Red");
      drawing.clear();
   drawing.add(tri);
drawing.add(cir);
     drawing.draw("Green");
}
}
```

Known Uses

1. ET++ (with its VObjects [WGM88]) and
2. Interviews (Style s [LCI+92],
3. Graphics [VL88] , and
4. The RT L Smalltalk compiler framework
5. RegisterTransferSet , is a Composite class for representing assignments that change several registers at once.
6. Financial domain , where a portfolio aggregates individual assets.

Related Patterns

1. Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.
2. Flyweight lets you share components, but they can no longer refer to their parents.
3. Iterator can be used to traverse composites.
4. Visitor localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.

Decorator Pattern

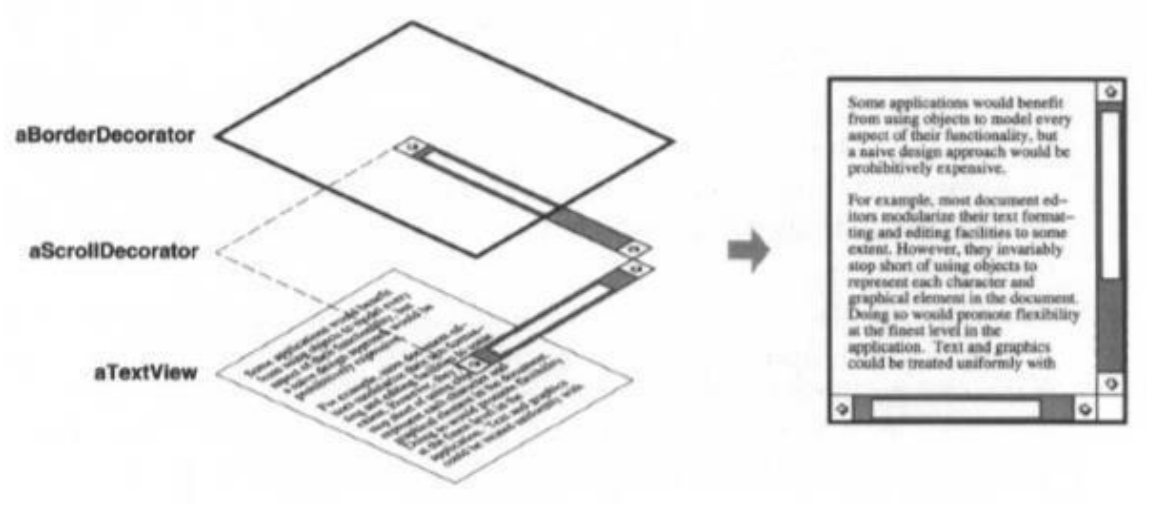
Intent: To attach additional responsibilities to an object dynamically. Decorator provides an alternative to subclassing for extending the functionality.

Also knows as: Wrapper

Motivation

- ✓ Sometimes we want to add responsibilities to individual objects , not to an entire class .
- ✓ A graphical user interface toolkit, for example, should let you add properties like borders or behaviors like scrolling to any user interface component.
- ✓ One way to add responsibilities is with inheritance.

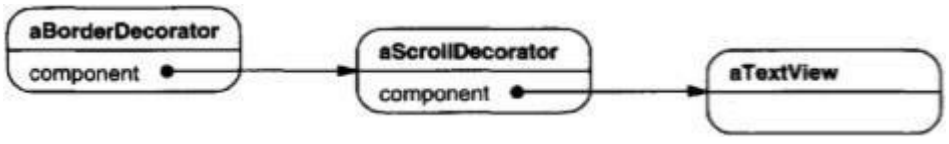
- ✓ Inheriting a border from another class puts a border around every subclass instance,this is inflexible. A client can't control how and when to decorate the component with a border.
- ✓ A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a decorator .
- ✓ The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding.



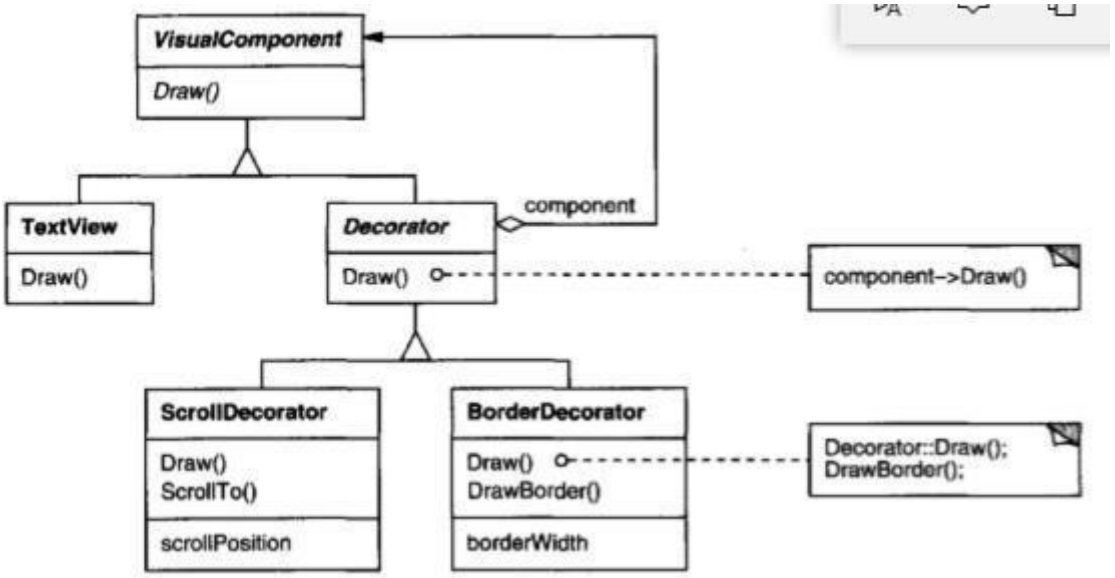
For example:

- ✓ Suppose we have a Text View object that displays text in a window.
- ✓ Text View has no scroll bars by default, because we might not always need them. When we do, we can use a ScrollDecorator to add them.
- ✓ Suppose we also want to add a thick black border around the Text View . We can use a BorderDecorator to add this as well.
- ✓ We simply compose the decorators with the Text Vie w to produce the desired result.

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



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

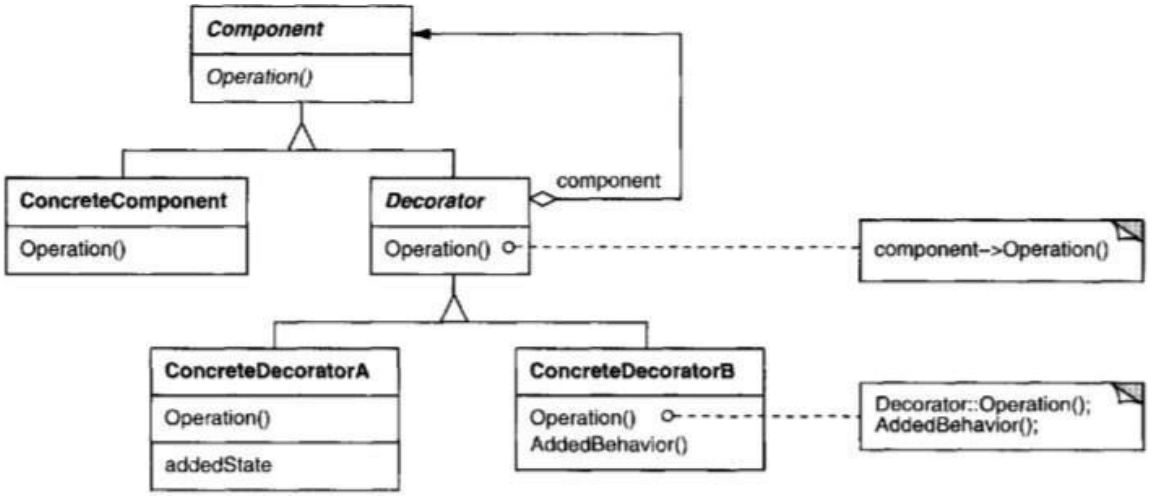


- ✓ Visual Component defines their drawing and event handling interface.
- ✓ Note how the Decorator class simply forwards draw requests to its component, and how Decorator subclasses can extend this operation. Decorator subclasses are free to add operations for specific functionality.

Applicability:

1. To add additional responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.
2. For responsibilities that can be withdrawn.
3. When extension by subclassing is impractical.

Structure: The structure of decorator pattern is as shown below:



Participants

• Component (VisualComponent) - defines the interface for objects that can have responsibilities added to them dynamically.

• ConcreteComponent (TextView) - defines an object to whic h additional responsibilitie s can be attached.

• Decorator - maintains a reference to a Component object and defines an interface that conforms to Component's interface.

• ConcreteDecorator (BorderDecorator, ScrollDecorator) - Adds responsibilities to the component

Collaborations: Decorator forwards requests to its component object. It may optionally perform additional operations before and after forwarding the request.

Consequences: The decorator pattern has atleast two key benefits and two liabilities:

1. More flexibility than static inheritance: The Decorator pattern providesa more flexible way to add responsibilities to objects than can be had withstatic (multiple) inheritance.

2. Avoids feature-laden classes high up in the hierarchy: Decorator offersa pay-as-you-go approach to adding responsibilities. Instead of trying tosupport all foreseeable features in a complex, customizable class, you candefine a simple class and add functionality incrementally with Decoratorobjects.

3. A decorator and its component aren't identical: A decorator acts as atransparent enclosure. But from an object identity point of view, adecorated component is not identical to the component itself.

4. Lots of little objects: A design that uses Decorator often results in systemscomposed of lots of little objects that all look alike. The objects differonly in the way they are interconnected, not in their class or in the valueof their variables.

Implementation: Following issues should be considered when applying the decorator pattern:

1. Interface conformance: A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class.

2. Omitting the abstract Decorator class: There's no need to define an abstract Decoratorclass when you only need to add one responsibility.

3. Keeping Component classes lightweight: To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data.

4. Changing the skin of an object versus changing its guts: We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy (349) pattern is a good example of a pattern for changing the guts.

Sample Code

The following cod e shows how to implement user interface decorators in C++.

```
class VisualComponent {
public:
    VisualComponent();

    virtual void Draw();
    virtual void Resize();
    // ...
};
```

We define a subclass of VisualComponent called Decorator

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

Subclasses of Decorator define specific decorations. For example, the class BorderDecorator adds a border to its enclosing component

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};

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

TextView is a VisualComponent, which lets us put it into the window:

window->SetContents(textView);

Bu t w e want a bordere d an d scrollabl e TextView. S o w e decorat e i t accordingl y before putting it in the window.

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

Known Uses

1. Interviews [LVC89 , LCI+92],
2. ET++ [WGM88] ,
3. ObjectWorks\Smalltalk class library
4. A DebuggingGlyph prints out debugging information before and after it forwards a layout request to its component.

Related patterns:

1. **Adapter:** A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.

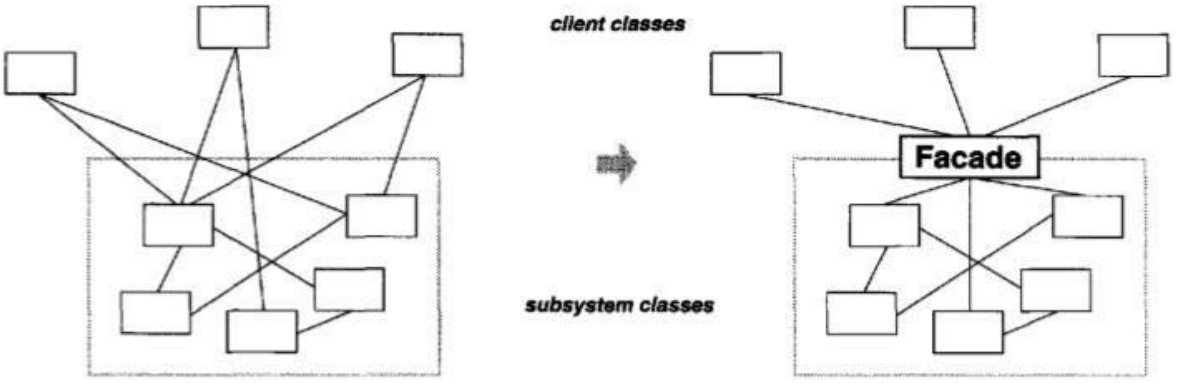
2. **Composite:** A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.
3. **Strategy:** A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

FAÇADE (Object Structure)

Intent: 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.

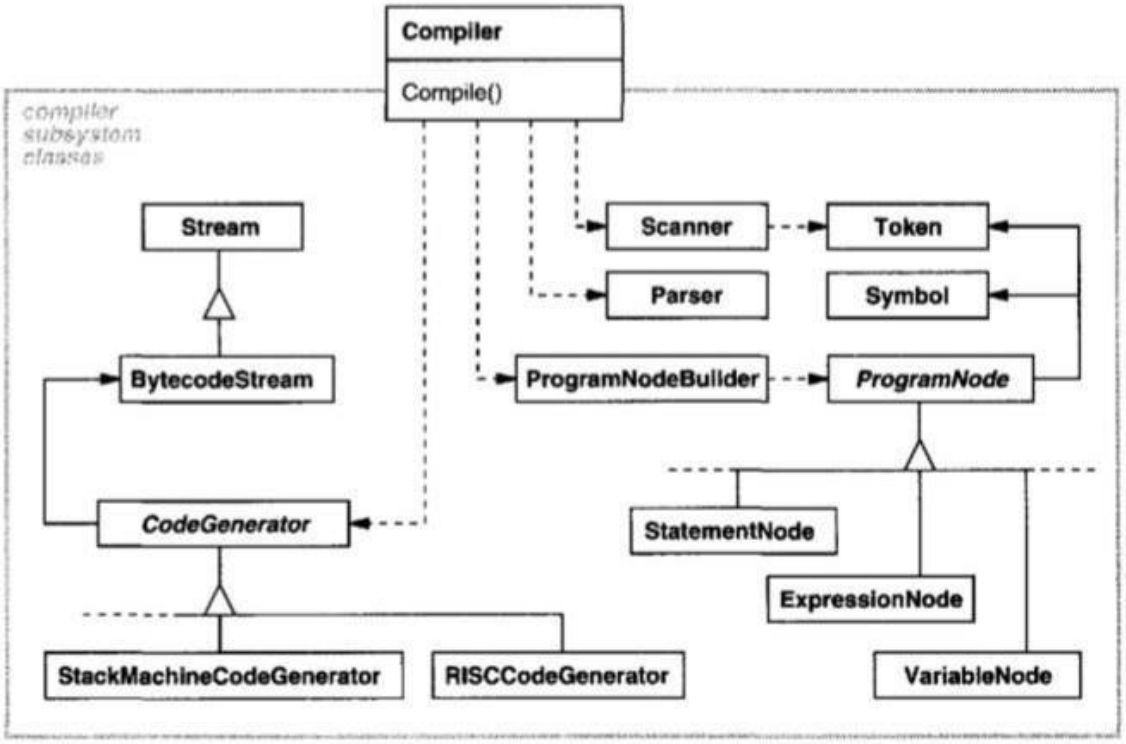
Motivation

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



Consider for example a programming environment that gives applications access to its compiler subsystem. This subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and ProgramNodeBuilder that implement the compiler. Some specialized applications might need to access these classes directly. But most clients of a compiler generally don't care about details like parsing and code generation; they merely want to compile some code. For them, the powerful but lowlevel interfaces in the compiler subsystem only complicate their task.

To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class. This class defines a unified interface to the compiler's functionality. The Compiler class acts as a facade: It offers clients a single, simple interface to the compiler subsystem.

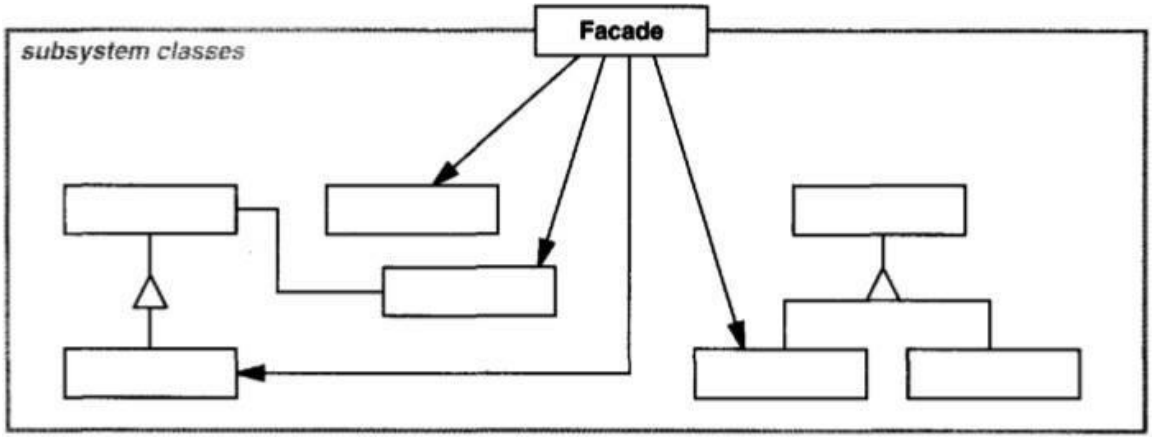


Applicability:

Use façade pattern :

1. To provide a simple interface to a complex system.
2. To decouple a subsystem from clients and other subsystems, thereby promoting system independence and portability.
3. To define an entry point to each subsystem level. If subsystems are dependent, then the dependencies can be simplified by making them communicate with each other solely through their façade.

Structure



Participants: The participants in the façade pattern are:

1. **Façade (compiler):** Knows which subsystem classes are responsible for a request. Delegates client requests to appropriate subsystem objects.
2. **Subsystem classes (scanner, parser, programnode….):** Implement subsystem functionality. Handle work assigned by the Façade object. Have no knowledge of the Façade.

Collaborations: Clients communicate with the subsystem by sending requests to the Façade, which forwards them to the appropriate subsystem object.

Consequences: The façade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.

2. It promotes weak coupling between the subsystem and its clients. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystem without affecting its clients.

3. It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Implementation: Following issues should be considered when implementing façade pattern:

1. **Reducing client-subsystem coupling:** The coupling between clients and the subsystem can be reduced even further by making Facade an abstract class with concrete subclasses for different implementations of a subsystem. Then clients can communicate with the subsystem through the interface of the abstract Facade class. This abstract coupling keeps clients from knowing which implementation of a subsystem is used.

2. **Public versus private subsystem classes:** A subsystem is analogous to a class in that both have interfaces, and both encapsulate something—a class encapsulates state and operations, while a subsystem encapsulates classes. And just as it's useful to think of the public and private interface of a class, we can think of the public and private interface of a subsystem.

Sample Code :
Let's take a closer look at how to put a facade on a compiler subsystem.
The Scanner class takes a stream of characters and produces a stream of tokens, one token at a time.

```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();

    virtual Token& Scan();
private:
    istream& _inputStream;
};
```

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

```
class Parser {
public:
    Parser();
    virtual ~Parser();

    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

Parser calls back on ProgramNodeBuilder to build the parse tree incrementally.

```
class ProgramNodeBuilder {
public:
    ProgramNodeBuilder();

    virtual ProgramNode* NewVariable(
        const char* variableName
    ) const;

    virtual ProgramNode* NewAssignment(
        ProgramNode* variable, ProgramNode* expression
    ) const;

    virtual ProgramNode* NewReturnStatement(
        ProgramNode* value
    ) const;

    virtual ProgramNode* NewCondition(
        ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart
    ) const;
    // ...
```

```
    ProgramNode* GetRootNode();
private:
    ProgramNode* _node;
};
```

ProgramNode defines an interface for manipulating the program node and its children, if any.

```
class ProgramNode {
public:
    // program node manipulation
    virtual void GetSourcePosition(int& line, int& index);
    // ...

    // child manipulation
    virtual void Add(ProgramNode*);
    virtual void Remove(ProgramNode*);
    // ...

    virtual void Traverse(CodeGenerator&);
protected:
    ProgramNode();
};
```

The Traverse operation takes a CodeGenerator object. ProgramNode subclasse s use this object to generate machine cod e in the form of Bytecode objects on a BytecodeStream.

```
class CodeGenerator {
public:
    virtual void Visit(StatementNode*);
    virtual void Visit(ExpressionNode*);
    // ...
protected:
    CodeGenerator(BytecodeStream&);
protected:
    BytecodeStream& _output;
};
```

ExpressionNode defines Traverse as follows:

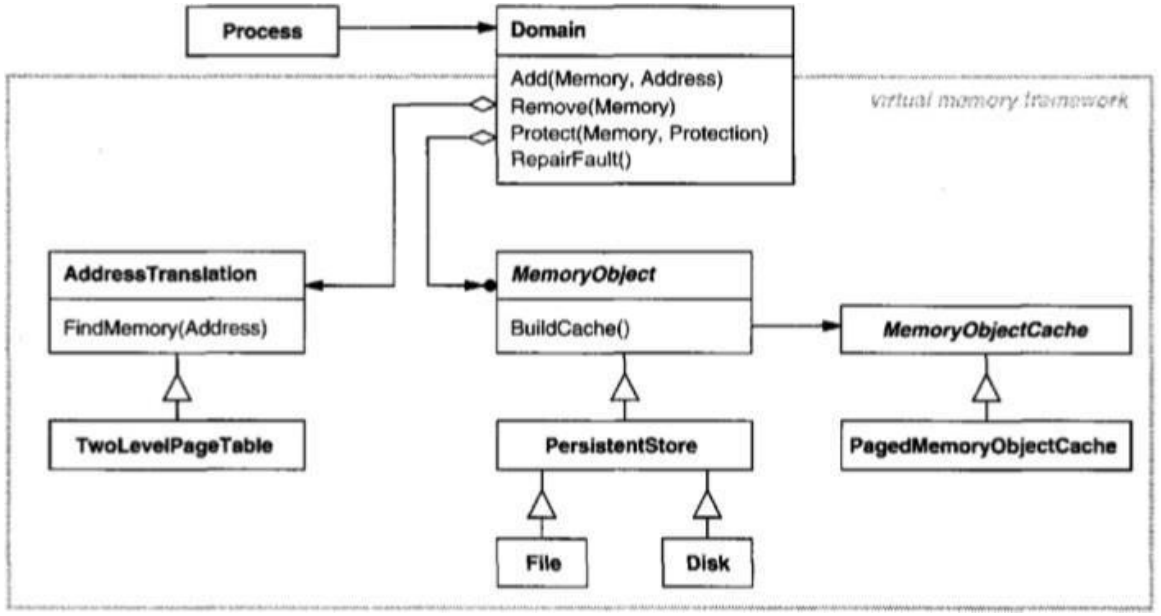
```
void ExpressionNode::Traverse (CodeGenerator& cg) {
    cg.Visit(this);

    ListIterator<ProgramNode*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Traverse(cg);
    }
}
```

Compiler provides a simple interface for compiling source and generating code for a particular machine.

```
class Compiler {
public:
    Compiler();

    virtual void Compile(istream&, BytecodeStream&);
};


void Compiler::Compile (
    istream& input, BytecodeStream& output
) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

Known Uses

1. In the ET++ application framework [WGM88] , an application can have built-in browsing tools for inspecting its objects at run-time. These browsing tools are implemented in a separate subsystem that includes a Facade class called "ProgrammingEnvironment."
2. The Choices operating system [CIRM93] uses facades to compose many frameworks into one . The key abstractions in Choice s are processes , storage, and address spaces.
3. The virtual memory framework has Domain as its façade



The main operations on Domain support are

- adding a memory object at a particular address,
- removing a memory object, and
- handling a page fault.

The virtual memory subsystem uses the following components internally:

- MemoryObject represents a data store.
- MemoryObjectCache caches the data of MemoryObjects in physical memory. MemoryObjectCache is actually a Strategy that localizes the caching policy.
- AddressTranslation encapsulates the address translation hardware.

Related patterns:

1. Abstract Factory
   - It can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way.
   - Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.
2. Mediator
   - Mediator is similar to Facade in that it abstracts functionality of existing classes.
   - However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them.
   - In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.
3. Usually only one Facade object is required. Thus Facade objects are often Singletons.

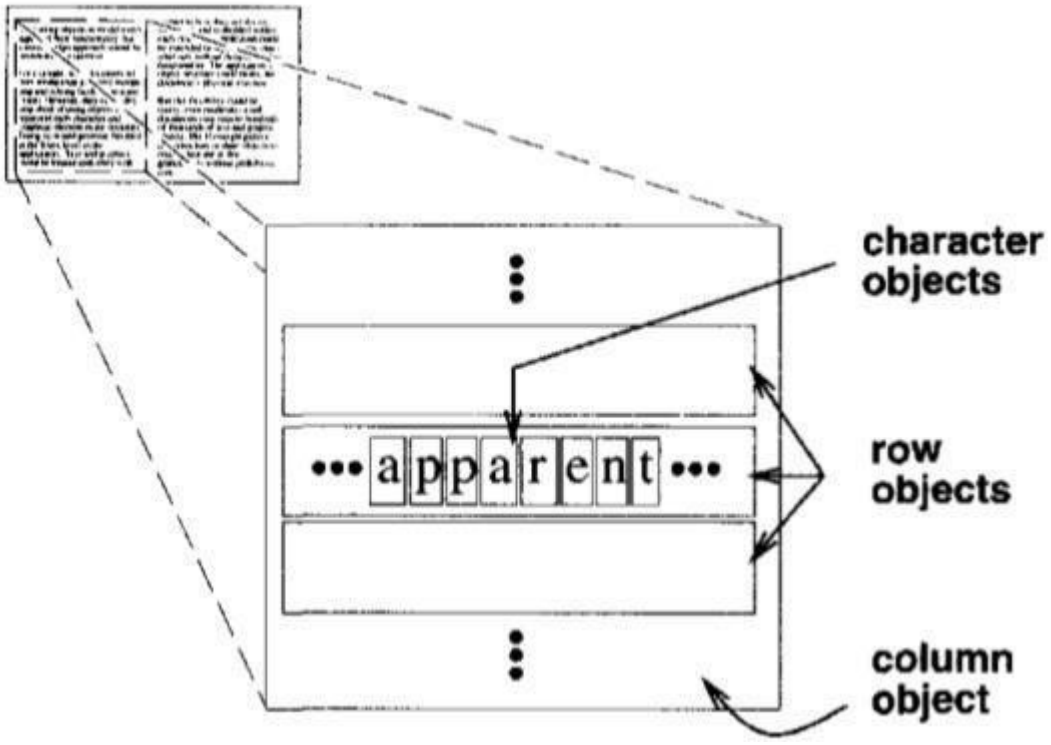
FLYWEIGHT (Object Structure)

Intent

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

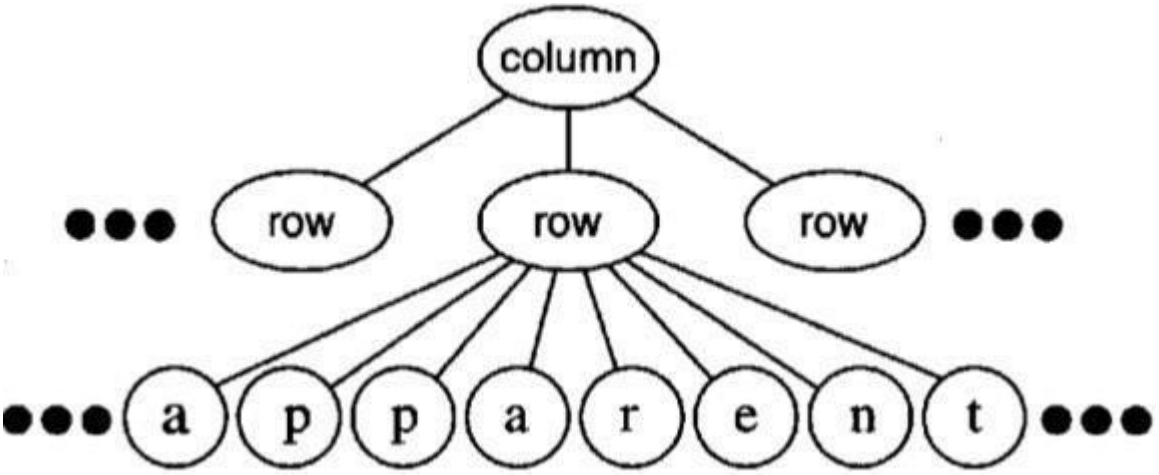
Motivation

- ✓ Most document editor implementations have text formatting and editing.
- ✓ Object-oriented document editors typically use objects to represent embedded elements like tables and figures.
- ✓ Characters and embedded elements could then be treated uniformly with respect to how they are drawn and formatted.
- ✓ The application could be extended to support new character sets without disturbing other functionality.
- ✓ The following diagram shows how a document editor can use objects to represent characters.

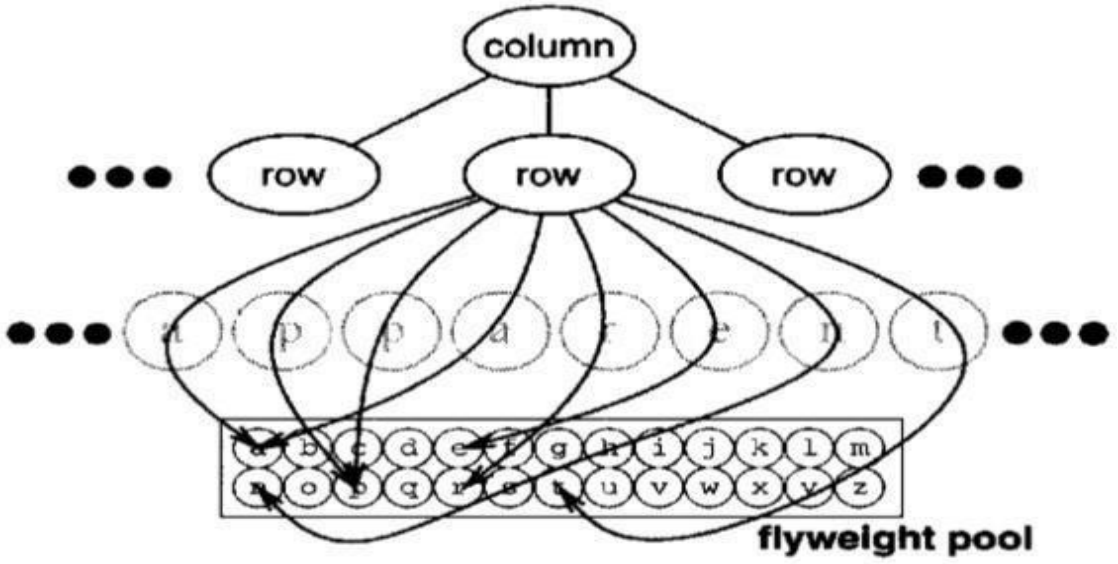


- ✓ The **drawback** of such a design is its cost.
- ✓ Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time overhead.
- ✓ The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost.
- ✓ A flyweight is a shared object that can be used in multiple contexts simultaneously.

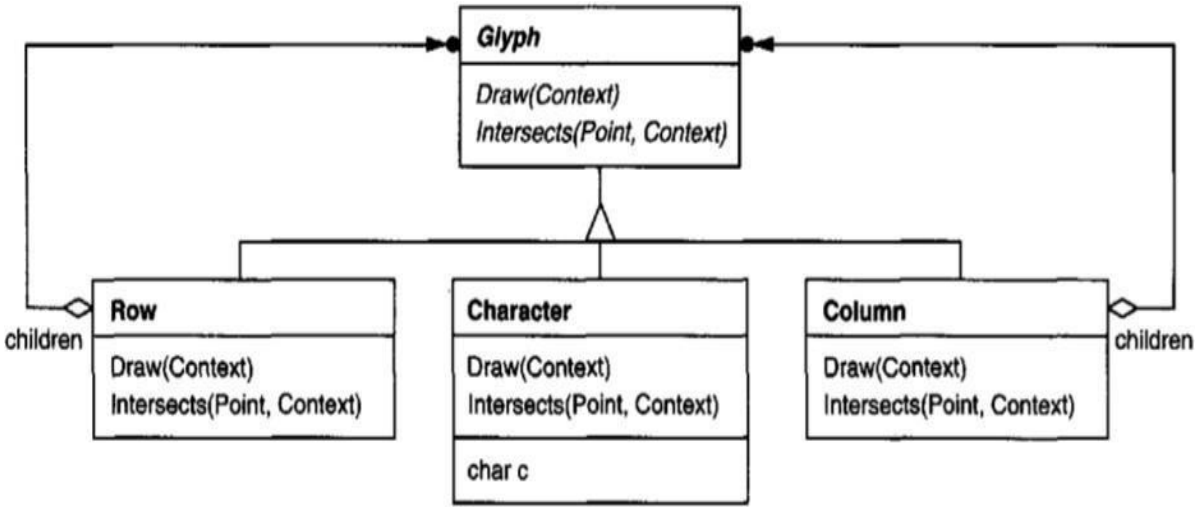
- ✓ The flyweight acts as an independent object in each context.
- ✓ The key concept here is the distinction *between intrinsic and extrinsic state*.
  - **Intrinsic state** is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable.
  - **Extrinsic state** depends on and varies with the flyweight's context and therefore can't be shared.
- ✓ Client objects are responsible for passing extrinsic state to the flyweight when it needs it.
- ✓ **For example,**
  - a document editor can create a flyweight for each letter of thealphabet.
  - Each flyweight stores a character code , but its coordinate position in the document and its typographic style can be determined from the text layout algorithms and formatting commands in effect wherever the character appears.
- ✓ The character code is intrinsic state, while the other information is extrinsic.
- ✓ Logically there is an object for every occurrence of a given character in the document:



- ✓ Each occurrence of a particular character object refers to the same instance in the shared pool of flyweight objects:



The class structure for these objects is shown next.



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

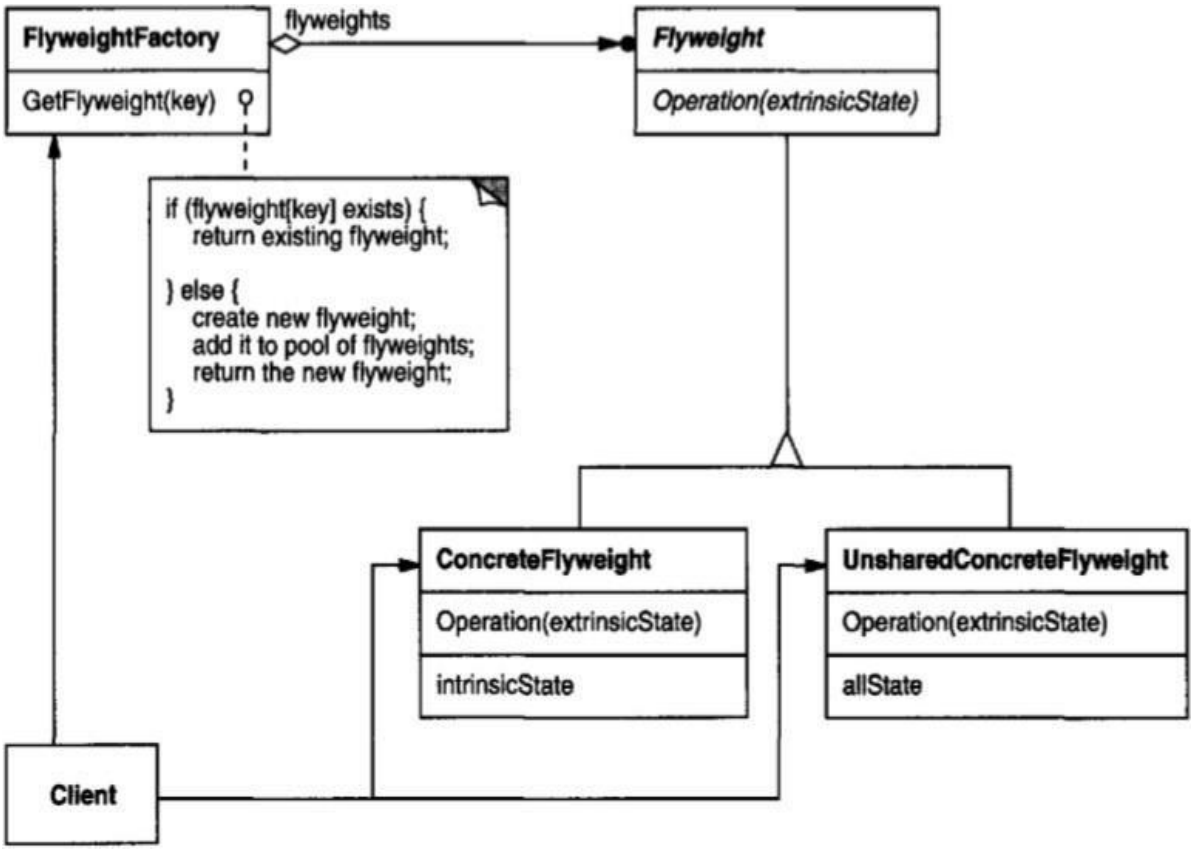
Applicability: Apply flyweight pattern when all of the following are true:

1. An application uses a large number of objects.
2. Storage costs are high because of the sheer quantity of objects.

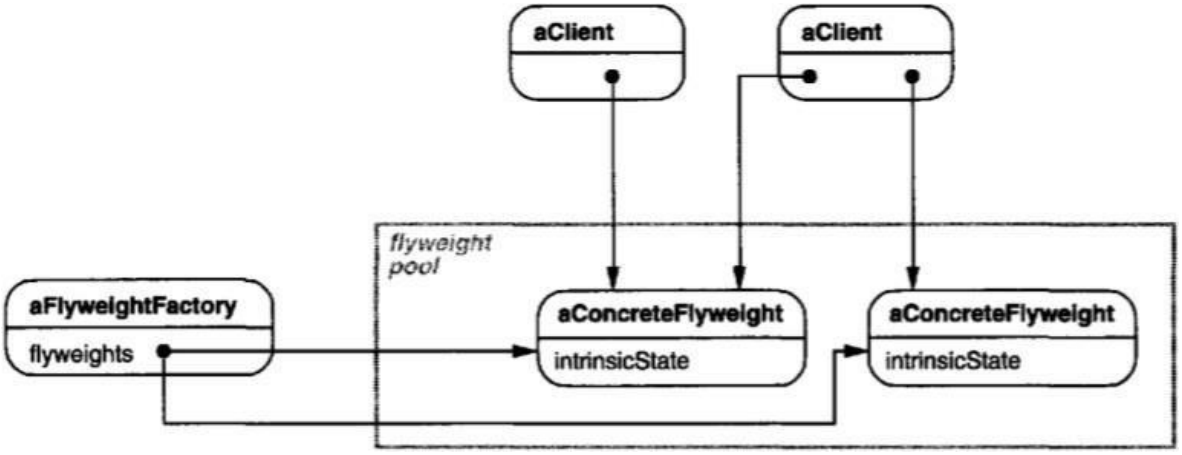
3. Most object state can be made extrinsic.
4. Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
5. The application doesn't depend upon object identity

Structure:

The structure of flyweight pattern is shown below:



The following object diagram shows how flyweights are shared:



Participants

1. Flyweight (Glyph) - declares an interface through which flyweights can receive and act on extrinsic state.
2. ConcreteFlyweight(Character): Implements the Flyweights interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable
3. UnsharedConcreteFlyweight(Row,column): Not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing, it doesn't enforce it.
4. FlyweightFactory: Creates and manages flyweight objects. Ensures that flyweights are shared properly.
5. Client: Maintains a reference to flyweight(s). Computes or stores the extrinsic state of flyweight(s).

Collaborations:

- ✓ State that a flyweight needs to function must be characterized as either intrinsic or extrinsic.
  - Intrinsic state is stored in the ConcreteFlyweight object;
  - extrinsic state is stored or computed by Client objects. Clients pass this state to the flyweight when they invoke its operations.

- ✓ Clients should not instantiate ConcreteFlyweights directly. Clients must obtain ConcreteFlyweight objects exclusively from the FlyweightFactory object to ensure they are shared properly.

Consequences:

Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state.

However, such costs are offset by space savings, which increase as more flyweights are shared.

Storage savings are a function of several factors:

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

Implementation: Following issues must be considered while implementing flyweight pattern:

1. **Removing extrinsic state.** The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing.

2. **Managing shared objects.** Because objects are shared, clients shouldn't instantiate them directly. FlyweightFactory lets clients locate a particular flyweight. FlyweightFactory objects often use an associative store to let clients look up flyweights of interest.

Sample Code

Returning to our document formatter example, we can define a Glyph base class for flyweight graphical objects. Logically, glyphs are Composite that have graphical attributes and can draw themselves. Here we focus on just the font attribute, but the same approach can be used for any other graphical attributes a glyph might have.

```
class Glyph {
public:
    virtual ~Glyph();

    virtual void Draw(Window*, GlyphContext&);

    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);

    virtual void First(GlyphContext&);
    virtual void Next(GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);

    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);
protected:
    Glyph();
};
```

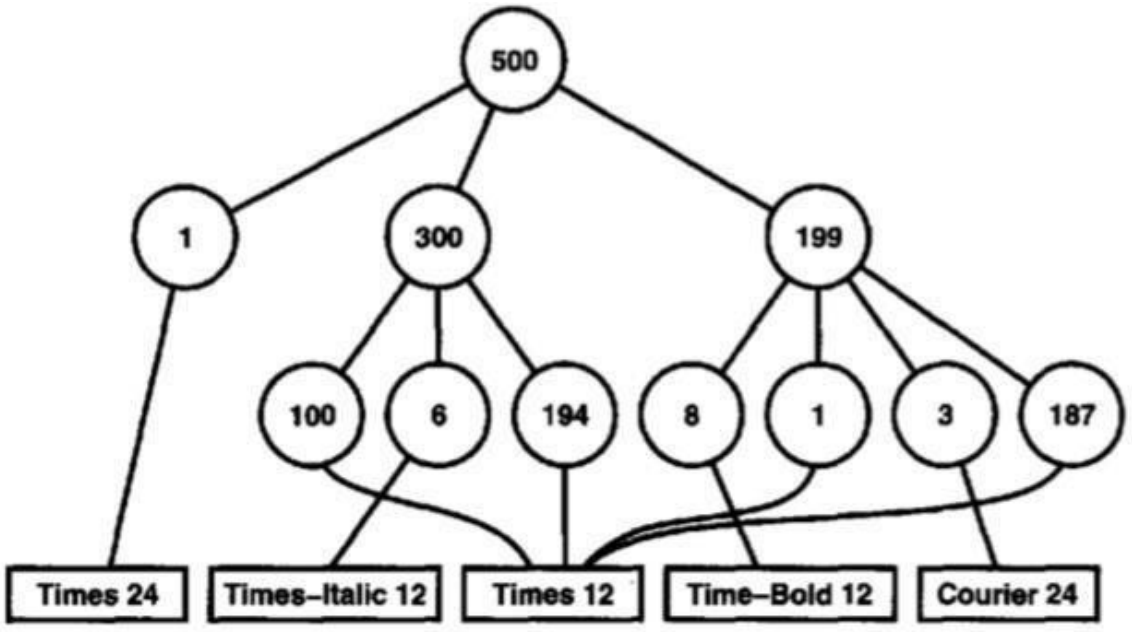
The Character subclass just stores a character code:

```
class Character : public Glyph {
public:
    Character(char);

    virtual void Draw(Window*, GlyphContext&);
private:
    char _charcode;
};
```

Glyph's child iteration and manipulation operations must update the GlyphContext whenever they're used.

```
class GlyphContext {
public:
    GlyphContext();
    virtual ~GlyphContext();

    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);
private:
    int _index;
    BTree* _fonts;
};
```

- GlyphContext must be kept informed of the current position in the glyph structure during traversal.
- GlyphContext : : Next increments index as the traversal proceeds
- GlyphContext : : GetFont uses the index as a key into a BTree structure that stores the glyph-to-font mapping.
- The BTree structure for font information might look like



Known Uses

1. The concept of flyweight objects was first describe d and explore d as a design technique in Interviews 3.0. Its developers built a powerful document editor called Doc as a proof of concept.
2. ET++ [WGM88] uses flyweights to support look-and-feel independence.The look-and-feel standard affects the layout of user interface elements (e.g., scroll bars, buttons, menus—known collectively as "widgets") and their decoration s (e.g., shadows, beveling).
3. The Layout objects are created and managed by Look objects. The Look class is an Abstract Factory that retrieves a specific Layout object with operations like GetButtonLayout, GetMenuBarLayout, and so forth .

Related patterns

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

Proxy Pattern(Object Structure)

Intent: To provide a surrogate or placeholder for another object to control access to it.

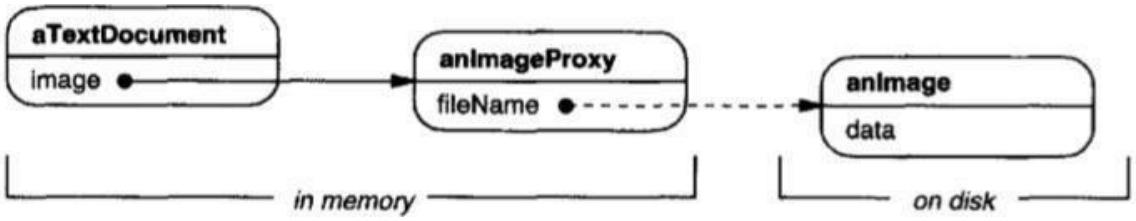
Also knows as: Surrogate

Motivation

- ✓ Consider a document editor that can embed graphical objects in a document.
- ✓ Some graphical objects, like large raster images, can be expensive to create.
- ✓ But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened.
- ✓ This isn't necessary anyway, because not all of these objects will be visible in the document at the same time.
- ✓ These constraints would suggest creating each expensive object on demand, which in this case occurs when an image becomes visible.

- But what do we put in the document in place of the image ?
- And how can we hide the fact that the image is created on demand so that we don't complicate the editor's implementation?

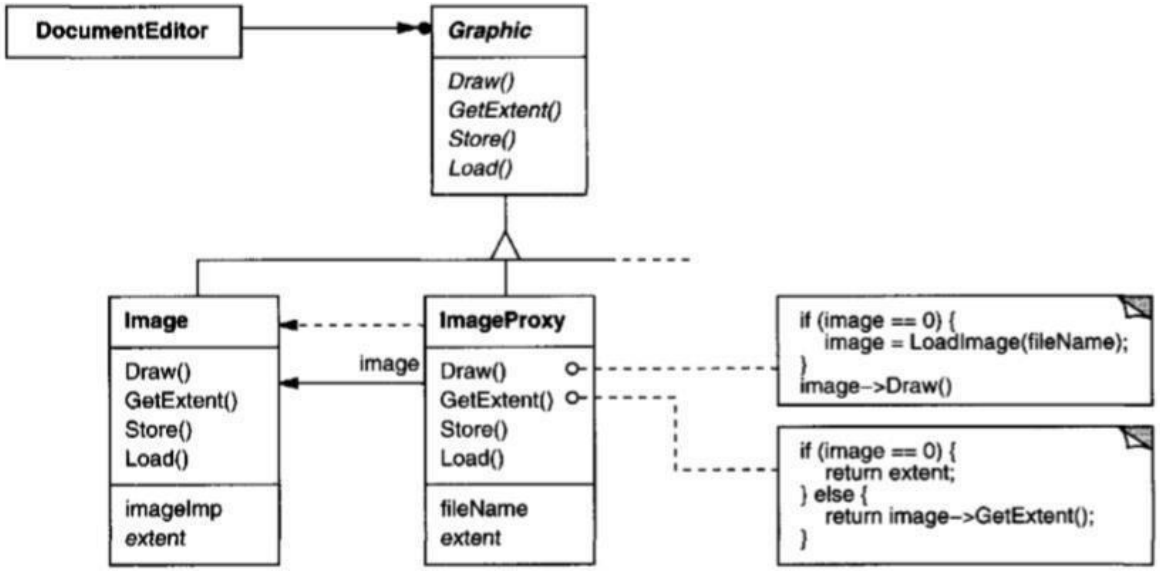
This optimization shouldn't impact the rendering and formatting code , **for example.** The solution is to use another object, an image proxy, that acts as a stand-in for the real image. The proxy acts just like the image and takes care of instantiating it when it's required



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

- Let's assume that images are stored in separate files .
- In this case we can use the file name as the reference to the real object.
- The proxy also stores its extent, that is, its width and height.
- The extent lets the proxy respond to requests for its size from the formatter without actually instantiating the image.

The following class diagram illustrates this example in more detail.



Applicability: Proxy pattern is applicable when:

1.A **remote proxy** provides a local representative for an object in a different address space.

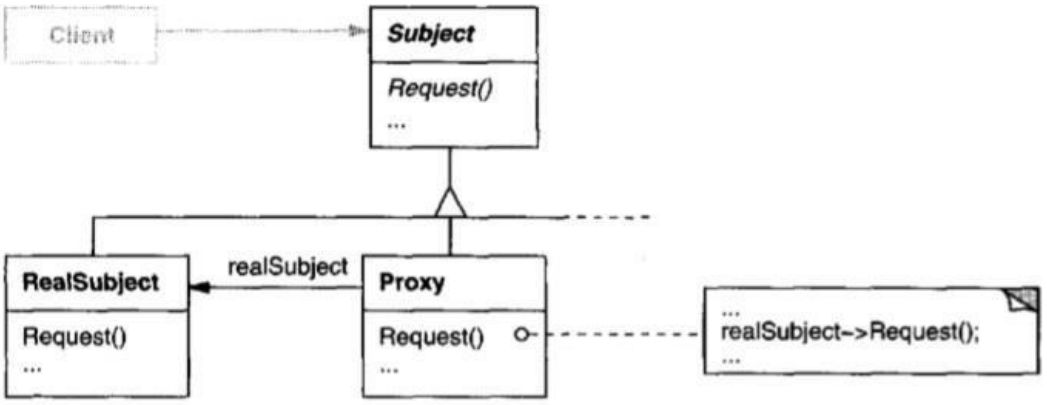
2.A **virtual proxy** creates expensive objects on demand.

3.A **protection proxy** controls access to the original object. Protection proxies are useful when objects should have different access rights.

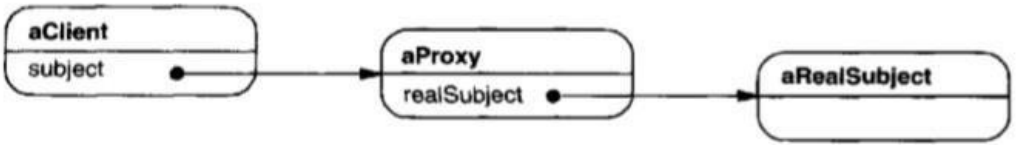
4. A **smart reference** is a replacement for a bare pointer that performs additional actions when an object is accessed.

- counting the number of references to the real object so that it can be freed automatically when there are no more references (also called smart pointers [Ede92]).
- loading a persistent object into memory when it's first referenced.
- checking that the real object is locked before it's accessed to ensure that no other object can change it.

Structure



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



Participants: The participants in proxy pattern are:

1. **Proxy (Image Proxy):** Maintains a reference that lets the proxy access the real subject. Provides an interface identical to Subject so that the Proxy can be substituted for the real subject. Controls access to the real subject.
   other responsibilities depend on the kind of proxy:
   a) remote proxies are responsible for encoding a request
   b) virtual proxies may cache additional information about the real subject so that they can postpone accessing it.
   c) protection proxies check that the caller has the access permissions required to perform a request.
2. **Subject(Graphics)**: Defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
3. **RealSubject(Image):** Defines the real object that the proxy represents.

Collaborations

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

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

1. A remote proxy can hide the fact that an object resides in a different address space.

2. A virtual proxy can perform optimizations such as creating an object on demand.

3. Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Implementation: The Proxy pattern can exploit the following language features:

1. Overloading the member access operator in C++: C++ supports overloading operator->, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced.

2. Using doesNotUnderstand in Smalltalk: Smalltalk provides a hook that you can use to support automatic forwarding of requests. Smalltalk calls doesNotUnderstand: aMessage when a client sends a message to a receiver that has no corresponding method. The Proxy class can redefine doesNotUnderstand so that the message is forwarded to its subject.

3. Proxy doesn't always have to know the type of real subject: If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubject classes uniformly. But if Proxies are going to instantiate RealSubjects (such as in a virtual proxy), then they have to know the concrete class.

Sample Code

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

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

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

The Image class implements the Graphic interface to display image files. Image overrides HandleMouse to let users resize the image interactively.

```
class Image : public Graphic {
public:
    Image(const char* file);  // loads image from a file
    virtual ~Image();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();


        virtual void Load(istream& from);
        virtual void Save(ostream& to);
    private:
        // ...
    };
```

ImageProxy has the same interface as Image:

```
    class ImageProxy : public Graphic {
    public:
        ImageProxy(const char* imageFile);
        virtual ~ImageProxy();

        virtual void Draw(const Point& at);
        virtual void HandleMouse(Event& event);

        virtual const Point& GetExtent();

        virtual void Load(istream& from);
        virtual void Save(ostream& to);
    protected:
        Image* GetImage();
    private:
        Image* _image;
        Point _extent;
        char* _fileName;
    };
```

The constructor save s a local copy of the name of the file that stores the image, and it initializes -extent and -image:

```
ImageProxy::ImageProxy (const char* fileName)  {
    _fileName = strdup(fileName);
    _extent = Point::Zero;  // don't know extent yet
    _image = 0;
}

Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}
```

The implementation of GetExtent returns the cache d extent if possible ; otherwise the image is loaded from the file.

```
const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
    return _extent;
}
```

```
void ImageProxy::Draw (const Point& at) {
    GetImage()->Draw(at);
}

void ImageProxy::HandleMouse (Event& event) {
    GetImage()->HandleMouse(event);
}
```

The Save operation saves the cache d image extent and the image file name to a stream. Load retrieves this information and initializes the corresponding members.

```
void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}
```

Finally, suppose we have a class TextDocument that can contain Graphic objects:

```
class TextDocument {
public:
    TextDocument();

    void Insert(Graphic*);
    // ...
};
```

We can insert an `ImageProxy` into a text document like this:

```
TextDocument* text = new TextDocument;
// ...
text->Insert(new ImageProxy("anImageFileName"));
```

Known Uses

1. The virtual proxy example in the Motivation section is from the ET++ text building block classes.
2. NEXTSTEP [Add94] uses proxies (instances of class NXProxy) as local representatives for objects that may be distributed.
3. McCullough [McC87] discusses using proxies in Smalltalk to access remote objects.
4. Pascoe [Pas86] describes how to provide side-effects on method calls and access control with "Encapsulators."

Related patterns:

1. An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.
2. Although decorators can have similar implementations as proxies, decorators have a different purpose. A decorator adds one or more responsibilities to an object, whereas a proxy controls access to an object.

Module 3

Behavioral Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. Behavioral patterns describe not just patterns of objects or classes but also the patterns of communication between them. These patterns characterize complex control flow that's difficult to follow at run-time.

Behavioral class patterns use inheritance to distribute behavior between classes. Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate toper form a task that no single object can carry out by itself.

Chain of Responsibility

Intent

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

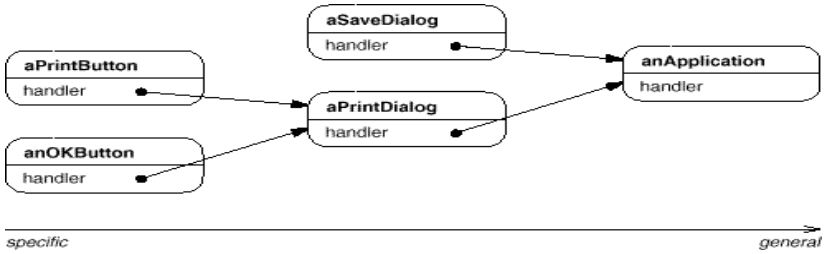
Motivation

Consider a context-sensitive **help** facility for a **graphical user interface**. The user can obtain help information on any part of the interface just by clicking on it. The help that's provided depends on the part of the interface that's selected and its context.

For example, **a button widget** in a **dialog box** might have different help information than a similar button in the main window. If no specific help information exists for that part of the interface, then the help system should display a more general help message about the immediate context the dialog box as a whole.

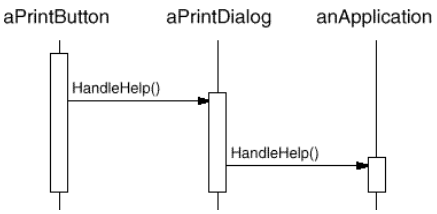
The problem here is that the object that ultimately provides the help isn't known explicitly to the object that initiates the help request. What we need is a way to decouple the button that initiates the help request from the objects that might provide help information. The Chain of Responsibility pattern defines how that happens.

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



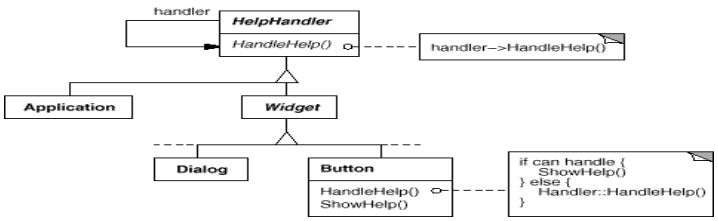
The first object in the chain receives the request and either handles it or forwards it to the next candidate on the chain, which does like wise. The object that made the request has no explicit knowledge of who will handle it—we say the request has an **implicit receiver**.

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



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

To forward the request along the chain, and to ensure receivers remain implicit, each object on the chain shares a common interface for handling requests and for accessing its **successor** on the chain. For example, the help system might define a HelpHandler classwith a corresponding HandleHelp operation. HelpHandler can be theparent class for candidate object classes, or it can be defined as amixin class. Then classes that want to handle help requests can makeHelpHandler a parent:



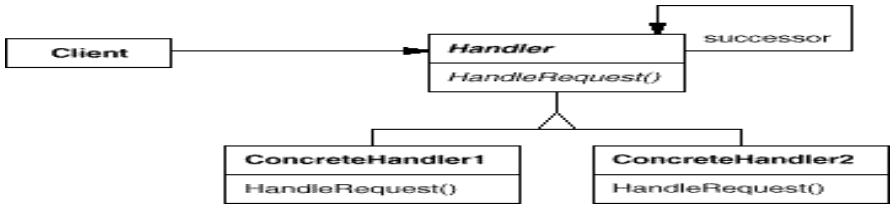
The Button, Dialog, and Application classes use HelpHandler operations to handle help requests. HelpHandler's HandleHelp operation forwards the request to the successor by default. Subclasses can override thisoperation to provide help under the right circumstances; otherwisethey can use the default implementation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

Applicability

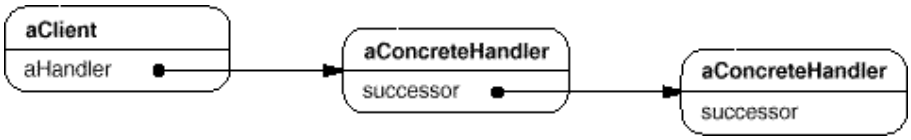
Use Chain of Responsibility when

- More than one object may handle a request, and the handler isn't knowna priori. The handler should be ascertained automatically.
- You want to issue a request to one of several objects withoutspecifying the receiver explicitly.
- The set of objects that can handle a request should be specifieddynamically.

Structure



A typical object structure might look like this:



Participants

- **Handler** (HelpHandler)
  - Defines an interface for handling requests.
  - (Optional) implements the successor link.
- **ConcreteHandler** (PrintButton, PrintDialog)
  - Handles requests it is responsible for.
  - Can access its successor.
  - If the ConcreteHandler can handle the request, it does so; otherwise it forwards the request to its successor.
- **Client**
  - Initiates the request to a ConcreteHandler object on the chain.

Collaborations

- When a client issues a request, the request propagates along the chainuntil a ConcreteHandler

object takes responsibility for handling it.

Consequences

Chain of Responsibility has the following benefits and liabilities:

1. **Reduced coupling**.The pattern frees an object from knowing which other object handles a request. An object only has to know that a request will be handled" appropriately." Both the receiver and the sender have no explicit knowledge of each other, and an object in the chain doesn't havetoknow about the chain's structure.

2. **Added flexibility in assigning responsibilities to objects**.Chain of Responsibility gives you added flexibility in distributing responsibilities among objects. You can add or change responsibilities for handling a request by adding to or otherwise changing the chain at run-time. You can combine this with subclassing to specialize handlers statically.

3. **Receipt isn't guaranteed**.Since a request has no explicit receiver, there'sno *guarantee*it'll be handled—the request can fall off the end of the chain without ever being handled. A request can also go unhandled whenthe chain is not configured properly.

Potential Drawbacks:

- Client can’t explicitly specify who handles a request
- No guarantee of request being handled (request falls off end of chain)

Implementation

Here are implementation issues to consider in Chain of Responsibility:

1. **Implementing the successor chain**.There are two possible ways to implement the successor chain:
   a. Define new links (usually in the Handler, but ConcreteHandlerscould define them instead).
   b. Use existing links.

   Our examples so far define new links, but often you can use existing object references to form the successor chain. For example, parent references in a part-whole hierarchy can define a part's successor. Awidget structure might already have such links.

   Using existing links works well when the links support the chain you need. It saves you from defining links explicitly and it saves space. But if the structure doesn't reflect the chain of responsibility your application requires, then you'll have to define redundant links.

2. **Connecting successors**.If there are no preexisting references for defining a chain, then we will introduce them ourself. In that case, the Handler not only defines the interface for the requests but usually maintains the successor as well. That lets the handler provide a default implementation of

HandleRequest that forwards the request to the successor. If a ConcreteHandler subclass isn't interested in the request, it doesn't have to override the forwarding operation, since its default implementation forwards unconditionally.

Here's a HelpHandler base class that maintains a successor link:

```
Class HelpHandler {
Public:
HelpHandler (HelpHandler* s):
_successor(s) { }
Virtual void HandleHelp ();
Private:
HelpHandler* _successor;
};
Void HelpHandler::HandleHelp ()
 {
if (_successor)
{
_successor->HandleHelp ();
} }
```

3. **Representing requests**: Different options are available for representing requests. In the simplest form, the request is a hard-coded operation invocation, as in the case of HandleHelp. This is convenient and safe, but you can forward only the fixed set of requests that the Handler class defines An alternative is to use a single handler function that takes a request code as parameter.This approach is more flexible, but it requires conditional statements for dispatching the request based on its code. Moreover, there's no type-safe way to pass parameters, so they must be packed and unpacked manually.Obviously this is less safe than invoking an operation directly.

To address the parameter-passing problem, we can use separate request objects that bundle request parameters. A Request class can represent requests explicitly, and new kinds of requests can be defined by subclassing. Subclasses can define different parameters.Handlers must know the kind of request to access these parameters.

To identify the request, Request can define an access or function that returns an identifier for the class. Alternatively, the receiver can use run-time type information if the implementation languages supports it.Here is a sketch of a dispatch function that uses request objects to identify requests.A GetKind operation defined in the base Request class identifies the kind of request:

```
Void Handler::HandleRequest (Request* theRequest)
{
Switch (theRequest->GetKind ())
{
Case Help:
// cast argument to appropriate type
HandleHelp((HelpRequest*)theRequest);break;
Case Print:
HandlePrint ((PrintRequest*) theRequest);
```

```
//        ...
break;
default:
//      ...
break
;}}
```

Subclasses can extend the dispatch by overridingHandleRequest. The subclass handles only the requests in which it's interested; other requests are forwarded to the parent class. In this way, subclasses effectively extend the HandleRequest operation.

4. Automatic forwarding in Smalltalk.You can use the doesNotUnderstand mechanism in Smalltalk to forward requests. Messages that have no corresponding methods are trapped in the implementation of doesNotUnderstand, which can be overridden to forward the message to anobject's successor.Thus it isn't necessary to implement forwarding manually; the classhandles only therequest in which it's interested and it relies on doesNotUnderstand to forward all others.

Sample Code

The following example illustrates how a chain of responsibility can handle requests for an on-line help system. The help request is an explicit operation. We'll use existing parent references in the widget hierarchy to propagate requests between widgets in the chain, and we'll define a reference in the Handler class to propagate help requests between non widgets in the chain.

The HelpHandler class defines the interface for handlinghelp requests. It maintains a help topic and keeps a reference to its successor on the chain of help handlers.The key operation is HandleHelp, which subclassesoverride. HasHelp is a convenience operation for checking whether there is an associated help topic.

```
Typedef int Topic;

const Topic NO_HELP_TOPIC = -1;
class HelpHandler {public:
HelpHandler(HelpHandler*=0,Topic= NO_HELP_TOPIC);
virtual bool HasHelp();
virtual void SetHandler(HelpHandler*, Topic);virtual void
HandleHelp();
private:

HelpHandler* _successor;

Topic _topic;

};
HelpHandler::HelpHandler (HelpHandler* h, Topic t ) : _successor(h), _topic(t)
{ }
BoolHelpHandler::HasHelp()
 {return_topic!= NO_HELP_TOPIC;
}
void HelpHandler::HandleHelp ()
{if (_successor != 0)
{
```

```
    successor->HandleHelp();
    }}
```

All widgets are subclasses of the Widget abstract class.Widget is a subclass of HelpHandler, since alluser interface elements can have help associated with them.

```
Class Widget: public HelpHandler {
Protected:
Widget (Widget* parent, Topic t = NO_HELP_TOPIC);private:
Widget* _parent;
};

Widget::Widget (Widget* w, Topic t);
HelpHandler (w, t) {
_parent = w;
}
```

Known Uses

Several class libraries use the Chain of Responsibility pattern to handle user events. They use different names for the Handler class, but the idea is the same:

- When the user clicks the mouse or presses akey, an event gets generated and passed along the chain.MacApp [App89] and ET++ [WGM88] call it "EventHandler,"Symantec'sTCL library [Sym93b] calls it "Bureaucrat," andNeXT's AppKit [Add94] uses thename "Responder."
- ET++ uses Chain of Responsibility to handle graphical update.

Related Patterns

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

Command

Intent

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

Also Known As

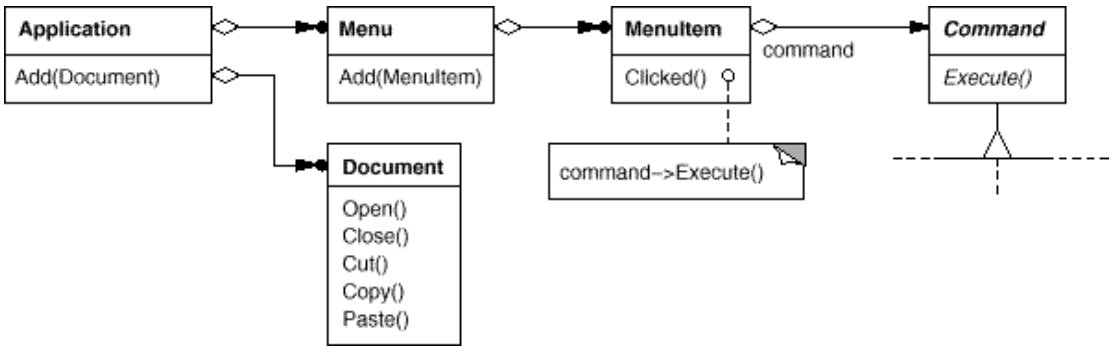
Action, Transaction

Motivation

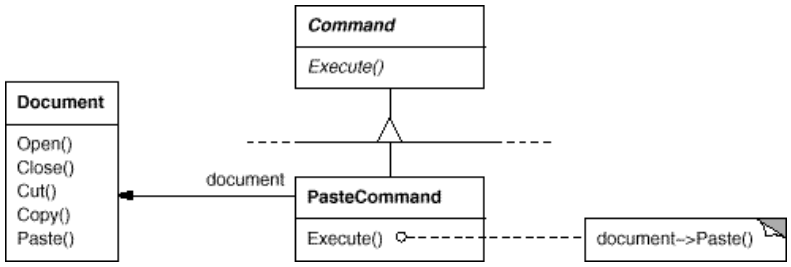
Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.

For example, user interface toolkits include objects like buttons and menus that carry out a request in response to user input.But the toolkit can't implement the request explicitly in the button or menu, because only applications that use the toolkit know what should be done on which object. As toolkit designers we have no way of knowing the receiver of the request or the operations that will carry it out.

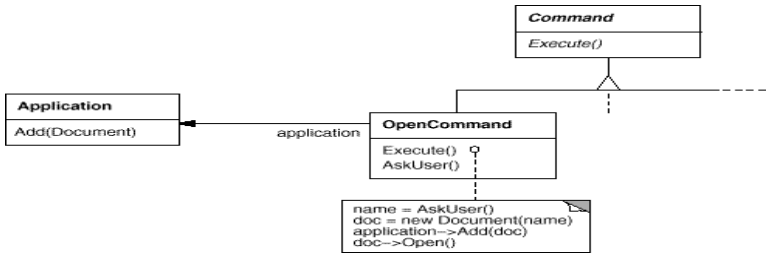
The Command pattern lets toolkit objects make requests of unspecified application objects by turning the request itself into an object. This object can be stored and passed around like other objects. The key to this pattern is an abstract Command class, which declares an interface for executing operations. In the simplest form this interface includes an abstract Execute operation. Concrete Command subclasses specify a receiver-action pair by storing the receiver as an instance variable and by implementing Execute to invoke the request. The receiver has the knowledge required to carry outthe request.



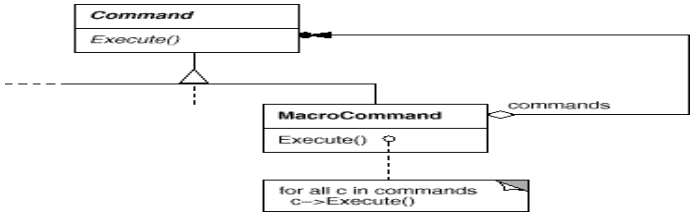
Menus can be implemented easily with Command objects. Each choice in a Menu is an instance of a MenuItem class. An Application class createsthese menus and their menu items along with the rest of the user interface.The Application class also keeps track of Document objects that a user hasopened. The application configures each MenuItem with an instance of a concrete Commandsubclass. When the user selects a MenuItem, theMenuItem calls Execute on its command, and Execute carries out theoperation. MenuItems don't know which subclass of Command they use.Command subclasses store the receiver of the request and invoke one or more operations on the receiver.



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



Sometimes a MenuItem needs to execute a sequence of commands.For example; a MenuItem for



centering a page at normal size could be constructed from a CenterDocumentCommand object and aNormalSize Command object. Because it's common to string commands together in this way, we can define a MacroCommand class to allow aMenuItem to execute an open-ended number ofcommands. Macro Command is a concrete Command subclass that simply executes a sequence

ofCommands. MacroCommand has no explicit receiver, because the commandsit sequences define their own receiver.

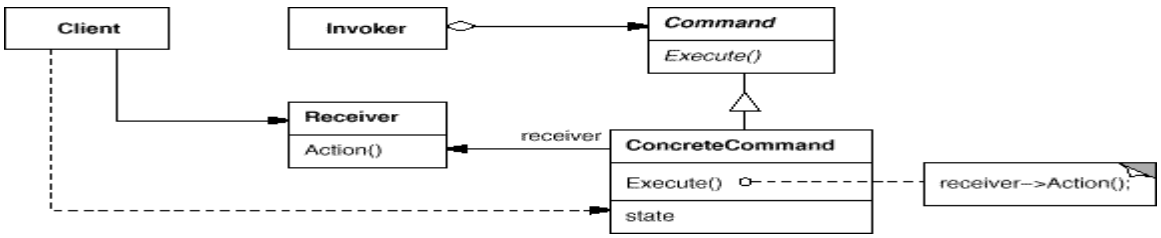
In each of these examples, notice how the Command pattern decouplesthe object that invokes the operation from the one having theknowledge to perform it. This gives us a lot of flexibility indesigning our user interfaceWe can replace commands dynamically, which would be useful for implementing context-sensitive menus. We can also support command scripting by composingcommands into larger ones. All of this is possible because the object that issues request only needs toknowhow to issue it; it doesn't need to know how the request will be carried out.

Applicability

Use the Command pattern when you want to

- **Parameterize objects to perform actions**.
- **Specify, queue, and execute requests at different times**.
- **Support undo**. The Command's Execute operation can store state for reversing its effects in the command itself.
- **Support logging changes**
- **Structure a system around high-level operations**

Participants



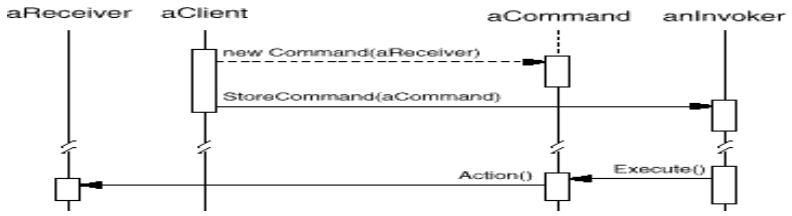
- **Command**

- Declares an interface for executing an operation.
  - **ConcreteCommand** (PasteCommand, OpenCommand)
- Defines a binding between a Receiver object and an action.
- Implements Execute by invoking the corresponding operation(s) onReceiver.
  - **Client** (Application)
- Creates a ConcreteCommand object and sets its receiver.
  - **Invoker** (MenuItem)
- asks the command to carry out the request.
  - **Receiver** (Document, Application)
- knows how to perform the operations associated with carrying outa request. Any

class may serve as a Receiver.

Collaborations

- The client creates a ConcreteCommand object and specifies its receiver.
- An Invoker object stores the ConcreteCommand object.
- The following diagram shows the interactions between these objects.It illustrates how Command decouples the invoker from the receiver (and the request it carries out).



Consequences

The Command pattern has the following consequences:

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

Implementation

Consider the following issues when implementing the Command pattern:

1. How intelligent should a command be?

   A command can have a wide range of abilities. At one extreme it merely defines a binding between a receiver and the actions that carryout the request. At the other extreme it implements everything itselfwithout delegating to a receiver at all.

2. Supporting undo and redo.

   Commands can support undo and redo capabilities if they provide a way to reverse their execution. A ConcreteCommand class might need to store additional state to do so. This state can include the Receiver object, which actually carries out operations inresponse to the request,

   - the arguments to the operation performed on the receiver, and
   - Any original values in the receiver that can changeas a result of handling the request. The receiver must provideoperations that let the command return the receiver to its prior state.

3 Avoiding error accumulation in the undo process.

4 Errors can accumulate as commands are executed, unexecuted, and reexecuted repeatedly so that an application's state eventually diverges from original values. It may be necessary therefore to store more information in the command to ensure that objects are restored to their original state.

5 Using C++ templates.For commands that

(1) Aren't undoable

(2) Don't require arguments, we can use C++ templates to avoid creating a Command subclass forevery kind of action and receiver.

Sample Code

The C++ code showed here sketches the implementation of the Command classesin the Motivation section. We'll define OpenCommand,PasteCommand, and MacroCommand. First theabstract Command class:

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

Open Command opens a document whose name is supplied by theuser. An Open Command must be passed an Application object in its constructor. Ask User is an implementation routine that prompts the user for the name of the document to open.

```
class OpenCommand : public Command {
public:OpenCommand(Application*);
virtual void Execute(); protected:
virtual const char* AskUser();private:
Application* _application;
char* _response;
};
OpenCommand::OpenCommand (Application* a) {
_application = a;
}
```

Known Uses

- Perhaps the first example of theCommand pattern appears in a paper by **Lieberman**[Lie85].
- **MacApp [App89**] popularizedthe notion of commands for implementing undoable operations.
- **ET++ [WGM88], InterViews [LCI+92], andUnidraw [VL90]** alsodefine classes that follow theCommand pattern.
- **InterViews** define Action abstract class thatprovides command functionality. It also defines an ActionCallbacktemplate, parameterized by action method that cans instantiate commandsubclasses automatically.
- The **THINK** class library [Sym93b] also uses commands to support undoable actions. Commands in THINK are called "Tasks
- **Unidraw's** command objects are unique in that they can behave likemessages.
- Coplien describes how to implement **functors**, objects thatare functions, in C++[Cop92].

Related Patterns

A Composite (183)can be used to implement MacroCommands.
A Memento (316)can keep state the command requires to undo its effect.

A command that must be copied before being placed on the historylist acts as aPrototype (133).

Interpreter

Intent

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

Motivation

If a particular kind of problem occurs often enough, then it might beworthwhileto express instances of the problem as sentences in a simple language. Then you can build an interpreter that solves theproblem by interpreting these sentences.

For example, searching for strings that match a pattern is a common problem. Regular expressions are a standard language for specifying patterns of strings. Rather than building custom algorithms to match each pattern against strings, search algorithms could interpret a regular expression that specifies a set of strings to match.

The Interpreter pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences.

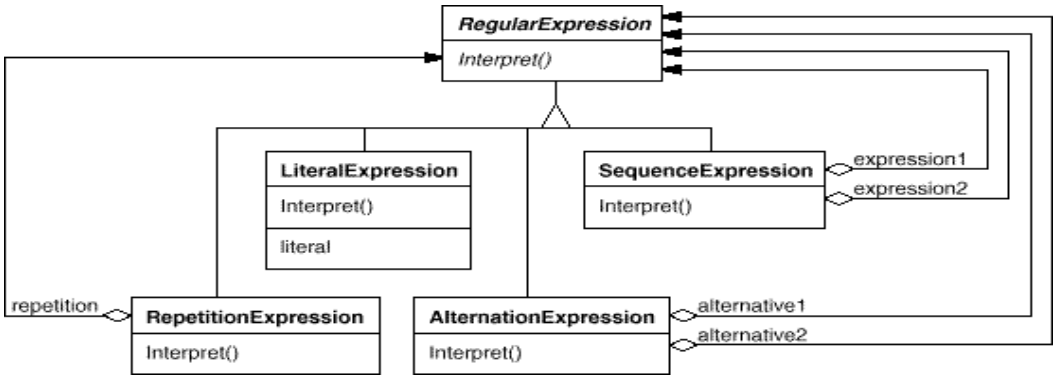
Example: the pattern describes how to define a grammar for regular expressions, represent aparticular regular expression, and how to interpret that regular expression.

Suppose the following grammar defines the regular expressions:

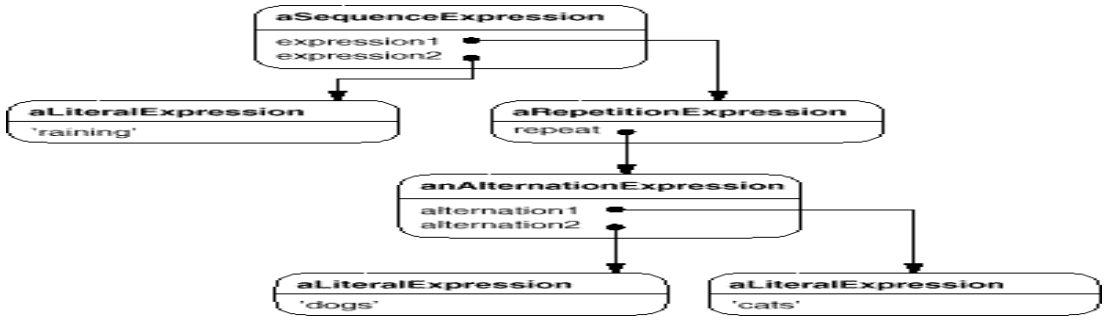
expression::= literal | alternation | sequence | repetition |
'(' expression ')'
alternation::= expression '|' expression
sequence ::= expression '&' expression
repetition ::= expression '*'
literal::= 'a' | 'b' | 'c' | ... { 'a' | 'b' | 'c' | ... }*

The symbol expression is the start symbol, and literalis a terminal symbol defining simple words.

The Interpreter pattern uses a class to represent each grammar rule.Symbols on the right-hand side of the rule are instance variables of these classes. The grammar above is represented by five classes: an abstract class Regular Expression and its four subclassesLiteralExpression, AlternationExpression, SequenceExpression, and Repetition Expression. The last three classes define variables that hold sub expressions.



Every regular expression defined by this grammar is represented by an abstract syntax tree made up of instances of these classes. For example, the abstract syntax tree represents the regular expression Traning & (dogs|cats)*



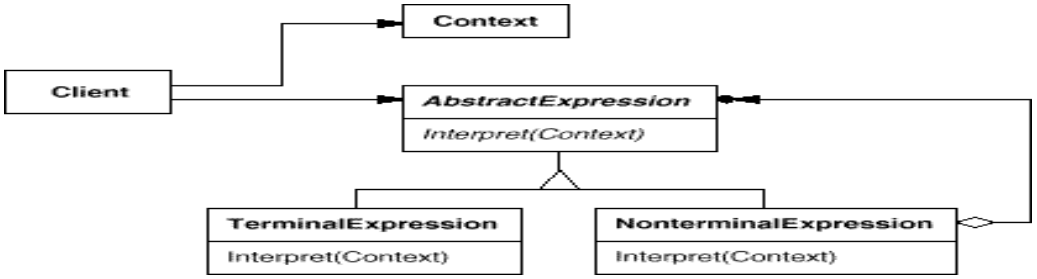
We can create an interpreter for these regular expressions by defining the Interpret operation on each subclass of Regular Expression.Interpret takes as an argument the context in which to interpret the expression. The context contains the input string and information on how much of it has been matched so far. Each subclass of Regular Expression implements Interpret to match the next part of the input string based on the current context

Applicability

Use the Interpreter pattern when there is a language to interpret, andyou canrepresent statements in the language as abstract syntax trees.The Interpreterpattern works best when

- The grammar is simple.
- Efficiency is not a critical concern. Structure

Participants



- **AbstractExpression** (RegularExpression)
  - Declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.
- **TerminalExpression** (LiteralExpression)
  - Implements an Interpret operation associated with terminal symbols in the grammar.
  - an instance is required for every terminal symbol in a sentence.
- **NonterminalExpression** (AlternationExpression,RepetitionExpression, SequenceExpressions)
  - One such class is required for every rule R ::= R1 R2 ... Rn in the grammar.
  - Maintains instance variables of type AbstractExpression for each of the symbols R1 through Rn.
  - Implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.
- **Context**
  - Contains information that's global to the interpreter.
- **Client**
  - Builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines.
  - Invokes the Interpret operation.

Collaborations

- The client builds (or is given) the sentence as an abstract syntaxtree of NonterminalExpression and TerminalExpression instances. Thenthe client initializes the context and invokes the Interpretoperation.
- Each NonterminalExpression node defines Interpret in terms ofInterpret on each subexpression. The Interpret operation of eachTerminalExpression defines the base case in the recursion.
- The Interpret operations at each node use the context tostore and access the state of the interpreter.

Consequences

The Interpreter pattern has the following benefits and liabilities:

1. It's easy to change and extend the grammar.
2. Implementing the grammar is easy too.
3. Complex grammars are hard to maintain.
4. Adding new ways to interpret expressions.

Implementation

The following issues are specific to Interpreter:

1. **Creating the abstract syntax tree**.The Interpreter pattern doesn't explain how to create an abstract syntax tree. In other words, it doesn't address parsing.The abstract syntax tree can be created by a table-driven parser, by a hand-crafted parser, or directly by the client.
2. **Defining the Interpret operation**.You don't have to define the Interpret operation in the expression classes. If it's common to create a new interpreter, then its better to use the Visitor pattern to put Interpret in aseparate "visitor" object.
3. **Sharing terminal symbols with the Flyweight pattern**.Grammars whose sentences contain many occurrences of a terminal symbol might benefit from sharing a single copy of that symbol. Terminal nodes generally don't store information about their position in the abstract syntax tree. Parent nodes pass them whatever context they need during interpretation. Hence there is a distinction between shared state and passed-in state, and the Flyweight pattern applies.

Sample Code

Here are two examples.

- The first is a complete example in Small talk for checking whether a sequence matches a regular expression.
- The second is a C++ program for evaluating Boolean expressions.

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

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

For example, the regular expression

(('dog ' | 'cat ') repeat & 'weather') matches the input string "dog dog cat weather".

To implement the matcher, we define the five classes The classSequence Expression has instance variables expression1 and expression2 for its children in the abstract syntax tree. Alternation Expression stores its alternatives in the instance variables alternative1 and alternative2, while Repetition Expression holds the expression it repeats in itsrepetition instance variable.LiteralExpression has a components instance variable that holds a list of. These represent the literal string that must match the input sequence.

The match: operation implements an interpreter for the regular expression. Each of the classes defining the abstract syntax tree implements this operation. It takes input State as an argument representing the current stateof the matching process, having read part of the input string.

This current state is characterized by a set of input streams representing the set of inputs that the regular expression could have accepted so far. The current state is most important to the repeat operation.

Output state usually contains more states than its input state, because a RepetitionExpression can match one, two, or many occurrences of repetition on the input state. The outputstates represent all these possibilities, allowing subsequent elementsof the regular expression to decide which state is the correct one.

Finally, the definition of match: forLiteralExpression tries to match its components against eachpossible input stream. It keeps only those input streams that have amatch:

The nextAvailable: message advances the input stream. This is the only match: operation that advances the stream.Notice how the state that's returned contains a copy of the inputstream, thereby ensuring that matching a literal never changes theinput stream. This is important because each alternative of anAlternationExpression should see identical copies ofthe input stream.

Known Uses

- The Interpreter pattern is widely used in compilers implemented with object-oriented languages, as the Smalltalk compilers are.
- SPECTalkuses the pattern to interpret descriptions of input fileformats [Sza92].
- The QOCA constraint-solving toolk it uses it to evaluate constraints [HHMV92].

Related Patterns

- Composite (183): The abstract syntax tree is an instance of the Composite pattern.
- Flyweight (218) shows how to share terminal symbols within the abstract syntaxtree.
- Iterator (289): The interpreter can use an Iterator to traverse the structure.
- Visitor (366) canbe used to maintain the behavior in each node in the abstract syntaxtree in one class.

Iterator

Intent

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

Also Known As

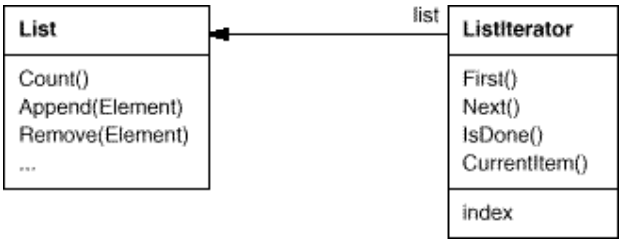
Cursor

Motivation

An aggregate object such as a list should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish.

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

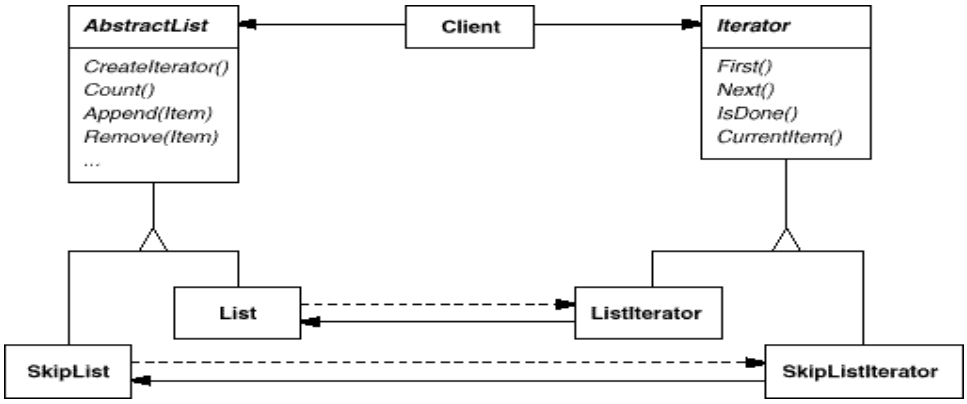
For example, a List class would call for a List Iterator with the following relationship between them:



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

Notice that the iterator and the list are coupled and the client mustknow thatit is a list that's traversed as opposed to some otheraggregate structure. Hence the client commits to a particularaggregate structure. It would be better if we could change the aggregateclass without changing client code. We can do this by generalizing the iterator concept to support **polymorphic iteration**.

We define an AbstractList class that provides a common interfacefor manipulating lists. Similarly, we need an abstract Iteratorclass that defines a common iteration interface. Then we can defineconcrete Iterator subclasses for the different list implementations.As a result, the iteration mechanism becomes independent of concreteaggregate classes.



The remaining problem is how to create the iterator. Since we want to write code that's independent of the concrete List subclasses, we cannot simply instantiate a specific class. Instead, we make the listobjects responsible for creating their corresponding iterator. This requires an operation like CreateIterator through which clients request an iterator object.

Applicability

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregatestructures

Structure

Participants

- **Iterator**

- Defines an interface for accessing and traversing elements.

- **ConcreteIterator**

- Implements the Iterator interface.
- Keeps track of the current position in the traversal of the aggregate.

- **Aggregate**

- Defines an interface for creating an Iterator object.

- **ConcreteAggregate**

- Implements the Iterator creation interface to return an instanceof the proper ConcreteIterator.

Collaborations

- A ConcreteIterator keeps track of the current object inthe aggregate and can compute the succeeding object in thetraversal.

Consequences

The Iterator pattern has three important consequences:

1. **It supports variations in the traversal of an aggregate**.Complex aggregates may be traversed in many ways. For example, codegeneration and semantic checking involve traversing parse trees. Codegeneration may traverse the parse tree inorder or preorder.Iterators makes it easy to change the traversal algorithm.
2. **Iterators simplify the Aggregate interface**.Iterator's traversal interface obviates the need for a similarinterface in Aggregate, thereby simplifying the aggregate's interface.
3. **More than one traversal can be pending on an aggregate**.An iterator keepstrack of its own traversal state. Therefore you canhave more than onetraversal in progress at once.

Implementation

Iterator has many implementation variants and alternatives. Someimportant onesfollow. The trade-offs often depend on thecontrol structures your language provides.

1. **Who controls the iteration**? A fundamental issue is deciding which partycontrols the iteration,the iterator or the client that uses the iterator.When the clientcontrols the iteration, the iterator is called an **externaliterator**, and when the iterator controls it, the iterator is an**internal iterator**.Clients that use anexternal iterator must advance the traversal and request the nextelement explicitly fromthe iterator. In contrast, the client handsan internal iterator an operation to perform, and the iterator applies that operation to every element in the aggregate.

2. **External iterators are more flexible than internal iterators**. It's easy to compare two collections for equality with an external iterator, for example, but it's practically impossible with internaliterators. Internal iterators are especially weak in a language likeC++ that does not provide anonymous functions, closures, orcontinuations like Smalltalk and CLOS. But on the other hand,internal iterators are easier to use, because they define the iterationlogic for you.

3. **Who defines the traversal algorithm**? The iterator is not the only place wherethe traversal algorithm canbe defined. The aggregate might define thetraversal algorithm anduse the iterator to store just the state of theiteration. We callthis kind of iterator a **cursor**, since it merely pointstothe current position in the aggregate. A client will invoke the Nextoperation on the aggregate with the cursor as an argument, and theNext operation will change the state of thecursor.
4. **If the iterator is responsible for the traversal algorithm**, then it'seasy to use different iteration algorithms on the same aggregate, andit can also be easier to reuse the same algorithm on differentaggregates. On the other hand, the traversal algorithm might need toaccess the private variables of the aggregate. If so, putting thetraversal algorithm in the iterator violates the encapsulation of theaggregate.

5. **How robust is the iterator?**It can be dangerous to modify an aggregate while you're traversing it.If elements are added or deleted from the aggregate,you might end upaccessing an element twice or missing it completely. A simplesolution is to copy the aggregate and traverse the copy, but that'stoo expensive to do in general.
6. **A robust iterator ensures that insertions and removalswon't interfere with traversal**, and it

eidoes it without copying theaggregate. There are many ways to implement robust iterators. Mostrely on registering the iterator with the aggregate. On insertion orremoval, the aggregate ther adjusts the internal state of iteratorsit has produced, or it maintains information internally to ensureproper traversal.

7. **Kofler provides a good discussion of how robust iterators areimplemented in ET++** [Kof93]. Murray discusses theimplementation of robust iterators for the USL StandardComponents'List class [Mur93].

8. **Additional Iterator operations**.The minimal interface to Iterator consists of the operations First,Next, IsDone, and CurrentItem.4Someadditional operations might prove useful. For example, orderedaggregates can have a Previous operation that positions the iteratorto the previous element. A SkipTo operation is useful for sorted orindexed collections. SkipTo positions the iterator to an objectmatching specific criteria.
9. **Using polymorphic iterators in C++.**Polymorphic iterators have their cost.They require the iterator object to be allocated dynamically by a factory method. Hence they should be used only when there's a need for polymorphism. Otherwise use concrete iterators, which can be allocated on the stack.

10. **Iterators may have privileged access**.An iterator can be viewed as an extension of the aggregate that created it. The iterator and the aggregate are tightly coupled. We can express this close relationship in C++ by making the iterator afriend of its aggregate. Then you don't need todefine aggregate operations whose sole purpose is to let iteratorsimplement traversal efficiently.

11. **Iterators for composites**.External iterators can be difficult to implementover recursiveaggregate structures like those in the Composite (183) pattern, because a position in the structure may span many levels ofnested aggregates. Therefore an external iterator has to store a paththrough the Composite to keep track of the current object. Sometimesit's easier just to use an internal iterator. It can record thecurrent position simply by calling itself recursively, thereby storingthe path implicitly in the call stack.

12. **Null iterators**.A NullIterator is a degenerate iterator that's helpful for handling boundary conditions.

By definition, a NullIterator is always done with traversal; that is, its Is Done operation always evaluates to true.NullIterator can make traversing tree-structured aggregates (likeComposites) easier.

Sample Code

We'll look at the implementation of a simple List class.We'll show two Iterator implementations, onefor traversing the List infront-to-back order, and another for traversing back-to-front

1. **List and Iterator interfaces**. First let's look at the part of the Listinterface that's relevant toimplementing iterators. for the full interface.

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

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

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

3. *Using the iterators.*Let's assume we have a List of Employee objects,and we would like to print all the contained employees. TheEmployee class supports this with a Prin toperation. To print the list, we define a PrintEmployees operation that takes an iterator as an argument. It uses the iteratorto traverse and print the list.

4. **Avoiding commitment to a specific list implementation**. Let's consider how a skiplist

variation of List would affectour iteration code. A SkipList subclass ofList must provide a SkipListIterator thatimplements the Iterator interface. Internally, theSkipListIterator has to keep more than just an index todo the iteration efficiently. But sinceSkipListIterator conforms to theIterator interface, the PrintEmployees operationcan also be used when the employees are stored in a SkipListobject.

5. **Making sure iterators get deleted**. To make life easier for clients,we'll provide an IteratorPtr that acts as a proxy for aniterator. It takes care of cleaning up the Iterator objectwhen it goes out of scope.
6. **IteratorPtr** is always allocated on thestack C++ automatically takes care of callingits destructor, which deletes the real iterator.IteratorPtr overloads bothoperator-> andoperator* in such a way that an IteratorPtr can betreated just like a pointer to an iterator. The members ofIteratorPtr are all implemented inline; thus they can incur nooverhead.
7. **An internal ListIterator**.As a final example, let's look at a possible implementation of aninternal or passive ListIterator class. Here the iteratorcontrols the iteration and it applies an operation to each element.

Known Uses

Iterators are common in object-oriented systems. Most collection class libraries offer iterators in one form or another.

Example:

Booch components [Boo94], apopular collection class library. It provides both a fixed size and dynamically growing implementation of aqueue..

Polymorphic iterators and the cleanup Proxy described earlier are provided by the ET++ container classes [WGM88].

ObjectWindows 2.0 [Bor94] provides a class hierarchy ofiterators for containers. You can iterate over different containertypes in the same way. The ObjectWindow iteration syntax relies onoverloading the postincrement operator ++ to advance theiteration.

Related Patterns

Composite (183):Iterators are often applied to recursive structures suchasComposites.

Factory Method (121):Polymorphic iterators rely on factory methods to instantiate theappropriate Iterator subclass.

Memento (316) isoften used in conjunction with the Iterator pattern. An iteratorcan use a memento to capture the state of an iteration. The iteratorstores the memento internally.

Mediator

Intent

Define an object that encapsulates how a set of objects interact.Mediator promotesloose coupling by keeping objects from referring toeach other explicitly, and it lets you vary their interactionindependently.

Motivation

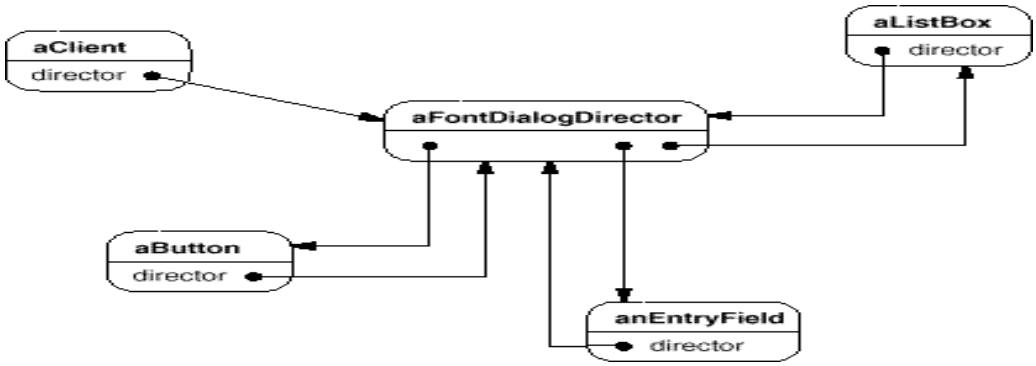
Object-oriented design encourages the distribution of behavioramong objects. Suchdistribution can result in an object structurewith many connections between objects; in the worst case, every objectends up knowing about every other.

As an example, consider the implementation of dialog boxes in agraphical userinterface. A dialog box uses a window to present acollection of widgets such asbuttons, menus, and entry fields, asshown here:

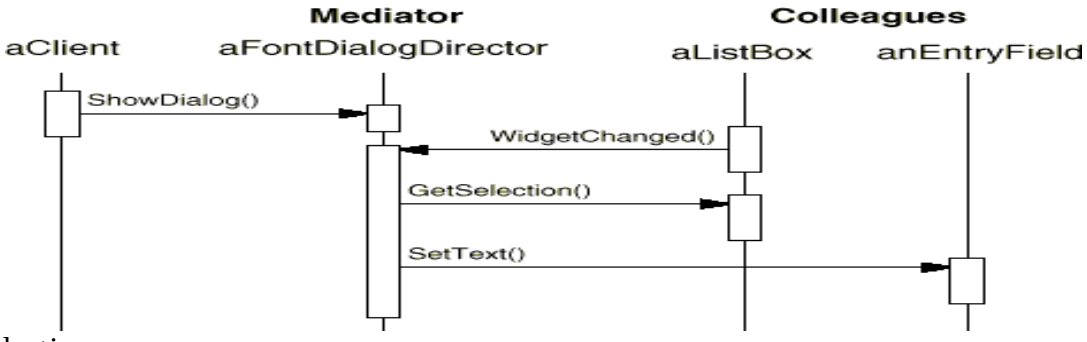
Often there are dependencies between the widgets in the dialog. Forexample, abutton gets disabled when a certain entry field is empty.Selecting an entry i**a** list of choices called a **list box**might change the

contents of an entry field.Conversely, typing textinto the entry field might automatically select one or morecorresponding entries in the list box. Once text appears in the entryfield, other buttons may become enabled that let the user do somethingwith the text,such as changing or deleting the thing to which it refers.

For example, **FontDialogDirector** can be the mediatorbetween the widgets in a dialog box. A FontDialogDirector object knowsthe widgets in a dialog and coordinatestheir interaction. It acts as a hub of communication for widgets:



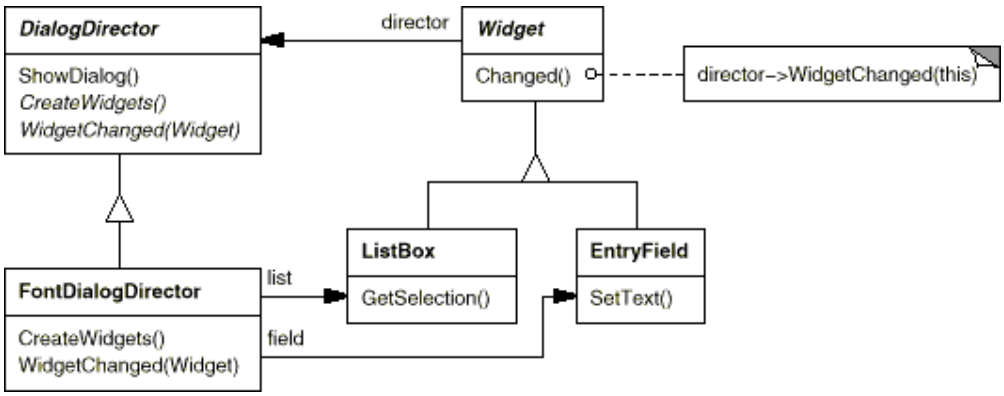
The following interaction diagram illustrates how the objects cooperate to handle a change in a list box's selection:



Here's the succession of events by which a list box's selection passesto an entry field:

1. The list box tells its director that it's changed.
2. The director gets the selection from the list box.
3. The director passes the selection to the entry field.
4. Now that the entry field contains some text, the directorenables button(s)for initiating an action (e.g., "demibold," "oblique").

Here's how the FontDialogDirector abstraction can be integrated into aclasslibrary:



DialogDirector is an abstract class that defines the overall behavior ofa dialog. Clients call the ShowDialog operation to display the dialog onthe screen.

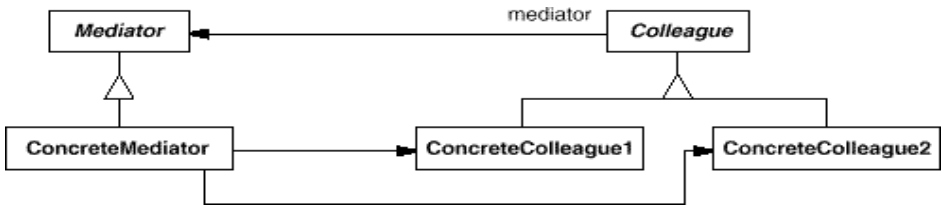
CreateWidgets is an abstract operation for creating thewidgets of a dialog. WidgetChanged isanother abstract operation; widgets call it to inform their director that they have changed.DialogDirector subclasses override CreateWidgets to create the properwidgets, and they override WidgetChanged to handle the changes.

Applicability

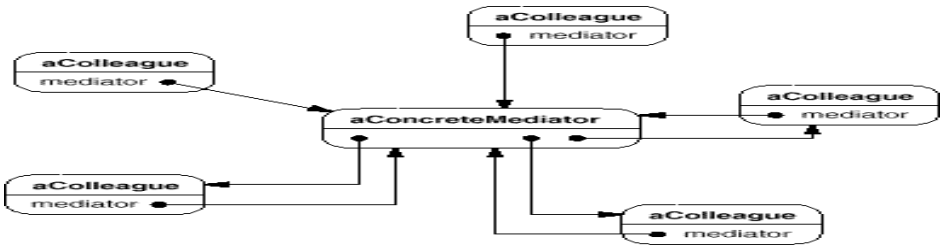
Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways. Theresulting interdependencies are unstructured and difficult tounderstand.
- reusing an object is difficult because it refers to and communicates with many other objects.
- a behavior that's distributed between several classes should becustomizable without a lot of subclassing.

Structure



A typical object structure might look like this:



Participants

- **Mediator** (DialogDirector)

o defines an interface for communicating with Colleague objects.

- **ConcreteMediator** (FontDialogDirector)

o implements cooperative behavior by coordinating Colleague objects.

o knows and maintains its colleagues.

- **Colleague classes** (ListBox, EntryField)

o each Colleague class knows its Mediator object.

o each colleague communicates with its mediator whenever it would haveotherwise communicated with another colleague.

Collaborations

- Colleagues send and receive requests from a Mediator object. Themediatorimplements the cooperative behavior by routing requestsbetween the appropriate colleague(s).

Consequences

The Mediator pattern has the following benefits and drawbacks:

1. It limits subclassing
2. It decouples colleagues.
3. It simplifies object protocols.
4. It abstracts how objects cooperate.
5. It centralizes control.

Implementation

The following implementation issues are relevant to the Mediatorpattern:

1. Omitting the abstract Mediator class.
2. Colleague-Mediator communication.

Sample Code

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

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

Widget is the abstract base class for widgets. Awidget knows its director.

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

Known Uses

Both ET++ [WGM88] and the THINK C class library [Sym93b] use director-like objects in dialogs as mediators between widgets.

Related Patterns

Facade (208) differs from Mediator in that it abstracts a subsystem of objects to provide a more convenient interface. Its protocol is unidirectional; that is, Facade objects make requests of the subsystem classes but notvice versa. In contrast, Mediator enables cooperative behaviorthat colleague objects don't or can't provide, and the protocol ismultidirectional.

Colleagues can communicate with the mediator using the Observer (326) pattern.

Memento

Intent

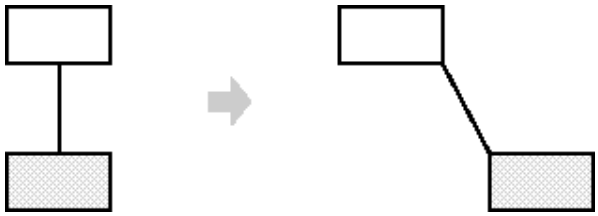
Without violating encapsulation, capture and externalize an object'sinternal state so that the object can be restored to this state later.

Also Known As

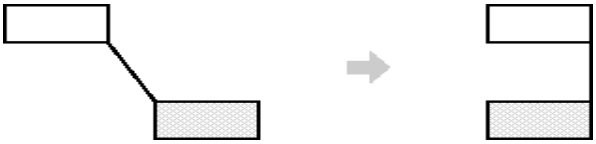
Token

Motivation

Sometimes it's necessary to record the internal state of an object.This is required when implementing checkpoints and undo mechanisms that let users back out of tentative operations or recover from errors. You must save state information somewhere so that you can restore objects to their previous states. Consider for example a graphical editor that supports connectivity between objects.A user can connect two rectangles with a line, and the rectangles stay connectedwhen the user moves either of them. The editor ensures that the line stretchesto maintain the connection.



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



In general, the ConstraintSolver's public interface might beinsufficient to allow precise reversal of its effects on otherobjects. The undo mechanism must work more closely withConstraintSolver to reestablish previous state, but we should alsoavoid exposing the ConstraintSolver's internals to the undo mechanism.

We can solve this problem with the Memento pattern. A memento is an object that stores a snapshot of theinternal state of another object — the memento's originator. The undo mechanism will request a mementofrom the originator when it needs to checkpoint the originator'sstate. The originator initializes the memento with information thatcharacterizes its current state. Only the originator can

store andretrieve information from the memento—the memento is "opaque" toother objects.

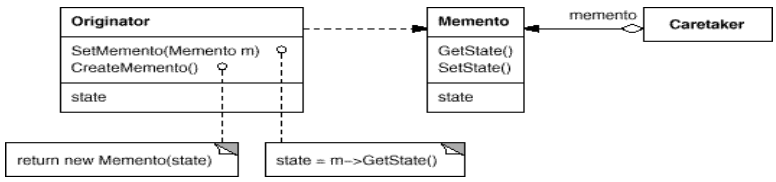
1. In the graphical editor example just discussed, the ConstraintSolver can actas an originator. The following sequence of events characterizes theundo process:
2. The editor requests a memento from the ConstraintSolver as aside-effect of the move operation.
3. The ConstraintSolver creates and returns a memento, an instance of aclass SolverState in this case. A SolverState memento contains datastructures that describe the current state of the ConstraintSolver'sinternal equations and variables.
4. Later when the user undoes the move operation, the editor gives theSolverState back to the ConstraintSolver.
5. Based on the information in the SolverState, the ConstraintSolverchanges its internal structures to return its equations and variablesto their exact previous state.

Applicability

Use the Memento pattern when

- a snapshot of (some portion of) an object's state must be saved sothat itcan be restored to that state later, *and*
- a direct interface to obtaining the state would exposeimplementation details and break the object's encapsulation.

Structure



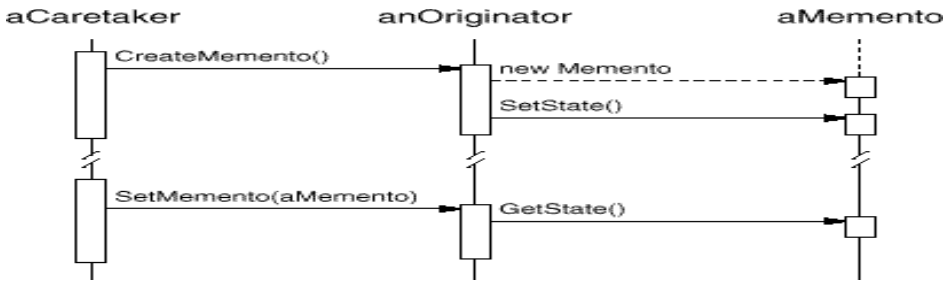
Participants

- **Memento** (SolverState)
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - Protects against access by objects other than the originator.
  - Mementos have effectively two interfaces. Caretaker sees a narrow interface to the Memento—it can only pass the memento to other objects. Originator, in contrast, sees a wide interface,
- **Originator** (ConstraintSolver)
  - creates a memento containing a snapshot of its current internalstate.
  - uses the memento to restore its internal state.
- **Caretaker** (undo mechanism)

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

Collaborations

- A caretaker requests a memento from an originator, holds it for atime, andpasses it back to the originator, as the followinginteraction diagram illustrates:



Sometimes the caretaker won't pass the memento back to the originator, because the originator might never need to revert to an earlier state.

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

Consequences

The Memento pattern has several consequences:

1. **Preserving encapsulation boundaries**.Memento avoids exposing information that only an originator shouldmanage but that must be stored neverthelessoutside the originator.The pattern shields other objects from potentiallycomplex Originatorinternals, thereby preserving encapsulation boundaries.
2. **It simplifies Originator**.In other encapsulation-preserving designs, Originator keeps theversions of internal state that clients have requested. That puts allthe storage management burden on Originator. Having clientsmanage the state they ask for simplifies Originator and keepsclientsfrom having to notify originators when they're done.
3. **Using mementos might be expensive**.Mementos might incur considerable overhead if Originator must copylarge amounts of information to store inthe memento or if clientscreate and return mementos to the originator often enough. Unlessencapsulating and restoring Originator state is cheap, the patternmight not be appropriate. See the discussion of incrementality in theImplementation section.
4. **Defining narrow and wide interfaces**.It may be difficult in some languages to ensure that only theoriginator can access the memento's state.
5. **Hidden costs in caring for mementos**.A caretaker is responsible for deleting the mementos it cares for.However, the caretaker has no idea how much stateis in the memento.Hence an otherwise lightweight caretaker might incur largestoragecosts when it stores mementos.

Implementation

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

1. *Language support.*Mementos have two interfaces: a wide one for originatorsand a narrowone for other objects. Ideally the implementation languagewillsupport two levels of static protection. C++ lets you do this bymakingthe Originator a friend of Memento and making Memento's wideinterfaceprivate. Only the narrow interface should be declaredpublic. For example:

```
class State;
class Originator {
public:
Memento* CreateMemento();
void SetMemento(const Memento*);
//      ...
private:
State* _state;
// internal data structures
// ...
};
class Memento
{public:
// narrow public interface
virtual          ~Memento();
private:
// private members accessible only to Originator
friend class Originator;
Memento();
void SetState(State*);
State* GetState();
//      ...
private:
State* _state;
// ...
};
```

2. ***Storing incremental changes.***When mementos get created and passed back to their originator in apredictable sequence, then Memento can save just the*incrementalchange* to the originator's internal state.

Sample Code

The graphical editor calls the command's Execute operationto move a graphical object and Unexecute to undo the move.The command stores its target, the distance moved, and an instance ofConstraintSolverMemento, a memento containing state from theconstraint solver.

```
class Graphic;
// base class for graphical objects in the graphical editor
class     MoveCommand     {
public:
MoveCommand(Graphic* target, const Point& delta);
void Execute();
Void          Unexecute();
private:
```

```
ConstraintSolverMemento*
_state;Point _delta;
Graphic* _target;
};
```

Known Uses

The preceding sample code is based on Unidraw's support for connectivitythroughits CSolver class [VL90].

Collections in Dylan [App92] provide an iteration interface thatreflects theMemento pattern.

Dylan's collections have the notion of a"state" object, which is a memento that represents the state of theiteration. Each collection can represent the current state of theiteration in any way it chooses; the representation is completelyhidden from clients.

The memento-based iteration interface has two interesting benefits:

1. More than one state can work on the same collection.
2. It doesn't require breaking a collection's encapsulationto support iteration. The memento is only interpreted by thecollection itself; no one else has access to it. Other approaches to iteration require breaking encapsulation by making iterator classesfriends of their collection classes. The situation is reversed in thememento-basedimplementation: Collection is a friend of theIteratorState.

The QOCA constraint-solving toolkit stores incremental information inmementos[HHMV92].

Related Patterns

Command (263): Commands can use mementos to maintainstate for undoable operations.

Iterator (289): Mementoscan be used for iteration as described earlier.

.

Observer

Intent

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

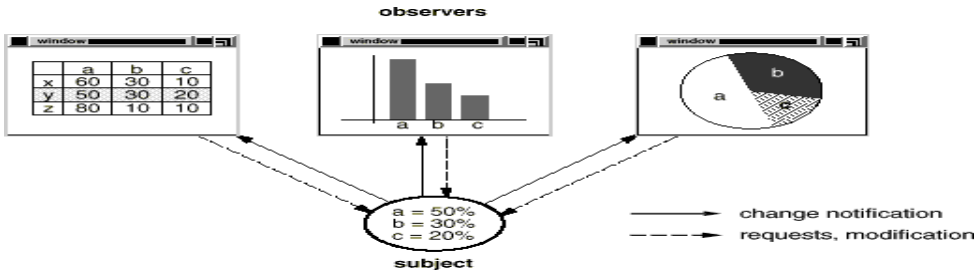
Also Known As

Dependents, Publish-Subscribe

Motivation

A common side-effect of partitioning a system into a collection ofcooperatingclasses is the need to maintain consistency betweenrelated objects. You don'twant to achieve consistency by making theclasses tightly coupled, because thatreduces their reusability.

Ex:Classes defining application data and presentations can be reusedindependently. They can work together, too. Both a spreadsheet objectand bar chart object can depict information in the same application dataobject using different presentations. The spreadsheet and the bar chartdon't know about each other, thereby letting you reuse only the one youneed. But they behave as though they do. When the user changes theinformation in the spreadsheet, the bar chart reflects the changesimmediately, and vice versa.

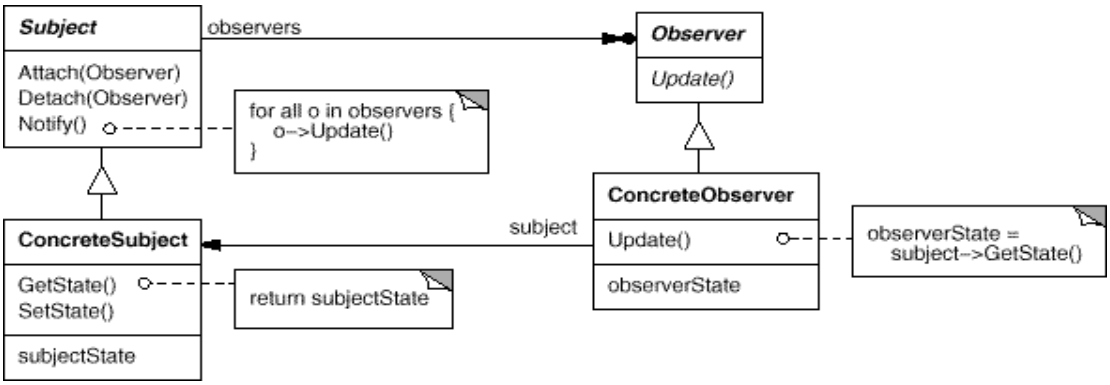


Applicability

Use the Observer pattern in any of the following situations:

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

Structure



Participants

- **Subject**

- knows its observers. Any number of Observer objects may observe asubject.
- provides an interface for attaching and detaching Observer objects.

- **Observer**

- defines an updating interface for objects that should be notifiedof changes in a subject.

- **ConcreteSubject**

- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.

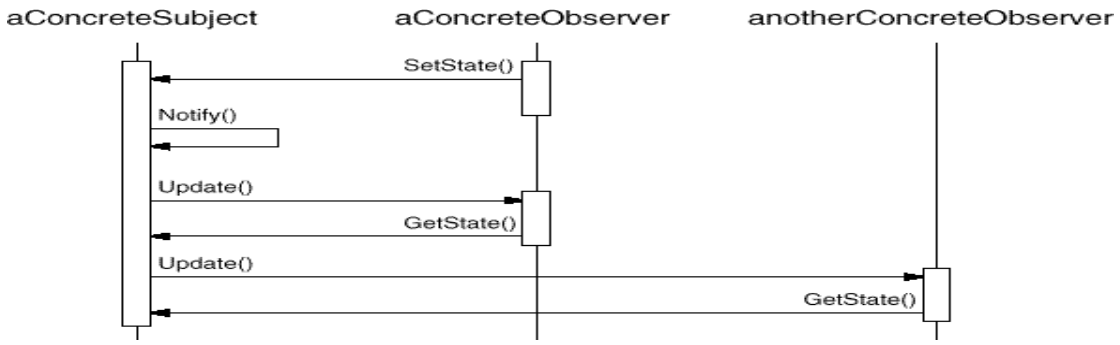
- **ConcreteObserver**

- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its stateconsistent with the subject's.

Collaborations

- ConcreteSubject notifies its observers whenever a changeoccurs that couldmake its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, aConcreteObserverobject may query the subject for information.ConcreteObserver uses this information to reconcile its state with thatof the subject.

The following interaction diagram illustrates the collaborationsbetweena subject and two observers:



Note how the Observer object that initiates the change requestpostponesits update until it gets a notification from the subject.Notify is not alwayscalled by the subject. It can be called by anobserver or by another kindof object entirely. The Implementationsection discusses some common variations.

Consequences

The Observer pattern lets you vary subjects and observersindependently. You can reuse subjects without reusing theirobservers, and vice versa. It lets you addobservers withoutmodifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include thefollowing:

1. **Abstract coupling between Subject and Observer**
2. **Support for broadcast communication**.
3. **Unexpected updates**.

Implementation

Several issues related to the implementation of the dependencymechanism arediscussed in this section.

1. *Mapping subjects to their observers.*The simplest way for a subject to keeptrack of the observers itshould notify is to store references to themexplicitly in thesubject. an associative look-up to maintainthesubject-to-observer mapping.
2. *Observing more than one subject.*It might make sense in some situations foran observer to depend onmore than one subject. For example, a spreadsheetmay depend on morethan one data source.
3. *Who triggers the update?*The subject and its observers rely on the notification mechanism tostay

consistent. But what object actually callsNotify to trigger theupdate? Here are two options:

a. Have state-setting operations on Subject call Notify after theychange the subject's state. The advantage of this approach isthatclients don't have to remember to call Notify on the subject. Thedisadvantage is that several consecutive operations will causeseveral consecutive updates, which may be inefficient.

b. Make clients responsible for calling Notify at the right time.Theadvantage here is that the client can wait to trigger the updateuntilafter a series of state changes has been made, therebyavoiding needless intermediate updates. The disadvantage is thatclients havean added responsibility to trigger the update. Thatmakes errors morelikely, since clients might forget to call Notify.

4. *Dangling references to deleted subjects.*Deleting a subject should notproduce dangling references in itsobservers. One way to avoid danglingreferences is to make thesubject notify its observers as it is deleted sothat they can resettheir reference to it. In general, simply deleting theobservers is not an option, because other objects may reference them, orthey may beobserving other subjects as well.

5. *Making sure Subject state is self-consistent beforenotification.*It's important to make sure Subject state is self-consistent beforecallingNotify, because observers query the subject for its currentstate in thecourse of updating their own state.

6. This self-consistency rule is easy to violate unintentionally whenSubject subclass operations call inherited operations. For example,the notification in the following code sequence is trigged when thesubject is in an inconsistent state:

```
void MySubject::Operation (int newValue) {
BaseClassSubject::Operation(newValue);
// trigger notification
_myInstVar += newValue;
// update subclass state (too late!)
}
```

7. *Avoiding observer-specific update protocols: the pushand pull models.* Implementations of the Observer pattern often havethe subject broadcastadditional information about the change. Thesubject passes this information as an argument to Update. The amountof information may varywidely.

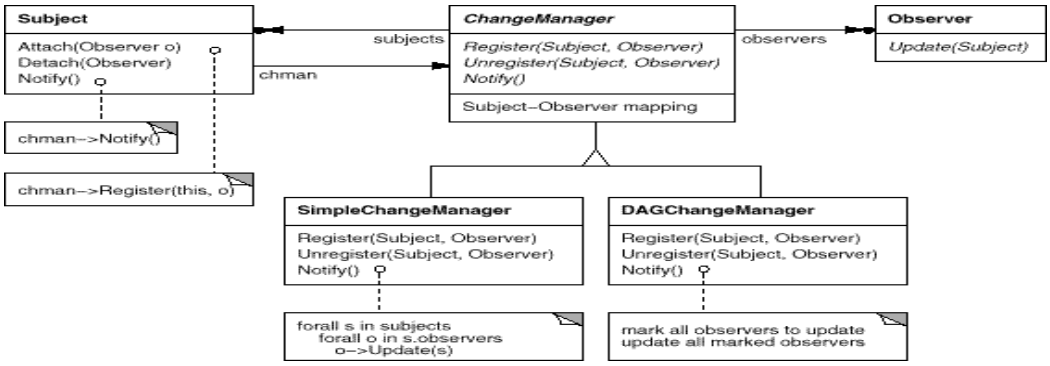
At one extreme, which we call the **push model**, the subjectsends observersdetailed information about the change, whether theywant it or not. At theother extreme is the **pull model**;the subject sends nothing but the mostminimal notification, andobservers ask for details explicitly thereafter.

8. *Specifying modifications of interest explicitly.*You can improve updateefficiency by extending the subject'sregistration interface to allow registering observers only forspecific events of interest. When such anevent occurs, the subjectinforms only those observers that have registeredinterest in thatevent. One way to support this uses the notion of**aspects**for Subject objects.
9. *Encapsulating complex update semantics.*When the dependency relationshipbetween subjects and observers isparticularly complex, an object thatmaintains these relationships mightbe required. We call such an object a**ChangeManager**. Itspurpose is to minimize the work required to make observers reflect achange in their subject.

ChangeManager has three responsibilities:

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

The following diagram depicts a simple ChangeManager-based implementation ofthe Observer pattern. There are two specialized ChangeManagers.SimpleChangeManager is naive in that it always updates allobservers ofeach subject. In contrast, DAGChangeManager handles directed-acyclicgraphs of dependencies between subjects and their observers. ADAGChangeManager is preferable to a SimpleChangeManager whenan observerobserves more than one subject. In that case, a change in twoor moresubjects might cause redundant updates. The DAGChangeManager ensuresthe observer receives just one update. SimpleChangeManager isfinewhen multiple updates aren't an issue.



ChangeManager is an instance of the Mediator (305) pattern. In general there is only one ChangeManager, and it is knownglobally. The Singleton (144)pattern would beuseful here.

10. *Combining the Subject and Observer classes.*Class libraries written in languages that lack multiple inheritance(like Smalltalk) generally don'tdefine separate Subject and Observerclasses but combine their interfacesin one class. That lets youdefine an object that acts as both a subject and an observer withoutmultiple inheritance. In Smalltalk, for example,the Subject andObserver interfaces are defined in the root class Object,making themavailable to all classes.

Sample Code

An abstract class defines the Observer interface:

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

This implementation supports multiple subjects for each observer. Thesubjectpassed to the Update operation lets the observerdetermine which subject changedwhen it observes more than one.

Known Uses

- The first and perhaps best-known example of the Observer pattern appearsin
- Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalkenvironment [KP88]. MVC's Model class plays the role ofSubject, whileView is the base class for observers. Smalltalk,ET++ [WGM88], and the THINK classlibrary [Sym93b] provide ageneral dependency mechanism by putting Subject and Observer interfacesin the parent class for all other classes in the system.
- Other user interface toolkits that employ this pattern areInterViews [LVC89], the AndrewToolkit [P+88], and Unidraw [VL90]. InterViewsdefines Observer andObservable classes explicitly.Andrew calls them "view" and "dataobject," respectively. Unidrawsplits graphical editor objects into View (forobservers) and Subjectparts.

Related Patterns

Mediator (305): Byencapsulating complex update semantics, the ChangeManager actsasmediator between subjects and observers.

Singleton (144):The ChangeManager may use the Singleton pattern to make ituniqueand globally

accessible.

State

Intent

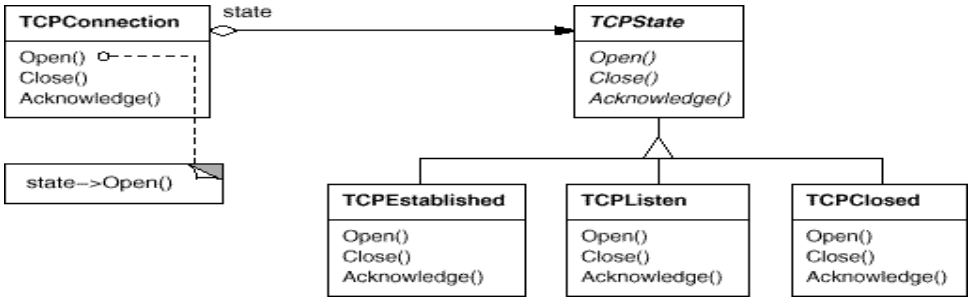
Allow an object to alter its behavior when its internal state changes.The object will appear to change its class.

Also Known As

Objects for States

Motivation

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



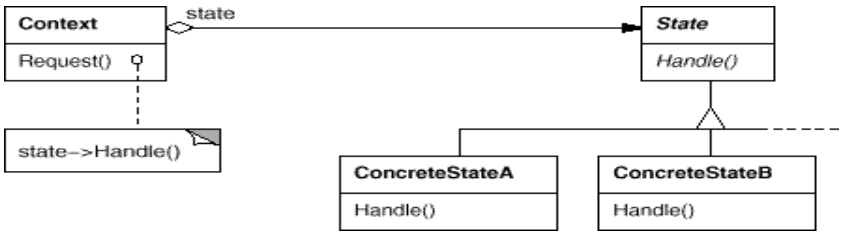
The class TCPConnection maintains a state object that represents the current state of the TCPconnection. The class TCPConnection delegates all state-specificrequests to this state object.TCPConnection uses its TCPStatesubclass instance to perform operations particularto the state of theconnection.

Whenever the connection changes state, the TCPConnection objectchanges the state object it uses. When the connection goes fromestablished to closed, for example, TCPConnection will replace itsTCPEstablished instance with a TCPClosed instance.

Applicability

Use the State pattern in either of the following cases:

- An object's behavior depends on its state, and it must change itsbehavior at run-time depending on that state.
- Operations have large, multipart conditional statements that depend onthe object's state. Structure



Participants

- **Context** (TCPConnection)
  - Defines the interface of interest to clients.
  - Maintains an instance of a ConcreteState subclass that defines thecurrent state.
  - **State** (TCPState)
  - Defines an interface for encapsulating the behavior associated with aparticular state of the Context.
  - **ConcreteState subclasses** (TCPEstablished, TCPListen, TCPClosed)
  - Each subclass implements a behavior associated with a state ofthe Context.

Collaborations

- Context delegates state-specific requests to the currentConcreteStateobject.
- A context may pass itself as an argument to the State objecthandling therequest. This lets the State object accessthe context if necessary.
- Either Context or the ConcreteState subclasses can decide whichstatesucceeds another and under what circumstances.

Consequences

The State pattern has the following consequences:

1. It localizes state-specific behavior and partitionsbehavior for different state
2. It makes state transitions explicit.
3. State objects can be shared.

Implementation

The State pattern raises a variety of implementation issues:

1. **Who defines the state transitions?** The State pattern does not specify which participant defines the criteria for state transitions. If the criteria are fixed, then they can be implemented entirely in the Context. It is generally more flexible and appropriate, however, to let the State subclasses themselves specify their successor state and when to make thetransition. This requires adding an interface to the Context that lets State objects set the Context's current state explicitly.
2. **A table-based alternative**.In *C++ Programming Style* , Cargilldescribes another way to impose structure on state-driven code:Heuses tables to map inputs to state transitions. For each state, atablemaps every possible input to a succeeding state. In effect,this approach converts conditional code into a table look-up.

   The main advantage of tables is their regularity: You can change thetransition criteria by modifying data instead of changing programcode.There are some disadvantages, however:

   - A table look-up is often less efficient than a (virtual)functioncall.
   - Putting transition logic into a uniform, tabular format makesthetransition criteria less explicit and therefore harder tounderstand.
   - It's usually difficult to add actions to accompany the statetransitions. The table-driven approach captures the states andtheirtransitions, but it must be augmented to perform arbitrarycomputationon each transition.
3. **Creating and destroying State objects**.A common implementation trade-off worth considering is whether(1) to create State objects only when they areneeded and destroy themthereafter versus (2) creating them ahead of timeand neverdestroying them.
4. **Using dynamic inheritance**.Changing the behavior for a particular request could be accomplishedby changing the object's class at run-time, but this is not possiblein most object- oriented programming languages. Objects in Selfcan delegate operations to other objects to achieve *aform* of dynamic inheritance. Changing the delegation target atrun-time effectively changesthe

inheritance structure. Thismechanism lets objects change their behavior and amounts to changingtheir class.

Sample Code

The following example gives the C++ code for the TCP connectionexample described in the Motivation section. This example is asimplified version of the TCP protocol; it doesn't describe thecomplete protocol or all the states of TCPconnections.[8]

First, we define the class TCPConnection, which provides aninterface fortransmitting data and handles requests to change state.

```
class TCPOctetStream; class
TCPState;
Class TCPConnection {
public:
        TCPConnection();
        void ActiveOpen(); void
        PassiveOpen();      void
        Close();
        void Send();
        void Acknowledge();void
        Synchronize();
```

TCPConnection keeps an instance of the TCPStateclass in the _state member variable. The classTCPState duplicates the state-changing interface ofTCPConnection. Each TCPState operation takes aTCPConnection instance as a parameter, lettingTCPState access data from TCPConnection andchange the connection's state.

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

Known Uses

- Johnson and Zweig [JZ91] characterize theState pattern and its application toTCP connection protocols.
- This technique is used in both the HotDraw [Joh92] and Unidraw [VL90] drawingeditor frameworks. It allows clients to define new kinds of tools easily. In HotDraw, the DrawingController class forwards the requests to the current Tool object.In Unidraw, the

corresponding classes are Viewer and Tool.

- Coplien's Envelope-Letter idiom [Cop92] is related toState. Envelope-Letter isa technique for changing an object's class atrun-time. The State pattern is more specific, focusing on how to dealwith an object whose behavior depends on its state.

Related Patterns

The Flyweight (218) pattern explains when and how State objects can be shared.
State objects are often Singletons (144).

Strategy

Intent

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

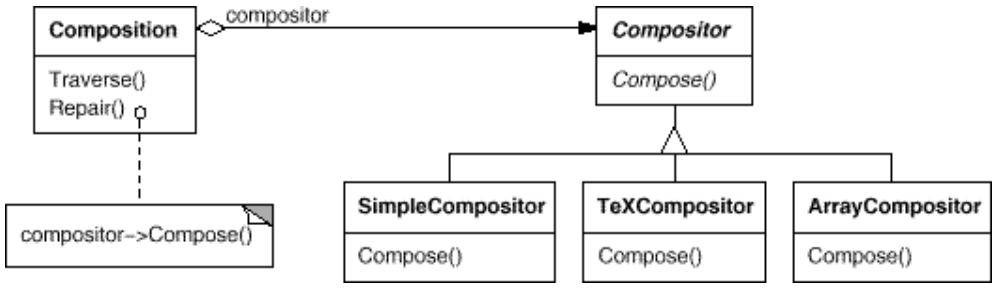
Also Known As

Policy

Motivation

Many algorithms exist for breaking a stream of text into lines.Hard-wiring allsuch algorithms into the classes that require themisn't desirable for severalreasons:

- Clients that need linebreaking get more complex if they includethelinebreaking code. That makes clients bigger and harder to maintain,especially if they support multiple linebreaking algorithms.



Suppose a Composition class is responsible for maintaining andupdating the linebreaks of text displayed in a text viewer.Linebreaking strategies aren't implemented by the class Composition.Instead, they are implemented separatelyby subclasses of the abstractCompositor class. Compositor subclasses implementdifferent strategies:

- **SimpleCompositor**implements a simple strategy that determines linebreaksone at atime.
- **TeXCompositor**implements the TeX algorithm for finding linebreaks. Thisstrategytries to optimize linebreaks globally, that is, one paragraph at atime.
- **ArrayCompositor**implements a strategy that selects breaks so that each rowhas a fixednumber of items. It's useful for breaking a collection of iconsintorows, for example.

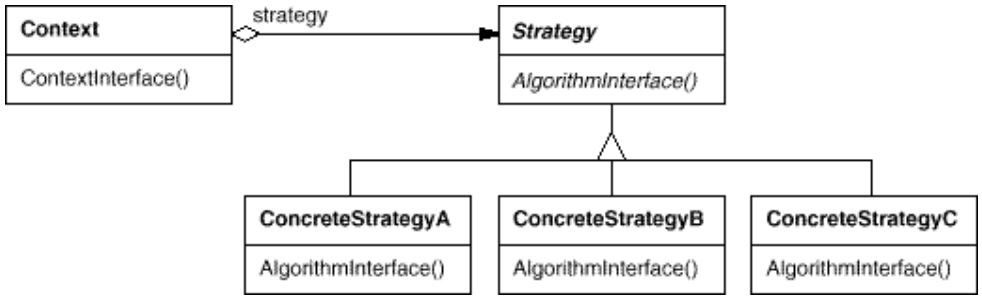
A Composition maintains a reference to a Compositor object. Whenever aCompositionreformats its text, it forwards this responsibility to itsCompositor object. The client of Composition specifies whichCompositor should be used by installing theCompositor it desires intothe Composition.

Applicability

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategiesprovide away to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might definealgorithms reflecting different space/time trade-offs.Strategies can be used when these variants are implemented as a classhierarchy ofalgorithms
- an algorithm uses data that clients shouldn't know about. Use theStrategy pattern to avoid exposing complex, algorithm-specific datastructures.
- a class defines many behaviors, and these appear as multipleconditionalstatements in its operations. Instead of manyconditionals, move related conditional branches into their ownStrategy class.

Structure



Participants

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

Collaborations

- Strategy and Context interact to implement the chosen algorithm. Acontextmay pass all data required by the algorithm to the strategywhen the algorithm is called.
- A context forwards requests from its clients to its strategy. Clientsusually create and pass a oncreteStrategy object to the context;thereafter, clients interact with the context exclusively. There isoften a family ofConcreteStrategy classes for a client to choosefrom.

Consequences

The Strategy pattern has the following benefits and drawbacks:

1. Families of related algorithms.
2. An alternative to subclassing
3. Strategies eliminate conditional statements.
4. A choice of implementations.
5. Clients must be aware of different Strategies.
6. Communication overhead between Strategy and Context..
7. Increased number of objects.

Implementation

Consider the following implementation issues:

1. *Defining the Strategy and Context interfaces.*The Strategy and Contextinterfaces must give a ConcreteStrategyefficient access to any data it needsfrom a context, and vice versa.
   One approach is to have Context pass data in parameters to Strategyoperations—in other words, take the data to the strategy. ThiskeepsStrategy and Context decoupled. On the other hand, Context mightpassdata the Strategy doesn't need.

2. *Strategies as template parameters.*In C++ templates can be used to configure a class with a strategy.This technique is only applicable if (1) the Strategycan be selectedat compile-time, and (2) it does not have to be changed atrun-time.In this case, the class to be configured (e.g., Context) isdefinedas a template class that has a Strategy class as aparameter:

```
template <class AStrategy>
class Context {
```

```
        void Operation() { theStrategy.DoAlgorithm(); }
                // ...
private:
        AStrategy theStrategy;
};
```

The class is then configured with a Strategy class when it'isnstantiated:

```
class MyStrategy {public:
        void DoAlgorithm();
};
Context<MyStrategy> aContext;
```

With templates, there's no need to define an abstract class that definesthe interface to the Strategy. Using Strategy as atemplate parameter alsolets you bind a Strategy to itsContext statically, which can increaseefficiency.

3. *Making Strategy objects optional.*The Context class may be simplified ifit's meaningful *not* tohave a Strategy object. Context checks to see if ithas a Strategyobject before accessing it. If there is one, then Context uses itnormally. If there isn't a strategy, then Context carries out defaultbehavior. The benefit of this approach is that clients don't havetodeal with Strategy objects atall *unless* they don't like thedefaultbehavior.

Sample Code

We'll give the high-level code for the Motivation example, which isbased on theimplementation of Composition and Compositor classes inInterViews.

The Composition class maintains a collection ofComponent instances, which represent text and graphicalelements in a document. A composition arranges component objects intolines using an instance of a Compositor subclass, whichencapsulates a linebreaking strategy. Each component has anassociated natural size, stretchability, and shrinkability. Thestretchability defines howmuch the component can grow beyond itsnatural size; shrinkability is how muchit can shrink. Thecomposition passes these values to a compositor, which usesthem todetermine the best location for linebreaks.

```
 class Composition {public:
 composition(compositor*);
void Repair();
 private Compositor* _compositor;
 component * _components;
 int _componentCount;
 int _lineWidth;
 int* _lineBreaks;// the position of linebreaks in components
  int _lineCount;// the number of lines
  };
```

Known Uses

Both ET++ [WGM88] and InterViews use strategies to encapsulatedifferentlinebreaking algorithms as we've described.

In the RTL System for compiler code optimization [JML92], strategies define different register allocation schemes (RegisterAllocator) and instruction set scheduling policies(RISCscheduler, CISCscheduler). This provides flexibility in targeting theoptimizer for different machine architectures.

The ET++SwapsManager calculation engine framework computes prices fordifferent financial instruments [EG92]. Its keyabstractions are Instrument and YieldCurve. Different instruments areimplemented as subclasses of Instrument. YieldCurve calculatesdiscount factors, which determine the present value of future cashflows. Both of these classes delegate some behavior to Strategyobjects. The framework provides a family of ConcreteStrategy classesfor generating cash flows, valuing swaps, and calculating discountfactors. You can create new calculation engines by configuringInstrument and YieldCurve with the different ConcreteStrategy objects.This approach supports mixing and matching existing Strategyimplementations as well as defining new ones.

The Booch components [BV90] use strategies as templatearguments. The Booch collection classes support three different kinds ofmemory allocation strategies: managed, controlled, andunmanaged.

RApp is a system for integrated circuit layout [GA89, AG90].RApp must lay out and route wires that connect subsystems on thecircuit. Routing algorithms in RApp are defined assubclasses of an abstract Router class. Router is a Strategy class.

Related Patterns

Flyweight (218): Strategy objects often make good flyweights.

Template Method

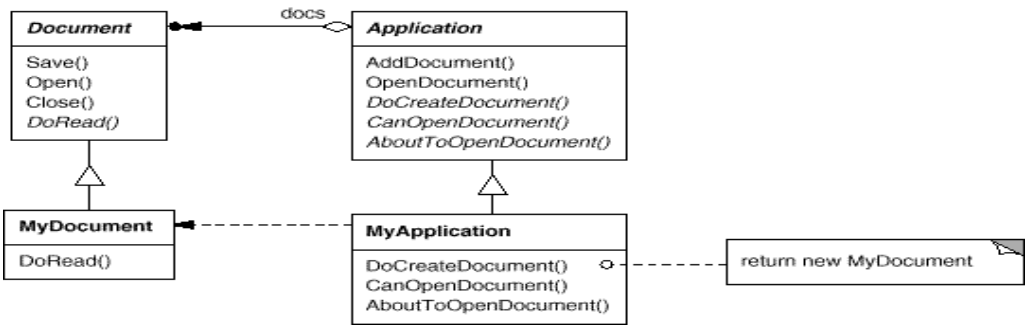
Intent

Define the skeleton of an algorithm in an operation, deferring somesteps to subclasses. Template Method lets subclasses redefinecertain steps of an algorithm without changing the algorithm'sstructure.

Motivation

Consider an application framework that provides Application andDocument classes.The Application class is responsible for openingexisting documents stored in an external format, such as a file. ADocument object represents the information ina document once it'sread from the file.

Applications built with the framework can subclass Application andDocument tosuit specific needs. For example, a drawing applicationdefines DrawApplication and DrawDocument subclasses; a spreadsheetapplication defines SpreadsheetApplication and SpreadsheetDocumentsubclasses.



The abstract Application class defines the algorithm for opening andreading adocument in its OpenDocument operation:

```
void Application::OpenDocument (const char* name)
{if (!CanOpenDocument(name)) {
// cannot handle this documentreturn;
        }

Document* doc = DoCreateDocument();
if (doc) {
    _docs->AddDocument(doc);
    AboutToOpenDocument(doc);
    doc->Open();
    doc->DoRead();
        }
    }
```

OpenDcument defines each step for opening a document. It checks ifthe document can be opened, creates the application-specific Documentobject, adds it to its set of documents, and reads the

Document from afile.

We call OpenDocument a **template method**. A template methoddefines an algorithmin terms of abstract operations that subclassesoverride to provide concrete behavior. Application subclasses definethe steps of the algorithm that check ifthe document can be opened(CanOpenDocument) and that create the Document (DoCreateDocument).Document classes define the step that reads the document (DoRead).The template method also defines an operation that lets Applicationsubclasses know when the document is about to be opened(AboutToOpenDocument), in case they care.

By defining some of the steps of an algorithm using abstractoperations, thetemplate method fixes their ordering, but it letsApplication and Document subclasses vary those steps to suit theirneeds.

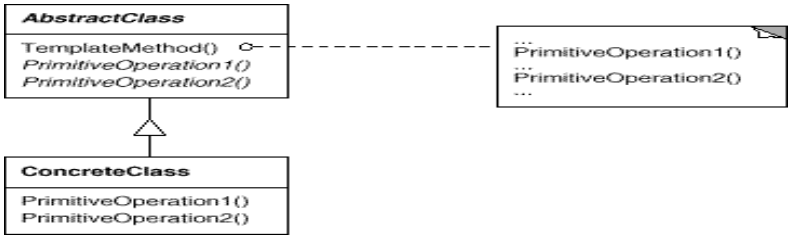
Applicability

The Template Method pattern should be used

- to implement the invariant parts of an algorithm once and leave it uptosubclasses to implement the behavior that can vary.
- when common behavior among subclasses should be factored and localizedina common class to avoid code duplication. This is a good example of"refactoring to generalize".
- to control subclasses extensions. You can define a template methodthat calls"hook" operations at specific points,thereby permitting extensions only at those points.

Structure

Participants



- **AbstractClass** (Application)
  - defines abstract **primitive operations** that concretesubclasses define to implement steps of an algorithm.
  - implements a template method defining the skeleton of an algorithm.The template method calls primitive operations as wellas operationsdefined in AbstractClass or those of other objects.
- **ConcreteClass** (MyApplication)

- implements the primitive operations to carry outsubclass-specificsteps of the algorithm.

Collaborations

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

Consequences

Template methods are a fundamental technique for code reuse. They areparticularlyimportant in class libraries, because they are the meansfor factoring out commonbehavior in library classes.

Template methods lead to an inverted control structure Template methods call the following kinds of operations:

- concrete operations
- concrete AbstractClass operations
- primitive operations
- factory methods
- **hook operations**, which provide default behavior thatsubclasses can extend if necessary. A hook operation often doesnothing by default.

It's important for template methods to specify which operations arehooks and which are abstract operations.To reusean abstract class effectively,subclass writers must understand which operations are designed foroverriding.

A subclass can *extend* a parent class operation's behavior byoverriding theoperation and calling the parent operation explicitly:

```
void DerivedClass::Operation () {
        // DerivedClass extended behavior
        ParentClass::Operation();
}
```

Unfortunately, it's easy to forget to call the inherited operation.We can transformsuch an operation into a template method to givethe parent control over how subclasses extend it. The idea is tocall a hook operation from a template methodin the parent class.Then subclasses can then override this hook operation:

```
void ParentClass::Operation () {
        //      ParentClass     behavior
        HookOperation();
}
```

HookOperation does nothing in ParentClass:

```
void ParentClass::HookOperation () { }
Subclasses override HookOperation to extend itsbehavior:
void DerivedClass::HookOperation () {
        // derived class extension
}
```

Implementation

Three implementation issues are worth noting:

1. **Using C++ access control**.In C++, the primitive operations that a templatemethod calls can bedeclared protected members. This ensures that they areonly called bythe template method. Primitive operations that *must* be overridden aredeclared pure virtual. The template method itself should notbeoverridden; therefore you can make the template method a nonvirtualmemberfunction.
2. **Minimizing primitive operations**.An important goal in designing templatemethods is to minimize thenumber of primitive operations that a subclassmust override to fleshout the algorithm. The more operations that needoverriding, the moretedious things get for clients.
3. **Naming conventions**.You can identify the operations that should be overridden by addingaprefix to their names. For example, the MacApp framework for Macintoshapplications [App89] prefixes template method nameswith "Do-":"DoCreateDocument", "DoRead", and so forth.

Sample Code

The following C++ example shows how a parent class can enforce aninvariant forits subclasses. The example comes from NeXT'sAppKit [Add94]. Consider a classView that supportsdrawing on the screen. View enforces the invariant that itssubclasses can draw into a view only after it becomes the "focus,"which requirescertain drawing state to be set up properly.

We can use a Display template method to set up this state.View defines two concrete operations,SetFocus and ResetFocus, that set up and clean upthe drawing state,respectively. View's DoDisplayhook operation performs the actual drawing. DisplaycallsSetFocus before DoDisplay to set up the drawingstate; Display calls ResetFocus afterwards torelease the drawing state.

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

To maintain the invariant, the View's clients always callDisplay, and Viewsubclasses always override

Do Display.

DoDisplay does nothing in View:

```
void View::DoDisplay () { }
Subclasses override it to add their specific drawing behavior:
void MyView::DoDisplay () {
        // render the view's contents
}
```

Known Uses

- Template methods are so fundamental that they can be found in almostevery abstract class. Wirfs-Brock et al. [WBWW90, WBJ90] provide a good overview anddiscussion of template methods.

Related Patterns

- Factory Methods (121) are often called by template methods. In the Motivationexample,the factory method DoCreateDocument is called by the template methodOpenDocument.

- Strategy (349): Template methods use inheritance to vary part of analgorithm.Strategies use delegation to vary the entire algorithm.

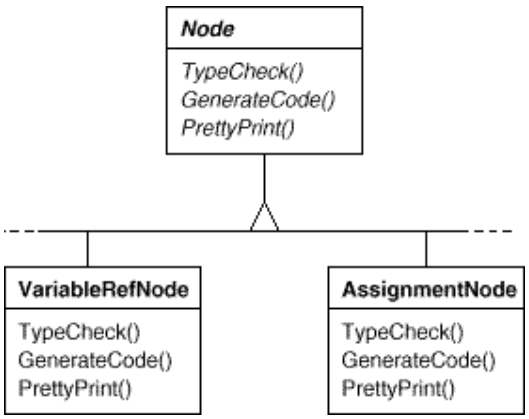
Visitor

Intent

Represent an operation to be performed on the elements of an objectstructure.

Motivation

Consider a compiler that represents programs as abstract syntax trees.It will need to perform operations on abstract syntax trees for "static semantic" analyses like checking that all variables are defined. we could use the abstract syntax trees for pretty-printing, program restructuring, code instrumentation, and computing various metrics ofaprogram.



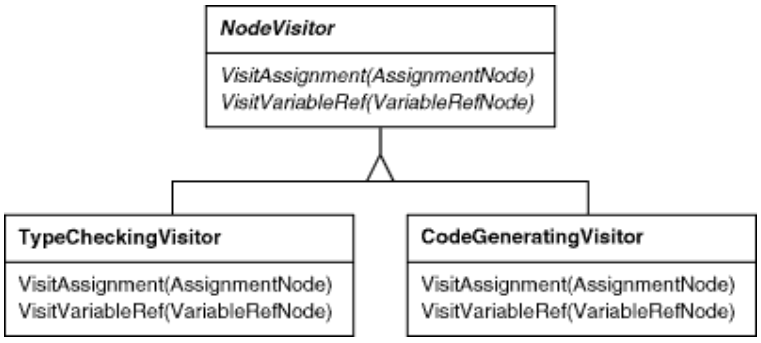
This diagram shows part of the Node class hierarchy. The problem here is thatdistributing all these operations across the various nodeclasses leads to a system that's hard to understand, maintain, andchange. It will be confusing to have pre-checking code mixed withpretty-printing code or flow analysis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

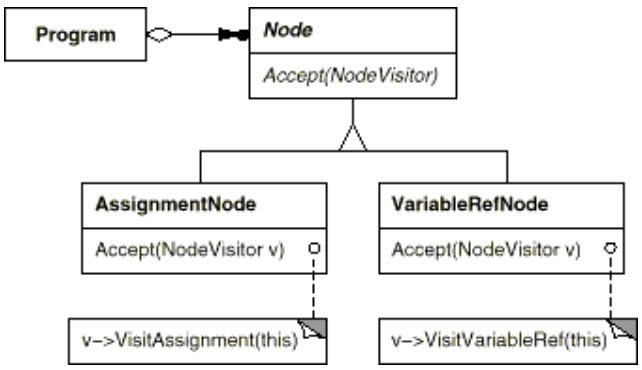
We can have both by packaging related operations from each class in aseparateobject, called a **visitor**, and passing it toelements of the abstract syntax treeas it's traversed. When an element"accepts"the visitor, it sends a request tothe visitor that encodesthe element's class. It also includes the element as anargument. Thevisitor will then execute the operation for that element—theoperationthat used to be in the class of the element.

For example, a compiler that didn't use visitors might type-check aprocedure by calling the

TypeCheck operation on its abstract syntaxtree. Each of the nodeswould implement TypeCheck by calling TypeCheckon its components . If the compilertype-checked a procedure using visitors, then itwould create aTypeCheckingVisitor object and call the Accept operation on theabstract syntax tree with that object as an argument. Each of thenodes wouldimplement Accept by calling back on the visitor: anassignment node calls VisitAssignment operation on the visitor, whilea variable reference calls VisitVariableReference. What used to be theTypeCheck operation in class AssignmentNode is now the VisitAssignmentoperation on TypeCheckingVisitor.

To make visitors work for more than just type-checking, we need anabstract parentclass NodeVisitor for all visitors of an abstract syntaxtree. NodeVisitor mustdeclare an operation for each node class. Anapplication that needs to computeprogram metrics will define newsubclasses of NodeVisitor and will no longer need to add application-specific code to the node classes. The Visitor pattern encapsulates the operations for each compilation phase in a Visitor associatedwith that phase.





With the Visitor pattern, you define two class hierarchies: one for theelementsbeing operated on

and one for the visitorsthat define operations on the elements . You create a new operationby adding a new subclass to the visitor classhierarchy. As long as the grammar that the compiler accepts doesn'tchange , we can addnew functionality simply by defining new NodeVisitor subclasses.

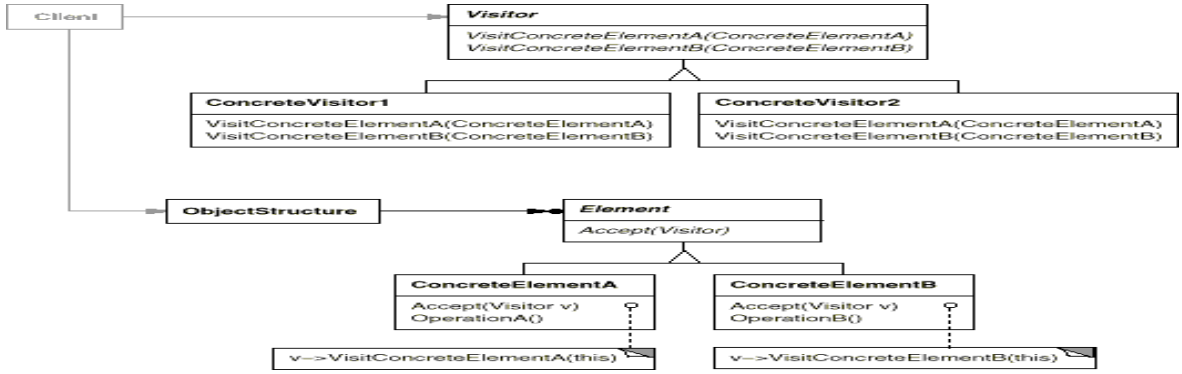
Applicability

Use the Visitor pattern when

- an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- many distinct and unrelated operations need to be performed on objectsinan object structure, and you want to avoid "polluting" theirclasses with these operations.
- Visitor lets you keep related operationstogether bydefining them in one class.
- the classes defining the object structure rarely change, but you oftenwant to define new operations over the structure.

Structure

Participants



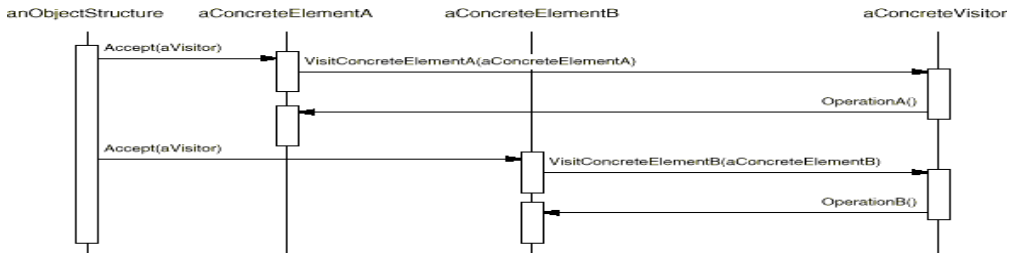
- **Visitor** (NodeVisitor)
  - Declares a Visit operation for each class of ConcreteElement in theobject structure. The operation's name and signature identifies theclass that sends the Visit request to the visitor. That lets thevisitor determine the concrete class of the element being visited.Then the visitor can access the element directly through its particular interface.
- **ConcreteVisitor** (TypeCheckingVisitor)
  - Implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the correspondingclass of object in the structure. ConcreteVisitor provides the context for the algorithm and stores its local state. This stateoften accumulates results during the traversal of the structure.
- **Element** (Node)

- defines an Accept operation that takes a visitor as an argument.
  - **ConcreteElement** (AssignmentNode,VariableRefNode)
- implements an Accept operation that takes a visitor as an argument.
  - **ObjectStructure** (Program)
- can enumerate its elements.
- may provide a high-level interface to allow the visitor to visitits elements.
- may either be a composite (see Composite (183)) or a collection suchas a list or a set.

Collaborations

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

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



Consequences

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

1. **Visitor makes adding new operations easy**.Visitors make it easy to addoperations that depend on the components ofcomplex objects. You can definea new operation over an object structuresimply by adding a new visitor.
2. **A visitor gathers related operations and separates unrelated ones**.Related behavior isn't spread over the classes defining the objectstructure; it's localized in a visitor. Unrelated sets of behavior arepartitioned in their own visitor subclasses. That simplifies both theclasses defining the elements and the algorithms defined in thevisitors.
3. **Adding new ConcreteElement classes is hard**.The Visitor pattern makes it hard to add new subclasses of Element. Eachnew ConcreteElement gives rise to a new abstract operation on Visitor anda corresponding implementation in every ConcreteVisitor class. Sometimes adefault implementation can be provided in Visitor that can be inheritedby most of the ConcreteVisitors, but

this is the exception rather thanthe rule.

4. **Visiting across class hierarchies.** An iterator canvisit the objects in astructure as it traverses them by calling theiroperations. But an iteratorcan't work across object structures with different types of elements.

5. **Accumulating state.** Visitors can accumulate state as they visit each element in the objectstructure. Without a visitor, this state would be passed as extraarguments to the operations that perform the traversal, or theymight appear as global variables.

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

Implementation

Each object structure will have an associated Visitor class. This abstract visitor class declares a VisitConcreteElement operation foreach class of ConcreteElement defining the object structure. EachVisit operation on the Visitor declares its argument to be aparticular ConcreteElement, allowing the Visitor to access theinterface of the ConcreteElement directly. ConcreteVisitor classesoverride each Visit operation to implement visitor-specific behaviorfor the corresponding ConcreteElement class.

The Visitor class would be declared like this in C++:

```
class Visitor {
public:
        virtual void VisitElementA(ElementA*);
        virtual void VisitElementB(ElementB*);
        // and so on for other concrete elements
protected:
        Visitor();
};
```

Here are two other implementation issues that arise when you apply theVisitorpattern:

1. **Double dispatch.** the Visitor pattern lets you add operations to classeswithout changing them. Visitor achieves this by using a techniquecalled double-dispatch. It's a well-known technique. Infact, some programming languages support it directly . Languages like C++ and Smalltalk supportsingle-dispatch.

2. **Who is responsible for traversing the object structure?**

A visitor must visit each element of the object structure. The questionis, how does it get there? We can put responsibility for traversal inany of three places: in the object structure, in the visitor, or in aseparate iterator object.

Sample Code

We will use Visitor to define operations for computing the inventory of materials and the total cost for a piece of equipment.The Equipment classes are so simple that using Visitor isn't really necessary, but they make it easy to see what'sinvolved in implementing the pattern.

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

};
```

The Equipment operations return the attributes of a piece ofequipment, such as its power consumption and cost. Subclasses redefine these operations appropriately for specific types of equipment

The abstract class for all visitors of equipment has a virtualfunction for eachsubclass of equipment, as shown next. All of thevirtual functions do nothing bydefault.

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

Known Uses

- The Smalltalk-80 compiler has a Visitor class called ProgramNodeEnumerator.It's used

primarily for algorithms that analyze source code.

- IRIS Inventor [Str93]is a toolkit for developing 3-D graphics applications. Inventor represents a three-dimensional scene as a hierarchy of nodes, each representing either a geometric object or an attribute of one. Inventor does this using visitors called "actions."
- To make adding new nodes easier, Inventor implements adouble-dispatch scheme for C
- Mark Linton coined the term "Visitor" in the X Consortium'sFresco Application Toolkit specification

Related Patterns

Composite (183): Visitors can be used to apply an operation over an objectstructure defined by the Composite pattern.

Interpreter (274): Visitor may be applied to do the interpretation.

Module-4
Interactive system and the MVC Architecture

4.1 Introduction

So far we have seen examples and case-studies involving relatively simple software systems. This simplicity enabled us to use a fairly general step-by-step approach, viz., specify the requirements, model the behaviour, find the classes, assign responsibilities, capture class interactions, and so on. In larger systems, such an approach may not lead to an efficient design and it would be wise to rely on the experience of software designers who have worked on the problem and devised strategies to tackle the problem. This is somewhat akin to planning our strategy for a game of chess. A chess game has three stages—an opening, a middle game and an endgame. While we are opening, the field is undisturbed and there are an immense number of possibilities; toward the end there are few pieces and fewer options. If we are in an endgame situation, we can solve the problem using a fairly direct approach using first principles; to decide how to open is a much more complicated operation and requires knowledge of 'standard openings'. These standard openings have been developed and have evolved along with the game, and provide a framework for the player. Likewise, when we have a complex problem, we need a framework or structure within which to operate. For the problem of creating software systems, such a structure is provided by choosing software **architecture**.

we start by describing a well-known software architecture (some- times referred to as an **architectural pattern**) called the **Model**–**View**–**Controller** or **MVC** pattern. Next we design a small interactive system using such an archi- tecture, look at some problems that arise in this context and explore solutions for these problems using design patterns. Finally, we discuss pattern-based solutions in software development and some other frequently employed architectural patterns.

4.2 The MVC Architectural Pattern

The pattern divides the application into three subsystems: model, view, and controller. The architecture is shown in Figure 4.1

The pattern separates the application object or the data, which is termed the Model, from the manner in which it is rendered to the end-user (View) and from the way in which the end-user manipulates it (Controller).

In contrast to a system where all of these three functionalities are lumped together (resulting in a low degree of cohesion), the MVC pattern helps produce highly cohesive modules with a low degree of coupling. This facilitates greater flexibility and reuse. MVC also provides a powerful way to organize systems that support multiple presentations of the same information.

1: Model : The model, which is a relatively passive object, stores the data. object can play the role of model.
2: View : The view renders the model into a specified format, typically something that is suitable for interaction with the end user. For instance, if the model stores information about bank accounts, a certain view may display only the number of accounts and the total of the account balances.
3: Controller : The controller captures user input and when necessary, issues method calls on the model to modify the stored data. When the model changes, the view responds by appropriately modifying the display.

In a typical application, the model changes only when user input causes the controller to inform the model of the changes. The view must be notified when the model changes. Instance variables in the controller refer to the model and the view. Moreover, the view must communicate with the model, so it has an instance variable that points to the model object. Both the controller and the view communicate with the user through the UI. This means that some components of the UI are used by the controller to receive input; others are used by the view to appropriately display the model and some can serve both purposes (e.g., a panel can display a figure and also accept points as input through mouseclicks). It is important to distinguish the UI from the rest of the system: beginners often mistake the UI for the view. This is easy error to make for two reasons. In most systems, due to the nature of the desired look and feel and the technologies available, there is a single window in which the entire

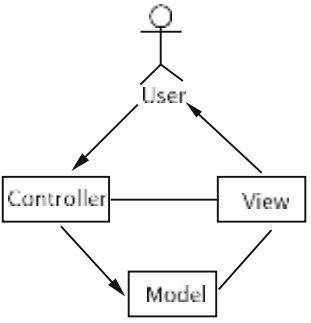


Figure 4.1 The model-view-controller architecture

application is housed. This means that there has to be a common subsystem that provides the functionality needed both for the view and the user interface. The other source of potential confusion is that the UI presents to the user an image of how the system looks, and this can be mistakenly construed as the view. This interfacemust include components that are in fact part of the controller (e.g., buttons for giving

commands). When we talk of MVC in the abstract sense, we are dealing with the architecture of the system that lies behind the UI; both the view and the controller are subsystems at the same level of abstraction that employ components of the UI to accomplish their tasks. From a practical standpoint, however, we have a situation where the view and the UI are contained in a common subsystem. *For the purpose of designing our system, we shall refer to this common subsystem as the view.* The view subsystem is therefore responsible for all the look and feel issues, whether they arise from a human–computer interaction perspective (e.g., kinds of buttons being used) or from issues relating to how we render the model. Figure 4.2 shows how we might present the MVC architecture while accounting for these practical considerations.

User-generated events may cause a controller to change the model, or view, or both. For example, suppose that the model stored the text that is being edited by the end-user. When the user deletes or adds text, the controller captures the changes and notifies the model. The view, which observes the model, then refreshes its display, with the result that the end-user sees the changes he/she made to the data. In this case, user-input caused a change to both the model and the view.

On the other hand, consider a user scrolling the data. Since no changes are made to the data itself, the model does not change and need not be notified. But the view now needs to display previously-hidden data, which makes it necessary for the view to contact the model and retrieve information.

More than one view–controller pair may be associated with a model. Whenever user input causes one of the controllers to notify changes to the model, all associated views are automatically updated.

It could also be the case that the model is changed not via one of thecontrollers, but through some other mechanism. In this case, the model must notify all associated views of the changes.

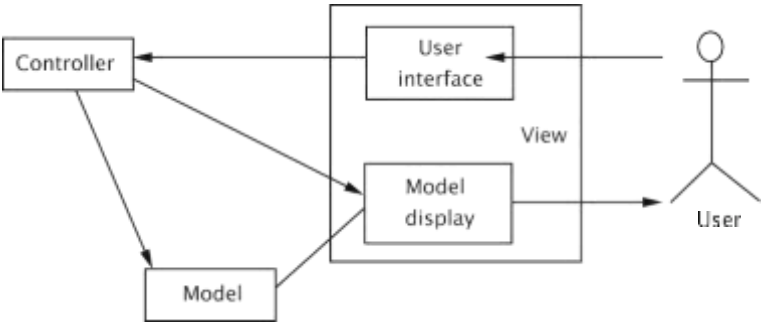


Figure. 4.2 An alternate view of the the MVC architecture

The view–model relationship is that of a subject–observer. The model, as the subject, maintains references to all of the views that are interested in observing it. Whenever an action that changes the model occurs, the model automatically notifies

all of these views. The views then refresh their displays. *The guiding principle here is that each view is a faithful rendering of the model.*

4.2.1 Examples

1. Suppose that in the library system we have a GUI screen using which users can place holds on books. Another GUI screen allows a library staff member to add copies of books. Suppose that a user views the number of copies, number of holds on a book and is about to place a hold on the book. At the same time, a library staff member views the book record and adds a copy. Information from the same model (book) is now displayed in different formats in the two screens.
2. A second example is that of a mail sever. A user logs into the server and looks at the messages in the mailbox. In a second window, the user logs in again to the same mail server and composes a message. The two screens form two separate views of the same model.
3. Suppose that we have a graph-plot of pairs of *(x, y)* values. The collection of data points constitutes the model. The graph-viewing software provides the user with several output formats—bar graphs, line graphs, pie charts, etc. When the user changes formats, the view changes without any change to the model.

4.2.2 Implementation

As with any software architecture, the designer needs to have a clear idea about how the responsibilities are to be shared between the subsystems. This task can be simplified if the role of each subsystem is clearly defined.

- The view is responsible for all the presentation issues.
- The model holds the application object.
- The controller takes care of the response strategy.

The definition for the model will be as follows:

```
public class Model extends Observable {
  // code
  public void changeData() {
    // code to update data
    setChanged();
    notifyObservers(changeInfo);
  }
}
```

Each of the views is an `Observer` and implements the `update` method.

```
public class View implements Observer {
  // code
  public void update(Observable model, Object data) {
    // refresh view using data
  }
}
```

If a view is no longer interested in the model, it can be deleted from the list of observers.

Since the controllers react to user input, they may send messages directly to the views asking them to refresh their displays.

For each feature, we start with a detailed list of specifications, stated clearly enough so that they can be classified as belonging to one of the three categories. In general, there is always an initiation step for each operation; the manner in which the user is to be shown the feature and the manner in which it is invoked are part of the presentation. What the system should do when the request is made is a part of the response strategy, and the controller manages this part of the show. This strategy may involve interacting with the user in tandem with making changes to the application object. What is needed from the user is part of the response strategy, but how the system communicates with the user is a presentation issue. Changes to the application object are made by invoking the methods of model. As the application object is modified, the display needs to be modified to reflect the changes. Modifying the display is again a matter for presentation.

Clearly, there is a lot of entanglement here between the three parts, and it is a challenge to keep everything separate. The controller invokes the methods provided by the model so that the separation is relatively easy to implement. There can be confusion around drawing a line between the responsibilities of the view and the controller for reasons explained earlier. Likewise, keeping the business logic away from the display (or model–view separation) can be tricky in situations where there is a close relationship between the stored data and the methods for rendering it. As we design and implement a case-study in the following pages, we make decisions as various situations arise. Although the philosophy behind this architecture is easily stated, the details are best explained by example.

The approach we use to resolve this is to create a UI with functionality to serve the purpose of both the view and the controller. Display components will be available to the view, which invokes the appropriate display commands. Components which capture events generated by user inputs are configured to pass on the message to the appropriate subsystem; note that events for some operations (like scrolling) are handled by the view, whereas others (like add, delete) are sent to the controller.

4.2.3 Benefits of the MVC Pattern

1. Cohesive modules: Instead of putting unrelated code (display and data) in the same module, we separate the functionality so that each module is cohesive.
2. Flexibility: The model is unaware of the exact nature of the view or controller it is working with. It is simply an observable. This adds flexibility.
3. Low coupling: Modularity of the design improves the chances that components can be swapped in and out as the user or programmer desires. This also promotes parallel development, easier debugging, and maintenance.
4. Adaptable modules: Components can be changed with less interference to the rest of the system.

5. Distributed systems: Since the modules are separated, it is possible that the three subsystems are geographically separated.

4.3 Analysing a Simple Drawing Program

We now apply the MVC architectural pattern to the process of designing a simple program that allows us to create and label figures. The purpose behind this exercise is twofold:

- *To demonstrate how to design with architecture in mind* Designing with architecture in mind requires that we start with a high-level decomposition of responsibilities across the subsystems. The subsystems are specified by the architecture. The designer gets to decide which classes to create for each subsystem, but the responsibilities associated with these classes must be consistent with the purpose of the subsystem.

- *To understand how the MVC architecture is employed* Weshall follow the architecture somewhat *strictly*, i.e., we will try to have three clearly delineated subsystems for Model, View, and Controller. Later on, we will explore and discuss variations on this theme.

As always, our design begins with the process of collecting requirements.

4.3.1 Specifying the Requirements

Our initial wish-list calls for software that can do the following.

1. Draw lines and circles.
2. Place labels at various points on the figure; the labels are strings. A separate command allows the user to select the font and font size.
3. Save the completed figure to a file. We can open a file containing a figure and edit it.
4. Backtrack our drawing process by undoing recent operations.

Compared to the kinds of drawing programs we have on the market, this looks too trivial! Nonetheless, it is sufficient to show how the **responsibilities** can be divided so that the MVC pattern can be applied. What we shall also see, later on, is how new features can be added without disrupting the existing classes.

In order to attain this functionality, the software will interact with the user. We need to specify exactly how this interaction will take place. It should, of course, be user-friendly, fast, etc., but as in earlier examples, these non-functional requirements will not be the focus of our attention. Without more ado, let us adopt the following 'look and feel:'

- The software will have a simple frame with a display panel on which the figure will be displayed, and a command panel containing the buttons. There will be buttons for each operation, which are labeled like Draw Line, Draw Circle, Add Label, etc. The system will listen to mouse-clicks which will be employed by the user to specify points on the display panel.
- The display panel will have a cross-hair cursor for specifying points and a_ (underscore) for showing the character insertion point for labels. The default cursor will be an arrow.
- The cursor changes when an operation is selected from the command menu. When an operation is completed, the cursor goes back to the default state.
- To draw a line, the user will specify the end points of the line with mouse-clicks. To draw a circle, the user will specify two diametrically opposite points on the perimeter. For convenient reference, the center of each circle will be marked with a black square. To create a label, the starting point will be specified by a mouse-click.

4.3.2 Defining the Use Cases

Wecan nowwrite the detailed use cases for each operation. The first one, for drawing a line, is shown in Table4.1.

Actions performed by the actor	Responses from the system
1. The user clicks on the Draw Line button in the command panel	
	2. The system changes the cursor to a cross-hair
3. The user clicks first on one end pointand then on the other end point of the line to be drawn	
	4. The system adds a line segment with thetwo specified end points to the figure being created. The cursor changes to the default

Table 4.1 Use-case table for drawing a line

Actions performed by the actor	Responses from the system
1. The user clicks on the Add Label button in the command panel	
	2. The system changes the cursor to a cross-hair cursor
3. The user clicks at the left end point of the intended label	
	4. The system places a_ at the clicked location
	5. The system waits for the user response

5. The user types a character or clicks the mouse at another location	
	6. If the character is not a carriage return the system displays the typed character followed by a_, and the user continues with Step 5; in case of a mouse-click, it goes to Step 4; otherwise it goes to the default state

Table 4.2 Use-case table for Adding aLabel

The use case for drawing a circle can be done analogously.

To give the system better usability, we allow for multiple labels to be added with the same command. To start the process of adding labels, the user clicks on the command button. This is followed by a mouse-click on the drawing panel, following which the user types in the desired label. After typing in a label, a user can either click on another point to create another label, or type a carriage return, which returns the system to the default state. These details are spelled out in the use case in Table4.2. The system will ignore almost all non-printable characters. The exceptions are the Enter (terminate the operation) and Backspace (delete the most-recently entered Character) keys. A label may contain zero or more characters.

We also have use cases for operations that do not change the displayed object. An example of this would be when the user changes the font, shown in Table4.3.

The requirements call for the ability to savethe drawing and open and edit the saved drawings. The use cases for saving, closing and opening files are left as exercises. In order to allow for editing we need at least the following two basic operations: *selection* and *deletion*. The use case Select an Item is detailed in Table4.4.

There are some details here that need to be fleshed out in later stages. We have not specified how the system would indicate the change to the *selection mode*. We could do this by changing the cursor or altering the display in some other way. This use case requires that the display should indicate which items have been selected. This can be done by drawing these items in a different colour.

It is possible that the user's click does not fall on any item; in that case, the system simply ignores the mouseclick and returns to the default mode.

Actions performed by the actor	Responses from the system
1. The user clicks on the Change Font button in the command panel	
	2. The system displays a list of all the fonts available
3. The user clicks on the desired font	
	4. The system changes to the specified font and displays a message to that effect

Table 4.3 Use-case table for Change Font

Actions performed by the actor	Responses from the system
1. The user clicks on the Select button in the command panel	
	2. The system changes the display to the *selection mode*
3. The user clicks the mouse on the drawing	
	4. If the click falls on an item, the system adds the item to its collection of selected items and updates the display to reflect the addition. The system returns the display to the default mode

Table 4.4 Use-case table Select an Item for

Deletion will be done by having a button in the GUI that the user can click; whenever this button is clicked, all the selected items are deleted. The use case for this is left as an exercise.

4.4 Designing the System

The process of designing this system is somewhat different from our earlier case studies owing to the fact that we have selected an architecture. Our architecture specifies three principal subsystems, viz., the Model, the View and the Controller. We have a broad idea of what roles each of these play, and our first step is to define these roles in the context of our problem. As we do this, we look at the individual use cases and decide how the responsibilities are divided across the three subsystems. Oncethis is takencare of, welook into thedetails ofdesigningeach ofthesubsystems.

4.4.1 Defining the Model

Our next step is to define what kind of an object we are creating. This is relatively simple for our problem; we keep a collection of line, circle, and label objects. Each line is represented by the end points, and each circle is represented by the *X*-coordinates of the leftmost and rightmost points and the *Y* -coordinates of the top and bottom points on the perimeter (see Figure. 4.3).

For a label, the model stores the coordinate's starting position, the text, and the style and size of the characters in the string. The collection is accessed by the view when the figure is to be rendered on the screen. The model also provides mech anisms to access and modify its collection objects. These would be methods like `addItem(Item), getItems()`, etc.

4.4.2 Defining the Controller

The controller is the subsystem that orchestrates the whole show and the definition of its role is thus critical. When the user attempts to execute an operation, the input is received by the view. The view then communicates this to the controller. This

communication can be effected by invoking the public methods of the controller. Let us examine in detail the various implementation steps for the processes described in the use cases.

Drawing a Line

1. The user starts by clicking the Draw line button, and in response, the system changes the cursor. Clearly, changing the cursor should be a responsibility of the view, since that is where we define the look and feel. This would imply that the view system (or some part thereof) listen to the button click. The click indicates that the user has initiated an operation that would change the model. Since such operations have to be orchestrated through the controller, it is appropriate that the Controller be informed. The controller creates a line object (with both ndpoints unspecified).

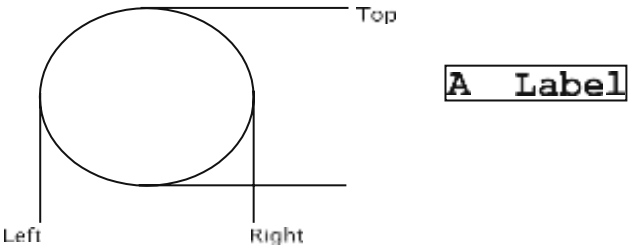


Figure. 4.3 Representing a circle and a label

2. The user clicks on the display panel to indicate the first end point of the line. We now need to designate a listener for the mouse clicks. This listener will extract the coordinates from the event and take the necessary action. Both the view and the controller are aware of the fact that a line drawing operation has been initiated. The question then is, which of these subsystems should be responding to the mouse-click? Having the controller listen directly to the mouse-clicks seems to be more efficient, since that will reduce the number of method invocations. However there are several reasons why this is not a good choice. First, the methods/interfaces (e.g., `MouseListener` in Java) to be implemented depend on the manner in which the view is being implemented. This means that the *controller is not independent of the view*, thus hurting reuse. A second reason is that we can have multiple ways to input the points. For instance, when trying to draw a precise figure, a user may prefer to specify the points as coordinates through some kind of dialog, instead of clicking the mouse. *These accommodations are part of the look and feel, and do not belong in the controller.* Finally, we have the problem of *reading and interpreting the input.* In our particular situation, this manifests itself as the process of mapping device coordinates to the image coordinates. Most of the graphical display tools available nowadays use a coordinate system where the origin corresponds to the top-left corner of the display

rectangle, with *X* coordinates increasing from left to right and *Y* coordinates increasing from top to bottom (also known as *device coordinates*). Programs that generate and use graphics often prefer the standard Cartesian coordinate system. Thus we might have a situation where the model isbeing created with Cartesian coordinates, whereas mouse clicks and graphical output must use device coordinates and points have to be mapped from one system to the other. The conversion of Cartesian coordinates to device coordinates is best done in the view since it knows and is responsible for the *nature and format of the output* (points specified as device coordinates). The reverse operation of converting device coordinates of input points to Cartesian coordinates must also, therefore, be done by the view, which means that the view must capture the input. Therefore, although a performance penalty is incurred, we favour the implementation where the mouse-click is listened to in the view. The view then communicates these coordinates to the controller, after performing any transformation or mapping that may be needed. At this point we need to decide how the system would behave during the period between the clicks. For instance, should the point for the first click be highlighted in any way? Since the use case does not specify anything, we can ignore this issue for the time being, i.e., no change happens until both end points are clicked.

3. The user clicks on the second point. Once again, the view listens to the click and communicates this to thecontroller. On receiving these coordinates, the controller recognises that the line drawing is complete and updates the line object.

4. Finally, the model notifies the view that it has changed. The view then redraws the display panel to show the modified figure.

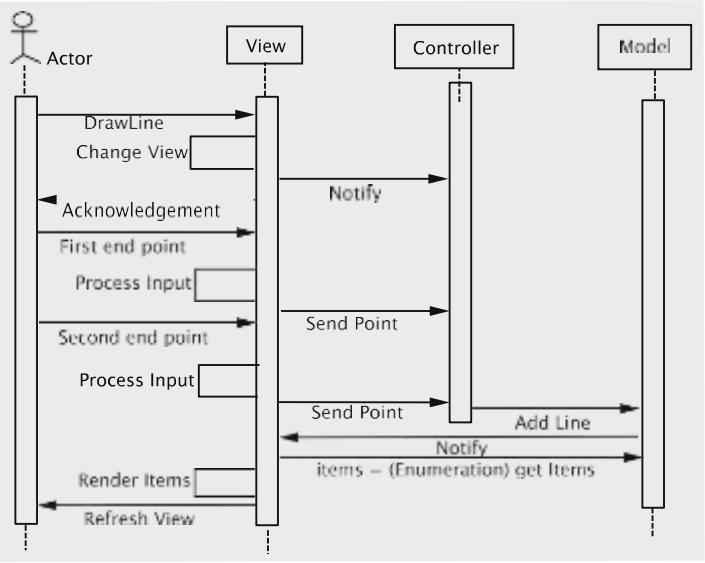


Figure. 4.4 Sequence of operations for drawing a line

This sequence of operations across the three subsystems can be captured by a high-level sequence diagram as shown in Figure. 4.4. Note that unlike the sequence diagrams in earlier chapters, this does not spell out all the classes involved or the names of the methods invoked.

Drawing a Circle

The actions for drawing a circle are similar. However, we now have some additional processing to be done, i.e., the given points on the diameter must be converted to the the four integer values, as explained in Figure.4.3. Note that this requires a mapping to convert the input to the form required by the model. This can be performed in the controller, since these representations are equivalent.

Adding a Label

This operation is somewhat different due to the fact that the amount of data is not fixed. The steps are as follows:

1. The user starts by clicking the Add Label button. In response, the system changes the mouse-cursor, which, as before is the responsibility of the view.
2. The user clicks the mouse, and the system acknowledges the receipt of the mouse click by placing a_ at the location. This would result in changing what the drawing looks like. As decided earlier, we will maintain the property that the view is a

faithful rendering of the model. The view therefore notifies the controller that the operation has been initiated, and the controller modifies the model. One issue that we have to resolve is that of assigning the appropriate size and style to the characters in the label. To implement this, we have to address the following:

- *Which subsystem 'remembers' the current style and size?* Since the user cannot be expected to specify the size and style with each character, these have to be stored somewhere. For our situation, we shall assume that these are stored in the view and passed on to the controller when the label construction operation is initiated.
- *When do the changes to size and style take effect?* To simplify our system, we assume that these will take effect for the next label that is created. What this means is that the style and size have to be uniform for any given label, and if a change is made to any of these while we are in the process of creating a label, these changes will not take immediate effect.

3. The user types in a character. Once again, the view listens to and gets the input from the keyboard, which is communicated to the controller. Once again the controller changes the model, which notifies the view.
4. The user clicks the mouse or enters a carriage-return. This is appropriately interpreted by the view. In both cases, the view informs the controller that the addition of the label is complete. In case of a mouse click, the controller is also notified that a new operation for adding a label has been initiated.

This sequence of steps is explained in Fig. 4.5. *Note that the view interprets the keystrokes: as per our specifications ordinary text is passed on directly to the controller, control characters are ignored; carriage-return is translated into a command, etc. All this is part of the way in which the system interacts with the user, and therefore belongs to the view.*

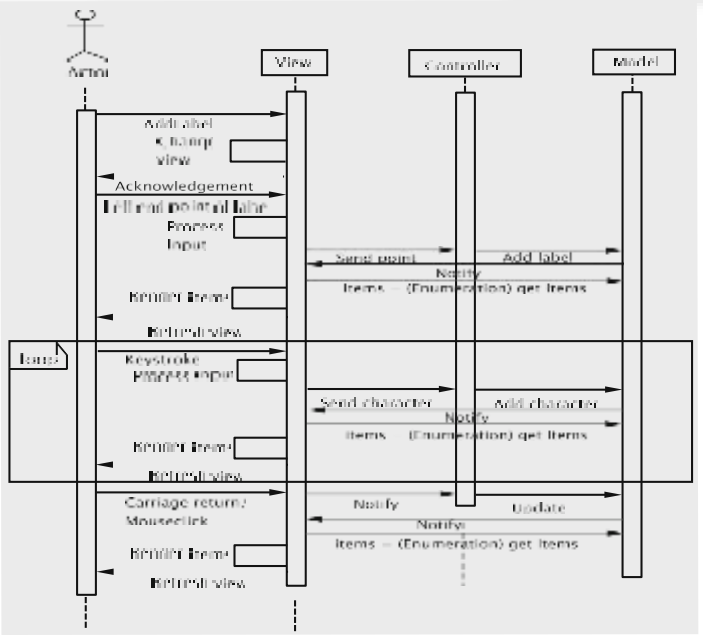


Figure. 4.5 Sequence of operations for adding a label

Sharing Responsibilities between the View and the Controller

When we employ the MVC architecture, there is often a gray area between the responsibilities of the controller and those of the view, particularly for the kind of software discussed in this case-study. Issues that fall in this area can be confusing to the beginner, particularly since widely varying opinions have been expressed. Some of these issues have come up in this section and need clarification.

Accepting user input In our approach above, all user input is received by the view. Indeed, the view is the only mechanism through which the user can interact and the view parses all the input that comes in. The idea here is that the system as a whole be 'UI agnostic', i.e., the design of the system does not depend on how the UI has been implemented.

Consider the situation where the user gives a command. This is done by a button click. It is tempting to let the controller, or one of its components, listen to the click and take action. However, this creates problems if the UI is changed so that the same commands can instead be given by keystrokes. In such a situation, a change in the UI, or even in the look and feel, can force changes in the controller. In addition, there could be situations where the same operation can be initiated in multiple ways. If the controller has to accommodate all of these, it adds to the complexity of the controller and causes tight coupling.

Once an operation has been initiated, we have the issue of accepting the

data. Once again, while some designers have argued that the data be received in the controller, this approach is fraught with problems. The data could be in one of several formats. For instance, a UI designer might want to accommodate for users to type in coordinate locations instead of clicking with the mouse. (This could be important for drawing precise geometric figures.) Having the controller deal with multiple formats is not desirable. A second, more serious issue is that when the data needs some 'correction' to adjust for the display. For instance, consider a situation where the figure is being drawn with *Cartesian coordinates* due to the nature of the application. The mouse-click specifies the value in coordinates with reference to the object that is being used for the display (in Java, this would be the `JPanel`, ora `JScrollPane`), which will have to be mapped to the Cartesian values. Doing this mapping in the controller would mean exposing the controller to all the details of the components used by the view. The important thing to keep in mind is that the view is providing the user with several input mechanisms, and therefore should be responsible for receiving and interpreting the data. *The task of accepting and standardising user input is therefore the responsibility of the view.*

4.4.3 Selection and Deletion

The software allows us to delete lines, circles, or labels by selecting the item and then invoking the `delete` operation. These shall be treated as independent operations since selection can also serve other purposes. Also, we can invoke selection repeatedly so that multiple items can be selected at any given time.

When an item is selected, it is displayed in red, as opposed to black. The selection is done by clicking with the arrow (default) cursor. Lines are selected by clicking on one end point, circles are selected by clicking on the center, and labels are selected by clicking on the label.

The steps involved in implementing this are as follows:

1. The user gives the command through a button click. This is followed by a mouse click to specify the item. Both of these are detected in the view and communicated to the controller.
2. In order to decide what action the controller must take, we need to figure out how the system will keep track of the selected items. Since the view is responsible for how these will be displayed (in red, for instance) the view must be able to recognise these as selected when updating the display. Since the view gets the items from the model, it would seem appropriate that the model have a mechanism to flag the selected items. This can be done by having a tag field for each item, or simply by moving the selected items to a separate container. We shall use the latter.
3. The next step is to iterate through the (unselected) items in the model to find the item (if any) that contains the point. Since the model is to be used strictly as a repository for the data, the task of iterating through the items is done in the controller, which then invokes the methods of the model to mark the item as

selected.

4. Model notifies view, which renders the unselected items in the default colour (black) and the selected items in red. View gets an enumeration of the two lists separately and uses the appropriate colour for each. Note that model only stores a separate list of the selected items. It is the view that decides how the two lists are to be rendered.

Deletion is a simpler operation. The button click is heard in the view and passed on to the controller, which simply requests the model to delete all selected items.

4.4.4 Saving and Retrieving the Drawing

The use cases for the processes of saving and retrieving are simply described: *the user requests a save/retrieve operation, the system asks for a file name which the user provides and the system completes the task*. This activity can be partitioned between our subsystems as follows:

1. The view receives the initial request from the user and then prompts the user to input a file name.
2. The view then invokes the appropriate method of the controller, passing the file name as a parameter.
3. The controller first takes care of any clean-up operation that may be required. For instance, if our specifications require that all items be unselected before the drawing is saved, or some default values of environment variables be restored, this must be done at the stage. The controller then invokes the appropriate method in the model, passing the file name as a parameter.
4. The model serializes the relevant objects to the specified file.

This completes the first step of distributing the responsibilities across the three sub-systems. Note that unlike the earlier case studies, we did not look for classes and methods and try to create a class interaction diagram right away. This would be fairly typical when we are designing a larger software system with some advance notice about the kind of architecture being employed. As we progress through the details, we might also realise that our partitioning of responsibilities across the subsystems may have to shift a little due to other considerations. This is not unusual, since the architecture only gives us broad guidelines, and not a detailed design.

4.5 Design of the Subsystems

In this stage, the classes and their responsibilities are identified and we get a more detailed picture of how the required functionality is to be achieved.

4.5.1 Design of the Model Subsystem

we know that the model should have methods for supporting the following operations:

1. Adding an item
2. Removing an item
3. Marking an item as selected
4. Unselecting an item
5. Getting an enumeration of selected items
6. Getting an enumeration of unselected items
7. Deleting selected items
8. Saving the drawing
9. Retrieving the drawing

The class diagram is shown in Figure 4.6.

The class `Item` represents a shape such as **line or label** and enables uniform treatment of all shapes within a drawing.

Since the methods, `getItems()` and `getSelectedItems()` return an enumeration of a set of items.The view uses these methods to get the objects from the model as an enumeration of the items.

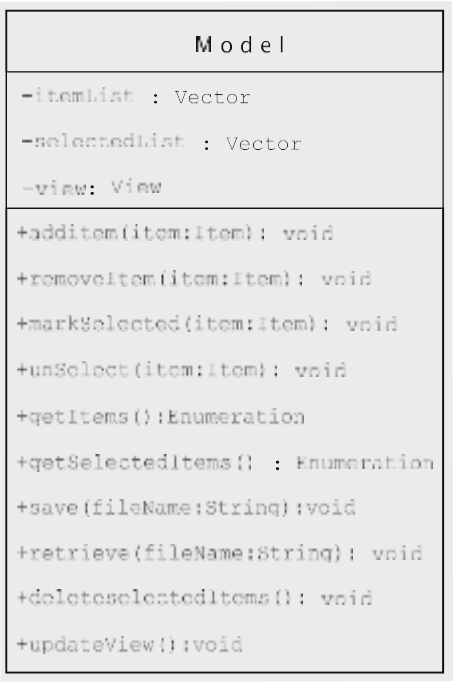


Figure. 4.6 Class diagram for model

The method `updateView` is used by the controller to alert the model that the display must be refreshed. It is also invoked by methods within the model whenever the model realizes that its data has changed. This method invokes a method in the view to refresh the display.

4.5.2 Design of Item and Its Subclasses

Clearly, `Item`will have several subclasses, one for each shape. Each subclass will store attributes that are relevant to the corresponding shape.

Rendering the items Rendering is the process by which the data stored in the model is displayed by the view. Regardless of how we implement this, the actual details of how the drawing is done are dependent on the following **two parameters**:

- *The technology and tools that are used in creating the UI* For instance, we are using the Java's Swing package, which means that our drawing panel is a `JPanel` and the drawing methods will have to be invoked on the associated `Graphics` object.
- *The item that is stored* If a line is stored by its equation, the code for drawing it would be very different from the line that is stored as two end points.

The technology and tools are known to the author of the view, whereas the structure of the item is known to the author of the items. Since the needed information is in two different classes, we need to decide which class will have the responsibility for implementing the rendering.

We have the following options:

Option 1 Let us say that the view is responsible for rendering, i.e., there is code in the view that accesses the fields of each item and then draws them. Since the model is storing these items in a polymorphic container, the view would have to query the type of each item returned by the enumeration in order to choose the appropriate method(s).

Option 2 If the item were responsible, each item would have a `render` method that accesses the fields and draws the item. The problem with this is that the way an object is to be rendered often depends on the tools that we have at our disposal. For instance, consider the problem of rendering a circle: a circle is almost always drawn as a sequence of short line segments. If the only method given in the toolkit is that for drawing lines, the circle will have to be decomposed into straight lines. In addition to the set of tools, there are other specific features that the technology has. Using the Swing package in Java, for instance, implies that all the drawing is done by invoking the methods on the **`Graphics`**object associated with the drawing panel.

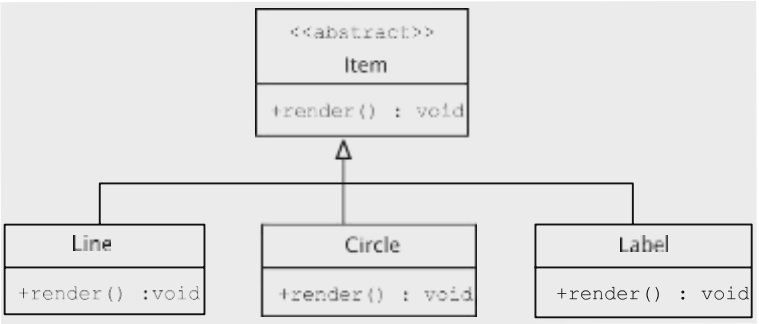


Figure. 4.7 The item class and its subclasses

At this point it appears that we are stuck between two bad choices! However, a closer look at the first option reveals a fairly serious problem: *we are querying each object in the collection to apply the right methods.* This is very much at odds with the object-oriented philosophy, i.e., *the methods should be packed with the data that is being queried.* This really means that the `render` method for each item should be stored in the item itself, which is in fact the approach of the second option.

The structure of the abstract `Item`class and its subclasses are shown in Fig. 4.7.

Catering to Multiple UI Technologies

Swing is just one package for drawing. Before it was developed, there was (and still is) the AWT (Abstract Windowing Toolkit) package available to Java programmers. Let us assume that we have available two new toolkits, which are called, for want of better names, `HardUI` and `EasyUI`. Essentially, what we want is that each item has to be customised for each kind of UI, which boils down to the task of having a different `render` method for each UI. One way to accomplish this is to use **inheritance**.

To adapt the design to take care of the new situation, we have the `Circle` class implement most of the functionality for circle, except those that depend on the UI technology. We extend `Circle` to implement the `SwingCircle` class. Similar extensions are now needed for handling the new technologies, `HardUI` and `EasyUI`. Each of the three classes has code to draw a circle using the appropriate UI technology. The idea is shown in Figure. 4.8.

In each case, the `render` method will decompose the circle into smaller components as needed, and invoke the methods available in the UI to render each component. In addition, each method would have to get any other contextual information. For instance, with the Swing package, the `render` method would get the graphics object from the view and invoke the `drawOval` method. The code for this could look something like this:

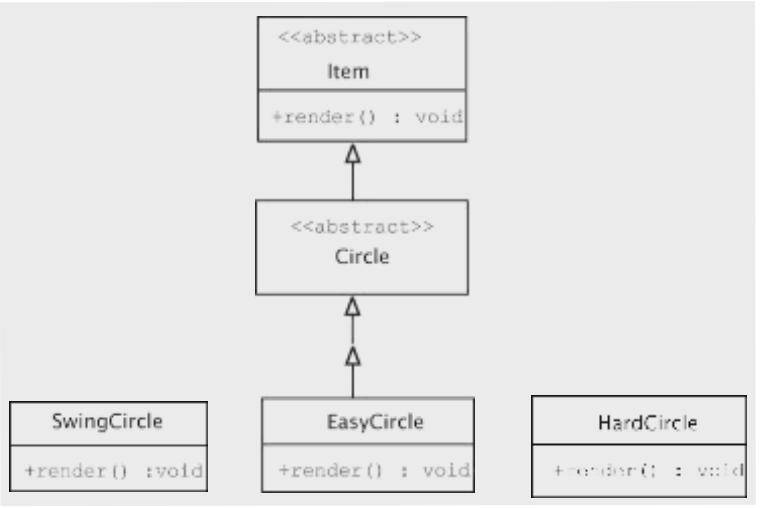


Figure. 4.8 Catering to multiple UI technologies

```
public class SwingCircle extends Circle {
  // circle class for SwingUI
  public void render() {
    Graphics g = (View.getInstance()).getGraphics();
    g.drawOval(/* parameters */);
  }
}
```

The actual parameters for `drawOval` would depend on any mapping needed, but would be computed using quantities stored in the `Circle` object. In addition to the `Graphics` object, we may need several other pieces of information from the context, such as the size of the drawing area, etc. The model could potentially employ several types of items, each of which has a corresponding abstractclass.

Clearly, we need abstract classes for implementing the technology-independent parts of lines (`Line`) and labels (`Label`). They are extended by classes such as `SwingLabel`, `SwingLine`, `EasyLabel`, etc. This extension adds another six classes. Each abstract class ends up with as many subclasses as the number of UIs that we have to accommodate.

Thenumber of classesneeded to accommodate such a solution is given by:

Number of types of items × *Number of UI packages*

As is evident from the pictorial view of the resulting hierarchy (see Figure. 4.9), this causes an unacceptable explosion in the number of classes.

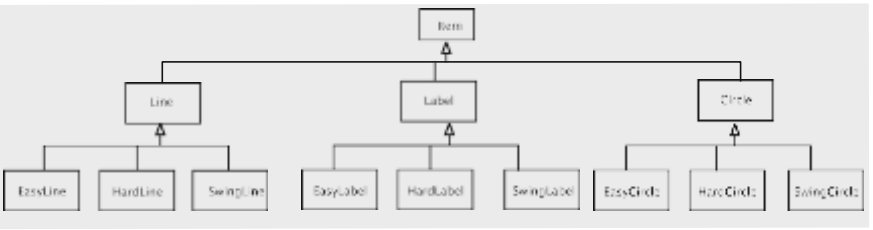


Figure. 4.9 Class explosion due to multiple UI implementations

Next, consider the situation where items are being created in the controller. Some kind of conditional will be needed to decide which concrete class should be instantiated, and this requires the code in the controller to be aware of the UI package that we are using.

A third and more subtle point is that of software upgrades. Suppose we create a version of our drawing program that supports the HardUI package and we use that to create a figure. All the items created in the model will belong to the HardUI subclasses, and can be used only with a system where the HardUI package is available. If a later version of the software does not support HardUI (or we move the files to a system that does not support it), we cannot access the old files anymore. If the objects created in the model were independent of the type of UI, this problem could be avoided.

Can all these problems be circumvented? What we have here are two subsystems viz., the model and the view, each of which has its own classification viz., the types of items and the types of UIs. Weare creating objects that account for both of these variations. Since the `Item` subclasses are being created in the model, the types of items are an ***internal variation***. On the other hand, the subclasses of `Circle`, `Line`, and `Label`(such as `HardCircle`) are an ***externalvariation***. Thestandardapproach for this is to factor out the external variations and keep them as a separate hierarchy, and then set up a *bridge* between the two hierarchies. This standard approach is therefore called the **bridge pattern.**

Figure 4.10 describes the interaction diagram between the classes and visually represents the bridge between the two hierarchies.

Since the only variation introduced in the items due to the different UIs is the manner in which the items were drawn, this behaviour is captured in the `UIContext` interface as shown in Figure. 4.11.

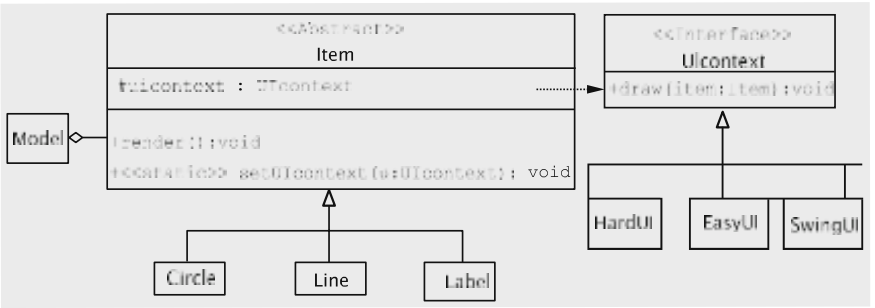


Figure. 4.10 Interaction diagram for the bridge pattern

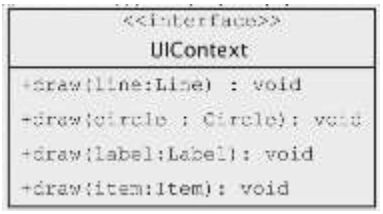


Figure. 4.11 UI Context interface

Using the Bridge Pattern

The intent of the bridge pattern is as follows: *Decouple an abstraction from its implementation so that two can vary independently.* In our example, the abstraction is the abstract class `Item`. The `render` method of this abstraction has different implementations for different UIs. Using inheritance to allow for the different implementations has the following drawbacks:

The abstractions and implementations cannot be modified and reused independently.

If the variations in the implementation are introduced from two independent sources, keeping them in the same hierarchy could have a multiplicative effect on the number of concrete classes.

- The bridge pattern takes care of these problems avoiding a permanent binding between the two. This gives our design the following desirable properties:
- Both abstraction and implementation are independently extensible (UI Context and items change indepently).
- Changes in the implementation do not affect the clients.
- Allows the implementation to be completely hidden from clients
- Reduces the number of classes.
- Multiple classes can share the same representation.

One of the guiding principles of object-oriented design states:
"Favour objects composition over class inheritance".

Note that the total number of classes is now reduced to

Number of types of items + Number of UI packages

Reflecting on the design The `UIContext` interface has a separate method for drawing each of the shapes, thereby establishing a one-to-one mapping with the shapes (circle, line, label). In general, such a one-to-one mapping is neither necessary nor realistic.

Assume that we want to start supporting a new shape, say `Triangle`, with the obvious semantics, in our drawing program. This is clearly an example of a change that one should expect in a drawing program and, within reason, it should impact as few interfaces and classes as possible. The class `Triangle` can then be written as below.

```
public class Triangle extends Item {
  private Line line1;
  private Line line2;
  private Line line3;
  // Fields, constructor, and other methods
  public void render() {
    uiContext.draw(line1);
    uiContext.draw(line2);
    uiContext.draw(line3);
  }
}
```

Similarly, we could support arbitrary polygons.

This demonstrates a couple of things. For one, it justifies the use of the bridge pattern in our design**. We are varying the `Item` hierarchy while requiring no changes at all to the `UIContext` hierarchy.**

In addition, it shows that the methods of `UIContext` can be quite 'general purpose' and not tied exclusively to one specific shape.

Suppose we restrict `UIContext` to the following:

```
public interface UIContext {
  public void draw(Point point1, Point point2); // for Line
  public void draw(String string, RenderInformation information);
                                                // for Label
}
```

As the reader might guess, `draw` with the two `Point` parameters renders a line connecting the given points. The other `draw` method draws a sequence of characters with information such as the font and font size specified in an as yet unimplemented class named `RenderInformation`. Clearly, the `Line` class's `render` method can call the first `draw` method of `UIContext` and the label can be drawn by calling the second `draw` method. We do not require any additional functionality, since any shape can be drawn by decomposing it into a large number of lines. Since there is no method to draw a circle, the `Circle` class must repeatedly invoke the first `draw` method to render the circle.

Employing option 1 Assume that rather than assigning the responsibility of drawing an `Item` object to the object itself, we have the view draw all the items. This could be accomplished by having methods such as `draw(Line line)` and `draw(Circle circle)` in the view subsystem. Every view will potentially have a different implementation of these methods. To render the items, a reference to the current view is obtained and the appropriate `draw` method is then called on that object.

While the methods that result from employing Option 1 are essentially the same as we get using the bridge pattern, there is a difference in that the bridge pattern employs a different class for each UI technology whereas Option 1 employs a set of `draw` methods for each view.

4.5.3 Design of the Controller Subsystem

We structure the controller so that it is not tied to a specific view and is unique to the drawing program.

The view receives details of a shape (type, location, content, etc.) via mouse clicks and key strokes. As it receives the input, the view communicates that to the controller through method calls. This is accomplished by having the fields for the following purposes.

1. For remembering the model;
2. Tostore the current line, label, or circle being created. Since we have three shapes, this would mean having three fields.

When the view receives a button click to create a line, it calls the controller method `makeLine`. To reduce coupling between the controller and the view, we should allow the view to invoke this method at any time: before receiving any points, after receiving the first point, or after receiving both points. For this, the controller has three versions of the `makeLine` method and keeps track of the number of points independently of the view.

The execution of `makeLine` causes the line to be part of the model. The view can set the endpoints of the line via the `setLinePoint`method.

The approach to add a label is similar to the one for adding a line. For a label, remember that by pressing the backspace the user can delete a character, so we provide a method `removeCharacter` for this purpose.

The controller also supplies a method (`selectItem`) that the view can call when it receives the command to select an item. The controller searches through the entire list of unselected items and determines if one of them is selected, and if so, it moves the item from the list of unselected items to the list of selecteditems.

The rest of the methods are for deleting selected items and for storing and retrieving the drawing and are fairly obvious. The class diagram is shown in Fig. 4.12.

To implement the saving and retrieval of files, the only objects to be serialized are the list(s) of the `Item` objects, which is a straightforward process. However, one of our stated goals is that of allowing a file to be retrievable even if the software has been modified so that we have a different version of the view, or if new featuresare

added. This means that in the new version of the software the concrete `UIContext` may be different from the one that was used to create the items in the serialized list. One solution to this could be to set `uiContext` to `null` in all the objects being stored to disk and then reset these when the objects are read from disc. This solution is inelegant and some what worrisome in that the objects are being modified when saved and retrieved.

This is a reason why we have made `Item` an abstract class (instead of an interface). This enables us to store `UIContext` as a `static` field in this class, along with the

We leave the circle implementation as an exercise, so we end up having only two fields in our design.

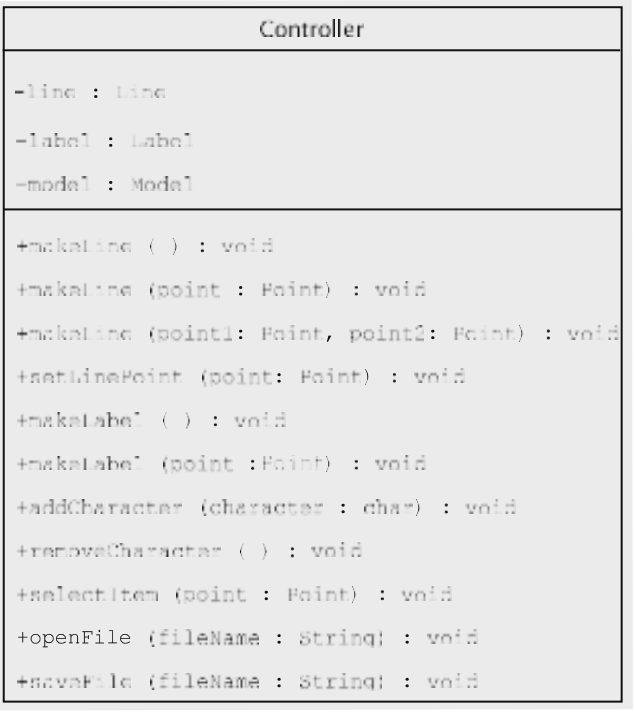


Figure. 4.12 Controller class diagram

static method `setUIContext` to modify it. The `UIContext` object is thus not a part of the object that is saved. This is consistent with the basic idea of the Bridge pattern, which calls for separation between the items and the manner in which they are rendered.

4.5.4 Design of the View Subsystem

The separation of concerns inherent in the MVC pattern makes the view largely independent of the other subsystems. Nonetheless, its design is affected by the controller and the model in two important ways:

1. Whenever the model changes, the view must refresh the display, for which the view must provide a mechanism.
2. The view employs a specific technology for constructing the UI. The corresponding implementation of `UIContext` must be made available to `Item`.

The first requirement is easily met by making the view implement the `Observer` interface; the `update` method in the `View` class, shown in the class diagram in Fig. 4.13, can be invoked for this purpose.

The issue regarding `UIContext` needs more consideration. The view consists of a drawing panel, which extends `JPanel` and needs to be updated using the appropriate instance of `UIContext`. A major question that arises is as to how and when this variable is to be set in `Item`. This can be achieved by having a public method, say `setUIContext`, in the model that in turn invokes the `setUIContext` on `Item`.

However, the time when we have to ensure that we are using the right instance of `UIContext` is just before a drawing is rendered by the view. Also, it is the view that knows which specific instance of `UIContext` is to be used in conjunction with itself. A logical way of doing this, therefore, would be to keep track of the appropriate `UIContext` in the view and invoke the `setUIContext` method in the model just before refreshing the panel that displays the drawing. In the Swing package, repainting is effected in the `paintComponent` method.

With multiple views, invoking the `setUIContext` method is problematic. Consider: more than one view might have scheduled repainting the screen, which would cause all of them to be executing `paintComponent` (or similar drawing method). If one of the views updates the `UIContext` field in the model while another is in the middle of painting the screen, chaos would result. This can be overcome by viewing the repainting code as a *criticalsection*.

Accepting input We have already decided that the user will issue commands by clicking on buttons. In the current implementation, we will assume that coordinate information (endpoints of lines, starting point of labels, etc.) will be specified by

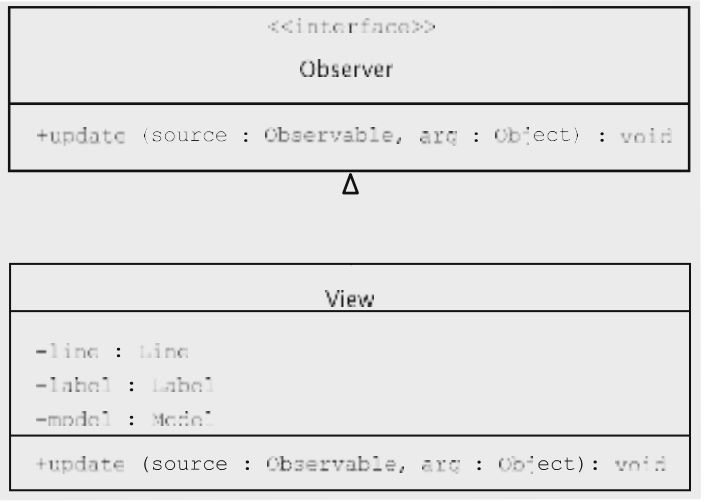


Figure. 4.13 Basic structure of the view class

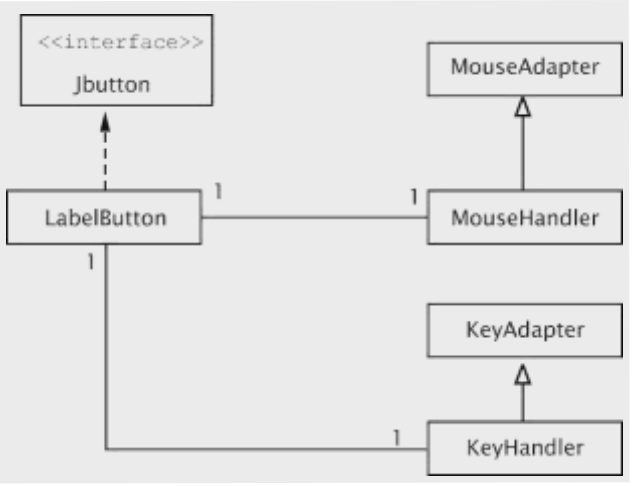


Figure. 4.14 Organization of the classes to add labels

clicking on the panel. To catch these clicks, we need a class that acts as a mouse listener, which in Java demands the implementation of the `MouseListener`[4] interface.

Commands to create labels, circles, and lines all require mouse listeners. Since the behaviour of the mouse listener is dependent on the command, we know from previous examples in the book that a truly object-oriented design warrants a separate

class for capturing the mouse clicks for each command. Since there is a one-to-one correspondence between the mouse listeners and the drawing commands, we have the following structure:

1. For each drawing command, we create a separate class that extends `JButton`. For creating labels, for instance, we have a class called `LabelButton`. Every button is its own listener.
2. For each class in (1) above, we create a mouse listener. These listeners invoke methods in the controller to initiate operations.
3. Each mouse listener (in (2) above) is declared as an inner class of the corresponding button class. This is because the different mouse listeners are independent and need not be known to each other.

The idea is captured in Fig. 4.14. The class `MouseHandler` extends the Java class `MouseAdapter` and is responsible for keeping track of mouse movements and clicks and invoking the appropriate controller methods to set up the label. In addition to capturing mouse clicks, the addition of labels requires the capturing of keystrokes. The class `KeyHandler` accomplishes this task by extending `KeyAdapter`.

In another implementation, the view may choose to have other listeners that keep track of events like resising the window, zooming-in, etc. These do not affect the model and can be handled by redrawing the figure.

If the user abandons a particular drawing operation, we could be in a tricky situation where there is more than one `MouseHandler` object receiving mouse clicks and performing conflicting operations such as one object attempting to create a line and another trying to add a label. To prevent this, we have two mechanisms in place.

1. The `KeyAdapter` class also implements `FocusListener` to know when key strokes cease to be directed to this class.
2. The drawing panel ensures that there is at most one listener listening to mouse clicks, key strokes, etc. This is accomplished by overriding methods such as `addMouseListener` and `addKeyListener`.

4.6 Getting into the Implementation

4.6.1 Item and Its Subclasses

This class `Item`is abstract and its implementation is as follows:

```
import java.io.*;
import java.awt.*;
public abstract class Item implements Serializable {
  protected static UIContext uiContext;
  public static void setUIContext(UIContext uiContext) {
    Item.uiContext = uiContext;
```

```
  }
  public abstract boolean includes(Point point);

  protected double distance(Point point1, Point point2) {
    double xDifference = point1.getX() - point2.getX();
    double yDifference = point1.getY() - point2.getY();
    return ((double) (Math.sqrt(xDifference * xDifference +
                    yDifference * yDifference)));
  }
  public void render() {
    uiContext.draw(this);
  }
}
```

The `UIContext` and its significance were discussed earlier in the context of using the bridge pattern. The `includes` method is used to check if a given point selects the item.

The `Line` class looks something like this:

```
public class Line extends Item {
  private Point point1;
  private Point point2;
  public Line(Point point1, Point point2) {


    this.point1 = point1;
    this.point2 = point2;
  }
  public Line(Point point1) {
    this.point1 = point1;
  }
  public Line() {
  }
  public boolean includes(Point point) {
    return ((distance(point, point1 ) < 10.0) || (distance(point, point2)
              < 10.0));
  }
  public void render() {
    uiContext.draw(this);
  }
  // setters and getters for the two points
}
```

The class provides three constructors. A client may thus construct a `Line` object without knowing either endpoint, or by specifying one point, or after gathering both endpoints.

Unlike `HardUI` and `EasyUI`, which are 'imaginary' UI technologies, we can readily construct an implementation of `UIContext` for the Java Swing technology.

```
public class SwingUI implements UIContext {
  private Graphics g;
  // Any other fields to hold context variables
  public void setGraphics(Graphics graphics) {
    g = graphics;
  }
  // any other methods to set context variables
```

```
  public void draw(Circle circle) {
    g.drawOval(/* parameters */);
  }
  public void draw(Line line) {
    g.drawLine(/* parameters */);
  }
  public void draw(Label label){
    g.drawString(/* parameters */);
  }
  public void draw(Item item) {
    // error message
  }
}
```

As was the case earlier, `draw` needs information from both the UI and the item. The UI information is obtained within the context object and the item is passed in as a reference. The only difference is that instead of doing all this in the `render` method of `Item`, we invoke the appropriate draw method on the UI object with which the view has been configured.

4.6.2 Implementation of the Model Class

The class maintains `itemList` and `selectedList`, which respectively store the items created but not selected, and the items selected. The constructor initialises these containers.

```
public class Model extends Observable {
  private Vector itemList;
  private Vector selectedList;
  public Model() {
    itemList = new Vector();
    selectedList = new Vector();
  }
  // other methods
}
```

The `setUIContext` method in the model in turn invokes the `setUIContext` on `Item`.

```
public static void setUIContext(UIContext uiContext) {
  Model.uiContext = uiContext;
  Item.setUIContext(uiContext);
}
```

As an `Observable`, the model notifies all of the views when it needs to inform them of changes. We have seen that this approach allows us to change `UIContext` dynamically, and also supports the displaying of multiple views simultaneously, where each view is using a different `UIContext`.

At the moment, we handle the drawing of items (including a possibly 'incomplete' one), especially labels, by having a method `updateView` in the model, which is called by the controller at appropriate moments, for example after each character is read in from the keyboard. The method simply asks that the view be refreshed.

```
public void updateView() {
  setChanged();
  notifyObservers(null);
}
```

The `addItem` method is simple: it just stores the item in `itemList` and redraws the screen.

```
public void addItem(Item item) {
  itemList.add(item);
  updateView();
}
```

The class also provides a method to delete an item.

```
public void removeItem(Item item) {
  itemList.remove(item);
  updateView();
}
```

When an item is selected by the user, the model marks it as selected by transferring the item from `itemList` to `selectedList` as below.

```
public void markSelected(Item item) {
  if (itemList.contains(item)) {
    itemList.remove(item);
    selectedList.add(item);
    updateView();
  }
}
```

Selected items are deleted using the `deleteSelectedItems`.

```
public void deleteSelectedItems() {
  selectedList.removeAllElements();
  updateView();
}
```

The `getItems`method is used bythe controller to determine which item is selected. The view uses the same method to render the items.

```
public Enumeration getItems() {
  return itemList.elements();
}
```

Implementation of the Controller Class

The class must keep track of the current shape being created, and this is accomplished by having the following fields within the class.

```
private Line line;
private Label label;
```

When the view receives a button click to create a line, it calls one of the following

controller methods. The controller supplies three versions of the `makeLine` method and keeps track of the number of points independently of the view.

```
public void makeLine() {
  makeLine(null, null);
  pointCount = 0;
}
public void makeLine(Point point) {
  makeLine(point, null);
  pointCount = 1;
}
public void makeLine(Point point1, Point point2) {
  line = new Line(point1, point2);
  pointCount = 2;
  model.addItem(line);
}
```

The variables `pointCount` and `model` are both fields within the `Controller` class that respectively keep track of the number of points received and the instance of the `Model` class.

The execution of `makeLine` causes the line to be part of the model. The view can set the endpoints of the line via the following method.

```
public void setLinePoint(Point point) {
  if (++pointCount == 1) {
    line.setPoint1(point);
  } else if (pointCount == 2) {
    pointCount = 0;
    line.setPoint2(point);
  }
  model.updateView();
}
```

After it receives each end-point, the controller calls the model's `updateView` method to inform it that the view should be updated.

The approaches to draw a circle and add a label are similar. For a label, remember that by pressing the backspace the user can delete a character. So we provide a method `removeCharacter` for this purpose.

The following method is called by the view when it receives the command to select an item. The controller searches through the entire list of unselected items and determines if one of them is selected, and if so, it moves the item from the list of unselected items to the list of selected items.

```
public void selectItem(Point point) {
  Enumeration enumeration = model.getItems();
  while (enumeration.hasMoreElements()) {
    Item item = (Item)(enumeration.nextElement());
    if (item.includes(point)) {
      model.markSelected(item);
      break;
    }
  }
}
```

Implementation of the View Class

The view maintains two panels: one for the buttons and the other for drawing the items.

```
public class View extends JFrame implements Observer {
  private JPanel drawingPanel;
  private JPanel buttonPanel;
  // JButton references for buttons such as draw line, delete, etc.
  private class DrawingPanel extends JPanel {
    // code to redraw the drawing and manage the listeners
  }
  public View() {
    // code to create the buttons and panels and put them in the JFrame
  }
  public void update(Observable model, Object dummy) {
    drawingPanel.repaint();
  }
}
```

The code to set up the panels and buttons is quite straightforward, so we do not dwell upon that.

The `DrawingPanel` class overrides the `paintComponent` method, which is called by the system whenever the screen is to be updated. The method displays all unselected items by first obtaining an enumeration of unselected items from the model and calling the `render` method on each. Then it changes the colour to red and draws the selected items.

```
public void paintComponent(Graphics g) {
  model.setUI(NewSwingUI.getInstance());
  super.paintComponent(g);
  (NewSwingUI.getInstance()).setGraphics(g);
  g.setColor(Color.BLUE);
  Enumeration enumeration = model.getItems();
  while (enumeration.hasMoreElements()) {
    ((Item) enumeration.nextElement()).render();
  }
  g.setColor(Color.RED);
  enumeration = model.getSelectedItems();
  while (enumeration.hasMoreElements()) {
    ((Item) enumeration.nextElement()).render();
  }
}
```

The `DrawingPanel` class also overrides the `addMouseListener`, `addKeyListener`, and `addFocusListener` methods. This is to ensure that there is at most one listener for each type of event on the drawing panel.

```
private MouseListener currentMouseListener;
public void addMouseListener(MouseListener newListener) {
  removeMouseListener(currentMouseListener);
  currentMouseListener = newListener;
  super.addMouseListener(newListener);
}
```

When this button is clicked, an instance of `MouseHandler` is created, and it becomes the sole listener of mouse clicks. `MouseHandler` overrides the `mouseClicked` method to determine the starting point of the label. Besides asking the controller to set up a `Label` object with the given starting point, the code makes the drawing panel receive further button clicks and keyboard events. Also note that the `KeyHandler` is a `FocusListener` as well, which lets it know when it longer receives keyboard input.

```
public void mouseClicked(MouseEvent event) {
  view.setCursor(new Cursor(Cursor.TEXT_CURSOR));
  Controller.instance().makeLabel(event.getPoint());
  drawingPanel.requestFocusInWindow();
  drawingPanel.addKeyListener(keyHandler);
  drawingPanel.addFocusListener(keyHandler);
}
```

In its `keyTyped` method, `KeyHandler` transmits all printable characters to the `Label` object via the controller. The `keyPressed` method distinguishes between the enter and backspace keys. For the former, it stops listening to mouse clicks and keyboard events. If the backspace is pressed, the label is made to delete the last typed character.

```
public void keyTyped(KeyEvent event) {
  char character = event.getKeyChar();
  if (character >= 32 && character <= 126) {
    Controller.instance().addCharacter(event.getKeyChar());
  }
}
public void keyPressed(KeyEvent event) {
  if (event.getKeyCode() == KeyEvent.VK_ENTER) {
    view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
    drawingPanel.removeMouseListener(mouseHandler);
    drawingPanel.removeKeyListener(keyHandler);
    drawingPanel.repaint();
  } else if (event.getKeyCode() == KeyEvent.VK_BACK_SPACE) {
    Controller.instance().removeCharacter();
  }
}
```

If the user terminates label creation by clicking on a button, as opposed to hitting the Enter key, the system executes the `focusLost` method of `KeyHandler`, which properly ends the command.

```
public void focusLost(FocusEvent event) {
  view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
  drawingPanel.removeMouseListener(mouseHandler);

drawingPanel.removeKeyListener(keyHandler);
  drawingPanel.repaint();
}
```

Finally, just before it refreshes the screen, the view sets up `UIContext` within the model appropriately:

```
public void paintComponent(Graphics g) {
  model.setUI(NewSwingUI.getInstance());
  // rest of the code not shown
}
```

The Driver Program

The driver program sets up the model. In our implementation the controller is independent of the UI technology, so it can work with any view. The view itself uses the Swing package and is an observer of the model.

```
public class DrawingProgram {
  public static void main(String[] args){
    Model model = new Model();
    Controller.setModel(model);
    Controller controller = new Controller();
    View.setController(controller);
    View.setModel(model);
    View view = new View();
    model.addObserver(view);
    view.show();
  }
}
```

A Critique of Our Design

The partial design of the view and the model are quite robust. We have examined some of the issues to be taken care of earlier on, and the implementation takes them into consideration. The controller appears to be quite straightforward, and we simply need to add methods to handle all the operations.

Let us see how the design stands up to the task of adding a new operation, say,to draw a polygon.

1. We need to provide a new button which informs the user that the new operation is available. We also should create a mouse handler to handle mouse clicks, etc. These changes are relatively obvious and clearly unavoidable. Even then, note that most of the classes in the view are left unchanged.
2. The model is not affected by adding new types of items, operations or new UIs.
3. The `UIContext` interface does not have to be necessarily extended when new kinds of items are added.
4. Thecontrollershould have new methodssuchas `makePolygon`and `addPoint ToPolygon`. It is not clear that this change is not a consequence of some basic flaw in our design. For instance, it might be possible to replace the methods `makeLine`, `makeCircle`, etc. by a single method, say`makeShape`.

Thus one drawback to our approach is that we need to change the controller

class every time new operations are added or even if we change the way things are implemented. In addition, the controller has all the implementation in one class, which makes things complicated.

A more tricky problem is that of implementing *undo.* Clearly some kind of a stack would be needed to remember the operations that have been completed. When an undo is requested, an element from the top of the stack is popped, and this element has to be 'decoded' to find out what the last operation was. This would require some kind of conditional, and the complexity of this method would increase with the number of different kinds of operations that we implement. In earlier chapters we have seen how such complexity can be reduced by *replacing conditional logic with polymorphism.* In the next section we examine a pattern that can help us improve the design of the controller.

Implementing the Undo Operation

In the context of implementing the undo operation, a few issues need to be high-lighted.

- *Single-level undo versus multiple-level undo* A simple formof undo is when only one operation (i.e., the most recent one) can be undone. This is relatively easy, since we can afford to simply clone the model before each operation and restore the clone to undo.
- *Undo and redo are unlike the other operations* If an undo operation is treated the same as any other operation, then two successive undo operations cancel each other out, since the second undo reverses the effect of the first undo and is thus a redo. The undo (and redo) operations must therefore have a special status as meta-operations if several operations must be undone.
- *Not all things are undoable* This can happen for two reasons. Some operations like 'print file' are irreversible, and hence undoable. Other operations like 'save to disk' may not be worth the trouble to undo, due to the overheads involved.
- *Blocking further undo/redo operations* It is easy to see that uncontrolled undo and redo can result in meaningless requests. In general, it is safer to block redo whenever a new command is executed. Consider a situation where we have the sequence: *Select(a), undo, Select(a), redo*. The redo tries to mark *a* as selected, and this could result in an exception depending on how things are implemented. A more severe problem arises with *Create Rectangle(r), Colour Rectangle(r, blue), undo, Delete(r), redo.* Here, the redo will attempt to colour a rectangle that does not exist any more.

- *Solution should be efficient* This constraint rules out naive solutions like saving the model to disk after each operation.

Keeping these issues in mind, a simple scheme for implementing undo could be something like this:

1. Create a stack for storing the history of the operations.
2. For each operation, define a data class that will store the information necessary to undo the operation.
3. Implement code so that whenever any operation is carried out, the relevant information is packed into the associated data object and pushed onto the stack.
4. Implement an `undo` method in the controller that simply pops the stack, decodes the popped data object and invokes the appropriate method to extract the information and perform the task of undoing the operation.

One obvious approach for implementing this is to define a class `StackObject` that stores each object with an identifying `String`.

```
public class StackObject {
   private String name;
   private Object object;
   public StackObject(String string, Object object) {
     name = string;
     this.object = object;
   }
   public String getName() {
     return name;
   }
   public Object getObject() {
     return object;
   }
}
```

Each command has an associated object that stores the data needed to undo it. The class corresponding to the operation of adding a line is shown below.

```
public class LineObject {
  private Line line;
  public Line getLine() {
    return line;
  }
  public LineObject(Line line) {
    this.line = line;
  }
}
```

When the operation for adding a line is completed, the appropriate `StackObject` instance is created and pushed onto the stack.

```
public class Controller {
  private Stack history;
  public void makeLine(Point point1, Point point2) {
    Line line = new Line(point1, point2);
```

```
      model.addItem(line);
      history.push(new StackObject("line", new LineObject(line)));
    }
    // other fields and methods
}
```

Decoding is simply a matter of popping the stack reading the `String`.

```
public void undo() {
  StackObject undoObject = history.pop();
  String name = undoObject.getName();
  Object obj = undoObject.getObject();
  if (name.equals("line")) {
    undoLine((LineObject)obj);
  } else if (name.equals("delete")) {
    undoDelete((DeleteObject)obj);
  } else if (name.equals("select")) {
    undoSelect((SelectObject)obj);
  }
  // one else if for each command
}
```

Finally, undoing is simply a matter of retrieving the reference to and removing the line form the model.

```
public class Controller {
  public void undoLine(LineObject object){
    Line line = object.getLine();
    model.removeItem(line);
  }
}
```

There are two obvious drawbacks with this approach:

1. *The long conditional statement in the undo method of thecontroller.*
2. *The need to rewrite the controller whenever we make changes such as adding or modifying the implementation of an operation.*

The object-oriented approach for dealing with the first drawback is to subclass the behaviour by creating an inheritance hierarchy and *replace conditional logic with polymorphism.*

Let us refactor the code to accomplish this. Before replacing the conditional, however, we see that `undo` in the controller is mostly working off the data stored in `StackObject` and our first order of business is to extract and move this method.

```
public class Controller {
  private Stack history;
  public void undo() {
    StackObject undoObject = history.pop();
    undoObject.undo(this);
  }
  // other fields and methods
}

public class StackObject {
```

```
  public void undo(Controller controller) {
    String name = getName();
    Object object = getObject();
    if (name.equals("line")) {
      controller.undoLine((LineObject)object);
    } else if (name.equals("delete")) {
      controller.undoDelete((DeleteObject)object);
    } else if (name.equals("select")) {
      controller.undoSelect((SelectObject)object);
    }
  }
  // other fields and methods
}
```

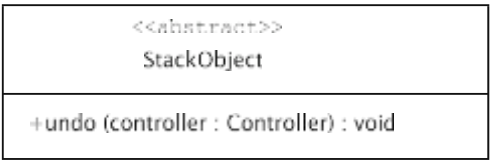


Figure. 4.15 Representing the drawing of a line

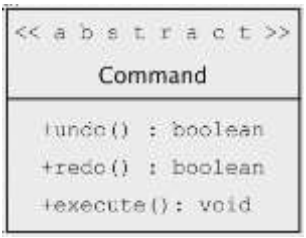
Now our conditional is in `StackObject` and we are ready to subclass this behav- iour. Since each kind of data object is associated with an operation, our hierarchy will have a subclass corresponding to each operation. For example, to represent the drawing of a line, we have the class `LineObject` as a subclass of `StackObject` (Figure 4.15).

This is a lot simpler and cleaner, although we have paid a price by increasing the number of method calls. Note that we no longer 'decode' the stored objectsand therefore the name field is not required. The `makeLine` method is simplified, so it just creates a `LineObject` and pushes it onto the stack.

```
public void makeLine(Point point1, Point point2) {
  Line line = new Line(point1, point2);
  model.addItem(line);
  history.push(new LineObject(line));
}
```

In the next subsection, we look into creating a fully reusable controller.

Figure. 4.16 The command class



4.7.1 Employing the Command Pattern

The reader may have noticed a familiar pattern in the above code. In its `undo` method, the controller passes itself as a reference to the `undo` method of the `StackObject`. In turn, each subclass of the `StackObject` (e.g., `LineObject`) passes itself as reference when invoking the appropriate undo method of the controller. This is an implementation of *double dispatch* that we used when employing the *visitor* pattern and was wholly appropriate when introducing new functionality into an existing hierarchy. In this context, however, we find that this results in unnecessarily moving a lot of data around. One of the lasting lessons of the object-oriented experience is the supremacy of data over process (The Law of Inversion), which we can utilise in this problem by using the **commandpattern.**

The intent of the command pattern is as follows

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

We have partially satisfied this intent in our scenario by associating an object with each operation. For instance, whenever we execute an operation to create a line, a `LineObject` is created and pushed onto the stack. What we have failed to recognise so far is that this object need not merely be a repository of associated data but can also encapsulate the routines that need access to this data.

The command pattern provides us with a template to address this. The abstract `Command` class has abstract methods to `execute, undo` and redo. Shown in figure 4.16

The default `undo` and `redo` methods in `Command` return `false`, and these need to be overridden as needed by the concrete command classes.

Adding a line Since every command is represented by a `Command` object, the first order of task when the Draw Line command is issued is to instantiate a `LineCommand` object. We assume that we do this after the user clicks the first endpoint although there is no reason why it could not have been created immediately after receiving the

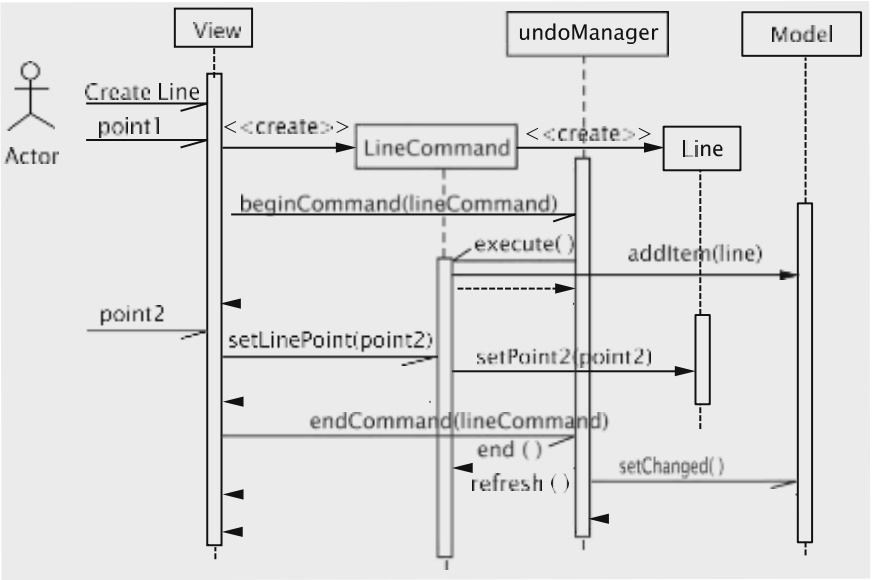


Figure. 4.17 Sequence diagram for adding a line

command. In its constructor, `LineCommand` creates a `Line` object with one of its endpoints specified.

The central idea behind the command pattern is to employ two stacks: one for storing the commands that can be undone (history stack) and the other for maintaining the commands that may be redone (redo stack). The class `UndoManager` maintains these stacks. (We refer to the corresponding object by the term **undo manager**.) The undo manager plays the role of the controller, but we have given it a new name to highlight its main function. We take the approach that as soon after the command object is created, the view informs the undo manager, which is then expected to initiate its bookkeeping operations. Similarly, when the view has received all of the data needed to complete the command, it notifies the `UndoManager` once more. The two methods `beginCommand` and `endCommand` are for these two purposes.

In the course of execution of the `beginCommand` method, the undo manager ensures the the `Line` object gets added to the model. This way, should the view be refreshed, the partial line will be shown on the screen.

When the command is completed and the `endCommand` method is executed, the undo manager pushes the command onto the history stack. This way the latest command is always at the top of this stack. we clear the redo stack whenever a new command is issued.

Assume that the user issues the sequence of commands:

Add Label (Label 1)

Draw Circle (Circle

Add Label (Label 2)

Draw Line (Line 1)

At this time, there are four `Command` objects, one for each of the above commands, and they are on the history stack as in Figure. 4.18. The redo stack is empty: since no commands have been undone, there is nothing to redo. The picture also shows the collection object in the model storing the two `Label` objects, the `Circle` object, and the `Line` object.

Undoing an operation Continuing with the above example, we now look at the sequence of actions when the undo request is issued immediately after the line (Line 1) has been completely drawn in the above sequence of commands. Obviously, the user views the command as undone if the line disappears from the screen: for this, the `Line` object must be removed from the collection. To be consistent with this action and to allow redoing the operation, the `LineCommand` object must be popped from the history stack and pushed onto the redo stack. The resulting configuration is shown in Figure. 4.19.

Not every command is undoable. So the general rule is that when the undo operation is requested, if the top of the undo stack is a command that can be undone, the command is undone and transferred to the redo stack.

The redo operation is simple enough: if the redo stack is not empty, the command must be reexecuted, and the top object in the redo stack must be transferred to

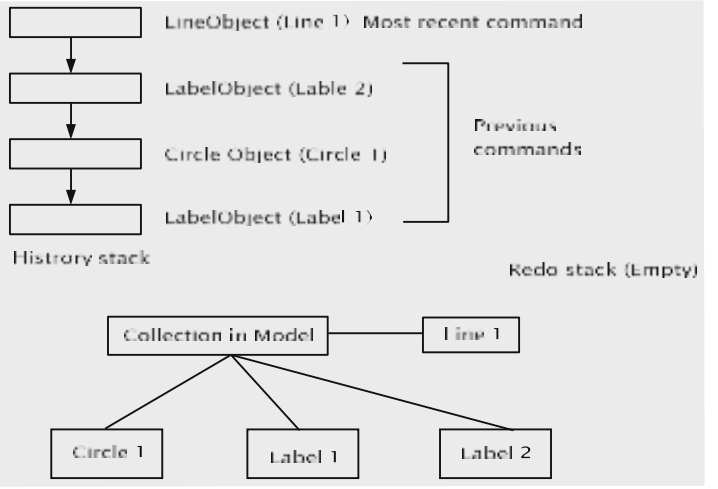


Figure. 4.18 Status of the stacks and the collection in the model

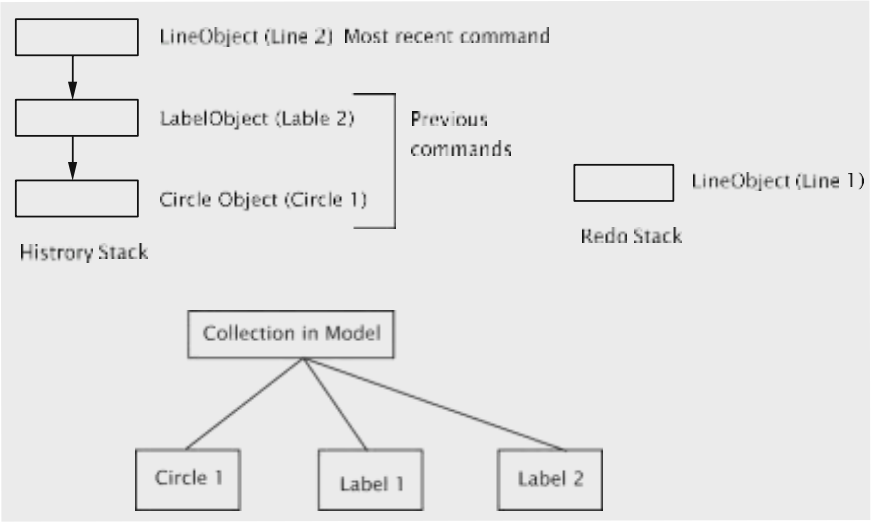


Figure. 4.19 Status of the stacks and the collection in the model after undo

As we noted earlier, not every command may be undoable. If an undoable operation is on the undo stack, the undo cannot proceed beyond that operation although there might be undoable operations underneath it in the stack. To get around this problem, we might choose to not push undoable commands onto the stack. This can be accomplished by making the command itself assume the responsibility for pushing onto the history stack. This can conveniently be done in the class's constructor.

A related issue concerns unfinished commands. We use the term *incomplete command* to refer to a command that has not yet been properly terminated. An *incomplete item* is an item, such as a line or a label, that might not have proper values for every field. We use the term *complete item* to refer to an item for which the user has supplied. all the input necessary for completely specifying the item. For example, suppose a user clicks the 'Create Line' button and clicks one point. Before clicking a second time to specify the second point, suppose the user clicks the 'Add Label' button. The Create Line command is incomplete. Moreover, the line is also incomplete at this stage, and it is already stored in the model, which now ends up containing incomplete data. One could argue that it was the user's fault, but the program must tolerate such errors and it would be nice if there was a way to fix this problem.

How should this be handled? We can suggest at least two ways:

1. We could prevent the possibility of users aborting commands in the middle. A popular approach is to **disable all command buttons** when a new command is finished and leave them disabled until the command is completed. When the

command is completed, all of the buttons are enabled.

2. A second possibility is to handle this with an additional method in both the undo manager and the command class.

The difficulty with the first approach is that the UI is responsible for ensuring data consistency. The responsibility for ensuring that items are complete must rest with the command classes and not with the user interface.

Weproceed with the second choice, for which we will have the undo manager keep the current command away from the history stack until the command itself 'certifies' that it is complete. For this purpose, every command class has an additional method, `end`, which checks whether the item is complete and attempts to fill the missing values if necessary. If there is not enough data to make the item complete, the method returns a `false` value and the undo manager does not put the command on the stack.

The pseudo-code for the `end` method is as follows:

```
public boolean end() {
  if item is incomplete
     attempt to complete using data already received;
     if cannot be completed
        return false;
     end if
  end if
  return true
}
```

The undo manager does not push a new command onto the stack until it is clear that the item is complete.

We now explain the implementation of the above concepts.

4.7.2 Implementation

Subclasses of `Command` The concrete command classes (such as `LineCommand`) store the associated data needed to undo and redo these operations. Just as the `makeLine` method in the previous implementation had three versions, the `LineCommand` class has three constructors, allowing some flexibility in the design of the view.

The implementation of methods specific to the `Command` class are shown below. The `execute` method simply adds the command to the model so the line will be drawn. To undo the command, the `Line` object is removed from the model's collection. Finally, `redo` calls `execute`.

```
public void execute() {
  model.addItem(line);
}
public boolean undo() {


  model.removeItem(line);
  return true;
}
```

```
public boolean redo() {
  execute();
  return true;
}
```

As explained earlier, the class has a method called `end`, which attempts to complete an unfinished command. The situation is considered hopeless if both endpoints are missing, so the object removes the line from the model (undoes the command) and returns a false value. Otherwise, if the line is incomplete (has at least one endpoint unspecified), the start and end points are considered the same. The implementation is:

```
public boolean end() {
  if (line.getPoint1() == null) {
    undo();
    return false;
  }
  if (line.getPoint2() == null) {
     line.setPoint2(line.getPoint1());
  }
  return true;
}
```

UndoManager It declares two stacks for keeping track of the undo and redo operations: (`history`) and (`redoStack`). The current command is stored in a field aptly named `currentCommand`.

```
public class UndoManager {
  private Stack history;
  private Stack redoStack;
  private Command currentCommand;
}
```

If the command was not properly terminated, we arrange matters such that `currentCommand` will not be `null` when a new command is issued. Recall that when a new command is issued, the `beginCommand` method of the undo manager is called. If `currentCommand` is not `null` at this time, the undo manager attempts to complete it by calling the command's `end` method. The `beginCommand` method is implemented as below.

```
public void beginCommand(Command command) {
  if (currentCommand != null) {
    if (currentCommand.end()) {
      history.push(currentCommand);
    }
  }
  currentCommand = command;
  redoStack.clear();
  command.execute();
}
```

The `undo` and `redo` are straightforward operations.

```
public void undo() {
```

```
    if (!(history.empty())) {
      Command command = (Command) (history.peek());
      if (command.undo()) {
        history.pop();
        redoStack.push(command);
      }
    }
  }

  public void redo() {
    if (!(redoStack.empty())) {
      Command command = (Command)(redoStack.peek());
      if (command.redo()) {
        redoStack.pop();
        history.push(command);
      }
    }
  }
```

When a command is complete, the view calls the `endCommand` method of the undo manager, which pushes `currentCommand` onto the history stack and sets `currentCommand` to `null`.

```
public void endCommand(Command command) {
  command.end();
  history.push(command);
  currentCommand = null;
  model.updateView();
}
```

Handling the input The view declares one button class for each command (add label, draw line, etc.). The class for handling line drawing is implemented as below.

```
public class LineButton extends JButton implements ActionListener {
  // fields for view, drawing panel, handlers, etc.
  public LineButton(UndoManager undoManager, View jFrame, JPanel jPanel) {
    // store the parameters and create the mouse listener
  }
  public void actionPerformed(ActionEvent event) {
    // change the cursor
    drawingPanel.addMouseListener(mouseHandler);
  }
  private class MouseHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent event) {
      if (first point) {
        lineCommand = new LineCommand(event.getPoint());
        UndoManager.instance().beginCommand(lineCommand);
        } else if (second point) {
        lineCommand.setLinePoint(event.getPoint());
        drawingPanel.removeMouseListener(this);
        view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
        UndoManager.instance().endCommand(lineCommand);
      }
    }
  }
}
```

The above class thus directly creates the appropriate command object when a request

comes from a user.

4.8 Drawing Incomplete Items

Recall the terms incomplete item and complete item we introduced in the previous section. There are a couple of reasons why in the drawing program we might wish to distinguish between these two types of items.

1. Incomplete items might be rendered differently from complete items. For instance, for a line, after the first click, the UI could track the mouse movement and draw a line between the first click point and the current mouse location; this line keeps shifting as the user moves the mouse. Likewise, if we were to extend the program to include triangles, which need three clicks, one side may be displayed after two clicks. Labels in construction must show the insertion point for the next character.
2. Some fields in an incomplete item might not have 'proper' values. Consequently, rendering an incomplete item could be more tricky. An incomplete line, for instance, might have one of the endpoints null. In such cases, it is inefficient to use the same render method for both incomplete items and complete items because that method will need to check whether the fields are valid and take appropriate actions to handle these special cases. Since we ensure that there is at most one incomplete item, this is not a sound approach.

We can easily distinguish between incomplete items and complete items by having a field that identifies the type. The render method will behave differently based on this field. The approach would be along the following lines.

```
public class Line {
  private boolean incomplete = true;
  public boolean isIncomplete() {
    return incomplete;
  }
  // other fields and methods
}

public class NewSwingUI implements UIContext {
  // fields and methods
  public void draw(Line line) {
    if (line.isIncomplete()) {
       draw incomplete line;
    } else {
       draw complete line;
    }
  }
}
```

In circumstances such as the above, where we have variant behaviour based on field values, the object-oriented philosophy dictates subclassing, i.e., we treat the incomplete item as a different class of object with its own rendering method. We create classes for incomplete items (such as `IncompleteLabel`) that are subclasses

of items (such as `Label`). Since the class `IncompleteLabel` is a subclass of `Label`, the model is unaware of its existence. Once the object is created, the incomplete object can be removed from the model.

The details are as follows.

```
import java.awt.*;
public class IncompleteLabel extends Label {
  public IncompleteLabel(Point point) {
    super(point);
  }
  public void render() {
    // code for rendering IncompleteLabel
  }
  public boolean includes(Point point) {
    return false;
  }
}
```

One problem we face with the above approach is that `UIContext` must include the method(s) for drawing the incomplete items (`draw(IncompleteLabel label)`, in our example). This suggests that `UIContext` needs to be modified. However, the manner in which incomplete items are rendered is an issue that largely relates to the look and feel of the system. For instance, `UIContext`might not have a method `draw(IncompleteLine line)` and creator of some view (`NewSwingUI`, for instance) might wish to include that. In general, we would like a solution that allows for a *customised presentation* which may require subclassing the behaviour of some concrete items. This can be accomplished through RTTI. In particular, the situation where the `NewSwingUI` wants its own method for drawing an incomplete line is implemented as follows:

```
public class NewSwingUI implements UIContext {
  // fields and methods
  public void draw(Line line) {
    if (line instanceof IncompleteLine) {
       this.draw((IncompleteLine) line);
    } else {
       //code to draw Line
    }
  }
}
```

The `LineCommand` object creates an `IncompleteLine` and adds this to the model. This new class is thus known only to the controller and `NewSwingUI`. When the label creation is complete, the `IncompleteLine` object is removed from the model and replaced with a `Line` object. This implementation therefore gives a solution where variability is contained.

Finally, we examine item creation in this new context. Assume that the user clicks on the ‘Add Label’ button. On the creation of the `LabelCommand` object, an `IncompleteLabel` object is created and stored within the command object. When label is completed, the `end` method of the command object is called, and

in this method, a `Label` object is created and data from the incomplete version is copied to it. The `IncompleteLabel` object is deleted from the model and the `Label` object takes its place. The relevant code from `LabelCommand` is shown below.

```
public void end() {
  model.removeItem(label);
  String text = label.getText();
  label = new Label(label.getStartingPoint());
  for (int index = 0; index < text.length(); index++) {
    label.addCharacter(text.charAt(index));
  }
  execute();
}
```

This completes the basic implementation of our simple graphical system. Note that if any new operation has to be added, all we have to do is create new classes that extend `Command` and `Item`, and modify the view to allow the user to invoke the new operation. Modifying the view is simply a matter of defining a new class that extends `JButton` and adding an instance of this class to the button panel. The model, the view and the controller are essentially repositories for the items, buttons, and commands respectively, and thus provide a framework for creating the specified system.

Adding a New Feature

Most interactive systems that are used to create graphical objects, allow users to define new kinds of objects on the fly. A system for writing sheet music may allow a user to define a sequence of notes as a group. This would enable the user to manipulate these notes as a group, making copies of these as needed. In a system for drawing electrical circuits, a set of components interconnected in a particular way could be clustered together as a 'sub-circuit' that can then be treated as a single unit. In a drawing program like the one we have created, a complex figure may be created as a collection of lines and circles, which may have to be moved around a single unit. In all these cases, the user-friendliness of the system would be considerably improved if a feature is provided to enable such operations.

Let us examine how our system needs to be modified to accommodate this. The process for creating such a 'compound' object would be as follows: *The user would select the items that have to be combined by clicking on them. The system would then highlight the selected items. The user then requests the operation of combing the selected items into a compound object, and the system combines them into one.*

Which Subsystem 'Owns' a Class?

In our original approach to designing this system using the MVC architecture, we were partitioning the responsibilities between the three subsystems. As we looked into the finer details of the implementation, we encountered some problems and found some suitable patterns that could improve our design. The

use of these patterns, however apparently 'blurs' some of the clear boundaries.

Consider for instance the bridge pattern. We created the `UIContext` interface within the model to house the `draw` methods of all theitems. The model does not have the information, however, to create a concrete instance of `UIContext` and this task is left to the `View` class. `UIContext` and its implementing classes belong to the view subsystem.

The original controller was replaced by a collection of classes including `UndoManager` and the various subclasses of `Command`, so they could be considered belonging to the controller subsystem. The undo manager defines the interface for the command but does not have any information on how each individual command should receive and process input.

The reader should realise that the subsystems are only providing a context within which the details can be fleshed out. The controller is providing a format for the creation of commands and also a system that manages these commands. When a command has to be added, a class is defined and the view is modified to allow for its invocation. Likewise the model provides a template for rendering all the kinds of items, but a complete knowledge of the view is needed to provide a concrete implementation.

Once a compound object has been created, it can be treated as a any other object. This process can be *iterated,* i.e., a compound object can be combined with other objects (which could themselves be compound or simple objects) to create another compound object. The system also allows the user to 'breakdown' a compound item into its constituent items by first selecting the item(s) to be broken down and then choosing the 'decompose' operation. Note that if a compound item is created by combining two compound items, then decomposing it will give us back the two original compound items. Finally, the system must have the ability to undo and redo these operations.

Since we have to store a collection of items, an obvious approach to implementing this would be to create a newkind of item that maintains a collection of the constituent items. This would be a concrete class and would look like this:

```
public class CompoundItem {
  List items;
  public CompoundItem(/* parameters */) {
    //instantiate lists
  }
  public Enumeration getItems() {
    //returns an enumeration of the objects in Items
  }
  // other fields and methods
}
```

Since `items` consists of both simple items and compound items, it seems logical that all entities stored in `items` are designated as belonging to the class `Object`. The model would also have to be modified so that the container classes would hold collections of type `Object`.

Consider now any class that examines at the collection of items in the model (i.e., a 'client' class). One of these would be the `SelectCommand`. When a `SelectCommand` object gets the coordinates of the mouse click, it iterates through the collection in the model to determine the selected item. If the object is a simple item, it would be cast as an `Item` and the `includes` method would be invoked; if the object is a compound item, it would be cast as a `CompoundItem` and the `getItems` method would be invoked to get an enumeration of the objects that make up the compound item. Clearly, this is not the most desirable state of affairs since the client method is querying the type of the object (which is akin to switching on the fields of the object) to determine what operation is to be performed. Our standard approach in such situations is to create an inheritance hierarchy and use dynamic

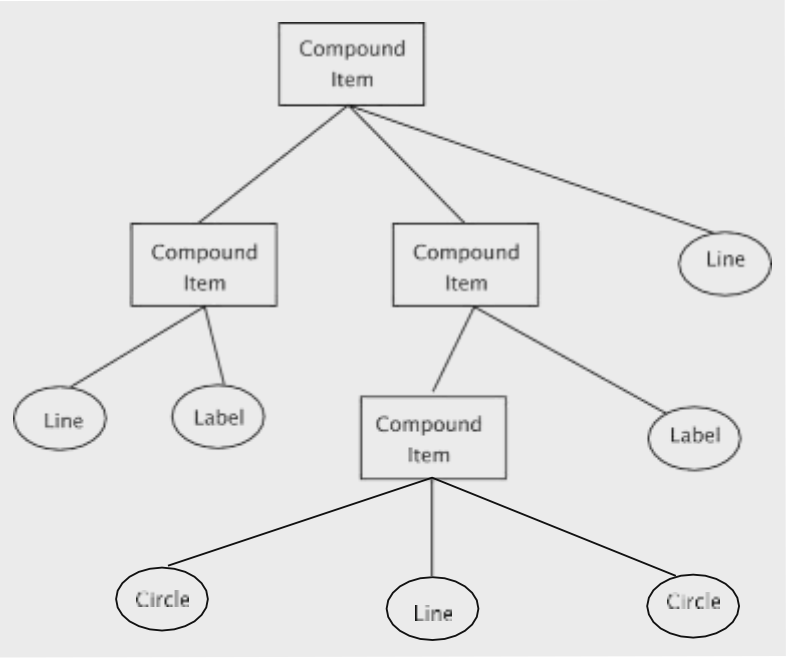


Figure. 4.20 Tree structure formed by compound items

binding. The dilemma here is that we have a two fundamentally different kinds of entities: *a simple item is a single item, whereas a compound item is a collection of items.* The **composite pattern** gives us an elegant solution to this problem.

The intent of the composite pattern is as follows (see footnote 1):

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

A compound item is clearly a composition of simple items. Since each compound itemcould itself consist of other compound items, we havethe requisite tree structure.

The class interaction diagram for the composite pattern is shown in Fig. 4.21. Note that the definition of the compound item is *recursive* and may remind readers of the recursive definition of a tree. Following this diagram, the class `CompoundItem` is redefined as follows:

```
public class CompoundItem extends Item {
  List items;
  public CompoundItem(/* parameters */){
    //instantiate lists
  }
  public void render(){
    // iterates through items and renders each one.
  }
```

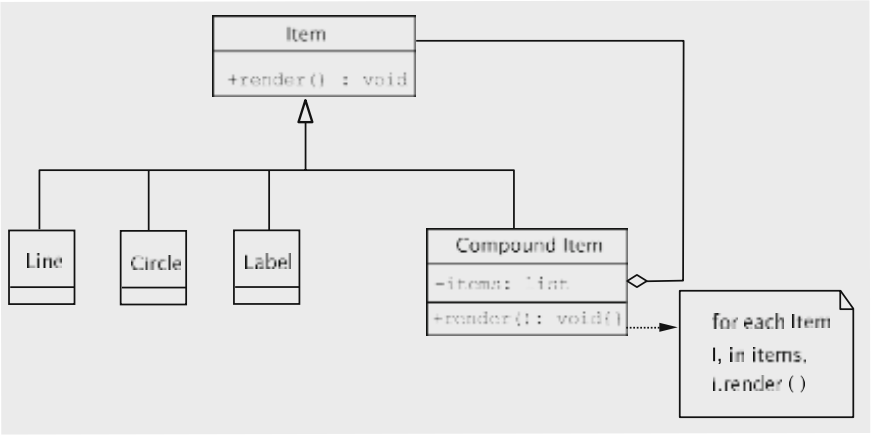


Figure. 4.21 Composite structure of the item hierarchy

```
  public boolean includes(Point point) {
    /* iterates through items and invokes includes on each item.
       Returns true if any of the items returns true and false otherwise. */
  }
  public void addItem(Item item) {
    // Adds item to items
  }
    // other fields and methods
}
```

Modifying the system to allow for creating compound objects is just like any of the operations discussed earlier. The system already has an operation for selecting items. Once that is complete, user chooses the 'create composite' operation. This would require that a new class be defined (extending `JButton`) and that the view be modified to add this button to the button panel. A new class, `CompositeCommand`

(extending `Command`) is defined. The `execute` method of this class removes all the selected items from the Model and adds them to a new `CompoundItem` object, which is then added to the Model. The view renders a `CompoundItem` exactly in the same way as it renders any other instance of `Item`. Note also that the select operation invokes the `includes` method on `CompoundItem` exactly as it would on simple items.

4.9 Pattern-Based Solutions

As explained earlier a pattern is a solution template that addresses a recurring problem in specific situations. In a general sense, these could apply to any domain. In the context of creating software, three kinds of patterns have been identified. At the highest level, we have the **architectural patterns**. These typically partition a system into subsystems and broadly define the role that each subsystem plays and how they all fit together.

Architectural patterns have the following characteristics:

- *They have evolved over time* In the early years of software development, it was not very clear to the designers how systems should be laid out. Over time, some kind of categorization emerged, of the kinds software systems that are needed. In due course, it became clearer as to how these systems and the demands on them change over their lifetime. This enabled practitioners to figure out what kind of layout could alleviate some of the commonly encountered problems.
- *A given pattern is usually applicable for a certain class of software system* The MVC pattern for instance, is well-suited for interactive systems, but might be a poor fit for designing a payroll program that prints paychecks.
- *The need for these is notobvious to the untrained eye* When a designer first encounters a new class of software, it is not very obvious what the architecture should be. One reason for this is that the designer is not aware of how the requirements might change over time, or what kind of modifications are likely to be needed. It is therefore prudent to follow the dictates of the wisdom of past practitioners. This is somewhat different from design patterns, which we are able to 'derive' by applying some of the well-established 'axioms' of object-oriented analysis and design. (In case of our MVC example, we did justify the choice of the architecture, but this was done by demonstrating that it would be easier to add new operations to the system. Such an understanding is usually something that is acquired over the lifetime of a system.)

At the next level, we have the ***design patterns***. These solve problems that could appear in many kinds of software systems. Once the principles of object-oriented analysis and design have been established it is easier to derive these. Examples of

these can be found throughout this text.

At the lowest level we have the patterns that are called **idioms**. Idioms are the patterns of programming and are usually associated with specific languages. They typically refer to the use of certain syntactic elements of the language. As programmers, we often find ourselves using the same code snippet every time we have to accomplish a certain task. Sometimes, we may save these as 'macros' to be copied and pasted as needed thus enabling us to be more productive in terms of code-generation. Idioms are something like these, but they are usually carefully designed to take the language features (and quirks!) into account to make sure that the code is safe and efficient. The following code, for instance, is commonly used toswap:

```
temp = a;
a = b;
b = temp;
```

In *Perl*, the list assignment syntax allows us to employ a more succinct expression:

```
($a, $b) = ($b, $a);
```

This would be an example of an idiom for Perl. In addition to safety and efficiency, the familiarity of the code snippet makes the code more readable and reduces the need for comments. Typical Perl programmers might be more comfortable with the second whereas a Java programmer would prefer the first.

Not all idioms are without conflict. There are two possible idioms for an infinite loop:

```
for (;;) {
// some code
}
while (true) {
// some code
}
```

It has been argued that the first one should be preferred for efficiency, since no expression evaluation is involved at the end of each iteration. However, with the availability of optimising compilers and increasing hardware capacity nowadays, some programmers are making a case for the second one based on readability and elegance.

4.10.1 Examples of Architectural Patterns

The Repository

This architecture is characterized by the presence of a single data structure called the *central repository*. Subsystems access and modify the data stored in this. An example of such a system could be software used for *managing an airline.* The subsystems in this case could be the ones for managing reservations, scheduling staff, and scheduling aircraft. All of these would access a central data repository that holds information about aircraft, staff, and passengers. These would be interrelated, since a choice of an aircraft could likely influence the choice of staff and

be influenced by the volume of passenger traffic. In such systems, the control flow can be dictated by the central repository (changes in the data characteristics could trigger some operations), or from one of the subsystems. Another application of such a system could be for managing a large bank. The account information would have to be centrally located and could be accessed and modified from several peripheral locations. A software development system or a compiler could also employ such an architecture by having a centralized parse-tree or symbol table.

The Client-Server

In such a layout, there is a central subsystem known as a *server* and several smaller subsystems known as *clients* which are typically quite similar. There is a fair amount of independence in the control flow, and each subsystem may be using a different thread. Synchronisation techniques are often employed to manage requests and transmit results.

The world-wide-web is probably the best example of such an architecture. The browsers running on PCs are like clients and the sites they access play the role of servers. The server could also be housing a database and the clients could be processes that are querying and updating the database. A variant/generalisation of this is the **peer-to-peer** architecture where the client/server role of the subsystems are interchangeable. These variants are typically hard to design due to the possibility of deadlocks and a myriad of other problems that can complicate the flow of control.

The Pipe and Filter

The system in this case is made up of *filters*, i.e., subsystems that process data, and *pipes*, which can be used to interconnect the filters. The filters are completely mutually independent and are aware only of the input data that comes through a pipe, i.e., the filter knows the form and content of the data that came in, not how it was generated. This kind of architecture produces a system that is very flexible and can be dynamically reconfigured. In their simplest form, the pipes could all be identical, and each filter could be performing a fixed task on data input stream. An example of this would be that of processing incoming/outgoing data packets over a computer network. Each 'layer' would be like a filter that adds to, subtracts from or modifies the packet and sends it forward.

The Unix operating system is a more sophisticated version of such an architecture, and allows the user to create more complex operations by linking together simpler ones. In its most general form, one could have pipes that 'reformat' the data, so that any sequence of filters could be used.

MODULE 5

Designing with Distributed Objects

Definition: Businesses usually install multiple computer systems that are interconnected by communication links, and applications run across a network of computers rather than on a single machine. Such systems are called **distributed systems.**

Advantages

a. It is more economical and efficient to process data at the point of origin.
b. Distributed systems make it easier for users to access and share resources.
c. They also offer higher reliability and availability: failure of a single computer does not cripple the system as a whole.
d. It is also more cost effective to add more computing power.

Drawbacks

a. The software for implementing them is complex.
   Distributed systems must coordinate actions between a number of possibly heterogeneous computer systems; if data is replicated, the copies must be made mutually consistent.
b. Data access may be slow because information may have to be transferred across communication links.
c. Securing the data is a challenge.
   As data is distributed over multiple systems and transported over communication links, care must be taken to guarantee that it is not lost, corrupted, or stolen.

5.1 Client server system

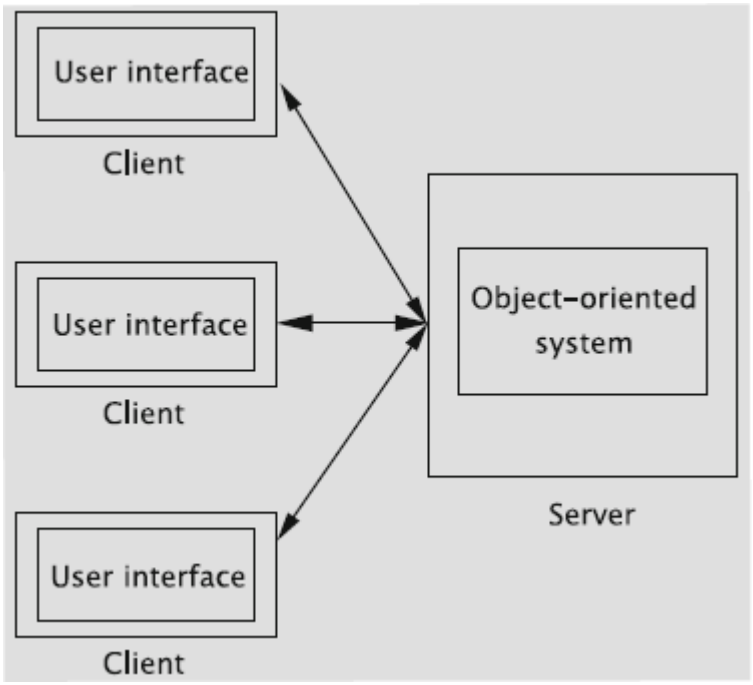
Distributed systems can be classified into :

1. **Peer-to-Peer systems** :
   Every computer system (or node) in the distributed system runs the same set of algorithms; they are all equals, in some sense
2. **Client-Server systems :**
   There are two types of nodes: clients and servers. A client machine sends requests to one or more servers, which process the requests, and return the results to the client.

5.1.1 Basic Architecture of Client/Server Systems

- Figure below shows a system with one server and three clients.
- Each client runs a program that provides a user interface, which may or not be a GUI.

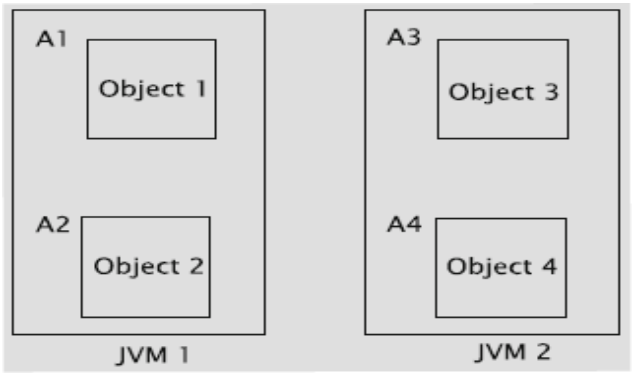
- The server hosts an object-oriented system.
- Like any other client/server system, clients send requests to the server, these requests are processed by the object-oriented system at the server, and the results are returned.
- The results are then shown to end-users via the user interface at the clients.



There is a basic difficulty in accessing objects running in a different Java Virtual Machine (JVM). Let us consider two JVMs hosting objects as in Fig. below.

- A single JVM has an address space part of which is allocated to objects living in it.

For example.



- Objects object 1 and object 2 are created in JVM 1 and are allocated at addresses A1 and A2 respectively. Similarly, objects object 3 and object 4 live in JVM 2 and are respectively allocated addresses A3 and A4.
- Code within Object 2 can access fields and methods in object 1 using address A1. However, addresses A3 and A4 that give the addresses of objects object 3 and object 4 in JVM 2 are meaningless within JVM 1.

This difficulty can be handled in one of two ways:

1. **By using object-oriented support software**:

The software solves the problem by the use of proxies that receive method calls on 'remote' objects, ship these calls, and then collect and return the results to the object that invoked the call. The client could have a custom-built piece of software that interacts with the server software. This approach is the basis of Java Remote Method Invocation.

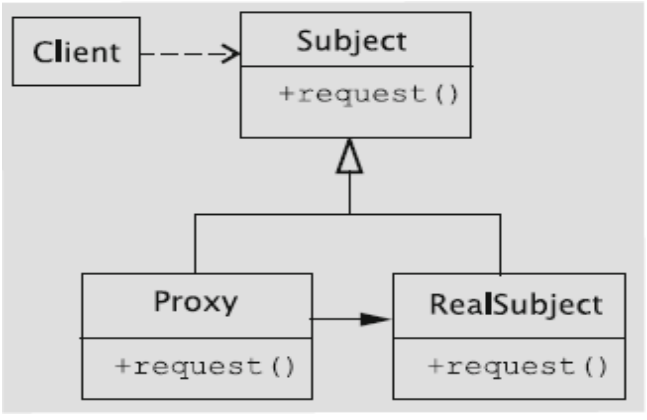
2. **By avoiding direct use of remote objects by using the Hyper Text Transfer Protocol (HTTP**).

The system sends requests and collects responses via encoded text messages. The object(s) to be used to accomplish the task, the parameters, etc., are all transmitted via these messages. This approach has the client employ an Internet browser, which is, of course, a piece of general-purpose software for accessing documents on the world-wide web.

5.2 Java Remote Method Invocation

The goal of Java RMI is to support the building of Client/Server systes where the server hosts an object-oriented system that the client can access programmatically.

- The objects at the server maintained for access by the client are termed **remote objects**.
- A client accesses a remote object by getting what is called a **remote reference** to the remote object.
- After that the client may invoke methods of the object.
- The basic idea behind RMI is to employ the **proxy** design pattern.
- Java RMI employs proxies to stand in for remote objects. All operations exported to remote sites (remote operations) are implemented by the proxy. Proxies are termed *stubs* in Java RMI. These stubs are created by the RMI compiler.



- The set-up is shown in above figure:
- When the client calls a remote method, the corresponding method of the proxy object is invoked. The proxy object then assembles a message that contains the remote object's identity, method name, and parameters. This assembly is called **marshalling**. In this process, the method call must be represented with enough information so that the remote site knows the object to be used, the method to be invoked, and the parameters to be supplied.
- When the message is received by it, the server performs **demarshalling**, whereby the process is reversed.

Setting up a remote object system is accomplished by the following steps:

1. Define the functionality that must be made available to clients. This is accomplished by creating remote interfaces.
2. Implement the remote interfaces via remote classes.
3. Create a server that serves the remote objects.
4. Set up the client.

5.2.1 Remote Interfaces

1. In the case of RMI, the functionality exported of a remote object is defined via what is called a ***remote interface.*** A remote interface is a Java interface that extends the interface java.rmi.Remote.
2. Clients are restricted to accessing methods defined in the remote interface. We call such method calls **remote method invocations.**

Remote method invocations can fail due to a number of reasons:

a. The remote object may have crashed,
b. the server may have failed, or
c. the communication link between the client and the server may not be operational, etc.

NOTE: Java RMI encapsulates such failures in the form of an object of type java.rmi.RemoteException; as a result, all remote methods must be declared to throw this exception.

```
import java.rmi.*;
public interface BookInterface extends Remote {
  public String getAuthor() throws RemoteException;
  public String getTitle() throws RemoteException;
  public String getId() throws RemoteException;
}
```

5.2.2 Implementing a Remote Interface

1. The next step is to implement via remote classes.
   - Parameters to and return values from a remote method may be of primitive type, of remote type, or of a local type.
   - All arguments to a remote object and all return values from a remote object must be serializable. Thus, they must implement the java.io.Serializable interface.
   - Parameters of non-remote types are passed by copy;
   - Intuitively, remote objects must somehow be capable of being transmitted over networks. A convenient way to accomplish this is to extend the class java.rmi.server.UnicastRemoteObject**.**

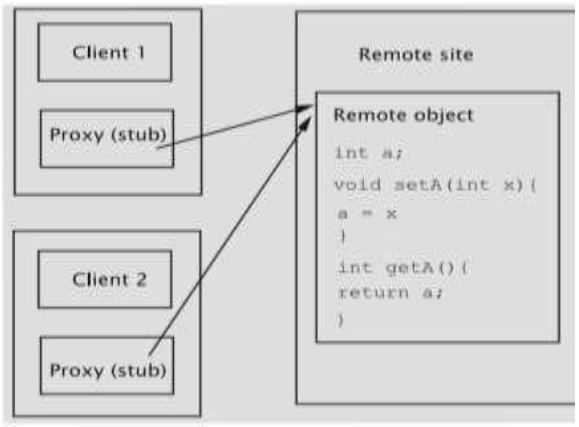
```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

public class Book extends UnicastRemoteObject implements
             BookInterface, Serializable {
   private String title;
   private String author;
   private String id;
   public Book(String title1, String author1, String id1)
          throws RemoteException {
     title = title1;
     author = author1;
     id = id1;
   }
   public String getAuthor()   throws RemoteException {
     return author;
   }
   public String getTitle()   throws RemoteException {
     return title;
   }
   public String getId()   throws RemoteException {
     return id;
   }
}
```

2. Since it is a remote class, Book must be compiled using the RMI compiler by invoking the command rmic as below.

   Rmic Book

   The compiler produces a file named Book_Stub.class, which acts as a proxy for calls to the methods of BookInterface. The stub contains a reference to the serialized object and implements all of the remote interfaces that the remote object implements. All calls to the remote interface go through the stub to the remote object.

3. Remote objects are thus passed by reference. This is depicted in Figure below:



- Here , we have a single remote object that is being accessed from two clients.
- Both clients maintain a reference to a stub object that points to the remote object that has a field named **a**.
- Suppose now that Client 1 invokes the method setA with parameter 5.
- The call goes through the stub to the remote object and gets executed changing the field a to 5. any changes made to the state of the object by remote method invocations are reflected in the original remote object.
- If the second client now invokes the method getA, the updated value 5 is returned to it.

NOTE: parameters or return values that are not remote objects are passed by value. Thus, any changes to the object's state by the client are reflected only in the client's copy, not in the server's instance.

5.2.3 Creating the Server

Before a remote object can be accessed, it must be instantiated and stored in an object registry, so that clients can obtain its reference. Such a registry is provided in the form of the class java.rmi.Naming. The method bind is used to register an object and has the following signature:

```
public static void bind(String nameInURL, Remote object)
      throws AlreadyBoundException, MalformedURLException,
      RemoteException
```

The first argument takes the form **//host:port/name** and is the URL of the object to be registered;

- *host* refers to the machine (remote or local) where the registry is located,
- *port* is the port number on which the registry accepts calls, and

- *name* is a simple string for distinguishing the object from the other objects in the registry.
- Both host and port may be omitted if its in localhost and port no is 1099.

The process of creating and binding the name is given below.

```
try {
  <interface-name> object = new <class-name>(parameters);
  Naming.rebind("//localhost:1099/SomeName", object);
} catch (Exception e) {
  System.out.println("Exception " + e);
}
```

The complete code for activating and storing the Book object is shown below.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
public class BookServer {
  public static void main(String[] s) {
    String name = "//localhost:1099/" + s[0];
    try {
      BookInterface book = new Book("t1", "a1", "id1");
      Naming.rebind(name, book);
    } catch (Exception e) {
      System.out.println("Exception " + e);
    }
  }
}
```

5.2.4 The Client

A client may get a reference to the remote object it wants to access in one of two ways:

1. It can obtain a reference from the Naming class using the method lookup.

```
SomeInterface object = (SomeInterface) Naming.lookup
                        ("//localhost:1099/SomeName");
```

2. It can get a reference as a return value from another method call.

In the following code, the getters of the Book Interface object are called and displayed.

```
import java.util.*;
import java.rmi.*;
import java.net.*;
import java.text.*;
import java.io.*;
public class BookUser {
  public static void main(String[] s) {
    try {
      String name = "//localhost/" + s[0];
      BookInterface book = (BookInterface) Naming.lookup(name);
      System.out.println(book.getTitle() + " " + book.getAuthor()
                          + " " + book.getId());
    } catch (Exception e) {
      System.out.println("Book RMI exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

5.2.5 Setting up the System

1. To run the system, create two directories, say server and client, and copy the files BookInterface.java, Book.java, and BookServer.java into server and the file BookUser.java into client.
2. Then compile the three Java files in server and then invoke the command

   rmic Book

3. This command creates the stub file Book_ Stub.class.
4. Copy the client program into client and compile it.
5. Run RMI registry and the server program using the following commands

```
start rmiregistry
java -Djava.rmi.server.codebase=file:C:\Server\BookServer
BookServer MyBook
```

- The first command starts the registry and
- The second causes the Book instance to be created and registered with the name MyBook.

6. Finally, run the client as below from the client directory.

```
java -Djava.rmi.server.codebase=file:C:\Client\BookUser
BookUser MyBook
```

7. The client code starts, looks up the object with the name MyBook, calls the object's getter methods, and displays the values.

5.3 Implementing an Object-Oriented System on the Web

5.3.1 HTML and Java Servlets

- System displays web pages via a browser has to create HTML code.
- HTML code displays text, graphics such as images, links that users can click to move to other web pages, and forms for the user to enter data.
- An HTML program can be thought of as containing a header, a body, and a trailer.

The header contains code like the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
   http-equiv="content-type">
  <title>A Web Page</title>
</head>
```

- The first four lines are usually written as given for any HTML file.
- observe words such as html and head that are enclosed between angled brackets (< and >). They are called tags.
- HTML tags usually occur in pairs: start tag that begins an entry and end tag that signals the entry's end. For example, the tag <head> begins the header and is ended by </head>.

- The text between the start and end tags is the element content.
- In the fifth line we see the tag title, which defines the string that is displayed in the title bar.

As a sample body, let us consider the following.

```
<body>
<h1>
    <span style="color: rgb(0, 0, 255);">
    <span style="font-family: lucida bright;">
    <span style="font-style: italic;">
    <span style="font-weight: bold;">
       An Application
    </span>
    </span>
    </span>
    </span>
</h1>
</body>
```

- The body contains code that determines what gets displayed in the browser's window.
- Some tags may have attributes, which provide additional information.

For example

```
<span style="color: rgb(0, 0, 255);">
```

- The tag span has its attribute style modified, so that the text will be in blue colour.
- Attributes always come in name/value pairs of the form name="value".
- They are always specified in the start tag of an HTML element.
- The body contains code to display the string An Application in the font Lucida bright, bolded, italicised, and in blue color.
- The last line of the file is : </html> it ends the HTML file

Entering and Processing Data

The complete code that allows us to enter some piece of text in the web page is given below.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
 <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
 <title>Sample Form</title>
</head>
<body>
  <form action="/servlet/apackage.ProcessInput" method="post">
    <table>
       <tr>
           <td align="right">Enter Data:</td>
           <td><input type="text" name="userInput"></td>
       </tr>
       <tr>
           <td><input type="submit" value="Process"></td>
       </tr>
    </table>
  </form>
</body>
</html>
```

The code that begins with the line :

<form action="/servlet/apackage.ProcessInput" method="post">

- The tag **form** begins the specification of a set of elements that allow the user to enter information.
- The **action** attribute specifies that the information entered by the user is to be processed by a Java class called ProcessInput.class, which resides in the package apackage.
- The tag <table> begins the creation of a table.
- Each row of the table is described using the tag <tr>, and the tag <td> defines a cell in the table.
- <input> tag has two attributes : type and name
  - **type:** which specifies what is the type of input : "text", which means plain text or "password", which makes the entry unreadable on the screen.
  - **name:** must be given a unique value

<td><input type="text" name="userInput"></td>

- a button of type "submit", which when clicked causes the form data to be sent to the server. The button has the label Process.

```
<td><input type="submit" value="Process"></td>
```

There are two primary ways in which form data is encoded by the browser:

1. **GET :** GET means that form data is to be encoded into a URL
2. **POST:** POST makes data appear within the message itself.

Server-Side Code

- The server-side code **ProcessInput** is an example of a **servlet,** which uses the request- response paradigm.
- Servlets can process data sent using the HTTP protocol via a form.
- They can handle multiple requests concurrently.
- We create a servlet by extending the class HttpServlet as below.

```
public class ProcessInput extends HttpServlet {
```

- Since we transmitted form data using the POST method, we need to override a method called **doPost.**
  - o This method has two parameters, **request and response** that respectively encapsulate the data sent by the client and the response to the client.
- The header of the doPost method is given below.

```
public void doPost(HttpServletRequest request, HttpServletResponse response) throws
IOException, ServletException {
```

- Data sent by the client through the form is retrieved using the request object as below: String input = request.getParameter("userInput");
- Afterthedataiscapturedandprocessed,theservletcreatesanHTMLpageusing the response object as below.

```
response.setContentType("text/html");
response.getWriter().println("<!DOCTYPE html PUBLIC \"-//W3C//DTD "+
                             "HTML 4.01 Transitional//EN\">");
```

- The first line states that the data is HTML and the second line begins the HTML code.
- The complete code for the servlet is given below

```
package apackage;
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
public class ProcessInput extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response) throws IOException,
                     ServletException {
    String input = request.getParameter("userInput");
    response.setContentType("text/html");
    response.getWriter().println("<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
                                 "HTML 4.01 Transitional//EN\">");
    response.getWriter().println("<html>");
    response.getWriter().println("<head>");
    response.getWriter().println("<meta content=\"text/html;" +
                                 " charset=ISO-8859-1\"" +
                                 "http-equiv=\"content-type\">");
    response.getWriter().println("<title>Response to Input</title>");
    response.getWriter().println("</head>");
    response.getWriter().println("<body>");
    response.getWriter().println("You entered " + input);
    response.getWriter().println("</body>");
    response.getWriter().println("</html>");
  }
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response) throws IOException,
                    ServletException {
    doPost(request, response);
  }
}
```

5.3.2 Deploying the Library System on the World-Wide Web

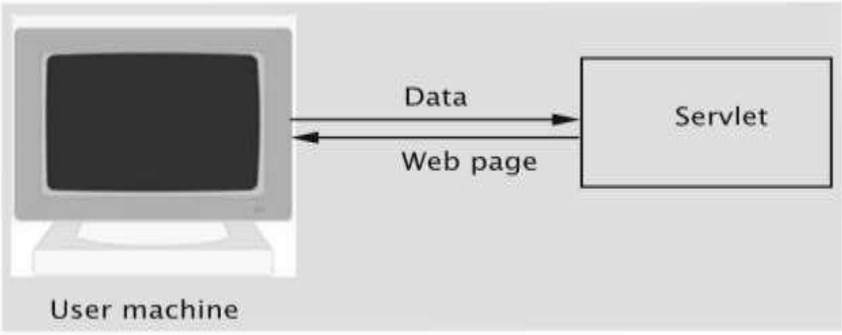


Figure 5.1: How servlets and HTML cooperate to serve web pages

Developing User Requirements

First task is to determine the system requirements : Example **Library system**

1. The user must be able to type in a URL in the browser and connect to the library system.

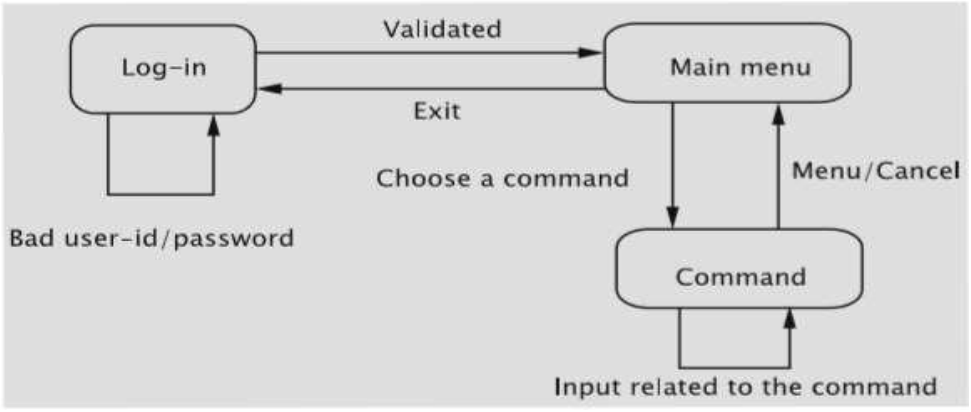
2. Users are classified into two categories:
   a. super users :
      - Superusers are essentially designated library employees, and ordinary members are the general public who borrow library books. superusers can execute any command when logged in from a terminal in the library.
   b. Ordinary members.
      - Ordinary members are the general public who borrow library books.
      - Ordinary members cannot access some 'privileged commands'.

   In particular, the division is as follows:
      a. Only superusers can issue the following commands: add a member, add a book, return a book, remove a book, process holds, save data to disk, and retrieve data from disk.
      b. Ordinary members and super users may invoke the following commands: issue and renew books, place and remove holds, and print transactions.
      c. Every user eventually issues the exit command to terminate his/her session.
3. Some commands can be issued from the library only. These include all of the commands that only the superuser has access to and the command to issue books.
4. A superuser cannot issue any commands from outside of the library. They can log in, but the only command choice will be to exit the system.
5. Superusers have special user ids and corresponding password. For regular members, their library member id will be their user id and their phone number will be the password.

Logging in and the Initial Menu

Figure below shows the process of logging in to the system.

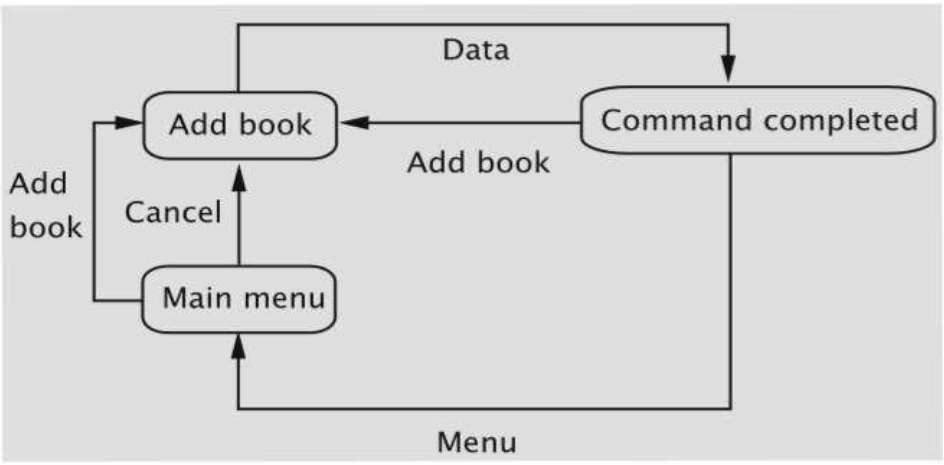
- When the user types in the URL to access the library system, the log in screen that asks for the user id and password is displayed on the browser.
- If a valid combination is typed in, an appropriate menu is displayed.
- What is in the menu depends on whether the user is an ordinary member or a superuser and whether the terminal is in the library or is outside.



1. The Issue Book command is available only if the user logs in from a terminal in the library.
2. Commands to place a hold, remove a hold, print transactions, and renew books are available to members of the library (not superusers) from anywhere.
3. Certain commands are available only to superusers who log in from a library terminal: these are for returning or deleting books, adding members and books, processing holds, and saving data to and retrieving data from disk.

Add Book

The State transition diagram for adding book is shown below:



- When the command to add a book is chosen, the system constructs the initial screen to add a book, which should contain three fields for entering the title, author, and id of the book, and then display it and enter the Add Bookstate.

- By clicking on a button, it should be possible for the user to submit these values to system.
- The system must then call the appropriate method in the Library class to create a Book object and enter it into the catalog.
- The result of the operation is displayed in the Command Completed state
- From the Command Completed state, the system must allow the user to add another book or go back to the menu.

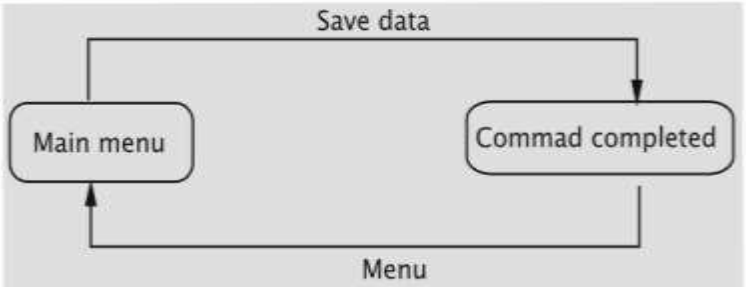
- In the Add Book state, the user has the option to cancel the operation and go back to the main menu.

Add Member, Return Book, Remove Book

- We need to accept some input (member details or book id) from the user, access the Library object to invoke one of its methods, and display the result.

Save Data

State transition diagram for saving data



- When the data is to be written to disk, no further input is required from the user.
- The system should carry out the task and print a message about the outcome.

Issue Book

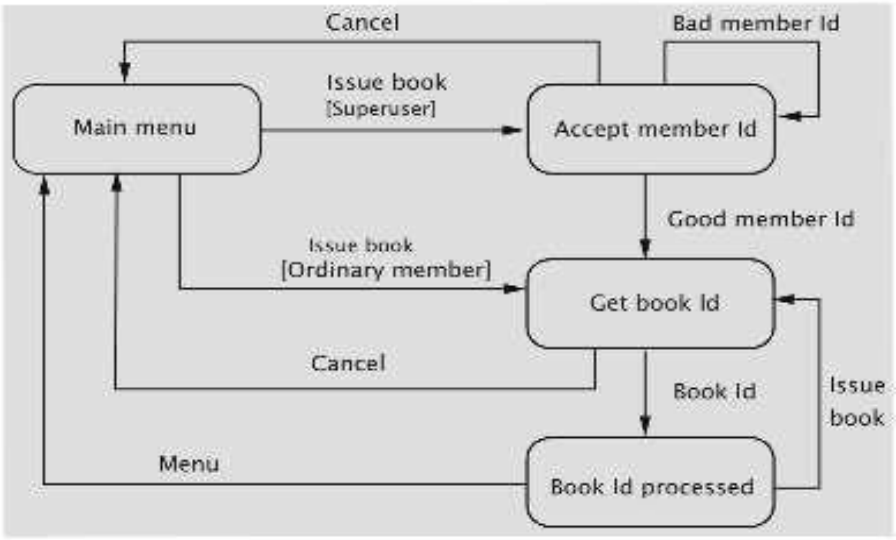
A book may be checked out in two different ways:

First, a member is allowed to check it out himself/herself. Here the system already has the user's member id, so that should not be asked again.

Second, he/she may give the book to a library staff member, who checks out the book for the member. Here the library staff member needs to input the membered to thesystem followed by the book id.

- After receiving a bookid, the system must attempt to check out the book.
- Whether the operation is successful or not, the system enters the Book Id Processed state. Complexity arises from the fact that any number of books may be checked out. Thus, after each book is checked out, the system must ask if more books need to be issued or not.
- The system must either go to the Get Book Id state for one more book id or to the Main Menu state.
- As usual, it should be possible to cancel the operation at any time.

State transition diagram for issuing books

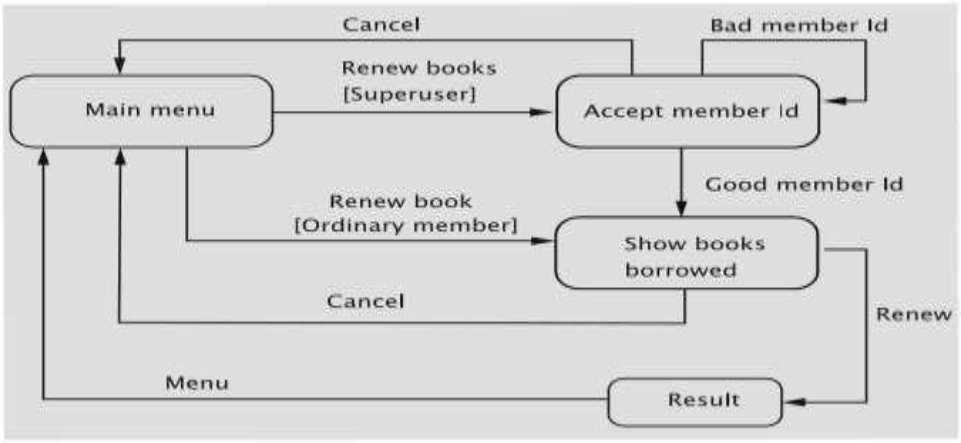


Renew Books

- The system must list the title and due date of all the books loaned to the member.
- For each book, the system must also present a choice to the user to renew the book.
- After making the choices, the member clicks a button to send any renew requests to the system.

- For every book renewal request, the system must display the title, the due date (possibly changed because of renewal), and a message that indicates whether the renewal request was honoured.
- After viewing the results, the member uses a link on the page to navigate to the main menu.

The state transition diagram is given in Figure below



Design and Implementation

To deploy the system on the web, we need the following:

1. Classes associated with the library
2. Permanent data (created by the save command) that stores information about the members, books, who borrowed what, holds, etc.
3. HTML files that support a GUI for displaying information on a browser and collecting data entered by the user. For example, when a book is to be returned, a screen that asks for the book id should pop up on the browser. This screen will have a prompt to enter the book id, a space for typing in the same, and a button to submit the data to the system.
4. A set of files that interface between the GUI ((3) above) and the objects that actually do the processing ((1) above). Servlets will be used to accomplish this task

Structuring the files HTML code for delivery to the browser can be generated in one of two ways:

1. Embed the HTML code in the servlets. This has the disadvantage of making the servlets hard to read, but more dynamic code can be produced.

2. Read the HTML files from disk as a string and send the string to the browser. This is less flexible because the code remains static.

We attempt to combine the two approaches so as to utilize the advantages

1. Create a separate HTML file for every type of page that needs to be displayed. For example, create a file for entering the id of the book to be returned, a second file for displaying the result of returning the book, a third file for inputting the id of the book to be removed, a fourth one for displaying the result of removing the book, etc.
2. Exploit the commonalities between the commands and create a number of HTML code fragments, a subset of which can be assembled to form an HTML file suitable for a specific context.

- The **first option** has the advantage of simplicity. However a rough calculation shows that at least 28 files are needed.
- Although the **second option** is more involved because of the need to assemble a big file from several fragments, we find that it presents some advantages over the first.

  First, it reduces the number of files somewhat and

  Avoids duplication of files.

  For example, to change the way the library's name is displayed in the screens, every one of the HTML files will need to be updated! We thus opt for the second choice.

Examples of HTML file fragment

Consider the two commands, one for returning and the other for removing books

- In both, the user must be presented with a web page that asks him/her to enter a book id. We have just one file that displays this page.
- However, the servlet that needs to be invoked will change depending on the context. Therefore, we code the servlet name as below.

  <form action="GOTO_WITH_BOOKID" method="post">

- By simply changing the string GOTO_WITH_BOOKID in the servlet, we can use the same HTML file in multiple situations

A similar approach is taken for accepting member ids.

- For every webpage, the header should display a title that depends on the context. We maintain just one file for the header. This file has a string TITLE that stands for the title of the web page. Depending on which page is being displayed, TITLE is replaced by an appropriate string, which gets displayed in the title bar.
- When a command is completed, we need to display a web page.
- we employ just one file, commandCompleted.html, to carry out this task. This file is adapted, however, in two different ways.

1. The result to be displayed will vary on the command as well as whether the operation was successful. To take care of this, the file has a string called RESULT

   <h3> RESULT
</h3>

Pseudocode

```
String result;
Member member;
String htmlFile = getFile("commandCompleted.html"):
if ((member = library.addMember(name, address, phone)) == null) {
  htmlFile = htmlFile.replace("TITLE", "Member not Added");
  result = "Member could not be added";
} else {
  htmlFile = htmlFile.replace("TITLE", "Member Added");
  result = member.getName() + " ID: " + member.getId() + " added";
}
htmlFile = htmlFile.replace("RESULT", result);
```

2. To reduce the number of mouse clicks, the user may be given the option to repeat the command whose result is displayed by the commandCompleted.html file. For example
   ,after completing the Add Book command, we need to give an option to issue the command once again so that the user can add another book.

   REPLACE_COMMAND

How to remember a user

- Servlets typically deal with multiple users.
- When a servlet receives data from a browser, it must somehow figure out which user sent the message, what the user's privileges are, etc.
- Each request from the browser to the server starts a new connection, and once the request is served, the connection is torn down.
- However, typical web transactions involve multiple request–response pairs. This makes the process of remembering the user associated with a connection somewhat difficult without extra support from the system.

- The system provides the necessary support by means of what are known as sessions, which are of type HttpSession.
- When it receives a request from a browser, the servlet may call the method getSession() on the HttpServletRequest object to create a session object, or if a session is already associated with the request, to get a reference to it.
- To check if a session is associated with the request and to optionally create one, a variant of this method getSession(boolean create) may be used.
- If the value false is passed to this method and the request has no valid HttpSession, this method returns null. When a user logs in, the system creates a session object as below.

```
HttpSession session = request.getSession();
```

- When the user logs out, the session is removed as below.

```
session.invalidate();
```

- The following code evaluates to true if the user does not have asession: that is, the user has not logged in:

```
request.getSession(false) == null
```

A session object can be used to store information about the session.

1. void setAttribute(String name, Object value)

   This command binds value, the object given in the second parameter, to the attribute specified in name. By setting the second parameter to null, the attribute can be removed.

2. Object getAttribute(String name)

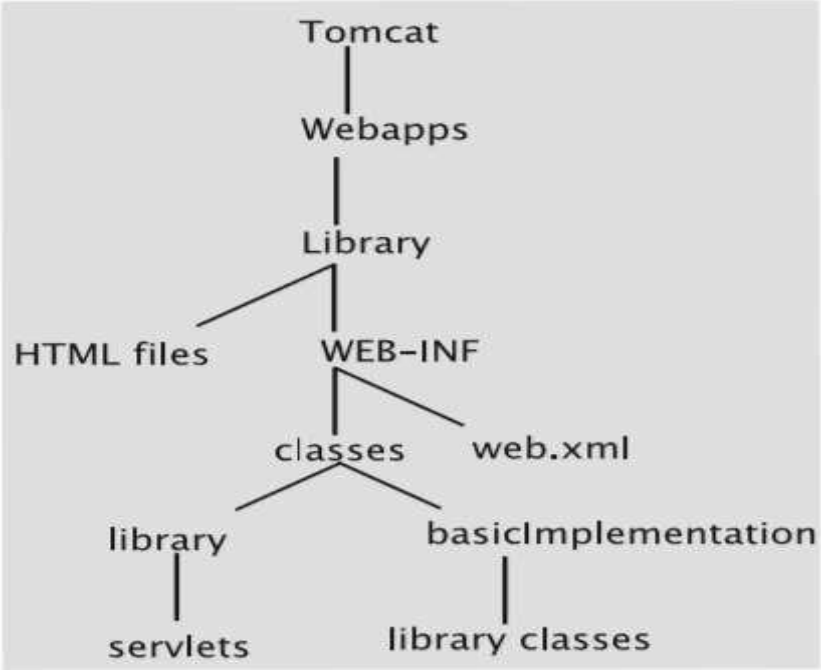
   The attribute value associated with name is returned.

3. void removeAttribute(String name)

   This method deletes the specified attribute from this session.

Configuration

- The server runs with the support of Apache Tomcat, which is a servlet container.
- A servlet container is a program that supports servlet execution.
- The servlets themselves are registered with the servlet container.
- URL requests made by a user are converted to specific servlet requests by the servlet container. The servlet container is responsible for initializing the servlets and delivering requests made by the client browser to the appropriate servlet.
- The directory structure is as in Figure below:



- We store the HTML files in a directory named Library, which is a subdirectory of webapps, which, in turn, is a subdirectory of the home directory of Tomcat.
- The servlets are in the package library, which is stored in Library/WEB-INF/classes.
- The implementation of the backend classes such as Member, Catalog, etc. is in the package basicImplementation.
- Our implementation requires that the user create an environment variable named LIBRARY-HOME that has as value the absolute path name of the directory that houses the HTML files.

- The deployment descriptor elements are defined in a file called web.xml. While this file permits a large number of tags, our use of them is limited to mapping the URLs to servlets. To understand how this is done, first examine the following lines of XML code.

```
<servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
</servlet-mapping>
```

- Thus when we write code such as **URL=login** in the HTMLfile, the string login is mapped to the servlet name LoginServlet.

- But the servlet name given by the tag <servlet-name> is just a name that is mapped to the fully-qualified class name of the servlet as below.

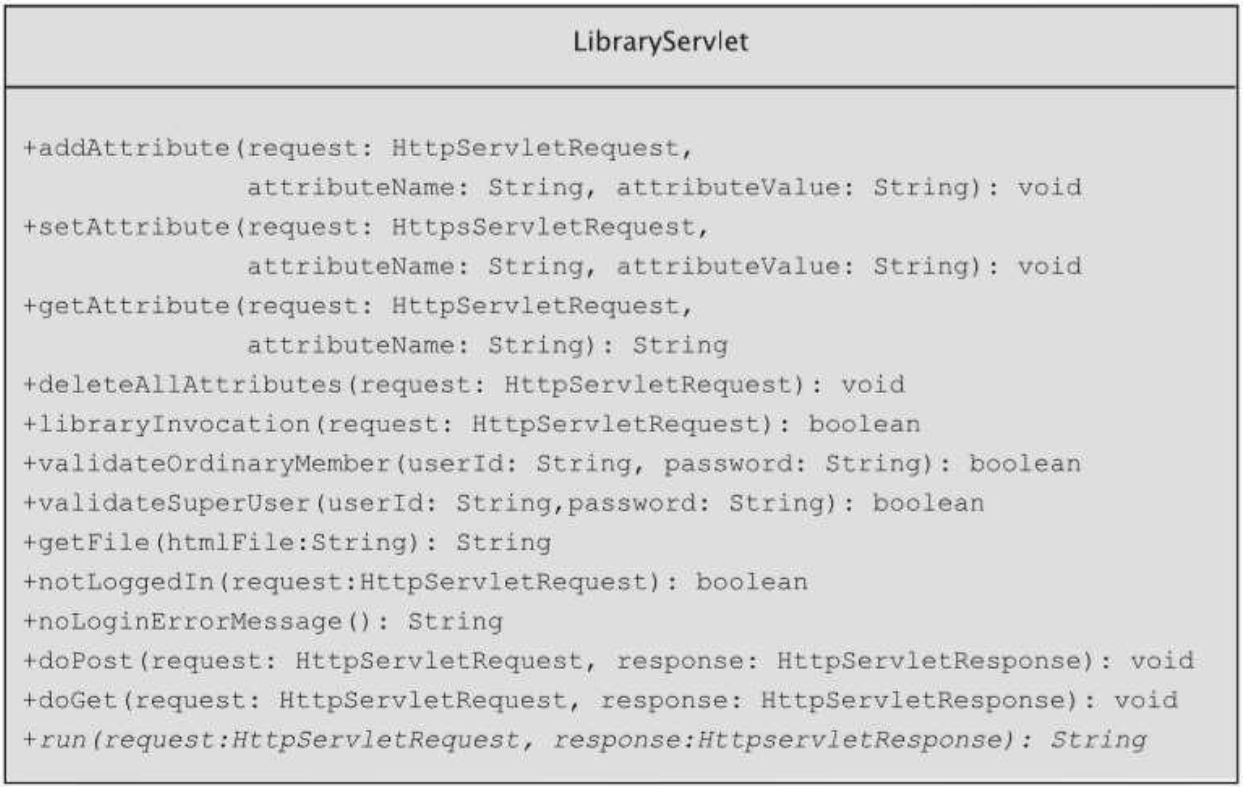
```
<servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>library.Login</servlet-class>
</servlet>
```

- The list of superusers and their passwords is stored in a file named Privileged Users.
- The IP addresses of all client machines located in the library are listed in a file named IPAddresses.
- Both files are to be stored in the same directory that has the HTML files.
- To run the system, first Tomcat needs to be started and then the library system needs to be accessed from a browser by typing in the URL of the Tomcat home concatenated with
  /Library.
- The file index.html in the library directory is then accessed; this file directs the request to the servlet Login.

Structure of servlets in the web-based library system

- A servlet receives data from a browser through a HttpServletRequestobject. This involves parameter names and their values, IP address of the user, and so on.
  For example, when the form to add book is filled and the Add button is clicked, the servlet's doPost method is invoked. As we have seen earlier, this method has two parameters: a request parameter of type HttpServletRequest and a response parameter of type HttpServletResponse.
- Each command is organized as a combination of one to three servlets.

- The methods and doPost and doGet are collected into a class named LibraryServlet.
- This class has the structure shown in Figure below.

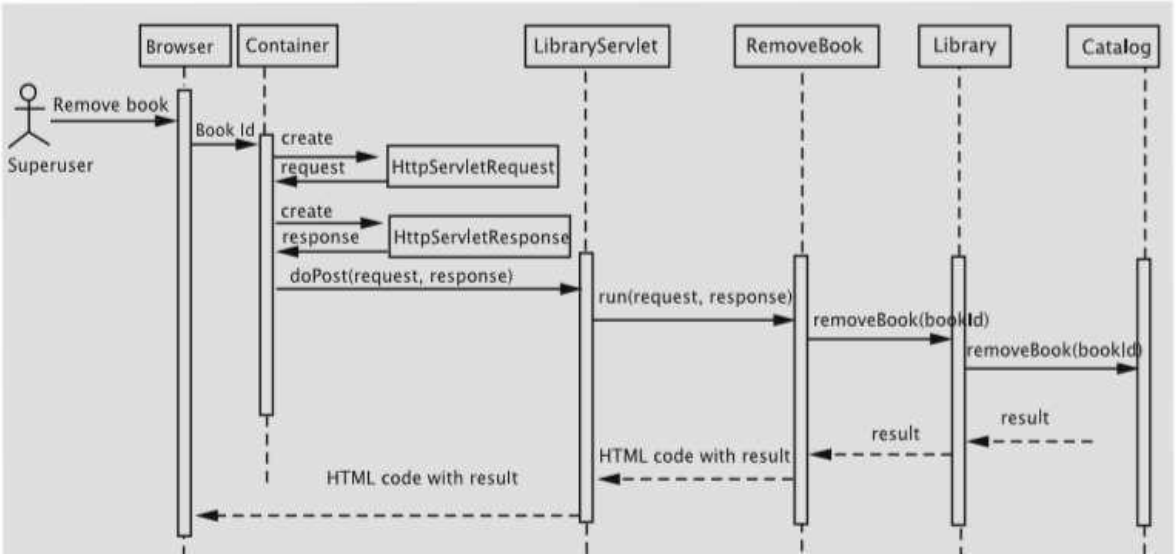


Most of the methods of LibraryServlet fall into one of five categories:

1. One group contains methods that store information about the user. This information includes the user id, the type of terminal from which the user has logged in, etc. and are stored in attributes associated with the session object. The methods are addAttribute, setAttribute, getAttribute, and deleteAllAttributes.
2. Methods to validate users and help assess access rights. The validateSuper Usermethod checks whether the user is a superuser and validateOrdinary Member does the same job for ordinary members. The method library Invocation returns true if and only if the user has logged in from a ter- minal located within the library.
3. The getFilemethod reads an HTML file and returns its contents as a String object.
4. The fourth group of methods are used for handling users who may have invoked a command without actually logging in. The method notLoggedIn returns true if and only if the user has not currently logged in. The method noLoginError Message returns HTML code that displays an error message when a person who has not logged in attempts to execute a command.
5. The final group of commands deal with processing the request and responding to it. The doGet message calls doPost, which does some minimal processing needed for all commands and then calls the abstract runmethod, which individual servlets override.

Execution flow

- Processing a request sometimes involves simply generating an HTML page,
- The course of the execution of the command is shown in Figurebelow:



The URL associated with the text is as below:

Remove book

The URL for the servlet is removebookinitialization; recall that this corresponds to the class RemoveBookInitialization, so when the link is clicked, the doPost method of that servlet is invoked. The code for this method is in LibraryServlet and is as follows:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
{
    response.setContentType("text/html");  String  page  =run(request, response); if (!notLoggedIn(requ
     {
      setAttribute(request, "page", page);
    }
    response.getWriter().println(page);
}
```

- The first line in the method specifies the type of the file for the response object: whatever is written to the response object is treated as HTML.

- The run method is invoked, which is implemented within the subclass. This method returns HTML code as a String object and is saved in the attribute named page of the session. This helps in the following way.
  - The system always remembers the last page displayed. If the user tries to login from a different window of the browser, that page is redisplayed.
  - It also helps when the user overwrites the current page by visiting some other site and wants to come back to the library system.
  - Finally, the page is written out and gets displayed in the browser.

The code for removing a book begins with the servlet RemoveBook Initialization, whose run method is given below.

```
package library;
import javax.servlet.*;
import javax.servlet.http.*;
public class RemoveBookInitialization extends LibraryServlet {
  public String run(HttpServletRequest request,
                    HttpServletResponse response) {
    if (notLoggedIn(request)) {
      return noLoginErrorMessage();
    }
    String htmlFile = getFile(HEADER);
    htmlFile = htmlFile.replace("TITLE", "Remove Book");


    htmlFile += getFile(GET_BOOK_ID);
    htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "removebook");
    htmlFile += getFile(CANCEL);
    htmlFile += getFile(END_PAGE);
    return htmlFile;
  }
}
```

- The first three lines in the runmethod check if the user has actually logged in and is not here via some othermeans.
- This can actually occur if the user has two windows connected to the library and the exit command is issued from one of the two.
- If that is indeed the case, the method noLoginErrorMessage() is called. This method simply generates an HTML page that displays ‘Not logged in’ and supplies a link to the log in screen.
- In the case that the user is actually logged in, the HTML page is assembled. It includes reading four files: one to begin the HTML page and the other to end it.
- In between, a form to enter the book id and a link to cancel the command are inserted.

- As a consequence, the browser at the client displays a page that either requires the user to enter the id of a book that should be removed or click on a link to cancel the command and return to the main menu.

The HTML code for entering the book id is given below.

```
<form action="GOTO_WITH_BOOKID" method="post">
<table>
<tr>
<td align="right">Id:</td>
                <td><input type="text" name="bookId"></td>
</tr>
<td><br><input type="submit" value="Enter Book Id"></td>
</tr>
</table>
</form>
```

In the normal course of action, the user would enter a book id and click the button labelled Enter Book Id. Notice the lines

<form action="GOTO_WITH_BOOKID"

method="post"> in the HTML file and the line

htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "removebook");

in the servlet. The place holder GOTO_WITH_BOOKID is replaced by the URL removebook. Therefore, when the user submits the form, the RemoveBook servlet is initiated.

Issuing books

An ordinary member may self-issue a book or may ask a library staff member, a superuser, to issue the book for himself/herself. In the former case, we need to skip asking the member's id and in the latter case, the system must present a screen for entering the member id.

Like all other commands, the user clicks on a link to issue books; the HTML file contains the lines

<td valign="top" width="160">

Issue book

</td>

- The click on Issue bookcauses the servlet IssueBookInitializationto execute.
- This servlet checks if the user is a superuser, and if so, a screen to accept the member id is displayed; otherwise, the member to whom the book should be issued is known and a screen to accept a book id is displayed.

The code is given below.

```
public class IssueBookInitialization extends LibraryServlet {
  public String run(HttpServletRequest request, HttpServletResponse response) {
    if (notLoggedIn(request)) {
      return noLoginErrorMessage();
    }
    boolean privileged = getAttribute(request, "userType").equals("Privileged");
    String memberId = getAttribute(request, "currentUserId");
    String htmlFile = getFile(HEADER);
    htmlFile = htmlFile.replace("TITLE", "Issue Book");
    if (privileged) {
      htmlFile += getFile(GET_MEMBER_ID);
      htmlFile = htmlFile.replace("GOTO_WITH_MEMBERID", "issuebookgetmemberid");
    } else {
      htmlFile += getFile(GET_BOOK_ID);
      htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "issuebookgetbookid");
    }
    htmlFile += getFile(CANCEL);
    htmlFile += getFile(END_PAGE);
    return htmlFile;
  }
}
```

We now discuss how we remember the member for whom the book is to be issued.

- The session object can store attributes and that commands such as issuing a book and placing a hold are always carried out for a specific member.
- That member's id is stored in the attribute currentUserId.
- If the session was for an ordinary member, the value for this attribute is the member's id itself. Otherwise, when a superuser is logged in, the value changes depending on the member for whom the command is being carried out; when the command does not involve a member , the value of this attribute is the empty string ("").

From the above discussion, clearly,

String memberId = getAttribute(request, "currentUserId");

would be the empty string if the user is a superuser and the logged-in-member's id

otherwise.

The servlet IssueBookGetMemberId retrieves the id of the member to whom books should be issued:

String memberId = request.getParameter("memberId");

If the member id is invalid, the HTML file consists of an error message and a form to accept the member id. In this case, note that control will come back to the same servlet.

```
if (!library.searchMembership(memberId)) {
  htmlFile +=  getFile(COMMAND_COMPLETED);
  htmlFile = htmlFile.replace("RESULT", "Could not locate member");
  htmlFile += getFile(GET_MEMBER_ID);
  htmlFile = htmlFile.replace("GOTO_WITH_MEMBERID", "issuebookgetmemberid");
  htmlFile = htmlFile.replace("REPLACE_JS", "");
  htmlFile = htmlFile.replace("REPLACE_COMMAND", "");
}
```

- The IssueBookGetBookIdservlet gets the book id from the form, retrieves the value of the attribute currentUserId to get the member id and calls the issueBook method of Library.
- The result is then used to replace the string RESULT in the commandCompleted HTML file.

```
String bookId = request.getParameter("bookId");
String memberId = getAttribute(request, "currentUserId");
Book book;
String result;
if ((book = library.issueBook(memberId, bookId)) == null) {
  result = "Book could not be issued";
} else {
  result = book.getTitle() + "issued.";
}
htmlFile = htmlFile.replace("RESULT", result);
htmlFile += getFile(GET_BOOK_ID);
htmlFile = htmlFile.replace("GOTO_WITH_BOOKID", "issuebookgetbookid");
```

The result of invoking issueBookis stored in the string RESULT as has been the case for other commands.

Renewing books

- The member id needs to be accepted if the user is a superuser; otherwise, that step can be bypassed.
- To allow renewal, the title and due date all of the books checked out to the user must be

displayed.

- Also, for each book a checkbox needs to be shown, so the user can check it if he/she wants the book to berenewed.

The HTML code, stored in the file renewBook.html, for this part of the process is given below.

```
<tr>
 <td> TITLE </td>
 <td> DUE_DATE </font> </td>
 <td> <input type="checkbox" name="RENEW" /> </td>
</tr>
```

- The type checkboxdenotes a checkbox control, which the user can click to indicate that a book should be renewed.
- The three strings, TITLE, DUE_DATE, and RENEW are placeholders for the book title, book id, and the name of the checkbox control. T
- The idea is that the above five lines of code will be replicated as many times as the number of books checkedout.
- The list of books must be assembled from two servlets: RenewBooks Initialization if the user is an ordinary member and RenewBooksGet MemberId if the user is a superuser.
- Since the code to perform this task is a bit lengthy, it is extracted into LibraryServlet.

The code, given below,

```
protected String assembleBooks(HttpServletRequest request, String memberId) {
  int counter = 0;
  String htmlFile = "";
  for (Iterator issuedBooks = library.getBooks(memberId);
              issuedBooks.hasNext(); counter++) {
```

```
        Book book = (Book) (issuedBooks.next());
        htmlFile += getFile(RENEW_BOOKS);
        htmlFile = htmlFile.replace("TITLE", book.getTitle());
        htmlFile = htmlFile.replace("DUE_DATE", book.getDueDate());
        setAttribute(request, "bookId" + counter, book.getId());
        setAttribute(request, "title" + counter, book.getTitle());
        setAttribute(request, "dueDate" + counter, book.getDueDate());
        htmlFile = htmlFile.replace("RENEW", "renew" + counter);
    }
    setAttribute(request, "numberOfBooks", counter + "");
    return htmlFile;
}
```

- First gets an iterator for the books checked out.
- The HTML file is built up from the file renewBook.html
- The strings TITLEand DUE_DATEare respectively replaced by the book's title and due date.
- A unique name for the checkbox is generated by replacing the string RENEWby the concatenation of renewand a counter that is incremented once per loop iteration.
- The RenewBooksservlet must somehow discover the book id and other details of the books that are to be renewed.
- Also, we list the title and due date (possibly changed) of each book to be renewed and a status message that says whether the book was renewed or not.
- This demands that we remember the details of all the books in the order we displayed them.
- These are stored in the attributes bookId, title, and dueDate, each concatenated with the value of the counter.
- Also, the number of books displayed is also stored in the attribute numberOfBooks.

Logging in and logging out

- When the class LibraryServlet is loaded, it reads the files PrivilegedUsers and IPAddressesand copies the information to main memory.
- When a user logs in, we have seen that control goes to the Login servlet.
- It assembles the log in screen for display by the browser.
- Assume now that the user types in a user id and password and sends them to the server.
- The Indexservlet reads in the user id and password and calls a method named getMenu in the class MenuBuilder.
- This class is responsible for checking the validity of the user and returning the appropriate menu. The class MenuBuilder itself is not a servlet, so to utilise the methods of LibraryServlet, it needs the reference to the Indexservlet.
- To call some of these methods, MenuBuilder also needs the request object. For

uniformity, we also pass the response object, although it is not currently used.

- The method thus has 5 parameters: a reference to the Index servlet, the request and response objects, and the user-id andpassword.

```
if (servlet.validateSuperUser(userId, password)) {
  servlet.setAttribute(request, "userType", "Privileged");
  validated = true;
} else if (servlet.validateOrdinaryMember(userId, password)) {
  servlet.setAttribute(request, "userType", "Ordinary");
  privileged = false;
  validated = true;
}
if (!validated) {
  return null;
}
```

- First, the code checks if the user is a superuser by calling the method validate SuperUser of LibraryServlet, and if so, the attribute userType is given the value Privileged.
- Otherwise, the LibraryServletclass's validate OrdinaryMembermethod is called to see if the user is a member of the library; in that case, the userType attribute is set as Ordinary.
- Also, note the use of the boolean variables privilegedand validated.
- In the event of an invalid user- id–password combination, a null value is returned to the Index servlet, which redisplays the log-in screen with an error message.

```
if (servlet.libraryInvocation(request)) {
  servlet.setAttribute(request, "location", "Library");
  location = LIBRARY;
} else {
  servlet.setAttribute(request, "location", "Outside");
}
servlet.setAttribute(request, "userId", userId);
if (!privileged) {
```

- With a successful log-in, the method checks whether the terminal used is within the library premises or outside.
- The attribute locationreflects this assessment.
- The currentUserIdis set to the user's id for ordinary users and to the empty string ("") for privileged users.

```
private String getMenu(LibraryServlet servlet, boolean privileged,
                                                boolean location) {
  boolean OUTSIDE = false;
  boolean LIBRARY = true;
  String html = servlet.getFile(LibraryServlet.HEADER);
  if (location == LIBRARY) {
    html += servlet.getFile(LibraryServlet.LIBRARY_COMMANDS);
  }
  if (privileged && location == LIBRARY) {
    html += servlet.getFile(LibraryServlet.PRIVILEGED_COMMANDS);
  }
  if (!privileged || location == LIBRARY) {
    html += servlet.getFile(LibraryServlet.GLOBAL_COMMANDS);
  }
  html += servlet.getFile(LibraryServlet.EXIT_COMMAND);
  html += servlet.getFile(LibraryServlet.END_PAGE);
  return html;
}
```

- The final step is to return the appropriate menu.
- This is done bythe method getMenu that has three parameters.
- The code assembles the HTML page by reading from four different files in addition to the files for beginning and ending the page.
- These meet the requirements we set forth under ‘Developing User Requirements’.
- If the user has logged in from the library, the Issue Book command is inserted into the menu.
- For privileged users, commands such as Add and Remove Book are inserted.
- Ordinary members always get to issue commands such as placing a hold and removing a hold.
- These commands are also available to superusers who log in from a library terminal.
- Finally, the exit command is available to all users from anywhere.