SOFTWARE ARCHITECTURE AND DESIGN PATTERNS/17IS72

Course Outcomes

CO1	Understand the basic concepts to identify state & behaviour of real world objects.
CO2	Apply Object Oriented Analysis and Design concepts to solve complex problems.
CO3	Construct various UML models using the appropriate notation for specific problem context.
CO4	Design models to Show the importance of systems analysis and design in solving complex problems using case studies.
CO5	Study of Pattern Oriented approach for real world problems.

Program Outcomes

COS/POS	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
CO1	3	-	-	-	-	-	-	-	-	-	-	-	-	-
CO2	3	-	-	-	-	-	-	-	-	-	-	-	-	-
CO3	-	3	-	-	-	-	-	-	-	-	-	-	2	-
CO4	-	-	3		-	-	-	-	-	-	-	-	3	-
CO5		-	-	3	-	-	-	-	-	-	-	-	-	-

Syllabus and Text Books

Module-1: 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. What is object-oriented development? , key concepts of object oriented design other related concepts, benefits and drawbacks of the paradigm

Module-2: 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.

Module-3: Design Pattern Catalog: Structural patterns, Adapter, bridge, composite, decorator, facade, flyweight, proxy.

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 incomplete items, adding a new feature , pattern based solutions.

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.

Text Books:

1. Object-oriented analysis, design and implementation, brahma dathan, sarnath rammath, universities press,2013
2. Design patterns, erich gamma, Richard helan, Ralph johman , john vlissides ,PEARSON Publication,2013.

Reference Books:

1. Frank Bachmann, RegineMeunier, 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

What Is a Design Pattern?

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

- ✓ Borrowed from Civil and Electrical Engineering domains.
- ✓ A technique to repeat designer success.
- ✓ A (Problem, Solution) pair

Essential Elements

A pattern has four essential elements:

- The *pattern name* that we use to describe a design problem,
- The *problem* that describes when to apply the pattern,
- The *solution* that describes the elements that make up the design, and
- The *consequences* that are the results and trade-offs of applying the pattern.

Design Patterns Are Not About Design

- Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is.
- Design patterns are not complex, domain-specific designs for an entire application or subsystem.
- One person's pattern can be another person's primitive building block.

What is and isn't a design pattern

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

What is and isn't a design pattern

- A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.
- The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.
- Each design pattern focuses on a particular object-oriented design problem or issue.
- It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

What is and isn't a design pattern

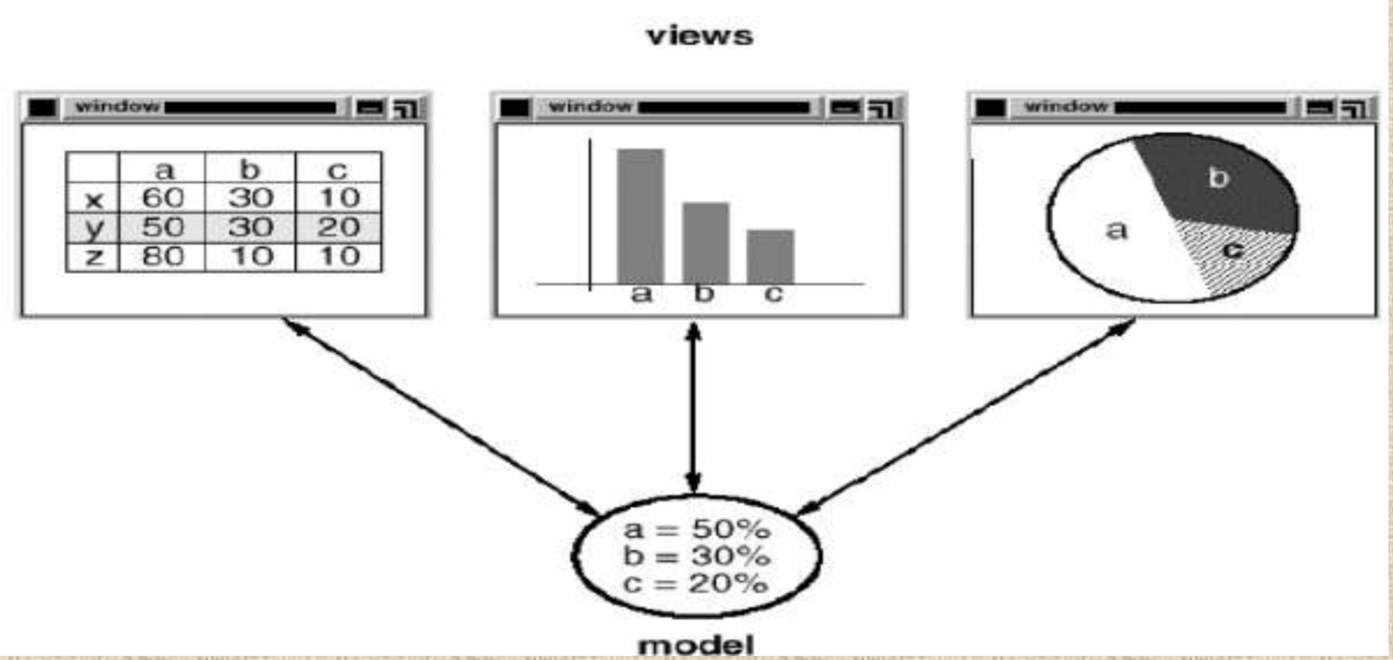
- Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages like Smalltalk and C++ rather than procedural languages(Pascal, C, Ada) or more dynamic object-oriented languages (CLOS, Dylan, Self)
- The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily.
- We might have included design patterns called "Inheritance", "Encapsulation," and "Polymorphism."

Design Patterns in Smalltalk MVC

The Model/View/Controller (MVC) triad of classes is used to build user interfaces in Smalltalk-80. MVC consists of three kinds of objects

1. **Model** is the application object,
2. **View** is its screen presentation,
3. **Controller** defines the way the user interface reacts to user input.

MVC decouples them to increase flexibility and reuse.



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

MVC decouples them to increase flexibility and reuse.

1. MVC decouples views and models by establishing a **subscribe/notify** protocol between them.
2. A view must ensure that its appearance reflects the state of the model.
3. Whenever the model's data changes, the model notifies views that depend on it.
4. In response, each view gets an opportunity to update itself.
5. This approach lets you attach multiple views to a model to provide different presentations.
6. We can also create new views for a model without rewriting it.
7. The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways.
8. The model communicates with its views when its values change, and the views communicate with the model to access these values.

Describing Design Patterns

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

- In order to reuse the design, one must record decisions, alternatives and trade-offs.
- Also need some concrete examples,
- Describe design pattern using consistent format.

A common way to describe a design pattern is the use of the following template:

1. Pattern Name and Classification
2. Intent
3. Also Known As
4. Motivation (Problem, Context)
5. Applicability (Solution)
6. Structure (a detailed specification of structural aspects)
7. Participants, Collaborations (Dynamics)
8. Implementation
9. Sample code
10. Known Uses
11. Consequences
12. Related patterns

The Catalog of Design Patterns (23 patterns)

Pattern Name	purpose
Abstract Factory (87)	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Adapter (139)	• Convert the interface of a class into another interface clients expect. • Lets classes work together with incompatible interfaces.
Bridge(1 51)	• Decouple an abstraction from its implementation so that the two can vary independently.
Builder (97)	• Separate the construction of a complex object from its representation • So that the same construction process can create different representations.
Chain of Responsibility (223)	• Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. • Chain the receiving objects and pass the request along the chain until an object handles it.

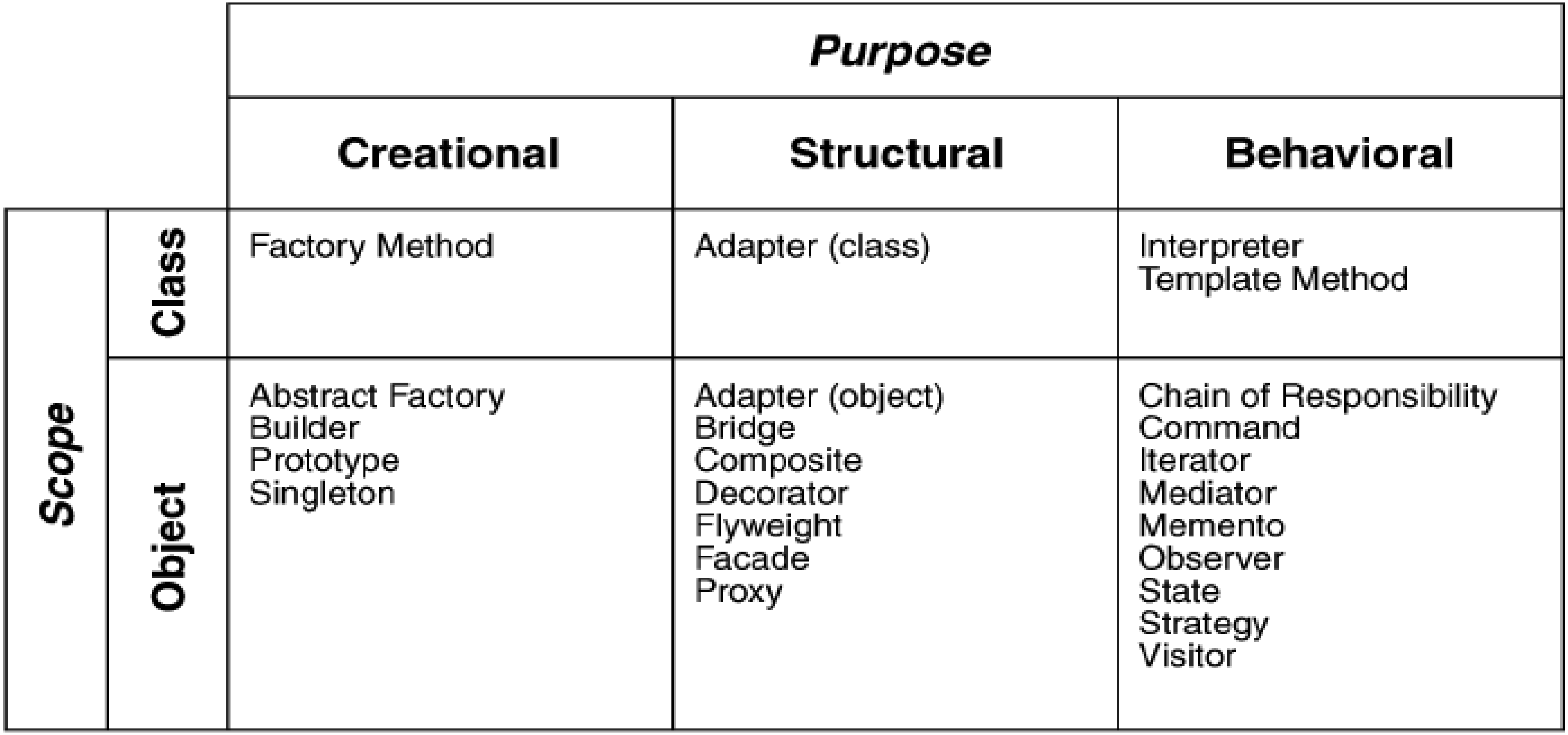
Pattern Name	purpose
Command (233)	• Encapsulate a request as an object, • Lets you parameterize clients with different requests, queue or log requests, and support undoable operations.
Composite(163)	• Compose objects into tree structures to represent part-whole hierarchies. • Lets clients treat individual objects and compositions of objects uniformly.
Decorator(175)	• Attach additional responsibilities to an object dynamically. • Provide a flexible alternative to sub classing for extending functionality.
Facade(185)	• Provide a unified interface to a set of interfaces in a subsystem. • Defines a higher-level interface that makes the subsystem easier to use.
Factory Method (107)	• Define an interface for creating an object, • But let sub classes decide which class to instantiate. • Lets a class defer instantiation to subclasses.

Pattern Name	purpose
Flyweight (195)	• Use sharing to support large numbers of fine-grained objects efficiently.
Interpreter (243)	• Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator (257)	• Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator (273)	• Define an object that encapsulates how a set of objects interact. • Promotes loose coupling by keeping objects from referring to each other explicitly, • It lets you vary their interaction independently.
Memento (283)	• Without violating encapsulation, capture and externalize an object's internal state • so that the object can be restored to this state later.
Observer (293)	• Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Prototype (117)	• **Specify the kinds of objects to create using a prototypical instance,** • **Create new objects by copying this prototype.**
Proxy (207)	• Provide a surrogate or placeholder for another object to control access to it.
Singleton (127)	• Ensure a class only has one instance, and provide a global point of access to it.
State (305)	• Allow an object to alter its behavior when its internal state changes. • The object will appear to change its class.
Strategy (315)	• Define a family of algorithms, encapsulate each one, and make them interchangeable. • Lets the algorithm vary independently from clients that use it.
Template Method (325)	• Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. • Lets subclasses redefine certain steps of an algorithm without changing the algorithm 's structure.
Visitor (331)	• Represent an operation to be performed on the elements of an object structure. • Lets you define a new operation without changing the classes of the elements on which it operates.

The Catalog of Design Patterns

Design patterns vary in their granularity and level of abstraction. All these patterns can be organized into catalog:



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

Table 1.1:Design pattern space

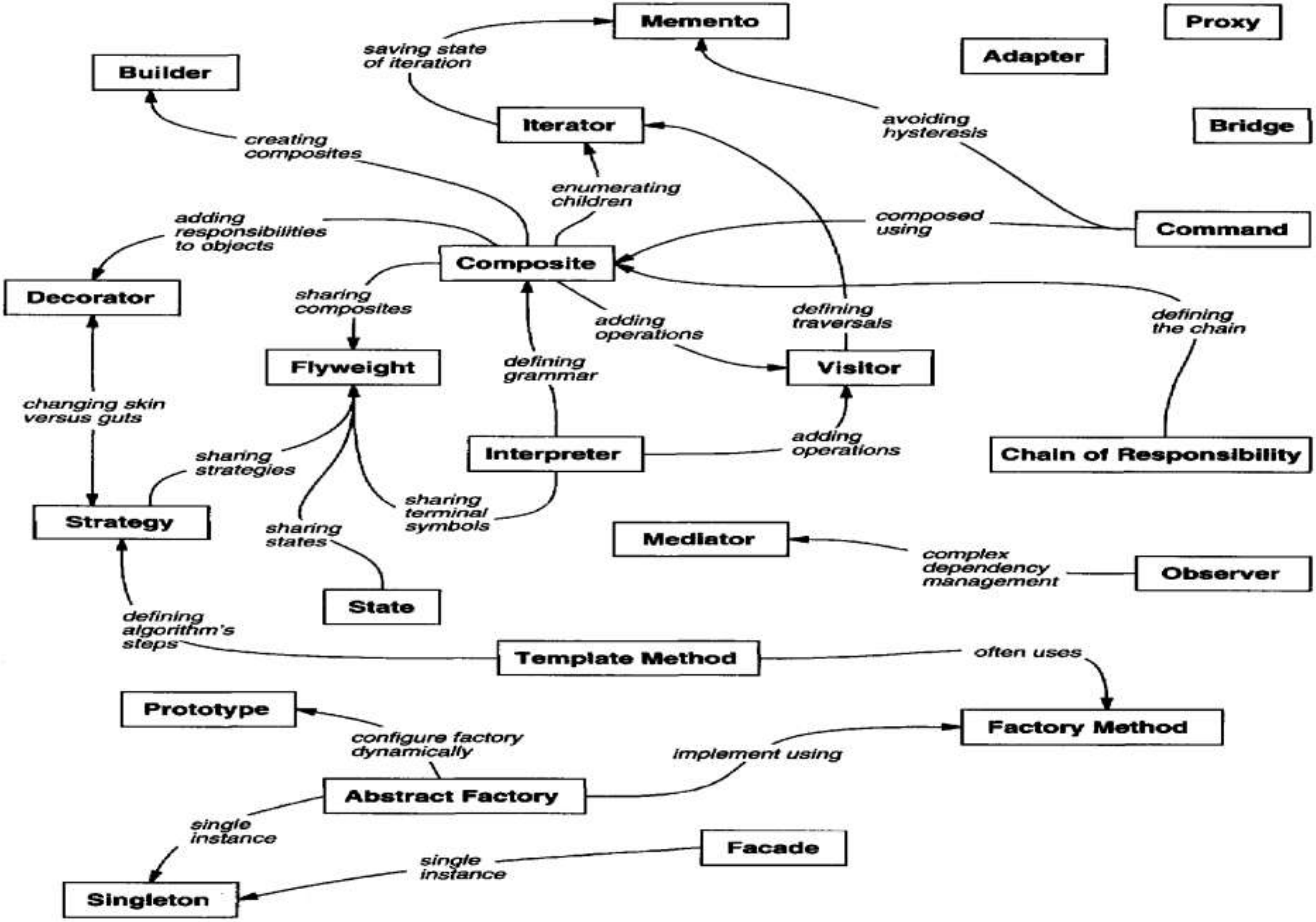


Figure 1.1: Design pattern relationships

Patterns can have creational, structural or behavioral purpose

1. ***Creational patterns*** : concerns the process of object creation
2. ***Structural patterns***: Deals with the composition of classes or objects
3. ***Behavioral patterns***: characterize the ways in which classes or objects interact and distribute responsibility.

Class patterns deal with relationships between classes and subclasses thro’ inheritance therefore they are static fixed at compile time.

Object Patterns deals with object relationships, which can be changed at run-time and are more dynamic.

So the only patterns labeled "class patterns" are those that focus on class relationships. Note that most patterns are in the Object scope.

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

How design patterns solve Design Problems

Design patterns solve many of the day-to-day problems :

1. ***Finding Appropriate Objects:***
   - Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's.
   - The abstractions that emerge during design are key to making a design flexible.
2. ***Determining Object Granularity:***
   - Objects can vary tremendously in size and number.
   - Represent everything down to the hardware or all the way up to entire applications.
   - How do we decide what should be an object?
   - Design patterns addresses these issues, Faced, Flyweight, Factory, Builder etc

How design patterns solve Design Problems

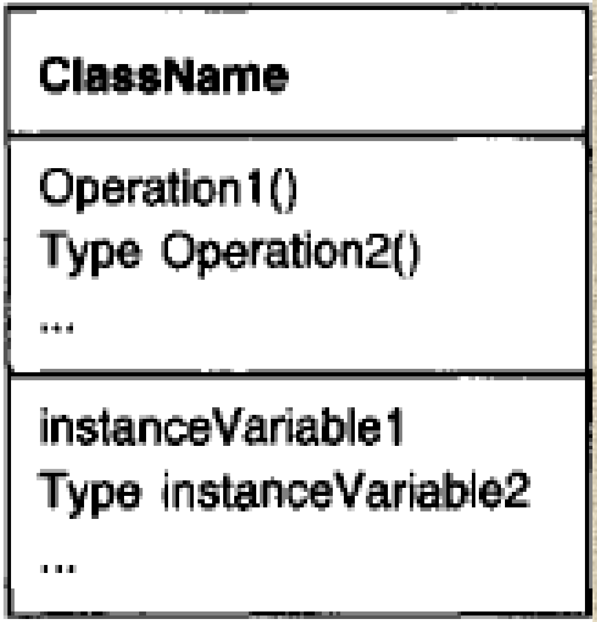
3. *Specifying Object Interfaces:*

- Every operation's takes signature (operation name, parameters, return val)
- 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**.
- **Polymorphism** (multiple operations with same name and different parameter list.
- Design patterns help you define interfaces by identifying their key

  elements and the kinds of data that get sent across an interface.

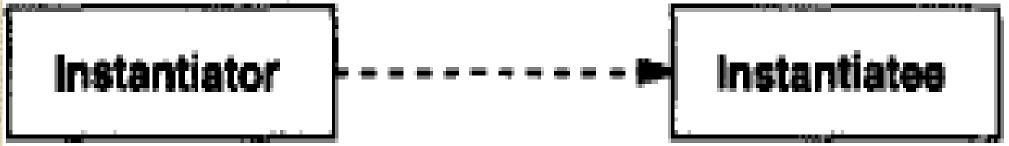
4. Specifying Object Implementations

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

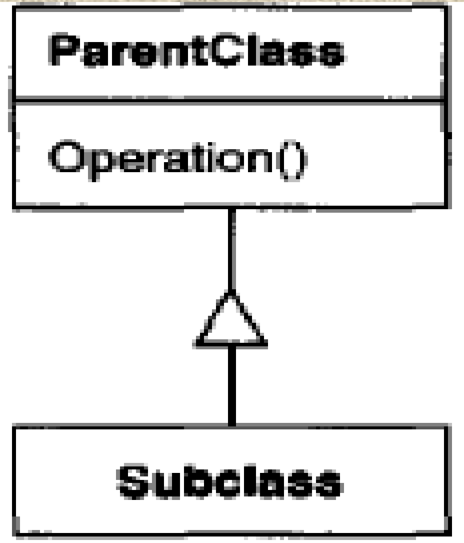


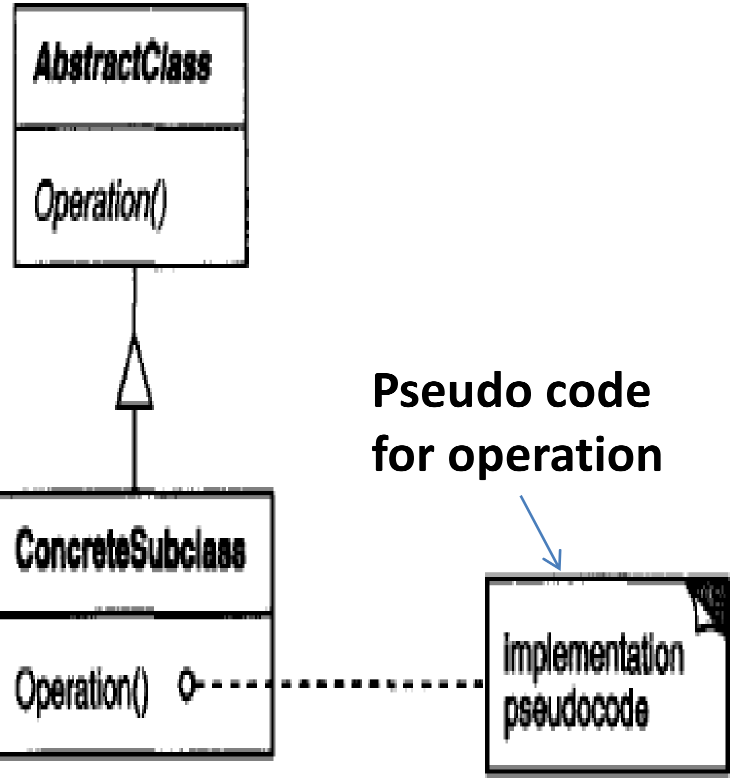
OMT-notation for class

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



4. *Specifying Object Implementations*



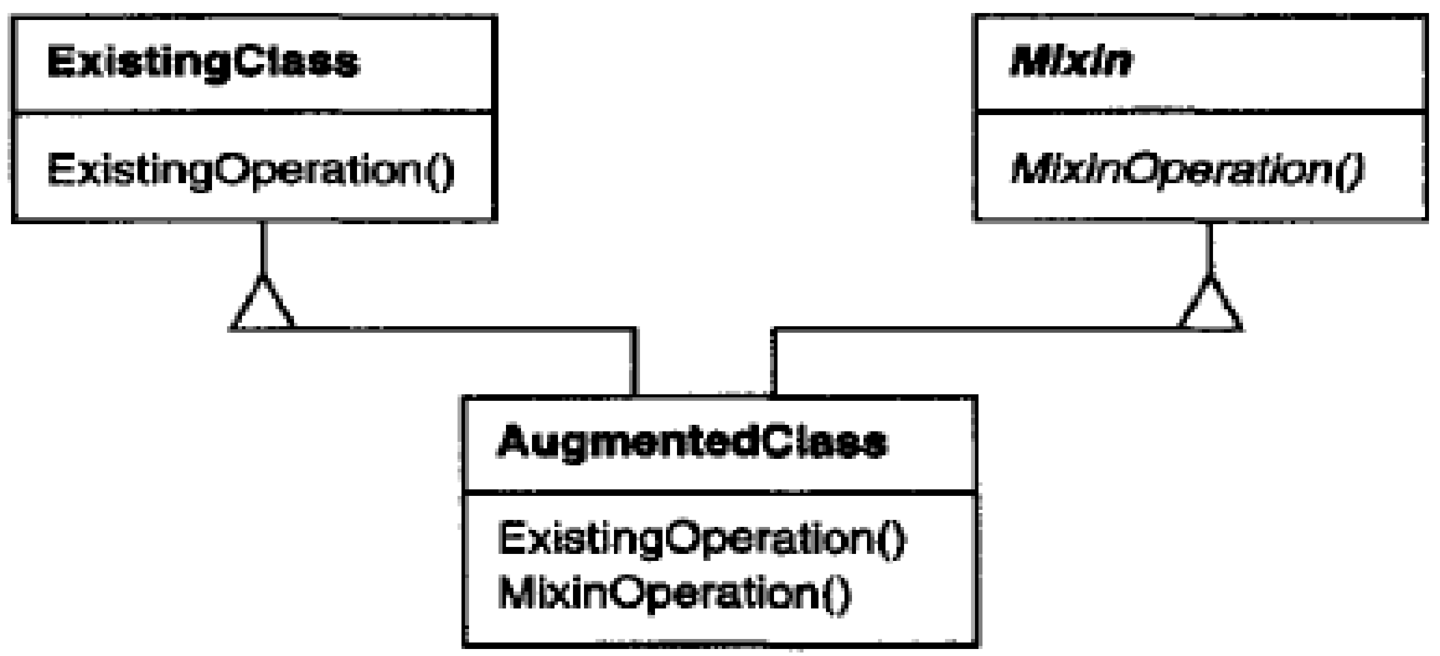


Pseudo code for operation

- New classes can be defined in terms of existing classes using class inheritance.
- An abstract cl ass is one whose main purpose is to define a common interface for its subclasses.
- An abstract class will defer some or all of its implementation to operations defined in subclasses;
- hence an abstract class cannot be instantiated.
- The operations that an abstract class declares but doesn't implement are called abstract operations.
- Classes that aren't abstract are called concrete classes.

4. *Specifying Object Implementations*

- A *mixin* class is a class that 's intended to provide an optional interface or functionality to other classes.
- It's similar to an abstract class in that it's not intended to be instantiated.
- Mixin classes require multiple inheritance:



5. Class versus Interface Inheritance:

- When we say that an object is an instance of a class, we imply that the object supports the interface defined by the class.

Difference between class inheritance and interface inheritance (or subtyping).

- *Class inheritance* defines an object's implementation in terms of another object's implementation. it's a mechanism for code and representation sharing.
- In contrast, *interface inheritance* (or subtyping) describes when an object can be used in place of another.
- The standard way to inherit an interface in C++ is to inherit publicly from a class that has (pure) virtual member functions.

6. Programming to an Interface, not an Implementation:

- ***Class inheritance*** is basically just a mechanism for extending an application's functionality by reusing functionality in parent classes.
- It lets you define a new kind of object rapidly in terms of an old one.
- It lets you get new implementations almost for free, inheriting most of what you need from existing classes.

There are two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

1. Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.

2. Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es)defining the interface.

7. Putting Reuse Mechanisms to Work:

- Concepts like objects, interfaces, classes, and inheritance are applied to build flexible, reusable software, and
- Design patterns shows effectively all these.

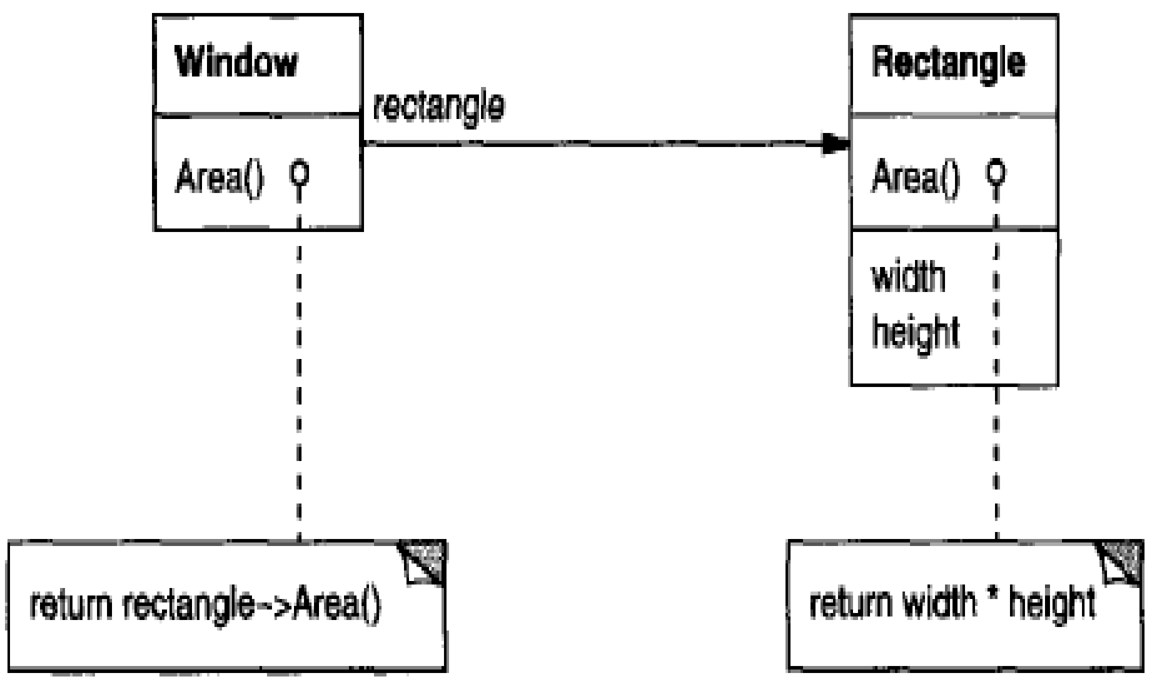
a. Inheritance versus Composition:

- The **two** most common techniques for reusing functionality in object-oriented systems are ***class inheritance*** and ***object composition.***
- Implementation of one class in terms of another's-> *inheritance.*
- Reuse by subclassing is often referred to as ***white-box reuse***.
- ***Object Composition*** is an alternative to class inheritance.
  - A new functionality is obtained by assembling or composing objects with *well-defined interfaces* to get more complex functionality called ***black-box reuse.***

Inheritance	Composition
Class inheritance is defined at compile time and straightforward to use	Object composition is defined dynamically at run-time through objects acquiring references to other objects.
Directly supported by languages	Composition requires objects to respect each others' interfaces-carefully designed interfaces
Easy to modify the implementation being reused	Objects are accessed solely through their interfaces, we don't break encapsulation.
Subclass can override the operation and make changes	Any object can be replaced at run-time by another as long as it has the same type.
you can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time.	A design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their Interrelationships.
"inheritance breaks encapsulation"	Object composition helps you keep each class encapsulated and focused on one task.
Implementation dependencies can cause problems when trying to reuse a subclass	There are substantially fewer implementation dependencies.

b. Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance. Best example of object composition.



Two objects are involved in handling a request: a receiving object delegates operations to its delegate. Analogous to subclasses deferring requests to parent classes.

The receiver passes itself to the delegate to let the delegated operation refer to the receiver. Example: Window class delegating its Area operation to a Rectangle instance Fig.

b. Delegation Advantages and Disadvantages

- It makes it easy to compose behaviors at run-time and to change the way they're composed.
- Example: Window can become circular at run-time simply by replacing its Rectangle instance with a Circle instance, assuming Rectangle and Circle have the same type.
- Delegation has a disadvantage it shares with other techniques that make software more flexible through object composition.
- Dynamic, highly parameterized software is harder to understand than more static software.
- There are also run-time inefficiencies,
- Delegation works best when it's used in highly stylized ways—that is, in standard patterns.
- Several design patterns use delegation, State(3 05),Strategy(315), and Visitor(331) patterns

c. Inheritance versus Parameterized Types

- **Parameterized types** another (not strictly object-oriented) technique for reusing functionality.
- Also known as generics (Ada, Eiffel) and templates (C++).
- The unspecified types are supplied as *parameters* at the point of use.
- Example: To declare a list of integers, you supply the type "integer" as a parameter to the List parameterized type.

Many designs can be implemented using any of these three techniques

1. An *operation implemented by subclasses* (an application of Template Method (325)),

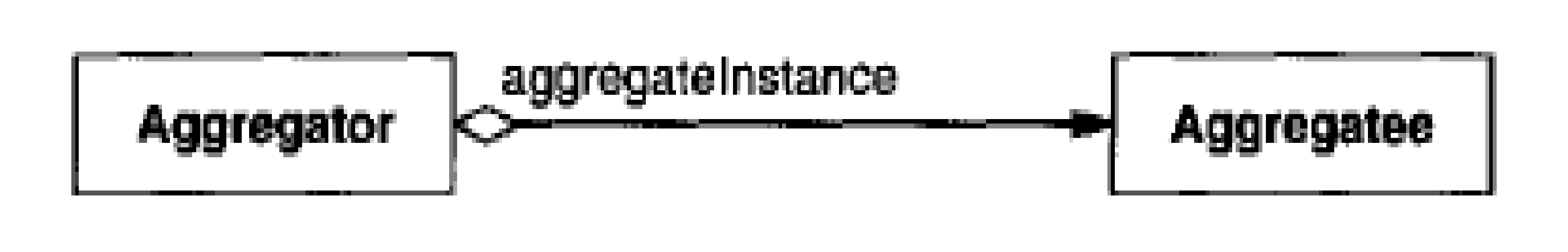
2. The *responsibility of an object* that 's passed to the sorting routine (Strategy(315)) ,

3. An *argument of a C++ template* or Ada generic that specifies the name of the function to call to compare the elements.

8. Relating Run-Time and Compile-Time Structures

- The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships.
- A program's run-time structure consists of rapidly changing networks of communicating objects.
- The two structures are largely independent.
- Object **aggregation and acquaintance** they manifest themselves at compile- and run-times.
- Aggregation implies that one object owns or is responsible for another object. i.e., an object *having* or being *part* of another object.
- Aggregation implies that an aggregate object and its owner have identical lifetimes.

- **Acquaintance** implies that an object merely *knows of* another object.
- Sometimes acquaintance is called "association" or the "using" relationship.
- Acquainted objects may request operations of each other, but they aren't responsible for each other.
- Acquaintance is a weaker relationship than aggregation and suggests much looser coupling between objects.



It's easy to confuse aggregation and acquaintance, because they are often implemented in the same way.

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.
- Redesign affects many parts of the software system, and unanticipated changes a re invariably expensive.
- Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.

some common causes of redesign along with the design pattern(s) that address them:

1. Creating an object by specifying a class explicitly.
2. Dependence on specific operations.
3. Dependence on hardware and software platform.
4. Dependence on object representations or implementations.
5. Algorithmic dependencies.
6. Tight coupling.
7. Extending functionality by subclassing.
8. Inability to alter classes conveniently.

The **role design patterns play** in the development of three broad classes of software:

1. application programs,
2. toolkit s, and
3. frameworks.

1. application programs

Design patterns makes

a. An application more maintainable when they're used to limit platform dependencies and to layer a system.

b. Enhance extensibility

c. Extending a class in isolation is easier if the class doesn't depend on lots of other classes.

2. Toolkits

- Often an application will incorporate classes from one or more libraries of predefined classes called toolkits.
- A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality.
- Toolkit design is arguably harder than application design,
- toolkits have to work in many applications to be useful.
- assumptions and dependencies that can limit the toolkit's flexibilityand consequently its applicability and effectiveness.

3. Frameworks

- A framework is a set of cooperating classes that make up 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 CAD
- The framework dictates the architecture of your application.
- It will define the overall structure, its 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.

Patterns and frameworks different in three major ways:

1. *Design patterns are more abstract than frameworks.*
2. *Design patterns are smaller architectural elements than frameworks.*
3. *Design patterns are less specialized than frameworks.*

How to Select a Design Pattern

Approaches to finding the design pattern that's right for your problem:

1. Consider how design patterns solve design problems.
2. Scan the intent sections
3. Study how the patterns interrelate
4. Study patterns of like purpose
5. Examine a cause of redesign
6. Consider what should be variable in your design

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

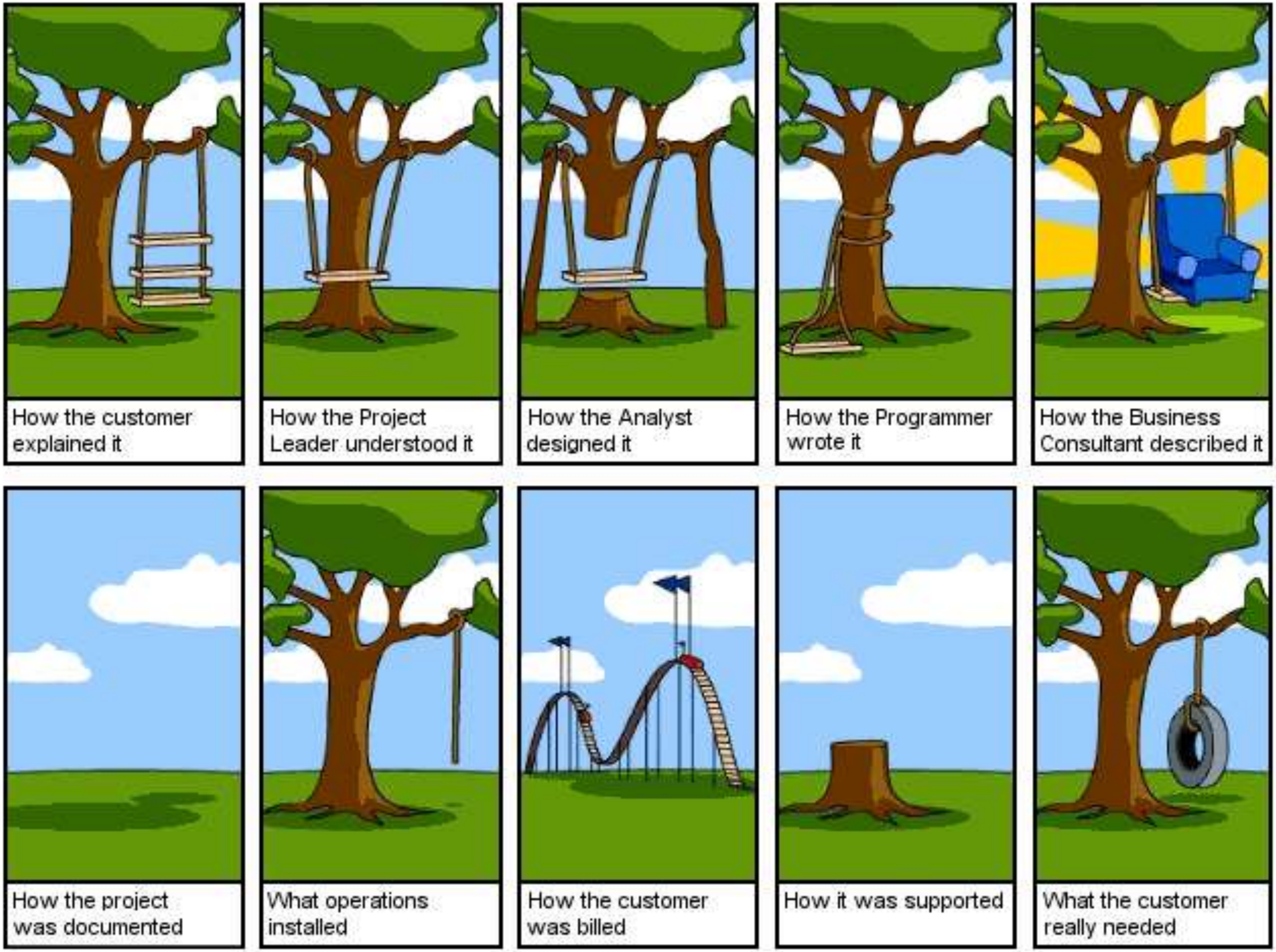
How to Use a Design Pattern

A step-by-step approach to applying a design pattern effectively:

1. *Read the pattern once through for an overview.*
2. *Go back and study the Structure, Participants, and Collaborations sections.*
3. *Look at the Sample Code section to see a concrete example of the pattern in code.*
4. *Choose names for pattern participants that are meaningful in the application context*
5. *Define the classes.*
6. *Define application-specific names for operations in the pattern.*
7. *Implement the operations to carry out the responsibilities and collaborations in t he pattern*

Basic Object Oriented Concepts

- What is Object-Oriented development?
- Key concepts of Object oriented design
- Other related concepts
- Benefits and drawbacks of the paradigm.


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Making the system design easy and understandable

- The success of many mechanical designs and systems are due to the easy way of its representation and
- Designs as a separate independent part. Can be reused again and again.
- A similar idea was implemented in software product development also where some proved off-the-shelf components are used.

Possible through:

- *An 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,*

The overall philosophy here is to define a software system as a collection of objects of various types that interact with each other through well-defined interfaces.

Key concepts of Object oriented design

1. ***The central role of objects:*** centrepiece of software design
2. ***The Notion of a Class:*** define hierarchies and engage with the ideas of specialisation and generalisation of objects.
3. ***Abstract Specification of Functionality:*** specification, called an **interface** or an **abstract class**

4. A Language to Define the System : The Unified Modelling Language (UML) as the standard tool for describing the end products of the design activities. Similar to 'blueprints'.

5. Standard Solutions: documenting of standard solutions, called **design patterns->** common form of reuse of solutions

6. An Analysis Process to Model a System: systematic way to translate a functional specification to a *conceptual design.*

7. The Notions of extendability and Adaptability: Inheritance, composition

Other Related Concepts

1. Modular Design and Encapsulation:

- ***Modularity:*** Putting together a large system by developing a number of distinct components independently and then integrating these to provide the required functionality.
- ***Encapsulation:*** Module hides details of its implementation from external agents, using ADT.

2. Cohesion and Coupling:

- ***Cohesion:***
  - **cohesion** of a module tells us how well the entities within a module work together to provide this functionality.
  - Highly cohesive modules tend to be more reliable, reusable, and understandable.
- ***Coupling:***
  - **Coupling** refers to how dependent modules are on each other.
  - high coupling means that changes in one module would necessitate changes in other modules, domino effect.

3. Modifiability and Testability

- ***Modifiability:*** Modification can be done to change both *functionality* and *design*.
- Modifiable systems are more adaptable.
- Improving the design through incremental change is accomplished by *refactoring*
- ***Testability :***
- **Testability** of a concept, in general, refers to both *falsifiability*,
- it can simply be stated as the ease with which we can find bugs in a software and the extent to which the structure of the system facilitates the detection of bugs.

Benefits and Drawbacks of the Paradigm

1. ***Objects often reflect entities in application systems***
2. ***Object-orientation helps increase productivity through reuse of existing software.***
3. ***It is easier to accommodate changes.***
4. ***The ability to isolate changes, encapsulate data, and employ modularity reduces the risks involved in system development.***

Module-2 :

Analysing a System

Contents :

1. *Overview of the analysis phase*
2. *Stage 1: Gathering the Requirements*
   - ➢ *Case Study Introduction*
3. *Functional Requirements Specification*
   - ➢ *Use Case Analysis*
4. *Defining Conceptual Classes and Relationships*
5. *Using the Knowledge of the Domain*
6. *Discussion and Further Reading*
7. *Design and Implementation*

1. Overview of the analysis phase

What should the system do?

Simple minded approach does not suffice for the real-life projects: reasons:

1. Systems are typically much bigger in scope and size.
2. Have complex and ambiguously-expressed requirements.
3. Usually a large amount of money involved, which makes matters quite serious.
4. Project deadlines for these ‘real-life’ projects are more critical

The process of building a system could be split into 3 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.

These activities could be iterative or nested

Stage 1: Gathering the Requirements

The purpose of requirements analysis is to define what the new system should do.

- System will be built based on the information garnered.
- 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.
- Requirements for a new system are determined by a team of analysts by interacting development (clients) and the user community.
- This interaction can be in the form of interviews, surveys, observations, study of existing manuals, etc.,

Requirements can be classified into 2 categories

1. ***Functional requirements :***

These describe the interaction between

- System and its users,
- System and any other systems, which may interact with the system by supplying or receiving data.

2. Non-functional requirements: Any requirement that does not fall in the above category is a non-functional requirement.

- Such as response time, usability and accuracy.
- May be considerations that place restrictions on system development; the use of specific hardware and software and budget and time constraints.

Case Study : A simple library system

Functionality is described as a list called ***business processes.*** The ***business processes*** of the ***library system*** are listed below.

1. **Register new members**
2. **Add books to the collection**
3. **Issue a book to a member (or user)**
4. **Record the return of a book**
5. **Remove books from the collection**
6. **Print out a user's transactions**
7. **Place/remove a hold on a book**
8. **Renew books issued to a member**
9. **Notify member of book's availability**

A simple library system In addition, the system must support three other requirements that are not directly related to the workings of a library:

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

Functional Requirements Specification

- The requirements be precisely documented.
- The requirements specification document serves as a contract between the users and the developers.
- No confusion as to what the expectations are.
- An accepted way of accomplishing this task is the ***use case analysis***,

Use Case Analysis:

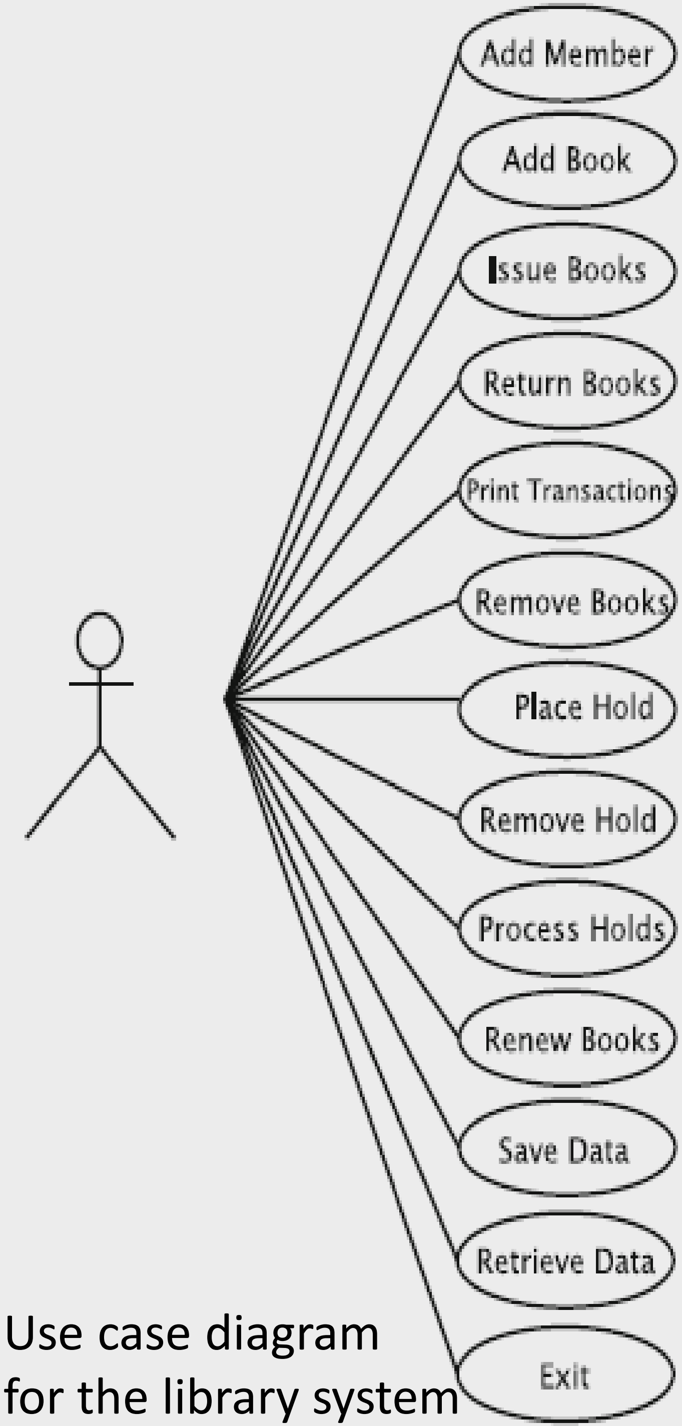
Def: Use case analysis is a *case-based way of describing* the uses of the system with the *goal of defining and documenting the system requirements.*

- 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*:
  1. ***Agents:*** Who interact with the system
  2. The **system itself**.

A simple library system…

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

A simple library system…

Use case for registering a user: specified using a 2-column format

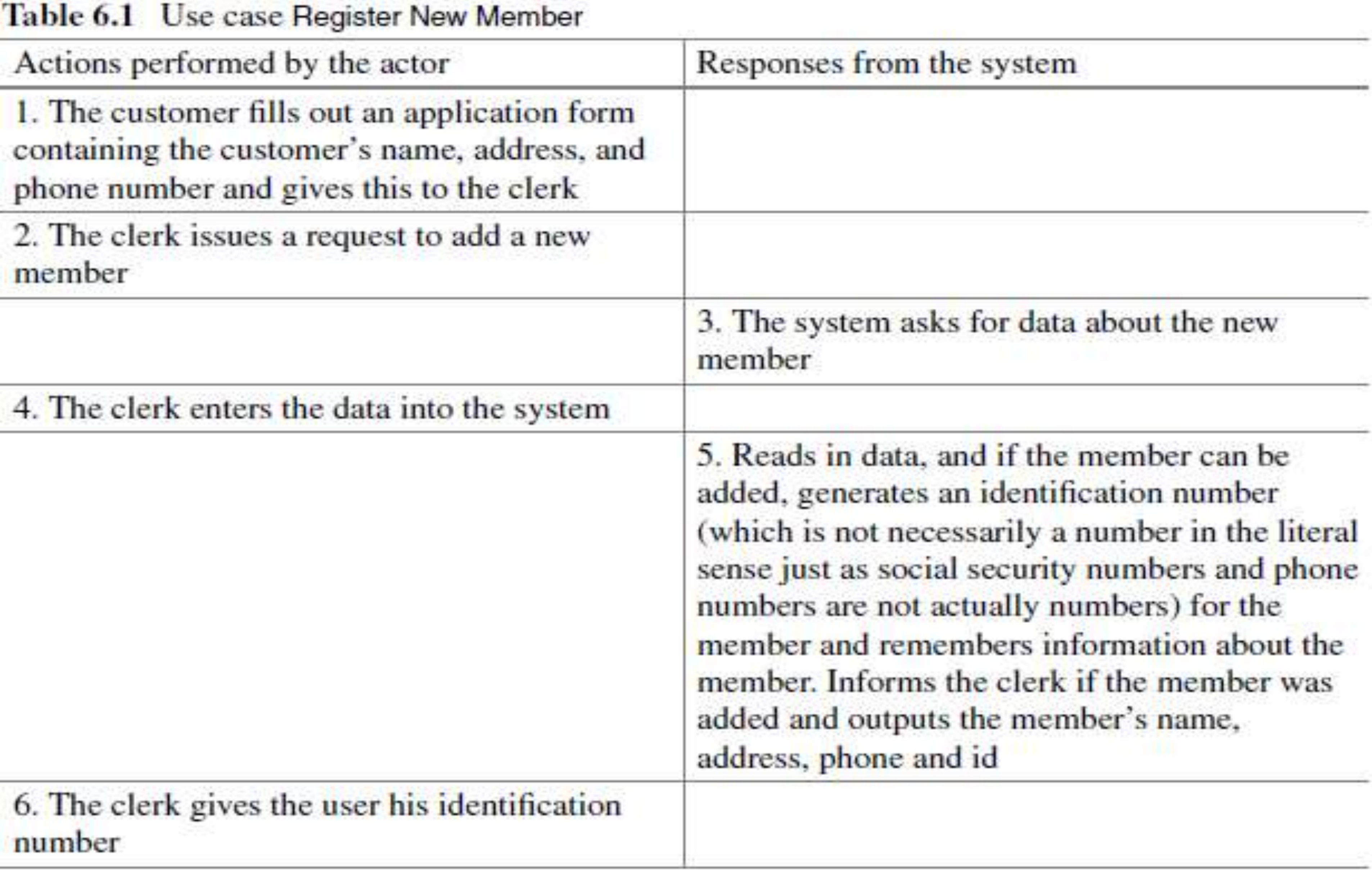


Table 6.1 Use case Register New Member

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	

A simple library system…

Illustrates several aspects of use cases.

1. Every use case has to be identified by a name.

   Ex: ***Register New Member***

2. Reasonably-sized activity in the organisation.
   - Not all actions and operations should be identified as use cases. Ex: stamping a due-date on the book should not be a use case
   - A use case is a relatively large end-to-end process description that captures some business process
   - A business process may be decomposed into more than one use case
   - when there is some intervening real-world event(s) for which the agent has to wait for an unspecified length of time.
3. The first step of the use case specifies a 'real-world' action that triggers the exchange described in the use case.

A simple library system…

4. The use case does not specify how the functionality is to be implemented. Ex: the details of how the clerk enters the required information into the system is unspecified.
5. The use case is not expected to cover all possible situations.
   - Use case does not specify what the system should do if there are errors.
   - The use case explains only the most commonly-occurring scenario- referred to as the *main flow.*

A simple library system…

Use case for adding books:

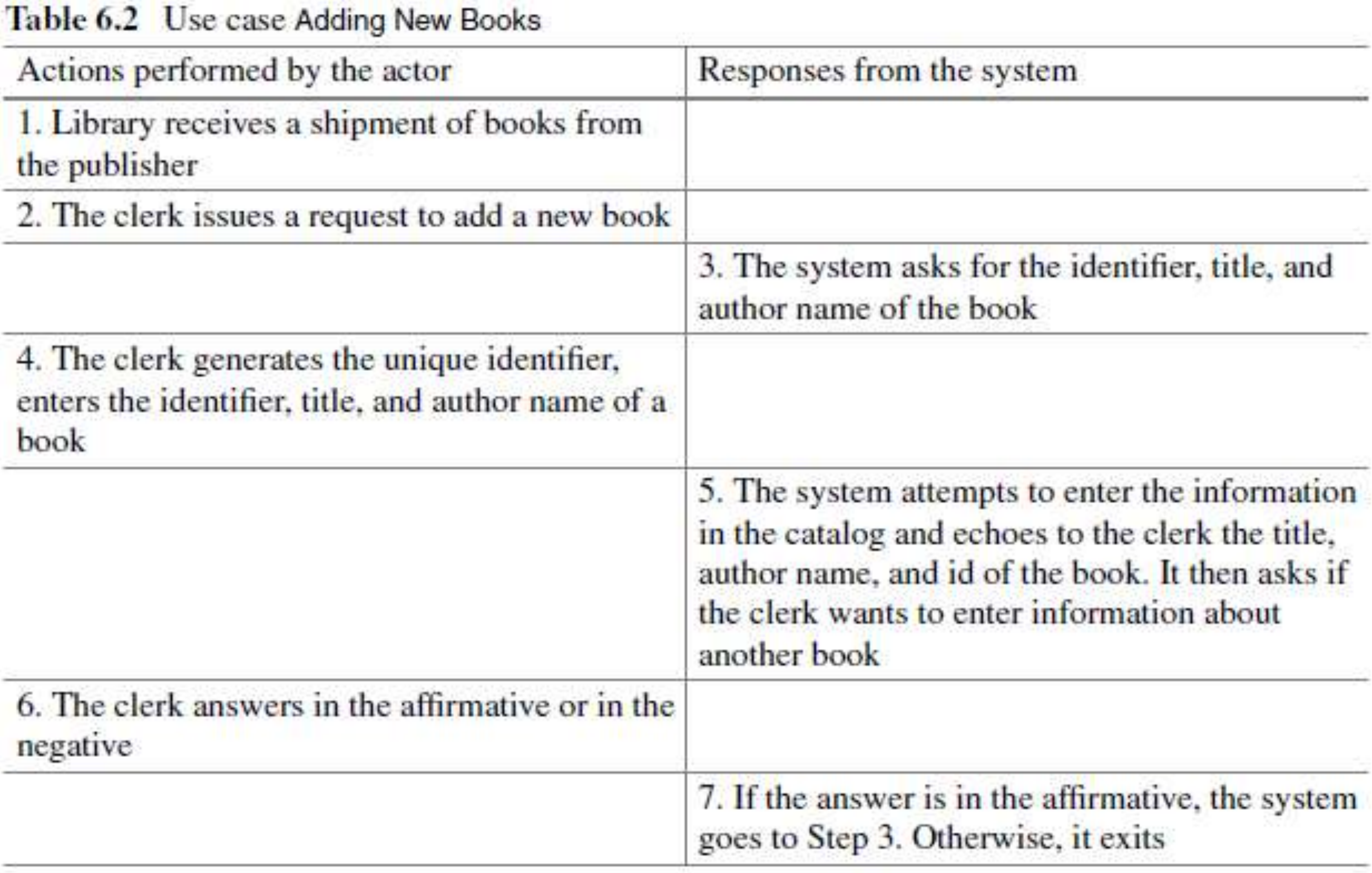


Table 6.2 Use case Adding New Books

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

A simple library system…

Use case for issuing books:

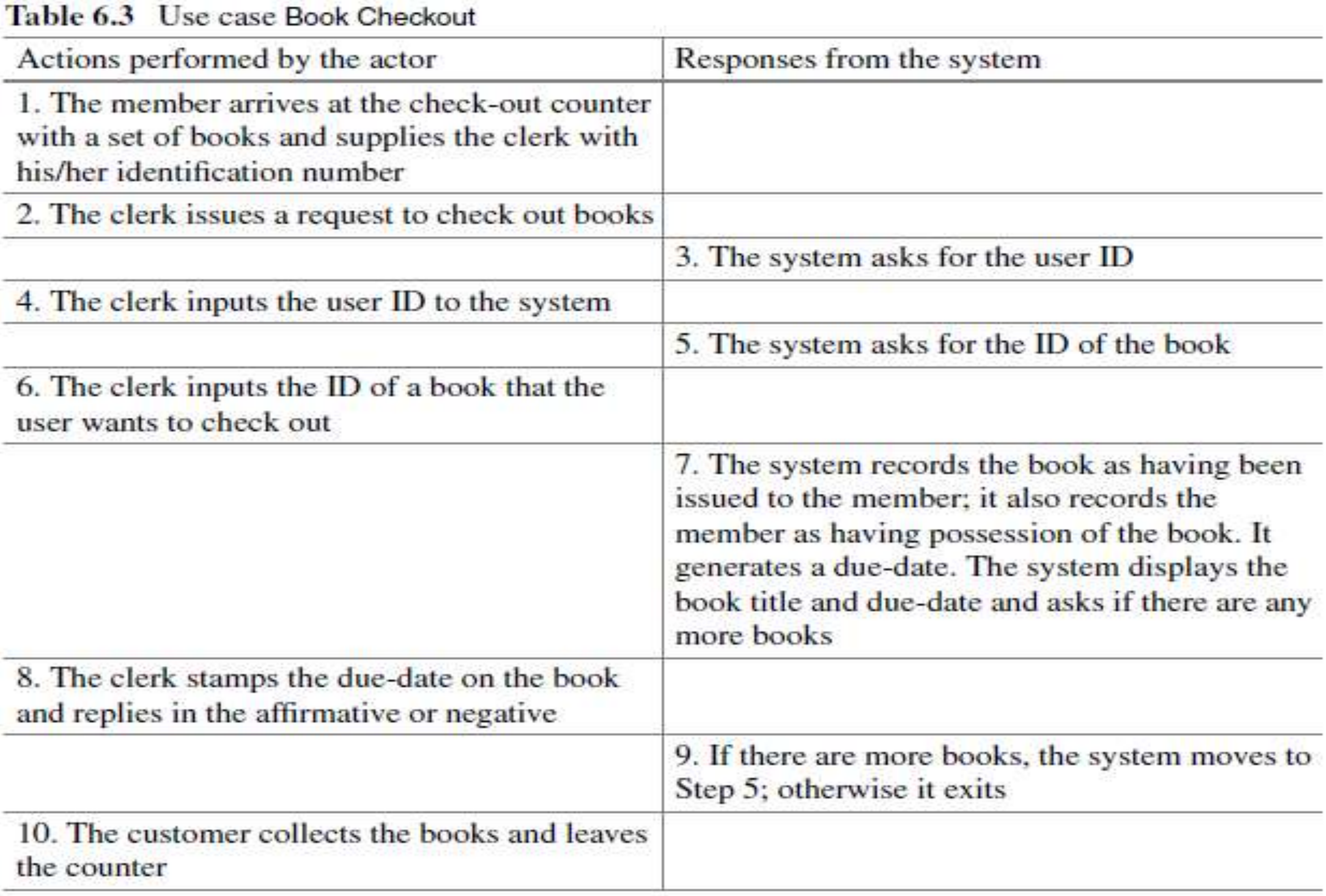


Table 6.3 Use case Book Checkout

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	

- Putting all these details in the use case would make the use case quite messy and harder to understand. **Business Rules**.
- A business rule may be applicable to one or more use cases.
- Example: The business rule for due-date generation is simple in our case.

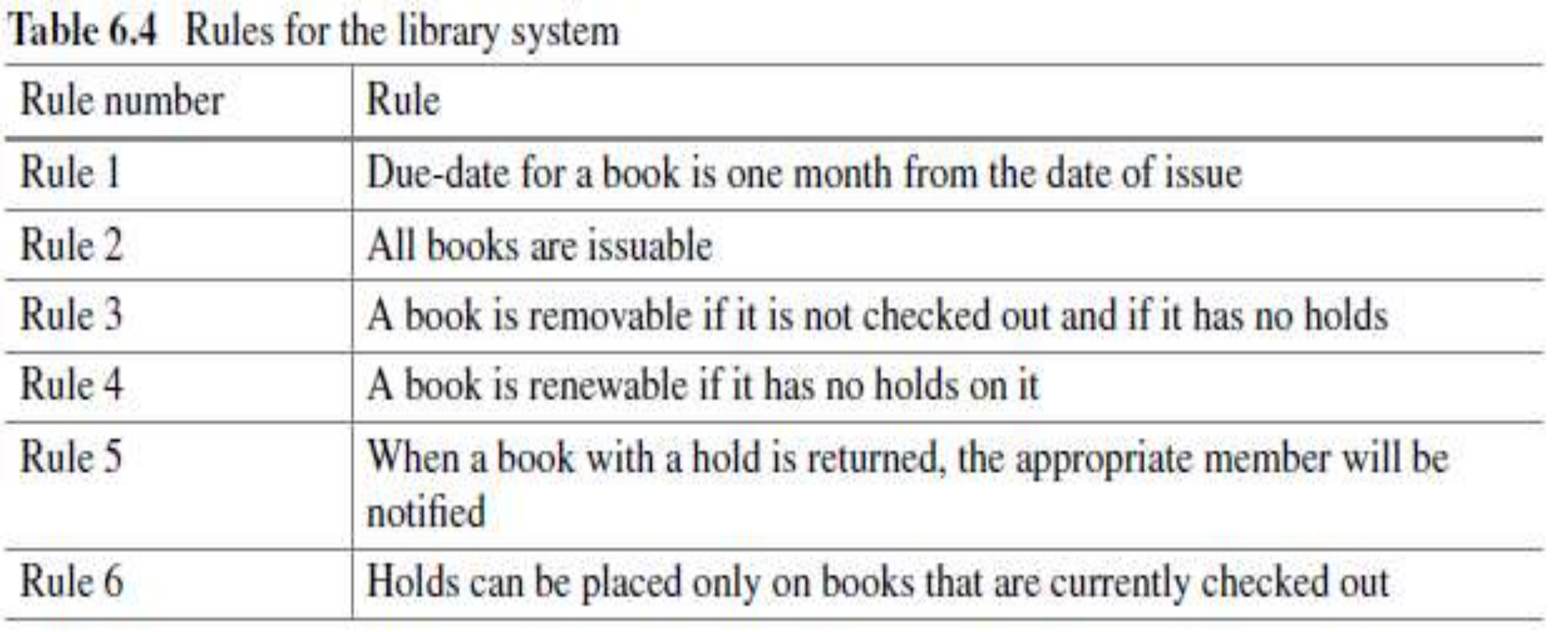


Table 6.4 Rules for the library system

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

After applying the rules in the Book Checkout use case

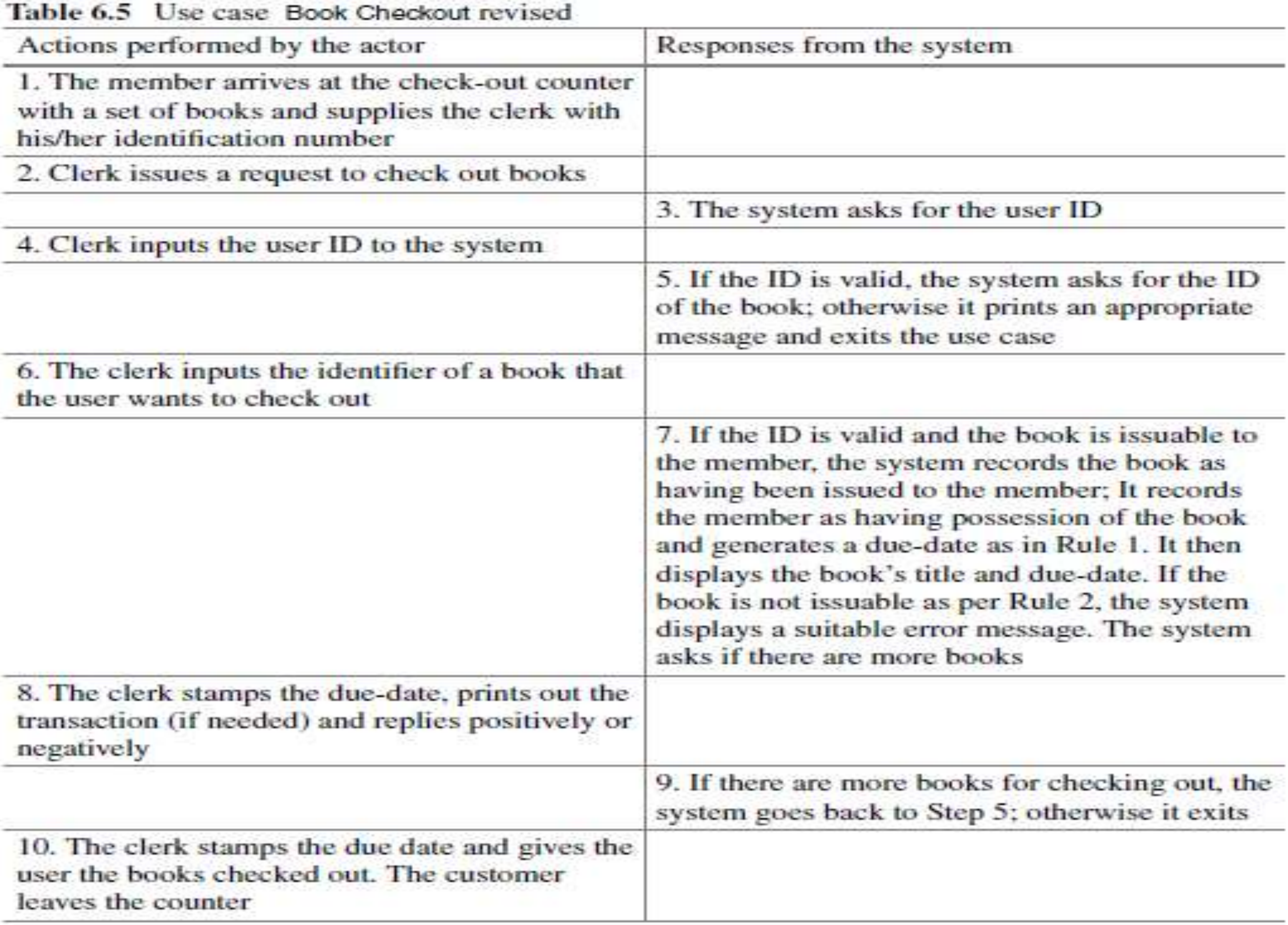


Table 6.5 Use case Book Checkout revised

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	

Similarly use case for the remaining can be written using table

1. Use case for returning books *table 6.6*
2. Use cases for removing (deleting) books, printing member transactions, placing a hold, and removing a hold *(Table 6.7, Table 6.8, Table 6.9, Table 6.10)*

How do Business Rules Relate to Use Cases?

- Business rules can be broadly defined as the details through which a business implements its strategy.
- Business analysts perform the task of gathering business rules, and these belong to one of four categories:

1. **Definitional rules**
2. **Factual rules**
3. **Constraints**
4. **Derivations**

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.* Usefulness of this step is:

1. **Design facilitation**
2. **Added knowledge**
3. **Error reduction**
4. **Useful documentation**

Guidelines to Remember When Writing Use Cases

1. A use case must provide something of value to an actor or to the business : *each use case has at least one actor.*
2. Use cases should be *functionally cohesive* : *they encapsulate a single service that the system provides.*
3. *Use cases should be temporally cohesive : constitute separate use cases*
4. *If a system has multiple actors, each actor must be involved in at least one, and typically several use cases.*
5. *The model that we construct is a set of use cases, i.e., there is no relationship between individual use cases.*
6. *Exceptional exit conditions are not handled in use cases: For instance, if a system should crash in the middle of a use case,*
7. *Use cases are written from the point of view of the actor in the active voice.*
8. *A use case describes a scenario, : i.e., tells us what the visible outcome is and does not give details of any other requirements that are being imposed on the system.*
9. *Use cases change over the course of system analysis: Use cases may be merged, added or deleted from the model at any time.*

In practice the analyst will use multiple methods to come up with the conceptual classes and their relationships

The text of use case, all nouns bold-faced:

(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** 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**.

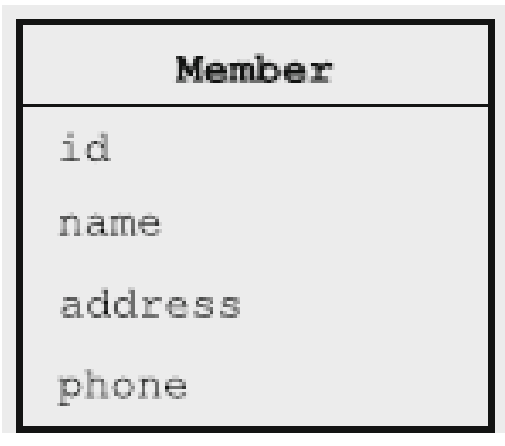
(6) The **clerk** gives the **user** his **identification number**.

First, let us eliminate duplicates to get list

customer, **application form**, **customer's name**, **address**, **phone number**, **clerk**, **request**, **system**, **data**, **identification number**, **member**, **user**, **member information**, and **member's name**. Some of the nouns such as **member** are composite entities that qualify to be classes.

1. **customer**: becomes a member, so it is effectively a synonym for member.
2. **user**: the library refers to members alternatively as users, so this is also a synonym.
3. **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.
4. **customer's name, address**, and **phone number**: They are attributes of a customer, so the Member class will have them as fields.
5. **clerk**: is just an agent for facilitating the functioning of the library, so it has no software representation.
6. **identification number**: will become part of a member.
7. **data**: gets stored as a member.
8. **information**: same as data related to a member.
9. **system**: refers to the collection of all classes and software.

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.



Member

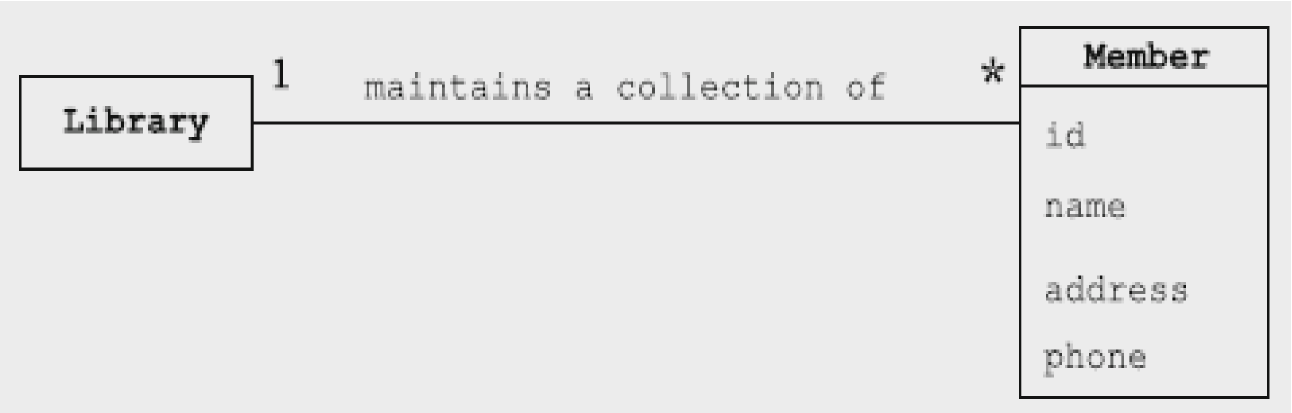
id

name

address

phone

UML diagram for the class Member



Library

1

maintains a collection of

*

Member

id

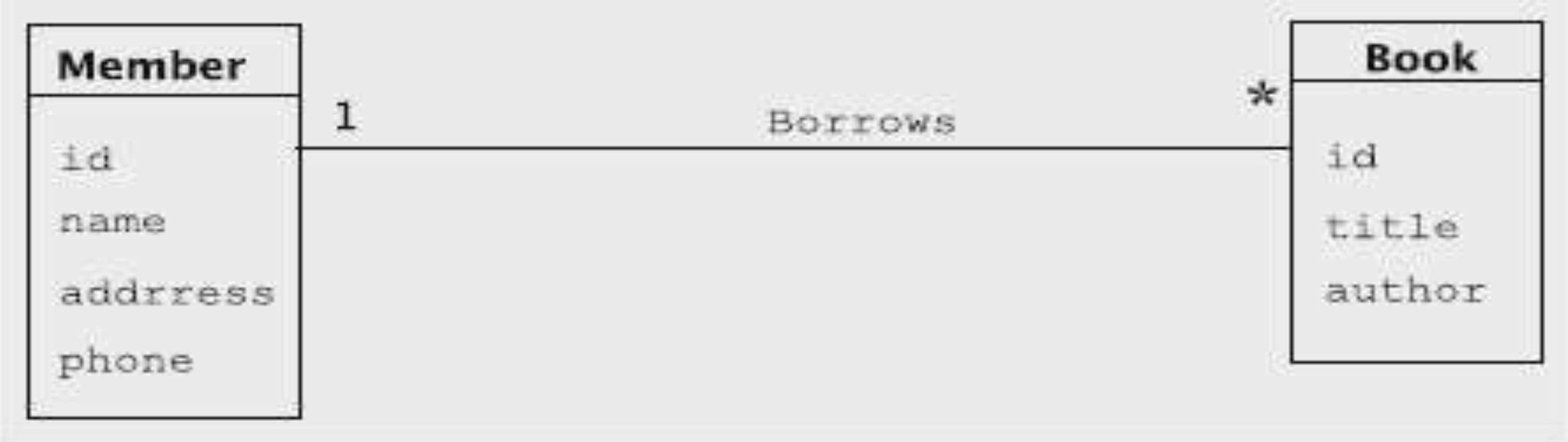
name

address

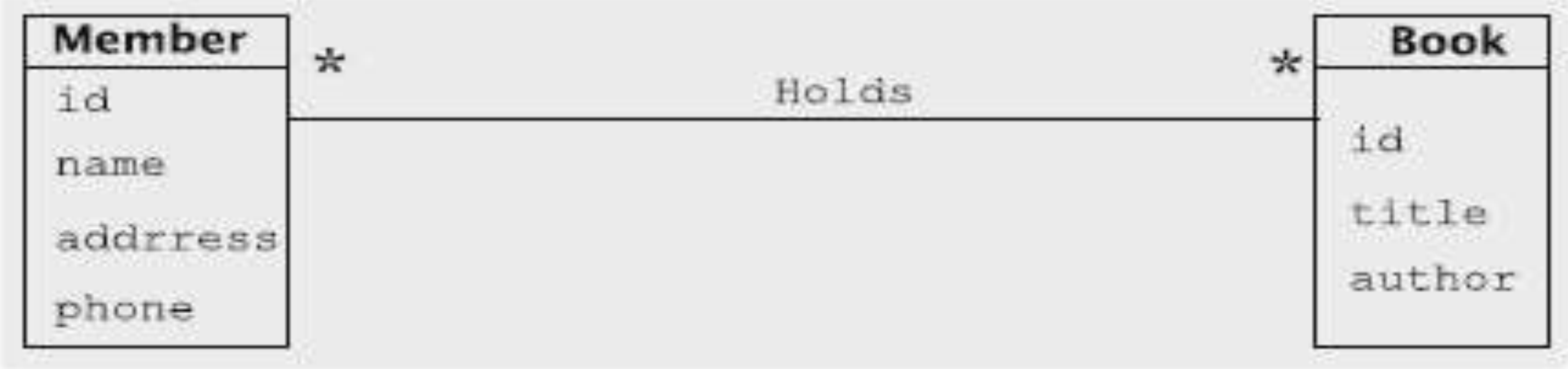
phone

UML diagram showing the association of Library and Member : one-to-many association: mean that one instance of the Library maintains a collection of zero or more members.

- In this example we have many ***conceptual classes*** like library members, system etc.,
- Some **associations are static**, i.e., permanent, whereas others are *dynamic*. ***Dynamic associations*** are those that change as a result of the transactions being recorded by the system.
- Such associations are typically ***associated with verbs***.



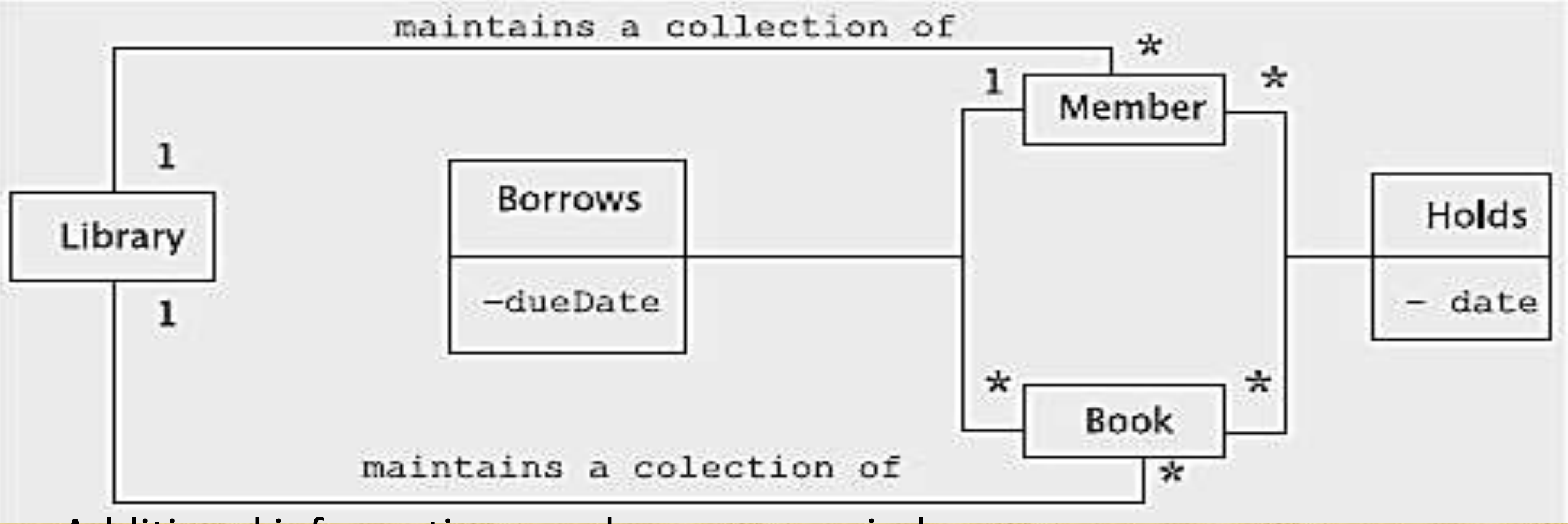
UML diagram showing the association Borrows between Member and Book



UML diagram showing the association Holds between Member and Book

- many-to-many between users and books.
- Several users can have holds placed on a book, and a user may place holds on an arbitrary number of books.

We capture all of the conceptual classes and their associations into a single diagram:



- Additional information can be accompanied.
- This is added when a user borrows a book and when a user places a hold on a book.
- Borrowing a book introduces new information into the system, viz., the date on which the book is due to be returned. Likewise, placing a hold introduces some information, viz., the date after which the book is not needed.
- conceptual classes are attached to the line representing the corresponding associations.

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 what parts are variable.

OR

- we identify and analyse common requirements from a specific application domain.
- we apply the knowledge we already have from our study of similar systems to speed up the creation of specifications, design, and code.
- ***one of the goals of this approach is reuse***.
- For example, we could say that the

domain (university applications) => domain (course management + student admissions + payroll applications, and so on)

- The interactions of the smaller domains that make up the bigger one.

In the domain of libraries includes the following.

1. The environment, including customers and users. Libraries have loanable items such as books, CDs, periodicals, etc. A library’s customers are members. Libraries buy books from publishers.
2. Terminology that is unique to the domain. For example, the Dewey decimal classification (DDC) system for books.
3. Tasks and procedures currently performed. In a library system, for example:

(a) Members may check out loanable items.

(b) Some items are available only for reference; they cannot be checked out.

(c) Members may put holds on loanable items.

(d) Members will pay a fine if they return items after the due date.

Finding the Right Classes

- In general, finding the right classes is non-trivial.
- Process is iterative,
- The following ***thumb rules*** and caveats come in handy:

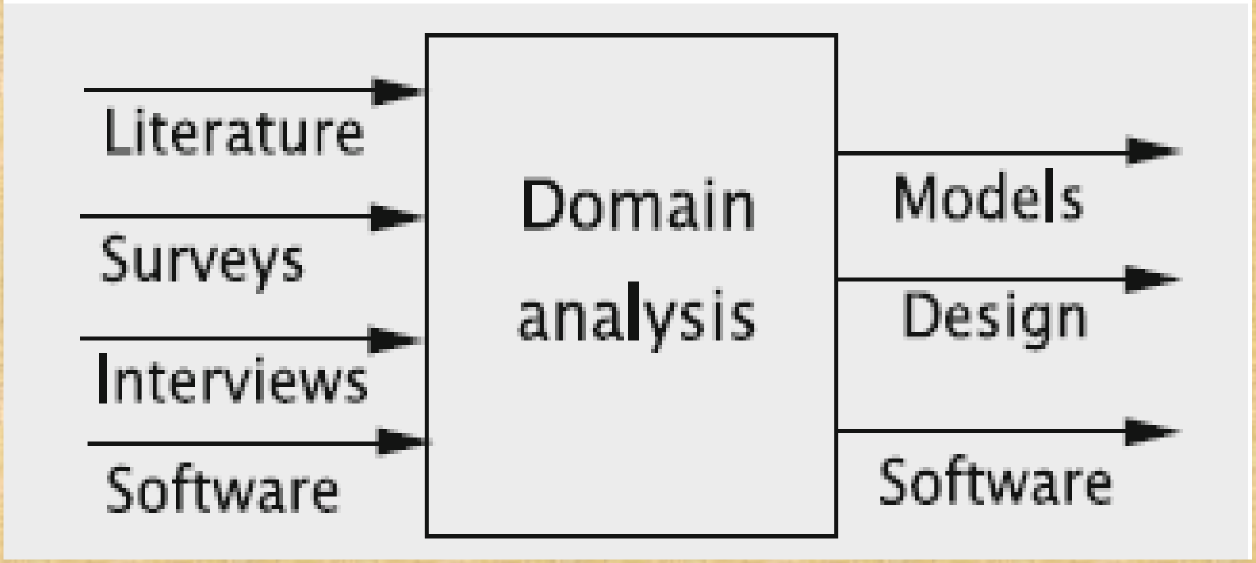
1. Do not build classes around functions. If class name is imperative, e.g., print, parse, etc., either the class is wrong or the name is wrong.
2. Remember that a class usually has more than one method; otherwise it is probably a method that should be attached to some other class.
3. Do not form an inheritance hierarchy too soon unless we have a preexisting taxonomy.

4. Be wary of classes that have no methods, Some situations in which they occur are:
   a) representing objects from outside world,
   b) encapsulating facilities, constants or shared variables,
   c) applicative classes used to describe non-modifiable objects,
5. Check for the following properties of the ideal class:
   a. a clearly associated abstraction, which should be a data abstraction (as opposed to a process abstraction),
   b. a descriptive noun/adjective for the class name,
   c. a nonempty set of runtime objects,
   d. queries and commands,
   e. abstract properties that can be described as pre/post conditions and invariants.

***One of the major activities of this analysis* is discovering the *business rules*, the rules that any properly-functioning system in that domain must conform to.**

Q. 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.



Domain analysis

Chapter 7: Design and Implementation

Contents:

1. Design
    a. Major Subsystem
    b. Creating the software classes
    c. Assigning responsibilities to the classes
    d. Class diagrams
    e. User interface
    f. Data storage
2. Implementing our design

Design:

We use the class structure produced by the analysis to design a system that behaves in the manner specified by the model.

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?
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 realising 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?

a. 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.

b. Creating 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.*
- The *software classes are more 'concrete'* in that they correspond to the *software components* that make up the 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.

- Several iterations may be needed and classes may need to be added, split, combined, or eliminated.
- The classes for the business logic module will be the ones instrumental in implementing the system requirements described in the use case model.

Conceptual classes of Library System are:

1. 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.

2. Library a library, can be viewed as borrowing and returning books, placing and removing holds, i.e., the *functionality* provided by the library.

- All the computation required of the business logic module must be executed on some current object; that object is a Library.
- So Library be class in its own.
- A class that has just *one instance is called a singleton. Both MemberList and Catalog are singletons.*

3. Borrows: *(An association class between member and books class)*

- 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.*

4. Holds: *(An association class between member and books class)*

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.*
- *Have a class for this relationship and make the Hold object accessible to the instances of Member and Book*

C. Assigning Responsibilities to the Classes:

- Having decided on an adequate set of software classes, next task is to assign responsibilities to these.
- The ultimate purpose of these classes is to enable the system to meet the responsibilities specified in the use case.
- For each system response listed in the right-hand column of the use case tables, we need to specify the following:

a. The sequence in which the operations will occur.- we need a complete algorithm

b. How each operation will be carried out.- Describes which classes will be involved in each step of the algorithm and how the classes will be engaged.

Register Member: Sequence Diagram:

This interaction occurs between the library staff member and the UserInterface instance. The clerk enters the requested data, which the UserInterface accepts.

The addMember method, algorithm here consists of three steps:

1. Create a Member object.
2. Add the Member object to the list of members.
3. Return the result of the operation.

To carry out the first two steps, we have two options:

Option1:

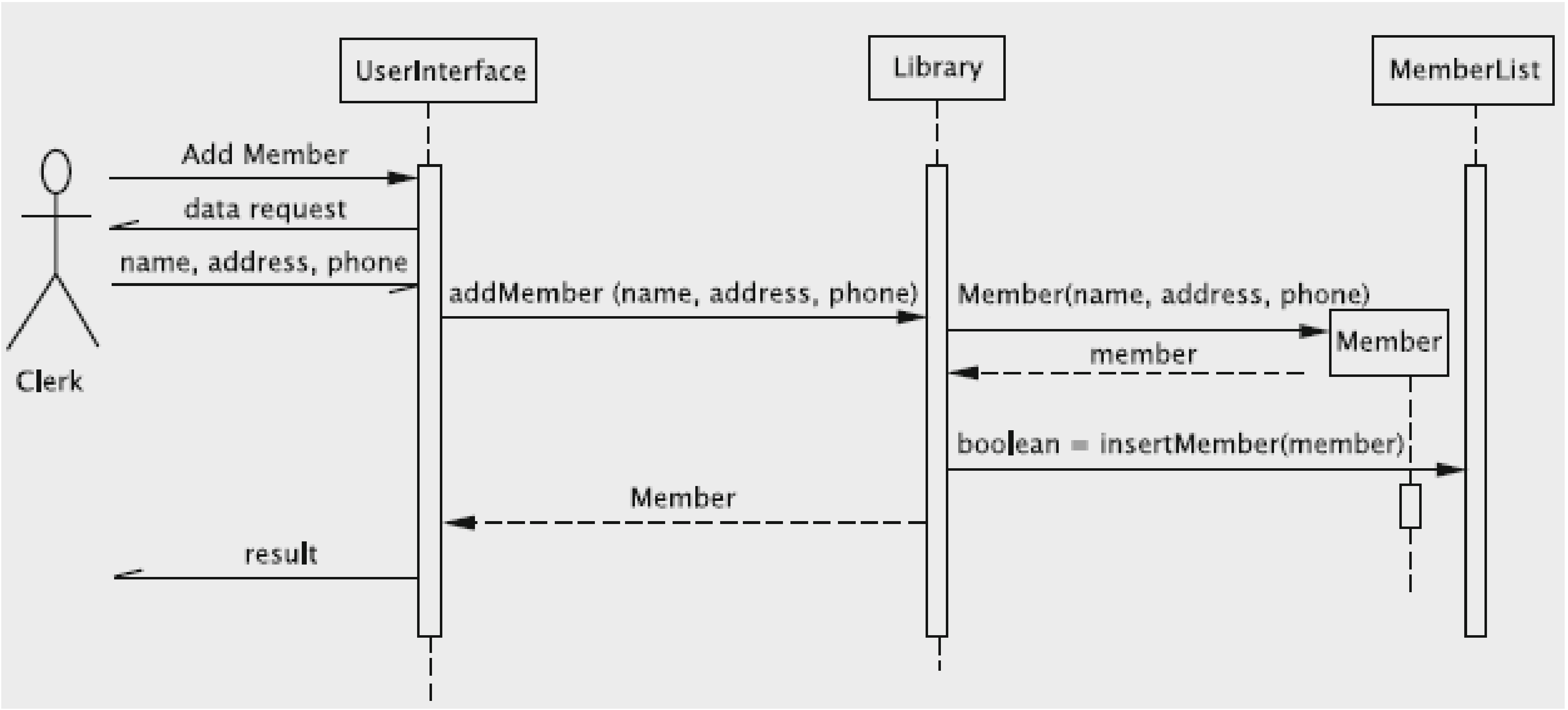
- Invoke the Member constructor from within the addMember method of Library.
- The constructor returns a reference to the Member object and an operation, insertMember, is invoked on MemberList to add the new member.

Option2:

- Invoke an addNewMember method on MemberList and pass as parameters all the data about the new member.
- MemberList creates the Member object and adds it to the collection.

To carry out the first two steps, we have two options:

Option1:



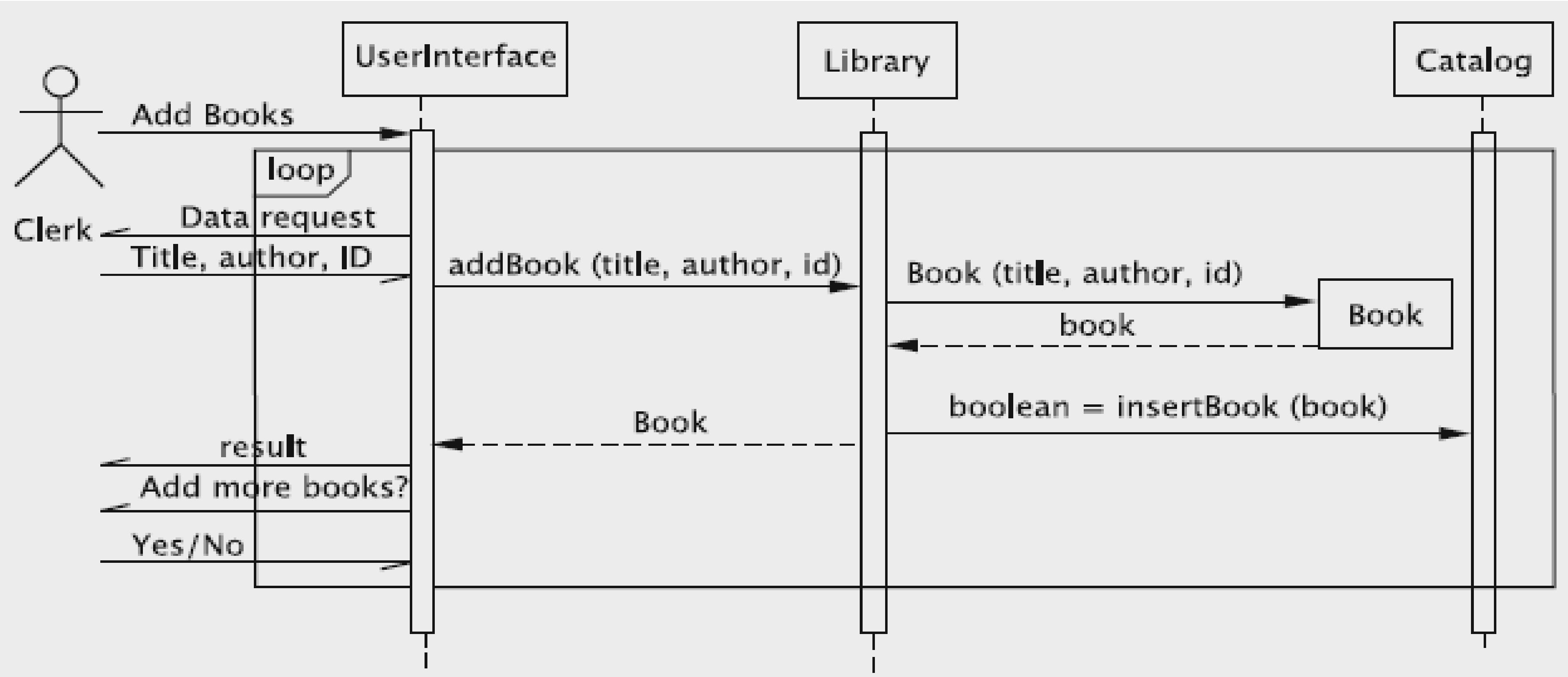
- Invoke the Member constructor from within the addMember method of Library.
- The constructor returns a reference to the Member object and an operation, insertMember, is invoked on MemberList to add the new member.

Add Books: This use case allows the insertion of an arbitrary number of books into the system. In this case, when the request is made by the actor, the system enters a loop. The algorithm here consists of the following steps:

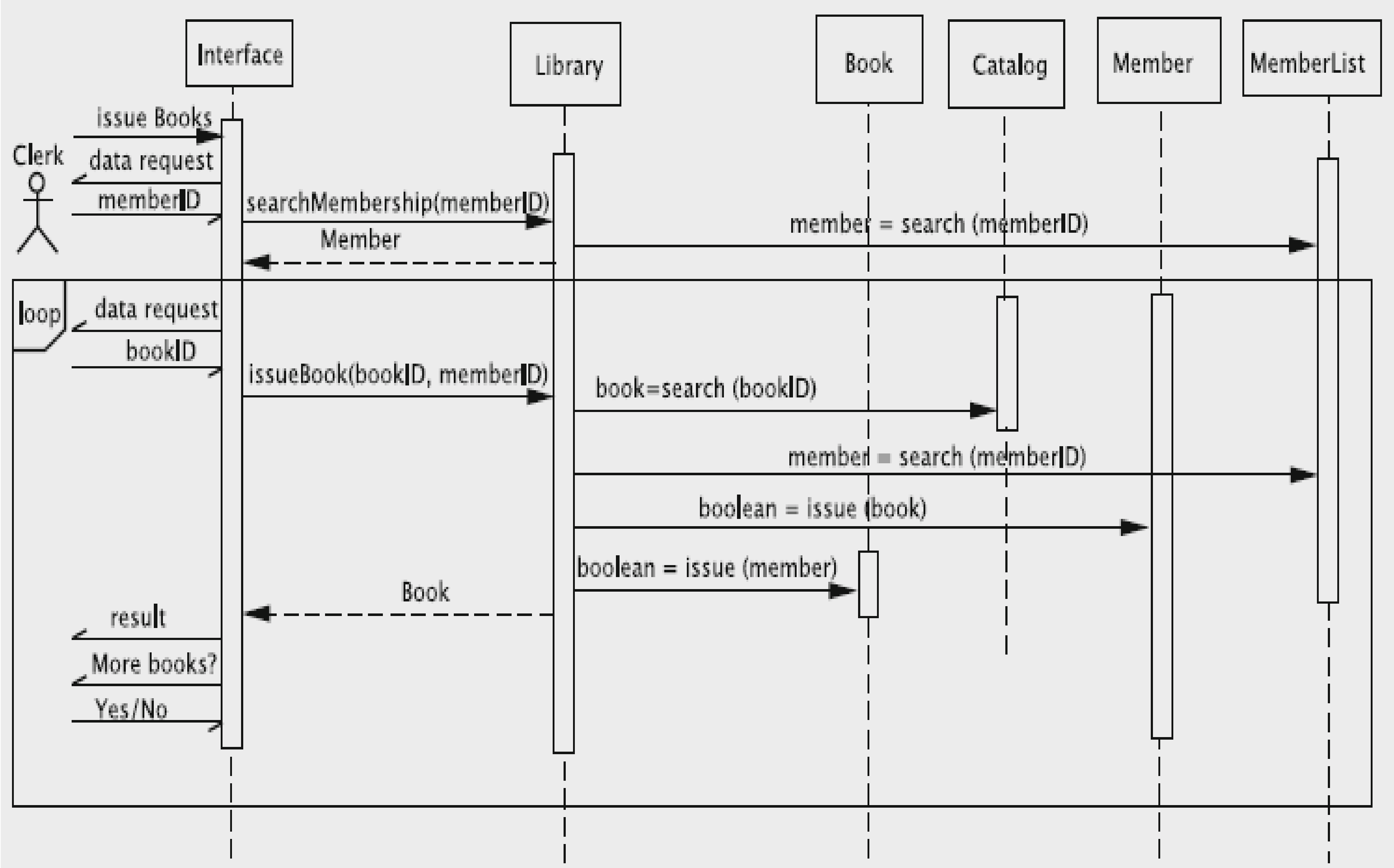
(i) create a Book object,

(ii) add the Book object to the catalog and

(iii) return the result of the operation.



Issue Books: When a book is to be checked out, the clerk interacts with the UI to input the user’s ID. The system has to first check the validity of the user. This is accomplished by invoking the

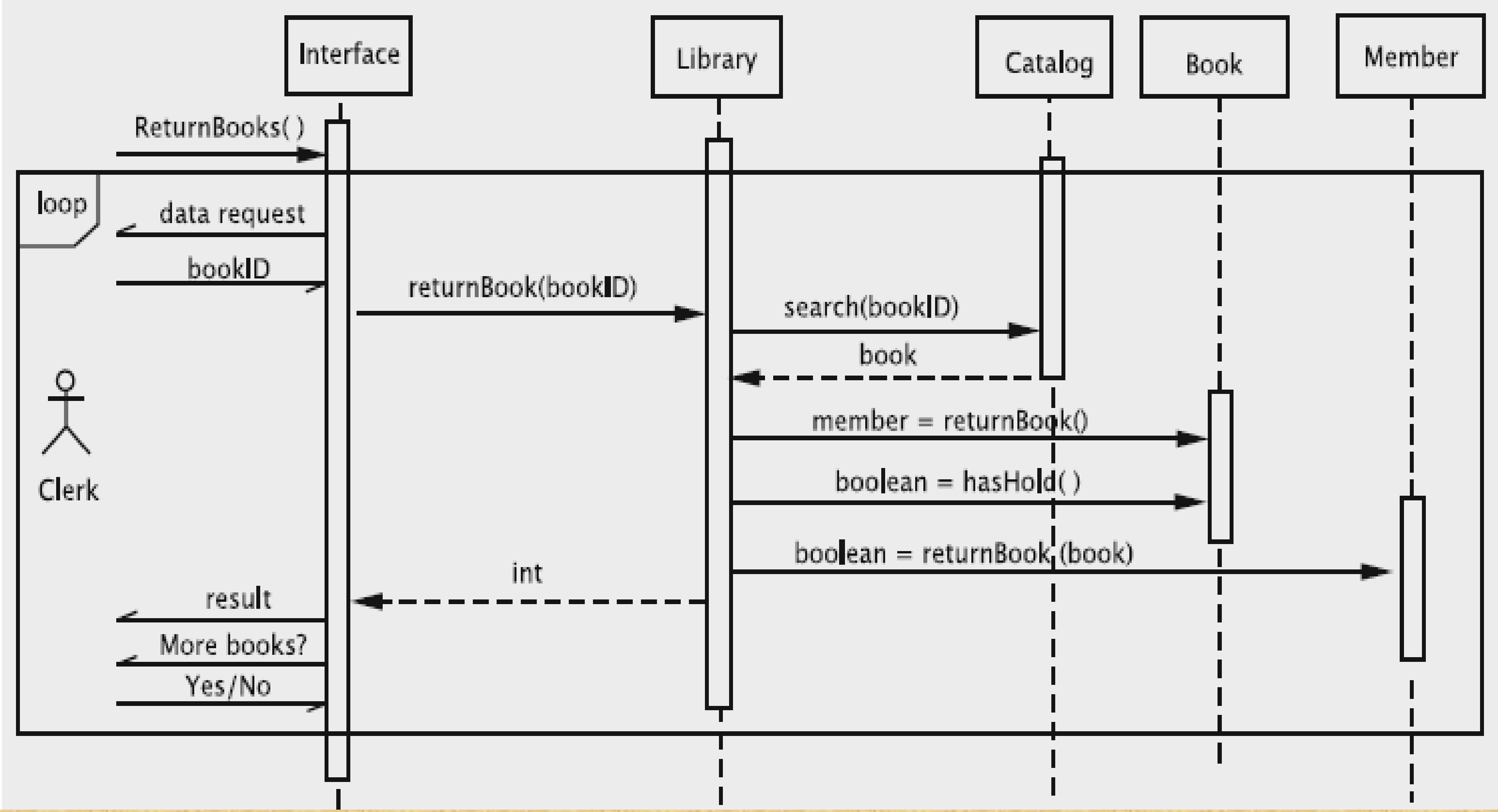


Return Books:

- For each book returned, the returnBook method of the Library class obtains the corresponding Book object from Catalog.
- The returnBook method is invoked using this Book object, and this method returns the Member object corresponding to the member who had borrowed the book.
- The returnBook method of the Member object is now called to record that the book has been returned.
- This operation has three possible outcomes that the use case requires the system to distinguish.

1. *The book's IDwas invalid,*whichwould result in the operation being unsuccessful;
2. *the operation was successful;*
3. *The operation was successful and there is a hold on the book.*

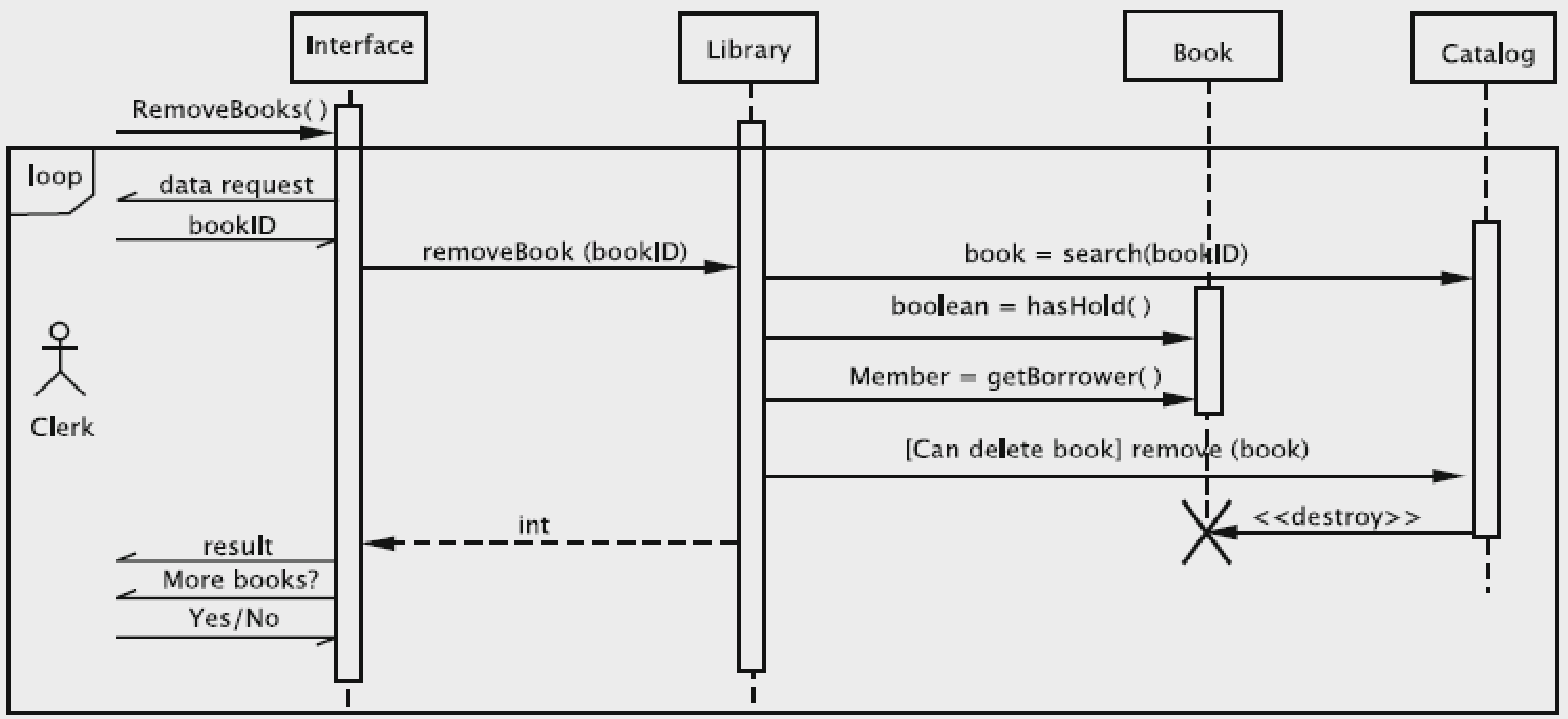
*T*he value returned by returnBook must enable UserInterface to make the distinction between these.



Sequence diagram for returning books

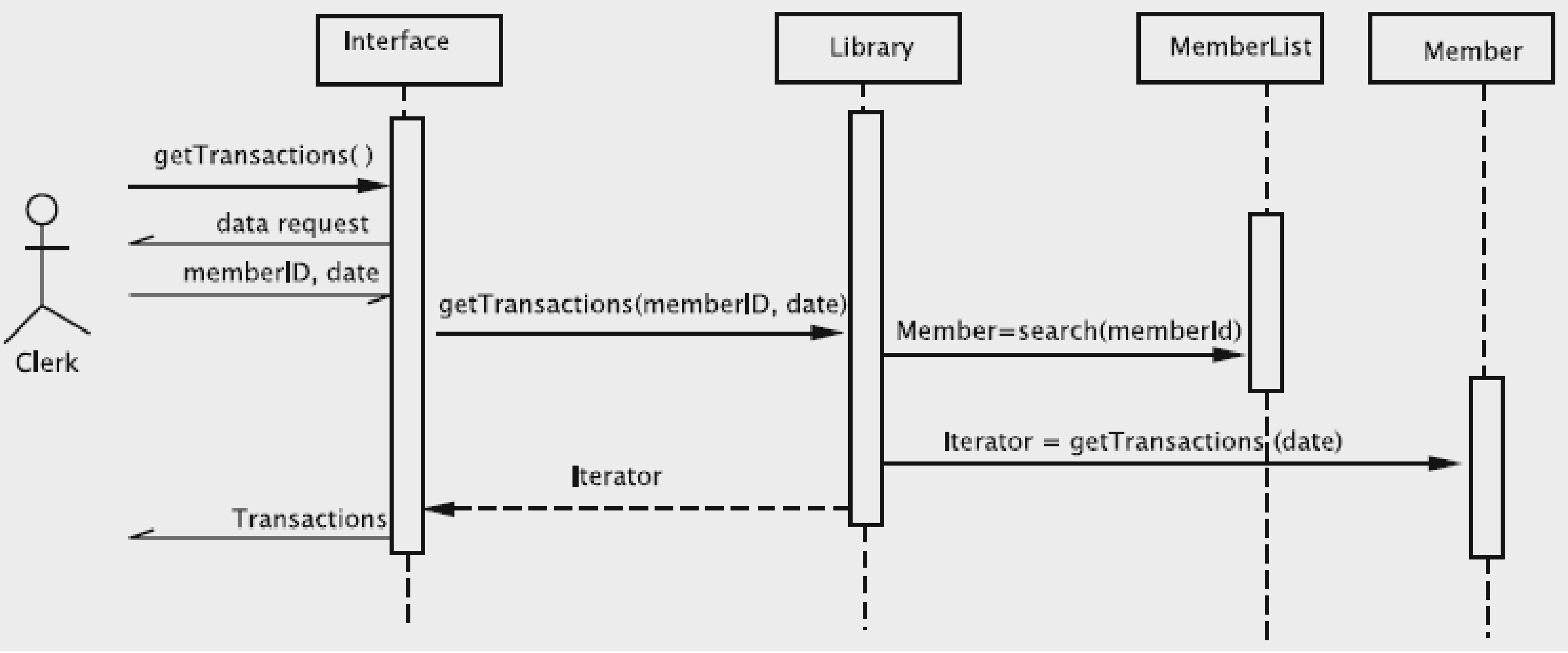
Remove Books:

- we remove only those books that are not checked out and do not have a hold. This logic for deciding whether the book is removable is in the removeBook method in Library.
- This method checks each property of the book in question and if all properties are satisfied, the remove method in Catalog is invoked, which then removes the book.



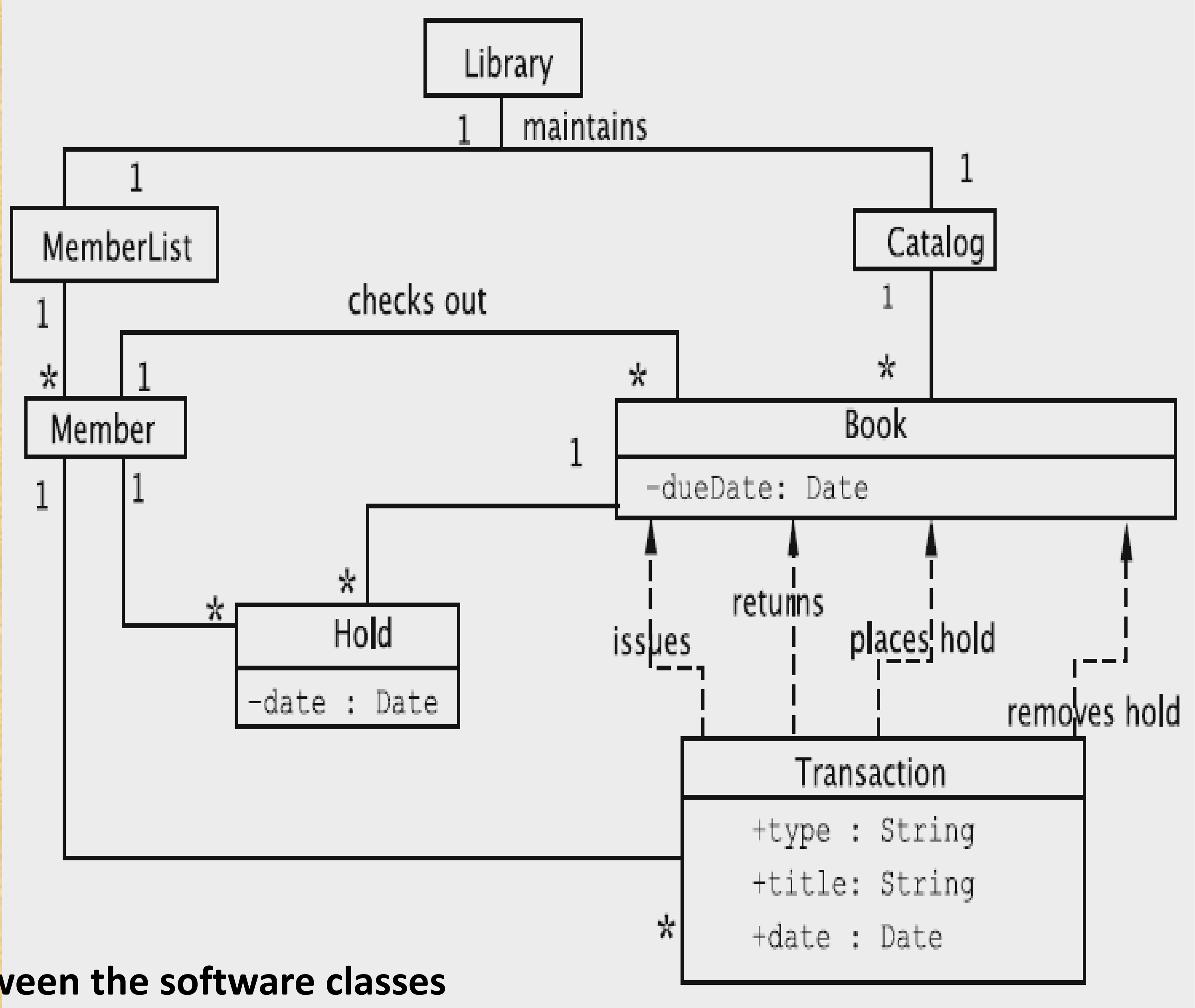
Member Transactions:

- The end-user (clerk) interacts with the Library class to print out the transactions of a given member.
- The Member class stores the necessary information about the transactions, but the UI would be the one to decide the format.
- It would, therefore, be desirable to provide the information to the UI as a collection of objects, each object containing the information about a particular transaction.

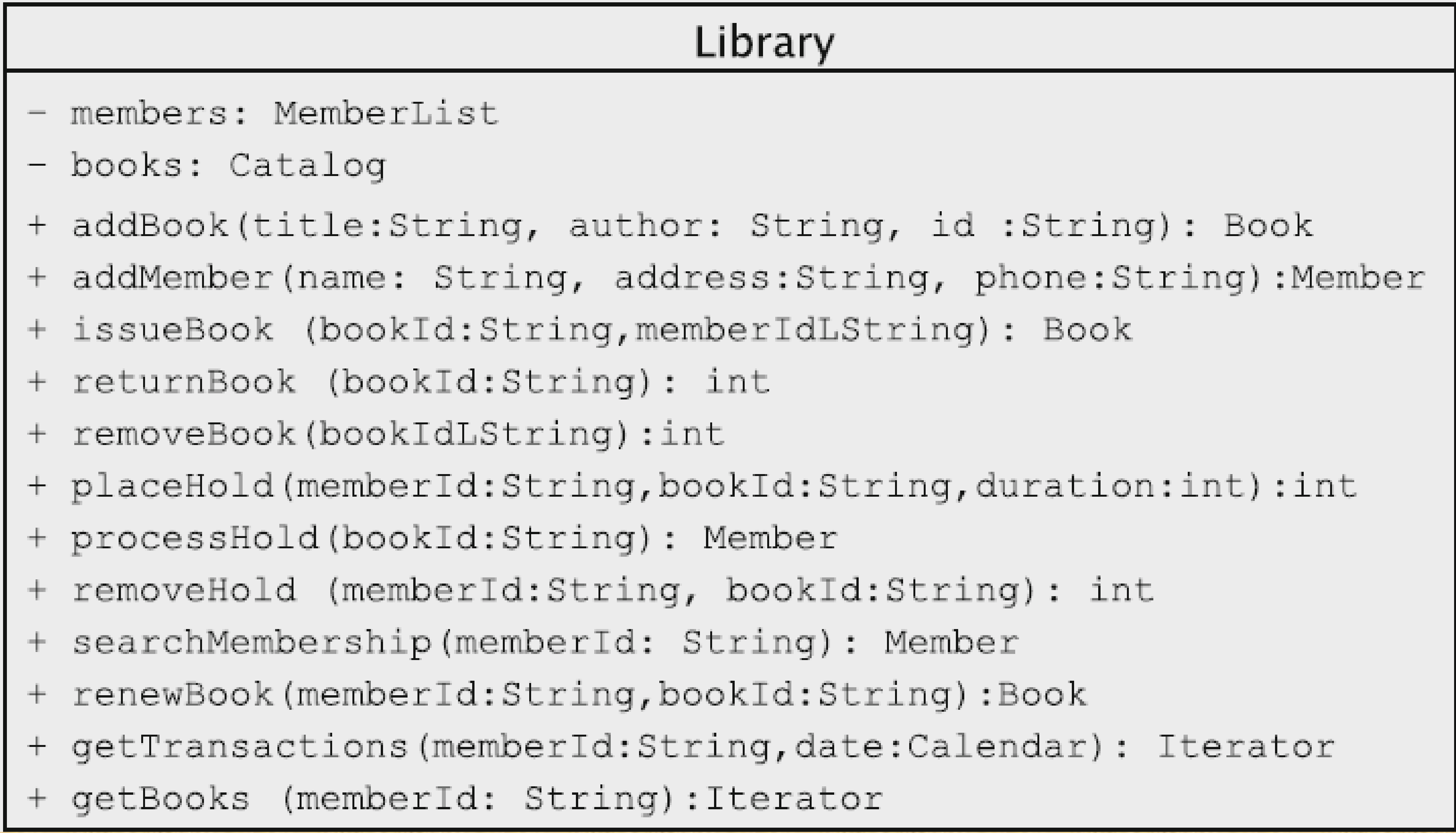


Class Diagrams:

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



Relationships between the software classes

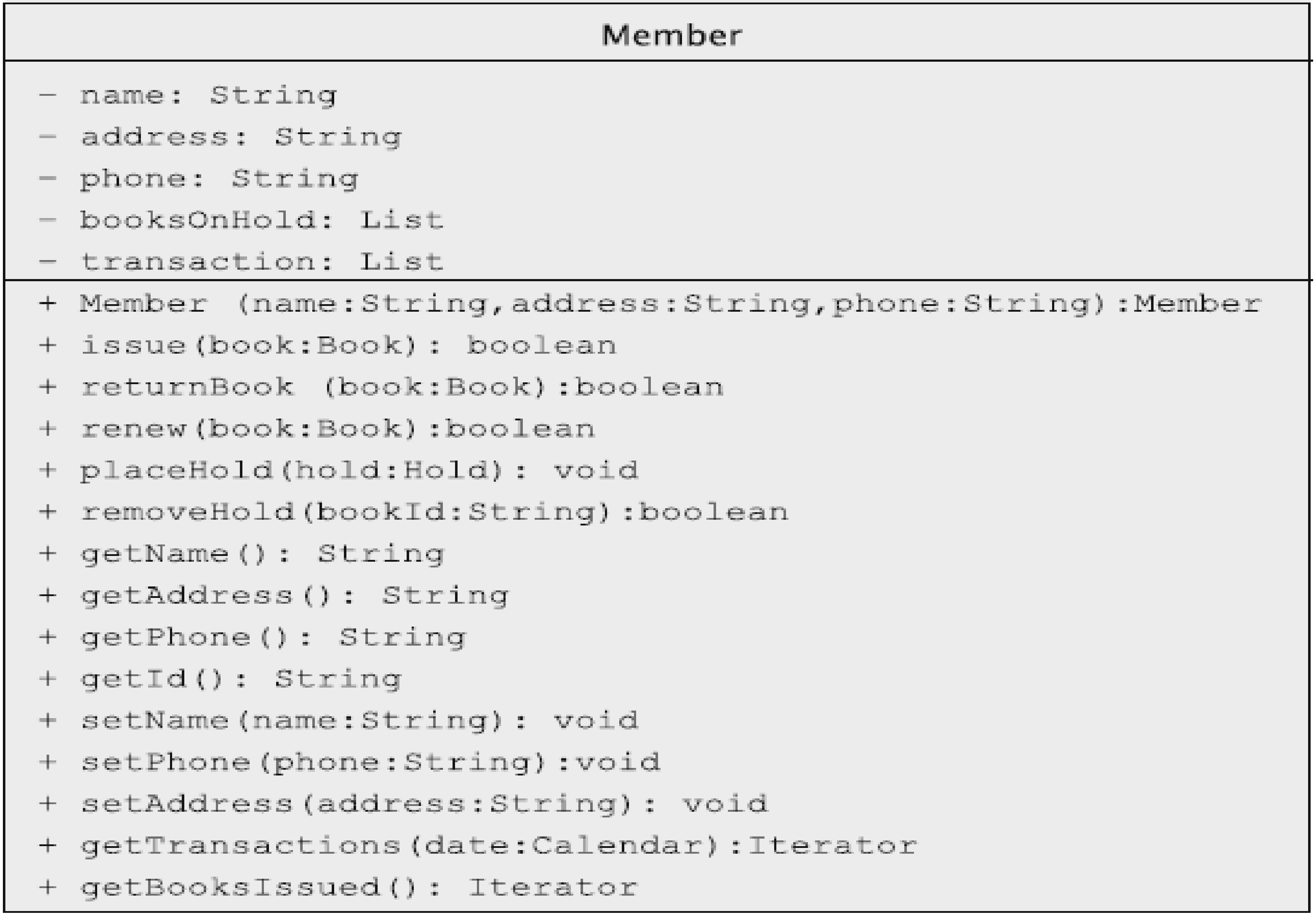


Library

```
- members: MemberList
- books: Catalog
+ addBook(title:String, author: String, id :String): Book
+ addMember(name: String, address:String, phone:String):Member
+ issueBook (bookId:String,memberIdLString): Book
+ returnBook (bookId:String): int
+ removeBook(bookIdLString):int
+ placeHold(memberId:String,bookId:String,duration:int):int
+ processHold(bookId:String): Member
+ removeHold (memberId:String, bookId:String): int
+ searchMembership(memberId: String): Member
+ renewBook(memberId:String,bookId:String):Book
+ getTransactions(memberId:String,date:Calendar): Iterator
+ getBooks (memberId: String):Iterator
```

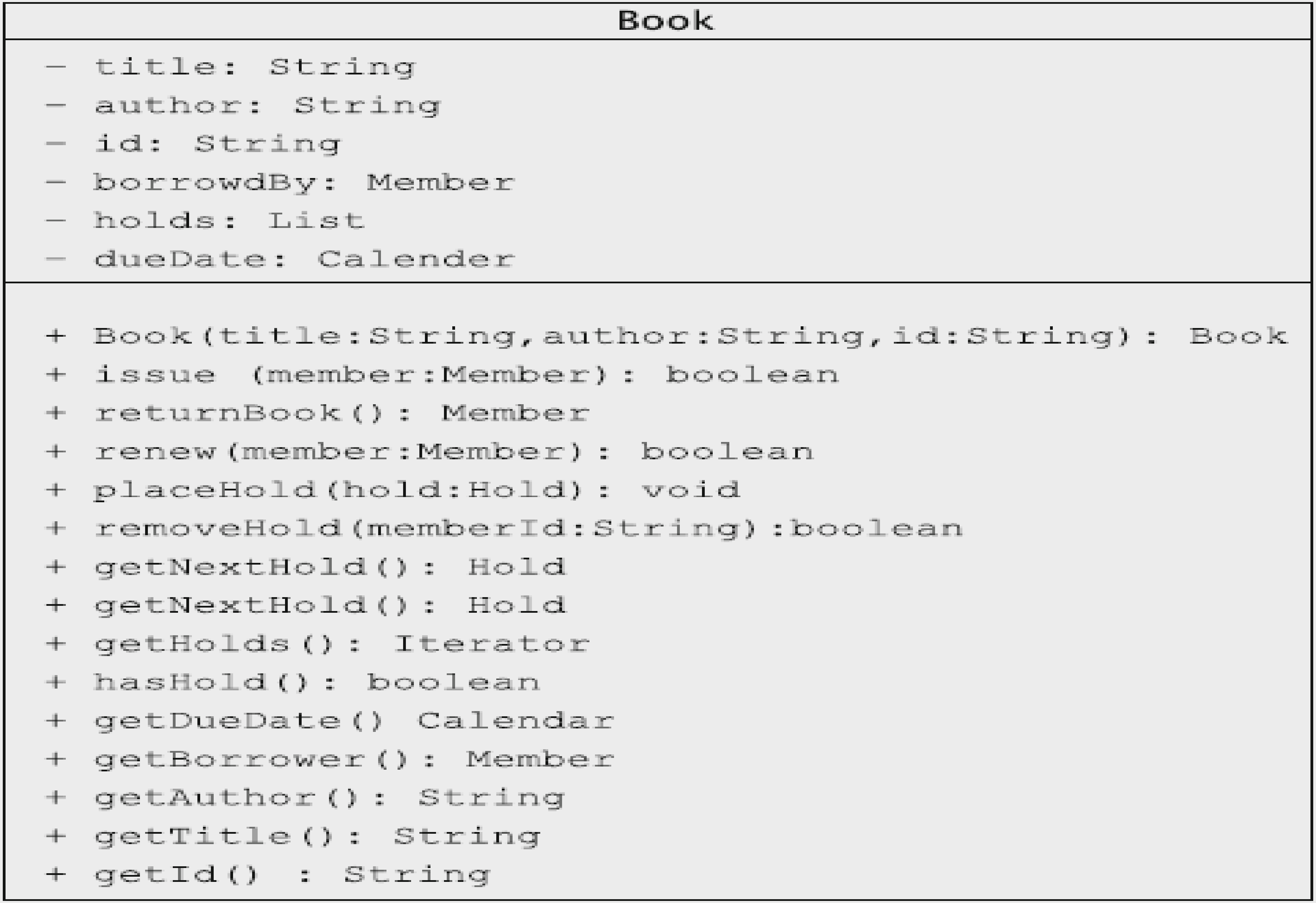
Class diagram for Library

Class Diagram for Member



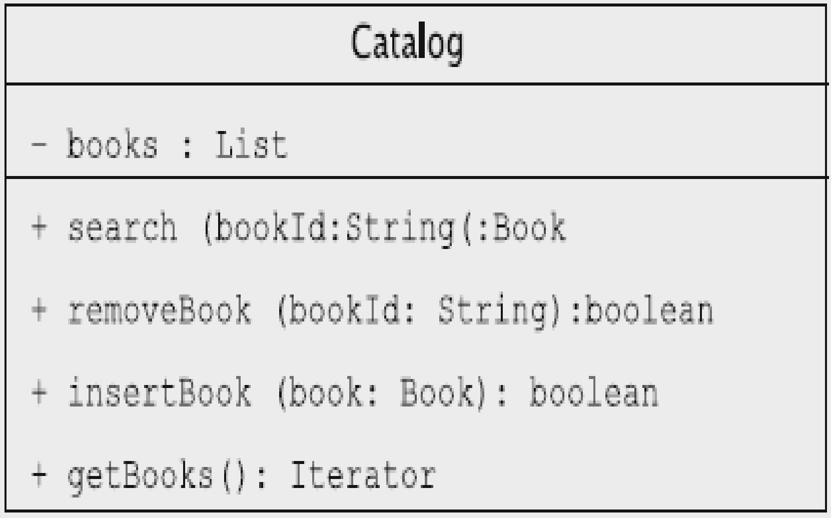
```
Member

- name: String
- address: String
- phone: String
- booksOnHold: List
- transaction: List

+ Member (name:String,address:String,phone:String):Member
+ issue(book:Book): boolean
+ returnBook (book:Book):boolean
+ renew(book:Book):boolean
+ placeHold(hold:Hold): void
+ removeHold(bookId:String):boolean
+ getName(): String
+ getAddress(): String
+ getPhone(): String
+ getId(): String
+ setName(name:String): void
+ setPhone(phone:String):void
+ setAddress(address:String): void
+ getTransactions(date:Calendar):Iterator
+ getBooksIssued(): Iterator
```

Class Diagram for Book



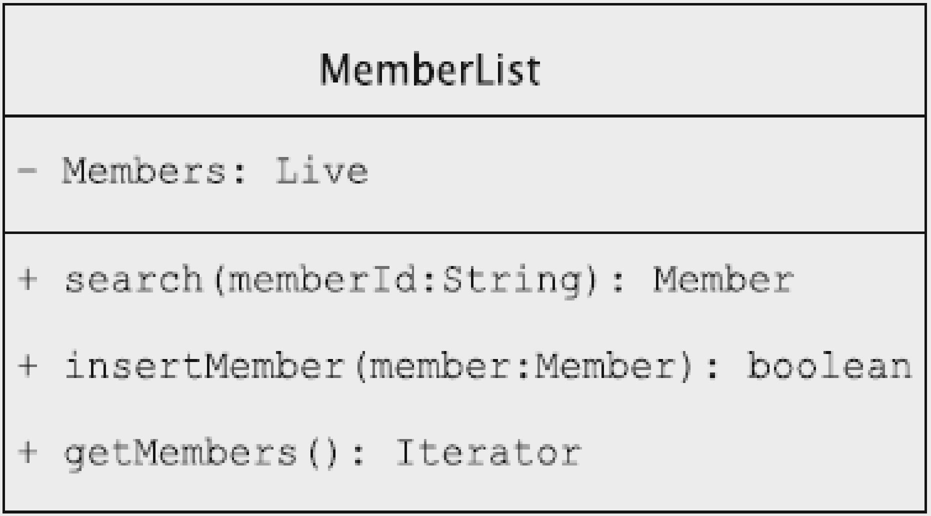
```
Book

- title: String
- author: String
- id: String
- borrowdBy: Member
- holds: List
- dueDate: Calender

+ Book(title:String,author:String,id:String): Book
+ issue (member:Member): boolean
+ returnBook(): Member
+ renew(member:Member): boolean
+ placeHold(hold:Hold): void
+ removeHold(memberId:String):boolean
+ getNextHold(): Hold
+ getNextHold(): Hold
+ getHolds(): Iterator
+ hasHold(): boolean
+ getDueDate() Calendar
+ getBorrower(): Member
+ getAuthor(): String
+ getTitle(): String
+ getId() : String
```

Class Diagram for Catalog



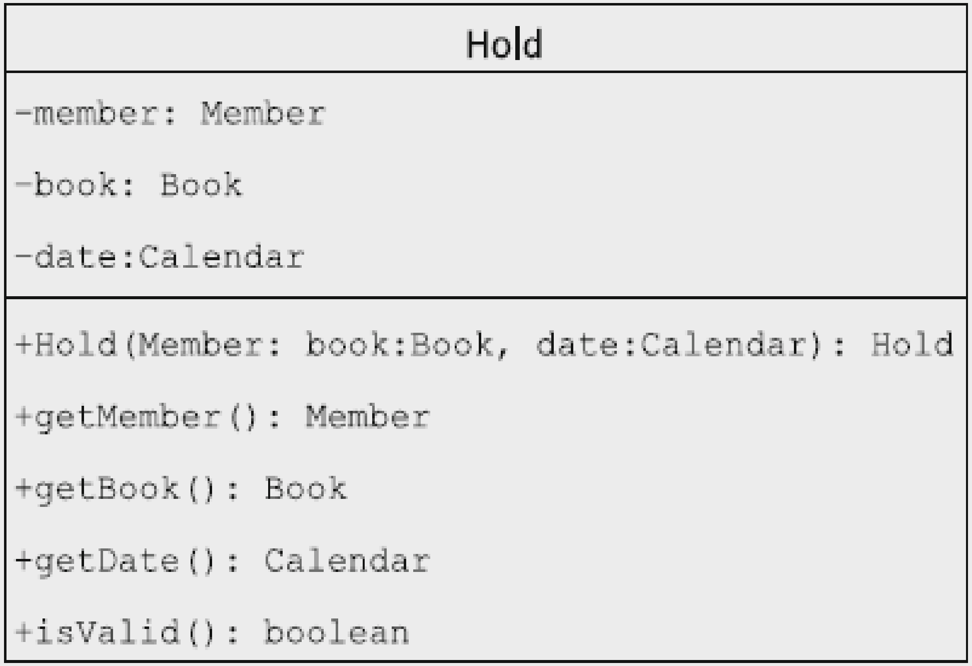
```
Catalog
- books : List
+ search (bookId:String(:Book
+ removeBook (bookId: String):boolean
+ insertBook (book: Book): boolean
+ getBooks(): Iterator
```

Class Diagram for MemberList



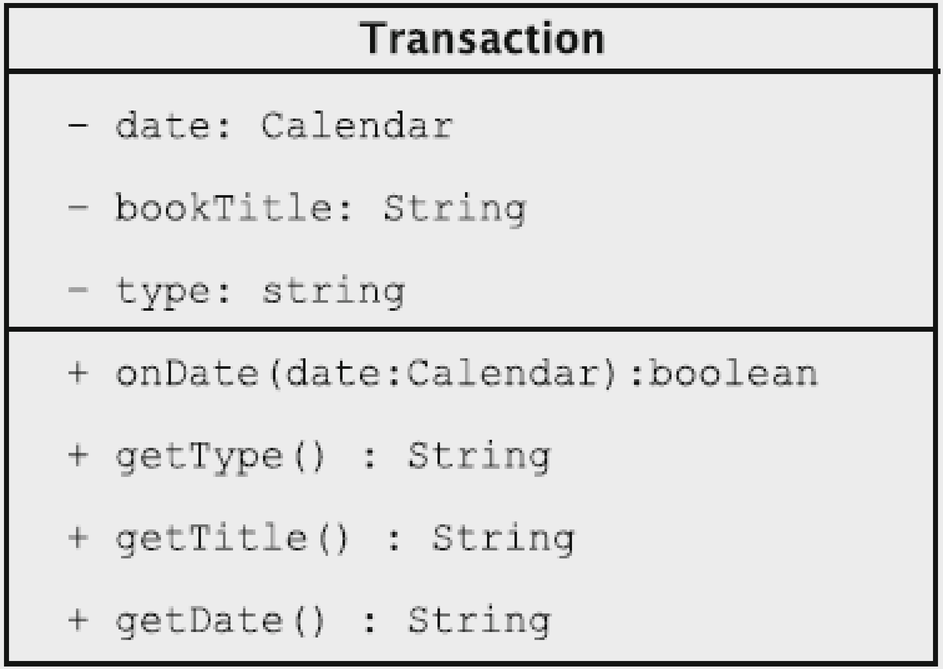
```
MemberList
- Members: Live
+ search(memberId:String): Member
+ insertMember(member:Member): boolean
+ getMembers(): Iterator
```

Class Diagram for Hold



```
Hold
-member: Member
-book: Book
-date:Calendar
+Hold(Member: book:Book, date:Calendar): Hold
+getMember(): Member
+getBook(): Book
+getDate(): Calendar
+isValid(): boolean
```

Class Diagram for Transaction



```
Transaction
- date: Calendar
- bookTitle: String
- type: string
+ onDate(date:Calendar):boolean
+ getType() : String
+ getTitle() : String
+ getDate() : String
```

User Interface: UI provides a menu with the following options

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

Data Storage

In a full-blown system, data is usually stored in a database, and this data is managed by a database management system.

Include the following commands in 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.

Implementing Our Design

- Library has several methods that return int values, and these values must be interpreted by the UI.
- A separate named constant is declared for each of these outcomes as shown below.

```
public static final int BOOK_NOT_FOUND = 1;
public static final int BOOK_NOT_ISSUED = 2;
// etc.
```

Setting Up the Interface:

The main program resides in the class UserInterface. When the main program is executed, an instance of the UserInterface is created (a singleton).

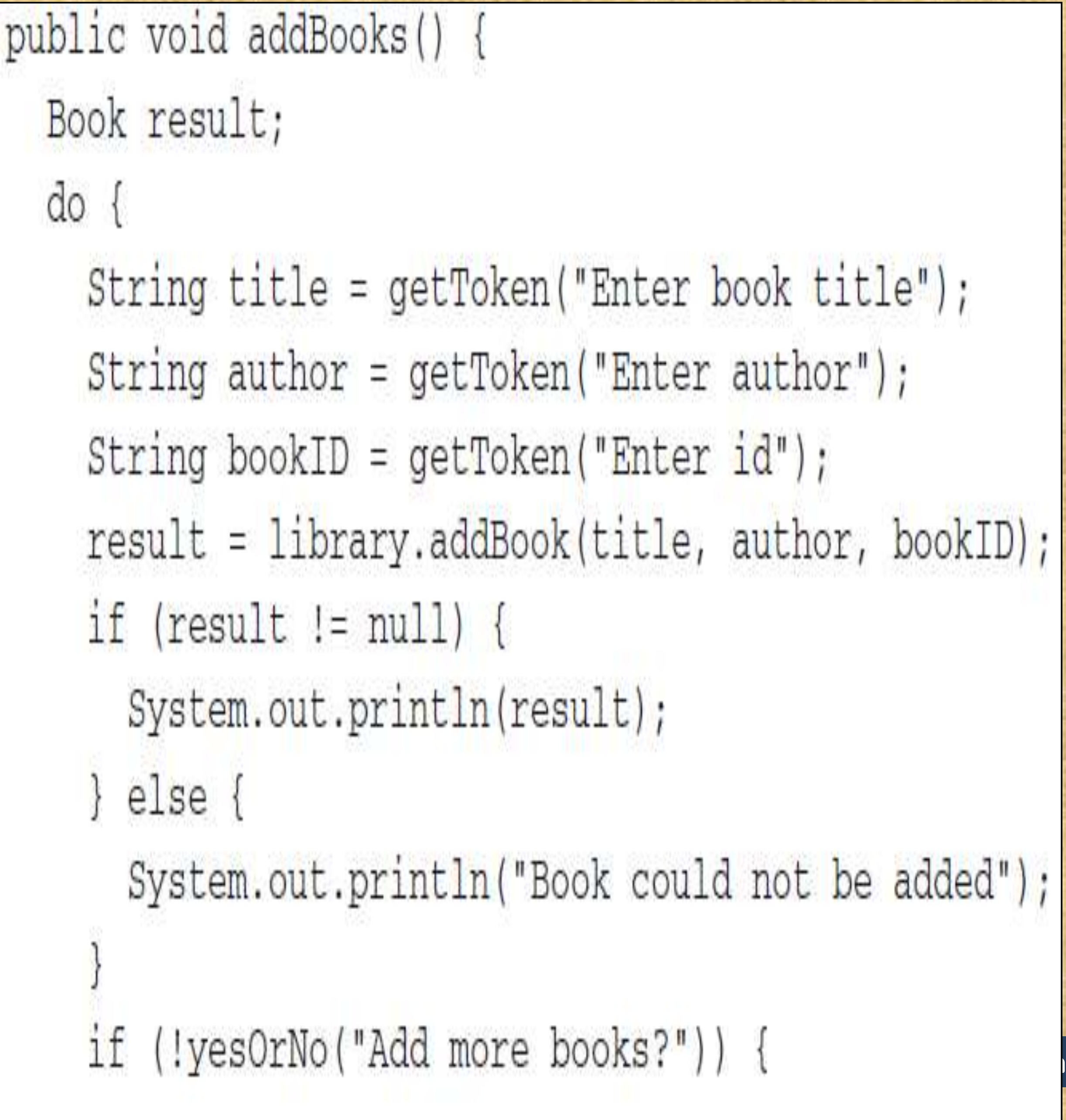
```
public static void main(String[] s)
{
UserInterface.instance().process();
}

public static UserInterface instance()
{
if (userInterface == null) {
return userInterface = new UserInterface();
} else {
return userInterface;
}
}
```

```
private UserInterface()
{
File file = new File("LibraryData");
if (file.exists() && file.canRead()) {
if (yesOrNo("Saved data exists. Use it?"))
{
retrieve();
}
}
library = Library.instance();
}
```

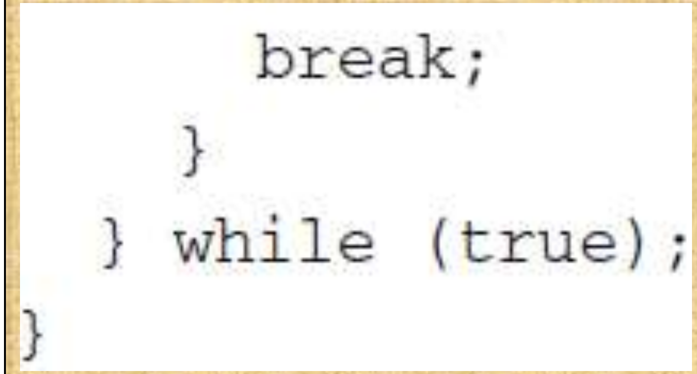
Process Method of UI

```
public void process() {
int command;
help();
while ((command = getCommand()) != EXIT) {
switch (command) {
case ADD_MEMBER: addMember();
break;
case ADD_BOOKS: addBooks();
break;
case ISSUE_BOOKS: issueBooks();
break;
// several lines of code not shown
case HELP: help();
break;
}
}
}
```

Adding New Books

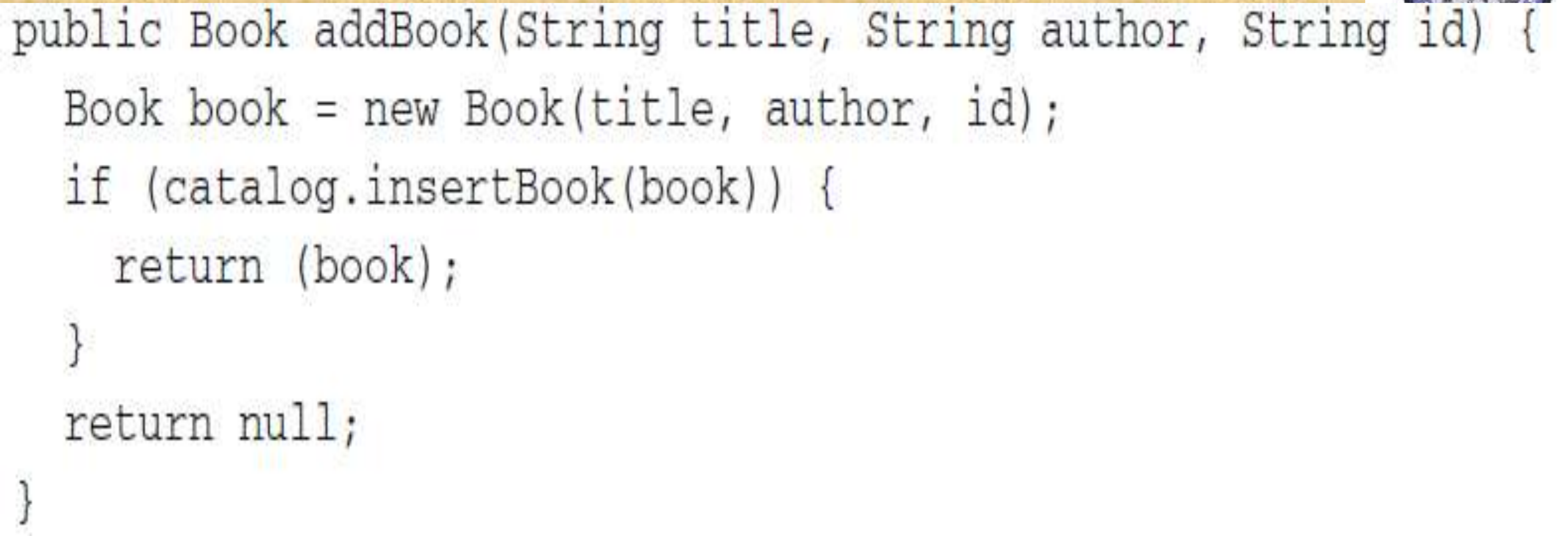


```
public void addBooks() {
  Book result;
  do {
    String title = getToken("Enter book title");
    String author = getToken("Enter author");
    String bookID = getToken("Enter id");
    result = library.addBook(title, author, bookID);
    if (result != null) {
      System.out.println(result);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



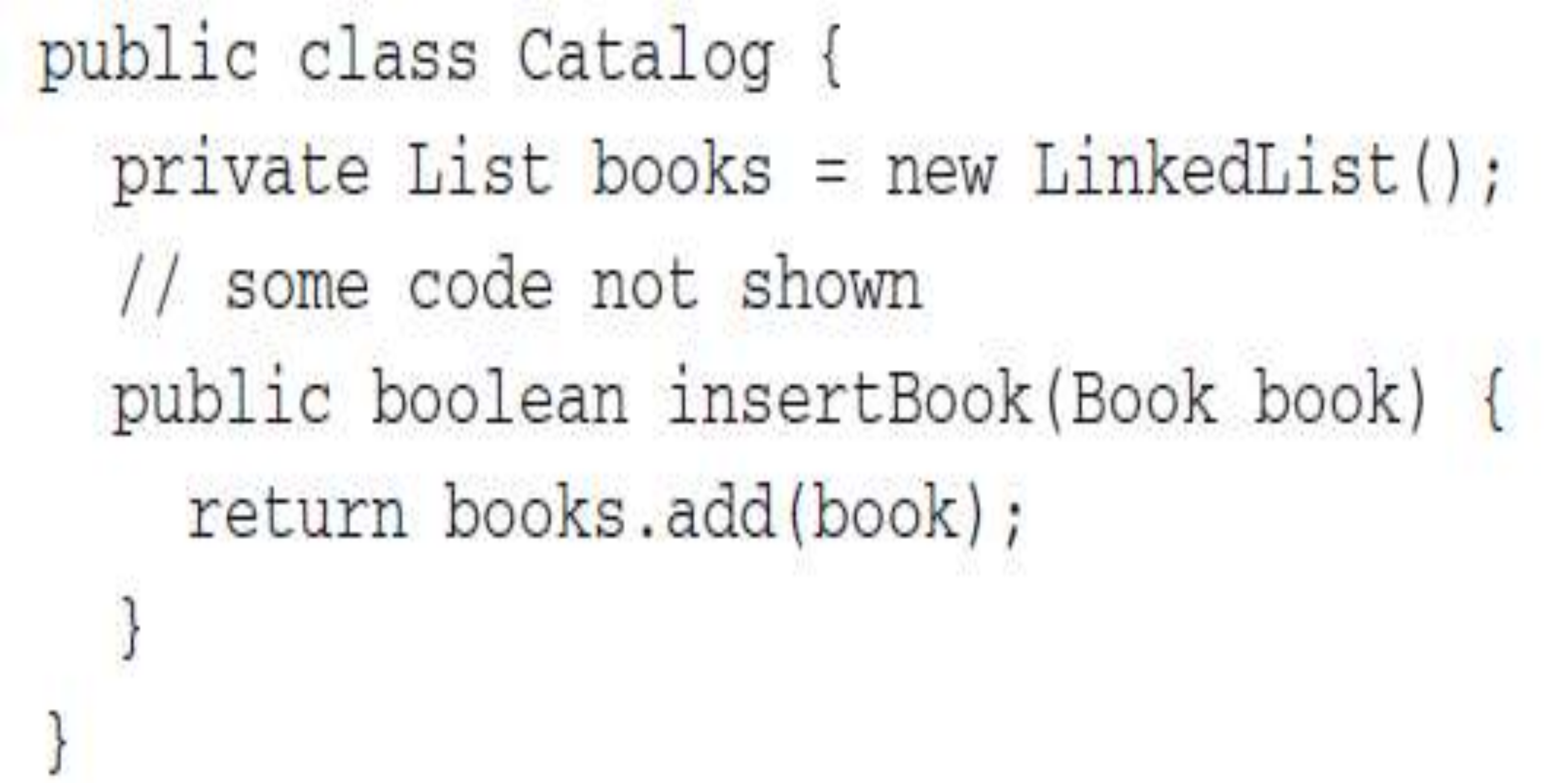
```
      break;
    }
  } while (true);
}
```

addBook() method in Library is invoked:



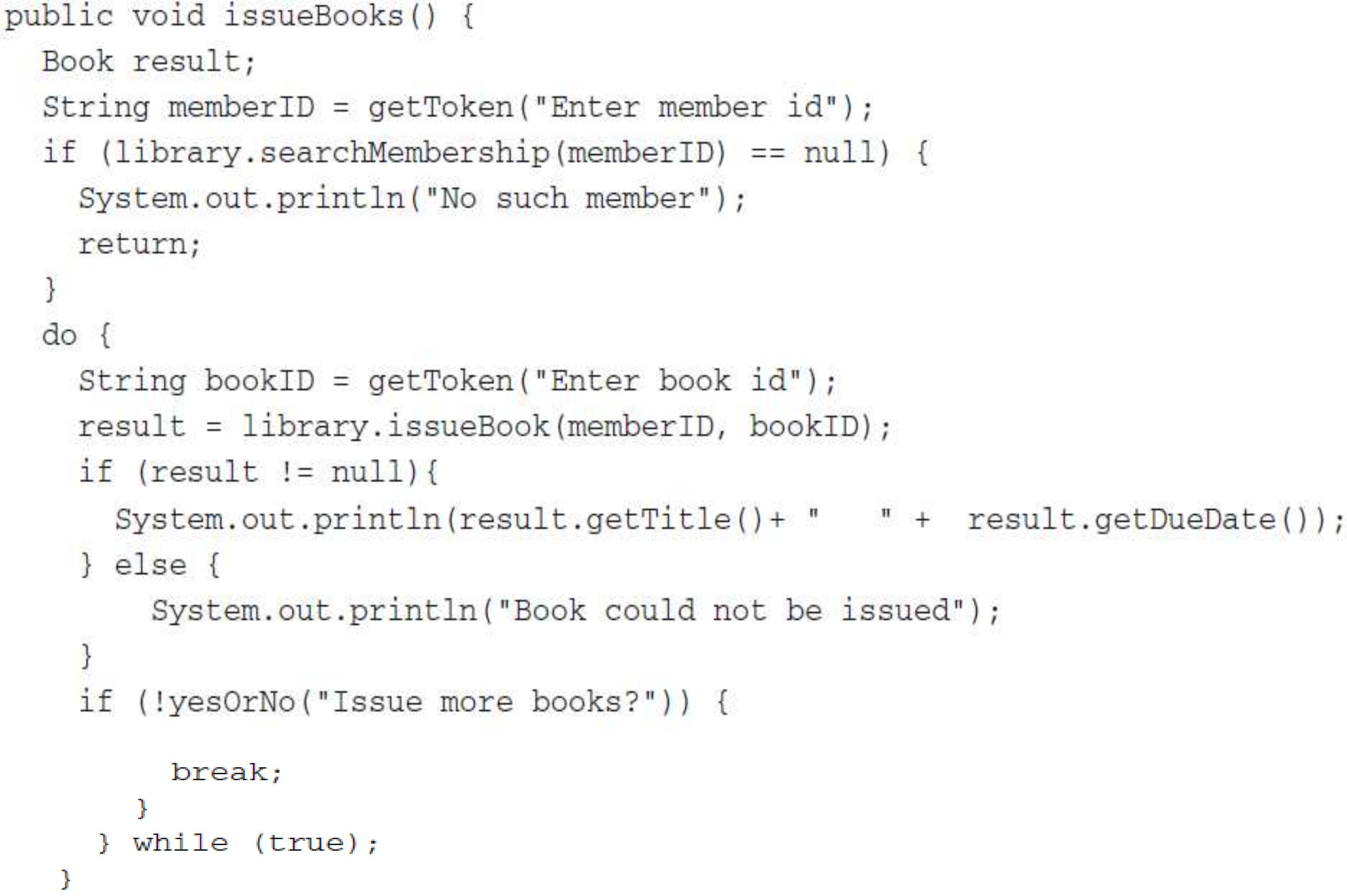
```
public Book addBook(String title, String author, String id) {
  Book book = new Book(title, author, id);
  if (catalog.insertBook(book)) {
    return (book);
  }
  return null;
}
```

The Catalog (which is also a singleton) is an adapter for the LinkedList class, so all it does is to invoke the add method in Java’s LinkedList class,



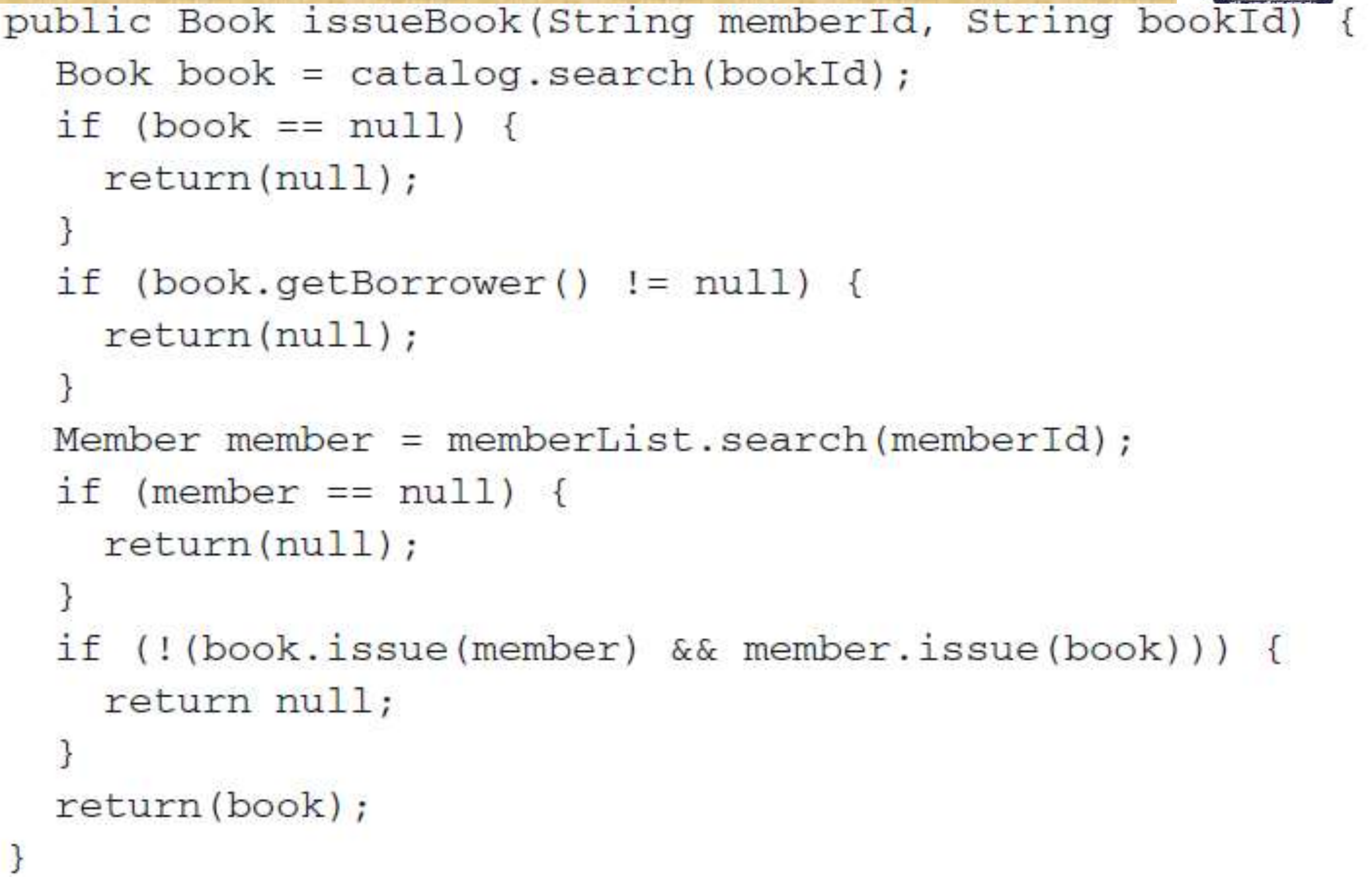
```
public class Catalog {
  private List books = new LinkedList();
  // some code not shown
  public boolean insertBook(Book book) {
    return books.add(book);
  }
}
```

Issuing Books



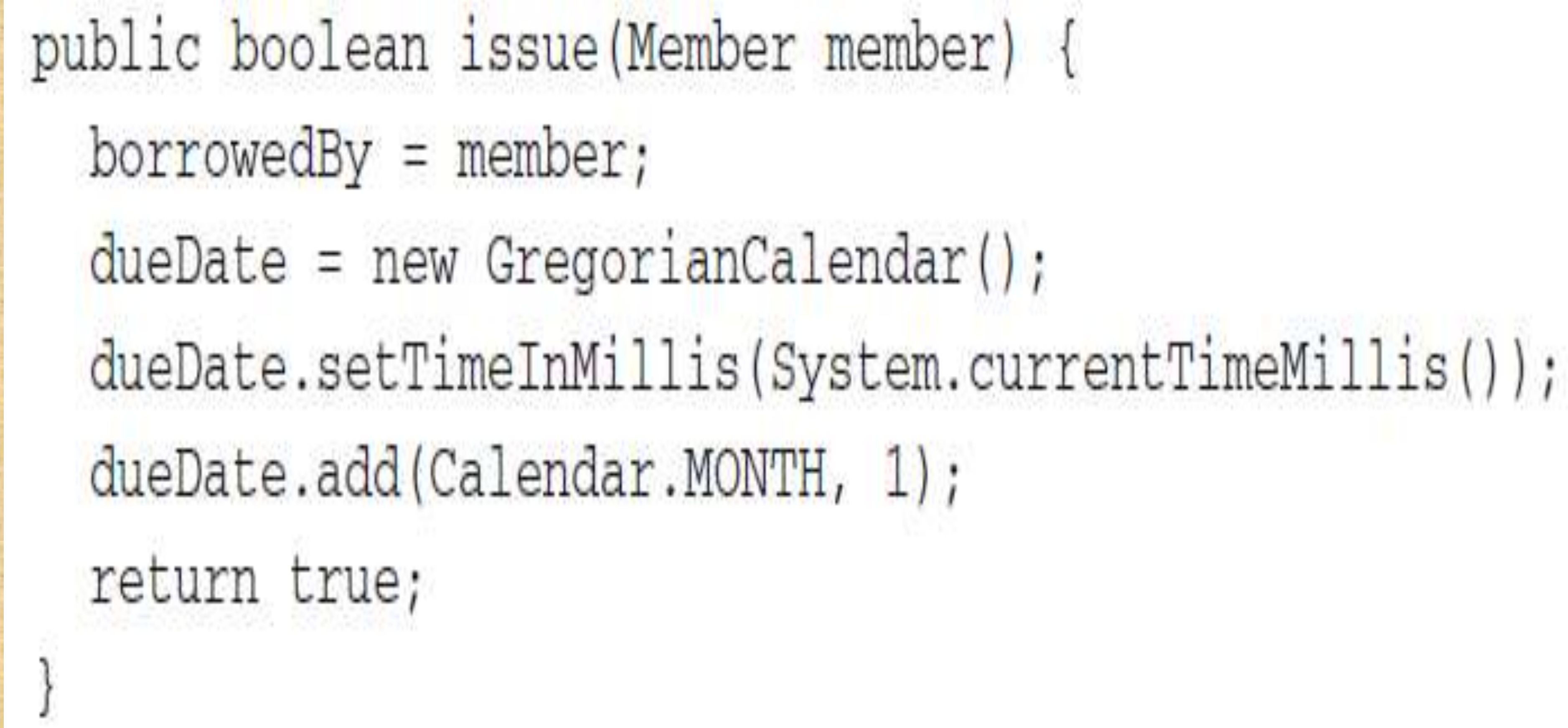
```
public void issueBooks() {
  Book result;
  String memberID = getToken("Enter member id");
  if (library.searchMembership(memberID) == null) {
    System.out.println("No such member");
    return;
  }
  do {
    String bookID = getToken("Enter book id");
    result = library.issueBook(memberID, bookID);
    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 issueBook( )s the necessary processing and returns a reference to the issued book.



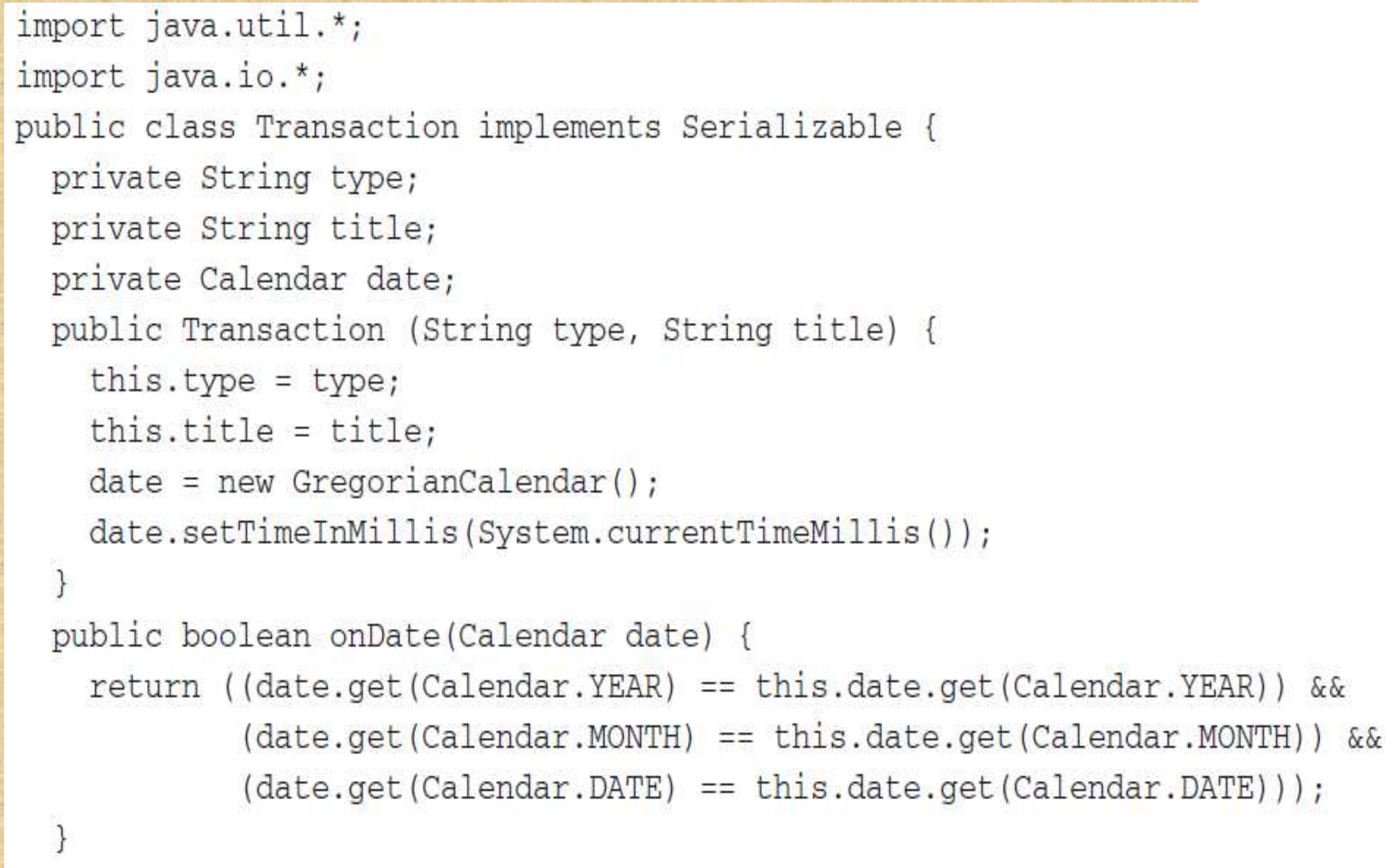
```
public Book issueBook(String memberId, String bookId) {
  Book book = catalog.search(bookId);
  if (book == null) {
    return(null);
  }
  if (book.getBorrower() != null) {
    return(null);
  }
  Member member = memberList.search(memberId);
  if (member == null) {
    return(null);
  }
  if (!(book.issue(member)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

The issue methods in Book and Member record the fact that the book is being issued. The method in Book generates a due date for our simple library by adding one month to the date of issue.



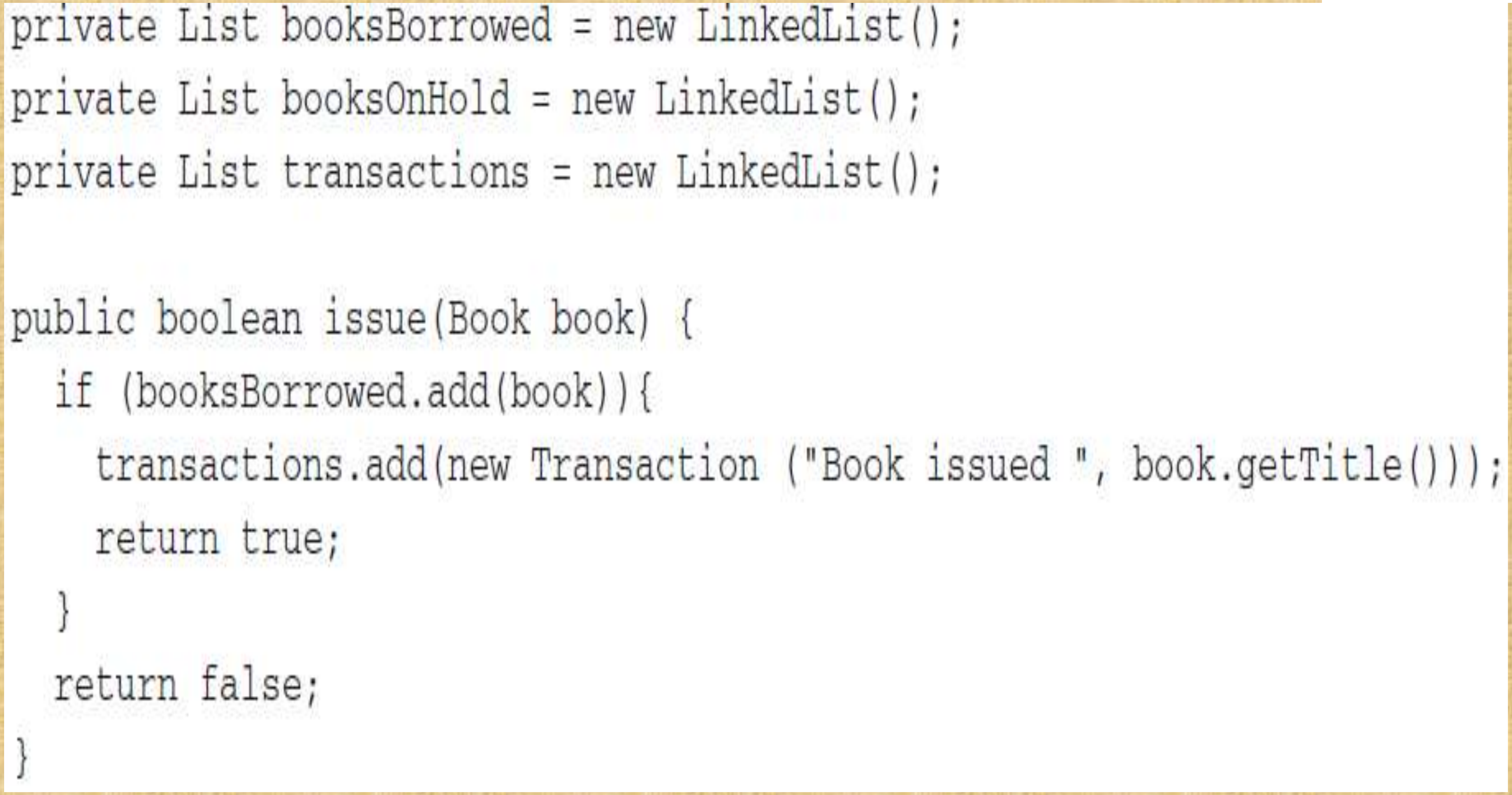
```
public boolean issue(Member member) {
  borrowedBy = member;
  dueDate = new GregorianCalendar();
  dueDate.setTimeInMillis(System.currentTimeMillis());
  dueDate.add(Calendar.MONTH, 1);
  return true;
}
```

Member is also keeping track of all the transactions (issues and returns) that the member has completed. This is done by defining the class Transaction.



```
import java.util.*;
import java.io.*;
public class Transaction implements Serializable {
  private String type;
  private String title;
  private Calendar date;
  public Transaction (String type, String title) {
    this.type = type;
    this.title = title;
    date = new GregorianCalendar();
    date.setTimeInMillis(System.currentTimeMillis());
  }
  public boolean onDate(Calendar date) {
    return ((date.get(Calendar.YEAR) == this.date.get(Calendar.YEAR)) &&
            (date.get(Calendar.MONTH) == this.date.get(Calendar.MONTH)) &&
            (date.get(Calendar.DATE) == this.date.get(Calendar.DATE)));
  }
  public String getType() {
    return type;
  }
  public String getTitle() {
    return title;
  }
  public String getDate() {
    return date.get(Calendar.MONTH) + "/" + date.get(Calendar.DATE) + "/"
                                          + date.get(Calendar.YEAR);
  }
  public String toString(){
    return (type + "   " + title);
  }
}
```

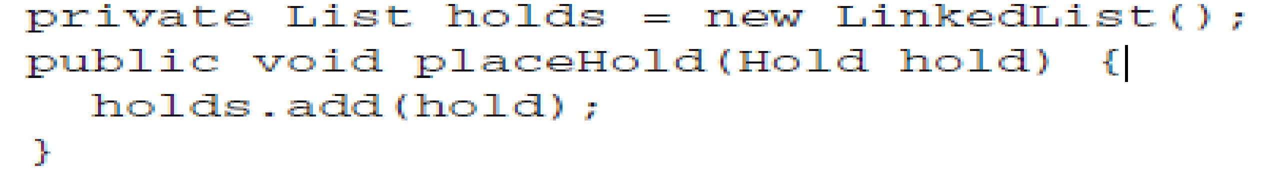
With each book issued, a record is created and added to the list of transactions, as shown in the following code snippet from Member.



```
private List booksBorrowed = new LinkedList();
private List booksOnHold = new LinkedList();
private List transactions = new LinkedList();


public boolean issue(Book book) {
  if (booksBorrowed.add(book)){
    transactions.add(new Transaction ("Book issued ", book.getTitle()));
    return true;
  }
  return false;
}
```

Placing and Processing Holds

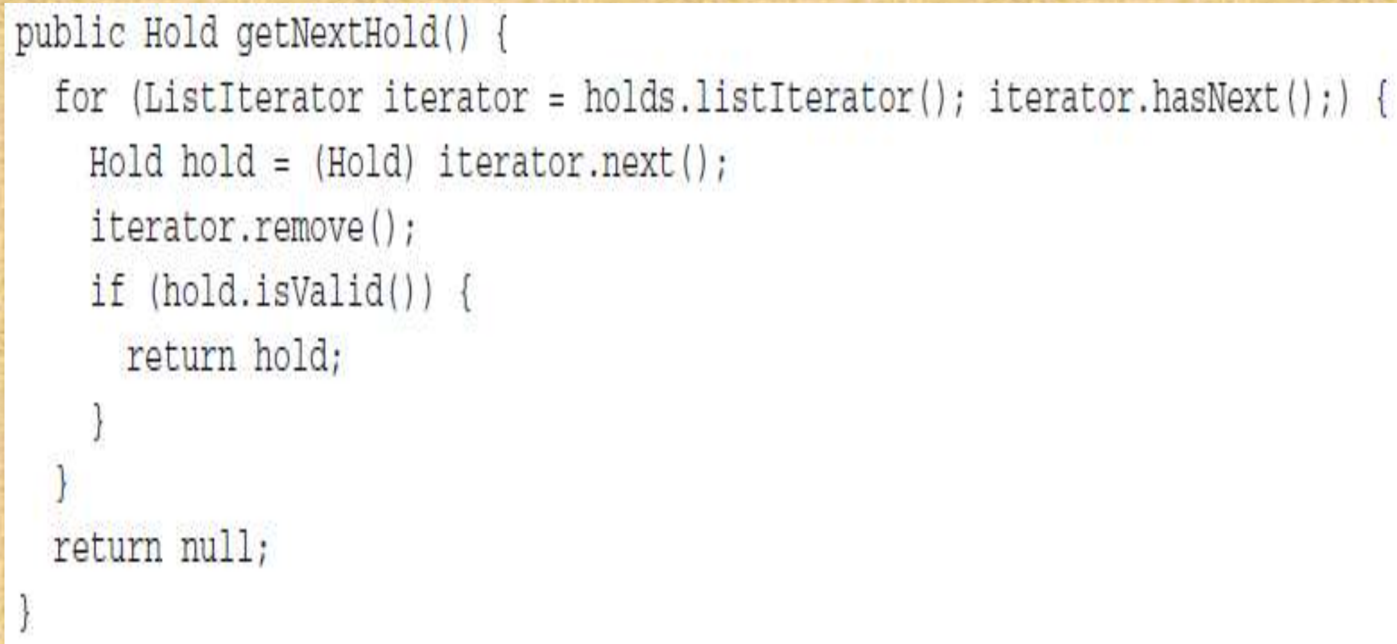
- When placing a hold, the information about the hold is passed to Library, which checks the validity of the information and creates a Hold object



```
private List holds = new LinkedList();
public void placeHold(Hold hold) {
   holds.add(hold);
 }
```

- The problem with this simple solution is that unwanted holds can stay in the system forever.
- To prevent this, we may want to delete all invalid holds periodically, perhaps just before the system is saved to disk.

To process a hold, Library invokes 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;
}
```

Storing and Retrieving the Library Object

Java Serialization

- long-term storage of the library data uses the Java *serialization* mechanism.
- The methods readObject()and writeObject (Object) in ObjectInputStream and ObjectOutputStream respectively can be used to read and write objects and that this can be easily done for simple cases by having the corresponding class implement the Serializable interface.
- The default serialization mechanism in Java does not store static fields.

Module-3

Design Pattern Catalog

Structural Patterns

Design Patterns are of 3 types:

1. Creational : address problems of creating an object in a flexible way, separate creation from operation/use.
2. Structural : address problems of using O-O constructs like inheritance to organize classes and objects.
3. Behavioral: address problems of assigning responsibilities to classes. Suggest both static relationships and patterns of communication (use case)

Types of Pattern

Creational Patterns

(concerned with abstracting the object-instantiation process)

- Factory Method Abstract Factory Singleton
- Builder Prototype

Structural Patterns

(concerned with how objects/classes can be combined to form larger structures)

- Adapter Bridge Composite
- Decorator Facade Flyweight
- Proxy

Behavioral Patterns

(concerned with communication between objects)

- Command Interpreter Iterator
- Mediator Observer State
- Strategy Chain of Responsibility Visitor
- Template Method

Structural Patterns

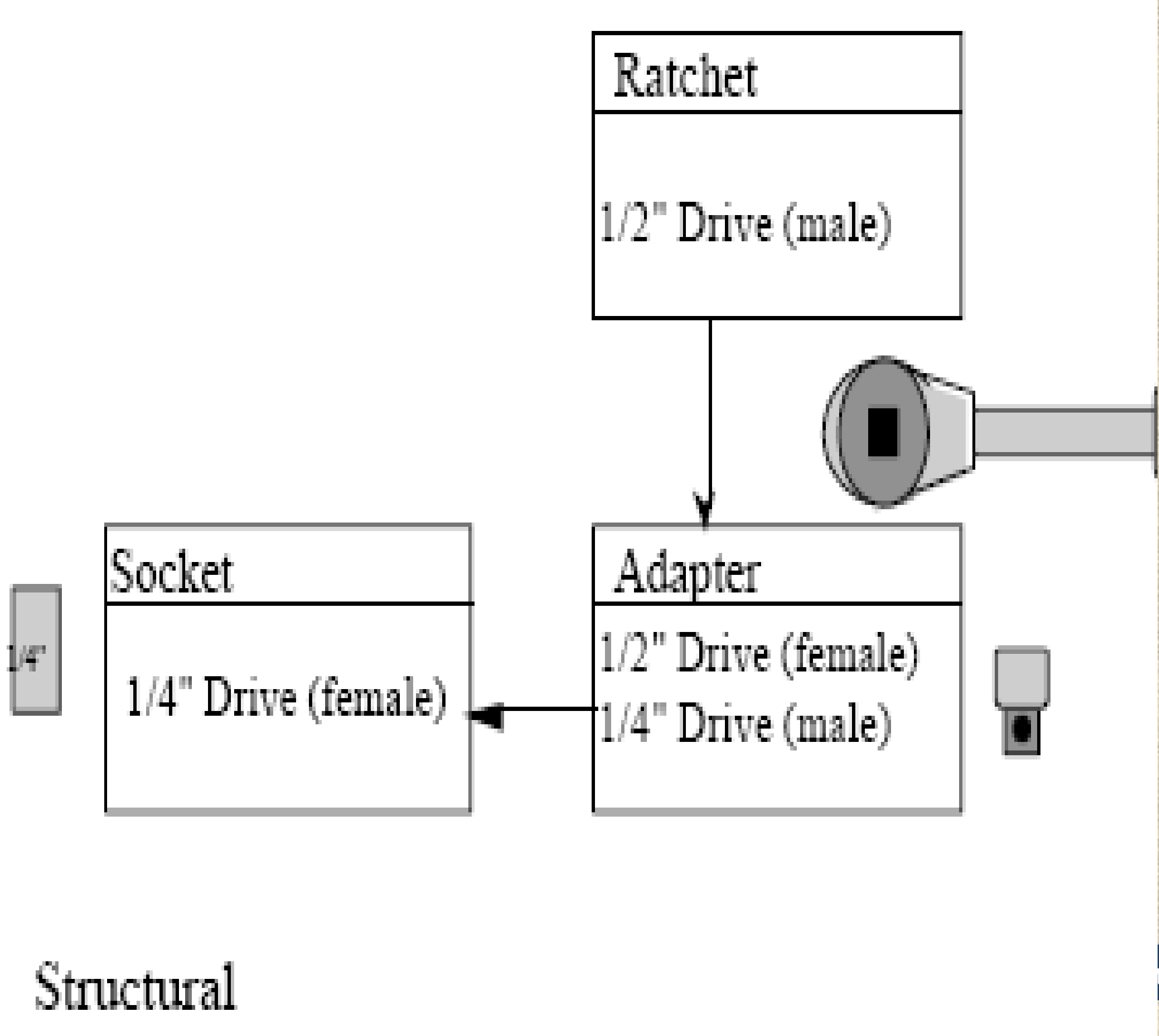
Structural patterns are concerned with how classes and objects are composed to form larger structures.

- Structural class patterns use ***inheritance*** to compose ***interfaces or implementations.***
- This pattern is particularly useful for making ***independently developed class libraries*** work together.
- Structural object patterns ***describe ways to compose objects to realize new functionality.***
- The added flexibility of object composition comes from the ***ability to change the composition at runtime***, which is impossible with static class composition.

A common ways to describe a design pattern is the use of the following template:

1. Pattern Name and Classification
2. Intent
3. Also Known As
4. Motivation (Problem, Context)
5. Applicability (Solution)
6. Structure (a detailed specification of structural aspects)
7. Participants, Collaborations (Dynamics)
8. Implementation
9. Example
10. Known Uses
11. Consequences
12. Related patterns

Adapter (Non software example)



Convert the interface of a class into another Interface clients expect.

Pattern Name: Adapter

Intent:

- Convert the *interface of a class into another interface* clients expect. Adapter lets *classes work together* that could not otherwise because of incompatible interfaces.

Collaborations:

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

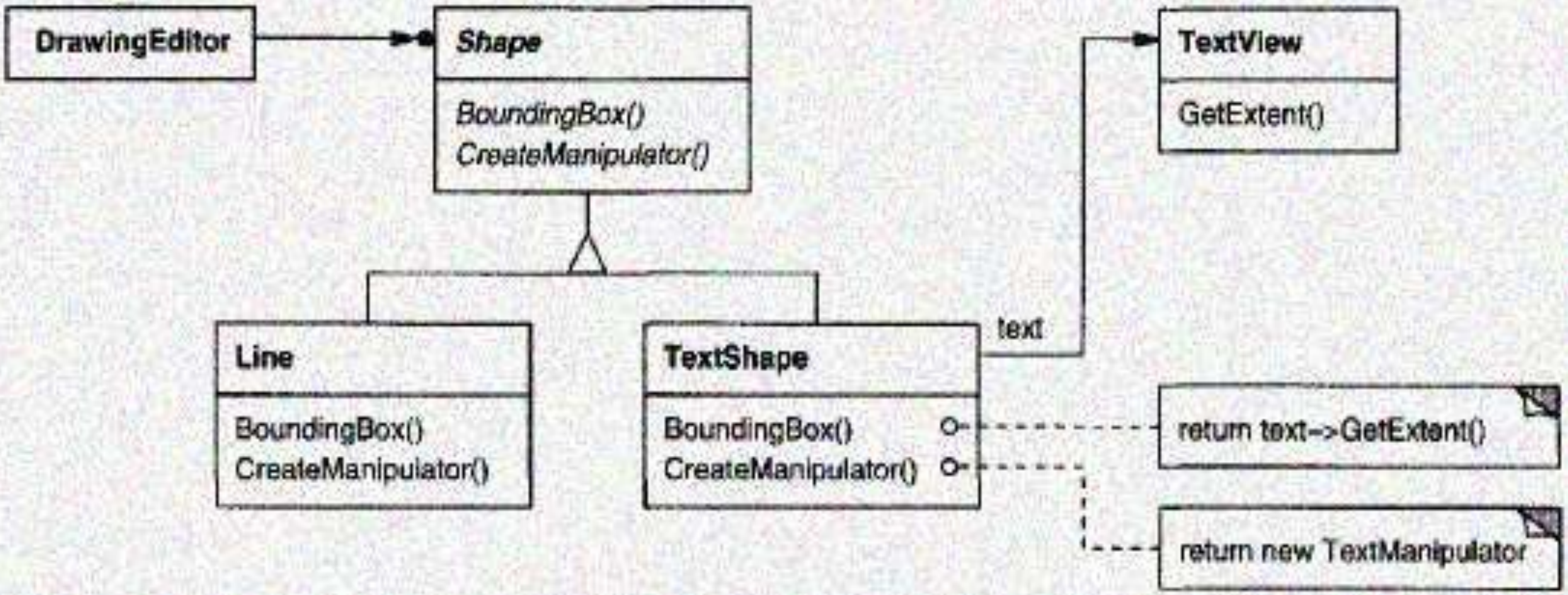
Also Known As: *Wrapper*

Adapter…
Motivation:

- Sometimes a *toolkit class* that's designed for reuse *isn't reusable* only because its *interface doesn't match the domain-specific interface an application requires.*
- Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams.
- The *interface for graphical objects* is defined by an *abstract class* called *Shape.*
- The editor defines a *subclass of Shape for each kind of graphical object.*
- Classes for elementary geometric shapes are easy to implement. But a **TextShape** subclass that can display and edit text is more difficult to implement. (screen updates, buffer updates)

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

- We could define **TextShape** so that it adapts the **TextView** interface to Shape's.
  1. by inheriting Shape's interface and TextView's implementation or
  2. by composing a TextView instance within a TextShape and implementing TextShape in terms of TextView's interface.



- The two approaches correspond to the *class and object versions of the Adapter pattern.* We call ***TextShape an adapter.***
- We can see how *BoundingBox req*uests, declared in *class Shape*, are converted to *GetExtent requests defined in TextView.*
- Since *TextShape adapts TextView* to the *Shape interface*, the *drawing editor* can reuse the otherwise incompatible TextView class.

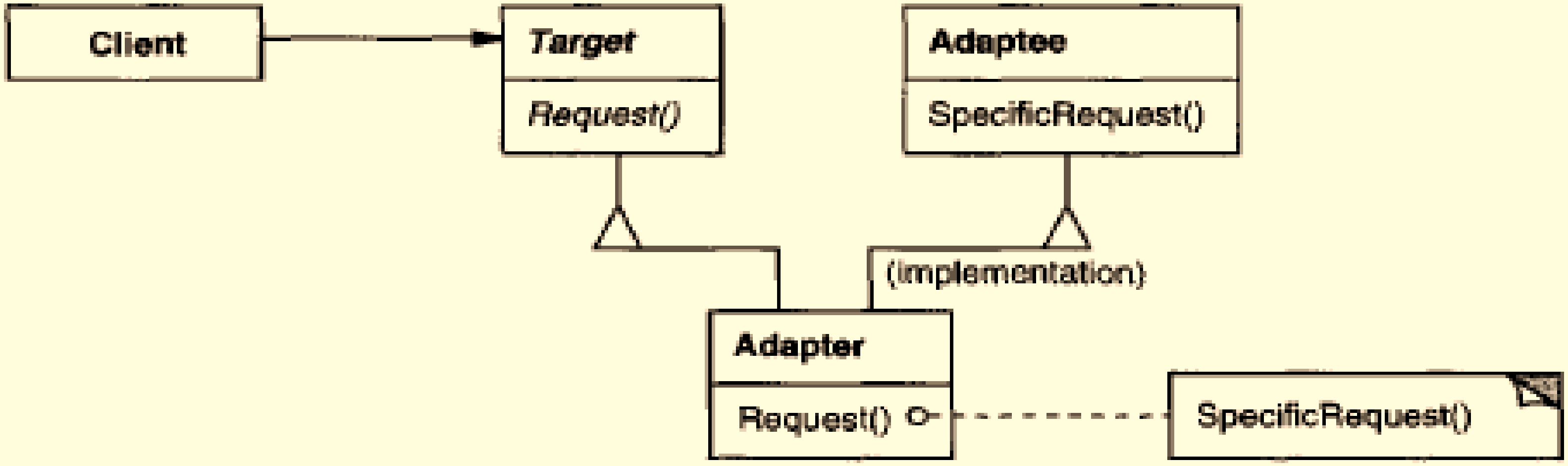
Applicability:

Use the Adapter pattern when

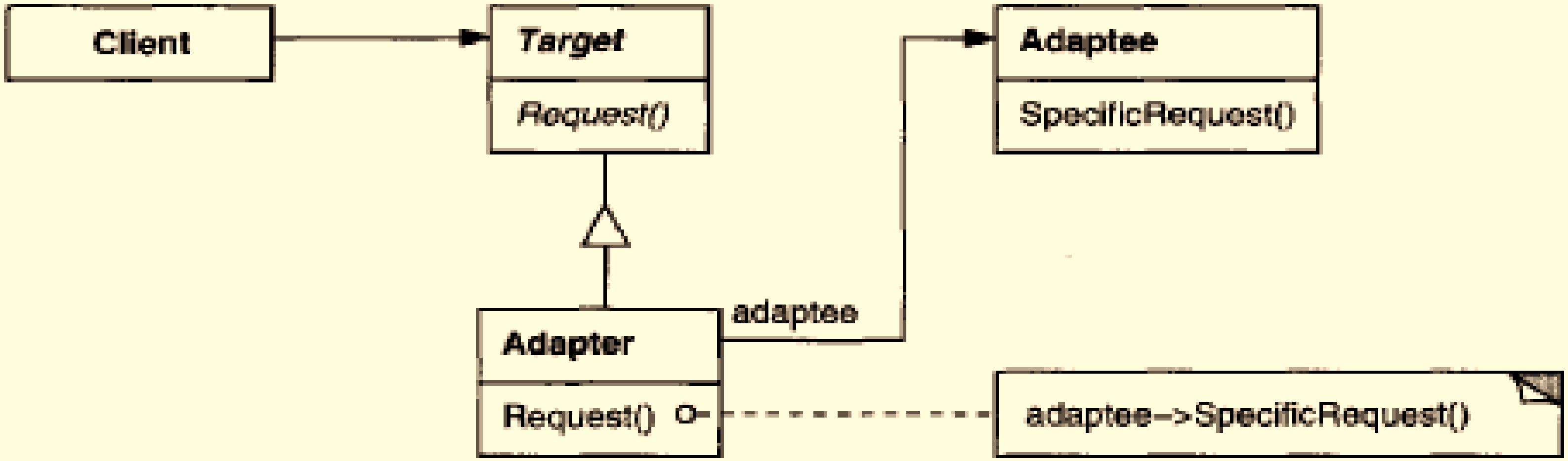
- ✓ you want to ***use an existing class***, and ***its interface does not match the one you need.***
- ✓ you ***want to create a reusable class*** that ***cooperates with unrelated or unforeseen classes,*** that is, ***classes that don't necessarily have compatible interfaces.***
- ✓ (object adapter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

Structure

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



An object adapter relies on object composition:



Participants

- **Target (Shape)**

- defines the domain-specific interface that Client uses.

- **Client (DrawingEditor)**

- Collaborates with objects conforming to the Target interface.

- **Adaptee (TextView)**

- defines an existing interface that needs adapting.

- **Adapter (TextShape)**

- adapts the interface of Adaptee to the Target interface.

Collaborations

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

Consequences

Class and object adapters have different trade-offs.

❖ *A class adapter*

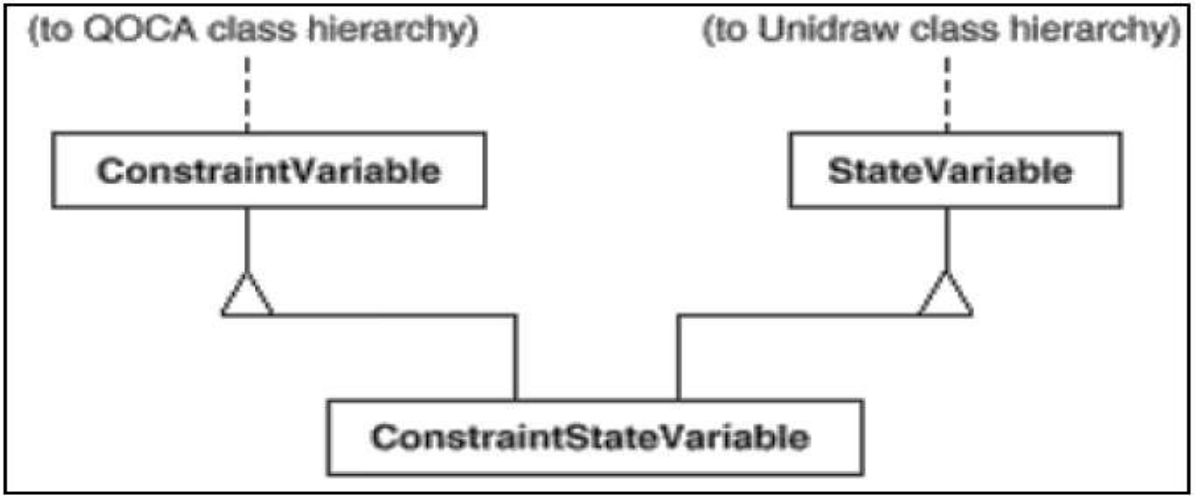
- adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class *and* all its subclasses.
- lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

❖ *An object adapter*

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

Issues to be consider when using the Adapter pattern:

- **How much adapting does Adapter do?**
- **Pluggable adapters: describe classes with built-in interface adaptation.**
- **Using two-way adapters to provide transparency.**



Implementation:

Issues to keep in mind while implementing Adapter:

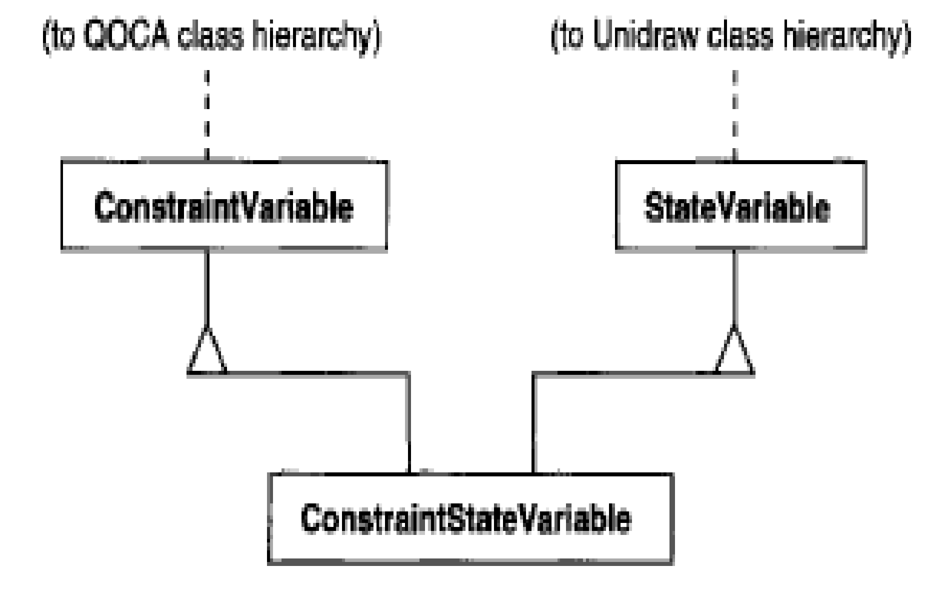
1. **Implementing class adapters in C++.** Adapter would inherit publicly from Target and privately from Adaptee.
   - Thus adapter would be subtype of target but not of Adaptee

2. **Pluggable adapters.**

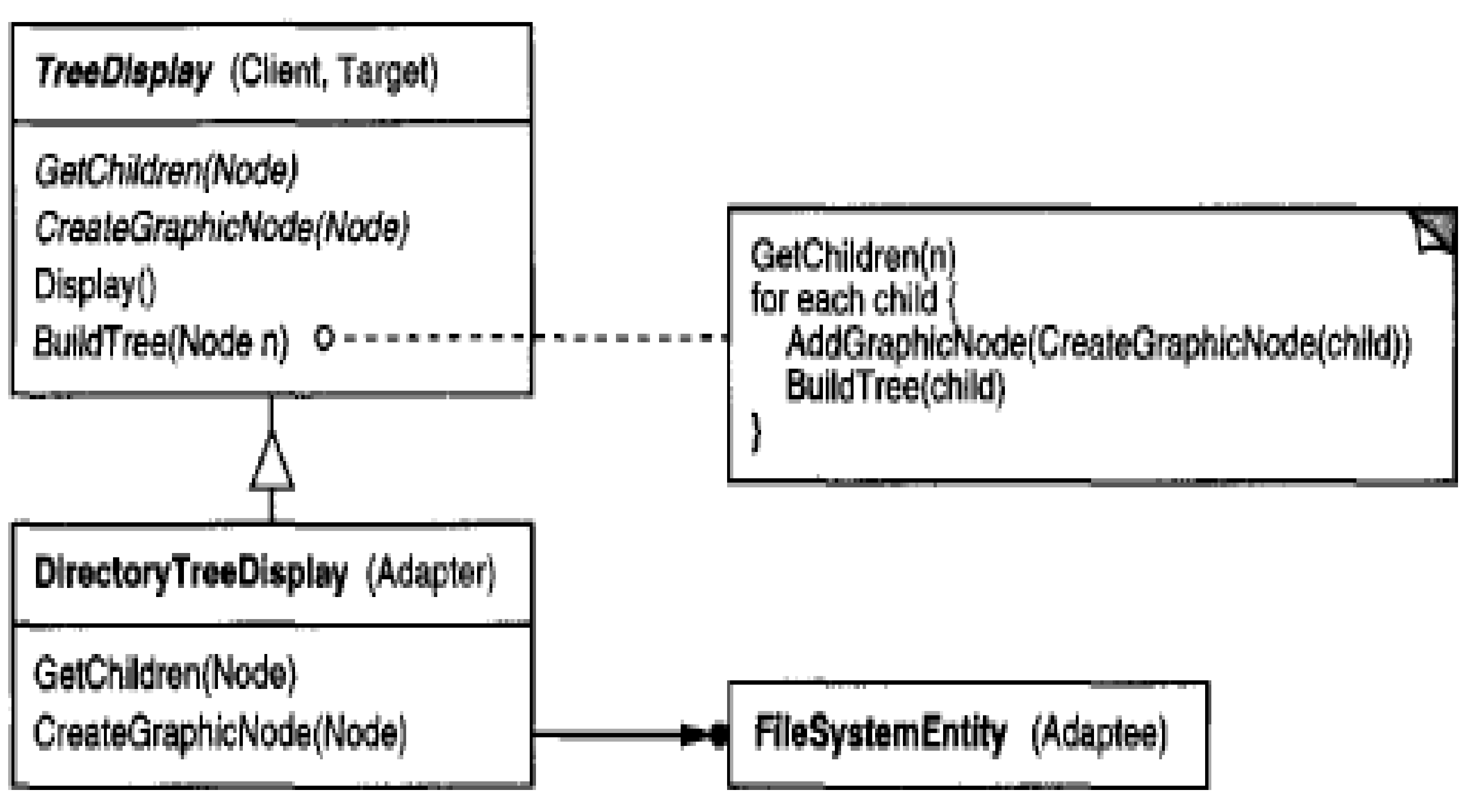
find a "narrow" interface for Adaptee the smallest subset of operations that lets us do the adaptation.

The ***narrow interface leads to three implementation approaches***:

a. Using abstract operations.
b. Using delegate objects.
c. Parameterized adapters.

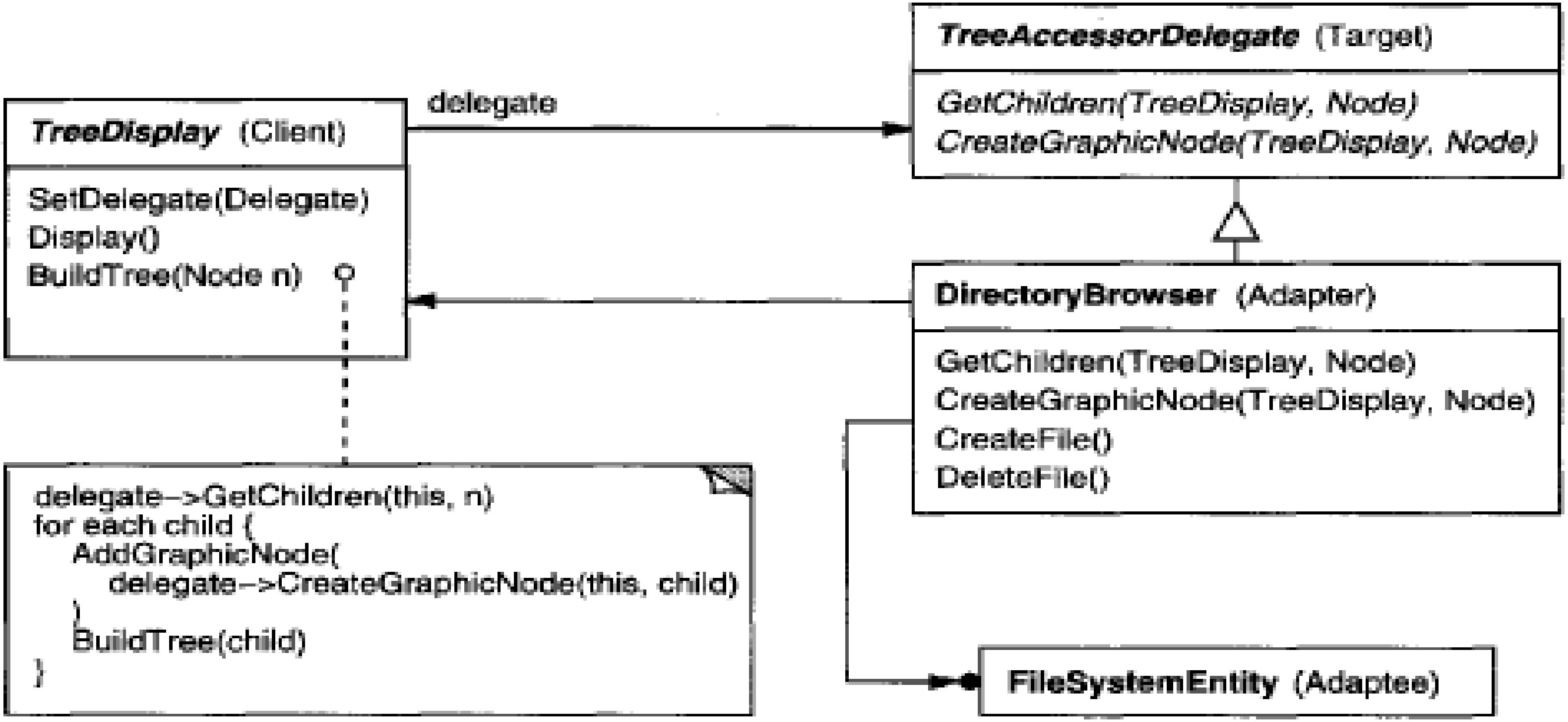


a. Using abstract operations.



```
GetChildren(n)
for each child {
    AddGraphicNode(CreateGraphicNode(child))
    BuildTree(child)
}
```

b. Using delegate objects.

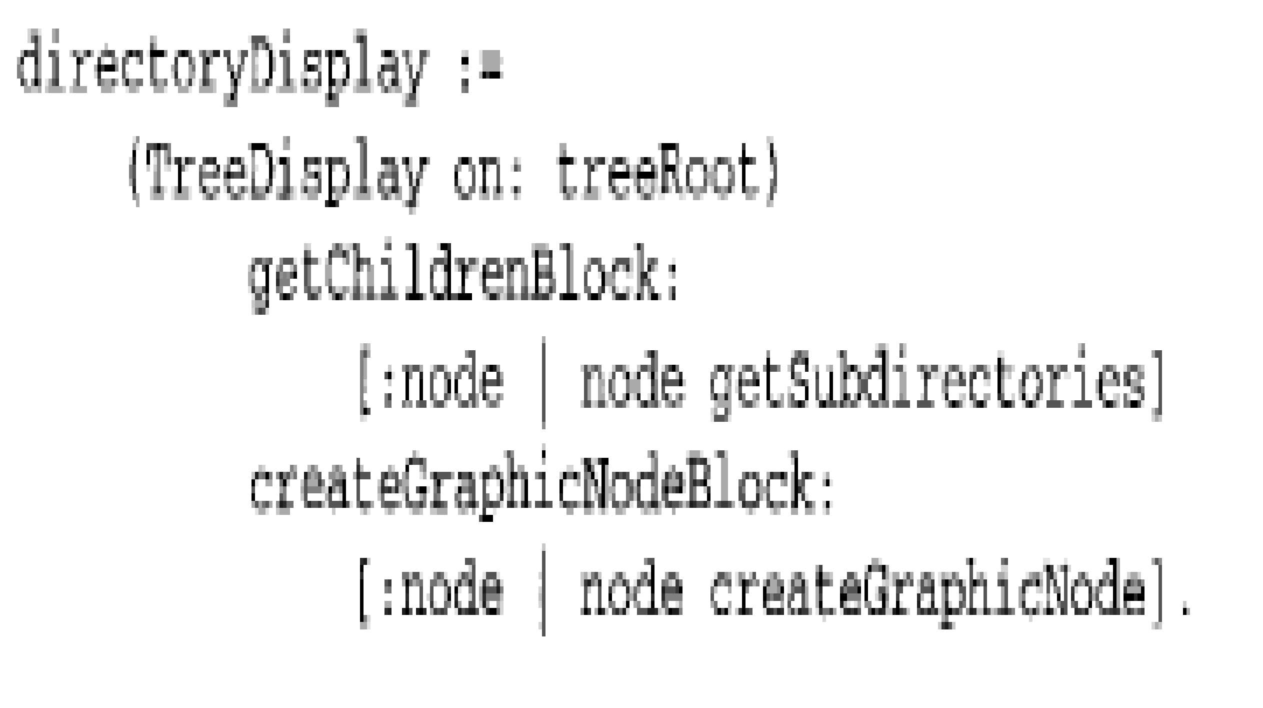


TreeDisplay (Client)
SetDelegate(Delegate)
Display()
BuildTree(Node n)
delegate
TreeAccessorDelegate (Target)
GetChildren(TreeDisplay, Node)
CreateGraphicNode(TreeDisplay, Node)
DirectoryBrowser (Adapter)
GetChildren(TreeDisplay, Node)
CreateGraphicNode(TreeDisplay, Node)
CreateFile()
DeleteFile()
FileSystemEntity (Adaptee)

```
delegate->GetChildren(this, n)
for each child {
    AddGraphicNode(
        delegate->CreateGraphicNode(this, child)
    )
    BuildTree(child)
}
```

c. Parameterized adapters

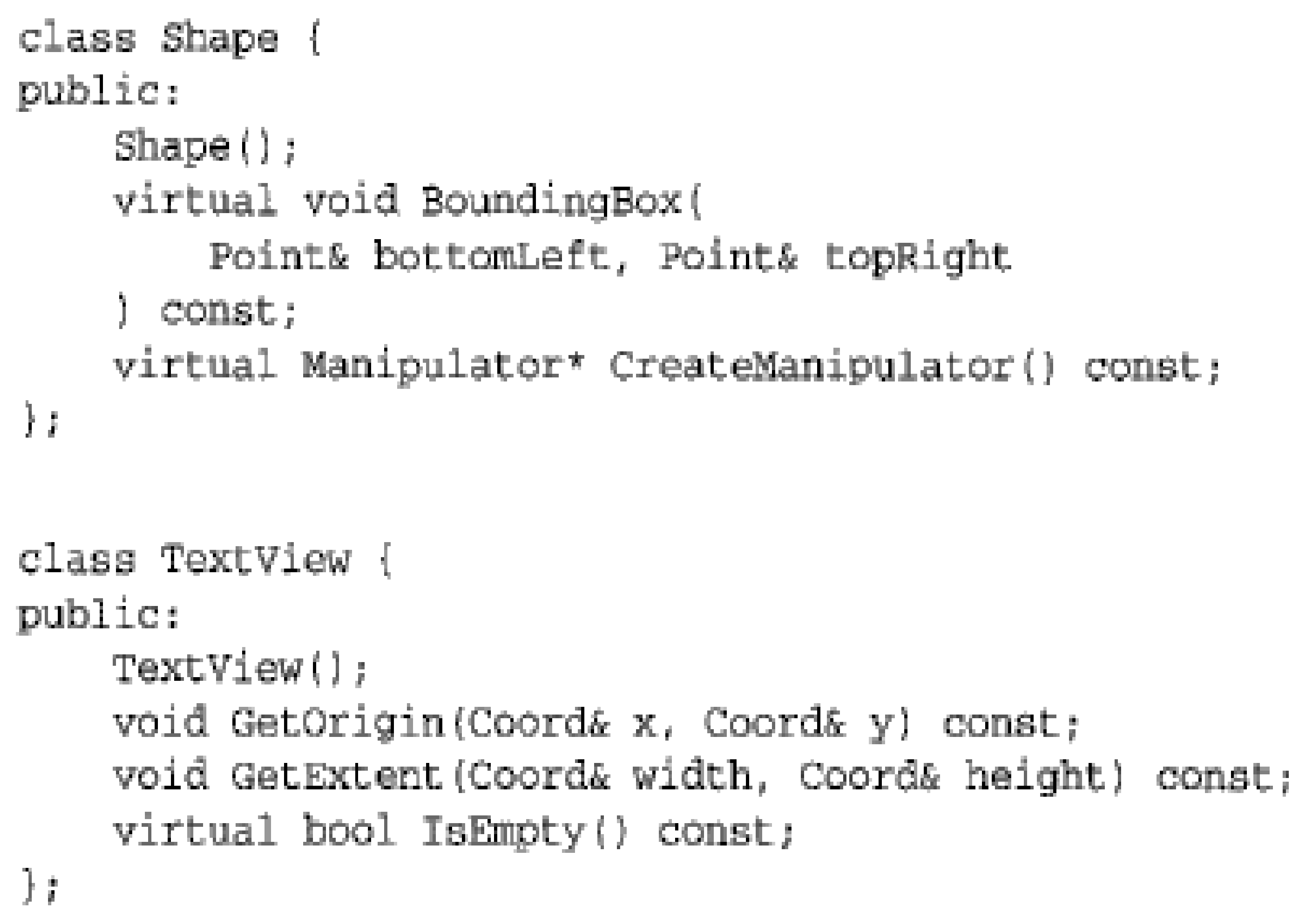
- The usual way to support pluggable adapters in Smalltalk is to parameterize an adapter with one or more blocks.
- The block construct supports adaptation without subclassing.
- A block can adapt a request, and the adapter can store a block for each individual request.



```
directoryDisplay :=
    (TreeDisplay on: treeRoot)
        getChildrenBlock:
            [:node | node getSubdirectories]
        createGraphicNodeBlock:
            [:node | node createGraphicNode].
```

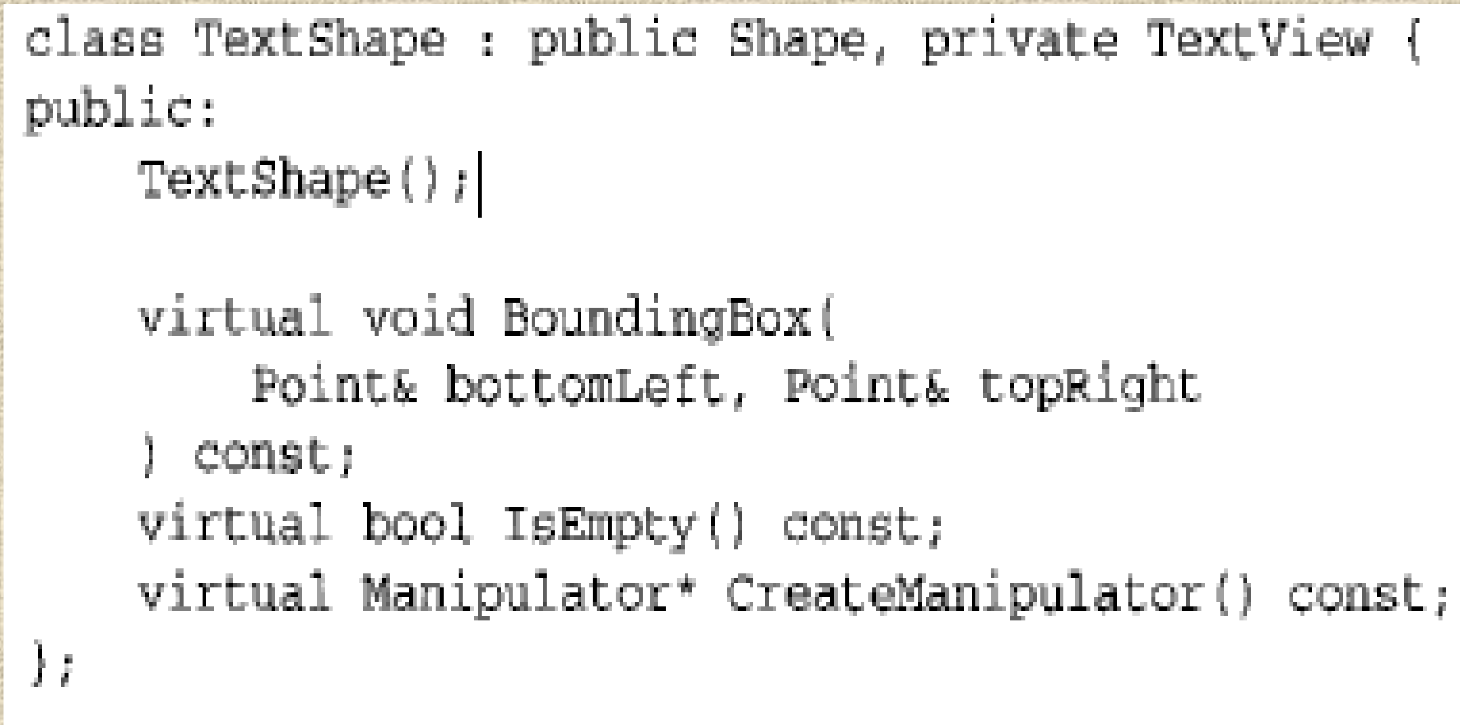
Sample Code

A brief sketch of the implementation of class and object adapters for *the Shape and TextView*.



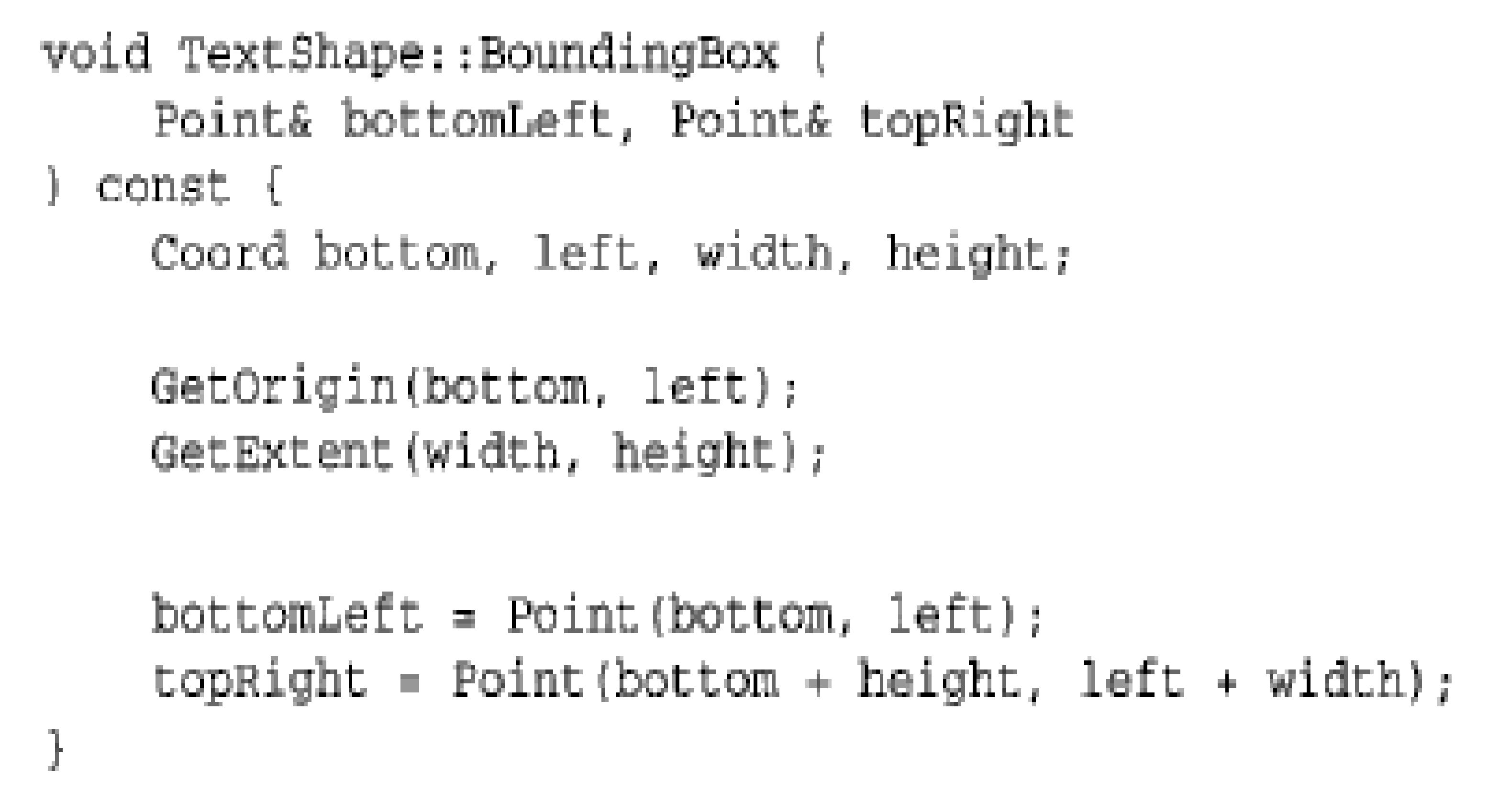
```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};


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

- **A class adapter uses multiple inheritance to adapt interfaces.**
- **The key to class adapters is to use one inheritance branch to inherit the interface and another branch to inherit the implementation.**
- **The usual way to make this distinction in C++ is to *inherit the interface publicly* and *inherit the implementation privately.***
- **We'll use this convention to define the Text Shape adapter.**



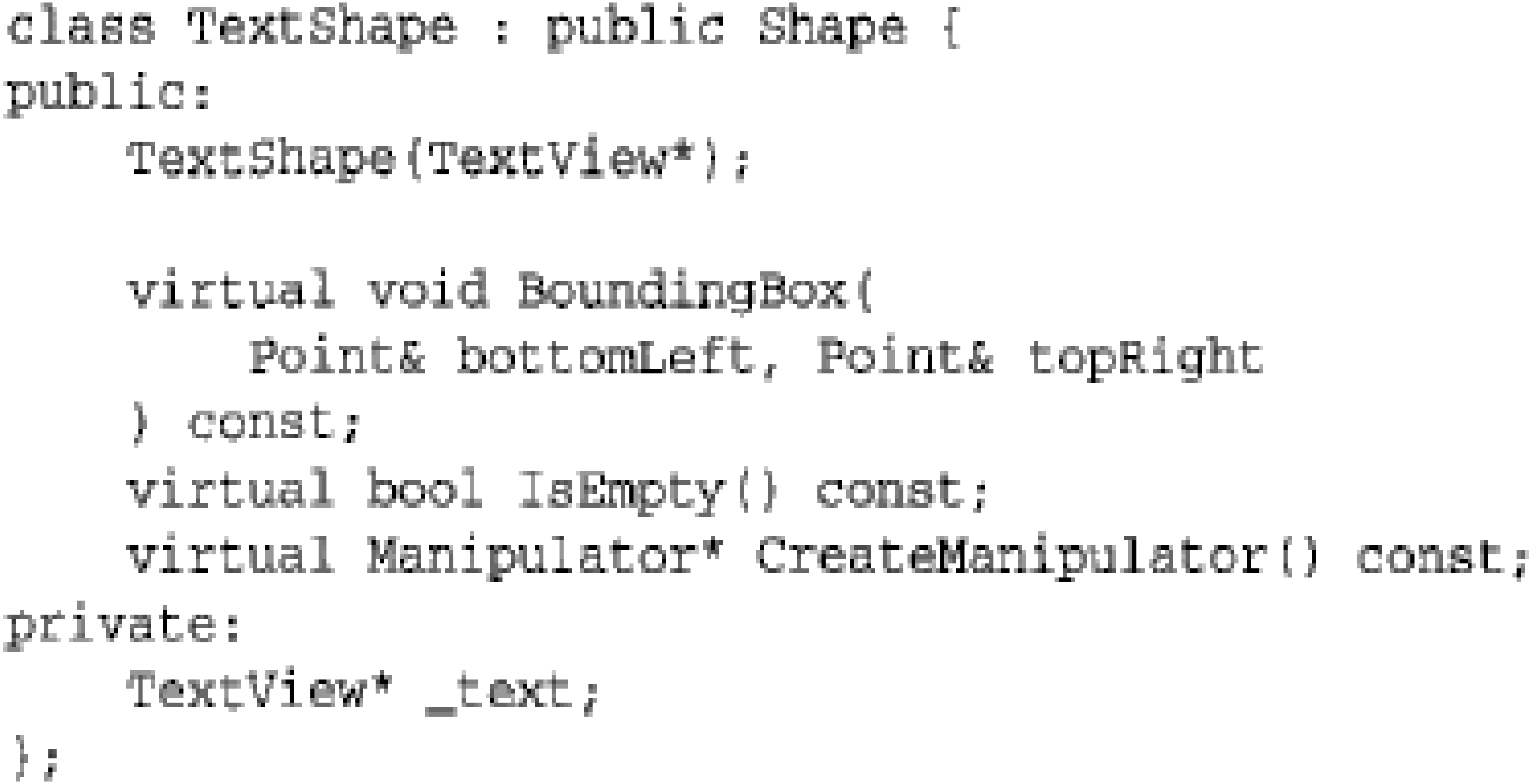
```
class TextShape : public Shape, private TextView {
public:
    TextShape();

    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.



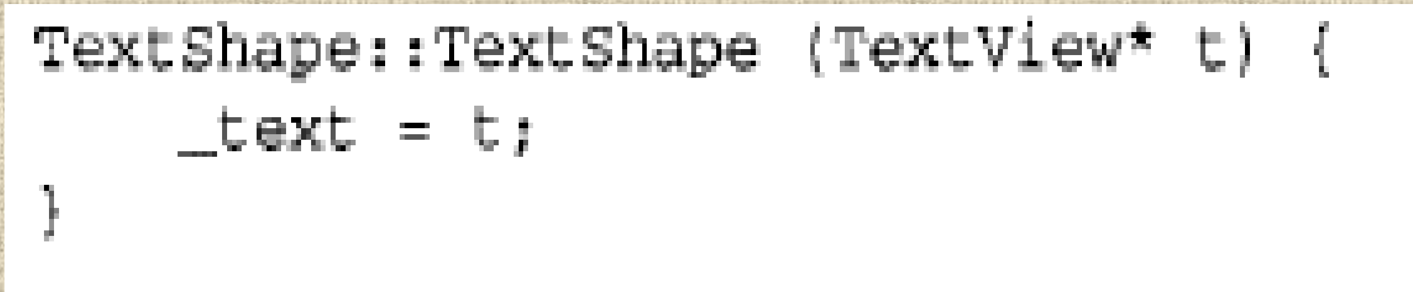
```
void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

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


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

The object adapt uses object composition to combine classes with different interfaces. In this approach, the adapterText Shape maintains a pointer to TextView.

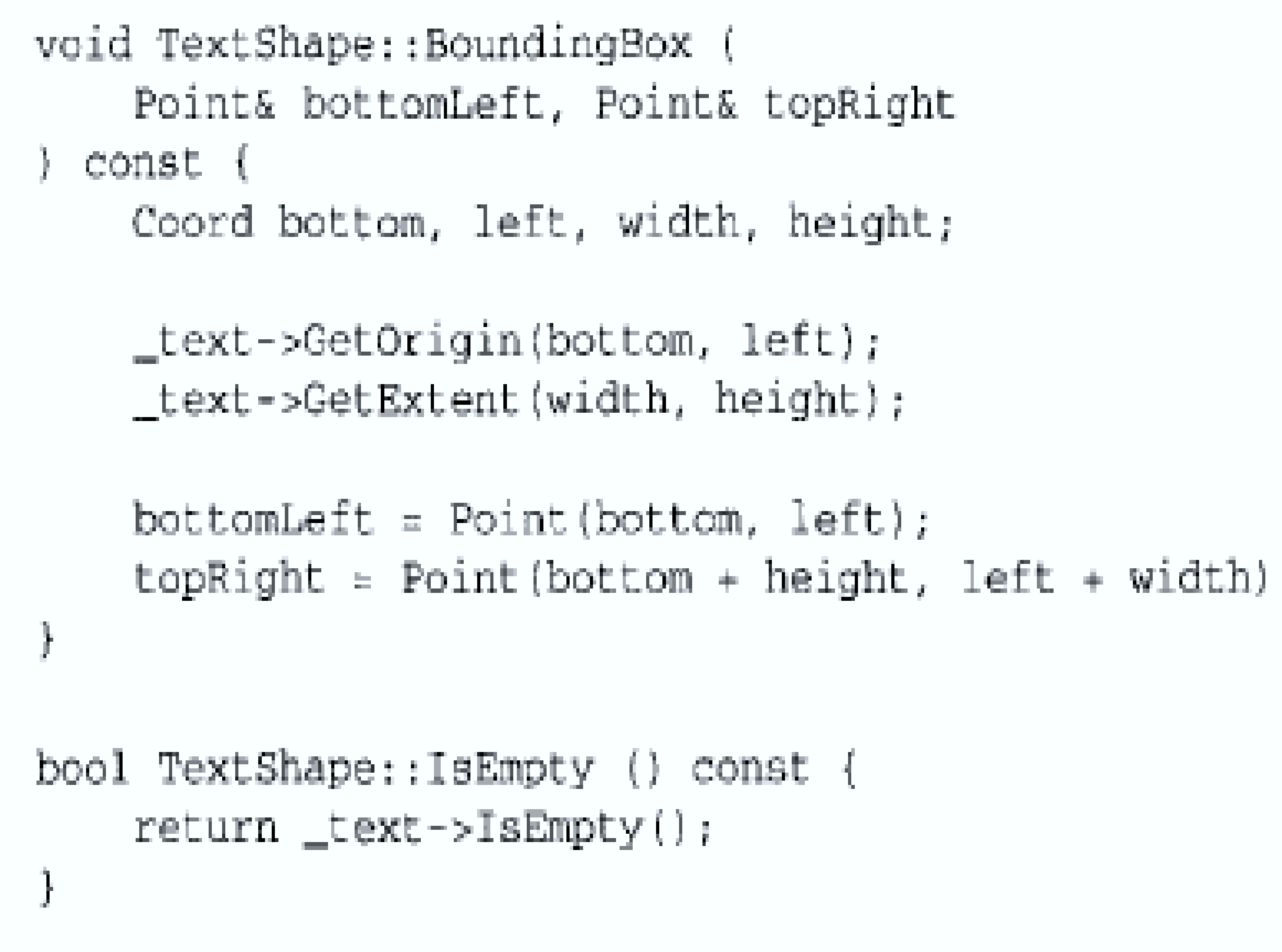


```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    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.
- It must also call operations on its TextView object whenever its own operations are called. In this example, assume that the client creates the TextView object and passes it to the TextShape 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

This pattern is used in the following toolkits:

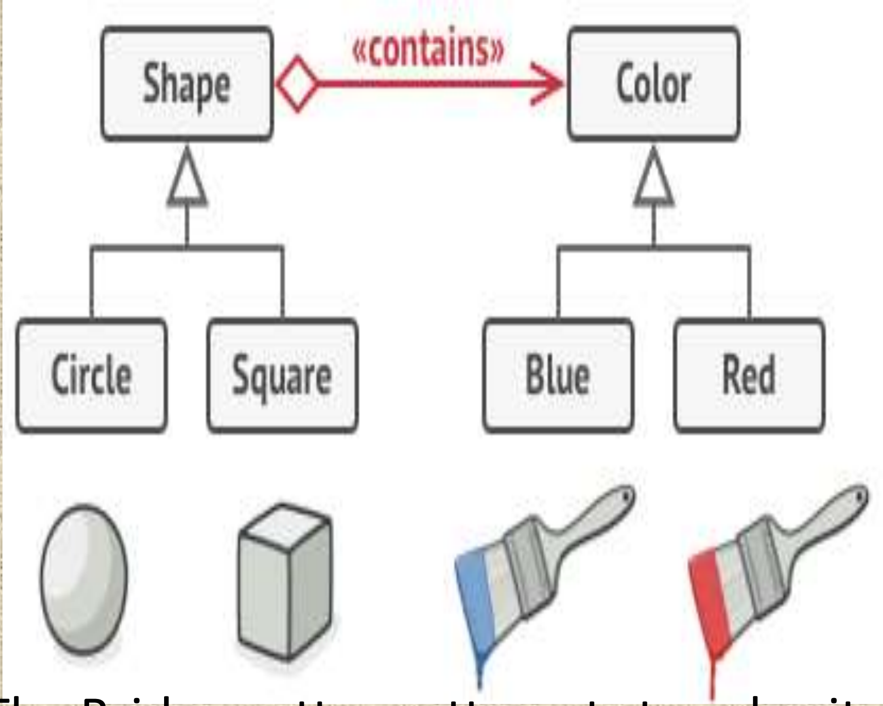
- ET++Draw,
- InterViews 2.6,
- ObjectWorks\Smalltalk,
- NeXT'sAppKit

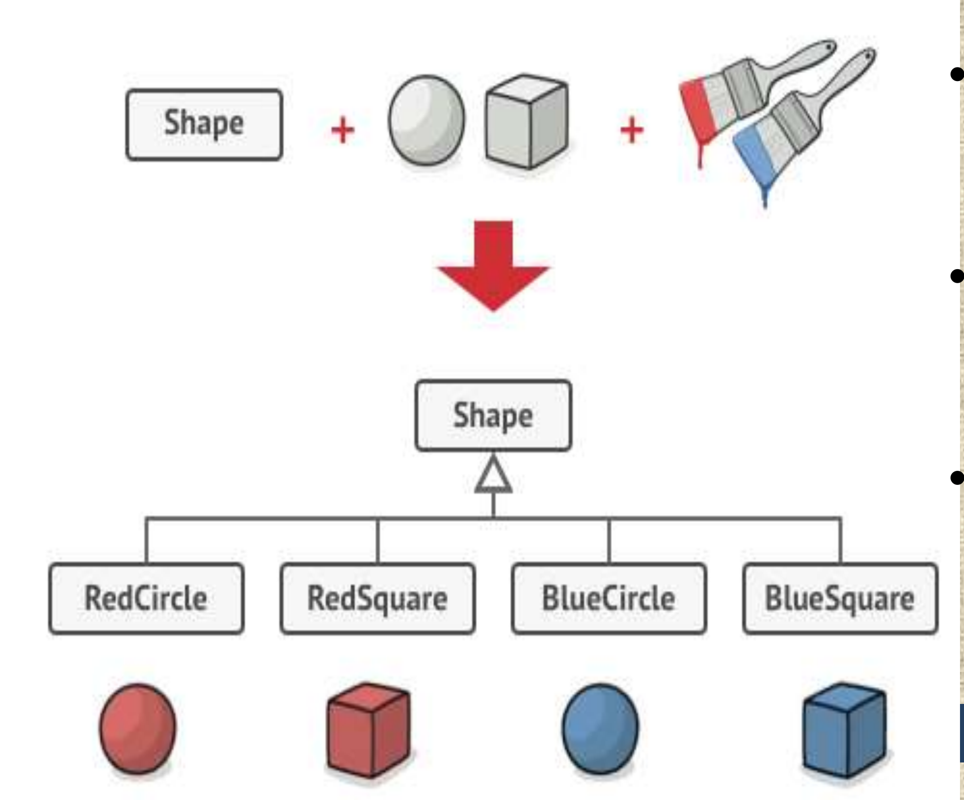
Related Patterns:

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

Pattern Name: BRIDGE

- Lets you split a giant class or a set of closely related classes into two separate hierarchies, abstraction and implementation,
- which can be developed independently of each other.





- The Bridge pattern attempts to solve it by replacing inheritance with delegations.
- You have to extract one of these "dimensions" into separate class hierarchy.
- Original classes will contain a reference to an object of the new hierarchy, instead of storing all of its state and behaviors inside of one class.

Pattern Name: BRIDGE

Intent:

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

Also Known As:

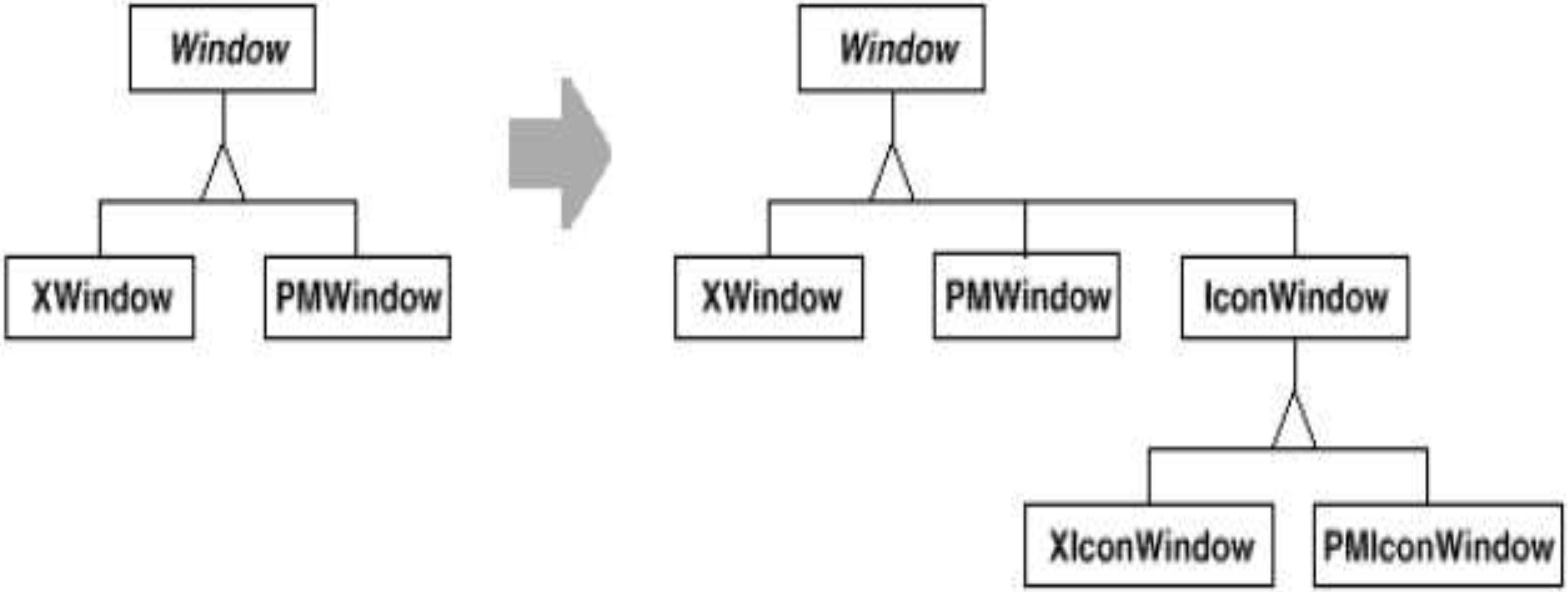
Handle/Body

Use the Bridge pattern when:

- You want run-time binding of the implementation
- You want to share an implementation among multiple objects

Motivation:

- When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.
- An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.
- But this approach isn't always flexible enough.
- Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently.



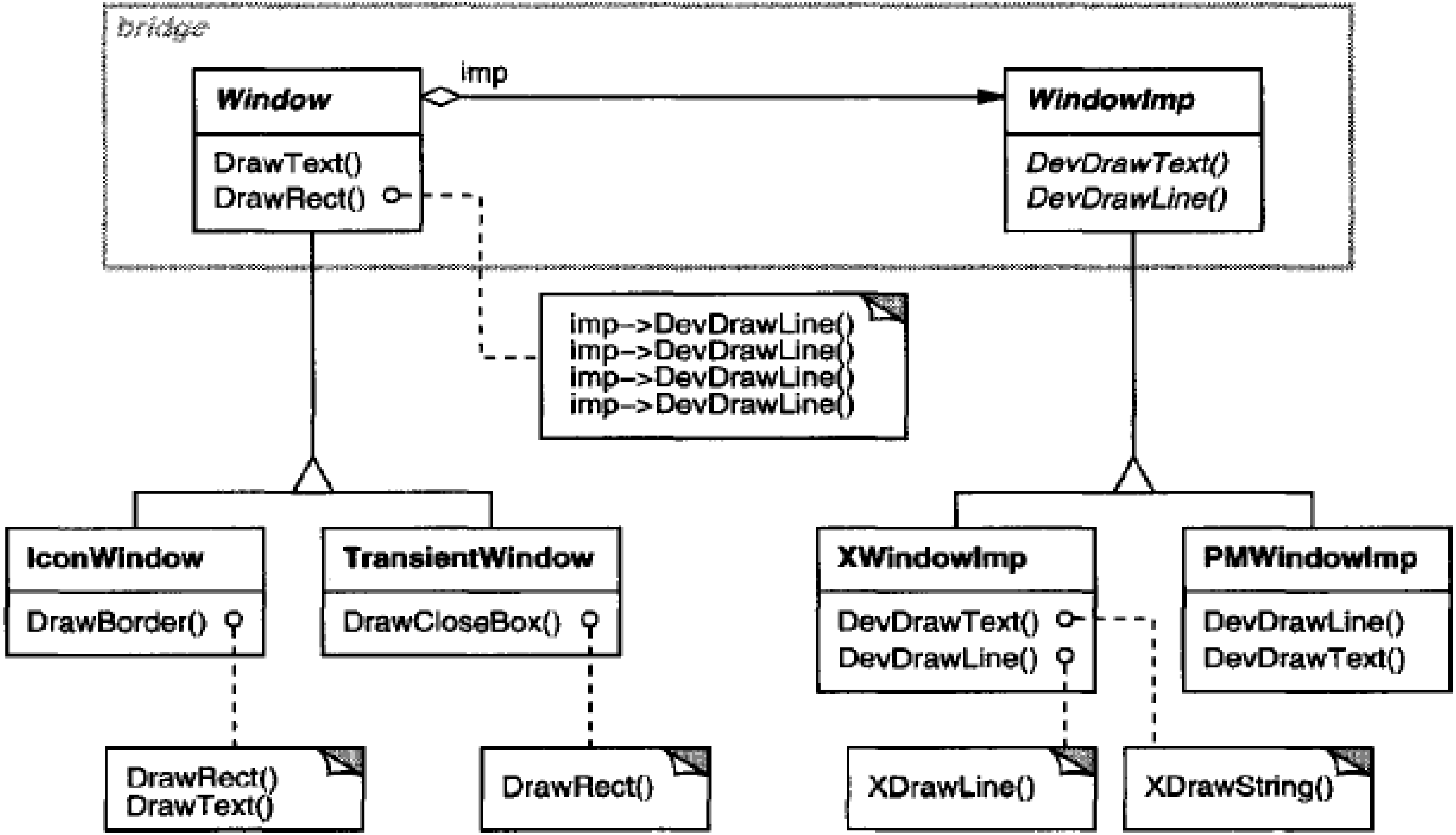
❖ But this approach has two drawbacks:

1. It's inconvenient to extend the Window abstraction to cover different kinds of windows or new platforms.

➢ But we'll have to define new classes for *every* kind of window. Supporting

2. It makes client code platform-dependent.ie., 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.*

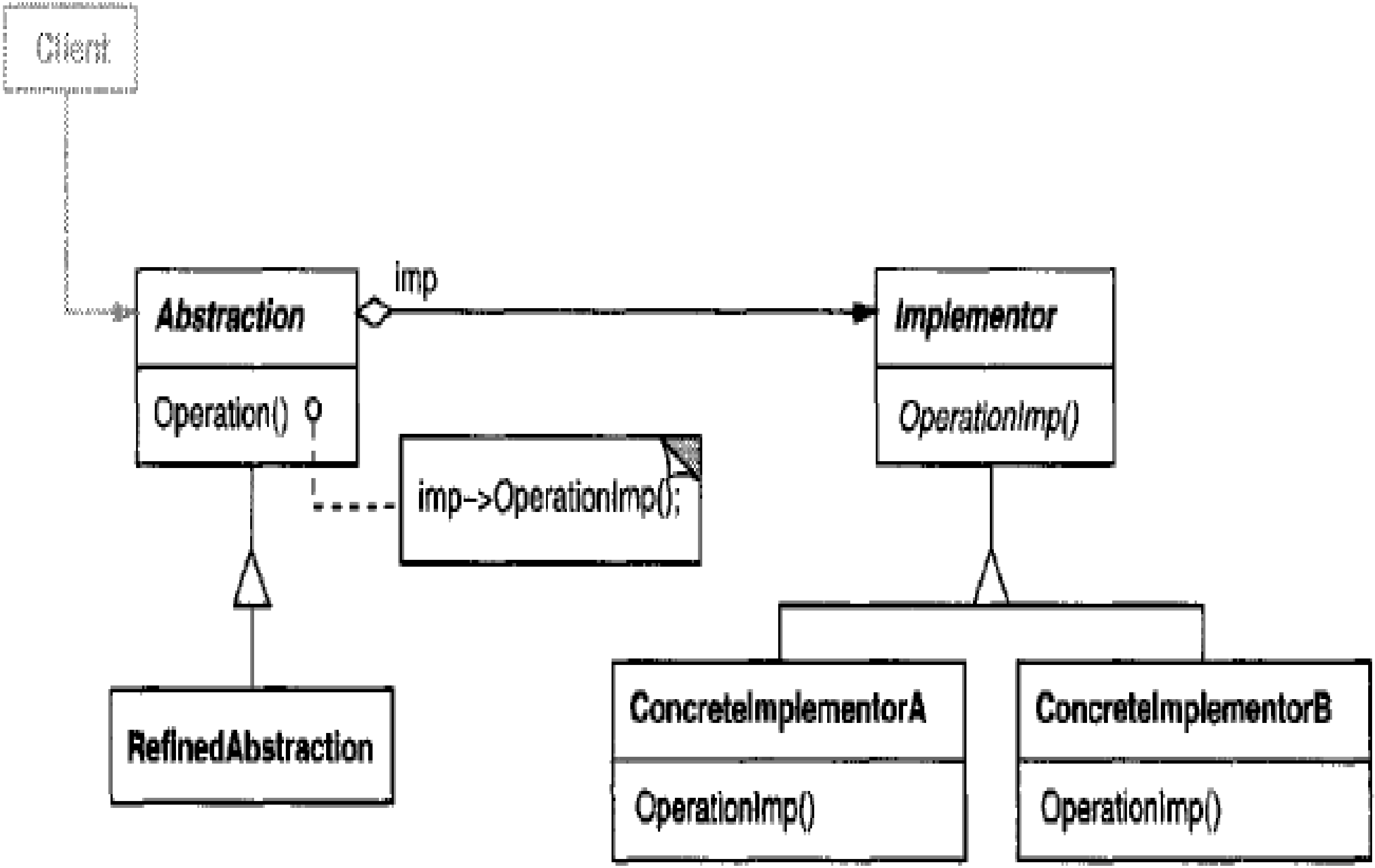


Applicability:

Use the Bridge pattern when

- you want to avoid a permanent binding between an abstraction and its implementation.
- both the abstractions and their implementations should be extensible by subclassing.
- changes in the implementation of an abstraction should have no impact on clients; their code should not have to be recompiled
- (C++) you want to hide the implementation of an abstraction completely from clients.
- you have a proliferation of classes as shown earlier in the first Motivation diagram:- need to use nested generalization
- you want to share an implementation among multiple objects, and this fact should be hidden from the client.

Structure:


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

Participants :

- Abstraction (Window)
  - ➢ defines the abstraction's interface.
  - ➢ maintains a reference to an object of type Implementor.
- RefinedAbstraction (IconWindow)

  - Extends the interface defined by Abstraction.
- Implementor (WindowImp)
  - ➢ defines the interface for implementation classes.
  - ➢ This interface doesn't have to correspond exactly to Abstraction's interface;
  - ➢ in fact the two interfaces can be quite different.
  - ➢ Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- ConcreteImplementor (XWindowImp, PMWindowImp)
  - ➢ implements the Implementor interface and defines its concrete implementation.

Collaborations :

- Abstraction forwards client requests to its Implementor object.

Consequences:

The Bridge pattern has the following consequences:

1. *Decoupling interface and implementation.*
   - An implementation is not bound permanently to an interface.
   - The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.
   - Decoupling Abstraction and Implementor eliminates compile-time dependencies on the implementation. Does not requires recompiling.

Consequences: contd…

2. *Improved extensibility.*
   - You can extend the Abstraction and Implementor hierarchies independently.
3. *Hiding implementation details from clients.*
   - You can shield clients from implementation details, like the sharing of implementor objects and the accompanying reference count mechanism (if any).

Implementation:

Consider the following implementation issues when applying the Bridge pattern:

1. ***Only one Implementor.***
   - 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.
   - a change in the implementation o f a class must not affect its existing clients— that is, they shouldn't have to be recompiled, just relinked.

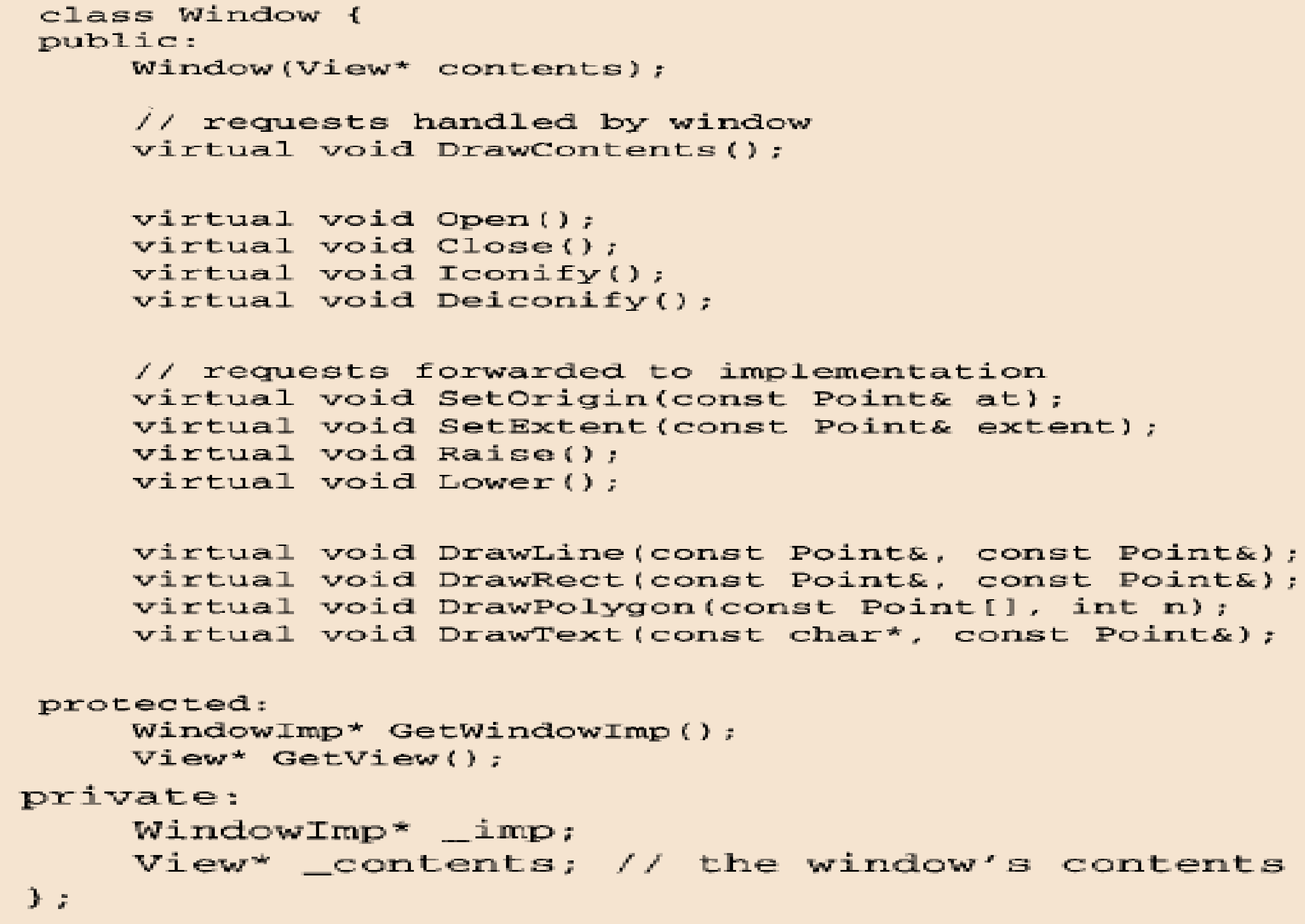
2. *Creating the right Implementor object.*

How, when, and where do you decide which Implementor class to instantiate when there's more than one?

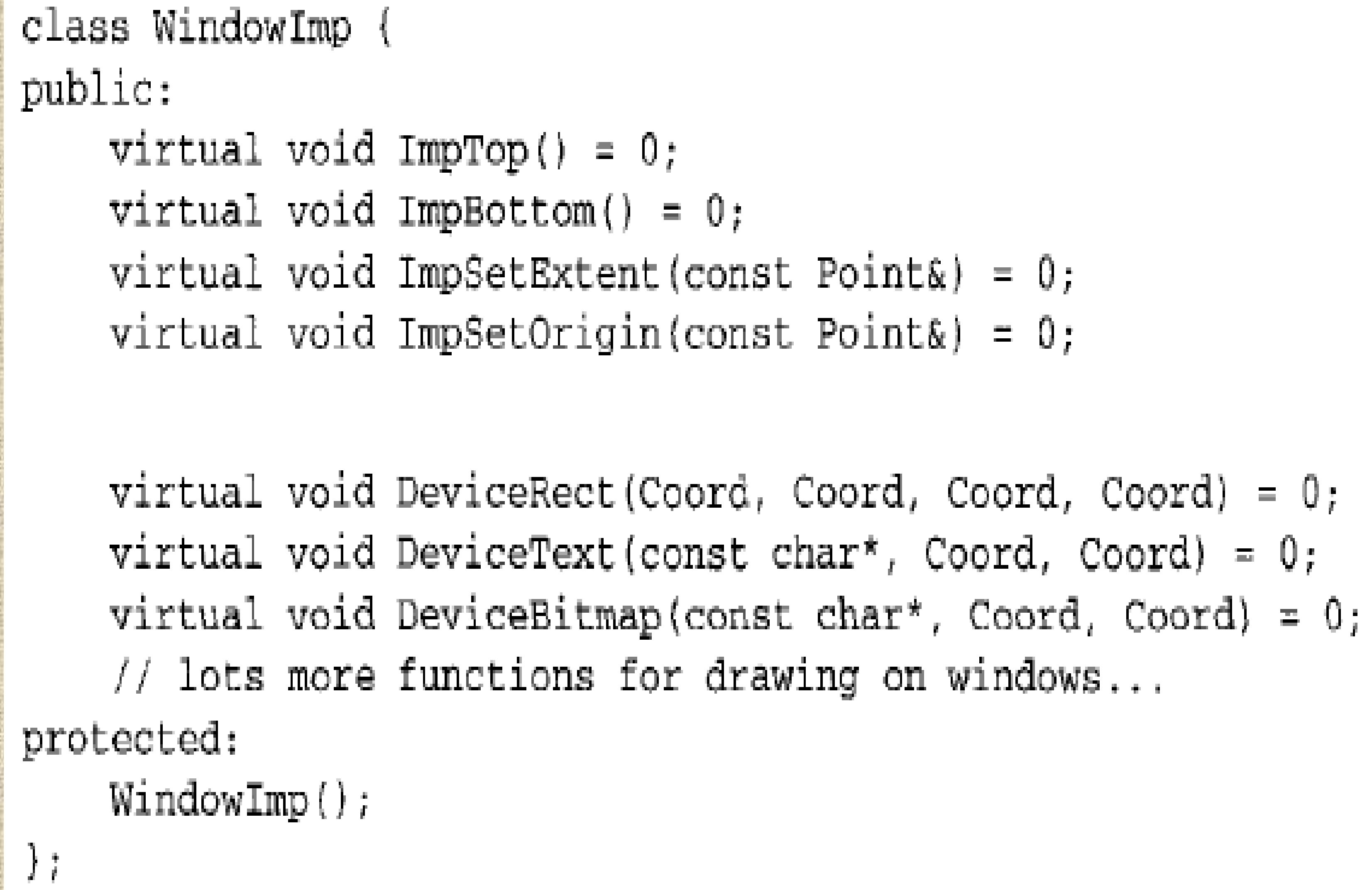
- If Abstraction knows about all ConcreteImplementor classes, - it can decide between them based on parameters passed to its constructor.
- Example: A collection class supports multiple implementations based on size of the collection particular implementation is called
- A linked list implementation can be used for small collections and a hash table for larger ones.
- Another approach is to choose a default implementation initially and change it later according to usage.
- It's also possible to delegate the decision to another object altogether.

3. ***Sharing implementors.***
    - share implementations among several objects
    - The Body stores a reference count that the Handle class increments and decrements.
4. ***Using multiple inheritance.***
    - You can use multiple inheritance in C++ to combine an interface with its implementation.
    - A class can inherit publicly from Abstraction and privately from a ConcreteImplementor.
    - But because this approach relies on static inheritance, it binds an implementation permanently to its interface. Therefore you can't implement a true Bridge with multiple inheritance—at least not in C++

Sample Code:

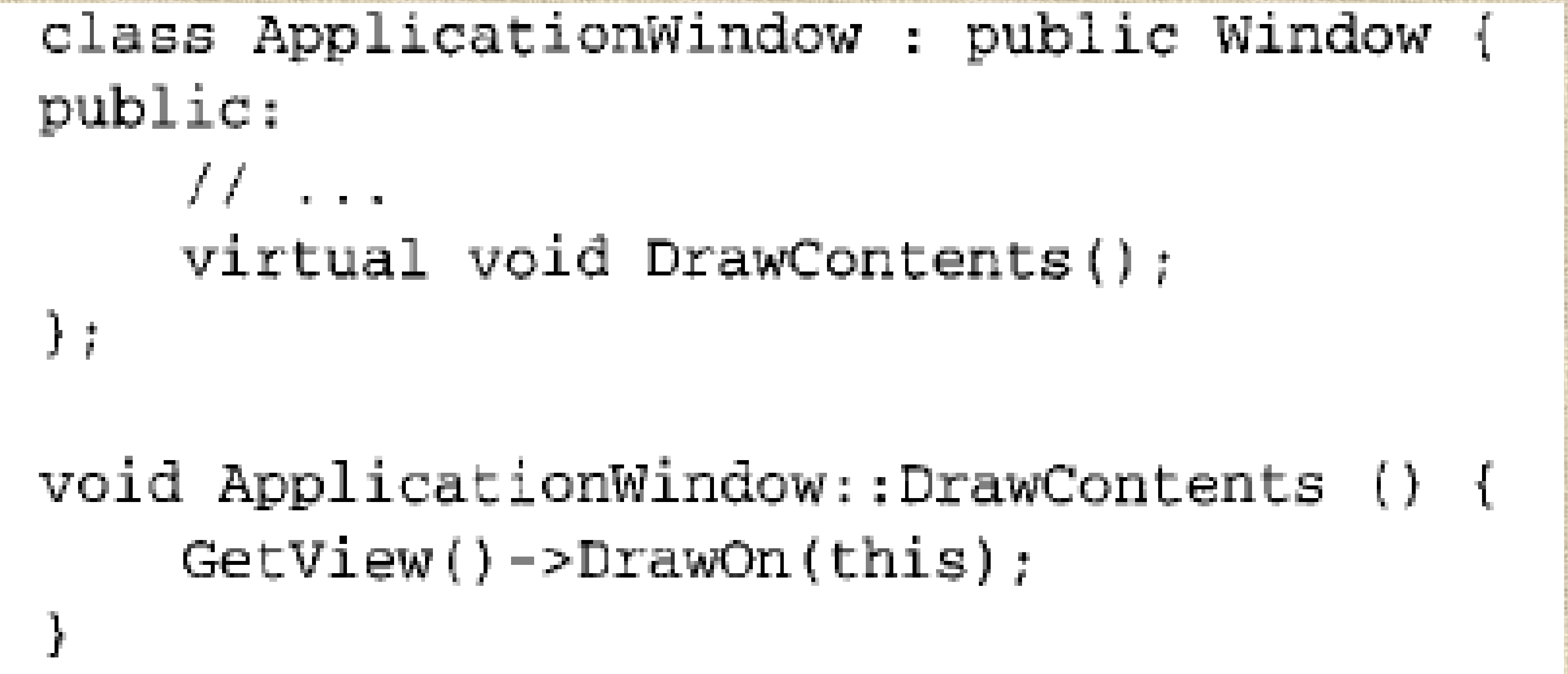


```
class Window {
public:
     Window(View* contents);

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

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

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

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

protected:
     WindowImp* GetWindowImp();
     View* GetView();
private:
     WindowImp* _imp;
     View* _contents; // the window's contents
};
```



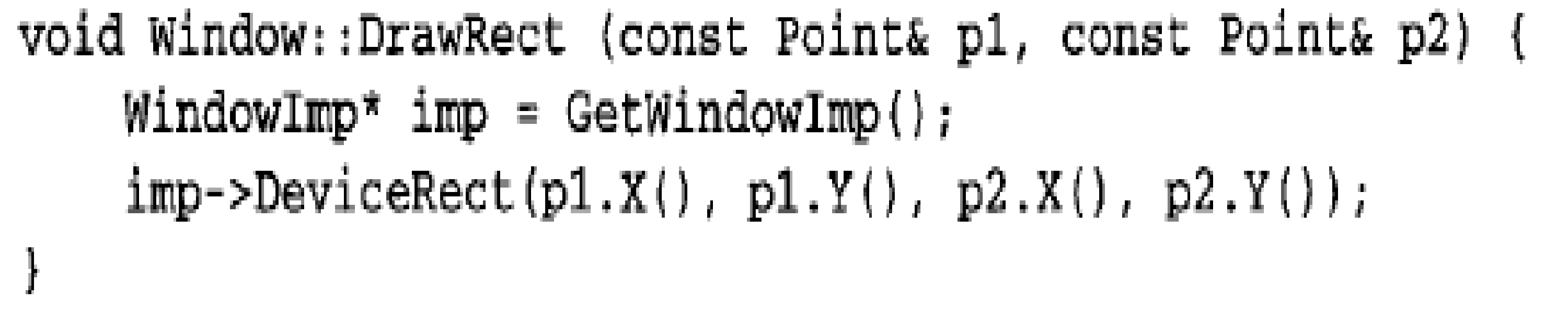
```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;


    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();
};
```

- **Subclasses of Window define the different kinds of windows the application might use, such as application windows, icons, transient windows for dialogs, floating palettes of tools, and so on.**
- **ApplicationWindow will implement DrawContents to draw the View instance it stores:**



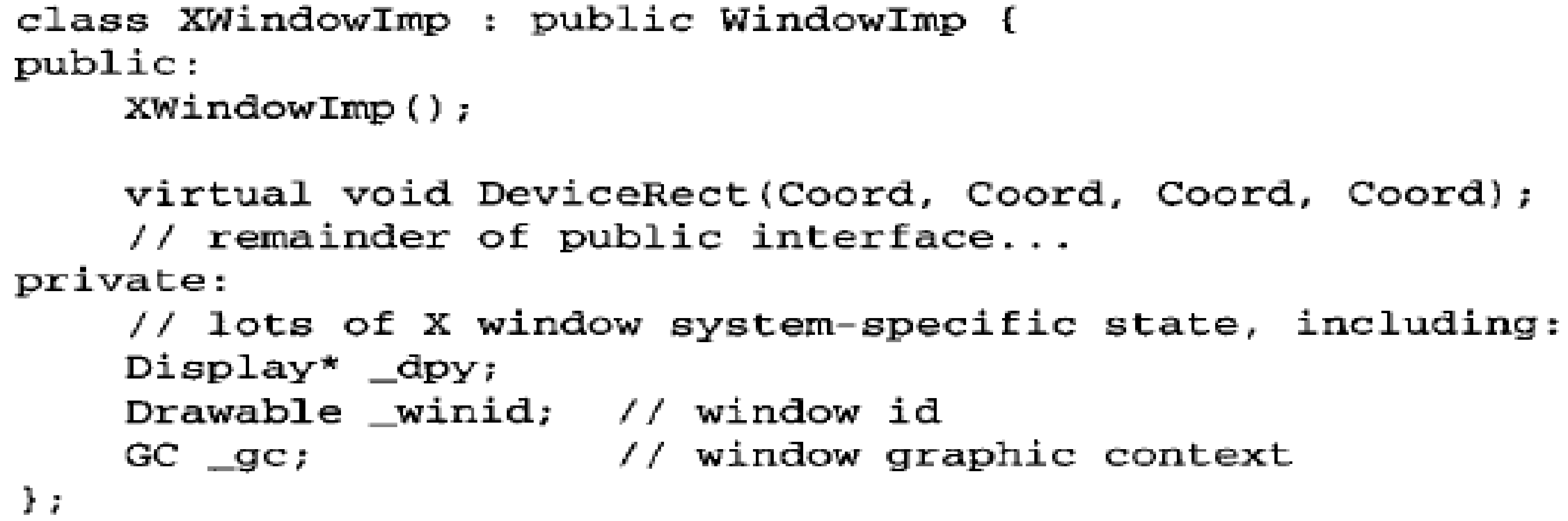
```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

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

DrawRect extracts four coordinates from its two Point parameters before calling the WindowImp operation that draws the rectangle in the window:



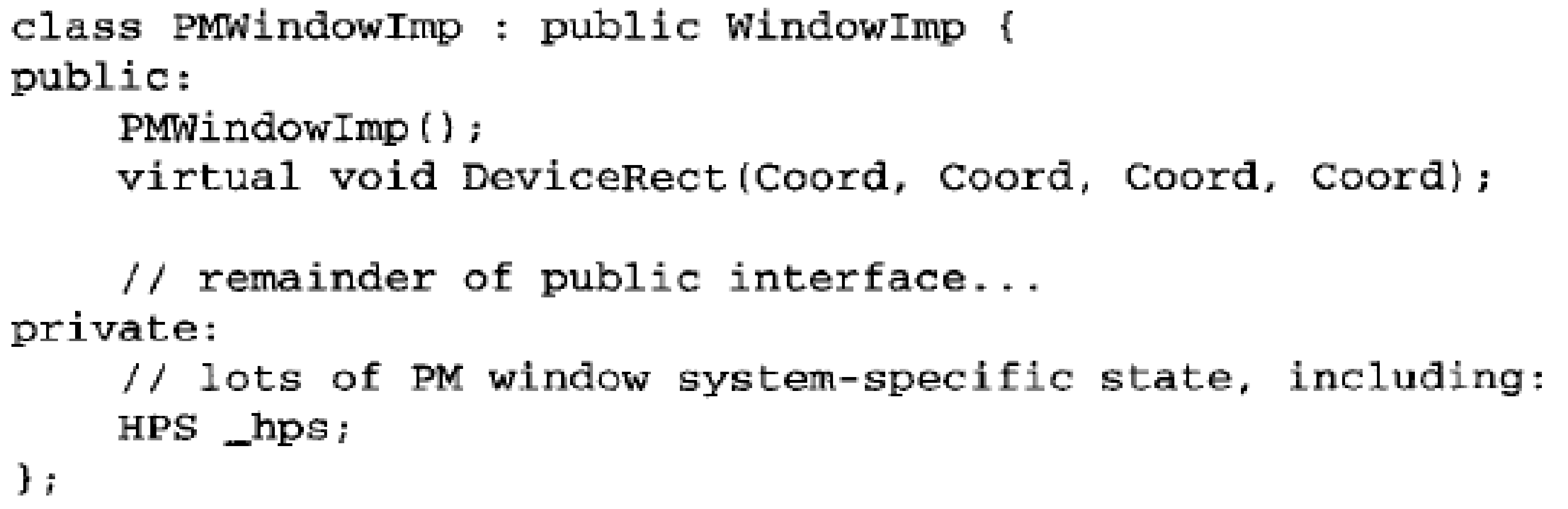
```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

Concrete subclasses of WindowImp support different window systems. The XWindowImp subclass supports the XWindow System



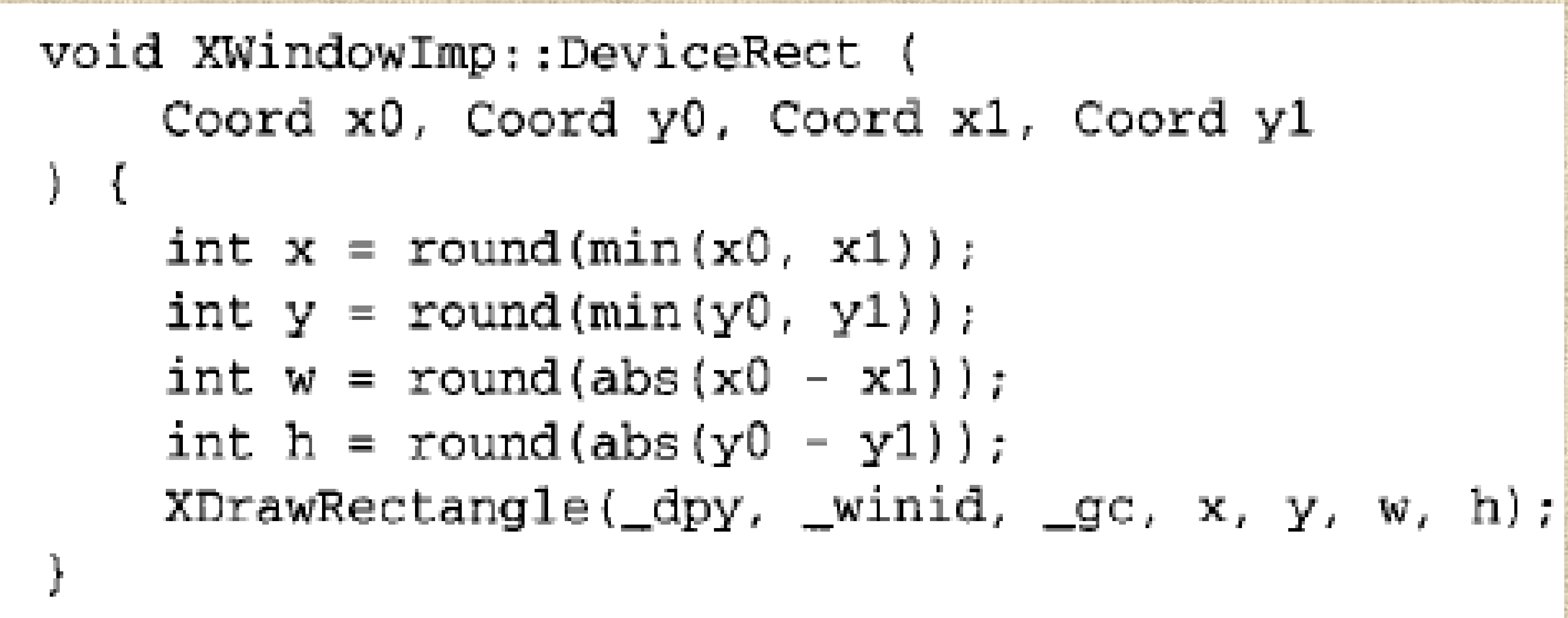
```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    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 PMWindowImp 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;
};
```

These subclasses implement WindowImp operations in terms of window system primitives. 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);
}
```

Known Uses:

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

Related Patterns:

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

COMPOSITE : Object Structural patterns

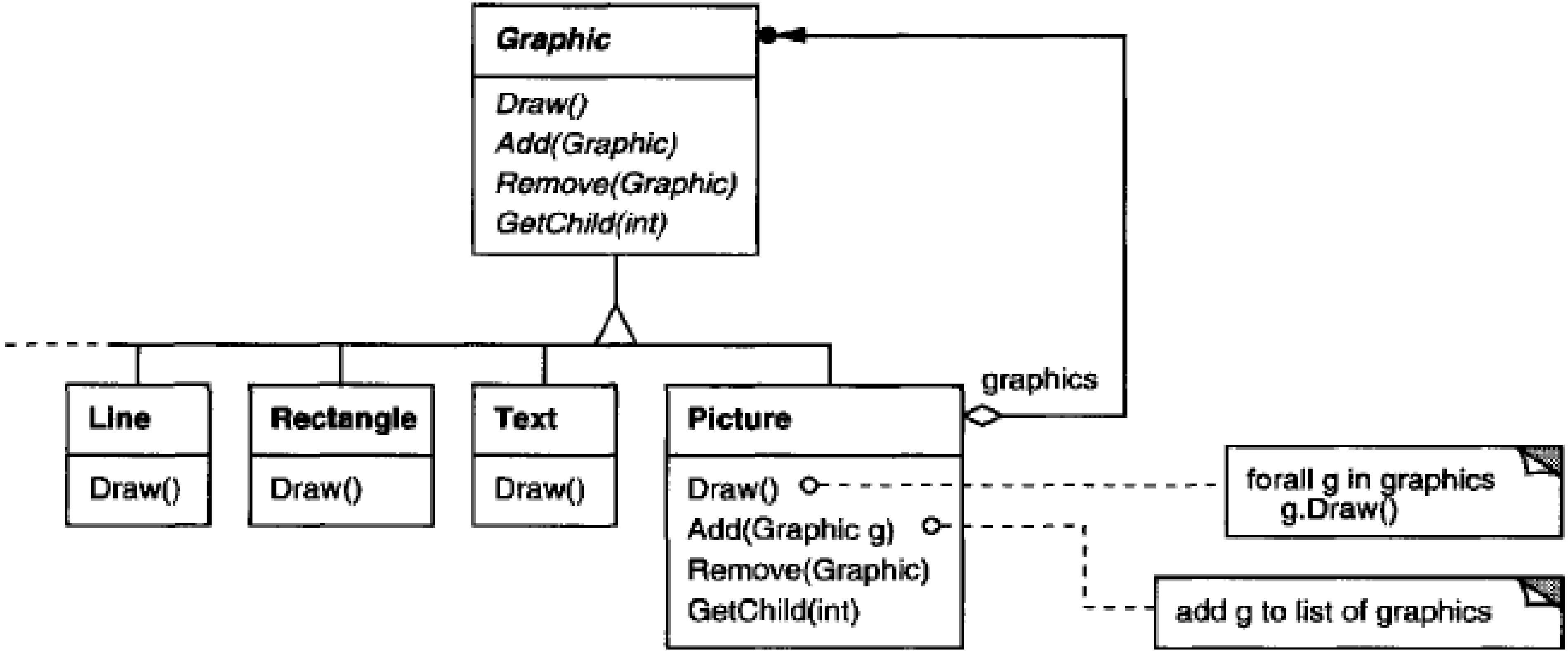
Intent:

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

Motivation:

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

- Example



Problem:

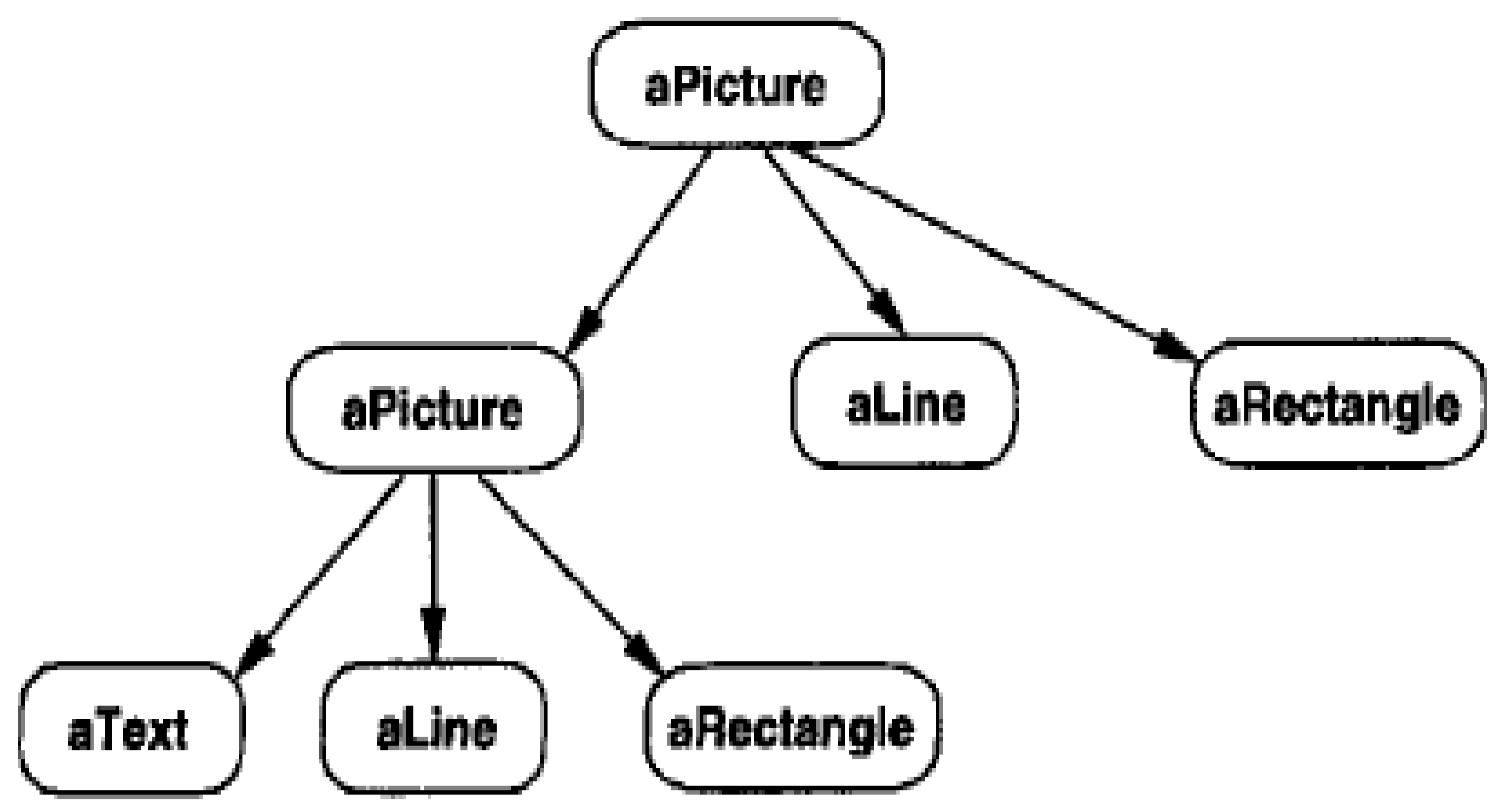
- Code that uses these classes must ***treat primitive and container objects differently,***
- Even if most of the time the user treats them identically. Having to ***distinguish these objects makes the application more complex***.

Solution:

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

- The key to the Composite pattern is an abstract class (Graphic) that represents both primitives and their containers.
- Graphic declares operations like Draw that are specific to graphical objects.
- It also declares operations that all composite objects share, such as operations for accessing and managing its children.
- The subclasses Line, Rectangle, and Text define primitive graphical objects.
- Since primitive graphics have no child graphics, none of these subclasses implements child-related operations.
- The Picture class defines an aggregate of Graphic objects.
- Because the Picture interface conforms to the Graphic interface, Picture objects can compose other Pictures recursively.

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

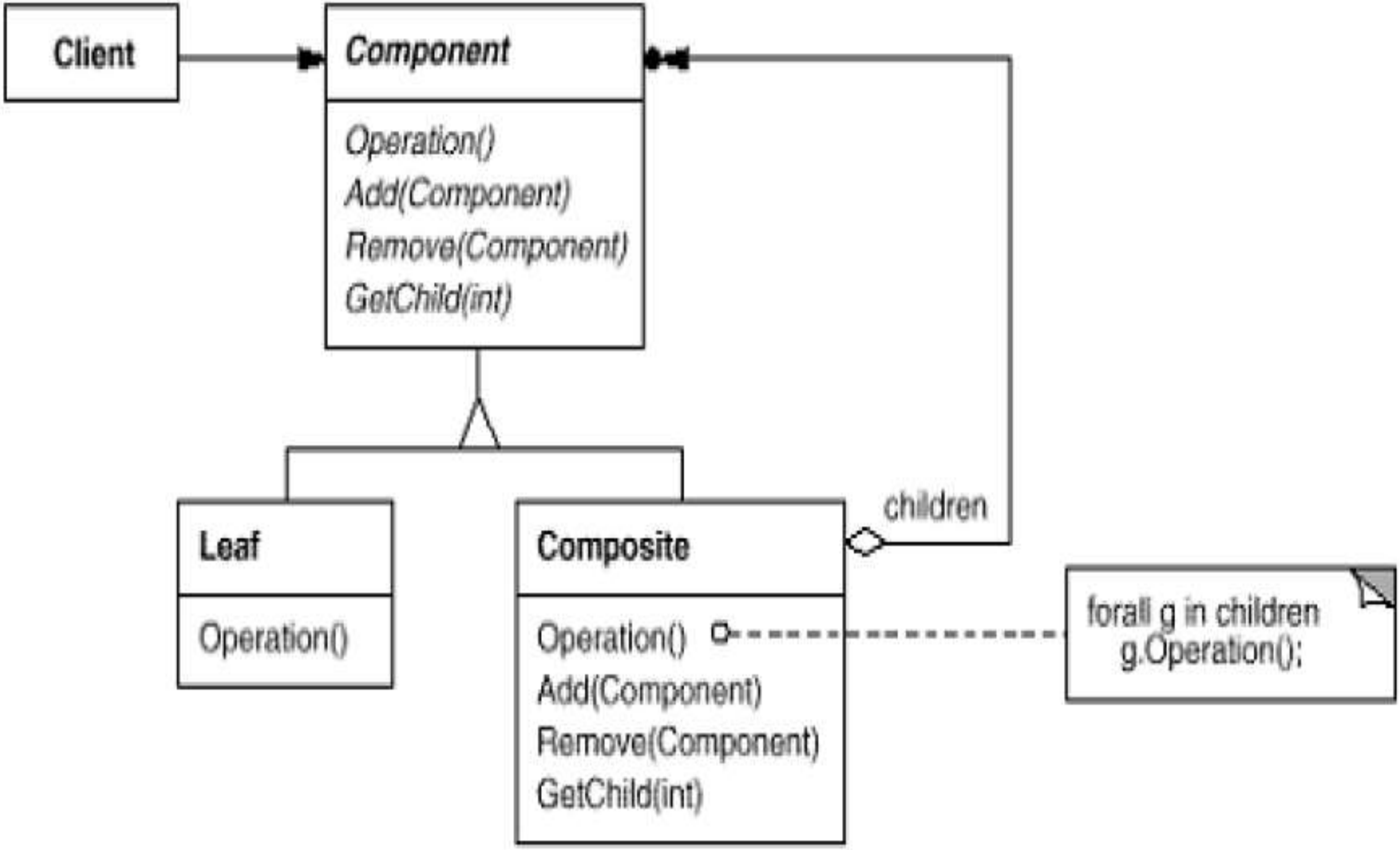


Applicability:

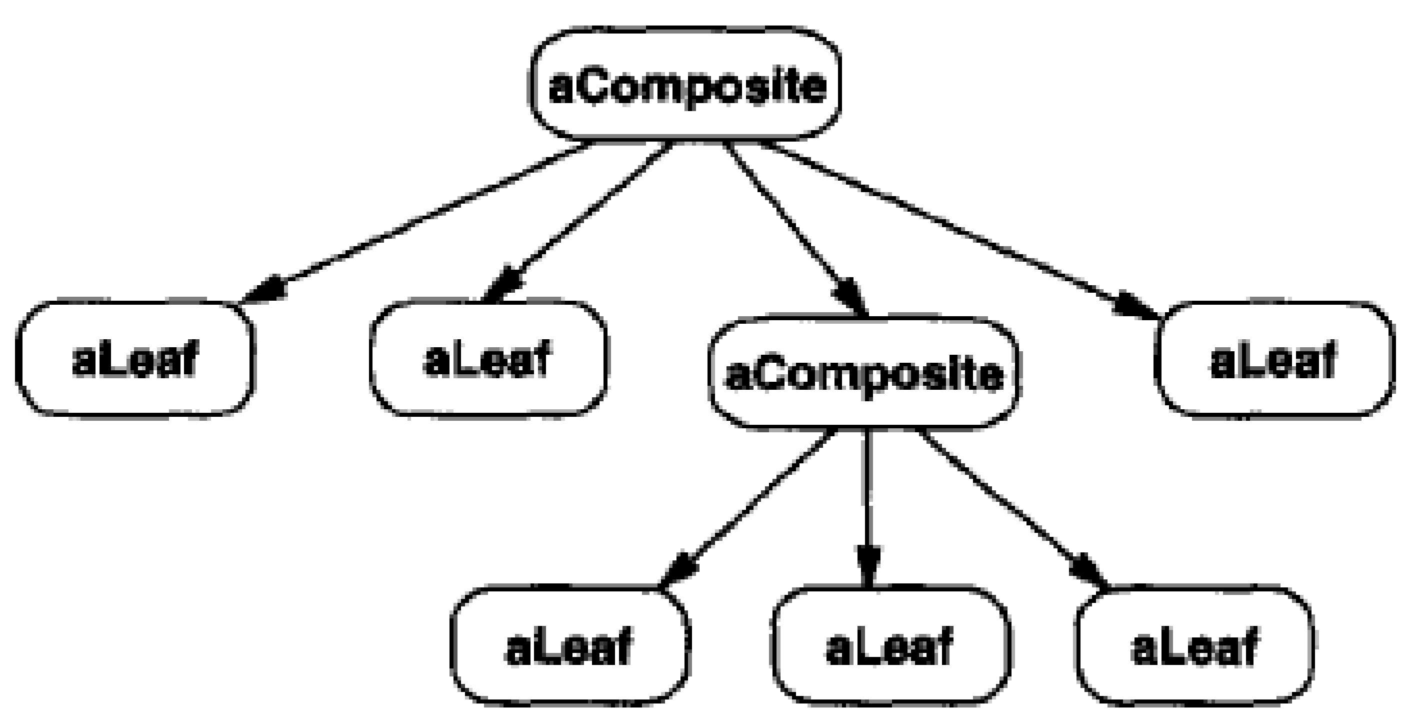
Use the Composite pattern when

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

Structure:


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation();

A typical Composite object structure might look like this:



Participants:

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

Collaborations:

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

Consequences:

I. Defines class hierarchies

- Defines class hierarchies consisting of primitive objects and composite objects.
- Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.
- Wherever client code expects a primitive object, it can also take a composite object.

II. makes the client simple.

- Clients can treat composite structures and individual objects uniformly.
- Clients normally don't know (and shouldn't care)whether they're dealing with a leaf or a composite component.
- This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

Consequences:

III. makes it easier to add new kinds o f components.

- Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.
- Clients don't have to be changed for new Component classes.

IV. 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.
- With Composite, you can't rely on the type system to enforce those constraints for you.
- You'll have to use run-time checks instead.

Implementation:

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

1. *Explicit parent references.*
2. *Sharing components.*
3. *Maximizing the Component interface.*
4. *Declaring the child management operations.*
5. *Should Component implement a list of Components?*
6. *Child ordering.*
7. *Caching to improve performance.*
8. *Who should delete components?*
9. *What's the best data structure for storing components?*

Implementation: Contd…

1. Explicit parent references.

- Maintaining references from child components to their parent can simplify the traversal and management.
- The parent reference simplifies moving up the structure and deleting a component.
- Parent references also help support the Chain of Responsibility (223) pattern.
- Usually parent reference defined in the Component class.
- Leaf and Composite classes can inherit the reference and the operations that manage it.
- With parent references, it's essential to maintain the invariant of all children.
- The easiest way to ensure this is to change a component's parent *only* when it's being added or removed from a composite.
- If this can be implemented once in the Add and Remove operations of the Composite class, then it can be inherited by all the subclasses, and the invariant will be maintained automatically.

2. Sharing components.

- Useful to share components for ex, to reduce storage requirements.
- when a component can have no more than one parent, sharing components becomes difficult.
- Solution:
  - children to store multiple parents :- But this can lead to ambiguities as a request propagates up the structure.
  - The Flyweight(195) pattern shows how to rework a design to avoid storing parents altogether
  - children can avoid sending parent requests by externalizing some or all of their state.

3. Maximizing the Component interface.

- **Goals of the Composite pattern is to make clients unaware of the specific Leaf or Composite classes they're using.-> This could be achieved by defining more 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.
- This sometimes makes operations to be defined that doesn't make sense for subclasses.
- Define a default operation for child access in the Component class that never *returns* any children.
- Leaf classes can use the default implementation, but Composite classes will reimplement it to return their children.

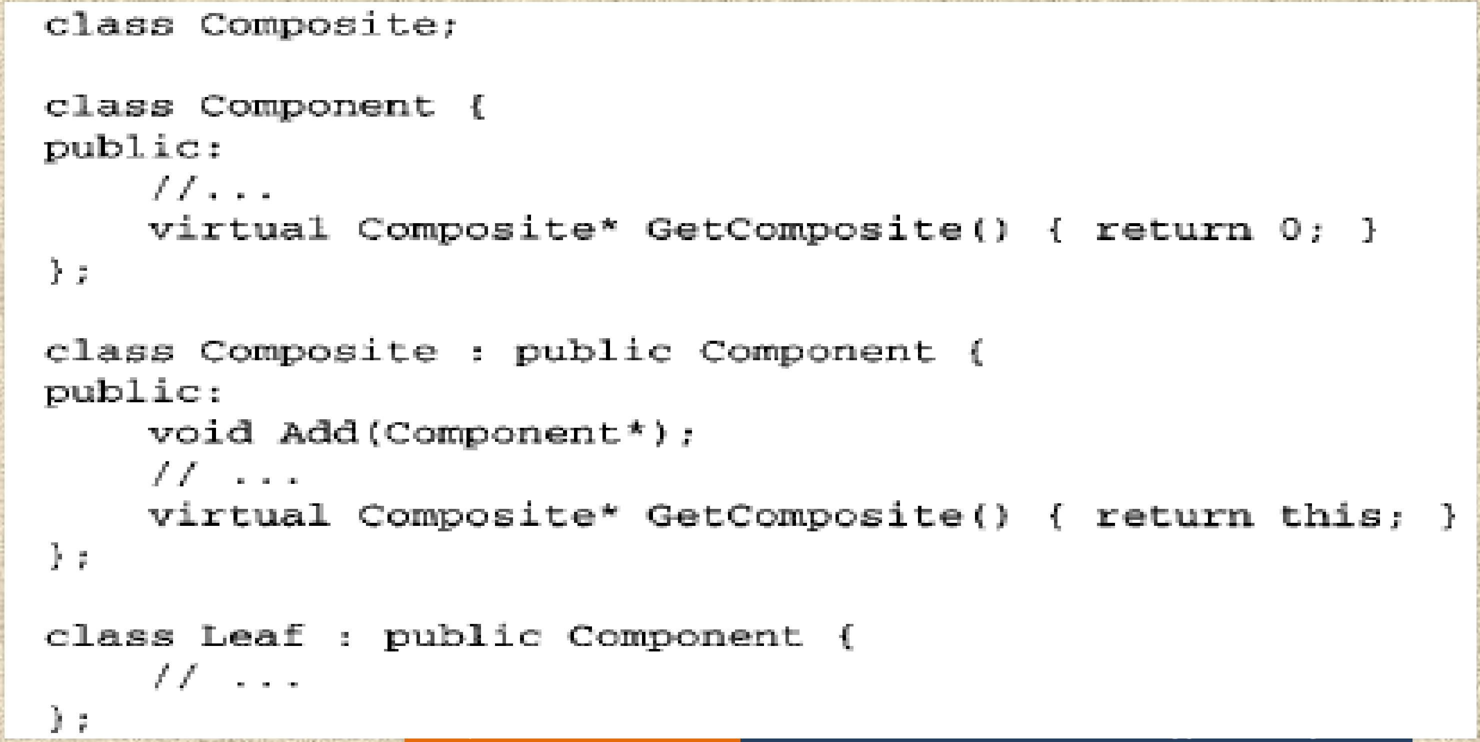
4. Declaring the child management operations.

- 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.
- Should we declare these operations in the Component and make them meaningful for Leaf classes, or should we declare and define them only in Composite and its subclasses?

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

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

- One approach is to declare an operation Composite *GetComposite ( ) in the Component class.
- Component provides a default operation that returns a null pointer.
- The Composite class redefines this operation to return itself through the this pointer:



```
class Composite;

class Component {
public:
    //...
    virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
    void Add(Component*);
    // ...
    virtual Composite* GetComposite() { return this; }
};

class Leaf : public Component {
    // ...
};
```

5. Should Component implement a list of Components?

- Usually we 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.
- This is worthwhile only if there are relatively few children in the structure.
- That is have few children

6. Child ordering.

- Many designs specify an ordering on the children of Composite.
- 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.
- When child ordering is an issue, one must design child access and management interfaces carefully to manage the sequence of children.
- The Iterator (257)pattern can guide you in this.

7. Caching to improve performance.

- 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.
- Changes to a component will require invalidating the caches of its parents.
- This works best when components know their parents.

8. Who should delete components?

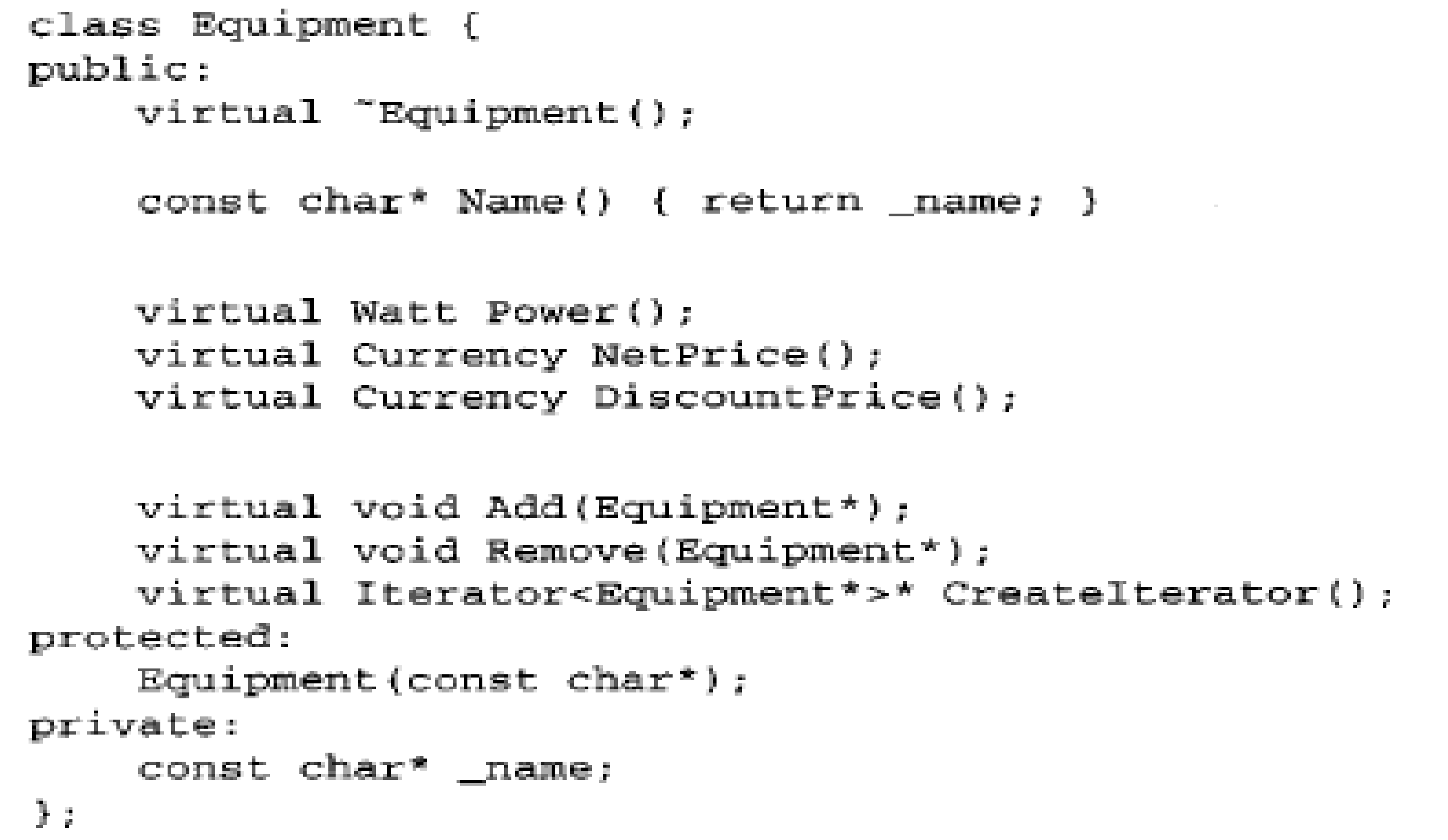
- In languages without garbage collection, it's usually best to make a Composite responsible for deleting its children when it's destroyed.
- An exception to this rule is when Leaf objects are immutable and thus can be shared.

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.
- It isn't even necessary to use a general-purpose data structure at all.
- Sometimes composites have a variable for each child, although this requires each subclass of Composite to implement its own management interface.

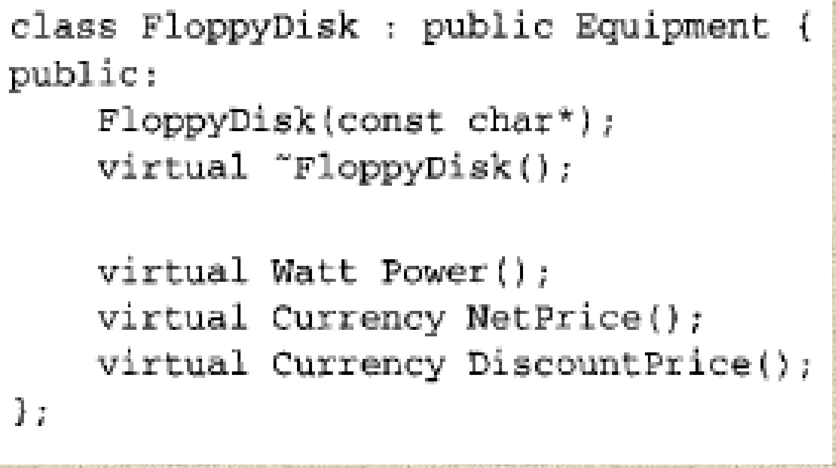
Sample Code

- ***Equipment class*** defines an interface for all equipment in the part-whole hierarchy.
- Equipment such as computers and stereo components are often organized into part-whole or containment hierarchies.



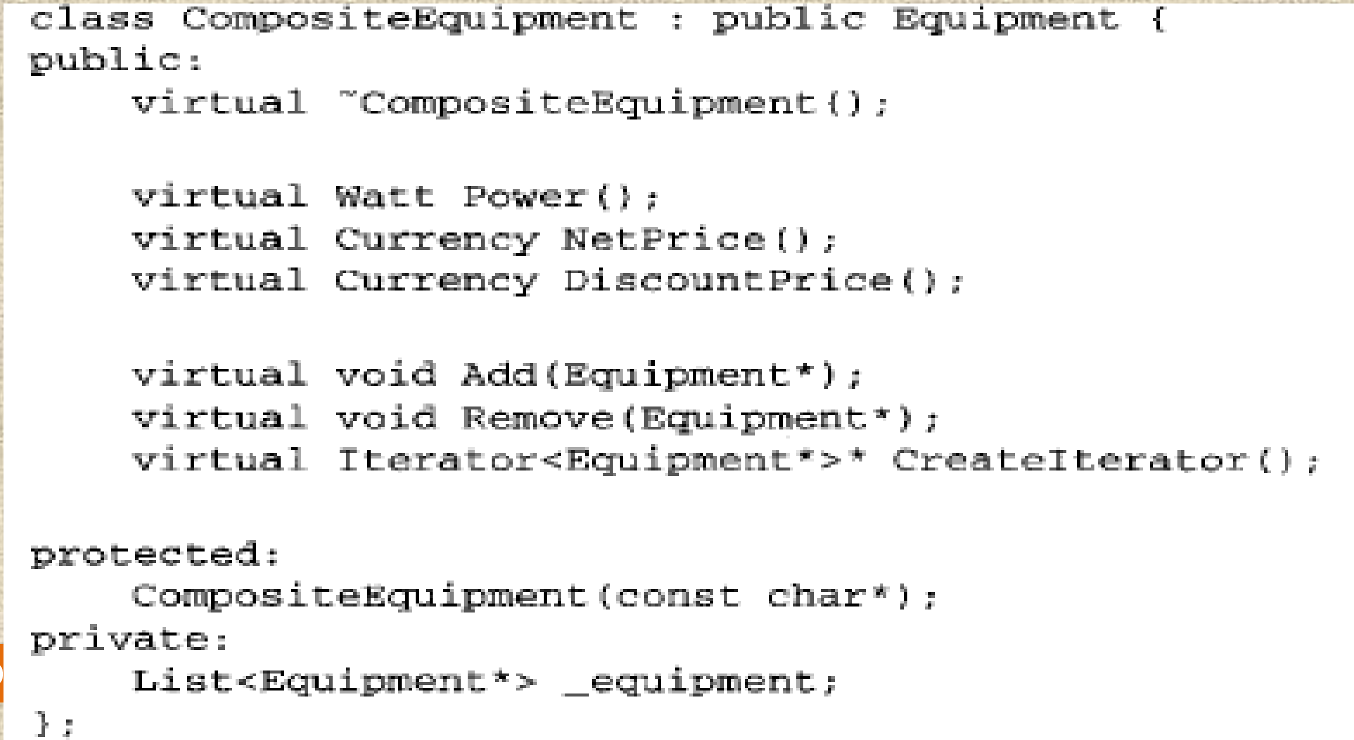
```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

- Subclasses of Equipment might include Leaf classes that represent disk drives, integrated circuits, and switches:



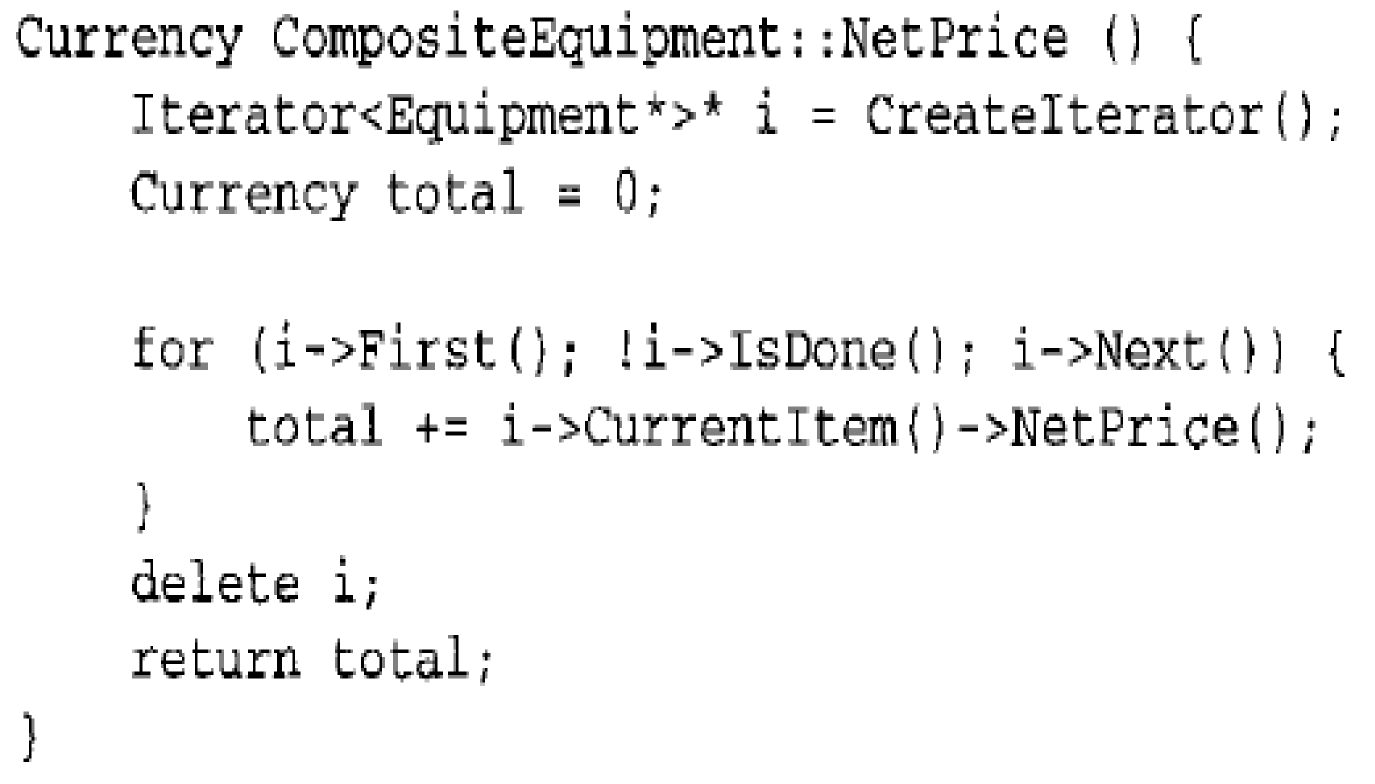
```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

CompositeEquipment is the base class for equipment that contains other equipment. It's also a subclass of Equipment.



```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List<Equipment*> _equipment;
};
```

A default implementation of NetPrice might use Createlterator to sum the net prices of the subequipment.



```
Currency CompositeEquipment::NetPrice () {
    Iterator<Equipment*>* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

Known Uses

- The original View class of Smalltalk Model/View/Controller [KP88] was a Composite,
- User interface toolkits and frameworks
- ET++, Interviews, Graphics, Glyphs

Related Patterns

- **Chain of Responsibility (223).**
- **Decorator (175) :** 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.
- **Flyweight (195)** lets you share components, but they can no longer refer to their parents.
- **Iterator (257)**can be used to traverse composites.
- **Visitor(331)** localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes.

DECORATOR: an Object structural pattern

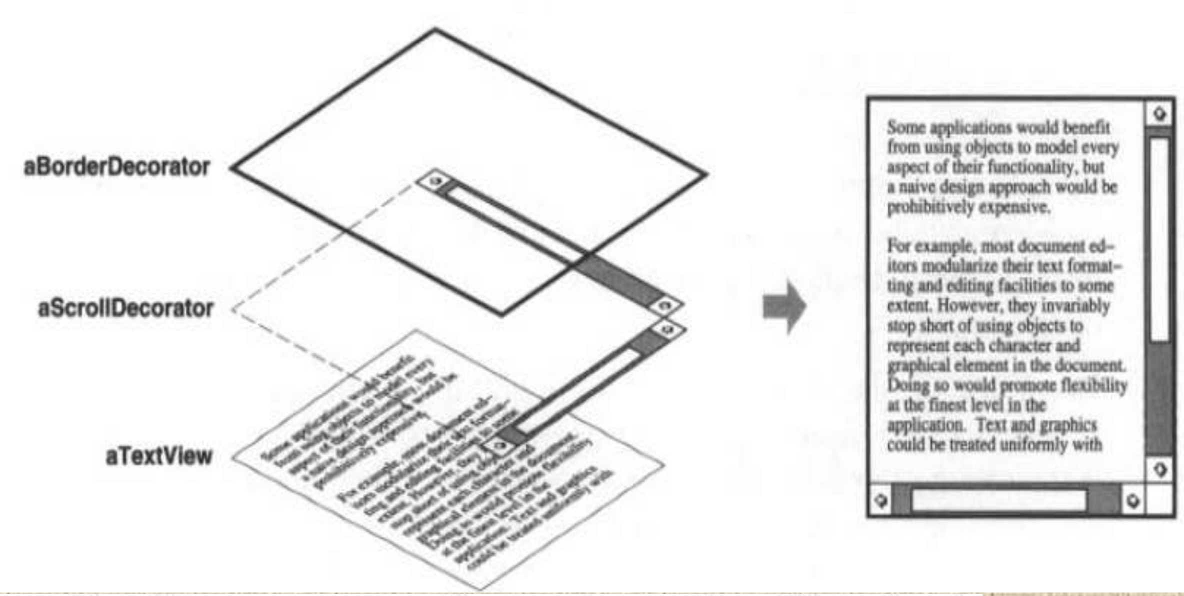
- **Intent:**

➢ Attach additional responsibilities to an object dynamically.

➢ Decorators provide a flexible alternative to subclassing for extending functionality.

- **Also Known As**

Wrapper



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

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

- Several classes with a similar operation (method), but different behavior.
- We want to use many combinations of these behaviors

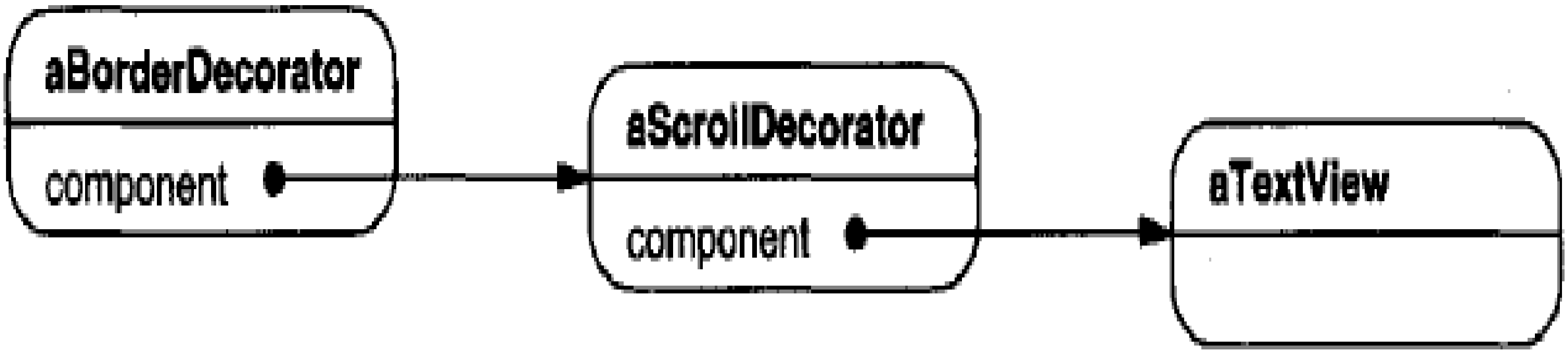
- **Motivation:**

- 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, because the choice of border is made statically.
- A client can't control how and when to decorate the component with a border.

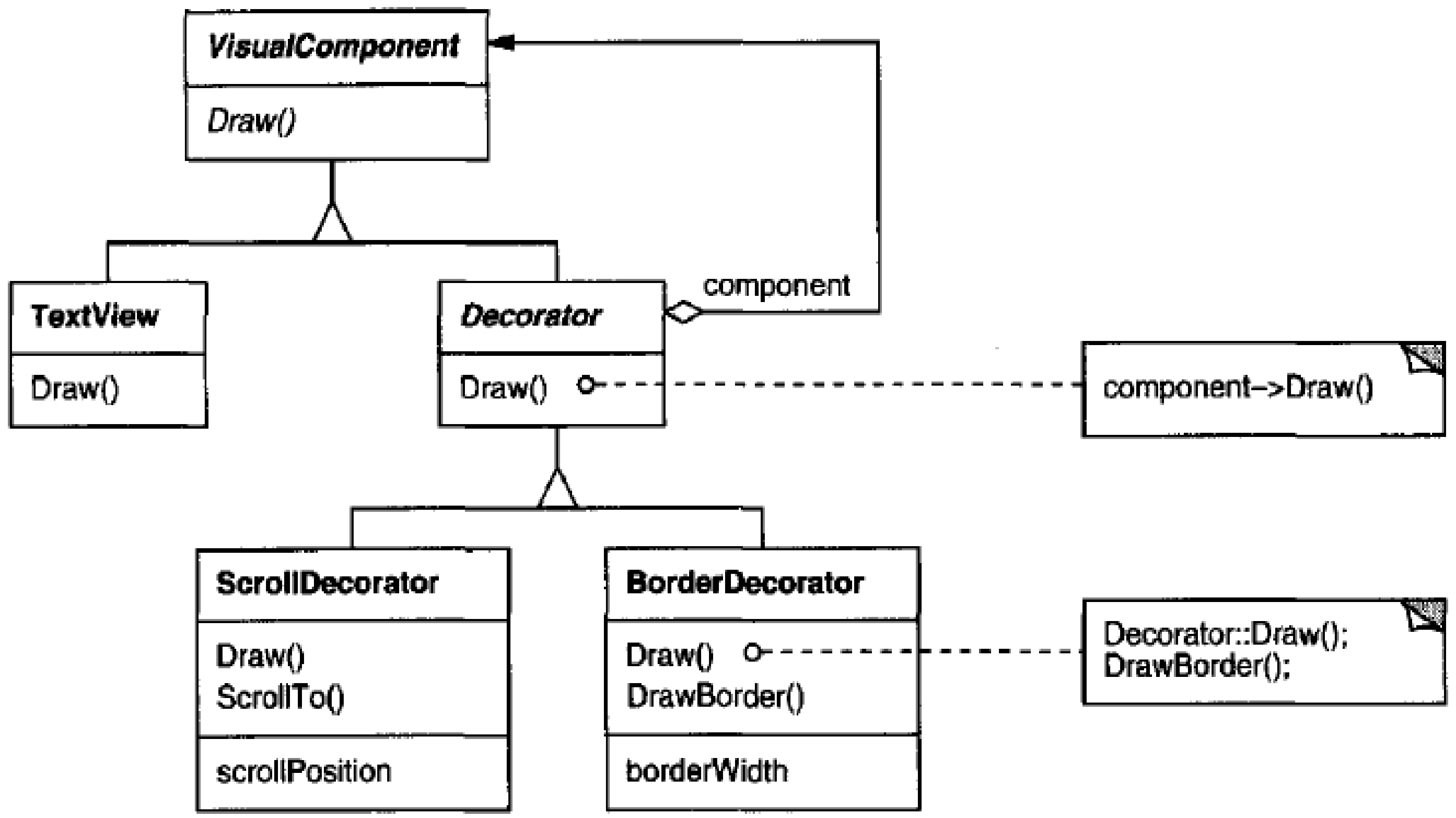
A more flexible approach is to enclose the component in another object that adds the border. The enclosing object is called a **decorator.**

- The decorator conforms to the interface of the component it decorates so that its presence is transparent to the component's clients.
- The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding.
- Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities.

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



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

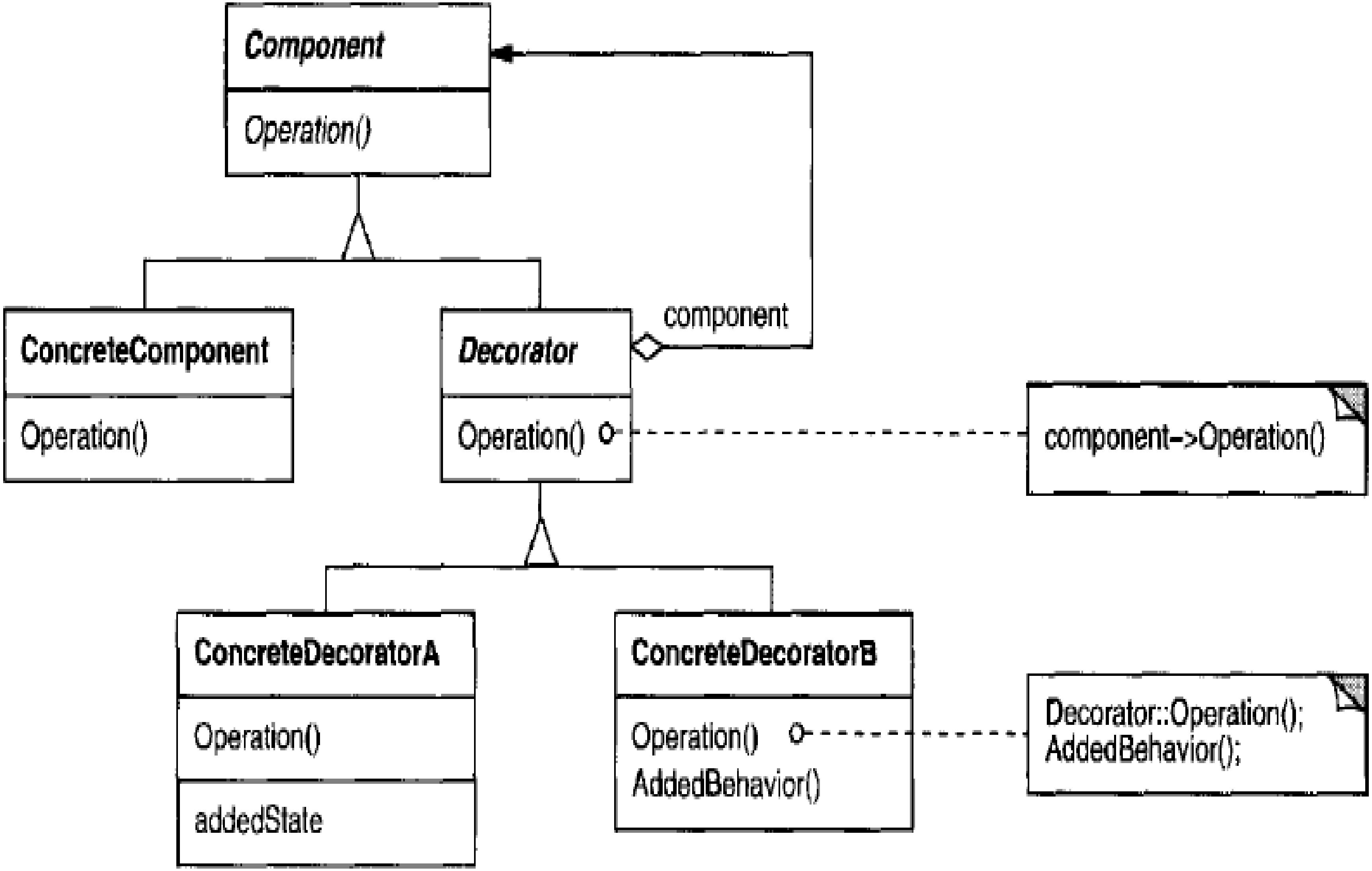


Applicability

Use Decorator

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

Structure


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

Participants

- **Component** (VisualComponent)

- defines the interface for objects that can have responsibilities added to them dynamically.

- **ConcreteComponent (TextView)**

- defines an object to which additional responsibilities can be attached.

- **Decorator**

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

- **ConcreteDecorator** (BorderDecorator, ScrollDecorator)

- adds responsibilities to the component.

Collaborations

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

Consequences

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

1. ***More flexibility than static inheritance*.**
   - The Decorator pattern provides a more flexible way to add responsibilities to objects
   - With decorators, responsibilities can be added and removed at run-time simply by attaching and detaching them.
   - In contrast, inheritance requires creating a new class for each additional responsibility:- Which increases number of classes and complexity of a system
   - Providing different Decorator classes for a specific Component class lets you mix and match responsibilities.
   - Decorators also make it easy to add a property twice

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

- Decorator offers a pay-as- you-go approach to adding responsibilities.
- Define a simple class and add functionality incrementally with Decorator objects.
- Functionality can be composed from simple pieces. A s a result, an application needn't pay for features it doesn't use.
- Easy to define new kinds of Decorators independently from the classes of objects they extend, even for unforeseen extensions.
- Extending a complex class tends to expose details of responsibilities

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

- A decorator acts as a transparent enclosure.
- From object identity point of view, a decorated component is not identical to the component itself.
- Shouldn't rely on object identity when you use decorators.

4. Lots of little objects.

- A design that uses Decorator often results in systems composed of lots of little objects that all look alike.
- The objects differ only in the way they are interconnected, not in their class or in the value of their variables.
- These systems are easy to customize, but can be hard to learn and debug.

Implementation

Several issues should be considered when applying the Decorator pattern:

1. *Interface conformance.*
2. *Omitting the abstract*
3. *Keeping Component classes lightweight*.
4. *Changing the skin of an object versus changing its guts*

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 (at least in C++).

2. *Omitting the abstract Decorator class.*

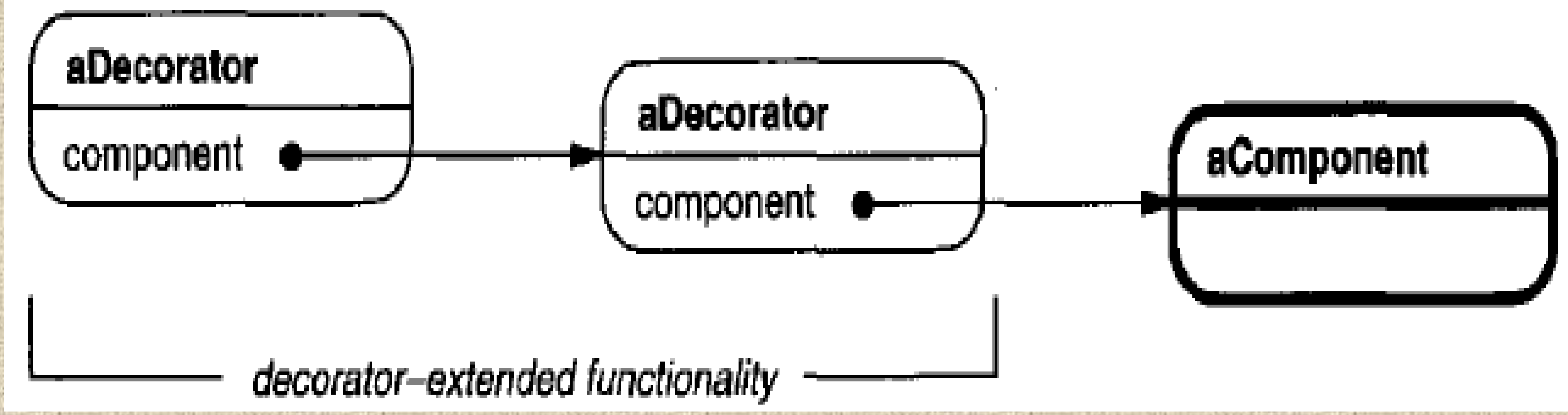
- No need to define an abstract Decorator class when only need to add one responsibility.
- Dealing with an existing class hierarchy rather than designing a new one.=> can merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.

3. Keeping Component classes lightweight

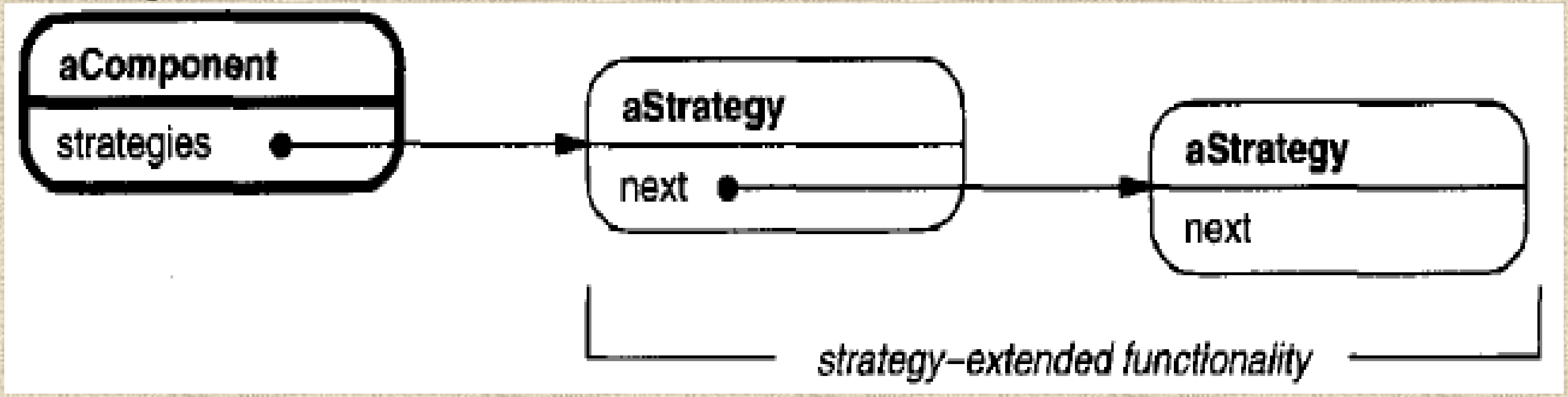
- Keep the common class lightweight; that is, it should focus on defining an interface, not on storing data.
- The definition of the data representation should be deferred to subclasses;
- Otherwise the complexity of the Component class might make the decorators too heavyweight.
- Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.

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

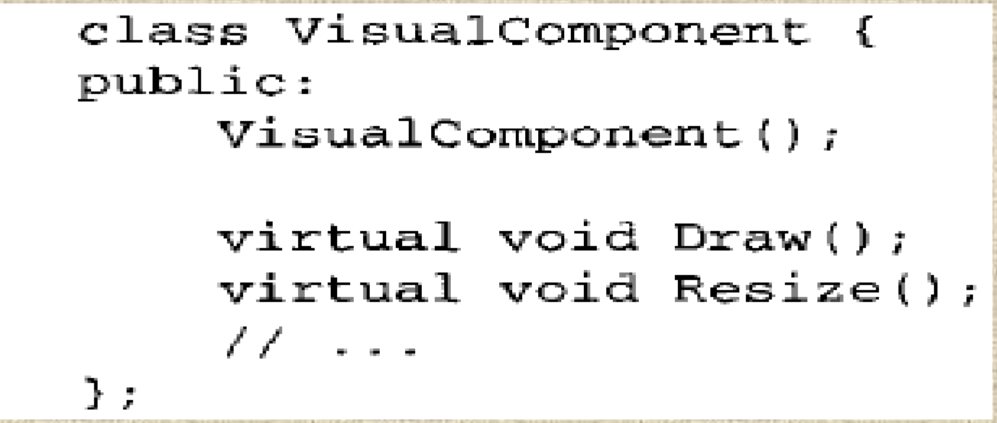
- A decorator as a skin over an object that changes its behavior.
- An alternative is to change the object's guts. Ex: Strategy (315) pattern for changing the guts.
- Since the Decorator pattern only changes a component from the outside, the component doesn't have to know anything about its decorators; that is, the decorators are transparent to the component:



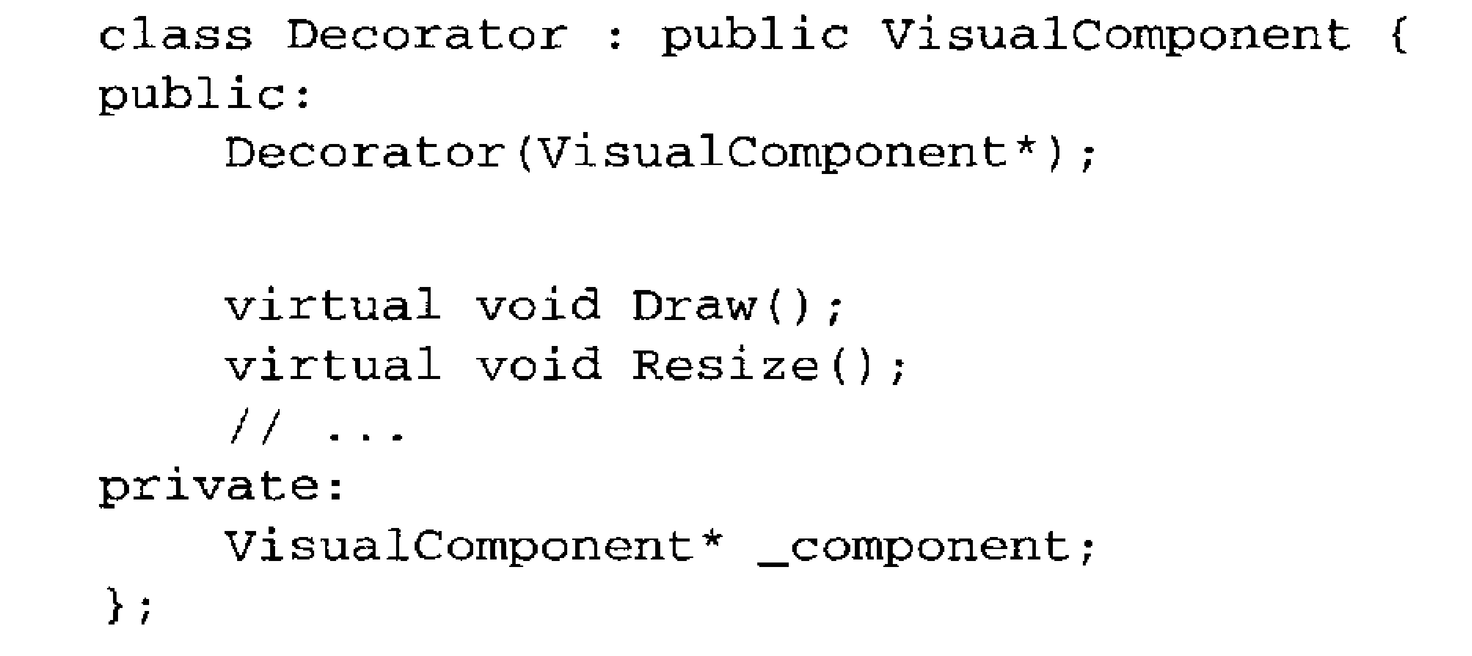
With strategies, the component itself knows about possible extensions. So it has to reference and maintain the corresponding strategies:



Sample Code

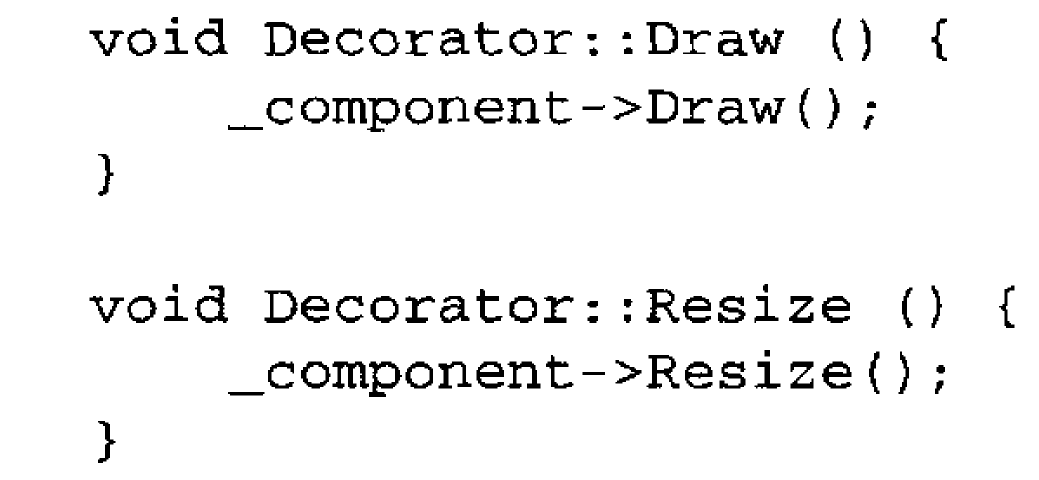


```
class VisualComponent {
public:
    VisualComponent();

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



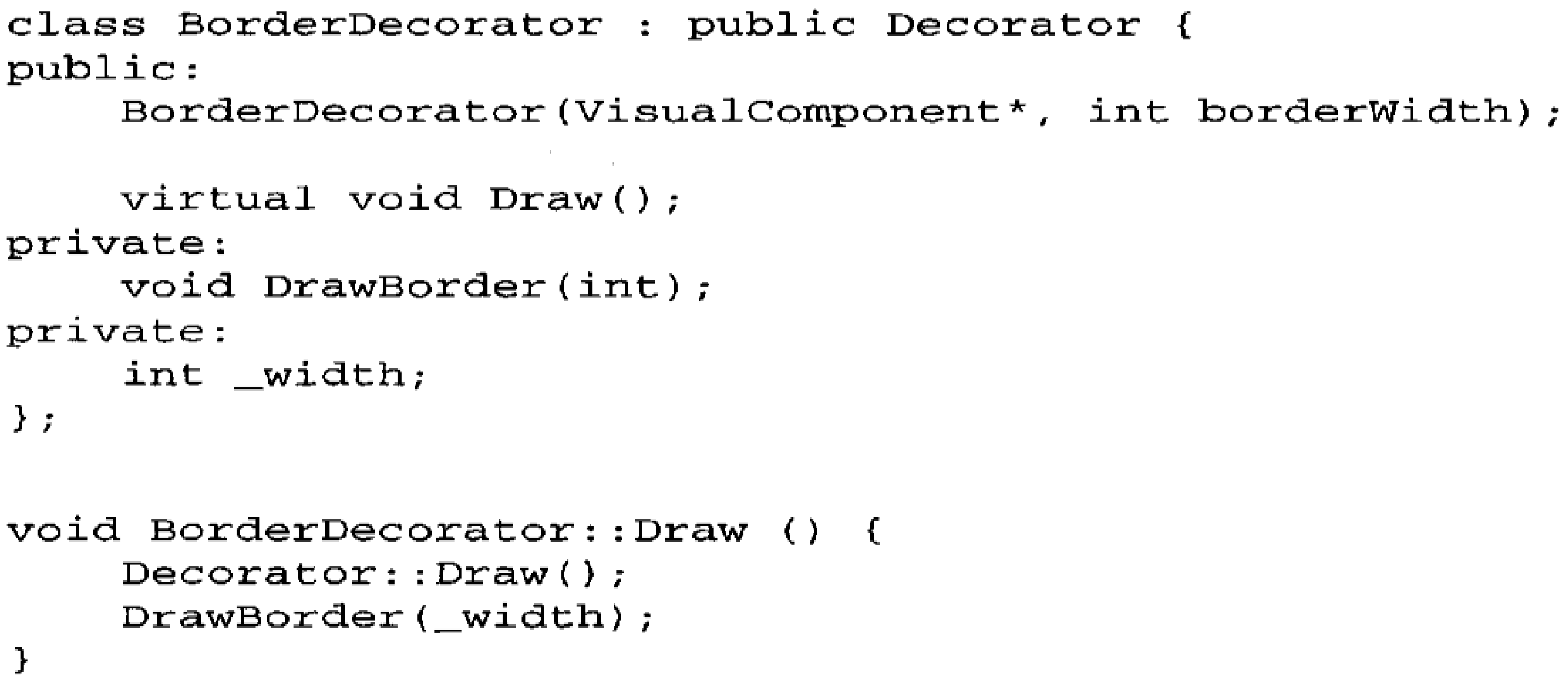
```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);


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

For each operation in **VisualComponent's interface,** Decorator defines a **default implementation** that **passes the request on to -component:**



```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

Subclasses of Decorator define specific decorations



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

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


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

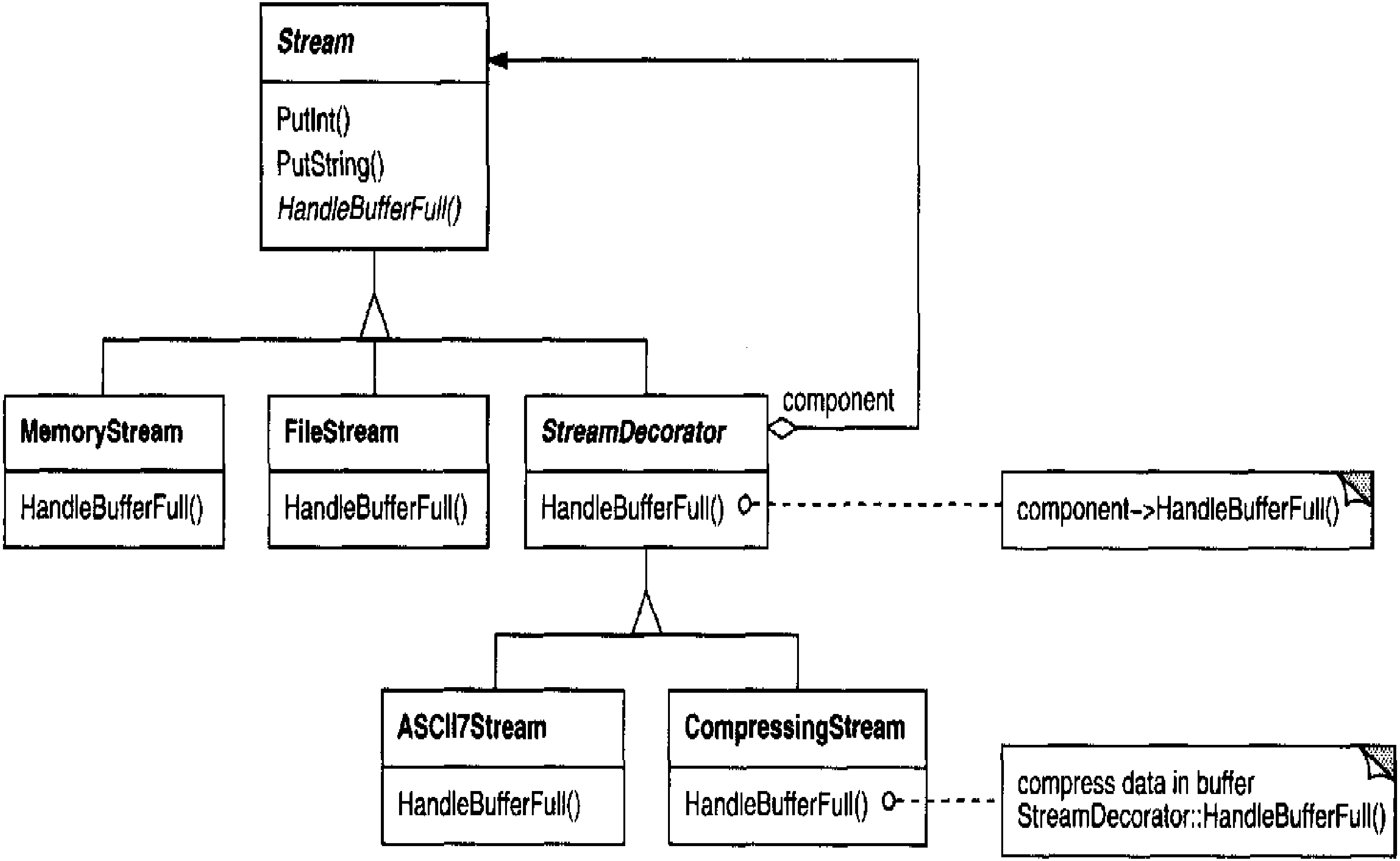
Known Uses

Many object-oriented user interface toolkits use decorators to add graphical embellishments to widgets.

Examples: Interviews, ET++, the ObjectWorks\Sma lltalkclass library

- The DebuggingGlyph from Interviews
- The PassivityWrapper from ParcPlace Smalltalk.

The Decorator pattern gives us an elegant way to add these responsibilities to streams. The diagram below



Related Patterns

Adapter (139) :

- 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.

Composite (163) :

- 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.

Strategy (315) :

- 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.

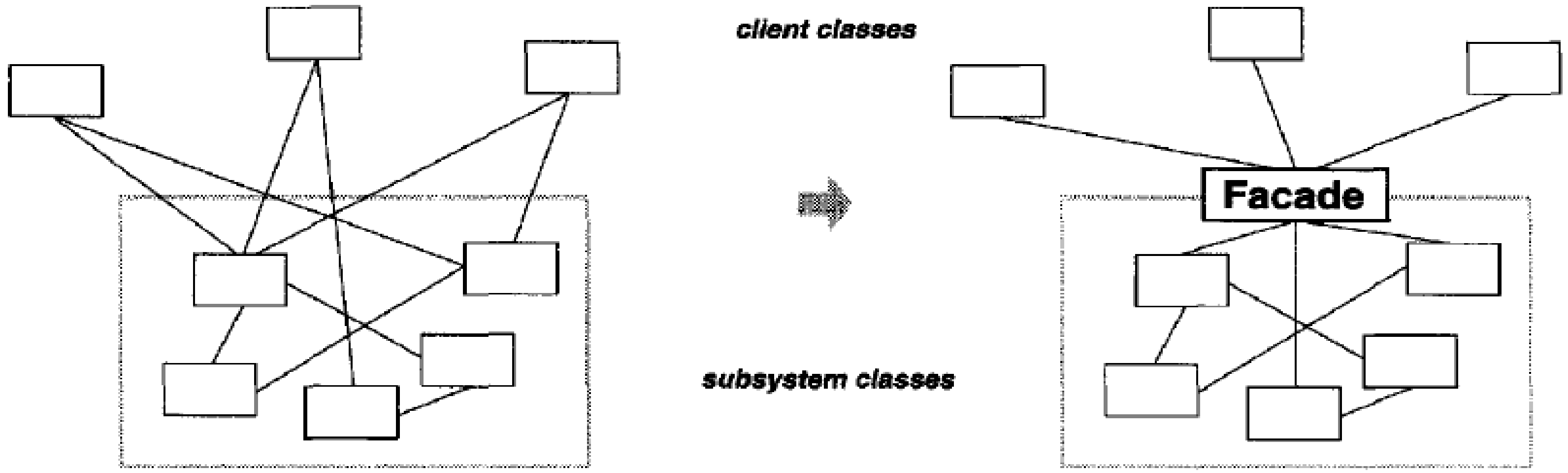
FACADE : Objects structural patterns

Intent

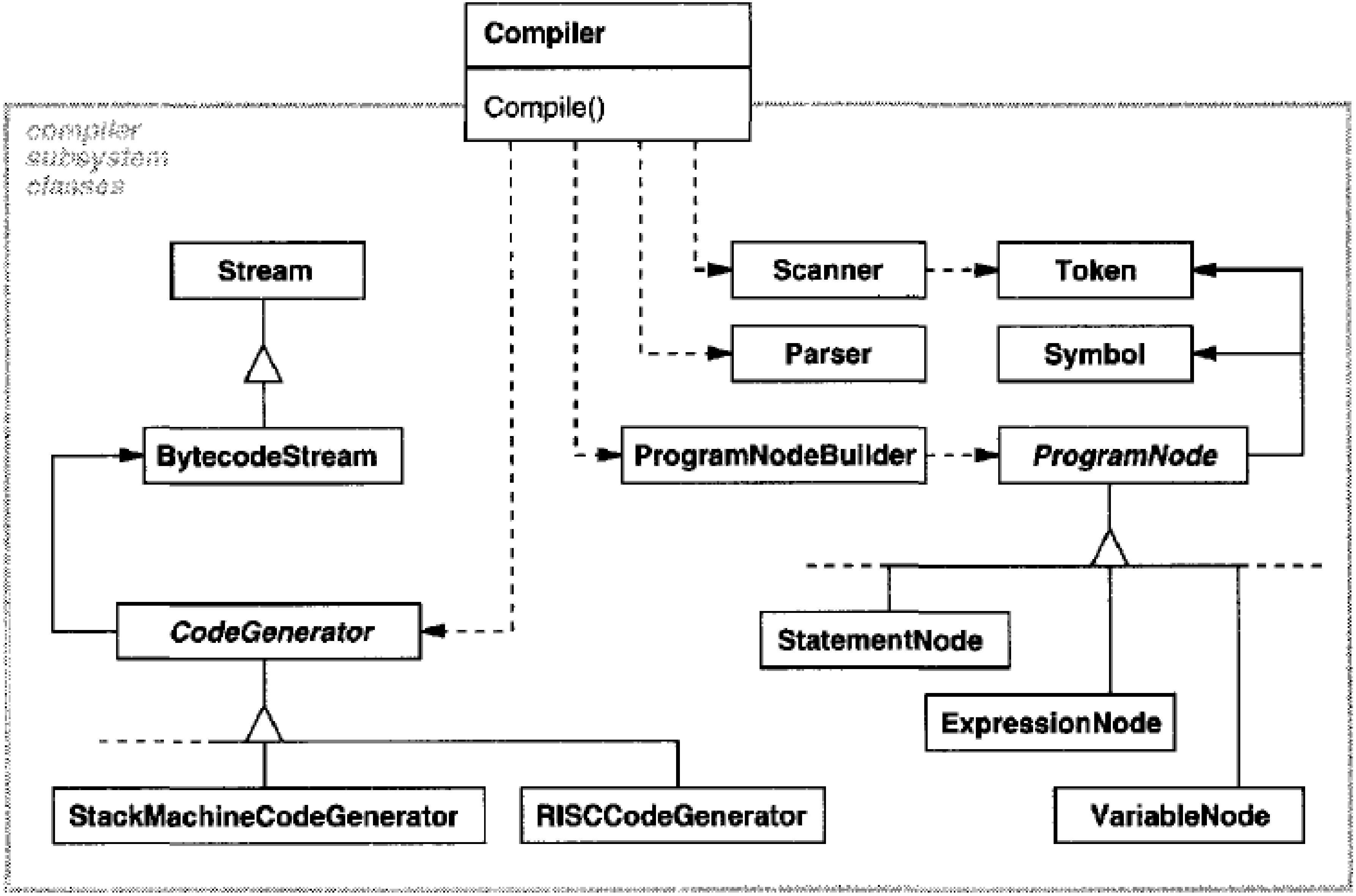
Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation

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



- **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.**
- **Most clients of a compiler generally don't care about details like parsing and code generation; they merely want to compile some code**
- **To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class=>** defines a unified interface to the compiler's functionality.
- The Compiler class acts as a façade=>
  - ➢ Offers clients a single, simple interface to the compiler subsystem.
  - ➢ It glues together the classes that implement compiler functionality without hiding them completely


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

Applicability

Use the Façade pattern when

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

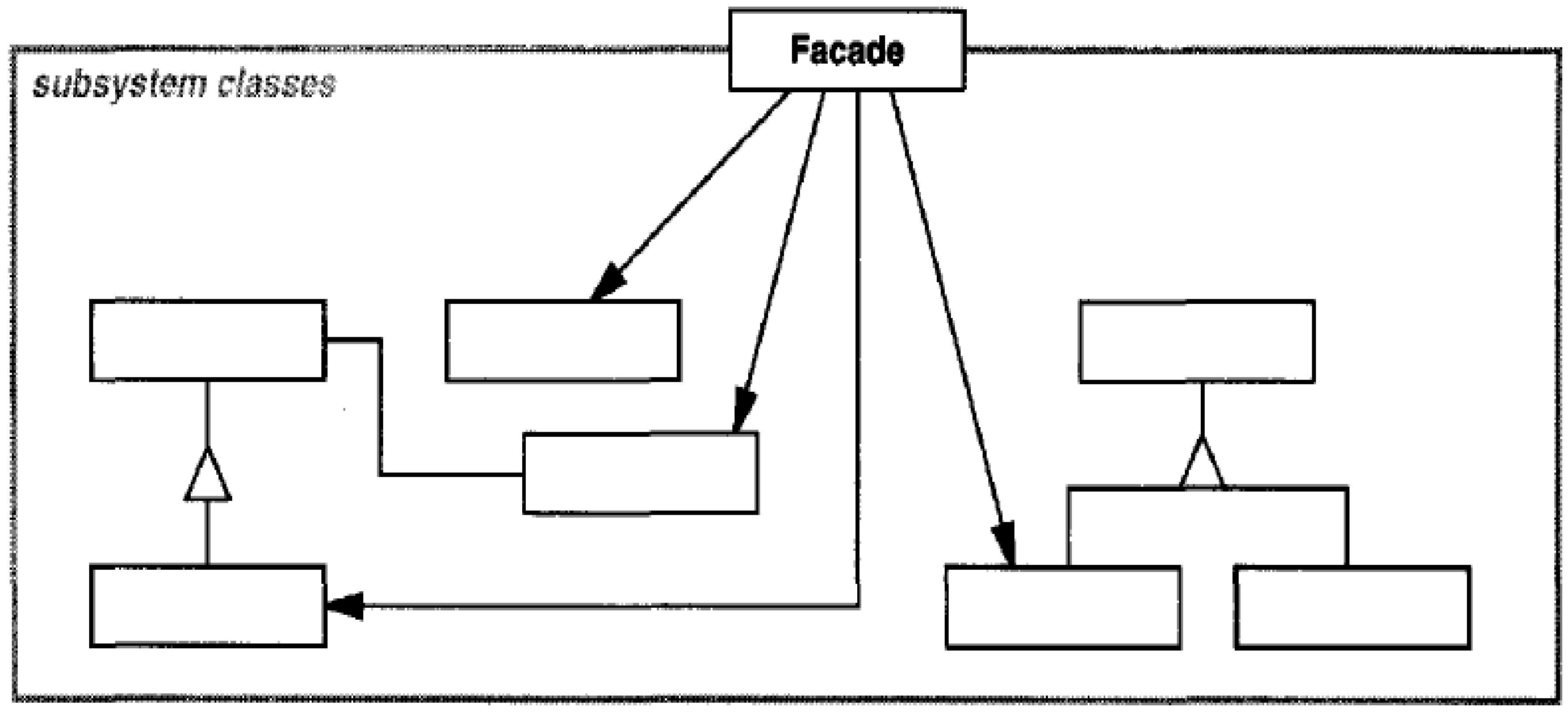
2. Many dependencies between clients and the implementation classes of an abstraction.

- ➢ Introduce a facade to decouple the subsystem from clients and other subsystems,
- ➢ Thereby promoting subsystem independence and portability.

3. You want to layer your subsystems.

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

Structure



Participants

- **Facade** (Compiler)

- knows which subsystem classes are responsible for a request.
- delegates client requests to appropriate subsystem objects.

- **subsystem classes** (Scanner, Parser, ProgramNode, etc.)

- implement subsystem functionality.
- handle work assigned by the Facade object.
- have no knowledge of the facade; that is they keep no references to it.

Collaborations

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

Consequences

The Facade 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

Consider the following issues when implementing a facade:

1. *Reducing client-subsystem coupling.*

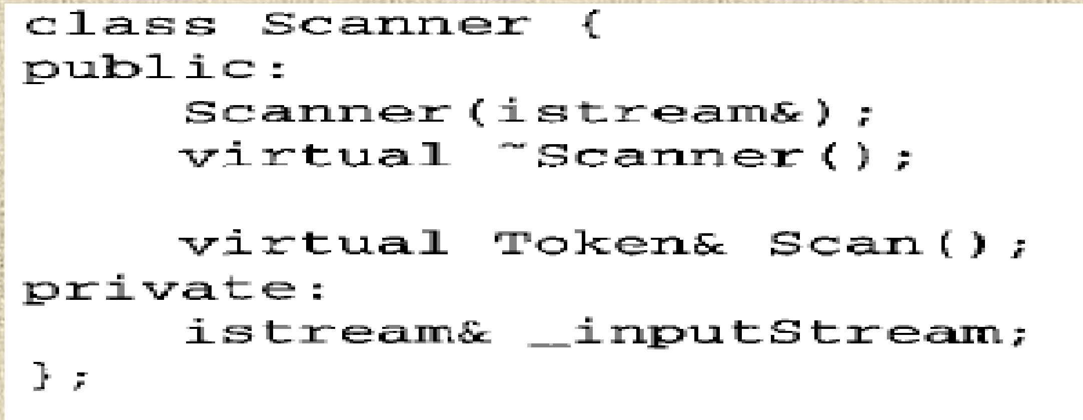
- The coupling between clients and the subsystem can be reduced even.
- Make Facade an abstract class with concrete subclasses for different implementations of a subsystem.
- Then clients can communicate with the subsystem through the interface of the abstract Facade class.
- This abstract coupling keeps clients from knowing which implementation of a subsystem is used.
- An alternative to subclassing is to configure a Facade object with different subsystem objects.
- To customize the facade, simply replace one or more of its subsystem objects.

2. Public versus private subsystem classes

- A subsystem is analogous to a class => both have interfaces, and both encapsulate something—
- a class encapsulates state and operations,
- while a subsystem encapsulates classes.
- Just as similar to public and private interfaces, consider public and private interfaces to subsystems.
- The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders.
- The Facade class is part of the public interface,

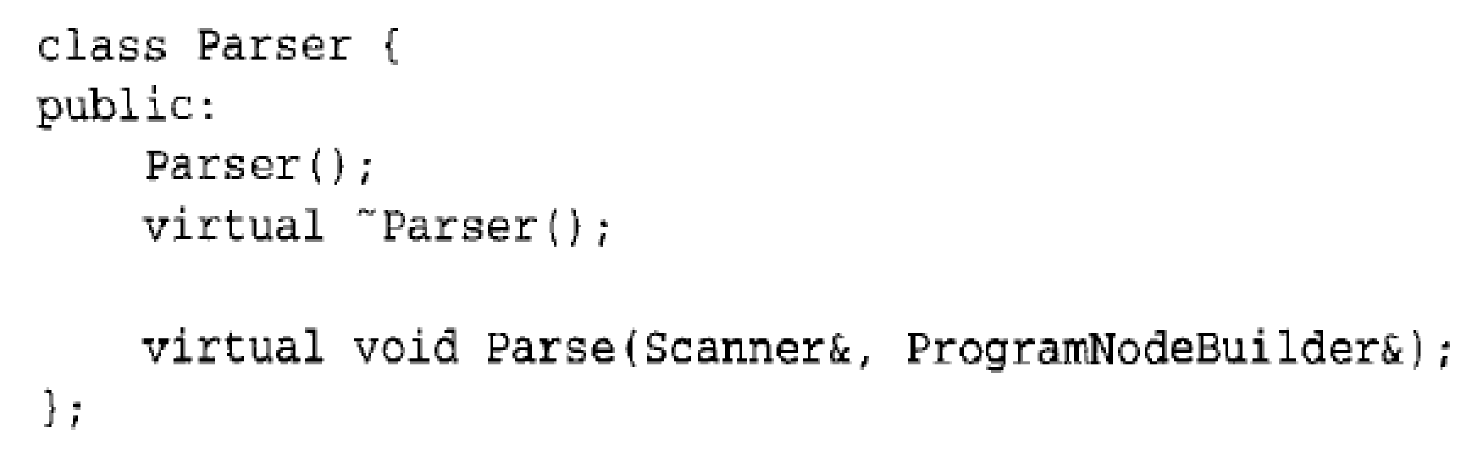
Sample Code

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



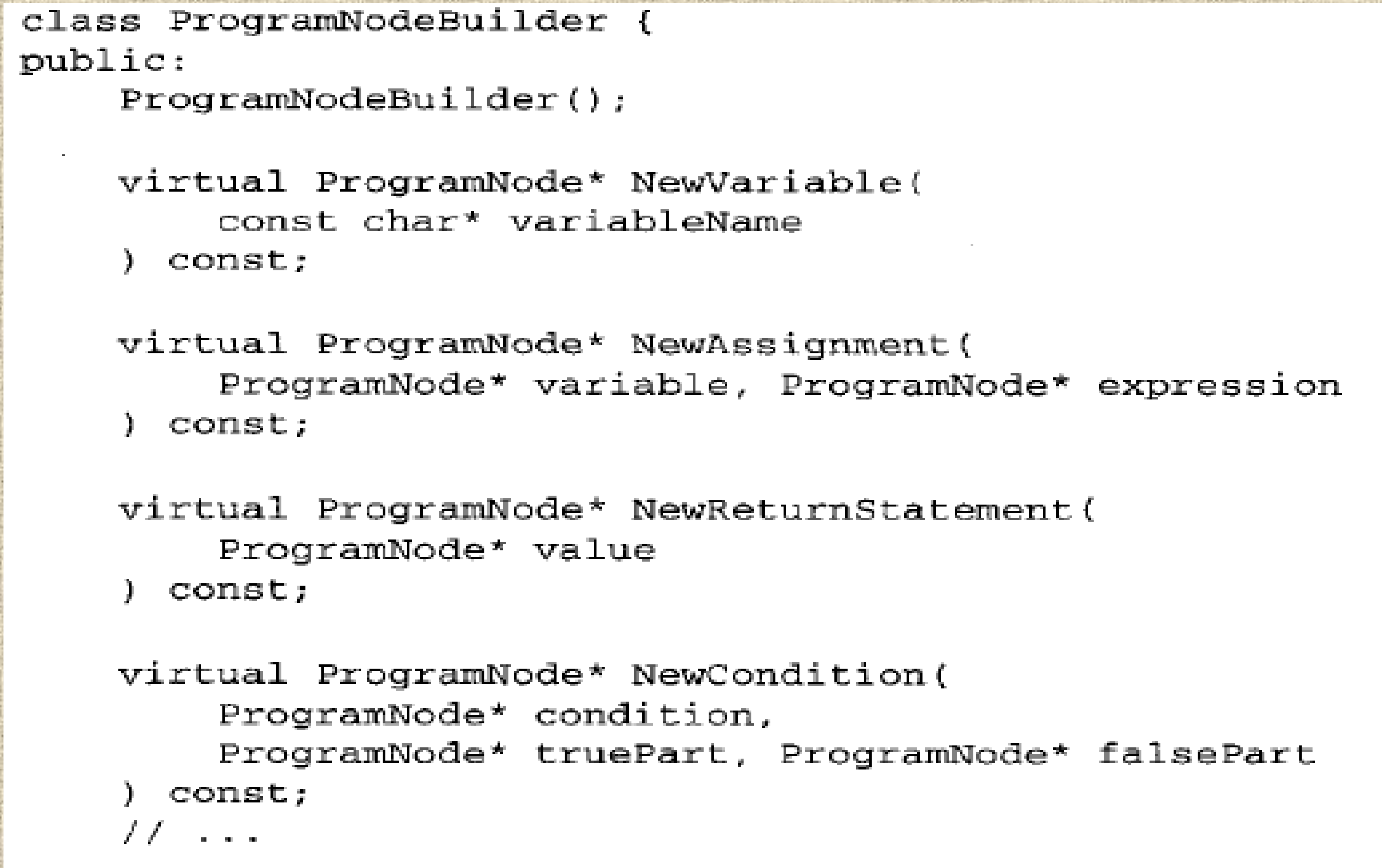
```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();

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

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



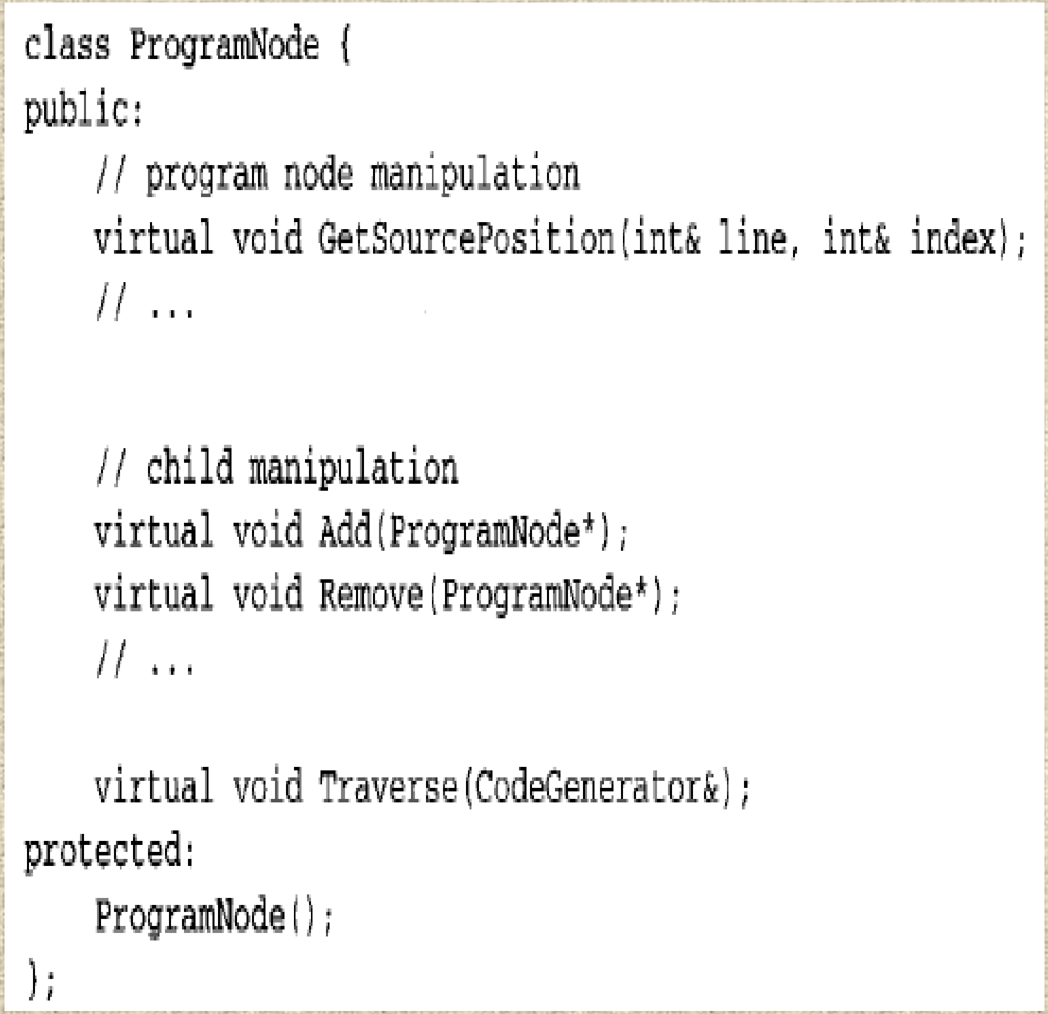
```
class Parser {
public:
    Parser();
    virtual ~Parser();

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

Parser calls back on ProgramNodeBuilder to build the parse tree incrementally. These classes interact according to the Builder (97) pattern.

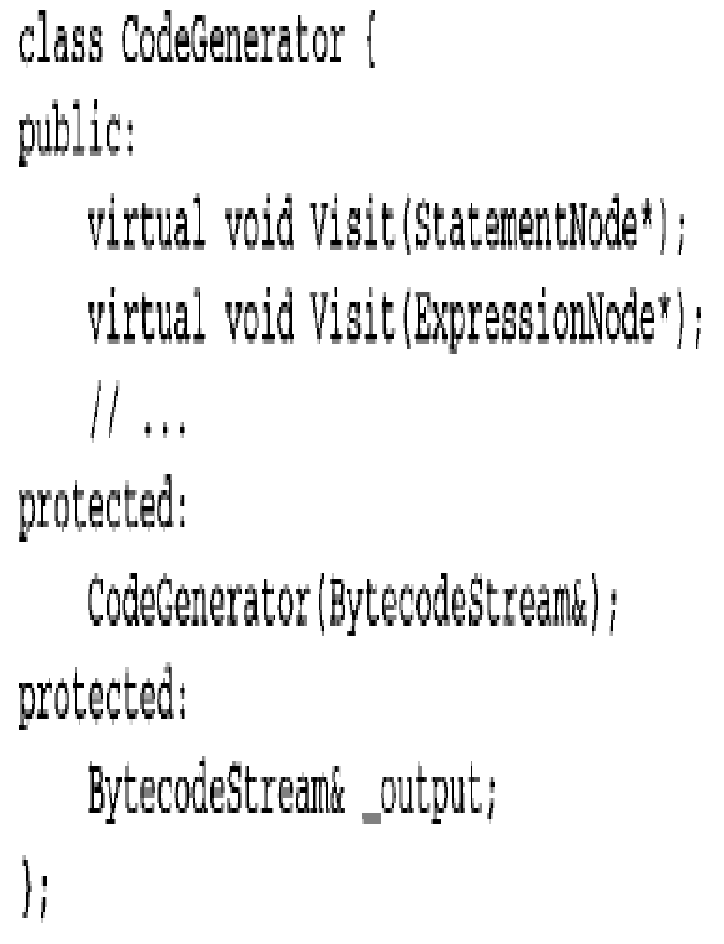


```
class ProgramNodeBuilder {
public:
    ProgramNodeBuilder();

    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;
    // ...
```

The parse tree is made up of instances of ProgramNode subclasses such as StatementNode, ExpressionNod e, and so forth



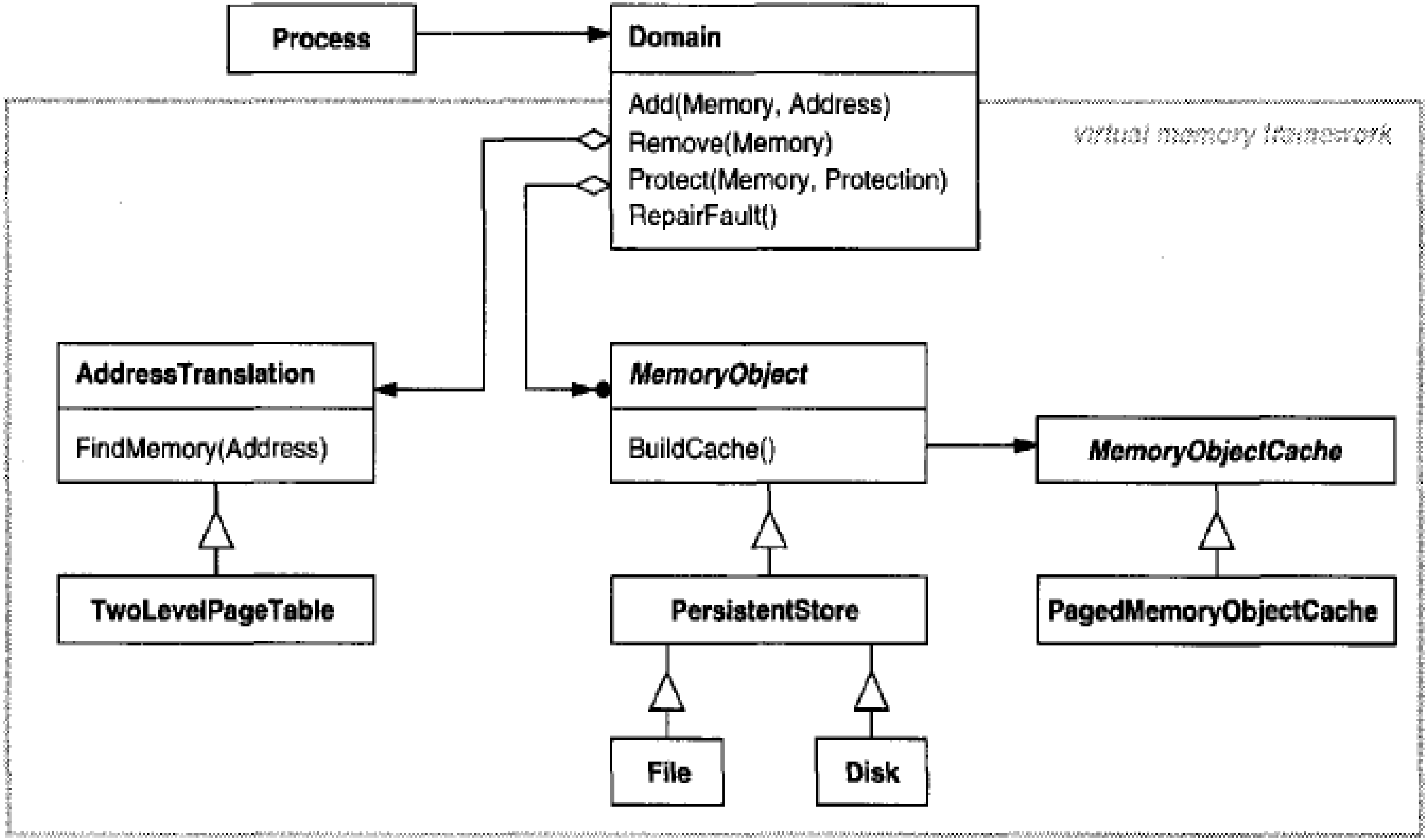
```
class ProgramNode {
public:
    // program node manipulation
    virtual void GetSourcePosition(int& line, int& index);
    // ...


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

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



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

Known Uses

- The compiler example in the Sample Code section was inspired by the Object- Works\Smalltalk compiler system
- In the ET++ application framework=> 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."
- The Choices operating system=> uses facades to compose many frameworks into one. The key abstractions in Choices are processes, storage, and address spaces.


Process
Domain
Add(Memory, Address)
Remove(Memory)
Protect(Memory, Protection)
RepairFault()
virtual memory framework
AddressTranslation
FindMemory(Address)
MemoryObject
BuildCache()
MemoryObjectCache
TwoLevelPageTable
PersistentStore
PagedMemoryObjectCache
File
Disk

Related Patterns

Abstract Factory (87)

- 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.

Mediator (273) is similar to Façade.

- Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one o f them.
- A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly.
- In contrast, a façade 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.
- Thus Facade objects are often Singletons(127).

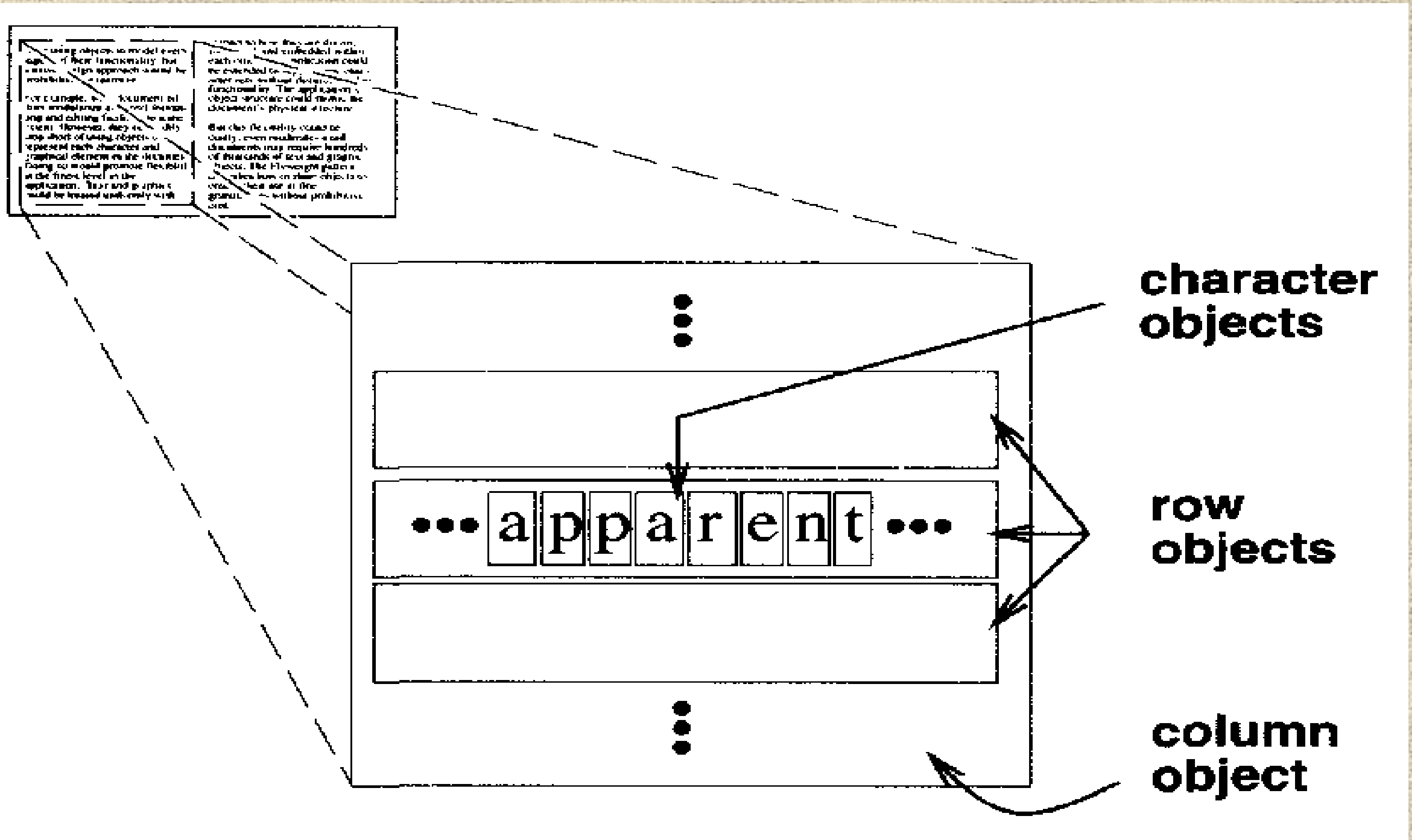
FLYWEIGHT is a Object structural Pattern:

Intent :

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

Motivation :

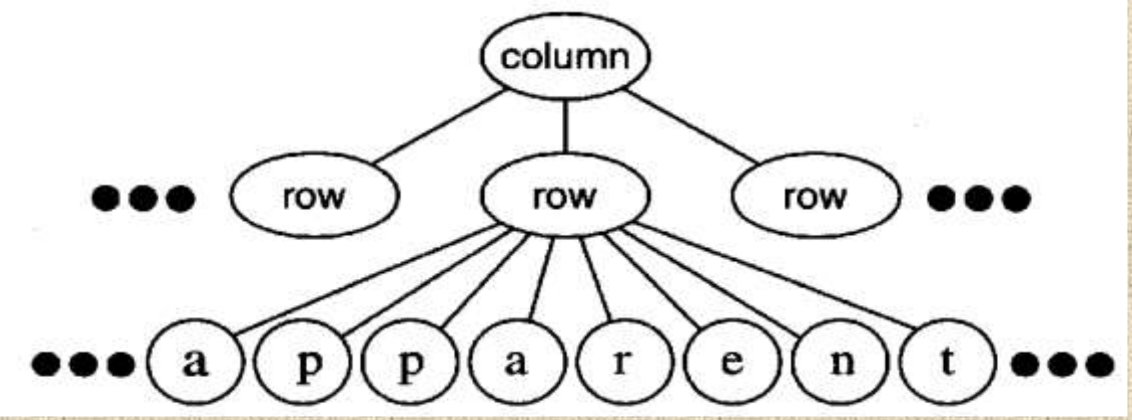
- Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.
- **For example,** Document editor implementations have text formatting and editing facilities that are modularized to some extent.
  - ✓ Object-oriented document editors typically use objects to represent embedded elements like tables and figures.
  - ✓ Uses an object for each character in the document=>Promotes flexibility at the finest levels in the application.
  - ✓ Characters and embedded elements could then be treated uniformly with respect to how they are drawn and formatted.
  - ✓ The application could be extended to support new character sets without disturbing other functionality.
  - ✓ The application's object structure could mimic the document's physical structure.



The drawback of such a design is its cost.

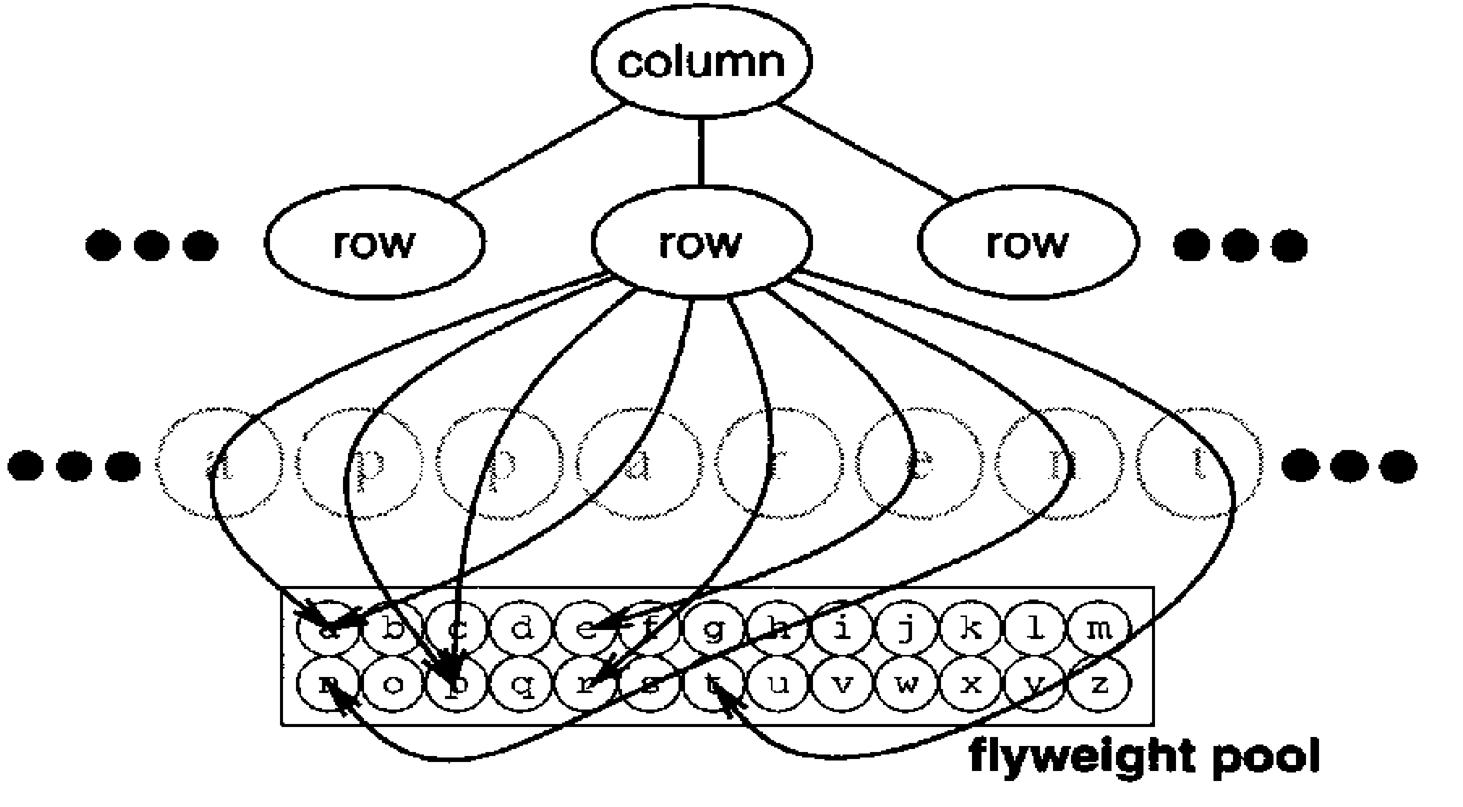
- Even moderate-sized documents may require hundreds of thousands of character objects, which will consume lots of memory and may incur unacceptable run-time overhead.
- The Flyweight pattern describes how to share objects to allow their use at fine granularities without prohibitive cost.

- **A flyweight** is a shared object that can be used in multiple contexts simultaneously.
- The flyweight acts as an independent object in each context—it's indistinguishable from an instance of the object that's not shared.
- Flyweights cannot make assumptions about the context in which they operate.
- The key concept 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, 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.



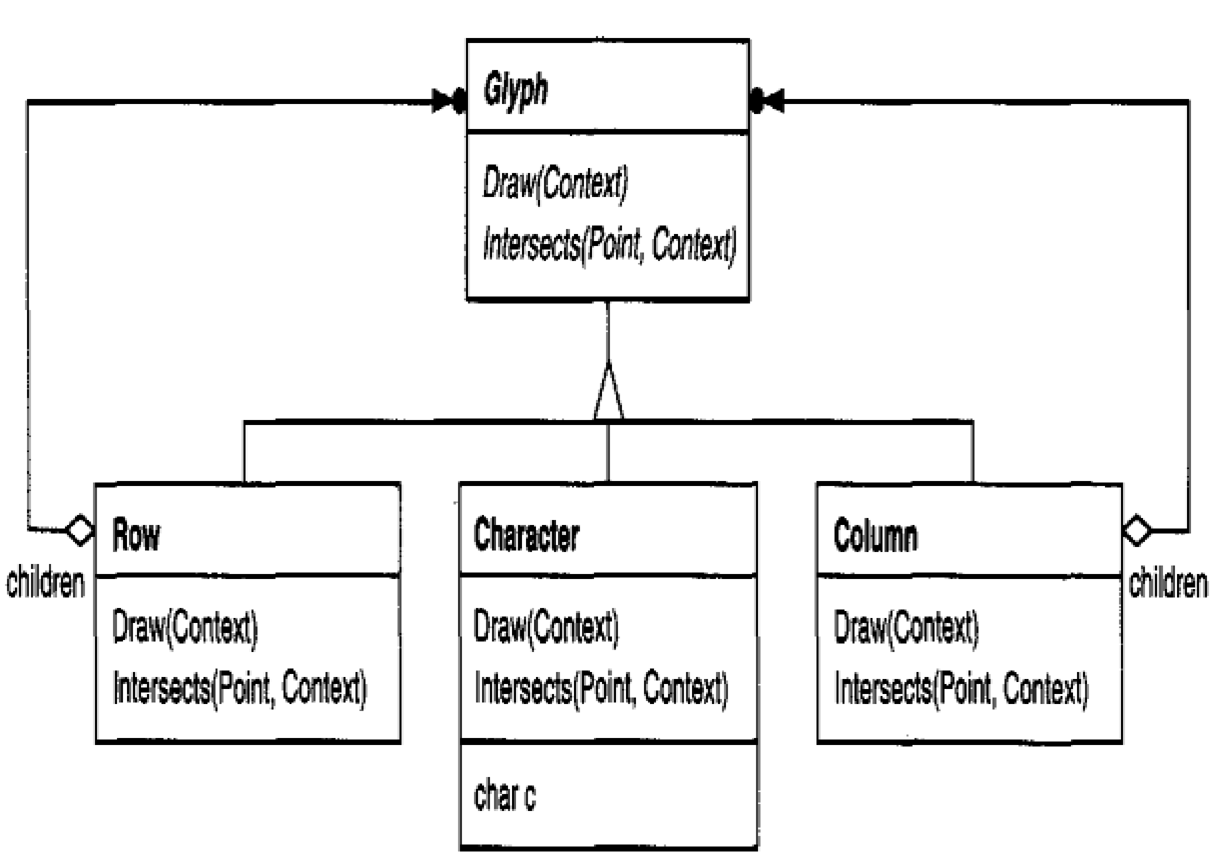
- ➢ A document editor can create a flyweight for each letter of the alphabet.
- ➢ Each flyweight stores a character code, but its coordinate position in the document and its typographic style can be determined from the text layout algorithms and formatting commands in effect wherever the character appears.
- ➢ 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

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



Class Structure

- 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.


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

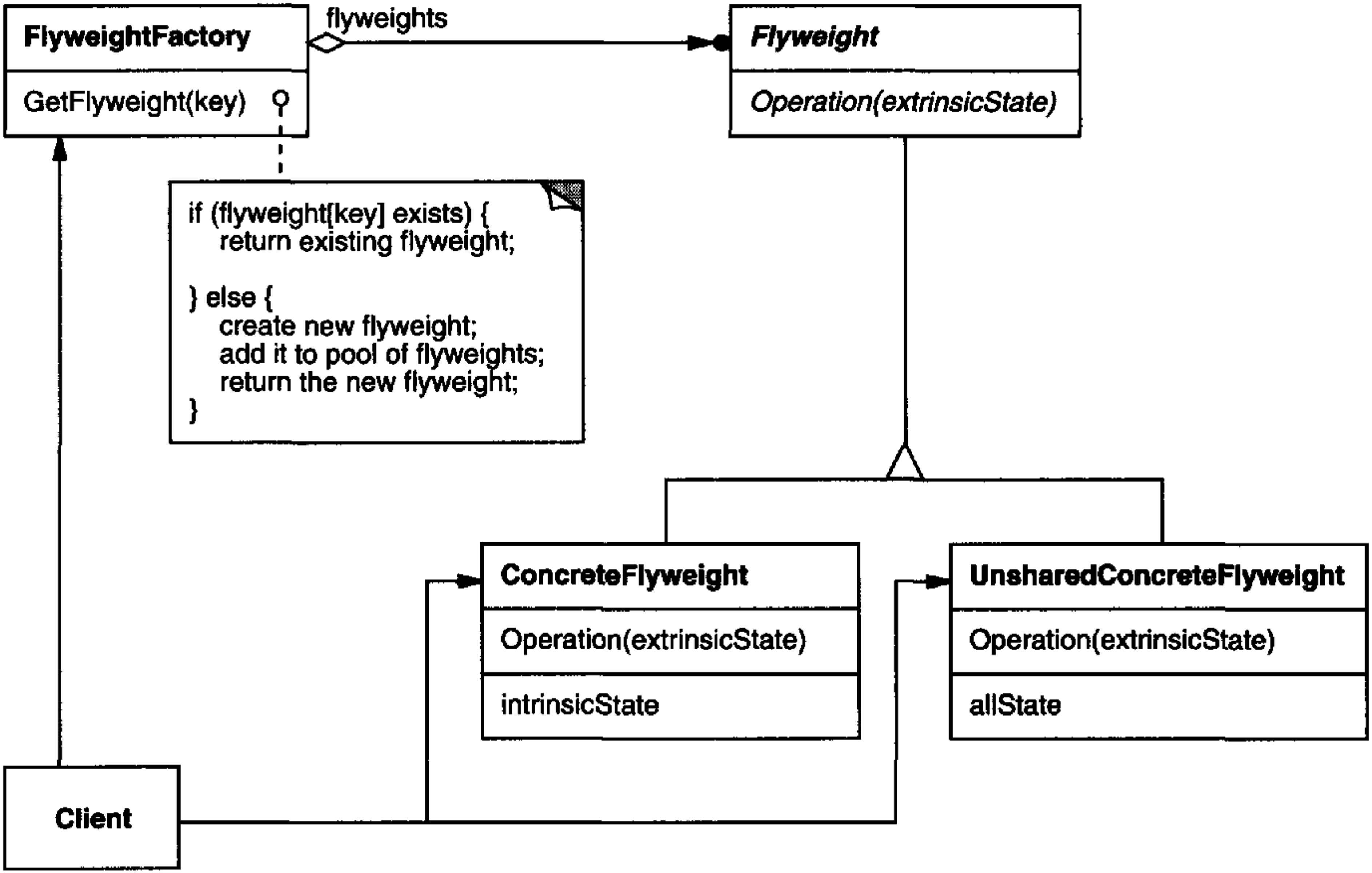
- A flyweight representing the letter "a" only stores the corresponding character code;
- it doesn't need to store its location or font.
- Clients supply the context dependent information that the flyweight needs to draw itself

Applicability:

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

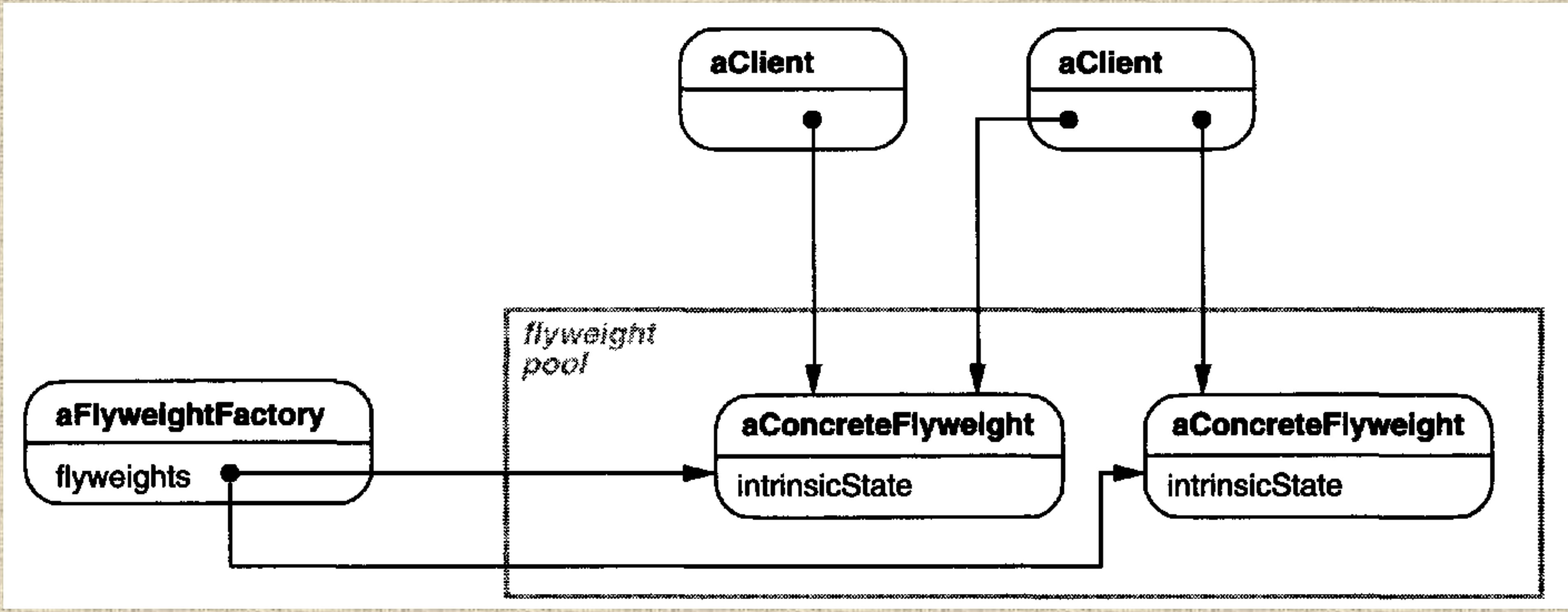
- An application uses a large number of objects.
- Storage costs are high because of the sheer quantity of objects.
- Most object state can be made extrinsic.
- Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
- The application doesn't depend on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

Structure


flyweights
FlyweightFactory
GetFlyweight(key)
Flyweight
Operation(extrinsicState)
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

```
if (flyweight[key] exists) {
    return existing flyweight;

} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
```

The following diagram shows how the 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 Flyweight interface and adds storage for intrinsic state, if any.
   - A ConcreteFlyweight object must be sharable.
   - Any state it stores must be intrinsic; it must be independent of the ConcreteFlyweight object's context.
3. **UnsharedConcreteFlyweight (Row ,Column)**
   - not all Flyweight subclasses need to be shared.
   - The Flyweight interface *enables* sharing; it doesn't enforce it.
   - It's common for UnsharedConcrete- Flyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).

Participants contd…

4. FlyweightFactory

- creates and manages flyweight objects.
- ensures that flyweights are shared properly. When a client requests a flyweight,
- the FlyweightFactory object supplies an existing instance or creates one, if none exists.

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.
- 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
- *W*hether extrinsic state is computed or stored.

The more flyweights are shared, the greater the storage savings. The savings increase with the amount of shared state.

Implementation

Consider the following issues when implementing the Flyweight pattern:

1. *Removing extrinsic state.*
2. *Managing shared objects.*

1. *Removing extrinsic state.*

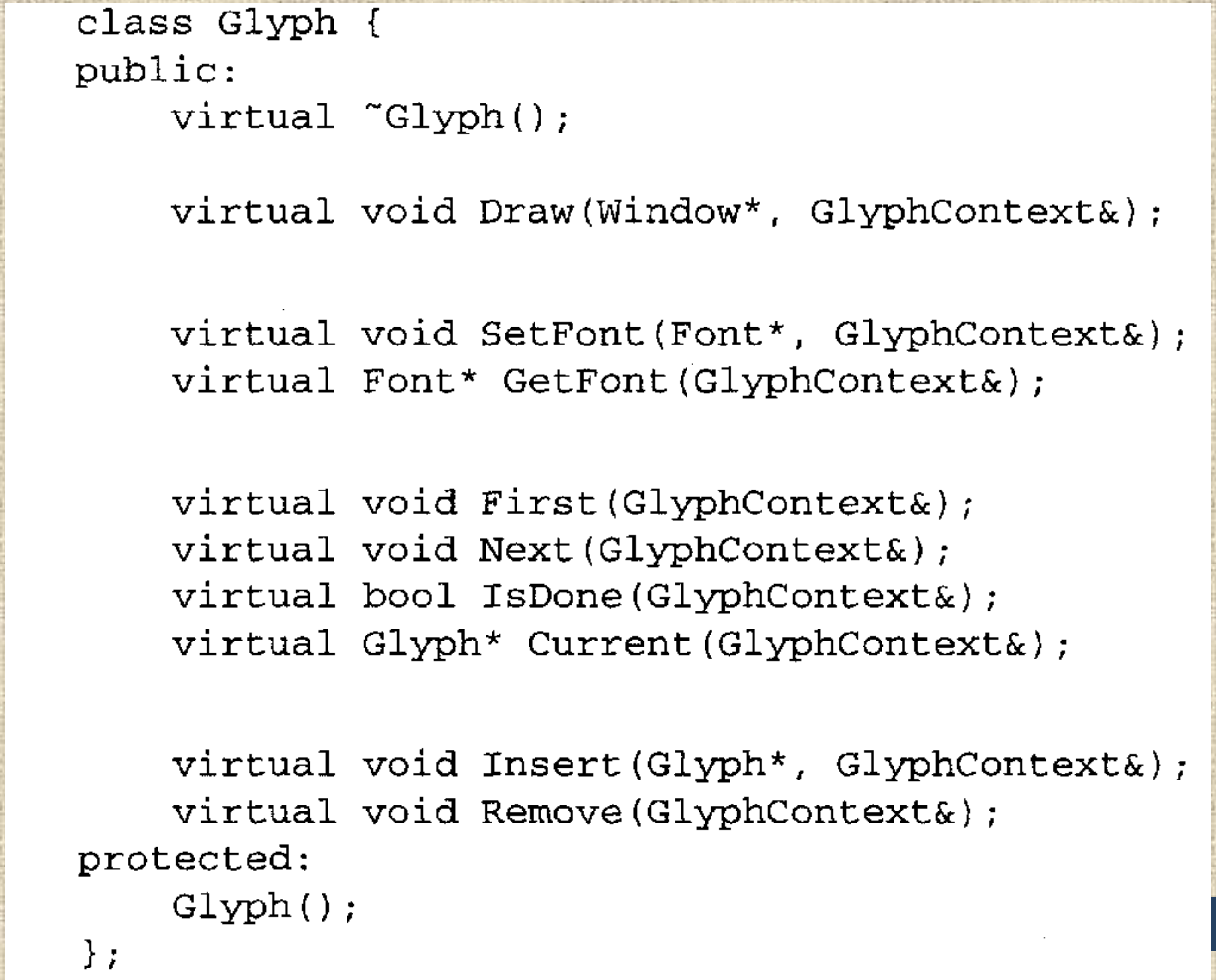
- Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing.
- Ideally, extrinsic state can be computed from a separate object structure, one with far smaller storage requirements.
- for example, we can store a map of typographic information in a separate structure rather than store the font and type style with each character object.
- When a character draws itself, it receives its typographic attributes as a side-effect of the draw traversal
- Storing this information externally to each character object is far more efficient than storing it internally.

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.
- For example, the flyweight factory in the document editor can keep a table of flyweights indexed by character codes.
- The manager returns the proper flyweight given its code, creating the flyweight if it does not already exist.
- Sharability also implies some form of reference counting or garbage collection to reclaim a flyweight's storage when it's no longer needed.
- If the number of flyweights is fixed and small=> flyweights are worth keeping around permanently.

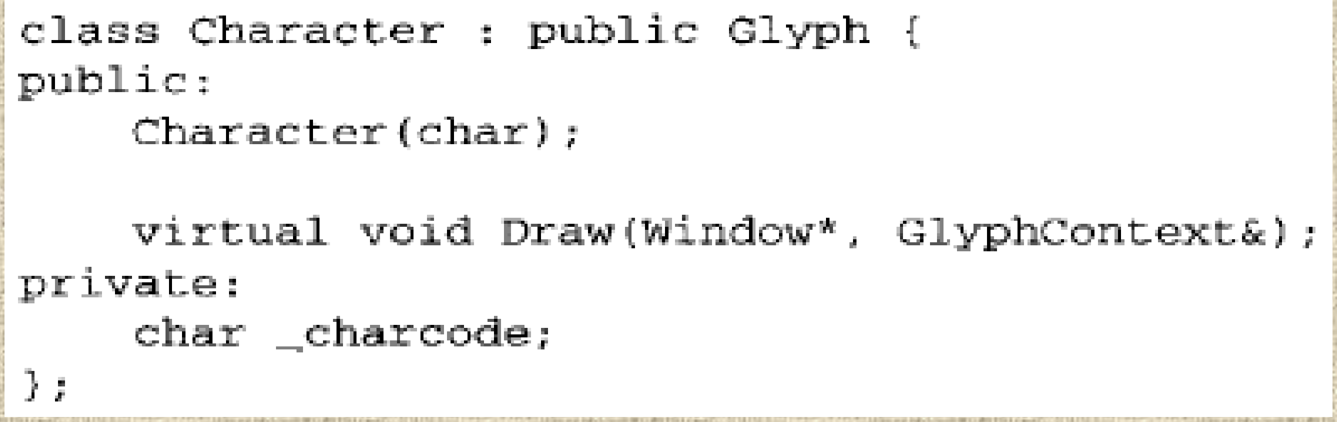
Sample Code

- Logically, glyphs (base class)are Composites that have graphical attributes and can draw themselves.



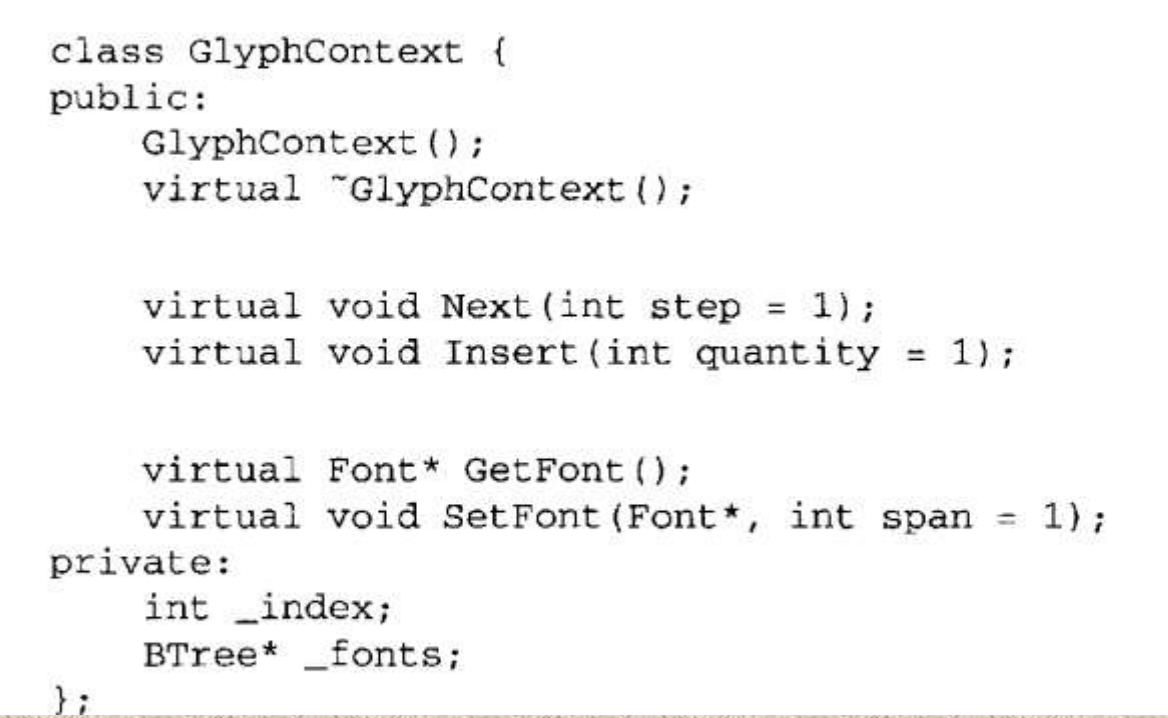
```
class Glyph {
public:
    virtual ~Glyph();

    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;
};
```

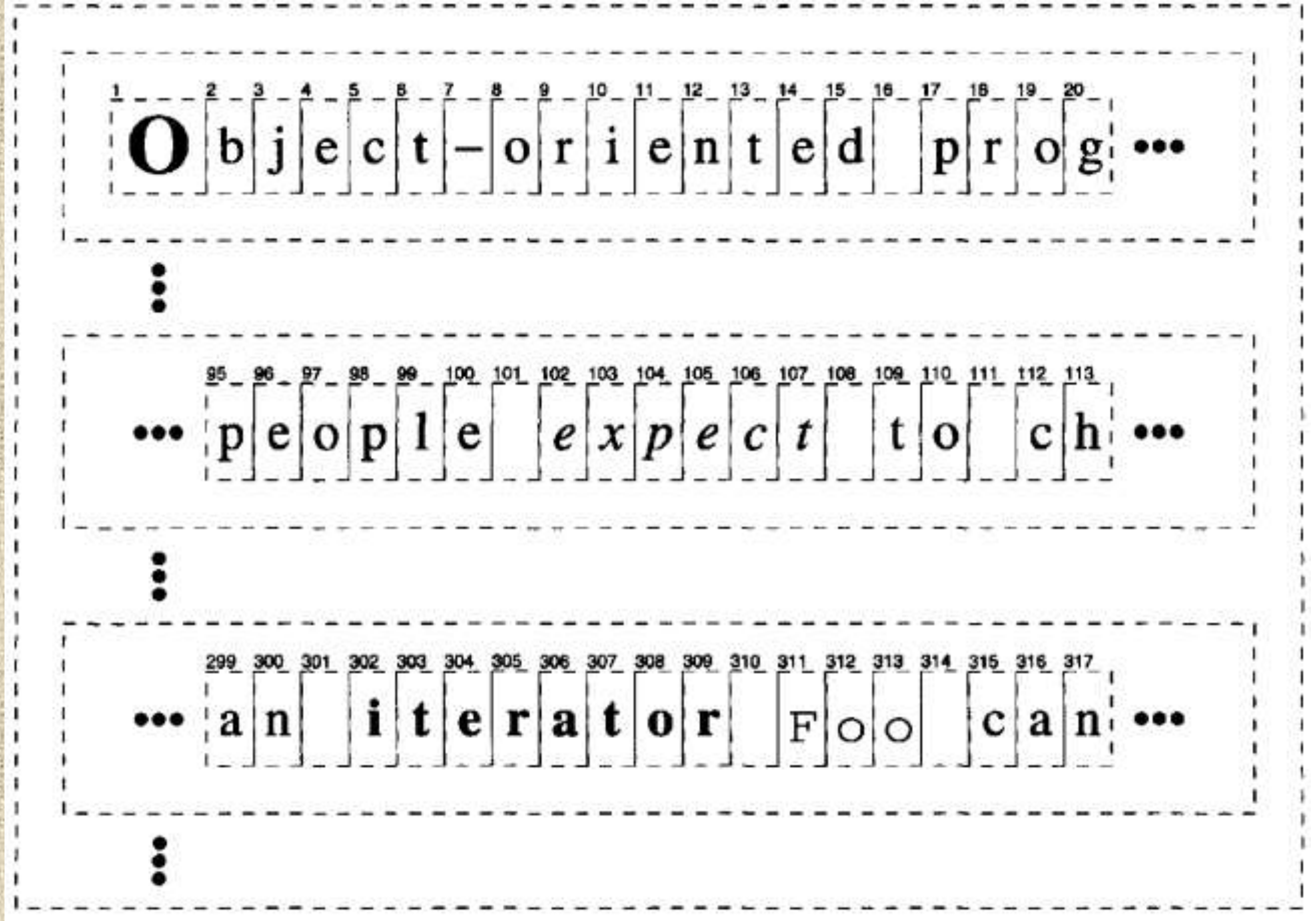
- To keep from allocating space for a font attribute in every glyph, we'll store the attribute extrinsically in a GlyphContext object.
- GlyphContext acts as a repository of extrinsic state.
- Any operation that needs to know the glyph's font in a given context will have a GlyphContext instance passed to it as a parameter. The
- operation can then query the GlyphContext for the font in that context.
- The context depends on the glyph's location in the glyph structure.
- Therefore Glyph's child iteration and manipulation operations must update the GlyphContext whenever they're used.



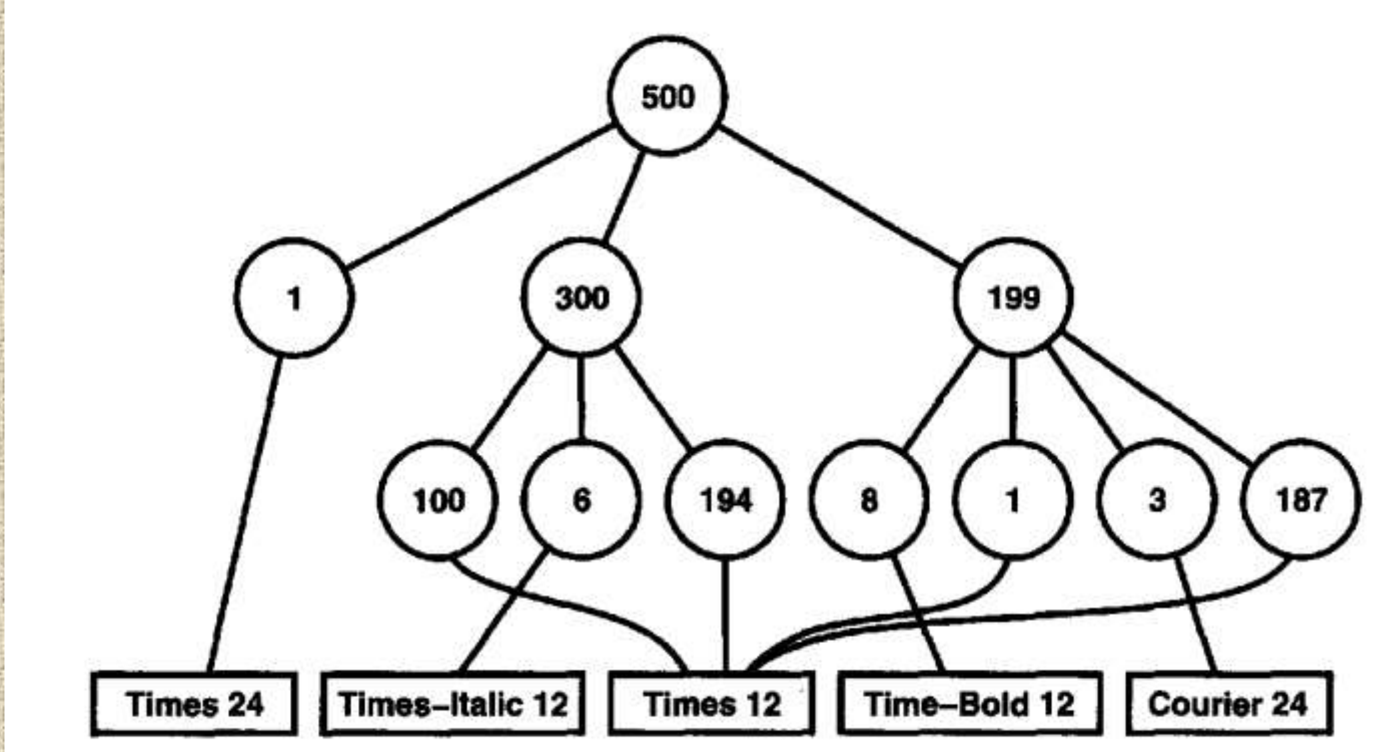
```
class GlyphContext {
public:
    GlyphContext();
    virtual ~GlyphContext();

    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.

Consider the following excerpt from a glyph composition:

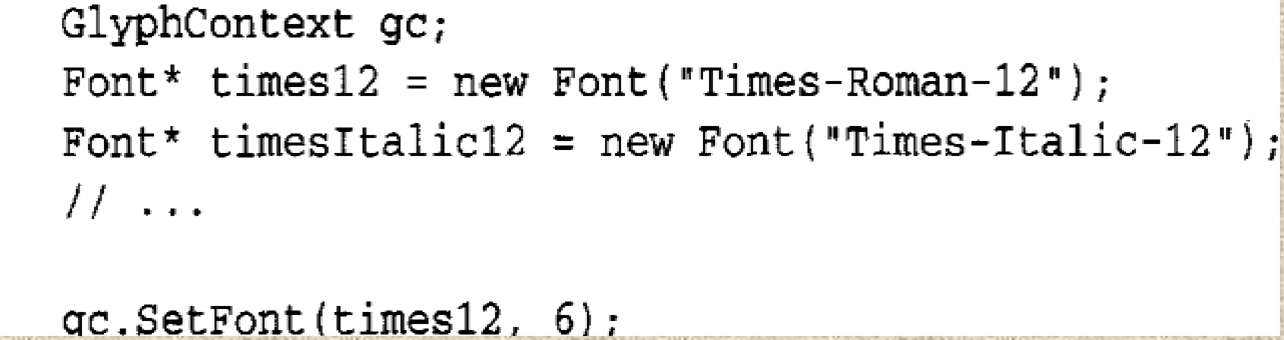


The BTree structure for font information might look like

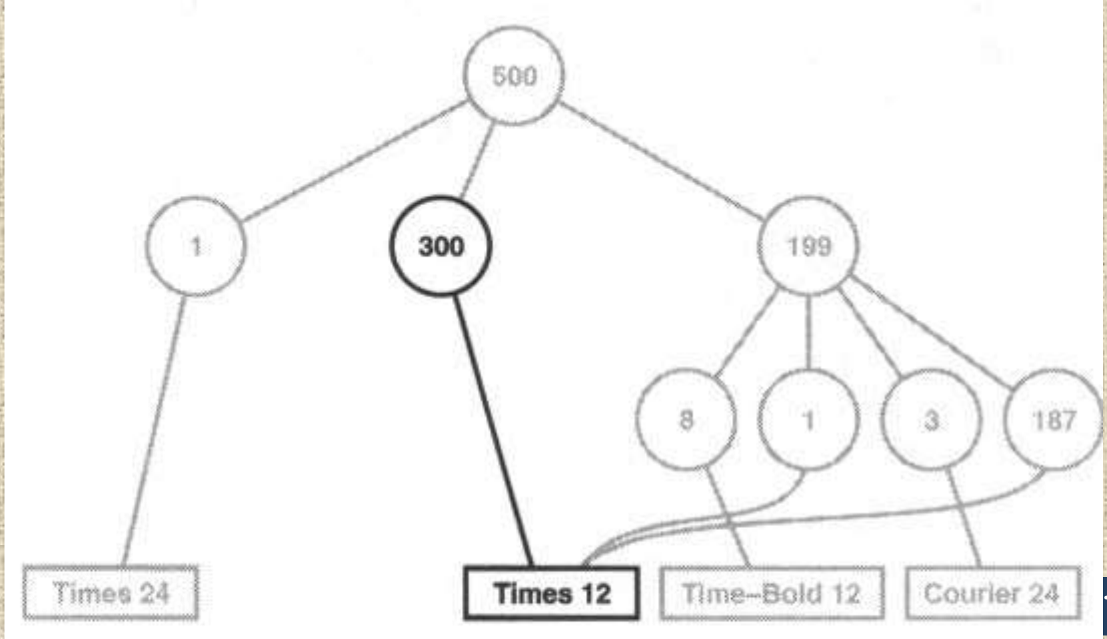


- Interior nodes define ranges of glyph indices.
- BTree is updated in response to font changes and whenever glyphs are added to or removed from the glyph structure.

For example, assuming we're at index 102 in the traversal, the following code sets the font of each character in the word "expect" to that of the surrounding text (that is, times12, an instance of Font for 12-point Times Roman):

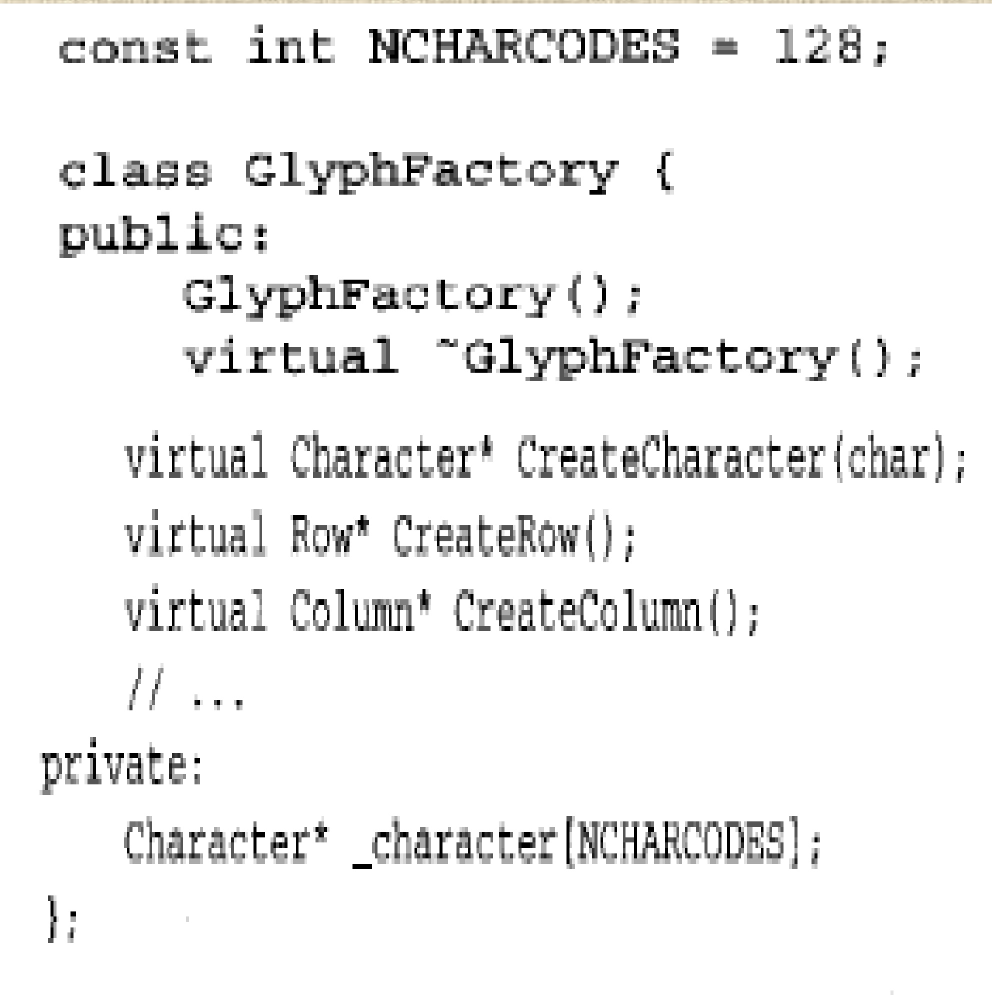


```
GlyphContext gc;
Font* times12 = new Font("Times-Roman-12");
Font* timesItalic12 = new Font("Times-Italic-12");
// ...

gc.SetFont(times12, 6);
```

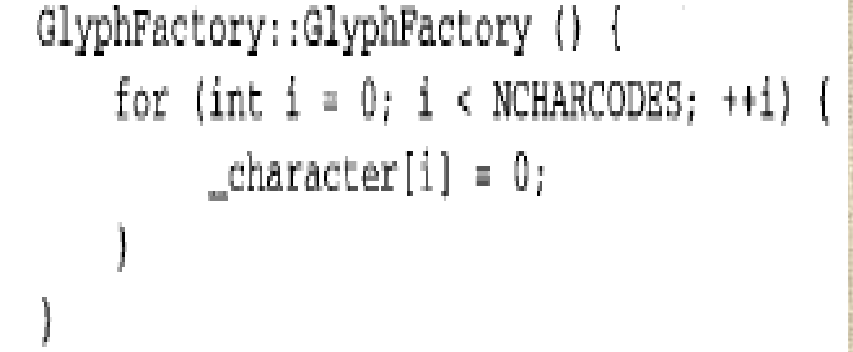


The new BTree structure (with changes shown in black) looks like

Class Glyph Factory instantiates Character and other kinds of glyphs.



```
const int NCHARCODES = 128;

class GlyphFactory {
public:
      GlyphFactory();
      virtual ~GlyphFactory();

    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...
private:
    Character* _character[NCHARCODES];
};
```



```
GlyphFactory::GlyphFactory () {
    for (int i = 0; i < NCHARCODES; ++i) {
        _character[i] = 0;
    }
}
```

Known Uses

1. **Interviews 3.0:**
   - Its developers built a powerful document editor called Doc as a proof of concept [CL92]. Doc uses glyph objects to represent each character in the document.
   - Only position is extrinsic, making Doc fast. Documents are represented by a class Document, which also acts as the FlyweightFactory.

2. **ET++:**
   - uses flyweights to support look-and-feel independence.
   - The look-and-feel standard affects the layout of user interface elements.
   - A widget delegates all its layout and drawing behavior to a separate Layout object. Changing the Layout object changes the look and feel, even at run-time.

Related Patterns

- The **Flyweight pattern** is often combined with the **Composite (163) pattern** to implement a **logically hierarchical structure** in terms of *a* **directed-acyclic graph** with **shared leaf nodes.**
- It's often best to implement State (305) and Strategy (315) objects as flyweights

PROXY: a Object Structural pattern

Intent

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

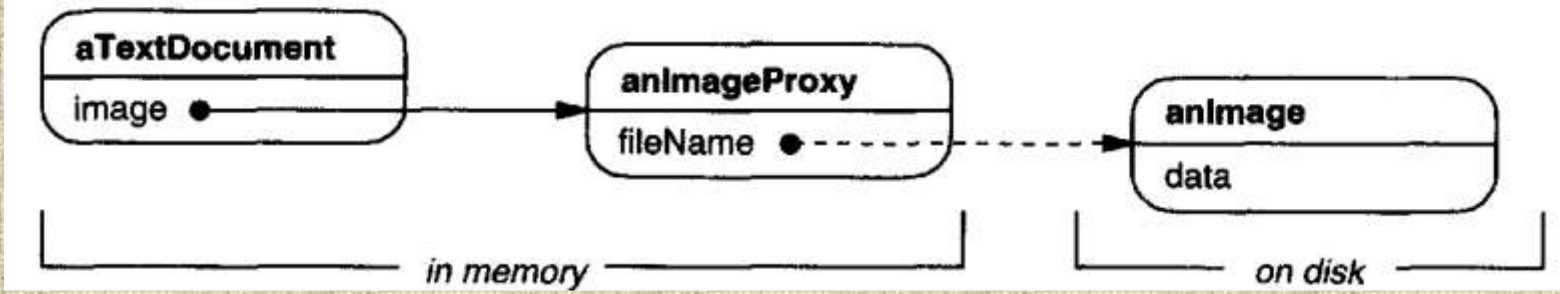
Also Known As

Surrogate

Motivation

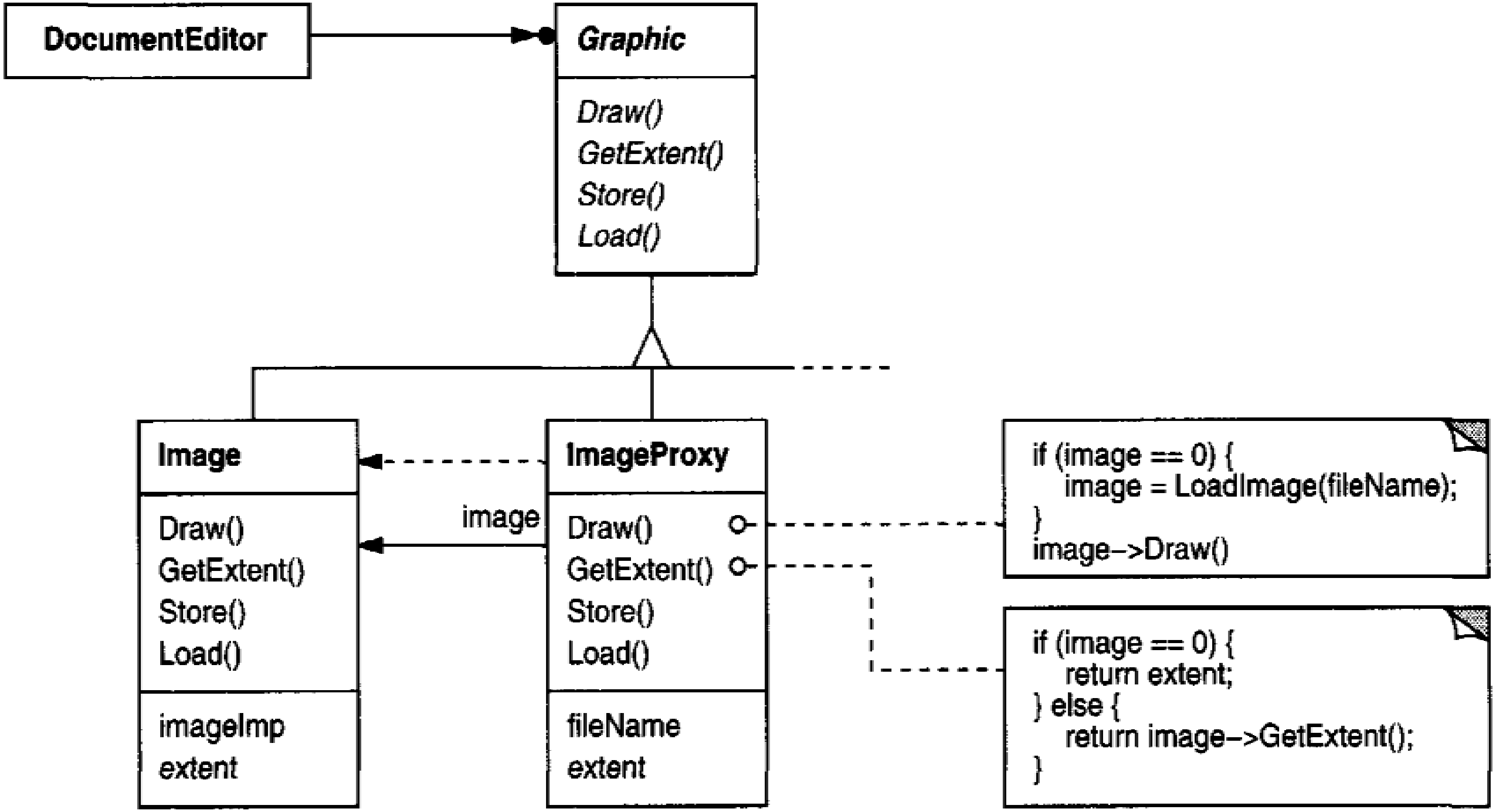
- One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it.
- **Example:** Consider a document editor that can embed graphical objects in a document.
- Some graphical objects, like large raster images, can be expensive 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.
- Because not all of these objects will be visible in the document at the same time.
- creating each expensive object on demand, which in this case occurs when an image becomes visible.

Solution: 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.
- 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.



```
if (image == 0) {
    image = LoadImage(fileName);
}
image->Draw()
```

```
if (image == 0) {
    return extent;
} else {
    return image->GetExtent();
}
```

- The document editor accesses embedded images through the interface defined by the abstract Graphic class.
- ImageProxy is a class for images that are created on demand.
- ImageProxy maintains the file name as a reference to the image on disk.
- The file name is passed as an argument to the ImageProxy constructor.
- ImageProxy also stores the bounding box of the image and a reference to the real Image instance.
- This reference won't be valid until the proxy instantiates the real image.
- The Draw operation makes sure the image is instantiated before forwarding it the request.
- GetExtent forwards the request to the image only if it's instantiated; otherwise ImageProxy returns the extent it stores.

Applicability

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

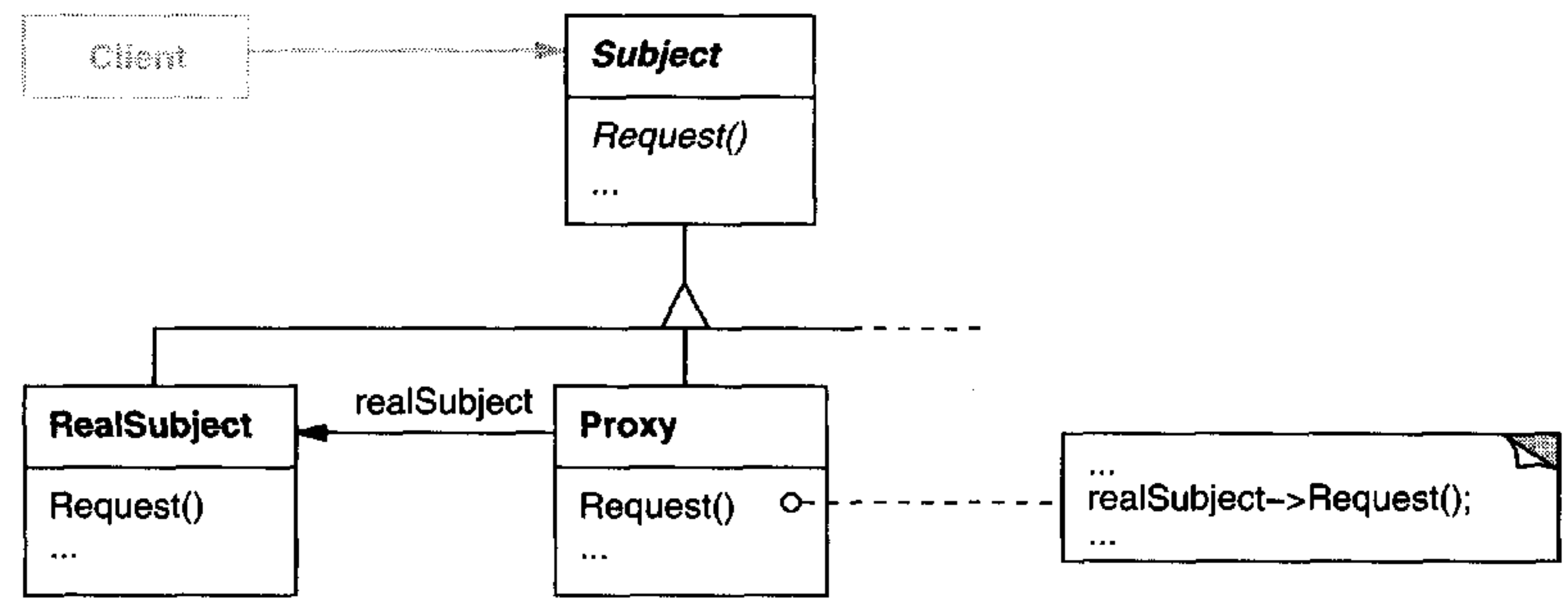
Common situations in which the Proxy pattern is applicable:

1. A **remote proxy** provides a local representative for an object in a different address space. NEXTSTEP [Add 94]uses the class NXProxy for this purpose. Coplien [Cop 92]calls this kind of proxy an "Ambassador.“
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.

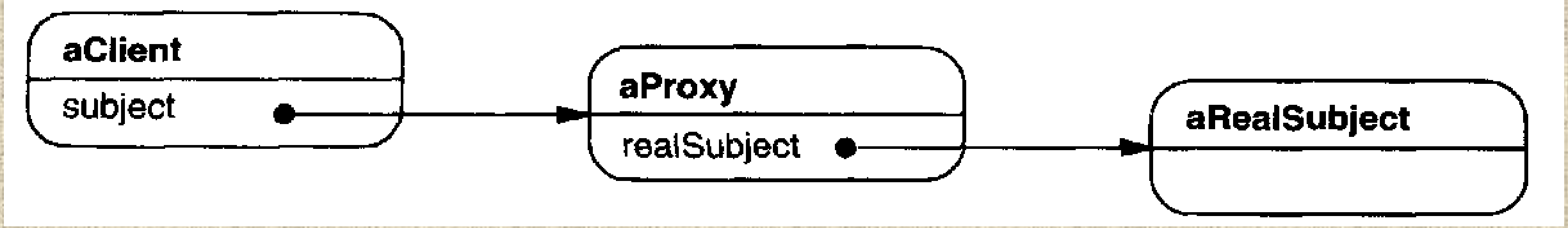
Ex: KernelProxies in the Choices operating system [CIRM93] provide protected access to operating system objects.

4. A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed. **Typical uses include**
   - Counting the number of references to the real object so that it can be freed automatically when there are no more references.
   - 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



Object diagram of a proxy structure at run-time



Participants

1. **Proxy (ImageProxy)**
   - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
   - Provides an interface identical to Subject's so that a proxy can by substituted for the real subject.
   - Controls access to the real subject and may be responsible for creating and deleting it.

Other responsibilities depend on the kind of proxy:

- *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
- *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real image's extent.
- *protection proxies* check that the caller has the access permissions required to perform a request.

2. **Subject** (Graphic)
    - 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.

There's another optimization that the Proxy pattern can hide from the client. It's called **copy-on-write ,** and it's related to creation on demand.

- There's another optimization that the Proxy pattern can hide from the client. It's called **copy-on-write,** and it's related to creation on demand.
- Copying a large and complicated object can be an expensive operation.
- If the copy is never modified, then there's no need to incur this cost.
- By using a proxy to postpone the copying process, we ensure that we pay the price of copying the object only if it's modified.
- To make copy-on-write work, the subject must be reference counted.
- Copying the proxy will do nothing more than increment this reference count. Only when the client requests an operation that modifies the subject does the proxy actually copy it.
- In that case the proxy must also decrement the subject's reference count.
- When the reference count goes to zero, the subject gets deleted.

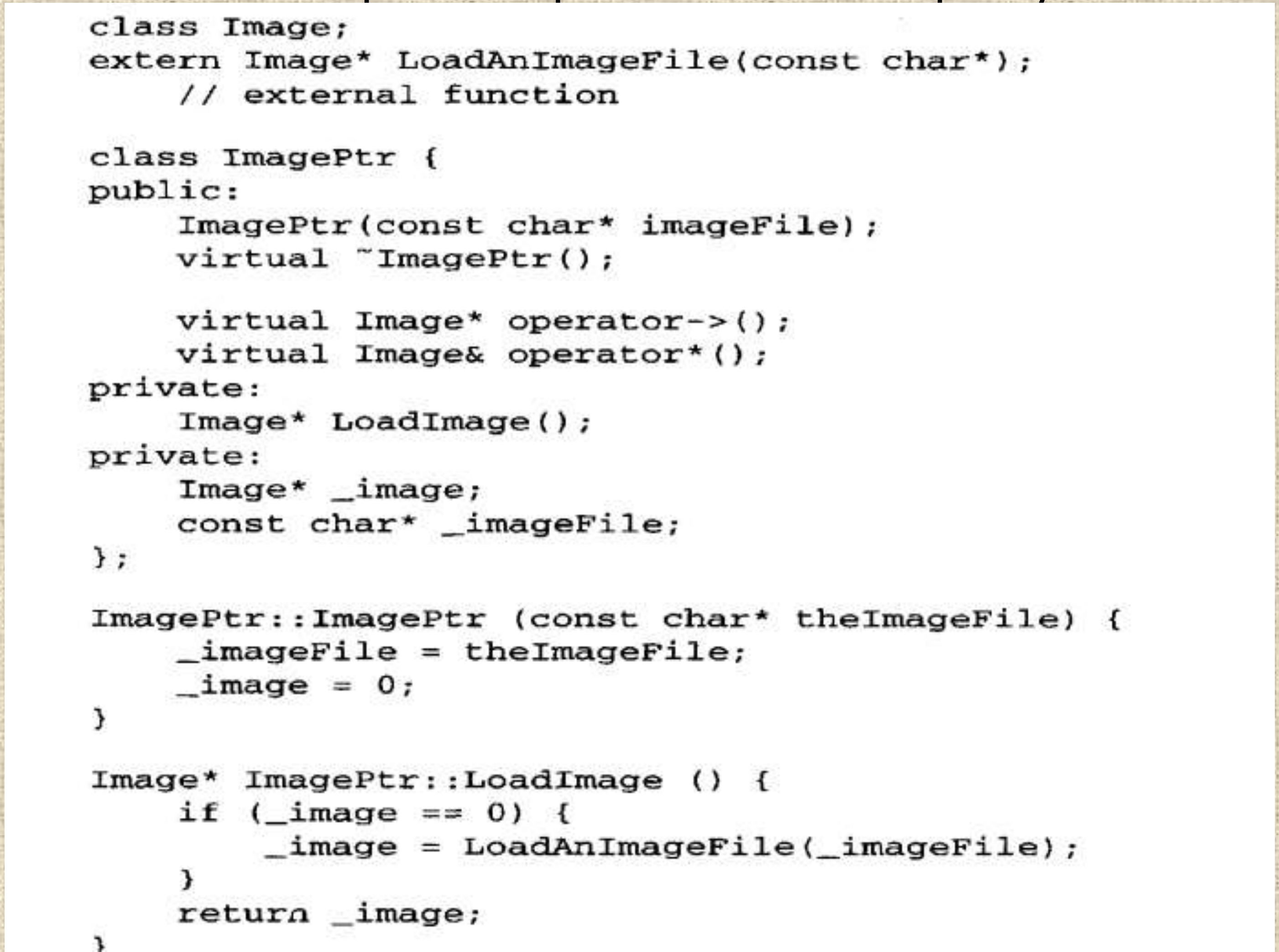
Implementation

The Proxy pattern can exploit the following language features:

1. *Overloading the member access operator in C++.*
2. *Using doesNotUnderstand in Smalltalk*
3. *Proxy doesn't always have to know the type of real subject.*

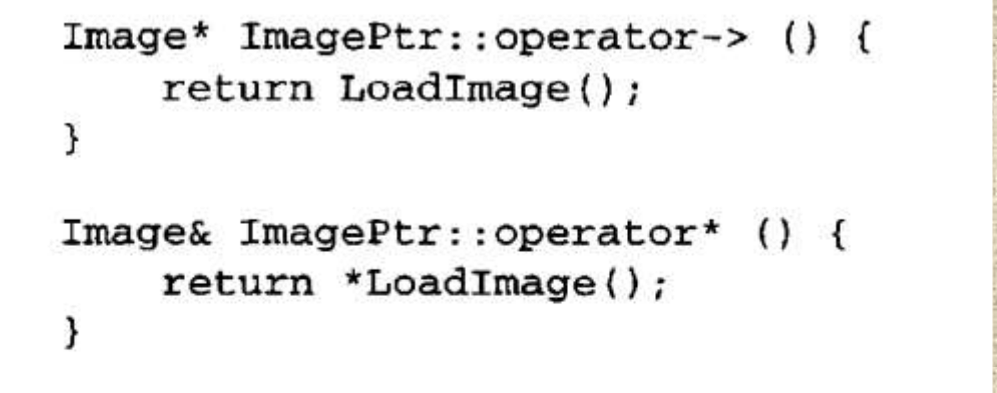
1. Overloading the member access operator in C++.

- *C++* supports overloading operator->, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced.
- use this technique to implement a virtual proxy called ImagePtr.



```
class Image;
extern Image* LoadAnImageFile(const char*);
    // external function

class ImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr();

    virtual Image* operator->();
    virtual Image& operator*();
private:
    Image* LoadImage();
private:
    Image* _image;
    const char* _imageFile;
};

ImagePtr::ImagePtr (const char* theImageFile) {
    _imageFile = theImageFile;
    _image = 0;
}

Image* ImagePtr::LoadImage () {
    if (_image == 0) {
        _image = LoadAnImageFile(_imageFile);
    }
    return _image;
}
```

The overloaded -> and * operators use LoadImage to return _image to callers (loading it if necessary).



```
Image* ImagePtr::operator-> () {
    return LoadImage();
}

Image& ImagePtr::operator* () {
    return *LoadImage();
}
```

2. Using doesNotUnderstand *in Smalltalk*

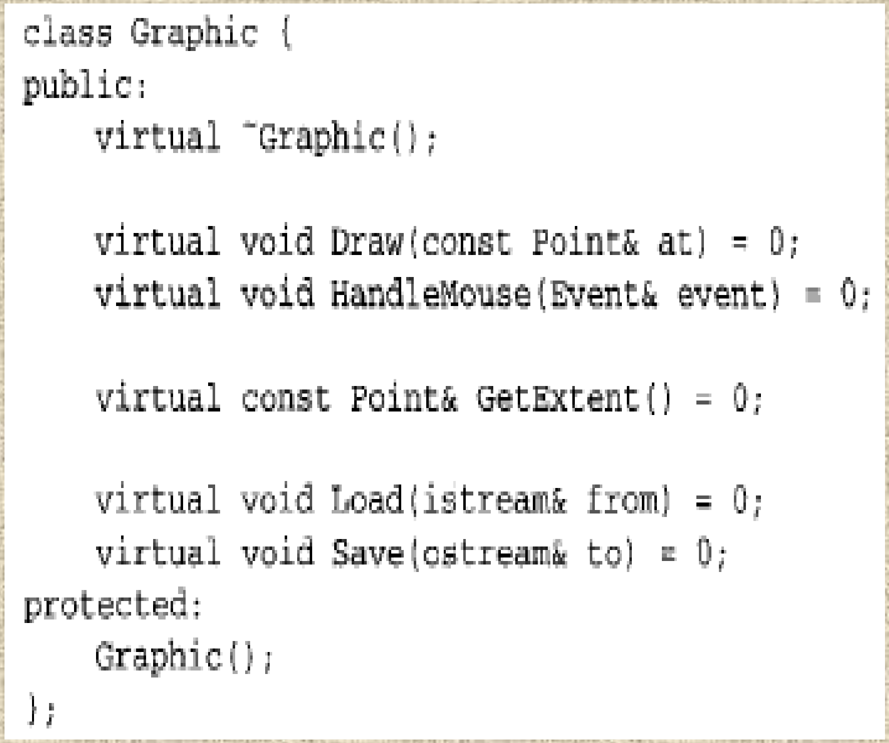
- Smalltalk calls doesNotUnderstand: aMessage when a client sends a message to a receiver that has no corresponding method.
- The Proxy class can redefine doesNotUnderstand so that the message is forwarded to its subject.
- To ensure that a request is forwarded to the subject and not just absorbed by the proxy silently, you can define a Proxy class that doesn't understand *any* messages.
- Smalltalk lets you do this by defining Proxy as a class with no superclass

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 t he concrete class.

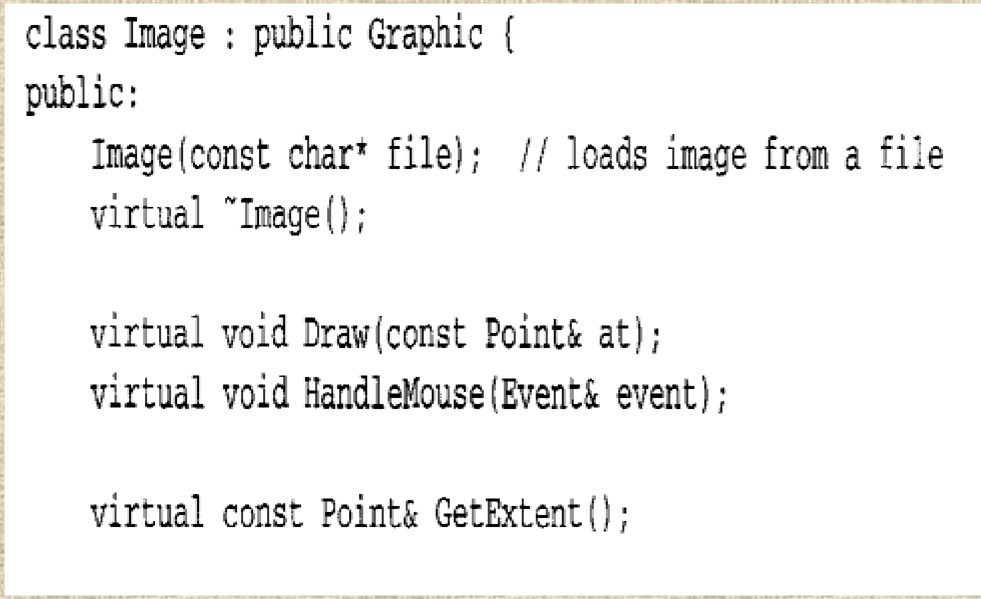
Sample Code

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



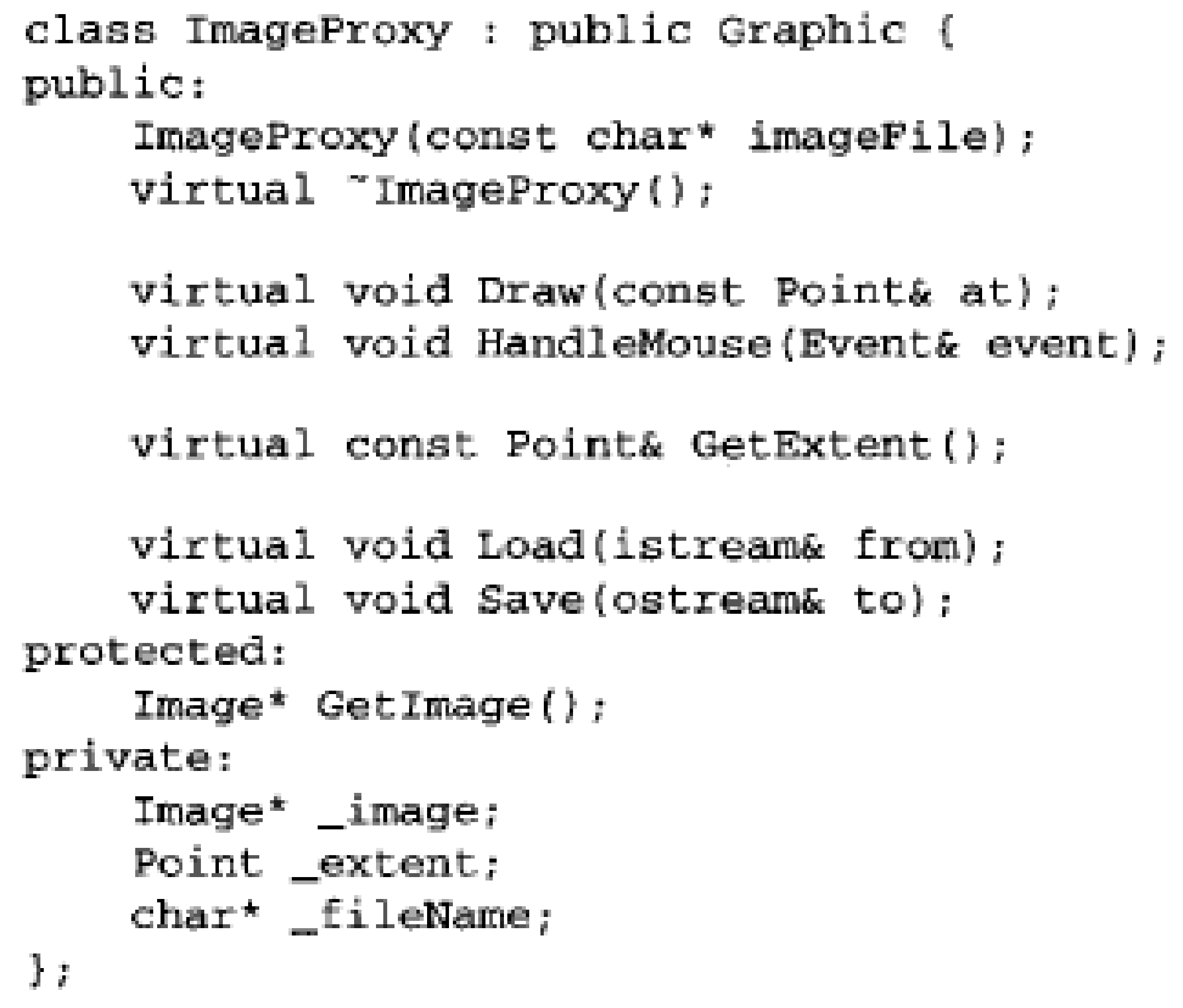
```
class Graphic {
public:
    virtual ~Graphic();

    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.



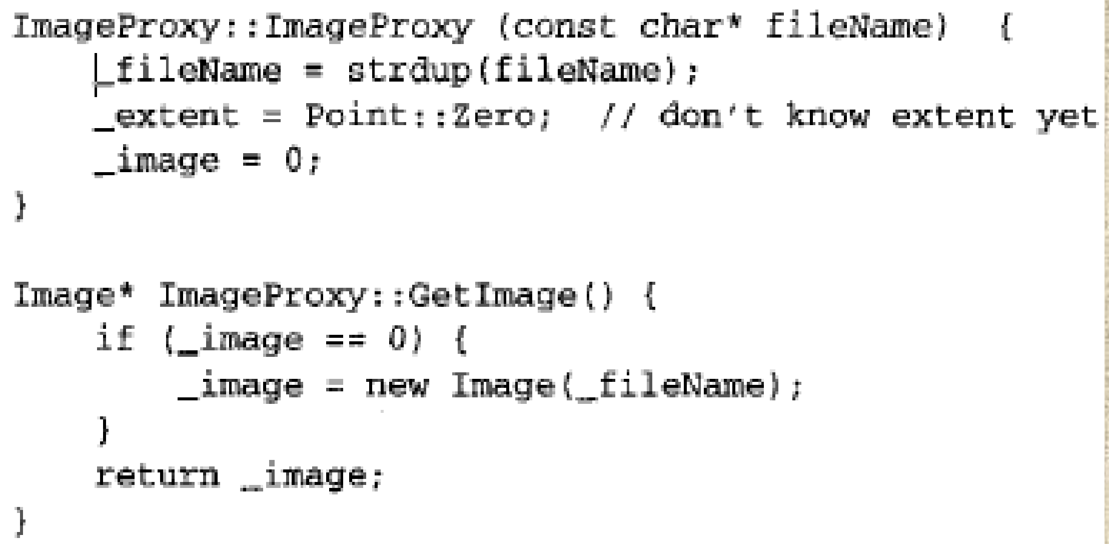
```
class Image : public Graphic {
public:
    Image(const char* file);  // loads image from a file
    virtual ~Image();

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

    virtual const Point& GetExtent();
```

ImageProxy has the same interface as Image:



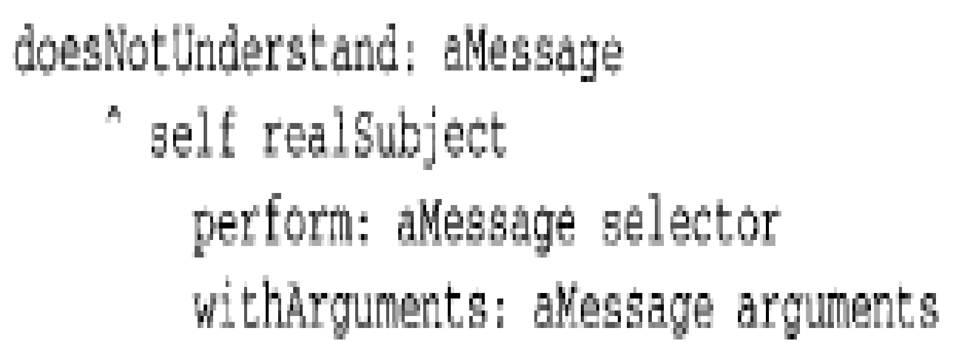
```
class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();

    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 saves a local copy of the name of the file that stores the image, and it initializes -extent and -image:



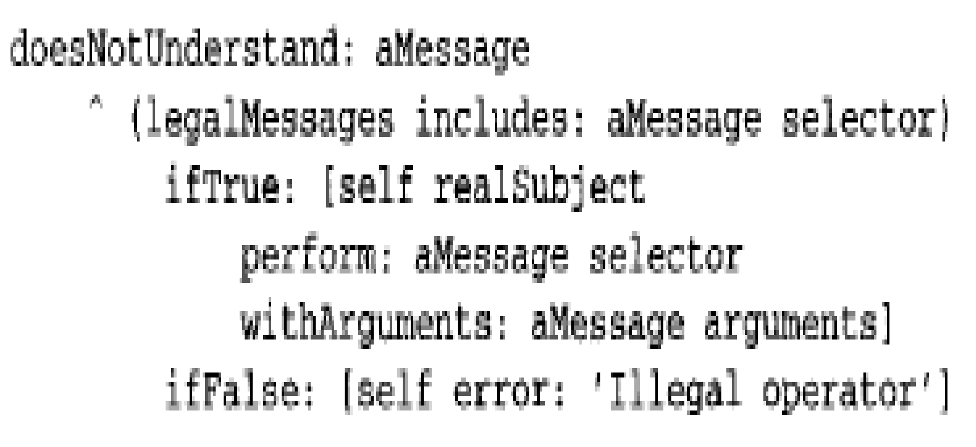
```
ImageProxy::ImageProxy (const char* fileName)  {
    _fileName = strdup(fileName);
    _extent = Point::Zero;  // don't know extent yet
    _image = 0;
}

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

***2. Proxies that use* doesNotUnderstand.** You can make generic proxies in Smalltalk by defining cl asses whose superclass is nil and defining the doesNotUnderstand: method to handle messages.



```
doesNotUnderstand: aMessage
    ^ self realSubject
        perform: aMessage selector
        withArguments: aMessage arguments
```

The following method assumes the proxy has a real Subject method that returns its real subject.



```
doesNotUnderstand: aMessage
    ^ (legalMessages includes: aMessage selector)
        ifTrue: [self realSubject
            perform: aMessage selector
            withArguments: aMessage arguments]
        ifFalse: [self error: 'Illegal operator']
```

Known Uses

- The virtual proxy example from ET++ text building block classes.
- NEXTSTEP: uses proxies (instances of class NXProxy)as local representatives for objects that may be distributed.
    - Server creates proxies for remote objects when clients request them.
    - On receiving a message, the proxy encodes it along with its arguments and then forward s the encoded message to the remote subject.

Related Patterns

1. **Adapter (139)** :
   - An adapter provides a different interface to the object it adapts. In contrast,
   - a proxy provides the same interface as its subject.
   - 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. **Decorator (175):**
   - 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.

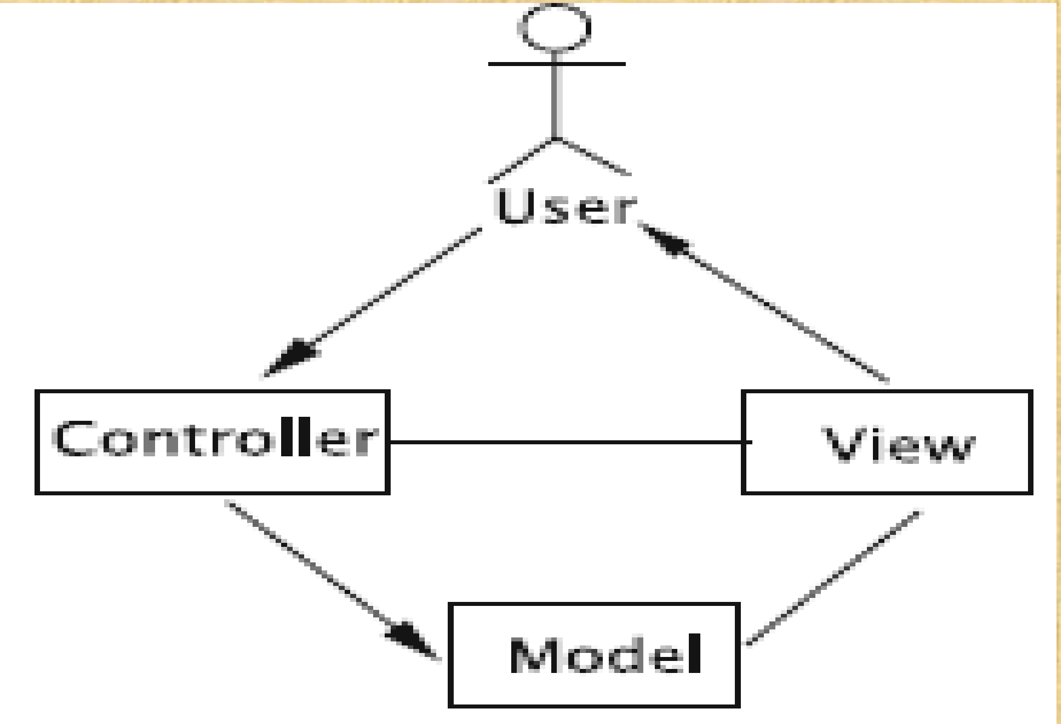
Module-4

Interactive systems and the MVC architecture(Chapter11,Book1)

- Introduction
- The MVC architectural pattern
- Analyzing a simple drawing program
- Designing the system
- Designing of the subsystems
- Getting into implementation
- Implementing undo operation
- Drawing incomplete items
- Adding a new feature
- Pattern based solutions.

The MVC Architectural Pattern

The model view controller is a relatively old pattern that was originally introduced in the Smalltalk programming language.

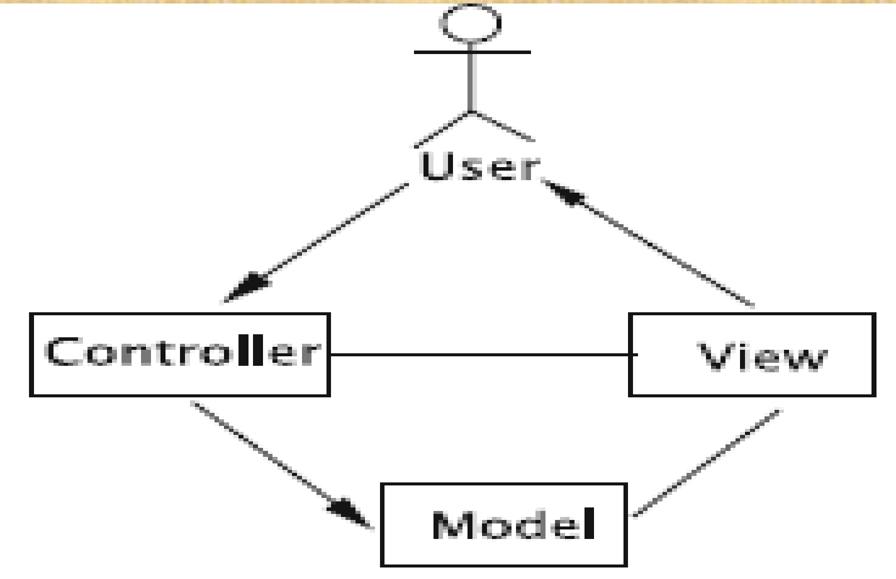


The model–view–controller architecture

The pattern divides the application into three subsystems:

1. model
2. view
3. controller.

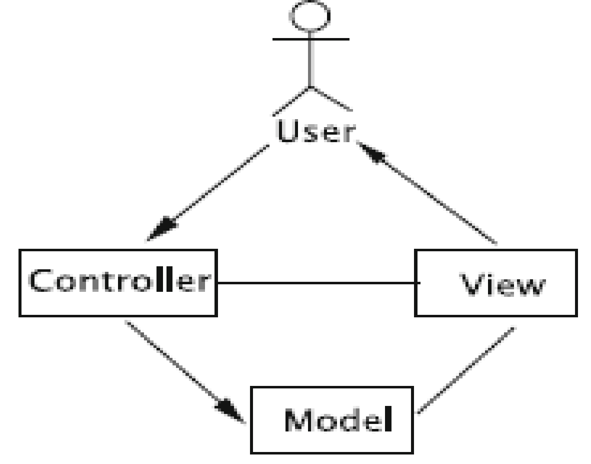
MVC provides a powerful way to organise systems that support multiple presentations of the same information.



MVC pattern helps produce highly cohesive modules with a low degree of coupling.

- 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)
- from the way in which the end-user manipulates it (Controller).

1. The model,
   - which is a relatively passive object,
   - stores the data.
   - Any object can play the role of model.



2. 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. The controller
   - captures user input
   - 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.

It is important to distinguish the UI from the rest of the system:

- Beginners often mistake the UI for the view. Why?

Two reasons:

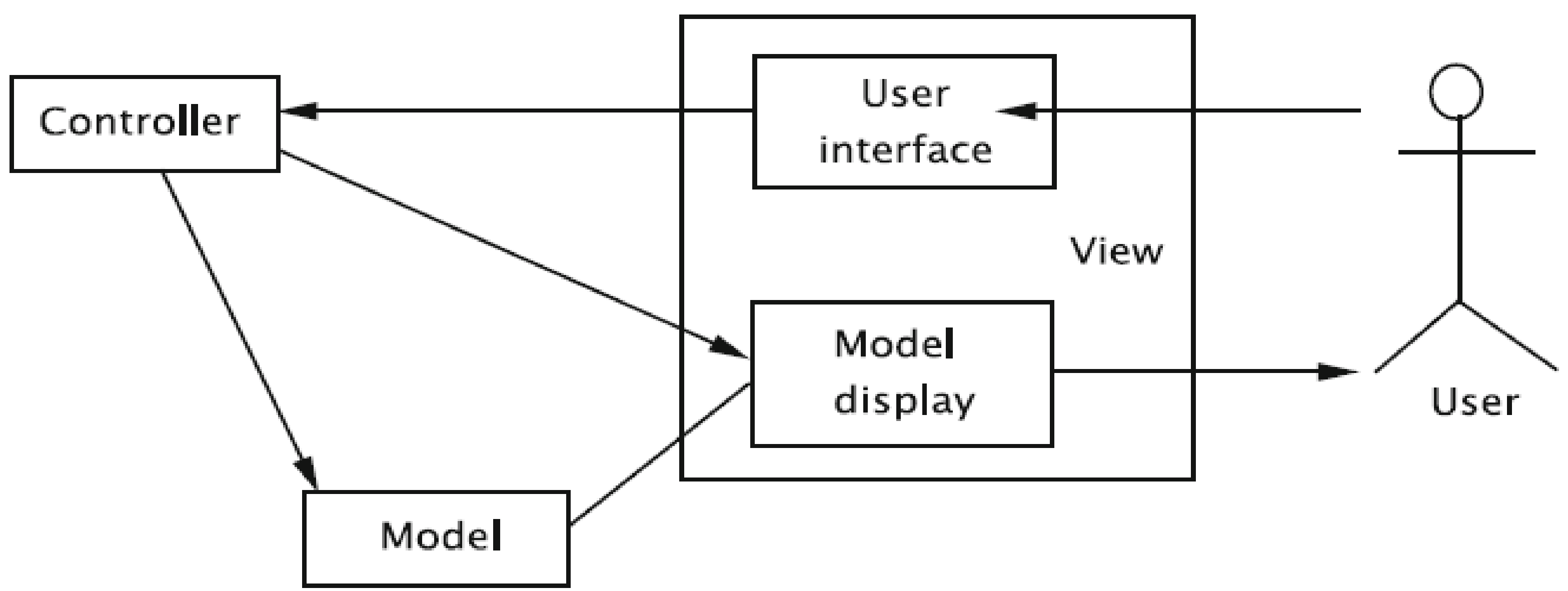
- ➢ In most systems, due to the nature of the desired look and feel from end users.
- ➢ The technologies available, there is a single window in which the entire application is housed.

- There has to be a common subsystem that provides the functionality needed both for the view and the user interface.
- 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.

MVC in the abstract sense:

- Architecture of the system 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.
- But in practice: the view and the UI are contained in a common subsystem

- *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)
    - ➢ from issues relating to how we render the model.



- 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,
- ✓ 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 the controllers, but through some other mechanism.
- In this case, the model must notify all associated views of the changes.

The view–model is similar to 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.***

Example 1: **library system**

- 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.

Example 2: **Mail Server**

- 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.

Example 3: **graph-plot of pairs of *(x, y)* values.**

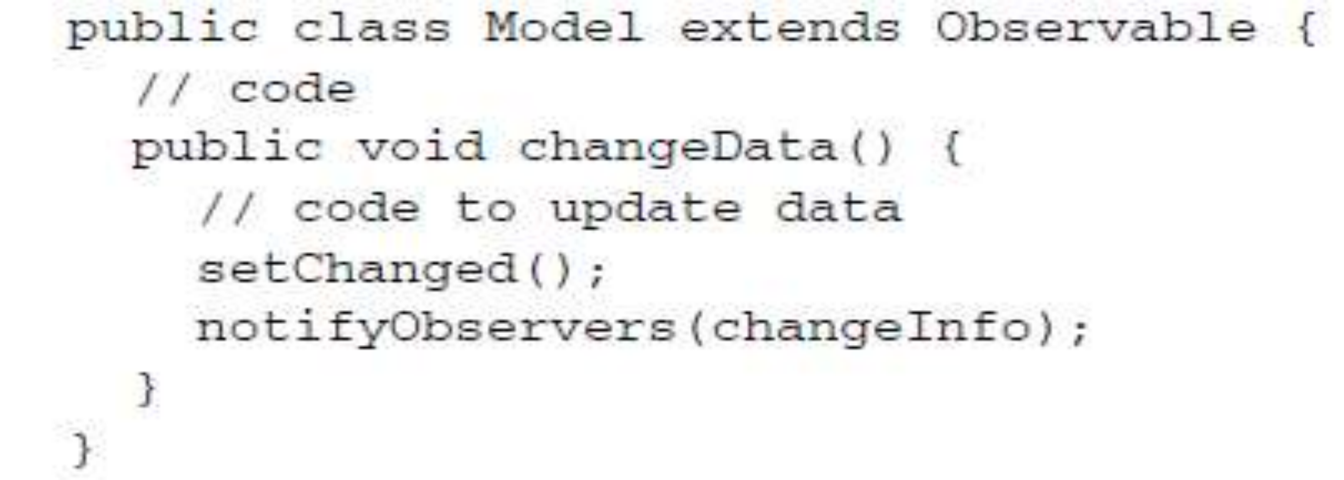
- The collection of data points constitutes the model.
- The graph-viewing software provides the user with several output formats—bar graphs, line graphs, pie charts, etc.
- When the user changes formats, the view changes without any change to the model.

Implementation: MVC

- 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.

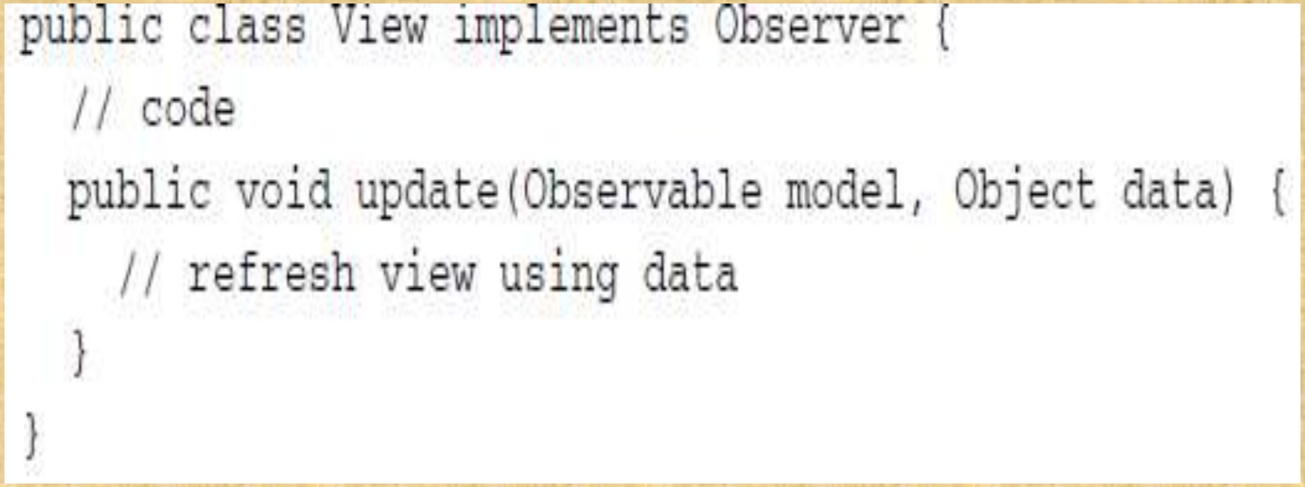
- The *view* is responsible for *all the presentation issues.*
- The *model* holds the *application object*.
- The *controller* takes care of the *response strategy*.

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, and as belonging to one of the *three categories*.
- There is always an initiation step for each operation;
- The manner in which the user is to be shown the feature
- The manner in which it is invoked are part of the presentation.
- 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.
- It is not always possible to have a clean division of the components such that some components are designated for data input and the rest are for data display.

- The approach =>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.

Benefits of the MVC Pattern:

1. *Cohesive modules:*
   - ✓ Instead of putting unrelated code (display and data) in the same module, 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=>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.
   - ✓ 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.

Analysing a Simple Drawing Program

- Apply the MVC architectural pattern design to create and label figures. The purpose is in two folds:
  - ✓ *To demonstrate how to design with an architecture in mind*
    - ➢ Start with a high-level decomposition of responsibilities across the subsystems.
    - ➢ 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*
    - ➢ try to have three clearly delineated subsystems for Model, View, and Controller.

Specifying the Requirements

Initial wish-list calls for software that can do the following

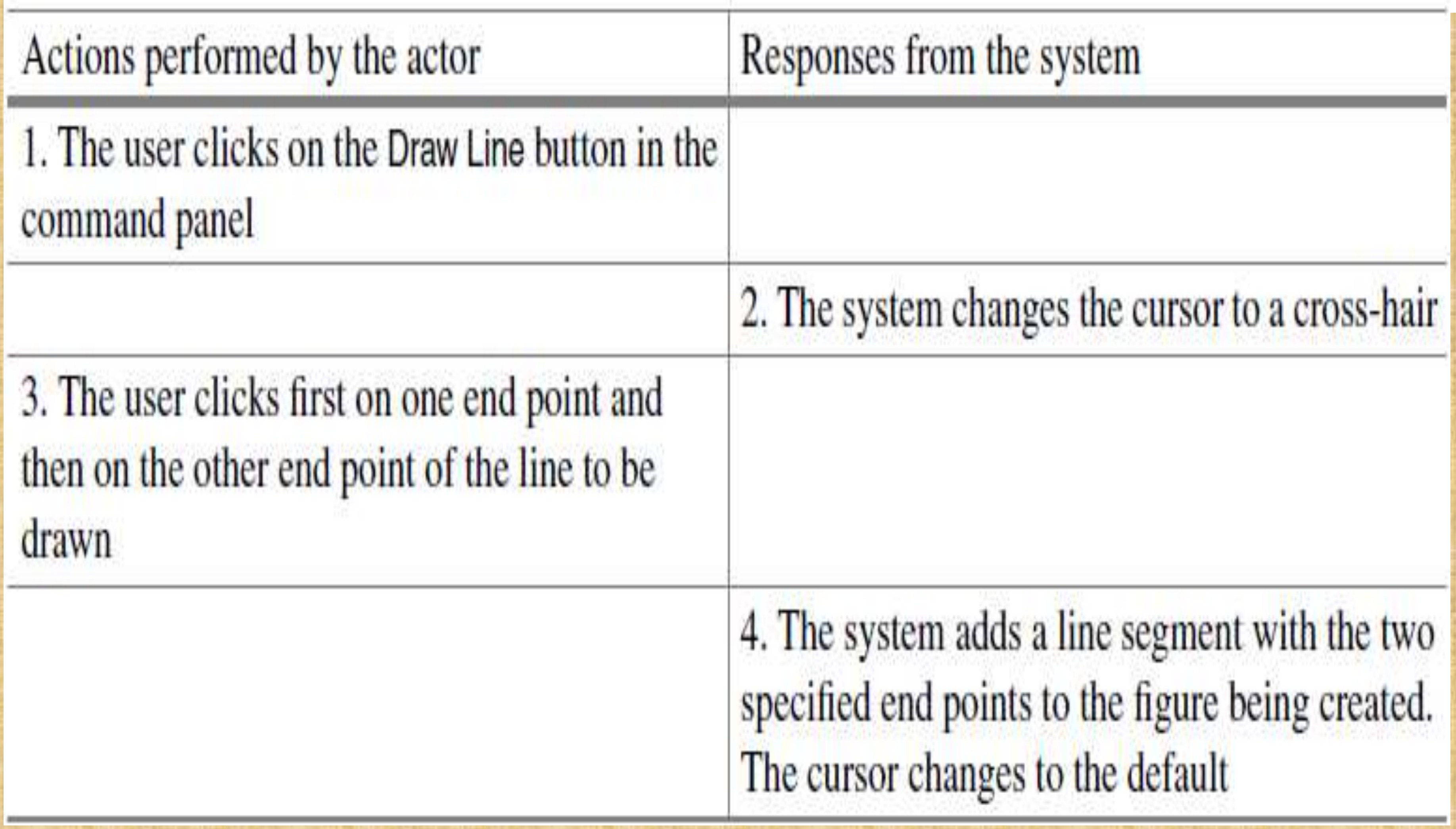
1. Draw lines and circles.
2. Place labels at various points on the figure; the labels are strings. A separate command allows the user to select the font and font size.
3. Save the completed figure to a file. We can open a file containing a figure and edit it.
4. Backtrack our drawing process by undoing recent operations.

Drawing Program 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,
  - ❖ 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.

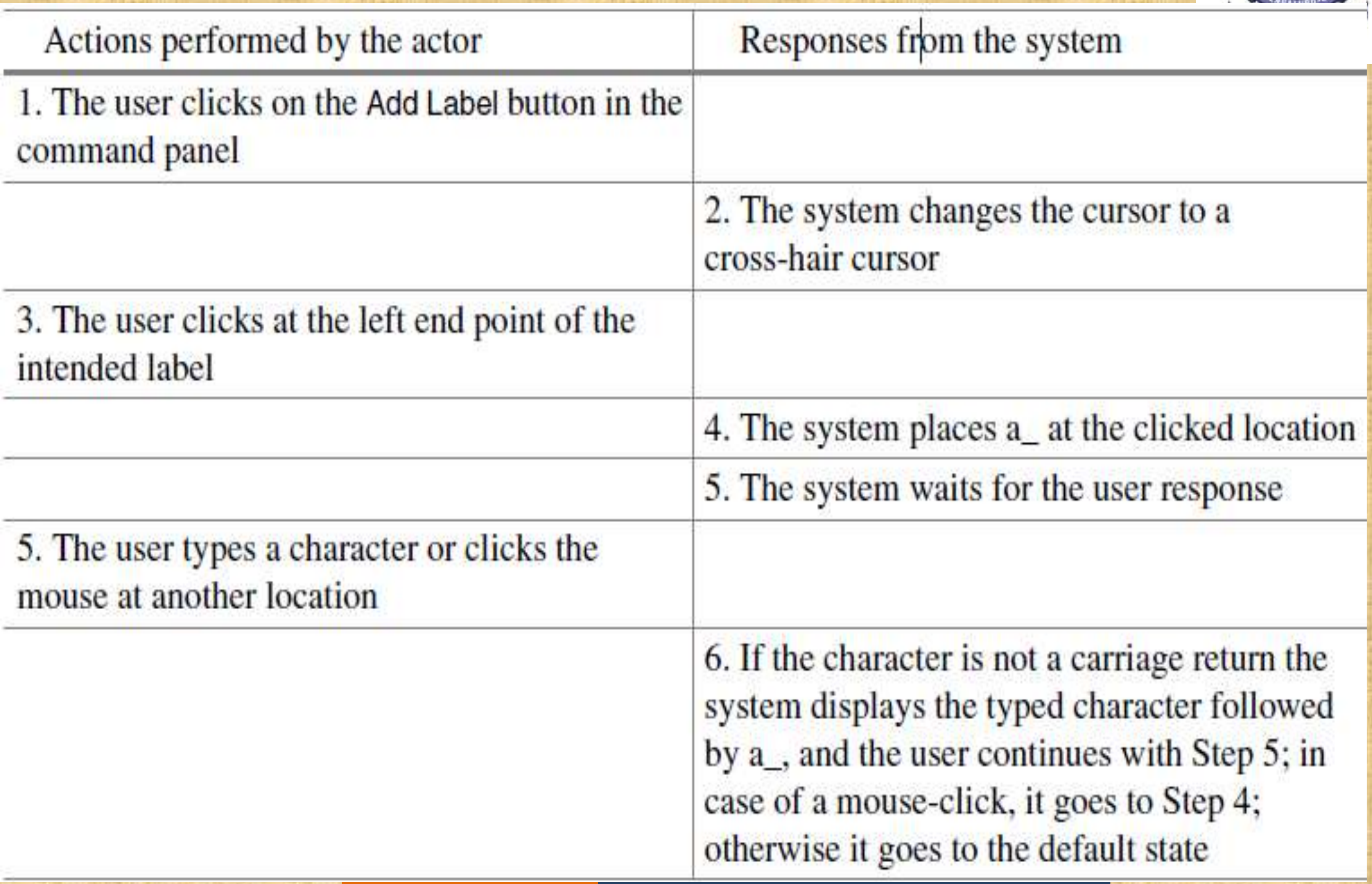
Defining the Use Cases

1. Use-case table for Drawing a line



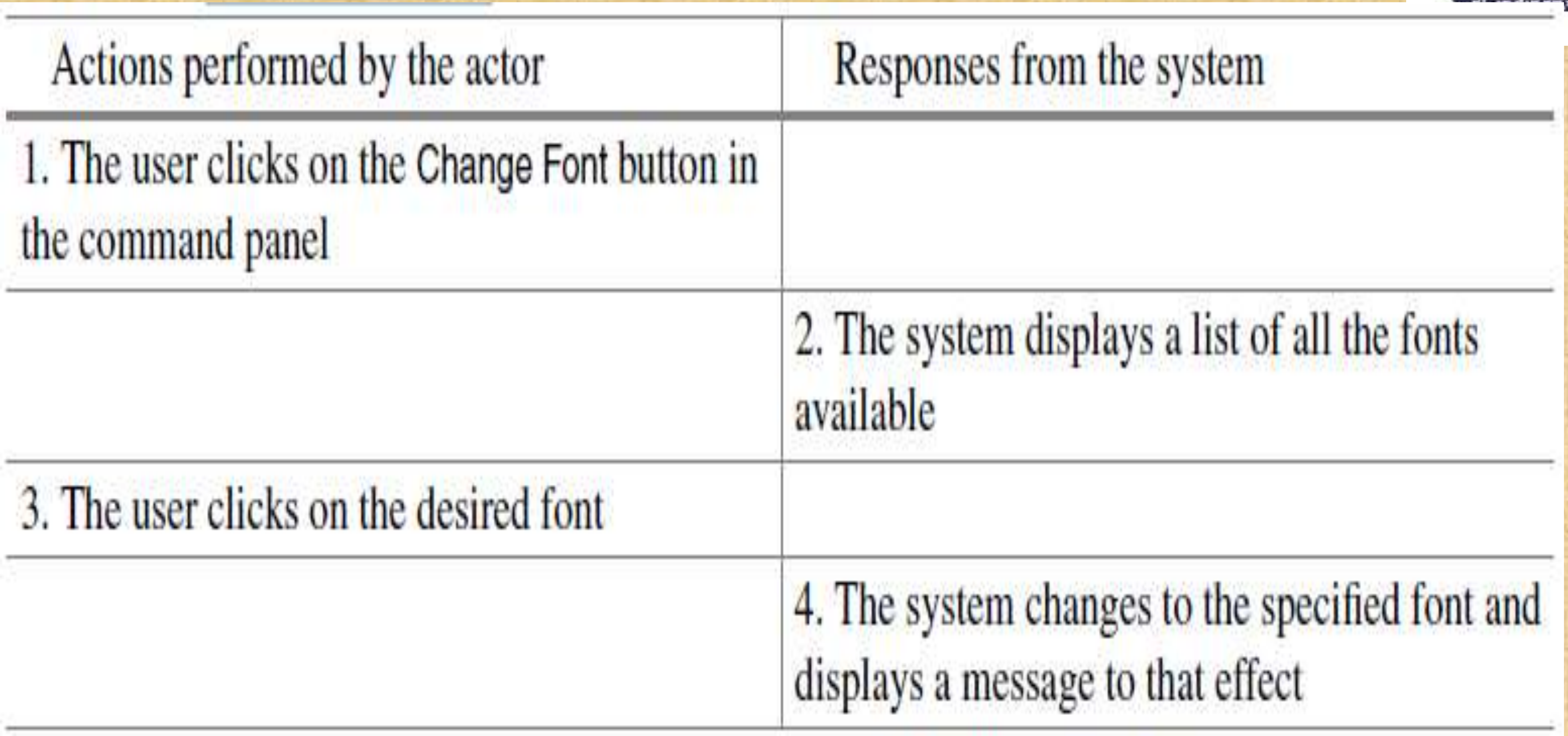
Actions performed by the actor	Responses from the system
1. The user clicks on the Draw Line button in the command panel	
	2. The system changes the cursor to a cross-hair
3. The user clicks first on one end point and then on the other end point of the line to be drawn	
	4. The system adds a line segment with the two specified end points to the figure being created. The cursor changes to the default

2. Use-case table for Adding a Label



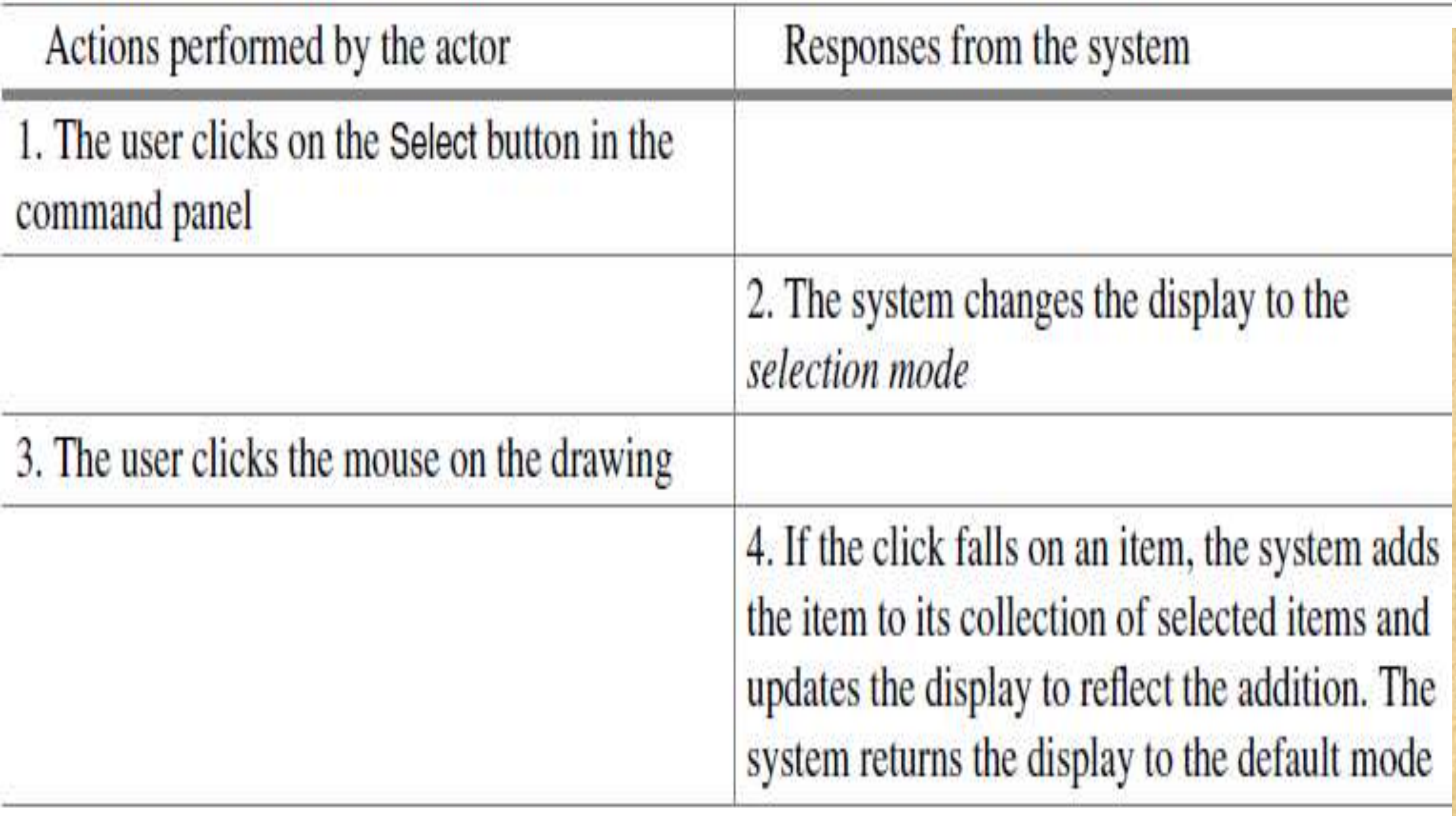
Actions performed by the actor	Responses from the system
1. The user clicks on the Add Label button in the command panel	
	2. The system changes the cursor to a cross-hair cursor
3. The user clicks at the left end point of the intended label	
	4. The system places a_ at the clicked location
	5. The system 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

3. Use-case table for Change Font



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

4. Use-case table for Select an Item



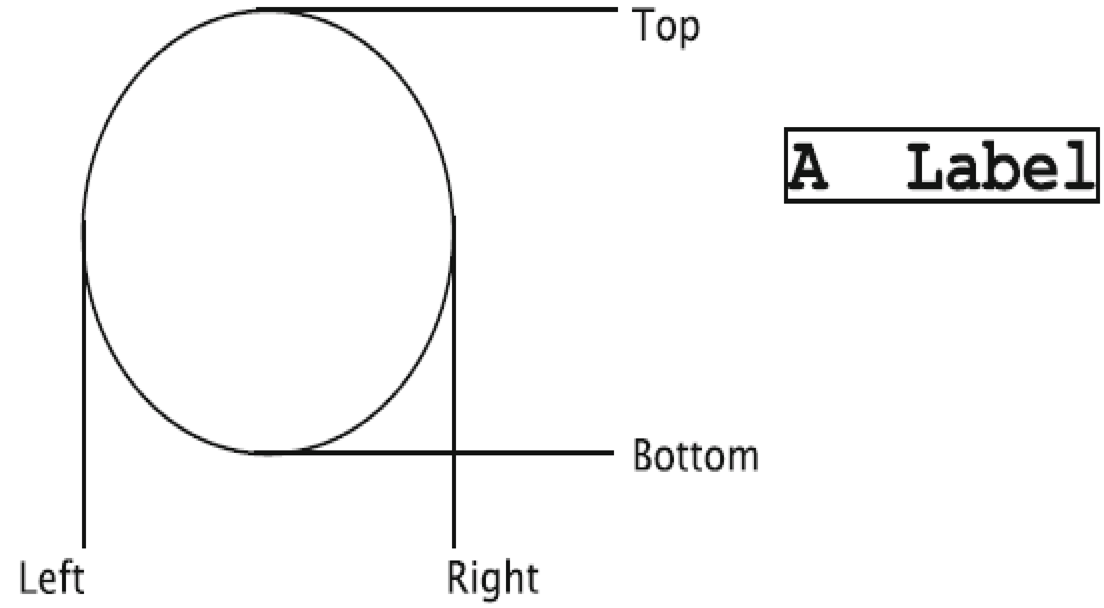
Actions performed by the actor	Responses from the system
1. The user clicks on the Select button in the command panel	
	2. The system changes the display to the *selection mode*
3. The user clicks the mouse on the drawing	
	4. If the click falls on an item, the system adds the item to its collection of selected items and updates the display to reflect the addition. The system returns the display to the default mode

Designing the System

- Our architecture specifies three principal subsystems, viz., the Model, the View and the Controller.
- Look at the individual use cases and decide how the responsibilities are divided across the three subsystems.

1. *Defining the Model*

- 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
- 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 model also provides mechanisms to access and modify its collection objects.



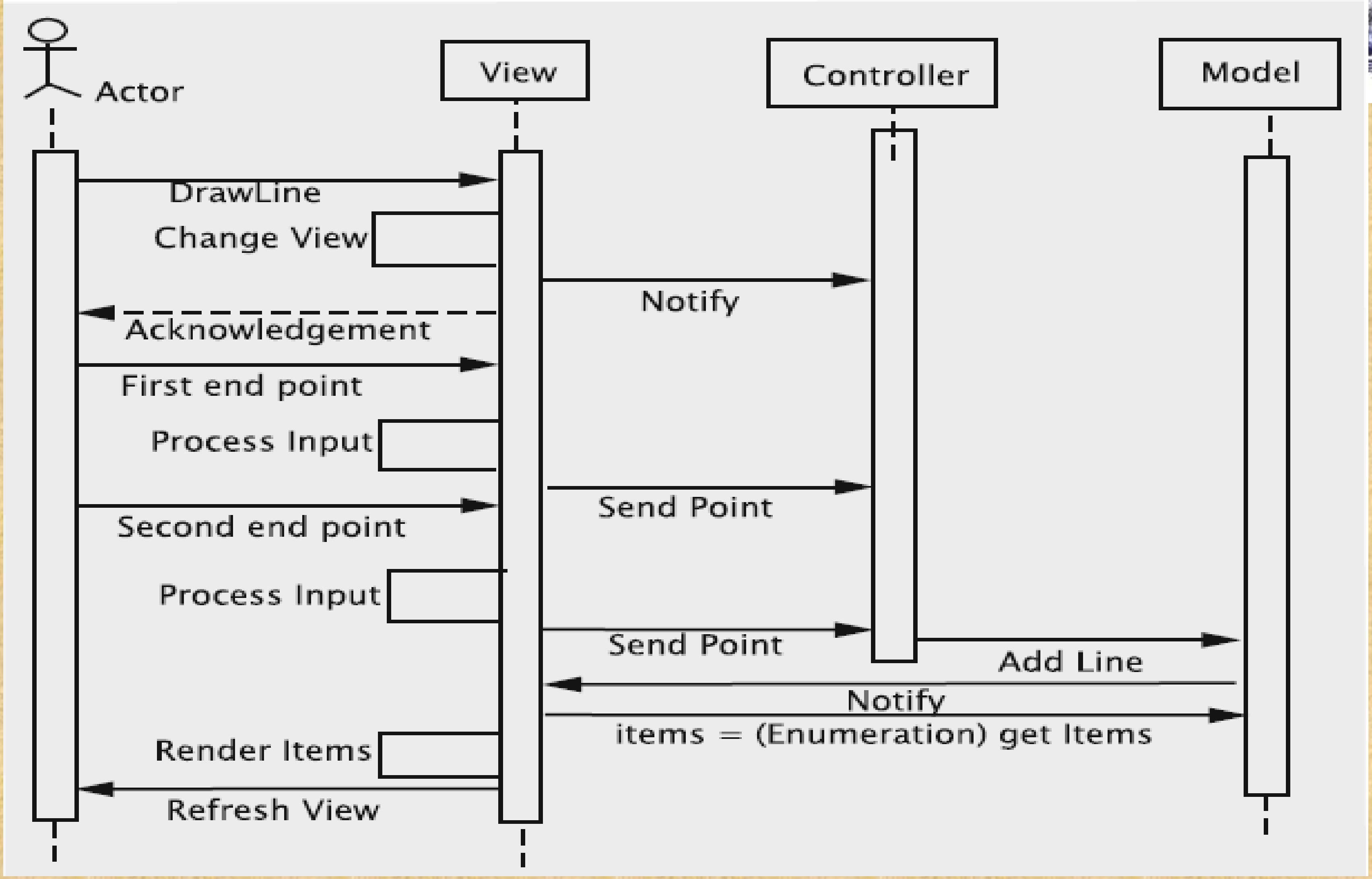
2. Defining the Controller

- The controller is the subsystem that orchestrates the whole process => thus defining its role is 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.

Example: Drawing a Line:

- ➢ The user starts by clicking the Draw line button, and in response, the system changes the cursor=> this is a responsibility of view
- ➢ The button click indicates that the user has initiated an operation that would change the model.
- ➢ This operation is informed to the controller that creates a line object with end points unspecified.

Example: Drawing a Line: *contd…*

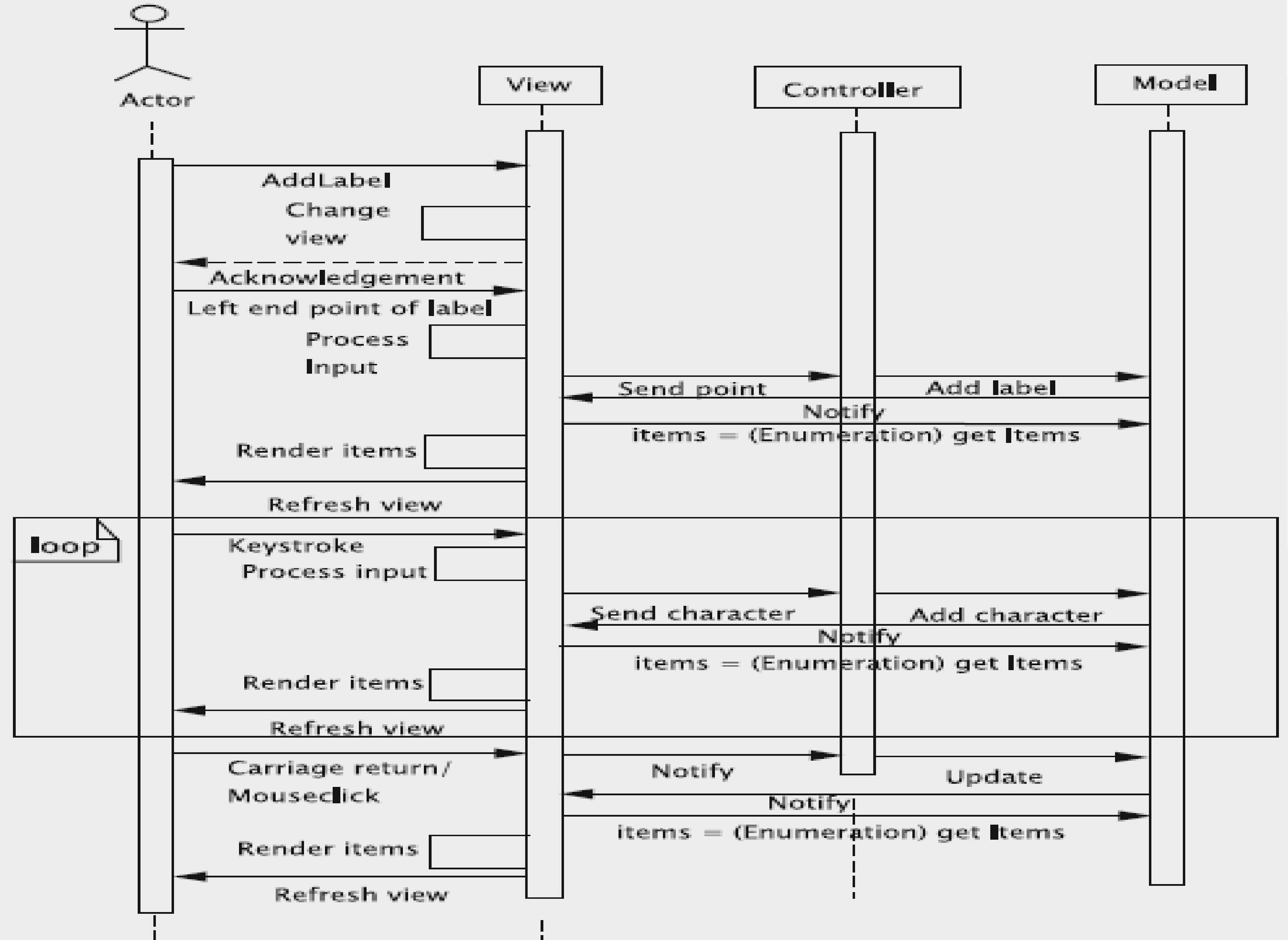


Sequence of operations for drawing a line

Drawing a Circle

- The actions for drawing a circle are similar. But some additional processing to be done,
- i.e., the given points on the diameter must be converted to the the four integer values,
- This requires a mapping to convert the input to the form required by the model.
- This can be performed in the controller.

Adding a Label



Dealing with the Environmental Variables

- Most interactive systems need to remember the values of certain parameters to make the system user-friendly.
- For instance, a word-processing system remembers the size and font of the characters so that the user does not have to specify these with every operation.
- In example, for creating a label, we choose to store these in the view, and this has some consequences for the behaviour of the system.

Selection and Deletion

- The two operations are treated as two independent operations
- 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. 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.
3. 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.

Saving and Retrieving the Drawing

User requests a save/retrieve operation, the system asks for a file name which the user provides and the system completes the task.

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.

Design of the Subsystems

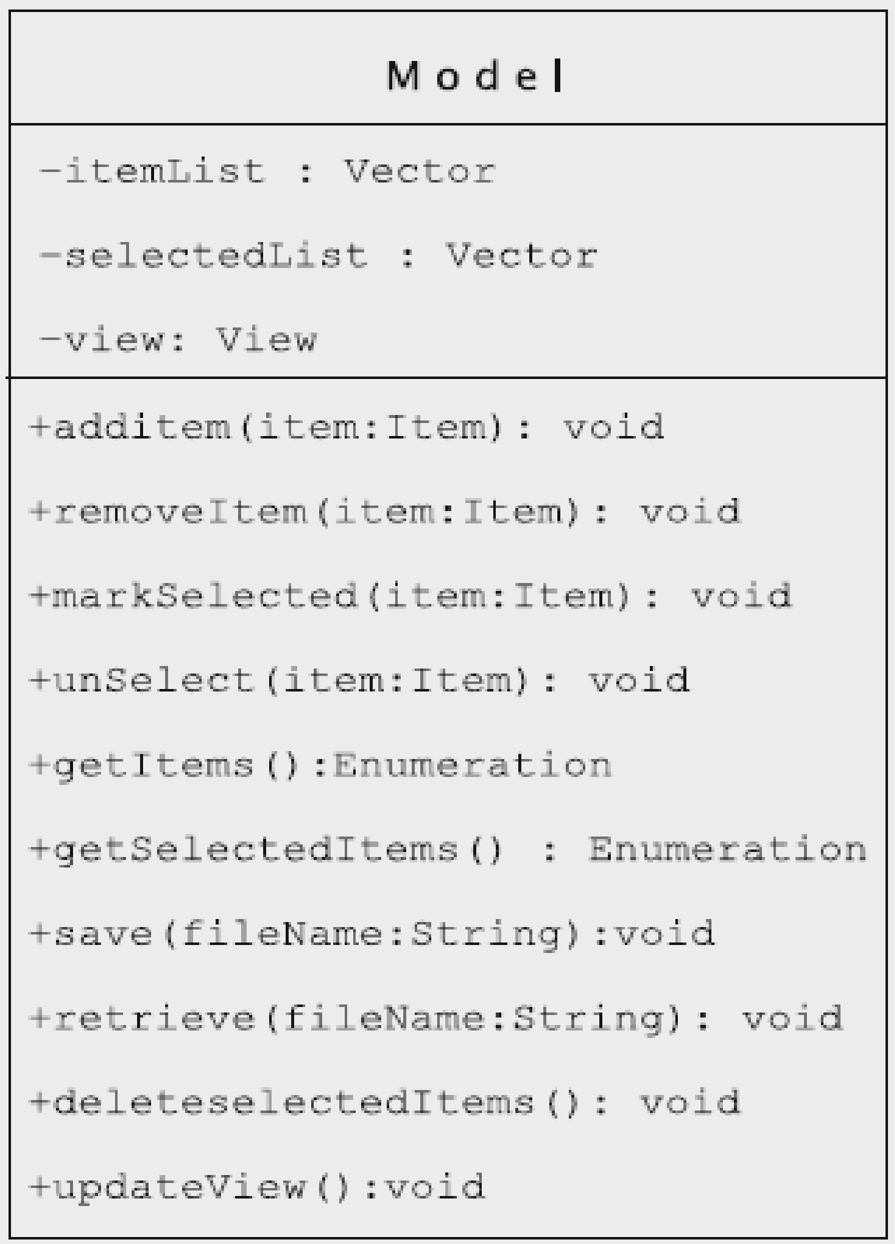
1. ***Design of the Model Subsystem***
2. ***Design of Item and Its Subclasses***
3. ***Design of the Controller Subsystem***
4. ***Design of the View Subsystem***

1. Design of the Model Subsystem

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

Class diagram for model



```
Model

-itemList : Vector
-selectedList : Vector
-view: View

+additem(item:Item): void
+removeItem(item:Item): void
+markSelected(item:Item): void
+unSelect(item:Item): void
+getItems():Enumeration
+getSelectedItems() : Enumeration
+save(fileName:String):void
+retrieve(fileName:String): void
+deleteselectedItems(): void
+updateView():void
```

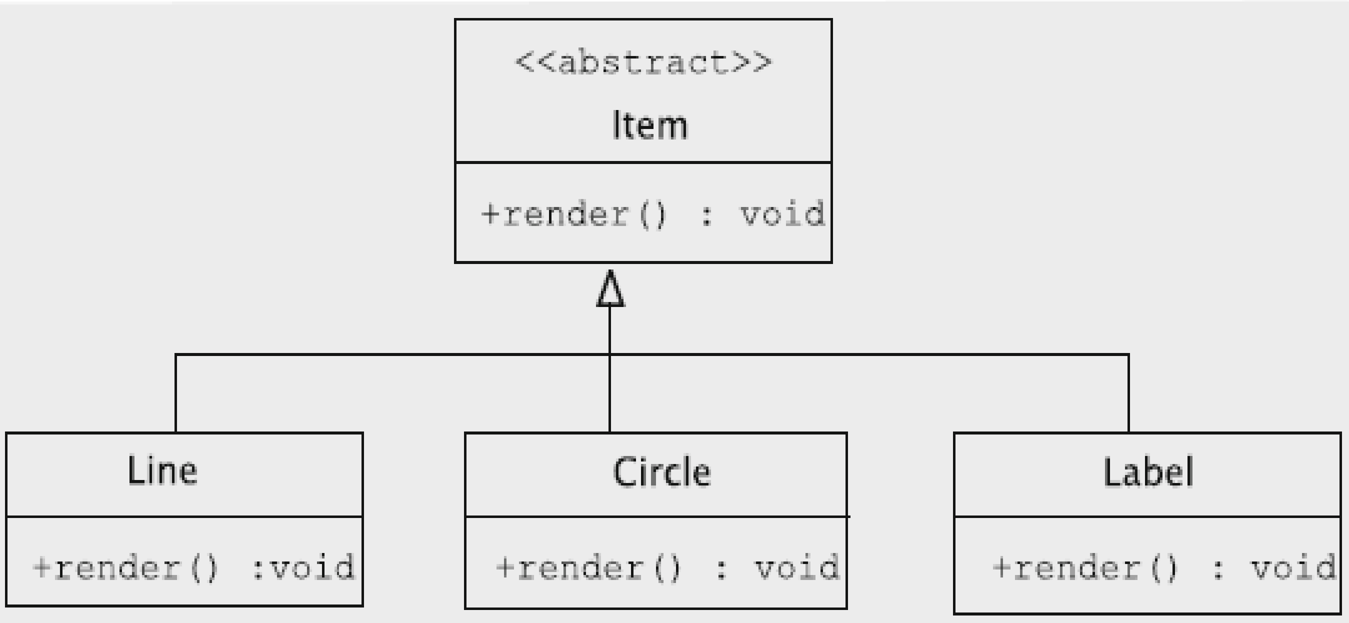
2. Design of Item and Its Subclasses

- 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.

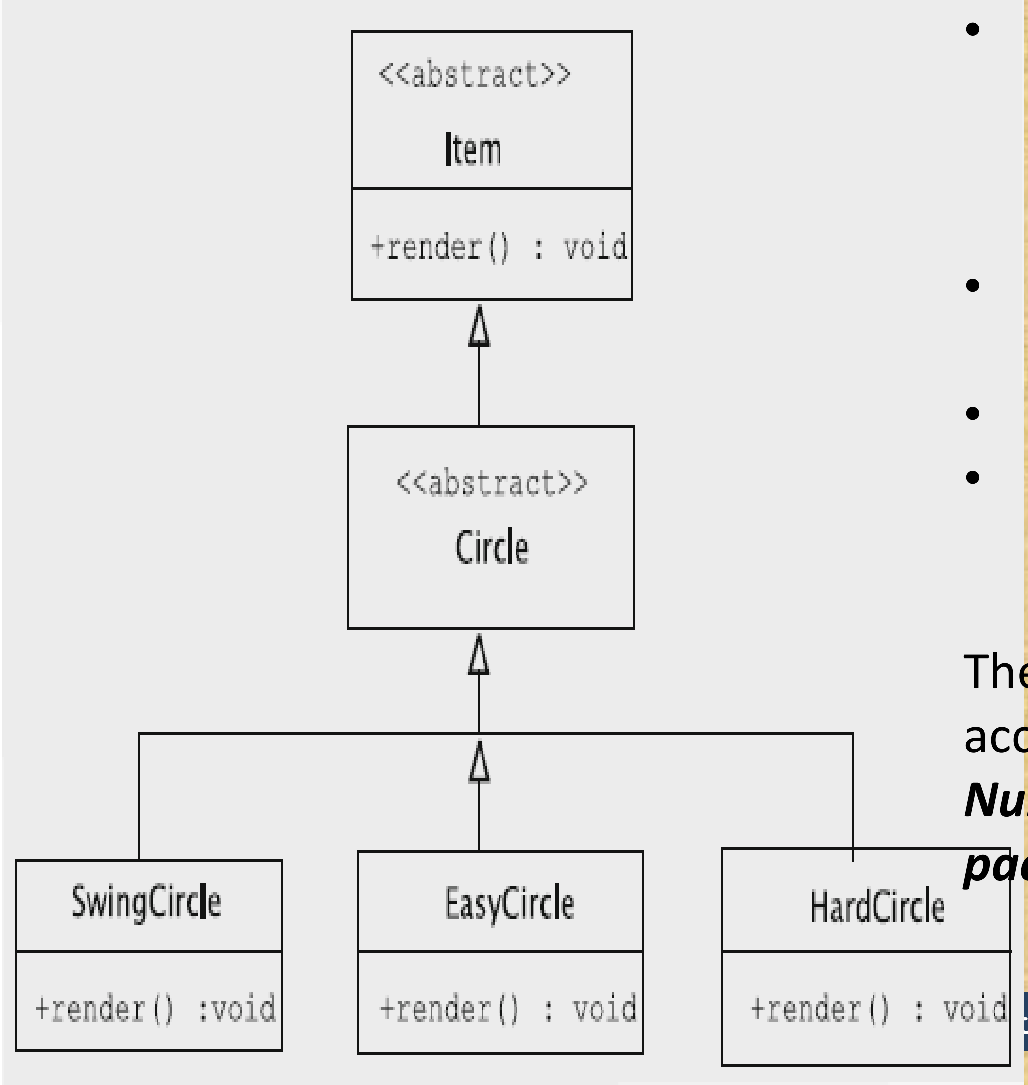
- How the drawing is done are dependent on the following two parameters:

1. *The technology and tools that are used in creating the UI*
2. *The item that is stored*



Catering to Multiple UI Technologies

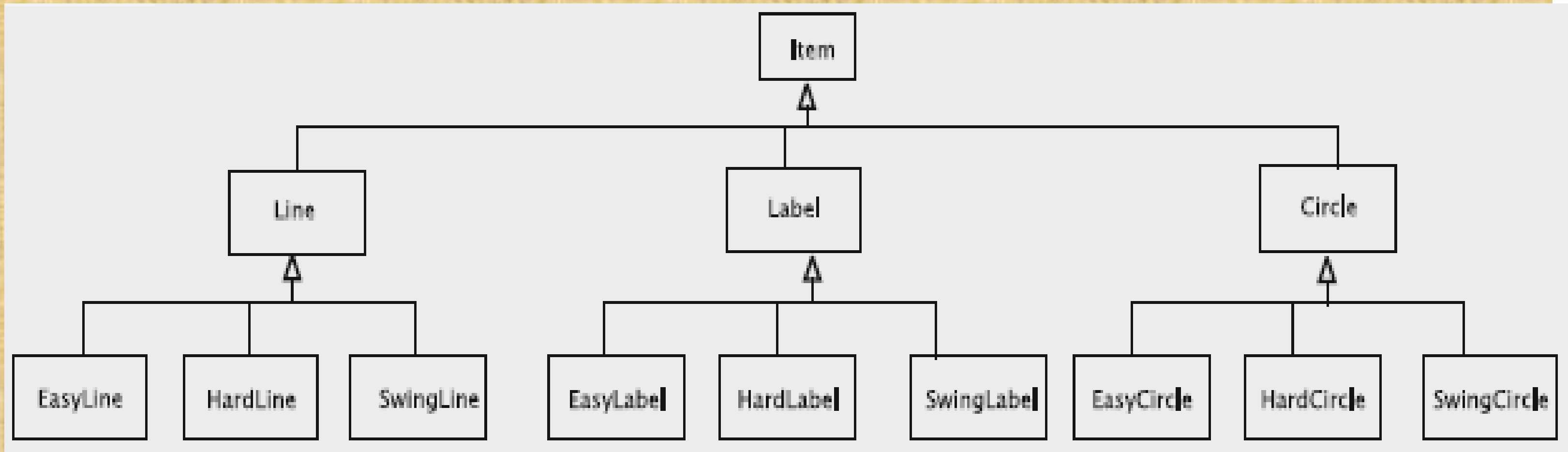
The render method will decompose the circle into smaller components as needed, and invoke the methods available in the UI to render each component.



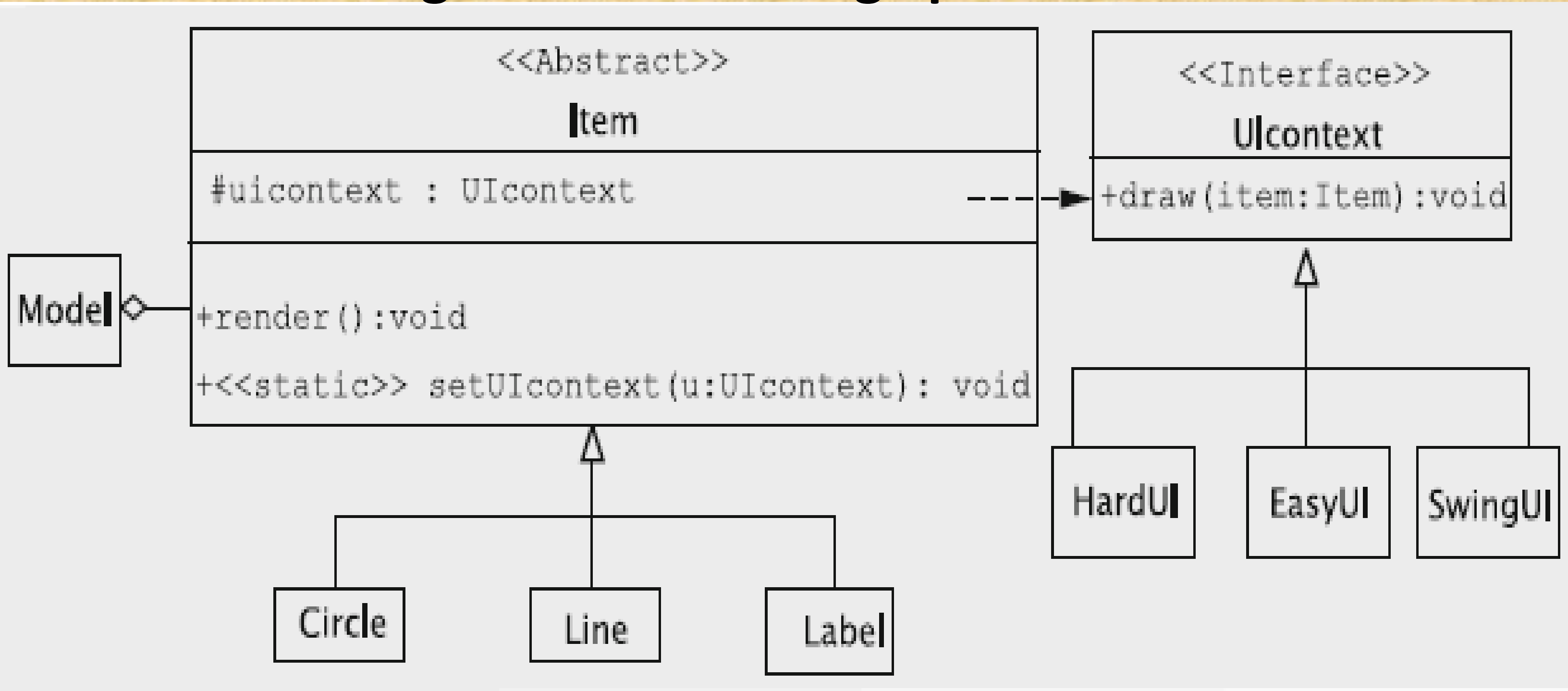
- we need abstract classes for implementing the technology-independent parts of lines (Line) and labels (Label).
- They are extended by classes such as SwingLabel, SwingLine, EasyLabel, etc.
- This extension adds another six classes.
- Each abstract class ends up with as many subclasses as the number of Uis that we have to accommodate.

The number of classes needed to accommodate such a solution is given by: ***Number of types of items × Number of UI packages***

Class explosion due to multiple UI implementations



Interaction diagram for the bridge pattern



Interaction diagram for the bridge pattern

- The intent of the bridge pattern is as follows: *Decouple an abstraction from its implementation so that two can vary independently*.
- The bridge pattern takes care of these problems avoiding a permanent binding
- The total number of classes is now reduced to

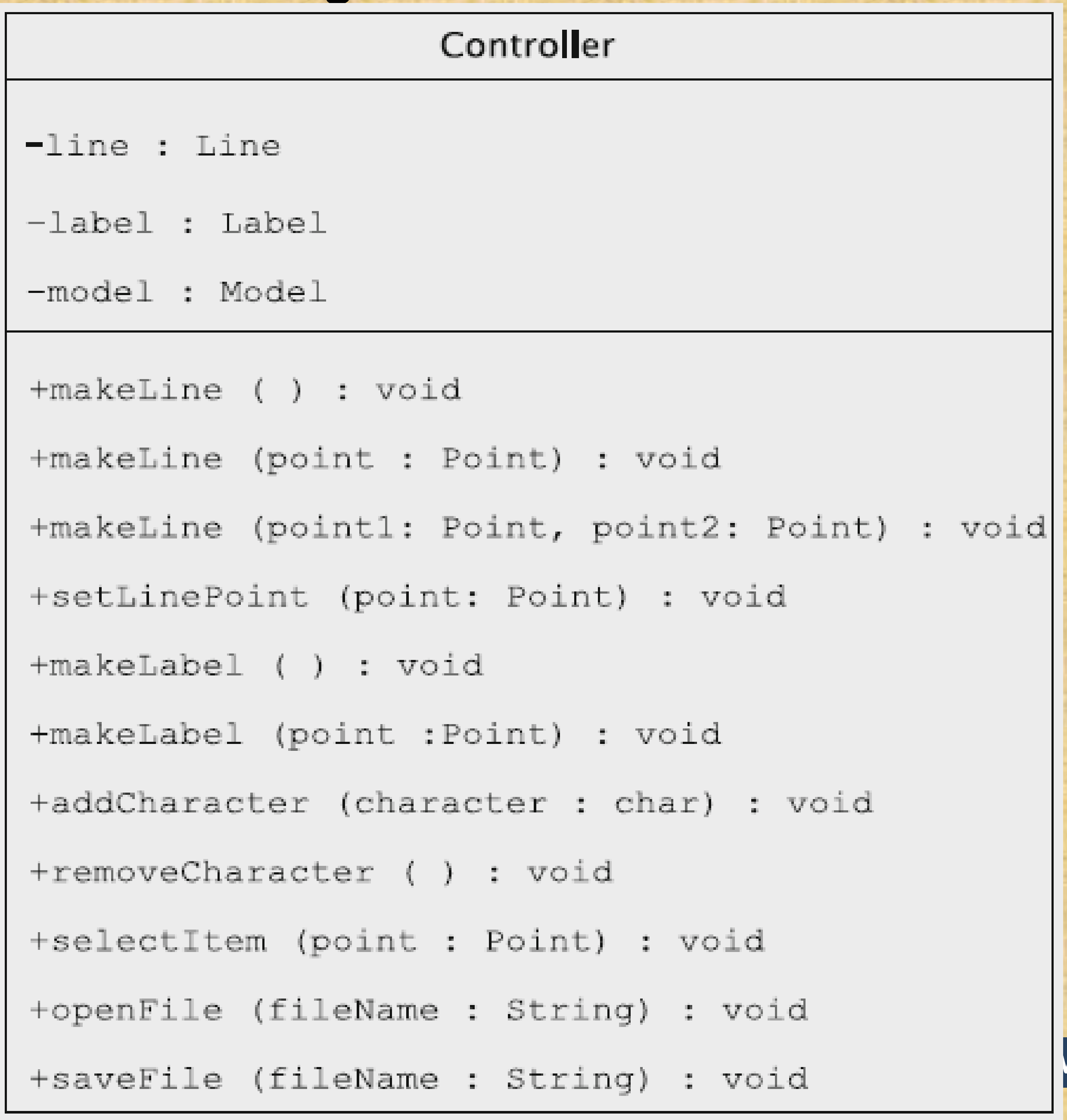
Number of types of items + Number of UI packages

3. Design of the Controller Subsystem

- Unlike the view, which by definition could be implemented in multiple ways,
- 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. To store the current line, label, or circle being created. Since we have three shapes, this would mean having three fields

Controller class diagram



Controller

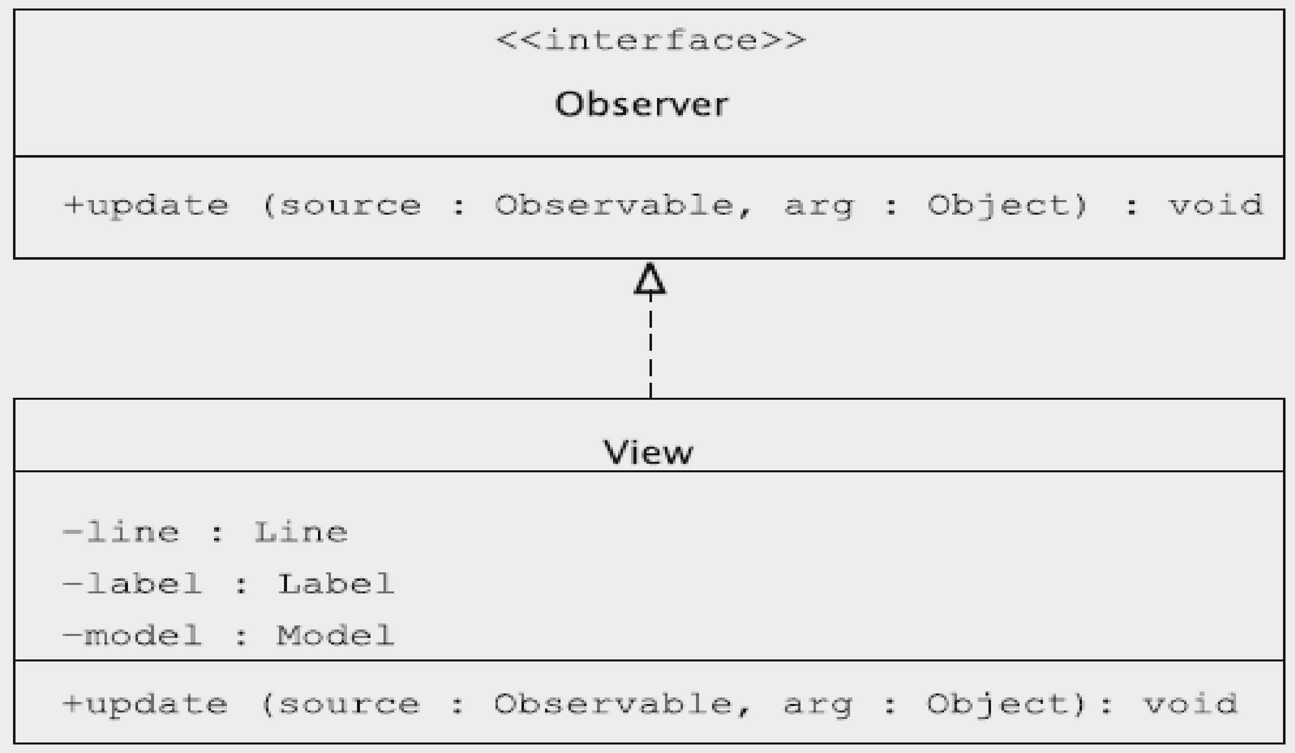
```
-line : Line

-label : Label

-model : Model
```

```
+makeLine ( ) : void

+makeLine (point : Point) : void

+makeLine (point1: Point, point2: Point) : void

+setLinePoint (point: Point) : void

+makeLabel ( ) : void

+makeLabel (point :Point) : void

+addCharacter (character : char) : void

+removeCharacter ( ) : void

+selectItem (point : Point) : void

+openFile (fileName : String) : void

+saveFile (fileName : String) : void
```

4. Design of the View Subsystem

- The MVC pattern makes the view largely independent of the other subsystems.
- 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.



<<interface>>

Observer

+update (source : Observable, arg : Object) : void

View

-line : Line

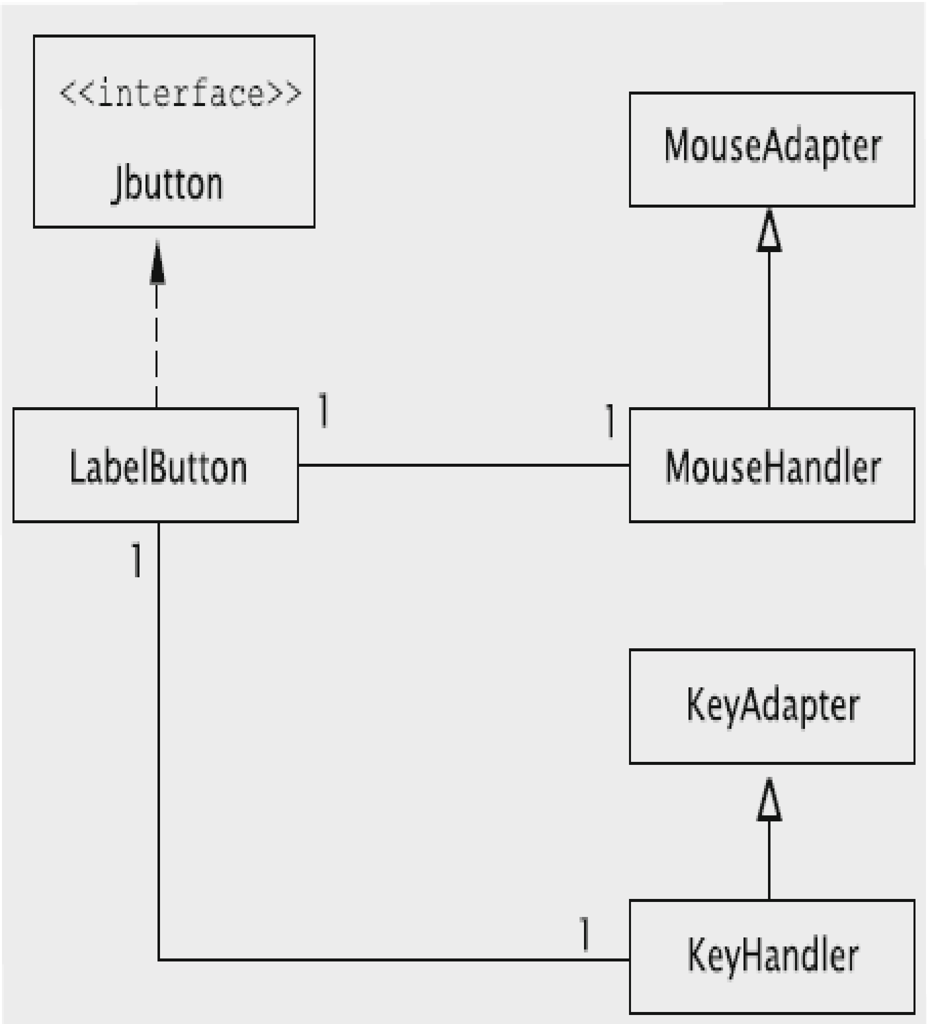
-label : Label

-model : Model

+update (source : Observable, arg : Object): void

Basic structure of the view class

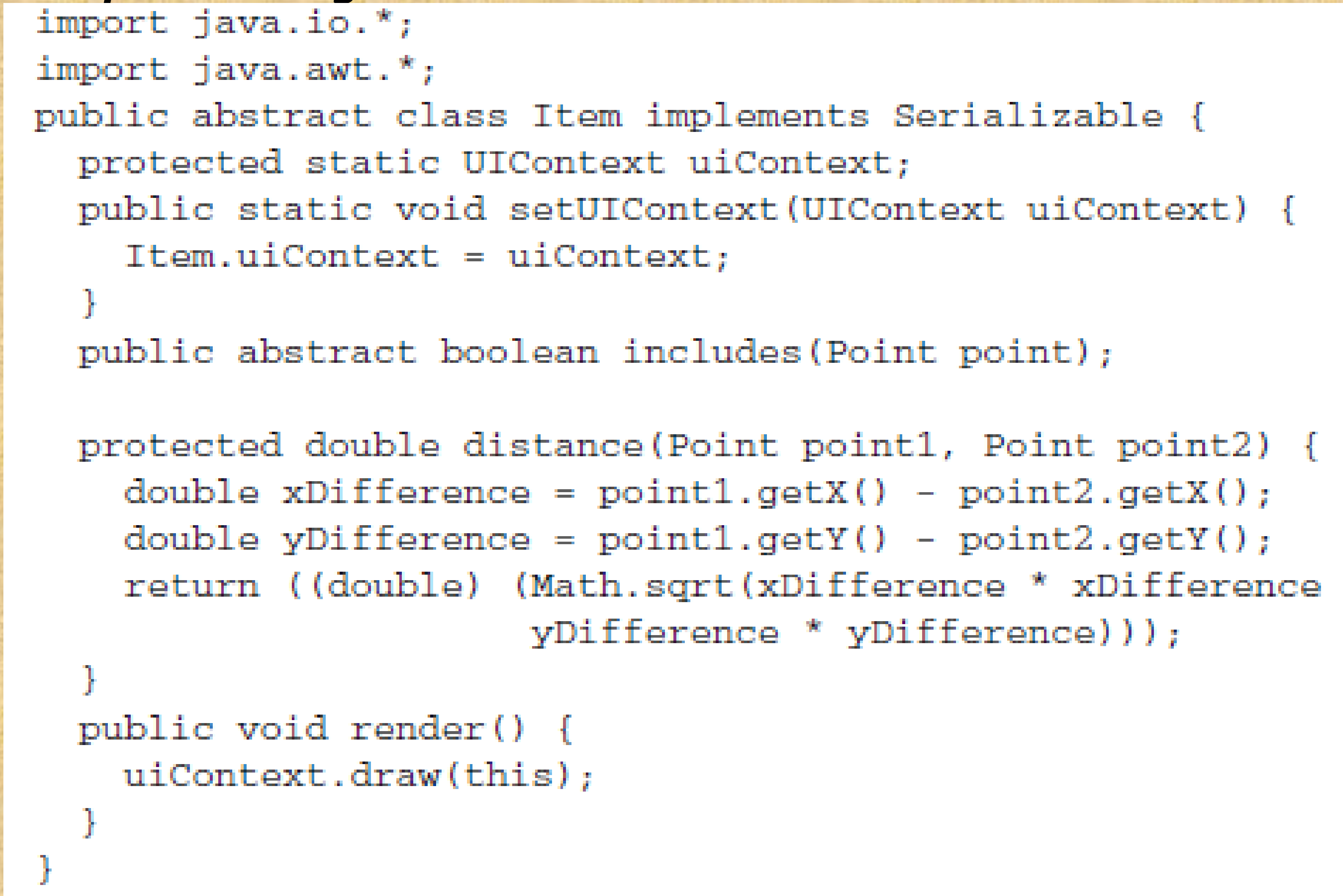
- Commands to create labels, circles, and lines all require mouse listeners.
- Since there is a one-to-one correspondence between the mouse listeners and the drawing commands,
- we have the following structure:



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.

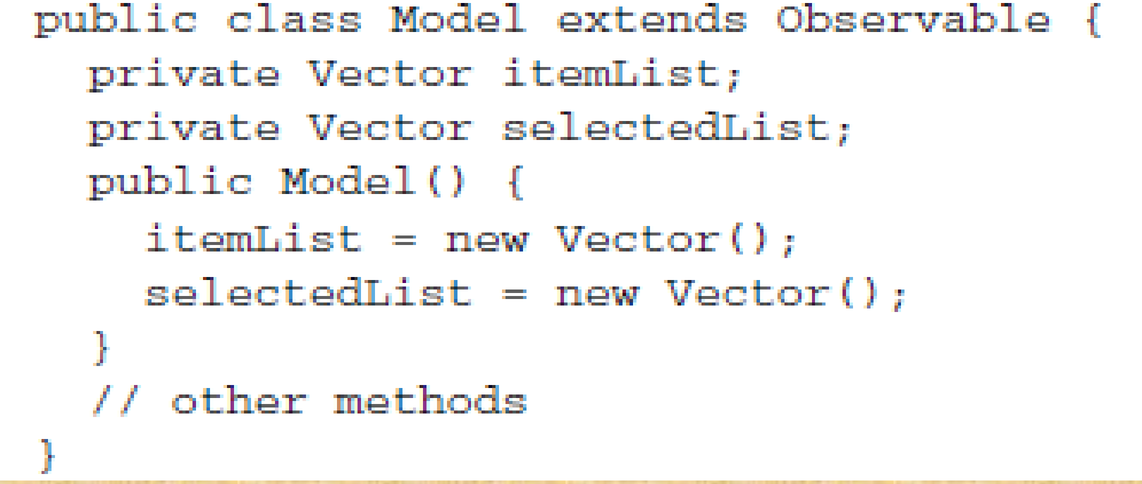
Getting into the Implementation

Implementing Item and Its Subclasses



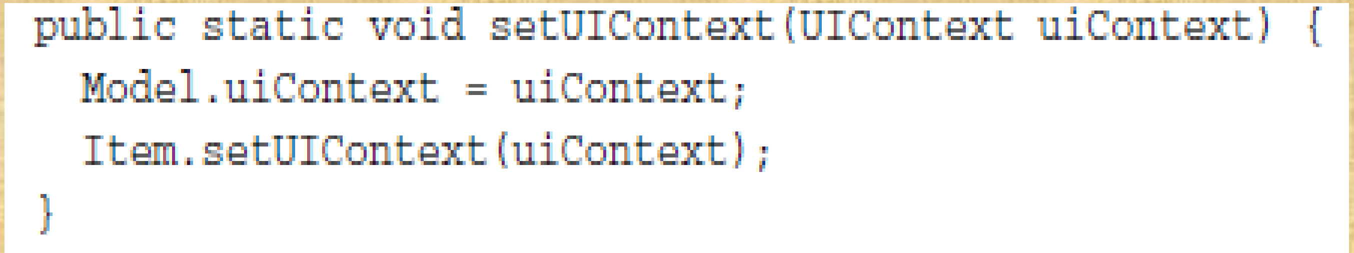
```
import java.io.*;
import java.awt.*;
public abstract class Item implements Serializable {
  protected static UIContext uiContext;
  public static void setUIContext(UIContext uiContext) {
    Item.uiContext = uiContext;
  }
  public abstract boolean includes(Point point);

  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);
  }
}
```

Implementation of the Model Class



```
public class Model extends Observable {
  private Vector itemList;
  private Vector selectedList;
  public Model() {
    itemList = new Vector();
    selectedList = new Vector();
  }
  // other methods
}
```

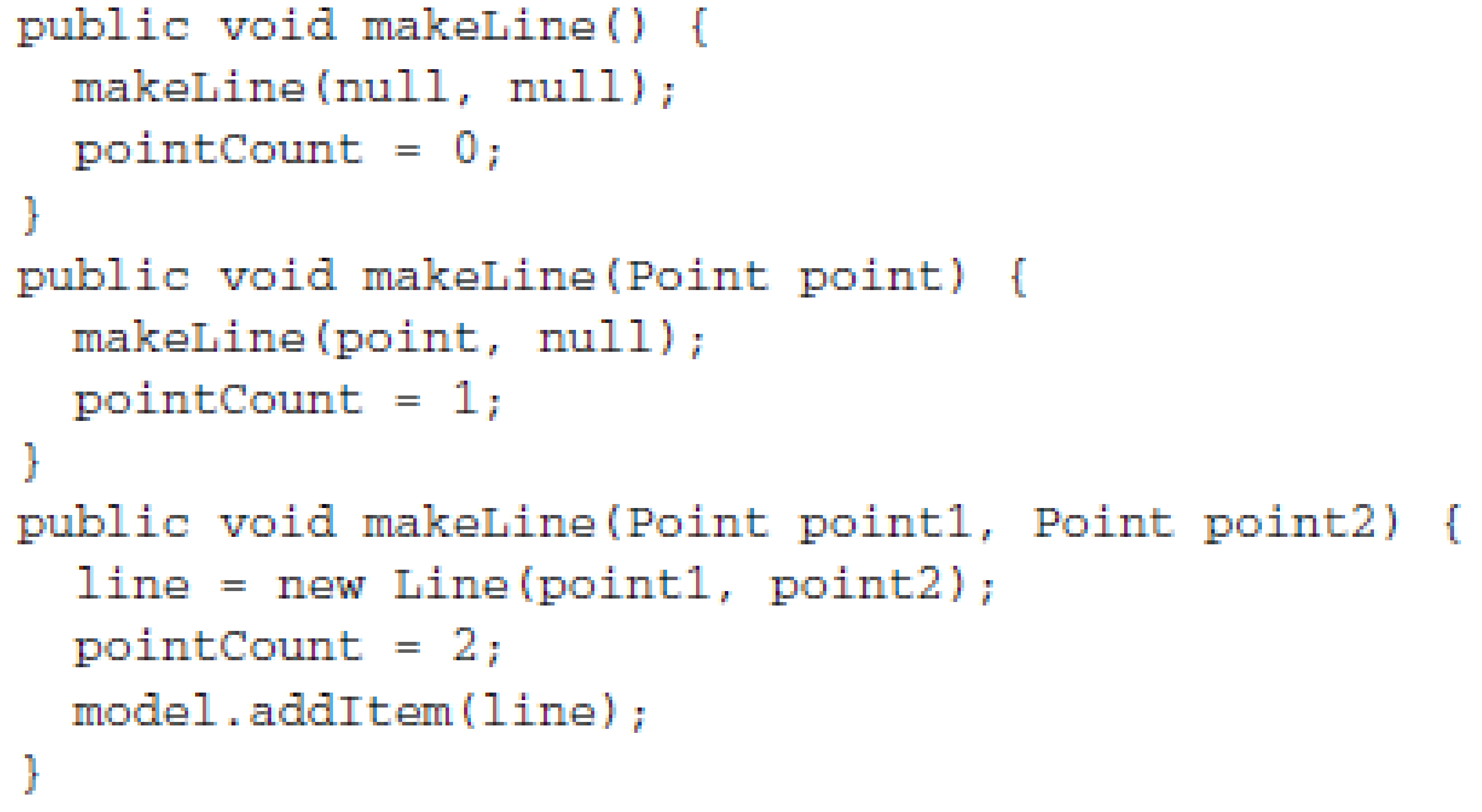
The setUIContext method in the model in turn invokes the setUIContext on Item.



```
public static void setUIContext(UIContext uiContext) {
  Model.uiContext = uiContext;
  Item.setUIContext(uiContext);
}
```

Implementation of the Controller Class

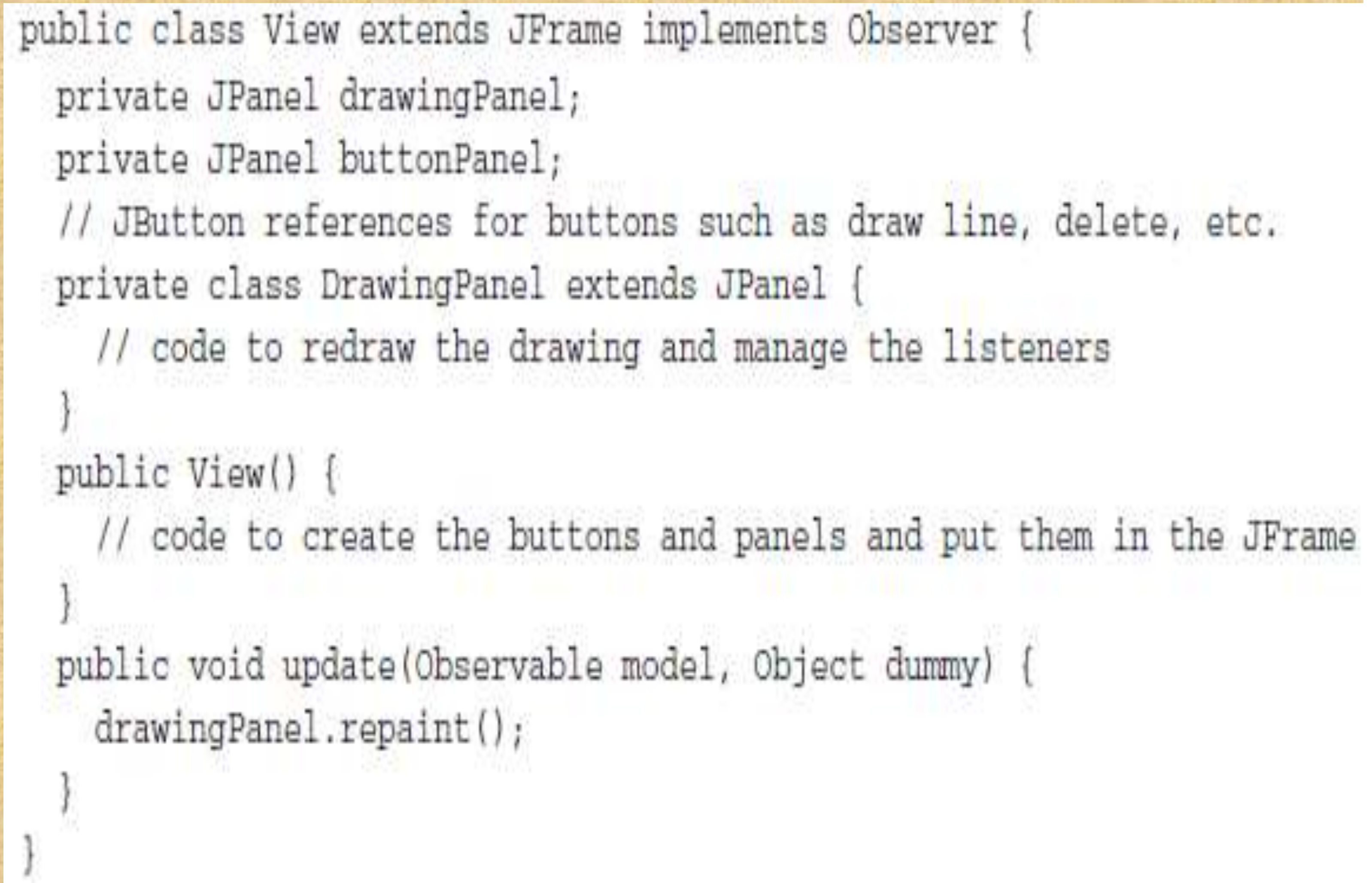
- The class must keep track of the current shape being created, and this is accomplished by having the following fields within the class.
- When the view receives a button click to create a line, it calls one of the following controllermethods.
- The controller supplies three versions of the makeLine 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);
}
```

Implementation of the View Class

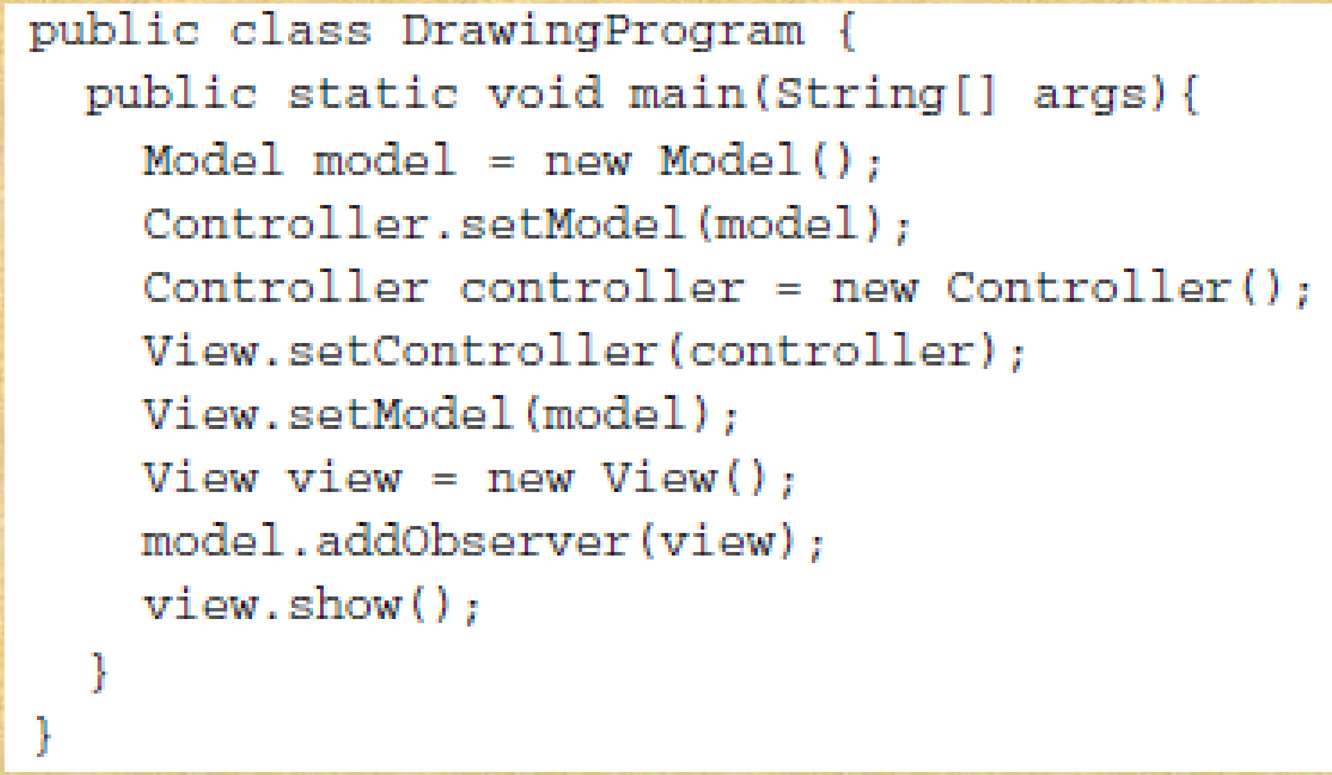
- The view maintains two panels: one for the buttons and the other for drawing the items.



```
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 Driver Program

- The driver program sets up the model.
- 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();
  }
}
```

11.7 Implementing the Undo Operation

In the context of implementing the undo operation, a few issues need to be highlighted.

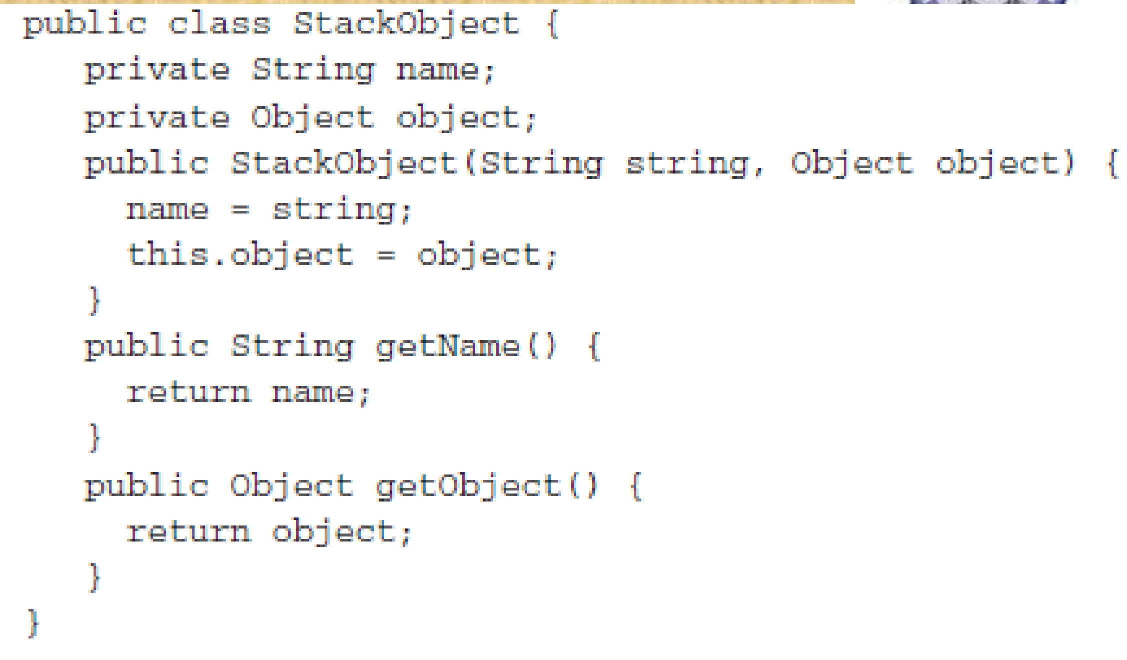
1. *Single-level undo versus multiple-level undo*
2. *Undo and redo are unlike the other operations*
3. *Not all things are undoable*
4. *Blocking further undo/redo operations*
5. *Solution should be efficient*

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.

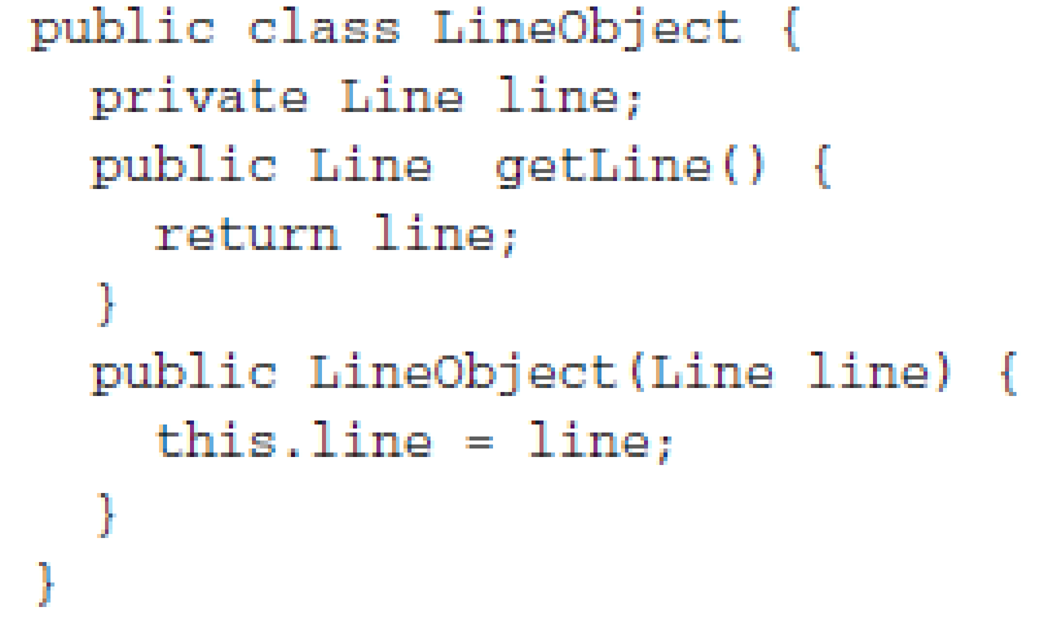
Implementing the Undo Operation contd...

- 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;
   }
}
```

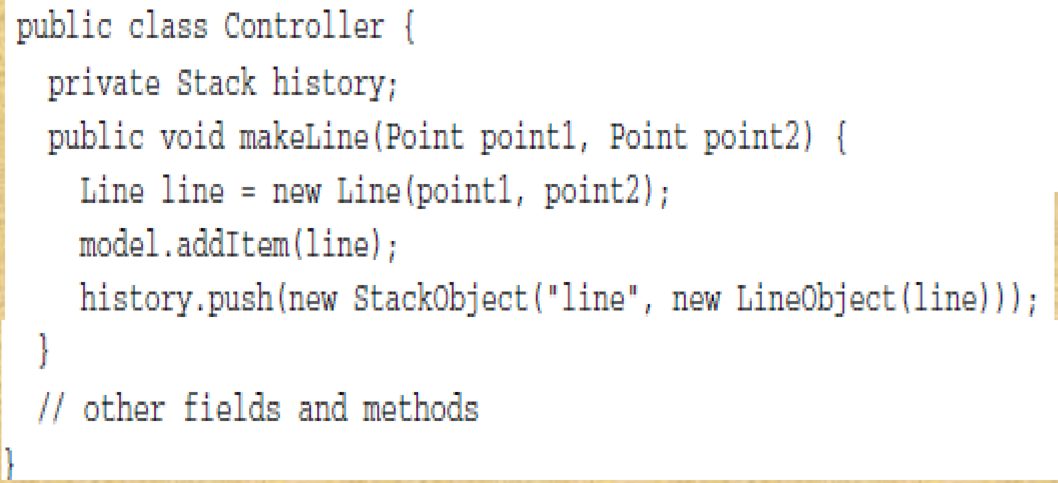
- Each command has an associated object that stores the data needed to undo it. The class corresponding to the operation of adding a line is shown below.



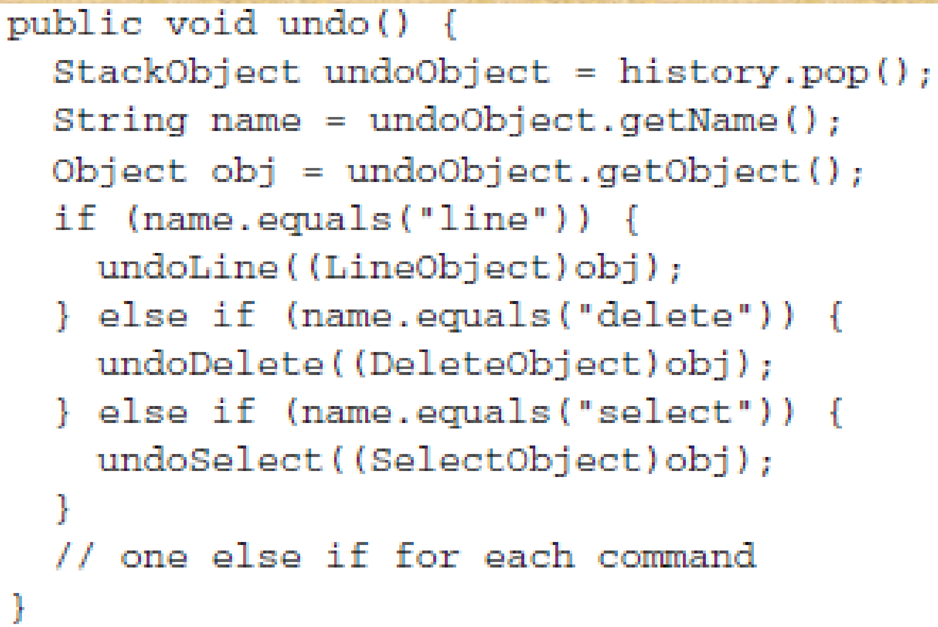
```
public class LineObject {
  private Line line;
  public Line  getLine() {
    return line;
  }
  public LineObject(Line line) {
    this.line = line;
  }
}
```

Implementing the Undo Operation contd…

- When the operation for adding a line is completed, the appropriate StackObject instance is created and pushed onto the stack.

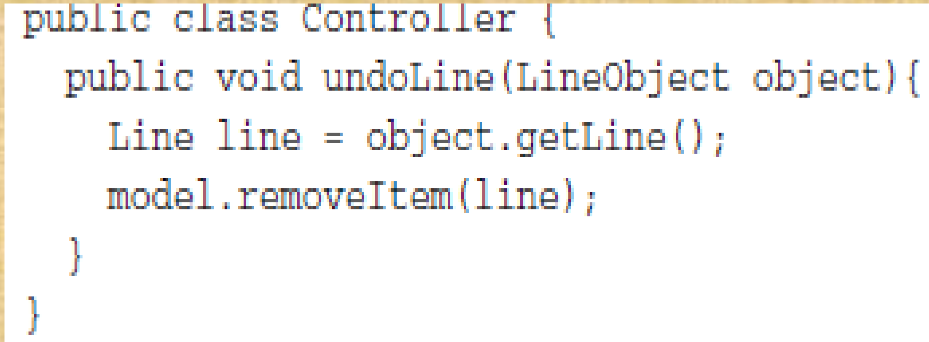


```
public class Controller {
  private Stack history;
  public void makeLine(Point point1, Point point2) {
    Line line = new Line(point1, point2);
    model.addItem(line);
    history.push(new StackObject("line", new LineObject(line)));
  }
  // other fields and methods
}
```



```
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
}
```

Decoding is simply a matter of popping the stack reading the String.

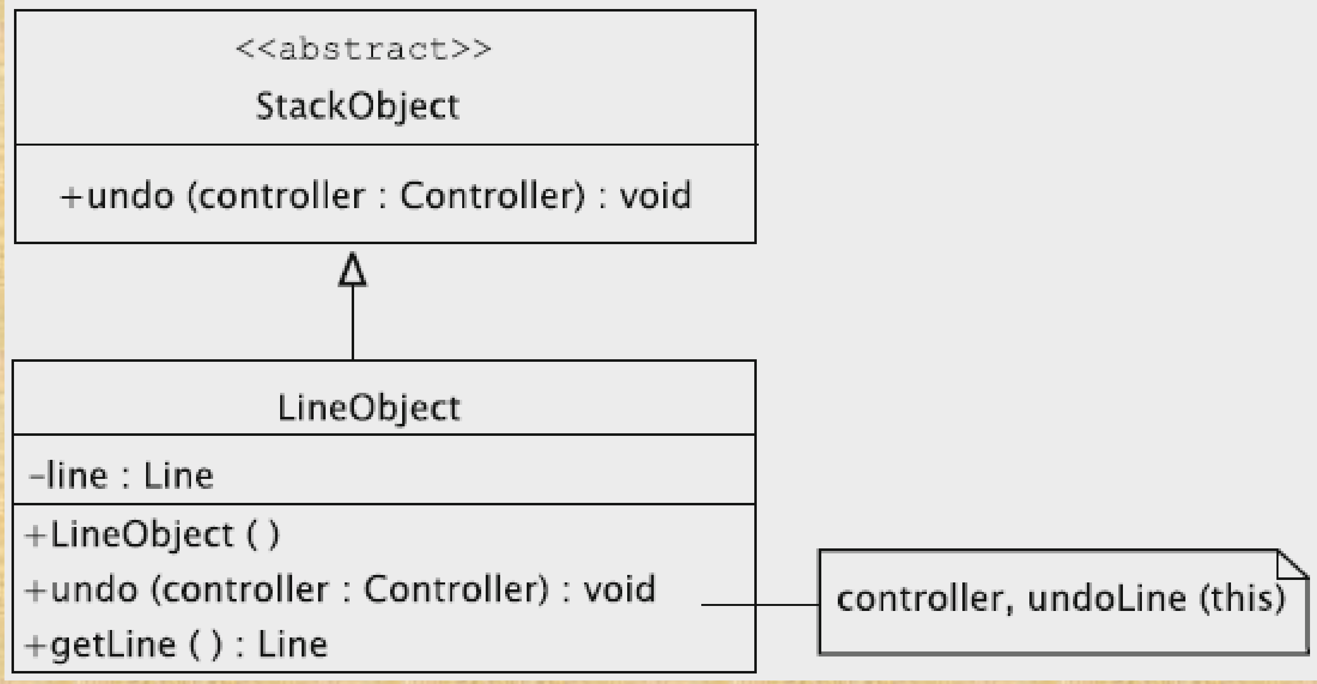


```
public class Controller {
  public void undoLine(LineObject object){
    Line line = object.getLine();
    model.removeItem(line);
  }
}
```

- Finally, undoing is simply a matter of retrieving the reference to and removing the line form the model.

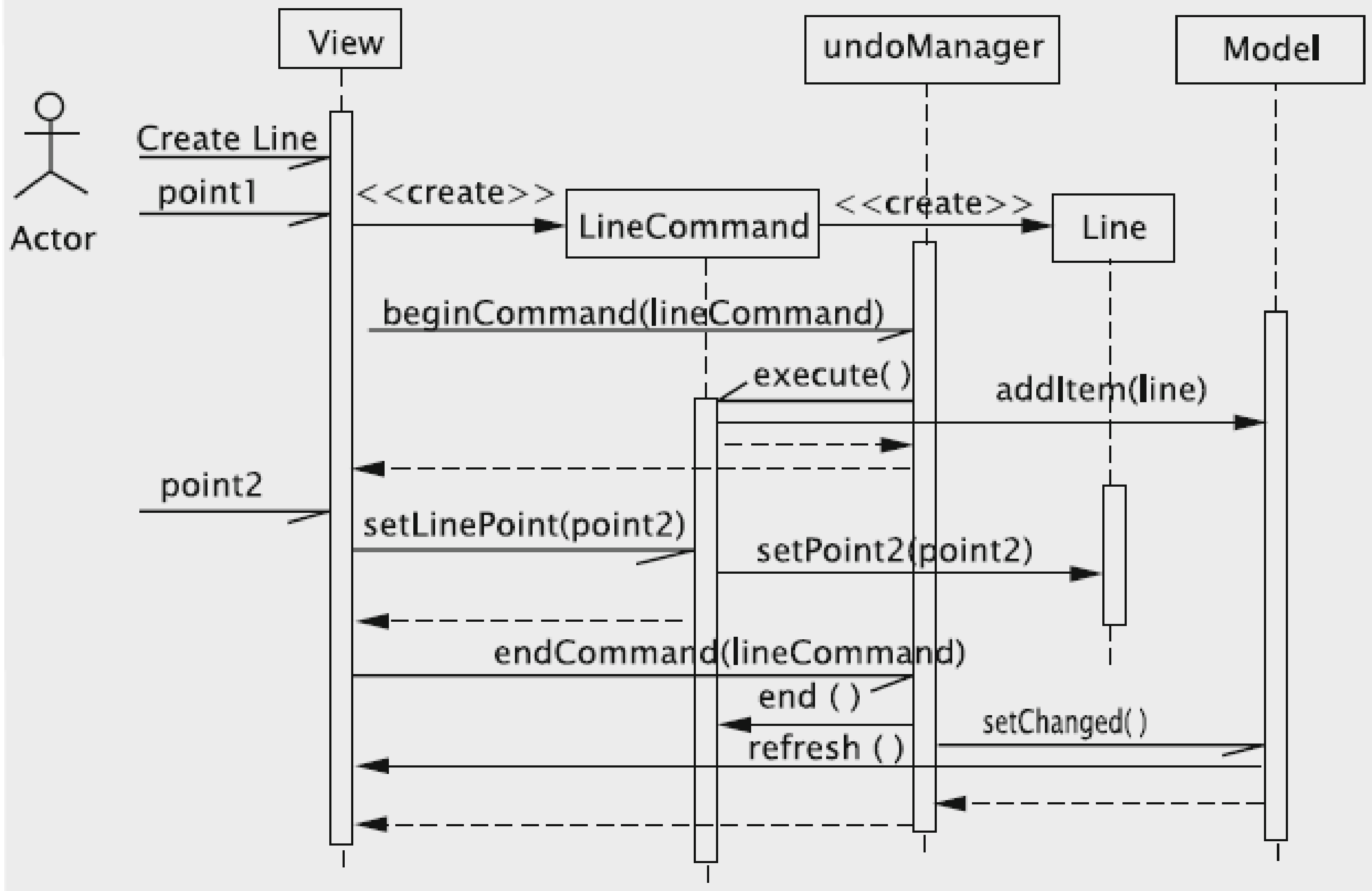
Implementing the Undo Operation contd…

- Representing the drawing of a line


<<abstract>>
StackObject
+undo (controller : Controller) : void
LineObject
-line : Line
+LineObject ( )
+undo (controller : Controller) : void
+getLine ( ) : Line
controller, undoLine (this)

Implementing the Undo Operation contd…

Sequence diagram for adding a line



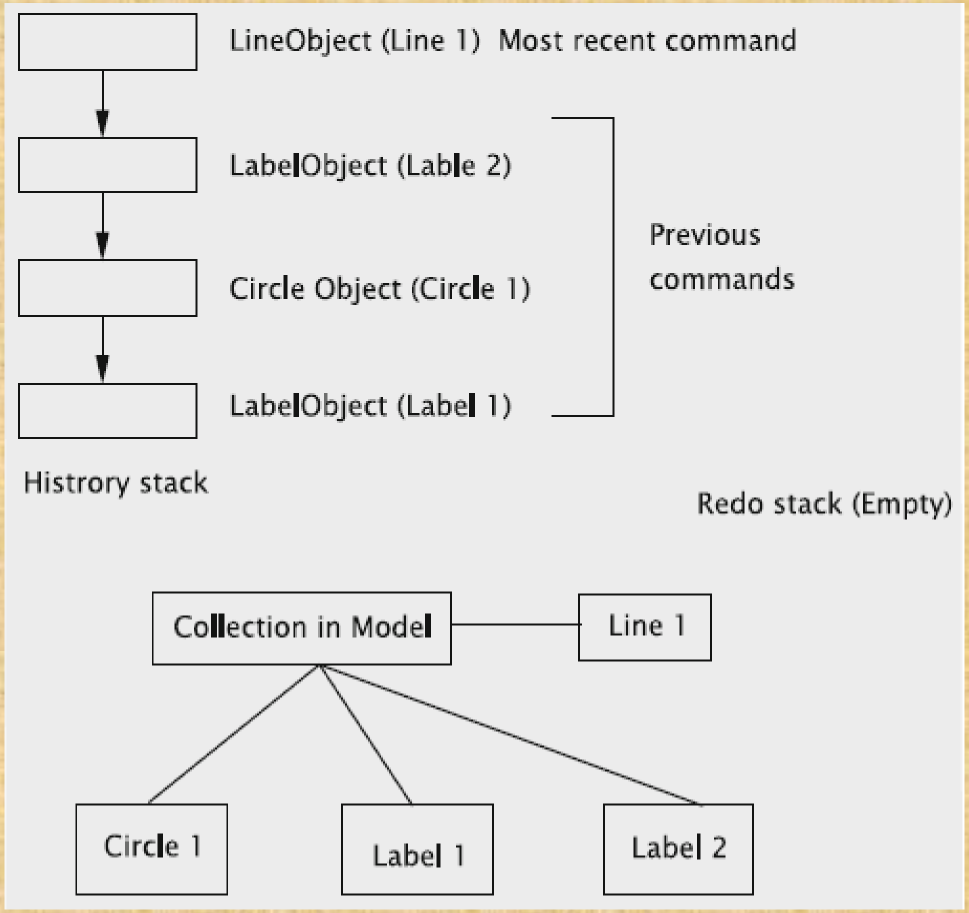
- The central idea behind the command pattern is to employ two stacks:

1. one for storing the commands that can be undone (history stack) and
2. The other for maintaining the commands that may be redone (redo stack).

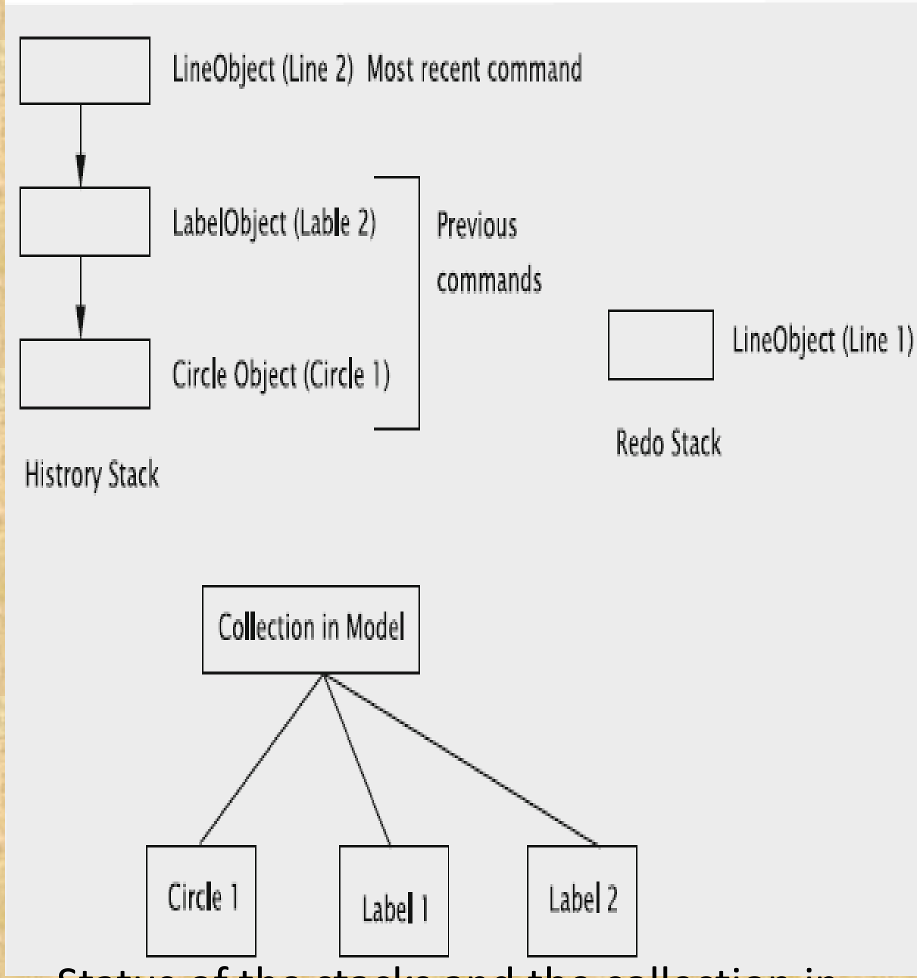
The class UndoManager maintains these stacks:

1. The undo manager plays the role of the controller
2. as soon after the command object is created
3. view informs the undo manager, which is then expected to initiate its bookkeeping operations.
4. when the view has received all of the data needed to complete the command, it notifies the UndoManager once more.
5. The two methods beginCommand and endCommand are for these two purposes.

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, thecommand is undone and transferred to the redo stack.



Status of the stacks and the collection in the model



Status of the stacks and the collection in the model after undo

Incomplete command

- 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.

EX: 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.

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.

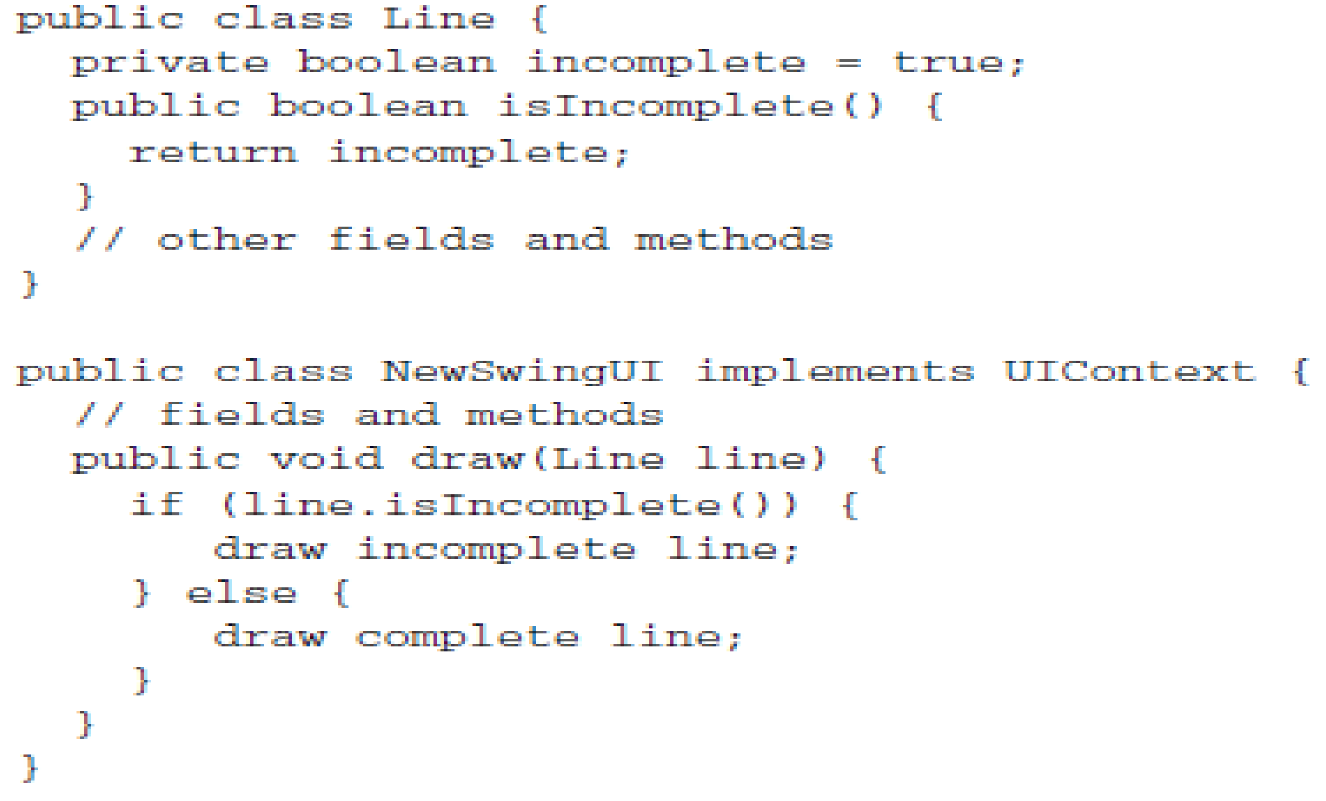
Implementation : Refer page number 368 to 370

Drawing Incomplete Items

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.
2. Some fields in an incomplete item might not have 'proper' values.

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;
      }
   }
}
```

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.

Ex1: 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.

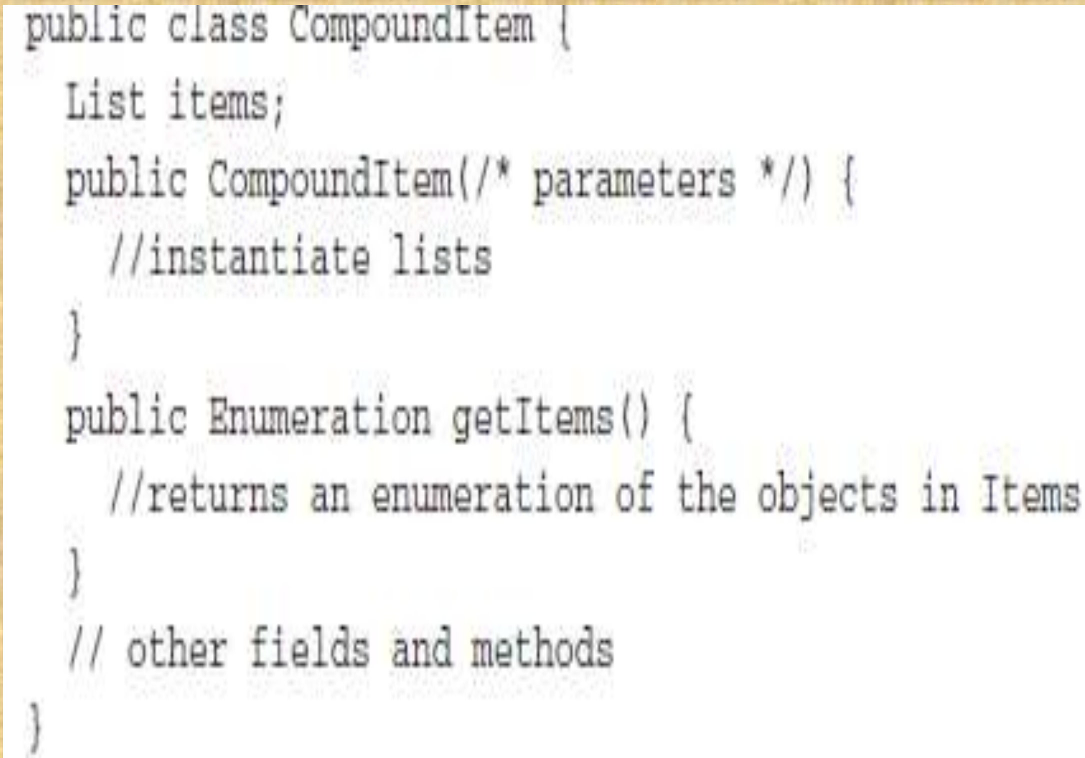
Ex2: In a system for drawing electrical circuits, a set of components interconnected in a particular way could be clustered together as a 'sub-circuit' that can then be treated as a single unit.

The process for creating such a 'compound' object would be as follows:

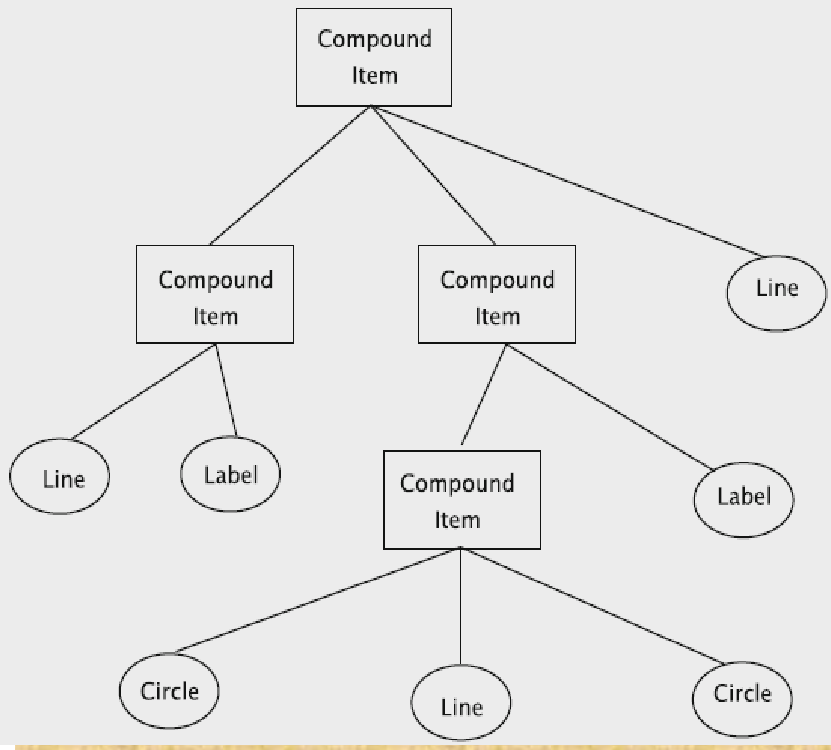
- *The user would select the items that have to be combined by clicking on them.*
- *The system would then highlight the selected items.*
- *The user then requests the operation of combing the selected items into a compound object, and the system combines them into one.*

Adding a New Feature   Contd…

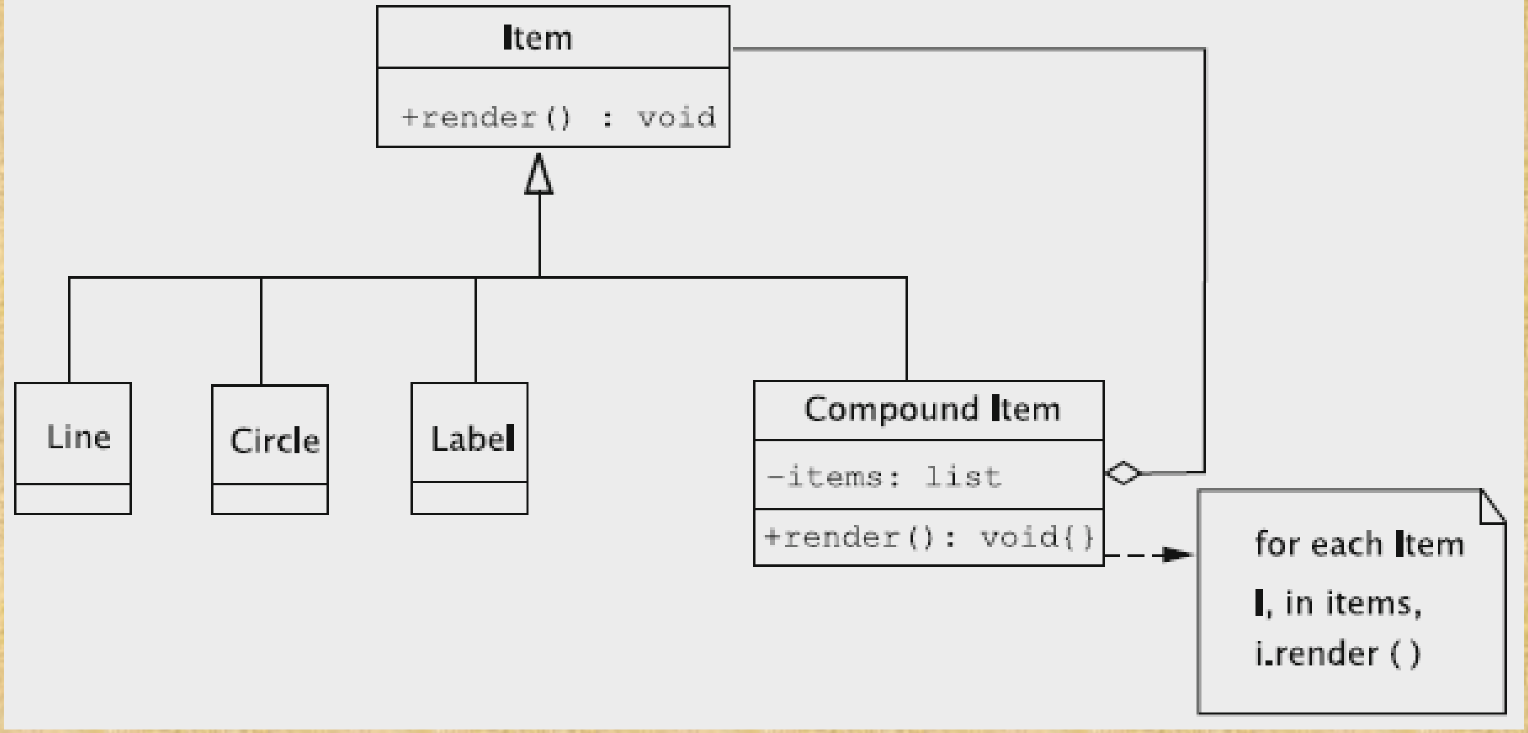
- Once a compound object has been created, it can be treated as a any other object.
- This process can be *iterated,* i.e., a compound object can be combined with other objects (which could themselves be compound or simple objects) to create another compound object.
- The system also allows the user 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.



```
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
}
```



Adding a New Feature contd…



Composite structure of the item hierarchy

Pattern-Based Solutions

- 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.

1. 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:

1. *They have evolved over time*
2. *A given pattern is usually applicable for a certain class of software system*
3. *The need for these is not obvious to the untrained eye*

Pattern-Based Solutions contd…

2. 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.

3. 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.
- 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.

11.10 Examples of Architectural Patterns

1. **The Repository :** This architecture is characterised by the presence of a single data structure called the *central repository*. Subsystems access and modify the data stored in this. An example : *managing an airline.*
2. **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 amountof 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.
   - Example: The world-wide-web is probably the best example of such an architecture.

3. 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.
- An example of this would be that of processing incoming/outgoing data packets over a computer network.

11.11 Discussion and Further Reading

1. *Separating the View and the Controller*
2. *The Space Overhead for the Command Pattern*
3. *How to Store the Items*
4. *Exercising Caution When Allowing Undo*
   - **What Should be Saved to Undo an Operation?**
   - **Designing and Implementing with Undo in Mind**
5. *Synchronising Updates*

Module-5

Designing with Distributed Objects (Chapter12,Book1)

Designing with Distributed Objects

- As businesses grow, =>set up operations over large geographic areas => may span multiple states or even countries and often find it desirable to process data at their point of origin or create results at the location where they are needed.
- Distributed processing offers a number of advantages.

1. It is more economical and efficient to process data at the point of origin.
2. Distributed systems make it easier for users to access and share resources.
3. They also offer higher reliability and availability:
   - ➢ failure of a single computer does not cripple the system as a whole.

4. It is also more cost effective to add more computing power.

Distributed computing is not without its share of drawbacks :

1. The software for implementing them is complex
   - It must coordinate actions between a number of possibly heterogeneous computer systems;
   - if data is replicated, the copies must be made mutually consistent.
2. Data access may be slow because information may have to be transferred across communication links
3. 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.

Two approaches to building a distributed system.

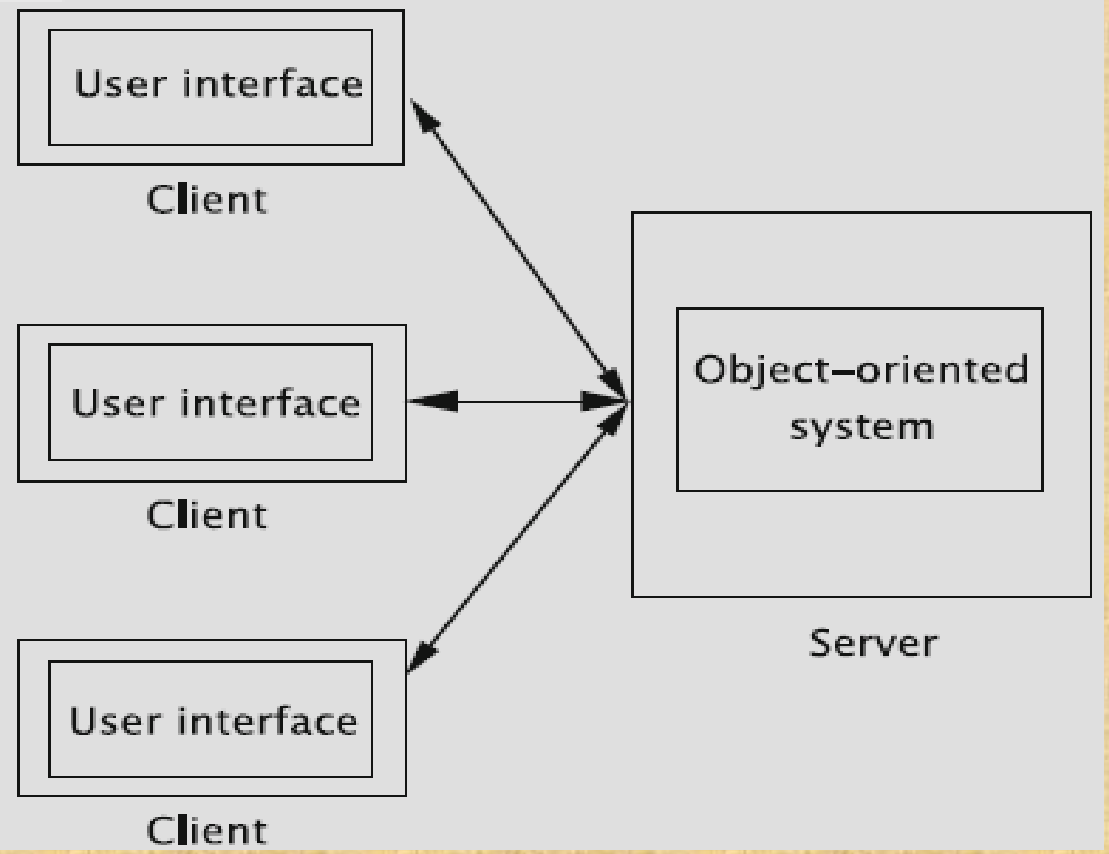
1. The first mechanism uses *Java Remote Method Invocation (Java RMI),* which is a piece of software, generally called middleware, that helps mask heterogeneity.
2. The second approach uses the world-wide web itself to access data processed at remote sites.

Client/Server Systems

Distributed systems can be classified into

1. **Peer-to-peer systems and**
   - In the former, 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.
   - Many applications can use this model and these days the software at many clients are *web browsers*.

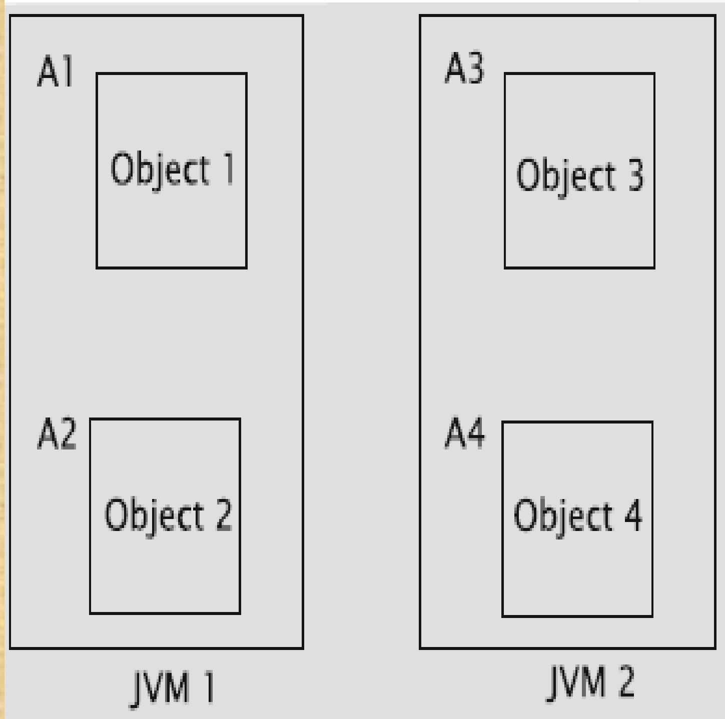
Basic Architecture of Client/Server Systems



Client/Server systems

- Each client runs a program that provides a user interface, which may or not be a GUI.
- The server hosts an object-oriented 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

Difficulty in accessing objects in a different JVM



This difficulty can be handled in one of two ways:

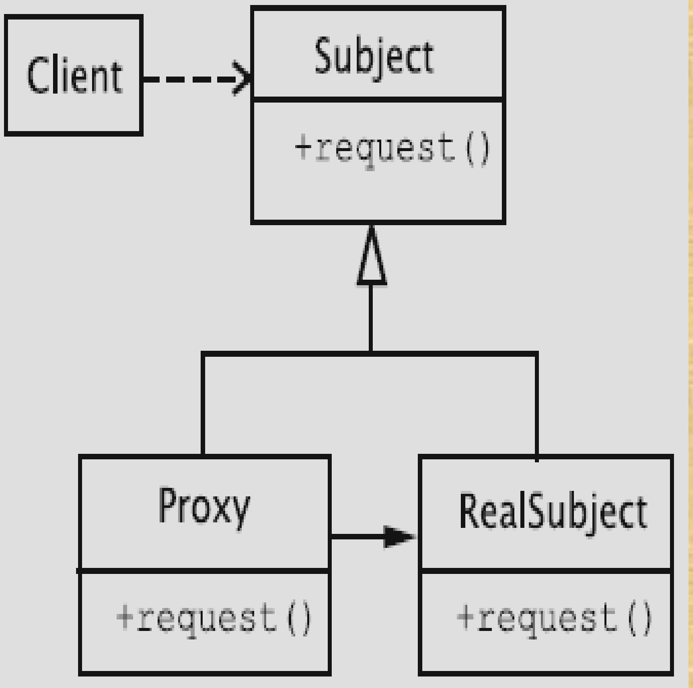
1. ***By using object-oriented support software: (Java RMI)***

- 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.

2. By avoiding direct use of remote objects by using the Hyper Text Transfer Protocol (HTTP).

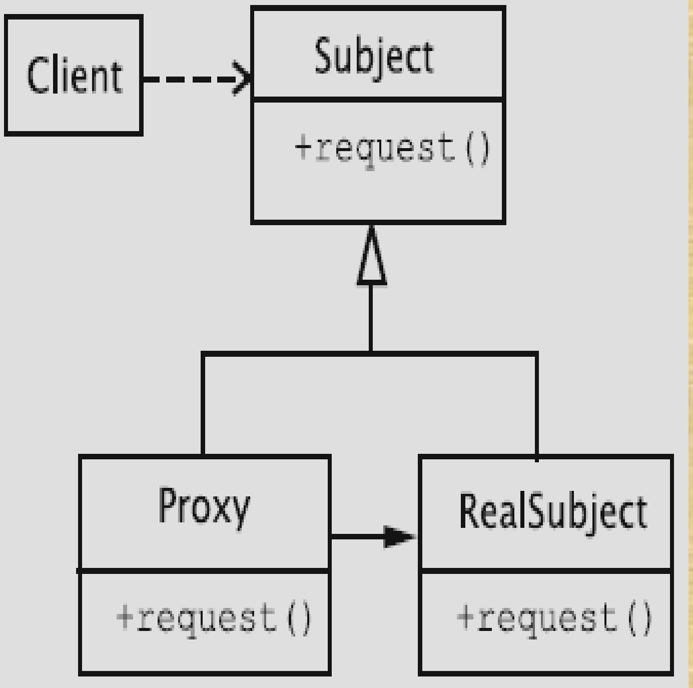
- The system sends requests and collects responses via encoded text messages.
- The object(s) to be used to accomplish the task, the parameters, etc., are all transmitted via these messages.

Java Remote Method Invocation



- Java RMI is to support the building of Client/Server systems
- 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.
- This pattern is used when it is inefficient or inconvenient (even impossible, perhaps) to use the actual object.
- The proxy pattern creates a *proxy* object at each client site that accesses the remote object.
- The proxy object implements all of the remote object's operations that the remote object wants to be available to the client.

Java Remote Method Invocation contd…



- 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.

Java Remote Method Invocation contd…

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.

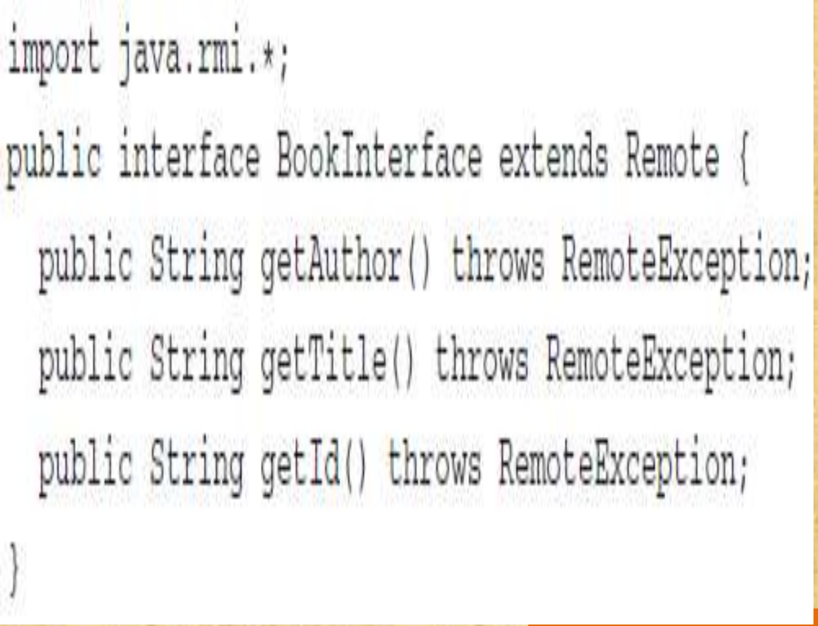
1. *Remote Interfaces*

a. Define the system functionality that will be exported to clients=> implies the creation of a Java interface

b. The functionality exported of a remote object is defined via what is called a *remote interface*.

c. A remote interface is a Java interface that extends the interface java.rmi.Remote, which contains no methods and simply serves as a marker.

d. Clients are restricted to accessing methods defined in the remote interface. Called **remote method invocation**



```
import java.rmi.*;
public interface BookInterface extends Remote {
  public String getAuthor() throws RemoteException;
  public String getTitle() throws RemoteException;
  public String getId() throws RemoteException;
}
```

Remote method invocations can fail due to a number of reasons:

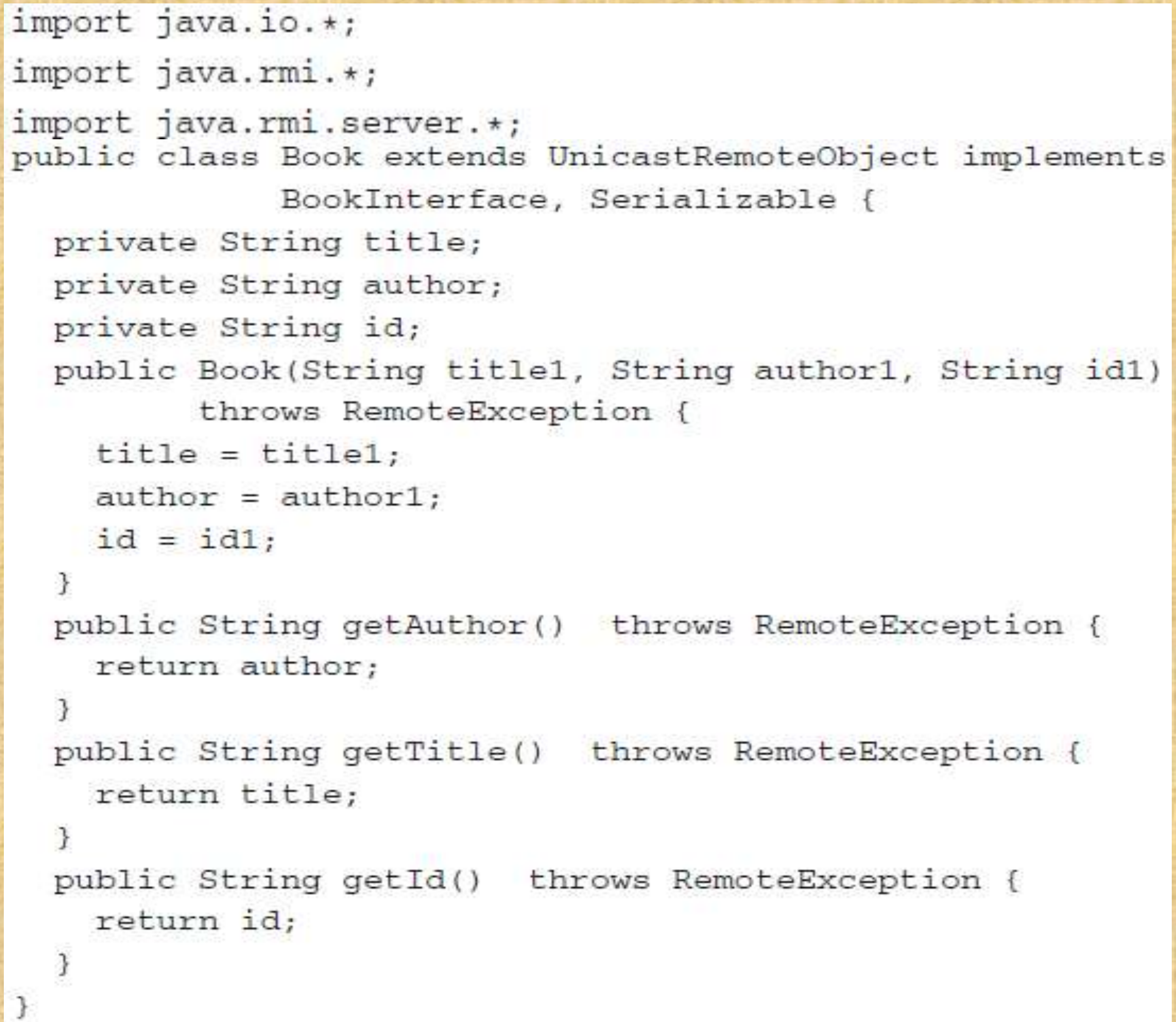
1. The remote object may have crashed,
2. The server may have failed, or
3. The communication link between the client and the server may not be operational, etc.

as a result, all remote methods must be declared to throw this exception.

2. Implementing a Remote Interface:

1. After the remote interfaces are defined, the next step is to implement them via *remote classes.*
2. Parameters to and return values from a remote method may be of primitive type, of remote type, or of a local type.
3. All arguments to a remote object and all return values from a remote object must be serializable.
4. They also implement the java.io.Serializable interface.
5. Parameters of non-remote types are passed by copy
6. They are serialized using the object serialization mechanism,
7. They too must implement the Serializable interface.
8. Remote objects must somehow be capable of being transmitted over networks.
9. A convenient way to accomplish this is to extend the class java.rmi.server.UnicastRemoteObject.

The implementation of BookInterface



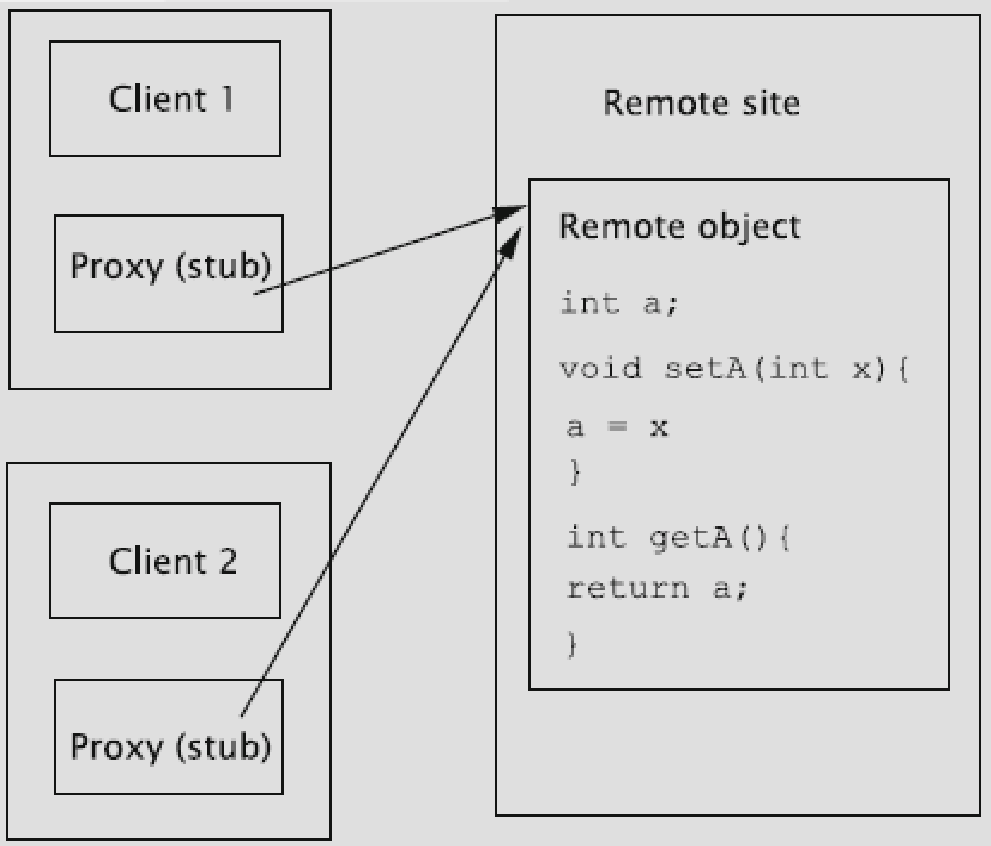
```
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
public class Book extends UnicastRemoteObject implements
             BookInterface, Serializable {
  private String title;
  private String author;
  private String id;
  public Book(String title1, String author1, String id1)
        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;
  }
}
```

Book must be compiled using the RMI compiler by invoking the command rmic as below.

rmic Book

Passing of remote objects as references:

- When an exported remote object is passed as a parameter or returned from a remote method call, the stub for that remote object is passed instead of the object itself.
- The stub itself 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.

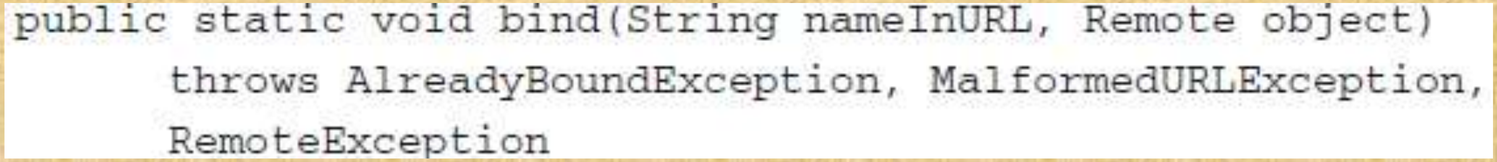


```
int a;
void setA(int x){
a = x
}
int getA(){
return a;
}
```

- Parameters or return values that are *not* remote objects are passed by value.
- Any changes to the object's state by the client are reflected only in the client's copy, not in the server's instance.
- Similarly, if the server updates its instance, the changes are not reflected in the client's copy.

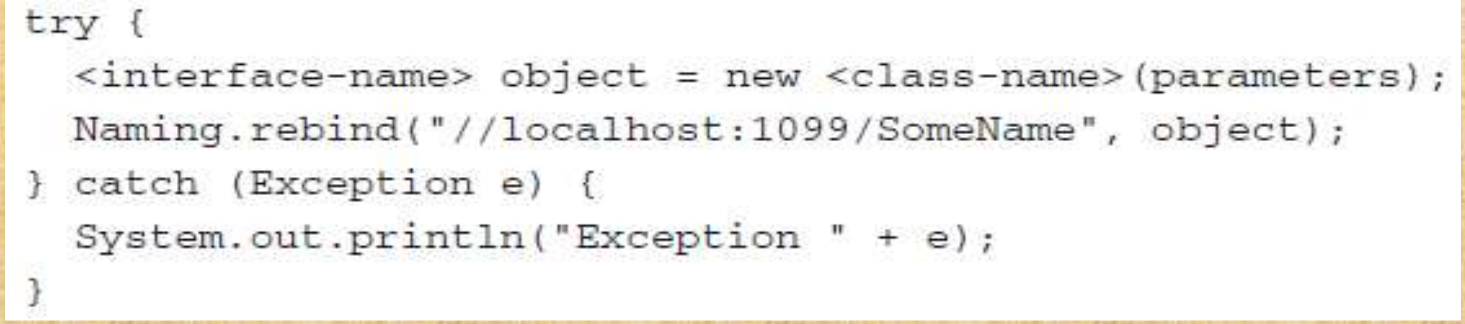
3. *Creating the Server*

- 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:



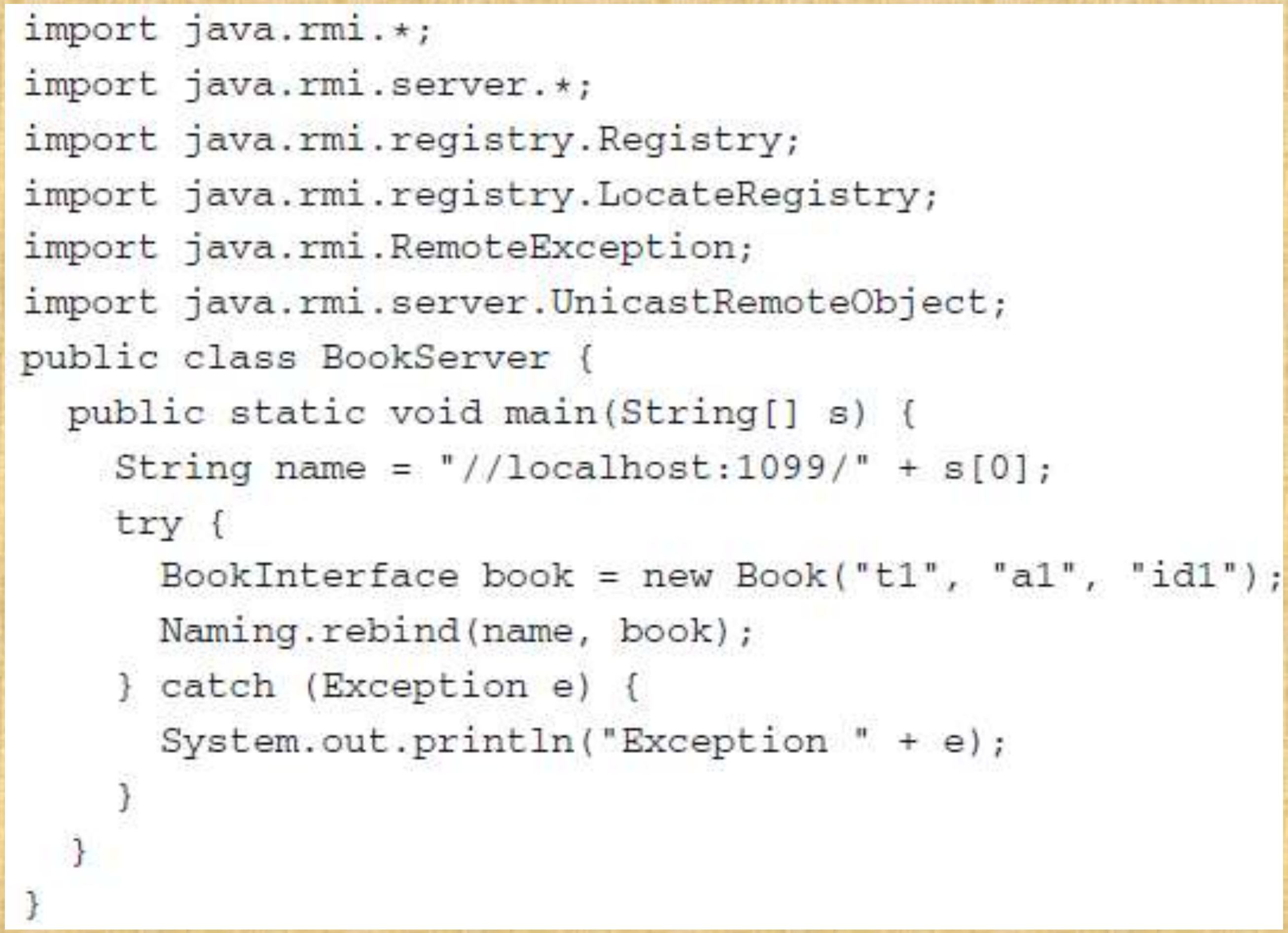
```
public static void bind(String nameInURL, Remote object)
      throws AlreadyBoundException, MalformedURLException,
      RemoteException
```

The 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);
    }
  }
}
```

4. The Client

A client may get a reference to the remote object it wants to access in one of two ways:

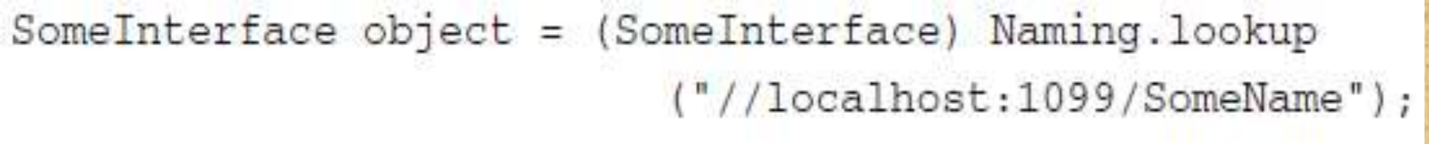
1. **It can obtain a reference from the Naming class using the method lookup.**
2. **It can get a reference as a return value from another method call.**

4. *The Client*

A client may get a reference to the remote object it wants to access in one of two ways:

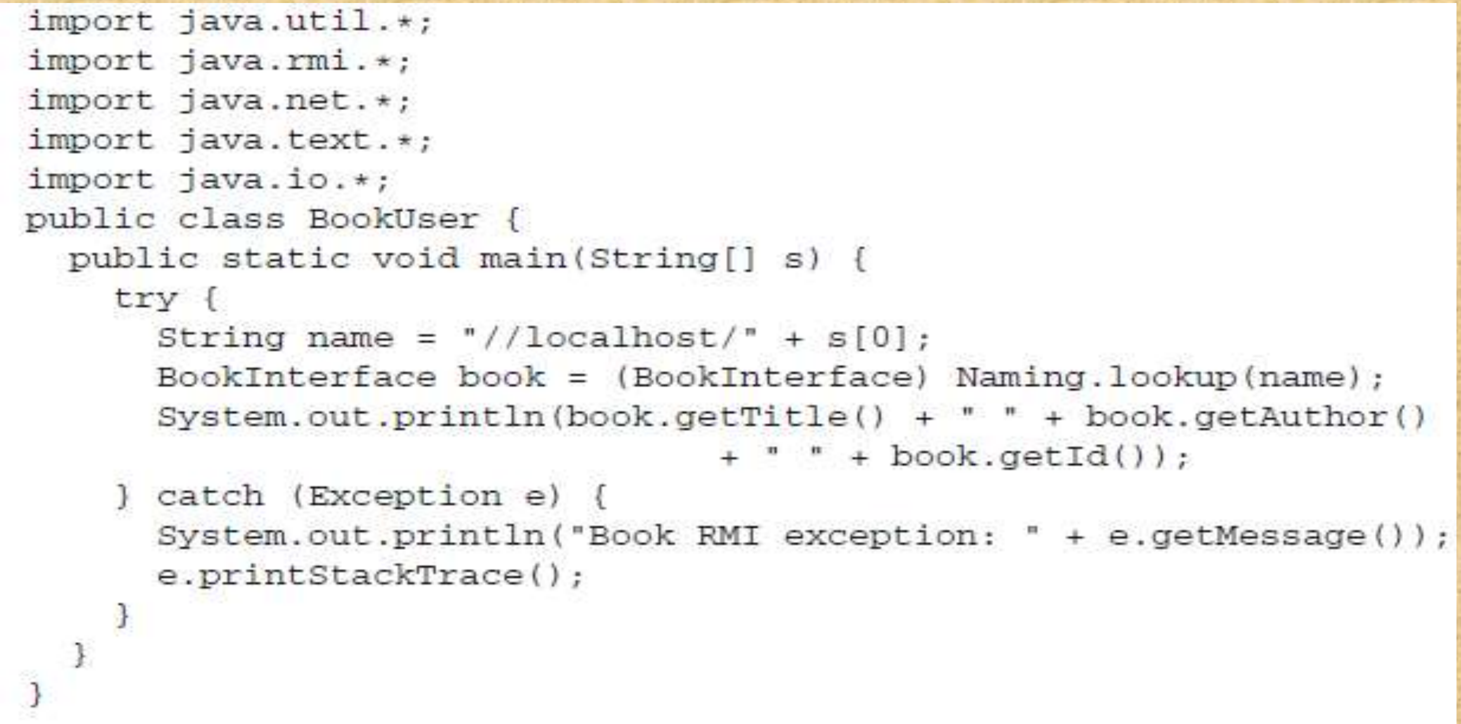
1. **It can obtain a reference from the Naming class using the method lookup.**

we assume that an object of type SomeInterface has been entered into the local registry under the name SomeName.



```
SomeInterface object = (SomeInterface) Naming.lookup
                                  ("//localhost:1099/SomeName");
```

The client can invoke remote methods on the object. the BookInterface object are called and displayed



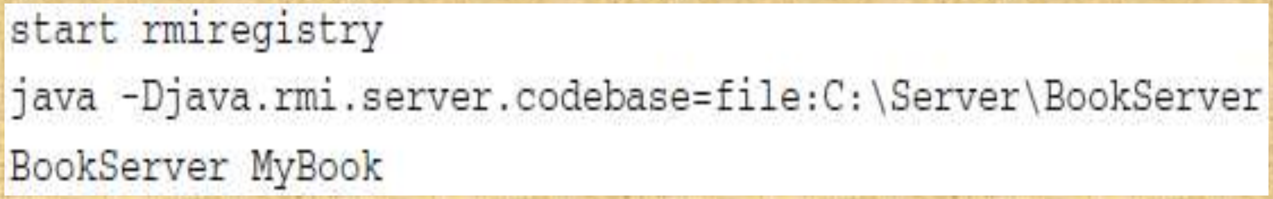
```
import java.util.*;
import java.rmi.*;
import java.net.*;
import java.text.*;
import java.io.*;
public class BookUser {
  public static void main(String[] s) {
    try {
      String name = "//localhost/" + s[0];
      BookInterface book = (BookInterface) Naming.lookup(name);
      System.out.println(book.getTitle() + " " + book.getAuthor()
                                  + " " + book.getId());
    } catch (Exception e) {
      System.out.println("Book RMI exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

12.2.5 Setting up the System

- 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.
- Then compile the three Java files in server and then invoke the command

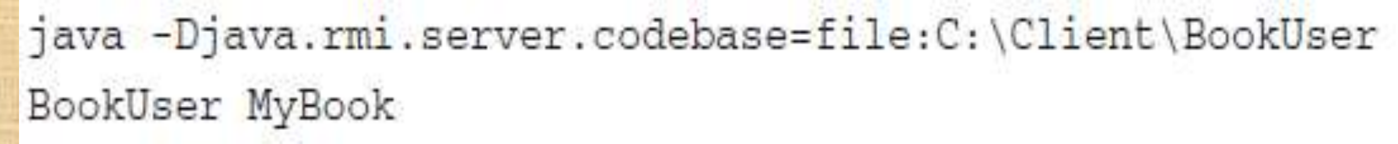
  rmic Book
- This command creates the stub file Book_Stub.class. Copy the client program into client and compile it.

Run RMI registry and the server program using the following commands (on Windows).



```
start rmiregistry
java -Djava.rmi.server.codebase=file:C:\Server\BookServer
BookServer MyBook
```

Finally, run the client as below from the client directory.



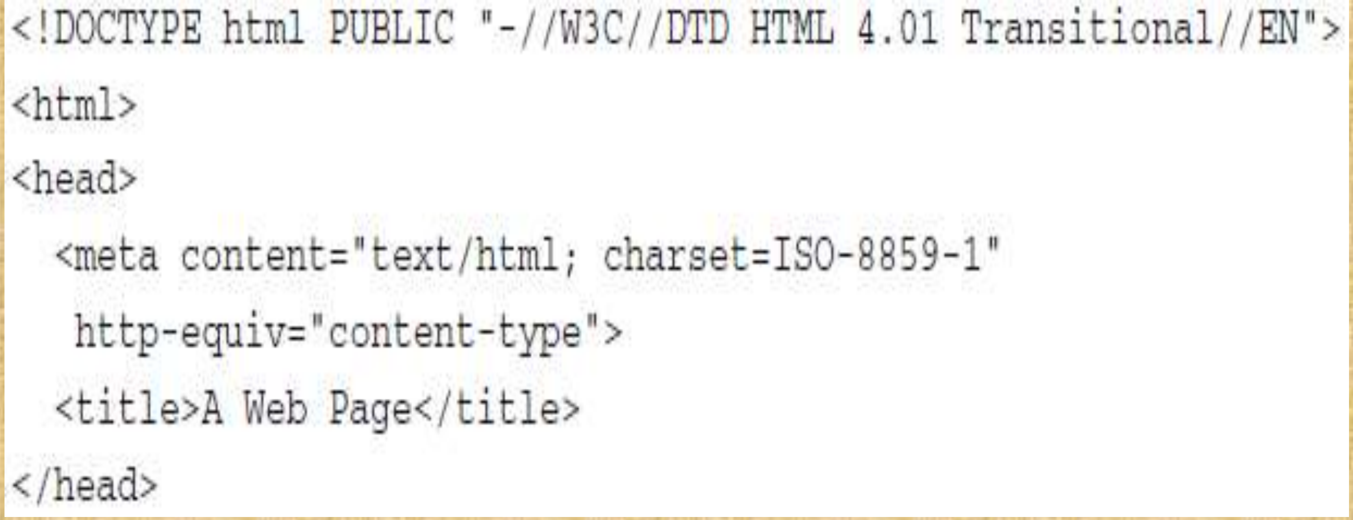
```
java -Djava.rmi.server.codebase=file:C:\Client\BookUser
BookUser MyBook
```

12.3 Implementing an Object-Oriented System on the Web

- world-wide web is the most popular medium for hosting distributed applications.
- The browser acts as a general purpose client that can interact with any application that talks to it using the Hyper Text Transfer Protocol (HTTP).
- All business logic and data processing take place at the server.
- Typically, the browser receives web pages from the server in HTML and displays the contents according to the format, a number of tags and values for the tags, specified in it.
- The browser simply acts as a ‘dumb’ program displaying whatever it gets from the application and transmitting user data from the client site to the server.
- The HTML program shipped from a server to a client often needs to be customised : the code has to suit the context.
- This requires that HTML code for the screen be dynamically constructed. This is done by code at the server.
- For server-side processing competing technologies such as Java Server Pages and Java Servlets, Active Server Pages (ASP), and PHP.

12.3.1 HTML and Java Servlets

- Any system that ultimately 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:**



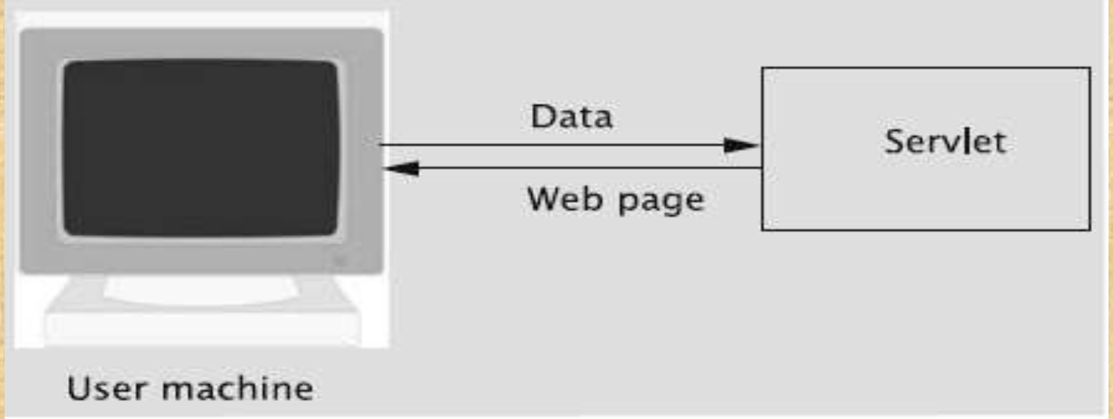
```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=ISO-8859-1"
   http-equiv="content-type">
  <title>A Web Page</title>
</head>
```

- There are two primary ways in which form data is encoded by the browser:
- One is GET and the other is POST.
- GET means that form data is to be encoded into a URL while
- POST makes data appear within the message itself.

- **Refer page number 400 and on for other HTML tags.**

12.3.2 Deploying the Library System on the World-Wide Web:

How servlets and HTML cooperate to serve web pages?



- HTML page is displayed on the client’s browser.
- The page includes, among other things, a form that allows the user to enter some data.
- The client makes some entries in the form’s fields and submits them, say, by clicking a button.
- The data in the form is then transmitted to the server and given to a Java servlet, which processes the data and generates HTML code that is then transmitted to the client’s browser, which displays the page.

12.3.2 Deploying the Library System on the World-Wide Web: contd…

1. Developing User Requirements
   - ➢ Interface requirements
2. Design and Implementation
   a. Structuring the files
   b. How to remember a user
   c. Configuration
   d. Structure of servlets in the web-based library system
   e. Execution flow

1. Developing User Requirements

- Determine the system requirements:

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: *superusers* and *ordinary members*
   - ➢ superusers can execute any command when logged in from a terminal in the library, whereas ordinary members cannot access some ‘privileged commands’.

   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.

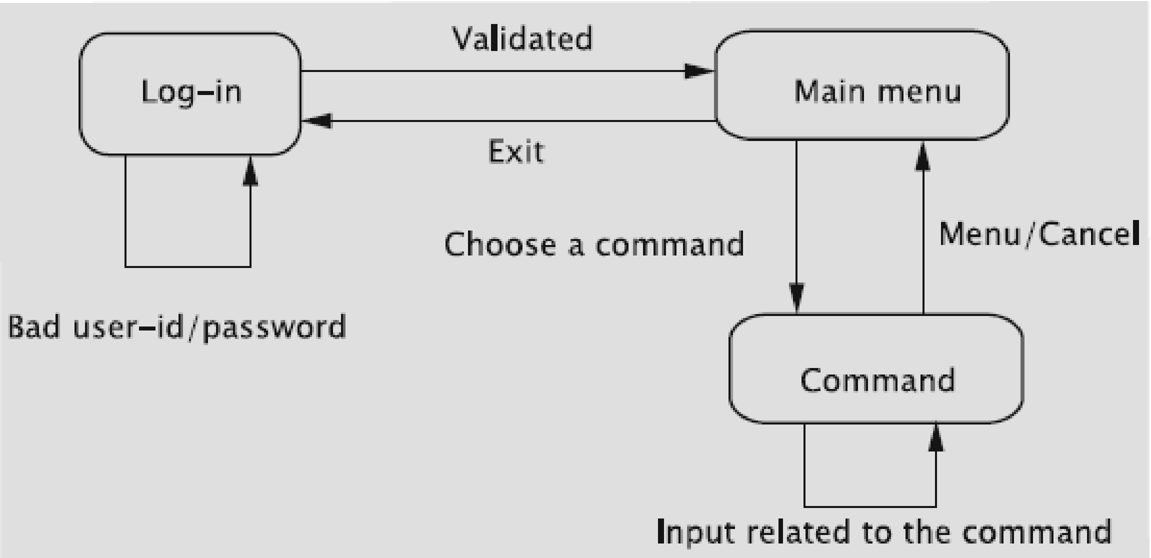
   b. Ordinarymembers and superusersmay invoke the following commands: issue and renew books, place and remove holds, and print transactions.

   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.

Interface requirements

- Large number of sequences of interactions are possible between the user and the interface.
- Depict the requirements through state transition diagrams.

Logging in and the Initial Menu:

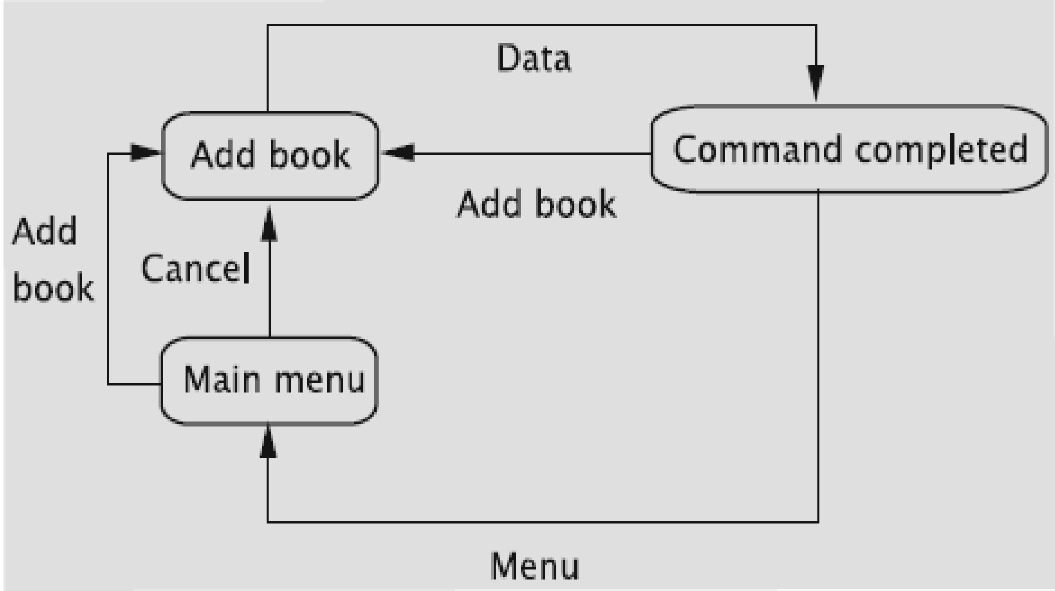


1. The Issue Book command is available only if the user logs in from a terminal in the library.

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.

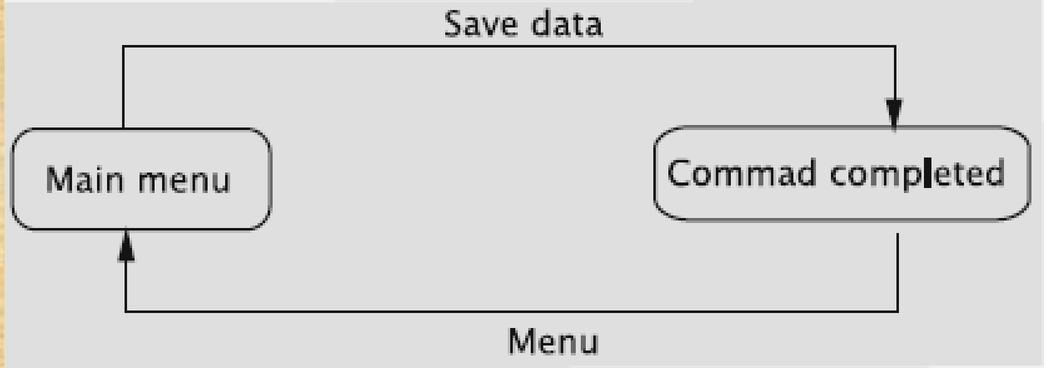
3. Certain commands are available only to superusers who log in from a library terminal: these are for returning or deleting books, adding members and books, processing holds, and saving data to and retrieving data from disk.

Add Book: State transition diagram for add book



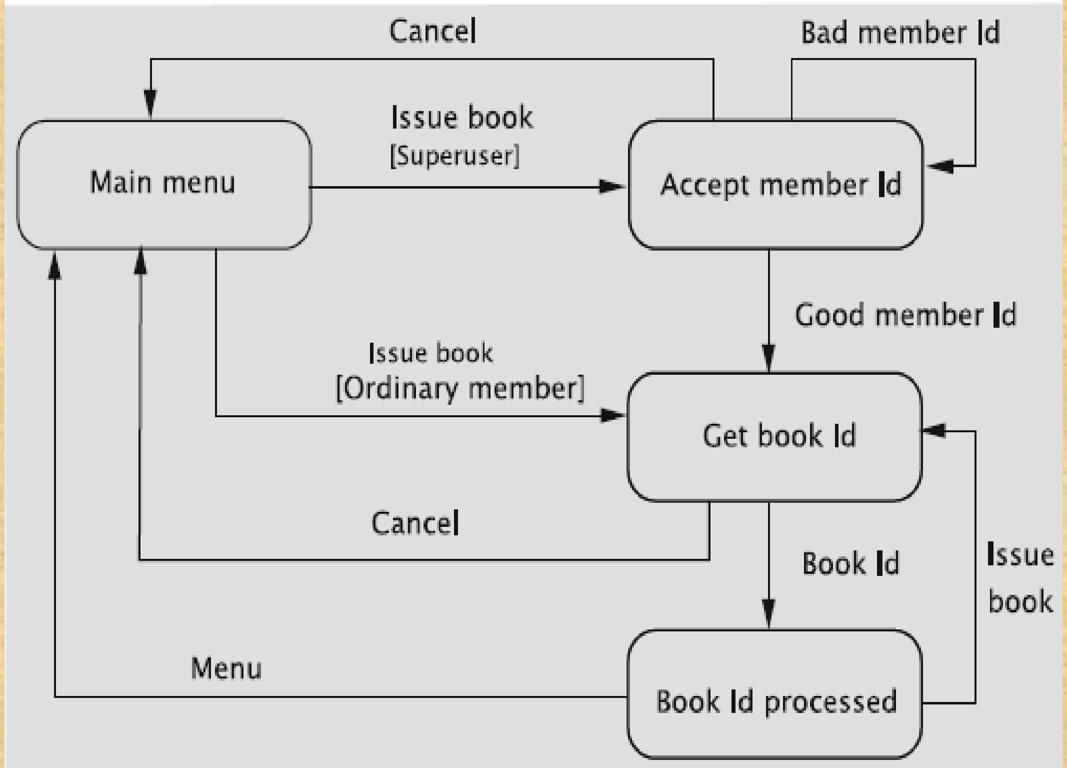
- When the command to add a book is chosen, the system constructs the initial screen to add a book,
- which should contain three fields for entering the title, author, and id of the book, and then display it and enter the Add Book state.
- By clicking on a button, it should be possible for the user to 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.

State transition diagram for saving data



- 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.

State transition diagram for issuing books

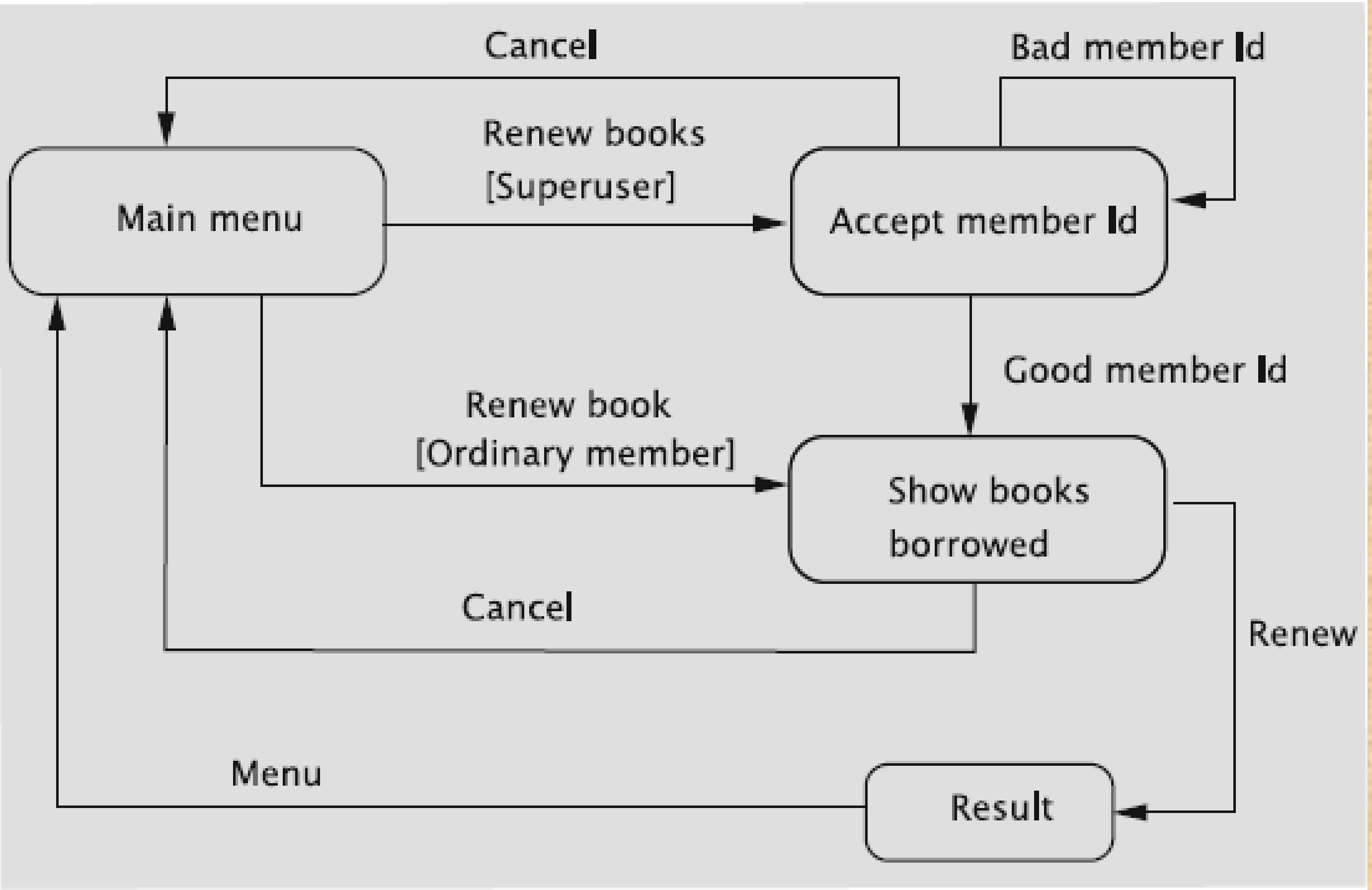


A book may be checked out in two different ways:

1. A member is allowed to check it out himself/herself.
2. he/she may give the book to a library staff member, who checks out the book for the member

- In the first case, the system already has the user’s member id, so that should not be asked again.
- In the second case, the library staff member needs to input the member id to the system followed by the book id.
- After receiving a book id, the system must attempt to check out the book. Whether the operation is successful or not, the system enters the Book Id Processed state.

State transition diagram for renewing books



2. Design and Implementation

To deploy the system on the web, we need the following:

1. Classes associated with the library, create classes such as Library, Member, Book, Catalog, and so on.
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 and the objects that actually do the processing. 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.

- **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.
- **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.

Examples of HTML file fragments

For implementation of library application refer page number from 410

How to remember a user

- Servlets typically deal with multiple users.
- When a servlet receives data from a browser, it must some how 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.
- Typical web transactions involve multiple request–response pairs.
- 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.
- 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();
- Requests other than log in requires the user to be logged in. The following code evaluates to true if the user does not have a session: that is, the user has not logged in. **request.getSession(false) == null**

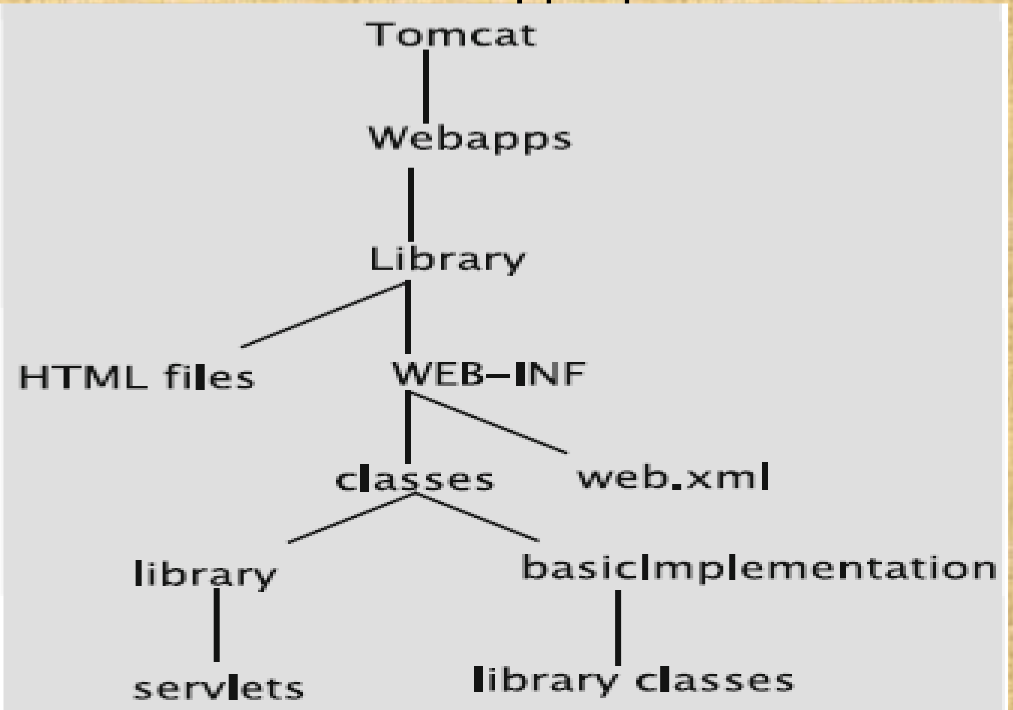
- A session object can be used to store information about the session. In the library system, we would like to store the user id, the type of terminal from which the user has logged in, and some additional information related to the user.

1. void setAttribute(String name, Object value)
2. Object getAttribute(String name)
3. void removeAttribute(String name)

Configuration

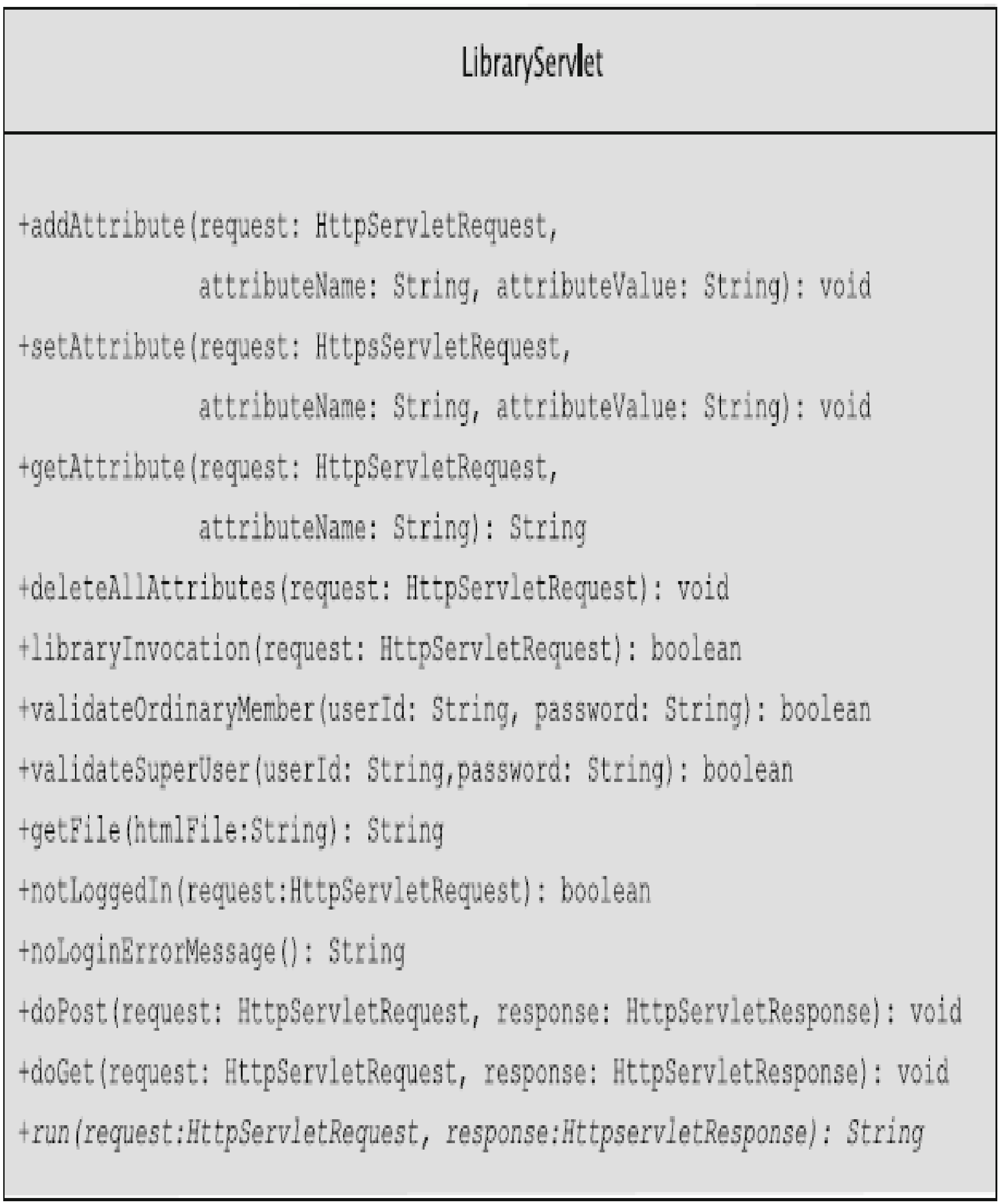
- The server runs with the support of Apache Tomcat, which is a **servlet container**.
- A servlet container is a program that supports servlet execution.
- The servlets themselves are registered with the servlet container. URL requests made by a user are converted to specific servlet requests by the servlet container.
- The servlet container is responsible for initialising the servlets and delivering requests made by the client browser to the appropriate servlet.

Directory structure for the servlets



Structure of servlets in the web-based library system

- A servlet receives data from a browser through a HttpServletRequest object.
- 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.
- This method has two parameters: a request parameter of type HttpServletRequest and a response parameter of type HttpServletResponse.
- These methods and doPost and doGet are collected into a class named LibraryServlet.



LibraryServlet

```
+addAttribute(request: HttpServletRequest,
              attributeName: String, attributeValue: String): void
+setAttribute(request: HttpsServletRequest,
              attributeName: String, attributeValue: String): void
+getAttribute(request: HttpServletRequest,
              attributeName: String): String
+deleteAllAttributes(request: HttpServletRequest): void
+libraryInvocation(request: HttpServletRequest): boolean
+validateOrdinaryMember(userId: String, password: String): boolean
+validateSuperUser(userId: String,password: String): boolean
+getFile(htmlFile:String): String
+notLoggedIn(request:HttpServletRequest): boolean
+noLoginErrorMessage(): String
+doPost(request: HttpServletRequest, response: HttpServletResponse): void
+doGet(request: HttpServletRequest, response: HttpServletResponse): void
+run(request:HttpServletRequest, response:HttpservletResponse): String
```

Most of the methods of LibraryServlet fall into one of five categories:

1. One group contains methods that store information about the user.
2. Methods to validate users and help assess access rights.
3. The getFile method 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.
5. The final group of commands deal with processing the request and responding to it. **Refer page number 414**