Maharaja Education Trust (R), Mysuru

Maharaja Institute of Technology Mysore

Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

Approved by AICTE, New Delhi,
Affiliated to VTU, Belagavi & Recognized by Government of Karnataka

Lecture Notes on

Software Architecture and Design Patterns (15IS72)

Prepared by

Department of Information Science and Engineering

Vision/ ಆಶಯ

"To be recognized as a premier technical and management institution promoting extensive education fostering research, innovation and entrepreneurial attitude"

ಸಂಶೋಧನೆ, ಆವಿಷ್ಕಾರ ಹಾಗೂ ಉದ್ಯಮಶೀಲತೆಯನ್ನು ಉತ್ತೇಜಿಸುವ ಅಗ್ರಮಾನ್ಯ ತಾಂತ್ರಿಕ ಮತ್ತು ಆಡಳಿತ ವಿಜ್ಞಾನ ಶಿಕ್ಷಣ ಕೇಂದ್ರವಾಗಿ ಗುರುತಿಸಿಕೊಳ್ಳುವುದು.

Mission/ ಧ್ಯೇಯ

- To empower students with indispensable knowledge through dedicated teaching and collaborative learning.

  ಸಮರ್ಪಣಾ ಮನೋಭಾವದ ಬೋಧನೆ ಹಾಗೂ ಸಹಭಾಗಿತ್ವದ ಕಲಿಕಾಕ್ರಮಗಳಿಂದ ವಿದ್ಯಾರ್ಥಿಗಳನ್ನು ಅತ್ಯತ್ಕೃಷ್ಟ ಜ್ಞಾನಸಂಪನ್ನರಾಗಿಸುವುದು.

- To advance extensive research in science, engineering and management disciplines.

  ವೈಜ್ಞಾನಿಕ, ತಾಂತ್ರಿಕ ಹಾಗೂ ಆಡಳಿತ ವಿಜ್ಞಾನ ವಿಭಾಗಗಳಲ್ಲಿ ವಿಸ್ತೃತ ಸಂಶೋಧನೆಗಳೊಡನೆ ಬೆಳವಣಿಗೆ ಹೊಂದುವುದು.

- To facilitate entrepreneurial skills through effective institute - industry collaboration and interaction with alumni.

  ಉದ್ಯಮ ಕ್ಷೇತಗಳೊಡನೆ ಸಹಯೋಗ, ಸಂಸ್ಥೆಯ ಹಿರಿಯ ವಿದ್ಯಾರ್ಥಿಗಳೊಂದಿಗೆ ನಿರಂತರ ಸಂವಹನಗಳಿಂದ ವಿದ್ಯಾರ್ಥಿಗಳಿಗೆ ಉದ್ಯಮಶೀಲತೆಯ ಕೌಶಲ್ಯ ಪಡೆಯಲು ನೆರವಾಗುವುದು.

- To instill the need to uphold ethics in every aspect.

  ಜೀವನದಲ್ಲಿ ನೈತಿಕ ಮೌಲ್ಯಗಳನ್ನು ಅಳವಡಿಸಿಕೊಳ್ಳುವುದರ ಮಹತ್ವದ ಕುರಿತು ಅರಿವು ಮೂಡಿಸುವುದು.

- To mould holistic individuals capable of contributing to the advancement of the society.

  ಸಮಾಜದ ಬೆಳವಣಿಗೆಗೆ ಗಣನೀಯ ಕೊಡುಗೆ ನೀಡಬಲ್ಲ ಪರಿಪೂರ್ಣ ವ್ಯಕ್ತಿತ್ವವುಳ್ಳ ಸಮರ್ಥ ನಾಗರೀಕರನ್ನು ರೂಪಿಸುವುದು.

VISION OF THE DEPARTMENT

To be recognized as the best centre for technical education and research in the field of information science and engineering.

MISSION OF THE DEPARTMENT

- To facilitate adequate transformation in students through a proficient teaching learning process with the guidance of mentors and all-inclusive professional activities.
- To infuse students with professional, ethical and leadership attributes through industry collaboration and alumni affiliation.
- To enhance research and entrepreneurship in associated domains and to facilitate real time problem solving.

PROGRAM EDUCATIONAL OBJECTIVES:

- Proficiency in being an IT professional, capable of providing genuine solutions to information science problems.
- Capable of using basic concepts and skills of science and IT disciplines to pursue greater competencies through higher education.
- Exhibit relevant professional skills and learned involvement to match the requirements of technological trends.

PROGRAM SPECIFIC OUTCOME:

Student will be able to

- **PSO1:** Apply the principles of theoretical foundations, data Organizations, and networking concepts and data analytical methods in the evolving technologies.
- **PSO2:**Analyse proficient algorithms to develop software and hardware competence in both professional and industrial areas

Program Outcomes

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Overview

SUBJECT: Software Architecture and Design Patterns **SUBJECT CODE: 15IS72**

The object oriented paradigm emphasizes modularity and re-usability.the goal of an object oriented approach is to satisfy "open closed principle". A module is open if it supports extension. Object oriented modeling (OOM) is a common approach to modeling applications, systems, and business domains by using the object oriented paradigm throughout the entire development life cycles.OOM is a main technique heavily used by both OOD and OOA activities in modern software engineering.

Software architecture is an abstract view of a software system distinct from the details of implementation, algorithms, and data representation .architecture is increasingly ,a crucial part of a software organizations business strategy. The software architecture of a program or computing system is the structure or structures of the system which comprise software components, the externally visible properties of those components, and the relationship among them.

Software architecture course provides a communication among stakeholders, captures early design decisions, acts as a transferable abstraction of a system ,defines constraints on implementation ,dictates organizational structure ,inhibits or enables a system's quality attributes, is analyzable and a vehicle for predicting, system qualities, makes it easier to reason about and manage change ,helps in evolutionary prototyping ,enables more accurate cost and schedule estimates.

Course Objectives

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

Course Outcomes

CO's	DESCRIPTION OF THE OUTCOMES
15IS72.1	Apply the range of design patterns to solve the given problem.
15IS72.2	Apply design principles in the design of object oriented systems and distributed systems.
15IS72.3	Analyze various components of object oriented system and patterns.
15IS72.4	Design the necessary code qualities needed to keep code flexible.
15IS72.5	Assess the quality of design with respect to design principles.

Prof. Smithashree	**Prof. Sneha D.P**	**Course Coordinator**

Facilitator	**NBA Coordinator**	**HOD**

Maharaja Institute of Technology Mysore

Department of Information Science and Engineering

Syllabus

SUBJECT: Software Architecture and Design Patterns **SUBJECT CODE: 15IS72**

Syllabus	Teaching Hours
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	**10 Hours**
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.	**10 Hours**
Module-3	
Design Pattern Catalog: Structural patterns, Adapter, bridge, composite, decorator, facade, flyweight, proxy.	**10 Hours**
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	**10 Hours**
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.	**10 Hours**
List of 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.	
List of Reference Books	
1. Frank Bachmann, ReginеMeunier, Hans Rohnert “Pattern Oriented Software Architecture” –Volume 1, 1996. 2. William J Brown et al., "Anti-Patterns: Refactoring Software, Architectures and Projects in Crisis", John Wiley, 1998.	

Index

MODULE 1

What is a Design Pattern?

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

In general, a pattern has four essential elements:

- The **pattern** name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary. It lets us design at a higher level of abstraction. It makes it easier to think about designs and to communicate them and their trade-offs to others.
  Finding good names has been one of the hardest parts of developing our catalog.

- The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.

- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

- The **consequences** are the results and trade-offs of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object- oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability.

Design patterns key points.

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

- A design pattern names, abstracts, and identifies the key aspects of a common Design structure that make it useful for creating a reusable object-oriented design.

- The design pattern identifies the participating classes and instances, their roles and collaborations, and the distribution of responsibilities.

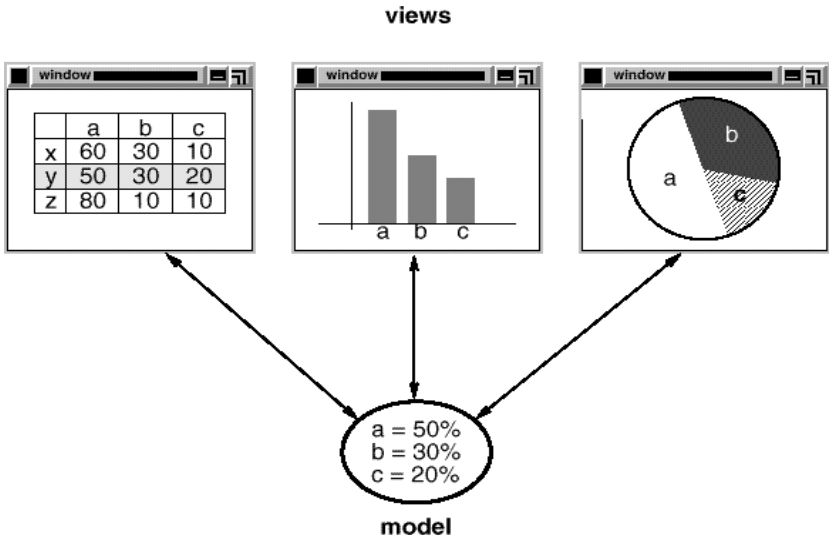
- Each design pattern focuses on a particular object-oriented design problem or issue.

- It describes when it applies, whether it can be applied in view of other design constraints, and the consequences and trade-offs of its use.

- A design pattern also provides sample C++ and (sometimes) Smalltalk code to illustrate an implementation.

Design Patterns in Smalltalk MVC

- The Model/View/Controller (MVC) is used to build user interfaces in Smalltalk-80.
- MVC consists of three kinds of objects.
- The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. MVC decouples them to increase flexibility and reuse.MVC decouples views and models by establishing a subscribe/notify protocol between them.
- A view must ensure that its appearance reflects the state of the model. Whenever the model's data changes, the model notifies views that depend on it. In response, each view gets an opportunity to update itself.
- This approach lets you attach multiple views to a model to provide different presentations. The following diagram shows a model and three views.
- The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways. The model communicates with its views when its values change, and the views communicate with the model to access these values.



- Taken at face value, this example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others.

- Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. The user interface for an object inspector can consist of nested views that may be reused in a debugger.

- MVC supports nested views with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.
- MVC also lets you change the way a view responds to user input without changing its visual presentation.
- For example, or have it use a pop-up menu instead of command keys. MVC encapsulates the response mechanism in a Controller object. There is a class hierarchy of controllers, making it easy to create a new controller as a variation on an existing one.
- A view uses an instance of a Controller subclass to implement a particular response strategy; to implement a different strategy, simply replace the instance with a different kind of controller.
- It's even possible to change a view's controller at run-time to let the view change the way it responds to user input. For example, a view can be disabled so that it doesn't accept input simply by giving it a controller that ignores input events.

Describing Design Patterns

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

1.Pattern Name and Classification
The pattern's name conveys the essence of the pattern succinctly. A good name is vit
al, because it will become part of your design vocabulary.

2. Intent
A short statement that answers the following questions: What does the design pattern do? What is its rationale and intent? What particular design issue or problem does it address?

3.Also Known As
Other well-known names for the pattern, if any.

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

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

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

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

8. Collaborations
How the participants collaborate to carry out their responsibilities.

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

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

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

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

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

The Catalog of Design Patterns

The catalog contains 23 design patterns.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

18. Proxy

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

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

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

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

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

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

Organizing the Catalog

- Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them. We classify design patterns by two criteria (Table 1.1).

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

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

Table 1.1: Design pattern space

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

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

Yet another way to organize design patterns is according to how they reference each other in their "Related Patterns" sections. Figure 1.1 depicts these relationships graphically.

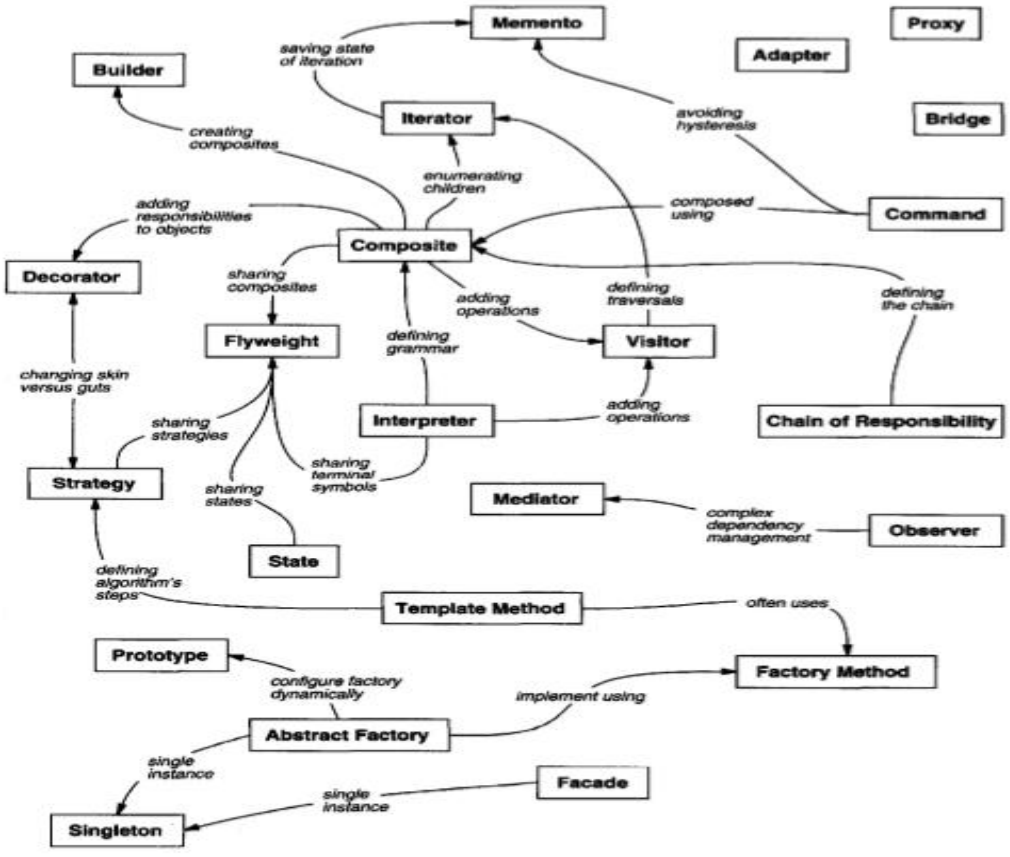


Figure 1.1: Design pattern relationships

How Design Patterns Solve Design Problems

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

Finding Appropriate Objects

- Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called **methods** or operations. An object performs an operation when it receives a request (or **message**) from a **client**. Design patterns help you identify less-obvious abstractions and the objects that can capture them.

- **For example**, objects that represent a process or algorithm don't occur in nature, yet they are a crucial part of flexible designs. The Strategy pattern describes how to implement interchangeable families of algorithms. The State pattern represents each state of an entity as an object. These objects are seldom found during analysis or even the early stages of design; they're discovered later in the course of making a design more flexible and reusable.

Determining Object Granularity

- Objects can vary tremendously in size and number. They can represent everything down to the hardware or all the way up to entire applications. How do we decide what should be an object?

- Design patterns address this issue as well. The Facade pattern describes how to represent complete subsystems as objects, and the Flyweight pattern describes how to support huge numbers of objects at the finest granularities. Abstract Factory and Builder yield objects whose only responsibilities are creating other objects. Visitor and Command yield objects whose only responsibilities are to implement a request on another object or group of objects.

Specifying Object Interfaces

- Every operation declared by an object specifies the operation's name, the objects it takes as parameters, and the operation's return value. This is known as the operation's signature. The set of all signatures defined by an object's operations is called the interface to the object. An object's interface characterizes the complete set of requests that can be sent to the object. Any request that matches a signature in the object's interface may be sent to the object.

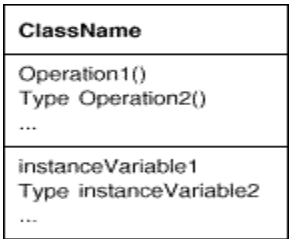
- Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what *not* to put in the interface. The Memento pattern is a good example. It describes how to encapsulate and save the internal state of an object so that the object can be restored to that state later. The pattern stipulates that Memento objects must define two interfaces: a restricted one that lets clients hold and copy

mementos, and a privileged one that only the original object can use to store and retrieve state in the memento.

- Design patterns also specify relationships between interfaces. In particular, they often require some classes to have similar interfaces, or they place constraints on the interfaces of some classes. For example, both Decorator and Proxy require the interfaces of Decorator and Proxy objects to be identical to the decorated and proxied objects. In Visitor, the Visitor interface must reflect all classes of objects that visitors can visit.

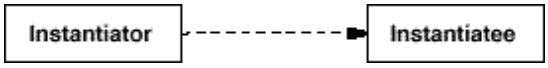
Specifying Object Implementations

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

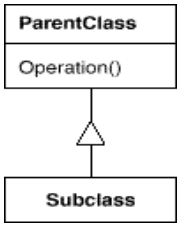


- Return types and instance variable types are optional, since we don't assume a statically typed implementation language.

- Objects are created by **instantiating** a class. The object is said to be an **instance** of the class. The process of instantiating a class allocates storage for the object's internal data (made up of instance variables) and associates the operations with these data. Many similar instances of an object can be created by instantiating a class.
- A dashed arrowhead line indicates a class that instantiates objects of another class. The arrow points to the class of the instantiated objects.

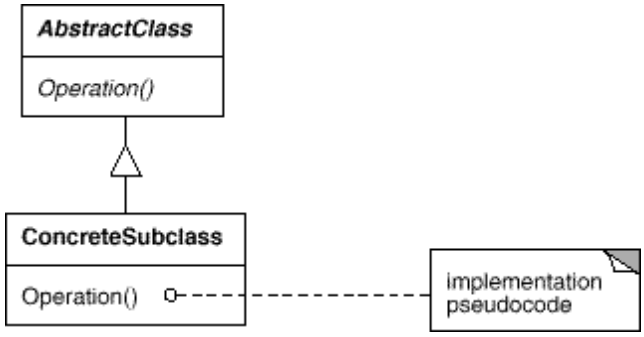


- New classes can be defined in terms of existing classes using class inheritance. When a subclass inherits from a parent class, it includes the definitions of all the data and operations that the parent class defines.

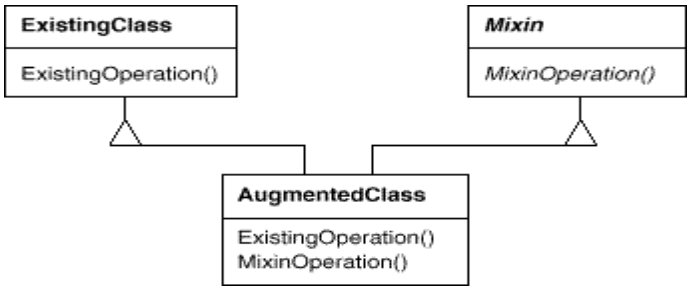


- An abstract class is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated.

- The operations that an abstract class declares but doesn't implement are called abstract operations. Classes that aren't abstract are called concrete classes.



- 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:



Class versus Interface Inheritance

- It's important to understand the difference between an object's class and its type. An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its operations. In contrast, an object's type only refers to its interface—the set of requests to which it can respond. An object can have many types, and objects of different classes can have the same type.

- It's also important to understand the difference between class inheritance and interface inheritance (or subtyping). Class inheritance defines an object's implementation in terms of another object's implementation. In short, 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.

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.

- When inheritance is used carefully (some will say *properly*), all classes derived from an abstract class will share its interface. This implies that a subclass merely adds or overrides operations and does not hide operations of the parent class. *All* subclasses can then respond to the requests in the interface of this abstract class, making them all subtypes of the abstract class.

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

- Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.
- Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class (es) defining the interface.

This so greatly reduces implementation dependencies between subsystems that it leads to the following principle of reusable object-oriented design:

Program to an interface, not an implementation.

Putting Reuse Mechanisms to Work

Inheritance versus Composition

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition.

- Class inheritance lets you define the implementation of one class in terms of another's Reuse by subclassing is often referred to as white-box reuse. The term "**white-box**" refers to visibility: With inheritance, the internals of parent classes are often visible to subclasses.

- Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or *composing* objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called **black-box** reuse, because no internal details of objects are visible. Objects appear only as "black boxes."

Inheritance and composition each have their advantages and disadvantages.

- Class inheritance is defined statically at compile-time and is straightforward to use, since it's supported directly by the programming language. Class inheritance also makes it easier to modify the implementation being reused. When a subclass overrides some but not all operations, it can affect the operations it inherits as well, assuming they call the overridden operations.

Class inheritance disadvantages.

- First, you can't change the implementations inherited from parent classes at run-time, because inheritance is defined at compile-time. Second, and generally worse, parent classes often define at least part of their subclasses' physical representation.

- Because inheritance exposes a subclass to details of its parent's implementation, it's often said that "inheritance breaks encapsulation". The implementation of a subclass becomes so bound up with the implementation of its parent class that any change in the parent's implementation will force the subclass to change.

- Object composition is defined dynamically at run-time through objects acquiring references to other objects. Composition requires objects to respect each others' interfaces, which in turn requires carefully designed interfaces that don't stop you from using one object with many others. But there is a payoff.

- Because objects are accessed solely through their interfaces, we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type. Moreover, because an object's implementation will be written in terms of object interfaces, there are substantially fewer implementation dependencies.

- Object composition has another effect on system design. Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task. Your classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters. On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class.

That leads us to our second principle of object-oriented design:

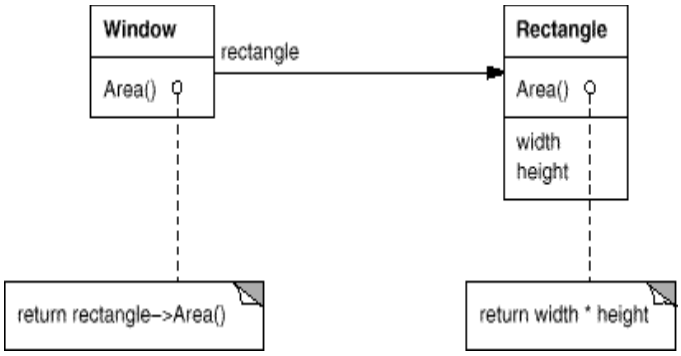
Favor object composition over class inheritance.

Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance. In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**.

- **For example**, instead of making class Window a subclass of Rectangle (because windows happen to be rectangular), the Window class might reuse the behavior of Rectangle by keeping a Rectangle instance variable and *delegating* Rectangle-specific behavior to it. In other words, instead of a Window *being* a Rectangle, it would *have* a Rectangle. Window must now forward requests to its Rectangle instance explicitly, whereas before it would have inherited those operations.

The following diagram depicts the Window class delegating its Area operation to a Rectangle instance.



- A plain arrowhead line indicates that a class keeps a reference to an instance of another class. The reference has an optional name, "rectangle" in this case.

- The main advantage of delegation is that it makes it easy to compose behaviors at run- time and to change the way they're composed.

- 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.

- Several design patterns use delegation. In the State pattern, an object delegates requests to a State object that represents its current state. In the Strategy pattern, an object delegates a specific request to an object that represents a strategy for carrying out the request.

Inheritance versus Parameterized Types

- Another (not strictly object-oriented) technique for reusing functionality is through parameterized types, also known as **generics** (Ada, Eiffel) and **templates** (C++).

- This technique lets you define a type without specifying all the other types it uses. The unspecified types are supplied as *parameters* at the point of use. For example, a List class can be parameterized by the type of elements it contains.

- Parameterized types give us a third way (in addition to class inheritance and object composition) to compose behavior in object-oriented systems. Many designs can be implemented using any of these three techniques. To parameterize a sorting routine by the operation it uses to compare elements, we could make the comparison

1. An operation implemented by subclasses
2. The responsibility of an object that's passed to the sorting routine
3. An argument of a C++ template or Ada generic that specifies the name of the function to call to compare the elements.

- There are important differences between these techniques. Object composition lets you change the behavior being composed at run-time, but it also requires indirection and can be less efficient. Inheritance lets you provide default implementations for operations and lets subclasses override them. Parameterized types let you change the types that a class can use. But neither inheritance nor parameterized types can change at run-time.

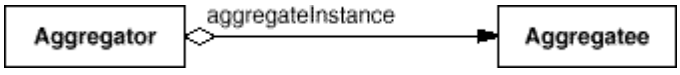
Relating Run-Time and Compile-Time Structures

- An object-oriented program's run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's run-time structure consists of rapidly changing networks of communicating objects.

- In fact, the two structures are largely independent. Consider the distinction between object aggregation and acquaintance and how differently they manifest themselves at compile- and run-times. Aggregation implies that one object owns or is responsible for another object. Generally we speak of 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.

In our diagrams, a plain arrowhead line denotes acquaintance. An arrowhead line with a diamond at its base denotes aggregation:



- Acquaintance and aggregation are determined more by intent than by explicit language mechanisms. The distinction may be hard to see in the compile-time structure, but it's significant. Aggregation relationships tend to be fewer and more permanent than acquaintance. Acquaintances, in contrast, are made and remade more frequently, sometimes existing only for the duration of an operation.

- Acquaintances are more dynamic as well, making them more difficult to discern in the source code. With such disparity between a program's run-time and compile-time structures, it's clear that code won't reveal everything about how a system will work. The system's run-time structure must be imposed more by the designer than the language. The relationships between objects and their types must be designed with great care, because they determine how good or bad the run-time structure is.

- Many design patterns capture the distinction between compile-time and run-time structures explicitly. Composite and Decorator are especially useful for building complex run-time structures. Observer involves run-time structures that are often hard to understand unless you know the pattern.

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.

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

1. ***Creating an object by specifying a class explicitly***. Specifying a class name when you create an object commits you to a particular implementation instead of a particular interface. This commitment can complicate future changes. To avoid it, create objects indirectly. **Design patterns**: Abstract Factory, Factory Method, Prototype.

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

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

4. ***Dependence on object representations or implementations***. Clients that know how an object is represented, stored, located, or implemented might need to be changed when the object changes. Hiding this information from clients keeps changes from cascading.
Design patterns: Abstract Factory, Bridge, Memento, Proxy.

5. ***Algorithmic dependencies***. Algorithms are often extended, optimized, and replaced during development and reuse. Objects that depend on an algorithm will have to change when the algorithm changes. Therefore algorithms that are likely to change should be isolated.
Design patterns: Builder, Iterator, Strategy, Template Method, Visitor.

6. ***Tight coupling***. Classes that are tightly coupled are hard to reuse in isolation, since they depend on each other. Tight coupling leads to monolithic systems, where you can't change or remove a class without understanding and changing many other classes. The system becomes a dense mass that's hard to learn, port, and maintain. Loose coupling increases the probability that a class can be reused by itself and that a system can be learned, ported, modified, and extended more easily. Design patterns use techniques such as abstract coupling and layering to promote loosely coupled systems.
Design patterns: Abstract Factory , Bridge, Chain of Responsibility, Command , Facade, Mediator, Observer.

7. ***Extending functionality by subclassing***. Customizing an object by subclassing often isn't easy. Every new class has a fixed implementation overhead (initialization, finalization, etc.). Defining a subclass also requires an in-depth understanding of the parent class. For example, overriding one operation might require overriding another. An overridden operation might be required to call an inherited operation. And subclassing
can lead to an explosion of classes, because you might have to introduce many new subclasses for even a simple extension.
Object composition in general and delegation in particular provide flexible alternatives to inheritance for combining behavior. New functionality can be added to an application by composing existing objects in new ways rather than by defining new subclasses of existing classes. On the other hand, heavy use of object composition can make designs harder to understand. Many design patterns produce designs in which you can introduce customized functionality just by defining one subclass and composing its instances
with existing ones.
Design patterns: Bridge,Chain of Responsibility ,Composite,Decorator,Observer, Strategy.

8. ***Inability to alter classes conveniently***. Sometimes you have to modify a class that can't be modified conveniently. Perhaps you need the source code and don't have it (as may be the case with a commercial class library). Or maybe any change would require modifying lots of existing subclasses. Design patterns offer ways to modify classes in such circumstances.
Design patterns: Adapter, Decorator, Visitor.

Application Programs

- Design patterns also make an application more maintainable when they're used to limit platform dependencies and to layer a system. They enhance extensibility by showing you how to extend class hierarchies and how to exploit object composition.

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. An example of a toolkit is a set of collection classes for lists, associative tables, stacks, and the like.
- The C++ I/O stream library is another example. Toolkits don't impose a particular design on your application; they just provide functionality that can help your application do its job. They let you as an implementer avoid recoding common functionality. Toolkits emphasize *code reuse*. They are the object-oriented equivalent of subroutine libraries.

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. A framework predefines these design parameters so that you, the application designer/implementer, can concentrate on the specifics of your application. The framework captures the design decisions that are common to its application domain.

Patterns and frameworks have some similarities. They are different in three major ways:

1. ***Design patterns are more abstract than frameworks***. Frameworks can be embodied in code, but only *examples* of patterns can be embodied in code. A strength of frameworks is that they can be written down in programming languages and not only studied but executed and reused directly. In contrast, the design patterns in this book have to be implemented each time they're used. Design patterns also explain the intent, trade-offs, and consequences of a design

2. ***Design patterns are smaller architectural elements than frameworks***. A typical framework contains several design patterns, but the reverse is never true.

3. ***Design patterns are less specialized than frameworks***. Frameworks always have a particular application domain. A graphical editor framework might be used in a factory simulation, but it won't be mistaken for a simulation framework. In contrast, the design patterns in this catalog can be used in nearly any kind of application. While more specialized design patterns than ours are certainly possible (say, design patterns for distributed systems or concurrent programming), even these wouldn't dictate an application architecture like a framework would.

How to Select a Design Pattern

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

1. **Consider how design patterns solve design problems.** discusses how design patterns help you find appropriate objects, determine object granularity, specify object interfaces, and several other ways in which design patterns solve design problems. Referring to these discussions can help guide your search for the right pattern.
2. **Scan Intent sections** lists the Intent sections from all the patterns in the catalog. Read through each pattern's intent to find one or more that sound relevant to your problem.
3. **Study how patterns interrelate**. Shows relationships between design patterns graphically. Studying these relationships can help direct you to the right pattern or group of patterns.
4. **Study patterns of like purpose**. The catalog has three chapters, one for creational patterns, and another for patterns, and a third for behavioral patterns. Each chapter starts off with introductory comments on the patterns and concludes with a section that compares and contrasts them. These sections give you insight into the similarities and differences between patterns of like purpose.
5. **Examine a cause of redesign**. Look at the causes of redesign starting to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign.

6. **Consider what should be variable in your design**. This approach is the opposite of focusing on the causes of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies a theme of many design patterns. Table 1.2 lists the design aspect(s) that design patterns let you vary independently; thereby letting you change them without redesign.

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

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

1. ***Read the pattern once through for an overview***. Pay particular attention to the Applicability and Consequences sections to ensure the pattern is right for your problem.
2. ***Go back and study the Structure, Participants, and Collaborations sections***. Make sure you understand the classes and objects in the pattern and how they relate to one another.
3. *Look at the Sample Code section to see a concrete example of the pattern in code.* Studying the code helps you learn how to implement the pattern.
4. ***Choose names for pattern participants that are meaningful in the application context***. The names for participants in design patterns are usually too abstract to appear directly in an application. Nevertheless, it's useful to incorporate the participant name into the name that appears in the application. That helps make the pattern more explicit in the implementation. **For example**, if you use the Strategy pattern for a text compositing algorithm, then you might have classes SimpleLayoutStrategy or TeXLayoutStrategy.
5. ***Define the classes***. Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.
6. ***Define application-specific names for operations in the pattern***. Here again, the names generally depend on the application. Use the responsibilities and collaborations associated with each operation as a guide. Also, be consistent in your naming conventions.
For example, you might use the "Create-" prefix consistently to denote a factory method.
7. ***Implement the operations to carry out the responsibilities and collaborations in the pattern.*** The Implementation section offers hints to guide you in the implementation. The examples in the Sample Code section can help as well.

What Is Object-Oriented Development?

- To define software system as a collection of objects of various types that interacts with each other through well-defined interface. A software object can be designed to handle multiple functions and can therefore participate in several processes. A software component is also capable of storing data, which adds another dimension of complexity to the process.
- Each component represents a data abstraction and is designed to store information along with procedures to manipulate the same. The execution of the original processes is then broken down into several steps, each of which can be logically assigned to one of the software components. The components can also communicate with each other as needed to complete the process.

- Its language to specify the output from each step of the process so that we can transition smoothly from one stage to the next, the ability to reuse earlier designs, standard solutions that adhere to well-reasoned design principles and, even the ability to incrementally fix a poor design without breaking the system.

Key Concepts of Object-Oriented Design

The Central Role of Objects

- Object-orientation, as the name implies, makes objects the centrepiece of software design. The design of earlier systems was centred around processes, which were susceptible to change, and when this change came about, very little of the old system was 're-usable'. The notion of an object is centred around a piece of data and the operations (or **methods**) that could be used to modify it.
- This makes possible the creation of an abstraction that is very stable since it is not dependent on the changing requirements of the application. The execution of each process relies heavily on the objects to store the data and provide the necessary operations; with some additional work, the entire system is 'assembled' from the objects.

The Notion of a Class

- Classes allow a software designer to look at objects as different types of entities. Viewing objects this way allows us to use the mechanisms of classification to categorise these types, define hierarchies and engage with the ideas of specialisation and generalisation of objects.

Abstract Specification of Functionality

- In the course of the design process, the software engineer specifies the properties of objects (and by implication the classes) that are needed by a system. This specification is abstract in that it does not place any restrictions on howthe functionality is achieved. This specification, called an **interface** or an **abstract class**, is like a *contract* for the implementer which also facilitates formal verification of the entire system.

A Language to Define the System

- The Unified Modelling Language (UML) has been chosen by consensus as the standard tool for describing the end products of the design activities. The documents generated in this language can be universally understood and are thus analogous to the 'blueprints' used in other engineering disciplines.

Standard Solutions

- The existence of an object structure facilitates the documenting of standard solutions, called **design patterns.** Standard solutions are found at all stages of software development, but design patterns are perhaps the most common form of reuse of solutions.

An Analysis Process to Model a System

- Object-orientation provides us with a systematic way to translate a functional specification to a *conceptual design.* This design describes the system in terms of *conceptual classes* from which the subsequent steps of the development process generate the *implementation classes* that constitute the finished software.

The Notions of Extendibility and Adaptability

- Software has a flexibility that is not typically found in hardware, and this allows us to modify existing entities in small ways to create new ones. *Inheritance*, which creates a new *descendant class* that modifies the features of an existing (**ancestor**) class, and **composition**, which uses objects belonging to existing classes as elements to constitute a new class, are mechanisms that enable such modifications with classes and objects.

Other Related Concepts

Modular Design and Encapsulation

- *Modularity* refers to the idea of putting together a large system by developing a number of distinct components independently and then integrating these to provide the required functionality. This approach, when used properly, usually makes the individual modules relatively simple and thus the system easier to understand than one that is designed as a monolithic structure.
- In other words, such a design must be *modular.* The system's functionality must be provided by a number of well-designed, cooperating modules. Each module must obviously provide certain functionality that is clearly specified by an interface. The interface also defines how other components may interact or communicate with the module. We would like that a module clearly specify what it does, but not expose its implementation.
- This separation of concerns gives rise to the notion of **encapsulation**, which means that the module hides details of its implementation from external agents. The **abstract data type (ADT)**, the generalization of primitive data types such as integers and characters, is an example of applying encapsulation. The programmer specifies the collection of operations on the data type and the data structures that are needed for data storage. Users of the ADT perform the operations without concerning themselves with the implementation.

Cohesion and Coupling

- Each module provides certain functionality; **cohesion** of a module tells us how well the entities within a module work together to provide this functionality. Cohesion is a measure of how focused the responsibilities of a module are. If the responsibilities of a module are unrelated or varied and use different sets of data, cohesion is reduced. Highly cohesive modules tend to be more reliable, reusable, and understandable than less cohesive ones.
- **Coupling** refers to how dependent modules are on each other. The very fact that we split a program into multiple modules introduces some coupling into the system. Coupling could result because of several factors: a module may refer to variables defined in another module or a module may call methods of another module and use the return values. The amount of coupling between modules can vary.
- In general, if modules do not depend on each other's implementation, i.e., modules depend only on the published interfaces of other modules and not on their internals; we say that the coupling is *low.* In such cases, changes in one module will not necessitate changes in other modules as long as the interfaces themselves do not

change. Low coupling allows us to modify a module without worrying about the ramifications of the changes on the rest of the system.

- By contrast, *high* coupling means that changes in one module would necessitate changes in other modules, which may have a domino effect and also make it harder to understand the code.

Modifiability and Testability

- A software component, unlike its hardware counterpart, can be easily modified in small ways. This modification can be done to change both *functionality* and *design*.
- The ability to change the functionality of a component allows for systems to be more **adaptable**; the advances in object-orientation have set higher standards for adaptability. Improving the design through incremental change is accomplished by *refactoring*, again a concept that owes its origin to the development of the object oriented approach

- **Testability** of a concept, in general, refers to both *falsifiability*, i.e., the ease with which we can find counterexamples, and the *practical feasibility* of reproducing such counterexamples.

- In the context of software systems, it can simply be stated as the ease with which we can find bugs in software and the extent to which the structure of the system facilitates the detection of bugs.

Benefits and Drawbacks of the Paradigm

The advantages listed below are largely consequences of the ideas presented in the previous sections.

1. Objects often reflect entities in application systems. This makes it easier for a designer to come up with classes in the design. In a process-oriented design, it is much harder to find such a connection that can simplify the initial design.

2. Object-orientation helps increase productivity through reuse of existing software. Inheritance makes it relatively easy to extend and modify functionality provided by a class. Language designers often supply extensive libraries that users can extend.

3. It is easier to accommodate changes. One of the difficulties with application development is changing requirements. With some care taken during design, it is possible to isolate the varying parts of a system into classes.

4. The ability to isolate changes, encapsulate data, and employ modularity reduces the risks involved in system development.

Drawbacks

1. The object-oriented development process introduces many layers of software, and this certainly increases overheads.

2. Object creation and destruction is expensive.

3. Objects tend to have complex associations, which can result in *non-locality*, leading to poor memory access times.

4. Programmers and designers schooled in other paradigms, usually in the imperative paradigm, find it difficult to learn and use object-oriented principles.

Module 2

Analysis a System

Overview of the Analysis Phase

The major goal of this phase is to address this basic question: what should the system do?

- Consider computer science student example: Typically, the program requirements are written up by the instructor: the student does some design, writes the code, and submits the program for grading. To some extent, the process of understanding the requirements, doing the design, and implementing that design is relatively informal. Requirements are often simple and any clarifications can be had via questions in the classroom, e-mail messages, etc.
- The above simple-minded approach does not quite suffice for 'real-life' projects for a number of reasons. For one reason, such systems are typically much bigger in scope and size. They also have complex and ambiguously-expressed requirements. Third, there is usually a large amount of money involved, which makes matters quite serious. For a fourth reason, hard as it may be for a student to appreciate it, project deadlines for these 'real-life' projects are more critical. Hence,

The process could be split into three activities:

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

Stage 1: Gathering the Requirements

- The purpose of *requirements analysis* is to define what the new system should do. The importance of doing this correctly cannot be overemphasized. Since the system will be built based on the information garnered in this step, any errors made in this stage will result in the implementation of a wrong system. Once the system is implemented, it is expensive to modify it to overcome the mistakes introduced in the analysis stage.

- Requirements for a new system are determined by a team of analysts by interacting with teams from the company paying for the development (clients) and the user community, who ultimately uses the system on a day-to-day basis. This interaction can be in the form of interviews, surveys, observations, study of existing manuals, etc. Broadly speaking, the requirements can be classified into two categories:

- ***Functional requirements*** These describe the interaction between the system and its users, and between the system and any other systems, which may interact with the system by supplying or receiving data.

- ***Non-functional requirements*** Any requirement that does not fall in the above category is a non-functional requirement. Such requirements include response time, usability and accuracy. Sometimes, there may be considerations that place

restrictions on system development; these may include the use of specific hardware and software and budget and time constraints

Case Study Introduction

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

The business processes of the library system are listed below.

Register new members The library receives applications from people who want to become library members, whom we alternatively refer to as **users**. While applying for membership, a person supplies his/her name, phone number and address to the library. The library assigns each member a unique identifier (ID), which is needed
for transactions such as issuing books.

Add books to the collection We will make the assumption that the collection includes just books. For each book the library stores the title, the author's name, and a unique ID. When it is added to the collection, a book is given a unique identifier by the clerk. This ID is based on some standard system of classification.

Issue a book to a member (or user) To check out books, a user (or member) must identify himself to a clerk and hand over the books. The library remembers that the books have been checked out to the member. Any number of books may be checked out in a single transaction.

Record the return of a book To return a book, the member gives the book to a clerk, who submits the information to the system, which marks the book as 'not checked out'. If there is a hold on the book, the system should remind the clerk to set the book aside so that the hold can be processed.

Remove books from the collection From time to time, the library may remove books from its collection. This could be because the books are worn-out, are no longer of interest to the users, or other sundry reasons.

Print out a user's transactions Print out the interactions (book checkouts, returns, etc.) between a specific user and the library on a certain date.

Place/remove a hold on a book When a user wants to put a hold, he/she supplies the clerk with the book's ID, the user's ID, and the number of days after which the book is not needed. The clerk then adds the user to a list of users who wish to borrow the book. If the book is not checked out, a hold cannot be placed. To remove a hold, the user provides the book's ID and the user's ID.

Renew books issued to a member Customers may walk in and request that several of the books they have checked out be renewed (re-issued). The system must display the relevant books, allow the user to make a selection, and inform the user of the result.

Notify member of book's availability Customers who had placed a hold on a book are notified when the book is returned. This process is done once at the end of each day. The clerk enters the ID for each book that was set aside, and the system returns the name and phone number of the user who is next in line to get the book.

In addition, the system must support three other requirements that are not directly related to the workings of a library, but, nonetheless, are essential.
• A command to save the data on a long-term basis.
• A command to load data from a long-term storage device.
• A command to quit the application. At this time, the system must ask the user if data is to be saved before termination.

Functional Requirements Specification

It is important that the requirements be precisely documented. The requirements specification document serves as a contract between the users and the developers.

Use Case Analysis

- 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. It is essentially a narrative describing the sequence of events (actions) of an external agent (actor) using the system to complete a process. It is a powerful technique that describes the kind of functionality that a user expects from the system. Use cases have two or more parties: *agents* who interact with the system and the *system* itself.
- In our simple library system, the members do not use the system directly. Instead, they get services via the library staff. We assume that some kind of a user-interface is required, so that when the system is started, it provides a menu with the following choices:

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

- The actors in our system are members of the library staff who manage the daily operations. This idea is depicted in the use case diagram in Fig. 6.1, which gives an overview of the system's usage requirements.

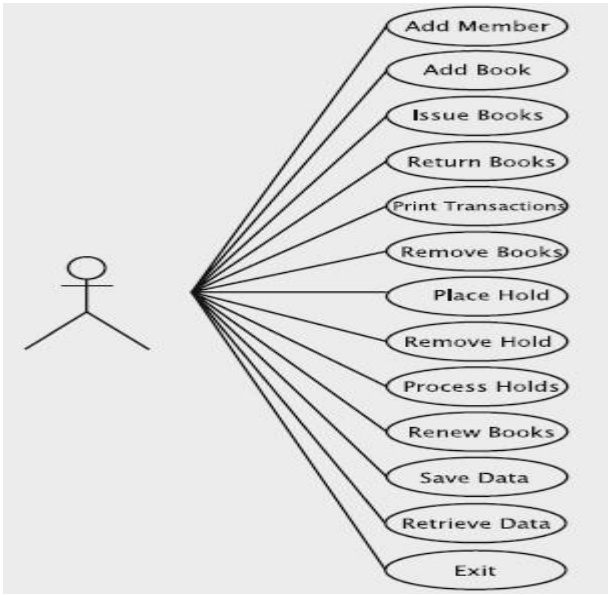


Fig. 6.1 Use case diagram for the library system

Use case for registering a user

Our first use case is for registering a new user and is given in Table 6.1.

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	

- Use cases are specified in a two-column format, where the left-column states the actions of the actor and the right-column shows what the system does.

The above example illustrates several aspects of use cases.

1. Every use case has to be identified by a name. We have given the name Register New Member to this use case.
2. It should represent a reasonably-sized activity in the organization. It is important to note that not all actions and operations should be identified as use cases.
3.The first step of the use case specifies a 'real-world' action that triggers the exchange described in the use case
4.The use case does not specify how the functionality is to be implemented. For example, the details of how the clerk enters the required information into the system are left unspecified.
5.The use case is not expected to cover all possible situations. While we would expect that the sequence of events that are specified in the above use case is what would actually happen in a library when a person wants to be registered, the use case does not specify what the system should do if there are errors.
In other words, the use case explains only the most commonly-occurring scenario, which is referred to as the *main flow*. Deviations from the main flow due to occurrences of errors and exceptions are not detailed in the above use case.

Use case for adding books the use case for adding new books in Table 6.2. Notice that we add more than one book in this use case, which involves a repetitive process captured by a *go-to* statement in the last step.

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

Use case for issuing books Consider the use case where a member comes to the check-out counter to issue a book. The user identifies himself/herself to a clerk, who checks out the books for the user. It proceeds as in Table 6.3.

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	

- There are some drawbacks to the way this use case is written. One is that it does not specify how due-dates are computed. We may have a simple rule (example: due-dates are one month from the date of issue) or something quite complicated (example: due- date is dependent on the member's history, how many books have been checked out, etc.).

- Putting all these details in the use case would make the use case quite messy and harder to understand. Rules such as these are better expressed as **Business Rules**. A business rule may be applicable to one or more use cases. The business rule for due- date generation is simple in our case. It is Rule 1 given in Table 6.4 along with all other rules for the system.

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

A second problem with the use case is that as written above, it does not state what to do in case things go wrong. For instance,

1. The person may not be a member at all. How should the use case handle this situation? We could abandon the whole show or ask the person to register.

2. The clerk may have entered an invalid book id. To take care of these additional situations, we modify the use case as given in Table 6.5.

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	

Use case for returning books Users return books by leaving them on a library clerk's desk; the clerk enters the book ids one by one to return them. Table 6.6 gives the details of the use case.

Table 6.6 Use case Return Book

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

Use cases for removing (deleting) books, printing member transactions, placing a hold, and removing a hold The next four use cases deal with the scenarios for removing books (Table 6.7), printing out member transactions (Table 6.8), placing a hold (Table 6.9), and removing a hold (Table 6.10).

Table 6.7 Use case Removing Books

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

Table 6.8 Use case Member Transactions

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

Table 6.9 Use case Place a Hold

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

Table 6.10 Use case Remove a Hold

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

Use case for processing holds Given in Table 6.11, this use case deals with processing the holds at the end of each day.

Table 6.11 Use case Process Holds

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

Use case for renewing books This use case (see Table 6.12) deals with situations where a user has several books checked out and would like to renew some of these

Table 6.12 Use case Renew Books

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

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. For example, in the library system, some of the major conceptual classes include members and books. Members borrow books, which establish a relationship between them.

- **Design facilitation** the design stage must determine how to implement the functionality. For this, the designers should be in a position to determine the classes that need to be defined, the objects to be created, and how the objects interact. This is better facilitated if the analysis phase classifies the entities in the application and determines their relationships.

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

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

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

In this case study, however, we use a simple approach: we examine the use cases and pick out all the nouns in the description of the requirements.

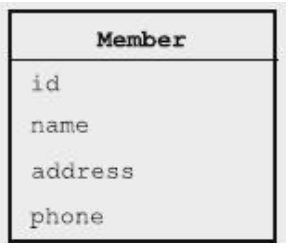
1. **customer**: becomes a member, so it is effectively a synonym for member.
2. **user**: the library refers to members alternatively as users, so this is also a synonym.
3. **application form** and **request**: application form is an external construct for gathering information, and request is just 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 noun **system** implies a conceptual class that represents all of the software; we call this class Library. We note its existence and represent it in UML without any attributes and methods (Fig. 6.2).

Fig. 6.2 UML diagram for the class Library

- A member is described by the attributes name, address, and phone number. Moreover, the system generates an identifier for each user, so that also serves as an attribute. 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. The UML diagram is shown in Fig. 6.3.

Fig. 6.3 UML diagram for the class Member



- The use case Register New Member (Table 6.1) says that the system 'remembers information about the member'. This implies an association between the conceptual classes Library and Member. This idea is shown in Fig. 6.4; note the line between the two classes and the labels 1, *, and 'maintains a collection of' just above it. They mean that one instance of the Library maintains a collection of zero or more members.

g. 6.4 UML diagram showing the association of Library and Member

- Just as we reasoned for the existence of a conceptual class named Member, we can argue for the need of a conceptual class called Book to represent a book. It has attributes id, title, and author. A UML description of the class is shown in Fig. 6.5.

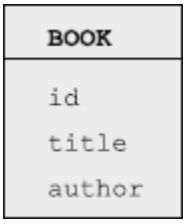


Fig. 6.5 UML diagram for the class Book

- It should come as no surprise that an association between the classes Library and Book, shown in Fig. 6.6, is also needed.

Fig. 6.6 UML diagram showing the association of Library and Book

- 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. As an example of a dynamic association, consider members borrowing books. This is an association between Member and Book, shown in Fig. 6.7

Fig. 6.7 UML diagram showing the association Borrows between Member and Book

In the diagram of Fig. 6.7, we state that there is no limit. It also states that two users may not borrow the same book at the same time.

- Another action that a member can undertake is to place a hold on a book. Several users can have holds placed on a book, and a user may place holds on an arbitrary number of books. In other words, this relationship is many-to-many between users and books. We represent this in Fig. 6.8 by putting a * at both ends of the line representing the association.

Fig. 6.8 UML diagram showing the association holds between Member and Book

- We capture all of the conceptual classes and their associations into a single diagram in Fig. 6.9. To reduce complexity, we have omitted the attributes of Library, Member, and Book. It is important to note that the above conceptual classes or their representation do not, in any way, tell us how the information is going to be stored or accessed. Those decisions will be deferred to the design and implementation phase.

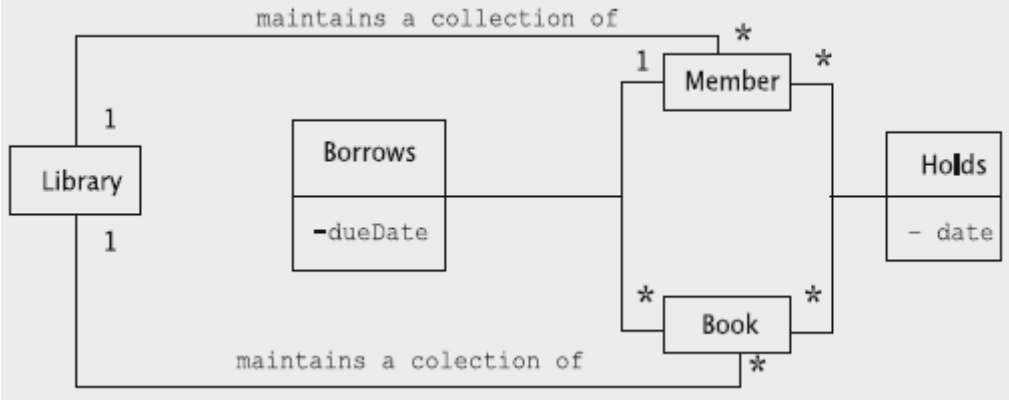


Fig. 6.9 Conceptual classes and their associations

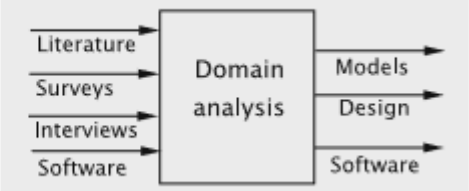
Using the Knowledge of the Domain

- 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. In other words, we identify and analyse common requirements from a specific application domain. In contrast to looking at a certain problem completely from scratch, we apply the knowledge we already have from our study of similar systems to speed up the creation of specifications, design, and code. Thus, one of the goals of this approach is reuse.
- Any area in which we develop software systems qualifies to be a **domain**. Examples include library systems, hotel reservation systems, university registration systems, etc before we analyze and construct a specific system, we first need to perform an exhaustive analysis of the class of applications in that domain. In the domain of libraries, for example, there are things we need to know including 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.

- 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. As shown in Fig. 6.10, a domain analyst analyses this knowledge to come up with specifications, designs, and code that can be reused in multiple projects.

Fig. 6.10 Domain analysis



Design and Implementation

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

1. On what platform(s) (hardware and software) will the system run? For example, will the system be developed for just one platform, say, Windows running on 386-type processors? Or will we be developing for other platforms such as Unix?
2. What languages and programming paradigms will be used for implementation?
3. What user interfaces will the system provide? These include GUI screens, printouts, and other devices (for example, library cards).
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?

Major Subsystems

The first step in our design process is to identify the major subsystems. We can view the library system as composed of two major subsystems:
1. **Business logic** This part deals with input data processing, data creation, queries, and data updates. This module will also be responsible for interacting with external storage, storing and retrieving data.
2. **User interface** This subsystem interacts with the user, accepting and outputting information. It is important to design the system such that the above parts are separated from each other so that they can be varied independently.

Creating the Software Classes

- The next step is to create the **software classes**. During the analysis, after defining the use case model, we came up with a set of conceptual classes and a conceptual class diagram for the entire system. 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.

- The classes for the business logic module will be the ones instrumental in implementing the system requirements described in the use case model. In our analysis, we came up with a set of *conceptual* classes and relationships.
- **Member and Book** These are central concepts. Each Member object comprises several attributes such as name and address, stays in the system for a long period of time and performs a number of useful functions. Books stay part of the library over a long time and we can do a number of useful actions on them. We need to instantiate books and members quite often. Clearly, both are classes that require representation in software.
- **Library** It keeps track of books and members. When a member thinks of a library, he/she thinks of borrowing and returning books, placing and removing holds, i.e., the *functionality* provided by the library.
- **Borrows** This class represents the one-to-many relationship between members and books. *In typical one-to-many relationships, the association class can be efficiently implemented as a part of the two classes at the two ends.* To verify this for our situation, for every pair of member *m* and book *b* such that *m* has borrowed *b*, the corresponding objects simply need to maintain a reference to each other.
- **Holds** Unlike Borrows, this class denotes a many-to-many relationship between the Member and Book classes. *In typical many-to-many relationships, implementation of the association without using an additional class is unlikely to be clean and efficient.* To attempt to do this without an additional class in the case of holds, we would need to maintain within each Member object references to all Book instances for which there is a hold, and keep 'reverse' references from the Book objects to the Member objects.

Assigning Responsibilities to the Classes

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

Register Member

- The sequence diagram for the use case for registering a member is shown in Fig. 7.1. The clerk issues a request to the system to add a newmember. The system responds by asking for the data about the newmember. This interaction occurs between the library staff member and the UserInterface instance. The clerk enters the requested data, which the UserInterface accepts.

- All that UserInterface needs to do is pass the three pieces of information—name, address, and phone number of the applicant—as parameters to the addMember method, which then assumes full responsibility for creating and adding the new member.

Let us see details of the addMember method. The 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:

Option 1 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.

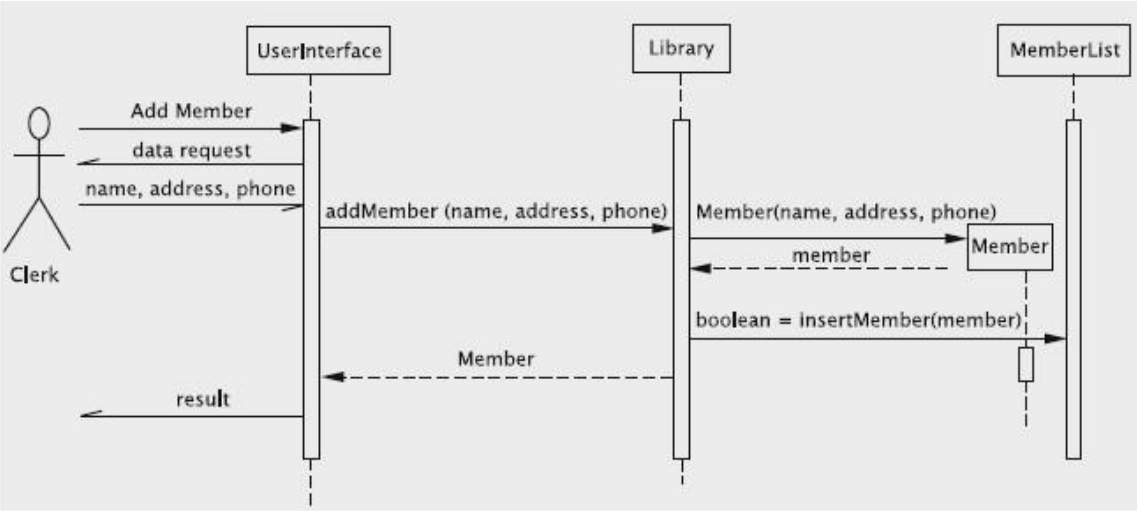


Fig. 7.1 Sequence diagram for adding a new member

- **Option 2** 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. The last step is to return the result so that UserInterface can adequately inform the actor about the success of the operation.

Add Books

- The next sequence diagram that we show is for the Add Books use case. 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. Since the loop involves interacting repeatedly with the actor, the loop control mechanism is in the UI itself. 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.

- The UI returns the result and continues until the actor indicates an exit. This repetition is shown diagrammatically by a special rectangle that is marked loop. All activities within the rectangle are repeated until the clerk indicates that there are no more books to be entered (Fig. 7.2).

Issue Books

The sequence diagram for the Issue Books use case is given next (Fig. 7.3). 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 method search Membership on the Library.

Two options suggest themselves for implementing the search:

• **Option 1** Get an enumeration of all Member objects from MemberList, get the ID from each and compare with the target ID.
• **Option 2** Delegate the entire responsibility to MemberList.

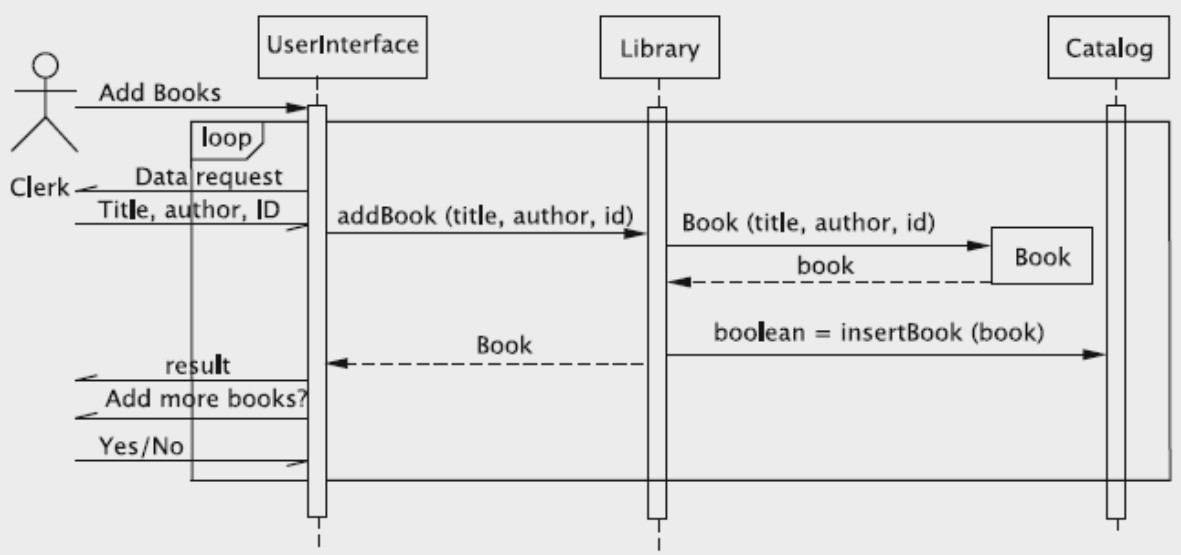


Fig. 7.3 Sequence diagram for issuing books

Return Books

The Return Book use case is implemented in Fig. 7.4 as a sequence diagram. 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.

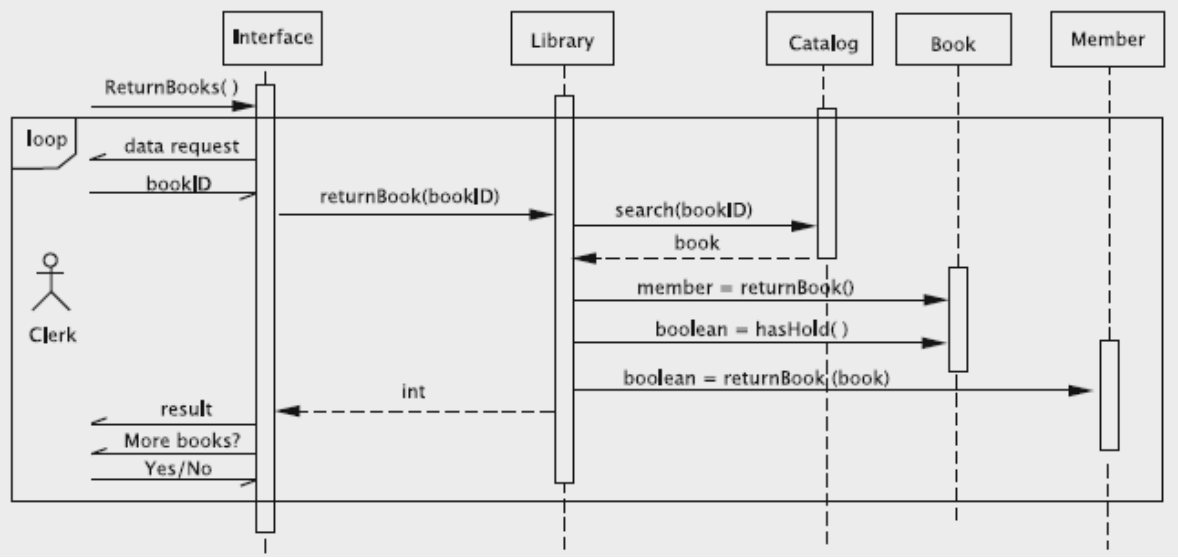


Fig. 7.4 Sequence diagram for returning books

- 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 (Step 5 in Table 6.5):

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.*

- The value returned by returnBook must enable UserInterface to make the distinction between these. This is done by having Library return a result code, which could simply be one of three suitably named integer constants.

Remove Books

The diagram in Fig. 7.5 shows the sequence diagram for removing books from the collection. Here, as discussed in the use case, we remove only those books that are not checked out and do not have a hold.

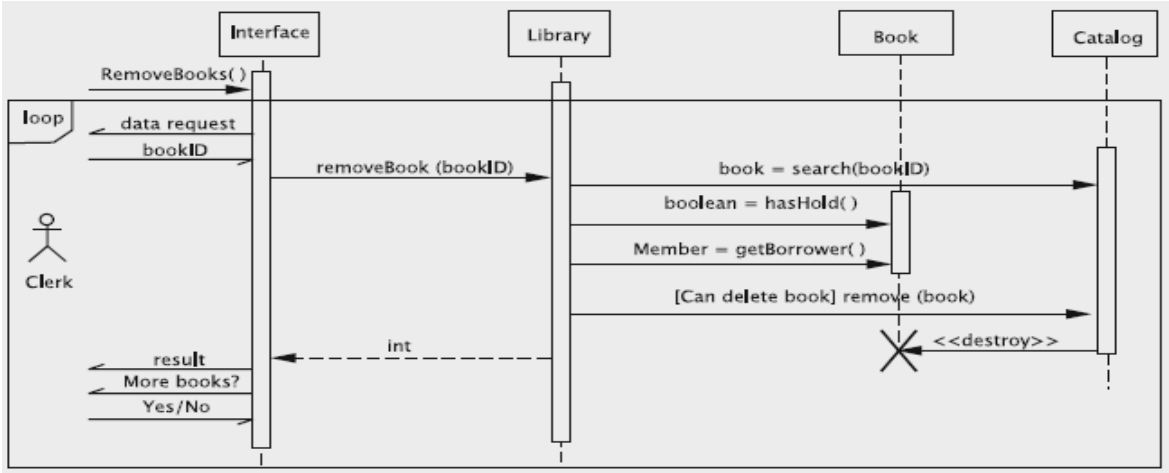


Fig. 7.5 Sequence diagram for removing books

Member Transactions

Following the earlier examples, it is no surprise that the end-user (clerk) interacts with the Library class to print out the transactions of a given member. From the descriptions given so far, the reader should have gained enough skill to interpret most of the sequence diagram in Fig. 7.6.

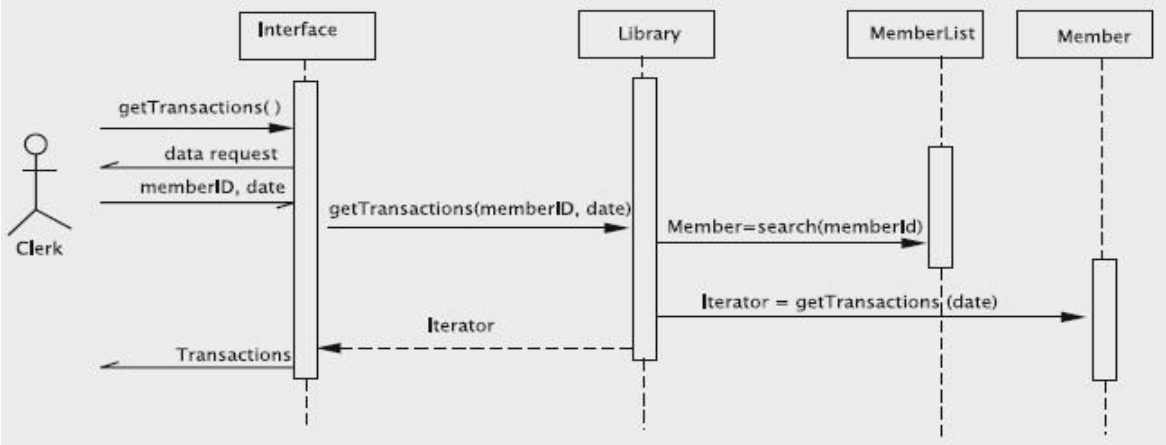


Fig. 7.6 Sequence diagram for printing a member's transactions

Place Hold

As discussed earlier, we create a separate Hold class for representing the holds placed by members. Each Hold object stores references to a Member object and a Book object, and the date when the hold expires (see Fig. 7.7). When a clerk issues request to the library to place a hold on behalf of a member for a certain book, the Library object itself creates an instance of Hold and makes both the Book and Member instances involved to store references to it. The UI is informed of the outcome by a result code.

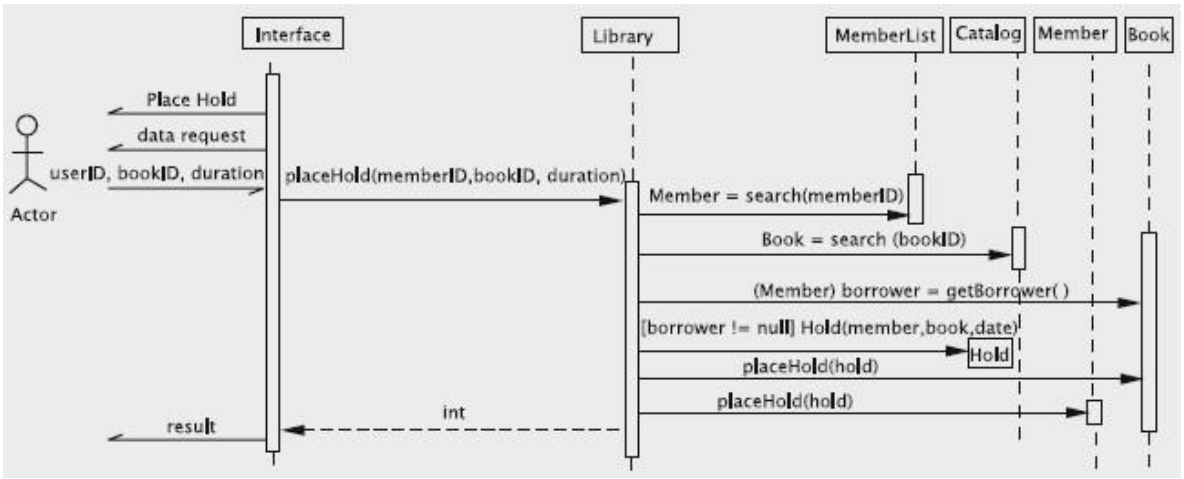


Fig. 7.7 Sequence diagram for placing a hold

Process Holds

The input here is only the ID for the book, fromwhich we get the next hold that has not expired. In this process, the bookwould quite likely find some holds that are not valid. These holds should obviously be removed from the system and the responsibility for this clean up is assigned to the getNextHold() method in Book. The Library gets a reference to the Member object from Hold (see Fig. 7.8) and returns this to the UI.

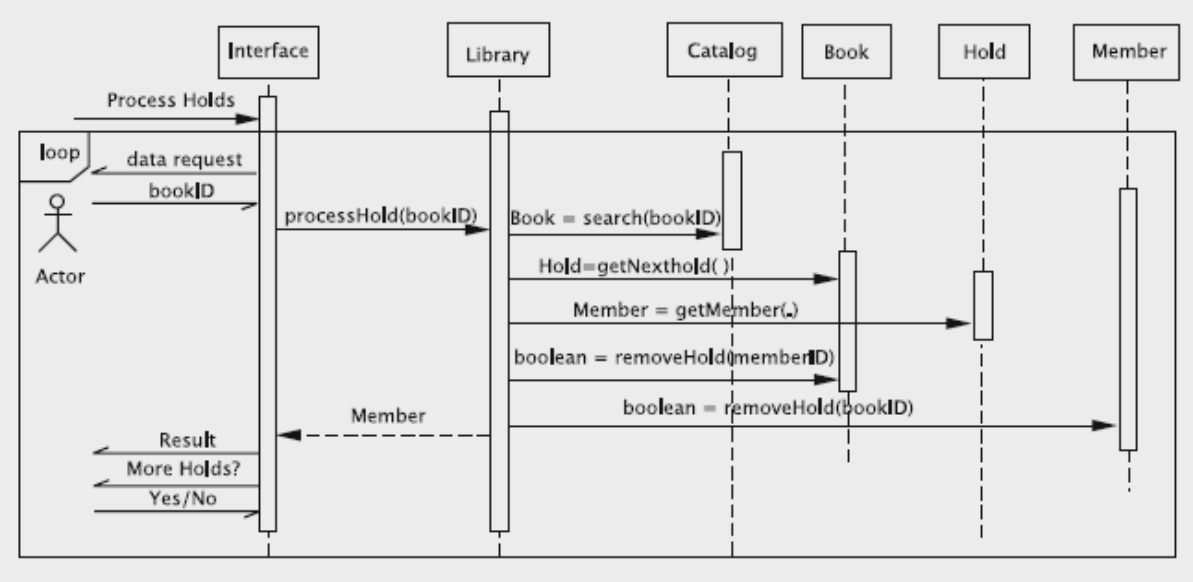


Fig. 7.8 Sequence diagram for processing holds

Remove Hold

The sequence diagram is given in Fig. 7.9. A request is issued to Library via the method removeHold. Library retrieves the corresponding Member and Book objects using MemberList and Catalog and then invokes the removeHold method on these objects to delete their references to the Hold object.

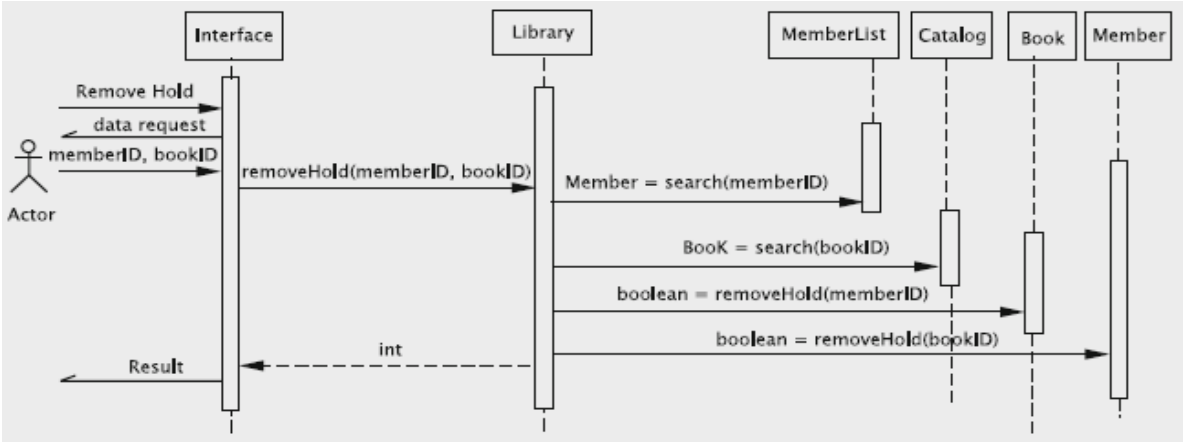


Fig. 7.9 Sequence diagram for removing a hold

Renew Books

Figure 7.10 details the implementation for renewing books. This process involves interactively updating the information on several members of a collection. We can accomplish this by allowing UserInterface to get an enumeration (Iterator) of the items in the collection, getting responses on each from the user and invoking the methods on the library to update the information.

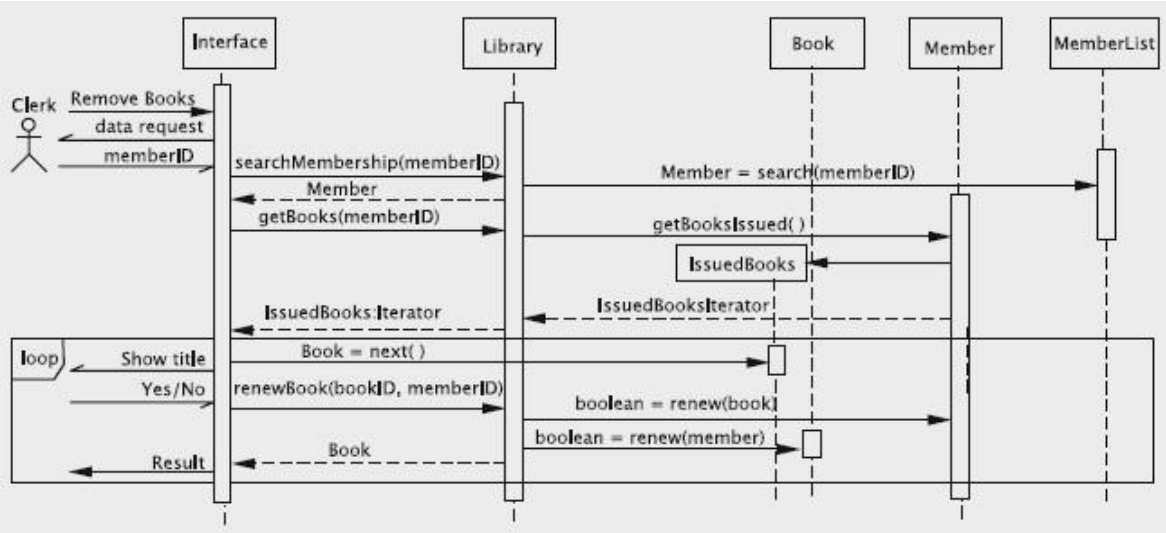


Fig. 7.10 Sequence diagram for renewing books

Class Diagrams

At this stage, we have come up with all the software classes.

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

The relationships between these classes is shown in Fig. 7.11. Note that Hold is not shown as an association class, but an independent class that connects Member and Book. The new class Transaction is added to record transactions; this has a dependency on Book since it stores the title of the book.

By inspecting the sequence diagrams, we can collect the methods of each of these classes, and draw a class diagram for each. In specifying the types of attributes, we have to make language-specific choices; in the process of doing this we transition from the software classes to the implementation classes.

Class Diagram for Library

The methods are simply a collection of methods with their parameters as given in the sequence diagrams. However, we have specified their return types, which were not clearly specified in the sequence diagrams. Whenever something is added to the system such as a member or a book or a hold, some information about the added object is returned, so that the clerk can verify that the data was correctly recorded.

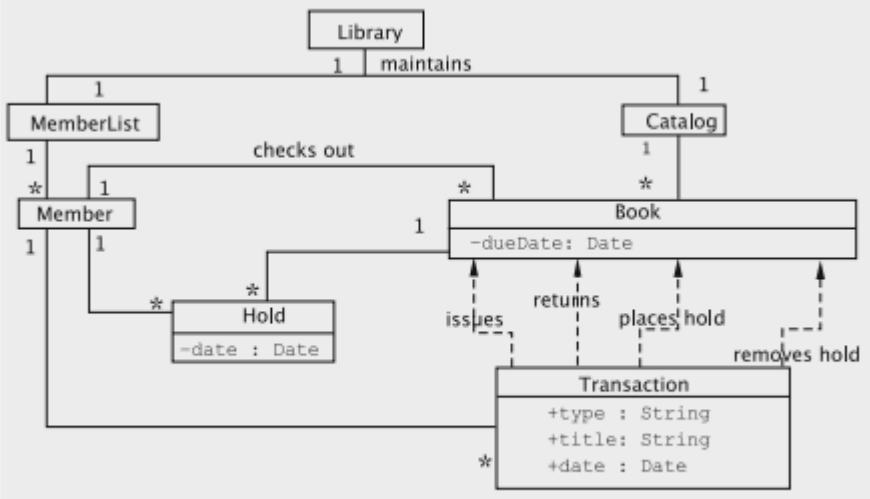


Fig. 7.11 Relationships between the software classes

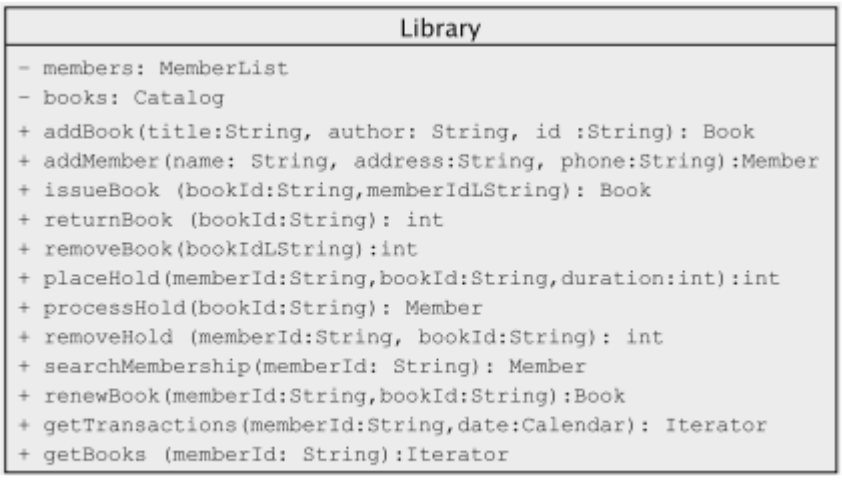


Fig. 7.12 Class diagram for Library

We have already seen that the class must maintain references to Catalog and MemberList. See Fig. 7.12 for the class diagram.

Class Diagram for Member

Once again, we get our methods and attributes by examining the sequence diagrams. In our design, we make the Member class generate the member ID. We need a mechanism to ensure that no two members get the same ID, i.e., there has to be some central place where we keep track of how ids are generated. It would be tempting to do this in the Library class, but the right solution would be to make it a static method in the Member class. This gives us decentralised control and places responsibilities close to the data. The class diagram is given in Fig. 7.13.

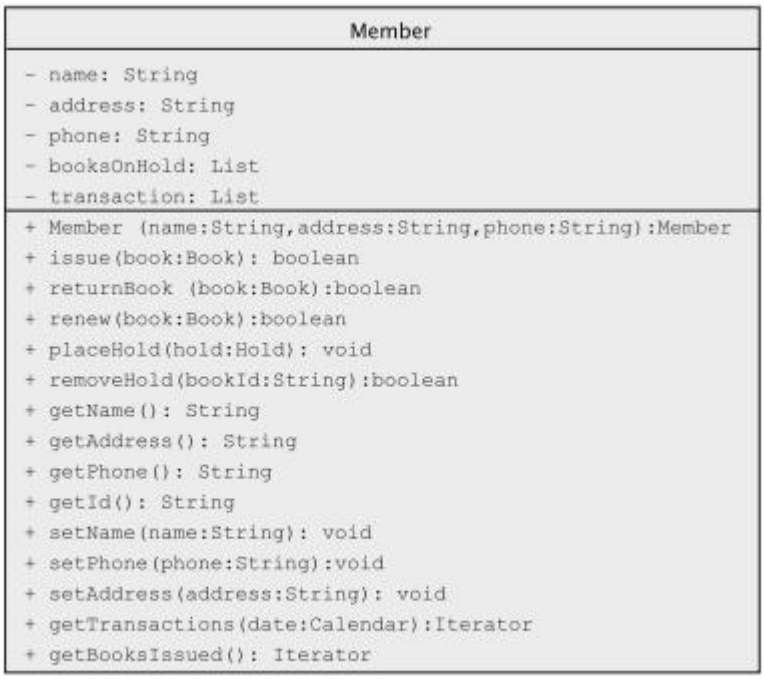


Fig. 7.13 Class diagram for Member

Class Diagram for Book

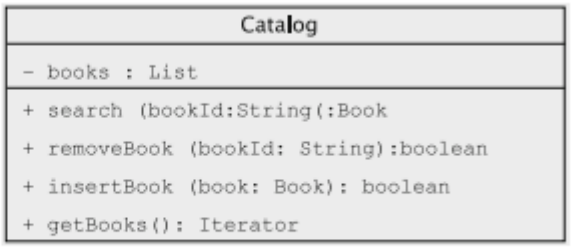
The approach to developing the class diagram for Book parallels that of the approach for the Member class. As in the other cases, we now add the attributes. However, there are no setters for the Book class because we don't expect to change anything in a Book object (see Fig. 7.14).

Class Diagram for Catalog

Typical operations on a list would be add, remove, and search for objects. Proceeding as in the case for the Library class, we obtain the methods shown in Fig. 7.15. The only attribute that we come up with is a List object that stores Book objects. The reader will also notice the method getBooks, whose return type is Iterator. This enables the Library to get an enumeration of all the books so that any specialised operations that have to be applied to the collection are facilitated.

Fig. 7.14 Class diagram for the Book class

Fig. 7.15 Class diagram for the Catalog class



Class Diagram for MemberList

The derivation of this is fairly straightforward after developing the Catalog class and is shown in Fig. 7.16. Since we never asked for the functionality of removing a member, there is no such method in the class. We need an attribute of type List to store the members

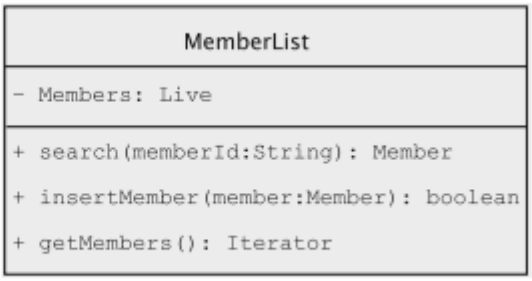


Fig. 7.16 Class diagram for the MemberList class

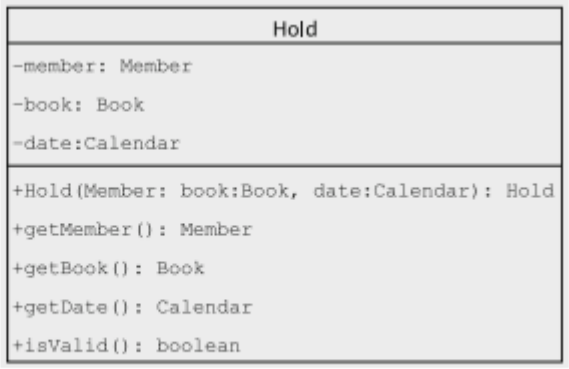


Fig. 7.17 Class diagram for Hold

Class Diagram for Hold

Besides the accessors, getMember, getBook, and getDate, the class diagram for Hold (Fig. 7.17) shows the isValid method, which checks whether a certain hold is still valid.

Class Diagram for Transaction

The class diagram is shown in Fig. 7.18. Note that we have to store the date for each transaction, i.e., we need to choose an appropriate type for this attribute. Java's util package has a class Calendar that provides the needed functionality.

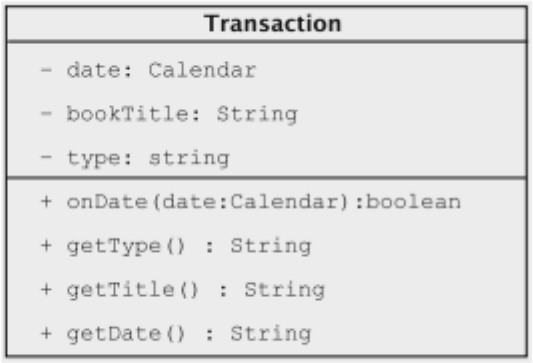


Fig. 7.18 Class diagram for Transaction

User Interface

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

1 Add a member

2 Add books

3 Issue books

4 Return books

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

Initially, the system will display a menu. The user can enter a number from 0 through 13 indicating the operation. (The options 0 and 13 will be used to exit the system and display the help screen respectively.) Parameters required for the operation will be prompted. The result of the operation is then displayed.

Data Storage

Ultimately, most applications will need to store data on a long-term basis. In a fullblown system, data is usually stored in a database, and this data is managed by a database management system. To avoid digressing, however, we will adopt a simple approach to store data on a long-term basis. Recall that we had decided to include the following commands in our UI.

1. A command to save the data on a long-term basis.

2. A command to load data from a long-term storage device.

When the first command is executed, we will copy all of the data onto secondary storage. Similarly, when the second command is executed, the data stored on the storage device is copied to recreate the object.

Implementing Our Design

In this phase, we code, test, and debug the classes that implement the business logic (Library, Book, etc.) and UserInterface. An important issue in the implementation is the communication via the return values between the different classes: in particular between Library and UserInterface; 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

We are now ready to complete our development by writing the code. 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;
  }
}
```

The private constructor checks whether a serialized version of the Library object exists. (We assume that it is stored in a file called ‘LibraryData’.) The File class in Java is a convenient mechanism to check the existence of files. The user is given an option to retrieve any serialized version of the Library object. (We will explain later how the problem of safely combining serialization and singletons is tackled.) In any case, UserInterface gets an instance of Library.

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


   }
   library = Library.instance();
}
```

Following this, the process method of UserInterface is executed, which initialises a loop that provides the user with a list of options. This code snippet is given below.

```
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

The addBooks method in UserInterface is shown below:

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

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

The loop is set up in UserInterface, all the input is collected, and the addBook method in Library is invoked. Following the sequence diagram, this method is implemented in Library as follows:

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

In the above code, the constructor for Book is invoked and the new book is added to the catalog. 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, as shown below.

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

Issuing Books

Once again, UserInterface gets the member's ID and sets up the loop. Here, UserInterface remembers the member's ID throughout the process. The issue Book method of Library is repeatedly invoked and the response to the actor is generated based on the value returned by each invocation.

```
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 method in Library does 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;
  }
```

Printing Transactions

Library provides a query that returns an Iterator of all the transactions of a member on a given date, and this is implemented by passing the query to the appropriate Member object. The method getTransactions in Member filters the transactions based on the date and returns an Iterator of the filtered collection.

```
public Iterator getTransactions(Calendar date) {
  List result = new LinkedList();
  for (Iterator iterator = transactions.iterator(); iterator.hasNext(); ) {
    Transaction transaction = (Transaction) iterator.next();
    if (transaction.onDate(date)) {
      result.add(transaction);
    }
  }
  return (result.iterator());
}
```

Library returns null when the member is not in MemberList; otherwise an iterator to the filtered collection is returned. The UI extracts the necessary information and displays it in the preferred format.

```
public void getTransactions() {
  Iterator result;
  String memberID = getToken("Enter member id");


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

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. In our implementation, the Member and Book objects store the reference to the Hold object. The placeHold method in both Book and Member simply appends the new hold to the list. (The code for Book is shown below.)

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

One 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. This is left as an exercise.

The list booksOnHold in Member keeps a collection of all the active holds the member has placed. In the Member class we also generate a transaction whenever a hold is placed.

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

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;
}
```

The Hold class is shown below. There are no modifiers for the attributes, since a hold cannot be changed once it has been placed. The method isValid() checks if the hold is still valid.

```
public class Hold implements Serializable {
  private Book book;
  private Member member;
  private Calendar date;
  public Hold(Member member, Book book, int duration) {
    this.book = book;
    this.member = member;
    date = new GregorianCalendar();
    date.setTimeInMillis(System.currentTimeMillis());
    date.add(Calendar.DATE, duration);
  }
  public Member getMember() {
    return member;
  }
  public Book getBook() {
    return book;
  }
  public Calendar getDate() {
    return date;
  }
  public boolean isValid() {
    return (System.currentTimeMillis() < date.getTimeInMillis());
  }
}
```

Once the reference to the Hold object has been found in the Book, the hold is removed from the book and from the corresponding member as well. The book's ID is passed to the removeHold method in Member, which is shown below.

```
public boolean removeHold(String bookId) {
  boolean removed = false;
  for (ListIterator iterator = booksOnHold.listIterator();
                               iterator.hasNext(); ) {
    Hold hold = (Hold) iterator.next();
    String id = hold.getBook().getId();
    if (id.equals(bookId)) {
      transactions.add(new Transaction ("Hold Removed ",
                                        hold.getBook().getTitle()));
      removed = true;
      iterator.remove();
    }
  }
  return removed;
}
```

Storing and Retrieving the Library Object

Java Serialization

Our approach to long-term storage of the library data uses the Java serializationmechanism. We saw that 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.

In our current example, Book and Hold can be serialized by simply declaring them to be Serializable. This is because they contain instance fields each of which is defined to be Serializable. (The reader can verify this by examining the documentation of the Java classes we use, such as GregorianCalendar and LinkedList and the definition of Book and Hold.) Member, MemberList, Catalog, and Library need more work because they all have static fields in them. The default serialization mechanism in Java does not store static fields.

Storing the Data

What should we do to store the entire data? Observe that Library has references to both the Catalog and MemberList objects, which, in turn, have references to the Book and Member objects respectively; the Hold objects are referred to by the Book objects and the Member objects. Thus, if we simply store the Library object, all of the data will be stored. As in our earlier use cases, we would like to keep these details out of the UI, and so UserInterface has a save method that simply invokes a save method on the Library object.

```
private void save() {
  if (library.save()) {
    System.out.println("The library has been successfully saved" );
  } else {
    System.out.println("There has been an error in saving \n" );
  }
}
```

The save method in Library could simply write the Library object to a file named 'LibraryData' and return true if nothing goes wrong, as shown below.

```
FileOutputStream file = new FileOutputStream("LibraryData");
ObjectOutputStream output = new ObjectOutputStream(file);
output.writeObject(library);
return true;
```

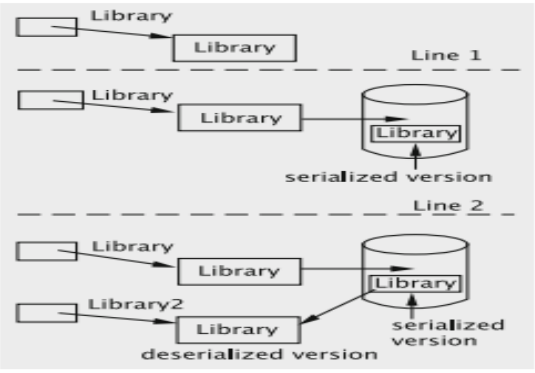
Maintaining the Singleton Property

The process of retrieving the data has some subtle complications associated with it. The Library, MemberList and Catalog objects are singletons: they cannot have more than one instance. Using the serialization mechanism, it is now possible to serialize an object and then deserialize it to get a second instance. For example, see the following pseudocode.

```
Library library = Library.instance();
Serialize library onto a disk file "library1";
Library library2 = Deserialized version of "library1";
Update library (add a member);
Update library2 (delete a book);
```

The first three lines of the pseudo-code are shown pictorially in Fig. 7.19. What has happened is that some user of the Library object initially obtained an instance of the Library object: essential and valid. In the second line, the user makes a copy of the object on disk: this is also perfectly legal and necessary. What follows in the third step is the problem. The user is now able to deserialize the object and obtain a second copy. The two copies can then diverge via independent updates as in the last two lines.

Fig. 7.19 A pitfall in using serialization with a singleton



To understand what the essential problem is, recall that the intent of the singleton pattern is to ensure that a class has only one instance and provide a global point of access to it. We now have two mechanisms that can create instances of a class: (i) constructors and (ii) deserialization. The first mechanism was controlled by making constructors private and requiring all instantiations to got through the instance method. We now need a way of restricting the creation mechanism of deserialization.

Fortunately, due to the manner in which the reading of objects takes place in Java, this is not a complicated task. The default readObject method can be overridden to ignore retrieval if a copy already exists in memory. This way, no other class such as UserInterface will be able to do direct deserialization.

```
private void readObject(java.io.ObjectInputStream input) {
  try {
    input.defaultReadObject();
    if (library == null) {


      library = (Library) input.readObject();
    } else {
      input.readObject();
    }
   } catch(IOException ioe) {
     ioe.printStackTrace();
   } catch(Exception e) {
     e.printStackTrace();
   }
 }
```

If there is no memory-resident copy of the Library object, the retrieve method reads the disk copy; otherwise, it returns the copy in memory. In case of an unexpected error, it returns null.

```
public static Library retrieve() {
  try {
    FileInputStream file = new FileInputStream("LibraryData");
    ObjectInputStream input = new ObjectInputStream(file);
    input.readObject();
    return library;
  } catch(IOException ioe) {
    ioe.printStackTrace();
    return null;
  } catch(ClassNotFoundException cnfe) {
    cnfe.printStackTrace();
    return null;
  }
}
```

Discussion and Further Reading

Conceptual, Software and Implementation Classes

In the analysis phase, we found the **conceptual** classes. These correspond to real world concepts or things, and present us with a conceptual or essential perspective. These are derived from and used to satisfy the system requirements at a conceptual level. At this level, for instance, we can identify a piece of information that needs to be recognised as an entity and make it a class; we can talk of an association between classes without any thought to how this will be realised.

As we go further into the design process and construct the sequence diagrams, we are now dealing with **software** classes. These can be implemented with typical programming

languages, and we need to identify methods and parameters that will be involved. We have to finalise which entities will be individual classes, which ones will be merged, and how associations will be captured.

The last step is the **implementation** class, which is a class created using a specific programming language such as Java or C++. This step nails down all the remaining details: identification and implementation of helper methods, the nitty-gritty of using software libraries, names of fields and variables, etc.

The process of going from conceptual to implementation classes is a progression from an abstract system to a concrete one and, as we have seen, classes may be added or removed at each step. For instance, Transaction and MemberIdServer were added as software and implementation class respectively, whereas the conceptual class Borrows was dropped.

Building a Commercially Acceptable System

Non-functional Requirements

A realistic system would have several non-functional requirements. Giving a fair treatment to these is beyond the scope of the book. Some issues like portability are automatically resolved since Java is interpreted and is thus platform independent. Response time (run-time performance) is a sticking point for object-oriented applications. We can examine this in a context where design choice affects performance; this is addressed briefly in a later case-study. **Functional Requirements**

It can be argued that for a system to be accepted commercially, it must provide a sufficiently large set of services, and if our design methodologies are not adequate to handle that complexity, then they are of questionable value. We would like to point out the following:

• Additional features can be easily added: Some of these will be added in the next chapter. Our decision to exclude several such features has been made based on pedagogical considerations.

• Allowing for variability among kinds of books/members: This variability is typically incorporated by using inheritance. To explain the basic design process, inheritance is not essential. However using inheritance in design requires an understanding of several related issues, and we shall in fact present these issues and extend our library system.

• Having a more sophisticated interface: Once again, we might want a system that allows members to login and perform operations through a GUI. This would only involve the interface and not the business logic. We shall see how a GUI can be modeled as a multi-panel interactive system, and how such features can be incorporated.

• Allowing remote access: Now-a-days, most systems of this kind allow remote access to the server, looks how such features can be introduced through the use of distributed objects.

The Facade Pattern

We discussed our preference for keeping the interface away from the complexity of the business logic implementation. This was done by having a Library class that provided a set of methods for the interface and thus served as a single point of entry to and exit from the

business logic module. In the language of design patterns, what we created is known as a **facade**.

The structure of the facade is shown in Fig. 7.20. The primary motivation behind using a facade is to reduce the complexity by minimising communication and dependencies between a subsystem and its clients (Fig. 7.21). The facade not only shields the client from the complexity but also enables loose coupling between the subsystem and its clients. Facades are not typically designed to prevent the client from accessing the components within the subsystem.

Fig. 7.20 Structure diagram for facade

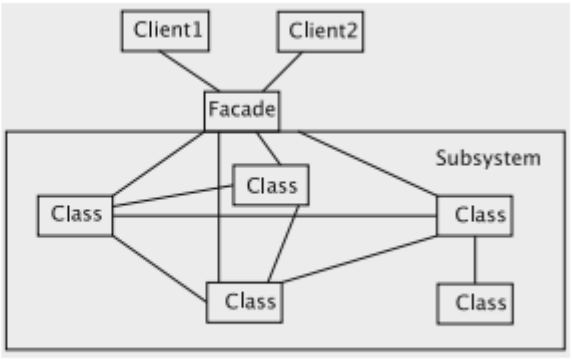
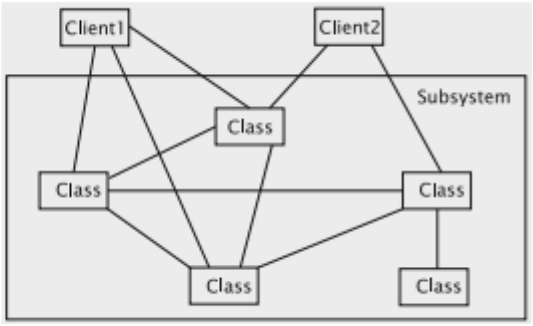


Fig. 7.21 Interactions with a subsystem without a facade



Implementing Singletons

Implementing a singleton correctly is not a trivial matter. We overcame the difficulties with creating a singleton hierarchy. In this chapter we have dealt with the issue of serialization. These solutions are very language specific and a careful study of the language features is needed when moving from the software classes to the implementation classes.

There do not appear to be any ‘standard mechanisms’ in the literature for handling implementation issues. Most languages provide a general collection of features that can be adapted for a variety of purposes. We have used the implementation of readObject and writeObject in Java to ensure that our purpose is served. Java also provides other methods like readResolve and writeReplace to override the effects of serialization and deserialization. The Externalisable interface can be employed when the serialization has to be fully customised.

Module 3

Design Pattern Catalog

Structural Patterns

ADAPTER

Intent

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

Also Known As

Wrapper

Motivation

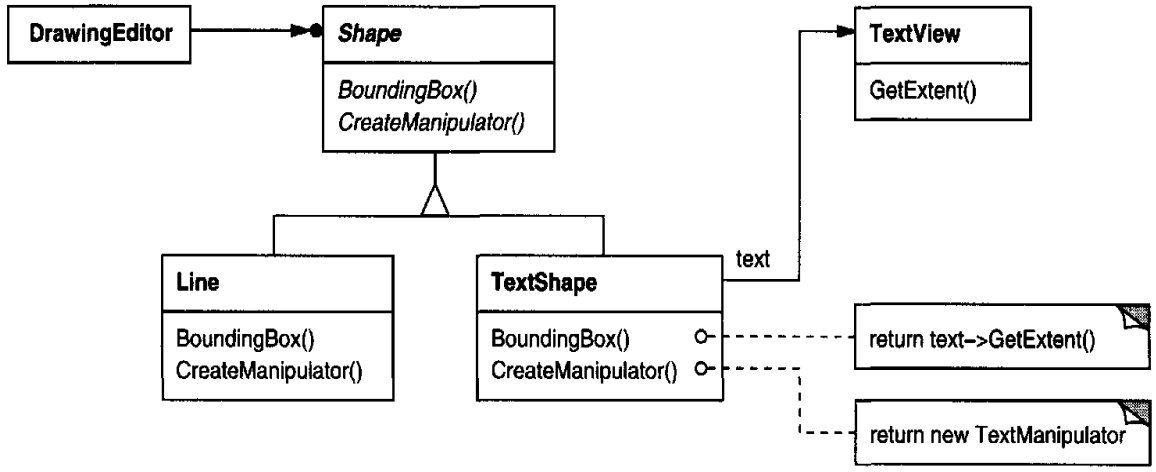
Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Consider for example a drawing editor that lets users draw and arrange graphical elements (lines, polygons, text, etc.) into pictures and diagrams.

The drawing editor's key abstraction is the graphical object, which has an editable shape and can draw itself. The interface for graphical objects is defined by an abstract class called Shape.

The editor defines a subclass of Shape for each kind of graphical object: a LineShape class for lines, a PolygonShape class for polygons, and so forth.

Classes for elementary geometric shapes like LineShape and PolygonShape are rather easy to implement.



Meanwhile, an off-the-shelf user interface toolkit might already provide a sophisticated TextView class for displaying and editing text.

This diagram illustrates the object adapter case. It shows how BoundingBox requests, declared in class Shape, are converted to GetExtent requests defined in TextView.

Since TextShape adapts TextView to the Shape interface, the drawing editor can reuse the otherwise incompatible TextView class.

Applicability

Use the Adapter pattern when

• you want to use an existing class, and its interface does not match the one you need.

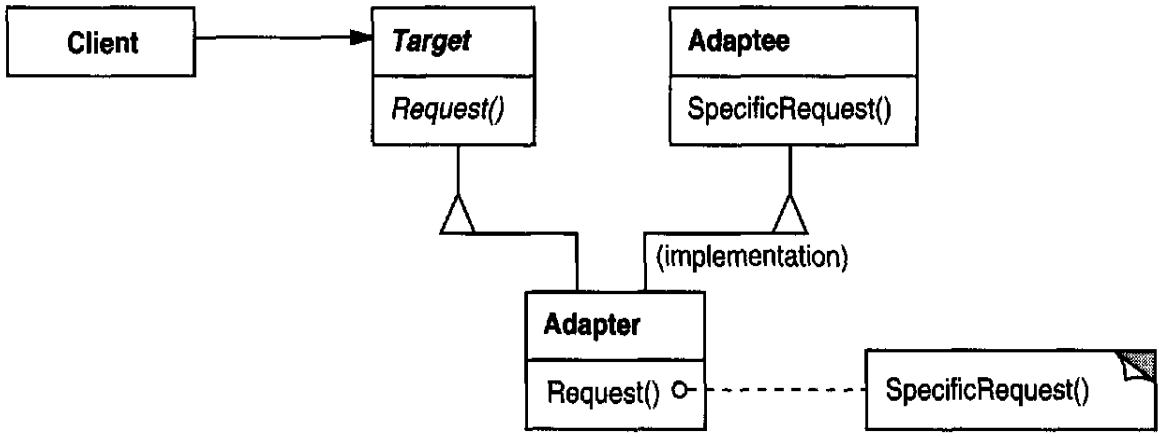
• you want to create a reusable class that cooperates with unrelated or unforeseen

classes, that is, classes that don't necessarily have compatible interfaces.

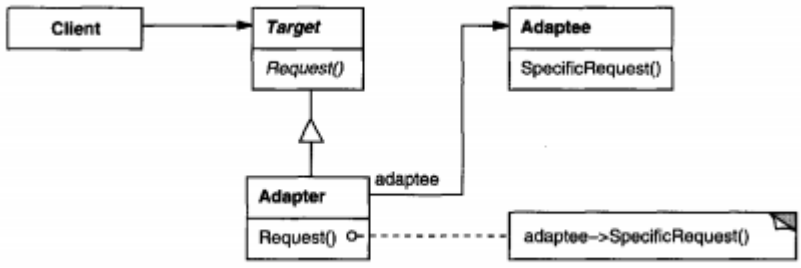
• *(object adapter only) you need to use several existing subclasses, but it's unpractical* to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class

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.

• Adaptec (TextView) - defines an existing interface that needs adapting.

• Adapter (TextShape) - adapts the interface of Adaptec to the Target interface.

Collaborations

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

Consequences

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

• adapts Adaptee to Target by committing to a concrete 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.

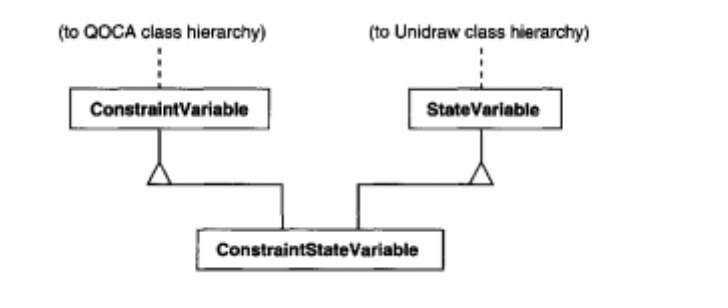
Here are other issues to consider when using the Adapter pattern:

1. How much adapting does Adapter do? Adapters vary in the amount of work they do to adapt Adaptee to the Target interface. There is a spectrum of possible work, from simple interface conversion—for example, changing the names of operations—to supporting an entirely different set of operations. The amount of work Adapter does depends on how similar the Target interface is to Adaptee's.

2. Pluggable adapters. A class is more reusable when you minimize the assumptions other classes must make to use it. By building interface adaptation into a class, you eliminate the assumption that other classes see the same interface. Put another way, interface adaptation lets us incorporate our class into existing systems that might expect different interfaces to the class. Object Works\Smalltalk [Par90] uses the term **pluggable adapter** to describe classes with built-in interface adaptation.

3. Using two-way adapters to provide transparency. A potential problem with adapters is that they aren't transparent to all clients. An adapted object no longer conforms to the Adaptec interface, so it can't be used as is wherever an Adaptec object can. Two-way adapters can provide such transparency. Specifically, they're useful when two different clients need to view an object differently.

Consider the two-way adapter that integrates Unidraw, a graphical editor framework [VL90], and QOCA, a constraint-solving toolkit [HHMV92]. Both systems have classes that represent variables explicitly: Unidraw has StateVariable, and QOCA has ConstraintVariable. To make Unidraw work with QOCA, ConstraintVariable must be adapted to StateVariable; to let QOCA propagate solutions to Unidraw, StateVariable must be adapted to ConstraintVariable



The solution involves a two-way class adapter ConstraintStateVariable, a subclass of both StateVariable and ConstraintVariable that adapts the two interfaces to each other. Multiple inheritance is a viable solution in this case because the interfaces of the adapted classes are substantially different. The two-way class adapter conforms to both of the adapted classes and can work in either system.

Implementation

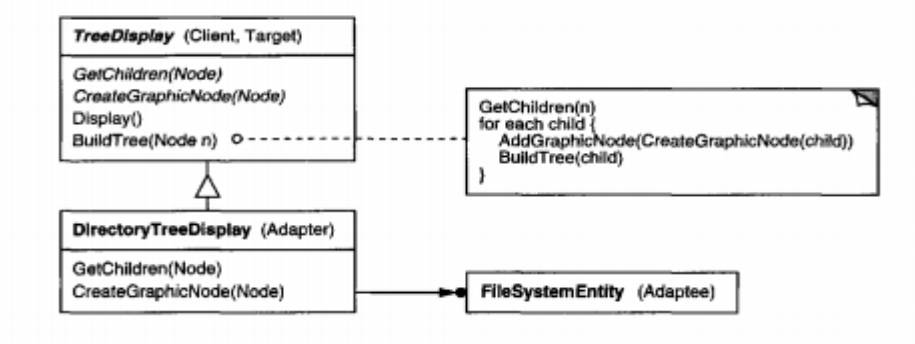
Although the implementation of Adapter is usually straightforward, here are some issues to keep in mind:

1. Implementing class adapters in C++.In a C++ implementation of a class adapter, Adapter would inherit publicly from Target and privately from Adaptec. Thus Adapter would be a subtype of Target but not of Adaptec.

2. Pluggable adapters. Let's look at three ways to implement pluggable adapters for the TreeDisplay widget described earlier, which can lay out and display a hierarchical structure automatically. The first step, which is common to all three of the implementations discussed here, is to find a "narrow" interface for Adaptec, that is, the smallest subset of operations that lets us do the adaptation. A narrow interface consisting of only a couple of operations is easier to adapt than an interface with dozens of operations. For TreeDisplay, the adaptee is any hierarchical structure. A minimalist interface might include two operations, one that defines how to present a node in the hierarchical structure graphically, and another that retrieves the node's children.

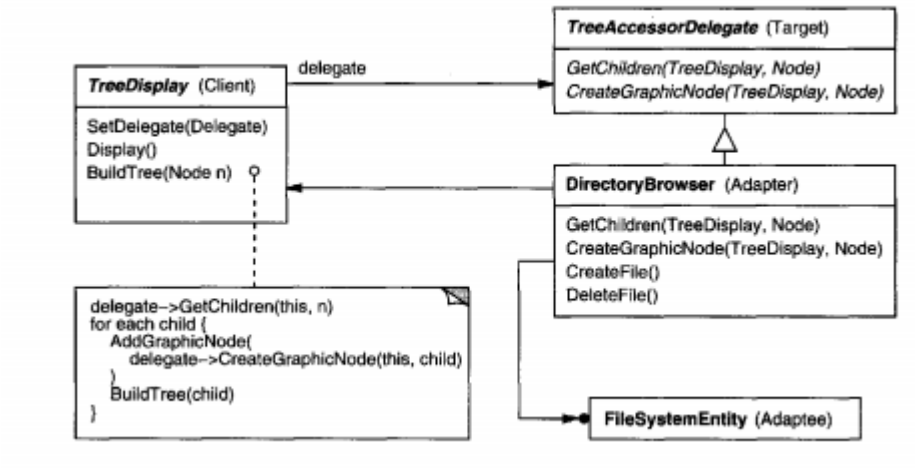
The narrow interface leads to three implementation approaches:

(a) Using abstract operations. Define corresponding abstract operations for the narrow Adaptee interface in the TreeDisplay class. Subclasses must implement the abstract operations and adapt the hierarchically structured object. For example, a DirectoryTreeDisplay subclass will implement these operations by accessing the directory structure.



(b) Using delegate objects. In this approach, TreeDisplay forwards requests for accessing the hierarchical structure to a delegate object. TreeDisplay can use a different adaptation strategy by substituting a different delegate.

For example, suppose there exists a DirectoryBrowser that uses a TreeDisplay. DirectoryBrowser might make a good delegate for adapting TreeDisplay to the hierarchical directory structure. In dynamically typed languages like Smalltalk or Objective C, this approach only requires an interface for registering the delegate with the adapter. Then TreeDisplay simply forwards the requests to the delegate. NEXTSTEP [Add94] uses this approach heavily to reduce subclassing.



(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. In our example, this means TreeDisplay stores one block for converting a node into a GraphicNode and another block for accessing a node's children.

For example, to create TreeDisplay on a directory hierarchy, we write

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

Known Uses

The Motivation example comes from ET++Drawing Editor.

Related Patterns

Bridge has a structure similar to an object adapter, but Bridge has a different intent: It is meant to separate an interface from its implementation so that they can be varied easily and independently. An adapter is meant to change the interface of an existing object.

Decorator enhances another object without changing its interface. A decorator is thus more transparent to the application than an adapter is. As a consequence, Decorator supports recursive composition, which isn't possible with pure adapters.

Proxy defines a representative or surrogate for another object and does not change its interface.

BRIDGE

Intent

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

Also Known As

Handle/Body

Motivation

* When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance.

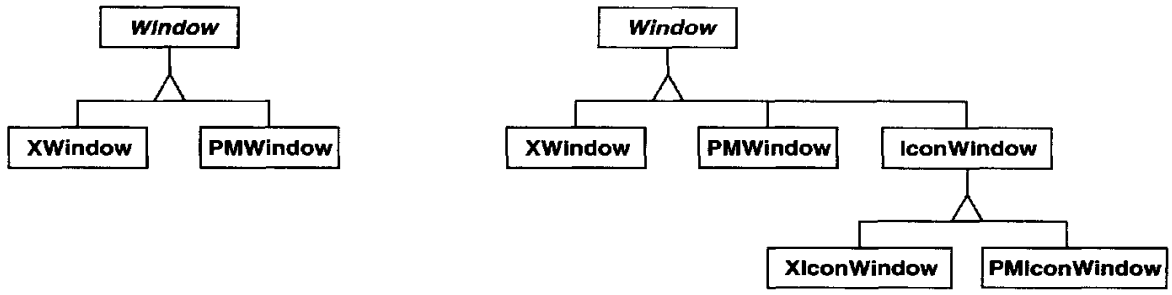
* An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways.

* But this approach isn't always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extends and reuse abstractions and implementations independently.

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

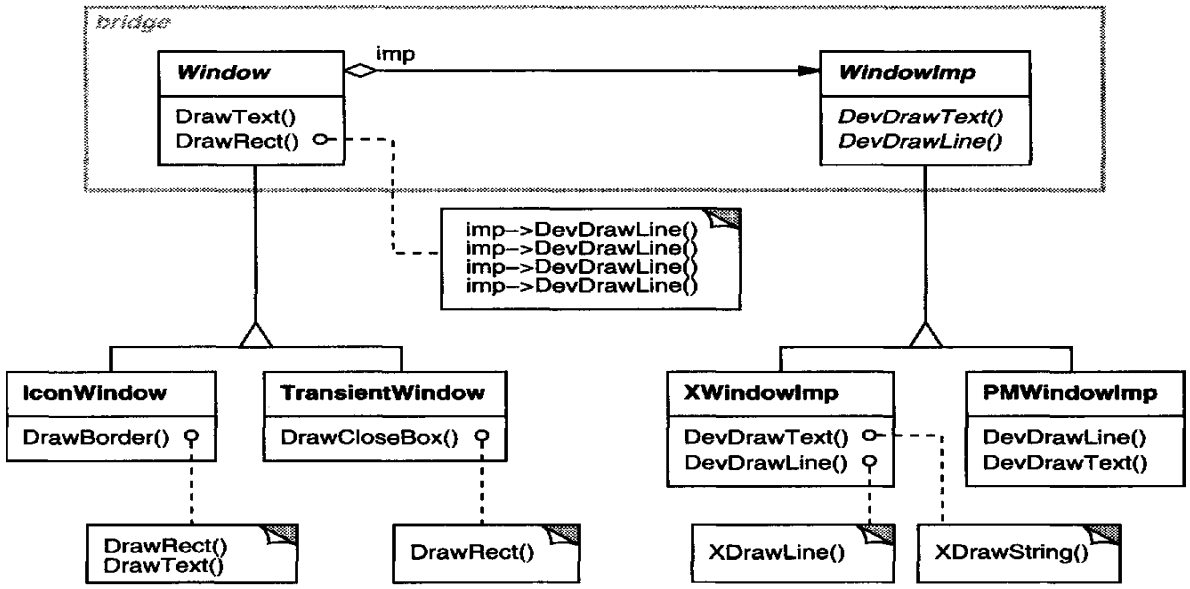
But this approach has two drawbacks:

1. It's inconvenient to extend the Window abstraction to cover different kinds of windows or new platforms. **Imagine an IconWindow subclass of Window that specializes the Window abstraction for icons** . To support IconWindows for both platforms, we have to implement *two new classes, XlconWindow* and PMIconWindow.
2. It makes client code platform-dependent.



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

* There is one class hierarchy for window interfaces (Window, IconWindow, TransientWindow) and a separate hierarchy for platform-specific window implementations, with Windowlmp as its root. The XWindowImp subclass, for example, provides an implementation basedon the XWindow System.



Applicability

Use the Bridge pattern when

• you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.

• both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.

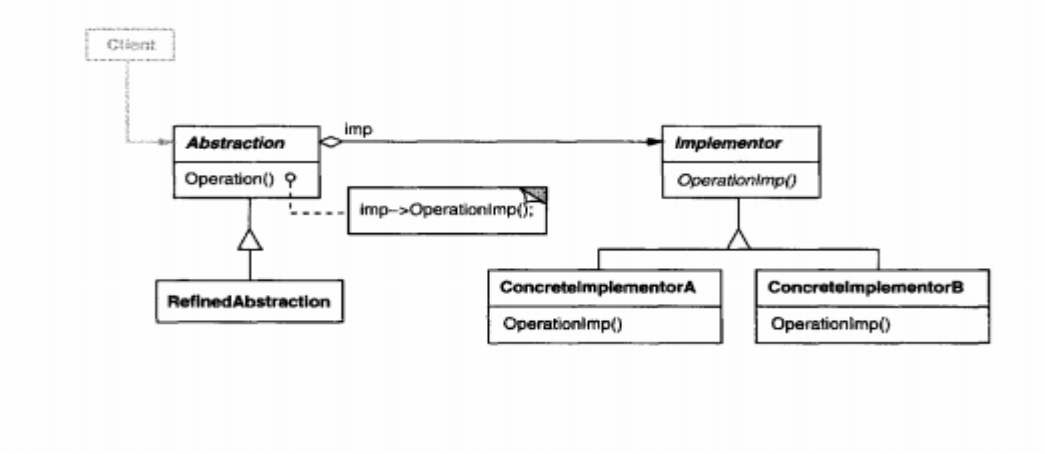
• changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.

• (C++) you want to hide the implementation of an abstraction completely from clients. In C++ the representation of a class is visible in the class interface.

• you have a proliferation of classes as shown earlier in the first Motivation diagram. Such a class hierarchy indicates the need for splitting an object into two parts. Rumbaugh uses the term "nested generalizations" [RBP+91] to refer to such class hierarchies.

• you want to share an implementation among multiple objects(perhaps using reference counting), and this fact should be hidden from the client. A simple example is Coplien's String class, in which multiple objects can share the same string representation (StringRep).

Structure



Participants

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

• RefmedAbstraction (IconWindow) – Extends the interface defined by Abstraction.

• Implementor (Windowlmp) - 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 also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't require recompiling the Abstraction class and its clients. This property is essential when you must ensure binary compatibility between different versions of a class library. Furthermore, this decoupling encourages layering that can lead to a better

structured system. The high-level part of a system only has to know about Abstraction and Implementor.

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

2. *Creating the right Implementor object*.

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

If Abstraction knows about all ConcreteImplementor classes, then it can instantiate one of them in its constructor; it can decide between them based on parameters passed to its constructor. If, for example, a collection class supports multiple implementations, the decision can be based on the size of the collection. A linked list implementation can be used for small collections and a hash table for larger ones.

Another approach is to choose a default implementation initially and change it later according to usage. For example, if the collection grows bigger than a certain threshold, then it switches its implementation to one that's more appropriate for a large number of items

3. *Sharing implementors*.

Coplien illustrates how the Handle/Body idiom in C++ can be used to share implementations among several objects [Cop92]. The Body stores a reference count that the Handle class increments and decrements. The code for assigning handles with shared bodies has the following general form:

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

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

    return *this;
}
```

4. *Using multiple inheritance.*

You can use multiple inheritance in C++ to combine an interface with its implementation .For example, 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 inheritances—at least not in C++.

Known Uses

The Window example above comes from ET++ [WGM88]. In ET++, Windowlmp is called "WindowPort" and has subclasses such as XWindowPortand SunWindowPort. The Window object creates its corresponding Implementor object by requesting it from an abstract factory called "WindowSystem." WindowSystem provides an interface for creating platform-specific objects such as fonts, cursors, bitmaps, and so forth.

The ET++ Window/WindowPort design extends the Bridge pattern in that the WindowPort also keeps a reference back to the Window. The WindowPort implementor class uses this reference to notify Window about WindowPort-specific events: the arrival of input events, window resizes, etc.

Related Patterns

An Abstract Factory can create and configure a particular Bridge.

The Adapter pattern is geared toward making unrelated classes work together. It is usually applied to systems after they're designed. Bridge, on the other hand, is used up-front in a design to let abstractions and implementations vary independently.

COMPOSITE

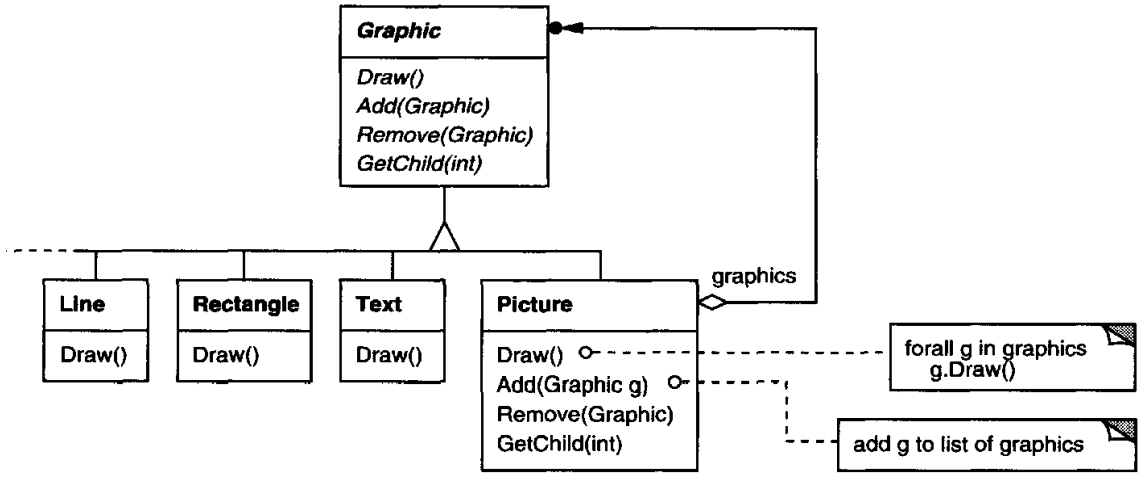
Intent

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

Motivation

Graphics applications like drawing editors and schematic capture systems let users build complex diagrams out of simple components. The user can group components to form larger components, which in turn can be grouped to form still larger components.

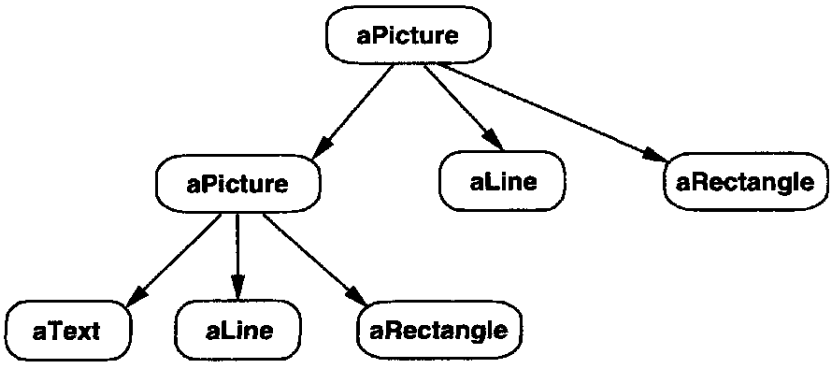
A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.



The key to the Composite pattern is an abstract class that represents *both primitives* and their containers. For the graphics system, this class is Graphic.

Graphic declares operations like Draw that are specific to graphical objects. It also declares operations that all composite objects share, such as operations for accessing and managing its children.

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



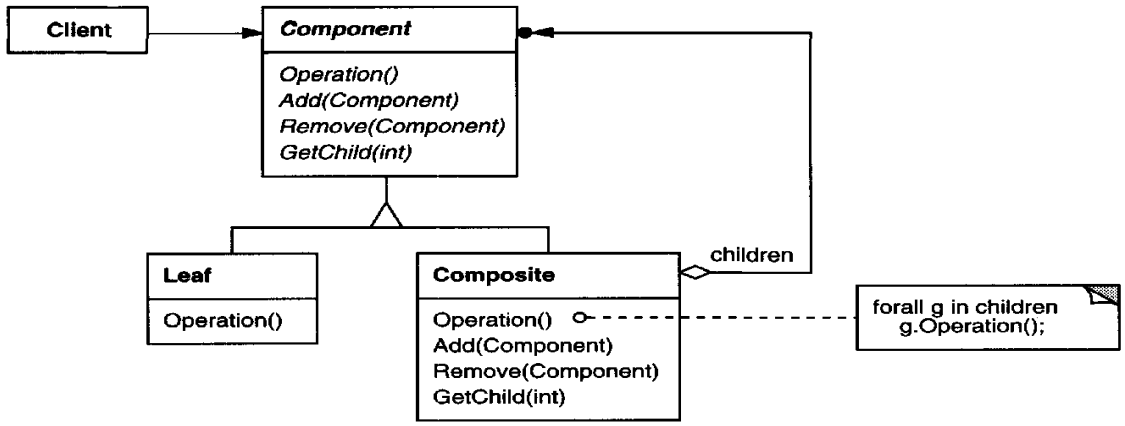
Applicability

Use the Composite pattern when

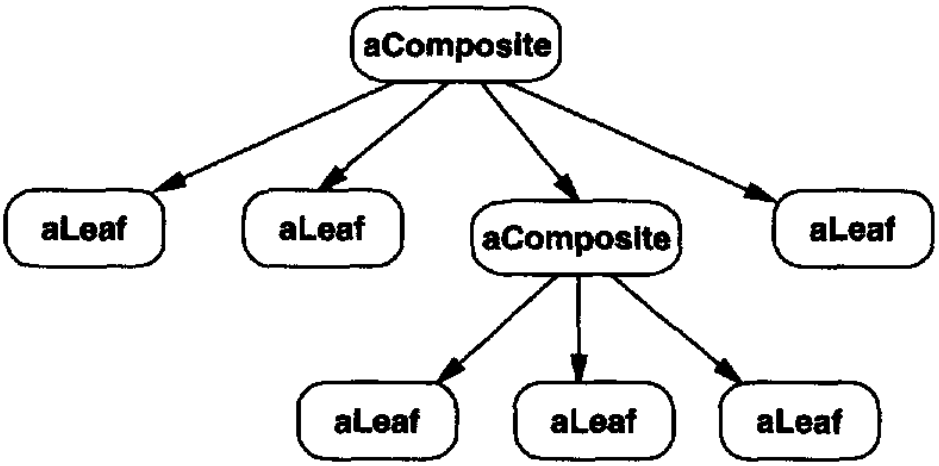
• you want to represent part-whole hierarchies of objects.

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

Structure



A typical Composite object structure might look like this:



Participants

• Component (Graphic)

- declares the interface for objects in the composition.

- Implements default behavior for the interface common to all classes, as appropriate.

- declares an interface for accessing and managing its child components.

- (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.

• Leaf (Rectangle, Line, Text, etc.)

- represents leaf objects in the composition. A leaf has no children.

- defines behavior for primitive objects in the composition.

• Composite (Picture)

- defines behavior for components having children.

- stores child components.

- implements child-related operations in the Component interface.

• Client

- manipulates objects in the composition through the Component interface.

Collaborations

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

Consequences

The Composite pattern

• defines class hierarchies consisting of primitive objects and composite objects. 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.

• *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.

• *makes it easier to add new kinds of 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. • 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*. Maintaining references from child components to their parent can simplify the traversal and management of a composite structure. The parent reference simplifies moving up the structure and deleting a component. Parent references also help support the Chain of Responsibility pattern.

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

3. *Maximizing the Component interface*. One of the goals of the Composite pattern is to make clients unaware of the specific Leaf or Composite classes they're using. To attain this goal, the Component class should define as many common operations for Composite and Leaf

classes as possible. The Component class usually provides default implementations for these operations, and Leaf and Composite subclasses will override them.

However, this goal will sometimes conflict with the principle of class hierarchy design that says a class should only define operations that are meaningful to its subclasses. There are many operations that Component supports that don't seem to make sense for Leaf classes.

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

• Defining the child management interface at the root of the class hierarchy gives you transparency, because you can treat all components uniformly. It costs you safety, however, because clients may try to do meaningless things like add and remove objects from leaves.

• 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 you lose transparency, because leaves and composites have different interfaces.

One approach isto 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 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 {
    // ...
};
```

GetComposite lets you query a component to see if it's a composite.You can perform Add and Remove safely on the composite it returns.

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
    test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
    test->Add(new Leaf); // will not add leaf
}
```

5. *Should Component implement a list of Components?* You might be tempted to define the set of children as an instance variable in the Component class where the child access and management operations are declared. But putting the child pointer in the base class incurs a space penalty for every leaf, even though a leaf never has children. This is worthwhile only if there are relatively few children in the structure.

6. *Child ordering*. Many designs specify an ordering on the children of Composite. In the earlier Graphics example, ordering may reflect front-to-back ordering. If Composites represent parse trees, then compound statements can be instances of a Composite whose children must be ordered to reflect the program. When child ordering is an issue, you must design child access and management interfaces carefully to manage the sequence of children. The Iterator pattern can guide you in this.

7. *Caching to improve performance*. If you need to traverse or search compositions frequently, the Composite class can cache traversal or search information about its children. The Composite can cache actual results or just information that lets it short-circuit the traversal or search. For example, the Picture class from the Motivation example could cache the bounding box of its children. During drawing or selection, this cached bounding box lets the Picture avoid drawing or searching when its children aren't visible in the current window.

Changes to a component will require invalidating the caches of its parents. This works best when components know their parents. So if you're using caching, you need to define an interface for telling composites that their caches are invalid.

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.

Known Uses

The RTL Smalltalk compiler framework [JML92] uses the Composite pattern extensively.

Related Patterns

Often the component-parent link is used for a Chain of Responsibility.

Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.

Flyweight lets you share components, but they can no longer refer to their parents.

Iterator can be used to traverse composites.

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

DECORATOR

Intent

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

Also Known As

Wrapper

Motivation

* Sometimes we want to add responsibilities to individual objects, not to an entire class.

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, however, 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.

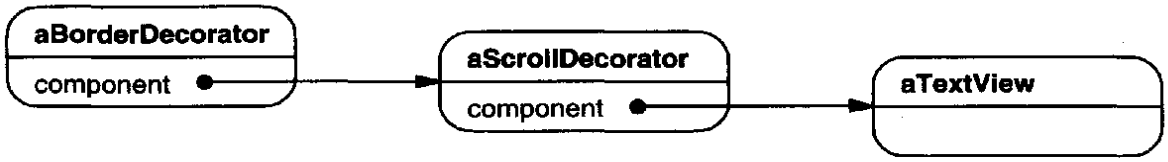
The decorator forwards requests to the component and may perform additional actions (such as drawing a border) before or after forwarding. Transparency lets you nest decorators recursively, thereby allowing an unlimited number of added responsibilities

For example, suppose we have a TextView object that displays text in a window.

TextView has no scroll bars by default, because we might not always need them. When we do, we can use a ScrollDecorator to add them.

Suppose we also want to add a thick black border around the TextView. We can use a BorderDecorator to add this as well.

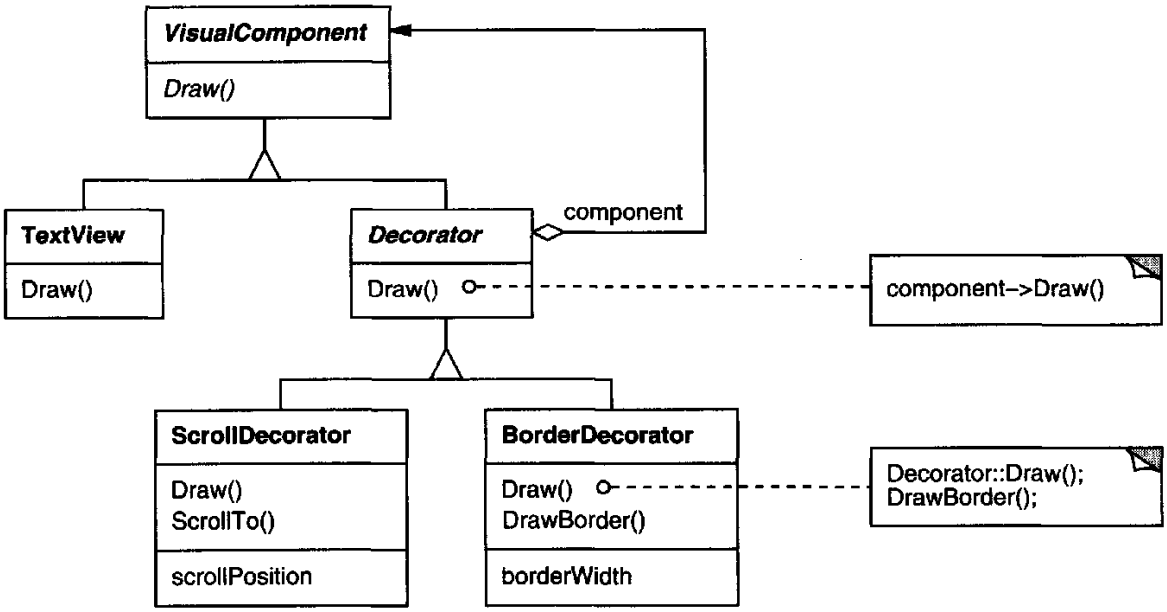
We simply compose the decorators with the TextView to produce the desired result



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

* VisualComponent is the abstract class for visual objects. It defines their drawing and event handling interface.

* Decorator subclasses are free to add operations for specific functionality. For example, ScrollDecorator's ScrollTo operation lets other objects scroll the interface *if they know there happens to be a ScrollDecorator object in the interface.*

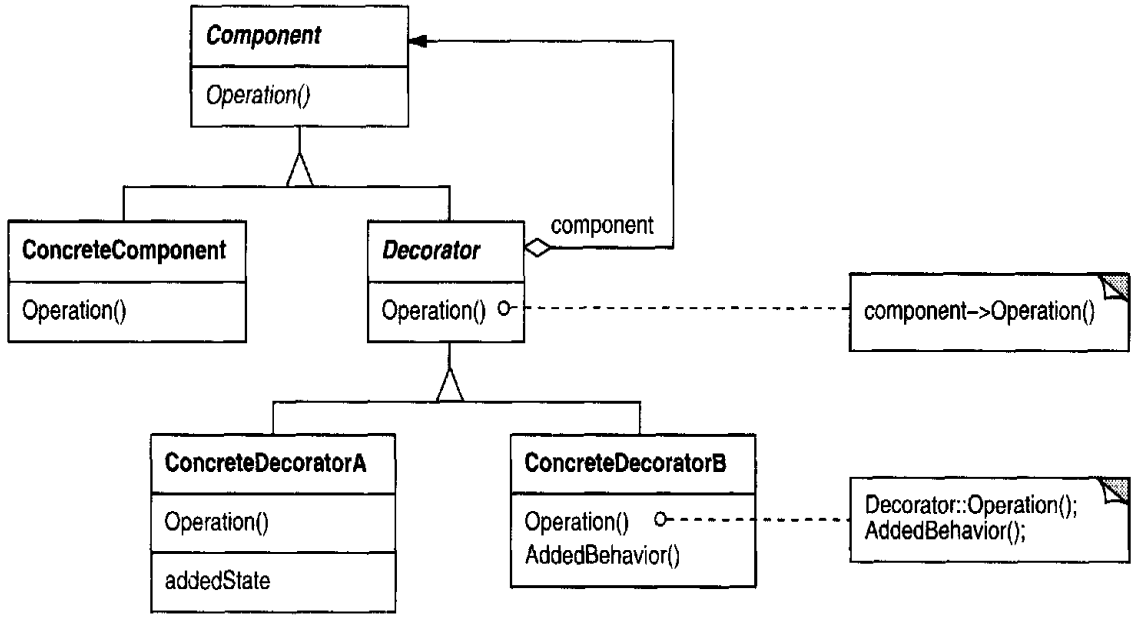


Applicability

Use Decorator

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

Structure



Participants

• Component (VisualComponent)

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

• ConcreteComponent (TextView)

- defines an object towhich additional responsibilities canbe 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 than can be had with static (multiple) inheritance. 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 (e.g., BorderedScrollableTextView, BorderedTextView). This gives rise to many classes and increases the complexity of a system. Furthermore, providing different Decorator classes for a specific Component class lets you mix and match responsibilities

2. *Avoids feature-laden classes high up in the hierarchy*. Decorator offers a payas-you-go approach to adding responsibilities. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects. Functionality can be composed from simple pieces. As a result, an application needn't pay for features it doesn't use. It's also 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 unrelated to the responsibilities you're adding.

3. *A decorator and its component aren't identical*. A decorator acts as a transparent enclosure. But from an object identity point of view, a decorated component is not identical to the component itself. Hence you 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. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug.

Implementation

Several issues should be considered when applying the Decorator pattern:

1. *Interface conformance*. A decorator object's interface must conform to the interface of the component it decorates. ConcreteDecorator classes must therefore inherit from a common class (at least in C++).

2. *Omitting the abstract Decorator class*. There's no need to define an abstract Decorator class when you only need to add one responsibility. That's often the case when you're dealing with an existing class hierarchy rather than designing a new one. In that case, you can merge Decorator's responsibility for forwarding requests to the component into the ConcreteDecorator.

3. *Keeping Component classes lightweight*. To ensure a conforming interface, components and decorators must descend from a common Component class. It's important to keep this common class lightweight; that is, it should focus on defining an interface, not on storing data. The definition of the data representation should be deferred to subclasses; otherwise the complexity of the Component class might make the decorators too heavyweight to use in quantity. Putting a lot of functionality into Component also increases the probability that concrete subclasses will pay for features they don't need.

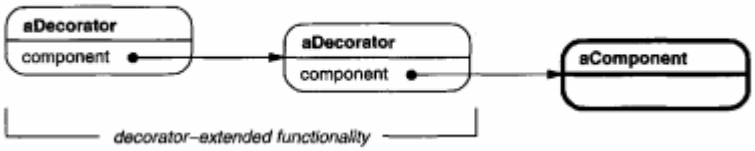
4. *Changing the skin of an object versus changing its guts*. We can think of a decorator as a skin over an object that changes its behavior. An alternative is to change the object's guts. The Strategy pattern is a good example of a pattern for changing the guts.

For example, we can support different border styles by having the component defer border-drawing to a separate Border object. The Border object is a Strategy object that encapsulates a border-drawing strategy. By extending the number of strategies from just one to an open-ended list, we achieve the same effect as nesting decorator recursively.

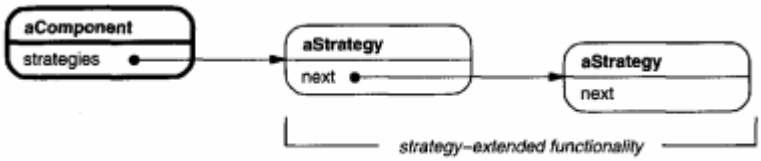
In MacApp 3.0 and Bedrock, for example, graphical components (called "views") maintain a list of "adorner" objects that can attach additional adornments like borders to a view

component. If a view has any adorners attached, then it gives them a chance to draw additional embellishments. MacApp and Bedrock must use this approach because the Viewclass is heavyweight. It would be too expensive to use a full-fledged View just to add a border.

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:



Known Uses

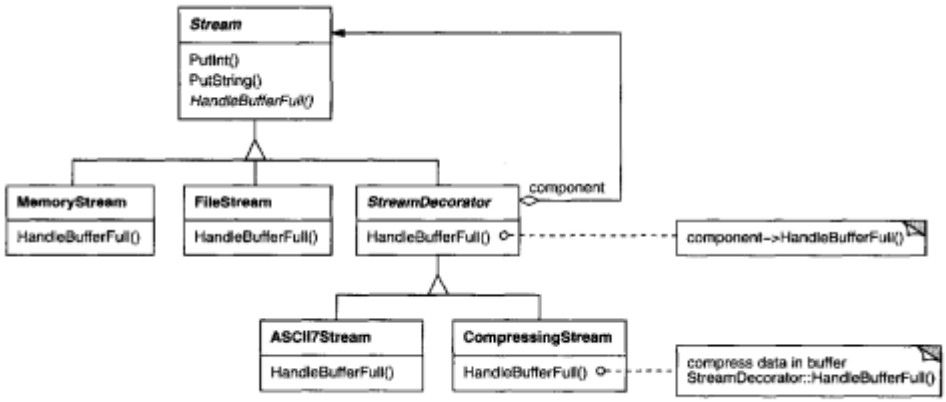
Many object-oriented user interface toolkits use decorators to add graphical embellishments to widgets. Examples include Interviews [LVC89, LCI+92], ET++ [WGM88], and the Object Works\Smalltalkclass library [Par90]. More exotic applications of Decorator are the DebuggingGlyph from Interviews and the PassivityWrapper from ParcPlace Smalltalk.

Streams are a fundamentalabstractionin most I/O facilities. Astream can provide an interface for converting objects into a sequence of bytes or characters. That lets us transcribe an object to a file or to a string in memory for retrieval later. A straightforward way to do this is to define an abstractStream class with subclasses MemoryStream and FileStream. But suppose we also want to be able to do the following:

• Compress the stream data using different compression algorithms (runlength encoding, Lempel-Ziv, etc.).

• Reduce the stream data to 7-bit ASCII characters so that it can be transmitted over an ASCII communication channel.

The Decorator pattern gives us an elegant way to add these responsibilities to streams. The diagram below shows one solution to the problem:



For example, the CompressingStream subclass compresses the data, and the ASCII7Stream converts the data into 7-bit ASCII. Now, to create a FileStream that compresses its data and converts the compressed binary data to 7-bitASCII, we decorate a FileStream with a CompressingStream and an ASCII7Stream:

```
Stream* aStream = new CompressingStream(
    new ASCII7Stream(
         new FileStream("aFileName")
    )
);
aStream->PutInt(12);
aStream->PutString("aString");
```

Related Patterns

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

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

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

FACADE

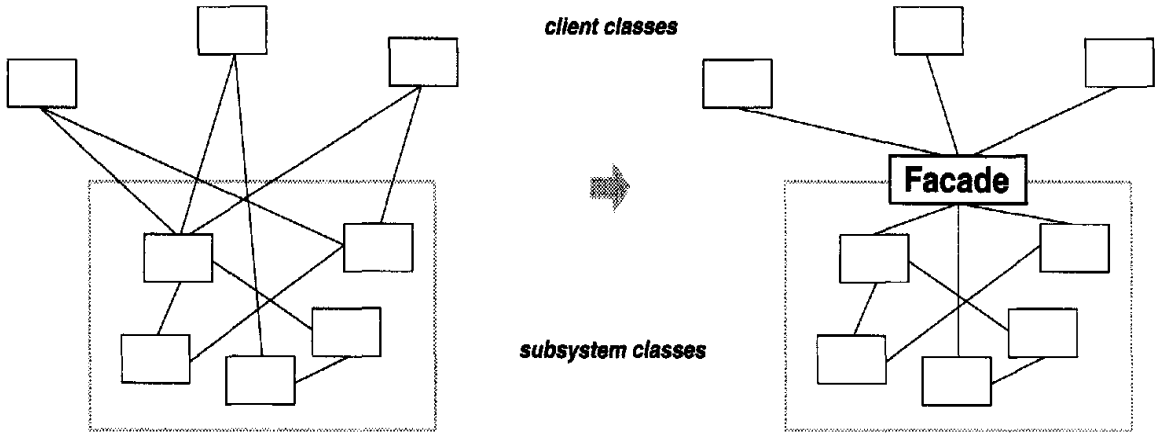
Intent

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

Motivation

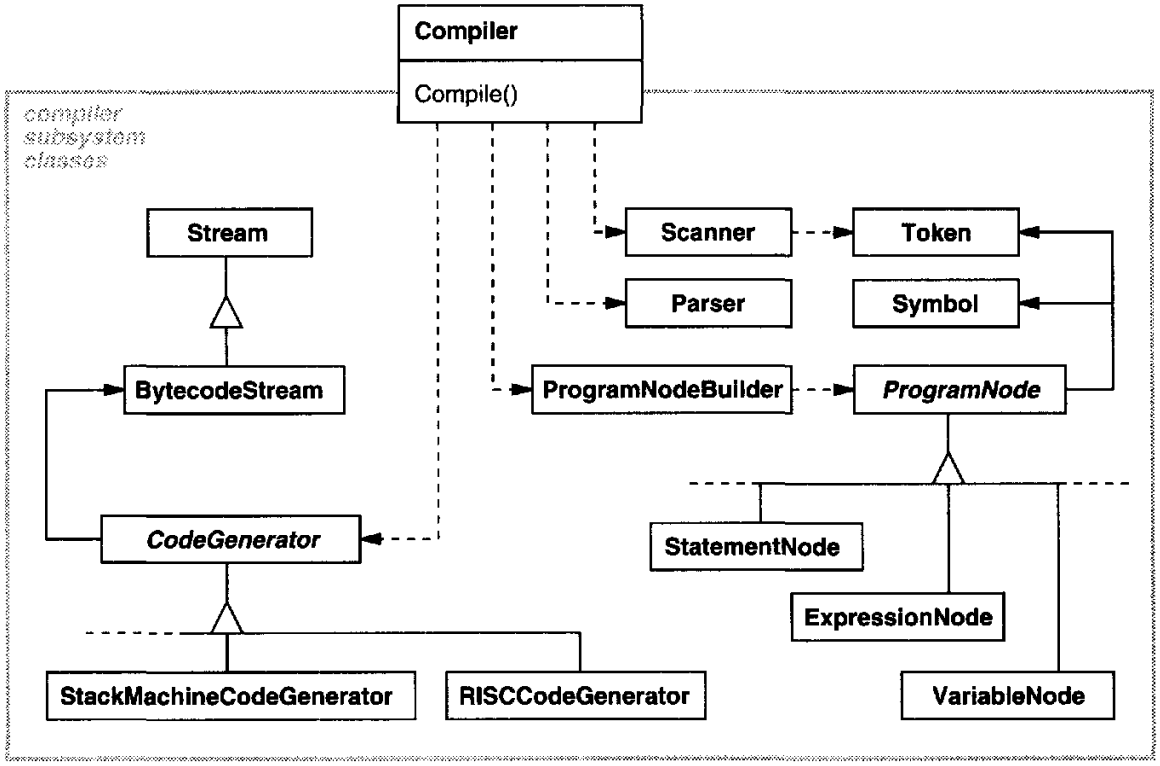
Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems.

One way to achieve this go al is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.



This subsystem contains classes such as Scanner, Parser, ProgramNode, BytecodeStream, and Program NodeBuilder that implement the compiler. To provide a higher-level interface that can shield clients from these classes, the compiler subsystem also includes a Compiler class. This class defines a unified interface to the compiler's functionality.

The Compiler class acts as a facade: It offers clients a single, simple interface to the compiler subsystem

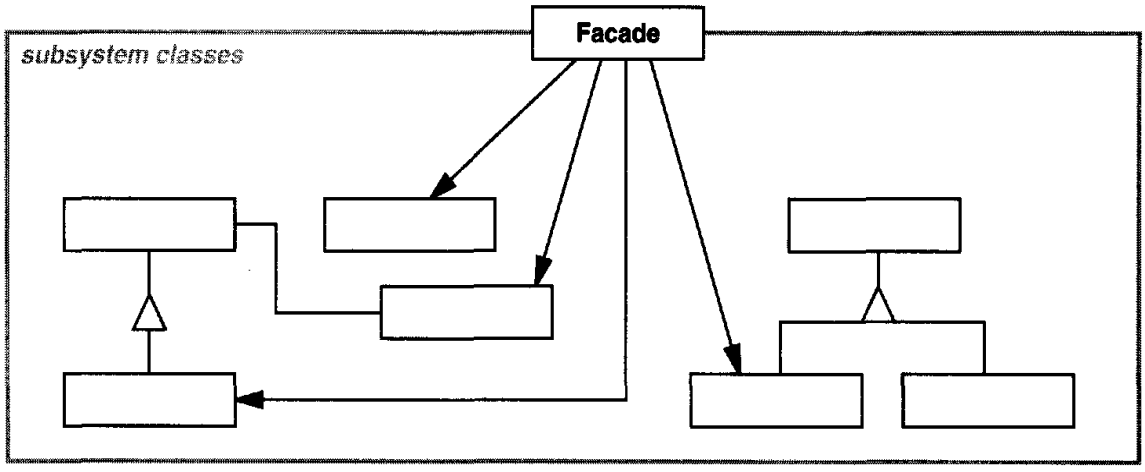


Applicability

Use the Façade pattern when

- you want to provide a simple interface to a complex subsystem
- there are many dependencies between clients and the implementation classes of an abstraction.
- you want to layer your subsystems.

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. Facades help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies. This can be an important consequence when the client and the subsystem are implemented independently.

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

Implementation

Consider the following issues when implementing a facade:

1. Reducing client-subsystem coupling. The coupling between clients and the subsystem can be reduced even further by making 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 in that both have interfaces, and both encapsulate something—a class encapsulates state and operations, while a subsystem encapsulates classes. And just as it's useful to think of the public and private interface of a class, we can think of the public and private interface of a subsystem.

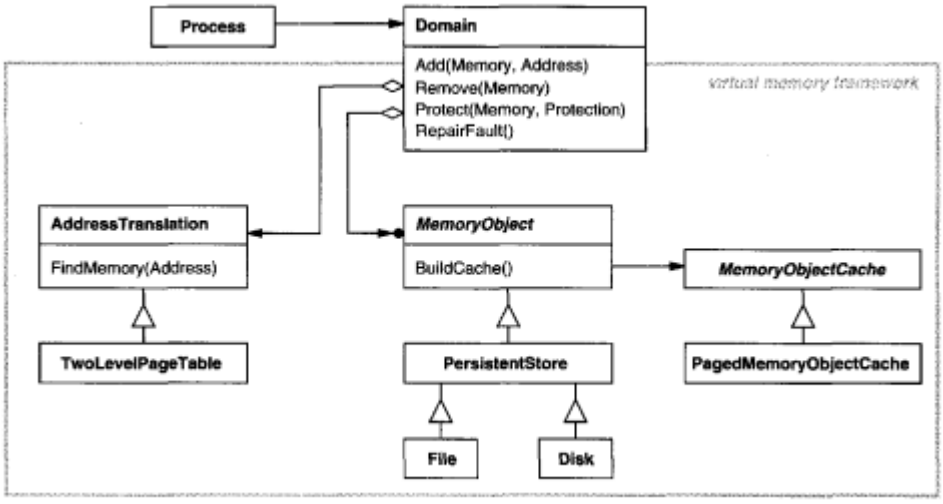
The public interface to a subsystem consists of classes that all clients can access; the private interface is just for subsystem extenders. For example, the classes Parser and Scanner in the compiler subsystem are part of the public interface

Known Uses

The compiler example in the Sample Code section was inspired by the Object Works\Smalltalk compiler system

In the ET++ application framework [WGM88], an application can have built-in browsing tools for inspecting its objects at run-time. These browsing tools are implemented in a separate subsystem that includes a Facade class called "ProgrammingEnvironment." This facade defines operations such as InspectObject and InspectClass for accessing the browsers.

The Choices operating system [CIRM93] uses facades to compose many frameworks into one. The key abstractions in Choices are processes, storage, and address spaces. For each of these abstractions there is a corresponding subsystem, implemented as a framework, that supports porting Choices to a variety of different hardware platforms. Two of these subsystems have a "representative" (i.e., facade). These representatives are FileSystemlnterface (storage) and Domain (address spaces).



For example, the virtual memory framework has Domain as its facade. A Domain represents an address space. It provides a mapping between virtual addresses and offsets into memory objects, files, or backing store. The main operations on Domain support adding a memory object at a particular address, removing a memory object, and handling a page fault.

As the preceding diagram shows, the virtual memory subsystem uses the following components internally:

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

Related Patterns

Abstract Factory 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 is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them. A mediator's colleagues are aware of and communicate with the mediator instead of communicating with each other directly. In contrast, a facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality, and subsystem classes don't know about it.

Usually only one Facade object is required. Thus Facade objects are often Singletons.

FLYWEIGHT

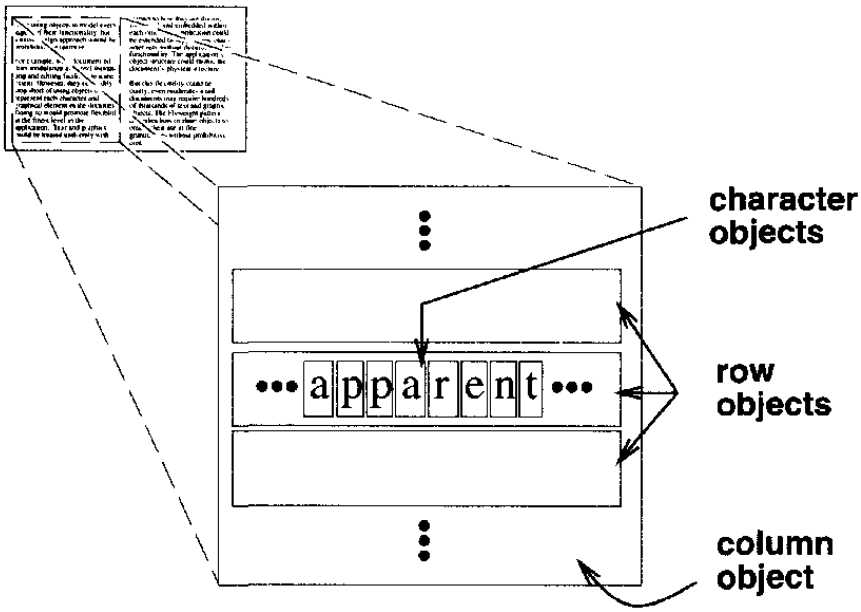
Intent

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

Motivation

Some applications could benefit from using objects throughout their design, but a naive implementation would be prohibitively expensive.

* Most document editor implementations have text formatting and editing facilities that are modularized to some extent. Object-oriented document editors typically use objects to represent embedded elements like tables and figures.



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.

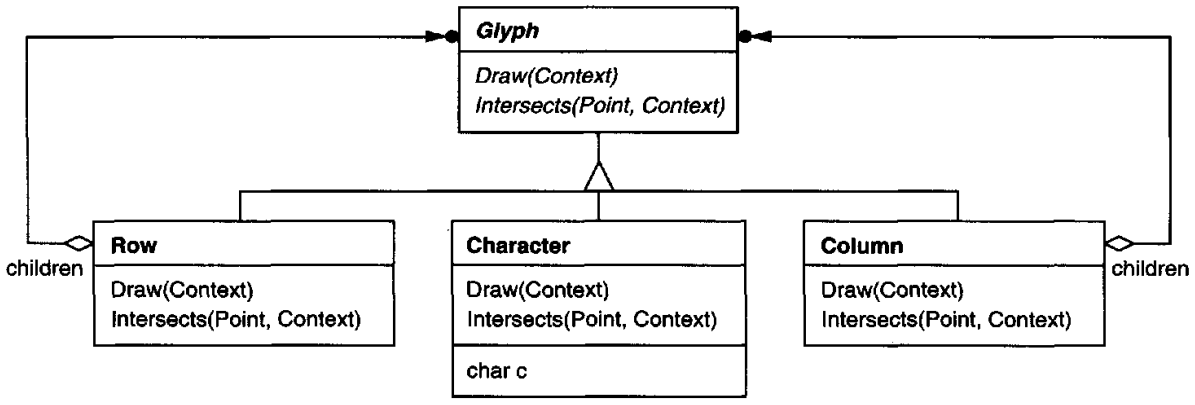
A flyweight is a shared object that can be used in multiple contexts simultaneously.

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 here is the distinction between **intrinsic and extrinsic state.**

Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable.

Extrinsic state depends on and varies with the flyweight‘s context and therefore can't be shared. Client objects are responsible for passing extrinsic state to the flyweight when it needs it.



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.

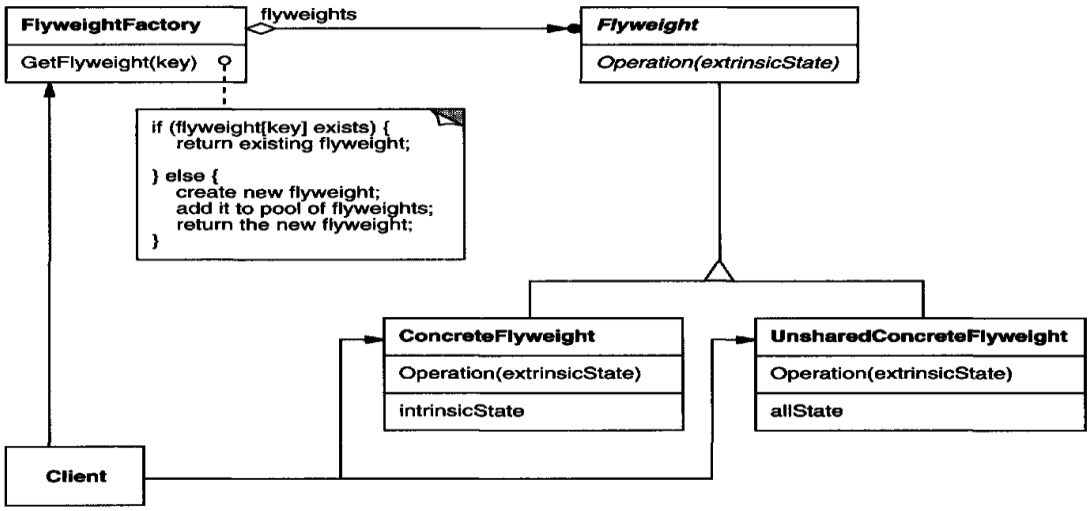
For example, a Row glyph knows where its children should draw themselves so that they are tiled horizontally.

Applicability

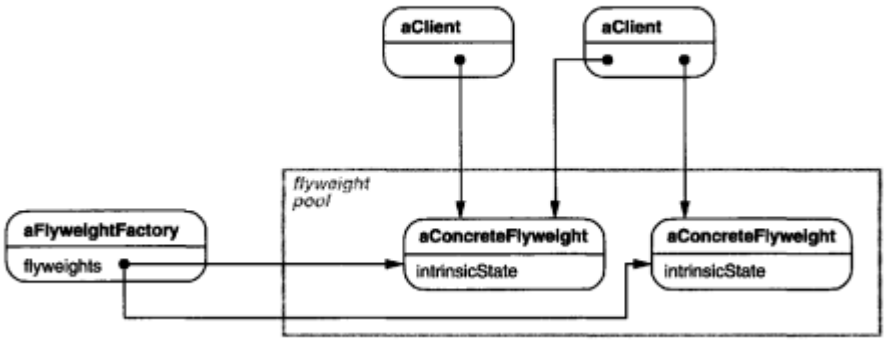
The Flyweight pattern's effectiveness depends heavily on how and where it's used.

- An application uses a large number of objects.
- Storage costs are high because of the rapid quantity of objects.
- Most object state can be made extrinsic
- The application doesn't depend on object identity

Structure



The following object diagram shows how flyweights are shared:



Participants

Flyweight (Glyph)

- declares an interface through which flyweights can receive and act on extrinsic state.

• ConcreteFlyweight (Character)

- implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the ConcreteFlyweight object's context.

• UnsharedConcreteFlyweight (Row, Column)

- not all Flyweight subclasses need to be shared. The Flyweight interface enables sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).

• FlyweightFactory

- creates and manages flyweight objects.

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

• Client

- maintains a reference to flyweight(s).

- computes or stores the extrinsic state of flyweight(s).

Collaborations

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

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

Consequences

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

• the reduction in the total number of instances that comes from sharing

• the amount of intrinsic state per object

• whether extrinsic state is computed or stored

Implementation

Consider the following issues when implementing the Flyweight pattern:

1. *Removing extrinsic state*. The pattern's applicability is determined largely by how easy it is to identify extrinsic state and remove it from shared objects. Removing extrinsic state won't help reduce storage costs if there are as many different kinds of extrinsic state as there are objects before sharing. Ideally, extrinsic state can be computed from a separate object structure, one with far smaller storage requirements.

In our document editor, 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. The map keeps track of runs of characters with the same typographic attributes. When a character draws itself, it receives its typographic attributes as a side-effect of the draw traversal. Because documents normally use just a few different fonts and styles, 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 example 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.

Known Uses

ET++ [WGM88] uses flyweights to support look-and-feel independence. The look-and-feel standard affects the layout of user interface elements (e.g., scroll bars, buttons, menus—known collectively as "widgets") and their decorations (e.g., shadows, beveling). 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.

The Layout objects are created and managed by Look objects. The Look class is an Abstract Factory that retrieves a specific Layout object with operations like GetButtonLayout, GetMenuBarLayout, and so forth. For each look-and-feel standard there is a corresponding Looksubclass (e.g., MotifLook, OpenLook)that supplies the appropriate Layoutobjects.

By the way, Layout objects are essentially strategies (see Strategy). They are an example of a strategy object implemented as a flyweight.

Related Patterns

The Flyweight pattern is often combined with the Composite pattern to implement a logically hierarchical structure in terms of a directed-acyclic graph with shared leaf nodes.

It's often best to implement State and Strategy objects as flyweights.

PROXY

Intent

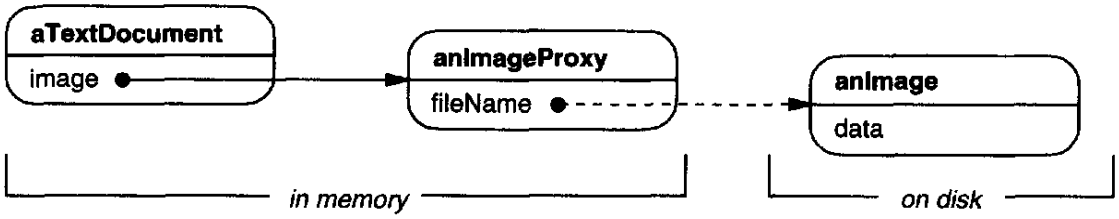
Provide a placeholder for another object to control access to it.

Also Known As

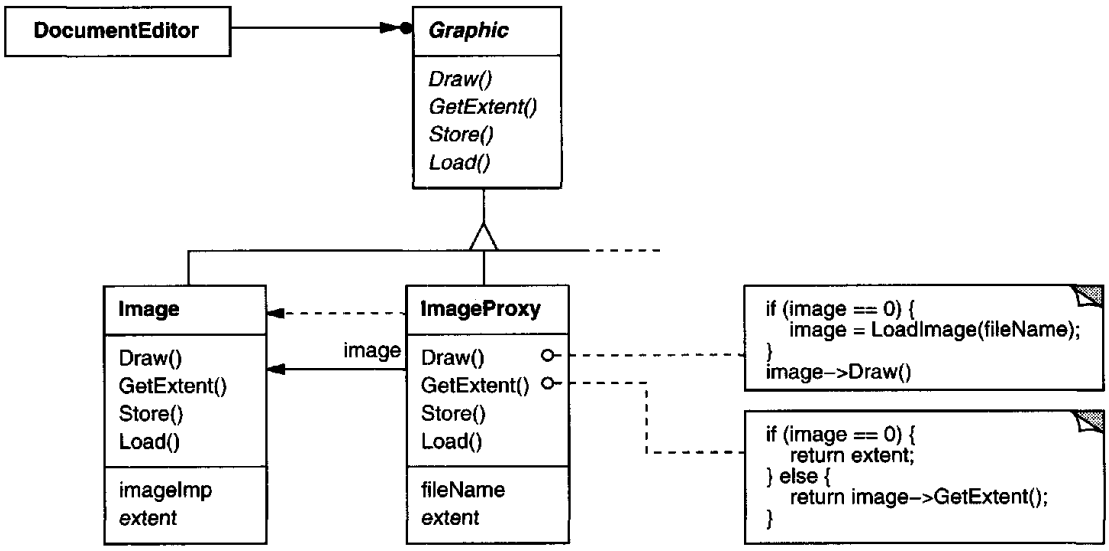
Surrogate

Motivation

Consider a document editor that can embed graphical objects in a document. Some graphical objects, like large raster images, can be expensive to create. But opening a document should be fast, so we should avoid creating all the expensive objects at once when the document is opened. This isn't necessary anyway, because not all of these objects will be visible in the document at the same time.



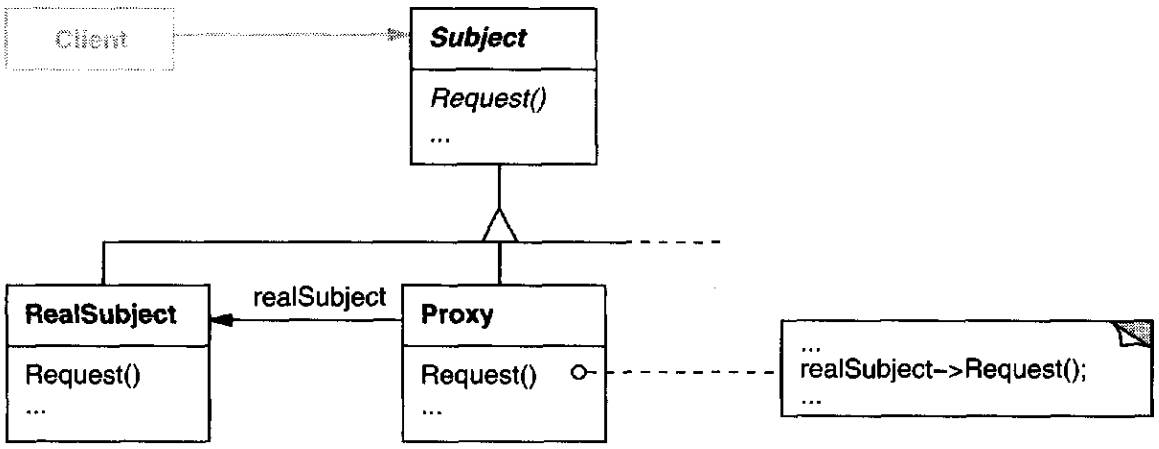
The image proxy creates the real image only when the document editor asks it to display itself by invoking its Draw operation. The proxy forwards subsequent requests directly to the image. It must therefore keep a reference to the image after creating it



Applicability

1. A remote proxy provides a local representative for an object in a different address space
2. A virtual proxy creates expensive objects on demand.
3. A protection proxy controls access to the original object
4. A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed

Structure



Participants

• Proxy (ImageProxy)

- maintains a reference that letsthe proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subjectinterfaces 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.

• Subject (Graphic)

- defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.

• RealSubject (Image)

- defines the real object that the proxy represents.

Collaborations

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

Consequences

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

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

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

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

There's another optimization that the Proxy pattern can hide from the client. It's called **copy-on-write**, and it's related to creation on demand. Copying a large and complicated object can be an expensive operation. 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.

Copy-on-write can reduce the cost of copying heavyweight subjects significantly.

Implementation

The Proxy pattern can exploit the following language features:

1. *Overloading the member access operator in C++.* C++ supports overloading operator->, the member access operator. Overloading this operator lets you perform additional work whenever an object is dereferenced. This can be helpful for implementing some kinds of proxy; the proxy behaves just like a pointer.

   The following example illustrates how to 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();
}
```

This approach lets you call Image operations through ImagePtr objects without going to the trouble of making the operations part of the ImagePtr interface:

```
ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
    // (image.operator->())->Draw(Point(50, 100))
```

Overloading the member access operator isn't a good solution for every kind of proxy.Some proxies need to know precisely which operation is called, and overloading the member access operator doesn't work in those cases.

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

3. *Proxy doesn't always have to know the type of real subject*. If a Proxy class can deal with its subject solely through an abstract interface, then there's no need to make a Proxy class for each RealSubject class; the proxy can deal with all RealSubjectclasses uniformly. But if

Proxies are going to instantiate RealSubjects (such as in a virtual proxy), then they have to know the concrete class.

Known Uses

The virtual proxy example in the Motivation section is from the ET++ text building block classes.

NEXTSTEP uses proxies (instances of class NXProxy) as local representatives for objects that may be distributed. A server creates proxies for remote objects when clients request them. On receiving a message, the proxy encodes it along with its arguments and then forwards the encoded message to the remote subject. Similarly, the subject encodes any return results and sends them back to the NXProxy object.

McCullough discusses using proxies in Smalltalk to access remote objects. Pascoe describes how to provide side-effects on method calls and access control with "Encapsulators."

Related Patterns

Adapter: An adapter provides a different interface to the object it adapts. In contrast, a proxy provides the same interface as its subject. However, a proxy used for access protection might refuse to perform an operation that the subject will perform, so its interface may be effectively a subset of the subject's.

Decorator: 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

Introduction

When we have a complex problem, we need a framework or structure within which to operate. For the problem of creating software systems, such a structure is provided by choosing software **architecture**. In this Module, we start by describing a well-known software architecture (sometimes referred to as an **architectural pattern**) called the **Model**–**View**– **Controller** or **MVC** pattern.

The MVC Architectural Pattern

- The model view controller is a relatively old pattern that was originally introduced in the Smalltalk programming language. As one might suspect, the pattern divides the application into three subsystems: model, view, and controller.
- The architecture is shown in Fig. 11.1. The pattern separates the application object or the data, which is termed the Model, from the manner in which it is rendered to the end-user (View) and from the way in which the end-user manipulates it (Controller).
- In contrast to a system where all of these three functionalities are lumped together (resulting in a low degree of cohesion), the MVC pattern helps produce highly cohesive modules with a low degree of coupling. This facilitates greater flexibility and reuse. MVC also provides a powerful way to organise systems that support multiple presentations of the same information.
- The model, which is a relatively passive object, stores the data. Any object can play the role of model. The view renders the model into a specified format, typically something that is suitable for interaction with the end user.
- For example, 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. The controller captures user input and when necessary, issues method calls on the model to modify the stored data. When the model changes, the view responds by appropriately modifying the display
- In a typical application, the model changes only when user input causes the controller to inform the model of the changes. The view must be notified when the model changes. Both the controller and the view communicate with the user through the UI.
- This means that some components of the UI are used by the controller to receive input; others are used by the view to appropriately display the model and some can serve both purposes (e.g., a panel can display a figure and also accept points as input through mouseclicks)

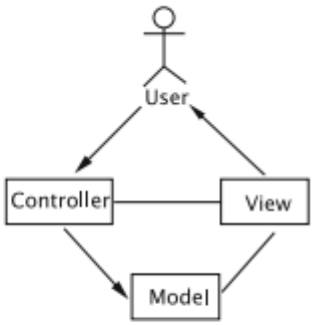


Fig. 11.1 The model–view–controller architecture

- When we talk of MVC in the abstract sense, we are dealing with the architecture of the system that lies behind the UI; both the view and the controller are subsystems at the same level of abstraction that employ components of the UI to accomplish their tasks.
- From a practical standpoint, however, we have a situation where the view and the UI are contained in a common subsystem. *For the purpose of designing our system, we shall refer to this common subsystem as the view.*
- The view subsystem is therefore responsible for all the look and feel issues, whether they arise from a human–computer interaction perspective (e.g., kinds of buttons being used) or from issues relating to how we render the model.
- Figure 11.2 shows how we might present the MVC architecture while accounting for these practical considerations. User-generated events may cause a controller to change the model, or view, or both.
- For example, suppose that the model stored the text that is being edited by the end- user. When the user deletes or adds text, the controller captures the changes and notifies the model. The view, which observes the model, then refreshes its display, with the result that the end-user sees the changes he/she made to the data. In this case, user-input caused a change to both the model and the view.

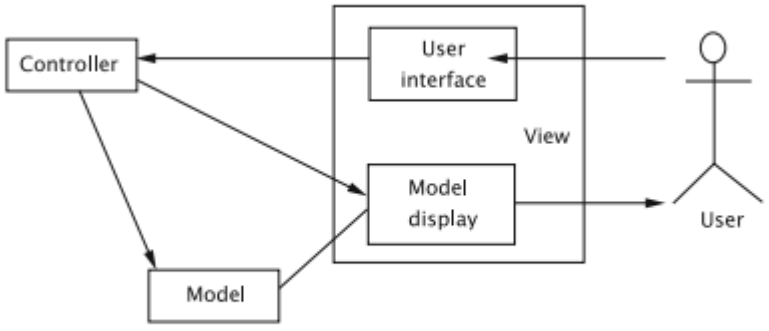


Fig. 11.2 An alternate view of the the MVC architecture

The view–model relationship is that of a subject–observer. The model, as the subject, maintains references to all of the views that are interested in observing it. Whenever an action that changes the model occurs, the model automatically notifies all of these

views. The views then refresh their displays. *The guiding principle here is that each view is a faithful rendering of the model.*

Examples

Suppose that in the library system we have a GUI screen using which users can place holds on books. Another GUI screen allows a library staff member to add copies of books. Suppose that a user views the number of copies, number of holds on a book and is about to place a hold on the book. At the same time, a library staff member views the book record and adds a copy. Information from the same model (book) is now displayed in different formats in the two screens.

Implementation

As with any software architecture, the designer needs to have a clear idea about how the responsibilities are to be shared between the subsystems. This task can be simplified if the role of each subsystem is clearly defined.

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

The definition for the model will be as follows:

```
public class Model extends Observable {
// code
public void changeData() {
// code to update data
setChanged();
notifyObservers(chan
geInfo);
}
}
```

.

The definition for the view will be as follows public class View implements Observer {

```
// code
public void update(Observable model, Object data) {
// refresh view using data
}
}
```

Benefits of the MVC Pattern

1. Cohesive modules: Instead of putting unrelated code (display and data) in the same module, we separate the functionality so that each module is cohesive.
2. Flexibility: The model is unaware of the exact nature of the view or controller it is

working with. It is simply an observable. This adds flexibility.
3. Low coupling: Modularity of the design improves the chances that components can be swapped in and out as the user or programmer desires. This also promotes parallel development, easier debugging, and maintenance.
4. Adaptable modules: Components can be changed with less interference to the rest of the system.
5. Distributed systems: Since the modules are separated, it is possible that the three subsystems are geographically separated.

Analyzing a Simple Drawing Program

We now apply the MVC architectural pattern to the process of designing a simple program that allows us to create and label figures. The purpose behind this exercise is twofold:

- *To demonstrate how to design with an architecture in mind*
- *To understand how the MVC architecture is employed*

Specifying the Requirements

Our initial wish-list calls for software that can do the following.

- Draw lines and circles.
- 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.
- Save the completed figure to a file. We can open a file containing a figure and edit it.
- Back track our drawing process by undoing recent operations.

Defining the Use Cases

We can now write the detailed use cases for each operation. The first one, for drawing a line, is shown in Table 11.1.

Table 11.1 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

Table 11.2 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

To give the system better usability, we allow for multiple labels to be added with the same command. To start the process of adding labels, the user clicks on the command button. This is followed by a mouse-click on the drawing panel, following which the user types in the desired label. After typing in a label, a user can either click on another point to create another label, or type a carriage return, which returns the system to the default state. These details are spelled out in the use case in Table 11.2.

The system will ignore almost all non-printable characters. The exceptions are the Enter (terminate the operation) and Backspace (delete the most-recently entered character) keys. A label may contain zero or more characters

We also have use cases for operations that do not change the displayed object. An example of this would be when the user changes the font, shown in Table 11.3.

Table 11.3 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

The requirements call for the ability to save the drawing and open and edit the saved drawings. The use cases for saving, closing and opening files are left as exercises. In order to allow for editing we need at least the following two basic operations: selection and deletion. The use case Select an Item is detailed in Table 11.4.

Table 11.4 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

Defining the Model

Our next step is to define what kind of an object we are creating. This is relatively simple for our problem; we keep a collection of line, circle, and label objects. Each line is represented by the end points, and each circle is represented by the *X*-coordinates of the leftmost and rightmost points and the *Y* -coordinates of the top and bottom points on the perimeter (see Fig. 11.3)

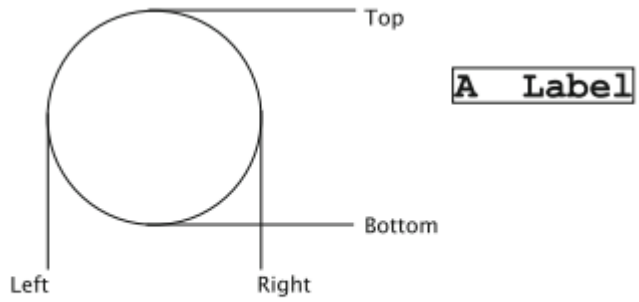


Fig. 11.3 Representing a circle and a label

Defining the Controller

The controller is the subsystem that orchestrates the whole show and the definition of its role is thus critical. When the user attempts to execute an operation, the input is received by the view. The view then communicates this to the controller. This communication can be effected by invoking the public methods of the controller.

Drawing a Line

- The user starts by clicking the Draw line button, and in response, the system changes the cursor. The click indicates that the user has initiated an operation that would change the model. Since such operations have to be orchestrated through the controller, it is appropriate that the controller be informed. The controller creates a line object (with both endpoints unspecified).

- The user clicks on the display panel to indicate the first end point of the line.

We now need to designate a listener for the mouse clicks. This listener will extract the coordinates from the event and take the necessary action. Both the view and the controller are aware of the fact that a line drawing operation has been initiated

- The user clicks on the second point. Once again, the view listens to the click and communicates this to the controller. On receiving these coordinates, the controller recognizes that the line drawing is complete and updates the line object.
- Finally, the model notifies the view that it has changed. The view then redraws the display panel to show the modified figure.

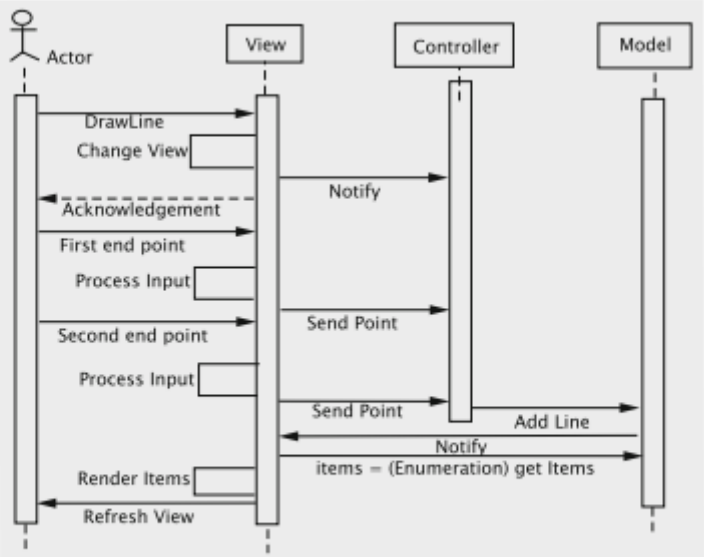


Fig. 11.4 Sequence of operations for drawing a line

This sequence of operations across the three subsystems can be captured by a highlevel sequence diagram as shown in Fig. 11.4.

Drawing a Circle

The actions for drawing a circle are similar. However, we now have some additional processing to be done, i.e., the given points on the diameter must be converted to the the four integer values, as explained in Fig. 11.3.

Adding a Label

This operation is somewhat different due to the fact that the amount of data is not fixed. The steps are as follows:

1. The user starts by clicking the Add Label button. In response, the system changes the mouse-cursor, which, as before is the responsibility of the view.

2. The user clicks the mouse, and the system acknowledges the receipt of the mouse click by placing a_ at the location.

3. The user types in a character. Once again, the view listens to and gets the input

from the keyboard, which is communicated to the controller. Once again the controller changes the model, which notifies the view.

4. The user clicks the mouse or enters a carriage-return. This is appropriately interpreted by the view. In both cases, the view informs the controller that the addition of the label is complete. In case of a mouse click, the controller is also notified that a new operation for adding a label has been initiated.

This sequence of steps is explained in Fig. 11.5. *Note that the view interprets the keystrokes: as per our specifications ordinary text is passed on directly to the controller, control characters are ignored; carriage-return is translated into a command, etc. All this is part of the way in which the system interacts with the user, and therefore belongs to the view.*

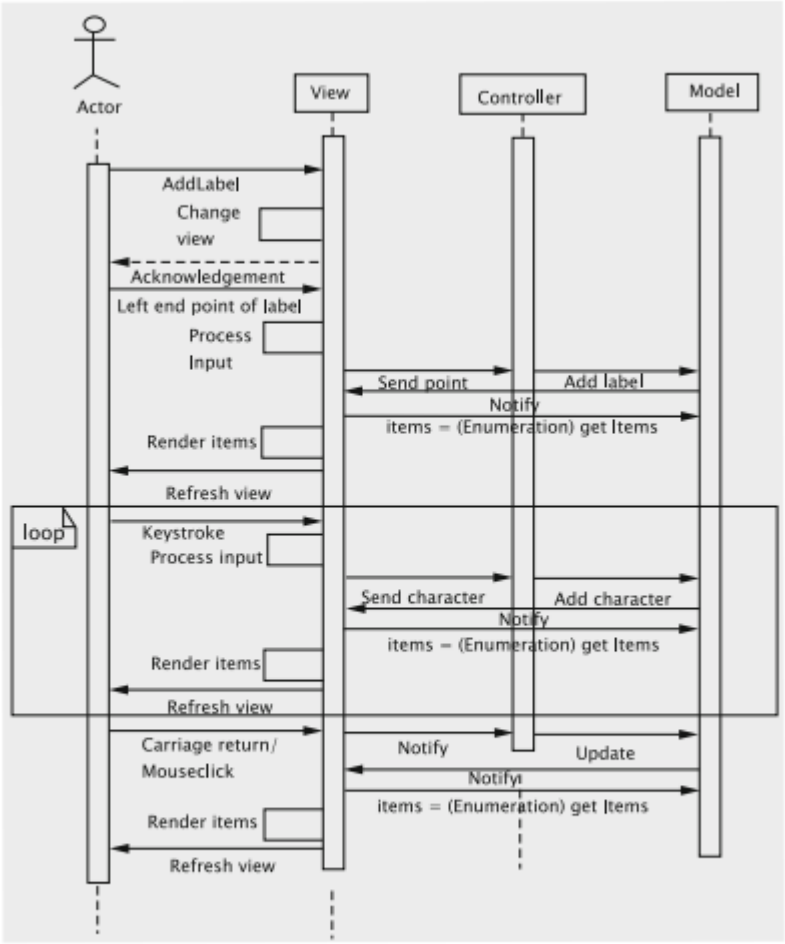


Fig. 11.5 Sequence of operations for adding a label

Selection and Deletion

- The software allows us to delete lines, circles, or labels by selecting the item and then invoking the delete operation. These shall be treated as independent operations since selection can also serve other purposes. Also, we can invoke selection repeatedly so that multiple items can be selected at any given time.

- When an item is selected, it is displayed in red, as opposed to black. The selection is done by clicking with the arrow (default) cursor. Lines are selected by clicking on one end point, circles are selected by clicking on the center, and labels are selected by clicking on the label.

The steps involved in implementing this are as follows:

- 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.
- In order to decide what action the controller must take, we need to figure out how the system will keep track of the selected items. Since the view is responsible for how these will be displayed (in red, for instance) the view must be able to recognize these as selected when updating the display.
- The next step is to iterate through the (unselected) items in the model to find the item (if any) that contains the point. Since the model is to be used strictly as a repository for the data, the task of iterating through the items is done in the controller, which then invokes the methods of the model to mark the item as selected
- Model notifies view, which renders the unselected items in the default color (black) and the selected items in red. View gets an enumeration of the two lists separately and uses the appropriate color for each.

Saving and Retrieving the Drawing

The use cases for the processes of saving and retrieving are simply described: *the user requests a save/retrieve operation, the system asks for a file name which the user provides and the system completes the task.* This activity can be partitioned between our subsystems as follows:

1. The view receives the initial request from the user and then prompts the user to input a file name.
2. The view then invokes the appropriate method of the controller, passing the file name as a parameter.
3. The controller first takes care of any clean-up operation that may be required. For instance, if our specifications require that all items be unselected before the drawing is saved, or some default values of environment variables be restored, this must be done at the stage. The controller then invokes the appropriate method in the model, passing the file name as a parameter.
4. The model serializes the relevant objects to the specified file.

Design of the Subsystems

In this stage, the classes and their responsibilities are identified and we get a more

detailed picture of how the required functionality is to be achieved.

Design of the Model Subsystem

We know that the model should have methods for supporting the following operations:

1. Adding an item
2. Removing an item
3. Marking an item as selected
4. Unselecting an item
5. Getting an enumeration of selected items
6. Getting an enumeration of unselected items
7. Deleting selected items
8. Saving the drawing
9. Retrieving the drawing

Based on the above list, it is straightforward to identify the methods. The class diagram is shown in Fig. 11.6. The class Item represents a shape such as line or label and enables uniform treatment of all shapes within a drawing.

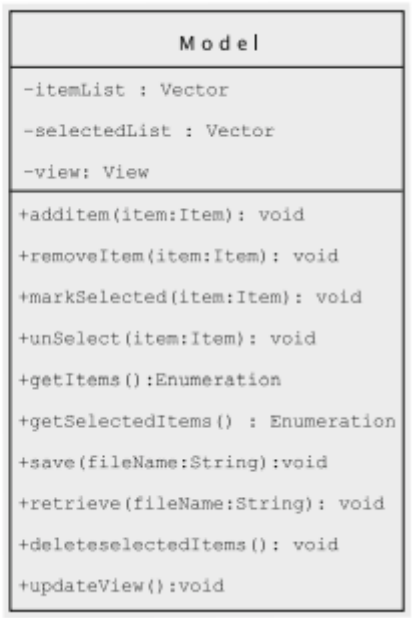


Fig. 11.6 Class diagram for model

- Since the methods, getItems() and getSelectedItems() return an enumeration of a set of items, we need polymorphic containers in the model. The view uses these methods to get the objects from the model as an enumeration of the items stored and draws each one on the display panel. The model must also keep track of the view, so it needs a field for that purpose.

- The method updateView is used by the controller to alert the model that the display must be refreshed. It is also invoked by methods within the model whenever the model realises that its data has changed. This method invokes a method in the view to refresh the display.

Design of Item and Its Subclasses

Rendering the items

Rendering is the process by which the data stored in the model is displayed by the view. Regardless of how we implement this, the actual details of how the drawing is done are dependent on the following two parameters:

- *The technology and tools that are used in creating the UI* For instance, we are using the Java's Swing package, which means that our drawing panel is a JPanel and the drawing methods will have to be invoked on the associated Graphics object.
- *The item that is stored* If a line is stored by its equation, the code for drawing it would be very different from the line that is stored as two end points.
- *The technology and tools are known to the author of the view, whereas the structure of the item is known to the author of the items.* Since the needed information is in two different classes, we need to decide which class will have the responsibility for implementing the rendering. We have the following options:

Option 1 Let us say that the view is responsible for rendering, i.e., there is code in the view that accesses the fields of each item and then draws them. Since the model is storing these items in a polymorphic container, the view would have to query the type of each item returned by the enumeration in order to choose the appropriate method(s).

Option 2 If the item were responsible, each item would have a render method that accesses the fields and draws the item. The problem with this is that the way an object is to be rendered often depends on the tools that we have at our disposal.

- At this point it appears that we are stuck between two bad choices! However, a closer look at the first option reveals a fairly serious problem: *we are querying each object in the collection to apply the right methods.* This is very much at odds with the object- oriented philosophy, i.e., *the methods should be packed with the data that is being queried.*
- This really means that the render method for each item should be stored in the item itself, which is in fact the approach of the second option. This simplifies our task somewhat, so we can focus on the task of fixing the shortcomings of the second option. The structure of the abstract Item class and its subclasses are shown in Fig. 11.7

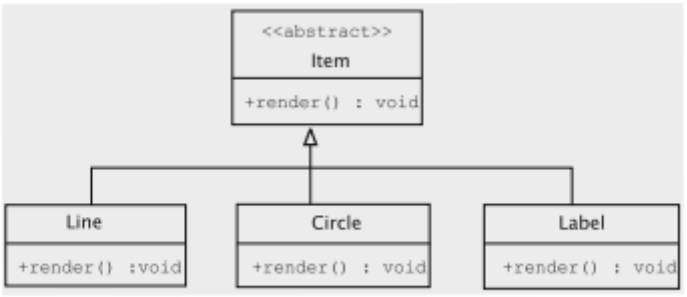


Fig. 11.7 The item class and its subclasses

Catering to Multiple UI Technologies

- Let us assume that we have available two new toolkits, which are called, for

want of better names, HardUI and EasyUI. Essentially, what we want is that each item has to be customised for each kind of UI, which boils down to the task of having a different render method for each UI. One way to accomplish this is to use inheritance.

- To adapt the design to take care of the new situation, we have the Circle class implement most of the functionality for circle, except those that depend on the UI technology. We extend Circle to implement the SwingCircle class. Similar extensions are now needed for handling the new technologies, HardUI and EasyUI. Each of the three classes has code to draw a circle using the appropriate UI technology. The idea is shown in Fig. 11.8
- In each case, the render method will decompose the circle into smaller components as needed, and invoke the methods available in the UI to render each component. For instance, with the Swing package, the render method would get the graphics object from the view and invoke the drawOval method. The code for this could look something like this:

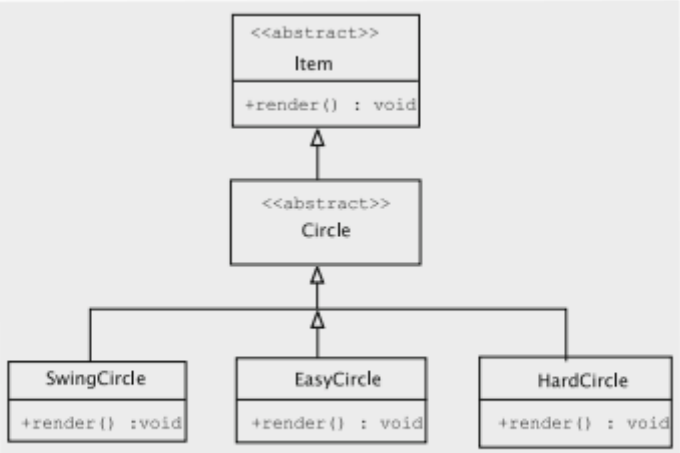


Fig. 11.8 Catering to multiple UI technologies

```
public class SwingCircle extends Circle {
  // circle class for SwingUI
  public void render() {
    Graphics g = (View.getInstance()).getGraphics();
    g.drawOval(/* parameters */);
  }
}
```

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

This solution has some drawbacks. The number of classes needed to accommodate such a solution is given by:

Number of types of items × Number of UI packages

As is evident from the pictorial view of the resulting hierarchy (see Fig. 11.9), this causes an unacceptable explosion in the number of classes.

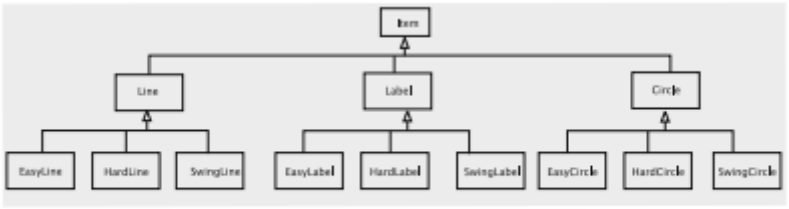


Fig. 11.9 Class explosion due to multiple UI implementations

- Since the Item subclasses are being created in the model, the types of items are an *internal variation*. On the other hand, the subclasses of Circle, Line, andLabel(such asHardCircle) areaan*external variation*.The standard approach for this is to factor out the external variations and keep them as a separate hierarchy, and then set up a *bridge* between the two hierarchies. This standard approach is therefore called the **bridge pattern.**

- The hierarchy of the UIs has an interface UIContext and as many concrete implementations as the number of different UIs we need. Figure 11.10 describes the interaction diagram between the classes and visually represents the bridge between the two hierarchies.

- Since the only variation introduced in the items due to the different UIs is the manner in which the items were drawn, this behaviour is captured in the UIContext interface as shown in Fig. 11.11

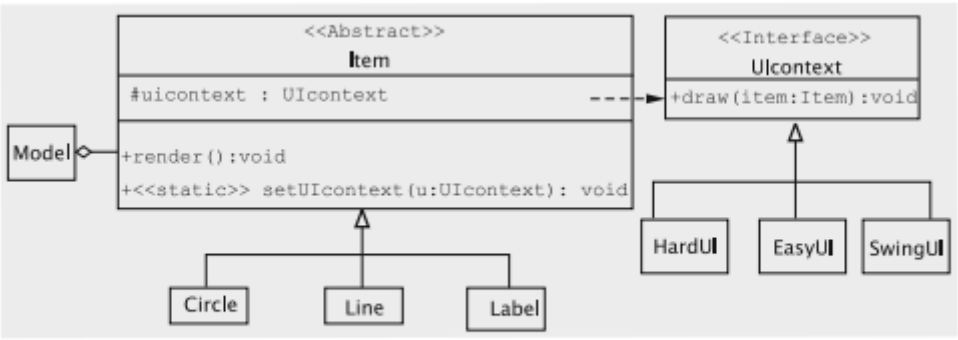


Fig. 11.10 Interaction diagram for the bridge pattern

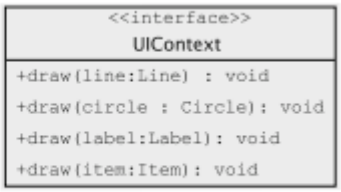


Fig. 11.11 UIContext interface

Note that the total number of classes is now reduced to

Number of types of items + Number of UI packages

- Since we have only one concrete class for each item, the creation process is simple. Finally, by factoring out the render method, we are no longer concerned with what kind of UI is being used to create the figure, or what UI will be used to edit it at a later stage. Our software for the model is thus 'completely' reusable.

Design of the Controller Subsystem

- We structure the controller so that it is not tied to a specific view and is unique to them 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

- When the view receives a button click to create a line, it calls the controller method makeLine. To reduce coupling between the controller and the view, we should allow the view to invoke this method at any time: before receiving any points, after receiving the first point, or after receiving both points.
- For this, the controller has three versions of the makeLine method and keeps track of the number of points independently of the view. The rest of the methods are for deleting selected items and for storing and retrieving the drawing and are fairly obvious. The class diagram is shown in Fig. 11.12

Fig. 11.12 Controller class diagram

Design of the View Subsystem

- The separation of concerns inherent in the MVC pattern makes the view largely independent of the other subsystems. Nonetheless, its design is affected by the controller and the model in two important ways:

1. Whenever the model changes, the view must refresh the display, for which the view must provide a mechanism.
2. The view employs a specific technology for constructing the UI. The corresponding implementation of UIContext must be made available to Item.

- The first requirement is easily met by making the view implement the Observer interface; the update method in the View class, shown in the class diagram in Fig. 11.13, can be invoked for this purpose.

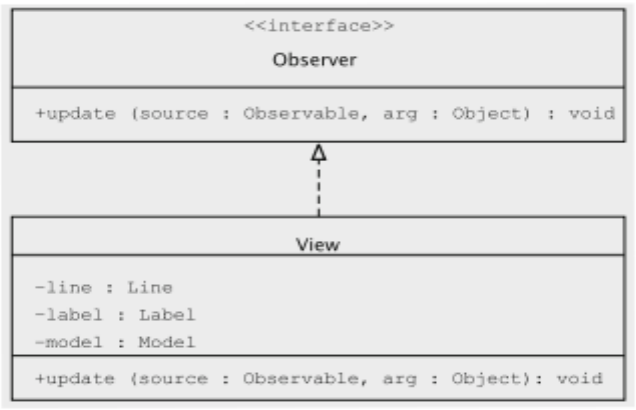


Fig. 11.13 Basic structure of the view class

- Commands to create labels, circles, and lines all require mouse listeners. Since

the behaviour of the mouse listener is dependent on the command, we know from previous examples in the book that a truly object-oriented design warrants a separate class for capturing the mouse clicks for each command. Since there is a one-to-one correspondence between the mouse listeners and the drawing commands, we have the following structure:

1. For each drawing command, we create a separate class that extends JButton. For creating labels, for instance, we have a class called LabelButton. Every button is its own listener.

2. For each class in (1) above, we create a mouse listener. These listeners invoke methods in the controller to initiate operations.

3. Each mouse listener (in (2) above) is declared as an inner class of the corresponding button class. This is because the different mouse listeners are independent and need not be known to each other.

The idea is captured in Fig. 11.14. The class MouseHandler extends the Java class MouseAdapter and is responsible for keeping track of mouse movements and clicks and invoking the appropriate controller methods to set up the label.

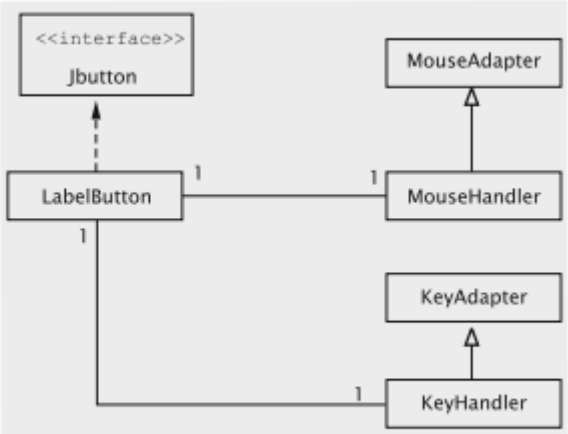


Fig. 11.14 Organisation of the classes to add labels

- If the user abandons a particular drawing operation, we could be in a tricky situation where there is more than one MouseHandler object receiving mouse clicks and performing conflicting operations such as one object attempting to create a line and another trying to add a label. To prevent this, we have two mechanisms in place.
    1. The KeyAdapter class also implements FocusListener to know when key strokes cease to be directed to this class.

    2. The drawing panel ensures that there is at most one listener listening to mouse clicks, key strokes, etc. This is accomplished by overriding methods such as addMouseListener and addKeyListener.

Getting into the Implementation

Item and Its Subclasses

This class Item is abstract and its implementation is as follows:

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

  protected double distance(Point point1, Point point2) {
    double xDifference = point1.getX() - point2.getX();
    double yDifference = point1.getY() - point2.getY();
    return ((double) (Math.sqrt(xDifference * xDifference +
                    yDifference * yDifference)));
  }
  public void render() {
    uiContext.draw(this);
  }
}
```

The includes method is used to check if a given point selects the item.

The Line class looks something like this:

```
public class Line extends Item {
  private Point point1;
  private Point point2;
  public Line(Point point1, Point point2) {

    this.point1 = point1;
    this.point2 = point2;
  }
  public Line(Point point1) {
    this.point1 = point1;
  }
  public Line() {
  }
  public boolean includes(Point point) {
    return ((distance(point, point1 ) < 10.0) || (distance(point, point2)
              < 10.0));
  }
  public void render() {
    uiContext.draw(this);
  }
  // setters and getters for the two points
}
```

Implementation of the Model Class

The class maintains itemList and selectedList, which respectively store the items created but not selected, and the items selected. The constructor initialises these containers.

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

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

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

.

Implementation of the Controller Class

The class must keep track of the current shape being created, and this is accomplished by having the following fields within the class.

```
private Line line;
private Label label;
```

When the view receives a button click to create a line, it calls one of the following controller methods. The controller supplies three versions of the makeLine method and keeps track of the number of points independently of the view.

```
public void makeLine() {
  makeLine(null, null);
  pointCount = 0;
}
public void makeLine(Point point) {
  makeLine(point, null);
  pointCount = 1;
}
public void makeLine(Point point1, Point point2) {
  line = new Line(point1, point2);
  pointCount = 2;
  model.addItem(line);
}
```

Implementation of the View Class

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

```
public class View extends JFrame implements Observer {
  private JPanel drawingPanel;
  private JPanel buttonPanel;
  // JButton references for buttons such as draw line, delete, etc.
  private class DrawingPanel extends JPanel {
    // code to redraw the drawing and manage the listeners
  }
  public View() {
    // code to create the buttons and panels and put them in the JFrame
  }
  public void update(Observable model, Object dummy) {
    drawingPanel.repaint();
  }
}
```

The code to set up the panels and buttons is quite straightforward, so we do not dwell upon that.

The DrawingPanel class overrides the paintComponent method, which is called by the system whenever the screen is to be updated. The method displays all unselected items by first obtaining an enumeration of unselected items from the model and calling the render method on each. Then it changes the colour to red and draws the selected items.

```
public void paintComponent(Graphics g) {
  model.setUI(NewSwingUI.getInstance());
  super.paintComponent(g);
  (NewSwingUI.getInstance()).setGraphics(g);
  g.setColor(Color.BLUE);
  Enumeration enumeration = model.getItems();
  while (enumeration.hasMoreElements()) {
    ((Item) enumeration.nextElement()).render();
  }
  g.setColor(Color.RED);
  enumeration = model.getSelectedItems();
  while (enumeration.hasMoreElements()) {
    ((Item) enumeration.nextElement()).render();
  }
```

The Driver Program

The driver program sets up the model. In our implementation the controller is independent of the UI technology, so it can work with any view. The view itself uses the Swing package and is an observer of the model.

```
public class DrawingProgram {
  public static void main(String[] args){
    Model model = new Model();
    Controller.setModel(model);
    Controller controller = new Controller();
    View.setController(controller);
    View.setModel(model);
    View view = new View();
    model.addObserver(view);
    view.show();
  }
}
```

Implementing the Undo Operation

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

- *Single-level undo versus multiple-level undo* A simple form of undo is when only one operation (i.e., the most recent one) can be undone. This is relatively easy, since we can afford to simply clone the model before each operation and restore the clone to undo.
- *Undo and redo are unlike the other operations* If an undo operation is treated the same as any other operation, then two successive undo operations cancel each other out, since the second undo reverses the effect of the first undo and is thus a redo. The undo (and redo) operations must therefore have a special status as meta-operations if several operations must be undone.
- *Not all things are undoable* This can happen for two reasons. Some operations like 'print file' are irreversible, and hence undoable. Other operations like 'save to disk' may not be worth the trouble to undo, due to the overheads involved.
- *Blocking further undo/redo operations* It is easy to see that uncontrolled undo and redo can result in meaningless requests. In general, it is safer to block redo whenever a new command is executed.
- *Solution should be efficient* This constraint rules out naive solutions like saving the model to disk after each operation.

Keeping these issues in mind, a simple scheme for implementing undo could be something like this:

1. Create a stack for storing the history of the operations.
2. For each operation, define a data class that will store the information necessary to undo the operation.
3. Implement code so that whenever any operation is carried out, the relevant information is packed into the associated data object and pushed onto the stack.
4. Implement an undo method in the controller that simply pops the stack, decodes the popped data object and invokes the appropriate method to extract the information and perform the task of undoing the operation.

One obvious approach for implementing this is to define a class StackObject that stores each object with an identifying String.

```
public class StackObject {
   private String name;
   private Object object;
   public StackObject(String string, Object object) {
     name = string;
     this.object = object;
   }
   public String getName() {
     return name;
   }
   public Object getObject() {
     return object;
   }
}
```

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

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

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

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


    }
    // other fields and methods
  }
```

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

```
public void undo() {
  StackObject undoObject = history.pop();
  String name = undoObject.getName();
  Object obj = undoObject.getObject();
  if (name.equals("line")) {
    undoLine((LineObject)obj);
  } else if (name.equals("delete")) {
    undoDelete((DeleteObject)obj);
  } else if (name.equals("select")) {
    undoSelect((SelectObject)obj);
  }
  // one else if for each command
```

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

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

There are two obvious drawbacks with this approach:

1. *The long conditional statement in the undo method of the controller.*
2. *The need to rewrite the controller whenever we make changes such as adding or modifying the implementation of an operation.*

Employing the Command Pattern

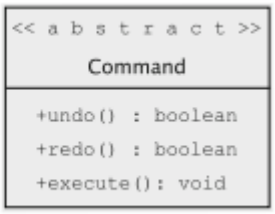
- In undo method, the controller passes itself as a reference to the undo method of the StackObject. In turn, each subclass of the StackObject (e.g., LineObject) passes itself as reference when invoking the appropriate undo method of the controller.
- This is an implementation of *double dispatch* that we used when employing the *visitor* pattern and was wholly appropriate when introducing new functionality into an existing hierarchy.
- In this context, however, we find that this results in unnecessarily moving a lot of data around. One of the lasting lessons of the object-oriented experience is the supremacy of data over process (The Law of Inversion), which we can utilise in this problem by using the **command pattern.**

The intent of the command pattern is as follows:

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

The command pattern provides us with a template to address this. The abstract Command class has abstract methods to execute, undo and redo. See Fig. 11.16

Fig. 11.16 The command class



The default undo and redo methods in Command return false, and these need to be overridden as needed by the concrete command classes.

The mechanism is best explained via an example, for which we develop a somewhat simplified sequence diagram for the command to add a line (Fig. 11.17).

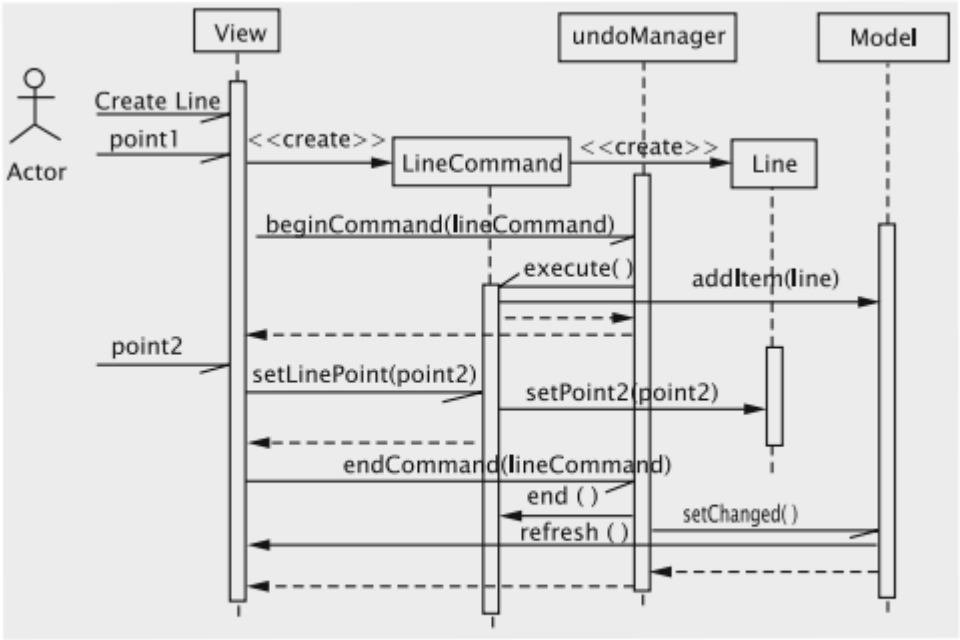


Fig. 11.17 Sequence diagram for adding a line

Adding a line Since every command is represented by a Command object, the first order of task when the Draw Line command is issued is to instantiate aLineCommand object. We assume that we do this after the user clicks the first endpoint although there is no reason why it could not have been created immediately after receiving the command. In its constructor, LineCommand creates a Line object with one of its endpoints specified

Assume that the user issues the sequence of commands:

Add Label (Label 1)
Draw Circle (Circle 1)

Add Label (Label 2)
Draw Line (Line 1)

At this time, there are four Command objects, one for each of the above commands, and they are on the history stack as in Fig. 11.18. The redo stack is empty: since no commands have been undone, there is nothing to redo. The picture also shows the collection object in the model storing the two Label objects, the Circle object, and the Line object.

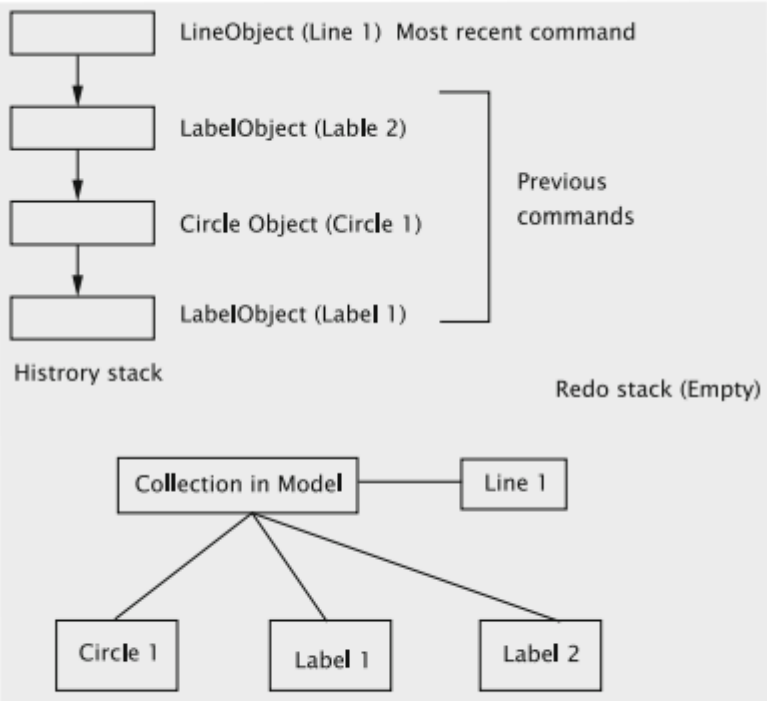


Fig. 11.18 Status of the stacks and the collection in the model

Undoing an operation

Continuing with the above example, we now look at the sequence of actions when the undo request is issued immediately after the line (Line 1) has been completely drawn in the above sequence of commands. Obviously, the user views the command as undone if the line disappears from the screen: for this, the Line object must be removed from the collection. To be consistent with this action and to allow redoing the operation, the LineCommand object must be popped from the history stack and pushed onto the redo stack. The resulting configuration is shown in Fig. 11.19

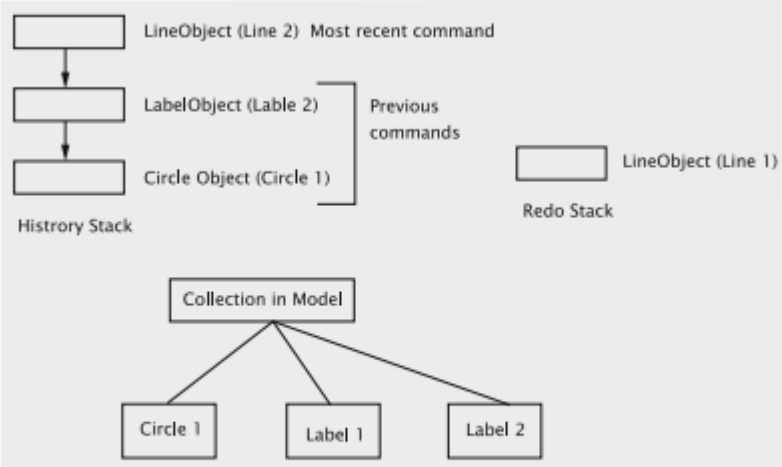


Fig. 11.19 Status of the stacks and the collection in the model after undo

Implementation

Subclasses of Command The concrete command classes (such as LineCommand) store the associated data needed to undo and redo these operations. Just as the makeLine method in the previous implementation had three versions, the LineCommand class has three constructors, allowing some flexibility in the design of the view

The implementation of methods specific to the Command class are shown below. The execute method simply adds the command to the model so the line will be drawn. To undo the command, the Line object is removed from the model's collection. Finally, redo calls execute

```
public void execute() {
  model.addItem(line);
}
public boolean undo() {


  model.removeItem(line);
  return true;
}
public boolean redo() {
  execute();
  return true;
}
```

As explained earlier, the class has a method called end, which attempts to complete an unfinished command. The situation is considered hopeless if both endpoints are missing, so the object removes the line from the model (undoes the command) and returns a false value. Otherwise, if the line is incomplete (has at least one endpoint unspecified), the start and end points are considered the same. The implementation is:

```
public boolean end() {
  if (line.getPoint1() == null) {
    undo();
    return false;
  }
  if (line.getPoint2() == null) {
     line.setPoint2(line.getPoint1());
  }
  return true;
}
```

UndoManager It declares two stacks for keeping track of the undo and redo operations: (history) and (redoStack). The current command is stored in a field aptly named currentCommand.

```
public class UndoManager {
  private Stack history;
  private Stack redoStack;
  private Command currentCommand;
}
```

Handling the input The view declares one button class for each command (add label, draw line, etc.). The class for handling line drawing is implemented as below.

```
public class LineButton  extends JButton implements ActionListener {
  // fields for view, drawing panel, handlers, etc.
  public LineButton(UndoManager undoManager, View jFrame, JPanel jPanel) {
    // store the parameters and create the mouse listener
  }
  public void actionPerformed(ActionEvent event) {
    // change the cursor
    drawingPanel.addMouseListener(mouseHandler);
  }
  private class MouseHandler extends MouseAdapter {
    public void mouseClicked(MouseEvent event) {
      if (first point) {
        lineCommand = new LineCommand(event.getPoint());
        UndoManager.instance().beginCommand(lineCommand);
        } else if (second point) {
        lineCommand.setLinePoint(event.getPoint());
        drawingPanel.removeMouseListener(this);
        view.setCursor(new Cursor(Cursor.DEFAULT_CURSOR));
        UndoManager.instance().endCommand(lineCommand);
      }
    }
  }
}
```

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. For instance, for a line, after the first click, the UI could track the mouse movement and draw a line between the first click point and the current mouse location; this line keeps shifting as the user moves the mouse. Likewise, if we were to extend the program to include triangles, which need three clicks, one side may be displayed after two clicks. Labels in construction must show the insertion point for the next character.

2. Some fields in an incomplete item might not have 'proper' values. Consequently, rendering an incomplete item could be more tricky. An incomplete line, for instance, might have one of the endpoints null. In such cases, it is inefficient to use the same render method for both incomplete items and complete items because that method will need to check whether the fields are valid and take appropriate actions to handle these special cases. Since we ensure that there is at most one incomplete item, this is not a sound approach.

We can easily distinguish between incomplete items and complete items by having a field that identifies the type. The render method will behave differently based on this field. The approach would be along the following lines.

```
public class Line {
  private boolean incomplete = true;
  public boolean isIncomplete() {
    return incomplete;
  }
  // other fields and methods
}

public class NewSwingUI implements UIContext {
  // fields and methods
  public void draw(Line line) {
    if (line.isIncomplete()) {
       draw incomplete line;
    } else {
       draw complete line;
    }
  }
}
```

We create classes for incomplete items (such as IncompleteLabel) that are subclasses of items (such as Label). Since the class IncompleteLabel is a subclass of Label, the model is unaware of its existence. Once the object is created, the incomplete object can be removed from the model.

The details are as follows.

```
import java.awt.*;
public class IncompleteLabel extends Label {
  public IncompleteLabel(Point point) {
    super(point);
  }
  public void render() {
    // code for rendering IncompleteLabel
  }
  public boolean includes(Point point) {
    return false;
  }
}
```

One problem we face with the above approach is that UIContext must include the method(s) for drawing the incomplete items (draw (IncompleteLabel label), in our example). This suggests that UIContext needs to be modified.

In general, we would like a solution that allows for a customised presentation which may require subclassing the behaviour of some concrete items. This can be accomplished through RTTI. In particular, the situation where the NewSwingUI wants its own method for drawing an incomplete line is implemented as follows:

```
public class NewSwingUI implements UIContext {
  // fields and methods
  public void draw(Line line) {
    if (line instanceof IncompleteLine) {
       this.draw((IncompleteLine) line);
    } else {
       //code to draw Line
    }
  }
}
```

Adding a New Feature

- Most interactive systems that are used to create graphical objects, allow users to define new kinds of objects on the fly. A system for writing sheet music may allow a user to define a sequence of notes as a group.

- 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.

- Let us examine how our system needs to be modified to accommodate this. The process for creating such a 'compound' object would be as follows: *The user would select the items that have to be combined by clicking on them. The system would then highlight the selected items. The user then requests the operation of combing the selected items into a compound object, and the system combines them into one.*

- 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 to create another compound object. The compound item is created by combining two compound items, and then decomposing it will give us back the two original compound items. Finally, the system must have the ability to undo and redo these operations.

- We have to store a collection of items to create a new kind of item that maintains a collection of the constituent items. This would be a concrete class and would look like this:

```
public class CompoundItem {
  List items;
  public CompoundItem(/* parameters */) {
    //instantiate lists
  }
  public Enumeration getItems() {
    //returns an enumeration of the objects in Items
  }
  // other fields and methods
}
```

- Since items consist of both simple items and compound items, it seems logical that all entities stored in items are designated as belonging to the class Object. The model would also have to be modified so that the container classes would hold collections of type Object
- Our standard approach in such situations is to create an inheritance hierarchy and use dynamic binding. The dilemma here is that we have a two fundamentally different kinds of entities: *a simple item is a single item, whereas a compound item is a collection of items.* The **composite pattern** gives us an elegant solution to this problem.

The intent of the composite pattern is as follows

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

A compound item is clearly a composition of simple items. Since each compound item

could itself consist of other compound items, we have the requisite tree structure (see Fig. 11.20).

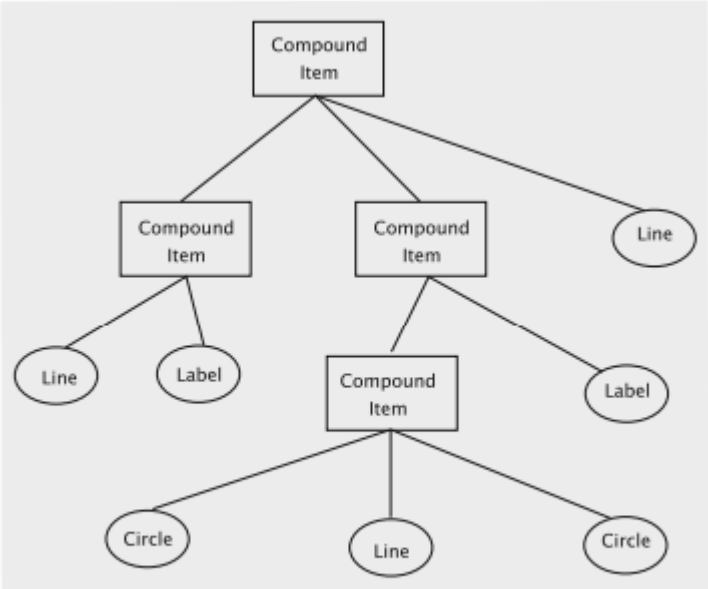


Fig. 11.20 Tree structure formed by compound items

The class interaction diagram for the composite pattern is shown in Fig. 11.21. Note that the definition of the compound item is recursive and may remind readers of the recursive definition of a tree. Following this diagram, the class CompoundItem is redefined as follows:

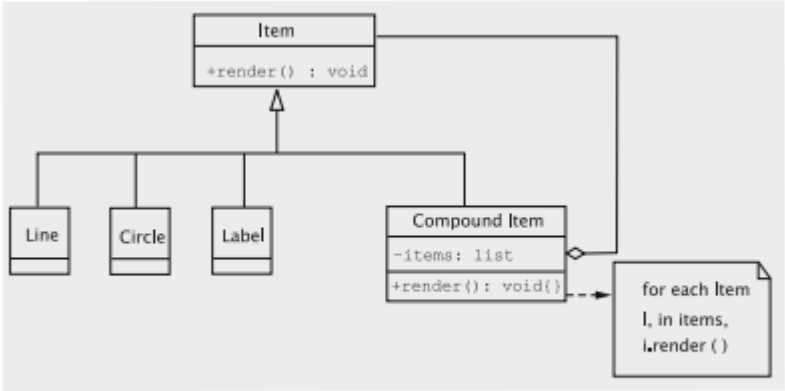


Fig. 11.21 Composite structure of the item hierarchy

```
public class CompoundItem extends Item {
  List items;
  public CompoundItem(/* parameters */){
    //instantiate lists
  }
  public void render(){
    // iterates through items and renders each one.
  }
```

```
public boolean includes(Point point) {
  /* iterates through items and invokes includes  on each item.
     Returns true if any of the items returns true and false otherwise. */
}
public void addItem(Item item) {
  // Adds item to items
}
  // other fields and methods
```

Pattern-Based Solutions

- A pattern is a solution template that addresses a recurring problem in specific situations. In a general , these could apply to any domain E.g.: A standard opening in chess, for instance, can be looked at as a 'chess pattern'. In the context of creating software, three kinds of patterns have been identified:

- At the highest level, we have the **architectural patterns.** These typically partition a system into subsystems and broadly define the role that each subsystem plays and how they all fit together. Architectural patterns have the following characteristics:

- *They have evolved over time* In the early years of software development, it was not very clear to the designers how systems should be laid out. Over time, some kind of categorisation emerged, of the kinds software systems that are needed. In due course, it became clearer as to how these systems and the demands on them change over their lifetime. This enabled practitioner to figure out what kind of layout could improve some of the commonly encountered problems.

- *A given pattern is usually applicable for a certain class of software system* The MVC pattern for instance, is well-suited for interactive systems, but might be a poor fit for designing a payroll program that prints paychecks.

- *The need for these is not obvious to the untrained eye* When a designer first encounters a new class of software; it is not very obvious what the architecture should be. The designer is not aware of how the requirements might change over time, or what kinds of modifications are likely to be needed. This is somewhat different from design patterns, which we are able to 'derive' by applying some of the well- established 'axioms' of object-oriented analysis and design

- 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.

- 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. As programmers, we often find ourselves using the same code snippet every time we have to accomplish a certain task.

- Idioms are something like these, but they are usually carefully designed to take the language features (and quirks!) into account to make sure that the code is safe and efficient. The following code, for instance, is commonly used to swap:

```
temp = a;
a = b;
b = temp;
```

- This is an example of an idiom for Perl. In addition to safety and efficiency, the familiarity of the code snippet makes the code more readable and reduces the need for comments. Not all idioms are without conflict. There are two possible idioms for an infinite loop:

```
for (;;) {
// some code
}
while (true) {
// some code
}
```

Module 5

Designing with Distributed Objects

Client/Server Systems

Distributed systems can be classified into peer-to-peer systems and client-server systems. 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. The latter, the client/server approach, is more popular in the commercial world. In client/server systems, there are two types of nodes: clients and servers.

We look at the implementation of object-oriented systems that use the client/server paradigm, which is the architecture itself

Basic Architecture of Client/Server Systems

We assume that although the client/server systems we build may have multiple clients, they will have just one server. It is not difficult to extend the techniques to multiple servers, so this is not a serious restriction. Figure 12.1 shows a system with one server and three clients. Each client runs a program that provides a user interface, which may or not be a GUI. The server hosts an object-oriented system. Like any other client/server system, clients send requests to the server, these requests are processed by the object-oriented system at the server, and the results are returned. The results are then shown to end-users via the user interface at the clients

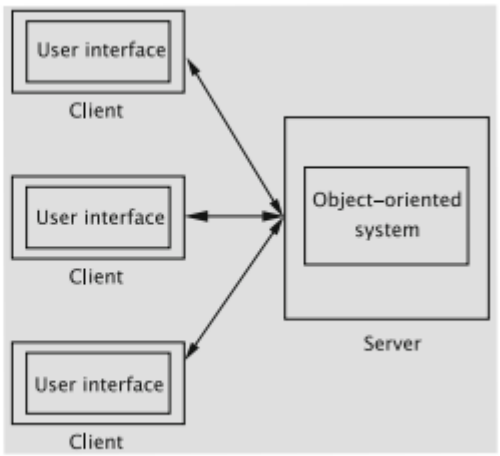


Fig. 12.1 Client/Server systems

There is a basic difficulty in accessing objects running in a different Java Virtual Machine (JVM). Let us consider two JVMs hosting objects as in Fig. 12.2. A single JVM has an address space part of which is allocated to objects living in it. For example, objects object 1 and object 2 are created in JVM 1 and are allocated at addresses A1 and A2 respectively. Similarly, objects object 3 and object 4 live in JVM 2 and are respectively allocated addresses A3 and A4. Code within Object 2 can access fields and methods in object 1 using address A1 (subject, of course, to access specifiers). However, addresses A3 and A4 that give the addresses of objects object 3 and object 4 in JVM 2 are meaningless within JVM 1. To see this, suppose A1 and A3 are equal. Then, accessing fields using address given by A3 from code within JVM 1

will end up accessing memory locations within object 1

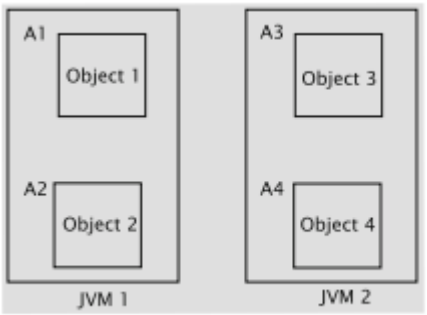


Fig. 12.2 Difficulty in accessing objects in a different JVM

This difficulty can be handled in one of two ways:

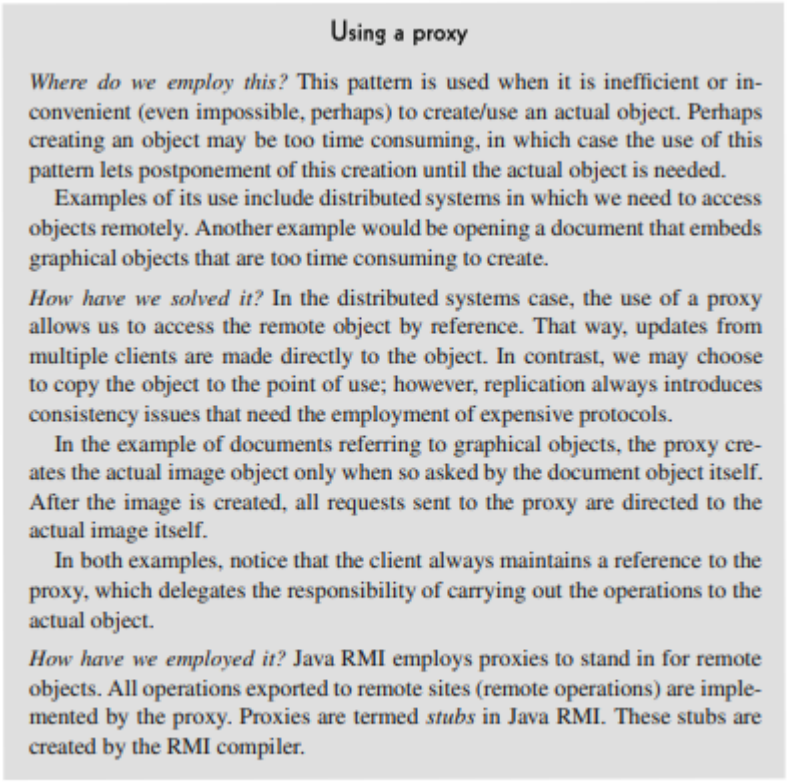
1. By using object-oriented support software: The software solves the problem by the use of proxies that receive method calls on 'remote' objects, ship these calls, and then collect and return the results to the object that invoked the call. The client could have a custom-built piece of software that interacts with the server software. This approach is the basis of Java Remote Method Invocation.

2. By avoiding direct use of remote objects by using the Hyper Text Transfer Protocol (HTTP). The system sends requests and collects responses via encoded text messages. The object(s) to be used to accomplish the task, the parameters, etc., are all transmitted via these messages. This approach has the client employ an Internet browser, which is, of course, a piece of general purpose software for accessing documents on the world-wide web. In this case, the client software is ignorant of the application structure and communicates to the server via text messages that include HTML code and data. This is the technique used for hosting a system on the Web.

Java Remote Method Invocation

The goal of Java RMI is to support the building of Client/Server systems where the server hosts an object-oriented system that the client can access programmatically. The objects at the server maintained for access by the client are termed **remote objects**. A client accesses a remote object by getting what is called a **remote reference** to the remote object. After that the client may invoke methods of the object.

The basic idea behind RMI is to employ the proxy design pattern. This pattern is used when it is inefficient or inconvenient (even impossible, perhaps) to use the actual object. (Refer to Fig. 12.3 for a description of the proxy pattern.)



Using a proxy

Where do we employ this? This pattern is used when it is inefficient or inconvenient (even impossible, perhaps) to create/use an actual object. Perhaps creating an object may be too time consuming, in which case the use of this pattern lets postponement of this creation until the actual object is needed.

Examples of its use include distributed systems in which we need to access objects remotely. Another example would be opening a document that embeds graphical objects that are too time consuming to create.

How have we solved it? In the distributed systems case, the use of a proxy allows us to access the remote object by reference. That way, updates from multiple clients are made directly to the object. In contrast, we may choose to copy the object to the point of use; however, replication always introduces consistency issues that need the employment of expensive protocols.

In the example of documents referring to graphical objects, the proxy creates the actual image object only when so asked by the document object itself. After the image is created, all requests sent to the proxy are directed to the actual image itself.

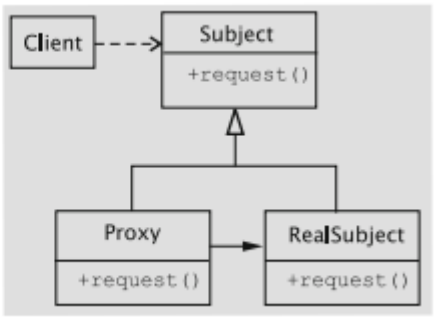
In both examples, notice that the client always maintains a reference to the proxy, which delegates the responsibility of carrying out the operations to the actual object.

How have we employed it? Java RMI employs proxies to stand in for remote objects. All operations exported to remote sites (remote operations) are implemented by the proxy. Proxies are termed *stubs* in Java RMI. These stubs are created by the RMI compiler.

Fig. 12.3 Using a proxy

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. The set up is shown in Fig. 12.4. 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. The actual call on the remote method of the remote object is made, and any return value is returned to the client via a message shipped from the server to the proxy object

Fig. 12.4 Client/Server systems



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.

Remote Interfaces

The first step in implementing a remote object system is to define the system functionality that will be exported to clients, which implies the creation of a Java interface. In the case of RMI, the functionality exported of a remote object is defined via what is called a ***remote interface***. A remote interface is a Java interface that extends the interface java.rmi.Remote, which contains no methods and hence simply serves as a marker. Clients are restricted to accessing methods defined in the remote interface. We call such method calls ***remote method invocations***

A remote interface must extend java.rmi.Remote and every method in it must declare to throw java.rmi.RemoteException. These concepts are shown in the following example.

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

Implementing a Remote Interface

The remote interfaces are defined; the next step is to implement them via **remote classes**. Parameters to and return values from a remote method may be of primitive type, of remote type, or of a local type.

All arguments to a remote object and all return values from a remote object must be serializable. Thus, in addition to the requirement that remote classes implement remote interfaces, we require that they also implement the java.io.Serializable interface.

Parameters of non-remote types are passed by copy; they are serialized using the object serialization mechanism, so they too must implement the Serializable interface.

Remote objects must somehow be capable of being transmitted over networks. A convenient way to accomplish this is to extend the class java.rmi.server.UnicastRemoteObject.

Thus, the implementation of BookInterface is as below

```
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;
    }
  }
```

Since it is a remote class, Book must be compiled using the RMI compiler by invoking the command rmic as below.

```
rmic Book
```

Remote objects are thus passed by reference. This is depicted in Fig. 12.5, where we have a single remote object that is being accessed from two clients. Both clients maintain a reference to a stub object that points to the remote object that has a field named a. Suppose now that Client 1 invokes the method setA with parameter 5.

As we have seen earlier, the call goes through the stub to the remote object and gets executed changing the field a to 5. The scheme has the consequence that any changes made to the state of the object by remote method invocations are reflected in the original remote object. If the second client now invokes the method getA, the updated value 5 is returned to it.

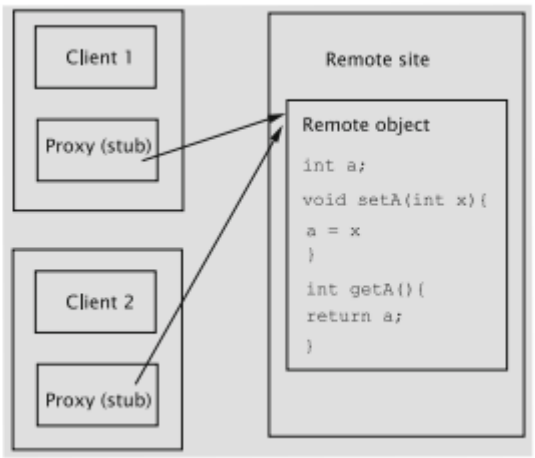


Fig. 12.5 Passing of remote objects as references

Creating the Server

Before a remote object can be accessed, it must be instantiated and stored in an object registry, so that clients can obtain its reference. Such a registry is provided in the form of the class java.rmi.Naming. The method bind is used to register an object and has the following signature:

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

The first argument takes the form //host:port/name and is the URL of the object to be registered; host refers to the machine (remote or local) where the registry is located, port is the port number on which the registry accepts calls, and name is a simple string for distinguishing the object from the other objects in the registry. Both host and port may be omitted in which case they default to the local host and the port number of 1099, respectively.

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);
    }
  }
}
```

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.

In the following 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");
```

After the above step, the client can invoke remote methods on the object. In the following code, the getters of 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();
    }
  }
}
```

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
```

While in the server directory. 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
```

The first command starts the registry and the second causes the Book instance to be created and registered with the name MyBook.

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

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

Implementing an Object-Oriented System on the Web

The world-wide web is the most popular medium for hosting distributed applications. Increasingly, people are using the web to book airline tickets, purchase a host of consumer goods, make hotel reservations, and so on. 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).

One major characteristic of a web-based application system is that the client (the browser), being a general-purpose program, typically does no application-related computation at all. 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. In this sense, 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. For example, when we make a reservation on a flight, we expect the system to display the details of the flight on which we made the reservation. This requires that HTML code for the screen be dynamically constructed.

This is done by code at the server. For server-side processing, there are competing technologies such as Java Server Pages and Java Servlets, Active Server Pages (ASP), and PHP. In this book we study Java Servlets.

HTML and Java Servlets

We have stated earlier, 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. We will now describe the essential code for doing these.

An HTML program can be thought of as containing a header, a body, and a trailer. The header contains code like the following:

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

The first four lines are usually written as given for any HTML file. We do not elaborate on these, but observe words such as html and head that are enclosed between angled brackets (< and >). They are called tags. HTML tags usually occur in pairs: start tag that begins an entry and end tag that signals the entry's end. For example, the tag begins the header and is ended by. The text between the start and end tags is the element content.

In the fifth line we see the tag title, which defines the string that is displayed in the title bar. The idea is that the string A Web Page will be displayed in the title bar of the browser when this page is displayed.

As a sample body, let us consider the following

```
<body>
<h1>
    <span style="color: rgb(0, 0, 255);">
    <span style="font-family: lucida bright;">
    <span style="font-style: italic;">
    <span style="font-weight: bold;">
       An Application
    </span>
    </span>
    </span>
    </span>
</h1>
</body>
```

The body contains code that determines what gets displayed in the browser's window. Some tags may have attributes, which provide additional information. For example, see the line

```
<span style="color: rgb(0, 0, 255);">
```

The body contains code to display the string An Application in the font Lucida bright, bolded, italicised, and in blue color.

The last line of the file is

```
</html>
```

Obviously, it ends the HTML file

Entering and Processing Data

Web pages that allow the user to enter information that the system processes. For example, a search engine provides a field in which we type in some search terms. When an accompanying button is clicked, the system transfers control to the search engine that displays results of the search.

This is accomplished by using what is called a form tag in HTML. The complete code that allows us to enter some piece of text in the web page is given below.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
 <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
 <title>Sample Form</title>
</head>
<body>
  <form action="/servlet/apackage.ProcessInput" method="post">
    <table>
       <tr>
           <td align="right">Enter Data:</td>
           <td><input type="text" name="userInput"></td>
       </tr>
       <tr>
           <td><input type="submit" value="Process"></td>
       </tr>
    </table>
  </form>
</body>
</html>
```

Let us get a general understanding of the above piece of code. Consider the code that begins with the line

```
<form action="/servlet/apackage.ProcessInput" method="post">
```

The tag form begins the specification of a set of elements that allow the user to enter information. The action attribute specifies that the information entered by the user is to be processed by a Java class called ProcessInput.class, which resides in the package apackage.

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. See Fig. 12.6 for the considerations in deciding which of these methods should be used.

The tag `<table>` begins the creation of a table. Each row of the table is described using the tag `<tr>`, and the tag `<td>` defines a cell in the table. The line

```
<td><input type="text" name="userInput"></td>
```

GET or POST?

While considering the question of which of the two methods, GET or POST, should be employed to transmit form data, it is helpful to remember that GET inserts the data in the URL itself whereas POST includes the data as part of the message. As a consequence, the URLs created for the POST and GET methods differ in that the latter completely identifies the server resource. This implies that the resource from the URL of the GET method can be used from other web pages to access the same resource, a capability that is not possible with the URL of POST.

A section in the HTTP/1.1 specifications talks about a kind of client/server interactions called **safe interactions**. In a safe interaction, users are not responsible for the result of the interaction, and GET is the appropriate method to use in such situations. To understand the concept of safe interactions, consider a web page (call it page 1) that asks the user to agree to some conditions by checking a box before allowing him/her to download a piece of software from a second page (say, page 2). Suppose that the form data, which includes the checkbox, from page 1 is transmitted using GET. Clearly, the URL completely identifies page 2. This URL can then be used to provide a link, called a deep link, to Page 2 from an unrelated web page (say, page 3), and any use of this link from page 3 is an unsecure way of accessing the resource.

It should be noted, however, that trying to hide the resource location is not a foolproof mechanism: one could look at the source file of the web page to craft a link to the resource.

One of the HTTP usage recommendations is that the GET method should be used only when the form processing is **idempotent**, that is, the result is the same whether the form is processed once or multiple times. This definition, however, should not be taken too literally. Generally speaking, if resubmitting a form does not change the application data stored at the server (even if it changes other entities such as log files), it is appropriate to use GET. In other circumstances, the POST method should be used to transmit form data.

Generally speaking, results from the GET method are cached, but data obtained from POST are not. As a consequence, GET method may execute faster than POST.

Fig. 12.6 Get and post: a brief comparison

The first line states that the data is HTML and the second line begins the HTML code. The complete code for the servlet is given below.

```
package apackage;
import javax.servlet.*;
import javax.servlet.http.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
public class ProcessInput extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response) throws IOException,
                     ServletException {
    String input = request.getParameter("userInput");
    response.setContentType("text/html");
    response.getWriter().println("<!DOCTYPE html PUBLIC \"-//W3C//DTD " +
                                 "HTML 4.01 Transitional//EN\">");
    response.getWriter().println("<html>");
    response.getWriter().println("<head>");
    response.getWriter().println("<meta content=\"text/html;" +
                                 " charset=ISO-8859-1\"" +
                                 "http-equiv=\"content-type\">");
    response.getWriter().println("<title>Response to Input</title>");
    response.getWriter().println("</head>");
    response.getWriter().println("<body>");
    response.getWriter().println("You entered " + input);
    response.getWriter().println("</body>");
    response.getWriter().println("</html>");
  }
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response) throws IOException,
                    ServletException {
    doPost(request, response);
  }
}
```

The architecture for serving web pages is depicted in Fig. 12.7. Assume that an 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.

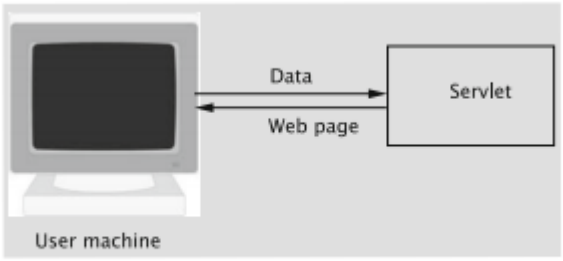


Fig. 12.7 How servlets and HTML cooperate to serve web pages

Deploying the Library System on the World-Wide Web

Developing User Requirements

As in any system, the first task is to determine the system requirements. We will, as has been the case throughout the book, restrict the functionality so that the system's size is manageable.

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 are essentially designated library employees, and ordinary members are the general public who borrow library books. The major difference between the two groups of users is that superusers can execute any command when logged in from a terminal in the library, whereas ordinary members cannot access some 'privileged commands'. In particular, the division is as follows:

   (a) Only superusers can issue the following commands: add a member, add a book, return a book, remove a book, process holds, save data to disk, and retrieve data from disk.

   (b) Ordinary members and superusers may invoke the following commands: issue and renew books, place and remove holds, and print transactions.

   (c) Every user eventually issues the exit command to terminate his/her session.

3. Some commands can be issued from the library only. These include all of the commands that only the superuser has access to and the command to issue books.

4. A superuser cannot issue any commands from outside of the library. They can log in, but the only command choice will be to exit the system.

5. Superusers have special user ids and corresponding password. For regular members, their library member id will be their user id and their phone number will be the password.

Interface requirements It turns out that due to the nature of the graphical user interface, an arbitrarily large number of sequences of interactions are possible between the user and the interface

Logging in and the Initial Menu

In Fig. 12.8, we show the process of logging in to the system. When the user types in the URL to access the library system, the log in screen that asks for the user id and password is displayed on the browser. If a valid combination is typed in, an appropriate menu is displayed. What is in the menu depends on whether the user is an ordinary member or a superuser and whether the terminal is in the library or is outside.

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

2. Commands to place a hold, remove a hold, print transactions, and renew books are available to members of the library (not superusers) from anywhere.

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

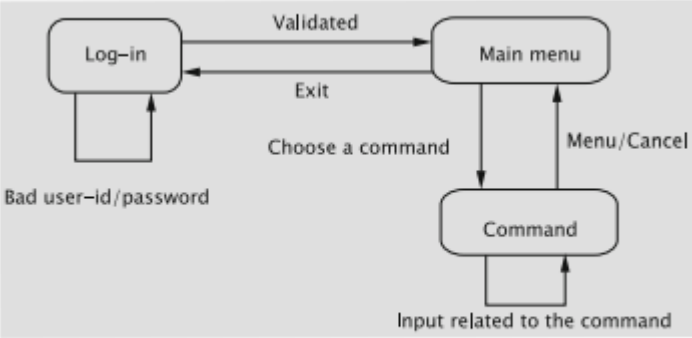


Fig. 12.8 State transition diagram for logging in

When the user types in the URL for the library, the system presents a log-in screen for entering the user id and password. If the user types in a bad user id/password combination, the system presents the log in screen again with an error message.

On successful validation, the system displays a menu that contains clickable options. The Command State in Fig. 12.8 denotes the general flow of a command. When a certain command is chosen, we enter a state that represents the command. How the transitions take place within a command obviously depends on what the command is. All screens allow an option to cancel and go back to the main menu. If this option is chosen, the system goes on to display the main menu awaiting the next command.

When the exit command is chosen, the system logs the user out and presents the log in screen again.

Add Book

The flow is shown in Fig. 12.9. 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.

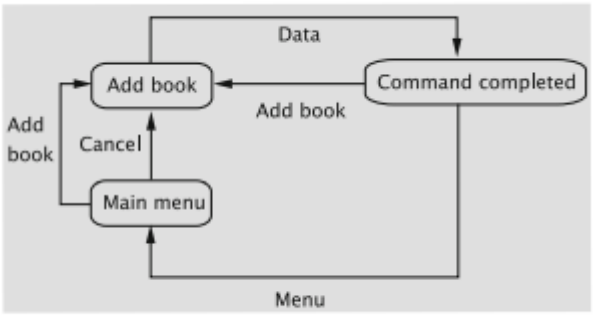


Fig. 12.9 State transition diagram for add book

From the Command Completed state, the system must allow the user to add another book or go back to the menu. In the Add Book state, the user has the option to cancel the operation and go back to the main menu.

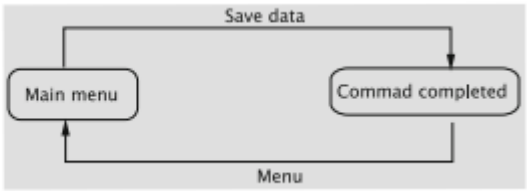
Add Member, Return Book, Remove Book

The requirements are similar to the ones for adding books. We need to accept some input (member details or book id) from the user, access the Library object to invoke one of its methods, and display the result. So we do not describe them here nor do we give the corresponding state transition diagrams.

Save Data

When the data is to be written to disk, no further input is required from the user. The system should carry out the task and print a message about the outcome. The state transition diagram is given in Fig. 12.10.

Fig. 12.10 State transition diagram for saving data



Retrieve Data

The requirements are similar to those for saving data.

Issue Book

This is one of the more complicated commands. As shown in the state transition diagram in Fig. 12.11, a book may be checked out in two different ways: First, a member is allowed to check it out himself/herself. Second, 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.

A second reason for the complexity arises from the fact that any number of books may be checked out. Thus, after each book is checked out, the system must ask if more books need to be issued or not. The system must either go to the Get Book Id state for one more book id or to the Main Menu state.

As usual, it should be possible to cancel the operation at any time.

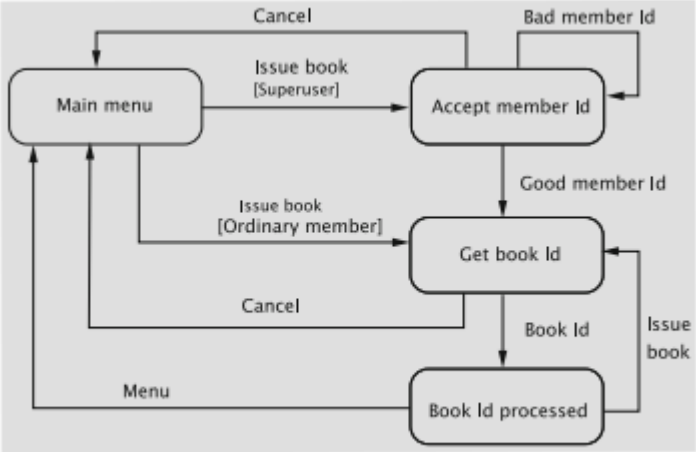


Fig. 12.11 State transition diagram for issuing books

Place Hold, Remove Hold, Print Transactions

The requirements for these are similar to those for issuing a book, so we omit their description.

Renew Books

The system must list the title and due date of all the books loaned to the member. For each book, the system must also present a choice to the user to renew the book. After making the choices, the member clicks a button to send any renew requests to the system. For every book renewal request, the system must display the title, the due date (possibly changed because of renewal), and a message that indicates whether the renewal request was honoured. After viewing the results, the member uses a link on the page to navigate to the main menu. The state transition diagram is given in Fig. 12.12.

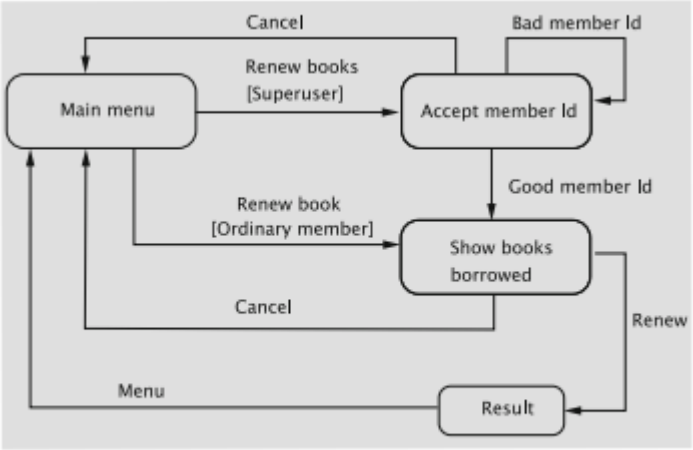


Fig. 12.12 State transition diagram for renewing books

Design and Implementation

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

1. Classes associated with the library; you will recall that this includes 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

Examples of HTML file fragments

To show how this approach works in practice, consider the two commands, one for returning and the other for removing books. In both, the user must be presented with a web page that asks him/her to enter a book id. We have just one file that displays this page. However, the servlet that needs to be invoked will change depending on the context. Therefore, we code the servlet name as below.

```
<form action="GOTO_WITH_BOOKID" method="post">
```

A similar approach is taken for accepting member ids.

For every web page, the header should display a title that depends on the context. We maintain just one file for the header. This file has a string TITLE that stands for the title of the web page. Depending on which page is being displayed, TITLE is replaced by an appropriate string, which gets displayed in the title bar.

When a command is completed, we need to display a web page. For most commands, the data to be displayed is small enough that it can be thought of as a simple string. We, therefore, employ just one file, commandCompleted.html, to carry out this task. This file is adapted, however, in two different ways.

1. The result to be displayed will vary on the command as well as whether the operation was successful. To take care of this, the file has a string called RESULT.

```
<h3> RESULT <br></h3>
```

This may be replaced by strings such as Book not found and Member added. Once the file is read into a string, the RESULT string is replaced by the appropriate result of executing the command. The following pseudocode gives the idea.

```
String result;
Member member;
String htmlFile = getFile("commandCompleted.html");
if ((member = library.addMember(name, address, phone)) == null) {
  htmlFile = htmlFile.replace("TITLE", "Member not Added");
  result = "Member could not be added";
} else {
  htmlFile = htmlFile.replace("TITLE", "Member Added");
  result = member.getName() + " ID: " + member.getId() + " added";
}
htmlFile = htmlFile.replace("RESULT", result);
```

2. To reduce the number of mouse clicks, the user may be given the option to repeat the command whose result is displayed by the commandCompleted.html file. For example, after completing the Add Book command, we need to give an option to issue the command once again so that the user can add another book. Since the code where control should go to depends on the command that was just executed, some adaptation is in order. This is facilitated by having the line

```
<a href="REPLACE_JS">REPLACE_COMMAND</a><br>
```

in the HTML file.

In the case of Add Member, we substitute REPLACE_COMMAND by Add Book, which provides a link that the user can click, and REPLACE_JS by addmemberinitialization, which locates the Java class that is given the control when the link is clicked.

```
htmlFile = htmlFile.replace("REPLACE_J,S", "addmemberinitialization");
htmlFile = htmlFile.replace("REPLACE_COMMAND", "Add Member");
```

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.

The directory structure is as in Fig. 12.13. We store the HTML files in a directory named Library, which is a subdirectory of webapps, which, in turn, is a subdirectory of the home directory of Tomcat. The servlets are in the package library, which is stored in Library/WEB-INF/classes. The implementation of the backend classes such as Member, Catalog, etc. is in the package basicImplementation.

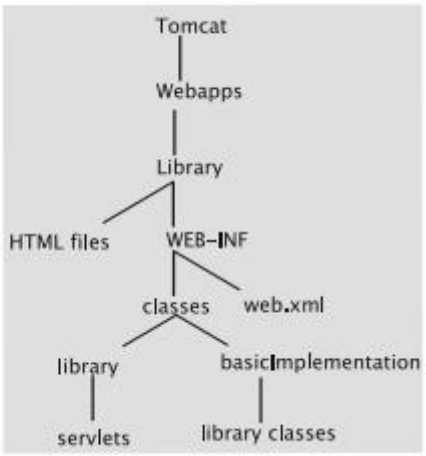


Fig. 12.13 Directory structure for the servlets

Our implementation requires that the user create an environment variable named LIBRARY-HOME that has as value the absolute path name of the directory that houses the HTML files.

The deployment descriptor elements are defined in a file called web.xml. While this file permits a large number of tags, our use of them is limited to mapping the URLs to servlets. To understand how this is done, first examine the following lines of XML code.

```
<servlet-mapping>
    <servlet-name>LoginServlet</servlet-name>
    <url-pattern>/login</url-pattern>
</servlet-mapping>
```

Thus when we write code such as

```
URL=login
```

in the HTML file, the string login is mapped to the servlet name LoginServlet.

But the servlet name given by the tag is just a name that is mapped to the fully-qualified class name of the servlet as below.

```
<servlet>
    <servlet-name>LoginServlet</servlet-name>
    <servlet-class>library.Login</servlet-class>
</servlet>
```

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. As we have seen earlier, this method has two parameters: a request parameter of type HttpServletRequest and a response parameter of type HttpServletResponse.

Each command is organised as a combination of one to three servlets. They need a number of common utility functions during the course of processing. These methods and doPost and

doGet are collected into a class named LibraryServlet. This class has the structure shown in Fig. 12.14.

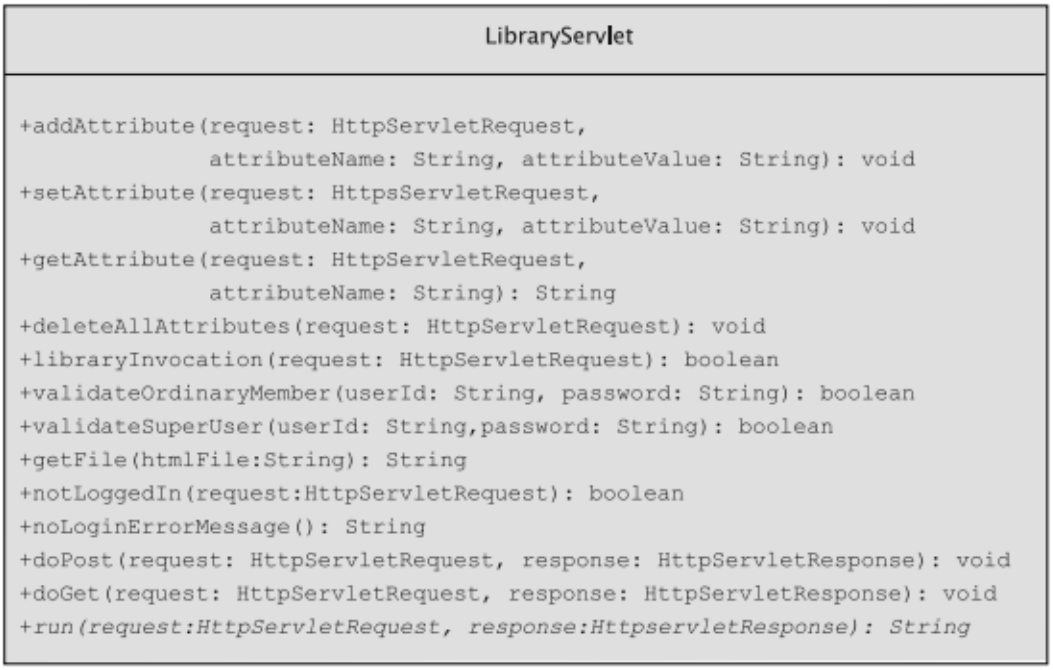


Fig. 12.14 Class diagram for Library servlet

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

1. One group contains methods that store information about the user. This information includes the user id, the type of terminal from which the user has logged in, etc. and are stored in attributes associated with the session object. The methods are addAttribute, setAttribute, getAttribute, and deleteAllAttributes.

2. Methods to validate users and help assess access rights. The validateSuper User method checks whether the user is a superuser and validateOrdinary Member does the same job for ordinary members. The method library Invocation returns true if and only if the user has logged in from a terminal located within the library.

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. The method notLoggedIn returns true if and only if the user has not currently logged in. The method noLoginError Message returns HTML code that displays an error message when a person who has not logged in attempts to execute a command.

5. The final group of commands deal with processing the request and responding to it. The doGet message calls doPost, which does some minimal processing needed for all commands and then calls the abstract runmethod, which individual servlets override.

Execution flow Processing a request sometimes involves simply generating an HTML page, which is quite straightforward. This is best understood by following a sample command. We choose as example, the command to remove a book. A somewhat simplified sequence of what takes place in the course of the execution of this command is shown in Fig. 12.15.

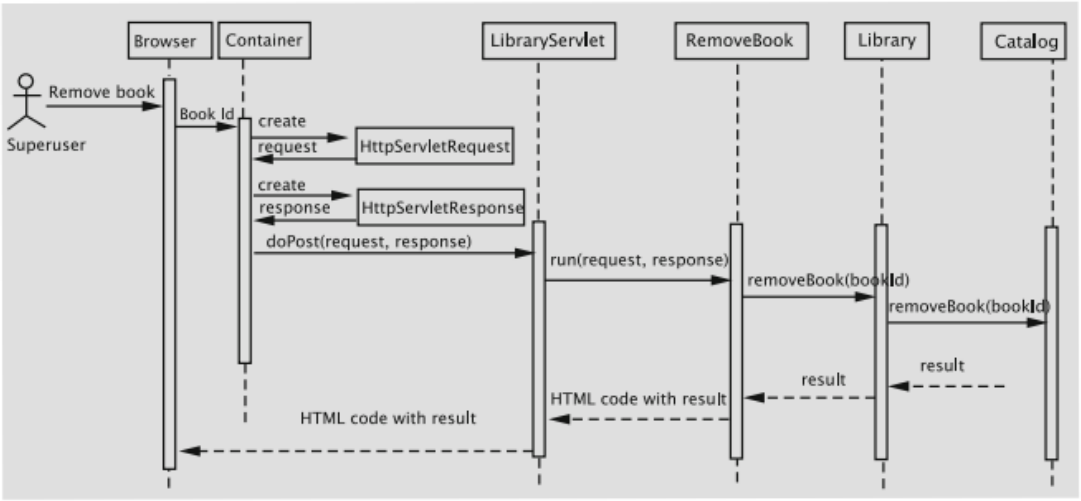


Fig. 12.15 Simplified sequence diagram for removing books

Discussion and Further Reading

RMI provides a level of abstraction much higher than the traditional communication mechanism in networks, viz. sockets. A socket is an endpoint of a communication channel to or from which data is transmitted in the network. Sockets are analogous to phones and a socket allocated on a machine is uniquely associated with a process running on it. The type of socket associated with a process depends on the transport layer in use (TCP or UDP, for example). A socket can have an associated port number using which processes may send messages to it. Socket programming is possible in many modern programming languages including C and Java

The Common Object Request Broker Architecture (CORBA), standardised by the Object Management Group (OMG), is another approach to distributed object-based computing. It allows a distributed, heterogeneous collection of objects to interoperate, and automates many common network programming tasks such as object registration, location, and activation, error-handling, parameter marshalling and demarshalling, security control and concurrency control.

Like RMI, the services that a CORBA object provides are defined by its interface. Again, as in RMI, object references are really of interface types. The Object Request Broker (ORB) is responsible for delivering requests from a client to a remote object and to return the results.

The Java Servlet technology is just one of the tools available for creating web-based systems. PHP is a scripting language that usually runs on the server side. It can have HTML code embedded into it and outputs web pages. ASP.NET is another competing scripting technology from Microsoft for building web-based applications. JSP is similar to PHP and

ASP, the difference being that we intersperse Java code with HTML code to create dynamic web pages. Other technologies such as Ruby on Rails (RoR) are also available.

A Note on Input and Output

Inputting numeric values through the keyboard has been a problem in Java. We need to read a string and extract a number from it. One way of inputting data is through a graphical user interface (GUI). A class called JOptionPane has a method named showInputDialog, which can be used for accepting a String. The String can then be parsed to retrieve the proper value.

```
String response;
response = JOptionPane.showInputDialog("Enter a number");
int num = Integer.parseInt(response);
```

The code opens up a dialog box for entering a string. After inputting the data, the user can click "O.K." The string is stored in response, which is parsed by the code

```
Integer.parseInt(response);
```

It returns the integer value stored in the string. (If the string does not have an integer in it, it would cause an "exception.")

Messages can also be displayed in a window using the method showMessageDialog in JOptionPane. The format is

```
JOptionPane.showMessageDialog(null, message-as-a-string);
```

Selection Statements

Java supports if else statements and switch statements. Both allow nesting. The syntax of the if else statement is

```
if <condition>
  <statement>
[else
   <statement>]
```

The else part is optional.

Here is a program that accepts the age of a person and prints out whether the person is eligible to vote.

```
import javax.swing.*;
public class VoteEligibility {
  public static void main(String[] s) {
    int age;
    age = Integer.parseInt(JOptionPane.showInputDialog(
                              "Please enter your age"));
    if (age >= 18) {
      JOptionPane.showMessageDialog(null, "you are eligible to
                vote");
    } else {
      JOptionPane.showMessageDialog(null, "wait " + (18 - age)
                           + " years!");
    }
    System.exit(0);
  }
}
```

The next example selects people younger than 20 and all females over 30.

```
selected = false;
if (age < 20) {
    selected = true;
} else if (age > 30) {
  gender =  JOptionPane.showInputDialog("Enter gender: ")
                  .charAt(0);
  if (gender == 'f' || gender == 'F') {
    selected = true;
  }
}
```

Logical operators are

&&	logical and
||	logical or
!	logical not

The switch statement allows us to handle the situation when there are numerous cases. Here is an example.

```
int month = Integer.parseInt(JOptionPane.showInputDialog(null,
                          "Enter month 1-12"));
switch (month) {
    case 1:    JOptionPane.showMessageDialog(null, "January");
               break;
    case 2:    JOptionPane.showMessageDialog(null, "February");
               break;
    case 3:    JOptionPane.showMessageDialog(null, "March");
               break;
    case 4:    JOptionPane.showMessageDialog(null, "April");
               break;
    case 5:    JOptionPane.showMessageDialog(null, "May");
               break;
    case 6:    JOptionPane.showMessageDialog(null, "June");
               break;
    case 7:    JOptionPane.showMessageDialog(null, "July");
               break;
    case 8:    JOptionPane.showMessageDialog(null, "August");
               break;
    case 9:    JOptionPane.showMessageDialog(null, "September");
               break;
    case 10:   JOptionPane.showMessageDialog(null, "October");
               break;
    case 11:   JOptionPane.showMessageDialog(null, "November");
               break;
    case 12:   JOptionPane.showMessageDialog(null, "December");
               break;
}
```

Loops

Java, like C and C++, allows three types of loops: for, while, and do.

while The while loop has a simple syntax.

```
while (condition)
  statement;
```

The statement is executed as long as the condition is true. Before each iteration, the condition is checked. If it is true, the loop is executed once and the condition is checked once again and the process repeats until the condition is false.

Here are some examples of the use of while loop.

```
int number = 10;
while (number <= 25) {
   System.out.println(number);
   number++;
}
```

for The for loop has the following syntax

```
for (expression1; condition; expression2)
     statement;
```

The code works as follows:

1. Evaluate expression1.
2. Evaluate condition.
3. If the evaluation in (2) returns true, enter the loop and execute the statement, which can be a block. Otherwise, exit the loop.
4. Evaluate expression2.
5. Go to (2) above.

do The do loop executes at least once. At the end of the first and succeeding iterations, a condition is checked. If the condition is true, the next iteration is performed. The syntax is

```
do
  statement
while (condition);
```

The following example makes the user enter “Yes”, “No”, or “cancel” (caseinsensitive).

```
String response;
do {
  response = JOptionPane.showInputDialog
                 ("Enter yes, no, or cancel");
} while (! response.equalsIgnoreCase("yes")
    && ! response.equalsIgnoreCase("no")
    && ! response.equalsIgnoreCase("cancel"));
```

Arrays

Java supports the creation of arrays of any number of dimensions. The process of creating an array can be thought of as consisting of two steps:

1. Declare a variable that refers to the array. This is not the array itself, but eventually contains the address of the array, which has to be dynamically allocated.

2. Allocate the array itself and make the variable declared in (1) above to point to this array.

The following code creates a variable that can serve as a reference to an array of integers

```
int[] a;
```

An array of five integers is created during execution by the following code.

```
new int[5];
```

The new operator returns the address of the array; this is termed the reference in Java. We make a hold the reference to the array by writing

```
a = new int[5];
```

The first cell of the array is indexed by 0. If the array has n elements, the last cell is indexed n − 1.

Array cells are referred by the notation a[index].

The following code stores 1 in a[0], 2 in a[1], etc. and then prints these values.

```
for (int index = 0; index < 5; index++) {
    a[index] = index + 1;
}
for (int index = 0; index < 5; index++) {
    System.out.println(a[index]);
}
```

The following program reads in a sequence of numbers and prints them in reverse. The number of numbers is the first number read in. An array large enough to hold the sequence is then allocated.

```
import javax.swing.*;
public class PrintInReverse {
  public static void main(String[] s) {
    int[] numbers;
    int numberOfNumbers = Integer.parseInt(
              JOptionPane.showInputDialog
                     ("Enter max. number of numbers"));
    numbers = new int[numberOfNumbers];
    boolean lookForAnotherNumber = true;
    int count = 0;
    while (lookForAnotherNumber) {
      if (count >= numbers.length) {
        lookForAnotherNumber = false;
      } else {
        String string = JOptionPane.showInputDialog
                                ("Enter a number");
        if (string.length() == 0) {
          lookForAnotherNumber = false;
        } else {
          int number = Integer.parseInt(string);
          numbers[count++] = number;
        }
      }
    }
    for (int index = count - 1; index >= 0; index--) {
      System.out.println(numbers[index]);
    }
    System.exit(0);
  }
}
```

Multi-dimensional Arrays

Let us look at an example of creating multi-dimensional arrays, which will suggest how to allocate arrays of higher dimension.

```
double [][] prices;
prices = new double[5][10];
prices[2][4] = 76.5;
```