MODULE 1

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Yet experienced object-oriented designers do make good designs. Meanwhile new designers are overwhelmed by the options available and tend to fall back on non-object-oriented techniques they've used before. It takes a long time for novices to learn what good object-oriented design is all about. Experienced designers evidently know something inexperienced ones don't. What is it?

One thing expert designers know *not* to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

An analogy will help illustrate the point. Novelists and playwrights rarely design their plots from scratch. Instead, they follow patterns like "Tragically Flawed Hero" (Macbeth, Hamlet, etc.) or "The Romantic Novel" (countless romance novels). In the same way, object-oriented designers follow patterns like "represent states with objects"

and "decorate objects so you can easily add/remove features." Once you know the pattern, a lot of design decisions follow automatically.

We all know the value of design experience. How many times have you had design *déjà-vu*—that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it. However, we don't do a good job of recording experience in software design for others to use.

The purpose of this book is to record experience in designing object-oriented software as **design patterns**. Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is to capture design experience in a form that people can use effectively. To this end we have documented some of the most important design patterns and present them as a catalog.

Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design "right" faster.

None of the design patterns in this book describes new or unproven designs. We have included only designs that have been applied more than once in different systems. Most of these designs have never been documented before. They are either part of the folklore of the object-oriented community or are elements of some successful object-oriented systems—neither of which is easy for novice designers to learn from. So although these designs aren't new, we capture them in a new and accessible way: as a catalog of design patterns having a consistent format.

Despite the book's size, the design patterns in it capture only a fraction of what an expert might know. It doesn't have any patterns dealing with concurrency or distributed programming or real-time programming. It doesn't have any application domain-specific patterns. It doesn't tell you how to build user interfaces, how to write device drivers, or how to use an object-oriented database. Each of these areas has its own patterns, and it would be worthwhile for someone to catalog those too.

1.1 What Is a Design Pattern?

Christopher Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [AIS+77, page *x*]. Even though Alexander was talking about patterns in buildings and towns, what he says is true about object-oriented design patterns. Our solutions are expressed in terms of objects and interfaces instead of walls

and doors, but at the core of both kinds of patterns is a solution to a problem in a context.

In general, a pattern has four essential elements:

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

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

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

4. The **consequences** are the results and trade-offs of applying the pattern. Though consequences are often unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

   The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability. Listing these consequences explicitly helps you understand and evaluate them.

Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block. For this book we have concentrated on patterns at a certain level of abstraction. *Design patterns* are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is. Nor are they complex, domain-specific designs for an entire application or subsystem. The design patterns in this book 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. Since we must eventually implement our designs, a design pattern also provides sample C++ and (sometimes) Smalltalk code to illustrate an implementation.

Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages like Smalltalk and C++ rather than procedural languages (Pascal, C, Ada) or more dynamic object-oriented languages (CLOS, Dylan, Self). We chose Smalltalk and C++ for pragmatic reasons: Our day-to-day experience has been in these languages, and they are increasingly popular.

The choice of programming language is important because it influences one's point of view. Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called "Inheritance," "Encapsulation," and "Polymorphism." Similarly, some of our patterns are supported directly by the less common object-oriented languages. CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor (page 331). In fact, there are enough differences between Smalltalk and C++ to mean that some patterns can be expressed more easily in one language than the other. (See Iterator (257) for an example.)

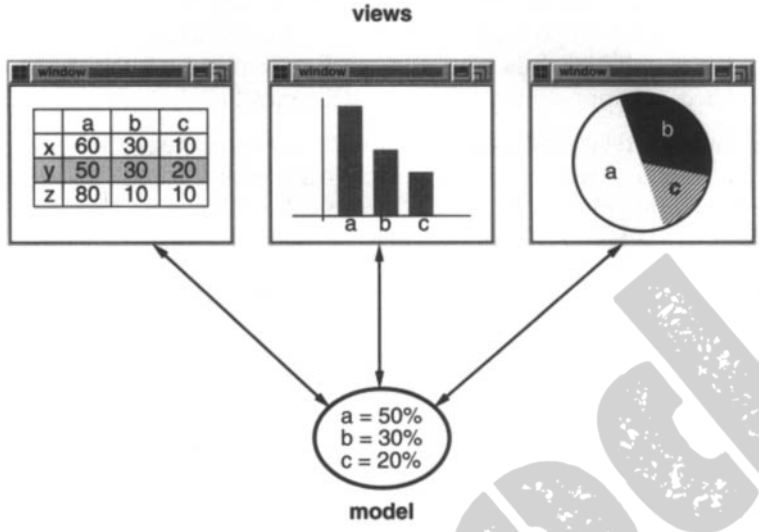
1.2 Design Patterns in Smalltalk MVC

The Model/View/Controller (MVC) triad of classes [KP88] is used to build user interfaces in Smalltalk-80. Looking at the design patterns inside MVC should help you see what we mean by the term "pattern."

MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.

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

The following diagram shows a model and three views. (We've left out the controllers for simplicity.) The model contains some data values, and the views defining a spreadsheet, histogram, and pie chart display these data in various ways. The model communicates with its views when its values change, and the views 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. This more general design is described by the Observer (page 293) design pattern.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. 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.

Again, we could think of this as a design that lets us treat a composite view just like we treat one of its components. But the design is applicable to a more general problem, which occurs whenever we want to group objects and treat the group like an individual object. This more general design is described by the Composite (163) design pattern. It lets you create a class hierarchy in which some subclasses define primitive objects (e.g., Button) and other classes define composite objects (CompositeView) that assemble the primitives into more complex objects.

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

The View-Controller relationship is an example of the Strategy (315) design pattern. A Strategy is an object that represents an algorithm. It's useful when you want to replace the algorithm either statically or dynamically, when you have a lot of variants of the algorithm, or when the algorithm has complex data structures that you want to encapsulate.

MVC uses other design patterns, such as Factory Method (107) to specify the default controller class for a view and Decorator (175) to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite, and Strategy design patterns.

1.3 Describing Design Patterns

How do we describe design patterns? Graphical notations, while important and useful, aren't sufficient. They simply capture the end product of the design process as relationships between classes and objects. To reuse the design, we must also record the decisions, alternatives, and trade-offs that led to it. Concrete examples are important too, because they help you see the design in action.

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

Pattern Name and Classification

The pattern's name conveys the essence of the pattern succinctly. A good name is vital, because it will become part of your design vocabulary. The pattern's classification reflects the scheme we introduce in Section 1.5.

Intent

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

Also Known As

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

Motivation

A scenario that illustrates a design problem and how the class and object structures

in the pattern solve the problem. The scenario will help you understand the more abstract description of the pattern that follows.

Applicability

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

Structure

A graphical representation of the classes in the pattern using a notation based on the Object Modeling Technique (OMT) [RBP+91]. We also use interaction diagrams [JCJO92, Boo94] to illustrate sequences of requests and collaborations between objects. Appendix B describes these notations in detail.

Participants

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

Collaborations

How the participants collaborate to carry out their responsibilities.

Consequences

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

Implementation

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

Sample Code

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

Known Uses

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

Related Patterns

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

The appendices provide background information that will help you understand the patterns and the discussions surrounding them. Appendix A is a glossary of terminology

we use. We've already mentioned Appendix B, which presents the various notations. We'll also describe aspects of the notations as we introduce them in the upcoming discussions. Finally, Appendix C contains source code for the foundation classes we use in code samples.

1.4 The Catalog of Design Patterns

The catalog beginning on page 79 contains 23 design patterns. Their names and intents are listed next to give you an overview. The number in parentheses after each pattern name gives the page number for the pattern (a convention we follow throughout the book).

Abstract Factory (87) Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Adapter (139) Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge (151) Decouple an abstraction from its implementation so that the two can vary independently.

Builder (97) Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Chain of Responsibility (223) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command (233) Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Composite (163) Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator (175) Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade (185) Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Factory Method (107) Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Flyweight (195) Use sharing to support large numbers of fine-grained objects efficiently.

Interpreter (243) Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator (257) Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator (273) Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Memento (283) Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer (293) Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Prototype (117) Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Proxy (207) Provide a surrogate or placeholder for another object to control access to it.

Singleton (127) Ensure a class only has one instance, and provide a global point of access to it.

State (305) Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy (315) Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor (331) Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

1.5 Organizing the Catalog

Design patterns vary in their granularity and level of abstraction. Because there are many design patterns, we need a way to organize them. This section classifies design patterns so that we can refer to families of related patterns. The classification helps you

		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

learn the patterns in the catalog faster, and it can direct efforts to find new patterns as well.

We classify design patterns by two criteria (Table 1.1). The first criterion, called **purpose**, reflects what a pattern does. Patterns can have either **creational**, **structural**, or **behavioral** purpose. Creational patterns concern the process of object creation. Structural patterns deal with the composition of classes or objects. Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.

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

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

There are other ways to organize the patterns. Some patterns are often used together. For example, Composite is often used with Iterator or Visitor. Some patterns are alternatives: Prototype is often an alternative to Abstract Factory. Some patterns result in similar designs even though the patterns have different intents. For example, the structure diagrams of Composite and Decorator are similar.

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

Clearly there are many ways to organize design patterns. Having multiple ways of thinking about patterns will deepen your insight into what they do, how they compare, and when to apply them.

1.6 How Design Patterns Solve Design Problems

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

Finding Appropriate Objects

Object-oriented programs are made up of objects. An **object** packages both data and the procedures that operate on that data. The procedures are typically called **methods** or **operations**. An object performs an operation when it receives a **request** (or **message**) from a **client**.

Requests are the *only* way to get an object to execute an operation. Operations are the *only* way to change an object's internal data. Because of these restrictions, the object's internal state is said to be **encapsulated**; it cannot be accessed directly, and its representation is invisible from outside the object.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, and on and on. They all influence the decomposition, often in conflicting ways.

Object-oriented design methodologies favor many different approaches. You can write a problem statement, single out the nouns and verbs, and create corresponding classes and operations. Or you can focus on the collaborations and responsibilities in your system. Or you can model the real world and translate the objects found during analysis into design. There will always be disagreement on which approach is best.

Many objects in a design come from the analysis model. But object-oriented designs often end up with classes that have no counterparts in the real world. Some of these are low-level classes like arrays. Others are much higher-level. For example, the Composite (163) pattern introduces an abstraction for treating objects uniformly that doesn't have a physical counterpart. Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible.

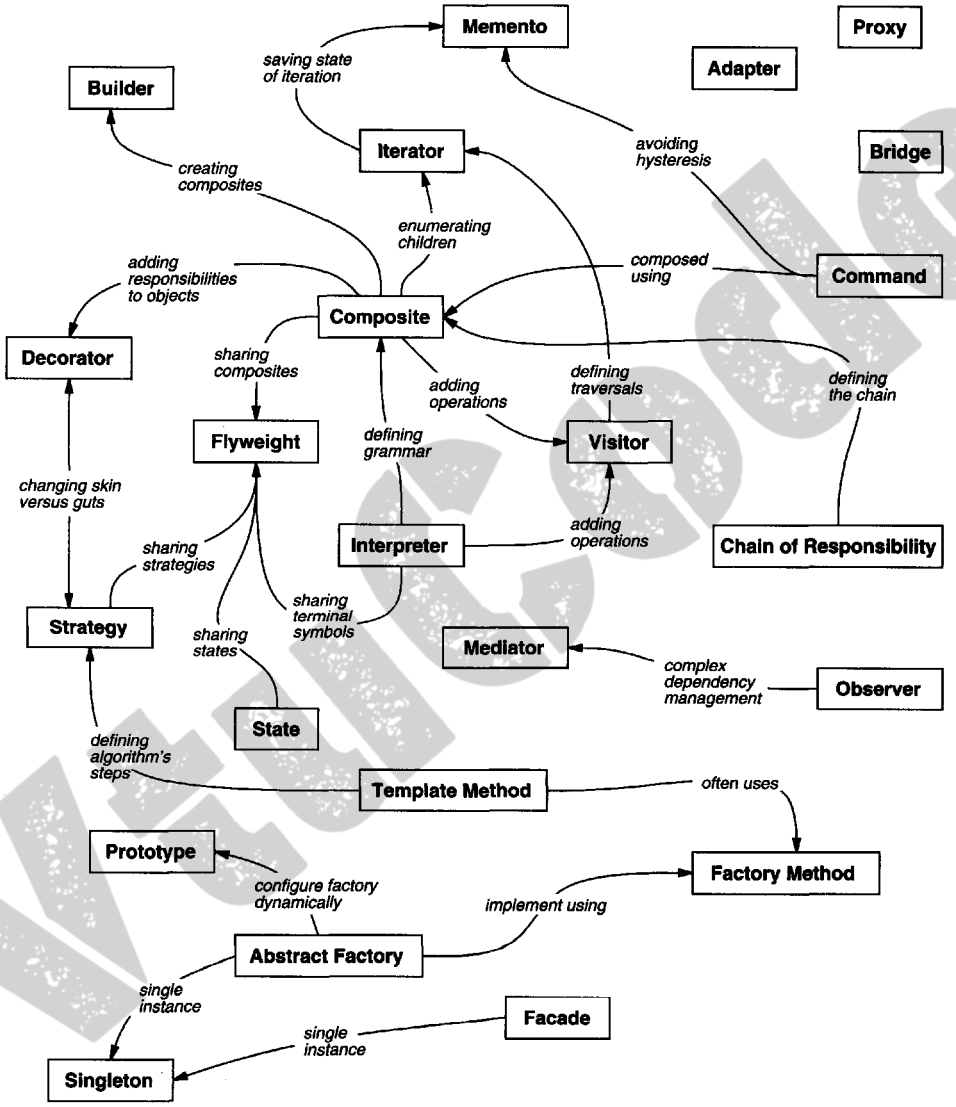


Figure 1.1: Design pattern relationships

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 (315) pattern describes how to implement interchangeable families of algorithms. The State (305) 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 (185) pattern describes how to represent complete subsystems as objects, and the Flyweight (195) pattern describes how to support huge numbers of objects at the finest granularities. Other design patterns describe specific ways of decomposing an object into smaller objects. Abstract Factory (87) and Builder (97) yield objects whose only responsibilities are creating other objects. Visitor (331) and Command (233) yield objects whose only responsibilities are to implement a request on another object or group of objects.

Specifying Object Interfaces

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

A **type** is a name used to denote a particular interface. We speak of an object as having the type "Window" if it accepts all requests for the operations defined in the interface named "Window." An object may have many types, and widely different objects can share a type. Part of an object's interface may be characterized by one type, and other parts by other types. Two objects of the same type need only share parts of their interfaces. Interfaces can contain other interfaces as subsets. We say that a type is a **subtype** of another if its interface contains the interface of its **supertype**. Often we speak of a subtype *inheriting* the interface of its supertype.

Interfaces are fundamental in object-oriented systems. Objects are known only through their interfaces. There is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation—different objects are free to implement requests differently. That

means two objects having completely different implementations can have identical interfaces.

When a request is sent to an object, the particular operation that's performed depends on *both* the request *and* the receiving object. Different objects that support identical requests may have different implementations of the operations that fulfill these requests. The run-time association of a request to an object and one of its operations is known as **dynamic binding**.

Dynamic binding means that issuing a request doesn't commit you to a particular implementation until run-time. Consequently, you can write programs that expect an object with a particular interface, knowing that any object that has the correct interface will accept the request. Moreover, dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as **polymorphism**, and it's a key concept in object-oriented systems. It lets a client object make few assumptions about other objects beyond supporting a particular interface. Polymorphism simplifies the definitions of clients, decouples objects from each other, and lets them vary their relationships to each other at run-time.

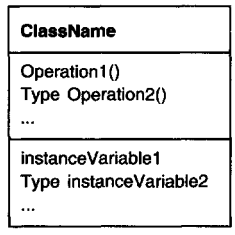
Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what *not* to put in the interface. The Memento (283) pattern is a good example. It describes how to encapsulate and save the internal 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 (175) and Proxy (207) require the interfaces of Decorator and Proxy objects to be identical to the decorated and proxied objects. In Visitor (331), the Visitor interface must reflect all classes of objects that visitors can visit.

Specifying Object Implementations

So far we've said little about how we actually define an object. An object's implementation is defined by its **class**. The class specifies the object's internal data and representation and defines the operations the object can perform.

Our OMT-based notation (summarized in Appendix B) depicts a class as a rectangle with the class name in bold. Operations appear in normal type below the class name. Any data that the class defines comes after the operations. Lines separate the class name from the operations and the operations from the data:

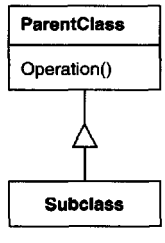


Return types and instance variable types are optional, since we don't assume a statically typed implementation language.

Objects are created by **instantiating** a class. The object is said to be an **instance** of the class. The process of instantiating a class allocates storage for the object's internal data (made up of **instance variables**) and associates the operations with these data. Many similar instances of an object can be created by instantiating a class.

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

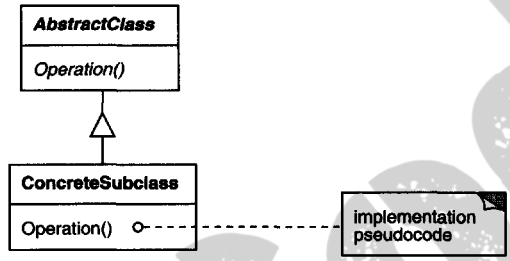
New classes can be defined in terms of existing classes using **class inheritance**. When a **subclass** inherits from a **parent class**, it includes the definitions of all the data and operations that the parent class defines. Objects that are instances of the subclass will contain all data defined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



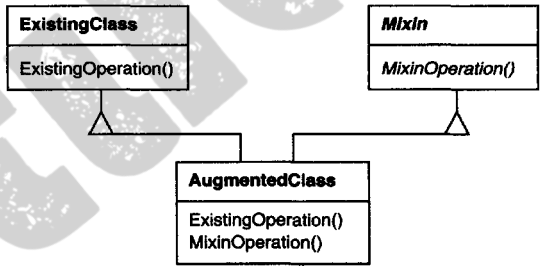
An **abstract class** is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated. The operations that an abstract class declares but doesn't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

Subclasses can refine and redefine behaviors of their parent classes. More specifically, a class may **override** an operation defined by its parent class. Overriding gives subclasses a chance to handle requests instead of their parent classes. Class inheritance lets you define classes simply by extending other classes, making it easy to define families of objects having related functionality.

The names of abstract classes appear in slanted type to distinguish them from concrete classes. Slanted type is also used to denote abstract operations. A diagram may include pseudocode for an operation's implementation; if so, the code will appear in a dog-eared box connected by a dashed line to the operation it implements.



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.

Of course, there's a close relationship between class and type. Because a class defines the operations an object can perform, it also defines the object's type. When we say that an object is an instance of a class, we imply that the object supports the interface defined by the class.

Languages like C++ and Eiffel use classes to specify both an object's type and its implementation. Smalltalk programs do not declare the types of variables; consequently, the compiler does not check that the types of objects assigned to a variable are subtypes of the variable's type. Sending a message requires checking that the class of the receiver implements the message, but it doesn't require checking that the receiver is an instance of a particular class.

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.

It's easy to confuse these two concepts, because many languages don't make the distinction explicit. In languages like C++ and Eiffel, inheritance means both interface and implementation inheritance. The standard way to inherit an interface in C++ is to inherit publicly from a class that has (pure) virtual member functions. Pure interface inheritance can be approximated in C++ by inheriting publicly from pure abstract classes. Pure implementation or class inheritance can be approximated with private inheritance. In Smalltalk, inheritance means just implementation inheritance. You can assign instances of any class to a variable as long as those instances support the operation performed on the value of the variable.

Although most programming languages don't support the distinction between interface and implementation inheritance, people make the distinction in practice. Smalltalk programmers usually act as if subclasses were subtypes (though there are some well-known exceptions [Coo92]); C++ programmers manipulate objects through types defined by abstract classes.

Many of the design patterns depend on this distinction. For example, objects in a Chain of Responsibility (223) must have a common type, but usually they don't share a common implementation. In the Composite (163) pattern, Component defines a common interface, but Composite often defines a common implementation. Command (233), Observer (293), State (305), and Strategy (315) are often implemented with abstract classes that are pure interfaces.

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.

However, implementation reuse is only half the story. Inheritance's ability to define families of objects with *identical* interfaces (usually by inheriting from an abstract class) is also important. Why? Because polymorphism depends on it.

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:

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

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.*

Don't declare variables to be instances of particular concrete classes. Instead, commit only to an interface defined by an abstract class. You will find this to be a common theme of the design patterns in this book.

You have to instantiate concrete classes (that is, specify a particular implementation) somewhere in your system, of course, and the creational patterns (Abstract Factory (87), Builder (97), Factory Method (107), Prototype (117), and Singleton (127)) let you do just that. By abstracting the process of object creation, these patterns give you different ways to associate an interface with its implementation transparently at instantiation. Creational patterns ensure that your system is written in terms of interfaces, not implementations.

Putting Reuse Mechanisms to Work

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

Inheritance versus Composition

The two most common techniques for reusing functionality in object-oriented systems are class inheritance and **object composition**. As we've explained, class inheritance lets

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

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

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.

But class inheritance has some disadvantages, too. 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" [Sny86]. 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.

Implementation dependencies can cause problems when you're trying to reuse a subclass. Should any aspect of the inherited implementation not be appropriate for new problem domains, the parent class must be rewritten or replaced by something more appropriate. This dependency limits flexibility and ultimately reusability. One cure for this is to inherit only from abstract classes, since they usually provide little or no implementation.

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.

Ideally, you shouldn't have to create new components to achieve reuse. You should be able to get all the functionality you need just by assembling existing components through object composition. But this is rarely the case, because the set of available components is never quite rich enough in practice. Reuse by inheritance makes it easier to make new components that can be composed with old ones. Inheritance and object composition thus work together.

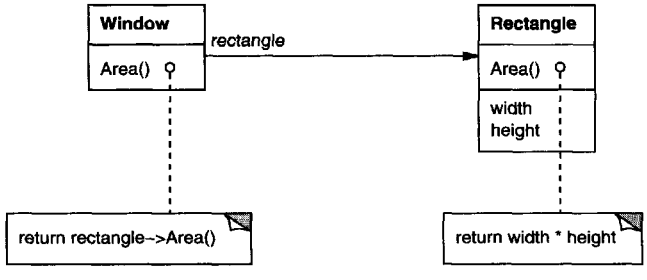
Nevertheless, our experience is that designers overuse inheritance as a reuse technique, and designs are often made more reusable (and simpler) by depending more on object composition. You'll see object composition applied again and again in the design patterns.

Delegation

Delegation is a way of making composition as powerful for reuse as inheritance [Lie86, JZ91]. In delegation, *two* objects are involved in handling a request: a receiving object delegates operations to its **delegate**. This is analogous to subclasses deferring requests to parent classes. But with inheritance, an inherited operation can always refer to the receiving object through the `this` member variable in C++ and `self` in Smalltalk. To achieve the same effect with delegation, the receiver passes itself to the delegate to let the delegated operation refer to the receiver.

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. Our window can become circular at run-time simply by replacing its Rectangle instance with a Circle instance, assuming Rectangle and Circle have the same type.

Delegation has a disadvantage it shares with other techniques that make software more flexible through object composition: Dynamic, highly parameterized software is harder to understand than more static software. There are also run-time inefficiencies, but the human inefficiencies are more important in the long run. Delegation is a good design choice only when it simplifies more than it complicates. It isn't easy to give rules that tell you exactly when to use delegation, because how effective it will be depends on the context and on how much experience you have with it. Delegation works best when it's used in highly stylized ways—that is, in standard patterns.

Several design patterns use delegation. The State (305), Strategy (315), and Visitor (331) patterns depend on it. 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. An object will only have one state, but it can have many strategies for different requests. The purpose of both patterns is to change the behavior of an object by changing the objects to which it delegates requests. In Visitor, the operation that gets performed on each element of an object structure is always delegated to the Visitor object.

Other patterns use delegation less heavily. Mediator (273) introduces an object to mediate communication between other objects. Sometimes the Mediator object implements operations simply by forwarding them to the other objects; other times it passes along a reference to itself and thus uses true delegation. Chain of Responsibility (223) handles requests by forwarding them from one object to another along a chain of objects. Sometimes this request carries with it a reference to the original object receiving the request, in which case the pattern is using delegation. Bridge (151) decouples an abstraction from its implementation. If the abstraction and a particular implementation are closely matched, then the abstraction may simply delegate operations to that implementation.

Delegation is an extreme example of object composition. It shows that you can always replace inheritance with object composition as a mechanism for code reuse.

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. To declare a list of integers, you supply the type "integer" as a parameter to the List parameterized type.

To declare a list of String objects, you supply the "String" type as a parameter. The language implementation will create a customized version of the List class template for each type of element.

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 (an application of Template Method (325)),
2. the responsibility of an object that's passed to the sorting routine (Strategy (315)), or
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. Which approach is best depends on your design and implementation constraints.

None of the patterns in this book concerns parameterized types, though we use them on occasion to customize a pattern's C++ implementation. Parameterized types aren't needed at all in a language like Smalltalk that doesn't have compile-time type checking.

Relating Run-Time and Compile-Time Structures

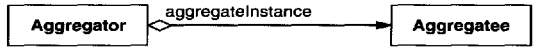
An object-oriented program's run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program's run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent. Trying to understand one from the other is like trying to understand the dynamism of living ecosystems from the static taxonomy of plants and animals, and vice versa.

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:



It's easy to confuse aggregation and acquaintance, because they are often implemented in the same way. In Smalltalk, all variables are references to other objects. There's no distinction in the programming language between aggregation and acquaintance. In C++, aggregation can be implemented by defining member variables that are real instances, but it's more common to define them as pointers or references to instances. Acquaintance is implemented with pointers and references as well.

Ultimately, 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 (in particular those that have object scope) capture the distinction between compile-time and run-time structures explicitly. Composite (163) and Decorator (175) are especially useful for building complex run-time structures. Observer (293) involves run-time structures that are often hard to understand unless you know the pattern. Chain of Responsibility (223) also results in communication patterns that inheritance doesn't reveal. In general, the run-time structures aren't clear from the code until you understand the patterns.

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.

To design the system so that it's robust to such changes, you must consider how the system might need to change over its lifetime. A design that doesn't take change into account risks major redesign in the future. Those changes might involve class

redefinition and reimplementation, client modification, and retesting. Redesign affects many parts of the software system, and unanticipated changes are invariably expensive.

Design patterns help you avoid this by ensuring that a system can change in specific ways. Each design pattern lets some aspect of system structure vary independently of other aspects, thereby making a system more robust to a particular kind of change.

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 (87), Factory Method (107), Prototype (117).

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 (223), Command (233).

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 (87), Bridge (151).

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 (87), Bridge (151), Memento (283), Proxy (207).

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 (97), Iterator (257), Strategy (315), Template Method (325), Visitor (331).

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 (87), Bridge (151), Chain of Responsibility (223), Command (233), Facade (185), Mediator (273), Observer (293).

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 (151), Chain of Responsibility (223), Composite (163), Decorator (175), Observer (293), Strategy (315).

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 (139), Decorator (175), Visitor (331).

These examples reflect the flexibility that design patterns can help you build into your software. How crucial such flexibility is depends on the kind of software you're building. Let's look at the role design patterns play in the development of three broad classes of software: application programs, toolkits, and frameworks.

Application Programs

If you're building an application program such as a document editor or spreadsheet, then *internal* reuse, maintainability, and extension are high priorities. Internal reuse ensures that you don't design and implement any more than you have to. Design

patterns that reduce dependencies can increase internal reuse. Looser coupling boosts the likelihood that one class of object can cooperate with several others. For example, when you eliminate dependencies on specific operations by isolating and encapsulating each operation, you make it easier to reuse an operation in different contexts. The same thing can happen when you remove algorithmic and representational dependencies too.

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. Reduced coupling also enhances extensibility. Extending a class in isolation is easier if the class doesn't depend on lots of other classes.

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.

Toolkit design is arguably harder than application design, because toolkits have to work in many applications to be useful. Moreover, the toolkit writer isn't in a position to know what those applications will be or their special needs. That makes it all the more important to avoid assumptions and dependencies that can limit the toolkit's flexibility and consequently its applicability and effectiveness.

Frameworks

A **framework** is a set of cooperating classes that make up a reusable design for a specific class of software [Deu89, JF88]. For example, a framework can be geared toward building graphical editors for different domains like artistic drawing, music composition, and mechanical CAD [VL90, Joh92]. Another framework can help you build compilers for different programming languages and target machines [JML92]. Yet another might help you build financial modeling applications [BE93]. You customize a framework to a particular application by creating application-specific subclasses of abstract classes from the framework.

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. Frameworks thus emphasize *design reuse* over code reuse, though a framework will usually include concrete subclasses you can put to work immediately.

Reuse on this level leads to an inversion of control between the application and the software on which it's based. When you use a toolkit (or a conventional subroutine library for that matter), you write the main body of the application and call the code you want to reuse. When you use a framework, you reuse the main body and write the code *it* calls. You'll have to write operations with particular names and calling conventions, but that reduces the design decisions you have to make.

Not only can you build applications faster as a result, but the applications have similar structures. They are easier to maintain, and they seem more consistent to their users. On the other hand, you lose some creative freedom, since many design decisions have been made for you.

If applications are hard to design, and toolkits are harder, then frameworks are hardest of all. A framework designer gambles that one architecture will work for all applications in the domain. Any substantive change to the framework's design would reduce its benefits considerably, since the framework's main contribution to an application is the architecture it defines. Therefore it's imperative to design the framework to be as flexible and extensible as possible.

Furthermore, because applications are so dependent on the framework for their design, they are particularly sensitive to changes in framework interfaces. As a framework evolves, applications have to evolve with it. That makes loose coupling all the more important; otherwise even a minor change to the framework will have major repercussions.

The design issues just discussed are most critical to framework design. A framework that addresses them using design patterns is far more likely to achieve high levels of design and code reuse than one that doesn't. Mature frameworks usually incorporate several design patterns. The patterns help make the framework's architecture suitable to many different applications without redesign.

An added benefit comes when the framework is documented with the design patterns it uses [BJ94]. People who know the patterns gain insight into the framework faster. Even people who don't know the patterns can benefit from the structure they lend to the framework's documentation. Enhancing documentation is important for all types of software, but it's particularly important for frameworks. Frameworks often pose a steep learning curve that must be overcome before they're useful. While design patterns might not flatten the learning curve entirely, they can make it less steep by making key elements of the framework's design more explicit.

Because patterns and frameworks have some similarities, people often wonder how or even if they differ. 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.

Frameworks are becoming increasingly common and important. They are the way that object-oriented systems achieve the most reuse. Larger object-oriented applications will end up consisting of layers of frameworks that cooperate with each other. Most of the design and code in the application will come from or be influenced by the frameworks it uses.

1.7 How to Select a Design Pattern

With more than 20 design patterns in the catalog to choose from, it might be hard to find the one that addresses a particular design problem, especially if the catalog is new and unfamiliar to you. Here are several different approaches to finding the design pattern that's right for your problem:

- *Consider how design patterns solve design problems.* Section 1.6 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.

- *Scan Intent sections.* Section 1.4 (page 8) 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. You can use the classification scheme presented in Table 1.1 (page 10) to narrow your search.

- *Study how patterns interrelate.* Figure 1.1 (page 12) shows relationships between design patterns graphically. Studying these relationships can help direct you to the right pattern or group of patterns.

- *Study patterns of like purpose.* The catalog (page 79) has three chapters, one for creational patterns, another for structural patterns, and a third for behavioral patterns. Each chapter starts off with introductory comments on the patterns and concludes with a section that compares and contrasts them. These sections give you insight into the similarities and differences between patterns of like purpose.
- *Examine a cause of redesign.* Look at the causes of redesign starting on page 24 to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign.
- *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.

1.8 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.

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

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.

These are just guidelines to get you started. Over time you'll develop your own way of working with design patterns.

No discussion of how to use design patterns would be complete without a few words on how *not* to use them. Design patterns should not be applied indiscriminately. Often they achieve flexibility and variability by introducing additional levels of indirection, and that can complicate a design and/or cost you some performance. A design pattern should only be applied when the flexibility it affords is actually needed. The Consequences sections are most helpful when evaluating a pattern's benefits and liabilities.

2.7 A Notation for Describing Object-Oriented Systems

We all know that it is important to document systems and programs. In this section, we introduce a notation called **Unified Modeling Language (UML)**, which is the standard for documenting object-oriented systems. Many different ideas had been suggested to document object-oriented systems in the past and the term "Unified" reflects the fact that UML was an attempt to unify these different approaches. Among the ones who contributed to the development of this notation, the efforts of Grady Booch, James Rumbaugh, and Ivor Jacobson deserve special mention. After the

initial notation was developed around 1995, the Object Management Group (OMG) took over the task of developing the notation further in 1997. As the years went by, the language became richer and, naturally, more complex. The current version is UML 2.0.

UML provides a pictorial or graphical notation for documenting the artefacts such as classes, objects and packages that make up an object-oriented system. UML diagrams can be divided into three categories.

1. **Structure diagrams** that show the static architecture of the system irrespective of time. For example, structure diagrams for a university system may include diagrams that depict the design of classes such as Student, Faculty, etc.
2. **Behaviour diagrams** that depict the behaviour of a system or business process.
3. **Interaction diagrams** that show the methods, interactions and activities of the objects. For a university system, a possible behaviour diagram would show how a student registers for a course.

Structure diagrams could be one of the following.

1. **Class diagrams**: They show the classes, their methods and fields.
2. **Composite structure diagrams**: They provide a means for presenting the details of a structural element such as a class. As an example, consider a class that represents a microcomputer system. Each object contains other objects such as CPU, memory, motherboard, etc, which would be shown as parts that make up the microcomputer system itself. The composite structure diagram for such a system would show these parts and exhibit the relationships between them helping the reader understand the details.
3. **Component diagrams**: Components are software entities that satisfy certain functional requirements specified by interfaces. These diagrams show the details of components.
4. **Deployment diagrams**: An object-oriented system consists of a number of executable files sometimes distributed across multiple computing elements. These diagrams show the assignment of executable files on the computing elements and the communication that involves between these entities.
5. **Object diagrams**: They are used to show how objects are related and used at runtime. For instance, in a university system we may show the object corresponding to a specific course and show other objects that represent students who have registered for the course. Since this shows an actual scenario that involves students and a course, it is far less abstract than class diagrams and contributes to a better understanding of the system.
6. **Package diagrams**: Classes may be grouped into packages and packages may reside in other packages. These diagrams show packages and dependencies among them: whether a change in one package may affect other packages.

Each of the six diagrams is a structure diagram. This hierarchy is illustrated in Fig. 2.2 as a tree with nodes representing these six diagrams as children of the Structure diagram node. It turns out that this method of showing a hierarchy is used in UML; so we are using UML notation itself to describe UML!

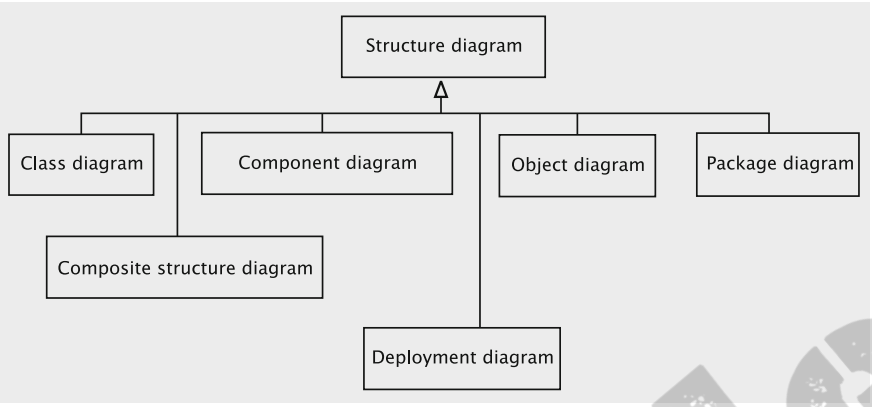


Fig. 2.2 Types of UML structure diagrams

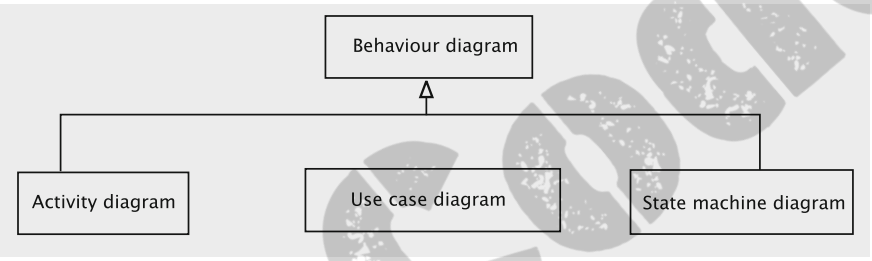


Fig. 2.3 Types of UML behaviour diagrams

Behaviour diagrams can be any of the following (see Fig. 2.3).

1. **Activity diagrams**: This is somewhat like a flowchart in that it shows the sequence of events in an activity. Just as a flowchart, it uses several types of nodes such as actions, decisions, merge points, etc. It accommodates objects with suitable types that depict objects, *object flows*, etc.
2. **Use case diagrams**: A use case is a single unit of some useful work. It involves a user (called an actor) and the system. An example of a use case in a university environment is a student registering for a course. A use case diagram shows the interaction involved in a use case.
3. **State machine diagrams**: It shows the sequence of states that an object goes through during its lifetime, e.g., the software that controls a washer for clothes. Initially, the washer is in the off state. After the soap is put in, the clothes are loaded and the on button pressed, the system goes to a state where it takes in water. In this state the system waits for a signal from the water sensor to indicate that the water has reached the required level. Then the system goes into the wash state where washing takes place. After this the system may go through further states such as rinse and spin and eventually reaches the washed state.

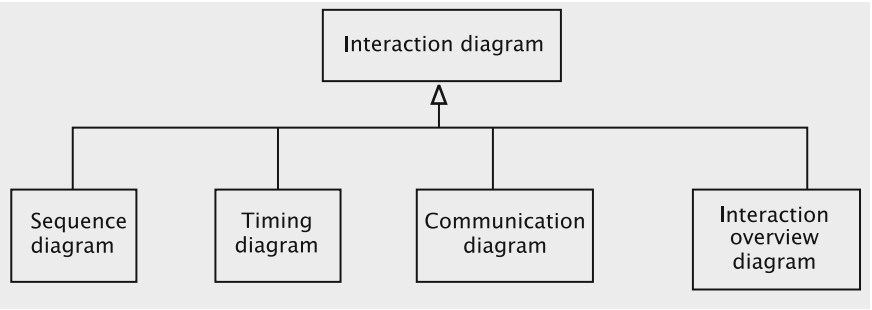


Fig. 2.4 Types of UML interaction diagrams

There are four types of interaction diagrams as shown in Fig. 2.4.

1. **Sequence diagrams**: A sequence diagram is an interaction diagram that details how operations are carried out—what messages are sent and when. Sequence diagrams are organised according to time. Time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence.
2. **Timing diagrams**: It shows the change in state of an object over time as the object reacts to events. The horizontal axis shows time and the state changes are noted on the vertical axis. Contrast this with sequence diagrams in which time is in the vertical axis.
3. **Communication diagrams**: A communication diagram essentially serves the same purpose as a sequence diagram. Just as in a sequence diagram, this diagram also has nodes for objects and uses directed lines between objects to indicate message flow and direction. However, unlike sequence diagrams, vertical direction has no relationship with time and message order is shown by numbering the directed lines that represent messages.
   Interactions that involve a large number of objects can be somewhat inconvenient to show using sequence diagrams because they must be arranged horizontally. Since no such restrictions are placed on communication diagrams, they are easier to draw. However, the order of messages can be harder to see in communication diagrams.
4. **Interaction overview diagrams**: An interaction overview diagram shows the high-level control flow in a system. It shows the interactions between interaction diagrams such as sequence diagrams and communication diagrams. Each node in the diagram can be an interaction diagram.

We will see examples of many of these diagrams as we develop concepts in this book. At this time, we show an example of a class diagram.

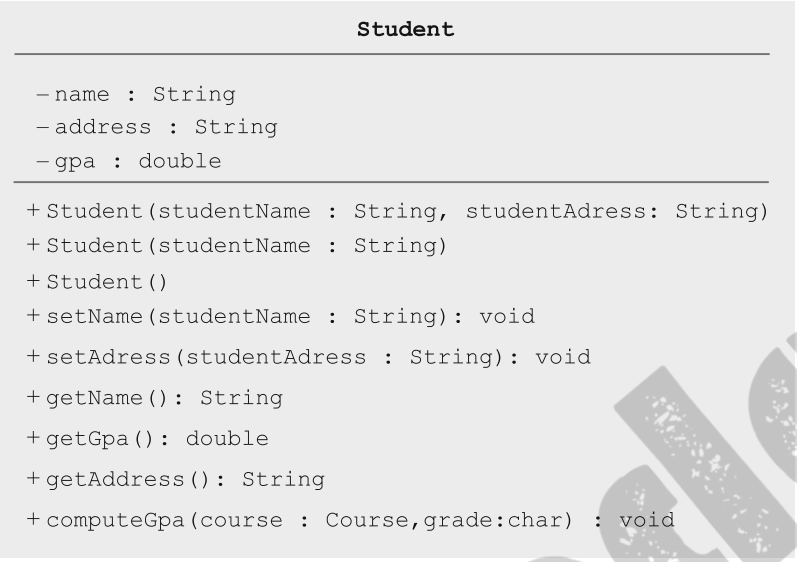


Fig. 2.5 Example of a class diagram

2.7.1 Class Diagrams

Figure 2.5 is an example of a class diagram. Each class is represented by a box, which is divided into three rectangles. The name of the class is given in the top rectangle. The attributes are shown with their names and their types in the second box. The third box shows the methods with their return types and parameters (names and types). The access specifier for each field and method is given just in front of the field name or method name. A $-$ sign indicates private access, $+$ stands for public access and # (not shown in this example) is used for protected access which we will discuss in Chap. 3.

2.7.2 Use Cases and Use Case Diagrams

A use case describes a certain piece of desired functionality of an application system. It is constructed during the analysis stage. It shows the interaction between an **actor**, which could be a human or a piece of software or hardware and the system. It does *not* specify *how* the system carries out the task.

As an example of a simple use case, let us describe what a simple ATM machine will do. A user may withdraw or deposit money into his bank account using this machine. This functionality is shown in the use case diagram in Fig. 2.6.

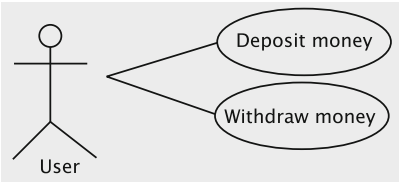


Fig. 2.6 Example of a use case diagram

Use cases may be verbally described in a table with two columns: The first column shows what the actor does and the second column depicts the system's behaviour.

We give below the use case for withdrawing money.

	Action performed by the actor		**Responses from the system**
1.	Inserts debit card into the 'Insert card' slot		
		2.	Asks for the PIN number
3.	Enters the PIN number		
		4.	Verifies the PIN. If the PIN is invalid, displays an error and goes to Step 8. Otherwise, asks for the amount
5.	Enters the amount		
		6.	Verifies that the amount can be withdrawn If not, display an error and goes to Step 8 Otherwise, dispenses the amount and updates the balance
7.	Takes the cash		
		8.	Ejects the card
9.	Takes the card		

Notice that the use case specifies the responsibilities of the two entities but does not show *how* the system processes the request. Throughout the book, we express use cases in a two-column format as above.

The use case as specified above does not say what the system is supposed to do in all situations. For example, what should the system do if something other than a valid ATM card is inserted? Such considerations may result in a more involved specification. What is specified above is sometimes called the **main flow.**

2.7.3 Sequence Diagrams

One of the major goals of design is to determine the classes and their responsibilities and one way of progressing toward the above goal is to create sequence diagrams for each use case we identify in the analysis stage. In such a diagram we break down the system into a number of objects and decide what each object should accomplish in the corresponding use case. That is, we delegate responsibilities.

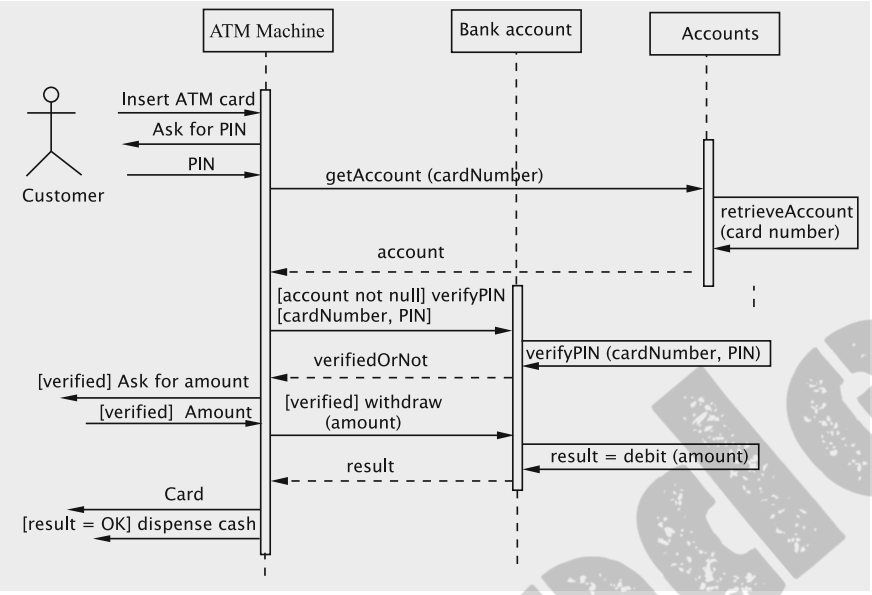


Fig. 2.7 Example of a simple sequence diagram

We have one column for each entity that plays a role in the use case. The vertical direction represents the flow of time. Horizontal arrows represent functionalities being invoked; the entity at the tail of the arrow invokes the named method on the entity at the head of the arrow.

For example, Fig. 2.7 shows the sequence diagram corresponding to the use case we gave above for withdrawing from an ATM. The rectangles at the top of the diagram represent the customer, the ATM, and two objects that reside in the bank database: `Accounts`, which stores all the account objects and `BankAccount`, which stores account-related information for a single account. For each object, we draw a dashed vertical line, called a **lifeline**, for showing the actions of the object. The long and thin rectangular boxes within these lifelines show when that object is active.

In many use cases, the actor interacts only with the left most entity, which usually represents some kind of interface. These interactions mirror the functionality described in the use case. The first arrow denotes the customer (actor) inserting the debit card into the ATM, which, in turn, asks for the PIN, as shown by the arrow from the ATM to the customer. Notice that the latter line is lower than the line that stands for the card insertion. This is because time increases as we go down in the diagram. The events in the sequence diagram that happen after the customer enters the PIN depend on how the system has been implemented. In our hypothetical example, we assume that the ATM has to access a central repository (viz., `Accounts`)

Fig. 2.8 An example of association

and attempt to retrieve the user's information.[1] If successful, the repository returns a reference to an object (`BankAccount`) representing the user's account, and the ATM then interacts with this object to complete the transaction.

The sequence diagram gives us the specifics of the implementation: the ATM calls the method `getAccount` on the `Accounts` object with the card number as parameter. The `Accounts` object either returns the reference to the appropriate `BankAccount` object corresponding to the card number, or `null` if such an account does not exist. When the `getAccount` method is invoked, the `Accounts` object calls the method `retrieveAccount` to get the `BankAccount` object to which the card number corresponds. Note the self-directed arc on the lifeline of the `Accounts` object, indicating that this computation is carried out locally within `Accounts`. The `getAccount` method invocation and its return are on separate lines, with the return shown by a dotted line.

The ATM then invokes the `verifyPIN` method on the `BankAccount` object to ensure that the PIN is valid. If for some reason the card is not valid, `Accounts` would have returned a `null` reference, in which case further processing is impossible. Therefore, the call to verify the PIN is conditional on reference being non-null. This is indicated in the sequence diagram by writing `[account not null]` along with the method call `verifyPIN`. Such a conditional is called a **guard**.

Just as `Accounts` called a method on itself, `BankAccount` calls the method `verifyPIN` to see if the PIN entered by the user is valid. The result, a boolean, is returned and shown on a separate dotted line in the diagram. If the PIN is valid, the ATM asks the user for the amount to be withdrawn. Once again, note the guard associated with this action. After receiving the value (the amount to be withdrawn), the machine sends the message `withdraw` with the amount as parameter to the `BankAccount` object, which verifies whether the amount can be withdrawn by calling the method `debit` on itself. The result is then returned to the ATM, which dispenses cash provided the result is acceptable.

Association

In our example that involved the ATM, `Accounts` and `BankAccount`, the `Accounts` instance contained all of the `BankAcount` objects, each of which could be retrieved by supplying a card number. This relationship can be shown using an association as in Fig. 2.8. Notice the number 1 above the line near the rectangle that represents `Accounts` and 0...* at the right end of the line near BankAccount. They mean that one `Accounts` object may hold references to zero or more BankAccount objects.

[1] This may not reflect a real ATM's behaviour, but bear in mind that this is a pedagogical exercise in UML, not banking.

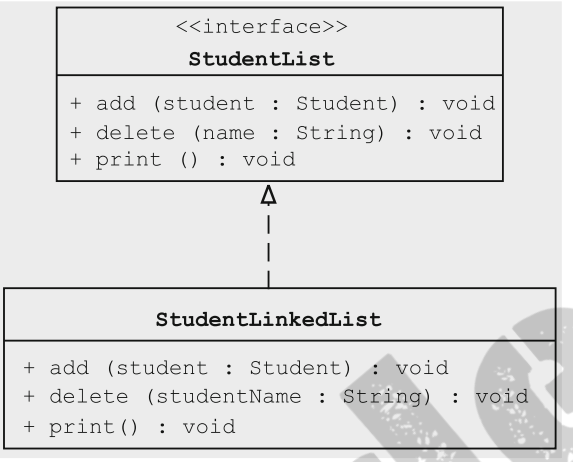


Fig. 2.9 Depicting interfaces and their implementation

Interfaces and Their Implementation

Interfaces and their implementation can be depicted in UML as in Fig. 2.9. With the `StudentList` interface and the class `StudentLinkedList` class that implements it, we draw one box to represent the interface and another to stand for the class. The methods are shown in both. The dotted line from the class to the interface shows that the class implements the interface.

Analysing a System

In Chaps. 6–8, we examine the essential steps in object-oriented software development: analysis, design, and implementation. To illustrate the process, we study a relatively simple example—a piece of software to manage a small library—whose function is limited to that of lending books to its members, receiving them back, doing the associated operations such as querying, registering members, etc., and keeping track of these transactions. In the course of these chapters, we go through the entire process of analysing, designing and implementing this system.

The software construction process begins with an analysis that determines the requirements of the system, which is what we introduce in this chapter. At this stage the focus is on determining what the system must perform without regard to the methodology to be employed. This process is carried out by a team of analysts, perhaps familiar with the specific type of application. The requirements are spelled out in a document known variously as the 'Requirements Specification', 'System Requirements', etc. Using these, the system analyst creates a model of the system, enabling the identification of some of the components of the system and the relationships between them. The end product of this phase is a *conceptual model* for the system which describes the functionality of the system, identifies its conceptual entities and records the nature of the associations between these entities.

Once the analysis has been satisfactorily completed, we move on to the design phase, which is addressed in the first part of Chap. 7. The design starts with a detailed breakdown of how the system will emulate the behaviour outlined in the model. In the course of this breakdown, all the parts of the system and their responsibilities are clearly identified. This step is followed by determining the software and hardware structures needed to implement the functionality discovered in the analysis stage. In the object-oriented world, this would mean deciding on the language or languages to be used, the packages, the platform, etc. The second part of Chap. 7 looks at implementation, wherein we discuss the lower-level issues, language features employed, etc.

A question that a conscientious beginner often ponders is: *Did I do a good job of the design*? or *Is my design really object-oriented*? Indeed, in the real world, it is often the case that designs conform to object-oriented principles to varying degrees.

Fortunately, in addition to the broad guidelines for what constitutes a good object-oriented design, there are some more specific rules that can be applied to look for common mistakes and correct them. These rules, known as *refactoring rules*, are more commonly presented as a means for improving the design of the existing code. They are, however, just as useful to check the design of a system before it is fully implemented. In Chap. 8, we introduce the concept of refactoring and apply these rules to our small system.

As our main focus in this book is the elaboration of the process of analysis, design, and implementation, we will bypass many software engineering and project management issues. We will not dwell on conceptual frameworks such as agile software development for managing the software development life cycle. We use UML notations in an appropriate manner that is sufficient to describe our design, but do not cover these exhaustively. For a detailed exposition on these topics, the reader is referred to the works cited at the end of each chapter.

6.1 Overview of the Analysis Phase

To put it in a simple sentence, the major goal of this phase is to address this basic question: what should the system do? A typical computer science student writes a number of programs by the time he/she graduates. 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. (Users are fussier than instructors!)

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

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

It is not always the case that these activities occur in the order listed. In fact, as the analysts gather the requirements, they will analyse and document what they have collected. This may point to holes in the information, which may necessitate further requirements collection.

6.2 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.

Imagine the scenario when you are asked to construct software for an application. The client may not always be clear in his/her mind as to what should be constructed. One reason for this is that it is difficult to imagine the workings of a system that is not yet built. Only when we actually use a specific application such as a word processor do we start realising the power and limitations of that system. Before actually dealing with it, one may have some general notions of what one would like to see, but may find it difficult to provide many details.

Incompleteness and errors in specifications can also occur because the client does not have the technical skills to fully realise what technology can and cannot deliver. Once again, the general concepts can be stated, but specifics are harder. A third reason for omissions is that it is all too common to have a client who knows the system very well and consequently either assumes a lot of knowledge on the part of the analyst or simply skips over the 'obvious details'.

Requirements 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.

It should be mentioned that initiating the development cycle for a software system is usually preceded by a phase that includes the initial conception and planning. A developer would be approached by a client who wishes to have a certain product

developed for his/her business. There would be a *domain* associated with the business, which would have its own jargon. Before approaching the developer, one would assume that the client has determined that a need for a product exists. Once all these issues are sorted out, the developer(s) would meet with the client and, perhaps several would-be end-users, to determine what is expected of the system. Such a process would result in a list of requirements of the system.

As mentioned at the beginning of this chapter, we study the development process by analysing, designing, and implementing a simple library system; this is introduced next.

6.2.1 Case Study Introduction

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

The business processes of the library system are listed below.

- **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. (For simplicity, let us assume that there is only one author per book. If there are multiple authors, let us say that the names will have to be concatenated to get a pretty huge name such as 'Brahma Dathan and Sarnath Ramnath'. As a result, to the system, it appears that there is just one author.) 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.

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

Our goal here is to present the reader with the big picture of the entire process so that the beginner is not overwhelmed by the complexity or bogged down in minutiae. Keeping this in mind, we will be designing a system that the reader may find somewhat simplistic, particularly if one compares this with the kinds of features that a 'real' system in today's market can provide. While there is some truth to this observation, it should be noted that the simplification of the system has been done with a view to reducing unnecessary detail so that we can focus instead on the development process, elaborate on the use of tools described previously, and explain through an example how good design principles are applied. In the course of applying the above, we come with a somewhat simplified *sample development process* that may be used as a template by someone who is getting started on this subject.

Assuming that we have a good grasp of the requirements, we need to document the functional requirements of the application and determine the system's major entities and their relationships. As mentioned earlier, the steps may be, and are often, carried out as an iterative, overlapping process; for pedagogical reasons, we discuss them as a sequence of distinct activities.

6.3 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. When it is time to deliver the system, there should be no confusion as to what the expectations are. Equally or perhaps even more important, it also tells the designers the expected functionality of the system. Moreover, as we attempt to create a precise documentation of the requirements, we will discover errors and omissions.

An accepted way of accomplishing this task is the *use case analysis*, which we study now.

6.3.1 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.

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

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

The above menu gives us the list of ways in which the system is going to be used. There are some implicit requirements associated with these operations. For instance,

when a book is checked out, the system must output a due-date so that the clerk can stamp the book. This and other such details will be spelled out when we elaborate on the use cases.

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. Notice that even in the case of issuing books, the functionality is invoked by a library staff member, who performs the actions on behalf of a member.

We are about to take up the task of specifying the individual use cases. In order to keep the discussion within manageable size and not lose focus, we make the following assumption: While the use cases will state the need for the system to display different messages prompting the user for data and informing the results of operations, the user community is not fussy about the minute details of what the messages should be; any meaningful message is acceptable. For example, we may specify in a use case that the system 'informs the clerk if the member was added'. The actual message could be any one of a number of possibilities such as 'Member added', or 'Member registered', etc.

Use case for registering a user Our first use case is for registering a new user and is given in Table 6.1. Recall from our discussion in Chap. 2 that use cases are specified

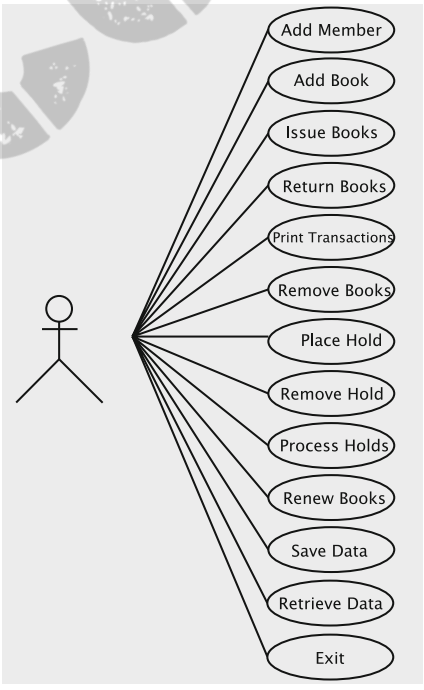


Fig. 6.1 Use case diagram for the library system

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	

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 organisation. It is important to note that not all actions and operations should be identified as use cases. As an extreme example, stamping a due-date on the book should not be a use case. A use case is a relatively large end-to-end process description that captures some business process that a client purchasing the software needs to perform. In some instances, a business process may be decomposed into more than one use case, particularly when there is some intervening real-world event(s) for which the agent has to wait for an unspecified length of time. An example of such a situation is presented later in this chapter.
3. The first step of the use case specifies a 'real-world' action that triggers the exchange described in the use case. This is provided mainly for the sake of completeness and does not have much bearing on the actual design of the system. It does, however, serve a useful purpose: by looking at the first steps of all the use cases, we can verify that all external events that the system needs to respond to have been taken care of.
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. Although we assume that the user interacts with the system through the menu, which was briefly described earlier, we do not specify the details of this mechanism. The use case also does not state how the system accomplishes the task of registering a user: what software components form the system, how they may interact, etc.

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 Next, we look at 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. Notice that details of how the identifier is generated are not specified. From the point of view of the system analyst, this is something that the actor is expected to take care of independently.

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.

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

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

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	

(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. We have resolved these issues in Step 7 by having the system check whether the book is issuable, which can be expressed as a business rule. This could check one (or more) of several conditions: *Is the member in good standing with the library? Is there some reason the book should not be checked out? Has the member checked out more books than permitted (if such limits were to be imposed)?* The message displayed by the system in Step 7 informs the clerk about the result of the transaction. In a real-life situation, the client will probably want specific details of

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	

what went wrong; if they are important to the client, these details should be expressed in the use case. Since our goal is to cover the basics of requirements analysis, we sidestep the issue.

Let us proceed to write more use cases. For the most part, these are quite elementary, and the reader may well choose to skip the details or try them out as an exercise.

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. Here, as in the use case for issuing books, the clerk may enter incorrect information into the system, which the use case handles. Notice that if there is a hold on the book, that information is printed for use by the clerk at a later time.

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). In the second of these, the system does not actually print out the transactions, but only displays them on the interface. We are assuming that the necessary facilities to print will be a part of the underlying platform.

In Step 5 in Table 6.7, we allow for the possibility that the deletion may fail. In this event, we assume that there will be some meaningful error message so that the clerk can take corrective action. We shall revisit this issue when we discuss the design and implementation in the next chapter.

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

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	

There may be some variations in the way these scenarios are played out. When placing or removing a hold, the library staff may actually want to see a message that the operation was successfully completed. These requirements would modify the manner in which the system responds in these use cases. While such information should be gleaned from the client as part of the requirements analysis, it is often necessary to go back to the client after the use cases are written, to ensure that the system interacts in the desired manner with the operator.

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

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 processing holds Given in Table 6.11, this use case deals with processing the holds at the end of each day. In this case, once the contact information for the member has been printed out, we assume that the library will contact the member. The member may not come to collect the book within the specified time, at which point the library will try to contact the next member in line. All this is not included in the use case. If we were to do so, the system would, in essence, be waiting on the user's response for a long period of time. We therefore leave out these steps and when the next user has to be contacted, we simply process holds on the book once again.

How do Business Rules Relate to Use Cases?

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

- **Definitional rules** which explain what is meant when a certain word is used in the context of the business operations. These may include special technical terms, or common words that have a particular significance for the business. For instance the term *Book* in the context of the library refers to a book owned by the library.
- **Factual rules** which explain basic things about the business's operations; they tell how the terms connect to each other. A library, for instance, would have rules such as 'Books are issued to Members,' and 'Members can place holds on Books'.
- **Constraints** which are specific conditions that govern the manner in which terms can be connected to each other. For instance, we have a constraint that says 'Holds can be placed only on Books that are currently checked out'.
- **Derivations** which are knowledge that can be derived from the facts and constraints. For instance, a bank may have the constraint, "The balance in an account cannot be less than zero," from which we can derive that if an amount requested for withdrawal is more than the balance, then the operation is not successful.

When writing use cases, we are mainly concerned with constraints and derivations. Typically, such business rules are in-lined with the logic of the use-case. The use-case may explicitly state the test that is being performed and cite the appropriate rule, or may simply mention that the system will respond in accordance with a specific rule.

In addition to the kinds of rules we have presented for this case study, there are always implicit rules that permeate the entire system. A common example of this is validation of input data; a zip code, for instance, can be validated against a database of zip-codes. Note that this rule does not deal with how entities are connected to one another, but specifies the required properties of a data element. Such constraints do not belong in use cases, but could be placed in classes that store the corresponding data elements.

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. The user may not remember the details of all of them and would perhaps like the system to prompt him/her. We shall assume that users only know the titles of the books to be renewed (they do not bring the books or even the book ids to the library) and that most users would have borrowed only a small number of books. In this situation, it is entirely appropriate for the system to display the title of each book borrowed by the user and ask if that book should be renewed.

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

It may be the case that a library has additional rules for renewability: if a book has a hold or a member has renewed a book twice, it might not be renewable. In the above interaction, the system displays all the books and determines the renewability only if the member wishes to renew the book. A different situation could arise if we require that the system display only the renewable books. (The system would have to have a way for checking renewability without actually renewing the book, which places additional requirements on the system's functionality.) For our simple library, we go with the scenario described in Table 6.5.

6.4 Defining Conceptual Classes and Relationships

As we discussed earlier, 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 establishes a relationship between them.

We could justify the usefulness of this step in at several ways:

1. **Design facilitation** Via use case analysis, we determined the functionality required of the system. Obviously, the design stage must determine how to implement the functionality. For this, the designers should be in a position to determine the classes that need to be defined, the objects to be created, and how the objects interact. This is better facilitated if the analysis phase classifies the entities in the application and determines their relationships.
2. **Added knowledge** The use cases do not completely specify the system. Some of these missing details can be filled in by the class diagram.
3. **Error reduction** In carrying out this step, the analysts are forced to look at the system more carefully. The result can be shown to the client who can verify its correctness.
4. **Useful documentation** The classes and relationships provide a quick introduction to the system for someone who wants to learn it. Such people include personnel who join the project to carry out the design or implementation or subsequent maintenance of the system.

In practice, an analyst will probably use multiple methods to come up with the conceptual classes and their relationships. 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. For example, from the text of the use case for registering new users, we can pick out the nouns.

Guidelines to Remember When Writing Use Cases

- A use case must provide something of value to an actor or to the business: when the scenario described in the use case has played out, the actor has accomplished some task. The system may have other functions that do not provide value; these will be just steps within a use case. This also implies that each use case has at least one actor.
- Use cases should be *functionally cohesive,* i.e., they encapsulate a single service that the system provides.
- Use cases should be *temporally cohesive.* This notion applies to the time frame over which the use case occurs. For instance, when a book with a hold is returned, the member who has the hold needs to be notified. The notification is done after some delay; due to this delay, we do not combine the two operations into one use case. Another example could be a university registration system—when a student registers for a class, he or she should be billed. Since the billing operation is not temporally cohesive with the registration, the two constitute separate use cases.
- If a system has multiple actors, each actor must be involved in at least one, and typically several use cases. If our library allowed members to check out books by themselves, "member" is another possible actor.
- The model that we construct is a *set* of use cases, i.e., there is no relationship between individual use cases.
- Exceptional exit conditions are not handled in use cases. For instance, if a system should crash in the middle of a use case, we do not describe what the system is supposed to do. It is assumed that some reasonable outcome will occur.
- Use cases are written from the point of view of the actor in the active voice.
- A use case describes a scenario, i.e., tells us what the visible outcome is and does not give details of any other requirements that are being imposed on the system.
- Use cases change over the course of system analysis. We are trying to construct a model and consequently the model is in a state of evolution during this process. Use cases may be merged, added or deleted from the model at any time.

Here is the text of that use case, once again, with all nouns bold-faced:

(1) The **customer** fills out an **application form** containing the **customer's name**, **address**, and **phone number** and gives this to the **clerk**. (2) The **clerk** issues a **request** to add a new **member**. (3) **The system** asks for **data** about the new **member**. (4) The **clerk** enters the **data** into the **system**. (5) Reads in **data**, and if the **member** can be added, generates an **identification number** for the **member** and remembers

information about the **member**. Informs the clerk if the member was added and outputs the **member's name**, **address**, **phone**, and **id**. (6) The **clerk** gives the **user** his **identification number**.

Let us examine the nouns. First, let us eliminate duplicates to get the following list: **customer**, **application form**, **customer's name**, **address**, **phone number**, **clerk**, **request**, **system**, **data**, **identification number**, **member**, **user**, **member information**, and **member's name**. Some of the nouns such as **member** are composite entities that qualify to be classes.

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

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`. Although we do not have as yet any specifics of this class, we note its existence and represent it in UML without any attributes and methods (Fig. 6.2). (Recall from Chap. 2 that a class is represented by a rectangle.)

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. 6.2 UML diagram for the class `Library`

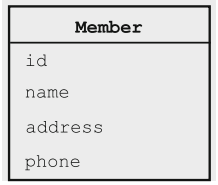


Fig. 6.3 UML diagram for the class `Member`

Recall the notion of association between classes, which we know from Chaps. 2 and 3 as a relationship between two or more classes. We note several examples of association in our case study. 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.

Obviously, members and books are the most central entities in our system: the sole reason for the library's existence is to provide service to its members and that is effected by letting them borrow books. 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. It should come as no surprise that an association between the classes `Library` and `Book`, shown in Fig. 6.6, is also needed. We show that a library has zero or more books. (Normally, you would expect a library to have at least one book and at least one member; But our design takes no chances!)

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

Fig. 6.5 UML diagram for the class `Book`

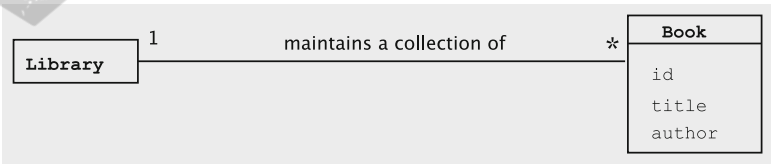


Fig. 6.6 UML diagram showing the association of `Library` and `Book`

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. At any instant in time, a book can be borrowed by one member and a member may have borrowed any number of books. We say that the relationship `Borrows` is a one-to-many relationship between the conceptual classes `Member` and `Book` and indicate it by writing 1 by the side of the box that represents a user and the * near the box that stands for a book.

This diagram actually tells us more than what the Issue Book use case does. That use case does not say some of the considerations that come into play when a user borrows a book: for example, how many books a user may borrow. We might have forgotten to ask that question when we learned about the use case. But now that we are looking at the association and are forced to put labels at the two ends, we may end up capturing missing information. 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. Recollect from Chap. 3 that an association does not imply that the objects of the classes are always linked together; we may therefore have a situation where no book in the library has been checked out.

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

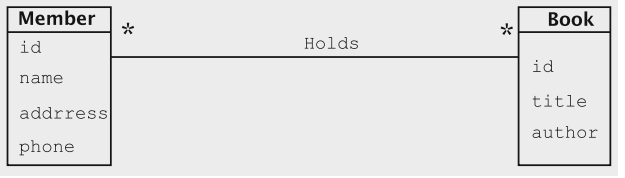


Fig. 6.8 UML diagram showing the association `Holds` between `Member` and `Book`

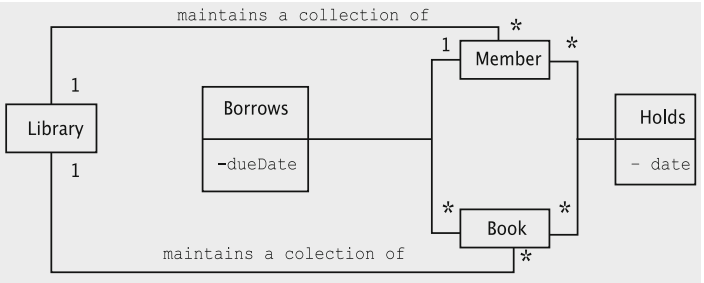


Fig. 6.9 Conceptual classes and their associations

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`. As seen before, a relationship formed between two entities is sometimes accompanied by additional information. This additional information is relevant only in the context of the relationship. There are two such examples in the inter-class relationships we have seen so far: when a user borrows a book and when a user places a hold on a book. Borrowing a book introduces new information into the system, viz., the date on which the book is due to be returned. Likewise, placing a hold introduces some information, viz., the date after which the book is not needed. The lines representing the association are augmented to represent the information that must be stored as part of the association. For the association `Borrows` and the line connecting `Member` and `Book`, we come up with a conceptual class named `Borrows` having an attribute named `dueDate`. Similarly, we create a conceptual class named `Holds` with the attribute called `date` to store the information related to the association `Holds`. Both these conceptual classes are attached to the line representing the corresponding associations.

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. For instance, there may be additional classes to support the operations of the `Library` class. We may discover that while some of the conceptual classes have corresponding physical realisations, some may disappear and the necessary information may be stored as fields distributed over multiple classes. We may discover that while some of the conceptual classes have corresponding physical realisations, some may disappear and the necessary information may be stored as fields distributed over multiple classes. We may choose to move fields that belong to an association elsewhere. For instance, the field `dueDate` may be stored as a field of the book or as a separate object, which holds a reference to the book object and the user object involved. Upon making that choice, the designer decides how the conceptual relationship between User and Book is going to be physically realised. The conceptual class diagram is simply that: **conceptual**.

6.5 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. We can sometimes divide a domain into several interrelated domains. For example, we could say that the domain of university applications includes the domain of course management, the domain of student admissions, the domain of payroll applications, and so on. Such a domain can be quite complex because of the interactions of the smaller domains that make up the bigger one.

Before we analyse 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.

Finding the Right Classes

In general, finding the right classes is non-trivial. It must be remembered that this process is iterative, i.e., we start with a set of classes and complete a conceptual design. In the process of walking through the use case implementations, we may find that some classes have to be dropped and some others have to be added. Familiarity with Design Patterns also helps in recognizing the classes. The following thumb rules and caveats come in handy:

- In general, do not build classes around functions. There are exceptions to this rule as we will see in Chap. 9. Write a class description. If it reads 'This class performs...' we most likely have a problem. If class name is imperative, e.g., print, parse, etc., it is likely that either the class is wrong or the name is wrong.
- Remember that a class usually has more than one method; otherwise it is probably a method that should be attached to some other class.
- Do not form an inheritance hierarchy too soon unless we have a pre-existing taxonomy. (Inheritance is supposed to be a relationship among well-understood abstractions.)
- Be wary of classes that have no methods, (or only query methods) because they are not frequent. Some situations in which they occur are:
  (i) representing objects from outside world, (ii) encapsulating facilities, constants or shared variables, (iii) applicative classes used to describe non-modifiable objects, e.g., integer class in Java generates new integers, but does not allow modification of integers.
- Check for the following properties of the ideal class: (i) a clearly associated abstraction, which should be a data abstraction (as opposed to a process abstraction), (ii) a descriptive noun/adjective for the class name, (iii) a non-empty set of runtime objects, (iv) queries and commands, (v) abstract properties that can be described as pre/post conditions and invariants.

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

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.

Clearly, a significant amount of effort has to be expended to domain analysis before undertaking the specific problem. The benefit is that after the initial investment of resources, the products (such as specifications, designs, code, test data, etc.) can be reused for the development of any number of applications in that domain. This reduces development time and cost.

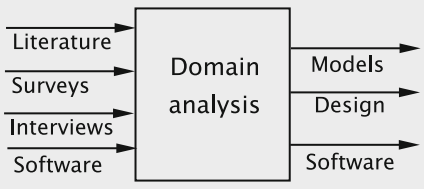


Fig. 6.10 Domain analysis

6.6 Discussion and Further Reading

A detailed treatment of object-oriented analysis methods can be found in [1]. The rules for finding the right classes are condensed from [2].

Obtaining the requirements specification is typically part of a larger '*plan and elaborate phase*' that would be an essential component of any large project. In addition to specification of requirements, this phase includes such activities as the *initial conception*, *investigation of alternatives, planning, budgeting* etc. The end product of this phase will include such documents as the *Plan* showing a schedule, resources, budget etc., a *preliminary investigation report* that lists the motivation, alternatives, and business needs, *requirements specification*, a *glossary* as an aid to understanding the vocabulary of the domain, and, perhaps, a *rough conceptual model*. Larger systems typically require more details before the analysis can proceed.

Use case modeling is one of the main techniques of a more general field of study called *usage modeling*. Usage modeling employs the following techniques: *essential use cases, system use cases*, *UML use case diagrams, user stories* and *features* [3]. What we have discussed here are essential use cases, which deal only with the fundamental business task without bringing technological issues into account. These are used to explore usage-based requirements.

Making sure that our use cases have covered all the business processes is in itself a non-trivial task. This area of study, called *business process modeling*, employs tools such as *data flow diagrams, flowcharts,* and *UML Activity Diagrams* [3] and is used to create process models for the business.

There are several UML tools available for analysis, and new variants are being constantly developed. What a practitioner chooses often depends on the development package being employed. A good, compact reference to the entire language can be found in [4]. The use case table and the class diagram with associations exemplify the very basic tools of object-oriented analysis.

There is no prescribed analysis or design technique that software designer must follow at all costs. There are several methodologies in vogue, and these ideas continue to evolve over time. In [5] it has been pointed out that while some researchers and developers are of the opinion that object-oriented methodologies are a revolutionary change from the conventional techniques, others have argued that object-oriented techniques are nothing but an elaboration of structured design. A comparative study of various object-oriented and conventional methodologies is also presented in that article.

Projects

1. **A database for a warehouse** A large warehousing corporation operates as follows:

   (a) The warehouse stocks several products, and there are several manufacturers for each product.

(b) The warehouse has a large number of registered clients. The clients place orders with the warehouse, which then ships the goods to the client. This process is as follows: the warehouse clerk examines the client's order and creates an invoice, depending on availability of the product. The invoice is then sent to the shop floor where the product is packed and shipped along with the invoice. The unfilled part of the order is placed in a waiting list queue.
(c) When the stock of any product runs low, the warehouse orders that product from one of the manufacturers, based on the price and terms of delivery.
(d) When a product shipment is received from a manufacturer, the orders in the waiting list are filled in first. The remainder is added to the inventory.

The business processes: The warehouse has three main operational business processes, namely,

(a) receiving and processing an order from a client,
(b) placing an order with the manufacturer,
(c) receiving a shipment,
(d) receiving payment from a client.

Let us examine the first of these. When an order is received from a client, the following steps are involved:

(a) Clerk receives the order and enters the order into the system.
(b) The system generates an invoice based on the availability of the product(s).
(c) The clerk prints the invoice and sends it over to the storage area.
(d) A worker on the floor picks up the invoice, retrieves the product(s) from the shelves and packs them, and ships the goods and the invoice to the client.
(e) The worker requests the system to mark the order as having been shipped.
(f) The system updates itself by recording the information.

This is an interesting business process because of the fact that steps of printing the invoice and retrieving the product from the shelves are performed by different actors. This introduces an indefinite delay into the process. If we were to translate this into a single end-to-end use case, we have a situation where the system will be waiting for a long time to get a response from an actor. It is therefore appropriate to break this up into two use cases as follows:
1. Use case create-invoice.
2. Use case fill-invoice.

In addition to these operational business processes, the warehouse will have several other querying and accounting processes such as:

(a) Registering a new client.
(b) Adding a new manufacturer for a certain product.
(c) Adding a new product.
(d) Printing a list of clients who have defaulted on payments.
(e) Printing a list of manufacturers who are owed money by the warehouse, etc.

Write the use cases, and determine the conceptual classes and their relationships.

2. **Managing a university registration system**

A small university would like to create a registration system for its students. The students will use this system to obtain information about courses, when and where the classes meet, register for classes, print transcripts, drop classes, etc. The faculty will be using this system to find out what classes they are assigned to teach, when and where these classes meet, get a list of students registered for each class, and assign grades to students in their classes. The university administrative staff will be using this database to add new faculty and students, remove faculty and students who have left, put in and update information about each course the university ofers, enter the schedules for classes that are being offered in each term, and any other housekeeping tasks that need to be performed.
Your task is to analyse this system, extract and list the details of the various business processes, develop the use cases, and find the conceptual classes and their relationships.
In finding the classes for this system, one of the issues that comes up is that of distinguishing a course from an offering of the course. For instance 'CS 430: Principles of Object-Oriented Software Construction' is a course listed in the university's course bulletin. The course is offered once during the fall term and once during the spring term. Each offering may be taught at a different time and place, and in all likelihood will have a different set of students. Therefore, all offerings have some information in common and some information that is unique to that offering. How will you choose a set of classes that models all these interactions?

3. **Creating an airline reservation and staff scheduling database**

An airline has a weekly flight schedule. Associated with each flight is an aircraft, a list of crew, and a list of passengers. The airline would like to create and maintain a database that can perform the following functions:
For passengers Add a passenger to the database, reserve a seat on a flight, print out an itinerary, request seating and meal preferences, and update frequent flier records.
For crew Assign crew members to each flight, allow crew members to view their schedule, keep track of what kinds of aircraft the crew member has been trained to operate.
For flights Keep track of crew list, passenger list, and aircraft to be used for that flight.
For aircraft Maintain all records about the aircraft and a schedule of operation.
Make an exhaustive list of queries that this system may be required to answer. Carry out a requirements analysis for the system and model it as a collection of use cases. Find the conceptual classes and their relationships.