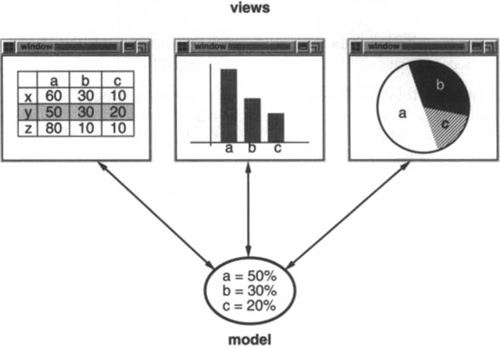
Chapter 1. Introduction



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

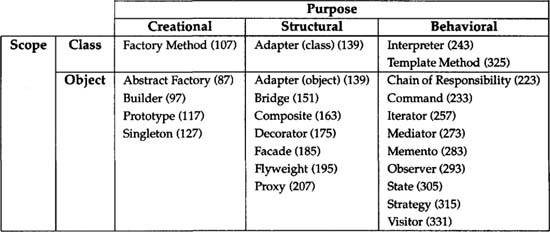
MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility 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. This example reflects a design that decouples views from models. But the design is applicable to a more general problem: decoupling objects so that changes to one can affect any number of others without requiring the changed object to know details of the others.

Another feature of MVC is that views can be nested. For example, a control panel of buttons might be implemented as a complex view containing nested button views. The user interface for an object inspector can consist of nested views that may be reused in a debugger. MVC supports nested views with the CompositeView class, a subclass of View. CompositeView objects act just like View objects; a composite view can be used wherever a view can be used, but it also contains and manages nested views.

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 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.

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



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.

**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.

**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.**

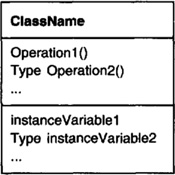
When a request is sent to an object, the particular operation that’s performed depends on both the request and the receiving object. Different objects that support identical requests may have different implementations of the operations that fulfill these requests. The run-time association of a request 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).

Moreover, dynamic binding lets you substitute objects that have identical interfaces for each other at run-time. This substitutability is known as polymorphism.

**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**](https://learning.oreilly.com/library/view/design-patterns-elements/0201633612/app01.html#gloss01_008). The class specifies the object’s internal data and representation and defines the operations the object can perform.

OMT-based notation:



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.

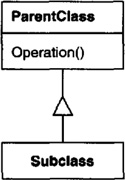
A dashed arrowhead line indicates a class that instantiates objects of another class. The arrow points to the class of the instantiated objects.

OMT-based notation

image

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:

OMT-based notation

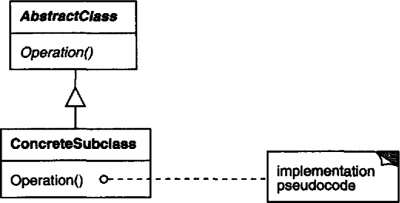


An abstract class is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated. The operations that an abstract class declares but doesn’t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

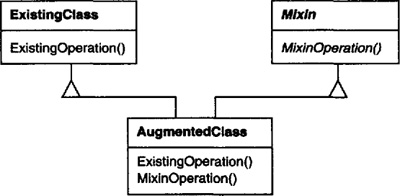
The names of abstract classes appear in slanted type to distinguish them from concrete classes. Slanted type is also used to denote abstract operations. A diagram may include pseudocode for an operation’s implementation; if so, the code will appear in a dogeared box connected by a dashed line to the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

OMT-based notation



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:

OMT-based notation



**Class versus Interface Inheritance**

The class defines the object’s internal state and the implementation of its operations. In contrast, an object’s type only refers to its interface—the set of requests to which it can respond.

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 anothe**r.

**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. 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.*

**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.”**

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.*

***Program to an interface, not an implementation.***

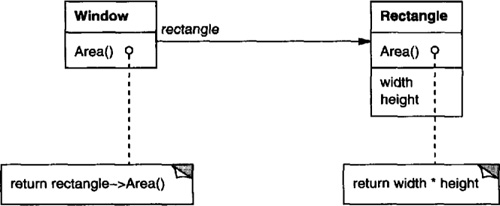
***Favor object composition over class inheritance.***

**Delegation**

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

OMT-based notation



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.

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.

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.

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.

**Relating Run-Time and Compile-Time Structures**

An object-oriented program’s run-time structure often bears little resemblance to its code structure. The code structure is frozen at compile-time; it consists of classes in fixed inheritance relationships. A program’s run-time structure consists of rapidly changing networks of communicating objects. In fact, the two structures are largely independent.

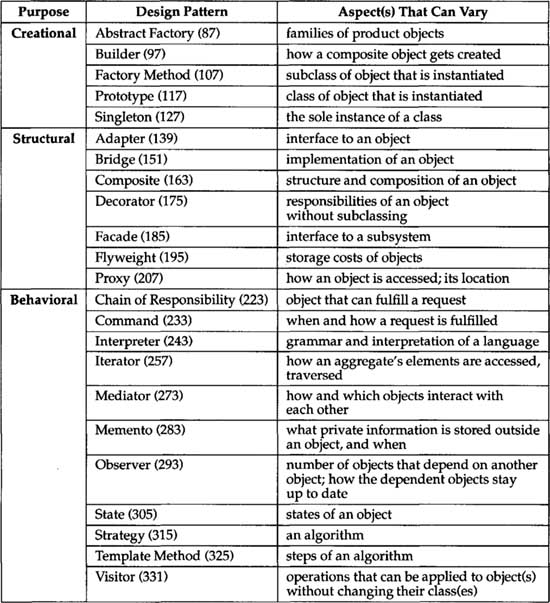
Consider the distinction between object **aggregation** and **acquaintance** and how differently they manifest themselves at compile- and run-times. Aggregation implies that one object owns or is responsible for another object. Generally, we speak of an object having or being part of another object. Aggregation implies that an aggregate object and its owner have identical lifetimes.

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

In our diagrams, a plain arrowhead line denotes acquaintance. An arrowhead line with a diamond at its base denotes aggregation (OMT-based notation):

image

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.



**Quiz:**

**How do expert programmers deal with novel problems?**

They find a good solution and apply it in similar situations.

They develop a perfect, precisely tailored solution.

They use the all-knowing search engines of the Internet.

There are no novel situations for an expert programmer.

**What are four elements of a design pattern?**

Name, Problem, Classes, Situations

Pattern Name, Problem, Solution, Consequences

Name, Situation, Solution, Principles

Earth, Air, Fire, Water