1. Introduction

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.

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]. 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 366). 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 (289) for an example.)

Design Patterns in Smalltalk MVC

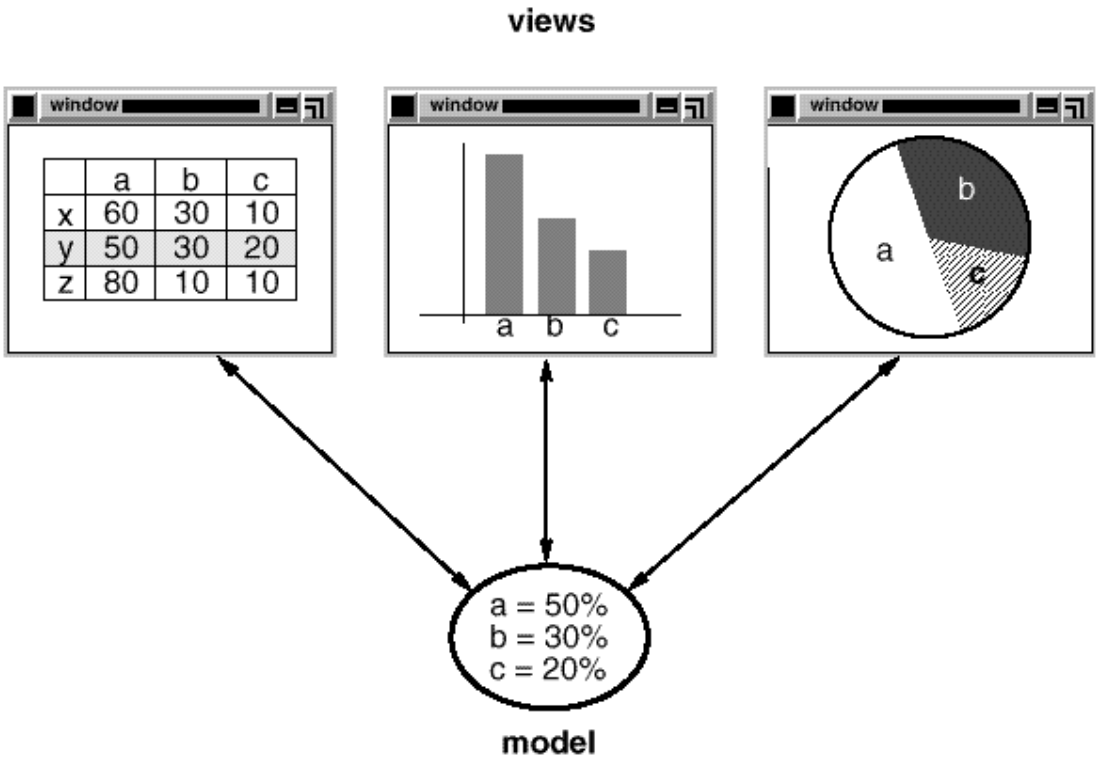
The Model/View/Controller (MVC) triad of classes [KP88] is used to build user interfaces in Smalltalk-80. Looking at the design patterns inside MVC should help you see what we mean by the term "pattern."

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 326) 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 (183) 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 (349) 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 (121) to specify the default controller class for a view and Decorator (196) to add scrolling to a view. But the main relationships in MVC are given by the Observer, Composite, and Strategy design patterns.

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.

The Catalog of Design Patterns

The catalog beginning on page 93 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 (99)

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

Adapter (157)

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 (171)

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

Builder (110)

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

Chain of Responsibility (251)

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 (263)

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

Composite (183)

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

Decorator (196)

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

Facade (208)

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 (121)

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 (218)

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

Interpreter (274)

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

Iterator (289)

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

Mediator (305)

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 (316)

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

Observer (326)

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

Prototype (133)

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

Proxy (233)

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

Singleton (144)

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

State (338)

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

Strategy (349)

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 (360)

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 (366)

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

Organizing the Catalog

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

	Purpose		
	Creational	**Structural**	**Behavioral**

Scope	**Class**	Factory Method (121)	Adapter (157)	Interpreter (274) Template Method (360)
	Object	Abstract Factory (99) Builder (110) Prototype (133) Singleton (144)	Adapter (157) Bridge (171) Composite (183) Decorator (196) Facade (208) Flyweight (218) Proxy (233)	Chain of Responsibility (251) Command (263) Iterator (289) Mediator (305) Memento (316) Observer (326) State (338) Strategy (349) Visitor (366)

Table 1.1: Design pattern space

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

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

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.

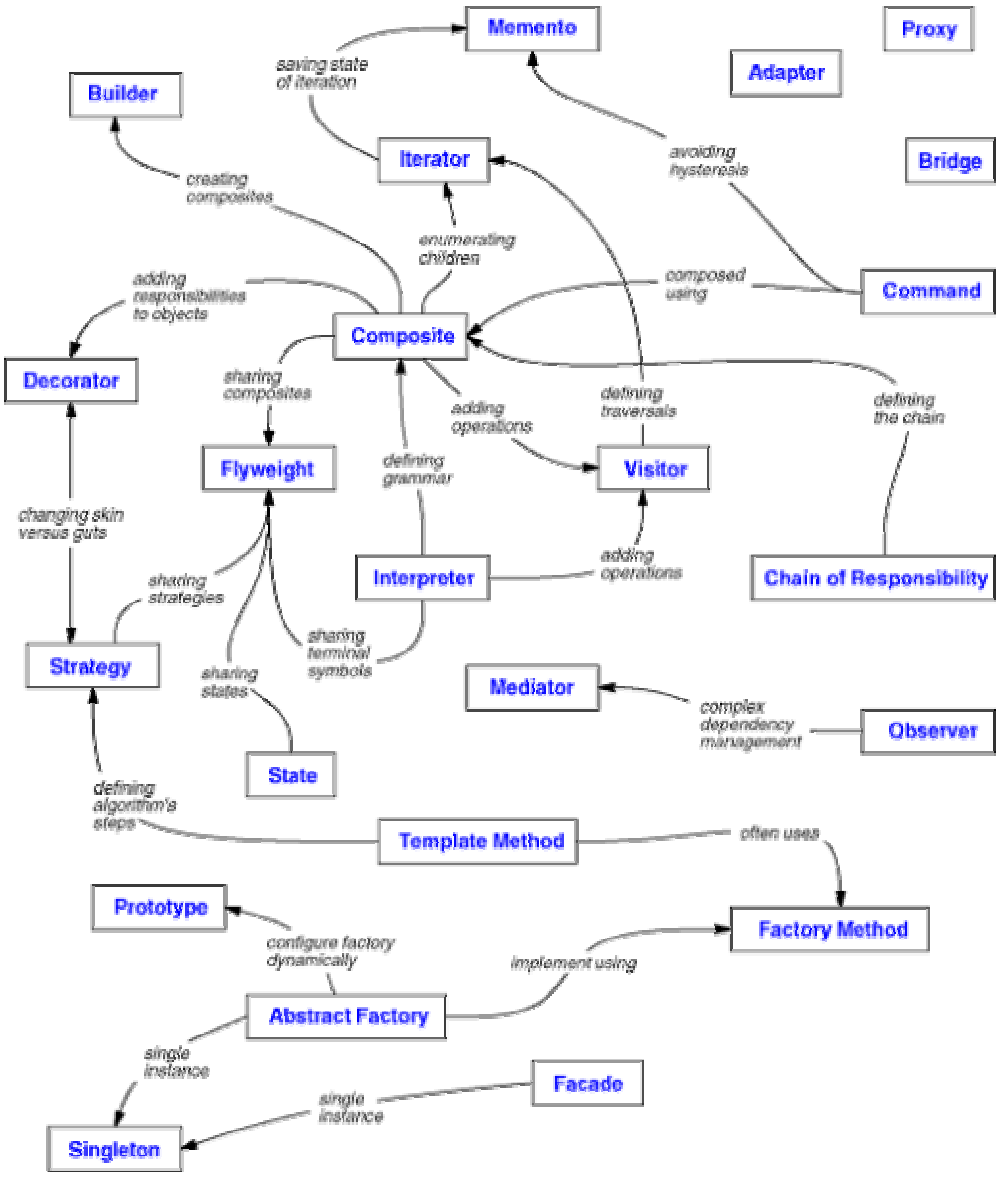


Figure 1.1: Design pattern relationships

How Design Patterns Solve Design Problems

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

Finding Appropriate Objects

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

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 (183) 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.

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 (349) pattern describes how to implement interchangeable families of algorithms. The State (338) 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 (208) pattern describes how to represent complete subsystems as objects, and the Flyweight (218) 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 (99) and Builder (110) yield objects whose only responsibilities are creating other objects. Visitor (366) and Command (263) 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.

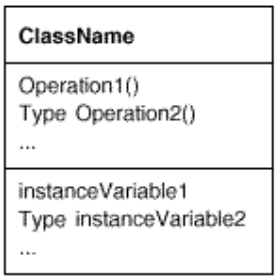
Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface. A design pattern might also tell you what *not* to put in the interface. The Memento (316) pattern is a good example. It describes how to encapsulate and save the internal 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 (196) and Proxy (233) require the interfaces of Decorator and Proxy objects to be identical to the decorated and proxied objects. In Visitor (366), 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.

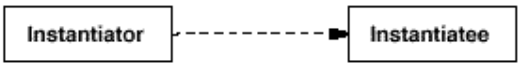
Our OMT-based notation (summarized in Appendix B) depicts a class as a rectangle with the class name in bold. Operations appear in normal type below the class name. Any data that the class defines comes after the operations. Lines separate the class name from the operations and the operations from the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



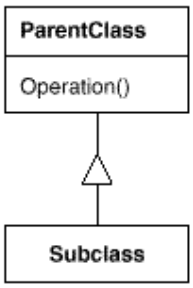
Return types and instance variable types are optional, since we don't assume a statically typed implementation language.

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:

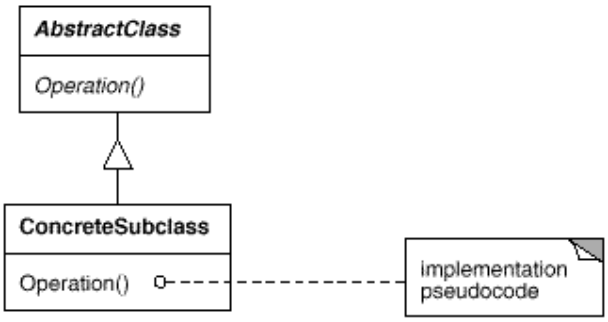


An abstract class is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to operations defined in subclasses; hence an abstract class cannot be instantiated.

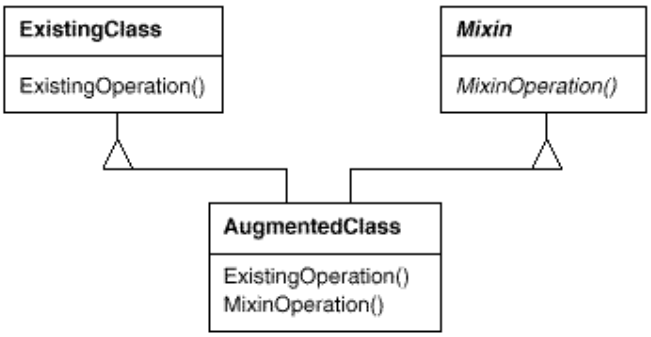
The operations that an abstract class declares but doesn't implement are called abstract operations. Classes that aren't abstract are called concrete classes.

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 (251) must have a common type, but usually they don't share a common implementation. In the Composite (183) pattern, Component defines a common interface, but Composite often defines a common implementation. Command (263), Observer (326), State (338), and Strategy (349) 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 (99), Builder (110), Factory Method (121), Prototype (133), and Singleton (144) 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.

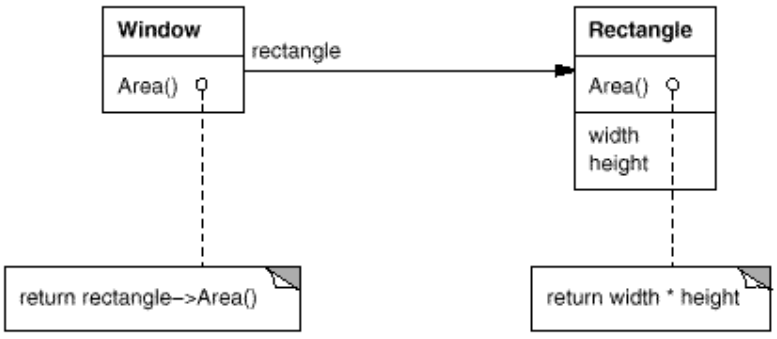
Delegation

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 (338), Strategy (349), and Visitor (366) 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 (305) 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 (251) 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 (171) 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 (360),
2. the responsibility of an object that's passed to the sorting routine (Strategy (349), 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.

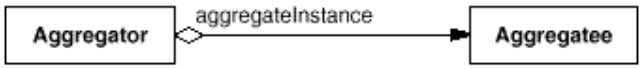
Relating Run-Time and Compile-Time Structures

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 (183) and Decorator (196) are especially useful for building complex run-time structures. Observer (326) involves run-time structures that are often hard to understand unless you know the pattern. Chain of Responsibility (251) 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 (99), Factory Method (121), Prototype (133).

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 (251), Command (263).

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 (99), Bridge (171).

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 (99), Bridge (171), Memento (316), Proxy (233).

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 (110), Iterator (289), Strategy (349), Template Method (360), Visitor (366).

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 (99), Bridge (171), Chain of Responsibility (251), Command (263), Facade (208), Mediator (305), Observer (326).

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 (171), Chain of Responsibility (251), Composite (183), Decorator (196), Observer (326), Strategy (349).

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 (157), Decorator (196), Visitor (366).

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.

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:

1. *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.
2. *Scan Intent sections.* Section 1.4 (page 18) 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 21) to narrow your search.
3. *Study how patterns interrelate.* Figure 1.1 (page 23) shows relationships between design patterns graphically. Studying these relationships can help direct you to the right pattern or group of patterns.
4. *Study patterns of like purpose.* The catalog (page 93) 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.
5. *Examine a cause of redesign.* Look at the causes of redesign starting on page 37 to see if your problem involves one or more of them. Then look at the patterns that help you avoid the causes of redesign.
6. *Consider what should be variable in your design.* This approach is the opposite of focusing on the causes of redesign. Instead of considering what might *force* a change to a design, consider what you want to be *able* to change without redesign. The focus here is on *encapsulating the concept that varies*, a theme of many design patterns. Table 1.2 lists the design aspect(s) that design patterns let you vary independently, thereby letting you change them without redesign.

Purpose	Design Pattern	Aspect(s) That Can Vary
Creational	Abstract Factory (99)	families of product objects
	Builder (110)	how a composite object gets created
	Factory Method (121)	subclass of object that is instantiated
	Prototype (133)	class of object that is instantiated
	Singleton (144)	the sole instance of a class
Structural	Adapter (157)	interface to an object
	Bridge (171)	implementation of an object
	Composite (183)	structure and composition of an object
	Decorator (196)	responsibilities of an object without subclassing
	Facade (208)	interface to a subsystem
	Flyweight (218)	storage costs of objects
	Proxy (233)	how an object is accessed; its location
Behavioral	Chain of Responsibility (251)	object that can fulfill a request
	Command (263)	when and how a request is fulfilled
	Interpreter (274)	grammar and interpretation of a language
	Iterator (289)	how an aggregate's elements are accessed, traversed
	Mediator (305)	how and which objects interact with each other
	Memento (316)	what private information is stored outside an object, and when
	Observer (326)	number of objects that depend on another object; how the dependent objects stay up to date
	State (338)	states of an object
	Strategy (349)	an algorithm
	Template Method (360)	steps of an algorithm
	Visitor (366)	operations that can be applied to object(s) without changing their class(es)

Table 1.2: Design aspects that design patterns let you vary

How to Use a Design Pattern

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

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

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. A Case Study: Design a Document Editor

This chapter presents a case study in the design of a "What-You-See-Is-What-You-Get" (or "WYSIWYG") document editor called **Lexi.**[1] We'llsee how design patterns capture solutions to design problems inLexi and applications like it. By the end of this chapter you willhave gained experience with eight patterns, learning them byexample.

Figure 2.1 depicts Lexi's user interface. AWYSIWYG representation of the document occupies the large rectangulararea in the center. The document can mix text and graphics freely ina variety of formatting styles. Surrounding the document are theusual pull-down menus and scroll bars, plus a collection of page iconsfor jumping to a particular page in the document.

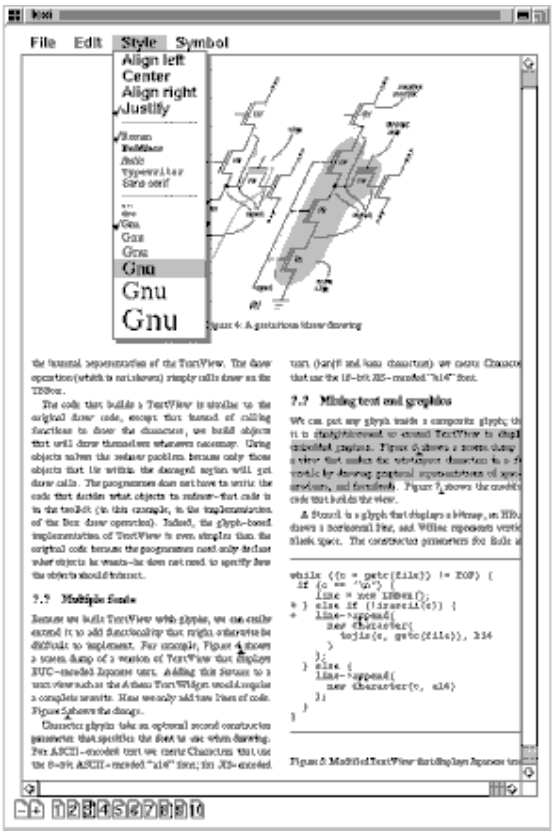


Figure 2.1: Lexi's user interface

Design Problems

We will examine seven problems in Lexi's design:

1. *Document structure.*The choice of internal representation for the document affects nearlyevery aspect of Lexi's design. All editing, formatting, displaying,and textual analysis will require traversing the representation. Theway we organize this information will impact the design of the rest ofthe application.
2. *Formatting.*How does Lexi actually arrange text and graphics into lines andcolumns? What objects are responsible for carrying out differentformatting policies? How do these policies interact with thedocument's internal representation?
3. *Embellishing the user interface.*Lexi's user interface includes scroll bars, borders, and drop shadowsthat embellish the WYSIWYG document interface. Such embellishments arelikely to change as Lexi's user interface evolves. Hence it'simportant to be able to add and remove embellishments easily withoutaffecting the rest of the application.
4. *Supporting multiple look-and-feel standards.*Lexi should adapt easily to different look-and-feel standardssuch as Motif and Presentation Manager (PM) without major modification.
5. *Supporting multiple window systems.*Different look-and-feel standards are usually implemented on differentwindow systems. Lexi's design should be as independent of the windowsystem as possible.
6. *User operations.*Users control Lexi through various user interfaces, includingbuttons and pull-down menus. The functionality behind theseinterfaces is scattered throughout the objects in the application.The challenge here is to provide a uniform mechanism both foraccessing this scattered functionality and for undoing its effects.
7. *Spelling checking and hyphenation.*How does Lexi support analytical operations such as checking formisspelled words and determining hyphenation points? How can weminimize the number of classes we have to modify to add a newanalytical operation?

We discuss these design problems in the sections that follow. Eachproblem has an associated set of goals plus constraints on how weachieve those goals. We explain the goals and constraints in detailbefore proposing a specific solution. The problem and its solutionwill illustrate one or more design patterns. The discussion for eachproblem will culminate in a brief introduction to the relevantpatterns.

Document Structure

A document is ultimately just an arrangement of basic graphicalelements such as characters, lines, polygons, and other shapes. Theseelements capture the total information content of the document. Yet anauthor often views these elements not in graphical terms but in termsof the document's physical structure—lines, columns,

figures,tables, and other substructures.[2]In turn, these substructures have substructures of theirown, and so on.

Lexi's user interface should let users manipulate thesesubstructures directly. For example, a user should be able to treat adiagram as a unit rather than as a collection of individual graphicalprimitives. The user should be able to refer to a table as a whole,not as an unstructured mass of text and graphics. That helps make theinterface simple and intuitive. To give Lexi's implementationsimilar qualities, we'll choose an internal representation thatmatches the document's physical structure.

In particular, the internal representation should support thefollowing:

- Maintaining the document's physical structure, that is, thearrangement of text and graphics into lines, columns, tables, etc.
- Generating and presenting the document visually.
- Mapping positions on the display to elements in the internalrepresentation. This lets Lexi determine what the user isreferring to when he points to something in the visual representation.

In addition to these goals are some constraints. First, we shouldtreat text and graphics uniformly. The application's interface letsthe user embed text within graphics freely and vice versa. We shouldavoid treating graphics as a special case of text or text as a specialcase of graphics; otherwise we'll end up with redundant formatting andmanipulation mechanisms. One set of mechanisms should suffice forboth text and graphics.

Second, our implementation shouldn't have to distinguish betweensingle elements and groups of elements in the internal representation.Lexi should be able to treat simple and complex elementsuniformly, thereby allowing arbitrarily complex documents. The tenthelement in line five of column two, for instance, could be a singlecharacter or an intricate diagram with many subelements. As long as weknow this element can draw itself and specify its dimensions, itscomplexity has no bearing on how and where it should appear on thepage.

Opposing the second constraint, however, is the need to analyze thetext for such things as spelling errors and potential hyphenationpoints. Often we don't care whether the element of a line is a simpleor complex object. But sometimes an analysis depends on the objectsbeing analyzed. It makes little sense, for example, to check thespelling of a polygon or to hyphenate it. The internalrepresentation's design should take this and other potentiallyconflicting constraints into account.

Recursive Composition

A common way to represent hierarchically structured information isthrough a technique called **recursive composition**, whichentails building increasingly complex elements out of simpler ones.Recursive composition gives us a way to compose a document out ofsimple graphical elements. As a first step, we can tile a set ofcharacters and graphics from left to right to form a line in thedocument. Then multiple lines can be arranged to form a column,multiple columns can form a page, and so on (seeFigure 2.2).

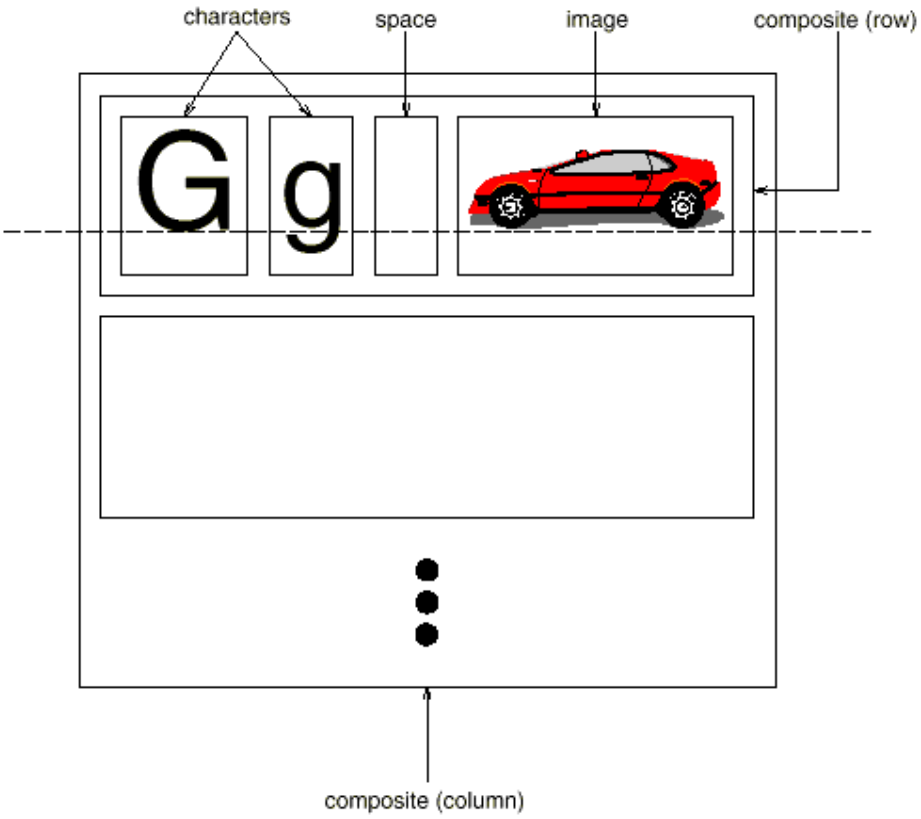


Figure 2.2: Recursive composition of text and graphics

We can represent this physical structure by devoting an object to eachimportant element. That includes not just the visible elements likethe characters and graphics but the invisible, structural elements aswell—the lines and the column. The result is the object structureshown in Figure 2.3.

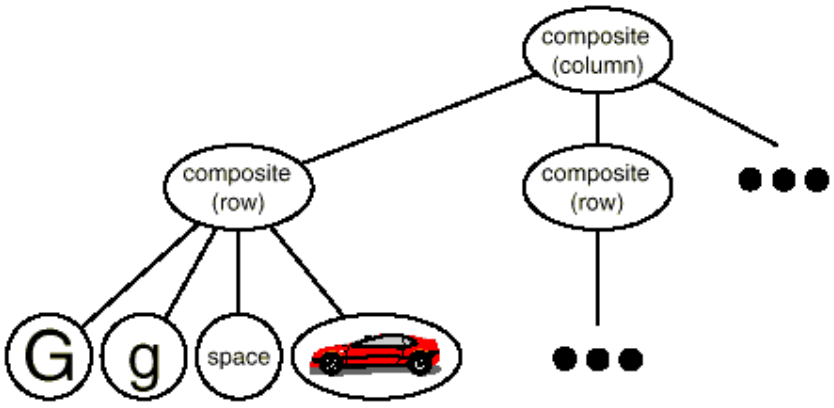


Figure 2.3: Object structure for recursive composition oftext and graphics

By using an object for each character and graphical element in thedocument, we promote flexibility at the finest levels of Lexi'sdesign. We can treat text and graphics uniformly with respect to howthey are drawn, formatted, and embedded within each other. We canextend Lexi to support new character sets without disturbing otherfunctionality. Lexi's object structure mimics the document'sphysical structure.

This approach has two important implications. The first is obvious:The objects need corresponding classes. The second implication, whichmay be less obvious, is that these classes must have compatibleinterfaces, because we want to treat the objects uniformly. The way tomake interfaces compatible in a language like C++ is to relate theclasses through inheritance.

Glyphs

We'll define a **Glyph** abstract class for allobjects that can appear in a document structure.[3] Its subclasses define bothprimitive graphical elements (like characters and images) andstructural elements (like rows and columns). Figure 2.4 depicts a representative partof the Glyph class hierarchy, and Table 2.1 presents the basic glyph interfacein more detail using C++ notation.[4]

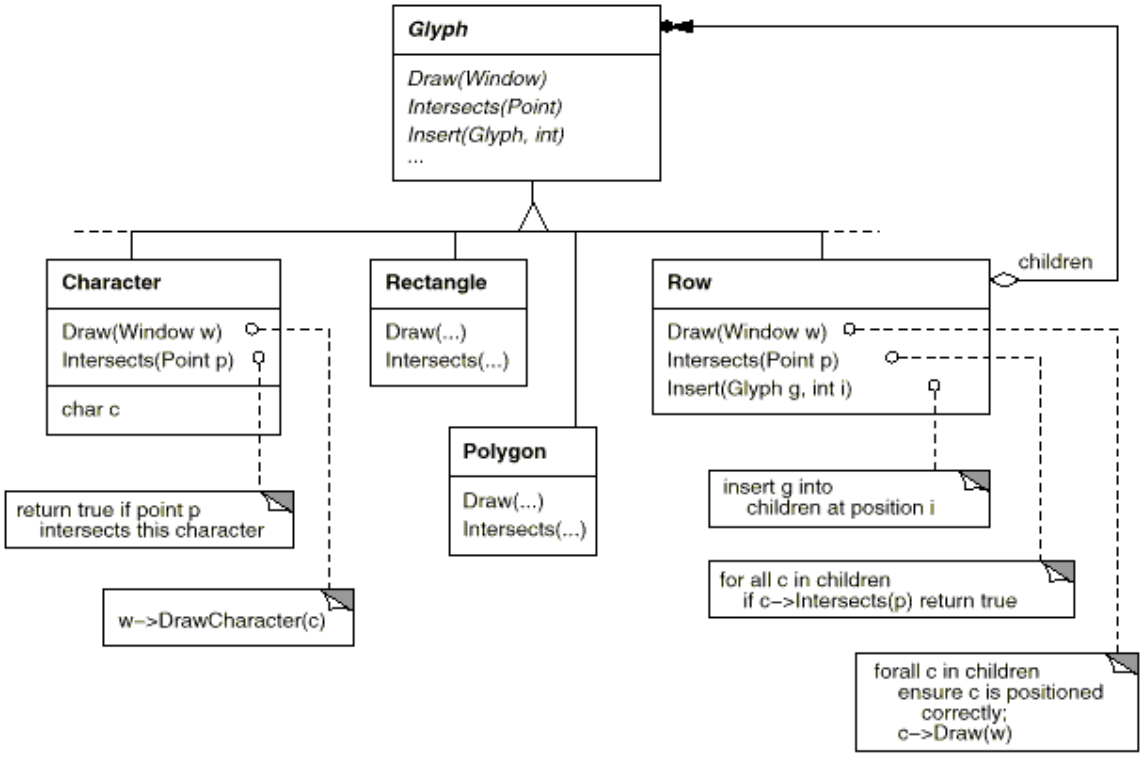


Figure 2.4: Partial Glyph class hierarchy

Responsibility	Operations
appearance	virtual void Draw(Window*) virtual void Bounds(Rect&)
hit detection	virtual bool Intersects(const Point&)
structure	virtual void Insert(Glyph*, int) virtual void Remove(Glyph*) virtual Glyph* Child(int) virtual Glyph* Parent()

Table 2.1: Basic glyph interface

Glyphs have three basic responsibilities. They know (1) how to drawthemselves, (2) what space they occupy, and (3) their children andparent.

Glyph subclasses redefine the Draw operation to renderthemselves onto a window. They are passed a reference to a Windowobject in the call to Draw. The **Window** class definesgraphics operations for rendering text and basic shapes in a window on thescreen. A **Rectangle** subclass of Glyph might redefineDraw as follows:

```
void Rectangle::Draw (Window* w) {
     w->DrawRect(_x0, _y0, _x1, _y1);
}
```

where _x0, _y0, _x1, and _y1are data members of Rectangle that define two opposing corners ofthe rectangle. DrawRect is the Window operation that makesthe rectangle appear on the screen.

A parent glyph often needs to know how much space a child glyph occupies,for example, to arrange it and other glyphs in a line so that none overlaps(as shown in Figure 2.3). TheBounds operation returns the rectangular area that the glyphoccupies. It returns the opposite corners of the smallest rectangle thatcontains the glyph. Glyph subclasses redefine this operation to return therectangular area in which they draw.

The Intersects operation returns whether a specified pointintersects the glyph. Whenever the user clicks somewhere in thedocument, Lexi calls this operation to determine which glyph orglyph structure is under the mouse. The Rectangle class redefinesthis operation to compute the intersection of the rectangle and thegiven point.

Because glyphs can have children, we need a common interface toadd, remove, and access those children. For example, a Row's childrenare the glyphs it arranges into a row. The Insertoperation inserts a glyph at a position specified by an integerindex.[5] The Removeoperation removes a specified glyph if it is indeed a child.

The Child operation returns the child (if any) at the givenindex. Glyphs like Row that can have children should use Childinternally instead of accessing the child data structure directly. That wayyou won't have to modify operations like Draw that iteratethrough the children when you change the data structure from, say, an arrayto a linked list. Similarly, Parent provides a standard interfaceto the glyph's parent, if any. Glyphs in Lexi store a reference totheir parent, and their Parent operation simply returns thisreference.

Composite Pattern

Recursive composition is good for more than just documents. We can useit to represent any potentially complex, hierarchical structure. TheComposite (183) pattern captures the essence ofrecursive composition in object-oriented terms. Now would be a goodtime to turn to that pattern and study it, referring back to thisscenario as needed.

Formatting

We've settled on a way to *represent* the document's physicalstructure. Next, we need to figure out how to construct a *particular* physical structure, one that corresponds to a properlyformatted document. Representation and formatting are distinct: Theability to capture the document's physical structure doesn't tell ushow to arrive at a particular structure. This responsibility restsmostly on Lexi. It must break text into lines, lines into columns,and so on, taking into account the user's higher-level desires. Forexample, the user might want to vary margin widths, indentation, andtabulation; single or double space; and probably many other formattingconstraints.[6]Lexi'sformatting algorithm must take all of these into account.

By the way, we'll restrict "formatting" to mean breaking a collection ofglyphs into lines. In fact, we'll use the terms "formatting" and"linebreaking" interchangeably. The techniques we'll discuss applyequally well to breaking lines into columns and to breaking columns intopages.

Encapsulating the Formatting Algorithm

The formatting process, with all its constraints and details, isn't easy toautomate. There are many approaches to the problem, and people have come upwith a variety of formatting algorithms with different strengths andweaknesses. Because Lexi is a WYSIWYG editor, an important trade-off toconsider is the balance between formatting quality and formatting speed. Wewant generally good response from the editor without sacrificing how goodthe document looks. This trade-off is subject to many factors, not all ofwhich can be ascertained at compile-time. For example, the user mighttolerate slightly slower response in exchange for better formatting. Thattrade-off might make an entirely different formatting algorithm moreappropriate than the current one. Another, more implementation-driventrade-off balances formatting speed and storage requirements: It may bepossible to decrease formatting time by caching more information.

Because formatting algorithms tend to be complex, it's also desirableto keep them well-contained or—better yet—completely independentof the document structure. Ideally we could add a new kind of Glyphsubclass without regard to the formatting algorithm. Conversely,adding a new formatting algorithm shouldn't require modifying existingglyphs.

These characteristics suggest we should design Lexi so that it'seasy to change the formatting algorithm at least at compile-time, ifnot at run-time as well. We can isolate the algorithm and make iteasily replaceable at the same time by

encapsulating it in an object.More specifically, we'll define a separate class hierarchy for objectsthat encapsulate formatting algorithms. The root of the hierarchy willdefine an interface that supports a wide range of formattingalgorithms, and each subclass will implement the interface to carryout a particular algorithm. Then we can introduce a Glyph subclassthat will structure its children automatically using a given algorithmobject.

Compositor and Composition

We'll define a **Compositor** class for objectsthat can encapsulate a formatting algorithm. The interface (Table 2.2) letsthe compositor know *what* glyphs to format and *when*to do the formatting. The glyphs it formats are the children ofa special Glyph subclass called **Composition**. Acomposition gets an instance of a Compositor subclass (specializedfor a particular linebreaking algorithm) when it is created, andit tells the compositor to Compose its glyphs whennecessary, for example, when the user changes a document.Figure 2.5 depicts the relationships between the Composition and Compositor classes.

Responsibility	Operations
what to format	void SetComposition(Composition*)
when to format	virtual void Compose()

Table 2.2 Basic compositor interface

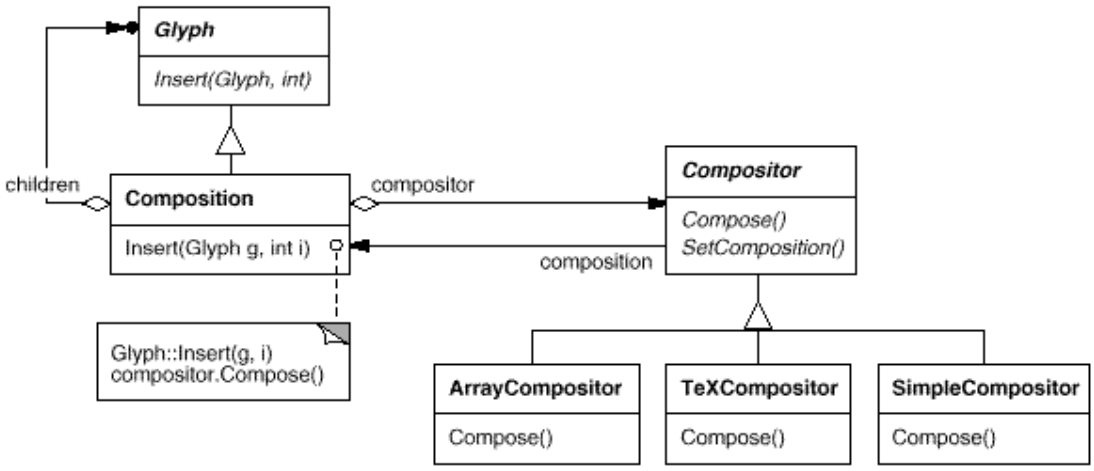


Figure 2.5: Composition and Compositor class relationships

An unformatted Composition object contains only the visibleglyphs that make up the document's basic content. It doesn't containglyphs that determine the

document's physical structure, such asRow and Column. The composition is in this state just after it'screated and initialized with the glyphs it should format. Whenthe composition needs formatting, it calls its compositor'sCompose operation. The compositor in turn iteratesthrough the composition's children and inserts new Row and Columnglyphs according to its linebreaking algorithm.[7] Figure 2.6 shows the resulting objectstructure. Glyphs that the compositor created and inserted intothe object structure appear with gray backgrounds in the figure.

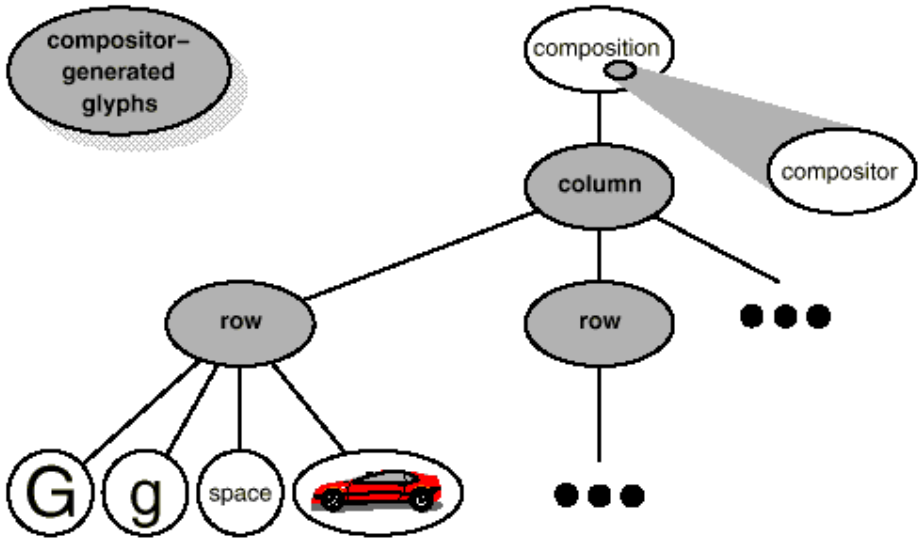


Figure 2.6: Object structure reflectingcompositor-directed linebreaking

Each Compositor subclass can implement a different linebreaking algorithm.For example, a SimpleCompositor might do a quick pass without regard forsuch esoterica as the document's "color." Good color means having an evendistribution of text and whitespace. A TeXCompositor would implement thefull TeX algorithm [Knu84], which takes things like color into accountin exchange for longer formatting times.

The Compositor-Composition class split ensures a strong separationbetween code that supports the document's physical structure and thecode for different formatting algorithms. We can add new Compositorsubclasses without touching the glyph classes, and vice versa. Infact, we can change the linebreaking algorithm at run-time by adding asingle SetCompositor operation to Composition's basic glyphinterface.

Strategy Pattern

Encapsulating an algorithm in an object is the intent of the Strategy (349) pattern. The key participants in thepattern are Strategy objects (which encapsulate different algorithms)and the context in which they operate. Compositors are

strategies;they encapsulate different formatting algorithms. A composition is thecontext for a compositor strategy.

The key to applying the Strategy pattern is designing interfaces forthe strategy and its context that are general enough to support arange of algorithms. You shouldn't have to change the strategy orcontext interface to support a new algorithm. In our example, thebasic Glyph interface's support for child access, insertion, andremoval is general enough to let Compositor subclasses change thedocument's physical structure, regardless of the algorithm they use todo it. Likewise, the Compositor interface gives compositions whateverthey need to initiate formatting.

Embellishing the User Interface

We consider two embellishments in Lexi's user interface. Thefirst adds a border around the text editing area to demarcate the pageof text. The second adds scroll bars that let the user view differentparts of the page. To make it easy to add and remove theseembellishments (especially at run-time), we shouldn't use inheritanceto add them to the user interface. We achieve the most flexibilityif other user interface objects don't even know the embellishments arethere. That will let us add and remove the embellishments withoutchanging other classes.

Transparent Enclosure

From a programming point of view, embellishing the user interface involves extending existing code. Using inheritance to do such extension precludesrearranging embellishments at run-time, but an equally serious problem is the explosion of classes that can result from an inheritance-basedapproach.

We could add a border to Composition by subclassing it to yield aBorderedComposition class. Or we could add a scrolling interface inthe same way to yield a ScrollableComposition. If we want both scrollbars and a border, we might produce a BorderedScrollableComposition,and so forth. In the extreme, we end up with a class for everypossible combination of embellishments, a solution that quicklybecomes unworkable as the variety of embellishments grows.

Object composition offers a potentially more workable and flexibleextension mechanism. But what objects do we compose? Since we knowwe're embellishing an existing glyph, we could make the embellishmentitself an object (say, an instance of class **Border**). Thatgives us two candidates for composition, the glyph and the border. Thenext step is to decide who composes whom. We could have the bordercontain the glyph, which makes sense given that the border willsurround the glyph on the screen. Or we could do the opposite—putthe border into the glyph—but then we must

make modifications to thecorresponding Glyph subclass to make it aware of the border. Our firstchoice, composing the glyph in the border, keeps the border-drawingcode entirely in the Border class, leaving other classes alone.

What does the Border class look like? The fact that borders have anappearance suggests they should actually be glyphs; that is, Bordershould be a subclass of Glyph. But there's a more compelling reasonfor doing this: Clients shouldn't care whether glyphs have borders ornot. They should treat glyphs uniformly. When clients tell a plain,unbordered glyph to draw itself, it should do so withoutembellishment. If that glyph is composed in a border, clientsshouldn't have to treat the border containing the glyph anydifferently; they just tell it to draw itself as they told the plainglyph before. This implies that the Border interface matches the Glyphinterface. We subclass Border from Glyph to guarantee thisrelationship.

All this leads us to the concept of **transparent enclosure**,which combines the notions of (1) single-child (orsingle-**component**) composition and (2) compatibleinterfaces. Clients generally can't tell whether they're dealing withthe component or its **enclosure** (i.e., the child's parent),especially if the enclosure simply delegates all its operations to itscomponent. But the enclosure can also *augment* the component'sbehavior by doing work of its own before and/or after delegating anoperation. The enclosure can also effectively add state to thecomponent. We'll see how next.

Monoglyph

We can apply the concept of transparent enclosure to all glyphs thatembellish other glyphs. To make this concept concrete, we'll define asubclass of Glyph called **MonoGlyph** to serve as an abstractclass for "embellishment glyphs," likeBorder (see Figure 2.7).MonoGlyph stores a reference to a component and forwards all requests toit. That makes MonoGlyph totally transparent to clients by default.For example, MonoGlyph implements the Draw operation like this:

```
void MonoGlyph::Draw (Window* w) {
      _component->Draw(w);
}
```

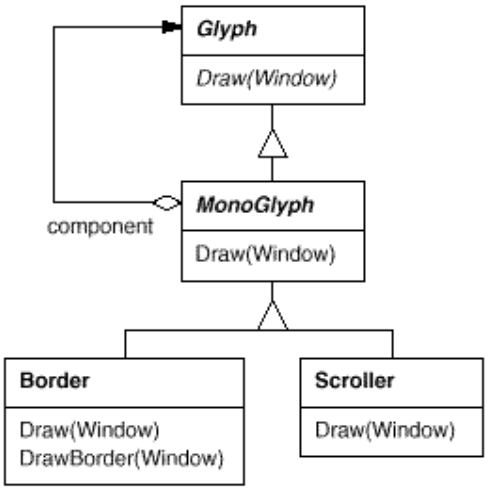


Figure 2.7: MonoGlyph class relationships

MonoGlyph subclasses reimplement at least one of these forwardingoperations. Border::Draw, for instance, first invokes the parentclass operation MonoGlyph::Draw on the component to let thecomponent do its part—that is, draw everything but the border. ThenBorder::Draw draws the border by calling a privateoperation called DrawBorder, the details of which we'llomit:

```
void Border::Draw (Window* w) {
        MonoGlyph::Draw(w);
        DrawBorder(w);
}
```

Notice how Border::Draw effectively *extends* the parentclass operation to draw the border. This is in contrast to merely*replacing* the parent class operation, which would omit the call toMonoGlyph::Draw.

Another MonoGlyph subclass appears in Figure 2.7. **Scroller** is a MonoGlyph that draws its component in differentlocations based on the positions of two scroll bars, which it adds asembellishments. When Scroller draws its component, it tells thegraphics system to clip to its bounds. Clipping parts of the componentthat are scrolled out of view keeps them from appearing on the screen.

Now we have all the pieces we need to add a border and a scrollinginterface to Lexi's text editing area. We compose the existingComposition instance in a Scroller instance to add the scrollinginterface, and we compose that in a Border instance. The resultingobject structure appears in Figure 2.8.

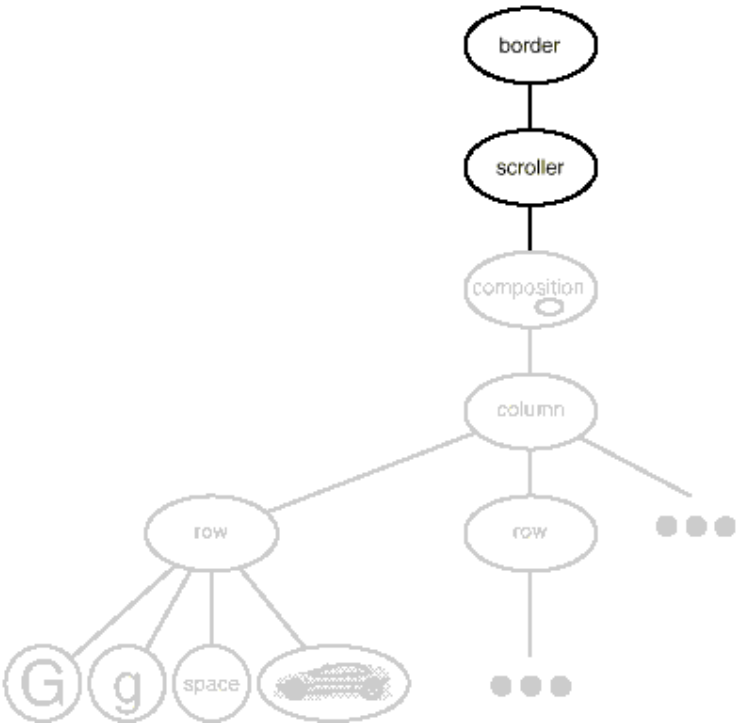


Figure 2.8: Embellished object structure

Note that we can reverse the order of composition, putting thebordered composition into the Scroller instance. In that case theborder would be scrolled along with the text, which may or may not bedesirable. The point is, transparent enclosure makes it easy toexperiment with different alternatives, and it keeps clients free ofembellishment code.

Note also how the border composes one glyph, not two or more. This isunlike compositions we've defined so far, in which parent objects wereallowed to have arbitrarily many children. Here, putting a borderaround something implies that "something" is singular. We couldassign a meaning to embellishing more than one object at a time, butthen we'd have to mix many kinds of composition in with the notion ofembellishment: row embellishment, column embellishment, and so forth.That won't help us, since we already have classes to do those kinds ofcompositions. So it's better to use existing classes for compositionand add new classes to embellish the result. Keeping embellishmentindependent of other kinds of composition both simplifies theembellishment classes and reduces their number. It also keeps us fromreplicating existing composition functionality.

Decorator Pattern

The Decorator (196) pattern captures class and objectrelationships that support embellishment by transparent enclosure.The term "embellishment" actually has

broader meaning than whatwe've considered here. In the Decorator pattern, embellishment refersto anything that adds responsibilities to an object. We can thinkfor example of embellishing an abstract syntax tree with semanticactions, a finite state automaton with new transitions, or a networkof persistent objects with attribute tags. Decorator generalizes theapproach we've used in Lexi to make it more widely applicable.

Supporting Multiple Look-and-Feel Standards

Achieving portability across hardware and software platforms is amajor problem in system design. Retargeting Lexi to a newplatform shouldn't require a major overhaul, or it wouldn't be worthretargeting. We should make porting as easy as possible.

One obstacle to portability is the diversity of look-and-feel standards,which are intended to enforce uniformity between applications. Thesestandards define guidelines for how applications appear and react to theuser. While existing standards aren't that different from each other,people certainly won't confuse one for the other—Motif applications don'tlook and feel exactly like their counterparts on other platforms, and viceversa. An application that runs on more than one platform must conform tothe user interface style guide on each platform.

Our design goals are to make Lexi conform to multiple existinglook-and-feel standards and to make it easy to add support for newstandards as they (invariably) emerge. We also want our design tosupport the ultimate in flexibility: changing Lexi's look and feelat run-time.

Abstracting Object Creation

Everything we see and interact with in Lexi's user interface is aglyph composed in other, invisible glyphs like Row and Column. Theinvisible glyphs compose visible ones like Button and Character and laythem out properly. Style guides have much to say about the look andfeel of so-called "widgets," another term for visible glyphs likebuttons, scroll bars, and menus that act as controlling elements in auser interface. Widgets might use simpler glyphs such as characters,circles, rectangles, and polygons to present data.

We'll assume we have two sets of widget glyph classes with which toimplement multiple look-and-feel standards:

1. A set of abstract Glyph subclasses for each category of widgetglyph. For example, an abstract class ScrollBar will augment the basicglyph interface

to add general scrolling operations; Button is anabstract class that adds button-oriented operations; and so on.

2. A set of concrete subclasses for each abstract subclass thatimplement different look-and-feel standards. For example, ScrollBarmight have MotifScrollBar and PMScrollBar subclasses that implementMotif and Presentation Manager-style scroll bars, respectively.

Lexi must distinguish between widget glyphs for different look-and-feelstyles. For example, when Lexi needs to put a button in its interface,it must instantiate a Glyph subclass for the right style of button(MotifButton, PMButton, MacButton, etc.).

It's clear that Lexi's implementation can't do this directly, say,using a constructor call in C++. That would hard-code the button of aparticular style, making it impossible to select the style atrun-time. We'd also have to track down and change every suchconstructor call to port Lexi to another platform. And buttons areonly one of a variety of widgets in Lexi's user interface.Littering our code with constructor calls to specific look-and-feelclasses yields a maintenance nightmare—miss just one, and you couldend up with a Motif menu in the middle of your Mac application.

Lexi needs a way to determine the look-and-feel standard that's beingtargeted in order to create the appropriate widgets. Not only must weavoid making explicit constructor calls; we must also be able toreplace an entire widget set easily. We can achieve both by *abstracting the process of object creation*. An example willillustrate what we mean.

Factories and Product Classes

Normally we might create an instance of a Motif scroll bar glyph with thefollowing C++ code:

```
ScrollBar* sb = new MotifScrollBar;
```

This is the kind of code to avoid if you want to minimizeLexi's look-and-feel dependencies. But suppose weinitialize sb as follows:

```
ScrollBar* sb = guiFactory->CreateScrollBar();
```

where guiFactory is an instance of a**MotifFactory** class. CreateScrollBarreturns a new instance of the proper ScrollBar subclass for thelook and feel desired, Motif in this case. As far as clients areconcerned, the effect is the same as calling the MotifScrollBarconstructor directly. But there's a crucial difference: There'sno longer anything in the code that mentions Motif by name. TheguiFactory

object abstracts the process of creatingnot just Motif scroll bars but scroll bars for *any*look-and-feel standard. And guiFactory isn't limitedto producing scroll bars. It can manufacture a full range of widgetglyphs, including scroll bars, buttons, entry fields, menus, andso forth.

All this is possible because MotifFactory is a subclass of**GUIFactory**, an abstract class that defines ageneral interface for creating widget glyphs. It includes operationslike CreateScrollBar and CreateButtonfor instantiating different kinds of widget glyphs. Subclasses ofGUIFactory implement these operations to return glyphs such asMotifScrollBar and PMButton that implement a particular look andfeel. Figure 2.9 showsthe resulting class hierarchy for guiFactory objects.

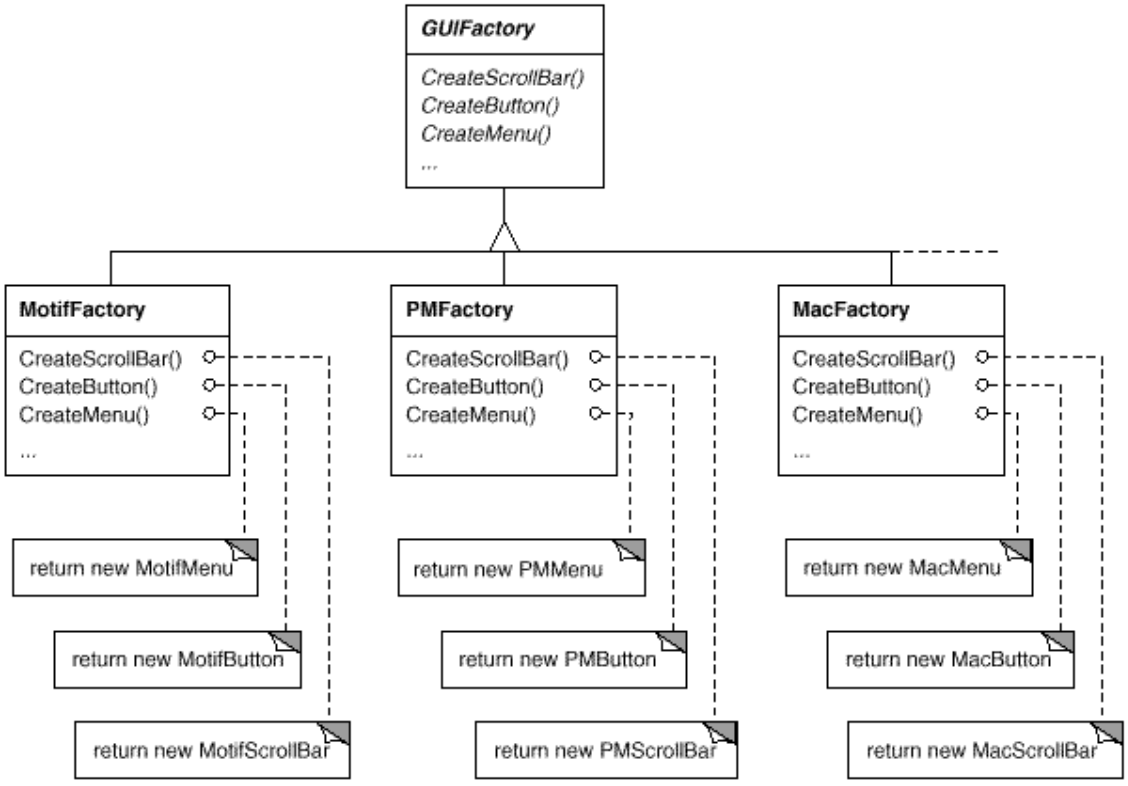


Figure 2.9: GUIFactory class hierarchy

We say that factories create **product** objects.Moreover, the products that a factory produces are related to oneanother; in this case, the products are all widgets for the samelook and feel. Figure 2.10 shows some of the product classes needed to make factories workfor widget glyphs.

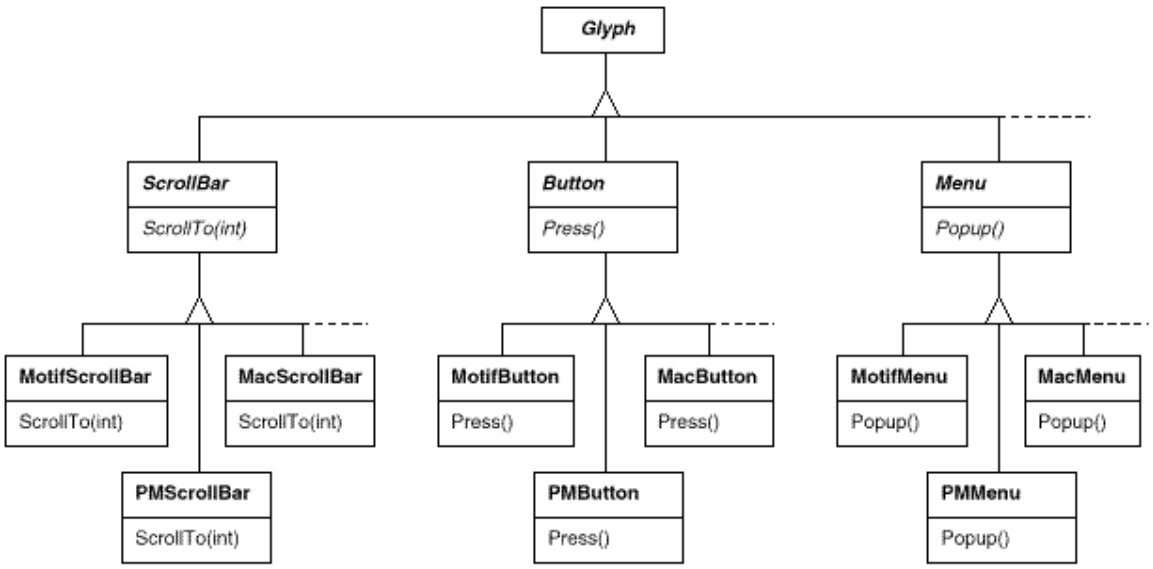


Figure 2.10: Abstract product classes and concrete subclasses

The last question we have to answer is, Where does the GUIFactoryinstance come from? The answer is, Anywhere that's convenient. Thevariable guiFactory could be a global, a static member of awell-known class, or even a local variable if the entire user interface iscreated within one class or function. There's even a design pattern,Singleton (144), for managing well-known, one-of-a-kindobjects like this. The important thing, though, is to initializeguiFactory at a point in the program *before* it's ever usedto create widgets but *after* it's clear which look and feel isdesired.

If the look and feel is known at compile-time, then guiFactorycan be initialized with a simple assignment of a new factory instanceat the beginning of the program:

```
GUIFactory* guiFactory = new MotifFactory;
```

If the user can specify the look and feel with a string name atstartup time, then the code to create the factory might be

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
       // user or environment supplies this at startup
if (strcmp(styleName, "Motif") == 0) {
       guiFactory = new MotifFactory;
} else if (strcmp(styleName, "Presentation_Manager") == 0) {
       guiFactory = new PMFactory;
```

```
} else {
      guiFactory = new DefaultGUIFactory;
}
```

There are more sophisticated ways to select the factory at run-time.For example, you could maintain a registry that maps strings tofactory objects. That lets you register instances of new factorysubclasses without modifying existing code, as the preceding approach requires. And you don't have to link all platform-specific factoriesinto the application. That's important, because it might not bepossible to link a MotifFactory on a platform that doesn't supportMotif.

But the point is that once we've configured the application with theright factory object, its look and feel is set from then on. If wechange our minds, we can reinitialize guiFactory with afactory for a different look and feel and then reconstruct theinterface. Regardless of how and when we decide to initializeguiFactory, we know that once we do, the application cancreate the appropriate look and feel without modification.

Abstract Factory Pattern

Factories and products are the key participants in the Abstract Factory (99) pattern. This pattern captures howto create families of related product objects without instantiatingclasses directly. It's most appropriate when the number and generalkinds of product objects stay constant, and there are differences inspecific product families. We choose between families by instantiatinga particular concrete factory and using it consistently to createproducts thereafter. We can also swap entire families of products byreplacing the concrete factory with an instance of a different one.The Abstract Factory pattern's emphasis on *families* of productsdistinguishes it from other creational patterns, which involve only onekind of product object.

Supporting Multiple Window Systems

Look and feel is just one of many portability issues. Another is thewindowing environment in which Lexi runs. A platform's window systemcreates the illusion of multiple overlapping windows on a bitmappeddisplay. It manages screen space for windows and routes input to them fromthe keyboard and mouse. Several important and largely incompatible windowsystems exist today (e.g., Macintosh, Presentation Manager, Windows, X).We'd like Lexi to run on as many of them as possible for exactly thesame reasons we support multiple look-and-feel standards.

Can We Use an Abstract Factory?

At first glance this may look like another opportunity to apply theAbstract Factory pattern. But the constraints for window system portabilitydiffer significantly from those for look-and-feel independence.

In applying the Abstract Factory pattern, we assumed we would definethe concrete widget glyph classes for each look-and-feel standard.That meant we could derive each concrete product for a particularstandard (e.g., MotifScrollBar and MacScrollBar) from an abstractproduct class (e.g., ScrollBar). But suppose we already have severalclass hierarchies from different vendors, one for each look-and-feelstandard. Of course, it's highly unlikely these hierarchies arecompatible in any way. Hence we won't have a common abstract productclass for each kind of widget (ScrollBar, Button, Menu, etc.)—and theAbstract Factory pattern won't work without those crucial classes.We have to make the different widget hierarchies adhere to a commonset of abstract product interfaces. Only then could we declare theCreate... operations properly in our abstract factory'sinterface.

We solved this problem for widgets by developing our own abstract andconcrete product classes. Now we're faced with a similar problem whenwe try to make Lexi work on existing window systems; namely,different window systems have incompatible programming interfaces.Things are a bit tougher this time, though, because we can't afford toimplement our own nonstandard window system.

But there's a saving grace. Like look-and-feel standards, windowsystem interfaces aren't radically different from one another, becauseall window systems do generally the same thing. We need a uniform setof windowing abstractions that lets us take different window systemimplementations and slide any one of them under a common interface.

Encapsulating Implementation Dependencies

In Section 2.2 we introduced a Window class for displaying a glyph or glyphstructure on the display. We didn't specify the window system thatthis object worked with, because the truth is that it doesn't comefrom any particular window system. The Window class encapsulatesthe things windows tend to do across window systems:

- They provide operations for drawing basic geometric shapes.
- They can iconify and de-iconify themselves.
- They can resize themselves.
- They can (re)draw their contents on demand, for example, when theyare de-iconified or when an overlapped and obscured portion of theirscreen space is exposed.

The Window class must span the functionality of windows from differentwindow systems. Let's consider two extreme philosophies:

1. *Intersection of functionality.*The Window class interface provides only functionality that's commonto *all* window systems. The problem with this approach is thatour Window interface winds up being only as powerful as the leastcapable window system. We can't take advantage of more advancedfeatures even if most (but not all) window systems support them.
2. *Union of functionality.*Create an interface that incorporates the capabilities of *all*existing systems. The trouble here is that the resulting interface maywell be huge and incoherent. Besides, we'll have to change it (andLexi, which depends on it) anytime a vendor revises its windowsystem interface.

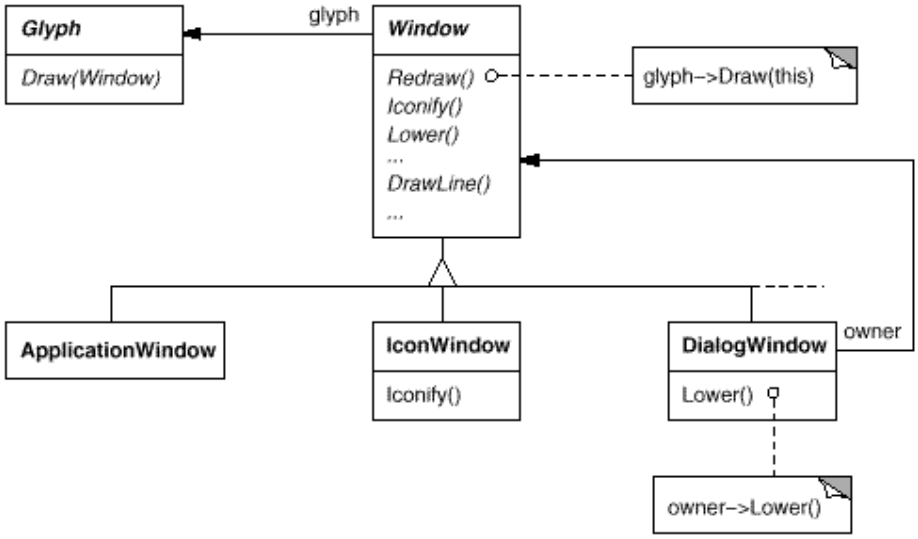
Neither extreme is a viable solution, so our design will fallsomewhere between the two. The Window class will provide a convenientinterface that supports the most popular windowing features. BecauseLexi will deal with this class directly, the Window class must alsosupport the things Lexi knows about, namely, glyphs. That meansWindow's interface must include a basic set of graphics operationsthat lets glyphs draw themselves in the window.Table 2.3 gives a sampling of the operations in the Window class interface.

Responsibility	Operations
window management	virtual void Redraw() virtual void Raise() virtual void Lower() virtual void Iconify() virtual void Deiconify() ...
graphics	virtual void DrawLine(...) virtual void DrawRect(...) virtual void DrawPolygon(...) virtual void DrawText(...) ...

Table 2.3: Window class interface

Window is an abstract class. Concrete subclasses of Window support thedifferent kinds of windows that users deal with. For example,application windows, icons, and warning dialogs are all windows, butthey have somewhat different behaviors. So we can define subclasseslike ApplicationWindow, IconWindow, and DialogWindow to capture thesedifferences. The resulting class hierarchy gives applications

likeLexi a uniform and intuitive windowing abstraction, one that doesn'tdepend on any particular vendor's window system:



Now that we've defined a window interface for Lexi to work with,where does the real platform-specific window come in? If we're notimplementing our own window system, then at some point our windowabstraction must be implemented in terms of what the target windowsystem provides. So where does that implementation live?

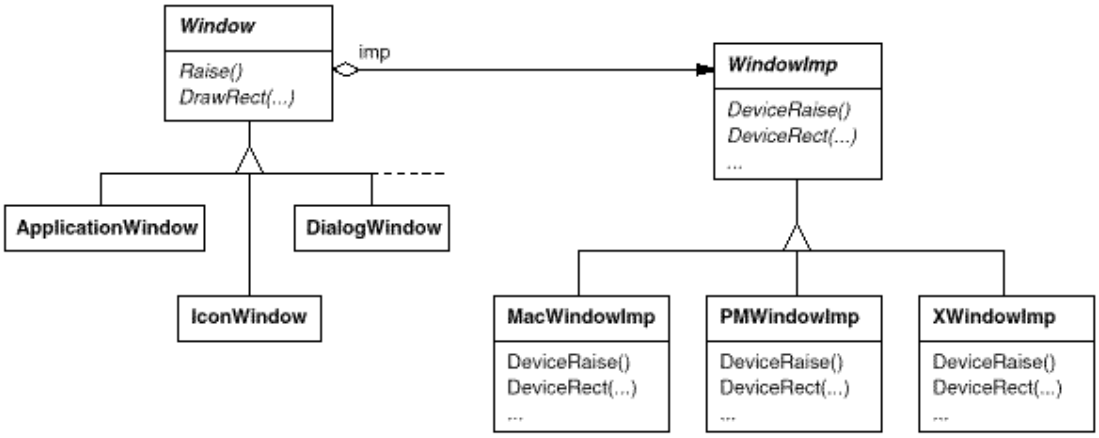
One approach is to implement multiple versions of the Window class andits subclasses, one version for each windowing platform. We'd have tochoose the version to use when we build Lexi for a given platform.But imagine the maintenance headaches we'd have keeping track ofmultiple classes, all named "Window" but each implemented on adifferent window system. Alternatively, we could createimplementation-specific subclasses of each class in the Windowhierarchy—and end up with another subclass explosion problem like the onewe had trying to add embellishments. Both of these alternatives haveanother drawback: Neither gives us the flexibility to change thewindow system we use after we've compiled the program. So we'll haveto keep several different executables around as well.

Neither alternative is very appealing, but what else can we do? Thesame thing we did for formatting and embellishment, namely, *encapsulate the concept that varies*. What varies in this case is thewindow system implementation. If we encapsulate a window system'sfunctionality in an object, then we can implement our Window class andsubclasses in terms of that object's interface. Moreover, if thatinterface can serve all the window systems we're interested in, thenwe won't have to change Window or any of its subclasses to supportdifferent window systems. We can configure window objects to thewindow system we want simply by

passing them the right windowsystem-encapsulating object. We can even configure the window atrun-time.

Window and WindowImp

We'll define a separate **WindowImp** class hierarchy in which tohide different window system implementations. WindowImp is an abstractclass for objects that encapsulate window system-dependent code. To makeLexi work on a particular window system, we configure each windowobject with an instance of a WindowImp subclass for that system. Thefollowing diagram shows the relationship between the Window and WindowImphierarchies:



By hiding the implementations in WindowImp classes, we avoid pollutingthe Window classes with window system dependencies, which keeps theWindow class hierarchy comparatively small and stable. Meanwhile wecan easily extend the implementation hierarchy to support new windowsystems.

WindowImp Subclasses

Subclasses of WindowImp convert requests into window system-specificoperations. Consider the example we used in Section 2.2. We defined theRectangle::Draw in terms of the DrawRect operation onthe Window instance:

```
void Rectangle::Draw (Window* w) {
      w->DrawRect(_x0, _y0, _x1, _y1);
}
```

The default implementation of DrawRect uses the abstractoperation for drawing rectangles declared by WindowImp:

```
void Window::DrawRect ( Coord x0, Coord y0, Coord x1, Coord y1    )
{       _imp->DeviceRect(x0, y0, x1, y1);    }
```

where _imp is a member variable of Window that stores theWindowImp with which the Window is configured. The windowimplementation is defined by the instance of the WindowImp subclassthat _imp points to. For an XWindowImp (that is, aWindowImp subclass for the X Window System), theDeviceRect's implementation might look like

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

DeviceRect is defined like this becauseXDrawRectangle (the X interface for drawing a rectangle)defines a rectangle in terms of its lower left corner, its width,and its height. DeviceRect must compute these valuesfrom those supplied. First it ascertains the lower left corner(since (x0, y0) might be any oneof the rectangle's four corners) and then calculates the width andheight.

PMWindowImp (a subclass of WindowImp for Presentation Manager) would define DeviceRect differently:

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

          PPOINTL point[4];
          point[0].x = left; point[0].y = top;
          point[1].x = right;point[1].y = top;
          point[2].x = right;point[2].y = bottom;
          point[3].x = left; point[3].y = bottom;
          if ( (GpiBeginPath(_hps, 1L) == false) ||
(GpiSetCurrentPosition(_hps, &point[3]) == false) ||
(GpiPolyLine(_hps, 4L, point) == GPI_ERROR)  ||
(GpiEndPath(_hps) == false) )
{
```

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

Why is this so different from the X version? Well, PM doesn't have anoperation for drawing rectangles explicitly as X does. Instead, PM has amore general interface for specifying vertices of multisegment shapes(called a **path**) and for outlining or filling the area theyenclose.

PM's implementation of DeviceRect is obviously quitedifferent from X's, but that doesn't matter. WindowImp hidesvariations in window system interfaces behind a potentially large butstable interface. That lets Window subclass writers focus on the windowabstraction and not on window system details. It also lets us addsupport for new window systems without disturbing the Window classes.

Configuring Windows with WindowImps

A key issue we haven't addressed is how a window gets configured withthe proper WindowImp subclass in the first place. Stated another way,when does _imp get initialized, and who knows what windowsystem (and consequently which WindowImp subclass) is in use? Thewindow will need some kind of WindowImp before it can do anythinginteresting.

There are several possibilities, but we'll focus on one that uses theAbstract Factory (99) pattern. We can definean abstract factory class WindowSystemFactory that provides aninterface for creating different kinds of window system-dependentimplementation objects:

```
class WindowSystemFactory {
public:
        virtual WindowImp* CreateWindowImp() = 0;
        virtual ColorImp* CreateColorImp() = 0;
        virtual FontImp* CreateFontImp() = 0;

        // a "Create..." operation for all window system resources
};
```

Now we can define a concrete factory for each window system:

```
class PMWindowSystemFactory : public WindowSystemFactory {
        virtual WindowImp* CreateWindowImp()
                { return new PMWindowImp; }
```

```
        // ...
};

class XWindowSystemFactory : public WindowSystemFactory {
        virtual WindowImp* CreateWindowImp()
                { return new XWindowImp; }
        // ...
};
```

The Window base class constructor can use theWindowSystemFactory interface to initialize the_imp member with the WindowImp that's right for the windowsystem:

```
Window::Window () {
        _imp = windowSystemFactory->CreateWindowImp();
}
```

The windowSystemFactory variable is a well-known instance ofa WindowSystemFactory subclass, akin to the well-knownguiFactory variable defining the look and feel. ThewindowSystemFactory variable can be initialized in the sameway.

Bridge Pattern

The WindowImp class defines an interface to common window systemfacilities, but its design is driven by different constraints thanWindow's interface. Application programmers won't deal withWindowImp's interface directly; they only deal with Window objects.So WindowImp's interface needn't match the application programmer'sview of the world, as was our concern in the design of the Windowclass hierarchy and interface. WindowImp's interface can more closelyreflect what window systems actually provide, warts and all. It can bebiased toward either an intersection or a union of functionalityapproach, whichever suits the target window systems best.

The important thing to realize is that Window's interface caters tothe applications programmer, while WindowImp caters to window systems.Separating windowing functionality into Window and WindowImphierarchies lets us implement and specialize these interfacesindependently. Objects from these hierarchies cooperate to letLexi work without modification on multiple window systems.

The relationship between Window and WindowImp is an example of the Bridge (171) pattern. The intent behind Bridge is to allowseparate class hierarchies to work together even as they evolveindependently. Our design criteria led us to create two separate classhierarchies, one that supports the logical notion of windows, andanother for capturing different implementations of windows. The Bridgepattern

lets us maintain and enhance our logical windowingabstractions without touching window system-dependent code, and viceversa.

User Operations

Some of Lexi's functionality is available through the document'sWYSIWYG representation. You enter and delete text, move the insertionpoint, and select ranges of text by pointing, clicking, and typingdirectly in the document. Other functionality is accessed indirectlythrough user operations in Lexi's pull-down menus, buttons, andkeyboard accelerators. The functionality includes operations for

- creating a new document,
- opening, saving, and printing an existing document,
- cutting selected text out of the document and pasting it back in,
- changing the font and style of selected text,
- changing the formatting of text, such as its alignment andjustification,
- quitting the application,
- and on and on.

Lexi provides different user interfaces for these operations.But we don't want to associate a particular user operation with aparticular user interface, because we may want multiple userinterfaces to the same operation (you can turn the page using either apage button or a menu operation, for example). We may also want tochange the interface in the future.

Furthermore, these operations are implemented in many differentclasses. We as implementors want to access their functionalitywithout creating a lot of dependencies between implementation and userinterface classes. Otherwise we'll end up with a tightly coupledimplementation, which will be harder to understand, extend, andmaintain.

To further complicate matters, we want Lexi to support undo andredo[8]ofmost *but not all* its functionality. Specifically, we want to beable to undo document-modifying operations like delete, with which auser can destroy lots of data inadvertently. But we shouldn't try toundo an operation like saving a drawing or quitting the application.These operations should have no effect on the undo process. We alsodon't want an arbitrary limit on the number of levels of undo andredo.

It's clear that support for user operations permeates the application.The challenge is to come up with a simple and extensible mechanismthat satisfies all of these needs.

Encapsulating a Request

From our perspective as designers, a pull-down menu is just anotherkind of glyph that contains other glyphs. What distinguishespull-down menus from other glyphs that have children is that mostglyphs in menus do some work in response to an up-click.

Let's assume that these work-performing glyphs are instances of aGlyph subclass called **MenuItem** and that they do their work inresponse to a request from a client.[9]Carrying out therequest might involve an operation on one object, or many operationson many objects, or something in between.

We could define a subclass of MenuItem for every user operation andthen hard-code each subclass to carry out the request. But that's notreally right; we don't need a subclass of MenuItem for each requestany more than we need a subclass for each text string in a pull-downmenu. Moreover, this approach couples the request to a particularuser interface, making it hard to fulfill the request through adifferent user interface.

To illustrate, suppose you could advance to the last page in thedocument both through a MenuItem in a pull-down menu *and* bypressing a page icon at the bottom of Lexi's interface (which mightbe more convenient for short documents). If we associate the requestwith a MenuItem through inheritance, then we must do the same for thepage icon and any other kind of widget that might issue such arequest. That can give rise to a number of classes approaching theproduct of the number of widget types and the number of requests.

What's missing is a mechanism that lets us parameterize menu items bythe request they should fulfill. That way we avoid a proliferation ofsubclasses and allow for greater flexibility at run-time. We couldparameterize MenuItem with a function to call, but that's not a completesolution for at least three reasons:

1. It doesn't address the undo/redo problem.
2. It's hard to associate state with a function. For example, afunction that changes the font needs to know *which* font.
3. Functions are hard to extend, and it's hard to reuse parts of them.

These reasons suggest that we should parameterize MenuItems with an*object*, not a function. Then we can use inheritance to extendand reuse the request's implementation. We also have a place to storestate and implement undo/redo functionality. Here we have anotherexample of encapsulating the concept that varies, in this case arequest. We'll encapsulate each request in a **command**object.

Command Class and Subclasses

First we define a **Command** abstract class toprovide an interface for issuing a request. The basic interfaceconsists of a single abstract operation called "Execute." Subclassesof Command implement Execute in different ways to fulfill differentrequests. Some subclasses may delegate part or all of the work toother objects. Other subclasses may be in a position to fulfillthe request entirely on their own (see Figure 2.11).To the requester, however, a Command object is a Command object—theyare treated uniformly.

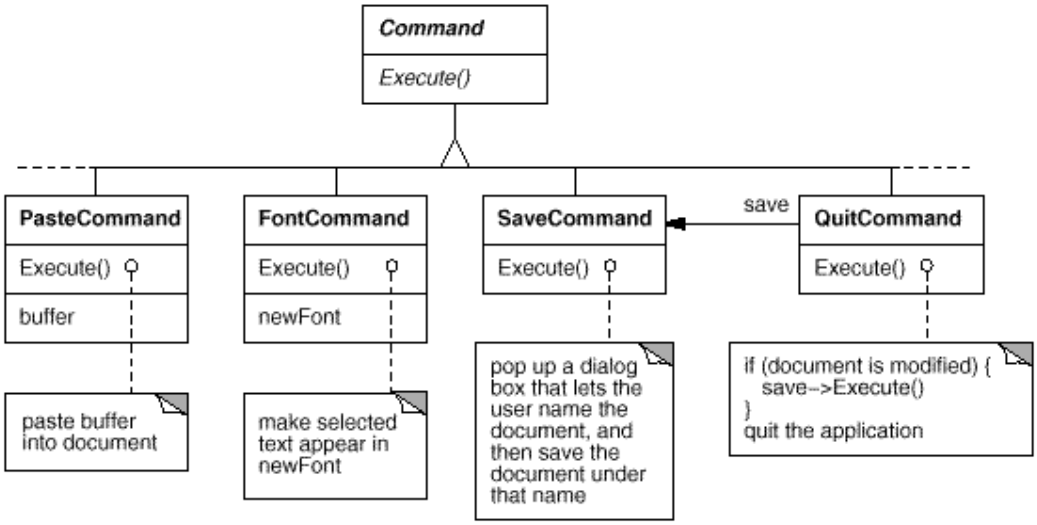


Figure 2.11: Partial Command class hierarchy

Now MenuItem can store a Command object that encapsulates arequest (Figure 2.12). We give each menu item objectan instance of the Command subclass that's suitable for that menuitem, just as we specify the text to appear in the menu item. Whena user chooses a particular menu item, the MenuItem simply callsExecute on its Command object to carry out the request. Note thatbuttons and other widgets can use commands in the same way menuitems do.

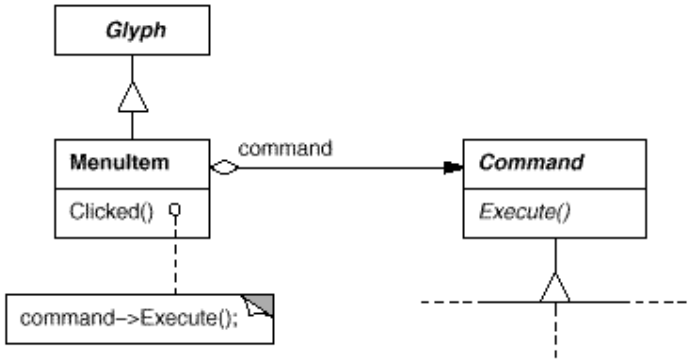


Figure 2.12: MenuItem-Command relationship

Undoability

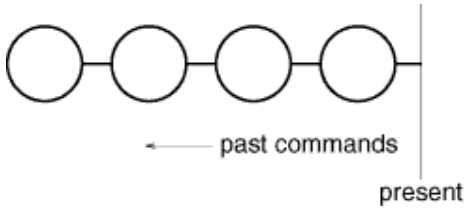
Undo/redo is an important capability in interactive applications. Toundo and redo commands, we add an Unexecute operation to Command'sinterface. Unexecute reverses the effects of a preceding Executeoperation using whatever undo information Execute stored. In the caseof a FontCommand, for example, the Execute operation would store therange of text affected by the font change along with the originalfont(s). FontCommand's Unexecute operation would restore the range oftext to its original font(s).

Sometimes undoability must be determined at run-time. A request tochange the font of a selection does nothing if the text alreadyappears in that font. Suppose the user selects some text and thenrequests a spurious font change. What should be the result of asubsequent undo request? Should a meaningless change cause the undorequest to do something equally meaningless? Probably not. If theuser repeats the spurious font change several times, he shouldn't haveto perform exactly the same number of undo operations to get back tothe last meaningful operation. If the net effect of executing acommand was nothing, then there's no need for a corresponding undorequest.

So to determine if a command is undoable, we add an abstractReversible operation to the Command interface. Reversible returns aBoolean value. Subclasses can redefine this operation to return trueor false based on run-time criteria.

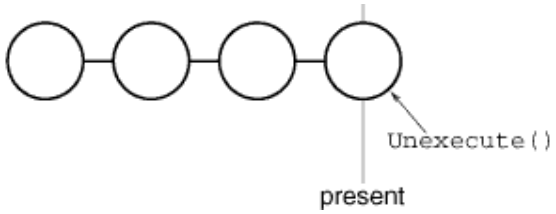
Command History

The final step in supporting arbitrary-level undo and redo is todefine a **command history**, or list of commands that havebeen executed (or unexecuted, if some commands have been undone).Conceptually, the command history looks like this:

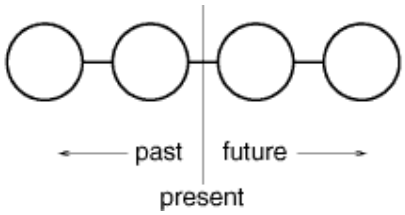


Each circle represents a Command object. In this case the user hasissued four commands. The leftmost command was issued first, followedby the second-leftmost, and so on until the most recently issuedcommand, which is rightmost. The line marked "present" keeps trackof the most recently executed (and unexecuted) command.

To undo the last command, we simply call Unexecute on the most recentcommand:

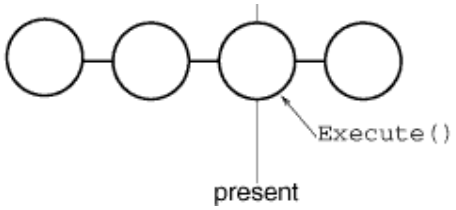


After unexecuting the command, we move the "present" line onecommand to the left. If the user chooses undo again, the next-mostrecently issued command will be undone in the same way, and we're leftin the state depicted here:

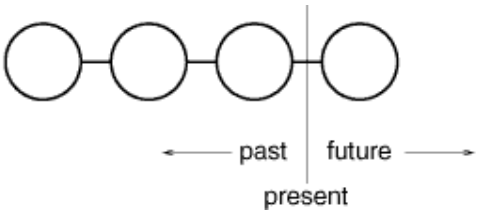


You can see that by simply repeating this procedure we get multiplelevels of undo. The number of levels is limited only by the length ofthe command history.

To redo a command that's just been undone, we do the same thing inreverse. Commands to the right of the present line are commands thatmay be redone in the future. To redo the last undone command, we callExecute on the command to the right of the present line:



Then we advance the present line so that a subsequent redo will callredo on the following command in the future.



Of course, if the subsequent operation is not another redo but an undo,then the command to the left of the present line will be undone. Thusthe user can effectively go back and forth in time as needed torecover from errors.

Command Pattern

Lexi's commands are an application of the Command (263) pattern, which describes how toencapsulate a request. The Command pattern prescribes a uniforminterface for issuing requests that lets you configure clients tohandle different requests. The interface shields clients from therequest's implementation. A command may delegate all, part, or noneof the request's implementation to other objects. This is perfect forapplications like Lexi that must provide centralized access tofunctionality scattered throughout the application. The pattern alsodiscusses undo and redo mechanisms built on the basic Commandinterface.

Spelling Checking and Hyphenation

The last design problem involves textual analysis, specifically checkingfor misspellings and introducing hyphenation points where needed forgood formatting.

The constraints here are similar to those we had for the formattingdesign problem in Section 2.3.As was the case for linebreaking strategies, there's more than oneway to check spelling and compute hyphenation points. So here toowe want to support multiple algorithms. A diverse set of algorithmscan provide a choice of space/time/quality trade-offs. We shouldmake it easy to add new algorithms as well.

We also want to avoid wiring this functionality into the documentstructure. This goal is even more important here than it was in theformatting case, because spelling checking and hyphenation are justtwo of potentially many kinds of analyses we may want Lexi tosupport. Inevitably we'll want to expand Lexi's analyticalabilities over time. We might add searching, word counting, acalculation facility for adding up tabular values, grammar checking,and so forth. But we don't want to change the Glyph class and all itssubclasses every time we introduce new functionality of this sort.

There are actually two pieces to this puzzle: (1) accessing theinformation to be analyzed, which we have scattered over the glyphsin the document structure, and (2) doing the analysis. We'll look atthese two pieces separately.

Accessing Scattered Information

Many kinds of analysis require examining the text character bycharacter. The text we need to analyze is scattered throughout ahierarchical structure of glyph objects. To examine text in such astructure, we need an access mechanism that has knowledge about thedata structures in which objects are stored. Some glyphs might storetheir children in linked lists, others might use arrays, and stillothers might use more esoteric data structures. Our access mechanismmust be able to handle all of these possibilities.

An added complication is that different analyses access information indifferent ways. *Most* analyses will traverse the text frombeginning to end. But some do the opposite—a reverse search, forexample, needs to progress through the text backward rather thanforward. Evaluating algebraic expressions could require an inordertraversal.

So our access mechanism must accommodate differing data structures, andwe must support different kinds of traversals, such as preorder,postorder, and inorder.

Encapsulating Access and Traversal

Right now our glyph interface uses an integer index to let clientsrefer to children. Although that might be reasonable for glyph classesthat store their children in an array, it may be inefficient forglyphs that use a linked list. An important role of the glyphabstraction is to hide the data structure in which children arestored. That way we can change the data structure a glyph class useswithout affecting other classes.

Therefore only the glyph can know the data structure it uses. Acorollary is that the glyph interface shouldn't be biased toward onedata structure or another. It

shouldn't be better suited to arraysthan to linked lists, for example, as it is now.

We can solve this problem and support several different kinds oftraversals at the same time. We can put multiple access and traversalcapabilities directly in the glyph classes and provide a way to chooseamong them, perhaps by supplying an enumerated constant as aparameter. The classes pass this parameter around during a traversalto ensure they're all doing the same kind of traversal. They have topass around any information they've accumulated during traversal.

We might add the following abstract operations to Glyph's interface tosupport this approach:

```
void First(Traversal kind)
void Next()
bool IsDone()
Glyph* GetCurrent()
void Insert(Glyph*)
```

Operations First, Next, and IsDonecontrol the traversal. First initializes the traversal. Ittakes the kind of traversal as a parameter of typeTraversal, an enumerated constant with valuessuch as CHILDREN (to traverse the glyph's immediate childrenonly), PREORDER (to traverse the entire structure inpreorder), POSTORDER, and INORDER.Next advances to the next glyph in the traversal, andIsDone reports whether the traversal is over or not.GetCurrent replaces theChild operation; it accesses the current glyph in thetraversal. Insert replaces the old operation; it insertsthe given glyph at the current position.An analysis would use the following C++ code to do a preordertraversal of a glyph structure rooted at g:

```
Glyph* g;

for (g->First(PREORDER); !g->IsDone(); g->Next()) {
        Glyph* current = g->GetCurrent();

        // do some analysis
}
```

Notice that we've banished the integer index from the glyph interface.There's no longer anything that biases the interface toward one kindof collection or another. We've also saved clients from having toimplement common kinds of traversals themselves.

But this approach still has problems. For one thing, it can't supportnew traversals without either extending the set of enumerated valuesor adding new operations. Say we wanted to have a variation on preordertraversal that automatically skips

non-textual glyphs. We'd have tochange the Traversal enumeration to include something likeTEXTUAL_PREORDER.

We'd like to avoid changing existing declarations.Putting the traversal mechanism entirely in the Glyph class hierarchymakes it hard to modify or extend without changing lots of classes.It's also difficult to reuse the mechanism to traverse other kinds ofobject structures. And we can't have more than one traversal inprogress on a structure.

Once again, a better solution is to encapsulate the concept thatvaries, in this case the access and traversal mechanisms. We canintroduce a class of objects called **iterators** whose solepurpose is to define different sets of these mechanisms. We can useinheritance to let us access different data structures uniformly andsupport new kinds of traversals as well. And we won't have to changeglyph interfaces or disturb existing glyph implementations to do it.

Iterator Class and Subclasses

We'll use an abstract class called **Iterator** todefine a general interface for access and traversal. Concretesubclasses like **ArrayIterator** and**ListIterator** implement the interface to provideaccess to arrays and lists, while **PreorderIterator**,**PostorderIterator**, and the like implement differenttraversals on specific structures. Each Iterator subclass has areference to the structure it traverses. Subclass instances areinitialized with this reference when they are created. Figure 2.13 illustrates theIterator class along with several subclasses. Notice that we'veadded a CreateIterator abstract operation to the Glyph classinterface to support iterators.

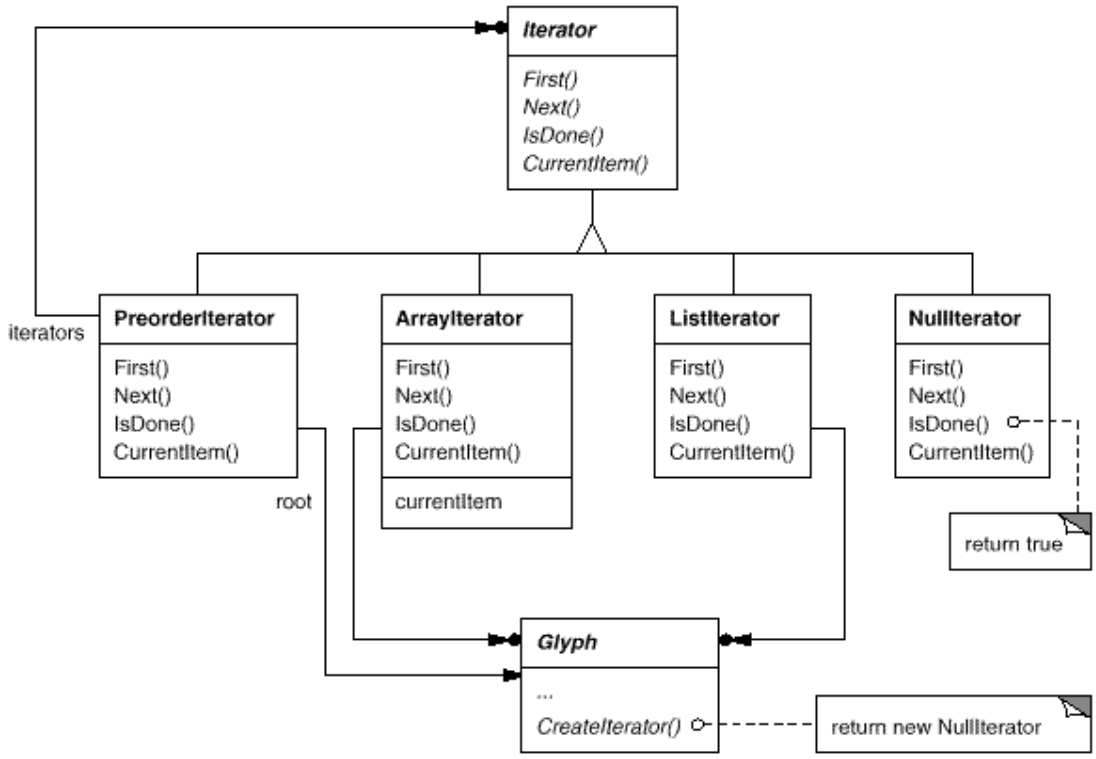


Figure 2.13: Iterator class and subclasses

The Iterator interface provides operations First, Next, and IsDone forcontrolling the traversal. The ListIterator class implements First topoint to the first element in the list, and Next advances the iteratorto the next item in the list. IsDone returns whether or not the listpointer points beyond the last element in the list. CurrentItemdereferences the iterator to return the glyph it points to. AnArrayIterator class would do similar things but on anarray of glyphs.

Now we can access the children of a glyph structure without knowingits representation:

```
Glyph* g;
Iterator<Glyph*>* i = g->CreateIterator();

for (i->First(); !i->IsDone(); i->Next()) {
        Glyph* child = i->CurrentItem();

        // do something with current child
}
```

CreateIterator returns a NullIterator instance by default. ANullIterator is a degenerate iterator for glyphs that have nochildren, that is, leaf glyphs. NullIterator's IsDone operationalways returns true.

A glyph subclass that has children will override CreateIterator toreturn an instance of a different Iterator subclass. *Which*subclass depends on the structure that stores the children. If theRow subclass of Glyph stores its children in a list_children, then its CreateIterator operation would looklike this:

```
Iterator<Glyph*>* Row::CreateIterator () {
          return new ListIterator<Glyph*>(_children);
}
```

Iterators for preorder and inorder traversals implement theirtraversals in terms of glyph-specific iterators. The iterators forthese traversals are supplied the root glyph in the structure theytraverse. They call CreateIterator on the glyphs in the structure anduse a stack to keep track of the resulting iterators.

For example, class PreorderIterator gets the iterator fromthe root glyph, initializes it to point to its first element, and thenpushes it onto the stack:

```
void PreorderIterator::First () {
          Iterator<Glyph*>* i = _root->CreateIterator();
          if (i) {
                    i->First();
                    _iterators.RemoveAll();
                    _iterators.Push(i);
          }
}
```

CurrentItem would simply call CurrentItem on theiterator at the top of the stack:

```
Glyph* PreorderIterator::CurrentItem () const {
          Return _iterators.Size() > 0 ? _iterators.Top()->CurrentItem() : 0;
}
```

The Next operation gets the top iterator on the stack andasks its current item to create an iterator, in an effort to descendthe glyph structure as far as possible (this is a preordertraversal, after all). Next sets the new iterator to thefirst item in the traversal and pushes it on the stack. ThenNext tests the latest iterator; if its IsDoneoperation returns true, then we've finished traversing the currentsubtree (or leaf) in the traversal. In that case, Next popsthe top iterator off the stack and repeats this process until it findsthe next incomplete traversal, if there is one; if not, then we havefinished traversing the structure.

```
void PreorderIterator::Next () {
Iterator<Glyph*>* i = _iterators.Top()->CurrentItem()->CreateIterator();

i->First();
_iterators.Push(i);

while ( _iterators.Size() > 0 && _iterators.Top()->IsDone() ) {
          delete _iterators.Pop();
          _iterators.Top()->Next();
}
}
```

Notice how the Iterator class hierarchy lets us add new kinds oftraversals without modifying glyph classes—we simply subclassIterator and add a new traversal as we have withPreorderIterator. Glyph subclasses use the sameinterface to give clients access to their children without revealingthe underlying data structure they use to store them. Becauseiterators store their own copy of the state of a traversal, we cancarry on multiple traversals simultaneously, even on the samestructure. And though our traversals have been over glyph structuresin this example, there's no reason we can't parameterize a class likePreorderIterator by the type of object in the structure.We'd use templates to do that in C++. Then we can reuse the machineryin PreorderIterator to traverse other structures.

Iterator Pattern

The Iterator (289) pattern captures these techniquesfor supporting access and traversal over object structures. It'sapplicable not only to composite structures but to collections aswell. It abstracts the traversal algorithm and shields clients fromthe internal structure of the objects they traverse. The Iteratorpattern illustrates once more how encapsulating the concept thatvaries helps us gain flexibility and reusability. Even so, theproblem of iteration has surprising depth, and the Iterator patterncovers many more nuances and trade-offs than we've considered here.

Traversal versus Traversal Actions

Now that we have a way of traversing the glyph structure, we need tocheck the spelling and do the hyphenation. Both analyses involveaccumulating information during the traversal.

First we have to decide where to put the responsibility for analysis.We could put it in the Iterator classes, thereby making analysis anintegral part of traversal. But we get more flexibility and potentialfor reuse if we distinguish between the traversal and the actionsperformed during traversal. That's because

different analyses oftenrequire the same kind of traversal. Hence we can reuse the same setof iterators for different analyses. For example, preorder traversalis common to many analyses, including spelling checking, hyphenation,forward search, and word count.

So analysis and traversal should be separate. Where else can we putthe responsibility for analysis? We know there are many kinds ofanalyses we might want to do. Each analysis will do different thingsat different points in the traversal. Some glyphs are moresignificant than others depending on the kind of analysis. If we'rechecking spelling or hyphenating, we want to consider character glyphsand not graphical ones like lines and bitmapped images. If we'remaking color separations, we'd want to consider visible glyphs and notinvisible ones. Inevitably, different analyses will analyze differentglyphs.

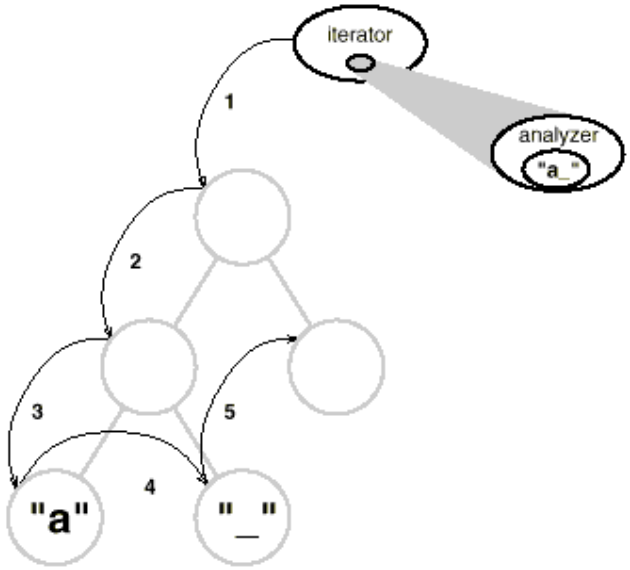
Therefore a given analysis must be able to distinguish different kinds ofglyphs. An obvious approach is to put the analytical capability into theglyph classes themselves. For each analysis we can add one or moreabstract operations to the Glyph class and have subclasses implementthem in accordance with the role they play in the analysis.

But the trouble with that approach is that we'll have to change everyglyph class whenever we add a new kind of analysis. We can ease thisproblem in some cases: If only a few classes participate in theanalysis, or if most classes do the analysis the same way, then we can supplya default implementation for the abstract operation in the Glyphclass. The default operation would cover the common case. Thus we'dlimit changes to just the Glyph class and those subclasses that deviatefrom the norm.

Yet even if a default implementation reduces the number of changes, aninsidious problem remains: Glyph's interface expands with every newanalytical capability. Over time the analytical operations will startto obscure the basic Glyph interface. It becomes hard to see that aglyph's main purpose is to define and structure objects that haveappearance and shape—that interface gets lost in the noise.

Encapsulating the Analysis

From all indications, we need to encapsulate the analysis in aseparate object, much like we've done many times before. We could putthe machinery for a given analysis into its own class. We could usean instance of this class in conjunction with an appropriate iterator.The iterator would "carry" the instance to each glyph in thestructure. The analysis object could then perform a piece of theanalysis at each point in the traversal. The analyzer accumulatesinformation of interest (characters in this case) as the traversalproceeds:



The fundamental question with this approach is how the analysis objectdistinguishes different kinds of glyphs without resorting to typetests or downcasts. We don't want a SpellingChecker classto include (pseudo)code like

```
void SpellingChecker::Check (Glyph* glyph) {
Character* c;
Row* r;
Image* i;

if (c = dynamic_cast<Character*>(glyph)) {
          // analyze the character
} else if (r = dynamic_cast<Row*>(glyph)) {
          // prepare to analyze r's children
} else if (i = dynamic_cast<Image*>(glyph)) {
          // do nothing
}
}
```

This code is pretty ugly. It relies on fairly esoteric capabilitieslike type-safe casts. It's hard to extend as well. We'll have toremember to change the body of this function whenever we change theGlyph class hierarchy. In fact, this is the kind of code thatobject-oriented languages were intended to eliminate.

We want to avoid such a brute-force approach, but how? Let's considerwhat happens when we add the following abstract operation to the Glyphclass:

```
   void CheckMe(SpellingChecker&)
```

We define CheckMe in every Glyph subclass as follows:

```
   void GlyphSubclass::CheckMe (SpellingChecker& checker)
{       checker.CheckGlyphSubclass(this);    }
```

where GlyphSubclass would be replaced by the name of theglyph subclass. Note that when CheckMe is called, thespecific Glyph subclass is known—after all, we're in one of itsoperations. In turn, theSpellingChecker class interface includes an operation likeCheckGlyphSubclass for every Glyphsubclass[10]:

```
class SpellingChecker {
public:
SpellingChecker();
virtual void CheckCharacter(Character*);
virtual void CheckRow(Row*);
virtual void CheckImage(Image*);
// ... and so forth
List<char*>& GetMisspellings();

protected:
virtual bool IsMisspelled(const char*);

private:
char _currentWord[MAX_WORD_SIZE];
List<char*> _misspellings;
};
```

SpellingChecker's checking operation forCharacter glyphs might look something like this:

```
void SpellingChecker::CheckCharacter (Character* c) {
const char ch = c->GetCharCode();
if (isalpha(ch)) {
          // append alphabetic character to _currentWord
} else {
          // we hit a nonalphabetic character
          if (IsMisspelled(_currentWord)) {
          // add _currentWord to _misspellings
          _misspellings.Append(strdup(_currentWord));
}
_currentWord[0] = '\0';
// reset _currentWord to check next word
}
}
```

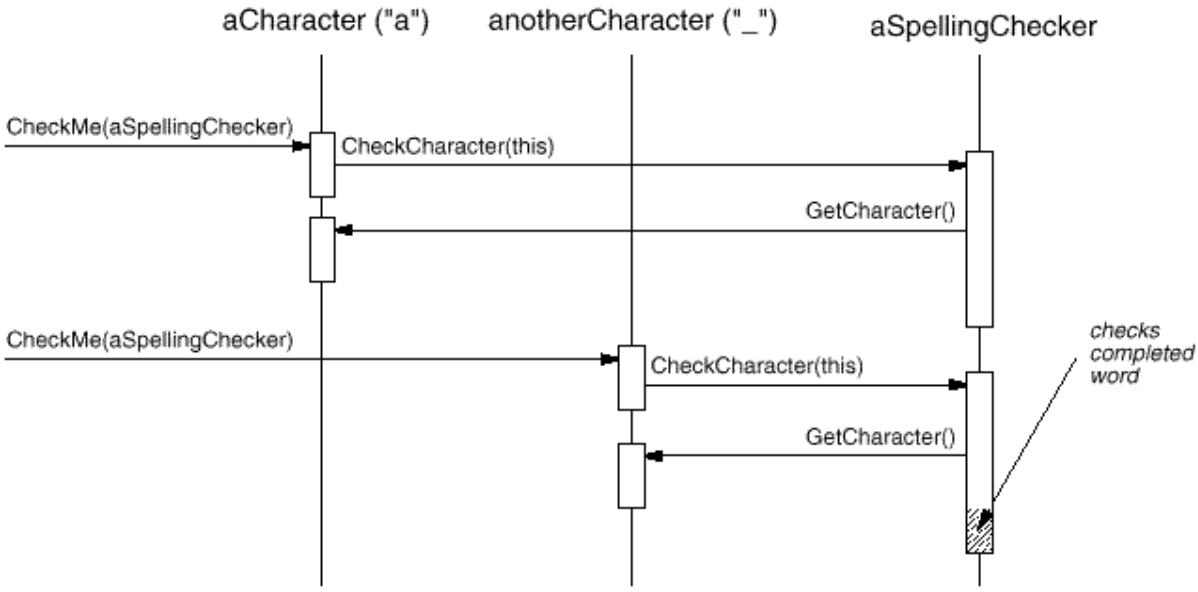
Notice we've defined a special GetCharCode operation onjust the Character class. The spelling checker can deal withsubclass-specific operations without resorting to type tests orcasts—it lets us treat objects specially.

CheckCharacter accumulates alphabetic charactersinto the _currentWord buffer. When it encounters anonalphabetic character, such as an underscore, it uses theIsMisspelled operation to check the spelling of theword in _currentWord.[11] If the word ismisspelled, then CheckCharacter adds the word to thelist of misspelled words. Then it must clear out the_currentWord buffer to ready it for the next word.When the traversal is over, you can retrieve the list of misspelledwords with the GetMisspellings operation.

Now we can traverse the glyph structure, callingCheckMe on each glyph with the spelling checker as an argument.This effectively identifies each glyph to the SpellingChecker andprompts the checker to do the next increment in the spelling check.

```
SpellingChecker spellingChecker;
Composition* c;
// ...
Glyph* g;
PreorderIterator i(c);
for (i.First(); !i.IsDone(); i.Next()) {
          g = i.CurrentItem();
          g->CheckMe(spellingChecker);
}
```

The following interaction diagram illustrates howCharacter glyphs and the SpellingChecker objectwork together:



This approach works for finding spelling errors, but how does it helpus support multiple kinds of analysis? It looks like we have to addan operation like CheckMe(SpellingChecker&) to Glyph andits subclasses whenever we add a new kind of analysis. That's true ifwe insist on an *independent* class for every analysis. Butthere's no reason why we can't give *all* analysis classes thesame interface. Doing so lets us use them polymorphically. Thatmeans we can replace analysis-specific operations likeCheckMe(SpellingChecker&) with an analysis-independentoperation that takes a more general parameter.

Visitor Class and Subclasses

We'll use the term **visitor** to refer generally to classesof objects that "visit" other objects during a traversal and dosomething appropriate.[12]In this case we can define aVisitor class that defines an abstract interface forvisiting glyphs in a structure.

```
class Visitor {
public:
virtual void VisitCharacter(Character*) { }
virtual void VisitRow(Row*) { }
virtual void VisitImage(Image*) { }

// ... and so forth
};
```

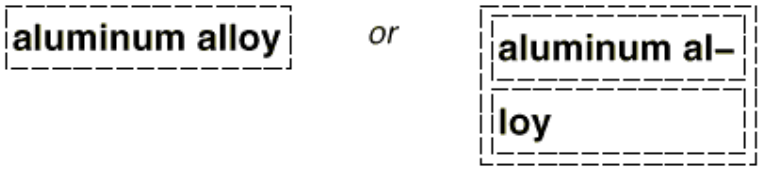
Concrete subclasses of Visitor perform different analyses.For example, we could have a SpellingCheckingVisitorsubclass for checking spelling, and a HyphenationVisitorsubclass for hyphenation. SpellingCheckingVisitor wouldbe implemented exactly as we implemented SpellingCheckerabove, except the operation names would reflect the more generalVisitor interface. For example,CheckCharacter would be called VisitCharacter.

Since CheckMe isn't appropriate for visitors that don'tcheck anything, we'll give it a more general name:

Accept. Its argument must also change to take aVisitor&, reflecting the fact that it can accept any visitor.Now adding a new analysis requires just defining a new subclass ofVisitor—we don't have to touch any of the glyph classes.We support all future analyses by adding this one operationto Glyph and its subclasses.

We've already seen how spelling checking works. We use a similarapproach in HyphenationVisitor to accumulate text. Butonce HyphenationVisitor's VisitCharacter operationhas assembled an entire word, it works a little differently. Insteadof checking the word for misspelling, it applies a hyphenationalgorithm to determine the potential hyphenation points in the word,if any. Then at each hyphenation point, it inserts a **discretionary** glyph into the composition. Discretionaryglyphs are instances of Discretionary, a subclass ofGlyph.

A discretionary glyph has one of two possible appearances depending onwhether or not it is the last character on a line. If it's the lastcharacter, then the discretionary looks like a hyphen; if it's not atthe end of a line, then the discretionary has no appearancewhatsoever. The discretionary checks its parent (a Row object) to seeif it is the last child. The discretionary makes this check wheneverit's called on to draw itself or calculate its boundaries. Theformatting strategy treats discretionaries the same as whitespace,making them candidates for ending a line. The following diagram shows howan embedded discretionary can appear.



Visitor Pattern

What we've described here is an application of the Visitor (366) pattern. The Visitor class and itssubclasses described earlier are the key participants in the pattern.The Visitor pattern captures the technique we've used to allow anopen-ended number of analyses of glyph structures without having tochange the glyph classes themselves. Another nice feature of visitorsis that they can be applied not just to composites like our glyphstructures but to *any* object structure. That includes sets,lists, even directed-acyclic graphs. Furthermore, the classes that avisitor can visit needn't be related to each other through a commonparent class. That means visitors can work across class hierarchies.

An important question to ask yourself before applying the Visitorpattern is, Which class hierarchies change most often? The pattern ismost suitable when you want to be able to do a variety of differentthings to objects that have a stable class structure. Adding a newkind of visitor requires no change to that class structure, which isespecially important when the class structure is large. But whenever youadd a subclass to the structure, you'll also have to update all yourvisitor interfaces to include a Visit... operation for thatsubclass. In our example that means adding a newGlyph subclass called Foo will require changingVisitor and all its subclasses to include aVisitFoo operation. But given our design constraints, we'remuch more likely to add a new kind of analysis to Lexi than a newkind of Glyph. So the Visitor pattern is well-suited to our needs.

Summary

We've applied eight different patterns to Lexi's design:

1. Composite (183)to represent the document's physical structure,
2. Strategy (349) to allow differentformatting algorithms,
3. Decorator (196) for embellishingthe user interface,
4. Abstract Factory (99) forsupporting multiple look-and-feel standards,